Improving Parallel Job Scheduling Using Runtime Measurements

Fabricio Alves Barbosa da Silva[1], Isaac D. Scherson[1,2]

[1] Université Pierre et Marie Curie, Laboratoire ASIM, LIP6, Paris, France.
fabricio.silva@lip6.fr
[2] Information and Comp. Science, University of California, Irvine, CA 92717 U.S.A.
isaac@uci.edu

Abstract. We investigate in this paper the use of runtime measurements to improve job scheduling on a parallel machine. Emphasis is on gang scheduling based strategies. With the information gathered at runtime, we define a task classification scheme that is used to provide better service to I/O bound and interactive jobs under gang scheduling through the utilization of idle times due to idle slots and blocked tasks and also by controlling the spinning time of a task as a function of the workload on node. Simulation results are presented and show improvements in both throughput and machine utilization for a gang scheduler using runtime information compared with gang schedulers for which this type of information is not available.

1 Introduction

In this paper we analyze the utilization of runtime information in parallel job scheduling to improve throughput and utilization on a parallel computer. Our objective is to use information such as number of I/O calls, duration of I/O calls, number of messages arrived, number of messages sent, number of barriers, time spent in spinning while waiting for message/synchronization arrival and other information available as a function of the architecture in order to associate a specific task in a given moment of time to one class belonging to a set of predefined classes with the help of fuzzy sets and Bayesian estimators. Observe that the classification of a task may change over time, since we consider, as in [2], that characteristics of jobs may change during execution.

Some possible uses for the task classification information are, for instance, to decide which task to schedule next, to decide what to do in the case of an idle slot in gang scheduling, or to define spinning time of a task as a function of the total workload on a processor. One possible utilization of these concepts is to give better service to I/O bound jobs in gang scheduling, by using task classification to identify I/O bound tasks in order to reschedule them in idle slots or if a gang scheduled task blocks itself. This approach is different from the one proposed in Lee et al. [19] since it does not interrupt running jobs.

In this paper we will give emphasis to gang scheduling based strategies. Gang scheduling can be defined as follows: Given a job composed of N tasks, in

gang scheduling these N tasks compose a process working set[21], and all tasks belonging to this process working set are scheduled simultaneously in different processors, i.e., gang scheduling is the class of algorithms that schedule on the basis of whole process working sets. Gang scheduling allows both the time sharing as well as the space sharing of the machine, and it was originally introduced by Ousterhout[21]. Performance benefits of gang scheduling the set of tasks of a job has been extensively analyzed in [16, 10, 13, 29] Packing schemes for Gang scheduling were analyzed in [9].

In section 2 we discuss some previous work in parallel/distributed job scheduling that considers the use of runtime information to modify scheduling-related parameters at runtime. Section 3 presents the task classification mechanism based on runtime information we use in this paper. How to use this information to improve throughput and utilization in parallel job scheduling through a priority computation mechanism is discussed at section 4. Section 5 discusses the utilization of task classification information to control spin time in order to give better service to I/O bound and interactive jobs in gang scheduling. Our experimental results are presented and discussed in section 6 and section 7 contains our final remarks.

2 Previous Work

In [1], Arpaci-Dusseau, Culler and Mainwaring use information available at run time (in this case the number of incoming messages) to decide if a task should continue to spin or block in the pairwise cost benefit analysis in the implicit cosheduling algorithm.

In [14], Feitelson and Rudolph used runtime information to identify activity working sets, i.e. the set of activities (tasks) that should be scheduled together, through the monitoring of the utilization pattern of *communication objects* by the activities. Their work can be considered complementary to ours in the sense that our objective here is not to identify activity working sets at runtime but to improve throughput and utilization of parallel machines for different scheduling strategies using such runtime information.

In [19], Lee et al., along with an analysis of I/O implications for gang scheduled workloads, presented a method for runtime identification of gangedness, through the analysis of messaging statistics. It differs from our work in the sense that our objective is not to explicitly identify gangedness, but to provide a task classification, which may vary over time as a function of the application, which can also be used to verify the gangedness of an application in a given moment of time among other possibilities.

This paper is an extension of some of our previous work [24, 25] where we describe the Concurrent Gang scheduling algorithm. In this work we present a more robust task classification scheme, and we investigate new ways of providing better service to I/O and interactive applications in gang scheduling, through utilization of idle slots and idle time due to blocked tasks and by the variation

of the spinning time of a task, taking into account the determination of the spin time information about other tasks.

3 Task Classification using Runtime information

As described in the introduction, our objective is the utilization of various runtime measurements, such as I/O access rates and communication rates, to improve the utilization and throughput in parallel job scheduling. This is achieved through a task classification scheme using runtime information. In this section we detail the task classification made by the operating system based on runtime measurements using fuzzy logic theory. A discussion of the utilization of Bayesian estimators to increase the robustness of the first scheme based on fuzzy logic follows, and a "fuzzy" variation of the Bayesian estimator is presented.

3.1 Task Classification

We will use the information gathered at runtime to allow each PE to classify each one of its allocated tasks into classes. Examples of such classes are: I/O intensive, communication intensive, and computation intensive. Each one of these classes is similar to a fuzzy set [30]. A fuzzy set associated with a class A is characterized by a membership function $f_A(x)$ with associates each task T to a real number in the interval [0,1], with the value of $f_A(T)$ representing the "grade of membership" of T in A. Thus, the nearer the value of $F_A(T)$ to unity, the higher the grade of membership of T in A, that is, the degree to which a task belongs to a given class. For instance, consider the class of I/O intensive tasks, with its respective characteristic function $f_{IO}(T)$. A value of $f_{IO}(T) = 1$ indicates that the task T belongs to the class I/O intensive with maximum degree 1, while a value of $f_{IO}(x) = 0$ indicates that the task T has executed no I/O statement at all. Observe the deterministic nature of grade of membership associations. It is also worth noting that the actual number of classes used on a system depends on the architecture of the machine.

