IVY: A Shared Virtual Memory System for Parallel Computing

Kai Li
Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

A *shared virtual memory* system can provide a virtual address space shared among all processors in a loosely-coupled multiprocessor. This paper shows that such a memory can solve many problems in message passing systems on loosely-coupled multiprocessors, and describes the design and implementation of a prototype shared virtual memory system, IVY, implemented on an Apollo ring network. The experiments on the prototype system show that parallel programs using a shared virtual memory yield almost linear and occasionally super-linear speedups and that it is practical to implement such a system on existing architectures.

Introduction

Much of the work on distributed computing has focused on message passing models such as Hoare's communicating sequential processes [16] and Actor [15], perhaps because message passing matches the basic communication mechanism in loosely-coupled multiprocessors. Many people have studied shared memory models for tightly coupled multiprocessors, but few have studied that model for loosely-coupled multiprocessors. Because not enough work has been done, it has not been clear whether a message passing model is better than a shared memory model for parallel computation on loosely-coupled multiprocessors. It has also not been clear whether it is possible to design an efficient system to support the shared memory model on loosely-coupled multiprocessors.

Systems based on message passing suffer mainly in two aspects: passing complex data structures and process migrations. This paper shows that a solution to these problems is to build a *shared virtual memory* . The shared virtual memory 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 that processes can run on different processors in parallel. The shared virtual memory keeps its memory pages coherent all the time and data can naturally *migrate* between processors on demand [23,22]. 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*.

A prototype shared virtual memory has been implemented on a network of Apollo workstations. A number of practical parallel program examples are chosen to run on the prototype system. The experimental results show that parallel programs using such a not well-tuned, user-mode shared virtual memory system yield almost linear and occasionally super-linear speedups over a uniprocessor. The success of this implementation suggests a new operating mode for loosely-coupled multiprocessor 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.

Shared Memory vs. Message Passing

Message passing in concurrent systems is characterized by multiple threads of control. A pure message passing system usually does not have any shared global data; instead processes access ports or mailboxes to achieve interprocess communication. Parallel programs need to use primitives such as *send* and *receive* explicitly through channels, ports, or mailboxes. Although programmers can use these primitives to synchronize parallel programs, they need to be conscious of data movement between processes at all times.

Remote procedure call is a mechanism for language-level transfer of control and data between programs in disjoint address spaces whose primary communication medium is a narrow channel [24]. A remote procedure call mechanism allows programmers to worry less about data movement and provides clients with a fairly transparent interface so that remote procedure calls look much like local procedure calls. However, the transparency of remote procedure calls is limited because a remote procedure call mechanism actually simulates the execution in the same address space using completely different address spaces.

Since both message passing and remote procedure calls deal with multiple address spaces, they both have difficulties with passing complex data structures. In fact, the difficulty of passing complex data structures is the main drawback of message passing and remote procedure calls for parallel programming. For example, passing a list data structure by sending messages will introduce considerable complexity in programming and substantial overhead in both space and time [14]. In a remote procedure call, there is no good way to pass a pointer argument [24]. This problem becomes more severe when the data structures are fundamental to a language being implemented on a parallel machine.

In contrast, a shared memory multiprocessor has no difficulty passing pointers because processors can share a single address space. Therefore, there is no need to pack and unpack the data structures containing pointers in messages. Passing a list data structure simply requires passing a pointer.

Another problem with message passing systems is the difficulty of process migration because there are multiple address

spaces. When migrating a process, all the operating system resources allocated by the process have to be moved together; this is expensive [25]. In the case where a process has a few opened ports and files, the pending messages and file access control blocks need to be transferred. Furthermore, the code and the stack of the process have to be moved because there is no easy way to translate the contents of different address spaces efficiently on the fly.

In a shared memory multiprocessor system, a process migration only requires moving a process from the ready queue on the source processor to the ready queue on the destination processor because process control block, code, and stack are all in the same address space.

Some systems use a set of primitives to access a global space that is used to store shared data structures of processes [8,5]. Although programming the global space does not require data movement as much as message passing, programmers still have to explicitly use the primitives. In a primitive global-space system, passing complex data structures and process migration are as difficult as in message passing systems, since accessing the data structures and process migration are by value or by name. Furthermore, using primitives may greatly reduce the efficiency of parallel programs because a primitive operation requires at least one procedure call, which costs much more than a simple memory reference.

