Swap File Organizations in Parallel Virtual Memory Systems *

Isaac D. Scherson, Verônica L. M. Reis and Fan Chen

Technical Report TR-96-46
October 7, 1996

Department of Information and Computer Science
University of California, Irvine,
Irvine, California 92717-3425, U.S.A.
Phone: 1-714-824-7713
FAX: 1-714-824-4056

{isaac,veronica,fchen}@ics.uci.edu

Abstract

This paper addresses the problem of providing a parallel virtual memory system with an efficient swap file structure. It has been observed, experimentally, that the way data is organized in a parallel I/O environment greatly influences the I/O performance [9, 6]. Therefore, if virtual memory is to be implemented in parallel machines, its disk swap space organization will be one of the key factors determining the virtual memory system's efficiency. We analyze different parallel file organizations in order to determine which one better suits the needs of a parallel virtual memory system.

Simulation results collected thus far indicate that the best strategy is to assign to each I/O node data that will be utilized mostly by neighbor processors. We observed that this approach better load balances I/O requests among I/O nodes and better explores inter-processor locality.

*This research was supported in part by the AFOSR under grant number F49620-92-J-0126, the CNPq under grant number 200358-92.8, the NASA under grant number NAG5-2561, and the NSF under grant numbers MIP-9106949 and MIP-9205737

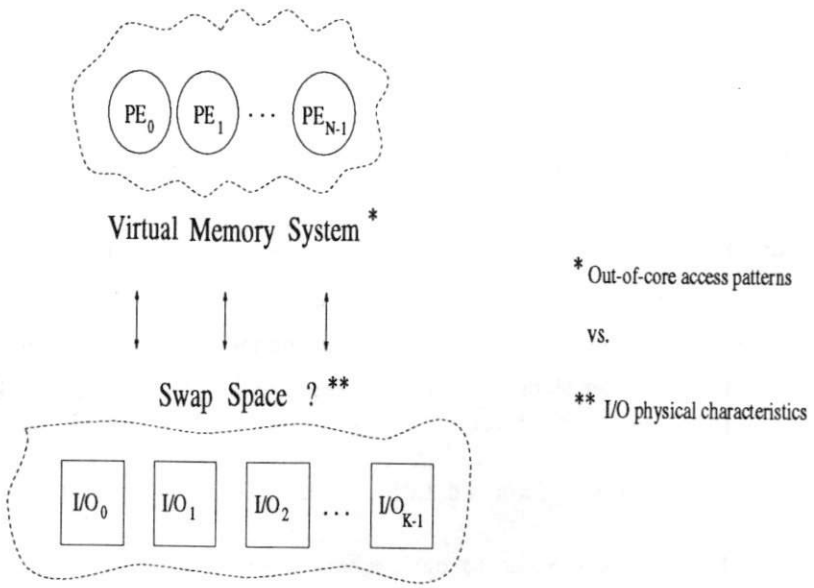

Figure 1. About the need of a swap space for a parallel virtual memory system.

1 Introduction

Modern parallel machines, such as the CRAY T3D, do not provide virtual memory: it is the programmer's responsibility to adapt the data to the physical memory available or to code any required out-of-core space. Previous attempts to provide virtual memory have not been successful [15]. Several unsolved issues have prevented virtual memory in parallel supercomputers from becoming reality. These issues include the lack of a complete understanding of locality of reference in parallel programs, how to efficiently manage the out-of-core data, and parallel I/O scalability and performance.

This paper examines the management of the out-of-core space required by a virtual memory system. In order to do so, we first empirically analyze data locality in parallel programs. We also present two virtual memory policies for managing a program's data space. After defining the virtual memory system, we propose different ways of spreading public data pages across the available I/O nodes: blocking and stripping (with various stripe sizes). The main contribution of this work is to show how different parallel file organizations influence the efficiency of a parallel virtual memory system. The analysis was done through simulation of public data access patterns of some parallel applications. Although we have tried to use applications with different data access patterns, we do not claim to have exhausted all possible situations but rather to have used different applications to demonstrate the need of flexibility in an eventual parallel virtual memory system. Figure 1 illustrates the proposed problem: given a parallel virtual memory system, how to organize a disk swap space in order to minimize its access time?

The paper is organized as follows. Section 2 defines the target environment for the proposed virtual memory system (physical machine and programming model) and describes the parallel I/O characteristics necessary to implement a swap file. Those characteristics are present in many commercial systems [7]. Section 3 presents two parallel virtual memory management strategies. These strategies were proposed in a previous paper [14] and define how a program's

public data is divided among and managed by the processors executing that program. Section 4 describes different organizations for a parallel swap file. The performance of the different file organizations are analyzed in Section 5.

2 Background and Parallel I/O Characteristics

The target architecture is MIMD, with physically distributed memory and I/O nodes equally accessible by all computing nodes through the interconnection network. Among the available commercial machines, CRAY T3D and Thinking Machines CM-5 fall in this category. The programming model is data-parallel and a globally addressable space (virtual shared memory), bigger than the available core memory, is provided by the parallel virtual memory system.

The virtual memory system divides a program's data in two categories:

- **private data**: the data that is accessed by one processor only. We include here one copy of the code and all local variables;
- **public data**: the data that is shared by more than one processor.

Because virtual memory for private data may be implemented in the same way as sequential virtual memory (as long as each processor has a local disk), the problem of managing the public data space cannot be resolved in the same way as private data. Public data will be used by more than one processor and this raises more complicated issues, as will be seen in the next section. We consider only public data throughout this paper.

