Parallel Virtual Memory for Time Shared Environments *

Verônica L. M. Reis, Luis Miguel Campos and Isaac D. Scherson

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

Abstract

This paper analyses the issues involved in providing virtual distributed shared memory for time-shared parallel machines. We study the performance of two different page management policies, namely, *static* and *dynamic* page allocation under two widely accepted scheduling policies: Gang scheduling and independent processor scheduling. The performance of each page management policy is studied under different replacement scopes (*local* versus *global* replacement).

Results obtained after extensive simulations show that dynamic page allocation performs better throughout all the environments simulated. We also observe a better performance of independent processor over gang scheduling as well as a similar performance between local and global replacement scope.

Sumário

Este artigo analisa os problemas envolvidos em se realizar memória virtual em ambientes distribuídos de memória logicamente compartilhada (DSM) em máquinas paralelas executando em tempo compartilhado. São analisadas as performances de duas políticas de gerenciamento de páginas: alocação *estática* e *dinâmica* sob duas políticas de escalonamento amplamente aceitas: grupo e independente. Estuda-se ainda o desempenho dessas duas políticas de gerenciamento sob diferentes estratégias de substituição de página: local e global.

Os resultados obtidos após várias simulações indicam alocação dinâmica de páginas como a melhor opção. Observamos também um melhor desempenho do escalonamento independente sobre o de grupo, e um desempenho equivalente entre as duas políticas de substituição de páginas estudadas.

*This research was supported in part by CNPq under grant number 200358-92.8, JNICT under grant number BD 538, AFOSR under grant number F49620-92-J-0126, NASA under grant number NAG5-2561 and NSF under grant numbers MIP-9106949 and MIP-9205737.

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 [8]. 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. In previous work [7], we proposed two strategies to manage public data pages in a parallel virtual memory system and analyzed their performance in a single-user environment. We now expand that analysis to a time shared environment.

In such an environment issues like job scheduling play a significant role in the definition of virtual memory policies. We study the influence of two widely accepted scheduling policies, namely, gang scheduling and independent processor scheduling on virtual memory policies. In this paper we analyze the performance of the previously proposed page management policies under those two scheduling policies. Also, in terms of page replacement strategies, we consider two replacement scopes: *local replacement*, where the page to be replaced is chosen among the resident pages of the process that generated the page fault, and *global replacement*, which considers all pages in main memory to be candidates for replacement, regardless of which process owns a particular page.

Even though we are aware of the importance of application's page misses induced by the OS itself as studied in [10] we did not take this fact into consideration during our simulations since reliable measurements of OS activity as related with cache utilization can not usually be taken from machine simulators.

In this paper we concentrate our research efforts on the set of policies offered by the operating system but we have to keep in mind that any virtual memory management system needs both hardware and software support. Any unrealistic assumption about the underlying hardware would undermine our conclusions and therefore we assume only the existence of hardware support available in commercial systems in all the algorithms presented.

The main contribution of this work is to show how different parallel virtual memory policies perform in a time shared environment. 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. Our results show that dynamic page allocation seems to be a better choice when implementing virtual distributed shared memory. This confirms our previous finding when analyzing the single user case [7], therefore dynamic page allocation works better both in single user and time-shared environments. We also observed a better performance of independent over gang scheduling. It is important to notice, however, that this result depends both on the availability of special features in the machine to quickly context-switch between jobs and to buffer, for example, incomming messages for a job not currently executing.

The paper is organized as follows. Section 2 defines the target environment for the proposed virtual memory systems (physical machine, operating system and programming model). Section 3 presents the two parallel virtual memory management strategies. Those strategies define how a program's public data is divided among and managed by the processors executing that program. This section also describes the issues involved in implementing the proposed parallel virtual memory in a time shared environment. Definition of the scheduling, page replacement and resident set management strategies under study are presented. The performance of the different policies are analyzed in Section 4.

