# Gang Scheduling with a Queue for Large Jobs

B. B. Zhou
School of Computing & Mathematics
Deakin University
Geelong, VIC 3217, Australia

R. P. Brent
Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford OX1 3QD, UK

**Abstract**

*Applying gang scheduling can alleviate the blockade problem caused by exclusively space-sharing scheduling. To simply allow jobs to run simultaneously on the same processors as in conventional gang scheduling, however, may introduce a large number of time slots in the system. In consequence the cost of context switches will be greatly increased, and each running job can only obtain a small portion of resources including memory space and processor utilisation and so no jobs can finish their computations quickly. Therefore, the number of jobs allowed to run in the system should be limited. In this paper we present some experimental results to show that by limiting real large jobs time-sharing the same processors and applying the backfilling technique we can greatly reduce the average number of time slots in the system and significantly improve the performance of both small and large jobs.*

## 1 Introduction

Many job scheduling strategies have been introduced for parallel processing. (See a good survey in [5].) These scheduling strategies can be classified into either *space sharing*, or *time sharing*. Because a time-shared environment is more difficult to establish for multiple processor systems, currently most commercial parallel systems only adopt space sharing. One major drawback of space sharing is the blockade situation, that is, small jobs can easily be blocked for a long time by large ones. The *backfilling* technique was then introduced to alleviate this problem [7, 9]. With backfilling one attempts to allocate idle processors to small jobs which are behind in the queue of waiting jobs if the allocation does not cause starvation of larger jobs. As more parallel software packages are developed for various kinds of applications and more and more ordinary users are getting familiar with parallel systems, it is expected that the workload on such systems will become heavy in the near future. With exclusively space-sharing scheduling, however, the blockade can still be a serious problem under heavy workload. To alleviate this problem, time sharing needs to be considered.

It is known that coordinated scheduling of parallel jobs across the processors is a critical factor to achieve efficient parallel execution in a time-shared environment. Coordinated scheduling strategies can be classified into two different categories. The first is called *implicit coscheduling*. This approach does not use a global scheduler, but local schedulers on each processor to make scheduling decisions based on the communication behavior of local processes. There are two types of implicit coscheduling, that is, *dynamic coscheduling* with which scheduling decisions are made based on message arrivals [10], and *two-phase blocking* which uses more information such as response time, message arrivals and the amount of scheduling progress made by each local process [3]. Implicit coscheduling is attractive for loosely coupled clusters without a central resource management system.

The second type of coscheduling is called *explicit coscheduling* [8], or *gang scheduling* [6]. With explicit coscheduling processes of the same job will run simultaneously for a certain amount of time which is called *scheduling slot*, or *time slot*. When a time slot is ended, the processors will context-switch at the same time to give the service to processes of another job. Controlled by a global scheduler, all parallel jobs in the system take turns to receive the service in a coordinated manner. It gives the user an impression that the job is not blocked, but executed on a dedicated slower machine when the system workload is heavy. In this paper we shall only consider how to improve the performance of explicit coscheduling, or gang scheduling.

Although gang scheduling is currently the most popular scheduling strategy for parallel processing in a time-shared environment, there are still certain fundamental problems which remain to be solved. One major problem is *resource contention*, that is, a number of parallel jobs compete for limited resources in a system. The first of this kind is called *memory pressure*, that is, a number of jobs simultaneously running on the same processors demands a memory space larger than the actual memory a system provides. Using the paging mechanism might alleviate the problem. However, research indicates that paging may cause a great degradation of job performance [1]. The second contention problem is associated with limited computing power of a given system. If many jobs time-share the same set of processors, each job can only obtain very small portion of processor utilisation and no job can complete quickly.

Recently several methods have been proposed to alleviate this kind of contention problem. For example, the reported experimental results in [1] show that using a queue to delay job execution is more efficient than running jobs all together with paging applied. In [12], for another example, the authors first set a *multiprogramming level*, or a limit for the maximum number of jobs which are allowed to run simultaneously on the same processor. If the maximum level is reached, the new arrivals have to be queued. The author then combines the gang scheduling and the backfilling technique to achieve a good performance.

