A Hierarchical Processor Scheduling Policy for Distributed-Memory Multicomputer Systems

Sivarama P. Dandamudi and Thanalapati K. Thyagaraj
Centre for Parallel and Distributed Computing
School of Computer Science, Carleton University
Ottawa, Ontario K1S 5B6, Canada
{sivarama, thyag}@scs.carleton.ca

A Hierarchical Processor Scheduling Policy for Distributed-Memory Multicomputer Systems

Sivarama P. Dandamudi and Thanalapati K. Thyagaraj
Centre for Parallel and Distributed Computing
School of Computer Science, Carleton University
Ottawa, Ontario K1S 5B6, Canada
TECHNICAL REPORT TR-97-15

ABSTRACT - Processor scheduling policies for distributed-memory systems can be divided into space-sharing or time-sharing policies. In space sharing, the set of processors in the system is partitioned and each partition is assigned for the exclusive use of a job. In time sharing policies, on the other hand, none of the processors is given exclusively to jobs; instead, several jobs share the processors (for example, in a round robin fashion). There are advantages and disadvantages associated with each type of policies. Typically, space-sharing polices are good at low to moderate system loads and when jobs parallelism do not vary much. However, at high system loads and widely varying job parallelism, time sharing policies provide a better performance. In this paper we propose a new policy that is based on a hierarchical organization that incorporates the merits of these two types of policies. The new policy is a hybrid policy that uses both space-sharing as well as time-sharing to achieve better performance. We demonstrate that, at most system loads of interest, the proposed policy outperforms both space-sharing and time-sharing policies by a wide margin.

1. INTRODUCTION

Distributed-memory multicomputer systems are an important class of parallel processing systems for building large scale computer systems. The Intel Paragon, NCUBE2S, and Cray T3E systems are examples of commercial multicomputer systems. For instance, the Intel TeraFlop system, which is also a distributed-memory system, uses more than 9000 Pentium Pro processors to achieve 1 Teraflop processing rate. Similarly, the Ncube2S system can be expanded up to 8192 processors. These systems do not provide shared memory as shown in Figure 1 and use message passing for communication among processors. In this paper, our focus is on processor scheduling policies suitable for such large multicomputer systems.

We have known that, in the context of uniprocessor systems, sharing the processing power equally among the jobs is important for obtaining good average response times. For example, the performance of the preemptive round robin based policy is independent of the service time variance in jobs whereas the non-preemptive first-come-first-serve (FCFS) policy is extremely sensitive to this variance.

It has been observed that processor scheduling policies for multiprocessor systems should also have a similar property of sharing the processing power equally among the jobs [9,10].

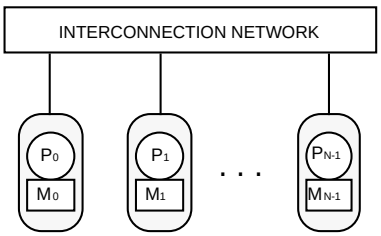

Figure 1 Distributed-memory multicomputer system (P stands for processor and M for memory)

However, due to the presence of multiple processors in these systems, processor sharing can be done in one of two basic ways: sharing them *spatially* or *temporally*. Policies that belong to the first category are called space-sharing policies in which the system of *P* processors is partitioned into *N* partitions and each partition is allocated for the exclusive use of a job. In the extreme case, this type of policy ends up allocating a single processor for each job provided there are at least *P* jobs in the system. Sharing processing power equally among the jobs implies using equal partition sizes. In policies that are temporal (called time-sharing policies), jobs are not given the exclusive use of a set of processors; instead, several jobs share the processors as in, for example, round robin fashion. There are advantages and disadvantages associated withe each type of policies. In Section 2 we discuss these two types of policies and their respective advantages and disadvantages.

We propose a new hierarchical policy that combines the best features of space- and time-sharing policies. This policy is based on the hierarchical run queue organization proposed in [5]. A detailed description of this policy is given in Section 3. A performance comparison of these three policies – space-sharing, time-sharing and hierarchical – is given in Section 4. The results presented in this section demonstrate that the hierarchical policy performs substantially better than the space- and time-sharing policies, particularly at moderate to high system loads. The paper concludes with a summary.

2. SPACE AND TIME SHARING POLICIES

Here we describe the two basic types of scheduling policies suitable for distributed-memory multicomputer systems: space-sharing and time-sharing.

