Divide-and-Conquer Barnes-Hut Implementations

Maik Nijhuis

February x, 2003

Chapter 1

Introduction

Currently, there are very few challenging applications that can be efficiently run in parrallel, using a divide-and-conquer system. In this thesis we investigate if the Barnes-Hut algorithm [1] can be efficiently parallelized using such a system. We also investigate if 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. The Satin programming environment [8] (built on top of Ibis [6]) provides the divide-and-conquer system we use. We based our code, which is written in the Java programming language, on a Java RMI implementation of the Barnes-Hut algorithm [7].

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 had no experience in using a divide-and-conquer system to implement the Barnes-Hut algorithm, we did not know how to use the system optimally. We therefore have designed and implemented three 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 for Satin, which is then submitted to it. 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 three 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 equal to $O(N^2)$.

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. A precision factor 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 a 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 measured as 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 three versions of the Barnes-Hut algorithm. The presented pseudo code describes an iteration of the Barnes-Hut algorithm. That part contains the major differences between the three 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, and Satin is not used anymore.

The threshold value requires some tuning. If it is too low, the divide-and-conquer system will have trouble balancing the load over the various compute

```
void barnesNaiveIteration( Body[] bodies ) {
    TreeNode root;
    JobResult result;

    root = new TreeNode(bodies); // build tree
    root.computeCentersOfMass();

    result = spawn barnesNaiveJob(root, root); // force calculation
    sync();

    updateBodies(bodies, result);
}

JobResult barnesNaiveJob( TreeNode job, TreeNode root ) {
    JobResult result;
    int i;

    if (! job.isLeaf()) {
        JobResult childResults = new JobResult[8];
        for (i = 0; i < 8; i++) {
            childResults[i] = spawn barnesNaiveJob(job.children[i], root);
        }
        sync();
        result = combineResults(childResults);
    } else {
        //do force calculation for the bodies in this leaf node
        result = forceCalcLeafNode(job);
    }
    return result;
}
```

Figure 2.1: Pseudo code for the 'naive' version

nodes used in the computation. If it is too high, there is a lot of unnecessary overhead in Satin. The command line option `-t` can be used to set the threshold value.

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 quite inefficient. In the other two versions we describe two 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.

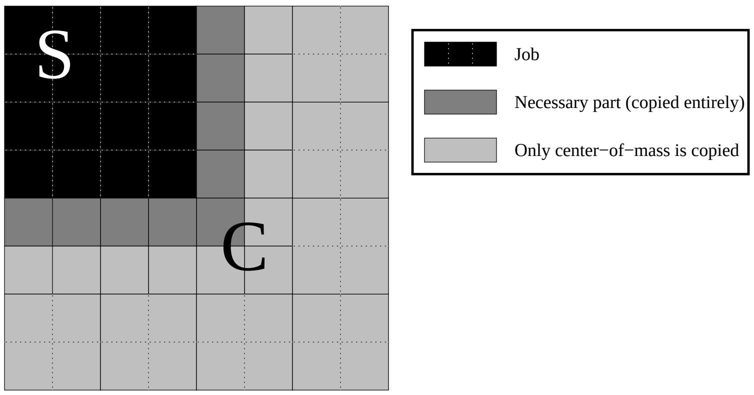
S
C
Job
Necessary part (copied entirely)
Only center−of−mass is copied

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

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 copy of the whole tree that contains only the parts that are needed to do the force calculation for the bodies 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. If a job, including its input data, is sent over a network, sending only the necessary tree instead of the whole tree reduces communication overhead, as we will explain later.

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 the subtree of the examined part of the tree will be used in the force calculation for the bodies inside the job, or if this part 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) examined part of the tree is used. In fact, we simulate a body in the job's tree which is as close as possible to the examined part of the tree. If this simulated body is far enough away from the examined part to use its center of mass in the force calculation, all bodies inside the job can also use that center of mass. This is because these bodies are even further away from the examined part.

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 a lot of 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.

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 smallest blocks represent leaf nodes in the tree of bodies. The value of theta is 1.0 in this example.

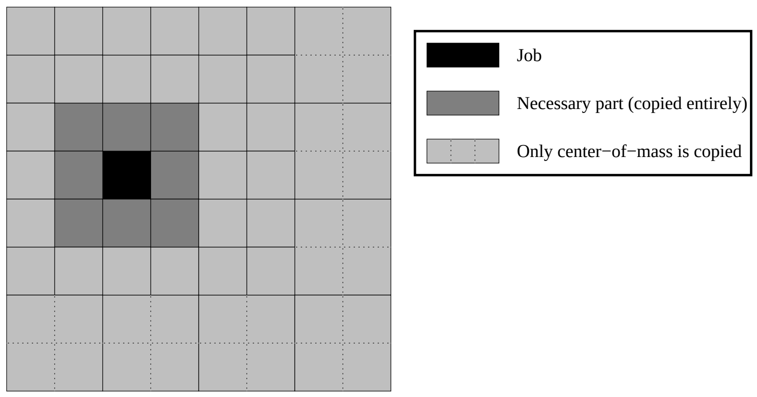
Job
Necessary part (copied entirely)
Only center−of−mass is copied

Figure 2.3: The necessary tree for 1/64 of the 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. These are indicated by the light grey blocks at the right and bottom in Figuree 2.2. The internal nodes at the higher levels which glue the tree together are also part of the necessary tree, including their center of mass information.

