

Divide-and-Conquer Barnes-Hut Implementations

Maik Nijhuis

May 26, 2003

Chapter 1

Introduction

Divide-and-conquer parallelism is an interesting, easy-to-use, high level programming paradigm for writing parallel applications [3] [9]. However, only very few realistic and challenging applications have so far been implemented successfully in this paradigm. In this thesis we investigate whether the Barnes-Hut algorithm [1] can be efficiently parallelized in this paradigm. Because a pure divide-and-conquer implementation does not yield optimal performance, we also investigate whether extensions to the divide-and-conquer system can be used to enhance performance.

The Barnes-Hut algorithm is an $O(N \log N)$ N-body simulation. It is one of the challenging applications of the SPLASH-2 suite. Efficiently parallelizing the Barnes-Hut algorithm is not trivial. Special techniques have to be designed to minimize the (communication) overhead when the program is run in parallel.

The Satin programming environment [9] (built on top of Ibis [7]) provides the divide-and-conquer system we use. We based our code, which is written in the Java programming language, on an earlier Java RMI implementation of the Barnes-Hut algorithm [8].

In Chapter 2 we describe the divide-and-conquer strategies we designed for the Barnes-Hut algorithm [1]. Chapter 3 describes the implementation of those strategies and the problems we encountered during the implementation. In Chapter 4 we compare the performance of the three versions. Finally, in Chapter 5 we present our conclusions.

Chapter 2

Algorithms

Since we did not know about any earlier work that uses a divide-and-conquer system to implement the Barnes-Hut algorithm, it is not clear what the optimal parallelization strategy is. We therefore have designed and implemented four versions of the Barnes-Hut algorithm, which differ in the kind and amount of jobs they generate for the divide-and-conquer system.

We will use pseudo code to explain the algorithms we use. If a function uses the Satin divide-and-conquer system, the `spawn` keyword and `sync()` function call indicate interaction with Satin.

The `spawn` keyword is used in front of a function call to indicate that this function call is transformed into a job, which is then submitted to Satin. After a spawn of a job, the program does not wait for the result, but it continues running. At this time, variables which hold a result of a spawned function call cannot be used, since the spawned function call may not have completed.

When the `sync()` function of Satin is called, the program waits until all spawned jobs are run to completion. After a call to the `sync()` function, the results of the spawned function calls have been stored in their respective variables. The variables can now be safely used.

In the following section we describe some characteristics of the Barnes-Hut algorithm. In the next sections we describe the four versions we designed.

2.1 The Barnes-Hut algorithm

The Barnes-Hut algorithm simulates the evolution of a large set of bodies under the influence of forces. It can be applied to various domains of scientific computing, including but not limited to astrophysics, fluid dynamics, electrostatics and even computer graphics. In our implementation, the bodies represent planets and stars in the universe. The forces are calculated using Newton's gravitational theory.

The evolution of the bodies is simulated in discrete time steps. Each of these time steps corresponds to an iteration of the Barnes-Hut algorithm. An iteration consists of, among others, a force calculation phase, in which the force exerted on each body by all other bodies is computed, and an update phase, in which the new position and velocity is computed for each body. If all pairwise interactions between two bodies are computed, the complexity is $O(N^2)$.

The Barnes-Hut algorithm reduces this complexity to $O(N \log N)$, by using a hierarchical technique. During the calculation of the forces exerted on a certain body, this technique exploits the fact that a group of bodies which is relatively far away from that body can be approximated by a single virtual body at the center of mass of the group of bodies. This way, many pairwise interactions do not have to be computed. The precision factor *theta* indicates if a group of bodies is far enough away to use the optimization.

To make this optimization possible, the bodies are organized in a tree structure that represents the space the bodies are in. Since the space we represent is a three dimensional universe, we use an oct-tree to represent it. The bodies are located in the leaf nodes of the tree. The tree is recursively subdivided as more bodies are added.

The precision factor *theta* indicates when a part of the tree with bodies is approximated by its center of mass. The approximation is done if the distance to the part of the tree is greater than the size of this part of the tree multiplied by *theta*. The size of a part of the tree is the length of a side of the cube represented by the part of the tree.

In each iteration, the tree structure has to be rebuilt because the bodies have moved to a different part of space. The center of mass fields of the internal nodes of the tree also have to be computed each iteration, before the force calculation starts. An iteration thus consists of the following four phases:

1. Tree construction
2. Center of mass calculation
3. Force calculation
4. Body update (calculate new position and velocity)

Since typically more than 90% of the execution time is spent in the force calculation phase, we chose to parallelize only this phase. The other three phases are executed sequentially.

The following sections describe how we used divide-and-conquer techniques to parallelize the force calculation phase. We have designed four versions of the Barnes-Hut algorithm. The presented pseudo code describes an iteration of the Barnes-Hut algorithm. This piece of code contains the major differences between the four versions.

In all versions, we use the structure of the tree of bodies to recursively spawn jobs for *Satin*. A job is a part of the tree. It does the force calculation for the bodies in the leaf nodes in this part of the tree. When all leaf jobs are finished, the results of the jobs are recursively combined. The node that spawned the root job then has to update the acceleration fields of the bodies using the result of the root job.

Because spawning a job incurs some overhead in *Satin* and the number of jobs is very large, we have implemented a threshold value for the number of jobs that is spawned. The threshold value indicates the recursion depth at which no jobs are spawned anymore. All jobs are executed sequentially when the threshold is reached.

The threshold value requires some tuning. If it is too low, *Satin* will have difficulties balancing the load over the various compute nodes used in the computation. If it is too high, there is unnecessary overhead in *Satin*.

```

1 void barnesNaiveIteration( Body[] bodies ) {
2     TreeNode root;
3     JobResult result;
4
5     root = new TreeNode(bodies); // build tree
6     root.computeCentersOfMass();
7
8     result = spawn barnesNaiveJob(root, root); // force calculation
9     sync();
10
11     updateBodies(bodies, result);
12 }
13
14 JobResult barnesNaiveJob( TreeNode job, TreeNode root ) {
15     JobResult result;
16     int i;
17
18     if (! job.isLeaf()) {
19         JobResult childResults = new JobResult[8];
20         for (i = 0; i < 8; i++) {
21             childResults[i] =
22                 spawn barnesNaiveJob(job.children[i], root);
23         }
24         sync();
25         result = combineResults(childResults);
26     } else {
27         //do force calculation for the bodies in this leaf node
28         result = forceCalcLeafNode(job);
29     }
30     return result;
31 }

```

Figure 2.1: Pseudo code for the 'naive' version

2.2 The 'naive' version

The naive version implements a naive way of spawning jobs. The jobs are simply generated using the tree structure and each job has two parameters. One parameter is the tree the job represents. The other is the root of the whole tree, which is used for the force calculation of the bodies in the tree that the job represents. Pseudo code for an iteration of the naive version is given in Figure 2.1.

The input data for a job in the naive version thus consists of the whole tree of bodies (the job itself is part of it). This means that if a job is sent over the network, the whole tree is sent, which is inefficient. In the other versions we describe ways to increase the efficiency. The naive version is mainly used for comparison with the other two versions, e.g., for testing the correctness of the other versions.