2.1. Space-Sharing Policies

Space-sharing policies can further be classified into either static or dynamic. The dichotomy in the policies is particularly important in this case. In static space-sharing policies, processor allocation remains fixed during the lifetime of a job. For example, if a job is given eight processors, none of them is freed until job completion. A problem with this approach is that some processors may be idling if the scheduled job's parallelism is less than the number of allocated processors. For example, if the serial phase of a job is 20% of the total job execution time, all but one processors are idle for this duration. This scenario is referred to as *fragmentation*. The idea behind dynamic policies is to obviate fragmentation by allocating idling processors to other jobs that can fruitfully utilize them. Thus, in dynamic policies, processor allocation varies over the lifetime of a job by responding to changes in the parallelism of a job. Implicitly, dynamic policies assume that reassigning processors from one job to another does not involve much overhead. However, for the distributed-memory systems, such processor reassignments are expensive. Therefore, static policies are suitable for this type of systems. Dynamic space-sharing policies are typically used in shared-memory systems. Thus, we will discuss only static space sharing policies in the remainder of the paper. In addition, scheduling for these policies is done by a central allocator that can

become a bottleneck for the systems and loads that are being considered here.

Several static policies have been proposed for distributed-memory systems [11]. Due to space limitations, we will only describe the best space sharing policy that we have implemented for comparing the performance of the hierarchical policy. Rosti et. al. [11] have evaluated a set of space-sharing policies in order to study their behavior under various workloads. These policies are based on partitioning the available set of processors in space sharing fashion. The "adaptive" nature of the policies adjusts to various workloads by computing the partition size of each job at schedule time. On each such partition, a program exclusively runs until completion.

The policy that we have implemented is a modified version of their AP2 policy as the performance of this policy is best overall among the policies for all the various types of workloads considered. In the original policy, the partition size is computed as follows:

$$\textit{partition size} = \mathbf{Max}\left(\mathbf{1},\ \mathbf{ceil}\left(\frac{\textit{total processors}}{\textit{queue length} + \mathbf{1}}\right)\right)$$

From the expression above, we notice that AP2 tends toward premature queueing. For example, on a 32-node system, when two jobs arrive at an idle system, the first job is allocated 11 processors and the second 16 processors. However, on the arrival of a third job, a partition size of 16 is assigned for this job. As there are only 5 free nodes, the job waits until a partition is freed. The main reason for this behaviour is that this policy considers only the queued jobs to determine the partition size. In order to obviate this problem, we have modified the basic policy by taking into account the total number of jobs in the system, which is defined as the sum of queued jobs and the scheduled jobs. This modification is shown below.

$$\textit{partition size} = \mathbf{Max}\left(\mathbf{1},\ \mathbf{ceil}\left(\frac{\textit{total processors}}{\textit{queue length} + \mathbf{1} + f * S}\right)\right)$$

where S is the number of scheduled jobs and $0 \le f \le 1$. The value of f can be used to control the contribution of the already scheduled jobs to the computation of the partition size. Note that we get the original AP2 policy by setting f to 0. The motivation behind the modification is two fold. One, the goal of equal partition for each job is better realized by taking all the jobs into account (this is the case when $f = 1$). Two, it provides a good heuristic during processor allocations. We have observed appreciable improvement in performance with the modified policy (relative to the original policy). The amount of improvement obtained is a function of the parameter f, system load, and workload. For the results reported in Section 4, we have used a value of 0.5 for f. More details on the performance sensitivity of the modified policy to parameter f are available in [16].

2.2. Time-Sharing Policies

Time-sharing policies use preemption to rotate processors among a number of jobs that is usually specified by the multiprogramming level (MPL). This is a system parameter that is associated with each processor and represents the maximum number of tasks that can be present in a processor's ready queue. The round robin policy used in uniprocessor systems is not good for multiprocessor systems as the processing power distribution is proportional to the number of tasks in a job. For this reason, this policy is called RR_process or RR_task. Thus, a job with a large number of tasks (typically implying a large job) tends to get more share than a job that is small, which contravenes our basic principle of sharing the processing power equally. What we need is a policy that shares the processing power (more or less) equally among the jobs on a temporal basis. Majumdar et al. [10] have suggested such a policy, called the RR_job, and its performance has been reported in [9]. In this policy, the per job quantum Q is fixed and the per task quantum q is varied depending on the number of tasks in a job. For example, assuming that the number of tasks in a job T is limited to the number processors in the system P, per-task quantum is computed as $q = Q/T$.

