Irregular Applications under Software Shared Memory

Liviu Iftode, Jaswinder Pal Singh and Kai Li
Department of Computer Science
Princeton University
Princeton, NJ 08544
liv,jps,li@cs.princeton.edu

Abstract

Shared Virtual Memory (SVM) provides an inexpensive way to support the popular shared address space programming model on networks of workstations or personal computers. Despite recent advances in SVM systems, their performance for all but coarse-grained or regular applications is not well understood. Nor is there an understanding of whether and how fine-grained, irregular programs should be written differently for SVM, with its large granularities of communication and coherence, than for the more familiar hardware coherent at cache line granularity. In this paper we try to understand the performance and programming issues for emerging, irregular applications on SVM systems. We examine performance on both an aggressive all-software system as well as one with a little hardware support in the network interface. We also present approaches to improve the performance of irregular applications at both the programming and the system level. As a result of our experiences, we identify a set of guidelines and techniques that pertain specifically to programming SVM systems, beyond the guidelines commonly used for programming hardware-coherent systems as well. We also present a further relaxation of the memory consistency model, called scope consistency, which is particularly effective for such applications.

1 Introduction

The shared address space programming model is widely considered attractive for parallel computing. However, providing efficient support for a coherent shared address space has high engineering cost, and has been demonstrated convincingly only with substantial hardware intrusion into the underlying processor-memory system. Given the importance of staying on the technology curve, a lot of research attention has recently been devoted to supporting a shared address space with less intrusion into the underlying commodity parts. The space of proposals is diverse, with various efforts trying to integrate the communication controller that provides the necessary support further and further away from the processor-cache subsystem. These efforts include using a flexible customized controller integrated on the memory bus [11], a commodity controller on the memory bus [14], and a decoupled controller partially on the memory bus and partially on the I/O bus.

One extreme in the spectrum that predates these efforts is to support the coherent shared address space entirely in software, using networks of commodity workstations or personal computers with no additional hardware support. This approach, called shared virtual memory (SVM), uses the operating system's support for virtual memory management to provide naming, replication and coherence at the granularity of pages instead of cache lines [12]. However, in addition to the problem of software overhead for communication it introduces, this approach suffers from problems related to the large granularity of communication and coherence (an entire page). The two major problems are false sharing and fragmentation. False sharing occurs when two processors access logically unrelated data that happen to fall on the same page and at least one of them writes the data, causing the coherence mechanism to ping-pong the page between the processors. Fragmentation causes unnecessary traffic when a processor needs only a part of the page but has to fetch the entire page. These disadvantages often dominate the advantages of the large communication granularity; namely prefetching and amortizing the communication overhead.

Two recent interesting trends have made the inexpensive SVM approach more promising , however. First the advent of relaxed memory consistency models has helped alleviate the performance problems due to large granularity by delaying the need of propagating data or protocol messages (e.g. invalidations) until synchronization points. As we shall see, this alleviates the effects of false sharing as well as fragmentation, and reduces both the frequency of communication as well as the amount of data communicated. It is now widely accepted that exploiting relaxed models is essential for SVM systems to possibly be successful.

Technical Report TR-514-96.

The second trend indicates an interesting convergence in the directions taken for commodity-oriented shared address space support coming from the two extremes. While cache-line based approaches are trying to reduce hardware support, some SVM systems have explored integrating a small amount of hardware support into the network interfaces of commodity workstations or PCs. An example is the network interface of the SHRIMP multicomputer [3], a network of commodity PCs being built at Princeton University. The network interface supports a fine-grained *automatic update* mechanism. The idea, explained later, is to have snoop the memory bus for writes to certain remotely mapped pages (possible without modification of the bus or its protocol, as long as these pages are write-through) and propagate these writes at word granularity to the remote copies. As we shall see, a software-based SVM coherence protocol can leverage this mechanism to reduce the overhead of communication

Despite all these developments, the performance of SVM or *software shared memory* systems is very sensitive to the data referencing patterns of applications, much more so than that of hardware-coherent systems [7]. Performance evaluations so far have focused on applications that are quite regular or have coarse-grained data sharing patterns (such as traditional array-based scientific applications) [5, 4, 6]. They indicate that SVM can work reasonably when data sharing is either small or coarse-grained. As experience with parallel computing grows, however, the applications of interest become increasingly complex and irregular, both in emerging computational domains and also as traditional scientific domains model their problems more accurately (nature is, after all, irregular and dynamic). These applications have naturally fine-grained, dynamically changing and unpredictable data access patterns. While many such applications have been demonstrated to perform very well on hardware cache-coherent architectures, their performance and potential under SVM is not well understood at all.

In this paper, we try to understand the performance and programming issues in emerging, complex irregular applications on SVM systems. To establish a basis for comparison, we begin with applications that have been written assuming hardware-coherent shared memory and perform very well on such platforms, delivering close to linear speedups on 32-64 processor systems for the problem sizes we use. Four interesting questions arise:

- How well do these applications perform "out of the box" using the two SVM approaches described above (all-software, and the hardware-assisted automatic update approach)?
- What are the key bottlenecks that are most severe compared to hardware-coherent shared memory at cache line granularity?
- Can we modify the programs or data structures in well-defined ways to alleviate these bottlenecks (without changing the algorithms, and without resorting to region-based or object-based approaches that change the programming interface and impose a much higher programming burden)?
- Are there common guidelines we can establish for programming irregular applications for SVM, beyond or as distinct from programming for hardware-coherent shared memory.

We address these questions in this paper, using two very different irregular N-body applications and three applications from computer graphics (namely, volume rendering, ray tracing and radiosity). While we present results for both types of SVM schemes, using an aggressive example from each, our goal is not to compare particular schemes or approaches in any detail but rather to understand the interactions of applications with the mechanisms they use. In addition to understanding the performance of these irregular applications in the SVM context and providing some guidelines for programming them, we also present one system-level enhancement that can improve performance for such applications.

The next section discusses shared virtual memory further, describing relaxed consistency models and the two protocols we use as examples: lazy release consistency and automatic update release consistency. Section 3 describes the simulation environment and architectural parameters we use in our experiments. The next four sections discusses the individual applications and addresses the above questions for them. Section 10 provides guidelines for programming irregular, fine-grained applications for SVM systems that emerge from our work, and Section 9 discusses a new consistency model that can enhance performance. We conclude in Section 11.

2 Shared Virtual Memory

The basic idea in SVM is to provide the illusion of shared memory by exploiting the virtual memory mechanism. The local memory of processor both hold locally allocated data, and also serve as caches for non-local pages in the global virtual address space, as shown in Figure 1.

