Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-Coherent Multiprocessors

Dongming Jiang, Hongzhang Shan and Jaswinder Pal Singh

Department of Computer Science
35 Olden Street
Princeton University
Princeton, NJ 08544

{dj, shz, jps}@cs.princeton.edu

Abstract

The performance portability of parallel programs across a wide range of emerging coherent shared address space systems is not well understood. Programs that run well on efficient, hardware cache-coherent systems often do not perform well on less optimal or more commodity-based communication architectures. This paper studies this issue of performance portability, with the commodity communication architecture of interest being page-grained shared virtual memory. We begin with applications that perform well on moderate-scale hardware cache-coherent systems, and find that they do not do so well on SVM systems. Then, we examine whether and how the applications can be improved for SVM systems —through data structuring or algorithmic enhancements—and the nature and difficulty of the optimizations. Finally, we examine the impact of the successful optimizations on hardware-coherent platforms themselves, to see whether they are helpful, harmful or neutral on those platforms. We develop a systematic methodology to explore optimizations in different structured classes. The results, and the difficulty of the optimizations, lead insight not only into performance portability but also into the viability of SVM as a platform for these types of applications.

1 Introduction

Coherent shared address space multiprocessors provide an attractive parallel programming model. Hardware cache-coherent machines have been shown to deliver good parallel performance, at least at moderate scale, but they are expensive to design and purchase. Many efforts have therefore been made to support a coherent shared address space using commodity-oriented parts for the communication architecture—both the controller and the network [8, 17]. One extreme in the spectrum is to support the abstraction entirely in software on networks of commodity workstations (or personal computers) with no additional hardware. This approach, called shared virtual memory (SVM), provides the coherent shared address space at page granularity through virtual memory management. Applications may interact well or poorly with this large granularity, and many techniques have been developed to alleviate poor interactions like false sharing and fragmentation (transferring a page of data when only a fraction of it is needed). Nevertheless, software communication and synchronization costs can be high as can protocol overhead, and the performance potential of this approach across a wide range of applications is not well understood.

Previous research has studied parallel application performance on particular shared memory systems [11, 12, 7, 5, 17, 2, 14, 20]. Studies on shared virtual memory have largely used applications as they were written for hardware cache-coherent machines. The performance evaluations so far point out that SVM is very sensitive to data referencing and communication patterns, and that for certain classes of applications there is a large performance gap between hardware cache-coherent and SVM systems. However, it should be possible to modify or restructure applications to interact better with page granularity and to reduce the frequency of operations that particularly hurt performance on SVM systems, perhaps at the cost of increasing other, less important overheads. The modifications are not likely to be traditional optimizations like tuning inner loops or interchanging loops to change the traversal order, but rather larger-scale restructurings of data structures and algorithms. What these restructurings are, whether they improve performance to a substantial extent, and if so their conceptual and programming difficulty are not well understood. Nor is the question of performance portability; that is, do the modifications yield improvement on all types of shared address space platforms or only on SVM, leaving performance on other systems unchanged or even worse. These issues have important implications for the success and viability of SVM.

In this paper, we try to understand these programming and performance issues for parallel applications with a range of behaviors. We begin with well-written applications designed for hardware cache-coherent machines, and apply to them a set of systematic optimizations in different structured classes to improve their performance on SVM systems. The optimizations include padding and alignment, reorganization of major data structures, and algorithmic changes, typically going from the simplest to the most difficult. In particular, we are interested in answering four questions:

- Can application performance on SVM systems be improved significantly by certain classes of optimizations?
- How difficult are the optimizations that succeed, both conceptually and in implementation?
- How do these optimizations affect performance on hardware cache-coherent multiprocessors, with both centralized and distributed memory? That is, are the optimizations "performance-portable"?
- Are there any guidelines we can establish for programming on SVM systems, beyond or as distinct from programming for hardware cache-coherent shared memory?

We focus in this paper on relatively small-scale systems, with 16 processors.

The next section introduces the experimental platforms and applications we use. Section 3 describes the approach and methodology of the study in more depth. In Section 4, we present the classes of optimizations, what they translate to for different applications, and the results on the SVM platform. Section 5 explores the impact of the optimizations on hardware cache-coherent multiprocessors. Section 6 summarizes our results and some guidelines for programming on SVM that we found useful, and section 7 outlines future work.

2 Experimental Environment

This section introduces the platforms and applications we use in this paper.

2.1 Platforms

We use an SVM platform, a cache-coherent shared address space platform with physically distributed memory (a DSM platform), and a cache-coherent platform with centralized shared memory. Our SVM and DSM platforms are detailed simulators of the corresponding systems. For SVM, this is because we do not yet have available to us a real system with the protocols we want and the performance characteristics we consider realistic today. For DSM, simulation then allows us to keep many of the important characteristics comparable. (We have recently obtained an SGI Origin2000 system, and early results on it seem to validate the qualitative conclusions based on the simulator). Our bus-base system is a real machine, providing some realism, and we plan to do a fuller evaluation on real systems for both SVM and DSM. We use 16-processor systems in each case.

2.1.1 Shared Virtual Memory Platform

The shared virtual memory (SVM) platform we use simulates an all-software home-based lazy release consistency (HLRC) protocol [13], which has memory overhead and scalability advantages over non home-based protocols such as that in TreadMarks [12]. HLRC has recently been shown to equal or outperform non home-based LRC protocols as well, at least on the platform studied [21]. The simulator models an architecture of processing nodes connected by a commodity interconnect that is modeled on Myrinet [1]. In our experiments, each node can be considered to have a 200Mhz x86 processor running at 1 CPI (without memory effects). The data cache hierarchy consists of an 8KB first-level direct mapped write-through cache and a 512KB second-level 2-way set associative cache, each with a line size of 32 bytes. The peak communication bandwidth is 400MB/sec for memory buses and 100MB/sec for I/O buses (through which network packets flow). The page size is 4KB. Buffering and contention are modeled in detail at all levels except in the network links and routers themselves.

The HLRC protocol assigns each page to a *home* node. To alleviate the false-sharing problem at page granularity, HLRC implements a multiple-writer protocol based on using "twins" and "diffs". After an acquire operation, each writer of a page is allowed to write into its local copy once a clean version of the page (twin) has been created. Changes are detected by comparing the current (dirty) copy with the clean version (twin) and recorded in a structure called a *diff*. At a release operation, diffs are propagated to the designated home of the page (not to the other sharers). The home copy is thus kept up to date. Upon a page fault following a causally related acquire operation, the entire page is fetched from the home [21].

2.1.2 SGI Challenge

The SGI Challenge is a bus-based, symmetric shared-memory multiprocessor with centralized main memory. The machine we use has sixteen 150Mhz processors, each with separate 16KB first-level instruction and data caches and a unified 1MB second-level cache. The second-level cache line size is 128 bytes, and the bus bandwidth is 1.2GB/sec.

2.1.3 Distributed Shared Memory Platform

Our detailed DSM simulator models an aggressive cache-coherent shared multiprocessor with physically distributed memory and one 300Mhz processor per node. Every processor has separate direct-mapped 16KB first-level instruction and data caches and a unified 4-way set associative 1MB second-level cache. Caches are kept coherent across nodes by a distributed directory protocol [4]. The second-level cache line size is 64 bytes. Peak node-to-network communication bandwidth is 400MB/sec. Buffering and contention are modeled everywhere except the network links and routers.

We choose speedup (over the best sequential version) as our performance metric. The speedup of a particular application of P processors is defined as its execution time on P processors divided by its execution time on uniprocessor. When different classes of optimizations are applied, the speedup of an optimized version is measured with respect to the uniprocessor execution time of the original version the optimization starts with.