This center of mass information is necessary for sub jobs 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 sub job that contains 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 isn't part of the input data for a job). During the construction of the necessary tree for sub job 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 with bodies, 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 the Satin prefers large jobs over small jobs when it decides which job will be stolen if a remote node sends a steal request.

	Large job (Fig 2.2)		Small job (Fig 2.3)	
Node type	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

```
void barnesNTCIteration( Body[] bodies ) {
    TreeNode root;
    JobResult result;

    root = new TreeNode(bodies); // build tree
    root.computeCentersOfMass();

    result = spawn barnesNTCJob(root, root); // force calculation
    sync();

    updateBodies(bodies, result);
}

JobResult barnesNTCJob( TreeNode job, TreeNode necessaryTree ) {
    TreeNode subjobNecessaryTree;
    JobResult result;
    int i;

    if (! job.isLeaf()) {
        JobResult childResults = new JobResult[8];
        for (i = 0; i < 8; i++) {

            /*
             * invoke the necessary tree constructor to create the
             * necessary tree for this sub job
             */
            subjobNecessaryTree =
                new TreeNode(necessaryTree, job.children[i]);

            childResults[i] = spawn
                barnesNTCJob(job.children[i], subjobNecessaryTree);
        }
        sync();
        result = combineResults(childResults);
    } else {
        //do force calculation for the bodies in this leaf node
        result = forceCalcLeafNode(job);
    }
    return result;
}
```

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

```
TreeNode( TreeNode original, TreeNode job )
    double distance;
    int i;

    // copy original's position, size, center of mass, and cut off
    // distance (just like a standard copy constructor) (not shown)

    distance = minimalDistance(original.centerOfMass, job);
    if (distance < cutOffDistance) {
        if (original.isLeafNode()) {
            bodies = original.bodies;  // copy the bodies in original
        } else {
            // copy the necessary parts of the children of original
            // using this constructor
            children = new TreeNode[8];
            for (i = 0; i < 8; i++) {
                children[i] = new TreeNode(original.children[i], job);
            }
        }
    } // else this part can be cut off, nothing has to be copied
}
```

Figure 2.5: Pseudo code for the necessary tree constructor

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.

Figure 2.5 shows pseudo code for the necessary tree constructor. 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.

2.4 The tuple space version

In the tuple space version, we use an extension to the Satin divide-and-conquer system called the Satin tuple space. 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 put in it . This is done at the moment they are put in the tuple space.

The tuple space version uses the same hierarchical technique as the necessary tree 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

```
void barnesTupleIteration( Body[] bodies, int iteration ) {
    TreeNode root;
    String rootIdentifier = "root" + iteration;
    JobResult result;

    root = new TreeNode(bodies); // build tree
    root.computeCentersOfMass();

    TupleSpace.put(root, rootIdentifier);

    result = spawn barnesTupleJob(root, rootIdentifier); // force calculation
    sync();

    TupleSpace.remove(rootIdentifier);

    updateBodies(bodies, result);
}

JobResult barnesTupleJob( TreeNodePointer jobPointer,
                          String rootIdentifier ) {
    TreeNode root = TupleSpace.get(rootIdentifier);
    TreeNode job = root.findNode(jobPointer);
    int i;
    JobResult result;

    if (! job.isLeaf()) {
        JobResult childResults = new JobResult[8];
        for (i = 0; i < 8; i++) {
            childResults[i] = spawn barnesTupleJob(job.children[i],
                                                   rootIdentifier);
        }
        sync();
        result = combineResults(childResults);
    } else {
        //do force calculation for the bodies in this leaf node
        result = forceCalcLeafNode(job);
    }
    return result;
}
```

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

```
class Updater() {
    JobResult prevResult;

    Updater(JobResult jr) {
        prevResult = jr;
    }

    handleTuple() { // active tuple method, called at each node upon receipt
        BarnesHut.updateBodies(BarnesHut.bodies, prevResult);

        BarnesHut.root = new TreeNode(BarnesHut.bodies); //build tree
        BarnesHut.root.computeCentersOfMass();
    }
}
```

Figure 2.7: Pseudo code for the updating active tuple

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.

