PicUp3D/Spis Technical Notes
PicUp3D Project

C++/Java comparative benchmark

DRAFT V0.1

Prepared by :
J.Forest

Avril 2003

PicUp3D project, SPINE network

Abstract

In the framework of the SPIS project and in the aim of the selection of the computing language used for the low level simulation kernels, a benchmark was developped to compare performances and facility of developement of C++ versus Java languages. The two main methodes, i.e the Poisson's equation solver and the particle pusher, of the simualtion kernel of PicUp3D have been transleted frm Java to c++. Two versions od the code have written in c++, one in the most straight forward manner, without specific addaptation to c++, a second one including corrections in odrer to be the closest possible to the C++ ANSI norme. Benchmarchs have surprisinly that the differences

1 Document Status Sheet

DOCUMENT STATUS SHEET			
1.DOCUMENT TITLE :			
2.ISSUE	3.REVISION	4.DATE	5.REASON OF CHANGE
V0.1 V0.2		December 2000 February 2000	Creation Extension, Correction

2 Document Change Records made since last issue

This section is not applicable.

Contents

1 Document Status Sheet **I**

2 Document Change Records made since last issue **I**

3 Introduction **1**
3.1 Purpose 1
3.2 Remark 1
3.3 Overview 1
3.4 Definitions, acronyms and abbreviations 1

4 Comment about virtual machines and compilers **2**
4.1 Java Virtual Machines and Java compilers 2
4.2 C++ compilers 3

5 Studied sources codes **3**
5.1 Java versus C++ translation 3
5.2 Poisson's equation solver 4
5.3 Particle pusher 4
5.4 Native version of PicUp3D 4

6 Benchmarks and results **5**
6.1 Benchmarks conditions 5
6.2 Poisson's solver 5
6.3 Particle pusher 6
6.4 PicUp3D, native versus byte-codes versions 6
6.5 Memory cost 7

7 Conclusion **7**

3 Introduction

3.1 Purpose

The purpose of this document is to present methods used to describe the geometry of the satellite and the related technique considered to detect the particle/surface intersection. This last point, based on a technique issued from the ray-tracing domain, is exposed in detail.

3.2 Remark

This document refers to the versions 2.7.2 and later of the PicUp3D/Spis system and the Spacecraft3D class.

3.3 Overview

In first part, the model choosen for the spacecraft geometry is presented, included its representationin memory. In a second part, the theory of the method of detection of reacing particles is presented in details. In a third part, results of tests are presented and discussed. Last, propositions of optimisations and extension of these technics are briefly introduced.

3.4 Definitions, acronyms and abbreviations

Table of the symbols used in this document:

All characters in bold correspond to vectors.

In the implementation part, the words methods and class refer to expression commonly defined in the object oriented computing methods.

References

4 Comment about virtual machines and compilers

4.1 Java Virtual Machines and Java compilers

Initially, Java, developed by SUN, is a half-interpreted language, where source files are compiled into byte-code files, themselves executed by an interpreter called by Java Virtual Machines (JVM). This approach allows a very high portability of the binaries, while a JVM exists for the present platform. At the present date, JVMs exist for a very large set of platforms, including i386/Linux, i386/Windows and almost all common working stations. The Java language is the property of SUN inc. and the language specifications are defined uni-laterally by SUN. This approach leads to a strict definition of the various versions of the language and compilers, increasing the portability on various JVM and development kits.

JVMs have done a lot of progress with the introduction of the Just-In-Time compiler technique, where classes are directly compiled into native codes during them loading in memory. These classes are then run as almost equivalent to native sub-processes (threads) by the operating system. This approach is especially efficient with large classes which will be re-called or re-used many times during the code execution. This is generally the case in intensive numerical simulation, where the main loop can be summarised into a limited set of classes.

Modern JVMs (version higher then 1.2) allow multi-threading, either internally (green threads) in the JVM, either directly by the operating system (native threads) via a set of sub-processes.

