Scaling Application Performance on a Cache-coherent Multiprocessors

Dongming Jiang and Jaswinder Pal Singh

Department of Computer Science
Princeton University
Princeton, NJ 08544

{dj, jps}@cs.princeton.edu

Abstract

Hardware-coherent, distributed shared address space systems are increasingly successful at moderate scale. However, it is unclear whether, or with how much difficulty, the performance of a load-store shared address space programming model scales to large processor counts on real applications. We examine this question using an aggressive case-study machine, the SGI Origin2000, up to 128 processors. We show for the first time that scalable performance can indeed be achieved in this programming model on a wide range of applications, including challenging kernels like FFT. However, this does not come easily, even for applications considered to be already highly optimized, and is very often not simply a matter of increasing problem size. Rather, substantial further application restructuring is often needed, which is usually quite algorithmic in nature. We examine how the restructurings compare with those needed for performance portability to shared virtual memory on clusters, and we comment on common programming guidelines for performance portability and scalability as well as on how the programming difficulty compares with that of explicit message passing. We also examine where applications spend their time on this large machine, the impact of special hardware features that the machine provides, and the impact of mapping to the network topology.

1 Introduction

Scalable, coherent shared address space (SAS) multiprocessing has been a major goal of research and development for many years. Over the last decade, many hardware cache-coherent, nonuniform memory access architectures (so-called hardware-DSM or CC-NUMA machines) have been built and shown to perform well at the moderate scale of about 32 processors [1, 13, 2, 17, 8, 14, 18, 19]. In fact, such machines are fast becoming the dominant form of tightly-coupled multiprocessor built by commercial vendors. An open question has been the scalability of application performance to larger processor counts. While a simulation study [5] has explored this question, indicating that several applications scale well and the problem sizes needed are surprisingly small, scalability has not yet been demonstrated or even explored on real systems, due in part to the lack of appropriate systems and applications. It is important to do this, however, since simulation has well-known problems in the ability to run large enough problems as well as in accuracy, especially in modeling contention which is increasingly critical on real systems at large scale.

This paper examines the performance of a wide range of SAS parallel applications on a 128-processor hardware cache-coherent machine (the SGI Origin2000), to take a snapshot of the state and potential of the load-store SAS programming model. While we study only one machine, we select a modern high-performance machine with an aggressive communication architecture and relatively few organizational artifacts that would limit our conclusions. Generalizability is left to the reader.

We drive our evaluation with eleven applications, eight from the SPLASH-2 [20] suite and three new ones. Our "original" versions of the applications are their best known forms for hardware cache-coherence so far. They use optimized partitioning techniques for load balance and interprocessor communication, are blocked for data reuse where relevant, use optimized data structures to increase spatial locality and reduce false sharing, and perform proper data placement across physical memories where needed. For the applications that were used in the earlier simulation study [5], they are the best versions used in that study, not the original ones used there.

Starting with these programs, we explore the following questions:

- Do the programs scale well on a 128-processor machine under problem-constrained (constant problem size) scaling? The initial problem sizes we use are the ones that either (a) delivered good speedups on a 256-processor machine in the simulation study (for the four applications that were also used there; 128-processor executions were not examined), or (b) delivered very good speedups on a 32-processor Origin2000 in a previous study [8]. Scaling well is defined as achieving 60% parallel efficiency (speedup divided by number of processors), which is a speedup of 76.8 on 128 processors. This number is somewhat arbitrary, but has been used in previous studies as well [5].
- If the basic problem sizes don't scale well, does increasing problem size to reasonable extents solve the scaling problem, and how large are the necessary problems for a 128-processor machine? Where do the programs spend their time on a machine of this scale, and what are the key bottlenecks?
- Are there application restructurings that are needed to substantially improve the scalability? How extensive, difficult or architecture-specific are these restruc-

turings, and to what extent do they approach programming with explicit message passing? How do they relate to the restructurings that were developed earlier to improve application performance dramatically on page-based software shared virtual memory (SVM) systems on moderate-scale clusters [6], but which were found to not affect performance very much on a moderate-scale Origin2000? Longer-term, we would like to develop a consistent set of algorithm- and program-structuring guidelines to achieve both performance portability and scalability across the range of emerging shared address space platforms: tightly-coupled and clusters.

- Do the special hardware features provided by the Origin2000 to enhance performance—namely prefetch instructions, dynamic page migration support, and at-memory fetch-and-op support for synchronization—help performance substantially?
- What is the impact of mapping processes to the network topology and of the machine having two processors share a memory and communication controller per node?

Overall, we show for the first time that scalable performance can indeed be achieved for a wide range of applications by using a load-store cache-coherent SAS programming model on a machine like the Origin2000. This is very positive news for the programming model, and the algorithmic and especially programming complexity are still a lot lower than needed for message passing in irregular programs. However, we find that the task is far from easy. Supposedly "optimized" SPLASH-2 and other applications that work very well in the 32-processor range most often do not scale. Increasing problem size to reasonable extents very often does not help enough either, and substantial application restructuring is needed. (Fortunately, after restructuring the scalability is good even for very reasonable problem sizes.) The restructurings needed are algorithmic and motivated by high-level issues, so they are not easy but they are not architecture-specific either. Interestingly, they are most often along the same directions as those needed for performance portability to SVM on moderate-scale clusters. The restructurings needed for scalability on Origin are sometimes less aggressive along these directions than those needed for performance portability to SVM, but the latter most often help scalability further as well. It therefore appears possible to construct common programming guidelines for performance portability and scalability. While we summarize some of our findings, developing good formal guidelines needs further research.

After Section 2 very briefly describes the Origin2000 platform and the new applications we use, one section of the paper is devoted to addressing each of the above questions. Our conclusions, including those for the rest of the above questions, are summarized in Section 8.

2 Platform, Applications and Metrics

The descriptions in this section are very brief. A fuller description of the architecture can be found in [12] and of the original application versions in [5, 20].

2.1 The SGI Origin2000

We use two SGI Origin2000 machines with different topologies. For our 32- and 64-processor results, we use a 64-processor machine with 16GB of main memory and routers

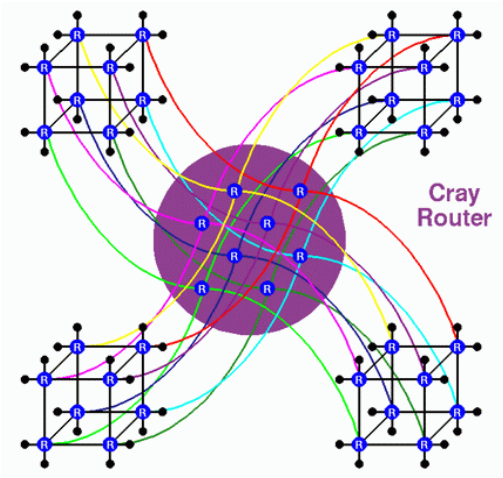

Figure 1: A 128-processor SGI Origin2000 connected by meta-routers. R inside each of the four hyper-cubes denotes a router. Each router connects two nodes to the network, and each node has two processors sharing a memory and coherence controller called Hub. Thus, four processors share a router. The links between corresponding nodes of the four hyper-cubes are not direct, as in a full hyper-cube, but rather pass through shared "meta-routers."

connected in a full hyper-cube topology. For the 96- and 128-processor results, we use a machine with four 32-processor hyper-cube modules connected by eight meta routers (see Figure 1,) and 32GB of main memory. Both machines have two 195MHz MIPS R10000 microprocessors within each node. The two processors in a node share a "Hub" memory and communication controller (which sees all cache misses and incoming transactions) and a non-coherent memory bus, and two nodes share a router. Each processor has separate 32KB first-level instruction and data caches, and a unified, 2-way set associative, 4MB second-level cache with a 128-byte block size. The page size is 16KB. To illustrate the appropriateness of this machine as an aggressive representative of its class, Table 1 shows the latency characteristics of some modern cache-coherent DSM machines.

