# Memory Coherence in Shared Virtual Memory Systems

KAI LI
Princeton University
and
PAUL HUDAK
Yale University

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

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*network communications*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; D.4.2 [**Operating Systems**]: Storage Management—*distributed memories*; *virtual memory*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

General Terms: Algorithms, Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Loosely coupled multiprocessors, memory coherence, parallel programming, shared virtual memory

## 1. INTRODUCTION

The benefits of a virtual memory go without saying; almost every high performance sequential computer in existence today has one. In fact, it is hard to believe that loosely coupled multiprocessors would not also benefit from virtual memory. One can easily imagine how virtual memory would be incorporated into a *shared*-memory parallel machine because the memory hierarchy need not be much different from that of a sequential machine. On a multiprocessor in which the physical memory is *distributed*, however, the implementation is not obvious.

This research was supported in part by National Science Foundation grants MCS-8302018, DCR-8106181, and CCR-8814265. A preliminary version of this paper appeared in the *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing* [36].

Authors' addresses: K. Li, Department of Computer Science, Princeton University, Princeton, NJ 08544; P. Hudak, Department of Computer Science, Yale University, New Haven, CT 06520.

The *shared virtual memory* described in this paper provides a virtual address space that is shared among all processors in a loosely coupled distributed-memory multiprocessor system. Application programs can use the shared virtual memory just as they do a traditional virtual memory, except, of course, that processes can run on different processors in parallel. The shared virtual memory 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 swaps *processes*, so does the shared virtual memory. Thus the shared virtual memory provides a natural and efficient form of *process migration* between processors in a distributed system. This is quite a gain because process migration is usually very difficult to implement. In effect, process migration subsumes *remote procedure calls.*

The main difficulty in building a shared virtual memory is solving the *memory coherence problem.* This problem is similar to that which arises with *multicache* schemes for shared memory multiprocessors, but they are different in many ways. In this paper we concentrate on the memory coherence problem for a shared virtual memory. A number of algorithms are presented, analyzed, and compared. A prototype system called IVY has been implemented on a local area network of Apollo workstations. The experimental results of nontrivial parallel programs run on the prototype show the viability of a shared virtual memory. The success of this implementation suggests an operating mode for such architectures in which parallel programs can exploit the total processing power and memory capabilities in a far more unified way than the traditional "message-passing" approach.

## 2. SHARED VIRTUAL MEMORY

A *shared virtual memory* is a single address space shared by a number of processors (Figure 1). Any processor can access any memory location in the address space directly. *Memory mapping managers* implement the mapping between local memories and the shared virtual memory address space. Other than mapping, their chief responsibility is to keep the address space *coherent* at all times; that is, 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.

A shared virtual memory address space is partitioned into *pages.* Pages that are marked *read-only* can have copies residing in the physical memories of many processors at the same time. But a page marked *write* can reside in only one processor's physical memory. The memory mapping manager views its local memory as a large cache of the shared virtual memory address space for its associated processor. Like the traditional virtual memory [17], the shared memory itself exists only *virtually.* A memory reference causes a page fault when the page containing the memory location is not in a processor's current physical memory. When this happens, the memory mapping manager retrieves the page from either disk or the memory of another processor. If the page of the faulting memory reference has copies on other processors, then the memory mapping manager must do some work to keep the memory coherent and then continue the faulting instruction. This paper discusses both centralized manager algorithms and

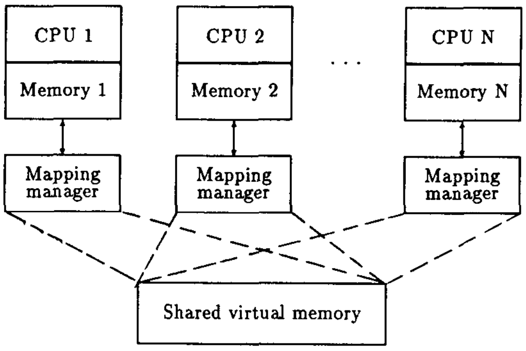

Fig. 1. Shared virtual memory mapping.

distributed manager algorithms, and in particular shows that a class of distributed manager algorithms can retrieve pages efficiently while keeping the memory coherent.

Our model of a parallel program is a set of *processes* (or threads) that share a single virtual address space. These processes are "lightweight"—they share the same address space, and thus the cost of a context switch, process creation, or process termination is small, say, on the order of a few procedure calls (Roy Levin, personal communication, 1986). One of the key goals of the shared virtual memory, of course, is to allow processes of a program to execute on different processors in parallel. To do so, the appropriate process manager and memory allocation manager must be integrated properly with the memory mapping manager. The process manager and the memory allocation manager are described elsewhere [34]. We refer to the whole system as a *shared virtual memory system.*

The performance of parallel programs on a shared virtual memory system depends primarily on two things: the number of parallel processes and the degree of updating of shared data (which creates contention on the communication channels). Since any processor can reference any page in the shared virtual memory address space and memory pages are moved and copied on demand, the shared virtual memory system does not exhibit pathological thrashing for unshared data or shared data that is read-only. Furthermore, updating shared data does not necessarily cause thrashing if a program exhibits *locality of reference.* One of the main justifications for traditional virtual memory is that memory references in sequential programs generally exhibit a high degree of locality [16, 17]. Although memory references in parallel programs may behave differently from those in sequential ones, a single process is still 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.

There is a large body of literature related to the research of shared virtual memory. The closest areas are virtual memory and parallel computing on loosely coupled multiprocessors.

Research on virtual memory management began in the 1960s [15] and has been an important topic in operating system design ever since. The research focused on the design of virtual memory systems for uniprocessors. A number of the early systems used memory mapping to provide access to different address spaces. The representative systems are Tenex and Multics [5, 13]. In these systems, processes in different address spaces can share data structures in mapped memory pages. But the memory mapping design was exclusively for uniprocessors.

Spector proposed a remote reference/remote operation model [42] in which a master process on a processor performs remote references and a slave process on another processor performs remote operations. Using processor names as part of the address in remote reference primitives, this model allows a loosely coupled multiprocessor to behave in a way similar to CM* [24, 29] or Butterfly [6] in which a shared memory is built from local physical memories in a static manner. Although implementing remote memory reference primitives in microcode can greatly improve efficiency, the cost of accessing a remote memory location is still several orders of magnitude more expensive than a local memory reference. The model is useful for data transfer in distributed computing, but it is unsuitable for parallel computing.

Among the distributed operating systems for loosely coupled multiprocessors, Apollo Aegis [2, 32, 33] and Accent [20, 38] have had a strong influence on the integration of virtual memory and interprocess communication. Both Aegis and Accent permit mapped access to data objects that can be located anywhere in a distributed system. Both of them view physical memory as a cache of virtual storage. Aegis uses mapped read and write memory as its fundamental communication paradigm. Accent has a similar facility called *copy-on-write* and a mechanism that allows processes to pass data *by value*. The data sharing between processes in these systems is limited at the object level; the system designs are for distributed computing rather than parallel computing.

Realistic parallel computing work on loosely coupled multiprocessors has been limited. Much work has focused on message passing [11, 19, 39]. It is possible to gain large speedups over a uniprocessor by message passing, but programming applications are difficult [11]. Furthermore, as mentioned above, message passing has difficulties in passing complicated data structures.

Another direction has been to use a set of primitives, available to the programmer in the source language, to access a global data space for storing shared data structures [8, 11]. The chief problem with such an approach is the user's need to control the global data space explicitly, which can become especially complex when passing large data structures or when attempting process migration. In a shared virtual memory such as we propose, no explicit data movement is required (it happens implicitly upon memory reference), and complex data is moved as easily as simple data. Another serious problem with the explicit global data space approach is that efficiency is impaired even for local data since use of a primitive implies at least the overhead of a procedure call. This problem becomes especially acute if one of the primitive operations occurs in an inner loop, in which case execution on one processor is much slower than that of the best sequential program, that is, one in which the operation is replaced with a standard memory reference. In contrast, when using our shared virtual memory, the inner loop would look just the same as its sequential version, and thus the overhead for accessing local data would be exactly the cost of a standard memory reference.

The point being that, once the pages holding a global data structure are paged in, the mechanism for accessing the data structure is precisely the same as on a uniprocessor.

The concept of a shared virtual memory for loosely coupled multiprocessors was first proposed in [36] and elaborated in the Ph.D. dissertation [34]. Details of the first implementation, IVY, on a network of workstations was reported in [34] and [35]. On the basis of this early work, a shared virtual memory system was later designed for the Lotus operating system kernel [21]. Most recently, the concept has been applied to a large-scale interconnection network based shared-memory multiprocessor [12] and a large-scale hypercube multiprocessor [37].

Other related work includes software caches and analysis of memory references. The VMP project at Stanford implements a software virtual addressed cache [10] to provide multicomputers with a coherent shared memory space. Their initial experience shows that a cache line size can be as large as 128 or 256 bytes without performance degradation. The cache consistency protocol is similar to the dynamic distributed manager algorithm for shared virtual memory in this paper and its preliminary version [34]. Finally, techniques for analyzing memory references of parallel programs [18, 45] may be applicable to analyzing the behaviors of parallel programs using a shared virtual memory system.

## 3. MEMORY COHERENCE PROBLEM

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. An architecture with one memory access path should have no coherence problem. A single access path, however, may not satisfy today's demand for high performance. The memory coherence problem was first encountered when caches appeared in uniprocessors (see [40] for a survey) and has become more complicated with the introduction of "multicaches" for shared memories on multiprocessors [9, 23, 25, 31, 43, 46 and Chuck Thacher, personal communication, 1984]. The memory coherence problem in a shared virtual memory system differs, however, from that in multicache systems. A multicache multiprocessor usually has a number of processors sharing a physical memory through their private caches. Since the size of a cache is relatively small and the bus connecting it to the shared memory is relatively fast, a sophisticated coherence protocol is usually implemented in the multicache hardware such that the time delay of conflicting writes to a memory location is small. On the other hand, a shared virtual memory on a loosely coupled multiprocessor has no physically shared memory, and the communication cost between processors is nontrivial. Thus conflicts are not likely to be solved with negligible delay, and they resemble much more a "page fault" in a traditional virtual memory system.

There are two design choices that greatly influence the implementation of a shared virtual memory: the granularity of the memory units (i.e., the "page size") and the strategy for maintaining coherence. These two design issues are studied in the next two subsections.

