Job Re-Packing for Enhancing the Performance of Gang Scheduling

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

[1] Computer Sciences Laboratory, Australian National University, Canberra, ACT 0200, Australia
[2] Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK
[3] Department of Computer Science, Australian National University, Canberra, ACT 0200, Australia

Abstract. This paper presents some ideas for efficiently allocating resources to enhance the performance of gang scheduling. We first introduce a job re-packing scheme. In this scheme we try to rearrange the order of job execution on their originally allocated processors in a scheduling round to combine small fragments of available processors from different time slots together to form a larger and more useful one in a single time slot. We then describe an efficient resource allocation scheme based on job re-packing. Using this allocation scheme we are able to decrease the cost for detecting available resources when allocating processors and time to each given job, to reduce the average number of time slots per scheduling round and also to balance the workload across the processors.

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.

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* [4], or *gang scheduling* [3]. With gang scheduling processes of the same job will run simultaneously for only certain amount of time which is called *scheduling slot*. When a scheduling 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.

Because there are multiple processors in a system, the resource allocation will include both space partitioning and time sharing in the gang scheduling context. One disadvantage associated with the 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 frequently broadcast messages to all the processors telling which job should obtain the service next. When the size of a system is large, efficient space partitioning policies are not easily incorporated mainly due to this frequent signal broadcasting. 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. To ensure a high efficiency of resource utilisation, however, we believe that allocation of resources in both time and space has to be considered simultaneously.

We have designed a new coscheduling scheme called *loose gang scheduling*, or *scalable gang scheduling* [7, 8]. Using our scheduling scheme the disadvantages associated with conventional gang scheduling are significantly alleviated, especially the requirement for frequent signal-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 [6]. This enables us to consider the allocation of resources in both space and time at the same time in a more effective way to significantly enhance the performance of gang scheduling.

In this paper we present some resource allocation schemes for achieving high system and job performance. We first give some simple examples in Section 2 to show that regularity is still an important factor in designing resource allocation policies even for a parallel system with an unstructured interconnection or fully connected pattern between processors, and that resource allocation in different time slots should be considered at the same time to achieve a higher efficiency in resource utilisation. It is known that allocation schemes which take regularity into account can cause the problem of *fragmentation*. In Section 3 we introduce a job re-packing scheme. Using this scheme small fragments in different time slots can under certain conditions be combined together to form a much larger and useful one in a single time slot. Based on job re-packing we then describe in Section 4 a simple and practical resource allocation scheme which considers the workload conditions in both space and time dimensions at the same time (rather than just in each individual time slot) when allocating resources to a given job. With this scheme we are able to reduce the average number of time slots per scheduling round, to balance workloads across the processors and thus to achieve high system and job performance.

2 Motivation

To design efficient processor allocation policies for conventional distributed-memory MPPs we should consider two important factors, *regularity* and *locality*, in order to minimise communication costs and to avoid communication contention between different jobs. On certain parallel machines such as clusters of workstations or PCs or SMPs interconnected via a Gigabit Ethernet or a crossbar switch network, however, the communication costs may not depend on the location of processors. Thus regularity and locality may become less important issues when only space partitioning is considered. In this case a simple *random allocation* scheme may be preferable, that is, we can arbitrarily choose a set of available processors to a given job regardless of their localities. With this simple scheme we may alleviate the problem of fragmentation caused by those allocation schemes taking regularity into consideration.

The question is if the simple random allocation scheme can also be incorporated in gang scheduling to efficiently allocate resources in each time slot. It seems that the answer to this question is positive. When a new job arrives, we first search to see if there are enough available processors in an existing time slot. If there are, a set of available processors regardless of their localities is allocated to that job. A new time slot will be created if we cannot find enough available resources in any existing time slot.

S_4	J_4	J_4				J_4		
S_3	J_3		J_3			J_3		
S_2						J_2		J_2
S_1		J_1	J_1		J_1			J_1
	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8

Fig. 1. One possible situation caused by using the random allocation scheme.

