An Efficient Resource Allocation Scheme for Gang Scheduling

B. B. Zhou & P. Mackerras
ANU-Fujitsu CAP Project
Australian National University
Canberra, ACT 0200, Australia

C. W. Johnson & D. Walsh
Department of Computer Science
Australian National University
Canberra, ACT 0200, Australia

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

Abstract

Gang scheduling is currently the most popular scheduling scheme for parallel processing in a time shared environment. One major drawback of using gang scheduling is the problem of fragmentation. The conventional method to alleviate this problem is to allow jobs running in multiple time slots. However, our experimental results show that simply applying this method alone cannot solve the problem of fragmentation, but on the contrary it may eventually degrade the efficiency of system resource utilisation. In this paper we introduce an efficient resource allocation scheme which effectively incorporates the ideas of re-packing jobs, running jobs in multiple slots and minimising time slots into the buddy based system to significantly improve the system and job performance for gang scheduling. Because there is no process migration involved in job re-packing, this scheme is particularly suitable for clustered parallel computing systems.

1. Introduction

With the rapid developments in both hardware and software technology the performance of scalable systems such as clusters of workstations/PCs/SMPs has significantly been improved. It is expected that this kind of system will dominate the parallel computer market in the near future because of the continued cost-effective growth in performance. For this type of machine to be truly utilised as general-purpose high-performance computing servers for various kinds of applications, effective job scheduling facilities have to be developed to achieve high efficiency of resource utilisation.

Many job scheduling schemes have been introduced for parallel computing systems. (See a good survey in [4].) These scheduling schemes can be classified into either *space sharing*, or *time sharing*. Because a time shared environment is more difficult to establish for parallel processing in a multiple processor system, currently most commercial parallel systems only adopt space sharing schemes such as the LoadLeveler scheduler from IBM for the SP2 [8]. However, 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. Thus time sharing schemes need 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. Currently the most popular scheme for coordinated scheduling is *explicit coscheduling* [6], or *gang scheduling* [5]. With gang scheduling processes of the same job will run simultaneously for a certain amount of time, which is called the *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. All parallel jobs in the system take turns to receive the service in a coordinated manner. If space permits, a number of jobs may be allocated in the same time slot and run simultaneously on different subsets of processors. Thus gang scheduling can be considered as a scheduling scheme which combines both space sharing and time sharing together.

One disadvantage associated with conventional gang scheduling for clustered (or networked) computing systems is its purely centralised control for context switches across the processors. That is, a central controller is used to broadcast messages to all the processors telling which job should be scheduled next. When the size of a system is large, efficient space partitioning policies are not easily incorporated, mainly due to this frequent broadcasting. To deal with this problem we designed a new coscheduling scheme

called *loose gang scheduling*, or *scalable gang scheduling* [11, 12]. Using our scheduling scheme the disadvantages associated with conventional gang scheduling are significantly alleviated, especially the requirement for frequent broadcasting. The basic structure of this scheduling scheme has been implemented on a 16-processor Fujitsu AP1000+. Although the function of the current coscheduling system is limited and needs to be further enhanced, the preliminary experimental results show that the scheme works as expected [10]. This enables us to consider more effective methods for resource allocation to significantly enhance the performance of gang scheduling.

Currently most allocation schemes 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 in this kind of resource allocation is the problem of fragmentation. Because resource allocation is considered independently in different time slots, some freed resources due to job termination 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 whenever possible [2, 9]. When jobs are allowed to run in multiple time slots, the buddy based allocation scheme will perform much better than many other existing allocation schemes in terms of average job turnaround time [2].

The buddy based scheme was originally developed for memory allocation [7]. To allocate resources to a job of size p using the buddy based scheme, the processors in the system are first divided into subsets of size n for $n/2 < p \leq n$. The job is then assigned to one such subset if there is a time slot in which all processors in the subset are idle. Although the buddy scheme causes the problem of internal fragmentation, jobs with about the same size tend to be head-to-head aligned in different time slots. If one job is completed, the freed resources can easily be reallocated to other jobs running on the same subset of processors. Therefore, jobs have a better chance to run in multiple time slots. An interesting point is that we cannot guarantee the improvement in system resource utilisation by simply running jobs in multiple time slots. We shall show that simply running jobs in multiple time slots may eventually degrade the efficiency in system resource utilisation unless special care is taken.