Both data structure passing and process migration are important for implementing parallel programming languages. Although some implementations of parallel programming languages are based on a message passing facility, implementing existing parallel languages on a shared memory multiprocessor can greatly simplify the implementations. In summary, shared memory is highly desirable for parallel computation.

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. In short, a shared virtual memory provides clients with the same interface as the shared memory address space on a shared-memory multiprocessor.

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 traditional virtual memory [11], the shared memory itself exists only *virtually*. A memory reference will cause 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 faulting memory reference is the target of a write operation, then the memory mapping manager must guarantee the atomicity of the operation [23].

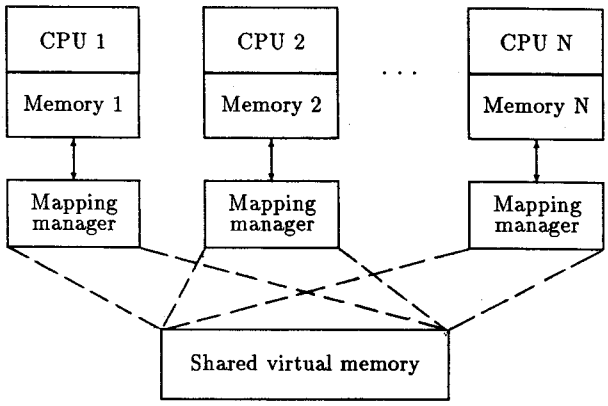

Figure 1: Shared virtual memory mapping

In a shared virtual memory system, the model of a parallel program is a set of *processes* (or threads) that share a single virtual memory address space. These processes are "lightweight"—they share the same address space and thus the cost of a process context switch, process creation, or process termination is small, say, on the order of a few procedure calls [20]. 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 performance of parallel programs on a shared virtual memory system depends mainly on two things: the number of parallel processes and the degree of data sharing (i.e. contention). Theoretically, performance improves as the number of parallel processes increases and contention decreases. Contention is less if a program exhibits *locality of references*. One of the main justifications for the traditional virtual memory is that memory references in sequential programs generally exhibit a high degree of locality [10,12]. 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.

Prototype Implementation

In order to answer the question of whether it is practical to build a shared virtual memory on a loosely-coupled multiprocessor and whether most parallel application programs will get speedup on such a system, a user-mode prototype system has been implemented on the Apollo Domain [1,21], an integrated system of personal workstations and server computers connected by a 12M bit/sec baseband, single token ring network. IVY is implemented on top of the modified operating system Aegis of the Domain environment. The implementation is not particularly efficient but simple and tractable.

IVY consists of 5 modules, namely, remote operation, memory mapping, process management, memory allocation, and initialization. The hierarchy of the system is shown in Figure 2. The three top modules in the hierarchy form the IVY client in-

terface. Each consists of a set of primitives that can be used by application programs.

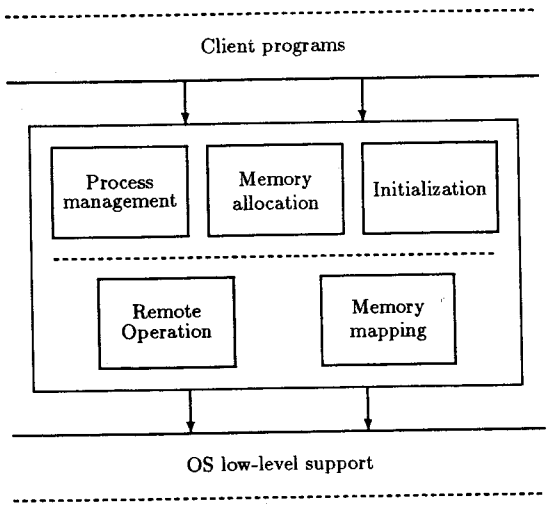

Figure 2: IVY hierarchy.

