An Efficient Adaptive Scheduling Scheme for Distributed Memory Multicomputers

Thyagaraj Thanalapati and Sivarama Dandamudi, *Member*, *IEEE*

Abstract—Traditional multiprocessor scheduling schemes have been one of either space-sharing or time-sharing. Space-sharing schemes perform better than time-sharing at low to moderate system loads. However, they have a disadvantage of wasting processing power within partitions at medium to high system loads. Time sharing schemes tend to perform better at medium to high system loads. Almost all the scheduling schemes proposed so far have been tested under ad hoc workload considerations. In light of recent knowledge about workloads, it is imperative to develop an integrated scheduling scheme that combines the advantages of space- and time-sharing while overcoming their individual drawbacks. We propose such a scheduling scheme, called Hierarchical Scheduling Policy, which is efficient as well as general enough to accommodate multiple workloads. Simulation results indicate that our scheme significantly outperforms the best space- and time-sharing mechanisms at medium to high system loads even in the absence of knowledge regarding individual job characteristics.

Index Terms—Multicomputer systems, job/task scheduling, space partitioning, time sharing, hierarchical scheduling, multiple workloads, performance evaluation.

1 Introduction

Scheduling policies for large-scale distributed-memory multicomputers (DMMs) can be broadly classified into *space-sharing* and *time-sharing* policies. Space-sharing policies partition the system processors and each partition is allocated exclusively for a job. In time-sharing policies, processors are temporally shared by jobs.

Each type of policy has its advantages and disadvantages. Space-sharing schemes perform well under low system loads. However, at medium to high load ranges, they cause starvation of processing power. Furthermore, space-sharing utilizes processors poorly when there is high variance in execution times. Time-sharing, on the other hand, tends to sacrifice the performance of individual applications due to their aggressive nature to achieve better system utilization.

Recent studies in the realm of workload characterization at various high performance computing centers have shown several interesting characteristics [6], [13], [15], [16]. For instance, observations of high variance in service time distributions of jobs [6], uniform distributions of resource requirements of large and small jobs [13], interarrival time and service time distributions were dispersive with multiple phases [16], and so on.

In lieu of the insights gathered about characterization of many workloads, it now seems that the aforementioned problems arise even under very conservative and optimistic considerations. A study of various scheduling schemes for multicomputers makes it apparent that a few issues determine the robustness and efficiency of a scheduling policy:

- the presence of multiple job queues so as to overcome any scheduling bottlenecks,
- combined space-sharing and time-sharing to exploit their advantages while minimizing their individual disadvantages. Several studies have indicated that space partitioning together with time sharing within a partition leads to the best performance [12], [14], [23],
- adaptive load distribution for suitable management of processing power based on current system load,
- user level implementation to avoid changes to the underlying operating system as well as higher portability,
- ease of programming from an application programmer's view point.

These issues are the motivation for us to propose hierarchical scheduling policy (HSP) for scheduling in large-scale multicomputer systems. HSP is robust in that it is based on the hierarchical task queue organization [7]. It achieves efficiency by combining space partitioning and time sharing while being able to adaptively distribute the system load based on current load. In addition, it can be implemented purely in software at the user level in a straightforward manner. Furthermore, it allows a uniform framework for application programming. Detailed simulation results incorporating a large number of parameters from actual workload characterizations show that HSP outperforms the best known space- and time-sharing policies while exhibiting many advantages over gang scheduling [7].

The rest of the paper is organized as follows: Section 2 presents a brief survey of proposed schemes for

- *T. Thanalapati is with IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7, Canada. E-mail: tkthyag@ca.ibm.com.*
- *S. Dandamudi is with the Centre for Parallel and Distributed Computing, School of Computer Science, Carleton University, Ottawa, Ontario K1S 5B6, Canada. E-mail: sivarama@scs.carleton.ca.*

Manuscript received 2 Oct. 1998; revised 6 July 2000; accepted 1 Feb. 2001. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 107493.

1045-9219/01/$10.00 © 2001 IEEE

multiprocessors as well as workload characterization. A comprehensive description of HSP is provided in Section 3. Section 4 describes the space partitioning and time sharing policies that we have used for comparison purposes together with a description of our simulation environment. Detailed simulation results are presented and discussed in Section 5. Conclusions are given in the last section.

2 Background and Related Work

In this paper, we refer to a program as a *job* or an *application*. A job consists of one or more *tasks* or *processes*. During execution, depending on the number of processors allocated, a job may run as a single entity or as multiple parts (tasks) of the single entity. A job is said to be complete only when all of its tasks are executed.

2.1 Scheduling Policies

Several scheduling schemes have been proposed for shared-memory multiprocessors and distributed-memory multicomputers (DMMs). In this section, we highlight only a few of them which exhibit certain interesting features that are in the scope of this discussion.

A fundamental result for scheduling in parallel systems is *co-scheduling*, due to Ousterhout [20], in which all the tasks of a parallel application were scheduled to run simultaneously on as many processors in a round-robin fashion. Feitelson and Rudolph propose *gang scheduling* in which co-scheduling is implemented, but within nonoverlapping, multiple partitions [14]. McCann and Zahorjan proposed two families of scheduling policies, called *folding* and *equipartition*, whose aim was to provide fairness and equal processor sharing to jobs [18]. Setia et al. [23] proposed adaptive multiprogrammed partitioning to overcome the problems with static partitioning so as to avoid application reconfiguration and data migration by space-sharing and time slicing a partition of processors among multiple programs. Adaptive space partitioning was proposed by Rosti et al. [22] in which a characterization of various schemes was done when given little or no knowledge of job characteristics.

2.2 Scheduler Organization

Performance of a scheduling policy in DMMs is directly influenced by the scheduler (task queue) organization. Hence, a number of studies have concentrated on testing various designs [14], [7], [12]. Of these, only Feitelson and Rudolph [14] focus on distributed memory systems. So far, the consensus has been to either use a single global queue or have a distributed organization. Only recently has the hierarchical task queue organization been proposed as a viable alternative [12].

In terms of the scheduler organization, Feitelson and Rudolph proposed and implemented a tree-based queue organization completely in hardware using simple chips [14]. Processors were then connected as the leaf nodes of the tree. The control at the queue level was implemented in software. Our proposed policy is different from the scheme used by Feitelson and Rudolph. Their goal in presenting the hierarchical control was to implement gang scheduling, thus, the need for a hardware-oriented approach. In contrast, we propose a scheme that can be implemented entirely in software. Our software-oriented approach is attractive from a practical standpoint as it can be modified without requiring changes to the hardware. In addition, our policy is completely general and can be implemented on a DMM even after the design has been completed.