### 3.1 Granularity

In a typical loosely coupled multiprocessor, sending large packets of data (say one thousand bytes) is not much more expensive than sending small ones (say less than ten bytes) [41]. This similarity in cost is usually due to the software protocols and overhead of the virtual memory layer of the operating system. If these overheads are acceptable, relatively large memory units are possible in a shared virtual memory. On the other hand, the larger the memory unit, the greater the chance for contention. Detailed knowledge of a particular implementation might allow the clients' programmer to minimize contention by arranging concurrent memory accesses to locations in different memory units. Either clients or the shared virtual memory storage allocator may try to employ such strategies, but this may introduce inefficient use of memory. So, the possibility of contention indicates the need for relatively small memory units.

A suitable compromise in granularity is the typical *page* as used in conventional virtual memory implementations, which vary in size on today's computers from 256 bytes to 8K bytes. Our experience indicates that a page size of about 1K bytes is suitable with respect to contention, and as mentioned above should not impose undue communications overhead. We expect that smaller page sizes (perhaps as low as 256 bytes) work well also, but we are not as confident about larger page sizes, due to the contention problem. The right size is clearly application dependent, however, and we simply do not have the implementation experience to say what size is best for a sufficiently broad range of parallel programs. In any case, choosing a page size consistent with that used in conventional virtual memory implementations has the advantage of allowing one to use existing page fault schemes. In particular, one can use the protection mechanisms in a hardware Memory Management Unit (MMU) that allow single instructions to trigger page faults and to trap appropriate fault handlers. A program can set the access rights to the pages in such a way that memory accesses that could violate memory coherence cause a page fault; thus the memory coherence problem can be solved in a modular way in the page fault handlers and their servers.

## 3.2 Memory Coherence Strategies

It is helpful to first consider the spectrum of choices one has for solving the memory coherence problem. These choices can be classified by the way in which one deals with *page synchronization* and *page ownership*, as shown in Table I.

*Page Synchronization.* There are two basic approaches to page synchronization: *invalidation* and *write-broadcast*. In the *invalidation* approach, there is only one owner processor for each page. This processor has either write or read access to the page. If a processor $Q$ has a write fault to a page $p$, its fault handler then

—invalidates all copies of $p$,

—changes the access of $p$ to write,

—moves a copy of $p$ to $Q$ if $Q$ does not have one already, and

—returns to the faulting instruction.

After returning, processor $Q$ "owns" page $p$ and can proceed with the write operation and other read or write operations until the page ownership is relinquished to some other processor. Processor $Q$, of course, does not need to move the copy of the page if it owns the page for reading. If a processor $Q$ has a read

Table I. Spectrum of Solutions to the Memory Coherence Problem

| Page synchronization method | Page ownership strategy | | | |
|---|---|---|---|---|
| | | Dynamic | | |
| | | | Distributed manager | |
| | Fixed | Centralized manager | Fixed | Dynamic |
| Invalidation | Not allowed | Okay | Good | Good |
| Write-broadcast | Very expensive | Very expensive | Very expensive | Very expensive |

fault to a page $p$, the fault handler then

—changes the access of $p$ to read on the processor that has write access to $p$,
—moves a copy of $p$ to $Q$ and sets the access of $p$ to read, and
—returns to the faulting instruction.

After returning, processor $Q$ can proceed with the read operation and other read operations to this page in the same way that normal local memory does until $p$ is relinquished to someone else.

In the *write-broadcast* approach, a processor treats a read fault just as it does in the *invalidation* approach. However, if a processor has a write fault, the fault handler then

—writes to all copies of the page, and
—returns to the faulting instruction.

The main problems with this approach is that it requires special hardware support. *Every* write to a shared page needs to generate a fault on the writing processor and update all copies because the philosophy of a shared virtual memory requires that pages be shared freely. To prevent the processor from having the same page fault again when returning to the faulting instruction, the hardware must be able to skip the faulted write cycle. We do not know of any existing hardware with this functionality.

The theoretical analysis on "snoopy cache" coherence [30] suggests that combining the invalidation approach with the write-broadcast approach may be a better solution. However, whether this approach can apply to the shared virtual memory is an open problem because the overhead of a write fault is much more than a write on a snoopy cache bus. Since the algorithms using write-broadcast do not seem practical for loosely coupled multiprocessors, they are not considered further in this paper.

*Page Ownership.* The ownership of a page can be *fixed* or *dynamic.* In the fixed ownership approach, a page is always owned by the same processor. 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. As with the write-broadcast approach, fixed page ownership is an expensive solution for existing loosely coupled multiprocessors. Furthermore, it constrains desired modes of parallel computation. Thus we only consider dynamic ownership strategies, as indicated in Table I.

The strategies for maintaining dynamic page ownership can be subdivided into two classes: *centralized* and *distributed.* Distributed managers can be further classified as either *fixed* or *dynamic,* referring to the distribution of ownership data.

The resulting combinations of strategies are shown in Table I, where we have marked as "very expensive" or "not allowed" all combinations involving write-broadcast synchronization or fixed page ownership. This paper only considers the remaining choices: algorithms based on invalidation using either a centralized manager, a fixed distributed manager, or a dynamic distributed manager.

## 3.3 Page Table, Locking, and Invalidation

All of the algorithms for solving the memory coherence problem in this paper are described by using page fault handlers, their servers, and the data structure on which they operate. The data structures in different algorithms may be different, but they have at least the following information about each page:

—*access*: indicates the accessibility to the page,

—*copy set*: contains the processor numbers that have read copies of the page, and

—*lock*: synchronizes multiple page faults by different processes on the same processor and synchronizes remote page requests.

Following uniprocessor virtual memory convention, this data structure is called a *page table.* Every processor usually has a page table on it, but the same page entry in different page tables may be different.

There are two primitives operating on the lock field in the page table:

```
lock(PTable[p].lock):
  LOOP
    IF test-and-set the lock bit THEN EXIT;
    IF fail THEN queue this process;

unlock(PTable[p].lock):
  clear the lock bit;
  IF a process is waiting on the lock THEN
    resume the process;
```

These two primitives are used to synchronize multiple page fault requests on the same processor or different processors.

Another primitive that we use in memory coherence algorithms is *invalidate.* There are at least three ways to invalidate the copies of a page: *individual, broadcast,* and *multicast.* The individual invalidation is just a simple loop:

```
Invalidate:
  Invalidate(p, copy_set)
    FOR i in copy_set DO
      send an invalidation request to processor i;
```

Broadcast or multicast invalidation does not need a copy set; each just requires a simple broadcast message.

The server of the invalidation operation is simple:

```
Invalidate server:
  PTable[p].access := nil;
```

Although there are many ways to implement remote operations, it is reasonable to assume that any remote operation requires two messages, a request and a reply, and that a reliable communication protocol is used so that once a processor sends a request (no matter whether it is a point-to-point message, broadcast, or multicast), it eventually receives a reply. With such an assumption, for $m$ copies on an $N$ processor system, an individual invalidation requires $2m$ messages, $m$ for requests, and $m$ for replies. A broadcast invalidation sends $m + 1$ messages and receives $N + m - 1$ messages of which $N - 1$ messages are received in parallel. A multicast invalidation needs to send $m + 1$ messages and receive $2m$ messages of which $m$ messages are received in parallel.

The cost of receiving $k$ messages in parallel is greater than or equal to that of receiving one message and less than or equal to that of receiving $k$ messages sequentially. If all $k$ processors are idle, receipt of these messages in parallel costs *nothing* since the idle time would otherwise be wasted. On the other hand, if all $k$ processors are busy, receipt of the messages would cost more since all $k$ processors would need to be interrupted in order to process the messages. Clearly, multicast invalidation has the best performance, although most loosely coupled systems do not have a multicast facility that can use the page table information. Broadcast invalidation is expensive when $N$ is large.

A copy set can be represented by a bit vector [1] when $N$ is small (e.g., less than 64). When $N$ is large, we may need to compact the copy set field. Three simple compaction methods are considered:

—*linked bit vector*: represents a copy set as a linked list that only links meaningful bit-vectors together to save space.

—*neighbor bit vector*: uses a bit vector as its copy set for neighbor processors (directly connected processors). This method requires processors to propagate invalidation requests.

—*vaguely defined set*: uses a tag to indicate whether there is a valid copy set. This allows the shared virtual memory to dynamically allocate memory for copy sets.

More detailed discussion on page table compaction can be found in [34].

## 4. CENTRALIZED MANAGER ALGORITHMS

### 4.1 A Monitor-Like Centralized Manager Algorithm

Our centralized manager is similar to a *monitor* [7, 27] consisting of a data structure and some procedures that provide mutually exclusive access to the data structure. The coherence problem in multicache systems has a similar solution [9]. 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.

**Algorithm 1** *MonitorCentralManager*

```
Read fault handler:
    Lock( PTable[ p ].lock );
    IF I am manager THEN BEGIN
        Lock( Info[ p ].lock );
        Info[ p ].copyset
            := Info[ p ].copyset ⋃ {ManagerNode};
        receive page p from Info[ p ].owner;
        Unlock( Info[ p ].lock );
        END;
    ELSE BEGIN
        ask manager for read access to p and a copy of p;
        receive 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 ].copyset
            := Info[ p ].copyset ⋃ {RequestNode};
        ask Info[ p ].owner to send copy of p to RequestNode;
        receive confirmation from RequestNode;
        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 ].copyset );
        Info[ p ].copyset := {};
        Unlock( Info[ p ].lock );
        END;
    ELSE BEGIN
        ask manager for write access to p;
        receive p;
        send confirmation to manager;
        END;
```

Figure 2

```
        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 ].copyset );
            Info[ p ].copyset := {};
            ask Info[ p ].owner to send p to RequestNode;
            receive confirmation from RequestNode;
            Unlock( Info[ p ].lock );
            END;
```

Figure 2. *(continued)*

(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 we describe shortly.

Each processor also has a page table called PTable that 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, and 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 writeable 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 ownership 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 queues the request until the entry is released.

As for all manager algorithms in this paper, the centralized manager algorithm in Figure 2 is characterized by the fault handlers and their servers.

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.

A read-page fault on the manager processor needs two messages, one to the owner of the page, another from the owner. A read-page fault on a nonmanager processor needs four messages, one to the manager, one to the owner, one from the owner, and one for confirmation. A write-page fault costs the same as a read-page fault except that it includes the cost of an invalidation.