The virtual global shared address space is kept through the local page tables. Valid page table entries exist for those shared pages which are currently cached in the local memory. When a node attempts to access a shared page that is not allocated or cached

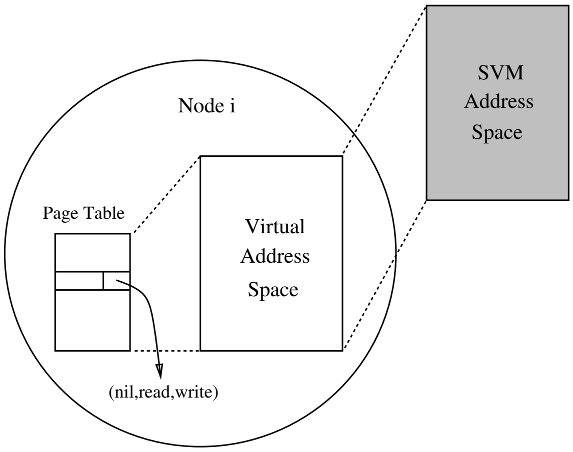

Figure 1: Shared Virtual Memory

locally, a page fault occurs. In handling the fault, the SVM system fetches a copy of the page into the local memory from the remote node and establishes the mapping.

SVM systems usually maintain coherence among multiple page copies with an invalidation-based protocol. Early approaches [12] implement the sequential consistency model, which expects updates to shared memory to become visible in the same order at all nodes, just as if they had been produced by concurrent processes on a uniprocessor system. They implement this model by propagating the effects of writes through coherence actions (invalidations typically) as soon as the writes happen. This approach has two drawbacks. First it is very sensitive to the false-sharing that occurs when different processors read and write different variables that happen to fall on the same page. Every write to the page may result in an invalidation and the immediately following read in a page miss. This causes lot of unnecessary communication of pages, and the frequent communication causes high protocol overhead since the protocol runs in software. The second problem is fragmentation which arises from the amount of data being much smaller than the page that is fetched. Relaxed consistency models help to alleviate these problems.

Under relaxed consistency models, coherence actions do not need to be propagated until synchronization events, so the invalidations and consequent read misses do not occur until the synchronization events. The intuition behind these models is that a user cannot distinguish the behavior of a system running under such relaxation from one that uses a sequential consistent model, as long as programs are well written without data races. Thus, once processes each have a copy of a page after a synchronization point, they can each read and write that copy heavily (without any coherence action or data communication) until the next synchronization point. At a synchronization event, the page will be invalidated and the processor will incur a page miss on the next read to the page.

In fully exploiting relaxed consistency, a pge may be written locally by multiple writers. A processor that accesses the page after a subsequent synchronization point must see the latest data for the page, so these writes must somehow be merged into the page that the processor accesses. A protocol that accomplishes that is called a *multiple writer protocol.* Let us describe the two multiple writer protocols we use, one an all software SVM system and the other for a SVM with automatic update support. Both protocols are assumed to use a lazy release consistency so that invalidations are propagated only at acquire synchronization points (e.q a lock or departure from a barrier) not at the preceding release point (e.g. unlock or barrier arrival).

All Software Release Consistency

Both Release Consistency (RC) [] and Lazy Release Consistency (LRC) [] have been implemented as software multiple-writer protocols for SVM. Munin [1] and TreadMarks [8] are examples of RC and LRC, respectively. In these systems, a processor locally records the changes it makes to each shared page between two synchronization events. The record of changes for a given page is called a *diff*. At the first access to a page following an invalidation, the page fault handler must collect and merge these diffs to reconstitute the coherent page. LRC is more aggressive in reducing communication so, we focus on it.

Figure 2 shows an example of how the all software LRC protocol actually works. **P0** and **P1** are two processors actively writing the same page. To detect the writes, shared pages are mapped read-only when they are first brought in on a page miss. At the first write, a protection violation occurs. At this time, the SVM system creates a clean copy (called a **twin**) of the faulting page, and removes the write protection. In our example this happens when **P0** writes variable **a** and when **P1** writes **c**. Subsequent writes before the next synchronization event, like the write of variable **b** at **P0** occur without system intervention. When a processor performs a *release* synchronization event (such as an unlock or an arrival at a barrier), the writes are detected by comparing the current dirty copy with the clean copy (twin), and the differences are recorded as a **diff**, a run-length encoded record of the modifications to that page.

At the next acquire synchronization event, issued by a processor, an invalidation-based protocol invalidates the copy of the modified page. Eventually **P0** accesses the page and incurs a page fault. The page fault handler requests the diffs for that page from the other processors that may have more recent copies (a

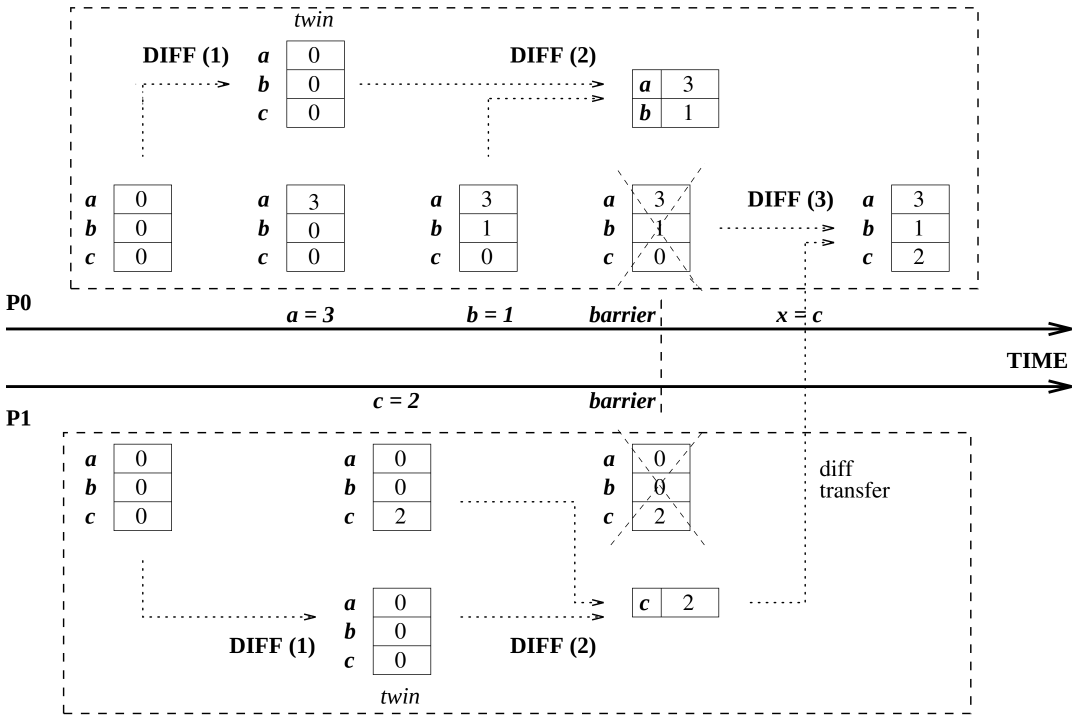

Figure 2: All Software Multiple Writer Protocol. The top and bottom dashed boxes show events on two processors; time proceeds from left to right

virtual timestamp mechanism [9] that we shall not describe here is used to determine this) and causes the relevant diffs to be fetched from those processors (**P1** in our case). These diffs are then applied to the local copy in the right order.