Shared Virtual Memory Mapping

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. The memory coherence problem is similar to that encountered in cache and multicache designs for shared memories on multiprocessors (see [27,2] for a survey), but most memory coherence techniques for multicaches do not apply, because a loosely-coupled multiprocessor has no physically shared memory and the communication cost between processors is non-trivial. [23] gives a detailed description and analysis of the algorithms for memory coherence.

Since memory coherence memory coherence of a shared virtual memory is maintained at page level, it is important to choose the right page size. On a stock loosely-coupled multiprocessor, one has to use a page size which is consistent with or the multiple of that provided in a Memory Management Unit (MMU) in order to use its protection mechanisms to detect incoherent memory references and trap them to appropriate fault handlers. These page fault handlers and their servers implement memory coherence strategy that keeps the memory space coherent at all times. Since sending large packets of data (say 1,000 bytes) in a loosely-coupled multiprocessor is not much more expensive than sending small ones (say 100 bytes) [28], relatively large page sizes are possible in a shared virtual memory. On the other hand, the larger the memory unit, the greater the chance for contention. The possibility of contention indicates the need for relatively small page sizes. Our experience with a page size of 1K bytes has been pleasant and we expect that smaller page sizes (perhaps as low as 256 bytes) will 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 IVY, each user address space is divided into two portions. The shared virtual memory address space is in the high portion and the private memory is in the low portion. For simplicity, the data structure of the page table is a vector of records and each record is a table entry. The whole table is stored in the private memory.

The memory coherence strategies implemented IVY use *invalidation* approach. In this approach, all read-only copies of a page are invalidated (changed to nil access) before a processor writes to a page. For experimental purposes, we implemented three algorithms: the improved centralized manager algorithm, the fixed distributed manager algorithm, and the dynamic distributed manager algorithm. These algorithms and other algorithms for solving the memory coherence problem have been studied in depth [23]. Briefly, The *centralized manager* algorithm is similar to the cache coherence solution [6]. The centralized manager resides on a single processor, and maintains all ownership information. When having a page fault, a processor will ask the manager for the copy of the page. The manager will then ask the owner of the page to send a copy to the requesting processor.

The *fixed distributed manager* algorithm gives every processor a predetermined set of pages to manage. 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 [2] is similar). 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.

The *dynamic distributed manager algorithm* keeps track of the ownership of all pages in each processor's local page table, using a field called *probOwner* in each page entry. The value of this field 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 will at least provide 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. As the system runs, each processor uses the *probOwner* field to keep track of the last change of the ownership of a page. This field is updated whenever a processor receives an invalidation request, relinquishes ownership of the page, or forwards a page fault request.

The fixed distributed manager algorithm, the dynamic distributed manager algorithm, and their variations are more appropriate than others.

Process and Process Scheduling

The process management module implements all the operations for process control, process migration, and process synchronization. The module provides clients with a set of calls for writing parallel programs.

All the processes in IVY are lightweight. The program code of a process is stored in its private memory; therefore, IVY need not have its own loader. The stack of a process is allocated from the shared memory portion. Each process has a process control

block (PCB) that contains necessary information like process state, stack, context, and other process control-dependent information. The PCBs are stored in the private memory of the address space. Therefore, the PID of a process is represented as a pair—processor number and the address of its PCB.

The process scheduling mechanism is designed to be simple. Each processor has a local ready queue using a last-in-first-out policy, that is, processes do not have priorities. The process dispatcher always picks up the process in the front of the ready queue. If there is no ready process available, the dispatcher runs a system process called the null process.

The null process implements a passive load balancing algorithm. It normally waits on two low level eventcounts, one for timeout and another for new ready processes. The null process is invoked when either of them is advanced. When a timeout event occurs, the null process will run the passive load balancing algorithm. The main idea of the algorithm is to let each processor ask for work when it is idle using some hints. The eventcount for new ready processes can be advanced only when a process is migrated to the current processor, a remote resume operation is performed, or a remote notification operation results in waking up a process. Of course, when a new ready process is available, the null process will suspend itself. The dispatcher will then do another schedule.

