SUPPORTING A COHERENT SHARED ADDRESS SPACE ACROSS SMP NODES: AN APPLICATION-DRIVEN INVESTIGATION

ANGELOS BILAS, LIVIU IFTODE, RUDRAJIT SAMANTA AND
JASWINDER PAL SINGH

DEPARTMENT OF COMPUTER SCIENCE
35 OLDEN STREET
PRINCETON UNIVERSITY
PRINCETON, NJ 08544

{BILAS, LIV, RUDRO, JPS}@CS.PRINCETON.EDU

Abstract.
As the workstation market moves form single processor to small-scale shared memory multiprocessors, it is very attractive to construct larger-scale multiprocessors by connecting symmetric multiprocessors (SMPs) with efficient commodity network interfaces such as Myrinet. With hardware-supported cache-coherent shared memory within the SMPs, the question is what programming model to support across SMPs. A coherent shared address space has been found to be attractive for a wide range of applications, and shared virtual memory (SVM) protocols have been developed to provide this model in software at page granularity across uniprocessor nodes. It is therefore attractive to extend SVM protocols to efficiently incorporate SMP nodes, instead of using a hybrid programming model with a shared address space within SMP nodes and explicit message passing across them. The protocols should be optimized to exploit the efficient hardware sharing within an SMP as much as possible, and invoke the less efficient software protocol across nodes as infrequently as possible.

We present a *home-based* SVM protocol that was designed with these goals in mind. We then use detailed, application-driven simulations to understand how successful such a protocol might be and particularly whether and to what extent the use of SMP nodes improves performance over the traditional method of using SVM across uniprocessor nodes. We examine cases where the home-based SVM protocol across nodes is supported entirely in software, and where the propagation of modifications to the home is supported at fine grain in hardware. We analyze how the characteristics of our ten applications and their algorithms interact with the use of SMP nodes, to see what classes of applications do and do not benefit from SMP nodes, and determine the major bottlenecks that stand in the way of improved performance.

1. Introduction. Small-scale, shared-memory SMPs (symmetric multiprocessors) have become increasingly widespread. Inexpensive SMPs based on Intel PC processors are on the market, and SMPs from other vendors are increasingly popular. Given this development, it is very attractive to build larger multiprocessors by putting together SMP nodes rather than uniprocessor nodes. Commodity network interfaces and networks have progressed to the point where relatively low latency and high bandwidth are achievable, making such clusters of SMPs all the more attractive.

The question is what programming model to use across nodes. The choices are to extend the coherent shared address space abstraction that is available within the nodes, or to use a shared address space within nodes and explicit message passing between nodes, or to use explicit message passing everywhere by using the hardware-supported shared memory within a node only to accelerate message passing, not to share data among processors. A coherent shared address space has been found to be an attractive programming model: It offers substantial ease of programming advantages over message passing for a wide range of applications—especially as applications become increasingly complex and irregular as we try to solve more realistic problems—and it has also been shown to deliver very good performance when supported in hardware in tightly coupled multiprocessors, at least up to the 64-128 processor scale where experiments have been performed. It is also the programming model of choice for small-scale multiprocessors (especially the SMP nodes), so provides a graceful migration path. The last of the programming model possibilities (message passing everywhere) does not take full advantage of hardware coherence within the SMP, and the second one provides an awkward hybrid model that is unattractive to programmers.

Unfortunately, commodity SMP nodes and networks do not provide hardware support for a coherent shared address space *across* nodes. However, shared virtual memory (SVM) protocols have been developed that provide a shared address space model in software at page granularity across *uniprocessor* nodes by leveraging the support provided in microprocessors for virtual memory management. Relaxed memory consistency models are used to reduce the frequency of invocation of the expensive software protocol operations [20]. Much research has been done in this area, and many good protocols developed. One way to provide the programming model of choice in clusters, then, is to extend these SVM protocols to use multiprocessor (SMP) rather than uniprocessor nodes. Another view of this approach is that the less efficient SVM is used not as the basic mechanism with which to build multiprocessors out of uniprocessors, but as a mechanism to extend available small-scale machines to build larger machines while

preserving the same desirable programming abstraction. The key is to use the hardware coherence support available within the SMP nodes as much as possible, and resort to the more costly SVM protocol across nodes only when necessary. If successful, this approach can make a coherent shared address space a viable programming model for both tightly-coupled multiprocessors (using hardware cache coherence) and loosely coupled clusters.

A recent and particularly promising form of SVM protocols is the class of so-called *home-based* protocols. This paper describes a protocol for home-based SVM across SMP nodes that accomplishes the goal above, and that we have implemented both in simulation and on a set of eight Pentium Pro Quad SMPs connected by a Myrinet network. The SVM protocol can operate completely in software, or can exploit hardware support for *automatic update* propagation of writes to remote memories as supported in the SHRIMP multicomputer [4] (and in a different way in the DEC Memory Channel [13]). Having described the protocol, we use detailed simulation to examine how using k, c-processor SMPs connected this way compares in performance to using SVM across $k * c$ uniprocessor nodes, and whether the performance characteristics look promising overall. Clustering processors together using a faster and finer-grained communication mechanism has some obvious advantages; namely prefetching, cache-to-cache sharing, and overlapping working sets [11]. The hope is that for many applications a significant fraction of the interprocessor communication may be contained within each SMP node. This reduces the amount of expensive (high latency and overhead) cross-node SVM communication needed during the application's execution. However, it unfortunately *increases* the bandwidth (i.e. communication per unit time) demands on the node-to-network interface. This is because the combined computation power within the SMP node typically increases much faster with cluster size c (linearly) than the degree to which the per processor cross-node communication volume is reduced. This means that depending on the constants, the node-to-network bandwidth may become a bottleneck if it is not increased considerably when going from a uniprocessor to an SMP node.

We explore these issues with both the all-software and automatic update hardware-supported home-based protocols. Our study is application driven. In particular, we examine how the algorithms and data structures of ten very different types of applications interact with the clustering and the page-based SVM nature of the protocol, to see what classes of applications do and do not benefit from SMP nodes. We find the performance of both protocols improves substantially with the use of SMP rather than uniprocessor nodes in five of the ten applications. In three applications there is a smaller improvement (or they perform the same as in the uniprocessor node case) and for the other two results differ across all-software and automatic update protocols, with the latter performing worse with SMPs than with uniprocessors.

The major advantages and disadvantages of a shared address space programming abstraction compared to explicit message passing are described in [8] (Chapter 3) and [25] and will not be covered here . Section 2 introduces the uniprocessor home-based protocols and describes the extensions to use SMP nodes. It identifies many of the tradeoffs that arise in designing such a protocol, and the positions that our protocol takes along them. Section 4 describes the detailed architectural simulator we use, and Section 5 measures the basic performance characteristics of the simulated system using a set of microbenchmarks. The next two sections are focused on methodological issues. Section 6 briefly describes the most relevant characteristics of the applications and algorithms used, and Section 7 provides an overview of the metrics we use and the way in which we present performance results. Section 8 provides for both SMP and uniprocessor nodes for the all-software and hardware-supported protocols. Detailed breakdowns of execution time are used to understand the results in light of application characteristics. Finally, Section 9 describes some related work, and Section 10 summarizes the main conclusions of the paper.

2. SVM Protocols. Shared virtual memory is a method of providing coherent replication in a shared address space across uniprocessor nodes without specialized hardware support beyond that already available in uniprocessors. The idea is to provide the replication and coherence in main memory through the virtual memory system, so main memory is managed as a cache at page granularity. The coherence protocol runs in software, and is invoked on a page fault, just as a hardware cache coherence protocol is invoked on a cache miss. The problem with page-level coherence is that it causes a lot of false sharing when two unrelated items that are accessed by different processors (and written by at least one of them) happen to fall on the same page. Since protocol operations and communication are expensive, this false sharing is particularly harmful to performance. To alleviate the effects of false sharing, protocols based on relaxed memory consistency models have been developed, which allow coherence information to be propagated only at synchronization points rather than whenever shared data are modified. This means that if one processor is repeatedly writing a word on a page and another processor is repeatedly reading another unrelated word on the same page, they can keep doing this independently until they reach synchronization points, at which time the pages are made consistent. To allow multiple writers to the same page to write their

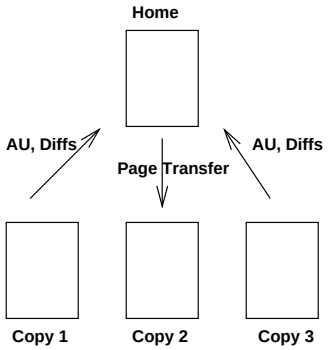

FIG. 1. *Updates are sent to the home node with a protocol specific mechanism: Diffs for* HLRC *and* AU *for* AURC. *Whole pages are fetched from the home on demand.*

separate copies independently until a synchronization point, so called multiple-writer protocols have been developed. These greatly alleviate the effects of false sharing, but communication and the propagation of coherence information are still expensive when they do occur.

The home-based protocols we examine are based on a lazy implementation of the *release consistency* model, called lazy release consistency (*LRC*). The all-software protocol is called home-based lazy release consistency (*HLRC*), and the protocol that exploits hardware automatic update support is called automatic update release consistency(*AURC*) [1]. Both protocols use timestamps to maintain ordering of events. The rest of this section first briefly discusses lazy release consistency, *HLRC* and *AURC* for uniprocessor nodes. More detailed descriptions can be found in the literature [15, 20, 30]. Then, we discuss the major design choices for extending the protocols to use SMP nodes efficiently, and the the specific choices made in our implementation.

2.1. Lazy Release Consistency. Lazy Release Consistency is a particular implementation of release consistency (RC). RC is a memory consistency model that guarantees memory consistency only at synchronization points. These are marked as acquire or release operations. In implementations of an eager variation of release consistency the updates to shared data are performed globally at each release operation. Lazy Release Consistency (*LRC*) [20] is a relaxed implementation of RC which further reduces the read-write false sharing by postponing the coherence actions from the release to the next related acquire operation. To implement this relaxation, the *LRC* protocol uses *time-stamps* to identify the time intervals delimited by synchronization operations and establish the happened-before ordering between causally-related events. To reduce the impact of write-write false sharing *LRC* has most commonly been used with a software or hardware supported multiple-writer scheme. The first software-based multiple writer scheme was used in the TreadMarks system from Rice University [19, 20]. In this scheme, every writer records any changes it makes to a shared page during each time interval. When a processor first writes a page during a new interval it saves a copy of the page, called a *twin*, before writing to it. When a release synchronization operation ends the interval, the processor compares the current (dirty) copy of the page with the (clean) twin to detect modifications and consequently records these in a structure called a *diff*. The *LRC* protocol may create diffs either eagerly at the end of each interval or on demand in a lazy manner.

