The usability of Java for developing parallel applications

M. Dewanchand and R. Blankendaal
Faculty of Sciences,
Division of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

vrije Universiteit *amsterdam*

Abstract

The Java programming language offers several possibilities to build parallel applications. To explore these possibilities and to test their applicability, we have built four parallel Java programs. These programs are also written for Manta, a native Java compiler developed at the Vrije Universiteit Amsterdam, based on Sun’s RMI model and JavaParty’s RMI model. In this paper, the performance and applicability of both systems are evaluated and explained by means of four parallel programs: Fast Fourier Transformation, Radix sort, Checkers and Water.

Table of Contents

1 Introduction

Currently, there is much research in the field of parallel programming using Java. The Java object−oriented programming language is clean and type−safe, which makes it attractive for writing distributed and multi−threaded applications. Multi−threaded applications communicate through shared memory, while distributed Java applications use a message−passing protocol based on Remote Procedure Calls, called Remote Method Invocation (RMI).

Manta[10] is a high−performance Java system, which implements the Java RMI model using different communication protocols for different networks. The system uses a native Java compiler rather than an interpreter or Just In Time (JIT) compiler. Manta provides some Java extensions (see section 3.2) to build parallel programs with a highly efficient, easy to use communication mechanism almost identical to the JavaParty model[9]. But still, Manta complies with the Java RMI model[3].

Our goal is to gain experience with parallel programming using Sun's Java system and the Manta system, and to look at the usability of both systems. The usability of each system is determined by looking at its performance and its ease of use to the programmer. In order to do this, we have written four Java applications, using each system's own communication model. The performance is measured by developing four different versions of each program: A sequential version, a multi−threaded version, a Java RMI version and a Manta RMI version. All versions are compared to each other to calculate the efficiency of the parallel versions.

In this paper we first describe Java's possibilities for parallel programming with multi−processors and multi−computers. Next we will look at Manta and its extensions. In section 5 to section 8 the four applications are described in detail. Problems are discussed in section 9. In section 10 our results and conclusions are presented.

2 Multi−threaded programming

Parallel programming in Java on a multi−processor (shared memory) machine is accomplished by using multiple threads. A thread is a portion of a computer program (a sub−process) that may execute concurrently with other threads in the same program, while sharing the same address space. Threads communicate with each other through shared variables and they synchronize using locks, semaphores or monitors [2].

Java offers two object−oriented ways to the programmer to create a new thread, via the *Runnable* interface and the *Thread* class. A thread object is created by declaring the object to implement the *Runnable* interface which requires the programmer to write a *run()* method, or by declaring the object to be a subclass of *Thread* and thus override the *run()* method. To start a thread after its creation, the programmer must call its *start()* method, which creates a new native thread and executes the object's *run()* method.

In Java every object can be used as a lock in a *synchronized* statement, or as a monitor using synchronized methods. When a thread enters a synchronized method, the object is locked, and unlocked when the method returns. This will ensure that at any given time, only one thread is executing a synchronized method.

A thread may block inside a monitor using the *wait()* method. The thread will then be blocked and the object unlocked, to allow other threads to call synchronized methods of the object. To wake up all waiting threads, the *notifyAll()* method can be used. When a thread is notified, it locks the object and continues execution of the synchronized method. Note that the *wait()* and *notifyAll()* methods can only be used inside a synchronized method.

Another way to achieve synchronization is by using a so−called barrier. A thread may use a barrier to block its execution, until all other threads have arrived at the same barrier. Then all threads are woken up and will continue execution. Unlike the Sun Java language, the Manta runtime system offers an interface to its underlying communication layer called Panda, which has a barrier construct built−in that may be used by the programmer. In Sun Java, the programmer has to write his / her own barrier functionality. From the programmer's view, there are no other significant differences between Manta and Sun threads.

3 Remote Method Invocation

3.1 Sun's RMI system

The Java Remote Method Invocation system allows an object running in one Java Virtual Machine (JVM) to invoke methods on an object running in another Java VM. RMI provides remote communication between programs written in the Java programming language [3].

Java RMI applications are often client−server oriented. Typically, the server application creates some remote objects, makes their references accessible, and waits for clients to invoke methods on these objects. A typical client application receives a remote reference to one or more remote objects in the server and then invokes methods on them. An object becomes remote by extending the *java.rmi.server.UnicastRemoteObject* class and by implementing a remote interface, which has the following characteristics:

- A remote interface extends the interface *java.rmi.Remote*
- Each method of the interface declares *java.rmi.RemoteException* in its throws clause, in addition to any application−specific exceptions

To obtain references to remote objects, the RMI *registry* is used. Remote objects are registered via Java's simple naming facility, the *registry*, by *bind*ing the object to a name. A client can retrieve an object's reference by using the *Naming.lookup()* method. When this object is located on the *same* machine, the client only has to specify the object's name. When a reference of an object on *a different* machine is needed, *Naming.lookup()* also requires the hostname of the machine where the object is located. Once the reference has been acquired, the client application can perform (remote) communication by using standard Java method invocation. Java makes no difference whether the object is remote or local, it uses RMI in both cases. Only those methods defined in the object's Interface can be invoked. An object's data can only be retrieved or altered by use of a method defined in the interface. When no more communication is necessary, objects should be unbound by using *Naming.unbind().*

When a reference to a remote object, the hostname can be retrieved by creating a file in which remote objects can write their hostname or can read other hostnames. All four applications use this method to locate other objects.

Parameters passed to methods of a remote class are always sent by value, even arbitrary complex data structures. This is accomplished by Java's automatic serialization protocol which converts (serializes) the data objects into an array of bytes. Serialization produces a deep copy. Then, when the data is received, it is automatically deserialized by Java's runtime system.

3.2 Manta's RMI system

Manta supports both the standard Java RMI programming model and the JavaParty [9] model. For our Manta applications we have used the JavaParty [9] model. With this model, Manta does not require interfaces such as *java.rmi.Remote* to declare an object to be remote. As in the JavaParty[9] system, it uses the keyword '*remote*', a Java language extension. When a class is tagged with the new *remote* keyword, every call to a method in this class will be translated into a Remote Procedure Call.

With Java RMI, remote objects are always created on the local CPU, while Manta and JavaParty allow the programmer to create remote objects on any CPU using the *RuntimeSystem.setTarget(CpuNr)* method. It is thus possible for code being executed on CPU1 to create a remote object on CPU4, and to immediately receive a reference to that object. Therefore, the Manta system does not need a registry to locate remote objects.

Both Manta and Java RMI do not allow the programmer direct access to a remote object's data members. With Java this follows from the language definition, since variables cannot be declared in an Interface. Although Manta does not require an Interface, a compile−time error will be generated as well, when an attempt is made to access a remote object's fields directly.

There is an important difference in the way Manta and JavaParty handle method calls inside a remote class. Consider the following example:

```
remote class A {

        methodA(myObject obj) { ... }

        methodB(myObject obj) {
                methodA(obj);
        }
}
```

In this example, methodB calls methodA and passes the parameter obj, which is an object. In Manta, no RMI call takes place and the object is transferred ‘by reference’. JavaParty on the other hand does perform a RMI call and thus uses ‘call by value’. So, in this example JavaParty is slower than Manta because of the overhead induced by the extra RMI call.

4 The Distributed ASCI Supercomputer (DAS)

The system that we used to run and test the parallel Java applications is called the DAS. It is a wide−area distributed cluster designed by the Advanced School for Computing and Imaging (ASCI). DAS consists of four clusters, located at four Dutch universities. The cluster at the VU contains 128 nodes, the other three clusters have 24 nodes (200 nodes in total). The system was built by Parsytec and runs the Red Hat Linux version 5.2 operating system. Each node contains:

- A 200 Mhz Pentium Pro (Siemens−Nixdorf D983 motherboard)
- 128 MB EDO−RAM in DIMM modules (64 MB for the clusters in Delft and UvA)
- A 2.5 GByte local disk
- A Myrinet interface card
- A Fast Ethernet interface card

The nodes within a local cluster are connected by a Myrinet SAN network, which is used as high−speed interconnect, mapped into user−space. The network interfaces are interconnected in a torus topology using 32 switches and 2 x 1.28 Gbit/s cables. In addition, Fast Ethernet is used as OS network (file transport). The four local clusters are connected by wide−area ATM networks, so the entire system can be used as a 200−node wide−area distributed cluster. In this work, however, we only use the VU cluster.

While Java uses TCP/IP in combination with Fast Ethernet to implement RMI calls, Manta can use both UDP/IP (Fast Ethernet) and LFC (Myrinet). We have used Myrinet when measuring Manta application performance, unless otherwise noted. The Java applications were compiled with JDK1.2 and Manta. When compiled with JDK1.2 the applications were run on Fast Ethernet, with Manta they were run on Myrinet.

Measurements for the shared memory applications were performed on a dual Pentium Pro 200 Mhz, the das0fs. The das0fs SMP is connected to the VU−DAS cluster and is actually the file server of the DAS. Unfortunately we were not able to measure performance of the multi−threaded versions compiled with Manta on the das0fs. This because Manta makes use of Panda which does not supply for kernel−space threads, only user−space threads, causing a multi−threaded program to use only one CPU. So, it would have been of little use to measure the performance of our multi−threaded programs using Manta, because no more than one CPU would be used. Java does support kernel−space threads and allows the multi−threaded program to use multiple CPUs.

5 Fast Fourier Transformation

5.1 Introduction

Linear transforms, especially Fourier and Laplace transforms, are widely used in solving problems in science and engineering. The Fourier transform is used in linear systems analysis, antenna studies, optics, random process modeling, probability theory, quantum physics, and boundary−value problems and has been successfully applied to restoration of astronomical data [8]. This program calculates a one−dimensional Fast Fourier Transform, using the transpose algorithm [6].

5.2 The sequential algorithm

The sequential FFT program is a direct conversion of an Orca application written by Aske Plaat, which on its turn is based on existing SPLASH−2 code [5, 6, 7]. It consists of two classes, i.e. 'FFT' which forms the main program and a helper class to store the matrix values in.

The program starts by setting some global variables and by placing random values into the main matrix. After that, the Fast Fourier Transformations take place. The matrix is transposed three times and the values are manipulated using a relatively complicated algorithm consisting of three inner loops.

The values stored in the matrix are manipulated directly from within the main program by defining the matrix−data ''friendly'. Therefore, it is not necessary to call member functions in order to retrieve or alter the values, because they are directly available. The speed gain that we obtained by using this friendly data emphasizes our goal of high performance. The benefits of 'information hiding', which states that an object should never allow the user to gain direct access to its member variables in order to create a certain form of abstraction, are considered less important.

5.3 Parallelism

Multi−threaded

The multi−threaded version of FFT is based on the same Orca code that we used to write the sequential application. The distribution of the problem task to the various threads is accomplished by splitting the original matrix into equally sized parts. Although each thread has access to the entire matrix (shared memory), it performs the transformations only on the parts that it is assigned to. Whenever a thread wants to perform FFT on a certain part of the matrix (or wants to transpose it), the thread waits for all other threads by using a barrier. Then a copy is made of the matrix values, and the data manipulations take place on that copy. Wrong matrix values would be calculated if the other threads were to manipulate the values at the same time. So, by using a barrier, all threads run synchronized with each other and we can be sure that it is safe to access or modify a piece of the data. When the Fast Fourier Transformations on the copy are finished, the new data is copied back into the original matrix.

The main reason for copying the data is to avoid the high synchronization overhead. If the matrix elements were modified directly, a synchronization would be required after every modification. By using blocks of data, this number of synchronizations is greatly reduced.

Manta's Remote Method Invocation

The differences between the multi−threaded FFT and the Manta RMI version are relatively small. The main program distributes the Slaves (which extend RemoteThread) to all available CPUs. This is accomplished by marking the Slave class 'remote' and setting the appropriate CPU target. Then, when an instance of a Slave is created (using new) it actually resides on the other CPU.

As in the multi−threaded version, the matrix is split into multiple sub−matrices. All Slaves have access to this array of sub−matrices, but the remote Matrix **objects** themselves are not necessarily located on the same CPU. While the slaves in the multi−threaded FFT could directly access and/or modify the values of a matrix, this is not possible in Manta's RMI system. Instead, the slaves must use the remote *get()* and *set()* methods of the Matrix class. For reasons mentioned earlier, matrix manipulations require barriers for the algorithm to remain correct.

When performing measurements on the Manta RMI version of FFT, we found that the application performed much communication. Because the FFT calculation involves three matrix transpositions (all−to−all communication in essence) and two data copies (all−to−all communication as well), an optimized version was written to reduce this communication overhead. It is accomplished by distributing the sub−matrices in such a way that the matrix manipulations can be performed locally (on the same processor) as much as possible. The Matrix class is no longer remote, since all matrices are created inside the Slave objects now and there is no centralized object needed anymore to handle access to the matrices. If a slave still wants to access a matrix on another CPU, it performs a Remote Method Invocation to obtain the matrix' values to the owner directly. The same holds for setting the values of a matrix. To show the impact of this optimization in relation to the amount of communication performed, consider this example with M=18 on 32 CPUs:

Data transfer	RMI calls	Bytes transferred
Normal	3,399	133,003,834
Optimized	1,178	1,690,368

Notice that these values are maxima. Because of a load balancing problem with the unoptimized Manta application, the 'Bytes transferred' value is extremely high for one particular CPU, while the other processors send only a fraction of this amount. The distribution of data transfer is much better with the optimized Manta RMI version.

Sun's Remote Method Invocation

This implementation of FFT is quite a bit larger than the Manta version, mostly because of the use of the RMI registry, the try { } catch(...) Exception handling blocks and the unavoidable file operations to establish a link between all slaves and the Master process.

The most important feature of this Java version is the use of a centralized Master class. A Master object is created on one CPU, which stores all Matrix values. All slaves have access to this remote Master object via its Interface. The Interface offers methods to retrieve or alter sub−matrices, and to synchronize with the other CPUs by use of a barrier. It goes without saying that this centralized solution doesn't improve the scalability of the program. When increasing the number of processors, the Master node becomes more and more a bottleneck because of the communication.

Like Manta, an optimized version was built to reduce the communication overhead, but unfortunately we were not able to run the application since the Java runtime system crashes when using more than two CPUs. Therefore, no performance measurements are available.

5.4 Results

All performance measurements for FFT are available in Appendix A.

The first graph shows the performance of the multi−threaded application vs. the sequential one, for three different matrix sizes.

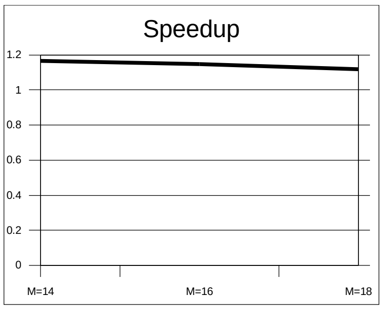


Fig. 5.1

As can be seen, the speedup scores are rather poor (1.17 with M=14 at best, where the amount of transformed complex numbers equals 2^M). They can be explained by looking at the next picture were the application execution time is split into several parts. For example, it shows that the multi−threaded version is using more time waiting in barriers and doing communication as the matrix size increases, while the sequential one does not need communication or barriers at all. That is the most important reason for the poor performance of the multi−threaded application. Another reason is that the sequential program can manipulate the values in the matrix directly (even without method calls) while the multi−threaded application needs to copy the data to a temporary buffer first. Then, when the updates are done, the buffer is transferred back to the original matrix again. These extra copy operations are visible in the graph when examining the 'pure FFT' bars. Although each CPU with the multi−threaded program needs to perform exactly half the FFT calculations, the execution time is not halved because of the back−and−forth copying overhead.

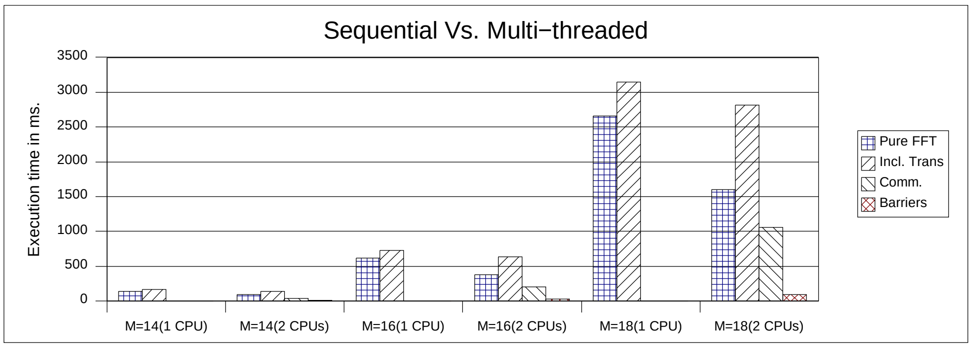


Fig. 5.2

The bar named 'Incl. Trans' shows the time needed for pure FFT calculation plus transpositioning and communication, basically the application's total execution time excluding initialization.

When comparing the multi−threaded application with two CPUs to the same application using a single CPU, the speedup is much better (close to 1.75 in all three cases, not shown).

The following graphs show the performance of the non−optimized Fast Fourier Transformation programs on both the Java RMI system and Manta's RMI system using different problem sizes. These two applications can be compared best since the algorithm used is similar. Notice the different scales used in the graphs. Manta clearly outperforms Java, not only in absolute application execution times, but also in the multi−processor efficiency factors. As can be seen, the Java version does not get any speedup since the speedup factor is below 1 in all cases, while Manta achieves a maximal speedup of 13.98 using 32 processors and a maximum efficiency of 0.61 with 2 CPUs. Unfortunately, the efficiency rapidly declines when using more than 8 CPUs because of communication overhead.