The information related to a task is gathered during system calls and context switches. Information that can be used to compute the grade of membership are the type, number and time spent on system calls, number and destination of messages sent by a task, number and origin of received messages, and other system dependent data. These informations can be stored, for instance, by the operating system on the internal data structure related to the task.

When applying fuzzy sets for task classification, the value of $f(T)$ for a class is computed by the PE in a regular basis, at the preemption of the related task. As an example, let's consider the I/O intensive class. The exact way of computing being system dependent, one way of doing the computation is as follows: On each I/O related system call, the operating system will store information related to the call on the internal data structure associated to the task, and at the end of the time slice, the scheduler computes the time spent on I/O calls in the previous slice. One possible way of computing the grade of membership of a task based

on duration of system calls to the class I/O intensive is to consider an average of the time spent in I/O is made over the last N times where the task was scheduled (N can be, for instance, 3). This average determines the grade of membership of a particular task to the class I/O intensive. As many jobs proceed in phases, the reason for using an average over the last N times a task was scheduled is detection of phase change. If a task changes from a I/O intensive phase to a computation intensive phase, this change should be detected by the scheduler. In general, the computation of the degree of membership of a task to the class I/O intensive will always be a function of the number and/or duration of the I/O system calls made by the task. The same is valid for the communication intensive class; the number and/or duration of communication statements will define the grade of membership of a task to this class. For the class computing intensive, grade of membership will also be a function of system calls and communication statements, but in another sense: for a smaller the number of system calls and communications there is a increase of the grade of membership of a given task to the class computing intensive.

In the next subsection we present a more robust way for computing the grade of membership of a task related to a class than the average over N slices presented in this subsection, through the use of Bayesian estimators.

3.2 Task Classification using Bayesian Estimators

The objective of this section is to introduce a more robust task classification mechanism than the one described in the last section, which is the average of the last N measurements, using elements of Bayesian decision theory. Bayesian decision theory is a formal mathematical structure which guides a decision maker in choosing a course of action in the face of uncertainty about the consequences of that choice[17]. In particular we will be interested in this section in defining a task classifier using a Bayesian estimator adapted to the fuzzy theory.

A Bayesian model is a statistical description of an estimation problem which has two main components. The first component, the prior model $p(u)$ (this probability function is also known as prior probability distribution) is a probabilistic description of the world or its properties before any sense data is collected. The second component, the sensor model $p(d|u)$, is a description of the noisy or stochastic process that relate the original (unknown) state u to the sampled input image or sensor values d. These two probabilistic models can be combined to obtain a posterior model, $p(u|d)$ (posterior probability distribution), which is the probabilistic description of the current estimate of u given the data d. To compute the posterior model we use Bayes' rule:

$$p(u|d) = \frac{p(d|u)p(u)}{p(d)} \tag{1}$$

where

$$p(d) = \sum_{u} p(d|u)p(u) \tag{2}$$

The fuzzy version of equation 1 to compute the grade membership of a task T to a class i as a function of measurement E can be written as[18]:

$$S_E(i) = \frac{S_i(E) f_i(T)}{\sum_1^k S_j(E) f_j(T)} \tag{3}$$

Where $S_j(k)$ represents subsethood between two fuzzy sets j and k. In our case $S_i(E)$ is the subsethood between the two fuzzy sets represented by measurement E on task T and class i, that is, the grade of membership of task T relative to class i considering only the data gathered at measurement E. $f_i(T)$ is the grade of membership of task T relative to class i before measurement E. $S_E(i)$ in our case represents the grade of membership of task T relative to class i after the measurement E and becomes $f_i(T)$ in the next interval computation.

4 Scheduling Using Runtime measurements

In this section we will illustrate one possible use of task classification to improve scheduling in parallel machines. Our emphasis here is to improve throughput and utilization of gang schedulers. Observe that the strategies described in this section can be applied to a large number of gang scheduler implementations, including traditional gang schedulers[3, 15] and distributed hierarchical control schedulers [11, 12].

We may consider two types of parallel tasks in a gang scheduler: Those that should be scheduled as a gang with other tasks in other processors and those for which gang scheduling is not mandatory. Examples of the first class are tasks that compose a job with fine grain synchronization interactions [13] and communication intensive jobs[8]. Second class task examples are local tasks or tasks that compose an I/O bound parallel job, for instance. On the other hand a traditional UNIX scheduler does a good job in scheduling I/O bound tasks since it gives high priority to I/O blocked tasks when data become available from disk. As those tasks typically run for a small amount of time and then block again, giving them high priority means running the task that will take the least amount of time before blocking, which is coherent to the theory of uniprocessors scheduling where the best scheduling strategy possible under the sum of completion times is Shortest Job First [20](in [20] authors define the sum of completion times as total completion time). Another example of jobs where gang scheduling is not mandatory are embarrassingly parallel jobs. As the number of iterations among tasks belonging to this class of jobs are small, the basic requirement for scheduling an embarrassingly parallel job is to give those jobs the greater possible fraction of CPU time, even in an uncoordinated manner.

