The Utility of Exploiting Idle Workstations for Parallel Computation*

Anurag Acharya, Guy Edjlali, Joel Saltz
UMIACS and Department of Computer Science
University of Maryland, College Park 20742
{acha,edjlali,saltz}@cs.umd.edu

Abstract

In this paper, we examine the utility of exploiting idle workstations for parallel computation. We attempt to answer the following questions. First, given a workstation pool, for what fraction of time can we expect to find a cluster of k workstations available? This provides an estimate of the opportunity for parallel computation. Second, how stable is a cluster of free machines and how does the stability vary with the size of the cluster? This indicates how frequently a parallel computation might have to stop for adapting to changes in processor availability. Third, what is the distribution of workstation idle-times? This information is useful for selecting workstations to place computation on. Fourth, how much benefit can a user expect? To state this in concrete terms, if I have a pool of size S, how big a parallel machine should I expect to get for free by harvesting idle machines. Finally, how much benefit can be achieved on a real machine and how hard does a parallel programmer have to work to make this happen? To answer the workstation-availability questions, we have analyzed 14-day traces from three workstation pools. To determine the equivalent parallel machine, we have simulated the execution of a group of well-known parallel programs on these workstation pools. To gain an understanding of the practical problems, we have developed the system support required for adaptive parallel programs as well as an adaptive parallel CFD application.

1 Introduction

Exploiting idle workstations has been a popular research area. This popularity has been fueled partly by studies which have indicated that a large fraction of workstations are unused for a large fraction of time [9, 17, 19, 25] and partly by the rapid growth in the power of workstations. Batch-processing systems that utilize idle workstations for running sequential jobs have been in production use for many years. A well-known example is Condor [15], which has been has been in operation at the University of Wisconsin for about 8 years and which currently manages about 300 workstations [6].

The utility of harvesting idle workstations for parallel computation is less clear. First, the workstation-availability results [9, 17, 19, 25] that have held out the promise of free cycles assume, at least implicitly, that progress of execution on one workstation, or the lack thereof, has no effect on the progress of execution on other workstations. This assumption does not hold for most parallel computation. This is particularly so for data-parallel programs written in an SPMD style (most data-parallel programs are written in an SPMD style). When a workstation running a parallel job is reclaimed by its primary user, the remaining processes of the same job have to stop to allow the computation to be reconfigured. Reconfiguration may need one or more of data repartitioning, data/process migration and updating data location information. To make progress, a parallel job requires that a group of processors be continuously available for a sufficiently long period of time. If the state of a large number of processors rapidly oscillates between *available* and *busy*, a parallel computation will be able to make little progress even if each processor is available for a large fraction

*This research was supported by ARPA under contract #F19628-94-C-0057, Syracuse subcontract #353-1427

of time. Second, parallel programs are often not perfectly parallel. That is, they are able to run only on certain configurations - for example, configurations with powers-of-two processors. Addition or deletion of a single workstation may have no effect, a small effect or a very significant effect on the performance depending on the application requirements and the number of available machines.

In this paper, we examine the utility of exploiting idle workstations for parallel computation. We attempt to answer the following questions. First, given a workstation pool, for what fraction of time can we expect to find a cluster of k workstations available? This provides an estimate of the opportunity for parallel computation. Second, how stable is a cluster of free machines and how does the stability vary with the size of the cluster? This indicates how frequently a parallel computation might have to stop for adapting to changes in processor availability. Third, what is the distribution of workstation idle-times? That is, what is the probability that a workstation that is currently idle will be idle for longer than time t? This information is useful for selecting workstations to place computation on. Fourth, how much benefit can a user expect? To state this in concrete terms, if I have a pool of size S, how big a parallel machine should I expect to get for free by harvesting idle machines. Finally, how much benefit can be achieved on a real machine and how hard does a parallel programmer have to work to make this happen?

We have addressed these questions in three different ways. To answer the workstation-availability questions, we have analyzed 14-day traces from three workstation pools of different sizes (40, 60 and 300 workstations) and at different locations (College Park, Berkeley and Madison). To determine the equivalent parallel machine, we have simulated the execution of a group of well-known parallel programs on these workstation pools. To gain an understanding of the practical problems that arise when trying to run parallel programs in an adaptive fashion, we have developed system support that allows programs to detect changes in their environment and to adapt to these changes. We have also developed an adaptive version of a computational fluid dynamics program and have measured its actual performance using an IBM SP-2 as a cluster of workstations and one of the workstation availability traces mentioned above as the sequential workload.

Previous research into using idle workstations for parallel computation has taken one of three approaches. Leutenegger and Sun [14] use an analytic-model-based approach to study the feasibility of running parallel applications on non-dedicated workstation pool. Their study is based on simple synthetic models of both workstation availability and parallel program behavior. It is difficult to draw conclusions about behavior of real parallel programs on real workstation pools from their work. Carreiro et al [4] and Pruyne&Livny [21] propose schemes based on a master-slave approach. If the workstation on which a task is being executed is reclaimed, the task is killed and is reassigned by the master to a different workstation. There are two problems with this approach. First, most parallel programs are not written in a master-slave style. Second, rewriting existing parallel programs as master-slave programs would greatly increase the total communication volume and would require very large amounts of memory on the master processor. Arpaci et al [2] study the suitability of dedicated and non-dedicated workstation pools for executing parallel programs. They take a trace-based-analysis approach and base their study on a workstation availability trace, a job arrival trace for a 32-node CM-5 partition and a suite of five data-parallel programs. Their results show that a 60-workstation pool is able to process the workload submitted to a 32-node CM-5 partition. Our approach is closest to that of Arpaci et al but there are several basic differences. Arpaci et al focus on the interactive performance of parallel jobs and assume a time-sliced scheduling policy. They deduce the need for interactive response from the presence of a large number of short-lived parallel jobs in the CM-5 job trace. Most large parallel machines are, however, run in a batch mode. Usually, a small number of processors are provided for interactive runs. To better understand the need for interactive performance for parallel jobs, we analyzed long-term (six months to a year) job execution traces from three supercomputer centers (Cornell, Maui and San Diego). We found that over 90% of short-lived jobs used 16 processors or less (for details, see section 3.2). We take the position that the need for interactive response can be met by a small dedicated cluster and that throughput should be the primary goal of schemes that utilize non-dedicated workstations. In doing so, we follow the lead of Miron Livny and the Condor group at the University of Wisconsin which has had excellent success in utilizing idle workstations for sequential jobs.

We first examine the workstation-availability questions. We describe the traces and the metrics computed to estimate the opportunity for parallel computation. Next, we describe our simulation experiments and their results. We then describe our experience with the implementation and execution of an adaptive parallel program. Finally, we present a summary of our conclusions.