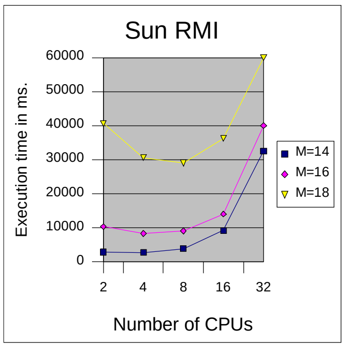


Fig. 5.3a

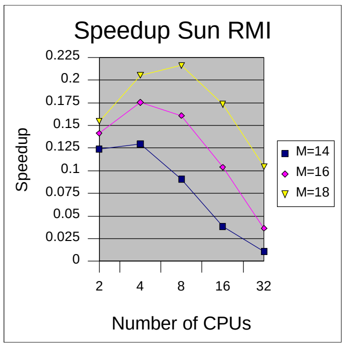


Fig. 5.3b

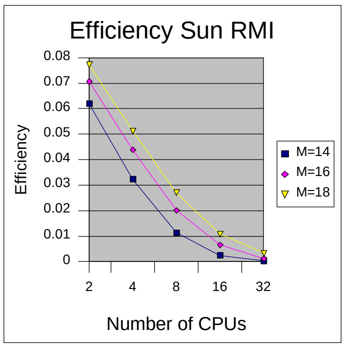


Fig. 5.3c

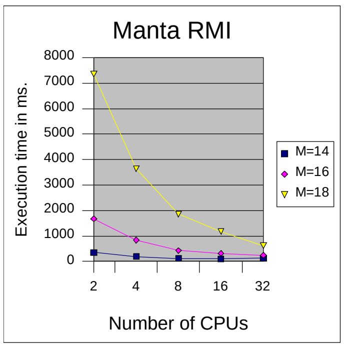


Fig. 5.4a

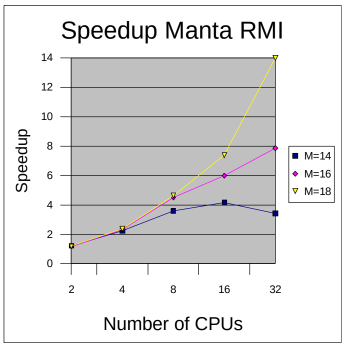


Fig. 5.4b

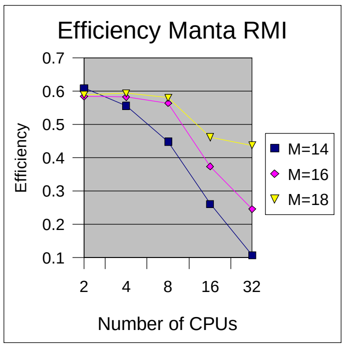


Fig. 5.4c

The optimized Manta RMI version performs better. The speedup factor increases in nearly all cases when the number of CPUs increases, and the efficiency figures are mostly above 0.55. For comparison: on a problem−size of M=18 (262.144 values) running on 16 CPUs, Java achieves an efficiency factor of just 0.01, where optimized Manta reaches 0.60.

It may seem somewhat surprising that the speedup of the optimized Manta application exceeds the speedup of the multi−threaded program. This is probably due to the fact that the shared−memory computer does not have a reservation system like the multi−processor machine. This means that, while the multi−processor machine guarantees 100% CPU time for the user, the multi−threaded application will be scheduled among other processes of other users, which may strongly influence the performance measurements. In the worst case, if only one of the CPUs is available for a moment, only one of the threads (thread A) can run. But since the threads need to be synchronized every once in a while, A will have to wait for the other one (B) to be scheduled and reach the barrier as well. The same may happen to thread B.

Measurements on the Orca version are given in Fig. 5.6 for comparison. Not only does the Orca implementation clearly outperform Manta in execution times, it also runs more efficient on multiple processors. Notice that some Orca efficiency factors shown in the graphs slightly exceed 1.0, which means super−linear speedup is achieved. This is probably due to the fact that the speedup is measured by comparing the multi−processor Orca performance against the same application running on a single CPU. Since the application was specifically written to be used on a processor pool, it runs rather inefficiently on one processor. Other reasons for the high parallel speedup may be cache−related advantages and some amount of inaccuracy in the measurements.

The results show that Orca achieves better efficiency factors than Manta, especially with a large matrix. Both implementations show a decline in efficiency when using more than 16 CPUs.

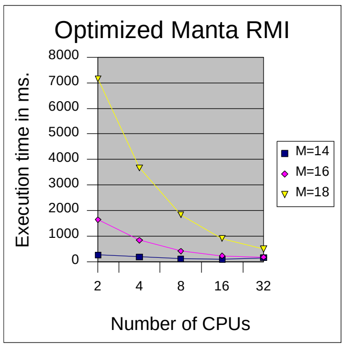


Fig. 5.5a

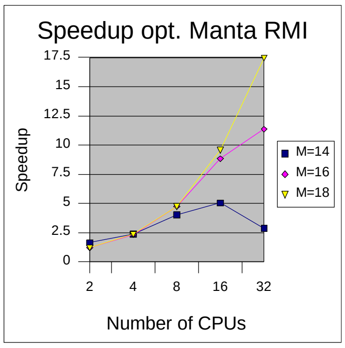


Fig. 5.5b

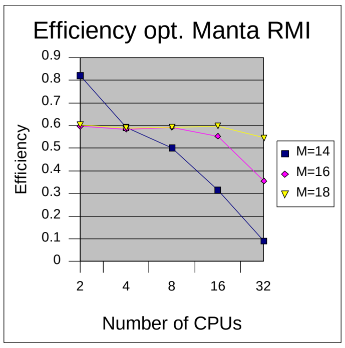


Fig. 5.5c

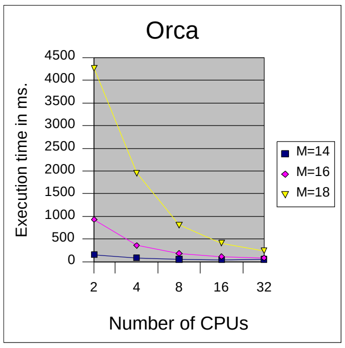


Fig. 5.6a

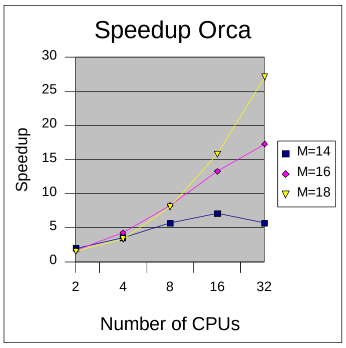


Fig. 5.6b

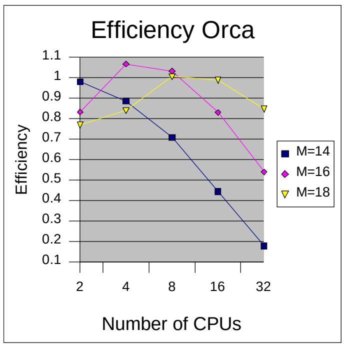


Fig. 5.6c

The next graph shows the execution times (using a single CPU) split into various sub−tasks.

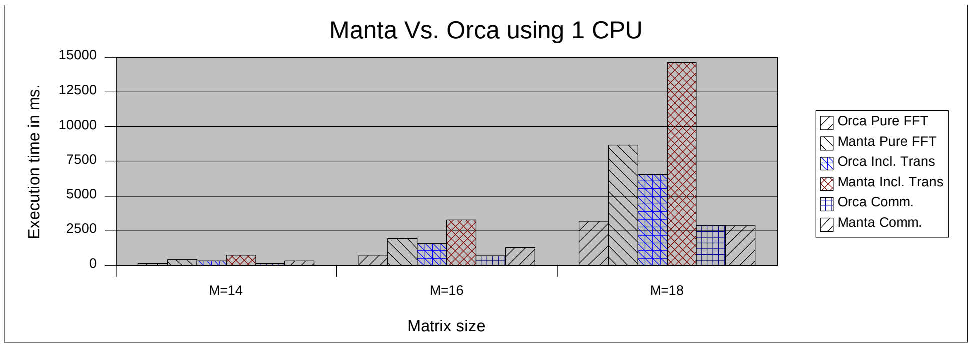


Fig. 5.7

This graph clearly shows that the pure FFT calculations take longer on Manta than with Orca. Orca beats Manta by a factor of about 2.65, which is the main reason for the Manta application to take nearly 2.25 times more total execution time.

When comparing the Orca program (using 1 CPU) with the original sequential Manta application (the one without communication), the total execution time factor is improved to about 1.30. Notice that Manta communication is not shown in the graph because it is not applicable in this case, while Orca does have communication.

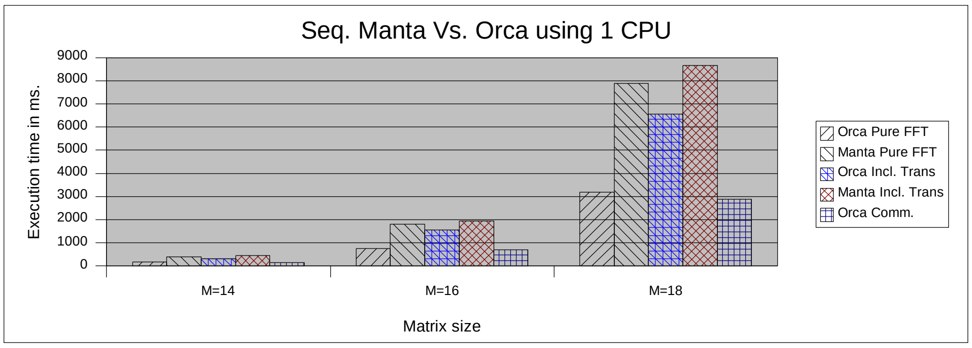


Fig. 5.8

6 Radix sort

6.1 Introduction

Sorting is one of the most frequently studied problems in computer science. It is important in a wide variety of practical applications and it is a simple combinatorial problem with interesting and diverse solutions. Sorting is also an important benchmark for parallel computers. Radix sort is a simple and efficient sorting method. It does not use comparisons to determine the relative ordering of the elements. It relies on the binary representation of the elements to be sorted. The radix sort algorithm is a stable algorithm, it preserves the order of input elements with the same value.

Radix sorting is used in many applications for sorting integers, strings, floating−point numbers etc. The radix sort described in this thesis is one for sorting integers. The problem is in the order of O(n log n).

6.2 Sequential algorithm

If b is the number of bits in the binary representation of an element, the radix sort algorithm examines the elements to be sorted R bits at a time, where $R< b$, starting with the least significant block of R bits in each element. So the radix sort algorithm requires b / R iterations. During each iteration the elements are sorted according to the R−bit block currently being considered.

The algorithm is implemented in two different ways: one is a traditional algorithm which uses the bucket sort algorithm[16] and the other is the SPLASH version which uses the counting sort algorithm[11].

Radix sort using Bucket Sort (RBS)

The idea is to sort numbers (keys) by distributing them into buckets. For each possible key there is one bucket. So if R bits are examined at a time, the possible amount of keys is 2^R and 2^R buckets are needed. On each iteration the algorithm puts the elements in their respective buckets (according to their key value) and collects them during a traversal of the buckets.

The most important classes are the 'Sequential' class which forms the main program and the 'Radixsort' class where the actual sorting is performed. The 'Sequential' class starts by generating a random array of integers that must be sorted. These integers are passed to the 'Radixsort' class which sorts them according to the basic algorithm. For the implementation of buckets, lists are used, because it is not known beforehand how many keys a bucket will contain. Most of the data variables and methods are friendly or public so that variables can be accessed directly. No method calls are needed to access data in another class. Although this is against OO principles, it is faster.

SPLASH (Radix sort using Counting Sort (RCS))

Radix sort is one of the applications of the SPLASH−2 application suite. It is based on the counting sort algorithm and it is written in the C−language. In each iteration a histogram is generated over the keys. This histogram is used to permute / sort the keys into an array for the next iteration. The algorithm first counts to determine the rank of each key −its position in the output order− and then permutes these keys to their respective locations.

The program consists of the same classes as radix sort using bucket sort. In the class 'Radixsort' the histogram is generated by first building a frequency table of the numbers to be sorted. After the histogram is built, the numbers to be sorted are moved into an array according to their position in the histogram. This is repeated for the next iteration.

6.3 Parallelism

The main idea to parallelize Radix sort is as follows: divide the numbers to be sorted over the CPUs. Each CPU sorts its own numbers and then merges the numbers into one sorted array.

Radix sort using bucketsort (RBS) follows the main idea. The integers to be sorted are distributed over the CPUs. Each CPU uses the sequential radix sort algorithm to sort its own numbers. After each CPU has sorted its numbers the numbers are merged into one array. The first half of the CPU pool merges its numbers with the other half of the CPU pool, then a quarter of the CPU pool merges its numbers with the other quarter and so on. For example, if there are eight CPUs in the pool of CPUs CPU1 to CPU4 start merging their numbers with CPU5 to CPU8. Next CPU1 and CPU2 merge their numbers with CPU3 and CPU4. The first CPU finally contains the sorted array of numbers.

The SPLASH version of radix sort (RCS) works somewhat differently. The array of keys to be sorted is partitioned into regions. Every CPU gets a region to sort. The merging is done during the process of sorting. On each iteration three phases can be distinguished:

- LocalSort: Each CPU sorts its own keys using the sequential splash−radix sort.
- Histogram merge: After each CPU has built its own (local) histogram, the histograms are merged into a global histogram which determines for each key owned by a processor where it should be stored. The merging involves computing a vector prefix sum and a vector sum.
- Permutation: After the global histogram is determined all processors move their keys into their destination regions.

During the histogram merge phase a combining tree algorithm [11] is used to combine partial results and to send the vector sum to every other CPU. The amount of communication during the permutation phase is a subset of all−to−all communication.

Multi−threaded

With the multi−threaded version of RBS the distribution of the problem task is accomplished by splitting the array to be sorted into parts. Each thread receives a copy of a part of the array and sorts its part. After each thread has finished sorting its part, half of the threads start merging their array with the arrays of the other half of threads, and so on. Each thread has direct access to the sorted array of other threads, but should only start merging when the other threads have finished sorting. This is accomplished by using a barrier.

The multi−threaded version of RCS is also based upon the SPLASH−2 code. The distribution of the problem task is accomplished by dividing the array of integers to be sorted into regions. Each thread gets access to a region and sorts its own region. After each thread has built its local histogram they start merging their histograms into a global histogram. The global histogram is shared among all threads, each thread has access to it, and it is used to determine the position of each key. For the global histogram a tree−like datastructure is used because this datastructure improves the efficiency of the merging of local histograms and reading from the global histogram. Since threads merge their arrays into the global histogram, synchronization is necessary. This is done with the *wait()−notify()* methods. When threads move their keys into the sorted array no synchronization is needed, because every key has a unique position. But a key can be placed in a region of another thread. So the next iteration of sorting must begin after all threads have moved their keys. This is accomplished with the use of a barrier as well.

Manta’s Remote Method Invocation

With RBS two classes can be distinguished, a master class, which forms the main program and a worker class, which does the actual sorting. Both master and worker classes are remote. The master is remote so that workers can get the address of the other workers when they need to merge their data. Each worker receives a copy of a part of the array to be sorted from the master and starts sorting it. During sorting no communication is necessary. Only when the worker objects start merging, some of the workers need to send their array to the other half, so communication is needed.

RCS contains the same classes as radix sort using bucketsort. Each worker receives the keys in their region and builds a local histogram. In this phase no communication is needed. During the histogram merge, all−to−all communication is necessary since the global histogram is shared among all worker objects. Each worker object needs to know what the other objects have merged. Because the global histogram is shared and every worker object reads and writes to the histogram, synchronization is necessary so that each object works with the correct values. This is accomplished through *wait()−notify()* methods and barriers. During the movement of the keys communication is only needed when keys have to be moved into another region. No synchronization is necessary because every key has a unique position. But between every iteration of sorting synchronization is necessary, because every worker object has to wait until all keys are placed into their correct position before sorting its region.

It should be noted that Manta does not support broadcasting. So during the histogram merge each worker does RMI calls to the other ones.

Java’s Remote Method Invocation

The idea of RBS and RCS with Java RMI is the same as with Manta RMI only the implementation is somewhat different. Communication between objects is accomplished through interfaces. A difference with Manta is the use of a centralized object which decides if a Master class or a Slave class should be created depending on the host−number. Java does not support broadcasting either.

6.4 Results

Below several graphs are given that show the performance of RBS and RCS with Sun Java and the Manta system. Performance is measured using different input sizes and a variable number of CPUs. Unfortunately, there are no results of RCS with Sun Java RMI because Java continuously crashes. With RBS, Java could not handle lists containing more than 500000 numbers.
Java also has trouble with RBS on more than 8 CPUs. The multi−threaded versions were only tested on a machine containing 2 CPUs.
All performance measurements can be found in Appendix B.

RBS

With RBS figure 6.1 shows that Manta is somewhat faster than Java. With an input size of 500,000 numbers the difference is less than with an input size of 100,000 numbers. This is probably due to some overhead of the JIT, which is relatively bigger with small problem sizes.

With the multi−threaded version of RBS Java gained a speedup of approximately 1.42 with 100000 numbers and 1.39 with 500000 numbers (figure 6.2). The efficiency fluctuates between 0.69 and 0.71.

With the RMI versions the speedup gained with Manta is also higher than with Java. With RBS Manta reached a speedup of 9.18 with one million numbers on 32 CPUs (figure 6.3b), while Java reached a speedup of 2.26 with 500,000 numbers on 8 CPUs (figure 6.4b). Figure 6.3c and 6.4c show that the efficiency of Java on RBS decreases at a faster rate than that of Manta when the number of CPUs increases.

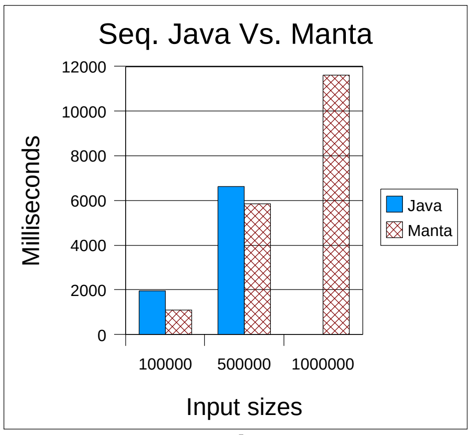


Fig. 6.1 Sequential RBS

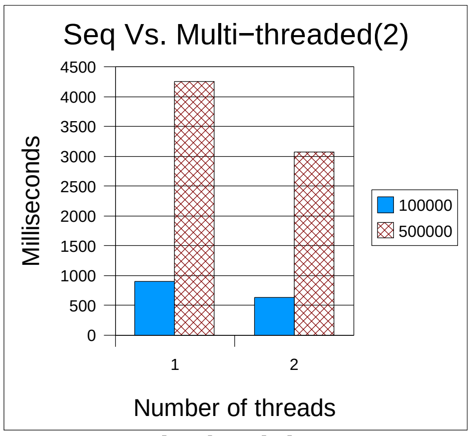


Fig. 6.2 Multi−threaded RBS

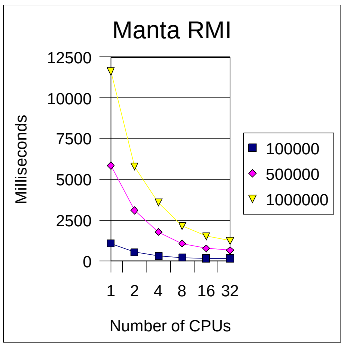


Fig. 6.3a RBS

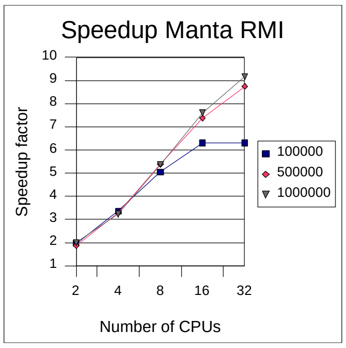


Fig. 6.3b RBS

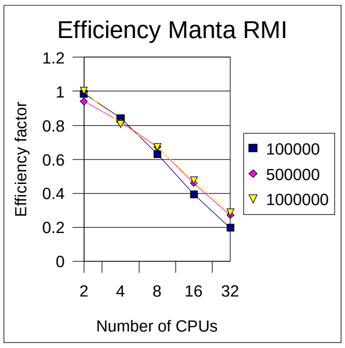


Fig. 6.3c RBS

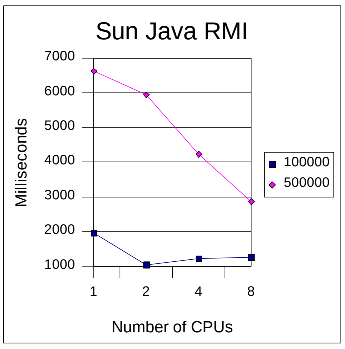


Fig. 6.4a RBS

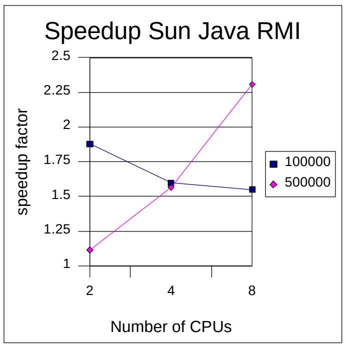


Fig. 6.4b RBS

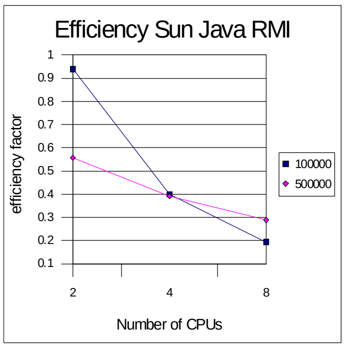


Fig. 6.4c RBS

RCS

The Splash version (RCS) is much faster than RBS, up to a factor 8.2 with Java and a factor 7.69 with Manta (figure 6.1 and figure 6.5). This is due to the many simple operations in RCS. With RCS Manta is somewhat faster than Java (figure 6.5) as well. The multi−threaded Java version of RCS gained an average speedup of approximately 1.54 (figure 6.6). The efficiency fluctuates between 0.625 and 0.845.