Given the differences in the platforms, we do not compare performance or even speedup directly across them (though we do collocate the results initially to illustrate that speedups are much worse on SVM for the original applications). Rather, we focus on the value of the optimizations within a platform. Of course, these too are affected by platform parameters, but our choices of reasonable modern parameters and the magnitudes of the effects we observe make this not a significant problem for the effects we are studying.

2.2 Applications

We choose applications to obtain good coverage along several axes. First we use both regular and irregular applications. Second, we explore applications with a range of behaviors: different inherent communication and data referencing patterns, and different access granularities to data that interact with SVM page granularity to produce different "induced" sharing patterns [14]. By fine-grained access we mean that the accesses to data by a process are not highly spatially contiguous, which usually implies that accesses from different processes (at least compared to page size) are interleaved at quite fine granularity in the address space. As per the classification in [14], we are also concerned with how many processors write (produce) and read (consume) a unit of communication or coherence. Third, we choose applications from different domains of computation. Our application suite contains 7 applications, each with several versions. Six are originally from the SPLASH-2 [18] suite, and one is a recently published parallel shear-warp volume rendering program [9, 15] that is also described in another paper in these proceedings [3]. Let us briefly describe the access patterns in the original applications. Details of the applications themselves can be found in [18] and the references there.

2.2.1 Regular Applications

LU performs the blocked LU factorization of a dense matrix. We begins with the non-contiguous version of LU, which uses the natural 2-d arrays to represent the 2-d matrix. Its inherent data sharing pattern (at word granularity) is one producer with multiple consumers. Read and write accesses are both fine-grained. Since a page spans multiple sub-rows from different blocks, it suffers false sharing and fragmentation.

Ocean simulates eddy currents in an ocean basin. It consists largely of nearest neighbor calculations on regular grids. Both its inherent and induced (at page granularity) data referencing patterns are generally one producer with one consumer. Read and write accesses are both coarse grained internally to a partition and along row-oriented partition boundaries, but fine grained along column-oriented boundaries; i.e. when a process reads a word from its neighbor along a column-oriented boundary, because of the way memory is laid out, it reads only a single word on each page. Thus, there is significant fragmentation in communicating remote data in pages at column boundaries.

2.2.2 Irregular Applications

Volrend renders three-dimensional volume data into an image using a ray casting method. Its inherent data referencing pattern on data that are written (task queues and image data) is migratory, while its induced pattern at page granularity is multiple producers with multiple consumers. Both the read accesses to the read-only volume and the write accesses to task queues and image data are fine grained, so it suffers both fragmentation and false sharing.

Shear-Warp renders three-dimensional volume data into an image using a shear-warp factorization algorithm. There are two phases in the rendering (see Figure 1) [15, 9]. First, the run-length encoded volume (not shown) is composited

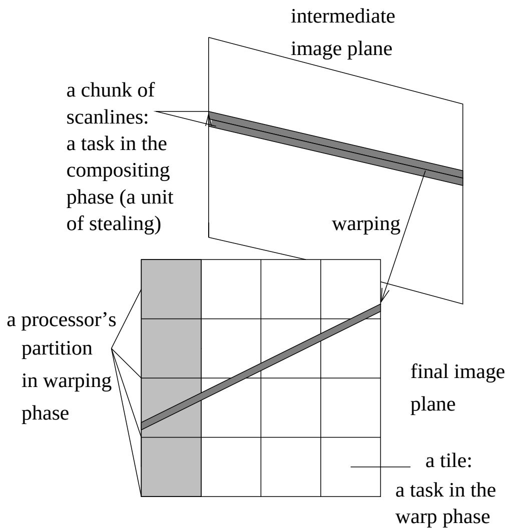

Figure 1: Shear-warp volume rendering algorithm.

into an intermediate image, by traversing the volume in scanline order slice by slice and writing the image. This is done by partitioning the intermediate image plane into chunks of scanlines assigned to processors in an interleaved manner, and having the processor traverse the appropriate scanlines of the volume so that only one processor writes a given scanline of the image. After the (distorted) intermediate image has been composited, it is warped into a final image. In the original application, the warping phase partitions the final image into blocks of tiles that are assigned to processors. Shear-Warp's accesses to volume data are much coarser grained since they follow along scanlines, but it suffers similar problems to Volrend particularly on image data.

Raytrace renders complex scenes in computer graphics using an optimized ray tracing method. Its sharing pattern is very similar to that of Volrend, but less predictable.

Barnes simulates the interaction of a system of bodies in three dimensions over a number of time-steps, using a hierarchical N-body method. Both its inherent and induced data referencing patterns for at least some data and phases (for example, tree building) are multiple-producer multiple-consumer. The read and write accesses are both fine grained, thus it suffers high false sharing and fragmentation. It also has a lot of lock-based synchronization when building its shared tree, which is very expensive on SVM systems.

Radix sorts a series of integer keys in ascending order. Its inherent data referencing pattern is one producer with one consumer, but its induced pattern at page granularity (in the permutation phase of the sort) is multiple producers with one consumer. Read accesses are coarse grained but write accesses are fine grained and scattered. It suffers substantial false sharing at page granularity.

3 Methodology

We begin with the applications as they appear in the SPLASH-2 suite (and for Shear-Warp as it appears in [9]). They are all quite well tuned for hardware cache-coherence. We perform data distribution on the SVM and DSM platforms as suggested in SPLASH-2. For LU and Ocean, we start from the "non-contiguous" versions that use 2-dimensional arrays to represent the 2-d matrix and grids. For Barnes, we in fact begin with the most natural data structures as used in SPLASH—not SPLASH-2—version. In this version, a processor's pointers to its particles and cells are maintained contiguously in the shared array but the particles/cells themselves are not.

We use problem sizes that are as large as or larger than the defaults in SPLASH-2, but that can be reasonably simulated. In particular, we run LU on a 1024×1024 matrix, Ocean on 514×514 grids, Volrend and Shear-Warp on a $256 \times 256 \times 225$ Computed Tomography head, Raytrace on the car data set with a 128×128 resolution image, Barnes on 16K particles, and Radix on 4M integers.

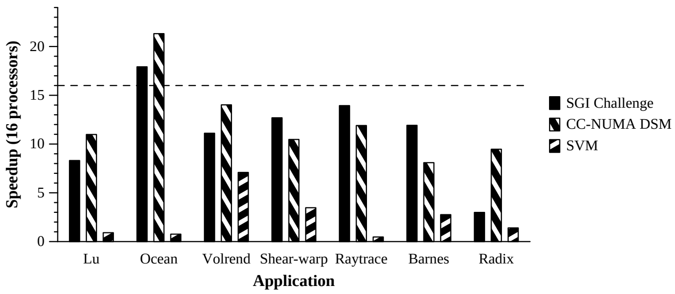

Figure 2: Speedups for the original versions across the shared address space multiprocessors.

To motivate this study, Figure 2 shows the speedups obtained by the applications on the different platforms. While the speedups cannot be fairly compared directly, since the platforms have different node parameters, the differences between SVM and others are clear. Generally speaking, all the applications have good to reasonable performance across the hardware cache-coherent systems, whether bus-based or with physically distributed memory. Performance on the SVM platform, however, is a different story. Many of the applications, like Volrend, Shear-Warp, Barnes, and Radix, do not perform very well. Moreover, applications like LU, Ocean, and Raytrace perform even worse than a uniprocessor. This implies that parallel applications written for cache-coherent multiprocessors often may not be ported directly to SVM and achieve good performance automatically. The challenge is to improve the parallel performance on SVM substantially, and understand what kinds of techniques this takes.