Dandamudi and Cheng first proposed the hierarchical task queue organization that we use here [7]. The motivation for this proposal was to improve the run queue performance in shared-memory multiprocessors. Based on simulation experiments, they observed that the hierarchical organization performed much better than the corresponding distributed organization while achieving the load sharing property of the centralized queue system.

Studies on shared-memory multiprocessor systems have shown that good performance can be achieved using the hierarchical scheme [28], [7]. Zhou and Brecht [28] have proposed a scheduling policy that is based on the concept of a processor pool. Note that their two-level policy is for multiprocessors, not for DMMs. In contrast, our policy is targeted exclusively for DMMs and more general in nature. The number of levels in our hierarchy, for example, is not fixed and is design- and system-dependent.

Conceptually, our algorithm could be applied to other types of systems. However, the algorithm as presented here is designed for DMMs. The algorithm takes into account that DMMs are loosely coupled, have higher communication latencies, and use nonshared memory and message passing. These facts make space-sharing appropriate in some of these systems. While such space-sharing policies can also be used in SMPs, there are better policies considering the tightly coupled nature of these systems. For instance, Equipartition policy proposed by McCann and Zahorjan [18] is better for SMPs that are small in size and have low latency communication. As another example, the processor pool policy proposed by Zhou and Brecht [28] is for large-scale NUMA systems. While this policy can be used in DMMs, it needs to be modified to take the previously mentioned characteristics of DMMs.

2.3 Workload Characterization

The traditional approach to workload characterization when proposing scheduling schemes has been to use a model that characterizes the structure of a single class of applications [23], [12]. Research in workload characterization has been very modest until recently. Almost all the studies carried out during this period provide characteristics about the execution profile of single applications [24]. Only recently, reports about the characteristics of workloads containing multiple jobs have started to trickle in [13], [15], [16]. Observations indicate very general characteristics than those suggested by analytical techniques for single applications. For example, the often used exponential service time seems very restrictive as against a combination of service time distributions such as exponential, hyper-exponential, and even simple Erlang in the case of real workloads [16].

Feitelson and Nitzberg [13] analyzed traces of a production parallel workload on a 128-node Intel iPSC/860. They

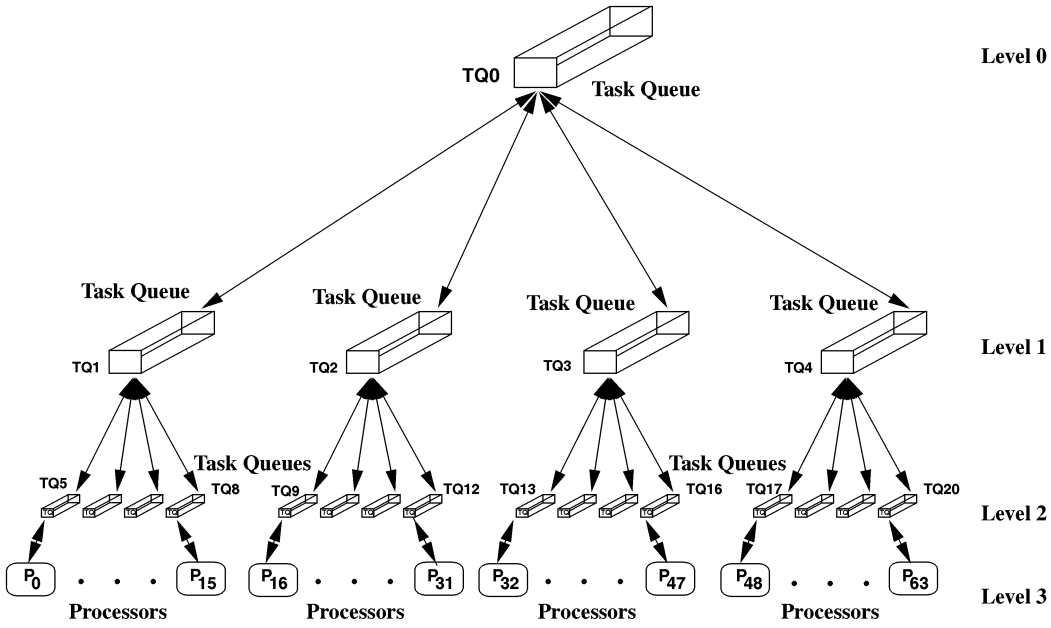


Fig. 1. A logical view of hierarchical task queue with $P = 64$, $B = 4$, and $L = 3$.

observed that large jobs tend to consume more resources than smaller ones, thus showing a roughly uniform distribution of the user jobs. In addition, a high coefficient of variation was observed in job runtimes and interarrival times, indicating that they may be hyperexponentially distributed.

In a characterization of the workload traces from a supercomputing environment, Jann et al. [16] observed that the jobs are quite dispersive in terms of the sizes as well as arrival patterns. This is conversant with the observations of Feitelson and Nitzberg [13]. However, they suggest that the service times should be modeled by a Hyper Erlang distribution on the basis of their observations.

Recent studies have brought out a number of parameters that could not have been anticipated in the traditional workload models. For example, variance in expected mean service requirements, as suggested by the study of the NAS application benchmarks [5], arrival patterns in the study of production workload mixes at supercomputing centers [13], high coefficient of variation in service demand for workload mixes in academic environments [6], [13], and so on. Implications of these studies have motivated us to model and use them for comparing and contrasting HSP against adaptive space partitioning (AP2) [22] and round-robin time sharing (RRTS) [17] policies. As per the literature, the performance of these two are comparable to some of the best schemes in their respective classes.

We chose the average response time of individual applications, as well as system utilization, as the performance metrics to evaluate their relative performances. This was an important choice prompted by an attempt to evaluate performance at two levels, namely, the application level and the overall system level.

3 Hierarchical Scheduling for DMMs

3.1 Description of Hierarchical Task Queue Organization

The hierarchical task queue (HTQ) is logically structured in the form of a tree of schedulers or task queues. Leaf nodes in the tree are the processors and the tree nodes are represented by the task queues. For the sake of description, we assume that the tree is L levels deep with the processors residing at level L. The number of children of a node is referred to as its branching factor B. Without loss of generality, we assume that all nodes have the same value of B. However, it should be noted that HTQ does impose this condition: HTQ can be designed with variable B value. However, for convenience, we use a constant B value in our experiments. The effect of B value is discussed in [9]. An example of HTQ is shown in Fig. 1. Each scheduler communicates only with its parent and all children. No parent mediates between the workings of any of its children. In addition, higher level queues do not keep track of any details of their children. All the processors reachable from a task queue constitute its *partition reach*. Note that the partition reach is a function of B and the task queue level in the hierarchy. For example, in Fig. 1, the partition reach of a task queue at level 2 is B, at level 1 is B^2, etc. It can be seen that this concept induces the space partitioning mechanism. For instance, jobs residing at a task queue can only be scheduled on the processors determined by its partition reach. Additionally, the goal of uniform load sharing is also possible by viewing each task queue as a central queue serving the processors in its partition reach.