On an acquire operation, the requesting processor invalidates all pages by consulting the information about updated pages received in conjunction with the lock. Consequently, the next access to an invalidated page causes a page fault. In the style of protocol used in TreadMarks, the page fault handler collects all the diffs for the page from either one or multiple writers and applies them locally in the proper causal order to reconstitute the page coherently.

2.2. Home-based *LRC* Protocols. Home-based LRC protocols are much like the protocol described above, except in how they manage the propagation of updates (writes). Instead of writers retaining their diffs and the faulting processor obtaining the diffs from all the writers upon a fault, the idea here is for writers to propagate their changes to a designated home copy of the page before a release operation. The writes from different processors are merged into the home copy, which is therefore always up to date according to the consistency model. On a page

[1] Although these are versions of the same basic home-based protocol, we will refer to them as separate protocols to ease the discussion.

fault, the faulting processor simply obtains a copy of the page from the home. As a result of fetching the whole page rather than diffs, this protocol may end up fetching a greater amount of data in some cases, but it will reduce the number of messages sent since the data have to be fetched from only one node.

The all-software implementation of home-based *LRC*, called the *HLRC* protocol, also uses software write detection and a diff-based write propagation scheme. Diffs are computed at the end of each time interval for all pages updated in that interval. Once created, diffs are eagerly transferred to the home nodes of the pages, where they are immediately applied. Therefore, diffs are transient, both at the writer nodes and at the home nodes. Writers can discard their diffs as soon as they are dispatched, greatly reducing the memory requirements of the protocol. Home nodes apply arriving diffs to the relevant pages as soon as they arrive, and immediately discard them too. Later, during a page fault, following a coherence invalidation, the faulting node fetches the correct version of a whole page from the home node.

Some recent network interfaces also provide hardware support for the propagation of writes at fine granularity (a word or a cache line, say) to a remotely mapped page of memory [4, 13]. This facility can be used to accelerate home-based protocols by eliminating the need for diffs, leading to a protocol called automatic update release consistency or *AURC* [16]. Now, when a processor writes to pages that are remotely mapped (i.e. writes to a page whose home memory is remote), these writes are automatically propagated in hardware and merged into the home page, which is thus always kept up to date. At a release, a processor simply needs to ensure that it's updates so far have been flushed to the home. At a page fault, a processor simply fetches the page from the home as before.

While the disadvantage of home-based protocols is that they may fetch more data by fetching whole pages rather than diffs on a fault, the advantages can be summarized as follows: accesses to pages on their home nodes cause no page faults even if the pages have been written to by other processors, non-home nodes can always bring their shared pages up-to-date with a single round-trip message, and protocol data and messages are much smaller than under standard *LRC*. Studies on different platforms have indicated that home-based protocols outperform outperform traditional *LRC* implementations, at least on the platform and applications tested, and also incur much smaller memory overhead [16, 30].

Having understood the basic protocol ideas, let us proceed to the main goal of this paper, to examine how and how well the protocols can be used to extend a coherent shared address space in software across SMP nodes.

3. Extending Home-based Protocols to SMP Nodes.

3.1. Protocol Design. Consider the *HLRC* protocol for simplicity. Implementing the *HLRC* protocol on SMPs requires several non-trivial changes due to the interactions of hardware-coherent intra-node shared memory with the software-coherent inter-node sharing. In this section we discuss some of the critical issues related to the efficiency of an SVM implementation for SMPs. Even simple operations such as a full page fetch from the home present complications. For instance, if there are other processes on this node writing to the page being fetched then this full page fetch will overwrite their updates, causing them to be lost forever. Details such as this one will not be discussed were but were challenging issues during the implementation.

3.1.1. Shared-nothing model. The uniprocessor implementations can be ported to work with SMP nodes with virtually no modifications, if the protocol treats each processor as if it were a separate node. The processors do not share any application or protocol data and the hardware shared memory in a node is used merely as a fast communication layer. However, such a model does not leverage the cache-coherent shared memory provided within the SMP.

3.1.2. Shared-everything model. At the other extreme we consider a model where all the processors within a given node share both the application data and all the data structures used by the SVM system. In such a model the node would appear to contain a single processor to the outside world. When coherence actions are performed they apply to all the processors within the node. For example when a processor acquires a lock from a remote node the page invalidations are performed for all the processors in this node. This is of course conservative, since the other processors in the node do not need to see these invalidations yet according to the consistency model. However, acquires within a node (local acquires) will require almost no protocol overhead, since the updates performed locally will be made available by the intra-node hardware cache-coherence. Since all the processes within a node always have the same state for any given page, we can use a single page table for all the processes. This is akin to the thread model of computation within the node. The propagation of diffs occurs at barrier synchronization and remote lock acquires. They also occur during a lock release if there is an outstanding remote request for this lock. As a result, lock releases are very cheap, except when there is a remote lock request waiting for this lock.

Unlike the previous model, this one does utilize the hardware cache-coherence to share application data within the SMP and also to share a number of data structures required by the SVM system itself. However, propagating invalidations unnecessarily to all processors in an SMP node can degrade performance significantly. In particular, the effects of page-level false sharing can be large in this *eager invalidation* scheme resulting in a large number of page faults and page fetches.

3.1.3. A hybrid model: Lazy invalidations. The shared-everything model utilizes the SMP hardware as much as possible, but it is the shared-nothing model in which coherence information is propagated only when absolutely necessary (i.e. as lazily as possible). To provide both these desirable features, we propose and implement a scheme with *lazy invalidations*. In this scheme all processors within a node share all the application data and a number of data structures used by the system. However, each process has its own page-table, and a given page in the system may have different states for different processes. Now, during a remote lock acquire, invalidations are performed *only* for the acquiring process (this will help to make the acquire faster). However a local lock acquire will now require invalidations to be performed (hence, it will be more expensive than the shared-everything scheme, but we must design it to be much less expensive than the shared-nothing scheme). We can see how the coherence actions are performed at different times by comparing Figures 2 and 3. The figures assume that all the processors in the node will acquire the lock; when this is not the case, the eager invalidation scheme will prove to be more expensive.

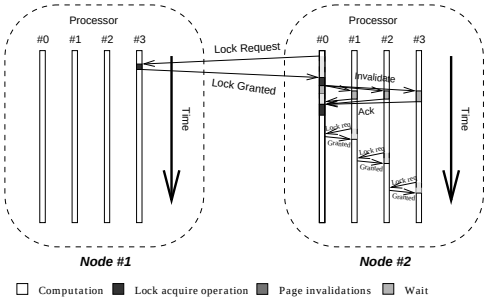

FIG. 2. *Eager invalidation scheme*

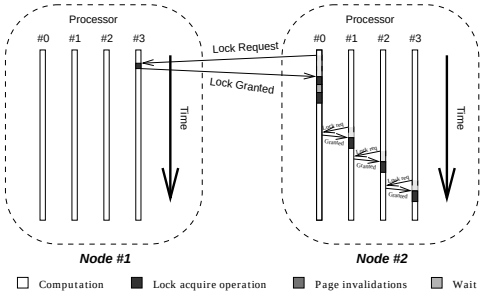

FIG. 3. *Lazy invalidation scheme*

Similarly to the previous scheme, we perform diffs during barriers and remote lock requests, as well as lock releases when there is an outstanding remote request for this lock. Barrier operations are almost identical in this and the shared-everything schemes.

3.1.4. Translation–Lookaside Buffer (TLB) Coherence. Previous studies have discussed TLB synchronization as a major obstacle for an SVM implementation to achieve good performance on a cluster of SMPs. TLB synchronization or TLB shootdown are terms used for the global operation (within an SMP) of flushing the TLB of all the processors of the same SMP.

In the eager invalidation scheme, all the processes share one page table. When one processor performs a change to the page table, all others in this node should see this change, hence we need to flush all their TLBs to ensure consistency. However, in the lazy invalidation scheme, each process has it's own page table and hence such

synchronization is not necessary. Of course, the TLB for this processor needs to be flushed. Another relevant issue is process migration. Since the Pentium family of processors does not support entries of multiple processes in the same TLB, TLBs are flushed on every context-switch. Hence, process migration does not pose a problem on this architecture.

3.1.5. Synchronization. Barrier synchronization in SVM systems is usually implemented (in the absence of hardware support) with messages that are sent by each processor to a barrier master The barrier master, gathers the control information and distributes it to all the nodes after they have reached the barrier. The number of messages exchanged depends on the algorithm used. Barriers may create hot spots in the network if they are not implemented carefully. In an SMP configuration, two–level hierarchical barriers not only reduce hot spots, but reduce the number of messages exchanged as well. The lower level is concerned with intra-node synchronization that does not involve any messages at all, and the higher level with inter-node synchronization, which is achieved by exchanging messages. Hierarchical barriers in an SMP configuration match the underlying architecture well.

An important tradeoff in barrier implementation is the amount of processing needed at the barrier master. The gathered control information (invalidations and time stamps) can either be processed locally in the barrier master first and then sent to each node only if necessary, or it can be sent to all nodes and then the appropriate information is extracted in each node. The first approach reduces the size of messages that are sent but turns the barrier master into a serialization point. The second approach uses bigger messages but exhibits higher parallelism.

Similarly, locks within an SMP node need not exchange messages. This makes local lock acquires very cheap. Depending on the invalidation scheme used (as discussed above) local lock acquires can be as cheap as a few memory references.

3.1.6. Protocol handling. In all SVM implementations remote requests sent over a network need to be serviced. On a uniprocessor node there is little choice in this regard; the processor must be interrupted or it must somehow poll periodically. However, with SMP nodes there are a number of choices. The two basic ideas are either to dedicate a processor within the SMP to handle network requests exclusively (by polling) or to handle the requests by interrupting one of the computation processors.

A dedicated processor implementation helps to avoid interrupts, which are a performance bottleneck in most systems. However, this choice wastes a valuable compute resource. In our experiments we notice that this dedicated processor has low occupancy, since actual protocol processing overhead even with SMP nodes, is still not very high.

If we use the compute processors to handle requests then we could either statically assign one compute processor for this purpose or we could perform a round-robin assignment as the requests arrive. To reduce interrupts we can instruct idle compute processors to poll for requests and interrupt a random compute processor only when there are no idle processors [18].

It is interesting to note that each solution presented for protocol handling is expected to perform better but is more complex than the previous one. On a real system some choices may be difficult or too expensive to perform due to architectural and operating system limitations. For instance, Linux 2.0.x (the OS we use) sends interrupts only to processor 0, and it is not possible to distribute interrupts among processors within an SMP. We therefore use the method of a statically assigned compute processor for protocol handling.

3.1.7. Protocol optimizations. Another important issue is how each protocol interacts with the system on which it is implemented. Several system aspects can influence performance substantially and change the tradeoffs in protocol design. These include various architectural and operating system costs, e.g., interrupts, network latency and bandwidth, etc. When a protocol is designed for a specific system, these issues need be taken into account.