The following characteristics should be provided by the I/O nodes to the parallel virtual memory system:

- **control declustering**: that is, to be able to define how data is to be distributed across multiple disks;
- **independent I/O**: each computing node should be able to access any page [1], at any location, independently of other accesses that other computing nodes might perform.

Current systems that provide these features include the Paragon PFS, the nCUBE and IBM Vesta [7].

Typically, massivelly parallel machines have more computing nodes than I/O nodes. An I/O node is a processor dedicated to I/O operations. It may contain more than one disk (or RAIDs [10]) as well as buffers.

We use performance figures from the CRAY T3D for disk latency (5.56 ms) and I/O node bandwidth (20 μs per word) [13] in our simulations.

3 A Parallel Virtual Memory

Implementing parallel virtual memory cannot be done by simply expanding sequential virtual memory. For instance, data-parallel programs do not have the same locality of reference characteristics as sequential programs do. Sequential programs present two types of locality: temporal and spatial. Temporal locality means that if one address

[1] A "page" is the minimum amount of data transferred from any I/O node to any computing node.

is referenced , it is likely to be referenced again in the near future. Spatial locality means that if one address is referenced, another address nearby is likely to be referenced in the near future. The main cause for both localities are loops: a loop will cause an instruction to be referenced many times (temporal locality) and will generally loop through some organized structure, element after element (spatial locality). When a data-parallel program is executed, loops are flattened across processors, loosing most of its locality. On the other hand, although public data uses up the majority of the program's memory, it only accounts for a small number of the references [2, 8]. These references, however, are much more time consuming, given that they usually imply inter-processor communications, therefore justifying any attempt to optimize them. It has also been observed that some parallel programs do present another type of locality: **inter-processor locality** [11]. Inter-processor locality means that although multiple processors may reference addresses that are not contiguous, the aggregate requests reference a contiguous space [11].

Other important issues when implementing parallel virtual memory are the management levels of such a system and the data migration policy. Management level addresses the data search strategy. For example, if a cache-miss occurs, where should the data be searched for next? Local main memory, some remote main memory or disk?

Data migration addresses the inter-processor communication problem and is tightly connected with management level. It must be decided whether public data pages are allowed to migrate among processors or should remain bounded to one processor.

In previous work, we suggested two management policies to implement parallel virtual memory [14]. They are:

- **Static page allocation**: the public data space is divided among processors such that each processor is responsible for fetching any of its assigned pages every time the page is requested and not found in its memory. For example, if any processor needs page P, and page P is under responsibility of processor A, then page P is either in processor A main memory or processor A must load it from disk. If a processor other than A needs data from page P, once P is loaded, A forwards the requested data.

- **Dynamic page allocation**: the responsibility of knowing the present location of a page is divided among processors. For example, if processor B needs page P, and page P is not in processor B's main memory, B will query that page's manager. Suppose the manager for page P is A. A will check its manager table and locate P. If P is swapped out, A will tell B to load the page and will update its manager table, showing B as the current owner of page P. Contrary to the previous scheme, pages will migrate across processors as they are needed.

  This scheme is an extension of the virtual shared memory management proposed by Kai Li [12].

Figure 2 outlines the two management policies proposed: in static page allocation, each processor owns a fixed sub-set of the public data pages and is responsible to provide any other processor with data from its sub-set. In the dynamic page allocation, the public data sub-set owned by each processor will change in time and the processor that needs some swapped out page is the one that fetches it from disk and keeps it in its memory until some other processor requests that page or it has to be swapped out to make room for other pages. Observe that there are two main differences between those two policies:

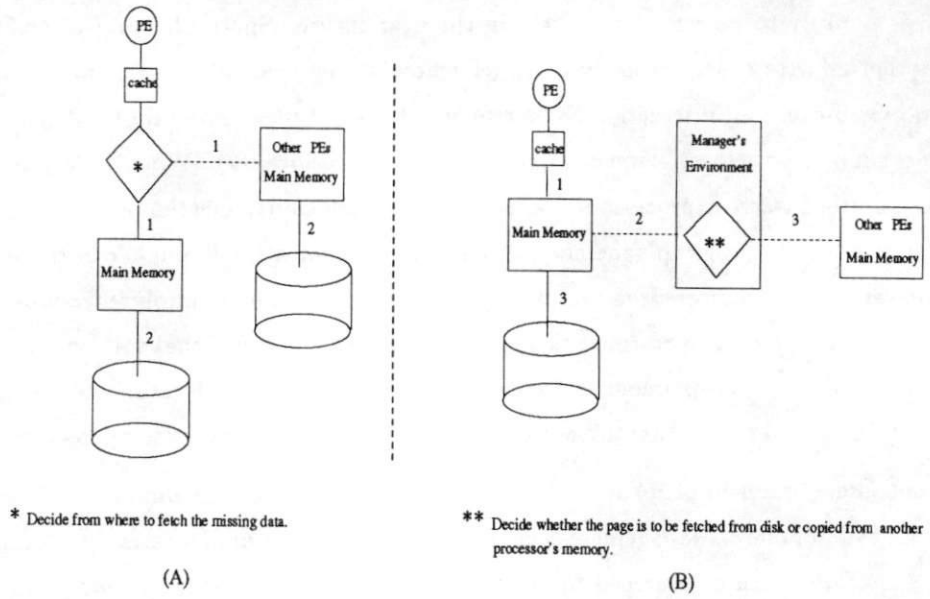

Figure 2. Parallel virtual memory management policies: static (A) and dynamic (B).