3.2 Job and Task Transfer Process

All incoming jobs are enqueued at the root in the hierarchy (Level 0 in Fig. 1). HSP employs *self-scheduling* while transferring work, that is, jobs and tasks. The *transfer factor* (Tr) determines the number of tasks that can be moved

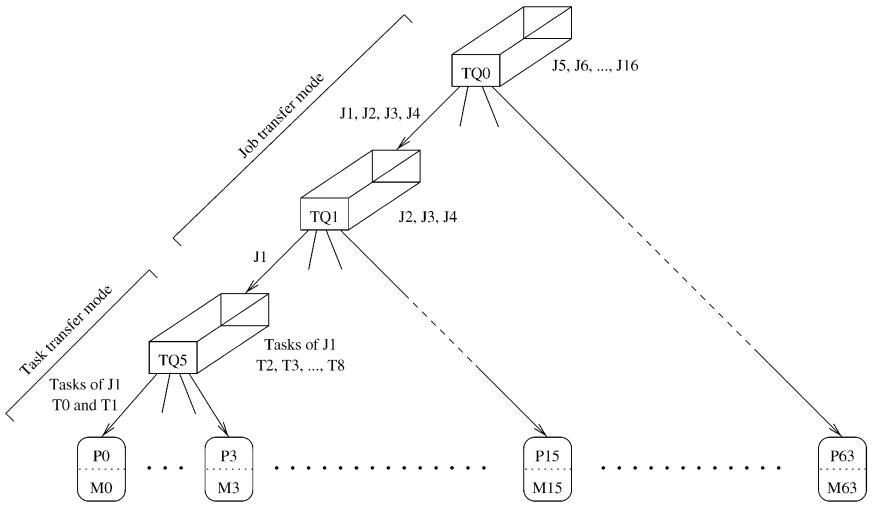


Fig. 2. Job and task transfer process in HSP in a system with number of nodes $P = 64$ and branching factor $B = 4$.

from a parent to any one of its child task queues [7]. Tr is defined as

$$Tr = \frac{\text{Number of tasks moved one level down the tree}}{\text{Partition-reach of child task queue}}$$

To illustrate the process, assume that TQ0 at level 0 has a job with 50 tasks. When TQ0 receives a request for work from TQ1, it would transfer (Tr^* partition reach of TQ1) tasks. Assuming $Tr = 1$, this results in transferring 16 tasks as the partition reach of TQ1 is 16. By this action, space partition is accomplished, i.e., these 16 tasks can only be executed on the partition consisting of processors P_0 to P_{15}. The remaining $50 - 16 = 34$ tasks can be scheduled on any processor in the system as they are at the root node. If, for example, we change Tr to two, we would have transferred 32 tasks instead of 16. Thus, increasing Tr value decreases the scope of load sharing. The effect of B and Tr parameters on the performance of HTQ is described in [7], [9].

Task/job transfer process depends on the level of the task queue. We always move tasks from a processor's parent queue to the processor ready queue. At higher levels of the tree, transfers can be tasks or entire jobs. The following description provides details on these two cases.

3.2.1 Task Movement at Level $L - 1$

We use MPL to represent the *multiprogramming level*. MPL indicates the maximum number of tasks that can be scheduled on a processor at any time. When the number of tasks in a processor's local queue becomes less than MPL, it requests

$$R_t = (\text{MPL} - \text{Number of tasks in the processor ready queue})$$

tasks from its parent task queue that is at level $L - 1$. If the queue has N_t tasks, it satisfies the request by sending $\min(R_t, \lceil N_t/B \rceil)$ tasks to the processor queue. The second factor is used to achieve load sharing by distributing tasks in proportion to the number of processors (which is B) in the partition reach of the queue.

3.2.2 Task Movement at Higher Tree Levels

When a queue Q at a higher tree level (i.e., at a level less than $L - 1$) receives a request for work and the queue is empty, it forwards the first and only request from any of its children to its parent. If the queue is not empty, it satisfies the request by transferring either tasks or jobs. If the number of jobs in queue Q is greater than or equal to B, it works in the *job transfer mode* and sends (number of jobs in this queue/B) jobs to the child node that requested work. The rationale for this is to distribute jobs in proportion to the partition reach of the child node. If the number of jobs in queue Q is less than B, it operates in the *task transfer mode*. In this mode, it will attempt to transfer (Tr^* partition reach of the child task queue of Q) tasks if queue Q has enough tasks available for the transfer; otherwise, whatever tasks Q has will be sent to the child queue.

We illustrate the task/job transfer method with an example. Assume that all ready queues are empty and the root node has 16 jobs, J1 through J16, waiting to be executed on a 64-node multiprocessor (see Fig. 2). A request message from an idle processor (assume P0) will be received by the root queue as all the task queues along the branch $P0 \rightarrow TQ5 \rightarrow TQ1$ are empty. Since the branching factor B is 4 and there are more jobs than B at the root, the policy operates in the job transfer mode. In this mode, the entities transferred between successive levels of the tree are the entire jobs (assuming that the number of tasks in these jobs is satisfied by the task queue's partition reach). In the example, TQ0 transfers four jobs one level down the tree to TQ1 leaving the rest of the jobs at TQ0. By this move, J1 through J4 are restricted to the partition consisting of the processors P0 through P15. This is analogous to space-sharing. At TQ1, HSP still applies a space-sharing component as the number of jobs is equal to B. Therefore, J1 is moved from TQ1 to TQ5. This effectively allocates J1 a partition defined by the partition reach of TQ5. This consists of four processors P0 through P3. At this time, since the number of jobs at TQ5 is less than B, HSP switches to the task transfer mode to implement time sharing among the tasks of J1 and assigns at most MPL number of tasks to the processors P0 through P3 based on requests from the same. In this case, because TQ5 is at level $L - 1$, whether or not the number of jobs is less than B, HSP switches to task transfer mode. Notice that this process leaves behind a varying number of jobs/tasks at different schedulers

along the path $TQ0 \rightarrow TQ1 \rightarrow TQ5$. Due to this, certain requests could be satisfied by intermediate task queues, thereby reducing contention for the root queue.

