Application Scaling under Shared Virtual Memory on a Cluster of SMPs

Dongming Jiang, Brian O'Kelley, Xiang Yu, Sanjeev Kumar, Angelos Bilas*, and Jaswinder Pal Singh

Department of Computer Science
Princeton University
Princeton, NJ 08544
{dj, cokelley, xyu, skumar, jps}@cs.princeton.edu

*Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario M5S 3G4, Canada
bilas@eecg.toronto.edu

Abstract

In this paper we examine how application performance scales on a state-of-the-art shared virtual memory (SVM) system on a cluster with 64 processors, comprising 4-way SMPs connected with a fast system area network. The protocol we use is home-based and takes advantage of general-purpose data movement and mutual exclusion support provided by a programmable network interface. We find 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, achieving at least half the parallel efficiency of a high-end hardware-coherent system and often much more. We explore further application restructurings than those developed earlier for smaller-scale SVM systems, examine the main remaining system and application bottlenecks, and point out directions for future research.

1 Introduction

A shared address space with transparent coherent replication of data is recognized to be an attractive programming model. Research and development in the last decade have shown that it can be implemented efficiently in hardware with physically distributed main memory. Beyond small-scale bus-based machines, most vendors of tightly-coupled multiprocessors are now building machines of this sort. The performance of these machines has been demonstrated to be quite successful on real applications at moderate scale [14, 1, 19, 11]. Recent work [12] has shown that these systems scale well, at least up to 128 processors. At the same time, less tightly coupled clusters of commodity workstations or multiprocessors (especially symmetric multiprocessors or SMPs) have emerged as important platforms for high-performance computing. These, however, do not provide hardware support for a shared address space or cache coherence, so this programming model must be implemented in software.

A lot of research has been done in the area of software shared memory protocols and systems for clusters, as well as 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 [20, 18, 6, 13, 17]. Protocols and systems have also been developed to exploit the features of network interfaces in system area networks to enhance software shared memory performance [20, 8, 2, 15].

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 sixteen 4-way PentiumPro SMP nodes connected by a Myrinet [3] point-to-point system area network, running an extended, high performance, user-level virtual memory mapped communication library (VMMC) [5]. We use the *GeNIMA* home-based, page-level shared virtual memory (SVM) system, which adds general purpose data movement and synchronization support in the network interface and exploits it in the SVM protocol [2]. In this work we enhance the *GeNIMA* system to improve performance and memory scalability.

Examining scalability is very 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 [10] 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 [10, 2], we do not present results for those versions here but use the restructured versions from [10] as our "original" versions.

To establish context, we loosely compare the speedups achieved for all applications and versions with those ob-

tained in [12] on a 64-processor SGI Origin2000 hardware-coherent machine. Since the cluster is a lot cheaper (by almost an order of magnitude) than the Origin2000[1], 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.

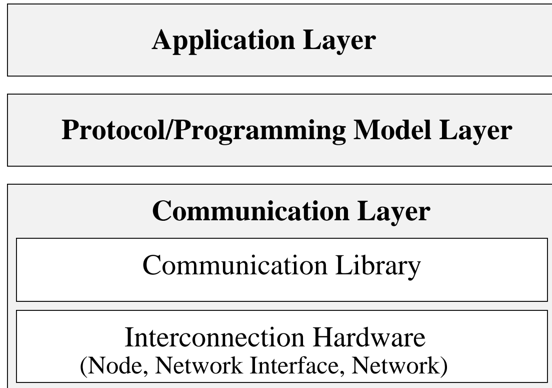

Figure 1: The three major layers of a cluster SVM architecture.

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 (in particular, in addition to the restructurings already developed in [10], are new restructurings needed at larger scale that were not needed at smaller scale); (iv) what are the key remaining system bottlenecks and in which layer (Figure 1) 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, achieving at least half the parallel efficiency of a high-end hardware-coherent system and often much more.

The rest of this paper is organized as follows. Section 2 discusses related work and what is known so far. Section 3 describes the architecture of our systems (both hardware and software) in some detail, including new optimizations. In Section 4 we present our results and analyze the system behavior. Finally, in Section 5 we draw our conclusions and discuss future directions for work on SVM clusters.

2 Related Work

The scalability of a hardware-coherent shared address space multiprocessors has been demonstrated to a 128-processor scale for a real, modern system, the SGI Origin2000 [12]. We use the 64-processor speedup results from that platform as a loose standard against which to compare our results in this paper.

