Memory Coherence in Shared Virtual Memory Systems[1]

Kai Li and Paul Hudak

Department of Computer Science
Yale University
New Haven, CT 06520

Abstract

This paper studies the memory coherence problem in designing and implementing a shared virtual memory on loosely-coupled multiprocessors. Two classes of algorithms for solving the problem are presented. A prototype shared virtual memory on an Apollo ring has been implemented based on these algorithms. Both theoretical and practical results show that the memory coherence problem can indeed be solved efficiently on a loosely-coupled multiprocessor.

1 Introduction

The benefits of a virtual memory go without saying, and almost every high-performance sequential computer in existence today incorporates one. Virtual memories are so useful that it is hard to believe that parallel architectures would not also benefit from them. Indeed, one can easily imagine how virtual memory would be incorporated into a *shared*-memory parallel machine, since the memory hierarchy need not be much different from that of a sequential machine. On the other hand, on a "loosely-coupled multiprocessor" in which the physical memory is *distributed*, the implementation is not as obvious, and to our knowledge no such implementation exists.

The *shared virtual memory* described in this paper provides a virtual address space which is shared among all processors in a loosely-coupled multiprocessor system, as shown graphically in Figure 1. The shared memory itself exists only *virtually*. Application programs can use it in the same way as a traditional virtual memory, except, of course, that processes can run on different processors in parallel.

The shared virtual memory that we will describe not only "pages" data between physical memories and disks, as in a conventional virtual memory system, but it also "pages" data between the physical memories of the individual processors. Thus data can naturally *migrate* between processors on demand. Furthermore, just as a conventional virtual memory also pages *processes*, so does the shared virtual memory. Thus our approach provides a very natural and efficient form of *process migration* between processors in a distributed system, normally a very difficult feature to implement well (and in effect subsuming the notion of *remote procedure call*).

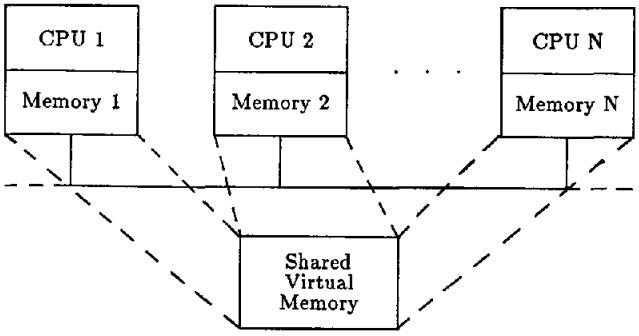

Figure 1: Shared virtual memory mapping.

The main difficulty in building a shared virtual memory is solving the *memory coherence problem*. This problem is similar to that which arises with conventional caches (see [14] for a survey), but in particular with *multicache* schemes for shared memory multiprocessors [16,1,7,18,6,19,13]. In this paper we concentrate on the memory coherence problem for a shared virtual memory. A number of algorithms are presented, analyzed, and compared. Several of the algorithms have been implemented on a local area network of Apollo workstations. We present experimental results on non-trivial parallel programs that demonstrate the viability of shared virtual memory even on very loosely-coupled systems such as the Apollo network. Our success suggests a

[1]This research was supported in part by NSF Grants MCS-8302018 and DCR-8106181.

radically different viewpoint of such architectures, in which one can exploit the total processing power and memory capabilities of such systems in a far more unified way than the traditional "message-passing" approach.

2 Design Choices for Memory Coherence

Our design goals require that the shared virtual memory be *coherent*. A memory is coherent if the value returned by a read operation is always the same as the value written by the most recent write operation to the same address. Coherence can be maintained if a shared virtual memory satisfies the following single constraint:

- A processor is allowed to update a piece of data only while no other processor is updating or reading it.

This allows many processors to read a piece of data as long as no other processor is updating it, and is a form of the well-known readers/writers problem.

There are two design choices that greatly influence the implementation of a shared virtual memory: the granularity of the memory units, and the strategy for maintaining coherence.

2.1 Granularity

The size of the "memory units" that are to be coherently maintained is an important consideration in a shared virtual memory. We discuss in this section several criteria for choosing this granularity.

In a typical loosely-coupled multiprocessor system, sending large packets of data (say one thousand bytes) is not much more expensive than sending small ones (say less than ten bytes) [15]. This is usually due to the typical software protocols and overhead of the virtual memory layer of the operating system. This fact makes relatively large memory units seem feasible.

On the other hand, the larger the memory unit, the greater the chance for contention. Memory contention occurs when two processors attempt to write to the same location (as in a shared memory system) as well as when two processors attempt to write to different locations in the same memory unit. Although clever memory allocation strategies might minimize contention by arranging concurrent memory accesses to locations in different memory units, such a strategy would lead to the inefficient use of memory space and introduce an inconvenience to the programmer. Thus the possibility of contention pushes one toward relatively small memory units.

A suitable compromise in granularity is the typical *page* used in a conventional virtual memory implementation. The page sizes of today's computers vary, typically from 256 bytes to 2k bytes. Choosing this size of a memory unit has several advantages. First, experience has shown that such sizes are suitable with respect to contention, and by our previous argument they should not impose undue communications overhead as long as a page can fit into a packet. In addition, such a choice allows us to use existing page-fault schemes (i.e., hardware mechanisms) that allow single instructions to trigger page-faults and trap to appropriate fault handlers. This can be done by setting the access rights to the pages in such a way that memory accesses that could violate memory coherence cause a page fault, and thus the memory coherence problem can be solved in a modular way in the page fault handlers.

