Performance Comparison of Processor Scheduling Strategies in a Distributed-Memory Multicomputer System *

Yuet-Ning Chan, Sivarama P. Dandamudi
School of Computer Science
Carleton University
Ottawa, Ontario K1S 5B6, Canada

Shikharesh Majumdar
Dept. of Systems and Computer Engineering
Carleton University
Ottawa, Ontario K1S 5B6, Canada

TECHNICAL REPORT TR-97-1†

Abstract — Processor scheduling policies on systems that run multiple applications simultaneously can be broadly divided into space-sharing and time-sharing policies. Space-sharing policies partition the system processors and each partition is allocated exclusively to a job. In time-sharing policies, processors are temporally shared by jobs (e.g., in a round robin fashion). Space-sharing and time-sharing policies have their advantages and disadvantages. It has also been suggested that a hybrid policy that combines space-sharing and time-sharing is beneficial in improving the overall performance. Processor scheduling has received considerable attention in the context of shared-memory multiprocessor systems but has not received as much attention in distributed-memory multicomputer systems. Furthermore, most previous research in this area has either used a simulation model or an analytical model in evaluating the performance of various policies. Very often these models neglect several practical aspects of the system and workload.

Our goal is to implement processor scheduling policies on a real system and evaluate their performance. We have used a Transputer-based system to implement two policies — one space-sharing policy and one time-sharing policy — in order to study the relative performance trade-offs involved. We have also implemented a hybrid space-sharing and time-sharing policy. We report the performance sensitivity of these policies to various system and workload parameters. These include three types of applications, four types of interconnection networks, two kinds of software architecture. An aspect, which is not present in most previous studies, is that our study reports the impact of such practical issues as the contention for communication links, network topology, contention for memory, memory size limitation, and software architecture on the performance of the scheduling policies.

Key words: Distributed-memory systems, Processor scheduling, Performance evaluation, Space-sharing, Time-sharing.

1 Introduction

Parallel processing systems can be broadly divided into two basic architectures: *shared-memory* and *distributed-memory* architectures. In shared-memory systems (often referred to as multiprocessors) the system memory is shared by all processors. This shared memory is the medium by which processors communicate. Some examples of multiprocessors are the BBN Butterfly, Sequent Balance, IBM RP3, the Stanford DASH, and the Illinois Cedar.

In distributed-memory systems (often referred to as multicomputers), on the other hand, each processor has its own local memory. While each processor can access its own memory, accessing other processor's local memory is not possible. As a result, these systems communicate by means of message passing. Parallel machines from Intel and Ncube, and machines based on Transputers are all examples of commercially available distributed-memory systems.

Both types of architectures have their advantages and disadvantages. The shared-memory systems present a single address space abstraction to the programmer while the distributed-memory architecture lacks this desirable feature. On the other hand, shared-memory systems are limited to small to medium size systems whereas the distributed-memory architecture is suitable for large systems. Our focus here is on the distributed-memory systems.

Processor scheduling policies can be broadly divided 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 (e.g., in a round robin fashion). Each type of policy has its advantages and disadvantages. For example, space-sharing policies utilize

* A preliminary version of this paper appears in *Proc. Int. Parallel Processing Symposium*, Switzerland, April 1997.

† This report can be obtained from http://www.scs.carleton.ca

the processors poorly when there is high variance in job service times. Time-sharing policies, on the other hand, introduce context switch overhead which is not present in space-sharing policies. Systems like the Intel iPSC/860 support space-sharing. In this system, a job would have to first get a certain number of nodes (a power of 2) allocated to the job before running it. Such under-utilization of processors can be reduced if time-sharing is implemented. In this case, processors can be switched to another job. There have also been proposals to implement a hybrid policy that combines the space-sharing and time-sharing policies to derive benefit from the positive aspect of both policies [22]. For an overview of this topic, see [26].

Processor scheduling has been studied extensively in shared-memory architectures. There does not exist as much literature in the distributed-memory area. Our goal is to study the performance implications of the three types of processor scheduling policies in distributed-memory systems. Our objective is to study the relative performance of these policies on a real system and how it is affected by various system and workload parameters.

Model-based performance evaluation can be done either by analytical techniques or by simulation. Both of these techniques, however, suffer from the fact that they are abstractions and often, for simplicity and tractability, omit several practical issues. Most previous studies have used one of these (or a combination of the two) modeling techniques to evaluate the performance of scheduling policies in distributed-memory systems. In contrast to previous studies, we have chosen the route of implementation on a real system to evaluate the performance.

We have implemented three scheduling policies — a space-sharing policy, a time-sharing policy and a hybrid policy — on a 16-node Transputer system. We present results that report the relative performance of these three policies. Since our results are based on implementation, we can provide insights into the system behaviour that includes the effect of various system overheads that were not possible in other studies based on simulation or analytical modeling. We have used three applications — a matrix multiplication application which involves low communication between processes, a sorting application that involves high communication among processes, and a generic application that gives control to vary computation and communication as required. This generic application facilitates performing experiments to answer "what if" questions.

We have also conducted experiments to investigate the impact of the interconnection network on the performance of the scheduling policies. Four interconnection networks are used — linear array, ring, mesh, and hypercube. Note that the communication systems from Intel are based on the hypercube and mesh networks while those from Ncube are based on the hypercube network.

We also consider two types of software architectures — fixed and adaptive. In the fixed software architecture, the program structure and its parameters are set at the time of compilation. Thus, it is independent of the actual number of processors allocated to the job, which is known only at run time. For example, we might write a sort program that divides the main sort job into F processes. We assume that a job consists of a set of processes, each of which with its own thread of control. If the number of processors allocated to such a job is less than F, more than one task is mapped to a processor. The rationale for having such an architecture will become clear later in the paper when we discuss the sort application.

In the adaptive software architecture, the program structure and its parameters such as the number of processes are matched to the number of processors allocated to the job at run time. Many distributed-memory machines support this (e.g., systems from Intel and Ncube). We will elaborate more on this topic in Section 4, which discusses the system workload.

Our results also bring out performance dependence of the policies on communication link congestion, network topology, contention for memory, size of memory etc. The rest of this paper is organized as follows. The next section describes the scheduling policies in detail. Section 3 describes the Transputer environment used in conducting our experiments. The system workload is described in Section 4. This section also describes the two software architectures considered in our experiments. The results are presented in Section 5. The last section concludes the paper and provides some pointers to future work.

2 Processor Scheduling Policies for Multicomputers

It is well-known that, in the realm 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 policy is independent of the the service time variance in jobs whereas the non-preemptive first-come/first- served (FCFS) policy is extremely sensitive to the variance.

Processor scheduling policies designed for multicomputer systems should also have a similar property of sharing the processing power equally among the jobs. However, since there are multiple processors in these systems, processor sharing can be done in one of two basic ways: *spatially* or *temporally*. Policies that belong to the first category are called the *space-sharing* policies in which the system of P processors is partitioned into p 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[1]. In policies using temporal sharing of processors, called *time-sharing* policies, jobs are not given the exclusive use of a set of processors; instead, several jobs share the processors in round robin fashion. There are advantages and disadvantages associated with each type of policies. Next we will discuss these two types of policies in detail.

2.1 Space-Sharing Policies

A typical characteristic of many parallel programs is that the speedup is not linear and there is an optimum number of processors beyond which there is no further gains in speedup. That is, giving more processors reduces a job's execution time initially because of work sharing; however, after allocating a certain number of processors, further allocation of processors only increases the execution time as the overheads dominate the gains obtained from work sharing. These overheads include algorithmic overhead, paralellization overhead, synchronization overhead, and communication overhead [9]. Allowing more applications, running on smaller partitions, will better utilize the system resources. This could potentially improve the mean response time due to the speedup characteristic of parallel programs mentioned before. This is the motivation for proposing space-sharing policies, which is the focus of the next section. Space-sharing policies can be divided into static, semi-static, or dynamic policies. In the following we will give a brief description of each category.

