Supporting Dynamic Space-sharing on Clusters of Non-dedicated Workstations *

Abdur Chowdhury Lisa D. Nicklas Sanjeev K. Setia Elizabeth L. White
Department of Computer Science
George Mason University

Abstract

Clusters of workstations are increasingly being viewed as a cost-effective alternative to parallel supercomputers. However, resource management and scheduling on workstations clusters is complicated by the fact that the number of idle workstations available for executing parallel applications is constantly fluctuating. In this paper, we present a case for scheduling parallel applications on non-dedicated workstation clusters using dynamic space-sharing, a policy under which the number of processors allocated to an application can be changed during its execution. We describe an approach that uses application-level checkpointing and data repartitioning for supporting dynamic space-sharing and for handling the dynamic reconfiguration triggered when failure or owner activity is detected on a workstation being used by a parallel application. The performance advantages of dynamic space-sharing are quantified through a simulation study, and experimental results are presented for the overhead of dynamic reconfiguration of a grid-oriented data parallel application using our approach.

1 Introduction

Several resource management and scheduling issues arise in using clusters of workstations for parallel computing that have no counterpart in traditional parallel supercomputing environments. Most of these issues arise due to the fact that workstations are typically "owned" by a user who may resent the presence of an external parallel computation on his or her computer [12, 2]. It has also been shown that parallel applications need a dedicated environment for good performance[3]. To keep workstations owners happy, and to provide good performance for parallel applications, it becomes necessary to ensure that parallel applications only execute on workstations that are idle.

*This work was partially supported by NSF grants CCR-9409697 and CCR-9625202

Another important issue that arises in cluster environments is the need to handle failures on a subset of the nodes involved in a parallel computation. Mechanisms such as checkpointing [12, 10, 19] that enable a parallel application to recover from the failure of one or more of the nodes on which it is executing thus become necessary. Both these issues – providing mechanisms for withdrawing a parallel computation from a workstation on detecting owner activity, and providing mechanisms for fault tolerance – are related since they deal with the common problem that underlying resources available to a parallel computation are changing.

The fluctuating available processing capacity of a non-dedicated cluster of workstations also poses several challenging problems for the job scheduler responsible for allocating idle workstations to executing parallel applications. The job scheduler must balance the needs of both interactive applications belonging to workstation owners and multiple batch applications trying to scavenge idle cycles from the cluster.

One of the scheduling policies that has been shown to have good performance in dedicated parallel environments is dynamic space-sharing [6, 13, 17]. Under dynamic space-sharing, the processors of a parallel system are divided into disjoint partitions that are allocated to individual jobs. However, the number of processors allocated to a partition can be changed dynamically in response to events such as new job arrivals or job departures.

In this paper, we present a case for using dynamic space-sharing for scheduling parallel jobs in non-dedicated workstation clusters. We show that having the ability to reconfigure and change the processor allocation of an executing parallel job makes dynamic space-sharing especially attractive in cluster environments since the available processing resources are constantly changing.

One of the challenges in implementing dynamic space-sharing is the need to provide mechanisms for

0-8186-7813-5/97 $10.00

dynamically reconfiguring an executing parallel application. This is an active area of research and various approaches have been proposed for different classes of applications, e.g., process control [22] for programs based on the task-queue model, WoDi[20] for master-slave applications, and DRMS [14] and Adaptive Multi-block PARTI[1] for two classes of SPMD applications. In this paper, we describe an application-level approach for dynamic reconfiguration of iterative grid-based applications, an important class of parallel applications. Our approach uses an integrated set of mechanisms for supporting dynamic space-sharing and for handling dynamic reconfiguration triggered when owner activity or the failure of a workstation is detected.

The rest of this paper is organized as follows. Section 2 discusses the reasons for using dynamic space-sharing as the jobs scheduling policy on non-dedicated workstation clusters. Section 3 discusses the design and implementation of our approach for application-level dynamic reconfiguration. Section 4.1 contains a simulation study quantifying the performance advantages of dynamic space-sharing. Section 4.2 discusses issues that arise in the reconfiguration of grid-based parallel applications, and presents measurement results for the cost of dynamic reconfiguration. Section 5 describes future work and conclusions.

2 Motivation for Dynamic Space-sharing

In the last few years, several studies have examined job scheduling strategies for dedicated parallel environments [6, 13, 11, 7, 17]. Policies that employ dynamic space-sharing have been shown to outperform other policies such as static space-sharing and gang-scheduling with static partitioning for many workloads.

In addition to its advantages in dedicated environments, dynamic space-sharing has some natural advantages in cluster environments. Figure 1 plots the number of idle workstations that can be harnessed for a parallel computation on a cluster with 53 workstations in our department on a typical day. Consider a parallel program with a parallelism of 32 processors executing on this non-dedicated cluster. From Figure 1 it is clear that for a portion of the day, there would not be enough idle workstations for the parallel application to run with one process per processor.