2 Workstation availability

To determine the availability of free workstation clusters for parallel computation, we have analyzed three two-week traces from three workstation pools. For each of these traces, we have computed two metrics. First, for what fraction of time can we expect to find a cluster of k free workstations. We refer to this as the *availability* metric. Second, for how long, on the average, is a cluster of k workstations stable? That is, how long can a parallel computation running on k processors expect to run undisturbed? We refer to this as the *stability* metric. In addition, we have computed two other measures for each trace. First, for what fraction of time is a workstation available on the average and second, how does the number of available workstations vary with time? These measures are for comparison with previous studies. Finally, we have computed the probability distribution for idle-times for all the workstations in this study. We first describe the three traces. We then present the parallel-availability metrics and the other measures for all three traces.

2.1 Traces

The first trace is from the workstation cluster of the CAD group at the UC Berkeley and contains data for about 60 workstations. This trace covers a 46-day period between 02/15/94 and 03/31/96. The trace we received had the busy and availability periods marked in for each workstation. This trace was used by Arpaci et al in [2]. We extracted the 14-day segment which had the largest number of traced workstations. We refer to this trace as the `ucb` trace.

The second trace is from the Condor workstation pool at the University of Wisconsin and contains data for about 300 workstations. This trace covers a 14-day period between 09/24/96 and 10/07/96. For the purpose of this trace, a workstation was considered to be available whenever the Condor status monitor marked it available. Condor uses several criteria, including user preferences, to decide if a workstation is available for batch jobs. We collected this trace by sampling the Condor status information once every three minutes using the web interface provided by the Condor project [5]. We refer to this as the `wisc` trace.

The third trace is from the public workstation cluster of the Department of Computer Science, University of Maryland. This trace contains data for about 40 workstations and covers a 14-day period from 09/24/96 to 10/07/96. For the purpose of this trace, a workstation was considered to be available if the load average was below 0.3 for more than five minutes. We refer to this as the `umd` trace.

The number of workstations participating in the pools was not constant throughout the tracing periods. The average number of participating workstations was 52 for the `ucb` trace, 277 for the `wisc` trace and 39 for the `umd` trace. We use these figures as a measure of the size of the corresponding pools.

In addition to the variations in size, time period and location, these pools also vary in the way they are (were) used. The College Park pool consists of publicly available machines which are primarily used by junior computer science graduate students for course assignments as well as for personal purposes (mail etc). The Berkeley pool consists of workstations belonging to a single research group and is used for both personal purposes and compute-intensive research. The Madison pool includes both compute servers and personal workstations. It spans several departments. We expect that together these pools are representative of most workstation clusters available in university environments.

2.2 Parallel-availability metrics

Figure 1 presents the *availability* metric for all three pools. Each graph shows how the metric varies with cluster size. For each pool, the fraction of time for which a cluster of k workstations is available drops more or less linearly with k. Note, however, that for each pool, a substantial fraction (20-70%) of the pool is *always* available. Except for the `umd` trace, the drop is relatively slow – clusters larger than half the total size of the pool are available for over half the time.

Figure 2 presents the *stability* metric for all three pools. Each graph shows how the metric varies with cluster size. These graphs show that clusters up to half the size of the pool are stable for four to fifteen minutes and clusters up to a third of the pool are stable for five to thirty minutes. This holds out promise for parallel applications. Even if the cost of reacting to a reclamation event is as high as one minute, it is possible to make significant progress. An important point to note is that even though Figure 1 shows that large workstation clusters are available at any given time, these clusters are not stable. For example, a cluster

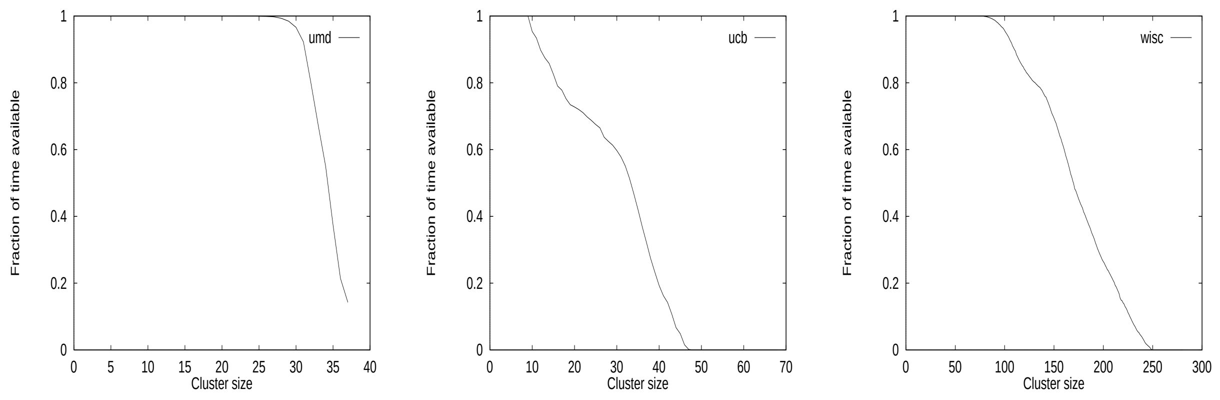


Figure 1: Availability for the three pools. These graphs show for what fraction of time can we expect to find a cluster of k free workstations and how this fraction varies with the cluster size k. For comparison, the average number of participating workstations was 52 for `ucb`, 277 for `wisc` and 39 for `umd`.

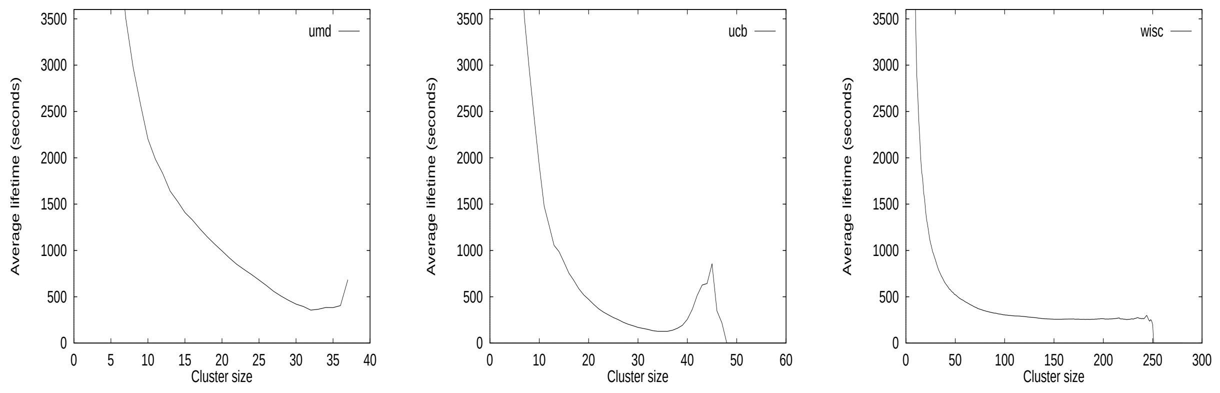


Figure 2: Stability for the three pools. These graphs plot the average period a cluster is stable for against the cluster size.

of 88 workstations can always be found in the `wisc` pool as per Figure 1 but a cluster of 88 workstations is stable only for five and a half minutes (see Figure 2). The upturns at the right end of the graphs for the `ucb` and `umd` traces correspond to a small number of idle periods on weekend nights.