3.3 Implementation Methodology

In terms of implementing HTQ, there is a need to evaluate both the hardware and software approaches. One hardware-based approach is that of the distributed hierarchical control as proposed by Feitelson and Rudolph [14]. They argue that the hardware approach is advantageous because it allows various optimizations to be performed by the architect with different processors. This is driven by limited hardware requirements and control. In contrast, ours is a completely software driven approach. Reasons for this decision are cited below:

- Our fundamental consideration is a loosely coupled *homogeneous* system. Therefore, all pertinent optimizations are equally applicable over all processors.
- The software approach allows flexibility during modifications of the tree such as its branching factor, etc. This is available to a very small extent in a hardware-based hierarchy.
- The task queues do not need any extra memory since, even with $B = 2$ (the case of maximum number of queues), they could all be mapped onto distinct processors.
- A software-based hierarchy allows a programming interface for scheduling that can be provided at either the kernel or application level.
- The tremendous advances in hardware-software interfaces for high speed communication deems our arguments plausible as observed in various studies [25], [19], [27].

It can be seen from this description that our policy offers significant benefits compared to other policies, including the processor pool policy [28] and the Gang scheduling policy proposed by Feitelson and Rudolph [14]. One of the main reasons is that our policy does not use "all-or-nothing" allocation. Our policy also postpones task-to-processor binding as much as possible; further, it keeps these unbound tasks as high in the tree as possible so that partition reach can be as large as possible. These features substantially improve the overall performance, as reported in Section 5. In addition, our policy does not need a central scheduler, which makes it scalable.

4 Scheduling Policies for Performance Comparison

This section describes two other policies that are used for performance comparison.

4.1 Robust Partitioning—Space Partitioning

Rosti et al. [22] proposed and evaluated five adaptive policies, named AP1, AP2, AP3, AP4, and AP5. 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 of interest to us is AP2 as its performance is comparable to the best policies while, at the same time, being less complex in implementing the split and merge operations. In this case, the partition size is computed as follows:

$$partition_size \leftarrow max\left(1, \left\lfloor \frac{Total\ Processors}{Queue\ length + 1} + 0.5 \right\rfloor\right).$$

AP2 has a tendency toward premature queuing as jobs tend to wait until there are more free processors than requested. In addition, scheduling decision is made by considering only the queued jobs. This acts against fair sharing of processing power amongst the jobs. In order to obviate this problem, the basic policy has been modified by taking into account the total number of jobs in the system [11]. This modification is shown below.

$$partition_size \leftarrow max \left(1, \left\lfloor \frac{Total\ Processors}{Queue\ length + (f^{*}Scheduled\ jobs) + 1} + 0.5 \right\rfloor\right).$$

The resulting adaptive policy is referred to as MAP2. The parameter f, which ranges between 0 and 1, controls the contribution of the already scheduled jobs to the computation of the partition size. Note that, at 0, MAP2 degenerates to AP2. MAP2 has been shown to provide better performance than the original AP2 policy [11]. The amount of improvement obtained by MAP2 is a function of the parameter f, system load, and workload. We have used a value of $f = 0.5$ as this value provided the best performance, in most cases, under various system and workload conditions.

4.2 Job-Based Round Robin—Time Sharing

We have chosen the job based round-robin policy proposed by Leutenegger and Vernon [17] from the time-sharing class. The reason to consider job granularity rather than task is uneven processing power distribution. That is, a large job tends to execute for more quanta than smaller ones. This is in contravention to the basic principle of equal processor sharing, which is known to enhance the performance [17]. In this scheme, the per-job quantum Q is fixed while the per-task quantum q is varied depending on the number of tasks T in a job. q is set as Q/T.

In the original policy, Leutenegger and Vernon considered a system with a centralized queue. They then timeshared the tasks on individual processors, putting back context switched jobs at the end of the queue. In large systems, this creates unacceptable access contention for the central scheduler. In order to overcome this problem, we have used a two-level architecture model wherein each processor has a local ready queue in addition to the central queue for the system. All the local queues are uniform and their size is determined by the multiprogramming level (MPL). Note that a choice for the value of MPL has a direct bearing on the load sensitivity of the scheduling scheme. MPL, in a sense, represents the upper threshold used in load sharing policies [8]. Therefore, with the modification, each processor would attempt to get MPL number of tasks for each access to the centralized queue. After this, it would timeshare the processor among the local tasks.

4.3 System and Workload Parameters for Simulations

4.3.1 System Parameters

For all comparisons, we modeled a system consisting of $P = 64$ homogeneous processors. Jobs arrive into the system for service in a Poisson stream with rate λ and are queued at the root task queue (TQ0 in Fig. 1). For each job, the number of tasks is distributed uniformly between 1 and 64. We do not account for the cost of dividing a job into constituent tasks, etc. Within the system, we do not consider the effects of memory requirements and synchronization delays. However, for each job, these are modeled appropriately when generating its execution time based on Sevcik's execution time function [24].

We account for the time quantum and context switch delays during multiprogramming based on the actual overheads in certain existing systems, such as Fujitsu AP3000, NCUBE 3, etc. For instance, in one set of the experiments, the quantum length was 50 ms and context switch time 10 μs.

For modeling communication cost explicitly, we use the model presented by Xu and Hwang [27]. In this model, each message is assumed to incur a cost given by a C_{comm} defined by:

$$C_{comm} = t_0 n + \frac{m}{r_\infty n},$$

where

- t_0 is the start up time or latency,
- n is the number of nodes involved in the communication,
- r_∞ is the asymptotic bandwidth, which is defined as the maximal bandwidth achievable when the message length approaches infinity,
- m denotes the number of bytes of data to be transmitted.

4.3.2 Workload Parameters

A wide range of representative applications is encompassed in Sevcik's execution time function [24]. In addition, it allows for overheads in a parallel computation when characterizing the jobs

$$T(p) = \phi \frac{W}{p} + \alpha + \beta p,$$

where

- ϕ is an indication of the load imbalance in the computation,
- W is the cumulative service demand for the job,
- p is the number of processors allocated to the job,
- α represents the amount of sequential computation,
- β represents the amount of communication and congestion overhead that increases with the number of processors.

Note that $\frac{W}{p}$ represents the ideal division of an application's basic work across p processors. By choosing different values for W, ϕ, α, and β, we obtain jobs with different characteristics and inherent structures. For our experiments, we choose ϕ to be 1, as measurements performed by Wu indicated that in most cases it can be treated as a constant never exceeding 1.14 [26]. For the value of W, we use the values obtained from the executions of the NAS parallel benchmarks [5], as well as the statistics gathered at various centers [13], [15], [16]. W is generated using a 2-stage hyperexponential distribution as in typical simulation experiments [6], [12], [1], [2]. One set of the values consisted of jobs with mean service demand 16.36 and coefficient of variation (CV) 10.