2.3 The 'necessary tree construction' version

In this version, a job does not get the whole tree as a parameter. A 'necessary tree' is used instead. This necessary tree is a partial copy of the whole tree. It contains only the parts that are needed to do the force calculation for the bodies

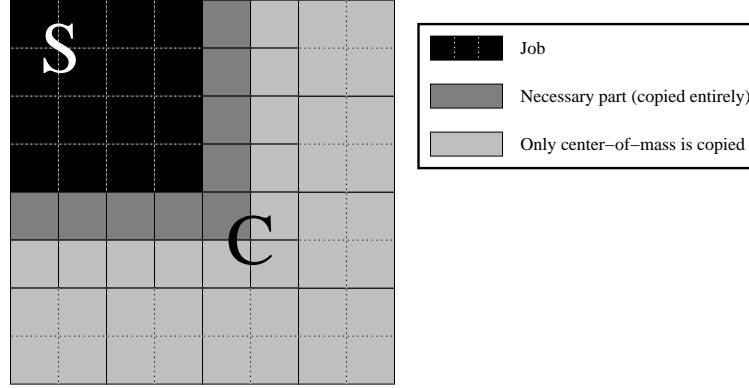


Figure 2.2: The necessary tree for 1/4 of the tree

in the job's part of the tree. The Barnes-Hut implementation by Blackston and Suel [2] uses a similar technique to generate locally essential trees.

The reason for constructing the necessary tree for each job is that a job, including its input data, has to be sent over the network when it is to be executed at a remote node. Sending the whole tree of bodies along with each stolen job causes much communication overhead. The necessary tree, however, is usually only slightly larger than the job it is constructed for. We will demonstrate this in the following subsection.

Before a job is spawned, the necessary tree for that job is constructed. The parts of the tree that are unnecessary are identified by examining if a subtree is far enough away to be represented by its center of mass. To do this, the minimum distance between the job's tree and the (center of mass of the) subtree is used. In fact, we simulate a body in the job's tree which is as close as possible to the subtree. If this simulated body is far enough away from the subtree to use its center of mass in the force calculation, all bodies inside the job can use that center of mass. This is because these bodies are even further away from the subtree.

2.3.1 Example

In Figure 2.2, the necessary tree for a job that has a size of 1/4 of the whole tree is shown. For clarity, a two dimensional space is used in this example (the real application simulates a three dimensional space). The value of theta is 1.0.

The smallest squares represent leaf nodes in the tree of bodies. The dark gray squares represent leaf nodes that are full copies of the corresponding leaf nodes in the original tree. The light gray squares represent nodes that only contain center of mass information. These nodes correspond to either leaf nodes or internal nodes in the original tree.

The job itself is also part of the necessary tree. Apart from this part of the tree, the necessary tree contains 9 leaf nodes that are full copies of the corresponding leaf nodes in the original tree. There are 11 leaf nodes and 7 internal nodes that only contain center of mass information. The internal nodes at the higher levels which glue the tree together are also part of the necessary

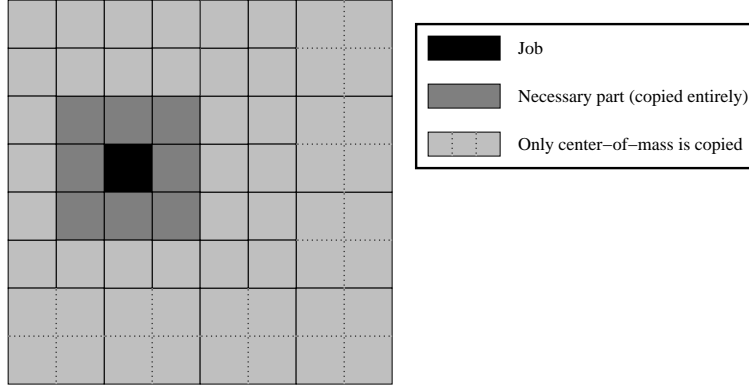


Figure 2.3: The necessary tree for 1/64 of the tree

Node type	Large job (Fig 2.2)		Small job (Fig 2.3)	
	Job	Nec. Tree	Job	Nec. Tree
Root	-	1	-	1
Internal 1/4	1	3	-	4
Internal 1/16	4	12	-	16
Leaf (only center of mass)	-	11	-	27
Total internal:	5	27	-	48
Leaf (complete)	16	9	1	8

Table 2.1: Number of nodes in a large and a small job

tree, including their center of mass information.

This center of mass information is necessary for subjobs of this job. To illustrate this, we have labeled two internal nodes in Figure 2.2 as 'S' and 'C'. These nodes both contain 4 leaf nodes. S is the subjob that contains the upper left part of the job. Note that the necessary tree for S is constructed using the necessary tree for the whole job, since the whole tree is unavailable (it is not part of the input data for a job). During the construction of the necessary tree for subjob S, it only needs center of mass information of C. This information thus has to be present in the necessary tree for the whole job.

If a job is relatively large, its necessary tree is relatively small. This becomes clear by comparing the large job shown in Figure 2.2, which represents 1/4 of the total tree, to the small job shown in Figure 2.3, which represents 1/64 of the total tree. Note that fully copied leaf nodes form the biggest part of a tree, since these nodes contain several bodies. The other nodes only contain center of mass information and tree structure fields. To indicate the size of the necessary tree relative to the job it is generated for, we will therefore use the ratio between the number of fully copied leaf nodes in the necessary tree and the number of leaf nodes in the job (which are always fully copied).

Table 2.1 shows the detailed differences between the two jobs. The ratio between the size of the necessary tree and the corresponding job is 9 to 16 for the large job, and 8 to 1 for the small job. The overhead of sending the necessary

tree along with the job thus increases when the jobs get smaller. This is exactly what we want, since Satin prefers large jobs over small jobs when it decides which job will be stolen if a remote node sends a steal request.

2.3.2 Code structure

In each iteration, a job that holds the root node of the tree is spawned. This job recursively spawns jobs that hold its child nodes, until a leaf node is reached. Jobs which hold a leaf node finally do the force calculation for the bodies in that leaf node. Figure 2.4 shows pseudo code for an iteration of the force calculation phase in the necessary tree version. The highlighted part indicates the difference to the 'naive' version.

Figure 2.5 shows pseudo code for constructor that creates the necessary tree. The `cutOffDistance` variable in a `TreeNode` object indicates at which distance its `centerOfMass` field can be used as an approximation of the bodies below it. This value is dependent on the size of the node and the precision factor `theta`.

2.4 The tuple space version

As shown in Chapter 4, the necessary tree version does not exhibit optimal efficiency. We have therefore designed the tuple space version, which uses a different approach to increase the efficiency of the 'naive' version. The tuple space version uses an extension to the Satin divide-and-conquer system called the Satin tuple space. This extension was built with the Barnes-Hut algorithm in mind.

With the Satin tuple space a program can efficiently share Java objects between the various compute nodes. A program can put objects in the tuple space, get objects from the tuple space and remove objects from the tuple space. To identify an object in the tuple space, a key is used which is attached to the object when it is put in the tuple space. This key can then be used to get or remove the object. The tuple space works by broadcasting the objects that are inserted into it. This is done at the moment they are inserted into the tuple space.

The tuple space version uses the same hierarchical technique as the naive version to generate jobs. The data distribution to the various compute nodes is entirely different, however. At the start of the force calculation phase, the whole tree of bodies is put in the Satin tuple space. Satin then broadcasts it to all compute nodes. A job's input only consists of a pointer to a node in the tree, and a tuple space key that is used to lookup the tree for this iteration in the tuple space. All other data is already present at the node where the job is run. At the end of the force calculation phase, the tree is removed from the tuple space to save memory. Pseudo code for the tuple space version is given in Figure 2.6. The highlighted parts indicate the differences to the 'naive' version.