Static Policies

In static policies, the partition size is fixed on a long term basis. For example, partition sizes could be determined based on the expected workload characteristics at the system start-up time. The allocation of jobs to partitions is either specified by the user or scheduled by the system scheduler. These allocations are based on the characteristics of the job such as priority and requirement of resources and whether the allocated job in a partition will run to completion.

Normally, in a system that uses the static policy, a job is inserted into the system ready queue at its arrival and is allocated a partition when there is a suitable partition available for the job. The job gets the exclusive use of the partition and the partition is returned to the free partition pool when the job is completed.

There have been several specific policies proposed for matching jobs with a partition [24]. A simple example of this kind of policy involves dividing the system into equal partitions. The jobs are allocated partitions based on a first-come/first-served (FCFS) basis. Several improvements have also been suggested to this simple policy. There are also proposals based on priority and service demand in making the job allocation decision.

The advantage of this type of policy is that the implementation is simple. If the workload of the system is stable, the system administrator can choose a configuration that best suits the anticipated workload of the system to provide a reasonably good performance. However, a major disadvantage of this type of policy is that it does not adapt to the changes in the system load condition and resource requirements of jobs. As a consequence poor resource utilization may result under certain conditions. Furthermore, such policies also exhibit high sensitivity to variance in service requirements of jobs. The semi- static policies try to avoid such resource under utilization, which are discussed next.

Semi-static Policies

Semi-static policies eliminate the mismatch between a job's processor requirements and the partition size. In these policies, the partition size is determined at the time of allocation. Thus, a match can be achieved eliminating the wastage of resources that is a major drawback of the static policies. However, as in the static policies, once a partition is allocated to a job, it remains fixed during the lifetime of the job. Several semi-static policies have been proposed for distributed-memory systems [5, 10, 15, 21, 22, 23, 25, 26]. As explained in Section 5.1, in our experimental environment there is practically no difference between the semi-static and static policies.

Dynamic Policies

While semi-static policies are better than the static policies, there is still the problem of wasting processing power. For example, if a job is given eight processors, it keeps all eight processors until it is completed. Thus, some processors may be idling during an interval in which the job's parallelism is less than the number of processors allocated. For example, if the serial phase of a job is 20% of the total job execution time, all but one processor are idle for this duration. Changing the number of processors in order to react to a change in execution behaviour of jobs is suggested in [15].

Dynamic space-sharing policies eliminate this problem and address the drawback mentioned before i.e., sensitivity to service time variance. The idea behind dynamic policies is that idle processors should be taken away from a job and allocated to another job that can fruitfully utilize the processor cycles. It can be expected that the performance of the dynamic policies is better than the static and semi-static policies as dynamic policies move processors around by match-

[1] There may be reasons — such as giving priority to a class of jobs — that dictate unequal partition sizes. We will not consider such cases in our study and assume a homogeneous workload.

ing the temporal processor requirements of the jobs. This improvement in performance comes at a cost — the overhead associated with taking a processor away from a job and allocating it to another job. If this overhead is small we can realize performance benefits by using a dynamic policy. This is typically the case for uniform memory access (UMA) shared-memory systems but the overhead is not small for distributed-memory systems. However, dynamic policies have been proposed for distributed-memory systems [7, 17].

2.2 Time-Sharing Policies

A special type of dynamic policies are time-sharing policies that use preemption to rotate processors among the jobs. There are two principal ways in which preemption can be done: *coordinated* or *uncoordinated*. In coordinated preemption, also called coscheduling [19], all processes of a job are given time slice at the same time and all are preempted simultaneously moving the set of processors to the next job. In uncoordinated preemption, on the other hand, each processor is treated independently. Thus, these policies allow preemption of a process from one job while another process of the same job is executing on another processor. Gang scheduling [8] is an example of the coordinated preemptive policy. Coordinated scheduling performs better when the granularity of synchronization[2] is small; otherwise, it performs poorly [2, 13]. Another problem with coordinated preemption is that it is difficult to implement on a distributed-memory system with a large number of processors. For these reasons, we focus on uncoordinated preemptive policies.

A straightforward implementation of the round robin policy used in uniprocessor systems that allocates an equal share of the processing power to each process in the multiprocessing system is not good as the processing power distribution is proportional to the number of precesses in a job [14]. The RR_process policy studied in [14] allocates a fixed quantum size for each process. Recall that the goal is to distribute the processing power uniformly independent of the the number of processes in a job. Thus, a job with a large number of processes (typically implying a large job when there is correlation between the number of processes and job service demand) tends to get more share than a job that is small, which contravenes our basic principle of fairness What is needed is a policy that shares the processing power equally (more or less) among the jobs on a temporal basis. Majumdar et al. [14] have suggested such a policy, called RR_job, and its performance has been reported in [13]. In this policy, the quantum Q is varied depending on the number of processes in the job. For example, assuming that the number of processes in a job T is bounded by the number processors in the system P, the quantum size Q for the processes of a job is calculated as $Q = \frac{P}{T} \times q$, where q is a basic quantum size.

[2] Granularity of scheduling refers to the number of instructions executed by processes between communications [8, 13].

2.3 Hybrid Policy

It has been reported that a combination of space-sharing and time-sharing policy performs better than either one of these policies [3, 22]. In such a hybrid policy, the system is partitioned as in the static policies. However, unlike in the static policies, each partition is not exclusively allocated to a single job; instead, a set of jobs is allocated to a partition. The set of jobs mapped to a partition share the processors in the partition in a round-robin fashion as in the time-sharing policies. The set size (i.e., the number of jobs in the set) is a design parameter.

We have implemented one space-sharing policy, one time-sharing policy and the hybrid policy. A detailed description of the policies implemented is given in Section 5.1.

3 Experimental Environment

3.1 Hardware Description

The Transputer system used consists of sixteen T805 transputers, each with 4MB of local memory and has four communication links. The sixteen processors are hardwired into four pipelines of four processors each of which is called a nap (see Figure 1). The other links can be used to connect to processors in the same nap or in the adjacent nap using INMOS C004 transputer switches. By using these switches, almost all commonly used network topologies can be configured.

We have implemented four topologies: linear array, ring, mesh, and hypercube. The size of each of these networks is varied from 1 to 16 in powers of 2 (i.e., 1, 2, 4, 8, 16). Because one transputer is required to provide a link to the front-end, host workstation a 16 processor hypercube topology is not possible. Note that when the number of processors is 1 or 2, there is no distinction among the four topologies. Similarly, when the network size is 4, ring, mesh and hypercube topologies are equivalent.

The T805 transputer has hardware support for multiprogramming. It supports two priorities by maintaining two ready queues. High priority processes run to completion or until blocked for I/O, IPC, or timer. Low priority processes are time shared and each has a fixed quantum of 2ms. A low priority process must yield control to a high priority process whenever a process in the high priority queue is ready and the unfinished quantum of the preempted low priority process is lost as a result. We have used these ready queues to implement the static and time-sharing policies.