Since the centralized manager plays the role of helping other processors locate where a page is, a traffic bottleneck at the manager may occur as $N$ becomes large and there are many page faults. The number of messages for locating a page is a measure of the complexity of the algorithm. When a nonmanager processor has a page fault, it sends a message to the manager and gets a reply message from the manager, so the algorithm has the following property:

PROPOSITION 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 nonmanager processor. Eliminating the *confirmation* operation is the motivation for the following improvement to this algorithm.

## 4.2 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 in a table called Owner kept on the manager, but an entry in the PTable on each processor now has three fields: *access*, *lock*, and *copy set*. The copy set field 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 can be found in Appendix A.

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 is 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 $P_2$ is queued until $P_1$ finishes paging. Therefore, both $P_1$ and $P_2$ receive access to the page in turn.

Compared with the cost of a read-page fault in the monitor-like algorithm, this algorithm saves one send and one receive per page fault on all processors, an obvious improvement. Decentralizing the synchronization improves the overall performance of the shared virtual memory, but for large $N$ there still might be a bottleneck at the manager processor because it must respond to every page fault.

## 5. 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. This section discusses several ways of distributing the managerial task among the individual processors.

### 5.1 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. The distributed directory map solution to the multicache coherence problem [3] is similar. For example, suppose there are $M$ pages in the shared virtual memory and that $I = \{1, \ldots, M\}$. An appropriate hashing 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

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

where $s$ is the number of pages per *segment.* Thus defined, this function distributes manager work by segments.

Other variations include using a suitable hashing function or providing a default mapping function that clients may override by supplying their own mapping. In the latter case, the map could be tailored to a particular application and its expected behavior with respect to concurrent memory references.

With this approach there is one manager per processor, each responsible for the pages specified by the fixed 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 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 fixed distribution function that fits all applications well. Thus we would like to investigate the possibility of distributing the work of managers *dynamically.*

### 5.2 A Broadcast Distributed Manager Algorithm

An obvious way to eliminate the centralized manager is to use 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, 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 invalidates all copies. All broadcast operations are atomic. The fault handlers and servers for such a naive algorithm are given in Appendix B.

The simplicity of this approach is appealing. Yet, the correctness of the algorithm is not obvious at first. Consider the case in which two write faults to the same page happen simultaneously on two processors $P_1$ and $P_2$, and consider the instant when the owner of the page, $P_3$, receives a broadcast request from $P_1$ and gives up its ownership but $P_1$ has not yet received the message granting ownership. At this point, $P_2$ sends its broadcast request, but there is no owner. However, this is not a problem because $P_2$'s message is queued on $P_1$ waiting for the lock on the page table entry; after $P_1$ receives ownership, the lock is released, and $P_2$'s message is then processed by $P_1$ (recall the definition of the *lock* and *unlock* primitives given in Section 3.3).

A read-page fault causes a broadcast request that is received by $N - 1$ processors but is replied to by only one of them. The cost for a write-page fault is the same, plus the overhead of an invalidation. Although the algorithm is simple, for a large $N$, performance is poor because all processors have to process each broadcast request, slowing down the computation on all processors. Our experiments show that the cost introduced by the broadcast requests is substantial when $N \geq 4$. A parallel program with many read- and write-page faults does not perform well on a shared virtual memory system based on a broadcast distributed manager algorithm.

## 5.3 A Dynamic Distributed Manager Algorithm

The heart of a dynamic distributed manager algorithm is keeping track of the ownership of all pages in each processor's local PTable. To do this, the *owner* field is replaced with another field, *probOwner*, whose value can be either the true owner or the "probable" owner of the page. The information that it contains is just a hint; it is not necessarily correct at all times, but if incorrect it at least provides the beginning of a sequence of processors through which the true owner can be found. Initially, the *probOwner* field of every entry on all processors is set to some default processor that can be considered 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 *probOwner* field for that page. If that processor is the true owner, it proceeds as in the centralized manager algorithm. If it is not, it forwards the request to the processor indicated by its *probOwner* field. When a processor forwards a request, it need not send a reply to the requesting processor.

The *probOwner* field changes on a write-page fault as well as a read-page fault. As with the centralized manager algorithms, 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 *probOwner* field is updated whenever a processor receives an invalidation request, a processor relinquishes ownership of the page, which can happen on a read- or write-page fault, or a processor forwards a page-fault request.

In the first two cases, the *probOwner* field is changed to the new owner of the page. Changing ownership of the page on a read-page fault makes analysis simpler. Later, we modify the algorithm so that a read-page fault no longer changes ownership. In the last case, the *probOwner* is changed to the original requesting processor, which becomes the true owner in the near future. The algorithm is shown in Figure 3.

The two critical questions about this algorithm 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 *probOwners* 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 *probOwner* for page $p$ on processor $i$ is $j$. Using this approach, we can show

THEOREM 1. *Using Algorithm 2, a page fault on any processor reaches the true owner of the page using at most $N - 1$ forwarding request messages.*

PROOF. See Appendix C. □

THEOREM 2. *Using Algorithm 2, the worst-case number of messages for locating the owner of a single page $K$ times is $O(N + K \log N)$.*

PROOF. See Appendix C. □

COROLLARY 1. *Using the dynamic distributed manager algorithm, if $q$ processors have used a page, an upper bound on the total number of messages for locating the owner of the page $K$ times is $O(p + K \log q)$ if all contending processors are in the $q$ 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.* Our experiments show that usually few processors are using the same page at the same time. Normally, $p = 2$. Furthermore, an invalidation operation can collapse all the read-copy paths in the graph. These facts suggest that the dynamic distributed manager algorithm has the potential to implement a shared virtual memory system on a large-scale multiprocessor system.

Note that in Algorithm 2, the read-fault handler and its server change the *probOwner* graph in the same way as the write-fault handler and its server, except that the write-fault handler does invalidation according to the copy set field. This was done primarily to make the cost analysis easier; for a real implementation, the read-fault handler and its server should be slightly modified to get

**Algorithm 2** *DynamicDistributedManager*

```
Read fault handler:
    Lock( PTable[ p ].lock );
    ask PTable[ p ].probOwner for read access to p;
    PTable[ p ].probOwner := self;
    PTable[ p ].access := read;
    Unlock( PTable[ p ].lock );

Read server:
    Lock( PTable[ p ].lock );
    IF I am owner THEN BEGIN
        PTable[ p ].copyset
            := PTable[ p ].copyset ⋃ {Self};
        PTable[ p ].access := read;
        send p and PTable[ p ].copyset;
        PTable[ p ].probOwner := RequestNode;
        END
    ELSE BEGIN
        forward request to PTable[ p ].probOwner;
        PTable[ p ].probOwner := RequestNode;
        END;

    Unlock( PTable[ p ].lock );

Write fault handler:
    Lock( PTable[ p ].lock );
    ask PTable[ p ].probOwner for write access to page p;
    Invalidate( p, PTable[ p ].copyset );
    PTable[ p ].probOwner := self;
    PTable[ p ].access := write;
    PTable[ p ].copyset := {};
    Unlock( PTable[ p ].lock );

Write server:
    Lock( PTable[ p ].lock );
    IF I am owner THEN BEGIN
        PTable[ p ].access := nil;
        send p and PTable[ p ].copyset;
        PTable[ p ].probOwner := RequestNode;
        END
    ELSE BEGIN
        forward request to PTable[ p ].probOwner;
        PTable[ p ].probOwner := RequestNode;
        END;
    Unlock( PTable[ p ].lock );

Invalidate server:
    PTable[ p ].access := nil;
    PTable[ p ].probOwner := RequestNode;
```

Figure 3

better performance:

```
Read-fault handler:
  Lock(PTable[p].lock);
  ask PTable[p].probOwner for read access to p;
  PTable[p].probOwner := ReplyNode;
  PTable[p].access := read;
  Unlock(PTable[p].lock);
Read server:
  Lock(PTable[p].lock);
  IF I am owner THEN BEGIN
    PTable[p].copyset
      := PTable[p].copyset ∪ {RequestNode};
    PTable[p].access := read;
    send p to RequestNode;
    END
  ELSE BEGIN
    forward request to PTable[p].probOwner;
    PTable[p].probOwner := RequestNode;
    END;
  Unlock(PTable[p].lock);
```

The modified read-fault handler and its server still compress the access path from the faulting processor to the owner, but they do not change the ownership of a page; rather, they change its *probOwner* field. The modified algorithm reduces one message for each read-page fault, an obvious improvement. For the modified algorithm, the worst case number of messages for locating $K$ owners of a single page is difficult to define cleanly because the read-fault handler and its server behave as a pure forwarding address scheme, which can be reduced to the set union find problem of a compressing path with naive linking [22], while the write-fault handler and its server behave differently.

The algorithm proposed in this section needs a broadcast or multicast facility only for the invalidation operation. If the invalidation is done by sending individual messages to the copy holders, there is no need to use the broadcast facility at all, and the benefits of the general approach can still be gained.

## 5.4 An Improvement by Using 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 the performance for $K$ page faults on different processors in this case:

THEOREM 3. *After a broadcast request or a broadcast invalidation, 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 *probOwner* graph after a broadcast. The first fault uses 1 message to locate a page and every fault after that 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 $M$ 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. The necessary changes to

Table II. Longest Path First Finds

| Number of nodes | Average number of messages/find | | | |
|---|---|---|---|---|
| $n$ | $M = N/4$ | $M = N/2$ | $M = 3N/4$ | $M = N$ |
| 4 | 1.00 | 1.50 | 1.30 | 1.75 |
| 8 | 1.50 | 1.75 | 2.00 | 2.13 |
| 16 | 1.75 | 2.13 | 2.42 | 2.63 |
| 32 | 2.13 | 2.63 | 2.83 | 3.00 |
| 64 | 2.63 | 3.00 | 3.25 | 3.45 |
| 128 | 3.00 | 3.45 | 3.71 | 3.88 |
| 256 | 3.45 | 3.88 | 4.14 | 4.35 |
| 512 | 3.88 | 4.35 | 4.62 | 4.80 |
| 1,024 | 4.35 | 4.80 | 5.05 | 5.26 |

Algorithm 2 are fairly straightforward, and we leave the details to the reader. It is interesting to note that when $M = 0$, this algorithm is functionally equivalent to the broadcast distributed manager algorithm, and when $M = N - 1$, it is equivalent to the unmodified dynamic distributed manager algorithm.

