Adaptive Scheduling for Master-Worker Applications on the Computational Grid

Elisa Heymann[1], Miquel A. Senar[1], Emilio Luque[1] and Miron Livny[2]

[1] Unitat d'Arquitectura d'Ordinadors i Sistemes Operatius
Universitat Autònoma de Barcelona
Barcelona, Spain
{e.heymann, m.a.senar, e.luque}@cc.uab.es

[2] Department of Computer Sciences
University of Wisconsin– Madison
Wisconsin, USA
miron@cs.wisc.edu

Abstract*. We address the problem of how many workers should be allocated for executing a distributed application that follows the master-worker paradigm, and how to assign tasks to workers in order to maximize resource efficiency and minimize application execution time. We propose a simple but effective scheduling strategy that dynamically measures the execution times of tasks and uses this information to dynamically adjust the number of workers to achieve a desirable efficiency, minimizing the impact in loss of speedup. The scheduling strategy has been implemented using an extended version of MW, a runtime library that allows quick and easy development of master-worker computations on a computational grid. We report on an initial set of experiments that we have conducted on a Condor pool using our extended version of MW to evaluate the effectiveness of the scheduling strategy.

1. Introduction

In the last years, Grid computing [1] has become a real alternative to traditional supercomputing environments for developing parallel applications that harness massive computational resources. However, by its definition, the complexity incurred in building such parallel Grid-aware applications is higher than in traditional parallel computing environments. Users must address issues such as resource discovery, heterogeneity, fault tolerance and task scheduling. Thus, several high-level programming frameworks have been proposed to simplify the development of large parallel applications for Computational Grids (for instance, Netsolve [2], Nimrod/G [3], MW [4]).

Several programming paradigms are commonly used to develop parallel programs on distributed clusters, for instance, Master-Worker, Single Program Multiple Data

* This work was supported by the CICYT (contract TIC98-0433) and by the Commission for Cultural, Educational and Scientific Exchange between the USA and Spain (project 99186).

(SPMD), Data Pipelining, Divide and Conquer, and Speculative Parallelism [5]. From the previously mentioned paradigms, the Master-Worker paradigm (also known as task farming) is especially attractive because it can be easily adapted to run on a Grid platform. The Master-Worker paradigm consists of two entities: a master and multiple workers. The master is responsible for decomposing the problem into small tasks (and distributes these tasks among a farm of worker processes), as well as for gathering the partial results in order to produce the final result of the computation. The worker processes execute in a very simple cycle: receive a message from the master with the next task, process the task, and send back the result to the master. Usually, the communication takes place only between the master and the workers at the beginning and at the end of the processing of each task. This means that, master-worker applications usually exhibit a weak synchronization between the master and the workers, they are not communication intensive and they can be run without significant loss of performance in a Grid environment.

Due to these characteristics, this paradigm can respond quite well to an opportunistic environment like the Grid. The number of workers can be adapted dynamically to the number of available resources so that, if new resources appear they are incorporated as new workers in the application. When a resource is reclaimed by its owner, the task that was computed by the corresponding worker may be reallocated to another worker.

In evaluating a Master-Worker application, two performance measures of particular interest are *speedup* and *efficiency*. Speedup is defined, for each number of processors n, as the ratio of the execution time when executing a program on a single processor to the execution time when n processors are used. Ideally we would expect that the larger the number of workers assigned to the application the better the speedup achieved. Efficiency measures how good is the utilization of the n allocated processors. It is defined as the ratio of the time that n processors spent doing useful work to the time those processors would be able to do work. Efficiency will be a value in the interval [0,1]. If efficiency is becoming closer to 1 as processors are added, we have linear speedup. This is the ideal case, where all the allocated workers can be kept usefully busy.

In general, the performance of master-worker applications will depend on the temporal characteristics of the tasks as well as on the dynamic allocation and scheduling of processors to the application. In this work, we consider the problem of maximizing the speedup and the efficiency of a master-worker application through both the allocation of the number of processors on which it runs and the scheduling of tasks to workers at runtime.