Part of the justification for using page size granularity, of course, is that memory references in sequential programs generally have a high degree of *locality* [3,4]. Although memory references in parallel programs may behave differently from those in sequential ones, a single process remains a sequential program, and should exhibit a high degree of locality. Contention among parallel processes for the same piece of data depends on the algorithm, of course, but a common goal in designing parallel algorithms is to minimize such contention for optimal performance.

2.2 Memory Coherence Strategies

It is helpful first to consider the spectrum of strategies one may choose from to solve the memory coherence problem. These strategies may be classified by the way in which one deals with *page synchronization* and *page ownership*, as shown in Table 1.

Page synchronization

There are two basic approaches to page synchronization: *invalidation* and *writeback*. In the *invalidation* approach, if a processor has a write fault, the fault handler will copy the true page containing the memory location, invalidate all other copies of the page, change the access of the page to write, and return to the faulting instruction. After returning, the processor "owns" that page and can proceed with the write operation and other read or write operations until the page ownership is relinquished to some other processor.

In the *writeback* approach, if a processor has a write fault, the fault handler will write to all copies of the page, and then return to the faulting instruction. In a sense this approach seems ideal in that it supports the broadest notion of sharing (indeed it simulates a centralized shared memory!), but note that *every* write to a shared page will generate a fault on the writing processor and update *all* copies. Clearly doing these updates will be very expensive, and algorithms using writeback do not seem appropriate for loosely coupled multiprocessors. Thus we do not consider them further in this paper, as indicated in Table 1.

Page ownership

The ownership of a page can be handled either *statically* or *dynamically*. In the static approach, a page is always owned by the same processor. This means that other processors are never given full write access to the page; rather they must negotiate with the owning processor, and must generate a write fault every time they need to update the page.

Page synchronization method	Page ownership strategy			
	Static	Dynamic		
		Centralized manager	Distributed manager	
			Fixed	Dynamic
Invalidation	not appropriate	okay	good	good
Writeback	not appropriate	not appropriate	not appropriate	not appropriate

Table 1: Spectrum of solutions to the memory coherence problem.

As with the writeback approach, this also is an expensive solution for existing loosely-coupled multiprocessors, and furthermore is rather constraining to desired modes of parallel computation. Thus in this paper we only consider *dynamic* ownership strategies, as indicated in Table 1.

The strategies for maintaining dynamic page ownership can be subdivided into two classes: *centralized* and *distributed*. We refer to the process that controls page ownership as the *manager*, and thus we can have centralized or distributed managers. Distributed managers can be further classified as either *fixed* or *dynamic*, referring to the distribution of ownership data (to be described later).

The resulting combinations of strategies are shown in Table 1, where we have marked as inappropriate all combinations involving writeback synchronization or static page ownership. In this paper we only consider the remaining choices.

As mentioned earlier, the page size granularity allows us to use hardware page protection mechanisms to cause a fault when an invalid memory reference occurs, and thus resolve memory coherence problems in page-fault handlers. Therefore, our algorithms for solving the memory coherence problem are manifested as fault handlers, their servers (i.e., the processes that handle remote requests from faulting processors), and the page tables on which they operate. In the next few sections we investigate several such algorithms.

3 Centralized Manager Algorithms

3.1 A Monitor-like Centralized Manager Algorithm

Our centralized manager is similar to a *monitor* [8], consisting of a data structure and some procedures that provide mutually exclusive access to the data structure. The centralized manager resides on a single processor, and maintains a table called *info* which has one entry for each page, each entry having three fields:

1. The *owner* field contains the single processor that owns that page; namely, the most recent processor to have write access to it.
2. The *copy_set* field lists all processors that have copies of the page. This allows an invalidation operation to be performed without using broadcast.
3. The *lock* field is used for synchronizing requests to the page, as will be described shortly.

Each processor also has a page table called *ptable* which has two fields: *access* and *lock*. This table keeps information about the accessibility of pages on the local processor.

In this algorithm, a page does not have a fixed owner, but there is only one manager that knows who the owner is. The owner of a page sends a copy to processors requesting a read copy. As long as a read copy exists, the page is not writable without an *invalidation* operation, which causes invalidation messages to be sent to all processors containing read copies. Since this is a monitor-style algorithm, it is easy to see that the successful writer to a page always has the truth of the page. When a processor finishes a read or write request, a *confirmation* message is sent to the manager to indicate completion of the request.

Both *info* table and *ptable* have page-based locks. They are used to synchronize the local page faults (i.e., fault handler operations) and remote fault requests (i.e., server operations). When there is more than one process on a processor waiting for the same page, the locking mechanism prevents the processor from sending more than one request. Also, if a remote request for a page arrives and the processor is accessing the page table entry, the locking mechanism will queue the request until the entry is released.

The algorithm is characterized by fault handlers and their servers:

Read fault handler:

```
lock( ptable[ p ].lock );
IF I am manager THEN BEGIN
   lock( info[ p ].lock );
   info[ p ].copy_set := info[ p ].copy_set ⋃ {manager_node};
   receive page p from info[ p ].owner;
   unlock( info[ p ].lock );
   END;
ELSE BEGIN
   ask manager for read access to p;
   send confirmation to manager;
   END;
```

```
    ptable[ p ].access := read;
    unlock( ptable[ p ].lock );

  Read server:
    lock( ptable[ p ].lock );
    IF I am owner THEN BEGIN
        ptable[ p ].access := read;
        send copy of p;
        END;
    unlock( ptable[ p ].lock );
    IF I am manager THEN BEGIN
        lock( info[ p ].lock );
  info[ p ].copy_set := info[ p ].copy_set ⋃ {request_node};
        ask info[ p ].owner to send copy of p to request_node;
        receive confirmation from request_node;
        unlock( info[ p ].lock );
        END;

  Write fault handler:
    lock( ptable[ p ].lock );
    IF I am manager THEN BEGIN
        lock( info[ p ].lock );
        invalidate( p, info[ p ].copy_set );
        info[ p ].copy_set := {};
        unlock( info[ p ].lock );
        END;
    ELSE BEGIN
        ask manager for write access to p;
        send confirmation to manager;
        END;
    ptable[ p ].access := write;
    unlock( ptable[ p ].lock );

  Write server:
    lock( ptable[ p ].lock );
    IF I am owner THEN BEGIN
        send copy of p;
        ptable[ p ].access := nil;
        END;
    unlock( ptable[ p ].lock );
    IF I am manager THEN BEGIN
        lock( info[ p ].lock );
        invalidate( p, info[ p ].copy_set );
        info[ p ].copy_set := {};
        ask info[ p ].owner to send p to request_node;
        receive confirmation from request_node;
        unlock( info[ p ].lock );
        END;
```

