On the Development of an Efficient Coscheduling System

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 the 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. In this paper we present some experimental results to show that to properly divide jobs into different classes and to apply different scheduling strategies to jobs of different classes can greatly reduce the average number of time slots in the system and significantly improve the performance in terms of average slowdown.

1 Introduction

Scheduling strategies for parallel processing can be classified into either space sharing, or time sharing. Currently most clusters for parallel processing only adopt space-sharing strategies, in which each partitioned processor subset is dedicated to a single job and the job will exclusively occupy the subset until completion. However, one major drawback of space sharing is the blockade situation, that is, short jobs can easily be blocked for a long time by long jobs. Though the backfilling technique can be applied to alleviate this problem to certain extent [5, 7], under heavy workload the blockade can still be a serious problem. As more parallel software packages have been developed for various kinds of applications and more and more ordinary users are getting familiar with multiple processor systems, it is expected that the workload on machines with multiple processors will become heavy in the near future. To alleviate this blockade problem, time-sharing strategies have to be considered.

Because processes of the same parallel job need to coordinate with each other during the computation, coordinated scheduling of parallel jobs across the processors is a critical factor to achieve efficient parallel execution in a time-shared environment. Currently the most popular scheme for coordinated scheduling is explicit coscheduling [6], or gang scheduling [4]. With gang scheduling time is divided into time slots and all parallel jobs, controlled by a global scheduler, take turns to receive the service in a coordinated manner.

One major fundamental problem associated with conventional gang scheduling is resource contention. Currently nearly all time-sharing strategies assume that the resources in a system are unlimited. This assumption is not true and makes the proposed strategies impractical. In a real system the processing speed of processors is limited. If there is a large number of jobs running simultaneously on the same set of processors, no job is able to complete quickly. Because the memory space in a real system is also limited, it is quite possible that the system can run out of memory space if a number of jobs are allowed to run simultaneously and then some jobs have to be paged or swapped out to the secondary memory. However, the experimental results show that simply applying the methods of paging and swapping may seriously harm the process coordination of parallel jobs and thus degrade the system and job performance [1]. Therefore, there is an urgent need to design new time-sharing strategies that take both processor and memory constraints into consideration.

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 [9], 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 processors. 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 reasonably good performance.

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. 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 long. The results of our recent study show that limiting the number of long jobs to time-share the same processors can improve both the average job performance and processor utilisation [12]. To ensure an efficient utilisation of the limited computing power and at the same time to satisfy the performance requirements of various kinds of applications in a give parallel system, therefore, priorities need to be considered and assigned for different jobs.

Our project to develop an effective and practical coscheduling system is divided into three key stages. In the context of gang scheduling computing resources are two dimensional. In designing resource allocation strategies we have to consider resources in both time and space. At the first stage we investigated effective resource allocation (packing) and re-allocation (re-packing) schemes for gang scheduling. We designed a job re-packing strategy for resource re-allocation and time slot reduction. Combining certain existing efficient allocation and re-allocation strategies, we can greatly enhance both resource utilisation and job performance [10, 11].

At the second stage we try to introduce priority scheduling into gang scheduling by dividing jobs into classes, such as, long, medium and short according to their required execution times. Different allocation strategies are then used for jobs of different classes to satisfy performance requirements of different application. For example, we may queue long jobs to limit the number of long ones time-sharing the same processors and to allow short ones to be executed immediately without any delay. The method to classify jobs into classes and treat them differently is not new at all. However, it has not been studied systematically in the context of gang scheduling. We believe that the performance of gang scheduling can be significantly improved by taking the priority scheduling into consideration. Since the computing power is limited, to give one class of jobs a special treatment will no doubt affect the performance of jobs in other classes. A hard question is how to design scheduling strategies such that the performance of jobs in one class can be improved without severely punishing the others.

