Effective Scheduling in a Mixed Parallel and Sequential Computing Environment

B. B. Zhou, X. Qu and R. P. Brent
Computer Sciences Laboratory
The Australian National University
Canberra, ACT 0200, Australia

Abstract

In this paper we describe a two-level scheduling scheme for mixed parallel and sequential workloads on scalable parallel machines. The design of this scheduling system is based on two principles, that is, parallel programs should be scheduled in a coordinated manner so that they will not severely interfere with each other and the performance for parallel computing becomes predictable, and parallel programs may time-share resources with sequential programs so that the efficiency of processor utilisation can greatly be enhanced and good response to interactive clients can be maintained. We also discuss the organisation of a registration office through which the two-level scheduling is realised.

1 Introduction

The trend of parallel computer developments is towards networks of workstations [2], or scalable parallel systems [1]. In this type of system each processor, having a high-speed processing element, a large-size memory space and full functionality of a standard operating system, can operate as a stand-alone workstation for sequential computing. Interconnected by a high-bandwidth and low-latency network, the processors can also be used for parallel computing. Therefore, the system actually provides a mixed parallel and sequential computing environment. To efficiently utilise the system resources, it is very important to design an effective scheduling algorithm for mixed parallel and sequential workloads.

There exist many scheduling schemes for parallel machines. The simplest and most commonly used method is *batch job system*. This system only allows one parallel programs running at a time. Once a parallel job enters the service, it will continuously be serviced until finished and all other parallel jobs have to wait in a *batch* queue outside of the system. If the number of parallel jobs waiting in the batch queue is large, however, it is likely that short jobs may be blocked by several longer ones. In general batch systems are not user-friendly and the resources may not be used efficiently.

To solve problems encountered by the simple batch scheme, multiprogramming should be considered. Multiprogramming on parallel systems is much more complicated than that on sequential machines. We need to consider the problem of *time-sharing*, that is, the problem of interactive use of processors by many jobs simultaneously. We should also consider the problem of *space-sharing*, that is, the problem of static/dynamic location and allocation of processors to different jobs. In the following discussion we mainly discuss the issue of time-sharing. However, we must stress that an effective resource management system which addresses both time-sharing and space-sharing needs to be studied to solve the overall scheduling problem. The discussion of this type of management system is beyond the scope of this paper.

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 in 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 jobs 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 [3, 4, 7, 9, 11].

One method to alleviate this problem is to use *two-*

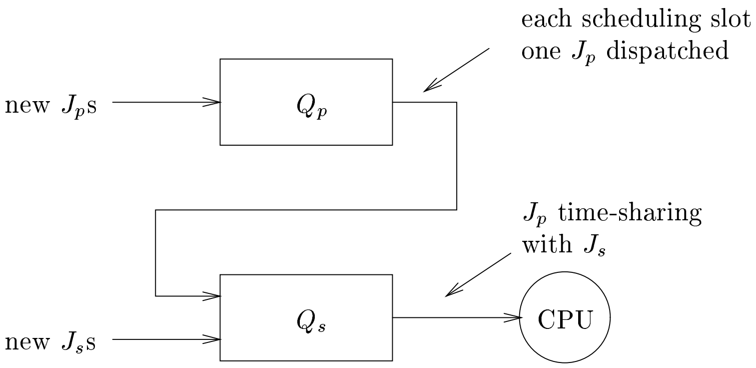

Figure 1: A two-level scheduling scheme.

phase blocking [16, 6] which is also called *implicit coscheduling* in [6]. 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 the response is still not received. The reported experimental results show that for parallel workloads this scheduling scheme performs better than the simple local scheduling. However, one problem associated with local scheduling is that the scheduling of parallel jobs is independent of their service times. Thus the performance is unpredictable. If the system is busy, for example, short jobs may not be completed quickly.

The second type of scheduling scheme is *Coscheduling* [13] (or *gang scheduling* [7]), 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. In each scheduling slot there is only one process running on each processor and the process simply does busy-waiting during communication/synchronisation. This will waste processor cycles and decrease the efficiency of processor utilisation.