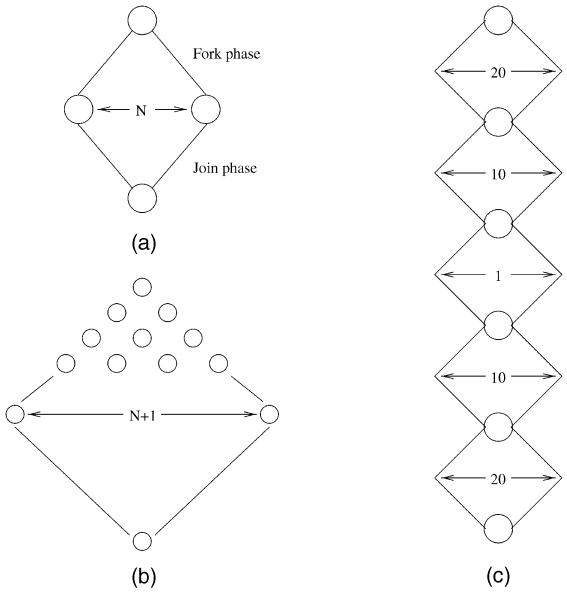


Fig. 3. Application structure as represented by the three workloads WK1, WK2, and WK3.

Note that we are accounting for the communication delays and parallelization overheads while computing every job's execution time function. The communication and congestion overhead β is derived from the job's maximum parallelism p_{max} and can be calculated from

$$p_{max} = \begin{cases} \infty & \text{if } \beta = 0 \\ \sqrt{\frac{W}{\beta}} & \text{otherwise.} \end{cases}$$

Since Feitelson and Nitzberg observed uniform distribution of jobs with different sizes, we uniformly generated jobs with parallelism, p_{max}, ranging between 1 and 64. Given W and p_{max}, β can be obtained from the last equation as

$$\beta = \frac{W}{p_{max}^2}.$$

Based on the observations by Anastasiadis [1], the parallelism overhead α is given by,

$$\alpha = W\left(\frac{1}{p_{max}^2}\right)^{\mu}$$

for some $\mu \in R \cup \{+\infty\}$. Based on these parameters, we used four workloads in our experiments, as defined below. The characteristic structure of applications represented by the workloads are illustrated in Fig. 3 and are derived as in [1].

TABLE 1
Load and System Parameters Used in Our Experiments

Parameter	Description of values
Processors P	64 homogeneous processors
Execution time T(p)	$\phi(p)\frac{W}{p} + \alpha + \beta p$
Essential work W	$\mathrm{F(W)} = 0.125*(1 - e^{-\frac{W}{101}}) + 0.875*(1 - e^{-\frac{W}{1.3}})$
	$\mathrm{E(W)} = 16.36$ and $\mathrm{CoV} = 10.0$
Maximum Parallelism p_{max}	Uniform between [1, 64]
Parameter β	$\frac{W}{p_{max}^2}$
Parameter α	$\mathrm{W}(\frac{1}{p_{max}^2})^{\mu}$
Workload 1	$\mu = +\infty$; $\mathrm{E}(\alpha) = 0.0$; $\mathrm{E(T(1))} = 14.06$
Workload 2	$\mu = 0.4$; $\mathrm{E}(\alpha) = 2.0$; $\mathrm{E(T(1))} = 16.06$
Workload 3	$\mu = 0.2$; $\mathrm{E}(\alpha) = 5.0$; $\mathrm{E(T(1))} = 19.06$
Workload 4	$\mu \in \{0.2, 0.4, +\infty\}$; $\mathrm{P}(0.2) = \mathrm{P}(0.4) = \mathrm{P}(\infty) = \frac{1}{3}$
Inter arrival Time	$f(t) = \lambda e^{-\lambda t}$ with $\frac{1}{\lambda} = \frac{E(T(1))}{\text{Processors} * \text{Load}}$
Pt.-to-Pt. communication delay	$t = t_0 + \frac{m}{r_\infty}$
Scheduling overhead	1% of the execution time
Preemption overhead	10 μsecs
Transfer factor Tr	1
Branching Factor	4
MPL	2
Total # of jobs/batch	10000

1. Workload WK1 consists of curves with relatively good speedup as indicated by p_{max}. They correspond to $\mu \to +\infty$. Matrix multiplications type of applications fall under this category.
2. Workload WK2 was derived with $\mu = 0.4$ and indicated speedup not as good as WK1. Applications addressing gravitational interactions of planets are representative examples of this.
3. Workload WK3 represented jobs with very poor speedup obtained with $\mu = 0.2$. Mean value analysis kinds of applications exhibit this property.
4. Workload WK4 was derived out of an equiprobability combination of the job mix consisting of WK1, WK2, and WK3.

Our workload characterization is based on realistic and significant observations made on the workload at a number of high performance computing centers. Table 1 summarizes all the system and load parameters together with their values as used in our simulation experiments.

5 Performance Analysis

In this section, we present the results of the simulation experiments. We use the mean response time as the performance metric. Unless otherwise specified, the default parameters given in Table 1 are used. For the RRTS and HSP policies, MPL and context switch (preemption) overhead given in this table are used. In addition, for the RRTS policy, average quantum-per-task q is 50 ms (that is, for a job with about 32 tasks).

We do not simulate jobs with partially ordered tasks. However, the synthetic model described in Section 4.3 captures enough details of these job structures (shown in Fig. 3) from the performance point of view. In addition, our previous work on scheduling that simulated four different job structures showed that a simple fork-and-join job structure can be used to compare various scheduling policies [4], [11]. The basic observation from these two studies was that, while the absolute performance is dependent on the job structure, we get the correct relative performance when comparing performance of two scheduling policies. We have presented the results of a simple fork-and-join workload model in [10].

5.1 Relative Performance under Workload WK1

Fig. 4 shows the relative performance for mean service demand 16.36 and service demand CV of 10. Note that WK1 represents the best case speedup among the applications. At low loads, MAP2 tends to perform better than the other two policies. This is because RRTS and HSP incur a penalty due to the MPL of 2. This leads to situations where some processors will have two tasks to execute while others may be idling. The same is true with HSP as well. In addition, due to its self-scheduling nature, HSP incurs additional overhead during task transfers. During the experiments, it was noted that the penalties due to context switching in RRTS and HSP were negligible. HSP performs better than

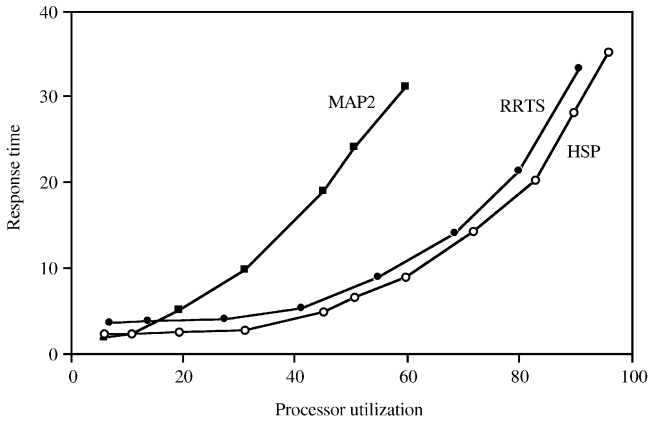


Fig. 4. Response time versus utilization for WK1 with overheads.

