Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

J. Parallel Distrib. Comput. 63 (2003) 1257–1276

Journal of Parallel and Distributed Computing

http://www.elsevier.com/locate/jpdc

# Shared virtual memory clusters: bridging the cost-performance gap between SMPs and hardware DSM systems ☆

Angelos Bilas,[a,*] Dongming Jiang,[b] and Jaswinder Pal Singh[b]

[a] *Computer Science Department, University of Crete, P.O. Box 2208, Heraklion, GR 714 09, Greece*
[b] *Department of Computer Science, 35 Olden Street, Princeton University, Princeton, NJ 08544, USA*

Received 19 October 2001; revised 26 June 2003

**Abstract**

Although the shared memory abstraction is gaining ground as a programming abstraction for parallel computing, the main platforms that support it, small-scale symmetric multiprocessors (SMPs) and hardware cache-coherent distributed shared memory systems (DSMs), seem to lie inherently at the extremes of the cost-performance spectrum for parallel systems. In this paper we examine if shared virtual memory (SVM) clusters can bridge this gap by examining how application performance scales on a state-of-the-art shared virtual memory cluster. We find that: (i) The level of application restructuring needed is quite high compared to applications that perform well on a DSM system of the same scale and larger problem sizes are needed for good performance. (ii) However, surprisingly, SVM performs quite well for a fairly wide range of applications, achieving at least half the parallel efficiency of a high-end DSM system at the same scale and often much more.


*Keywords:* Shared virtual memory; Clusters; System area networks; Scalability; Parallel applications

## 1. Introduction

As the shared address space (SAS) abstraction is gaining popularity in both low-end SMPs and high-end distributed shared memory systems (DSMs) there is increasing interest in providing the attractive SAS programming model on systems that bridge the cost-performance gap between SMPs and DSM systems. Clusters of workstations, PCs, or symmetric multiprocessors (SMPs) are becoming important platforms for parallel computing, and a promising candidate for bridging this gap.[1] In this way, SMPs, clusters, and hardware DSM systems are emerging as the three major types of platforms available to users of multiprocessing, offering different cost-performance ratios. Users would like to write parallel programs once and run them efficiently on all types of platforms, and programming models that do not allow this may be at a disadvantage. Thus, despite (and in fact, because of) the success of SMPs, and hardware-coherent DSM systems, software-shared memory clusters remain an important topic of research.

Table 1 presents application speedups for two and four processors on a Quad SMP system (a single node of our cluster) and for 64 processors on the SGI Origin 2000 [26]. We see that applications perform well on both platforms. However, SMPs do not scale to larger numbers of processors, and DSM systems, although they can be scaled down to smaller numbers of processors, they are still very costly compared to SMPs and clusters. Fig. 1 depicts how the cost-performance configuration space is roughly covered by the different platforms.

Supporting a programming model gives rise to a layered communication architecture that is shown in Fig. 2. The lowest layer is the *communication layer*,

☆ Part of the results in this work have been presented in [24].

*Corresponding author. Fax: +30 2810391661. Also with the Institute of Computer Science, Foundation of Research and Technology—Hellas.

*E-mail addresses:* bilas@csd.uoc.gr (A. Bilas), dj@cs.princeton.edu (D. Jiang), jps@cs.princeton.edu (J.P. Singh).

[1] The Quad SMP nodes in our cluster cost between US$15K-20K each, whereas a 64-processor SGI Origin2000 has an estimated cost of about US$2.5M. The full 64-processor cluster that we use costs about US$300K (assuming today's 450 MHz processors as opposed to 200 MHz in our system), for a configuration similar to the Origin. Prices of first quarter of 1999.

0743-7315/$ - see front matter © 2003 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2003.08.001

Table 1
Application speedups for the base problem size on a Quad SMP for two and four processors and on the SGI Origin 2000 for 64 processors

| Application | Speedup (base problem size) | | |
|---|---|---|---|
| | SMP-2 | SMP-4 | O-64 |
| FFT | 1.55 | 2.05 | 45.84 |
| LUcontiguous | 1.99 | 3.77 | 45.39 |
| OceanRowwise | 1.79 | 1.97 | 9.41 |
| BarnesSpatial | 1.85 | 3.47 | 36.89 |
| RadixLocal | 1.80 | 2.31 | 21.57 |
| SampleSort | 0.751 | 1.161 | 18.63 |
| WaterNsquared | 1.90 | 3.61 | 44.71 |
| WaterSpatial | 1.80 | 3.37 | 35.90 |
| WaterSpatialFL | 1.87 | 3.63 | N/A |
| Volrend | 1.92 | 3.89 | 22.65 |
| Raytrace | 2.01 | 3.98 | 36.68 |

See Sections 3 and 4 for information on applications and problem sizes.

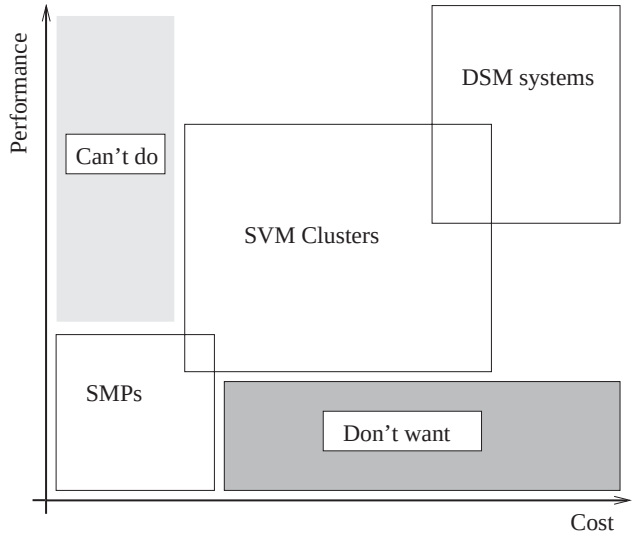


Fig. 1. Pictorial view of the cost-performance area that clusters can cover.

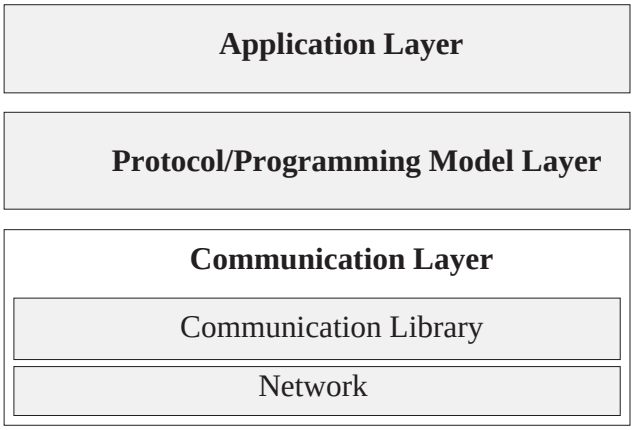


Fig. 2. The layers that affect the end application performance in software-shared memory.

which consists of the communication hardware and the low-level communication library that provide basic messaging facilities. Next is the *protocol* layer that provides the programming model to the parallel application programmer. In this paper we are interested in all software DSM protocols and we focus on page based shared virtual memory (SVM). Finally, above the programming model or protocol layer runs the *application* itself.

A lot of research has been done in the area of software-shared memory protocols and systems for clusters, in optimizing communication layers, and recently in structuring applications for software-shared memory. By putting these together, several systems have achieved reasonable performance on clusters with small processor counts [14,29,36,37,42]. Protocols and systems have also been developed to exploit the features of network interfaces in system area networks to enhance software-shared memory performance [5,22,35,42].

All this research has helped narrow the performance gap between SVM on clusters and hardware DSM systems for an expanding range of applications at the 16-processor scale. It is now time to examine whether performance can scale-up to larger processor counts, especially on a wide range of realistic applications that are known to scale on hardware cache-coherent machines. Since clusters used for high-performance computing typically use SMP nodes, we examine a 64-processor cluster composed of 16 four-way PentiumPro SMP nodes connected by a Myrinet [9] point-to-point system area network, running an extended, high performance, user-level virtual memory-mapped communication library (VMMC) [12]. We use the *GeNIMA* home-based, page-level SVM system, which adds general purpose data movement and synchronization support in the network interface and exploits it in the SVM protocol [5]. In this work we enhance the *GeNIMA* system to improve performance and memory scalability.

Examining scalability is challenging because the true potential for scalable performance cannot be understood via systems research alone; it requires also examining how applications can be restructured to perform and scale better as well as the programming or algorithmic difficulty this entails, and "opening up" the systems and applications together. Otherwise, limitations of the applications will affect the conclusions. It was shown in [25] that starting from SPLASH-2 applications (that are already optimized for moderate-scale hardware-coherent systems), substantial algorithmic application restructuring was needed in several cases to achieve good performance under SVM, at least for a simulated 16-processor, uniprocessor-node cluster. We therefore use both the original SPLASH-2 versions of

the applications as well as these restructured versions, and even some new or further restructured versions as needed. In fact, for the cases where the original SPLASH-2 versions perform very poorly even on a small-scale system [5,25], we do not present results for those versions here but use the restructured versions from [25] as our "original" versions.

To establish context, we loosely compare the speedups achieved for all applications and versions with those obtained in [26] on a 64-processor SGI Origin2000 hardware-coherent machine. Since the cluster is of lower cost (by almost an order of magnitude) than the Origin2000, we say an application performs or scales well if it delivers 50% of the speedup that the Origin2000 does for the same problem size and processor count.

The main high-level questions we ask are: (i) does performance scale well for a set of reasonably chosen "basic" problem sizes for which it scales well on the Origin2000; (ii) if not, does running larger (but still realistic) problem sizes easily solve the scaling problem; (iii) if not, does application restructuring help solve the scaling problem, and how difficult is it; (iv) what are the key remaining system bottlenecks and in which layer (Fig. 2) do they appear. Answering these questions helps us identify the next critical areas of research in SVM clusters.

Our highest-level conclusion is that while the level of application restructuring needed is quite high compared to applications that perform well on a hardware-coherent system of this scale, and larger problem sizes are needed for good performance, SVM, surprisingly, performs quite well at the 64-processor scale for a fairly wide range of applications.

The rest of the paper is organized as follows. Section 2 provides the necessary background on the specific family of shared memory protocols we use and discusses *GeNIMA*, the protocol we use. Section 3 presents the platform we use in this work and the performance of the base system. Section 4 presents our main scalability results. Finally, Section 5 presents related work and Section 6 presents concluding remarks.

## 2. Shared virtual memory protocol

The SVM protocol we use, *GeNIMA* [5], is based on home-based lazy release consistency (HLRC) [46]. In this section we briefly review lazy release consistency (LRC) and HLRC and then discuss basic design and implementation issues in the context of HLRC and provide an overview of GeNIMA.

### 2.1. Home-based lazy release consistency protocol

HLRC [46] is a variation of LRC [30]. LRC is a particular implementation of release consistency (RC). RC is a memory consistency model which 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. LRC [30] 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 causal-related events. To reduce the impact of write–write false sharing LRC has been most commonly been used with a software or hardware supported multiple-writer scheme.

In a software-based multiple writer 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 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 traditional implementation of LRC as TreadMarks [29], 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.

