Parallel Tree Building on a Range of Shared Address Space Multiprocessors: Algorithms and Application Performance

Hongzhang Shan and Jaswinder Pal Singh
Department of Computer Science
Princeton University
{shz, jps}@cs.princeton.edu

Abstract

Irregular, particle-based applications that use trees, for example hierarchical N-body applications, are important consumers of multiprocessor cycles, and are argued to benefit greatly in programming ease from a coherent shared address space programming model. As more and more supercomputing platforms that can support different programming models become available to users, from tightly-coupled hardware-coherent machines to clusters of workstations or SMPs, to truly deliver on its ease of programming advantages to application users it is important that the shared address space model not only perform and scale well in the tightly-coupled case but also port well in performance across the range of platforms (as the message passing model can). For tree-based N-body applications, this is currently not true: While the actual computation of interactions ports well, the parallel tree building phase can become a severe bottleneck on coherent shared address space platforms, in particular on platforms with less aggressive, commodity-oriented communication architectures (even though it takes less than 3 percent of the time in most sequential executions). We therefore investigate the performance of five parallel tree building methods in the context of a complete galaxy simulation on four very different platforms that support this programming model: an SGI Origin2000 (an aggressive hardware cache-coherent machine with physically distributed memory), an SGI Challenge bus-based shared memory multiprocessor, an Intel Paragon running a shared virtual memory protocol in software at page granularity, and a Wisconsin Typhoon-zero in which the granularity of coherence can be varied using hardware support but the protocol runs in software (in the last case using both a page-based and a fine-grained protocol). We find that the algorithms used successfully and widely distributed so far for the first two platforms cause overall application performance to be very poor on the latter two commodity-oriented platforms. An alternative algorithm briefly considered earlier for hardware coherent systems but then ignored in that context helps to some extent but not enough. Nor does an algorithm that incrementally updates the tree every time-step rather than rebuilding it. The best algorithm by far is a new one we propose that uses a separate spatial partitioning of the domain for the tree building phase—which is different than the partitioning used in the major force calculation and other phases—and eliminates locking at a significant cost in locality and load balance. By changing the tree building algorithm, we achieve improvements in overall application performance of more than factors of 4-40 on commodity-based systems, even on only 16 processors. This allows commodity shared memory platforms to perform well for hierarchical N-body applications for the first time, and more importantly achieves performance portability since it also performs very well on hardware-coherent systems.

1 Introduction

As hierarchical techniques are applied to more and more problem domains, and applications in these domains model more complete and irregular phenomena, building irregular trees from leaf entries efficiently in parallel becomes more important. N-body problems are among the most important applications of tree-based simulation methods today, and we use them as the driving domain in this paper. The performance of N-body applications has been well-studied on two kinds of platforms: message passing machines [4, 2] and tightly coupled hardware cache-coherent multiprocessors [9]. Due to their irregular and dynamically changing nature, a coherent shared address space programming model has been argued to have substantial ease of programming advantages for them, and to also deliver very good performance when cache coherence is supported efficiently in hardware.

Recently, clusters of workstations or multiprocessors have become widely and cheaply available to high-end application users as well. Since they are capable of delivering high computational performance, they are important and viable platforms for next-generation supercomputing. As a result, the coherent shared address space model (like message passing) has been supported on a much greater variety of systems with both different granularities as well as varying levels of hardware support in the communication architecture. The focus is on using more commodity-oriented communication architectures, either by relaxing the integration and specialization of the communication controller [11] or by leveraging the virtual memory mechanism to produce coherence at page granularity (shared virtual memory) [5], or by providing access control in software [6, 7], in almost all cases running its protocol in software. For the shared address space model to truly deliver on its ease of programming advantages for complex, irregular applications such as tree-based N-body applications to end users—for whom it may be important that their codes run well across a range of available platforms—it is very important that performance not only be good on hardware cache-coherent systems but also port well across these important platforms. Although message passing may have ease of programming disadvantages, it ports quite well in performance across all these systems. This

performance portability advantage may overcome the ease of programming advantages of the coherent shared address space model if the latter cannot deliver good performance on commodity-based systems, so users in domains like tree-based N-body applications may prefer to use the more difficult model. The question of performance portability of a shared address space has begun to be studied [8], but there is a lot more work to be done. By studying performance across platforms in some detail for hierarchical N-body applications, we were led to study and develop new tree building algorithms; these algorithms and the resulting performance of the N-body applications across the range of systems are the subject of this paper.

Having specified the initial positions and velocities of the n bodies, the classical N-body problem is to find their positions after a number of time steps. In the last decade, several O(NlogN) algorithms have been proposed. The Barnes-Hut method [1] is the one widely used on sequential and parallel machines today; while the tree building issues and algorithms we discuss apply to all the methods, we use a 3-dimensional Barnes-Hut galaxy simulation as an example application.

The sequential Barnes-Hut method has three phases in each time step of a simulation. In the first tree-building phase, an octree is built to represent the distribution of all the bodies. This is implemented by recursively partitioning the space into eight subspaces until the number of particles in the subspace is below a threshold value. The lowest level cells contain bodies, and higher level cells contain the summary effect of the bodies in their rooted subtree. The root cell represents the whole computational space . The second phase computes the force interactions. In this phase, each body traverses the tree starting from the root. If the distance between the body and a visited cell is large enough, the whole subtree rooted at it will be approximated by that cell; otherwise, the body will visit all the children of the cell, computing their effect individually and recursively in this way. In the third phase each body updates its position and velocity according to the computed forces.

By studying a range of such systems (that support a coherent shared address space programming model with varying degrees of efficiency in hardware or software) for hierarchical N-body applications, we find that while the sequentially dominant ($> 97\%$) force calculation phase parallelizes and scales well on all platforms, the seemingly unimportant tree building phase performs poorly and very quickly becomes the major bottleneck on commodity based communication architectures, often taking up more than half the execution time even on 16-32 processor system and even resulting in application slowdowns compared to a uniprocessor, and not being easily alleviated by running larger problems. The parallel tree building algorithms for which these results are observed were developed for hardware-coherent systems, and deliver very good application performance on those at least at moderate scale. Programming for portably good performance across shared address space systems thus requires that different tree building algorithms be developed and used. The goal of this paper is to examine alternative tree building algorithms on the range of coherent shared address space systems, and substantially improve the performance of the tree building phase and hence the entire application. As a result of this research, a new and very different tree building algorithm is designed that dramatically improves performance on page-based shared virtual memory (SVM) systems, while still being competitive on efficient hardware-coherent and other systems, and thus provides performance portability at least at the scale examined.