To alleviate the problem of fragmentation we proposed another scheme, namely job re-packing [13]. In this scheme we try to rearrange the order of job execution on their originally allocated processors so that small fragments of idle resources from different time slots can be combined together to form a larger and more useful one in a single time slot. 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 resource allocation and also to balance the workload across the processors.

In this paper we introduce an efficient resource allocation scheme. This scheme effectively incorporates the techniques of re-packing jobs, running jobs in multiple slots and minimising the time slots into the buddy based system to significantly enhance system and job performance. In Section 2 we first describe the workload model used in our experiments. The ideas of job re-packing and workload tree for the buddy based system are then presented in Section 3. Section 4 gives some experimental results which show that simply running jobs in multiple time slots cannot solve the problem of fragmentation, but on the contrary may eventually degrade the efficiency of system resource utilisation. Our efficient allocation scheme is described and some experimental results are also presented in Section 5. Finally the conclusions are given in Section 6.

2. The Workload Model

In our experiments we adopted a workload model proposed in [1]. 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 120. Assuming the length of a time slot is five seconds, the longest job will then be 10 minutes and the average job length is about two minutes. In the experiment we also assume that there are 128 processors in the system.

We are more interested in the transient behaviors, rather than the steady state of a system. In the experiment each time only a small set of 200 jobs were used to evaluate the performance of each scheduling scheme. For each estimated system workload, however, 20 different sets of jobs

were generated using the workload model and the final results are the average of the 20 experiments for each scheduling scheme.

During the simulation we collect the following statistics:

- average processor active ratio r_a: the average number of time slots in which a processor is active divided by the overall system computational time in time slots. If the resource allocation scheme is efficient, the obtained result should be close to the estimated average system workload ρ which is defined as $\rho = \lambda \bar{p} \bar{t} / P$ where λ is job arrival rate, $\bar{t}$ and $\bar{p}$ are the average job length and size and P is the total number of processors in the system.

- average number of time slots n_a: 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 $n_a = \sum_{i=0}^{n_l} i t_i / \sum_{i=0}^{n_l} t_i$ where n_l is the largest number of time slots encountered in the system during the computation.

- average turnaround time t_a: The turnaround time is the time between the arrival and completion of a job. In the experiment we measured the average turnaround time t_{ta} for all 200 jobs. We also divided the jobs into three classes, that is, small (between 1 and 12 time slots), medium (between 13 and 60) and large (greater than 60) and measured the average turnaround time for these classes, t_{sa}, t_{ma} and t_{la}, respectively.

3. Job Re-Packing and Workload Tree

In the following discussion we assume that processors in a parallel system are *logically* organised as a one-dimensional linear array. Note that the term *one-dimensional linear array* is purely defined in the gang scheduling context. A logical *one-dimensional array* is defined as a set of N processors which are enumerated from 1 to N (or from 0 to $N-1$) regardless of their physical locations in the system. Thus we can simply use a two-dimensional global scheduling matrix such as the one in Fig. 1. Using the term *linear array* we mean that only consecutively numbered processors can be allocated to a given job. Thus this kind of regularity is only associated with the global scheduling matrix, but not with the physical locations of processors.

