Resource Allocation Schemes for Gang Scheduling

B. B. Zhou[1], D. Walsh[2], and R. P. Brent[3]

[1] School of Computing and Mathematics, Deakin University, Geelong, VIC 3217, Australia
[2] Department of Computer Science, Australian National University, Canberra, ACT 0200, Australia
[3] 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. In this paper we first describe the ideas of job re-packing and workload tree for efficiently allocating resources to enhance the performance of gang scheduling. We then present some experimental results obtained by implementing four different resource allocation schemes. These results show how the ideas, such as re-packing jobs, running jobs in multiple slots and minimising the average number of time slots in the system, affect system and job performance when incorporated into the buddy based allocation scheme for gang scheduling.

1 Introduction

Many job scheduling strategies have been introduced for parallel computing systems. (See a good survey in [4].) These scheduling strategies 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 such as the LoadLeveler scheduler from IBM for the SP2 [9]. 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. For parallel machines to be truly utilised as general-purpose high-performance computing servers for various kinds of applications, time sharing has to be seriously 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 strategy for coordinated scheduling is *explicit coscheduling* [7], or *gang scheduling* [5]. With gang scheduling 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. 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 strategy which combines both space sharing and time sharing together.

Currently most 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 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, 10]. 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 [8]. 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.

To alleviate the problem of fragmentation we proposed another scheme, namely job re-packing [11]. In this scheme we try to rearrange the order of job execution on the 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 shall present some simulation results to show how the ideas, such as re-packing jobs, running jobs in multiple slots and minimising the number of time slots in the system, affect system and job performance when incorporated into the buddy scheduling system. In Section 2 we briefly discuss job re-packing. The construction of the binary workload tree for the buddy based system is described in Section 3. Section 4 first discusses four different allocation schemes to be compared and the workload model used in our experiments and then presents some simulation results. Finally the conclusions are given in Section 5.

2 Job Re-Packing

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.

S_3	J_5	J_5	J_5	J_6	J_6		J_7	J_7
S_2	(J')	(J')	(J')	(J')	J_4	J_4	(J'')	(J'')
S_1	J_1	J_1	J_2	J_2		J_3	J_3	J_3
	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8

(a)

S_3	J_5	J_5	J_5	J_6	J_6		J_7	J_7
S_2	J_1	J_1	J_2	J_2	J_4	J_4	J_7	J_7
S_1	J_1	J_1	J_2	J_2		J_3	J_3	J_3
	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8

(b)

S_2	J_1	J_1	J_2	J_2	J_4	J_4	J_7	J_7
S_1	J_5	J_5	J_5	J_6	J_6	J_3	J_3	J_3
	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8

(c)

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

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 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 distributed memory 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 available processors in different time slots can be combined into a larger and more useful one. This kind of job re-packing can effectively be done based on the following two simple properties [11].

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.

Note that adopting job re-packing may increase the scheduling overhead in a clustered computing system because messages notifying the changes in the global scheduling matrix have to be broadcast to processors so that the local scheduling tables on each processor can be modified accordingly. However, there is no need to frequently re-pack jobs between time slots. The re-packing is applied only when the working condition is changed, e.g., when a job is terminated, or when a new job arrives. Thus the extra system cost introduced by the re-packing may not be high. In the next section we shall see that, when job re-packing is incorporated in the buddy based system, we can set up a workload tree. With this workload tree the procedure for searching available resources can be simplified and then the overall system overhead for resource allocation is actually reduced.

3 Workload Tree

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.

The workload tree has $logN + 1$ levels where N is 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

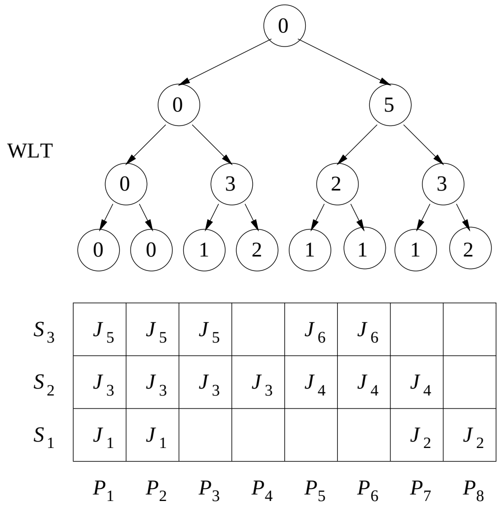

Fig. 2. The binary workload tree (WLT) for the buddy based allocation system.

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 presented in the previous section. 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 may 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, 10], 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 node at the third level of the workload tree is nonzero and job J_2 is currently running within the associated subset of processors. It can then be allocated an additional time slot (S_3 in this case) and run in multiple time slots.

To enhance the system and job performance it is also important to minimise the number of time slots in the system. (See our experimental results presented in the next section.) 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 presented in the previous section these idle fragments can be combined together in a single time slot which can then be eliminated. Assume that job J_1 in Fig. 2 is terminated. The values of the leaf nodes associated with processors P_1 and P_2 become nonzero. This will cause the value of their parent node to become nonzero. The information about the change of workload condition is continuously propagated upward and then the root value will become nonzero. It is easy to see in this particular example that, after job J_2 is legally shifted to time slot S_3, time slot S_1 will become completely empty and can then be deleted.