We address this goal by first proposing a generalized master-worker framework, which allows adaptive and reliable management and scheduling of master-worker applications running in a computing environment composed of opportunistic resources. Secondly, we propose and evaluate experimentally an adaptive scheduling strategy that dynamically measures application efficiency and task execution times, and uses this information to dynamically adjust the number of processors and to control the assignment of tasks to workers.

The rest of the paper is organized as follows. Section 2 reviews related work in which the scheduling of master-worker applications on Grid environments was studied. Section 3 presents the generalized Master-Worker paradigm. Section 4 presents a definition of the scheduling problem and outlines our adaptive scheduling strategy for master-worker applications. Section 5 describes the prototype implementation of the scheduling strategy and section 6 shows some experimental data obtained when the proposed scheduling strategy was applied to some synthetic applications on a real grid environment. Section 7 summarizes the main results presented in this paper and outlines our future research directions.

2. Related Work

One group of studies has considered the problem of scheduling master-worker applications with a single set of tasks on computational grids. They include AppLeS [6], NetSolve [7] and Nimrod/G [3].

The AppLeS (Application-Level Scheduling) system focuses on the development of scheduling agents for parallel metacomputing applications. Each agent is written in a case-by-case basis and each agent will perform the mapping of the user's parallel application [8]. To determine schedules, the agent must consider the requirements of the application and the predicted load and availability of the system resources at scheduling time. Agents use the services offered by the NWS (Network Weather Service) [9] to monitor the varying performance of available resources.

NetSolve [2] is a client-agent-server system, which enables the user to solve complex scientific problems remotely. The NetSolve agent does the scheduling by searching for those resources that offer the best performance in a network. The applications need to be built using one of the API's provided by NetSolve to perform RPC-like computations. There is an API for creating task farms [7] but it is targeted to very simple farming applications that can be decomposed by a single bag of tasks.

Nimrod/G [3] is a resource management and scheduling system that focuses on the management of computations over dynamic resources scattered geographically over wide-area networks. It is targeted to scientific applications based on the "exploration of a range of parameterized scenarios" which is similar to our definition of master-worker applications, but our definition allows a more generalized scheme of farming applications. The scheduling schemes under development in Nimrod/G are based on the concept of computational economy developed in the previous implementation of Nimrod, where the system tries to complete the assigned work within a given deadline and cost. The deadline represents a time which the user requires the result and the cost represents an abstract measure of what the user is willing to pay if the system completes the job within the deadline. Artificial costs are used in its current implementation to find sufficient resources to meet the user's deadline.

A second group of researchers has studied the use of parallel application characteristics by processor schedulers of multiprogrammed multiprocessor systems, typically with the goal of minimizing average response time [10, 11]. However, the results from these studies are not applicable in our case because they were focussed basically on the allocation of jobs in shared memory multiprocessors in which the computing resources are homogeneous and available during all the computation. Moreover, most of these studies assume the availability of accurate historical performance data, provided to the scheduler simultaneously with the job submission.

They also focus on overall system performance, as opposed to the performance of individual applications, and they only deal with the problem of processor allocation, without considering the problem of task scheduling within a fixed number of processors as we do in our strategy.

3. A Generalized Master-Worker paradigm

In this work, we focus on the study of applications that follow a generalized Master-Worker paradigm because it is used by many scientific and engineering applications like software testing, sensitivity analysis, training of neural-networks and stochastic optimization among others. In contrast to the simple master-worker model in which the master solves one single set of tasks, the generalized master-worker model can be used to solve of problems that require the execution of several batches of tasks. Figure 1 shows an algorithmic view of this paradigm.