2.5 The active tuple version

To minimize the communication overhead of the tuple space version, we implemented another version that uses the Satin tuple space. In this version, the result of the force calculation phase is broadcast, 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.

To make this work, the bodies and the tree with bodies are replicated at all nodes. Active tuples are used to update these replicated objects. An active tuple is a special tuple. Apart from the data that is present in a 'normal' tuple, it 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 broadcast. This active tuple initializes each node. An active tuple with the force calculation result of the previous iteration is broadcast at the start of each iteration. Figure 2.7 shows pseudocode for this tuple. The active tuple first updates the bodies at each node. 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. Pseudocode for an iteration of the active tuple version is given in Figure 2.8.

```
static Body[] bodies; //these fields are updated by the active tuples
static TreeNode root;

JobResult barnesActiveTupleIteration( JobResult prevResult ) {
    Updater u; //active tuple
    JobResult result;

    u = new Updater(prevResult); //broadcast updates from previous iteration
    TupleSpace.add(u);

    result = spawn barnesActiveTupleJob(root); //force calculation
    sync();

    return result; //result will be broadcast in the next iteration
}

JobResult barnesActiveTupleJob( TreeNodePointer jobPointer,
                          String rootIdentifier ) {
    TreeNode root = TupleSpace.get(rootIdentifier);
    TreeNode job = root.findNode(jobPointer);
    int i;
    JobResult result;

    if (! job.isLeaf()) {
        JobResult childResults = new JobResult[8];
        for (i = 0; i < 8; i++) {
            childResults[i] = spawn barnesTupleJob(job.children[i],
                                                      rootIdentifier);
        }
        sync();
        result = combineResults(childResults);
    } else {
        //do force calculation for the bodies in this leaf node
        result = forceCalcLeafNode(job);
    }
    return result;
}
```

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

Chapter 3

Implementation

All code is written in the Java programming language. The Satin divide-and-conquer system [8] is used. Satin is built on top of Ibis [6]. Ibis is an efficient communication library which is entirely written 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 [7]. 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 precision factor theta and 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. This is the case in the necessary tree and the tuple space version. In these 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. Figure 3.1 shows a schematic represantation 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 merging these lists. When all jobs in an iteration of the force calculation phase are done, the root

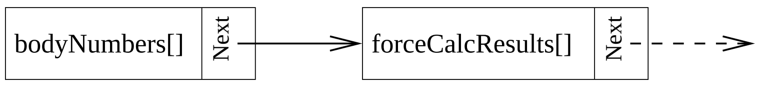

Figure 3.1: The result of a leaf job

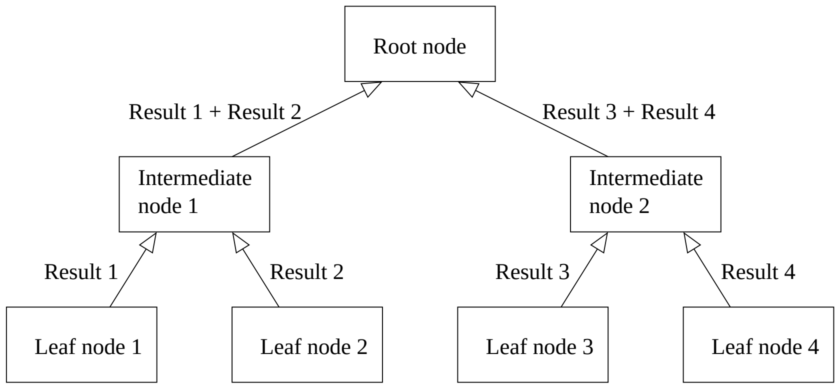

Figure 3.2: Recursively combining four jobs

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.

??? divide-and-conquer job results altijd optimizen???

The approach with linked lists is not without problems, because job results may be sent multiple times over the network. Figure 3.2 shows an example of this. The rectangles indicate compute nodes and the arrows indicate network traffic. In this figure, two jobs are generated by the root node, which are stolen by two intermediate nodes. Both intermediate nodes generate two sub jobs, which are stolen by the leaf nodes. The leaf nodes compute 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 two times over the network. 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 sub jobs.