2.2 Applications

In addition to eight SPLASH-2 applications [20], three new applications are used in this study: a Shear-Warp volume renderer [16, 11], a probabilistic inference application for belief networks (Infer, applied to a medical diagnosis problem [9]), and an application for protein structure determination in the presence of uncertainty (Protein) [3]. Results for the last two are available only up to 64 processors so far. Taken together, these applications exercise a wide range of characteristics in communication-to-computation ratio, communication pattern, load balance, synchronization, and spatial and temporal locality. We briefly describe only the three new applications in their original forms here. Better descriptions are available in the literature cited above.

Shear Warp is a faster algorithm for the volume rendering done by the SPLASH-2 Volrend. It has two main phases for each frame: a compositing phase to traverse the volume and compute a distorted intermediate image (which takes over 90% of the sequential time), followed by a warp phase to read the intermediate image, and write it, undistorted, into a final image. For the compositing phase, the original version partitions the intermediate image in an interleaved assignment of chunks of scanlines, with task stealing for subsequent load balancing; for the warp phase, the final image

Machines	Local (ns)	Remote Clean (ns)	Remote Dirty in 3rd node (ns)	Remote/Local Ratio (Clean)	Remote/Local Ratio (Dirty)
Origin2000	338	656	892	2:1	3:1
Convex Exemplar X	450	1315	1955	3:1	5:1
Data General NUMALiiNE	240	2400	3400	10:1	14:1
Hal S1	240	1065	1365	5:1	6:1
Sequent NUMAQ	240	2500	N/A	10:1	N/A

Table 1: Latencies and remote-to-local latency ratios on different systems. The latencies are from processor request to the response coming back to the processor.

is partitioned. This avoids write-write data sharing [7].

Infer, in a highly simplified description, takes a probabilistic belief network as input and first converts it to a tree of large nodes or "cliques". It then traverses the clique tree, first upward and then downward (with substantial computation in cliques and communication across them), to propagate inference and arrive at a diagnosis. There is parallelism both across and within cliques. The application exploits both by starting with a reasonable static assignment of cliques or their subsets to processors (starting from the leaves and keeping partitions coarse-grained and localized) and then stealing chunks of work from other cliques [9].

Protein is a new hierarchical algorithm for protein structure determination. The computational structure is again a tree, with the edges expressing cross-node dependences. Each node of the tree represents a substructure of the protein with a lot of parallelizable work. Nodes are initially assigned to groups of processors based on estimates of their relative workloads. For complex reasons, dynamic load balancing is implemented not by task stealing but rather by a new technique called *process regrouping*: having an idle processor group either take over a free node or join a currently working processor group [3].

2.3 Metrics

Several different problem sizes are studied for each application and machine size. We use parallel efficiency (speedup divided by number of processors) as our primary performance metric, measuring speedup relative to the same sequential program in all cases for an application. We also present per-processor breakdowns of execution time in as much detail as we can measure them with the available performance tools. In general, speedup or parallel efficiency can be inflated by the "superlinear" effect of cache capacity misses (or by limited local memory size on a single node). While this is a real effect, a metric can be designed to eliminate it if local and remote memory stall time can be separated (e.g. if there is little extra work in the parallel program, the sum of the busy plus local memory stall times on all processors can be used in place of sequential execution time). However, the machine does not allow this separation. Fortunately, with the large caches of the Origin, the cases where the capacity effects are dominant are few, and we shall point them out as appropriate.

3 Speedups for Basic Problem Sizes

We first choose a "basic" problem size for each application, as described in the introduction. Table 2 shows these problem sizes and their sequential execution times. Execution times in this paper are measured for many fewer time-steps or frames than the application would actually be run for, but enough to obtain representative behavior.

We run these basic problem sizes for several different processor counts on a dedicated machine, and show results for 32, 64, 96 and 128 processors. (Since using all processors on a machine can lead to lower speedups than leaving a couple out, e.g. due to OS intervention, we tested the speedups on 62, 126 and other numbers of processors to ensure that we were not having this problem.) Figure 2 shows that all the applications, except Raytrace, do not scale well at least beyond about 64 processors. To understand why, we use the performance tools the machine provides to investigate where the programs spend their time. In particular, we divide the per-processor execution time into three categories: busy time in computation (Busy), stall time waiting for cache misses to be satisfied (Memory, which we unfortunately cannot distinguish into stall time on local versus remote memory), and time spent at synchronization events (Synchronization). We use the best synchronization methods chosen from a recent detailed study of synchronization on this machine [10], though there is usually not much difference between methods for our problem sizes (see Section 6.3). An average execution time breakdown over 128 processors for all applications (except Infer and Protein) is shown in Figure 3. Per-processor breakdowns will be examined soon.

Application	Basic Problem Size	Sequential Time (ms)
Barnes	$16K$ bodies	7556556
Infer	CPCS-422 network	640000
FFT	2^{20} points	2631816
Ocean	1026×1026 grids	28488206
Protein	helix16	1713000
Radix	$4M$ keys	4554729
Raytrace	128×128 image (ball)	38186372
Shear-Warp	$256 \times 256 \times 256$ head	8905678
Volrend	$256 \times 256 \times 256$ head	934163
Water-Nsquared	4096 molecules	69031748
Water-Spatial	4096 molecules	7786852

Table 2: Applications, basic problem sizes, and sequential execution times. Four of the applications, Barnes, FFT, Ocean and Radix, were used in an earlier simulation study [5], so we choose their basic problem sizes as the ones that delivered good speedups on a 256-processor machine there (128-processor executions were not used). For the other applications, the basic problem sizes are those that deliver very good speedups on a 32-processor Origin2000 [8].

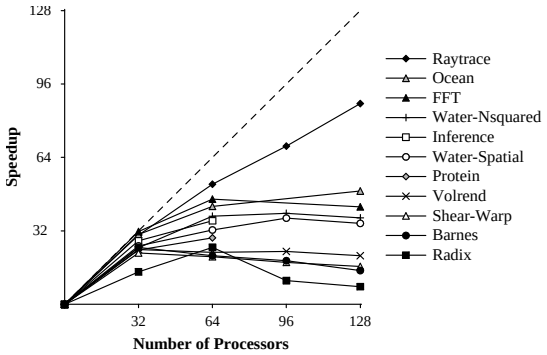

Figure 2: Application speedups for basic problem sizes.

Figure 3, together with uniprocessor breakdowns, shows that memory overhead usually dominates for most of the applications with the basic problem size. The exception is Water-Spatial, in which synchronization time is the dominant bottleneck. However, synchronization time is substantial in many other applications too. We find that most of the synchronization time is spent in barriers. Instrumenting locks and barriers shows that it is the idle or waiting time, especially due to imbalances in computation or data access costs, and not the overhead of the synchronization operations themselves, that almost always dominates (see [10] for more detail).

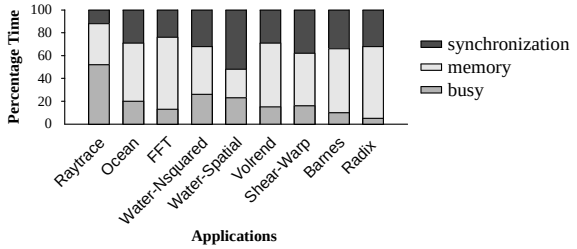

Figure 3: Average execution time breakdown for 128-processor executions with the basic problem sizes.

