Self-Adjusting Scheduling of Master-Worker Applications on Distributed Clusters*

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: Strategies for scheduling parallel applications on a distributed system must trade-off processor application speed-up and resource efficiency. Most existing strategies focus mainly on achieving high application speed-up without taking into account the efficiency factor. This paper presents our experiences with a self-adaptive scheduling strategy that dynamically adjusts the number of resources used by an application based on performance measures gathered during its execution. The strategy seeks to maximize resource efficiency while minimizing the impact in loss of speedup. It also uses the measured times to decide how to assign tasks to resources. This work has been carried out in the context of opportunistic clusters of machines and we report the results achieved by our strategy when it was applied to an image thinning application run on a Condor pool.

Keywords: Scheduling, resource management, cluster computing, Master-Worker applications.

1 Introduction

Scheduling of parallel tasks is one of the crucial issues that must be solved in order to achieve efficient execution in large-scale clusters of machines. Researchers have focused on the development of heuristic methods to solve the scheduling problem. In some cases, task scheduling is done prior to execution and is done only once –called static scheduling-. This static scheduling can be quite effective for computations for which a precise knowledge of their run-time behavior is available. However, this information is not usually available a priori for most applications. For these cases, it might be better to perform the scheduling periodically during run-time, as the problem's variable behavior more closely matches available computational resources. These techniques are usually referred to as dynamic scheduling.

* This work was supported by the CICYT (contract TIC98-0433), by the Commission for Cultural, Educational and Scientific Exchange between the USA and Spain (project 99186) and partially supported by the Generalitat de Catalunya (Grup de Recerca consolidat 1999SGR86).

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 742-751, 2001.

Dynamic scheduling techniques lend themselves well to many parallel programming paradigms (like Master-Worker, Divide and Conquer, or Speculative Parallelism [11]) which exhibit a dynamic behaviour that precludes the use of static scheduling techniques. From the previously mentioned paradigms, the Master-Worker paradigm is especially attractive because, as has been shown by empirical evidence [9], tasks executed by workers in successive iterations tend to behave similarly, so that the measurements taken at run-time may be good predictors of near future behavior. We focus on the dynamic scheduling of master-worker applications.

We propose an approach for increasing system utilization in a cluster environment by using application-level agents which negotiate with a resource manager for an appropriate level of resource allocation. Agents try to allocate and schedule the tasks of a given master-worker application by following five main criteria: 1) dynamically measure application performance and task execution times 2) predict the resource requirements from measured history, 3) schedule tasks on the resources according to that prediction in order to minimize the completion time of the application, 4) voluntary relinquish resources when they are not plentifully utilized by the application, and 5) allocate more resources whenever a significant loss in speedup is detected.

We have designed and tested a scheduling agent for iterative master-worker applications that allows adaptive and reliable management and scheduling of the application running in a cluster environment. We have experimentally evaluated the effectiveness of our scheduling strategy using an image thinning application.

The rest of the paper is organized as follows. In section 2 we present the background and the parameters considered in our problem. Section 3 outlines our adaptive scheduling strategy for master-worker applications. In section 4 we show some experimental data obtained when the proposed scheduling strategy was applied to a thinning application. In section 5 we survey some related work and section 6 summarises the main conclusions of this work.

2 Problem Motivation and Background

We focused on the study of dynamic scheduling strategies for parallel applications that fit the iterative Master-Worker paradigm running onto a distributed cluster of machines. This model has been used to solve a significant number of problems such as Monte Carlo simulations [2] or material science simulations [9]. A Master-Worker application 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 receive a message from the master with the next task, process the task, and send back the result to the master. The master process may carry out some computations while tasks of a given batch are being 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 (after K cycles or *iterations*).