The hint information about the number of ready processes is important for minimizing the number of rejections of migration requests. The processors in IVY keep each other up to date on their current work loads by adding a few extra bits to the messages transmitted for remote operations. Usually, a byte will be enough to transfer the information. This byte can be packed into every message at almost no extra cost.

Experiments with many parallel application programs show that the algorithm will not work well if the number of ready processes on each processor is used as the only criterion for migrating processes. A better way is to use the number of processes (including both ready and suspended) controlled by thresholds [22]. When such a number is less than the lower threshold, the processor will try to ask for work. When such a number is greater than the upper threshold, the processor will migrate processes to other processors upon requests.

Process migration

A process in IVY is either migratable or non-migratable, indicated by a field in its PCB. Clients can modify the field by using a primitive so that a migratable process can become non-migratable or vice versa at run time. Only a ready, migratable process can migrate from one processor to another. When a process is migrated, a forwarding pointer is put into its PCB and the migrated attribute is set. The PCBs of migrated processes are used for storing forwarding pointers. The collection of non-reachable PCB's has not been implemented in IVY.

Since PCBs are stored in the private memory portion of the address space, a process migration must

- send the PCB of the process to the destination processor and put it into a PCB,
- copy the current page of the process's stack to the destination processor and transfer the ownership of the page,
- transfer the ownership of all the pages in the upper portion of the stack to the destination processor, and
- put the PCB in the ready queue on the destination processor.

The reason for moving the current page of the process's stack is to avoid a page fault in the process dispatcher (Figure 3).

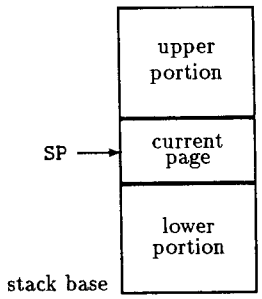

Figure 3: A process stack.

The upper portion of the stack need not move to the destination processor because its content is meaningless. Ownership transfer is inexpensive because it only requires setting the protection bits of the page frames. There is no need to do anything with the lower portion of the stack because the stack can grow without having further page faults after the current page and the upper portion of the stack become writable.

Eventcount Implementation

In a shared virtual memory system, it is possible to implement a process synchronization mechanism based on either global memory or message passing. Eventcount [26] is the process synchronization mechanism in IVY. The main reason for choosing eventcount is that the Aegis operating system uses eventcounts as its synchronization mechanism.

An eventcount synchronization mechanism has four primitive operations:

- `Init( ec )` — initializes an eventcount.
- `Read( ec )` — returns the value of the eventcount.
- `Await( ec, value )` — suspends the calling process itself until the value of the eventcount reaches the value specified.
- `Advance( ec )` — increments the value of the eventcount by one and wakes up awaiting processes.

After an eventcount is initialized, any process can use it without knowing where it resides.

The implementation of these primitives is based on shared virtual memory. The atomic operation is implemented by pinning memory pages and using test-and-set instructions. This implementation is much cleaner than that based on message-passing; furthermore, the performance is better when there is more than one process on each processor because eventcount primitives become local operations when the eventcount data structure has been paged into the local processor.

The data structures of an eventcount usually reside together in one page. The shared virtual memory mapping mechanism can move this page on demand when an eventcount operation is performed and on a processor where there is no such eventcount data structures. If the data structures of an eventcount require more than one page, then the additional pages will be linked together. This mechanism increases the locality of the eventcount

data structure. In most cases, only one page is needed for each eventcount.

Memory Allocation

IVY has a simple memory allocation module that uses a "first fit" algorithm with one-level centralized control. The processor with which the user directly contacts will be appointed to the centralized memory manager. To reduce the memory contention, the memory allocator allocates each piece of memory to the boundary of a page.

Both allocate and free are atomic operations. IVY uses a binary lock on each processor for memory allocation purposes. At the beginning of each memory management primitive, a test-and-set operation is performed on the lock. A failed process will be put into a queue and will be awakened by an unlock operation on the lock which is done at every end of each primitive.

A more efficient approach is two-level memory management. In this approach, each processor has a local allocator maintaining a big chunk of memory allocated from the central memory allocator. This big chunk of memory serves for the local memory allocations. When there is not enough free memory left in the big chunk, the local allocator will allocate another big chunk from the central allocator. This approach has not been implemented yet, though it is expected to have better performance.

