Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids

Daniel Paranhos da Silva, Walfredo Cirne, and Francisco Vilar Brasileiro

Universidade Federal de Campina Grande, Departamento de Sistemas e Computação, Av. Aprígio Veloso s/n, Bodocongó, 58.109-970. Campina Grande, PB, Brazil.
{danielps,walfredo,fubica}@dsc.ufcg.edu.br

Abstract. Scheduling independent tasks on heterogeneous environments, like grids, is not trivial. To make a good scheduling plan on this kind of environments, the scheduler usually needs some information such as host speed, host load, and task size. This kind of information is not always available and is often difficult to obtain. In this paper we propose a scheduling approach that does not use any kind of information but still delivers good performance. Our approach uses task replication to cope with the dynamic and heterogeneous nature of grids without depending on any information about machines or tasks. Our results show that task replication can deliver good and stable performance at the expense of additional resource consumption. By limiting replication, however, additional resource consumption can be controlled with little effect on performance.

1 Introduction

Recent years have seen increased availability of powerful computers and high-speed networks. This fact has made it possible to aggregate geographically dispersed resources for the execution of large-scale resource-intensive applications. This aggregation of resources has been called a *Computational Grid*. Grids may be composed by a wide variety of computers, visualization devices, storage systems and databases, scientific instruments, all connected through combinations of local and wide area networks. Furthermore, multiple users can simultaneously use these resources to execute a variety of large parallel applications. This characterizes grids as being heterogeneous and very dynamic.

Due to the wide distribution, heterogeneity and dynamicity of grids, loosely coupled parallel applications are better suited for execution on grids than tightly coupled applications. In particular, one can argue that Bag-of-Tasks (BoT) applications (i.e. applications whose tasks are completely independent) are the most suitable kind of application for current grid environments. Moreover, there are many important BoT applications, including data mining, massive searches (such as key breaking), parameter sweeps, Monte Carlo simulations, fractals calculations (such as Mandelbrot),

H. Kosch, L. Böszörményi, H. Hellwagner (Eds.): Euro-Par 2003, LNCS 2790, pp. 169–180, 2003.
© Springer-Verlag Berlin Heidelberg 2003

and image manipulation applications (such as tomographic reconstruction). In short, BoT applications are a class of relevant applications that are well suited for execution on grids.

However, scheduling BoT applications on grids is still an open problem. Good scheduling requires good information about the grid resources, which is often difficult to obtain. Known knowledge-free scheduling algorithms usually have worse performance than algorithms that have full knowledge about the environment.

We developed the *Workqueue with Replication* (WQR) algorithm to solve this problem. WQR delivers good performance without using any kind of information about the resources or tasks. It basically adds task replication to the classic *Workqueue* algorithm to cope with the dynamic and heterogeneous nature of *Computational Grids*. When a task is replicated, the first replica that finishes is considered as the valid execution of the task and the other replicas are cancelled. With this approach we minimize the effects of the dynamic machine load, machine heterogeneity and task heterogeneity, and do so without relying on information on machines or tasks. A way to think about WQR is that it allows us to trade some additional CPU cycles for the need of information about the resources on the grid.

The performance of WQR is similar or even better than solutions that have full knowledge about the environment (which is not feasible to obtain in practice), at the expense of consuming more cycles. In some scenarios, the additional cycles consumed by WQR are negligible. In the scenarios this is not the case, the extra cycles from replication can be controlled by limiting replication, a solution that shows little impact on the performance attained by WQR. Moreover, BoT applications often use cycles that would otherwise go idle cycles [7]. Thus, trading CPU cycles for the need of information can be advantageous in practice.

2 Scheduling *Bag-of-Tasks* (BoT) Applications on Grids

Scheduling applications on such a heterogeneous and dynamic environment like a *Computational Grid* is not a trivial task. Some characteristics that are intrinsic to grids should be considered during the scheduling, such as resource heterogeneity, the dynamic nature of the machine and network loads, the bandwidth, latency and topology of the network.

Of course, how difficult scheduling is, depends on the characteristics of the target application. Even scheduling BoT applications, which one could think is easy due to its simplicity, is not a trivial task. Scheduling BoT applications on grids is difficult due to the dynamic behavior and the intrinsic resource heterogeneity exhibited by most grids. Grid environments are typically composed by shared resources and thus the contention created by other applications running simultaneously on these resources causes delays and degrades the quality of service. Also, resources on grids are heterogeneous and may not perform the same way for all applications. Achieving good performance in this situation usually requires the use of good information to make the scheduling plan.

