# Reducing Contention for Run Queue in Shared-Memory Multiprocessor Systems

Sivarama P. Dandamudi
School of Computer Science
Carleton University
Ottawa, Ontario K1S 5B6
Canada
sivarama@scs.carleton.ca

**ABSTRACT**

Performance of parallel processing systems is sensitive to various hardware and software overheads and contention for hardware and software resources. Hardware resources such as interconnection network and memory introduce communication contention and memory contention that could seriously impact overall system performance. Software resources include critical data structures maintained by application software as well as by the system software.

In process scheduling context, run queue is a critical data structure that could potentially affect the overall system performance. There are two basic run queue organizations: *centralized* and *distributed*. Many shared-memory multiprocessor systems use the centralized organization in which a single global run queue is shared by all processors in the system. We will first identify performance problems associated with these two organizations and then discuss some means to alleviate these problems. We then present a hierarchical task queue organization that incorporates the merits of these two queue organizations. Performance of the hierarchical organization is shown to inherit the good characteristics of the centralized and distributed organizations while avoiding the pitfalls associated with them. Thus, the hierarchical organization is useful in building large-scale multiprocessor systems.

**Index Terms**: Hierarchical organization, Multi-processor systems, Parallel systems, Performance evaluation, Process scheduling, Run queue organization.

## 1. INTRODUCTION

Shared-memory multiprocessor systems are an important class of parallel processing systems. These systems provide the programmer a single address space much like that of a traditional uniprocessor system. Communication among processors is through shared memory variables. Shared-memory multiprocessor systems, often simply called multiprocessors, are currently evolving towards general-purpose, multiuser systems [Fei90].

We can divide shared-memory multiprocessors into either *uniform memory access* (UMA) systems or *non-uniform memory access* (NUMA) systems. In UMA multiprocessors, the cost accessing a memory location by a processor in the system is the same. In NUMA multiprocessors, however, the shared memory is physically distributed among the processors. Thus, some memory locations are closer to a processor than others resulting in a non-uniform memory access cost.

In a UMA multiprocessor the shared memory is global to all processors as shown in Figure 1. An interconnection network facilitates communication between processors and global shared memory. Typically, UMA multiprocessors use a single bus as the interconnection network. The Sequent Symmetry and Encore Multimax are examples of commercial bus-based UMA systems.

Using a common bus as an interconnection network severely limits system scalability (i.e., expandability of the system) because of bus bandwidth limitations. Furthermore, this type of interconnection network allows only one processor to communicate with the memory, leading to performance degradation. To avoid this problem, the shared memory is divided into $M$ memory modules (Figure 2). It is common to provide concurrent access to all memory modules, provided there are enough processors requesting access to memory modules and no two processors wish to access the same memory module. For example, such concurrent access can be provided if we use the crossbar network as an interconnect. However, its cost is O($MP$) making it too expensive for large multiprocessor systems with hundreds of processors. Large multiprocessor systems typically use a multistage interconnection network. For example, the NYU Ultracomputer uses an Omega switching network as the interconnect.

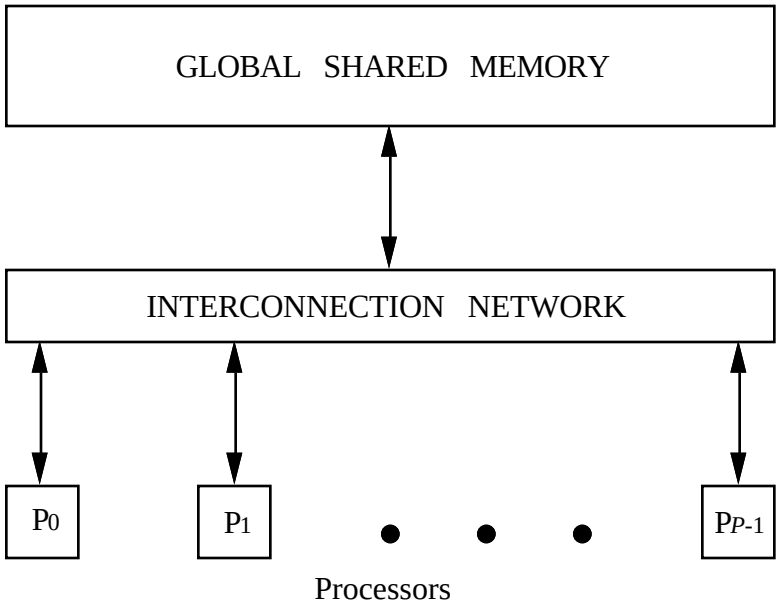



**Figure 1** A typical UMA multiprocessor system

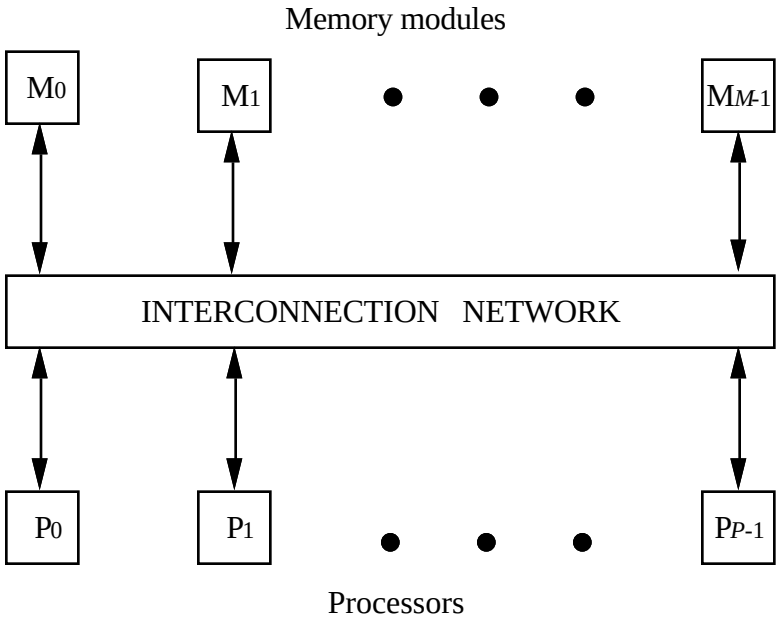



**Figure 2** A switching network-based UMA multiprocessor system

The *P* processors in the system can access these memory modules by using a multistage interconnection network. Usually, the number of memory modules *M* is equal to the number of processors *P* because if $M < P$, *memory contention* may occur if all processors request access to the shared memory [Ste88]. Since each memory module can typically service one access request at a time, having *P* memory modules is sufficient assuming that the interconnection network can connect the *P* processors to the *P* distinct memory modules. Note that, even with this concurrent access, memory contention can still occur if several processors concurrently request access to a memory module. Similarly, communication contention can also occur if several processors contend for a common link in the interconnection network. In order to reduce both memory contention and network contention, process-

