The Interaction of Parallel and Sequential Workloads on a Network of Workstations

Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat,
Lok T. Liu, Thomas E. Anderson, and David A. Patterson

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

Abstract

This paper examines the plausibility of using a network of workstations (NOW) for a mixture of parallel and sequential jobs. Through simulations, our study examines issues that arise when combining these two workloads on a single platform. Starting from a dedicated NOW just for parallel programs, we incrementally relax uniprogramming restrictions until we have a multi-programmed, multi-user NOW for both interactive sequential users and parallel programs. We show that a number of issues associated with the distributed NOW environment (e.g., daemon activity, coscheduling skew) can have a small but noticeable effect on parallel program performance. We also find that efficient migration to idle workstations is necessary to maintain acceptable parallel application performance. Furthermore, we present a methodology for deriving an optimal delay time for recruiting idle machines for use by parallel programs; this *recruitment threshold* was just 3 minutes for the research cluster we measured. Finally, we quantify the effects of the additional parallel load upon interactive users by keeping track of the potential number of *user delays* in our simulations. When we limit the maximum number of delays per user, we can still maintain acceptable parallel program performance. In summary, we find that for our workloads a 2:1 rule applies: a NOW cluster of approximately 60 machines can sustain a 32-node parallel workload in addition to the sequential load placed upon it by interactive users.

1 Introduction

Exploiting idle resources in a network of workstations (NOW) has been a popular topic for many years. Research efforts have aimed at using these idle cycles for sequential load sharing [Theimer et al. 1985, Douglis & Ousterhout 1991], and today production load sharing systems [Mutka & Livny 1991, Zhou et al. 1992] are widely available. As one large example, Sun Microsystems uses DrEAM [Delisle 1994] on a cluster of 350 dedicated workstations to run many independent simulations.

This work is supported in part by the Advanced Research Projects Agency (F30602-95-C-0014), the National Science Foundation (CDA-94 01156), California MICRO, the AT&T Foundation, Digital Equipment Corporation, Hewlett Packard, Intel, Siemens Corporation, Sun Microsystems, SynOptics, Thinking Machines, and Xerox Corporation. Anderson was also supported by a National Science Foundation Presidential Faculty Fellowship. The authors can be reached at: (remzi,dusseau,vahdat,ltliu,tea,patterson)@CS.Berkeley.EDU

To appear in SIGMETRICS/PERFORMANCE '95

There has long been widespread interest in using NOWs for parallel computing [Gelernter 1985, Kronenberg et al. 1986, Carriero & Gelernter 1989, Sunderam 1990, Blumrich et al. 1994, Cox et al. 1994, Reinhardt et al. 1994]. In the last few years massively parallel processors (MPPs) and NOWs have become more alike. MPPs are using the same processors, memory, and even operating systems found in workstations, while NOWs have started to use switches in their local-area networks (LANs) similar to MPP-style networks [Anderson et al. 1993, Boden et al. 1995]. Thus, MPPs are trying to take advantage of the workstation price-performance curve, while the disparity between LAN and MPP network performance is being addressed by recent LAN developments.

Although both sequential load sharing and parallel processing have been attempted on a variety of systems, combining the two on a single platform has gone largely ignored. The goal of our research is to build a NOW that runs parallel programs with the same performance as a dedicated MPP and runs sequential programs with performance equivalent to a dedicated uniprocessor. In this paper, we quantitatively examine the impact of mixing parallel and sequential workloads on a NOW. We employ both trace-driven and direct simulation to evaluate the impact sequential and parallel jobs may have on one another. Starting from a dedicated NOW just for parallel programs, we incrementally relax the restrictions until we have a multiprogrammed, multiuser NOW for both interactive sequential users and parallel programs. At each step along this path we focus on measurements that help answer policy questions for several aspects of the NOW system.

Combining sequential and parallel workloads on a NOW introduces a new set of resource management policy concerns. To provide desktop performance equivalent to a dedicated uniprocessor, we assume the system must deliver the equivalent of a dedicated workstation to each active user. (One example of what happens when this issue is ignored comes from SRC's distributed load balancing system, *dp*, where many users would periodically tap their keyboards to prevent their workstation from being harvested by the system.) If parallel programmers are going to use idle workstations in a NOW, they must receive guarantees of predictable performance similar to a smaller but dedicated MPP or workstation cluster. We show that relatively simple techniques can maintain both interactive response times and parallel program throughput on a NOW.

Some of our results confirm intuition. For example, the literature demonstrates that it is crucial to coschedule parallel programs to achieve acceptable performance [Ousterhout

1982, Gupta et al. 1991]. We show that this requirement also holds true on a NOW by quantifying the impact of not coscheduling. Other results run counter to popular opinion. Although the set of idle machines changes over time, the total *number* of idle machines stays relatively constant for our traces, even during the busiest time of the work-day. Consequently, we can use relatively simple scheduling policies on the NOW: when a machine running a program goes from idle to active, another idle machine is likely to be available for running the dislocated process. By measuring machine availability in our target cluster, we quantify the need for efficient process migration in a non-dedicated environment. Further, by limiting the number of times per day that users discover a parallel program on their machine, we can spread the workload more evenly across machines as well as reduce the worst-case number of delays seen by a given sequential user. For our traces, we find that a 2:1 rule applies: a NOW cluster of approximately 60 machines can easily sustain a 32-node parallel workload in addition to the sequential load placed upon it by interactive users.

The paper is organized as follows. Section 2 describes our methodology. In Section 3, we describe some of the problems associated with running parallel jobs in a distributed environment. Section 4 adds consideration of a non-dedicated environment where processors running a parallel job may be interrupted at any time. Section 5 quantifies the effects the parallel workload may have on interactive users. The paper concludes with related work and a summary of our results.

2 Methodology

In this section, we give an overview of the experimental method used in this study. Two complimentary styles of simulation are employed. The first, *direct simulation*, is a technique that measures the effects of various scheduling perturbations on real parallel applications. The second, *trace-driven simulation*, interleaves traces from both a parallel machine and from a network of workstations to discover the impact on parallel program throughput as well as the potential effect upon workstation users. By utilizing both of these simulation techniques (an example of *hierarchical modeling*), we are able to better understand the interaction of the two workloads on a NOW.

2.1 Direct Simulation

To measure the effects of disturbances on real parallel applications, we use direct simulation. This technique measures the run time of a real parallel program executing on a massively parallel machine, while varying parameters such as scheduling policies, interaction with other simulated parallel jobs, and interaction with simulated sequential jobs. The simulation testbed is a 64-node Thinking Machines' CM-5, a message-passing MPP [Leiserson 1992].

We start with a simple example. To simulate the interaction of a sequential application running on one processor while a parallel application runs on 64 processors, we periodically interrupt one processor and force it to execute an idle loop. This "disturbance" appears to the parallel program as a sequential program using a time-slice on one of the processors. By measuring the execution time of the parallel application, we can observe the slowdown induced by the simulated sequential application. In all experiments, we generate each data point five times, reporting the average.

In our simulations, messages which arrive for a non-scheduled process are buffered in a small, fixed-size buffer until the target process is scheduled. Currently, we do not model network or memory traffic congestion. Cache effects due to time-slicing might also affect some results, but we have found that the effect is amortized by using a longer time-slice.

We composed a suite of five parallel applications, written in Split-C [Culler et al. 1993a] for direct simulation. The first, `cholesky` , performs LU factorization on sparse, symmetric matrices. `Column` is an implementation of the column-sort algorithm [Leighton 1985]; a description of the implementation can be found in [Culler et al. 1994]. The program `em3d` simulates the propagation of electro-magnetic waves through objects in three dimensions [Culler et al. 1993a]. A version of the sample sort algorithm [Blelloch et al. 1991] is implemented by `sample`. Finally, `connect` uses a randomized algorithm to find the connected components of an arbitrary graph [Krishnamurthy et al. 1994].