It seems that using a waiting queue to delay jobs execution is a good way to alleviate the problem of resource contention. The question is, however, which jobs should be queued. Different answers to this question will lead to different system and job performance. We decide to apply a waiting queue for large jobs based on three main reasons:

1. If the size of a job is large, the user normally do not expect that the job will be completed in a short time. However, the user will demand a short response time when submitting a small job. Queuing large jobs will enhance the performance of small jobs.

2. Conventionally, jobs are not distinguished according to their execution times when gang scheduling is considered. It should be pointed out that the simple round robin scheme used in gang scheduling works well only if the sizes of jobs are distributed in a wide range. Gang scheduling using the simple round robin may not perform as well as even a simple FCFS scheme in terms of average response time, or average slowdown, when all the incoming jobs are large. To limit the number of large jobs simultaneously running on the same processors should improve the performance of both large and small jobs.

3. A large job usually involves a large amount of data set (with some exceptions). The memory presure can be alleviated if the number of simultaneously running large jobs is limited.

In this paper we shall present some simulation results to show that, by limiting large jobs simultaneously running on the same processors, we can improve the performance in terms of slowdown for all jobs, and slowdown for large jobs as well if the backfilling technique is applied, and we can also decrease the average number of time slots in the system.

The paper is organised as follows: In Section 2 we briefly describe the gang scheduling system implemented for our experiments. A workload model used in our experiments is discussed in Section 3. Experimental results and discussions are presented in Sections 4, 5 and 6. Finally the conclusions are given in Section 7.

## 2 Our Gang Scheduling System

The gang scheduling system implemented for our experiments is mainly based on a *job re-packing* allocation strategy which is introduced for enhancing both resource utilisation and job performance [13, 14].

Conventional resource allocation strategies for gang scheduling only consider processor allocation within the same time slot and the allocation in one time slot is independent of the allocation in other time slots. One major disadvantage of this kind of allocation is the problem of fragmentation. Because processor allocation is considered independently in different time slots, freed processors due to job termination in one time slot may remain idle for a long time even though they are able to be re-allocated to existing jobs running in other time slots.

One way to alleviate the problem is to allow jobs to run in multiple time slots [4, 11]. When jobs are allowed to run in multiple time slots, the buddy based allocation strategy will perform much better than many other existing allocation schemes in terms of average slowdown [4].

Another method to alleviate the problem of fragmentation is job re-packing. In this scheme we try to rearrange the order of job execution on the originally allocated processors so that small fragments of idle processors from different time slots can be combined together to form a larger and more useful one in a single time slot. Therefore, processors in the system can be utilised more efficiently. When this scheme is incorporated into the buddy based system, we can set up a *workload tree* to record the workload conditions of each subset of processors. With this workload tree we are able to simplify the search procedure for available processors, to balance the workload across the processors and to quickly determine when a job can run in multiple time slots and when the number of time slots in the system can be reduced.

With a combination of job re-packing, running jobs in multiple time slots, minimising time slots in the system, and applying buddy based scheme to allocate processors in each time slot we are able to achieve high efficiency in processor utilisation and a great improvement in job performance [14].

To conduct our experiments we add a waiting queue in the gang scheduling system described above. The queue is used only to queue large jobs. Small jobs can enter the system and be executed immediately on their arrivals without any restrictions. In our first two experiments we use a simple FCFS queue. However, the backfilling technique is adopted to try to enhance the performance of large jobs in our third experiment.

We introduce two parameters in our experimental system. By varying these two parameters we are able to see how the added queue affects the system performance.

The first parameter is $j_{max}$, the maximum number of large jobs which are allowed to run simultaneously on the same processors. When $j_{max} = 1$, large jobs are not allowed to time-share the same processors. However, it is just a conventional gang scheduler when $j_{max}$ becomes very large because a large number of large jobs will be allowed to run simultaneously on the same processors and no job will be queued before being executed.

Assume the execution time of the largest job is $t^e$. A job will be considered "large" in each test if its execution time is greater than $\alpha t^e$ for $0.0 \leq \alpha \leq 1.0$. These "large" jobs will first enter the queue before being executed.