Remote Operation

The remote operation module implements a remote request/reply mechanism that handles all the remote operations of other modules. Such a mechanism (also called simple RPC) is similar to remote procedure call facility [24,3], but it is simpler than the general one and has a few special features for implementing shared virtual memory system.

One of such features is broadcast or multicast remote operation mechanism. A broadcast or a multicast request has three reply schemes: a reply from any receiving processor, replies from all receiving processors, and no reply at all. The first option is useful for broadcasting page fault requests to locate page owners (see [23]). The second option can be used for implementing invalidation operations. The third option is for broadcasting approximate information for process scheduling.

Another feature is a forwarding request mechanism that allows a processor to forward a request to another processor. For example, processor 1 can send a request to processor 2, processor 2 forwards the request to processor 3, and so on until processor k performs the operation and sends a reply back to processor 1. There are no intermediate replies involved in the operation. This mechanism is particularly useful for implementing the dynamic distributed manager algorithm.

The retransmission protocol is based on the philosophy of resending replies only when necessary. Such a design is based on two assumptions: local computation is always correct, and communication may be unreliable, but once a packet is received, its content is always correct. The protocol is reliable only when these assumptions hold. In practice, the assumptions are reasonable. Retransmission checking is done in a null process, which checks all the outgoing channels every half second when there is nothing to do.

Programming in the IVY Environment

Programmers can use any programming language in the Apollo DOMAIN to write parallel programs as long as they can interact with the procedure calls in the Apollo DOMAIN Pascal in which IVY is implemented. Since all the languages in the Apollo DOMAIN are designed for sequential programming, the programmer has to program parallel constructs explicitly with the primitives provided by IVY.

Programmers or compilers using IVY need to decide which piece of data puts into shared virtual memory and which into private memory. Programs later do not need to know where the shared data structures are in the sense that references to these data structures are the same as to other data structures. If IVY had its own loader, explicit memory allocation would not be necessary.

Clients can use primitives provided by the process management module to create lightweight processes (or threads) for a parallel computation. The programmer can choose how to schedule processes when calling an initialization procedure at the beginning of the program. There are two options: manual scheduling and system scheduling. If system scheduling is used, the programmer only needs to create and terminate processes. But if manual scheduling is chosen, the programmer needs to tell where and when a process goes. It is the programmer's responsibility to program process synchronization. The methodology of such programming is the same as that of "conventional" concurrent programming developed since the 1960s. Although there is no parallel programming language, such a primitive environment has proven to be convenient enough to write benchmark programs.

IVY does not have any special debugging tools. Initial debugging programs can be done on a single processor. Since an IVY image file can run on any number of processors, there is no need to have a simulator. If a program follows IVY parallel programming conventions, debugging on a single processor is usually easy. After debugging on a single processor, the programmer should debug on two and then three processors. My experience indicates that if a program can run on three processors correctly, there are few bugs left.

Experiments

Given the difficulties of finding practical parallel programs, the only reasonable way to do experiments is to select a set of application programs from different fields as a benchmark suite. All benchmarks have the following two properties:

- *reasonably fine granularity of parallelism, and*
- *side-effects in shared data structures.*

Parallel programs with rather coarse granularity can obviously perform well in the shared virtual memory system. There are parallel functional programs that do not have any side-effects in their data structures at run time. The shared virtual memory system is clearly a big win in these applications. The main goal in using the two criteria is to avoid weighing the experiments in favor of the shared virtual memory system by picking problems that suit the system well. The benchmark set in the experiments consists of six parallel programs that are written in Pascal. All of them are transformed manually from sequential algorithms into parallel ones in a straightforward way.

Linear Equation Solver This program implements a parallel Jacobi algorithm for solving linear equations. The algorithm

is transformed from the traditional, sequential Jacobi algorithm that solves the linear equation $Ax = b$ where A is an n by n matrix. In each iteration, $x^{(k+1)}$ is obtained by

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

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 eventcount. The data structures A, x, and b are stored linearly in the shared virtual memory, and the processes access them freely without regard to their location.

