An Efficient Scheduling Algorithm for Multiprogramming on Parallel Computing Systems

Zhou B. B., Brent R. P. and Qu X.

Computer Sciences Laboratory
The Australian National University
Canberra, ACT 0200, Australia

{bing,rpb,quxun}@cslab.anu.edu.au

Abstract

In conventional coscheduling, or gang scheduling of parallel workloads a round-robin queueing algorithm is adopted and the length of scheduling slots is fixed. However, the characteristics of parallel workloads can be quite different from sequential workloads. The system may not perform effectively using the simple round-robin algorithm. In this paper we introduce a new queueing algorithm. Our new system consists of two queues, a service queue which can hold more than one processes and a waiting queue which has multiple levels. This system has several potential advantages over some conventional queueing systems in scheduling parallel workloads. For example, it may achieve a higher system throughput and also a higher cache hit ratio, so the problems encountered in conventional coscheduling are alleviated. The issue of implementation of our algorithm is also discussed.

Keywords Multiprogramming, process scheduling and parallel computing systems.

1 Introduction

Many scheduling schemes for multiprogramming on parallel machines have been proposed in the literature. The simplest scheduling method 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. 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 with each other for the resource 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 [1, 2, 5, 6, 10].

One method to alleviate this problem is to use *two-phase* blocking [14] or *adaptive two-phase* blocking algorithms [4]. 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 mixed parallel and sequential workloads.

Another method commonly used in practice is that the system only allows one or two parallel programs running at a time. Once a parallel job enters the service, it will continuously be serviced (and may time-share the service with sequential jobs) 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. However, it is desirable from the system performance point of view that short jobs should be allowed to run first.

Coscheduling [12] (or *gang scheduling* [5]) of parallel programs may be a better scheme in adopting *short-job-first* policy. Using this method a number of parallel programs are 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 obtain the service in a coordinated manner across the processors. Thus parallel programs never interfere with each other and short jobs are likely to be completed more quickly.

The characteristics of parallel workloads can be quite different from those of sequential workloads.

rpb169: In *Proceedings of the 20th Australasian Computer Science Conference*, **Sydney, Australia, February 5–7 1997, pp 336–345.**

For example, parallel workloads are usually associated with relatively larger data sets, and also require longer time to complete. With conventional coscheduling processes are cycled in a round-robin fashion to access the service and the length of scheduling slots is fixed. This simple approach may not be ideal in scheduling parallel workloads.

Assume that the scheduling slot is fixed at 30 seconds. It is likely that a job requiring a service of 10 seconds will take about 300 seconds, or 5 minutes to complete if there are 10 jobs running at the same time. Thus the scheduling slot should not be too long. The problem may be alleviated by reducing the length of the scheduling slot. With a very short scheduling slot, however, the overhead of context switch may dramatically be increased. For example, a job requiring a service of 3 minutes is context-switched 6 times during the computation if the scheduling slot is 30 seconds. If the length of the scheduling slot is reduced to 3 seconds, however, the job will encounter 60 context switches. This increase in total number may greatly degrade the performance. Therefore, it is desirable to give long jobs longer scheduling slots to reduce the overhead of context switch, but less number of times to access the service so that short jobs will not frequently be blocked by longer ones and may be completed quickly [3]. The lack of flexibility in varying the scheduling slot and in determining the nature of processes may make the conventional coscheduling algorithm not very efficient.

The next problem associated with conventional coscheduling is the cache reloading effect. When the processes in the system are sharing the service in a round-robin manner, the caches may lose any useful contents related to an earlier computation, which will result in a very low hit ratio [6, 11, 13]. Cache size has greatly been increased recently with the rapid progress of VLSI technology, which may alleviate the cache reloading effect to some extent. When there is a large number of processes in the system, however, the cache reloading effect can still be significant. To alleviate this problem we may allow only certain processes to be *more active* than the others in the queue. The data associated with those processes can then remain in the cache for a longer period of time and the processes are more likely to complete before the data is cached out. Therefore, the hit ratio may remain relatively high. However, this cannot be achieved by using simple round-robin scheduling. More sophisticated scheduling schemes have to be considered.

Another potential problem, which is also associated with the simple round-robin scheduling algorithm, can occur when a large number of long processes is sharing the service. Assume that there are 10 processes sharing the service in round robin and that each process requires a service of 5 minutes. Then it will take about fifty minutes for each process to complete. Obviously a *batch processing algorithm* will be much more effective in this situation.