Previous work on examining the performance of software shared memory clusters at significant scale includes the following. The Cashmere-2L study uses 32 processors in a system of eight, 4-way DEC Alpha SMPs connected by the DEC Memory Channel. The focus is on protocol-level issues for extending the Cashmere protocol to clusters of SMPs and examining the benefits achieved from using SMP clusters rather than uniprocessor clusters, not so much on understanding scalability. It also does not use a wide enough range of applications that is needed to understand whether or not SVM works at larger scale. Three "standard" SPLASH-2 applications are used, together with some other parallel programs; while the two simpler ones (LU and WaterNsquared) scale quite well from 4 to 8 SMP nodes, the Barnes-Hut application does not. The MGS system [22] examined clustering issues for SVM systems by using a hardware cache-coherent system to investigate different node partitioning schemes. They used a TreadMarks-like protocol, and they find synchronization and data movement across nodes to be a problem. Fine-grained software shared memory [16] has also been explored, most recently in the Shasta system [18, 6] 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 [9]. It examined both 16- and 32-processor clusters, and studied the impact of problem size. However, it used rather small problem sizes, did not examine application restructuring, and was limited by not being performed on a real system. Another study examined the all-software home-based HLRC and the original Treadmarks protocols on a 64-processor Intel Paragon multiprocessor [23]. This study focused on the ability of a communication coprocessor to overlap protocol processing with useful computation in the two protocols, but it also compared the protocols at this large scale. However, the architectural features and performance parameters of the Paragon system are quite different from those of modern clusters. Also, the study used mostly simple, not very demanding computational kernels and only two real applications (WaterNsquared and Raytrace).

Finally, an interesting contemporary study [4] compares the scaling of the all-software TreadMarks and HLRC protocols on a cluster of uniprocessors, with 32 uniprocessor nodes connected with a gigabit Ethernet interconnect. The results show that the protocols perform similarly overall, and that neither protocol seems to scale well to this point for non-trivial applications (generally in their original forms). Here too, relatively few real applications were used and the focus was on comparing the two families of protocols rather than revealing system bottlenecks.

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 [10]. Similar restructurings were found to be needed to scale hardware-coherent machine performance, especially to larger processor counts than we consider here [12], so the restructurings are not specific to shared virtual memory and we study them on Origin2000 as well. Overall, while a lot of progress has been made, the question of whether SVM can be made to work well for modern clusters at substantial scale, and how dif-

[1]The cluster we use has a cost in the range of US$300K (assuming today's 450MHz processors as opposed to 200MHz in our system), whereas the Origin2000 has an estimated cost of about US$2.5M for a configuration similar to the cluster we use.

ficult this is on the programming front, has not been adequately addressed.

3 Platform

Each node in our cluster is an Intel PentiumPro Quad SMP. It contains four 200 MHz processors, a shared snooping-coherent bus, and 512 MB of main memory. Each processor has separate 8KB instruction and data caches and a 512KB unified 4-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 [3] 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 (133MBytes/s) on which the NI sits.

In the next subsections we describe the system that we have built, following the layered architecture of Figure 1, in a bottom-up fashion. A description of the relevant features of the Origin2000 system can be found in [12].

3.1 Communication Layer

The communication layer we use in this system is Virtual Memory Mapped Communication (VMMC) for the Myrinet network [5]. 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 [2]. It is important to note that these features are very general and can be used beyond SVM protocols. Table 1 shows the cost for the basic VMMC operations.

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

Table 1: Basic VMMC costs. 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 Protocol Layer

We use *GeNIMA* [2] as our base system. *GeNIMA* 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. 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:

(i) On the performance side, we schedule operations in barriers to achieve maximal overlap of useful work, communication, and idle time. The cost of `mprotect`ing 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.

(ii) 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 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 2 shows that the new system has performance either comparable or better than the Linux version[2]. The base

[2]The 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 con-

system achieves parallel speedups close to those on a hardware cache-coherent multiprocessor, an SGI Origin2000, at the 16-processor scale for many classes of applications.

Application	Problem Size	Speedup (16 procs)		
		SVM		Origin
		Linux	NT	2000
FFT	1M points	6.1	4.1	13.4
LUcontiguous	4096×4096 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

Table 2: Application speedups on 16 processors on three systems: (i) *GeNIMA* on Linux [2], (ii) *GeNIMA* on WindowsNT for this work, and (iii) SGI Origin2000, a hardware cache-coherent multiprocessor. The second column shows the problem sizes that were used.