Unfortunately, due to grid's very wide distribution, it is usually difficult to obtain good information about the entire grid. There were some efforts to instrument the grid

to obtain dynamic information such as host load and network latency and bandwidth [5] [8] [14]. Some initial results are encouraging [8] [14], but making such monitoring scale to full grid size involves a number of obstacles [5]. Moreover, sometimes it is not possible to install monitoring systems at the user's will. Sites often place administrative restrictions on what can be ran and network traffic is filtered by firewalls. Finally, monitoring per se is not enough. In order to assure good scheduling, performance prediction based on monitored values is often desirable, what further complicates the issue.

Moreover, it is also difficult to obtain information about the tasks (execution time) to elaborate the scheduling plan. Sometimes the only way to obtain it is to run the task on a given processor. Of course this strategy only works in cases that all tasks have the same complexity or are very similar. A task can be executed in all processors and use its execution time on them to predict the execution time of the remaining tasks.

Despite the difficulties in obtaining good information about both grid resources and tasks, most efforts in scheduling applications composed by independent tasks assume good information is available (e.g. [1] [6] [9] [10]).

We developed a dynamic algorithm (WQR) for scheduling BoT applications on grid environments. This solution does not need any kind of information. WQR will be detailed in Section 2.4. To compare the performance of our solution we considered only dynamic algorithms because of their better suitability on grids. The algorithms that we chose to be compared are: *Dynamic Fastest Processor to Largest Task First* (*Dynamic FPLTF*), *Sufferage* and *Workqueue*. These algorithms were chosen because they are well known and studied algorithms. *FPLTF* is a static scheduler that presents good performance on dedicated environments [10]. *Dynamic FPLTF* is the result of a modification we made on static *FPLTF* to make it adaptive and hence usable on grid environments. *Sufferage* has been shown to attain good performance on grids [1]. *Workqueue* was obviously chosen because our solution extends it.

2.1 Dynamic FPLTF

A good representative of static schedulers for Bag-of-Tasks applications is *Fastest Processor to Largest Task First* (*FPLTF*) [10]. However, the dynamicity and heterogeneity of resources present on grids make static schedulers not a good solution to be used on grids. To cope with this problem we changed *FPLTF*, making it dynamic. *Dynamic FPLTF* has a good ability to adapt to the dynamicity and heterogeneity of the environment. *Dynamic FPLTF* needs three types of information to schedule the tasks properly: `Task Size`, `Host Load` and `Host Speed`. `Host Speed` represents the speed of the host and its value is relative. A machine that has `Host Speed = 2` executes a task twice faster than a machine with `Host Speed = 1`. `Host Load` represents the fraction of the machine that is *not* available to the application. Finally, `Task Size` is the time necessary for a machine with `Host Speed = 1` to complete a task when `Host Load = 0`. In the beginning of the algorithm, the Time to Become Available (TBA) of each host is initialized with 0 and the tasks are sorted by size in a descending way. Therefore, the largest task is allocated first. A task is allocated to the

host that provides the better completion time `CT (CT = TBA + Task Cost`, where `Task Cost = ( Task Size / Host Speed ) / ( 1 - Host Load))`.

When a task is allocated to a host, the `TBA` value corresponding to this host is incremented by `Task Cost`. Tasks are allocated until all machines of the grid are used. After that, the execution of the application begins. When a task finishes, all tasks that are not running are unscheduled and rescheduled again until all machines become used. This scheme continues until all tasks are completed. This strategy tries to minimize the effects of the dynamicity of the grid. A problem would arise if a lightly loaded machine becomes heavily loaded. This load variation would compromise the whole application since the tasks scheduled to this machine would execute much slower. The task rescheduling process corrects this problem by allocating larger tasks prioritizing the currently faster machines. This scheme leads to a good performance, as we shall see in Section 3.2, but requires too much information about the environment (`Task Size, Host Load` and `Host Speed`), making it hard to deploy in practice.

