The Fast Multipole Method on Ram, the Altix 3700 at Oak Ridge National Laboratory*

Collin McCurdy[1,2], Scott Studham[2], and Charles Fischer[1]

[1]Department of Computer Sciences, University of Wisconsin–Madison
[2]Oak Ridge National Laboratory

Abstract

This paper describes experiments with two parallel implementations of the Fast Multipole Method – one using the shared memory programming model (OpenMP), the other the message passing model (MPI) – on the SGI Altix 3700 at Oak Ridge National Laboratory. The purpose of our experiments was to test our hypothesis that direct access to shared memory enabled by hardware-based cache coherence is a better match for the fine-grained communication requirements of the tree-code computations of the algorithm, while the message passing version's gather/scatter approach is more efficient for the more coarse-grained interaction computation.

After providing some context for the discussion with a description of the algorithm and our parallelizations, we begin the analysis of our experiments by describing the unexpected initial results on the Altix, and how we went about obtaining more representative data. Then, having addressed the problem, we turn our attention to the larger implications of our experiments.

We find that, as we expected, the tree-code is much better suited to shared memory, both from a communication and an actual performance standpoint. On the other hand, contrary to our expectations, and though communication requirements appear to favor the gather/scatter approach to interaction computation taken by the MPI version, the shared memory version consistently outperforms the message passing version.

*Prepared by OAK RIDGE NATIONAL LABORATORY P.O. Box 2008 Oak Ridge, Tennessee 37831-6285 managed by UT-Battelle, LLC for the U.S. DEPARTMENT OF ENERGY under contract DE-AC05-00OR22725

1 Introduction

This paper describes experiments with two parallel implementations of the Fast Multipole Method – one using the shared memory programming model (OpenMP), the other the message passing model (MPI) – on the SGI Altix 3700 at Oak Ridge National Laboratory.

The Fast Multipole Method (FMM) is an example of a class of algorithms that uses hierarchical data structures (i.e., trees) to rapidly approximate the numerical solution of the $O(N^2)$ N-Body problem. The FMM has been shown to approximate the solution with high accuracy in $O(N)$ time, though the constant factor is quite large.

A potential application of the FMM of particular interest to molecular scientists is the computation of long-range interactions in molecular dynamics simulations. The algorithm exhibits a high degree of spatial locality, which should result in low communication requirements on parallel supercomputers. This feature makes it an attractive alternative to the communication intensive FFT approach to speeding up long-range interactions currently used by the Blue Gene project [1].

On the other hand, irregular communication patterns make efficient parallelizations of the algorithm difficult, if not impossible, to achieve, particularly for the distributed memory architecture that is the hallmark of most existing large-scale supercomputers.

While the bulk of the run-time of the algorithm is spent in interaction computations – whose communication requirements, though irregular, can be handled efficiently on distributed memory architectures – the remaining time is spent doing fine-grained tree-traversals which can become a bottleneck as processors are added to the problem.

The purpose of our experiments was to test our hy-

pothesis that direct access to shared memory enabled by hardware-based cache coherence is a better match for the fine-grained communication requirements of the tree-code computation, while the message passing version's gather/scatter approach would be more efficient for the interaction computation.

Initial results strongly supported our hypothesis: those pieces that we expected to perform well did in fact perform well, and those that we expected to perform poorly performed poorly. However, the *degree* to which the Altix performance degraded, on the pieces we expected to do poorly, as we added processors, led us to question the results.

After taking some time to understand the performance of our applications on the Altix, we were able to obtain much more realistic results which, interestingly, in some respects contradicted our hypothesis.

Our analysis of the experiments in this paper follows a similar trajectory: after providing context for the discussion – with a brief explanation of the Fast Multipole Method algorithm in Section 2, and our parallelization strategies in Section 3 – we begin the analysis in Section 4 by describing the unexpected initial results on the Altix, and how we went about obtaining more representative data. Then, having addressed the problem, we turn our attention to examining the implications of the data for our hypothesis.

Finally, in Section 5 we draw conclusions, both about programming the Altix, and more generally about programming shared memory.

2 The Fast Multipole Method

While the computational kernel is actually due to Anderson [2], the algorithmic structure of our codes is exactly the same as Greengard and Rokhlin's Fast Multipole Method [4].

As we have already noted, the FMM is a member of a class of algorithms known as "hierarchical methods". In general, hierarchical methods gain computational advantage in the solution of the N-body problem by aggregating the effects of groups of bodies a sufficient distance away from the evaluation point.

Hierarchical methods create and maintain groupings of bodies using a data structure known as an *octtree*. One constructs an octtree starting with a root box that contains all of the bodies and then recursively subdividing boxes into 8 sub-boxes of equal size until a stopping condition is met. Recursion in an *adaptive* octtree may stop at different levels at different locations in the tree, depending on the distribution of bodies. Figure 1 gives an example of an adaptive quadtree, the two-dimensional version of an octtree.

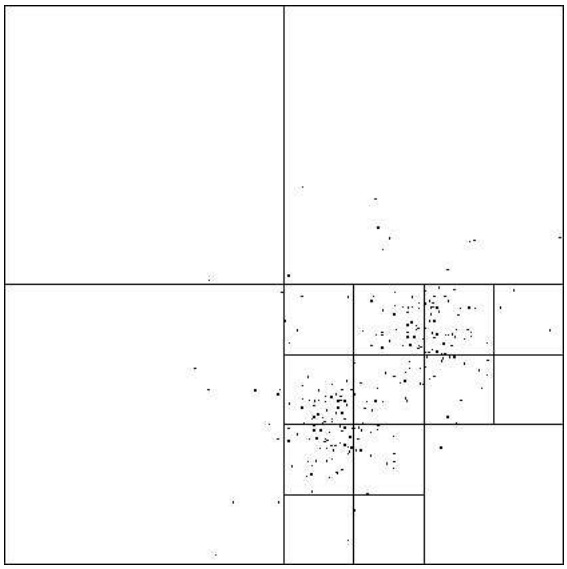


Figure 1: Example of an adaptive *quad*tree.

