Job Scheduling Strategies for Networks of Workstations

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

[1] Computer Sciences Laboratory, Australian National University,
Canberra, ACT 0200, Australia
[2] CAP Research Program, Australian National University,
Canberra, ACT 0200, Australia
[3] Electrotechnical Laboratory, 1-1-4 Umezono,
sukuba, Ibaraki 305, Japan

Abstract. In this paper we first introduce the concepts of utilisation ratio and effective speedup and their relations to the system performance. We then describe a two-level scheduling scheme which can be used to achieve good performance for parallel jobs and good response for interactive sequential jobs and also to balance both parallel and sequential workloads. The two-level scheduling can be implemented by introducing on each processor a registration office. We also introduce a loose gang scheduling scheme. This scheme is scalable and has many advantages over existing explicit and implicit coscheduling schemes for scheduling parallel jobs under a time sharing environment.

1 Introduction

The trend of parallel computer developments is toward *networks of workstations* [3], or *scalable parallel systems* [1]. In this type of system each processor, having a high-speed processing element, a large memory space and full functionality of a standard operating system, can operate as a stand-alone workstation for sequential computing. Interconnected by high-bandwidth and low-latency networks, the processors can also be used for parallel computing. To establish a truly general-purpose and user-friendly system, one of the main problems is to provide users with a *single system image*. By adopting the technique of *distributed shared memory* [12], for example, we can provide a single addressing space for the whole system so that communication for transferring data between processors is completely transparent to the client programs. In this paper we discuss another very important issue relating to the provision of single system image, that is, effective *job scheduling strategies* for both *sequential and parallel processing* on networks of workstations.

Many job scheduling schemes have been introduced in the literature and some of them implemented on commercial parallel systems. These scheduling schemes for parallel systems can be classified into either *space sharing*, or *time sharing*, or a combination of both. With space sharing a system is partitioned into subsystems, each containing a subset of processors. There are boundary

Dror G. Feitelson, Larry Rudolph (Eds.): JSSPP'98, LNCS 1459, pp. 143–157, 1998.

lines laid between subsystems and so only processors of the same subsystem can be coordinated to solve problems assigned to that subsystem. During the computation each subsystem is allocated only for a single job at a time.

The space partition can be either *static*, or *adaptive*. With static partitioning the system configuration is determined before the system starts operating. The whole system has to be stopped when the system needs to be reconfigured. With adaptive partitioning processors in the system are not divided before the computation. When a new job arrives, a job manager in the system first locates idle processors and then allocates certain number of those idle processors to that job according to some processor allocation policies, e.g., those described in [2,10,14,15,17,18,20]. Therefore, the boundary lines are drawn during the computation and will disappear after the job is terminated. Normally the static partitioning is used for very large systems, while the adaptive partitioning is adopted in systems, or subsystems of small to medium size. One disadvantage of space partitioning is that short jobs can easily be blocked by long ones for a long time before being executed. However, in practice short jobs usually demand a short turnaround time. To alleviate this problem jobs can be grouped into classes and a special treatment will be given to the class of short jobs [15]. However, it can only partially solve the problem. Thus *time sharing* needs to be considered.

Many scheduling schemes for time-sharing of a parallel system have been proposed in the literature. They may be classified into two basic types. The first one is *local scheduling*. With local scheduling there is only a single queue on each processor. Except for higher (or lower) priorities being given, processes associated with parallel jobs are not distinguished from those associated with sequential jobs. The method simply relies on existing local schedulers on each processor to schedule parallel jobs. Thus there is no guarantee that the processes belonging to the same parallel job can be executed at the same time across the processors. When many parallel programs are simultaneously running on a system, processes belonging to different jobs will compete for resources with each other and then some processes have to be blocked when communicating or synchronising with non-scheduled processes on other processors. This effect can lead to a great degradation in overall system performance [4,6,9,11,13]. One method to alleviate this problem is to use *two-phase* blocking [8,22] which is also called *implicit coscheduling* in [8]. In this method a process waiting for communication spins for some time in the hope that the process to be communicated with on the other processor is also scheduled, and then blocks if a response has not been received. The reported experimental results show that for parallel workloads this scheduling scheme performs better than the simple local scheduling. However, the problem is that the scheduling policy is based on communication requirements. Then it tends to give special treatment to jobs with a high frequency of communication demands. The policy is also independent of service times. The performance of parallel computation is thus unpredictable.