One way to alleviate the problem of fragmentation is to allow jobs to run in multiple time slots whenever possible. A simple example is depicted in Fig. 1. In this example the system has eight processors and originally three slots are created to handle the execution of nine jobs. Now assume that two jobs J' and J'' in slot S_2 are terminated. If jobs are allowed to run in multiple time slots, jobs J_1 and J_2 in

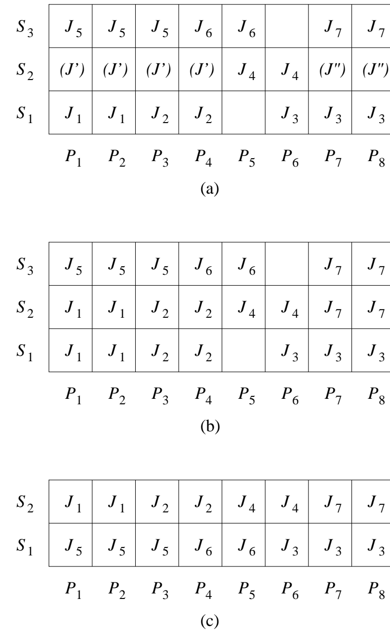


Figure 1. An example of alleviating the fragmentation problem by (b) running jobs in multiple time slots and (c) re-packing jobs to reduce the total number of time slot.

slot S_1 and job J_7 on S_3 can occupy the freed resources in S_2, as shown in Fig. 1(b). Therefore, most processors can be kept busy all the time. However, this kind of resource reallocation may not be optimal when job performance is considered. Assume now there arrives a new job which requires more than one processor. Because the freed resources have been reallocated to the running jobs, the fourth time slot has to be created and then the performance of the existing jobs which run in a single time slot will be degraded.

Now consider job re-packing. We first shift jobs J_1 and J_2 from slot S_1 to slot S_2 and then move jobs J_5 and J_6 down to slot S_1 and job J_7 to slot S_2. After this rearrangement or re-packing of jobs, time slot S_3 becomes completely empty. We can then eliminate this empty slot, as shown in Fig. 1(c). It is obvious that this type of job re-packing can greatly improve the overall system performance. Note that during the re-packing jobs are only shifted between rows from one time slot to another. We actually only rearrange the order of job execution on their originally allocated processors in a scheduling round and there is no process migration between processors involved. This kind of job rearrangement is particularly suitable for clustered parallel machines in which process migration is expensive.

Since processes of the same job need coordination and they must be placed in the same time slots all the time during the computation, therefore, we cannot re-pack jobs in

an arbitrary way. A shift is said to be legal if all processes of the same job are shifted to the same slot at the same time. In job re-packing we always utilise this kind of legal shift to rearrange jobs between time slots so that small fragments of idle resources in different time slots can be combined into a larger and more useful one.

When processors are logically organised as a one-dimensional linear array, we have two interesting properties which are described below. (The proofs of these properties can be found in [13].

Property 1 *Assume that processors are logically organised as a one-dimensional linear array. Any two adjacent fragments of available processors can be grouped together in a single time slot.*

Property 2 *Assume that processors are logically organised as a one-dimensional linear array. If every processor has an idle fragment, jobs in the system can be re-packed such that all the idle fragments will be combined together in a single time slot which can then be eliminated.*

Based on job re-packing we can set up a *workload tree* (WLT) for the buddy scheduling system, as depicted in Fig. 2, to balance the workload across the processors and also to simplify the search procedure for resource allocation.

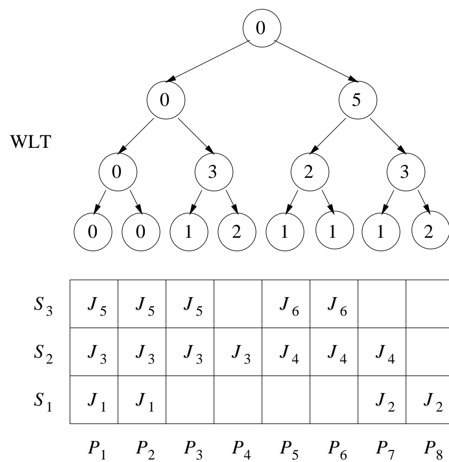


Figure 2. The binary workload tree (WLT) for the buddy based allocation system.

