The Design of a Supporting Environment for On-Line Parallel Debugging[*]

Zhou B. B., Brent R. P. and Qu X.
E-mail: bing,rpb,quxun@cslab.anu.edu.au
Computer Sciences Laboratory
The Australian National University
Canberra, ACT 0200, Australia
Phone: +61-6-2798644
Fax: +61-6-2798645

Abstract

On-line parallel debugging can provide very accurate and reliable information in diagnosis of parallel programs. Unfortunately, commercially available tools for on-line parallel debugging are hardly seen. This is mainly due to the lack of a suitable environment for multiprogramming of mixed parallel and sequential workloads so that the resources cannot be utilised efficiently. In this paper we present a two-level scheduling scheme for mixed parallel and sequential workloads on parallel machines – a key step in the establishment of a proper environment for on-line parallel debugging.

1 Introduction

On-line parallel debugging is defined as debugging a parallel program during its execution on a *real* parallel system. It is in contrast to *off-line parallel debugging* which means the debugging of parallel programs is done by running a simulator on a sequential machine, or by accumulating trace data during the program execution and then analysing them post-mortem.

There are several problems associated with off-line parallel debugging. When a simulation is carried out on a sequential machine, parallel computation is actually serialised. Then the simulation may not exactly match the real situation. Problems caused by asynchronous events (e.g. asynchronous messages between different processors on distributed memory machines and critical sections to be updated by several processes on shared memory machines) may not be detected. Another example is that the transient behavior of a parallel program may not be observed when trace-based methods are applied. With on-line parallel debugging users can control the execution of their parallel programs, for example, users can freely set breakpoints in their programs. On-line parallel debugging tools then allow us to pinpoint precisely where the problems are just like sequential debugging tools. Therefore, on-line debugging will be more accurate and reliable than off-line debugging.

On-line parallel debugging requires both *parallel* and *sequential* processing and is a typical procedure consisting of *computing* and *thinking* stages. To make on-line debugging work properly and the resources of a given parallel system be used efficiently, we need an interactive environment for multiprogramming of mixed parallel and sequential workloads. Unfortunately, currently existing parallel systems do not provide such an environment and many only support parallel batch jobs. On-line debugging will then be very expensive on this kind of system.

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. With this type of system it is thus possible for us to establish an environment suitable for on-line parallel debugging.

There are many problems associated with the design of a user-friendly on-line parallel debugger and the required supporting environments. In this paper we only discuss an effective scheduling scheme for mixed parallel and sequential workloads, a key issue relating to the establishment of the required interactive environment.

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.

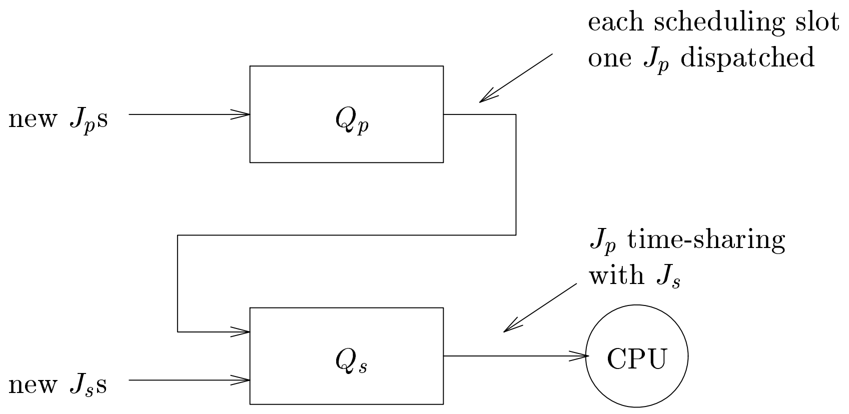
The paper is organised as follows. Section 2 briefly discuss related work. In Section 3 we describe the basic structure of our two-level scheduling system. The key feature is the introduction of registration office. With a registration office introduced on each processor parallel workloads can be serviced coordinately across the processors and parallel and sequential processes are also allowed to time-share the resources on each individual processor. The issue of tuning the system to meet the needs of on-line debugging is discussed in Section 4. The conclusion is given in Section 5.