4 Effects of Increasing Problem Size

The above results raise the question of whether the basic problem sizes are large enough for a 128-processor machine. In this section, we explore the impact of problem size on application performance, increasing problem size to either the largest sizes we are able to run on the machine or the largest available for the application. These sizes are usually quite large compared to real usage of the applications.

The results are shown in Figure 4 for three different processor counts each (we ignore 96 processors to keep the graphs readable; the results are where expected between 64 and 128 processors). Increasing problem size keeps improving performance for many applications such as Ocean, Water-Spatial, Protein, Volrend, Shear-Warp, and Barnes, and on larger processor counts for FFT and Radix as well, but it starts hurting that of others at some point, such as Raytrace and Water-Nsquared. The surprising result is that increasing problem size to reasonable extents, even to the largest size we are able to run, does not by itself deliver the desired parallel efficiency beyond 64 processors for most of the supposedly highly-optimized applications we examined. Even larger problem sizes would likely cross the 60% mark in some cases, but these would be large problems for these applications. They would also exaggerate measurement problems for speedup due to capacity effects (which are already significant contributors to speedup for some applications at our problem sizes). Most importantly, we would like the machine to obtain good speedups on more reasonable problem sizes, which also do not rely on this capacity-related speedup exaggeration to reach the threshold.

Increasing problem size tends to improve many inherent program characteristics, such as load balance, inherent communication to computation ratio and spatial locality, but can increase working set size. By examining execution time breakdown on a per-processor basis, we gain insight into why increasing problem size helps some applications and hurts others. Where breakdowns show similar effects for multiple applications, we will show only one for space reasons.

4.1 Where Time is Spent on 128 Processors: Positive Impact of Problem Size

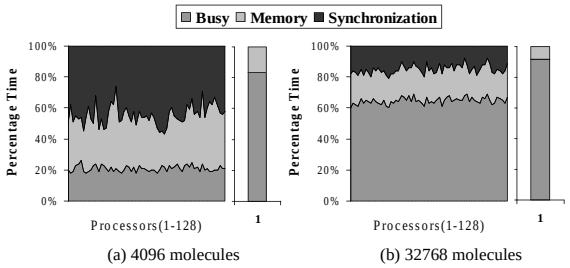

Figure 5: Execution time breakdown for Water-Spatial with both the basic and a large problem size for all 128 processors. The X axis represents the individual processors and the Y axis is the percentage of execution time. What would otherwise be 128 bars are merged together into a continuum to make the figure more easily readable. To the right of each breakdown figure, the breakdown for a uniprocessor execution is also shown for that problem size, to help indicate whether capacity effects are significant contributors to parallel speedup.

Water-Spatial finally achieves more than 60% parallel efficiency on 128 processors with 32K molecules (see Figure 4). The impact of problem size is smaller on fewer processors, as we expected. Figure 5 shows per-processor breakdowns of execution time for all 128 processors, arranged as a continuum rather than as separated bars. Analysis shows that larger problems improve performance in Water-Spatial mainly in two ways. First, the communication to computation ratio is improved due to the nearest-neighbor communication pattern among the cells in the three-dimensional space. Local memory behavior doesn't change much, as the important working sets are small (see [20] and the uniprocessor breakdowns in Figure 5), so the overall memory stall time is reduced. Second, and much more important, the reduction in communication also reduces the load imbalance in communication cost among processors, which is quite dramatic at the smaller problem size; synchronization time, most of which is waiting time, is therefore reduced dramatically as well. Computational load balance is improved too, but this is a much smaller effect. For large problems, Water-Spatial spends most of its time busy computing.

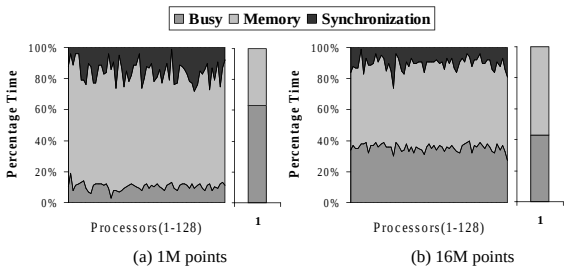

Figure 6: Execution time breakdown for FFT with both the basic and a large problem size for all 128 processors.

FFT almost reaches 60% efficiency at 128 processors by increasing problem size. However, large problems hurt parallel efficiency at smaller processor counts. The reason is as follows. Increasing problem size reduces communication to computation ratio, albeit slowly, and improves spatial locality on remote data (in particular, it reduces the transfer of unnecessary data [5]). However, for smaller machines, a large problem size also generates a lot of local cache capac-

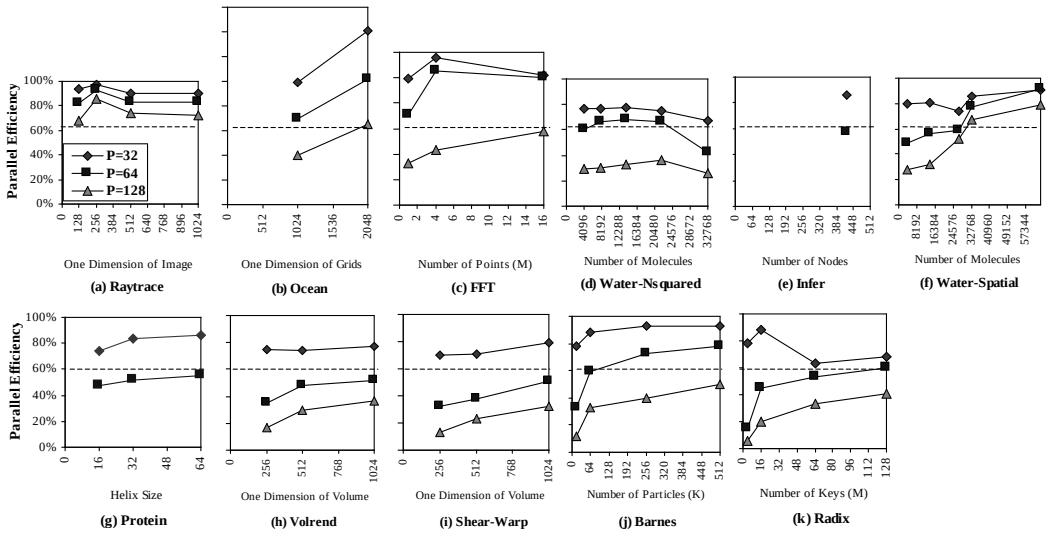

Figure 4: Impact of problem size on parallel efficiency of applications. The Y axis of each graph (one per application) is the parallel efficiency (speedup over best sequential execution time divided by number of processors) and the X axis is the problem size in units relevant to that application (see Table 2). A horizontal line is drawn at the "desirable" 60% level of parallel efficiency. Due to superlinear effect, Ocean and FFT achieve parallel efficiencies higher than 100% for some cases. There are three curves for each application, for three different numbers of processors. The 128-processor curve is generally the lowest one. For Infer, we have only one real input data set from medical diagnosis.

ity misses (see uniprocessor breakdowns in Figure 6), which apparently contend for the Hub and memory system with communication accesses and thus slow the machine down. Larger machines have fewer capacity misses, so only the improvements are seen. It may have been better for local capacity misses not to have to go through the Hub controller, unless memory or bus contention dominates (which we cannot determine). FFT speedup also benefits from superlinearity effects due to diminishing local capacity misses with more processors. It is not clear how much this contributes to the good speedup we observe since we cannot separate local from remote memory stall time, but we verify via calculations on problem and cache size that it is not a dominant effect. Ocean speedup, which also just achieves 60% parallel efficiency, benefits generally from superlinearity effects. In fact, the required problem size (2050-by-2050 grids) does not even fit in the local memory of a single node. Therefore, the sequential execution accesses remote memory for logically "local" capacity misses, which the parallel execution doesn't. Problem sizes small enough to avoid this superlinearity effect do not achieve 60% efficiency for Ocean.