In this situation, the job scheduler has three alternatives depending on the policy being used. It could schedule multiple processes belonging to the application on the same workstation. However, most parallel programs are written using a programming model

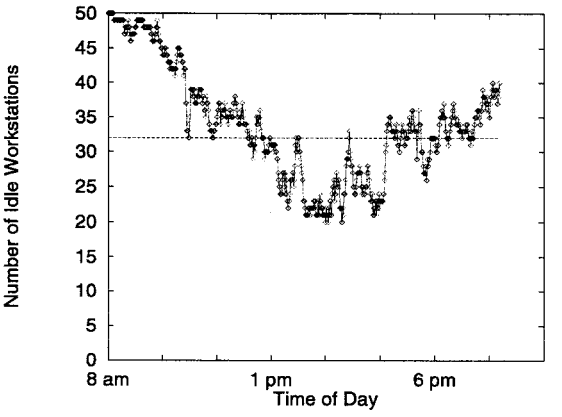

Figure 1: The number of workstations that can be harnessed for parallel computation in a cluster of 53 workstations on a typical day.

that assumes that there is a one-to-one mapping between the component processes of the program and processors. Moreover, the program's data and work are partitioned among these processes assuming that they will be executing on separate processors. Allocating more than one process to a single node violates these assumptions, and results in poor performance. Thus a better alternative would be to suspend the job that requires 32 processors, and run jobs that require fewer processors. A third alternative, which is only feasible if the scheduling policy being used is dynamic space-sharing, is that the job could be dynamically reconfigured to execute on as many processors as were available.

The third alternative is clearly the best from the point of view of the user who submitted the parallel job. Jobs with high degrees of parallelism would suffer under the second policy. Quantitative results that back up these arguments are presented in Section 4. Furthermore, as mentioned above, several studies [13, 17] have established that dynamic space-sharing results in higher system throughput than other policies. Thus, dynamic space-sharing has advantages from both user and system performance perspectives.

3 Dynamic Parallel Application Reconfiguration

3.1 Overview

In a non-dedicated workstation cluster environment, the need for dynamic reconfiguration of a parallel application can arise in three situations: (i) when a node being used by the application fails, (ii) when owner activity is detected on a node, and (iii) if the job scheduler decides to reduce or increase the num-

ber of processors allocated to the application. Our approach handles all these cases within an integrated framework that uses language-level checkpointing and application-specific state modification.

There are two scenarios possible while reconfiguring an application. Under the first scenario, the number of processes belonging to an application does not change during reconfiguration. However, the underlying set of physical processors allocated to the application changes. This scenario will occur when there is an idle workstation available in the system to replace a workstation which has become unavailable because of failure or owner activity. In this case, reconfiguration involves creating a process on the new workstation and initializing its internal state based on the state of the process that was executing on the replaced workstation.

Under the second scenario, the application changes its degree of parallelism during reconfiguration. This scenario will occur if an insufficient number of processors is available or if the job scheduler decides to change the number of processors allocated to an application in response to a job arrival or departure. Reconfiguration under the second scenario is more complex. The actions that need to be taken for this are application-specific. While not every parallel application is amenable to such reconfiguration, several large and important classes of applications, e.g., iterative grid-based scientific applications, fork-join applications, and master-slave type applications can be reconfigured at certain points in their execution and continue to function correctly even after their degree of parallelism is changed. The reader is referred to [15] for a more detailed discussion of this issue.

In this paper, we restrict our attention to iterative grid-based parallel applications. This is a class of problems that occur frequently in science and engineering [8] in which the problem domain can be visualized as a multi-dimensional grid. We note that while the details of the reconfiguration will be different for other classes of applications, the issues involved are similar. These are illustrated through the example below.

Consider a iterative parallel program that operates on a two-dimensional grid (represented by a two-dimensional array). Suppose that the grid is partitioned among the component processes of the program using a column-partitioning data decomposition scheme, and that the processes of the program are organized in a linear array for purposes of communication (see Figure 2). During every iteration, the work done by each process consists of modifying its portion of the grid and involves communication with its neighbors in the linear array. At the end of an iteration, each process synchronizes and a convergence criterion is checked to see if the program should terminate.

Suppose this program initially executes on four processors, but a scheduler decision reduces its degree of parallelism to three. Figure 2 shows that reconfiguration involves modifying the following components of each process' internal state: (i) the logical communication channels between the component processes are changed to reflect the new logical structure, (ii) the data is redistributed among the application processes, and (iii) process control variables are modified to reflect the new work allocated to a process.

It is important to note that such a reconfiguration can only be initiated at points in the application's execution where there exists a consistent snapshot of the application's data set and where modifications to the data decomposition do not result in any change in the application's semantics. For the grid-based program in the example above, reconfiguration can occur at the synchronization point at the end of an iteration.

In order to be able to recover from workstation or network failures, it is necessary to periodically save the application's internal state at such a synchronization point (or *schedulable and observable point (SOP)* using the terminology of [14]) on stable storage, i.e., create a checkpoint on disk. After a failure, recovery will involve reconfiguring the application based on the state stored in the checkpoint.

