Using On-The-Fly Simulation For Estimating the Turnaround Time on Non-Dedicated Clusters*

Mauricio Hanzich[2], Josep L. Lérida[1], Matías Torchinsky[2], Francesc Giné[1], Porfidio Hernández[2] and Emilio Luque[2]

[1] Dept. Computer Science, University of Lleida, Spain.
{sisco,jlerida}@diei.udl.es
[2] Dept. Computer Architecture and Operating Systems, University Autònoma of Barcelona, Spain. {porfidio.hernandez,emilio.luque}@uab.es, {mauricio,matias}@aomail.uab.es

Abstract. The computation capacity of the workstations of an open laboratory in almost every university is enough to execute not only the local workload but some distributed computation. Unfortunately, the local workload introduces a big uncertainty into the predictability of the system, which hinders the applicability of the job scheduling strategies. In this work, we introduce into our job scheduling system, termed CISNE, a simulator, which allows its scheduling decisions to be enhanced by estimating the future cluster state. This process of estimation is backed by analytic procedures which are also described in this study. Likewise, the simulation let us assure some limit to the turnaround time for the parallel user. This paper analyses the performance of the simulation process in relation to different scheduling policies. These results reveal that those policies that respect an FCFS order for the waiting jobs are more predictable than those that alter the job ordering, like Backfilling.

1 Introduction

Several studies [1] have revealed that a high percentage of computing resources (CPU and memory) in a Network Of Workstations (NOW/Cluster) are idle. The possibility of using this computing power to execute distributed applications with a performance equivalent to a Massively Parallel Processor (MPP) and without perturbing the performance of the local users applications on each workstation has led to a proposal for new resource management environments [8,5].

With the aim of taking advantage of these idle computing resources (CPU and memory) available across the cluster, we have developed a new scheduling environment, named CISNE [5], which combines space sharing and time sharing scheduling techniques. The space sharing scheduling component of CISNE is a job scheduler, named LoRaS (Long Range Scheduler). When a parallel job is submitted to the LoRas, the job waits in a queue until it is scheduled and executed. Thus, LoRaS must deal with the *Job Selection* process from a waiting

* This work was supported by the MEyC-Spain under contract TIN 2004-03388.

queue, together with the problem of selecting the best set of nodes for executing a job (*Node Selection* policies). This is performed by taking into account the state of the cluster system together with the characteristics of the local and parallel workload. Based on those considerations, different policies are implemented to assign jobs to processors in the LoRaS system.

Once a parallel job is executed, the time sharing component of CISNE, named CCS (Cooperating CoScheduling) [3], takes control of the progression of each parallel job. CCS provides an execution environment where the parallel applications can be dynamically coscheduled. It means that the tasks belonging to the same parallel job are coscheduled according to its communication requirements [12]. In addition, the resources given to parallel tasks are balanced and the interactive responsiveness of the local applications (local workload) is fully preserved by means of a job interaction mechanism, and even when using a MultiProgramming Level of parallel jobs (MPL_{paral}) greater than one [4].

In order to take better scheduling decisions, the CISNE system needs to forecast the future state of the cluster [10]. Likewise, this prediction capacity could help to guarantee some limit in the turnaround time for the applications of any parallel user.

These considerations have stimulated some works focused on the estimation process. The most evaluated alternative is to use a historical system that records the past executions of an application [6,2]. This kind of system normally looks for a state that is similar to the current one by defining a comparison function that determines how similar one state is to another. This focus is valid when the state is defined by a small set of variables, but in our case we have to consider a more complex cluster state (i.e. more variables), due to the non-dedicated characteristic of the environment. Besides, a historical system needs a learning phase to become accurate and, the larger the set of variables to consider, the longer the time needed to achieve precision. Following studies like [11,7], we decided to use a simulation approach to represent our scheduling system. However, in such studies the authors use a historical system for predicting the execution time of each parallel application, while we focus on an analytic schema. The results obtained from those studies can reach an error of around 37% for the execution time prediction of any single application. Our analytic results give an error of around 41%, but with an estimation time that remains constant whatever the cluster state, while a historical system needs a linear time (depending on the number of cases to be studied), for estimating the same value. It should be noted that the number of cases to be evaluated by the historical system in a non-dedicated environment is much greater than those available in studies such as [11,7].