2.2 Sufferage

The idea behind *Sufferage* [1] is that a machine is assigned to a task that would "suffer" the most if that machine would not be assigned to it. The sufferage value of a task is defined by the difference between its second best and its best completion time (`CT`). The sufferage value of each task in the "bag of tasks" is calculated and a task is possibly assigned to the machine that gives its best completion time. If another task was previously assigned to the machine, the sufferage values of the task previously assigned and of the new task are compared. The task that stays in the host is the one that has the greater sufferage value, the other one returns to the "bag of tasks" to be assigned later.

Of course, the sufferage values of each task vary during the application execution due to the load dynamicity intrinsic in grid environments. To cope with that, we invoke *Sufferage* many times during the execution of an application. Every time a task finishes, all tasks that have not started yet are unscheduled and the algorithm is invoked again, using the current sufferage values. The algorithm runs again to schedule the remaining tasks, but this time with the updated load of the machines. This scheme is repeated until all tasks are completed. The problem of this algorithm is the same as *Dynamic FPLTF*; it needs too much information to calculate the completion times of the tasks.

2.3 Workqueue

Workqueue is a knowledge-free scheduler in the sense that it does not need any kind of information for task scheduling. Tasks are chosen in an arbitrary order in the "bag of tasks" and sent to the processors, as soon as they become available. After the completion of a task, the processor sends back the results and the scheduler assigns a new task to the processor. That is, the scheduler starts by sending a task to every available host. Once a host finishes its task, the scheduler assigns another task to the host.

The idea behind this approach is that more tasks will be assigned to the fast/idle machines while the slow/busy machines will process a small load. The great advantage here is that *Workqueue* does not depend on performance information. Unfortunately, *Workqueue* does not attain performance comparable to schedulers based on full knowledge about the environment, as we shall see in Section 3.2. The problem with *Workqueue* arises when a large task is allocated to a slow machine towards the end of the schedule. When this occurs, the completion of the application will be delayed until the complete execution of this task.

2.4 Workqueue with Replication (WQR)

Due to the difficulty on consistently obtaining good performance information about grid machines and application tasks, we have decided to implement a scheduling algorithm that is not based on performance information. We call our solution *Workqueue with Replication* (WQR), since it basically adds task replication to the *Workqueue* algorithm. Its performance is equivalent to solutions that have full knowledge about the environment (which are not feasible in practice), at the expense of consuming more cycles.

The WQR algorithm uses task replication to cope with the heterogeneity of hosts and tasks, and also with the dynamic variation of resource availability due to contention created by others users. The beginning of the algorithm execution is the same as *Workqueue* and continues the same until "the bag of tasks becomes empty". At this time, in *Workqueue*, hosts that finish their tasks would become idle during the rest of the application execution. Using the replication approach, these hosts are assigned to execute replicas of tasks that are still running. Tasks are replicated until a predefined maximum number of replicas is achieved. When a task finishes, its replicas are cancelled. With this approach, we increase the performance on situations that tasks are delaying the complete execution because they were assigned to slow/busy hosts. When a task is replicated, there is a greater chance that a replica is assigned to a fast/idle host.

Note that the replication assumes that the tasks do not cause collateral effects (e.g. Database accesses by independent tasks can generate inconsistency). This assumption is reasonable when talking about grid environments because it is not common to have this kind of application on environments like that due to their widely distribution.

The good point of our solution is that it does not use information about hosts (speed, load) or task sizes. The negative point is that to achieve good performance it wastes CPU cycles with task replicas that are cancelled. That is, the CPU used was not useful for application processing. As we shall see, however, when the application granularity is not high, WQR wastes few cycles compared to the cycles need to execute the application (up to 40% more). Alas, with high application granularity the extra resource consumption of WQR can be significant (up to 105%, in our experiments). In these cases, replication can be limited, keeping cycle's wasting below 50% and still attaining good performance.

In summary, our approach tries to minimize the problem of scheduling independent tasks in grid environments, while avoiding dependence on any kind of information like

task size and processor performance. This kind of information is helpful to make a good scheduling plan but it can be difficult to obtain due to the distributed nature of grids. Further, when it is possible to obtain this information, the predictions for computing and networking resources are not always accurate and reliable.

3 Performance Evaluation

The main objective of our experiments is to evaluate the performance of different scheduling algorithms. The experiments help to evaluate the influence of the grid machines heterogeneity (different speeds), the application tasks heterogeneity (size variation) and the granularity of application tasks (number of tasks per machine).