With the RMI versions Manta gained a speedup of 6.95 with 3,000,000 numbers on 16 CPUs (figure 6.7b). After 16 CPUs the speedup of Manta decreases. This is caused by communication overhead. The speedup is less than with RBS because the sequential version of RCS is already very fast, which makes it harder to gain speedup. The efficiency of the Manta declines as the number of CPUs increases (figure 6.7c).

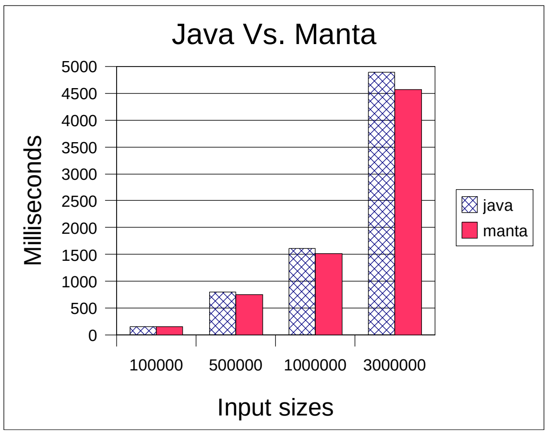


Fig. 6.5 Sequential RCS

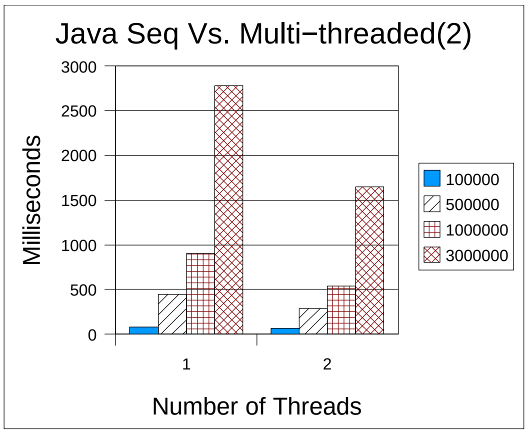


Fig. 6.6 Multi−threaded RCS

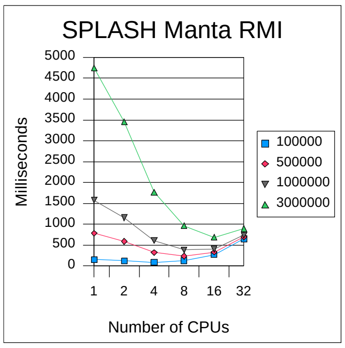


Fig. 6.7a RCS

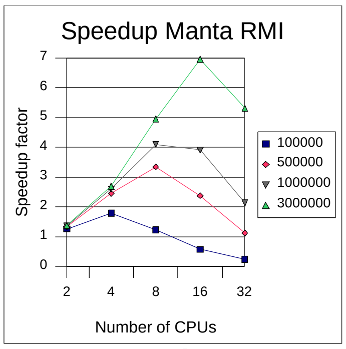


Fig. 6.7b RCS

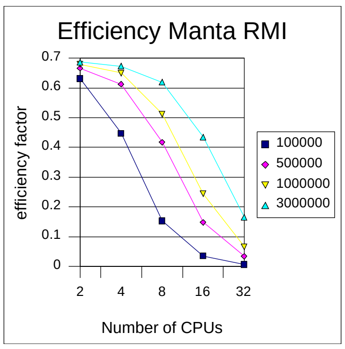


Fig. 6.7c RCS

Manta RCS Vs. C RCS

Still the Manta version of RCS is approximately a factor 2 slower than the C−version of RCS (written at the VU) with 3,000,000 numbers (fig 6.8a). This is caused by the Java language itself and the fact that broadcasting is not supported by Manta and Java. With the C−version the global histogram is broadcasted. Also the C−version makes use of pointer arithmetic which is not possible in Java. The C−version gained a speedup of approximately 10 with 3,000,000 numbers on 32 CPUs (figure 6.8b).

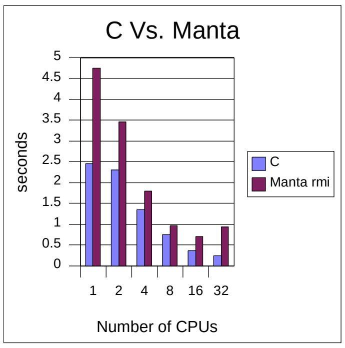


Fig. 6.8a

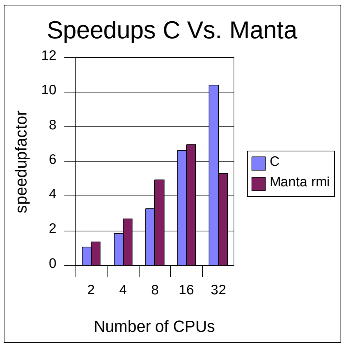


Fig. 6.8b

In both RBS and RCS no linear speedup is reached. With RBS on Manta there is a loss of 71.4% on 32 CPUs. From figure 6.9a it can be seen that this is due to the merging time and the communication time. The merging time is the time needed to merge all the local sorted arrays and the communication time is the time needed to move the arrays from CPU to CPU. The sorting time is the time needed for each worker object to sort its numbers. With 32 CPUs the merging and communication time take 71.4% of the total time. This is exactly the loss that occurred.

RCS had a loss of 57% on 16 CPUs. In figure 6.9b the permute time is the time needed to calculate the position of the keys and move them to their correct position. The merge time is the time needed to merge the local histogram into the global histogram. Both times include the communication time. The sort time and the histogram time show the time needed for the local sort and for building the local histogram. From figure 6.9b it can be seen that the merging time increases linearly (or more), when the number of CPUs increases. With 16 CPUs the merging time takes about 33−35% of the time. The permutation time also decreases slowly from 54−55% on 2 CPUs to approximately 35% on 16 CPUs and 28% on 32 CPUs. With 16 CPUs the permute time is not even halved. Thus on 16 CPUs the permutation time and merging time almost completely cause the loss of 57%.

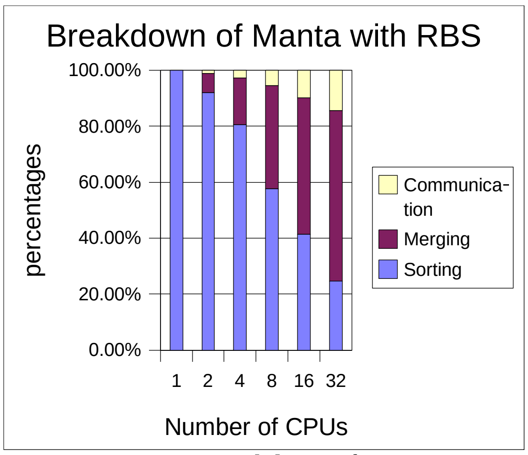


Fig. 6.9a Breakdown of RBS

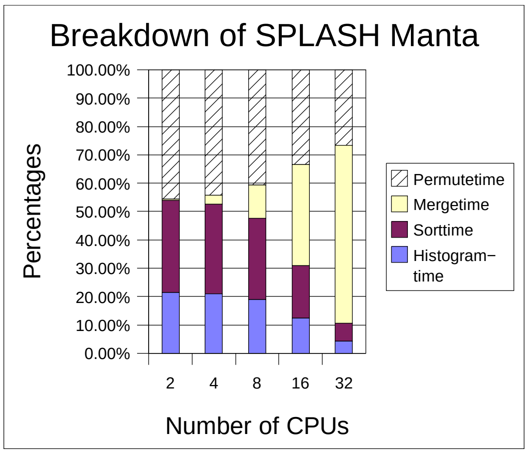


Fig. 6.9b Breakdown of RCS

7 Water

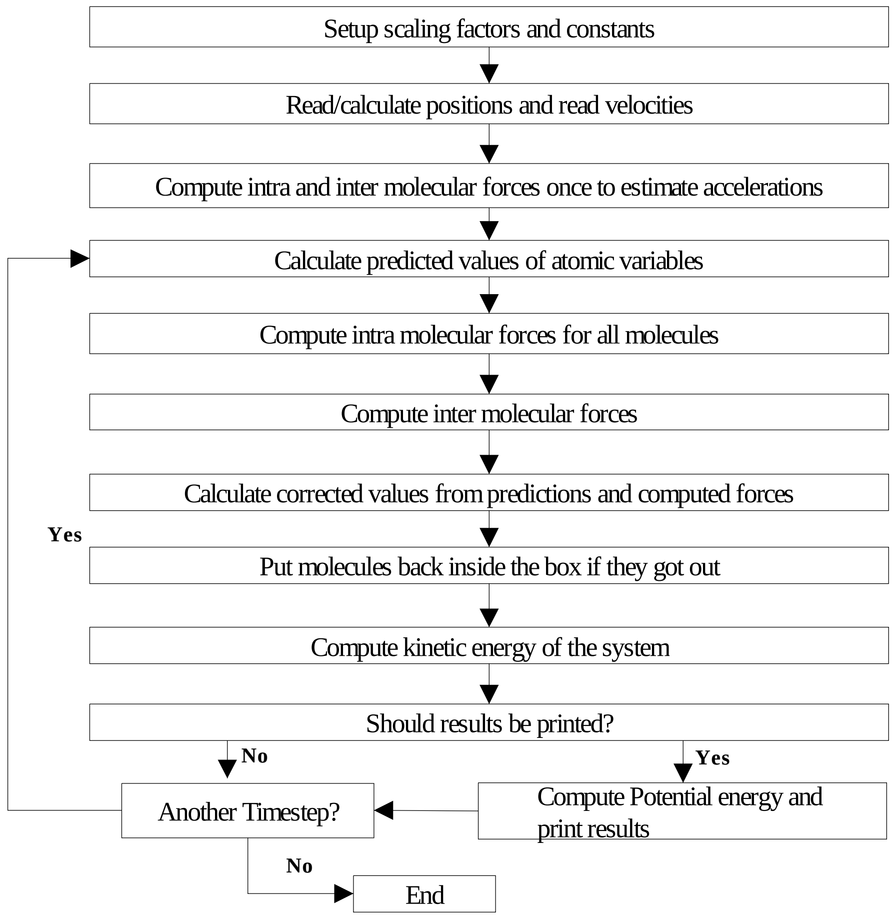
7.1 Introduction

Water [13, 15] is an N−body simulation that computes how H_2O particles move in an imaginary box. It can be used to predict a variety of static and dynamic properties of liquid water. This program requires lots of floating point computations. Also interactions between molecules and between atoms inside a molecule exists. Interaction between molecules is accomplished by means of gravitational forces. Water is one of the applications of the SPLASH application suite.

The computations are performed over a user−specified number of timesteps. Every timestep involves setting up and solving the Newtonian equations of motion for the water molecules in the cubical box with periodical boundary conditions, using Gear's sixth−order predictor−corrector method. The total potential energy is computed as the sum of intra− and inter molecular potentials. The box' size is computed to be large enough to hold all molecules. This implementation of water is based on the Orca implementation [12].

7.2 The sequential algorithm

The structure of the program can be divided in a number of steps:


Setup scaling factors and constants
Read/calculate positions and read velocities
Compute intra and inter molecular forces once to estimate accelerations
Calculate predicted values of atomic variables
Compute intra molecular forces for all molecules
Compute inter molecular forces
Calculate corrected values from predictions and computed forces
Yes
Put molecules back inside the box if they got out
Compute kinetic energy of the system
Should results be printed?
No
Yes
Another Timestep?
Compute Potential energy and print results
No
End

During initialization, the program reads input data like the number of timesteps, the duration of a timestep, the number of molecules that should be simulated, their velocities and how often the results should be printed. Initial positions of molecules can be read or calculated. The size of the box is calculated together with some other input dependent and independent constants. Finally some implementation dependent initializations are done.

After the initialization phase, the program repeatedly does predictions, calculates intra and inter molecular forces, calculates the kinetic energy, places molecules that moved out of the box back inside and corrects the predictions. For the calculation of inter molecular forces only pairs of molecules that have a distance of less than half a box are considered.

The main data structure is the molecule object. Every molecule is represented as an object containing a three−dimensional array ‘f’, and a one−dimensional array ‘‘vm’, as can be seen below.

```
class Molecule {
        double[][][] f;      //usage: f[max_order][nr_directions][nr_atoms]
        double[] vm;
}
```

The‘‘f’ field of a molecule contains various kinds of values, the first seven entries of the first dimension represent the position (displacement), velocities, acceleration, and subsequent derivatives. The last entry contains the force upon a certain atom. The second dimension contains the various directions (XDIR, YDIR, ZDIR), the third dimension contains the atoms of a molecule (H_1, H_2, O). A possible access looks like: molecule.f[FORCES][XDIR][H1]. The ‘‘vm’’ field contains intermediate results.

Calculation of the inter and intra−molecular forces and the kinetic and potential energy is put in one class ‘ForcesEnergy’. The reason for this is that the data needed to perform these calculations is almost the same. The data is then given once, during creation of the object. Predictions, corrections and boundary checking are implemented as methods in a class called ‘‘PredCorr’. The class ‘Water’ starts the applications and reads the input data.

There are also some constants that are used by almost every class. Instead of defining them in each class that needs them, these constants are put in an interface ‘ConstInterface’. Every class that needs these constants implements the interface.

7.3 Parallelism

The main structure of the program is the same as the sequential algorithm except that the work is to be executed by more than one CPU. Distribution of the work is achieved by partitioning the array of molecules over the available processors. So each processor ‘owns’ a number of molecules and performs all computations necessary for those molecules. The computations to be performed are the same as for the sequential algorithm.

As soon as the input data is read or calculated, and part of the initialization is done, the molecules are assigned to the processors. The distribution of molecules is done statically. Each processor then calculates the intra– and inter−molecular forces in order to initialize some variables of a molecule. Next each processor performs the necessary calculations. Performing these calculations is done ‘‘timestep’’ times. A scheme of what each processor does is given below.

Initialization:
- Calculate intra molecular forces to initialize each molecule’s forces using its own position
- Calculate inter molecular forces to estimate the accelerations of the molecules. As the function name suggests, other molecules are also involved and they can reside on other processors. Only interactions of molecule pairs that have a distance of less than half a box’ size are considered.

Every timestep:
- Predictions. The position and derivatives of each molecule are updated.
- Calculate intra molecular forces. Each CPU calculates inter molecular forces for their own molecules and updates them. So no communication is necessary.
- Calculate inter molecular forces and update molecules. The inter molecular forces are only calculated for molecule pairs that have a distance of less than half a box. This method requires communication between molecule pairs.
- Correct some predicted values. Each CPU corrects its predicted values. No communication is necessary.

– Check the boundaries, if molecules got out of the box put them back in. Again no communication is needed.
– Calculate kinetic energy. Every CPU calculates its kinetic energy based on the velocities of its own molecules.
– If results should be printed, calculate potential energy en print some results. The calculation of the potential energy also requires positions of molecule pairs that have a distance of less than half a box. So communication is needed.

In order to calculate inter molecular forces (for the initialization and for each timestep) between molecule pairs the distance between these molecules is needed. To calculate these distances the positions of the molecule pairs are needed. Since molecules can be stored on other CPUs, communication is needed to acquire those positions. During the calculation both molecules must be updated. Again communication is needed if the molecules reside on a different CPU.

The calculation of the potential energy also requires the position of all molecules within half a box' distance. No updates to molecules are done, only the positions are read. Only some shared variables must be updated in order to print some results.

To print results each processor maintains some local variables. These variables are summed into shared variables. Summation of these local variables into shared variables requires extra communication and synchronization to avoid lost updates. All other calculations mentioned are just calculations on molecules and variables, no communication is necessary.

Multi−threaded

The multi−threaded version of water follows the parallel algorithm exactly. After the input data is read and some global variables are set, the master object distributes the work to its worker−threads.
Distribution is done by giving each worker−thread a begin index and an end index of the molecules to work on. The calculations of the forces, energy, predictions and corrections are the same as the sequential algorithm, thus the same classes and datastructures as those in the sequential algorithm are used.

Communication during the calculation of the inter molecular forces and the potential energy is accomplished by giving each processor a pointer to the array of molecules. Updates are written directly to the molecules, even if one of the molecules is owned by another thread. No synchronization is needed because threads that are calculating the 'interf' read the positions (or the acceleration during initialization) of molecules and update the forces of those molecules. For the calculation of the potential energy only the positions are read, nothing is updated, so no synchronization is needed.

There are some points of synchronization i.e. after the calculation of the intra molecular forces, after calculation of the kinetic and potential energy and after each timestep. After these calculations some shared variables should be updated. Synchronization during these updates is accomplished through the use of synchronized blocks, where a global variable must be locked before a thread can perform its update. Since threads can only continue after all updates have taken place, they have to wait for each other. This is done with the use of barriers. The shared variables are mostly used for printing some results.

Manta's Remote Method Invocation

With Manta RMI the master object distributes the work by giving each worker object a part of the array of molecules together with some variables. Both the master object and the worker object are remote. The master is remote because every thread must get references to the worker object(s). After the distribution each worker object begins with the initialization of the molecules and then starts performing each timestep. The classes and datastructures used are the same as those in the multi−threaded version and the sequential algorithm.