Unfortunately, different policies for distributing the parallel applications may have different effects on the estimation methods and vice-versa. This could make some predicting schemes more suitable for some specific policy but not for others. This has motivated us to propose some estimation methods oriented to non-dedicated clusters and evaluate their performance in relation to different distribution policies. With this aim, a simulation tool has been implemented in

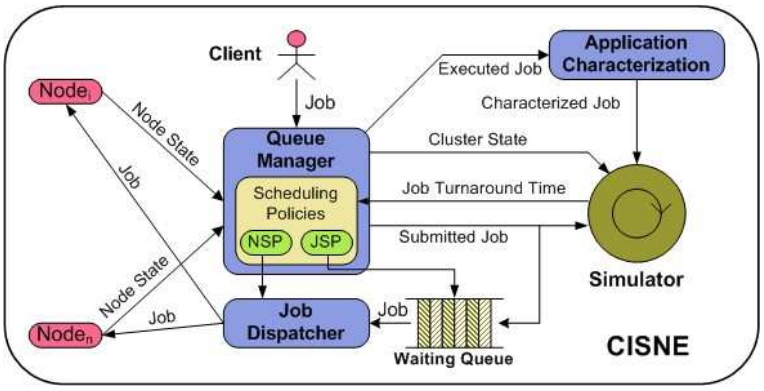

Fig. 1. The integration of the simulation into the CISNE system.

the CISNE system. In this framework, we have observed that those policies that distribute the resources in a more balanced way are more predictable reaching an accuracy of 12%. On the other hand, those policies that could alter the order of the waiting jobs, such as backfilling, are inherently more inaccurate.

The outline of this work is as follows, section 2 depicts our simulation method. Some proposals for the estimation process are explained in section 3. Next, the experimental results are analyzed in section 4. Finally, some conclusions and the future work are explained.

2 The Simulation Process on the CISNE System

As stated above, the simulation process is integrated into the CISNE system [5]. In order to deal with this estimation in a non-dedicated environment, CISNE needs information from two different inputs: the characterization of the parallel applications and the modelling of the current cluster state, including the local load activity. As we can see in Fig. 1, this information is provided by the queue manager and application characterization block, respectively. At the end of the simulation, the estimated job turnaround time is returned to the queue manager.

The behavior of the parallel applications is obtained by means of running the parallel application in isolation. This is preferred over the information given by the user about the resources used by the applications, which is normally an inaccurate method [9]. For a fixed number of processors per application (n), CISNE collects:

- $ExeTime_{tot}(J)$: execution time of the job J.
- $CPUtime_{tot}(J)$: amount of CPU time used by the job J.
- $CPU(J)$: CPU percentage ($CPUtime_{tot}(J)/ExeTime_{tot}(J)$) used by the job J.

The queue manager collects the usage of the resources in each node together with the state of each application. The following set of data models the cluster state:

- *JSP and NSP* policies: Job and Node Selection Policies used by CISNE, respectively.
- $ExeTime_{cur}(J)$: current running time for the job J.
- $CPUtime_{cur}(J)$: amount of CPU time used by the job J from its beginning.
- $nodes(J)$: set of nodes where the job J is running.
- $CPU_{local/paral}(n)$: sum of the CPU usage of each local/parallel task running in the node n.
- $MPL_{paral/local}(n)$: number of parallel/local tasks executing simultaneously in the node n. As we demonstrated in several previous studies [5,4], the time sharing component of CISNE allows the execution of more than one parallel application in the same set of nodes, whenever it does not disturb the local user.
- $tasks(n)$: set of parallel tasks running in the node n.

Once all the needed elements are collected, CISNE is ready to start the simulation process described in the next section.