It can be seen from the above discussion that both local scheduling and coscheduling have problems in scheduling mixed parallel and sequential workloads on scalable parallel computers. In this paper we then describe a new scheduling system. The system design is based on two principles, that is, first parallel workloads should be scheduled in a coordinated way so that they will not severely interfere with each other and the performance of parallel computing becomes predictable, and second both parallel and sequential workloads may time-share resources on each processor so that the efficiency of processor utilisation can be enhanced and good response to interactive clients can be maintained on each processor.

In our new system parallel workloads are scheduled at two different levels. At the first, or *global* level they are coscheduled across the processors, while at the second, or *local* level processes associated with parallel jobs may then time-share resources with sequential processes on each processor, which is controlled by a local scheduler. Thus our scheduling scheme is actually a combination of local scheduling and coscheduling.

The big question is how to realise this kind of two-level scheduling. To achieve this we designed a *registration office* which is attached to each processor to coordinate parallel workloads and to balance the resource utilisation for parallel and sequential jobs. This registration office is constructed on top of the conventional queueing system. Thus our scheduling system for mixed parallel and sequential workloads can be constructed without significant modifications to the

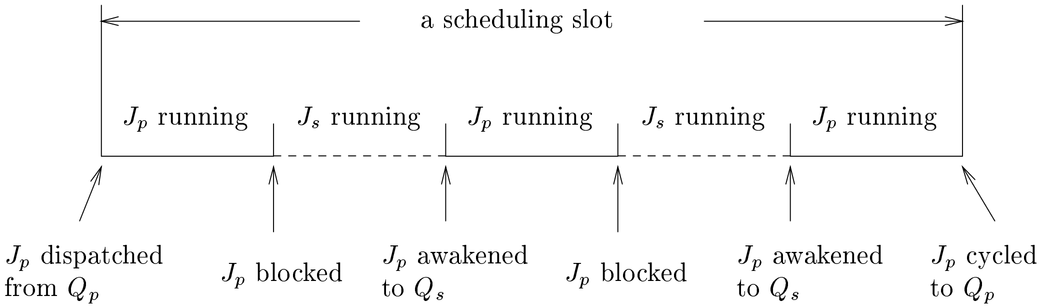

Figure 2: The normal situation in a scheduling slot.

conventional operating system adopted on each processor.

The paper is organised as follows. In Section 2 we describe the basic structure of our two-level scheduling system. We then introduce the organisation of the registration office and show how parallel workloads can be serviced coordinately across the processors in Section 3. In the section we also discuss a *loose gang scheduling* technique. Using this technique we can alleviate the disadvantages of conventional gang scheduling which uses a centralised controller. The conclusion is given in Section 4.

To simplify the description, in this paper processes associated with parallel jobs are called *parallel processes* to distinguish them from those *sequential processes* associated with sequential jobs.

2 The Basic Two-Level Structure

The basic structure of the two-level scheduling scheme on each processor is depicted in Fig. 1. This system consists of two queues, a queue Q_p at the first level and a conventional, or sequential queue Q_s at the second level. Because it is used to coordinate parallel workloads across the processors, Q_p is then called parallel queue in the following discussion. While new sequential processes directly come to the sequential queue, all parallel processes will first enter the parallel queue and then be dispatched to the sequential queue before receiving a service. Since coscheduling is applied, each time only one parallel process can be dispatched from the parallel queue and thus at any time instant there may only be one parallel process in the sequential queue. If parallel processes associated with the same job are placed at the same place in each parallel queue across the processors and the same scheduling algorithm is applied, they can then be dispatched at the same time.

After entering the sequential queue the parallel process on each processor may *time-share* the service with sequential processes. Unlike the conventional coscheduling parallel processes can be blocked during communication/synchronisation and then sequential processes can be serviced, as shown in Fig. 2. When the parallel process is awakened, instead of entering the parallel queue it goes to the sequential queue so that it can continuously be serviced within its own scheduling slot. In each scheduling slot the parallel process may be blocked several times. By the end of the scheduling slot it will be cycled to the parallel queue and wait there for the next service. This *normal* situation in a scheduling slot is depicted in Fig. 2.

