Parallel Programming in Java: Porting a distributed Barnes-Hut implementation

E.M.Thieme
Faculty of Sciences,
Division of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

vrije Universiteit *amsterdam*

1 Table of contents

2 Introduction

Java has gained much in popularity during the past few years. Java's clean and easy to learn programming model, for example, makes it a good starting point for inexperienced programmers. Due to the cleanness of its object oriented programming model, Java is more and more being considered an attractive alternative to traditional programming languages like "C".

The simplicity of its programming model and inherent structuring of the code is one of the reasons that the scientific computing community is displaying a growing interest in Java for use in high-performance (parallel) programming [13]. Java's built-in support for distributed computing makes it attractive to use in large-scale parallel and distributed programs. Java, however, also has its drawbacks; its strict floating-point standard causes a performance gap between programs written in Java and those in traditional, compiled languages on most current architectures. Java's remote method invocation (RMI) protocol makes distributed programming possible, without having to program the communication routines explicitly. Its implementation, however, still leaves much to be desired, in terms of performance. Most of the critical communication code is interpreted, and requires extra objects to be created, resulting in performance levels of as much as a factor 35 slower than similar communication code (for example, remote procedure calls) in other programming languages, as indicated by Maassen et al. [9].

The goal of this thesis is to investigate some of the difficulties in porting parallel programs written in other languages to Java, focusing on communication aspects. As a case study, a coarse grained parallel implementation of an N-Body algorithm using the Barnes-Hut method, originally written in C by Torsten Suel and David Blackston [4], is ported to Java. Two Java implementations have been made; the first version is written for Sun's JDK 1.2, using its RMI protocol for communication. The second implementation uses Manta [3], which supports a different RMI protocol, based on the JavaParty programming model [10]. In this thesis, the first section gives a description of the Barnes-Hut method for computing N-Body problems. In section 2, the original C program is discussed; sections 3 and 4 deal with the two Java implementations. The next, section 5, gives an overview of the problems encountered, and section 6 discusses the performance of the programs on a Myrinet based cluster computer. Finally, the conclusions are presented and a look on future work is given.

3 The Barnes-Hut method

3.1 Background

The Barnes-Hut method as described in this paper is an efficient method for the implementation of classical N-body problems. N-body problems simulate the evolution of a large set of bodies under the influence of a force. N-body methods are used in various domains of scientific computing, including astrophysics, fluid dynamics, electrostatics, molecular dynamics and even computer graphics. Most N-body methods simulate the movement of the bodies in discrete time steps. These time steps are generally split up in two phases: a force computation phase, in which the force exerted on each body by all other bodies is computed, and an update phase in which new positions and velocities for each body are computed. If the force is computed directly, by taking into account all pairwise interactions, the complexity is equal to $O(n^2)$. This complexity is undesirable, as for realistic problem sizes, the system usually contains thousands of bodies, resulting in huge amounts of interactions. The Barnes-Hut method uses a hierarchical technique to reduce the complexity to $O(n \log n)$.

Hierarchical algorithms are based on the fact that many natural processes show a range of scales of interest in the available information; information from further away in the physical domain is not as important as information from nearby. Hierarchical algorithms exploit this property to optimize the amount of computation. The Barnes-Hut method takes advantage of this property as well, by approximating a group of bodies (cells[1]) that are far enough away by one equivalent particle[2] (the center of mass) during the force computation. This is illustrated in figure 1.

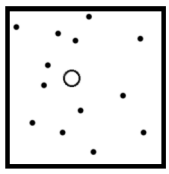


Fig. 1 A 2D cell containing bodies and its center of mass

In this paper, we discuss two implementations of an astrophysical problem using the Barnes-Hut method, where the bodies are planets and stars, and the applicable law is Newton's gravitational law. Bodies are modeled as point masses.

3.2 The sequential Barnes-Hut method

To structure the physical domain, the Barnes-Hut method adds all bodies to an oct-tree, recursively subdividing it as more bodies are added, distributing the bodies to the appropriate child nodes. The result is an oct-tree, which is more subdivided in denser

[1] In the context of the oct-tree, the terms 'cell' and 'node' are used interchangeably.
[2] The terms 'body' and 'particle' are used interchangeably throughout the paper.

areas of the domain. During the force computation phase, each body can interact directly with whole parts of the oct-tree, instead of interacting with all bodies contained in this part of the tree, if the distance between the body and the cell is large enough. Figure 2 illustrates this idea for a two-dimensional space using a quadtree. The bodies in cell B can interact directly with the center of mass of cell A, as A and B are sufficiently far separated.

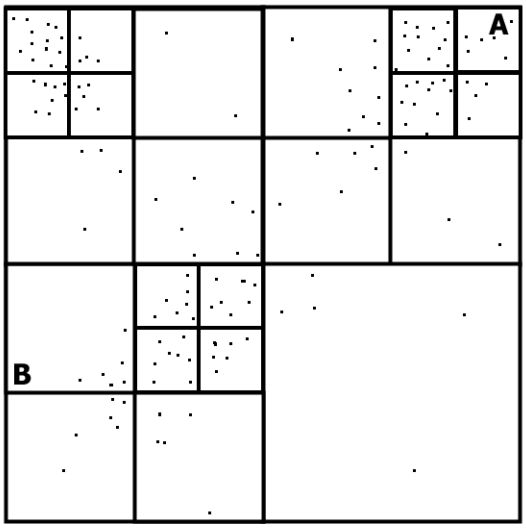


Fig. 2 a quadtree representation of a system. The quadtree is more subdivided in denser areas of the system.

In the Barnes-Hut method, as in most classical N-body methods, the evolution of the system is observed across a (large) number of discrete time steps. In the Barnes-Hut method, however, the force computation phase can be split up into three separate phases, resulting in a total of four different phases:

1. **Root cell and tree construction.** During this phase, the size of the root node of the oct-tree is computed. This needs to be done for each time step owing to its dynamic nature. When the size of the root node has been determined, all bodies are inserted into the tree, subdividing nodes if a leaf node contains more than a certain amount of bodies. This results in an oct-tree, subdivided more in denser areas of the domain.
2. **Center of mass computation.** An upward pass is made through the tree to compute the center of mass for each cell, taking into account all child cells and bodies.
3. **Force computation.** During the force computation phase, the oct-tree is traversed, for each body, starting at the root cell. For each visited node in the oct-tree, the distance from the node to the body is measured. If it is far enough away, the body will interact with the center of mass of this node, otherwise its child nodes are visited (if it is an internal cell) or it interacts with all bodies (if it is a leaf cell). “Far enough” away can be defined in a number of different ways, but usually the ratio between the sidelength of the cell and the distance from the body is used to determine this. The ratio can be adjusted to obtain a better precision, resulting in a larger computation time.
4. **Update body properties.** The positions and velocities of the bodies are updated during this phase.

The four phases above are repeated during each time step. Sing et al [5] indicate that in typical problems, more than 90% of the total execution time is spent in the force calculation phase.

3.3 Parallelism in the Barnes-Hut method

All four phases of the Barnes-Hut method contain some form of internal parallelism. The way in which this parallelism is most efficiently exploited depends on the type of implementation and the architecture of the system on which this implementation is to be used. Fine-grained, particle level parallelism for instance is possible on shared memory machines; a more coarse-grained parallel method is needed in order to achieve decent scalability on a distributed system. A detailed description of one of the many different parallel methods, used in an implementation by Suel [4] is given in this paper, which uses message passing for communication.

The method Suel uses in his implementation is based on a sender driven replicated approach, which was first implemented by Warren and Salmon [6]. This approach is based on the observation that within the force computation phase, the information in the oct-tree is used in a read-only fashion; only the forces acting upon the particles are modified during this phase. If each processor is assigned a region of the domain, and the oct-tree of each processor contains all essential global information, then each processor can run the force computation phase on its local part of the domain without any form of synchronization and communication. The tree construction phase, centers of mass computation phase and update phase do not require any communication and synchronization either, as they all operate on the local part of the domain.

This approach does require an extra phase, however, to make sure that all processors are supplied with the essential global information needed to accurately compute the forces on the local particles. Replication of (only the essential parts of) the oct-trees belonging to the other processors is used to supply each processor with its individual "view" of the global system. Subtrees from other processors containing regions that are well separated from the local region are not replicated entirely, but replaced directly by their center of mass, as the center of mass will be used in the force computation as well.