These benchmarks represent a cross-section of parallel applications with different message sizes, and with different communication and synchronization frequencies. The average message size in `column` is 20 Kbytes, whereas the other algorithms communicate using small messages (between 20 and 70 Bytes on average). The two sorting algorithms, `column` and `sample` are bulk-synchronous, alternating between computation and communication phases that each last on the order of seconds, synchronizing only between phases. `Cholesky` communicates at a medium granularity, computing for several hundred microseconds between message sends, and synchronizing frequently between pairs of processors. The two most fine-grain applications, `connect` and `em3d` communicate every 40 and 100 μs, respectively. `Em3d` performs a barrier synchronization across processors every 8 ms, compared to an interval in the 100s of milliseconds in the other applications.

2.2 Trace-driven Simulation

While the direct simulation technique is appropriate when studying the behavior of a single parallel application, the approach quickly becomes intractable when examining the execution of many parallel jobs. For this reason, we use a trace-driven simulator to further investigate the combination of parallel and sequential workloads on a single NOW.

For our study, we first collected traces from a workstation cluster used by the U.C. Berkeley CAD group. The cluster consists of 53 DECstation 5000/133s each with 64 MB of memory and is used by graduate students in the electrical engineering department. Two user-level daemons logged information every two seconds on CPU, memory, disk, keyboard and mouse activity. Data was collected for two months (February and March, 1994), resulting in roughly 3000 workstation-days of traces. The workstation traces used in these simulations are randomly selected from different weekday traces, allowing us to simulate a cluster of more than 53 workstations.

For our parallel machine trace, we obtained a month's worth of data from the CM-5 at Los Alamos National Laboratories. The data consists of submission time, CPU time, number of nodes and amount of memory utilized by each of the jobs executed during the month of October, 1993. The LANL CM-5 has nodes statically partitioned into sizes of 512, 256, 128, 64, and two of size 32. Jobs at LANL are divided into two classes: development and production. To capture the effects of both classes of jobs, we use traces from one of the 32 node partitions, which consists of a mix of production and development runs. We use a trace from

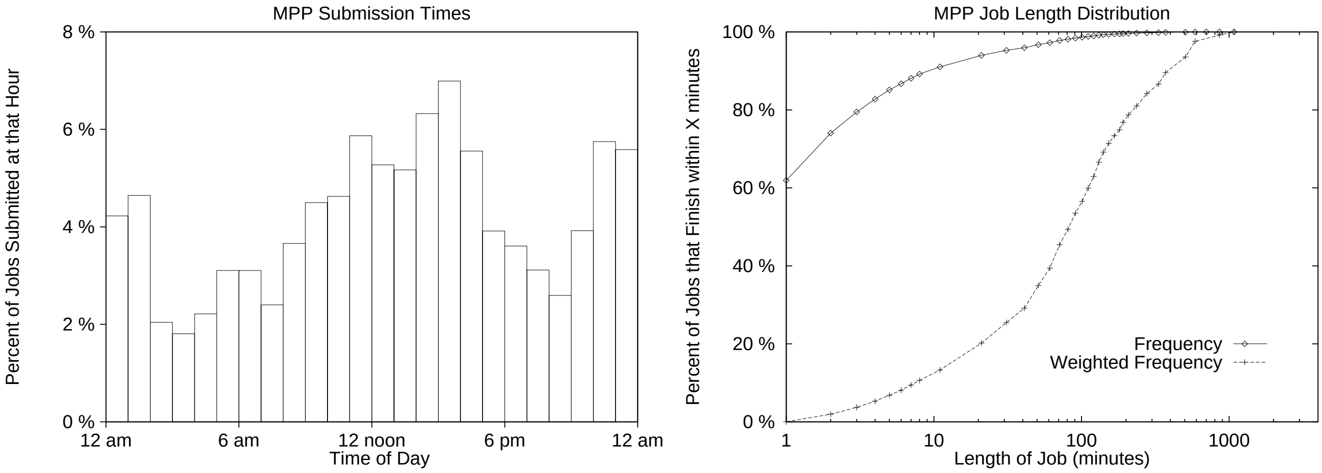


Figure 1: *LANL Job Characteristics.* The leftmost diagram shows the percent of all jobs submitted at a particular hour. The rightmost diagram shows both the frequency and weighted frequency of job lengths; for example, while almost 80% of jobs are 3 minutes or less in length, jobs that run for 90 minutes or more account for over 50% of the total execution time. Traces are taken from all partitions.

a particularly heavy day of submissions to examine "worst-case" behavior. On this day, the LANL partition was 75% utilized, more than twice as much as the average day.

The workload combination results presented in this paper are obtained by using the trace-driven simulator to schedule the LANL parallel workload on top of the sequential workload presented by the workstations. Parallel jobs are run one at a time in a round-robin fashion with a two second time quanta. We use the matrix algorithm [Ousterhout 1982] to schedule the parallel jobs.

The simulator produces two metrics for evaluating the performance of a NOW: normalized parallel program slowdown and the number of times a user is delayed by parallel programs. Non-normalized slowdown (a.k.a. response time) is defined as the simulated "wall clock" time of a job divided by its CPU time. Slowdown can be caused by multiprogramming among parallel jobs as well as the effects of the non-dedicated NOW environment. To isolate the latter, slowdown on a NOW is normalized by dividing by the slowdown on a dedicated MPP. For example, if a job experiences a slowdown of 2 due to queuing delay on an MPP and a slowdown of 3 on a NOW, we report a normalized slowdown of 1.5.

Delays to everyday workstation users are the other main effect of running parallel programs on a NOW. For example, on returning to their workstations, users may notice a significant delay because their machine's virtual memory and file cache were flushed by the execution of a parallel program. Therefore, our simulation tracks the number of such potential user delays to quantitatively determine the effect of different scheduling policies.

3 Dedicated Decentralized System

It has been suggested that idle resources in a NOW can be harvested during off-hours to provide a "supercomputer by night". However, our traces indicate that such an arrangement would be unacceptable to LANL's user community. The importance of maintaining interactive response time for parallel users is shown in Figure 1. Roughly half of the parallel jobs are submitted during regular working hours, and 80% of the jobs run for three minutes or less, indicating that fast response times are important to parallel users as well as sequential users in a NOW.

The first step in moving parallel applications from an MPP environment to a multi-programmed, multi-user NOW is to run the applications on a dedicated NOW, i.e., a cluster devoid of interactive users. This environment differs from an MPP in that each workstation runs a full, independent operating system and the interconnect may provide performance inferior to an MPP. In this first section, we examine how differences in the two environments—such as scheduling policy, clock skew, local daemon activity, and communication performance— affect the execution time of parallel applications.

3.1 Coscheduling versus Local-scheduling

MPPs, such as the CM-5 or Intel Paragon, have operating systems which are specialized for scheduling parallel applications; on the other hand, networks of workstations have independent UNIX kernels on each processor. Scheduling techniques for MPPs are well understood: coscheduling [Ousterhout 1982] parallel applications generally results in good parallel program performance [Crovella et al. 1991, Gupta et al. 1991, Feitelson & Rudolph 1992]. In coscheduling, all processes of one parallel job are scheduled simultaneously, with coordinated time-slicing between different parallel jobs.

One approach to scheduling parallel jobs on a NOW is to allow the underlying UNIX kernel on each workstation to schedule the parallel applications independently; we term this *local-scheduling*. This policy, employed by parallel environments such as PVM [Sunderam 1990], has the advantage that no kernel modifications are required. However, earlier studies [Crovella et al. 1991, Feitelson & Rudolph 1992] demonstrate that local-scheduling leads to unacceptable execution times for frequently-communicating processes. The slowdown occurs because a process stalls when it communicates or synchronizes with a non-scheduled process; in our environment, messages which arrive for non-scheduled processes are buffered and responses are returned only after the destination process is scheduled.