In order to choose the value of $M$, it is necessary to consider the general case in which the same processor may have any number of page faults because Theorem 3 only shows the performance for page faults on *distinct* processors. In the last section we showed that the worst-case number of messages for locating $K$ owners for a single page is $O(N + K \log N)$, but our intuition says that the performance of $K$ page faults right after a broadcast message should be better because in the starting *probOwner* graph, all the processors know the true owner.

Since it is difficult to find a function describing the relationship between $M$ and $N$ for the general case, two simulation programs were run for two different situations: approximated worst-case and approximated random-case behavior. The initial *probOwner* graph used in both programs is the graph after a broadcast in which all the processors know the true owner. The programs record the number of messages used for each find (locating an owner) operation.

The first program approximates the worst case by choosing at each iteration a node with the longest path to the owner. Table II shows the average number of messages for each find operation for $M = N/4$, $M = N/2$, $M = 3N/4$, and $M = N$. The table shows that the average number of messages steadily increases as $N$ gets large. Although picking the node with the longest path does not always generate the worst-case *probOwner* graph, our experiments show that the program actually converges when $M$ is very large. For example, the average number of messages becomes stable when $N = 64$ and $M > 1024$ from our experiments. Whether the case in which the average number of messages becomes stable is the worst case is an open problem.

The second program approximates random behavior by choosing a node randomly at each iteration. The average number of messages for each find operation is shown in Table III. The table was produced by running the program four times and computing the average values among all the executions. In

Table III. Random Finds

| Number of nodes | Average number of messages/find | | | |
|---|---|---|---|---|
| $N$ | $M = N/4$ | $M = N/2$ | $M = 3N/4$ | $M = N$ |
| 4 | 1.00 | 1.50 | 1.67 | 1.75 |
| 8 | 1.50 | 1.75 | 1.99 | 2.08 |
| 16 | 1.75 | 1.96 | 2.22 | 2.53 |
| 32 | 1.93 | 2.39 | 2.79 | 2.90 |
| 64 | 2.09 | 2.78 | 2.90 | 3.12 |
| 128 | 2.06 | 2.68 | 2.80 | 3.16 |
| 256 | 2.20 | 2.77 | 3.18 | 3.39 |
| 512 | 2.46 | 3.09 | 3.32 | 3.56 |
| 1,024 | 2.34 | 3.08 | 3.34 | 3.64 |

comparing Table II with Table III, note that, on average, random finds use fewer numbers of messages.

To choose an appropriate value of $M$, the two tables should be used together with the information about the expected number of contending processors because the performance of the dynamic distributed manager algorithm is only related to such a number rather than the number of processors in the system (Corollary 1).

## 5.5 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 real *copy set.*

These facts suggest an alternative 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 *probOwner* fields. The tree is used during faults as follows: A *read* fault collapses the path up the tree through the *probOwner* fields to the owner. A *write* fault invalidates all copies in the tree by inducing a wave of invalidation operations starting at the owner and 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 algorithm in Figure 4 is a modified version of the original dynamic distributed manager algorithm.

Since a write-page fault needs to find the owner of the page, the lock at the owner synchronizes concurrent write-page fault requests to the page. If read faults on some processors occur concurrently, the locks on the processors from which those faulting processors are copying synchronize the possible conflicts of

**Algorithm 3** *DynamicDistributedCopySet*

```
    Read fault handler:
        Lock( PTable[ p ].lock );
        ask PTable[ p ].probOwner for read access to p;
        PTable[ p ].probOwner := ReplyNode;
        PTable[ p ].access := read;
        Unlock( PTable[ p ].lock );

    Read server:
        Lock( PTable[ p ].lock );
        IF PTable[ p ].access ≠ nil THEN BEGIN
            PTable[ p ].copyset
                := PTable[ p ].copyset ∪ {RequestNode};
            PTable[ p ].access := read;
            send p;
            END
        ELSE BEGIN
            forward request to PTable[ p ].probOwner;
            PTable[ p ].probOwner := RequestNode;
            END;
        Unlock( PTable[ p ].lock );

    Write fault handler:
        Lock( PTable[ p ].lock );
        ask PTable[ p ].probOwner for write access to p;
        Invalidate( p, PTable[ p ].copyset );
        PTable[ P ].probOwner := self;


        PTable[ p ].access := write;
        PTable[ p ].copyset := {};
        Unlock( PTable[ p ].lock );

    Write server:
        Lock( PTable[ p ].lock );
        IF I am owner THEN BEGIN
            PTable[ p ].access := nil;
            send p and PTable[ p ].copyset;
            PTable[ p ].probOwner := RequestNode;
            END
        ELSE BEGIN
            forward request to PTable[ p ].probOwner;
            PTable[ p ].probOwner := RequestNode;
            END;
        Unlock( PTable[ p ].lock );
```

Figure 4

```
Invalidate server:
    IF PTable[ p ].access ≠ nil THEN BEGIN
        Invalidate( p, PTable[ p ].copyset );
        PTable[ p ].access := nil;
        PTable[ p ].probOwner := RequestNode;
        PTable[ p ].copyset := {};
        END;
```

Figure 4. *(continued)*

the write-fault requests and read-fault requests. In this sense, the algorithm is equivalent to the original one.

Distributing *copy sets* in this manner improves system performance for the architectures that do not have a broadcast facility in two important ways. First, 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 takes time proportional to log $m$ for $m$ read copies. This faster invalidation response shortens the time for a write fault.

Second, and perhaps more important, 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.

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.

## 6. EXPERIMENTS

Since parallel programs are complex and the interactions between parallel processes are often unpredictable, the only convincing way to justify a shared virtual memory on loosely coupled multiprocessors is to implement a prototype system and run some realistic experiments on it. We have implemented a prototype shared virtual memory system called IVY (Integrated shared Virtual memory at Yale). It is implemented on top of a modified Aegis operating system of the Apollo DOMAIN computer system [2, 33]. IVY can be used to run parallel programs on any number of processors on an Apollo ring network.

We have tested a number of memory coherence algorithms, including the improved centralized manager, the dynamic distributed manager, and the fixed distributed manager. To exercise the system we selected a set of benchmark programs that represent a spectrum of likely practical parallel programs that have a reasonably fine granularity of parallelism and side effects to shared data structures. Our goal in using these criteria was to avoid weighing the experiments in favor of the shared virtual memory system by picking problems that suit the system well.

In this paper we present the results of running four parallel programs. All of them are written in Pascal and transformed manually from sequential algorithms into parallel ones in a straightforward way. In order to measure performance, each processor at run time records statistical information into a file, including the number of read page faults, the number of write page faults, process migration data, and memory page distribution data. Information about disk paging and network traffic is obtained from the Aegis operating system.

## 6.1 Four Parallel Programs

The first is a parallel Jacobi program for solving three dimensional partial differential equations (PDEs). It solves a linear equation $Ax = b$ where $A$ is an $n$-by-$n$ sparse matrix. In each iteration, $x^{(k+1)}$ is obtained by

$$x_i^{(k+1)} = \left( \frac{b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)}}{a_{ii}} \right).$$

The parallel algorithm creates a number of processes to partition the problem by the number of rows of matrix $A$. All the processes are synchronized at each iteration by using an event count. Since matrix $A$ is a sparse matrix and it is never updated, we adopt the standard technique in scientific computing of encoding its contents in-line (William Gropp, personal communication, 1985). The vectors $x$ and $b$ are stored in the shared virtual memory, and thus the processes access them freely without regard to their location. Such a program is much simpler than that which results from the usual message-passing paradigm because the programmer does not have to perform data movement explicitly at each iteration. The program is written in Pascal and transformed manually from the sequential algorithm into a parallel one in a straightforward way.

The second program is parallel sorting; more specifically, a variation of the block odd-even based merge-split algorithm described in [4]. The sorted data is a vector of records that contain random strings. At the beginning, the program divides the vector into $2N$ blocks for $N$ processors, and creates $N$ processes, one for each processor. Each process sorts two blocks by using a quicksort algorithm [26]. This internal sorting is naturally done in parallel. Each process then does an odd-even block merge-split sort $2N - 1$ times. The vector is stored in the shared virtual memory, and the spawned processes access it freely. Because the data movement is implicit, the parallel transformation is straightforward.

The third program is a parallel matrix multiply that computes $C = AB$ where $A$, $B$, and $C$ are square matrices. A number of processes are created to partition the problem by the number of rows of matrix $C$. All the matrices are stored in the shared virtual memory. The program assumes that matrix $A$ and $B$ are on one processor at the beginning and that they are paged to other processors on demand.

The last program is a parallel dot-product program that computes

$$S = \sum_{i=1}^{n} x_i y_i.$$

A number of processes are created to partition the problem. Process $i$ computes the sum

$$S_i = \sum_{j=l_i}^{u_i} x_j y_j.$$

S is obtained by summing up the sums produced by the individual processes

$$S = \sum_{i=1}^{m} S_i,$$

where $m$ is the number of processes. Both vectors $x$ and $y$ are stored in the shared virtual memory in a random manner, under the assumption that $x$ and $y$ are not fully distributed before doing the computation. The main reason for choosing this example is to show the weak side of the shared virtual memory system; dot-product does little computation but requires a lot of data movement.

## 6.2 Speedups

The speedup of a program is the ratio of the execution time of the program on a single processor to that on the shared virtual memory system. The execution time of a program is the elapsed time from program start to program end, which is measured by the clock in the system. The execution time does not include the time of initializing data structures in the program because the initialization has little to do with the algorithm itself. For instance, the initialization of the merge-split sort program initializes an unsorted vector of records with random strings in their key fields. The time spent on the initialization depends on the generation of random strings; a complicated random string generating algorithm can well consume a lot of time. Indeed, if this initialization is included in the execution time of the program, and such an initialization is performed in parallel, it is possible to get a better speedup than the ideal speedup since ideally this parallel algorithm does not yield a linear speedup.

In order to obtain a fair speedup measurement, all the programs in the experiments partition their problems by creating a certain number of processes according to the number of processors used. As a result of such a parameterized partitioning, each program does its best for a given number of processors. If a fixed partitioning strategy were used, one could demonstrate better (or worse) speedups, but such an approach is unreasonable. To help assess overall system performance, all the speedups are compared against the ideal.

