Evaluation of an Adaptive Scheduling Strategy for Master-Worker Applications on Clusters of Workstations

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 investigate the problem arising in scheduling parallel applications that follow a master-worker paradigm in order to maximize both the resource efficiency and the application performance. We propose a simple scheduling strategy that dynamically measures application execution time and uses these measurements to automatically adjust the number of allocated processors to achieve the desirable efficiency, minimizing the impact in loss of speedup. The effectiveness of the proposed strategy has been assessed by means of simulation experiments in which several scheduling policies were compared. We have observed that our strategy obtains similar results to other strategies that use a priori information about the application, and we have derived a set of empirical rules that can be used to dynamically adjust the number of processors allocated to the application.

1. Introduction

The use of loosely coupled, powerful and low-cost commodity components (PCs or workstations, typically) connected by high-speed networks has resulted in the widespread usage of a technology popularly called cluster computing [1]. The availability of such clusters made them an appealing vehicle for developing parallel applications. However, not all parallel programs that run efficiently in a traditional parallel supercomputing environment can be moved to a cluster environment without significant loss of performance. In that sense, the Master-Worker paradigm is attractive because it can achieve similar performance in both environments as no high communication performance is usually required from the network infrastructure [2].

In this paradigm, a master process is responsible basically for distributing tasks among a farm of worker processes. Moreover, it is a good example of adaptive parallel computing because it can respond quite well to a scenario where applications are executed by stealing idle CPU cycles (we refer to these environments as non-

* 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).

dedicated clusters). The number of workers can be adapted dynamically to the number of available resources in such an opportunistic environment so that, if new resources appear they are incorporated as new workers for the application.

However, the use of non-dedicated clusters introduces the need for complex mechanisms such as resource discovery, resource allocation, process migration and load balancing. In the case of master-worker applications, the overhead incurred in discovering new resources and allocating them can be significantly alleviated by not releasing the resource once the task has been completed. The worker will be kept alive at the resource waiting for a new task. However, by doing so, an undesirable scenario may arise in which some workers may be idle while other workers are busy. This situation will result in a poor utilization of the available resources in which all the allocated workers are not kept usefully busy and, therefore, the application efficiency will be low. In this case, the efficiency may be improved by restricting the number of allocated workers.

If we consider the execution time, a different criteria will guide the allocation of workers because the more workers allocated for the application the lower the total execution time of the application. Then, the speedup of the application directly depends on the allocation of as many workers as possible.

In general, the execution of a master-worker application implies a trade-off between the speedup and the efficiency achieved. On the one hand, our aim is to improve the speedup of the application as new workers are allocated. On the other hand, we want to also achieve a high efficiency by keeping all the allocated workers usefully busy.

Obviously, 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. So, 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 processors during 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 cluster composed of opportunistic computing resources. Secondly, we propose and evaluate by simulation a scheduling strategy that dynamically measures application efficiency and task execution times to control the assignment of tasks to workers.

The rest of the paper is organized as follows. Section 2 presents the model of the Master-Worker applications that we are considering in this paper. Section 3 gives a more precise definition of the scheduling problem, introduces our scheduling policy and reviews some related work. Section 4 presents some simulation results obtained in the evaluation of the proposed strategy, by comparing our policy with other scheduling policies. Section 5 summarizes the main results presented in this paper.

2. The model for master-worker applications

In this work, we focus on the study of applications that follow a Master-Worker model that has been used to solve a significant number of problems such as Monte Carlo simulations [3] and material science simulations [4]. In this generalized master-worker model, the master process iteratively solves a batch of tasks. After completion of one task, the master process may perform some intermediate computations with the partial result obtained by the task. Subsequently, when the complete batch of tasks is finished the master may carry out some additional processing. 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 referred as *iterations*).

As can be seen in fig. 1, we are considering 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). Workers execute *Function (task)* and *PartialResult* is collected by the master. The completion of a given batch induces a synchronization point in the iteration loop which facilitates also the collection of job's statistics in the Master process.

```
Initialization
Do
        For task = 1 to N
          PartialResult  =  +   Function (task)
        end
        act_on_batch_complete( )
while (end condition not met).
```

Figure 1. A model for generalized Master-Worker applications.

In addition to these characteristics, empirical evidence has shown that, for a wide range of applications, 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 [4]. In the rest of the paper we will investigate to what extent an adaptive and dynamic scheduling mechanism may use historical data about the behavior of the master-worker application to improve its performance in an opportunistic environment.