While classifying programming improvements is difficult, since they can be quite ad hoc, we use structured classes of optimizations starting from the simplest to the most challenging, improved by the structure used in [2]. Specifically, we divide optimizations into three classes:

- *Padding and Alignment* is the simplest optimization. It involves padding and aligning data structures (both major and minor, as appropriate) to the granularity of communication and/or coherence—cache line size for hardware cache-coherent machines and page size for SVM systems—so as to reduce false sharing and fragmentation of communication.
- *Reorganization of Major Data Structures.* This is harder to implement than just padding/alignment. Examples include moving from two-dimensional to four-dimensional arrays to represent two-dimensional grids [11], organizing records of particles by field rather than by particle, etc. The changes are performed to increase spatial locality, and thus to decrease fragmentation and false sharing.
- *Algorithm Redesign* is usually the most challenging optimization. It may involve performing synchronization differently, changing the partitioning method, changing the basic sequential algorithm for specific phases of the computation, or even changing the entire algorithm used to solve the problem.

The optimizations are usually applied cumulatively, one after the other. If the simple optimizations like padding and alignment are successful in improving the performance on SVM, that sends a very different message for the success of SVM than if it requires major programming or algorithmic enhancements beyond those needed for hardware CC-NUMA machines.

4 Improving Performance on Shared Virtual Memory

In this section we present the results for each application as it undergoes the classes of optimizations. For each application, we describe the specific optimizations that each class translates to, and analyze their impact on SVM performance. We also discuss the difficulties of diagnosing the performance problems that lead to the need for optimization, conceptualizing a solution, and implementing the solution. Finally, we describe the remaining bottlenecks in SVM performance even after all optimizations have been performed. We begin with the regular, predictable applications, whose performance is relatively easier to diagnose, and then move on to the irregular applications. The rest of this section is organized based on the fact that we have described the basic data accessing patterns and algorithmic properties of all the original applications in section 2.2, so here, we start from optimization steps for each application directly.

4.1 Regular Applications

4.1.1 LU

Padding and Alignment: The matrix allocated as a 2-d array is the main data structure in LU. Padding and aligning each sub-row within a block to a page would eliminate false sharing. However, there is little performance improvement since this change does not reduce fragmentation in communication (i.e. fetching more data than needed). This padding is also very inefficient storage-wise since even with the 32 by 32 blocks, we use only 32×8 or 256 bytes out of each 4KB page. Finally, it greatly complicates the indexing of elements in the 2-d array when padding is inserted within each array dimension.

Data Structure: To eliminate false sharing and fragmentation, this optimization uses an alternative data structure design, allocating the matrix as a 4-d array so that each block can be allocated contiguously in the address space (the "contiguous" version in SPLASH-2). Performance improves dramatically, and achieves a super-linear speedup of 18.7 for 16 processors. Figure 3, however, indicates that there is still a remaining bottleneck in data communication experienced by processor 10, though it is known that this overhead is very small on hardware cache-coherent machines. This bottleneck indicates that the high super-linear speedup is partially due to a reduction in local capacity or conflict misses compared to a uniprocessor execution, since speedups are being measured with respect to a uniprocessor 2-d array version which has much more conflict misses than with the optimized data structure. The high data waiting time for processor 10 is partly due to the barrier implementation in the SVM system, and partly due to the program's data accessing pattern. Processor 10 is chosen as the manager of the most important barrier in this application, and is responsible for dealing with the protocol related actions for that barrier. Being the first one to pass that barrier, it may not be able to fetch the data pages it needs from other processors until they pass the barrier as well. Processor 10 also fetches more pages than others, which turns out to be due to page alignment problems. By aligning the contiguous blocks assigned to the processors to page boundaries, we eliminate this bottleneck. The new data structure makes LU finally achieve a speedup as high as 20.6 for 16 processors.

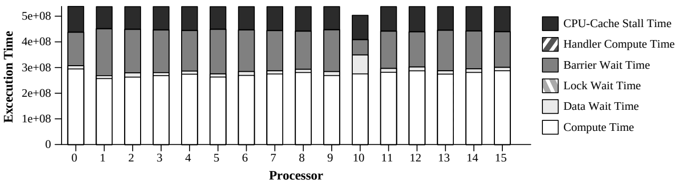

Figure 3: Execution time breakdown of LU contiguous version without padding/alignment. In this figure, and other figures in the rest of this section, Compute Time is the time the processor spends actually executing application instructions. Data Wait Time is the time spent waiting for data to arrive at remote page faults, i.e. the time spent waiting for communication. Barrier and Lock Wait Time are the time spent waiting at barriers and locks, respectively, including both the wait time and the overhead of the synchronization events. Handler Compute Time is the time the processor spends in protocol processing on incoming or outgoing transactions, including the time spent computing and applying diffs. CPU-Cache Stall Time is the time the processor spends stalled waiting for local cache misses to be satisfied. All times are in cycles.

Algorithm: A possible alternative partitioning algorithm is to assign blocks to each processor in a more complex and less structured way to improve load balancing; i.e. not to use the standard 2-d interleaved scatter decomposition of blocks. However, this compromises communication a lot, which is expensive, and turns out to be not beneficial for performance on SVM.

Summary: LU performs very well on SVM once the appropriate data structure is used and is padded and aligned to page boundaries. It is easy to understand the performance bottleneck and conceptualize the data restructuring. The implementation of the data restructuring into a 4-d array is painful, but it is also known to be very useful on hardware cache-coherent machines. In fact, the "contiguous" version of the SPLASH-2 LU uses this data structure. While simply padding and aligning the original 2-d array data structure is not enough, once the data structure is altered the access patterns become coarse-grained, padding and alignment helps SVM performance significantly. Algorithm redesign of LU to improve the performance further on SVM turns out to be difficult but unnecessary.

4.1.2 Ocean

Padding and Alignment: The grids allocated as 2-d arrays are the main data structures in Ocean. Like LU, the square-like sub-grids assigned to a processor are not contiguous in the address space, so there is a lot of false sharing and fragmentation. Simply padding and aligning each sub-row within a sub-grid does not reduce fragmentation and has all the same problems as in LU. Padding and aligning other more minor data structures is not very useful to performance either.

Data Structure: Using 4-d arrays to represent 2-d grids—with the first two dimensions specifying the partition (processor) and the next two identifying the particular grid point within the partition—allows partitions to be allocated contiguously, greatly reducing false sharing and fragmentation. It also allows the sub-grid assigned to a processor to be allocated in its local memory, reducing remote access and artifactual communication. The result is a speedup of 8.5 for 16 processors. This is much improved from the non-contiguous data structure, but is still not satisfactory. Figure 4 reveals that the time spent waiting at barriers is high, and the data communication overhead is both high and imbalanced across processors. Barriers are in general expensive in SVM with relaxed consistency models, since a lot of protocol activity and information exchange occur as part of them, and this application has many barriers. However, there is room to improve the data wait time.

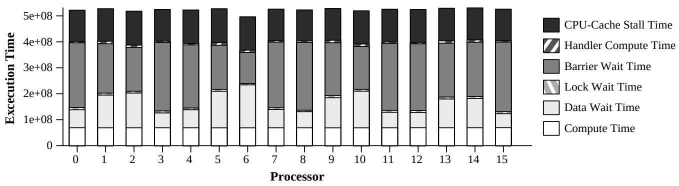

Figure 4: Execution time breakdown of Ocean contiguous version.

Algorithm: We mentioned in section 2.2 that near-neighbor accesses in Ocean are coarse-grained at row boundaries, but fine-grained at column-oriented boundaries. Figure 4 shows that even with 4-d arrays, processors that own partitions that share two column-oriented boundaries with other processors fetch almost twice as many remote pages than those having only one column-oriented boundary. To eliminate communication at the column boundaries, an alternative partitioning method is to assign blocks of n/p contiguous whole rows to each processor instead of square-shaped sub-grids. Although this has a worse inherent near-neighbor communication to computation ratio than with square subgrids, now the only communication in grids is coarse-grained along row-oriented boundaries. There is little fragmentation, and page-grained communication is in fact useful due to prefetching. An added substantial benefit is that partitions are now contiguous in the address space even with a two-dimensional array representation, so this much easier-to-program data structure does not cause false sharing. Figure 5 shows that data communication is quite balanced across processors with this partitioning method, and data wait time is no longer a major performance bottleneck. The speedup with 16 processors increases from 8.5 to 13.2, which is quite satisfactory (In both these cases too, local cache misses are reduced substantially compared to the uniprocessor execution, which is the 2-d non-contiguous version that has a lot of conflict misses, so speedup as we measure it is high despite its large barrier time). The major remaining bottleneck is the barrier cost.