3.2. Protocol Implementation. This section presents the specific choices made in our implementation of *HLRC* across SMP nodes. The data structures that allow us to implement these choices easily, and the exact mechanisms used will also be described. Readers not interested in implementation issues may skip the details in this section.

3.2.1. Operations and Data structures. To illustrate the data structures, we first need to define some key terms. The time during the actual execution of a parallel program is broken into *intervals*. With uniprocessor nodes, intervals are maintained on a per-process basis. An interval is the time between two consecutive releases by the process. These intervals are numbered in a monotonically increasing sequence. Each process maintains a vector called the *update-list*, which records all the pages that have been modified (by this process) in the current interval. Intervals are ended when locks are released.

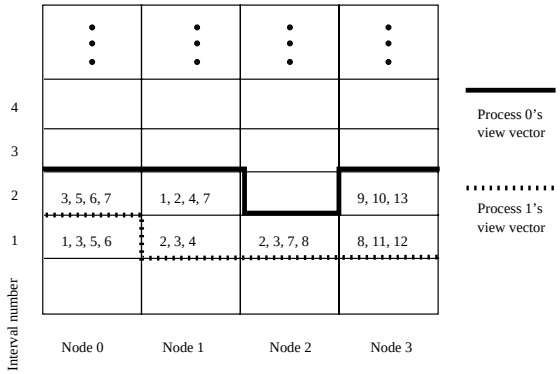

FIG. 4. *The bins data structure*

When we end an interval, this update-list is placed in a data structure called the *bins* (see Figure 4). This is the key data structure used by the SMP protocol. In our SMP protocol we use a single column for each node and not for each processor. Intervals are therefore maintained per node rather than per process. Thus, the bins data structure scales with the number of nodes and not the number of processors in the system, providing better scalability (by increasing both cluster size and number of nodes) than schemes whose data structures are proportional in size to the number of processors. This would be even more important if the nodes were large, e.g., if we were using SVM to connect several DSM machines.

To provide laziness within a node, we need a per-process data structure. We refer to it as the *view vector*. Essentially this is the "view of the world" that a process has. This vector maintains the information on what portion of the bins has been seen (i.e., the invalidations corresponding to those intervals from different nodes have been performed) by this particular process. Thus, when one process fetches new bin information from another node, the new information is available to all processes in its SMP node, if they want to access it (this makes later acquires by them cheap). The other processors however, will not *act* on this information (e.g. invalidate the pages) unless their individual view vectors say they should. Figure 5 shows how this works.

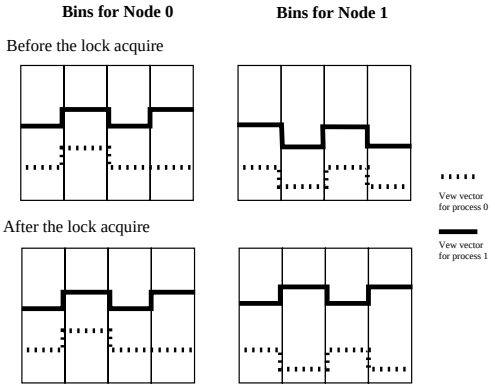

FIG. 5. *A remote lock acquire*

During a remote lock acquire operation (when the requested lock is available at a remote node), the requester sends over its view vector and a vector that indicates what bins are currently present at this node. Any portions of the bins that are not available at the requester are sent back in the form of *write notices* along with the view vector of the releaser of the lock. The requester then matches its view vector with the lock releaser's view vector (at the time of the lock release operation), and invalidates, for itself only, all the pages indicated in the bins that are "seen" by one view vector but not the other. This operation is illustrated in Figure 5. In this example process 1 on node 1 is acquiring a lock which was previously released by process 1 on node 0.

The scheme we use for locking allows a local lock acquire operation to be completely local with no external or internal messages. All that is required is the matching of the requester's view vector with the releaser's, hence

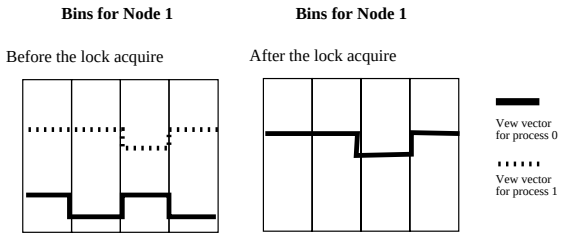

FIG. 6. *A local lock acquire*

there is a tiny amount of protocol processing with only necessary invalidations taking place. As we can see in Figure 6, this operation requires its own bins, and no data transfers across nodes are involved.

For a barrier operation all the bins that have been generated at this node and not have been propagated yet are sent to the barrier manager. The barrier manager then disperses this information to all the nodes (not processors) in the system. At this time all processes in each node match their view vectors to one that includes all the available bins, invalidating all the necessary pages.

Finally, we should mention how data are fetched when a more recent version of a page is needed. Since this is a home-based scheme, the home node of a page always has the most current version of the page. However, it is desirable when requesting a page to specify the version that is absolutely necessary and not any later one. To achieve this we use a system of *lock time-stamps* and *flush time-stamps*. Each page at the home has associated with it a flush time-stamp that indicates what is the latest interval for which the updates are currently available at the home. One may think of this as a "version" of the page. The lock time-stamp sent by the requester to the home indicates what the flush time-stamp of the page should be in order to ensure that all relevant changes to the page by other processors are in place. The lock time-stamp specifies the version of the page we should have. This time-stamp is sent to the home when we request a page, so that the appropriate decision is made.

3.2.2. Other Issues. At page fetches, the page tables of all processors in the node are invalidated to make sure that more recent data will not be overwritten by the fetched page, whereas at locks only the pages of the processor acquiring the lock are invalidated (and TLB shootdown is not needed). The former problem can be avoided by computing and applying diffs of the page at the requester rather than overwriting the whole page (essentially, the requester diffs the page that is fetched from the home with its current "twin" copy and applies only the diffs to the current local page; if the program is release-consistent, there can be no conflicts between these words and those being written by other processors in the local node). However, this adds complexity, and whether the diffing cost in the critical path is worthwhile is another tradeoff that we plan to investigate. For now, we do not implement it.

Synchronization within nodes does not use interrupts, which are needed only to service remote page fetch and synchronization requests. These protocol requests are handled by a statically assigned processor in each node, as discussed earlier. One difference between the protocol in the simulator and our real implementation is the treatment of barriers. In the simulator the barrier owner sends all write notices to all nodes, and they decide which ones are relevant.

4. Simulated Platforms. The simulation environment we use is built on top of augmint [24], an execution driven simulator using the *x86* instruction set runs on *x86* systems.

The simulated architecture (Figure 7) assumes a cluster of c–processor SMPs connected with a commodity interconnect like Myrinet [5]. Contention is modeled at all levels except the network links. The processor is P6-like, but is assumed to be a 1 intstruction per cycle (IPC) processor. The data cache hierarchy consists of a 8 KBytes first-level direct mapped write-through cache and a 512 KBytes second-level two-way set associative cache, each with a line size of 32B. The write buffer [27] has 26 entries, 1 cache line wide each, and a retire-at-4 policy. Write buffer stalls are simulated. The read hit cost is one cycle in the write buffer and first level cache and 10 cycles in the second-level cache. The memory subsystem is fully pipelined.

The memory bus is split-transaction, 64 bits wide, with a clock cycle 4x slower than the processor clock. Arbitration takes one bus cycle, and the priorities are, in decreasing order: second level cache, write buffer, memory, incoming path of the network interface, outgoing path of network interface. The I/O bus is 32 bits wide and has a clock speed half that of the memory bus. The *relative* bus bandwidths and processor speed match modern

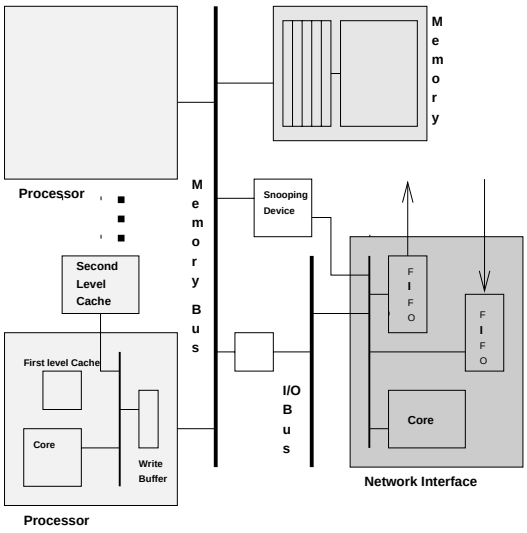

FIG. 7. *Simulated node architecture.*

systems. If we assume that the processor has a clock of 200MHz, the memory and I/O buses are 400 MBytes/s and 100 MBytes/s respectively.

Each network interface (NI) has two 1 MByte memory queues for incoming and outgoing packets. Network links operate at processor speed and are 16 bits wide. We assume a fast messaging system [9, 10, 22] that supports explicit messages. Initiating a message takes on the order of tens of I/O bus cycles. If the network queues fill, the NI interrupts the main processor and delays it to allow queues to drain.

In the *AURC* protocol simulations, a snooping device on the memory bus forwards automatic update traffic to the NI. The NI sets up network packets using its programmable core, which incurs a cost per packet. This cost must be paid in these commodity NIs proposed for use in next-generation systems.

Issuing an Interprocessor Interrupt (IPI) costs 500 processor cycles, and invoking the handler is another 500 cycles. This is very aggressive compared to what current operating systems provide, but is implementable and prevents interrupt cost from swamping out the effects of other system parameters. Protocol handlers cost a variable amount of cycles. The page size is 4 KBytes, and the cost to access the TLB from a handler running in the kernel is 50 processor cycles. In accordance to simple experiments, the cost of creating and applying a diff in HLRC is computed by adding 10 cycles for every word that needs to be compared and 10 additional cycles for each word actually included in the diff.

In setting the simulation parameters we tried to be as close as possible to an actual configuration. Our main goal was not so much to use the exact absolute values of the parameters but to maintain the important relations among them. Since the processor is less aggressive than the latest generation of processors (1 IPC at 200MHz versus 2-3 IPC at 200-400 MHz), we scaled down the values that affect the performance of the memory subsystem and the NI as well. Thus we use somewhat smaller caches and slower memory and I/O buses.

The simulator provides detailed statistics about all events in hardware, as well as statistics that help identify contention in the various components of the system. Unfortunately, protocol handlers cannot be simulated since the simulator itself is not multi-threaded. Handlers are ascribed a cost depending on the number of instructions they execute.