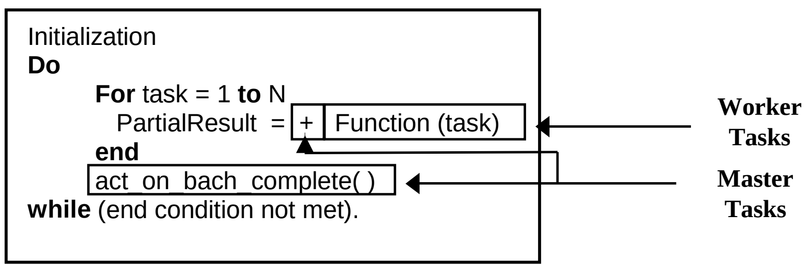

Fig. 1. Generalized Master-Worker algorithm

A Master process will solve the *N* tasks of a given batch by looking for Worker processes that can run them. The Master process passes a description (input) of the task to each Worker process. Upon the completion of a task, the Worker passes the result (output) of the task back to the Master. The Master process may carry out some intermediate computation with the results obtained from each Worker as well as some final computation when all the tasks of a given batch are completed. After that a new batch of tasks is assigned to the Master and this process is repeated several times until completion of the problem, that is, *K* cycles (which are later refereed as *iterations*).

The generalized Master-Worker paradigm is very easy to program. All algorithm control is done by one process, the Master, and having this central control point facilitates the collection of job's statistics, a fact that is used by our scheduling mechanism. Furthermore, a significant number of problems can be mapped naturally to this paradigm. N-body simulations [12], genetic algorithms [13], Monte Carlo simulations [14] and materials science simulations [15] are just a few examples of natural computations that fit in our generalized master-worker paradigm.

4. Challenges for scheduling of Master-Worker applications

In this section, we give a more precise definition of the scheduling problem for master-worker applications and we introduce our scheduling policy.

4.1. Motivations and background

Efficient scheduling of a master-worker application in a cluster of distributively owned resources should provide answers to the following questions:

- How many workers should be allocated to the application? A simple approach would consist of allocating as many workers as tasks are generated by the application at each iteration. However, this policy will incur, in general, in poor resource utilization because some workers may be idle if they are assigned a short task while other workers may be busy if they are assigned long tasks.
- How to assign tasks to the workers? When the execution time incurred by the tasks of a single iteration is not the same, the total time incurred in completing a batch of tasks strongly depends on the order in which tasks are assigned to workers. Theoretical works have proved that simple scheduling strategies based on list-scheduling can achieve good performance [16].

We evaluate our scheduling strategy by measuring the efficiency and the total execution time of the application.

Resource efficiency (E) for n workers is defined as the ratio between the amount of time workers spent doing useful work and the amount of time workers were able to perform work.

$$E = \frac{\sum_{i=1}^{n} T_{work,i}}{\sum_{i=1}^{n} T_{up,i} - \sum_{i=1}^{n} T_{susp,i}}$$

n: Number of workers.

$\boldsymbol{T_{work,i}}$: Amount of time that worker i spent doing useful work.

$\boldsymbol{T_{up,i}}$: Time elapsed since worker i is alive until it ends.

$\boldsymbol{T_{susp,i}}$: Amount of time that worker i is suspended, that is, when it cannot do any work.

Execution Time (ET_n) is defined as the time elapsed since the application begins its execution until it finishes, using n workers.

$$ET = T_{finish,n} - T_{begin,n}$$

$\mathbf{T_{finish,n}}$: Time of the ending of the application when using n workers.

$\mathbf{T_{begin,n}}$: Time of the beginning of the application workers.

As [17] we view efficiency as an indication of benefit (the higher the efficiency, the higher the benefit), and execution time as an indication of cost (the higher the execution time, the higher the cost). The implied system objective is to achieve efficient usage of each processor, while taking into account the cost to users. It is important to know, or at least to estimate the number of processors that yield the point at which the ratio between efficiency to execution time is maximized. This would represent the desired allocation of processors to each job.

4.2. Proposed Scheduling Policy

We have considered a group of master-worker applications with an iterative behavior. In these iterative parallel applications a batch of parallel tasks is executed K times (iterations). The completion of a given batch induces a synchronization point in the iteration loop, followed by the execution of a sequential body. This kind of applications has a high degree of predictability, therefore it is possible to take advantage of it to decide both the use of the available resources and the allocation of tasks to workers.

Empirical evidence has shown that the execution of each task in successive iterations tends to behave similarly, so that the measurements taken for a particular iteration are good predictors of near future behavior [15]. As a consequence, our current implementation of adaptive scheduling employs a heuristic-based method that uses historical data about the behavior of the application, together with some parameters that have been fixed according to results obtained by simulation.