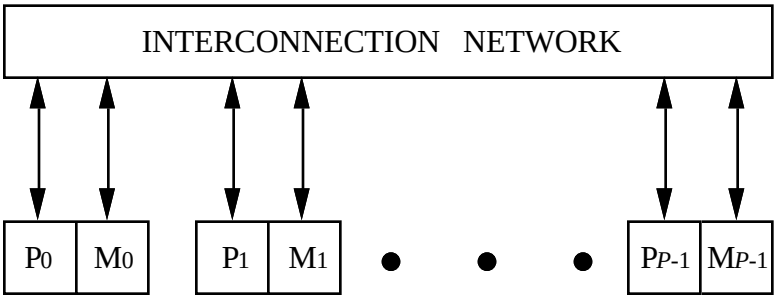



**Figure 3** A NUMA multiprocessor system

ors usually maintain some local memory that can be directly accessed by the associated processor without going through the interconnection network. For details on memory design techniques that reduce contention in shared-memory systems, the reader is referred to [Ste88].

In NUMA multiprocessors, the shared memory is physically distributed among the processors in the system as shown in Figure 3. In this architecture, memory module $M_i$ is local to processor $P_i$, $0 \le i \le P-1$. Thus, for example, processor $P_0$ can access a memory location in its local memory (i.e., in module $M_0$) without going through the interconnection network; whereas accessing a remote memory location (that is mapped to a memory module other than $M_0$) involves the interconnection network, and hence increased delay. Example NUMA multiprocessors include the Stanford DASH, Illinois CEDAR, Toronto's Hector and the BBN Butterfly systems.

An advantage of the NUMA type systems is that they can reduce the traffic on the interconnection network by exploiting locality in program reference behaviour. The local memory can be used to store parts of code and data that is frequently accessed. Therefore, NUMA architecture is suitable for large systems without putting undue demands on the interconnection network. It is generally believed that UMA systems are good for systems with tens of processors while the NUMA architecture allows building systems with hundreds of processors.

Even with the best system and memory design, memory contention and communication contention is still possible. A critical data structure maintained in the shared memory, for instance, can create this situation. Such critical data structures could be maintained either by application software or by system software. For example, if the system software maintains a single run queue of jobs waiting to be scheduled, it can potentially become a bottleneck for large systems leading to both memory and communication contention. Such a system bottleneck can lead to serious performance degradation. Our interest in this paper is to reduce contention for the run queue in the context of processor scheduling.

For small systems with, say, 10 to 30 processors, a single global run queue works fine. This is typically the case with UMA multiprocessors. However, this run queue organization is clearly not suitable for large systems. To avoid contention for the run queue in such large systems, each processor can maintain a local run queue. This distributed run queue organization, while avoiding the contention problem, creates load imbalance problem in which some processors may be idling while there is work to do at some other processors. In the next section, we start our discussion of run queue organizations with these two basic organizations. We will then discuss several techniques to mitigate some of the drawbacks associated with these organizations. However, individually taken, none of these techniques is completely satisfactory and introduce problems of their own. We will then introduce a hierarchical run queue organization that combines several of the techniques introduced to improve the performance of the two basic run queue structures. We will demonstrate that the hierarchical organization incorporates the merits of the centralized and distributed run queue organizations while minimizing the drawbacks associated with them.

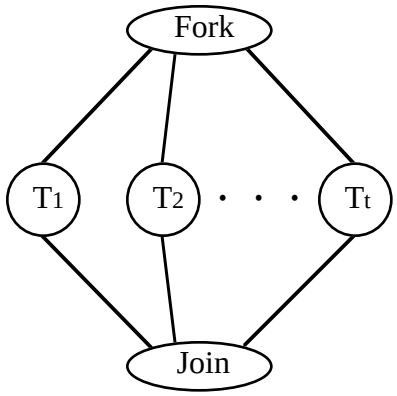



**Figure 4** Fork-and-join job structure

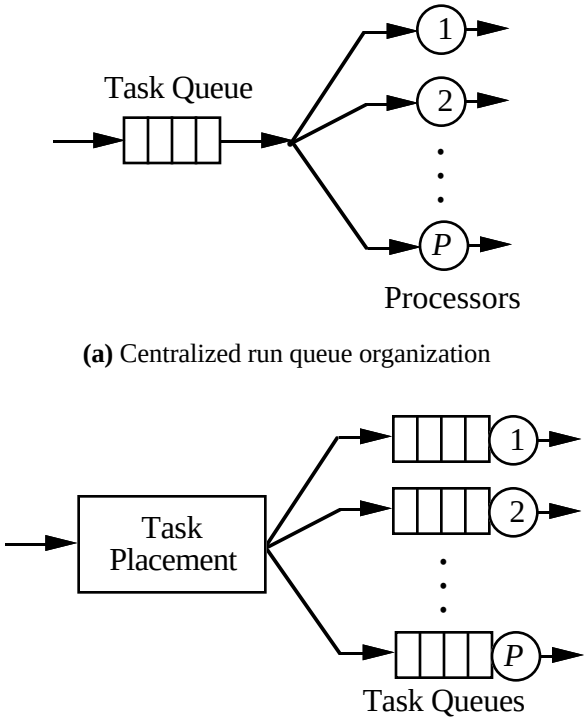
In this paper, we use the term *job* to refer to a parallel program. A job typically consists of several *tasks*, which are schedulable entities. Our focus in this paper is on the fork-and-join type of jobs whose structure is shown in Figure 4. As an example, consider multiplication of two $N \times N$ matrices: **C** = **A** * **B**. The matrix multiplication can be implemented as:

```
for i ← 1 to N
   for j ← 1 to N
      temp ← 0
      for k ← 1 to N
         temp ← temp + A[i][k] × B[k][j]
      end_for
      C[i][j] ← temp
   end_for
end_for
```

For example, if $N$ is 100 and we would like to execute the multiplication on a 10 processor system, we can create 10 tasks where each task would compute ten rows of the result matrix **C**. This results in an uniform allocation of work to all ten tasks. On the other hand, it is also possible to have tasks that have a non-uniformly distributed amount work even though the number iterations are evenly distributed. This, for example, would be the case if the matrices involved are sparse. In this case, arithmetic operations on zero elements can be avoided. In later sections, we will use this type of job structure to evaluate the performance of the run queue organizations.