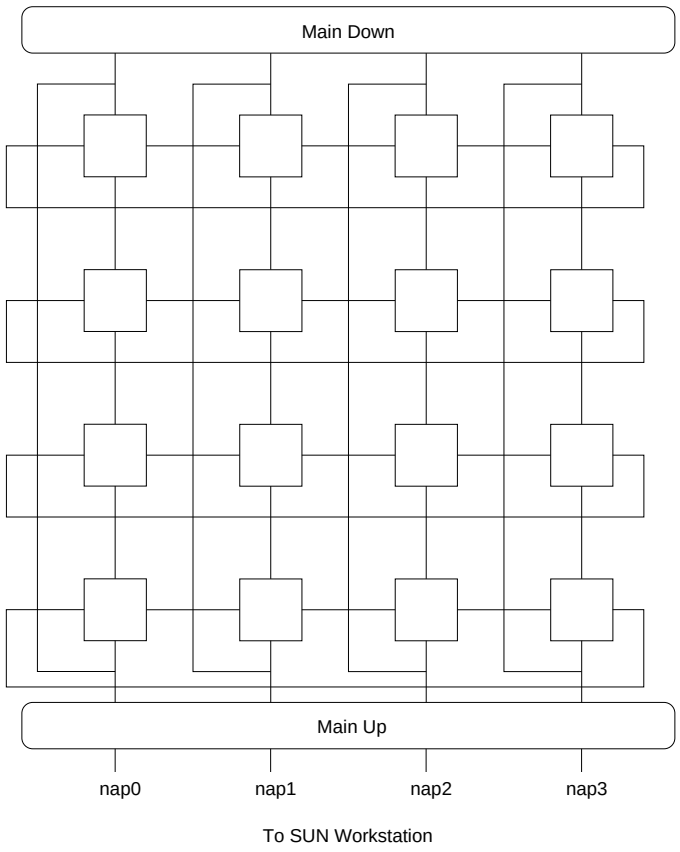

Figure 1: Layout of the Transputer system used in the experiments (from *PARAstation User Manual*, Transtech Parallel Systems, 1992)

3.2 Software Description

We have used a hierarchical approach to implement the scheduling policies. There are three schedulers: a super scheduler, a partition scheduler for each partition in the system, and a local scheduler for each processor. The super scheduler is global and manages the system-wide ready queue. A job submitted to the system is handled by the super scheduler and it is dispatched to one of the partition schedulers. The partition scheduler is responsible for managing the processors in the partition allocated to the job. The partition scheduler then notifies the local schedulers that manage the local ready queue of each processor. Depending on the scheduling policy, the partition scheduler can accept one or more jobs in one partition. Local schedulers support time sharing by using their own preemption control (for details, see [4]).

The transputer system uses a point-to-point communication scheme. In this scheme messages from one processor to another (if they are not adjacent) are required to pass through intermediate processors using store-and-forward switching. The channel package provided by the transputer's software library supports communication only between adjacent processors. We have implemented a mailbox-based asynchronous communication system that facilitates communication between any pair of processors. Because communication is store-and-forward, each intermediate node on path from the source to destination processor must reserve buffers to store the message. A memory management unit (MMU) has been implemented to manage the shared memory in a processor. One of the functions of the MMU, which is present in every processor, is to allocate buffers for message storage (i.e., for mailboxes). Note that a message can suffer delay if an intermediate processor delays allocation of memory for the mailbox. In Section 5 we will show that this has an impact on the performance of the scheduling policies.

4 Experimental Workload

We have used three applications to drive our experiments: matrix multiplication, sort application, and a synthetic generic workload. The next section gives details on these applications. Section 4.4 gives a description of the two

software architectures, fixed and adaptive, used in the experiments.

4.1 Matrix Multiplication

A matrix multiplication application is chosen to represent the programs that exhibit fork-and-join type of structure as shown in Figure 2a. The matrix multiplication application performs multiplication of two matrices. The application can be speeded up by distributing the work to different processors. For example, in a system with 4 processors, each processor is responsible for calculating only 1/4 of the resulting matrix. The application has the software structure shown in Figure 2b in which the fork as well as the join functions are performed by a single coordinator process.

The coordinator process performs multiplication of two matrices A and B by distributing work to worker processes. The coordinator process is also responsible for assembling the parts of the result matrix returned by the worker processes. We want to use this application to represent a workload with low communication among the worker processes. For this reason, we have implemented a specific matrix multiplication algorithm in which the second matrix (matrix B) is distributed to all p worker processes. Matrix A is divided into p parts with each part consisting of R/p rows where R is the number of rows in Matrix A. Each part is sent to a different worker process. Once the required data has been sent to a worker process, it can compute part of the final result matrix without further communication with the coordinator/other worker processes. Note that the coordinator process, after distributing the work, also performs multiplication just like the other worker processes. The shaded bubble in Figure 2b represents a single process that acts as the coordinator process as well as the worker process. Thus, when p worker processes are required, coordinator process creates only $p - 1$ additional worker processes.

4.2 Sorting

Implementation of divide-and-conquer algorithms on parallel system has received significant attention (for example, see [12]). Large numbers of parallel programs have been developed using this strategy. The parallel implementations of divide-and-conquer algorithms exhibits a partitioning structure shown in Figure 3a. The processes created by this type of program perform one of the following three operations: DIVIDE that splits the input, WORK that performs the actual computation desired, and MERGE that combines the output produced by WORK processes.

We have used sorting as a representative example of this important class of applications. The software structure of this application follows the divide-and-conquer structure. Typically, the divide and merge phases are done by a single coordinator process as shown in Figure 3b. For the sorting example, the divide phase involves splitting the array into sub-arrays and sending these sub-arrays to other worker/coordinator processes; the same processes is also responsible for merging the sorted sub-arrays returned to it. As in the matrix multiplication application, a coordinator processes, after performing the divide operation, assumes the role of a worker process and sorts a sub-array. For example, the coordinator process at level 1 also acts as the coordinator at level 2 and the worker process at level 3 (shown lightly shaded in Figure 3b). This example creates a total of four processes (two of which play multiple roles and are shown shaded).

4.3 Generic Workload

The third workload used is a synthetic one. The rationale for using such a model is to exercise control over various workload parameters to answer "what if" questions. The generic workload proposed by Nanda et al. [18] is used to emulate the behaviour of different programs in order to represent various workload patterns. The generic workload generator receives a set of parameters to identify the characteristics of a specific workload and emulates the execution of the workload. The structure of the generic workload is based on the divide-and-conquer model used in the sort application. The important input parameters of this model are: *comp, comm, proc, B*. The parameter *comp* specifies the average time to be spent in computation by each process; the computation time of each process can be different to provide variance among process service times. The parameter *comm* represents the average communication time. The parameter *proc*, which represents the number of processes to be used, can be used to control the number of processes so that both fixed and adaptive software architectures (discussed next) can be implemented. Another parameter — the branching factor B — controls the number of processes that can be forked from one process. For example, in the sort application, we have used a branching factor of 2.

The generic workload has a structure similar to that of the sort application discussed in Section 4.2. Instead of doubling the number of processes at each level, the increase in number of processes depends on the branching factor B such that the number of processes at level i is B^{i-1}. The number of levels n depends both on the number of processes $proc$ and the branching factor B and is equal to $log_B(proc)$.

The computation time of worker and coordinator processes can be specified separately. For example, to mimic the sort application of the last section, we can use $O(n^2)$ for the worker process computation time and $O(n)$ for the coordinator processes, where is n is the array size. More details on the this workload model can be found in [4].

