Appears in: "Supercomputing '94," Nov. 1994. Reprinted by permission of IEEE.

Paging Tradeoffs in Distributed-Shared-Memory Multiprocessors*

Douglas C. Burger, Rahmat S. Hyder, Barton P. Miller, David A. Wood

Computer Sciences Department
University of Wisconsin–Madison
1210 W. Dayton Street
Madison, WI 53706 USA
wwt@cs.wisc.edu

Abstract

Massively parallel processors have begun using commodity operating systems that support demand-paged virtual memory. To evaluate the utility of virtual memory, we measured the behavior of seven shared-memory parallel application programs on a simulated distributed-shared-memory machine. Our results (i) confirm the importance of gang CPU scheduling, (ii) show that a page-faulting processor should spin rather than invoke a parallel context switch, (iii) show that our parallel programs frequently touch most of their data, and (iv) indicate that memory, not just CPUs, must be "gang scheduled". Overall, our experiments demonstrate that demand paging has limited value on current parallel machines because of the applications' synchronization and memory reference patterns and the machines' high page-fault and parallel-context-switch overheads.

1 Introduction

Demand-paged virtual memory is a ubiquitous feature of high-performance workstations but has been rarely supported on massively parallel supercomputers. Traditional paging systems "automatically" manage both main memory and CPU utilization by (i) allowing a program to execute with only part of its code and data in main memory, and (ii) executing other programs when the first must fetch missing code or data from disk. Conversely, parallel supercomputers—e.g., the Thinking Machines CM-5 and Intel Touchstone Delta—have generally required that programs fit in physical memory. By eliminating the uncertain overhead of demand paging, these systems maximize processor utilization but require programmers to explicitly manage memory.

Recently, however, massively parallel supercomputers have begun running modified workstation operating systems: the Intel Paragon and Convex SPP-1 run modified versions of Mach OSF/1 AD and the Meiko CS-2 runs a modified version of Solaris. Furthermore, clusters of workstations are emerging as increasingly popular alternatives to dedicated parallel supercomputers [3, 18, 7]. Parallel applications on these systems must co-exist with the operating system's demand-paged virtual memory.

In this paper we examine the performance of seven shared-memory scientific applications and argue that demand paging has limited value on distributed-shared-memory parallel computers. Running multiple programs that compete for memory on the same set of processor nodes is likely to cause unsatisfactory performance degradation. Furthermore, even when run by themselves, most of our applications show dismal performance unless nearly their entire data set resides in physical memory. The traditional benefits that paging provides on uniprocessors are diminished by the interactions between (i) the CPU scheduling discipline, (ii) the synchronization patterns within the application programs, (iii) the overheads of context switching and paging, and (iv) the page reference patterns of these applications.

Our results have implications for both CPU and

*This work is supported in part by NSF Presidential Young Investigator Award CCR-9157366, NSF Grants MIP-9225097, CCR-9100968, and CDA-9024618, Office of Naval Research Grant N00014-89-J-1222, Department of Energy Grant DE-FG02-93ER25176, and donations from Thinking Machines Corporation, Xerox Corporation, and Digital Equipment Corporation.

© ISSN 1063-9535. Copyright(c) 1994 IEEE. All rights reserved.

Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE. For information on obtaining permission, send a blank email message to info.pub.permission@ieee.org.

By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

memory scheduling policies. Specifically:

- Our results confirm the importance of gang CPU scheduling for parallel programs. Without gang scheduling, the performance of five of seven applications degrades badly.
- A page fault should not cause a parallel context switch (i.e., a gang-scheduling operation). The high parallel context switch overhead on current MPP machines (e.g, the CM-5) is greater than the cycles lost by spinning. This is especially true when paging to a memory server [9] or fast paging device rather than a traditional disk.
- The parallel programs we studied frequently access most of their data. All seven applications slow down by at least a factor of two—three of them slow down by more than a factor of eight—when we constrain the available physical memory to 90% of the data set size.
- Parallel computers require gang *memory* scheduling, not just gang CPU scheduling. Even a single memory-constrained node can degrade performance by over a factor of two.

The frequency of (blocking or spinning) synchronization is key to determining the appropriate CPU and memory scheduling policies. Coarse-grain parallel applications—those with little synchronization—can be scheduled exactly as sequential tasks [11]. As synchronization grows more frequent, however, the impact of delaying any one node increases dramatically. A page fault on one node can cause cascading delays on other nodes. Our results suggest that operating systems for massively parallel machines can provide virtual memory, but should determinedly attempt to prevent fine-grain parallel applications from making use of it.

The next section describes our simulation testbed, target system, and benchmarks, and Section 3 analyzes the performance of three CPU scheduling disciplines. Section 4 evaluates the performance of demand paging as we constrain the available physical memory and examines the applications' page reference patterns to explain their poor performance. Section 5 discusses the implications of these results and Section 6 summarizes our results and conclusions.