The second type of scheduling schemes for time sharing is *coscheduling* [16] (or *gang scheduling* [9]), which may be a better scheme in adopting *short-job-first* policy. Using this method a number of parallel programs is allowed to enter a *service* queue (as long as the system has enough memory space). The processes of the same job will run simultaneously across the processors 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 programs in the service queue take turns to receive the service in a coordinated manner across the processors. Thus programs never interfere with each other and short jobs are likely to be completed more quickly. There are also certain drawbacks associated with coscheduling. A significant one is that it is designed only for parallel workloads. For networks of workstations we need an effective scheduling strategy for both sequential and parallel processing. The simple coscheduling technique is not a suitable solution.

The future networks of workstations should provide a *programming-free* environment to general users. By providing a variety of high-performance computing libraries for a wide range of applications plus user-friendly interfaces for the access to those libraries, parallel computing will no longer be considered just as client's special requests, but become a *natural and common phenomenon* in the system. Along with many other critical issues, therefore, highly effective job management strategies are required for the system to meet various client's requirements and to achieve high efficiency of resource utilisation. Because of the lack of efficient job scheduling strategies, most networks of workstations are currently used exclusively either as an MPP for processing parallel batch jobs, or as a group of separate processors for interactive sequential jobs. The potential power of this type of system are not exploited effectively and the system resources are not utilised efficiently under these circumstances.

In this paper we discuss some new ideas for effectively scheduling both sequential and parallel workloads on networks of workstations. To achieve a desired performance for a parallel job on a network of workstations with a variety of competitive background workloads, it is essential to provide a sustained ratio of CPU utilisation to the associated processes on each processor, to allocate more processors to the job if the assigned utilisation ratio is small and then to coordinate the execution across the processors. We first introduce the concepts of *utilisation ratio* and *effective speedup* and their relations to the system performance in Section 2. In this section we also argue that, because the resources in a system are limited, one cannot guarantee every parallel job to have a sustained CPU utilisation ratio in a time sharing environment. One way to solve the problem is that we give short jobs sustained utilisation ratios to ensure a short turnaround time, while to each large job we allocate a large number of processors and assign a utilisation ratio which can vary in a large range according to the current system workload so that small jobs will not be blocked and the resource utilisation can be kept high. we then present in Section 3 a *two-level scheduling* scheme which can be used to achieve good performance for parallel jobs and good response for interactive sequential jobs and also to balance both parallel and sequential workloads. The two-level scheduling can be implemented by introducing on each processor a *registration office* which is described in Section 4. We discuss a scalable coscheduling scheme – *loose gang scheduling* in Section 5. This scheme requires both global and local job managers. It is scalable because the coscheduling is mainly controlled by local job managers on each processor so that frequent signal-broadcasting for simultaneous context switch across the processors is avoided. Using a global job manager we believe that the system can work more efficiently than those using only local schedulers. With a local job manager on each processor the system will become more flexible and more effective in handling more complicated situations than those adopting only the conventional gang scheduling policy. Finally the conclusions are given in Section 6.

2 Utilisation Ratio and Effective Speedup

Assuming that the overall computational time for a parallel job on p *dedicated* processors is $T_d(p)$, the *conventionally defined speedup* is then obtained as

$$S_d(p) = \frac{T_d(1)}{T_d(p)}. \tag{1}$$

This speedup can only be achieved by using dedicated processors. It may be impossible to achieve on a network of workstations because there a parallel job usually has to time-share resources with other sequential/parallel jobs. If we provide a *sustained ratio of CPU utilisation* for a job on each processor and use *more processors*, however, we can still achieve the desired performance in terms of time.

Define utilisation ratio β for $0 \leq \beta \leq 1$ as the ratio of CPU utilisation for a given job on each processor. By a given β the job on a processor can on the average obtain a service time $\beta \Delta T$ in each unit of time ΔT. In our scheduling strategy each parallel job will be assigned a utilisation ratio which is usually determined based on the current system working conditions. Different ratios can also be given on different processors for naturally unbalanced parallel jobs to achieve better system load-balancing.

Assume that the same utilisation ratio β is assigned to a parallel job across all the associated processors and that the job's processes are gang scheduled. The *turnaround time* $T_e(p)$ for that job can then be calculated as

$$T_e(p) = \frac{T_d(p)}{\beta} \tag{2}$$

where $T_d(p)$ is the computational time obtained on p dedicated processors.

Defining effective speedup $S_e(p)$ as the ratio of $T_d(1)$ and $T_e(p)$, then

$$S_e(p) = \frac{T_d(1)}{T_e(p)} = \beta \frac{T_d(1)}{T_d(p)} = \beta S_d(p). \tag{3}$$

where $S_d(p)$ is the conventional speedup obtained on p dedicated processors.