Alternatively, a hardware-based multiple-writer scheme can be also used in conjunction with a LRC protocol. The AURC protocol [23] implements LRC without performing any diff operations by taking advantage of the SHRIMP multicomputer's automatic update hardware mechanism [7,8]. Automatic update provides write-through communication between local memory and remote memory. This mechanism is used to merge writes performed by multiple writers on different copies of the same page at a central location (referred to as the *home* of the page).

HLRC [46] is a variation of LRC [30] which uses a software write detection and diff-based write propagation scheme. HLRC uses a "home" node for each shared page at which updates from writers of that page are collected. This protocol is inspired by the design of Automatic Update Release Consistency (AURC) [22]. The difference is that updates are detected in software and propagated using diffs unlike in AURC, so HLRC does not require any special hardware support.

In HLRC diffs are computed at the end of each time interval (for instance, during a release operation) for all pages that were updated in that interval. Once created, diffs are eagerly sent to the home nodes of the pages where they are immediately applied. Writers can discard their diffs as soon as they are dispatched. Home nodes apply arriving diffs to the relevant pages as soon as they arrive, and immediately discard them. A vector time-stamp approach is used to preserve the LRC model [30]. During a page fault following a coherence invalidation (performed at acquires), the faulting node fetches the up-to-date version of a whole page from the home node.

HLRC has some advantages when compared to standard LRC: (i) accesses to pages on their home nodes cause no page faults, (ii) diffs do not need to be created when the writer is the home node, (iii) non-home nodes can always bring their shared pages up-to-date with a single round-trip message, and (iv) protocol data and control messages are much smaller than under standard LRC. The disadvantages are that whole pages are fetched and good home assignment may be important. An implementation of HLRC on the Intel Paragon has showed that HLRC outperforms a traditional LRC implementations, at least on the platform and applications tested, and also incurs much smaller memory overhead [46]. HLRC achieves ordering of all messages completely in software. Finally, 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. Next, we discuss basic design and implementation issues for HLRC-SMP our base protocol that support SMPs.

### 2.2. High-level design alternatives

The first important issue in HLRC-SMP is how to take advantage of hardware cache-coherence within SMP nodes. There are three major alternatives for managing the ''processes'' and their view of the data within an SMP node. One extreme for each process have its own separate physical address space and page table and not share any state with the other processes in the SMP through the hardware-coherence mechanism. All processes, within or across SMP nodes, communicate and synchronize via the SVM software protocol, and the hardware within an SMP is used only to accelerate the underlying message passing used by the SVM library. The other extreme is to use threads within an SMP. The threads on and SMP share a physical address space and the page table. Hardware sharing and synchronization are used within an SMP, but the disadvantage is that all threads in an SMP have the same view of the data: when one thread decides to invalidate a page according to the consistency model, that page is automatically (''eagerly'') invalidated for the other threads as well. The required global TLB flushes for each page invalidation can prove to be expensive in this scheme. However, since all threads within the same SMP have the same state for a given page, local acquire operations are very cheap. The first alternative has been shown to perform poorly in [42] due to high overheads.

HLRC-SMP uses an intermediate model that achieves both hardware sharing and laziness within the SMP node. Hardware sharing and (most) synchronization is achieved by having the processes in an SMP share a physical address space. Laziness is achieved by allowing the processes to have separate page tables so they can each have their own view of the state of a given page. The challenge is to support laziness within a node with little software overhead for intra-node activity. Our simulation results show that the threads model leads to more page faults and invalidations in some irregular applications. For example, Barnes exhibits a 10% greater number of page faults in the thread model when compared to the process model.

Another major design issue is how protocol processing should be performed for incoming remote requests. One approach is to dedicate one or more processors in the SMP solely to protocol processing, having them poll for incoming requests. However, protocol activity is usually small and this approach is very wasteful of computational resources. Alternatively, in SVM systems where code instrumentation of compute processes for polling is not used, interrupts are delivered to some process at protocol requests. The alternatives include using a separate process (which can run on any compute processor) for protocol processing or having one of the compute processes perform all protocol processing, along with useful their computation. HLRC-SMP uses the first approach, since it allows more opportunities for load balancing the protocol processing across processors and because it leads to a cleaner implementation. Finally, HLRC-SMP implements hierarchical barriers and TLB invalidations are avoided with a scheme very similar to the 2-way-diffing described in [42].

### 2.3. Implementation issues

This section presents the data structures that allow us to easily implement the above choices in the HLRC-SMP and how the synchronization and page fault operations are managed using them.

#### 2.3.1. Data structures

To illustrate the data structures, we first need to define some key terms. The time during the execution of a parallel program is broken into *intervals*. With uniprocessor nodes, intervals are maintained on a per-process basis and numbered in a monotonically increasing sequence. An interval is the time between two consecutive synchronization operations by a single

process. In HLRC-SMP, intervals are maintained on a per-node, and not a per-process basis. The interval number is incremented when *any* process within the SMP performs a synchronization operation. Each process maintains a list called the *update-list*, which records all the pages that have been modified by this process in the current interval. When any synchronization operation happens, the process performing this operation ends an interval for the node.

When a process ends its interval, its update-list is placed in a data structure called the *bins* (see Fig. 3). Once a process "grabs" a bin, no other process will be able to write to the same bin. 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 process. Thus, the bins data structure scales with the number of nodes and not the number of processors in the system, providing better scalability 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 medium-scale DSM machines.

Each SMP node contains a copy of the entire bins data structure. The entry in a given bin (say interval 2, node 1) will be the same in every node's copy of the bins data structure. However, for a particular node A, when the entry for another node B for a particular interval will appear in A's bins depends on when A does a causally related acquire from B and receives them. Essentially the bins inform us which pages were modified in relation to which synchronization operation, and in which order, thus telling us which pages need to be invalidated during a synchronization operation to maintain consistency. This information is also used to infer version information about the pages, so that when we fetch pages from the home node, we know which versions to demand.

So far the bins data structure does not provide different processes within an SMP with different views of the world. To provide laziness within a node, only a simple per-process data structure is required. We refer to this as the *view vector*. Essentially this is the "view of the world" that a process has. This vector maintains the information regarding what portion or height of the bins for different nodes has been "seen" (i.e. the invalidations corresponding to those intervals from different nodes have been performed) by this particular process. When one process fetches new bin information from another node, on an acquire the new information is available to all processes in its SMP node. However, this and the other processes do not *act* on all this information immediately. Processes only invalidate pages for their own acquires, and only the pages (i.e. the entries in the bins) that they are required to "see" (as dictated by the last releaser of the lock).

### 2.3.2. Operations

Let us see how the bins and the view vectors work in conjunction with various synchronization operations.

*2.3.2.1. Lock acquires.* During a remote lock acquire operation (i.e. when the requested lock is available at a remote SMP node), the requester sends over a vector which indicates what bins are currently present at this node (i.e. a vector that specifies the height of its bins). Next, any portion of the bins that are not available at the requester but exist at the releaser are sent back in the form of *write notices*, along with the view vector of the process that released the lock (as it was when the lock was released). Finally, the requesting process matches its view vector with the lock releaser's view vector, by invalidating, for itself only, all the pages indicated in the bins that are "seen" by the releaser's view vector but not by the requester. This operation is illustrated in Fig. 4. In this example process 1 on node 1 is acquiring a lock which was previously released by process 1 on node 0.

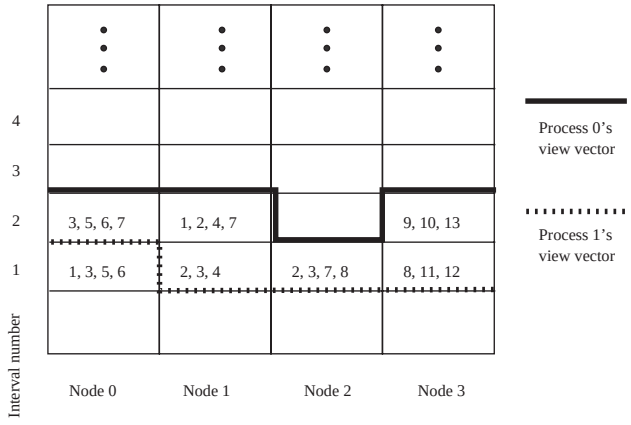


Fig. 3. The bins data structure.

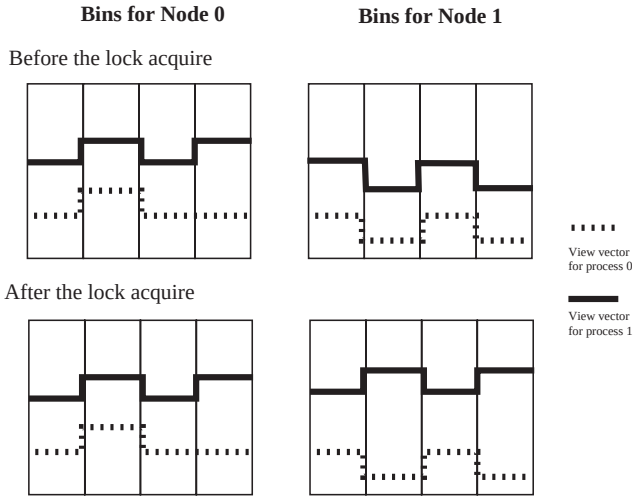


Fig. 4. A remote lock acquire.

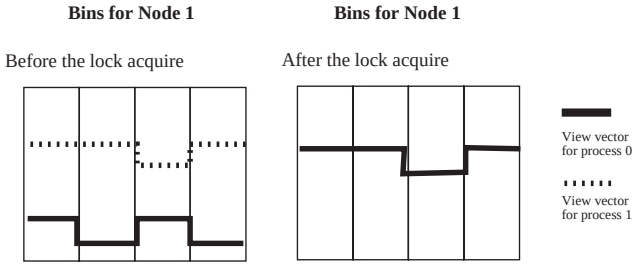


Fig. 5. A local lock acquire.

A local lock acquire operation (i.e. the acquirer and releaser reside on the same SMP) is completely local with no external or internal messages. All that is required is the matching of the requesting process' view vector with the releaser's, and the necessary invalidations, so there is only a small amount of protocol processing. As shown in Fig. 5, this operation only involves bins available at this node and no data transfers across nodes are involved.

*2.3.2.2. Barriers.* For a barrier operation, the processes within a node barrier first and update the bins. Then, all the bins that have been generated at this node and have not 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. Now all the bins at all nodes are filled to the same height. 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.

*2.3.2.3. Local copy updating.* Finally, let us see how data is fetched when a more recent version of a page is needed. The home node of a page always has the most current version of the page. However, updates (in the form of diffs) may be in flight, so it is desirable when requesting a page to specify the version that is absolutely necessary. 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 each node for which the updates are currently available at the home. One may think of this vector as a "version" of the page. On a page fault, the requester sends to the home a lock time-stamp, corresponding to its last lock acquire, which indicates to the home what the flush time-stamp of the page should at least be, in order to ensure that all relevant changes to the page by other processors are in place. If the flush time-stamp is less up to date then the lock time-stamp, then diffs must be on their way so the home waits for the diffs to arrive before satisfying the page request.