Figure 3 shows how the fraction of workstations that are idle varies with time for the three pools. Weekdays are indicated by the dips; nights by the humps and weekends by the shorter-than-usual dips. In each graph, the horizontal line labeled `avg` shows the average fraction of the pool that is available. These results indicate that, on the average, 60% to 80% of the workstations in a pool are available. This agrees with previous results [2, 9, 17].

2.3 Distribution of workstation idle-time

In this section, we try to answer the question – what is the probability that an idle workstation will be idle for longer than time t? This question has been previously looked at by several researchers [2, 9]. The common experience has been that machines that have been idle for a short time are more likely to be reclaimed than machines that have been idle for a relative long period. Douglis&Ousterhout [9] mention that for their cluster, machines that were idle for 30 seconds were likely to be idle for an average of 26 minutes; Arpaci et al [2] mention that, in their study, a recruitment threshold of 3 minutes provided the best throughput. Given the relative plenty in terms of workstation availability, we did not focus on the issue of recruitment. Instead, we looked at distribution of relatively long idle periods (tens of minutes to several hours). Our goal was to help select between multiple available workstations for the placement of computation.

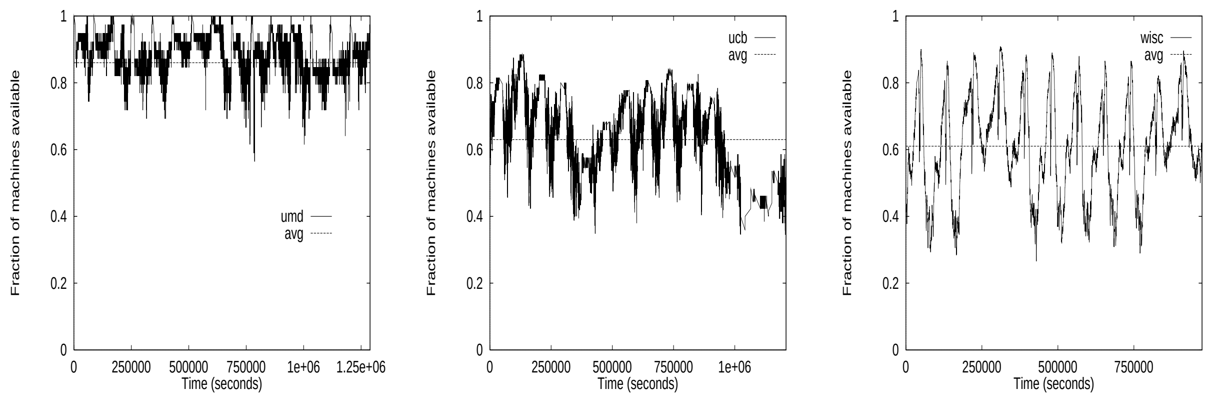


Figure 3: Fraction of workstations available for the three pools.

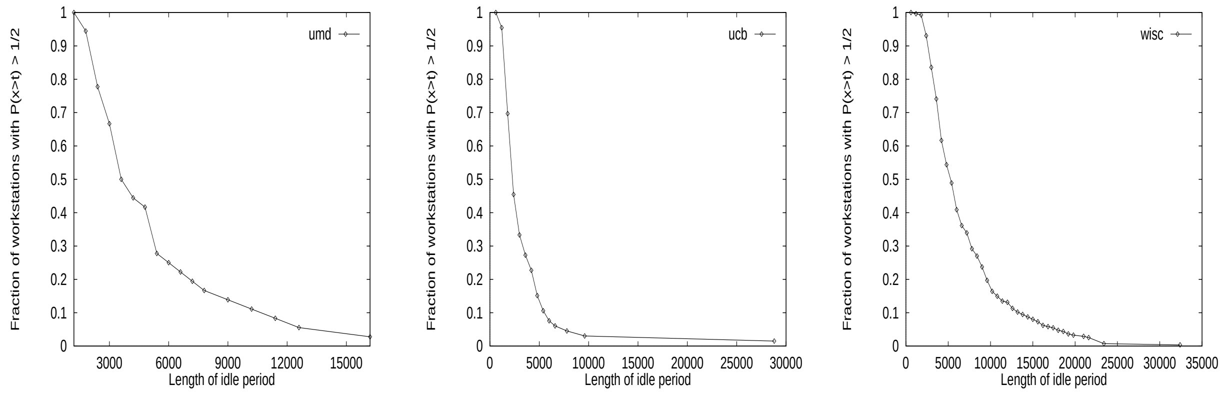


Figure 4: Cumulative distribution of the idleness-cutoff for the three pools.

For each workstation occurring in the workstation-availability traces, we computed the probability $P(x > t)$ that an idle period would last longer than time t. We considered only the idle periods that were at least five minutes long. We found that the probability distribution varied widely. To summarize the information, we characterized each workstation by the time T such that $P(x > T) = 0.5$. We refer to this measure as the *idleness-cutoff*. That is, idle periods shorter than T had a probability greater than half; idle periods longer than T had a probability less than half. The minimum value of the idleness-cutoff was 18 minutes and the maximum value was 9 hours. Figure 4 shows the cumulative distribution of the *idleness-cutoff*. The average value of the *idleness-cutoff* was 40 minutes for the `ucb` trace, 70 minutes for the `umd` trace and 90 minutes for the `wisc` trace. Given the large value of the idleness-cutoff, simple strategies (such as LIFO, FIFO, random etc) for selecting between available workstations should suffice. We note that all of these values are significantly higher than the 26 minutes reported by Douglis [8] in 1990 for the Sprite workstations.

3 How much benefit can a user expect?

To estimate the benefit that parallel programs might achieve in shared workstation environments, we simulated the execution of a group of well-known parallel programs on all three pools. We selected a suite of eight programs which includes the NAS parallel benchmarks [22] and three programs that have been studied by one or more research groups working on parallel processing. We simulated two scenarios: (1) repeated execution of individual applications without gaps; (2) repeated execution of the entire set of applications, also without gaps. Since these scenarios keep the pool busy at all times, they provide an approximate upper bound on the throughput. The equivalent parallel machine is used as the metric.

We first describe the programs we used as benchmarks. We then describe our simulations and the information used to drive them. Finally, we present the results.

3.1 Benchmarks

All programs in this suite are programmed in the SPMD model. Figure 5 shows the speedups for the benchmarks running on dedicated parallel machines. These numbers have been obtained from publications [1, 3, 22, 23, 26]. The programs themselves are described below. We used class B datasets for all the NAS benchmarks.