In particular, our adaptive scheduling strategy collects statistics dynamically about the average execution time of each task and uses this information to determine the number of processors to be allocated and the order in which tasks are assigned to processors. Tasks are sorted in decreasing order of their average execution time. Then, they are assigned to workers according to that order. At the beginning of the application execution, no data is available regarding the average execution time of tasks. Therefore, tasks are assigned randomly. We call this adaptive strategy *Random and Average* for obvious reasons.

Initially as many workers as tasks per iteration (N) are allocated for the application. We first ask for that maximum number of workers because getting machines in an opportunistic environment is time-consuming. Once we get the maximum number of machines at the start of an application, we release machines if needed, instead of getting a lower number of machines and asking for more.

Then, at the end of each iteration, the adequate number of workers for the application is determined in a two-step approach. The first step quickly reduces the number of workers trying to approach the number of workers to the optimal value. The second step carries out a fine correction of that number. If the application exhibits a regular behavior the number of workers obtained by the first step in the initial iterations will not change, and only small corrections will be done by the second step.

The first step determines the number of workers according to the workload exhibited by the application. Table 1 is an experimental table that has been obtained from simulation studies. In these simulations we have evaluated the performance of different strategies (including *Random and Average* policy) to schedule tasks of master-worker applications. We tested the influence of several factors: the variance of tasks execution times among iterations, the balance degree of work among tasks, the number of iterations and the number of workers used [18].

Table 1 shows the number of workers needed to get efficiency greater than 80% and execution time less than 1.1 the execution time when using *N* workers. These values would correspond to a situation in which resources are busy most of the time while the execution time is not degraded significantly.

Table 1. Percentage of workers with respect to the number of tasks.

Workload	<30%	30%	40%	50%	60%	70%	80%	90%
%workers (largest tasks similar size)	Ntask	70%	55%	45%	40%	35%	30%	25%
%workers (largest tasks diff. size)	60%	45%	35%	30%	25%	20%	20%	20%

The first row contains the *workload*, defined as the work percentage done when executing the largest 20% tasks. The second and third rows contain the workers percentage with respect to the number of tasks for a given workload in the cases that the 20% largest tasks have similar and different executions times respectively.

For example, if the 20% largest tasks have carried out 40% of the total work then the number of workers to allocate will be either *N**0,55 or *N**0,35. The former value will be used if the largest tasks are similar, otherwise the later value is applied. According to our simulation results the largest tasks are considered to be similar if their execution time differences are not greater than 20%.

The fine correction step is carried out at the end of each iteration when the workload between iterations remains constant and the ratio between the last iteration execution time and the execution time with the current number of workers given by table 1 is less than 1.1. This correction consists of diminishing by one the number of workers if efficiency is less than 0.8, and observing the effects on the execution time. If it gets worse a worker is added, but never surpassing the value given by table 1. The complete algorithm is shown in figure 2.

```
1.  In the first iteration Nworkers = Ntasks
    Next steps are executed at the end of each iteration i.
2.  Compute Efficiency, Execution Time, Workload and the Differences of the execution times
    of the 20% largest tasks.
3.  if (i == 2)
          Set Nworkers = NinitWorkers according to Workload and Differences of Table 1.
    else
          if (Workload of  iteration i != Workload of iteration i-1)
             Set Nworkers = NinitWorkers according to Workload and Differences of  Table 1
          else
             if (Execution Time of it. i DIV Execution Time of it. 2 (with NinitWorkers)  <= 1.1)
                   if (Efficiency of iteration i < 0.8)
                              Nworkers = Nworkers – 1
            else
                   Nworkers = Nworkers + 1
```

Fig. 2. Algorithm to determine *Nworkers.*

5. Current implementation

To evaluate both the proposed scheduling algorithm and the technique to adjust the number of workers we have run experiments on a Grid environment using MW library as a Grid middleware. First, we will briefly review the main characteristics of MW and then we will summarize the extensions included to support both our generalized master-worker paradigm and the adaptive scheduling policy.