## 2. TWO BASIC RUN QUEUE ORGANIZATIONS

In this paper, we use the terms task queue and run queue interchangeably. We have briefly alluded to the two basic run queue organizations shown in Figure 5: *centralized* organization and *distributed* organization. In a centralized organization, there is a single global queue of ready tasks that is accessible to all processors in the system. On the other hand, in a distributed organization, private run queues are associated with the processors. In the distributed organization, each arriving task is placed in a queue depending on the task placement policy that is in effect. A simple task placement policy assigns each arriving task to a random queue. A better placement policy would assign tasks in a cyclic fashion. That is, if the last task has been assigned to task queue $i$, the next task will be assigned to ($i$+1 **mod** $P$) task queue, where $P$ is the number of processors in the system. This placement policy is assumed in the remainder of the paper.




**(a)** Centralized run queue organization

**(b)** Distributed run queue organization

**Figure 5** The two basic run queue organizations

Access to a run queue should be allowed on a mutually exclusive access basis. Since each access takes a finite amount of time, the centralized organization can lead to access contention for sufficiently large number of processors. In this paper, we model the task queue access contention by a single parameter $f$, which represents the task queue access time as a fraction of the average task execution time $S$. For instance, when we say $f$ = 1%, we mean that the task queue access time is 1 per cent of the average task execution time. That is, the queue access time is $0.01S$.

### Performance Comparison

The relative performance of the centralized and distributed queue organizations is shown in Figure 6. The y-axis is the response time ratio $Rd/Rc$, where $Rd$ and $Rc$ are the average response times of the distributed and centralized organizations, respectively. These results were obtained for a 64-node system with exponential task service times and Poisson job arrivals. The run queue access time, as modelled by $f$, is varied from 0% to 4%. The scheduling policy is assumed to be FCFS (first-come/first-served) policy. This is a non-preemptive policy in which tasks run to completion once they are scheduled.

In Figure 6, a point above the unity line (shown dotted) indicates that the centralized organization is better and below this line indicates that the distributed

organization is better. A key observation is that when the queue access time is negligible (i.e., $f$ = 0%), the centralized organization provides a better performance than the distributed organization. This improvement increases with increasing utilization. The main reason for this performance superiority is that the centralized organization provides perfect load sharing. However, as the queue access time increases, the central run queue becomes a bottleneck affecting the performance. Thus, for higher values of $f$, the distributed organization provides better response times.

In the centralized organization, the run queue can become a bottleneck for smaller values of $f$ as the system size increases. It has been shown in [Dan91] that the $f$ values for which the run queue becomes the bottleneck is given by

$$f > \frac{1}{2P - 1} \quad (1)$$

where $P$ is the number of processors in the system. Some sample $f$ values above which the run queue becomes the bottleneck are shown in Table 1. Note that this analysis assumed a non-preemptive FCFS scheduling policy. If, on the other hand, a preemptive scheduling policy like the round robin or processor sharing is used, the upper bound on $f$ values will be much lower than the values shown in Table 1. This is because, preempted tasks are returned to the run queue, thereby increasing the frequency of run queue access. Thus, the centralized organization is good for small multiprocessor systems. This organization is widely used in several multiprocessors including Dynix on the Sequent Symmetry and Mach on the Encore Multimax.

**Table 1** Sample $f$ values above which the run queue becomes a bottleneck

| $P$ | $f = \frac{1}{2P - 1}$ |
|---|---|
| 8 | 6.667% |
| 16 | 3.226% |
| 32 | 1.587% |
| 64 | 0.787% |
| 128 | 0.392% |
| 256 | 0.196% |
| 512 | 0.098% |
| 1024 | 0.049% |

The distributed organization avoids this run queue bottleneck problem by adding more queues to the system as the system expands. The problem with this organization, however, is that it can cause load imbalance resulting in poorer performance as shown in Figure 6 (see $f$ = 0% line). This load imbalance causes severe performance problems as the variance in task service times increases. The impact of variance in task service time (expressed as the coefficient of variation (CV)) is shown in Figure 7 for 70% system load.

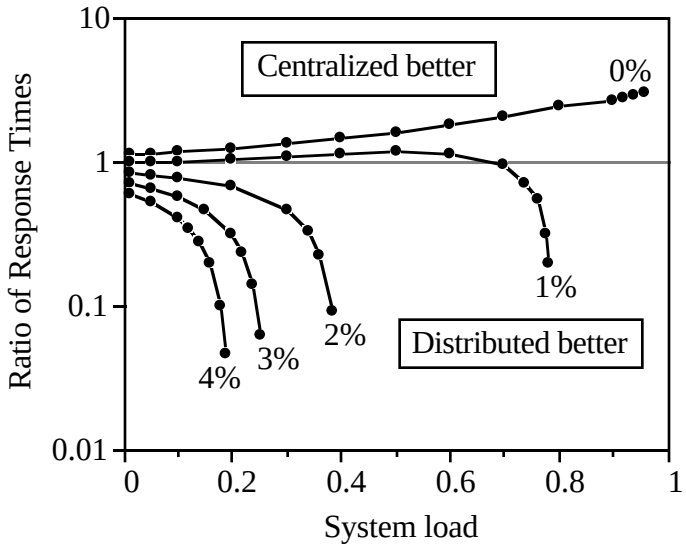



**Figure 6** Performance of the centralized and distributed organizations as a function of system load ($f$ value is varied from 0% to 4%)

