EPCC-SS2001-01

Portable Lattice-Boltzmann in Java

Rubén Jesús García Hernández

Abstract

Java performance has been increasing steadily for the past few years, and can now compete with that of C and Fortran. This paper describes the porting of Ludwig, which is a versatile code for the simulation of Lattice-Boltzmann models in 3D on cubic lattices, to Java in order to increase its portability. The performance difference between the original C code and the Java port is analysed. The Java port has been benchmarked on different Java Virtual Machines, and the performance differences among the different JVMs for the main routines of the code is shown. The difficulties encountered in the porting operation are discussed. Two parallel versions of the program have been developed, using JOMP (Java OpenMP) and mpiJava (Message Passing Interface for Java), respectively. However, the mpiJava version is still under development, so no benchmarking results are yet available. The problems encountered in the JOMP port are discussed, and an explanation of the different algorithms programmed to overcome a speedup limitation of the original C code is provided. The problems found in the mpiJava port are addressed, and will be dealt with in a latter project.

Contents

1 Introduction

1.1 What is the Boltzmann equation?

The Boltzmann equation is any equation of the form

$$\frac{\partial}{\partial t}\Phi(x,t) = -v \cdot \Phi(x,t) + v \cdot \int_{-\infty}^{\infty} K(x,s) \cdot \Phi(s,t) ds \qquad (1)$$

This equation first appeared in the theory of particle transport in the end of the 19th century. The Φ the equation refers to is the particle density in the phase-space.

It is now also used in many fields of physics, e.g. Nuclear physics. [3]

1.2 What is Lattice-Boltzmann?

Lattice-Boltzmann is a method for solving the Boltzmann Equation (in the original form). The simulation code will eventually be able to handle multicomponent fluids, amphiphilic systems, and flow in porous media as well as colloidal particles and polymers. Future development might include detergency, binary fluids in porous media, mesophase formation in amphiphiles, colloidal suspensions, and liquid crystal flows. So far, we have restricted our attention to simple binary fluids.

The first step in doing so is discretizing the equation. The discretized equation has the form

$$\Phi(x,n+1) = \int_{-\infty}^{\infty} K(x,s) \cdot \Phi(s,n) \cdot ds \qquad (2)$$

Both the densities and the velocities have been discretized as well, so we now have a set of velocities, shown in Figure 1.

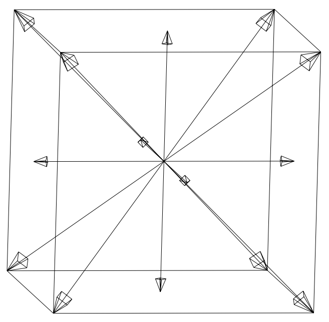

Figure 1 : D3Q15 Model: 15 velocities, one with speed zero (a rest particle), six with $speed^2 = 1$ (to nearest neighbours), and eight with $speed^2 = 3$ (to next next nearest neighbours).

We solve for each velocity:

$$f_i(\vec{r}+\vec{c}_i, t+1) - f_i(\vec{r},t) = -\omega(f_i^{eq}(\vec{r},t) - f_i(\vec{r},t)) \qquad (3)$$

Where $f_i(\vec{r},t)$ is the density of particles with velocity $\vec{c}_i$ resident at node $\vec{r}_i$ at time t. This particle density will propagate to the site $\vec{r}+\vec{c}_i$. $\vec{c}_i$ is a lattice vector. The model is characterized

by a finite set of velocities $\{\vec{c}_i\}$, as seen in the graph above (Fig 1). $f_i^{eq}(\vec{r},t)$ is the equilibrium distribution of $f_i(\vec{r},t)$ and characterizes the type of fluid simulated. The right-hand side of Eq.(1) describes a mixing of the different particle densities (collision): the f_i distribution relaxes towards f_i^{eq} at a rate determined by ω, the relaxation parameter. The relaxation parameter is related (through $\eta = \frac{2\omega^{-1}-1}{6}$) to the viscosity η of the fluid. The hydrodynamic quantities, such as the local density, ρ, momentum, $\rho\vec{v}$ and stress, $\mathcal{P}_{\alpha\beta}$ are given as moments of the densities of particles $f_i(\vec{r},t)$ as $\sum_i f_i = \rho$, $\sum_i f_i\vec{c}_i = \rho\vec{v}$, and $\sum_i f_i c_{i\alpha}c_{i\beta} = \mathcal{P}_{\alpha\beta}$
Equation (1) shows that Lattice-Bolzmann can be divided into two main routines:

- Propagation (Left-hand side of the equation): Nested loops performing memory-to-memory copies.
- Collision (Right-hand side of the equation): Strong degree of spatial loclity and relies on basic add and multiply operations.

The LB model is extended to describe a binary mixture of fluids, of tunable miscibility, by adding a second distribution function, g_i. This function describes the order parameter field, ϕ.

1.3 What is Ludwig?

Ludwig is a general purpose parallel Lattice-Boltzmann code, capable of simulating the hydrodynamics of complex fluids in 3D. A C version was developed in 1998.[5]
The Boltzmann equation was divided for computational purpuses into two main routines in the Ludwig code:

- Propagation.
- Collision.

1.3.1 Propagation

The density of the particles in the prisma evolves with time. This routine uses a mesoscale simulation technique to calculate the density of the particles in the next instant of time.

1.3.2 Collision

Once the propagation has ended, we have particles from neighbouring sites crashing in every site. This stage calculates the movement of the particles after the crash. This routine has three main parts:

- Get gradients.
- Fix gradients.
- The collision loop itself.

Interested readers should refer to [5] for a detailed explanation.

Here is one of the simulated results using Ludwig:

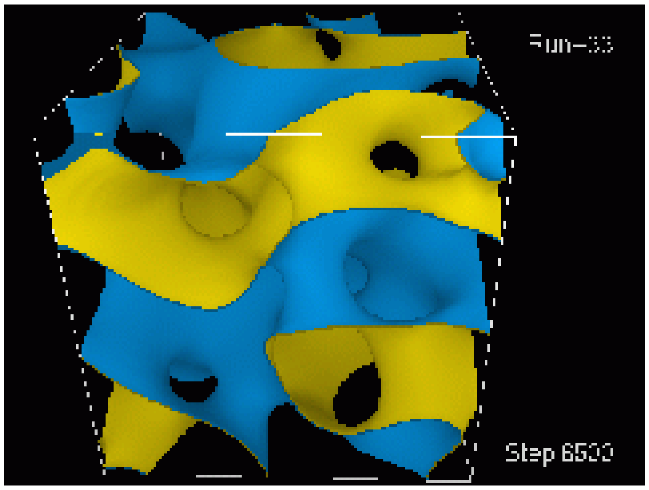

Figure 2 : Evolution of the fluid-fluid interface

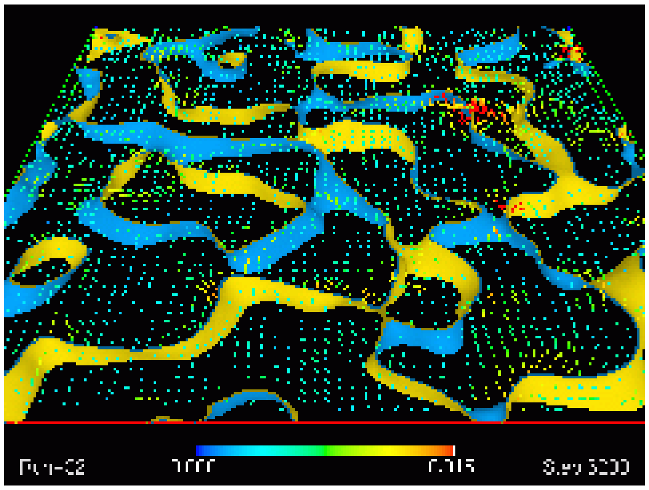

Figure 3 : Time-resolved velocity maps (cropped for clarity to a thin section). See [4] for a detailed explanation.