The problem, however, is not associated with how to allocate resources in each time slot to new jobs, rather it is with how to effectively reuse freed resources due to the job termination if there are no new arrivals. Let us consider a simple example. Originally the system was very busy and four time slots had to be created. After a certain period of time small jobs were all terminated and there left only a few large jobs scattered across the processors in different time slots, as shown in Fig. 1. We know that the processes of the same parallel job need coordination during the computation. Because processors are arbitrarily allocated to jobs in each time slot, one possible situation, as depicted in Fig. 1, is that neither the total number of time slots can be reduced, nor can the freed resources be efficiently reallocated to those running jobs even though processors in the system are only active less than fifty percent of time on the average. The simulation results presented in [1] show that this situation can significantly be improved if a well known regular allocation strategy such as the *first fit*, or the *buddy* is adopted instead. The main reason is that, when regularity is taken into account, the number of unification counts can greatly be increased, jobs will have better chances to run in multiple time slots and then both the performance of parallel jobs and the efficiency of resource utilisation can be enhanced. (The number of *unification counts* is defined as the number of times the same set of processors in two time slots are united and allocated to a single parallel job [1] and a parallel job running in more than one time slot is sometimes called *multi-slice* job [5].)

It can be seen from the above discussion that to design efficient job allocation strategies for gang scheduling regularity is an important factor. Many existing schemes for space partitioning take regularity into consideration. By simply adopting one of such schemes and allocating resources independently in each time slot as we conventionally do, however, efficient resource utilisation may still not be guaranteed.

First regular allocation schemes can have the problem of small fragments, that is, a fragment of available processors in a time slot is too small to be allocated to any job. The second main problem associated with the conventional allocation method is that space partitioning and time sharing are not considered simultaneously. Consider two space partitioning schemes, that is, the *first fit* and the *buddy*, which are widely used in Gang scheduling for resource allocation in each time slot. Although the internal fragmentation caused by the buddy allocation scheme may be more serious than the external fragmentation caused by the first fit, simulation results show that the DHC allocation scheme [2] which is based on the buddy performs better than those based on other regular allocation schemes if slot unification is allowed [1]. The main reason may be as follows: With the buddy allocation scheme each time processors are divided into two subsets of equal size. If the same policy is applied to every time slot, two jobs allocated to different subsets of processors, whether in the same time slot or not, will never overlap with each other. When a job is terminated, the freed resources in one time slot are more likely to be united with those in another time slot and then reallocated to a single job. Thus space partitioning and time sharing are implicitly (though not thoroughly) considered at the same time. Since the allocation of resources is essentially independent in different time slots and small internal fragments cannot be grouped together, however, using a simple buddy based allocation scheme may still be difficult to achieve an optimal solution in resource utilisation.

3 Job Re-Packing

We cannot totally avoid the problem of fragmentation in each time slot when regularity is taken into consideration. However, we have to adopt regular allocation schemes in order to achieve a better system performance as discussed in the previous section. The question is thus if we can find a way to alleviate the fragmentation problem. In this section we introduce a job re-packing scheme. Using this scheme we are able to combine certain small fragments from different time slots into a larger and more useful one in a single time slot.

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 regularity is only associated with the global scheduling matrix, but not with the physical locations of processors.

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		
S_1						J_3	J_3	J_3
	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8

(b)

S_3							J_7	J_7
S_2	J_1	J_1	J_2	J_2	J_4	J_4		
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)

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

(d)

Fig. 2. Job re-packing to reduce the total number of time slot in a scheduling round.

We first give a simple example, as shown in Fig. 2, to demonstrate the basic ideas of our re-packing scheme. 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. When using the idea of unification, certainly jobs J_1 and J_2 in slot S_1 (or J_5 on S_3) and job J_7 on S_3 can occupy the freed resources in S_2 to become multi-slice jobs. This might be the best assignment if the performance of these jobs were just required to be enhanced. However, the assignment may not be optimal from the overall system performance point of view because there are still some small fragments which are not utilised and other running jobs in the system cannot obtain any benefit. Things become worse if there arrives a new job which requires six or more processors – the fourth time slot has to be created and then the performance of the existing jobs will be degraded.

With certain rearrangement or re-packing of jobs, however, we can eliminate one time slot under the same situation. The procedure is as follows: First we reallocate jobs J_1 and J_2 from slot S_1 to slot S_2, as shown in Fig. 2(b). After this step two fragments in S_1 and S_2 are combined into a larger one in slot S_1. This new fragment can further be combined with the one in slot S_3 by taking jobs J_5 and J_6 down to slot S_1, as shown in Fig. 2(c). Finally we simply bring job J_7 down to slot S_2. Slot S_3 now becomes empty and can then be removed. It is obvious that this type of job re-packing can greatly improve the overall system performance. Note that during the re-packing only processes of the same job are shifted from one time slot to another. Therefore, this kind of re-packing is actually to rearrange the order of job execution on their originally allocated processors in a scheduling round and there is no process migration between processors involved.