Rather than requiring each body to traverse the tree individually, as in the $O(n \log n)$ Barnes-Hut algorithm [3], the FMM makes use of the observation that when two boxes are "well-separated", the far-field effect of all the bodies in one box on those in the other (and vice-versa) can be approximated as a single interaction between the *boxes*. Such interactions between well-separated boxes occur at all levels of the tree, and the savings in computation enable the FMM to compute far-field forces in $O(n)$ time.

In the remainder of this section we describe, with the aid of Figure 2, the steps of the *adaptive* algorithm, developed for use with adaptive octtrees.

The first step is to create the octtree. Once the tree is formed, to simplify implementation and maximize cache locality, the algorithm traverses the tree to compute several types of interaction lists for each box.

There are three types of interaction computations and therefore three lists (see Figure 3): boxes in *List1* are adjacent leaf boxes (and therefore are not sufficiently distant from each other to allow approximation); boxes in *List2* are the same size and well-separated (that is, sufficiently distant from each other to allow approximation); finally, boxes in *List3/4* are different sized and well-separated from the perspec-

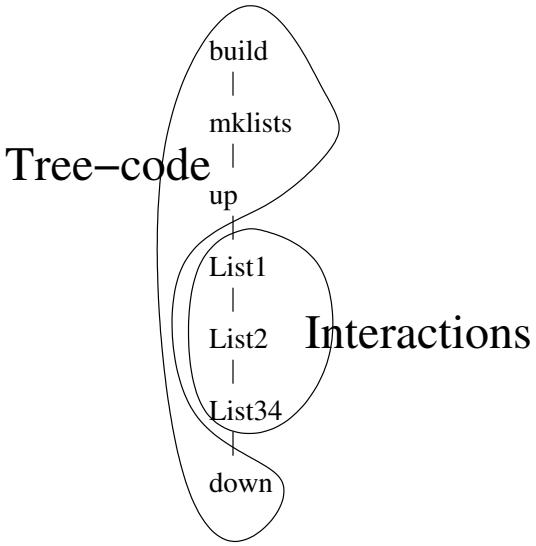


Figure 2: The steps of the adaptive FMM algorithm and how we divide them into the *Tree-code* and *Interactions* summary categories.

tive of one of the boxes but not the other.

An upward pass over the boxes in the tree establishes the far-field approximations for each box: At the leaves, the approximation for a box is computed from the bodies within; at higher levels, the approximation for a box is computed from approximations for boxes it contains.

Then each box computes interactions with all the other boxes on its interaction lists.

Finally a downward pass propagates results, collected at each level in the form of a "local-field potential," from box to box, all the way down to the bodies at the leaves.

3 Parallelization Strategies

3.1 Message Passing: MPI

We introduced our MPI implementation in [7]. It draws on three sources: The FMM program in the SPLASH-2 suite from Stanford [10], an HPF library implementation [5] of the FMM, and Warren and Salmon's "hashed-octtree" approach to a different tree-code [9]. We showed in [7] that, due to more efficient communication mechanisms, the performance of our code compared favorably to the HPF version on up to 128 processors of a Cray T3E.

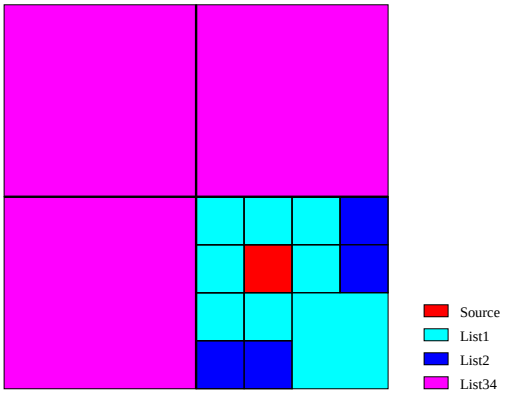


Figure 3: Interaction lists for a box in the example quad-tree from Figure 1.

Non-local Data Access. The primary barrier to overcome in a message passing implementation of the FMM is the irregular access of data imposed by the adaptive octtree data structure. Our implementation uses a form of inspector-executor adapted to the hash-table representation of the octtree we borrowed from Warren and Salmon.

Each node in the tree is represented by a unique identifier, derived from the node's location in the tree, which is used as a key for accessing nodes in a hash-table representing the tree. This approach simplifies management of distributed trees in two ways. First, the identifier for a node is the same on *all* processors. Second, integration of non-local nodes into a local tree is simple: data for non-local nodes is simply added to a processor's hash-table.

The following pseudo-code demonstrates how the MPI implementation inspects a typical loop-nest for non-local references and communicates required non-local data:

```
foreach Node n accessed in Loop l
    if n is NOT in localtree then
        nonlocal.Insert(n)

othersNonlocalIds = Alltoall(nonlocal.ids)
found = localtree.Search(othersNonlocalIds)

myNonlocals = Alltoall(found)
localtree.Insert(myNonlocals)
```

Body Distribution. Another significant problem that the FMM poses for a distributed memory implementation is the initial distribution of bodies across processors such that communication is minimized and load balance is maximized. We have found that sort-

ing bodies according to their position along a Hilbert space-filling curve [8] is an excellent match for this algorithm: since both the Hilbert curve and the octtree recursively divide space in half along each dimension, all bodies in the same leaf of the octtree are contiguous after the sort. We can then partition the sorted sequence of bodies among the processors by assigning each a contiguous range of bodies and, with a little care, ensure that each processor is assigned all bodies in a given subtree of the octtree before the tree is even built.