To achieve a desired performance, we may set a performance target γ and require

$$T_d(1) \geq \gamma T_e(p), \tag{4}$$

or

$$S_e(p) \geq \gamma. \tag{5}$$

If the effective speedup for a given job is lower than that target, the performance will be considered unacceptable.

¿From equations in (3) and (5) we can obtain

$$S_d(p) \geq \frac{\gamma}{\beta}. \tag{6}$$

Using the above inequality we can easily determine how many processors should be allocated to a given job in order to achieve a desired performance when a particular β is given. Assuming $\gamma = 2$ and $\beta = 0.5$, for example, $S_d(p)$ must be greater than, or equal to 4. Allocating 5 processors or more to that job can then achieve a desired performance if $S_d(5) \geq 4$. When the current system workload is not heavy, we may need to use less number of processors to achieve the same performance. If there are several idle processors, we may set $\beta = 1$ in the above example. Then only 3 processors may be required if $S_d(3) \geq 2$.

In practice the exact speedup $S_d(p)$ may not be known except for those programs in standard general-purpose parallel computing libraries. Thus the values can only be approximate in those cases. However, good approximations can often be obtained. For example, the results of the Linpack Benchmark [7] can be used as a good approximation for problems of matrix computation.

The utilisation ratios of the existing jobs may be decreased whenever a new job enters the system to time-share the resources. The problem is how to ensure a sustained ratio of CPU utilisation for each job so that the performance can be predictable in a time sharing environment. Since the resources in a system are really limited, the answer to this question is simply that we cannot guarantee every job to have a sustained ratio when the system workload is heavy.

One way to solve the above problem is to adopt the following scheme. First we set a limit to the length of each *scheduling round* ΔT (or a limit to the number of jobs in the system). A common misunderstanding about time-sharing for parallel jobs is that good performance will be obtained as long as parallel jobs can enter the system and start operation quickly. As we mentioned previously that the resources in a system are limited, however, good performance just cannot be guaranteed if the length of scheduling round is unbounded. Consider a simple example when several large jobs are time-sharing the resources in a round robin manner. In this case the conventional gang scheduling simply fail to produce good performance in terms of turnaround time.

Because of the limit to the length of each scheduling round short jobs still can be blocked for a long time. We then adopt a scheduling policy, that is, small jobs should have sustained utilisation ratios to ensure a short turnaround time, while each large job should be assigned a large number of processors, but given a utilisation ratio which can vary in a large range according to the current system workload. In this way we think that small jobs will not be blocked, the resource utilisation can be kept high and reasonably good performance for large jobs may also be obtained.

Based on the above ideas a *multi-class time/space sharing system* is designed. A detailed description of this system is beyond the scope of this paper. Interested readers may refer [24] for more details.

3 Two-Level Scheduling

It can be seen from the previous section that our scheduling strategy is based on the utilisation ratios assigned to parallel jobs. In this section we introduce a *two-level scheduling* scheme for balancing the workloads for both sequential and parallel processing,

At the top level, or *global level* the *gang scheduling*, or a *loose gang scheduling* scheme to be discussed in the next section, is adopted to coordinate parallel computing. Each *scheduling round* ΔT is divided into *time slots.* An example of the time distribution for different processes on each processor is shown in Fig. 1. In the figure time slot $\Delta t_s^{(i)}$ is allocated only to *sequential processes* associated with sequential jobs, while slot $\Delta t_p^{(i)}$ is assigned to a single *parallel process* associated with a parallel job. A parallel process may share its time slots with sequential processes through the scheduling at the bottom level, or *local level.* However, no parallel processes will share the same time slots. This is to avoid many different types of parallel jobs competing for resources at the same time and then to guarantee that each parallel process can obtain its proper share of resources. The relation between a scheduling round and those time slots satisfies the following equation

$$\Delta T = \Delta T_s + \sum_{i=1}^{n} \Delta t_p^{(i)} \tag{7}$$

where $\Delta T_s = \sum_{i=1}^{m} \Delta t_s^{(i)}$ is the total time dedicated for sequential jobs in a scheduling round and is distributed to gain good response to interactive clients.

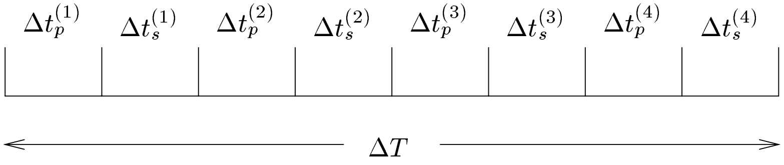

Fig. 1. The time distribution in a scheduling round.