Differentiation among tasks that should be gang scheduled and those for which a more flexible scheduler is better is made using the grade of membership information computed by each PE (as explained in the last subsection) for each task allocated to a processor. The grade of membership of the task currently

scheduled is computed at the next preemption of the task, and it is that information that is used to decide if gang scheduling is mandatory or not for a specific task.

When using task classification information, the local task scheduler on each PE computes a priority for each task allocated to the PE. This priority defines if a task T is a good candidate for being rescheduled if another task blocks or in case of a idle slot. The priority of each task is defined based on the grade of membership of a task to each one of the major classes described before. As an example of the computation of the priority of a task T in a PE we have:

$$Pr(T) = max(\alpha \times f_{IO}, f_{COMP}) \tag{4}$$

Where f_{IO}, f_{COMP} are the grade for membership of task T to the classes I/O intensive and Computation intensive. The objective of the parameter α is to give greater priority to I/O bound jobs ($\alpha > 1$). The choices made in equation 4 intend to give high priority to I/O intensive jobs and computation intensive job, since such jobs can benefit the most from uncoordinated scheduling. The multiplication factor α for the class I/O intensive gives higher priority to I/O bound tasks over computation intensive tasks, since those jobs have a greater probably to block when scheduled than computing bound tasks. By other side, communication and synchronization intensive jobs have low priority since they require coordinated scheduling to achieve efficient execution and machine utilization[13, 8]. A communication intensive phase will reflect negatively over the grade of membership of the class computation intensive, reducing the possibility of a task be scheduled by the local task scheduler. Among a set of tasks of the same priority, the local task scheduler uses a round robin strategy. The local task scheduler also defines a minimum priority β. If no parallel task has priority larger than β, the local task scheduler considers that all tasks in the PE do intensive communication and or synchronization, thus requiring coordinated scheduling. Observe that there is no starvation of communication intensive jobs, as they will be scheduled in a regular basis by the gang scheduler itself, regardless of the decisions made by the local task schedulers.

Observe that the parameters α and β define the bounds of the variation of the priority of a task in order to it be considered to rescheduling, as stated in the next proposition.

Proposition 1. *$\alpha \leq Pr(T) \leq \beta$, in order to a task be considered for rescheduling.*

Proof. β is the lower bound by definition. For the upper bound, observe that $f_{IO}^{max} = 1$. So, as $\alpha > 1$, the upper bound is $\alpha \times 1 = \alpha$

Simulations in [25] of a scheduling algorithm (Concurrent Gang) that uses a simpler version of the priority mechanism/task classification described here have shown that the priority computation has better performance than other algorithms that can be used to choose the task that runs next, such as round robin.

Interactive tasks can be regarded as a special type of I/O intensive task, where the task waits for a input from the user at regular intervals of time. These tasks also suffer under gang scheduling, and should have priority as I/O intensive tasks.

5 Adjusting Spinning Time as a function of the workload

Another parameter that can be adjusted in order to improve throughput of I/O bounds and interactive jobs in gang scheduling is the spinning time of a task. Our objective is to make changes not only as a function of the runtime measurements of the related job, but also considering other jobs where tasks are allocated to the same processor. We consider that a typical workload will be composed of a mix of jobs of different types and it is important to achieve a compromise in order to give a good response for all types of jobs.

The anticipated blocking of a job performing synchronization or communication can benefit those jobs that do not need coordinated scheduling, such as I/O intensive and embarrassingly parallel. So the idea is to determine the spinning time of a task as a function of the workload allocated in a processor. For instance, in a given moment of time if a processor has many I/O intensive jobs allocated to it, this would have a negative impact in spinning time duration. As described in [1], a minimum spin time should be guaranteed in order to insure that processes stay coordinated if already in such a state (baseline spin time). This minimum amount of time ensures the completion of the communication operation when all involved processes are scheduled and there is no load imbalance among tasks of the same job.

Considering gang scheduling the spinning time of a task may vary between a baseline spin time and a spin only state with no blocking. The main external factor that will have influence in the variation of the spin time is the number of interactive and I/O bound tasks in the workload allocated to one processor. A large number of these tasks would imply a smaller spinning time, in order to use the remaining time until the next global preemption to schedule those tasks, providing better service to I/O bound and interactive tasks. The algorithm we propose to set up the spinning time as a function of the workload on a given PE for a gang scheduling based algorithm is as follows: If there is one or more tasks in a PE classified as I/O intensive or interactive, a task doing communication will block just after the baseline spin time if the two following conditions are satisfied:

- At least one of the tasks classified as interactive or I/O bound is ready
- There is a minimum amount of time δ between the end of baseline and the next context switch epoch.

If any of the two conditions are not satisfied the task doing communication will spin until receiving the waited response. The δ time is a function of the context switch time of the machine. Given γ, the context switch time of the machine, it is clear that $\delta > \gamma$. We can define that $\delta > 2 \times \gamma$, in order to give the

job at least the same amount of CPU time that the system will spend in context switch. In our experiments we empirically define it as being 4 times the average amount of time required for a context switch.

If both conditions are satisfied, the tasks will spin for a time corresponding to the baseline spin time, and if no message is received the task blocks and the I/O bound or interactive task can be scheduled. The reason of minimizing the spinning time is the need of I/O and interactive tasks to receive better service in gang scheduling, and the fact that in gang scheduling tasks are coordinated due to the scheduling strategy itself; so an application with no load imbalances would need only the time corresponding the baseline to complete the communication.