1. Who fetches a page from disk: is the static case, the owner of the page does it, whereas in the dynamic case, the processor that will use the page next does it;

2. The amount of data transferred through the interconnection network: is the static case only a few bites are sent, while in the dynamic case an entire page is sent.

4 Parallel File Organizations for Swap Files

One of the most observed phenomena in parallel I/O is the existence of a great variety in access patterns and the sensitivity of current I/O systems to these access patterns [9]. Because virtual memory systems are constantly performing I/O operations, they are not immune from this problem. Next we propose different ways of spreading public data pages across the available I/O nodes: blocking and stripping (with two different stripe sizes).

Suppose a parallel program public data space consists of n pages $[0...n-1]$.

Blocking will partition the n pages among the i I/O nodes in blocks of continuous pages (see Figure 3), such that I/O node J stores pages $[J \times (n/i)...(J+1) \times (n/i)]$. Therefore, if page P is not found in main memory, I/O node $\lfloor P \times i \div n \rfloor$ should fetch it.

Stripping means spreading pages across all I/O nodes. Pages may be spread individually (stripe 1) or in small blocks (stripe C). When using stripe 1, page P will be stored in I/O node $P \bmod i$ (see Figure 4), and when using stripe C, page P will be stored in I/O node $\lfloor (P \bmod (C \times i) \div C \rfloor$ (Figure 5).

Because the proposed virtual memory systems do divide public data among processors, stripping will be simulated with both stripe 1 and stripe $n/partition$, where n is the public data space size and *partition* is the number of processors executing the job. In other words, all public data assigned to a processor will be swapped out to the

same I/O node.

Because disk access patterns of parallel virtual memory systems are not known, it is not clear which swap file structure better suits it. Given the two virtual memory policies presented earlier and the three file organizations presented above, we performed event-driven simulations in order to shed some light in this matter. The simulations performed will be presented in the next section, as well as the results and their analysis.

5 Simulating Swap Files for Virtual Memory

In order to evaluate the different swap file organizations described in the previous section, an event-driven simulator was built. This simulator allowed us to compare the efficiency of different file organizations not only for different types of application in terms of public data access pattern and locality, but also for different number of I/O nodes in the system (ratio of number of computing nodes to number of I/O nodes).

The next subsection describes the simulator structure, assumptions and strategies. Subsection 5.2 presents the results, that are further analyzed in subsection 5.3.

5.1 Description

The simulator was written in Modsim, an event-driven simulation language.

Machine assumptions and OS environment: An MIMD, constant network delay, 16 processor machine was simulated. The performance figures of the T3D, such as IO latency and memory access times were used (for a complete list see Table 2). Each processor has equal access all I/O nodes. Requests received by I/O nodes are processed in a first-come-first-served basis. I/O nodes have an input queue (for incoming requests) and a constant number of buffers to store pages. If an I/O node receives a request for a page already buffered, disk access is avoided. When a page is fetched from disk, it is stored in the I/O buffer. Pages in the I/O buffer are replaced in a LRU-first basis. The number of I/O nodes varied across simulations.

The simulations performed did not allow time sharing. Each job loaded had the entire "machine", with each processor "executing" one virtual process (VP) [2] at a time. VPs were interrupted due to synchronization or page faults.

Barrier synchronization was used: all VPs synchronized after executing for some δT time (common to all). This δT was randomly defined at the end of the previous synchronization and was proportional to the percentage of public data access and the memory requirement of the job.

Page faults happened to both private and public data. Page faults to public data interrupt both the processor requesting the page and the page owner (static policy) or the page manager and page owner (dynamic policy).

In the static case, the requesting processor will block, waiting for its data while the owner will either send the data (if present) or page fault and suspend its VP's execution until the page is loaded.

[2] We call a virtual process the subset of a parallel program running on one computing node. A VP is composed of a copy of the data-parallel program, a copy of the private data set and some sub-set of the public data.

In the dynamic case, the requesting processor will first try to locate the page in its local memory. If it page faults, the manager is triggered while the requesting processor blocks. If the manager has the page, it sends it over. If a third processor has the page, it receives a message from the manager with instructions to send the page over to the requesting processor. Finally, if the page was never loaded, the requesting processor is told, by the manager, to do it.

Public data virtual address is represented as a triple (PE, Page, Offset), where PE is either the page owner (static policy) or the page manager (dynamic policy), Page is the page offset inside a block assigned to a PE and Offset is an address inside a page. Each VP stores its last public data access. When next access is to happen, its address is calculated based on the locality information of the application. Next access time is decided randomly and it is a function of the percentage of reference to PD of the application being simulated.

We need next to quantify locality. Again, we define locality as the probability of the next reference to public data to fall into the same page as the last one. The values used in the simulation were determined empirically and can be seen in Table 1.

In the static page allocation case, it is assumed that public data sent to another processor is used for read only and that all updates from other processors are done at the barrier synchronizations. *Release consistency* is assumed in the dynamic-page case. Every time a synchronization happens, all the copies of a page are removed and only the one that was fetched last is maintained. This was done to keep the two policies compatible: in the static case, if many processors request the same data at the same time, each one will receive one copy, so the same should be true to the dynamic case. We assume, therefore, *release consistency* [4], in which multiple copies are allowed until next synchronization point, when only the last update will be maintained and all other copies will be invalidated.