To ensure a proper balancing of the workload between the processors, a mechanism to structure the physical domain is required. Two well-known efficient techniques for structuring the domain are orb-trees[3] and costzones. Suel's method uses orb-trees, as this is more suited to message passing implementations; costzones yield better performance on shared memory machines (a thorough description of each of these techniques is given by Singh et al. [5]). A disadvantage of orb-trees is that the number of processors is restricted to powers of two, whereas the costzones scheme can use any number of processors. In Suel's implementation, every processor contains the same global orb-tree, but the processor itself is only responsible for the bodies in the region of one of its leaf

[3] "Orb" is an acronym for orthogonal recursive bisection. Orb-trees can be used to hierarchically subdivide (non-uniform) distributions into 2^n equal partitions, by starting with a root cell containing the entire distribution, which is then recursively subdivided into two child nodes with an equal cost.

nodes. The orb-tree is kept consistent using an extra phase, in which the bounds are updated. Load balancing is performed per tree level; all regions in the subtrees are adjusted to divide the bodies more evenly.

Including the extra phases, the parallel version of the Barnes-Hut method as implemented by Suel is structured as follows:

1. **Orb-tree update.** At the beginning of each time step, the global orb-tree needs to be adjusted to accommodate all bodies in the system, as their positions may have changed beyond the boundaries of the system during the last time step. This is done by having each processor determine the boundaries of its local region of the system, and exchanging those with all the others. With this information, the processors are now able to compute the global boundaries of the system. The root node of the orb-tree and any orb-tree nodes sharing a side with the root node are now updated to match the new bounds. All bodies that are not contained any longer within the local part of the orb tree are exchanged with the other processors.
2. **Load checking and balancing.** When the load imbalance surpasses a certain threshold, a load-balancing phase is required. A good way to determine the load is to assign a weight to each particle based on the number of interactions that were needed to compute forces during the last time step. The load of a processor is thus equal to the total weight of all its local particles, which is approximately the total number of interactions needed in the force computation phase of the last time step. The load checking procedure determines the minimum orb-tree level that needs to be balanced to bring the imbalance to an acceptable level. The load balancing process uses the processors responsible for the parts of the subtree that need to be balanced to determine the most efficient subspaces, which are again subdivided, until the entire subtree is balanced. A parallel median finder is used to find the separating hyperplane. Bodies crossing region boundaries are exchanged with the other processors.
3. **Root cell and tree construction.** This phase is the same as in the sequential version of the Barnes-Hut method.
4. **Center of mass computation.** This phase is the same as in the sequential version of the Barnes-Hut method.
5. **Essential tree exchange.** In this phase, each processor sends out the parts of its local oct-tree that are required by other processors during the force computation phase. This is done recursively, in a similar way as in the force computation phase, for each processor: for each visited cell, starting at the root of the oct-tree, determine if the cell is well-separated from the region of the processor which is to receive the information. If so, send its center of mass. Otherwise, visit all subcells, or send all bodies if the cell is a leaf cell. The received essential bodies and essential centers of mass are then added to the local oct-tree, as if they were part of the local domain. A distinction is made, however, between the essential bodies and the local bodies, which is required during the force computation phase. At the end of this phase, each processor has its individual version of the imaginary “global” oct-tree.
6. **Center of mass update.** This phase is the same as the center of mass computation phase, except that only the centers of mass of parts of the oct-tree need to be updated where essential tree information from other processors is added.
7. **Force computation.** This phase is the same as the corresponding phase in the sequential version of the Barnes-Hut method, except that only the local bodies need

to have the forces acting upon them computed; the essential bodies in the tree are only used in the computation process itself.

8. **Update body properties.** This phase is essentially the same as the corresponding phase in the sequential version of the Barnes-Hut method.

4 The Suel Implementation

4.1 Overview

The reference code was written in C by Torsten Suel and David Blackston [4] and uses the Green BSP communication model for communication between the processors. Rutger Hofman ported the code to the DAS [7], by creating a version of BSP layered on top of Panda [8].

4.2 Communication

The Suel code uses a SPMD style approach to perform its parallel processing. The processors communicate using BSP, a bulk-synchronous message passing mechanism [11]. This mechanism provides functions to send and receive packets, as well as a synchronization function. This synchronization function is used to structure the communication in the program into separate "supersteps", which are sections of the program between calls to the synchronization function. Packets that are sent during a particular superstep (between calls to the synchronization function) are received only during the next step. In figure 3, an example is given, which sends a message to another processor in the first step, and receives messages during the second step.

```
typedef struct {
  int source;
  char message[16];
} helloPacket;

void HelloWorld( int dest ) {

  myPacket pkt, *pktPtr;

  // Step 1

  pkt.source = myProc;
  strcpy( &pkt.message, "hello world!" );
  bspSendPkt( dest, (void *)&pkt, sizeof( pkt ) );

  // Synchronize here

  bspSynch();

  // Step 2: Packets sent during the last step can now be received

  while ((pktPtr = (myPacket *)bspGetPkt()) != NULL ) {
    printf("received messafe from %d: %s\n",
      pktPtr->source, pktPtr->message );
  }
}
```

Fig. 3 Example C code using the BSP mechanism.

Since packets are likely to be small, message combining is done to optimize the actual communication in the BSP implementation internally. The addition of message

combining makes BSP an excellent choice for coarse-grained problems and high latency parallel architectures. The BSP mechanism can be thought of as a variation of an asynchronous, packet oriented message passing system, using barriers as the synchronization primitive.

Most of the communication is performed by two specific functions, which are used multiple times throughout the code:

1. **TotalExchangeInt**. This function is used to exchange integers with other processors. It accepts an array of integers as parameter, and sends the integer at each position of the array (from 0 to the number of processors minus 1) to the corresponding processor. After synchronization, the integers from the other processors are received and placed in the array at the position corresponding to its source processor. To indicate that no communication with a certain processor is needed, the value "-1" should be set at its corresponding position in the array. A "-1" value in the array after the function was called means that no value from the processor indicated by its position was received.
2. **ExchangeBodiesLimitedIO**. The purpose of this function is to exchange bodies in the local region of the domain with other processors. This function accepts an array of integers as parameters (the same size as the local body array), indicating the new destination processor of each body in the array. First, the number of bodies that are leaving are exchanged using TotalExchangeInt. A packet is then sent for each leaving body, containing a stripped-down version of each body structure, in which only the required fields are present to minimize the data traffic. After synchronization, all bodies are received and extracted from the packets, and inserted into the local body array. The supersteps have been split up into smaller steps to protect against possible buffer overflows.

In addition to three of the phases of each time step, communication is required during initialization and finalization as well. A summary of all communication in the program is given here:

1. **Initialization**: The first processor generates all bodies, and distributes them among all processors. It sends the bodies, one at a time, to all processors (including itself), round robin style. Special care has been taken not to overflow the buffers used in the BSP implementation, by breaking the sending process into smaller supersteps, as large quantities of bodies have to be buffered.
2. **Orb-tree update phase**: To determine the global boundaries of the orb-tree, the processors need to exchange their local boundaries. Each processor determines the minimum and maximum values in each dimension of the local body positions, and sends it using an array of 6 doubles to all processors. After synchronization and receipt of all min-max packets, the global minimum and maximum can be computed. Bodies that are not part of the local region anymore due to their displacement during the last time step need to be migrated to the processors responsible for the part of the domain in which they are now located. The function ExchangeBodiesLimitedIO as described above is used for this purpose.
3. **Load checking and balancing phase**: The load balancing function is the most complex function in the code, as far as communication is concerned. For each orb-tree level that needs to be balanced, multiple supersteps are needed. First, each processor needs to know the weight of the bodies in the subtrees, which is determined

using the TotalExchangeInt function and adding the resulting weights. A new median is then determined using the findMedian function. When the median is known, the local bodies that need to be moved to the subtree on the other side of the median can be exchanged. This is done using the ExchangeBodiesLimitedIO function. To reconstruct all orb-tree levels below the current level that are affected by the change of the median, another call to TotalExchangeInt is needed in order to broadcast the new median for the subtree to the other processors. The findMedian function implements a parallel median finder, which uses all processors in the subtrees of the orb-tree level being balanced. The function contains a for loop iterating across the number of bits used to shift the positions, communicating twice during each iteration. A part of the local array of sums is sent to each other processor in the subtree. After synchronization, the received subsets are added to the local subset. The local subset is then exchanged with the other processors in the subtree, to complete the local array of counts.