The results for the parallel 3-D PDE solver are shown in Figure 5, where $n = 50^3$. The dashed line in the figure is the ideal (linear) speedup curve. Note that the program experiences *super-linear speedup.*

At first glance this result seems impossible because the fundamental law of parallel computation says that a parallel solution utilizing $p$ processors can improve the best sequential solution by at most a factor of $p$. Something must be interacting in either the program or the shared virtual memory implementation. Since the algorithm in the program is a straightforward transformation from the sequential Jacobi algorithm and all the processes are synchronized at each

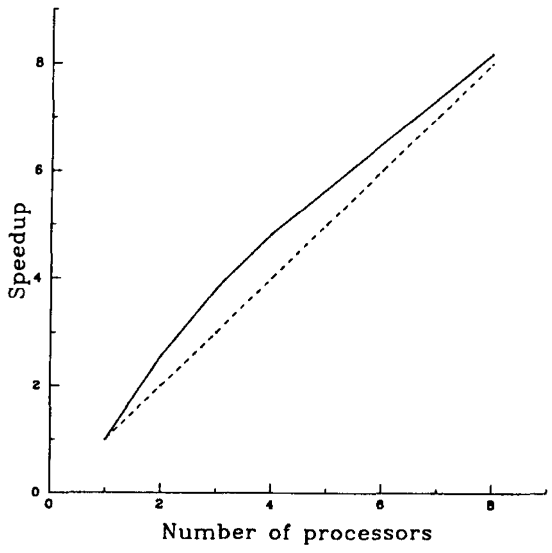

Fig. 5. Speedups of a 3-D PDE where $n = 50^3$.

iteration, the algorithm cannot yield super-linear speedup. So, the speedup must be in the shared virtual memory implementation.

The shared virtual memory system can indeed provide super-linear speedups because the fundamental law of parallel computation assumes that every processor has an infinitely large memory, which is not true in practice. In the parallel 3-D PDE example above, 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 is a large amount of paging between the physical memory and disk.

Figure 6 shows the disk paging performance on one and two processors. The solid line in the figure shows the number of disk I/O pages when the program runs on one processor. The dashed and dotted lines show the numbers of disk I/O pages when the program runs on two processors—the dashed line for the processor with initialized data (processor 1) and the dotted line (which can hardly be seen) for the processor without initialized data (processor 2). The two curves are very different because the program initializes its data structures only on one processor. Since the virtual memory paging in the Aegis operating system performs an approximated LRU strategy, and the pages that move to processor 2 are recently used on processor 1, processor 1 had to page out some pages that it needs later, causing more disk I/O page movement. This explains why the number of disk I/O pages on processor 1 decreases after the first few iterations.

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 7 shows another speedup curve for the 3-D PDE program, but now with $n = 40^3$, in which case the data structure of the problem is not larger than the

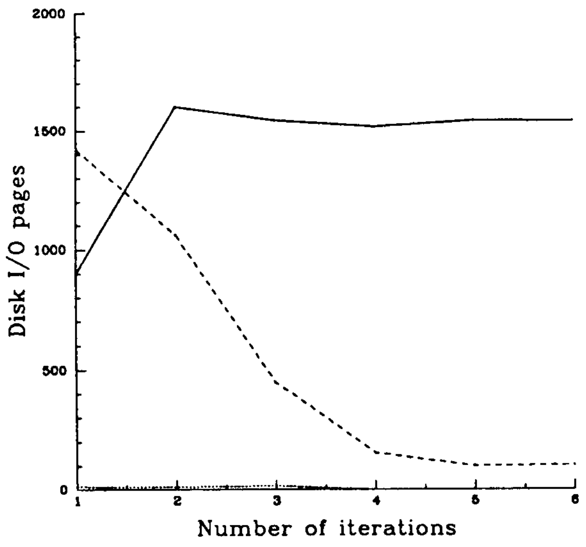

Fig. 6. Disk paging on one processor and two processors.

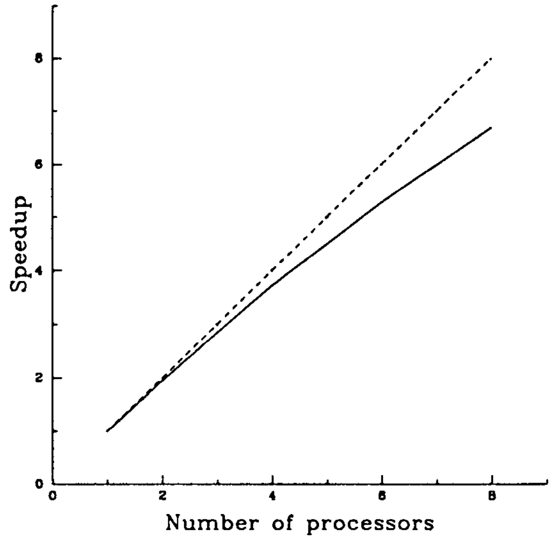

Fig. 7. Speedups of a 3-D PDE where $n = 40^3$.

physical memory on a processor. This curve is what we see in most parallel computation papers. The curve is quite similar in fact to that generated by similar experiments on CM*, a pioneer shared memory multiprocessor [14, 24, 28]. Indeed, the shared virtual memory system is as good as the best curve in the published experiments on CM* for the same program, but the efforts and costs of the two approaches are dramatically different. In fact, the best curve in CM* was obtained by keeping the private program code and stack in the local memory

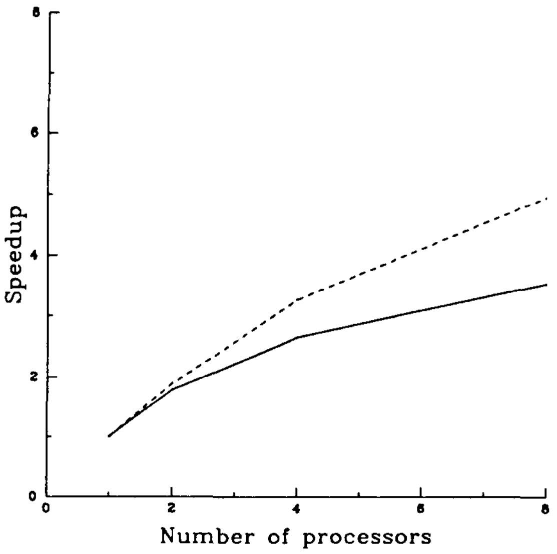

Fig. 8. Speedup of the merge-split sort.

on each processor. The main reason that the performance of this program is so good in the shared virtual memory system is that the program exhibits a high degree of locality on each execution path. While the shared virtual memory system pays the cost of local memory references, CM* pays the cost of remote memory references across its K maps.

Parallel sorting on a loosely coupled multiprocessor is generally difficult. The speedup curve of the parallel merge-split sort of 200K elements shown in Figure 8 is not very good. In theory, even with no communication costs, this algorithm does not yield linear speedup. Thus, to provide a better comparison of performance we added the dashed line in the figure to show the speedup when the costs of all memory references are the same. Also recall that our program uses the best strategy for a given number of processors. For example, one merge-split sorting is performed when running the program on one processor, four when running on two processors, and $n^2$ when running on an $n$ processor. Using a fixed number of blocks for any number of processors would result in a better speedup, but such a comparison would be unfair.

Figure 9 shows the speedup curve of the parallel dot-product program in which each vector has 128K elements. It is included here so as not to paint too bright a picture. To be fair, the program assumes that the two vectors have a random distribution on each processor. Even with such an assumption, the speedup curve is not good, as indicated by the solid line in Figure 9. If the two vectors are located on one processor, there is no speedup at all, as indicated by the dotted curve in Figure 9, because the ratio of the communication cost to the computation cost in this program is large. For programs like dot-product, a shared virtual memory system is not likely to provide significant speedup.

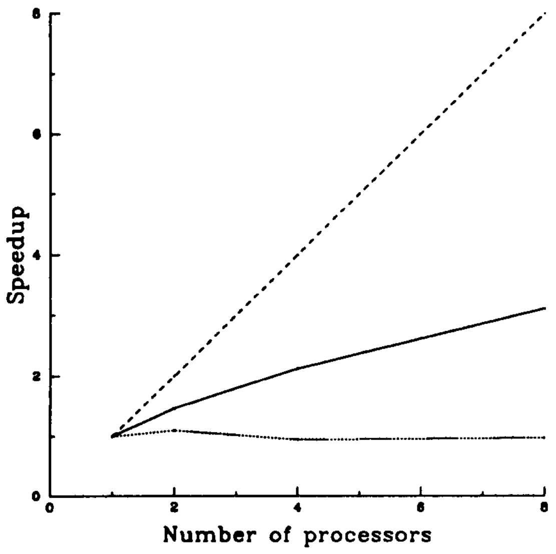

Fig. 9. Speedup of the dot-product program.

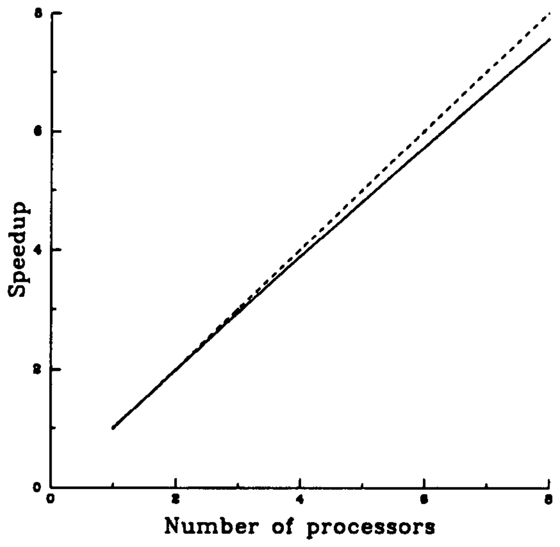

Fig. 10. Speedup of the matrix multiplication program.

Figure 10 shows the speedup curve of the matrix multiplication program for $C = AB$ where both $A$ and $B$ are 128-by-128 square matrices. This example shows the good side of the shared virtual memory system. The speedup curve is close to linear because the program exhibits a high degree of localized computation, even though the input matrices are shared heavily.