2 Methodology

The performance studies in this paper are based on a hypothetical distributed-shared-memory (DSM) system running seven shared-memory programs. The benchmark programs are scientific applications drawn from a variety of sources, including the Stanford SPLASH benchmarks [15], the NAS benchmarks [4], and the Schlumberger Corp. We simulated the execution of these programs on our hypothetical DSM system using the Wisconsin Wind Tunnel [13].

2.1 A DSM Machine Model

Our target hardware system contains 32 processing nodes, each with 33 Mhz SPARC CPU, 256-KB cache (4-way associative, 32-byte blocks, random replacement), and the local portion of the distributed shared memory. The nodes are connected by a point-to-point network (100 cycle constant latency) and coherence is maintained through a full-map directory protocol (i.e., dir_nNB [1]). The address space is globally shared, with 4KB shared pages assigned to nodes round-robin.

On a page fault, the target operating system selects a victim page using the Clock [5] algorithm. To maintain inclusion, the system invalidates all cached blocks from the victim page. The page-fault service time includes a fixed 1 ms overhead on the page's home node to model kernel overhead, the time to flush the victim's cache blocks, and the time to fetch the referenced page from the paging device. The home node resumes execution after initiating the page transfer and only processors that actually need the missing data stall or context switch.

2.2 Benchmarks

Our experiments used a suite of seven benchmark applications, summarized in Table 1. Of the seven, Barnes, Locus, Mp3d, and Ocean are from the SPLASH benchmark suite [15]. Appbt is a locally-parallelized version of one of the NAS Parallel Benchmarks [4]. Laplace was developed at Wisconsin [17], and Wave is a proprietary code from the Schlumberger Corporation. The last column in Table 1 contains the total number of data pages touched by the applications. All applications use a locally-modified version of the PARMACS macro package and assume a process-per-processor computation model (i.e., processes are always scheduled on the same processing node).

2.3 Simulation Environment

The Wisconsin Wind Tunnel (WWT) [13] is a parallel, discrete-event simulator for cache-coherent, shared-memory multiprocessors that runs on a Thinking Machines CM-5. By exploiting similarities between the

Name	Application Description (Input Data Set)	Data Pages
Appbt	Computational fluid dynamics ($32 \times 32 \times 32$, 3 iterations)	5861
Barnes	Hierarchical Barnes-Hut N-body (8192 bodies, 10 iterations)	477
Laplace	Boundary integral N-body problem (256 bodies, 20 elem./body, 10 iter.)	1735
Locus	VLSI standard cell router (Primary1.grin)	671
Mp3d	Monte Carlo rarefied fluid flow (32000 molecules, 50 iterations)	522
Ocean	Column-blocked 2D hydrodynamics (384×384, 1 day)	7613
Wave	3D acoustic finite difference ($48 \times 48 \times 48$, 20 iterations)	2711

Table 1: Benchmark Programs

hypothetical *target* system and the CM-5 *host*, WWT permits simulation of large applications.

To simulate a paging device, we partition the CM-5's physical memory into two components: *Logical Main Memory* (LMM) and *Logical Disk* (LD). On a page fault, the requested page is moved from LD to LMM and the victim page is moved from LMM to LD.

3 Scheduling

The goal of this study is to evaluate the utility of demand-paged virtual memory for parallel processors. In uniprocessors, the two primary benefits are improved physical memory utilization and improved CPU utilization. Paging improves memory utilization by allowing processes to run with only a subset of their code and data resident in main memory. It improves CPU utilization by permitting a higher degree of multiprogramming, achieved by switching to another process when one process incurs a page fault. In this section, we evaluate the feasibility of the second "benefit"—context switching on page faults—for distributed-shared-memory multiprocessors.

3.1 CPU Scheduling Policy

The CPU scheduling policy is central to this evaluation. Synchronization between the individual processes that make up a parallel job can result in one processor's delay (e.g., a page fault) causing cascading delays on other processors. Scheduling processors independently can magnify these delays by descheduling a process involved in the synchronization.

To address this problem, some previous studies have argued that parallel machines should employ gang scheduling, where all processing nodes simultaneously switch to the same parallel job [12, 10]. Others have argued for space-sharing [8], where processing nodes are dedicated to a parallel program until it completes. Space sharing can also be thought of as the limiting case of gang scheduling, with the scheduling quantum set to infinity. Our view is that future parallel systems must support time-sharing, albeit with large (e.g., 1 second) scheduling quanta.