Only the fetching of pages (both local and public data) was simulated. A small overhead was considered when a page was copied back to disk (to update system tables). This was done because computing nodes must wait for I/O nodes to complete their task when loading a page, which is not the case when writing a page: they may proceed after asking I/O nodes to write a page without waiting for the completion of the task.

The simulations can be divided in terms of swap file organization, number of I/O nodes, virtual memory management policy and application type.

Three **swap file organizations** were simulated: blocking, stripe 1 and stripe C ($C = n/partition$). In all cases, the processor responsible for fetching the page will use the page's address to define to which I/O node to send the request. The reference address to I/O node identification conversion can be seen in Table 3.

Simulations were done for the following **number of I/O nodes**: 2, 4, 6 and 8.

The two **virtual memory management policies** simulated were static and dynamic page allocation, as previously defined.

Application characteristics: Three types of applications, whose traces were obtained from the literature, were simulated: WATER, MP3D and CHOLESKY(from the SPLASH benchmark [16]). From those applications we consider two characteristics: public data access pattern and public data locality of reference. Access pattern is the

Workload Characteristics	Distribution	
Job Ideal execution time(per VP)	Normal	avg 450 secs, std deviation 75 secs
Total Public Data Space (Size)	Uniform	[512 M , 50 G] pages
Percentage of public data access		18 (WATER), 29 (CHOLESKY) and 40 (MP3D)
Locality of reference of public data		0% (WATER), 2% (CHOLESKY) and 1% (MP3D)
Time between barrier synchs		Function of (% PD Access, PD Space)

Table 1. Statistical distributions used in the applications workload.

percentage of public data references out of all references. Locality is the probability that the next access to public data will be to the same page as the previous one.

The WATER problem simulates the evolution of a system of water molecules. This is done through short-range N-body [16]. The volume of water considered in the application is divided among processors and each processor works on the molecules in one region. Public data sharing happens in the "borders", when a processor needs data from the other side of the border in order to calculate its molecules movement. WATER presents no locality in terms of public data. Access to public data corresponds to 18% of all references [2].

MP3D simulates rarefied fluid flow, done through particle-in-cell, Monte Carlo methods [16]. Each processor is responsible for a subset of molecules and "follows" its subset through space. The active space considered is divided in "cells", for the purpose of efficient collision pairing: molecules can only collide with other molecules in the same cell at the same time. Public data sharing happens during collisions and during accesses to the space array. Because the partitioning of molecules is not related to their position in space, which changes considerably, each processor will access the space array in a non-regular pattern, many times sharing space cells with other processors. Therefore MP3D presents racing conditions, some locality, and its access rate to public data is 40% [2].

CHOLESKY factorizes a sparse positive definite matrix A into a lower triangular matrix L such that $A = LL^T$ [16]. The non-zero elements of the matrix are stored in an array with pointers to the first non-zero element of each column, with an auxiliary array storing the row number of each element. Sets of columns with similar non-zero structures are clustered into supernodes, the "data element" of this application. Only one step of the algorithm is used in the SPLASH benchmark, namely, the elimination of non-zero elements of certain rows in order to obtain a lower triangular matrix. In that step, a supernode may be modified by many processors until all modifications to that supernode are complete. It will be then placed in a "task queue" from where it will be removed and used by only one processor to modify other supernodes. CHOLESKY presents a little racing but good locality, and its access rate to public data is 29% [2].

Each application type was simulated separately for ten different program sizes in respect to execution time and public data space. Table 1 depicts the distributions used.

5.2 Results

Simulations ran until ten jobs were "executed", that is, simulations ran ten jobs to completion.

Figures 6 and 9 show the average completion time for WATER under different number of I/O nodes. Figure 6

Machine Characteristics	
Memory per PE for PD	8192 pages (8Mb)
Page size	128 words (1Kb)
Cache size	16 words
Number of PEs	16
Main memory latency	52 ns per word
Other PE memory latency	1 μs per word
IO bandwidth	20 μs per word
Disk latency	5.56 ms
IO node buffer size	4 pages
1 word	8 bytes
1 clock cycle	6.6 ns
OS overheads	
Context switch	8× main memory latency
Table update	between 1.4(local) and 3.4(remote) μs per entry
Enqueue delay	2.1 μs per object (job or VP)
To fetch a line from main memory and update table	2.232×10^{-6} secs.
To fetch a line from other PE main memory and update table	1.880×10^{-5} secs.
To fetch a page from other PE main memory and update table	1.315×10^{-4} secs.
To fetch a page from disk and update table	8.1249×10^{-3} secs.

Table 2. Hardware and Operating System numbers used in the simulation.

File Organizations	Public Data Reference	I/O node
Blocking	PE,Page	(PE*C+Page) DIV (TotSpace DIV IOnodes)
Stripe 1	PE,Page	(PE*C+Page) MOD IOnodes
Stripe C	PE,Page	PE MOD IOnodes

Table 3. Public data reference to I/O node identification conversion table.

shows the results for the execution under static page allocation, while Figure 9 depicts results for the dynamic page allocation. Figures 7 and 10 show results for CHOLESKY (static and dynamic page allocation, respectively). Finally Figures 8 and 11 refer to the MP3D application. **Blocking** outperformed the other two file organizations in most cases. WATER presents a more irregular pattern, and **stripe** C presents better results when the number of I/O nodes increases. **Stripe** C also performed better for MP3D with static page allocation and 8 I/O nodes.