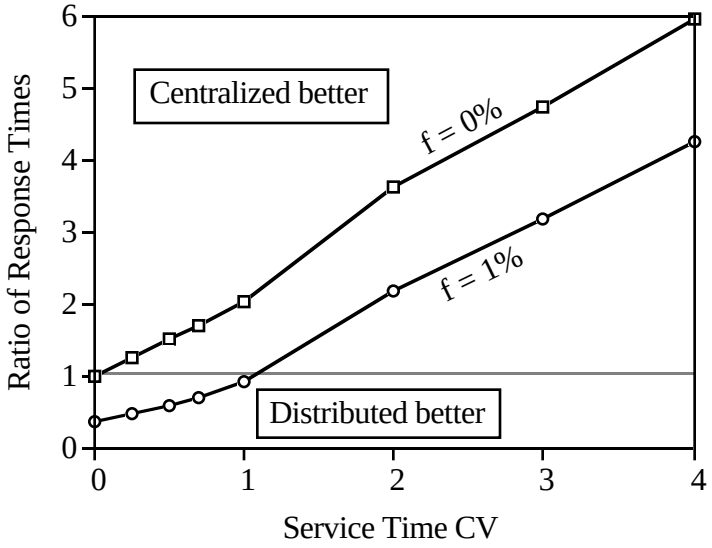



**Figure 7** Performance sensitivity to service time variance

We can see that, when there is no access contention problem (i.e., $f$ = 0%), the relative performance of the distributed organization deteriorates with increasing variance in task service times. For non-negligible run queue access times (as indicated by the $f$ = 1% line), the distributed organization performs better for lower variances in task service times. Intuitively, this makes sense as the cyclic task placement policy is assumed for the distributed organization.

## 3. TECHNIQUES TO IMPROVE PERFORMANCE

Now let us see how we can improve the performance of the centralized and distributed organizations. In the centralized organization, we have to devise ways to eliminate/minimize the run queue contention problem. In the distributed organization, we need strategies to eliminate/minimize load imbalance. While we discuss several techniques to do this, none of them is acceptable by itself. Later, we will effectively combine some of these techniques to derive a new run queue organization.

### 3.1. IMPROVING PERFORMANCE OF CENTRALIZED QUEUE

The goal here is to minimize the run queue access contention problem. To achieve this, we have to find ways to decrease the frequency of access to the run queue. We present two methods discussed in [Nel90] to do this while maintaining a single global run queue. The basic principle behind these two methods is that, whenever the run queue is accessed, remove larger chunks of work. Both these methods require that the processors maintain local queues.

The first method that we discuss is called *autonomous* policy [Nel90]. In this policy, every time a processor accesses the run queue, instead of removing only a single task from the queue, it removes a set of tasks. In other words, processors self-schedule larger chunks of work, which results in fewer trips to the run queue. It is important to control the set size appropriately. Removing a constant number of tasks can result in load imbalance leading to performance problems [Nel90]. One reason for this is that the optimal set size is a function of the system load and increases with system load [Nel90]. Furthermore, performance deterioration can also occur if there is a large variance in task service times.

The second method to reduce access contention to the global queue is called the *cooperative* policy [Nel90]. In this policy, when a processor accesses the run queue, it not only schedules tasks for itself but also schedules tasks for other processors in a cooperative manner. The number of tasks scheduled on each processor is dynamically adjusted depending on the system load. The policy proposed in [Nel90] moves tasks from the central queue to processors local queues in a “join the shortest queue” fashion. As a result, this policy provides better load balance than the autonomous policy. The analysis presented in [Nel90] shows that the cooperative policy performs better than the autonomous policy and distributed organization. A problem with the cooperative policy is that it is difficult to implement, particularly for larger systems. The difficulty arises as we have to maintain state information on processor local queues in order to implement task movement from the global queue to local queues on a shortest queue basis.

Another way to reduce contention for the run queue is to use multiple run queues. Ni and Wu [Ni89] have proposed a method that maintains $m$ run

queues, where $m < P$. The $P$ processors in the system are partitioned into $m$ groups. Each group of processors is served by a dedicated run queue. Each arriving task is placed randomly in one of the $m$ queues. The number of queues $m$ represents a tradeoff between load sharing and access contention. If we use a small value for $m$, it increases load sharing but increases access contention for the queues. Using a large value has the opposite effect. Obviously, if we let $m = P$, this organization corresponds to the distributed organization. Ni and Wu have presented a method to determine the optimum value for $m$. However, this solution is not satisfactory as the optimum value for $m$ is system load dependent. In addition, it can also cause load imbalance problem. The extent of this load imbalance problem depends on the value of $m$. To reduce the load imbalance problem, we can use the same techniques that are useful in the context of distributed queue organization, which are discussed next.

### 3.2. IMPROVING PERFORMANCE OF DISTRIBUTED QUEUE

To improve the performance of the distributed queue organization, we have to tackle the load imbalance problem. We will discuss two methods to reduce the load imbalance. The first method tries to avoid load imbalance by distributing tasks more intelligently than the random or cyclic placement policies. In the other method, when load imbalance occurs, an idle processor tries to get work from other processors.

Studies have shown that the cyclic placement policy substantially improves the performance of the distributed queue organization over the random placement policy [Dan91]. We can improve the performance further by using an adaptive placement policy such as the shortest queue placement policy. In this placement policy, when a task arrives, it is routed to the queue that has the shortest queue among the $P$ queues. Even with this placement policy, load imbalance occurs when the variance in task service times is high.

We can reduce the adverse effect of high variance in task service times if we know task service times (or at least a good estimate of it) *a priori*. Then, we can rout each arriving task to a queue that has the smallest waiting time (which is the sum of the task service times of the tasks waiting to be scheduled in a queue). When we know exact the task service times *a priori*, using this shortest waiting time placement policy gives performance that is very close to that of the centralized organization. In practice, however, it is difficult to obtain a good estimate of task service times *a priori*.

Both these adaptive placement policies provide substantial performance improvements, but there are implementation problems associated with collecting queue state information [Dan91]. Another strategy to cope with the load imbalance problem is to use a simple and implementable task placement policy such as the cyclic placement policy and handle load imbalance when it occurs. In this strategy, if a processor finds its local queue empty, it then probes the local queues of other processors for tasks [And89, Nel90]. This is similar to what is called the *receiver-initiated* load sharing strategy used in distributed systems [Shi92]. For example, an idle processor can randomly select a processor and probe the state of its local task queue. If the queue is empty, select another processor randomly and repeat the process for a maximum of some predefined probe limit value or until a task arrives at its local queue; otherwise, get a task from that processor's local queue. The analysis presented in [Nel90] shows that while probing other processor queues reduces load imbalance considerably, it does not eliminate the problem. Thus, despite the suggested improvements, load imbalance continues to be a problem with the distributed organization. This leads us to the hierarchical organization, which is described next.