We note that workstation failures in a cluster environment are a relatively rare event. On the other hand, dynamic reconfiguration because of owner activity is likely to occur quite frequently on a cluster of non-dedicated workstations. In this case, it is not necessary to use a previously saved checkpoint during reconfiguration. Instead, we can initiate reconfiguration as soon as the application has reached a safe point in its execution.

From the example above, we can make the following observations: First, dynamic reconfiguration involving a change in a program's degree of parallelism requires a knowledge of internal application details such as the location of SOPs and the data partitioning scheme being used, and is therefore application-specific, or at least domain-specific. Second, reconfiguration involves reconstructing the state of an application. This will typically involve data redistribution, and may involve reconstructing the stack of each process. To support such dynamic reconfiguration on a cluster of heterogeneous workstations, high-level architecture-independent mechanisms are necessary.

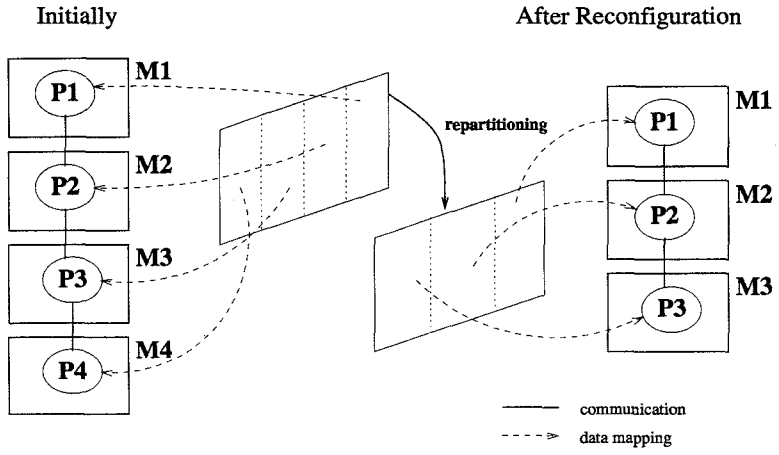

Figure 2: Reducing the parallelism from four to three using dynamic reconfiguration.

Thus any system that supports dynamic application reconfiguration has to deal with the following issues: (i) what enhancements or modifications need to be made to a parallel application to make it dynamically reconfigurable? (ii) what support is required to make dynamic reconfiguration efficient? These issues are discussed below.

3.2 Support for Dynamic Reconfiguration

Writing Reconfigurable Programs Under our approach for dynamic reconfiguration, two aspects of a reconfigurable parallel program distinguish it from a conventional parallel program. First, the reconfigurable program includes checkpointing routines that perform the task of saving the portion of the internal state of each process used during reconfiguration after a node failure. Second, the program includes reconfiguration routines that reconstruct the new internal state of each process.

Ideally, programmers should not be involved in making a parallel application reconfigurable. The routines for checkpointing and reconfiguration should be part of a library linked automatically to an application. However, as the example in Section 3.1 shows, dynamic reconfiguration involves a knowledge of application-specific details such as the location of safe points for reconfiguration, the data decomposition scheme used, and the logical configuration of processes.

We believe that the only programming model applicable to several classes of applications that can support dynamic reconfiguration *without programmer involvement* is the task-queue or master-slave model. Run-time systems such as Piranha [9], Process Control [22], and WoDi [20] that support reconfigurable parallel applications all use a variant of this model. The generality of this model comes at a price; for some classes of applications using the task-queue model results in relatively poor performance [15, 1]. We believe that application developers will not be willing to pay the price of poor performance to make their applications reconfigurable.

Our approach to making programs reconfigurable involves extending the existing domain-specific run-time system or programming model being used by the application. Specifically, routines that perform the tasks of checkpointing and reconfiguration are added to the run-time library. These routines can either be written explicitly by the application programmer or generated automatically based on specifications and annotations provided by the programmer. The second alternative is clearly preferable since it reduces programmer involvement in the reconfiguration.

For many classes of applications such as iterative numerical programs that operate on structured and unstructured grids, it is possible to come up with simple annotations for the programmer to specify the actions to be taken during reconfiguration. We believe that in the case of grid-oriented scientific applications, requiring the programmer to annotate their program in this manner is not too burdensome a requirement, given the fact that similar annotations are required when using a data-parallel language (e.g., HPF).

For other classes of applications, specifically highly irregular applications, this approach may not be feasible since these applications require the use of dynamic data structures and dynamic load-balancing [23]. For such applications, it is necessary to extend the run-time system to support dynamic reconfiguration. By

building these reconfiguration procedures into the run-time system, they can be made transparent to the programmer who uses the run-time system library.