Also observe that dynamic page allocation produced better results (smaller average completion times) in all cases. This is coherent with previous simulation results [14], where disk access time was assumed to be constant.

Figures 12, 13 and 14 present the average disk access time (the average I/O wait time among all processors) for each application and each page allocation policy. **Blocking** presents smaller average waiting times in all cases.

Figures 15, 16 and 17 present the maximum disk access times, among all processors, for each application and page allocation policy. Observe that the maximum waiting times are much bigger than the average ones. We also observed that although the maximum waiting times were not very frequent, they did influence the overall performance. This happened because of the characteristics of the programs executed: using barrier synchronization whenever a synchronization was required, which forced all processors to wait for the slowest ones.

While the graphs depicting average cases are "well behaved" for all applications, that is not the case with the graphs that show the maximum waiting times, where the increase in the number of I/O nodes did not linearly decrease the maximum waiting time. In some cases, the maximum waiting time increased, with the increase in the number

of I/O nodes! This behavior demonstrates the sensitivity of public data access to public data partitioning: when a different partitioning was used, more processors competed for the same I/O node, even though more I/O nodes were available.

5.3 Analysis of the Results

Results obtained thus far point to **blocking**, with dynamic page allocation, as the best option. We believe this is the case because **blocking** tends to store in the same I/O node pages that are more often accessed by a processor and its neighbors (roughly). While each processor cannot fetch more than one page at a time (thus minimizing the I/O queues), inter-processor locality between neighbor processors may be explored.

Although **stripe C** was designed with the division of the public data space among processors in mind, its results were not as good as **blocking**. This phenomenon might be caused by the spread, among I/O nodes, of pages originally "assigned" to neighbor processors, and therefore not exploring an eventual inter-processor locality. Notice, for instance, that although the disk access averages of **stripe C** and **blocking** are relatively close, the maximum disk access values are much bigger for **stripe C**.

We also observe irregular performances in the WATER application for the same page allocation policy/file organization. While **blocking** performs better with 4 I/O nodes, **stripe C** presents better results for both 6 and 8 I/O nodes (notice that this is true for both page allocation policies). WATER cycles through public data of the neighbor processors only: each region will need information of the "borders" only, and the "borders" will be in a limited number of neighbor processors [3]. This behavior, somehow, was sensible to stripping (and blocking) across different number of I/O nodes.

In terms of the number of I/O nodes in the system, we observe that for the simulated system with 16 processors, we have a significant performance improvement when moving from 2 and 4 I/O nodes to 6 I/O nodes. Not much improvement is acquired when moving from 6 to 8 I/O nodes, though. This result is compatible with previous observations [3, 5], that stated the existence of an ideal number of I/O nodes per number of processor nodes in a parallel system.

Notice the small difference, in terms of average completion time, among different file organizations for the same application and page allocation policy. We believe that this is the case due to the low latency of the T3D's I/O gateways. So, in order to better substantiate our results we intend, next, to repeat these experiments using another platform. Because of the cost/performance benefits and accessibility of networks of workstations (NOWs) as well as the recent results in implementing distributed shared memory in a NOW [1], NOWs seem to be the natural choice as the next platform to be experimented with. Our next step, then, is to repeat these experiments to a NOW consisting of 16 workstations and varying number of network file systems (NSF) servers.

[3] In the other two applications each processor cycles through the entire public data set (with different frequencies and strides).

6 Conclusions

This paper described two public data page management policies for a parallel virtual memory system (PVM). We proposed different parallel swap file organizations for this PVM and simulated their behavior for three parallel applications. The experiments performed indicate that **blocking** better load balances I/O requests among I/O nodes and better explores inter-processor locality. Thus, among the file organizations analyzed in this study, the one that better suits PVM seems to be **blocking**.

To organize the swap file in blocks is to maintain the compiler's decision to keep together data that will be mostly accessed by a processor and its neighbors.

Although parallel virtual memory system is not yet reality, we believe such a system not only is possible but very useful. Time sharing, for example, can be implemented with much more flexibility and efficiency if virtual memory is present. A PVM system will also relieve the user from the need of programming out-of-core and from the trouble of worrying with its data access patterns and related efficiency of the parallel I/O system. Notice that an eventual PVM can be further improved by prefetching techniques, which were not considered in this study's simulations.

Future work includes the experimentation of the proposed swap file organizations and page allocation policies for other platforms, and the analysis of alternative swap file organizations, as the understanding of locality of reference in parallel programs evolves. We also intend to analyze parallel virtual memory policies and swap file organizations in a time-shared environment.

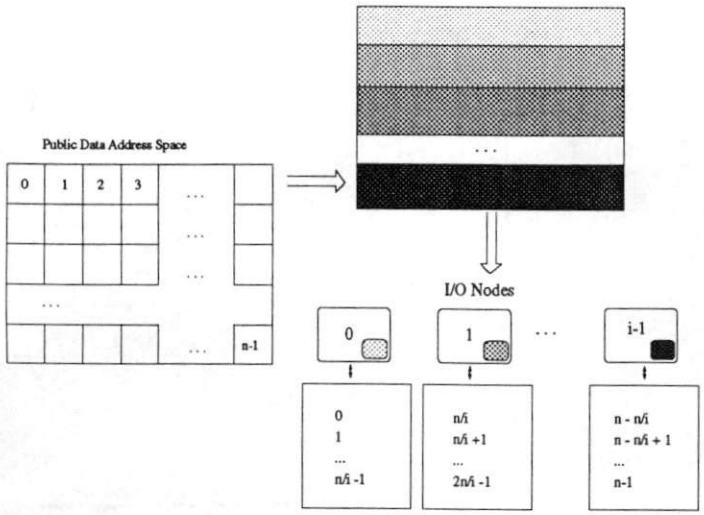

Figure 3. Blocking.