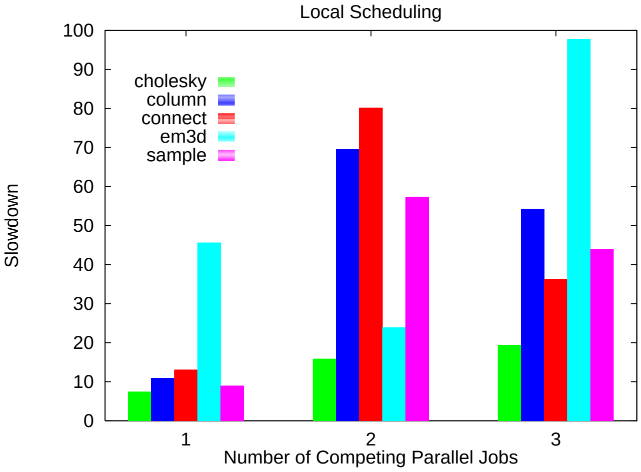


Figure 2: *Local Scheduling versus Coscheduling.* One real parallel application and one to three simulated parallel jobs are scheduled in a round-robin fashion with a 100 ms time quantum. Slowdown is the ratio of the measured locally-scheduled execution time to the coscheduled time.

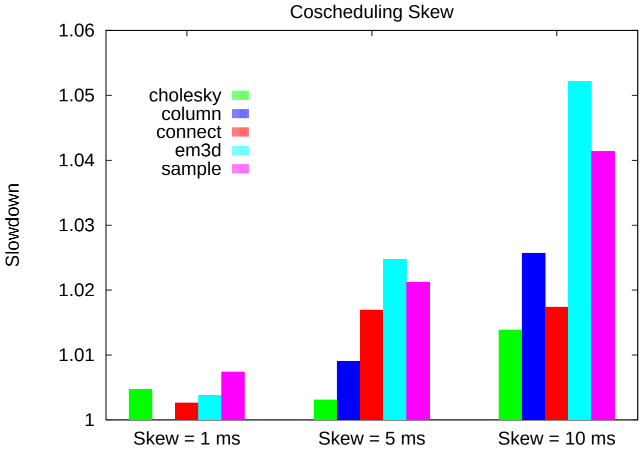


Figure 3: *Coscheduling Skews.* Each parallel application is coscheduled with a simulated parallel job with a time quantum of 100 ms. The slowdowns of the applications with coscheduling skews of 1, 5, and 10 ms are reported.

We reproduce this performance result for local-scheduling on our five benchmark applications using direct simulation. Figure 2 shows the slowdown of locally-scheduling each application in a round-robin fashion while varying the number of simulated competing parallel jobs between one and three. The reported slowdown is relative to the execution time of the applications when coscheduled. Our results show that the performance of locally-scheduled parallel applications is incredibly poor. When resources are shared between two parallel applications, each application is slowed down by at least a factor of eight. As expected, the application that synchronizes the most frequently, `em3d`, is slowed down the most significantly, i.e., by nearly 50 times. Even the performance of `column` (which communicates infrequently) is severely degraded as a result of insufficient buffering on the destination processor for the very large messages (20 Kbytes on average).

As the number of competing parallel applications is increased from two to four, the execution times increase for two reasons. First, when communicating with a destination process, the likelihood that the destination process is not scheduled increases with more competing processes. Second, if a destination process is not scheduled, then the time remaining before that process is scheduled increases. For similar reasons, increasing the time quantum length also results in greater slowdowns.

Thus, we have found that parallel applications are slowed down dramatically when locally scheduled with one or more other parallel applications. Coscheduling of parallel applications guarantees an acceptable level of performance regardless of the number of competing applications. Coscheduling also allows longer time quanta to be used to amortize cache effects and scheduling overheads.

3.2 Coscheduling Skew

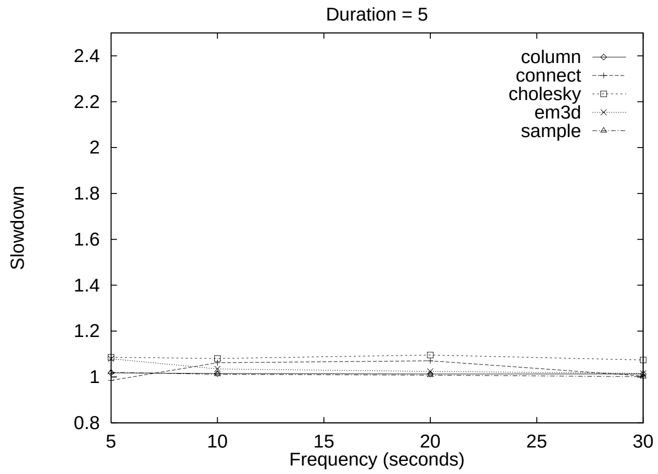
In the previous section, we assumed perfect coscheduling—that is, context switches across processors began and ended at precisely the same moment. Such coscheduling is possible in an MPP because a common clock may be distributed to all nodes. However, it is unlikely that a mechanism for perfect coscheduling will exist in a physically distributed network of workstations. For example, if context switches are coordinated by broadcasting a signal from a master workstation, different processes may receive the signal at different times. Even if coordination exists because the clocks of the workstation have been synchronized [Lamport 1990], it is likely that some clock skew exists across workstations. In this section, we quantify the effect of imprecise coscheduling on parallel application performance.

We define *coscheduling skew* as the maximum difference between the quantum start times across processors. In our simulations, we coschedule a parallel application with one simulated parallel job, offsetting the start time of each processor's time quanta by a random amount. Figure 3 shows that with a time quantum of 100 ms, the execution time of the application increases with the coscheduling skew; with a 10 ms skew, some applications are slowed down by 5% when compared to the perfectly coscheduled time. Increasing the time quantum to one second eliminates the trend of rising execution times; with a 10 ms skew, application are slowed down less than 2%.

In conclusion, a NOW should use either a large time quanta (1 sec) or small coscheduling skew (less than 1 ms) to achieve negligible slowdowns. Even with a smaller quantum (100 ms) and a larger skew (10 ms), the slowdown felt by parallel programs is at most 5%. Depending on the particular circumstances, this may or may not be acceptable.

3.3 Daemon Processes

Even a network of workstations devoid of interactive users has some serial processes: those associated with the operating system which must run periodically. Table 1 shows measurements of the average run time and inter-arrival times for the most frequently running daemons on a DECstation 5000 running Ultrix 4.2a. These measurements indicate that a daemon process runs approximately once every 10 seconds for up to 5 milliseconds on each workstation. Assuming these daemon processes execute across workstations at non-overlapping intervals, then on a cluster of 64 workstations, a daemon process is executing 3% of the time. If, due to communication dependencies, parallel applications can not



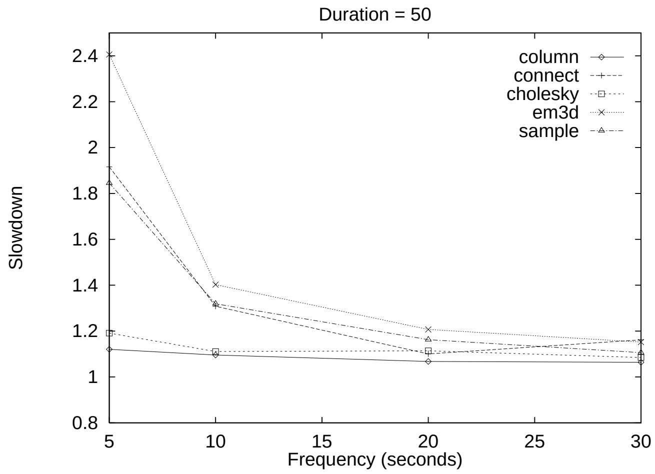


Figure 4: *Slowdown with Independent Daemon Processes.* The duration of the daemon processes is 5 ms in the left-most graph; 50 ms in the right-most graph. The inter-arrival time of daemon processes is varied between 5 and 30 seconds. For characteristic daemon activity, i.e., inter-arrival times of 10 seconds and durations of 5 ms, parallel applications experience slowdowns less than 10%.