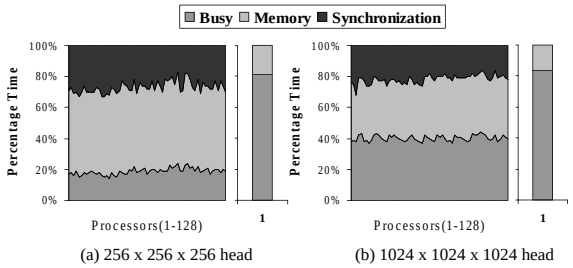

Figure 7: Execution time breakdown for Shear-Warp with both the basic and a large problem sizes for all 128 processors.

Barnes and Shear-Warp behave similarly to Water-Spatial, discussed earlier, except they don't achieve 60% efficiency for 128 processors. Both memory and synchronization overhead decrease relative to useful work as problem size increases. However, Figure 7 shows that memory time remains a bottleneck for the largest problem size we have, using Shear-Warp as an example (see Figure 10 later for Barnes-Hut). The working sets of these programs are known to still fit in the 4MB caches, as revealed by uniprocessor breakdowns too, and there is little false sharing; it is communication misses that don't decrease quickly and that stand in the way. Both programs exhibit a small amount of computational load imbalance, which is exaggerated by corresponding imbalances in their memory and communication accesses. Much of the time is spent stalled on remote misses, even for large problems.

Radix has been observed to be hurt by larger problem sizes in a simulation study [5], due to memory and controller contention between local accesses, remote accesses and especially protocol transactions like writebacks. However, with the large caches and efficient Hub controller on Origin we do not observe this behavior on 64 processors and beyond. At 64 processors, Radix (quite surprisingly) delivers greater than 60% efficiency for the largest problem size, a showcase for the machine's aggressive communication architecture. (Most other machines have behaved poorly for Radix, even at smaller processor counts, even though capacity-related superlinearity tends to help Radix as well.) However, at 128 processors even this problem size cannot overcome the contention caused by communication and protocol transactions, and parallel performance is poor.

4.2 Where Time is Spent on 128 Processors: Negative Impact of Problem Size

Increasing problem size can increase capacity misses, which can help speedup if they are mostly local (due to the capacity-induced superlinearity effect) and hurt speedup if they are mostly to remote data. Raytrace and Water-Nsquared are examples of the latter case. Raytrace has a large and somewhat diffuse working set of mostly remote data (see Figure 8). However, its speedup remains high enough. For Water-Nsquared (not shown), we find that al-

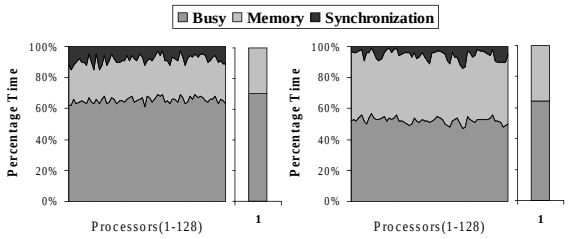

Figure 8: Execution time breakdown for Raytrace with both a small and a large problem size for all 128 processors.

though it performs $O(n^2)$ computation on $O(n)$ data (that are distributed properly), the memory stall time percentage increases with problem size once a threshold is crossed. Although the fraction of time spent in synchronization decreases dramatically, speedup diminishes. We will examine this problem and its time breakdown in more detail in Section 5.

To summarize, while increasing problem size improves parallel speedups for several applications, doing this to reasonable extents (as discussed earlier) enables us to achieve 60% parallel efficiency for only two of them (Ocean and Water-Spatial). This is despite the fact that cache capacity can yield a superlinearity effect for some applications with large local working sets (a significant factor in Ocean). For most of the applications, to achieve good scalability our only recourse is to restructure them even though they all deliver very good speedups at the 32-processor scale (well over 60% parallel efficiency) and they are supposed to be already optimized in all areas (like partitioning, data structure design, and data placement).

5 Impact of Application Restructuring

Execution time breakdowns, augmented with knowledge of the applications, allow us to understand the causes of poor scaling as well. For example, for Barnes-Hut, Shear-Warp and Water-Nsquared the problem is mostly data access time (in these cases, communication). Using the processor hardware counters together with microbenchmark latency results allows us to determine whether the problem is due to the frequency of misses or due to contention. And profiling the programs further (using *pixie* and *prof*) helps us determine in which routines the memory problem lies. Based on this information we try to restructure the applications that did not achieve 60% efficiency so far.

Figure 9 superimposes parallel efficiency versus problem size curves for the restructured applications on those for the original version. The improvements are large at the large processor counts. We describe the restructurings briefly to understand their effects here.

5.1 Restructuring Applications on the SGI Origin2000

Consider **Barnes-Hut**. Profiling shows that the memory bottleneck is largely in the phase of building the shared tree. In the original parallel tree-building algorithm, processes load their assigned particles one by one into a globally shared tree, locking tree cells when they need to be modified. Since the tree is partitioned in quite contiguous pieces among processors, the communication and contention for the locks is expected to be small. However, even for large problem sizes, the tree-building phase takes a very large fraction of the execution time on 128 processors (31% for 512K particles) compared to only about 2% of the time on a uniprocessor. To improve performance, it is necessary to change the way trees are built in parallel, particularly to reduce communication. An alternative tree-building algorithm, called MergeTree, operates by having each processor construct a local tree out of only its own particles without any communication (the same particles that were assigned to it for force calculation).

and finally merging the trees. The recursive merging algorithm is complicated, and we shall not describe it here. The merging is 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 have do successively more work, including more communication and locking. This leads to greater imbalance at the barrier than in the original algorithm, though there is less time spent in locks. There is also significant extra computation. Nonetheless, the reduction in communication outweighs the loss of load balance consistently for all problem sizes and processor counts, and the restructured version achieves 58% parallel efficiency on 128 processors with 512K particles, compared to the original 50% (see Figure 10(a) and (b)). Somewhat larger problem sizes would cross the 60% mark.

In **Shear-Warp** volume rendering, the algorithmic restructuring is more difficult to describe in a short space. The problem in the original version is the loss of locality in the interface between the compositing and the warp phases. Fixing this loss of locality without causing load imbalance requires developing a fundamentally new parallel algorithm based on profiling for load balancing [7]. The new algorithm, originally developed for a hardware DSM machine, is organized so that the process that writes a (now contiguous rather than interleaved) partition of the intermediate image in the compositing phase is in fact the one that reads that same partition in the warp phase [7], and it writes a warped piece of the final image accordingly. Memory stall time diminishes greatly and, since there is little compromise of load balance, speedup increases substantially for all problem sizes and processor counts. Large problems achieve over 60% parallel efficiency.