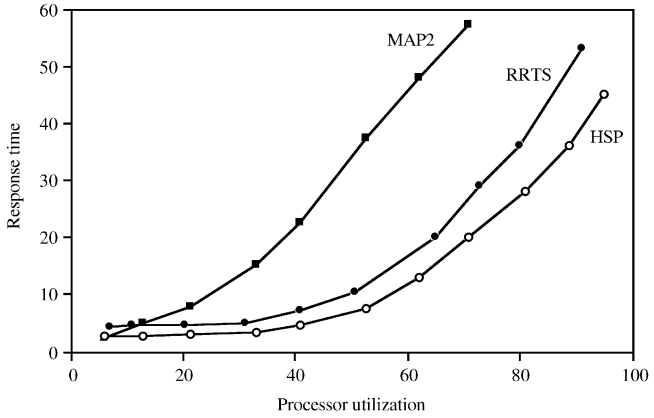


Fig. 5. Response time versus utilization for WK2 with overheads.

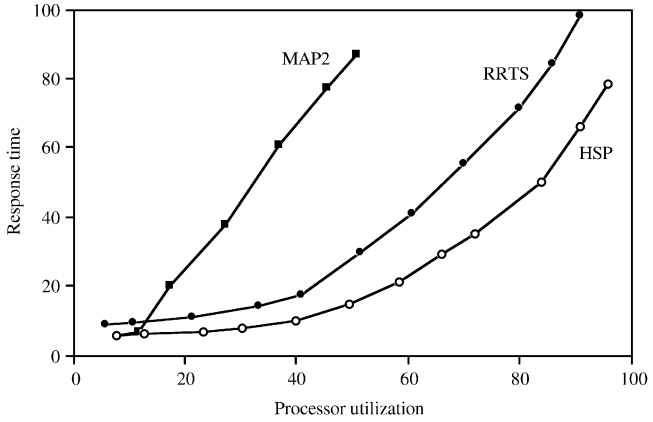


Fig. 6. Response time versus utilization for WK3 with overheads.

RRTS as the space-sharing nature of HSP induces less overhead.

At moderate to high system loads, HSP outperforms the other two. Its hybrid nature is able to transfer work along the appropriate branch at the root much more efficiently at these loads. The primary reason for the poor performance of RRTS is the high access contention for the central queue. At these loads, even with $\mathrm{MPL} = 2$, it behaves more or less like a first-come first-serve (FCFS) policy. Previous studies indicate similar performance for RRTS at these loads [6], [7], [12]. The performance of space-sharing suffers with increasing system load due to its run-to-completion and FCFS nature. It is a well-known fact that FCFS suffers when there is high variance in service demand, which is the case in this experiment.

5.2 Relative Performance under Workload WK2

The relative performance of the policies is shown in Fig. 5 for workload WK2. Note that WK2 exhibits slightly inferior speedup compared to WK1. Due to this, the performance of the three policies is relatively inferior than for the case of WK1. The trend of each individual policy is as before. In addition to the relative performance, robustness is an equally important requirement. Robustness is a measure of the stability of any policy under varying workload considerations [22]. Based on the performance of HSP under WK1 and WK2, we can conclude that it is indeed robust.

5.3 Relative Performance under Workload WK3

Workload WK3 is by far the worst application in terms of speedup. The behavior of the three policies is shown in Fig. 6. Similar to the previous instance, we notice quick deterioration of MAP2 as against RRTS and HSP. The robustness of HSP is again evident.

The performance degradation of MAP2 is quite rapid in this case. This is because of the high arrival CV of 10 that we are using together with the fragmentation. The average size of the partition allocated to jobs ranged between 16 and 9 until about 40 percent of the system load. This induces a lot of queuing and overhead and, hence, the rapid performance degradation. At about 50 percent, the partition size was 4. This means that, at most, six jobs were scheduled on the system. In fact, this is the cause for the enormous queuing delays, which leads to total performance degradation.

5.4 Relative Performance under Workload Combinations

To obtain a better understanding about the policies under real workloads, we evaluated them under a combination of WK1, WK2, and WK3. We assumed that these would occur with equal probability during generation. Fig. 7 illustrates the relative performance of the three policies, which is similar to the previous cases.

The space-sharing policy shows marked improvement in comparison to its performance under WK2 and WK3. This is due to the presence of jobs belonging to WK1 in the workload mix. This can be further generalized to mean that individual workload types have a deterministic influence on scheduling policies even when occurring in a combination of heterogeneous workloads. In fact, this drives home the point that performance of any scheduling policy has to be studied under workload mixes. Thus, our motivation to consider them has been justified at this point. A similar effect has been observed due to multiple workloads in the realm of network of workstations as well [3]. Additionally, in our case, observations of the performance of RRTS and MAP2 suggest that single schedulers are a major source of bottlenecking for real instances as against HTQ type organization.

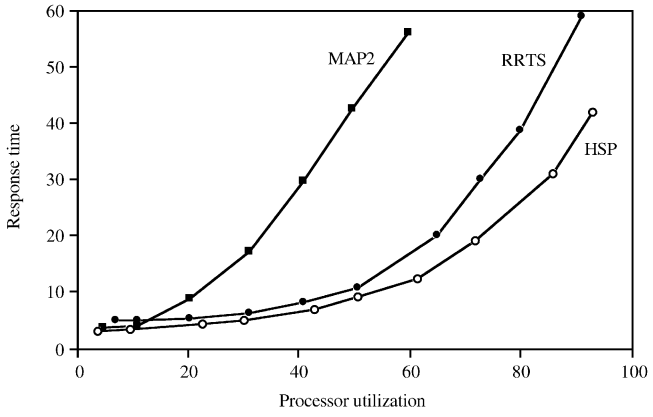


Fig. 7. Response time versus utilization for workload combinations.

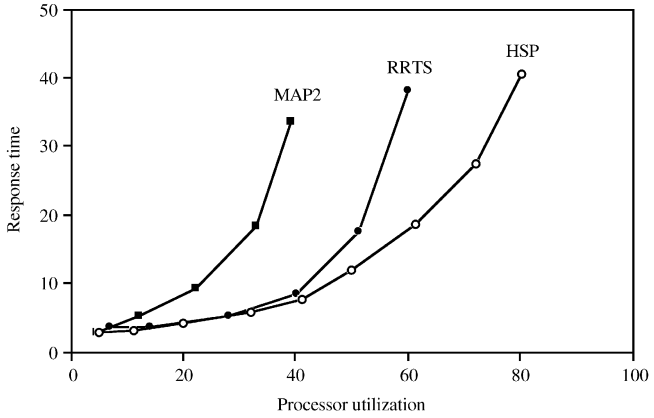


Fig. 8. Response time versus utilization for high variance in service demand.

