Feature

Reducing Run Queue Contention in Shared Memory Multiprocessors

No single method for mitigating the performance problems of centralized and distributed run queues is entirely successful. A hierarchical run queue succeeds by borrowing the best features of both.

Sivarama P. Dandamudi
Carleton University

Performance of parallel processing systems, especially large systems, is sensitive to various types of overhead and contention. Performance consequences may be serious when contention occurs for hardware resources such as memory or the interconnection network. Contention can also occur for software resources such as critical data structures maintained by either system or application software.

A run queue is one such critical data structure that can affect overall system performance. There are two basic types of run queues, centralized and distributed. Both present performance problems. There are also several techniques to mitigate their drawbacks, but none is completely satisfactory. Instead, I propose a different run queue organization, a hierarchical organization that inherits the best features of the centralized and the distributed queue organizations while avoiding their pitfalls. Thus, the hierarchical organization is suitable for building large-scale multiprocessor systems.

SHARED MEMORY MULTIPROCESSORS

Shared memory multiprocessors give programmers a single address space much like in a traditional uniprocessor system. Processors communicate through shared-memory variables. Often simply called multiprocessors, the term I use here, shared memory multiprocessors are evolving toward general-purpose multi-user systems.[1]

Multiprocessors provide either uniform memory access or nonuniform memory access. In UMA multiprocessors, the cost of accessing a memory location is the same for any processor in the system. In NUMA multiprocessors, memory access cost varies. Generally, a UMA architecture is good for systems with tens of processors, while a NUMA architecture lets us build systems with hundreds of processors.

Uniform memory access

Figure 1 illustrates a UMA multiprocessor, in which the shared memory is global to all processors. An interconnection network facilitates communication between the processors and the 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, limited-bandwidth bus as an interconnection network severely restricts system scalability. Furthermore, this type of interconnection network allows only one processor at a time to communicate with the memory, leading to performance degradation. To avoid this problem, the shared memory is divided into memory modules, as Figure 1 shows.

Typically, UMA multiprocessors allow concurrent access to all memory modules, as long as there are processors requesting access to memory modules and no two processors wish to access the same memory module. Concurrent access can be provided via a crossbar interconnection network. However, crossbar connections are too expensive for systems with hundreds of processors. Instead, many large multiprocessor systems use a multistage interconnection network. For example, the New York University Ultracomputer uses an Omega switching network.

When a multistage interconnection network is used, the number of memory modules usually equals the number of processors. If there are fewer memory modules than processors, memory contention may occur when all the processors try to access the shared memory.[2] It is sufficient to have one memory module per processor, assuming the interconnection network can connect the processors to the distinct memory modules, since each memory module can typically service one access request at a time.

0018-9162/97/$10.00 © 1997 IEEE

But even with this concurrent access, memory contention can still occur if several processors concurrently request access to the same memory module. Similarly, communication contention can occur if several processors contend for a common link in the interconnection network. To reduce both memory contention and network contention, processors usually maintain some local memory that can be directly accessed by the associated processor without using the interconnection network. Per Stenstrom[2] discusses memory design techniques that reduce contention in shared memory systems.

Nonuniform memory access

Figure 2 shows a NUMA multiprocessor. Here the shared memory is physically distributed among the processors. In this architecture, memory module M_i is local to processor P_i, where i can be in the range 0 through $N-1$. Thus processor P_i can access a memory location in module M_i without going through the interconnection network. To access a remote memory location (one mapped to a memory module other than M_i), the processor must use the interconnection network, thereby increasing the delay. The Stanford Dash, the Illinois Cedar, Toronto's Hector, and the BBN Butterfly system are NUMA multiprocessors.

NUMA systems can reduce interconnection network traffic by exploiting locality; for example, the local memory can store frequently accessed parts of code and data. Therefore, NUMA architecture is suitable for large systems and does not put undue demand on the interconnection network.

Run queue problems

Even with the best system and memory design, memory and communication contention are still possible. The cause can be a critical data structure maintained in the shared memory. For example, if the system software in a large system maintains a single, or centralized, run queue of jobs waiting to be scheduled, this can become a bottleneck, leading to both memory and communication contention—and ultimately to serious performance degradation. Thus, we want to reduce contention for the run queue to facilitate processor scheduling. Figure 3a on the next page shows this centralized run queue, with a single global queue of ready tasks accessible to all processors in the system. Run queue access should be allowed on a mutually exclusive access basis. Since each access takes a finite amount of time (called queue access time), the centralized organization can lead to access contention for a sufficiently large number of processors.