The workload tree has $logN + 1$ levels for N the number of processors in the system. Each node in the tree is associated with a particular subset of processors. The node at the top level is associated with all N processors. The N processors are divided into two subsets of equal size and each subset is then associated with a child node of the root. The division and association continues until the bottom level is reached. Each node in the tree is assigned an integer value. At the bottom level the value assigned to each leaf node is equal to the number of idle time slots on the associated processor. For example, the node corresponding to processor P_1 is given a value 0 because there is no idle slot on that processor, while the value assigned to the last node is equal to 2 denoting there are currently two idle slots on processor P_8. For a non-leaf node the value will be equal to the sum of the values of its two children when both values are nonzero. Otherwise, it is set to zero denoting the associated subset of processors will not be available for new arrivals.

For the conventional allocation method, adding this workload tree may not be able to assist the decision making for resource allocation. This is because the information contained in the tree does not tell which slot is idle on a processor, but processes of the same job have to be allocated in the same time slot. With job re-packing, however, we know that on a one-dimensional linear array any two adjacent fragments of available processors can be grouped together to form a larger one in a single time slot according to Property 1. To search for a suitable subset of available processors, therefore, we only need to check the values at a proper level. Consider the situation depicted in Fig. 2 and assume that a new job of size 4 arrives. In this case we need only to check the two nodes at the second level. Since the value of the second node at that level is nonzero (equal to 5), the new job can then be placed on the associated subset of processors, that is, the last four processors. To allocate resources we first re-pack job J_6 into time slot S_1 and then place the new job in time slot S_3. Since the workload conditions on these processors are changed after the allocation, the values of the associated nodes need to be updated accordingly.

There are many other advantages in using this workload tree. To ensure a high system and job performance it is very important to balance workloads across the processors. Using the workload tree it will become much easier for us to handle the problem of load balancing. Because the value of each node reflects the information about the current workload condition on the associated processor subset, the system can easily choose a subset of less active processors for an incoming job by comparing the node values at a proper level.

To enhance the efficiency of resource utilisation jobs should be allowed to run in multiple time slots if there are free resources available. Although the idea of running jobs in multiple time slots was originally proposed in [2, 9], there were no methods given on how to effectively determine whether an existing job on a subset of processors can run in multiple time slots. Using the workload tree this procedure becomes simple. In Fig. 2, for example, the rightmost two nodes at the third level of the workload tree are nonzero and jobs J_2 and J_6 are currently running within each of the

scheme	ρ	r_a	n_l	n_a	t_{ta}	t_{sa}	t_{ma}	t_{la}
BC	0.20	0.19	3	1.16	31.10	5.67	40.52	112.45
BR		0.19	3	0.45	30.01	5.52	39.29	108.18
BRMS		0.19	4	0.57	29.25	5.37	37.86	106.56
BC	0.50	0.46	6	2.84	70.00	14.04	89.27	246.08
BR		0.46	5	2.27	58.65	11.68	75.23	206.45
BRMS		0.45	15	6.11	57.28	15.99	73.83	184.98
BC	0.70	0.55	10	5.23	129.65	25.03	166.21	456.72
BR		0.58	8	4.09	102.21	20.27	130.17	359.00
BRMS		0.53	30	14.39	96.95	35.78	128.23	271.12
BC	0.90	0.58	14	7.62	189.60	35.91	246.73	670.42
BR		0.65	11	6.00	150.18	29.50	195.49	526.64
BRMS		0.56	43	20.17	120.53	51.84	170.61	356.94

Table 1. Some experimental results 1.

two associated subsets of processors, respectively. These two jobs can then be allocated an additional time slot and run in multiple time slots.

Since the root of the workload tree is associated with all the processors, we are able to know quickly when a time slot can be deleted by simply checking the node value. If it is nonzero, we immediately know that there is at least one idle slot on each processor. According to Property 2 these idle fragments can be combined together in a single time slot which can then be eliminated.

4. Running Jobs in Multiple Time Slots

In this section we present some experimental results obtained from implementing three different allocation schemes to show that simply running jobs in multiple time slots may not solve the problem of fragmentation. The first one is just the conventional buddy (BC) system in which the workload balancing is not seriously considered and each job only runs in a single time slot. The second scheme (BR) utilises the workload tree to balance the workload across the processors and re-packs jobs when necessary to reduce the average number of time slots in the system, but it does not consider to run jobs in multiple time slots. The third allocation scheme (BRMS) is a modified version of the second one, in which jobs are allowed to run in multiple time slots whenever possible. When a job is given an extra time slot in this scheduling scheme, it will keep running in multiple time slots to completion and never relinquish the extra resources gained during the computation.