The most significant drawback of the conventional coscheduling algorithm is that it is designed only for parallel jobs. 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 greatly decrease the efficiency of processor utilisation.

We have designed an effective system for scheduling mixed parallel and sequential workloads on scalable parallel computing machines [15]. The design of this system is based on two principles, that is,

- parallel programs may scheduled in a coordinated manner so that they will not severely interfere with each other and
- parallel programs should time-share resources with sequential programs so that the efficiency of processor utilisation can greatly be enhanced.

Thus there are essentially two levels of scheduling for parallel workloads. At the first, or *global* level parallel programs are scheduled in a coordinated way across the processors, while at the second, or *local* level processes associated with parallel jobs are scheduled together with processes associated with sequential jobs so that they can time-share resources on each processor. Multiprogramming on parallel systems will be effective if processes at both levels are efficiently managed. The overall structure of our scheduling system and a scheme for scheduling processes at the second level are described in [15]. In this paper we discuss how to effectively coschedule parallel workloads at the first level.

The paper is organised as follows. Section 2 briefly describes the organisation of a registration office which is used at the first scheduling level to coordinate parallel workloads across the processors. In Section 3 we analyse a simple two-queue scheduling system which is used to alleviate the cache reloading effect and to prevent a large number of long processes from sharing the service in round robin. The multilevel two-queue scheduling system is then derived in Section 4 and the issue of its implementation discussed in Section 5. The conclusions are given in Section 6.

2 The Organisation of a Registration Office

In this section we briefly describe how the system discussed in [15] handles parallel workloads. The two-level process scheduler on each processor consists of two queues, a *sequential* or *conventional*

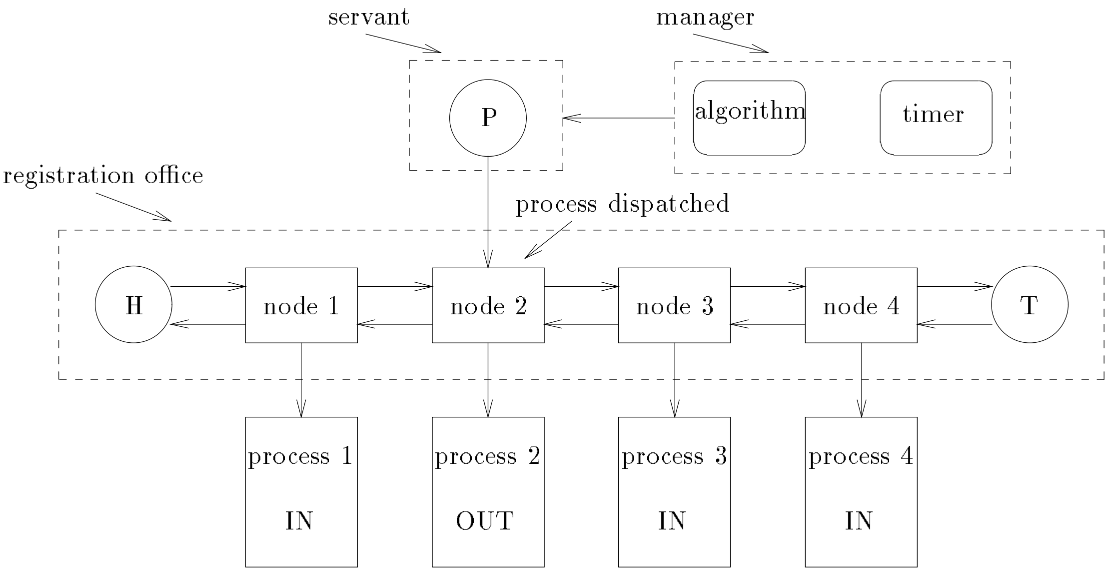

Figure 1: The organisation of a registration office.

queue and a queue, which is called *parallel queue* for coscheduling parallel workloads. The parallel queue is actually a *linked list* which is called *registration office*, as shown in Fig. 1. 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 can then enter the sequential queue and receive a service. A process is marked *out* if the corresponding node is pointed by pointer P, but all other processes are marked *in* so that they are kept in the parallel queue. Therefore, at any time there is only one process being dispatched.

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 workloads becomes programming the pointer (the servant) to move around a linked list (the registration office) on each processor.