To ensure the efficiency of parallel computing across the processors implicit coscheduling should be applied at the local level.

Since parallel processes may time-share resources with sequential processes, coordination of parallel workloads becomes more complicated. Assume that the parallel queue is constructed as a conventional queue, that is, parallel processes will be detached from the queue after being dispatched. To time-share resources with sequential processes, parallel processes will be either in *running* state, or *ready* and *blocked* states just like sequential processes. The situation in Fig. 2 only shows that the parallel process is in running state at the end of its scheduling slot. Then the process can easily be found and cycled back to the parallel queue. When a parallel process is still in either ready, or blocked state at the end of the scheduling slot, however, the system has to look for it from the queues for processes in ready and blocked states. Otherwise, the system will lose control to this process and parallel workloads cannot properly be coordinated. In the next section we introduce a structure of *registration office* for parallel queue Q_p to avoid complicated procedures for searching parallel processes.

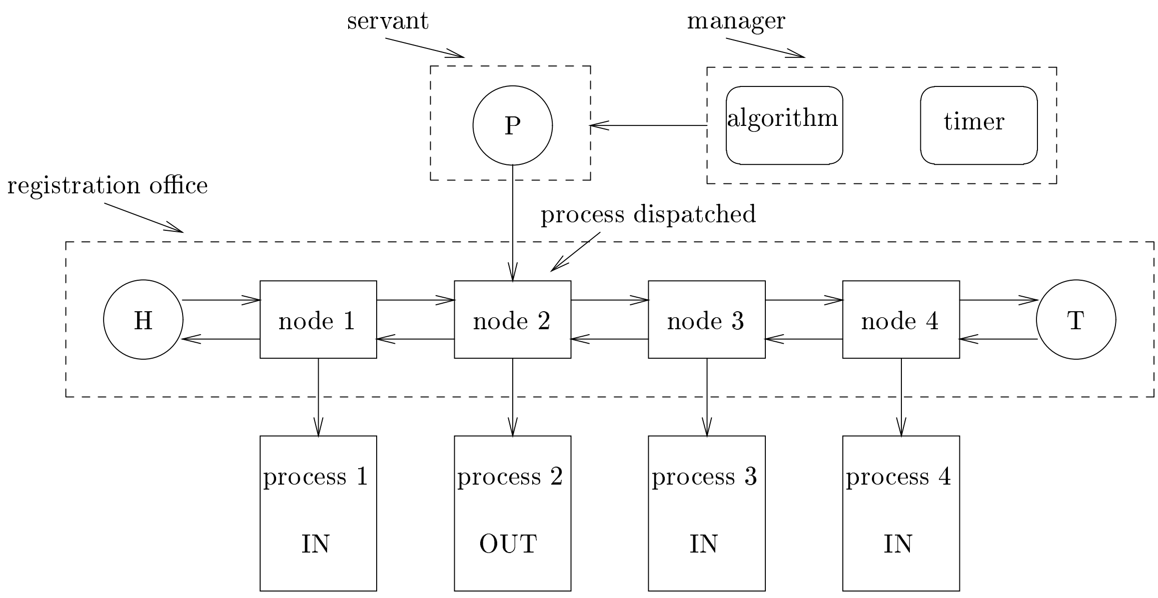

Figure 3: The organisation of a registration office.

3 The Organisation of a Registration Office

To avoid complicated procedures for searching parallel processes, we introduce a *registration office* which is constructed by using a linked list as shown in Fig. 3. 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.

In certain special cases, parallel processes may be assigned a very high priority so that they can occupy the whole time slots allocated to them. In those cases the execution of sequential workloads can be seriously deteriorated. To alleviate this problem we may introduce certain time slots 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 scheduling 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 will receive a service, or *be dispatched.* When a process is dispatched, it will be marked *out.* Other processes which are not receiving services 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 scheduling slot.

