

# Object-Oriented Parallel Barnes-Hut

David Garmire  
<[strive@cs.berkeley.edu](mailto:strive@cs.berkeley.edu)>

Emil Ong  
<[emilong@cs.berkeley.edu](mailto:emilong@cs.berkeley.edu)>

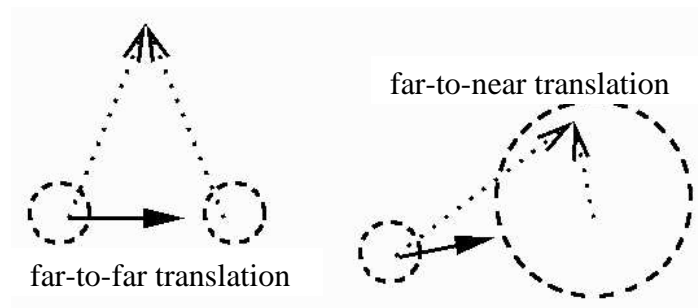
January 3, 2002

## ABSTRACT

*The naïve (NV) implementation of the  $n$ -body problem runs in  $O(n^2)$  where all particle interactions are computed, while the Barnes-Hut (BH) algorithm runs on average  $O(n \log n)$  floating point operations for a given accuracy and fixed distribution of particles. For an rms error of  $10^{-3}$ , BH takes close to  $O(n)$  floating point operations. BH is a hierarchical tree method which presents a difficult problem in effectively coding the parallel solution. Moreover, BH can be used for several different types of applications (e.g. gravity simulations and charged particle simulations). We implement and analyze extensible object-oriented versions in MPI Ruby and Titanium. MPI Ruby, a scripting language, achieves good prototyping capabilities and permits an investigation of parallel schemes for BH. Titanium, a superset of Java 1.0 for parallel architectures, achieves reasonable performance characteristics on different parallel schemes while still maintaining clear, extendable, and portable class structures.*

## 1 Introduction

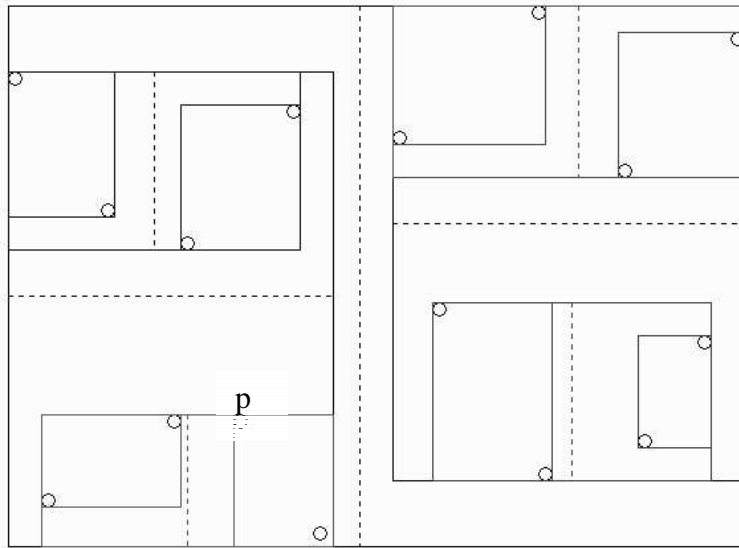
Let an array of  $n$  particles be given,  $P_i = (m \text{ (mass)}, c \text{ (charge (or mass in the case of gravity))}, x \text{ (position)}, v \text{ (velocity)}, f \text{ (force)})$ ,  $i \in (0, n - 1)$ , with a force of one particle on another prescribed by an equation,  $f_i = F(c_j, x_i, x_j)$ . A naïve approach to simulating the motion of the particles over time calculates the forces of all particles on all other particles (an obviously  $O(n^2)$  set of operations). A significant improvement is found if the system obeys the solution of a Laplacian equation as do Coulombic and gravitational potentials. (Note that the force is the gradient of the potential.) Essentially, a set of particles within a sphere can be approximated by a set of orthonormal polynomials (the more terms,  $p$ , the more accurate the approximation), such that either as you get farther away from the sphere you get a better approximation (a multipole expansion) or as you get closer to the center of the sphere you get a better approximation (a Taylor series expansion). These approximations can be translated to get the potential at different locations or go from a far approximation to the near field effects of that approximation in a local region. For more in depth information on the mathematics, follow David Blackston's thesis [1].



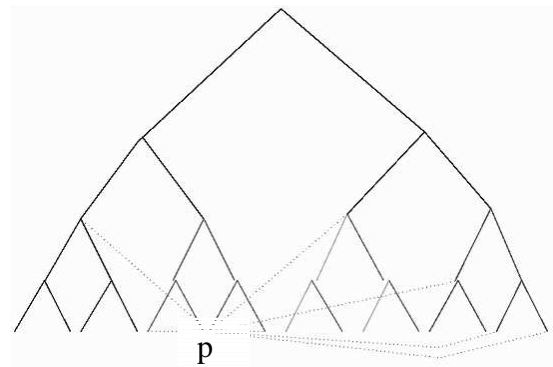
**Figure 1: Dark Arrow is Translation. Dashed Arrows are Evaluation at Point**

The Barnes-Hut (BH) algorithm [2] decomposes space into a hierarchical tree (could be a quad tree or octree or the one described below). The accuracy of the simulation can be bound by only evaluating the force approximation of a group of particles on an individual particle, if the circle that bounds that group is outside a certain distance away proportionate to its size. (Otherwise the group is subdivided into smaller groups.) The equation generally used is if the distance of the particle to the center of mass of the group times a tuning parameter,  $\theta$ , is greater than the bounding radius of that group then the particle is well-separated from that group. As  $\theta$  becomes closer to 0, the simulation approaches NV. As  $\theta$  becomes closer to 1, the simulation relies more heavily on the number of terms,  $p$ , used in the field approximations and approaches  $O(n)$  interactions.