The Manta RMI implementation is based on the parallel algorithm described earlier. Each CPU receives approximately the same number of molecules. No provision is made to dynamically balance the load. This is very hard to achieve and results have shown that the load imbalance is very small.

To optimize the communication between molecule pairs *message combining* is used. The idea is that to obtain positions of other molecules each CPU asks approximately half of the other CPUs for the positions of their molecules before the 'interf' is performed. A naive algorithm is that during the calculation of the 'interf' each CPU asks explicitly for the position of the molecule on another CPU. Thus an RMI call is performed for every other molecule that resides on another CPU.

With *message combining* the communication time is greatly reduced. This is because the number of RMI calls performed per CPU to obtain the positions of molecules is approximately half of the total number of CPUs. With the naive algorithm the number of RMI calls per CPU is approximately half of the number of molecules. The number of molecules is usually much greater than the number of CPUs. In the worst case, when positions of molecules received earlier are not remembered, RMI calls have to be done for each molecule a CPU owns, with approximately half of the other molecules. The number of RMI calls can then increase to n^2 /2c (where n is the number of molecules and c is the number of CPUs).

A consequence of message combining is that the amount of data send/received in each RMI is greater than with a naive algorithm. But still it is faster, because the time loss during the latency does not increase that much.

The positions received are kept in a four dimensional array. Updates of molecules on other CPUs when calculating the‘‘interf’ are also kept in a four dimensional array. These updates are sent after the total calculation of the ‘interf’, again using message combining. Each CPU receives the updates of all its molecules in one message. Sending the update afterwards is possible because the updates are not needed during the calculation of the ’interf’.

We also tried a pool of threads in order to make the communication faster. Java uses synchronized communication, thus when a RMI call is performed the CPU has to wait for an answer before it can continue with other work. By using a thread to execute the RMI call, the CPU can continue with its work, like starting another thread to handle another RMI call. The idea is that having several threads handling the RMI calls instead of waiting for each RMI call, the communication time can be overlapped with the computation time. In our case it did not increase performance. Probably the bottleneck was the marshalling / unmarshalling of the data.

Another optimization is to explicitly call the garbage collector from the application. By explicitly calling the garbage collector every CPU starts garbage collecting at the same time. So it is known beforehand at which point all CPUs will collect garbage. This is done instead of letting each CPU decide for itself when to collect garbage, because that can lead to unwanted side effects (each CPU can collect garbage at a different time and can delay the current iteration). Explicitly calling the garbage collector saves time. Also the garbage collector is not called on every iteration. Instead it is called every other iteration. This has as effect that only these iterations will take a little bit longer. This makes the process of garbage collecting more deterministic.

Updates of shared variables are accomplished with the use of barriers to synchronize updates.

Java’s Remote Method Invocation

The Java RMI implementation is almost the same as that of the Manta RMI implementation of water, except for the RMI which is done using Java’s communication structure. Also *message combining* is used to reduce the amount of communication. Explicitly calling the garbage collector is also done. At the start of the application the program maintains an array of worker−interfaces so that the communication with other workers becomes faster. Barriers are used to synchronize the updates of shared variables. We also tried a pool of threads but it did not quite work well.

7.4 Results

The graphs below show the performance of Water using Manta, Java and Orca with 1728 molecules, 2 iterations. Again with Java measurements with more than 4 CPUs were not possible because Java crashes. Also measurements with Java are done without the JIT compiler because with the JIT compiler Java gives wrong answers when run on more than one CPU. This is due to the large number of parameters that must be passed on. Somehow the JIT seems to pass the wrong (nil) values. This is a bug in the JIT compiler. Performance measurements can be found in Appendix C.

The multi−threaded version of Water with Java gave incorrect results due to a bug in Java. Sometimes updates to shared variables seem to get lost, while these updates were performed inside a synchronized block. So there are no results of the multi−threaded version of water, otherwise we would be giving results that are not very reliable.

From figure 7.1 it can be seen that Java without the JIT is a factor 5 slower than Manta and a factor 17 slower than Orca on one CPU. Manta is a factor 3.37 slower than Orca on one CPU. With the JIT compiler on one CPU (then the JIT performs well) Manta is only a factor 1.26 faster and Orca a factor 4 faster. The cause of the Manta and Java system being slower than the Orca system on one CPU is, especially, due to the better optimization of floating point calculations. Another factor of influence is the Java language itself.

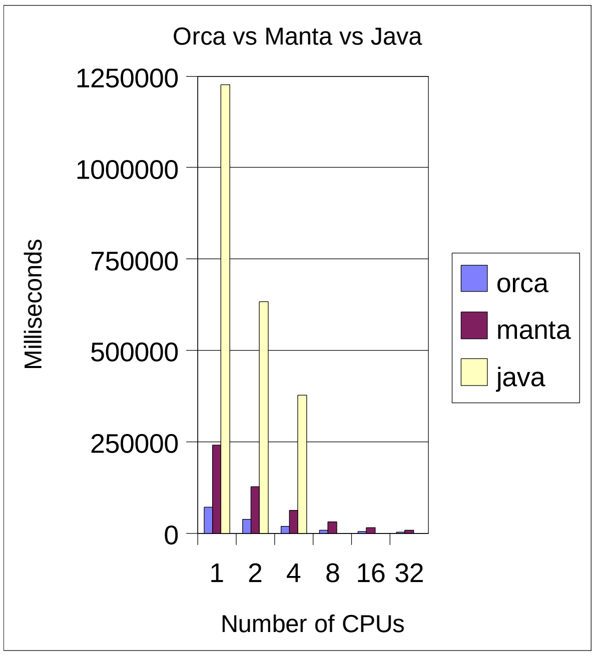


Fig. 7.1

Both Manta and Orca gained almost linear speedup. From figure 7.2 it can be seen that both achieved a speedup of 30 on 32 CPUs. From that figure it can also be seen that the speedups of Manta are almost the same as the speedups of Orca. Java also achieves good speedups; 1.9 on two CPUs and 3.2 on four CPUs (figure 7.4a). Thus Java reaches almost linear speedup too. The efficiency of Manta fluctuates between 0.94 and 0.99 (figure 7.4b). So the efficiency is high and stable. With Java the efficiency for 2 and 4 CPUs is also quite high (0.97 on 2 CPUs and 0.81 on 4 CPUs). After 4 CPUs the Java system fails.

The explicit call of the garbage collector also improved the results of Manta. For 4 to 16 CPUs the explicit call of the garbage collector improved the results by approximately 2%−5%. For 2 and 32 CPUs it made no difference. With Java it did not improve the results much. With 2 CPUs the improvement was about 0.3%.

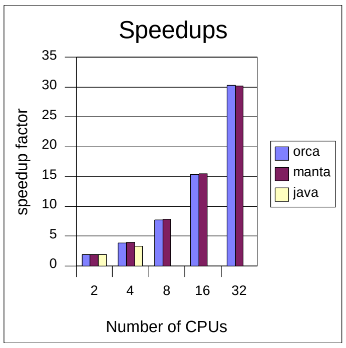


Fig. 7.2 Water

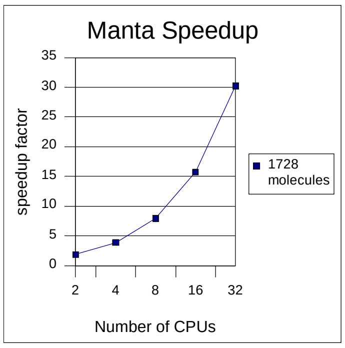


Fig. 7.3a Water

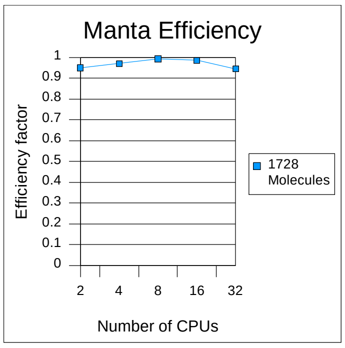


Fig. 7.3b Water

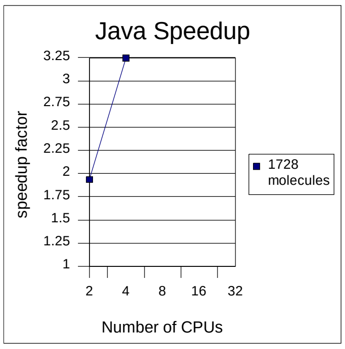


Fig. 7.4a Water

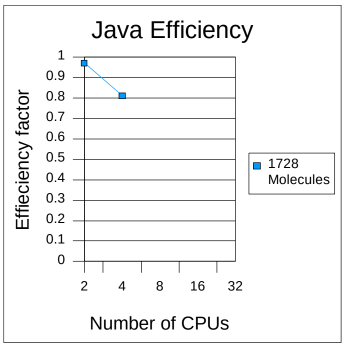


Fig. 7.4b Water

Still the Manta system has a loss of approximately 6% on 32 CPUs. This loss is caused by some communication overhead. This can be seen from figure 7.5 where ’fptime’ is the time necessary to calculate the inter molecular forces and the potential energy, and the‘‘intraf + predcorr’+ misc’ is the time to calculate intra molecular forces, predictions, corrections etc. Calculations of the inter molecular forces and the potential energy acquire most of the time as can be seen from the figure. During these calculations also some preparations for later communication are made. Temporary results are kept in four−dimensional arrays in order to send them later to other CPUs (message combining). These preparations can be seen as part of the communication.

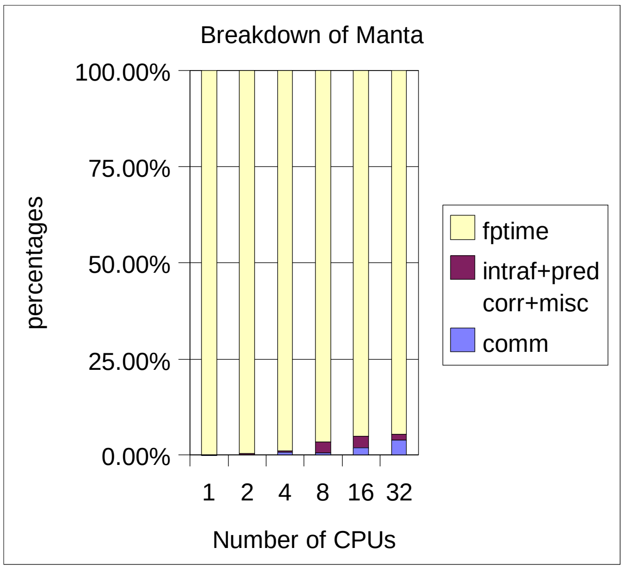


Fig. 7.5 Water

8 Checkers

8.1 Introduction

Checkers has been a popular game all over the world for hundreds of years, and many variations exist nowadays. Only two versions, however, have a large international playing community. What is commonly known as Checkers (or draughts) in North America is widely played in the United States and the British Commonwealth. This variation is played on a 8x8 game−board, with checkers moving one square forward and kings moving one square in any direction. Captures take place by jumping over an opposing piece, and a player is allowed to jump multiple men in one move (known as double jumps). Checkers promote to kings when they advance to the last rank of the board. Another major version of Checkers (the so−called International Checkers) is popular in the Netherlands and the former Soviet Union. This game is played on a 10x10 board, where checkers may capture pieces backwards as well and kings are allowed to move many squares in one direction in a single move, just like bishops in Chess. The program mentioned in this document implements the 8x8 Checkers variant.

8.2 The sequential algorithm

The sequential Java Checkers application is based on a checkers program written by Don Dailey for the Cilk parallel C/C++ language library [20]. The original Cilk code uses the negamax game−tree searching algorithm with alpha−beta pruning and implements iterative deepening. The quiescence heuristic is used as well. This program was ported to Java and extended to add support for double jumps. Furthermore, the game node evaluation−function was improved and a transposition table is used to reduce search overhead.

8.2.1 Overview

The entire system consists of two major components, a client application and the main search engine. These two applications run completely separate from each other. Whenever the client wants the search engine to determine the best move for a given game situation, it sends a (Sun) remote method invocation with this game position (together with the minimal search depth) to the engine. The game engine then starts to search the game−tree for the best move. As soon as it is finished, the result is sent back to the client. Figure 8.1 shows the main application structure.

8.2.2 Negamax with alpha−beta pruning

One of the most efficient serial game−tree search algorithms is Negamax in combination with alpha−beta search, because it can avoid searching the entire tree by proving that certain sub−trees need not be examined (can be pruned). The alpha−beta algorithm used can substantially reduce the size of the tree, especially if the best moves are considered first. The exact algorithm is explained in [1]. Techniques such as iterative deepening, the use of a transposition table, refutation, chopper and the history heuristic can greatly improve move ordering. The first two are used in our application [28, 30].

Many other search algorithms exist nowadays, such as SSS* and MTD(f), but these are not as common as alpha−beta search and are more difficult to implement [27, 29].

8.2.3 Evaluation function

Since Cilk's tree−node / game−position evaluation function was rather primitive (counting the number of checkers and kings and multiplying each with a predefined value) we decided to add some more advanced heuristic components based on Chinook's evaluation function [21, 22]. Our function incorporates the checker−advancement heuristic, and takes back−row occupation and piece−centering in consideration as well to improve the application's level of play.

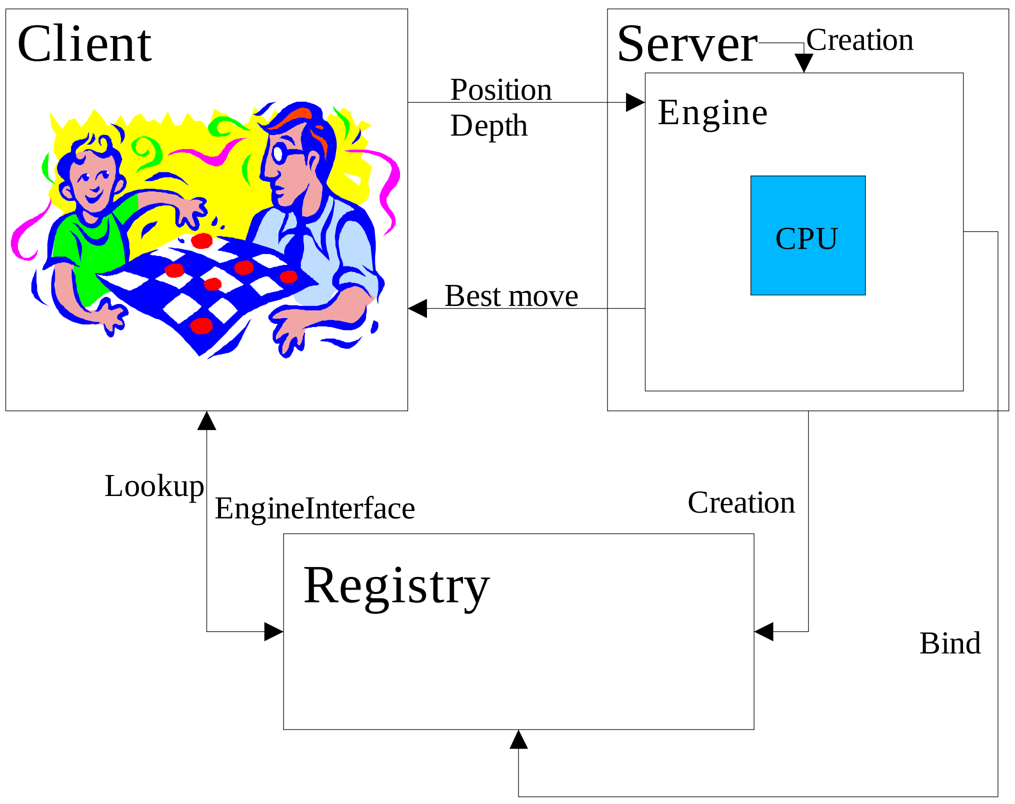


Fig. 8.1

8.2.4 Transposition Table

Although the game’s search space is usually referred to as a tree, it is actually a directed acyclic graph, because game positions can be reached via different sequences of moves. These positions are called transpositions. A transposition table (TT) is used to save information about visited nodes, in the event that the same nodes are revisited later, which allows one to eliminate the need to re−search large portions of the ‘tree’.

The transposition table is basically a hash table of previously evaluated game positions, also called nodes. After a node is searched and evaluated, we create or update the corresponding entry in the TT. The information stored includes the node’s score, the move that leads to this score, the search depth and a special tag value. Then, before searching a node, we first check to see if it is already present in the TT. If this is the case, both tag values correspond and the node’s depth of search was at least as deep as the one to score now, we can skip the search and return the evaluation score immediately. However, often when we get a TT hit (a succeeding lookup), the depth is insufficient for the current search. But even in this case the TT is still useful because it gives us the best move found by an earlier search, and often the best move at a shallower depth is the best move at a deeper depth. So, by using this move as the first to evaluate, we increase our chances to predict the best move for the given position which reduces the search overhead because more nodes are likely to be pruned.

Since the TT is implemented as a hash−table, we need to be able to generate a hash−code for each game position. We did this as follows. First, a table with random values is generated for all board fields and all different game pieces. Then, to calculate a position’s signature, the various values are XOR−ed together, which allows us to determine the hash−index by taking this number’s first few bytes. The last few bytes determine the entry’s so−called tag value. The idea of using XOR−ed values allows the programmer to incrementally determine a position’s signature [31]. When moving a piece from position A to B, the piece has to be XOR−ed with A’s signature and XOR−ed with the same signature for position B. These two XOR operations remove the need to recalculate the new signature from scratch, which would have taken significantly more time.

Whenever two different game positions coincidentally map to the same index, they will usually have another tag value, so their difference will be detected. Although there is a non−zero chance that multiple positions will map to the same signature, most game−playing applications just accept this risk.

8.3 Parallelism

All versions of Checkers described in this paper use the two components mentioned earlier. The client application is a straightforward graphical Swing program that is exactly the same in each variant. It shows the checkers board on which the user can click with the mouse to move a certain piece. A pull−down menu option gives the user the opportunity to send a request to the engine to determine the best move for the current board position. So, although there is communication between these two components, the actual parallelization takes place where it matters: in the game search engine. From now on, we will no longer look at the client application, but focus on the search engine instead. The overall system as used in the various parallel implementations is shown in figure 8.2.

8.3.1 Multi−threaded

Like the sequential program, it is the Server class that starts the engine. The engine on its turn creates the transposition table and all CPU slave threads. Because the search task needs to be split in various sub−tasks (in order to solve the problem in parallel), a job−queue is created [24]. This data−structure is used to temporarily store jobs, which can be fetched by any slave thread on a remote CPU whenever it wants a job to work on. Jobs consist of a game position in combination with a valid move for that position. The *resulting* game position is evaluated by the slaves. With the multi−threaded application, all valid moves from the top−level game−position are put in the job−queue. Then, when a job is finished, its evaluation score is put in another centralized queue maintained by the engine called results−queue.

Young Brothers Wait

An advanced search algorithm is used in the multi−threaded application and most parallel Manta variants to reduce search overhead, called the *Young Brothers Wait Concept* [26]. Quote:

YBWC: The search of a successor *v*.j of a node *v* in the game tree must not be started before the leftmost brother *v*.1 of *v*.j is completely evaluated.