## 3 The Workload Model

In our experiment we adopted one workload model proposed in [2]. Both job runtimes and sizes (the number of processors required) in this model are distributed uniformly in log space (or uniform-log distributed), while the interarrival times are exponentially distributed. This model was constructed based on observations from the Intel Paragon at the San Diego Supercomputer Center and the IBM SP2 at the Cornell Theory Center and has been used by many researchers to evaluate their parallel job scheduling algorithms.

Since the model was originally built to evaluate batch scheduling policies, we made a few minor modifications in our simulation for gang scheduling. In many real systems jobs are classified into two classes, that is, interactive and batch jobs. A batch job is one which tends to run much longer and often requires a larger number of processors than interactive ones. Usually batch queues are enabled for execution only during the night. In our experiments we only consider interactive jobs. Job runtimes will have a reasonably wide distribution, with many short jobs but a few relatively large ones and they are rounded to the number of time slots within a range between 1 and 240.

In following sections we present some experimental results. We assume that there are 128 processors in the system. In each experiment we measure the average slowdown and the average number of time slots which are defined as follows:

Assume the execution time and the turnaround time for job $i$ are $t_i^e$ and $t_i^r$, respectively. The slowdown for job $i$ is $s_i = t_i^r / t_i^e$. The average slowdown $s$ is then $s = \sum_{i=0}^{m} s_i/m$ for $m$ being the total number of jobs.

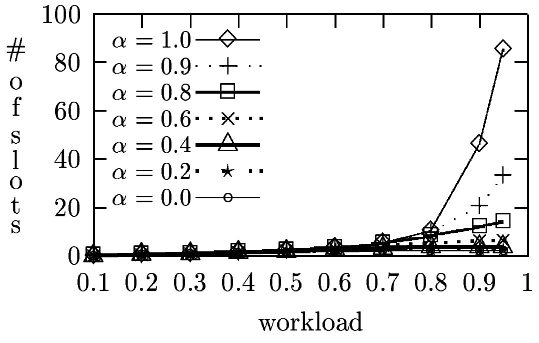
If $t_i$ is the total time when there are $i$ time slots in the system, the average number of time slots in the system during the operation can be defined as $n = \sum_{i=0}^{l} it_i / \sum_{i=0}^{l} t_i$ where $l$ is the largest number of time slots encountered in the system during the computation.

For each estimated system workload, 10 different sets of 20000 jobs were generated using the workload model described above and the final result is the average of these 10 runs.

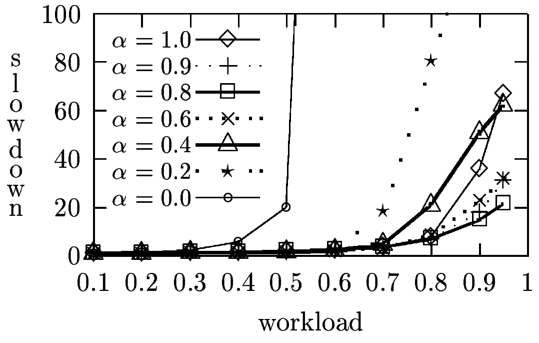
## 4 Experiment One

In our first experiment a FCFS queue is adopted for queuing large jobs. The number of "large" jobs which can be simultaneously executed on the same processors is set to two, that is, $j_{max} = 2$. By varying the second parameter $\alpha$ we are able to determine which job will be queued before being executed.

There are two extreme cases for determining "large" jobs. When $\alpha = 0.0$, every job will be considered "large" and have to enter the system before being executed. In this case the scheduling system acts very



(a)



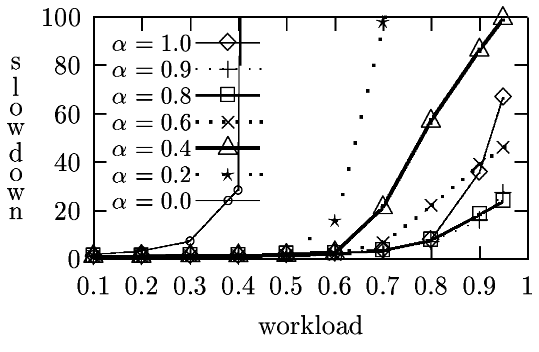
(b)

Figure 1: (a) average number of slots and (b) average slowdown when $j_{max} = 2$.