2.5 The active tuple version

In the tuple space version, we noticed that inserting the whole tree with bodies into the tuple space incurs significant communication overhead. To minimize this overhead, we implemented another version that uses the Satin tuple space.


```

1 void barnesNTCIteration( Body[] bodies ) {
2     TreeNode root;
3     JobResult result;
4
5     root = new TreeNode(bodies); // build tree
6     root.computeCentersOfMass();
7
8     result = spawn barnesNTCJob(root, root); // force calculation
9     sync();
10
11     updateBodies(bodies, result);
12 }
13
14 JobResult barnesNTCJob( TreeNode job, TreeNode necessaryTree ) {
15     TreeNode subjobNecessaryTree;
16     JobResult result;
17     int i;
18
19     if (! job.isLeaf()) {
20         JobResult childResults = new JobResult[8];
21         for (i = 0; i < 8; i++) {
22             /*
23              * invoke the necessary tree constructor to create the
24              * necessary tree for this subjob
25              */
26             subjobNecessaryTree =
27                 new TreeNode(necessaryTree, job.children[i]);
28
29             childResults[i] = spawn
30                 barnesNTCJob(job.children[i], subjobNecessaryTree);
31         }
32         sync();
33         result = combineResults(childResults);
34     } else {
35         //do force calculation for the bodies in this leaf node
36         result = forceCalcLeafNode(job);
37     }
38     return result;
39 }

```

Figure 2.4: Pseudo code for the 'necessary tree construction' version

```

1  TreeNode( TreeNode original, TreeNode job )
2      double distance;
3      int i;
4
5      // copy original's position, size, center of mass, and cut off
6      // distance (just like a standard copy constructor) (not shown)
7
8      distance = minimalDistance(original.centerOfMass, job);
9      if (distance < cutOffDistance) {
10         if (original.isLeafNode()) {
11             bodies = original.bodies; // copy the bodies in original
12         } else {
13             // copy the necessary parts of the children of original
14             // recursively using this constructor
15             children = new TreeNode[8];
16             for (i = 0; i < 8; i++) {
17                 children[i] = new TreeNode(original.children[i], job);
18             }
19         }
20     } // else this part can be cut off, nothing has to be copied
21 }

```

Figure 2.5: Pseudo code for the necessary tree constructor

In this version, the result of the force calculation phase is inserted into the tuple space, instead of the whole tree with bodies. The result of the force calculation phase is a set of three dimensional vectors, one for each body. Distributing these structures is significantly more efficient than distributing the tree with bodies.

To make this work, the bodies and the tree with bodies are replicated at all nodes. The sequential phases in each iteration (tree building, center of mass calculation and body updating), which are normally only run at the root node, are run in parallel at all compute nodes. Because all nodes except the root node would otherwise be idle during the sequential phases, replicating this work should not slow down the program.

The Satin tuple space also had to be adjusted to support active tuples. These active tuples are used to distribute the force calculation result and to do the replicated execution of sequential work. Active tuples implement function shipping (data shipping is used for 'normal' tuples, as in the tuple space version). Apart from the data that is present in a 'normal' tuple, an active tuple also contains a method called 'handleTuple'. When an active tuple is put into the tuple space, it is broadcast to all nodes. At each node, the 'handleTuple' method of the active tuple object is then invoked. Active tuples are not stored in the tuple space. They can therefore not be retrieved or removed from the tuple space.

At the start of the program, an active tuple with the parameters for the application, such as the number of bodies, is inserted. This active tuple initializes each node. An active tuple with the force calculation result of the previous iteration is inserted at the start of each iteration. Figure 2.7 shows pseudo code for this tuple. The active tuple first updates the bodies at each node, using the data in the tuple. Then it builds the tree with bodies and it computes the center of mass fields for the tree. This way, each node has an updated copy of the tree with bodies when the force calculation begins. Pseudo code for an iteration of

```

1 void barnesTupleIteration( Body[] bodies, int iteration ) {
2     TreeNode root;
3     JobResult result;
4     String rootIdentifier = "root" + iteration;
5     root = new TreeNode(bodies); // build tree
6     root.computeCentersOfMass();
7
8     // force calculation
9     TupleSpace.put(root, rootIdentifier);
10    result = spawn barnesTupleJob(root, rootIdentifier);
11    sync();
12    TupleSpace.remove(rootIdentifier);
13
14    updateBodies(bodies, result);
15 }
16 JobResult barnesTupleJob( TreeNodePointer jobPointer,
17     String rootIdentifier ) {
18     TreeNode root = TupleSpace.get(rootIdentifier);
19     TreeNode job = root.findNode(jobPointer);
20
21     int i;
22     JobResult result;
23
24     if (! job.isLeaf()) {
25         JobResult childResults = new JobResult[8];
26         for (i = 0; i < 8; i++) {
27             childResults[i] = spawn barnesTupleJob(job.children[i],
28                 rootIdentifier);
29         }
30         sync();
31         result = combineResults(childResults);
32     } else {
33         //do force calculation for the bodies in this leaf node
34         result = forceCalcLeafNode(job);
35     }
36     return result;
37 }

```

Figure 2.6: Pseudo code for the 'tuple space' version

```

1 class Updater() implements ActiveTuple {
2     JobResult prevResult;
3
4     Updater(JobResult jr) {
5         prevResult = jr;
6     }
7
8     // active tuple method, called at each node upon receipt
9     handleTuple() {
10         BarnesHut.updateBodies(BarnesHut.bodies, prevResult);
11
12         BarnesHut.root = new TreeNode(BarnesHut.bodies); //build tree
13         BarnesHut.root.computeCentersOfMass();
14     }
15 }

```

Figure 2.7: Pseudo code for the updating active tuple

```

1 class BarnesHut {
2     //these fields are updated by the active tuples
3     static Body[] bodies;
4     static TreeNode root;
5
6     JobResult barnesActiveTupleIteration( JobResult prevResult ) {
7         JobResult result;
8         Updater u; //active tuple
9
10        //broadcast updates from previous iteration
11        u = new Updater(prevResult);
12        TupleSpace.add(u);
13
14        result = spawn barnesActiveTupleJob(root); //force calculation
15
16        sync();
17
18        // 'result' will be passed to this function as the parameter
19        // in the next iteration
20        return result;
21    }
22
23    JobResult barnesActiveTupleJob( TreeNodePointer jobPointer ) {
24        TreeNode job = root.findNode(jobPointer);
25
26        JobResult result;
27        int i;
28
29        if (! job.isLeaf()) {
30            JobResult childResults = new JobResult[8];
31            for (i = 0; i < 8; i++) {
32                childResults[i] =
33                    spawn barnesActiveTupleJob(job.children[i]);
34            }
35            sync();
36            result = combineResults(childResults);
37        } else {
38            //do force calculation for the bodies in this leaf node
39            result = forceCalcLeafNode(job);
40        }
41        return result;
42    }
43 }

```

Figure 2.8: Pseudo code for the 'active tuple' version

the active tuple version is given in Figure 2.8. The highlighted parts indicate the differences to the 'naive' version.

Chapter 3

Implementation

All four algorithms described in Chapter 2 are implemented in the Java programming language. The Satin divide-and-conquer system [9] is used. Satin is built on top of Ibis [7], which is an efficient communication library written entirely in the Java programming language.