3 Turnaround Time Prediction by Simulation

The simulation process is triggered whenever a new application arrives to the CISNE system. Every time that the simulation is started, the turnaround time for each application, either running or waiting, is estimated and adjusted, giving some extra information to the job scheduler about the future cluster state. If the simulator is working when a new application arrives, the whole process has to be restarted considering the new job to be executed. Alg. 1 depicts our simulation method.

The core of the simulation algorithm relies on a *while* that loops as long as any parallel application is running (Alg. 1:3-16). For each iteration, the algorithm estimates the *Remaining Execution Time (RemainTime)* of every job in the running queue (*DRQ*), selects the next job that will finish (J_i) and removes it from *DRQ* (Alg. 1:4-6). After that, the $CPUtime_{cur}$ used by each of the remaining jobs in *DRQ* is calculated (Alg. 1:7) to be used in the next simulation step (Alg. 1:3 loop). Next, another loop tries to execute some waiting jobs using the system scheduling policy, the available resources and those resources released by J_i (Alg. 1:8). Finally, the waiting time for every job in *DQ* is updated (Alg. 1:14) and the simulation step advances to t_i(Alg. 1:15).

In order to carry out this simulation, we need a pair of extra functions which define the estimation process. The first is the *RemainTime* (Alg. 1:4), which estimates the *remaining execution time* for a given application considering the current cluster and application state. The second tries to predict the *CPU time* ($CPUtime_{cur}$) that the application has used in the past (Alg. 1:7). Our approaches to solving both functions are depicted in the following subsections.

Algorithm 1 Simulation process

1: Duplicate the system state in a *dummy* system state: DQ as a copy of the jobs waiting queue, DRQ as a copy of the running jobs queue and CL_{sim} as a copy of the cluster nodes with their state
2: Store the current time (t_0), as the moment when the simulation has begun.
3: **while** ($\exists$ J in DRQ) **do**
4: **forall** (J in DRQ) Calculate the $RemainTime(J)$.
5: Assume that the application J_i is the next one to finish in time t_i.
6: Update the estimated $ExeTime_{tot}(J_i)$ to t_i and remove J_i from DRQ.
7: **forall** (J in DRQ) Calculate the $CPUtime_{cur}(J)$ in $[t_0, t_i]$.
8: **while** ($\exists$ usable resources in Cl_{sim} and any job waiting in DQ) **do**
9: Look for an application J_x in DQ that could be executed in the Cl_{sim} state.
10: Select the best subset of Cl_{sim} for executing J_x, using the system policy.
11: Execute the application J_x in the selected subset of Cl_{sim} and add it to DRQ.
12: Increment the estimated $WaitingTime(J_x)$ in $[t_0, t_i]$.
13: **end while**
14: **forall** (J in DQ) Increment the estimated $WaitingTime(J)$ in $[t_0, t_i]$.
15: Set t_0 to t_i.
16: **end while**

3.1 *Remaining Execution Time* approaches

The easiest estimation is to think that the future will be similar to the past. With this in mind, the remaining execution time of a job J, denoted as $RemainTime(J)$, is calculated according to the following equation:

$$RemainTime(J) = \frac{ExeTime_{cur}(J) \times (CPUtime_{tot}(J) - CPUtime_{cur}(J))}{CPUtime_{cur}(J)} \quad (1)$$

Note that Eq.1 assumes that the CPU time ($CPUtime_{tot}(J)$) used by the job J during its complete execution ($RemainTime(J) + ExeTime_{cur}(J)$) is proportional to the CPU time ($CPUtime_{cur}(J)$) used during the current execution time ($ExeTime_{cur}(J)$).