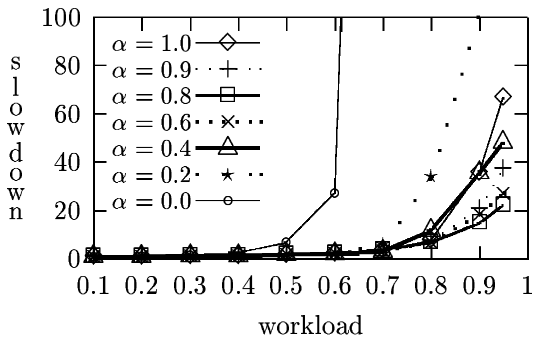
much like a simple FCFS system except the number of time slots can reach two during the computation (because $j_{max} = 2$). When $\alpha = 1.0$, at the other extreme, no job will be considered "large" and every job will be executed immediately on the arrival. This is exactly the same as the conventional gang scheduling system. In our experiment we start from $\alpha = 0.0$, slowly increase $\alpha$ and by comparing other cases with these two extreme ones we are interested to see if the performance can be improved by queuing real large jobs. Some experimental results are given in Fig. 1.

When $\alpha = 0.0$, every job has to be queued before being executed and the number of time slots encountered during the computation will not exceed two no matter how busy the system is. As $\alpha$ increases, more jobs are allowed to run without first entering the waiting queue. This means more jobs will time-share the same processors and then the average number of time slots will increase, especially when the system workload is heavy. It should be noted that the increase in average number of time slots is not significant at the beginning. However, this increase becomes dramatic after $\alpha > 0.6$, especially when $\alpha$ becomes close to 1.0. This is a clear indication that to limit the number of large jobs time-sharing the same processors will greatly decrease the number of time slots in the system.

We have seen from Fig. 1(a) that all other curves are bounded by the two curves, one associated with $\alpha = 0.0$ being the lower bound and the other with $\alpha = 1.0$ being the upper bound. However, this is not the case for average job slowdown, as depicted in Fig. 1(b). The average slowdown for $\alpha = 0.0$ is very high even when the workload is light. As $\alpha$ increases, more jobs are allowed to run without being queued and the performance is improved. After $\alpha$ passes a certain value, however, the performance starts degrading and the degradation of performance becomes significant when $\alpha$ is close to 1.0. This situation may be explained as follows: When $\alpha$ is small, some jobs being queued are not large jobs. Queuing small jobs will degrade the performance in terms of slowdown. When $\alpha$ reaches a certain value, jobs being queued are real large ones. Thus the system can effectively prevent these large jobs from time-sharing processors and the performance is significantly improved. Increasing $\alpha$ after that point, more jobs are allowed to time-share processors. These jobs will compete with each other for resources and each job can only obtain a very small portion of processor utilisation in each scheduling round. In consequence no jobs (even small ones requiring an execution time of only a few time



(a)



(b)

Figure 2: (a) average slowdown when $j_{max} = 1$ and (b) average slowdown when $j_{max} = 3$.

slots) will be completed quickly. This will markedly degrade the overall system performance.

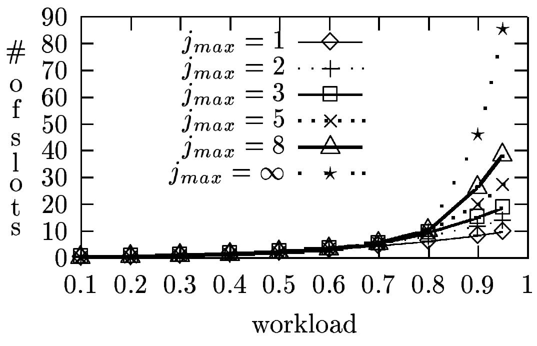
We observed similar results using different $j_{max}$ in our experiment. Fig. 2 shows some experimental results obtained when $j_{max}$ is set to one and three respectively.