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, a centralized run queue is clearly not suitable for large systems. To avoid run queue contention in large systems, each processor can maintain a local run queue. Figure 3b shows a distributed organization, in which private run queues are associated with the processors. A task placement policy determines which queue arriving tasks are placed in. A simple task placement policy assigns arriving tasks to a queue randomly. A better placement policy assigns 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 task queue ($i + 1$ mod N), where N is the number of processors. This article assumes that the cyclic placement policy is used. This distributed run queue organization avoids contention but creates a load imbalance: Some processors may be idle while others have work waiting.

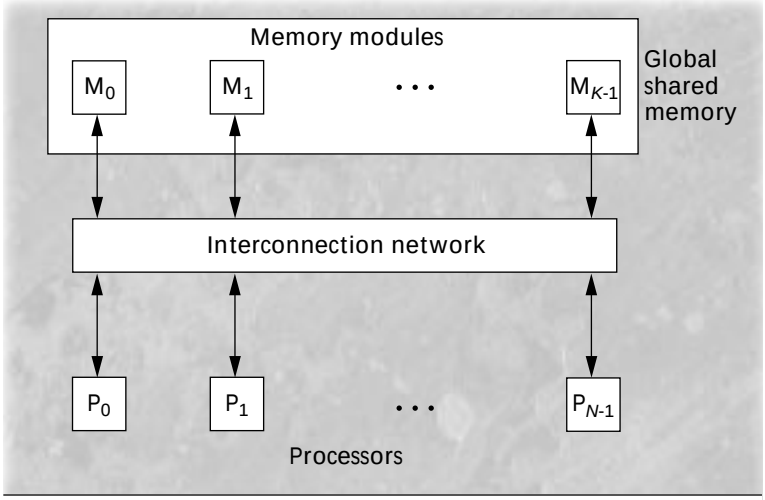

Figure 1. A uniform-memory-access multiprocessor system.

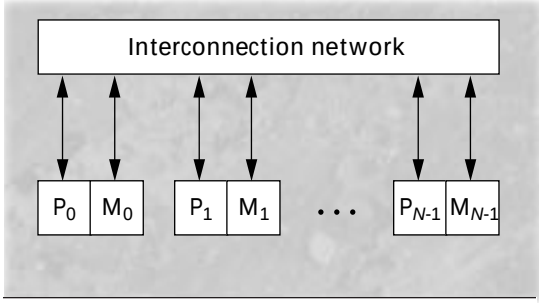

Figure 2. A nonuniform-memory-access multiprocessor system.

IMPROVING RUN QUEUE PERFORMANCE

To improve performance in the centralized organization, we must devise ways to eliminate or minimize run queue contention. For the distributed organization, we need strategies to eliminate or minimize load imbalance. There are several useful techniques, though none is acceptable by itself.

To improve performance we must eliminate or minimize run queue contention in the centralized organization and load imbalance in the distributed organization.

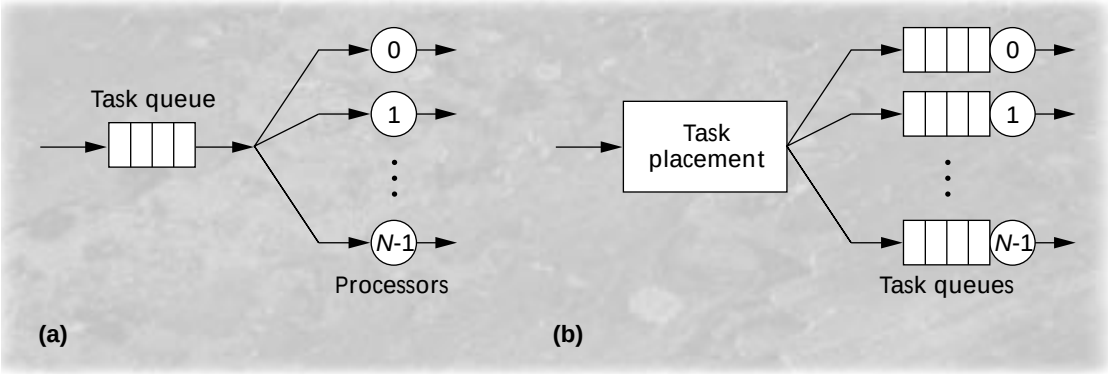

Figure 3. Two basic run queue organizations: (a) centralized and (b) distributed with cyclic task placement.