3.3 Applications Layer

We use most of the SPLASH-2 shared-address-space parallel applications [21]. In particular, for applications that have been restructured to perform well on a simulated 16-processor SVM system in [10], we apply the best versions from there as the original versions in this scalability study, because their original SPLASH-2 versions perform poorly even on the small- to moderate-scale SVM systems. More specifically: (i) FFT, LUcontiguous, WaterNsquared, and WaterSpatial are unmodified SPLASH-2 versions. 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 [7]. (ii) BarnesSpatial, OceanRowwise, RadixLocal, Volrend, and Raytrace are the restructured applications from [10]. 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 synchronizations. (iii) Finally for better scaling, 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.

figuration (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.

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 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? For simplicity of presentation, we consider both the original application and its further restructured version, if applicable, at every stage of the performance analysis in this section (corresponding to the different questions that were raised above).

In studying the scalability of SVM on clusters we use two problem sizes: (i) a "basic" problem size, which has been demonstrated to scale reasonably well at the 16-processor scale [2] (see Table 2), 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 3 presents the problem sizes we use and the size of the application shared data for each problem size.

Table 4 presents the application speedups on 16, 32 and 64 processors in our system for each problem size. For reference we also include speedups on an SGI Origin2000 system. Figures 7 and 6 present analytic execution time breakdowns and speedup curves for each application. 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 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 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.

FFT: FFT (Figure 6) 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.

For the basic problem size (1M points), the speedup with 16, 32, and 64 processors is 4.13, 3.94, and 4.65 respectively (13.4, 31.72, and 45.84 on Origin). We see that FFT not only exhibits a relatively low speedup at the 16-processor scale (because of the high data wait time and the local memory bus contention), but it scales very poorly as well. As the number of nodes that fetch data from each node increases, the contention in the network interface of the servicing node increases. This results in higher remote memory access time. At the same time, since these remote requests need to access the memory bus, the additional contention for the memory bus affects local memory access time as well. Therefore, the poor scalability of FFT is due to the increased remote and local memory access time at larger processor counts. We tried a restructured version for FFT, where the transpose is performed implicitly while computing, to reduce burstiness

[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.

Application	Basic Problem Size	Amount of Data (MB)	Large Problem Size	Amount of Data (MB)
FFT	1M points	48	4M points	192
LUcontiguous	4096×4096 matrix	128	6144×6144 matrix	288
OceanRowwise	514×514 grids	95	1026×1026 grids	N/A
BarnesSpatial	32K particles	46	128K particles	N/A
RadixLocal	4M integers	33	32M integers	257
SampleSort	4M integers	53	32M integers	412
WaterNsquared	4096 mols	3	15625 mols	10
WaterSpatial	15625 mols	62	32768 mols	107
WaterSpatialFL	15625 mols	62	32768 mols	107
Volrend	256^3 head	22	512^3 head	168
Raytrace	256×256 car	50	512×512 car	80

Table 3: Application problem sizes and amount of application shared data for each problem size.

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

Table 4: Application speedups for each problem size on the SVM cluster and the Origin2000 for 16, 32, and 64 processors.

of communication compared to an explicit transpose phase. However, this increases the number of messages and data wait time gets even worse (not shown here), which indicates that contention at the end points remains a problem.

Using a bigger problem size (4M points) improves performance only marginally, and scalability is still very poor. Thus, the bottleneck for FFT is contention in the network interface and the memory bus; improving the performance and scalability of FFT would require faster network interfaces and memory buses in SMP nodes.

LU: LUcontiguous (Figure 6) is expected to perform well, since its communication to computation ratio is very low. As we can see, it achieves a speedup of 12.95 on the base system with 16 processors (14.32 on Origin). It keeps scaling reasonably well up to 64 processors, with speedups of 22.99 and 37.19 for 32 and 64 processors respectively (26.86 and 45.39 on Origin). Figure 7 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. The speedup on 64 processors increases from 37.19 to 39.09 mainly due to the the reduction of barrier synchronization cost. 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)).

Ocean: OceanRowwise (Figure 6) 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. Figure 7 reveals that compute, data communication, and barrier time do not scale down proportionally to the increased 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 row-wise partitioning. The barrier time increases because of high protocol costs as well as increased load imbalances.

We do not present data for bigger problem sizes for Ocean, since they have unrealistically high memory requirements (for benchmarks). Increasing the problem size will improve Ocean's inherent characteristics in communication to computation ratio. At the same time, larger problems will aggravate contention on the memory bus within each node. Therefore, it is not clear whether bigger problem sizes will improve the scalability significantly.

Ocean combines part of the problems encountered in FFT (memory bus contention, but not data wait time) and the problems encountered in LU (high barrier cost). Im-

proving scalability for Ocean would require addressing both of these issues, with barrier cost being the most important one.