## 5 Experiment Two

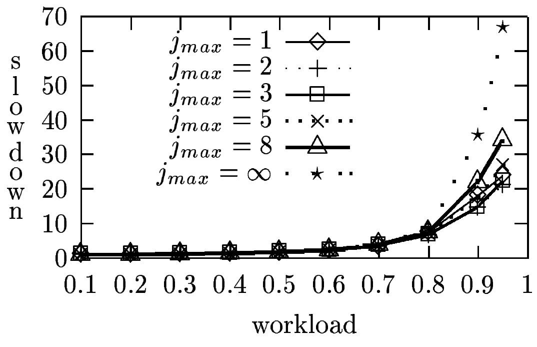
In our first experiment we see that to queue large jobs or to limit the number of large jobs simultaneously running on the same processors can improve the job performance in terms of average slowdown. It should be noted that the resources of a given system are limited and thus the performance gain is actually obtained by giving priority to short jobs and penalising the large ones. This may be reasonable because we normally do not expect a quick response time when submitting a large job. However, the problem is how hard the large jobs are penalised when such a waiting queue is introduced. In our second experiment we set $\alpha = 0.8$ and allow $j_{max}$ to vary from 1 to $\infty$. When the number is set to $j_{max} = 1$, large jobs are not allowed to time-share the same processors. When the number $j_{max}$ becomes very large, however, no large jobs will be queued and it is just a simple gang scheduling system. By varying $j_{max}$ we are interested to see how the performance of large jobs is affected by the simple FCFS queue.

Some results of our second experiment are depicted in Fig. 3. It can be seen that, by limiting the number of large jobs running simultaneously on the same processors the average number of slots in the system will decrease. We see from Fig. 3(b) that the performance continues to improve as $j_{max}$ decreases starting from $j_{max} = \infty$. However, there is one exceptional case, that is, the performance for $j_{max} = 1$ is not as good as that for $j_{max} = 2$, or $j_{max} = 3$. This may be because a simple FCFS waiting queue is used in the system when $j_{max} = 1$ and the blockade problem is thus significant. Consider a simple example in which the first job in the queue requires 64 processors, but currently there are only 32 processors available. This job will then block all the following jobs in the queue from being executed even though some of them require less than 32 processors.

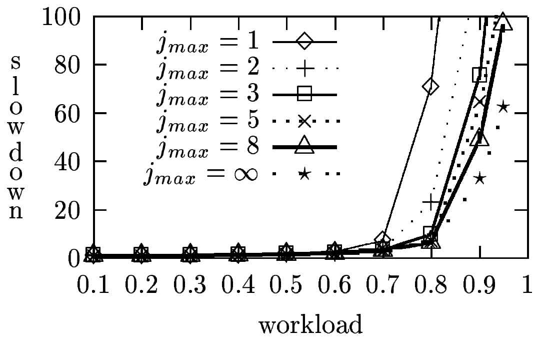
We have seen that the performance in terms of average number of slots and slowdown for all jobs can be improved by limiting the number of large jobs running simultaneously on the same processors. Unfortunately, the performance for large jobs is significantly degraded after the introduction of this simple waiting



(a)



(b)



(c)

Figure 3: (a) Average number of time slots, (b) average slowdown for all jobs and (c) average slowdown for large jobs when a simple FCFS queue is applied.

queue. It can be seen from Fig. 3(c) that slowdown for large jobs is significantly increased as $j_{max}$ decreased, especially when the system workload is heavy. The main reason we believe is still the blockade problem caused by the simple FCFS queue.

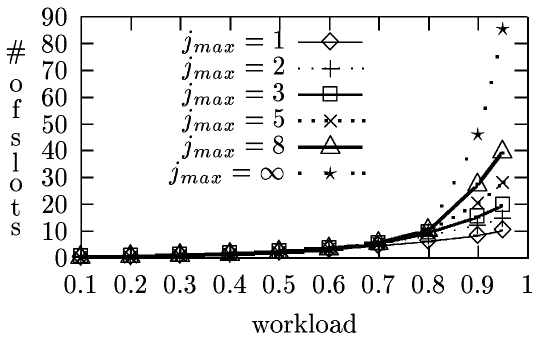
A good way to alleviate the blockade problem is to use backfilling. However, the question is if we can improve the performance of large jobs and at the same time still keep a similar performance for average number of slots as shown in Fig. 3(a) and a similar performance for slowdown for all jobs in Fig. 3(b) after adopting the backfilling technique.