The major problem in implementing RR_job type of policies is that they require the use of a central ready queue. We can see that this central ready queue can become a system bottleneck as did the central allocator in the static policies. It has been shown that access contention for such central ready queue can limit the throughput of the system substantially even when preemption is not used (for example, with FCFS non-preemptive scheduling of tasks). This is caused by the removal of tasks from the central ready queue. The use of preemptive RR_job type of policy exacerbates the situation as each preempted task will have to be returned to the central ready queue. This increases the frequency of accesses to the central queue by the processors. The second problem becomes more severe as the quantum size decreases. On the other hand, if we increase the quantum size to decrease the contention problem, performance becomes sensitive to service time variance.

For large systems, it is imperative that we apply round robin policy on a local per-processor level than at the global level. One solution is to use a two-level architecture. There is a per-processor local ready queue and a single central ready queue. Each processor applies the RR policy only on the tasks that are in their local queues (we call this local_RR_job policy). Since preempted tasks are inserted back in the processor's local queue, we can eliminate the second problem mentioned above. The size of the local queue can be set to the multiprogramming level. Setting the MPL level too high causes load imbalance and too low causes an increase in sensitivity to service time variance (at the extreme, if MPL is 1, it degenerates to FCFS policy). In this policy, whenever the local queue length falls below MPL, it transfers tasks from the central ready queue to its local queue. In summary, time-sharing is desired to improve performance when job's parallelism varies. This is particularly important at moderate to high system loads.

The next section discusses the use of hierarchical ready queue organization to effectively combine the best features of the space-sharing and time-sharing policies discussed here.

3. HIERARCHICAL SCHEDULING POLICY

It has been demonstrated that a combination of space- and time-sharing is beneficial in making scheduling decisions for large scale systems [12]. However, this study has shown the potential benefits in principle only. As far as deriving a policy that incorporates this hybrid nature is concerned, ours has been the first attempt in this direction for distributed-memory systems. We propose the use of hierarchy to implement such a policy, which is described next.

The hierarchical scheduling policy is based on the hierarchical run queue organization described in [3,5]. In this organization, a set of ready 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 2 shows an example hierarchical organization for an 8-processor system with a tree branching factor B of 2.

The hierarchical policy is a self-scheduling policy. This implies that there is no central scheduler making processor allocation decisions. Hierarchical policy combines the merits of both space-sharing and time-sharing policies to derive its performance advantage. The policy works as follows.

In the hierarchical organization, all incoming jobs are placed in the root queue. Let L be the leaf node level (i.e., processor level) in the tree. When a processor is looking for work, it first queries its associated task queue at level (L–1). If that queue is empty it in turn queries its parent node at level (L–2) and the process is repeated up the tree until it finds a job or task to be scheduled (unless the root queue is empty). However, in order to reduce access

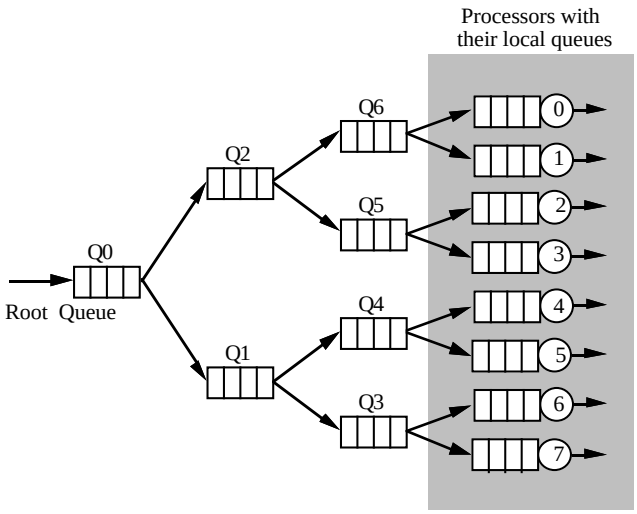

Figure 2 Hierarchical ready queue organization for $P = 8$ processors with a branching factor B of 2