4 Experiments

In this section we present some experimental results to show how the techniques of re-packing jobs, running jobs in multiple slots and minimising the average number of time slots in the system, affect system and job performance when incorporated into the buddy based allocation system for gang scheduling.

4.1 Allocation schemes

Four different resource allocation schemes are evaluated in the experiment. 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. The fourth allocation scheme (BRMMS) is designed to consider the minimisation of the average number of time slots in the system while allowing jobs to run in multiple slots. In this scheme jobs running in multiple time slots may have to relinquish the additional resources gained during the computation if a new arrival cannot fit into the existing time slots, or if a time slot in the system can be deleted. Therefore, we can expect that the average number of time slots in the system will never be greater than the number created by using the second scheduling scheme BR.

4.2 The workload model

Experimental results show that the choice of workload alone does not significantly affect the relative performance of different resource management algorithms [6]. To compare the performance of the above four different resource allocation schemes, we adopted one 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 second, the longest job will then be 10 minutes and the average job length is about two minutes.

4.3 Results

We assume that there are 128 processors in the system. 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

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
BRMMS		0.19	3	0.44	28.66	5.24	37.09	104.68
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
BRMMS		0.47	5	2.06	44.05	8.77	55.82	159.75
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
BRMMS		0.61	7	3.58	66.23	13.96	83.19	234.05
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
BRMMS		0.68	10	5.51	98.51	20.80	124.69	345.48

Table 1. Some experimental results obtained in the first experiment.

operation can be defined as $n_a = \sum_{i=0}^{n_l} it_i / \sum_{i=0}^{n_l} t_i$ 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.

We conducted two experiments. In our first experiment we measured transient behaviors of each system. 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 runs for each scheduling scheme.

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 $\rho = 0.20$.

When jobs are allowed to run in multiple time slots, situations become a bit more complicated. We can see that both n_l and n_a are dramatically increased

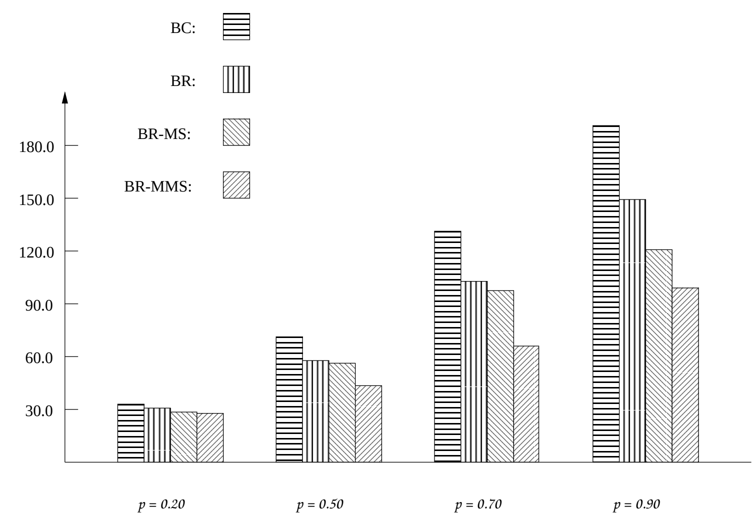

Fig. 3. Average turnaround time for all jobs t_{ta}.

when the third scheduling scheme BRMS is adopted. In order to give a better view for the comparison we show three pictures for average turnaround time for all jobs t_{ta}, average turnaround time for short jobs t_{sa} and average processor active ratio r_a, respectively.

The average turnaround time for all jobs is depicted in Fig. 3. It is seen that schemes BRMS and BRMMS which allow jobs to run in multiple slots can reduce 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 as shown in Fig. 4. From the user's perspective it is short jobs that need to be completed more quickly. 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 Fig. 5 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 time slots

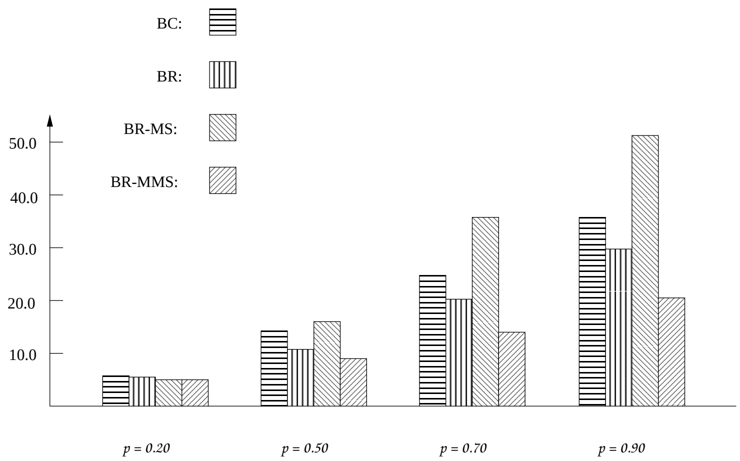

Fig. 4. Average turnaround time for small jobs t_{sa}.