Daemon	Run Time (ms)	Interval (s)
`cron`	3 to 6	60
`update`	3 to 5	30
`routed`	1 to 2	25
`Xdec`	1	25
`init`	1	200
others	negligible	negligible

Table 1: *Daemon Activity on a DECstation 5000 running Ultrix 4.2a.* This table shows the average execution times and intervals between invocations of daemon processes as measured over a three hour period when the machine was idle of all user activity. `Cron` wakes up and runs jobs periodically; `update` syncs the disks; `routed` is the network routing daemon; `Xdec` is the X11 server; lastly, `init` is the parent of all other processes.

make progress when a daemon is executing on one of the workstations, then it may be beneficial to coschedule daemon processes. In this section, we examine the effects of locally-scheduled daemon processes on parallel application performance.

We simulate the effect of daemons arriving independently across workstations at intervals of 5, 10, 20, and 30 seconds for durations of 5 ms. Our results in Figure 4 show that parallel applications are slowed down up to 10% with interruptions characteristic of current daemon activity. Interestingly, with larger interruption durations of 50 ms, most applications see a significant increase in execution time; in the most extreme case, `em3d`, the application is slowed down by more than a factor of two. Thus, if periodic system activity occurs with execution times near 50 ms, then our results indicate that coscheduling of such activity is necessary. It is perhaps ironic to note that this increased activity may be caused by the daemons required to build a NOW (i.e., those used to monitor workstation availability).

3.4 Local-Area Networks

Bus-based local-area networks, such as Ethernet and FDDI, are not able to support the aggregate bandwidth and low latency needed by many MPP applications. For this reason, NOWs have historically only run coarse-grained parallel programs. The recent introduction of switch-based LANs, such as ATM [Biagioni et al. 1993], switched Ethernet, AN-2 [Anderson et al. 1993], and Myrinet [Boden et al. 1995], increases the feasibility of running general-purpose parallel applications on NOWs. For example, the delivered bandwidth on today's SynOptics, Fore, and Cisco ATM equipment is comparable to the delivered 10 MB/s bandwidth of Thinking Machines CM-5. Similarly, Myrinet is effectively an MPP backplane repackaged as a LAN, and offers the same bandwidth in both environments.

Communication overheads and latencies in LANs, however, appear to lag those in MPP networks by a generation. As defined by the LogP [Culler et al. 1993b] model, *overhead* is the length of time that a processor is engaged in the transmission or reception of a message; *latency* is the upper bound on the delay from the source processor to the destination. Today, communication protocols such as TCP/IP add unnecessary overhead to the base hardware cost. Current research implementing Active Messages [von Eicken et al. 1992] on NOWs has shown that overheads can be reduced to approximately $10\mu s$ [von Eicken et al. 1994, Martin 1994], which is much closer to the $2\mu s$ Active Message overhead on the CM-5 than the milliseconds of overhead in TCP/IP implementations. Likewise, latencies also appear to be converging. MPP routers have much lower latency than current ATM switches; however, this imbalance is partially ameliorated by the fact that MPP routers are typically much smaller than LAN switches; therefore, more hops are required to cross an MPP network.

Scaling a parallel application's execution time to account for this difference in communication performance is a nontrivial problem, requiring knowledge not only of the number and length of message transfers, but also of the overlap between communication and computation. Furthermore, any scaling should take into account the increased processing power of modern workstations compared to the processing

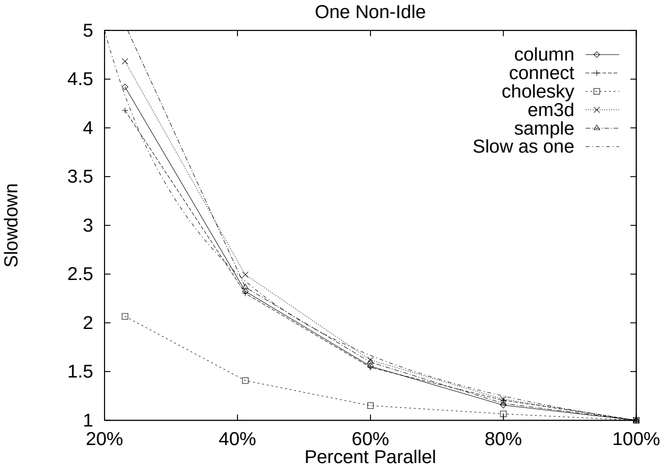


Figure 5: *One Non-Idle Workstation.* The parallel application is time-sliced in a round-robin fashion with a sequential process on one workstation out of 64. The sequential process is always given a 100 ms time quantum. The x-axis is the percentage of time the parallel application runs on the non-idle node.

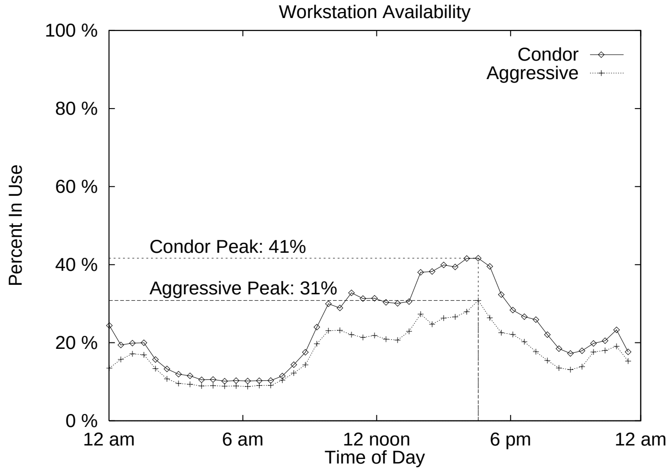


Figure 6: *Machine Availability.* Depending on the definition of available, at least 60 to 70% of the machines are available at any given time. The *Aggressive* definition requires one minute of low CPU and no keyboard activity, whereas the *Condor* definition waits for 15 minutes before declaring a machine idle.

nodes of most MPPs. Because of these difficulties, we feel that scaling parallel program execution times would result in artificial results. Also, further advances in LAN technologies may lead to MPP-like network performance, reducing the need to introduce any scaling factors. However, we consider this an interesting area for further research.

4 Non-Dedicated System

The next step in building a realistic NOW from an MPP is to integrate the scheduling of jobs from interactive users with our parallel applications. In the following sections we show how interactive jobs affect the performance of parallel applications, demonstrating the need to migrate parallel processes away from non-idle workstations. We then explore the effect of migration time on the throughput of parallel applications. Finally, we define and quantify the effect of the *recruitment threshold* on parallel program throughput.

4.1 One Interactive User

We assume that users demand interactive performance from their workstations even when they are sharing resources with parallel applications. The policy resulting in the best compromise between serial-job response time and parallel-application throughput is likely to depend upon a number of factors, such as the number of idle workstations, the total serial workload, and the characteristics of the parallel applications. Our first step in determining the best policy for resource sharing is to examine the effect of interruptions on the execution time of our parallel benchmarks through direct simulation.