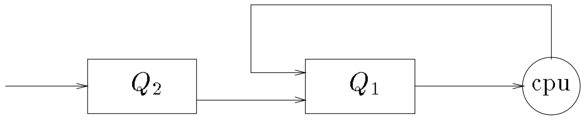

Figure 2: A simple two-queue system.

The office manager can be either centralised, or distributed. A detailed description of this and other important issues can be found in [15]. In the following sections we just focus on the design of efficient scheduling algorithms for parallel workloads to enhance the system throughput and to alleviate the cache reloading effect.

3 A Simple Two-Queue System

In this section we consider a simple two-queue system, as shown in Fig. 2. In this system there is a primary (or *service*) queue Q_1 which can only hold a maximum of N processes. The processes in this queue are serviced in a *round-robin* fashion. If there are more than N processes in the system, the remaining will be queued in a secondary (or *waiting*) queue Q_2 and can enter Q_1 only if the number of processes in Q_1 is less than N. It should be noted that all processes in the system are in runnable state. However, we purposely make the first N processes *more active*. The system becomes

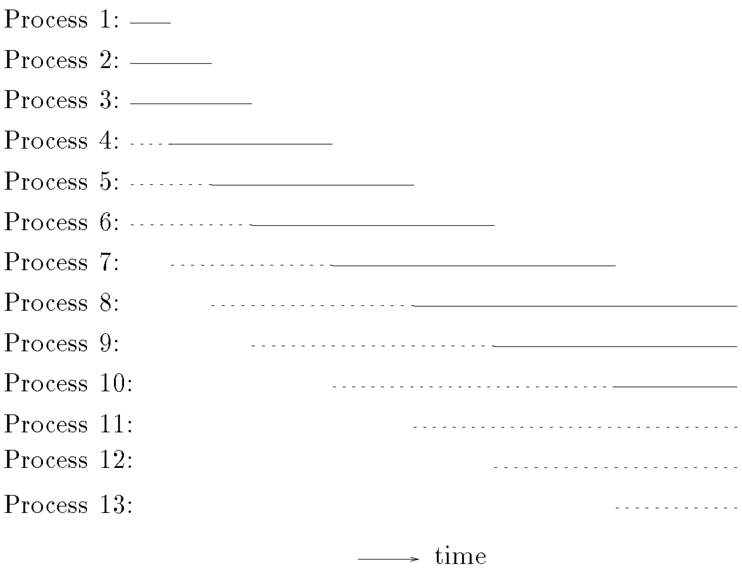

Figure 3: Time threads ($M = 6$ and $N = 3$). Solid (or dash) lines denote processes in Q_1 (or Q_2).

a simple round-robin queue if N is set to be infinity and a simple *batch* queue when $N = 1$.

Using this system each time we allow only N processes to share the service. If N is not large, the cache hit ratio may remain relatively high and the problem of large numbers of long processes sharing the service in round robin can also be avoided. In the following we do a simple analysis to see how the system throughput changes as N varies, which leads to our efficient scheduling system.

Assume that there are large numbers of processes in the system and that the scheduling slot for each process in the service queue is fixed. If a process requires a service of T_0 second for T_0 greater than one scheduling slot, the time spent by this process in the service queue is then

$$T_s^{(0)} = NT_0 \tag{1}$$

which includes a *required service time* T_0 and a *wasted time* $(N-1)T_0$ because the process always share the service with other $N-1$ processes in the service queue. It should be stressed that this does not mean that decreasing N will reduce the *turnaround time* of each process, or increase the system throughput. To obtain the *turnaround time* of each process we need also to calculate the time spent by each process in the waiting queue. When processes are placed in an increasing order in terms of their required service times, that is, $T_j \geq T_k$ if $j > k$ where T_i denotes the required service time of the i^{th} process, it can be seen that the system throughput will monotonically increase as N decreases. We give a simple deterministic analysis below.

In the following analysis we assume that the system is perfectly balanced, that is, at any time instant the number of processes in the system is a constant, say M for $M \geq N$, and that each process requires at least one scheduling slot to complete. To further simplify the discussion we also assume that $M = LN$ for L an integer greater than one.