The implementation of the methods that simulate Newton's theory is based on the corresponding methods in a Java RMI implementation of the Barnes-Hut algorithm [8]. These methods compute the center of mass of a node of the tree of bodies, the interaction between two bodies in the force calculation phase, and the new position and velocity of a body in the body update phase, for example. Various parameters for the Barnes-Hut algorithm, such as the potential softening value used in the force calculation, are copied from the Barnes-Hut version as found in the SPLASH-2 suite.

3.1 Job results

When a divide-and-conquer system is used to spawn jobs recursively, the results of these jobs also have to be recursively combined. In all versions a linked list is used to store the result of a job. Each body has a unique number attached to it, which is an index in a (local) array at the root node of the computation that contains all bodies. The box in Figure 3.1 shows a schematic representation of the result of a leaf job. The linked list contains two elements. The first element contains an array with the numbers of the bodies in the job. The second element contains an array with the results of the force calculation for those bodies. The result of the force calculation for a body is a three dimensional vector.

Using linked lists, results can be easily combined by concatenating these lists. When all jobs in an iteration of the force calculation phase are done, the root

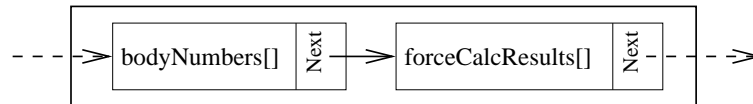


Figure 3.1: The result of a leaf job

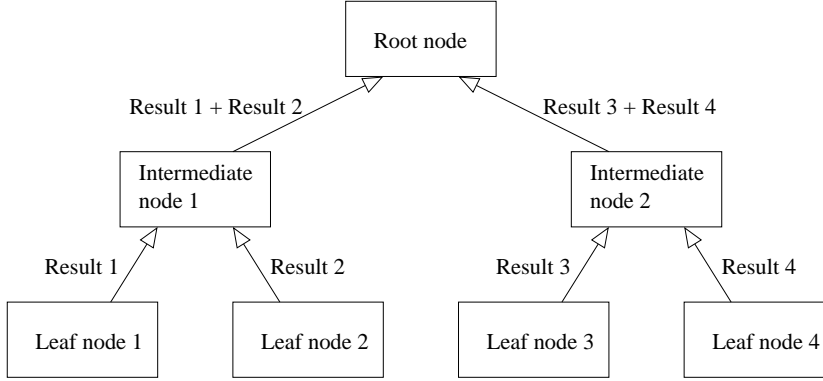


Figure 3.2: Recursively combining four jobs

node has one linked list with several arrays in it. The root node then copies the force calculation results from the arrays in the linked list to the corresponding bodies in its local array of bodies.

When linked lists are combined recursively this way, it results in linked lists with many small array elements, as the recursion depth decreases. Because of this, we have implemented a method that optimizes a linked list by combining the arrays in the list. This results in a linked list with only two elements, one array with body numbers and one array with force calculation results. The list processing code at the root node did not have to be adjusted to this optimization.

The list optimizing method is called by jobs at the recursion threshold, to optimize the results of the sequentially executed jobs. This is because the result of a sequentially executed job is never sent over the network. Combining results within a sequentially executed job will only add overhead. A job that is submitted to the divide-and-conquer system can, however, be stolen by a remote node. Its result will then be serialized and sent over the network to the parent job. In this case, reducing the amount of objects by optimizing the linked list causes less communication overhead, at the price of little computation overhead.

The approach with linked lists is not without problems, because job results may be sent over the network multiple times. Figure 3.2 shows an example of this. The rectangles indicate compute nodes and the arrows indicate network traffic. In this figure, two jobs have been generated by the root node, which have been stolen by two intermediate nodes. Both intermediate nodes have generated two subjobs, which were stolen by the leaf nodes. The leaf nodes have computed their jobs, and return a linked list with the result to the intermediate nodes. The intermediate nodes concatenate the linked lists they have received and send the concatenated list to the root node. This way, all results are sent over the network two times. For clarity reasons, the maximum recursion depth is two in the figure and two jobs are combined at each recursion level. In reality, however, the recursion depth is larger and each job can consist of up to eight subjobs.

In the worst case scenario the number of times a result is sent over the network is equal to the recursion threshold. This occurs when all jobs are run on a different node than the one that generated the job. This was demonstrated in Figure 3.2. However, a job is usually run at the same node as its parent. The

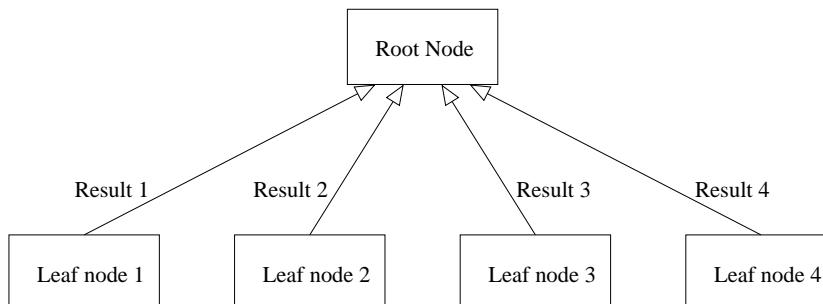


Figure 3.3: Using RMI to return the result of four jobs

result does not have to be sent over the network then.

To circumvent that a result is sent multiple times, we have tried a different approach of getting the results to the root node. We tried using Remote Method Invocation to send each result of a leaf job directly to the root node. This behavior is shown in Figure 3.3. Because using the efficient Remote Method Invocation implementation in Ibis [6], in combination with the Satin divide-and-conquer system (which also uses Ibis), was currently not possible, we had to revert to using the standard Java RMI implementation. This resulted in much performance loss compared to the linked lists approach, so we decided to use the linked list approach instead of standard Java RMI.

3.2 Inlining Vec3 objects

Originally, we implemented all Barnes-Hut versions using `Vec3` objects. These are simple objects that represent a three dimensional coordinate or vector. A `Vec3` object contains a `x`, `y`, and `z` value. It also contains some methods to manipulate these values, for example a method that adds the values of another `Vec3` object to those of this object.

However, the use of `Vec3` objects causes a tree with bodies to consist of many objects. Each tree node contains two `Vec3` objects and each body contains four `Vec3` objects. The use of these objects yields very clear code. However, it incurs some overhead in the Java Virtual Machine, for example in the garbage collector. Moreover, all these objects (that are not transient) have to be serialized and sent over the network when the tree is to be sent over the network.

To investigate the overhead of using `Vec3` objects, we have made a copy of the program in which all `Vec3` objects are inlined. In this new program, all `Vec3` objects are replaced by three separate `x`, `y`, and `z` variables. All code that uses `Vec3` objects is refactored accordingly. For example, the linked list with the result of a job now contains three separate arrays for the force calculation results, instead of one. These three arrays contain the `x`, `y`, and `z` values of the force calculation results. As we will show in Chapter 4, inlining the `Vec3` objects indeed improved performance.

Chapter 4

Performance

In this chapter, we evaluate the performance of our implementations of the Barnes-Hut algorithm. We first compare the inlined version to the version that uses `Vec3` objects. Then we show the performance of the inlined version for the naive, necessary tree, tuple space and active tuple implementations. Finally a comparison is made between the necessary tree, tuple space and active tuple implementations.

All measurements are done on the DAS-2 cluster at the Vrije Universiteit. The DAS-2 system [5] consists of five clusters at five universities in the Netherlands. Each node in a cluster contains two 1.00-GHz Pentium III processors and at least 1GByte RAM. It has a Fast Ethernet and a Myrinet [4] network interface. Our program uses the Myrinet network for its communication. The IBM Java 2 Runtime Environment, Standard Edition, version 1.4.1 is used to run the program.