The platforms we study range from fine grained hardware coherent machines to fine-grained software coherent machines to page-coherent SVM systems, and from centralized shared memory systems to distributed ones. They include the SGI Challenge (centralized shared memory, hardware cache-coherent) [12], SGI Origin 2000 (distributed shared memory, hardware cache-coherent) [12], Intel Paragon (page grained SVM)[10] and Typhoon-zero (hardware support for variable coherence-granularity but protocols running in software on a coprocessor)[11] (in the last case, we use a page-based SVM protocol as well as a fine-grained sequentially consistent protocol). We use five different algorithms to build the tree, and keep the other two phases of a time step the same.

The next section describes the five tree building algorithms. Section 3 introduces the four platforms. The experimental results are analyzed in section 4, related work is discussed in section 5, and section 6 summarizes our conclusions.

2 Parallel Tree Building Algorithms

The five methods we use differ from each other in data structures and/or algorithms. We call them as ORIG, ORIG-LOCAL, UPDATE, PARTREE, SPACE. The first two algorithms, ORIG and ORIG-LOCAL, come from the Stanford Parallel Application Suites. They correspond to the versions in the SPLASH and SPLASH-2 programs, respectively. The PARTREE and UPDATE algorithms have been developed previously in the context of efficient hardware cache-coherent systems, but in previous work, a small modification to the original algorithm was found to work just as well in that context. We examine these algorithms on modern hardware -coherent machines and study their impact on commodity-based platforms as well. The SPACE algorithm is a new algorithm we developed motivated by an understanding of performance bottlenecks on the commodity platforms. Here we only discuss the tree building phase; the force calculation and update phases are the same in all cases and are discussed in [9].

2.1 ORIG

In this and the next three algorithms, in the tree building phase each processor is responsible for only those particles which were assigned to it for force calculation (and update) in its previous time step. (For the first time step, the particles are evenly assigned to processors). The global octree is built by having processors load their particles concurrently into a single shared tree, using locks to synchronize as necessary First, the dimensions of the root cell of the tree are determined from the current positions of the particles. Each processor then adds its own particles one by one to the root cell. Whenever the number of particles in a cell exceeds a fixed number k, the cell is subdivided into 8 sub-cells, each of them representing one sub-space, and the particle recursivelly inserted in the appropriate sub-cell. The tree is therefore adaptive in that it will have more levels in the higher density regions. When a particle is actually inserted or a cell actually subdivided, a lock is required to guarantee that only one processor is modifying the cell at a time.

All the cells used are allocated from a single contiguous shared array (the shared array of global cells in Figure 1), which is itself allocated at the beginning of the program. A processor obtains the next index in the array dynamically when it needs to “create” a new cell in the tree. Each processor has a separate array of cell pointers to record which cells are assigned to it (local arrays of cell pointers in Figure 1). The arrays of bodies and body pointers are represented similarly. When bodies or cells are reassigned across time-steps, only the value of the pointers in the local arrays change.

After the tree has been built, the center of mass of the cells are computed in parallel from the bottom of the tree to the top. Each processor is responsible for computing the center of mass of the cells it "created".

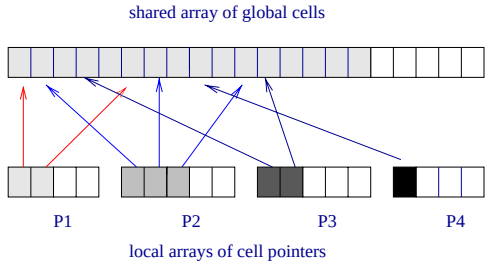

Figure 1. The array of global cells and local arrays of cell pointers used in ORIG Algorithm. The shaded cells are actually used while the blank are not.

Figure 3 shows a 2-dimensional problem, in which a quad-tree is built instead of an octree using the body distribution shown on the left part. The 3-dimensional case is very similar to this (ignore the boxes and dashed lines in the tree for now, they will be useful when describing the UPDATE algorithm).

2.2 ORIG-LOCAL

The ORIG-LOCAL version of the application is an optimized version of the ORIG program. The main differences from the ORIG version are the data structures used to represent the cells. First, ORIG-LOCAL distinguishes internal cells in the tree from leaf cells and uses different data structures to represent them. Particles can be contained only in the leaves. Second, each processor allocates and manages its own cell and leaf arrays, instead of allocating them in a single global array and local pointer arrays (See Figure 2). By keeping its assigned cells (shaded) contiguous in the address space, this allows a processor to more easily allocate its cells in its local memory and reduce false sharing among cells, which is particularly important at larger coherence granularities. If bodies are reassigned across time-steps, they are moved from one processors's array (declared in a shared arena) to another's. Each processor also maintains a private data structure which includes some frequently accessed variables such as the number of cells used and the number of leaves used. In the ORIG version, these variables are allocated together in shared arrays, increasing the potential for false sharing. In the performance section, we will find that these data structure changes have a great effect on the execution time of N-body applications.