Techniques for centralized queues

To minimize access contention in the centralized organization, we must find ways to decrease the frequency of run queue access. Randolph Nelson and Mark Squillante discuss two ways to do this while maintaining a single global run queue.[3] The principle behind both methods is to remove larger chunks of work with each access. Processors must maintain local queues, as in the distributed organization.

In their first method, called the autonomous policy, every time a processor accesses the run queue it removes a *set* of tasks from the queue instead of a single task. Thus processors need fewer trips to the run queue. Because removing a constant number of tasks can result in load imbalance and lead to performance problems, the scheduler must control the set size. The optimal set size increases with system load.[3] In addition, a large variance in task service times can cause performance to deteriorate.

Nelson and Squillante's second method, called the cooperative policy, can also reduce queue access contention. Here, a processor accessing the run queue schedules tasks not only for itself but also for other processors, in a cooperative manner. The scheduler dynamically adjusts the number of tasks scheduled on each processor, according to system utilization. The cooperative policy moves tasks from the central queue to processors' local queues in a "join the shortest queue" fashion and thus provides better load balance than the autonomous policy. Nelson and Squillante's analysis shows that the cooperative policy performs better than both the autonomous policy and the distributed organization.[3] However, the cooperative policy is difficult to implement, particularly for larger systems, because the scheduler must maintain state information on processors' local queues to move tasks from the global queue to local queues on a shortest queue basis.

Another way to reduce run queue contention is to use multiple run queues. Lionel Ni and Ching-Farn Wu[4] proposed a method that maintains a number of run queues that is less than the number of processors. The system's N processors are partitioned into m groups, and each group is served by a dedicated run queue. Arriving tasks are placed randomly in one of the m queues. The number of queues represents a trade-off between load sharing and access contention. A small number of queues increases load sharing but also increases access contention. A large number of queues has the opposite effect. Obviously, if the number of queues equals the number of processors, this organization would correspond to the distributed organization.

Ni and Wu presented a method to determine the optimum value for m. However, their solution is not satisfactory, because the optimum value for m is system-load dependent. Their solution can also cause load imbalance, the extent of which depends on the value of m. However, the same techniques that are useful in the context of distributed queue organization can reduce this load imbalance.

Techniques for distributed queues

To improve performance in the distributed organization, we must reduce load imbalance. One way to do this is to distribute tasks more intelligently than the random or cyclic placement policy. Another way is to have an idle processor get work from other processors when a load imbalance occurs.

Studies have shown that the cyclic placement policy performs substantially better than the random placement policy in the distributed queue organization.[5] We can improve the performance further by using an *adaptive* placement policy. An example is the shortest queue placement policy, in which an arriving task is routed to the shortest queue. Performance improvements come at the cost of collecting queue length information on the processors' local queues. Even this placement policy, however, does not eliminate load imbalance for large variances in task service times, so performance may still suffer.