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. It is demonstrated in

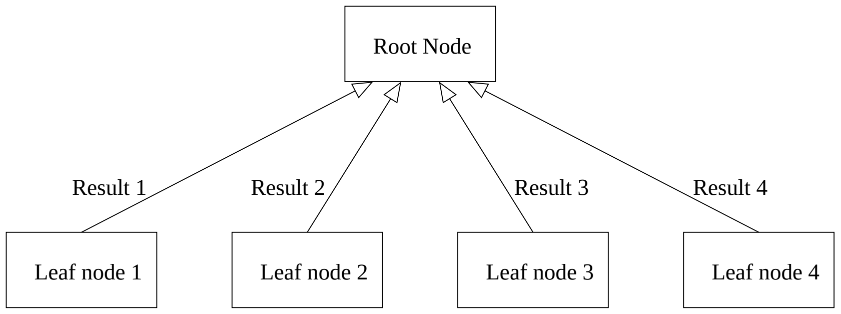

Figure 3.3: Using RMI to return the result of four jobs

Figure 3.2. However, a job is usually run at the same node as its parent. The 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 behaviour is shown in Figure 3.3. Because using the efficient Remote Method Invocation implementation in Ibis [5], 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 implementation.

All measurements are done on the DAS-2 cluster at the Vrije Universiteit. The DAS-2 system [4] consists of four clusters at four 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 [3] network interface. 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 a theta value 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 is dependend 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 behaviour resembles the behaviour of the necessary tree construction 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 again to find the next leaf node, and so on. To exploit cache behaviour, the tree with bodies is traversed in depth-first order. This way, the force calculation can be seen as being split up in different time slots. In each time slot, the force calculation for bodies that are near each other in the represented three dimensional space is done. Because the force calculation

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

for bodies that are near each other usually need the same parts of the tree, the memory cache is used efficiently this way. ??? rob hoe zeg ik dit wat handiger ???

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. 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, this version is used to compute the speed up of the parallel runs.

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 ???.??? seconds when a problem size of 2.000.000 bodies is used and theta is set to 1.0. With the same parameters, the inlined version runs for ???.??? seconds. ??? dit is de satin versie

The elimination of the use of `Vec3` objects thus increases performance by a factor greater than five???. 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 when the use of a lot of objects also incurs a lot of communication overhead.

4.2 The 'naive' version

In Figure 4.1 we show the relative speedup of the 'naive' version at two, three, four and five compute nodes, using different recursion thresholds. 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 reasonable speedup at two nodes. At three, four and five 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 at five nodes with a threshold value of five the largest difference in run time between two iterations is 33 %.

Because the run time of the iterations differs, the total run time also differs. This explains the random behaviour 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 bodies is sent over. This causes the the maximum speed up to be limited to about 1.7.

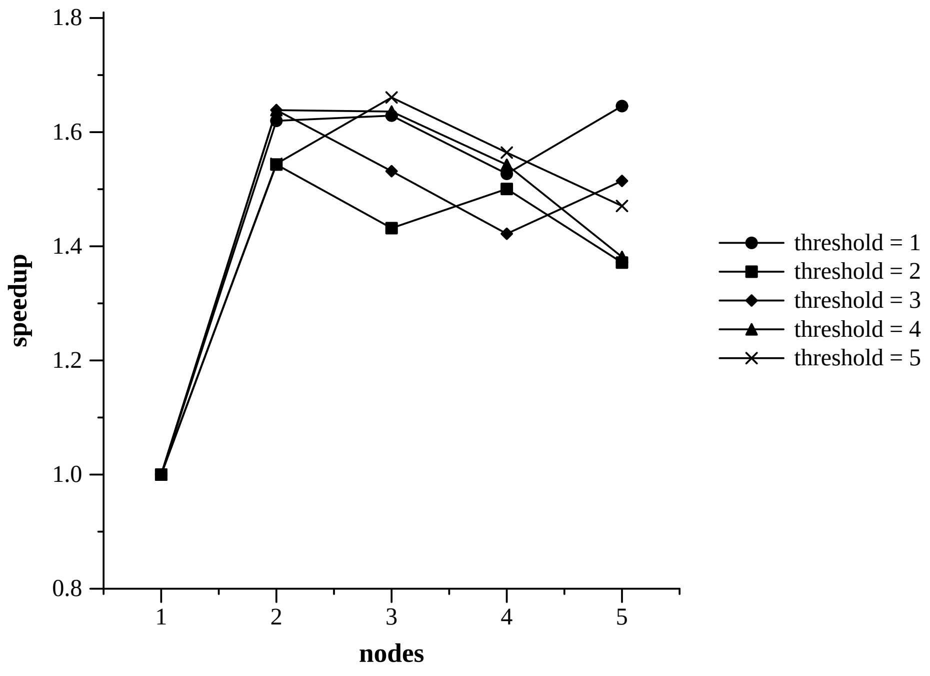

Figure 4.1: (Non-)Performance of the 'naive' version

4.3 The 'necessary tree' version

4.4 The tuple space version

4.5 The active tuple version

Chapter 5

Conclusions

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. pages ??–??, 1997.

[3] 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.

[4] The Distributed ASCI Supercomputer 2. http://www.cs.vu.nl/das2/.

[5] 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.

[6] 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.

[7] 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, 1999 ???

[8] 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.