Some experimental results are given in Table 1. First consider that jobs only run in a single time slot. When job re-packing is applied to reduce the number of time slots in the system and the workload tree is used to balance the workload across the processors, we expect that both job performance and system resource utilisation should be improved. Our experimental results confirm this prediction. It can be seen from the table that scheme BR consistently outperforms BC under all categories although the improvement is not significant for the estimated system workload $\rho = 0.20$.

It is seen that schemes BRMS which allows jobs to run in multiple slots can reduce the average turnaround time t_{ta}. This is understandable since a job running in multiple slots may have a shorter turnaround time. An interesting point, however, is that applying BRMS will result in a much longer average turnaround time for short jobs. The main reason why BRMS can cause a longer average turnaround time for short jobs may be as follows: If jobs are allowed to run in multiple slots and do not relinquish additional slots gained during the computation, the number of time slots in the system may become very large most of the time. Note that long jobs will stay in the system longer and then have a better chance to run in multiple time slots. However, the system resources are limited. When a short job arrives, it can only obtain a very small portion of CPU utilisation if allocated only in a single time slot.

It seems that we can increase the average processor active ratio if jobs are allowed to run in multiple time slots. However, another interesting point is that using the allocation scheme BRMS will eventually decrease the efficiency in resource utilisation. As shown in Table 1 the average processor active ratio can even be lower than that obtained by using the conventional buddy scheduling scheme BC. The main reason may be that, when a job running in multiple slots finishes, the processors on which it was running will be idle in those multiple time slots until a change in the workload condition occurs, such as a new job arriving to fill the freed resources, or some slots becoming totally empty which can be eliminated.

5. Efficient Allocation Scheme

It can be seen from Table 1 that simply running jobs in multiple time slots will greatly increase the number of time slots in the system. To confirm that this great increase in system slot numbers is the main cause for the degradation of system performance, We designed two allocation schemes which consider the reduction of the number of time slots in the system while allowing jobs to run in multiple slots.

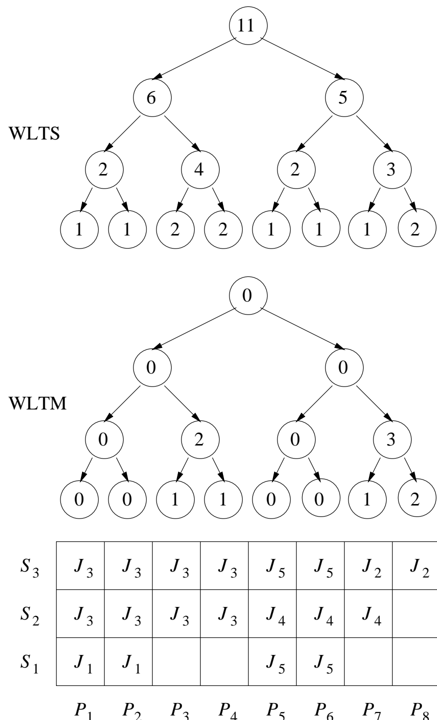


Figure 3. Two workload trees (WLTS and WLTM) used for reducing the number of time slots in the system.

Because the system tries to reduce the number of time slots, jobs running in multiple time slots may have to relinquish the additional resources gained during the computation. Now the problem is how to determine when a time slot should be deleted and when a job should be allowed to run in multiple time slots. These two issues usually conflict with each other. To deal with this problem we set up two workload trees WLTM and WLTS. An example is depicted in Fig. 3. The two trees are the same except their node values. If all the jobs are running in a single time slot, the node values in both trees are exactly the same. When some jobs, for example, jobs J_3 and J_5 in Fig. 3, are given extra time slots, we only update the node values in tree WLTM, but leave WLTS intact.