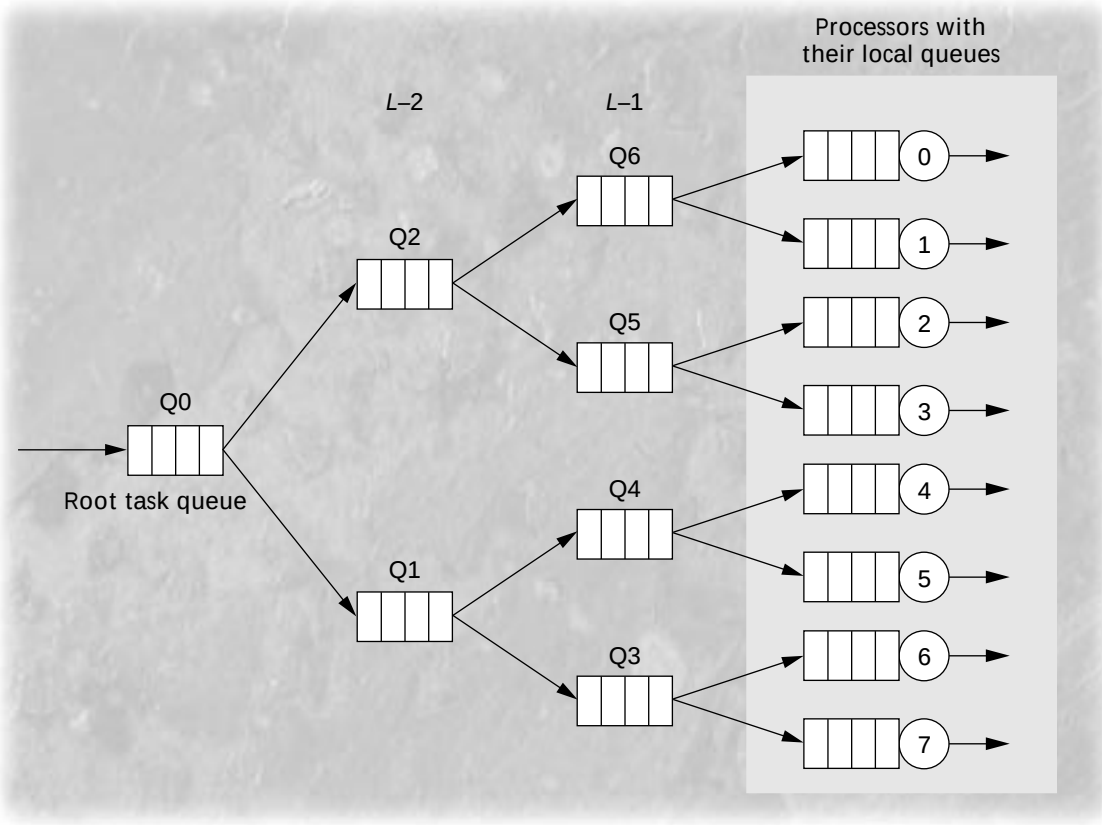

Figure 4. Hierarchical run queue organization for N = 8 processors with a branching factor B = 2.

We can reduce the adverse effect of high variance in task service times if we know these times (or at least have a good estimate of them) in advance. Then we can route each arriving task to the queue with the shortest waiting time (the sum of the service times for the tasks waiting to be scheduled in a queue). When we can presume the exact task service times, this shortest waiting time placement policy provides performance that nearly matches that of the centralized organization. In practice, however, it is difficult to estimate task service times.

Both of these adaptive placement policies provide substantial performance improvements, but there are implementation problems associated with collecting queue state information.[5] Another strategy for coping with load imbalance is to use the simple cyclic task placement policy and handle load imbalance when it occurs. In this strategy, if a processor finds its local queue empty, it probes the local queues of other processors for tasks.[3,6] (This is similar to the receiver-initiated load-sharing strategy used in distributed systems.[7]) For example, an idle processor can probe the state of a randomly selected processor's local task queue. If the queue is empty, it randomly selects another processor and repeats the process for a maximum of some predefined probe limit value or until a task arrives at its local queue. Nelson and Squillante show that this probing technique reduces load imbalance considerably but does not eliminate it. Thus, despite these possible improvements, load imbalance continues to be a problem with the distributed organization.

HIERARCHICAL RUN QUEUE ORGANIZATION

The run queue options discussed above involve certain design trade-offs. The ideal organization would allow good load sharing (as in the centralized organization) and also provide distributed task queues to eliminate the bottleneck problem (as in the distributed organization). Because the hierarchical organization incorporates the performance improvements suggested for the basic centralized organization while following the main concept of distributed organization, it achieves these objectives.

Description and operation

In the hierarchical organization, a set of task queues is organized as a tree, and the processors with their local queues are attached to the bottom level of the tree as leaf nodes. Figure 4 shows an example hierarchical organization for eight processors with a tree branching factor of two. Each task queue in the hierarchy can be viewed as a centralized task queue serving only the tree nodes (which can be either task queues or processor local queues) directly below it. Moreover, we can view the centralized organization as a special case of this structure, with just one level (the root queue) having a branching factor equal to the number of processors in the system.