Application-level Checkpointing We use an application-level approach for checkpointing, i.e., each application includes routines that save a subset of its internal state on stable storage. Further, a checkpoint does not contain any internal machine information such as register and stack values. Only a subset of the variables, those that are needed for reconfiguration and recovery, are checkpointed. This subset is determined with the help of the application programmer as discussed above. For the iterative applications under consideration, reconfiguration is always initiated at the top of a loop inside the main function. As such, it is not necessary to capture and restore the stack of a process during reconfiguration.

The use of application-level checkpointing has two advantages. First, it permits reconfiguration in a heterogeneous environment since checkpoints do not contain any machine-specific information. Second, it results in a reduction in the amount of data saved in a checkpoint. This can lead to substantial savings over system-level approaches [12] that checkpoint the entire address space of each process belonging to the application [21].

Despite the use of application-level checkpointing, the size of a checkpoint can be quite large since many scientific programs operate on large data sets. To reduce this overhead, we utilize the local disks attached to each workstation for saving checkpoints. Each process of the parallel application saves its portion of the checkpoint on both its local disk and on a remote disk. Storing a copy of the checkpoint on two nodes guarantees that the application can recover from the failure of a single node. Two copies of a checkpoint are maintained on the local disk to recover from failures occurring during the process of propagating the checkpoint to disk. Processes synchronize among themselves to maintain a consistent view of the current version of the checkpoint.

Reconfiguration Procedure The overall architecture of our execution environment includes a global scheduler that manages the allocation of workstations to jobs, and a resource manager process on each workstation that monitors owner activity and CPU load. The resource process periodically notifies the global scheduler of the current status of its workstation. As such, this process also serves as a failure daemon.

When the global scheduler decides to reconfigure an executing parallel application, it notifies the application via a signal. On receiving such a signal, the application initiates reconfiguration at the next safe point in its execution, i.e., at the end of the current iteration.

During a reconfiguration that does not involve a node failure, processes are spawned on any new nodes allocated to the application. The new logical configuration of the application is established, and the data partitioning algorithm being used by the application is invoked. Using this information, each process in the previously executing group of processes determines the new owner of its portion of the grid and then sends this data to these processes. On receiving this data, each process maps the data to the appropriate local data structures. In addition, each process initializes the relevant control variables based on the current configuration of the application. After these steps, processes executing on processors that are no longer allocated to the application are aborted, and each process proceeds with the execution of the program.

In the case of reconfiguration involving recovery from node failure, the application is first restarted (in its original logical configuration) using the checkpoints saved on disk. The process that was executing on the failed workstation is replaced by a process on a different workstation. After this step, reconfiguration proceeds in the manner discussed above.

Implementation Status We are implementing this system on a cluster of Sparc workstations. A public domain implementation of MPI from the Ohio Supercomputer Center (LAM 6.1) is used as the message-passing library by the applications. LAM 6.1 provides several extensions to the MPI-1 standard[18]. In particular, LAM's facilities for dynamic process creation, failure detection, and signals have proved invaluable for our implementation. Currently, reconfiguration and checkpointing stub routines are manufactured by hand. However, work is underway to automate the generation of these stubs from annotations and declarations made by the programmer. We are using an approach similar to that used by the DRMS project [14].

4 Results

4.1 Dynamic Space-sharing Performance

In this section, we present results from a simulation study that compares the performance of dynamic space-sharing with static space-sharing on a non-dedicated workstation cluster. The goal of this study is to examine the effect of factors such as job parallelism, time of job submission, and reconfigura-

	Mean	CV	Min.	Max.
Available workstations	30.48	0.29	11	50
Idle interval(minutes)	20.67	3.14	1	720
Busy interval(minutes)	8.49	2.34	1	535

Table 1: Statistics for the traces used in the simulation. The total number of workstations monitored is 50.

tion overhead on the relative performance of dynamic space-sharing and static policies.

For this study, we monitored user-activity and CPU load on 53 workstations in our department over a one month period. Thirty five of these workstations are privately owned while 18 are in a public lab used by students and faculty. Background processes logged keyboard, mouse, and terminal activity as well as CPU load on each workstation. Traces collected during the daytime hours (8 am to 8 pm) on five days on 50 of the 53 workstations monitored were used for driving the simulations described below.

A workstation was classified as idle if there was no user activity and the average CPU utilization over 20 seconds was less than 10%. However, an "idle" workstation was not considered available for scheduling parallel jobs unless the workstation stayed idle for 6 minutes. In general, our traces show that a large fraction of workstations are idle at any given time, and can be used by parallel applications in search of CPU cycles. The idle capacity observed is similar to that reported by earlier studies [16, 2]. Overall statistics for the five traces used in the simulation are summarized in Table 1.

We note, however, that these statistics do not capture the wide variation in the number of available workstations during the day. Our traces show that the number of busy workstations varies from hour to hour during the day with peak usage occurring in the afternoon. This trend is shown in Figure 1, which plots the number of available workstations at different times during a typical day. In general, a higher recruitment threshold results in a smaller number of available workstations.