The main benefit of the all software LRC scheme is that it reduces the communication traffic and the number of times protocol overhead is incurred by reducing both false sharing and fragmentation. Fragmentation is reduced not only by reducing the number of communications but also by sending diffs (i.e. only the modified words plus the diff information overhead) instead of whole pages. The disadvantages of this scheme are as follows. First, the creation of a twin, the creation of a diff and the application (merging) of diffs to a page all have software overhead, and also pollute the processor cache. Second, to obtain diffs more than one other processor may have to be visited. Third, even when consecutive diffs of the same page are obtained from one place, they are maintained as separate diffs so the faulting processor may receive several updates even when they correspond to the same word. Now let us see how LRC may be supported with automatic update.

Automatic Update Release Consistency

Recall that automatic update is a mechanism that automatically propagates updates at fine granularity if a mapping has been established between the local page being written to and a remote page. For example if local page A at processor P0 is mapped to page B at processor P1, then all writes which are performed locally on page A are also automatically propagated to page B at node P1.

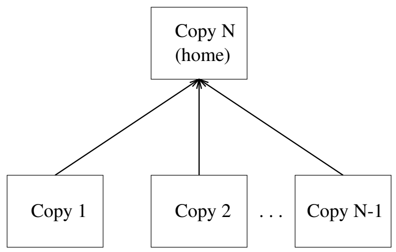

Figure 3: AU-based Write Collection

This simple hardware support is enough to build a multiple writer protocol [6, 10], as long as coherence is still supported in software as before. The basic approach is to have a *home* memory for every shared page (just as in hardware coherent machines) and to set up mappings such that writes to other copies of a page get automatically updated at the home. (see Figure 3). The home copy is thus always kept up-to-date by this mechanism. To keep the other copies

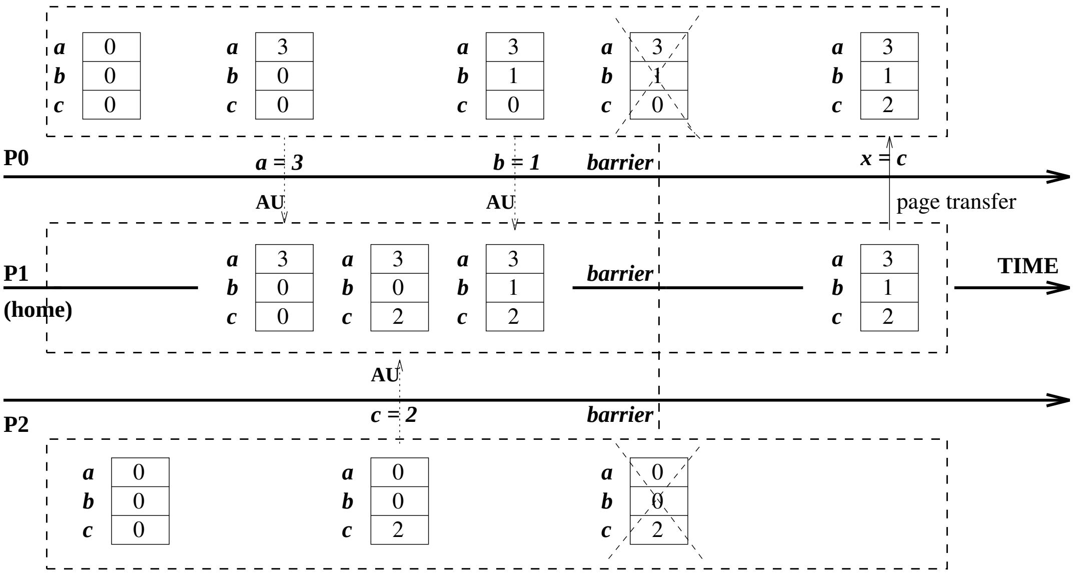

Figure 4: Automatic Update-based Multiple Writer

coherent an invalidation-based software protocol must supplement the automatic update hardware support.

Figure 4 shows an example of how the AU-based multiple writer protocol works. *P0*, *P1* and *P2* are three processors writing the same page. Assume *P1* is designated as the home. Consequently, writes to variable *a* and *b* and *c* performed by *P0* and *P2* are propagated to *P1* as well. At the barrier, the software invalidation-based protocol invalidates both copies of the modified page. Eventually *P0* accesses the page again when reading *c*, at which time the software protocol will update the local copy by transferring the entire page from the home (note that no diffs are used).

The basic tradeoffs between AURC and LRC are simple. The automatic propagation of writes by AURC can lead to unnecessary traffic and hence higher bandwidth requirements (suppose that the a processor writes the same word on a mapped page multiple times). Also, a page miss brings in the whole page, rather than only the diffs (however, if a given memory allocation is less than a page than only the allocated part of the page is sent) On the other hand, AURC does not suffer the software overheads of write detection, twin and diff creation, and write merging. And on a miss, AURC obtains the page from a single location (the home) rather than having to obtain diffs from possibly multiple writers. Generally, an AU based system will tend to have higher communication traffic (communication bandwidth needs) while an all software system with diffs will tend to have more protocol processing overhead. How these tradeoffs translate to end performance depends on the application's sharing patterns and granularities, as well as on architectural parameters such as available bandwidth and processor speed.

3 Simulation Environment

To understand the application performance we run a set of irregular applications through a detailed SVM simulator built using the TangoLite reference generator. All instruction in our simulated processors as well as cache hits cost 1 CPU cycle. We simulate in detail both the all-software shared memory (all **SSM**) scheme as well as an AU-based software shared memory (AU-based **SSM**) Since our main goal in this study focuses on irregular application behavior and not on any particular architecture evaluation, we assumed a generic but aggressive next-generation architectural model.

The model assumed a 300 MHz 1 CPI processor, a 800 MB/sec memory bus , 266 MB/sec I/O bus, 200 ns memory latency, 1 MB of one level two-way set associative cache, with 64 bytes cache-line and a 8 cache-line deep merging write-buffer. The merging write-buffer absorbs individual writes in any cache-lines it contains except for the one whose transfer is in progress. We also assume that the cache policy is programmable per page and can be either write-back with write-allocate or write-through with write-around programmable. We also assume an aggressive memory-mapped network interface connected to both I/O and memory bus which supports 1 microsecond latency user-level DMA and one cycle latency for remote writes through automatic update. with doubling effect in the local memory. which combines writes in its internal FIFO as long as they are consecutive.

The simulator handles contention in detail both at the memory bus level as well as in the network interface between the incoming and outgoing traffic, but there is no contention simulated in the network itself.

For each application we studied, we computed four distinct ratios by dividing the source of overhead (communication, synchronization, protocol, stalling to memory) by the computation time.