5.1. Overview of MW

MW is a runtime library that allows quick and easy development of master-worker computations on a computational grid [4]. It handles the communication between master and workers, asks for available processors and performs fault-detection. An application in MW has three base components: Driver, Tasks and Workers. The Driver is the master, who manages a set of user-defined tasks and a pool of workers. The Workers execute Tasks. To create a parallel application the programmer needs to implement some pure virtual functions for each component.

Driver: This is a layer that sits above the program's resource management and message passing mechanisms. (Condor [19] and PVM [20], respectively, in the implementation we have used). The Driver uses Condor services for getting machines to execute the workers and to get information about the state of those machines. It creates the tasks to be executed by the workers, sends tasks to workers and receives the results. It handles workers joining and leaving the computation and rematches running tasks when workers are lost. To create the Driver, the user needs to implement the following pure virtual functions:

- get_userinfo(): Processes arguments and does initial setup.
- setup_initial_tasks(): Creates the tasks to be executed by the workers.
- pack_worker_init_data(): Packs the initial data to be sent to the worker upon startup.
- act_on_completed_task(): This is called every time a task finishes.

Task: This is the unit of work to be done. It contains the data describing the tasks (inputs) and the results (outputs) computed by the worker. The programmer needs to implement functions for sending and receiving this data between the master and the worker.

Worker: This executes the tasks sent to it by the master. The programmer needs to implement the following functions:

- unpack_init_data(): Unpacks the initialization data passed in the Driver pack_worker_init_data() function.
- execute_task(): Computes the results for a given task.

5.2. Extended version of MW

In its original implementation, MW supported one master controlling only one set of tasks. Therefore we have extended the MW API to support our programming model, the *Random and Average* scheduling policy and to collect useful information to adjust the number of workers.

To create the master process the user needs to implement another pure virtual function: **global_task_setup**. There are also some changes in the functionality of some others pure virtual functions:

- **global_task_setup()**: It initializes the data structures needed to keep the tasks results the user want to record. This is called once, before the execution of the first iteration.
- **setup_initial_tasks (iterationNumber)**: The set of tasks created depends on the iteration number. So, there are new tasks for each iteration, and these tasks could depend on values returned by the execution of previous tasks. This function is called before each iteration begins, and creates the tasks to be executed in the *iterationNumber* iteration.
- **get_userinfo()**: The functionality of this function remains the same, but the user needs to call the following initialization functions there:
  - **set_iteration_number (n)**: This is used to set the number of times tasks will be created and executed, that is, the number of iterations. If **INFINITY** is used to set the iterations number, then tasks will be created and executed until an end condition is achieved. This condition needs to be set in the function **end_condition()**.
  - **set_Ntasks (n)**: This is used to set the number of tasks to be executed per iteration.
  - **set_task_retrive_mode (mode)**: This function allows the user to select the scheduling policy. It can be FIFO (**GET_FROM_BEGIN)**, based on a user key (**GET_FROM_KEY**), random (**GET_RANDOM**) or random and average (**GET_RAND_AVG**).
  - **printresults (iterationNumber)**: It allows the results of the *iterationNumber* iteration to be printed.

In addition to the above changes, the MW*Driver* collects statistics about tasks execution time, workers' state (when they are alive, working and suspended), and about iteration beginning and ending.

At the end of each iteration, function **UpdateWorkersNumber()** is called to adjust the number of workers accordingly with regard to the algorithm explained in the previous section.

6. Experimental study in a grid platform

In this section we report the preliminary set of results obtained with the aim of testing the effectiveness of the proposed scheduling strategy. We have executed some synthetic master-worker applications that could serve as representative examples of the generalized master-workers paradigm. We run the applications on a grid platform and we have evaluated the ability of our scheduling strategy to dynamically adapt the number of workers without any *a priori* knowledge about the behavior of the applications.

We have conducted experiments using a grid platform composed of a dedicated Linux cluster running Condor, and a Condor pool of workstations at the University of Wisconsin. The total number of available machines was around 700 although we restrict our experiments to machines with Linux architecture (both from the dedicated cluster and the Condor pool). The execution of our application was carried out using the grid services provided by Condor for resource requesting and detecting, determining information about resources and fault detecting. The execution of our application was carried out with a set of processors that do not exhibit significant differences in performance, so that the platform could be considered to be homogeneous.