We performed approximately 8,000 experiments to compare the performance of different scheduling heuristics for BoT applications. Three other algorithms were evaluated together with the *Workqueue with Replication* approach: *Workqueue*, *Dynamic Fastest Processor to Largest Task First* (*Dynamic FPLTF*) and *Sufferage*. Each experiment consists of six simulations. All algorithms (*Workqueue*, *Dynamic FPLTF*, *Sufferage*, WQR 2x, WQR 3x and WQR 4x) are simulated with the same set of machines and tasks. 2x, 3x and 4x denote the maximum amount of replication WQR is allowed to do.

3.1 Simulation Environment

To run our experiments, we used the Simgrid toolkit [13]. This toolkit provides basic functions for the simulation of distributed applications in grid environments. In our work, we make the assumption that the network transfer times are negligible because we are dealing with applications that have small input/output data. This means either that the tasks are CPU bound or that large amounts of data have been previously staged in the grid, as it is common practice for data grid applications [4] [12]. Another point that has to be emphasized about our experiments is that the information-based algorithms (*Dynamic FPLTF* and *Sufferage*) are fed with perfect data about machines and tasks, something that is almost impossible in the real world.

All experiments have a fixed value for grid power (sum of processor capacities of all machines) and application size. The fixed value for the grid power is 1000. That, for example, could be 1000 machines with power 1. Machine power represents how fast the hosts can execute tasks. A host with power 2 can execute a "10 seconds task" in 5 seconds. The fixed value for the application size is 3600000 seconds. In a perfect world (where all machines are 100% free and all tasks finish simultaneously), this application could be completed in exactly 1 hour (3600 seconds) within a grid whose power is 1000. Note that, by fixing grid power and application size, differences in application completion time can be credited solely to the scheduler.

In order to simulate the heterogeneity of the machines in approximately five years accordingly to the Moore's Law [3], the speed of the machines is taken from the uniform distribution $U(10-(hm/2), 10+(hm/2))$. The possible values for *hm* are 1, 2, 4, 8 and 16. The major purpose of this distribution is to keep the average speed of all machines in approximately 10. Thus, the scheduler is made the responsible for the appli-

cation performance because all grids used in the experiments have roughly the same number of machines. In particular, *hm = 1* means that the grid is homogeneous (all machines have speed 10). When *hm* is equal to 2, the speed of machines varies accordingly to the uniform distribution *U(9, 11)* (difference of 2 units, keeping the average speed in 10) and so on. Machines are added to the grid accordingly to each distribution until the sum of their speeds reaches the grid power which is always 1000 (roughly 100 machines).

The load on each host is simulated by traces obtained from NWS [14] measurements on actual systems. These traces contain the percentage of free CPU as a function of time. The simulator uses these traces to make machines, in the experiment, behave similarly to real machines.

The experiments were defined in such a way that it could create 20 different types of application to be evaluated. The types of applications are divided in four major groups, where the application granularity varies. The mean task size of these groups are 1000, 5000, 25000 and 125000 seconds, following an exponential scale.

To simulate the heterogeneity of tasks in each on of the four major groups, the size of tasks varies, but the mean task size remains the same within the group. The variation on task sizes can be of 0%, 25%, 50%, 75% and 100% relative to the mean tasks size of the group. The group of applications whose mean task size is 5000 seconds, for example, is subdivided in five groups. In one group, all tasks have the same size (5000 seconds, variation 0%), homogeneous tasks. In the group where the variation is 25%, the time of execution of each task varies accordingly to the uniform distribution *U(4375, 5625)* (from 12,5% less than 5000 to 12,5% more than 5000 seconds). For the group which the variation is 50%, the task sizes follow the uniform distribution *U(3750, 6250)* (from 25% less than 5000 to 25% more than 5000 seconds) and so on. Tasks are added to the application accordingly to each distribution until the sum of their sizes reaches the application size which is always 3,600,000 seconds.

With the experiments defined this way, we could evaluate not only the influence of grid resources heterogeneity and tasks heterogeneity, but also the granularity of the application tasks. Varying the mean sizes of tasks (1000, 5000, 25000 and 125000 seconds), the number of tasks of the applications is also changing as can be seen in Table 1.

Table 1. Granularity of Application Tasks.