To solve the problem of memory pressure we need to consider scheduling and memory management simultaneously. Another advantage of dividing jobs into classes is that we are able to choose a particular type of jobs for paging and swapping to alleviate the memory pressure without significantly degrade the overall job performance. Therefore, in our future work, that is, the third stage of our project we will consider to combine memory management with gang scheduling to directly solve the problem of memory pressure.

In this paper we shall present some simulation results from our second stage research, to show that, by properly classifying jobs (which are generate from a particular workload model) and choosing different scheduling strategies to different classes of jobs, we are able to improve the overall performance without severely degrading the performance of long jobs.

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. Finally the conclusions are given in Section 5.

2 Our Experimental 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 [10, 11].

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 [3, 8]. 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 [3].

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 [11].

Our experimental system is based on the gang scheduling system described above. In this experimental system, however, jobs are classified and limits are set to impose restriction on how many jobs are allowed to run simultaneously on the same processors.

To classify jobs we introduce two parameters. Assume the execution time of the longest job is t^e. A job will be considered "long" in each test if its execution time is longer than $\alpha_l t^e$ for $0.0 \leq \alpha_l \leq 1.0$. A job is considered "medium" if its execution time is longer than $\alpha_m t^e$, but shorter than or equal to $\alpha_l t^e$ for $0.0 \leq \alpha_m \leq \alpha_l$. Otherwise, the job will be considered "short". By varying these two parameters we are able to make different job classifications and to see how different classifications affect the system performance.

We introduce a waiting queue for medium and long jobs in our coscheduling system. To alleviate the blockade problem the backfilling technique is adopted. Because the backfilling technique is applied, a long job in front of the queue will not block the subsequent medium sized jobs from entering the system. Therefore, one queue is enough for both classes of jobs. A major advantage of using a single queue for two classes of jobs is that the jobs will be easily kept in a proper order based on their arriving times. Note that in our experimental system short jobs can be executed immediately on their arrivals without any delay.

To conduct our experiments we further set two other parameters. One parameter k_m is the limit for the number of both medium and long jobs to be allowed to time-share the same processors. If the limit is reached, the incoming medium and long jobs have to be queued. The other parameter k_l is the limit for the number of long jobs to be allowed to run simultaneously on the same processors. If that limit is reached, the incoming long jobs have to be queued. By varying these two parameters we are able to see how the added queue affects the system performance.

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 long 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 $n = \sum_{i=0}^{l} i t_i / \sum_{i=0}^{l} t_i$ where l is the largest number of

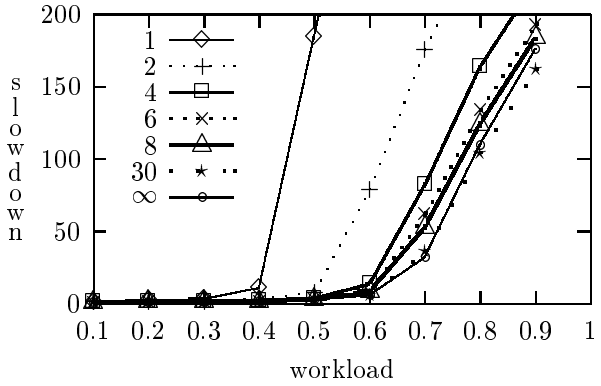


Figure 1: Average slowdown when $\alpha_m = 0.0$ and $\alpha_l = 1.0$.

time slots encountered in the system during the computation.

For each estimated system workload, 10 different sets of 10000 jobs were generated using the workload model described above and the final result is the average of these 10 runs.

4 Experimental Results

We conducted four different experiments. Some of our experimental results are presented in the following subsections.

4.1 Experiment one

In our first experiment α_m and α_l are fixed, that is, $\alpha_m = 0.0$ and $\alpha_l = 1.0$. With this setting all jobs are treated as equal and they may have to enter the queue before being executed if k_m is not set to infinity. The number of jobs to be allowed to time-share the same processor is determined by k_m. Thus the system performance will be affected by varying this parameter. Some experimental results for average slowdown are given in Fig. 1.