The *confirmation* message indicates the completion of a request to the manager, so that the manager can give the page to someone else. Together with the locking mechanism in the data structure, the manager synchronizes the multiple requests from different processors.

Since the centralized manager plays the role of helping other processors locate where a page is, we can consider the number of messages for locating a page as one measure of its complexity:

Theorem 3.1 *The worst case number of messages to locate a page in the centralized manager algorithm is two.*

Although this algorithm uses only two messages in locating a page, it requires a confirmation message whenever a fault appears on a non-manager processor. Eliminating the *confirmation* operation is the motivation for the following improvement to this algorithm.

3.2 An Improved Centralized Manager Algorithm

The primary difference between the improved centralized manager algorithm and the previous one is that the synchronization of page ownership has been moved to the individual owners, thus eliminating the *confirmation* operation to the manager. The locking mechanism on each processor now deals not only with multiple local requests, but also with remote requests. The manager still answers the question of where a page owner is, but it no longer synchronizes requests.

To accommodate these changes, the data structure of the manager must change. Specifically, the manager no longer maintains the *copy_set* information, and a page-based lock is no longer needed. The information about the ownership of each page is still kept in a table called *owner*, but an entry in the *ptable* on each processor now has three fields: *access, lock*, and *copy_set*. The *copy_set* field in an entry is *valid* if and only if the processor that holds the page table is the owner of the page.

The fault handlers and servers for this algorithm are as follows:

```
  Read fault handler:
    lock( ptable[ p ].lock );
    IF I am manager THEN
        receive page p from owner[ p ];
    ELSE
        ask manager for read access to p;
    ptable[ p ].access := read;
    unlock( ptable[ p ].lock );

  Read server:
    lock( ptable[ p ].lock );
    IF I am owner THEN BEGIN
  ptable[ p ].copy_set := ptable[ p ].copy_set ⋃ {request_node} ;
        ptable[ p ].access := read;
        send p;
        END
    ELSE IF I am manager THEN BEGIN
        lock( manager_lock );
        forward request to owner[ p ];
        unlock( manager_lock );
        END;
    unlock( ptable[ p ].lock );

  Write fault handler:
    lock( ptable[ p ].lock );
    IF I am manager THEN
        receive page p from owner[ p ];
    ELSE
        ask manager for write access to p;
    invalidate( p, ptable[ p ].copy_set );
    ptable[ p ].access := write;
    ptable[ p ].copy_set := {};
    unlock( ptable[ p ].lock );
```

```
Write server:
  lock( ptable[ p ].lock );
  IF I am owner THEN BEGIN
     send p and ptable[ p ].copy_set;
     ptable[ p ].access := nil;
     END
  ELSE IF I am manager THEN BEGIN
     lock( manager_lock );
     forward request to owner[ p ];
     owner[ p ] := request_node;
     unlock( manager_lock );
     END;
  unlock( ptable[ p ].lock );
```

Although the synchronization responsibility of the original manager has moved to individual processors, the functionality of the synchronization remains the same. For example, consider a scenario in which two processors P_1 and P_2 are trying to write into the same page owned by a third processor P_3. If the request from P_1 arrives at the manager first, the request will be forwarded to P_3. Before the paging is complete, suppose the manager receives a request from P_2, then forwards it to P_1. Since P_1 has not received ownership of the page yet, the request from P_2 will be queued until P_1 finishes paging. Therefore, both P_1 and P_2 will receive access to the page in turn.

The overall performance of the shared virtual memory has been improved by decentralizing the synchronization, but for large N there still might be a bottleneck at the manager processor, since it must respond to every page fault.

4 Distributed Manager Algorithms

In the centralized manager algorithms described in the previous section, there is only one manager for the whole shared virtual memory. Clearly such a centralized manager can be a potential bottleneck. In this section we consider distributing the managerial task among the individual processors.

4.1 A Fixed Distributed Manager Algorithm

In a *fixed* distributed manager scheme, every processor is given a predetermined subset of the pages to manage. The primary difficulty in such a scheme is choosing an appropriate mapping from pages to processors. The most straightforward approach is to distribute pages evenly in a fixed manner to all processors. For example, suppose there are M pages in the shared virtual memory, and that $I = \{1, \ldots, M\}$ An appropriate mapping function H could then be defined by:

$$H(p) = p \bmod N \qquad (1)$$

where $p \in I$ and N is the number of processors. A more general definition is:

$$H(p) = (\frac{p}{s}) \bmod N \qquad (2)$$

where s is the number of pages per *segment*. Thus defined, this function distributes manager work by segments. Another approach would be to use a suitable hashing function.[2]

With this approach there is one manager per processor, each responsible for the pages specified by the static mapping function H. When a fault occurs on page p, the faulting processor asks processor $H(p)$ where the true page owner is, and then proceeds as in the centralized manager algorithm.