During the page fetch operation, there is one major complication. If there are outstanding updates to a page (which have not been communicated to the home via diffs) or if other processes in the SMP are actively writing to the page in question, fetching the page in its entirety will destroy these updates to the page. To avoid this problem we use a "dirty bit" which detects this scenario. This bit is set on the first write by any process in the SMP to that page. It is cleared when a diff is performed on the page and there are currently no active writers to the page. When a page is observed to be "dirty" we perform a *safe page fetch*. Essentially a safe page fetch diffs the page returned by the home with the twin of the page and applies these updates to the current local page as well as the current twin for this page. This scheme is similar to the "2 way diffing" used in Cashmere-2L [42].

### 2.4. GeNIMA

*GeNIMA* [5] uses general purpose network interface support to significantly improve protocol overheads and narrow the gap between SVM clusters and hardware DSM systems at the 16-processor scale. We now describe a minimal set of extensions to the network interface that are used by *GeNIMA* to remove the need for asynchronous protocol processing, and how the resulting message and protocol handling differs from that of the base protocol.

The base protocol (HLRC-SMP), uses the communication layer only as a fast interrupt-based messaging system. Each protocol request causes an interrupt that schedules the protocol process on one of the processors. The incoming protocol requests that require interrupts in the base protocol are (i) page fetches, (ii) lock acquisition, and (iii) diff application at the home. Other requests that require interrupting host processors in some protocols include page home allocation and migration requests. These however, are infrequent and not so critical for common-case system performance. We now describe a minimal set of extensions to the network interface that can be used to remove the need for asynchronous protocol processing, and how the resulting message and protocol handling differs from that of the base protocol.

#### 2.4.1. Remote deposit

The communication layer we use (VMMC) already allows for data explicitly transferred to a remote node via a send message to be deposited in specified destination virtual addresses in main memory without involving a remote host processor. This is different from transparently updating remote copies of data structures via memory bus snooping, code instrumentation, or specialized NI support [8,17,22,32]. In our implementation, non-contiguous pieces of data are sent directly to remote data structures with separate messages, and are not packed into bigger messages or combined by

scatter-gather support. Many communication systems support this or similar type of operations [8,13,17,21,28].

We use the remote deposit mechanism in all our subsequent protocols to exchange small pieces of control information during barrier synchronization and to directly update other remote protocol data structures (e.g. page timestamps, barrier control information). In addition, there are two major cases where this operation is used:

- First, we use it to propagate *coherence information* in a sender-initiated way rather than in response to incoming messages, without causing interrupts. In traditional interrupt-based lazy protocols coherence information (page invalidations) is transferred as part of lock transfers. When the last owner of a lock hands the lock to another process, it also sends the page invalidations that the requester needs to maintain the release-consistent view of the shared data. Thus, in the base protocol, when the protocol handler asynchronously services a remote acquire, it sends to the requester both the lock (mutual exclusion part) and the page invalidations (coherence information).

  The mutual exclusion and the coherence information parts can be separated. We will see further motivation for this when we examine servicing lock acquire messages without interrupts, so the host processor at the last owner does not even know about the lock acquire. In *GeNIMA* we *propagate* coherence information eagerly to all nodes at a release, using remote deposit directly into the remote protocol data structures. Invalidations are still *applied* to pages at the next acquire, preserving LRC.
- Second, we use the asynchronous send mechanism with remote deposit to remotely apply diffs and hence update shared *application data* pages at the home nodes. In the base protocol diffs are propagated to the home at the next incoming acquire of a lock. Diffs for the same page are packed in a single message and then sent to the home, where they interrupt the processor and are applied to the page by the protocol handler. In *GeNIMA* when the local processor computes a diff, instead of storing it in a local data structure and then sending this diff data structure to the home of the page, it directly sends each contiguous run of different words to the home as it compares the page with its twin. We call this method of computing and applying the differences in shared data *direct diffs*.

  Direct diffs save the cost of packing the diff, interrupting a processor at the home of the page and having it unpack and apply the diff on the receive side. However, they may substantially increase the number of messages that are sent, since they introduce one message per contiguous run of modifications within a page rather than one message per page (or multiple pages) that has been modified.

  Since synchronization points do not involve interrupts any more (as we shall see shortly), diffs must now be computed at release points rather than incoming acquires. However, if another processor in the same node as the releaser is waiting to acquire the lock next, then no diffs need to be computed. Thus, diff computation is done with a hybrid method that is eager for synchronization transfers across nodes and potentially lazy for transfers within nodes.

In all cases, the remote deposit (send) messages used are asynchronous. Thus, blocking of the sending processor is avoided, except when the post queue between the processors and the network interface is full and must be drained before new requests are posted.

#### 2.4.2. Remote fetch

We extend the communication system to support (in NI firmware) a remote fetch operation to fetch data from arbitrary exported remote locations in virtual memory to arbitrary addresses in local virtual memory. Again, the remote fetches must be for contiguous data in our implementation, but there is little occasion for non-contiguous fetches in the protocol.

Remote pages are fetched in the base protocol by sending a request to the home node. We use the remote fetch operation to avoid interrupts at page fetches. When a remote page is needed, the local processor first requests the timestamp of the remote page and then immediately requests the page itself. The request messages are asynchronous, so the request for the page is sent before the timestamp arrives. If the timestamp is determined to be incorrect, i.e. the necessary diffs have not been applied at the home, the requester retries.

Another important advantage of the remote fetch operation is that it significantly enhances protocol scalability. When remote deposit is used to transfer pages in VMMC in response to a request, each home node needs to be able to send its pages to every node in the system. With memory-mapped communication layers, this requires that each node, as a requester, export (and pin) all the shared pages in the entire application, limiting the amount of shared memory. With the remote fetch operation, the requester itself fetches updated page versions from the home, so the requester rather than the home needs to "directly" access remote pages. Thus, each node needs to export (and pin) only those shared pages for which it is the home. Other uses of a remote fetch operation are possible as well, e.g. for fetching protocol data as mentioned earlier.

Remote fetch can also be used instead of remote deposit to avoid interrupts at coherence information propagation: a processor can "pull" the necessary write

notices with a point-to-point remote fetch operation at lock acquires rather than pushing it to all nodes at a release. The total traffic is usually about the same in both cases since invalidations for all intervals generally do need to be sent to all nodes in the system at some point, and propagating this information at the releases or at the acquires does not change the number of intervals. However, the former method can increase the number of messages: If multiple intervals have to be communicated from a releaser to an acquirer, this will be done via one broadcast operation at each release rather than a single fetch of all intervals at the acquire. Thus, the former approach increases the remote release cost while the latter approach increases the remote acquire cost. We choose the former method rather than remote fetch for this purpose, since it results in smaller messages that are easier to pipeline in the node and NI and since it spreads out the traffic over a longer period of time throughout the execution of the application. Thus, while the protocol is still lazy in applying invalidations, the coherence information is propagated eagerly.

Interestingly, direct diffs require that pages be fetched with remote fetches and retries rather than by involving the home processor. The reason is that since direct diffs do not interrupt or involve a host processor at the home at diff application, home processors do not know when they have the updated version of a page and are ready to service queued page requests. Remote fetches do not rely on home processors having this knowledge, since the requester retries whenever it fails to fetch the right version of a page. This is why we will present results for direct diffs only after presenting those for remote fetch.

#### 2.4.3. Network interface locks

With coherence information propagation already separated from mutual exclusion as described above, the communication layer is extended to provide support for mutual exclusion in the NI as well. Lock acquisition and release for mutual exclusion become communication system rather than SVM protocol operations, and no host processors other than the requester are involved.

In the base protocol, lock synchronization is implemented as follows: Every lock is statically assigned a home. When a process needs to acquire a lock, it sends a message to the home of the lock. The home forwards the message to the last owner and the owner releases the lock to the requester. The requests at the home and at the last owner are both handled using interrupts, and may involve protocol activity such as preparing and propagating coherence information as well. The host processor at the home of each lock is in charge of maintaining a distributed linked list of nodes waiting for the lock. The last owner keeps the lock until another processor needs to acquire it.

The implementation of locks in the network interface firmware is similar to the algorithm described earlier. However, no coherence information is involved and the distributed lists for locks are maintained in the network interface processors, without host processor involvement or interrupts. Coherence propagation is decoupled and managed as described earlier. Associated with each lock is one timestamp, which is interpreted and managed by the protocol. The network interface does not need to perform any interpretation or operations on this timestamp, but the current implementation requires that this piece of information be stored and transferred as part of the lock data structure in the network interface.

On the protocol side, each process knows what invalidations it needs to apply at acquires by looking at protocol timestamps that are exchanged with the locks. Flags are used to ensure that invalidations for each interval have reached the node before they are applied. The only requirement in the communication layer is in-order delivery of messages between two processes. There are no requirements for global or other strict forms of ordering.

In the locking mechanism we use, we move all the functionality for mutual exclusion into the communication layer, including a lock algorithm. Alternatively, the communication layer or NI can simply provide remote atomic operations, and the locking algorithm can be built into the protocol layer while still avoiding interrupts. This makes the NI support simpler, and hence more likely to be implemented in hardware in commodity NIs. It also allows flexibility in the locking algorithm chosen at the protocol level. The performance tradeoffs between the two approaches are unclear, and more investigation is necessary.

## 3. Platform

Each node in our cluster is an Intel PentiumPro SMP. It contains four 200 MHz processors, a shared snooping-coherent bus, and 512 MB of main memory. Each processor has separate 8 KB instruction and data caches and a 512 KB unified four-way set associative second-level cache. In the analysis of the results, we explicitly state cases where the small caches may result in artificially high speedups due to the reduction of the working set size in the parallel execution. The operating system is WindowsNT 4.0.

Each SMP node connects to a Myrinet [9] system area network interface (NI) via a PCI bus. The 16 Myrinet NIs are connected together via a single 16-way Myrinet crossbar switch, thus minimizing contention in the interconnect. Each NI has a 33 MHz programmable processor and connects the node to the network with two unidirectional links of 160 MByte/s peak bandwidth each. Actual node-to-network bandwidth is

usually constrained by the PCI bus (133 MBytes/s) on which the NI sits.

The hardware cache-coherent system we use in this study is an SGI Origin 2000 [33], containing 64 300 MHz R12000 processors and running Cellular IRIX 6.5.7f. The 64 processors are distributed in 32 nodes, each with 512 MBytes of main memory, for a total of 16 GBytes of system memory. The nodes are assembled in a full hypercube topology with fixed-path routing. Each processor has separate 32 KByte instruction and data caches and a 4 MByte unified two-way set associative second-level cache. The main memory is organized into pages of 16 KBytes. The memory buses support a peak bandwidth of 780 MBytes/s for both local and remote memory accesses. A more description of the relevant features of the Origin2000 system can be found in [26].

In the next subsections we describe the system that we have built, following the layered architecture of Fig. 2, in a bottom-up fashion.

### 3.1. Communication layer

The communication layer we use in this system is VMMC for the Myrinet network [12]. VMMC provides protected, reliable, low-latency, high-bandwidth user-level communication. The key feature of VMMC that we use is the remote deposit capability. The sender can directly deposit data into exported regions of the receiver's memory, without process (or processor) intervention on the receiving side. The communication layer has been extended to support a remote fetch operation and system-wide network locks in the network interface without involving remote processors, as described in [5]. It is important to note that these features are very general and can be used beyond SVM protocols. Table 2 shows the cost for the basic VMMC operations.