The second proposal considers both the past and current states. It starts by calculating the remaining execution time that the job J would need if it were executed in isolation ($RemainTime_{isol}(J)$). This value, following the same reasoning as Eq. 1, is calculated as follows:

$$RemainTime_{isol}(J) = \frac{ExeTime_{tot}(J) \times (CPUtime_{tot}(J) - CPUtime_{cur}(J))}{CPUtime_{tot}(J)} \quad (2)$$

Next, the maximum MPL ($MPL_{max}(J)$) is defined as:

$$MPL_{max}(J) = max(MPL_{paral}(n) + MPL_{local}(n) \mid n \in nodes(J)). \quad (3)$$

It is worth pointing out that Eq. 3 returns the maximum number of tasks, both local ($MPL_{local}(n)$) and parallel ($MPL_{paral}(n)$), executing concurrently with the job J among the nodes where it is running ($nodes(J)$). Taking $MPL_{max}(J)$ and $RemainTime_{isol}(J)$ into account, the remaining execution time of job J is calculated according to the following equation:

$$RemainTime(J) = RemainTime_{isol}(J) \times MPL_{max}(J) \tag{4}$$

Our last approach considers not only the number of tasks executing concurrently (MPL) but also the *CPU requirements* of those tasks (in percentage). According to this, the $RemainTime(J)$ is calculated as follows:

$$RemainTime(J) = RemainTime_{isol}(J) \times \frac{CPU(J)}{CPU_{feas}(J)} \tag{5}$$

where:

$$CPU_{feas}(J) = min(CPU(J), \frac{CPU(J)}{CPU_{max}(J)}), \tag{6}$$

is the maximum CPU usage (in percentage) that we expect the job J could use, and:

$$CPU_{max}(J) = max(CPU_{paral}(n) + CPU_{local}(n) \mid n \in nodes(J)) \tag{7}$$

is the maximum CPU usage requirements (in percentage) among the nodes where the job J is running ($nodes(J)$).

It is important to emphasize that no matter which the chosen approach is, the value for $CPUtime_{cur}(J)$ is accurate only at the beginning of the simulation process (Alg. 1:3), but for each simulation step it is necessary to estimate this value again (Alg. 1:7). Therefore, in the next subsection we describe some proposals for estimating this value.

3.2 Used *CPU time* proposals

This section describes two different proposals for estimating the current CPU time for a given job J at a specific moment (t_i), denoted as $CPUtime_{cur}(J, t_i)$, considering that this value has been measured in the past ($CPUtime_{cur}(J, t_{i-1})$).

In our first approach, we assume that the application CPU usage is proportional to the maximum MPL calculated in Eq. 3. The following expression represents this proposal.

$$CPUtime_{cur}(J, t_i) = CPUtime_{cur}(J, t_{i-1}) + \frac{(t_i - t_{i-1}) \times CPUtime_{tot}(J)}{MPL_{max}(J) \times ExeTime_{tot}(J)} \tag{8}$$

Our second proposal is based on the same idea used for the remaining time in Eq. 5, but applied to the *CPUtime*. The following equation represents that idea.

$$CPUtime_{cur}(J, t_i) = CPUtime_{cur}(J, t_{i-1}) + (t_i - t_{i-1}) \times CPU_{feas}(J) \quad (9)$$

4 Experimentation

In order to carry out the experimentation process, we need two different kinds of workload. On one hand, we need to simulate the local user activity and, on the other hand, we need some parallel applications that arrive at some interval.

The local user activity is represented by a benchmark that could be parameterized in such a way that it uses a percentage of CPU, memory and network. To parameterize this benchmark realistically, we measure our open laboratories for a couple of weeks and use the collected values to run the benchmark (15% CPU, 35% Mem., 0,5KB/sec LAN).

The parallel workload was a list of 30 PVM NAS parallel jobs (CG, IS, MG, BT) with a size of 2, 4 or 8 tasks that entered the system following a Poisson distribution of inter-arrival times with mean=$15s$. These jobs were merged so that the entire workload had a balanced requirement for computation and communication. It is important to mention that the MPL_{paral} reached for the workload depends on the system state at each moment, but in no case will surpass an $MPL_{paral} = 4$ [4].