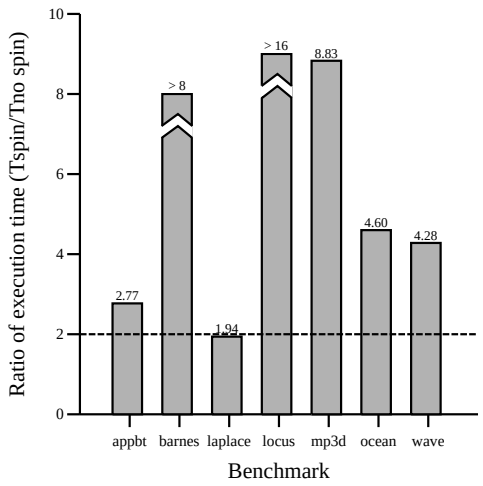

Figure 1: Effects of serial task interference on parallel execution time

To evaluate the need for gang scheduling in our environment, we used the WWT testbed to simulate the effect of sharing the CPU with some other process. In this experiment, each CPU spent half of its time executing and half its time spinning (to approximate a sequential process's execution). Each node alternates periods of execution (randomly generated, exponentially distributed with a mean of 100 ms) with periods of spinning (100 ms fixed delay). These results are optimistic, since we ignore both context switching overhead and cache pollution.

Figure 1 shows the normalized completion times of the seven applications for this experiment. In the absence of synchronization, each program should take roughly twice as long to complete, as each node spins half the time (on average). Our results show that for most applications, performance degrades significantly: five of seven applications slow down by over a factor of four, and three slow down by over a factor of eight. Running these application programs on a system without gang scheduling should be even worse because of context switch overhead and cache pollution.

3.2 Page Fault Scheduling Policy

The synchronous behavior that motivates gang scheduling also has ramifications for demand paging. When a page fault occurs, the operating system must determine whether to schedule another process, or to simply spin waiting for the page transfer to complete. In this section, we consider three policies that determine which action to take in the event of a fault:

SPIN: the faulting node spins until the page transfer completes,

LUG: the faulting node switches to a non-parallel process, and

PCS: all nodes synchronously switch to another parallel program.

Under the SPIN policy, the faulting node incurs the full latency of the page transfer. Remote nodes may also be delayed if they attempt to synchronize with the faulting process. Nonetheless, this policy is best if our objective is to minimize the parallel job's total execution time (i.e., its latency). The node's cache state is unperturbed, and the faulting process resumes immediately after the page transfer completes. However, because the faulting node's CPU is unutilized for the entire page-fault service time, this policy seems unlikely to provide the highest system throughput.

The LUG (*local-under-gang*) policy attempts to improve system throughput by having only the faulting node context switch to a local sequential job. LUG scheduling enables the faulting node to overlap useful computation with the page transfer. A local context switch on current microprocessors takes from 100 μs to 1 ms, leaving most of the service time free for computation. Although the LUG policy improves system throughput, it tends to increase the execution latency of the parallel job. The "local" process causes cache pollution by displacing entries from the node's cache and TLB. This results in a cold-start transient and degraded performance when the parallel process resumes.

The PCS (*parallel-context-switch*) policy attempts to improve system throughput by having all processing nodes synchronously switch to a new parallel program when one node incurs a page fault. The scheduler does not attempt to resume any process in the parallel program until after the page transfer completes. The PCS policy is closest to the traditional uniprocessor policy, and eliminates the synchronization delays that both the SPIN and LUG policies potentially incur. However, it has the drawback of incurring the high overhead of a parallel context switch on every page fault.

P	Number of processors in the system
T_{pcs}	Time required for a parallel context switch, including cache and TLB reload time
T_{pf}	Total latency required to service a page fault
N_{pf}	Global number of page faults incurred
F_{pf}	Fraction of page fault service latency that contributes to total (parallel) execution time
T_{epf}	Latency contributed by a page fault to the execution time of a parallel process
$T_{nopaging}$	Execution time of a benchmark without paging
T_{paging}^{policy}	Execution time of a benchmark with paging when using scheduling policy *policy*

Table 2: **Summary of Notation**

The remainder of this section focuses on evaluating the tradeoffs between the SPIN and PCS policies. The efficacy of the LUG policy depends highly on i) having a mix of parallel and sequential jobs and ii) the cache and TLB "footprint" [16] of the scheduled local process. While we believe such hybrid workloads may become common, characterizing their overheads is outside the scope of this paper.

The relative throughputs of the PCS and SPIN policies hinge on two factors: (i) the average overhead of a parallel context switch (T_{pcs}) and (ii) the average *effective* overhead of a page fault (T_{epf}) under the SPIN policy. The total execution time under the PCS policy (T_{paging}^{PCS}) is simply:

$$T_{paging}^{PCS} = T_{nopaging} + N_{pf} \cdot T_{pcs} \quad (1)$$

where $T_{nopaging}$ is the computation time in the absence of page faults. We assume that T_{pcs} includes the overhead of reloading the cache and TLB state after each parallel context switch and that page faults do not simultaneously occur on multiple nodes.