Barnes: The original version of Barnes (where both the tree build and the force calculation phases are run in parallel) performs very poorly even at 16 processors, and it does not scale at all (not shown here). By applying a totally different tree-building algorithm to eliminate fine-grained and global locking, BarnesSpatial (Figure 6) performs and scales well even for the basic problem size. BarnesSpatial appears to scale on the SVM cluster even better than on Origin on 16 and 64 processors. The reason is that the uniprocessor performance on the cluster is worse than on Origin because the cluster has a much smaller cache with each processor than the Origin, resulting in more capacity misses. Figure 7 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. This turns out to be due to contention in remote page access, since the number of pages fetched by different processors is actually relatively balanced.

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: Radix performs very poorly (Figure 6). Figure 7 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 (Figure 6), by substantially reducing data communication overhead and synchronization time on locks. The remaining bottleneck in sample sort is high barrier cost (Figure 7).

Raytrace: Raytrace performs well on 16 processors with a speedup of 11.3 (14.23 on Origin). Moreover, it scales reasonably well to 32 and 64 processors for the basic problem size. Figure 7 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. In Raytrace, the lock time in each processor is high either because most of the lock acquires are remote or because local lock acquires have to wait for remote ones to be completed. Moreover, at the 64-processor scale, the high cost of lock acquisition reduces the effectiveness of task stealing and computation is very imbalanced (Figure 2).

Increasing the problem size for Raytrace improves performance and scalability significantly. Figure 7 shows that the improved behavior is due to lower lock synchronization costs and better load balancing at the 64 processor scale. The bigger problem size results in better load balance, and less need for task stealing and lock synchronization.

Thus, the major problem for Raytrace is lock synchronization. The high cost of remote locks leads to performance losses. Task stealing is not effective in improving load balance due to the high locking overhead and the small task size. In our experiments, in order to decrease the number of remote lock acquires and lock contention, we always steal from processors within a SMP node first. When the work of a node is done, processors start stealing from other nodes. For Raytrace and the problem sizes we use, this SMP stealing scheme does not help much. This suggests that further improvements to Raytrace require improving the cost of lock synchronization even further, as well as restructuring the application to make task stealing more effective.

Volrend: 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 better load balance [10]. Therefore, Volrend performs the best with task stealing disabled for the problem sizes we use. Volrend performs and scales reasonably well up to 32 processors for the basic problem size, but it does not scale to 64 processors (Figure 7). Figure 3 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 improves scalability dramatically. As shown in Figure 6 it scales very well up to 64 processors. Again, due to smaller cache on uniprocessor, the speedups on the cluster appears to be better than on Origin. Compute time is more balanced and this leads to smaller barrier wait time and barrier synchronization costs in general.

We see that Volrend suffers mainly from imbalances in compute time for the base problem size. Thus, 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.

WaterNsqured: 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 (speedup of 9.88; 44.71 on Origin). 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-based 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 performance and scalability on 16, 32, and 64 processors (speedups of 13.01, 22.09, and 29.60 respectively; 15.37, 30.4, and 58.88 on Origin). Less contented locking, due to much increased computational complexity

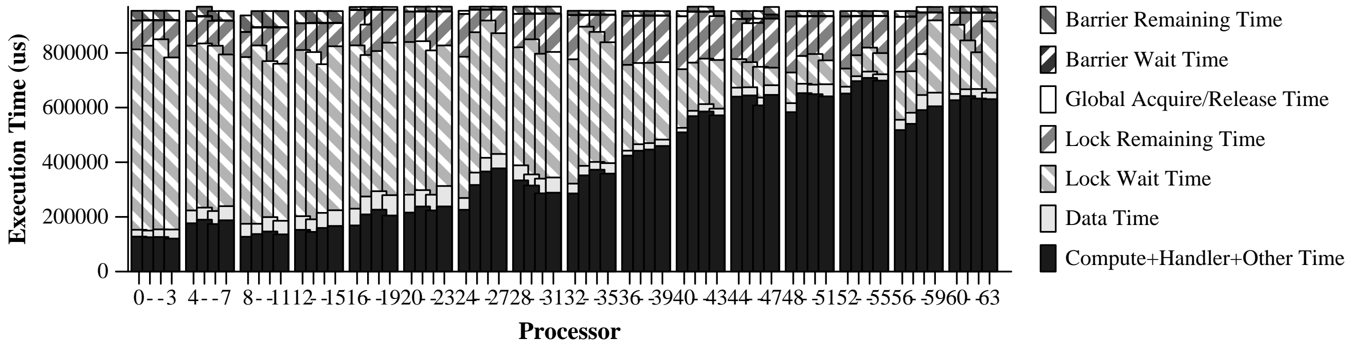

Figure 2: Execution time breakdown for Raytrace (base problem size).