When $k_m = 1$, it is just a simple FCFS scheduling system with backfilling. It can be seen from this figure that the slowdown is very dramatically increased after workload becomes greater than 0.4. Therefore, exclusively space-sharing scheduling can only perform well under light workload. When the system workload becomes heavy, time sharing should be considered.

When k_m is increased, the performance is improved as indicated in the figure. After k_m reaches certain

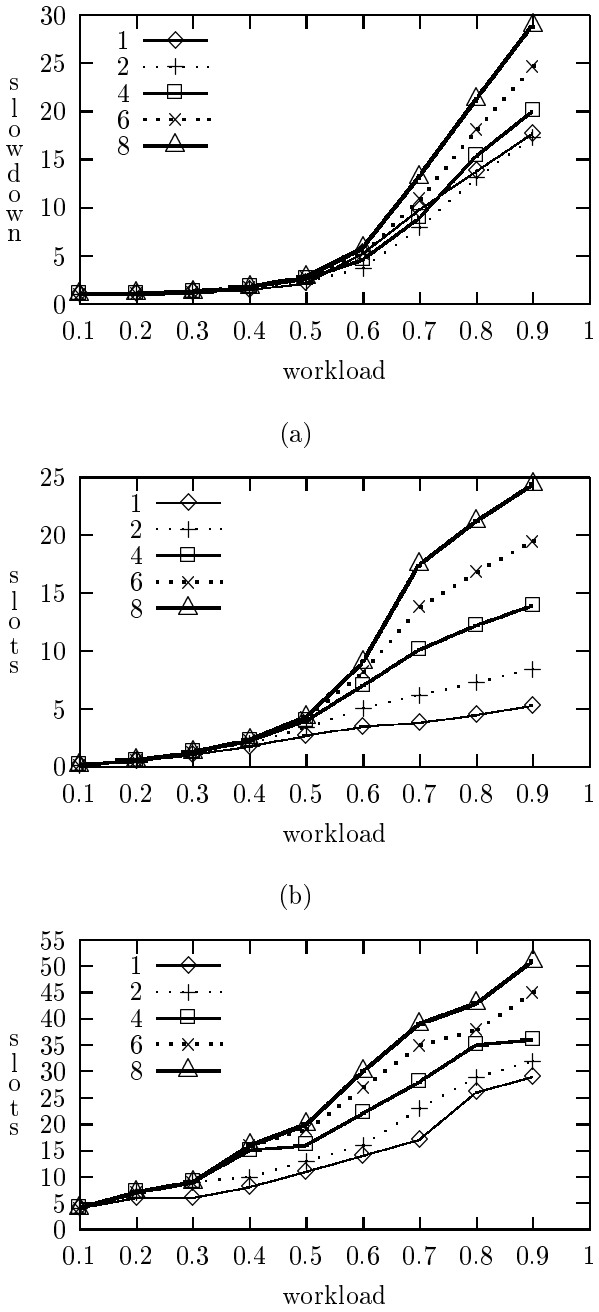


Figure 2: (a) Average slowdown, (b) average number of time slots and (c) maximum number of time slots, when $\alpha_m = 0.8$ and $\alpha_l = 1.0$.

value, however, further increase in k_m will not lead to a great increase in performance. An interesting point is that the simple gang scheduling system (by setting $k_m = \infty$) will not perform as well as one with $k_m = 30$ when the system workload becomes heavy. This is because the computing power of a given system is limited. If too many jobs time-share the same set of processors, each job can only obtain a very small portion of processor utilisation and no job can complete quickly. Thus the system performance will be degraded.

4.2 Experiment two

In our first experiment jobs are not distinguished based on their execution times. Though the performance is improved by increasing k_m, the maximum number of jobs which are allowed to time-share the same processors, the improvement is not that significant.