When a Master-Worker application is running on a distributed cluster of machines, one of the machines will be occupied by the master process and one worker process

will be running in each one of the other available machines. This means that using as many workers as possible is a natural way to reduce the computation times of a given Master-Worker application. With this allocation scheme we would expect that the larger the number of workers assigned to the application, the better the speedup achieved (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). However, most applications exhibit a temporal pattern in their individual tasks that implies that not all the allocated workers can be kept usefully busy. As a consequence, efficiency, 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, will be low. For these applications, it is important to choose a processor allocation carefully so that under-utilized processors are released back to the system.

Releasing under-utilized processors could be beneficial both for the whole system and for the particular user. From the system perspective, released processors could be allocated to other users which, in turn, will improve the overall throughput of the cluster. A particular user will also benefit because cluster job managers normally make use of priority and aging mechanisms in their allocation policies. Every user has a priority and the job manager uses that priority to directly decide how many resources are going to be allocated to him. The better the priority, the more resources the user will get. The aging mechanism assigns a lower priority to a user when he has already been allocated resources for a long time. This mechanism will ensure that the resources will be fairly allocated to all users through time. Therefore, the priority of a user for allocating resources will be more negatively affected when his applications are running on a set of under-utilized resources.

In a previous work, we evaluated several scheduling policies for applications that followed the Master-Worker programming model mentioned above [6]. All the strategies assigned tasks to machines in decreasing order of execution time. The evaluated strategies differ in the amount of precise knowledge that they have about the expected execution time of the tasks.

We evaluated the scheduling policies using different workload distributions and, in general, our results showed that for any given workload distribution, a similar scenario is found. We observed that for any application there is an interval in the number of machines that corresponds to the situation in which the application is using an *ideal* number of workers. Efficiency is high and speedup is also high. All the workers are doing useful work and the application is close to its maximum parallelism utilization. The use of a number of machines belonging to the *ideal interval* guarantees that the largest tasks of the batch are executed alone in a single machine each (or together with some small tasks), and small tasks are executed together sharing some other machines. Using a number of machines belonging to this interval guarantees, in general, a good ratio between execution time and efficiency.

3 Self-Adjusting Scheduling Strategy

The facts mentioned above were used in [7] to design an early version of a self-adjusting algorithm that was responsible for both assigning tasks to workers and determining the number of workers to be allocated. Tasks were sorted in decreasing

order of average execution time. This sorting criterion succeeded in obtaining good performance even if task execution times exhibit significant variations from one iteration to another. At each iteration, they were assigned to workers according to the sorted list. The number of allocated workers was adjusted dynamically at run-time by analysing the particular workload of a given application and determining the appropriate number of workers according to an empirical table. This table categorized applications according to the distribution of task execution times, and provided the number of machines that should be allocated to them. There were two main drawbacks with that strategy. First, the computation cost incurred at runtime to evaluate the workload distribution exhibited by an application in order to determine the appropriate table entry. Second, the sensitivity of the method to small variations in task execution times in successive iterations. This problem resulted in scenarios in which some machines were released and immediately reclaimed back because the workload of the application was oscillating between two table entries.

Our new strategy, presented in next subsection, tries to overcome the problems related to the allocation of workers by, on the one hand, being more conservative in releasing machines and, on the other hand, trying to approach the "ideal" number of processors in a more gentle way once the application runs with a number of machines close to the upper limit of the ideal interval. The assignment of tasks to workers has not changed from our previous work.

3.1 Description of the Self-Adjusting Scheduling Algorithm

Initially as many workers as tasks per iteration (*N*) are allocated for the application. Later, at the end of each iteration, the Self-Adjusting algorithm (shown in figure 3.1) computes the number of workers (*Nworkers*) that should be allocated to the application using two main criteria:

1. First the *AdjustBySpeedup* function computes *Nworkers* by evaluating *asp* (achievable speedup), defined as the ratio between the execution time of the whole application (by adding all the time tasks) and the execution time of the largest task (*ItMaxTaskExecTime*) obtained in the last iteration. From our theoretical studies we know that the upper limit of the ideal number of machines is *asp*. Therefore, the number of workers (*Nworkers*) is set to !a*sp* + 1. This procedure is always used when the application has not allocated all the machines requested in a previous iteration. It is possible that the requirement of workers has changed from one iteration to the next one. Therefore, *asp* is recomputed to check whether the previous requirement of workers is still valid or not.
2. When the application is running with the number of workers previously computed in *Nworkers*, the adjusting criterion to update *Nworkers* is based on two metrics: the execution time (*ItExecutionTime*) and efficiency (*ItEfficiency*) obtained in the last iteration. If the execution time is greater than the execution time of the largest task plus a given threshold, then one more worker is allocated. We have fixed the threshold as being the maximum between the time of the smallest tasks (*ItMinTaskExecTime*) and 15% of the largest tasks. This threshold was fixed empirically as it proved able to detect most of the situations in which the application is not exploiting all its parallelism due to lack of workers, and it does

not yield unstable situations in which workers are claimed and released too frequently. When the second metric is applied, a machine is released when efficiency is smaller than 0.8.

It is important to point out that the criteria described in point 2 above are applied only when the application runs during a whole iteration with a stable number of machines. In this way we do not consider metrics obtained under unstable situations, in which a new machine that was requested previously is allocated in the middle of an iteration and used for executing pending tasks. This situation may produce a temporarily contradictory result in the efficiency or in the execution time metrics. This refinement is not shown in figure 3.1 for the sake of simplicity.

In our experimental system the number of machines is handled cumulatively. This means that when *Nworkers* machines are requested and the application already has allocated *CurrentNworkers* machines, if (*Nworkers* > *CurrentNworkers*), only *Nworkers* - *CurrentNworkers* machines will be added to the application. Otherwise, *CurrentNworkers* - *Nworkers* machines will be released.

```
1. In the first iteration Nworkers = Ntasks
For next iterations (While convergence condition is not met) {
2. Compute ItEfficiency, ItExecutionTime, ItMinTaskExecTime, ItMaxTaskExecTime,
           CurrentNworkers.

3. if (CurrentNworkers < Nworkers ) // We have not got the number of workers needed
       Nworkers = AdjustBySpeedup()
   else
       if (ItExecutionTime > (ItMaxTaskExecTime +
                              MAX (ItMinTaskExecTime, 15%(ItMaxTaskExecTime))))
           Nworkers = Nworkers + 1
       else
           if (ItEfficiency < 0.8)
               Nworkers = Nworkers – 1
}
```

Figure 3.1. Algorithm to determine *Nworkers*

Our self-adjusting algorithm is based on two main assumptions: application parallelism will not exhibit drastic increases over time, and the value of a*sp* obtained after the first iteration will not change significantly in the near future. None of these assumptions were violated in our experiments. However, there are simple extensions that can be included in our basic algorithm to deal with scenarios in which the above-mentioned assumptions were not valid.

3.2 Implementation

We have included our adaptive algorithm into the MW [4] middleware library to experimentally evaluate its performance. MW is a runtime library that allows quick and easy development of master-worker computations on a distributed cluster of

machines. It handles the communication between master and workers using PVM, and performs processor allocation and fault-detection through the services provided by a Condor job manager [8]. An application in MW has three base components: a Driver that is the master and manages a set of user-defined tasks and a pool of workers; and the Workers that execute the Tasks. We have extended MW to support both the iterative master-worker paradigm and the self-adjusting scheduling algorithm.

4 Experimental Study

In this section, we report the results obtained with the aim of testing the effectiveness of the proposed adaptive scheduling algorithm. We have executed an image thinning application. We run the applications on a distributed cluster 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 application.

4.1 Thinning Application

Our thinning algorithm for binary images was adapted from the AFP3 (Fully Parallel Algorithm) described in [5]. The application works in the following way. Initially the image is divided into M horizontal parts. Each part contains the pixels of a piece of the image, plus border pixels from neighboring parts. One task is created to compute the thinning operation of one part, which basically consists of deleting pixels. At the end of each iteration, workers send the image back to the master, which updates the border pixels. If there are no more pixels to delete, the part achieves the local convergence criterion and finishes. When all the parts have finished then the global convergence criterion is met, the skeleton image is reconstructed combining the parts in order, and the application finishes. Figure 4.1 shows an original image and the result obtained by the thinning algorithm.