Mean Sizes of Tasks (seconds)	# of Tasks	Tasks per Machine
1000	3.600	36
5000	720	7,2
25000	144	1,44
125000	29	0,29

The main objective of this variation on the tasks per machine relation is to evaluate the performance impact on each scheduler when there are much more tasks than ma-

chines (3600 tasks/100 machines) and gradually reduce this relation to a situation where there is more machines than tasks (29 tasks/100 machines).

3.2 Result Analysis

Figure 1A shows the mean execution time for each scheduling algorithm working on the four major application groups. Note that each data point summarizes the five levels of machines heterogeneity and tasks heterogeneity.

The tendency that can be noted in this figure is that, in situations of smaller grain, the schedulers tend to have closer performances. This is because, when there are many tasks per machine, dynamic schedulers can keep all processors busy most of the time. Only during the very end of application execution (when some machines go idle), this situation changes due to the load unbalance and harms performance. As these grains grow, however, the differences between the schedulers' behaviors come up. Accordingly to Figure 1A, WQR achieves better performance than the other schedulers on applications that have up to 25000 of mean task size. By growing the mean task size, the trend is that all scheduling algorithms performance become worse because higher application granularity raises difficulties on the scheduling process. The larger the task size, the greater chances are that the performance of the machine executing it degrades. This machine can be shared among other users making it so slow that the execution of the whole application becomes impaired. This fact indicates that, schedule a big task to the faster machine in the moment is not always the best choice.

This phenomenon affects *Sufferage* and *Dynamic FPLTF*, algorithms that use information about the environment and the applications to make their scheduling decisions. As can be seen in Figure 1A, as the mean task size grows these algorithms have a nearly linear fall in their performances.

Workqueue achieves the same level of *Sufferage* and *Dynamic FPLTF* performances in situations where the grains (mean task size) are small, up to 5000 seconds. With larger tasks (25000 and 125000 seconds), *Workqueue* gets worse. This happens because this algorithm does not have any mechanism to avoid the scheduling of a large task to a slow machine at the end of the scheduling plan. *Workqueue* is better when the number of tasks is considerable higher than the number of machines, more tasks per machine.

The three WQR versions achieve better performance than the others schedulers in situations where there are more tasks than machines on the grid, up to 25000 seconds of mean task size. This better performance can be credited to the replication strategy. With this strategy, even if a task is initially scheduled to a slow machine, this task can be replicated on a faster machine and finishes before the "original one". Furthermore, even if it does not happen, the impairment to the final execution time can be not so big because the tasks are not very large. For the applications where the tasks are larger and the relation tasks per machine is less than 1 (mean task size = 125000 seconds), WQR does not achieve a good performance. Moreover, *Sufferage* and *Dynamic FPLTF* achieve better performance than WQR. The fall of the performance can be credited to the high application granularity. In cases like that, a large task and also its replica(s) can be scheduled to slow machines, impairing the whole application execution.

Figure 1B exhibits the impact of the grid machines heterogeneity on the schedulers' performances. Similar as before, each data point summarizes the five levels of tasks heterogeneity and the four major types of applications.

Workqueue does not achieve a good performance compared to the others schedulers, but shows some constancy on its performance while the machines heterogeneity level is less or equal to 8. On level 16, its performance is considerably worse. This happens because, as higher is the heterogeneity, higher can be the difference of a fast machine and a slow one. Choosing a slow machine to execute a large task causes a great impact on application execution time.