In the more familiar **Water-Nsquared**, the water molecules are allocated contiguously in an array of n molecules, and partitioned among processors into contiguous pieces of n/p molecules each. In the force calculation phase, a processor accesses its own n/p molecules and the following $\frac{n}{2} - \frac{n}{p}$ molecules in the array that are assigned to other processors (and hence are remote). The natural way to structure the loop nest is to have the outermost loop for a process iterate over the particles assigned to it. For each local particle, an inner loop iterates over the next $\frac{n}{2}$ particles that it interacts with. This is what the SPLASH-2 program does. Unfortunately, if n is large, the $\frac{n}{2}$ remote molecules may not fit in the cache, so when a remote molecule is accessed again to interact with the next local molecule it will no longer be in the cache. Capacity misses are incurred on remote data, and a lot of artifactual communication is generated. The solution is essentially to interchange the loops. A remote molecule is accessed once, interactions are computed between it and all local molecules (reusing it $O(n/p)$ times in the cache), and only then is the next remote molecule accessed. Temporal locality is high on remote molecules and is low on local molecules, which are fewer and also cheaper to miss on (see Figure 10(d) and (e)). The loop is very complex and has many subroutine calls within it, so this is not

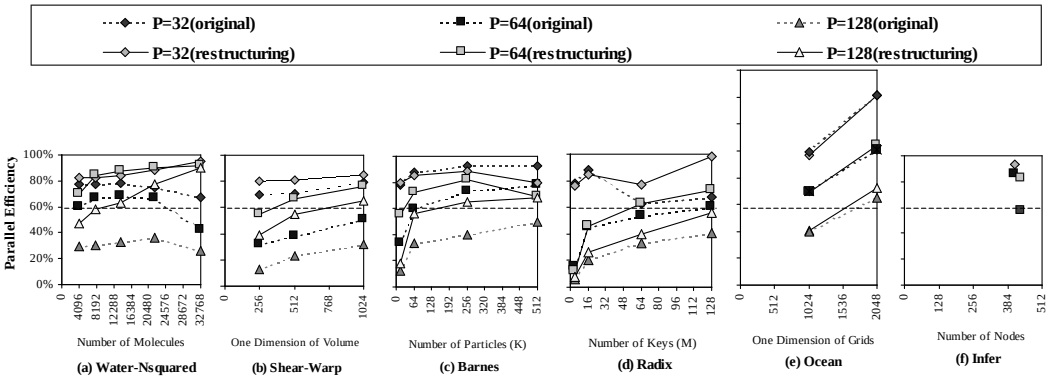

Figure 9: Impact of application restructuring on parallel efficiency. The dotted lines represent the original versions (copied from Figure 4), and the bold lines represent the restructured versions.

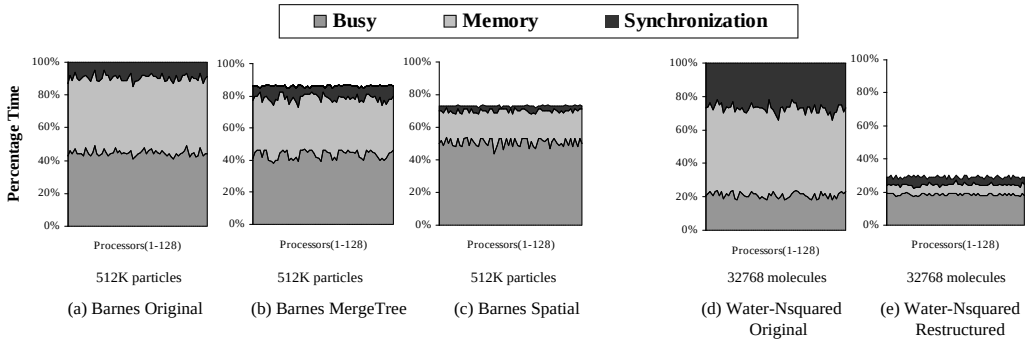

Figure 10: Execution time breakdowns for original and restructured application versions of Barnes-Hut and Water-Nsquared on 128 processors. The total execution time (height of the bars) of the restructured versions is normalized to that of the original version. This shows how much the restructuring improves performance, and where most of the performance benefits come from.

the kind of loop restructuring that a compiler could readily do. Also, it is motivated by the knowledge of which capacity misses are remote and hence more expensive, which a compiler may not have. This restructuring is very important for large problems. Increasing problem size hurts performance for the original version, and it is not even close to 60% parallel efficiency for any problem size on 128 processors. The restructured version, however, starts achieving 60% efficiency for the relatively small problem size of 8K molecules on 128 processors, and continues to ascend for even larger problem sizes.

In the **Infer** application, the restructuring needed is to move away from the dynamic partitioning approach that was very successful at 32-processor scale, and to a completely different, static partitioning algorithm that exploits parallelism only within each clique (tree node). The algorithm uses knowledge of the dependences between elements in parent and children nodes to maximize locality in crossing over from one node to the other [9]. The coarser level of parallelism across nodes is sacrificed, but communication and locality are much more important for realistic belief networks at larger scale and the restructured version achieves an efficiency much higher than 60% on 64 processors.

For **Radix** sorting, which does surprisingly well at moderate scale, the key problem at large scale is the permutation phase. In this phase, the pattern of writes from a process to the destination array is temporally scattered, but ultimately ends up in small contiguous chunks. One possible solution, therefore, is to first write the keys to a set of small contiguous local buffers and then transfer them (contiguously) to the corresponding chunks of the shared output array. However, while this reduces the scattering of remote writes, it does not reduce burstiness of write-based communication and all the resulting protocol traffic. The local copying outweighs any savings in contention, and in fact performs quite a bit worse. To overcome the problem, we use a different parallel sorting algorithm as our restructured version. **Sample sort** uses two local sorting phases, separated by a short phase to compute splitter keys and a communication phase to copy a contiguous set of remote keys to a local array (to prepare for the next local sort). The local sorts can use any sequential sorting method; here, Radix sort is used. Unlike parallel Radix sort, the all-to-all communication is based on stride-one remote reads rather than scattered remote writes, and is therefore better behaved. The disadvantage is that the local sorting is done twice, so (ignoring memory data access) the parallel efficiency is limited to 50%. Here, Sample sort achieves an efficiency of 50% with 128M keys, some of which is due to capacity effects as in Radix sort.

Finally, we also tried to improve the scaling of FFT and Ocean, to rely less on capacity effects for achieving 60% parallel efficiency. For FFT, we tried performing the transpose implicitly while computing rowwise FFTs, to reduce burstiness of communication compared to an explicit transpose phase, but this did not help. For Ocean, we tried using rowwise rather than tiled partitioning to reduce fragmentation when fetching remote words [6]. This changes performance only slightly, with the direction depending on problem and machine size. Overall, while we are still unable to reach 60% efficiency on 128 processors for two applications (Vol-

rend and Radix) with our problem sizes, we are finally able to do so for the rest.

5.2 Restructurings for Performance Portability and Scalability

So far, we have examined application restructurings to the extent needed to obtain 60% parallel efficiency on a 128-processor machine. A set of restructurings was also developed for several applications, starting from the same original "optimized" versions as here, to achieve performance portability to shared virtual memory (SVM) systems on small- to moderate-scale (16-processor) clusters of workstations or SMPs [6]. SVM systems on clusters have very different protocols and performance characteristics for communication and synchronization than hardware-coherent systems, as well as very different granularities of coherence and communication (a page). In that study, those restructurings were found to neither help nor hurt performance very much on a 16-32 processor Origin2000, and were therefore performance-portable.

In almost all cases, we find that those restructurings are either similar to or more substantial than the ones used here for hardware DSM scalability. Where they are more substantial, we examine here whether those restructurings significantly further enhance scalability on a hardware-coherent machine like the Origin2000 than what we have seen so far, or whether they degrade it or are neutral. We also examine whether the restructurings that suffice here perform well on moderate-scale SVM clusters, and hence whether similar restructurings lend themselves both to performance portability and to scalability on hardware DSM. (It is worth noting that while the restructurings for performance portability improve SVM performance dramatically, the resulting parallel performance is not nearly so good as that of an Origin2000 at similar scale, and that scalability on SVM clusters has not been demonstrated. However, performance portability to deliver even acceptable performance on clusters is nonetheless important, since users want to write programs once and run them on both high-end machines and clusters.)