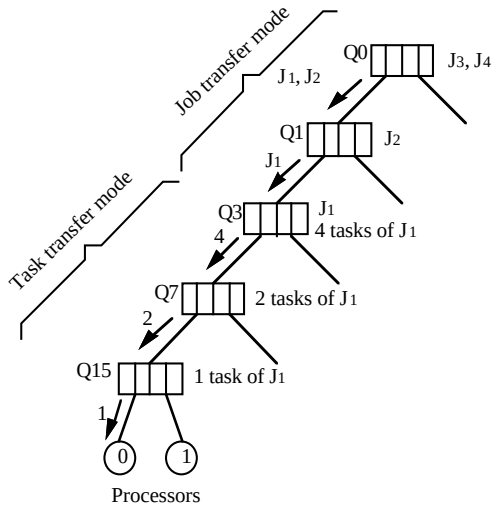

Figure 3 Job and task transfer process of the hierarchical policy (number of processors $P = 32$ and the branching factor $B = 2$)

contention at higher levels, when a ready queue is queried, a set of jobs is moved one level down the tree if the number of jobs at the task queue is greater than or equal to B. Otherwise, tasks are transferred analogously. The size of the set decreases progressively as one goes down the tree (taking into the account the availability of 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).

In an attempt to enhance performance, the hierarchical policy first applies the space-sharing policy to assign a partition to a waiting job. To explain the policy, let us assume that all ready queues are empty and the root node has four jobs J_1 to J_4 waiting to be executed on a 32-processor system with $B = 2$. An idle processor (assume processor 0) queries the root queue as all the queues along the path are empty. Since the branching factor is 2 and there are more than 2 jobs waiting at Q0, the policy operates in job transfer mode. In this mode, the entities transferred between successive levels of the tree are the entire jobs. In the example, P0 moves $1/B$ of the four waiting jobs one level down the tree to Q1 as shown in Figure 3 (leaving J_3 and J_4 at the root node). By this move the jobs J_1 and J_2 are restricted to a partition in the left half of the system (left 16 of the 32 processors). This is analogous to space sharing. At Q1 the policy still applies space sharing component as the number of jobs is greater than or equal to the branching factor B. Therefore, processor P0 moves J_1 to Q3. At this point, J_1 is allocated a partition that consists of all the processors for which the ready queue Q3 acts as the root node. Thus, the partition allocated to J_1 consists of 8 processors P0 through P7. At this point in the transfer process, the hierarchical policy switches to task transfer mode to implement a time-sharing policy.

The task transfer policy works as follows. 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:

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

In this paper we use a Tr of 1 as it has been shown to provide good overall performance [3,5]. As J_1 is assigned 8 processors, it is split into 8 tasks. Since $Tr = 1$, four tasks of J_1 are moved from Q3 to Q7 as shown in Figure 3. The process is repeated at Q7, which results in two tasks moving from Q7 to Q15 and finally moving one task to processor P0. Notice that this process leaves varying number of tasks or jobs along the path. Thus, it is not necessary for some queries from other processors to reach the root queue to get work. For example, if processor P1 has a number of tasks one less than its multiprogramming level, it can simply access Q15 and receive a new task. Similarly, a processor (for example, processor P_{10}) can work on job J_2 without accessing the root queue. A formal description of the hierarchical policy is presented in [15,16].

It can be seen that the set of ready queues that form the tree can all be distributed to different processors without causing any hot spots. For a system with P processors, the maximum number of ready queues are required when the branching factor B is 2 and this number is equal to $(P–1)$. Since there are P nodes these $(P–1)$ queues can all be distributed without causing any memory or communication contention problems.

4. WORKLOAD MODEL

We use a fork-and-join type of job structure, in which a job is divided into a set of tasks. On processing, these tasks synchronize at a single point. We report results for two classes of workload that are representative of the relative performance of the three policies.

In the first workload model (the ideal case), we assume that all tasks of a job have equal service demand. For example, if a job's cumulative service demand is 8 minutes and it is divided into 4 tasks, each task will have a service demand of 2 minutes. This workload shows the advantages of the space-sharing policy. We refer to this as the workload W1.

In the other workload, referred to as W2, the job cumulative service demand is unevenly distributed among the tasks of a job to represent the effects of varying job parallelism. We achieved this by assigning either 50% or 25% of the evenly distributed task service time (i.e., the task service time used in workload W1) to half the tasks and correspondingly increasing the service time of the remaining tasks. (Results for other assignments are given in [16].) The motivation for this is to represent, at least optimistically, applications such as the Barnes and Hut clustering algorithm [2] which are characterized by a variable fork-join parallelism structure. In this workload, if a job arrives with a total service demand of 8 minutes and the job is divided into four tasks, two tasks will have a service demand of 1 minute (i.e., 50% of 2 minutes) and the other two tasks will have 3 minutes of service demand. This type of workload shows the negative aspects of the

space sharing and demonstrates the advantages of the time sharing policies.