The hierarchy idea isn't new. Thomas Anderson and colleagues used a two-level hierarchy consisting of a global pool of ready tasks and processors with local queues.[6] Dror Feitelson and Larry Rudolph also proposed a distributed hierarchical control for parallel processing, but they assumed that specialized

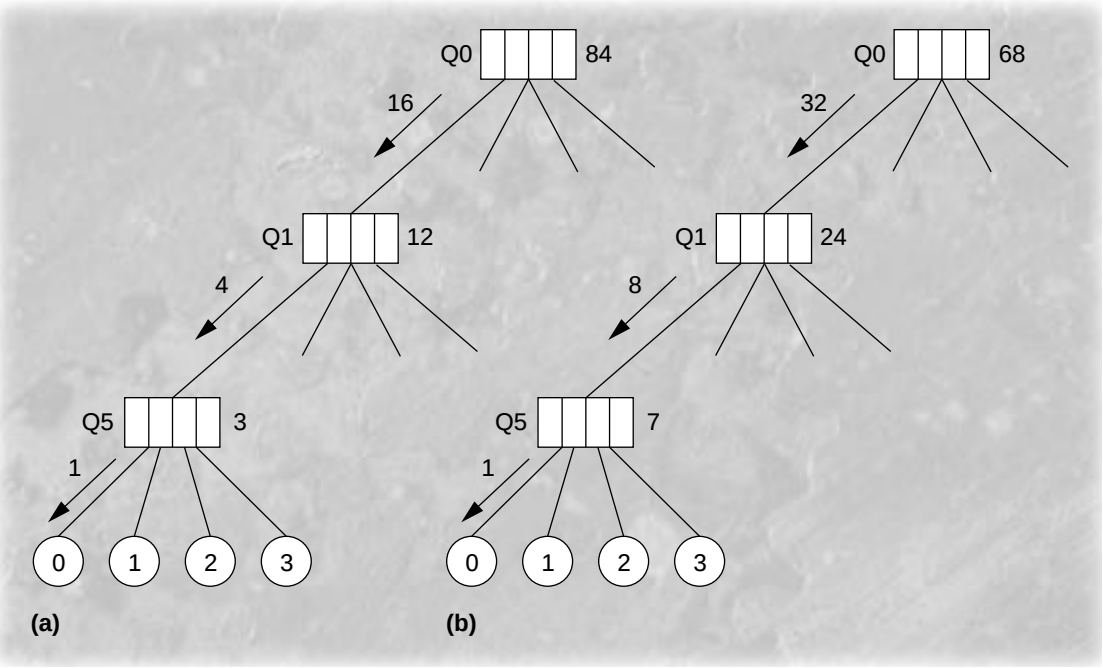

Figure 5. Task transfer process in the hierarchical organization for N = 64 processors with a branching factor B = 4: (a) Tr = 1, (b) Tr = 2.

hardware is available for scheduling tasks.[1]

In the hierarchical organization, all incoming tasks are added to the root task queue. Let L be the leaf-node, or processor, level in the tree. A processor looking for work first checks its associated task queue at level $L - 1$. If that queue is empty it checks the parent node at the next level, $L - 2$. This process is repeated up the tree until the processor finds a task to be scheduled (unless the root queue is empty). When a task queue is accessed, a set of tasks is moved one level down the tree, as in the autonomous policy. This reduces access contention at higher levels. The size of the set decreases progressively as you go down the tree (to account for the increasing number of task queues per level). At the bottom of the tree, set size is reduced to just one task that is scheduled on the associated processor. A parameter called the transfer factor, *Tr*, indicates the number of tasks transferred from a parent queue to a child queue. *Tr* is defined as

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

Figure 5 illustrates the task transfer process for a system with 64 processors. The hierarchical structure is assumed to have a branching factor of 4. Figure 5a assumes a transfer factor of 1. When processor 0 looks for work (assuming that both Q5 and Q1 are empty and Q0 has 100 tasks), 16 tasks are transferred from the root queue (Q0) 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 earlier. Of the 16 tasks moved to Q1, four are transferred to Q5, leaving 12 tasks at Q1. Finally, processor 0 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 necessary to eliminate contention, because there are fewer and fewer task queues as we move up the tree. Figure 5b shows the task transfer process when *Tr* is 2 instead of 1.