Consider **Barnes-Hut.** The tree-building phase was the major bottleneck on the SVM system as well. Using the merging-based tree-building algorithm improves SVM speedup substantially, but from 2.76 to only 5.65 on 16 processors. This is because there is still significant communication, because the higher cost of communication than on Origin increases the load imbalance in the execution, and especially because a significant amount of locking remains when merging trees recursively. Unlike on Origin, locks are expensive and cause substantial serialization on SVM systems, since they are where software protocol activity is incurred and (often large) software messages are exchanged. Lock time, rather than communication time as on Origin, was the major bottleneck for SVM. Therefore, the successful restructuring for SVM was to develop a new tree building algorithm that eliminates locking.

In the new tree-building algorithm, called Spatial, bodies are assigned *differently* to processes than in the force calculation and other phases. The space is divided up into many pieces (which match subtrees), and the pieces are assigned to processors. The shared "supertree" of these subspaces (i.e. the very small tree that has these subspaces as its leaves) is built by one processor. Processors individually build the subtrees corresponding to their assigned subsets of spaces (without locking), and then simply attached their subtrees to the unique corresponding leaves of the supertree (again without locking due to the uniqueness of the place of attachment). The elimination of locking dramatically improves performance, even though load balance in the tree building is compromised and some locality is lost between the tree building and other phases. This restructured version ended up achieving a speedup of 10.5 on SVM with 16K particles on 16 processors in [6]. Since it greatly reduces communication as well in the tree building, it addresses the main problem on the Origin too, for the same reason as it eliminates locking: separating out partitions cleanly into large, structured subtrees and eliminating write-sharing of data. Interestingly, the loss of load balance outweighs the reduction in communication on smaller Origin systems, so this restructured version is actually a little worse than the original version on 32 processors. But for larger systems, the communication reduction wins and this more greatly restructured version outperforms even the merging-based version to achieve 69% parallel efficiency on the 128-processor Origin (compare Figures 10(a), (b) and (c)).

Shear-Warp is another example where the same restructuring dramatically improves both performance portability and hardware scalability. Like on the Origin, communication due to loss of locality between the compositing and warp phases causes performance problems on SVM. Moreover, contention caused by the expensive communication and synchronization increases load imbalance as well, which is not easily alleviated by task stealing in SVM (see Volrend later). By attacking these problems, the same restructuring improves performance on SVM from the original speedup of 3.41 to 9.21 on 16 processors.

There are applications for which further restructuring is critical for performance portability to SVM, but these restructurings do not help Origin scalability much. These include Ocean and Raytrace, where the performance is already very good on the large-scale Origin, and Volrend, where the problem size is not large enough but restructuring doesn't help much. In **Ocean**, the large communication granularity of a page requires that rowwise partitioning be used to minimize the unnecessary communication (fragmentation) that occurs at column-oriented boundaries, even though this compromises inherent communication to computation ratio. This increases SVM speedup from 8.5 to 13.2 for a 1026-by-1026 Ocean simulation. However, as we have seen, its effect on Origin is very small and depends on problem and machine size. In **Raytrace**, removing an unnecessary statistics lock used for each ray improves speedup from 0.5 to 11.7, whereas it improves parallel performance by only about 4% on Origin where synchronization is much more efficient. In **Volrend**, the SVM restructuring is to substantially change the initial partitioning of tasks to processors so that task stealing (which is very expensive and much less effective in SVM due to high locking cost) is greatly reduced. This improves parallel performance by about 68% on SVM. Task stealing is very effective on Origin, so the better balanced initial assignment doesn't buy more than a few percent in performance. The real problem for Volrend scaling on Origin is that we do not have large enough problem sizes .

Finally, sometimes we find restructurings that are necessary on the Origin2000 but are not relevant to SVM clusters. These have to do with the fact that remote data are replicated only in the hardware cache on a machine like the Origin, while they are replicated in main memory in SVM. Water-Nsquared is the example here. The original loop order is not a problem on SVM systems, since once a remote molecule is accessed for the first time it is replicated in lo-

cal main memory, and subsequent cache misses to it are no more expensive than misses on locally assigned molecules.

In summary, the restructurings needed for scalability on hardware coherent systems and for performance portability to SVM on clusters are often similar, despite the great difference in the performance and granularity characteristics as well as in the protocols and consistency models. They are not very architecture-specific, but address high-level bottlenecks and are algorithmic. Sometimes the manifestations of the problem are different (e.g. communication time on Origin versus synchronization time on SVM in the original tree-building algorithm in Barnes-Hut), but the fundamental reason and hence the approach to restructuring needed is the same. Where the restructurings needed are different, we find that most often they take the form of needing to push harder along similar guidelines for performance portability than for scaling, rather than requiring very different approaches or goals. Thus, while some restructurings are often critical to SVM but do not matter much for hardware-coherent scaling (e.g. dramatically reducing fine-grained synchronization as in Volrend and Raytrace and being much more sensitive to system granularities as in Ocean), even these are indeed performance portable back to moderate- or large-scale hardware-coherent systems. This is a positive observation, since it indicates that generalizable programming guidelines may (and should) be developed for both scalability and performance portability across a range of platforms that support the coherent shared address space programming model.

5.3 Programming Guidelines

While much more work is needed to develop programming guidelines for scalability and/or performance portability that can be formalized and presented to programmers, let us summarize some of the early guidelines we have observed in the course of this research. The guidelines often have to be pursued further for performance portability to even small- to moderate-scale SVM than what suffices for scaling on an Origin-like hardware-DSM system.

- Partition as statically or with as much control over locality as possible (while preserving load balance), even if higher levels of available parallelism must be sacrificed. Very dynamic approaches used for load balancing often don't scale. Examples are Infer and Shear-Warp.
- Related to the above, while it is difficult to generalize, on small- to moderate-scale hardware-coherent systems load balance is often the biggest problem and should not be compromised. However, at larger scale or on commodity clusters, communication frequently becomes a greater bottleneck, often due to the contention it causes. A good example is tree building in Barnes-Hut: The original strategy with better load balance is better at moderate scale on Origin, but the new strategies are much better at large scale or on SVM.
- Separate out partitions as much as possible. Fine-grained read-write sharing of data is okay at moderate scale (32 or so processors), but at larger scale it seems necessary to move to parallel algorithms that separate out the computation and data accessed by different processors within a phase of computation into large, well-structured partitions. An example is the tree-building in Barnes-Hut, where instead of loading particles individually into a global tree or even merging irregular subtrees, the processors build spatially cleanly divided subtrees independently without interaction or shared access, and then simply place them into the global tree. This approach is conceptually closer to what one is usually forced to do in message passing programming, so the algorithmic gap in partitioning and parallel algorithm design appears to shrink with scale or commodity approaches. However, we find that the programming and orchestration is still much easier in SAS for all the familiar reasons [15], and that for many very irregular applications (like our graphics ones) substantial algorithmic advantages also remain for obtaining good performance.
- Structure parallel algorithms and partitioning schemes so that they are single-writer, i.e. so that only one processor writes a given data item (or cache block or page, depending on the granularity of coherence). Multiple writers lead to both communication, which is expensive on both types of systems, and synchronization which is very expensive in SVM.
- Beware loss of locality across computational phases, as in Shear-Warp. Compromising some load balance or even communication within a phase to preserve this is often useful.

Other guidelines that are better known include (i) exploiting temporal locality on remote data rather than local in hardware-coherent CC-NUMA systems if there is a choice, as in Water-Nsquared, (ii) perhaps violating or compromising inherent properties to achieve better interactions with large system granularities as in Ocean under SVM, (iii) reducing the need for task stealing when synchronization is very expensive as in SVM, (iv) structuring and distributing data properly, etc.