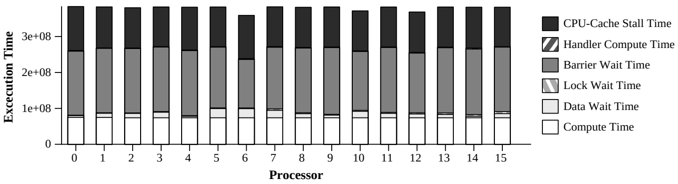

Figure 5: Execution time breakdown of Ocean row-wise version.

Summary: Ocean also achieves good performance on SVM after different classes of optimizations are explored. Like LU, understanding the performance bottlenecks in the original program and conceptualizing the data restructuring is quite easy, but the implementation of the data restructuring is painful. The algorithmic change was a little less obvious to come to, but in fact reduces programming effort greatly and is the most important optimization by far. Ocean demonstrates that SVM is very sensitive to data referencing and communication patterns, and that interactions with page granularity are often far more important than inherent algorithm properties like communication to computation ratio. It also reinforces that padding and alignment are in themselves not enough to improve performance, but can be important once more substantial problems have been solved.

4.2 Irregular Applications

4.2.1 Volrend

Padding and Alignment: In Volrend, the task queues constitute one important write-shared data structure, as different processors enqueue and dequeue entries. To decrease false sharing on task queues, each queue entry is padded and aligned to a page. This turns out not to be very beneficial. False sharing is reduced, but fragmentation is increased substantially and prefetching reduced when remote task queues are accessed.

Data Structure: Another source of the false sharing in Volrend is the final image plane: When a processor computes a task, it writes the image pixels corresponding to that task. The image plane is divided into p blocks or partitions which contain small square-like tiles of pixels. Each block is assigned to a processor, and a tile is the unit of stealing. The image is not very large, so as in LU and Ocean each page contains parts of the partitions belonging to different processors. By using a 4-d array to represent the 2-d image, we can ensure that the pixels initially assigned to a processor partition are contiguous in the address space. By then padding and aligning each partition to a page, we expect to reduce false sharing while not increasing fragmentation much. Surprisingly, this new data structure hurts performance, resulting in a speedup of only 6.27 down from the original 7.09 for 16 processors. The reason is that new data structure increases the cost of accessing the pixels, which interacts with task stealing to imbalance the computation a little more.

Figure 6 illustrates that data communication and lock-based synchronization are substantial bottlenecks in the original version. There is a lot of false sharing and fragmentation due to the image plane partitioning and dynamic tasking: The writes to different pixel tiles are separated by accesses to task queues, which are synchronized, so the false sharing is visible to the SVM protocol despite delaying coherence actions to synchronization points. The most interesting observation is that unlike on hardware cache-coherent machines, task stealing does not seem to help by reducing load imbalance across processors. The main reason is that the cost of the synchronization needed in stealing tasks is very high. This is because of the inherent cost of synchronization through explicit messages and the fact that coherence protocol activity is incurred at synchronization points, but especially because page misses within critical sections dilate them artificially, increasing serialization at the locks dramatically. (To diagnose the latter, we pretended in the simulator that the page faults within the critical sections are free, and saw the speedups rise to be almost perfect). The result is that by the time a processor succeeds in synchronizing to steal a task from a task queue, the owning processor is likely to have completed most of its (quite small) tasks, leaving little to steal. What we need is a better initial partitioning algorithm for dividing tasks among processors, so there is little need for stealing.

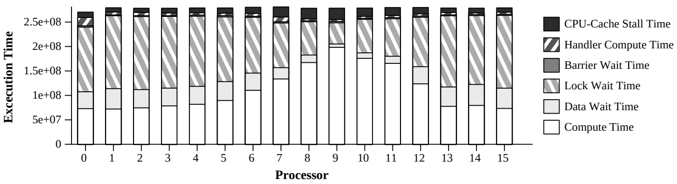

Figure 6: Execution time breakdown of Volrend for the SPLASH-2 version.

Algorithm: We break up the task blocks or partitions into smaller pieces (a few tiles each) and adopt a round-robin partitioning of blocks to processors to assign more, smaller blocks to each processor in an interleaved manner. This improves the initial load balance and achieves a speedup of 11.42 for 16 processors, more than 60% more efficient than the original algorithm. Computations are more balanced, stealing is reduced, and hence so is the synchronization overhead. Figure 7 demonstrates this effect.

To see the impact of stealing on SVM further, we eliminate stealing in Volrend with the new, more balanced task

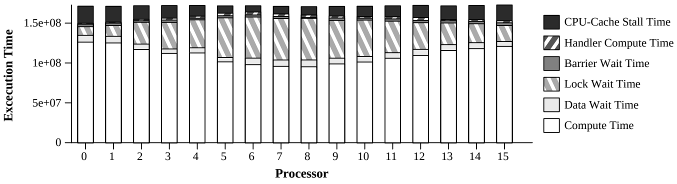

Figure 7: Execution time breakdown of Volrend with a more balanced task partition algorithm and stealing.

partitioning. This increases load imbalance and hence the time spent waiting at barriers, but reduces the time spent waiting at lock synchronizations. The two effects balance each other improving performance a little for this platform and data set. The speedup is now 11.70 for 16 processors. Figure 8 clearly shows that now the dominant overhead in execution time is the wait time at barriers with no task stealing, instead of at locks with stealing. At the same time, turning off task stealing decreases data wait time further, and increase the overall performance.

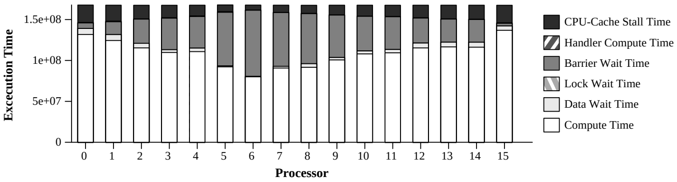

Figure 8: Execution time breakdown of Volrend with a more balanced task partition algorithm and no stealing.

Summary: Volrend shows that due to the high cost of synchronization in SVM systems, task stealing is not nearly so effective as it is on hardware cache-coherent machines. Particularly when tasks are relatively small, an initial task partitioning to get load balance and reduce task stealing to a minimum is very important on SVM systems. Understanding the performance bottleneck of Volrend on SVM was difficult. Without looking at the detailed execution time breakdowns obtained from the simulator it was tempting to believe that the problem was in poor interactions with page granularity in accessing the volume data, which has poor spatial locality. However this turns out to be a negligible problem compared to task queues and the image plane. Diagnosing that the problem with lock synchronization is caused mostly by dilation of small critical sections due to page faults required that we instrument the simulator to separate out the time waiting for the lock to become available from the overhead of actually fetching the lock and performing the associated protocol actions (or that we pretend that the page faults within the critical sections were free). In this application too, the image plane data restructuring optimization is more difficult to implement than the algorithmic optimization of changing the initial assignment, once the latter was diagnosed and conceptualized, while the algorithmic optimization is far more important. Improving performance further appears difficult. It requires further reducing the need for synchronization on task queues by improving the initial assignment of tasks to processors, which is difficult for unpredictable applications and data sets that are encountered in computer graphics. Or it requires reducing the cost of remote page faults.