The code has now been ported to Java (Sequential, Parallel using JOMP and Parallel using mpiJava). This paper explains the issues concerning the Java port.

1.4 Reasons for the Java port.

Due to the fact that most programs last longer than the machines they are executed on (especially in High Performance Computing, where the mean life expectancy of a computer is four years), considerable effort is spent in porting the programs to new architectures. Java was designed so that no porting would ever be necessary (Write Once, Run Anywhere(TM) is the Java logo)

C is a very portable language, but it has no builtin support for high performance parallel programming or graphics. This means most programs have to include non-standard code which must be modified when the hardware or Operating System changes.

Since Java is an Object Oriented language, Software Engineering is easier in this language. The availability of parallel programming support and graphics mean no porting costs. Also threads and RMI (Remote Method Invocation) are built in the language and some advanced parallel programming libraries have been written in pure Java. Nevertheless, parallelism in Java is less mature than in C or Fortran. This can be seen in the fact that the popular standards for parallel computing (MPI and OMP) are still under development in their Java bindings.

Now that Java compilers and interpreters have a similar performance to that of C ones, it would be interesting to know if the performance penalty of Java code is important or not. The aim of this project is being able to compare the performance between the C and Java implementation.

Java is a relatively new language. Its design goals were portability, Object Orientation and security. The portability is obtained by means of creating a Virtual Machine, and having all programs compiled into bytecode for the Java Virtual Machine. Then the bytecode is interpreted on the real machine.

It has built in visualization routines and parallel support, which makes it potentially attractive for Physics and Engineering work.

Since Java is an interpreted language, a few years ago there used to be an order of magnitude difference between the programs written in C and in Java. This made the language unsuitable for High Performance Computing, and few efforts were dedicated to Java in that area. Now, new methods (Just In Time compiling, incremental compiling) allow Java to run at speeds comparable to C.

There is still some work to do on the interpreters, and the differences in performance among VM is noticeable, but the tendency is to have similar performance on well developed VMs. [8]

Furthermore, due to the fact that Java is an interpreted language, the VM has much more information on the program than a C compiler, so it can (in theory) optimize more, therefore getting better performance than the C equivalent.

Some tests show better timing in the Java language for specific programs, and the trend will continue as the compilers and VM mature.

2 Comparison Sequential C versus Sequential Java

2.1 Porting problems for the Sequential Java

It was decided to do a direct translation into Java in order to measure the difference in performance. As LUDWIG now includes a large number of 'specialised' routines (*e.g.* Lees-Edwards, wetting, etc.), it was proposed that only the core routines were to be translated in the first instance. These included:

- main()
- COM_init()
- COM_halo()
- COM_write_site()
- COM_finish()
- RAN_init()
- MODEL_init()
- MODEL_collide()
- MODEL_propagate()

- MODEL_bounce_back()
- MODEL_write_phi()
- MODEL_finish()
- MISC_print_velocity_profile()
- MISC_find_max_velocity()
- MISC_find_max_phi()

as well as any other functions which were called by the above (*e.g.* the BS_* and BC_* routines required to implement the solid sites and wet links linked lists). The code was validated by comparing the output of the Java implementations with that of the native C code prior undertaking any benchmarking.

The starting version of LUDWIG for this project was a serial-only trimmed down version of 4,616 lines (vs. 19,126 for the latest fully-featured release). As the timescale of the project was quite short, it was important to ensure that the method adopted to port the code to Java maximised copy/paste operations wherever practical. The few non-essential features left in the code (such as user-defined ASCII/Binary I/O operations, and possibly linked lists to implement solid sites) were also left aside in the first instance, but were completed afterwards.

Since most of the program consisted of calculations using simple data types (arrays and structures, mostly) and these data types and operations have the same syntax in both C and Java (including numbers, definite (for) and indefinite (while) loops, function calls, vectors and structures) it was decided to keep the C code and modify only the code which differed in syntax. Therefore only I/O (including files and screen), pointer arithmetic, object creation and destruction and timing code had to be modified. The resulting program behaved exactly like the C version except for two semantic differences:

- The Java standard forces the output of floating point results to be accurate enough so that the value can be restored from the text representation.
  This meant that the ASCII result files where different from the C ones, and had to be compared by hand.
- Java arrays of objects behave like C arrays of pointers to that data. This means that assigning from one position of the array into another position effectively links the two positions so that changes in one are visible from the other.
  On the other hand, C copies the data into the other position in the array. This lead to a subtle bug in the Java version which was finally corrected. An example may clarify this point:

```
class A {
int i;
}
```

Java code:

```
void f (A a[2])         a=0,0
{
a[0].i=5;               a=5,0
/* Some code */
a[1]=a[0];              a=5,5
/* Some code */
a[1].i=2;               a=2,2
}
```

C code:

```
void f (A a[2])         a=0,0
{
a[0].i=5;               a=5,0
/* Some code */
a[1]=a[0];              a=5,5
/* Some code */
a[1].i=2;               a=5,2
}
```

The problem is further obscured by the fact that Java arrays of primitive types behave like their C counterparts, while C arrays of pointers (used very often with objects) behave like the Java ones.

2.2 Benchmarking results

As can be seen in Figure 4, the calculation time of the unoptimized version of Java is aproximately 1.5 times that of the C version.

Nevertheless, some hand-made optimization of the Java code shows a behaviour identical to the C code. The optimizations included data transfer optimizations to take care of processor cache and pre-calculation of constant expressions in loops.

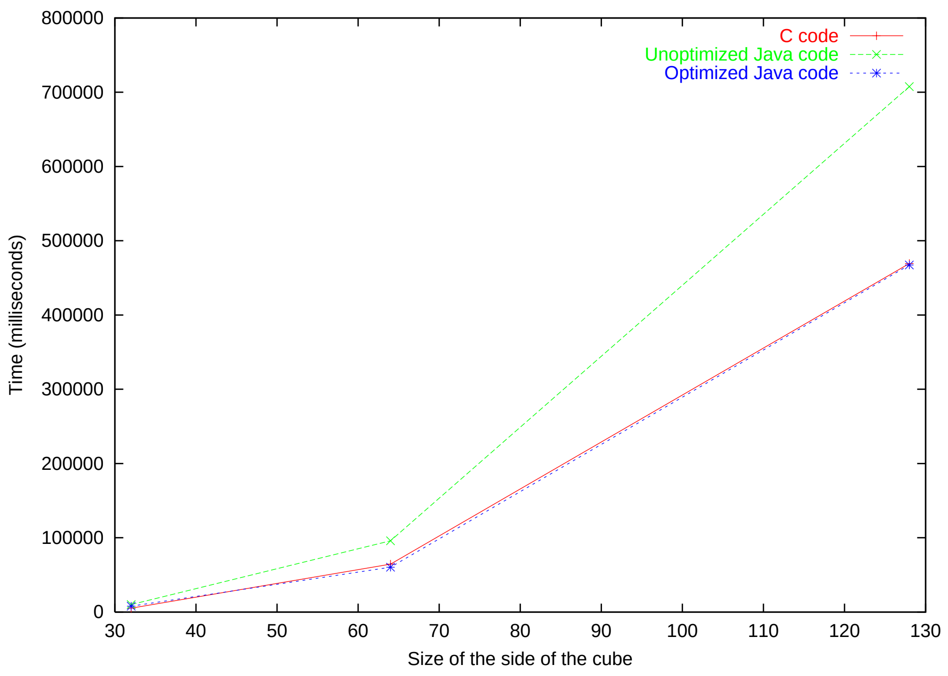

Figure 4 : Graphic comparison C vs Java.