The width of each time slot is determined by the corresponding utilisation ratio $\beta_p^{(i)}$, or $\beta_s^{(i)}$. We can then calculate the width of each time slot as

$$\Delta t_p^{(i)} = \beta_p^{(i)} \Delta T \tag{8}$$

and

$$\Delta T_s = \beta_s \Delta T \tag{9}$$

where $\beta_s = \sum_{i=1}^{m} \beta_s^{(i)}$.

There are many ways to distribute ΔT_s. For example, each slot for a parallel process can be followed by a small slot for sequential processes and ΔT_s is *uniformly* distributed across the whole scheduling round. Then

$$\Delta t_s^{(i)} = \frac{\Delta T_s}{n}. \tag{10}$$

We can also distribute ΔT_s *proportionally* to the width of each time slot for parallel processes, that is,

$$\Delta t_s^{(i)} = \frac{\beta_p^{(i)}}{\sum_{j=1}^{n} \beta_p^{(j)}} \Delta T_s. \tag{11}$$

The calculation for *proportional distribution* is a bit more complicated than that for *uniform distribution.* However it is useful when a *proper-share* policy, which will be described later in the section, is applied at the local level.

Different local policies can be adopted to schedule processes within each time slot. In those time slots dedicated for sequential processing conventional local scheduling schemes of any standard operating system will be good enough. In the following we discuss how to schedule processes in each time slot $\Delta t_p^{(i)}$ in which parallel processing is involved.

To ensure that a parallel process can obtain its assinged share of CPU utilisation, the whole slot $\Delta t_p^{(i)}$ may be dedicated just to the associated parallel process. In that case a very high priority will be given and the process simply does *busy-waiting*, or *spins* during communication/synchronisation so that no other processes can disturb its execution within each associated time slot. One problem associated with this policy is that the performance of sequential jobs, especially of those which demand good interactive response, may significantly be affected. Therefore, its use will be treated as special cases under the environment of networks of workstations to achieve certain client's special requests .

To prevent great performance degradation of sequential interactive jobs, *implicit coscheduling* scheme can be adopted. However, a potential problem is that the execution of a parallel process may be disturbed by several sequential processes and then it is possible that certain parallel processes may not receive their proper shares in their associated time slots.

The above problem may be alleviated by adopting a *proper-share* policy. In this policy we do not consider individual shares allocated for each sequential job. Except for special ones, e.g., multimedia workloads, which may be treated in the same way as parallel jobs to achieve constant-rate services, only a combined share of sequential processes $\Delta t_s^{(i)}$ is considered. Each distributed time slot for sequential processes $\Delta t_s^{(i)}$ is also integrated with its associated time slot $\Delta t_p^{(i)}$ to form a single time slot of width $\Delta t^{(i)}$, that is,

$$\Delta t^{(i)} = \Delta t_p^{(i)} + \Delta t_s^{(i)}. \tag{12}$$

In each integrated time slot implicit coscheduling is applied to support both parallel and sequential processing. When its allocated share is not used up in time $\Delta t_p^{(i)}$, a parallel process can still obtain services till the end of the integrated time slot $\Delta t^{(i)}$ though $\Delta t^{(i)}$ is longer than $\Delta t_p^{(i)}$. When a parallel process has consumed its share before the end of an integrated time slot, however, it will be blocked and the services in the remaining time slot then dedicated to sequential processes. With this policy parallel processes and sequential processes as a whole may be guaranteed to obtain their proper shares during the computation.

Similar to the one described in [5], the policy may be realised by applying the *proportional-share* technique which are originally used for *real-time* applications [19,21]. However, our scheduling scheme is much simpler and easier to implement because only the proper share of a *single* parallel process is considered against a *combined* share of sequential processes in each time slot.

Now the problem is how to distribute the total time ΔT_s allocated for processing sequential jobs. The uniform distribution using the equation in (10) is easy to calculate. However, the resulting $\Delta t_s^{(i)}$ may be too small to compensate the lost share of parallel processes which have large $\beta_p^{(i)}$s. Therefore, the proportional distribution using (11) may be a more proper one.

Normalising ΔT, that is, setting $\Delta T = 1$, the equation in (7) will become

$$1 = \beta_s + \sum_{i=1}^{n} \beta_p^{(i)}. \tag{13}$$

Using equations in (8), (9), (11) and (13), we obtain

$$\Delta t^{(i)} = \Delta t_s^{(i)} + \Delta t_p^{(i)} = \frac{\beta_p^{(i)}}{\sum_{j=1}^{n} \beta_p^{(j)}} \Delta T. \tag{14}$$

The width of an integrated time slot $\Delta t^{(i)}$ can directly be obtained by using the equation in (14) and thus there is no need to explicitly calculate $\Delta t_s^{(i)}$s.