To study the factors affecting the performance of job scheduling strategies, we evaluated the average response time under dynamic and static space-sharing on the monitored cluster. Under dynamic space-sharing, if the number of available idle processors falls below the number required by a parallel application, the application is reconfigured to execute on the available processors. Under static space-sharing, however, if the number of available processors falls below the degree of parallelism of the job, it is suspended. Under both policies, if a processor being used becomes busy, but an idle processor is available, the job is reconfigured by migrating the affected process to the idle processor.

We considered a simplified parallel workload consisting of N parallel jobs, all submitted at the same time, each having the same maximum parallelism (P) and the same execution time. Thus the total demand for processors imposed by the parallel workload is $TPD = N \times P$. Each parallel application is considered to have a perfect speedup curve, a pessimistic assumption for dynamic space-sharing. A constant overhead (ovr) is charged for all reconfigurations that involve a change in the degree of parallelism of an application, but reconfigurations that do not involve a change in the degree of parallelism of an application, i.e., process migrations, are assumed to have an overhead of $0.5 \times ovr$. These assumptions are based in part on the results of Section 4.2, which show that reconfigurations involving only process migration are less expensive than other reconfigurations.

The primary metric used for evaluating the performance of a policy is the normalized response time which is defined as the ratio of the parallel job's response time on a non-dedicated cluster to its execution time on a dedicated cluster of workstations. For all the experiments described below, the parallel workload was simulated using traces for five different days, and the average normalized response time (denoted by $\overline{R}$) is reported. We performed several experiments in which we varied the input parameters to the simulation. A subset of our results is discussed below; for a more detailed discussion the reader is referred to [4].

Our first set of results examines the impact of varying P and TPD on the relative performance of static and dynamic space-sharing. By varying P we can examine the effect of varying the average parallelism of jobs in the workload on the performance of a scheduling policy, and by varying TPD we can examine the extent to which a parallel workload can be sustained on a non-dedicated workstation cluster.

We performed two sets of simulation experiments (referred to as Experiment A and Experiment B respectively) in which we varied the execution time and arrival time of jobs in various parallel workloads with $TPD = 16, 24$ and 32, and $N = 1, 2$, and 4. In Experiment A, each program has a run-time of 3 hours and is submitted at 9 am, while in Experiment B, each program has a run-time of 1 hour and is submitted at 12 pm, a time when typically there are fewer worksta-

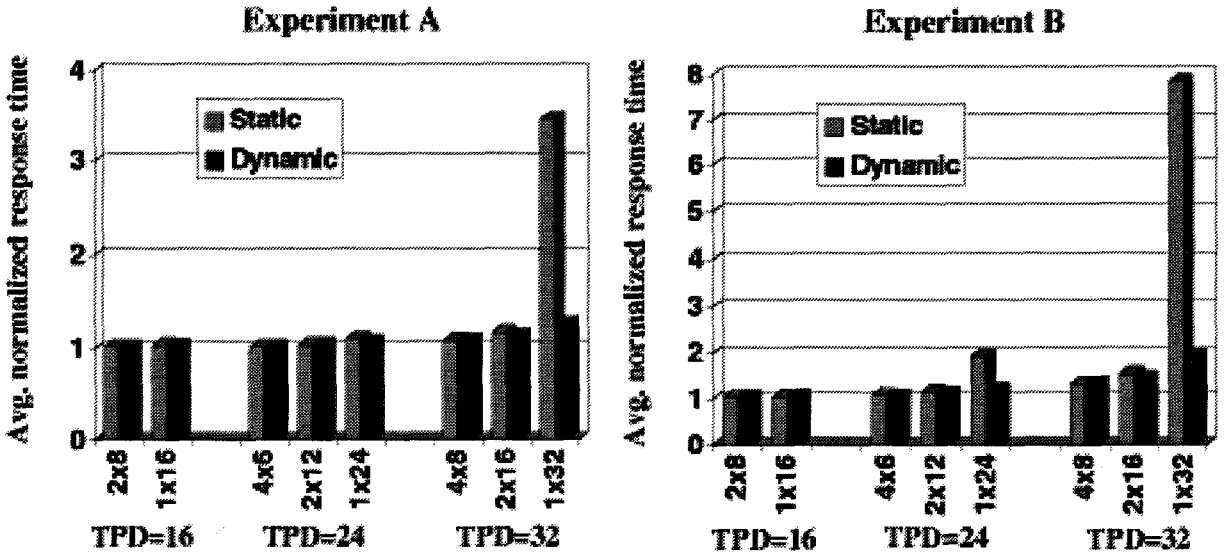

Figure 3: The average normalized response time as a function of TPD and P on a non-dedicated cluster with 50 workstations for (a) Experiment A (b) Experiment B. The reconfiguration overhead (ovr) for these experiments is 36 seconds.

tions that can be harnessed for parallel computation than at 9 am (see Figure 1).