In this last configuration, and if the OS is multi-task and manage multi-processor computer, threads are automatically distributed on the various CPUs and are run in parallel. This approach offers the possibility of local parallelism without special library such as MPI.

Tests have shown that global performances of byte-coded java codes are strongly dependent on the JVM/OS couple. Currently, the following table summaries the best results for various operating systems:

Operating System / platform	JVM
Linux/i586	IBM 1.4
Windows/i586	SUN 1.4

Java compilers should be divided into two categories, compilers able to produce byte-codes, such as javac furnished with JDKs, and compilers able to produce directly natives executable codes, such as GCJ or IBM VisualAge for Java.

Classic byte-code compilers, like as javac, offer generally very poor optimisations capabilities. Better performances may obtained using optimised compilers such as KJC of the Kopi open source project. Java compilers are generally quite severe regarding the security of the source code, with, for example, errors in case of variables not initialised or exceptions controls. On the other hand,

The last versions of gcc 3.2.0, the general gnu compiler, includes a front-end, called gcj, able to compile Java source code either directly to executable native codes than classic byte-codes. Currently, GCJ has still a few limitations, especially regarding the graphical functionalities (AWT based classes). They may generally be compiled but not properly executed. Practically, calls to methods of AWT classes should be removed of source codes. The optimisation mechanisms are based on methods implemented in gcc and possibilities of combinaisons of control flags of control are high. Flags may influence strongly the final performances of the generated codes (two factor at least). Gcj is still under development and the global level of optimisation stays lower than the equivalent version of gcc for C++. The $-Wall$ flag leads to sever control during the compilation than standard compilers and maybe very useful to debug and/or optimised manually compiled codes (e.g. removed unused variable). Gcj has been ported on a large set of platforms, including Linux and Windows. However only the Linux version has been tested here.

In the framework of the present tests, the following Java compilers has been used:

Operating System / platform	Compiler	byte-code	native
Linux/i586	IBM 1.4	X	
Linux/i586	SUN 1.4	X	
Linux/i586	KJC	X	
Linux/i586	GCJ	X	X
Windows/i586	SUN 1.4	X	
Windows/i586	IBM 1.4	X	

4.2 C++ compilers

Two C++ compilers have been used for the present tests, the standard Gnu G++ compiler and the Intel ICC compiler, optimised for Pentium class processors.

G++ is the front-end of the open source GNU gcc compiler for C++. Released under the GPL license, it has been ported onto a very large set of platform and became *de facto* a standard today. The most distributed version, 2.95.xx, is progressively replaced by the 3.2.xx, that offers better performances (about 20% faster than 2.95.xx). However, one observe frequently difficulties to compile with g++ 3.2.xx codes previously fully compliant with g++ 2.95.xx. This is the case, for example for the GEANT4 and the ROOT libraries.

The second c++ compiled tested is ICC 7.0, a compiler optimised for Pentium processors and developed by Intel. A trial version maybe freely downloaded from the Web for Linux. This compiler produces codes generally faster than g++ 3.2 from 8 to 10 %.

Because, gcc is released under the GPL license, exists on almost all platforms and has proof its reliability and its global performances, it will be considered here as the standard compiler and development tool.

5 Studied sources codes

5.1 Java versus C++ translation

Java and C++ languages present very close syntax and the translation from Java to C++ is quite straight forward, especially for simulations kernels, without calls to system or graphical features. The table below summarises the main keywords to be translated:

Java	C++
System.out.println("hello world !")	cout << "hello world !" << \n;
public class MyClass{...}	class PlasmaLEO { public : ....};
float[][] Particle;	float **Particle;

rho = new float [Ngx][Ngy][Ngz];	rho=(float ***)calloc(Ngx,sizeof(float));
	for(i=0;i¡Ngx;i++)
	rho[i]=(float **)calloc(Ngy,sizeof(float));
	for(j=0;j¡Ngy;j++)
	for(i=0;i¡Ngx;i++)
	rho[i][j]=(float*)calloc(Ngz,sizeof(float));

There are a few fundamental diffrences between Java en C++.