4 Registration Office

When a parallel process has used up its time slot, it will be *preempted* at the global level and another parallel process be *dispatched.* After being dispatched, parallel processes may time-share resources with sequential processes on each processor. Just like sequential processes, parallel processes will then be either in *running* state, or *ready* and *blocked* states, which is controlled by a local scheduler. Because in our two-level scheduling the execution of parallel processes are controlled at both global and local levels, special care has to be taken to avoid potential scheduling conflicts. For example, the global scheduler wants to preempt a parallel process which is currently not in running state. To solve this problem we introduce a *registration office* on each processor.

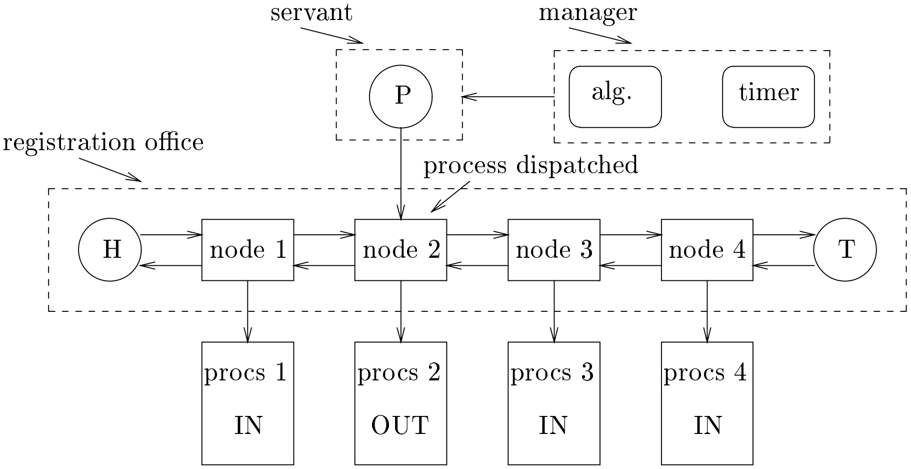

Fig. 2. The organisation of a registration office.

The registration office is constructed by using a linked list as shown in Fig. 2. When a parallel job is initiated, each associated process will enter the local sequential queueing system the same way as sequential processes on the corresponding processor. Just like sequential processes, parallel processes can be either in running state, or in ready state requesting for service, or in blocked state during communication/synchronisation. However, every parallel process has to be *registered* in the registration office, that is, on each processor the linked list will be extended with a new node which has a pointer pointing to the process just being initiated. Similarly, when a parallel job is terminated, it has to *check out* from the office, that is, the corresponding node on each processor will be deleted from the linked list.

As we discussed in the previous section, certain parallel processes may be assigned a very high priority so that they can occupy the whole time slots allocated to them. In that case the execution of sequential workloads can be seriously deteriorated. To alleviate this problem we may introduce certain time slots $\Delta t_s^{(i)}$ which are dedicated to sequential jobs only. This can be done by introducing *dummy nodes* in the linked list. A dummy node is the same type of nodes in a linked list except its pointer points to NULL, the *constant zero*, instead of a real parallel process. It seems that there is a *dummy parallel process* associated with that node. When a service is given to that dummy parallel process, the whole time slot will be dedicated to sequential processes.

There is a *servant* working in the office. When the servant comes to a place, or a node in the linked list, the process associated with that node can receive services, or *be dispatched*. When a process is dispatched, it will be marked *out*. Other processes which are not dispatched will be marked *in*. In practice a process may be *blocked* if it is marked *in*. Therefore, a parallel process can come out of the blocked status only if it is *ready* for service (controlled by the local scheduler) and the event *out* occurs (controlled by the top level scheduler). By letting only one parallel process be marked *out* on each processor at any time, we can guarantee that only one parallel process time-shares resources with sequential processes in each time slot.

When a time slot is ended for the current parallel process, the servant will move to a new node. The parallel process associated with that node can then be serviced next. However, the movement of the servant is totally controlled by an *office manager* which has a *timer* to determine when the servant is to move and an *algorithm* to determine which node the servant is to move to. The algorithm can be simple ones such as the conventional round-robin. (To obtain a high system throughput, however, other more sophisticated scheduling schemes may also be considered.) The timer is to ensure that processes can obtain their allocated service times, that is, $\Delta t_p^{(i)}$s, or $\Delta t_s^{(i)}$s in each scheduling round.