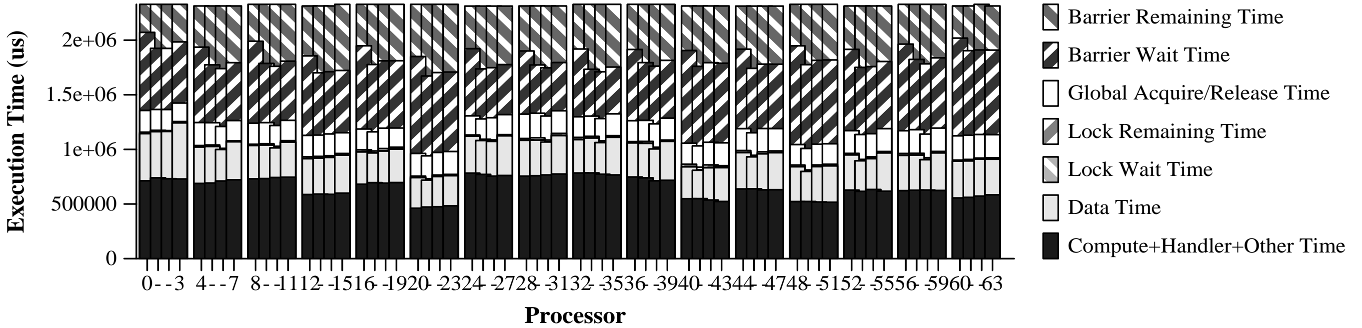

Figure 3: Execution time breakdown for Volrend (base problem size).

(O(n^2)), results in more balanced lock time and therefore lower barrier synchronization costs.

Thus, despite the good performance and scalability of WaterNsquared, lock synchronization overhead and imbalances due to contention are high (Figure 4) and they constitute the dominant bottleneck for further improving application performance.

WaterSpatial: In WaterSpatial (Figure 6), although the performance of the base system is reasonable for base problem size, performance does not scale up at all at the 32- or 64-processor scales. 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 15625 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 (Figure 5) resulting in low overall performance. To scale well, we restructured WaterSpatial to reduce the amount of locking (WaterSpatialFL). The final performance improves substantially (Figure 6). Figure 7 show that WaterSpatialFL performs and scales very well even at the 64-processor scale.

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.

5 Conclusions

We have investigated the scaling of application performance on a shared virtual memory cluster of SMPs for a wide range of application classes, using a hardware-coherent SGI Origin2000 multiprocessor with similar processor speed (but much higher cost) as a rough comparison point. The 64-processor system we use is composed of sixteen 4-way SMPs connected by a Myrinet network running VMMC as the communication layer. The home-based SVM protocol we use, *GeNIMA*, generally exhibits very good performance at the 16-processor scale, taking advantage of general-purpose network interface support for data movement and mutual exclusion.

Using our metric of 50% of the parallel efficiency obtained at the same scale on the Origin2000 (which has much higher cost than the cluster), 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 4). However, this does not come easily. Applications already optimized to the level of SPLASH-2 run well on a 64-processor Origin2000, but often very poorly on even a 16-processor cluster and most always 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 [12]) help a lot, but are still not enough for many applications: the basic problem sizes perform poorly, and increasing the problem size helps achieve 50% of the Origin speedup for a few applications (e.g. WaterNsquared) but not for many others (e.g. FFT, Ocean, RadixLocal, and WaterSpatial). 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 substan-

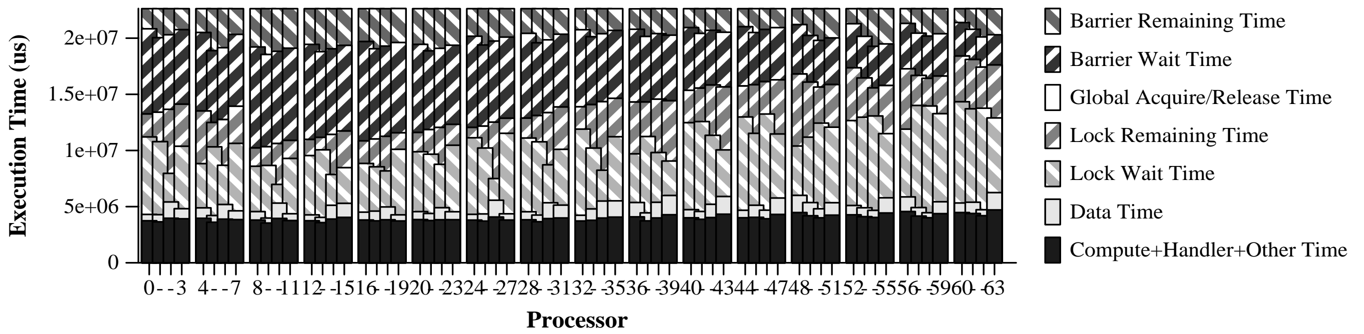

Figure 4: Execution time breakdown for WaterNsquared (base problem size).