This workload was executed with different combination of Job Selection (JSP) and Node Selection policies (NSP). Regarding the JSP policies, *FCFS* (First-Come-First-Served) and Backfilling techniques were tested. A backfilling policy consists of executing a job, not at-the-head of the FCFS queue, whenever this does not delay the start of the job at the head. The set of nodes onto which the selected job will be launched, was chosen according to two different *NSP* policies. The first one, termed *Normal,* selects the nodes for executing a parallel application considering only the resource usage level throughout the cluster, so it does not overload any node in detriment of the local user interactiveness. An example can be observed in Fig. 2.b, where the J_3 shares its nodes with $J_1$and J_2. The second approach, called *Uniform,* selects the nodes respecting not only the resource usage but also the job distribution. In this case, the policy executes a pair of jobs of the same size in the same set of nodes whenever possible. Besides, the system tries to execute tasks of different oriented applications (i.e. communication bound vs. computation bound), in the same set of nodes trying to enhance the underlying Time-Sharing schema of our CISNE system [5]. An example can be seen in Fig. 2.a, where J_3 only shares its nodes with J_2. Finally, and for the purpose of comparison, we include a *Basic* policy made up of a *Normal+FCFS* policy with the $MPL_{paral} = 1$, rather than 4 as in the other policies.

The whole system was evaluated in an Linux cluster using 16 P-IV (1,8GHz) nodes with 512MB of memory and a fast Ethernet interconnection network.

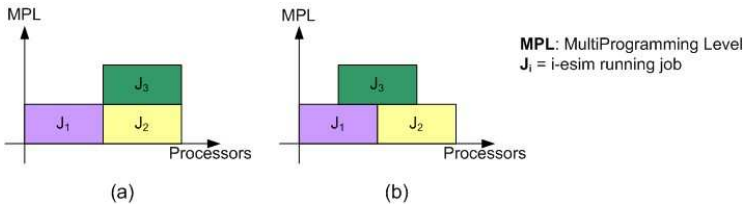

Fig. 2. *Uniform* (a) and *Normal* (b) Node Selection Policy.

4.1 Experimental Results

In this subsection, we present some results showing the effect of the different JSP and NSP policies over the different mixes for the *Remaining Execution Time* (*RemainTime*) and the *Used CPU Time* (*CPUtime*) estimation methods. Fig. 3 shows the estimated turnaround deviation (in %) from the real turnaround time for the different policies and estimation methods. These results were obtained considering that 25% of the nodes had some kind of local activity.

From Fig. 3 and considering the *RemainTime* methods, we can see that a *Uniform* policy favors the predictability of the system over a *Normal* policy because it tries to balance the resources given to the parallel applications. In such a case, the tasks forming a parallel application have the same resources and then they can evolve jointly letting the estimation methods be more accurate. Likewise, Fig. 3 shows that a Basic policy performs worse than *Normal* or *Uniform* for some cases. This is mainly due to the *Basic* restriction of the MPL_{paral} (MPL_{paral}=1). In such a scenario, the waiting queue length increases and hence the waiting time prediction become less accurate. As a consequence, the turnaround time prediction is worse. In addition, this figure reflects that a scheduling policy that includes a backfilling scheme is more unpredictable. This is due to the difficulty of tracking the variation in the order of the elements in the waiting queue. However, and even considering that the resulting estimations are not as good as for those policies without a backfilling scheme, the results are almost always pessimistic, due to the possibility of backfilling some of them, and hence reducing their waiting time. This means that the parallel user has a turnaround time that in the worst case, is overestimated, but never underestimated. Finally, and as was to be expected, the *Proportional* (Prop., Eq. 1) *RemainTime* method performs badly. This is due to the assumption that the future behaves like the past. This assumption is not true when the environment state changes continuously due to the local and parallel loads.

From the same Fig. 3, and focusing now on the *Used CPU Time* methods, it is clear that the estimation via the *CPU* usage (right columns, Eq. 9) is more reliable in most of the cases, compared with the estimation through the *MPL* (left columns, Eq. 8) method. This happens because the *CPUtime* method represents reality better by considering the real percentage of CPU consumed by each task, while the *MPL* method assumes that every task consumes the same

percentage of CPU. There is a special case for the *Basic* policy, where these results are apparently contradictory. However, these results are due to the light parallel load imposed on the system (on Basic, the parallel MPL is at most 1), that results in a longer, and hence more unpredictable, waiting queue.