4. **Essential tree exchange phase**: The function TransmitEssential takes care of sending the essential trees to the other processors. It sends the centers of mass and bodies where needed directly, using special packet structures, to minimize the amount of data that has to be sent. After synchronization, the bodies and centers of mass are extracted from the packets and inserted into the local oct-tree.
5. **Finalization**: After the last time step, the bodies may have to be printed, or possibly written to an output file. As the bodies are still divided between all processors, they have to be gathered somewhere to be able to process them collectively. This is done in the finalOutput function, in which each processor sends all of its bodies to processor 0, which receives the bodies and puts them into an array to process them later on. Several smaller supersteps are used here to avoid buffer overflows.

4.3 Comments on the code structure

The code itself is written in a typical “C” manner. Dynamic memory allocation has been avoided by putting all sets of structures into arrays, with extra space allocated to allow them to grow within reasonable bounds. The main advantage of this method is that it prevents a large number of malloc calls for relatively small amounts of memory during each timestep. Both the orb-tree and oct-tree are laid out in memory this way; they use indices into the arrays instead of pointers to identify their child nodes.

During the initialization, a set of bodies is generated if no input file is used. The set of bodies is generated using an implementation of the Plummer model for galaxies. For practical reasons, a pseudo-random number generator is used which generates the same set of “random numbers” for each execution of the program. This feature facilitates debugging, as each run using the same parameters generates the same output.

5 The Sun JDK RMI based Java implementation

5.1 Overview

A Java version of the Suel code has been written to evaluate the applicability and performance of both Java's built-in remote method invocation (RMI) protocol [12] and the JavaParty RMI protocol [10] as used in the Manta compiler, in coarse-grained parallel problems. In this section, the Sun JDK RMI based version is discussed; the next section discusses the differences between the Sun JDK RMI and Manta RMI versions.

Both Java versions of the program are designed to be run on multiple nodes on the DAS [7], but the Sun RMI based version can easily be adapted to other distributed environments. The program uses the Fast Ethernet network of the DAS for communication, as it is not yet possible to use the Myrinet network of the DAS for SUN JDK RMI calls.

5.2 Communication

Java's lack of pointer arithmetic, multicasting and asynchronous communication calls for a different approach than to just implement the BSP routines on top of RMI. To achieve acceptable performance, message combining is essential. Without message combining, Java's lack of asynchronous communication mechanism would implicate that a (blocking) RMI call has to be made for each call to the bspSendPkt equivalent. As even simple RMI calls have a huge overhead (see Maassen et al [9]), this solution leads to unacceptable results. An array-based approach is used to limit the number of RMI calls to an acceptable number for the equivalent Java code of each superstep in the original C code; however, the flexibility of the packet-based approach is lost.

To be able to communicate point-to-point with the other processors, each processor should have access to at least one remote object of each other processor, on which it can invoke methods. To keep the program structure clean and communication as simple as possible, only one remote interface is defined, which handles all communication. This interface, "Processor", defines all methods that may be called by remote processors. This interface is shown in Figure 4.

The communication code is located in the class "ProcessorImpl", which implements the Processor interface. Each program creates a local ProcessorImpl object and registers this at the local RMI registry at startup. The processors then locate the ProcessorImpl object at processor 0, on which they call the barrier function as soon as the object is available. When the processors leave the barrier, they can be sure that all processors have started, and have a ProcessorImpl object registered at their local RMI registry. After each processor has obtained references to all remote ProcessorImpl objects, the barrier function is called again, to wait until all processors have obtained all remote ProcessorImpl objects. An overview of the communication is given in figure 5.

```
package NBody;

import java.rmi.*;
import java.lang.*;

interface Processor extends java.rmi.Remote {

  void barrier() throws RemoteException;

  void setBodyCount( int Count ) throws RemoteException;

  void setMinMax( int Source, vec3 Min, vec3 Max ) throws RemoteException;
  void setTotalExchangeInt( int Source, int Value ) throws RemoteException;

  void setExchangeBodies( int source, Body Bodies[] ) throws RemoteException;
  void setExchangeIntArray( int source, int array[], int offset, int size ) throws
RemoteException;

  void setEssential( int Source, int bCount, Body [] b, int cCount, CenterOfMass [] c )
throws RemoteException;

  void setEssential( int Source, int bCount, double [] bp, int cCount, double [] cp )
throws RemoteException;
 }
```

Fig. 4 The Processor interface

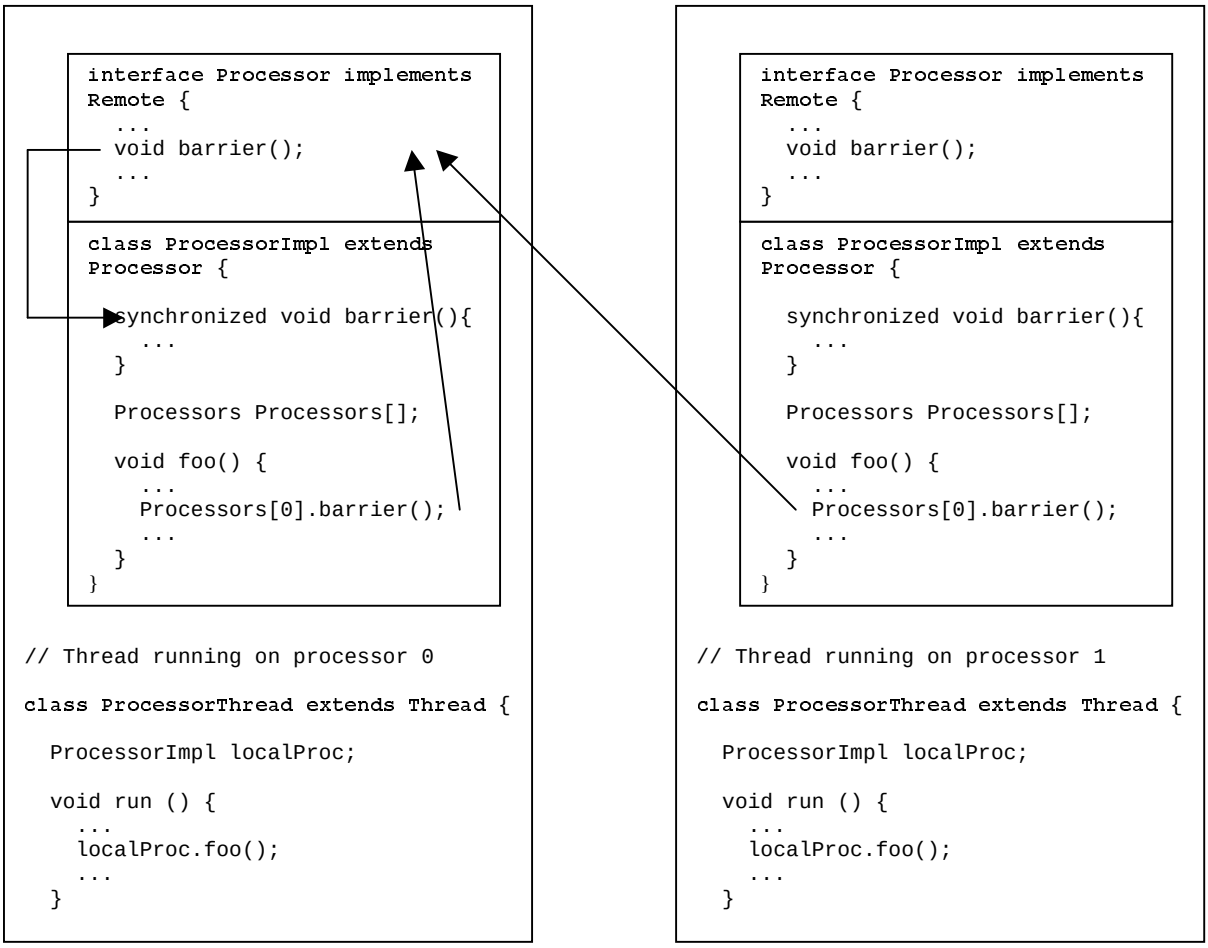


Fig. 5 Communication in the Sun JDK 1.2 version using RMI, running on 2 processors

The Java code simulating the supersteps makes use of a barrier to synchronize all processors. The code used for the barrier is displayed in Figure 6. The code makes use of Java's wait/notify mechanism, by having all processors call the synchronized barrier method on processor 0. All processors except the last one will call the wait function; the last one entering the barrier will wake them up by calling the notifyAll function.

```
public synchronized void barrier()  throws Exception {

  if (myProc==0) {

    if (SyncCounter++<(ProcessorCount-1)) {

      try {
        wait();
      } catch ( InterruptedException e ) {
        System.out.println( "barrier: Caught exception: " + e );
      }

    } else {
      SyncCounter = 0;
      notifyAll();
    }

  } else {

    // call the barrier function on proc 0

    Processors[0].barrier();
  }
}
```

Fig. 6 the barrier function

To offer mechanisms to handle the diverse communication patterns present in the Suel code, several functions have been created that provide enough flexibility to carry out all communication. These functions are methods of the class ProcessorImpl; as they are not defined in the Processor interface, they can only be called locally.