A way of building the BH tree, is to choose the longest axis of the current bounding box and divide it in half and repeat on each half. The advantage over the quad tree approach is that arbitrary distributions of particles can be handled (you do not have sets of boxes with no particles in them). The downside is that it requires a bit more computation. It takes  $O(n \log n)$  to build the tree (at each stage finding the bounding box), but the computation is minor in comparison with the approximation translations. At each stage the far field approximations for the left and right children can be translated to the center of the parent box and added to find the far field approximation of the parent. The actual run of BH requires that for each leaf of the tree, recurse from the root and sum effects of well-separated groups. Supposing a particle,  $p$ , is being evaluated according to BH as depicted in **Figure 2**. For some arbitrary  $\theta$ , the far-to-near translations are depicted in **Figure 3**.



**Figure 3: Example Set of Particles**



**Figure 2: far-to-near Translations for P**

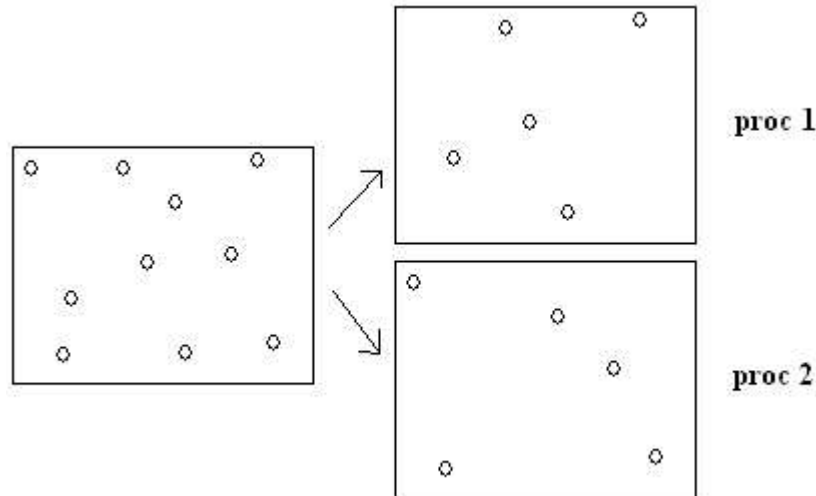
It should be noted that while for a given accuracy, BH runs in  $O(n \log n)$  time, the constants are quite high (initializing the approximations and the tree structure). A general technique is to cut off leaves when they contain fewer than a certain number of particles, referred to here as *LeafSize*. NV is then run within that set of particles. The tolerance for

well-separatedness also has to change to the maximum of the leafsize's radius and the comparison group's radius.

## 2 Parallel Schemes

We consider two ways to break down the computation among processors. Since the particles are stored in an array generally to make copying between processors more efficient, an intuitive way to divide the particles is by their location in the array. BH then can be run on each processor's tree independently and then instead of starting from the root of some global tree, start from the root of every other processor's tree for the far-to-near translations. While this method obviously has a quick initialization time and is easy to understand under the SPIMD paradigm, the bounding boxes may overlap between processors so more far-to-near translations are required than a method that would account for spatial locality. In fact in the worst case scenario, the amount of computation for a processor would be  $O(\frac{n \log(n)}{p})$  in problem size \*  $p$  other trees to look at for each

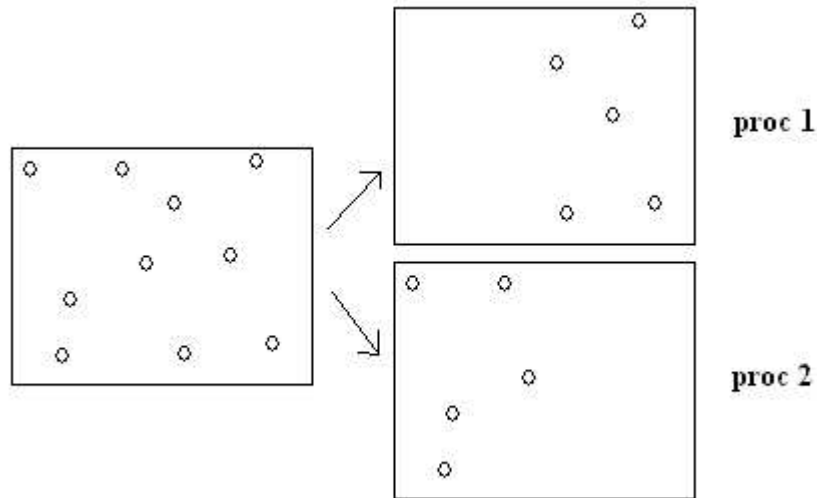
particle which would be  $O(n \log(n))$  time per processor (if every tree overlapped completely with every other one, which is unlikely). But in such a case, as more far to near translations are chosen, the approximation will become more accurate. To maintain the same accuracy,  $\theta$  may be increased and then the computation time will again decrease. It becomes more complicated to figure out the correct  $\theta$  and # polynomial terms to achieve a desired rms accuracy.