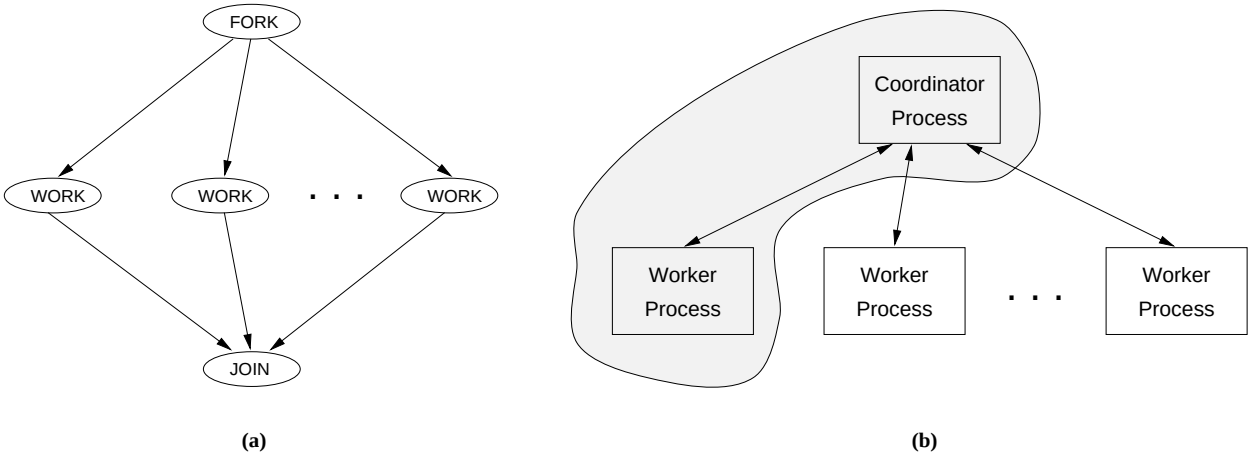

Figure 2: Structure of workload 1: (a) fork-and-join programs; (b) structure of matrix multiplication application (shaded area represents a single process)

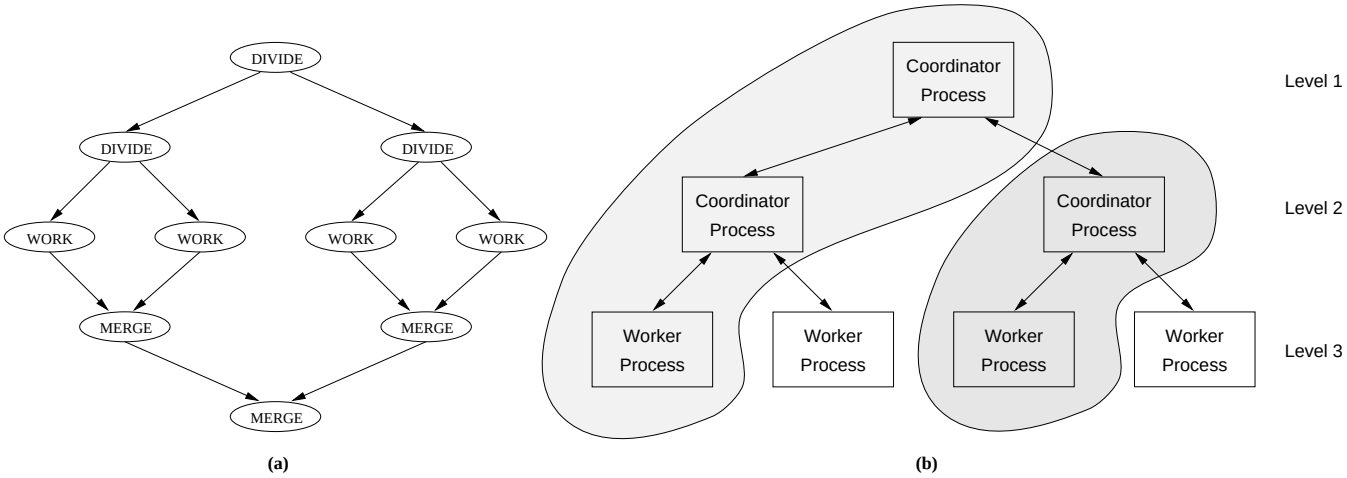

Figure 3: Structure of workload 2 for four worker processes: (a) structure of divide-and-conquer programs; (b) structure of sorting application (shaded areas represent a single process)

4.4 Software Architecture

We consider two types of software architectures in our experiments: fixed and adaptive. In the fixed architecture, the program structure is independent of the number of processors allocated to the job. The number of processes created by the job is determined at the time the program is written or at compilation time. In the adaptive architecture, the number of processes created by the program is dependent on the number of processors allocated to it. Thus the number of processes can be varied by the application at run time.

There are at least three reasons why a software structure would have many more processes than the number of processors allocated to it. First, by using a large number small processes we can achieve better load sharing. This is particularly important in multiprogrammed, time-shared systems. Second, there may be applications for which there is benefit in partitioning the problem into large number of small work units. Sort is an example of such an application. We will elaborate on this in Section 5.3. The third reason is that some applications may not lend themselves well to partitioning at run time. These form the rationale for investigating the fixed software architecture.

In the absence of the above conditions, the adaptive software architecture is beneficial [27]. In the adaptive software architecture, the number of processes created by a job is equal to the number of processors allocated to it. Of course, this information is available only at run time. In order to implement adaptive partitioning of a problem, we need a run time function that can give us the number of processors allocated to it. Such functions are available in current distributed-memory systems (e.g., in Intel and Ncube sys-

tems).

5 Experimental Results and Discussion

This section presents the results of the experiments conducted on a 16-node Transputer system. We use the mean response time as the performance metric. The response time of a job is the waiting time to get processors allocated plus the execution time. Due to lack of space all results are not presented. For details, the reader is referred to [4].

5.1 Implementation Details

We have conducted several experiments to assess the performance of the three policies. We have used a batch of 16 applications. Each batch consists of 12 small jobs and 4 large jobs in order to introduce variance in service times. We use the generic workload to study the impact of higher variances in service times (see Section 5.4). In the fixed software architecture, each job consists of 16 processes; the number of processes in the adaptive architecture is equal to the partition size allocated to the job.

As stated in Section 2, sharing processing power equally implies equal partitions in static and hybrid policies. For this reason, the 16-processor Transputer system is equally partitioned. The number of partitions in the system depends on the partition size. For a partition of size p, the number of partitions in the system is $16/p$. In the static scheduling policy, one job is assigned to each free partition and the job is run to completion. Other jobs wait in a FCFS queue until a partition becomes available. Because of the use of equal partitions and the homogeneous job characteristics, there is no difference between the static and some semi-static policies (such as FCFS, first fit, best fit policies).

A pure time-sharing policy is implemented by considering the whole system as a single partition and all 16 jobs in a batch are assigned to this partition. Thus the multiprogramming level is 16.

In the hybrid policy, the system is partitioned into $16/p$ partitions and all 16 jobs in a batch are distributed equally among the partitions. This automatically determines the multiprogramming level of a partition. For example, when the number of partitions is 2, the multiprogramming level is 8 as eight jobs are allocated to each partition. Notice that the pure time-sharing policy is a special case of the hybrid policy with a single partition. Thus we refer to these two types of policies as time-sharing policies in the following discussion. Since we use equal partitions and jobs in fixed software architecture have 16 processes, there is no difference between RR_process and RR_job policies when there is no variance in service times of processes of a job (which is true in majority of our experiments).

The performance of the static policy is influenced by the order in which the 12 small and 4 large jobs are executed. The reason is that if the large jobs are executed first, the response times of other jobs will increase. To ensure fairness in comparing the performance of static and time-sharing policies, the response time in the static policy is taken as the average of best and worst response times. Note that the best response times are obtained by giving priority to smaller jobs and worst times are obtained by executing the larger jobs first.

5.2 Matrix Multiplication

As stated, each batch consists of 12 large and 4 small jobs. A small job multiplies two 50×50 matrices and a large job multiplies two 100×100 matrices[3]. Two job sizes are used as we want to introduce some degree of variance in job service demand.

Figure 4 shows the result for the fixed software architecture. In this architecture, each job consists of 16 processes. The X-axis represents the partitions size p. The letter following the partition size indicates the interconnection network topology used in each partition. We use letter 'L', 'R', 'M', and 'H' to represent linear array, ring, mesh, and hypercube networks, respectively. For example, label '8L' means that there are 16/8=2 partitions and each partition has 8 processors configured as a liner array. Figure 5 shows the corresponding results when the adaptive software architecture is used. Note that, in adaptive software architecture, the number of processes in a job is equal to the partition size.