Since WLTM contains the information on the actual workload condition in the system, it can be used to determine when a job can be given an extra time slot. For example, in Fig. 3 the rightmost node at the third level is nonzero and job J_2 is running within the associated subset of processors. If there are no new arrivals, J_2 can run in both time slots S_1 and S_3.

Tree WLTS does not have any information about running jobs in multiple time slots. If it is used to make resource allocation decisions, however, some jobs have to relinquish extra time slots. Since the value of the root node is nonzero in Fig. 3, for example, we can then delete a time slot in the system. To achieve this job J_2 is first shifted to slot S_1 and then jobs J_3 and J_5 relinquish one time slot they gained during the computation. Thus slot S_3 will become completely empty and can be deleted.

To reduce time slots in the system our first allocation scheme BRMMS* works as follows: The workload tree WLTM is used to allocate resources to new arrivals and to determine when a job can run in multiple time slots. This is exactly the same as that in BRMS. However, the workload tree WLTS is also used to determine when a time slot can be eliminated. Thus the average number of time slots in the system will become smaller than the number created by using BRMS.

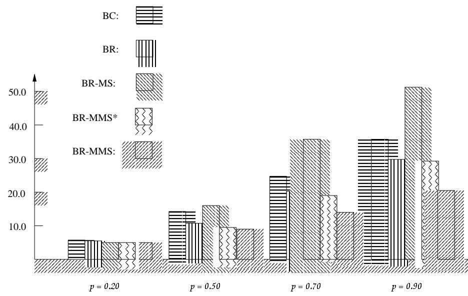
In BRMMS* jobs relinquish their extra time slots only when a time slot is to be eliminated. However, our second allocation scheme BRMMS is more vigorous in reducing time slots. This scheme works more like allocation scheme BR. It uses workload tree WLTS to allocate the resources to new arrivals and to determine the reduction of time slots in the system. The workload tree WLTM is only used to determine when running jobs can run in multiple time slots if there are no new arrivals. Thus jobs may have to relinquish the extra time slots when a new job arrives, but it cannot be allocated in the existing time slots. Therefore, we can expect that the average number of time slots in the system will never be greater than the number created by using scheme BR.

Some experimental results are given in Table 2. The results obtained by using BRMS are also relisted in the table. It is easy to see from the table that reducing time slots in the system can significantly improve the performance. We can also see that scheme BRMMS performs much better than BRMMS* under all categories. To enhance the system and job performance, therefore, it is more important to minimise the number of time slots in the system than to simply run jobs in multiple time slots.

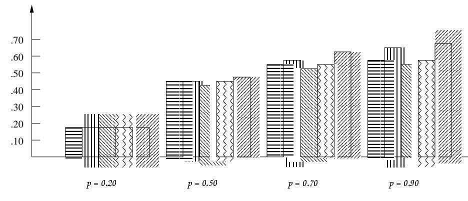
In order to give a better view for the comparison we show two pictures for average turnaround time for short jobs t_{sa} and average processor active ratio r_a in Fig. 4. We can see from the figure that using scheme BRMS will result in a long average turnaround time for short jobs and a low average processor active ratio and the results are even worse

scheme	ρ	r_a	n_l	n_a	t_{ta}	t_{sa}	t_{ma}	t_{la}
BRMS	0.20	0.19	4	0.57	29.25	5.37	37.86	106.56
BRMMS*		0.19	3	0.45	28.81	5.26	37.21	105.38
BRMMS		0.19	3	0.44	28.66	5.24	37.09	104.68
BRMS	0.50	0.45	15	6.11	57.28	15.99	73.83	184.98
BRMMS*		0.46	7	2.78	49.19	9.98	62.18	178.27
BRMMS		0.47	5	2.06	44.05	8.77	55.82	159.75
BRMS	0.70	0.53	30	14.39	96.95	35.78	128.23	271.12
BRMMS*		0.55	13	6.14	81.47	18.50	103.76	280.47
BRMMS		0.61	7	3.58	66.23	13.96	83.19	234.05
BRMS	0.90	0.56	43	20.17	120.53	51.84	170.61	356.94
BRMMS*		0.58	20	9.86	117.71	29.07	153.45	391.67
BRMMS		0.68	10	5.51	98.51	20.80	124.69	345.48

Table 2. Some experimental results 2.