4.2.2 Shear-Warp

Padding and Alignment: The main data structures written in Shear-Warp are the intermediate and final image planes. It is difficult to pad and align the final image plane because the tiles are assigned in an interleaved rather than contiguous manner. However, because the tasks in the compositing phase are chunks of entire intermediate image scanlines, padding and aligning the intermediate image plane to page boundaries is more efficient. However, since the tasks (scanline chunks) are quite small due to run-length encoding and are assigned in an interleaved manner for load balance, padding increases fragmentation and the performance improvement is only about 10%.

Algorithm: No data structure reorganization optimization is used. The breakdown of execution time in Figure 9 shows that the original algorithm has high data communication overhead and synchronization wait time at barriers. These two overheads are related. The data communication overhead is high due to the redistribution of intermediate image data between the phase of compositing the intermediate image and that of warping the intermediate image into the final image. Due to the different partitioning strategies adopted in these two phases for load balancing, most of the data that a processor reads in the warp phase are data written by other processors in the compositing phase.

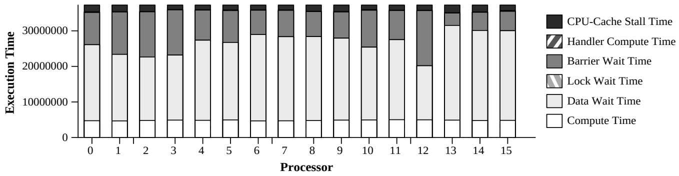

Figure 9: Execution time breakdown of original Shear-Warp.

This communication causes significant contention (the simulator shows that the cost per page fault is significantly higher than the unloaded cost) and is quite imbalanced in cost across processors (if not in actual number of page faults encountered), leading to large imbalances at barriers. The new algorithm totally changes the task partitioning scheme. It partitions the intermediate image initially into p (number of processors) contiguous blocks of scanlines not interleaved small chunks of scanlines as earlier. It uses the *same* partitions of the intermediate image for the compositing and warping phases. In the warp, a processor reads its partition of the intermediate image (which it itself wrote) and writes the relevant portions of the final image. Load balancing is accomplished by a combination of dynamic profiling of scanline costs to determine the initial partitions, and then stealing small chunks of scanlines if necessary. How this is done is beyond the scope of this paper but it relies on application insight. It is described in another paper in these proceedings [3].

The result is that redistribution of intermediate image data is greatly reduced, the expensive barrier between the two phases can in fact be eliminated, and write-synchronization on the final image during the warp can be eliminated by the use of local host rows at partition boundaries (one of the two adjacent processors is designated to warp the boundary rows, instead of each writing both). This greatly improves the speedup on SVM, from 3.47 to 9.21 on 16 processors. Figure 10 shows the breakdown of execution time for the new Shear-Warp algorithm.

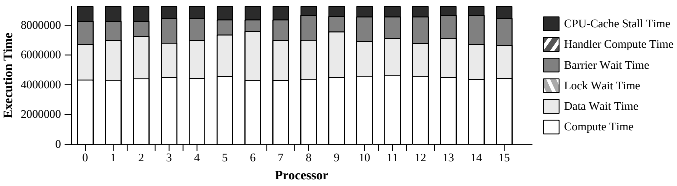

Figure 10: Execution time breakdown of optimized Shear-Warp.

Summary: Shear-Warp is an example in which restructuring the application can improve performance tremendously due to communication and memory system interactions, but it requires major changes to this parallel algorithm as well as insight into the application and not just the algorithms (for profile-based load balancing) [3]. It also demonstrates how contention invoked by expensive communication and synchronization causes load imbalance. Understanding the performance bottlenecks in Shear-Warp was difficult—particularly since the warp phase is relatively insignificant in sequential execution—as was conceptualizing and implementing the optimizations [3].

4.2.3 Raytrace

Padding and Alignment: Like Volrend, padding and aligning the write-shared task queues does not help much in Raytrace.

Data Structure: Like Volrend, the final image plane is decomposed into small square tiles of pixels. Unlike Volrend however, Raytrace uses round-robin rather than contiguous task assignment of tiles to begin with, so it is difficult to implement a new data structure to have all the tiles assigned to a processor allocated in a contiguous region and padded and aligned to reduce false sharing. Since this didn't help so much in Volrend anyway, we skip this optimization here.

Algorithm: While Raytrace is expected to perform well in parallel, the original SPLASH-2 version of Raytrace performs the worst on SVM of all the applications in our suite. The reason is not the poor interaction of the irregularly accessed scene data with page granularity, nor the false-sharing on the image plane. Figure 11 shows that what kills performance is synchronization overhead. Since the initial assignment of tasks is round-robin, task stealing should not be enough to cause such dramatic synchronization problems. It turns out that variables that keep track of some global program statistics in the algorithm are protected by locks, and the frequent accesses to them (e.g. once per ray) introduce the high lock overheads. The overhead of this synchronization is relatively insignificant on hardware-coherent platforms, where locks are cheap and are simply locks, but on SVM the locks cause a lot of expensive protocol communication, as well as tremendous serialization due to dilation of the critical section by page faults. By simply eliminating this lock, the performance jumps from "speedup" of 0.5 to a speedup of 11.05 for 16 processors.

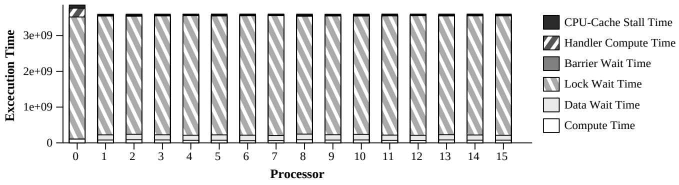

Figure 11: Execution time breakdown of Raytrace for the splash2 version.

Even with round-robin partitioning, the load imbalance can become somewhat high, since the behavior of the rays is much more unpredictable and varied than in Volrend as they bounce around striking objects. This imbalance evokes task stealing. Because of the greater unpredictability, it is not enough to simply turn off the task stealing as we finally did in Volrend. However, stealing does cause substantial synchronization overhead and contention at the task queues. To further reduce the synchronization cost, we split the task queues of each processor into two—one for local access and the other for potential stealing by other processors—and manage the movement of tasks between them. In this way, we eliminate the locks for local task queues, and at the same time diminish the contention for the shared task queues. This finally makes Raytrace achieve a speedup up to 11.72 for 16 processors.

The resulting breakdown in Figure 12 shows an interesting effect. Computation and data wait time are distributed almost evenly across processors, except for processor 0. Processor 0 has far less data wait time because a much greater fraction of its accesses are satisfied in its local memory. Pages are allocated in a round-robin manner. However, processor 0 is the one that initializes the scene data structures by reading them in from the scene description file, so it ends up with copies of many of the pages that were not initially allocated to it in the round-robin allocation (initialization time is serial, and is not counted in execution time). Since it therefore performs its tasks faster, it steals more and ends up doing more work. This data-access induced imbalance is the remaining bottleneck in Raytrace.

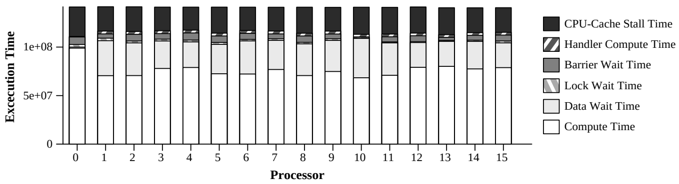

Figure 12: Execution time breakdown of optimized Raytrace.

Summary Raytrace also reveals that synchronization is very expensive on SVM, so using locks frequently for non-critical aspects like statistics gathering is very dangerous even though it doesn't matter on hardware cache-coherent machines. It also reinforces the much greater need for good initial partitioning when using task stealing, and for managing task queues more carefully on SVM to reduce synchronization and contention. Since the synchronization bottleneck experienced in Raytrace on SVM platforms was very unexpected, identifying it required help from performance tools. Once it was diagnosed, however, it was trivial to fix, though at the cost of some accuracy in statistics gathering. The subsequent, relatively small, improvement from changing the task queue structure was in fact much more difficult to implement. Imbalance in data communication cost is the main overhead to reduce for further performance improvement. This is difficult due to the unpredictable bouncing rays and the reasons discussed above.