Our experiments have shown that the fixed distributed manager algorithm is substantially superior to the centralized manager algorithms when a parallel program exhibits a high rate of page faults. However, it is difficult to find a good static distribution function that fits all applications well. Indeed, for any given function it is always possible to find a pathological case that produces performance no better than the centralized scheme. So we would like to investigate the possibility of distributing the work of managers *dynamically*.

4.2 A Broadcast Distributed Manager Algorithm

An obvious way of eliminating the centralized manager is by using a *broadcast* mechanism. With this strategy, each processor manages precisely those pages that it owns, and faulting processors send broadcasts into the network to find the true owner of a page. Thus the *owner* table is eliminated completely, and the information of ownership is stored in each processor's *ptable*, which in addition to *access, copy_set* and *lock* fields, also has an *owner* field.

More precisely, when a read fault occurs, the faulting processor P sends a *broadcast read request*, and the true owner of the page responds by adding P to the page's *copy_set* field and sending a copy of the page to P. Similarly, when a write fault occurs, the faulting processor sends a *broadcast write request*, and the true owner of the page gives up ownership and sends back the page and its *copy_set*. When the requesting processor receives the page and the *copy_set*, it will invalidate all copies.

Although the work on all processors is fairly balanced in this algorithm, when a processor broadcasts a message all other processors must respond to the request (if only by ignoring it). This makes the communications subsystem a potential bottleneck.

4.3 A Dynamic Distributed Manager Algorithm

The heart of a dynamic distributed manager algorithm is to attempt to keep track of the ownership of all pages in each processor's local *ptable*. To do this, the *owner* field is replaced with another field, *prob_owner*, whose value can

[2] It is also conceivable to provide a default mapping function that clients may override by supplying their own mapping. In this way, the map could be tailored to the data structure in the application and the expected behavior of concurrent memory references.

be either *nil* or the "probable" owner of the page. The information that it contains is not necessarily correct at all times, but if incorrect it will at least provide the beginning of a sequence of processors through which the true owner can be found. Initially, the *prob_owner* field of every entry on all processors is set to some default processor that can be considered as the initial owner of all pages. It is the job of the page fault handlers and their servers to maintain this field as the program runs.

In this algorithm a page does not have a fixed owner or manager. When a processor has a page fault, it sends a request to the processor indicated by the *prob_owner* field for that page. If that processor is the true owner, it will proceed as in the centralized manager algorithm. If it is not, it will forward the request to the processor indicated by its *prob_owner* field. As with the centralized algorithm, a read fault results in making a copy of the page, and a write fault results in making a copy, invalidating other copies, and changing the ownership of the page. The *prob_owner* field is updated whenever:

- a processor receives an invalidation request,
- a processor relinquishes ownership of the page, or
- a processor forwards a page fault request.

In the first two cases, the *prob_owner* field is changed to the new owner of the page. In the last case, the *prob_owner* is changed to the original requesting processor, which will become the true owner in the near future.

The algorithm is as follows:

```
Read fault handler:
  lock( ptable[ p ].lock );
  ask ptable[ p ].prob_owner for read access to p;
  ptable[ p ].prob_owner := reply_node;
  ptable[ p ].access := read;
  unlock( ptable[ p ].lock );

Read server:
  IF I am owner THEN BEGIN
     lock( ptable[ p ].lock );
  ptable[ p ].copy_set := ptable[ p ].copy_set ⋃ {request_node};
     ptable[ p ].access := read;
     send p and ptable[ p ].copy_set;
     ptable[ p ].copy_set := {};
     ptable[ p ].prob_owner := request_node;
     unlock( ptable[ p ].lock );
     END
  ELSE BEGIN
     forward request to ptable[ p ].prob_owner;
     ptable[ p ].prob_owner := request_node;
     END;

Write fault handler:
  lock( ptable[ p ].lock );
  ask ptable[ p ].prob_owner for write access to page p;
  invalidate( p, ptable[ p ].copy_set );
  ptable[ P ].prob_Owner := self;
  ptable[ p ].access := write;
  ptable[ p ].copy_set := {};
  unlock( ptable[ p ].lock );

Write server:
  IF I am owner THEN BEGIN
     lock( ptable[ p ].lock );
     ptable[ p ].access := nil;
     send p and ptable[ p ].copy_set;
     ptable[ p ].prob_owner := request_node;
     unlock( ptable[ p ].lock );
     END
  ELSE BEGIN
     forward request to ptable[ p ].prob_owner;
     ptable[ p ].prob_owner := requesting_node;
     END;

Invalidate server:
  ptable[ p ].access := nil;
  ptable[ p ].prob_owner := request_node;
```

The two critical questions about the *prob_owners* are whether forwarding requests eventually arrive at the true owner and how many forwarding requests are needed. In order to answer these questions it is convenient to view all the *prob_owners* of a page p as a directed graph $G_p = (V, E_p)$ where V is the set of processor numbers $1, \ldots, N$, $|E_p| = N$, and an edge $(i, j) \in E_p$ if and only if the *prob_owner* for page p on processor i is j. By induction on the number of page faults, we can prove the following lemma:

Lemma 4.1 *Except for a distinguished node that points to itself, every prob_owner graph is acyclic.*

The uniqueness of page ownership is expressed by:

Lemma 4.2 *There is exactly one node i such that $(i, i) \in E_p$.*

Proof: (Outline) Initially each page p only has one owner. The only possible place where an edge (i, i) can be generated is on line 4 in the write fault handler. In order to execute that line, the request on line 3 must have been completed. When replying to a request, the write server's probable owner is changed to the requesting processor. This is done using a lock. Finally, since the receiving queue automatically serializes the arriving messages, an owner cannot reply to more than one requesting node. □

Theorem 4.1 *A page fault on any processor eventually reaches the true owner of the page.*

Proof: (Outline) By lemmas 4.1 and 4.2, the *prob_owner* graph of a page is acyclic except for the edge from the owner i to itself. Furthermore, if processor j forwards a page fault request to processor k, then processor j has more recent knowledge about the ownership than processor k. Thus, for any node $j \in V$, there is a path to i. □

Theorem 4.1 guarantees the correctness of a *prob_owner* graph whenever no fault is in progress. Since the fault handlers and their servers use locking mechanisms to guarantee atomicity in their operations, it is easy to see the correctness of the algorithm.