We should caution the reader to note that the performance of the pure time-sharing policy is shown in these plots only when the partition size is 16. That is, time-sharing performance corresponds to '16L', '16R' and '16M' TS lines. The other TS lines in these plots actually represent the performance of the hybrid policy. It can be clearly seen from these two plots that the hybrid policy performs much better than the true time-sharing policy. However, it does not do well when compared to the static policy. In the rest of the paper, for convenience sake, we use time-sharing to refer to the performance of the hybrid policy as well.

These results show that time-sharing always performs worse than the static policy for this application. One characteristic of this particular workload is that the variance in job sizes is not high enough to show the time-sharing policy in a better light[4]. We will show in Section 5.4 that when there is higher variance in service demand, time-sharing actually performs better than the static policy. Another significant factor contributing to the performance difference be-

[3] The sizes of the matrices is restricted by the size of the memory available at each processor such that the maximum multiprogramming level of 16 can be achieved when the number of partitions is one.

[4] Recall that the sizes of the matrices is restricted by the size of memory available at each processor.

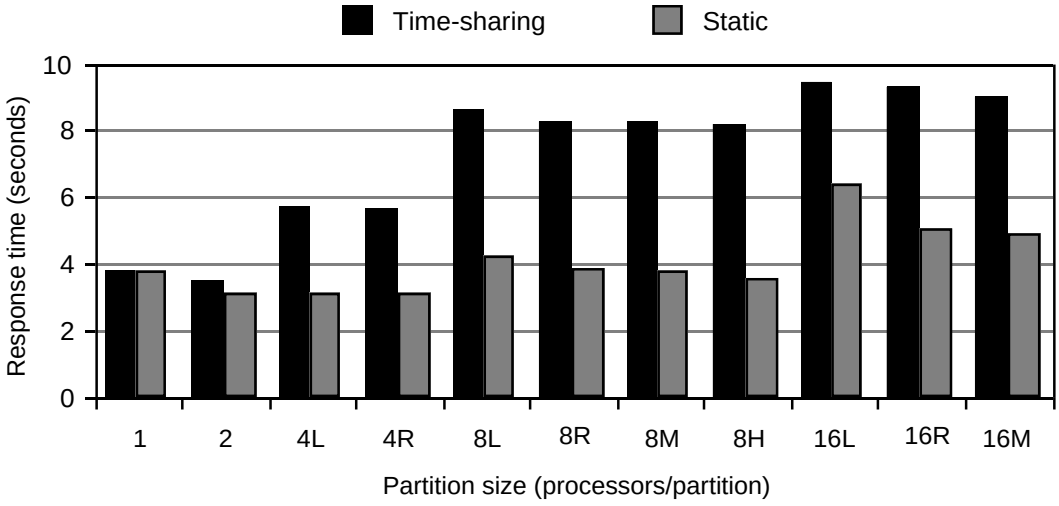

Figure 4: Mean response time for the matrix multiplication application — Fixed software architecture

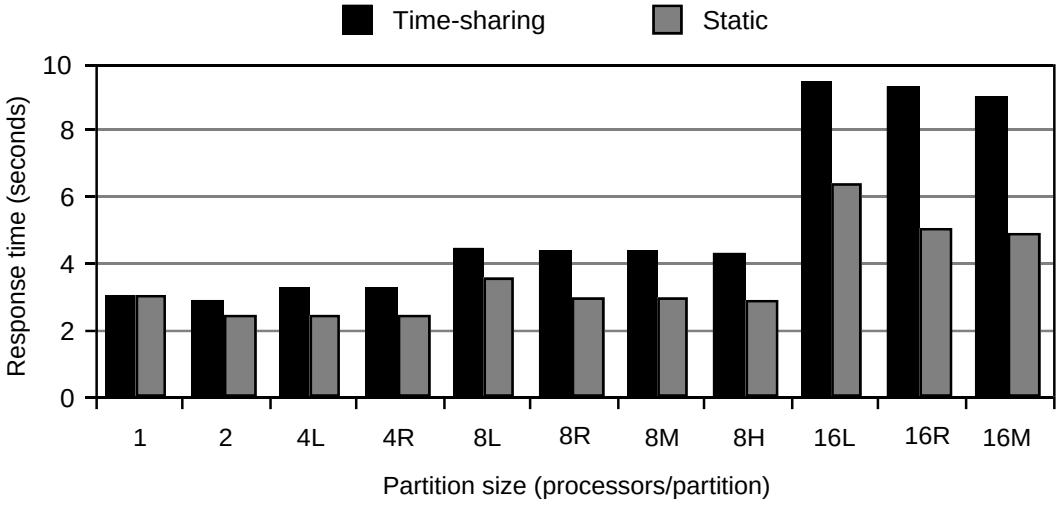

Figure 5: Mean response time for the matrix multiplication application — Adaptive software architecture

tween the static and time-sharing policies is due to the contention for the memory and message congestion. This is because the time-sharing policy loads and starts execution of all 16 jobs whereas the static policy executes only a number of jobs that is equal to the number of partitions in the system. Thus, for our workload, when there are 16 partitions of 1 processor each, both policies behave the same way. In this case, there is no communication and each job runs on a single dedicated processor.

As we increase the partition size, the multiprogramming level in the time-sharing policy increases. At the other extreme, for example, when there is only one partition, the multiprogramming level in time-sharing policy is 16. In such a system, the static policy executes the jobs in a serial fashion by allocation all 16 processors to each job. Since one matrix is distributed to each processor in a partition, time-sharing with high multiprogramming level leads to memory contention. Also, as the multiprogramming level increases, several jobs are simultaneously in partial execution state and possibly exchanging massages. Remember that messages typically go through several intermediate processors where there is contention for communication links; furthermore, since the Transputer uses a store-and-forward switching, there is also demand for memory to store message copies at these intermediate processors. Thus, as the number of partitions decreases (i.e., as we move from left to right in Figure 5), the performance difference between the two types of policies increases.

The fixed architecture exhibits slightly different (in that the difference is also larger when two partitions are used)

due to the presence of a fixed number (16 in our case) of processes. Note that a process sending a message to itself will also go through the same store-and-forward communication mechanism placing demands on memory for buffers etc. Precisely for the same reasons, the adaptive software architecture is better than the fixed architecture for this application. Note that when the number of partitions is one, both software architectures are equivalent and produce the same results (see Figures 4 and 5). For reasons discussed before, the time-sharing policy also exhibits more sensitivity to the type of software architecture.

Performance of both policies is affected by the network topology. Because of the contention and congestion problems associated with increasing the multiprogramming level, the time-sharing policy is more sensitive to the network topology. In particular, the low degree, long diameter networks (as exemplified by the linear network) cause substantial performance deterioration when time-sharing is used. Network topology, however, does not change the relative performance of the two policies. It can be expected that the use of wormhole routing can significantly reduce the need for buffers at intermediate processors. Wormhole routing, by eliminating the need for store-and-forward, can also significantly reduce the performance sensitivity of these policies to the network topology.

5.3 Sorting

We have conducted experiments similar to the ones described for the matrix multiplication application. As in the previous experiments, each batch consists of 16 jobs: 12 small jobs and 4 large jobs. A small job in these experiments sorts 6000 elements while a large job sorts 14000 elements[5]. The results are shown in Figures 6 and 7.