The problem size used in the experiments is set to 2,000,000 bodies. We have done measurements with theta values of 1.0, 2.0 and 3.0. The maximum amount of bodies per leaf node is set to the value that yields optimal performance when the sequential implementation of the inlined version is used. It depends on the value of theta. In Table 4.1 the values are listed.

In all experiments we compute seven iterations of the Barnes-Hut algorithm. However, since the first iteration is always significantly slower than the other iterations, the measurements do not include the first iteration.

All speedups presented in this chapter are computed relative to the run time of a purely sequential implementation of the Barnes-Hut algorithm. Its behavior resembles the behavior of the naive version. The program uses the tree with bodies to find all leaf nodes. When a leaf node is encountered, the force calculation for the bodies in this leaf node is done. Then the tree is used to find the next leaf node, and so on. To exploit cache behavior, the tree with bodies is traversed in depth-first order. This way, the force calculation can

Theta	1.0	2.0	3.0
Max. bodies / leaf node	160	80	60

Table 4.1: The maximum amount of bodies per leaf node

be seen as being split up in different time slots. In each time slot, the force calculation is done for bodies that are near each other in the represented three dimensional space. Because the force calculation for bodies that are near each other usually requires the same parts of the tree, the memory cache is used efficiently this way.

The program can be compiled in two ways. It can be compiled as a normal Java program and it can be compiled with the Satin compiler, which rewrites the Java byte code to add function calls to the Satin runtime environment. When the rewritten program is run sequentially, these function calls add little overhead. We noticed that the performance of the sequential implementation is slightly better when the rewritten byte code is used. This is probably due to cache effects. Since the implementation with rewritten byte code is the fastest sequential version, the run time of this version is used to compute the speedup of the parallel runs.

All presented numbers are the average of the results of at least two measurements. This is because the program sometimes did not produce a result because of a deadlock in one of the lower communication layers. Since Ibis is an experimental system, not all problems in it had been solved when we ran the program to produce measurements. When a deadlock occurred, we reran the program with the same parameters. We usually reran the program multiple times, until at least two measurements were produced.

When a run was at least ten percent slower than other runs with the same parameters, we compared the timings of the iterations of the runs. If the iterations of the slow run were consistently slower than the iterations in the other runs, the result of the slow run was discarded. If the number of successful runs with the same parameters had dropped below two this way, the program was rerun to produce more measurements, as explained above.

To investigate the run time behavior of the program, we have run it with enabled Satin statistics. These statistics give insight into the job distribution between the nodes and the corresponding network traffic. They also specify how much time is spent in Satin. For the speedup figures presented, these statistics were disabled to achieve maximum performance.

Among others, the Satin statistics show the percentage of the total load balancing time and the time needed to broadcast the tuples that are put in the Satin tuple space. The load balancing time is the time used by Satin to send steal request and reply messages, and to process these messages.

Unfortunately, the Satin statistics can not be used to present a detailed performance breakdown. This is because these statistics are gathered during the entire run of the program. This includes the first iteration, which is not measured by the Barnes-Hut program.

4.1 Inlining Vec3 objects

For comparing the inlined version to the version that uses `Vec3` objects, we have started our comparison with the sequential implementations of these versions. We noticed a large performance difference. On average, the version that uses `Vec3` objects runs for 1755 seconds when a problem size of 2.000.000 bodies is used and θ is set to 1.0. With the same parameters, the inlined version runs for 499 seconds.

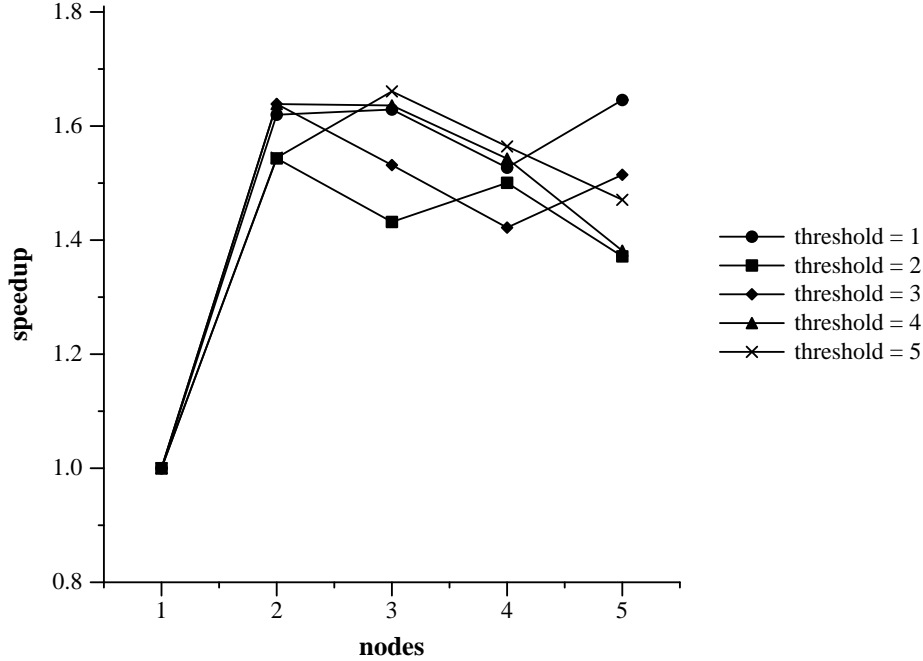


Figure 4.1: Performance of the naive version

The elimination of the use of `Vec3` objects thus increases performance by more than a factor three. Apparently, the use of many small objects incurs a lot of overhead in the Java Virtual Machine. The garbage collector, for example, has to keep track of all these objects.

Because of this large performance difference when only the sequential run time is compared, we decided not to compare the parallel run time of both versions. The difference between both versions would be even larger, because the use of many objects also incurs much communication overhead.

4.2 The 'naive' version

In Figure 4.1 we show the relative speedup of the 'naive' version, using different recursion thresholds. The number of compute nodes varies between 1 and 5. Theta is set to 2.0. Because we ran out of memory, the number of bodies is set to 1,000,000. The maximum number of bodies per leaf node is set to 50.

The program achieves a moderate speedup on two nodes. On more nodes the speedup hardly increases. The run times of the iterations within one run show large differences, especially when a high threshold and a large number of compute nodes are used. In a run on five nodes with a threshold value of five the largest difference in run time between two iterations is 33 %.

Because the run times of the iterations differ, the total run time also differs. This explains the chaotic behavior of the curves in Figure 4.1.