2.2.1 Comparison among different Java Virtual Machines

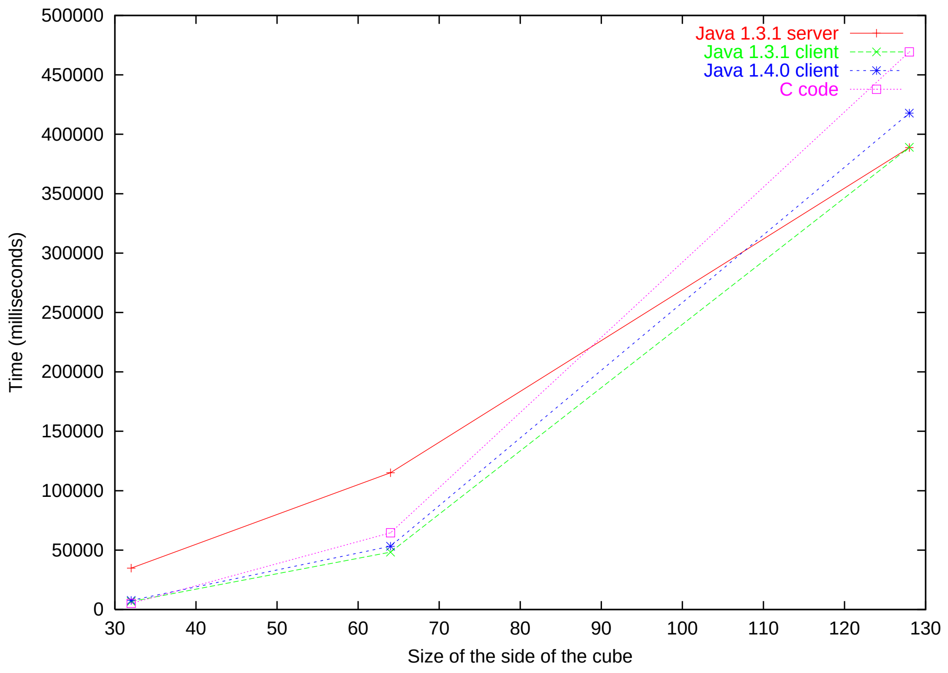

Figure 5 : Graphic of time of different Java Virtual Machines.
The C code has been added for comparison purposes.

As can be seen, some of the Java virtual machines outperform slightly the C code. The server version of the 1.4.0 JVM was tried as well, but due to bugs in the JVM code, it terminated with signal 11 (Segmentation violation) in all executions.

In the following pages a set of graphs comparing the different routines of Ludwig (Propagation, Collision and Halo) is given: Figures 6, 7 and 8. The results show that Java 1.4.0, being a beta version, usually performs worse than 1.3.1. Java 1.3.1 server, having been designed for high memory demands, usually performs worse than the other on small size problems.

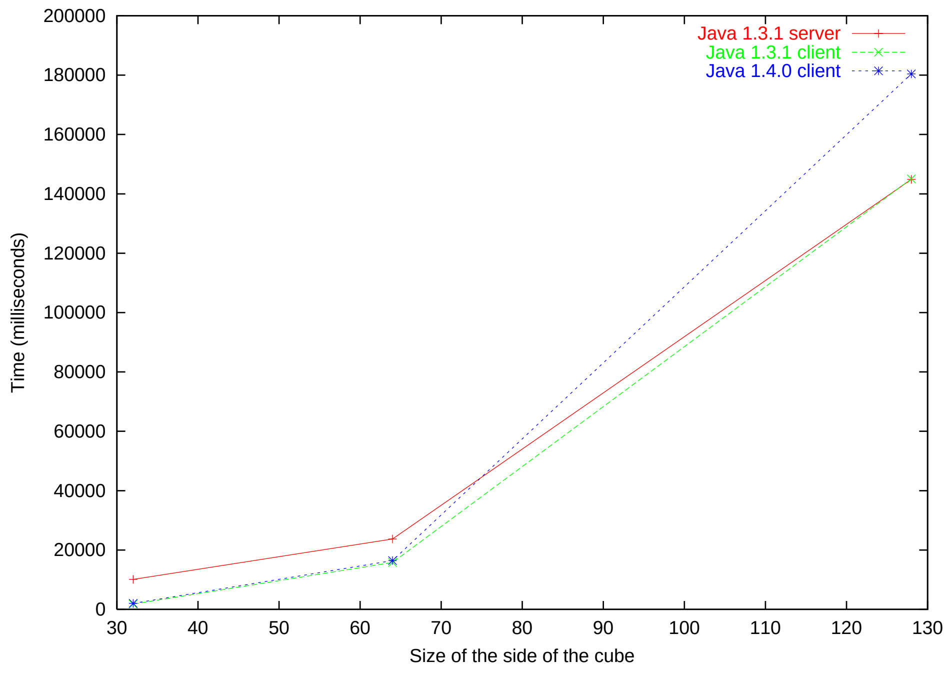

Figure 6 : Graphic of time of different Java Virtual Machines for the propagation algorithm.

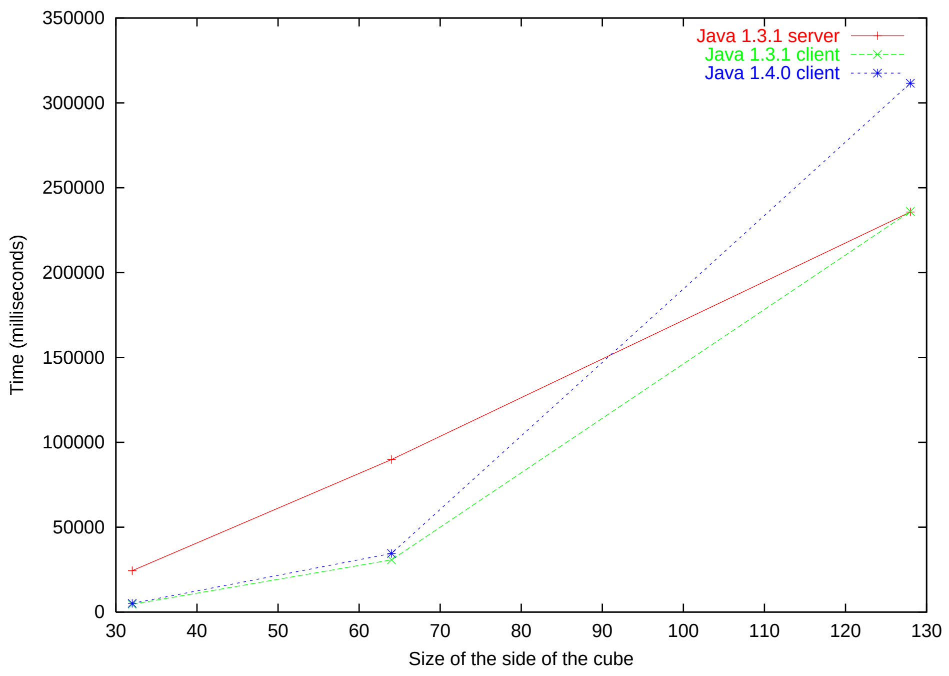

Figure 7 : Graphic of time of different Java Virtual Machines for the collision algorithm.

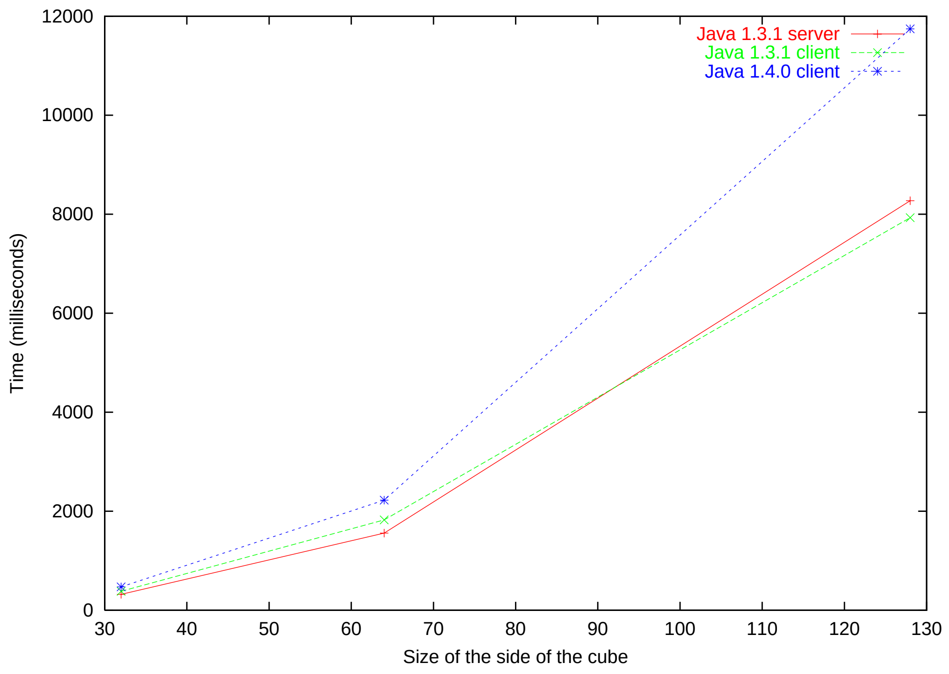

Figure 8 : Graphic of time of different Java Virtual Machines for the halo algorithm.