1. **TotalExchangeInt**: This function has exactly the same purpose (and syntax) as its "C" equivalent; it accepts an array of integers as parameter, which is exchanged with all other processors. The implementation of the Java version, however, is quite different. In the Java implementation of the TotalExchangeInt function, each processor calls the interface method setTotalExchangeInt on all remote processors. This method takes two integer parameters: an index and a value. The method stores the value at the specified index in the receiving integer array at the remote processor. After all integers have been set, the barrier function is called for synchronization and the received integers are copied back into the original array.
2. **ExchangeBodies**: This function is comparable to the ExchangeBodiesLimitedIO function in the Suel code. It takes an array of integers as parameter, which indicate the new destination for the body in the local array at that index. It operates quite differently as well; first, the array is iterated to count the number of bodies migrating to each processor. Using these numbers, a new array is created for each remote processor, exactly large enough to hold all bodies that move to the processor. The

index array is again iterated over, now adding the bodies to the newly created arrays. The arrays are set at the remote processors by invoking on each remote ProcessorImpl object the setExchangeBodies method, passing the array containing the references to the bodies moving to this processor. The bodies that remain at the local processor are also handled by this function, but in that case, setExchangeBodies is called locally, to circumvent the RMI processing overhead. The barrier method is then called, to synchronize all processors. The next step reconstructs the local body array by concatenating each received array.

3. **ExchangeMinMax**: This function is used to compute a global minimum and maximum, by exchanging all local boundaries. It is not present in the Suel code, as it uses BSP directly for this purpose. It is based on the same idea as the TotalExchangeInt function; it invokes a method (setMinMax) on each remote ProcessorImpl object, this time setting a pair of vec3 objects, instead of an integer. The vec3 objects, containing minimum and maximum values in the x, y, and z directions, are received in a local array. After synchronization, the global minimum and maximum are computed; the values can be retrieved by calling the getMinMax function on the local ProcessorImpl object.
4. **ExchangeIntArray**: The findMedian function in the original Suel code uses BSP code to partially exchange an array of integers. To implement this in Java, extra object creation and copying would be required. As the actual array used in the Suel code is relatively small (128 elements), a decision was made to send the complete array, including an offset and the size of the required fragment. In the Java code, special functions are provided to set the fragment offsets and sizes for each remote processor. The array itself is passed as a parameter to the ExchangeIntArray function. This function sets the array, offset and fragment size by invoking the setExchangeIntArray function on the remote ProcessorImpl objects. The received arrays, offsets and fragment sizes can be retrieved by calling respectively the getExchangeIntArray, getExchangeIntArrayOffset and getExchangeIntArraySize methods on the local ProcessorImpl object.
5. **ExchangeEssential**: The ExchangeEssential function is used to perform the communication during the essential tree transmission phase. The Suel code uses BSP directly, but the transmission and receipt of the essential information, required to reconstruct the trees, is spread out across multiple functions. To keep the code structured and simple, the Java code provides methods to buffer the information, to exchange it and retrieve the information when required. The essential information is sent in two forms: bodies and centers of mass. Both are represented by different objects; they need to be buffered separately, although the transmission can be done using the same RMI. Buffering is done using the sendEssentialBody and sendEssentialCenterOfMass funtions. After the local oct-tree has been traversed for each remote processor, and all essential trees are buffered, the ExchangeEssential method can be invoked. The exchange mechanism works more or less the same as the other exchange functions mentioned above, by invoking a method (setEssential) on each of the remote ProcessorImpl objects. A number of different implementations of the buffering/exchange mechanism have been implemented, which require two versions of the setEssential method to be present in the Processor interface. Performance issues are the main reason to experiment with multiple implementations, as the essential tree exchange phase appears to be the bottleneck of the program (this will become clear later on, see the section on performance). The different implementations are described below:

a. The first implementation is the most obvious; both sendEssentialBody and sendEssentialCenterofMass add the object to the end of a pre-allocated array (created during the initialization), and increment a counter that keeps track of the number of elements in the array. Different arrays and counters are required for each remote processor. The ExchangeEssential function uses the setEssential(int Source, int bc, Body [] b, int cc, CenterOfMass [] c) version to set the complete arrays and their number of elements at the remote ProcessorImpl object. This implies a serialization, traffic and object creation overhead, as a part of the array is not used. This part can be large, due to the fact that far less communication is required with remote regions of the domain, and more communication is required with processors responsible for adjacent regions in the domain. The setEssentialMethod stores the counters and references to the arrays; the stored items can be looked up when needed using the getEssentialBodies and getEssentialBodyCount for the bodies, and getEssentialCenterOfMass and getEssentialCenterOfMassCount methods for the centers of mass. This implementation will be used by default.
b. The second implementation is a variant of the first, using extra arrays, dynamically allocated to fit exactly the required number of elements. This requires extra object allocation on the sending side (an extra array for each remote processor, every iteration), but implies less network traffic and less serialization overhead. The difference lies in the ExchangeEssential method, where the extra array allocation and copying takes place. This implementation is used if the "-trim-arrays" command line argument was specified.
c. The third implementation is the most radical, as the serialization is performed explicitly by the program itself in this implementation. It exploits the fact that all required fields of the Body and CenterOfMass objects can be broken down to doubles, and arrays of doubles (like all simple types) have the advantage that they don't need serialization on their elements. In fact, an array of doubles can in theory be passed on directly to the network layer. The sendEssentialBody and sendEssentialCenterOfMass copy the required doubles from the objects to an array of doubles (pre-allocated during initialization). The second variant of the setEssential interface method is used to transmit the arrays of doubles and the number of elements to the remote ProcessorImpl objects, which reconstruct the arrays of Bodies and CentersOfMass. The objects in these arrays are pre-allocated during initialization, which imposes memory overhead, but requires less object creation during run-time. This implementation is a little harsh, as it does away with one of the main advantages of Java's remote method invocation paradigm, but indicates the performance penalties of serialization and lack of pointer arithmetic. This implementation is used if the "-serialize" command line argument is specified.

Variations of the above methods, especially the third, can be used to tune performance a little more.

5.3 Comments on the code structure

The oct-tree code uses a typical Java structure, using dynamic object creation instead of a pre-allocated array of tree nodes as used in the original C-code. This structuring does not impose a noticeable performance penalty, as the tree traversal code is quite efficient, and not the most time-critical. There are a number of exceptions, most of which are formed by the replacement of vec3 objects by three separate doubles. Some of the orb-tree code on the other hand is quite similar to the original “C” code. This has been done deliberately to facilitate debugging, which proved to be cumbersome nevertheless. Since the original code uses a large set of global data, a GlobalData class has been created to give some of the objects access to the required data. This is typically not a good way to start writing a Java program from a software engineering point of view, but it proved to be a convenient way to get around a lot of parameter-passing annoyance.

6 The Manta RMI based version of the Java Code

6.1 Overview

The Manta project [2,3,9] is a native code generating Java compiler designed to optimize polymorphic RMIs. The generated RMI code reaches performance levels similar to those of user level RPCs, a factor 36 improvement over Sun's JDK 1.1.4. Manta uses its own RMI protocol, implementing the same programming interface as JavaParty [10]. Manta's RMI protocol implementation, layered on top of Panda [8], is optimized for efficient communication, whereas the Sun JDK RMI protocol has been designed to be flexibile and is much more inefficient in terms of raw performance.

6.2 Code structure

Because of the difference between the Sun JDK RMI protocol and Manta's JavaParty based RMI protocol, a special version of the Java program had to be created. Most of the code could be reused however; only minor changes had to be made to the BarnesHut, ProcessorThread, GlobalData and ProcessorImpl classes. The Processor interface was left out for practical reasons.