3D PDE Solver This program solves three dimensional partial differential equations (PDEs) using a parallel Jacobi algorithm. The algorithm and its transformation are similar to the linear equation solver except that in the equation $Ax = b$, A is a sparse matrix. Since this matrix is never updated in the program, the practical PDE solvers in scientific computing usually eliminate the matrix by coding it into programs to save space and time. In practice, matrix A is large and it is read-only, coding it into program will not be in favor of the shared virtual memory performance. To be more realistic, we choose to do so. The vectors x and b are stored linearly in the shared virtual memory.

Traveling Salesman Problem The traveling salesman problem is to find a tour that visits each city once with the minimum cost. The cities are represented as the nodes in an undirected graph. The cost of a tour is the sum of the weights of the edges on the tour. The algorithm used in the program is a simplified version of the branch-and-bound approach proposed in [13]. At each step, an 1-tree (a variation of the minimum spanning tree) of the remaining graph is computed. The sum of the cost of the subtour and the 1-tree is compared with the cost of the current least upper bound. If the cost is less than the upper bound, it will replace the upper bound and the subtour is still valid; otherwise, the subtour will be thrown away. The available branches, the graph, and the least upper bound are stored in the shared virtual memory. The program creates a process for each processor that performs the branch-and-bound algorithm on a branch obtained from the shared virtual memory. These processes run in parallel until the tour is found. Each process is not much different from the sequential one except it needs to access shared data structures mutual exclusively.

Matrix Multiply This program 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 columns of matrix B. 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 they will be paged to other processors on demand.

Dot-product The dot-product program computes

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

A number of processes are created to partition the problem. Each process computes a partial sum and S is obtained by summing up the partial sums produced by the individual processes. Both vector 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.

Split-merge Sort This program implements 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 [17]. 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 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. 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, any program does its best for any given number of processors. Unlike message-passing systems or primitive global-space systems, IVY has almost no extra overhead when programs run on a single processor. The only additional costs are in creating processes, which takes milliseconds in total, and mutual exclusion, which takes two 68000 instructions for each locking. Since there are few locking operations in the programs above, the programs using one processor run just as fast as their sequential programs.

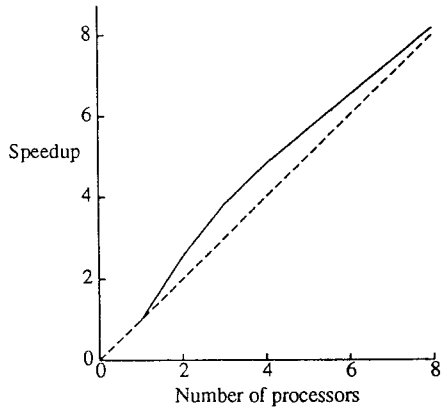

Figure 4: Super-linear speedup

The 3D PDE program, when matrix A is 50^3 by 50^3, experienced super-linear speedup as shown in Figure 4. At first glance, the 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. Since the algorithm in both programs is a straightforward transformation from the sequential Jacobi algorithm and all the processes are synchronized at each iteration, the algorithm cannot yield super-linear speedup. The reason is that the fundamental law of parallel computation assumes that every processor has an infinitely large memory, which is not true in practice. For instance, in the parallel 3-D PDE example, 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.

Table 1 shows the total number of disk I/O page transfers of the first six iterations when the 3D PDE program runs on one processor and two processors. Obviously, the number of the disk I/O page transfers on two processors is substantially less than that on one processor. In the two-processor case, the program initializes its data structures only on one processor, this processor causes most disk I/O transfers because it cannot hold all the data structures in its physical memory. As the program runs, the processes start to access some portions of the data structures, causing the shared virtual memory page faults to move pages from one processor to another. When the shared virtual memory distributes the data structure into individual physical memories whose cumulative size is large enough, few disk I/O data movements will occur. On the other hand, IVY is a user-mode system implemented on top of the Aegis virtual memory system which performs an approximate LRU page replacement strategy; the pages recently moved from the processor with initialized data structures may not be replaced because these pages are also most recently referenced ones. This explains why the number of disk I/O page transfers in the two-processor case decreases gradually.