In general, the observations made about the matrix multiplication application also hold for the sort application. However, there is one significant difference between the fixed and variable software architectures that is specific to the sort application. First of all, the software structure of the matrix multiplication is that of fork-and-join structure in which there is one synchronization phase at the end. The software structure of the sort application is divide-and-conquer as discussed in Section 4. This kind of software structure involves work in three phases: a divide phase, a work phase, and a merge phase. The divide and merge phases are done by a single process. For the sorting example, the divide phase involves sending the sub-array to a worker/coordinator process; the same process is also responsible for merging the sorted sub-arrays returned to it. The complexity of this phase is O(n) where is n is the size of the sub-array involved. The worker phase involves sorting the sub-array received. The complexity of this phase can be O(n^2) for sorting algorithms such as insertion sort, selection sort etc. and O($n\ log_2 n$) for merge and heap sort algorithms. We have used the selection sort for sorting the sub-arrays. Therefore, the complexity of the worker phase is much more than that of the merge phase. As a result, if the size of the sub-arrays given to the sort processes is smaller, the execution time decreases substantially. Because of this characteristic, software architecture has a significant impact on the performance.

In the fixed software architecture, the number of processes is fixed at 16 independent of the number of processes allocated to the job. Thus, in this case, each sub-array to be sorted is approximately 1/16 of the original size. On the other hand, in the variable software architecture, the number of processes is equal to the number of processors allocated to the job, which is at most equal to 16 when there is a single partition. Thus the adaptive architecture exhibits substantial speedups. It is also clear from this discussion that fixed architecture is better suited to this type of applications.

5.4 Generic Workload

As indicated in Section 4, the generic workload allows us to vary the various workload parameters to study "what if" type of questions. We have modeled the matrix multiplication and sorting applications by using our generic workload model in order to validate the model. For the sake of brevity, these results are omitted. We have also conducted several experiments using the generic workload model. Due to space restrictions, we will present only a subset of these results here (for details on this, see [4]).

The impact of communication on the performance of the scheduling policies as a function of network topology is shown in Figure 8. The communication factor (CF) is defined as the ratio of communication to computation time [16]. The case CF = 0 represents negligible communication (we call such jobs as "compute-intensive" jobs) and CF = ∞ represents negligible computation time (we call such jobs as "communication-intensive" jobs). From the data presented in these two plots, the following observations can be made:

- When there is little communication, the static policy performs substantially better and this improvement in performance increases with partition size. The difference between the performance of the static and time-sharing policies decreases as the communication among processes increases. This can be explained as follows: A message sent from a source processor to a destination processor typically goes through several intermediate processors. In static scheduling only one job per partition is allowed to be executed at a time and

[5] As in the matrix multiplication application, the sizes of the arrays is restricted by the size of memory available at each processor such that the maximum multiprogramming level of 16 can be achieved when the number of partitions is one.

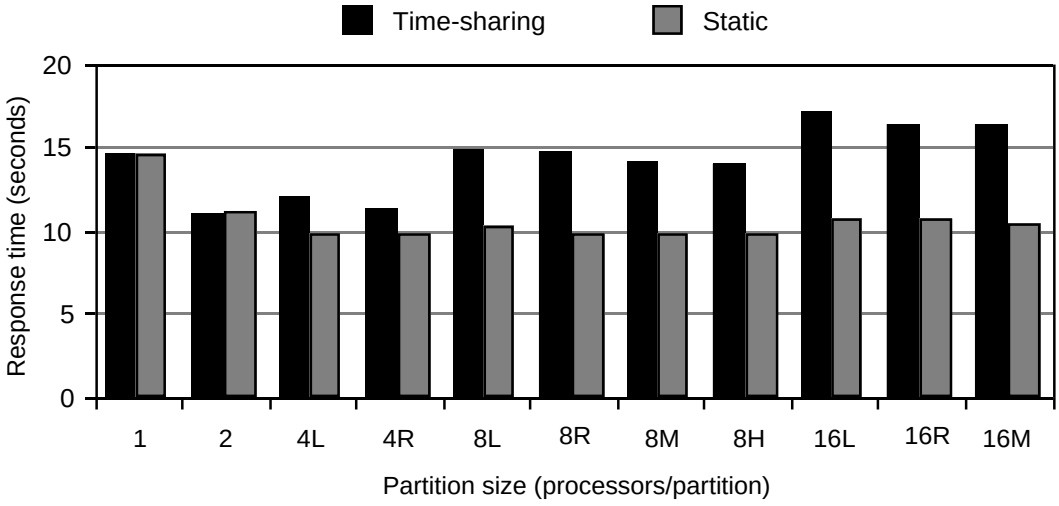

Figure 6: Mean response time for the sort application — Fixed software architecture

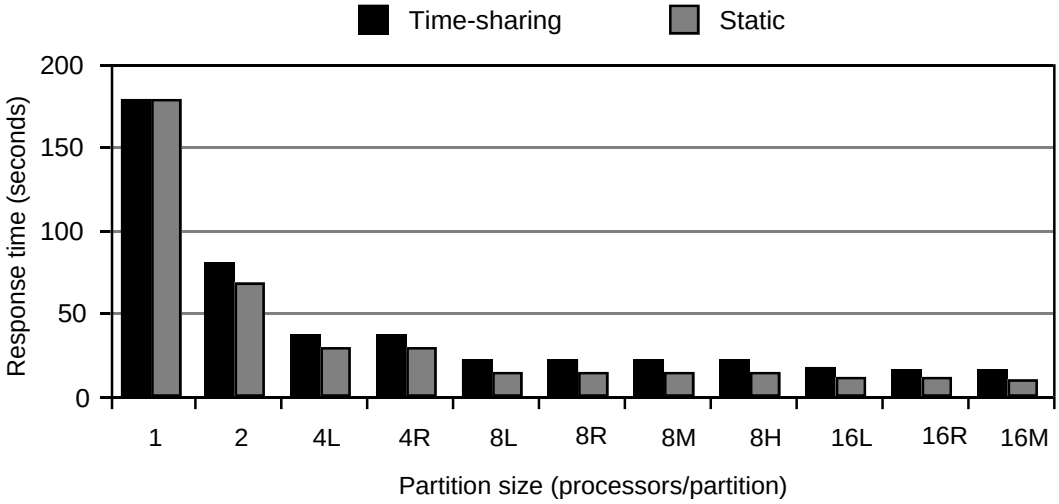

Figure 7: Mean response time for the sort application — Adaptive software architecture

the processors are idle until the communication is completed. In time-sharing, a processor can be switched to another job, which may in turn initiate another message transmission. Thus, time-sharing not only allows efficient utilization of processors but also causes message to be transmitted in a pipelined fashion. Due to these factors, the static policy suffers much more in performance than the time-sharing policy as the communication factor increases.

- When the network topology is linear, the time-sharing policy performs better than the static policy as shown in Figure 8(b). This is because of the message pipeline phenomenon discussed before. Note that, among the four networks considered, the linear array topology presents the maximum average inter-processor distance as measured by number of hops. Therefore, the message pipeline has the most effect with this network topology.