Under the SPIN policy, the total execution time (T_{paging}^{SPIN}) is

$$T_{paging}^{SPIN} = T_{nopaging} + N_{pf} \cdot T_{epf} \quad (2)$$

In a uniprocessor, the effective page fault overhead is simply the time to service a local page fault (T_{pf}); the entire page fault service time adds to the total execution time. But on a parallel processor, the effective overhead depends upon the frequency of synchronization. In a completely synchronous system, e.g., the MasPar MP-2, each page fault would delay every node for the full page fault service time ($T_{epf} = T_{pf}$). Conversely, if there were no synchronization and page faults were evenly distributed over all the processing nodes, then each page fault would increase execution time by roughly $T_{epf} = T_{pf}/P$, where P is the number of processing nodes. Real programs on our target distributed-shared-memory system will fall somewhere in between these two extremes.

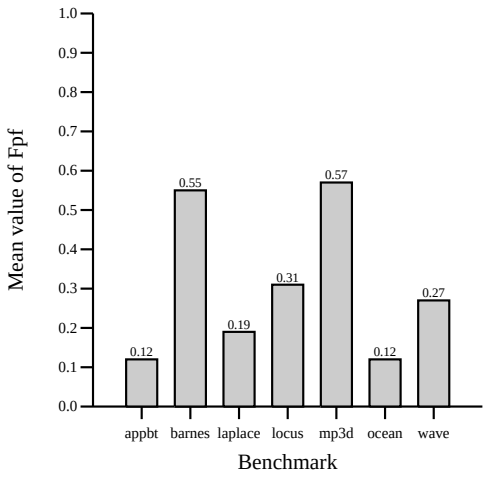

Figure 2: Mean values of F_{pf} under SPIN policy

We characterize the effective page fault overhead by F_{pf}, the (average) fraction of page fault service time that is added to a parallel program's execution time.

$$F_{pf} = T_{epf}/T_{pf} \tag{3}$$

Because F_{pf} depends upon the frequency and type of synchronization present in an application, we determine it experimentally using the WWT testbed. In this experiment, each simulated processing node periodically invalidates a virtual memory page, to approximate the effect of having that page frame assigned to another process. Pages are selected for invalidation using the Clock [5] algorithm; pages that have not been referenced recently (as calculated by Clock), are candidates for invalidation. Pages are invalidated at random times (exponential interarrival time distribution with mean 512 ms). We timed the execution of each program in two ways. First, we simulated the program without causing any page invalidations to measure $T_{nopaging}$. Then we simulated the program with page invalidations (as above) using the SPIN policy to measure T_{paging}^{SPIN}. We calculate F_{pf} as:

$$F_{pf} = (T_{paging}^{SPIN} - T_{nopaging})/(N_{pf} \cdot T_{pf}) \tag{4}$$

where N_{pf} is the number of page faults that occur during an execution.

Figure 2 graphs the values of F_{pf} for the seven applications. F_{pf} ranges from as low as 0.12 for Appbt and Ocean to 0.55 and 0.57 for Barnes and Mp3d, respectively. F_{pf} averages 0.30 over all applications, indicating that a page fault adds roughly one-third its service time to the total execution time, on average.

This estimate of F_{pf} helps us analyze the relative benefits of the SPIN and PCS policies. For the PCS policy to provide significant improvement over the SPIN policy, the round-trip overhead of the parallel context switch must be significantly less than T_{epf}:

$$T_{pcs} \ll F_{pf} \cdot T_{pf} \tag{5}$$

Assuming average disk service times of 16 ms and using the mean measured F_{pf} (just under one-third), the parallel context switch time must be less than approximately 5 ms. Unfortunately, current massively parallel processors incur substantial overhead for a full parallel context switch. For example, the Thinking Machines CM-5 incurs a minimum overhead of 4 ms, with typical times closer to 10 ms [14]. Under these assumptions, the SPIN policy is clearly superior—for both throughput and latency—to the PCS policy.

4 Performance of Virtual Memory

On a uniprocessor, demand-paged virtual memory "automatically" manages physical memory and allows processes to execute with only a subset of their code and data pages resident in memory. Paging both improves memory utilization and facilitates execution of programs whose data sets are larger than the available physical memory. However, in this section we show that this latter "benefit" does not apply to distributed-shared-memory multiprocessors. Specifically, the performance of our parallel applications degrades rapidly when physical memory is constrained. The poor performance results from two factors: (i) the high overhead of page faults caused by synchronization delays, and (ii) the pernicious reference patterns of these parallel applications.

The first factor follows directly from our results in Section 3. The synchronization inherent in these parallel applications makes page faults ten times more expensive than on a uniprocessor. This is because, on average, each page fault effectively stalls all 32 processing nodes for one-third of its service time.

In the remainder of this section, we examine the second factor. First, we quantify the performance of these applications when we constrain the available physical memory. Then we analyze the working set behavior and memory access patterns of these applications to understand why paging performs so poorly.