## 4. HIERARCHICAL RUN QUEUE ORGANIZATION

### 4.1. MOTIVATION

From the previous discussion, the following design trade-offs can be identified. The centralized organization is preferred when the task queue access contention is small or negligible. In this case, the load sharing characteristic of this organization is very attractive. Furthermore, the centralized organization is less sensitive to variance in task service times. However, the major drawback is that the central task queue can potentially become a bottleneck for large system sizes. It has been shown that the centralized organization causes performance problems even for small system sizes [And89].

The distributed organization, on the other hand, eliminates this task queue bottleneck disadvantage of the centralized organization by associating a private task queue with each processor. A chief disadvantage of this organization is that there may be load imbalances in the sense that some local task queues may be empty while others have tasks to be scheduled. This load imbalance causes performance problems, including high sensitivity to variance in task service times.

Ideally, therefore, we would like an organization that combines the desirable characteristics of both these organizations while avoiding the disadvantages associated with them (or at least substantially reduce their adverse effects on performance). That is, the goal is to have an organization that allows good load

sharing (as in the centralized organization) and that distributes task queues to eliminate the bottleneck problem (as in the distributed organization). This is the motivation for proposing a hierarchical organization [Dan95]. The hierarchical organization incorporates the performance improvements suggested to the basic centralized organization. Later we present some results comparing the performance of the three queue organizations and show that the hierarchical organization indeed achieves these objectives.

### 4.2. DESCRIPTION

In hierarchical organization, a set of task queues is organized as a tree with all the processors with their local queues attached to the bottom level of the tree (i.e., as leaf nodes). Figure 8 shows an example hierarchical organization for $P = 8$ processors with a tree branching factor $B = 2$. Each task queue can then be viewed as a task queue in the centralized organization serving only the tree nodes (these nodes could themselves be task queues or processor local queues) directly below it. The centralized organization can be considered as a special case of this structure with just one level (just the root node) having a branching factor equal to the number of processors in the system.

The use of hierarchy has been proposed previously as well. Anderson et al. [And89] use a two-level hierarchy that consists of a global pool of ready tasks with processors having local queues. Feitelson and Rudolph [Fei90] have also proposed a distributed hierarchical control for parallel processing, but they assume that specialized hardware is available for scheduling tasks.

In the hierarchical organization, all incoming tasks are added to the root task queue. Let $L$ be the leaf node level (i.e., processor level) in the tree. When a processor is looking for work, it first checks its associated task queue at level $(L-1)$. If that queue is empty it checks the parent node of this node at level $(L-2)$ and the process is repeated up the tree until it finds a task to be scheduled (unless the root queue is empty). However, in order to reduce access contention at higher levels, when a task queue is accessed, a set of tasks is moved one level down the tree (like in the autonomous policy). The size of the set decreases progressively as one goes down the tree (taking into the account the availability of the increasing number of task queues per level). At the bottom of the tree this set size is reduced to just one task (corresponding to scheduling that task on the associated processor). We use a parameter called the *transfer factor Tr* to indicate the number of tasks transferred from a parent queue to one of its child queues. The parameter *Tr* is defined as follows:

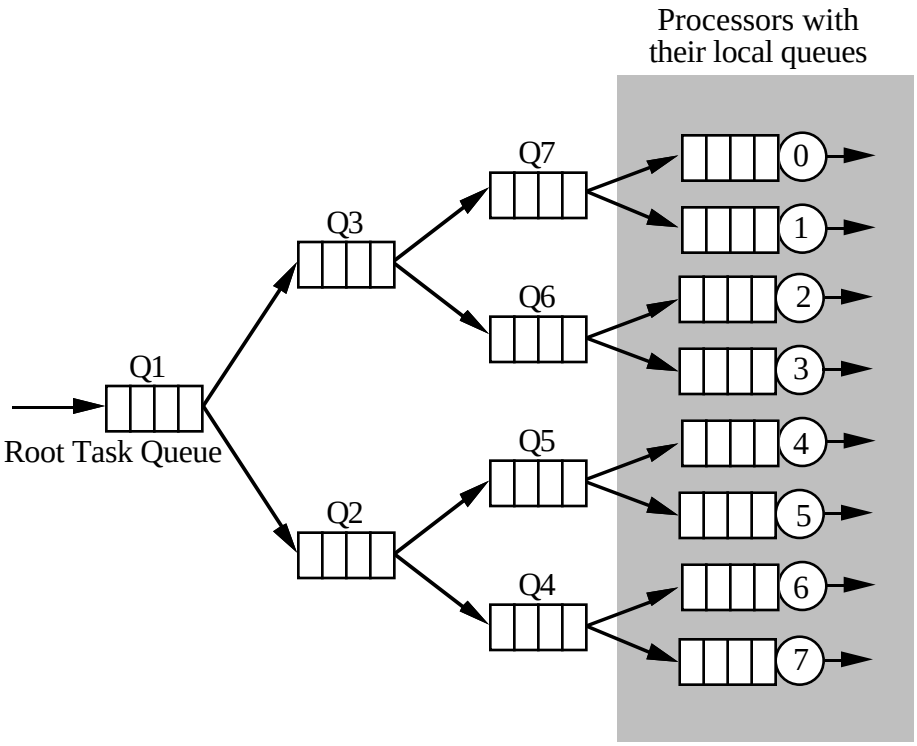
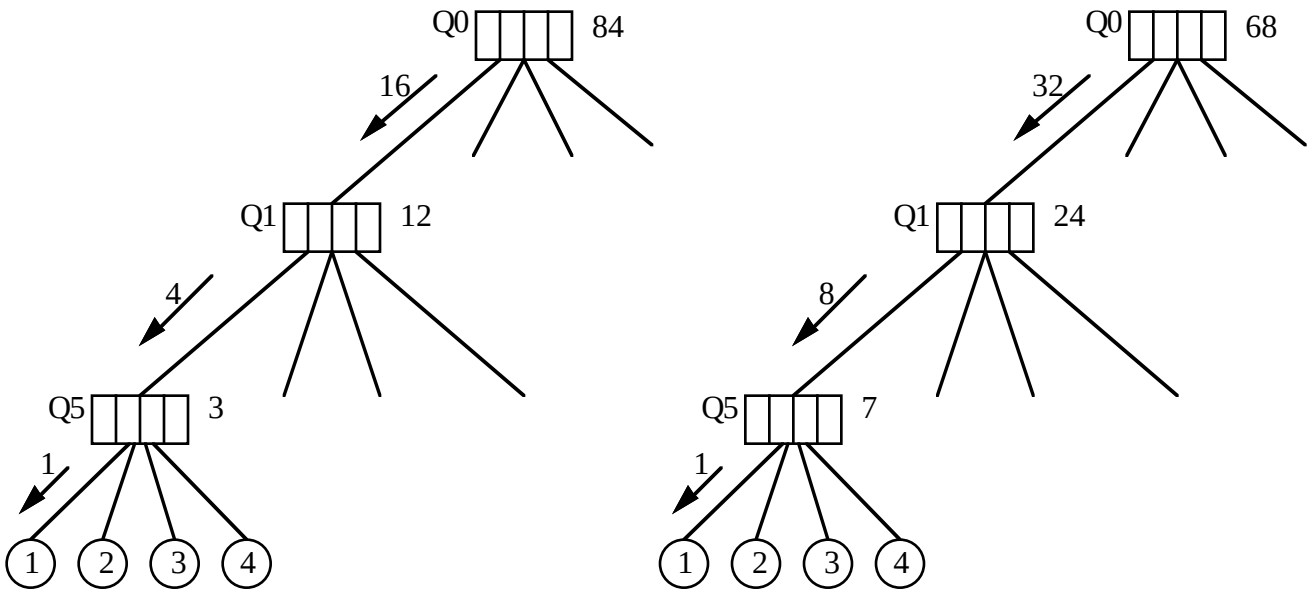