It is known that the number of processes in the system is M at any time. The i^{th} process enters the waiting queue as the $(i-M)^{th}$ process departs from the system. At the same time the $(i-M+N)^{th}$ process is allowed to enter the service queue because the service queue can hold N processes. The i^{th} process can leave the waiting queue for the service queue immediately after the $(i-N)^{th}$ process departs from the system. The total time spent by the i^{th} process in the waiting queue is thus the time difference between when the $(i-N)^{th}$ process departs from and when the $(i-M+N)^{th}$ process enters the service queue. (The first N processes can enter the service queue without any delay. However,we are interested in more general cases, that is, the cases when $i > M$.) It can be figured out that the $(i-N)^{th}$ process enters the service queue immediately after the $(i-2N)^{th}$ process departs, the $(i-2N)^{th}$ process entered this queue immediately after the $(i-3N)^{th}$ process departed, and so on. Therefore, the time spent by the i^{th} process in the waiting queue is

$$T_w^{(i)} = \sum_{k=1}^{L-1} T_s^{(i-kN)} = N \sum_{k=1}^{L-1} T_{i-kN}. \tag{2}$$

The turnaround time of the i^{th} process is thus

$$T^{(i)} = T_w^{(i)} + T_s^{(i)} = N \sum_{k=0}^{L-1} T_{i-kN}. \tag{3}$$

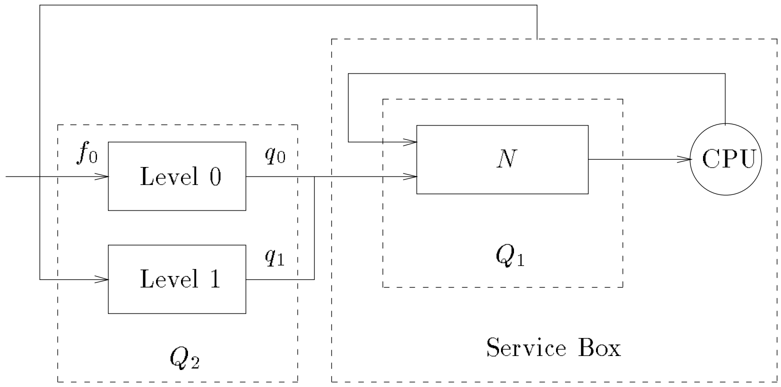

Figure 4: A modified two-queue system.

We give an example of $M = 6$ and $N = 3$ in Fig. 3. In this figure the sold lines denote processes in the service queue, while the dash lines denote processes in the waiting queue. At any time instant there are only six processes in the system, three in the service queue and the other three in the waiting queue. Since $L = M/N = 2$, it is easy to see that the time for the i^{th} process in the waiting queue is equal to the time for the $(i-3)^{th}$ process in the service queue, that is, $T_w^{(i)} = T_s^{(i-3)}$.

Given the equation in (3), we can show that the system throughput will decrease as N increases. Assume that

$$M = L_1 N_1 = L_2 N_2$$

and that

$$N_1/N_2 = L_2/L_1 = c$$

for c an integer greater than one. Let N_1 and N_2 be two different values for N. We then have

$$N_1 \sum_{k=0}^{L_1-1} T_{i-kN_1} = N_2 \sum_{k=0}^{L_1-1} cT_{i-kcN_2}$$

and

$$N_2 \sum_{k=0}^{L_2-1} T_{i-kN_2} = N_2 \sum_{k=0}^{L_1-1} \sum_{j=0}^{c-1} T_{i-(kc+j)N_2}.$$

Since processes are placed in an increasing order in terms of their required service times, then for $j > 0$ we have

$$T_{i-(kc+j)N_2} < T_{i-kcN_2}.$$

Thus

$$\sum_{j=0}^{c-1} T_{i-(kc+j)N_2} < cT_{i-kcN_2}.$$

Therefore, it is easy to see from the above equations that the i^{th} process will have shorter turnaround time when $N = N_2$.

More sophisticated analysis may be done to show that in general cases the system throughput will monotonically decrease as N increases if processes are placed in an increasing order in terms of their required service times. However, it is easy to see that, because short processes always enter the service queue before longer ones, they will share resources there with a smaller number of succeeding longer processes and can then be completed more quickly as N becomes smaller.

In the above discussion we assumed that processes enter the system in an increasing order based on their required service times. However, this condition can hardly be satisfied in practice. If processes are in an arbitrary order, the throughput of the two-queue system may not be as high as the simple round-robin system because the service queue may be filled with long processes and so short processes can be blocked in the waiting queue for a long time. This two-queue system is essentially a so called *selfish scheduling system* [8]. Theoretical studies show that this system may give short processes on the average even longer turnaround times than the simple round-robin system if processes are in an arbitrary order. However, the above discussion is still worthwhile because it tells us that the system throughput may greatly be enhanced if we can dynamically determine the nature of each process and arrange processes accordingly so that they enter the service queue in a more proper order. This motivates us to design a multilevel two-queue system which is discussed in the next section.