3. Challenges for scheduling of Master-Worker applications

In this section we present the scheduling problem adopted in this work and we present also our proposed policy to solve it.

3. 1 Problem statement and related work

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 result, 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 should tasks be assigned 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.

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

Resource efficiency [5] for *n* workers is defined as the ratio between the amount of time workers have actually spent doing useful work and the amount of time workers were able to perform work, i.e. the time elapsed since worker *i* is alive until it ends minus the amount of time that worker *i* is suspended.

Execution Time is defined as the time elapsed from when the application begins its execution until it finishes, using *n* workers.

The problem of scheduling master-worker applications on cluster environments has been investigated recently in the framework of middleware environments that allow the development of adaptive parallel applications running on distributed clusters. They include NetSolve [6], Nimrod [7] and AppLeS [5]. NetSolve and Nimrod provide APIs for creating task farms that can only be decomposed by a single bag of tasks. Therefore, no historical data can be used to allocate workers. The AppLeS (Application-Level Scheduling) system focuses on the development of scheduling agents for parallel applications but in a case-by-case basis, taking into account the requirements of the application and the predicted load and availability of the system resources at scheduling time.

There are other works in the literature that have studied the use of parallel application characteristics by processor schedulers of multiprogrammed multiprocessor systems, typically with the goal of minimizing average response time [8]. The results from these studies are not directly applicable in our case because they were focussed on the allocation of jobs in shared memory multiprocessors without considering the problem of task scheduling within a fixed number of processors. However, their experimental results also confirm that iterative parallel applications usually exhibit regular behaviors that can be used by an adaptive scheduler.

3. 2 Proposed scheduling policy

Our adaptive and dynamic scheduling strategy employs a heuristic-based method that uses historical data about the behavior of the application. It dynamically collects statistics about the average execution time of each task and uses this information to determine the order in which tasks are assigned to processors. Tasks are sorted in decreasing order of their average execution time. Then, they are assigned dynamically to workers in a list-scheme, according to that order. At the beginning of the application execution, as no data is available regarding the average execution time of tasks, tasks are assigned randomly. We call this adaptive strategy *Random & Average*, although the random assignment is done only once, simply as a way to obtain information about the tasks' execution time.

4. Experimental study

In this section, we evaluate the performance of several scheduling strategies with respect to the efficiency and the execution time obtained when they are applied to schedule master-worker applications on homogeneous systems. As we have stated in previous sections, we focus our study on a set of applications that are supposed to exhibit a highly regular and predictable behavior. We will test different scheduling strategies that include both pure static strategies that do not take into account any runtime information and adaptive and dynamical strategies that try to learn from the application behavior.

As a main result from these simulation experiments, we are interested in obtaining information about how the proposed adaptive scheduling strategy performs on average, and some bounds for the worst case situations. Therefore, in our simulations we consider that the number of processors is available through the whole execution of the application (i.e. this would be the ideal case in which no suspensions occur).

4.1 Policies Description

The set of scheduling strategies used in the comparison were the following:

- **LPTF (Largest Processing Time First)**: For each iteration this policy first assigns the tasks with largest execution time. Before an iteration begins, tasks are sorted decreasingly by execution time. Then, each time a worker is ready to receive work, the master sends the next task of the list, that is, the task with largest execution time. It is well known that *LPTF* is at least ¾ of the optimum [9]. This policy needs to know the exact execution time of the tasks in advance, which is not generally possible in a real situation, therefore it is only used as a sort of upper bound in the performance achievable by the other strategies.
- **LPTF on Expectation**: It works in the same way as *LPTF*, but tasks are initially sorted decreasingly by the expected execution time. In each iteration tasks are assigned in that predefined order. If there is no variation of the execution time of the tasks, the behavior of this policy is the same as *LPTF*. This policy is static and non-adaptive, and represents the case in which the user has an approximately good knowledge of the behavior of the application and wants to control the execution of the tasks in the order that he specifies. Obviously, it is possible for a user to have an accurate estimation of the distribution of times between the tasks of the application, but in practice, small variations will affect the overall efficiency because the order of assignment is fixed by the user at the beginning.
- **Random**: For each iteration, each time a worker is ready to get work, a random task is assigned. This strategy represents the case of a pure dynamic method that does not know anything about the application. In principle, it would obtain the worst performance of all the presented strategies, therefore it will be used as a lower bound in the performance achievable by the other strategies.