4 Molecular dynamics

4.1 Problem and Algorithm

The problem is to simulate the evolution of a system of molecules over a period of time. Each step starts with a prediction for molecule motion and ends with a motion correction, both of which involve local computation on a molecule. In our analysis this predictor and corrector steps are lumped together as one *Pos*ition phase. The correction is based on the calculation of intra- and intermolecular (short-range) forces, which are computed in the *Fcal* phase, followed by the mostly local computation of kinetic energy in the *Kine* phase. The water-spatial program solves the intermolecular force calculation problem using a 3-d data structure of space cells to represent the cubical domain. Cells are sized to be at least as large as the cutoff radius used by the algorithm to determine which molecules a given molecule should compute intermolecular forces with. Each cell contains a linked list of molecules that are currently in that cell, and the cells are partitioned equally among processors so that a processor's cells are contiguous in the physical space. To compute the interactions, each processor which owns a cell looks only at its neighboring cells (see Figure ***). This makes the algorithm O(n), keeping the execution time reasonable even for large numbers of molecules. As a result of these interactions, molecules move in the 3-d space, potentially leaving and entering cells in each time step and requiring locking of the relevant cells to modify their lists. The movement of molecules among cells is slow but unpredictable, and is what gives rise to the irregularity of this problem.

4.2 Performance of Original Version

The performance of water-spatial is good under both software shared memory schemes, although better under AURC (see Table 5). Let us first observe the main results, and then explain them. For 16 processors, the contributions of overhead are negligible except for communication overhead in the all-software scheme. For 16 processors, approximately 90% of both computation and communication is spent in *Fcal* phase (6, 7). For 32 processors the contribution of the *Fcal* phase drops to 80% and the relative contribution of communication doubles. The **Kine** phase is responsible for most of the remaining execution time.

The good performance of water-spatial is due to a number of reasons.

METRIC	All SSM 16	All SSM 32	AU-based SSM 16	AU-based SSM 32
Comm/Comp	23 %	41 %	11 %	19 %
Synch/Comp	9 %	51 %	9 %	42 %
Stall/Comp	2 %	2 %	2 %	2 %
Protocol/Comp	4 %	6 %	2 %	3 %
Initial Speedups	**12**	**16**	**13**	**19**
Final Speedups	**12**	**17**	**13**	**21**

Figure 5: Performance of Water Spatial

1. low inherent communication to computation ratio
2. good temporal locality: The data for a given molecule belonging to a neighboring cell is reused by all molecules of a given cell.
3. fine-grained memory allocation: each molecule and array dimension is allocated separately, thus avoiding both granularity effects (false sharing and fragmentation) due to different molecules falling on the same page.
4. little movement of molecules across cells, so the modification of lists which could be expensive in SVM has little impact on the overall execution time

4.3 AURC versus LRC

Why does AURC perform better than LRC in this case? The reason is that its multiple-writer approach turns out to be cheaper. For extensive migratory data like the head pointers of the cell's molecule list, the aggregate size of the diffs in the LRC scheme can exceed the page size AURC. This means that when the granularity effects are reduced due to a fine grained allocation, the AURC scheme saves both in overhead and communication over LRC.

4.4 Improving Performance

An interesting performance effect is the synchronization overhead during the *Kine* phase. We found that this is caused by the contention incurred when all processors try to update a global kinetic energy vector once for each dimension, the last update being just before a barrier. While the critical section is computationally tiny, so the serialization effect is also very small in hardware shared memory, it is greatly exaggerated in SVM. Each acquire for the contended

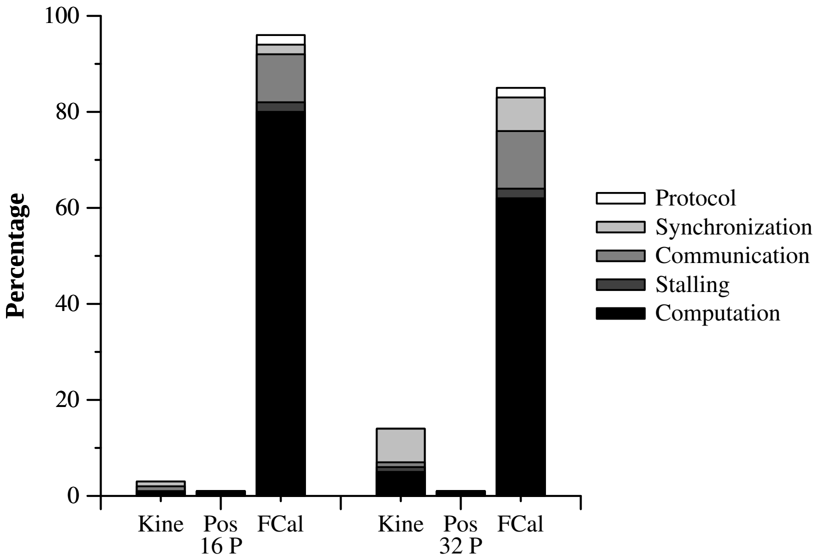

Figure 6: Phase Breakdown for Water-Spatial - AURC

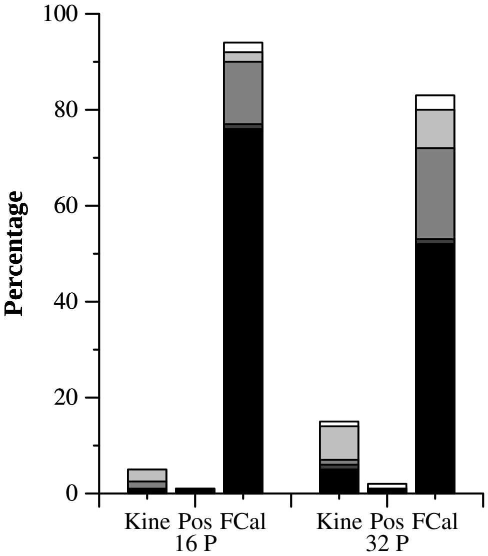

Figure 7: Phase Breakdown for Water-Spatial - LRC

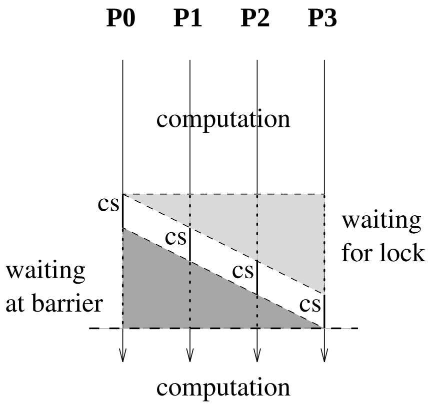

Figure 8: Serial Critical Section Effect

lock is expensive, and the update by each processor in the critical section causes a page miss while the lock is held, thus greatly expanding the duration of the critical section. The cascaded serialization hurts even more for the last spatial dimension because it is followed by a barrier: Even the first processor to get the lock immediately waits at the barrier for all the others to get there after incurring their long critical sections (see figure 8). Some processors wait to enter the critical section, others wait at the barrier, and the net performance loss is proportional to the (expanded) size of the critical section multiplied by the number of processors.

Can we improve performance? While we obtain good performance with 16 processors, it does not scale quite so well to 32 processors as the hardware CC-

NUMA scheme. There is not much we can do in the force calculation section without drastically changing the program, but we may be able to improve the critical sections. Using local accumulators we reduced out the number of lock accesses per processor from three to one, thus reducing the contention and, correspondingly, the synchronization overhead. This led to a improvement of speedups of both protocols for 32 processors, though the improvement was small since this did not improve the situation with the critical section just before the barrier. This lesson from this example is that even seemingly small critical sections that don't affect performance much for hardware CC-NUMA can hurt performance significantly under SVM when they are heavily contended. However, this is not the key bottleneck in overall performance. We can pronounce water-spatial satisfactory, and not worry about it further.