6 Effectiveness of Special Hardware Features

Let us now return to the Origin2000 itself. In this section, we discuss the effectiveness of three specific hardware support features provided by the machine to enhance performance: software controlled prefetch instructions, hardware/software support for dynamic page migration, and an efficient at-memory fetch&op primitive.

6.1 Prefetching Remote Data

We examined prefetching only for remote data by inserting prefetch directives to the compiler in the communication loops. We find it to help FFT a little (less than 5% in execution time at most) on 32 processors, but substantially on larger machines especially for larger problem sizes: up to 20% on 64 processors and up to 35% on 128 processors. Prefetching helps Sample Sort too, by about 20% in execution time on 128 processors for the larger problem sizes. There is more communication with more processors, so more latency to be hidden. As for larger problem sizes, they don't reduce communication to computation ratio much in these applications, and they are less dominated by other overheads like synchronization wait time, so the impact of the prefetching benefits are larger. Prefetching does increase network traffic, but communication bandwidth is adequate even for all-to-all matrix transpose communication on the 128-processor machine . If data are not placed properly, so traffic is greatly increased in FFT, it was shown

Application	Problem Size	Manual	Round Robin	Round Robin + Page Migration
FFT	2^{24} points	55	26	25
Radix	$128M$ keys	38	24	25
Ocean	2050×2050 grids	64	34	33

Table 3: Comparison of performance (speedup) with different data distribution strategies on 64 processors.

at smaller scale in [8] that the sharing of the Hub coherence/communication controllers by two processors and of routers by two Hubs can become a bottleneck to prefetching improvement.

Prefetching remote data does not help much in our other applications. In our irregular applications, this is because of the difficulty of predicting cache misses and thus scheduling prefetches early enough. In Ocean, which is regular and predictable, it is not efficient due to the difficulty of scheduling prefetches early enough for remote data. In Radix, it is successful only in the less important phase of accumulating histograms using a prefix tree. Prefetching local data as well may be more generally helpful, but we do not examine it here.

6.2 Dynamic Page Migration

Proper data distribution can be very important for applications that have large capacity miss rates, especially for regular applications like FFT, Radix and Ocean. Origin2000 incorporates hardware support in its protocol that supports dynamic page migration, which together with appropriate OS migration policies can help make capacity misses local without placing the burden on the programmer [12]. We compared the performance of three page placement strategies for FFT, Radix and Ocean with large problem sizes: manual, round robin across nodes, and initially round robin with page migration enabled. From Table 3, we see that the gap between appropriate manual placement and the others is large for these problem sizes, while turning on dynamic page migration does not seem to improve performance. We have tried different thresholds for the migration policy, and it is unclear whether this is due to migration costs or poor policy. We do not presently have the tools to examine this.

6.3 Fetch&op for Synchronization

The at-memory, uncached fetch&op primitive is designed to improve synchronization performance [12]. We use it to implement different well-known barrier and lock algorithms, and compare it with using the processor-provided load-linked and store-conditional (LL-SC) instructions. We find that neither sophisticated barrier (or lock) algorithms nor the fetch&op primitive help performance significantly, compared to a simple ticket lock and tournament barrier implemented with LL-SC, which are what we have used in the results above. As mentioned earlier, for the problem sizes used in this study the load imbalance and wait time at synchronization dominates the cost of the synchronization operation itself. A more detailed study of synchronization methods using microbenchmarks and real applications on this machine [10] finds similar results for smaller problem sizes as well, and more generally finds using the fetch&op support for implementing locks and barriers not to be especially valuable compared to well-known algorithms that use LL-SC.

7 Effects of Interconnect Topology

Finally, in this section, we examine the last of the questions asked in the introduction: the impact of mapping processes appropriately to the network topology and the impact of the machine having two processors sharing a memory and communication controller per node. A full set of results is available, but we only mention some examples due to space limitations. Recall that the topologies described in Section 2 for our machines connect routers, not nodes or processors directly.

7.1 Mapping to Network Topology

Mapping to a network topology has largely been intentionally ignored in modern performance guidelines for parallel programming (for example [4]). However, its impact has not been evaluated on scalable cache-coherent systems, in which messages are finer-grained and more frequent than in explicit message passing. We focus the discussion on three representative applications with different communication characteristics, namely an irregular application (Barnes-Hut), a regular application with nearest-neighbor communication (Ocean), and a regular application with all-to-all communication (FFT).

Barnes-Hut: It is usually difficult to map optimally for an irregular application. We compare a random mapping of processes to the network topology and a linear mapping—process i goes to processor i. Linear mapping performs consistently better than random mapping for all problem sizes and processor counts (increasing speedup from 8.5 to 14.7 for 16K particles and from 59 to 63 for 512K particles on 128 processors). If we ensure that a node is always assigned a pair of neighbor processes in the linear process ordering and then map these pairs to nodes in the topology, the differences between linear and random mapping of nodes are similar. The other irregular applications support these results. Linear mapping also tends to help more for smaller problems on larger machines, when inherent communication is larger.

Ocean: For applications with nearest-neighbor communication such as Ocean, the most important aspect is to ensure that the two processes mapped to a node are assigned neighboring partitions of a grid. Mapping to the network topology beyond this is not very beneficial up to 64 processor systems that have a full hypercube topology. On larger systems that use metarouters, mapping matters more: Even for a large 2050-by-2050 grid, an appropriate near-neighbor mapping of process-pairs to nodes in the network topology is about 20% better than a random mapping and 10% better than a linear mapping on 128 processors. It is not clear how much of this is due to the use of metarouters and how much simply to the larger scale. Interestingly, if data are not distributed appropriately across main memories, generating a lot of artificial communication, mapping seems to matter a lot even with full hypercubes and at smaller scale: Some randomly generated mappings of process pairs to nodes perform substantially worse than an appropriate near-neighbor mapping (50% worse for a 2050-by-2050 grid in Ocean on 64 processors, and 80% worse on 128 processors), while others are almost as good. Similar effects are seen for a simple SOR solver, indicating that mapping matters for near-neighbor programs, especially at larger scale.

FFT: Applications with all-to-all communication tell a different story. We use the version that prefetches remote data in these cases. The matrix transpose in FFT is staggered so that process i starts to transpose data from process $i+1$, to avoid generating a hot-spot. Surprisingly, we found that a random mapping of processes to processors performs better than a linear mapping (about 10% better even for 16M points on 128 processors). Experiments with several types of mappings show that what's most important is that we not start with a situation in which one processor in a node starts transposing from the other processor in the same node while the other starts transposing from a processor on a remote node. Thus, if the program is left as it is, it is important that neighboring partitions of the matrix *not* be mapped to the same node; and if they are mapped this way (e.g. linearly), then the transpose loop in the program should be changed so that both processes on a node start transposing from off-node or randomly chosen processes. The two solutions perform equivalently well, and beyond this the mapping or transpose ordering doesn't seem to matter. In fact, profiling shows that it is the first of the $P-1$ (P is number of processors used) outer most loop iteration of transpose communication (i.e. the first other process that a process communicates with) in which the difference between the good and bad cases differ the most by far (the good or "random" cases take half the time of the bad cases in this portion). The difference is much smaller in later iterations. Of course, at some point the "bad" communication situation (one on-node, one off-node) will be encountered algorithmically, but by this time the two processors are out of synch enough that the impact is much smaller (if it is the first iteration then the full impact is felt since they start at the same time).

Overall, the most important aspect of mapping on this machine seems to be related to which processes are mapped to the same node. Once this is taken care of, mapping to the topology beyond that does not appear to matter much if data distribution is done properly. The (important) exception is near-neighbor grid computations at large scale, though it is unclear whether this has to do with metarouters or scale. Interestingly, 64-processor experiments with and without metarouters show that metarouters help the performance of FFT on large systems, by reducing the impact of contention due to the increase in latency they cause.