**Figure 4: Method 1 (Particle Processor Division)**

Another method is for each processor to keep a global tree, and assign itself to a branch of that tree (so all of its particles will be within a bounding box). Dynamic balancing can be imposed by counting the # of particles in the current branch of the tree and see if it is within a range of the average # particles per process given an even split. This approach is quite difficult in practice to implement as we found while prototyping in MPI Ruby. The unique design features of Titanium allowed us to make an implementation without too

much difficulty. The run-time should approach an asymptotically better bound than the previous method, as a given processor will generally not have to communicate with processors that are responsible for boxes far away. Another benefit is that the correct  $\theta$  and # polynomial terms to achieve a desired rms accuracy is the same as in the serial algorithm.



**Figure 5: Method 2 (Tree Processor Division)**

### 3 Object-Oriented Design

A concern of this project was to generate code that could be extended to physical systems of higher dimensions and different force considerations, as well as alternate tree building algorithms (e.g. octrees) and approximation techniques. The natural way of carrying out the division of code labor would be to treat each of these items separately from more basic to more general: Cartesian vectors, polar vectors, complex numbers, particles, approximations, and trees. In C, these can be written as structs and then the functions that act on them can be separated into libraries.

```
typedef struct vector_struct {
    double x, y, z;
} vector_t, *vector_ptr;

typedef struct polar_struct {
    double r, t, p;
} polar_t, *polar_ptr;

typedef struct particle_struct {
    double c;
    vector_t force;
    vector_t pos;
} particle_t, *particle_ptr;

typedef struct complex_struct {
    double r, i;
} complex_t, *complex_ptr;
```

```

typedef struct approx_struct {
    int          p;
    complex_t    *terms;
} approx_t, *approx_ptr;

typedef struct tree_struct {
    particle_ptr *particles;    /* particles inside box */
    int          num;           /* num particles */
    approx_ptr   approx;        /* approximation */
    vector_t     start;         /* smaller box coords */
    vector_t     end;           /* larger box coords */
    vector_t     center;        /* center box coords */
    double       radius;
    double       size;
    struct tree_struct *left, *right;
} tree_t, *tree_ptr;

```

Unfortunately, this breakdown makes it hard to understand what acts on what exactly. It is nice to have these data structures be the mathematical and physical objects that they represent. They can then expose the things that can operate on them and transform them.

Also this coding paradigm is hard to extend without changing the entire data structures and libraries that use them. In an object-oriented paradigm, the `approx` object can be extended and the `tree` object would not have to change. The same with the underlying `particles` object – it can be changed to another set of dimensions with only changing a bit of the `approx` object so that it knows the ways to generate terms for these added dimensions. An outline of this class structure in Java is depicted below (and is similar in MPI Ruby).

```

public class Complex {
    public double r;
    public double i;
    public Complex()
    public Complex(double r, double i)
    public void mul(Complex a, Complex b)
}

public class Cart3d {
    public double x, y, z;
    public Cart3d()
    public Cart3d(double x, double y, double z)
    public void subtract(Cart3d a, Cart3d b)
    public double magnitude()
    public double distance(Cart3d a)
    public void toPolar3d(Polar3d p3d)
}

public class Polar3d {
    public double r, t, p;
    public Polar3d()
    public Polar3d(double r, double t, double p)
    public void toCart3d(Cart3d c3d)
    public String toString() { return ("+r+", "+t+", "+p+"); }
}

```

```

public class Particle3d {
    public double c;
    public Cart3d f;
    public Cart3d pos;
    public Particle3d();
    public Particle3d(double c, double x, double y, double z)
}

public class Approx {
    public int p;
    public int p2;
    public Complex terms[];
    public Approx(int p)
    public void Y(double theta, double phi)
    public void fillApproxNear(double c, Polar3d p3d)
    public void fillApproxFar(double c, Polar3d p3d)
    public void addApprox(Approx a)
    public void scalApprox(double [] s)
    public void tFN(Approx a, Polar3d p3d)
    public void tFF(Approx a, Polar3d p3d)
}

public class Tree {
    public Particle3d [] particles; /* particles inside box */
    public Approx a; /* approximation */
    public Cart3d start; /* smaller box coords */
    public Cart3d end; /* larger box coords */
    public Cart3d center; /* center box coords */
    public double radius;
    public double size;
    public Tree left;
    public Tree right;
    public static int LeafSize;

    public Tree()

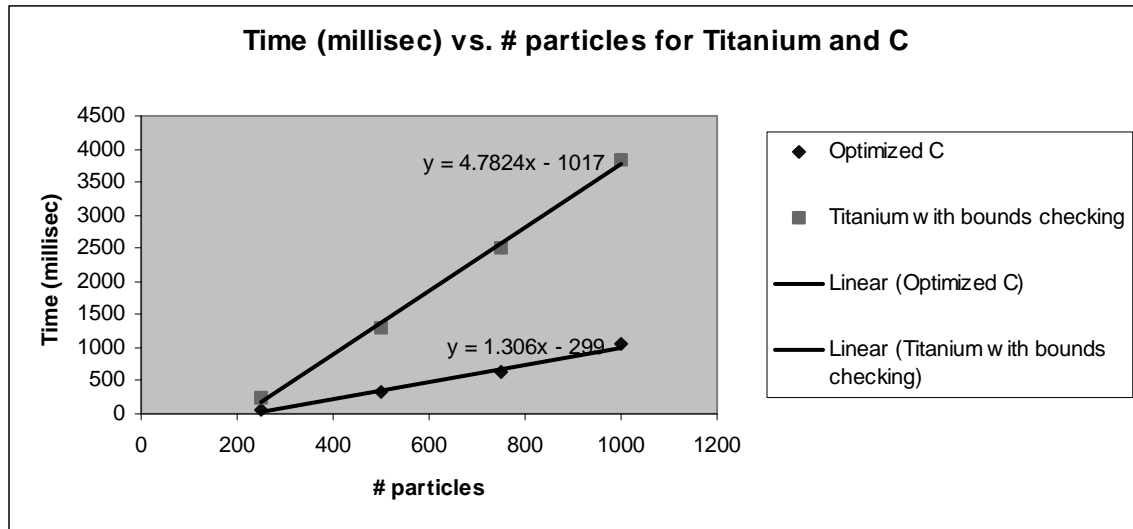
    public static void calculateForces(Particle3d [] dest)
    public static void calculateForces(Particle3d [] dest, Particle3d [] src)
    public static Tree buildTree(Particle3d [] p3d, int p)
    public static void evalGrad(Particle3d [] p3d, Tree t)
    public static void computeWellSepForces(Tree leaf, Tree r, double theta)
    public static void compute(Tree r, Tree root, double theta)
}