## 6 Experiment Three

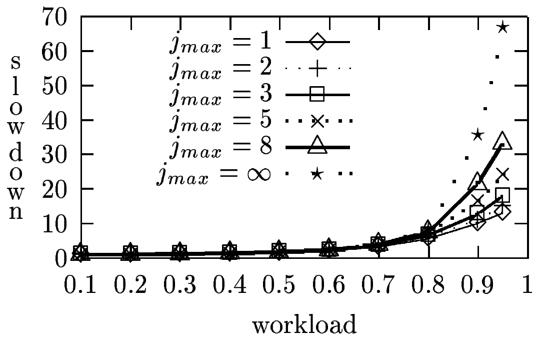
The backfilling technique is adopted in our third experiment. Using the backfilling technique we need to calculate job turnaround time. With gang scheduling, however, it is difficult to accurately measure turnaround time because jobs can run in multiple time slots and the number of time slots in the system may also vary during the computation. As experimental results show, however, inaccuracy in estimation of job turnaround time has little impact on average system performance [12]. Thus in our experiment we compute the turnaround time as a product of the required execution time and the maximum number of time slots currently encountered, and each time when the system working condition changes, i.e., when a large job is terminated, we first check to see if the first large job in the queue can be executed to make sure that the first queued job will not be delayed for too long because of the inaccuracy in measuring turnaround time.

Some experimental results are depicted in Fig. 4. The average number of slots is shown in Fig. 4(a). Comparing this figure with that in Fig. 3(a) we can see that they are about the same. Thus adopting backfilling technique will not greatly increase the average number of time slots in the system.

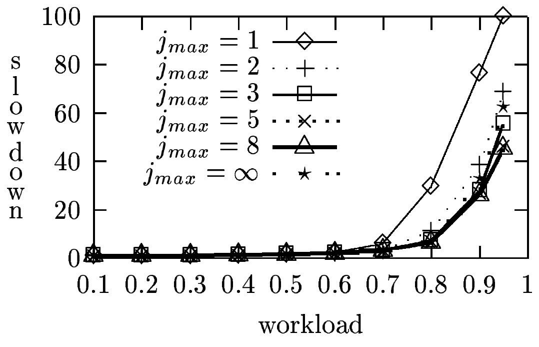
The performance for large jobs is depicted in Fig 4(c). From this figure we can see that initially the performance is improved by increasing $j_{max}$. However, the slowdown will increase if $j_{max}$ is further increased after a certain point. It can be seen from the figure that the slowdown for $j_{max} = \infty$ (simple gang scheduling) is greater than that for $j_{max} \geq 3$. This result is expected and can be explained as follows: When $j_{max}$ is small, the problem of blockade is more significant and increasing $j_{max}$ will alleviate the problem. When $j_{max}$ reaches a certain value (which is 8 for using the particular workload model in our experiment), however, the problem of limited computing power in a



(a)



(b)



(c)

Figure 4: (a) Average number of time slots, (b) average slowdown for all jobs and (c) average slowdown for large jobs when the backfilling technique is applied.

given system will become more significant, that is, if more large jobs are allowed to time-share the same processors, each job can only obtain a very small portion of processor utilisation and then the system performance will be degraded.

Other two points can be made about slowdown by comparing Fig. 4 with Fig. 3. First we have the best average performance for all jobs when $j_{max} = 1$ and with the same $j_{max}$ we have smaller slowdown in our third experiment. Second when backfilling is applied the slowdown for large jobs is greatly decreased. Combining these two facts, we conclude that using the backfilling technique to improve the performance for large jobs does not heavily penalise smaller ones.