The programming model provided by the simulator is threads and the ANL macros. The simulator performs first touch allocation. To avoid allocating all the pages to the thread that initializes the pages, we do not simulate the initialization phase. Statistics are reset in accordance with SPLASH-2 guidelines.

5. Micro-benchmark Analysis. To understand the costs of the basic protocol and synchronization operations in this complex system, and to gain confidence in the simulator, we use a set of micro-benchmarks. These measure:

- The time to fetch a page, including the request message, the page transfer itself, and the handlers at both ends.

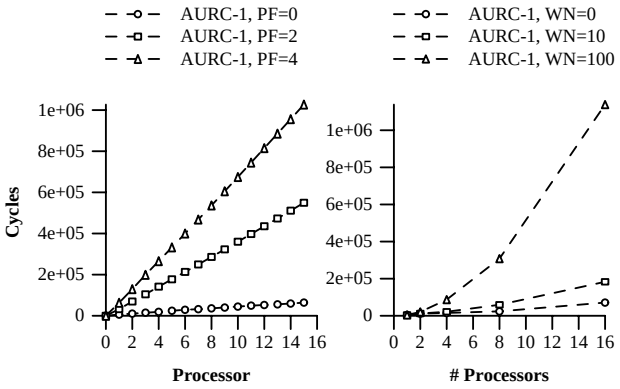

FIG. 8. *Lock and Barrier times for simple cases. The first graph from the left shows the time needed by 16 processors to acquire a lock one after the other. Each curve represents different number of page fetches in the critical section. The second graph presents the barrier cost for various numbers of processors and write notices.*

- The cost to acquire a lock for different numbers of competing processors, page fetches (misses) in the critical section, and write notices created in the critical section.
- The cost of a barrier for different numbers of processors, write notices, and diff sizes.

The unloaded cost of a page fetch is about 15000 processor cycles, or 75μs with a 200 MHz processor and the default network configuration. The one–way latency for a one-word message is about 10μs, and the time to deliver interrupts and run handlers is similar. This results in an uncontended network bandwidth of about 70-75 MBytes/s for a one-way page transfer, out of the theoretical 100 MBytes/s of the NI. The latency and bandwidth numbers are in agreement with the reported performance numbers for a real implementation [9].

Uncontended lock acquisition from a remote node costs 5800 processor cycles or 29μs with no write notices or page fetches in the critical section, and from the local node it is about 2100 cycles. As page fetches inside the critical section are increased, lock wait time increases dramatically in SVM systems due to serialization, (Figure 8).

The cost of barrier synchronization can be seen to go up sharply with the number of write notices produced in the previous interval, since these are communicated to the barrier master and from there to all nodes.

6. Applications. In our evaluation we use the SPLASH-2 [28] application suite. We will now briefly describe the basic characteristics of each application. A more detailed classification and description of the application behavior for SVM systems with uniprocessor nodes is provided in the context of *AURC* and *LRC* in [16]. The applications can be divided in two groups, regular and irregular.

6.1. Regular Applications. The applications in this category are FFT, LU and Ocean. Their common characteristic is that they are optimized to be single-writer applications; a given word of data is written only by the processor to which it is assigned. Given appropriate data structures they are single-writer at page granularity as well, and pages can be allocated among nodes such that writes to shared data are mostly local. In AURC we do not need to use a write through cache policy, and in *HLRC* we do not need to compute diffs. Protocol action is required only to fetch pages. The applications have different inherent and induced communication patterns [16, 28], which affect their performance and the impact on SMP nodes.

FFT: The FFT kernel is a complex 1-D version of the radix-$\sqrt{n}$ six-step FFT algorithm described in [1], which is optimized to minimize interprocessor communication. The data set consists of the n complex data points to be transformed and another n complex data points referred to as the roots of unity. Both sets of data are organized as matrices, which are partitioned so that every processor is assigned a contiguous set of $\sqrt{n}/p$ rows that are allocated in its local memory. Communication occurs in three matrix transpose steps, which require all-to-all interprocessor communication. Every processor transposes a contiguous submatrix of $\sqrt{n}/p$-by-$\sqrt{n}/p$ elements from every other processor to itself—thus reading remote data and writing local data—and transposes one submatrix locally. The transposes are blocked to exploit cache line reuse. To avoid memory hot-spotting, submatrices are communicated in a staggered fashion, with processor i first transposing a submatrix from processor $i+1$, then one from processor $i+2$, etc. More details can be found in [29]. We use two problem sizes, 256K(512x512) and 1M(1024x1024)

elements.

LU: The LU kernel factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense n-by-n matrix A is divided into an N-by-N array of B-by-B blocks ($n = NB$) to exploit temporal locality on submatrix elements. To reduce communication, block ownership is assigned using a 2-D scatter decomposition, with blocks being updated by the processors that own them. The block size B should be large enough to keep the cache miss rate low, and small enough to maintain good load balance. Fairly small block sizes (B=8 or B=16) strike a good balance in practice. Elements within a block are allocated contiguously to improve spatial locality benefits, and blocks are allocated locally to processors that own them. See [29] for more details. We use two versions of LU that differ in their organization of the matrix data structure. The contiguous version of LU uses a four-dimensional array to represent the two-dimensional matrix, so that a block is contiguous in the virtual address space. It then allocates on each page the data of only one processor. The non-contigous version uses a two-dimensional array to represent the matrix, so that successive subrows of a block are not contiguous with one another in the address space. In this version, data written by multiple processors span a page. LU exhibits a very small communication to computation ratio but is inherently imbalanced. We used a 512x512 matrix.

Ocean: The Ocean application studies large-scale ocean movements based on eddy and boundary currents. It partitions the grids into square-like subgrids rather than groups of columns to improve the communication to computation ratio. Each 2-D grid is represented as a 4-D array in the "contiguous" version, with all subgrids allocated contiguously and locally in the nodes that own them. The equation solver used is a red-black, W-cycle multigrid solver. The communication pattern in the Ocean simulation application is largely nearest-neighbor and iterative on a regular grid. We run both the contiguous (4-D array) and non-contiguous (2-D array) versions of Ocean on two problem sizes, 258x258 and 514x514, with an error tolerance of 0.001.

6.2. Irregular Applications. The irregular applications in our suite are Barnes, a hierarchical N-body simulation; Radix, an integer sorting program; Raytrace, a ray tracing application from computer graphics; Volrend, a volume rendering application; and Water, a molecular dynamics simulation of water molecules in liquid state.

Barnes: The Barnes application simulates the interaction of a system of bodies (galaxies or particles, for example) in three dimensions over a number of time-steps, using the Barnes-Hut hierarchical N-body method. It represents the computational domain as an octree with leaves containing information about the bodies and internal nodes representing space cells. Most of the time is spent in partial traversals of the octree (one traversal per body) to compute the forces on individual bodies. The communication patterns are dependent on the particle distribution and are quite unstructured. No attempt is made at intelligent distribution of body data in main memory, since this is difficult at page granularity and not very important to performance. We ran experiments for different data set sizes, but present results for 8K and 16K particles. Access patterns are irregular and fine-grained. We use two versions of Barnes, which differ in how the shared octree is built and managed across time-steps. The first version (Barnes-rebuild) builds the tree from scratch after each computation phase. The second version, Barnes(space) [17], is optimized for SVM implementations—in which synchronization is expensive—and it avoids locking as much as possible. It uses a different tree-building algorithm, where each processor first builds its own partial tree, and all partial trees are merged to the global tree after each computation phase.

Radix: The integer radix sort kernel is based on the method described in [3]. The algorithm is iterative, performing one iteration for each radix r digit of the keys. In each iteration, a processor passes over its assigned keys and generates a local histogram. The local histograms are then accumulated into a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration. This permutation step requires all-to-all, irregular communication. The permutation is inherently a sender-determined one, so keys are communicated through scattered, irregular writes to remotely allocated data. See [6, 29] for details.

Raytrace: This application renders a three-dimensional scene using ray tracing. A hierarchical uniform grid (similar to an octree) is used to represent the scene, and early ray termination and antialiasing are implemented, although antialiasing is not used in this study. A ray is traced through each pixel in the image plane, and reflects in unpredictable ways off the objects it strikes. Each contact generates multiple rays, and the recursion results in a ray tree per pixel. The image plane is partitioned among processors in contiguous blocks of pixel groups, and distributed task queues are used with task stealing for load balancing. The major data structures represent rays, ray trees, the hierarchical uniform grid, task queues, and the primitives that describe the scene. The data access patterns are highly unpredictable in this application. See [26] for more information. The version we use is modified from the SPLASH-2 version [28] to run more efficiently on SVM systems. A global lock that was not necessary

Application	Page Faults			Page Fetches			Local Locks			Remote Locks			Barriers
	1	4	8	1	4	8	1	4	8	1	4	8	1
FFT (20)	397.12	251.89	270.32	393.31	167.17	91.59	0.00	0.00	0.00	0.00	0.00	0.00	1.14
LU(contiguous) (512)	81.36	56.61	48.07	71.78	34.94	11.86	0.02	0.22	0.25	0.27	0.07	0.04	19.24
Ocean(contiguous) (514)	647.61	117.34	103.17	646.97	24.92	7.20	0.00	0.76	1.31	2.17	1.41	0.86	13.05
Water(nsquared) (512)	69.19	22.06	8.04	68.26	19.01	7.29	0.01	120.36	158.14	203.20	82.85	45.06	3.30
Water(spatial) (512)	97.86	21.42	9.23	93.81	17.73	6.04	0.01	1.83	2.60	3.94	2.16	1.39	4.19
Radix (1K)	208.82	82.73	98.40	203.69	44.92	13.41	0.10	0.44	3.30	4.52	4.11	1.33	1.04
Volrend (head)	105.09	44.06	34.49	104.78	29.35	6.53	0.00	29.34	43.80	44.34	17.64	3.97	1.61
Raytrace (car)	89.80	25.64	6.83	89.79	25.57	6.76	0.03	2.21	3.96	4.89	3.26	1.34	0.10
Barnes(rebuild) (8K)	211.22	103.02	55.47	207.72	90.90	40.31	0.07	33.92	71.76	127.74	93.81	55.18	1.44
Barnes(space) (8K)	48.06	10.43	7.67	46.20	9.92	3.48	0.00	0.16	0.21	0.24	0.07	0.03	1.79

TABLE 1

Normalized number of page faults, page fetches, local and remote lock acquires and barriers per 10^7 cycles per processor for each application for 1,4 and 8 processors per node.

was removed, and task queues are implemented better for SVM and SMPs. Inherent communication is small. We present results only for the SMP protocols due to simulation cycle limitations.