A key property of the hierarchical organization is that the access frequency to any task queue is the same regardless of its location in the hierarchy. For example, let's look at the state of the queues shown in Figure 5a. After processor 0 schedules a task, Q5 can handle up to three more accesses without generating an access request to Q1. Thus, Q1 receives a queue access for every four queue accesses to Q5 (a branching factor of 4). In other words, as far as Q1 is concerned, it is serving only the four task queues directly below it. Applying this argument recursively shows that every queue has the same access frequency and that no queue in the hierarchy is subject to an extraordinary access demand.

The hierarchical organization's performance is similar to that of the centralized organization, which shows that the hierarchical organization achieves almost perfect load sharing under a variety of workload conditions.

The set of task queues that form the tree can all be distributed to different memory modules, thereby permitting concurrent access (provided the interconnection network allows the particular set of connections). For a system with N processors, the maximum number of task queues, $N - 1$, is required when the branching factor is two. Since there are N memory modules, these $N - 1$ queues can all be distributed without causing memory or communication contention.

The hierarchical organization successfully incorporates the following desirable features found 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).

Table 1. Average number of queue accesses required to schedule a task in the hierarchical organization. *N* = processors, *B* = branching factor, *Tr* = transfer factor.

	N = 64				*N* = 256				*N* = 1,024			
	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	1.250	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	1.070	1.031	1.499	1.166	1.071	1.033

- Tasks are moved from a parent task queue to a child queue in a cooperative manner on behalf of the processors in a group (as in the cooperative policy).
- More than one task can be moved to processors' local queues (as in the autonomous and cooperative policies), thereby reducing contention. Although moving multiple tasks into local queues is not always necessary, it *is* necessary for one particular task scheduling policy discussed elsewhere.[8]
- Multiple queues reduce the contention problem by generalizing the scheme suggested by Ni and Wu.[4]

As described, the hierarchical organization does not incorporate the scheme suggested for handling load imbalance in the distributed organization, that is, an idle processor probing other processor queues for tasks. This scheme is useful in a hierarchical organization when the task queues on a path from a leaf node to the root node are all empty. In such a case, the idle processor can probe other task queues in the hierarchy. However, the performance analysis that Philip Cheng and I conducted[9] indicates that it is not really necessary to incorporate such a scheme to improve performance of the hierarchical organization.

Performance

A potential problem with the hierarchical organization is that more queue accesses are required to schedule a task than in the centralized and distributed organizations, which require only one task queue access 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*, we have shown[9] that the average number of queue accesses per scheduled task is

$$1 + \frac{N - B}{N * Tr * (B - 1)}$$

When $N \gg B$ and $B \gg 1$, this reduces to $1 + (1/(Tr * B))$.

Table 1 shows some sample values for various branching factors, transfer factors, and system sizes. 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 percent, even for a system with 1,024 processors.