4.2.4 Barnes

Padding and Alignment: We first padded and aligned the contiguous partitions (sub-arrays) of cell and particle pointers for each processor, as well as the data structures from which cells are allocated. This in itself does not help performance much. An alternative is to pad each particle, cell and leaf separately (recall that in this version the particles, cells or leaves assigned to a processor are not contiguous in the address space as the computation evolves). However, this is a huge waste of memory since the individual data structures are small. It also eliminates prefetching benefits.

Data Structure: The SPLASH-2 version of Barnes restructures the data to ensure that the cells and leaves assigned to a process are always in its local memory (they are allocated out of a local heap as the tree is built). This reduces fine-grained communication at least on data assigned to a processor. However, even this does not improve performance much: The speedup increases from 2.76 to 2.94 for 16 processors.

Algorithm: Figure 13 shows that even with the new data structure, Barnes still has high communication and synchronization overhead on SVM. The very fine-grained communication in the algorithm raises false sharing and fragmentation, but there is also a huge number of locks used. For 16k particles, for example, it uses almost 66k remote locks in 2 steps, i.e. 57 remote locks per processor. It turns out that the problem is not due to the irregular accesses in the major, force-calculation phase, but due to locks and irregular access in the phase of building a shared tree as well. This induces contention and imbalances in communication. In fact, compared to only about 2% of the execution time on a uniprocessor and a few percent on a hardware cache-coherent system, the tree-building phase takes 43% of the time with this algorithm under SVM.

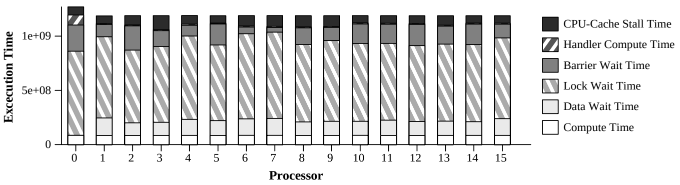

Figure 13: Execution time breakdown of Barnes for splash2 version.

To improve performance, it is necessary to change the way trees are built in parallel, particularly to reduce the frequency of locking. One algorithm, called Update-Tree, incremently updates the tree every time step instead of rebuilding it, by starting from the tree as it was in the previous time-step and moving only those particles that have crossed cell boundaries. Since the distribution evolves slowly, the hope is that not many particles will have to be moved. However, moving a particle needs to access remote data and acquire locks, and is expensive. This algorithm achieves a speedup of 5.56 for 16 processors for the whole application, which is substantially better than the original algorithm but still not good enough.

An alternative tree-building algorithm to reduce locking further, called Partree, operates by first determining the global extent of the distribution as before, then having each processor construct a local tree out of only its own particles, and finally merging the trees recursively. The merging algorithm is somewhat complicated, and we shall not describe it here. The building of local trees does not require any locking, though the merging does. The merging is also highly imbalanced; for example, the first processor to merge into the empty global root just redirects the root pointer, while later processors to reach the merge phase do successively more work (including locking) since more of the global tree is already there to be merged into. This imbalance in work, and particularly in locking increases the synchronization wait time at the barrier at the end of the tree building phase. Building local trees and then merging them also involves some extra computation compared to simply building a shared tree, but this is relatively a negligible effect. Due to the increased load imbalances, this algorithm obtains only a little better performance than Update-Tree, increasing the application speedup to 5.65 for 16 processors.

The tree construction phase is still the major synchronization and communication bottleneck. It still takes about 30% of the total execution time with the Partree algorithm, since the merging is too expensive. The final algorithm we examine is to change the partitioning used for tree-building. Instead of using the same partitioning of particles as in the force calculation to build the tree in parallel, we ignore this partitioning for the tree building phase instead we partition the domain spatially and assign the sub-spaces, instead of particles, to each processor. Each processor first builds the sub-tree only within its partitioned (equal) subspace, without synchronization, like in Partree. The difference is that when the global tree merging finally happens, little communication and synchronization is needed, since disjoint spaces are being merged is equal rather than particles and cells that may overlap. Each processor is required to obtain the particles that belong to its sub-space from the particle data structure (since some of these particles may have been assigned to other processors for the force calculation phase), but this cost is far less than the data communication and synchronization overhead in building the tree on SVM.

Note that since equal subspaces are assigned, the load imbalance in building local trees can be significant for nonuniform particle distributions. However, this is a negligible effect in comparison to the benefit. This Barnes-Spatial algorithm reduces the tree construction time significantly, leading to the speedup of 10.5 for the application on 16 processors. Figure 14 shows that overall computation time is quite balanced across processors. The main remaining bottleneck is that some processors have much higher data waiting time than others. This turns out to be due to contention in remote page access, since the number of pages fetched by different processors is actually quite balanced. As we have seen, such contention-induced imbalance is a frequent problem given the large granularity and high communication overheads on SVM systems.

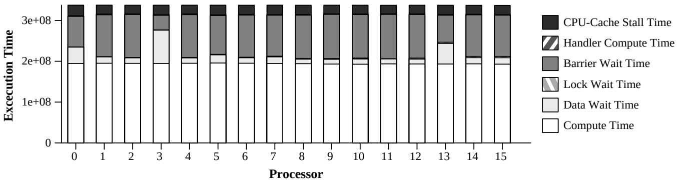

Figure 14: Execution time breakdown of Barnes for spatial version.

Summary: Through series of algorithm redesigns, Barnes finally performs quite well on SVM. The differences among the tree building algorithms implies that reducing locking as much as possible always helps performance due to the high overhead and serialization at locks, even at the cost of some imbalance or extra work. Barnes turned out to be a difficult application to optimize on SVM, from the viewpoints of understanding that the problem lay in synchronization in tree building and not in irregular access in force calculation, designing the algorithmic solutions, and implementing the improved algorithms. Improving performance further seems to require reducing barrier cost, perhaps partitioning the space less equally for late buildings and finding ways to reduce contention on remote page faults. (We experimented with simulating 32K particles instead of 16K, but the differences in speedup were small.)

4.2.5 Radix

Padding and Alignment: The main data structures that are write-shared in Radix are the global histogram and the arrays that are the destinations of the global permutation of keys in an iteration. It is very difficult to pad and align these data structures appropriately, due to the highly scattered and unpredictable remote writes to them by processors. Therefore, padding and alignment has little impact on performance in Radix.

Data Structure: For the same reasons, it is also difficult to adopt a new data structure to reduce false sharing and communication without modifying the algorithm, so we do not do this.

Algorithm: Figure 15 illustrates the major performance problems in Radix. We see that there is a very high synchronization overhead at barriers. Data communication is expensive as well, and is very imbalanced across processors. However, the number of pages fetched by each processor is quite balanced, indicating that the problem is contention due to the high data and control traffic (not synchronization) arising from scattered writes to remote data in the permutation phase. To reduce scattered remote writes, an alternative algorithm is to have a processor first write the output of its permutation into a local buffer, thus gathering the changes into consecutive subsequences of keys locally before writing them to the global output array in a less scattered way [18]. This improves the speedup from 1.4 to 2.24 on 6 processors, but it is still terrible.

Improving the performance of sorting seems to require a completely different algorithm that can use more contiguous patterns of access (e.g. sample sorting) or a very much larger number of keys to reduce false sharing and page granularity in the permutation [18]. As we will see later, Radix turns out to be a challenge for hardware cache-coherent machines as well.