As mentioned in Section 1, the simple round robin scheduling strategy will not perform well when a number of long jobs are running simultaneously on the same processors. To demonstrate this in our second experiment we set $\alpha_m = 0.8$ and $\alpha_l = 1$, which means jobs are divided into two classes, that is, "long" jobs whose execution time is longer than $0.8t^e$ for t^e the execution time of the longest job, and "short" jobs whose execution time is shorter than or equal to $0.8t^e$. By varying k_m we can determine how many long jobs can run simultaneously on the same processors.

Some experimental results are depicted in Fig. 2. We can see in the figure that the average slowdown, the average number of slots and the maximum slot number (i.e. the maximum number of jobs which are running simultaneously on the same processors during the computation) are all reduced when k_m decreases. It is a clear indication that limiting the number of long jobs to run simultaneously on the same processors can indeed improve overall system performance.

Comparing Fig. 2(a) with Fig. 1, we see that a much smaller average slowdown can be obtained by only queueing long jobs, but allowing other jobs to run immediately on their arrivals. As depicted in Fig. 1, for example, the average slowdown will become greater than 150 when the workload is 0.9. However, it is a very good result when the method of combining the gang scheduling with backfilling is applied. By queueing only long jobs, the average slowdown can be even lower than 20 with $k_m = 1$ (or $k_m = 2$), which is a significant improvement.

It can also been seen from Fig. 2(b) that for $k_m = 1$ the average number of slots is only about 5 when the system workload is 0.9. thus queueing long jobs can also decrease the average number of time slots in contrast with the conventional gang scheduling.

4.3 Experiment three

Although the average slowdown is significantly decreased by queueing long jobs in gang scheduling, the maximum slot number encountered during the computation is relatively high in contrast to the strategy which queues every incoming job once a hard limit for the number of time slots is reached. The question is if we can produce similar performance with reduced maximum slot number.

In our third experiment we first set $\alpha_m = 0.0$, $k_m = 6$ and $k_l = 1$, that is, we set a limit for maximum number of time slots to 6 and another limit for long jobs which can time-share the same processors to 1. Thus the maximum slot number will never exceed 6 during the computation. By varying α_l we can determine what jobs should be considered as long such that a good performance can be obtained by blocking them from running simultaneously on the same processors. Some experimental results are depicted in Fig. 3(a).

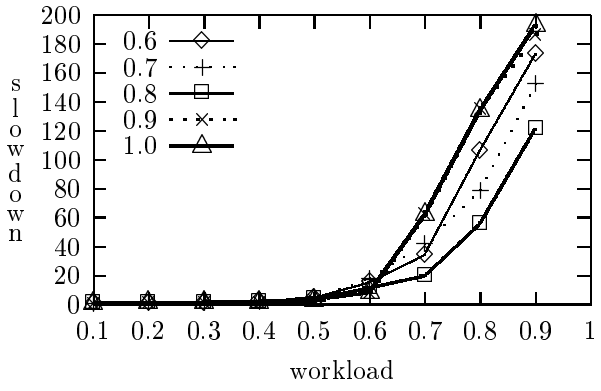
When $\alpha_l = 1.0$, no jobs will be treated as long. This is the same as that in our first experiment by combining the gang scheduling with backfilling and then setting $k_m = 6$. The performance is first improved with α_l decreased starting from 1.0. However, further decreasing α_l will cause an increase in average slowdown. We can see from the figure that the best performance is obtained when $\alpha_l = 0.8$.

Next we set $\alpha_m = 0.0$, $k_m = 6$ and $\alpha_l = 0.8$. We want to see how the system performs by varying k_l. Some experimental results are depicted in Fig. 3(b) and (c). It is clearly shown in the figures that to allow more long jobs to time-share the same processors can only degrade the performance.