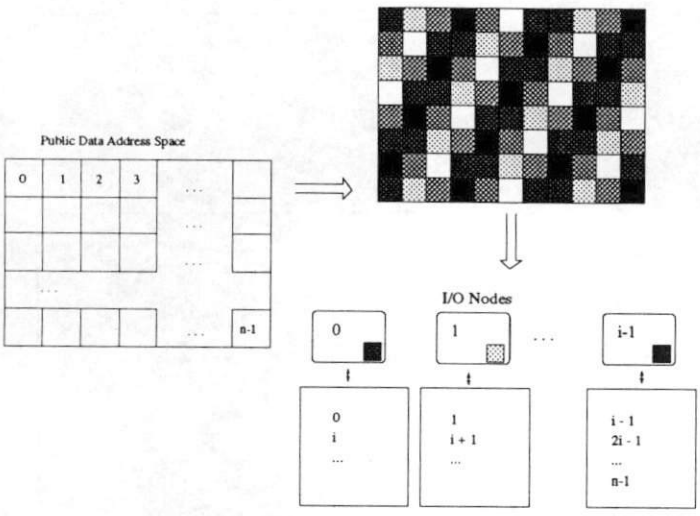

Figure 4. Stripping (1).

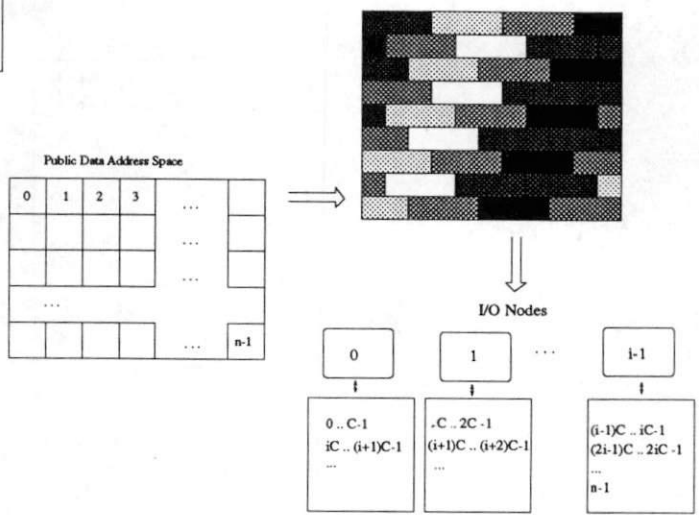

Figure 5. Stripping (C).

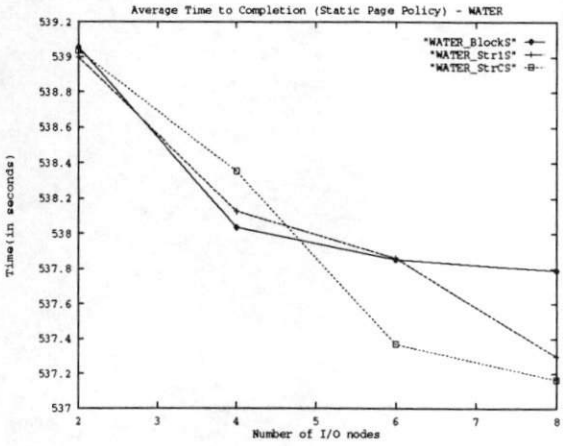

Figure 6.

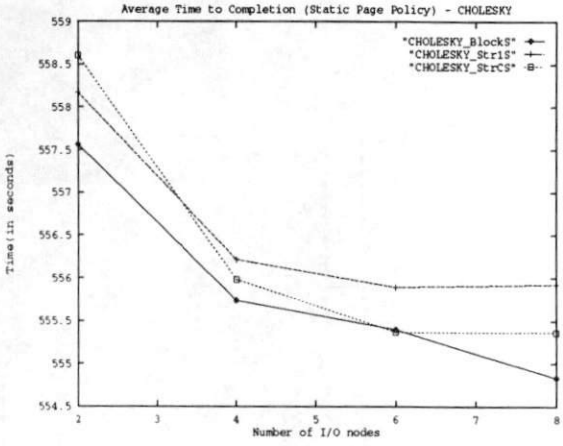

Figure 7.

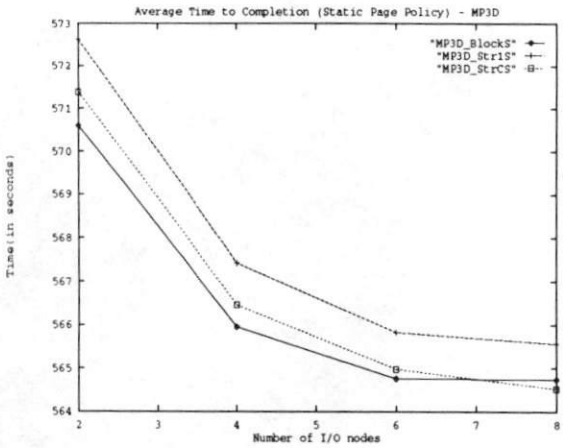

Figure 8.