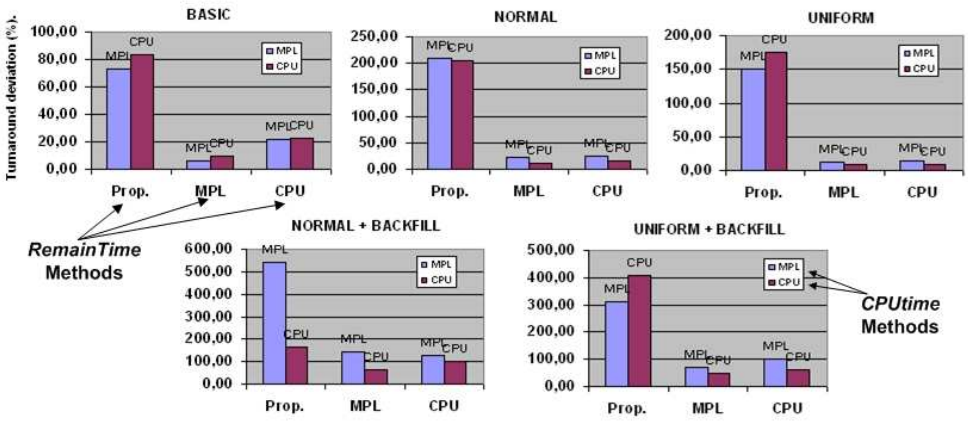

Fig. 3. Turnaround deviation (%) for the different *RemainTime* and *CPUtime* mixes for the scheduling policies evaluated.

In addition, we want to analyze the influence of the local load on the estimation methods and scheduling policies. Table 1 shows the turnaround deviation for two different combinations to estimate the *RemainTime* and $CPUtime$ values respectively: *MPL-MPL* (MPL in table 1) and *CPU-CPU* (CPU in table 1). Both combinations were tested varying the number of nodes with local load from 25% to 100% of nodes. From this table, we can see that the CPU-CPU estimation method gives us a better estimate than the MPL-MPL method in most of the cases. This agrees with our expectations, because an estimation process that considers the CPU consumption of each task instead of assuming that every task uses the same amount of CPU, as the MPL-MPL method does, tends to be more accurate. There are, however, some cases where the CPU-CPU method does not give us the best results. One of them was found for the *Normal* policy and 100% of nodes loaded with local tasks. In this case, these bad results are due to an unbalanced distribution of the resources throughout the cluster, which complicates the CPU-CPU method capacity for tracking the CPU usage of the parallel and local loads. The other case that it is not favorable to the CPU-CPU method is for the *Basic* policy. In this case, when the system is unloaded (local load of 25%) and the MPL is at most 1, the CPU usage calculation is misleading because the whole set of tasks could evolve faster than the estimation process as-

sume. However, in such a situation, the estimation process is always pessimistic, so that the application will always finish before the estimated finish time.

The second effect to note is the increment in the accuracy of the estimation methods for the policies that include *Backfilling* when the local load increases. This might seem contradictory, but is in fact perfectly understandable because when the local load increases, the available resources decrease, as do the opportunities to backfill a job.

	Norm+BF		**Unif+BF**		**Basic**		**Normal**		**Uniform**	
Local Load	MPL	CPU	MPL	CPU	MPL	CPU	MPL	CPU	MPL	CPU
25%	144,33	99,67	69,67	63,33	6,33	23,00	23,00	15,33	12,67	8,67
50%	58,67	59,67	55,00	31,33	14,00	18,67	20,50	21,33	22,00	21,67
100%	46,33	45,00	54,00	46,67	34,33	8,67	16,67	28,33	25,33	18,33

Table1. Turnaround deviation (%) for the different *RemainTime* and *CPUtime* mixes for the evaluated policies and different local loads.