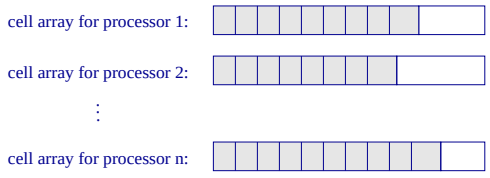

Figure 2. The local cell arrays used in ORIG-LOCAL Algorithm. The shaded cells are actually used while the blank cells are not

2.3 UPDATE

Experience with the application shows that since the particle distributions in N-body computation tend to evolve slowly with time, the number of particles that move out of the cell in which they were placed in the previous time step is small. In the UPDATE algorithm, instead of rebuilding the tree each time step we incremently update the tree every time step as necessary. For every particle we start from the cell it was in in the previous time step, and move it only if it has crossed its cell boundary. Since the size of the whole space changes in each time step, to do this, we need more variables in the cell and leaf data structures to record the space bounds they represented in the previous time step. However, the relative positions in the tree structure that cells represent remain the same across time-steps. If the particle must be moved out of its current leaf cell, we compare it with its parent recursively until a cell in which it should belong in this time step has been found. Then we move it out of the original leaf cell and insert it in the subtree represented by the newly found cell, locking and even creating new cells as necessary.

Considering Figure 3 as an example, the body in the shaded box moves to the box above it in the next time step. When updating the tree, it will be first moved out of the leaf containing it in this time step, and checked whether it will stay in the cell labeled by number 9 in the tree (which represents the box with the thick border in the figure on the left). Since this is not true, it is checked with the root, and inserted from there. The dashed line in the tree shows the path by which this particle moves. As a result, the only particle in the source leaf cell disappears, so the leaf is reclaimed, and the particle will be added to its new cell.

2.4 PARTREE

In the previous three algorithms, particles are loaded directly into a single shared tree. As a result, whenever a cell is actually modified (e.g. particles or new children added) a lock is needed to implement the necessary mutual exclusion. This causes a lot of synchronization, contention and remote access, which are usually expensive in these systems. Especially for large numbers of processors, a large amount of contention will be caused at the first few levels of the tree, when processors are waiting at locks to insert their own particles. To reduce these overheads, another algorithm which we call PARTREE can be used. The following is the code skeleton to build the global tree:

```
MakeGlobalTree()
{
        cell   *Local_Tree;

        Local_ Tree = InitRoot();
        InsertParticlesInTree(Local_Particles, Local_Tree);
        MergeLocalTrees (Local_Tree, Global_Tree);
        BARRIER;
}
```

In this algorithm, The function InsertParticlesInTree is responsible for building a local tree (Local_Tree) for each processor using only those particles (Local_Particles) that are currently assigned to it (from the previous time step). Figure 4 shows the local trees built for each processor assuming the particle distribution shown in Figure 3. The building of the local trees does not require any communication or synchronization. After the local trees have been built, they are merged into a global tree by MergeLocalTrees function. The dimensions of the local trees are precomputed to be

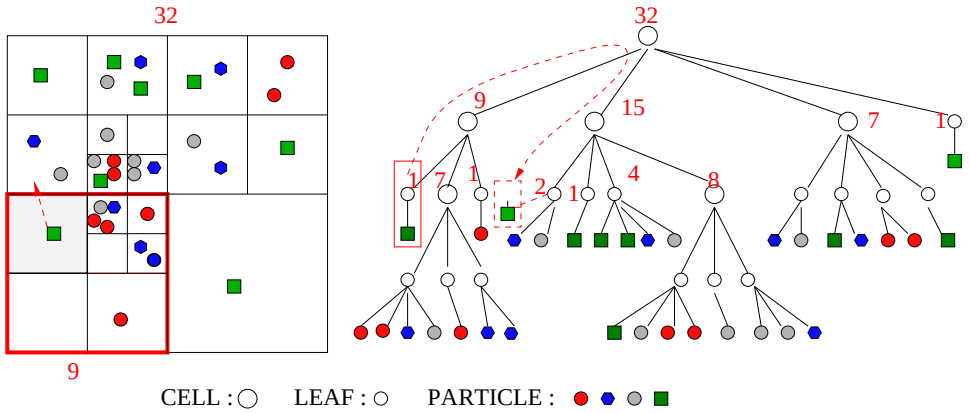

Figure 3. With 4 processors, each subspace at most have 4 particles, the number beside the cells or leaves indicating the number of particles contained in it. The bodies assigned to different processors are represented by different shape.

the same as those in the final global tree (i.e. the space represented by the root cell). This also means that a cell in one tree represents the same subspace as the corresponding cell in another tree. This fact allows the MergeLocalTree to make merging decisions based only on the types of the local and global cells not have to worry about sizes and positions.

This will result in the same global tree for this distribution as that in Figure 3.

The work unit for merging into the global tree has been changed from a particle to a cell or subtree. The number of global insert operations will therefore be greatly reduced, and hence so will the number of synchronizations. This reduction comes at a potential cost of redundant work. Instead of directly being loaded into the global tree, particles will be first loaded into the local tree and then the local trees will be merged. If the partitioning incorporates physical locality, this overhead should be small while the reduction in locking overhead can be substantial.

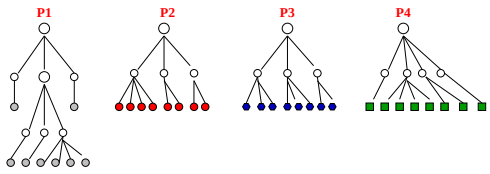

Figure 4. PARTREE: The local trees created by processors with the distribution in Figure 3

2.5 SPACE

The PARTREE algorithm reduces locking substantially, but still has a fair amount of it. The space algorithm eliminates locking entirely in the tree building phase by using a completely different partitioning of bodies and cells than that used in the other phases (like force calculation and update). Instead of using the same partitions of particles from the previous time step, as all the other algorithms so far, in this case the space itself is partitioned among processors anew (differently) for tree-building. Each processor is responsible for loading into the tree those particles that are in subspaces assigned to it. The idea is that if the subspaces match cells (leaf or hopefully internal) of the tree and each subspace (cell) is assigned to a different processor, then two particles assigned to different processors will never have to be inserted in the same cell and there is no need for locking. The problem, of course, is imbalance in tree-building if a simple spatial decomposition is used. To alleviate this, the three dimensional space is first divided into several sub-spaces, and the number of particles in each sub-space computed. If the number in some sub-space exceeds a threshold, then that sub-space is recursively divided again, and so on until the number of particles in every sub-space is below the threshold. This resulting partitioning tree corresponds exactly to the actual octree used in the application, but of course is subdivided to fewer levels, usually below 4. The problem with using the existing partitions is that they are based on load balancing and come for other plans, and its particle assignments to processors do not correspond to disjoint internal space cells.