For our simulation experiments, we derived the different values of the workload in the following manner. For each job *j*, we generated job service demand using a 2-stage hyper-exponential distribution with mean 13.76 minutes and coefficient of variation (CV) of 10 in accordance with the results made available by different supercomputing centers. Each such job was broken down uniformly into *t* tasks with $t \in [1,32]$. The parameter *t* in this case represents the maximum parallelism of the job as defined in [14]. We have modelled the various overheads associated with the policies described above. These include the overheads in moving jobs and tasks between queues, polling a queue to see if tasks are available, context switching between tasks in processor ready queues, etc.

We have used a quantum size of 20 ms. This is a reasonable value as Fujitsu AP3000 and Ncube use a 10ms quantum length and FLASH uses a quantum length of 30ms. The context switch time is assumed to be 30 μs, which again is based on the observation that the context switch time of FLASH is between 30-50μs and that of Ncube systems is about 10μs. The communication delay in task and job transfers is a function of the amount of data being transferred and is given by the formula [17]:

$$StartUpTime + (MsgSize/AsympBandwidth)$$

where *StartUpTime* is the start-up time, *MsgSize* is the size of the message in bytes, and *AsympBandwidth* is the asymptotic bandwidth of the network. The size of one task along with the data required by the task (this is necessary because we are considering a distributed-memory system) is assumed to be 1KB. We have used a *StartUpTime* of 40μs (based on the data from the IBM SP2 running pure MPI) and the *AsympBandwidth* is assumed to be 5 MBps. A detailed discussion of the overheads is available in [15,16]. In the following we present some sample simulation results for the two types of workloads described.

5. PERFORMANCE COMPARISON

We have conducted a detailed simulation study of the three policies described in Sections 2.1, 2.2, and 3. This section presents a few sample results of the simulation experiments to demonstrate the performance superiority of the hierarchical policy.

5.1. Performance with Workload W1

Figure 4 shows the performance of the three policies as a function of offered system load for a system with 64 processors. Offered system load is the load caused by executing the tasks and does not include any type of overhead or the processor idle times forced by a scheduling policy (e.g., space sharing policy). The results reported here used the default parameter values given in the last section. The multiprogramming level used in the time sharing as well as the hierarchical policies is 2. We have also conducted experiments with other multiprogramming level values but we will not discuss the performance sensitivity of these two policies to this parameter here. Since all tasks of a job have equal service demand in this workload, the space sharing policy performs better than the time sharing policy. The performance superiority of space sharing increases with system load. This performance superiority is because of two reasons. One, all the tasks of the job have equal service demand. Therefore, no fragmentation occurs as all tasks of a job finish at the same time. Two, time sharing suffers due to multiprogramming the tasks and the corresponding context switch delays.

The hierarchical policy exhibits superior performance than the other two policies. The improved performance of the hierarchical policy over the space sharing policy is due to the fact that even though it uses space sharing, because of self-scheduling nature of the policy, entire partition of free processors is not necessary in order to start executing a job. The space sharing policy suffers due to this requirement. For example, if the partition size for a job is 10, and there are only 9 free processors, space sharing policy will keep the job waiting until another processor becomes available before scheduling the job. This causes severe performance problems, particularly at high system loads. The hierarchical policy successfully avoids such problems to provide substantial performance improvements.

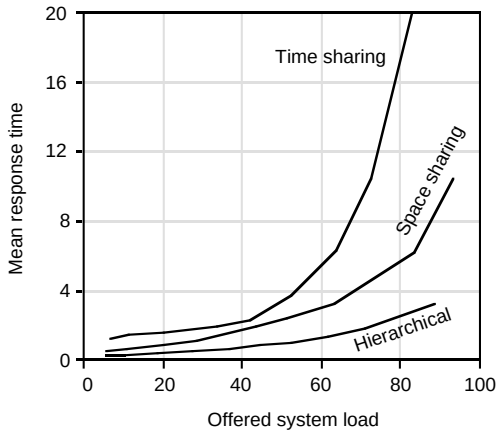

Figure 4 Performance of the three policies under workload W1

5.2. Performance with Workload W2

This section presents the results of the simulation experiments in which the task size is not evenly distributed among the tasks of a job. The first two sections assume that 50% of the tasks get 50% of the evenly distributed task sizes (we refer to this as 50-50 distribution). The last section discusses the impact of 50-25 task distribution on the performance.