4.2. Simulation Framework

All described scheduling policies have been simulated systematically, to obtain efficiency and execution time, with all the possible number of workers ranging from 1 to as many workers as numbers of tasks, considering the following factors:

- *Workload (W)*: This represents the work percentage done when executing the 20% largest tasks. We have considered 30%, 40%, 50%, 60%, 78% 80% and 90% workload values. A 30% workload would correspond to highly balanced applications in which near all the tasks exhibit a similar execution time. On the contrary, a 90% workload would correspond to applications in which a small number of tasks are responsible for the largest amount of work. Moreover, the 20% largest tasks can have similar or different execution times. They are similar if their execution time differences are not greater than 20%. The same happens to the other 80% of tasks. For each workload value we have undertaken simulations with the four possibilities (referred as *i-i* in figures of section 4.3).
- *Iterations (L)*: This represents the number of batches of tasks that are going to be executed. We have considered the following values: 10, 35, 50 and 100.
- *Variation (D)*: From the workload factor, we determine the base execution times for the tasks. Then, for each iteration a variation is applied to the base execution times of each task. Variations of 0%, 10%, 30%, 60% and 100% have been considered. When a 0% variation was used, the times of the task were constant along the different iterations. This case would correspond to very regular applications where the time of tasks is nearly the same in successive iterations. When a 100% variation was used, tasks exhibit significant changes in their execution time in successive iterations, corresponding to applications with highly irregular behavior.
- *Number of Tasks (T)*: We have considered applications with 30, 100 and 300 tasks. Thus we examine systems with a small, a medium or a large amount of tasks, respectively.

For each simulation scenario (fixing a certain value for *workload*, *iterations* and *variation*) the efficiency and execution time have been obtained using all the workers from 1 to *Number of Tasks*.

4.3. Simulation Results

Although we have conducted tests for all the commented values, in this section we present only those results that are the most interesting. We will illustrate with figures the results for 30 tasks since they prove to be representative enough for the results obtained with a larger number of tasks. Moreover, we emphasize those results with 30% and 100% deviation, representing low and high degrees of regularity. In real applications 100% deviation is not expected, but it allows us to evaluate the strategies under the worst case scenario.

In the rest of the section some relevant result figures for both efficiency and execution time are presented. The X-axis always contain the number of workers. The Y-axis contain the efficiency and the execution time values respectively. Five values *W*, *i-i*, *D*, *T* and *L* appear at the top of each figure. *W* stands for the workload, *i-i* describes the similarity of tasks, *D* stands for variation applied to task execution time at each iteration, *T* stands for the number of tasks and *L* for the number of iterations (loop). We now review the most relevant results obtained from our simulations.

***Effect of the number of iterations (L)*:** The number of iterations (L) that tasks are executed does not significantly affect efficiency for an adaptive strategy such as

Random & Average. Figure 2 shows the effect of varying the number of iterations, considering 30% workload and 100% deviation. This is the case when the effect of the number of iterations is the most significant. As can be seen when the number of iterations varies from 10 to 35 the gain in efficiency is less than 5%. When the number of iterations was greater than 35, no significant gain in efficiency was observed. Therefore, our proposed strategy achieves a good efficiency without needing a long number of iterations to acquire a precise knowledge of the application.

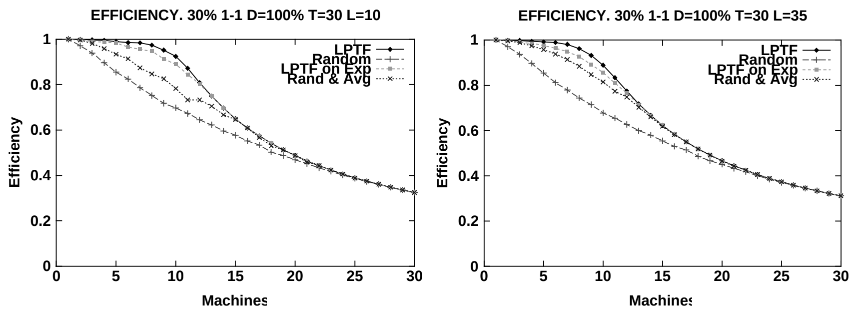

Figure 2. Effect of varying the number of iterations. (a) L=10 (b) L=35

***Effect of the workload (W)*:** Figure 3 shows the effect of varying the workload, considering 30% and 60% workload, 0% deviation and the same execution time for all the largest tasks, and for all the smallest tasks. As expected, for large workloads the number of workers that can usefully be busy is smaller than for small workloads. Moreover, when the workload is higher, efficiency declines faster. A large workload also implies a smoother curve in efficiency. It is important to point out that in all cases there is a point from which efficiency continuously declines. Before that point, small changes in the number of workers may imply significant and contradictory changes in efficiency.