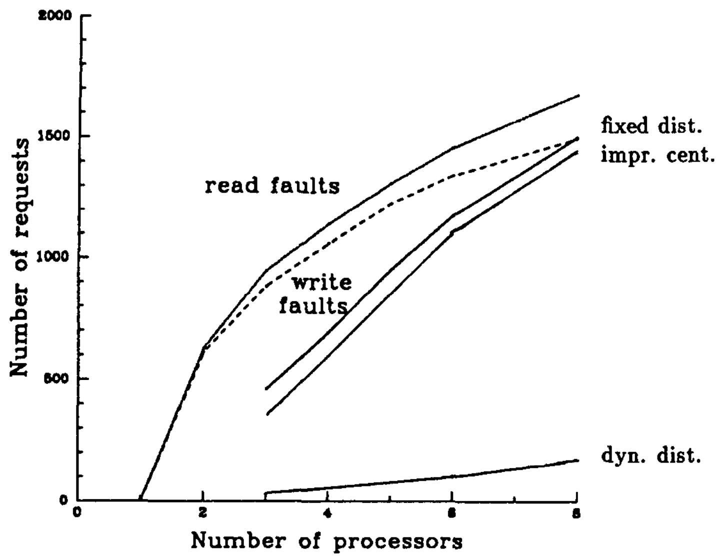

Fig. 11. Forwarding requests.

In general, the experimental results show that a shared virtual memory implementation is indeed practical even on a very loosely coupled architecture such as the Apollo ring.

## 6.3 Memory Coherence Algorithms

Our experiments indicate that all three memory coherence algorithms have similar numbers of page faults. Due to the limitation on the number of processors, we could not show explicit differences among the three algorithms by comparing their system performance. The alternative approach we took was to measure the total number of messages used in an algorithm. In fact, the number of forwarding requests was used as a criterion for comparing algorithms.

Figure 11 shows the number of forwarding requests for locating true pages during the first six iterations of the 3D PDE program using the improved centralized manager algorithm, the fixed distributed manager algorithm, and the dynamic distributed manager algorithm.

In the fixed distributed manager algorithm, the manager mapping function is $H(p) = p \bmod N$, where $p$ is a page number and $N$ is the number of processors. The curve of the forwarding requests of the fixed distributed manager algorithm is similar to that of the improved centralized manager algorithm because both algorithms need a forwarding request to locate the owner of the page for almost every page fault that occurs on a nonmanager processor. Since the workload of the fixed distributed manager algorithm is a little better than that of the improved centralized manager algorithm, the performance of the former is a little better than the latter as the number of processors increases.

The figures show that the overhead of the dynamic distributed manager algorithm is much less than that of the other two algorithms. This confirms the analysis on the dynamic distributed manager algorithm. The main reason it is better is that the *prob-owner* fields usually give correct hints (thus the number of forward requests is very small) and within a short period of time the number of processors sharing a page is small.

## 7. CONCLUSIONS

This paper has studied two general classes of algorithms (centralized manager and distributed manager) for solving the memory coherence problem, and both of them have many variations. The centralized manager algorithm is straightforward and easy to implement, but it may have a traffic bottleneck at the central manager when there are many page faults. The fixed distributed manager algorithm alleviates the bottleneck, but, on average, a processor still needs to spend about two messages to locate an owner. The dynamic distributed manager algorithm and its variations seem to have the most desirable overall features. As mentioned earlier, the dynamic distributed manager algorithm may need as little as one message to locate an owner. Furthermore, Theorem 3 shows that by using fewer broadcasts the average number of messages for locating a page is a little less than two for typical cases. Further refinement can also be done by distributing copy sets.

In general, dynamic distributed manager algorithms perform better than other methods when the number of processors sharing the same page for a short period of time is small, which is the normal case. The good performance of the dynamic distributed manager algorithms shows that it is possible to apply it to an implementation on a large-scale multiprocessor.

Our experiments with the prototype system indicate that many parallel programs exhibit good speedups on loosely coupled multiprocessors using a shared virtual memory. Performance is generally good even for parallel programs that share data in a fine-grained way, as long as most of the references are read-only. We conjecture that the main class of parallel programs that would perform poorly are those with frequent updates to shared data or those with excessively large data sets that are only read once (such as the dot-product example).

Because of resource limitations we could not run our experiments on more than eight processors. Thus we do not know for sure how well the shared virtual memory system will scale. On the other hand, the data we have gathered on the distributed manager algorithms gives us every reason to believe that it will scale well, as argued earlier. A more difficult question is what the best page size should be. Currently we only have experience with two sizes: 1K byte and 32K bytes. Since this parameter is not only system dependent but also applications dependent, much more experience with a real implementation is necessary before all of the trade-offs are fully understood.

The memory coherence algorithms proposed in this paper and the success of our experiments on our prototype system suggest the possibility of using a shared virtual memory system to construct a large-scale shared memory multiprocessor system. Such a project is, in fact, underway at Princeton in collaboration with the DEC Systems Research Center. Through this project we intend to address many of the unanswered questions raised by this research.

## APPENDIX A
## IMPROVED CENTRALIZED MANAGER ALGORITHM

**CentralManager**

```
    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 and a copy of p;
        PTable[ p ].access := read;
        Unlock( PTable[ p ].lock );

    Read server:
        Lock( PTable[ p ].lock );
        IF I am owner THEN BEGIN
            PTable[ p ].copyset
                := PTable[ p ].copyset ⋃ {RequestNode};
            PTable[ p ].access := read;
            send p;
            END
        ELSE IF I am manager THEN BEGIN
            Lock( ManagerLock );
            forward request to owner[ p ];
            Unlock( ManagerLock );
            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 and p's copyset;

        Invalidate( p, PTable[ p ].copyset );
        PTable[ p ].access := write;
        PTable[ p ].copyset := {};
        Unlock( PTable[ p ].lock );


    Write server:
        Lock( PTable[ p ].lock );
        IF I am owner THEN BEGIN
            send p and PTable[ p ].copyset;
            PTable[ p ].access := nil;
            END
    ELSE IF I am manager THEN BEGIN
        Lock( ManagerLock );
        forward request to owner[ p ];
        owner[ p ] := RequestNode;
        Unlock( ManagerLock );
        END;
    Unlock( PTable[ p ].lock );
```

## APPENDIX B
## BROADCAST DISTRIBUTED MANAGER ALGORITHM

**BroadcastManager**

*Read fault handler:*

```
    Lock( PTable[ p ].lock );
    broadcast to get p for read;
    PTable[ p ].access := read;
    Unlock( PTable[ p ].lock );
```

*Read server:*

```
    Lock( PTable[ p ].lock );
    IF I am owner THEN BEGIN
        PTable[ p ].copyset :=
            PTable[ p ].copyset ⋃ [ RequestNode ];
        PTable[ p ].access := read;
        send p;
        END;
    Unlock( PTable[ p ].lock );
```

*Write fault handler:*

```
    Lock( PTable[ p ].lock );
    broadcast to get p for write;
    Invalidate( p, PTable[ p ].copyset );
    PTable[ p ].access := write;
    PTable[ p ].copyset := {};
    PTable[ p ].owner := self;
    Unlock( PTable[ p ].lock );
```

*Write server:*

```
    Lock( PTable[ p ].lock );
    IF I am owner THEN BEGIN
        send p and PTable[ p ].copyset;
        PTable[ p ].access := nil;
        END;
    Unlock( PTable[ p ].lock );
```

## APPENDIX C PROPERTIES OF DYNAMIC DISTRIBUTED MANAGER ALGORITHMS

In the following, we first show some properties of the *probOwner* graph by assuming that page faults are generated and processed sequentially. In other words, it is assumed that if processor $i$ has a fault on page $p$, then no other processor has a fault on page $p$ until processing the page fault on processor $i$ is complete. We then show the correctness of the concurrent page fault case by reducing it to a sequential case.

LEMMA 1. *If page faults of page p occur sequentially, every probOwner graph* $G_p = (V, E_p)$ *has the following properties:*

(1) *there is exactly one node i such that* $(i, i) \in E_p$;
(2) *graph* $G'_p = (V, E_p - (i, i))$ *is acyclic; and*
(3) *for any node x, there is exactly one path from x to i.*

PROOF. By induction on the number of page faults on page $p$. Initially, all the *probOwners* of the processors in $V$ are initialized to a default processor. Obviously, all three properties are satisfied.

After $k$ page faults, the *probOwner* graph $G_p$ satisfies the three properties as shown in Figure 12(a). There are two cases when a page fault occurs on processor $j$.

(1) If it is a read-page fault, the path from $j$ to $i$ is collapsed by the read-fault handler and its server in the algorithm such that all the nodes on the path now point to $j$ (Figure 12(b)). The resulting graph satisfies 1 since $(i, i)$ is deleted from $E_p$ and $(j, j)$ is added to $E_p$. It satisfies 2 because the subgraphs $g_j, g_u, \ldots, g_v, g_i$ ($g_x$ is a subgraph of $G_p$ rooted at node $x$) are acyclic and they are not changed. For any node $x \in V$, there is exactly one path to $i$. Suppose the first node in the path (in the node set $\{j, u, \ldots, v, i\}$) is $y$. The edge from $y$ is changed to $(y, j)$, so there is no other path from $y$ to $j$.

(2) If it is a write-page fault and there is no read-only copy of page $p$, then the resulting graph is the same as the read-page fault case. If it is a write-page fault and there are $r$ read-only copies on processors $v_1, \ldots, v_r$, then in addition to collapsing the path from $j$ to $i$, the invalidation procedure makes nodes $v_1, \ldots, v_r$ point to $j$ (Figure 12(c)). The resulting graph satisfies 1 since $(i, i)$ is deleted from the graph and $(j, j)$ is added. A subgraph $g_x$ is not equal to $g'_x$ only if there is a node $w$ in $g_x$ such that $w \in \{v_1, \ldots, v_r\}$. However, $g'_x$ is acyclic because making $w$ point to $j$ does not isolate the subgraph $g_w$ from $g_x$. Hence, the subgraphs $g'_j, g'_u, \ldots, g'_v, g'_i$ and $g_{v_1}, \ldots, g_{v_r}$ are acyclic, and the resulting graph satisfies 2. Similar to the read fault case, any node $x \in V$ has exactly one path to $i$. Suppose that the first node in the path (in the node set $\{j, u, \ldots, v, i, v_1, \ldots, v_r\}$) is $y$. The edge from $y$ is changed to $(y, j)$, so there is no other path from $y$ to $j$. □

Lemma 1 demonstrates that any page fault can find the true owner of the page if the page faults to the same page are processed sequentially. This shows the correctness of the algorithm in the sequential case.