The worst case number of forwarding messages is given by the following theorem:

Theorem 4.2 *If there are N processors in a shared virtual memory, then it will take at most $N-1$ messages to locate a page.*

Proof: By lemmas 4.1 and 4.2, the worst case occurs when the *prob_owner* graph is a linear chain:

$$E_p = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{N-1}, v_N), (v_N, v_N)\}$$

in which case a fault on processor v_1 will generate $N-1$ forwarding messages in finding the true owner v_N. □

Note that once this worse-case situation occurs, *all* processors know the true owner. Also note that if there is another fault on v_i at the same time, then the forwarding message from v_1 will be blocked due to the locking of the fault handler on v_i, soon after which v_i receives ownership. In this case it take only $i-1$ messages to locate the page.

At the other extreme, we can state the following best-case performance (which is better than any of the previous alorithms):

Theorem 4.3 *There exists a prob_owner graph and page fault sequence such that the total number of messages for locating N different owners of the same page is N.*

Proof: Such a situation exists when the a *prob_owner* graph is the same chain that caused the worst-case performance in Theorem 4.2. □

It is interesting that the worst-case single-fault situation is coincident with the best-case N-fault situation, since in parallel systems the performance when contention is high is very important. The immediate question that now arises is what is the *worst-case* performance for K faults to the same page. To answer this, note that the general problem is easily reduced to the set union-find problem. An upper bound on N unions and M finds for this problem has been shown to be $O(N + M \log N)$ for $M < N$ and $O(M \log_{1+M/N} N)$ for $M \geq N$. [11,17,5]. Since both read page faults and write page faults compress their traversing paths, it is easy to see that the abstraction of the algorithm can be reduced to the set union problem with find operations alone. The following theorem restates the upper bound with respect to our problem:

Theorem 4.4 *For an N-processor shared virtual memory, using the dynamic distributed manager algorithm, the worst-case number of messages for locating K owners of a single page is $O(N + K \log N)$ for $K < N$ and $O(K \log_{1+K/N} N)$ for $K \geq N$.*

Corollary 4.1 *Using the dynamic distributed manager algorithm, if p processors are using a page, an upper bound on the total number of messages for locating K owners of the page is $O(p + K \log p)$ for $K < p$ and $O(K \log_{1+K/p} p)$ for $K \geq p$, if all contending processors are in the p processor set.*

This is an important corollary, since it says that the algorithm does not degrade as more processors are added to the system, but rather degrades (logarithmically) only as more processors contend for the same page.

4.4 A Dynamic Distributed Manager With Fewer Broadcasts

In the previous algorithm, at initialization or after a broadcast, all processors know the true owner of a page. The following theorem gives an upper bound for this case:

Theorem 4.5 *After a broadcast request or a broadcast invalidation, an upper bound on the total number of messages for locating the owner of a page for K page faults on different processors is $2K-1$.*

Proof: This can be shown by the transition of a *prob_owner* graph after a broadcast. The first fault uses 1 message to locate a page and after that every fault uses 2 messages. □

This theorem suggests the possibility of further improving the algorithm by enforcing a broadcast message (announcing the true owner of a page) after every K page faults to a page. In this case, a counter is needed in each entry of the page table, and is maintained by its owner. (Interestingly, when $K = 0$ this algorithm is functionally equivalent to the broadcast distributed manager algorithm, and when $K = N-1$ it is equivalent to the unmodified dynamic distributed manager algorithm.) The algorithm is as follows:

```
Read fault handler:
  lock( ptable[ p ].lock );
  ask ptable[ p ].prob_owner for read access to p;
  ptable[ p ].prob_owner := reply_node;
  ptable[ p ].access := read;
  unlock( ptable[ p ].lock );

Read server:
  IF I am owner THEN BEGIN
     lock( ptable[ p ].lock );
ptable[ p ].copy_set := ptable[ p ].copy_set ⋃ {request_node};
     ptable[ p ].access := read;
     ptable[ p ].counter := ptable[ p ].counter + 1;
     send p and ptable[ p ].copy_set;
     ptable[ p ].copy_set := {};
     ptable[ p ].prob_owner := request_node;
     unlock( ptable[ p ].lock );
     END
  ELSE BEGIN
     forward request to ptable[ p ].prob_owner;
     ptable[ p ].prob_owner := request_node;
     END;

Write fault handler:
  lock( ptable[ p ].lock );
  ask ptable[ p ].prob_owner for write access to p;
  invalidate( p );
  ptable[ P ].prob_Owner := self;
  ptable[ p ].access := write;
  ptable[ p ].copy_set := {};
  unlock( ptable[ p ].lock );

Write server:
  IF I am owner THEN BEGIN
     lock( ptable[ p ].lock );
     ptable[ p ].access := nil;
```

```
    send p, ptable[ p ].copy_set, and ptable[ p ].counter;
    ptable[ p ].prob_owner := request_node;
    unlock( ptable[ p ].lock );
    END
  ELSE BEGIN
    forward request to ptable[ p ].prob_owner;
    ptable[ p ].prob_owner := request_node;
    END;

Invalidate( p ):
  IF ( ptable[ p ].counter > L )
       OR ( size( ptable[ p ].copy_set > L ) THEN
    broadcast invalidation;
  ELSE
    invalidate according to ptable[ p ].copy_set;

Invalidate server:
    ptable[ p ].access := nil;
    ptable[ p ].prob_owner := request_node;
```

Note the counter L used in the invalidation procedure; whether a broadcast invalidation message is sent depends on whether the number of copies of a page reaches L. The value L can be adjusted experimentally to improve system performance.