Because processes of the same job require to coordinate with each other during the computation, all processes of the same job must be shifted together to the same time slot at the same time. Thus there is a restriction to shift jobs during the re-packing. In Fig. 2(a), for example, processes of J_4 on S_2 cannot be shifted to either S_1 or S_3 because the size of the fragment in either slot is not big enough to accommodate all the processes of J_4. A shift is said to be legal if all processes of a job are shifted to the same slot at the same time. In job re-packing we try to use 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.

When processors are logically organised as a one-dimensional linear array, we have two interesting properties which are described below.

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.

Proof: It is trivial when two fragments are adjacent in the same slot. We thus assume that the two adjacent fragments are in different time slots, either sharing a common vertical boundary, or partially overlapping with each other.

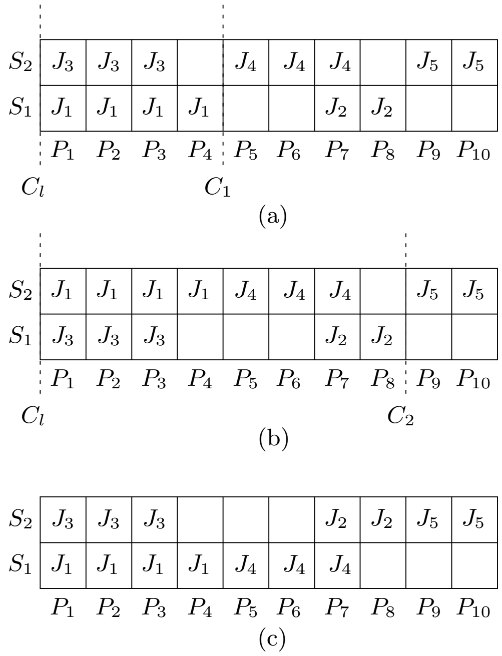


Fig. 3. Small fragments combined into a larger one in a single time slot.

Define a *cut* as a vertical line which is set between two processors and across certain (not necessarily consecutive) time slots in the global scheduling matrix to cut those slots into two parts. A cut is said to be a *legal cut* if it does not cut existing jobs into two parts, that is, all the existing jobs will have their two boundaries of the allocated processors on the same side of the cut. Let us introduce two legal cuts through two time slots in a given system. Then all the jobs bounded by these two cuts in one slot can legally exchange their positions at the same time with their counterpart in the other slot. This is because every job bounded between these legal cuts will have all its processes in the bounded region and they will still be in the same slot after the exchange. In the following discussion all cuts will be considered as legal cuts.

Because it is a one-dimensional linear array, its left (or right) end will form a natural boundary and no jobs can come across. We can thus set our first legal cut there, as shown in Fig 3(a). Our second cut will be set between the two fragments. It is also a legal cut because no jobs can reside on both sides of this cut when regularity is considered in the processor allocation. Thus jobs bounded by these two legal cuts can exchange their positions, which enables the two fragments to be grouped together in a single slot, as depicted in Fig. 3(b).

In the above example the two small fragments share a common boundary. When two fragments partially overlap with each other, our second legal cut can be set anywhere between the overlapped region and then a larger fragment can be produced. □

The proof is constructive, that is, it describes an algorithm for job re-packing. In this algorithm we use the left (or right) end of the array as a legal cut, then set another legal cut between the adjacent fragments and finally exchange jobs bounded by the two cuts.

We can continue this process in Fig. 3(b) by setting another cut and then four original small fragments in the two slots will be reduced to two larger ones in two such re-packing steps.

When we can introduce more than one cuts in the middle of the array through two time slots at the same time, however, the above algorithm will not be very efficient. In Fig. 3, J_1 and J_3 are swapped twice and then back to their original positions. Thus these exchanges are redundant and should be avoided. It is easy to verify that the following algorithm will work more efficiently: First find all the legal cuts which divide the slots into several regions and then swap only once the jobs between the two slots in alternating regions. The proof is simple and omitted. For the same problem as that in Fig. 3 we can simultaneously introduce two cuts to divide the array into three regions and then need only to swap jobs in the middle region to obtain the same result, as depicted in Fig. 4.