Table 2
Basic VMMC costs

| VMMC operation | Cost |
|---|---|
| 1-word send (one-way latency) | 14 μs |
| 1-word fetch (round-trip latency) | 31 μs |
| 1-page send (one-way latency) | 46 μs |
| 1-page fetch (round-trip latency) | 105 μs |
| Maximum ping–pong bandwidth | 96 MBytes/s |
| Maximum fetch bandwidth | 95 MBytes/s |
| Notification | 42 μs |
| Remote lock acquire | 53.8 μs |
| Local lock acquire | 12.7 μs |
| Remote lock release | 7.4 μs |
| Host overhead for asynchronous send/fetch | 2–3 μs |

All send and fetch operations are assumed to be synchronous, unless explicitly stated otherwise. These costs do not include contention in any part of the system.

### 3.2. GeNIMA performance and scalability extentions

Since our larger infrastructure used in this research supports the WindowsNT operating system, we ported the *GeNIMA* system from Linux to WindowsNT and switched from the process to the thread model. This has many implications for the protocol, since threads share the same address space (and the same page table in the operating system). To obtain the final version of the protocol we use here, we added certain protocol-level optimizations to the original *GeNIMA* system to enhance performance and memory scalability:

- On the performance side, we schedule operations in barriers to achieve maximal overlap of useful work, communication, and idle time. The cost of `mprotecting` pages was found to be substantial at barriers, so we protect ranges of contiguous pages with reduced per-page protocol processing (per page checks, etc.), in the common case. We reduce the cost for local lock acquires and releases by making sure there is no need to manipulate the OS page table and SVM protocol data; acquiring and releasing a local lock is practically equivalent with a local *test_and_set* operation over the memory bus. Finally, we modify the protocol to not broadcast the write notices at lock releases. Instead, each processor uses a remote fetch operation to get the write notices at the next acquire operation.
- On the memory side, most LRC-based SVM systems have very large extra memory overhead for keeping write notices and diffs which greatly restricts the sizes of problems that can be run. By propagating diffs to the home at release operations, a home-based protocol dramatically reduces the memory overhead for diffs compared to traditional LRC protocols. However, memory overhead for write notices is still large, which is especially a problem for applications that exhibit a lot of lock synchronization (barriers generate write notices too, but global garbage collection when needed at barriers addresses this problem). To reduce these requirements we use a scheme for managing dynamically the protocol data structures that hold the write notices even when they are accessed directly by remote nodes. More intrusive solutions would involve different methods for garbage collection that may introduce additional overheads (messages, protocol processing, etc.), or more eager protocols, and we do not examine them here.

With memory-mapped communication systems, another important issue is the amount of memory that needs to be pinned. With send-based communication systems, application and protocol data structures are updated remotely with direct-deposit operations and thus need to be pinned. With the

Table 3
Application speedups on 16 processors on three systems: (i) *GeNIMA* on Linux, (ii) *GeNIMA* on WindowsNT for this work, and (iii) SGI Origin2000, a hardware cache-coherent multiprocessor

| Application | Problem size | Speedup (16 procs) | | |
|---|---|---|---|---|
| | | SVM | | Origin2000 |
| | | Linux | NT | |
| FFT | 1M points | 6.1 | 4.1 | 13.4 |
| LUcontiguous | 4K × 4K matrix | 12.2 | 12.9 | 14.3 |
| OceanRowwise | 514 × 514 grids | 5.4 | 4.7 | 12.5 |
| BarnesSpatial | 32K particles | 8.8 | 12.2 | 12.9 |
| RadixLocal | 4M integers | 1.4 | 1.6 | 12.4 |
| WaterNsquared | 4096 mols | 9.1 | 9.6 | 13.8 |
| WaterSpatial | 15625 mols | 7.8 | 7.0 | 13.4 |
| Volrend | $256^3$ head | 11.7 | 12.2 | 12.9 |
| Raytrace | 256 × 256 car | 12.8 | 11.3 | 14.2 |

The second column shows the problem sizes that were used.

addition of the remote fetch operation, application and protocol data are updated by the local node, with the use of fetch operations. Thus, only the remotely read data structures need to be pinned. This means that for global application data we need to pin only home pages, and for protocol data structures only the local copy and not the remote replicas.

Table 3 shows that the new system has comparable performance to the Linux version.[2]

### 3.3. Applications layer

We use the SPLASH-2 [44] application suite. This section briefly describes the basic characteristics of each application relevant to this study. 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 [23]. The applications can be divided in two groups, regular and irregular.

*Regular applications*: The applications in this category are FFT [1,43,44], LU [43,44] and Ocean [10,25,41]. 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 almost all local. The applications have different inherent and induced communication patterns [23,44], which affect their performance and the impact on SMP nodes.

*Irregular applications*: The irregular applications in our suite are Barnes [2,19,25,39], Radix [6,20,43], Raytrace [40,44], Volrend [25,34,44] and Water [44].

In this work we use both original versions of several SPLASH-2 applications [23] as well as versions that have been restructured to improve performance on SVM systems [25]. The same restructurings are found to be very important on large-scale hardware-coherent machines [26], so they are not specific to SVM. (i) FFT, LUcontiguous, OceanContiguous, BarnesOriginal, WaterNsquared, and WaterSpatial are the original SPLASH-2 applications. These versions of the applications are already optimized to use good partitioning schemes and data structures, both major and minor, for both hardware coherence and release consistent SVM [20]. (ii) BarnesSpatial, OceanRowwise, RadixLocal, Volrend, and Raytrace are the restructured applications from [25]. With a drastic algorithmic change for one phase of Barnes-Hut, BarnesSpatial substantially reduces the amount of lock synchronization. The restructurings for the others are less intrusive and try to improve data assignment, make remote accesses less scattered, or eliminate unnecessary synchronization. The version of Raytrace we use eliminates a lock that assigns unique ids to rays, resulting in less locking. The restructured version of Volrend uses task stealing but the initial task assignment is such that it results in better load balancing and reduces the need for task stealing. (iii) Finally, WaterSpatial is further restructured in this paper to reduce the amount of locking, and SampleSort is introduced as an alternative algorithm to radix sorting. In WaterSpatial after the computation performed in each phase of the application, global variables are updated by locking. The restructured version, WaterSpatialFL, fuses these updates in one critical section after all phases, reducing the number of lock operations. Sample sort uses two local sorting phases, separated by a short phase to compute splitter keys, and a communication phase to copy a contiguous set of remote keys to a local array (to prepare for the next local sort). The local sorts can use any sequential sorting method; here, radix sort is used. Unlike parallel radix sort, the all-to-all communication is based on remote stride-one reads rather than scattered remote writes, and is therefore better behaved.

## 4. Results

In this section we present results on a per application basis, addressing the questions raised in the introduction for a system of this scale: (i) do these classes of

[2] The large difference in FFT is due to the fact that FFT is affected significantly by memory bus contention. We noted that the compute time for FFT (as computed in our system) differs between Linux and WindowsNT. This can be either due to compiler differences (gcc under Linux vs. Microsoft C under WindowsNT) or the system configuration (queue depth of the memory bus, PCI bus configuration, etc.) performed by the operating system and the drivers. These affect memory bus behavior and result in different compute times (and speedups) under the two systems, however, this is not related to the SVM protocol.

Table 4
Application problem sizes, amount of application shared data in MBytes, and uniprocessor execution times (in seconds) for each problem size

| Application | Basic problem | MB | Uniproc time (s) | | Large problem | MB | Uniproc time (s) | |
|---|---|---|---|---|---|---|---|---|
| | | | SVM | O2000 | | | SVM | O2000 |
| FFT | 1M points | 48 | 4.454 | 2.104 | 4M points | 192 | 16.923 | 9.830 |
| LUcontiguous | 4K × 4K matrix | 128 | 914.954 | 506.759 | 6K × 6K matrix | 288 | 2752.891 | 1720.742 |
| OceanRowwise | 514 × 514 grids | 95 | 257.281 | 6.340 | 1026 × 1026 grids | N/A | N/A | N/A |
| BarnesSpatial | 32K particles | 46 | 1781.031 | 16.600 | 128K particles | N/A | N/A | N/A |
| RadixLocal | 4M integers | 33 | 4.407 | 4.554 | 32M integers | 257 | 32.750 | 40.088 |
| SampleSort | 4M integers | 53 | 2.360 | 5.887 | 32M integers | 412 | 18.891 | 49.779 |
| WaterNsquared | 4096 mols | 3 | 223.515 | 69.315 | 15625 mols | 10 | 3483.391 | 1287.654 |
| WaterSpatial | 15 625 mols | 62 | 119.046 | 33.792 | 32768 mols | 107 | 231.329 | 60.943 |
| WaterSpatialFL | 15 625 mols | 62 | 121.516 | N/A | 32768 mols | 107 | 226.984 | N/A |
| Volrend | $256^3$ head | 22 | 35.812 | 0.888 | $512^3$ head | 168 | 275.906 | 7.160 |
| Raytrace | 256 car | 50 | 22.204 | 12.663 | 512 car | 80 | 86.015 | 50.462 |

The difference in the uniprocessor time between the cluster and the Origin for BarnesSpatial is due to a difference in an application configuration parameter. Since both platforms that we used in this work have recently changed configuration and re-runing the BarnesSpatial would make it inconsistent with the rest of the applications we have decided to use these results instead.

Table 5
Application speedups for each problem size on the SVM cluster and the Origin2000 for 16, 32, and 64 processors

| Application | Speedup | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Base problem size | | | | | | Large problem size | | | | | |
| | S-16 | O-16 | S-32 | O-32 | S-64 | O-64 | S-16 | O-16 | S-32 | O-32 | S-64 | O-64 |
| FFT | 4.13 | 13.40 | 3.94 | 31.72 | 4.65 | 45.84 | 4.39 | 19.36 | 5.50 | 37.12 | 6.86 | 67.20 |
| LUcontiguous | 12.95 | 14.32 | 22.99 | 26.86 | 37.19 | 45.39 | 12.15 | 14.94 | 21.78 | 28.42 | 39.09 | 47.51 |
| OceanRowwise | 5.25 | 12.57 | 6.49 | 20.52 | 7.47 | 9.41 | N/A | 20.47 | N/A | 30.98 | N/A | 44.21 |
| BarnesSpatial | 13.28 | 12.97 | 24.04 | 25.03 | 46.44 | 36.89 | N/A | 14.01 | N/A | 27.12 | N/A | 47.75 |
| RadixLocal | 1.64 | 12.43 | 1.93 | 16.77 | 2.43 | 21.57 | 2.18 | 12.87 | 2.84 | 23.17 | 5.46 | 27.21 |
| SampleSort | 3.12 | 21.43 | 4.21 | 27.08 | 6.20 | 18.63 | 12.44 | 16.06 | 9.82 | 23.72 | 17.86 | 28.48 |
| WaterNsquared | 9.61 | 13.82 | 13.60 | 26.24 | 9.88 | 44.71 | 13.01 | 15.37 | 22.09 | 30.40 | 29.60 | 58.88 |
| WaterSpatial | 7.02 | 13.42 | 8.29 | 25.95 | 6.77 | 35.90 | 8.06 | 14.11 | 8.78 | 27.62 | 8.73 | 48.98 |
| WaterSpatialFL | 8.66 | N/A | 12.33 | N/A | 20.46 | N/A | 10.97 | N/A | 17.13 | N/A | 29.28 | N/A |
| Volrend | 12.26 | 12.97 | 17.57 | 23.88 | 15.43 | 22.65 | 15.09 | 13.45 | 27.00 | 24.60 | 42.91 | 33.32 |
| Raytrace | 11.30 | 14.23 | 17.42 | 27.79 | 23.15 | 36.68 | 14.24 | 15.17 | 26.88 | 30.50 | 45.14 | 52.44 |

applications scale well, (ii) does increasing the problem size help, (iii) are application restructurings effective and how difficult are they, and (iv) what are the important bottlenecks in the system?