Figures 3(a) and (b) plot $\overline{R}$ under the two policies for experiments A and B respectively. From Figure 3(a), we observe that for all workloads where TPD is less than 32, the average normalized response time ($\overline{R}$) under both static and dynamic space-sharing is close to 1. This result shows that under the load conditions of experiment A, for workloads with $P = 16$ and 24, both static and dynamic space-sharing provide equivalent performance. For $TPD = 32$, however, the relative performance of the two policies depends on the parallelism (P) of the jobs in the workload. For $P = 8$ and 16, $\overline{R}$ under both policies is close to 1 but for $P = 32$, the performance of static space-sharing is extremely poor relative to that of dynamic space-sharing. The trends illustrated in Figure 3(b) for Experiment B are similar, except that static space-sharing also has poor performance relative to dynamic space-sharing for the workload with $P = 24$, and the normalized response times of both policies for $TPD = 24$ and 32 are larger than in Figure 3(a).

These results show that the ability to dynamically reconfigure parallel jobs allows dynamic space-sharing to support workloads consisting of jobs with higher degree of parallelism than can be supported under static space-sharing. We note that the increase in the response time under static space-sharing is because a parallel job is suspended if there are not enough idle processors, while the increase in response time under dynamic space-sharing is a result of the overhead of dynamic reconfiguration. Further, we note that for the parallel workload model used in our simulation experiments, dynamic space-sharing and space-sharing will have the same performance on a dedicated system. Thus the differences in the performance of these policies reflect the performance benefits of dynamic space-sharing on a non-dedicated workstation cluster.

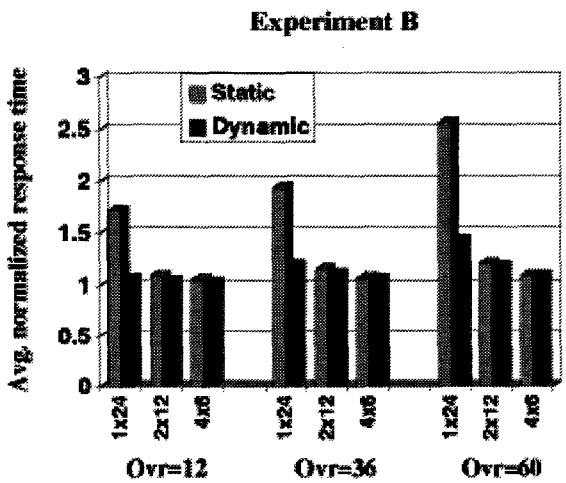

Figure 4: The average normalized response time as a function of overhead (ovr) for the workloads with $TPD = 24$ in Experiment B.

We now consider the impact of varying the reconfiguration overhead (*ovr*) on the relative performance of the policies. In Figure 4, we plot $\overline{R}$ as a function of the *ovr* for the workloads with $TPD = 24$ considered in Experiment B[1]. We observe that while the overhead has a significant impact on the performance of both policies, the relative performance of dynamic space-sharing is better than that of static space-sharing even for $ovr = 60$.

In summary, the results of our simulation study show that dynamic space-sharing can outperform static space-sharing for a variety of workloads. The performance advantages of dynamic space-sharing are greatest for workloads consisting of parallel jobs with high degrees of parallelism, and at times of the day with relatively low available processing capacity.

4.2 Reconfiguration Overhead

We now describe the results of experiments in which we measured the cost of dynamic reconfiguration for a grid-based parallel application. The reader is referred to [4] for more detailed performance measurements. The main goal of this study is to identify the various components that contribute to the overhead of dynamic reconfiguration. We conducted our experiments on a cluster of Sparc workstations interconnected by a 10 Mbps Ethernet, using LAM/MPI for message passing between processes. Each workstation has 32 MB of memory and a local disk which is used by the checkpointing service.

The benchmark application considered solves the Poisson equation on a square grid using the Jacobi relaxation method [8]. This is a well-known iterative technique which uses a five-point stencil to update each grid-point during an iteration, i.e., each point is updated with the average of the values of the four neighboring grid points from the previous iteration. This procedure is repeated until the solution converges. A global synchronization is required at the end of each iteration to check the convergence criterion.

The grid data is divided among the component processes of the application using a data partitioning algorithm called Recursive Bisection. This algorithm was proposed by Crandall and Quinn [5] for decomposing a two-dimensional grid for heterogeneous computing environments such as clusters of workstations. This algorithm recursively divides the grid into rectangles such that the number of grid points in the rectangle allocated to a processor is proportional to its computational power.

We measured the cost of reconfiguration under five different scenarios (i) reducing the number of processors allocated to the application from 8 to 4 (ii) increasing the number of processors allocated from 4 to 8 (iii) reducing the number of processors allocated from 8 to 7 (iv) increasing the number of processors allocated from 7 to 8 (v) migrating a process from one node to another while keeping the processor allocation fixed. The first two scenarios are likely to occur when the job scheduler makes a decision in response to the arrival of a new job or the departure of an executing job. The rest of the scenarios occur because of the non-dedicated nature of the workstation cluster, e.g., when user activity is detected on a workstation or when a new node becomes available.