On the average, without considering the cost of the broadcast message, this algorithm takes a little less than 2 messages to locate a page after a broadcast request or broadcast invalidation.

4.5 A Refinement: Distribution of copy_sets

Note that in the previous algorithm, the *copy_set* of a page is used only for the invalidation operation induced by a write fault. The *location* of the set is unimportant as long as the algorithm can invalidate the read copies of a page correctly. Further note that the *copy_set* field of processor i contains j if processor j copied the page from processor i, and thus the *copy_set* fields for a page are subsets of the original *copy_set*.

These facts suggest a refinement to the previous algorithms in which the *copy_set* data associated with a page is stored as a *tree* of processors rooted at the owner. In fact, the tree is bidirectional, with the edges directed from the root formed by the *copy_set* fields, and the edges directed from the leaves formed by *prob_owner* fields. The tree is used during faults as follows: A *read* fault collapses the path up the tree through the *prob_owner* fields to the owner. A *write* fault invalidates all copies in the tree by inducing a wave of invalidation operations starting at the owner, propagating to the processors in its *copy_set*, which in turn send invalidation requests to the processors in their *copy_sets*, and so on.

The following algorithm is a modified version of the original dynamic distributed manager algorithm:

```
Read fault handler:
  lock( ptable[ p ].lock );
  ask ptable[ p ].prob_owner for read access to p;
  ptable[ p ].prob_owner := reply_node;
  ptable[ p ].access := read;
  unlock( ptable[ p ].lock );

Read server:
  IF ptable[ p ].access ≠ nil THEN BEGIN
    lock( ptable[ p ].lock );
ptable[ p ].copy_set := ptable[ p ].copy_set ⋃ {request_node};
    ptable[ p ].access := read;
    send p;
    unlock( ptable[ p ].lock );
    END
  ELSE BEGIN
    forward request to ptable[ p ].prob_owner;
    ptable[ p ].prob_owner := request_node;
    END;

Write fault handler:
  lock( ptable[ p ].lock );
  ask ptable[ p ].prob_owner for write access to p;
  invalidate( p, ptable[ p ].copy_set );
  ptable[ P ].prob_Owner := self;
  ptable[ p ].access := write;
  ptable[ p ].copy_set := {};
  unlock( ptable[ p ].lock );

Write server:
  IF I am owner THEN BEGIN
    lock( ptable[ p ].lock );
    ptable[ p ].access := nil;
    send p and ptable[ p ].copy_set;
    ptable[ p ].prob_owner := request_node;
    unlock( ptable[ p ].lock );
    END
  ELSE BEGIN
    forward request to ptable[ p ].prob_owner;
    ptable[ p ].prob_owner := request_node;
    END;

Invalidate server:
  IF ptable[ p ].access ≠ nil THEN BEGIN
    invalidate( p, ptable[ p ].copy_set );
    ptable[ p ].access := nil;
    ptable[ p ].prob_owner := request_node;
    ptable[ p ].copy_set := {};
    END;
```

By distributing *copy_sets* in this manner, we improve system performance in two important ways. First of all, the propagation of invalidation messages is usually faster because of its "divide and conquer" effect. If the *copy_set* tree is perfectly balanced, the invalidation process will take time proportional to $\log i$ for i read copies. This faster invalidation response shortens the time for a write fault.

Secondly, and perhaps more importantly, a read fault now only needs to find a single processor (not necessarily the owner) that holds a copy of the page. To make this work, recall that a lock at the owner of each page synchronizes concurrent write faults to the page. A similar lock is now needed on processors having read copies of the page, to synchronize sending copies of the page in the presence of other read or write faults. The details may be found in the algorithm.

Overall this refinement can be applied to any of the foregoing distributed manager algorithms, but it is particularly useful on a multiprocessor lacking a broadcast facility.

5 Experimental Results

We have implemented a prototype shared virtual memory by modifying the AEGIS operating system on a ring network of Apollo workstations [12,10]. The system can be used to run parallel programs on any number of processors. The improved centralized manager algorithm, the dynamic distributed manager algorithm, and the fixed distributed manager algorithm have been implemented for experimental purposes. In this section we present the results of running three parallel programs.

The first program implements a parallel Jacobi algorithm for solving three dimensional PDE's. More specifically, we solve the equation $Ax = b$ where A is a n^3 by n^3 sparse matrix (in our experiments $n = 50$ and $n = 40$). A number of processes are created to partition the problem by the number of rows of the matrix. Since A is sparse, it is not represented explicitly as a matrix, but rather implicitly as index/value pairs. The vectors x and b are stored in the shared virtual memory, and the processes access them freely without regard to their location. Such a program is much simpler than what results from the usual message-passing style, because the programmer does not have to perform data movements explicitly at each iteration.

The second program is parallel sorting; more specifically, a block odd-even based merge-split algorithm [2]. The data blocks are stored in a large array in the shared virtual memory, and the recursively spawned processes access it freely. Again because the data movement is implicit, the program is very straightforward.

The third program is parallel matrix multiplication, $C = AB$. All of the matrices are stored in the shared virtual memory. A number of processes are created to partition the problem by the number of columns of matrix B. Initially, matrices A and B are stored on one processor, and are paged to other processors "by demand" as the processes on those processors reference them.