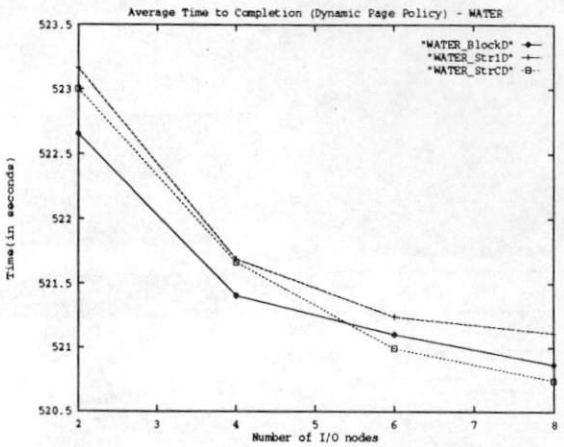

Figure 9.

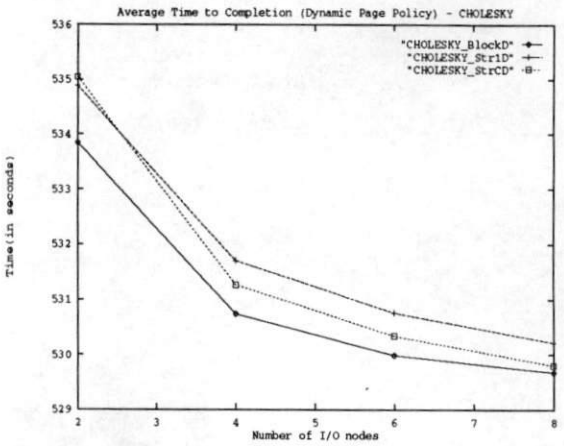

Figure 10.

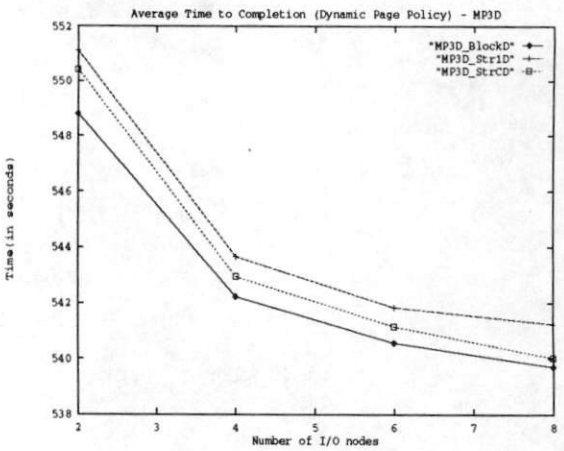

Figure 11.

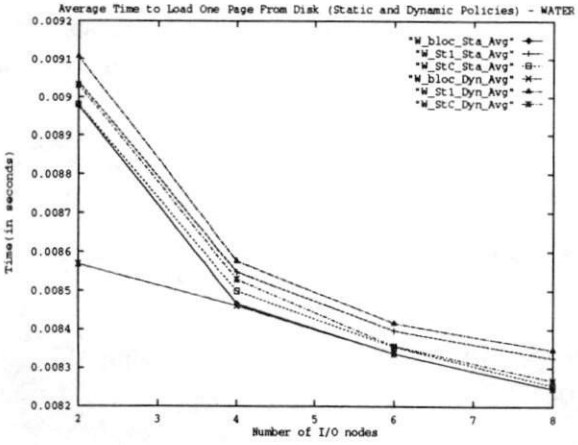

Figure 12.

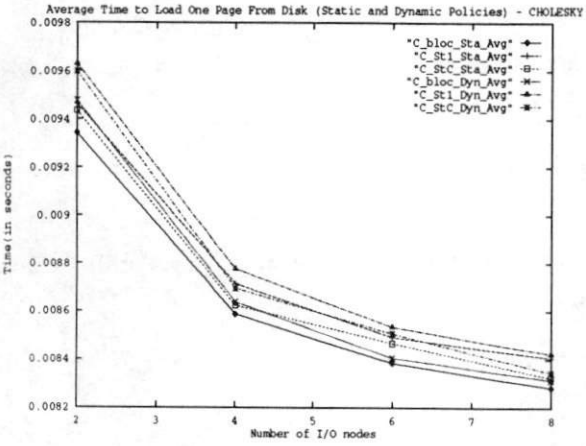

Figure 13.

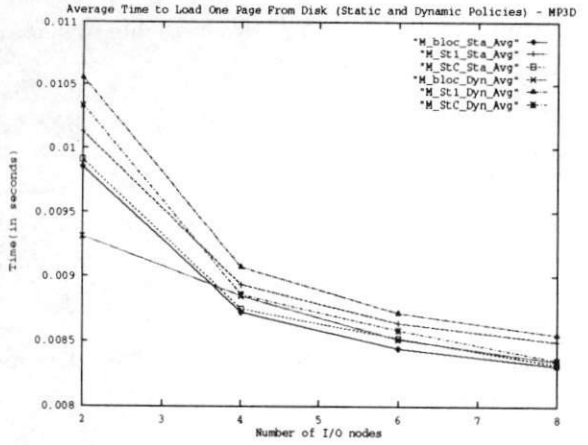

Figure 14.