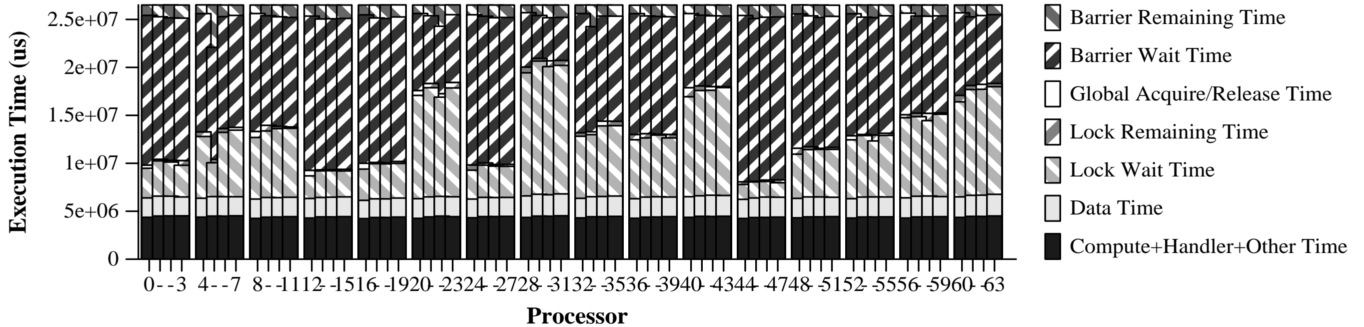

Figure 5: Execution time breakdown for WaterSpatial (large problem size).

tial applications restructurings than on hardware-coherent systems.

Empirically, we 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, WaterSpatialFL, and BarnesSpatial) scale well to 64 processors even for relatively small problem sizes. The second class is those that perform well at 16-processor scale despite using a lot of locking. These applications (WaterNsquared, Volrend, and Raytrace) do not scale well for the base 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 primarily due 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 primarily due to memory bus or communication bandwidth problems (e.g. FFT, Ocean, Radix). These applications require either using an entirely different algorithm or improving memory bus and communication layer performance.

In general, 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).

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, 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 synchronizations 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. They also need to be extended to take the advantage of a multi-level communication hierarchy as in a cluster of multiprocessor nodes.

6 Acknowledgments

We thank Yuqun Chen for initially porting VMMC to WindowsNT, Scott Karlin for helping debug the hardware during the port, Kai Li, Yuanyuan Zhou and Limin Wang for useful discussions, and the staff members of the Computer