We performed reconfigurations under all these scenarios for two cases: (i) assuming there was no node failure which implies that reconfiguration involves redistributing the current data among the continuing processes and any newly spawned processes and (ii) assuming there was a single node failure which implies that the processes must read checkpoint files on disk before reconfiguring. All experiments were repeated multiple times using several grid sizes with averages reported for the benchmark in Table 2.

grid size N x N	change	spawn	redist	other	total	read chkpt
1024 (4 Mb)	$8 \rightarrow 4$	0.00	0.73	0.15	0.88	3.04
	$4 \rightarrow 8$	0.87	0.82	0.11	1.80	6.07
	$8 \rightarrow 7$	0.00	1.08	0.08	1.16	3.04
	$7 \rightarrow 8$	0.49	1.06	0.09	1.64	3.89
	$8 \rightarrow 8$	0.50	0.56	0.03	1.09	3.04
2048 (16 Mb)	$8 \rightarrow 4$	0.00	2.62	0.48	3.10	13.82
	$4 \rightarrow 8$	0.90	3.18	0.13	4.21	20.58
	$8 \rightarrow 7$	0.00	4.03	0.15	4.18	13.82
	$7 \rightarrow 8$	0.50	4.19	0.11	4.80	16.78
	$8 \rightarrow 8$	0.51	2.14	0.05	2.70	13.82
3072 (36 Mb)	$8 \rightarrow 4$	0.00	6.12	0.92	7.04	29.69
	$4 \rightarrow 8$	0.90	22.04	5.45	28.39	60.08
	$8 \rightarrow 7$	0.00	8.90	0.26	9.16	29.69
	$7 \rightarrow 8$	0.50	9.34	0.13	9.97	31.46
	$8 \rightarrow 8$	0.50	4.78	0.06	5.34	29.69

Table 2: Reconfiguration overhead (in seconds) for the Poisson solver application.

Reconfiguration for grid-based applications involves several steps (i) spawning any new processes (ii) reestablishing the logical configuration of the processes (iii) figuring out the new logical data partitioning, e.g. by invoking the recursive bisection algorithm (iv) al-

[1]Note that on workstation clusters (as opposed to dedicated multiprocessors), static space-sharing will incur reconfiguration overhead for process migration.

locating memory for the newly assigned portion of the grid (v) figuring out the overlap of the current data assignment with the future data assignments (vi) sending out data now assigned to other nodes (vii) and receiving data from the current set of processes for the new data partitioning. In Table 2 we report the measured times for each of these steps. The time for step (i) is reported under the column labeled spawn. The time for steps (ii) through (v) are combined and reported under the column labeled other. The time for steps (vi) and (vii) are combined and reported under the column labeled redist. The last column labeled read chkpt reports the time to read in checkpoint files after a node failure. The sum of the two columns total and read chkpt would be the time for reconfiguration after a node failure.

The results in Table 2 show that the biggest component of reconfiguration overhead is the time for data redistribution. As expected, the data redistribution times increase with the size of the grid. The redistribution time is least when a simple migration is involved, moving one node's data in one message. The data redistribution costs when reconfiguring to and from 7 processes are slightly higher than the costs when reconfiguring to and from 4 processes. Reconfiguration between 7 and 8 processes requires more messages than reconfiguration between 4 and 8 processes when using recursive bisection to partition data among processes. The second largest component of the total time for reconfiguration is the time spent spawning any new processes. Spawning in MPI is a collective operation and is therefore dependent on the number of processes that participate in the spawn. Since the spawn time is independent of the grid size, the spawn time becomes a smaller percentage of the total time for reconfiguration as the grid size increases.

Note that for the third grid size (36 Mbytes), the benchmark has very poor performance when reconfiguring from 4 to 8 processors. In this instance, the memory requirements of each process exceeds the memory available on the workstation during reconfiguration causing several thousand pages to be written to disk, adversely affecting the performance of data retrieval and memory allocation.

Checkpointing Overhead Each process will have the additional overhead of taking periodic checkpoints of its current state to recover from a node failure. In our approach the checkpoint is made at the application level, reducing the amount of data that must be saved. For our benchmark application, using application-level checkpointing reduced the amount of data in each checkpoint file to less than one-half of a system-level checkpoint file.

When reconfiguring after a node failure, the failed node's remote checkpoint file must be retrieved from a remote node since its local checkpoint file is not accessible. Simultaneously the surviving nodes will read their local checkpoint files and wait for the reading of the remote checkpoint file to complete. We note that the time spent reconfiguring after a node failure is many times higher than the time spent reconfiguring under other scenarios. This is tolerable in a cluster environment where the mean time between node failures is high.

Our results show that, even in an environment where the interconnection network has low bandwidth, dynamic reconfiguration can be accomplished at the cost of 10s of seconds. As high-speed LANs start replacing Ethernets in cluster environments, these costs will be reduced, further increasing the performance benefits of dynamic space-sharing.