Parallelization. Below we provide details about the parallelization of each step in the algorithm from Figure 2:

build – Given the partitioning strategy described above, we are able to construct octtrees locally, except for a brief communication phase in which processors exchange information about *shared boxes* (boxes at upper levels of the tree whose sub-boxes lie on more than one processor) to ensure that the representation of these boxes is globally consistent.

mklists – The fine-grained nature of the computation in the list-construction algorithm, combined with large communication requirements, can cause this step to be a major bottleneck in the parallelized application if special care is not taken. We have transformed the uniprocessor list construction algorithm into a level-by-level form which allows us to use the inspector-executor strategy noted above, at each level, to gather non-local data before computing lists for that level.

up, down – As noted earlier, we replicate information about shared nodes at the uppermost levels of the tree to all processors and ensure that all nodes in a subtree below any non-shared node are located on the same processor. This partitioning strategy avoids communication in the downward pass and requires only a single communication step in the upward pass when moving from private nodes to the shared parents.

List1, List2, List34 – As in list construction, we communicate non-local data required during the interaction computations using our variation on the inspector-executor technique. To ensure load-balance, we move data for boxes involved in each of the three computation phases into a "weighted-block" distribution immediately prior to that phase. This involves looking in the work-list for each box involved in the computation to determine the amount of work it will do, and then minimally redistributing the boxes such that each processor will have approximately the same total amount of work.

3.2 Shared Memory: OpenMP

The shared memory implementation represents a merging of our MPI derivative back into its SPLASH parent, with the aid of OpenMP parallelization pragmas.

Non-local Data Access. The primary difference between the OpenMP and MPI codes is the former's ability to directly access non-local memory. This ability manifests as an advantage in two ways: first it makes the code, particularly the tree-code, much easier to write and understand. As a rough measure of this advantage, the our OpenMP version requires 2/3 the number of lines of C code relative to the MPI version.

Secondly, direct access of non-local memory potentially makes the code much more efficient, since there is no need for extra computation and synchronization associated with an inspection phase for irregularly accessed data. On the other hand, there is also potential for inefficiency due to the cache-based nature of hardware coherence: reuse of non-local data that does not stay cached, whether due to capacity or coherence misses, can severely degrade performance.

Body Distribution. Despite the ability to access global memory, due to locality concerns it is still beneficial for both load balance and communication to "distribute" data appropriately, especially when the architecture exposes local memory to the user. For this reason, we use exactly the same algorithm to partition bodies, sorting them according to their position along a Hilbert space-filling curve, as in the MPI code.

Parallelization. Below we provide details about each algorithmic step:

build – As with the MPI version, the partitioning strategy described above allows most of the octtree construction to be done locally. While merging the trees still requires some synchronization through locks, it is minimal compared to the amount required in the original SPLASH code. However, unlike the MPI version, there are no shared boxes in the OpenMP version, which guarantees a certain amount of contention as processors add their boxes from the top levels of their local trees to the global tree.

mklists, up, down – The fine-grained computation in these steps, coupled with their limited reuse of non-local data, makes them ideally suited to the direct memory access capability of shared memory.

List1, List2, List34 – We are not currently load-balancing interactions. While our use of inspector-executors to communicate non-local data in the MPI interaction computations dictates that all processors do all *List1* computations, at the same time, then all *List2* and then all *List34*, shared memory allows much more flexibility in terms of overlapping computations. Not yet having determined the optimal way to order the List computations for the shared memory version, and also requiring a meaningful way to compare computation times with the MPI version, we currently simply use the same strategy as the MPI version but without the load balancing step.

4 Experiments

In this section we describe our experiments with the two versions of the FMM, described in the previous section, on Ram, the Altix at Oak Ridge National Laboratory.

4.1 Setup

Ram has 256 Intel Itanium2 processors running at 1.5GHz, each with 6MB of L3 cache. The machine features 8 GB of memory per processor for a total of 2 TB of shared memory.

We use as controls for our experiments two systems based on the same Itanium2 processors as Ram, but which utilize very different interconnection networks. To validate OpenMP results we use the 64 processor HP Superdome at the University of Kentucky. For comparing MPI results we use the Teragrid array of workstations at NCSA.

For our experiments we have chosen to examine "constant work" scalability. For all runs, as we increase the number of processors, we increase the data size proportionately, keeping the number of bodies simulated per processor constant. Unless otherwise noted, that number is 4000. Since the FMM is purportedly an $O(N)$ algorithm, perfect scalability would result in constant run-times as the number of processors is increased, evidenced by a flat horizontal line in plots where processors increase on the X axis and times are measured on the Y axis.

We present two categories of results: 1) *summary* results, in which we divide total time into "tree-code" versus "interactions", and 2) *detailed* results, in which we further divide the summary data into its constituent parts; our division of the steps of the sequential algorithm between "tree-code" and "interactions" is as noted in Figure 2. As described in Section 3, both parallel algorithms add an initial *sort* step to the algorithm, which we add to the "tree-code" category.

While most of the measurements we present are times – measured with either *gettimeofday()* or where possible the more precise *clock_gettime()* interface – we present per-processor cycle and level three cache (L3) miss counts obtained from Itanium2 performance counters when appropriate. (L3 misses are significant since they must be satisfied from memory, potentially non-local, in which case they cause communication.) For simplicity, in most cases, the result we present is the average across all processors.

4.2 Performance Expectations

Before we start looking at results, a word about expectations. As discussed earlier, we expect the irregular, extremely fine-grained communication of the tree-code phases to favor hardware-based cache coherence.