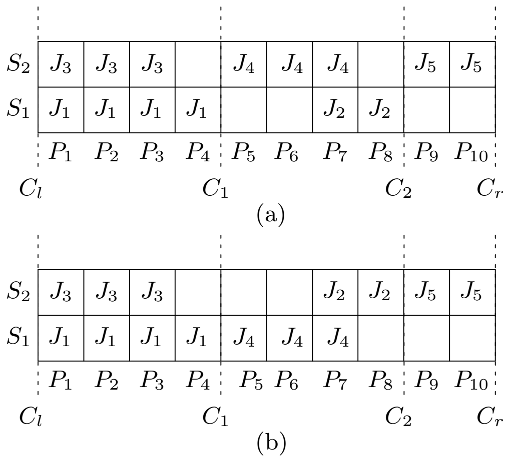


Fig. 4. A more efficient way for job re-packing between two time slots.

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.

Proof: It is actually a simple extension of Property 1. We have already given an example, as depicted in Fig. 2. We can easily prove that the property holds for general case by the following simple induction.

Assume that the first k small fragments on the first k processors have been combined together as a single fragment of size k in time slot S_i and that the fragment on processor P_{k+1} is in time slot S_j. Setting a cut at one end of the array and a cut between the two processors P_k and P_{k+1}, we can then combine the two fragments into one of size $k+1$ in either S_i or S_j according to Property 1. □

In the above discussion we assumed that processors are organised as a one-dimensional linear array. To merge two adjacent fragments actually only one cut is required because the boundaries of the array can be utilised as natural legal cuts. For a one-dimensional ring in which two fragments at the two ends of a linear array are able to be combined into one in the same time slot, however, the situation becomes a bit more complicated. Because there are no natural boundaries like those in a linear array, to merge two adjacent fragments in different slots we have to find two legal cuts. The first one can be considered as a cut used to break the ring and then the properties discussed above for one-dimensional array can be applied.

Note that the re-packing may increase the scheduling overhead on a clustered parallel system because a message notifying the changes in the global scheduling matrix should 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 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. In these cases certain local scheduling tables need to be updated even without job re-packing. 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 allowed, the procedure for searching available resources can be simplified and then the overall system overhead for resource allocation may even be reduced.

4 Resource Allocation Based on Job Re-Packing

Conventionally the procedure for allocating processors to a new job is first to search if there is a suitable subset of available processors in an existing time slot. The job will then be allocated if such a suitable subset can be found. The purpose of this search is to try to avoid creating a new time slot which is not necessary. Because the search is done slot by slot and only local information on each time slot is considered, however, the results can often be far from optimal. This search procedure may also become expensive for large systems.

Consider a simple example depicted in Fig. 5. In the figure there are currently four time slots in a scheduling round. Assume that there is now a new job which requires five processors. Using the conventional method the system first starts search for a subset of consecutive idle processors of size greater than or equal to five in an existing time slot. In this particular example such a suitable subset of available processors cannot be found and then a new time slot has to be created. If job re-packing is allowed, however, it is easy to see that the new job can be allocated to processors P_3 to P_7 (or P_4 to P_8) in time slot S_2 by simply shifting job J_4 to slot S_3. Thus the search effort is totally wasted and the creation of a new time slot is also unnecessary.

Based on job re-packing we can obtain a new scheme for resource allocation. This new scheme can significantly simplify the search procedure and make better decisions for resource allocation.

WLV	0	0	1	2	2	2	2	3

S_4	J_7	J_7	J_7	J_8	J_8			
S_3	J_5	J_5	J_6	J_6				
S_2	J_3	J_3			J_4	J_4	J_4	J_4
S_1	J_1	J_1	J_1			J_2	J_2	
	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8

Fig. 5. Resource allocation using a workload vector WLV.

In addition to the global scheduling matrix, we introduce a *workload vector* (WLV) of length equal to the number of processors in the system, as depicted in Fig. 5. An integer value is assigned to each entry to record the number of idle slots on the associated processor. For example, the entry corresponding to processor P_1 is given a value 0 because there is no idle slot on that processor, while the last entry value of the vector is equal to 3 denoting there are currently three idle slots on processor P_8.

For the conventional allocation method adding this workload vector may not be able to assist the decision making for resource allocation. This is because the information contained in the vector 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 into a single time slot according to Property 1 discussed in the previous section. To search for a suitable subset of available processors, therefore, we only need to count consecutive nonzero entries in the workload vector if job re-packing is allowed. Thus the problem for searching on the entire two-dimensional scheduling matrix to find a suitable subset of available processors becomes a simple one-dimensional search problem. It is easy to see in Fig. 5 that, because there are six consecutive nonzeros in the workload vector, with a simple job re-packing process the new job which requires five processors can be allocated without creating a new time slot. Therefore, problems caused by the conventional method can significantly be alleviated.