Although YBW restricts the use of parallelism, it can strongly narrow the search window for the right sons of *v* because of the result of *v*.1, possibly leading to more cut−offs and thus reducing search−overhead.

As said, there is a serious disadvantage to YBW. Parallelism can only start when the leftmost node of the tree has been evaluated. Then its brothers may be evaluated in parallel. However, in Checkers, the number of brothers is usually quite small compared to the number of processors. In effect, YBW causes a point of time at which all but one processor are idle for every inner node of the leftmost variation (called synchronization nodes). Unfortunately, this effect is worsened by our use of iterative deepening. Therefore, we use YBW in our application only at the root of the game tree, to restrict the number of synchronization nodes and to reduce processor idle time.

Abort

Whenever the alpha−beta algorithm discovers that at least one child has a greater score than beta, there is no need to search the rest of the children (this is called *failing high*). Thus, if other threads are already working on these children they have to be signaled to stop by sending an *abort* message [1]. However, this is not applicable to our program, since tasks are only distributed at the top of the search tree. Slaves do not further distribute their work, and can simply stop iterating over the children and return if one node fails high. Only the engine distributes work, but its children cannot fail high because the beta value of the game−tree’s root position is defined as ∞ (infinity). No child will have a higher score than that, so all children will have to be evaluated.

8.3.2 Manta Remote Method Invocation

We have built multiple applications using Manta’s ’remote’ keyword which use different ways of job distribution (load balancing) and different schemes to distribute the transposition table.

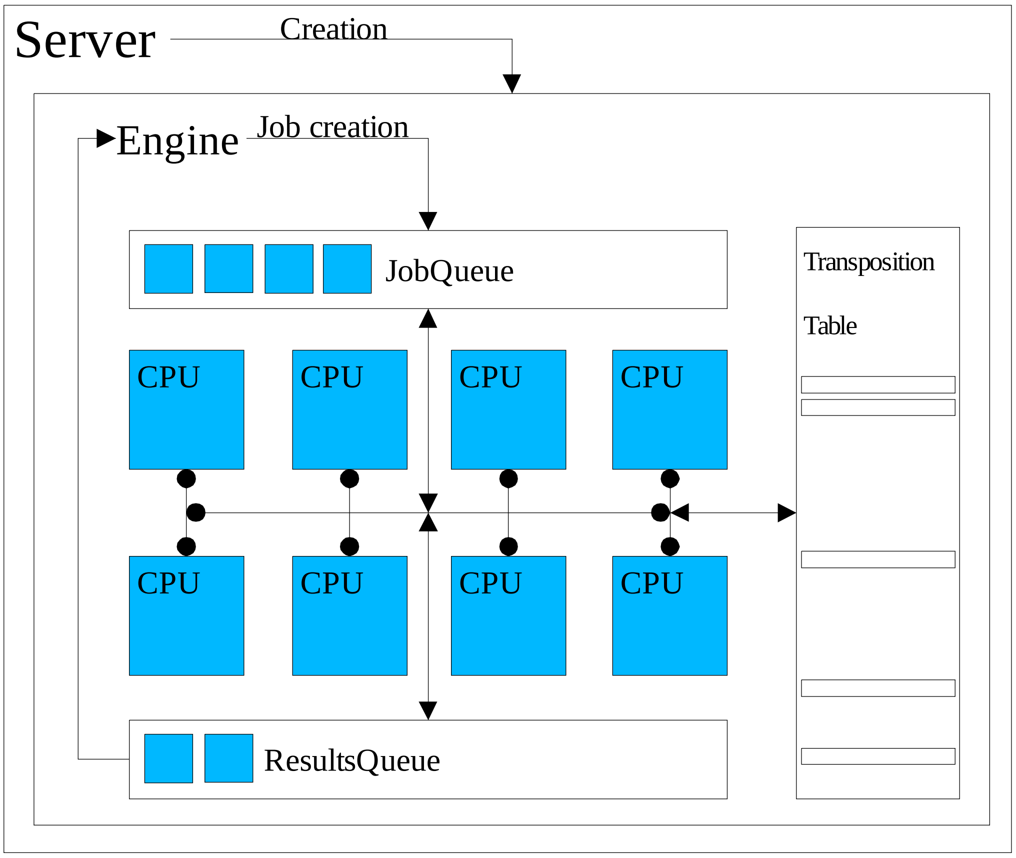


Fig. 8.2

Root Depth Vs. Fixed Depth search

Initially, all valid moves from the top−level game−position were put in the job−queue. We will call this root depth (RD) search. It works well when using only few slaves (CPUs), but because the average tree branching factor of Checkers is quite small (especially on a 8x8 board) the number of jobs becomes too small to equally distribute the jobs among the available resources when using more than four CPUs. Therefore, the algorithm was altered so that the engine would first search to a specific depth before putting the jobs in the queue. From now on, the term fixed depth (FD) search will be used to refer to this algorithm. Although RD search does not scale well, its relatively small search−overhead is the main reason to mention it here.

FD search greatly increases the number of jobs, thus improving the load−balancing. Unfortunately, it has some disadvantages as well. First of all, the engine has more work to do since it has to deliver jobs at a deeper depth, which means it has to go into recursion to determine the positions and their valid moves at that depth. Parallel execution can only start when all jobs have been created. Moreover, a special tree has to be constructed to be able to determine the root’s evaluation score when all slaves have finished. Then all tree leaves have their appropriate values and the engine can work its way back to the root of the tree to calculate its score and determine the move that leads to that score.

Another substantial problem introduced by the fixed depth search has to do with the alpha−beta algorithm itself. With the original RD search the engine immediately adjusts the root’s alpha value whenever a slave has finished a job. The engine then forwards this so−called alpha update to the job−queue, so jobs that are to be evaluated in the future would have a bigger chance of being pruned because of the reduced search window. More importantly, the engine informs all working slaves about the alpha update, which may reduce their search window as well.

When using FD search these alpha updates are not forwarded to the job−queue and/or the working slaves, since the slaves do not share the same parent game position. The alpha update is only of concern to those slaves that have the same parent node. More administration work would be necessary to locate all slaves that are working on the child−jobs of that parent. Because the alpha updates are not propagated to the slaves, they all start working with a maximum search window. This means less sub−trees can be pruned and (much) more nodes have to be evaluated, implying search−overhead.

Distributing the Transposition Table

There are various ways to distribute the transposition table. Much research has been done to find the best distribution strategy, but results show that it depends strongly on the kind of application [23, 24]. For the Manta RMI application, we have implemented 3 different ways:

- Local transposition tables
- Partitioned transposition tables
- Replicated transposition tables

Using a local transposition table on every processor has the advantage that each table lookup or update can be performed without the need for communication. So this distribution scheme has no communication overhead on behalf of the TT at all. Unfortunately, the chance of a TT hit decreases when the number of CPUs is increased because the chance that the CPU has already evaluated the same game position before decreases. This reduces the effectiveness of the TT, inducing search−overhead.

With the partitioned scheme, each CPU stores an equally sized part of the entire TT. Both lookups and updates are done remotely using RMI. Only if the lookup or update takes place in the part of the TT that resides on the same machine that wants to do the lookup or update, it can be performed without communication. With an increasing number of CPUs, this chance is becoming smaller, which may lead to communication overhead. However, with a large number of machines, the amount of communication scales almost linearly with the number of machines. Especially the lookups are expensive, because the client processor has to wait for the answer to arrive before it can continue (synchronous communication). Table updates can be done asynchronously, but unfortunately the Java language does not support asynchronous communication. To simulate asynchronous communication, a separate thread can be created to perform the RMI while the main alpha−beta thread can continue its execution. This solution was not successful because the overhead of thread creation and thread cleanup completely overshadows the advantage of the asynchronous communication.

When using the replicated TT variant, each machine has its own copy of the TT. This has the advantage that table lookups can be done locally, without communication. On the other hand, when a TT node insertion or update takes place, a broadcast message has to be sent to all slaves to update their replicas. Since a broadcast is not available in the Java language, it had to be programmed by hand. Our solution is derived from another application, ASP [25], using binary tree broadcast. Every slave sends RMI calls to the two underlying slaves of the imaginary tree, until all slaves have received the message. The main problem with replicated tables therefore is the communication overhead required for update operations.

8.3.3 Sun Remote Method Invocation

Apart from the differences mentioned earlier between the Manta and Sun system to create slave threads, our Sun version of Checkers does not significantly differ from the Manta one. Because the Manta application performed best with a partitioned TT, we have implemented the Sun version with a partitioned TT only.

8.3.4 Optimization

We have used various methods to improve the speed of the Checkers application. Multiple strategies to optimize sequential execution time and to limit communication overhead have been implemented.

To lessen the amount of work for the *garbage collector*, most Java objects/structures have been constructed by using as few bytes as necessary. For example, the class that stores a game position uses bytes instead of integers for its board fields. Since many board positions are created and destroyed during game−tree traversal (see the amount of ’nodes evaluated’ in the results section 8.4), it is highly important to reduce the board object its size. Another way to reduce execution time was by using a long integer (8 bytes) for a piece move, instead of using an object. Various kinds of bit manipulations are performed to store and retrieve (double) moves, cutting down on memory usage and object method calls.

In order to minimize communication overhead, *message combining* was used. Instead of sending TT updates immediately, each machine stores the updates in a fixed−size buffer, sending the entire buffer as one single message when it is full. This optimization greatly reduces the amount of messages sent. Although the number of bytes sent remains unchanged, communication overhead is reduced considerably. On the other hand, message combining increases the chance of reading stale information from the TT, implying search−overhead.

Another method used to limit the amount of RMI calls was by use of a so−called *lookup−threshold*. This threshold value specifies the minimum depth (counting bottom−up) at which a TT lookup may be remote. If the depth of search is less than that specific value, performing a remote lookup may be even slower than just calculating its value locally. Therefore, the TT is neglected and the node's score is determined on the client machine itself. The same holds for a TT update. Updates are always stored on the client machine, but only propagated to the other processor(s) if the depth of search was above this threshold value. Although the lookup and update thresholds could have been different values, we have chosen them to be the same.

8.3.5 Application testing

The following pages discuss the performance results of the various versions of Checkers. Each page describes the properties of one particular implementation.

To measure the performance of the various Checkers implementations, the existing Swing client application could not be used. Instead, a simple test object was written that sends RMI requests to the engine asking for the best move for multiple game positions. A total of 6 RMI requests are sent; three different game settings and their follow−up positions. The follow−up positions are used in order to make strong use of the TT. The three game settings consist of the game opening, a random mid−game board position and one position which has only one forced move, thus reducing the tree branching and − consequently − the number of jobs, considerably. All measurements are available in numeric format in Appendix D.

Debugging the Checkers application has proven to be somewhat troublesome. As in [1], determining whether the application is correct can be very hard. Not only can the multi−processor variants yield different moves as 'best move' during the tests, even the evaluation score for that move can fluctuate. Therefore, a minor programming mistake can easily remain unnoticed. Another important task in testing the various Checkers implementations deals with tuning the application. Clearly, determining the best performing values for the lookup threshold, the message buffer size, the FD search depth and the best way to use YBWC can be time−consuming undertakings. Obviously, these issues do not only apply to Manta but to Sun's Java and other programming languages in general as well.

8.4 Results

Figure 8.3a shows the breakdown of the sequential and multi−threaded implementations. It consists of seven different timings, namely the node evaluation time, the move generation time, the time necessary to create the next tree node (thus perform a move), the TT lookup and update times and the idle time. The miscellaneous value indicates everything not mentioned earlier, most importantly the time spent in the alpha−beta algorithm itself, the recursion, tree node sorting, etc.

Fig. 8.3b and 8.3c show the speedup and efficiency factors, while 8.3d shows the total number of tree nodes that were evaluated during the performance measurement.

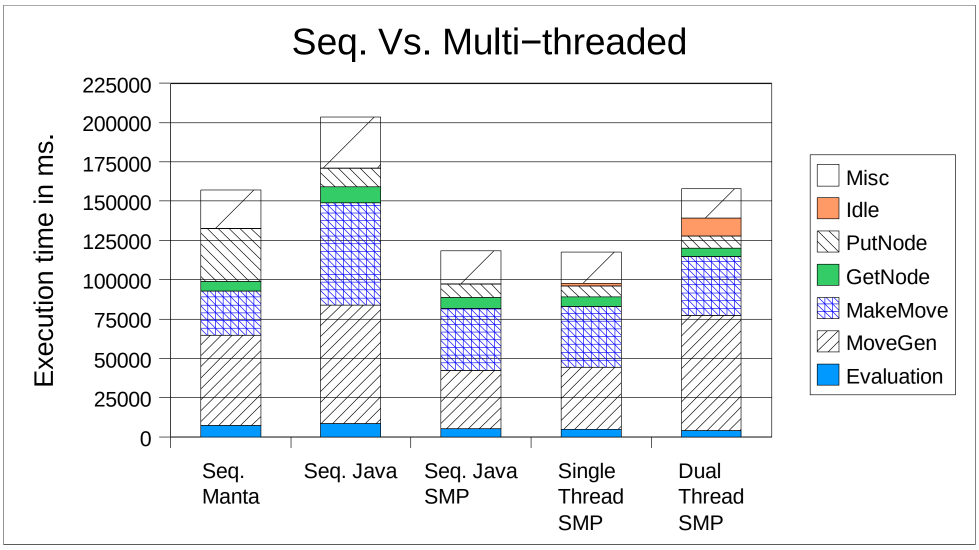
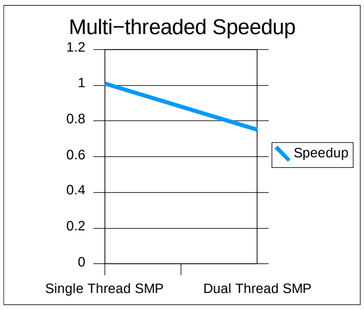
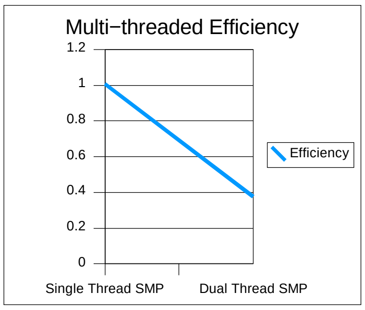


Fig. 8.3a





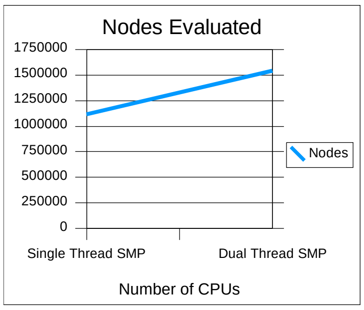


Fig 8.3b   Fig. 8.3c   Fig. 8.3d

With the sequential program, Manta outperforms Sun by a factor of almost 1.3.

As can be seen, the multi−threaded application using just one thread performs equally well as the strictly sequential variant, evaluating just as many nodes. Surprisingly, the dual−threaded program performs worse than the one using a single thread. This is the effect of the increased number of nodes that had to be evaluated. The dual−threaded variant is also suffering from some idle time, most probably imposed by the forced−move game position from the test−set which yields only one job, keeping one processor idle.

Next are the results of the Manta RMI variant using root−depth search and a partitioned transposition table. Because a message buffer size of 32 in combination with a lookup threshold of 2 gave the best results, we will only show those measurements. When using a smaller lookup threshold value than 2, the total execution time builds up although the amount of evaluated nodes decreases. That is because the program’s bottleneck shifts from a search−overhead to a TT lookup overhead (not shown). Increasing the threshold value importantly enlarges the search−overhead. Modifying the message buffer size does not significantly alter the application’s performance.

The breakdown shows a few new timings. GNHandling and PNHandling denote ’TT node lookup handling time’ and ’TT node update handling time’ respectively, which are both of minor importance to the total execution time. The ’Message’ timings show the amount of time needed to store the TT updates in the message buffer. All measurements are performed by using Java’s timing function ’CurrentTimeMillis’ which makes it possible for each processor to store its own breakdown bar. After all tests have completed, those bars are added up and the average values are determined. Unfortunately, this solution leads to a problem. In order to measure idle time, the idle time counter starts as soon as a job is finished and will only stop when a new one has arrived. Thus, the idle time measurements are not quite accurate since engine idle time and the time necessary to show the search results are included as well.

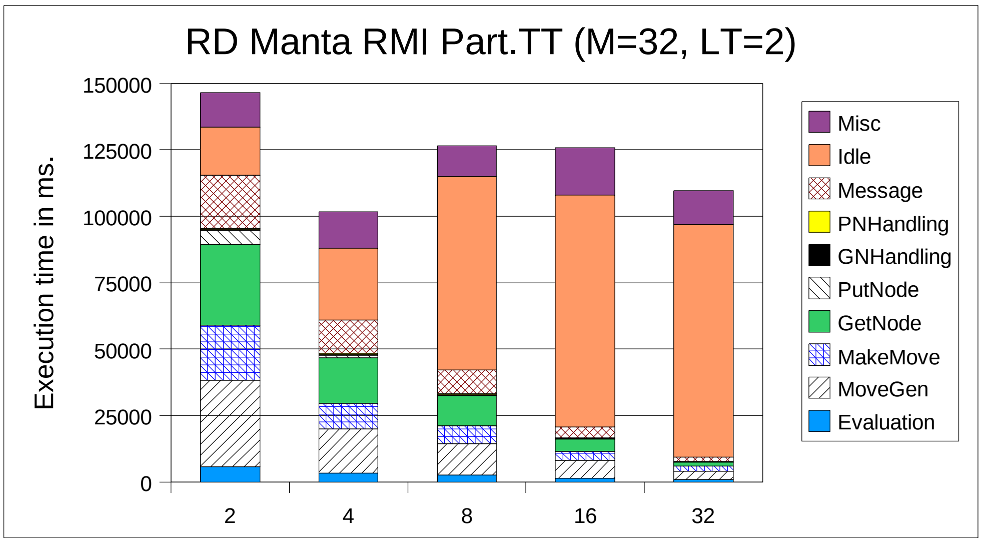
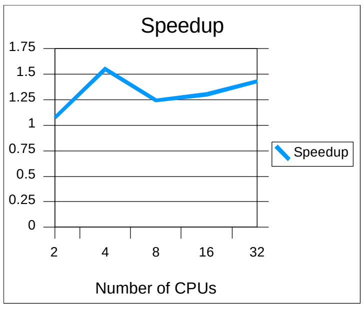
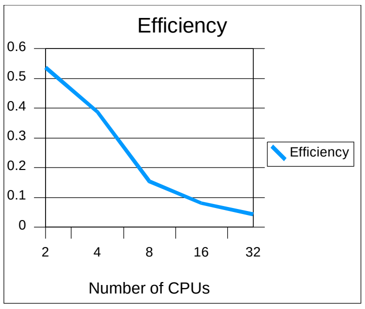


Fig. 8.4a





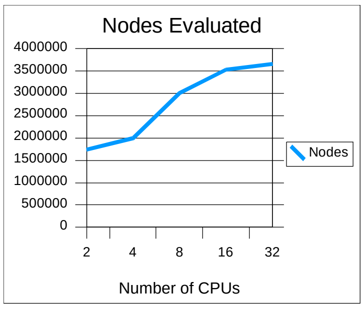


Fig. 8.4b Fig. 8.4c Fig. 8.4d

The graphs clearly show the most important problem of the RD search algorithm; the shortage of jobs, leading to much idle time. With an increasing number of CPUs, the relative idle time becomes larger. With 32 machines, idle time contributes for 80% to the total execution time. As an effect, the maximum speedup (1.55) is achieved by using only four machines. It is also due to the search−overhead. Notice, though, that the amount of nodes evaluated is not that much larger than the sequential Checkers variant. With 32 processors this amount increases just slightly and ’only’ 3.28 times as many nodes are evaluated, a relatively small value compared to the FD search applications.