- Another interesting point to note from the data in Figure 8(a) is that the time-sharing performs better than the static when the partition is small (2 or 4). This is so even when there is negligible communication (and therefore not due to the pipeline phenomenon discussed). This is explained as follows. First, the parallelism of the divide-and-conquer type of software structure varies over the lifetime of a job. For example, consider a job allocated to a partition of 16 processors. When the all 16 sort worker processes are active (i..e, during job's maximum parallelism phase), all processors in the partition are busy. But during the

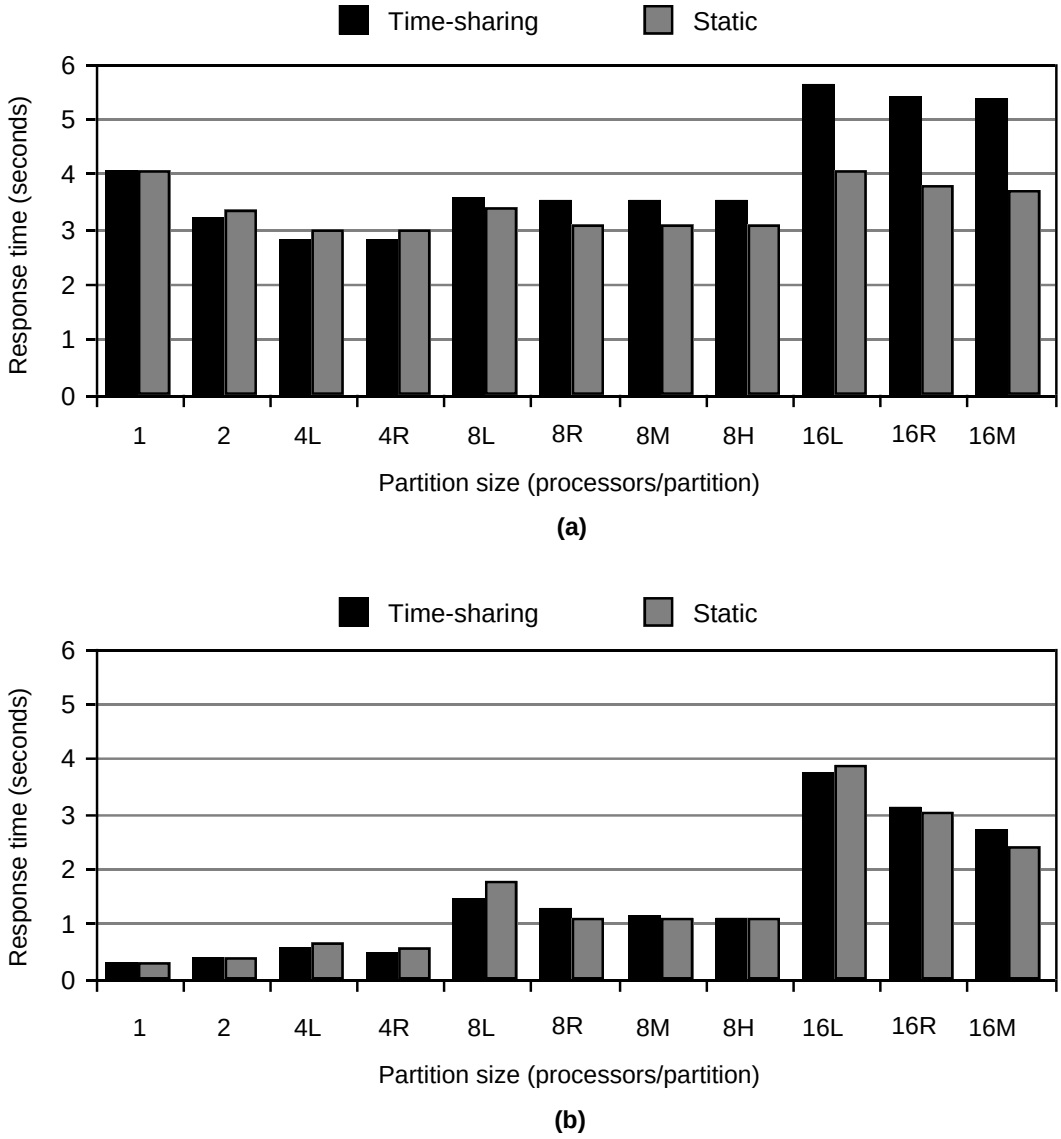

Figure 8: Mean response time for the generic workload for fixed software architecture (a) CF = 0 (b) CF = ∞

next merge phase, only 8 processors are busy merging. If we are using the static policy, the remaining 8 processors are idle until the completion of the job. During the second merge phase, only 4 of the 16 processors are busy leaving an additional 4 processors idle for the remainder of the job execution time. Thus the static policy causes under utilization of processors. In time-sharing, such processor under utilization can be reduced by switching to another job. Second, a larger number of partitions keep the number of jobs per partition small enough such that the overheads and contention introduced by time-sharing is less than the benefit gained from multiprogramming.

- For the data presented in Figures 8, the optimal partition size is 4 for compute-intensive jobs and reduces to 1 when the job are communication intensive. This is the case whether the system uses the time-sharing or static policy. Although not shown here, the optimum partition size is 2 when CF = 1 (i.e., for "balanced" jobs having equal computation and communication times). These results, which are intuitive, indicate that the partition size should be reduced as applications become more communication-dominant.

We conclude this section by presenting the impact of variance in process service times. Figure 9 shows the impact of high process service times (we have introduced a coefficient of variation of 15 in process service times) for bal-

anced jobs (i.e., CF = 1). When the variance is low, the static policy performs better than time-sharing for partition sizes greater than 4 (data for CF = 1 when the service time variance is low are not shown, see [4]). However, when there is high variance[6], time-sharing performs better independent of the partition size and network topology. When there is high variance, the utility of preemption becomes more important for performance.

6 Conclusions

Processor scheduling policies for distributed-memory systems can be broadly divided into either space-sharing and time-sharing policies. In order to study the relative performance of these two types of policies, we have implemented a static space-sharing policy and a round-robin time-sharing policy on a 16-node Transputer system. In addition, we have also implemented a hybrid policy that combines space-sharing and time-sharing. Most previous studies in this area have used either an analytical model or a simulation model. In contrast, we have implemented the policies on a real system in order to evaluate their performance sensitivity to various system and workload parameters. We have used three applications — matrix multiplication, sorting and a synthetic application. For each application two software architectures — fixed and adaptive — were investigated. In addition we have conducted experiments with four types of interconnection network topologies — linear array, ring, mesh, and hypercube. Because our study is based on the implementation of the policies rather than simulation, our results show the sensitivity of the performance to such system factors as the communication link congestion, contention for memory, size of memory etc.

The important conclusions that can be drawn from the data are summarized here.

Effect of memory contention, synchronization, and overheads

- We have shown that limited memory is a serious problem with time-sharing because this policy tends to execute more jobs in a given partition as opposed to a single job per partition in the static policy. Synchronization among different processes in a distributed-memory system is achieved through message passing. Because all jobs share the same partition, pure time-sharing gives rise to performance degradation because of communication buffer and link congestion problems. On the positive side, for time-sharing we have identified a communication pipeline phenomenon that can improve the performance of the time-sharing policies.

[6] Measurements at some supercomputer centers indicate that the service time variance can be very high [1, 20].

- Time-sharing introduces context switching overhead. Because of these and other overheads introduced by time-sharing, for large partition sizes (that is with higher multiprogram level for a partition) a static policy is preferred.

Effect of communication factor

- For a given workload, the optimum partition size tends to decrease as the communication factor increases. For communication-intensive applications, the partition size should be as small as possible provided there are enough jobs to keep the system busy.

Effect of software architecture

- The adaptive software architecture is observed to be better for applications having the following characteristics:

  1. Speedup improves with the number of processors but no additional benefit is gained by spawning processes more than the number of processors allocated to the job.
  2. There is no communication among worker processes.

  The fixed software architecture is preferable if the application's speedup depends on the number of processes spawned as well as the number of prcessors allocated.

Effect of the topology of the interconnection network

- We have shown that the network topology can affect system performance but, in most cases, the relative performance of the time-sharing and static policies is not affected by the topology. The use of other, more efficient switching techniques such as wormhole routing will further reduce the performance sensitivity to network topology.

Impact of process and job variance

- The static policy performs poorly when there is variability in the service times of the processes in a given application. For example, if the software architecture exhibits variable parallelism, as in the divide-and-conquer type of software structure, time-sharing provides a significant benefit over static space-sharing.

- When there is variance in the total service demand of jobs but no variance in the process service times of a job, the static partitioning is better in most cases. For a single class workload in which there is small variance in either total service demands of jobs or service demands of processes, static partitioning is expected to perform even better.