The resulting sub-spaces are assigned to processors. Since the particles a processor is assigned for tree building are not the same as the ones it is assigned for force calculation (the assignment for force calculation is determined after the tree has been built before), which may result in extra communication and loss of data locality, which is the disadvantage of this scheme.

A processor builds one local subtree for each sub-space assigned to it, and then attaches these subtrees to the global tree (that was partially constructed during the subdivision) without locking. The synchronization and contention problems in the previous four algorithms are almost completely avoided. The trade-off between load imbalance and partitioning time is influenced by the value of the threshold used in subdividing cells.

If we set the threshold value to 8 and the number of processors to 4, for the particle distribution in Figure 3, the two-dimensional space will be partitioned as the left part of Figure 5, and the subspaces and particles in them assigned to the 4 processors. And during the partitioning phase the global tree is also partially created as it is the UPPER part in Figure 5. The ellipses correspond to the space assigned to processors, and indicate the subtrees that need to be created by each processor. Thus P1, P2, P3, P4 will create two, four, one and two subtrees individually using the particles contained in the corresponding subspaces. Thus the bodies assigned to processor for tree building are different from those assigned by the other algorithms. The creating of the local trees does not need any communication. And after they have been created, these local trees can be directly added to the global tree without

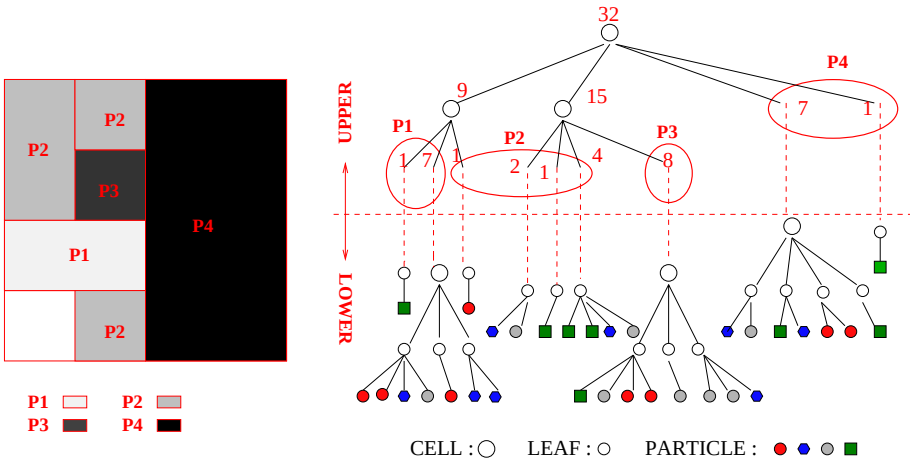

Figure 5. SPACE: the partitioned space and corresponding global tree created using the body distribution in Figure 3

locking, since only one processor will attach a subtree to a given cell position. They are shown as the LOWER part in Figure 5. Finally by pulling the UPPER and LOWER pieces togather we get the same octree as that in Figure 3.

3 Platforms

3.1 SGI Challenge

The Challenge is a bus-based, cache-coherent design with centralized memory. It is capable of holding up to 36 MIPS R4400 processors. Our system has 16 150MHz R4400 processors. They are connected by a 50MHz wide, high speed bus with 256 data bits and 40 address bits, called POWERpath-2. The sustained bus transfer rate can be 1.22GB/sec. This bus interconnects all the major system components and provides support for coherence. The coherence scheme is a 4-state write-invalidate cache coherence protocol. The memory can be up to 16GB. The total miss penalty for a secondary cache miss is about 1100ns.

3.2 SGI Origin 2000

This is a new hardware-coherent distributed shared memory system from SGI. Each node contains two 200Mhz MIPS R10000 processors connected by a system bus, a fraction of the main memory on the machine (1-4GB per node), a combined communication/coherence controller and network interface called the Hub, and an I/O interface called Xbow [13]. Our system has 30 processors with a 4MB secondary cache per processor. The local memory bandwidth is 512MB/s, shared by the local two processors and other devices on the bus. the maximum local memory access time is 313ns, and the remote maximum access time is 703ns. The interconnection network has a hypercube topology. The peak bisection bandwidth is over 16GB/s, and the per-link bandwidth is 1.2GB/s. There are 4 virtual channels for each link and they are bi-directional. The distributed directory protocol is used for cache coherence. The page size is 16kb.

3.3 Paragon

Unlike previous systems, the Paragon was designed without hardware support for a shared address space [15]. The shared address space and coherence are provided in software at page granularity through the virtual memory system. The protocol we used is the Home-based Lazy Release Consistency model (HLRC) [10]. The granularity of coherence is larger (which causes more false sharing of the data), as are the costs of communication and synchronization, but more data is transfered in each communication. So protocol actually is delayed till synchronization points, and the protocol is multiple-written. This means that synchronization events are particularly expensive on such platforms. In the Paragon system, each node has a compute processor and a communication co-processor, which share 64 MB of local memory. Both processors are 50MHz i860 microprocessors with 16k bytes of I-cache and 16 kbytes D-cache. The data caches are coherent between the two processors. The memory bus provides a peak bandwidth of 400 MBytes/sec. The nodes are interconnected with a worm-hole routed 2-D mesh network whose peak bandwidth is 200 Mbytes/s per link. The co-processor runs exclusively in kernel mode, dedicated to communication. The one way message passing latency of a 4-byte NX/2 message is about 50μs.