The Manta RMI protocol is based on a special language extension, the "remote" keyword, as introduced by JavaParty [10], which can be added to class declarations indicating that the class can be called remotely. The runtime system takes care of the distribution of remote objects, which are automatically distributed across the available processors, unless explicitly told not to. For simplicity, the only remote class in the Manta variant of the Barnes Hut application is the ProcessorThread, which has been renamed to ProcessorThread_Manta to distinguish it from the RMI version. It does not contain the remote keyword in its class declaration, however; the class is derived from the RemoteThread class, and inherits its remoteness from its superclass. The array of Processor interfaces, used in the ProcessorImpl, has been replaced by an array of ProcessorThread_Manta objects. The ProcessorThread_Manta class has been extended with "stub" code for the methods defined in the Processor interface, which calls the corresponding method on the local ProcessorImpl object. This is illustrated in figure 7.

During initialization, the ProcessorsThread_Manta objects are created on the right processor by explicitly setting the target processor. When all threads have been created, they are updated by setting references to the other ProcessorsThread_Manta objects using the setRemoteProcessor function. At the moment the threads are started, they already have access to all remote objects.

The runtime system of Manta contains a very fast barrier implementation: RuntimeSystem.barrier(). It is used by the default in the Manta version, but can be overriden by the "–java-barrier" commandline option.

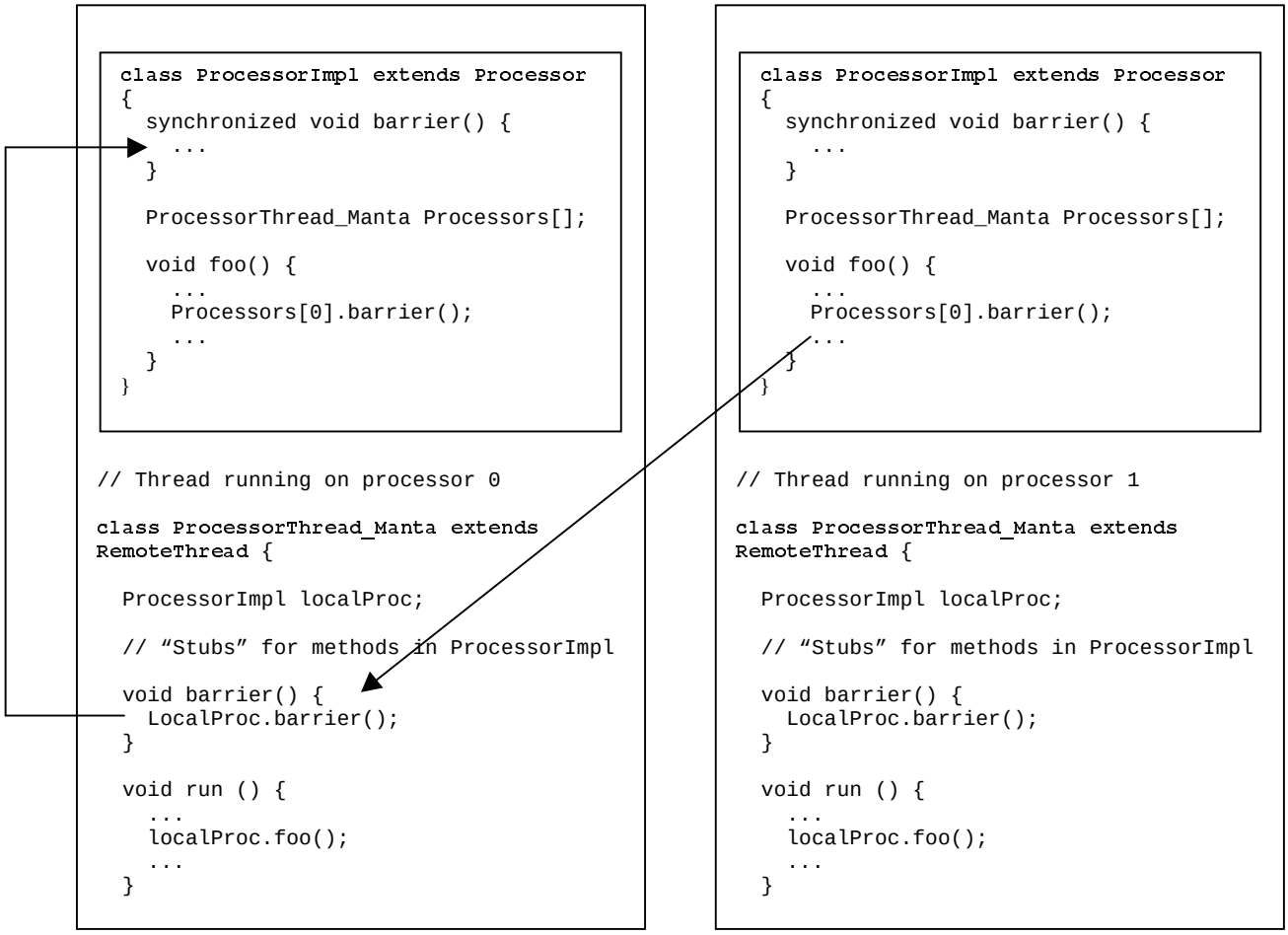


Fig. 7 Communication in the Manta version using RMI, running on 2 processors.

7 Problems porting (distributed) C code to Java

7.1 Floating point issues

One of the differences between C and Java is the floating-point handling. Java has a strictly defined floating-point model (using 64-bit precision for doubles), whereas the floating point precision of C usually depends on the CPU on which the software is run. On the x86 architecture of the DAS [7] for example, the floating-point precision for doubles is 80 bits. The difference is in practice almost negligible in terms of precision, but this difference can accumulate as results of operations are being used over and over again.

This difference in floating point handling also impedes the floating-point performance; to ensure floating point compatibility with the Java defined standard, the values have to be truncated to the 64 bit precision for every floating-point operation in hardware, making effective use of the floating point pipeline virtually impossible.

Throughout the Suel code, doubles are tested for equality by comparing them directly. In Java, checking two doubles for equality can't be done directly, as really small differences in both values may become visible, and possibly provide unexpected results. To circumvent this, the Barnes-Hut Java code uses epsilon values when comparing doubles. At one place in the original code, where the maximum of three (possibly equal) doubles was determined, inconsistencies in the results appeared. The C code had to be modified by including an epsilon value in the comparison to make sure the outcome of the result is consistent with the Java version.

7.2 Lack of pointer arithmetic

An advantage of C code is the use of pointers, and the miscellaneous operations possible on these pointers. Pointer arithmetic proves especially useful in dealing with arrays, which introduces significant overhead in object creation and copying when porting C code directly to Java, for instance if parts of arrays are addressed as arrays themselves. The ExchangeIntArray function is a good example; in the C program, parts of the array are adressed as separate arrays and sent to the remote processors. In the ExchangeIntArray function in the Java program, the whole array is sent to each remote processor, as the array size is relatively small.

7.3 Overhead of temporary object creation

Another issue that should be kept in mind when porting programs to Java is the fact that creating temporary objects is usually expensive. In C, a structure can be declared as a local variable to a function, so it is allocated directly on the stack. This structure can be used throughout the function, possibly passing a pointer to it on to other procedures (this pointer is valid only in the local function!). The allocation of this structure does not impose a performance penalty, as it is just a matter of incrementing the stack pointer during the stack frame creation. In Java, however, temporary objects can only be created by calling the *new* function, possibly invoking a constructor as well. These objects do

have the advantage that they are not restricted to local use, and can be used anywhere in the code. Sun's JDK handles the allocation of temorary local objects relatively efficiently by means of escape analysis, whereas Manta currently does not. This becomes painfully clear during dynamic object creation in recursive functions; replacing a vec3 object by three separate doubles in a recursive function somewhere in the Manta compiled (sequential) Barnes-Hut code improved the performance by a factor of two.

7.4 Garbage collector issues

In the Barnes-Hut application, support for explicitly calling the garbage collector has been added. As the parallel version is run on multiple machines, all running their own runtime system, each processor runs the garbage collector whenever it thinks it is the right moment to do so. This may produce the unwanted side effect of one processor delaying the entire application, as garbage collection may take a noticeable amount of time. For example, if the first processor calls the garbage collector during the first iteration, the second somewhere during the second iteration, and so on, a lot of time is wasted, as all non-garbage collecting processors are most likely idle during the garbage collection of the other(s). The garbage collecting process can be made more deterministic by calling the garbage collector explicitly every *n*-th iteration, where *n* can be tuned for maximum performance (this depends of course on the problem size and available memory). This effectively synchronizes the garbage collection among the different processors, so every *n*-th iteration will take a little longer

7.5 Bugs

A number of bugs were encountered in the Sun's current JDK 1.2 during the port of the Barnes-Hut application, which severely limit its applicability in the field of distributed computing:

1. With Sun's JDK 1.2 on Linux with the JIT enabled, the floating-point results were sometimes completely random, running the multithreaded variant of the Barnes-Hut RMI implementation. This is most likely a bug in the JIT, as running Java without the JIT showed correct results.
2. For some reason, it was not possible to run the JDK 1.2 version on more than 4 processors using Sun's RMI protocol (the program would cause a segmentation violation if more than 4 processors were used).