5.2.1. Performance with 50-50 distribution

The second workload distributes a job's total service demand unevenly as described in Section 4. This section assumes 50-50 distribution. Such imbalance in task service time distribution causes severe performance problems for the space sharing policy. For example, if a job with a total service demand of 16 minutes is allocated a four-processor partition, and two tasks of the job have a service demand of 2 minutes and the other two tasks have a service demand of 6 minutes, this allocation idles two processors for 2 minutes. Thus, the overall utilization for this partition is 16/24= 67% to execute this job. That is 1/3 of the processing power is wasted. As a result of such behaviour, the performance of the space sharing policy deteriorates rapidly as shown in Figure 5. For this workload, it has been observed that the average partition size under the space sharing policy is 16 for a 64-processor system and the partition utilization in some cases is as low as 40%.

The performance of the time sharing and hierarchical policies is substantially better than the space sharing. This is mainly achieved by multiprogramming the tasks on a processor. Amongst the two, time sharing performs better than hierarchical policy at low loads. This relatively poor performance of the hierarchical policy is due to the space sharing nature exhibited by hierarchical policy at low system loads.

At moderate to high system loads where the performance is of interest, the hierarchical policy performs better than space sharing. The performance difference between the two increases with the system load. The performance deterioration of the time sharing policy is a direct consequence of the contention for the global queue. Contention for the global is avoided by the hierarchical policy [3,5].

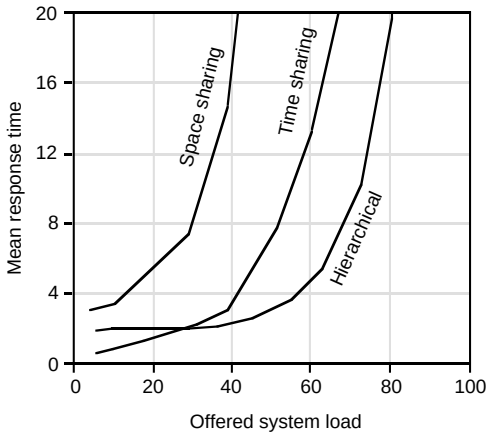

Figure 5 Performance of the three policies under 50-50 distribution (service demand CV = 10)

5.2.2. Sensitivity to variance in job service demand
Recent studies in workload characterization have reported very widely varying service demands at high performance centres [6,7,8]. This gives rise to workloads composed of varying job sizes. Figure 6 shows the performance of the three policies at low variance in service demand. Performance under high variance in service demand is illustrated in Figure 7. From Figures 5, 6, and 7, it can be seen that the performance of space sharing deteriorates with increasing variance in service demand. This behavior is due to the fact that in a mixture of jobs consisting of many small ones and a few large ones, there is a tendency for the smaller jobs to be enqueued for long durations because of larger jobs in front of them. This leads to premature queuing as there will arise situations wherein the number of free processors may not meet the needs of the large job for a very long time though they might easily satisfy the smaller succeeding ones.

The inferior performance of time sharing as against hierarchical policy is due to the following two factors:

(a) Higher variance in service demand implies a large number of small jobs and a small number of large jobs. Due to the presence of the large number of small jobs, variance in inter-access times of the central queue also increases--thereby increasing the potential for contention. This can be seen from the following example service demands of two job streams of six jobs each: 1, 1, 1, 1, 1, 7 and 1, 1, 2, 2, 3, 3. Even though both job streams have an average service demand of 2 time units, the first stream has a higher variance and causes the inter-access times of the central queue to be small for five jobs as there are five jobs with a service demand 1 as compared to the second job stream.

(b) The second effect is due the presence of the large jobs. Since round robin policy is applied to the local queues, large jobs tend to make the system work in a quasi-FCFS mode. To illustrate this point, let us assume that a large job with 64 tasks is moved into the local queues of a 64-processor system. If the multiprogramming level is 2, that leaves only one slot for executing other jobs. Clearly, with two such large jobs, all other (possibly) short jobs will have to wait until at least one of these jobs is done. As a result of this quasi-FCFS behaviour, time sharing exhibits more sensitivity to variance in service demand.

The hierarchical policy avoids this type of behaviour as it first uses space sharing to limit large jobs to a set of processors in a partition. On top of that, it applies time sharing to derive the benefits of the time sharing policy. As a result, it exhibits the least sensitivity to variance in job service demands.