```

Note that the 3d suffix at the end is not necessary and in future work should not be included out-of-hand. We could make an abstract class and then implement this class as a 3d class.

#### 4 Serial C, MPI Ruby, and Titanium Analysis

Compiled with bounds checking on millennium, the Titanium code takes about 3.6 times longer than optimized C code (with flattened code calling structure). Considering the code was written with object-orientedness in mind, this performance is certainly reasonable and could be improved by removing some of the previous considerations. The MPI Ruby serial code takes about 2.5 seconds for 100 particles on a faster machine (Pentium III 850 MHz) and is not worth nor meant to be included as a performance contender.



## 5 MPI Ruby Analysis

Our first attempt to put the BH code into Ruby was a translation of the original serial C code to serial Ruby code. The code was very similar in structure to the original except that it had a much more object oriented flavor. Vectors, approximations, and trees were all obvious objects to be manipulated in this code. The use of these objects made the code more readable (e.g. operator overloading provided a more natural look to the code) and more extensible (e.g. higher dimensions or different types of forces would take little modification). Ruby also provides automatic memory management which we would discover to be a curse as well as a blessing in our performance analysis.

We ran the serial Ruby code and the original C code on the same data and the results were very disappointing for Ruby. The C code was able to dispense with a 1000 particle problem in approximately 1 second while the same input took the Ruby version more than 500 seconds. The cause for this slowdown is two fold – 1) Ruby is interpreted and therefore naturally slower than C (with a noticeable startup time as well) and 2) Ruby’s automatic garbage collection turns out to be very inefficient on even moderately large problems such this 1000 particle input. Rudimentary profiling revealed that the garbage collector was responsible for at least 18%-20% of the time spent. At first pass, we assumed that reason 1) was the primary cause for the slowdown in the Ruby code, but when we parallelized the code, we found that the culprit was truly the garbage collector.

MPI Ruby is an extension that allows Ruby programs to access MPI functionality. It is more than a thin layer however because Ruby has no concept of buffers which MPI uses extensively. Instead, MPI Ruby does automatic marshaling and unmarshaling of Ruby objects into buffers. This allows the Ruby programmer to avoid buffers and data counts altogether as well as any notion of MPI datatypes. Because of these unique abilities, we were able to parallelize the serial Ruby program by adding less than 20 lines of code total.

```

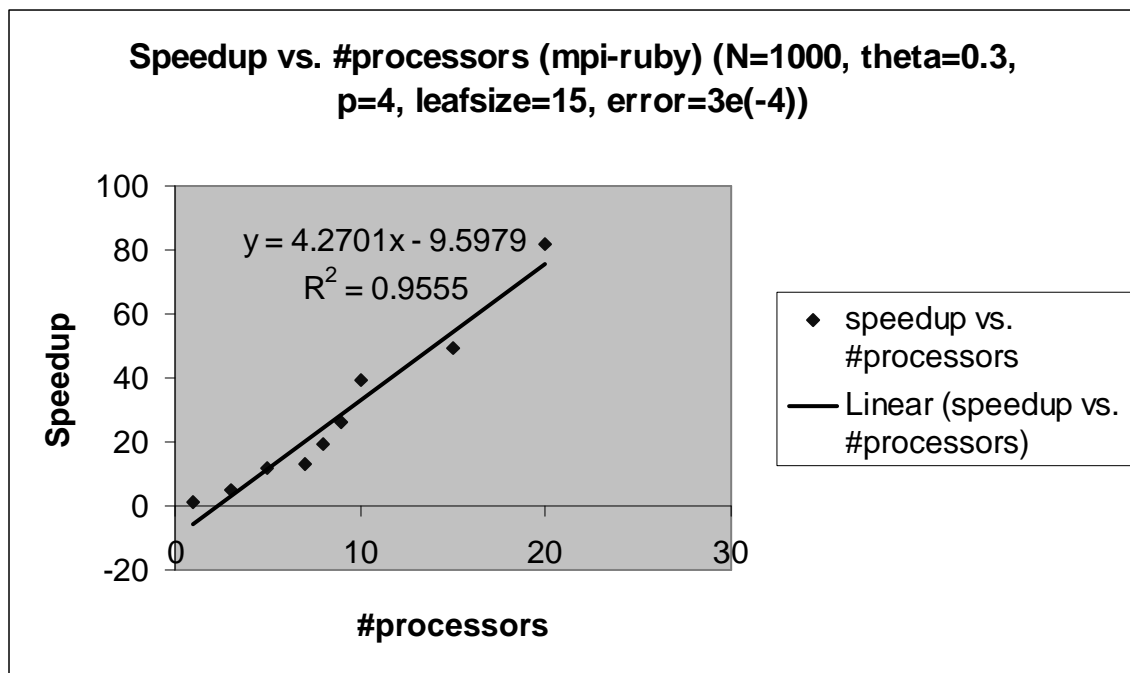
1 computeBH(myTree, myTree, theta)
2 sendTree = myTree
3 ($nprocs - 1).times { |i|
4   recvTree, status = MPI::Comm::WORLD.sendrecv(sendTree,
                                                    ($rank + 1) % $nprocs, 0,
                                                    ($rank - 1) % $nprocs, 0)
5   computeBH(myTree, recvTree, theta)
6   sendTree = recvTree
7 }