8 Performance

8.1 Overview

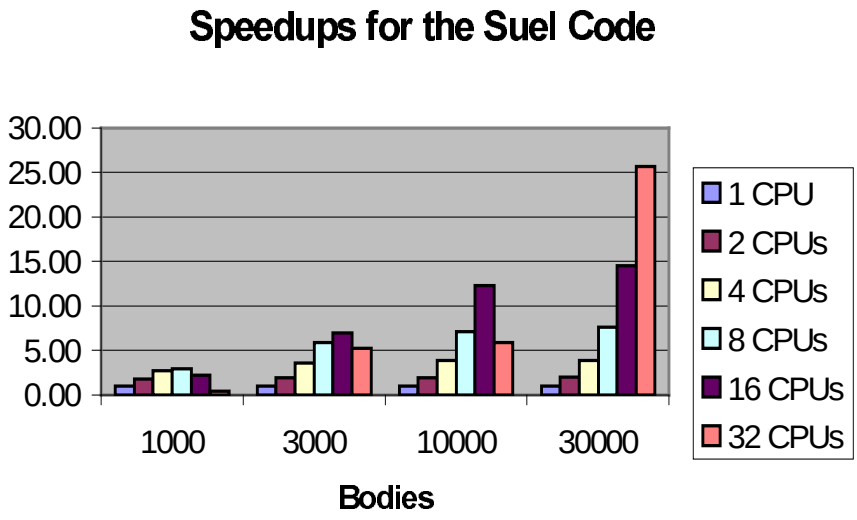
In this section, a performance comparison is given between the original C version, the Java RMI version and the Manta Java version. The experiments are run on the DAS (on a local cluster), consisting of 128 nodes. Each node is equipped with a 200 MHz Pentium Pro Processor, 128 MB of EDO-DRAM, and a 2.5 GB IDE harddisk. Two different networks connect the nodes: a switched 1.28 Gbit/sec Myrinet SAN network and a 100 Mbit/sec Fast Ethernet network. The operating system used is Red Hat Linux, version 2.0.36. All numbers were obtained by specifying the following command line options for the Barnes-Hut program: "-M10 –tstop 1.0 –dtime 0.025 –eps 0.000000625". These parameters make the program run 40 iterations, using a maximum of 10 bodies per leaf node, and an epsilon value of 0.000000625. The Java versions of the program use the extra command line options "-trim-arrays –gc-interval 5" unless noted otherwise, which specify explicit garbage collection every 5 iterations and an extra array copy in the communication code. The C program and the Manta Java version are run on the Myrinet network; the Sun JDK version is run on the Fast Ethernet network. The numbers and profiling information are acquired by accumulating the time spent in each phase for all iterations. The timing information is printed at the end of the program execution.

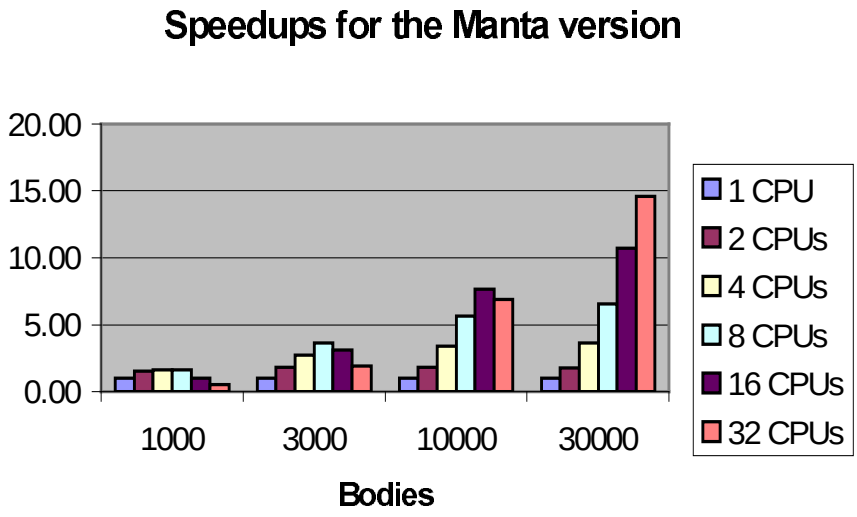
8.2 Speedups

In the test runs, the number of bodies is kept relatively low, because a problem with the Java virtual machine of the JDK 1.2 prevents it from running with large amounts of bodies. The JDK RMI version of the code is limited to only four processors, as the RMI version crashes when run using JDK 1.2 and more then four processors are used. Tests using Manta indicate that the source of this problem is also the Java virtual machine. A drawback of the relatively low amount of bodies is that decent speedups are only possible on few CPUs for Manta. The Suel code already reaches a speedup of 7.62 on 8 processors using 30000 bodies, whereas the Manta version reaches a speedup of 6.54 on 8 processors in this case. Figure 8 illustrates this. The numbers are given in Appendix A.

The profiling information generated by the Java programs as shown in figures 9 and 10 indicates that the bottleneck in both Java implementations is the relatively poor performance of the essential tree exchange phase, and, when large sets of bodies are used, the load balancing phase as well. This performance problem is mainly caused by two important aspects of both the Sun JDK RMI protocol and the JavaParty RMI protocol; both protocols are synchronous, and require serialization of objects.

The BSP implementation of the Suel code uses asynchronous communication, whereas both Java versions use synchronous communication to distribute the tree information. This is one of the reasons for the poor performance of the Java code on larger sets of processors and small sets of bodies. If asynchronous message passing has a two way latency of n microseconds, the two way latency of the total synchronous message passing is equal to n times the number of processors. As larger amounts of data are sent, the influence of the latency on the message passing diminishes and the influence of the actual