The use of registration offices is similar to that of the two-dimensional matrix adopted in the conventional coscheduling. Each column of the matrix corresponds to a time slot and each row to a processor. The coscheduling is then controlled based on that matrix. It is easy to see that the linked list on each processor plays the same role as a row of that matrix in coscheduling parallel processes. However, the key difference is that our two-level scheduling scheme allows both parallel and sequential jobs to be executed simultaneously.

5 Loose Gang Scheduling

The conventional gang scheduler is centralised. The system has a central controller. At the end of each time slot the controller broadcasts a message to all processors. The message contains the information about which parallel workload will receive a service next. The centralised system is easy to implement, especially when the scheduling algorithm is simple. However, frequent signal-broadcasting for simultaneous context switch across the processors may degrade the overall system performance on machines such as networks of workstations and space-sharing policies may not easily be adopted to enhance the efficiency of resource utilisation. Because in our system there is a registration office on each processor, we can adopt a *loose gang scheduling* policy to alleviate these problems.

In our system there is a *global job manager*. It is used to monitor the working conditions of each processor, to locate and allocate processors and to assign utilisation ratios to parallel jobs, and to balance parallel and sequential workloads. We believe that resources in networks of workstations cannot efficiently be utilised without an effective global job manager. This global job manager is also able to broadcast signals for the purpose of synchronisation to coordinate the execution of parallel jobs. However, the signals need not be frequently broadcast for simultaneous context switch between time slots across the processors. They are sent only once after each scheduling round, or even many scheduling rounds to adjust the potential skew of the corresponding time slots (or simply *time skew*) across the processors caused by using *local job managers* on each processor.

There is a local job manager on each processor. It is used to monitor and report to the global job manager the working conditions on that processor. It also takes orders from the global job manager to properly set up its registration office and to coordinate the execution of parallel jobs with other processors. With help of the global job manager the effective coscheduling is guaranteed by using local job managers on each processor.

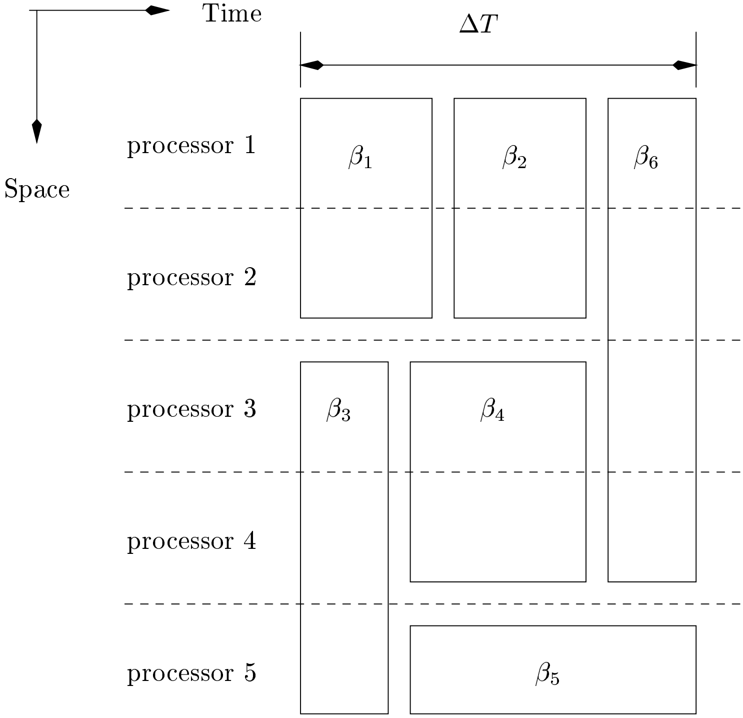

Fig. 3. The time/space allocation for six jobs on five processors.

In the following we give a simple example which demonstrates more clearly the effectiveness of using the loose gang scheduling scheme and which also presents another way of deriving the registration office for the scheme.

Our simple example considers the execution of six jobs on five processors. We assume that the time/space allocation has already been done, that is, the number of processor and the utilisation ratio have been assigned for each job, as depicted in Fig. 3. For various reasons such as described in the previous sections the shapes of time/space allocation may not be the same for each job as indicated in the figure. This will make it very difficult for a centralised controller to coschedule jobs. However, the problem can easily be solved by adopting our loose gang scheduling.

On each processor we run a local job manager and we also set up a *scheduling table* which is given by the global job manager. Parallel processes are then scheduled according to this scheduling table. In our example there are three different scheduling tables, as shown in Fig. 4(a). The processes and the lengths of their allocated time slots in a scheduling round are listed in each table in an ordered manner. It is easy to see that, if the processors are synchronised at the beginning of each scheduling round (It is also possible that the processors can be synchronised once many scheduling rounds.) and local job managers schedule parallel processes according to the given scheduling tables, the correct coscheduling across the processors is then guaranteed.