- `nas-bt`: this program uses an approach based on block-tridiagonal matrices to solve the Navier-Stokes equations [22]. The running time on one processor of the IBM SP-2 is 10942 seconds and the total memory requirement is 1.3 GB. This program runs on configurations with square number of processors.
- `nas-sp`: this program uses a pentadiagonal matrix-based algorithm for the Navier-Stokes equations [22]. The running time on one processor of the IBM SP-2 is 7955 seconds and the total memory requirement is 325.8 MB. This program runs on configurations with square number of processors.
- `nas-lu`: this program uses a block-lower-triangular block-upper-triangular approximate factorization to solve the Navier-Stokes equations. The running time on one processor of the IBM SP-2 is 8312 seconds and the total memory requirement is 174.8 MB. This program runs on configurations with powers-of-two processors.
- `nas-mg`: this implements a multigrid algorithm to solve the scalar discrete Poisson equation [22]. The running time on one processor of the IBM SP-2 is 228 seconds and the total memory requirement is 461 MB. This program runs on configurations with powers-of-two processors.
- `nas-fftpde`: this program solves a Poisson partial differential equation using the 3-D FFT algorithm [1]. The running time on sixteen processors of the IBM SP-1 is 286 seconds and the total memory requirement is 1.75 GB. This program runs on configurations with powers-of-two processors.
- `dsmc3d`: is a Monte-Carlo simulation used to study the flow of gas molecules in three dimensions [23]. The running time on one processor of the iPSC/860 is 4876 seconds and the total memory requirement is 30 MB.
- `unstructured`: this is a flow solver capable of solving the Navier-Stokes equations about complex geometries through the use of unstructured grids [3]. Its running time on one processor of the Intel Paragon is 68814 seconds and the total memory required is 134 MB. We have data for this program running on 1,2,3,4,5,10,25 and 50 processors.
- `hydro3d`: this is a parallel implementation of $3 + 1$-dimensional relativistic hydrodynamics [26]. Its running time on one processor of the Intel Delta is 406000 seconds and the total memory required is 89.2 MB. We have data for this program running on 1,8,16,32 and 64 processors.

3.2 Simulations

To compute the equivalent parallel machine for the scenarios mentioned above, we performed a high-level simulation of the execution of SPMD parallel programs on non-dedicated workstation pools. The simulator takes as input workstation availability traces and a description of the parallel programs to be simulated and computes the total execution time. The equivalent-parallel-machine measure is computed by determining the size of the parallel machine that would be required to complete the execution in the same time. All the programs used in this study can run only on a fixed set of configurations (e.g. powers-of-two). If the execution time falls in between two configuration, linear interpolation is used to compute the equivalent parallel machine.

All the programs used in this study are iterative. For the purpose of this study, we characterize the speed of execution by the time taken to execute each iteration. We obtained the time per iteration from the publications cited above. We characterize the size of each process by the size of its partition of the program

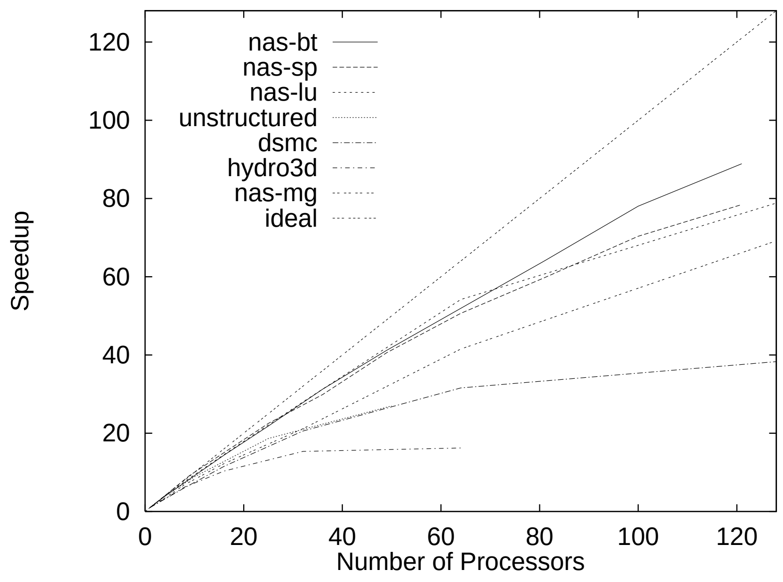


Figure 5: Speedups of the benchmark programs on dedicated parallel machines.

data. We obtained the size of the program data by code inspection for all benchmarks except the last two. For the last two programs, we obtained the size information from publications.

Many of these benchmarks have very large data sets which cannot reasonably fit in the memory of a single workstation. We assumed workstations with no more than 128 MB and did not perform experiments that required more than this amount on any of the machines.

There are many ways in which an SPMD program can adapt to a change in the number of available processors. For example, it could checkpoint the evicted process to disk and restart it elsewhere (as in Condor [24]) or it could stop the process and copy it over from memory to memory (as in Locus [20]), or it could copy the stack and a small number of pages over and fault the rest in lazily (as in Accent [28] and Sprite [8]). All of these schemes involve moving an executing process. Since SPMD applications run multiple copies of the same program which are usually in (loose) synchrony, there is another, possibly cheaper, alternative. Just the program data for the process can be moved; scratch data and text need not be moved. If sufficient workstations are not available, data is moved to processes that are already running; otherwise, the program has to pay the cost of starting up a new process at the new location (this cost is not specific to this scheme - expanding the number of processors requires new processes). There are two points to note. First, adaptation can happen only when the data is in a "clean" state and in a part of the code that every processor will reach. That usually means outside parallel loops. Second, the process startup cost also includes the cost of recomputing communication schedules. In our study, we have assumed that this adaptation technique is used.

The simulator assumes a point-to-point, 15 MB/s-per-link interconnect. It models the eviction cost in two parts: a fixed eviction cost that consists of the process startup cost and a variable part that includes the memory copy cost at both ends, the time on the wire and end-point congestion for the data motion required for eviction. The process startup cost is paid at least once – to account for the initialization time. Thereafter it is paid every time an application adapts to a change in processor availability. We used 64 ms/MB as the memory copy cost which we obtained empirically from a DEC Alpha Server 4/2100 running Digital Unix 3.2D. The simulator also models a *settling* period between the completion of one program and the start of another. We used a settling period of two seconds.