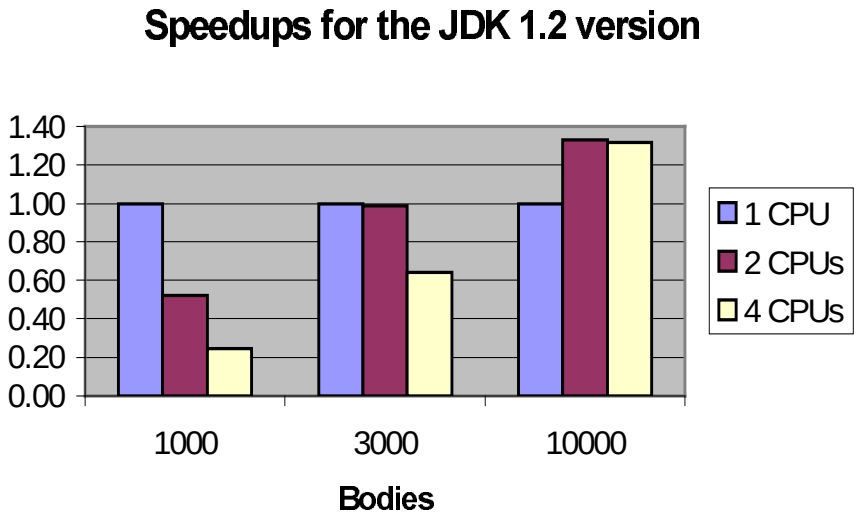


Fig 8. Speedups Charts

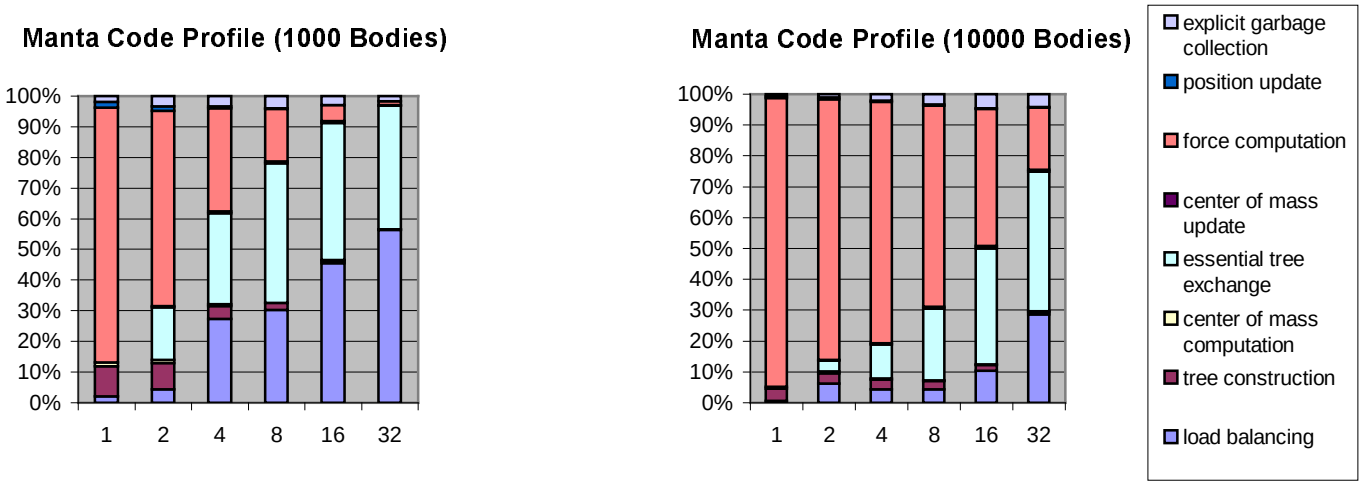


Fig 9. Manta Code Profile

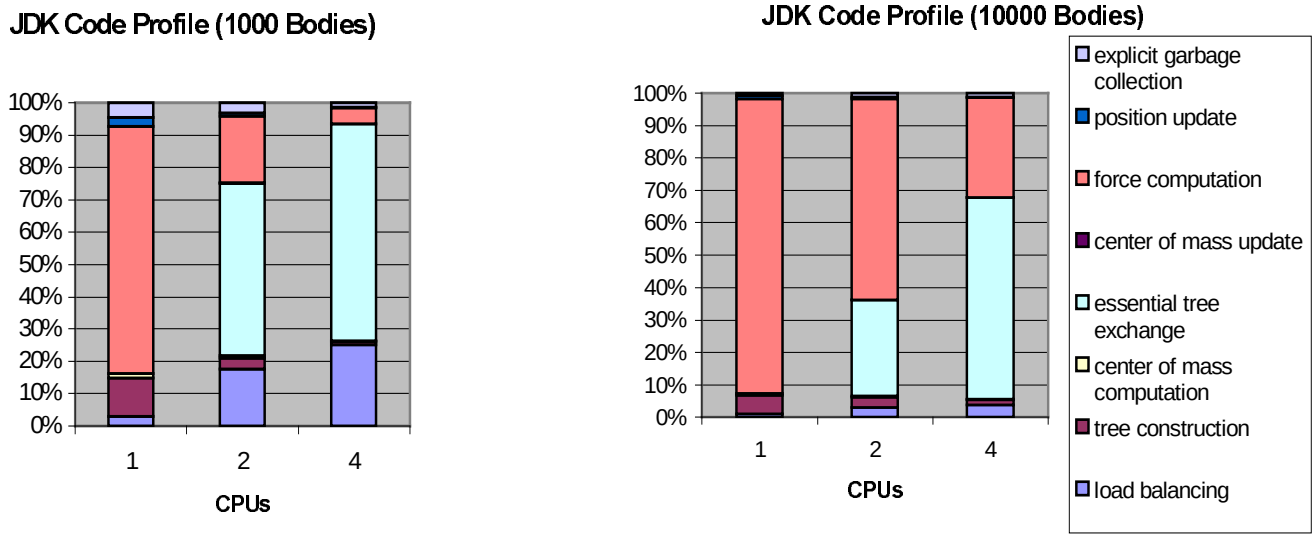
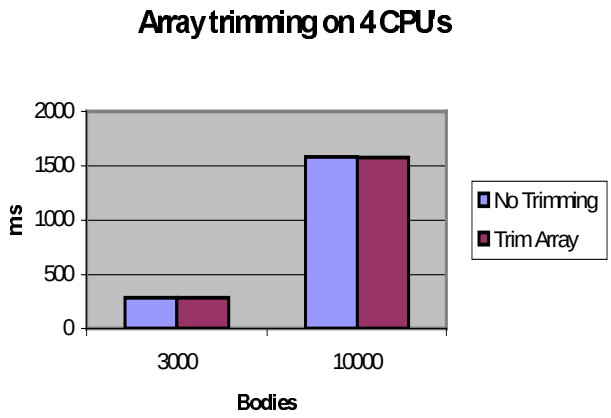
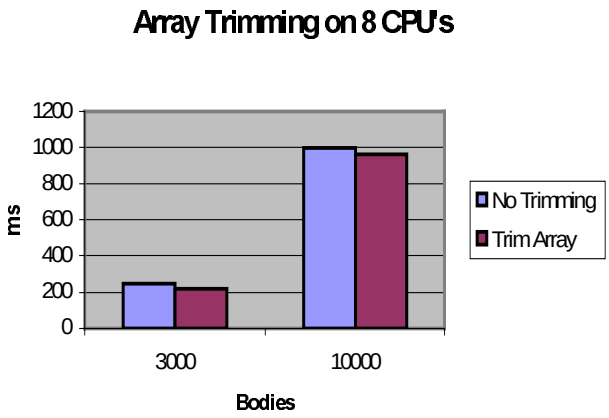


Fig 10. JDK Code Profile





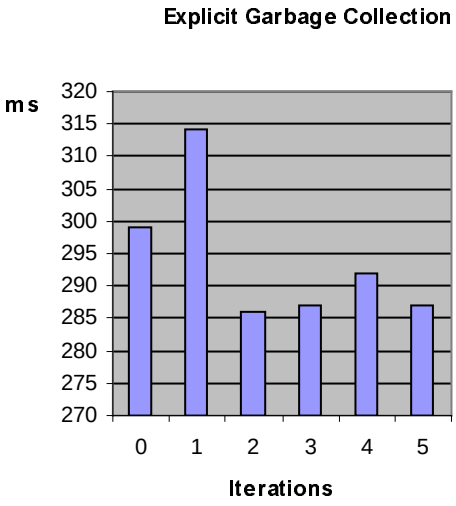


Fig 11. Optimizations

data transmission becomes more significant, which explains the better performance using larger sets of bodies.

Another source of overhead is the serialization required to send objects using RMIs. This is especially true for the Sun JDK version, as the Manta compiler generates optimized native serialization code. This is illustrated by the use of a slightly modified communication mechanism (explained on page 15) which performs its own serialization during the essential tree exchange phase, giving a 5-10 percent overall performance increase when run on 4 processors.

The Sun JDK RMI version crashes when run using as little as 30000 bodies. The speedups obtained by the Sun JDK RMI version are very poor; this is mainly caused by the overhead of the RMI protocol [9], which is partially interpreted and requires the creation of several temporary objects in its critical path. This limits the use of the Sun JDK RMI protocol to applications that involve very coarse-grained parallel problems where its high latency does not have a large influence on the overall performance. The performance of the force computation phase is practically the same in both Java implementations, indicating the heavy use of floating-point code in this phase, which cannot be optimized much by Manta.

8.3 Performance difference between and C / Java

In the essential tree exchange phase, the difference between Manta and JDK 1.2 is relatively small in terms of performance, and is about a factor 3 slower than the compiled C code. This is a direct result of Manta's strict adherence to the Java floating point standard; the load / store required for converting the floating-point values to 64 bits has a tremendous performance impact, hindering effective usage of the floating point unit. It is worth noting, however, that the floating-point performance does scale linearly with the number of CPUs. In the other phases, the performance of Manta starts approaching the C performance.

8.4 Optimizations

Explicitly calling the garbage collector, which synchronizes the garbage collection process between all processors, may increase the performance by as much as 5%, as shown in figure 11. However, as large amounts of bodies are used, garbage collection may still be done implicitly during the essential tree exchange phase, due to the high memory overhead caused by the creation of array objects. The trim-array option, which minimizes the data traffic between the processors during the essential tree exchange phase (discussed on page 15) may increase performance by another 5%, as shown in figure 11.

The actual performance gain for both optimizations depends highly on the number of processors and the number of bodies.

9 Conclusion and future work

This thesis describes the process of porting a distributed N-Body algorithm (Barnes-Hut) from C to Java, and gives a discussion of the problems encountered during the process. Two versions of the Java program were made, using two different RMI protocols (the Sun JDK RMI protocol and the Manta RMI protocol, which is based on the JavaParty model) for communication.