The control of spin time using task classification information is another mechanism available to the scheduler to provide better service to I/O bound and interactive jobs under gang scheduling along with the priority computation described in the previous section. Observe that the spin time control as a function of the workload is always used in conjunction with the priority mechanism described in section 4.

6 Experimental Results

In this section we present some simulation results that compares the performance of a gang scheduler that uses the algorithms described in sections 4 and 5 with another gang scheduler without such mechanisms, both of them using the same packing strategy (first fit). The implementation of gang scheduler used in this section is a simple one; our objective is to measure the benefits of using runtime measurements and task classification information by comparing a given scheduler that makes use of runtime information with another one that does not consider it. First we describe our simulation methodology, and then we present and comment the results obtained in our simulations.

6.1 Simulation Methodology

To perform the actual experiments we used a general purpose event driven simulator being developed by our research group for studying a variety of problems (e.g., dynamic scheduling, load balancing, etc). This simulation was first described in [23] and for the experiments of this section we used an improved version that supports the change of the spinning time of a task during a simulation.

We have modeled in our simulations a network of workstations connected by a network characterized by LogP[6, 5] parameters. The LogP parameters corresponds to those of a Myrinet network, and they were the similar to the ones used in [1], with Latency being equal to 10 μs, and overhead to 8.75 μs. We defined the baseline spin time as being equal to a request-response message pair, which in the LogP model is equal to 2L+4o. Therefore, the baseline time is equal to 55 μs. The number of processors considered were 8 and 16. I/O requests of a job were directed to the local disk of each workstation, and consecutive requests

were executed on a first come first serve basis. Quantum size is fixed as being equal to 200 ms and context switch time equal to 200 μs.

The values of the α and β parameters used for simulations were $\alpha = 2$ and $\beta = 0.3$. As stated in proposition 1 the priority should vary inside the bounds defined by α and β in order to a task be considered to reschedule.

For defining job inter arrival, time, job size and job duration we used a statistical model proposed in [7]. This is model of the workload observed on a 322-node partition of the Cornell Theory Center's IBM SP2 from June 25, 1996 to September 12,1996. The model is based on finding Hyper-Erlang distributions of common order that match the first three moments of the observed distributions. As the characteristics of jobs with different degrees of parallelism differ, the full range of degrees of parallelism is first divided into subranges. This is done based on powers of two. A separate model of the inter arrival times and the service times (runtimes) is found for each range. The defined ranges are 1, 2, 3-4, 5-8, 9-16, 17-32, 33-64, 65-128, 129-256 and 257-322. For the simulations for a 16 processors machine we used 5 ranges, and for a 8 processors machine 4 ranges. The time unit of the parameters found in [7] was seconds, and the duration of all simulations was defined as being equal to 50000 seconds. A number of jobs are submitted during this period in function of the inter arrival time, but not necessarily all submitted jobs are completed by the end of simulation. A long time was chosen in order to minimize the influence of start-up effects.

In order to avoid the saturation of the machine, we limited the number of tasks that can be allocated to a node at a given moment of time to 10. If a job arrives and there is no set of processors available with less than 10 tasks allocated to them, the task waits until the required number of processors become available.

We use a mix of four types of synthetic applications in our experiments:

- *I/O* - This job type is composed of bursts of local computations followed by bursts of I/O commands, as represented in figure 1. This pattern reflects the I/O properties of many parallel programs, where execution behavior can be naturally partitioned into disjoint intervals, each of which consist of a single burst of I/O with a minimal amount of computation followed by a single burst of computation with a minimal amount of I/O [22]. The interval composed of a computation burst followed by an I/O burst are know as phases, and a sequence of consecutive phases that are statistically identical are defined as a working set. The execution behavior of an I/O bound program is therefore comprised as a sequence of I/O working sets. This general model of program behavior is consistent with results from measurement studies [26, 27]. The time duration of the I/O burst was equal to 100 ms in average. The ratio of the I/O working set used in simulations was 1/1, that is, for a burst of 100 ms of I/O there was a burst of 100 ms of computation in average. Observe that I/O requests from different jobs to the same disk are queued and served by arrival order.
- *Embarrassingly parallel* - In this kind of application constituent processes work independently with a small amount or no communication at all a-

mong them. Embarrassingly parallel applications require fair scheduling of the constituent processes, with no need for explicit coordinated scheduling.

- *Msg* - In this type of synthetic application we model message passing jobs, where messages are exchanged between two processes chosen at random. Each process sends or receives a message every 10 *ms* in average. The communication semantics used here were the same of the PVM system[4], that is, asynchronous sends and blocking receives. For the modified version of gang scheduler, the one that incorporates spin control and priority computation, the spinning time of the receive call will be defined by the spin control mechanism described in section 5. The pure gang scheduler only implements the spin only mechanism, since the original gang schedulers do not know what to do if a task blocks.
- *BSP* - This type of application models Bulk Synchronous Parallel (BSP) style jobs[28], where there is a sequence of supersteps, each superstep being composed of a mix of computation/communication statements, with all processes being synchronized between two supersteps. In this type of applications, there is a synchronization call every 50 *ms* (in average) and all communication/computation generated previous to the barrier call is completed before the job proceeds in the next computation/communication superstep. Again, there is a spin time associated with the barrier and communication calls.