2 Related Work

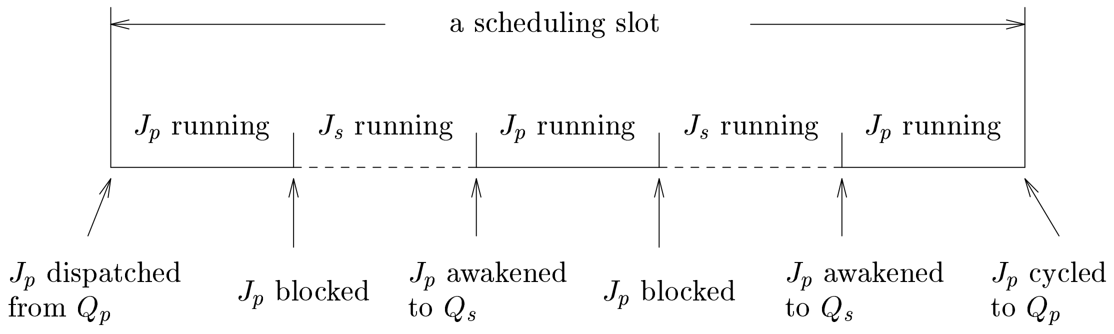
Many scheduling schemes for multiprogramming on parallel machines have been proposed in the literature. They can roughly be classified into two different types. The first type is called *local scheduling*, or *blocking*. With simple local scheduling parallel processes are not distinguished from sequential ones on each processor (except higher (or lower) priorities may be given to processes associated with parallel jobs). All processes are scheduled locally and there is no coordination between processors in scheduling 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, 6, 7, 10]. One method to alleviate this problem is to use *two-phase* blocking [12] or *adaptive two-phase* blocking algorithms [5]. In this method a process waiting for communication spins for some time, 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, the system performance will be degraded for fine-grain parallel programs and it is not clear how effective the method is for scheduling mixed parallel and sequential workloads.

Although multiprogramming of mixed parallel and sequential workloads is achieved by using local scheduling, a serious problem when used for on-line debugging is that parallel jobs cannot be executed in a coordinated manner. It is required in parallel computing that right data should be obtained from the right place at the right time. When timing is a critical issue for effective debugging, coordination in scheduling parallel workloads becomes a must. Thus local scheduling may not well suit on-line parallel debugging.

The second type of scheduling is called *Coscheduling* [11] (or *gang scheduling* [6]). With this type of scheduling 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. Parallel programs take turns to receive the service in a coordinated manner across the processors. However, a significant drawback of the conventional coscheduling 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 method will waste processor cycles and greatly decrease the efficiency of processor utilisation and is not suitable for on-line parallel debugging.



(a)



(b)

Figure 1: (a) A two-level scheduling scheme and (b) The normal situation in a scheduling slot.

3 The Two-Level Scheduling

In this section we describe an effective system for scheduling mixed parallel and sequential workloads on parallel machines. The system design is based on two principles, that is, first parallel workloads can be scheduled in a coordinated way so that they will not severely interfere with each other, and second both parallel and sequential workloads may time-share resources on each processor so that the efficiency of processor utilisation can be enhanced. Thus parallel workloads need to be scheduled at two different levels. At the first, or *global* level parallel workloads are coscheduled across the processors, while at the second, or *local* level processes associated with parallel jobs then time-share resources with sequential processes on each processor.

The basic structure of the two-level scheduling scheme on each processor is depicted in Fig. 1(a). This system consists of two queues, a queue Q_p at the first, or *global* level and a conventional, or sequential queue Q_s at the second, or *local* 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 will *time-share* the service with sequential processes. Assuming that parallel processes have higher priorities than sequential ones, they can immediately obtain the service once entering the sequential queue. Unlike the conventional coscheduling, parallel processes will be blocked during communication/synchronisation and then sequential processes can be serviced, as shown in Fig. 1(b). 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. 1(b).

Since parallel processes will 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 may be either in *running* state, or *ready* and *blocked* states just like sequential processes. The situation in Fig. 1(b) 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.

To avoid complicated procedures for searching missing processes we use a linked list which is called a *registration office*, as shown in Fig. 2. When a parallel job is initiated, each associated process will enter the conventional queueing system the same way as sequential processes on the corresponding processor. However, it 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. If nodes associated with the same parallel job are always added at the same place in each list, the linked lists on different processors will at any time remain identical in terms of the order of parallel workloads.