4 A Multilevel Two-Queue System

We first consider a modified two-queue system, as shown in Fig. 4. In this system the waiting queue now has two levels. New arrivals will first come to the higher level, level 0 where they are each assigned a number of scheduling slots f_0 with each slot being of length q_0. New arrivals enter the service queue in a *first-come-first-serve* order. When a new process is allowed to enter the *service box*, it will time-share with other $N-1$ processes in round robin and receive a service of f_0 scheduling slots. If it cannot be completed within f_0 scheduling slots, the process will be moved back to the waiting queue where it is placed at the lower level, level 1. To adopt the short-process-first policy, processes at level 0 are *implicitly* assigned higher priorities than those at level 1, and the values of f_0 and q_0 are chosen such that there is a sufficient time for an average short process to complete.

The service queue can only hold N processes. When the service queue is filled up and there is a new arrival, one process from level 1 has to be moved from the service queue back to the waiting queue because it has a lower priority. Thus there are two different types of process feedback from the service box to the waiting queue, that is,

- a process will be moved back to the waiting queue if it is not completed after f_0 scheduling slots and
- a process from level 1 will be moved out of the service queue when the service queue is full and there is a new arrival at level 0.

Now the tricky questions are

- where the process is to be queued when being moved back to the waiting queue and
- which process should be moved out of the service queue if there is a number of processes from level 1 and only one to be moved.

For a process having used up its f_0 scheduling slots we simply place it to the tail of the queue at level 1. This is because the process will no longer be considered as a short one, and so lose its priority. As a new comer to level 1, it should then follow the first-come-first-serve order.

For a process which is moved back from the service queue because there is a new arrival at level 0 we adopt a *longest-service-time-first* policy, that is, the process which has so far received the least service is moved out from the service queue and placed to the head of the queue at level 1. This decision is made under the consideration for reducing the cache reloading effect. When a process has been executed for a long time, it is hoped that most useful information may have been stored in the cache. Then the cache hit ratio will be high and this high hit ratio is also relatively stable if the system continues to give the service to that process. On the other hand, a process may still be in a *transit* state and then the cache hit ratio will be low if it is so far received only a short service. Thus a process should still be maintained in the service queue if it has already received a relatively longer period of service. For the same reason a process with the longest attained service should be dispatched first from level 1. With the longest-service-time-first policy processes are actually dispatched from the waiting queue to the service queue in a last-come-first-serve order. This is a significant difference from conventional multilevel scheduling systems which adopt at each level the first-come-first-serve policy [7].

To ensure that each process will receive a service of at least one scheduling slot once being dispatched to the service box, the feedback of processes from the service queue to the waiting queue is allowed only at the end of each *scheduling round* which contains N scheduling slots.

It is easy to see from the above discussion that our modified system works just like a simple two-queue system, except it gives new processes a special treatment, that is, it allows new processes to receive the service earlier. If it is a short one, a new process can be completed in f_0 scheduling slots without being blocked by long processes for a long time. If not, the new process will lose this priority, be moved back to level 1 of the waiting queue and then treated the same way as that in the simple two-queue system.

Our modified system may even be capable of discriminating more in favor of short processes than the simple round-robin system. This is because in the simple round-robin system short processes have to wait for longer time to receive the first service and then round-robin with a relatively greater number of processes to obtain further services if the number of processes in the system is large.

The two-queue system can be made more effective by adopting *multilevel feedback* technique [9]. This comes to our multilevel two-queue system, as depicted in Fig. 5.

In our multilevel two-queue system the service queue is the same as the one in the modified two-queue system. However, the waiting queue is further divided into multiple levels. A process at level i will be assigned a number of scheduling slots f_i, each having length q_i. When a process from level i has received a service of f_i scheduling slots, it will be moved back to the waiting queue, placed at the next lower level, level $i+1$ and further assigned f_{i+1} scheduling slots, each being of length q_{i+1}.