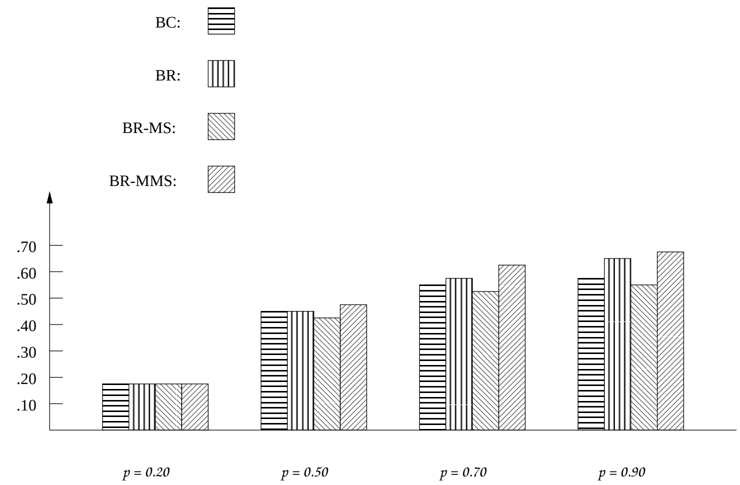

Fig. 5. Average processor active ratio r_a.

until a change in workload condition occurs such as a new job arriving to fill the freed resources, or some slots becoming totally empty which can be eliminated.

In our second experiment we measured the steady state performance of each system by increasing the number of jobs in each job set to 20,000. Some experimental results are depicted in Table 2. The results are obtained by taking the average of 5 runs using different job sets. It can be seen that the allocation scheme BR performs much better than BRMS. This conforms that simply running jobs in multiple time slots can eventually decrease the efficiency of system resource utilisation and degrade the overall performance of jobs.

scheme	ρ	r_a	n_l	n_a	t_{ta}	t_{sa}	t_{ma}	t_{la}
BC	0.20	0.20	6	1.21	32.15	5.70	40.62	114.25
BR		0.20	6	0.46	30.97	5.52	39.16	109.77
BRMS		0.20	7	1.20	30.99	5.51	39.11	110.15
BRMMS		0.20	6	0.45	29.58	5.21	37.20	105.60
BC	0.50	0.49	18	3.64	91.09	17.33	115.54	317.90
BR		0.49	13	2.60	68.32	12.95	86.93	237.94
BRMS		0.49	17	3.78	78.78	15.64	99.75	272.78
BRMMS		0.49	12	2.30	48.93	9.47	61.25	172.04
BC	0.70	0.68	79	35.28	874.81	166.87	1097.78	3077.22
BR		0.69	26	7.21	177.75	34.23	224.62	620.55
BRMS		0.69	64	23.37	441.42	91.33	551.02	1531.94
BRMMS		0.69	23	5.39	94.78	20.10	118.61	326.51
BC	0.90	0.69	645	346.33	8713.93	1603.65	10955.74	30834.47
BR		0.85	123	62.48	1151.24	296.40	1952.93	5438.99
BRMS		0.82	357	202.71	3836.59	800.92	4830.35	13191.87
BRMMS		0.86	90	39.94	716.46	151.14	887.45	2490.94

Table 2. Some experimental results obtained in the second experiment.

It can be seen from the above tables and pictures that BRMMS is the best of the four allocation schemes. It consistently outperforms all other three 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 average number of time slots in the system has to be seriously considered.

5 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.

With job re-packing we try to rearrange the order of job execution on their originally allocated processors to combine small fragments of available processors into a larger and more useful one. Based on job re-packing we can set up a workload tree to greatly improve the performance for the buddy scheduling system. With the workload tree we are able to simplify the search procedure for resource allocation, to balance the workload across the processors and to quickly detect when a job can run in multiple time slots and when the number of time slots in the system can be reduced. More importantly we are able to properly combine the ideas of job re-packing, running jobs in multiple time slots and minimising the average number of time slots in the system together to reduce job turnaround times and to enhance the efficiency of system resource utilisation. Our experimental results show that this combined allocation scheme, i.e., our fourth allocation scheme BRMMS can indeed improve the system and job performance significantly. Because there is no process migration involved, 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. V. Lo, J. Mache and K. Windisch, A comparative study of real workload traces and synthetic workload models for parallel job scheduling, In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (Eds.), Lecture Notes Computer Science, Vol. 1459, Springer-Verlag, 1998, pp.25-46.
7. J. K. Ousterhout, Scheduling techniques for concurrent systems, *Proceedings of Third International Conference on Distributed Computing Systems*, May 1982, pp.20-30.
8. J. L. Peterson and T. A. Norman, Buddy systems, *Comm. ACM*, 20(6), June 1977, pp.421-431.
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. 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.
11. 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.