In studying the scalability of SVM on clusters we use two problem sizes: (i) a "basic" problem size, which has already been demonstrated to scale reasonably well at the 16-processor scale, and (ii) when the basic problem size does not scale, we use a larger problem size. In general, increasing problem size tends to improve many inherent program characteristics, such as load balance, inherent communication to computation ratio, and spatial locality. On the other hand, it may increase working set size as well. Table 4 presents the problem sizes we use and the size of the application shared data for each problem size. It also shows the uniprocessor execution time on the cluster and the Origin 2000. Table 1 repeats shows the application speedups on a single SMP for two and four processors. Certain entries in the tables are marked as N/A due to the fact that the particular application version was not available on that platform at the time the original experiments were made.

Table 5 presents the application speedups on 16, 32 and 64 processors in our system for each problem size. Fig. 6 presents analytic execution time breakdowns for each application under *GeNIMA*. The components of the execution time are (i) the compute time, which is the time the system spends doing useful work,[3] (ii) the data wait time, which is the time each processor spends waiting on remote memory references, (iii) the lock time, which is the time the system spends acquiring and releasing locks (both local and remote), (iv) the acquire/release time, which is the time each processor spends in acquire/release primitives that are used in

[3] This time includes the local memory reference time and other system activity, like context switches, which may take place while the application is running.

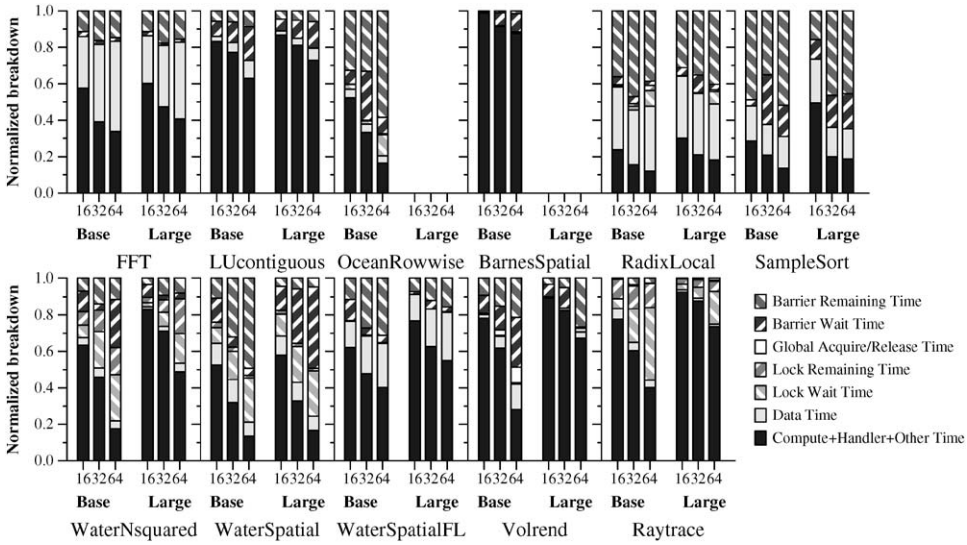


Fig. 6. Average execution time breakdowns for all applications under *GeNIMA*. Each graph presents the breakdown for one application, for all system and problem sizes; the bars on the left of each graph refer to the base problem size and on the right to the large problem size.

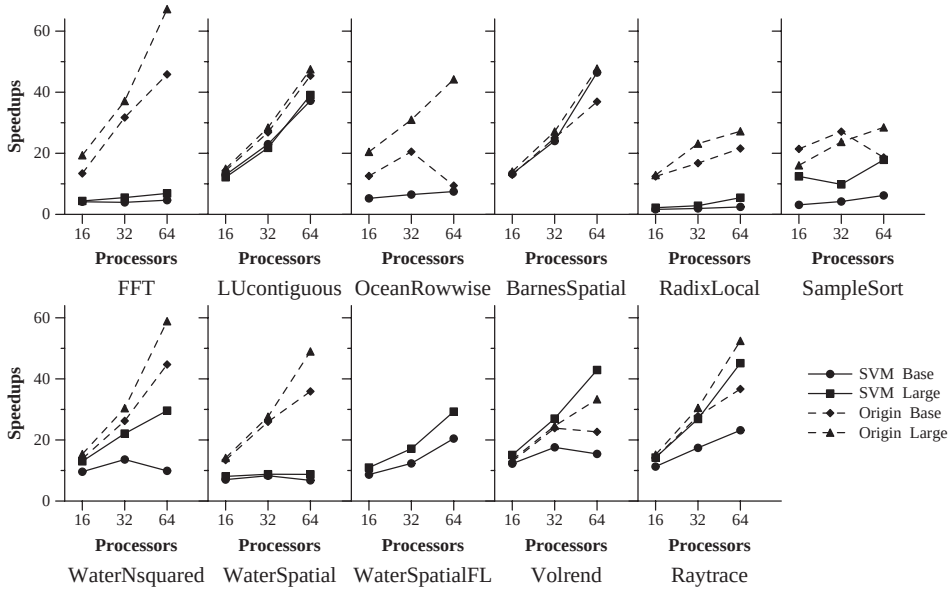


Fig. 7. Speedup curves for the SVM cluster and the hardware cache-coherent system. For each application there are two curves per system, one for the basic and one for the large problem size.

some applications to make them release consistent, and (v) the barrier time, which is the time spent in barrier synchronization. In the next few paragraphs we analyze the behavior of each application and reveal the remaining system bottlenecks.

*FFT* (Fig. 7) is a bandwidth intensive application because of the all-to-all communication during the transpose phase. Moreover, FFT exhibits an additional problem in our platform; when run with multiple processors per node, the memory bus is saturated and the local memory access time is increased. Experiments with dual-processor SMPs that have a 100 MHz bus (and 450 MHz Pentium-III processors) indicate that the memory bus is still saturated. We see that FFT not only exhibits a relatively low speedup at the 16-processor scale, but it scales very poorly as well. The poor scalability of FFT is due to the increased remote and local memory access time due to contention in the network interface and the memory bus of the servicing node. Using a bigger problem size (4M points) improves performance only marginally and scalability is still very poor.

*LUcontiguous* (Fig. 7) performs well, since its communication to computation ratio is very low and scales

reasonably well up to 64 processors. Fig. 6 shows the bottleneck for better speedup is barrier wait time due to imbalances. Using a larger problem size improves both performance and scalability of LU, by reducing barrier synchronization costs. The bigger problems size helps reduce both computational imbalances and barrier processing costs (because computation increases by $O(n^3)$ and the number of barriers by $O(n)$).

*OceanRowwise* (Fig. 7) performs poorly on the base system for two reasons: (i) local memory overhead is high due to capacity misses and contention on the memory bus because of multiple processors within each node and nearest neighbor computation patterns, and (ii) the barrier time is very high due to the large number of barriers in the application. Because of the same reasons, Ocean scales poorly as well. Fig. 6 reveals that compute, data communication, and barrier time do not scale proportionally to the number of processors used. The amount of data exchanged between neighbor processors does not change with number of processors, because processors exchange entire rows of size $n$ at inter-partition boundaries due to the rowwise partitioning. We do not present data for bigger problem sizes for Ocean, since they have unrealistically high memory requirements (for benchmarks).

*Barnes*: By applying a totally different tree-building algorithm to eliminate fine-grained and global locking, BarnesSpatial (Fig. 7) performs and scales well even for the basic problem size. This is partly due to the fact that uniprocessor performance on the cluster suffers from capacity cache misses due the small cache size. Fig. 6 shows the factor that limits scalability is the barrier cost. The barrier cost increases at the 64-processor scale, because of an imbalance in the data wait time. Since BarnesSpatial performs and scales very well with the basic problem size, we do not examine a bigger problem size. Improving performance further seems to require reducing barrier cost, partitioning the space in a way for a better initial load balance, and finding ways to reduce contention on remote page faults. As we have seen, such contention-induced imbalance is a frequent problem given the large granularity and high communication overheads on SVM systems.

*Radix* performs very poorly (Fig. 7). Fig. 6 shows that it suffers from very high data wait and synchronization overheads due to high communication traffic and contention, especially on larger processor counts. Increasing the problem size, helps substantially but overall performance is still very poor. Major improvements will likely require using a different sorting algorithm. We use sample sort as an alternative algorithm to radix sort. In sample sort the local sorting is done twice, so (ignoring memory data access) the parallel efficiency is limited to 50%. In spite of this algorithmic disadvantage, sample sort outperforms radix sort on the SVM cluster, and scales much better, by substantially reducing data communication overhead and synchronization time on locks. The remaining bottleneck in sample sort is high barrier cost (Fig. 8).

*Raytrace* performs well on 16 processors with a speedup of 11.3. Moreover, it scales reasonably to 32 and 64 processors for the basic problem size. Fig. 6 shows that the main reason for the losses in performance is imbalances in the compute time and high-lock synchronization costs. Given that remote lock acquires are substantially more expensive than local lock acquires, there are two issues that affect the cost of lock synchronization: The number of remote versus local lock acquires and the order in which locks are acquired. Moreover, at the 64-processor scale, the high cost of lock acquisition reduces the effectiveness of task stealing and computation is very imbalanced (Fig. 9). Increasing the problem size for Raytrace improves performance and scalability significantly. Fig. 6 shows that the improved behavior is due to lower lock synchronization costs and better load balance at the 64-processor scale. Controlling these factors by restructuring the application or transparently in the system is a promising direction for reducing synchronization costs. However, we do not explore this here.