Volrend: This application renders a three-dimensional volume using a ray casting technique. The volume is represented as a cube of voxels (volume elements), and an octree data structure is used to traverse the volume quickly. The program renders several frames from changing viewpoints, and early ray termination and adaptive pixel sampling are implemented, although adaptive pixel sampling is not used in this study. A ray is shot through each pixel in every frame, but rays do not reflect. Instead, rays are sampled along their linear paths using interpolation to compute a color for the corresponding pixel. The partitioning and task queues are similar to those in Raytrace. The main data structures are the voxels, octree, and pixels. Data accesses are input-dependent and irregular, and no attempt is made at intelligent data distribution. See [21] for details. The version we use is also slightly modified from the SPLASH-2 version [28], to provide a better initial assignment of tasks to processes before stealing. This improves SVM performance greatly. Inherent communication volume is small.

Water: This application evaluates forces and potentials that occur over time in a system of water molecules. The forces and potentials are computed every time-step, and a predictor-corrector method is used to integrate the motion of the water molecules over time. We use two versions of Water, Water-nsquared and Water-spatial. The first uses an $O(n^2)$ algorithm to compute the forces, while the second computes the forces approximately using a fixed cutoff radius, resulting in an $O(n)$ algorithm. Water-nsquared can be categorized as a regular application, but we put it here to ease the comparison with Water-spatial. In both versions, updates are accumulated locally between iterations and performed at once at the end of each iteration. The inherent communication to computation ratio is small. We use a data set size of 512 molecules.

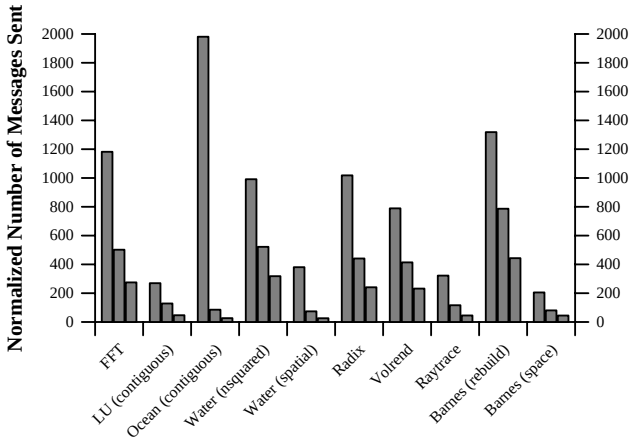

FIG. 9. *Normalized number of messages sent per processor for each application for 1,4 and 8 processors per node.*

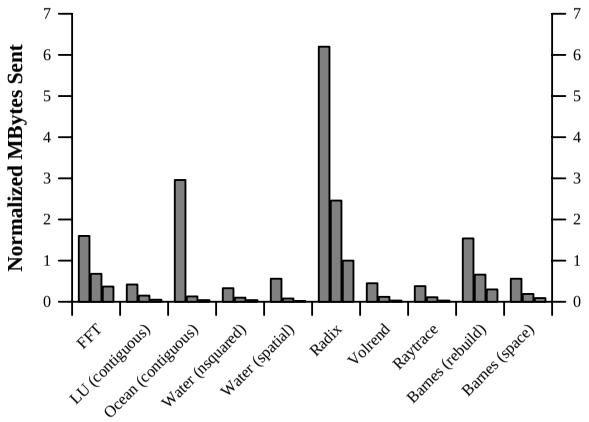

FIG. 10. *Normalized number of MBytes sent per processor for each application for 1,4 and 8 processors per node.*

Table 1 and Figures 9 and 10 can be used to characterize the applications. Table 1 presents counts of protocol events for each application, for 1-, 4- and 8-processors per node configurations. Figures-9 and 10 show for the same configurations the numbers of messages and MBytes of information (both data and protocol) that are sent by each processor in the system. These statistics are normalized to the compute time of each application (per 10^7 cycles). All the numbers presented are averages over all processors in the system.

7. Metrics and Presentation of Results. In the next sections we present our results that address the issues raised in the introduction. We compare two system configurations for the two different protocols. The first system has uniprocessor nodes, whereas the second uses SMP nodes. In all configurations the speed of the memory and I/O buses is set to 400 MBytes/s and 100 MBytes/s respectively, and we assume a 200MHz processor. As mentioned in Section 4, these values result in a realistic commodity configuration, given the relative performance of the different components. Note that the bandwidths are the same whether the nodes are uniprocessor or multiprocessor. We have performed some experiments where the bandwidths are scaled with node size as well. The simulator provides us with very detailed statistics for most hardware and protocol events.

Let us first discuss the performance metrics we use. While speedup is an important metric, factors unrelated to the SVM protocol can cause speedups to be high even when the protocol itself is not well suited to the application. For example, a sequential execution can perform very poorly due to the working set not fitting in the cache, a problem that may go away in a parallel execution if the application is such that the important working set diminishes as the number of processors is increased, and thus leads to very high speedups. We will see an example of this in the Ocean application. To understand how well protocols themselves perform, we use both speedups and a metric we call *protocol efficiency*.

We define protocol efficiency for a given application and protocol for N processors as:

$$f_N = \frac{\sum_{0 \le i \le N}(C_i + S_i)}{\sum_{0 \le i \le N} E_i},$$

where E_i, C_i, S_i are the elapsed, compute and cache stall time of the i^{th} processor in the parallel execution. Obviously $f_N \in [0, 1]$. Note that the costs in the sequential execution are not directly involved in this definition. Alternative definitions with similar goals are also possible.

We present performance data in two main forms: tables with speedups and protocol efficiency factors in each section, and bar-graphs with per-processor breakdowns of execution time for the most interesting cases (Figures 11–17).

In the bar graphs the execution cost is divided into the following components. *Thread Compute Time* is essentially a count of the instructions executed by each thread. *CPU Stall Time* is the time the CPU is stalled on local memory references. *Thread Data Wait Time* (or page fetch time) is the time each thread is waiting for data to arrive from a remote node (page fetches). *Thread Lock Time* and *Thread Barrier Time* are the times spent in synchronization events, including idle wait time and operation overhead. *I/O Stall Time* is the time the

processor spends initiating page and message transfers and waiting for the network interface queues to drain. The last component, *Handler Compute Time*, is the time spent in protocol handlers. Some of this cost is also included in other costs, for instance *Thread Data Wait Time*.

Tables 4-5 in Appendix A give more analytical statistics about each application. These can be consulted while reading the analysis of each application. Important statistics are highlighted in the analysis of each application as well.

To simplify the discussion, we now clarify a couple of issues up front. *Thread Data Wait Time* can be large either because many pages need to be fetched (*frequency*) or because the cost per page fetch is high due to *contention*. Imbalances in data wait time also stem from imbalances in either of these factors, and we will indicate which dominates. Lock and barrier synchronization times also have two components: the time spent waiting (for another processor to release the lock or for all processors to reach the barrier) and the time spent exchanging messages and doing protocol work (e.g., after the last processor arrives at the barrier). We will separate these out, calling the former *wait time* and the latter *protocol cost*. As we saw in Section 5, wait time for locks is often increased greatly in SVM systems due to page misses occurring frequently inside critical sections and increasing serialization [16], as well as to increased protocol activity at locks, which has the same effect. This makes locks much more expensive for SVM systems than for hardware-coherent systems.

Finally, we adopt a naming convention in which (for example) *AURC-1* is *AURC* on a system with one processor per node and *HLRC-4* is *HLRC* on a system with 4 processors per node.

Application	Problem Size	Speedups			
		AURC-1	AURC-4	HLRC-1	HLRC-4
FFT	18	5.24	6.20	4.43	5.74
	20	8.53	8.29	7.73	8.28
LU (contiguous)	512	10.78	12.42	10.16	12.25
Ocean (contiguous)	258	6.10	16.86	5.43	15.23
	514	6.52	12.83	6.12	12.80
Barnes (rebuild)	16K	5.30	6.24	2.45	3.82
Barnes (no-locks)	8K	12.94	12.81	11.69	12.19
	16K	13.30	12.95	10.94	11.52
Radix	1024	2.82	1.25	0.63	3.41
Raytrace	car	6.38	11.82	14.06	14.79
Volrend	head	9.14	11.95	7.86	8.81
Water (nsquared)	512	9.09	9.71	8.56	8.84
Water (spatial)	512	7.89	8.52	7.41	10.05

TABLE 2
Speedups for the uniprocessor and the SMP node configurations.

8. System Comparison. In this section we discuss how the protocols behave when moving from uniprocessor to SMP nodes. Tables 2 and 3 present these results.

From Table 2 we can divide the applications into different classes in terms of their behavior in going from uniprocessor to SMP nodes. The first class is applications for which both protocols improve with SMPs. These applications are LU, Ocean-contiguous, Barnes-rebuild, Volrend and Water-spatial. We differentiate the behavior of these applications into three subgroups.

The first group consists of Ocean (Figure 12), which improves dramatically because of the localized, near-neighbor pattern of communication and the high amount of barrier synchronization.

Ocean (Figure 12): In both *AURC-4* and *HLRC-4* data wait times are reduced significantly, compared to the uniprocessor configuration. The page fetch cost however, is increased by about 100% and 30% in *AURC-4* and *HLRC-4*, respectively, because of the larger contention in the memory bus and network interface. The reduction in data wait time comes from the sharing pattern in Ocean. The communication pattern is nearest neighbor, so if

Application	Problem Size	Efficiency Factors			
		AURC-1	AURC-4	HLRC-1	HLRC-4
FFT	18	36%	42%	26%	40%
	20	57%	65%	45%	59%
LU (contiguous)	512	58%	68%	53%	66%
Ocean (contiguous)	258	21%	51%	19%	47%
	514	33%	82%	29%	79%
Barnes(rebuild)	16K	33%	39%	15%	24%
Barnes (no-locks)	8K	86%	83%	74%	80%
	16K	89%	84%	70%	76%
Radix	1024	17%	8%	4%	21%
Raytrace	car	39%	72%	86%	90%
Volrend	head	58%	76%	50%	56%
Water(nsquared)	512	58%	62%	55%	57%
Water(spatial)	512	51%	55%	46%	63%

TABLE 3
Efficiency factors (as %) for the uniprocessor and the SMP configurations.