When a scheduling 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 in each scheduling round.

The use of registration offices is similar to that of the two-dimensional matrix adopted in the conventional coscheduling. Each row of the matrix corresponds to a scheduling slot and each column 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.

The conventional gang scheduler is centralised. The system has a central controller. At the end of each scheduling 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. Because in our system there is a registration office on each processor, we can adopt a *loose gang scheduling* policy to alleviate this problem.

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 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 can also be 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 scheduling slots across the processors. They are sent only once after each scheduling round, or even many scheduling rounds to adjust the potential skew of corresponding scheduling slots across the processors (or simply *time skew*) 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.

It is easy to see that with help of the global job manager the effective coscheduling is quaranteed if the local job managers of all the coordinated processors adopt the same scheduling algorithm. With the collaboration of the global and local job managers the system can then 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 scheduling slots are usually in order of seconds. This extra cost for running a coscheduling process will be relatively very small.

It should be noted that the registration office can easily be extended to having multiple lists. For example, parallel workloads can be classified based on their required service times. (They can also be grouped into different classes according to the number of processors they require [15].) Processes belonging to the same class will be linked to the same list. The different length of scheduling slots may also be allocated to processes of different classes. By using multiple lists plus a proper scheduling algorithm the system performance can greatly be enhanced.

Multiple servants can also be *employed* to work in each registration office. To activate, or dispatch a parallel process is just to mark it *out* in our coscheduling system. If multiple servants are used, therefore, several parallel processes can time-share the same scheduling slots if necessary.

4 Conclusions

In this paper we first presented a two-level scheduling scheme for mixed parallel and sequential workloads on scalable parallel machines, or networks of workstations. The design of this scheduling system is based on two basic principles, that is, parallel programs should be coscheduled so that they will not severely interfere with each other and their performance will become deterministic, and parallel programs can time-share resources with sequential programs so that the efficiency of processor utilisation can greatly be enhanced and good response to interactive clients can be maintained.

Our objective is achieved by introducing a concept of registration office which coordinates parallel workloads across the processors and allows the execution of parallel and sequential workloads simultaneously. Our scheme is also simple to implement because it is built on top of the conventional scheduling system so that it is not required to significantly modify the conventional operating system adopted on each processor of a given system.

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. 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.

The experiment is currently undertaken on the Fujitsu AP1000+, a distributed memory machine located at the Australian National University. It must be stressed that to make both parallel and sequential jobs to be executed effectively on a parallel system is not a simple task. A lot of problems must be solved through both theoretical and experimental studies which will be our future work.

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] 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.

[3] 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.

[4] 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.

[5] H. M. Deitel, *An Introduction to Operating Systems*, 2nd ed., Addison-Wesley, Massachusetts, 1990.

[6] A. C. Dusseau, R. H. Arpaci and D. E. Culler, Effective distributed scheduling of parallel workloads, *Proceedings of ACM SIGMETRICS'96 International Conference*, 1996.

[7] 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.

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

[9] 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.

[10] L. Lamport, concurrent reading and writing of clocks, *ACM Transactions on Computer Systems*, 8(4), April 1990, pp.305-310.

[11] 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.

[12] J. C. Mogul and A. Borg, The effect of context switches on cache performance, *Proceedings of 4th International Conference on Architect. Support for Prog. Lang. and Operating Systems* Apr. 1991, pp.75-84.

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

[14] A. Tucker and A. Gupta, Process control and scheduling issues for multiprogrammed shared-memory multiprocessors, *Proceedings of the 12th Symposium on Operating Systems Principles*, Litchfield, AZ, Dec 1989, pp.159-166.

[15] F. Wang, H. Franke, M. Papaefthymiou, P. Pattnaik, L. Rudolph and M. S. Squillante, A gang scheduling design for multiprogrammed parallel computing environments, *2nd Workshop on Job Scheduling Strategies for Parallel Processing*, April 16, 1996, Honolulu, Hawaii.

[16] 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.