```

The code shown is the MPI Ruby code that does the entirety of the parallel computation. This version of the code is parallelized by index rather than spatial locality. Each processor builds their own tree (`myTree`) and computes the tree's interactions with itself (line 1). Then each processor does a cyclic send/receive computation with each of the other processors trees (lines 3-7). Notice that in the MPI `sendrecv()` call, the `sendTree` is simply treated as an object and sent without any explicit packing or mention of buffers or datatypes. This convenience allows the programmer to write what looks almost like pseudocode that nonetheless works as expected.

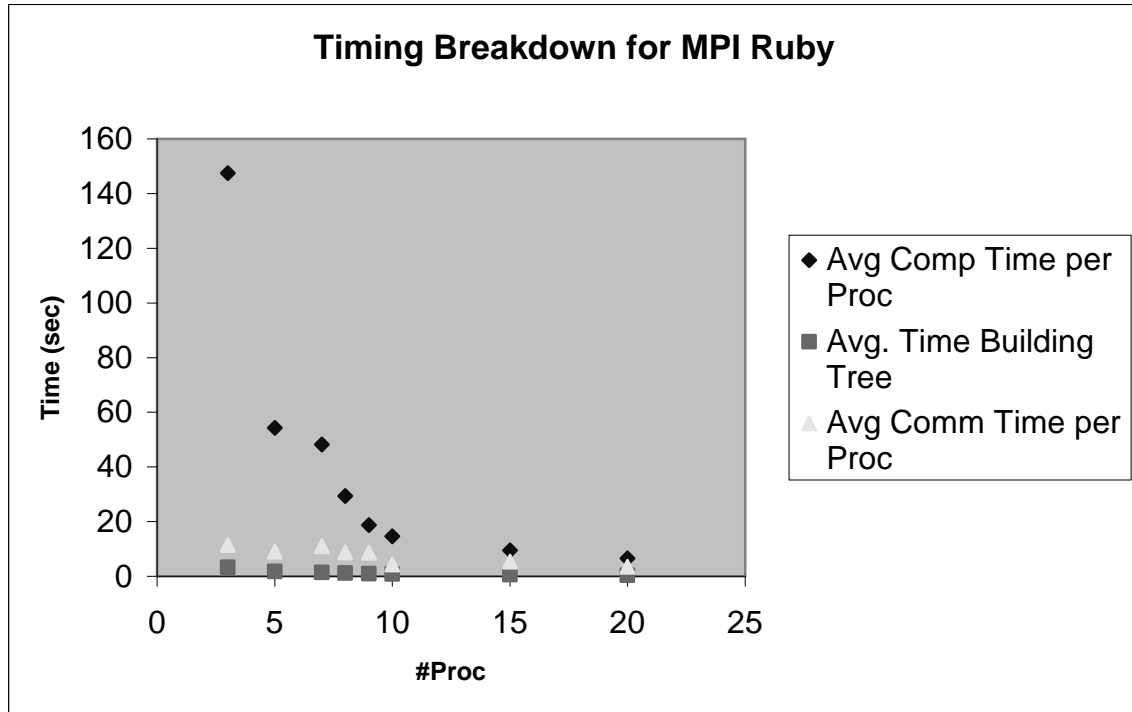
#### Parallel Results

Several timing tests were performed on the parallel code. In the first test we ran, we maintained the problem size (1000 particles) and increased the number of processors. We saw dramatic, almost unbelievable, speedup.



This almost linear speedup (with best-fit coefficient of 4.27!) was in fact due to the fact that the amount of data on each processor was decreasing and therefore leading to less