4.1 Simulated Application Performance

To evaluate the performance impact of demand paging, we simulated our benchmark applications on WWT using the Logical Main Memory (LMM) and

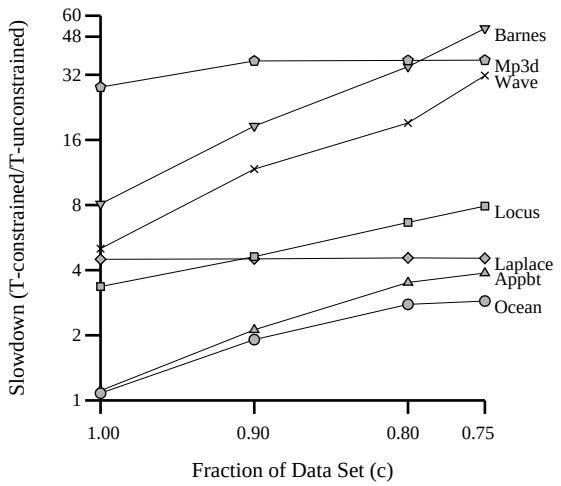

Figure 3: Constrained-memory performance

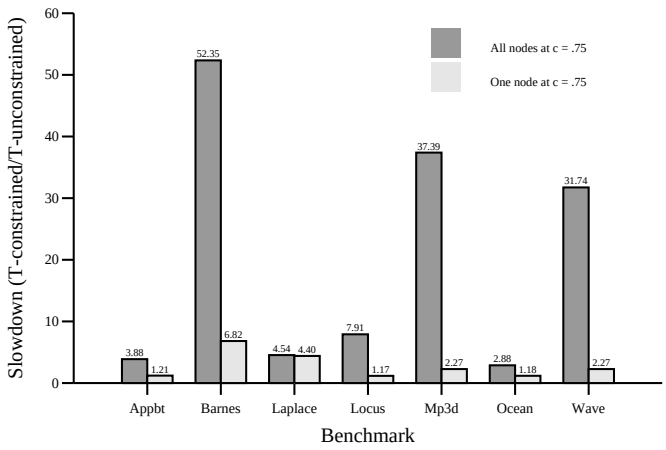

Figure 4: Performance with single constrained node

Logical Disk (LD) extensions. These experiments assume the SPIN scheduling policy discussed above, and vary the LMM size as a fraction, denoted c, of the total data set size. The data set size, denoted N, is simply the number of unique 4KB data pages referenced by an application (summarized in Table 1). We assume physical memory is uniformly distributed across processing nodes, so the per-node LMM size is simply $c \times N/P$, where P is the number of processing nodes. The metric of interest is *slowdown*, the execution time with constrained memory normalized by the execution time with unconstrained memory ($c = \infty$).

Figure 3 shows the slowdown as c decreases from 1.0 to 0.75 for a page transfer time of 16 ms (note that the y-axis uses a logarithmic scale). Our first observation is that programs running with $c = 1.0$ are noticeably slower than programs running with unconstrained physical memory. This occurs because while shared pages are distributed round-robin, private pages are allocated locally. Nodes with more private data will have more pages than physical frames, and thus may incur numerous page faults.

To quantify the memory imbalance, we measured the application performance for values of $c > 1.0$. Local memory partitions of up to 2.5 times as large are needed to eliminate the imbalance effect. Mp3d and Wave showed little slowdown at $c = 1.25$, Laplace at $c = 1.5$, Locus at $c = 2$, and Barnes at $c = 2.5$.

These results show that performance degrades rapidly when a parallel application lacks "sufficient" physical memory; that is, when c drops below an application-specific critical value. All seven applications slow down by at least a factor of two when physical memory is constrained to 90% of their data set size ($c = 0.9$). Further decreasing c to 0.75 has only moderate effect on 5 applications, but Barnes and Wave increase their slowdowns to 52 and 32, respectively.

We also performed a second experiment where memory was constrained on only a single node, chosen at random. The remaining 31 processing nodes had essentially infinite physical memory. Figure 4 shows that even a single memory-constrained node significantly degrades performance. Four of the applications slowdown by at least a factor of two.

4.2 Application Working Set Behavior

In the next two sections, we examine the memory reference patterns of the applications to identify why demand paging performs poorly. We first analyze the applications using the Working Set model of program behavior [6], which describes memory access patterns in terms of localities. A program's working set at time t with parameter τ is defined as the set of pages touched by the program during the last τ time units $(t - \tau, t)$. When applied to demand paging, the basic philosophy of Working Set is that "the past is a good predictor of the future". In other words, pages that have been recently referenced (i.e., are in the working set) are likely to be referenced in the near future and should therefore be kept resident. Page replacement algorithms based on this philosophy model have been effective on workstations and small multiprocessors.