The low speedup can be explained by the high communication overhead of the naive version. Whenever a job is sent over the network, the entire tree of

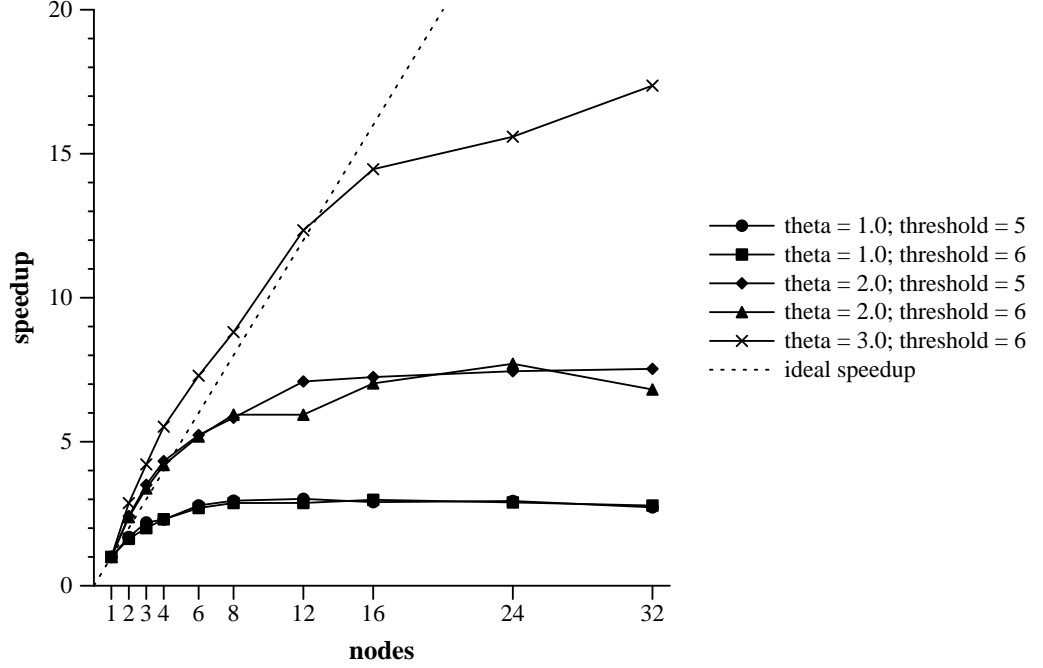


Figure 4.2: Performance of the necessary tree version

bodies is transferred. This causes the maximum speed up to be limited to 1.7.

4.3 The optimized versions

Figure 4.2, 4.3 and 4.4 show the performance of the necessary tree, tuple space and active tuple version, respectively. The recursion threshold in the measurements is set to the optimal value. When two recursion threshold settings showed similar results, both measurements are shown.

All graphs show better performance for higher values of θ . This is because the computation to communication ratio goes up when a higher value of θ is used. The program does more computations when a higher value of θ is used. A higher value of θ means that fewer tree nodes can be approximated by their center of mass (see Section 2.1).

When the necessary tree version is used, communication increases with θ because the necessary trees get bigger. Apparently, the computation increase outweighs the communication increase when the necessary tree version is used. When the tuple space or active tuple version is used, the amount of communication does not depend on θ .

The measurements on small numbers of nodes exhibit superlinear speedup when a θ value of 2.0 or 3.0 is used. This is probably due to cache effects. When more compute nodes are used, more cache memory is available to and used by the program.