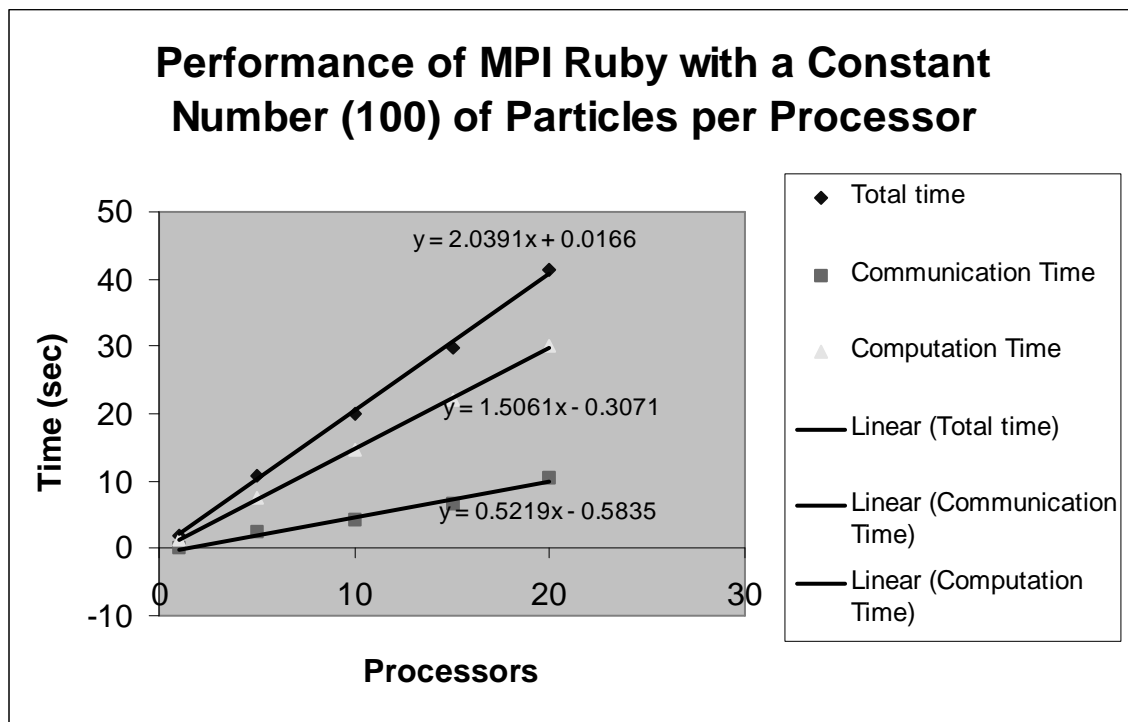
time spent in the garbage collector. The next graph illustrates that while the amount of work on each processor was decreasing linearly as the number of processors increased, the average computation time was decreasing much more dramatically, implying external forces. Profiling revealed the GC.



In order to obtain numbers that reflected the algorithm's properties rather than Ruby's, we ran additional tests where we maintained a constant number of particles (100) on each processor while increasing the number of processors. Under these conditions, the garbage collector was almost inactive, thereby giving a better indication of the performance of the algorithm. We discovered that the index method of parallelization gives reasonable results, but is less than spectacular. In particular, we found that the line of best-fit for the increase in time taken by the serial version while increasing the number of particles (with the GC accounted for) has a slope of approximately 2.7 while the slope of the line of best-fit in this test has a slope of approximately 2.0. In other words, we achieved only a modest speedup of 26% after throwing many processors at the problem.

In almost as much time as we took to parallelize the serial version of the Ruby code (~5 minutes), we also realized that implementing the spatially parallel version would be almost as expensive if not more so in terms of communication. This problem is due to the lack of one-sided operations in MPI-1, the version of MPI on which MPI Ruby is based.

## Performance of MPI Ruby with a Constant Number (100) of Particles per Processor



## 6 Titanium Analysis

The Titanium code was originally converted from Java taking ideas from the serial C code. The hope in generating parallelism with Titanium was to take advantage of the global memory layout, having the fact that a pointer may be off-processor transparent to the calling functions.

To create the first parallel scheme, very little was needed. A few barriers were necessary, as was exchanging the built roots, and looping through the other processor roots.

```
Ti.barrier();
Tree [1d] single leaves = new Tree[0:Ti.numProcs()-1];
Tree t = Tree.buildTree(myparticles, p);
leaves[Ti.thisProc()] = t;
leaves.exchange(leaves[Ti.thisProc()]);
Ti.barrier();
Tree.compute(t, t, theta);
for(j = Ti.thisProc()+1; j < Ti.numProcs(); j++) {
    Tree.compute(t, leaves[j], theta);
}
for(j = 0; j < Ti.thisProc(); j++) {
    Tree.compute(t, leaves[j], theta);
}
```

For the second parallel scheme, more thought was necessary. The processor “leaves” (where each processor takes over the global tree) had to be synchronized after each processor’s tree was built, then the parent tree pointers of each processor’s global tree could reference the off-processor trees transparently.

```

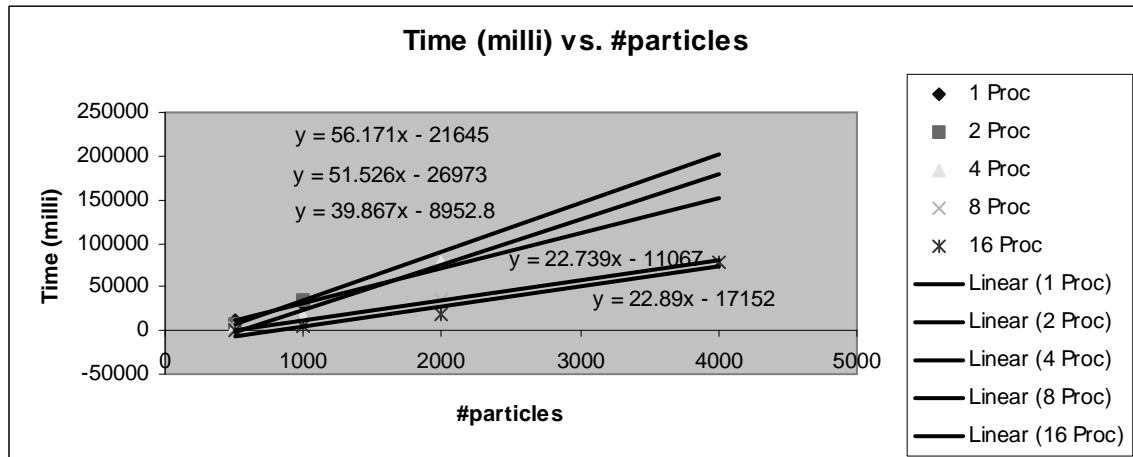
Tree t = Tree.buildTreeLocal(particles, leaves, false, new Integ(0), NperProc, p);
leaves.exchange(leaves[Ti.thisProc()]);
Tree start = Tree.connectTrees(t, leaves, p);
Ti.barrier();
if(start != null) Tree.compute(start, t, theta);

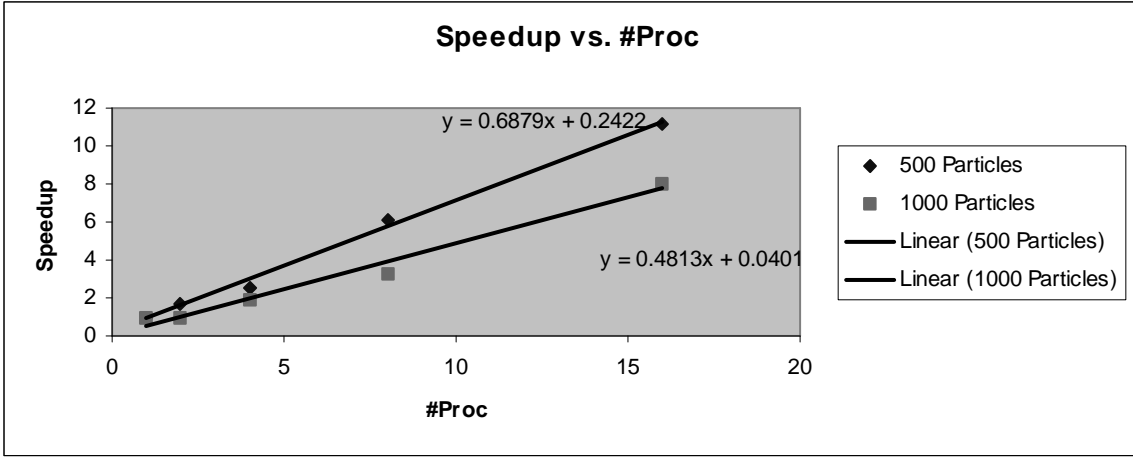
```