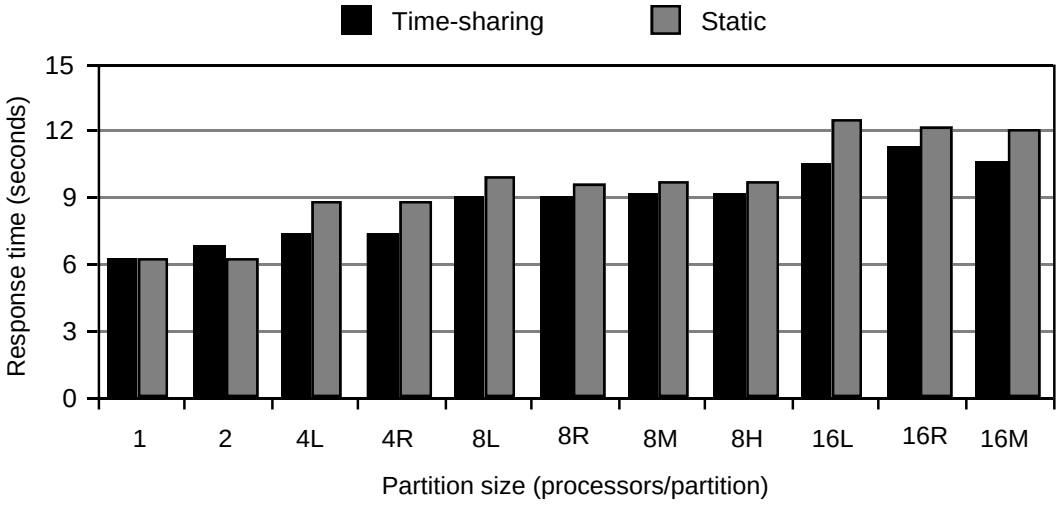

Figure 9: Mean response time of the generic workload for fixed software architecture with high process service time variance

The work presented here can be extended in several directions. Investigations of both time-sharing and space-sharing policies in the context of larger distributed-memory systems forms an important direction of future research. Study of the impact of hierarchical interconnection networks is also important as many large distributed-memory systems are based on a hierarchy (e.g., the Stanford Dash). Performance of a hierarchical scheduling policy has been studied in [3] for shared-memory systems, which is based on a hierarchy of process queues proposed in [6]. It is interesting to study the performance of this policy in the context of distributed-memory systems. Experimenting with systems in which the multiprogramming mix contains a variety of different programs is also worthy of investigation.

Acknowledgement

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

References

[1] T. Anderson, David Culler, David Patterson, and the NOW team, "A Case for NOW (Networks of Workstations)," *IEEE Micro*, 15(2), February 1995, pp. 54-64.

[2] S. L. Au and S. P. Dandamudi, "The Impact of Program Structure on the Performance of Scheduling Policies in Multiprocessor Systems," *Int. J. Computers and Their Applications*, Vol. 3, No. 1, April 1996, pp. 17–30.

[3] S. Ayachi and S. P. Dandamudi, "A Hierarchical Processor Scheduling Policy for Multiprocessor Systems," *IEEE Symposium on Parallel and Distributed Processing*, New Orleans, October 1996.

[4] Y. N. Chan, *Processor Scheduling in a Transputer System*, MCS Thesis, School of Computer Science, Carleton University, Ottawa, 1996.

[5] S. Chiang, R. K. Mansharamani and M. K. Vernon, "Use of Application Characteristics and Limited Pre-emption for Run-To-Completion Parallel Processor Scheduling Policies", *Proceedings of the ACM SIGMETRICS Conference*, 1994, pp. 33-44.

[6] S. P. Dandamudi and S. P. Cheng, "A Hierarchical Task Queue Organization for Shared-Memory Multiprocessor Systems", *IEEE Trans. Parallel and Dist. Syst.*, Vol. 6, Jan. 1995, pp. 1-16.

[7] K. Dussa, B. Carlson, L. Dowdy, and K.-H. Park, "Dynamic Partitioning in a Transputer Environment", *Proceedings of the ACM SIGMETRICS*, May 1990, pp. 203-213.

[8] D. G. Feitelson and L. Rudolph, "Distributed Hierarchical Control for Parallel Processing", *IEEE Computer*, 23(5), May 1990, pp. 65-77.

[9] G. C. Fox et al. *Solving Problems on Concurrent Processors Volume I: General Techniques and Regular Problems*, Prentice-Hall, 1988.

[10] D. Ghosal, G. Serazzi and S. K. Tripathi, "The Processor Working Set and Its Use in Scheduling Multiprocessor Systems", *IEEE Transactions on Software Engineering*, Vol. 17, No. 5, May 1991.

[11] A. Gupta, A. Tucker and S. Urushibara, "The Impact of Operating System Policies and Synchronization Methods on the Performance of Parallel Applications", *Proceedings of the ACM SIGMETRICS Conference*, May 1991, pp. 120-132.

[12] E. Horowitz and A. Zorat, "Divide-and-Conquer for Parallel Processing," *IEEE Trans. on Computers*, C-32, 1983, pp. 582–585.

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

[14] S. Majumdar, D. L. Eager and R. B. Bunt, "Scheduling in Multiprogrammed Parallel Systems", *Proceedings of the ACM SIGMETRICS Conference*, May 1988, pp.104-113.

[15] S. Majumdar, D. L. Eager and R. B. Bunt, "Characterization of Programs for Scheduling in Multiprogrammed Parallel Systems", *Performance Evaluation*, 13, 1991, pp.109-130.

[16] Y. M. Leung and S. Majumdar, "The Effect of Inter-Process Communication on Scheduling in Multiprogrammed Distributed Memory Systems," *Proc. First Int. Workshop on Parallel Processing*, Bangalore, India, December 1994, pp. 474-481.

[17] C. McCann and J. Zahorjan, "Processor Allocation Policies for Message-Passing Parallel Computers", *Proceedings of the ACM SIGMETRICS Conference*, May 1994, pp. 19-32.

[18] A. K. Nanda, H. Shing, T. H. Tzen and L. M. Ni' "Resource Contention in Shared-memory Multiprocessors: A Parameterized Performance Degradation Model," *J. Parallel and Distributed Computing,*, 12, 1991, pp. 313–328.

[19] J. K. Ousterhout, "Scheduling techniques for Concurrent Systems", *Proceedings of the 3rd International Conference on Distributed Computing Systems*, 22-30, Oct. 1982, pp. 22-30.

[20] E.W. Parsons, and K.C. Sevcik, "Multiprocessor Scheduling for High Variability Service Time Distributions," *Job Scheduling Strategies for Parallel Processing*, D.G. Feitelson, and L. Rudolph (eds.), Lecture Notes in Computer Science, Vol. 949, Springer Verlag, 1995, pp. 127–145.

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

[22] S. K. Setia, M. S. Squillante, and S. K. Tripathi, "Processor Scheduling on Multiprogrammed, Distributed Memory Parallel Computers", *Proceedings of the ACM SIGMETRICS Conference*, May 1993, pp. 158-170.

[23] S. K. Setia, M. S. Squillante, and S. K. Tripathi, "Analysis of Processor Allocation in Multiprogrammed, Distributed-Memory Parallel Processing Systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 4, Apr. 1994, pp. 401-420.

[24] S. Setia and S. Tripathi, "A Comparative Analysis of Static Processor Partitioning Policies for Parallel Computers," *Proc. MASCOTS*, 1993.

[25] K. C. Sevcik, "Characterizations of Parallelism in Applications and their Use in Scheduling", *Proc. ACM SIGMETRICS Conf.*, 1989, pp. 171-180.

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

[27] A. Tucker and A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors", *Proc. ACM Symp. Operating System Principles*, Dec. 1989, pp. 159-166.