Another interesting application feature that can be drawn from the graphs is that the portion of miscellaneous time remains nearly constant over a changing number of CPUs, while the other tasks scale inversely proportional to the increasing number of processors. This is probably due to the overhead imposed by the program’s engine which takes care of job creation and distribution. The time spent in the engine remains constant over the number of processors. Because less incoming RMI calls have to be handled per CPU, the application's total execution time slightly improves with an increasing amount of processors.

Fig. 8.5a shows the performance of our Manta program using the fixed depth search algorithm in combination with the Young Brothers Wait Concept. Here, a message buffer size of 8 performed best. A different threshold value has the same negative results as with the RD search variant. When using more than four CPUs, an increased message buffer size leads to a large search−overhead, since the TT is less often updated and thus less often useful (not shown).

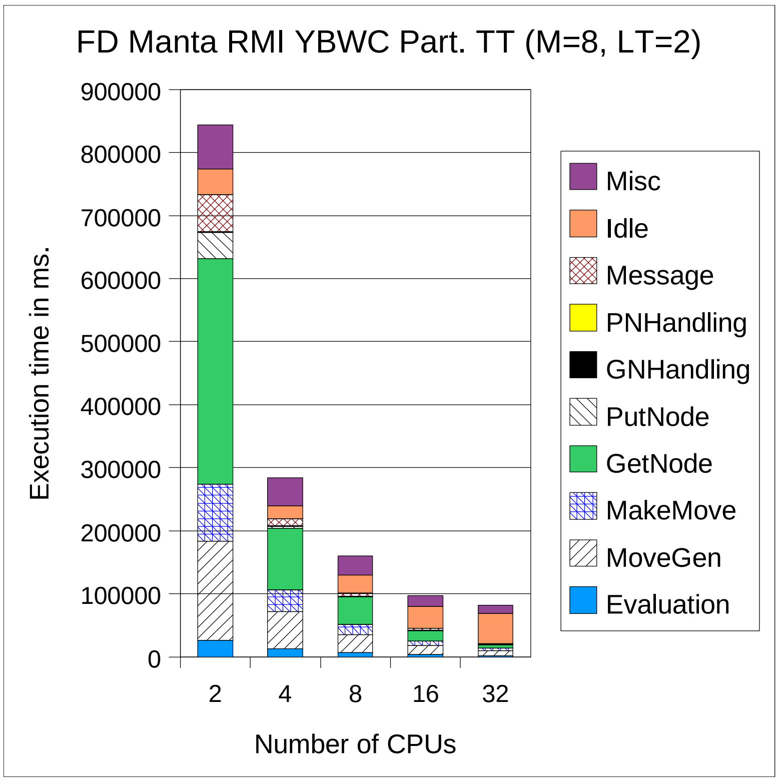


Fig. 8.5a

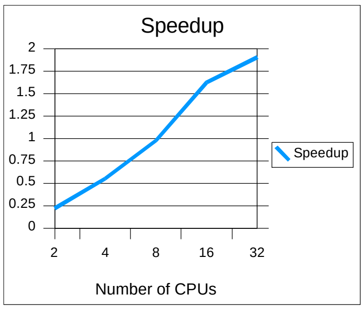


Fig. 8.5b

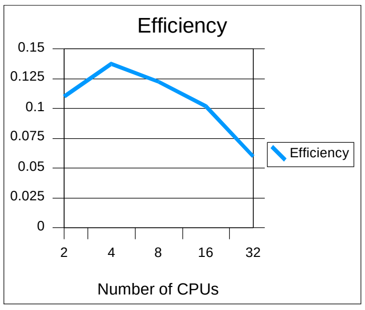


Fig. 8.5c

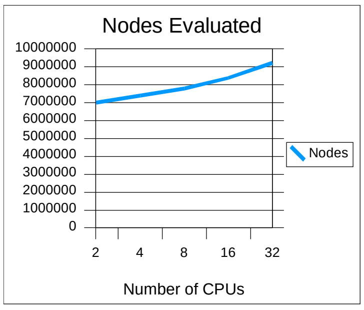


Fig. 8.5d

The graphs clearly confirm that FD search is able to distribute the search problem more effectively. Fig. 8.5a shows that, although the relative idle time is increasing with the number of CPUs, the lack of jobs no longer stalls the speedup factor at four machines. Unfortunately, the use of FD also increases the search space, which is the main reason that a maximum speedup of just 1.91 is reached with 32 CPUs. More than 9 million nodes are evaluated in that case, 8¼ times as much as the sequential variant. Thus, when compensating for the search−overhead, a max. speedup of 15¾ (on 32 CPUs) is reached. The poor efficiency then is mostly due to the relatively large amount of idle time, even with FD. Our FD search creates jobs at 2 deep, counting from the root of the game tree. When job creation would take place at a deeper depth, more jobs would have been created leading to less idle time, but to a large search−overhead as well. Our choice is a search vs. idle−time −overhead compromise, creating enough jobs to equally distribute them among the resources, most of the time.

The following graphs show the results of the Manta application without YBWC. The rules that applied for the Checkers variant *with* YBWC apply to this variant as well, so we show the results for the same message buffers size and TT lookup threshold.

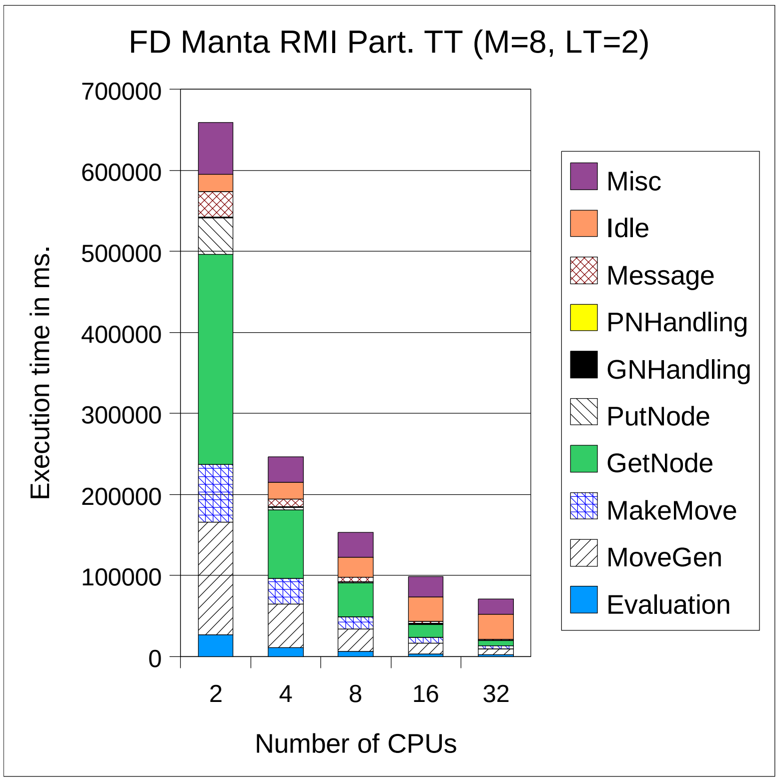


Fig. 8.6a

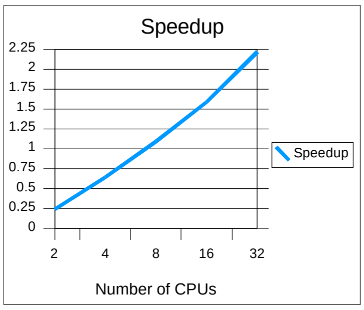


Fig. 8.6b

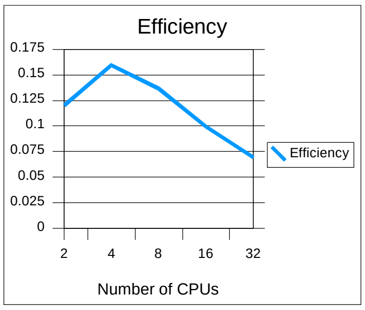


Fig. 8.6c

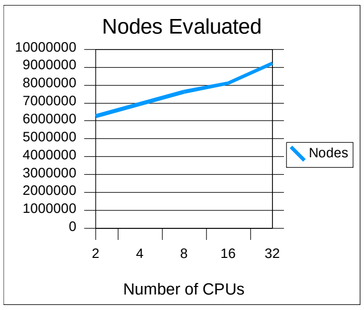


Fig. 8.6d

This variant of Checkers performs best. Without YBWC, less synchronization between the machines is necessary. Also, the number of evaluated nodes does not differ much. Apparently, our application does not benefit that much from the reduced search window when using YBWC. A maximum speedup of 2.22 is reached using 32 processors. Corrected for the number of evaluated nodes, the score would reach 18.35.

The breakdown shown in Fig. 8.6a closely resembles the one in Fig. 8.5a. Except for the miscellaneous and idle times, all timings scale inversely proportional with an increasing number of CPUs. Interestingly, the highest efficiency is scored using only four CPUs in both applications. This is a direct result of the extended division of the transposition table. When using four processors, the chance that a TT lookup is local reduces by a factor 2 (from ½ to ¼). Because the threshold value is used only for remote TT access, much less lookups and updates take place. This is the main reason for the improved efficiency, together with the fact that idle time only increases with more than four processors.

The next graphs show the results of the Manta application without YBWC using a replicated transposition table with a message buffer size of 128 and a threshold value of 4. Lowering either one of these values results in a massive broadcast overhead. As can be seen, two new values have appeared in our breakdown, namely the broadcast timings and the time spent in propagating an alpha update.

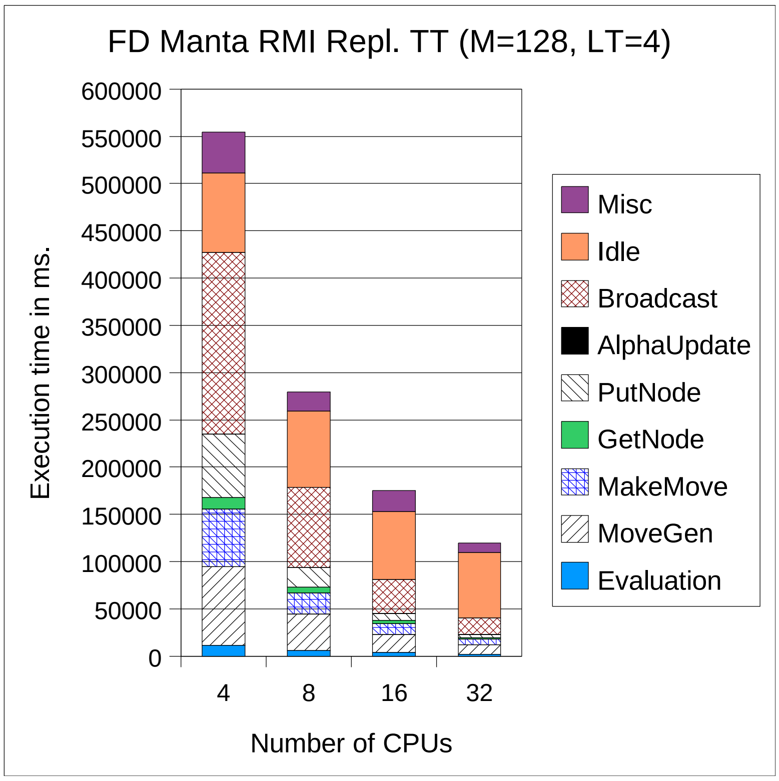
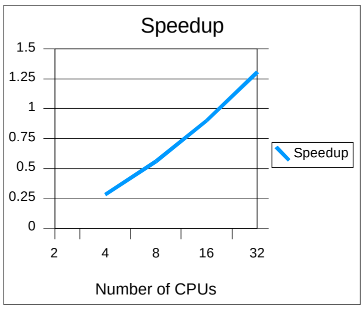
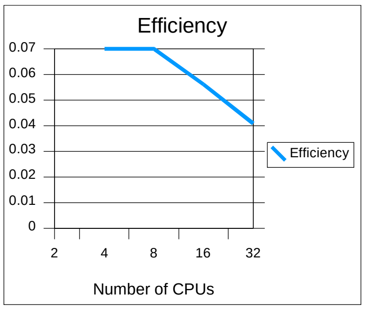


Fig. 8.7a





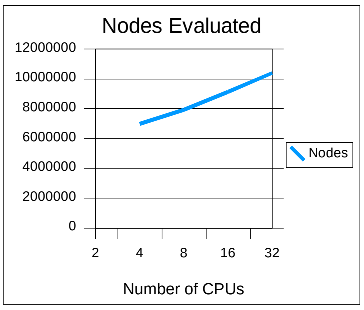


Fig. 8.7b Fig. 8.7c Fig. 8.7d

Unlike our partitioned TT variants where TT lookups were relatively expensive, here the TT update broadcast has a big impact on the total execution time. With a replicated TT, a lookup is always local and can be done relatively fast. The TT updates on the other hand require a broadcast. The large message buffer size and threshold values are mostly responsible for the search−overhead which is even worse than with the partitioned TT applications. A maximum speedup of 1.31 is reached using 32 processors. In that specific case, processor idle time contributes for 57% of the total execution time. Corrected for the number of evaluated nodes, the max. speedup would be 12.2.

One of the main reasons for the search−overhead with our parallel variants of Checkers, is that the applications do not sort the tree nodes optimally. With our application, only the TT is used to sort the possible follow−up nodes. If, for example, the history heuristic would have been used in combination with the other heuristics that *are* implemented, node sorting would be improved. This would allow more sub−trees to be pruned, especially when using multiple CPUs, reducing search−overhead.

Fig. 8.8a−d show the measurements for the FD Manta application with local transposition tables.

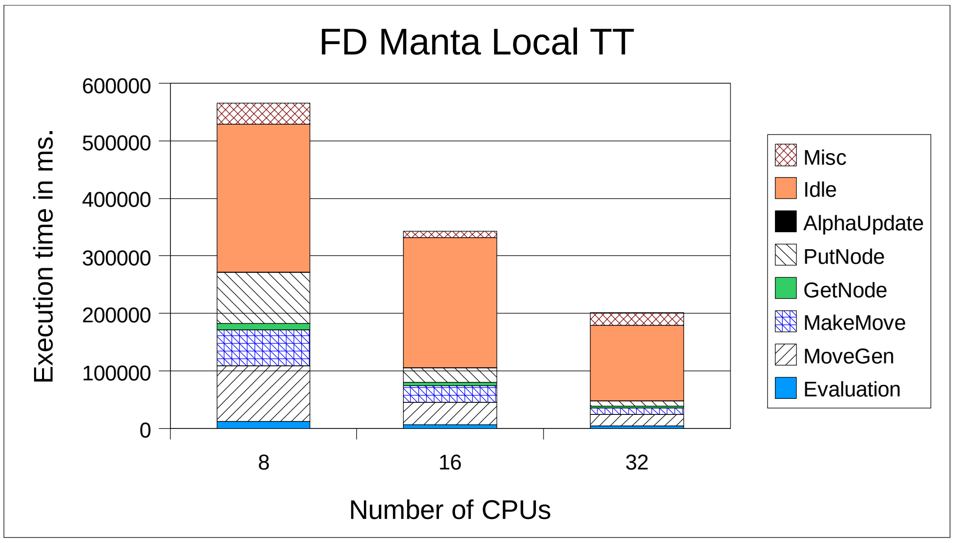


Fig. 8.8a

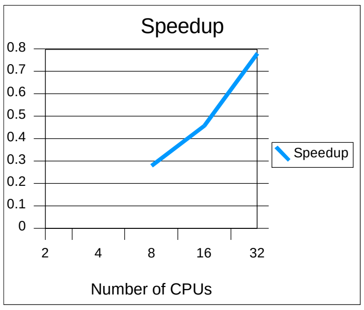


Fig. 8.8b

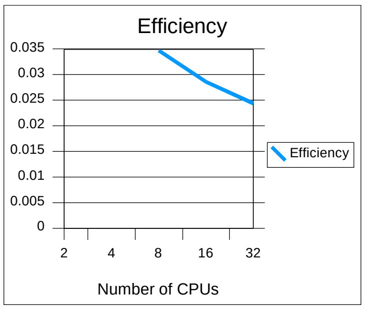


Fig. 8.8c

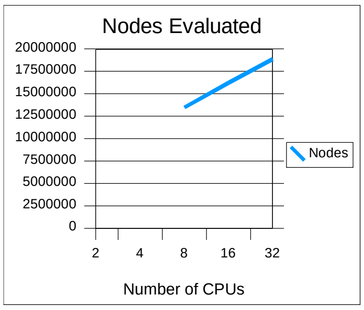


Fig. 8.8d

Graph 8.8d immediately shows the main bottleneck of this local TT variant: the huge search−overhead. Because the machines do not share their information with each other, they often perform the same work. With 32 CPUs, almost 17 times as many nodes are evaluated. As a direct result, a speedup could not be reached. Even with 32 processors, the application takes longer to complete than the sequential variant. A speeddown of 0.78 is scored, while the speedup corrected for the number of evaluated nodes would reach 13.21.

The next graphs show the results of the FD search, partitioned TT application written using the Sun RMI system. This program was compiled and run with Manta to obtain timing measurements, since the Java runtime system crashes. The best performance was achieved with a message buffer size of 8 and a threshold value of 2. Changing these values did not significantly alter the behavior of the application.