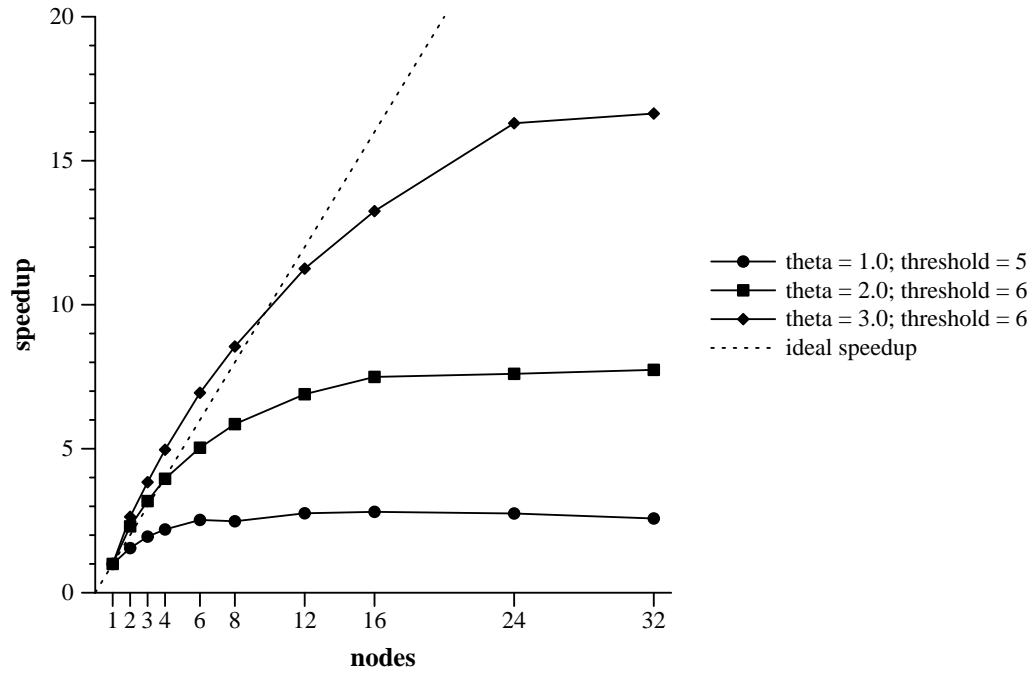


Figure 4.3: Performance of the tuple space version

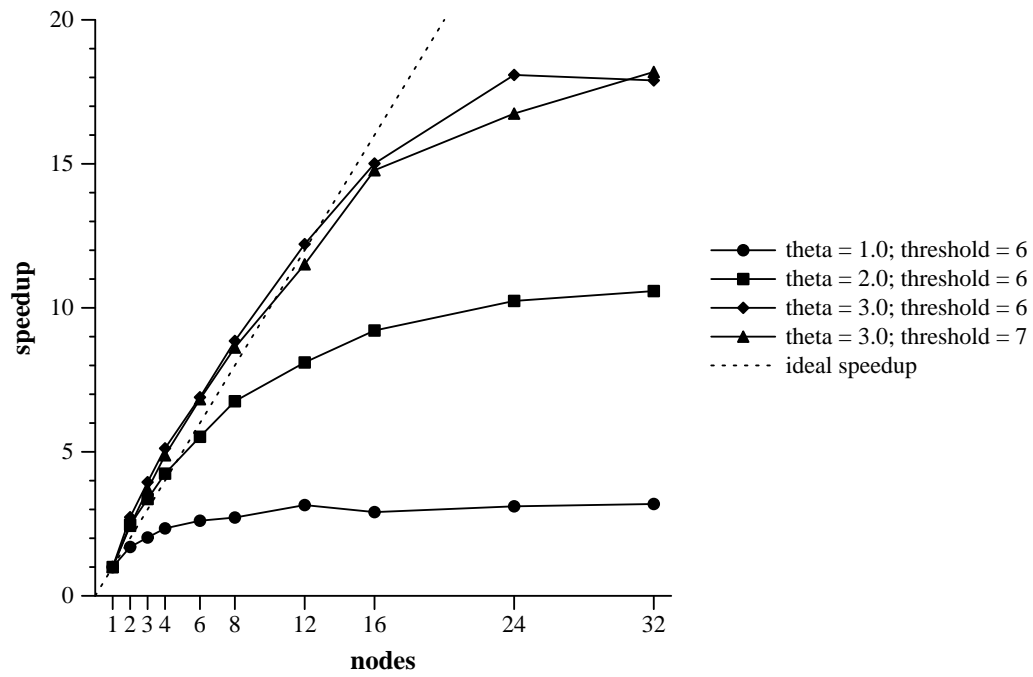


Figure 4.4: Performance of the active tuple version

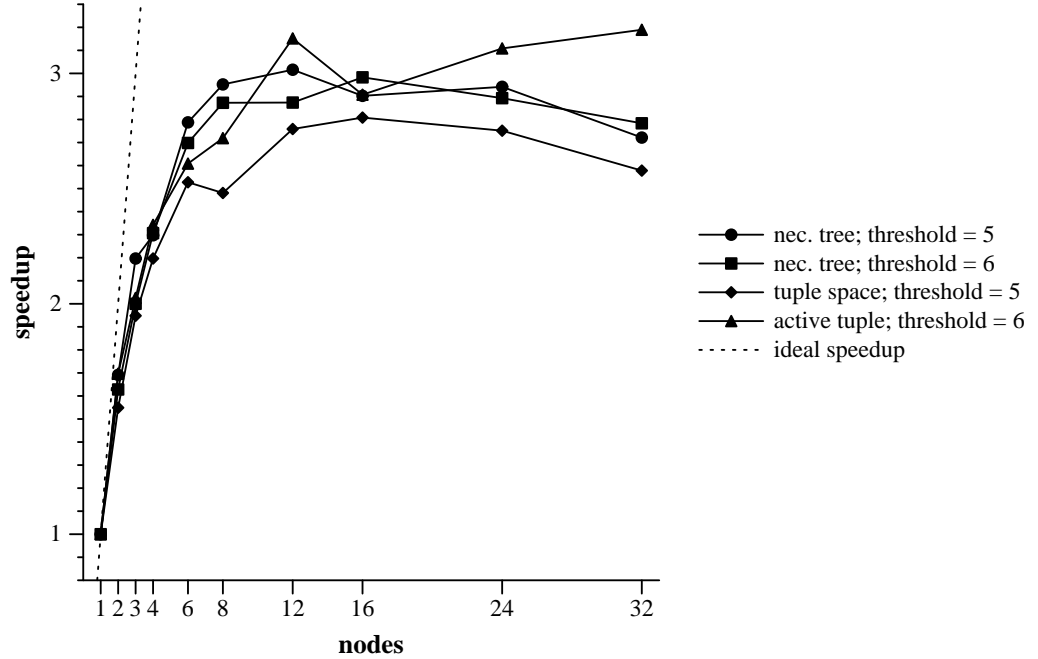


Figure 4.5: Performance comparison, $\theta = 1.0$

4.4 Comparison

4.4.1 Theta = 1.0

When θ equals one, the sequential phases of the program take approximately 30 seconds. The total sequential run time is approximately 500 seconds. The maximally achievable speedup is therefore 16.7.

However, Figure 4.5 shows that the best speedup achieved is approximately 3.2. We believe the high load balancing overhead is the main cause for the low speedup. The Satin statistics show that the percentage of the total time spent in load balancing increases with the number of nodes. For example, when the necessary tree version is run at 4 nodes, the nodes spend 40 % of their time in load balancing, on average. When the necessary tree version is run at 16 nodes, this percentage increases to 75.

The necessary tree version has a high load balancing overhead because the jobs it generates are large. If a job is stolen by a remote node, it will be part of a (positive) steal reply, which will also be large. Since the overhead of sending and processing the steal reply is part of the load balancing time, the load balancing time is dependant on the job size.

When one of the two tuple space versions is used, the jobs are smaller than the jobs of the necessary tree version. The load balancing overhead is thus less. However, time is spent to broadcast the tuples. This time also increases with the number of nodes. This is another reason why the speedup does not increase when more nodes are used. For example, a broadcast of a tuple with the entire tree of bodies (in the tuple space version) takes on average 12 seconds when the

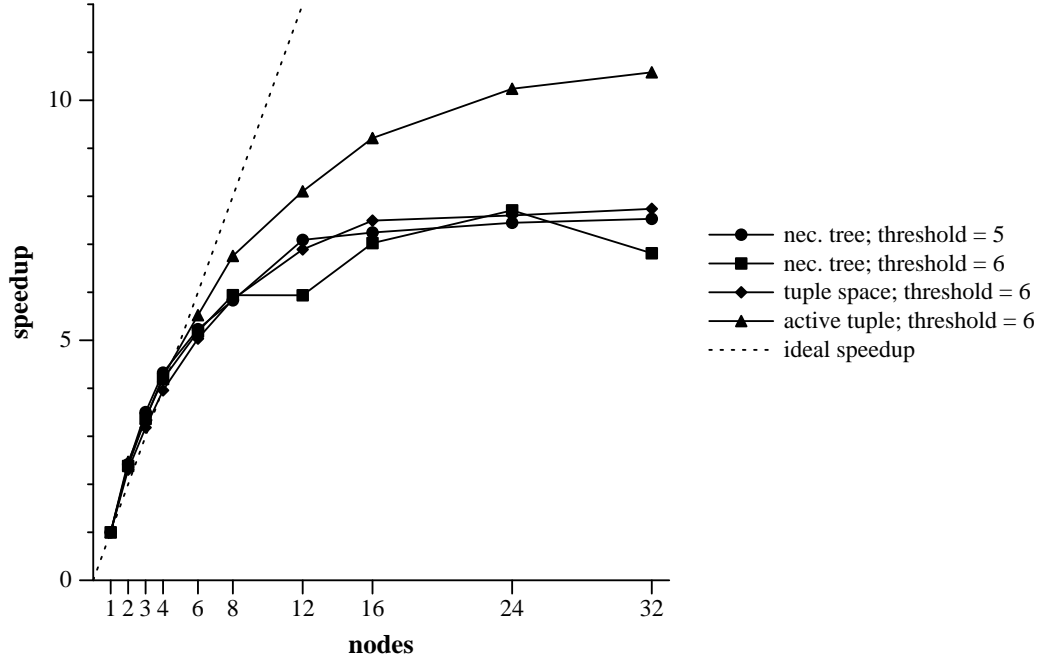


Figure 4.6: Performance comparison, $\theta = 2.0$

program is run at 4 nodes and 14 seconds when the program is run at 16 nodes.

The broadcasts of the tuples in the active tuple version are faster than those in the tuple space version. At 4 nodes a broadcast takes 2.5 seconds and at 16 nodes it takes 5 seconds, on average. Figure 4.5 also shows that the active tuple version performs better than the tuple space version. The difference can easily be explained. The active tuples contain very simple data structures (arrays of doubles), which can easily be serialized, while the 'normal' tuples contain trees, which are more difficult to serialize. This also causes the active tuples to be smaller. The size of an active tuple is 48,000.066 bytes. The size of a 'normal' tuple differs a little, because the tree with bodies changes every iteration. On average, the size of a 'normal' tuple is 84,774.777 bytes. The broadcast of an active tuple is thus more efficient than the broadcast of a 'normal' tuple.

As Figure 4.5 indicates, the active tuple version is the only version that keeps performing better as more nodes are added beyond 16. Apparently, the low load balancing overhead compared to the necessary tree version and the efficient tuple broadcast cause the active tuple version to scale better than the other versions.