5 Hierarchical N-body problems

5.1 Problem and Algorithm

Barnes-Hut is also an N-body computation that computes the interactions among particles. However, it solves a gravitational problem, includes long-range interactions, and uses a hierarchical algorithm. Barnes-Hut simulates of the interaction of a system of bodies (galaxies or particles) in three dimensions over a number of time-steps. The hierarchical method it uses consists of approximating progressively larger groups of particles by single equivalent particles (the center of mass of the group) as they get progressively further away. A hierarchical subdivision of space is used to accomplish this, resulting in an unbalanced tree representation of space since the particle distribution is nonuniform. Since particles move as a result of the interactions, the tree is reconstructed every time step.

The parallel algorithm distributes particles among processors in a way that preserves physical locality in space. Each processor is responsible for computing the forces exerted on its assigned particles by the rest of the system. The three major phases in a time step are (i) the tree building phase *Tbd* in which processors collaborate to build the tree by loading their particles into it, synchronizing by locking on cells as necessary, (ii) the center of mass calculation phase *CoM* in which processors collaborate in an upward pass through the tree to compute centers of mass of cells from their children, synchronizing as necessary, and (iii) the force calculation phase *FCal* in which processors read some particle and cell information from other processors but write only their own particles. After force calculation the new positions for the particles are computed at the end of the time step.

5.2 Performance of the Original Program

AURC delivers reasonable speedups, though not nearly as good as hardware CC-NUMA particularly

METRIC	All SSM		AU-based SSM	
	16	32	16	32
Comm/Comp	21 %	45 %	23 %	23 %
Synch/Comp	47 %	88 %	9 %	33 %
Stall/Comp	2 %	2 %	2 %	1 %
Protocol/Comp	71 %	74 %	5 %	10 %
Speedups	**6.5**	**8.4**	**11**	**15**

Figure 9: Performance of Barnes - Hut

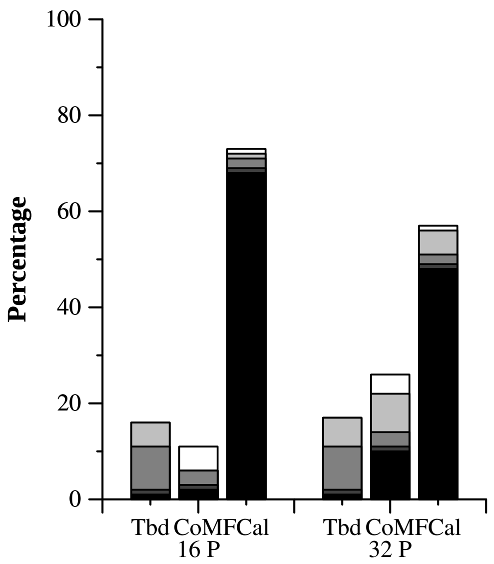

Figure 10: Breakdown for Barnes - AURC

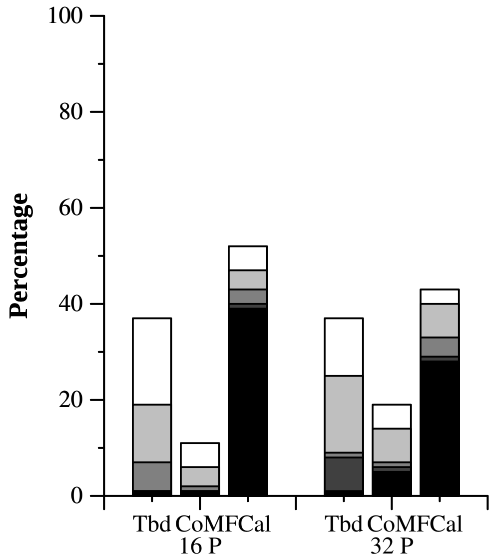

Figure 11: Phase Breakdown for Barnes - LRC

as the number of processors increases. The speedups for 16K particles are 10 for 16 processors and 15 for 32 processors, while LRC delivers poorer performance. Table 9 shows that the AURC produces more communication than LRC. However the overhead caused by the diff-based multiple writer scheme is even larger. In addition both protocols suffer from large synchronization overheads. Tables 10 and 11 show the contribution of each phase of the algorithm to the total elapsed time, for both protocols. Unlike in water-spatial, in barnes only 76 % of the elapsed time in AURC and less than 50 % in LRC is spent in the **Fcal** phase. However, this phase itself is sped up quite well. The tree building makes the big difference. Unlike in the hardware CC-NUMA case where it takes very little of the execution time, it becomes an expensive phase under software shared memory. This is because it involves a lot of very fine-grained communication (updates of other processors' cells to load particles in them, with the same cells and cells on the same page being written by the same processor). This fine-grained and false sharing is protected by synchronization on the cells, which causes the protocol to be invoked and data to be transferred. Thus, it produces large communication and protocol overhead for very little computation. Tree-building is more expensive in LRC (35 % of elapsed time for 16 processors) than in AURC (14 % for 16 processors).

On the positive side, Barnes-Hut also has a low inherent communication to computation ratio and high temporal locality (particularly in force calculation). The difference in performance compared with water-spatial are due to the following reasons:

1. somewhat larger inherent communication per computation ratio: the amount of computation for the interaction with a remote particle(cell) is smaller in barnes (gravitational interactions) than in water-spatial (molecular dynamics)

2. coarse grained allocation: it artificially increases the communication (in AURC) or overhead in update detection (in LRC) due to false sharing and fragmentation effects in particle accesses

3. frequent data structure building : at each step the octree is completely rebuilt

In Barnes the contribution of synchronization time is large mainly because of imbalances in communication time caused by the accesses to the tree in the tree-building phase: Cells toward the top of the tree are more frequently accessed and by more processors. cells at the bottom produce many more accesses. This effect produces higher contention for the locks corresponding to cells closer to the root of the tree. The effect is not seen too much in hardware CC-NUMA, but is clear here due to the much higher cost of communication and synchronization, and the expanded critical sections that worsen the contention. In addition, contention at certain home nodes also occurs because good data distribution is difficult at page granularity, and some pages end up being more contended than others.

5.3 AURC versus LRC

The reason the AURC perform better in this case is again due to its multiple-writer approach. Substantial fragmentation and false sharing, combined with intensive lock synchronization, forces LRC to incur large protocol overheads due to diff computation. It turns out that even the volume of data transferred is larger for LRC, which is not usually expected. This is because the intense lock synchronization creates many intervals and consequently, many more diffs are produced. They are sometimes located in more than one place, or they can sum up more than the size of the page. Also the frequent diff computation and communication, makes critical sections and the overall imbalance larger causing significant synchronization overhead to occur.