7.2 Impact of Two Processors Per Node

Finally, we assume a good mapping across nodes and examine the impact of the Origin's decision to have two processors share a memory and communication system per node. We do this by comparing a regular execution with one that uses only one processor per node (leaving the other idle), which reduces contention at the hub and on the memory bus but, of course, uses more nodes and hence potentially more network hops. Note that the bus is not coherent, so processors on the same node do not benefit from cache-to-cache sharing, only from more of the main memory being local. The loss in cache-to-cache sharing is offset by the latency and bandwidth gains of not having to cross a coherent bus at both ends of a remote transaction.

We examine applications with high communication and high capacity misses, such as FFT, Ocean, Sample Sort, Radix, and Raytrace. When problem sizes are relatively small and communication dominates, we found that the performance difference between using one processor per node and using two is small. Using two processors per node does not even matter for near-neighbor communication such as in Ocean. However, when problem sizes are large and capacity related contention at the Hub and memory emerges, the applications consistently perform better using only one processor per node. This impact is especially significant for smaller processor counts. For Sample Sort on 32 processors, for instance, using one processor per node (on 32 nodes) performs 40% better than using two processor per node (on 16 nodes) for large problem size—128M keys. In general, contention with local capacity misses seems to be a more substantial issue than communication-related contention at the Hub and memory system on this machine. It may have been alleviated by not having the Hub have to process local capacity misses. Overall, having two processors per node does not seem to be very beneficial from purely performance and programming ease viewpoints (witness the FFT transpose for the latter).

8 Discussion and Conclusions

We have investigated the scaling of application performance in a load-store cache-coherent shared address space, using a 128-processor SGI Origin2000 machine as an aggressive representative of hardware DSM systems. We find that the problem sizes predicted by an earlier simulation study [5] of a similar type of architecture, or even larger problem sizes that perform well on many 32-processor real systems, do not scale. Making problem sizes larger, at least to reasonable extents for these machines, achieves scalability for some applications (and then at very large problem sizes), but surprisingly not so for several. Rather, substantial further algorithm or application restructuring has to be used than is already present in these supposedly highly optimized applications. With this restructuring, we have demonstrated scalable performance to 128 processors on a wide range of applications in this programming model for the first time.

With the restructured applications, the problem sizes needed to achieve good performance are reasonable (and are much smaller than with the original versions, even in the cases where problem size alone can solve the scaling problem). However, they are still a lot larger than found in the simulation study. In fact, comparing the results of these studies for the four common applications strengthens the view that system-level simulation is valuable for qualitative estimation (and perhaps quantitative evaluation of specific tradeoffs) but less so for quantitative results about overall performance of machines.

While our study has focused on the Origin2000, the restructurings are algorithmic and target high-level tradeoffs, so we expect them to be generally valuable on most hardware-coherent machines as well. In fact, we find that the restructuring needed is most often along the same lines to that used earlier to improve performance on moderate-scale shared virtual memory systems on clusters [6], even if it sometimes ends up addressing different specific bottlenecks. The restructuring needed for scaling on hardware DSM is often less extreme, but the more extreme restructuring needed for SVM often improves the scaling further. This is interesting because it increases both the potential and the need for developing generalizable programming guidelines for the coherent shared-address-space programming model, to obtain both scalable and performance-portable execution across the range of key emerging platforms for parallel computing. We have summarized some of our early observations about such guidelines, but much more research is needed in this area

for both tightly coupled systems and clusters.

As for the Origin itself, it's communication architecture does surprisingly well even on challenging applications like FFT and sorting. When it fails, it is usually due to an overload of contention, or due to imbalances in communication cost across processes. The contention includes that between communication operations themselves or especially between local misses within a node and incoming remote operations, at either main memory or the controller. When traffic is high in these cases, the decision to share a Hub between two processors and a router between two Hubs can hinder performance. In general, having two processors share a node without a coherent bus does not seem to help application performance much. Prefetching remote data is found to help in FFT and Sample Sort at large scale but not much in other cases; using the at-memory fetch&op support for synchronization does not help application performance noticeably [10]; and we could not make dynamic page migration work successfully. Beyond node sharing issues, mapping processes to the network topology appears to have relatively small impact, despite the low-overhead communication and fine-grained messages, except for near-neighbor communication on large systems.

For our study, perhaps the greatest missing feature of the machine is the lack of tools to look more deeply into the machine's execution and memory system, e.g. to distinguish different types of cache misses (including simply local versus remote), to find out where pages of data are allocated, and to obtain information about the time taken by cache misses or about contention which is often the cause of poor memory system performance. This still limits our analysis in many cases—for example to distinguish whether speedup is due to beneficial aggregate capacity effects or goodness of the communication architecture in handling communication misses—and also makes it difficult to know whether efforts to place pages properly have in fact succeeded (we were deceived in this in our experiments for FFT in an earlier study of a smaller-scale machine [8]). The performance counters that count simple event frequencies were not nearly so valuable for our purposes as these features would have been.

Overall, we conclude that performance scalable to over 100 processors can indeed be achieved using load-store cache-coherent shared memory on a machine like the Origin2000 on a wide range of applications, and that the algorithmic and programming complexity are still a lot lower than needed for message passing. This demonstrated successful performance validates the architectural style and is very promising for the programming model. However, the programming process for achieving scalable performance is often far from easy even on this aggressive machine. A better understanding of programming issues and guidelines for scalability and for performance portability across tightly coupled machines and clusters in both major programming models is an important area for future work.

Acknowledgements

We thank SGI/Cray and NCSA, especially Larry Smarr and his colleagues, for access to Origin2000 systems. We thank Dan Lenoski, Jim Laudon, Rohit Chandra, Marco Zagha, and John McCalpin for valuable discussions. Cheng Chen, Alexander Kozlov, and Sanjeev Kumar provided us with some application and synchronization codes.

References

[1] G. A. Abandah and E. S. Davidson. Effects of architectural and technological advances on the HP/Convex Exemplar's memory and communication performance. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

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

[3] C. Chen, J. P. Singh, and R. Altman. Parallel hierarchical protein structure determination in the presence of uncertainty. In *SIAM Conference on Parallel Processing*, 1999.

[4] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[5] C. Holt, J. P. Singh, and J. Hennessy. 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.

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

[7] D. Jiang and J. P. Singh. 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.

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

[9] A. Kozlov and J. P. Singh. Parallel probabilistic inference on cache-coherent multiprocessors. *IEEE Computer*, 1996.

[10] S. Kumar, D. Jiang, R. Chandra, and J. P. Singh. Evaluating synchronization on shared address space multiprocessors: Methodology and performance. In *Proceedings of ACM Sigmetrics'99*, May 1999.

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

[12] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[13] D. Lenoski, J. Laudon, J. Truman, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems*, 4:41–61, 1993.

[14] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[15] J. P. Singh, A. Gupta, and J. L. Hennessy. Implications of hierarchical N-body techniques for multiprocessor architecture. *ACM Transactions on Computer Systems*, May 1995.

[16] J. P. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *Computer*, 27:45–55, 1994.

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

[18] R. Thekkath, A. P. Singh, J. P. Singh, J. L. Hennessy, and S. John. An evaluation of the Convex Exemplar SP-1200. In *Proc. Intl. Parallel Processing Symposium*, April 1997.

[19] H. J. Wasserman, O. M. Lubeck, Y. Luo, and F. Bassetti. Performance evaluation of the SGI Origin2000: A memory-centric characterization of LANL ASCI applications. In *Supercomputing '97*, Nov 1997.

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