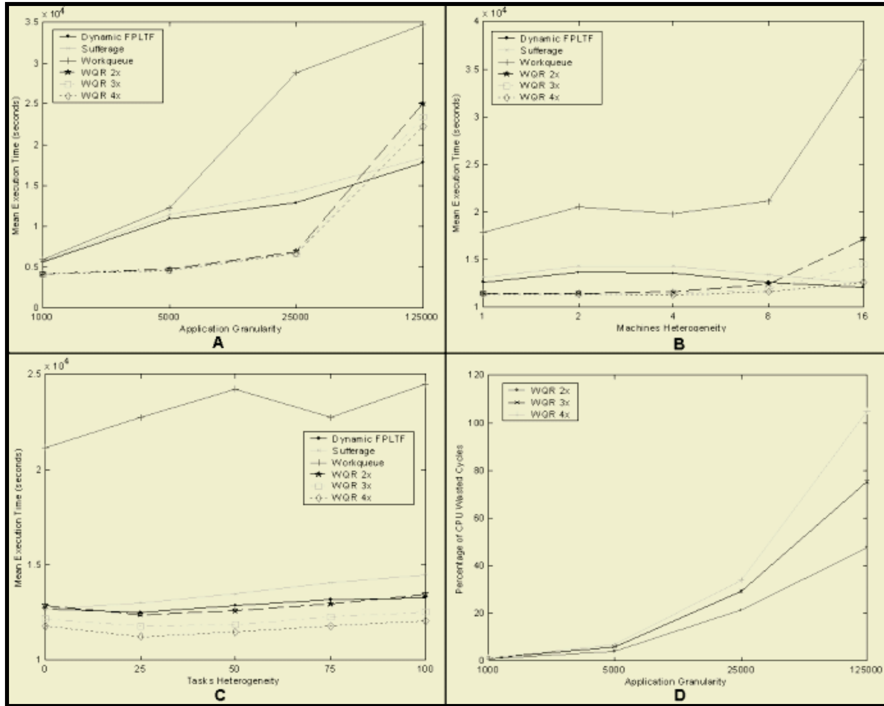

Fig. 1. (A) Schedulers Performance by Application Granularity. (B) Shedulers Performance by Machines Heterogeneity. (C) Schedulers Performance by Tasks Heterogeneity. (D) Percentage of Wasted CPU Cycles by Application Granularity

Although with better numbers, WQR displays a trend similar to *Workqueue*. Up to the level 8 of machines heterogeneity, WQR performance maintains practically unaltered. At the level 16 of machines heterogeneity, WQR suffers a fall of performance, but it is very slightly compared to the *Workqueue's* fall. Nevertheless, WQR performance can be increased using more replicas, reducing the fall. This fall occurs due to the same reason of *Workqueue* but its effects are minimized due to the replication strategy used by WQR. Large tasks assigned to slow machines can be replicated to faster machines and finish before the "original ones".

The *Sufferage* and *Dynamic FPLTF* have an almost inverse behavior related to the machines heterogeneity variation. Growing the level of grid machines heterogeneity, the performance of these algorithms gets better. On level 16, both performances overcome WQR. This performance improvement achieved by these algorithms can be credited to the ability of choosing powerful machines to execute the larger tasks. Small tasks assigned to slow machines do not have a relevant impact on application execution time.

Figure 1C shows the impact of application tasks heterogeneity on the performance of the scheduling algorithms. On this chart, *Workqueue* confirms again its poorer performance quality compared to the remainder algorithms.

Sufferage and *Dynamic FPLTF* achieve closer results, but the second one suffers the least the impact of different sizes of tasks. The *Dynamic FPLTF* better results can be credited to its ability to order the application tasks by their sizes (the larger tasks are firstly assigned). Prioritizing "the task that suffers the most" to make the scheduling plan, the *Sufferage* algorithm not always prioritize larger tasks. The 2x version of WQR achieves a performance very similar to *Dynamic FPLTF*, proofing that it is a good alternative when it is not possible to obtain information about the environment and application. Increasing the replication level (3x and 4x), WQR still achieves better results. These two versions overcome the performance of the algorithms that use information about the environment and application to make the scheduling plan.

To achieve the great results presented above, WQR wastes CPU cycles with the replication, as shown in Figure 1D. The percentage of wasted CPU cycles is obtained from the division of the total cycles wasted with replicas by the total of cycles consumed (wasted and useful).

The cycles waste is considerably smaller for applications with 1000 and 5000 seconds of mean tasks sizes, achieving less than 5% of wasting. This percentage increases when the mean tasks sizes grows to 25000 seconds, but still the percentage of wasted cycles does not exceeds 40%. In the 125000 seconds category of applications, the percentage of wasted CPU cycles can exceed 100%. The increase of cycles wasting when the application granularity grows occurs due to the quick start of replication. When the application has small grains, the replication only starts after executing many tasks of the application and the fraction of time that the remaining tasks represent is small. When the application has large grains (e.g. 125000, more machines available than application tasks), the replication starts immediately wasting more cycles. Nevertheless, the replication can be limited to 2x and the percentage of wasting cycles can be maintained fewer than 50% as shown in the figure.