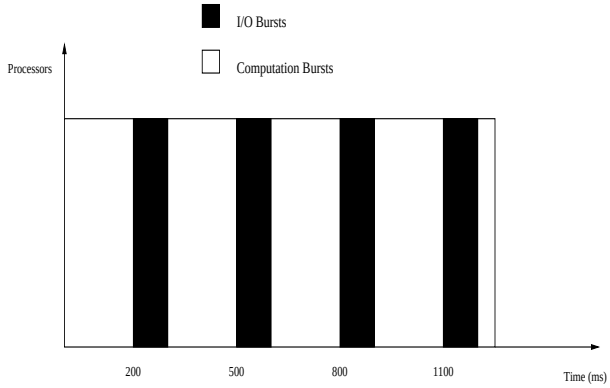

Fig. 1. I/O bound job with one I/O working set

In all simulations, the same sequence of jobs were submitted to both a Gang scheduler with the priority computation and spin time control mechanisms described in section 4 and 5 and another gang scheduler without such mechanisms. A different sequence is generated for each experiment. The packing strategy was first fit without thread migration. Each workload was composed of a mix of the 4 types of jobs previously defined:

- *IO*- This workload was composed of I/O bound jobs only. As I/O bound jobs suffer under gang scheduling, this workload was simulated in order to evaluate the performance impact of the modified gang scheduler if compared to a traditional gang scheduler.
- *IO/Msg* - This workload was composed of a mix of IO and Msg jobs. At each job arrival, the job type was chosen according with a uniform distribution, with a probability of 0.5 to both jobs
- *IO/BSP* - As in the previous workload, both job types had the same probability of being chosen at each job arrival.
- *IO/Msg/Embarrassingly* - Since the priority mechanisms intends to give better service to I/O bound and Compute intensive bounds, we included the Embarrassingly parallel type in the IO/Msg workload, to verify is there is any improvements in throughput due to the inclusion of computing intensive jobs.
- *IO/BSP/Embarrassingly* - Same case for the IO/BSP workload. As in previous cases, at each job arrival all three job types have equal probability to be chosen.
- *Emb/Msg and Emb/BSP* - These workloads were added to evaluate the impact of the priority mechanism over workloads that do not include I/O bound jobs. They are composed of Embarrassingly parallel jobs with Msg and BSP job types respectively. In this case the spin control is not activated since it is conceived to provide better service to I/O bound and interactive tasks only, as these are the type of jobs that have poor performance under gang scheduling.

A second set of experiments were performed using the workloads IO/BSP and IO/Msg to compare the performance of a gang scheduler with both the priority computation and spin control mechanisms with another gang scheduler having only the priority control mechanism in order to evaluate the impact of the spin control in the results presented.

6.2 Simulation Results

Simulations results for the IO workload are shown in figure 2. In the utilization column, the machine utilization (computed as a function of the total idle time of the machine on each simulation) of the modified gang scheduler was divided by the machine utilization of the non-modified version of the gang scheduler. In the throughput column, the throughput of the modified gang scheduler (The number of jobs completed until the end of the simulation, 50000 seconds) is divided by the throughput in the original gang. We can see a very significant improvement of the modified gang over the original gang scheduler, due to the priority mechanism. To explain the reason of such improvement, tables 1 and 2 show the actual results of simulations for 8 and 16 processors machines under the I/O bound workload. In [22] , Rosti et al. suggest that that the overlapping of the I/O demands of some jobs with the computational demands of other jobs may offer a potential improvement in performance. The improvement shown in figure 2 is

due to this overlapping. The detection of I/O intensive tasks and the immediate scheduling of one of these tasks when another task doing I/O blocks results in a more efficient utilization of both disk and CPU resources. As we consider an I/O working set composed by a burst of 100 ms of computation followed by another burst of 100 ms of I/O, the scheduler implementing the priority mechanism always tries to overlap the I/O phase of a job with the computation phase of another, which explains the results obtained. In the ideal case, the scheduling strategy will be able to interleave the execution of applications such that the ratio of the per-phase computation and I/O requirements is maintained very close to 1, thus achieving a total overlapping of computation and I/O. For this workload, since the utilization of the machine is doubled by using runtime information, we can conclude that the overlap of I/O phase is almost 100%, since the duration of the I/O phase is in average equal to the duration of the computation phase and the utilization obtained for the gang scheduler without runtime information is due only to the computation phase. The differences between throughput and utilization are due to long-running jobs that have not yet completed by the end of the simulation (50000 seconds). Another interesting point is that, in both machines, about half of the completed jobs were 1 task jobs, since a large amount of jobs generated by the workload model were 1 task jobs.

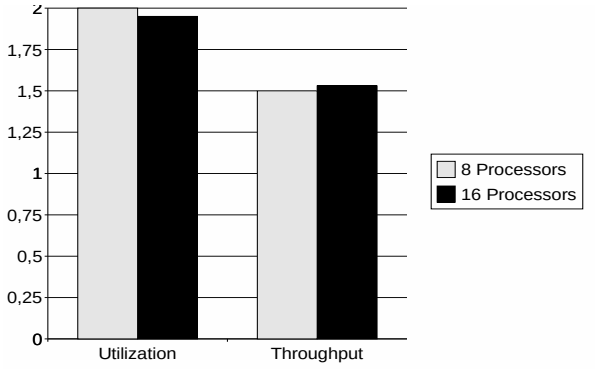

Fig. 2. I/O bound workload with one I/O working set

Table 1. Experimental results - I/O intensive workload - 8 Processors

8 Processors	Jobs Completed	Utilization (%)
With Runtime Information.	60	84
Without Runtime Information	40	42

Table 2. Experimental results - I/O intensive workload - 16 Processors

16 Processors	Jobs Completed	Utilization (%)
With Runtime Information.	55	84
Without Runtime Information	36	43

For the IO/Msg workload, results are shown in figure 3. Again, the modified gang achieved better results for both throughput and utilization. Since Gang schedulers have good performance for communication bound jobs, the improvement due the utilization of runtime measurements and task classification is smaller if compared to the results obtained for the IO workload, as the machine utilization of the gang scheduler without runtime information is better in this case if compared to the results related to the previous workload. Tables 3 and 4 show the absolute machine utilization for the experiments using the IO/Msg workload. As the machine utilization for the regular gang scheduler is around 60%, an improvement in utilization as observed with the IO workload is no longer possible.