Since idle workstations are relatively plentiful, our goal was to use as simple a scheduling strategy as possible. In their study, Arpaci et al [2] focus on the interactive performance of parallel jobs and assume a time-sliced scheduling policy. They deduce the need for interactive response from the presence of a large number of short-lived parallel jobs in the CM-5 job arrival trace. Most parallel machines, however, run in a batch mode. To better understand the need for interactive response from parallel jobs, we analyzed long-

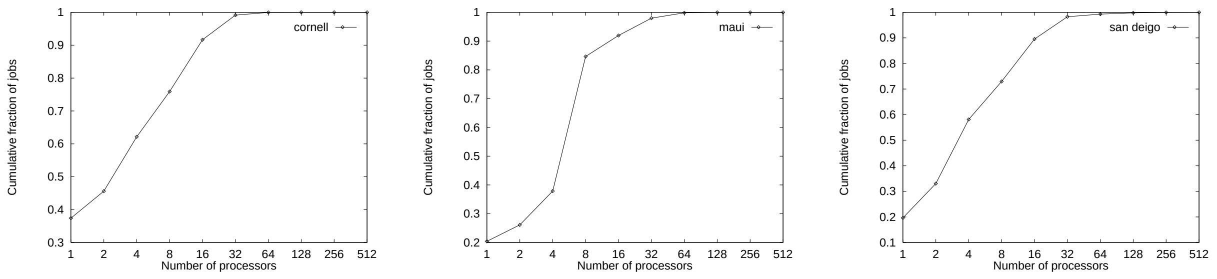


Figure 6: Processor usage distribution for short-lived jobs. The Cornell results are based on jobs executed between Jun 18 and Dec 2 1995; the Maui results are based on jobs executed between Jan 1 and Aug 31, 1996; and the San Diego results are based on jobs executed between Jan 1 and Dec 31, 1995. The total number of short-lived jobs are 53015 (San Diego), 13651 (Maui) and 14822 (Cornell). The average number of short-lived jobs per day is 145, 56 and 88 respectively.

term (six months to a year) job execution traces from three supercomputer centers (Cornell, Maui and San Diego). Figure 6 shows the processor usage distribution of short-lived jobs (jobs that run for two minutes or less) for the three traces. In all three cases, over 90% of the short jobs run on sixteen processors or less. Based on this and on our own experience with parallel machines, we speculate that interactive performance is usually desired for debugging and testing purposes; most production runs are batch jobs. We take the position that the need for interactive response can be met by a small dedicated cluster and that throughput should be the primary goal of schemes that utilize non-dedicated workstations. In doing so, we follow the lead of Miron Livny and the Condor group at the University of Wisconsin which has had excellent success in utilizing idle workstations for sequential jobs. In our study we assume a simple first-come-first-served batch scheduling policy.

We ran our experiments for one week of simulated time. This allowed us to study long-term throughput and to understand the effect of time-of-day/day-of-week variations in workstation usage.

3.3 Results

Table 1 presents the equivalent parallel machine implied by the performance of the different applications for week-long runs. We have computed two aggregate measures: the average equivalent machine and the median equivalent machine. The median measure was computed to avoid possible bias due to outliers. From these results, we conclude that harvesting idle workstations from these pools can provide the equivalent of 29 (College Park), 25 (Berkeley) and 92/109 (Madison) dedicated processors. The measures for the Berkeley pool match the 1:2 rule of thumb that Arpaci et al [2] suggest for the parallel machine equivalent to a non-dedicated workstation pool. However, the rule does not match the results for the other two clusters. We rule out the difference in the scheduling strategies as the primary cause of the difference as using a large quantum would eliminate most of the effects of time-slicing. Instead, we believe that the difference is due to (1) the *limited configuration* effect and (2) difference in the load characteristics. The *limited configuration* effect refers to the fact that parallel programs can run only on certain configurations. Addition or deletion of a single workstation may have no effect, a small effect or a very significant effect on the performance depending on the application requirements and the number of available machines. This effect is particularly important when the number of available workstations hovers around “magic numbers” like powers-of-two and squares.

Figure 7 shows the temporal variation in the performance over the period of the experiment. Since the benchmark programs run for widely varying periods, it is not possible to compute an aggregate number. We have selected `nas-bt` as the exemplar program. Beside the obvious diurnal variations, the graphs show the impact of the *limited configuration* effect. There are sharp changes in performance as the workstation availability crosses certain thresholds. Note that of all our benchmarks, `nas-bt` is the one that can run on the maximum number of configurations (it runs on square number of processors). Another point to note is the difference in the nature of the graphs for `umd` and `ucb` on one hand and the graph for `wisc` on the other hand. The graphs for `umd` and `ucb` are jagged whereas the graph for `wisc` consists mostly of a thick

	College Park	Berkeley	Madison
Average proc on	39	52	277
Average proc avail	34	32	169
Applications			
`dsmc3d`	31	27	109
`hydro3d`	32	27	61
`nas-bt`	29	28	113
`nas-fftpde`	30	22	114
`nas-lu`	31	25	115
`nas-mg`	26	23	68
`nas-sp`	30	29	117
`unstructured`	24	22	49
`roundrobin`	29	24	90
Average par mc	29 (0.74)	25 (0.48)	92 (0.33)
Median par mc	29 (0.74)	25 (0.48)	109 (0.39)

Table 1: Average per-application equivalent parallel machine over one week. The process startup time is assumed to be two seconds. The fraction in the parentheses is the ratio of the equivalent parallel machine and the size of the pool.

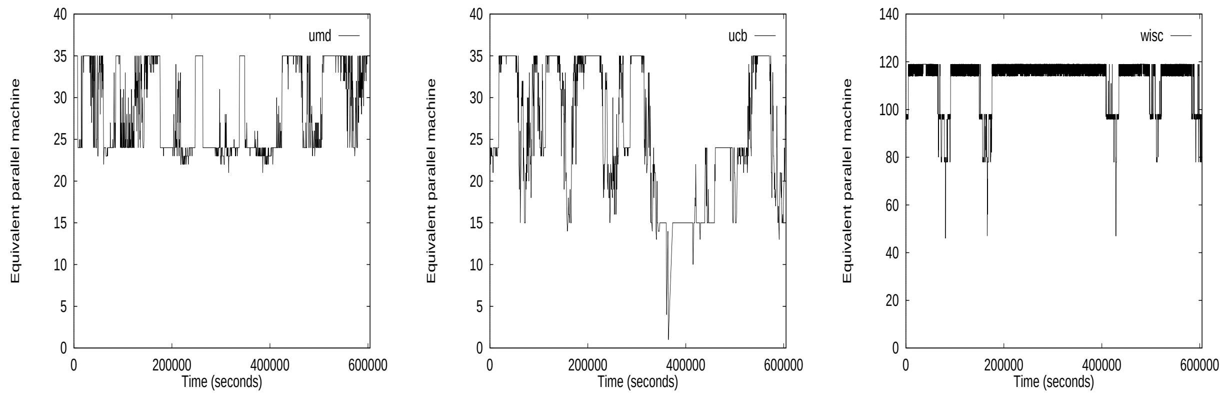
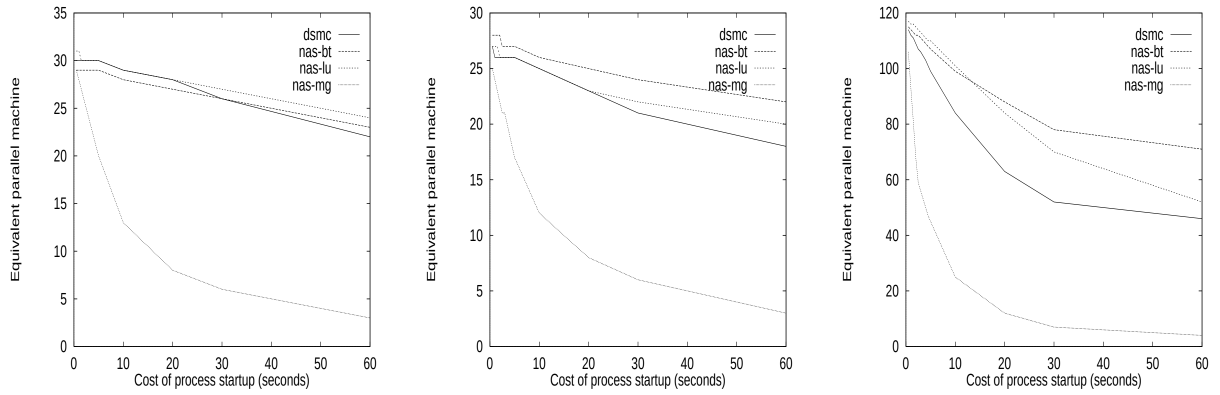