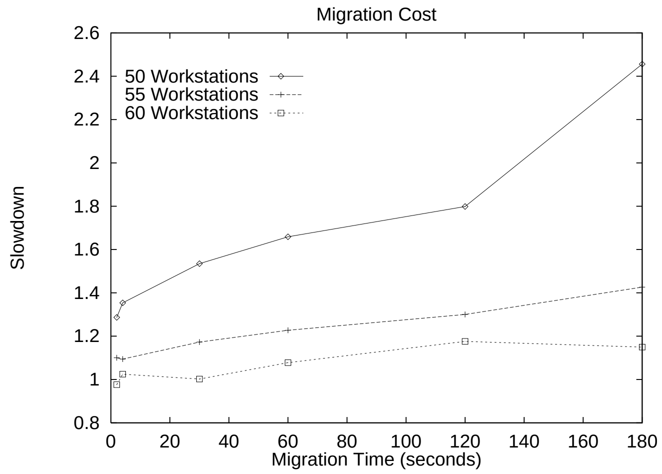
Figure 5 plots the slowdown when a parallel application executes on 64 workstations while one workstation is shared with a sequential user. The parallel application runs 100% of the time on the 63 idle workstations and is time-sliced in a round-robin fashion with the sequential process on the one non-idle workstation.

We found that application behavior falls into two categories. If the work is distributed evenly across processors (as in `column`) or if global synchronization is performed frequently (`em3d`), then all processors must wait for the slowest processor to complete. This forces the application to run at the rate of the non-idle workstation, as is shown in the graph. On the other hand, if a load-imbalance exists across processors and a processor with less work than the others is periodically interrupted, then the application is not slowed down to the speed of the slowest workstation. This is the case for `cholesky`. A load-imbalance also exists in `connect` and `sample`, but the interrupted workstation was already one of the slowest and so the performance curve follows this processor. If another workstation had been picked, the slowdown would not have been as poor.

To summarize, if a node running a parallel process begins executing a serial job, the parallel process should be moved to another idle workstation, although the performance effect is severe only for some applications. This migration is necessary to maintain acceptable parallel program execution times; detrimental effects on sequential response times are also present if the parallel program is not migrated. Of course, migration is a viable solution only if another idle workstation is available. Figure 6 shows that such availability is likely on our traced cluster. This result is in direct contrast to the popular belief that idle machines are only available during off hours [Lazowska 1994]. Even during the daytime hours, more than 60 to 70% of workstations are available at any given moment.

4.2 Migration Costs

We have seen that workstation cycles are available at all times of the day. A system with no migration overhead could utilize every one of those cycles. However, since migration can be costly, we next examine the effect of migration costs on parallel throughput. The benefit of migration depends upon a number of factors, such as the remaining execution time of the parallel application, the cost of migrating the process, and the time until the process is migrated again. To understand these interactions, we feed our high-level simulator with actual workstation and parallel job traces, mea-



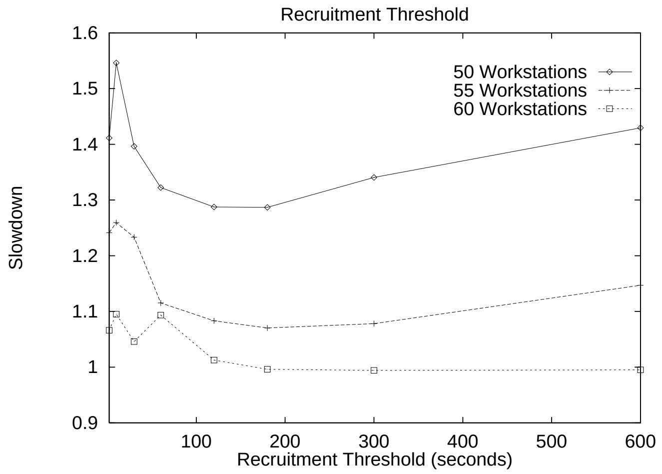


Figure 7: *Process Migration and Recruitment Thresholds.* The figure on the left demonstrates parallel program slowdown as a function of the cost of process migration. These results were generated using the trace-driven simulator, on clusters of 50, 55, and 60 nodes with a 32-node load of LANL jobs. Note that parallel jobs are migrated upon first detection of user activity at a workstation. On the right, the figure shows parallel program slowdown as a function of the recruitment threshold, assuming a two-second migration cost. The best slowdown is achieved at thresholds where a machine is likely to remain idle for a long period of time without wasting too many idle cycles.

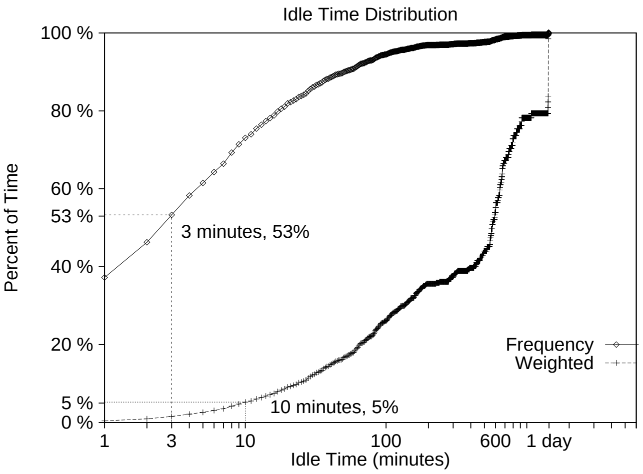


Figure 8: *Idle Time Distribution during the Day.* The *Frequency* line shows how different idle periods are distributed, with a line indicating that 53% of idle periods are three minutes or less. The *Duration* line multiplies frequency by duration. From that we see that 95% of idle time is spent in periods of time that are 10 minutes or longer. Measurements were taken from 3 weekdays days of activity from the CAD group cluster.

suring parallel program throughput as a function of the cost of process migration. Note that the simulator makes the conservative assumption that when one process is being migrated, the other processes of the parallel application make no progress.

One hypothesis is that since migration is the rare case [Eager et al. 1986, Douglis & Ousterhout 1991], its performance need not be optimized. Figure 7 shows that parallel programs on a 50-node cluster with high migration costs suffer a slowdown of two times or more than systems with lower migration costs. This slowdown can be attributed to the larger "dead time" the parallel program experiences while the program waits for the migration to complete before continuing. For example, some versions of the Condor system require two minutes for migration [Litzkow & Solomon 1992], corresponding to a slowdown of nearly 1.8 on a 50 node cluster. High migration costs can also have negative effects on interactive users as they may have to wait longer for parallel processes to be evicted from their workstations before resuming their work. Note that the effect is not nearly as dramatic in clusters with more workstations. Since more resources (and hence more idle machines) are available, migration occurs much less often, minimizing the impact of its cost.

4.3 Recruitment Threshold

Using a typical definition for machine availability (i.e., low load average and lack of keyboard activity) similar to [Mutka & Livny 1991, Douglis & Ousterhout 1991], we can also quantitatively derive the *recruitment threshold*, the amount of time the system waits before harvesting an available machine. Intuitively, choosing a small (perhaps 0-second) recruitment threshold would maximize parallel program performance. Douglis and Ousterhout [Douglis & Ousterhout 1991] demonstrate that machines that have only recently gone idle should be avoided since they are most likely to once again become unavailable. Our simulations confirm this result. Figure 7 plots parallel program slowdown as a function of recruitment threshold. For our traces, a 180 second recruitment threshold maximizes parallel program throughput on all three clusters. This value ensures that the machine is likely to remain available, and yet does not lead the system to squander a large amount of time that could have been used to run parallel programs.