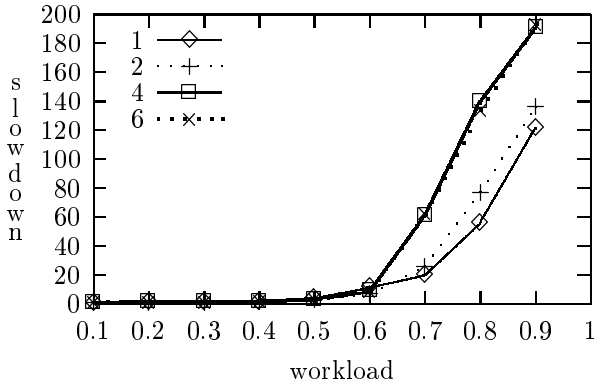
4.4 Experiment four

The results obtained from the third experiment is not desirable, that is, these results are not as good as those obtained by only queueing long jobs in the second experiment. In contrast with the results obtained by using the combination of the gang scheduling and backfilling in our first experiment, however, both the average slowdown and the average number of slots are reduced if we set limits both for all jobs and for long jobs to run simultaneously on the same processors.

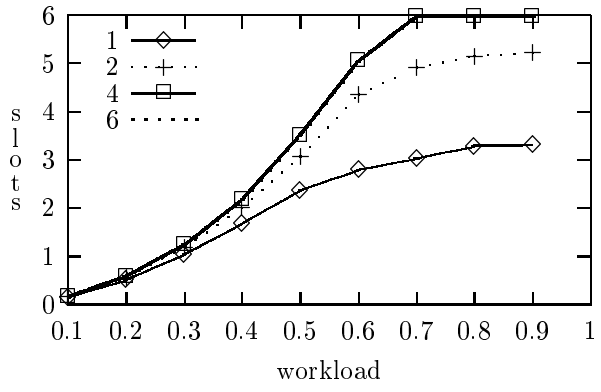
In our fourth experiment we still set $\alpha_l = 0.8$, $k_m = 6$ and $k_l = 1$, that is, the same set of long jobs will be limited to time-share the same processors



(a)

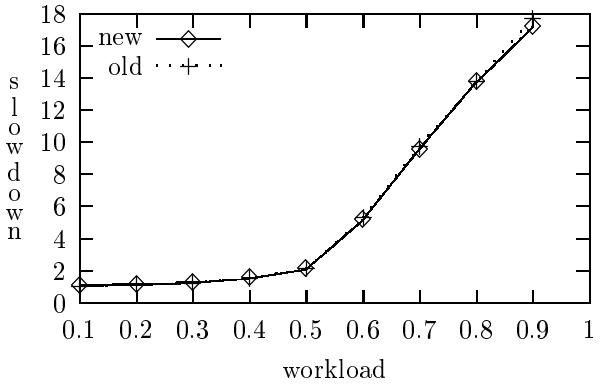


(b)

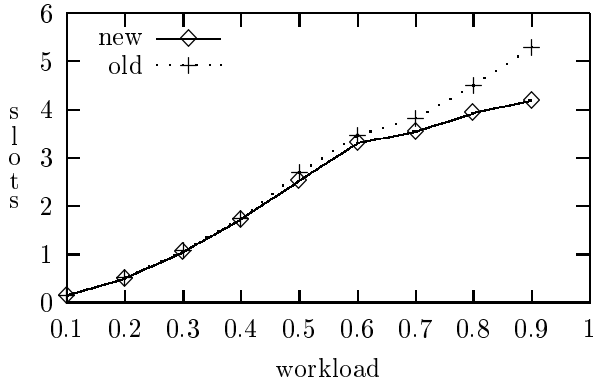


(c)