3.4 Typhoon-0

This system provides hardware support for coherence(access control) at any fixed granularity that is a multiple of the cache line size(64 bytes), but smaller than a page(4kb), and runs the protocol in software on a separate co-processor. The system consists of 16 dual-processor SUN SPARCStation 20s. Each contains two 66MHz Ross HyperSPARC processors, one used for computation and the other to run the coherence protocol. The cache-coherent 50MHz MBus connects all the processors and the memory. I/O devices reside on the 25MHz SBus. Each node contains a Myrinet network interface which includes a 7-MIPS custom processor(LANai) and 128KB of memory. The 16 nodes are connected by 3 Myrinet 8-port crossbar switches. Two ports of each switch are used to connect to other switches. With Myrinet hardware, the host(SPARC) processor and Myrinet LANai processor cooperate with each other to send and receive data. The uncontended latency for a 4-byte round trip message is 40 microseconds. Bandwidth is limited by the SBus. Each node also contains a Typhoon-0 card that logically performs fine-grain access control checks on all loads and stores by physically snooping memory bus transactions and exploiting inclusion. Several coherence protocols have been implemented in softwares from protocol based on

sequentially consistent hardware coherence at fine grain (but running in software) to protocols like HLRC based on SVM at coarse grain. We will examine both these schemes.

4 Performance

We now discuss the results on the four platforms. We report the actual execution times and the speedups obtained over a single processor on the same platform. We use the sequential execution time (in seconds) from the best sequential version of the application to measure speedups. Table 1 shows the sequential execution time for all four platforms. All times and most speedups in this section are for the entire application (though running for only a few time steps), and some time speedups are shown for only the tree building phase. Timing is begun after two time steps to eliminate unrepresentative cold-start and let the partitioning scheme settle down. We also present the percentage of time that is spent in the tree building phase for each algorithm. On some platforms, we can not run all the data sets due to memory limitations. These are marked with dashes in the table.

platform	no. of particles					
	8k	16k	32k	64k	128k	512k
Origin 2000	3.6	7.8	17.1	36.6	77.8	348.2
Challenge	16.0	34.9	76.1	161.3	339.3	–
Typhoon-0	15.4	35.4	72.9	154.2	330.7	–
Paragon	80.1	168.4	366.2	862.5	–	–

Table 1. The Best Sequential Time on the four Platforms

4.1 SGI Challenge

On this platform, all the five versions deliver good speedups (see Figure 6), ranging from 12 to 15 on 16 processors, and the data set size does not have much effect on it. The ORIG-LOCAL version gets the best result (15), followed by the SPACE and PARTREE versions; the ORIG version is the worst. The algorithms do not change performance very much because synchronization is supported in hardware and does not involve any extra communication or software protocol activity, and hence is quite inexpensive. Load imbalance plays a larger role, but should not be very large.

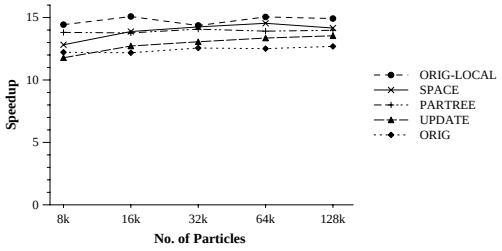

Figure 6. Speedups for 16 processors on SGI Challenge.

The ORIG version performs a little worse than the others. the reason is that its data structures cause more false sharing and communication as discussed earlier. Figure 7 shows the tree building cost for the 128k data set for each algorithm. We find that even on hardware cache-coherent machines, the total execution time is highly affected by tree building cost, but not for the good algorithms (ORIG-LOCAL, PARTREE, SPACE and even UPDATE at this scale).

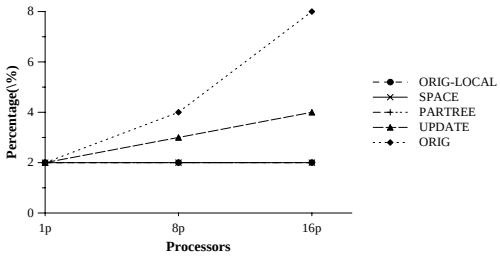

Figure 7. Tree-building cost for 128k particles and 16 processors on SGI Challenge, as a percentage of total execution time. The cost percentages of ORIG-LOCAL, PARTREE and SPACE algorithms are the same

4.2 SGI Origin 2000

The Origin machine we use currently has 30 processors. In this section we examine how the algorithms compare and how the comparison changes with the data set size and with the number of processors.

4.2.1 Effect of Data Set Size

The speedups on 30 processors of these five algorithms for different data set sizes is shown in Figure 8.

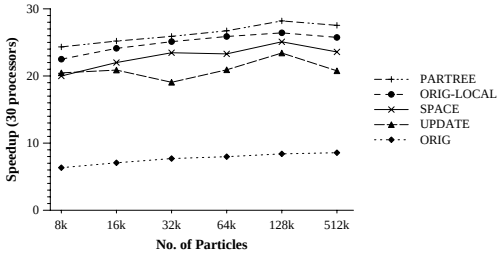

Figure 8. Speedups on SGI Origin 2000

We find that the ORIG-LOCAL, PARTREE and SPACE versions all work quite well, their performance is very close to each other, and they scale very well with larger data sets. The many synchronizations needed in building the tree do not affect the performance very much here either since locks are supported efficiently in hardware. In fact, SPACE does a little worse than PARTREE and ORIG-LOCAL due to load imbalance and locality.

The interesting thing is that unlike on the Challenge there is a big gap between the performance of ORIG version and the performance of the other four versions. The reason here is that a ”remote” (communication) cache miss costs much more than a local miss on the Origin, and also causes contention in the interconnect, while on the Challenge the costs and contention effects in the two cases are quite similar. There are many more remote misses in the ORIG version than the others, mostly because of false sharing in the data structure and to a lesser extent because a processor’s cells are not allocated locally.

Greater "remote access" will also further exacerbate the load imbalance. Table 2 shows the time spent for the BARRIER operation for 64k and 512k data sets using 16 processors. The time used for ORIG version is highest, almost 15 times longer than ORIG-LOCAL version and it is followed (distantly) by the SPACE version.

	ORIG	ORIG-LOCAL	UPDATE	PARTREE	SPACE
64k	0.51	0.022	0.044	0.025	0.076
512k	5.25	0.340	0.412	0.231	0.862

Table 2. Time (in seconds) spent for BARRIER operations in the application program for 64k and 512k data set

Figure 9 shows the speedup for the tree building phase alone (accumulated on the measured time-steps). Comparing with Figure 8, they reflect almost the same characteristics. The speedups for the whole application have relatively similar curves across schemes to the speedups of tree building phase, though the speedups for the tree building are much lower.