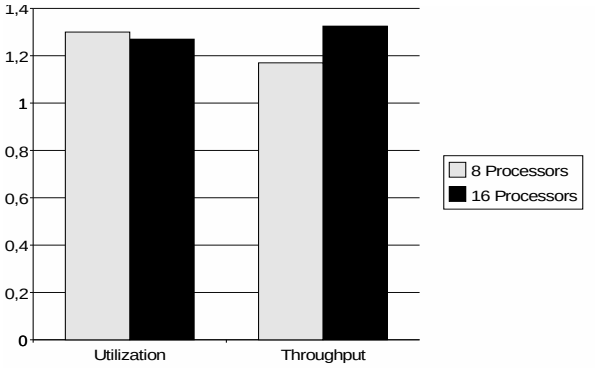

Fig. 3. IO/Msg workload

Table 3. Experimental results - IO/Msg workload - 8 Processors

8 Processors	Jobs Completed	Utilization (%)
With Runtime Information.	50	82
Without Runtime Information	43	63

Results for the IO/Msg/Emb workload are shown in figure 4. The greater flexibility of the modified gang algorithm to deal with I/O intensive and embar-

Table 4. Experimental results - IO/Msg workload - 16 Processors

16 Processors	Jobs Completed	Utilization (%)
With Runtime Information.	53	79
Without Runtime Information	40	62

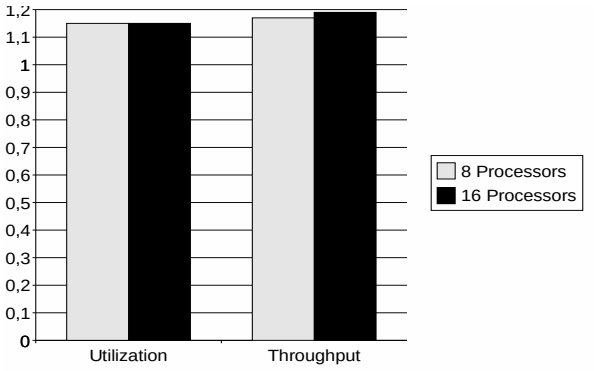

Fig. 4. IO/Msg/Emb workload

rassingly parallel jobs results in an increase in throughput and utilization. It is worth noting, however, that the influence of idle time due to I/O bound jobs is reduced, with the regular gang scheduler having even better machine utilization if compared to results for the IO/Msg workload, as shown in tables 5 and 6.

Table 5. Experimental results - I0/Msg/Emb workload - 8 Processors

8 Processors	Jobs Completed	Utilization (%)
With Runtime Information.	47	83
Without Runtime Information	40	72

When we substitute the Msg workload for the BSP workload in the previous experiments, results are similar in both relative and absolute values. The reason is that both types of jobs are communication/synchronization intensive, taking advantage of the gang scheduling strategy. Results for IO/BSP and IO/BSP/Emb workloads are shown in figures 5 and 6 respectively. As in previous cases, there is improvement over the gang scheduler without the priority computation and spin control mechanisms in both utilization and throughput. Again, the combination of the priority and spin control mechanisms explains the better results obtained by the scheduler using runtime measurements for both workloads.

To evaluate the impact of the spin control mechanism in the total performance of the modified gang scheduler, we compared the performance between