There is a *servant* working in the office. The servant has a pointer P. When this pointer points to a node in the linked list, the process associated with that node is said *being dispatched* and

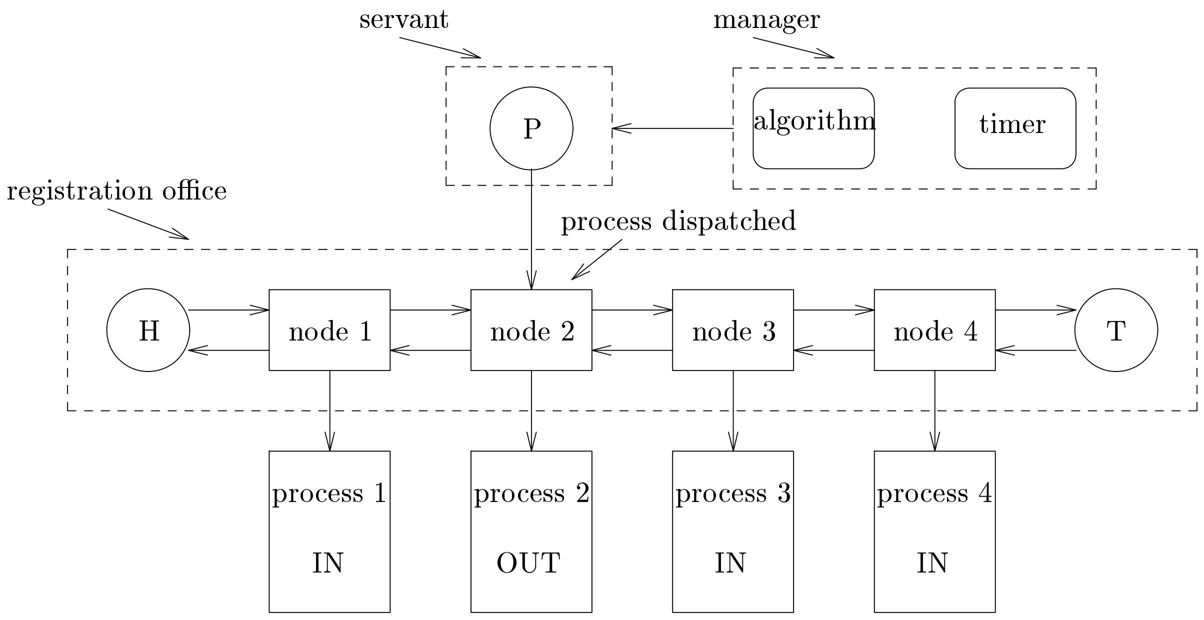


Figure 2: The organisation of a registration office.

can then enter the sequential queue and receive a service. (In practice there may be two pointers, one pointing to the process currently being served, one to a process to be served next.) When a process is *dispatched* from the parallel queue, it will be marked *out*. Other process not pointed by the pointer will be marked *in* so that they are kept in the parallel queue. By using the registration office dispatched processes are never detached from the office and then there are no search procedures required.

Parallel processes in the conventional queueing system also operate the same way as sequential processes. They are either in running state, or in ready state requesting for service, or, in blocked state during communication/synchronisation. If a parallel process is marked *in*, however, it cannot obtain the service whether it is in ready state or not. Therefore, the correct coscheduling is guaranteed. It may be more efficient in practice that all parallel processes marked *in* are *blocked*. A parallel process can come out of the *blocked* status only if it is *ready* for service and marked *out*.

When the scheduling slot is ended for the current process, the servant is moved to a new place, or pointer P is shifted to point to a new node. The associated process can then be served next. However, the movement of the servant is totally controlled by an *office manager* which has a *timer* to determine when the pointer is to move and an *algorithm* to determine which node the pointer is to point to. Thus coscheduling parallel processes becomes programming the pointer (the servant) to move around a linked list (the registration office).

The office manager can be either centralised, or distributed. The detailed description of this and other important issues relating to the enhancement of efficiency of this two-level scheduling can be found in [14].

4 The Interactive Environment

The two-level scheduling described in the previous section allows us to schedule both parallel and sequential workloads simultaneously. To establish an environment which is particularly suitable for on-line debugging, however, the system needs to be properly adjusted.

There may be several parallel programs running simultaneously in the system. Since good response to interactive users is the first priority in on-line parallel debugging, scheduling schemes such as *round robin* [9], or *fair share* [8] can be applied at the parallel level so that no program will be discriminated in obtaining the service. (It should be noted that these scheduling schemes may not be efficient if the target is to achieve short turnaround time [13].)

Suppose that there are ten parallel debugging programs running at the same time. Six of them are at the *checking stage*. Even though there is only four programs at the *computing stage*, they can only obtain the service once every ten scheduling slots if the round-robin scheduling is applied. There is no need to allocate scheduling slots to a program which is at the checking stage. This problem can easily be solved. In the information field of each node a one-bit information is added to indicate checking/computing status of the associated process. Before a scheduling slot is allocated to a process, the office manager will first check this information. If the program is at the checking stage. the servant will simply skip it and move to the next one which is at the computing stage. The four active programs in the above example can then obtain the service once every four scheduling slots. Therefore, the efficiency for executing parallel programs is greatly enhanced.