In *Volrend*, the work per stolen task is much smaller than in Raytrace, so the performance of locks and the success of task stealing are much more critical. The initial partitioning we use for this version of Volrend results in good load balance [25], therefore, Volrend performs best with task stealing disabled for the

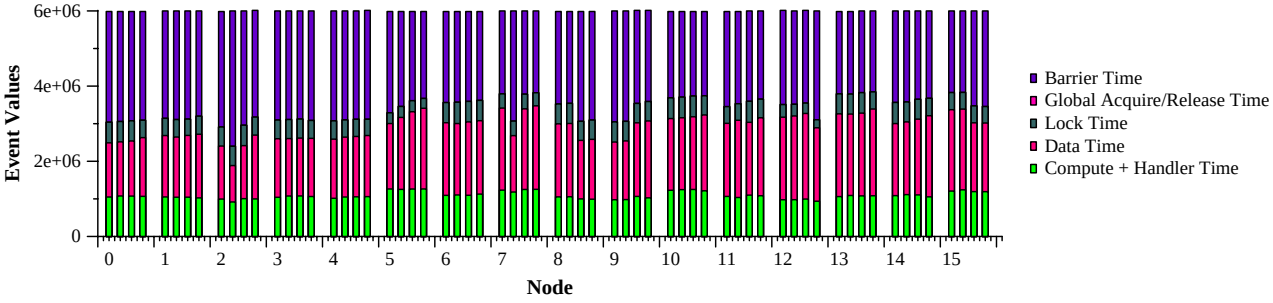


Fig. 8. Execution time breakdown for RadixLocal (large problem size).

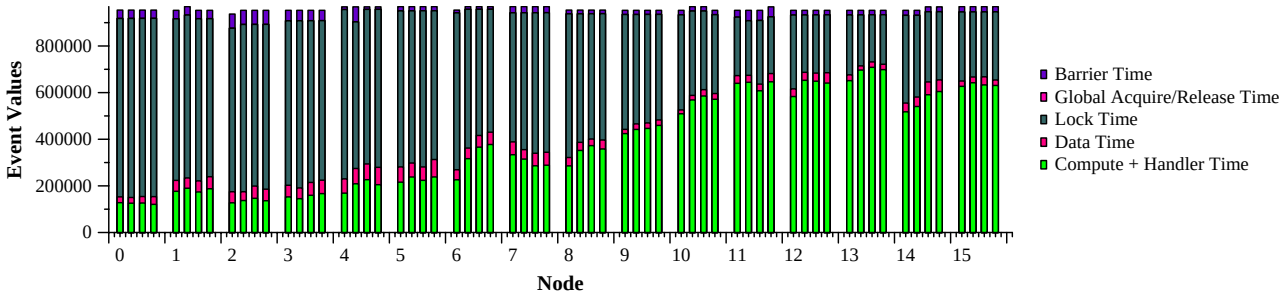


Fig. 9. Execution time breakdown for Raytrace (base problem size).

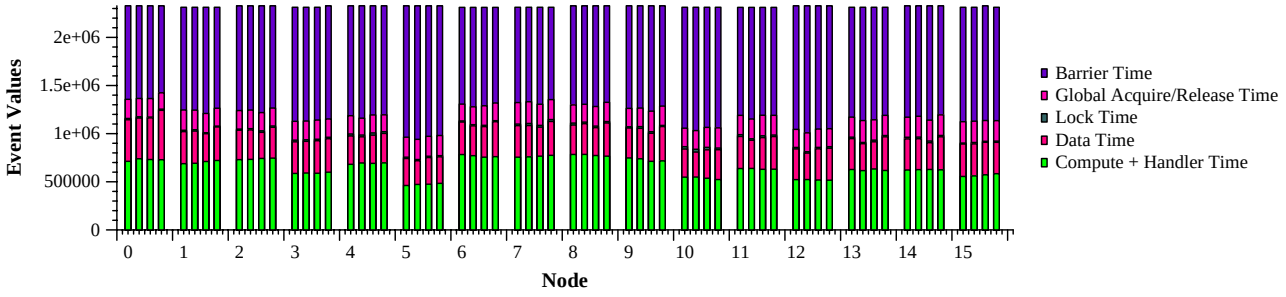


Fig. 10. Execution time breakdown for Volrend (base problem size).

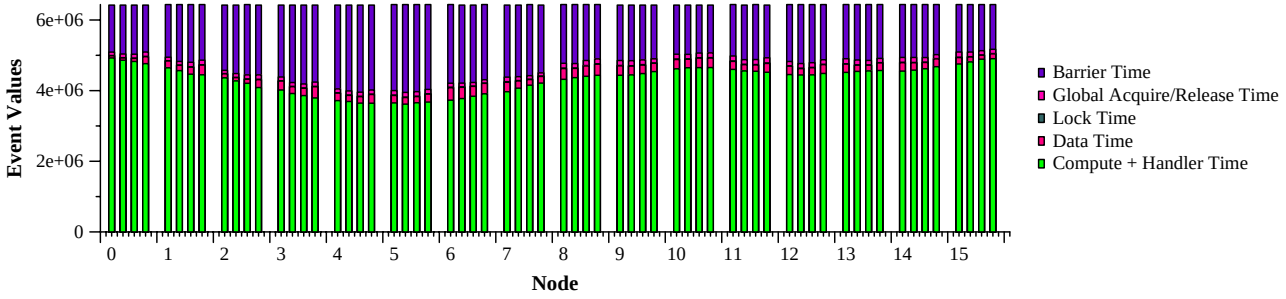


Fig. 11. Execution time breakdown for Volrend (large problem size).

problem sizes we use. Volrend performs and scales reasonably up to 32 processors for the basic problem size, but it does not scale to 64 processors (Fig. 6). Fig. 10 shows that on 64 processors the compute time is very imbalanced leading to imbalances and high barrier synchronization costs. Also, the cost of the acquire/release primitives is increased. Using a bigger problem size 11 improves scalability dramatically, mainly due to improved barrier synchronization. As in BarnesSpatial, due to the smaller processor caches on cluster (compared to the SGI Origin 2000), the speedups on the cluster appear close to ideal. We see that Volrend suffers mainly from imbalances in compute time. Enabling and improving task stealing for large configurations can lead to significant improvements. In addition, the applications and potentially the SVM protocols, could be structured in a way such that most of the lock acquires are local, taking the advantage of the hierarchical nature of the system (Fig. 11).

*WaterNsquared* performs well on 16 processors for the base problem size. Moreover, it scales well up to 32 processors. At 64 processors the lock synchronization time becomes extremely high and performance is poor. WaterNsquared uses per molecule locks to ensure that updates to the molecules are done properly. Thus, it exhibits a large amount of fine-grained locking causing lock contention and serialization, and resulting in high and highly imbalanced lock-synchronization costs at the 64-processor scale. We experimented with a version of WaterNsquared that uses per processor locks, but the overall performance is not affected. Lock time is reduced for certain processors, but is still very imbalanced and performance does not improve. Increasing the problem size leads to a significant improvement in

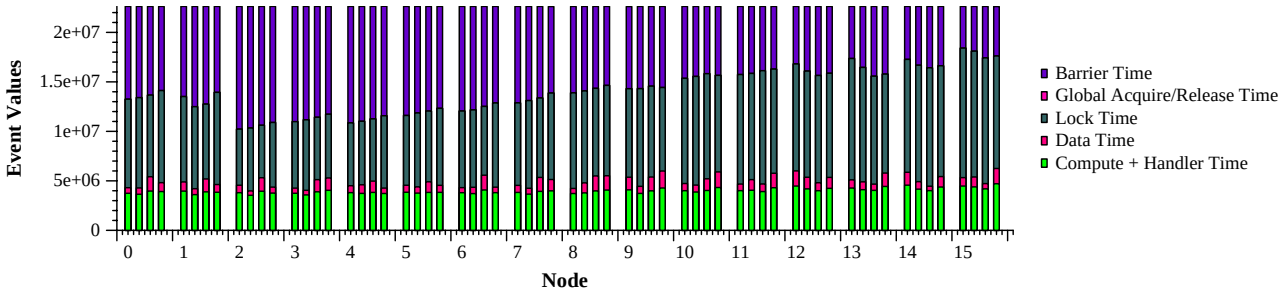


Fig. 12. Execution time breakdown for WaterNsquared (base problem size).

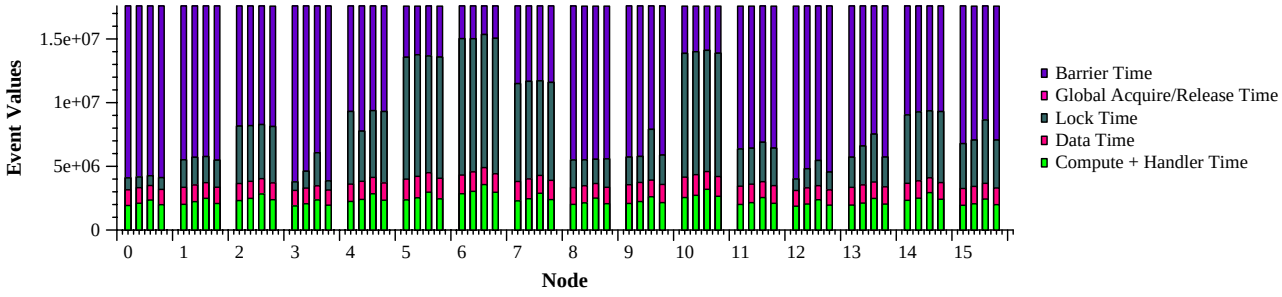


Fig. 13. Execution time breakdown for WaterSpatial (base problem size).

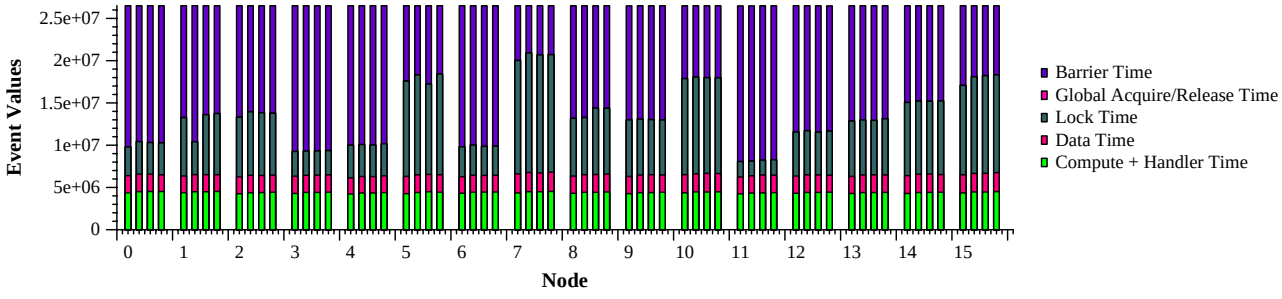


Fig. 14. Execution time breakdown for WaterSpatial (large problem size).

performance and scalability on 16, 32, and 64 processors (speedups of 13.01, 22.09, and 29.60, respectively). Less contented locking due to the larger number of molecules that need to be processed and improved communication to computation ratio ($O(n^2)$ for computation, $O(n)$ for communication), results in more balanced lock time and therefore lower barrier synchronization costs. Despite the good performance and scalability of WaterNsquared, lock synchronization overhead and imbalances due to contention are high (Fig. 12) and they constitute the dominant bottleneck for further improving application performance.

In *WaterSpatial* (Fig. 7), although the performance of the base system is reasonable for the base problem size, performance does not scale up at all at the 32- or 64-processor scale (Fig. 13). The main reason is very high imbalances in the lock wait time. These imbalances are due to the order in which locks are acquired by each processor, and most of the overhead for lock acquisition is wait time due to lock serialization. Moreover, the problem size of 15 625 molecules results in computational imbalances at the 32- and 64-processor scales.