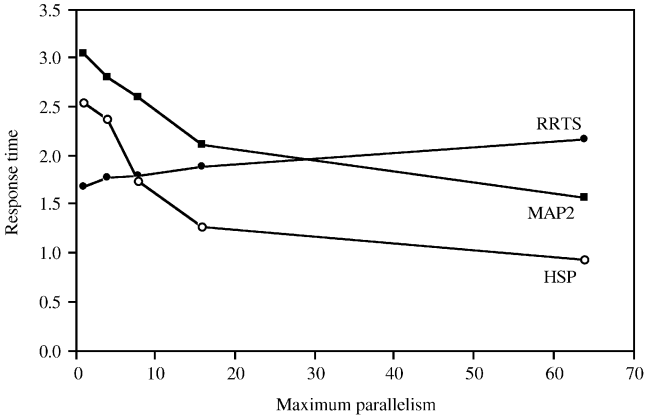


Fig. 9. Effect of variation of maximum parallelism at low loads.

5.5 Considerations for High Variance in Service Demands

Many recent studies have found that the CV of demand in high performance computing centers is very high, ranging from 3 to even 40 [6], [13], [21]. Fig. 8 shows the performance of the three policies as a function of system utilization when the service demand variance is 15.

We limited the demand to 15 so as to include MAP2 for analysis. We present the results just for WK3 as it by far exhibits the worst speedup. Due to this, the scheduling policies will be tested thoroughly under the worst case. The impact of drastically changing service demands can be gauged by comparing the results of this section with Fig. 6, wherein the variance was fixed at 10. From these two figures, we notice that there is definite performance deterioration in the three policies. However, MAP2 seems to suffer the most as it is very sensitive to clustered arrivals. This is further demonstrated in Fig. 8. The drastic degradation in the performance can be attributed to the inherent nature of MAP2, which uses run-to-completion after scheduling jobs in a strictly FCFS manner.

An additional point here is that, in comparison to HSP, the performance of both RRTS and MAP2 deteriorates dramatically at about 30 percent and 40 percent utilization, respectively. This can be attributed to the queue access contention in these disciplines leading to the FCFS situation. As job mixes at these high CVs consist of many small jobs with lesser service demands, as well as a few large jobs with very high demands, the access pattern to the queue is "bursty" though the time averaged behavior does not change much. HTQ overcomes these problems due to its structure and organization and has the least sensitive overall values.

5.6 Considerations for Parallelism Variation

To study the effect of parallelism, we compared the three schemes under the mixed workload model on the basis of limiting the maximum parallelism of the jobs. The results at low loads are shown in Fig. 9. From this, we notice that, at low parallelism, HSP and MAP2 perform worse than RRTS. The behavior of MAP2 is bad because of extreme fragmentation problems. For instance, during the experiments, we observed that on the average there were three jobs in the input queue at any given time. Assuming all of them arrived at the same time, each of them would be scheduled on a partition size of 21. Of these, up to 20 processors could potentially be idle in each partition. HSP's performance is better than MAP2 but worse than RRTS due to the additional cost of self-scheduling. However, this cost is minimal and is discernible only at low loads.

For increasing parallelism, we notice that hierarchy performs better than space-sharing and time-sharing. For these parallelism values, all the processors in a partition will be busy under MAP2. In this case, the only cost a job accrues is due to waiting if the system is already occupied by the preceding jobs. In the case of timesharing, multiprogramming induces additional costs and thus degrades its performance. As far as hierarchy is concerned, its enhanced performance is due to the adaptive nature of the transfer process. In the presence of a few jobs with large maximum parallelism, due to the task transfer process initiated at the root and with the MPL fixed at 2, every individual processor's requirement is better satisfied.

Based on observations at a number of high performance computing centers, it can be argued that the percent of sequential and very small jobs is but a very miniscule portion of the total workload [13], [16], [15]. Therefore, we conclude that HSP exhibits the best performance overall at low loads. The results for the case of high loads are illustrated in Fig. 10.

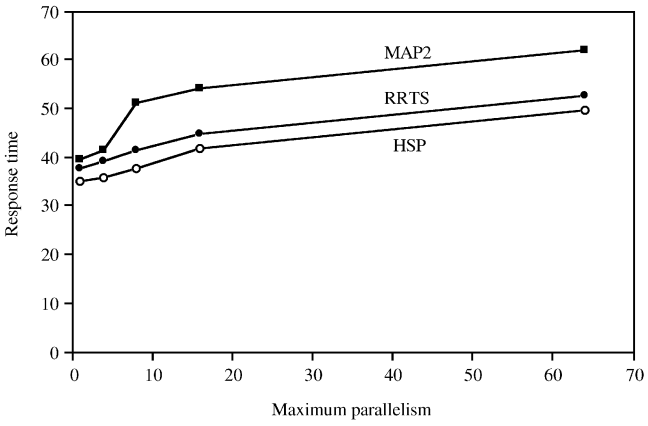


Fig. 10. Effect of variation of maximum parallelism at high loads.

In this case, MAP2's performance is the worst. As it tends to allocate smaller partitions at very high loads, the overheads arising in trying to pack jobs with very large maximum parallelism into these partitions is exorbitantly high. Time-sharing and hierarchy perform much better than this. Among them, RRTS performance is bad mainly because of access contention costs at these loads. Moreover, the absence of the same is the reason for the superior performance of HSP. Based on the observations, one can naturally conclude that the hierarchical policy provides the best performance gains under a plethora of conditions.

6 Conclusions

In this paper, we proposed the Hierarchical Scheduling Policy as a new scheme for scheduling in distributed-memory multicomputer systems. This policy combines space partitioning and time-sharing to schedule workloads within a hierarchical task queue organization. Based on recent observations of workloads at a number of high performance computing centers, we compared the HSP with a pure space- and time-sharing policy. Detailed simulation results under various realistic considerations indicate superior performance of HSP over a range of workloads. Reasons for this are:

1. HTQ offers a natural means for obviating contention for any one queue,
2. HSP does not use a "all-or-nothing" allocation policy,
3. HSP, together with HTQ, offers a solution for overcoming the fragmentation problem typically arising with pure space-sharing policies,
4. HSP time shares within an adaptively allocated partition (that is, it allows for partial allocation of processors), and
5. HSP is a quasi demand driven policy.

The combined effect of these characteristics is perceived in observations of lower response times, high utilization, as well as great degree of robustness.

Acknowledgments

The authors gratefully acknowledge the financial support provided by the National Sciences and Engineering Research Council and Carleton University. This work was done while Thyagaraj Thanalapati was at the School of Computer Science, Carleton University.