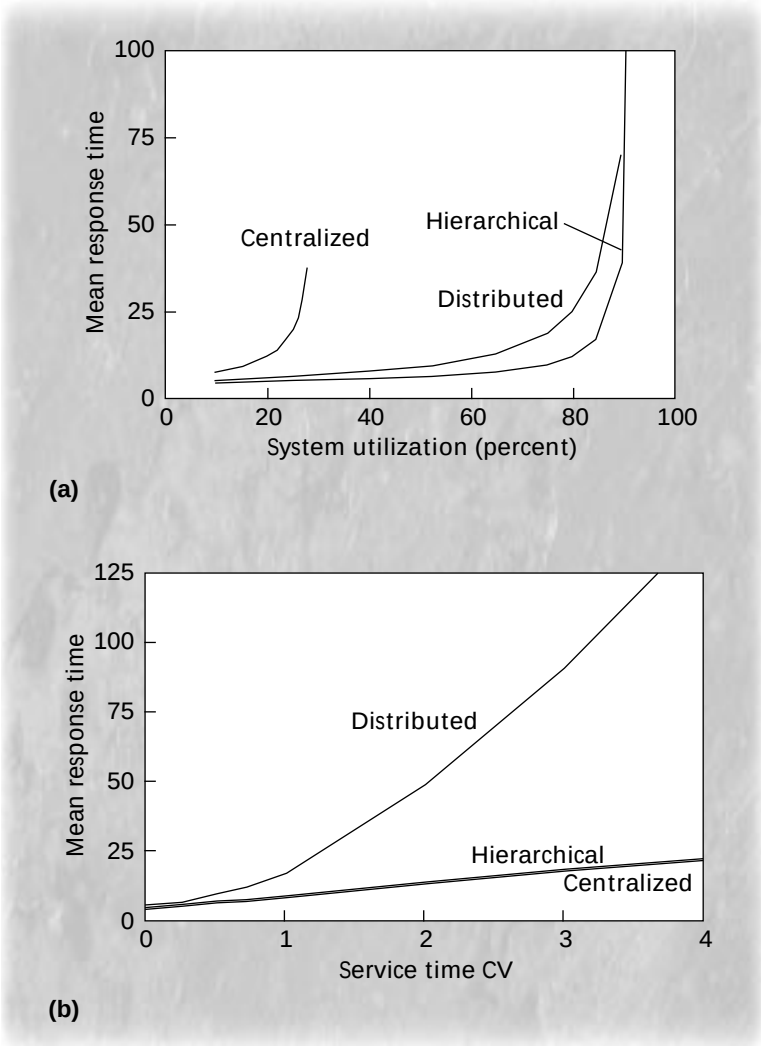

Figure 6. Performance comparison of the three run queue organizations: (a) with access contention in the centralized organization, (b) without access contention in the centralized organization.

Figure 6a compares the performance of the three run queue organizations for a system with 64 processors. (We present more extensive performance details elsewhere.[9]) For the hierarchical organization, a branching factor of 4 and a transfer factor of 1 are used. All other parameters, including queue access time, are the same for all three organizations.

For these results, we assume that a parallel job's tasks do not communicate with each other and can be executed to completion. However, since a job's execution is considered complete only when all of its tasks are done, a job's tasks all participate in a synchronization phase at the end of task execution. Cheng and

The hierarchical organization inherits the load-sharing property of the centralized organization and also eliminates access contention.

I used one time unit as the average task service time, and that is the time unit used in Figure 6.

Figure 6a shows the average response time of the three organizations as a function of system utilization. The centralized organization provides very poor performance because of queue access contention. The distributed organization's performance suffers because of load imbalance. The hierarchical organization provides the best performance for all levels of system utilization.

How good would the performance of the hierarchical organization be, relative to the centralized one, if there were no run queue contention? Let's assume this condition even though we know that the centralized organization would provide the best performance. Figure 6b shows the performance of the three organizations when system utilization is fixed at 75 percent and queue access time is negligible; the other parameters are the same as before. The *x*-axis represents the variance in task service times. We express this variance as the ratio of the standard deviation to mean, which is called the *coefficient of variation* (CV). Thus, the higher the CV, the larger the variance in task service times.

The results show that the hierarchical organization inherits the load-sharing property of the centralized organization. Furthermore, like the distributed organization, the hierarchical organization eliminates access contention. Thus, its performance matches that of the centralized organization without requiring additional information on either system state or task characteristics.

Performance sensitivity

The branching factor and the transfer factor are the key design issues in the hierarchical organization. Minimizing the number of queue accesses needed to schedule a task requires larger *B* and *Tr* values. With these larger values, the number of queue accesses required to schedule a task is very close to 1, as shown in Table 1. Larger *B* values also improve 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 larger value for *B* and a smaller value for *Tr*. However, access contention for the run queue imposes an upper limit on the branching factor. The detailed results[9] 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 is fixed statically. As with the autonomous policy, this may lead to load imbalance. Let's assume that in Figure 5a the whole tree of queues is empty except for the root queue, which has 16 tasks. Because the transfer factor is 1 in this example, when processor 0 goes up to the root node, it removes all 16 tasks from the root queue. This leaves 75 percent of the processors idle until a new job enters the system. Therefore, instead of using a fixed value for *Tr*, we may wish to dynamically vary it as a function of system utilization. The detailed results[9] indicate that such a dynamic scheme improves performance by less than 10 percent.