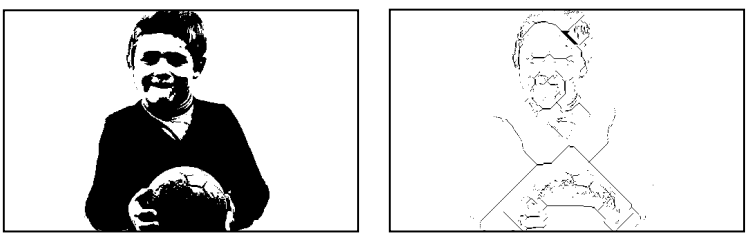

Figure 4.1. Reference image *Boy&Ball* and its thinning result

This application exhibits two characteristics that make its use attractive for evaluating a self-adjusting strategy. First, tasks corresponding to different parts of the application usually exhibit different execution times. Tasks that are assigned complex part of the image spend more time than tasks that deal with simple parts of the image. Therefore, a self-adjusting strategy must be able to schedule together short tasks to the same worker and relinquish spare workers. Secondly, the execution time of each

task gradually decreases as the image thinning approaches convergence. Again, the self-adjusting strategy must also be able to reduce the number of workers as the execution time of converging tasks is close to zero.

4.2 Experimental Results

We conducted experiments using a distributed cluster platform consisting of a Condor pool of machines at the University of Wisconsin. The total number of available machines was around 700 although we restricted our experiments to machines with Linux architecture. 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.

We ran the thinning application with 3 images: *Figures*, *Letters* and *Boy&Ball*. Images were initially divided into 8, 16 and 32 parts, which corresponded to the initial set of tasks created at the initial iteration. The number of iterations until thinning convergence was 92, 105 and 97 for the three images, respectively. We enlarged the size of the images so that the execution time of the largest task was initially in the range of 50 seconds when images were divided into 8 parts.

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

Results of efficiency and execution time (in seconds) are shown in table 4.1 when the thinning application was run both using our self-adjusting strategy (Self-Adjusting column) and without using it (No Self-Adjusting column). When no adaptive scheduling was used, the initial number of requested workers was equal to the initial number of tasks. Once a task met the convergence criterion, the corresponding worker was released. In contrast, in the self-adjusting case, workers were released only according to our strategy and no workers were released automatically on task completion. Tasks were assigned to workers in decreasing order of average execution time in both Self- and Non Self-Adjusting cases. Therefore, our results reflect mainly the effectiveness of our strategy to dynamically adjust the number of resources.

In addition to the results obtained for both strategies using an initial number of tasks of 8, 16 and 32, we also include the execution time of a sequential thinning application (column *InitialTasks* = 1) for comparison purposes. In the *NworkersAvg* rows the average of the number of workers used are shown.

As can be seen in table 4.1, self-adjusting obtains efficiency values above 0.8 in all cases, while no self-adjusting obtains efficiency values that are significantly smaller (between 0.4 and 0.65 in most cases). The execution time results indicate that the self-adapting strategy results in a penalty that in most cases is less than 15% compared to the non self-adjusting case. Only for the *Letters* example with 16 and 32 tasks, was the difference in execution time 17% and 19%, respectively. In general, the execution time of the application does not decrease linearly as the image is decomposed in more parts because the maximum parallelism is only achievable at the initial iterations of the algorithm. Later, as different parts of the image converge, parallelism decays and consists only of the tasks that compute the most complex parts of the images.