(a)



(b)

Figure 4. (a) Average turnaround time for small jobs t_{sa} and (b) Average processor active ratio r_a.

than those obtained by using a very simple buddy allocation scheme BC. We conclude that, to ensure a high system and job performance, simply running jobs in multiple time slots should be avoided.

Although BRMMS* performs better than BRMS, it can only produce a worse processor active ratio than BR. Note that in the allocation scheme BR jobs only run in a single time slots, which may result in a low average number of time slots in the system. This gives another clear indication of the importance of reducing time slots in the system.

It can be seen from the above tables and pictures that BRMMS is the best of the five allocation schemes. It consistently outperforms all other schemes under all categories. To improve job and system performance, jobs should be allowed to run in multiple time slots so that free resources can be more efficiently utilised. However, simply running jobs in multiple time slots cannot guarantee the improvement of performance. The minimisation of time slots in the system has to be seriously considered.

6. Conclusions

One major drawback of using gang scheduling for parallel processing is the problem of fragmentation. A conventional way to alleviate this problem was to allow jobs to run in multiple time slots. However, simply adopting this idea alone may cause several problems. The first obvious one is the increased system scheduling overhead. This is because simply running jobs in multiple time slots can greatly increase the average number of time slots in the system and then the system time will be increased to manage a large number of time slots. The second problem is the unfair treatment to small jobs. Long jobs will stay in the system for relatively a long time and then have a better chance to run in multiple time slots. However, the system resources are limited and in consequence a newly arrived short job may only obtain relatively a very small portion of CPU utilisation. Another very interesting point obtained from our experiment is that simply running jobs in multiple time slots may not solve the problem of fragmentation, but on the contrary it may eventually degrade the efficiency of system resource utilisation.

We can see from our experimental results that the minimisation of time slots in the system is very important to ensure a high system and job performance. We thus

highly recommend our resource allocation scheme BRMMS. This scheme effectively combines the techniques of job re-packing, running jobs in multiple time slots and minimising time slots in the system together so that job turnaround times are greatly reduced and the efficiency of system resource utilisation is significantly enhanced. Because there is no process migration involved when job re-packing is applied, this scheme is particularly suitable for clustered parallel computing systems.

It should be noted that in our experiment we assumed that the memory space is unlimited and characteristics of jobs are totally unknown. In practice, however, the size of memory in each processor is limited. Thus jobs may have to come to a waiting queue before being executed and large running jobs may have to be swapped when the system becomes busy. Along with the rapid development of high-performance computing libraries, characteristics of jobs may no longer be considered completely unknown before being executed. These conditions will be considered in our future research.

References

[1] 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.

[2] 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.

[3] D. G. Feitelson and L. Rudolph, Distributed hierarchical control for parallel processing, *Computer*, 23(5), May 1990, pp.65-77.

[4] 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.

[5] 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.

[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. L. Peterson and T. A. Norman, Buddy systems, *Comm. ACM*, 20(6), June 1977, pp.421–431.

[8] 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.

[9] 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.

[10] D. Walsh, B. B. Zhou, C. W. Johnson and K. Suzaki, The implementation of a scalable gang scheduling scheme on the AP1000+, *Proceedings of the 8th International Parallel Computing Workshop*, Singapore, Sep. 1998, P1-G-1 – P1-G-6.

[11] B. B. Zhou, R. P. Brent, D. Walsh and K. Suzaki, Job scheduling strategies for networks of workstations, In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (Eds.), Lecture Notes Computer Science, Vol. 1459, Springer-Verlag, 1998.

[12] B. B. Zhou, X. Qu and R. P. Brent, Effective scheduling in a mixed parallel and sequential computing environment, *Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing*, Madrid, Jan 1998.

[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.81–92.