**Figure 8** An example hierarchical organization for 8 processors with a branching factor of 2




**(a)** $Tr = 1$ **(b)** $Tr = 2$

**Figure 9** Task transfer process in the hierarchical organization for 64 processors with a branching factor of 4

$$Tr = \frac{\text{number of tasks moved one level down the tree}}{\text{number of processors below the child task queue}}$$

The task transfer process is illustrated in Figure 9 for a system with 64 processors. The hierarchical structure is assumed to have a branching factor of 4. Figure 9a assumes a transfer factor *Tr* of 1. When processor 1 is looking for work (assuming that both Q5 and Q1 are empty and Q0 has 100 tasks) 16 tasks are transferred from the root queue to Q1 because *Tr* is 1. Note that this processor is moving tasks in the hierarchy on behalf of the other processors in its group. This is similar in spirit to the cooperative policy described before. Of the 16 tasks moved to Q1, 4 tasks are transferred to Q5 leaving 12 tasks at Q1. Finally, processor 1 schedules a task by moving one task into its local queue. You can see that by leaving tasks at each intermediate task queue, we are decreasing the frequency of access to the task queues at higher levels of the tree. This is required to eliminate contention as there are fewer and fewer task queues as we move up the tree. Figure 9b shows the task transfer process when *Tr* is 2 instead of 1.

A key property of the hierarchical organization is that the frequency of access to any task queue is the same independent of where it is located in the hierarchy. We will illustrate this by means of an example. Let us look at the state of the queues shown in Figure 9a. After self-scheduling a task by processor 1, Q5 can handle up to three more accesses without generating an access request to Q1. Thus, Q1 receives a queue access for every 4, which is the branching factor for this example, queue accesses to Q5. In other words, as far as Q1 is concerned, it is just serving the four task queue directly below it. This argument can be recursively applied to see that every node has the same frequency of access and there are no "hot-spot" queues in the hierarchy.

The hierarchical organization shares the twin objectives of the guided self-scheduling (GSS) strategy - reducing access contention to loop indexes and load balancing - proposed by Polychronopoulos and Kuck [Pol87] for the loop scheduling problem. The GSS strategy is shown to achieve a very good (and often optimal) load balancing for the loop scheduling problem. Nelson and Squillante [Nel90] give the processor scheduling version of the GSS and show that there are better schemes (such as their "cooperative policy"). It has been shown that the performance of the hierarchical organization is similar to that of the centralized organization (thus showing that the hierarchical organization achieves almost perfect load sharing under a variety of workload conditions).

It can also be seen that the set of task queues that form the tree can all be distributed to different memory modules so that concurrent access can be permitted (provided the interconnection network allows the particular permutation). For a system with *P* processors, the maximum number of task queues are required when the branching factor *B* is 2 and this number is equal to $(P-1)$. Since there are *P* memory modules these $(P-1)$ queues can all be distributed without causing any memory or communication contention problems.

The hierarchical organization nicely incorporates the following desirable features that we have seen are useful in reducing contention for the centralized queue.

- In each access to a task queue, a set of tasks is removed rather than a single task (as in the autonomous policy).
- Tasks are moved from a parent task queue to a child queue in a cooperative manner on behalf of the processors in its group (as in the cooperative policy).
- As in the centralized organization, per-processor local run queues are used to reduce contention so that more than one task can be moved to processor local queues (as in autonomous and cooperative policies). Although moving more than one task into the processor local queues was not necessary for the FCFS scheduling policy, it is required to implement the preemptive round robin policy [Che92].
- Multiple queues are used to reduce the contention problem by generalizing the scheme suggested in [Ni89].

The hierarchical organization, as described, does not incorporate the scheme suggested to handle load imbalance in the distributed organization i.e., an idle processor probing other processor queues for tasks. In the hierarchical organization, this scheme is useful when the task queues on path from a leaf node to the root node are all empty. In this case, the idle processor can probe other task queues in the hierarchy. However, performance analysis presented in [Dan95] indicates incorporating such a scheme is not really necessary to improve performance of the hierarchical organization.

### 4.3. PERFORMANCE

A potential problem with the hierarchical organization is that the number of queue accesses required to schedule a task is higher than that required for the centralized and distributed organizations. In both the centralized and distributed organizations, only one task queue access is required to schedule a task.