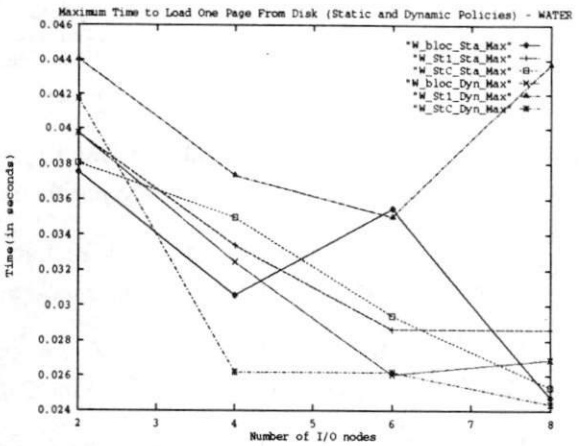

Figure 15.

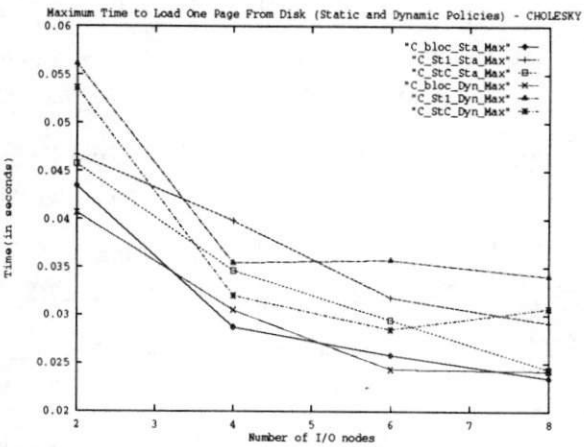

Figure 16.

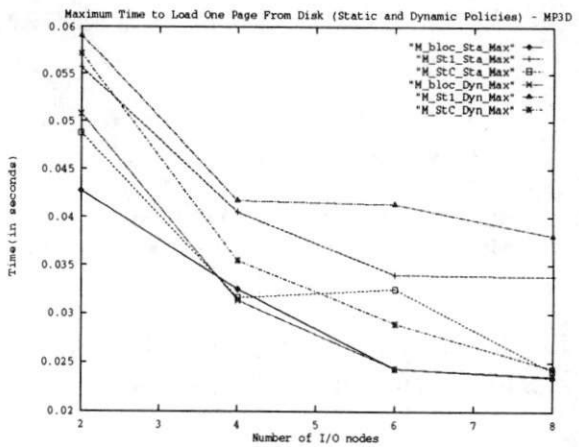

Figure 17.

References

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18 –28, February 1996.

[2] Luiz Andre Barroso and Michel Dubois. The Performance of Cache-Coherent Ring-based Multiprocessors. In *The 20th Annual International Symposium on Computer Architecture*, pages 268 – 277, May 1993.

[3] S. J. Baylor, C. Benveniste, and Y. Hsu. Performance evaluation of a parallel i/o architecture. In *Proc. International Conference on Supercomputing*, pages 404–413, Barcelona, July 1995.

[4] J. K. Bennet, J. C. Carter, and Z. Zwaenepoel. Munin: Distributed Shared Memory Using Multi-Protocol Release Consistency. *Lecture Notes on Computer Science 563*, pages 56 – 60, July 1991.

[5] Fan Chen, Veronica L. Reis, and Isaac D. Scherson. A Study of Parallel Input/Output Subsystems. In *APPT95, Beijing, China.*, September 1995.

[6] Thomas H. Cormen. *Virtual Memory for Data-Parallel Computing.* PhD thesis, Massachusetts Institute of Technology, February 1993.

[7] Thomas H. Cormen and David Kotz. Integrating Theory and Practice in Parallel File Systems. In *DAGS/PC Symposium on Parallel I/O and Databases*, pages 64 – 74, 1993.

[8] F. Darema-Rogers, G. F. Pfister, and K. So. Memory Access Patterns of Parallel Scientific Programs. *Performance Evaluation Review*, 15(1):45 – 58, May 1987.

[9] Juan Miguel del Rosario and Alok N. Choudhary. High-Performance I/O for Massively Parallel Computers: Problems and Prospects. *Computer*, 27(3):59 – 68, March 1994.

[10] P. M. Chen et al. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145 – 185, June 1994.

[11] Dror G. Feitelson, Peter F. Corbett, Sandra Johnson Baylor, and Yarsun Hsu. Parallel I/O Subsystems in Massively Parallel Supercomputers. *IEEE Parallel and Distributed Technology*, 3(3):33 – 47, Fall 1995.

[12] Kai Li. *Shared Virtual Memory on Loosely-coupled Multiprocessors.* PhD thesis, Yale University, October 1986.

[13] Wilfried Oed. The Cray Research Massively Parallel Processor System CRAY T3D. available by anonymous ftp from ftp.cray.com, November 1993.

[14] V. L. M. Reis and I. D. Scherson. A Virtual Memory Model for Parallel Supercomputers. In *10th International Parallel Processing Symposium (IPPS)*, pages 537 – 543, April 1996.

[15] Subhash Saini and Horst Simon. Enhancing Applications Performance on Intel Paragon through Dynamic Memory Allocation. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 232 – 239. Mississippi State University, October 1993.

[16] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *SIGArch Computer Architecture News*, 20(1), March 1992.