On the other hand, the more coarse-grained communication requirements of the interaction phases – while still irregular – favors a message-based approach for two reasons:

- For maximal efficiency, non-local data should *stay* where it is after has been communicated. In cache-based approaches, evicted non-local data must be re-fetched. One of the key observations in [7] is that there is heavy reuse of non-local data in the *List2* interactions, which would seem to translate into a high likelihood for redundant communication as non-local data is evicted from the cache.

- Again, for maximal efficiency, updates to non-local data should not interfere with one-another. In a message-based approach, updates are naturally handled as reductions. On the other hand, in the shared memory paradigm, updates are more naturally expressed using locks to synchronize access to the data, potentially resulting in both lock-contention and unnecessary communication traffic as the updated values propagate from cache to cache.

	OpenMP				MPI			
	Superdome		Altix		Altix		Teragrid	
Procs	Tree	Inter	Tree	Inter	Tree	Inter	Tree	Inter
4	0.036	0.48	0.027	0.42	0.035	0.58	0.044	0.79
16	0.048	0.53	0.047	0.51	0.087	0.67	0.090	0.90
64	0.11	0.63	0.11	2.8	0.80	0.90	0.34	1.1
128	N/A	N/A	0.37	7.0	6.3	1.0	0.92	1.1

Table 1: Initial results: OpenMP on Altix vs OpenMP on HP Superdome and MPI on Altix vs MPI on Teragrid cluster

4.3 Initial Results

Table 1 presents the initial results of our experiments. As expected, OpenMP-style programming is much more efficient on the tree code portion of the program than MPI, but is less efficient than MPI (which fetches all non-local data before computing) for interactions. However, the *degree* of inefficiency on the Altix, contrasted with the performance of the control architectures, leads us to question the results.

Specifically, while we expected from the outset that the Superdome's more SMP-like performance (albeit, at the cost of limited architectural scalability) would give it an advantage, the 4X performance difference at 64 processors was unexpected.

Worse, while we expected that the shared memory platform would be a *better* match for the collective communication calls (MPI_Alltoall, MPI_Alltoallv, MPI_Scatter) used heavily by the MPI version than the distributed network of workstations at NCSA, according to our results its actual performance 6X slower.

In the next sections we examine the causes of these unexpected results.

4.3.1 Altix OpenMP Performance

Figure 4(a) and (b) show cycle counts and L3 misses for each algorithmic step for the OpenMP version of FMM on the Altix. The correlation between the extremely high cycle and L3 miss counts in the List2 computation phase led us to hypothesize that a large number of non-local misses was saturating the network.

After discussions with an SGI representative, and closer inspection of the List2 interaction code, we narrowed the problem to the allocation of a large table heavily used by during the List2 interactions.

The Altix is a Non-Uniform Memory Access (NUMA) architecture, which means that memory modules are distributed along with processors across all the nodes of the machine, connected via a communication network. While all memory is globally addressable, processors can access memory located near them faster than memory across the network.

Such an architecture requires a policy to determine in which node's memory a page will live. The Altix uses a "first-touch" allocation policy: pages are allocated in the local memory of the first processor that touches them.

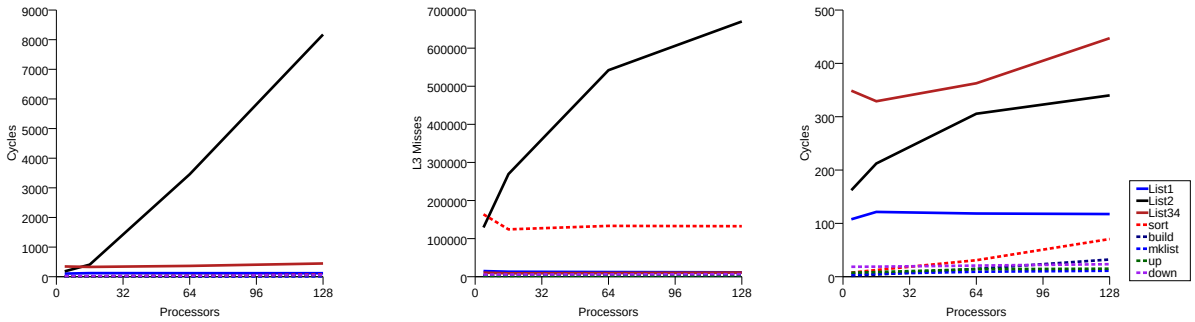
Because we allocated and initialized the table in sequential start-up code before entering the parallel section, it was first touched by Processor 0, and therefore all the pages associated with the table were stored in its node's local memory. As a consequence, every subsequent access to table data that missed in the issuing processor's cache had to be satisfied by the node containing Processor 0, leading to tremendous network congestion and poor performance.

A simple fix is to delay allocation until the parallel section, and then replicate the table by allowing each processor to allocate and initialize a private copy. The downside is an increase in the global memory footprint of the application.

Figure 4(c) demonstrates the performance improvement that replicating the table yields: at 128 processors, the number of cycles spent in List2 computation is reduced from nearly 9 billion to less than 350 million.