References

[1] S.V. Anastasiadis, "Parallel Application Scheduling on Networks of Workstations," Technical Report 342, Computer Systems Research Inst., Univ. of Toronto, Canada, 1996.

[2] S.V. Anastasiadis and K.C. Sevcik, "Parallel Application Scheduling on Networks of Workstations," *J. Parallel and Distributed Computing,* vol. 43, no. 2, pp. 109-124, June 1997.

[3] R. Arpaci, A. Vahdat, T.E. Anderson, and D.A. Patterson, "Combining Parallel and Sequential Workloads on Networks of Workstations," Technical Report 342, Univ. of California, Berkeley, 1994.

[4] S.L. Au and S.P. Dandamudi, "The Impact of Program Structure on the Performance of Scheduling Policies in Multiprocessor Systems," *J. Computers and Their Applications,* vol. 3, no. 1, pp. 17-30, Apr. 1996.

[5] D. Bailey, J. Barton, T. Lasinski, and H. Simon, "The NAS Parallel Benchmarks," Technical Report RNR-94-007, NASA Ames Research Center, 1994.

[6] S.-H. Chiang, R.K. Mansharmani, and M.K. Vernon, "Use of Application Characteristics and Limited Preemption for Run-to-Completion Parallel Processor Scheduling Policies," *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems,* pp. 33-44, May 1994.

[7] S.P. Dandamudi and P.S.P. Cheng, "A Hierarchical Task Queue Organization for Shared-Memory Multiprocessor Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 6, no. 1, pp.1-16, Jan. 1995.

[8] S.P. Dandamudi and M. Lo, "A Hierarchical Load Sharing Policy for Distributed Systems," *Proc. IEEE MASCOTS,* pp. 3-10, 1997.

[9] S.P. Dandamudi, "Reducing Run Queue Contention in Shared Memory Multiprocessors," *Computer,* vol. 30, no. 3, pp. 82-89, Mar. 1997.

[10] S.P. Dandamudi and T.K. Thyagaraj, "A Hierarchical Processor Scheduling Policy for Distributed Memory Multicomputer Systems," *Proc. IEEE Int'l Conf. High Performance Computing,* 1997.

[11] S.P. Dandamudi and H. Yu, "Performance of Adaptive Space Sharing Processor Allocation Policies for Distributed-Memory Multicomputers," *J. Parallel and Distributed Computing,* vol. 58, pp. 109-125, 1999.

[12] S.P. Dandamudi and S. Ayachi, "Performance of Hierarchical Processor Scheduling in Shared-Memory Multiprocessor Systems," *IEEE Trans. Computers,* vol. 45, no. 10, pp. 1202-1213, Oct. 1996.

[13] D.G. Feitelson and W. Nitzberg, "Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860," *Job Scheduling Strategies for Parallel Processing,* D.G. Feitelson and L. Rudolph, eds., pp. 337-360, Springer-Verlag, 1995.

[14] D.G. Feitelson and L. Rudolph, "Distributed Hierarchical Control for Parallel Processing," *Computer,* vol. 23, no. 5, pp. 65-77, May 1990.

[15] S. Hotovy, "Workload Evolution on the Cornell Theory Center IBM SP2," *Job Scheduling Strategies for Parallel Processing,* D.G. Feitelson and L. Rudolph, eds., pp. 27-40, Springer-Verlag, 1996.

[16] J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riordan, "Modeling of Workload in MPPs," *Proc. Third Workshop Job Scheduling Strategies for Parallel Processing,* pp. 95-116, 1997.

[17] S.T. Leutenegger and M.K. Vernon, "The Performance of Multiprogrammed Multiprocessing Scheduling Policies," *Proc. ACM SIGMETRICS Conf.,* pp. 226-236, May 1990.

[18] C. McCann and J. Zahorjan, "Processor Allocation Policies for Message-Passing Parallel Computers," *Proc. ACM SIGMETRICS Conf.,* pp. 19-32, May 1994.

[19] P.K. McKinley, Y.-J. Tsai, and D. Robinson, "Collective Communication in Wormhole-Routed Massively Parallel Computers," *Computer,* vol. 28, no. 12, pp. 39-50, Dec. 1995.

[20] J.K. Ousterhout, "Scheduling Techniques for Concurrent Systems," *Proc. Third Int'l Conf. Distributed Computing Systems,* pp. 22-30, Oct. 1982.

[21] E.W. Parsons, K.C. Sevcik, "Multiprocessor Scheduling for High-Variability Service Time Distributions," *Job Scheduling Strategies for Parallel Processing,* D.G. Feitelson and L. Rudolph, eds., pp. 127-145, Springer-Verlag, 1995.

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

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

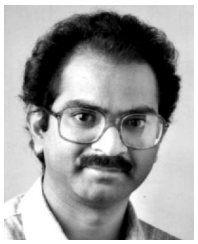
[24] K.C. Sevcik, "Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems," *Performance Evaluation,* vol. 19, pp. 107-140, 1994.

[25] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," *Proc. 19th Ann. Symp. Computer Architecture,* pp. 256-267, May 1992.

[26] C.-S. Wu, "Processor Scheduling in Multiprogrammed Shared Memory NUMA Multiprocessors," Technical Report 341, Computer Systems Research Inst., Univ. of Toronto, Canada, 1993.
[27] Z. Xu and K. Hwang, "Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2," *IEEE Parallel and Distributed Technology*, pp. 9-23, Spring 1996.
[28] S. Zhou and T. Brecht, "Processor Pool-Based Scheduling for Large-Scale NUMA Multiprocessors," *Proc. ACM SIGMETRICS Conf.*, pp. 133-142, May 1991.



Thyagaraj Thanalapati received the BSc degree from Bangalore University, India, and the MSc degree from Mysore University, India. He received the MCS degree in computer science from Carleton University, Ottawa, Canada, in 1997. He has been working at IBM, Toronto, since 1997. His research interests include parallel systems and scheduling.



Sivarama P. Dandamudi received the BE degree from the University of Myore, India, and the MTech degree in electrical engineering from the Indian Institute of Technology, Madras, India. He received the MSc and PhD degrees in computer science from the University of Saskatchewan, Saskatoon, Canada, in 1984 and 1988, respectively. He is currently a professor in the School of Computer Science at Carleton University. He is the director of the Centre for Parallel and Distributed Computing, School of Computer Science. He is the author of the books *Hierarchical Hypercube Multicomputer Interconnction Networks* (Ellis Horwood, 1991) and *Introduction to Assembly Language Programming* (Springer, 1998). His next book on computer organization and design will be published in 2001/2002 (Springer). His research interests include parallel systems, distributed systems, database systems, and performance evaluation (See www.scs.carleton.ca/~sivarama). He is a memeber of the IEEE and the IEEE Computer Society.

▷ **For further information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.