Figure 7: Variation of equivalent parallel machine over a week. `nas-bt` was used as the exemplar.

band. The jaggedness indicates that workstation availability often hovered around "magic numbers" and forced switches between different configurations. The thick band indicates that workstations were plentiful and that the program did not have to change configurations. Instead, when a workstation was taken away, a replacement was available. The deep dip in the middle of the graph for `ucb` corresponds to a period of intensive use (see the corresponding availability graph in Figure 3).

3.4 Impact of change in eviction cost

In the experiments described above, we assumed that the process startup time was two seconds. Recall that process startup time is fixed portion of the eviction cost. It includes the cost of initiating the adaptation, the cost of starting up a new process (if need be), and the cost of recomputing the communication schedules. This cost depends on the particular adaptation mechanism used. To determine the impact of eviction cost on the performance, we repeated out experiments for a wide range of process startup costs. Figure 8 shows how the equivalent parallel machine varies with process startup cost. In each graph, we plot the performance achieved for four applications - `dsmc3d`, `nas-bt`, `nas-lu` and `nas-mg`. The performance for the other four applications lies approximately between the curves for `dsmc3d`, `nas-bt` and `nas-lu`. We make two observations: (1) the



(a) College Park (b) Berkeley (c) Madison

Figure 8: Variation of the equivalent parallel machine with process startup cost.

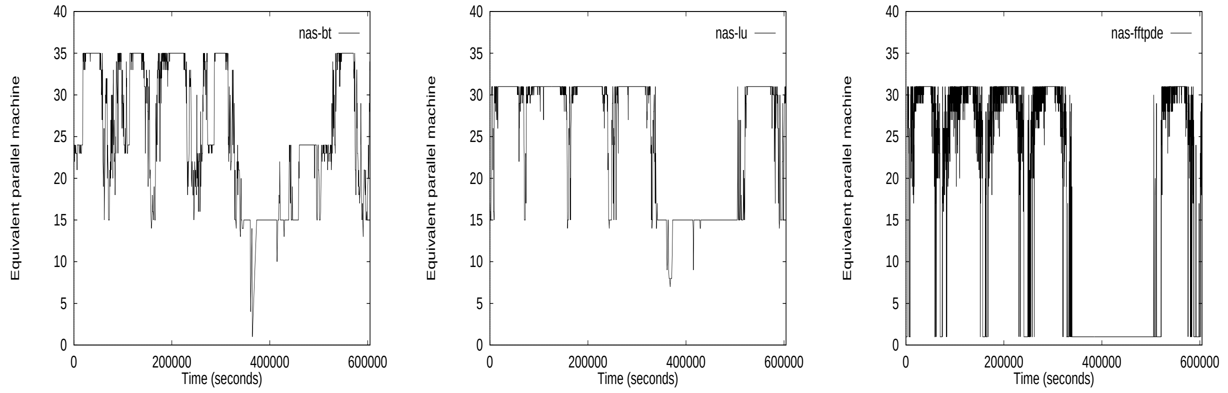


Figure 9: Impact of configuration flexibility.

performance for `nas-mg` drops sharply for all three pools; (2) the relative drop in the performance for the other applications is largest for `wisc`, followed by `ucb` and `umd`; the drops for `umd` being quite small.

The primary cause for the sharp drop in the performance of `nas-mg` is that it runs for a very short time. The total execution time is 228 seconds on a single processor and about 19 seconds on 16 processors. As a result, the performance for `nas-mg` is swamped by startup costs. The gradation in the performance difference across the pools can be attributed to differences in the frequency of reclamation events.

3.5 Impact of configuration flexibility

To examine the effect of configuration flexibility, we compared the performance of a single pool for three programs, `nas-bt`, `nas-lu` and `nas-fftpde` with different levels of configurability. We selected the Berkeley pool for this comparison as configuration flexibility is likely to have the maximum impact for situations with a relatively small number of processors and relatively frequent reclamations. The first of these programs, `nas-bt`, runs on square number of processors and the other two run on powers-of-two processors. However, the dataset of `nas-fftpde` is so large that it is cannot be run on configurations smaller than 16 processors. While the effect of configuration flexibility can be seen in several parts of the graph, it is most apparent in the central dip. The first two programs are able to salvage some computation during this time period, `nas-bt` being more successful towards the end since it can run on 25 processors. On the other hand, `nas-fftpde` makes virtually no progress in this period. We would like to point out that the period in question is of the order of two days.

Num processors	1	2	4	8	16
dataset 1	319 ms	196 ms	134 ms	106 ms	87 ms
dataset 2	510 ms	380 ms	209 ms	150 ms	130 ms

Table 2: Time per iteration for the two datasets.

4 Evaluation on a real machine

To gain an understanding of the practical problems that arise when trying to run parallel programs in an adaptive fashion, we have developed system support that allows programs to detect changes in their environment and adapt to these changes. We have also developed an adaptive version of a computational fluid dynamics program and have measured its actual performance using an IBM SP-2 as a cluster of workstations and one of the workstation availability traces mentioned above as the sequential workload.

Our system (called Finch) uses a central coordinator to keep track of the workstation availability and a per-application manager process which keeps track of the progress of the application. The central coordinator resembles the Condor central manager [24] and runs on a central machine. The application manager is created when the job is submitted and lives for the duration of the job. It runs on the submitting machine. Global resource allocation decisions are made by the central coordinator; coordination of application processes for the purpose of adaptation is done by the application manager. Currently, we assume a cooperative user environment and provide a pair of programs that the primary user of the workstation can use to make the workstation available and to reclaim it for personal use. User requests (reclamation or otherwise) are sent to the central coordinator which selects the application that must respond to the event. It then informs the corresponding application manager which coordinates the response. Finch is portable across Unix environments. Currently, it runs on Suns, Alphas and RS/6000s.