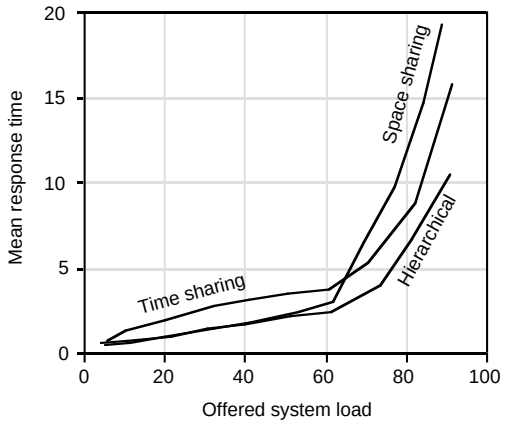

Figure 6 Performance of the three policies under 50-50 distribution (service demand CV = 1)

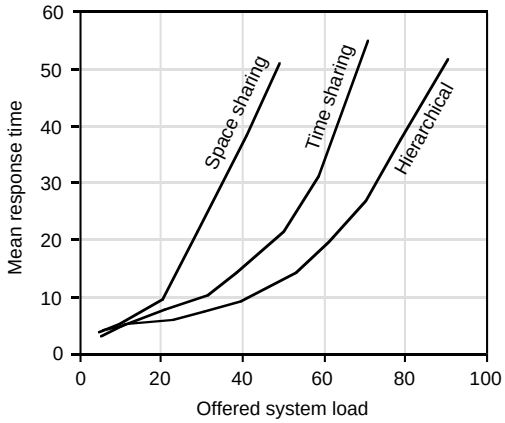

Figure 7 Performance of the three policies under 50-50 distribution (service demand CV = 15)

4.2.3. Performance under 50-25 distribution
A second variation in the imbalance was introduced as follows. Here, 50% of the tasks of a job received 25% of the service demand while the remaining job service demand was distributed evenly amongst the remaining 50% of the tasks. Figure 8 illustrates the relative performance of the three policies under this consideration. As in the previous case, the performance of space sharing is abysmally poor than the time sharing and hierarchical policies. The reason for this is the extensive fragmentation caused by the imbalance. To see this, consider the example introduced in the previous section. For this job, due to the 50-25 criterion, two tasks will have a service demand of 1 minute (25%) while the remaining two tasks (50%) will have a service demand of 7 minutes each. The utilization of the partition in this case would be 16\28 which is about 57%. Extrapolating this argument to encompass the whole system, we observed that more than 40% of the processors idle even when jobs are queued for service. In fact, this effect is sole cause for the almost exponential rise in mean response times of the jobs even at low loads for space sharing. Results for 50-75 distribution are given in [16].

Hierarchical and time sharing, on the other hand, are more stable and their relative performance is much better due to multiprogramming. In addition, the inherent self-scheduling nature of hierarchical policy overcomes the bottleneck problem for the global queue as perceived in time sharing. This is the prime reason for the better performance of the hierarchical policy.

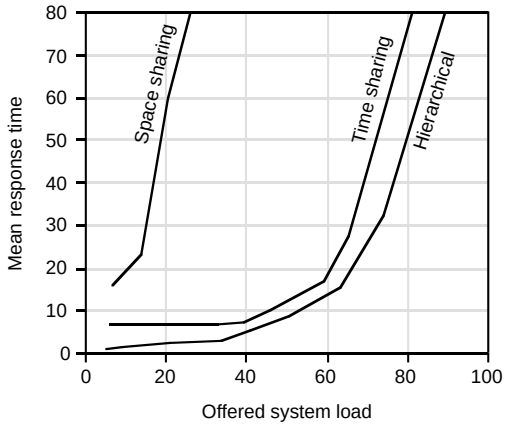

Figure 8 Performance of the three policies under 50-25 distribution (service time CV = 10)

5. CONCLUSIONS

In this paper we have proposed a hierarchical policy that combines the merits of the space sharing and time sharing policies while, at the same time, overcoming their potential problems. We have shown that the hierarchical policy outperforms the space sharing policy by a wide margin. As well, it provides far superior performance than the time sharing policy except at low system loads. Since large parallel systems are unlikely to operate at low system loads, for all practical purposes, the hierarchical policy provides substantial performance improvements over the space sharing and time sharing policies.

An important point to note is that we have given benefit of doubt to space sharing and time sharing. In our implementation, these two policies assume that the processors in the system are treated as a "pool of processors." For instance, if a partition size is four and there are four free processors in the system, the space sharing policy allocates these four processors to the partition no matter where these processors are located in the system. This kind of allocation is fair only if the system is entirely bus-based. For other types of interconnects such as mesh, processor proximity is important even if wormhole routing is employed.