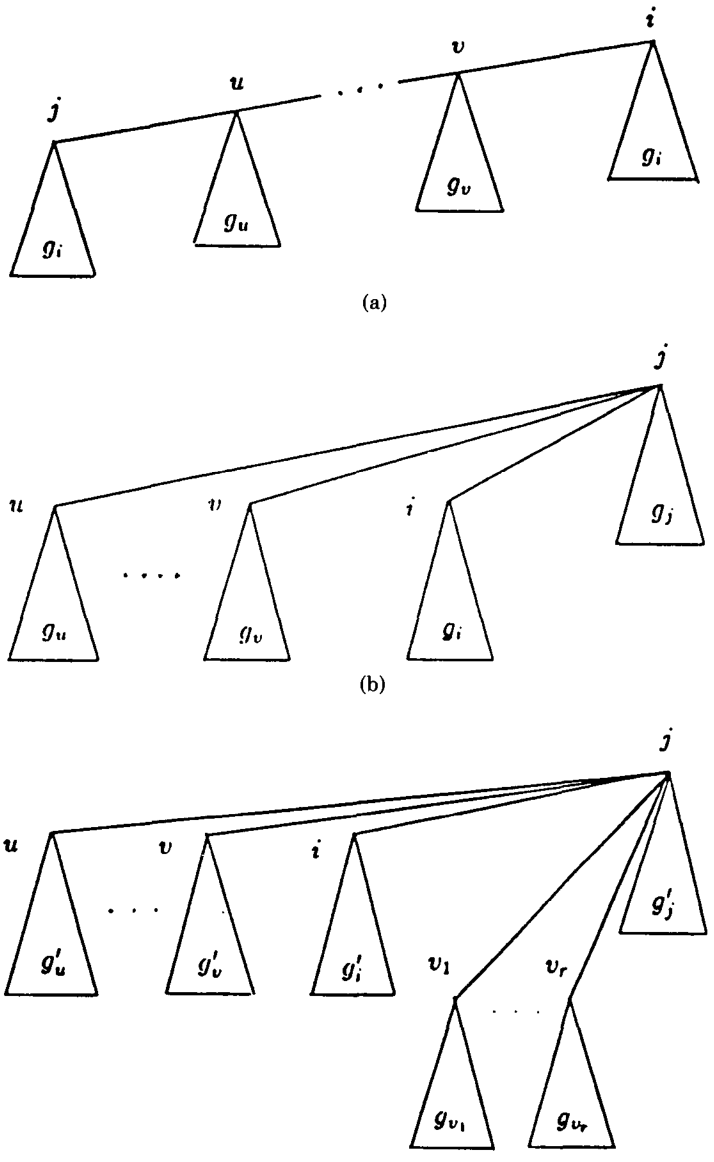

Fig. 12. Induction of a *prob Owner* graph.

The worst-case number of forwarding messages for the sequential case is given by the following corollary:

COROLLARY 2. *In the N-processor shared virtual memory system, it takes at most $N - 1$ messages to locate a page if page faults are processed sequentially.*

PROOF. By Lemma 1, there is only one path to the true owner and there is no cycle in the *probOwner* graph. So, the worst case occurs when the *probOwner* 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 $v_1$ generates $N - 1$ forwarding messages in finding the true owner $v_N$. □

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

LEMMA 2. *There exists a probOwner 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 Corollary 2 in which page faults occur on processors $v_N, v_{N-1}, \ldots, v_1$ sequentially. □

It is interesting that the worst-case single-fault situation is coincident with the best-case $N$-fault situation. Also, once the worst-case situation occurs, *all* processors know the true owner. The immediate question that now arises is what is the *worst-case* performance for $K$ faults to the same page in the sequential case. The following lemma answers the question:

LEMMA 3. *For an N-processor shared virtual memory, using Algorithm 2, the worst-case number of messages for locating the owner of a single page K times as a result of K sequential page faults is* $O(N + K \log N)$.

PROOF. The algorithm reduces to the *type-0 reversal find operation* for solving the set union-find problem [44].[1] For a *probOwner* graph $G_p = (V, E_p)$, define the node set $V$ to be the set in the set union-find problem and define the node $i \in V$ such that $(i, i) \in E_p$ to be the canonical element of the set. A read-page fault of page $p$ on processor $j$ is then a type-0 reversal find operation $Find(j, V)$ in which the canonical element of the set is changed to $j$. A write-page fault of page $p$ on processor $j$ is a type-0 reversal find operation plus the collapsing (by an invalidation) of the elements in the copy set of the page. Although the collapsing changes the shape of the graph, it does not increase the number in the find operation. Then, according to the proof by Tarjan and Van Leeuwen [44], the worst-case number of messages for locating $K$ owners of a page is $O(N + K \log N)$. □

Note that without changing the ownership on a read-page fault, the algorithm still works correctly, but the worst-case bound increases when $N$ is large. In that case, the total number of messages for locating $K$ owners depends on the configuration of the *probOwner* graph. If the graph is a chain, then it can be as bad as $O(K\ N)$. On the other hand, if the graph is a balanced binary true it is

[1] The reduction to set-union find was motivated by Fowler's analysis on finding objects [22], though the reduction methods are different.

$O(K \log N)$, and if the graph is at a state in which every processor knows the owner, it is $O(K)$.

All the lemmas and corollaries above are for the sequential page-fault case. In practice, however, the sequential page-fault case is unlikely to happen often, so it is necessary to study the concurrent page-fault case. An important property of the algorithm is the atomicity of each fault handler and its server, provided by the locking and unlocking mechanism. For convenience, we state it in the following lemma:

LEMMA 4. *If a page fault for page p traverses a processor q, then other faults for p that need to traverse processor q, but have yet to, cannot be completed until the first fault completes.*

PROOF. Suppose processing a page fault that occurred on processor $i$ has traversed processor $u$. The server of the fault handler atomically sets the *probOwner* field in the page table entry on processor $u$ to $i$. The requests for processing other page faults are forwarded to processor $i$. Since the page table entry on processor $i$ is locked, these requests are queued on processor $i$ until processing the page for processor $i$ is complete. □

Our intention is to show that there exists a sequential page fault processing sequence that matches any given concurrent page fault processing so that the results for the sequential case apply to the concurrent case. Processing concurrent page faults that occur on processors $v_1, \ldots, v_k$ (it is possible that $v_i = v_j$ when $i \neq j$) is said to be *matched* by a sequential processing of a $K$ page fault sequence $(v_1, \ldots, v_k)$ if processing the page fault on processor $v_i$ in the concurrent case traverses the same processors in the same order as in the sequential processing case.

Consider an example in which the owner of a page is $v$ and page faults occur on processor $i$ and processor $j$ concurrently. Suppose that the first common node in the *probOwner* graph that the requests for processing both page faults need to traverse is $u$ (Figure 13(a)). If the request from processor $i$ traverses $u$ first, the algorithm sets the *probOwner* field in the page table entry on processor $u$ to $i$. When the request from processor $j$ arrives at processor $u$, it is forwarded to processor $i$, but the page table entry on processor $i$ is locked until processing the page fault for processor $i$ is complete. So, the request from processor $j$ is queued at the page table entry of processor $i$, while the request from processor $i$ is traversing the rest of the path from $u$ to $v$. The *probOwner graph* when the processing is complete is shown in Figure 13(b). When the lock on processor $i$ is released, the request from processor $j$ continues. The resulting graph is shown in Figure 13(c). Thus, the request from processor $i$ traversed the path $i, \ldots, u, \ldots, v$ and the request from processor $j$ traversed the path $j, \ldots, u, i$. This is equivalent to the case in which the following events occur sequentially:

—a page fault occurs on processor $i$;

—the system processes the page fault;

—a page fault occurs on processor $j$; and

—the system processes the page fault.

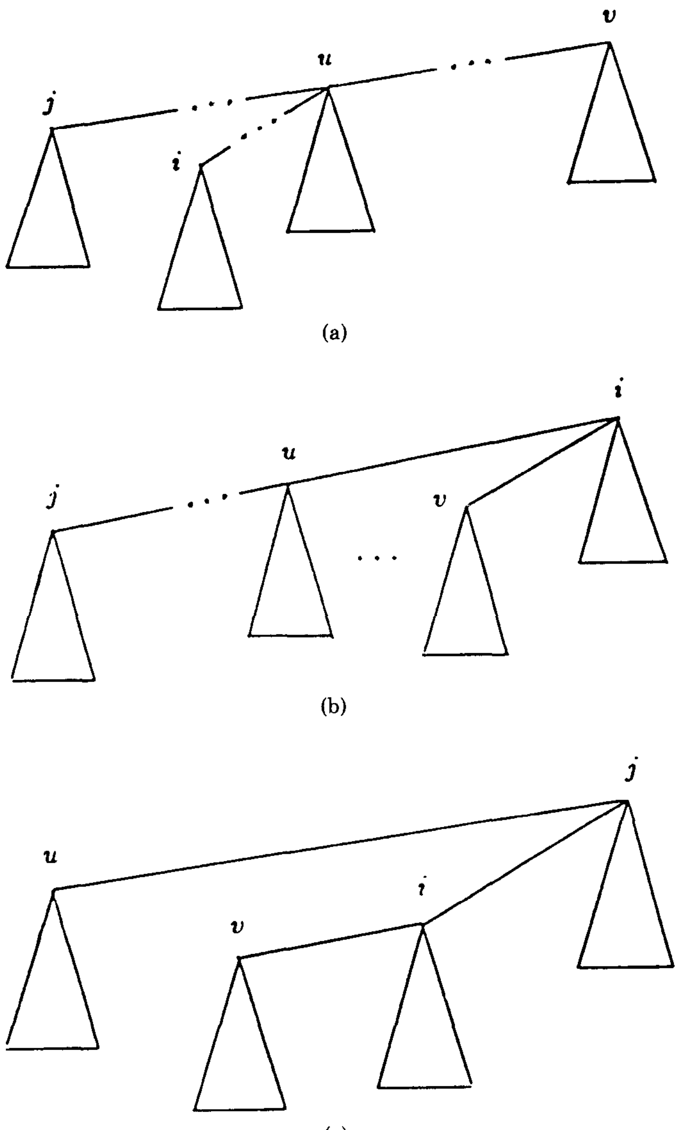

Fig. 13. Two concurrent page faults.

The matched sequential page-fault processing sequence is therefore $(i, j)$. Obviously, if the request from processor $j$ traverses processor $u$ first, then the matched sequential page-fault processing sequence is $(j, i)$.