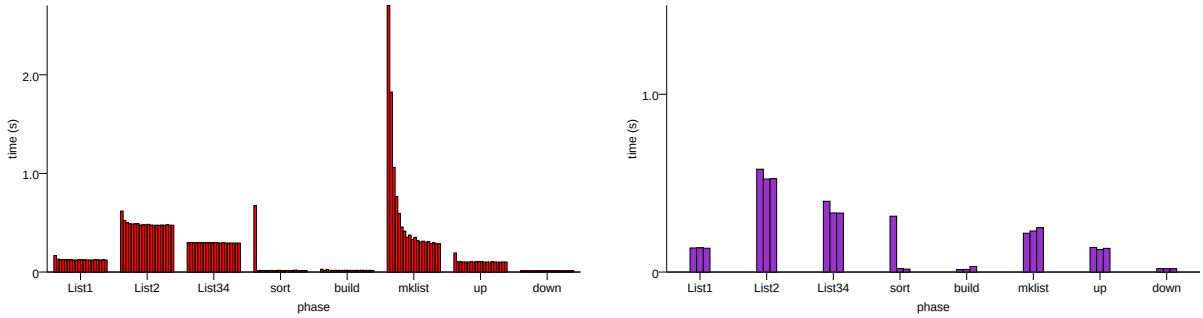
It is interesting to note that we did not run into this problem on the HP Superdome because it uses a different algorithm for distributing memory. Rather than first-touch at a page granularity, the HP distributes memory round-robin at a *block* granularity. While also a NUMA architecture (using a directory-based cache coherence protocol), a very fast *average* case access provides the feel of an SMP. However, maintaining acceptable average case behavior requires limiting the machine size. The largest Superdome currently available has 128 processors.

Lesson: on the Altix, programmers must be very aware of how and where they allocate data.



(a) (b) (c)

Figure 4: (a) OpenMP cycle counts, for each algorithmic step, on Ram: List2 interaction computation clearly dominates. (b) L3 miss counts: note correlation between List2 misses and cycles. (c) Cycle count results *after* replicating a large table so that each processing element has a copy in local memory: List2 interactions no longer dominate.



(a) (b)

Figure 5: Multiple iterations of MPI version: (a) 20 iterations on Altix, (b) 3 iterations on Teragrid.

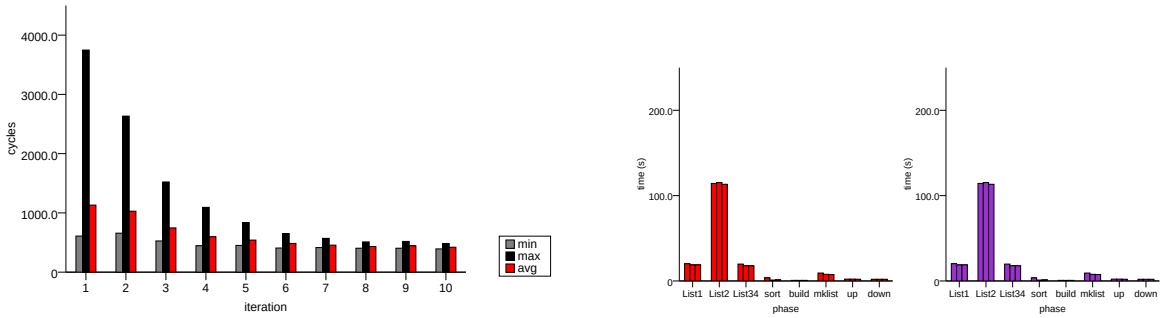
4.3.2 Altix MPI Performance

Our initial talks with an SGI representative concerning the performance of the MPI version uncovered a problem with our understanding of an argument to the *dplace* command, which ensures that a process to processor affinity is established and maintained. However, while fixing the command line did have a beneficial effect on the overall runtime, we were still left with a substantial (6X) gap in the time to perform *mklists* between the Altix and the Teragrid.

Recall that *mklists* is the list construction phase in which the tree is traversed to determine, for each box, all the boxes it needs to interact with during the interaction phase.

To test the hypothesis that the problem is related to startup overheads on the Altix that would amortized over a longer running time, we performed some experiments in which we ran the simulation for multiple iterations. Currently our FMM codes simply measure the potential at the end of a single timestep and do not convert that potential into a force with which to advance bodies. Hence, running multiple iterations is not physically meaningful. However, it does provide insight into the performance of *mklists* as demonstrated in Figures 5(a) and 5(b). While there is little difference between the iterations on the Teragrid, we notice a sharp decrease over the first few iterations on the Altix. Furthermore, as shown in Figure 6(a), an initial very large gap between the minimum number of cycles that a processor spends making the lists and the maximum, soon converges over the course of multiple iterations.

Further evidence that the Altix's poor performance in our initial experiment is attributable to startup overheads that are amortized over time is provided by Figure 6(b). This data demonstrates that if we increase the size of the data set by a factor of 200, thereby increasing the volume of communicated data



(a) (b)

Figure 6: (a) Convergence of min/max/avg on Altix. (b) Three iterations of runs using huge data sets – where the number of bodies per processor is increase by a factor of 200 – on Altix (left) and Teragrid (right).

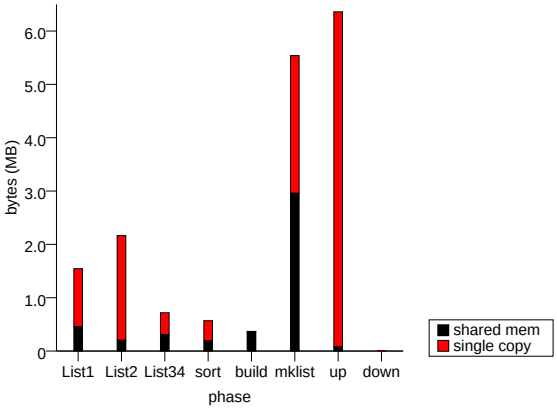


Figure 7: Bytes communicated by MPI version.