If we set $N = 1$ and $f_i = 1$ for all i, the system simply becomes a well known FB or *foreground-background* scheduling network [7]. In this system a process at a higher level is assigned a higher priority. Thus each time only the first one at the highest nonempty level is allowed to receive a service. After the scheduling slot is ended, the process will be preempted and placed at the next lower level if not completed. The most distinct feature of the FB scheduling algorithm is that it is capable of discriminating most in favor of short processes

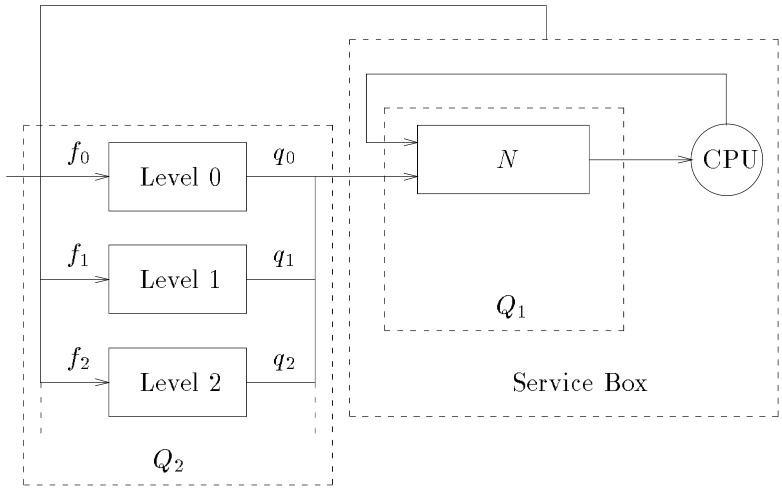

Figure 5: A multilevel two-queue system.

and thus can yield the highest system throughput. If the system always gives the service to the process with the least attained service, however, the waiting time for long processes may be arbitrarily large. This is a major problem associated with the FB scheduling algorithm.

Because of multiple levels our system has similar characteristics of the FB system, that is, a process at higher level will implicitly be assigned a higher priority. Thus each time only a process at the highest nonempty level can be dispatched to the service queue. Our multilevel system also works quite differently from conventional multilevel systems because the service queue can hold more than one process, there are processes moving back and forth between the service queue and the waiting queue and the movement is controlled by adopting the longest-service-time-first policy.

In general our multilevel two-queue system has several advantages over conventional queueing systems for scheduling parallel workloads. By using multiple levels the system is made more in favor of short processes and more capable of determining the nature of each process. Thus the system can achieve a higher system throughput than the simple round-robin system. Since the service queue can hold more than one process, processes from different levels can round-robin in the service box at the same time. Long processes may not suffer too heavily in contrast with the situation in conventional multilevel scheduling systems. The system has two queue and the longest-service-time-first policy is adopted. It tends to let certain processes be more active and then the cache reloading effect and the problem of large numbers of long processes sharing the service in round robin may also be alleviated.

By properly adjusting the parameters our multilevel two-queue system can be made very effective for coscheduling parallel workloads on parallel machines. If we set $q_0 = 5sec$ and $f_0 = 1$, then only processes which require a service of longer than $5sec$ will go to the lower levels. If we also set $q_1 = 15sec$, $f_1 = 8$, $q_2 = 25sec$, $f_2 = 7$ and so on, it is easy to see that a process which requires a service of five minutes may be preempted fifteen times. In contrast to conventional coscheduling where the scheduling slot is fixed, the cost for context switch can greatly be decreased. A simple example is that, if the scheduling slot is fixed to $5sec$, a process which requires a service of only two minutes may be preempted upto twenty four times in conventional scheduling.

The size of the service queue can also be made adaptive to further enhance the system performance. Assume that $N = 6$ and the parameters q_i and f_i are set as above. It is likely that the service queue is filled up with processes all from level 3 and below. If N is fixed, it is possible that a new process has to wait for over two minutes to receive its first service. This problem can be alleviated if we allow new arrivals to be inserted into the current scheduling round. At the end of each scheduling slot the system will check if there is a new process at level 0. If there is one, the next scheduling slot is assigned to that new arrival. If the process is not a short one, it will be moved back to the waiting queue and placed at the next lower level. The system will then continue the normal procedure for the current scheduling round. By adopting this special treatment to new arrivals, very short processes can be serviced and completed more quickly.