3 Comparison Optimized C versus Optimized Java

The optimizations performed were similar in both the C and Java codes, and the result obtained is a net gain in both codes. As can be seen in Figure 9, optimizing the C code obtains better results, since the optimizations map directly to the hardware. In Java, the JVM slows down the program. This shows that the C and Java compiler are performing similar optimizations, but neither are mature enough to perform the data transfer and pre-calculation of constants optimizations automatically.

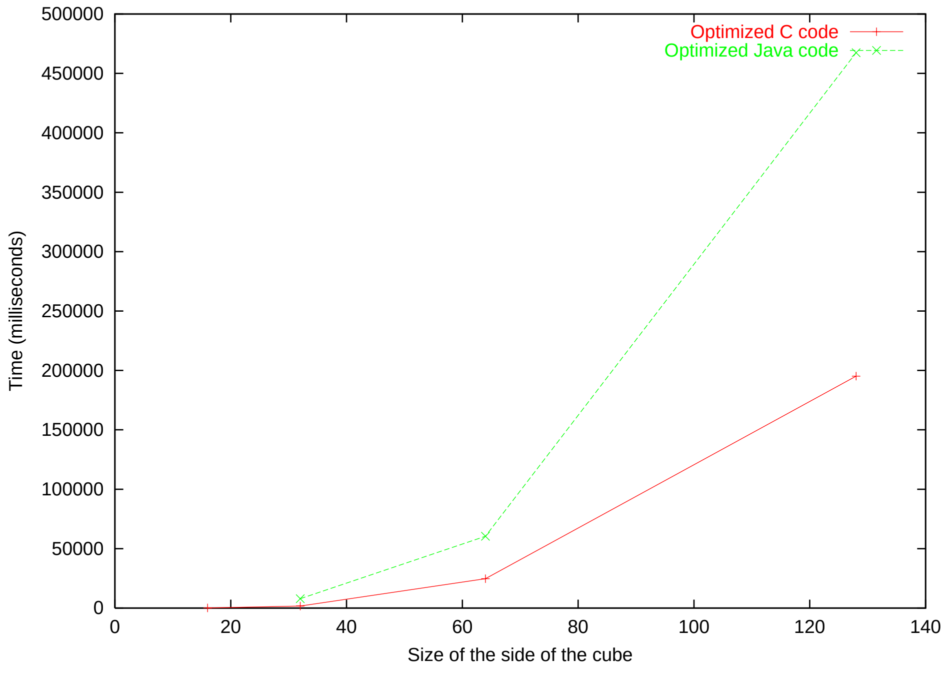

Figure 9 : Graphic of time of the optimized C and Java codes.

4 Parallel Java version

There are limitations in computer technology which impose a limit on the maximum speed of current machines. In order to bypass these limitations, the most popular solution nowadays is the design of parallel computers. The speed of each of these computers is still limited by the above statement, but the combined performance of all the processors can exceed that of any single computer. However, for this combined performance to be realised, well designed parallel programs are needed. Therefore an effort was made to port Ludwig to parallel architectures.

There are two main subarchitectures in parallel computing. One of them uses shared memory, and the other one uses distributed memory.

A shared memory machine has a memory bank accessible from all the processes. This means the data does not have to be distributed among the processors, and allows faster communication and synchronization. The drawback is that these architectures have a bottleneck in the processor-memory bus, which limits maximum number number of processors that can be added.

A distributed memory machine, on the other hand, has a set of memory and processor pairs. Each processor accesses only the local memory, and the comunication among processes are achieved by means of Message Passing only. This architecture allows better scaling, but has the disadvantage of a more difficult programming technique because the data has to be distributed among processors and all communication is to be explicitly coded.

Each of these architectures requires a different programming approach. Two standards have therefore been developed for these machines: Open Multi Processing (OpenMP) for shared memory machines, and Message Passing Interface (MPI) for distributed memory machines. Ludwig has been parallelized using both approaches. The OMP version is working, but the MPI version is still under development.

It is worth mentioning that both mpiJava[2] and JOMP[9] are still research prototypes, and have not yet been approven as standards. This is one example of the lack of mature parallelism of Java.

4.1 OMP

4.1.1 Algorithms developed

Some effort has been spent in improving the scalability of the Ludwig code. Three different propagation algorithms were developed in order to increase the scalability of Ludwig.

- First version
  A first version was developed which used the same approach as the C code. The benchmarking indicated that the propagation algorithm did not scale well. This made the overall scaling of the program low. The problem with the propagation algorithm was that a copy had to be made of the boundary planes of each processor because the other processors needed to access them to calculate the new values before they were overwritten by the processor calculating his own new values. As the number of processors increased, so did the amount of copying done. The final result was bad speedup in this zone of the program. When enough processors are used, Amdahl's Law limits the total speedup. Another drawback of this algorithm was the number of barriers it contains. These barriers limit the scalability, because the time to synchronize all the processors increases as more processors are used. This algorithm also had false sharing cache problems when used for high resolution cubes or with a small cache, because the size of the planes were a multiple of the cache size.

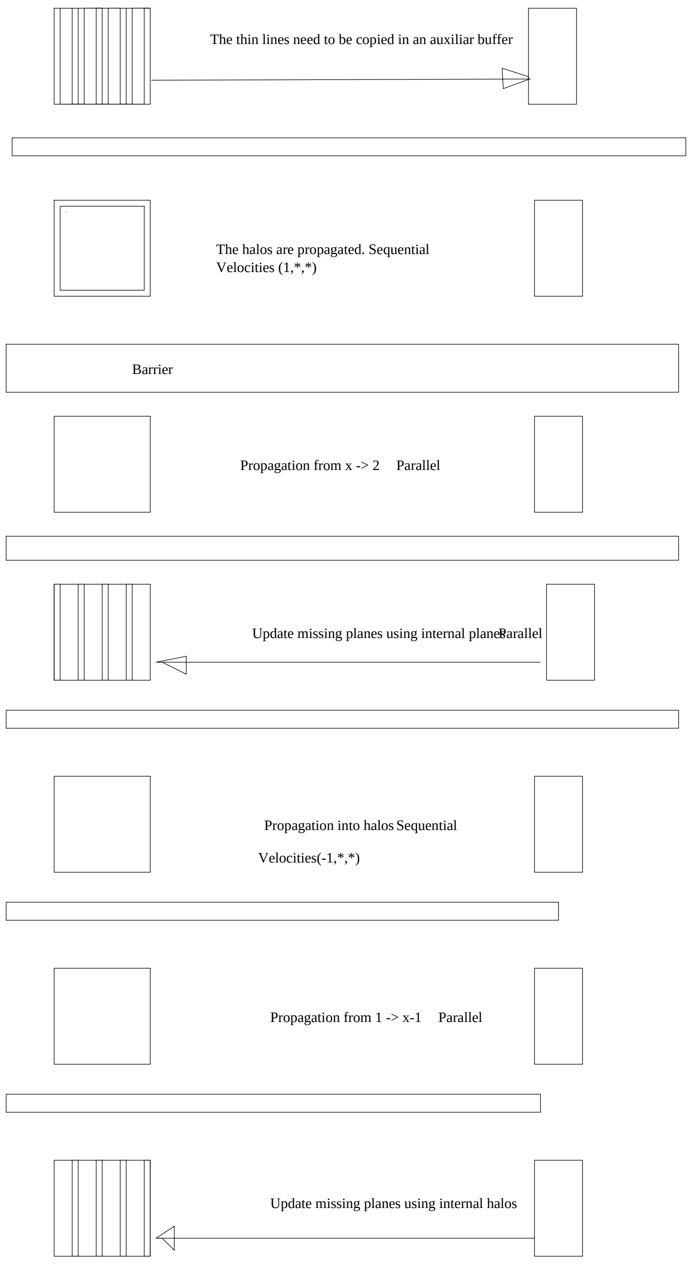
The thin lines need to be copied in an auxiliar buffer
The halos are propagated. Sequential
Velocities (1,*,*)
Barrier
Propagation from x -> 2 Parallel
Update missing planes using internal planes Parallel
Propagation into halos Sequential
Velocities(-1,*,*)
Propagation from 1 -> x-1 Parallel
Update missing planes using internal halos