The performance of the Barnes-Hut application in Java is promising. Although the Sun JDK version does not perform to well, the Manta version produces acceptable speedups. Using 8 processors and a system consisting of 30000 bodies, the Manta version reaches a speedup of 6.54, in which case the C code reaches a speedup of 7.62. The scalability of the Java programs, however, is worse than the scalability of the original C program. This is caused by the lack of asynchronous communication in the two RMI protocols, which has a dramatic impact on the total latency in the essential tree exchange phase, when large amounts of data have to be transmitted. The absolute performance of the Java versions is reduced by the poor floating performance, imposed by the strict adherence to the Java floating-point standard. This problem is widely recognized as one of the most important deficiencies of Java, and is currently being looked into.

Distributed programming using Java and RMIs for communication is feasible for coarse-grained parallel programs. However, the performance depends heavily on the structure of the communication code, in particular when using the Sun JDK RMI protocol. A communication aspect that may exert a large influence on the performance is the use of arrays of objects as parameters of RMIs, as the transmission of large arrays may affect the latency to a large extent because of network load and serialization overhead. Array copying and trimming may improve performance, if the reduction of the network traffic justifies extra object creation costs. The lack of asynchronous communication also has an impact on the latency when using larger sets of processors in a total point-to-point communication scheme. Synchronizing the garbage collection by explicitly calling the garbage collector may improve performance.

There is still some more performance to be gained, especially on larger systems. The use of threads to perform the actual RMIs may improve the latency when larger sets of processors, or large datasets are used.

10 Acknowledgements

I would like to thank Ronald Veldema, Rob van Nieuwpoort, and Jason Maassen, for their work on Manta and support, as well as the coffee breaks and discussions, Henri Bal and Aske Plaat for their guidance and patience, and John Romein for his support on the DAS hardware. I would also like to thank Rutger Hofman for his comments and explanations on the original code, and Ceriel Jacobs for his contributions to the Manta project.

11 References

[1] R. Veldema. *JCC, a native Java compiler*. Master's Thesis, Vrije Universiteit Amsterdam, August 1998. Online at: http://www.cs.vu.nl/~rveldema

[2] J. Maassen and R. van Nieuwpoort. *Fast Parallel Java,* Master's Thesis, Vrije Universiteit Amsterdam, August 1998. Online at: http://www.cs.vu.nl/~jason

[3] Manta, A native Java compiler, 1999, http://www.cs.vu.nl/manta

[4] D. Blackston and T. Suel. *Highly Portable and Efficient Implementations of Parallel Adaptive N-Body Methods*. SC'97: High Performance Networking and Computing, Proceedings of the 1997 ACM/IEEE SC97.

[5] J. Pal Singh, C. Holt, T. Totsuka, A. Gupta, J.L. Hennessy. *Load Balancing and Data Locality in Adaptive Hierarchical N-body Methods: Fast Multipole, and Radiosity.* Journal of Parallel and Distributed Computing, 27(2), pages 118-141, June 1995.

[6] Michael S. Warren, John K. Salmon. *Astrophysical N-body simulations using hierarchical tree data structures*. Supercomputing '92, pages 570-576, 1992.

[7] The Distributed ASCI Supercomputer, 1998, http://www.cs.vu.nl/das

[8] The Panda portability layer, 1999, http://www.cs.vu.nl/panda

[9] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat: *An Efficient Implementation of Java's Remote Method Invocation,* Proc. Principles and Practice of Parallel Programming'99, pages 173-182, Atlanta, GA, May 1999.

[10] M.Philippsen and M.Zenger. *JavaParty – Transparent Remote Objects in Java.* Concurrency: Practice and Experience, pages 1225-1242, November 1997. Online at http://wwwipd.ira.uka.de/JavaParty/.

[11] Leslie G. Valiant. A bridging model for parallel computation. Communications of the ACM, 33(8): pages 103-111, 1990.

[12] Sun Microsystems, Inc. Java™ Remote Method Invocation Specification, 1996. ftp://ftp.javasoft.com/docs/jdk1.1/rmi-specs.ps

[13] Java Grande Forum, 1999. http://jhpc.org/grande

Appendix A: Performance results

Suel C code (gcc -O3)

Average time per iteration (ms)

Bodies	Cpu's 1	2	4	8	16	32
1000	55	31	20	19	25	133
3000	270	141	76	46	39	51
10000	1713	871	448	241	139	292
30000	9820	4949	2520	1288	677	383

Speedups

Bodies	Cpu's 1	2	4	8	16	32
1000	1.00	1.77	2.75	2.89	2.20	0.41
3000	1.00	1.91	3.55	5.87	6.92	5.29
10000	1.00	1.97	3.82	7.11	12.32	5.87
30000	1.00	1.98	3.90	7.62	14.51	25.64

Manta Java code

Average time per iteration (ms)

(-trim-arrays -gc-interval 5)

Bodies	Cpu's 1	2	4	8	16	32
1000	145	94	88	88	142	298
3000	733	407	268	203	238	381
10000	4741	2603	1394	839	619	687
30000	27554	15718	7562	4211	2566	1886

Speedups

Bodies	Cpu's 1	2	4	8	16	32
1000	1.00	1.54	1.65	1.65	1.02	0.49
3000	1.00	1.80	2.74	3.61	3.08	1.92
10000	1.00	1.82	3.40	5.65	7.66	6.90
30000	1.00	1.75	3.64	6.54	10.74	14.61

JDK 1.2 Java code

Average time per iteration (ms)

(-trim-arrays -gc-interval 5)

Bodies	Cpu's 1	2	4	8	16	32
1000	176	336	719	Crash	Crash	Crash
3000	863	873	1344	Crash	Crash	Crash
10000	5354	4022	4065	Crash	Crash	Crash
30000	Crash	Crash	Crash	Crash	Crash	Crash

Speedups

Bodies	Cpu's 1	2	4	8	16	32
1000	1.00	0.52	0.24			
3000	1.00	0.99	0.64			
10000	1.00	1.33	1.32			
30000						

Performance ("C"=1)

Bodies	Version C	Manta	JDK
1000	1.00	2.64	3.20
3000	1.00	2.71	3.20
10000	1.00	2.77	3.13
30000	1.00	2.81	

Manta Java code Profile

1000 Bodies

CPU's	1	2	4	8	16	32
load balancing	116	163	942	1037	2539	6572
tree construction	557	310	147	79	45	30
center of mass computation	69	37	21	6	8	2
essential tree exchange	2	630	1025	1560	2500	4691
center of mass update	0	17	20	26	27	24
force computation	4713	2334	1159	583	289	149
position update	110	58	22	13	8	5
explicit garbage collection	101	120	115	133	160	183
total	5668	3669	3451	3437	5576	11656

10000 Bodies

CPU's	1	2	4	8	16	32
load balancing	1163	6234	2398	1439	2510	7679
tree construction	7299	3514	1700	843	435	226
center of mass computation	952	487	223	93	41	20
essential tree exchange	0	3738	5998	7614	9100	12154
center of mass update	0	90	154	191	173	160
force computation	173116	85700	42496	21330	10729	5392
position update	1270	644	305	144	72	40
explicit garbage collection	1122	1113	1094	1099	1088	1122
total	184922	101520	54368	32753	24148	26793

JDK 1.2 Code Profile

1000 Bodies

CPU's	1	2	4
load balancing	200	2317	7060
tree construction	821	456	293
center of mass computation	92	78	25
essential tree exchange	4	6999	18846
center of mass update	2	22	32
force computation	5248	2729	1336
position update	187	104	49
explicit garbage collection	318	434	400
total	6872	13139	28041

10000 Bodies

CPU's	1	2	4
load balancing	2141	4696	6017
tree construction	11926	4970	2599
center of mass computation	1282	656	313
essential tree exchange	1	46431	98544
center of mass update	0	120	203
force computation	189813	97094	48608
position update	2113	1079	532
explicit garbage collection	1530	1849	1755

Explicit garbage collection

GC Interval	0	1	2	3	4	5
Average Iteration Time(ms)	299	314	286	287	292	287

Array trimming

Bodies	No Trimming	Trim Array
3000	291	285
10000	1579	1570

4 CPUs
Average Iteration Time(ms)

Bodies	No Trimming	Trim Array
3000	245	220
10000	996	958

8 CPUs
Average Iteration Time(ms)

Custom serialization

Bodies	No Serialize	Serialize
3000	277	261
10000	1404	1359

4 CPUs
Average Iteration Time(ms)