Our applications executed 28 synthetic tasks at each iteration. The number of iterations was fixed to 35 so that the application was running in a steady state most of the time. Each synthetic task performed the computation of a Fibonacci series. The length of the series computed by each task was randomly fixed at each iteration in such a way that the variation of the execution time of a given task in successive iterations was 30%. We carried out experiments with two synthetic applications that exhibited a workload distribution of 30% and 50% approximately. In the former case, all large tasks exhibited a similar execution time. In the latter case, the execution time of larger tasks exhibited significant differences. These two synthetic programs can be representative examples for master-worker applications with a highly balanced distribution of workload and medium balanced distribution of workload between tasks, respectively. Figure 3 shows, for instance, the average and the standard deviation time for each of the 28 tasks in the master-worker with a 50% workload.

Different runs on the same programs generally produced slightly different final execution times and efficiency results due to the changing conditions in the grid environment. Hence, average-case results are reported for sets of three runs.

Tables 2 and 3 show the efficiency, the execution time (in seconds) and the speedup obtained by the execution of the master-worker application with 50% workload and 30% workload, respectively. The results obtained by our adaptive scheduling are shown in bold in both tables. In addition to these results, we show the results obtained when a fixed number of processors were used during the whole execution of the application. In particular, we tested a fixed number of processors of n=28, n=25, n=20, n=15, n=10, n=5 and n=1. In all cases the order of execution was carried out according to the sorted list of average execution time (as described in previous section for the *Random and Average* policy). The execution time for n=1 was used to compute the speedup of the other cases. It is worth pointing out that the number of processors allocated by our adaptive strategy was obtained basically through table 1. Only in the case of 30% workload, did the fine adjustment carry out the additional reduction of the number of processors.

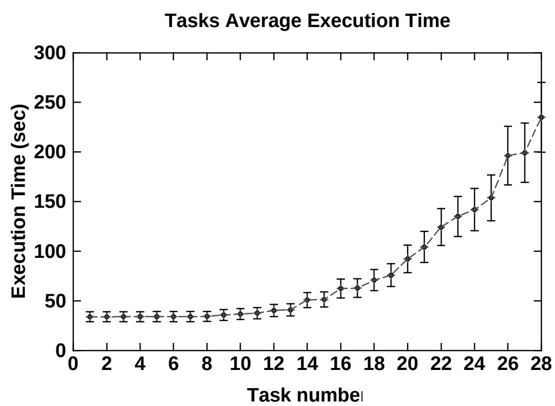

Fig. 3. Tasks execution times.

Table 2. Experimental results in the execution of a master-worker application with 50% workload using the *Random and Average* policy.

#Workers	1	5	**8**	10	15	20	25	28
Efficiency	1	0,94	**0,80**	0,65	0,43	0,33	0,28	0,22
Exec. Time	80192	16669,5	**12351**	12365	13025	12003	12300,5	12701
Speedup	1	4,81	**6,49**	6,49	6,16	6,68	6,52	6,31

Table 3. Experimental results in the execution of a master-worker application with 30% workload using the *Random and Average* policy.

#Workers	1	5	10	15	**18**	20	25	28
Efficiency	1	0,85	0,85	0,87	**0,78**	0,72	0,59	0,55
Exec. Time	36102	9269	4255	3027	**2459**	2710	2794	2434
Speedup	1	3,89	8,48	11,93	**14,68**	13,32	12,92	14,83

The first results shown in tables 2 and 3 are encouraging as they prove that an adaptive scheduling policy like *Random and Average* was able, in general, to achieve a high efficiency in the use of resources while the speedup was not degraded significantly. The improvement in efficiency can be explained because our adaptive strategy tends to use a small number of resources with the aim of avoiding idle time in workers that compute short tasks. In general, the larger the number of processors the larger the idle times incurred by workers in each iteration. This situation is also more remarkable when the workload of the application is more unevenly distributed among tasks. Therefore, for a given number of processors the largest loss of efficiency was obtained normally in the application with a 50% workload.