Table 4.1. Results of the master-worker thinning application

			Non Self-Adjusting			Self-Adjusting		
InitialTasks		**1**	**8**	**16**	**32**	**8**	**16**	**32**
Nworkers Avg.	Figures	1	5,5	9,12	12,85	2,45	4,17	7,37
	Letters	1	5,3	10,36	21,11	3,89	6,55	9,41
	Boy&Ball	1	5,57	7,02	11,34	2,85	4,01	8,92
Efficiency	Figures	1	0,41	0,41	0,48	0,88	0,89	0,86
	Letters	1	0,64	0,59	0,399	0,8	0,82	0,83
	Boy&Ball	1	0,59	0,64	0,7	0,88	0,86	0,87
Exec. Time (in seconds)	Figures	12746	4141	2634	1533	4473	2648	1703
	Letters	12803	3179	1562	1204	3230	1833	1399
	Boy&Ball	10080	2948	1678	1001	3094	1732	1002
Exec.Time/ Efficiency Ratio	Figures	12746	10100	6424,39	3193,75	5082,95	2975,28	1980,23
	Letters	12803	4967,18	2647,45	3010	4037,5	2234,36	1685,54
	Boy&Ball	10080	4996,61	2621,87	1430	3515,9	2013,95	1151,72

As a global index of performance, the last three rows of table 4.1 show the index between execution time and efficiency corresponding to both strategies. The lower the index, the better the use of resources achieved by a given strategy. This means that our strategy achieves a better trade-off between efficiency and execution time.

Although 8, 16 and 32 workers were claimed initially by both strategies, a smaller number of workers were effectively allocated throughout the computation. The Non Self-Adjusting strategy simply relinquished workers as tasks were completed. Our Self-Adjusting strategy further reduced the number of allocated workers, as can be seen in the *Nworkers Avg.* row which contains the average number of workers used from the beginning to the end of the computation. In general, our strategy saved between 20% to 55% of workers compared to the Non Self-Adjusting case.

Figure 4.2 shows a detailed example of one execution of the thinning application applied to the *Figures* image divided initially into 32 parts. This example is a representative illustration of the general behavior and the performance achieved by both the Self-Adjusting and the Non Self-Adjusting algorithms. We show the information related to number of workers, efficiency and execution time after iterations 1, 5, 10, 15, and so on. Execution times are shown in a logarithmic scale.

As can be seen, the allocation of resources is not serviced immediately after request. This implies, for instance, that the Non Self-Adjusting algorithm achieves a maximum number of 23 workers in iteration 15. At this time, some of the tasks have already finished (those corresponding to image borders) and, therefore, the application does not need the whole set of 32 workers requested at the beginning. In general, the Self-Adjusting algorithm is able to tune the number of workers from the initial iterations, fixing the maximum number of workers to 15 after iteration 10. Significant differences in the number of workers (and, consequently, in efficiency) are mainly observed at the central iterations of the computation (from iteration 15 to 75). In these stages, the execution time of each iteration is slightly better for the Non Self-Adjusting algorithm at the expense of sometimes using twice the number of workers that the Self-Adapting strategy uses. Later, the application is close to the end and the

number of workers is very small in both cases, so efficiency and execution time are very similar for both strategies.