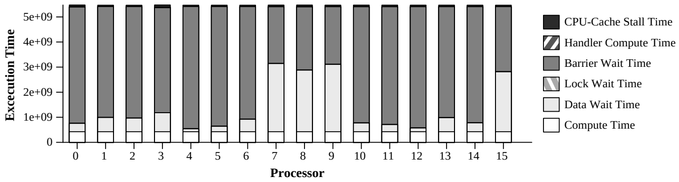

Figure 15: Execution time breakdown of Radix for splash2 version.

Summary: Radix is a difficult application on SVM due to the high communication traffic and contention. Understanding the exact source of the performance bottleneck is difficult since contention is difficult to diagnose. The "solution", such as it is, is relatively simple to implement (if a little harder to arrive at), but is not very successful. The major outstanding problems are still communication volume and contention.

In all, we have developed versions of many of the applications that greatly improve performance on SVM. Let us now turn to the performance portability to other platforms.

5 Performance Portability to SMP and DSM Platforms

As discussed in the introduction, the same optimized programs that were developed for SVM were also run on hardware-coherent SMP and DSM systems—namely the SGI Challenge and a detailed simulator of a CC-NUMA DSM machine—to study the portability of the performance optimizations. Figure 16 summarizes the effects of different optimization classes across applications on the different shared address space platforms. Let us look at the performance impact of these classes of optimizations on hardware cache-coherent machines.

5.1 Impact on the SMP Platform

Generally speaking, the optimizations used for SVM either deliver the same performance as the original programs on the SGI Challenge, or just a little better but not enough to really be worthwhile. The reason for some super-linear speedups (e.g. Ocean) was discussed earlier. This is due to the high memory stall overhead in the uniprocessors execution for the original "non-contiguous" version. Local misses decrease even in the parallel original version (for cache capacity reason) and more so with the optimized data structure where conflict misses are less problem. Note that on the hardware cache-coherent platforms, Ocean performs best with the square partitions and the 4-d array data structures, rather than with the row-wise partitions.

5.2 Impact on the CC-NUMA DSM Platform

On the CC-NUMA platform, the performance advantages of the optimizations are more pronounced than on the SMP. Note that data distribution is performed in all cases where it is reasonably allowed by the algorithms and data structures, and is not treated as an optimization in itself. Nonetheless,

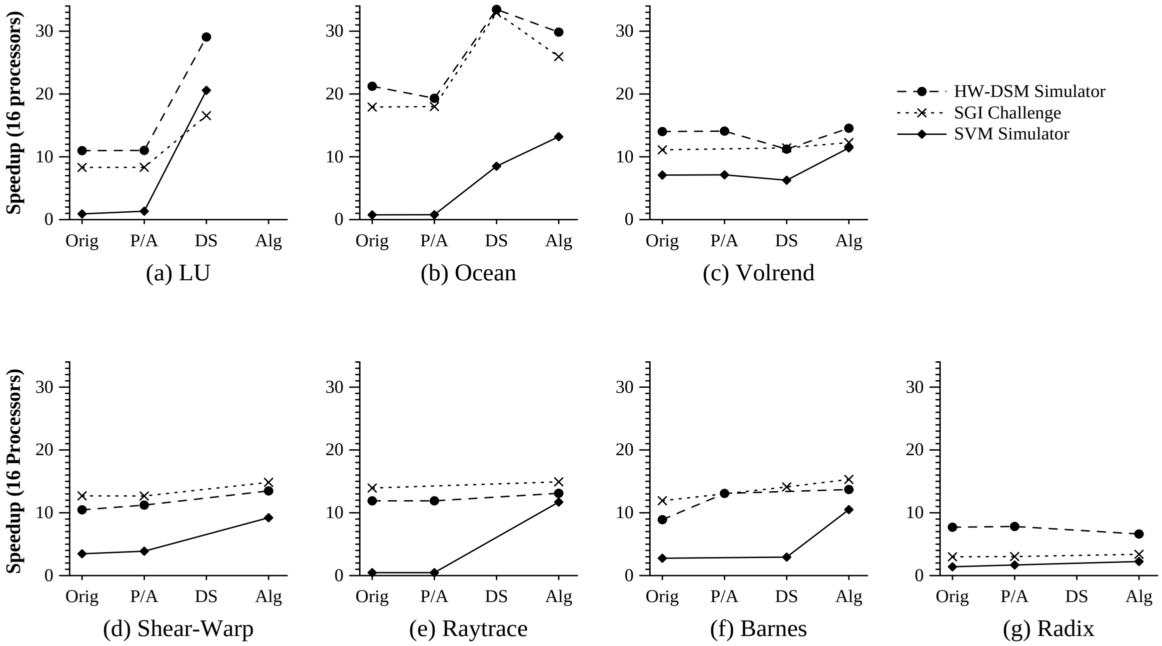

Figure 16: Performance with different optimization classes across shared-address-space multiprocessors. Here, Orig means the original algorithm we began with; P/A means optimizations with padding/alignment; DS means optimization with data structure reorganization; Alg means optimization with algorithmic change.

the performance improvements are much smaller than on SVM systems, and are sometimes insignificant. Other than efficient support for finer granularity and reduced overhead per communication, one major difference is that synchronization is much cheaper on CC-NUMA systems: Synchronization does not involve coherence protocol activity—which happens at access time—and cache misses that occur inside critical sections do not dilate the critical sections anywhere near as much as page faults. Thus, unlike on SVM, dynamic task queues and task stealing turn out to be quite inexpensive and hence very effective in providing load balance. Starting from the SPLASH-2 applications, load balance often turns out to be the most important performance issue on hardware coherent machines. To compare the effect of task stealing on the CC-NUMA platform with that on SVM, we can look at Volrend as an example in Figure 17. Running on both platforms of the application here uses the algorithmic optimization; one uses task stealing, the other does not (the tradeoff for SVM was discussed earlier).

To summarize, the optimizations that are beneficial on SVM are generally portable to hardware cache-coherent machines (in the sense that they do not hurt but usually help performance), but they have dramatically smaller impact. "Optimizations" for SVM that compromise load balance to improve communication and synchronization behavior can hurt performance on hardware cache-coherent machines, since communication and synchronization costs are much smaller in the latter so load balance matters more. The Radix sorting application does not perform well across all platforms, especially on SMP and SVM platforms. The heavy communication and capacity traffic in Radix hurt its performance on bus-based cache-coherent machines due to the bus bandwidth limitation, while the large remote data referencing and communication (and especially the associated contention at home nodes due to both data and coherence messages) hurt its performance on DSM machines.

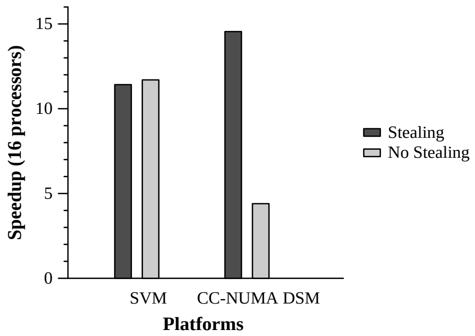

Figure 17: Speedups of Volrend with the algorithmic optimization with and without stealing on SVM and CC-NUMA DSM simulators.

6 Discussion and Conclusion

Through this study, we have found that many applications that perform well on moderate-scale hardware cache-coherent machines do not initially port well to SVM platforms. In addition to the expected differences due to granularity and overhead, aspects of performance such as synchronization that are not paid much attention to in hardware cache coherent machines matter a lot more in SVM, and phases of the computation that are not significant on moderate-scale CC-NUMA machines can become dominant under SVM. This is particularly true of irregular applications. The good news is that through algorithmic or programming effort, it appears that most of the applications can be made to perform quite well on SVM systems of the (relatively small) scale we have examined. The bad news is that the improvements needed are often substantial. Simple padding and alignment of data structures to page granularity is not the answer, although it can be ultimately important when used together with other types of optimizations. Padding does reduce false sharing, but the increased fragmentation and the loss of prefetching usually undermines the benefit. Also, in many applications where the natural data structures to be padded are many and very small, e.g. particles, so padding to page grain can waste much of the machine's memory. To achieve satisfactory performance on SVM, the optimizations needed are often algorithmic changes—either in partitioning or in the nature of the parallel algorithm—and require insights into both the application as well as the key aspect of SVM.