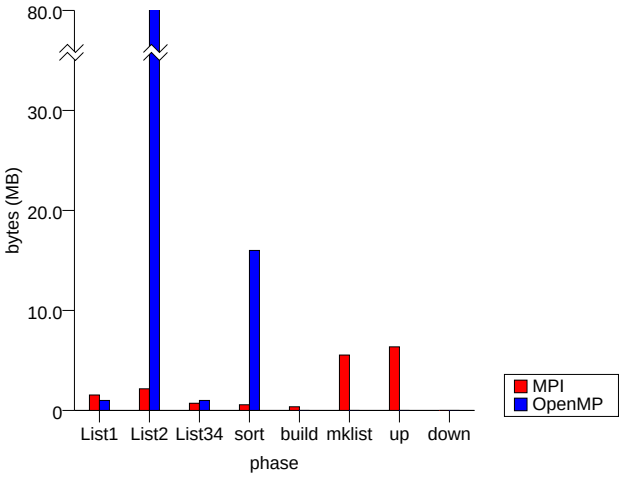


Figure 8: Rough comparison of bytes communicated by MPI vs OMP (128 procs).

by a similar factor, there is little difference between the *mklists* time on the two architectures, even when running for just a single iteration.

Accepting that the problem is some kind of initialization overhead, we are still left with the following nagging question: why are they only in visible in the *mklists* step of the algorithm?

One clue might be related to the way that message passing is implemented on the Altix. By default, messages are copied twice through shared memory: once out of the sender's memory into a temporary buffer and then again into the receiver's memory. An optimization which removes the copy to the temporary buffer is enabled by default for some collective communication calls, including MPI_Alltoall, for *large* messages.

Figure 7 plots data provided by several functions unique to the SGI MPI implementation which provide information about message types and volume of communicated. The figure provides us with two pieces of information which may be relevant to understanding the initialization overheads in the *mklists* step: first, there is a large volume of the less efficient double-copied messages in the *mklists* phase. Second, since *mklists* uses MPI_Alltoall exclusively, that implies a large volume of *small* messages in this phase.

One or both of those pieces of information

4.4 Extended Results

Having gained an understanding of the initial performance problems on the Altix, we now shift our focus to the original purpose of the experiments: to test our hypothesis that while shared memory maintained by hardware-based coherence mechanisms is more effi-

cient for fine-grained tree-code, message passing is a more efficient match for the coarse-grained interaction code.

One measure by which to compare the efficiency of two parallel implementations is the amount of communicated data to complete a given task. In Section 4.4.1 we make a preliminary comparison of the amount communication (in bytes) required by the two platforms for 128 processors.

Another measure is, of course, scalability. In Section 4.4.2 we examine runtime trends as we increase the number of processors.

4.4.1 Communication Estimates

In order to measure communication efficiency we would like to be able to measure the actual number of bytes communicated across the network over a given period of time. Unfortunately, we are not currently aware of how to perform this measurement precisely on on the Altix.

However, we do have access to data which to a certain degree approximates communication for both versions.

For the OpenMP version, as we have already seen, we can access hardware counters to determine L3 misses. Since not every L3 miss results in communication, as some percentage will be satisfied from memory on the node local to the processor, this measure overestimates communication. However, if the number of the L3 misses is low we can certainly infer that the number of misses that cause communication is low; and, as the number of L3 misses grows, the likelihood that the misses require communication, given that some misses do require communication, also grows.

Since the transfer size for cache misses is the block size, we can estimate the number of bytes communicated by multiplying the number of L3 misses by the block size.

We have also already seen, in the previous section, that SGI provides a mechanism for obtaining the number of bytes communicated by the MPI programs. Unfortunately, this method also has a shortcoming, in that it does not distinguish, for collective communication calls, between bytes sent to other processors and bytes "sent" to self. Hence, this measure also overestimates the amount of data communicated.

Bearing these caveats in mind, we can turn to Figure 8 which compares bytes communicated by the two versions of FMM at 128 processors. Note that in the fine-grain steps, most notably *build* and *mklists*, we see that the OpenMP version requires the communication relatively few bytes of data compared to the large number required by the MPI version.

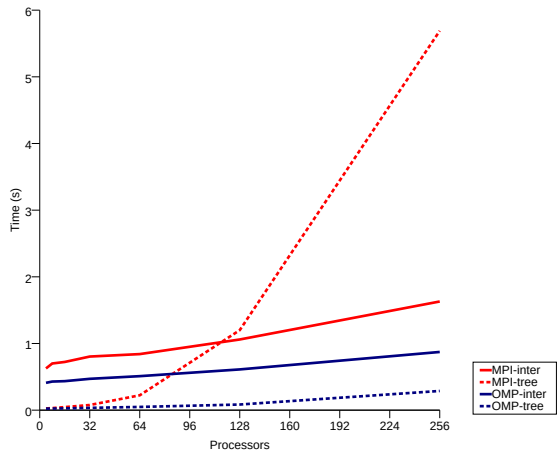


Figure 9: Results for all 256 processors of ORNL Altix (3rd iteration).

On the other hand, the OpenMP versions of the interaction phase, particularly the *List2* phase, appear to transfer much more data than the MPI version. While not currently a problem given the communication bandwidth of the machine, it could present a problem as the number of processors and/or bodies per processor grows.

This data supports our original hypothesis that while shared memory would more efficiently communicate non-local data in the tree-code steps, message passing would be more efficient for the interactions.

4.4.2 Scalability Trends

In Figure 9 we show results from a run in which we were given access to all 256 processors of Ram, the Altix at ORNL. All the way up to 256 processors, the shared memory version outperforms the MPI version, even on the interaction code. Since the gap between the two is fairly steady as the number of processors grows, we might attribute this to greater indexing efficiency in the shared memory version: recall that we use a hash table to represent the octtree in the MPI version which implies a hash lookup for every index into the tree rather than the simple pointer dereference enabled by globally shared memory.