2 Background

The target architecture is MIMD, with physically distributed memory. Among the available commercial machines, CRAY T3D, Thinking Machines CM-5 and IBM SP2 fall in this category. The programming model is data-parallel and a globally addressable space (distributed 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 independent access to disk), we consider only public data (PD) throughout this paper. The problem of managing the PD space cannot be resolved in the same way as private data since PD will be used by more than one processor. This fact raises more complicated issues, as will be seen in the next section.
We evaluate the schemes proposed through event-driven simulations using performance figures from the CRAY T3D [6]. We simulate applications from the SPLASH benchmark [9], whose traces were taken from the literature [1] added with profiling done by us.

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 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 [1, 3]. 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. Inter-processor locality means that although multiple processors may reference addresses that are not contiguous, the aggregate requests reference to a contiguous space [4].
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 PD 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 [7]. They are:

- **Static page allocation**: the PD 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 [5].

Figure 1 outlines the two management policies proposed.
In previous work, we analyzed those policies in a single-user environment and compared results against a system with virtual memory disabled (in which pages were rolled-in before computation started). We now expand our experiments to a more realistic situation (and one in which virtual memory is more desirable!): a time shared environment. Here, we are required to schedule more than one job at a time, so we must decide how to do it. Currently, most MIMD parallel machines use either gang scheduling (CM-5 and Intel Paragon) or independent processor scheduling (Meiko/ts), so we tested our policies under both scheduling strategies. Another issue is to decide the page replacement scope: if some page is to be swapped out, which

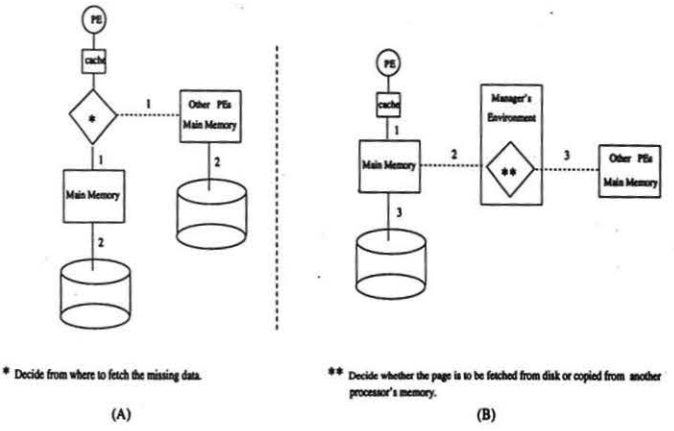

Figure 1. Parallel virtual memory management policies: static (A) and dynamic (B) page allocation.

pages should be considered? Only the pages of the job requiring space or all pages from all jobs? Again, we tested our policies under the two possibilities.
Next section describes the simulations performed and presents the results obtained.

4 Simulating Time Shared Virtual Memory

In order to evaluate the different virtual memory policies proposed in the previous section, an event-driven simulator was built. This simulator allowed us to compare the efficiency of different policies for different types of application in terms of public data access pattern and locality, but also for different scheduling policies.
The next subsection describes the simulator structure, assumptions and strategies. Subsection 4.2 presents the results, that are further analyzed in subsection 4.3.

4.1 Description

Machine assumptions and OS environment: An MIMD, constant network delay, 16 processor machine was simulated. The performance figures of the T3D, such as I/O latency and memory access times were used (for a complete list see Table 2). Each processor has equal access to all I/O nodes. Requests received by I/O nodes are processed in a first-come-first-served basis. I/O nodes have a constant delay in serving each request (the average delay of the T3D).
The simulations performed time shared among 2, 3 or 4 jobs at a time. Each job loaded had one virtual process (VP) [1] "executing" in each processor. VPs were interrupted due to end of time slice, synchronization or page faults.

[1] 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.