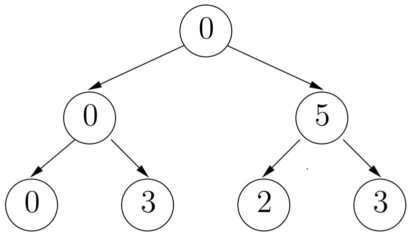
Our new scheme for resource allocation consists of three main steps. First we search in the workload vector WLV for a required number of consecutive nonzeros. A new time slot is created only if the required number of consecutive nonzeros in that vector cannot be found. This step determines a suitable subset of consecutive processors to be allocated to a new job. Then we trace adjacent idle fragments just within this subset of processors and group them into a single time slot through proper re-packing procedures such as those discussed in the previous section. This will determine in which time slot the new job resides. Finally we update the scheduling matrix and also local scheduling tables on each processors if there is any. Using this scheme the search for a suitable subset of available processors is simplified and the total number of time slots in a scheduling round can be kept low. Thus system performance may be enhanced.

To ensure a high system and job performance it is very important to balance workloads across the processors. Another advantage of our allocation scheme is that it is able to handle the problem of load balancing. Because the workload on each processor is recorded in the workload vector, the system can easily choose a subset of less active processors for an incoming job if there are several suitable subsets. In Fig. 5 there are two subsets of available processors suitable for a new job requiring five processors, that is, one from P_3 to P_7 and the other from P_4 to P_8. When the load balancing is taken into consideration, the second subset is preferable. It can be seen in the above example that the system can still allocate resources to a new job which requires six processors without creating a new time slot if the second subset is chosen.

With job re-packing the buddy based algorithm can be implemented in a more efficient way. Assume that a job of size p arrives for $n/2 < p \leq n$ and n being a power of 2. To find a suitable subset of available processors in an existing time slot, we need only to divide the workload vector into m groups of size n for $m = N/n$ and N being the number of processors in the system and check if there is a group in which all entry values are nonzero. If there are several available groups, we can choose the least loaded one by simply checking the entry values. Thus there is no need to scan all the existing time slots for finding a suitable subset and the workloads can more easily be balanced. Since there is a natural boundary between each pair of groups and no jobs can come across, re-packing jobs in the selected group will not affect the local scheduling tables of processors outside the group.

The procedure for detecting a suitable subset of available processors can further be simplified for the buddy based resource allocation system by introducing an additional binary tree structure to record the average group workload, as depicted in Fig. 6. The tree has $logN$ levels for N the number of processors in the system. 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 on the tree is assigned a value ac-

WLT



WLV	0	0	1	2	1	1	1	2

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

Fig. 6. The workload vector (WLV) and an additional binary tree (WLT) for recording the average group workload used for a buddy based resource allocation system.

cording to the values of its two children. It is just a sum of the two values of the children when both values are nonzero. Otherwise, it is set to zero. When the value is set to zero, the associated subset of processors will not be available for new arrivals under the current situation.

This binary tree is simple to manage and can greatly assist the decision making in resource allocation. We can let an existing job on a subset of processors run in multiple time slots when the value of the associated node is nonzero. It is easy to see in Fig. 6 that jobs J_2 and J_6 can run in multiple time slots because the values of two nodes on the right at the bottom level are nonzero. However, the value of the right node at the middle level is also nonzero. We are able to run job J_4 in multiple time slots by shifting J_6 down to time slot S_1 (or J_2 up to S_3).

Assume that job J_1 in Fig. 6 is terminated. The entry values associated with processors P_1 and P_2 in the workload vector become nonzero. Then the value of the two leftmost nodes at the bottom level of the binary tree will also be nonzero. This will cause the value of the left node at the middle level to become nonzero and so the root value. Since the top node is associated with all processors in the system, we know that there is at least one idle slot on each processor. According to Property 2 discussed in the previous section these idle fragments can be combined together in a single time slot which can then be eliminated. Using this binary tree, therefore, we are able to know quickly when a time slot can be deleted by simply checking the value of the root node to see if it is nonzero.