Increasing the problem size, helps performance and scalability in two ways: First compute time imbalances disappear, since tasks are divided more evenly across all processors. Second, synchronization overhead is reduced. However, lock-synchronization costs still remain highly imbalanced across nodes (Fig. 14) resulting in low overall performance. To scale well, we restructured WaterSpatial to reduce the amount of locking (WaterSpatialFL). The final performance improves substantially (Fig. 14). Fig. 6 shows that WaterSpatialFL performs and scales very well even at the 64-processor scale (speedup of 29.28 on 64 processors). As in Raytrace and Volrend, to improve WaterSpatial we need to take better advantage of the hierarchical features of the clustered architecture, whereas for WaterSpatialFL data wait time is the most significant bottleneck. (Fig. 15).

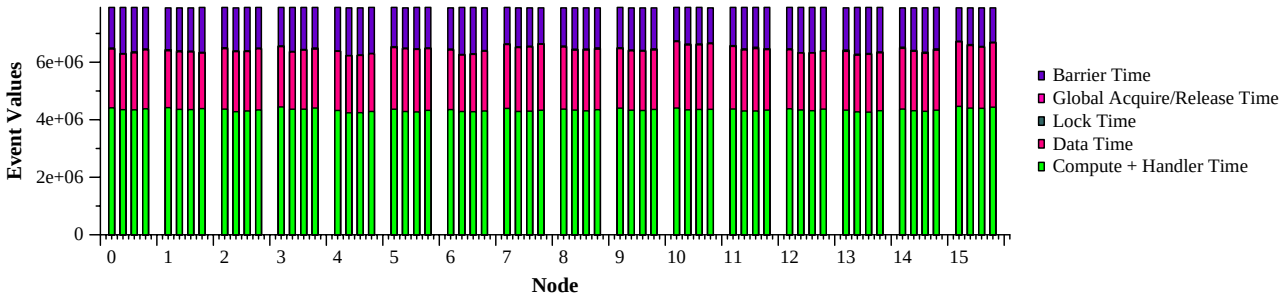


Fig. 15. Execution time breakdown for WaterSpatialFL (large problem size).

## 5. Related work

The scalability of a hardware-coherent shared address space has been demonstrated to a 128-processor scale for a real, modern system, the SGI Origin2000 [26]. The Cashmere-2L study uses 32 processors in a system of eight, four-way DEC Alpha SMPs connected by the DEC Memory Channel. The MGS system [45] examined clustering issues for SVM systems by using a hardware cache-coherent system to investigate different node partitioning schemes. Fine-grained software shared memory [35] has also been explored, most recently in the Shasta system [14,37] on a small-scale cluster of SMPs. For the scalability of uniprocessor-node systems, a simulation study compared a hardware-supported home-based protocol (the AURC protocol) with a non-home-based, original Treadmarks-style protocol [23], up to the 32-processor scale. Another study examined the all-software home-based HLRC and the original Treadmarks protocols on a 64-processor Intel Paragon multiprocessor [46]. The architectural features and performance parameters of the Paragon system are quite different from those of modern clusters. Also, a recent study [11] compares the scaling of the all-software TreadMarks and HLRC protocols on a cluster of 32 uniprocessor nodes connected with a gigabit Ethernet interconnect.

Previous work has shown that starting from "original" versions of applications that are tuned to perform well on moderate-scale (32-processor) hardware-coherent systems, SVM clusters require substantial algorithmic and data restructuring to work well even at relatively small scale (16 processors) for many real applications [25]. Similar restructurings were found to be needed to scale hardware-coherent machine performance [26] to the 128-processor scale.

Previous efforts to avoid interrupts other than for write propagation have mainly focused on using polling on the main processor to handle asynchronously arriving messages and requests. For page-based SVM, this approach has been investigated with SMP nodes in Cashmere-2L [31], where the compute processors poll the network queues for incoming messages via code instrumentation on program back-edges. They find that polling and interrupt based asynchronous protocol processing performs comparably on their system. A similar polling method was also used and compared with interrupts in [47], with similar results.

Related work in network interface support for SVM has discussed how NIs can be used for several purposes: (i) Fast communication to improve the performance of traditional send and receive communication. This type of support has been exploited in many SVM projects [15,22,31,36–38,42,46] and is also used in our base system, HLRC-SMP [36]. (ii) Protocol processing in the network interface. This choice lies at the other end of the spectrum. The network interface can be used not only to avoid interrupting the compute processor but also to perform full-blown protocol processing, including diff creation and application and the management of time-stamps and write notices. This approach was taken in [46]. Ref. [16] reserves a compute processor in an SMP node for protocol processing. The amount of protocol processing involved in SVM systems with SMP nodes was examined also in an earlier simulation study [27] and other research, and is found to be small compared to other system overheads. (iii) Transparent remote data and synchronization handling that can be utilized by protocols to alleviate key bottlenecks. In this case the remote compute processor is not involved in handling message requests, but remains responsible for all protocol processing and SVM-specific operations. This is the approach we have taken. Previous work in this area relies on more specialized network interface and/or network support [3,4,8,18,22,31,32,42].

## 6. Conclusions

This paper has examined the scalability of shared virtual memory on clusters of SMPs interconnected with system area networks. The high level goal of this work is to provide some insight as of whether shared virtual memory clusters can cover a wide range of performance and cost requirements for applications.

The protocol we use, *GeNIMA*, takes advantage of network interface support to decouple asynchronous message handling from protocol processing and to thereby eliminate the need for expensive interrupts or polling in SVM protocols. In particular, *GeNIMA* uses NI support for general-purpose, *explicit* data movement and synchronization operations that are not specific to SVM, and altered the SVM protocol to take advantage of these operations. In the final protocol, asynchronous message handling is done entirely in the NI, and protocol processing is done on the host processors but only at synchronous points with respect to application and protocol execution. The protocol propagates information more eagerly in some cases, but the need for asynchronous protocol processing and the related interrupts (or polling) is eliminated without requiring the NI to be tightly integrated in the node or to observe memory operations in it.

We have investigated the scaling of application performance on a 64-processor shared virtual memory cluster of four-way SMPs for a wide range of application classes. The 64-processor system we use is composed of 16 four-way SMPs connected by a Myrinet network running VMMC as the communication layer. We find that, surprisingly, most of the applications in our suite, except FFT and Ocean finally scale well on our 64-processor SVM cluster with reasonable problem sizes (Table 5). However, this does not come easily. Applications already optimized to the level of SPLASH-2 run well on a 64-processor hardware-cache coherent system, but often very poorly on a 16-processor cluster and most applications always perform very poorly on a 64-processor cluster. Restructurings of the sort used earlier for 16-processor SVM systems (and that turn out to be necessary on Origin to scale to 128 processors [26]) help a lot, but are still not enough for many applications: the basic problem sizes perform poorly, and increasing the problem size helps improve performance for a few applications (e.g. WaterNsquared) but not for many others (e.g. FFT, Ocean, RadixLocal, and Water-Spatial). Still further restructurings are needed in many cases. In general, in most cases where success is achieved, it requires significantly larger problem sizes and more substantial applications restructurings than on hardware-coherent systems.

Empirically, we can divide the applications into four classes based on their characteristics and their performance on a 16-processor cluster. The first class is those that perform well at the 16-processor scale and do not use much locking. These applications (LU, Water-SpatialFL, and BarnesSpatial) scale well to 64 processors even for relatively small problem sizes. The second class is those that perform well at the 16-processor scale despite using a lot of locking. These applications (WaterNsquared, Volrend, and Raytrace) do not scale well for the basic problem sizes but do scale well as problem size is increased. The third class is those that do not perform well at 16-processor scale due primarily to an abundance of locking and acquire/release primitives (e.g. Radix, and the original, SPLASH-2 forms of Barnes, Volrend, and Raytrace, that we do not present in this paper). Increasing problem size does not suffice here, and restructuring has to be used to dramatically reduce the frequency of locking. Finally, the fourth class is applications that do not perform well at the 16-processor scale due primarily to memory bus or communication bandwidth problems (e.g. FFT, Ocean, Radix). These application require either using an entirely different algorithm or improving memory bus and communication layer performance.

There are three major sources of poor performance: an abundance of fine-grained locking, imbalances in communication or other costs that are not fundamental due to bandwidth limitations, and memory bus or communication bandwidth limitations. The first two sources can be taken care of either by using larger problem sizes or by restructuring, even though restructuring is often algorithmic and substantial. In these cases, barrier time (either imbalances or protocol processing costs) often becomes the dominant bottleneck at large scale. The third source is more difficult to overcome on systems that do not provide enough effective bandwidth (memory bus or network), unless a completely different algorithm for the problem can be found (e.g. in sorting).

Overall, we find our final results surprisingly optimistic, given the nature of the applications used. However, the substantial remaining gap in the performance of the two architectures at the 64-processor scale and the tremendous need for algorithmic and programming restructuring and detailed performance analysis for SVM, imply that a lot more work is needed at all layers to make SVM on clusters of SMPs scalable. In particular: (i) The cost and bandwidth of communication in system area networks needs to be improved for certain very demanding applications. (ii) The cost of many protocol operations needs to be reduced, perhaps by providing additional support in the network interface and by adapting protocols to use it. This is particularly important for reducing the cost of lock and barrier synchronization at large-scale systems. (iii) The application restructuring methods must be further developed and codified into a set of guidelines for performance-portable and scalable parallel programming, for both tightly coupled systems and clusters, so that they can be practiced more naturally. (iv) There is a need for better understanding of the differences between coherent and non-coherent network interfaces, and their interactions with software systems layers and applications that can be restructured. (v) The emergence of hardware multi-threading makes our approach more relevant due to the fact that SMP nodes may now become wider resulting in

higher efficiencies. In particular, it may now become possible to build low-cost, single-box shared memory servers that support 32-128 hardware threads in a three-level hierarchy. The first level is multiple threads on a single CPU, the second level is hardware cache coherence across a few CPUs on single board, and the third level software cache coherent across a few boards in a single box. Our approach makes this possible by eliminating the need for a specialized third-level interconnect and presents an exciting possibility.

Finally, given the performance results presented in this work, other, higher-level issues that arise in clusters become more important. Providing a single image system for both applications programmers and users would allow more applications (especially commercially oriented applications) to take advantage of clusters as scalable servers. Also, the more distributed nature of clusters compared to SMP and DSM systems make reliability and availability issues more difficult to address. These issues emerge, to our experience, as the next set of issues that are not as much performance oriented and that need to be addressed in shared virtual memory clusters by future work.

## Acknowledgments

The authors thankfully acknowledge the help of Brian O'Kelley, Xiang Yu, and Sanjeev Kumar with earlier versions of this work. We thank Hongzhang Shan for making available to us the improved version of Barnes. We thank Yuqun Chen for initially porting VMMC to WindowsNT, Scott Karlin for helping debug the hardware during the port, and the staff members of the Computer Science Department for their help with cumbersome task of managing the system. We also thank the members of the PRISM group at Princeton, in particular Liviu Iftode, Kai Li, Rudrajit Samanta, Limin Wang, and Yuanyuan Zhou for useful discussions. We gratefully acknowledge the support of NSF, DARPA, and NSERC.