As we mentioned before, the system will perform badly when a large number of long processes is sharing the service in round robin. This problem has already been alleviated using our two-queue system. However, the results can further be im-

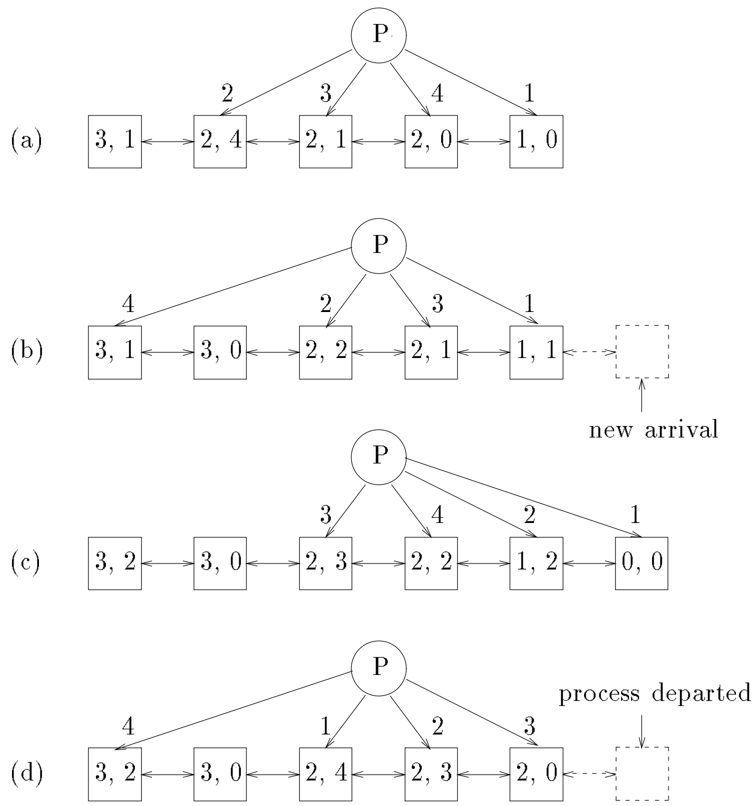

Figure 6: The movement of the pointer in four consecutive scheduling rounds.

proved by making the size of the service queue adaptive, that is, the size of the service queue may be decreased/increased when most of processes are at the lower/higher levels. Because the scheduling algorithm has to be deterministic when the distributed coscheduling scheme is applied [15], this adaptation can then be achieved by limiting the number of processes from each level to enter the service queue.

It should be noted that the parameters set in the above examples are purely imaginary. In practice extensive experiments are required in properly choosing the values for those parameters.

5 The Issue of Implementation

It seems that the multilevel two-queue system is complicated because there are two types of process feedback from the service box to the waiting queue. However, the system is simple to implement. In Section 2 we described that by introducing a registration office, scheduling processes becomes programming a pointer to move on a linked list. In the following we discuss how to move this pointer according to our multilevel two-queue scheduling algorithm. It should be noted that we can also use multiple lists for a multilevel queue, that is, each level is given an independent list. When a process is moved to the next lower level, it is first detached from the current list and then linked to a list which is associated with the new level. The use of a single list here is just for the description to be simpler.

To make the algorithm work correctly, we have to add two variables l and f to the information field of each node on the linked list. Variable l denotes the current level the associated process is at, while f denotes the service in number of scheduling slots the process has so far received since it comes to that level. The values of l and f will be updated each time the node is visited by the pointer. We also give the manager a *counter* which count scheduling slots for each scheduling round. With those items introduced the pointer is ready to move according to our scheduling algorithm.

It is better to show how the algorithm works by examples. In our particular example, as shown in Fig. 6, we assumed $N = 4$, $f_1 = 3$ and $f_2 = 5$. (Other parameters are not important in our discussion). The figure shows four consecutive scheduling round. The two numbers written in each node are the corresponding values of l (the first one) and f (the second one), while the number associated with each arrow denotes the order in which the pointer P is to move within each scheduling round.

In Fig. 6(a), the first of these four consecutive scheduling rounds, there are five nodes which associate five processes. Each node has values of l and f