4.4.2 $\theta = 2.0$

When a θ value of 2.0 is used, the sequential phases of the program take approximately 32 seconds. This is a little longer than the time of the sequential phases when θ is 1.0. The value of θ does not directly affect the sequential phases. Only the result of the force calculation phases is directly dependant on the value of θ . This causes differences in the sequential phases,

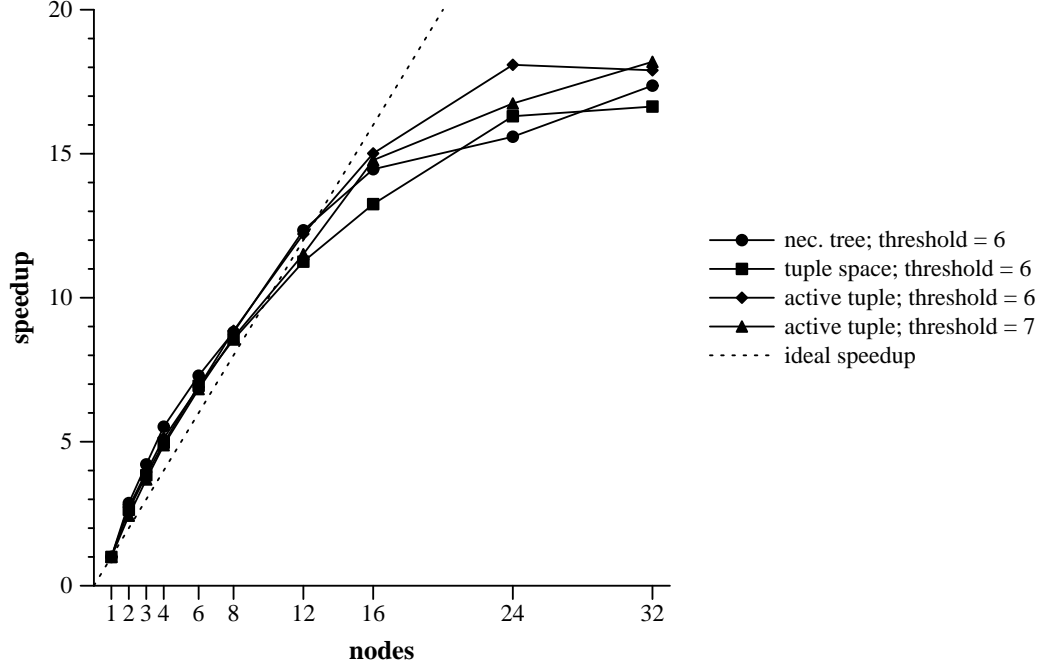


Figure 4.7: Performance comparison, $\theta = 3.0$

as a different force calculation result causes different updates to be applied. Consequently, different trees are built.

The total sequential run time (approximately 2150 seconds) is clearly affected by the value of θ . The maximally achievable speedup increases from 16.7 for a θ value of 1.0 to 67 when θ is 2.0.

The active tuple version clearly performs better than the other versions when θ is 2.0, as shown in Figure 4.6. The speedup of the necessary tree and tuple space version seems to be limited to about 8. Beyond 16 nodes the speedup hardly increases. The active tuple version reaches a speedup of 10.6 at 32 nodes. The speedup limit of the active tuple version is probably a little higher, since the active tuple curve still rises at the end.

This result can again be attributed to the low load balancing overhead of the active tuple version and its efficient tuple design.

4.4.3 $\theta = 3.0$

The reason for doing measurements with a θ value of 3.0 is to investigate whether increasing the computation to communication ratio results in better scalability. θ values of 1.0 and 2.0 are common values for the Barnes-Hut algorithm. By using a θ value of 3.0, the algorithm complexity gets closer to $O(N^2)$. This is because the optimizations that cause the algorithm to have a complexity of $O(N \log N)$ will be used less when θ is 3.0 (see Section 2.1).

The sequential phases take approximately 35 seconds when θ is 3.0. The total sequential run time is approximately 6380 seconds. The maximally achiev-

able speedup is therefore 182. With theta values of 1.0 and 2.0 the maximally achievable speedup is 16.7 and 67, respectively.

When theta equals 3.0, scalability indeed increases. The necessary tree and active tuple versions achieve a reasonable speedup when the program is run at 16 nodes or less. There is no significant performance difference between the necessary tree and active tuple versions. At more than 16 nodes, the active tuple version performs best, although the necessary tree and tuple space versions also perform quite well. Apparently, the active tuple version still benefits from its low load balancing overhead and efficient tuple design.

Chapter 5

Conclusions and Future work

Our goal was to investigate whether the Barnes-Hut algorithm can be efficiently parallelized using a divide-and-conquer system. We have shown that this goal can not be achieved using a naive divide-and-conquer implementation, because of high communication overhead.

The accuracy of the Barnes-Hut algorithm can be tuned with a parameter called theta. When a high value of theta is used, the program does more computations, but its communication pattern does not change. By altering the value of theta, we could therefore alter the computation to communication ratio and investigate whether this would affect performance.

To minimize communication overhead, we have designed three optimized implementations. The necessary tree implementation, described in Section 2.3, uses only divide-and-conquer techniques to parallelize the program. The tuple space and active tuple implementations also use a tuple space, which is an extension to the divide-and-conquer system. The tuple space and active tuple implementations use the tuple space for data shipping and function shipping, respectively.

The theta values we used are 1.0, 2.0 and 3.0. The maximally achievable speedup and the measured performance indeed increase when theta is increased. The active tuple version exhibits the best performance in all cases, except when theta is set to 1.0 and the program is run at 8 compute nodes or less. In this case, the necessary tree version performs best.

The tuple space version has the worst performance of the three versions when theta is set to 1.0. For theta values of 2.0 and 3.0, its performance is almost equal to that of the necessary tree version.

We have shown that the performance and scalability increase with the value of theta. This is true for all three optimized versions. The Barnes-Hut algorithm can thus efficiently be parallelized using a divide-and-conquer system, but only when high values of theta are used. By extending the divide-and-conquer system with extra functionality, performance can be improved.

The active tuple version is thus our best divide-and-conquer Barnes-Hut implementation. The use of active tuples resembles replicated method invocation (RepMI), because both use function shipping. One important difference is that

the consistency model of RepMI is more strict than that of the tuple space.

The Ibis communication library we used supports RepMI. However, at the time of writing, this functionality could not be used in combination with the Satin divide-and-conquer system, which is used by the Barnes-Hut program. When the combination is possible, RepMI could be used to create a new divide-and-conquer implementation of the Barnes-Hut algorithm. This new implementation could be used to investigate if the tuple space version benefits from its weak consistency model.

Another issue that remains open is if our implementations can run efficiently at a wide area network, such as the Grid. When a wide area network is involved, broadcasts become relatively expensive. Therefore, the two tuple space implementations might not perform well on the Grid. The necessary tree version might perform better, because it is a pure divide-and-conquer program. Van Nieuwpoort has shown in [9] that (pure) divide-and-conquer programs are very well suited to be run at the Grid.

Bibliography

- [1] J. Barnes and P. Hut. A hierarchical $O(n \log n)$ force calculation algorithm. *Nature*, 324:446–449, December 1986.
- [2] D. Blackston and T. Suel. Highly portable and efficient implementations of parallel adaptive N -body methods. In *Proceedings of SC'97: High Performance Networking and Computing*, November 1997.
- [3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*, pages 207–216, Santa Barbara, CA, July 1995.
- [4] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.
- [5] The Distributed ASCI Supercomputer 2. <http://www.cs.vu.nl/das2/>.
- [6] J. Maassen. *Method Invocation Based Communication Models for Parallel Programming in Java*. PhD thesis, Faculty of Sciences, Division of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, June 2003.
- [7] R. V. v. Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Ibis: an Efficient Java-based Grid Programming Environment. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 18–27, Seattle, Washington, USA, November 2002.
- [8] E. Thieme. Parallel programming in java: Porting a distributed barnes-hut implementation. Master's thesis, Faculty of Sciences, Division of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, August 1999.
- [9] R. van Nieuwpoort. *Efficient Java-Centric Grid-Computing*. PhD thesis, Faculty of Sciences, Division of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, September 2003.