Figure 8 gives some insight as to why a 3-minute wait period before recruitment is tolerable, even during daytime hours. Though half of all idle periods are 3 minutes or less, over 95% of idle *time* is spent in intervals of 10 minutes or more.

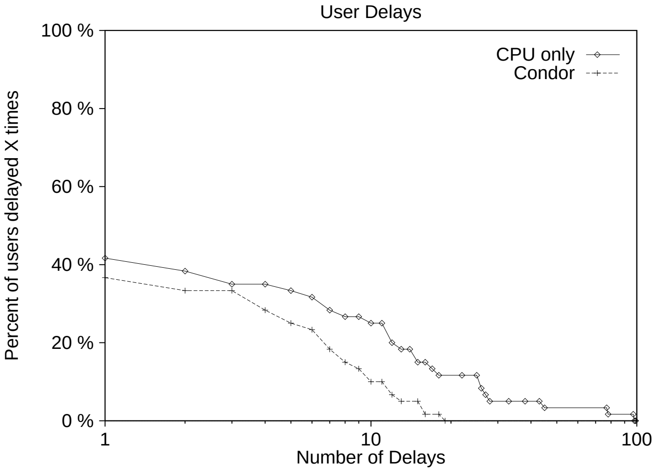


Figure 9: *Delays Per Day.* The x-axis shows the number of times a user is delayed, and the y-axis shows the percent of times a user is delayed x times. The *CPU only* line uses a definition that runs jobs on machines when there is little or no CPU activity, while the *Condor* line uses a conservative definition that is sensitive to both the CPU and keyboard input.

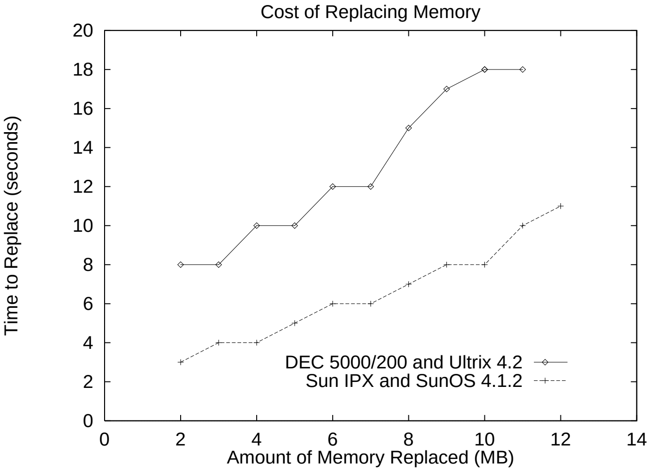


Figure 10: *Potential Cost per Interruption.* A program which utilized a varying amount of memory was run, then stopped. A cleaner that uses all of physical memory was then run. After it completed, the original program was continued, and the time to page in the working set was measured by monitoring local disk activity.

Program	Original Run Time	Slowdown (FC)	(M)	(both)
LaTeX	15.0 s	1.08	1.27	1.46
`make`	42.0 s	1.31	1.31	1.39
`gzip`	23.3 s	1.01	1.09	1.09
`user`	101.7 s	1.09	1.02	1.13

Table 2: *Replacement Effects.* This table displays the effects of cleaning the file cache (FC), memory (M), or both on the run time of each of the benchmarks. LaTeX formatted a 12000-word set of input files, `make` built the tcsh 6.03 distribution, `gzip` compressed a 1.2 MB file, and `user` is a shell script composed of typical Unix commands. Experiments were performed on a DECstation 5000/200, Ultrix 4.2a.

5 Time-shared System

In this section, we consider the effects that parallel programs may have on a NOW's interactive users. Though parallel processes will be migrated away when the user returns to a workstation, they are likely to notice the heavy parallel workload in the form of extra paging activity, slower response times, and decreased application performance. In this section, we quantify how often users may notice the "extra" parallel load, estimate potential costs per interruption, and propose a *social contract* for reducing annoyances to interactive users to a tolerable level.

5.1 User Interruptions

In order for a NOW to be successful, the number of user delays must be kept to a bare minimum. Here, *user delays* are defined as the number of times when interactive users return to their workstation to find a parallel job running on their machines or their workstation state changed by a parallel job that ran to completion. These delays consist of two components: the time to import the user's previous context and the time to export the parallel job from the machine (if there is one running at the time of the user's return). A machine *recruitment policy* identifies when a workstation is available to run a parallel job. For example, one policy might consider a machine available if its average CPU utilization is less than 20% for at least one minute.

Figure 9 shows a cumulative graph on how two recruitment policies affect the number of delays to interactive users. The first recruitment policy, *CPU only*, classifies a machine as available if its average CPU utilization is less than 20% for a one minute period. With such a policy, one unlucky user would be delayed 80 times during the day on our measured workload! The second recruitment policy uses a method similar to that in Condor [Mutka & Livny 1991]: a machine is available if it's average CPU utilization is less than 20% and there is no keyboard activity, both for a minimum of 15 minutes. This user-sensitive policy does significantly better; however, there remain three users who are delayed at least 15 times during the day. We next show that each of these delays can be potentially quite lengthy.

5.2 Potential Cost per Interruption

When users return to their workstation, not only might they have to wait for parallel programs to vacate their machine, but they must also wait for their previous contexts to be restored. Consider the four main resources of the workstation: CPU, main memory, disk, and network interface. Reclaiming the CPU is inexpensive; restoring the state of registers, cache, and TLB costs no more than a few milliseconds. A similar argument can be made for the network interface because little state changes upon a context switch. However, considerably more state is associated with the workstation's main memory and file cache.

Figure 10 shows that replacing main memory pages can be quite costly. Interactive users will quickly become frustrated if they expect near-instantaneous response times and instead must wait for their state to be swapped back each time they resume work on their workstation. Table 2 quantifies the effect of flushing a machine's main memory and/or

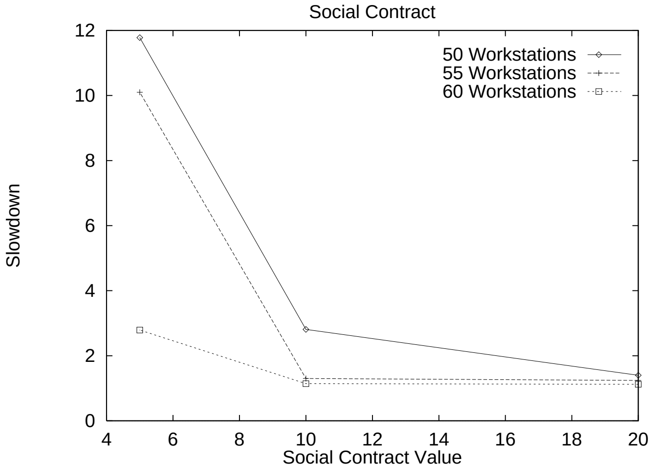


Figure 11: *Social Contract Results.* The graph shows the slowdown of parallel programs as a function of social contract. The simulation assumes a 2-second migration cost.

file cache. Flushing either one can have significant effect on the execution time of some typical UNIX programs. Depending on the application, slowdowns up to 46% were observed.

This problem has many possible solutions: avoiding recruitment of machines with large amounts of active memory, pinning "important" memory pages down (e.g. the X server, emacs code pages, etc.), or using bulk parallel I/O to recover the interactive user's state quickly. The idea of recruiting remote memory as a fast backing store could also be of help [Dahlin et al. 1994, Felten & Zahorjan 1991]. Having a fixed-memory partition between parallel and sequential jobs [Ashok & Zahorjan 1992] may also help, but this may not be possible without kernel modifications. We plan on exploring some of these alternatives as a part of our future work.