5.4 Improving Performance

In Barnes there is one large memory allocation for the entire array of particles. The particles are assigned to processors based on their position in space, not based on their positions in the array. Since Barnes has a lot of fine-grained lock synchronization and communication in the tree-building phase (one lock per body or group of bodies), false sharing is likely to occur. A brute-force solution is to allocate each particle separately. For many thousands of particles, this wastes a large portion of the local memory since each allocation is on a separate page most of which gets wasted. It

METRIC	All SSM		AU-based SSM	
	16	32	16	32
Comm/Comp	46 %	104 %	69 %	165 %
Synch/Comp	48 %	96 %	52 %	81 %
Stall/Comp	4 %	4 %	4 %	4 %
Protocol/Comp	29 %	31 %	2 %	1 %
Initial Speedups	**7**	**9**	**7**	**8.5**
Final Speedups	**9**	**11**	**7.8**	**9**

Figure 12: Performance of Volrend

also eliminates any prefetching effect which does occur with multiple particles falling on a page page.

What we want is a solution which can eliminate the false sharing but preserve the prefetching effect. This is possible, but it requires an even more relaxed memory consistency as we will see later.

6 Volume Rendering

6.1 Problem and Algorithm

The problem is to render three-dimensional volume data into an image using a ray casting method. The program renders several frames from changing viewpoints. In every frame, a ray is shot through each pixel into the volume. Rays do not reflect, but are sampled along their linear paths using interpolation to compute a color and opacity for the corresponding pixel. the frame is divided into small tiles of pixels, each processor is assigned some tiles, and dynamic task stealing is used for load balancing when a processor has finished shooting rays through the pixels in its assigned tiles. The accesses to the scene data into which rays are shot in this program are read-only and relatively uninteresting in software shared memory other than the fact that they have poor spatial locality. The interesting communication occurs in the task stealing using distributed task queues, and in updating pixels in the image plane, aspects with relatively little performance impact in the hardware-coherent case. Both type of access patterns causes a lot of false sharing at page level.

6.2 Performance of Original Version

Both protocols deliver satisfactory speedups (12) for 16 processors, but the speedups increase only slightly for 32 processors due to a lot more communication wait time (104% for LRC and 165 % for AURC).

The synchronization time is high due to the nature of the application as well as the high cost of synchronization in software shared memory. To keep the load well enough balanced, the tasks in the task queues are relatively small. Accesses to the task queues are protected by locks, which makes the ratio of synchronization per computation very high for software shared memory. A load imbalance due to imbalance in communication cost also increase the synchronization wait time.

An interesting question is where the large communication traffic itself comes from, since this program has very little communication volume in a hardware-coherent machine. It turns out that both task queue operations and image writing contributes to it. All task queue pointers are allocated within a global data structure, causing a potential source of false sharing. In addition, even the true sharing is affected by fragmentation due to the large granularity of communication. Image writing is more interesting. While there is no synchronization around the writes to the image, the false sharing of the image (of which there is a lot at page granularity) still causes a lot of communication even with the relaxed memory model and multiple writers protocols. This is because of the synchronization for the task queue again. Even though it is unrelated to the image updates and not needed for them, this frequent synchronization interspersed between the updates causes a lot of communication of image pages to happen (image pages get invalidated at the synchronization points due to false sharing, and then incur page misses when they are written next). This is not a problem in hardware cache-coherence since false sharing is less likely to occur in the same cash line.

6.3 AURC versus LRC

In Volrend the two protocols behave much as we expected. The communication is higher in AURC because of fragmentation, and because of the image false sharing descried above (AURC fetches an entire image page on each miss while LRC fetches only the small diffs). But this is partially compensated by the higher protocol overhead in LRC, caused by the same reason (large fragmentation, which increase the diff computation overhead). For 32 processors, the disadvantage of extra communication here is worse than the extra diff computation, thus LRC becomes slightly better than AURC (effect possibly caused by the large first level cache assumed in the simulations, which reduces the pollution effects caused by diff processing in LRC).

6.4 Improving Performance

How can we fix these performance problems? For the problem with task queues themselves, we allocate each task queue head with a separate memory allocation (malloc) and replace the task queue pointer array in the global data structure with an array of pointers to

METRIC	All SSM		AU-based SSM	
	16	32	16	32
Comm/Comp	4 %	18 %	16 %	19 %
Synch/Comp	121 %	752 %	136 %	620 %
Stall/Comp	16 %	16 %	15 %	14 %
Protocol/Comp	15 %	17 %	2 %	7 %
Initial Speedups	**7**	**3.7**	**6.6**	**4.4**
Final Speedups	**10**		**9.2**	

Figure 13: Performance of Raytrace

the task queue pointers themselves (so this array in the global data structure is not written). Since the number of the task queues is equal to the number of processors, wasted memory is not a problem. With this small optimization, the performance increased, more significantly for LRC (about 20%). For the second problem, we would like a way to disable the coherence for the image writing until the barrier which separates steps. In the next section we will show how this too can be achieved with a more relaxed memory consistency model.

Overall, volrend is an application with fine-grained sharing performs satisfactory under software shared memory, although there are some chances to write it much better.

7 Ray Tracing

7.1 Problem and Algorithm

This is also a rendering application which renders complex scenes in computer graphics using an optimized ray tracing method. Unlike in Volrend, in ray tracing rays reflect off the objects they strike. each contact generates multiple rays and the recursion results in a tree of per pixel. Due to the reflections, the amount of data accessed by a processor is much larger than in Volrend. Task queues with stealing are used similarly as in Volrend, with a similar decomposition into tiles as well.

7.2 Performance of Original Version

For 16 processors the performance is similar to that of Volrend, although the synchronization cost is much larger (13). Although Raytrace reads much more data from the scene, this data is read-only, so in software shared memory it can be completely cached in the local memory and thus it doesn't affect the performance. The big difference compared with Volrend, is that in Raytrace the computation per task is lower (reflection instead of interpolation). This produces many more small tasks which further cause large contention for task queues, which explains why synchronization to computation ratio is so large and becomes really huge for 32 processors, killing any acceptable performance.

7.3 AURC versus LRC

The performance of the two protocols is similar since most of the overhead is caused by synchronization events and not communication. For reasons similar with those explained for Volrend, namely very large fragmentation, LRC performs slightly better than AURC

7.4 Improving Performance

We implemented several possible improvements to the Raytrace application to try to improve performance at 32 processors. We tried the optimizations we have seen before (like using small memory allocations for task queues as in Volrend). We also undid two programming techniques that were used in the original program written for hardware cache coherence, but that turn out to be bad for SVM. One is padding data structures to align them to the coherence unit and avoid false sharing. While padding does avoid false sharing, it introduces a lot of either useless communication or protocol overhead due to fragmentation effects when there is true sharing. This results in either useless transfer of the pad data on the page (in AURC) or useless diffing of the pad data (in LRC). We therefore eliminated padding and replaced it with small memory allocations for the data structures where it seemed useful. The second problem was caused by a useless update performed in the critical sections used to manipulate queues when stealing tasks (this useless update was to a per-process variable used as a hint to a subsequent process trying to steal a task, to tell it whether there is expected to be something in the queue; but it is not necessary since the process can just read the head pointer of the queue to get this hint). This does not affect performance in the hardware-coherent case. It is fortunately well handled in LRC because updates are detected with a comparison, but in AURC such an update will actually cause a page transfer to occur inside the critical section, thus making the duration of the critical section a lot longer and causing serialization when the task queues are contended.