To measure the working set behavior of our benchmark applications, we used WWT to collect page reference information. Each processing node tracks the set of pages referenced within the last τ. At each multiple of τ, we take the union of these sets (eliminating duplicates) to compute the global working set. Figure 5 shows how the global working set size varies over the programs' executions for three values of τ

($\tau = \{32, 128, 512\}$ ms). τ is usually a small multiple of the page fault service time; in these experiments τ spans a much greater range. Figure 5 also plots the number of pages referenced since the program began execution (infinite τ) and the total number of pages that will be referenced (dashed line).

The graphs in Figure 5 show two important results: (i) that the working set size of most applications grows rapidly as a function of τ, and (ii) many of the applications touch a large fraction of their total data set even for relatively small values of τ. If the rate of adding new pages to a program's working set is low, indicating high locality in its reference stream, we would expect the working size to grow to some reasonable value and then level off as τ increases. The lack of temporal locality in our application programs causes the global working set size to steadily grow as the parameter τ grows. Even when τ is large (512 ms, which is small relative to the execution time of the program), the working set sizes have either reached the data set sizes or are still rapidly increasing. In most cases, working set sizes for large values of τ follow the cumulative page curve closely.

These results indicate that demand paging will not be effective for our parallel benchmarks. Our applications exhibit little locality, and tend to frequently access large portions, if not all, of their data. The Working Set model indicates that nearly all of the application's data must be kept resident in memory.

4.3 Application Reference Behavior

To further understand the poor paging behavior, we examined the memory reference patterns of these applications. Barnes, Wave, and Mp3d exhibit the worst performance under constrained memory. Figure 5 shows that Barnes touches 70% of its data set every 32 ms and its entire data set every 2 seconds. The latter corresponds to the iteration time for this input. In each iteration, Barnes inserts "bodies" into an oct-tree that maintains their positions in three-space, then uses a hierarchical algorithm to compute the forces between the bodies. Thus each iteration references all bodies and all cells used to represent the oct-tree. Wave touches its entire data set even more frequently than Barnes: over 90% of its data pages are touched every 128 ms. This follows since Wave is essentially a 3D stencil computation and touches its entire data set in each phase of every iteration. Mp3d iterates over its molecules, calculating their new positions and moving them to new space cells as needed. This results in it touching nearly half its data pages every 32 ms and over 90% with a τ of 128 ms. Furthermore, all three applications frequently use barriers to synchronize their internal phases. This frequent synchronization explains the high values of F_{pf} observed in Section 3.

Laplace and Locus have relatively long initialization phases which exhibit good locality, followed by less well-behaved compute phases. Laplace's working set remains less than 20% during the initialization of the primary matrix A (which dominates Laplace's data set). During the computation phase, the algorithm iteratively solves the equation $X = A \cdot X + B$, touching most of the primary matrix in each iteration. Laplace also exhibits the most severe slowdown when there are just enough page frames to hold the data set ($c = 1.0$), This is caused by a highly unequal distribution of private data across the nodes, which cause some nodes to fault.

Locus exhibits similar two-phase behavior, but touches more of its data set in the second (compute) phase. Locus searches a VLSI standard cell, trying to find the lowest cost route for each wire. It touches roughly half its pages every 32 ms, and 75% every 128 ms. Furthermore, Locus has more synchronization than Laplace, resulting in a higher value of F_{pf}: Locus has an F_{pf} value of 0.31 versus 0.19 for Laplace.

Appbt and Ocean behave similar to one another. Matrices are the major data structures in both applications: five three-dimensional matrices in Appbt and 25 two-dimensional matrices in Ocean. Both applications touch essentially their entire data set in each iteration, yet exhibit reasonable short-term locality. Appbt touches less than 15% every 32 ms during the first phase of each iteration, during which it computes the block tridiagonal matrix. The second phase, Gaussian elimination, is less well behaved, yet still touches only half the pages every 128 ms. Ocean achieves its locality by referencing its matrices sequentially: each phase produces a matrix that is used as input to the next phase. This locality is one reason Appbt and Ocean exhibit relatively little slowdown due to constrained physical memory, despite touching their entire data sets each iteration. A second reason is minimal synchronization. Synchronization in these programs is dominated by barriers between phases within an iteration. When all processors have equally constrained memory, page faults on different nodes tend to overlap. The relatively low values of F_{pf} in Figure 2 corroborate this observation.

5 Discussion

The results of our experiments show that demand paging offers little benefit for distributed-shared-memory machines. The performance of our parallel shared-memory application programs degrades rapidly at the onset of paging; synchronization dependencies cause a page fault on one processing node to delay computation on other nodes. The obvious way to prevent propagation of these page fault delays is to perform a parallel context switch on each fault. However, the overhead of this operation is so high on current parallel machines that it is actually more efficient to simply spin. Coupled with these applications' large working set requirements and the high overhead of servicing page faults, our results suggest that demand paging should be avoided for parallel applications.