It can also be observed in both tables that the adaptive scheduling strategy obtained in general an execution time that was similar or even better than the execution time obtained with a larger number of processors. This result basically reflects the opportunistic nature of the resources that were used in our experiments. The larger the number of processors allocated, the larger the number of task suspensions and reallocations incurred at run time. The need to terminate a task prematurely when the user claimed back the processor prevented normally the benefits in execution time obtained by the use of additional processors. Therefore, from our results, we conclude that, the reduction in the number of processors allocated to an application running in an opportunistic environment is good not only because it improves overall efficiency, but it also avoids side effects on the execution time due to suspensions and reallocations of tasks.

As is perhaps to be expected, the best performance was normally obtained when the largest number of machines were used, although better machine efficiencies were obtained when a smaller number of machines were used. These results may seem to be obvious, but it should be stressed that they have been obtained from a real test-bed, in which resources were obtained from a total pool of non-dedicated 700 machines. In this test-bed our adaptive scheduler used only statistics information collected at runtime, and the execution of our applications should copse with the effects of resource obtaining, local suspension of tasks, task reassume and dynamic redistribution of load.

We carried out an additional set of experiments in order to evaluate the influence in the order of task assignment. Due to time constraints, this article only contains the results obtained when a master-worker application with 50% workload was scheduled using a *Random* policy. In this policy, when a worker becomes idle, a random task from the list of pending tasks is chosen and assigned to it. As can be seen when tables 2 and 4 are compared, the order in which tasks are assigned has a significant impact when a small number of workers is used. For less than 15 processors the *Random and Average* policy performs significantly better than the *Random* policy, both in efficiency and in execution time. When 15 or more processors are used, differences between both policies were nearly negligible. This fact can be explained because when the *Random* policy has a large number of available processors, the probability to assign a large task at the beginning is also large. Therefore, in these situations the assignments carried out by both polices are likely to follow a similar order. Only in the case of 20 processors, was *Random's* performance significantly worse than *Random & Average*. However, this could be explained because the tests of the Random policy with 20 processors suffered from many task suspensions and reallocations during their execution.

Table 4. Experimental results for Random scheduling with a master-worker application with 50% workload.

#Workers	1	5	10	15	20	25	28
Efficiency	1	0,80	0,56	0,40	0,34	0,26	0,26
Exec. Time	80192	20055	14121	13273	13153	12109	12716
Speedup	1	4,00	5,68	6,04	6,10	6,62	6,31

7. Conclusions and future work

In this paper, we have discussed the problem of scheduling master-worker applications on the computational grid. We have presented a framework for master-worker applications that allow the development of a tailored scheduling strategy. We have proposed a scheduling strategy that is both simple an adaptive and takes into account the measurements taken during the execution of the master-worker application. This information is usually a good predictor of near future behavior of the application. Our strategy tries to allocate and schedule the minimum number of processors that guarantees a good speedup by keeping the processors as busy as possible and avoiding situations in which processors sit idle waiting for work to be done. The strategy allocates the suitable number of processors by using the runtime information obtained from the application, together with the information contained in an empirical table that has been obtained by simulation. Later, the number of processors would eventually be adapted dynamically if the scheduling algorithm detects that the efficiency of the application can be improved without significant losses in performance.

We have built our scheduling strategy using MW as a Grid middleware. And we tested the scheduling strategy on a Grid environment made of several pools of machines, the resources of which were provided by Condor. The preliminary set of tests with synthetic applications allowed us to validate the effectiveness of our scheduling strategy. In general, our adaptive scheduling strategy achieved an efficiency in the use of processors close to 80% while the speedup up of the application was close to the speedup achieved with the maximum number of processors. Moreover, we have observed that our algorithm quickly achieves a stable situation with a fixed number of processors.

There are some ways in which this work can be extended. We have tested our strategy on a homogeneous Grid platform where the resources were relatively closed and the influence of the network latency was negligible. A first extension will adapt the proposed scheduling strategy to handle a heterogeneous set of resources. In order to carry this out, a normalizing factor should be applied to the average execution times to index table 1. Another extension will focus on the inclusion of additional mechanisms that can be used when the distance between resources is significant (for instance, by packing more than one task to a distant worker in order to compensate network delays). A second extension will be oriented to the extension of the scheduling strategy to be applied for applications that are not iterative or that exhibit different behaviors at different phases of the execution. This extension would be useful for applications that follow, for instance, a Divide and Conquer paradigm or a Speculative Parallelism paradigm.