The number of queue accesses required to schedule a task in the hierarchical organization is a function of the branching factor $B$ and the transfer factor $Tr$. Assuming a complete tree with a branching factor $B$, it has been shown in [Dan95] that the average number of queue accesses per scheduled task is given by

$$1 + \frac{P - B}{P * Tr * (B-1)} \qquad (2)$$

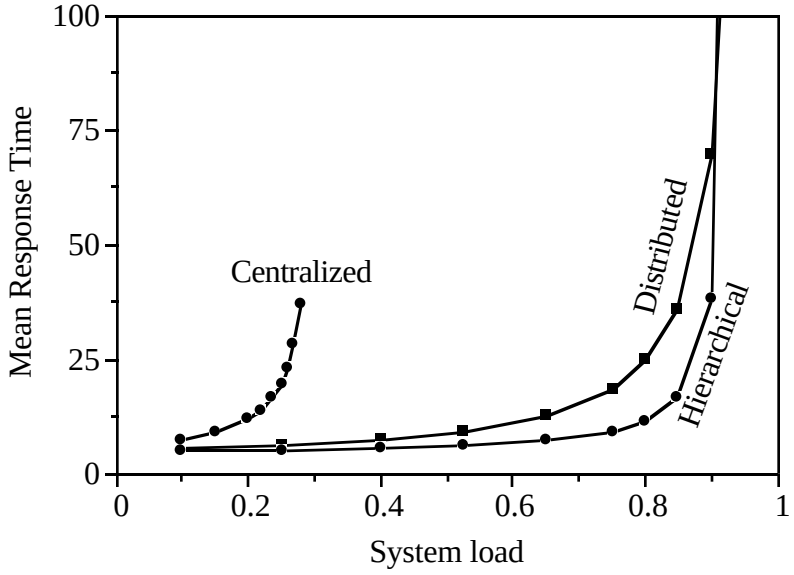
When $P \gg B$ and $B \gg 1$, this reduces to $1 + \frac{1}{Tr * B}$. Table 2 shows some sample values for various branching factor $B$ values, transfer factor $Tr$ values, and system sizes $P$. The results in Table 2 show that for higher values of $B$ and $Tr$, the average number of queue accesses is very close to one. For example, when $Tr$ = 1 and $B$ = 16, the increase in the number of queue accesses is less than 7% even for a system with 1024 processors.

The performance of the hierarchical organization has been reported in detail in [Dan95]. Here, we provide a brief comparison of the three organizations. Figure 10a shows the performance of the three task queue organizations for a system with $P = 64$ processors. The average number of tasks per job is assumed to be 64. The queue access time is set at $f$ = 4%. For the hierarchical organization, a branching factor $B$ of 4 and a transfer factor $Tr$ of 1 are used.

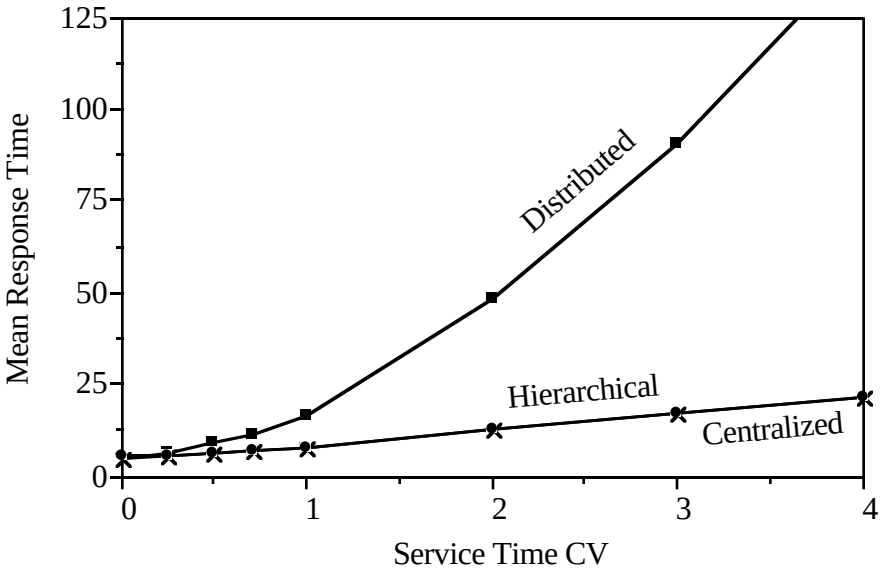
Figure 10a shows the average response time of the three organizations as a function of system load. The centralized organization provides very poor performance due to the queue access contention. The distributed organization suffers in performance due to load imbalance. The hierarchical organization provides the best performance among the three queue organizations for all system loads.

**Table 2** Average number of queue accesses required to schedule a task in the hierarchical organization (from Eq. (2))

| | $P$ = 64 | | | | $P$ = 256 | | | | $P$ = 1024 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $B$ = 2 | $B$ = 4 | $B$ = 8 | $B$ = 16 | $B$ = 2 | $B$ = 4 | $B$ = 8 | $B$ = 16 | $B$ = 2 | $B$ = 4 | $B$ = 8 | $B$ = 16 |
| $Tr$=0.5 | 2.938 | 1.62 | 1.25 | 1.10 | 2.984 | 1.656 | 1.277 | 1.125 | 2.996 | 1.664 | 1.283 | 1.131 |
| $Tr$=1 | 1.969 | 1.31 | 1.125 | 1.05 | 1.992 | 1.328 | 1.138 | 1.063 | 1.998 | 1.332 | 1.142 | 1.066 |
| $Tr$=2 | 1.484 | 1.15 | 1.063 | 1.03 | 1.496 | 1.164 | 1.07 | 1.031 | 1.499 | 1.166 | 1.071 | 1.033 |




**(a)**




**(b)**

**Figure 10** Performance comparison of the three queue organizations

The question that naturally arises at this point is: How good is the performance of the hierarchical organization relative to the centralized one if there were no queue contention? We will answer this question by assuming that there is no run queue access contention (we model this by setting $f$ to 0%). In this case, we know that the centralized organization provides the best performance. Figure 10b presents the sensitivity of the three organizations when the system load is fixed at 75% and $f$ = 0% while keeping the other parameters the same as before. The results presented here show that the hierarchical organization inherits the load sharing property of the centralized organization. The hierarchical organization eliminates the access contention like the distributed organization and yet provides performance similar to that of the centralized organization without requiring any additional information on either system state or task characteristics. Thus, when task queue access contention is taken into account, the hierarchical organization provides the best performance among the three organizations.