These results have important implications for the operating systems of parallel machines. Specifically, rather than managing each processor node as an independent workstation, the operating system should manage physical memory as a global resource. Furthermore, simple schemes that allocate fixed memory partitions on each node are unlikely to be effective, since many applications have unequal requirements for private pages. While there have been numerous proposals to manage processors globally [2], we believe these are the first results indicating the importance of doing so for physical memory.

Demand paging would become more attractive for parallel applications if we can decrease either the page fault service time (T_{pf}) or parallel context switch overhead (T_{pcs}). A lower page fault service time would reduce the magnitude of any synchronization delays, as well as decrease the probability that these delays even occur. T_{pf} could be reduced using standard techniques such as faster disks or dedicated paging memory (e.g., "solid-state disks"). Alternatively, Iftode, et al., have proposed dedicating some processing nodes as "memory servers" [9], which use their physical memories as fast paging stores. Such an approach might make use of the memory of idle or underutilized workstations in a network or cluster of workstations.

Faster parallel context switches would allow useful work to be overlapped with the page fault service time. This would increase system throughput, at the expense of delaying the completion of the parallel job. Decreasing the overhead of parallel context switches requires hardware support to i) allow the faulting node to quickly interrupt all other processors and ii) virtualize the network. However, even the CM-5—which has support for both—is quite inefficient.

Interestingly, these two improvements have opposite implications for the appropriate action to take on a page fault: lower service latencies support the SPIN (or LUG) policy, while lower parallel context switch overheads support use of the PCS policy.

While this work studies shared-memory applications, the concepts are applicable to fine-grain message-passing systems as well. Message-passing versions of these parallel scientific applications will have similar synchronization and communication characteristics. Thus we believe our results apply, at least qualitatively, to these other systems.

6 Conclusions

Demand-paged virtual memory attempts to optimize both CPU and physical memory utilization. The tradeoffs, which are well known for uniprocessors, are not nearly so clear for the next-generation of parallel processors, which will be built of workstation-like nodes and run commodity operating systems.

In this paper, we evaluated the feasibility of demand paging for such systems. We first enumerated and evaluated several CPU scheduling policies. As expected, we found that the fine-grain synchronization in our parallel applications makes gang scheduling necessary. Furthermore, the obvious solution of context switching all nodes to a new parallel job is ineffective because of the high overhead of parallel context switches on current machines. Instead, we found it more efficient to simply spin until the page fault completes. We also considered, but did not evaluate, a third policy (LUG) that executes a sequential job on the faulting node. This policy has significant potential if the workload consists of a mix of parallel and sequential jobs.

Second, we analyzed the efficacy of demand paging for our applications as the available physical memory is constrained. All seven applications slow down by at least a factor of two—three of them by more than a factor of eight—when the available physical memory is reduced to 90% of the data set size. We show that this behavior results from the large working sets of these applications, which frequently touch their entire data sets, and fine-grain synchronization.

We conclude from these results that operating systems for massively parallel machines can provide demand-paged virtual memory, but should schedule processors and memory to minimize paging. In particular, operating systems should use gang scheduling policies for both processors *and* memory.

Acknowledgements

We would like to thank Steve Reinhardt, Babak Falsafi and Alain Kagi for helping extend the Wisconsin Wind Tunnel for this study, Mark Hill for his helpful comments on a draft of this paper, and other members of the Wisconsin Wind Tunnel project for providing the infrastructure that made this study possible.

References

[1] Anant Agarwal, Richard Simoni, Mark Horowitz, and John Hennessy. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, 1988.

[2] Tom Anderson. NOW: Distributed supercomputing on a network of workstations, September 1993. Presentation at 1993 Fall ARPA HPC Software PI's meeting.

[3] Tom Anderson, David Culler, and David Patterson. A case for networks of workstations: NOW. Technical report, Computer Science Division (EECS), University of California at Berkeley, July 1994.

[4] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS parallel benchmarks. Technical Report RNR-91-002 Revision 2, Ames Research Center, August 1991.

[5] F. J. Corbato. A paging experiment with the Multics system. Technical Report MAC-M-384, MIT, May 1968.

[6] P. J. Denning. The working set model of program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[7] J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, 7(2):166–174, March-April 1993.

[8] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.

[9] Liviu Iftode, Kai Li, and Karin Petersen. Memory servers for multicomputers. In *Proceedings of 1993 Spring CompCon*, pages 538–547, February 1993.

[10] Scott T. Leutenegger. *Issues in Multiprogrammed Multiprocessor Scheduling*. PhD thesis, University of Wisconsin–Madison, August 1990.