8. References

1. I. Foster and C. Kesselman, “The Grid: Blueprint for a New Computing Infraestructure”, Morgan-Kaufmann, 1999.

2. H. Casanova and J. Dongarra, "NetSolve: Network enabled solvers", IEEE Computational Science and Engineering, 5(3) pp. 57-67, 1998.
3. D. Abramson, J. Giddy, and L. Kotler, "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?", in Proc. of IPPS/SPDP'2000, 2000.
4. J.-P. Goux, S. Kulkarni, J. Linderoth, M. Yoder, "An enabling framework for master-worker applications on the computational grid", Tech. Report, University of Wisconsin – Madison, March, 2000.
5. L. M. Silva and R. Buyya, "Parallel programming models and paradigms", in R. Buyya (ed.), "High Performance Cluster Computing: Architectures and Systems: Volume 2", Prentice Hall PTR, NJ, USA, 1999.
6. F. Berman, R. Wolski, S. Figueira, J. Schopf and G. Shao, "Application-Level Scheduling on Distributed Heterogeneous Networks", Proc. of Supercomputing'96.
7. H. Casanova, M. Kim, J. S. Plank and J. Dongarra, "Adaptive scheduling for task farming with Grid middleware", International Journal of Supercomputer Applications and High-Performance Computing, pp. 231-240, Volume 13, Number 3, Fall 1999.
8. G. Shao, R. Wolski and F. Berman, "Performance effects of scheduling strategies for Master/Slave distributed applications", Technical Report TR-CS98-598, University of California, San Diego, September 1998.
9. R. Wolski, N. T. Spring and J. Hayes, "The Network Weather Service: a distributed resource performance forecasting service for metacomputing", Journal of Future Generation Computing Systems", Vol. 15, October, 1999.
10. T. B. Brecht and K. Guha, "Using parallel program characteristics in dynamic processor allocation policies", Performance Evaluation, Vol. 27 and 28, pp. 519-539, 1996.
11. T. D. Nguyen, R. Vaswani and J. Zahorjan, "Maximizing speedup through self-tuning of processor allocation", in Proc. of the Int. Par. Proces. Symp. (IPPS'96), 1996.
12. V. Govindan and M. Franklin, "Application Load Imbalance on Parallel Processors", in Proc. of the Int. Paral. Proc. Symposium (IPPS'96), 1996.
13. E. Cantu-Paz, "Designing efficient master-slave parallel genetic algorithms", in J. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. Fogel, M. Garzon D. E. Goldberg, H. Iba and R. Riolo, editors, Genetic Programming: Proceeding of the Third Annual Conference, San Francisco, Morgan Kaufmann, 1998.
14. J. Basney, B. Raman and M. Livny, "High throughput Monte Carlo", Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, San Antonio Texas, 1999.
15. J. Pruyne and M. Livny, "Interfacing Condor and PVM to harness the cycles of workstation clusters", Journal on Future Generations of Computer Systems, Vol. 12, 1996.
16. L. A. Hall, "Aproximation algorithms for scheduling", in Dorit S. Hochbaum (ed.), "Approximation algorithms for NP-hard problems", PWS Publishing Company, 1997.
17. D. L. Eager, J. Zahorjan and E. D. Lazowska, "Speedup versus efficiency in parallel systems", IEEE Transactions on Computers, vol. 38, pp. 408-423, 1989.
18. E. Heymann, M. Senar, E. Luque, M. Livny. "Evaluation of an Adaptive Scheduling Strategy for Master-Worker Applications on Clusters of Workstations". Proceedings of 7th Int. Conf. on High Performance Computing (HiPC'2000) (to appear).
19. M. Livny, J. Basney, R. Raman and T. Tannenbaum, "Mechanisms for high throughput computing", SPEEDUP, 11, 1997.
20. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, "PVM: Parallel Virtual Machine A User's Guide and Tutorial for Networked Parallel Computing", MIT Press, 1994.