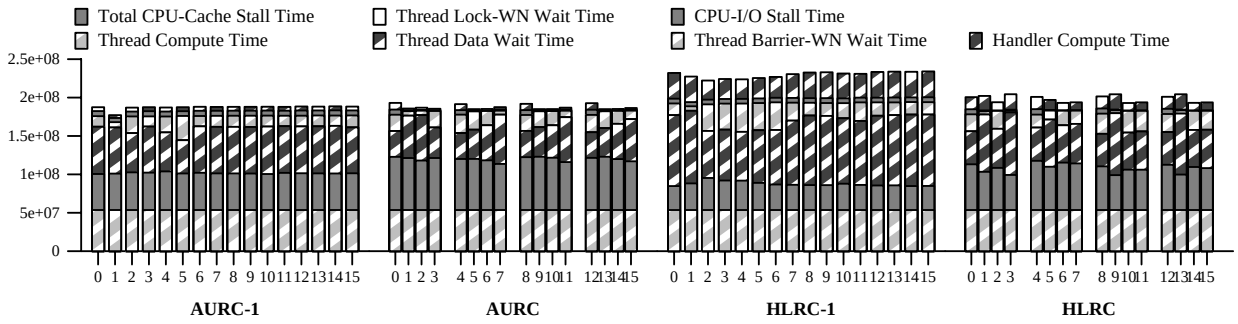

FIG. 11. *Cost breakdown for the 1M FFT for* AURC *and* HLRC.

the processes assigned to an SMP node are adjacent in the grid then much of the sharing will be contained locally. The number of page fetches is greatly reduced, and performance improves even though the cost per page fetch goes up by 50% due to contention at the bus and the node-to-network interfaces. Also, in Ocean-contiguous the data accessed by each processor are mostly allocated in the local SMP node. The lower data wait time results in lower synchronization time.

We see that *AURC* improves significantly lot by using SMPs. If we look at the efficiency factor of *AURC* for Ocean-contiguous, we see that it is very low for the uniprocessor node case, and much better for the SMP case. This is because the relatively good speedup in Ocean-contiguous in the uniprocessor case comes mainly from cache effects. The application, in terms of the protocol overheads, performs very poorly, which gives a low efficiency factor. In the SMP case, the protocol overheads reduce dramatically, and this is captured by the efficiency factors.

AURC-4 gives a speedup of around 16. As was mentioned before, the super-linear speedup is due to cache effects. More precisely, the sequential run suffers from very high local stall time (2-3 times the compute time). The parallel application takes advantage of the smaller working set size in each processor, and the stall time is reduced substantially.

Cache effects are noticeable in the parallel execution for the larger problem size as well. The working set does not fit in the cache of each processor, and the stall time is increased substantially. Since multiple processors share the same memory bus, there is a great deal of contention in the system. The application spends most of the time

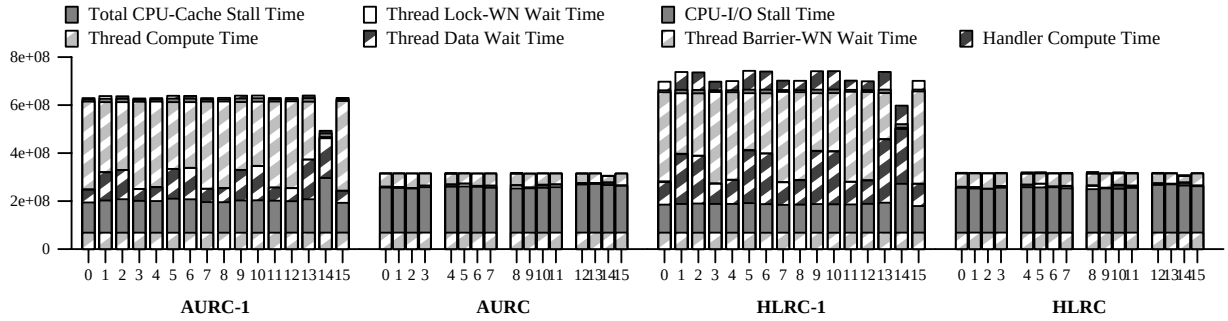

FIG. 12. *Cost breakdown for Ocean-contiguous (514x514) for* AURC *and* HLRC.

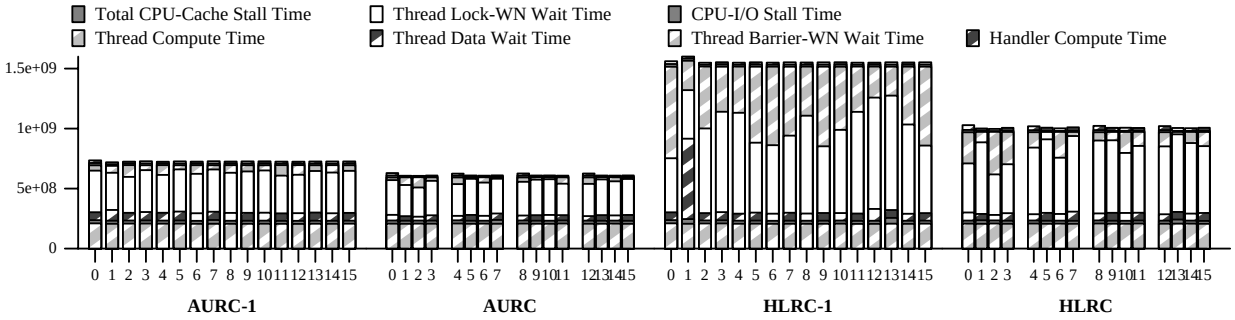

FIG. 13. *Cost breakdown for Barnes-rebuild for* AURC *and* HLRC.

in waiting for memory operations, with a CPU stall time of (268%, 261%, 296%). Thus, speedups are much lower for the SMP node case. Protocol efficiencies increase however. The main reason is that relative protocol overheads are reduced in the larger problem size and poorer performance comes from cache effects.

In the second group are LU and Barnes-rebuild (Figure 13), where the improvement comes again from sharing of data and lower synchronization costs as well, but at a much smaller degree than Ocean. These are applications for which clustering helps in data sharing and prefetching, without the inherent communication pattern of the applications to match the two-level hierarchy as in Ocean. Cheaper synchronization also makes a noticeable difference.

LU: The improvement in performance compared to the uniprocessor configuration comes from cheaper synchronization, namely hierarchical barriers, and sharing of data in each node, which results in lower data wait times. For instance, the barrier cost is reduced by 33% in *AURC-4* and by 39% in *HLRC-4*. Similarly, the reduction in data wait time is 6% and 14% respectively.

Barnes-rebuild (Figure 13): Barnes-rebuild performs quite poorly under both protocols due to extremely high lock costs. In the SMP case improvements in data wait time due to sharing and prefetching, and in lock times due to local acquires, are relatively small. Overall performance improves, but not by much. In *HLRC-4* with SMP nodes, data wait time is smaller and more balanced, but lock acquire costs, which dominate performance, remain expensive (163%, 261%, 312%), and imbalanced. The reduction in the number of remotely acquired locks due to the use of SMP nodes is not very large.

The third group in this class contains applications where there is an improvement but of a different degree for *AURC-4* and *HLRC-4*. These are Volrend (Figure 16), and Water-spatial (Figure 17).

Volrend (Figure 16): Volrend uses a task stealing mechanism to achieve load balancing. This mechanism uses locks to perform atomic operations on the task queue. Using SMPs results in an improvement in all protocol costs without introducing additional problems. In *AURC-4* page fetches are reduced by about 60% due to sharing and prefetching. Moreover, lock time is reduced by about 50% and computation is not as imbalanced, since the task stealing method takes advantage of the multiple nodes per SMP (it tries to steal from local processors first,

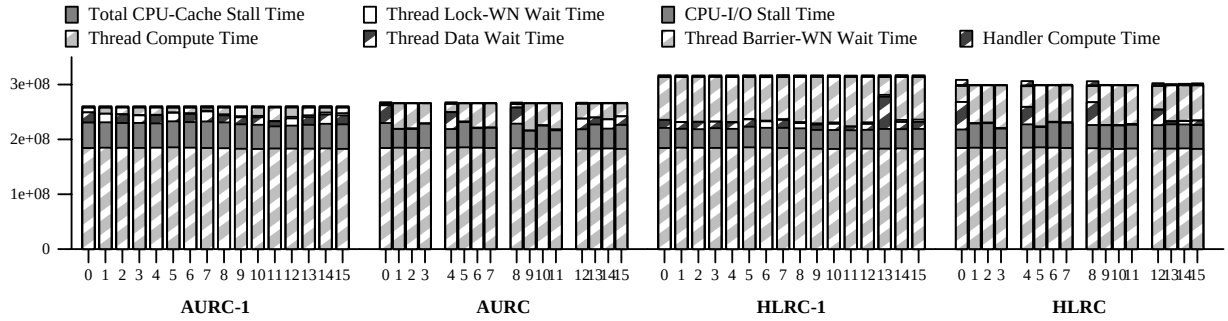

FIG. 14. *Cost breakdown for Barnes-nolocks for* AURC *and* HLRC.

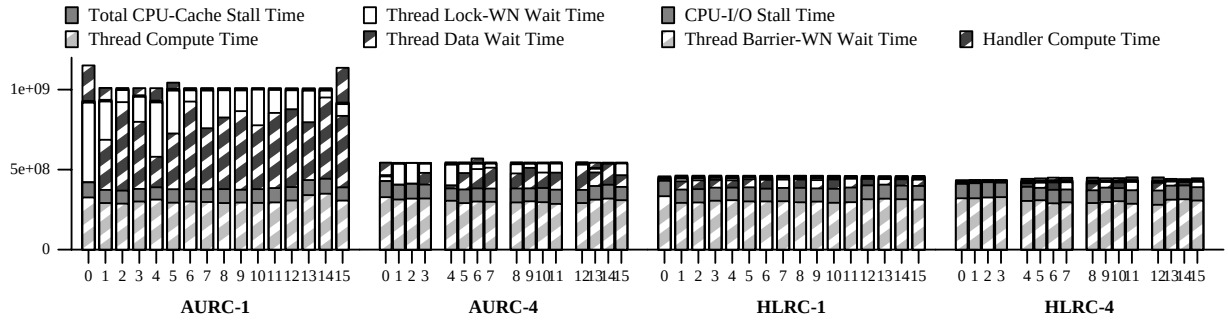

FIG. 15. *Cost breakdown for Raytrace for* AURC *and* HLRC.

converting remote locks and communication to inexpensive local locks and communication). *HLRC-4* improves somewhat but not as much as *AURC-4*. The main reason is that lock imbalances are still present because remote locks are more expensive in *HLRC*, and the distribution of local versus remote locks in each processor is somewhat more uneven in *HLRC*, which leads to higher imbalances.

Water-spatial (Figure 17): In *AURC-4* performance improves slightly compared to *AURC-1*. Total data wait time is reduced, but it is not balanced among processors. The reason for the imbalance turns out to be contention in the outgoing queue in some network interfaces because of automatic update traffic, which slows the progress of outgoing requests. The net improvement in performance is small.

In *HLRC-4* the picture changes. *HLRC-4* performs considerably better than *HLRC-1*. Data wait time decreases considerably because of sharing and prefetching. Also synchronization costs are much lower. Unlike *AURC-4*, though, protocol overheads remain balanced and the improvement in performance is larger. There is practically no contention in the outgoing queue, and request messages are sent out immediately for all the processors.