## References

[1] D.H. Bailey, FFTs in external or hierarchical memories, J. Supercomput. 4 (1990) 23–25.

[2] J.E. Barnes, P. Hut, A hierarchical $O(N \log N)$ force calculation algorithm, Nature 324 (4) (1986) 446–449.

[3] R. Bianchini, L.I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, C.L. Amorim, Hiding communication latency and coherence overhead in software DSMs, Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS7), October 1996.

[4] A. Bilas, L. Iftode, J.P. Singh, Evaluation of hardware support for shared virtual memory clusters, Proceedings of the 12th ACM International Conference on Supercomputing (ICS98), July 1998.

[5] A. Bilas, C. Liao, J.P. Singh, Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems, Proceedings of the 26th International Symposium on Computer Architecture (ISCA26), May 1999.

[6] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith, M. Zagha, A comparison of sorting algorithms for the connection machine CM-2, Proceedings of the First Annual ACM SIGPLAN Symposium on Parallel Algorithms and Architectures (SPAA91), July 1991, pp. 3–16.

[7] M. Blumrich, C. Dubnick, E. Felten, K. Li, Protected, User-level DMA for the SHRIMP network interface, Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture (HPCA2), February 1996.

[8] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, J. Sandberg, A virtual memory mapped network interface for the shrimp multicomputer, Proceedings of the 21st International Symposium on Computer Architecture (ISCA21), April 1994, pp. 142–153.

[9] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, W. Su, Myrinet: a gigabit-per-second local area network, IEEE Micro 15 (1) (1995) 29–36.

[10] A. Brandt, Multi-level adaptive solutions to boundary-value problems, Math. Comp. 31 (138) (1977) 333–390.

[11] A.L. Cox, E. de Lara, Y.C. Hu, W. Zwaenepoel, A performance comparison of homeless and home-based lazy release consistency protocols in software shared memory, Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture (HPCA5), February 1999.

[12] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, K. Li, VMMC-2: efficient support for reliable, connection-oriented communication, Proceedings of the 1997 IEEE Symposium on High Performance Interconnects (HOT Interconnects V), August 1997 (A short version of this appears in IEEE Micro, January/February 1998).

[13] D. Dunning, G. Regnier, The virtual interface architecture, Proceedings of the 1997 IEEE Symposium on High Performance Interconnects (HOT Interconnects V), August 1997.

[14] S. Dwarkadas, K. Gharachorloo, L. Kontothanassis, D. Scales, M.L. Scott, R. Stets. Comparative evaluation of fine- and coarse-grain software distributed shared memory, Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture (HPCA5), February 1999.

[15] A. Erlichson, N. Nuckolls, G. Chesson, J. Hennessy, SoftFLASH: analyzing the performance of clustered distributed virtual shared memory, Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS7), October 1996, pp. 210–220.

[16] B. Falsafi, D.A. Wood, Scheduling communication on an SMP node parallel machine, Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture (HPCA3), 1997, pp. 128–138.

[17] R. Gillett, M. Collins, D. Pimm, Overview of network memory channel for PCI, Proceedings of the IEEE Spring COMPCON '96, February 1996.

[18] N. Hardavellas, G.C. Hunt, S. Ioannidis, R. Stets, S. Dwarkadas, L. Kontothanassis, M.L. Scott, Efficient use of memory-mapped network interfaces for shared memory computing. Newsletter of the IEEE CS Technical Committee on Computer Architecture, March 1997, pp. 28–33.

[19] L. Hernquist, Hierarchical N-body methods, Comput. Phys. Comm. 48 (1988) 107–115.

[20] C. Holt, J.P. Singh, J. Hennessy, Architectural and application bottlenecks in scalable DSM multiprocessors, Proceedings of the 23rd International Symposium on Computer Architecture (ISCA23), May 1996.

[21] R.W. Horst, D. Garcia, ServerNet SAN I/O Architecture, Proceedings of the 1997 IEEE Symposium on High Performance Interconnects (HOT Interconnects V), August 1997.

[22] L. Iftode, C. Dubnicki, E.W. Felten, K. Li, Improving release-consistent shared virtual memory using automatic update, Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture (HPCA2), February 1996.
[23] L. Iftode, J.P. Singh, K. Li, Understanding application performance on shared virtual memory, Proceedings of the 23rd International Symposium on Computer Architecture (ISCA23), May 1996.
[24] D. Jiang, B. Cokelley, X. Yu, A. Bilas, J.P. Singh, Application scaling under shared virtual memory on a cluster of SMPs, Proceedings of the 13th ACM International Conference on Supercomputing (ICS99), June 1999, pp. 165–174.
[25] D. Jiang, H. Shan, J.P. Singh, Application restructuring and performance portability across shared virtual memory and hardware-coherent multiprocessors, Proceedings of the 1997 ACM Symposium on Principles and Practice of Parallel Programming (PPoPP97), June 1997.
[26] D. Jiang, J.P. Singh, Does application performance scale on cache-coherent multiprocessors: a snapshot, Proceedings of the 26th International Symposium on Computer Architecture (ISCA26), May 1999.
[27] M. Karlsson, P. Stenstrom, Performance evaluation of cluster-based multiprocessor built from ATM switches and bus-based multiprocessor servers. Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture (HPCA2), February 1996.
[28] M.G.H. Katevenis, E.P. Markatos, G. Kalokerinos, A. Dollas, Telegraphos: a substrate for high-performance computing on workstation clusters, J. Parallel Distrib. Comput. 43 (2) (1997) 94–108.
[29] P. Keleher, A.L. Cox, S. Dwarkadas, W. Zwaenepoel, TreadMarks: distributed shared memory on standard workstations and operating systems, Proceedings of the Winter USENIX Conference, January 1994, pp. 115–132.
[30] P. Keleher, A.L. Cox, W. Zwaenepoel, Lazy consistency for software distributed shared memory, Proceedings of the 19th International Symposium on Computer Architecture (ISCA19), May 1992, pp. 13–21.
[31] L.I. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira Jr., S. Dwarkadas, M.L. Scott, VM-based shared memory on low-latency, remote-memory-access networks. Proceedings of the 24th International Symposium on Computer Architecture (ISCA24), June 1997, pp. 157–169.
[32] L.I. Kontothanassis, M.L. Scott, Using memory-mapped network interfaces to improve the performance of distributed shared memory, Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture (HPCA2), February 1996.
[33] J.P. Laudon, D. Lenoski, The SGI origin2000: a scalable CC-NUMA server, Proceedings of the 24th International Symposium on Computer Architecture (ISCA24), June 1997.
[34] J. Nieh, M. Levoy, Volume rendering on scalable shared-memory MIMD architectures, Proceedings of the Boston Workshop on Volume Visualization, October 1992.
[35] S.K. Reinhardt, J.R. Larus, D.A. Wood, Tempest and typhoon: user-level shared memory, Proceedings of the 21st International Symposium on Computer Architecture (ISCA21), April 1994, pp. 325–336.
[36] R. Samanta, A. Bilas, L. Iftode, J.P. Singh, Home-based SVM protocols for SMP clusters: design, simulations, implementation and performance, Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture (HPCA4), February 1998.
[37] D.J. Scales, K. Gharachorloo, A. Aggarwal, Fine-grain software distributed shared memory on SMP clusters, Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture (HPCA4), January 1998, pp. 125–136.
[38] I. Schoinas, B. Falsafi, M.D. Hill, J.R. Larus, C.E. Lucas, S.S. Mukherjee, S.K. Reinhardt, E. Schnarr, D.A. Wood, Implementing fine-grain distributed shared memory on commodity SMP workstations, Technical Report 1307, University of Wisconsin-Madison, March 1996.
[39] J.P. Singh, A. Gupta, J.L. Hennessy, Implications of hierarchical N-body techniques for multiprocessor architectures, ACM Trans. Comput. Systems 13 (2) (May 1995) 141–202. Stanford University Technical Report no. CSL-TR-92-506, January 1992).
[40] J.P. Singh, A. Gupta, M. Levoy, Parallel visualization algorithms: performance and architectural implications, IEEE Computer 27 (67) (July 1994) 45–55.
[41] J.P. Singh, J.L. Hennessy, Finding and exploiting parallelism in an ocean simulation program: experiences, results, implications, J. Parallel Distrib. Comput. 15 (1) (1992) 27–48.
[42] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, M. Scott, Cashmere-2L: software coherent shared memory on a clustered remote-write network, Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP16), October 1997.
[43] S.C. Woo, J.P. Singh, J.L. Hennessy, The performance advantages of integrating message-passing in cache-coherent multiprocessors, Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS6), 1994.
[44] S. Cameron Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The SPLASH-2 programs: characterization and methodological considerations, Proceedings of the 22nd International Symposium on Computer Architecture (ISCA22), Santa Margherita Ligure, Italy, June 1995, pp. 24–36.
[45] D. Yeung, J. Kubiatowicz, A. Agarwal, MGS: a multigrain shared memory system, Proceedings of the 23rd International Symposium on Computer Architecture (ISCA23), May 1996.
[46] Y. Zhou, L. Iftode, K. Li, Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems, Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI96), October 1996.
[47] Y. Zhou, L. Iftode, J. P. Singh, K. Li, B.R. Toonen, I. Schoinas, M.D. Hill, D. Wood, Relaxed consistency and coherence granularity in DSM systems: a performance evaluation, Proceedings of the 1997 ACM Symposium on Principles and Practice of Parallel Programming (PPoPP97), June 1997.

**Angelos Bilas** is an Associate Professor in the Department of Computer Science at the University of Crete, Greece and the Institute for Computer Science at the Foundation of Research and Technology—Hellas. He received his B.S.E. degree in Computer Science from the Computer Engineering and Informatics Department, Patras University, Patras, Greece in 1993 and his M.A. and Ph.D. degrees in Computer Science from Princeton University in 1995 and 1998 respectively. His current interests include parallel architectures, interconnection networks for clusters, storage systems, programming paradigms, parallel applications, performance analysis and evaluation, distributed systems, and distributed operating systems.

**Dongming Jiang** received her Ph.D in computer science from Princeton University in June, 2000. She is a co-founder of firstRain Inc., provider of business monitoring software and solutions. Since graduation from Princeton, she has been a principal architect and scientist for firstRain, leading research and development. Her research interests include parallel computing and scalable computing systems, infrastructure and applications.

**Jaswinder Pal Singh** is an Associate Professor in the Computer Science Department at Princeton University. He obtained his Ph.D. from Stanford University in 1993, and his B.S.E. degree from Princeton in 1987. His research interests are at the boundary of parallel and distributed applications and multiprocessor systems, both architecture and software, and in applications of high-performance and distributed computing. At Stanford, he participated in the DASH and FLASH multiprocessor projects, leading the applications efforts there. He has led the development and distribution of the SPLASH and SPLASH-2 suites of parallel programs, which are widely used in parallel systems research. At Princeton, he has led the PRISM research group, which does application-driven research in supporting programming and communication models on a variety of communication architectures, as well as in novel applications of high-performance computing such as simulating the immune system. He has co-authored a graduate textbook called "Parallel Computer Architecture: A Hardware-Software approach." He is a Sloan Research Fellow and a recipient of the Presidential Early Career Award for Scientists and Engineers (PECASE).