By fixing these problems we improved the performance by 30% for both LRC and AURC for 16 processors, primarily due to the first problem. However, this was still not enough to alleviate the synchronization wait time at 32 processors. On close inspection, we found that this wait time is due to a substantial imbalance not in computation but in the communication related behavior of different processors. It is caused by contention for some "hot" pages at some processor memories, and not in a simple way that we could easily remove. At the end, raytrace performs well for this data set for 16 processors but not for 32, though more effort could probably alleviate this contention. Running a larger data set may also help, but we are currently unable to do this due to memory limitations in our simulation environment.

8 Radiosity

8.1 Problem and Algorithm

Radiosity is a very complex computer graphics application which computes the equilibrium distribution of light in a scene using the iterative hierarchical diffuse radiosity method. A scene is initially modeled as a number of large input polygons. Light transport interactions are computed among these polygons and they are hierarchically subdivided to improve the accuracy. In each step the algorithm iterates over the current interaction list of patches, subdivides patches recursively and modifies interaction lists as necessary. The access patterns are extremely irregular and unpredictable.

Each processor uses multiple data structures (patches, interactions, edges, elements, task queues, etc.), allocated dynamically from a large pre-allocated pool of memory. Some of these allocations occur more frequently than others causing visible contention for the corresponding lock. The parallelism is managed dynamically as well using distributed task queues with task stealing. A lot of locking is used to protect updates to the data structures, and the sharing patterns are very fine-grained.

8.2 Performance of Original Version

Under both SVM protocols, the original radiosity code performs extremely poorly, mainly because of very large synchronization and communication due to false sharing and fragmentation. We obtained almost no speedup, and the overhead times overflowed our counters making it difficult for us to present reliable numbers for the execution time breakdown at this time.

8.3 Improving Performance

Out of all irregular applications we have studied, we found radiosity as the most fertile code for improvements and optimizations in respect to SVM. All the global pools for various data structures were allocated as arrays in one big global structure. Such allocation is the source of huge false sharing and fragmentation effects in the original code. Padding, when used, eliminate false sharing but increases fragmentation, resulting in little improvement. We therefore separated out the different types of data (tasks, elements, etc.) and even allocated each unit individually when possible. We also replaced the distributed allocation from a global pool with local allocation from local pools and use global allocation only when local pool is consumed, thus eliminating a part of the excessive synchronization for these global pools.

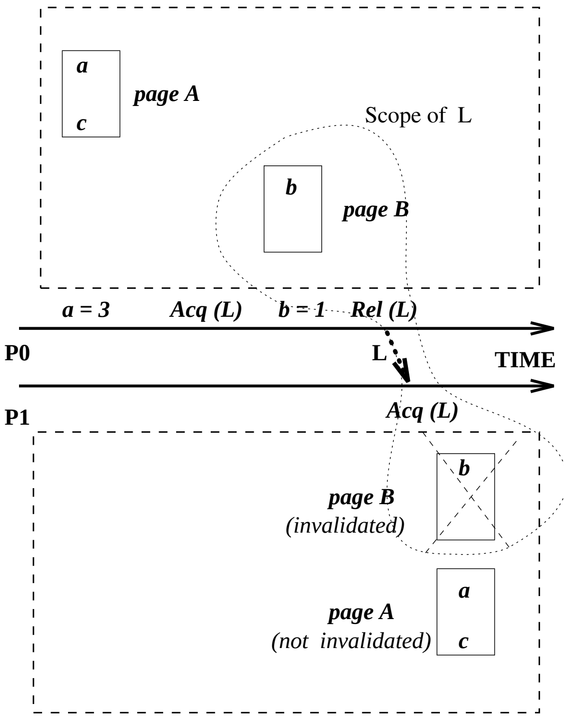

Figure 14: Scope Consistency: the intuition

As a consequence of these changes, we reduced the communication by a factor of 3. The speedups were almost double, but still very small (around 3 for 16 processors for both LRC and AURC). At the time of this submission we are still working to improve the code with other similar optimizations, but have to so far declare SVM a failure for this application.

9 Relaxing the Model: Scope Consistency

As we have discussed, data such as the particles/cells in Barnes-Hut and the image in Volrend and Raytrace are allocated in large arrays. The access patterns to these cause substantial false-sharing at page granularity (not nearly so much with cache lines), and these accesses are often either within critical sections or separated by lock accesses to unrelated data (such as the task queues in Volrend). In many cases, the alternative we have advocated of of using small, page-aligned allocation cannot be used because the individual objects are small and this would waste a lot of memory since each little malloc would be on a separate page.

Another way to reduce false sharing effects is to relax the consistency model beyond lazy release consistency. The well-known Entry Consistency (EC) [2] model accomplishes this by requiring the program to explicity bind data to synchronzation variables (e.g. locks), and at a synchronization event enforce coherence only for the data protected by that synchronization variable. This can reduce false sharing invalidations, but it has a high programming cost for making these explicit associations. Scope consistency [13] is new consistency model which achieves this effect by assuming an implicit association between data and synchronization such as locks, relying on the association naturally assumed by the programmer in most cases. The idea is that if a variable is updated by one processor in a critical section protected by some lock then it is fair to assume that it will not be read by another processor before that lock is acquired, since the program is intended to be race-free. Based on this intuition, in scope consistency locks are used to define scopes and the system guarantees to keep scope coherent rather then the entire memory (a barrier defines a global scope and makes all the local pages coherent by invalidation at each processor). This scheme doesn't require any annotations or explicit associations, but a programming precaution must be taken: We must ensure that critical sections are sized to cover all the updates which are intended to be visible to the processor that next acquires the lock. Details can be found in [13].

Figure 14 illustrates how scope consistency works through an example which actually corresponds to the sharing patterns encountered in Barnes or Raytrace. Lock L defines a consistency scope in which page B was modified by processor *P0*. Thus when processor *P1* acquires L from *P0*, it invalidates page B, but not page A as well (which would have happened under lazy release consistency, even though *P0* had updated variable a on page A before acquiring lock L. Under Scope Consistency, since the update of variable a at *P0* was not performed in the scope L, the propagation of its effect to *P1* after entering L is not guaranteed. (If we want the effect to be propagated, we must include the update of a in the critical section as well).

While Scope Consistency can provide similar benefits to Entry Consistency [2] and exploits the same intuition, it does not require explicit binding between data and synchronizations and hence does not change the programming interface. Scope Consistency can be supported on the same kind of hardware as AURC [13]. Our results for Barnes and Raytrace show that Scope Consistency greatly reduces the number of page faults caused by false sharing and provide an additional 11-12% performance improvement over AURC for both applicatios.

10 Guidelines

We learned a number of lessons while trying to improve the performance of the irregular applications for SVM protocols, which may translate to useful guidelines for programming for SVM systems rather than hardware-coherent cache-line based systems. We repeat and summarize these here.