More information needed to be sent to building the local tree, so that it would know where to cut off computation for the local processor “leaf”, and also fill in the roots of each processor leaves. Another function needed to be added to the Tree class so that the leaves could be attached to the global tree properly and remaining far tree approximations could be constructed in the global tree.

Timings of the two approaches were conducted on mcurie (a cray T3E at NERSC). The simulations were run using a fixed as possible rms of 0.001 using 4 polynomial terms of approximation, a separation constant,  $\theta$  of 0.33 and a leafsize of 10. Unfortunately, the T3E died at around 2000 particles per processor (reported a bounds error). Our hunch it is because the T3E version of Titanium lacks a garbage collector and so was running out of memory.

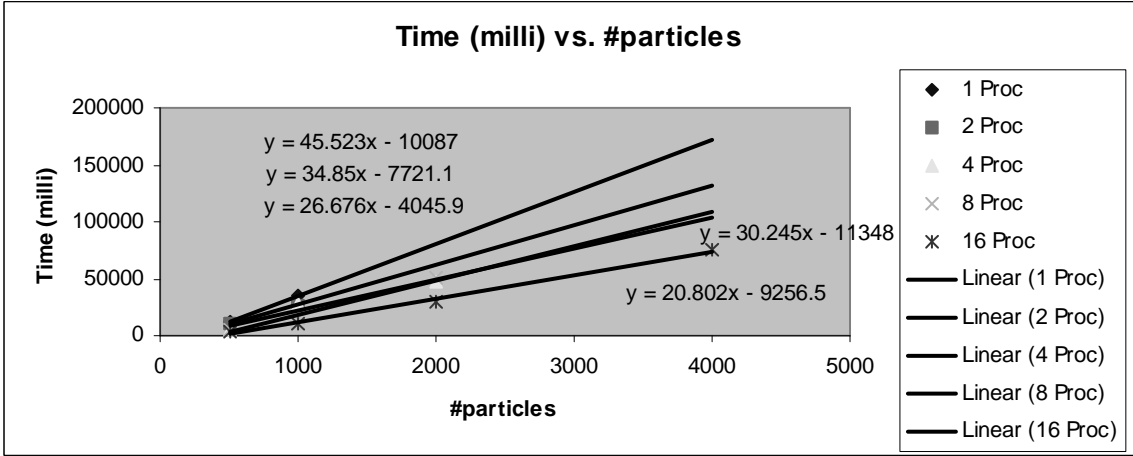
For the first parallel scheme, we found:



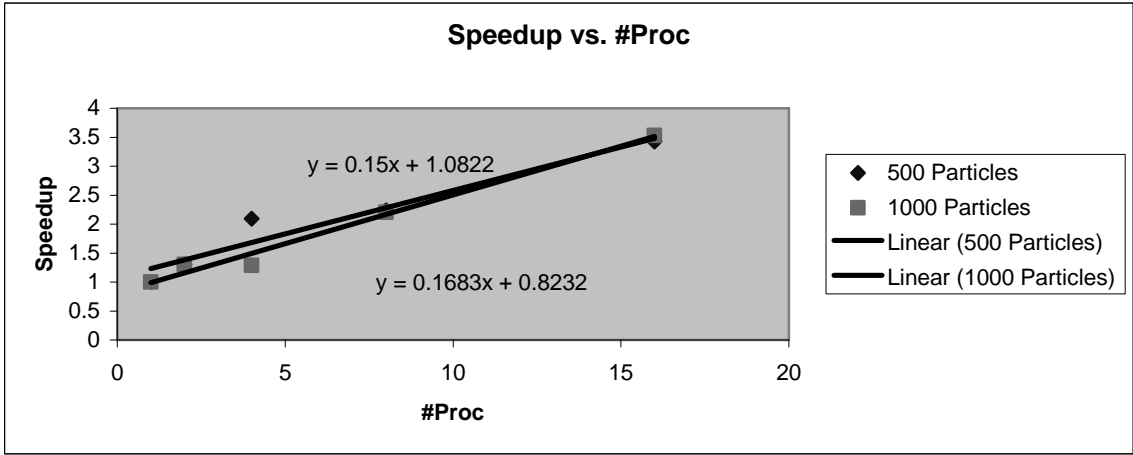


Note that the speedup, although quite good, diminishes as particles are added (because of more interactions being computed most likely).