Figure 10 : Diagram of the first version of the Propagation algorithm

- Second version
  A close examination of the sequential loops showed that part of the calculation done dealt with obtaining the first and last elements each processor had. That calculation was independent of the loop index and was moved outside the loop, so that it was only calculated once. This provided a small increment in speed.

- Third version
  The final approach was a new algorithm which consisted in using a double buffer for the sites array. This meant that no copying of the buffers was necessary, since the original array was not modified. This eliminated the non-scaling part of the algorithm at the cost of doubling the amount of memory needed. Since in High Performance the scaling is

much more important than the memory needs (because what is wanted is to be able to use more processors as new machines are available and a new machine usually means more memory as well), this new algorithm is preferable.

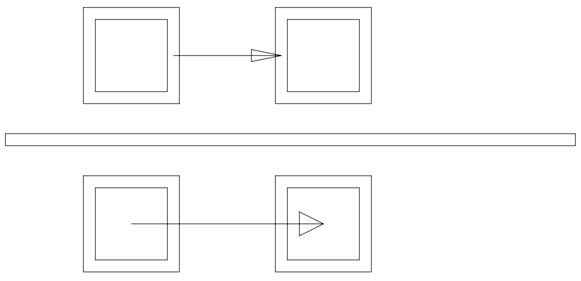

Figure 11 : Diagram of the double buffer version of the Propagation algorithm

- A naïve implementation was also discussed, which consisted in calculating the different loops each in one processor. The major disadvantage of this approach is that the number of loops is fixed, so the algorithm cannot use more processors than the number of loops.

4.1.2 Benchmarking results

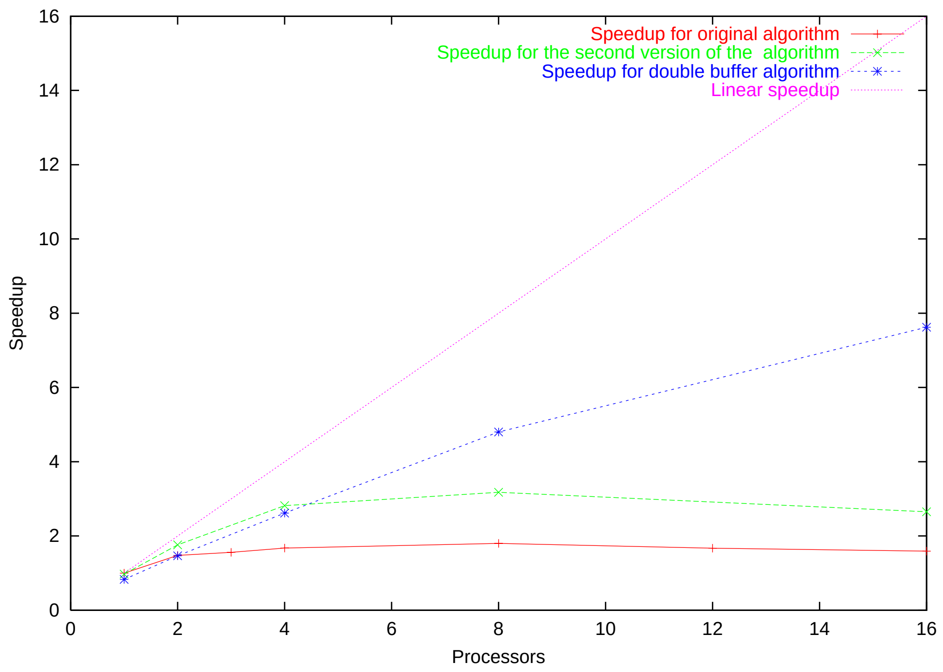

Figure 12 : Comparison between the scaling of the three different parallel versions developed.

Nevertheless, the speedup achieved was still less than expected. Further measures of the Bandwidth needed for the propagation algorithm indicated that the algorithm was suffering from the bandwidth limitation in the machine. In Figure 11 are presented the Bandwidth measures for the main routines of Ludwig, together with Stream, a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels.[7]

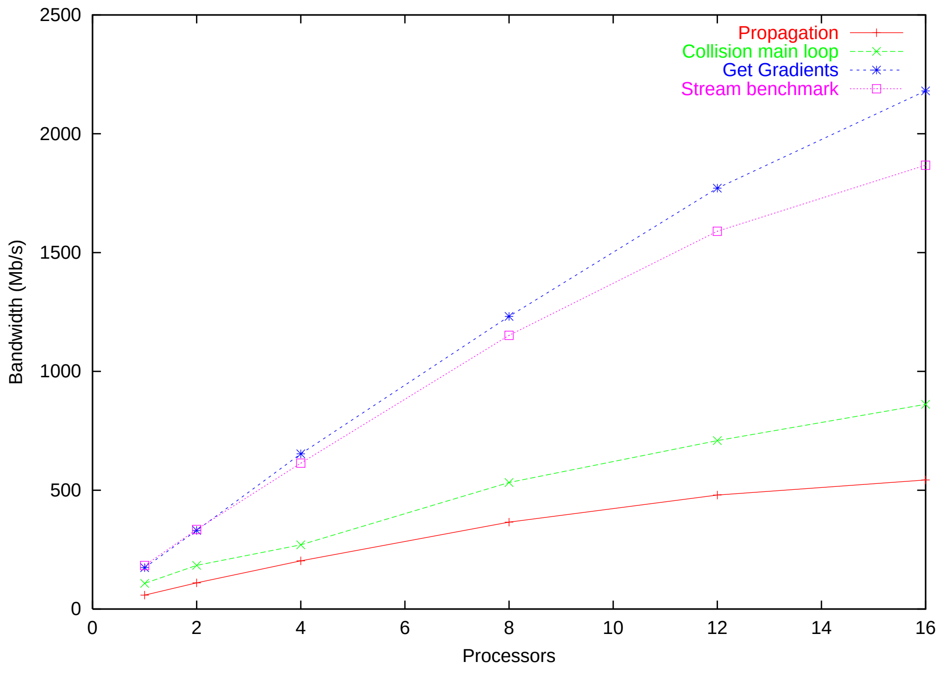

Figure 13 : Bandwidth of the different routines in Ludwig, together with the Bandwidth benchmark Stream version 4.0-BETA for C.