1. **Fine-Grained Allocation** Many irregular applications have collections of logically small data structures allocated in large shared arrays; e.g particles in Barnes-Hut, rays, scene elements and task queues in Raytrace. These units are accessed in a spatially haphazard manner by processors, which leads to substantial fragmentation and false sharing. A simple way to alleviate these granularity effects in SVM systems is to allocate the small data structures separately (in separate small mallocs) whenever possible. This makes the units fall on separate pages, but also allows the system to communicate the data and process protocol overhead for the data at the granularity given by the allocation size rather than the whole page. In most programs, this optimization can be easily done for many of the important and frequently written shared data structures, and we found it to be possible in almost all the applications we studied. Task queues are obvious and important candidates: instead of keeping all the queue pointers for different queues contiguously in an array while accesses and modifications are performed under distinct locks, we allocate each queue pointer separately. This simple and effective technique led up to 20% performance improvement for our applications. The potential problem with the above approach is the wasted memory since each little malloc is aligned to a page.

2. **Fine-grained Allocation versus Padding** An alternative solution for false-sharing that is often used in hardware-coherent systems is padding, which also keeps unrelated data on distinct coherence units. However, padding is not so good in SVM. Padding to a page does reduce false sharing, but can cause a lot of useless data to be either communicated (in AURC, which communicates the whole page including padding on a fault) or compared in protocol processing (in LRC when computing diffs). Both padding and fine-grained allocation waste memory, but fine-grained allocation is preferable for shared virtual memory.

3. **Careful Synchronization** Synchronization is a much more significant issue for irregular programs under shared virtual memory than under hardware shared memory. Much of the synchronization in these programs happens through relatively fine-grained locking. This works well

enough for moderate-scale hardware cache coherent machines that there is little need to do anything different. However, synchronization is much more expensive in shared virtual memory systems. It invokes the software protocol and generates a lot of protocol overhead and subsequent communication. For locks, the subsequent page faults tend to greatly expand the duration of seemingly small critical sections, and hence substantially magnify serialization. While synchronization is critical since it is what causes consistency to be maintained in invalidation-based SVM protocols, for exactly the same reason it is very expensive too. Programs with lots of locks (Radiosity, Raytrace) often work well in hardware-coherent machines, but can perform poorly in SVM.

   There are many simple ways to reduce the frequency of synchronization and the contention it incurs. These techniques are often not needed in hardware-coherent machines, but are crucial in SVM. One technique is to use private local accumulators for partial per-process accumulation, and make locked updates to global variables as infrequently as possible (e.g. once per processor before a barrier, and not once per particle or ray). Another example is reducing/eliminating critical sections that are not needed for the correct execution (like those that protect global statistics variables, such as a global variable keeping the number of subrays processed in an earlier version of Raytrace). In general, algorithms that reduce locking and other synchronization even at the expense of some extra computation are likely to be very beneficial for SVM.

4. **Distribution of Pages to Home Memories** In AURC, a page's home serves as the collector of automatic updates for the page (all copies map to this page for automatic update), and it is to the home that page faults go to request the page on a miss. By default, homes are selected by the system, perhaps based on programmer hints. In LRC the home is not important, since every diff is maintained locally and a page fault gets diffs from previous updaters or consumers that are determined dynamically according to the program. Where the first allocated home is does not matter. If the distribution of pages among home memories is not good in AURC, then this can cause a lot of contention at a popular home, which also results in major imbalances among processors and high synchronization times (see Raytrace and Volrend). Because of the page granularity and the fact that all misses go to the home, the contention can be much greater than in hardware-coherent machines.

   For regular applications, it is often relatively easy to produce a good distribution of homes for pages. But for irregular applications with fine-grained data sharing in the virtual space this is often extremely difficult. The best we can do in many of these cases is hope that a round-robin distribution of pages works well, or adjust the distribution when contention is detected.

5. **Modifying Rather Than Rebuilding Distributed Data Structures** Many irregular programs use a distributed global data structure such as a tree, list, or queue. These structures are dynamic, and must be adjusted periodically (in each time-step, for example). The tree in Barnes-Hut is an example. Hardware-coherent programs often rebuild the data structure since this is not a major expensive phase of the program for moderate-scale machines. However, this rebuilding usually requires a lot of fine-grained communication and synchronization for very little computation, and therefore quickly becomes very expensive in SVM (as in Barnes-Hut). In most cases the rebuilding can be replaced by incremental modification of data structures. In hardware cache-coherent systems the overhead of writing incremental modification algorithms is often not worthwhile (e.g. for Barnes-Hut on moderate-scale machines), but it may be very helpful in SVM since it reduces the number of synchronizations (locks) used. We would like to investigate this issue further.

11 Conclusions

We found that many of the irregular applications perform reasonably well on 16-processor SVM systems, even as written for cache-line based hardware coherence, though the performance falls significantly short of that obtained on hardware-coherent machines. For several of the applications, however, the efficiency of both the aggressive SVM systems we examined drops considerably as the number of processors increases to 32, although these applications with the same problem sizes perform very well under hardware coherence. It would be useful to examine whether using somewhat larger problem sizes would close this gap, but we have so far been unable to do this in many cases due to simulation constraints (we will try to do this for the final version, if accepted). However, using larger problems is not expected to help as much or as quickly as it does for many regular applications. This is mainly because while increasing the problem size does reduce inherent communication to computation ratio (though somewhat slowly for these applications), it does not much help the spatial scattering of accesses to data common to irregular applications, and hence may not alleviate the fragmentation and false sharing problems so much.

However, our experience indicates that it is possible to substantially improve the performance of SVM by following some additional programming guidelines and techniques, several of which we have identified. More research will likely identify more such techniques, which may require more substantial changes to a program from the way it is written adequately for cacheline based hardware coherence (i.e. assuming small coherence units, and relatively cheap fine-grained communication and synchronization). Further research may establish SVM as a good platform for a wider range of irregular applications, together with a set of programming guidelines, and proceed further toward identifying the classes of emerging irregular applications for which this attractive, inexpensive platform is appropriate.

References

[1] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–176, Seattle, Washington, March 1990.

[2] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the IEEE COMPCON '93 Conference*, February 1993.

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

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

[5] S. Dwarkadas, P. Keleher, A. L. Cox, and W. Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proceedings of the 20th Annual Symposium on Computer Architecture*, pages 144–155, May 1993.

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

[7] L. Iftode, J. P. Singh, and Kai Li. Understanding the performance of shared virtual memory from an applications perspective. In *Proceedings of the 24th Annual Symposium on Computer Architecture*, May 1996.

[8] 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.

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

[10] L. I. Kontothanassis and M. L. Scott. Using memory-mapped network interfaces to improve the performance of distributed shared memory. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[11] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 302–313, April 1994.

[12] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[13] Jaswinger Pal Singh Liviu Iftode and Kai Li. Scope consistency: a bridge between release consistency and entry consistency. Technical Report TR-509-96, Princeton, NJ, January 1996.

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