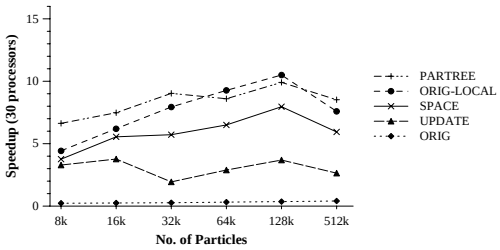

Figure 9. Speedups on Origin 2000 for the tree building phase with 30 processors

4.2.2 Effect of Number of Processors

Figure 10 shows the speedups for 16, 24, and 30 processors with different tree building algorithms for a 512k particle data set.

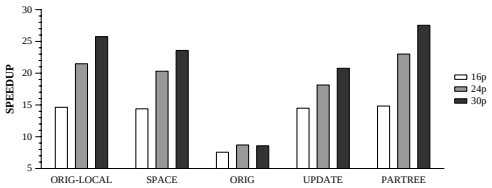

Figure 10. Speedups on Origin 2000 for 16, 24, 30 Processors with a 512k-particle data set

The PARTREE, ORIG-LOCAL, and SPACE versions scale very well with the number of processors, with the PARTREE version always yielding the best speedup (it has the least communication, which is more expensive here). The UPDATE version behaves a little worse. The corresponding tree building cost is shown in Figure 11, and clearly correlates well with overall performance. In the ORIG version, the tree bulding cost takes up almost 60% of the total execution time at 30 processors.

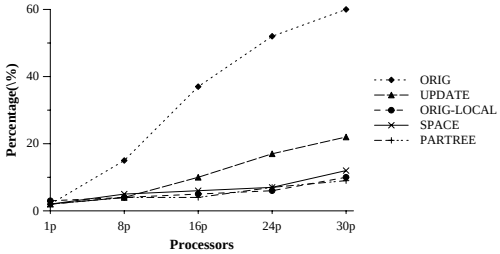

Figure 11. Percentage of execution time spent in Tree Building on Origin 2000 for 512k particles on 30 processors

4.3 Paragon

With the shared address space supported in software at page granularity using shared virtual memory, the performance characteristics of this platform reveals a completely different story. Even for 16k particles, the time needed for ORIG-LOCAL, ORIG, UPDATE versions is almost intolerably long given the limited access we had to the machine. They lead to very substantial slowdown over a uniprocessor execution. So we only collect data for the SPACE and PARTREE versions. The reason for poor performance is that we use a software protocol with a relaxed memory consistency model on this platform. All of the (expensive) communication and protocol activity occurs at synchronization events such as locks with the relaxed software protocol (to reduce false sharing at page granularity which would otherwise have destroyed performance anyway). Also there is tremendous serialization at locks because critical sections are greatly dilated by expensive page faults and protocol activity within them [14]. The ORIG, ORIG-LOCAL, and UPDATE algorithms use a lot of locking, so they perform very poorly (see the end of section 4 for measurements of dynamic lock counts). Since the PARTREE version is more coarse grained and needs less locking, it performs better. However, the only algorithm that performs well is the SPACE algorithm, since it completely avoids using locks during the tree building phase.

Figure 12 gives the speedups for the application using the SPACE and PARTREE versions, showing that the former performs much better. The reduction in synchronization far outweighs any increase in load imbalance or in communication due to loss of locality across phases.

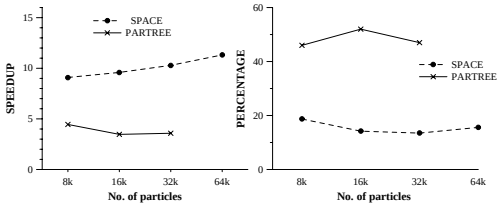

Figure 12. Speedups and Percentage of Time spent in Tree Building on Paragon with 16 Processors

For the SPACE algorithm, the tree building cost is usually less than 20% of the total execution time. But for the PARTREE algorithm, the tree-building cost usually approximates 50 percent. Tree building clearly becomes the bottleneck quite quickly, and it is important to use algorithms with very little synchronization on

such SVM platforms. Another page-grained SVM platform will be examined next.

4.4 Typhoon-0

In this system, the granularity of coherence can be varied (in powers of two) with hardware support for access control but the protocol runs in software. We test two protocols on this platform. One is the same HLRC SVM protocol as on the Paragon with 4k pages. This allows us to examine the same protocol on platforms with very different performance characteristics. The other is sequential consistency with 64 bytes coherence granularity, the protocol for which this machine was designed[11]. With sequential consistency, protocol activity occurs at memory operations and is not delayed till synchronization points. Large coherence granularities like 4k bytes would thus perform very poorly due to false sharing, but synchronization operations are not nearly as expensive as in relaxed SVM protocols.

4.4.1 HLRC SVM

We obtain results for all five algorithms from 8k to 64k particles on this platform. 64k particles is the largest data set size we can run due to memory limitations. Figure 13 shows the speedups on 16 processors across problem sizes.

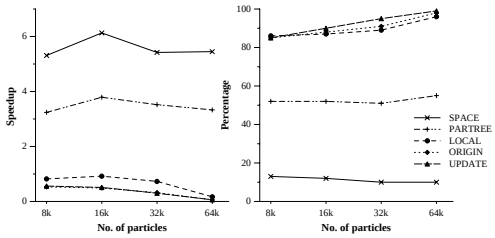

Figure 13. Speedups and Percentage of Time spent in Tree Building on Typhoon-Zero for 16 Processors using the page-based HLRC SVM protocol

Once again the application with the SPACE version of tree building vastly outperforms all the others. And the one with the PARTREE version is second. The other three usually deliver a slowdown. With the 64k data set, they are almost 16 times slower than the sequential time. The results verify that page-grained SVM platforms are not suitable for fine-grained program phases with high frequency of synchronization.