Qualitatively, we can divide our experience with each application into (1) the difficulty of understanding the major performance bottleneck for SVM, (2) the difficulty of arriving at or conceptualizing a solution, and (3) the difficulty of implementing the solution. This qualitative view is summarized in the following table.

Application	Understanding Bottleneck	Conceptualizing Solution	Implementing Solution
LU	easy	well known	painful
Ocean	easy	well known	painful
Volrend	needed tools	moderate	easy
Shear-Warp	difficult	difficult	difficult
Raytrace	needed tools	moderate	easy
Barnes	needed tools	difficult	difficult
Radix	moderate	difficult	difficult

In most cases, especially the irregular applications, at least one of the steps needed was quite nontrivial. In particular, the detailed simulator served as an excellent though slow performance debugging tool, which enabled us to find bottlenecks and understand their reasons (whether data wait time and imbalances in it are due to the number of page faults or contention, whether synchronization time is due to the overhead of synchronization or due to serialization because of a dilated critical section, which phase of the computations the problems occur in, how well the placement of pages is working, etc). Incorporating the ability to deliver such information in real SVM systems would be very useful.

Finally, our experience has shown that some common types of useful optimizations can be identified for SVM, particularly for irregular applications. One of our longer-term goals was to see if we can establish a set of programming guidelines for SVM systems, beyond those used for hardware cache-coherent multiprocessors. From this experience, we emerge with the following common themes.

Padding and Alignment are usually most useful after other optimizations like data structure reorganization have been used to make access patterns coarser-grained.

Task Queues and Task Stealing should be used with care. Stealing can induce very high communication, contention and serialization costs on SVM, so it should be minimized. This means that a nearly load balanced initial assignment of tasks is much more important in SVM than in hardware-coherent systems when task stealing is used, even at the cost of increasing some inherent communication or loss of locality.

Synchronization events are very expensive on SVM for several reasons mentioned earlier, so it is crucial to reduce the frequency of synchronization events, especially fine-grained locking.

Locality versus Load Balance. While with task stealing it is important to start with a more balanced assignment even at the cost of some locality since the cost will be more than regained by reducing stealing. Without task stealing, communication and data locality often play a much more important role than computational load balance compared to hardware cache-coherent systems. In fact, the load imbalance that ends up hurting performance most is usually an imbalance in the amount or cost of communication. Contention is a particularly important reason for imbalances in communication cost among processors.

Data Distribution. Particularly with home-based protocols, proper data distribution is important, just as it is on hardware cache-coherent DSM systems. This is true not only to reduce the number and hence latency of remotely satisfied page faults, but also to reduce contention generated by these page faults.

7 Future work

This study has focused on relatively small-scale systems, and has been somewhat limited by the use of simulation, both in problem sizes and in realism. We plan to expand our study to real systems, both for hardware DSM and software SVM. We have recently started experimenting with an SGI Origin2000, a scalable shared-address-space multiprocessor with physically distributed shared memory. Our early results on 16-processor Origin, running larger problems, validate the qualitative conclusions based on the DSM simulator. We are also developing and tuning SVM protocols on a platform of Pentium Pro PCs and PC SMPs connected by a Myrinet network, and would like to validate our results there. In particular, it will be interesting to study how to take advantage in the applications of the two-level communication hierarchy when SMP nodes are connected by SVM, and how the programming and performance issues change in this case. We would also like to validate that our conclusions indeed hold true when full-sized problems are run on real systems.

This study has also focused on small-scale machines (16 processors). When we use real systems, we plan to investigate the issues with larger numbers of processors. For example, it would be interesting to see if the optimizations that are useful at smaller scale for SVM but not CC-NUMA become more useful for CC-NUMA machines at larger scale [2], and how problem size affects these results. We would also like to look at the impact of these optimizations on systems that support fine-grained coherence with either more commodity-oriented controllers [16] or in software [10, 19], thus completing the performance portability picture, and to enlarge our coverage by including more applications in our suite. Finally, it may be interesting to examine how optimizations performed on the applications compare with the use of custom protocols to improve the performance of the original applications on commodity-based protocols [6].

Acknowledgment

We would like to thank Angelos Bilas and Liviu Iftode for their help with the SVM simulators.

References

[1] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[2] Holt C., Singh J. P., and Hennessy J. Application and Architectural Bottlenecks in Large Scale Distributed Shared Memory Machines. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, pages 134–145, May 1996.

[3] Jiang D and Singh J.P. Parallel Shear-Warp Volume Rendering on Shared Address Space Multiprocessors. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1997.

[4] Lenoski D., Laudon J., Joe T., Nakahira D., Stevens L., Gupta A., and Hennessy J. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 92–103, May 1992.

[5] Agarwal A. et al. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22th International Symposium on Computer Architecuture*, pages 2–13, June 1995.

[6] Babak Falsafi et al. Application-specific Protocols for Shared Memory Multiprocessors. In *Proceedings of Supercomputing95*, November 1995.

[7] Heinrich M. et al. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the 6th International Conference on Architecutural Support for Programming Language and Operating Systems*, pages 274–285, October 1995.

[8] Kuskin J. et al. The Stanford Flash Multiprocessor. In *Proceedings of the 21th International Symposium on Computer Architecuture*, pages 302–313, April 1994.

[9] Lacroute P. G. *Fast Volume Rendering Using a Share-Warp Factorization of the Viewing Transformation.* PhD thesis, Stanford University, 1995.

[10] Schoinas I., Falsafi B., Hill M., Larus J., Lucas C., Mukherjee S., Reinhardt S., Schnarr E., and Wood D. Implementing Fine-Grain Distributed Shared Memory On Commodity SMP Workstations. Technical Report 1307, Computer Sciences Department, University of Wisconsin–Madison, March 1996.

[11] Singh J.P., Joe T., Hennessy J., and Anoop Gupta. An Empirical Comparison of the KSR-1 ALLCACHE and Stanford DASH Multiprocessors. In *Supercomputing '93*, November 1993.

[12] P. Keleher, A.L. 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, January 1994.

[13] Iftode L., Singh J. P., and Li K. Scope Consistency: a Bridge Between Release Consistency and Entry Consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[14] Iftode L., Singh J. P., and Li K. Understanding Application Performance on Shared Virtual Memory Systems. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, pages 122–133, May 1996.

[15] Singh J. P., Gupta A., and Levoy M. Paralle Visualization Algorithms: Performance and Architectural Implications. *Computer*, 27:45–55, 1994.

[16] Pfile R. Typhoon-Zero Implementation: The Vortex Module. Technical Report CS-TR-95-1290, Computer Sciences Department, University of Wisconsin–Madison, October 1995.

[17] Reinhardt S., Larus J., and Wood D. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21th International Symposium on Computer Architecuture*, pages 325–336, April 1994.

[18] Woo S.C., Ohara M., Torrie E., Singh J. P., and Gupta A. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, June 1995.

[19] D.J. Scales, K. Gharachorloo, and C.A. Thekkath. Shasta: A Low Overhead, SOftware-Only Approach for Supporting Fine-Grain Shared Memory. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[20] Radhika Thekkath, Amit Pal Singh, Jaswinder Pal Singh, John L. Hennessy, and Susan John. An Evaluation of the Convex Exemplar SP-1200. In *Proc. Intl. Parallel Processing Symposium*, April 1997.

[21] 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*, October 1996.