For this study, we used a template extracted from a multiblock computational fluid dynamics application that solves the thin-layer Navier-Stokes equations over a 3D surface (multiblock TLNS3D [27]). This is an iterative SPMD program, each iteration corresponds to a different timestep. We chose the top of the time-step loop as the safe point for eviction. If a reclamation request is received when the program is at any other point, eviction is delayed till all processes reach this point. As described later in this section, the additional delay introduced, at least for this program, is quite small. We used the Adaptive Multiblock PARTI library [10] from the University of Maryland for parallelizing the application. This library performs the data partitioning for normal execution as well as the repartitioning for adaptation. It also manages the normal data communication as well as the data motion needed for eviction. To achieve efficient communication, this library pre-computes communication schedules. Changing the number or the identity of its processors requires recomputation of the schedule. Adaptive Multiblock PARTI is not unique in providing these services. The DRMS system [16] from IBM Research provides similar functionality. The point we would like to make is that this support does not have to be implemented by a parallel programmer.

We needed to make four changes to the program to allow it to run in an adaptive fashion. First, we added a call to initialization code which includes contacting the central coordinator for resources. Second, we added code to the top of the time-step loop to check for adaptation events and a call to an adaptation routine if the check succeeds. Third, we wrote the adaptation routine which repartitions the data arrays and moves it to destination nodes. Finally, we added a call to a finalization routine which, among other things, informs the central coordinator about the completion of this program.

We evaluated the performance of Finch and this application using a 16-processor IBM SP-2 as the workstation pool and 16 workstation availability traces from the College Park pool as the sequential workload. We ran this program in powers-of-two configurations from one to sixteen processors. We used two input datasets for our experiments with different meshes. Table 2 shows the time per iteration for the different configurations.

We designed our experiments to allow us to compute three measures. First, the cost of the running the adaptive version when no adaptation is required. Second, the time for eviction. That is, the time a user has to wait for her workstation once she has made a reclamation request. We have divided this time into two

Num processors	1	2	4	8	16
dataset 1	0.1%	0.1%	0.1%	0.1%	0.5%
dataset 2	0.1%	0.1%	0.1%	0.1%	0.4%

Table 3: Slowdown relative to the non-adaptive version. The workstation pool was assumed to be unused for the period of this experiment.

Num of src proc	1	1	1	1	2	2	2	4	4	8
Num of dest proc	2	4	8	16	4	8	16	8	16	16
Remap time	125 ms	188 ms	214 ms	250 ms	62 ms	93 ms	115 ms	28 ms	48 ms	19 ms

Table 4: Application-level cost of adaptation (dataset 1).

parts. The first part consists of the time spent by the application (the time to repartition, move the data as well as compute the new communication schedules) and the second part consists of time spent by the central coordinator and the application manager. Finally, we computed the equivalent parallel machine.

Table 3 shows the slowdown of adaptive version of the code compared to the original non-adaptive version. For the period of this experiment, the workstation pool was assumed to be quiescent and no adaptation was required. We note that the overhead of using the adaptive version is negligible. This is understandable since the check for an adaptation event is no more than checking whether there is a pending message on a socket. The rest of the adaptation code is not used if there are no adaptations.

Table 4 presents the application-level cost of adapting between different configurations. The cost is roughly proportional to the magnitude of the change in the number of processors and the size of the data partition owned by each processor.

Figure 10 shows the equivalent parallel machine for one, two and four copies of the program running together. In these experiments, the first copy is allowed to start first and others follow in sequence. The first copy is assigned as many nodes as it wants at start time and the other copies compete for the remaining nodes and for the nodes that become available during the computation. As a result, the first copy achieves better performance than the others. The largest equivalent parallel machine is 11 processors for the first dataset and 13 processors for the second data set. That corresponds to 69% and 81% of the size of the pool. For comparison, the equivalent parallel machine for the entire set of `umd` traces was computed to be 74% (see section 3.3).

The average time the user had to wait for a guest process to leave depended on the number of processors and the size of data for the job the guest process was a part of. For a single program running by itself on the pool, the average wait time for eviction was 1.191 seconds. For multiple programs running together, the average wait time for eviction was 1.669 seconds. The number of adaptation events over the period of this experiment was 487.

5 Other Related work

In this paper, we considered the use of idle workstations as compute servers. With the current growth in the number and the size of data-intensive tasks, exploiting idle workstations for their memory could be an attractive option. Dahlin et al [7] study the feasibility of using idle memory to increase the effective file cache size. Feely et al [11] describe a low-level global memory management system that uses idle memory to back up not just file pages but all of virtual memory as well. They show that this scheme is able to use idle memory to improve the performance of a suite of sequential data-intensive tasks by a factor between 1.5 and 3.5. Franklin et al [12] describe a unified memory management scheme for the servers and all the clients in a client-server database system. Their goal was to avoid replication of pages between the buffer pools of all the clients as well as the buffer pools of the servers. Explicit memory servers have been proposed

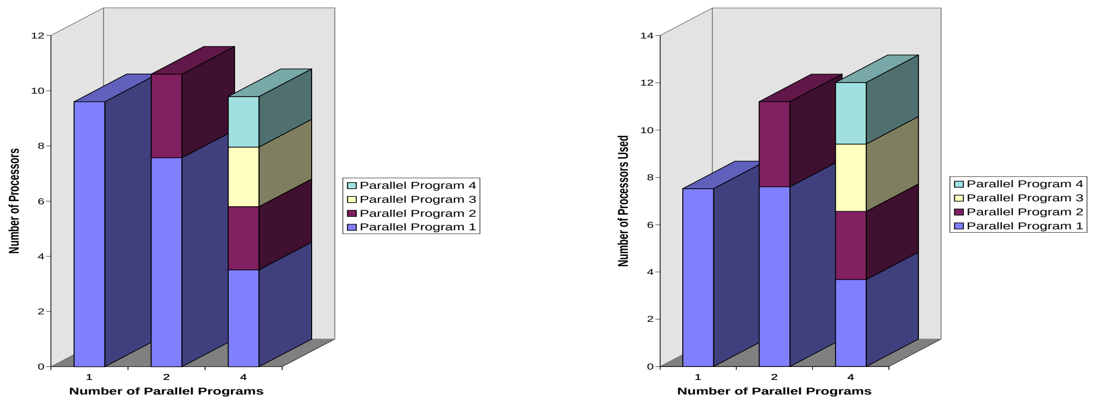


Figure 10: Equivalent parallel machine for one, two and four programs. The graph on the left is for the first dataset and the graph on the right is for the second dataset.

by Narten&Yavagkar [18] and Iftode et al [13]. Narten&Yavagkar describe a memory server similar in spirit to the Condor central manager. It keeps track of the idle memory available and ships memory objects to the corresponding machines as needed. Iftode et al propose extending the memory hierarchy of multicomputers by introducing a remote memory server layer.

Harvesting idle workstations for their memory imposes fewer requirements on the system support than harvesting them for their computation. If done properly, memory can be often be shared for long periods without significant impact on the interactive performance, particularly for today's machines which have large primary memories. Eviction of guest memory pages does not have the same urgency as the eviction of guest processes.