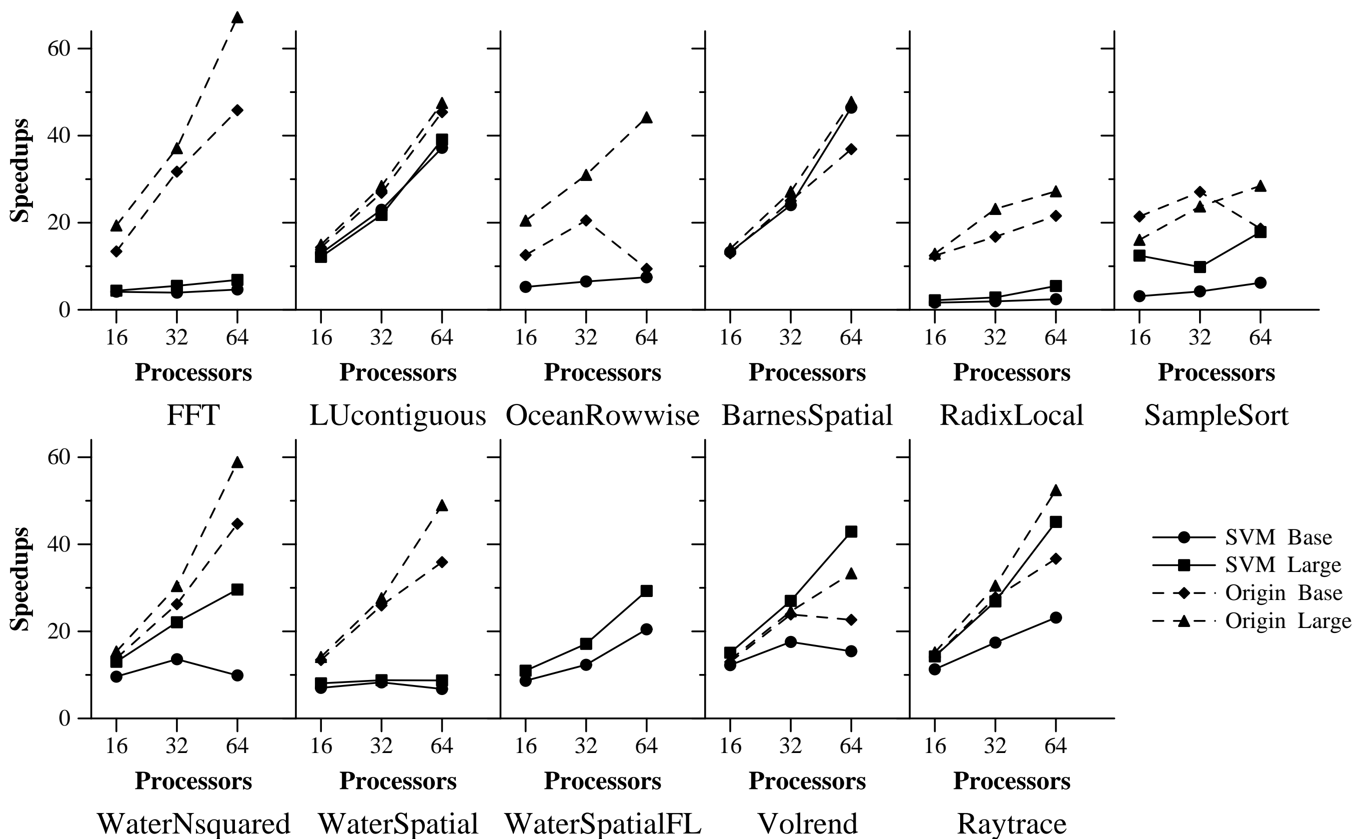

Figure 6: Speedup curves for the SVM cluster and the hardware cache coherent system. For each application there are two curves per system, one for the base and one for the large problem size.

Science Department for their help with cumbersome task of managing the system.

References

[1] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, and D. Nussbaum. The MIT Alewife machine: A large-scale distributed memory multiprocessor. In M. Dubois and S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*, pages 239–262. Kluwer Academic Publishers, May 1992.

[2] A. Bilas, C. Liao, and J. P. Singh. Accelerating shared virtual memory using commodity ni support to avoid asynchronous message handling. In *The 26th International Symposium on Computer Architecture*, May 1999.

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

[4] A. Cox, E. de Lara, Y. Hu, and W. Zwaenepoel. A performance comparison of homeless and home-based lazy release consistency protocols in software shared memory. In *The Fifth International Symposium on High Performance Computer Architecture*, Feb 1999.

[5] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: efficient support for reliable, connection-oriented communication. In *Proceedings of Hot Interconnects*, Aug. 1997.

[6] S. Dwarkadas, K. Gharachorloo, L. Kontothanassis, D. Scales, M. L. Scott, and R. Stets. Comparative evaluation of fine- and coarse-grain software distributed shared memory. Submitted for publication.

[7] C. Holt, J. P. Singh, and J. Hennessy. Application and architectural bottlenecks in large scale distributed shared memory machines. In *Proc. of the 23th Annual Intl. Symp. on Computer Architecture*, pages 134–145, May 1996.

[8] L. Iftode, M. Blumrich, C. Dubnicki, D. L. Oppenheimer, J. P. Singh, and K. Li. Shared virtual memory with automatic update support. In *Technical Report TR-575-98, Princeton University, 1998. Also at the 7th Workshop of Scalable Shared Memory Multiprocessors*, June 1998.

[9] L. Iftode., J. P. Singh, and K. Li. Understanding application performance on shared virtual memory systems. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, pages 122–133, May 1996.

[10] D. Jiang, H. Shan, and J. Singh. Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1997.

[11] D. Jiang and J. P. Singh. A methodology and an evaluation of the SGI Origin2000. In *Proceedings of ACM Sigmetrics98/Performance 98*, June 1998.

[12] D. Jiang and J. P. Singh. Does application performance scale on modern cache-coherent multiprocessors: A case study of a 128-processsor sgi origin2000. In *The 26th International Symposium on Computer Architecture*, May 1999.