## 7 Conclusions

It is known that exclusively space-sharing scheduling can cause blockade problem under heavy workload and that this problem can be alleviated by applying the gang scheduling strategy. Using gang scheduling to simply allow jobs to run simultaneously on the same processors, however, may introduce a large number of time slots in the system. In consequence the cost of context switches will be greatly increased, and each running job can only obtain a small portion of resources including memory space and processor utilisation and so no jobs can complete quickly. Therefore, the number of jobs allowed to run in the system should be limited. The question is which job should be queued so that the overall performance can be improved, or at least will not be significantly degraded in comparison with conventional gang scheduling. In this paper we presented some results obtained from our experiments to show that by limiting real large jobs time-sharing the same processors and applying the backfilling technique we can greatly reduce the average number of time slots in the system and significantly improve the performance of both small and large jobs.

In our experiments we only used one queue for large jobs. To better deal with the problem of resource contention we may need multiple queues, for example, one for large jobs, one for medium-sized jobs and one for small jobs. One question is how to dispatch jobs from these queues to balance the performance of different types of jobs and to achieve the best possible overall performance. This problem will be considered in our future research.

To limit the number of large jobs simultaneously running on the same processors may alleviate memory pressure. This is because in practice a large job usually requires a large memory space. However, simply limiting the job number can only be considered as an indirect method for the problem because it does not directly take memory requirements into consideration. In the future we shall combine memory management with scheduling to solve the problem of memory pressure.

## References

[1] A. Batat and D. G. Feitelson, Gang scheduling with memory considerations, *Proceedings of 14th International Parallel and Distributed Processing Symposium*, Cancun, May 2000, pp.109-114.

[2] A. B. Downey, A parallel workload model and its implications for processor allocation, *Proceedings of 6th International Symposium on High Performance Distributed Computing*, Aug 1997.

[3] A. C. Dusseau, R. H. Arpaci and D. E. Culler, Effective distributed scheduling of parallel workloads, *Proceedings of ACM SIGMETRICS'96 International Conference*, 1996.

[4] D. G. Feitelson, Packing schemes for gang scheduling, In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Lecture Notes Computer Science, Vol. 1162, Springer-Verlag, 1996, pp.89-110.

[5] D. G. Feitelson and L. Rudolph, Job scheduling for parallel supercomputers, in Encyclopedia of Computer Science and Technology, Vol. 38, Marcel Dekker, Inc, New York, 1998.

[6] D. G. Feitelson and L. Rudolph, Gang scheduling performance benefits for fine-grained synchronisation, *Journal of Parallel and Distributed Computing*, 16(4), Dec. 1992, pp.306-318.

[7] D. Lifka, The ANL/IBM SP scheduling system, In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (Eds.), Lecture Notes Computer Science, Vol. 949, Springer-Verlag, 1995, pp.295-303.

[8] J. K. Ousterhout, Scheduling techniques for concurrent systems, *Proceedings of Third International Conference on Distributed Computing Systems*, May 1982, pp.20-30.

[9] J. Skovira, W. Chan, H. Zhou and D. Lifka, The EASY - LoadLeveler API project, In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (Eds.), Lecture Notes Computer Science, Vol. 1162, Springer-Verlag, 1996.

[10] P. G. Sobalvarro and W. E. Weihl, Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors, In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (Eds.), Lecture Notes Computer Science, Vol. 949, Springer-Verlag, 1995.

[11] K. Suzaki, H. Tanuma, S. Hirano, Y. Ichisugi and M. Tukamoto, Time sharing systems that use a partitioning algorithm on mesh-connected parallel computers, *Proceedings of the Ninth International Conference on Distributed Computing Systems*, 1996, pp.268-275.

[12] Y. Zhang, H. Franke, J. E. Moreira and A. Sivasubramaniam, Improving parallel job scheduling by combining gang scheduling and backfilling techniques, *Proceedings of 14th International Parallel and Distributed Processing Symposium*, Cancun, May 2000, pp.133-142.

[13] B. B. Zhou, R. P. Brent, C. W. Johnson and D. Walsh, Job re-packing for enhancing the performance of gang scheduling, *Proceedings of 5th Workshop on Job Scheduling Strategies for Parallel Processing*, San Juan, April 1999, pp.129-143.

[14] B. B. Zhou, D. Walsh and R. P. Brent, Resource allocation schemes for gang scheduling, *Proceedings of 6th Workshop on Job Scheduling Strategies for Parallel Processing*, Cancun, May 2000, pp.45-53.