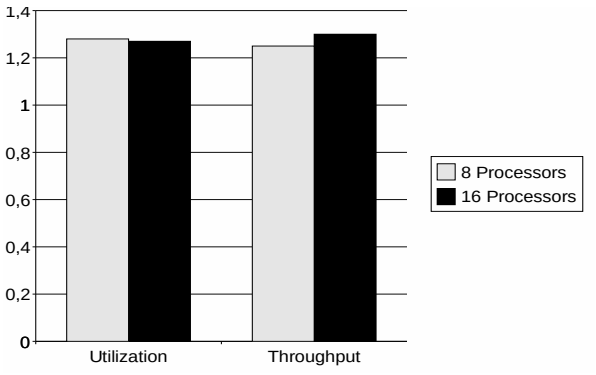

Fig. 5. IO/BSP workload

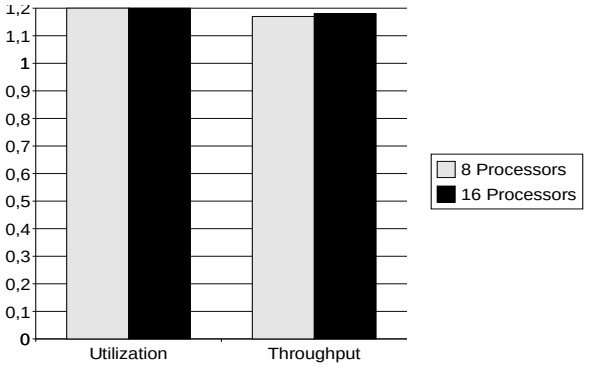

Fig. 6. IO/BSP/Emb workload

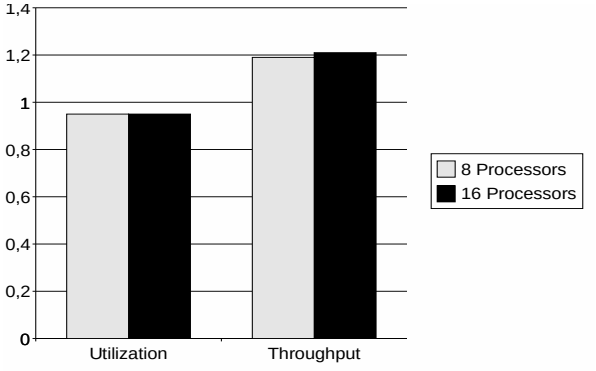

Fig. 7. Evaluation of the spin control mechanism - IO/BSP workload

Table 6. Experimental results - IO/Msg/Emb workload - 16 Processors

16 Processors	Jobs Completed	Utilization (%)
With Runtime Information.	61	81
Without Runtime Information	51	70

a modified gang with both the priority and spin control mechanisms and other version of the modified gang where only the priority computation was active. Results for workloads IO/Bsp and IO/Msg are shown in figures 7 and 8 respectively. In figures 7 and 8 the performance of the scheduler with spin control and priority mechanism is divided by the performance of the gang scheduler with the priority computation only. The gain in throughput is due to the better service provided to I/O bound jobs, while in utilization gang scheduling with only the priority mechanism has slightly better performance. This can be explained by the fact the I/O bound jobs run for some time and then block again, while BSP and Msg jobs keep spinning and runs again after receiving the message. As said before, the objective of the spin control mechanism is to achieve a compromise in order to have a better performance for I/O intensive tasks, because these tasks suffer under gang scheduling. In gang scheduling with spin control and priority, this compromise is achieved by given a better a service to I/O bound jobs, having as consequence a reduction in the spin time of synchronization/communication intensive tasks.

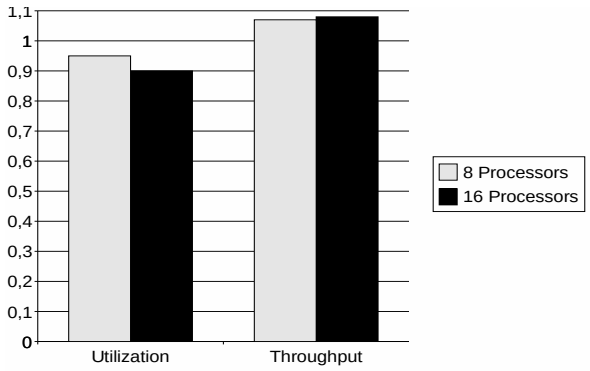

Fig. 8. Evaluation of the spin control mechanism - IO/Msg Workload

To evaluate the performance impact for workloads with no I/O intensive jobs, we have simulated two workloads composed of embarrassingly parallel jobs with Msg and BSP jobs respectively. Comparative results are displayed in figures 9 and 10. Since gang scheduling has a good performance for both synchronization and communication intensive jobs, the improvement is reduced if compared to

the previous workloads. Observe that the performances of both the regular gang scheduler and the gang scheduler using runtime information are quite similar. The main improvement in these cases is in utilization and its due mainly to the scheduling of tasks belonging to embarrassingly parallel jobs on idle slots in the Ousterhout matrix[21], that is, those time slices where a processor do not has a parallel task to schedule.

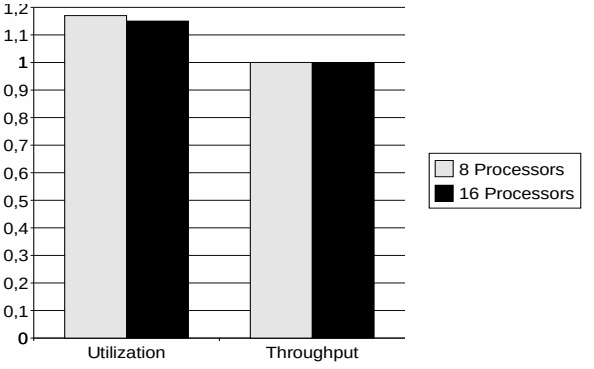

Fig. 9. Emb/Msg workload

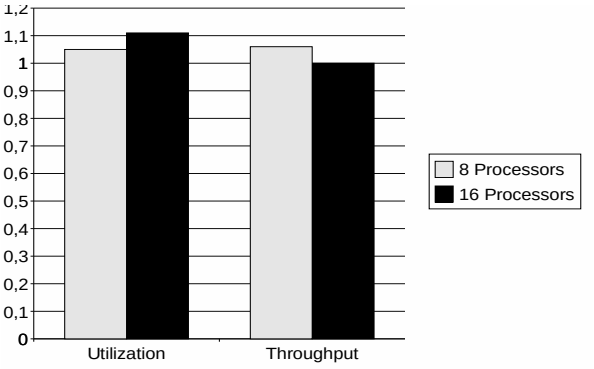

Fig. 10. Emb/BSP workload

7 Conclusion

In this paper we present some possible uses of runtime measurements for improving throughput and utilization in parallel job scheduling. We believe that

incorporating such information in parallel schedulers is a step in the right direction, since with more information available about running jobs in a given moment of time a scheduler will be able to do a intelligent choice about many events in parallel task scheduling, such as what task should have higher priority as a function of the base scheduling algorithm used, how to change operating systems parameters in order to improve machine utilization, etc. The increase in throughput and utilization is confirmed by the experimental results we obtained.