Because both content and size of each table vary from time to time during the computation, it is quite natural to implement the scheduling tables using linked lists, which results in our registration office. A registration office on processor 1 is depicted in Fig. 4(b). Note each node in the linked list has a pointer which points at the corresponding process so that any unnecessary search for parallel processes can be avoided.

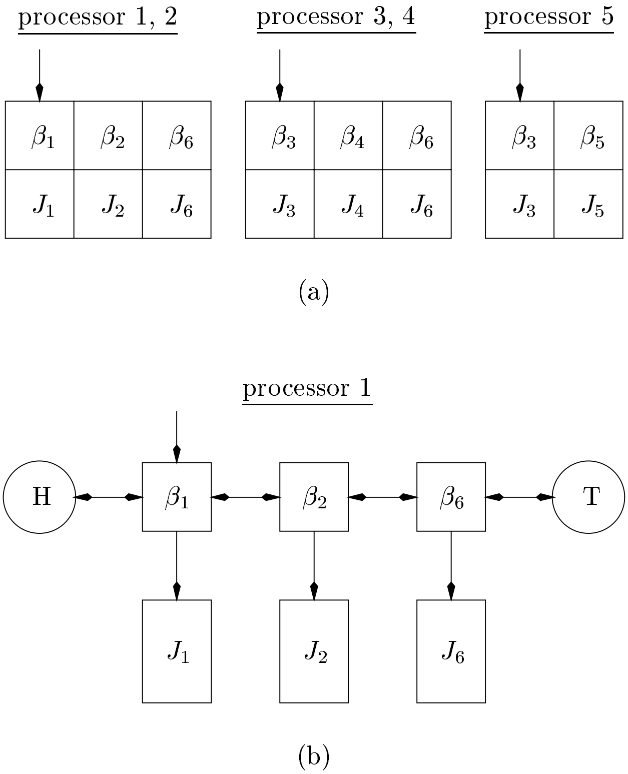

Fig. 4. (a) The scheduling tables assigned for each processor and (b) The registration office on processor 1.

With the collaboration of the global and local job managers the system can work correctly and effectively. A potential disadvantage of the loose gang scheduling is that there is an additional cost for executing the coscheduling algorithm on each processor. However, in practice time slots $\Delta t_p^{(i)}$, or $\Delta t^{(i)}$ are usually in order of seconds. This extra cost for running a process for coscheduling will be relatively very small.

6 Conclusions

In this paper we discussed some new ideas for effectively scheduling both parallel and sequential workloads on networks of workstations.

To achieve a desired performance in a system with a variety of competitive background workloads, the key is to assign a sustained CPU utilisation ratio on each processor to a parallel job so that the performance becomes predictable. Because the resources in a system are limited, however, we cannot guarantee that every job will be given a sustained utilisation ratio. One way to solve this problem is that small parallel jobs are assigned a sustained ratio of resource utilisation, while each large parallel job is allocated a large number of processors and assigned a utilisation ratio which can vary in a wide range according to the current system workload. Thus small jobs are not blocked by larger ones and a short turnaround time is guaranteed, high efficiency of resource utilisation can be achieved and reasonably good performance for large jobs may also be obtained.

To balance the workloads for both sequential and parallel processing, we introduced a two-level scheduling scheme. At the global level parallel jobs are coscheduled so that they can obtain their proper shares without interfering with each other and they can also be coordinated across the processors to achieve high efficiency in parallel computation. At the local level many different policies, e.g., the busy-waiting (or spinning) and the implicit coscheduling (or two-phase blocking), can be considered to schedule both parallel and sequential processes. We introduced a proper-share policy for effectively scheduling processes at the local level. By adopting this policy we can obtain good performance for each parallel job and also maintain good response for interactive sequential jobs. The two-level scheduling can be implemented by adopting a registration office on each processor. The organisation of the registration office (which is also described in [23]) is simple and the main purpose is to effectively schedule parallel processes at both global and local levels.

We also introduced a loose gang scheduling scheme to coschedule parallel jobs across the processors. This scheme requires both global and local job managers. The coscheduling is mainly controlled by local job managers on each processor, so frequent signal-broadcasting for simultaneous context switch across the processors is avoided. There is only a bit extra work for global job manager to adjust potential time skew. The name *loose gang* has two meanings. First the coscheduling is achieved by mainly using local job managers but not just a central controller and second parallel processes may time-share their allocated time slots with sequential processes. Since both global and local job managers play effective roles in job scheduling, we think this may lead a way for us to find good strategies for efficiently scheduling both parallel and sequential workloads on networks of workstations.