Prototypes and headers In C++, prototypes of classes and methods should be declared before their definition, generally in headers files. This is not the case in Java, where prototypes are not required.

Memory management Java and C++ differ by some aspect regarding the memory allocation. Both languages allows dynamical memory allocation. However, in Java, all elements (variable, object) are dynamical. The memory release is is automatically managed by a garbage collector, which removes all unused elements (i.e. when their handle is pointing on NULL). The developer has not to take care

about memory release and risk of memory licking, that are almost impossible. The negative side of this approach is a significant extra-cost in memory, due to the fact the garbage-collector technique is still unoptimised in the current JVMs. However, a strict definition and use of destructors may reduce this problem. In C++, objects and variables maybe instanced statically and dynamically. Static instantiation may lead to better performances in terms of memory cost and time access, but leads to more rigid codes. In case of dynamical memory allocation, one of the most constraint is the need the release correctly the memory after use of the object by a correct definition of the destructor. The no respect of this rule may may lead to severe memory overflows and segmentation faults during the execution. This problem may be very constraining in case of complex software or when objects are instanced via a additional layer, such as a script language.

In the context of this benchmark, the main class of the PicUp3D/Spis library, PlasmaLEO, and its associated mathematical class, VectorAlgebra3, were translated into C++. PlasmaLEO provides a complete PIC model. Tests have especially been done the Poisson's equation solver and the Particle-Pusher.

Two versions of the C++ version was developed. The first one corresponds to a "direct and rough translation", with the minimum of modifications from Java source code. The second one includes the specificities of the C++ language and try to be the closest possible to the ANSI C++ norm.

In both languages, Java and C++, codes were written in the simplest manner possible, without optimisation, specific to one language or the other one.

5.2 Poisson's equation solver

The Poisson's equation solver was the first method tested (PotSolver.ppt and PotSolver.java); This solver is based on Gauss-Seidel's method with a Chebychev acceleration (REF TO NUMERICAL). This is an iterative method mainly based on a set of loops on multi-dimensional arrays and algebra operations on floats. Because until now, many JVMs present a very poor and inefficient implementation of the for-loops, these last ones were converted into do-while loops on the most critical loops.

Poisson's equation was solved two times on a $60 \times 60 \times 60$ grid, with a punctual charge of 10.0 at the centre of the grid.

5.3 Particle pusher

In a second time, the particle pusher, called *Motion* in the PlasmaLEO class, was tested. This method is based on an explicit Leap-Frog scheme using Boris's algorithm and including the magnetic moment. The *Motion* method includes a linear (CIC) interpolation scheme. The method includes many large loops (on particles and surrounding grid), algebraic operations on floats and many if-tests on floats. The method implements and calls VectoAlgebra3 objects.

In the test, a large set of particles (100000) is let orbiting around a central charge during 1000 time steps ($dt = 0.051/omegq_{pe}$). The potential map is computed before by the *Poisson* method on the same grid than previously.

5.4 Native version of PicUp3D

The current version of PicUp3D (2.8.16) and its associated library (PicUp3D/Spis) was integrally compiled into native code for Linux with GCJ 3.2. This has required only the following modifications:

- Modification of the general makefile in order to define a new target
- Suppression of all unused variables (presence of the -Wall flag during the compilation)
- Suppression in the PicUp3D.java main file of all references (instantiation and calls) to graphical methods.

6 Benchmarks and results

6.1 Benchmarks conditions

As possible, tests and benchmarks have been done on "realistic systems" regarding the demand of PIC models. Poisson's equation has been integrated two times on 60 grid. In all cases, the total computation time was long enough to neglect the duration of the initialisation phase (initialisation of the JVM, memory allocation, compilation JIT). Excepted special mention, disk access and graphical outputs have been removed in order to do not introduce perturbation linked to iterations with the operating system. The job duration have been obtained from the "user" value returned by the time command under UNIX. Under Windows, tests have been done under the Sygwin environment, that provides the equivalent command. Tests have been done both on an AMD K-6/Athlon processor at 700 Mhz with 512Mo of PC133 RAM and an INTEL Celeron processor at 700 Mhz with 380Mo of PC133 RAM. In both cases, the amount or RAM memory was enough to avoid any swap disk.