### 4.4. PERFORMANCE SENSITIVITY OF THE HIERARCHICAL ORGANIZATION

The branching factor *B* and the transfer factor *Tr* are the two key design issues in the hierarchical organization. To minimize the number of queue accesses needed to schedule a task higher *B* and *Tr* values should be used (see Eq. (2)). For larger values of *B* and *Tr* the number of queue accesses required would be very close to 1 as shown in Table 2. Larger *B* values also improve the system performance by improving load sharing. On the other hand, larger *Tr* values decrease the scope of load sharing. Thus, for better load-sharing, we should use a higher value for *B* and a smaller value for *Tr*. However, the access contention for the ready queue imposes an upper limit on the branching factor that can be used. The results presented in [Dan95] show that there is a wide range of values for which the performance of the hierarchical organization is relatively insensitive.

In our discussion so far, the transfer factor *Tr* is fixed statically. As in the case of the autonomous policy, this may lead to load imbalance. For example, in Figure 9a, assume that the whole tree of queues is empty except for the root queue, which has 16 tasks. When Processor 1 goes up to the root node, it removes all 16 tasks from the root queue as we are using a transfer factor of 1 in this example. This leaves 75% of the system processors idle until a new job arrives into the system. Therefore, instead of using a fixed value for *Tr*, we can dynamically vary this as a function of system load. Results in [Dan95] indicate that the improvement in performance with such a dynamic scheme is not substantial (less than 10%).

We have assumed a non-preemptive FCFS task scheduling policy. However, it is well-known that preemptive policies like round robin provide a better performance, particularly when there is a high variance in service times. In the hierarchical organization, round robin policy can be implemented at the local queue level. Implementing at the global (i.e., root) queue level causes the same contention problem as in the centralized organization. The results in [Che92] show that, for a wide range of parameter values of interest, the FCFS task scheduling policy performs as well as or better than the round robin policies.

## 5. SUMMARY

In parallel systems, it is important to avoid resource contention. This is particularly so for large multiprocessor systems. Resource contention can be for hardware resources such as memory or communication network or for software resources such as a critical data structure. Our focus here has been on reducing access contention for the system run queue.

There are two basic run queue organizations: centralized or distributed. In the centralized organization, a global run queue is shared by all processors in the system. In the distributed organization, each processor maintains its own private run queue. The centralized organization gives the best performance by providing perfect load sharing. However, for large systems, this run queue organization is not suitable as the global queue can become a bottleneck. The distributed organization is scalable to large systems but it suffers from the load imbalance problem.

We have discussed several ways to reduce the queue access contention in the centralized organization and load imbalance in the distributed organization. None of these techniques is completely satisfactory and they introduce problems of their own.

We have introduced a hierarchical organization to incorporate the desirable features of the centralized and distributed organizations while minimizing the disadvantages associated with them. We have shown that the hierarchical organization performs best among the three run queue organizations considered in this paper.

We should note that, to implement the hierarchical run queue organization, it is not necessary for the underlying architecture to be hierarchical. However, when the system architecture is based on a hierarchy, there is a natural mapping that may fix the branching factor of the hierarchical structure. Currently, there is a trend toward building large-scale multiprocessor systems that are based on some sort of hierarchy. Some example systems include the Stanford DASH, Illinois CEDAR, Toronto's Hector. Hierarchical organization is also useful in devising better scheduling policies for such systems [Zho91, Aya95]. We strongly feel that the hierarchical organization presented here is a useful mechanism for organizing the run queue for such large-scale shared memory systems.

## ACKNOWLEDGEMENTS

The description of the hierarchical organization presented here is based on [Dan95], which represents a joint work done with Philip Cheng. I am grateful to the financial support provided by NSERC and Carleton University in carrying out this research.

## REFERENCES

[And89]

T. E. Anderson, E. D. Lazowska, and H. M. Levy, "The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors," *IEEE Trans. Computers*, Vol. C-38, No. 12, December 1989, pp. 1631-1644.

[Aya95]

S. Ayachi, *A Hierarchical Processor Scheduling Policy for NUMA Systems*, MCS thesis, School of Computer Science, Carleton University, Ottawa, 1995.

[Che92]

S. P. Cheng and S. P. Dandamudi, "Scheduling in Parallel Systems with a Hierarchical Organization of Tasks," *Proc. ACM Int. Conf. on Supercomputing*, Washington, D.C., July 1992, pp. 377-386.

[Dan91]

S. P. Dandamudi, "A Comparison of Task Scheduling Strategies for Multiprocessor Systems," *IEEE Symp. Parallel and Distributed Processing*, Dallas, Texas, December 1991, pp. 423-426.

[Dan95]

S. P. Dandamudi and S. P. Cheng, "A Hierarchical Task Queue Organization for Shared-Memory Multiprocessor Systems," *IEEE Trans. Parallel and Distributed Systems*, Vol. 6, No. 1, January 1995, pp. 1-16.

[Fei90]

D. G. Feitelson and L. Rudolph, "Distributed Hierarchical Control for Parallel Processing," *Computer*, May 1990, pp. 65-77.

[Ni89]

L. M. Ni and C. E. Wu, "Design Tradeoffs for Process Scheduling in Shared Memory Multiprocessor Systems," *IEEE Trans. Software Engng*, Vol. SE-15, No. 3, March 1989, pp. 327-334.

[Nel90 ]

R. Nelson and M. Squillante, "Analysis of Contention in Multiprocessor Scheduling," *Performance 90 - Proc. Int. Symp. Computer System Modelling, Measurement and Evaluation*, September 1990, pp. 391-405.

[Pol87]

C. D. Polychronopoulos and D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers," *IEEE Trans. Computers*, Vol. C-36, No. 12, December 1987, pp. 1425-1439.

[Shi92]

N. G. Shivaratri, P. Krueger, and M. Singhal, “Load Distributing for Locally Distributed Systems,” *IEEE Computer*, December 1992, pp. 33-44.

[Ste88]

P. Stenstrom, "Reducing Contention in Shared-Memory Multiprocessors," *IEEE Computer*, November 1988, pp. 26-37.

[Zho91]

S. Zhou and T. Brecht, "Processor Pool-Based Scheduling for Large-Scale NUMA Multiprocessors," *Proc. of ACM Sigmetrics Conf.*, 1991, pp. 133-142.