The second class of applications consists of FFT (Figure 11), Barnes-nolocks (Figure 14), and Water-nsquared. These applications do not benefit (or benefit little) from the use of SMP nodes with either protocol.

FFT (Figure 11): Since communication is all-to-all rather than localized in the matrix transpose in FFT, the clustering accomplished by using SMP nodes reduces the amount of inherent remote communication by roughly a factor of k/p, where k is the number of processors in an SMP node, and n is the total number of processors. However, in the SMP configuration we find that page fetches are reduced dramatically for the small problem size compared to the uniprocessor node case. This is because the problem size is such that the data that need to be fetched by different processors lie on the same page. Using multiple processors in each node thus results in substantial prefetching and in a reduction in data wait time. However, data wait time is imbalanced among processors within a node because the number of page fetches differs among them, so the overall reduction does not translate to a large improvement in performance. Barrier synchronization time is high due to this imbalance, and protocol efficiency low.

For the larger problem size, speedups and efficiency factors are somewhat better in all cases because less

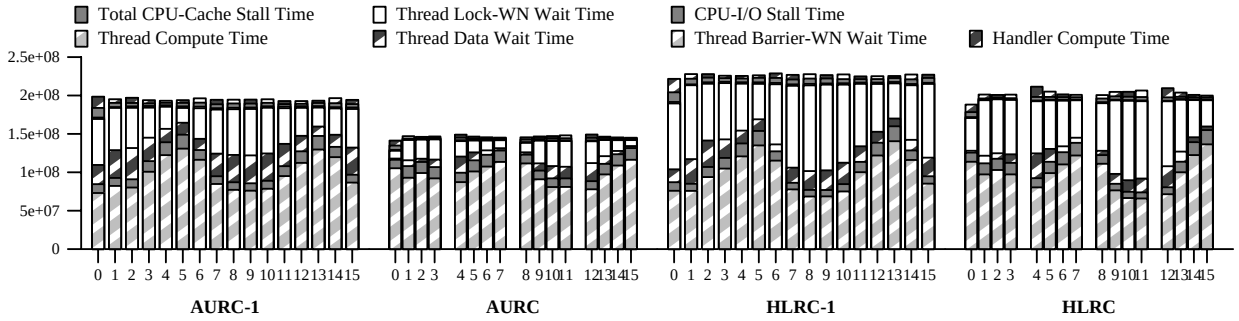

FIG. 16. *Cost breakdown for Volrend for* AURC *and* HLRC.

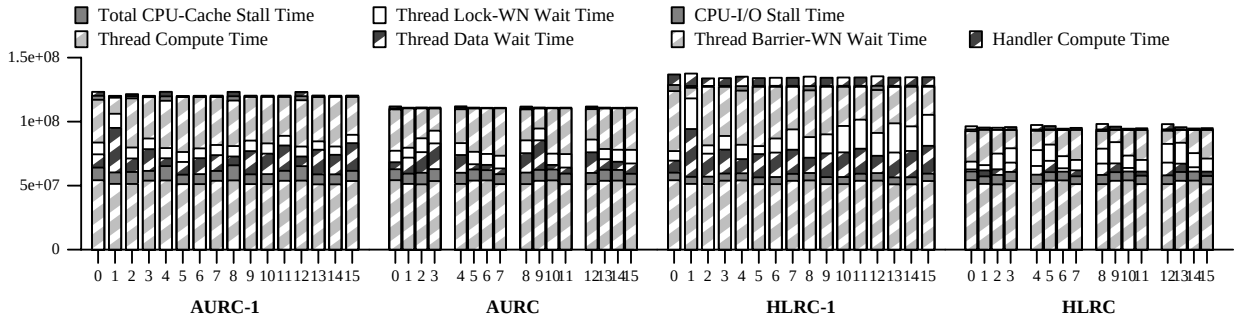

FIG. 17. *Cost breakdown for Water-spatial for* AURC *and* HLRC.

of the data that are fetched on a page are wasted. In both *AURC-4* and *HLRC-4* data wait time once again improves because of sharing and prefetching, but imbalances in the data wait time limit the effect on performance. Moreover, with the larger working sets, local memory stall time is considerably higher than in the uniprocessor node configuration due to contention on the memory bus. For instance, in *AURC-4* stall time is (112%, 115%, 129%) as opposed to (87%, 84%, 93%) in *AURC-1*. (i.e., 112% minimum across processors, 115% average, and 129% maximum, and so on).

Barnes-nolocks (Figure 14): As in the uniprocessor configuration, Barnes-nolocks performs very well under all protocols in the SMP case as well. Clustering does not help much. The reason is that while the barriers that this application uses often in tree building (instead of locks) are cheaper, data wait time is still imbalanced due to different number of page fetches among processors. This shows up as synchronization time, and the benefits from clustering are negligible.

Water-nsquared: Here there is a large overall reduction in inter-node communication in the SMP configuration due to prefetching, but as in FFT imbalances are created in the numbers of page fetches among processors within an SMP. The same is true for lock accesses. Thus, despite the large overall reduction in communication, the improvement in performance is small.

In the applications in this second class, using SMP nodes does reduce aggregate communication and synchronization costs substantially due to sharing and prefetching, but the increases often do not translate to large performance increases because the reductions are imbalanced on a per-processor basis within the nodes.

In the third class of applications are Radix and Raytrace (Figure 15). These exhibit different relative behavior under different protocols.

Radix: All protocols perform very poorly with Radix. Among the *AURC* protocols, *AURC-4* performs worse than *AURC-1*. The amount of data sent as replies to page requests decreases to about one fourth in the SMP configuration, as with FFT, yet performance is much worse because *AU* traffic per processor is about the same as in *AURC-1*, while bus and network interface bandwidth are now shared. This creates contention in all components of the path between the sender and the receiver, delaying both *AU* and request/control messages. Consequently,

all forms of communication and synchronization are slowed down. For instance the average page fetch cost is huge, (466935, 1312487, 1942957) cycles. Protocol efficiency is very low for *AURC-4*.

HLRC-4 performs much better than *HLRC-1*. *HLRC-4* does not suffer from increased traffic as much as *AURC-4*, since there is no automatic update traffic. Messages are delivered faster, and fetch time, lock time and barrier costs are much smaller, so performance improves. However, SVM protocols do not seem to be able to handle the bursty, scattered, remote-write communication of Radix, and do not do well overall.

Raytrace (Figure 15): In Raytrace, *AURC* benefits greatly from sharing, whereas the improvement for *HLRC* is very small. The reason for the improvement in *AURC* is the small automatic update messages that are used to update data. When SMP nodes are used, the number of messages is reduced substantially (the amount of data is reduced by a factor of 4 with 4-way SMPs), and as a result the contention problem is much less severe than the uniprocessor node case. In *HLRC* the number of messages is not a problem since coalesced, large messages (diffs) are used to update the home nodes. Thus the performance of *HLRC-1* is much better than *AURC-1* and the benefit of using SMP nodes is much smaller.

9. Related Work. Our study on uniprocessor systems is consistent with the results obtained in [16] for their *slow bus* case, although we compare *AURC* against *HLRC* instead of *LRC*. *HLRC* and *LRC* have been compared for some applications on the Intel Paragon in [30].

Holt et al. in [14] present a metric similar to protocol efficiency. They use it to characterize the performance of applications on a hardware cache coherent machine.

Several papers have discussed the design and performance of shared virtual memory for SMPs [7, 12, 18, 23] for different protocols. Erlichson et al. [12] conclude that the transition from uniprocessor to multiprocessor nodes is nontrivial, and performance can be seriously affected unless great care is taken. However, the protocol and system they assume is very different from ours.

[2] is a preliminary version of this work. For this work, we use a much more detailed simulation environment and improved implementations of the the protocols.

10. Summary and Conclusions. The proliferation of small-scale, bus-based shared memory multiprocessors has made it very attractive to use clusters of these as scalable multiprocessors. Communication between SMPs in these clusters is performed primarily by software. An important question for these systems is whether the coherent shared address space communication abstraction provided within the SMP node can be extended effectively in software across nodes as well. Since shared virtual memory systems have been developed to provide this abstraction across uniprocessors in software, it is attractive, though non-trivial, to extend them to exploit SMP nodes effectively. If successful, this approach can make a coherent shared address space a viable programming model for both tightly-coupled multiprocessors (using hardware cache coherence) and loosely coupled clusters.

This paper has described such an SVM system for SMP nodes, which attempts to use the hardware sharing within the SMP as much as possible and reduce the frequency of software protocol involvement, and we have tried to understand its performance, particularly in comparison with the baseline SVM protocol across uniprocessors.

We find that for the same total processor count, using SMP nodes improves performance for both the all-software *HLRC* protocol and the *AURC* protocol that uses hardware remote write support, particularly when the protocols and applications are designed and used properly. This is despite the fact that the traffic pressure on the memory and I/O buses tends to increase. In some cases imbalances are introduced because of increased contention, and overall benefits are reduced. However, the reduction in the number of remote page fetches and in synchronization cost tends to overshadow the increased contention, resulting in a small improvement in some applications and a substantial improvement in others. Preliminary experiments show that even when performance degrades due to contention on the I/O bus (which interfaces to the network), it can be alleviated by increasing I/O bus bandwidth with the number of processors in an SMP node.

We find that out of ten applications, both protocols improve substantially with the use of SMPs in five of them, in three there is a smaller improvement (or they perform the same as in the uniprocessor node case), and for the other two results differ for each protocol with *AURC-4* performing worse than *AURC-1* for one application.

For two out of the three regular applications performance improves significantly for both protocols. An exception to this is FFT, with its all-to-all communication, which exhibits no significant improvement. Among the irregular applications, Barnes-rebuild, Volrend, and Water-spatial benefit in both protocols, Barnes-nolocks and Water-nsquared do not exhibit any significant improvement, and the last two applications, Radix and Raytrace, behave differently under each protocol. Radix improves with *HLRC-4* but performs worse with *AURC-4*, and Ray-

trace improves with *AURC-4* and exhibits no improvement with *HLRC-4*. In many cases there is a large reduction in the number of remote (cross-node) page fetches due to the use of SMP nodes, even larger than expected from inherent interprocess communication patterns, due to the interactions with page granularity. However, performance does not increase as much as expected even in these cases, because the reduction is uneven across processors so there is significant load imbalance in communication costs.

In doing this research, we found that modifying and restructuring applications a little—sometimes trivially and sometimes algorithmically—can go a long way toward improving the performance of SVM systems, with or without automatic update support. The tree-building in Barnes, the elimination of a lock and management of task queues in Raytrace, and the change in initial assignment of tasks in Volrend are examples. The performance impact is much greater than the tradeoff among these protocols. Finally, while overall parallel performance appears promising at this scale, for some applications such as Radix we are still not able to achieve good parallel performance from shared virtual memory systems.