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 happen 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 VP will block, waiting for its data while the owner will either send the data (if present) or page fault.
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 VP 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*), were *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.
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* [2], 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).
The simulations can be divided in terms of scheduling strategy, page replacement scope, number of jobs time shared, virtual memory management policy and application type.
Two scheduling strategies were simulated: gang and independent processor. In gang scheduling, all processors are given to the same job per time slice. If some of the VPs of that job happen to be blocked, the processors responsible for executing those VPs will remain idle. In independent processor scheduling, each processor may execute VPs from different jobs as they become available to run. See figure 2.
In terms of page replacement scope, our LRU varied between considering only pages of the job that page faulted (local) or all pages from all jobs (global).

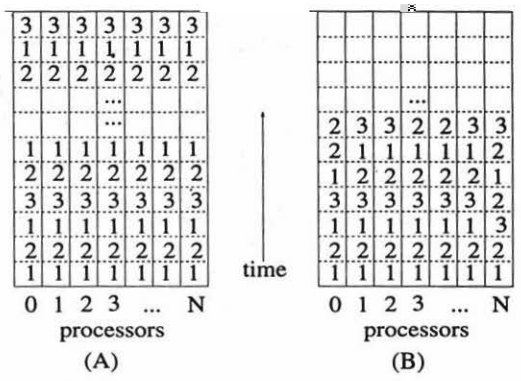

Figure 2. Gang (A) and Independent Processor (B) Scheduling.

Simulations were done for the following number of time sharing jobs: 2, 3 and 4.
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 [9]). From those applications we consider two characteristics: public data access pattern and public data locality of reference. Access pattern is the percentage of public data references out of all references.

The WATER problem simulates the evolution of a system of water molecules. This is done through short-range N-body [9]. 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 [1].

MP3D simulates rarefied fluid flow, done through particle-in-cell, Monte Carlo methods [9]. 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% [1].

CHOLESKY factorizes a sparse positive definite matrix A into a lower triangular matrix L such that $A = LL^T$ [9]. 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.

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), 40 (MP3D)
Locality of reference of public data		0% (WATER), 2% (CHOLESKY), 1% (MP3D)
Time between barrier synchs		Function of (% PD Access, PD Space)

Table 1. Statistical distributions used in the applications workload.

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
I/O bandwidth	20 μs per word
I/O latency	5.56 ms
1 word	8 bytes
1 clock cycle	6.6 ns
OS overheads	
Context switch	8× main memory latency
Table update (μs per entry)	between 1.4(local) and 3.4(remote)
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 m. memory and update table	1.880×10^{-5} secs.
To fetch a page from other PE m. 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.

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% [1].
Each application type was simulated separately for six simulation hours and different program sizes in respect to execution time and public data space. Table 1 depicts the distributions used.

4.2 Results

Simulations ran for six simulation hours. The first hour was disconsidered in order to avoid initialization effects, therefore the results presented here reflect a 5-hour window of the environments simulated.

	T.S.	Static		Dynamic	
		Independent	Gang	Independent	Gang
WATER	2	968.85	1250.88	961.21	1167.63
	3	1438.83	1886.22	1427.10	1737.63
	4	1913.62	2548.81	1891.04	2314.69
CHOLESKY	2	971.32	1370.35	965.99	1303.12
	3	1447.22	2079.87	1430.82	1951.38
	4	1927.98	2761.08	1892.18	2595.57
MP3D	2	964.70	1517.54	985.06	1476.44
	3	1437.65	2281.29	1433.44	2191.16
	4	1913.84	3060.68	1896.39	2924.82

Table 3. Average completion times in seconds.

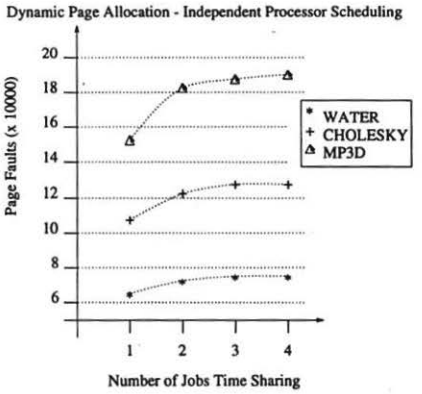

Figure 3. Average number of page faults per processor.