6.2 Poisson's solver

Numerical results obtained with both versions were compared. No significant difference was observed between both results. In all cases, the difference stays lower than the incerterty of the numerical method used.

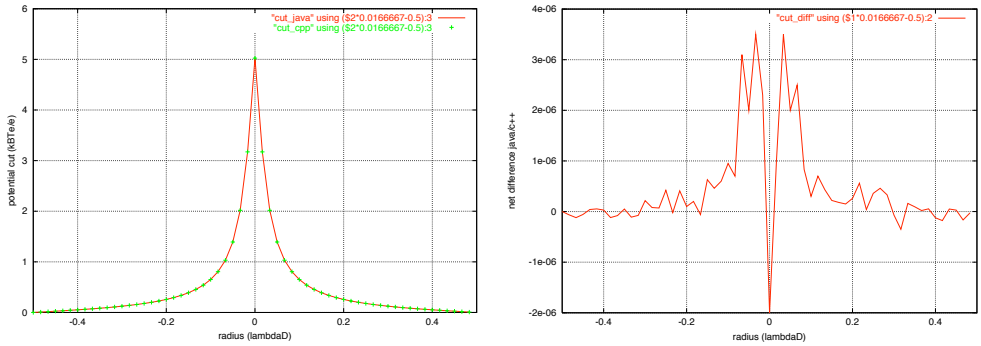

Figure 1: The left panel shows an radial cut of the potential map computed with the various version of the code. The right panel shows the net difference between these results obtained with the java and c++ versions. The difference is always lower than the accuracy of the implemented numerical method and is not related to the language.

Without optimisation flag In all tests, the Java version of the code was tested under its byte-code forme with the IBM 1.4 and SUN 1.4 JVMs, and as native code, after compilation with GCJ 3.2. The C++ version was compiled with G++ 3.2.0 and ICC 7.0.

A first set of trial was done without any optimisation flags, in order to evaluate the global performances in the most basic mode. The performances obtained with the C++ version compiled with G++ 3.2.0 in this configuration were considered as reference.

Fig. 2 shows the total duration of test runs for each version (java/c++) of *PotSolver*. The two first bars (IBM 1.4 and SUN 1.4) correspond to Java byte-coded version of *PotSolver.java*, respectively compiled and run with the IBM 1.4 and SUN 1.4 JVMs. In this configuration JVMs gives results almost equivalent to the C++ version compiled with gcc 3.2. Actually, the java byte-coded version run with the IBM JVM is 1.6% faster than the c++ version, with respectively a run duration of 154.39 s and 156.95 s. This good result is a direct consequence of the JIT technique and confirms that, after compilation the PlasmaLEO class is run almost as a native code. Surprisingly, the java native version compiled with gcj presents very bad results (73According to CITEFAQGCJ, this is due to a very unoptimised implementation of the array bound checking method in the current version of

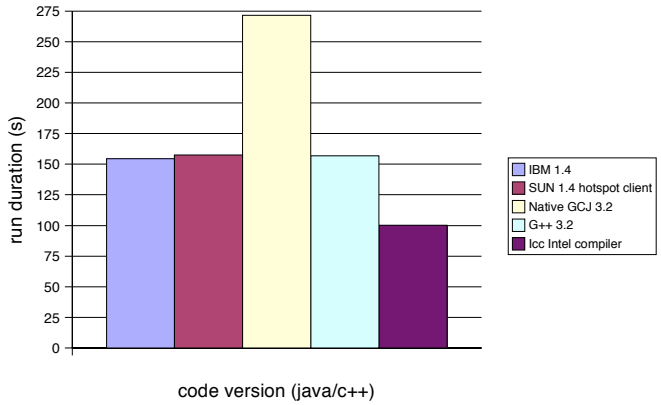

Figure 2: Duration of the tests runs in configuration without optimisation flag. The bars IBM1.4 and SUN 1.4 correspond to the Java version of the test code, run as byte-codes under respectively IBM 1.4 and SUN 1.4 JVMs. The GCJ bar corresponds to the Java version compiled into native code with GCJ 3.2. The two last bars correspond to the C++ version respectively compiled with G++ 3.2 and ICC 7.0.