When presenting the results in Figure 6, I assumed that a job's tasks do not communicate with each other, so that tasks can be executed independently. What happens if the tasks cannot be executed independently because they must frequently synchronize their activities? Cheng and I found[10] that in this case, under certain conditions, the distributed run queue organization performs better than the hierarchical organization. Nevertheless, the hierarchical organization is useful because the distributed organization is embedded within it. In other words, the hierarchy of queues can simply be ignored for those applications that do not require it.

The hierarchical run queue organization can be implemented on architectures that are not 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 run queue structure. The trend is toward building large-scale multiprocessor systems that are based on some sort of hierarchy. Example systems include Dash, Cedar, and Hector. A hierarchical organization is also useful in devising better processor scheduling policies for such systems.[11,12] I strongly believe that the hierarchical run queue organization presented here will prove to be a useful mechanism when implemented in such a large-scale shared memory system. ❖

Acknowledgments

The hierarchical organization described here is based on work done with Philip Cheng. I am grateful for the financial support of Carleton University and the Natural Sciences and Engineering Research Council of Canada.

References

1. D.G. Feitelson and L. Rudolph, "Distributed Hierarchical Control for Parallel Processing," *Computer*, May 1990, pp. 65-77.
2. P. Stenstrom, "Reducing Contention in Shared-Memory Multiprocessors," *Computer*, Nov. 1988, pp. 26-37.
3. R. Nelson and M. Squillante, "Analysis of Contention in Multiprocessor Scheduling," *Proc. Int'l Symp. Computer System Modeling, Measurement and Evaluation*, North-Holland, Amsterdam, 1990, pp. 391-405.
4. L.M. Ni and C.E. Wu, "Design Tradeoffs for Process Scheduling in Shared Memory Multiprocessor Systems," *IEEE Trans. Software Eng.*, Mar. 1989, pp. 327-334.

5. S.P. Dandamudi, "A Comparison of Task Scheduling Strategies for Multiprocessor Systems," *Proc. Third IEEE Symp. Parallel and Distributed Processing*, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 423-426.
6. T.E. Anderson, E.D. Lazowska, and H.M. Levy, "The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors," *IEEE Trans. Computers*, Dec. 1989, pp. 1,631-1,644.
7. N.G. Shivaratri, P. Krueger, and M. Singhal, "Load Distributing for Locally Distributed Systems," *Computer*, Dec. 1992, pp. 33-44.
8. S.P. Cheng and S.P. Dandamudi, "Scheduling in Parallel Systems with a Hierarchical Organization of Tasks," *Proc. ACM Int'l Conf. Supercomputing*, ACM Press, New York, 1992, pp. 377-386.
9. S.P. Dandamudi and S.P. Cheng, "A Hierarchical Task Queue Organization for Shared-Memory Multiprocessor Systems," *IEEE Trans. Parallel and Distributed Systems*, Jan. 1995, pp. 1-16.
10. S.P. Dandamudi and S.P. Cheng, "Performance Impact of Run Queue Organization and Synchronization on Large-Scale NUMA Multiprocessor Systems," to be published in *J. Systems Architecture*, Apr. 1997.
11. S. Zhou and T. Brecht, "Processor Pool-Based Scheduling for Large-Scale NUMA Multiprocessors," *Proc. ACM Sigmetrics Conf.*, ACM Press, New York, 1991, pp. 133-142.
12. S. Ayachi and S.P. Dandamudi, "A Hierarchical Processor Scheduling Policy for Multiprocessor Systems," *Proc. Eighth IEEE Symp. Parallel and Distributed Processing*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 100-109.

Sivarama P. Dandamudi is an associate professor in the School of Computer Science at Carleton University in Ottawa, Ontario, Canada. His research interests include parallel and distributed systems, database systems, performance evaluation, and computer architecture. He is the author of Hierarchical Hypercube Multicomputer Interconnection Networks *(Ellis Horwood, 1991). Dandamudi received a BE in electronics engineering from the University of Mysore, India, an M.Tech in electrical engineering from the Indian Institute of Technology, Madras, and an MSc and a PhD in computer science from the University of Saskatchewan, Canada.*

Contact Dandamudi at Carleton University, School of Computer Science, 5328 Herzberg Bldg., 1125 Colonel By Drive, Ottawa, Canada, K1S 5B6; sivarama@scs.carleton.ca.