It is worthwhile pointing out that the time cost spent by all our proposals to estimate the turnaround time of a single application is always lower than $4ms$. It means that this is at least two orders of magnitude lower than the execution time of the parallel applications (minutes).

5 Conclusions and Future Work

In order to improve the prediction capacity of a resource management environment over a non-dedicated cluster, this work presents a simulation algorithm that merges different estimation methods for predicting the turnaround time of parallel applications. The proposed estimation methods focus on two different goals. The first set tries to predict the *Remaining Execution Time* for a given running application and a defined cluster state, while the second set of methods estimates the *CPU usage* that an application could absorb for a given time interval and cluster state. The relationship between these estimation methods and different job scheduling policies are evaluated. We conclude that those methods that consider not only the MultiProgramming Level (MPL) of the parallel and local tasks in each node but also the CPU consumption of each one, are more accurate in the general case. Besides, those policies including a backfilling scheme are inherently more difficult to estimate accurately due to the possibility of altering the job ordering in the queue. However, this estimation always tends to be pessimistic and hence the jobs finish before the predicted finish time. Another effect that could be observed was the influence of the job distribution on the estimation process. For those policies that balance the resources it is easier to generate a more accurate estimation, reaching an accuracy of 12%.

In the future, we want to introduce an hybrid system that merges our simulator with a *Remaining Execution Time* method that uses a historical system.

This way it will be possible to generate an estimation method that becomes more and more accurate over time, but without the cost of assuming a whole historical prediction system that has to manage a lot of variables.

References

1. A. Acharya and S. Setia. Availability and utility of idle memory in workstation clusters. In *Proceedings of the ACM SIGM./PERF.'99*, pages 35–46, 1999.
2. Allen B. Downey. Predicting queue times on space-sharing parallel computers. In *IPPS '97: Proceedings of the 11th International Symposium on Parallel Processing*, pages 209–218, Washington, DC, USA, 1997. IEEE Computer Society.
3. F. Giné, F. Solsona, P. Hernández, and E. Luque. Cooperating coscheduling in a non-dedicated cluster. *EuroPar 2003 Parallel Processing, LNCS*, 2790:212–218, 2003.
4. M. Hanzich, F. Giné, P. Hernández, F. Solsona, and E. Luque. Coscheduling and multiprogramming level in a non-dedicated cluster. *EuroPVM/MPI 2004, LNCS*, 3241:327–336, 2004.
5. M. Hanzich, F. Giné, P. Hernández, F. Solsona, and E. Luque. Cisne: A new integral approach for scheduling parallel applications on non-dedicated clusters. *EuroPar 2005, Parallel Processing. LNCS*, 3648:220–230, 2005.
6. Benjamin J. Lafreniere and Angela C. Sodan. Scopred—scalable user-directed performance prediction using complexity modeling and historical data. *Job Scheduling Strategies for Parallel Processing, LNCS*, 3834:62–90, 2005.
7. H. Li, D. Groep, J. Templon, and L. Wolters. Predicting job start times on clusters. *4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2004)*, April 2004.
8. M. Litzkow, M. Livny, and M. Mutka. Condor- a hunter of idle workstations. *8th Int'l Conference of Distributed Computing Systems*, 1988.
9. A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Transaction on Parallel & Distributed Systems*, 12(6):529–543, 2001.
10. W. Smith, V. Taylor, and I. Foster. Using run-time predictions to estimate queue wait times and improve scheduler performance. *Job Scheduling Strategies for Parallel Processing, LNCS*, 1659:202–219, 1999.
11. W. Smith and P. Wong. Resource selection using execution and queue wait time predictions. *NAS Technical Reports*, 2002.
12. P. Sobalvarro, S. Pakin, W. Weihl, and A. Chien. Dynamic coscheduling on workstation clusters. *Job Scheduling Strategies for Parallel Processing, LNCS*, 1459:231–256, 1998.