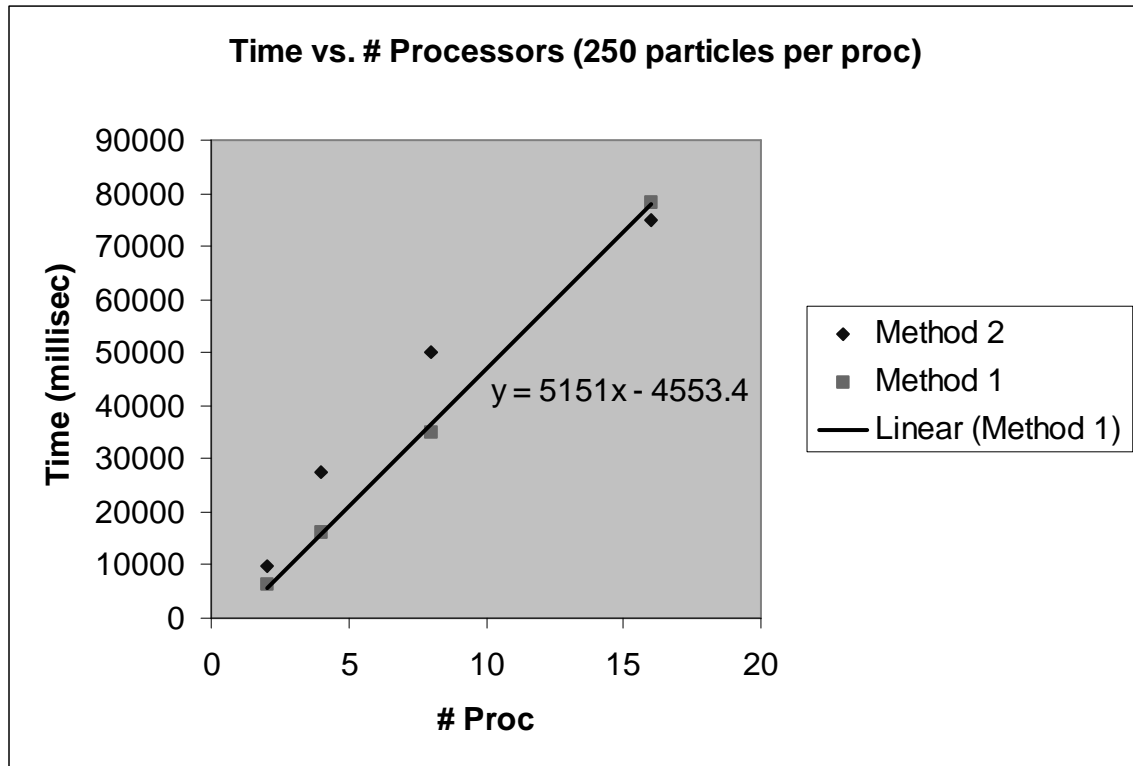
For the second parallel scheme, we found:



Note that these lines are sloped better than the first scheme's, but start out higher.



Also, the speedups behave more as they are supposed to by not diminishing as the number of particles are changed. A better comparison is to fix the # of particles per processor the same and compare times.



Notice that as the number of processors increase, the time it takes rises linearly with respect to the number of processors times the number of particles per processor (or # particles). The slope is actually similar to that of the MPI Ruby code  $((5156/1000) / (250/100)) = 2.06$ . The second method performs worse initially but begins to exceed the performance of the first method at around 15 processors. The hypothesis stands that as the number of processors increases, a scheme that takes advantage of spatial locality performs better in the long run because there are fewer interactions between processors owning particles that are far apart from one another.

## 8 Conclusions

Our experience with MPI Ruby was useful in that we were able to explore implementation in MPI very quickly and see the properties of the indexed parallel algorithm. It was also useful in showing us what was not possible with MPI-1 in respect to this problem. Unfortunately, the problem with the garbage collection leads us to believe that the use of Ruby and MPI Ruby is not yet appropriate for extensive prototyping purposes. The code itself however is very readable, easy to write, and understandable which means that if the implementation of Ruby improved, acceptance might be quick.

Titanium allowed the creation of clean parallel object oriented extendable code. We were able to test two different parallel approaches and see the scalability of each. While not as fast as optimized C code, the Titanium code was within a reasonable range even with bounds checking turned on. Further work should still be done on determining the effects of rms accuracy under the first parallel scheme, but at least we now have a valid Barnes-Hut implementation from which to work.

## 9 References

[1] D. T. Blackston. *Pbody: A Parallel N-body Library*. (Can get from Jim Demmel.) 2000.

[2] S. Pfalzner and P. Gibbon. *Many-Body Tree Methods in Physics*. Cambridge University Press. NY. 1996.

Also would like to note

- Dan Bonachea and the Titanium manual were valuable references.
- The serial C code was originally developed by David Garmire at Carnegie Mellon University under Guy Blelloch.
- The MPI extension to Ruby was developed by Emil Ong.