5 Conclusions

In this paper, we have presented a case for using dynamic space-sharing for scheduling multiple parallel jobs on a cluster of non-dedicated workstations. Using trace-driven simulation, we have analyzed the factors that affect the performance of dynamic space-sharing, and quantified the performance advantages of dynamic space-sharing relative to other policies.

We have designed and implemented a run-time system that supports dynamic space-sharing. Our system uses high-level checkpointing and application-specific data repartitioning to dynamically reconfigure grid-based parallel applications. Mechanisms for handling the withdrawal of an application from a node are integrated with mechanisms that support dynamic changes in the degree of parallelism of an application. Performance measurements were presented showing that the cost of dynamic reconfiguration largely depends on the bandwidth of the underlying interconnection network. We show that for long-running applications, reconfiguration can be implemented at an acceptable cost, even on a cluster of workstations interconnected by a 10 Mbps Ethernet.

We are continuing work on automating the generation of checkpointing and reconfiguration stubs, and integrating our dynamic reconfiguration service into a common framework. We also plan to explore issues involved in supporting the dynamic reconfiguration of other classes of applications, and in improving the efficiency of dynamic reconfiguration.

References

[1] A. Acharya, G. Edjlali, and J. Saltz. The Utility of Exploiting Idle Workstations for Parallel Computation. Technical Report CS-TR-3710, University of Maryland, College Park, Computer Science Department, November 1996.

[2] T. Anderson, D. Culler, and D. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[3] R. Arpaci, A. Dusseau, A. Vahdat, L. Liu, T. Anderson, and D. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proc. of ACM SIGMETRICS '95*, Ottawa, May 1995.

[4] A. Chowdhury, L. Nicklas, S. Setia, and E. White. Supporting Dynamic Space-sharing on Clusters of Non-dedicated Workstations. Technical Report CS-97-01, George Mason University, March 1997.

[5] P. Crandall and M. Quinn. Block Data Decomposition for Data-Parallel Programming on a Heterogeneous Workstation Network . In *Proceedings of the 2nd Intl. Symp. on High-Performance Distributed Computing*, pages 42–59, July 1993.

[6] K. Dussa, B. Carlson, L. Dowdy, and K.-H. Park. Dynamic partitioning in transputer environments. In *Proc. of the ACM SIGMETRICS Conf.*, 1990.

[7] D. Feitelson. A survey of scheduling in multiprogrammed parallel systems. Technical Report RC 19790, IBM Research Division, October 1994.

[8] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors Volume I: General Techniques and Regular Problems.* Prentice Hall, 1988.

[9] D. Gelernter and D. Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with Piranha. Technical Report RR-883, Department of Computer Science, Yale University, 1991.

[10] J. Leon, A. Fisher, and P. Steenskite. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, School of Computer Science, Carnegie Mellon University, February 1993.

[11] S. Leutenegger and M. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proc. of Sigmetrics '90*, May 1990.

[12] M. Litzkow, M. Livny, and M. Mutka. Condor: a hunter of idle workstations. In *Proc. of 8th Intl. Conf. on Distributed Computing Systems*, pages 104–111, June 1988.

[13] C. McCann and J. Zahorjan. Processor allocation policies for message-passing parallel computers. In *Proceedings of 1994 ACM Sigmetrics Conference*, pages 19–32, Nashville, May 1994.

[14] J. Moreira, V. Naik, and R. Konuru. A System for Dynamic Resource Allocation and Data Distribution. Technical Report RC 20257, IBM Research Division, November 1995.

[15] J. Moreira, V. Naik, and R. Konuru. Designing Reconfigurable Data-Parallel Applications for Scalable Parallel Computing Platforms. Technical Report RC 20455, IBM Research Division, May 1996.

[16] M. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, 12:269–284, 1991.

[17] V. K. Naik, S. K. Setia, and M. S. Squillante. Performance Analysis of Job Scheduling Policies in Parallel Supercomputing Environments. In *SuperComputing '93*, November 1993.

[18] Ohio Supercomputer Center. *MPI Primer / Developing with LAM*, November 1996.

[19] J. Plank, Y. Kim, and J. Dongarra. Algorith-based diskless checkpointing for Fault-tolerant Matrix Operations. In *Proc. of 25th Intl. Symposium on Fault-Tolerant Computing*, pages 351–360, June 1995.

[20] J. Pruyne and M. Livny. Parallel Processing on dynamic resources with CARMI. In D. Feitelson and L. Rudolph, editors, *Job scheduling strategies for parallel processing*, volume 949 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[21] E. Seligman and A. Beguelin. High-level Fault Tolerance in Distributed Programs. Technical Report CMU-CS-94-223, School of Computer Science, Carnegie Mellon University, December 1994.

[22] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proc. of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[23] K. A. Yelick. Programming Models for Irregular Applications. In *Proceedings of the Workshop on Languages, Compilers and Run-Time Environments for Distributed Memory Multiprocessors*, volume 28(1). ACM SIGPLAN Notices, January 1993.