We can also quickly find a suitable subset of available processors for a new arrival simply by reading the values of those nodes at a proper level. Consider the situation depicted in Fig. 6 again and assume that a new job of size 4 arrives. In this case we need only to check the two nodes at the middle level. Since the value of the second node is nonzero, we know that the second subset of processors is available. We may then re-pack job J_6 into time slot S_1 and allocate four processors from P_4 to P_8 in time slot S_3 to the new arrival. Finally the values of the associated node and its children are updated. (In this particular case they are all set to zero.)

In the buddy based algorithm processors are continuously divided into two equal subsets and processes of the same job are allocated only in a selected subset. As we mentioned previously, with this arrangement the number of unification counts can be increased and jobs may have better chances to run in multiple time slots to enhance system and job performance. However, simply running jobs in multiple time slots may not be desirable when *fairness* is taken into consideration. In the above example jobs J_2 and J_6 can run in both time slots S_1 and S_3 before the new job arrives. Assume that no existing jobs are completed when the new job arrives. A new time slot has to be created and then the performance of those jobs running in a single time slot will be degraded.

Another potential problem associated with simply running jobs in multiple time slots is that the total number of time slots may become large when the system is busy. The system overhead will be increased when trying to manage a large number of time slots. Therefore, a better way to achieve a high system and job performance is to combine the procedures of slot unification and slot minimisation together. This combination can easily be implemented with simple modifications to our allocation technique discussed in this section.

5 Conclusions

In this paper we presented some ideas for resource allocation to enhance the performance of gang scheduling.

We introduced a job re-packing scheme. In this scheme we try to rearrange the order of job execution on their originally allocated processors in a scheduling round to combine small fragments of available processors into a larger and more useful one. We presented two interesting properties for re-packing jobs on a parallel system which is logically organised as a one-dimensional linear array. These two properties indicate that job re-packing is simple, the system costs may not be high and thus the scheme can be practical.

Based on job re-packing we developed an efficient resource allocation scheme. When processors are logically organised as a one-dimensional linear array any adjacent fragments can be grouped together to become a larger one in a single time slot. Thus we can use a workload vector which records the number of idle slots on each processor to detect a suitable subset of available processors for a given job. The problem for searching available processors on a two-dimensional global scheduling matrix then becomes a simple one-dimensional search problem. Because the scheme considers workload conditions in both space and time dimensions simultaneously, it is possible that the average number of slots per scheduling round can be kept low and workloads also be well balanced across the processors. Therefore, the resources in the system may be utilised more efficiently and the performance of parallel jobs may also be enhanced significantly.

There are many interesting and open problems relating to job re-packing. In this paper we only discussed how to re-pack jobs between rows in the global scheduling matrix. Therefore, an optimal solution in minimising the average number of time slots per scheduling round is achievable only when process or thread migration is not allowed. When processes can also be moved between columns in the global scheduling matrix, situations will become much more complicated because we must seriously consider the system overhead.

We only considered in the paper a simple system configuration, that is, processors in the system are logically organised as a one-dimensional linear array. The allocation schemes may work well for clusters of workstations/PCs/SMPs. However, there are parallel systems in which processor localities have to be considered in order to reduce the communication cost and to alleviate the problem of communication contention. An interesting problem is thus how to effectively re-pack jobs on other system configurations.

With job re-packing we are able to combine the procedures of slot unification and slot minimisation together. An interesting problem is how to determine when slot unification should be applied and when the total number of slots needs to be reduced such that the overall system and job performance can be enhanced.

Job re-packing may introduce extra system costs because of the extra requirement for updating the local scheduling tables on a distributed parallel system. On the other hand job re-packing may reduce the system overhead because the procedure for finding a suitable subset of available processors becomes cheaper especially for a buddy based resource allocation system. Extensive testing on real parallel machines is required to measure the actual system overhead and to try to find effective methods to further minimise the system costs.

Experiments based on the ideas described in this paper are to be undertaken on distributed-memory parallel machines, the Fujitsu AP1000+ and AP3000, at the Australian National University.

References

1. 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.
2. D. G. Feitelson and L. Rudolph, Distributed hierarchical control for parallel processing, *Computer*, 23(5), May 1990, pp.65-77.
3. 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.

4. J. K. Ousterhout, Scheduling techniques for concurrent systems, *Proceedings of Third International Conference on Distributed Computing Systems*, May 1982, pp.20-30.
5. 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.
6. 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.
7. 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.
8. 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.