11. Acknowledgments. We are indebted to NEC Research and particularly to James Philbin and Jan Edler for providing us with simulation cycles and a stable environment. We would like to thank Hongzhang Shan for making available to us improved versions of some of the applications.

REFERENCES

[1] D. H. Bailey, *FFTs in External or Hierarchical Memories*, Journal of Supercomputing, 4 (1990), pp. 23–25.

[2] A. Bilas, L. Iftode, D. Martin, and J. Singh, *Shared virtual memory across SMP nodes using automatic update: Protocols and performance*, Tech. Rep. TR-517-96, Princeton, NJ, Mar. 1996.

[3] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, *A comparison of sorting algorithms for the connection machine CM-2*, in Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures, July 1991, pp. 3–16.

[4] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg, *A virtual memory mapped network interface for the shrimp multicomputer*, in Proceedings of the 21st Annual Symposium on Computer Architecture, Apr. 1994, pp. 142–153.

[5] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, *Myrinet: A gigabit-per-second local area network*, IEEE Micro, 15 (1995), pp. 29–36.

[6] J. P. S. Chris Holt and J. Hennessy, *Architectural and application bottlenecks in scalable DSM multiprocessors*, in Proceedings of the 23rd Annual International Symposium on Computer Architecture, May 1996.

[7] A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel, *Software versus hardware shared-memory implementation: A case study*, in Proceedings of the 21st Annual Symposium on Computer Architecture, Apr. 1994, pp. 106–117.

[8] D. Culler and J. P. Singh, *Parallel Computer Architecture*, Morgan Kaufmann Publishers, 1998.

[9] C. Dubnicki, A. Bilas, K. Li, and J. Philbin, *Design and implementation of virtual memory-mapped communication on myrinet*, in Proceedings of the 1997 International Parallel Processing Symposium, April 1997.

[10] T. Eicken, D. Culler, S. Goldstein, and K. Schauser, *Active messages: A mechanism for integrated communication and computation*, in Proceedings of the 19th Annual Symposium on Computer Architecture, May 1992, pp. 256–266.

[11] A. Erlichson, B. Nayfeh, J. Singh, and K. Olukotun, *The benefits of clustering in shared address space multiprocessors: An applications-driven investigation.*, in Supercomputing '95, 1995, pp. 176–186.

[12] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy, *SoftFLASH: analyzing the performance of clustered distributed virtual shared memory*, in The 6th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1996, pp. 210–220.

[13] R. Gillett, M. Collins, and D. Pimm, *Overview of network memory channel for PCI*, in Proceedings of the IEEE Spring COMPCON '96, Feb. 1996.

[14] C. Holt, M. Heinrich, J. P. Singh, , and J. L. Hennessy, *The effects of latency and occupancy on the performance of dsm multiprocessors*, Tech. Rep. CSL-TR-95-xxx, Stanford University, 1995.

[15] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li, *Improving release-consistent shared virtual memory using automatic update*, in The 2nd IEEE Symposium on High-Performance Computer Architecture, Feb. 1996.

[16] L. Iftode, J. P. Singh, and K. Li, *Understanding application performance on shared virtual memory*, in Proceedings of the 23rd Annual Symposium on Computer Architecture, May 1996.

[17] D. Jiang, H. Shan, and J. P. Singh, *Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors*, in Sixth ACM Symposium on Principles and Practice of Parallel Programming, June 1997.

[18] M. Karlsson and P. Stenstrom, *Performance evaluation of cluster-based multiprocessor built from atm switches and bus-based multiprocessor servers*, in The 2nd IEEE Symposium on High-Performance Computer Architecture, Feb. 1996.

[19] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel, *Treadmarks: Distributed shared memory on standard workstations and operating systems*, in Proceedings of the Winter USENIX Conference, Jan. 1994, pp. 115–132.

[20] P. Keleher, A. Cox, and W. Zwaenepoel, *Lazy consistency for software distributed shared memory*, in Proceedings of the 19th Annual Symposium on Computer Architecture, May 1992, pp. 13–21.

[21] J. Nieh and M. Levoy, *Volume rendering on scalable shared-memory MIMD architectures*, in Proceedings of the Boston Workshop on Volume Visualization, Oct. 1992.

[22] S. Pakin, M. Buchanan, M. Lauria, and A. Chien, *The Fast Messages (FM) 2.0 streaming interface*. Submitted to Usenix'97, 1996.

[23] D. Scales, K. Gharachorloo, and C. Thekkath, *Shasta: A low overhead, software-only approach for supporting fine-grain shared memory*, in The 6th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 1996.

[24] A. Sharma, A. T. Nguyen, J. Torellas, M. Michael, and J. Carbajal, *Augmint: a multiprocessor simulation environment for Intel x86 architectures*, tech. rep., University of Illinois at Urbana-Champaign, March 1996.

[25] J. P. Singh, A. Gupta, and J. L. Hennessy, *Implications of hierarchical N-body techniques for multiprocessor architecture*, ACM Transactions on Computer Systems, (1995). To appear. Early version available as Stanford Univeristy Tech. Report no. CSL-TR-92-506, January 1992.

[26] J. P. Singh, A. Gupta, and M. Levoy, *Parallel visualization algorithms: Performance and architectural implications*, IEEE Computer, 27 (1994).

[27] K. Skadron and D. W. Clark, *Design issues and tradeoffs for write buffers*, in The 3nd IEEE Symposium on High-Performance Computer Architecture, Feb 1997.

[28] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, *Methodological considerations and characterization of the SPLASH-2 parallel application suite*, in Proceedings of the 23rd Annual Symposium on Computer Architecture, May 1995.

[29] S. C. Woo, J. P. Singh, and J. L. Hennessy, *The performance advantages of integrating message-passing in cache-coherent multiprocessors*, in Proceedings of Architectural Support For Programming Languages and Operating Systems, 1994.

[30] Y. Zhou, L. Iftode, and K. Li, *Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems*, in Proceedings of the Operating Systems Design and Implementation Symposium, Oct. 1996.

Appendix A: Detailed Performance Data. Table 4 compares *AURC-1* and *AURC-4*, and Table 5 compares *HLRC-1* and *HLRC-4* in more detail. For each application there are two columns. The first column presents the average percentage cost of each component of the execution time with respect to the thread compute time for the base case. The second column presents the change from the base case for the other protocol. In these tables *Barriers* refers to the total barrier cost, whereas Barrier Wait refers to the component of this cost until all processors reach the barrier (due to imbalances). Similarly *Locks* refers to the total lock cost and *Lock Wait* to the component up the point where the lock is released and can be granted to the next processor. Finally *PFetch Time* is the average cost of a page fault.

Cost Breakdown	FFT		LU-contiguous		Ocean-contiguous		Barnes-rebuilt		Radix		Volrend		Water-nsquared		Water-spatial	
Protocol	9%	-2%	1%	0%	12%	-11%	8%	-3%	4%	-2%	7%	-4%	5%	-3%	2%	-1%
I/O Wait	12%	-8%	2%	-1%	16%	-16%	6%	-2%	6%	-4%	6%	-4%	6%	-4%	2%	-2%
Barriers	29%	2%	91%	-33%	441%	-373%	28%	-11%	171%	296%	2%	-1%	24%	-3%	67%	-10%
Locks	0%	0%	0%	0%	1%	1%	162%	-26%	20%	243%	61%	-35%	41%	1%	15%	3%
Data Wait	109%	-26%	17%	-6%	139%	-131%	32%	-11%	336%	279%	36%	-25%	16%	-11%	29%	-9%
CPU Stall	89%	34%	42%	3%	203%	75%	12%	0%	10%	1%	17%	-1%	15%	0%	17%	-1%
Compute	100%	0%	114%	0%	100%	0%	100%	0%	100%	0%	133%	-8%	104%	0%	103%	0%
Barrier Wait	24%	4%	60%	-5%	344%	-304%	26%	-10%	164%	282%	1%	-1%	18%	2%	57%	-9%
Lock Wait	0%	0%	0%	0%	0%	1%	150%	-24%	14%	75%	53%	-31%	17%	11%	14%	3%
PFetch time	21625	28443	20695	9417	20069	20156	20850	8232	172213	1140274	24025	4480	23765	11311	28062	59742
# PFetches	2705	-1793	258	-135	4591	-4434	3243	-1740	2082	-1565	1136	-808	394	-298	511	-399
# Local Locks	0	0	0	0	0	5	1	511	1	7	0	223	0	748	0	6
# Remote Locks	0	0	0	0	15	-6	2226	-514	47	-7	430	-200	1170	-748	19	-5

TABLE 4
Changes in Protocol Costs from AURC-1 *to* AURC-4.

Cost Breakdown	FFT		LU-contiguous		Ocean-contiguous		Barnes-rebuilt		Radix		Volrend		Water-nsquared		Water-spatial	
Protocol	57%	-31%	11%	-6%	82%	-79%	8%	4%	5%	2%	7%	2%	5%	0%	13%	-9%
I/O Wait	11%	-7%	2%	-1%	16%	-16%	8%	-3%	1233%	-1231%	9%	-4%	8%	-4%	3%	-3%
Barriers	42%	-10%	100%	-39%	426%	-355%	235%	-169%	2142%	-1849%	2%	-1%	23%	3%	66%	-32%
Locks	0%	0%	1%	-1%	2%	0%	337%	-76%	9%	-7%	118%	-15%	57%	-2%	30%	-9%
Data Wait	153%	-49%	29%	-14%	231%	-222%	50%	-21%	588%	-469%	30%	-12%	13%	-7%	36%	-26%
CPU Stall	63%	39%	37%	4%	181%	94%	13%	-1%	10%	1%	18%	1%	15%	0%	10%	3%
Compute	100%	0%	114%	0%	100%	0%	100%	0%	100%	0%	143%	6%	104%	0%	103%	0%
Barrier Wait	28%	1%	70%	-12%	364%	-319%	181%	-116%	2012%	-1741%	1%	0%	20%	5%	57%	-25%
Lock Wait	0%	0%	1%	-1%	1%	1%	289%	-62%	2%	-1%	79%	-4%	18%	7%	26%	-7%
PFetch time	32395	27367	31978	6890	33270	10448	28459	11698	298878	-38977	20077	17310	19687	18655	34089	5742
# PFetches	2541	-1569	285	-162	4647	-4492	3298	-1778	2056	-1584	1033	-699	382	-282	543	-408
# Local Locks	0	0	0	0	0	5	1	528	1	6	0	192	0	725	0	7
# Remote Locks	0	0	0	0	15	-6	2228	-531	47	-6	438	-188	1170	-725	17	-4

TABLE 5
Changes in Protocol Costs from HLRC-1 *to* HLRC-4.