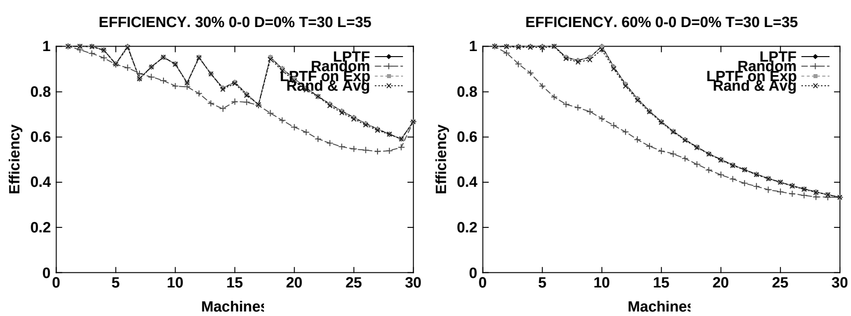

Figure 3. Effect of varying the workload. (a) W=30% (b) W=60%

***Effect of the tasks sizes (i-i)*:** The 20% largest tasks determine when the drop of efficiency begins. If they have the same execution time the decay in efficiency is delayed. The 80% smallest tasks have less influence, they basically determine the smoothness of the efficiency curve. If the 80% smallest tasks have the same execution times the efficiency curve have more peaks.

***Effect of the variation (D)*:** When deviation is higher, efficiency declines more. But it is worth noting that it does not decline abruptly even when deviation is 100%. For all policies, even for high values of deviation (60% or 100%), efficiency was never worsen more than 10% of the efficiency obtained with 0% deviation.

Finally, Figure 4 illustrates the overall behavior that we have obtained for the execution time when using the different scheduling policies. The execution time is measured in terms of the relative differences with the execution time of *LPTF* policy. As can be see, the *Random* policy always exhibits the worst execution time, especially when an intermediate number of processors are used. *Random & Average* and *LPTF on Expectation* achieve an execution time comparable to the execution time of *LPTF* even in the presence of a high variation in the execution time of the tasks.

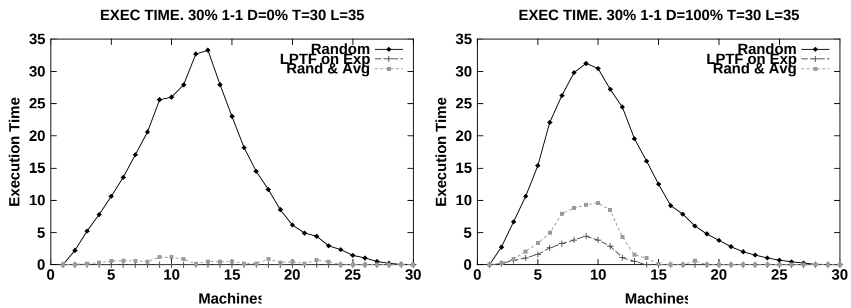

Figure 4. Execution time. (a) D=0% (b) D=100%

4.4. Discussion

We now summarize the main results that have been derived from all the simulations.

The number of iterations does not significantly affect either efficiency or execution time. The behavior of the policies was very similar for all the number of workers, but it was strongly affected by the variation of the execution times of the tasks in different iterations, by the workload and by having significant differences among the execution times of the 20% largest tasks.

Table 1 shows the efficiency bounds obtained for the previously described scheduling policies, always relative to *LPTF* policy. The first column contains the upper bound that is never surpassed in 95% of cases. The second column shows the upper bound for all the cases, which always corresponded to 30% 0-0 workload with D=100%, that is, tasks without significant execution time differences and with high variance. As can be seen, both *LPTF on Expectation* and *Random & Average* in most cases obtained an efficiency similar to the efficiency obtained by a policy such as *LPTF* that uses perfect information about the application. Even in the worst case (scenarios in which all tasks have a similar execution time but a high deviation (100%)) the loss of efficiency for both strategies was 17% approximately.

Table 1. Worst efficiency bounds for scheduling policies.