There coexist parallel and sequential workloads in the system. The local scheduler at the second level must ensure that neither of the two types of workloads will severely be obstructed in execution. Usually parallel processes are assumed to have higher priorities than sequential ones because they need coordination across processors.

Although each time there may be only one parallel process time-sharing resources with sequential processes on each processor, the execution of sequential workloads can still be seriously deteriorated when most of parallel workloads are fine-grained. In that case *dummy parallel jobs* can be introduced. A dummy parallel job is not a real job and the purpose is just to insert into each linked list a *dummy node* which is the same type of nodes as others in the list except its pointer points to NULL, the *constant zero*, instead of a real parallel process. When pointer P is pointering to a dummy node the whole scheduling slot will be dedicated to sequential processes. Therefore, the deterioration of system performance for sequential workloads can be alleviated.

It should be noted that the registration office can easily be extended to having multiple lists. This gives us a great flexibility in designing a system which can meet various needs of on-line parallel debugging.

Multiple lists may be required for on-line parallel debugging. The first one is just a *normal* list described previously. Processes attached on this list will be blocked during communication/synchronisation and coscheduling is applied to ensure that parallel processes of the same job will be executed at the same time across the processors.

To detect local errors users may wish to run the code and check information on each individual processor. Although different processors are used, the execution of a parallel job should be serialised. It is known that to activate, or dispatch a parallel process is just to mark it *out* using the registration office. When processes of the same job are to be executed sequentially, They are all marked *out* and given priorities which are no difference to those sequential processes. For the same reason as to identify checking/computing status, a one-bit information is also added in the information field of each node to indicate parallel/sequential status of a parallel process. When a process is in sequential status, no scheduling slots are allocated to it at the parallel level so that other processes which are active and in parallel status can be executed more efficiently. A new list can be added for detection of local errors without adding one-bit checking information. All processes attached on this list will be marked *out* and they can then be executed without any coordination. After the sequential operations, the associated processes will be placed back to the normal list.

To detect problems caused by asynchronous events, timing becomes a very critical issue and then batch processing is required. Another list can be applied to cope with this problem. When a scheduling slot is allocated, a process attached on this list will be given a very high priority and busy-waiting is applied during communication/synchronisation to ensure that the whole slot is dedicated to just that process. Coscheduling is also applied so that processes of the same job can run simultaneously across the processors.

5 Conclusions

On-line parallel debugging can provide very accurate and reliable information in diagnosis of parallel programs. Unfortunately, commercially available tools for on-line parallel debugging are hardly seen. This is mainly due to the lack of a suitable environment for multiprogramming of mixed parallel and sequential workloads. With the rapid advance of computer technology it is now possible for us to establish such an environment. In this paper we then presented a two-level scheduling scheme for mixed parallel and sequential workloads on parallel machines. With the introduction of registration office we can have a great flexibility in tuning the system to meet various needs of on-line debugging.

It should be noted that there are other problems closely relating to the establishment of an effective environment for multiprogramming of mixed parallel and sequential workloads. For example, one problem is memory management. On conventional parallel systems parallel jobs are not swapped because the speed of disks is relatively very slow and then swapping may severely deteriorate the performance. With multiprogramming, however, there may be numbers of parallel and sequential jobs time-sharing the resources at the same time. Because the memory space on a parallel system is limited, effective swapping and/or paging techniques need to be studied. Data produced by the same job may be distributed to different disks. Then another problem is how to store data so that they can easily be accessed and displayed to users in a proper manner at the run time.

To make the system work effectively in practice, extensive experiments are required. That 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] A. C. Dusseau, R. H. Arpaci and D. E. Culler, Effective distributed scheduling of parallel workloads, *Proceedings of ACM SIGMETRICS'96 International Conference*, 1996.

[6] 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.

[7] 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.

[8] G. J. Henry, The fair share scheduler, *AT & T Bell Laboratories Technical Journal*, 63(8), Oct. 1984, pp.1845-1858.

[9] L. Kleinrock, *Queueing Systems, Volume II: Computer Applications*, Wiley-Interscience, New York, 1976.

[10] 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.

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

[12] 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.

[13] B. B. Zhou, R. P. Brent and X. Qu, An efficient scheduling algorithm for multiprogramming on parallel computing systems, submitted to *20th Australasian Computer Science Conference*, Sydney, Feb. 1997.

[14] B. B. Zhou, X. Qu and R. P. Brent, Effective scheduling in a mixed parallel and sequential computing environment, submitted to *The 11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997.