	Disk page transfers of each iteration					
	1	2	3	4	5	6
1 processor	899	1600	1543	1515	1542	1540
2 processors	1432	1072	466	156	101	105

Table 1: Disk page transfers

When the data structure of the problem is not larger than the physical memory on a processor (matrix A is 40^3 by 40^3), the result of the 3D PDE is no longer super-linear, as shown in Figure 5. They are similar to what we see in the past. For example, the result is similar to that generated by similar experiments on CM*, a shared memory multiprocessor [18,9]. 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 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. While the shared virtual memory system pays the cost of local memory references, CM* pays the cost of remote memory references across its Kmaps.

The dot-product program did not perform well on IVY, as indicated in Figure 5. It is included here so as not to paint too bright a picture. Since this program only references each element once, the ratio of the communication cost to the computation cost in this program is large. For programs like dot-product, it is not appropriate to use a shared virtual memory system, unless the communication cost can be reduced.

Matrix multiply and traveling salesman problem perform well on IVY system. They show the good side of the shared virtual memory system. Both programs exhibit a high degree of localized computation. Since the algorithm used in the traveling salesman problem is a parallel branch-and-bound, there are

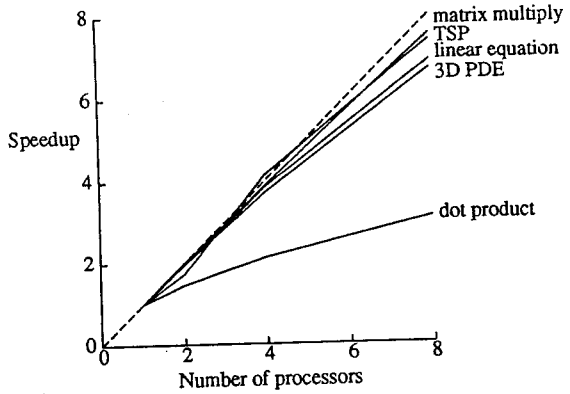

Figure 5: Speedups

anomalies [19]. It is possible that the program gets super-linear speedup or no speedup at all. In this example, it happens to have super-linear speedup.

Figure 6 shows the speedup of merge-split sort program. The curve does not look very good because even with no communication costs, the algorithm does not yield linear speedup. The program uses the best strategy for any given number of processors. For example, there is one merge-split sorting when running the program on one processor, there are 4 blocks when running the program on two processors, and so on. Using a fixed number of blocks for any number of processors would result in a better speedup, but such an approach is not reasonable.

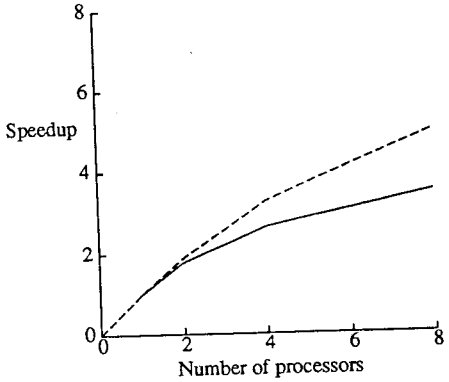

Figure 6: Speedup of merge-split sort

Conclusion

The difficulties with passing complex data structures and process migration are the main drawbacks of the message passing model for parallel computing. Shared virtual memory on loosely-coupled multiprocessors can solve these problems. The success of implementing the prototype shared virtual memory system IVY and the experiments show that it is practical to implement such a system on existing loosely-coupled multiprocessors such as local area networks.

The implementation experience shows that, although it is possible to implement a shared virtual memory without modi-

fying an existing system like the Aegis operating system, it is necessary to modify the existing system to get acceptable performance. IVY is a user-mode implementation, so it has a lot of overhead. A system-mode implementation ought to provide a substantial improvement. It is expected that a well-tuned system-mode implementation should improve the performance of remote operations and page moving by a factor of at least two according to the performance comparison with some well tuned systems [28,7]. I/O overlaps among the lightweight processes do not exist in IVY. An integrated heavyweight and lightweight process scheduler is highly desirable. The disk I/O overlap may also greatly improve IVY's performance.