Table 3 shows the average completion time for all programs in all environments, for the global page replacement. We did not include the local page replacement values because they are very close to the global ones. We observe that dynamic allocation gives better results in all experiments. We also observe that Independent Processor is a more efficient scheduling strategy. Figure 3 depicts the average number of page faults per processor under dynamic allocation and independent processor scheduling for different number of time sharing jobs. We observe that there is a big increase in the number of page faults between one job executing alone and two jobs time sharing, but that this increase is not that big between two and three jobs time sharing and almost inexistent between three and four jobs time sharing.
Figure 4 shows the number of memory references for the CHOLESKY application, for both static and dynamic allocation. Notice that the number of disk accesses is much smaller in the dynamic case.

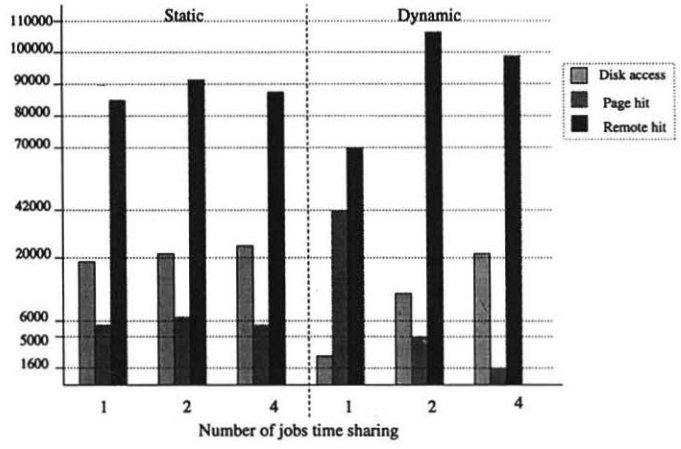

Figure 4. Total number of references of the CHOLESKY application (Average per processor).

4.3 Analysis of the Results

Results obtained thus far point to dynamic allocation as a marginally better option. The performance increase of the dynamic page allocation over the static one is usually not more than 10%, we believe, because of the low latencies observed in the CRAY T3D. On the other hand, we believe this performance increase is due to the fact that each processor is responsible for fetching the pages it is going to use next, instead of interrupting some other processor and have that other processor fetch the page (as is the case in static allocation). Also, we observed that the number of disk accesses is smaller in the dynamic case. This is so because pages are transferred to a new owner whenever necessary, so the pages needed locally will not compete and displace the pages needed remotely.

As expected, gang is not as efficient as independent processor scheduling. However, it is more widely used by commercial machines for practical reasons such as the fact that many parallel programs are written with the assumption that it owns the entire machine, and that view is maintained by gang scheduling. Also, independent processor scheduling puts an extra burden on the operating system in terms of buffering and managing incoming messages for VPs not currently executing.

As already stated, the results for local and global page replacement were very similar. In practical terms, local replacement is more often utilized because it is easier to manage and less prone to cause starvation.

5 Conclusions

This paper described and analyzed the simulations performed to evaluate the behavior of two proposed parallel virtual memory policies in a time shared environment.

The results demonstrated a better performance of the dynamic page allocation, which is similar to the results obtained in a single user environment. We also showed that independent processor scheduling performs better across the board than gang scheduling, which was already expected. We must notice, however, that independent scheduling incurs extra burden on the operating system and that it may not perform as good in environments with big context-switch overheads.
We intend, next, to analyze the two page management policies presented in another platform, such as a network of workstations.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

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

[7] Veronica L. M. Reis and Isaac D. Scherson. A Virtual Memory Model for Parallel Supercomputers. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 537 – 543, April 1996.

[8] 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.

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

[10] Josep Torrellas, Anoop Gupta, and John Hennessy. Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System. In *ASPLOS-V*, pages 162 –174, 1992.