But what happens to performance as we keep increasing processor counts? To help answer this question we were fortunate that our contact at SGI pro-

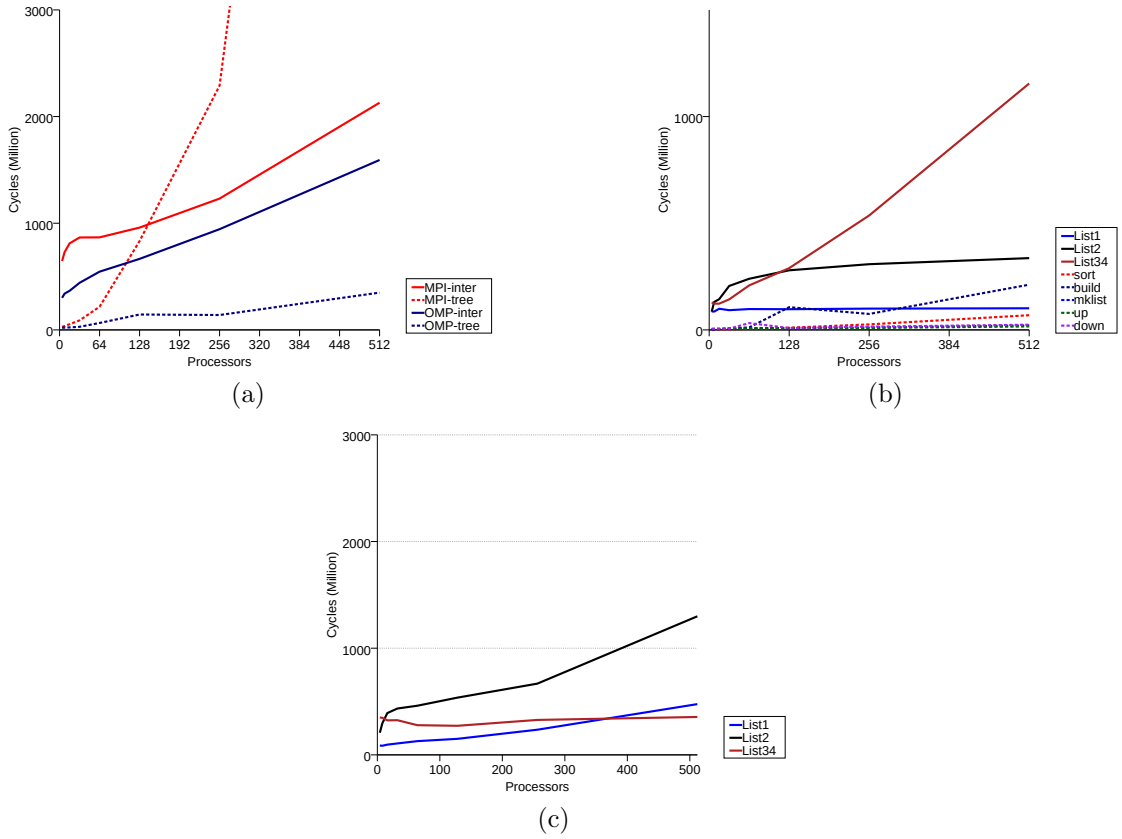


Figure 10: Results for 512 processor Altix Bx2 at SGI (3rd iteration): overview (a), and detailed: (b) OpenMP (c) MPI (tree-code steps removed).

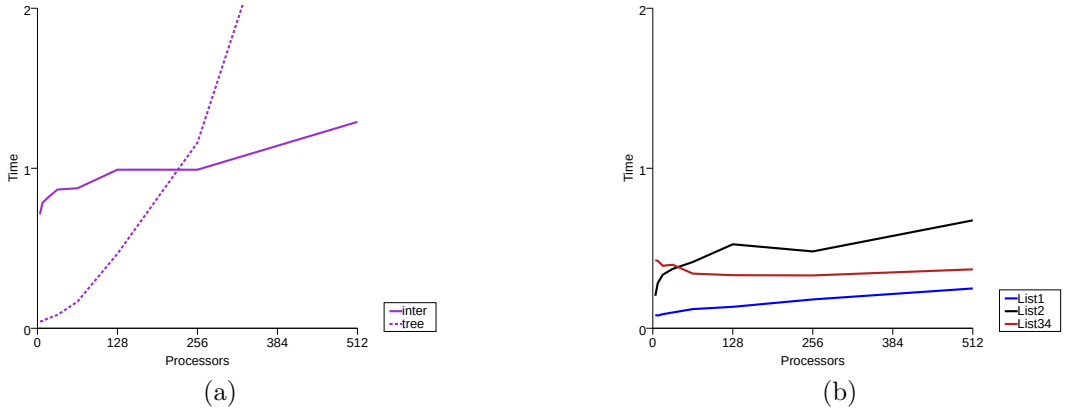


Figure 11: Results for 512 Teragrid processors (3rd iteration): (a) overview, and (b) detailed (tree-code steps removed).

vided us with data from runs on a 512 processor Altix Bx2 at SGI.

Though the processor cycle time is the same as on Ram (1.5GHz), and despite a smaller L3 cache (4MB vs 6MB), both versions of the application generally run faster on the Bx2, particularly at lower processor counts. This performance difference is apparently due to greater bandwidth between nodes provided by the NUMAlink 4 interconnect. However, as we add processors, performance is not quite as steady as on Ram, perhaps due to the smaller cache.

Figure 10(a) shows the summary results for the Bx2 at SGI, while Figure 10(b) and (c) show results for each algorithmic step for OpenMP and MPI respectively.