LEMMA 5. *For any concurrent page-fault processing, there exists a matched sequential page-fault processing sequence.*

PROOF. By induction on the number of page faults. For one page fault, it is obviously true. For $k + 1$ page fault processing, we look at a page fault on processor $i$. By Lemma 4, the following is true:

(1) $k_1$ page fault processing activities are done before processing the page fault for processor $i$ is complete;

(2) there are $k_2$ page fault processing activities after processing the page fault for processor $i$ is complete; and

(3) $k_1 + k_2 = k$.

According to the assumption, there is a matched sequential page-fault sequence $(u_1, \ldots, u_{k_1})$ for the $k_1$ concurrent page-fault processings and there is a matched sequential page-fault sequence $(v_1, \ldots, v_{k_2})$ for the $k_2$ concurrent page-fault processings. The matched page-fault processing sequence is therefore $(u_1, \ldots, u_{k_1}, i, v_1, \ldots, v_{k_2})$. □

PROOF OF THEOREMS 1 AND 2. This lemma not only enables us to apply all the results for the sequential case to the general case but also shows that the find operations in the set-union problem can be done in parallel if each find can be broken into small atomic operations in the same manner as this algorithm. We can therefore prove the two theorems, one for the correctness (Theorem 1 by Lemma 5 and Corollary 2) and one for the worst-case performance of the algorithm (Theorem 2 by Lemma 5 and Lemma 3). □

## ACKNOWLEDGMENTS

We wish to thank John Ellis for his invaluable suggestions and helpful discussions in the early stages of this research. We also wish to thank Professor Alan Perlis for his continual help and inspiration.

## REFERENCES

1. AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass. 1974.
2. APOLLO COMPUTER. *Apollo DOMAIN Architecture.* Apollo Computer, Inc., Chelmsford, Mass., 1981.
3. ARCHIBALD, J., AND BAER, J. An economical solution to the cache coherence problem. In *Proceedings of the 11th Annual Symposium on Computer Architecture* (Ann Arbor, Mich., June 1984). pp. 355–363.
4. BITTON, D., DEWITT, D. J., HSAIO, D. K., AND MENON, J. A taxonomy of parallel sorting. *ACM Comput. Surv. 16*, 3 (Sept. 1984), 287–318.
5. BOBROW, D. G., BURCHFIEL, J. D., MURPHY, D. L., AND TOMLINSON, R. S. TENEX: A paged time-sharing system for the PDP-10. *Commun. ACM 15*, 3 (Mar. 1972), 135–143.
6. BOLT, BERANEK, AND NEWMAN. *Butterfly Parallel Processor Overview.* Bolt, Beranek, and Newman, Advanced Computers Inc., Cambridge, Mass., 1985.
7. BRINCH, H. *Operating System Principles.* Prentice-Hall, Englewood Cliffs, N.J., 1973.
8. CARRIERO, N., AND GELERNTER, D. The S/Net's Linda kernel. *ACM Trans. Comput. Syst. 4*, 2 (May 1986), 110–129.
9. CENSIER, L. M., AND FEAUTRIER, P. A new solution to coherence problems in multicache systems. *IEEE Trans. Comput. C-27*, 12 (Dec. 1978), 1112–1118.
10. CHERITON, D. R. The VMP multiprocessor: Initial experience, refinements and performance evaluation. In *Proceedings of the 14th Annual Symposium on Computer Architecture* (Pittsburgh, Pa., June 1987).
11. CHERITON, D. R., AND STUMM, M. The multi-satellite star: Structuring parallel computations for a workstation cluster. *J. Distributed Comput.* To appear.

12. COX, A. L., AND FOWLER, R. J. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATIMUM. Tech. Rep. 263, Dept. of Computer Science, University of Rochester, Rochester, N.Y., Mar. 1989.
13. DALEY, R. C., AND DENNIS, J. B. Virtual memory, processes, and sharing in MULTICS. *Commun. ACM 11*, 5 (May 1968), 306-312.
14. DEMINET, J. Experience with multiprocessor algorithms. *IEEE Trans. Comput. C-31*, 4 (Apr. 1982).
15. DENNING, P. J. Virtual memory. *ACM Comput. Surv. 2*, 3 (Sept. 1970), 153-189.
16. DENNING, P. J. On modeling program behavior. In *Proceedings on the Spring Joint Computer Conference* (Atlantic City, N.J., May 16-18, 1972). AFIPS Press, Montudle, N.J., 1972, pp. 937-944.
17. DENNING, P. J. Working sets past and present. *IEEE Trans. Softw. Eng. SE-6*, 1 (Jan. 1980), 64-84.
18. EGGERS, S. J., AND KATZ, R. H. A characterization of sharing in parallel programs and its applications to coherence protocol evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture* (Honolulu, June 1988).
19. FINKEL, R., AND MANBER, U. BIB—A distributed implementation of backtracking. In *The 5th International Conference on Distributed Computing Systems* (Denver, Colo., May 1985).
20. FITZGERALD, R., AND RASHID, R. F. The integration of virtual memory management and interprocess communication in Accent. *ACM Trans. Comput. Syst. 4*, 2 (May 1986) 147-177.
21. FLEISCH, B. D. Distributed shared memory in a loosely coupled distributed system. In *Proceedings of the ACM SIGCOMM 87 Workshop, Frontiers in Computer Communications Technology* (Stowe, Vt., Aug. 11-13, 1987). ACM, New York, 1987, pp. 317-327.
22. FOWLER, R. J. Decentralized object finding using forwarding addresses. Ph.D. dissertation, Dept. of Computer Science, Univ. of Washington, Seattle, 1986.
23. FRANK, S. J. Tightly coupled multiprocessor system speeds memory-access times. *Electronics 57*, 1 (Jan. 1984), 164-169.
24. FULLER, S., OUSTERHOUT, J., RASKIN, L., RUBINFELD, P., SINDHU, P., AND SWAN, R. Multimicroprocessors: An overview and working example. In *Proceedings of the IEEE 66*, 2 (Feb. 1978) pp. 214-228.
25. GOODMAN, J. R. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual Symposium on Computer Architecture* (Stockholm, June 1983), pp. 124-131.
26. HOARE, C. A. R. Quicksort. *Comput. J. 5*, 1 (1962), 10-15.
27. HOARE, C. A. R. Monitors: An operating system structuring concept. *Commun. ACM 17*, 10 (Oct. 1974), 549-557.
28. JONES, A. K., AND SCHWARZ, P. Experience using multiprocessor systems—A status report. *ACM Comput. Surv. 12*, 2 (June 1980), 121-166.
29. JONES, A. K., CHANSLER, R. J., DURHAM, I. E., SCHWANS, K., AND VEGDAHL, S. StarOS, a multiprocessor operating system for the support of task forces. In *Proceedings of the 7th Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 10-12, 1979). ACM, New York, 1979, pp. 117-127.
30. KARLIN, A. R., MANASSE, M. S., RUDOLPH, L., AND SLEATOR, D. D. Competitive snoopy caching. In *Proceedings of the 27th Symposium on Foundation of Computer Science* (Toronto, 1986). pp. 244-254.
31. KATZ, R. H., EGGERS, S. J., WOOD, D. A., PERKINS, C. L., AND SHELDON, R. G. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual Symposium on Computer Architecture* (Boston, Mass., June 1985). pp. 276-283.
32. LEACH, P. J., STUMPF, B. L., HAMILTON, J. A., AND LEVINE, P. H. UIDs as internal names in a distributed file system. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Ottawa, Ontario, Canada, Aug. 18-20, 1982). ACM, New York, 1982, pp. 34-41.
33. LEACH, P. J., LEVINE, P. H., DOUROS, B. P., HAMILTON, J. A., NELSON, D. L., AND STUMPF, B. L. The architecture of an integrated local network. *IEEE J. Selected Areas in Commun. SAC-1*, 5 (1983).
34. LI, K. Shared virtual memory on loosely coupled multiprocessors. Ph.D. dissertation, Dept. of Computer Science, Yale University, New Haven, Conn., Oct. 1986. Also Tech. Rep. YALEU-RR-492.

35. LI, K. *IVY: A shared virtual memory system for parallel computing.* In *Proceedings of the 1988 International Conference on Parallel Processing* (Aug. 1988). Pennsylvania State University Press, 1988, pp. 94–101.
36. LI, K., AND HUDAK, P. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing* (Calgary, Alberta, Aug. 11–13, 1986). ACM, New York, 1986, pp. 229–239.
37. LI, K., AND SCHAEFER, R. A hypercube shared virtual memory. In *Proceedings of the 1989 International Parallel Processing Conference* (Dufage, Ill., Aug. 1989).
38. RASHID, R. F., AND ROBERTSON, G. G. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th Symposium on Operating Systems Principles* (Pacific Grove Calif., Dec. 14–16). ACM, New York, 1981, pp. 64–75.
39. SEITZ, C. L. The cosmic cube. *Commun. ACM 28*, 1 (Jan. 1985), 22–33.
40. SMITH, A. J. Cache memories. *ACM Comput. Surv. 14*, 3 (Sept. 1982), 473–530.
41. SPECTOR, A. Z. Multiprocessing Architectures for Local Computer Networks. Ph.D. dissertation, STAN-CS-81-874, Stanford University, Dept. of Computer Science, Stanford, Calif., Aug. 1981.
42. SPECTOR, A. Z. Performing remote operations efficiently on a local computer network. *Commun. ACM 25*, 4 (Apr. 1982), 260–273.
43. TANG, C. K. Cache system design in the tightly coupled multiprocessor system. In *Proceedings of AFIPS National Computer Conference* (New York, N.Y., June 7–10, 1976). AFIPS Press, Montvale, N.J. 1976, pp. 749-753.
44. TARJAN, R. E., AND VAN LEEUWEN, J. Worst-case analysis of set union algorithms. *J. ACM 31*, 2 (Apr. 1984), 245–281.
45. THOMPSON, J. Efficient analysis of caching systems. Ph.D. dissertation, University of California at Berkeley, Dept. of Computer Science, Oct. 1987. Also Tech Rep. UCB/CSD 87/374.
46. YEN, W. C., YEN, D. W. L., AND FU, K. Data coherence problem in a multicache system. *IEEE Trans. Comput. C-34*, 1 (Jan. 1985), 56–65.

Received February 1987; revised December 1987; October 1988, and May 1989; accepted May 1989