The different bandwidth observed corresponds to the cache misses in the different algorithms. Get gradients is an algorithm with a very strong spatial locality, so the number of cache misses is extremely low; therefore the bandwidth observed exceeds that of the Stream benchmark. Propagation and collision, on the contrary, have less spatial locality, and thus the lower bandwidth needed to reach saturation. These figures have been taken using java version "1.3.1"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1-b24)
Java HotSpot(TM) Client VM (build 1.3.1-b24, mixed mode)
in an 18 processor SunOS e6500 5.7 Generic_106541-16 sun4u sparc SUNW,Ultra-Enterprise

4.1.3 Comparison among different Java Virtual Machines

Four different JVM have been tested:

- java version "1.3.1"
  Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1-b24)
  Java HotSpot(TM) Client VM (build 1.3.1-b24, mixed mode)
- java version "1.3.1"
  Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1-b24)
  Java HotSpot(TM) Server VM (build 1.3.1-b24, mixed mode)
- java version "1.4.0-beta"
  Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.0-beta-b65)
  Java HotSpot(TM) Client VM (build 1.4.0-beta-b65, mixed mode)

- java version "1.4.0-beta"
  Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.0-beta-b65)
  Java HotSpot(TM) Server VM (build 1.4.0-beta-b65, mixed mode)

As can be seen in figures 14-17, the performance of the JVMs is similar. None of the JVMs is preferable to the others, their relative performance depending on the underlying program.

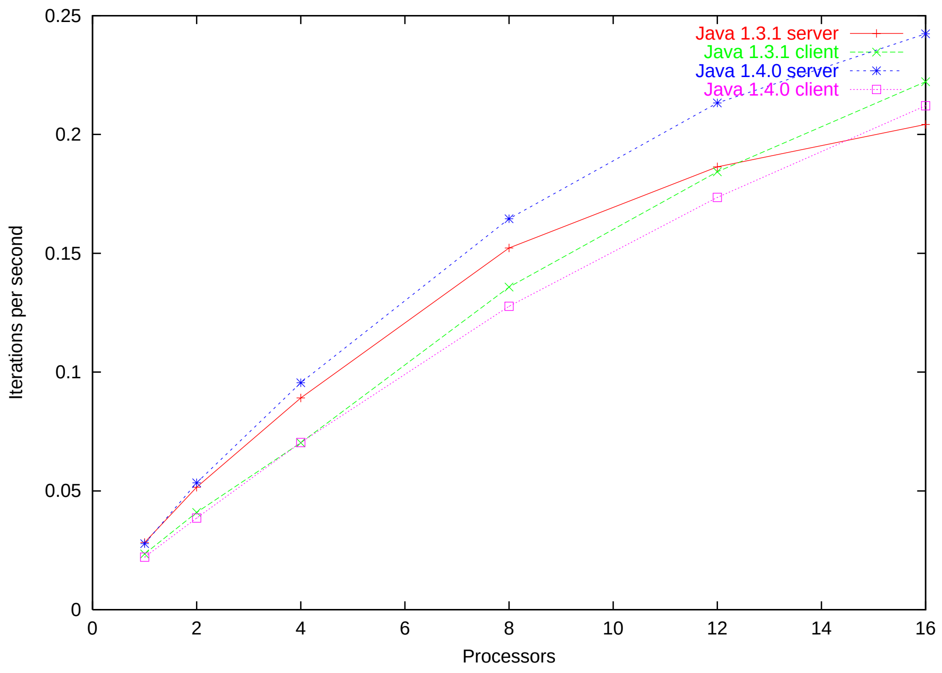

Figure 14 : Time of the Ludwig algorithm, with a 128 cube and no solids, using each of the Java Virtual Machines.

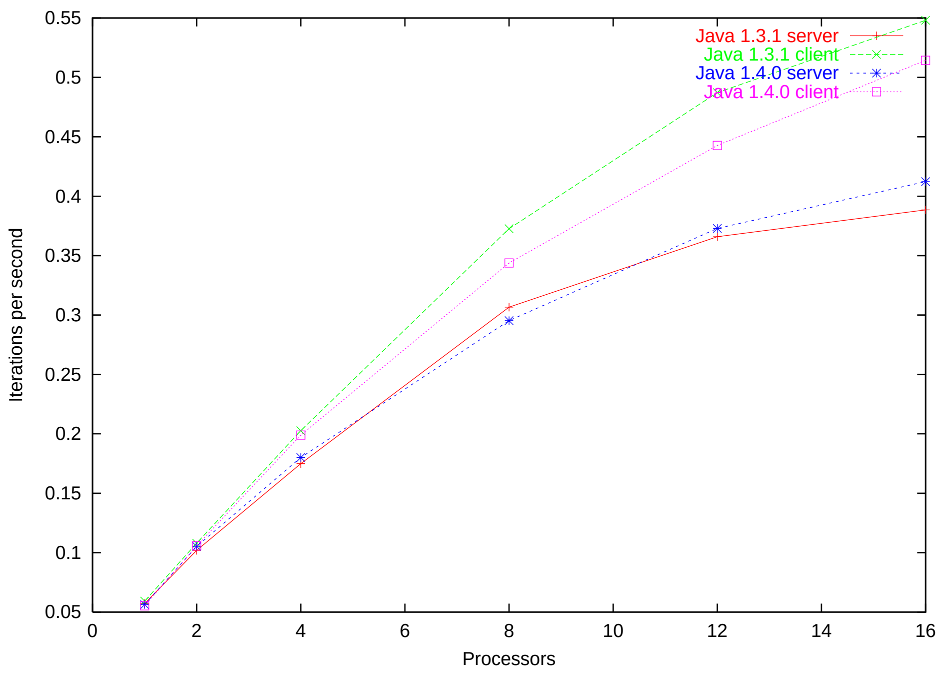

Figure 15 : Time of the Ludwig propagation algorithm, with a 128 cube and no solids, using each of the Java Virtual Machines.

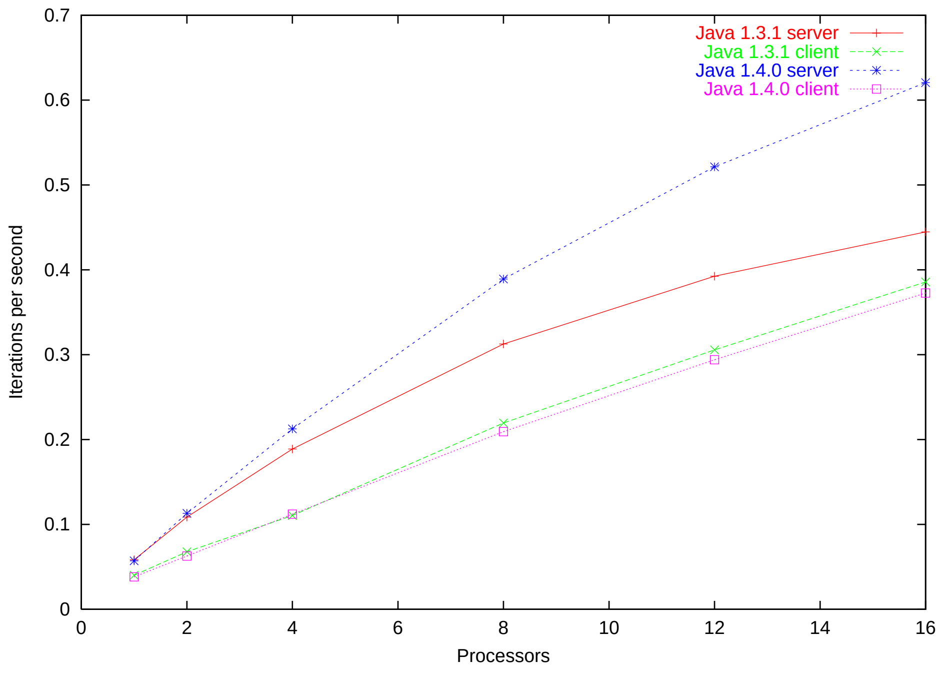

Figure 16 : Time of the Ludwig collision algorithm, with a 128 cube and no solids, using each of the Java Virtual Machines.