5.3 Social Contract

In this subsection, we demonstrate that a very simple scheduling mechanism can vastly reduce the worst-case number of user delays without adversely affecting the performance of parallel programs. A *social contract* guarantees that an individual will not be delayed more than a specified number of times during any day. Once a user has been delayed the threshold number of times, that user's workstation will no longer be a candidate to run parallel programs.

Figure 11 shows the slowdown of parallel programs for the heavy LANL workload as a function of the social contract value. The high resource demands of the parallel workload forces a tradeoff: as the number of tolerated user interruptions decreases, parallel performance worsens. On the 60-node cluster, if users are willing to tolerate 10 daily interruptions then parallel program slowdown is at a reasonable factor of 1.14. On the 50-node cluster, however, users must be willing to tolerate 20 such interruptions to achieve a slowdown equal to 1.4.

5.4 NOW Cluster Size

The previous subsections demonstrate that our target cluster of 60 workstations can comfortably support the parallel workload submitted to the 32-node partition of LANL's CM-5. Unfortunately, as the number of available workstations decreases (or as the parallel workload increases), both the number of interruptions to interactive users and the runtime of the parallel programs increases dramatically. Furthermore, when available resources are tight, process migration cost becomes increasingly important to overall system performance. A similar pattern holds for the system's recruitment threshold: if few workstations are available, then the recruitment threshold must be chosen carefully to maintain overall system throughput.

In summary, for each combination of sequential and parallel workloads, system designers must find the requisite number of workstations to keep interruptions to interactive users at an acceptable level while maintaining reasonable response times for the parallel jobs. On our target network of workstations, a 2:1 rule appears to apply: a cluster of approximately 60 machines can sustain both a sequential and parallel workload. For clusters exhibiting heavier load characteristics, a fixed number of uniprocessors dedicated to running sequential jobs may be added to the network both to increase parallel job performance and to reduce interruptions to interactive users.

6 Related Work

There have been an abundance of studies on workstation clusters and their potential to support distributed and parallel computing. Theimer and his colleagues estimated that on average one-third of their 25 V-system workstations were free, even at the busiest times of the day [Theimer & Lantz 1989]. Nichols measured 15 to 20% of workstations available during the day in the Butler system, which increased to roughly 30% at night [Nichols 1987]. Douglis and Ousterhout found that about two-thirds of machines in Sprite were available on average [Douglis & Ousterhout 1991], and Mutka and Livny [Mutka & Livny 1991] found similar results. A key difference between these widely varying results is the definition of machine availability; for example, Butler defines a machines to be available when no one is logged in, where Sprite waits for only one minute of inactivity before claiming the workstation for job use.

Previous cluster studies have also tended to concentrate on aggregate system availability. We show that other factors are also relevant. The distribution of idle times—that although most idle periods are short, most idle time is spent in large intervals—is also relevant.

There have also been a large number of studies on multiprocessor scheduling techniques; however, most have been based on shared-memory architectures and have used synthetic inputs to drive parallel program behavior. Ousterhout first introduced the idea of *coscheduling* [Ousterhout 1982]. The idea has since been included in many studies of multiprocessor scheduling techniques [Tucker & Gupta 1989, Crovella et al. 1991, Feitelson & Rudolph 1992, Leutenegger & Vernon 1990]. Ousterhout assumed that coscheduling is a good idea, and implemented and compared three algorithms. The matrix algorithm was the simplest and close to the most efficient. For this reason, we chose to implement it in our trace-driven simulator. Gupta et al. [Gupta et al. 1991] found that coscheduling was as good as their space-sharing method (known as *process control*), if the time quantum was long enough (25 milliseconds) to amortize cache effects. Others have studied how to determine which processors and how many processors to allocate to a parallel application[Sevcik 1989, Naik et al. 1993, Chiang et al. 1994, McCann & Zahorjan 1994]. However, the parallel jobs in our study require a fixed number of nodes throughout their life-times; assigning a smaller number of nodes to the job is not possible. Finally, we do not assume *a priori* knowledge of the available parallelism in the programs we schedule.

The impact of memory management has also been scrutinized in a multiprocessor environment [Chandra et al. 1994]. Ashok and Zahorjan [Ashok & Zahorjan 1992] studied mixing an interactive and batch workload onto a parallel, shared-memory supercomputer. They found that partitioning memory statically between batch and interactive jobs performs better than policies that can vary dynamically. Our study differs in that batch and interactive jobs never share the same node. Peris and colleagues also conclude that memory management is an important issue [Peris et al. 1994], and has a noticeable impact on processor allocation schemes (i.e. how many nodes should be given to a particular application). Again, we assume that the number of processors is completely decided by the requesting application, not the operating system.

While forays into merging parallelism and the local area network environment have been made, no effort has been directed at bringing fine-grained parallel applications onto the desktop. The classic example of bringing parallelism to NOWs is the Parallel Virtual Machine [Sunderam 1990], better known as PVM. PVM allows users to run parallel applications on a heterogeneous network of machines. While PVM allows programs with many separate processes to be run, it cannot feasibly run jobs that need to communicate and/or synchronize frequently, because of high message latencies and overheads [Douglas et al. 1993].

In another study of mixing parallel programs into a workstation cluster, [Leutenegger & Sun 1993], simulation is used to study whether parallel applications can run in a non-dedicated environment (such as a NOW). While demonstrating that this may be possible without a significant impact on the parallel programs, their work does not discuss any potential impact upon interactive users. Further, all the simulations are driven by synthetic models of both workstation and parallel program behavior.

7 Conclusions

There is a large body of literature on harvesting idle cycles in a NOW for sequential load sharing, as well as executing parallel programs on dedicated NOWs. Building upon earlier work, we examined the feasibility of combining sequential and parallel jobs on a single platform. We show that parallel programs can tolerate some of the difficulties present in distributed systems (such as coscheduling skew, local daemon processes) with some performance degradation. Enough idle cycles were present in the cluster we traced to effectively support both of our sequential and parallel workloads given a reasonable *recruitment threshold* and an efficient implementation of process migration. The success of the proposed system hinges upon maintaining the response time of the interactive sequential users of the cluster. Thus, parallel jobs were only run on otherwise inactive machines. However, because of secondary memory effects, moving a parallel process away from a workstation can potentially be quite costly to interactive users upon their return. A *social contract* can be used to minimize the number of interruptions to any one sequential user while still maintaining parallel throughput. For our traces we found that a 2:1 rule applies: a NOW cluster of 60 machines can sustain a 32-node parallel workload in addition to the sequential load placed upon it by interactive users.

8 Acknowledgements

We would like to thank Jerry DeLapp of LANL for the CM-5 traces, as well as Brad Krebbs of the CAD group for his assistance in gathering the workstation characteristics. We also would like to thank Doug Ghormley for his detailed study of daemon processes. Lastly, we would like to thank Ann Chervenak, David Culler, Mike Dahlin, John Ousterhout, and Rich Martin for their comments on various drafts of this paper.

References

[Anderson et al. 1993] Anderson, T., Owicki, S., Saxe, J., and Thacker, C. High Speed Switch Scheduling for Local Area Networks. In *ACM Transactions on Computer Systems*, pp. 319–352, 1993.

[Ashok & Zahorjan 1992] Ashok, I. and Zahorjan, J. Scheduling a Mixed Interactive and Batch Workload on a Parallel, Shared-Memory Supercomputer. In *Proceedings of Supercomputing '92*, pp. 616–625, November 1992.

[Biagioni et al. 1993] Biagioni, E., Cooper, E., and Sansom, R. Designing a Practical ATM LAN. *IEEE Network*, 7(2):32–39, March 1993.

[Blelloch et al. 1991] Blelloch, G., Leiserson, C., and Maggs, B. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Symposium on Parallel Algorithms and Architectures*, July 1991.