[11] Scott T. Leutenegger and Xian-He Sun. Distributed computing feasibility in a non-dedicated homogenous distributed system. In *Proceedings of ACM Supercomputing '93*, 1993.

[12] John K. Ousterhout, Donald A. Scelza, and Pradeep S. Sindhu. Medusa: An experiment in distributed operating system structure. *Communications of the ACM*, 23(2):92–105, February 1980.

[13] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[14] Eric Sharakan. Personal communication., April 1994.

[15] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. Splash: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[16] D. Thiebaut and H.S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329, November 1987.

[17] F. Traenkle. Parallel programming models and boundary integral equation methods for microstructure electrostatics. Master's thesis, University of Wisconsin–Madison, 1993.

[18] Songnian Zhou, Jingwen Wang, Xiaohu Zheng, and Pierre Delisle. Utopia: A load sharing system for large, heterogeneous distributed computer systems. Technical report, Computer Systems Research Institute, University of Toronto, April 1992. CSRI Technical Report #257.

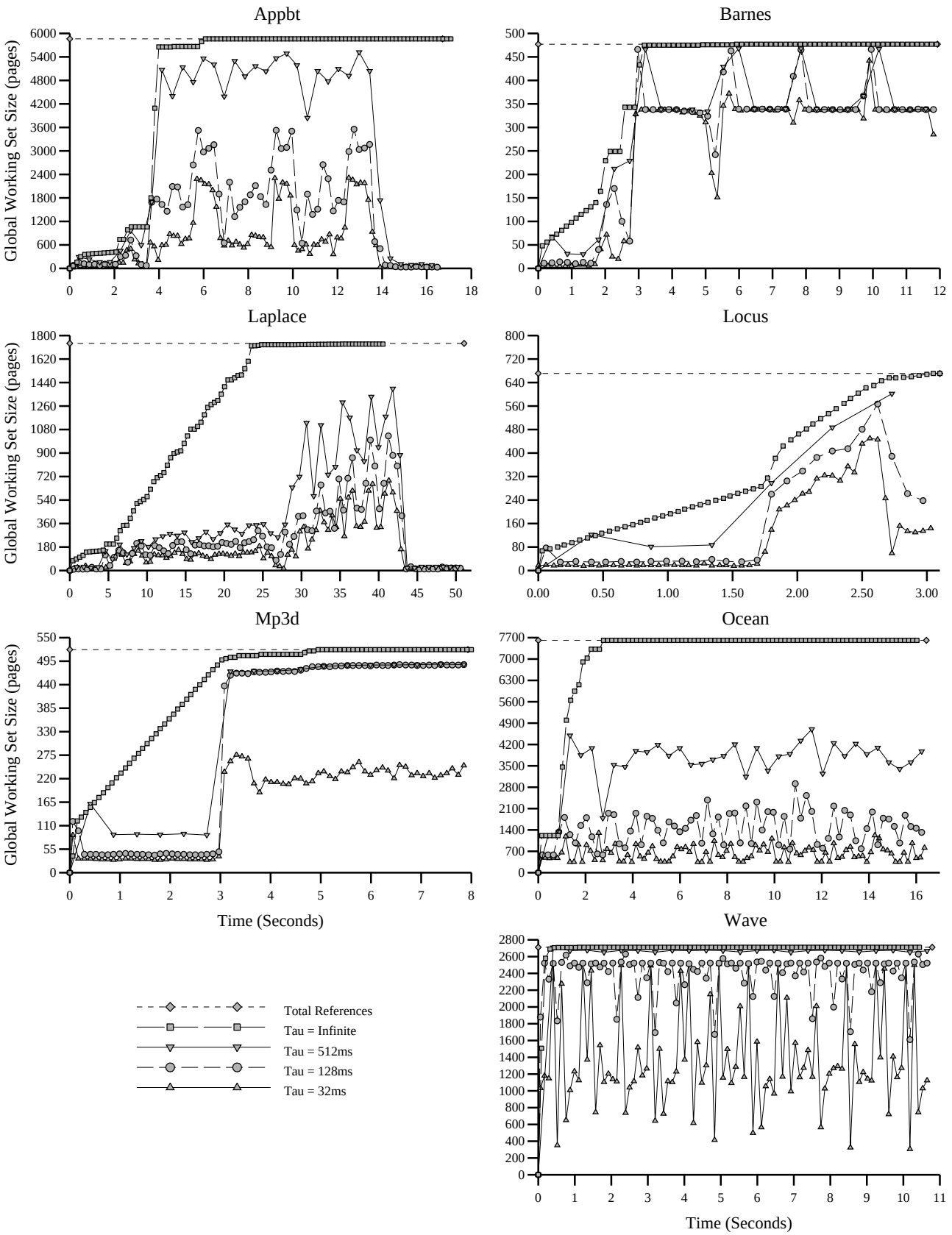

Figure 5: Global Working Sets