Large-scale distributed multicomputers tend to use hierarchical interconnection networks. In such systems, it is important to allocate nodes on a cluster-by-cluster basis. Our hierarchical policy performs processor allocations in this manner. In our implementation, the other two policies are not restricted this way (e.g., it is possible to assign four nodes with each node belonging to a different cluster). When such restrictions are imposed on space and time sharing, their performance will be even worse than that reported in this paper.

The hierarchical policy has also been shown to provide better performance in shared-memory NUMA systems [1]. Furthermore, the principle of the hierarchical strategy has been adapted for parallel processing of database queries on networks of workstations [4].

We are currently investigating the performance of these three policies for a more realistic workload. These results strongly support the conclusions given here [15,16]. As a future work, we are planning to implement the policies on a network of workstations to experimentally evaluate the relative performance of these policies.

ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support provided by the Natural Sciences and Engineering Research Council of Canada and Carleton University.

REFERENCES

[1] S. Ayachi and S. Dandamudi, "A Hierarchical Processor Scheduling Policy for Multiprocessor Systems," *Symp. Parallel and Distributed Processing*, New Orleans, 1996, pp. 100-109. (An expanded version is available as a technical report TR-96-21 from http://www.scs.carleton.ca).

[2] J. Barnes and P. Hut, "A Hierarchical O(n lg n) Force Calculation Algorithm, *Nature*, Vol. 324, 1986, pp. 446-449, .

[3] S. P. Dandamudi "Reducing Run Queue Contention in Shared Memory Multiprocessors," *IEEE Computer*, Vol. 30, No. 3, March 1997, pp. 82-89.

[4] S. P. Dandamudi and G. Jain, "Architectures for Database Query Processing on Networks of Workstations," *Int. Conf. Parallel and Distributed Computing Systems*, New Orleans, 1997 (available from `http://www.scs.carleton.ca`).

[5] 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.

[6] D. G. Feitelson and B. Nitzberg, "Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860," *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, and L. Rudolph (eds.), Vol. 949, Lecture Notes in Computer Science, Springer-Verlag, 1995,.pp. 337-360.

[7] S. Hotovy, "Workload Evolution on the Cornell Theory Center IBM SP2," *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, and L. Rudolph (eds.), Vol. 1162, Lecture Notes in Computer Science, Springer-Verlag, 1996,.pp. 27-40.

[8] J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riordan,"Modeling of workload in MPPs," In *Job Scheduling Strategies for Parallel Processing*, D.G. Feitelson, and L. Rudolph (eds.), Lecture Notes in Computer Science, Springer-Verlag, 1997.

[9] S. T. Leutenegger and M. K. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies," *Proc. of ACM Sigmetrics Conf.*, Boulder, 1990, pp. 226-236.

[10] S. Majumdar, D. L. Eager, and R. B. Bunt, "Scheduling in Multiprogrammed Parallel Systems," *Proc. of ACM Sigmetrics Conf.*, Santa Fe, NM, May 1988, pp. 104-113.

[11] E. Rosti, E. Smirni, G. Serazzi, L.W. Dowdy, and B.M. Carlson, "Robust Partitioning Policies for Multiprocessor Systems," *Performance Evaluation*, Vol. 19, 1994, pp. 141-165,

[12] S. K. Setia, M. S. Squillante, and S. K. Tripathi, "Processor Scheduling on Multiprogrammed, Distributed Memory Parallel Systems," *Proc. ACM SIGMETRICS Conference*, May 1993, pp.158-170.

[13] K. C. Sevcik, "Characterizations of Parallelism in Applications and Their Use in Scheduling," *Proc. ACM SIGMETRICS Conference*, May 1989, pp.171-180.

[14] K. C. Sevcik, "Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems," *Performance Evaluation*, Vol. 19, 1994, pp. 107-140.

[15] T. K. Thyagaraj and S. Dandamudi, *HSP: A Novel Method for Job and Task Scheduling for Distributed Memory Multicomputers*, Technical Report, School of Computer Science, Carleton University, Ottawa, 1997 (available from http://www.scs.carleton.ca).

[16] T. K. Thyagaraj, *A Hierarchical Scheduling Policy for Large-Scale Distributed memory Multicomputer Systems*, MCS Thesis, School of Computer Science, Carleton University, Ottawa, 1997 (available from http://www.scs.carleton.ca).

[17] Z. Xu and K. Hwang, "Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2," *IEEE Parallel and Distributed Technology*, Spring 1996, pp. 9-23.