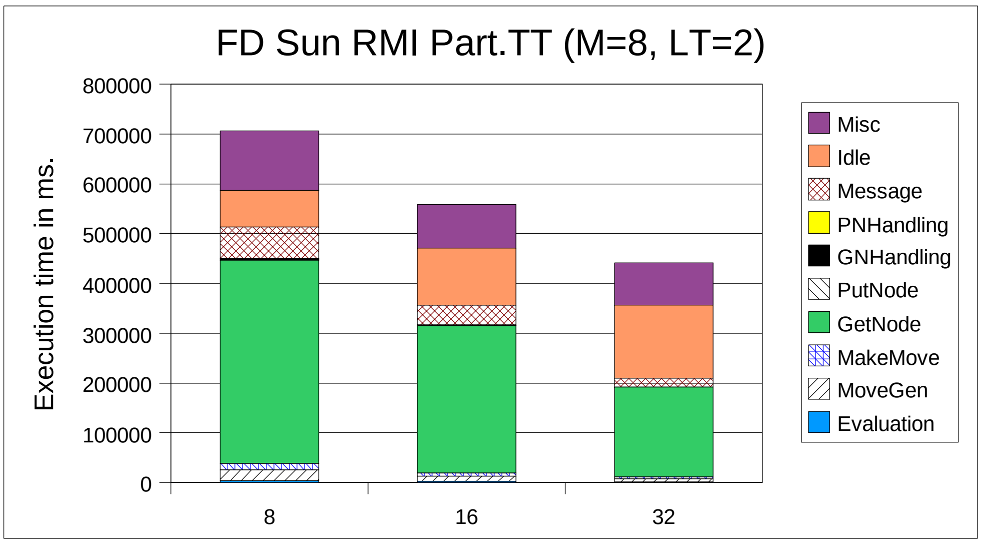
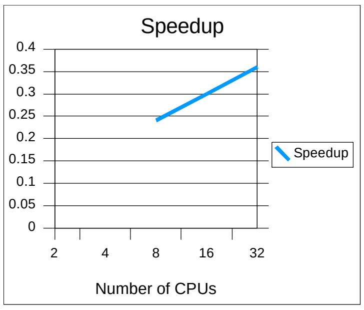
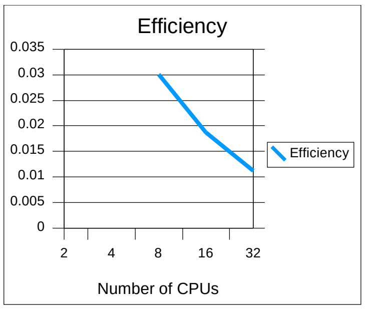


Fig. 8.9a





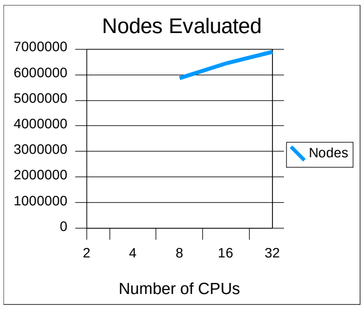


Fig. 8.9b Fig. 8.9c Fig. 8.9d

The graphs show that this is the worst performing application of all variants. No speedup is achieved at all, mostly because of the TT lookup overhead and the increasing portion of idle time. The huge communication overhead is due to the fact that instead of the Myrinet network, fast ethernet is used in this application. Corrected for the number of evaluated nodes, a max. speedup of just 2.20 is achieved.

Taking all performance measurements in consideration, we can conclude that our parallel Checkers implementations run rather inefficiently. The Sun Java application and the ’Manta Local TT’ version both fail to reach a speedup whatsoever, suffering from a communication−overhead and a search−overhead respectively. Our ’RD Manta RMI Part. TT’ Checkers implementation performs better, reaching a maximum speedup of just 1.55 using four CPUs. Its main problem is the shortage of jobs, leading to load imbalance. The fixed depth replicated transposition table variant is suffering from the lack of an efficient broadcast implementation. Other reasons for the poor performance include the lack of asynchronous communication and the large contribution of idle time in relation to the total execution time. The use of the Young Brothers Wait Concept does not improve the performance of our program because the amount of evaluated nodes does not decrease significantly, while extra synchronization between slaves is introduced. For this reason, the fixed depth partitioned transposition table variant without YBWC achieves the highest speedup of 2.22 using 32 CPUs.

9 Problems

In this section we will discuss some problems encountered. They will be distinguished in the following categories: Java's usability for developing parallel programs, programming problems and bugs in Sun's JDK and the Manta system.

The usability of Java for developing parallel applications

The lack of asynchronous communication in Java can have a great impact on the performance of the applications. We have used threads to simulate asynchronous communication but that gives rise to other kinds of overhead like thread creation, garbage collection and context switching. This reduces the benefits of asynchronous communication.

The fact that broadcasting, multi−casting and all−to−all communication is not supported in Java can also decrease the performance of the applications. Having to perform an RMI call for every connection is very expensive and time consuming.

The lack of pointer arithmetic in Java can lead to some loss of performance. Sometimes pointers can be very useful and fast especially with arrays. With SPLASH radix sort extra variables had to be created to temporarily store results.

Object creation is very expensive. Creating temporary objects lessens the performance of the application. So for a better performance the creation of temporary objects should be kept minimal, or should even be avoided. This is against the OO paradigm.

According to the OO paradigm all the data within a class should be private (data abstraction). If an object wants to access the data of another class, it should be accomplished through method calls. But this has a negative impact on the performance of the parallel application if these method calls are executed frequently. So one has to make a choice between good OO programming or speed.

The times at which the garbage collector (GC) in Java's runtime system is activated, is nondeterministic. This can sometimes lead to performance problems, especially when all CPUs collect their garbage at different times. When a CPU initiates the garbage collector, the running thread is blocked until the GC has finished. So, incoming RMI calls are put on hold and outgoing RMIs are only performed after garbage collecting. Thus, the process of synchronization between multiple CPUs can be delayed because one or more CPUs can be collecting garbage.

Developing parallel applications with Manta takes place in a more intuitive manner. The distribution of work and the creation of all the worker objects can easily be accomplished by a master object in Manta. The number of worker objects that must be created can easily be arranged in the master object. If more than one worker must be started in Sun Java, only one worker object can be started from a master object. The other worker objects must be started manually on every other machine.

Results from the Java implementation of water are different from that of the C−Orca version and the Manta version of water. This is caused by the difference in floating point handling in Java, Manta and C−Orca. Java has a strictly defined floating point model, 64−bit precision for doubles, whereas the floating point precision for C−Orca and Manta depends on the computer−platform on which they are run (80 bits on a x86 platform). This strictly defined floating point model of Java leads to loss of floating point precision and loss of performance. Loss of performance because Java repeatedly needs to convert the 80−bits precision into 64−bits precision on a x86 architecture.

Programming problems

There were a few problems that occurred in every application. One problem concerned the matter of file creation used in the Sun Java RMI versions to locate remote objects. The problem is that the time necessary to create the file and write the hostname is unknown at runtime. Thus, the objects that want to read from the file have to wait some time before they can read from the file. And waiting is something you don't want to do in a parallel program.

Another problem concerning the use of a file to read the hostname from (this is also necessary for slave objects which can find the master object by reading its hostname from a file) is that it often led to exceptions. More than once, the slaves had already been created but the master had not written its hostname in the file, leading to IO−exceptions.

When using the SMP machine to test our multi−threaded applications, we encountered a problem with the Java registry. It seems that whenever another person starts a registry process on the SMP, our multi−threaded applications (that make use of Java's RMI system as well, such as Checkers) must use the same registry. Unfortunately, this resulted in 'Stub not found' error messages, apparently because the other person’s registry does not have access to our application’s directory that contains the appropriate Stub classes.

JDK and Manta system bugs

Another problem was the creation of slave objects when using Java RMI. Java’s RMI system seems troublesome when the programmer uses many processors for parallel execution. More than once, we got (runtime) IOExceptions, SIGSEGVs (segmentation violations), broken pipes etc. which must be caused by the Java runtime system, since the programmer theoretically cannot produce these kinds of errors in Java. Starting the Swing Checkers client application was sometimes causing similar problems, even before a connection with the server was made. In certain cases the stub and skeleton compiler (RMIC) crashed on a segmentation violation.

With water the JIT compiler gave incorrect results. This is caused by the fact that many parameters have to be passed on to worker objects. Somehow the JIT passes on wrong values. According to the Java language specification there is no limit on the number of parameters that can be passed on.

The multi−threaded version of water makes use of synchronized blocks to update certain shared variables. These synchronized blocks do not work properly. When using four or more threads certain updates seem to get lost.

The biggest problem we had with Manta is the fact that it is a project which is in its early stages. Numerous bugs were discovered.

One of the problems of Manta is that it doesn’t support kernel−space threads, caused by its underlying thread package, Panda. A multi−threaded application therefore would not gain advantage from a SMP machine, because only user−space threads are created which run on just one CPU. More information on this subject can be found at the end of section 4.

With the Manta version of Water the floating point optimization produces different results from the version without floating point optimization. This is caused by the 16th decimal which differs in both versions. Since Water is very accurate with decimals this 16th decimal causes the difference between the Manta version with floating point optimization and the one without floating point optimization to be quite high (with 1728 molecules there is a difference of one, a deviation of approximately 30%).

10 Conclusion

This thesis describes the usability of the Java system and Manta system by means of four applications (FFT, Radix sort, Water and Checkers) which are implemented using four different strategies (sequential, multi−threaded, Java RMI and Manta RMI). Of each variant, the performance is discussed. The ease of use to the programmer of both systems is discussed as well.

Besides the several performance problems encountered with Java, the use of Java for parallel application development is becoming more promising. Although the Sun Java version does not perform well, the Manta version produces better and acceptable results. With the Water application Manta achieved almost linear speedup. Still the scalability of Java with these applications is worse than the scalability of C or Orca with the same applications. But with Manta the difference has lessened. The Manta versions are a factor 2.5 slower on the average. These differences in performance are mostly due to the lack of asynchronous communication and broadcasting which has a great impact on the total latency. The poor floating point performance of Java, imposed by the strict adherence to the Java floating point standard, also reduces the overall performance of the Java versions. Other language features that have a great impact on the performance of Java are the lack of pointer arithmetics and the fact that object creation is very expensive (see section 9).

Parallel programming in Java with RMI is feasible at least to coarse−grained applications. However, the performance depends heavily on the structure of the communication. Using arrays or objects as parameter of RMIs can exert a large influence on the performance, as the transmission of large arrays and objects can effect the latency to a large extent because of network load and serialization overhead. The lack of asynchronous communication makes it worse. Synchronizing garbage collection by explicitly calling the garbage collector can improve the performance. Message combining may improve performance as well.

Overall it can be stated that on these four applications, Sun Java using Fast Ethernet performs worse than Manta using Myrinet. In all applications Manta is faster and gives better results. Manta reaches higher speedups and better efficiencies, while Sun Java often crashes with multiple CPUs.

The use of the *remote* keyword in Manta simplifies the work for a programmer. It lessens the amount of programming code and it makes porting a multithreaded program to RMI easier. Also the programmer does not have to know about registries, security managers, Java policies, file handling (opening a file, reading data from a file, closing the file) etc.

Although Java lacks certain features like broadcasting and asynchronous communication, parallel application development with Java RMI seems promising. With Sun Java the performance is poor while with Manta the performance is acceptable. Taking the performance into account we conclude that Java with Manta is more promising than Sun Java.

11 Acknowledgments

We would like to thank Henri Bal, Thilo Kielmann and Aske Plaat for their guidance and patience, Jason Maassen, Rob van Nieuwpoort, and Ronald Veldema for their support and their work on Manta, as well as for the coffee breaks, and John Romein for his explanations of some algorithms and hints. We would also like to thank Ceriel Jacobs for his contribution to the Manta project and Rice University for giving us their implementation of Water.

12 References

[1] C.F. Joerg and B.C. Kuszmaul. *Massively Parallel Chess.* MIT Laboratory for Computer Science, Cambridge, Massachusetts. DIMACS'94 Challenge, October 1994. http://thor.cs.yale.edu/~bradley/

[2] R. Winder, G. Roberts. *Developing Java Software*, 1998. Online at: http://www.dcs.kcl.ac.uk/DevJavaSoft/

[3] A. Wollrath, J. Waldo. *Trail: RMI*. Online at: http://www.java.sun.com/docs/books/tutorial/rmi/index.html

[4] J. Maassen, R. van Nieuwpoort. *Fast Parallel Java*. Vrije Universiteit Amsterdam, August 1998.

[5] A. Plaat, H.E. Bal, R.F.H. Hofman. *Sensitivity of parallel applications to large differences in bandwidth and latency in two−layer interconnects.* Vrije Universiteit Amsterdam. Online at: http://www.cs.vu.nl/albatross/

[6] V. Kumar, A. Grama, A. Gupta G. Karypis. *Introduction to parallel computing: Design and analysis of algorithms*. Benjamin Cummings, November 1993.

[7] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta. *The SPLASH−2 programs: Characterization and methodological considerations*. In proceedings of the 22nd International Symposium on Computer Architecture, pages 24−36, June 1995.

[8] F. Hoffman. *An introduction to Fourier theory*. Online at: http://aurora.phys.utk.edu/~forrest/papers/fourier/index.html

[9] M.Philippsen and M.Zenger. Javaparty−transparant remote objects in java. *Concurrency: Practice and Experience*, Vol. 9(No.11):1125−1242, 1997. http://wwwipd.ira.uka.de/~phlipp/party.ps.gz.

[10] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. *An Efficient Implementation of Java's Remote Method Invocation.* Proc. PPoPP'99, pp. 173−182, Atlanta, GA, May 1999.,

[11] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith, M. Zagha. *A comparison of sorting algorithms for the Connection Machine CM−2*. In Proc. Of the 1991 Symposium on Parallel Algorithms and Architectures, Hilton Head, SC, July 1991.

[12] Bal, H.E., Bhoedjang, R., Hofman, R., Jacobs, C., Langendoen, K., Ruhl, T., and Kaashoek, M.F.,: "*Performance Evaluation of the Orca Shared Object System*", ACM Transactions on Computer Systems, Vol. 16, No. 1, Febr. 1998. Copyright 1998 by ACM, Inc.

[13] J.Romein. *Water−an N−body Simulation Program on a Distributed Architecture*, 1994. Online at: http://www.cs.vu.nl/~john/

[14] John W. Romein, Henri E. Bal: *Parallel N−Body Simulation on a Large−Scale Homogeneous Distributed System*, EuroPar'95, August 29−31, 1995, Stockholm, Sweden.

[15] Jaswinder Pal Singh, Wolf−Dietrich Weber, and Anoop Gupta. SPLASH: *Stanford Parallel Applications for Shared−Memory*. In Computer Architecture News, vol. 20, no. 1, pages 5−44. March 1992. http://www−flash.stanford.edu/apps/

[16] Robert Sedgewick, *Algorithms in C++*, [3rd rev. ed.], 8th print. Reading, Mass. Addison−Wesley 1995.

[17] R.A.F. Bhoedjang. *A Comparison of Radix Sort on Four Distributed Memory Systems,* February 3, 1998.

[18] A.C. Dusseau, D.E. Culler, K.E. Schauser, R.P. Martin. *Fast Parallel Sorting under LogP: Experience with the CM−5*, IEEE Transactions on Parallel and Distributed System, August 1996.

[19] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta. *The SPLASH Programs: Characterisation and Methodological Considerations.* In the 22nd Annual International Symposium on Computer Architecture, Santa Margherita, Italy, June 1995.
http://www−flash.stanford.edu/apps/

[20] *The Cilk Project*. Massachusetts Institute of Technology.
http://supertech.lcs.mit.edu/cilk/

[21] Jonathan Schaeffer, R. Lake. *Solving the Game of Checkers.* Edmonton, Canada.

[22] Jonathan Schaeffer, R. Lake, P. Lu, M. Bryant. *Chinook: The World Man−Machine Checkers Champion.* AI Magazine 1996
http://www.cs.ualberta.ca/~chinook/

[23] John W. Romein, Henri E. Bal, Aske Plaat. *Data Distribution Techniques for Shared Distributed Transposition Tables in Heuristic Search.* Faculty of Sciences, Department of Computer Science. Amsterdam, The Netherlands.

[24] John W. Romein, Henri E. Bal, Dick Grune. *An Application Domain Specific Language for Describing Board Games.* International Conference on Parallel and Distributed Processing Techniques and Applications, June 30 − July 3, 1997, Las Vegas, NV.

[25] R. van Nieuwpoort, J. Maassen, H.E. Bal, T. Kielmann, R. Veldema. *Wide−area parallel computing in Java.* Faculty of Sciences, Department of Computer Science. Amsterdam, The Netherlands, 1999.
http://www.cs.vu.nl/~kielmann/jg99.ps.gz

[26] R. Feldmann and P. Mysliwietz and B. Monien. *Studying Overheads in Massively Parallel Min/Max−Tree Evaluation.* In Proceedings of SPAA '94 pp. 94−103, 1994. Department of Mathematics and Computer Science, University of Paderborn, Germany.

[27] A. Plaat, J. Schaeffer, W. Pijls, A. de Bruin. *Best−First and Depth−First Minimax Search in practice.* In Proceedings of the International Joint conference of AI (IJCAI−95), August '95 vol.1, pp. 273−279. Erasmus University, Rotterdam, The Netherlands and the University of Alberta, Edmonton, Canada.

[28] J. Schaeffer, A. Plaat. *New advances in Alpha−Beta Searching.* Erasmus University, Rotterdam, The Netherlands and the University of Alberta, Edmonton, Canada.

[29] A. Plaat, J. Schaeffer, W. Pijls, A. de Bruin. *Best−First Fixed−Depth Minimax Algorithms.* Artificial Intelligence(1−2) pp. 255−293, Nov '96. Erasmus University, Rotterdam, The Netherlands and the University of Alberta, Edmonton, Canada.

[30] J. Schaeffer. *The History Heuristic and Alpha−Beta Search Enhancements in Practice.*In IEEE Transactions on Pattern Analysis and Machine Intelligence '89 pp. 1203−1212.

[31] A.L. Zobrist. *A new hashing method with application for game playing.* Technical report 88, Computer Science Department, University of Wisconsin, Madison, 1970. Reprinted in: ICCA Journal, 13(2):69−73, 1990.

13 Appendix A − FFT results

All timings are in milliseconds.