The experimental results of running some non-trivial parallel programs on the prototype system strongly support the idea of shared virtual memory on loosely-coupled multiprocessors. The results demonstrate that the shared virtual memory can effectively exploit not only the available processors but also the combined physical memories of a multiprocessor system.

The experimental results reported in this paper are limited because there were only up to eight processors available for running the modified Aegis operating system at the time. Experiments on more processors will show more insights of shared virtual memory and behaviors of parallel programs. To answer many unanswered questions, we plan to perform more experiments on a shared virtual memory system being implemented on a large-scale multiprocessor at Princeton.

Acknowledgement

I wish to thank John Ellis for his invaluable suggestions and helpful discussions and Nat Mishkin for his help with understanding the Aegis kernel, which made my modifications to the OS possible. I would like to thank Paul Hudak and Alan Perlis for their continual help inspiration. I also wish to thank Jeff Naughton and the referees for their helpful comments.

References

[1] Apollo. *Apollo DOMAIN Architecture.* Apollo Computer Inc., Chelmsford, Mass., 1981.

[2] J. Archibald and J. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[3] A.D. Birrell and B.J. Nelson. *Implementing Remote Procedure Calls.* Technical Report CSL-83-7, Xerox PARC, December 1983.

[4] D. Bitton, D.J. DeWitt, D.K. Hsaio, and J. Menon. A Taxonomy of Parallel Sorting. *ACM Computing Surveys*, 16(3):287–318, September 1984.

[5] N. Carriero and D. Gelernter. The S/Net's Linda Kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.

[6] L.M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[7] David R. Cheriton. The V kernel: A Software Base for Distributed Systems. *IEEE Software*, 1(2):19–43, 1984.

[8] D.R. Cheriton and M. Stumm. The Multi-Satellite Star: Structuring Parallel Computations for A Workstation Cluster. To appear, 1988.

[9] Jarek Deminet. Experience with Multiprocessor Algorithms. *IEEE Transactions on Computers*, C-31(4), April 1982.

[10] Peter J. Denning. On Modeling Program Behavior. In *Proceedings of Spring Joint Computer Conference*, pages 937–944, AFIPS Press, 1972.

[11] Peter J. Denning. Virtual Memory. *ACM Computing Surveys*, 2(3):153–189, September 1970.

[12] Peter J. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[13] M. Heid and R.M. Karp. The Traveling-salesman Problem and Minimum Spanning Trees. *Operation Research*, 17(12):1139–1167, December 1970.

[14] M. Herlihy and B. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[15] Carl Hewitt. The Apiary Network Architecture for Knowledgeable Systems. In *Proceedings of the Lisp Conference*, pages 107–117, August 1980.

[16] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(11):666–677, August 1978.

[17] C.A.R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[18] A. K. Jones and P. Schwarz. Experience Using Multiprocessor Systems — A Status Report. *ACM Computing Surveys*, 12(2), June 1980.

[19] T. Lai and S. Sahni. Anomalies in Parallel Branch-and-Bound Algorithms. *Communications of the ACM*, 27(6):594–602, June 1984.

[20] B. M. Lampson and D. D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[21] P.J. Leach, P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson, and B.L. Stumpf. The Architecture of an Integrated Local Network. *IEEE Journal on Selected Areas in Communications*, 1983.

[22] Kai Li. *Shared Virtual Memory on Loosely-coupled Multiprocessors.* PhD thesis, Yale University, October 1986. Tech Report YALEU-RR-492.

[23] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986. A journal version will appear in *ACM Transactions on Computer Systems.*

[24] Bruce J. Nelson. *Remote Procedure Call.* PhD thesis, Carnegie-Mellon University, May 1981.

[25] M.L. Powell and B.P. Miller. Process Migration in DEMOS/MP. In *Proceedings of the ninth Symposium on Operating Systems Principles*, pages 110–119, 1983.

[26] David P. Reed and Rajendra K. Kanodia. Synchronization with Eventcounts and Sequencers. *Communications of the ACM*, 22(2):115–123, February 1979.

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

[28] Alfred Z. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM*, 25(4):260–273, April 1982.