	Eff. Bound in 95% of cases	**Worst Efficiency Bound**
Random	25,4 %	26,96 %
Random & Average	8,65 %	16,86 %
LPTF on Expectation	8,91 %	17,29 %

Slightly better results were obtained for execution time. *Random & Average* and *LPTF on Expectation* never performed worse than 4% in more than 95% of the cases. Only in the presence of high variations were the differences increased to 8%. In all cases, the execution time of the *Random* policy was always between 25% and 30% worse than *LPTF*.

As a consequence of the simulations carried out, we can conclude that a simple adaptive strategy such as *Random & Average* will perform very well in terms of efficiency and execution time in most cases. Even in the presence of highly irregular applications the overall performance will not significantly worsen. Similar results have been obtained for the *LPTF on Expectation* policy, but the use of this policy implies that the user needs a good knowledge of the application. Therefore, *Random & Average* appears to be a promising strategy for solving the master-worker scheduling problem.

From our simulation we have also derived an empirical rule to determine the number of workers that must be allocated in order to get a good efficiency and a good execution time. The number of workers depends on the workload factor, on the differences among the execution times of the 20% largest tasks and on the variation of the execution times for different iterations. From our simulation results we have derived empirical table 2 which shows the number of processors that should be allocated, according to our simulations, for obtaining efficiency higher than 80% and execution time lower than 1.1 the time of executing the tasks with as many workers as tasks. This table gives an empirical value for the number of workers that ensures a smooth decrease in efficiency if more workers are added.

Table 2. 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%

4.5. Implementation on a Condor pool

The effectiveness of the *Random & Average* strategy has been tested in a real test bed, using a Condor [10] pool at the University of Wisconsin. Our applications consisted on a set of synthetic tasks that performed the computation of Fibonacci series. The execution of the application was carried out by using the services provided by MW [11]. In general, we have obtained efficiency values close to 0.8 and speedup values close to the maximum possible for the application [12].

5. Conclusions

In this paper we have discussed the problem of scheduling master-worker applications on clusters of homogeneous machines. We have proposed a scheduling policy that is both simple and adaptive, and takes into account the measurements taken during the execution of the tasks of the master-worker application. Our strategy tries to allocate and schedule the minimum number of processors that guarantee a good speedup by keeping the processors as busy as possible.

We have compared our strategy by simulation with several scheduling strategies using a large set of parameters to model different types of master-worker applications. And we also tested a preliminary version of the scheduling strategy on a cluster of machines, the resources of which were provided by Condor. The preliminary set of tests with synthetic applications allowed us to validate the results obtained in our simulations and 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 of the applications was similar to the speedup achieved with a higher number of processors.

We will continue this work by first adapting the proposed scheduling strategy to handle an heterogeneous set of resources. Another extension will focus on the inclusion of additional mechanisms that can be used when the distance between resources is significant.

6. References

1. R. Buyya (ed.), “High Performance Cluster Computing: Architectures and Systems”, Volume 1, Prentice Hall PTR, NJ, USA, 1999.
2. 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.
3. J. Basney, B. Raman and M. Livny, “High throughput Monte Carlo”, Proc. of the Ninth SIAM Conf. on Parallel Processing for Scientific Computing, San Antonio Texas, 1999.
4. 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.
5. G. Shao, R. Wolski and F. Berman, “Performance effects of scheduling strategies for Master/Slave distributed applications”, TR-CS98-598, U. of California, San Diego, 1998.
6. 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.
7. D. Abramson, R. Sosic, J. Giddy and B. Hall, “Nimrod: a tool for performing parameterised simulations using distributed workstations”, Symposium on High Performance Distributed Computing, Virginia, August, 1995.
8. 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.
9. L. A. Hall, “Aproximation algorithms for scheduling”, in Dorit S. Hochbaum (ed.), “Approximation algorithms for NP-hard problems”, PWS Publishing Company, 1997.
10. M. Livny, J. Basney, R. Raman and T. Tannenbaum, “Mechanisms for high throughput computing”, SPEEDUP, 11, 1997.

11. J.-P. Goux, S. Kulkarni, J. Linderoth, M. Yoder, "An enabling framework for master-worker applications on the computational grid", Tech. Rep, U. of Wisc. – Madison, 2000.
12. E. Heymann, M. Senar, E. Luque, M. Livny. "Adaptive Scheduling for Master-Worker Applications on the Computational Grid". Proceedings of the First International Workshop on Grid Computing (GRID 2000). (to appear)