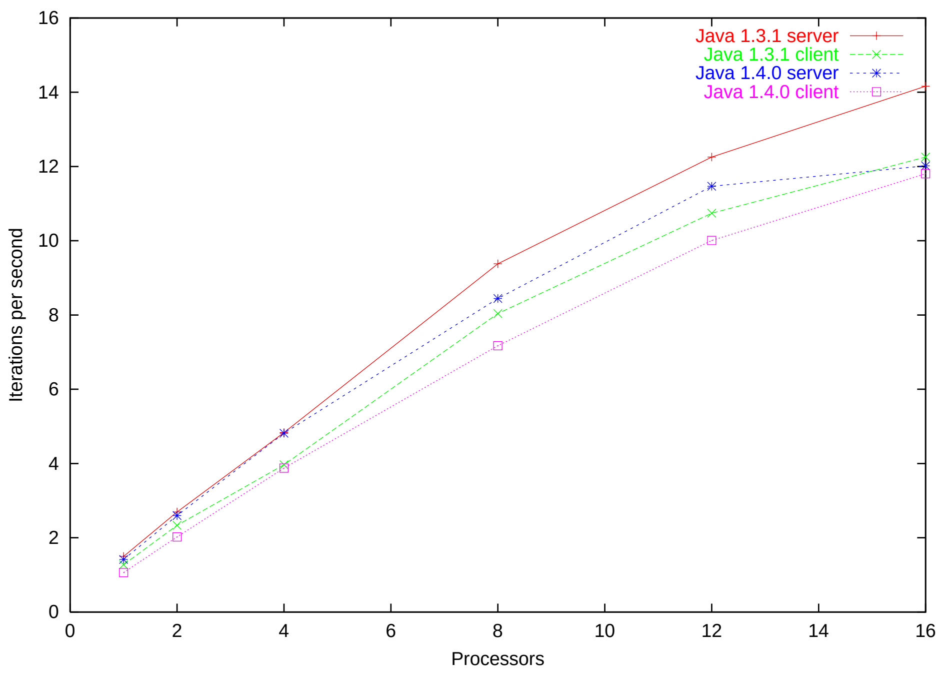

Figure 17 : Time of the Ludwig halo algorithm, with a 128 cube and no solids, using each of the Java Virtual Machines.

4.1.4 Benchmarking in NUMA architectures

The double buffer algorithm was benchmarked in an SGI Origin 2000 and in an SGI Origin 3000. The Origin 3000 and the Origin 2000 are NUMA (Non-Uniform Memory Access) machines. This means that each processor has its own memory, but it also serves other processors asking for data on the processor. These arquitectures have the scalability of the Distributed Memory machines with the additional advantage that they can be programmed as Shared Memory machines. However, there is one drawback. Attempting to access memory from other processor results in an important delay. Since the memory is distributed, care must be taken to distribute the data among all the processors so that they access local memory, in the same fashion as in a Distributed environment.

The difference between the Origin 2000 and the Origin 3000 is that the 2000 has two processors per node whereas the 3000 has four.

The LU factorization benchmark from the Java Grande Forum [6] was used to compare the results to a known scalable program. The resulting measurements are shown in Figure 16 (Origin 2000) and Figure 17 (Origin 3000).

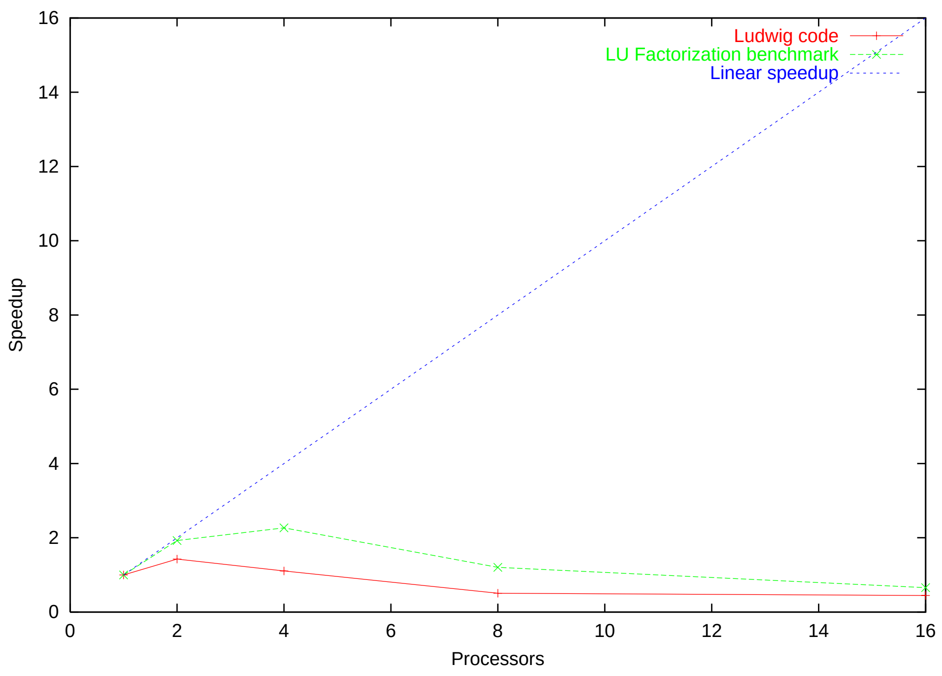

Figure 18 : Speedup of Ludwig and LU factorization in an SGI Origin 2000 using Java 1.2.1.

The JVM used in these measurements is: java version "1.2.1"
Classic VM (build JDK-1.2.1, native threads, mipsjit)
As can be seen, none of the algorithms scale at all.

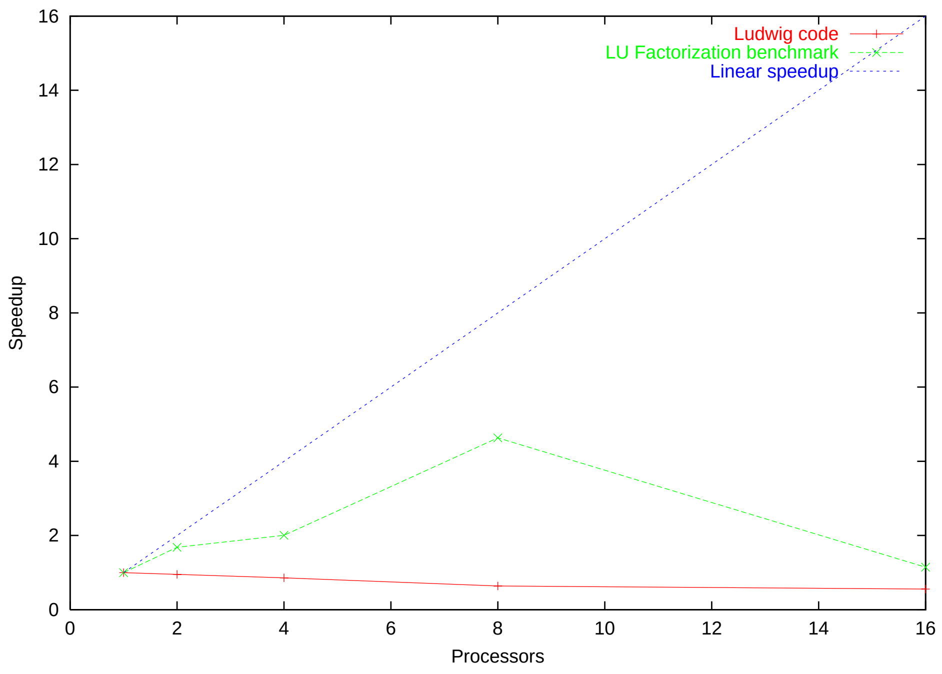

Figure 19 : Speedup of Ludwig and LU factorization in an SGI Origin 3000 using JavaVM-1.3

The JVM used was
java version "JavaVM-1.3"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3)
Classic VM (build JavaVM-1.3, build 1.3, native threads, mipsjit).

It is interesting to note that in this machine, the LU code scales reasonably well until 8 processes. As can be seen, the LU code scales well for two nodes in both processors. However, when more nodes are used (8 processes in the 2000, 16 in the 3000), the remote memory access limits the speedup. Wether this is a hardware problem or a consequence of the JVM Thread allocation procedure is not still clear.