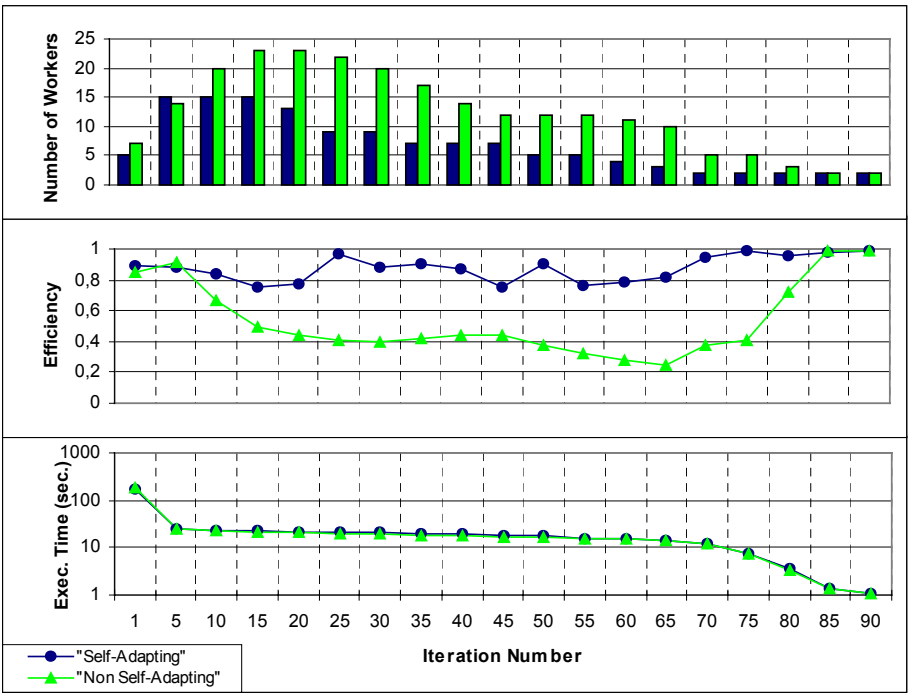

Figure 4.2. Number of workers, efficiency and execution time obtained with the *Figures* image divided in 32 parts

5 Related Work

The problem of self-adaptive scheduling has been investigated recently in different frameworks. There are several middleware environments that allow the development of adaptive parallel applications running on distributed clusters. They include NetSolve [3], Nimrod [1] and AppLeS [10]. NetSolve and Nimrod provide API 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, and their adaptive algorithms rely on different metrics to the ones adopted here. In AppLeS, the application programmer is supplied information about the computing environment and is given a library to allow them to react to changes in available resources.

6 Conclusions

In this paper we have discussed the problem of scheduling master-worker applications on distributed cluster environments. We have presented a self-adjusting strategy that takes into account runtime information about the application. This information is used to allocate and schedule the minimum number of processors that guarantees 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 is rather straightforward at the moment and is not guaranteed to adjust the number of workers to the optimal in all cases. However, our early experimental results with a thinning application running in a homogeneous cluster of machines are encouraging, as they have shown that our algorithm worked well in practice. In general, our adaptive strategy achieved an efficiency higher than 80% in the use of processors, while the execution time was only slightly worse than the execution time achieved with a significantly larger number of processors.

References

1. 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,
2. J. Basney, B. Raman and M. Livny, "High throughput Monte Carlo", Proceedings of the Ninth SIAM Conf. on Par. Proc. for Scientific Computing, San Antonio Texas, 1999.
3. 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.
4. J.-P. Goux, S. Kulkarni, J. Linderoth, M. Yoder, "An enabling framework for master-worker applications on the computational grid", Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9), pp. 43-50 , 2000.
5. Z. Guo and R. Hall. "Fast Fully Parallel Thinning Algorithms". CVGIP: Image Understanding. Vol. 55, No. 3, pp. 317-328, May 1992.
6. E. Heymann, M. A. Senar, E. Luque and M. Livny, "Evaluation of an Adaptive Scheduling Strategy for Master-Worker Applications on Clusters of Workstations", Proc. of 7th Int. Conf. in High Perf Comp. (HiPC 2000), LNCS series, Vol. 1970, pp. 310-319, 2000.
7. E. Heymann, M. A. Senar, E. Luque and M. Livny, "Adaptive Scheduling for Master-Worker Applications on the Computational Grid", Proc. of 2000 Int. Workshop on Grid Computing (GRID'2000), LNCS series, Vol. 1971, pp. 214-227, 2000.
8. M. Livny, J. Basney, R. Raman and T. Tannenbaum, "Mechanisms for high throughput computing", SPEEDUP, 11, 1997.
9. 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.
10. G. Shao, R. Wolski and F. Berman, "Performance effects of scheduling strategies for Master/Slave distributed applications", Tech. Rep. TR-CS98-598, University of California, San Diego, September 1998.
11. 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.