Figures 2 and 3 show the number of forwarding requests for locating true pages during one iteration of the PDE program using the dynamic distributed manager and the improved centralized manager. The dynamic distributed manager obviously outperforms the centralized one. This is because the *prob_owner* fields usually give correct hints, and within a short period of time the number of processors sharing a page is small; whereas in the centralized manager case, every page fault on a non-manager processor needs a forwarding request to locate the owner of the page.

Figure 4 shows the speedup curve for the 3-D PDE program. Note that the program experiences better than linear speedup! This is because the data structure for the problem is greater than the size of physical memory on a single processor, so when the program is run on one processor there

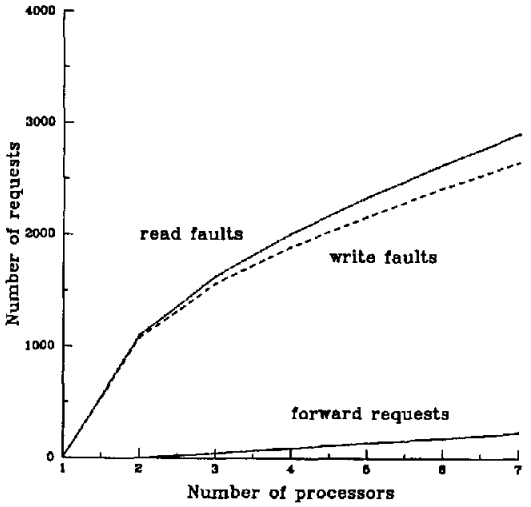

Figure 2: Dynamic distributed manager algorithm

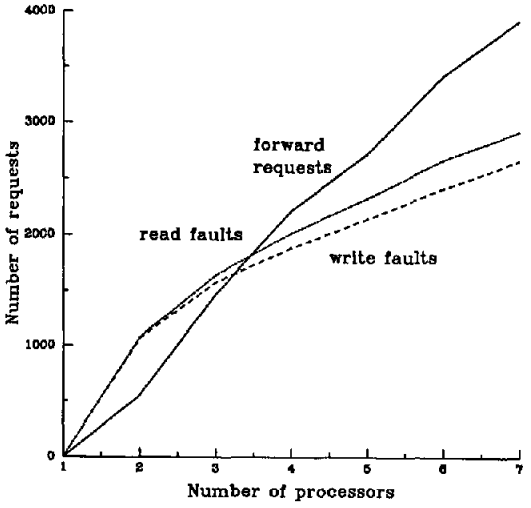

Figure 3: Centralized manager algorithm

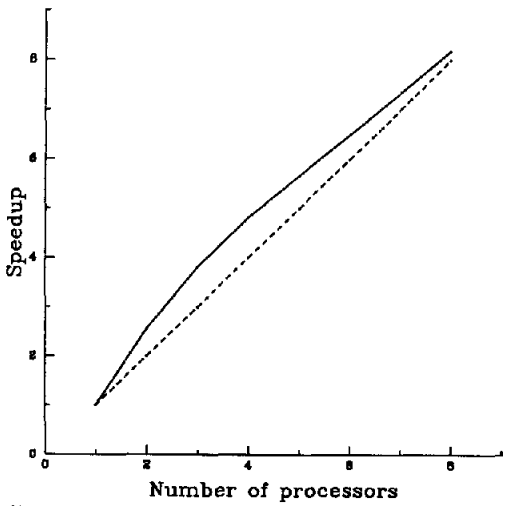

Figure 4: Speedups of a 3-D PDE where n = 50

is a large amount of paging between the physical memory and disk. The shared virtual memory, on the other hand, distributes the data structure into individual physical memories, whose cumulative size is large enough to inhibit disk paging. It is clear from this example alone that the shared virtual memory can indeed exploit the combined physical memories of a multiprocessor system.

Figure 5 shows another speedup curve for the 3-D PDE program, but now $n = 40$, in which case the data structure of the problem is not larger than the physical memory on a processor. The curve is very similar to that generated by similar experiments on CM*, an architecture that could be viewed as a hardware implementation of shared virtual memory [9]. Indeed, it is as good as the best curve in the published experiments on CM* for the same program, while the efforts and costs of the two approaches are not comparable at all.

Parallel sorting on a loosely-coupled multiprocessor is generally very difficult, and is included here so as not to paint too bright a picture. The speedup curve of the parallel merge-split sort of 200k elements shown in Figure 6 is not very good. In theory, even with no communication costs, this algorithm does not yield linear speedup. To make matters worse, our curve is obtained by trying to use the best strategy for any given number of processors. For example, there is no merge-split sorting at all when running the program on one processor, there are 4 blocks when running the program on two processors, etc.

Figure 7 shows the speedup curve of the matrix multiplication program for $C = AB$ where both A and B are 128 by 128 square matrices. The speedup curve is close to linear since the program exhibits a high degree of localized computation.

In general, we feel that our results indicate that a shared virtual memory is indeed practical, even on a very loosely-coupled architecture such as the Apollo ring. More details on both the algorithmic and experimental aspects of shared virtual memory may be found in [10].