With the SPACE algorithm, the percentage of time spent in tree building is close to 10 percent for most problem sizes on 16 processor systems, but for PARTREE it again reaches 50 percent. For UPDATE, ORIG, and ORIG-LOCAL, with increasing data set size almost all the time is spent on the tree building (which takes <3% of the time on an uniprocessor). The speedups for tree building phase alone are shown in Figure 14. Speedups are poor. With 16 processors, the SPACE version can get 1.5 times faster, and all the other versions lead to a slowdown. In this case the data structure difference between ORIG and ORIG-LOCAL are unimportant compared with the synchronization cost, the opposite of efficient hardware coherent distributed machines like the Origin 2000.

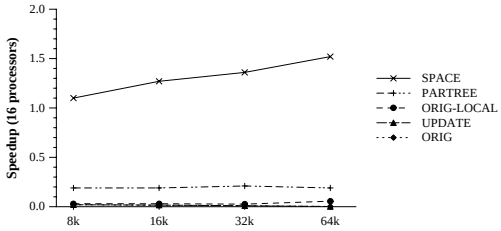

Figure 14. Speedups for the tree building phase on 16 processors using the page-grained HLRC SVM protocol

4.4.2 Fine-grained Sequential Consistency

The coherence granularity, coherence protocol and consistency model in this case are similar to those on the Origin, but the protocol is run in software and with less aggressive hardware support. Latencies and communication controller occupancies are much larger, and bandwidth is smaller. Compared to HLRC on the same platform, under the Sequential Consistency protocol with a granularity of 64bytes the performance difference between algorithms becomes much smaller. The ORIG-LOCAL version gets the best speedup (7 on 16 processors with 16k particles), the SPACE, PARTREE, and UPDATE are almost the same with a speedup of 4, the ORIG version a little worse. In this model the synchronization cost is greatly reduced since there is no protocol overhead incurred in implementing the synchronization as discussed earlier, (the consistency model is not relaxed), and all the communication does not happen at synchronization points but at the memory operations themselves. The poor performance of the ORIG version is caused by whatever false sharing occurs at 64 byte granularity, since communication is much more expensive here than on the Origin. Even on the same platform, the picture is very different than that of HLRC, where synchronization is far more important than load balance or false sharing.

To conclude this performance section, we present the number of locks dynamically used for each processor by different algorithms in the tree building phase on two platforms, SGI Origin2000 and Typhoon-zero with HLRC protocol (Figure 15). On the Typhoon-zero platform, more locks are required than on the Origin because the HLRC protocol requires additional synchronization to make the code release consistent. At synchronization points, the shared pages will be invalidated if they have been written by others. Thus more locks will potentially cause more page faults, as will greater communication and false sharing on the Typhoon-zero platform.

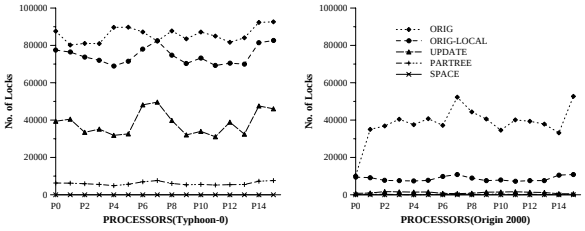

Figure 15. The number of locks of each processor executed in the tree building phase for two time steps with 64k particles using 16 processors

Figure 15 show that from the ORIG version to the SPACE version, the number of locks falls off very quickly. This validates the design strategy of our algorithms: For commodity-oriented SVM platforms, the goal is to reduce synchronization frequency (in addition to providing load balance and reducing communication), even if this impacts load balance a little. The SPACE algorithm runs dramatically better on page-based SVM platforms that other algorithms and well enough on the other platforms. From ORIG, ORIG-LOCAL, UPDATE, PARTREE to SPACE, in order to avoid the expensive synchronization on commodity hardware oriented machines, the number of lock operations becomes less and less.

5 Related Work

Parallelizing hierarchical N-body applications for hardware-coherent (CC-NUMA) and message passing machines has been well studied earlier. Salmon [4] implemented a Barnes-Hut application on a message passing machine. He used a *orthogonal recursive bisection* partitioning technique to obtain good speedups on a 512 processor NCUBE. Singh et al studied this application on a prototype CC-NUMA machine, the Stanford DASH, for both Barnes-Hut and the Fast Multipole Method, and developed a new partitioning technique called *costzones* that achieved better performance and ease of programming [3]. That paper also introduced the PARTREE algorithm in an appendix, and showed it to improve performance when only a single particle was allowed per leaf cell in the Barnes-Hut tree. However, later it was found that allowing multiple particles per leaf cell improved application performance substantially and essentially eliminated the difference between tree-building algorithms, at least on hardware CC-NUMA machines at the time, so the algorithm was no longer used and distributed. Finally, Warren and Salmon implemented an approach very similar to *costzones* using message passing and a hashing approach to maintain a global tree [2]. There has been other research in parallel N-body applications, but none focusing on tree building methods for shared address space systems.

This research focuses on performance across a whole range of coherent shared address space platforms, from modern, commercial hardware cache-coherent machines with centralized and distributed memory to page-based software shared virtual memory systems. The particular focus is on the tree building phase, which brings down overall performance on commodity-based systems. Previously proposed algorithms are tested, and new algorithms proposed that outperform them and are the best across the range of systems.

Another recent paper made a start on studying performance portability for a shared address space [8]. It includes a hierarchical N-body application as one of its many applications examined, but it does not describe the tree-building algorithms or examine performance on real commodity-based systems (only through simulation, and only for page-based shared virtual memory). This paper is a much more in-depth study of parallel tree-building algorithms, describes the new algorithms for the first time , and quite comprehensively studies the resulting performance of N-body applications across a wide range of real platforms.

6 Conclusions

While tree building is not a significant portion of the sequential execution time in hierarchical N-body applications, and also not very significant on efficient hardware-coherent systems at moderate scale, it becomes crucial to parallel performance of the whole application on modern commodity-based coherent shared address space platforms such as networks of workstations and SMPs. On these platforms, which implement coherence in software, using the tree building algorithms that are accepted for hardware-coherent systems often leads to either poor application speedups or even substantial slowdowns. If shared memory programming is to port effectively to these commodity communication architectures like message passing does, which is very important in making this attractive model well accepted by application users (so a single code, hopefully made much easier to develop by the shared memory model, will perform well enough on all available systems), then for these applications the tree building performance problem must be solved in a performance portable manner.