which will be updated each time the node is visited by the pointer. After the first round, for example, the values in the rightmost node is updated from (1, 0) to (1, 1), while those in the leftmost node remains unaltered because the node is not visited in that scheduling round. Since a process at a higher level is given a higher priority, the pointer must go to the highest level at the beginning of each round. At the same level the pointer always moves from left to right because the longest-service-time-first policy is adopted. Note that the value in the second leftmost node is updated from (2, 4) to (3, 0) as the associated process has received the service of $f_2 = 5$ scheduling slots and so is moved to the next lower level of the waiting queue. At that level ($l = 3$) a process associated with the leftmost node has already received a service of one scheduling slot. Thus the pointer will first visit the leftmost node when coming to that level, as shown in Fig. 6 (b). When reaching the rightmost node of the current level before the end of the scheduling round, the pointer will visit nodes at the next lower level. This is to ensure that each process, once entering the service queue, can receive a service of at least one scheduling slot. When the size of the service queue is fixed, or nonadaptive, a new arrival has to wait in the waiting queue before the current scheduling round is ended (Fig. 6 (b)). Since the new arrival is placed at the highest level, it will be serviced first in the next scheduling round (Fig. 6 (c)). Assume that this new arrival is a short process and can complete within just one scheduling slot. The system will then return to serve long processes. It can be seen that the same processes are serviced in the second and the fourth scheduling round (Fig. 6 (b) and (d)). If the cache size is large enough, the cache may lose little useful contents to the process associated with the leftmost node. Then a relatively high cache hit ratio may be obtained. This is a potential advantage over conventional multilevel scheduling schemes which may give the service to processes at the same level in a round-robin manner.

To summarise, the pointer always points to the leftmost node of the highest level at the beginning of each scheduling round and then move rightward to visit the nodes at the same level. When it reaches the end of the current level, the pointer will come to the next lower level and travel the same way. The value of f is incremented by one each time the node is visited by the pointer. When reaching the required number, f is reset to zero and l incremented by one. The associated process will then restart at the next lower level. The procedure continues until the current scheduling round is ended. A new round is scheduled exactly the same way.

In order to make the algorithm work correctly the counter has to be reset once the pointer reaches the head node of the linked list so that the proper order for l and f can be maintained, that is, the values of l, as well as f at the same level, are always in a nonincreasing order in the linked list.

Finally, it should be noted that the procedure will become a bit more complicated when the size of the service queue is made adaptive.

6 Conclusions

The characteristics of parallel workloads are usually different from those of sequential workloads. Problems may occur when the simple round-robin queueing system is adopted in coscheduling parallel workloads on parallel computing machines. To alleviate those problems we derived a new queueing system. Our new system consists of two queues, a service queue which can hold more than one process and a waiting queue which has multiple levels. This system has several potential advantages over some conventional systems in coscheduling parallel workloads:

1. By adopting multilevel feedback technique the system is made more in favor of short processes and can then obtain a higher system throughput than the simple round-robin system.

2. Because the service queue can hold more than one process, several processes from different levels can receive service in the same scheduling round. Then long processes may not suffer too heavily as is the case in applying conventional multilevel systems.

3. The system has two queues and the longest-service-time-first policy is adopted in moving processes back and forth between the waiting queue and the service queue, It tends to let certain processes be more active than the others. Thus a higher cache hit ratio may be achievable and the problem of large number of long processes sharing the service in round robin can be alleviated.

It is easy to see that the system becomes the simple round-robin if f_0 and N are set to infinity and the multilevel FB if $N = 1$ and $f_i = 1$ for all i as we mentioned previously. Thus the well known round-robin and multilevel FB systems are just special cases of our multilevel two-queue system. By properly adjusting the parameters the system can be made very effective in coscheduling parallel workloads on parallel computing machines.

Although the scheduling algorithm is more complicated than many existing ones, it is not difficult to implement. Extensive experiments, as well as further theoretical studies, will be carried out in the near future after this system is implemented on the Fujitsu AP1000+, a distributed memory machine located at the Australian National University.

References

[1] 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.

[2] 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.

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

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

[5] D. G. Feitelson and L. Rudolph, Gang scheduling performance benefits for fine-grained synchronisation, *Journal of Parallel and Distributed Computing*, 16(4), Dec. 1992, pp.306-318.

[6] A. Gupta, A. Tucker and S. Urushibara, The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications, *Proceedings of SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1991, pp.120-132.

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

[8] L. Kleinrock and J. Hsu, A continuum of computer processor-sharing queueing models, *Proceedings of the Seventh International Teletraffic Congress*, Stockholm, Sweden, June 1973, pp.431/1-431/6.

[9] L. Kleinrock and R. R. Muntz, Multilevel processor-sharing queueing models for time-shared models, *Proceedings of the Sixth International Teletraffic Congress*, Aug. 1970, pp.341/1-341/8.

[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. 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.

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

[13] 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.

[14] 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.

[15] 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.