The Ludwig code does not scale at all. The Ludwig OMP code was developed taking into account the properties of true Shared Memory machines. The net result is that most of the data is stored in process 0, making all the other processes need to access the memory in process 0. Since (as was seen earlier) Ludwig's scalability problems lie in memory bandwidth, no speedup can be achieved. In order to take advantage of NUMA machines, there are two possible alternatives:

- Change the program so that memory allocation is done among all the processes. This would make each process have the part of the data which they are going to use most afterwards.
- Use the MPI version of Ludwig. Since the MPI version has to distribute the data, these problems disappear.

These measurements were made requesting the processors needed, so the rest of the processors were used concurrently by other users. Therefore the results are less accurate than in the benchmarks shown earlier. The results shown are the best execution on the machine, since the load on the machine tended to slow all the processes down.

4.2 MPI

4.2.1 Porting problems for the MPI version

Java Ludwig was intended to be ported to mpiJava (Message Passing Interface for Java) during the project. However, the speedup problems detected in the OpenMP implementation have reduced the time spent on the mpiJava port. The mpiJava port is currently under development, with most of the routines ported.

5 Future work

5.1 MPI

The next step towards the completion of the Java port of Ludwig will be the completion of the MPI version. This would allow Ludwig to be run on distributed memory machines with the additional advantage, as was explained earlier, that it would also allow the use of computer resources with NUMA architecture effectively.

5.2 Object Oriented version

Another proposed addition to the Java Ludwig port is a rewrite of the application using Object Oriented techniques in order to improve maintenance.

This would allow measurement of Object Oriented performance penalties and might encourage further the use of Java if they prove to be negligible.

5.3 Adding specialized routines and graphics

- The Java version does not include all the different algorithms present in the C version. The C version contains two lattice geometries D3Q15 and D3Q19 to choose from. It also contains Lees-Edwards boundary conditions. Adding them to the Java version would improve usability.
- The DIVE library is a portable system for on-line visualisation of remote distributed parallel simulations using MPI graphic library.[1] Adding it to the Java port would be an important help for end users of the application.

5.4 64-bit mode

Currently there is a limitation in Java Virtual Machines to 32-bit addressing. This means only 2 Gb of memory can be referenced. With this limitation, the maximum resolution of Ludwig is a 128 cubic Lattice. In order to increase the resolution of Ludwig for more accurate simulations, a 64 bit VM is needed. In theory, none of the code of Java Ludwig will need to be modified to take advantage of new 64-bit Java Virtual Machines when they come out.

6 Conclusion

The aim of this project was to develop a medium-scale realistic application in Java. The basis for this project was be an existing code, called LUDWIG, which is written in C (and OMP/MPI). The major part of this project consisted of translating the sequential C code into Java. This allowed direct performance comparison between the two versions. As a further part of the project, parallel versions of the Java code (using mpiJava and JOMP) were developed and benchmarked in different parallel architectures.

We have seen that the speed of Java is high enough to allow High Performance Applications to be written in it. We have shown that it is possible to have performance similar to that of classic languages (C in this case). We have discussed the differences among different Java Virtual Machines. We would also like to note that the absence of stable 64-bit Java Virtual Machines limits the usability of Java in Grande applications.

7 Appendix: LUDWIG manual

Ludwig is a general purpose parallel Lattice-Boltzmann code, capable of simulating the hydrodynamics of complex fluids in 3D. Ported to Java by Rubén J. García Hernández during Summer 2001 for EPCC SSP SS-2001-01

7.1 Requisites

For sequential and OMP

    java 1.2.1 or newer.

For MPI

    java 1.4.0 or newer.

7.2 Build

From this directory, type

```
make
```

Remaking after modification:

```
make clean
make
```

If only one version is needed

```
cd <directory>
make
```

7.3 Directory Structure

.	This directory contains the makefile for the project.
sequential	This directory contains the source to the sequential code.
OMP	This directory contains the source to the OMP parallel code.
MPI	This directory contains the source to the MPI parallel code.
examples	Examples of configuration files for ludwig.
extras	C programs needed for config files (see extras/README): new_link_maker.

The MPI is not yet fuctional.

7.4 Installation

After typing make and waiting for all the files to build, there will be a set of .class files in each of the subdirectories. These files can be copied to a suitable directory (/usr/local/ludwig/sequential, /usr/local/ludwig/OMP and /usr/local/ludwig/MPI, or any other directory)

Then the directory of the program which is to be used should be added to the classpath, for example for the OMP program:

```
CLASSPATH= $ CLASSPATH:/usr/local/ludwig/OMP
export CLASSPATH
```

Take into account that since the name of the class is the same, only one of the programs can be in the CLASSPATH at the same time.

7.5 Running the program

java ludwig [<name of the input file>]
For big files it is usually necessary to add the -mx switch

```
java -mx1600m ludwig [<name of the input file>]
```

To tell the OMP version how many threads to use:

```
java -mx1600m -Djomp.threads=<n> ludwig [<name of the input file>]
```

7.6 Other files

input: This file contains the parameters of the simulation The file is organized in lines. Each line contains a keyword, a space, and a value.
Parameters not included in this file are set to sensible default values in the routine MODEL_init.
The most used keywords include:

size: size of the lattice. Underline separated list of 3 numbers with the X, Y and Z size of the lattice.
N_cycles: Number of cycles to simulate
A, B and K: free energy parameters. They determine the quench depth (ratio $\frac{A}{B} = 1$ for a deep quench) and interfacial tension ($\sigma = \sqrt{\frac{8KA^3}{9B^2}}$).
C and H: Used for the wetting (not yet implemented): $As = 0.5C\phi^2 - H\phi$
viscosity: Viscosity of the fluid
mobility: Mobility of the fluid
freq_measure: After how many cycles must the phi and vel parameters be displayed?
freq_config: After how many cycles must the whole configuration be dumped to disk?
output_format: BINARY or ASCII
input_site_data: file with the location of the solids. See *.sd below.

Example:

```
size 32_32_32
N_cycles 10
A -0.00625
B 0.00625
K 0.004
C 0.0
H 0.0
viscosity 0.005
mobility 4.0
freq_measure 100
freq_config 100
output_format BINARY
input_site_data wall.sd
```

$*$.sd: This file contains the solids present in the simulation. These files are generated by the C program new_link_maker. See [5].

config.out∗: This file is generated by the program and contains the final configuration.

8 Bibliography

References

[1] Bálint Joó. DIVE: Distributed Interface to Visualization Environment. http://www.epcc.ed.ac.uk/ssp/Reports/ss9612.ps.gz, 1996.

[2] Bryan Carpenter. http://www.npac.syr.edu/projects/pcrc/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html.

[3] Giulio M. Occhionero. The Boltzmann Equation. http://ourworld.compuserve.com/ homepages/ Giulio_Occhionero/ boltz.htm.

[4] I. Pagonabarraga,J.-C. Desplat,A.J. Wagner,M.E. Cates. Interfacial dynamics in 3-D binary fluid demixing: animation studies. New Journal of Physics 3 (9), 2001.

[5] J.-C. Desplat, I. Pagonabarraga, P. Bladon. LUDWIG: A parallel Lattice-Boltzmann code for complex fluids. Computer Physics Communications 134, pp 273-290, 2001.

[6] J. M. Bull,L. A. Smith,M. D. Westhead,D. S. Henty, R. A. Davey. A Benchmark Suite for High Performance Java. Concurrency: Practice and Experience 12, pp 375-388, 2000.

[7] John McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/.

[8] Lorna Smith,Mark Bull. Java for High Performance Computing. Edinburgh Parallel Computing Centre.

[9] M. E. Kambites,J.Obdrzalek,J.Mark Bull. An OpenMP-like Interface for Parallel Programming in Java. To appear in Concurrency and Computation: Practice and Experience.

Rubén J. García Hernández is studying his 5th year of Computer Engineering in the University of Granada (Spain). He is a student in the Edinburgh Parallel Computing Centre Summer Scholarship Programme 2001 in Edinburgh.

Supervisors in this project:
Dr J-C Desplat
Dr Mark Bull
Thanks to Dr Alexander Wagner and Dr Lorna Smith for their help.