This paper studied several different tree-building algorithms and data structures on four very different platforms that span the range of commercial and research shared address space systems. We found that the original algorithms developed and distributed before perform quite well on efficient hardware-coherent systems, they do not perform well at all on commodity page-based shared virtual memory systems. An algorithm called PARTREE was previously discussed. It improves performance a lot on the commodity platforms, but not enough. An algorithm that incrementally updates the tree based on particle movements instead of rebuilding it every time step doesn't do even as well as PARTREE. By understanding that lock-based synchronization in the tree building phase is the key performance bottleneck on shared virtual memory platforms, we developed and implemented a new, space-based tree-building method called the SPACE method that uses different partitions for tree-building than for the rest of the computation in a time-step. This method eliminates synchronization within the tree-building phase, but at the cost of some communication, locality and load balance.

Our goal was to understand what algorithms might best be performance portable across all the platforms, and how different algorithms interact with different platforms. All the algorithms have been described (some for the first time) in this paper.

We found that no single version always delivers absolutely the best performance on all platforms: hardware-coherent and commodity-based. From the analysis of the results, the important bottlenecks are different on different platforms, so different algorithms do better. The ORIG-LOCAL version is best on SGI Challenge, the PARTREE version is best on the SGI Origin 2000 platform, the SPACE version is the best on the Paragon and on Typhoon-Zero for the HLRC shared virtual memory (SVM) protocol, and the ORIG-LOCAL version is the best on Typhoon-zero under sequential consistency. However, while the differences are small on the Challenge and the Origin (except for the original ORIG version on Origin2000), they are especially large on the SVM platforms. Overall, the new SPACE algorithm has by far the best overall performance portability across all systems. Its performance is dramatically better on commodity systems when it is better, and not much worse on other systems when it is worse. This is because it sacrifices only a little communication and load imbalance to the benefit of a huge amount of synchronization, and the latter is extremely important on SVM platforms since synchronization is where protocol activity is incurred and critical sections are often dilated a lot by page faults that occur within them, increasing serialization. In fact, the SPACE algorithm is the only algorithm that delivers effective application speedups on these page based platforms. The PARTREE algorithm comes next in overall performance portability, and also because it reduces communication at the cost of some load imbalance. We might draw some more general conclusions from this, which corroborate the conclusions in [8]. First, coarse grain applications that synchronize less fre-

quently have much better performance portability across the range of systems than fine grain applications, even at the cost of some load balance and locality. Second, obtaining good speedup on the more commodity oriented machines for this class of applications is overall much more difficult than on the hardware cache-coherent machines.

Future work includes understanding how the different algorithms compare at large scale even on hardware-coherent systems—i.e. whether algorithms that port well to commodity-based platforms are also the right algorithms for tightly-integrated systems but only reveal this at larger scale—and to see whether even the best tree-building algorithms allow performance to scale up on commodity-based communication architectures.

7 Acknowledgments

Chris Holt developed earlier versions of some of the tree building codes. The SPACE algorithm was revisited by us, after having earlier shelved a similar approach for hardware-coherent machines, as a result of discussions with Mukund Raghavachari. We thank David Wood and Mark Hill for use of the Typhoon-zero platform.

References

[1] J. Barnes and P.Hut, "A hierarchical O(NlogN) force calculation algorithm", nature,vol. 324, p.446, 1986.

[2] M. Warren, J. Salmon, "A Parallel Hashed Oct-Tree N-Body Algorithm", Proceedings of Supercomputing'93.

[3] JP. Singh, C.Holt, et al, "Load Balancing and Data Locality in Hierarchical N-body Methods",Technique Report, Stanford University, CSL-TR-92-505, 1992.

[4] John K. Salmon. "Parallel Hierarchical N-body Methods", PhD thesis, California Institute of Technology, Decenber 1990.

[5] K. Li, P.Hudak, "Memory coherence in shared virtual memory systems", ACM Transaction on Computer Systems, 7(4):321-359, 1989

[6] Ioannis Schoinas et al, "Fine-grain Access Control for Distributed Shared Memory", Sixth International Conference on Architectural Support for Programming Languages and Operating Systems October, 1994.

[7] D.J.Scales, K. Gharachorloo, And C.A.Thekkath. "Shasta: A low Overhead, Software-only Approach for supporting Fine-Grained Shared Memory", International Conference on Architectural Support for programming Languages and Operating systems, October 1996.

[8] D. Jiang, H. Shan, JP. Singh, "Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-Coherent Multiprocessors", in Proceedings of Principles and Practice of Parallel Programming, June 1997.

[9] JP Singh, C.Holt, T. Totsuka, A.Gupta and J.L.Hennessy, "Load Balancing and Data Locality in Adaptive Hierarchical N-body Methods: Barnes-Hut, Fast Multipole, and Radiosity", Journal of Parallel and Distributed Computing, June 1995

[10] Y. Zhou, L. Iftode, and K. Li, "Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems", In proceedings of the Operating Systems Design and Implementation Symposium, October, 1996.

[11] Reinhardt S., Larus J., and Wood D., "Tempest and Typhoon: User-Level Shared Memory", In proceedings of the 21th International Symposium on Computer Architecture, April 1994.

[12] "Performance Tuning for the Origin2000", http://www.sgi.com

[13] J. Laudon, D. Lenoski, "The SGI Origin2000 : A CC-NUMA Highly Scalable Server", In proceedings of the 24th International Symposium on Computer Architecture, June 1997.

[14] L. Iftode, JP Singh, K. Li, "Understanding Application Performance on Shared Virtual Memory Systems", In proceedings of the 23th International Symposium on Computer Architecture, May 1996.

[15] ROSS Technology, Inc. SPARC RISC User's Guide: hyperSPARC Edition, September 1993.