[Blumrich et al. 1994] Blumrich, M., Li, K., Alpert, R., Dubnicki, C., Felten, E., and Sandberg, J. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

[Boden et al. 1995] Boden, N. J., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C. L., Seizovic, J. N., and Su, W.-K. Myrinet—A Gigabet-per-Second Local-Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[Carriero & Gelernter 1989] Carriero, N. and Gelernter, D. Linda in Context. *Communications of the ACM*, April 1989.

[Chandra et al. 1994] Chandra, R., Devine, S., Verghese, B., Gupta, A., and Rosenblum, M. Scheduling and Page Migration for Multiprocessor Computer Servers. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 12–24, October 1994.

[Chiang et al. 1994] Chiang, S.-H., Mansharamani, R. K., and Vernon, M. K. Use of Application Characteristics and Limited Preemption for Run-To-Completion Parallel Processor Scheduling Policies. In *Proceedings of the 1994 ACM SIGMETRICS Conference*, pp. 33–44, February 1994.

[Cox et al. 1994] Cox, A. L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., and Zwaenepoel, W. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 106–117, April 1994.

[Crovella et al. 1991] Crovella, M., Das, P., Dubnicki, C., LeBlanc, T., and Markatos, E. Multiprogramming on multiprocessors. Technical Report 385, University of Rochester, Computer Science Department, February 1991. Revised May.

[Culler et al. 1993a] Culler, D., Dusseau, A., Goldstein, S., Krishnamurthy, A., Lumetta, S., von Eicken, T., and Yelick, K. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, 1993.

[Culler et al. 1993b] Culler, D. E., Karp, R. M., Patterson, D. A., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[Culler et al. 1994] Culler, D., Dusseau, A., Martin, R., and Schauser, K. *Portability and Performance for Parallel Processing*, chapter 4: Fast Parallel Sorting under LogP: from Theory to Practice, pp. 71–98. John Wiley & Sons Ltd., 1994.

[Dahlin et al. 1994] Dahlin, M., Wang, R., Anderson, T., and Patterson, D. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the First Conference on Operating Systems: Design and Implementation*, November 1994.

[Delisle 1994] Delisle, P. Personal Communication, October 1994.

[Douglas et al. 1993] Douglas, C. C., Mattson, T. G., and Schultz, M. H. Parallel Programming Systems for Workstation Clusters. Technical Report 975, Yale University, Computer Science Department, August 1993.

[Douglis & Ousterhout 1991] Douglis, F. and Ousterhout, J. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–85, August 1991.

[Eager et al. 1986] Eager, D. L., Lazowska, E. D., and Zahorjan, J. Adaptive Load Shating in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, May 1986.

[Feitelson & Rudolph 1992] Feitelson, D. G. and Rudolph, L. Gang Scheduling Performance Benefits for Fine-Grained Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–18, December 1992.

[Felten & Zahorjan 1991] Felten, E. W. and Zahorjan, J. Issues in the Implementation of a Remote Paging System. Technical Report 91-03-09, University of Washington, Department of Computer Science, March 1991.

[Gelernter 1985] Gelernter, D. Parallel Programming in Linda. In *Proceeding of the International Conference on Parallel Processing*, pp. 255–263, August 1985.

[Gupta et al. 1991] Gupta, A., Tucker, A., and Urushibara, S. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *Proceedings of the ACM SIGMETRICS Conference*, pp. 120–32, May 1991.

[Krishnamurthy et al. 1994] Krishnamurthy, A., Lumetta, S., Culler, D., and Katherine, K. Y. Connected Components on Distributed Memory Machines. In *The Third DIMACS International Algorithm Implementation Challenge*, 1994.

[Kronenberg et al. 1986] Kronenberg, N. P., Levy, H. M., and Strecker, W. D. VAXclusters: A Closely-Coupled Distributed System. *ACM Transactions on Computer Systems*, 4(2), 1986.

[Lamport 1990] Lamport, L. Concurrent Reading and Writing of Clocks. *ACM Transactions on Computer Systems*, 8(4):305–310, April 1990.

[Lazowska 1994] Lazowska, E. Yet Another Workstation Network. Talk presented at the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, October 1994.

[Leighton 1985] Leighton, T. Tight Bounds on the Complexity of Parallel Sorting. *IEEE Transactions on Computers*, April 1985.

[Leiserson 1992] Leiserson, C. E. et al. The Network Architecture of the Connection Machine CM-5. In *Symposium on Parallel Algorithms and Architectures*, April 1992.

[Leutenegger & Sun 1993] Leutenegger, S. T. and Sun, X.-H. Distributed Computing Feasibility in a Non-Dedicated Homogenous Distributed System. In *Supercomputing 93*, 1993.

[Leutenegger & Vernon 1990] Leutenegger, S. T. and Vernon, M. K. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of the ACM SIGMETRICS Conference*, pp. 226–36, May 1990.

[Litzkow & Solomon 1992] Litzkow, M. and Solomon, M. Supporting Checkpointing and Process Migration Outside the UNIX Kernel. In *Winter 1992 USENIX Conference*, pp. 283–290, January 1992.

[Martin 1994] Martin, R. P. HPAM: An Active Message Layer for a Network of Workstations. In *Proceedings of the 2nd Hot Interconnects Conference*, July 1994.

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

[Mutka & Livny 1991] Mutka, M. M. and Livny, M. The Available Capacity of a Privately Owned Workstation Environment. *Performance Evaluation*, 12(4):269–84, July 1991.

[Naik et al. 1993] Naik, V. K., Setia, S. K., and Squillante, M. S. Performance Analysis of Job Scheduling Policies in Parallel Supercomputing Environments. In *Proceedings of Supercomputing '93*, pp. 824–833, November 1993.

[Nichols 1987] Nichols, D. Using Idle Workstations in a Shared Computing Environment. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pp. 5–12, November 1987.

[Ousterhout 1982] Ousterhout, J. K. Scheduling Techniques for Concurrent Systems. In *Third International Conference on Distributed Computing Systems*, pp. 22–30, May 1982.

[Peris et al. 1994] Peris, V. G., Squillante, M. S., and Naik, V. K. Analysis of the Impact of Memory in Distributed Parallel Processing Systems. In *Proceedings of the 1994 ACM SIGMETRICS Conference*, pp. 5–18, February 1994.

[Reinhardt et al. 1994] Reinhardt, S. K., Larus, J. R., and Wood, D. A. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 325–336, April 1994.

[Sevcik 1989] Sevcik, K. C. Characterizations of Parallelism in Applications and their Use in Scheduling. In *Proceedings of the 1989 ACM SIGMETRICS and PERFORMANCE Conference on Measurement and Modeling of Computer Systems*, pp. 171–180, May 1989.

[Sunderam 1990] Sunderam, V. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[Theimer & Lantz 1989] Theimer, M. M. and Lantz, K. A. Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Transactions on Software Engineering*, 15(11):1444–57, November 1989.

[Theimer et al. 1985] Theimer, M., Landtz, K., and Cheriton, D. Preemptable Remote Execution Facilities for the V System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pp. 2–12, December 1985.

[Tucker & Gupta 1989] Tucker, A. and Gupta, A. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. *Operating Systems Review*, 23(5):159–66, 1989.

[von Eicken et al. 1992] von Eicken, T., Culler, D. E., Goldstein, S. C., and Schauser, K. E. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[von Eicken et al. 1994] von Eicken, T., Avula, V., Basu, A., and Buch, V. Low-Latency Communication over ATM Netowrks using Active Messages. In *Proceedings of Hot Interconnects II*, Stanford, CA, August 1994.

[Zhou et al. 1992] Zhou, S., Wang, J., Zheng, X., and Delisle, P. Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computing Systems. Technical Report CSRI-257, University of Toronto, 1992.