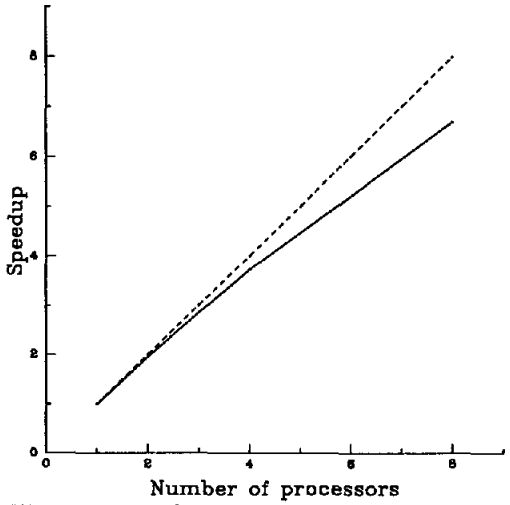

Figure 5: Speedups of a 3-D PDE where n = 40

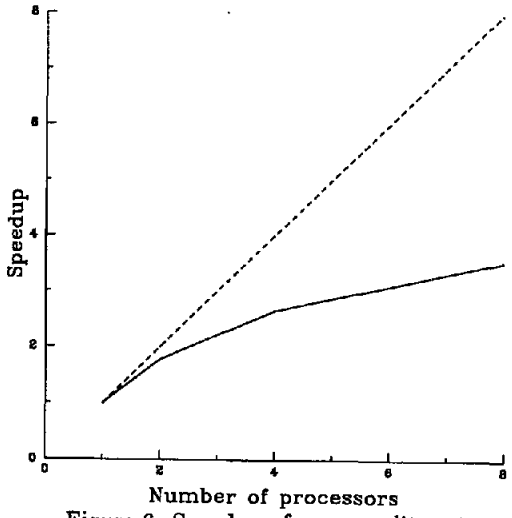

Figure 6: Speedup of merge-split sort

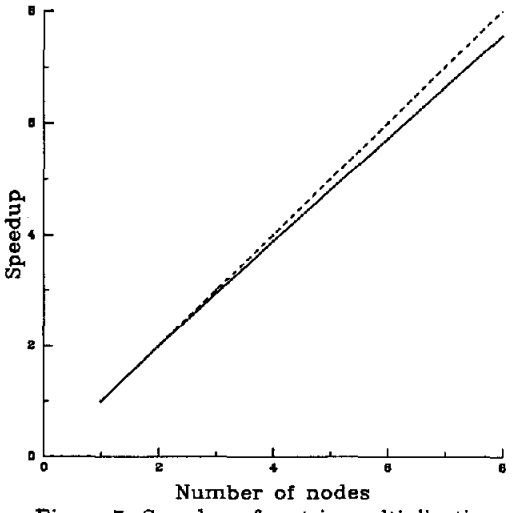

Figure 7: Speedup of matrix multiplication

6 Conclusions

We have discussed two classes of algorithms for solving the memory coherence problem—centralized manager and distributed manager—and both of them have many variations.

The centralized algorithm is straightforward and easy to implement, but may have a communications bottleneck at the central manager when there are many read and write page faults. The fixed distributed manager algorithm alleviates the bottleneck, and on average a processor needs about two messages to locate an owner.

The dynamic distributed manager algorithm and its variations seem to have the most desirable overall features. Theorem 4.5 states that by using fewer broadcasts, we can reduce the worst case number of messages for locating a page to a little less than two, which is the same as the worst cast for a centralized manager. A further refinement can be made by distributing *copy_sets*. Generally speaking,

dynamic distributed manager algorithms will outperform other methods when the number of processors sharing the same page for a short period of time is small, which is the normally the case. The good performance of the dynamic distributed manager algorithms in both theory and practice seems to make them feasible for implementation on a large-scale multiprocessor. In general, our experiments with an unoptimized prototype indicate that implementing a shared virtual memory is indeed useful and practical.

Acknowledgement

We wish to thank John Ellis for his invaluable suggestions and helpful discussions at the early stage of the work. Also, thanks to people at DECSRC, in particular, Andrew Birrel, Mark Brown, Butler Lampson, Roy Levin, Mike Schroeder, Larry Stewart, and Chuck Thacker for the helpful questions and suggestions in Summer 1984. Finally, we wish to thank Professor Alan Perlis for his continual help and inspiration.

References

[1] L.M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[2] D.K. Hsaio D. Bitton, D.J. DeWitt and J. Menon. A taxonomy of parallel sorting. *ACM Computing Surveys*, 16(3):287–318, September 1984.

[3] Peter J. Denning. On modeling program behavior. In *Proceedings of Spring Joint Computer Conference*, pages 937–944, AFIPS Press, 1972.

[4] Peter J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[5] Robert J. Fowler. *Decentralized Object Finding Using Forwarding Addresses.* PhD thesis, University of Washington, 1986.

[6] Steven J. Frank. Tightly coupled multiprocessor system speeds memory-access times. *Electronics*, :164–169, January 1984.

[7] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 124–131, June 1983.

[8] C.A.R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[9] A. K. Jones and P. Schwarz. Experience using multiprocessor systems — a status report. *ACM Computing Surveys*, 12(2), June 1980.

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

[11] Micheal S. Paterson. unpublished manuscript, 1973.

[12] P.J. Leach, P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson and B.L. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, 1983.

[13] R.H. Katz, S.J. Eggers, D.A. Wood, C.L. Perkins and R.G. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual Symposium on Computer Architecture*, pages 276–283, June 1985.

[14] Alan J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[15] Alfred Z. Spector. *Multiprocessing Architectures for Local Computer Networks.* Ph.D. thesis STAN-CS-81-874, Stanford University, August 1981.

[16] C.K. Tang. Cache system design in the tightly coupled multiprocessor system. In *Proceedings of AFIP National Computing Conference*, pages 749–753, 1976.

[17] Robert E. Tarjan and Jan Van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, April 1984.

[18] Chuck Thacker and et al. Private communications, 1984.

[19] D.W.L. Yen W.C. Yen and K. Fu. Data coherence problem in a multicache system. *IEEE Transactions on Computers*, C-34(1):56–65, January 1985.