3.3 Implementation

We are finishing the process of implementing *Workqueue with Replication* (WQR) as part of MyGrid, a user-level tool for running BoT applications on grids [2][11]. So far, we have been able to observe that MyGrid's overhead for task starting and cancellation is small. For tasks created "nearby" (the scheduler and the remote machine within the same local network), task overhead has not exceeded 7 seconds in 300 trials. For tasks created "far away" (the scheduler in Campina Grande, Brazil and the remote

machine in San Diego, USA), task overhead has not exceeded 17 seconds within 300 trials. Since the overhead has been small (compared to task size) and stable (has not changed much with the trials), we think it will not affect our results. In fact, since the overhead is stable, we could just include it in the task execution time, little changing our experiments.

4 Conclusions

In this paper we have proposed the *Workqueue with Replication* (WQR) task scheduling algorithm for computational grids. WQR targets Bag-of-Tasks applications, which are composed by independent tasks with no inter-task communication. WQR is similar to the classic *Workqueue* algorithm, but when hosts would otherwise go idle, they are used to compute replicas of the tasks that are still running. The idea here is that among a set of task replicas, the first one to finish is considered as being the normal execution of the task in question and the other copies are killed to make the hosts available.

WQR is not based on any kind of performance information, yet unlike other knowledge-free approaches, achieves good and consistent performance. This good performance is attained with additional increase in resource consumption to compensate the fact that WQR does not base its scheduling plan on information about machines and tasks. It is important to note that this additional resource consumption can be controlled by limiting replication, a solution that has little impact on performance. We feel that WQR is an important and practical contribution to a real problem. We are currently deploying WQR as part of MyGrid [2] [11]. Our hope is that this effort will enable more people to effectively compute on the grid.

Future work will be based on improvements to our grid model, including modeling file transfers present in I/O intensive applications that have not staged their data. A failure model will be included to encompass problems that make machines unavailable. We will also study the emergent behavior caused by multiple instances of our scheduler in the same grid. Of course, additional resource consumption imposes greater total load on the grid. On the other hand, individual applications finish earlier with WQR leaving a less loaded grid for applications that arrive in the future. Finally, we will try a simulation-based perfect algorithm to make a performance comparison with the replication approach. The idea is to determine a lower bound for our scheduling problem.

References

[1] H. Casanova, A. Legrand and D. Zagorodnov et al. *Heuristics for Scheduling Parameter Sweep Applications in Grid Environments*. HCW. 2000.

[2] W. Cirne, D. Paranhos and L.Costa et al. *Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach*. Submitted for publication. April 2003.

[3] R. Dragan. *The Meaning of Moore's Law*. http://www.pcmag.com/article2/0,4149,4092,00.asp. Online on February 14th 2003.

[4] W. Elwasif, J. Plank and R. Wolski. *Data Staging Effects in Wide Area Task Farming Applications*. IEEE ISCC and the Grid, Brisbane, Australia, May 2001.

[5] P. Francis, S. Jamin and V. Paxson et al. *An Architecture for a Global Internet Host Distance Estimation Service*. Proceedings of IEEE INFOCOM, 1999.

[6] H. James, K. Hawick and P. Coddington. *Scheduling Independent Tasks on Metacomputing Systems*. The University of Adelaide. DHPC-066, 1999.

[7] M. Litzkow, M. Livny, and M. Mutka. *Condor: A Hunter of Idle Workstations*. Proc. 8th International Conference of Distributed Computing Systems, pp. 104–111, 1988.

[8] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. *A Resource Query Interface for Network-Aware Applications*. 7th IEEE HPDC, July 1998.

[9] M. Maheswaran, S. Ali and H. Siegel et al. *Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems*. HCW, 1999.

[10] D. Menascé, D. Saha and S. Porto et al. *Static and Dynamic Processor Scheduling Disciplines in Heterogeneous Parallel Architectures*. JPDC, pp. 1–18. 1995.

[11] *MyGrid Web Page*. http://dsc.ufcg.edu.br/mygrid/. Online on February 14th 2003.

[12] J. Plank, M. Beck and W. Elwasif et al. *The Internet Backplane Protocol: Storage in the network*. In NetStore '99: Network Storage Symposium. Internet2, October 1999.

[13] *Simgrid*. http://grail.sdsc.edu/projects/simgrid/. Online on February 14th 2002.

[14] R. Wolski. *Dynamically Forecasting Network Performance Using the Network Weather Service*. Cluster Computing, 1(1): 119–132, 1998.