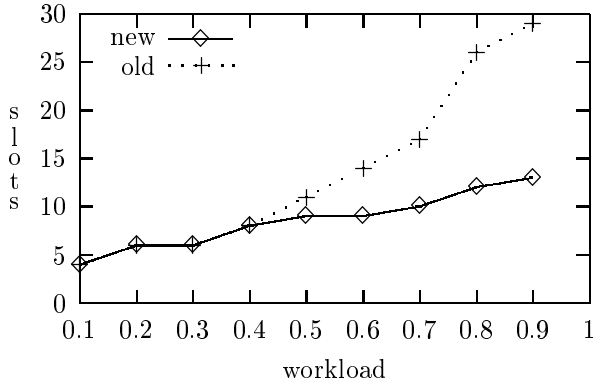
Figure 3: (a) Average slowdown when $\alpha_m = 0.0$, $k_m = 6$ and $k_l = 1$, (b) average slowdown and (c) average number of slots when $\alpha_m = 0.0$ and $\alpha_l = 0.8$ and $k_m = 6$.



(a)



(b)



(c)

Figure 4: Comparison of the new results (new) with the best results (old) obtained from experiment two. (a) average slowdown, (b) average number of slots and (c) maximum number of slots.

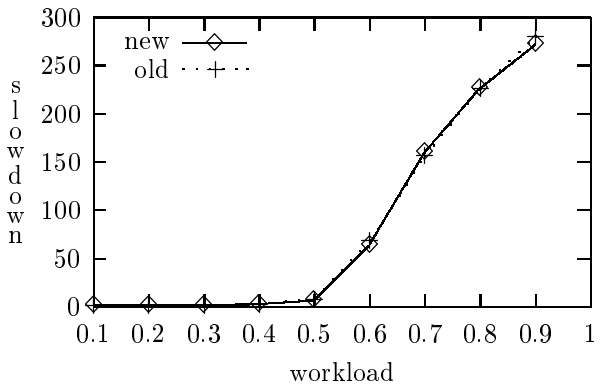


Figure 5: Comparison of the two strategies (old and new) in average slowdown for long jobs

as that in the third experiment. However, we allow a (small) number of real short jobs to run immediately on their arrivals by setting $\alpha_m = 0.15$. In this way we hope that the number of time slots will not be increased greatly during the computation and at the same time the average slowdown will significantly be reduced. Some experimental results (new) are depicted in Fig. 4. In order to provide a clearer view about the performance, the best results (old) obtained by setting $\alpha_m = 0.8$ and $k_m = 1$ in our second experiment, are also presented in the figure. We can see that the two strategies (old and new) are comparable in terms of average slowdown. Under heavy system workload, however, a smaller average number of time slots and a much smaller maximum number of time slots is obtained in our fourth (or new) experiment.

Since a number of short jobs are allowed to run immediately without delay, the performance of long jobs might be severely degraded. As depicted in Fig. 5, however, we find that no serious performance degradation for long jobs in terms of slowdown occurs in our experiment. To allow short jobs to run immediately may enhance the system utilisation. This may be the main reason why the overall system performance is enhanced in our fourth experiment.

5 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 what kind of jobs should be queued so that the overall performance can be improved, or at least will not be significantly degraded in comparison with the conventional gang scheduling. In this paper we presented some results obtained from our experiments to show that to properly divide jobs into different classes and to apply different scheduling strategies to jobs of different classes can greatly reduce the average number of time slots in the system and significantly improve the performance in terms of average slowdown.

In our experiments we showed that a good overall system performance can be obtained by first classifying jobs into short, medium and long and then using conventional gang scheduling for short, the combination of the gang and backfilling for medium and the combination of the FCFS and backfilling for long jobs.

Although the average number of time slots is significantly reduced, which may greatly alleviate memory pressure, our method can only be considered as an indirect method for the problem of memory pressure because it does not directly take memory requirements into consideration. In our future research in the development of an efficient and practical coscheduling system we shall combine memory management with scheduling to directly 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] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] B. B. Zhou and R. P. Brent, Gang scheduling with a queue for large jobs, accepted by *15th International Parallel and Distributed Processing Symposium*, San Francisco, April 2001.