Sequential Sun Java 1.2 (JIT)	Pure FFT	Total exec. time	Communication
M=14	306	352	0
M=16	1240	1462	0
M=18	5367	6271	0
Sequential Manta			
M=14	401	433	0
M=16	1796	1953	0
M=18	7879	8651	0
Sequential Orca			
M=14	165	312	146
M=16	740	1559	708
M=18	3195	6563	2871

Sequential SMP	Pure FFT	Total exec. time	Communication	Barrier	Speedup
M=14	140	162	0	0	
M=16	612	726	0	0	
M=18	2657	3150	0	0	
Multi−threaded SMP Single Thread					
M=14	165	247	70	0	
M=16	722	1132	400	0	
M=18	3170	4868	1526	0	
Multi−threaded SMP Dual thread					
M=14	88	139	38	8	1.17
M=16	377	611	206	24	1.19
M=18	1600	2725	905	89	1.16

Unoptimized Manta RMI ***1 CPU***	Pure FFT	Total exec. time	Communication	Barrier	Speedup	Efficiency
M=14	460	795	338	0	0.54	0.54
M=16	1963	3297	1320	0	0.59	0.59
M=18	8747	15640	6834	0	0.55	0.55
2 CPUs						
M=14	221	356	136	12	1.22	0.61
M=16	992	1673	671	62	1.17	0.58
M=18	4509	7350	2733	390	1.18	0.59
4 CPUs						
M=14	111	195	77	11	2.22	0.56
M=16	539	840	299	42	2.33	0.58
M=18	2194	3650	1306	150	2.37	0.59
8 CPUs						
M=14	56	121	60	13	3.58	0.45
M=16	253	433	168	33	4.51	0.56
M=18	1092	1865	698	93	4.64	0.58
16 CPUs						
M=14	30	104	71	22	4.16	0.26
M=16	192	327	139	88	5.97	0.37
M=18	823	1173	356	293	7.38	0.46
32 CPUs						
M=14	21	127	386	39	3.41	0.11
M=16	68	249	180	63	7.84	0.25
M=18	308	619	303	119	13.98	0.44

Optimized Manta RMI ***1 CPU***	Pure FFT	Total exec. time	Communication	Barrier	Speedup	Efficiency
M=14	438	765	329	0	0.56	0.56
M=16	1930	3260	1312	0	0.59	0.59
M=18	8665	14606	2875	0	0.59	0.59
2 CPUs						
M=14	229	264	130	28	1.64	0.82
M=16	982	1639	606	40	1.19	0.60
M=18	4368	7166	2545	161	1.21	0.60
4 CPUs						
M=14	108	183	68	7	2.37	0.59
M=16	518	836	289	58	2.34	0.58
M=18	2305	3673	1238	231	2.36	0.59
8 CPUs						
M=14	58	108	44	9	4.01	0.50
M=16	248	412	148	16	4.74	0.59
M=18	1106	1827	635	83	4.74	0.59
16 CPUs						
M=14	31	86	48	11	5.03	0.31
M=16	123	221	88	12	4.84	0.55
M=18	550	904	302	35	9.57	0.60
32 CPUs						
M=14	32	151	116	38	2.87	0.09
M=16	68	172	96	15	11.35	0.35
M=18	290	496	182	25	17.44	0.55

Sun RMI ***2 CPU*s**	Pure FFT	Total exec. time	Communication	Barrier	Speedup	Efficiency
M=14	227	2841	2340	491	0.12	0.06
M=16	978	10346	10352	1872	0.14	0.07
M=18	4350	40621	42175	1425	0.15	0.08
4 CPUs						
M=14	111	2721	1901	1122	0.13	0.03
M=16	454	8335	6919	1329	0.18	0.04
M=18	2212	30577	28413	4078	0.21	0.05
8 CPUs						
M=14	62	3885	1615	2542	0.09	0.01
M=16	235	9091	6358	3870	0.16	0.02
M=18	1785	29031	25036	6286	0.22	0.03
16 CPUs						
M=14	40	9171	4045	6504	0.04	0.00
M=16	136	14064	8668	7380	0.10	0.01
M=18	802	36217	30402	9980	0.17	0.01
32 CPUs						
M=14	406	32582	–	–	0.01	0.00
M=16	–	–	–	–	0,00	
M=18	–	–	–	–	0,00	

Orca ***1 CPU***	Pure FFT	Total exec. time	Communication	Speedup	Efficiency
M=14	165	312	146		
M=16	740	1559	708		
M=18	3195	6563	2871		
2 CPUs					
M=14	83	159	52	1.96	0.98
M=16	366	933	386	1.67	0.83
M=18	1586	4264	1898	1.54	0.77
4 CPUs					
M=14	39	88	30	3.55	0.89
M=16	183	364	132	4.27	1.07
M=18	791	1953	852	3.36	0.84
8 CPUs					
M=14	21	55	19	5.67	0.71
M=16	91	188	67	8.27	1.03
M=18	398	815	302	8.05	1.01
16 CPUs					
M=14	10	44	16	7.09	0.44
M=16	45	117	47	13.29	0.83
M=18	198	415	160	15.81	0.99
32 CPUs					
M=14	6	55	22	5.67	0.18
M=16	24	90	36	17.28	0.54
M=18	99	242	92	27.12	0.85

14 Appendix B – Radixsort results

RBS with Sun JDK

Multithreaded

#threads/input size	100000	500000	1000000
2	477	2459	Nav
Sequential/Sun RMI			
RBS execution time			
CPUs/input size	100000	500000	1000000
1	1954	6620	Nav
2	1040	5940	2341
4	1223	4231	3211
8	1262	2869	4160
16	Nav	Nav	Nav
32	Nav	Nav	Nav
RBS Speedups			
CPUs/input size	100000	500000	1000000
2	1.88	1.4	Nav
4	1.6	1.56	Nav
8	1.55	2.31	Nav
16	Nav	Nav	Nav
32	Nav	Nav	Nav
RBS efficiency			
CPUs/input size	100000	500000	1000000
2	0.94	0.56	Nav
4	0.4	0.39	Nav
8	0.19	0.29	Nav
16	Nav	Nav	Nav
32	Nav	Nav	Nav

Breakdown			
Radixsort			
CPUs/input size	100000	500000	1000000
2	810	4917	942
4	503	2717	472
8	212	1099	263
16	Nav	Nav	Nav
32	Nav	Nav	Nav
Merging			
CPUs/input size	100000	500000	1000000
2	161	523	827
4	179	738	1213
8	199	888	1665
16	Nav	Nav	Nav
32	Nav	Nav	Nav

Communication			
CPUs/input size	100000	500000	1000000
2	69	500	572
4	541	776	1526
8	851	882	2232
16	Nav	Nav	Nav
32	Nav	Nav	Nav

RBS with Manta			
Sequential/Manta RMI			
RBS execution time			
CPUs/input size	100000	500000	1000000
1	1091	5854	11619
2	554	3112	5792
4	324	1785	3591
8	216	1087	2161
16	173	794	1528
32	173	669	1266
RBS Speedups			
CPUs/input size	100000	500000	1000000
2	1.97	1.88	2
4	3.37	3.28	3.24
8	5.05	5.39	5.38
16	6.31	7.37	7.6
32	6.31	8.75	9.18
RBS Efficiency			
CPUs/input size	100000	500000	1000000
2	0.98	0.94	1
4	0.84	0.82	0.81
8	0.63	0.67	0.67
16	0.39	0.46	0.48
32	0.2	0.27	0.29
Breakdown			
Radixsort			
CPUs/input size	100000	500000	1000000
2	506	2879	5326
4	252	1410	2895
8	125	620	1247
16	62	312	633
32	30	157	311
Merging			
CPUs/input size	100000	500000	1000000
2	39	198	398
4	59	322	597
8	70	347	795
16	73	400	744
32	78	401	771
Communication			
CPUs/input size	100000	500000	1000000
2	9	35	68
4	13	53	99
8	21	120	119
16	38	82	151
32	65	111	184

RCS with Manta				
Multithreaded				
#threads/input size	100000	500000	1000000	3000000
2	65	291	535	1646
Multithreaded speedup				
#threads/input size	100000	500000	1000000	3000000
2	2.4	2.97	3.06	2.98

Sequential/Manta RMI				
RCS Execution time				
CPUs/input size	100000	500000	1000000	3000000
1	154	782	1573	4750
2	122	587	1158	3458
4	86	319	605	1764
8	125	234	384	959
16	267	329	402	683
32	649	696	737	894
RCS Speedup				
CPUs/input size	100000	500000	1000000	3000000
2	1.26	1.33	1.36	1.37
4	1.79	2.45	2.6	2.69
8	1.23	3.34	4.1	4.95
16	0.58	2.38	3.91	6.95
32	0.24	1.12	2.13	5.31
RCS Efficiency				
CPUs/input size	100000	500000	1000000	3000000
2	0.63	0.67	0.68	0.69
4	0.45	0.61	0.65	0.67
8	0.15	0.42	0.51	0.62
16	0.04	0.15	0.24	0.43
32	0.01	0.04	0.07	0.17
Breakdown				
Histogramtime				
CPUs/input size	100000	500000	1000000	3000000
2	25	125	247	740
4	12	63	125	370
8	7	32	63	186
16	3	16	34	94
32	3	9	17	51
Sorttime				
CPUs/input size	100000	500000	1000000	3000000
2	36	187	374	1126
4	19	94	188	561
8	9	48	95	281
16	6	24	47	141
32	3	13	25	72
Mergetime				
CPUs/input size	100000	500000	1000000	3000000
2	7	8	9	10
4	29	31	30	38
8	94	89	92	90
16	250	265	247	249
32	636	683	658	669
Permutetime				
CPUs/input size	100000	500000	1000000	3000000
2	54	266	523	1572
4	33	133	263	792
8	31	79	141	394
16	71	94	125	248
32	203	214	228	287

15 Appendix C – Water results

Water with Sun JDK, without JIT
1728 molecules, 2 iterations
2nd iteration measured

CPUs / Sun RMI	Execution time incl. Poteng	Speedup	Efficiency
1	1225926 (with JIT 305312)		
2	632509	1.94	1.94
4	377560	3.25	3.25
8	Nav	Nav	Nav
16	Nav	Nav	Nav
32	Nav	Nav	Nav

Water with Manta

1728 molecules, 2 iterations

2nd iteration measured

CPUs/Manta RMI	Execution time excl. Poteng	Execution time incl. Poteng	Speedup	Efficiency
1		241486		
2	90059	127159	1.9	0.95
4	43681	62215	3.88	0.97
8	20627	30391	7.95	0.99
16	10281	15305	15.78	0.99
32	5454	7992	30.22	0.94

Breakdown			
CPUs/Manta RMI	Fptime	Communication time	Time intraf + predcorr + misc
1	241216	0	270
2	126525	132	502
4	61536	430	249
8	29397	185	809
16	14556	290	459
32	7554	308	130

Water : Orca Vs. Manta Vs. Java

With 1728 molecules

Execution time			
CPUs / OS	Orca	Manta	Java
1	71553	241486	1225926
2	38528	127159	632509
4	18757	62215	377560
8	9308	30391	Nav
16	4674	15305	Nav
32	2359	7992	Nav
Speedups			
CPUs / OS	Orca	Manta	Java
2	1.86	1.9	1.94
4	3.81	3.88	3.25
8	7.69	7.95	Nav
16	15.31	15.78	Nav
32	30.33	30.22	Nav

16 Appendix D – Checkers result

All timings are in milliseconds.

	Nodes	Evaluation	Move Generation	Make Move	Get Node	Put Node	Idle	Misc.	Total	Speedup	Efficiency
Seq. Manta	*1116417*	7066	57588	27895	6326	33830	0	24573	157276		
Seq. Java	*1116417*	8374	75474	65051	10125	11982	0	32673	203678		
Seq. Java SMP	*1116417*	5020	37337	39584	6548	8648	0	21443	118580		
Single Thread SMP	*1116417*	4890	39468	38418	6265	7110	1450	19966	117567	1.01	1.01
Dual Thread SMP	*1544294*	3793	73543	37635	5265	7681	11379	18563	157857	0.75	0.38

FD Sun Part. TT 8 CPUs	*Nodes*	Evaluation	Move Gen.	Make Move	Get Node	Put Node	Get Hndl	Put Hndl	Message	Idle	Misc.	Total	Speedup
M:32 LT:2	*5862405*	4913	23709	12733	398018	1183	1719	369	22373	62528	126286	653830	0.24
16 CPUs													
M:32 LT:0	*6451732*	2720	12357	7023	285953	410	998	216	19692	104290	91945	525604	0.30
32 CPUs													
M:8 LT:2	*6881785*	1456	6583	3694	179568	109	555	131	18280	146597	84075	441048	0.36

RD Manta Part. TT 2 CPUs	*Nodes*	Evaluation	Move Gen..	Make Move	Get Node	Put Node	Get Hndl	Put Hndl	Message	Idle	Misc.	Total	Speedup
M:32 LT:2	*1737395*	5619	32747	20607	30415	5306	239	554	19886	18094	13078	146544	1.07
4 CPUs													
M:32 LT:2	*1995557*	3351	16463	9822	17148	943	181	476	12643	26934	13618	101579	1.55
8 CPUs													
M:32 LT:2	*3000992*	2510	11795	6825	11285	255	139	393	8790	73022	11537	126551	1.24
16 CPUs													
M:128 LT:2	*3532517*	1467	6532	3615	5360	81	82	216	2467	82410	18498	120728	1.30
32 CPUs													
M:32 LT:2	*3663289*	747	3185	1873	1648	22	42	123	1628	87502	12952	109722	1.43

FD YBWC Manta Part. TT 2 CPUs	*Nodes*	Evaluation	Move Gen.	Make Move	Get Node	Put Node	Get Hndl	Put Hndl	Message	Idle	Misc.	Total	Speedup
M:128 LT:2	*6983833*	26691	139690	84587	309218	41140	690	153	10033	38586	69032	719818	0.22
4 CPUs													
M:8 LT:2	*7377753*	12507	58769	35281	96756	3634	480	144	11770	20341	44192	283873	0.55
8 CPUs													
M:8 LT:2	*7777522*	6600	28867	15551	43587	1173	270	81	5396	27972	30585	160081	0.98
16 CPUs													
M:8 LT:2	*8377580*	3567	14213	7338	16215	1186	153	47	2546	34576	16832	96673	1.63
32 CPUs													
M:8 LT:2	*9211210*	1975	7658	3743	5795	239	81	25	1094	47824	13699	82134	1.91

FD Manta Local TT	*Nodes*	Evaluation	Move Gen.	Make Move	Get Node	Put Node	Alpha Update	Idle	Misc.	Total	Speedup
8 CPUs	*13486502*	11499	97023	62987	10419	89973	33	256995	36520	565449	0.28
16 CPUs	*16239678*	6269	39362	28259	5617	25846	153	225762	11738	343006	0.46
32 CPUs	*18894213*	3615	20419	11739	3231	8815	138	131621	21844	201422	0.78

FD Manta Part. TT 2 CPUs	*Nodes*	Evaluation	Move Gen.	Make Move	Get Node	Put Node	Get Hndl	Put Hndl	Message	Idle	Misc.	Total	Speedup
M:8 LT:2	*6270120*	27248	138575	71390	259354	44703	590	167	31994	21378	63471	658868	0.24
M:128 LT:2	*6623963*	23583	127889	90360	283134	38253	657	138	6488	28909	60549	659959	0.24
4 CPUs													
M:8 LT:0	*5602169*	9270	49677	25724	186687	2839	2338	626	29221	38911	25310	370603	0.42
M:8 LT:2	*6794173*	11252	53458	31608	84980	2869	436	133	9934	20122	31851	246643	0.64
M:8 LT:4	*9474392*	15013	72554	41907	74366	3990	83	31	9103	32517	44742	294306	0.53
M:32 LT:0	*5766238*	9775	49631	27700	192979	3456	2335	500	12686	35713	31712	366486	0.43
M:32 LT:2	*6939740*	11657	53941	31832	87148	2484	445	105	3466	19913	35647	246638	0.64
M:32 LT:4	*10007478*	16068	75701	44390	84894	5255	95	24	2458	33178	43911	305972	0.51
M:128 LT:0	*5980262*	10118	49906	30343	204343	3009	2398	503	8369	46773	29237	384998	0.41
M:128 LT:2	*7332458*	12069	55818	34351	95279	3119	460	106	2366	20791	38752	263110	0.60
M:128 LT:4	*10623427*	16946	80222	47289	97891	5224	87	21	1588	34475	43437	327178	0.48
8 CPUs													
M:8 LT:0	*5673623*	4789	22357	12976	90089	667	1373	377	12669	37897	18493	201687	0.78
M:8 LT:2	*7576178*	6386	27399	15208	42138	1151	263	80	5491	24081	31099	153295	1.03
M:8 LT:4	*11645689*	9541	40121	22438	25783	1967	54	19	2506	43458	32452	178340	0.88
M:32 LT:0	*6087517*	5213	23532	14100	98709	717	1416	302	5624	36689	29217	215519	0.73
M:32 LT:2	*7641239*	6425	27131	15460	43698	1127	256	64	1726	25506	22536	143930	1.09
M:32 LT:4	*12835969*	10454	43514	25251	28694	2224	56	13	743	41301	33393	185644	0.85
M:128 LT:0	*6461766*	5492	25044	15030	103763	684	1529	329	3903	32491	42050	230315	0.68
M:128 LT:2	*8808132*	7388	31751	17558	50714	1148	299	61	957	31983	27786	169645	0.93
M:128 LT:4	*15324370*	12429	52272	30667	39424	2429	70	14	361	50307	35843	223815	0.70
16 CPUs													
M:8 LT:0	*5996447*	2521	11102	6380	48561	249	757	213	7270	39346	22865	139265	1.13
M:8 LT:2	*8113324*	3425	13513	7035	15549	1182	151	45	2561	30171	25419	99051	1.59
M:8 LT:4	*14599987*	6069	23075	11511	8393	3994	33	12	845	53621	34566	142118	1.11
M:32 LT:0	*6476488*	2761	12132	6754	53893	259	795	174	3051	35668	17945	133432	1.18
M:32 LT:2	*9213128*	3889	15308	8081	17807	1256	163	35	963	34374	25764	107640	1.46
M:32 LT:4	*17641292*	7278	28076	14354	10380	4242	40	8	332	66978	36305	167994	0.94
M:128 LT:0	*7440380*	3173	13811	7726	61459	238	903	196	2336	39977	30276	160095	0.98
M:128 LT:2	*11771428*	4960	19774	10596	24365	1183	198	39	673	40851	34746	137384	1.14
M:128 LT:4	*23221214*	9534	37724	20134	14758	3974	48	8	175	88684	47482	222521	0.71
32 CPUs													
M:8 LT:0	*6629109*	1397	5726	3118	22386	95	440	124	4364	42034	22050	101734	1.55
M:8 LT:2	*9225551*	1973	7597	3732	6089	450	82	25	1156	31127	18647	70878	2.22
M:8 LT:4	*18389770*	3850	14544	7042	4036	2004	19	6	377	63271	42537	137686	1.14
M:32 LT:0	*7392783*	1570	6409	3475	25894	114	455	98	1850	43516	30304	113684	1.38
M:32 LT:2	*11825732*	2520	9710	4751	7740	561	98	21	627	42913	25246	94188	1.67
M:32 LT:4	*25673907*	5314	20350	9852	5751	2749	28	5	212	87072	57534	188868	0.83
M:128 LT:0	*9288760*	1983	8228	4489	33905	110	552	112	1488	51075	32083	134024	1.17
M:128 LT:2	*16944618*	3591	13882	7072	12167	666	136	26	550	63459	42009	143557	1.10
M:128 LT:4	*34957535*	7226	27936	13602	8663	3532	42	4	110	117806	59229	238150	0.66

FD Manta Repl. TT 4 CPUs	*Nodes*	Evaluation	Move Gen.	Make Move	Get Node	Put Node	Alpha Update	Broadcast	Idle	Misc.	Total	Speedup
M:128 LT:4	*6997315*	11233	83247	60905	12037	67558	18	192430	83586	43683	554683	0.28
8 CPUs												
M:128 LT:4	*7947059*	6241	38481	22028	5890	21581	34	84630	80222	20517	279622	0.56
16 CPUs												
M:128 LT:4	*9136056*	3569	18961	11879	3256	7254	67	36007	71750	22198	174940	0.90
32 CPUs												
M:128 LT:4	*10404242*	2087	9978	5897	1839	2702	131	17893	68947	10513	119988	1.31