Directing our attention first to the OpenMP summary results, we see a steeper slope than optimal for both curves in the summary graph. From the per-algorithmic-step results we can attribute most of the growth to two phases: the *List34* interaction phase and the *build* tree-code phase. It is likely no coincidence that both phases make use of locks to synchronize data access.

As described in 3, we have already done significant work to try to reduce the effect of synchronization in the *build* phase. It appears that we need to find a more structured means of completing the global tree that requires even less synchronization.

Locks are required in the *List34* phase due to potential non-local writes: recall that List34 is actually two symmetric lists, such that if Box A is on Box B's List3 then Box B is on A's List4. To simplify the list construction phase, only one end of this dual transaction is recorded in a list (List34) and both computations are taken care of at the same time in the *List34* interaction phase. As a result, the computation may result in writes to non-local data since Box A may be located on a different processor than Box B.

The MPI version takes advantage of the fact that each write is only an update to a value which is not actually used until the phase is over, and turns a collection of non-local writes into a single post-computation reductions. A similar strategy would obviate the need for the locks in the OpenMP program, though potentially at the expense of significantly complicating the code.

Turning our attention to the MPI results: strikingly, despite the OpenMP problems with *List34* phase locks, *and* the MPI version's apparently substantial advantage in terms of communication requirements for the *List2* interaction computations, Figure 10(a) shows MPI interaction performance consistently lagging OpenMP. The per-algorithmic-step results point to the *List2* interactions as the primary source of the problem.

Comparing these results with 512 processor results for the Teragrid in Figure 11(a) and (b), we see that, while the SGI curve may be slightly steeper than on the Teragrid, there is an equivalent upward trend.

One potential explanation for this trend may be that we are hitting the limits of our current inspector-executor strategy, as described in Section 3. Recall that currently every processor gathers the non-local requests of every other processor to determine whether it owns the data requested. While simple to implement, this all-to-all approach clearly does not take full advantage of data locality inherent in the algorithm.

Another factor may be that there is some communication involved in redistributing data in order to load balance the interaction computations in the MPI version. As noted earlier, we are not currently load balancing the OpenMP interactions.

5 Conclusions

We described experiments with two implementations of the Fast Multipole Method application – one using the shared memory programming model (OpenMP), the other the message passing model (MPI) – on the SGI Altix 3700 at Oak Ridge National Laboratory. The purpose of the experiments was to test our hypothesis that the fine-grained communication requirements of the FMM tree-code phases would favor the OpenMP implementation, while the course-grained communication would favor the gather/scatter approach of the message passing implementation.

In the first part of our discussion we explained the Altix's poor performance in our initial results. For both shared memory and message passing, it is important to be aware of the NUMA architecture, and the careful placement of data that it requires. The good news is that taking that extra care can yield good results.

Having addressed the performance problems on the Altix, in the second part of the discussion we turned our attention understanding implications of the data on our hypothesis.

We found that, as we expected, the tree-code is much better suited to shared memory programming

paradigm, both from a communication and an actual performance standpoint. On the other hand, contrary to our hypothesis, and though communication requirements appear to favor the gather/scatter approach to interaction computation taken by the MPI version, the OpenMP version outperforms the message passing version. It would appear that overheads in the inspector-executor paradigm, or at least in our particular implementation of it, outweigh the benefits of reduced communication requirements.

Coda: User Controllable Coherence. Finally, we would like to point out that one of the motivations for this work is to demonstrate the value of a hardware architecture that combines the benefits of hardware-based cache-coherence with those of a distributed memory platform.

One proposal for such an architecture is "User-Controllable Coherence" [6]. User-controllable coherence provides mechanisms which allow users and/or compilers to turn on and off cache coherence at a cache-block granularity using new flavors of load and store instructions. The idea is to use the efficient cache-coherence mechanisms to get data where it is needed, but to not be restricted by either the size of the cache or unnecessary synchronization.

The FMM is a particularly good showcase for user-controllable coherence with its mixture of fine-grained and coarse-grained communication. In future work we plan to show that, while enabling the same code and performance for the fine-grained accesses in the tree-code phases, appropriate use of mechanisms provided by user-controllable coherence could alleviate problems associated with coarse-grained performance with minimal effort.

6 Acknowledgments

The authors wish to thank Kevin McMahon of SGI for his invaluable help in conducting and understanding the results of our experiments. We would also like to thank the University of Kentucky for granting us access to the HP Superdome, NCSA for allowing us access to the Teragrid, and the consultants at ORNL for their help with Ram.

References

[1] F. Allen et al. Blue Gene: A vision for protein science using a petaflop computer. *IBM Systems Journal*, 2001.

[2] C. R. Anderson. An implementation of the fast multipole method without multipoles. *SIAM J. Sci. Stat. Comput*, 13(4):923–947, July 1992.

[3] J. Barnes and P. Hut. A hierarchical $o(n \log n)$ force calculation algorithm. *Nature*, 324:446–449, 1986.

[4] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Physics*, 73:325–348, 1987.

[5] Y. C. Hu, S. L. Johnsson, and S.-H. Teng. High Performance Fortran for highly irregular problems. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–24, Las Vegas, NV, June 1997.

[6] C. McCurdy and C. Fischer. User-Controllable Coherence for High Performance Shared Memory Multiprocessors. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, 2003.

[7] C. McCurdy and J. Mellor-Crummey. An Evaluation of Computing Paradigms for N-body Simulations on Distributed Memory Architectures. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, 1999.

[8] H. Sagan. *Space-Filling Curves*. Springer-Verlag, New York, NY, 1994.

[9] M. Warren and J. Salmon. A parallel hashed-octtree n-body algorithm. In *Proceedings of Supercomputing '93*, Portland, OR, Nov. 1993.

[10] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.