6 Summary of conclusions

There are two primary conclusions of our study. First, that there is significant utility in harvesting idle workstations for parallel computation. There is, however, considerable variance in the performance achieved. For the three non-dedicated pools we studied, we found that they could achieve performance equal to a dedicated parallel machine between one-third to three-fourths the size of the pool. Supporting evidence for this conclusion is provided by our experience with Finch and an adaptive Navier-Stokes template. Second, the parallel throughput achieved by a non-dedicated pool depends not only on the characteristics of sequential load but also on the flexibility of the parallel jobs being run on it. Jobs that can run only on a small number of configurations are less able to take advantage of the dynamic changes in availability; jobs that can run on a large set of configurations achieve better throughput. This effect is particularly important when the number of workstations available hovers around "magic numbers" like powers-of-two and squares.
The other conclusions of our study are:

- On the average, 60% to 80% of the workstations of a pool are available. The fraction of time for which a cluster of k workstations is available drops more or less linearly with k. Clusters larger than half the total size of the pool are available for over half the time. Moreover, a substantial fraction (20%-70%) of the workstations is always available.

- Even though large clusters are available at any given time, these clusters are not stable. Clusters up to half the size of the pool are stable for four to fifteen minutes and clusters up to a third of the pool are stable for five to thirty minutes.

- There is a wide variance in the distribution of the length of idle periods across different workstations. The expected length of an idle period varied from a minimum of 18 minutes to a maximum of 9 hours. On the average, workstation that has been idle for five minutes can be expected to be idle for another 40-90 minutes.
- It is not too difficult to convert SPMD programs to run in an adaptive environment. This conversion is benign. That is, the modifications do not have an adverse impact on the performance of the programs. Also, useful gains are possible on real machines.
- The eviction delay seen by a user is not unacceptably large. However, we would like to caution the reader that this conclusion is based on a scheme that does no checkpointing and as such is unable to recover from failures.

Acknowledgments

We would like to thank Remzi Arpaci for the UC Berkeley workstation availability traces. We would like to thank the Condor group at the University of Wisconsin for providing the web interface to the Condor status monitor. We would like to thank Steven Hotovy of the Cornell Theory Center for the trace of the jobs submitted to their IBM SP-2, Regan Moore and George Kremenek of the San Diego Supercomputing Center for the trace of the jobs submitted to their Intel Paragon, and Peter Young of the Maui High Performance Computing center for the trace of the jobs submitted to their IBM SP-2.

References

[1] R.C. Agarwal, F.G. Gustavson, and M. Zubair. An efficient algorithm for the 3-D FFT NAS parallel benchmark. In *Proceedings of SHPCC'94 (Scalable High-Performance Computing Conference)*, pages 129–33, May 1994.

[2] R.H. Arpaci, A.D. Dusseau, A.M. Vahdat, L.T. Liu, T.E. Anderson, and D.A. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 267–78, May 1995.

[3] D. Bannerjee, T. Tysinger, and W. Smith. A scalable high-performance environment for fluid-flow analysis on unstructured grids. In *Proceedings of Supercomputing'94*, pages 8–17, November 1994.

[4] N. Carriero, D. Gelernter, M. Jourdenais, and D. Kaminsky. Piranha scheduling: strategies and their implementation. *International Journal of Parallel Programming*, 23(1):5–33, Feb 1995.

[5] The Condor status monitor. *http://www.cs.wisc.edu/cgi-bin/condor_status/-server*, 1996.

[6] The Condor summary status monitor. *http://www.cs.wisc.edu/cgi-bin/condor_status/-server+-tot*, 1996.

[7] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: using remote memory to improve file system performance. In *Proceedings of the first Symposium on Operating System Design and Implementation*, pages 267–80, Nov 1994.

[8] F. Douglis. *Transparent Process Migration in the Sprite Operating System*. PhD thesis, Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Sep 1990.

[9] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software Practice and Experience*, 21(8):757–85, August 1991.

[10] G. Edjlali, G. Agrawal, A. Sussman, and J. Saltz. Data parallel programming in an adaptive environment. In *Proceedings of the ninth International Parallel Processing Symposium*, pages 827–32, April 1995.

[11] M. Feely, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the fifteenth ACM Symposium on Operating System Principles*, pages 201–12, Dec 1995.

[12] M. Franklin, M. Carey, and M. Livny. Global memory management in client-server DBMS architectures. In *Proceedings of the eighteenth International Conference on Very Large Data Bases*, pages 596–609, Aug 1992.

[13] L. Iftode, K. Li, and K. Petersen. Memory servers for multicomputers. In *COMPCON Spring'93 Digest of Papers*, pages 538–47, Feb 1993.

[14] S. Leutenegger and X.-H. Sun. Distributed computing feasibility in a non-dedicated homogeneous distributed system. In *Proceedings of Supercomputing'93*, pages 143–52, November 1993.

[15] M. Litzkow and M. Livny. Experiences with the Condor distributed batch system. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, pages 97–101, Oct 1990.

[16] J. Moreira, V. Naik, and R. Konuru. A programming environment for dynamic resource allocation and data distribution. Technical Report RC 20239, IBM Research, May 1996.

[17] Matt Mutka and Miron Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, 12(4):269–84, July 1991.

[18] T. Narten and R. Yavagkar. Remote memory as a resource in distributed systems. In *Proceedings of the third Workshop on Workstation Operating Systems*, pages 132–6, April 1992.

[19] David Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the Eleventh ACM Symposium on Operating Systems*, pages 5–12, November 1987.

[20] G. Popek and B. Walker. *The LOCUS Distributed System Architecture*. The MIT Press, 1985.

[21] J. Pruyne and M. Livny. Parallel processing on dynamic resources with CARMI. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 259–78, April 1995.

[22] W. Saphir, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.1 Results. Technical Report NAS-96-010, NASA Ames Research Center, August 1996.

[23] S. Sharma, R. Ponnuswami, B. Moon, Y-S Hwang, R. Das, and J. Saltz. Runtime and compile-time support for adaptive irregular problems. In *Proceedings of Supercomputing'94*, pages 97–108, November 1994.

[24] T. Tannenbaum and M. Litzkow. The Condor distributed processing system. *Dr. Dobbs' Journal*, 20(2):42–4, Feb 1995.

[25] Marvin Theimer and Keith Lantz. Finding idle machines in a workstation-based distributed system. *IEEE Transactions on Software Engineering*, 15(11):1444–57, November 1989.

[26] A.S. Umar, D.J. Dean, C. Bottcher, and M.R. Strayer. Spline methods for hydrodynamic equations: parallel implementation. In *Proceedings of the Sixth SIAM conference on parallel processing for scientific computing*, pages 26–30, March 1993.

[27] V.N. Vatsa, M.D. Sanetrik, and E.B. Parlette. Development of a flexible and efficient multigrid-based multiblock flow solver; AIAA-93-0677. In *Proceedings of the 31st Aerospace Sciences Meeting and Exhibit*, January 1993.

[28] E. Zayas. *The Use of Copy-on-Reference in a Process Migration System*. PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh PA, April 1987.