A new system based on these ideas is currently under construction on a distributed memory parallel machine, the Fujitsu AP1000+, at the Australian National University.

References

1. T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias and M. Snir, SP2 system architecture, *IBM Systems Journal*, 34(2), 1995.
2. S. V. Anastasiadis and K. C. Sevcik, Parallel application scheduling on networks of workstations, *Journal of Parallel and Distributed Computing*, 43, 1997, pp.109-124.
3. T. E. Anderson, D. E. Culler, D. A. Patterson and the NOW team, A case for NOW (networks of workstations), *IEEE Micro*, 15(1), Feb. 1995, pp.54-64.
4. R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson and D. A. Patterson, The interaction of parallel and sequential workloads on a network of workstations, *Proceedings of ACM SIGMETRICS'95/PERFORMANCE'95 Joint International Conference on Measurement and Modeling of Computer Systems*, May 1995, pp.267-278.
5. A. C. Arpaci-Dusseau and D. E. Culler, Extending proportional-share scheduling to a network of workstations, *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 1997.
6. M. Crovella, P. Das, C. Dubnicki, T. LeBlanc and E. Markatos, Multiprogramming on multiprocessors, *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, Dec. 1991, pp.590-597.
7. J. J. Dongarra, Performance of various computers using standard linear equations software, Technical Report CS-89-95, Computer Science Department, University of Tennessee, Nov. 1997.
8. A. C. Dusseau, R. H. Arpaci and D. E. Culler, Effective distributed scheduling of parallel workloads, *Proceedings of ACM SIGMETRICS'96 International Conference*, 1996.
9. 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.
10. D. Ghosal, G. Serazzi and S. K. Tripathi, The processor working set and its use in scheduling multiprocessor systems, *IEEE Transactions on Software Engineering*, 17(5), May 1991, pp.443-453.
11. A. Gupta, A. Tucker and S. Urushibara, The impact of operating system scheduling policies and synchronisation methods on the performance of parallel applications. *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1991, pp.120-131.
12. K. Li, IVY: A shared virtual memory system for parallel computing, *Proceedings of International Conference on Parallel Processing*, 1988, pp.94-101.
13. S.-P. Lo and V. D. Gligor, A comparative analysis of multiprocessor scheduling algorithms, *Proceedings of the 7th International Conference on Distributed Computing Systems*, Sept. 1987, pp.205-222.

14. V. K. Naik, S. K. Setia and M. S. Squillante, Performance analysis of job scheduling policies in parallel supercomputing environments, *Proceedings of Supercomputing'93*, Nov. 1993, pp.824-833.
15. V. K. Naik, S. K. Setia and M. S. Squillante, Processor allocation in multiprogrammed distributed-memory parallel computer systems, IBM Research Report RC 20239, 1995.
16. J. K. Ousterhout, Scheduling techniques for concurrent systems, *Proceedings of Third International Conference on Distributed Computing Systems*, May 1982, pp.20-30.
17. E. Rosti, E. Smirni, L. Dowdy, G. Serazzi and B. M. Carlson, Robust partitioning policies of multiprocessor systems, *Performance Evaluation*, 19(2-3), 1994, pp.141-165.
18. S. K. Setia, M. S. Squillante and S. K. Tripathi, Analysis of processor allocation in multiprogrammed, distributed-memory parallel processing systems, *IEEE Transactions on Parallel and Distributed Systems*, 5(4), April 1994, pp.401-420.
19. I. Stoica, H. Abdel-wahab, K, Jeffay, S. Baruah, J. Gehrke and C. G. Plaxton, A Proportional share resource allocation algorithm for real-time, time-shared systems, *IEEE Real-Time Systems Symposium*, Dec. 1996.
20. K. Suzaki, H. Tanuma, S. and Y. Ichisugi, Design of combination of time sharing and space sharing for parallel task scheduling, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, Nov. 1997.
21. C. A. Waldspurger and W. E. Weihl, Stride scheduling: deterministic proportional-share resource management, Technical Report MIT/LCS/TM-528, MIT Laboratory for Computer Science, MIT, June 1995.
22. J. Zahorjan and E. D. Lazowska, Spinning versus blocking in parallel systems with uncertainty, *Proceedings of the IFIP International Seminar on Performance of Distributed and Parallel Systems*, Dec. 1988, pp.455-472.
23. 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.
24. B. B. Zhou, R. P. Brent, D. Walsh and K. Suzaki, A multi-class time/space sharing system, Tech. Rep., DCS and CSLab, Australian National University, 1998, in process.