However, there a number of possibilities not explored in this paper that are subject of our current and future research. For instance, questions that we are investigating are the use of runtime information and task classification to improve parallel/distributed scheduling without explicit coordination, the effects of the utilization of runtime measurements on other implementations of gang scheduling such as the distributed hierarchical control algorithm, the utilization of task classification to identify gangedness of an application, and other ways of using task classification information to improve parallel job scheduling.

Acknowledgments The first author is supported by Capes, Brazilian Government, grant number 1897/95-11. The second author is supported in part by the Irvine Research Unit in Advanced Computing and NASA under grant #NAG5-3692.

References

1. A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of ACM SIGMETRICS'98*, pages 233–243, 1998.
2. J. Edmonds, D.D. Chinn, T. Brecht, and X. Deng. Non-Clairvoyant Multiprocessor Scheduling of Jobs with Changing Execution Characteristics (extended abstract). In *Proceedings of the 1997 ACM Symposium of Theory of Computing*, pages 120–129, 1997.
3. A. Hori et al. Implementation of Gang Scheduling on Workstation Cluster. *Job Scheduling Strategies for Parallel Processing*, LNCS 1162:126–139, 1996.
4. Al Geist et al. *PVM : Parallel Virtual Machine - A User's guide and tutorial for networked parallel computing.* The MIT Press, 1994.
5. D. Culler et al. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of 4th ACM SIGPLAN Symposium on Principles an Practice of Parallel Programming*, pages 1–12, 1993.
6. D. Culler et al. A Practical Model of Parallel Computation. *Communication of the ACM*, 93(11):78–85, 1996.
7. J. Jann et al. Modeling of Workloads in MPP. *Job Scheduling Strategies for Parallel Processing*, LNCS 1291:95–116, 1997.
8. Patrick G. Solbalvarro et al. Dynamic Coscheduling on Workstation Clusters. *Job Scheduling Strategies for Parallel Processing*, LNCS 1459:231–256, 1998.
9. D. Feitelson. Packing Schemes for Gang Scheduling. *Job Scheduling Strategies for Parallel Processing*, LNCS 1162:89–110, 1996.
10. D. Feitelson and M. A.Jette. Improved Utilization and Responsiveness with Gang Scheduling. *Job Scheduling Strategies for Parallel Processing*, LNCS 1291:238–261, 1997.

11. D. Feitelson and L. Rudolph. Distributed Hierarchical Control for Parallel Processing. *IEEE Computer*, pages 65–77, May 1990.
12. D Feitelson and L. Rudolph. Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hiearchical Control. In *Proceedings of the 1990 International Conference on Parallel Processing*, 1990.
13. D. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.
14. D. Feitelson and L. Rudolph. Coscheduling Based on Runtime Identification of Activity Working Sets. *International Journal of Parallel Programming*, 23(2):135–160, 1995.
15. A. Hori, H. Tezuka, and Y. Ishikawa. Overhead Analysis of Preemptive Gang Scheduling. *Job Scheduling Strategies for Parallel Processing*, LNCS 1459:217–230, 1998.
16. M. A. Jette. Performance Characteristics of Gang Scheduling In Multiprogrammed Environments. In *Proceedings of SC'97*, 1997.
17. J.J.Martin. *Bayesian Decison Problems and Markov Chains*. John Wiley and Sons Inc., New York, N.Y., 1967.
18. B. Kosko. *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach for Machine Intelligence*. Prentice Hall, Inc., 1992.
19. W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. Implications of I/O for Gang Scheduled Workloads. *Job Scheduling Strategies for Parallel Processing*, LNCS 1291:215–237, 1997.
20. R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. *Theoretical Computer Science*, 130(1):17–47, 1994.
21. J.K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the 3rd International Conference on Distributed Comp. Systems*, pages 22–30, 1982.
22. E. Rosti, G. Serazzi, E. Smirni, and X. M. S. Squillante. The Impact of I/O on Program Behavior and Parallel Scheduling. In *Proceedings of ACM SIGMETRICS'98*, pages 56–64, 1998.
23. F.A.B. Silva, L.M. Campos, and I.D. Scherson. A Lower Bound for Dynamic Scheduling of Data Parallel Programs. In *Proceedings EUROPAR'98*, 1998.
24. F.A.B. Silva and I.D. Scherson. Towards Flexibility and Scalability in Parallel Job Scheduling. In *Proceedings of the 1999 IASTED Conference on Parallel and Distributed Computing Systems*, 1999.
25. F.A.B. Silva and I.D. Scherson. Improving Throughput and Utilization on Parallel Machines Through Concurrent Gang. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium 2000*, 2000.
26. E. Smirni, R. A. Aydt, A. A. Chien, and D. A. Reed. I/O Requirements of scientific aplications: an evolutionary view. In *Proceedings of the IEEE international Symposium of High Performance Distributed Computing*, pages 49–59, 1996.
27. E. Smirni and D. A. Reed. Lessons from characterizing the input/output behavior of parallel scientific applications. *Performance Evaluation*, 33:27–44, 1998.
28. L. G. Valiant. A bridging model for parallel computations. *Communications of the ACM*, 33(8):103 – 111, 1990.
29. F. Wang, M. Papaefthymiou, and M. S. Squillante. Performance Evaluation of Gang Scheduling for Parallel and Distributed Multiprogramming. *Job Scheduling Strategies for Parallel Processing*, LNCS 1291:277–298, 1997.
30. L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8:338–353, 1965.