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

[14] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. Design of Scalable Shared-

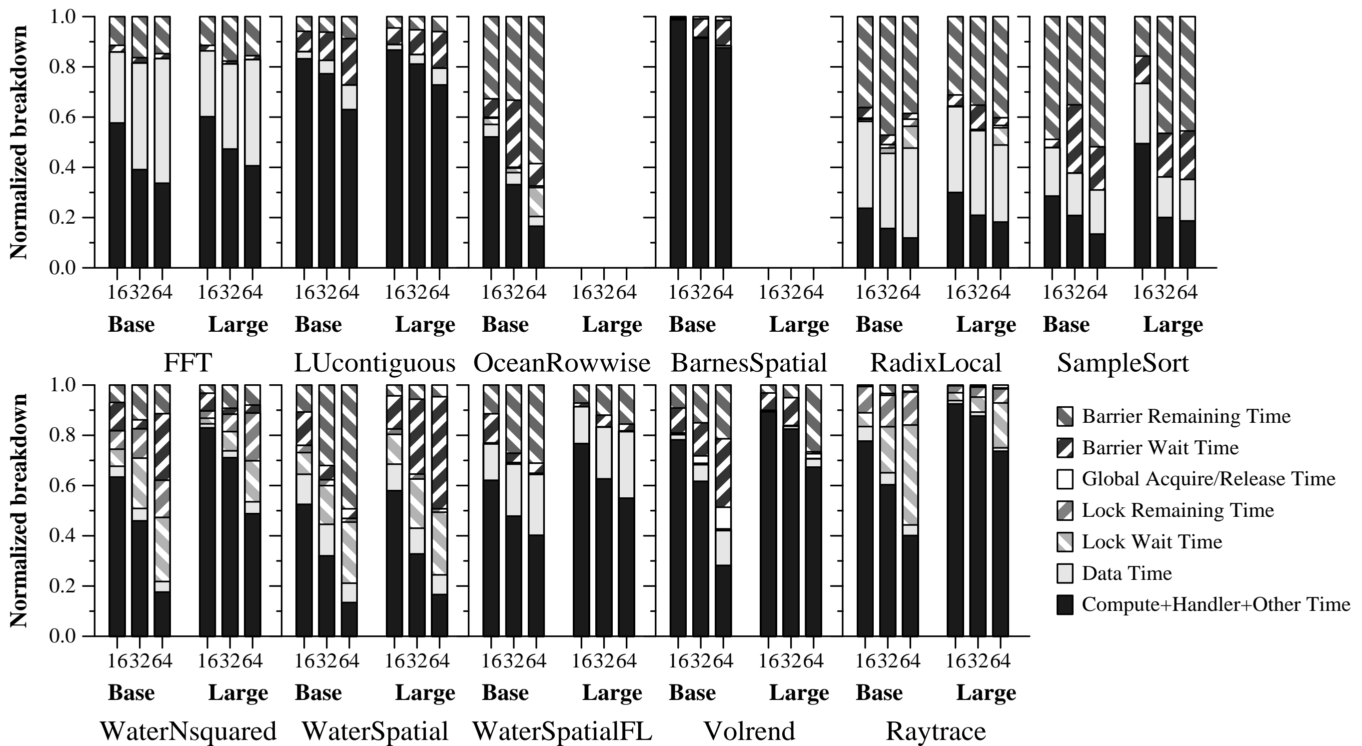

Figure 7: Average execution time breakdowns for all applications. Each graph presents the breakdown for one application, for all system and problem sizes; the bars on the left of each graph refer to the base problem size and on the right to the large problem size.

Memory Multiprocessors: The DASH Approach. In *Proceedings of COMPCON'90*, pages 62–67, 1990.

[15] S. Reinhardt, J. Larus, and D. Wood. Tempest and typhoon: User-level shared memory. In *Proceedings of the 21th International Symposium on Computer Architecuture*, pages 325–336, Apr. 1994.

[16] S. Reinhardt, J. Larus, and D. Wood. Tempest and typhoon: User-level shared memory. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 325–336, Apr. 1994.

[17] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-based svm protocols for smp clusters: Design, simulations, implementation and performance. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture, Las Vegas*, February 1998.

[18] D. J. Scales, K. Gharachorloo, , and A. Aggarwal. Fine-grain software distributed shared memory on smp clusters. In *The 3rd International Symposium on High Performance Computer Architecture*, Feb 1997.

[19] J. P. Singh, T. Joe, J. L. Hennessy, and A. Gupta. An empirical comparison of the KSR-1 ALLCACHE and Stanford DASH multiprocessors. In *Supercomputing '93*, 1993.

[20] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, Oct. 1997.

[21] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of the 22th Annual Intl. Symp. on Computer Architecture*, June 1995.

[22] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: a multigrain shared memory system. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

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