GCJ. This problem may be avoided with the *–no-bound-check* flag during the compilation. The best performances are given with the c++ version compiled with the INTEL ICC compiler, that is 36 % faster the reference. In this configuration basic configuration java byte-coded version is slower than the fastest c++ native version of about 50%.

With optimisation flags In a second time, various options of optimisation were tested during the compilation (see Fig.3). As in the previous case, ICC with the -O3 option gives the best result. However, on a Celeron processor the increasment of the performances with respect the configuration without optimisation flag seems very weak. The same observation was done on an AMD K-6. However, many optimisation algorithms implemented in ICC are optimised for the Pentium CPU familly and more especially the Pentium-IV, not tested here. Results strongly different maybe expected in this case. Regarding the java byte-coded version, IBM 1.4 JVM gives the best results. As for ICC, the increase of performances with respect to the previous test is negligible. This is probably due to the very low level of optimisations of the java compiler (*javac* provided with the JDKs). In this case, the java byte-coded version stays significatively slower than the c++ version compiled with ICC. However, the used of the *-O3 –no-bounds-check -ffast-math -finline-functions -fexpensive-optimizations* increases dramatically the performances of the native java version compiled with GCJ, that is then faster (7.6%) than the c++ version compiled with GCC. In this configuration, the java native version is slower than the c++ compiled with ICC of only 2.74%.

6.3 Particle pusher

6.4 PicUp3D, native versus byte-codes versions

PicUp3D (version 2.8.16), and its related libraries, was integrally compiled without modifications of the source code, excepted the elimination of unused variables (to resect the $-Wall$ flag) and all calls to graphical methods (AWT based). All other features, including disk access in ASCII and binary modes, were untouched and compiled without problems. This global test confirm the The performances of the final executable seem very dependent on the compilation and linking flags.

The best performances where found with the following control flag configuration: containing the main method:

```
-Wall -O3 --no-bounds-check -ffast-math -finline-functions -fexpensive-optimizations
```

for the compilation and with the /it -static during the linkage.

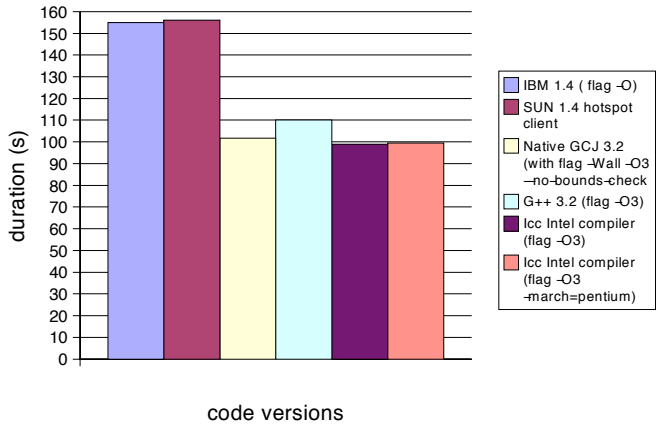

Figure 3: Run duration for the various versions of the test code *PotSolver*. Best results are given by the c++ version compiled with the Intel ICC compiler. The fastest java byte-coded version is about 55Java native version is faster (7.6%)) than the c++ version compiled with G++ 3.2 and slower of 2.74% than the version compiled with ICC.

6.5 Memory cost

Java more costly

7 Conclusion

GCJ very sensitive on flags.

close syntax surprisingly good performances of Java, with a very weak difference with C++ on th studied configurations. Difficulty of portability of C++ form one compiler to another one. memory cost for Java. Present elements maybe used for future tests of integration into framework languages such as Python/Jython.