A Java vs. C++ performance evaluation: a 3D modeling benchmark

L. Gherardi D. Brugali D. Comotti

University of Bergamo, DIIMM, Italy
{luca.gherardi,brugali,daniele.comotti}@unibg.it

Abstract. Along the years robotics software and applications have been typically implemented in compiled languages, such as C and C++, rather than interpreted languages, like Java. This choice has been due to their well-known faster behaviors, which meet the high performance requirements of robotics. Nevertheless, several projects that implement robotics functionality in Java can be found in literature and different experiments conduced by computer scientists have proved that the difference between Java and C++ is not so evident.
In this paper we report our work on quantifying the difference of performance between Java and C++ and we offer a set of data in order to better understand whether the performance of Java allows to consider it a valid alternative for robotics applications or not. We report about the execution time of a Java implementation of an algorithm originally written in C++ and we compare this data with the performance of the original version. Results show that, using the appropriate optimizations, Java is from 1.09 to 1.51 times slower than C++ under Windows and from 1.21 to 1.91 times under Linux.

1 Introduction

Robot software systems are concurrent, distributed, embedded, real time, and data intensive. Computational performance is a major requirement, especially for autonomous robots, which process large volumes of sensory information and have to react to events occurring in the robotics operational environment.

In order to meet performance requirements, robotics algorithms have been typically implemented in C and C++. Robotics developers in fact have always considered C++ significantly faster than Java. Despite that, the idea of using it in robotics is not really new: it has been followed in several projects (see section 2) and recently Willow Garage and Google have started a project for developing a Java-based porting of ROS [4].

In this paper we report our work on the comparison of performance between Java and C++. Our goal is to quantify this difference and to offer a set of data in order to better understand whether the performance of Java allows to consider it a valid alternative to C++ or not. For this purpose we implemented in Java a well known algorithm originally written in C++ and we executed a comparison study. The chosen algorithm is the Delaunay triangulation and its implementation comes from the OSG library[2]. It was developed in the computer vision field but it is typically used also in robotics for reconstructing environment surfaces from a set of 3D points. The algorithm is well suited for the purpose of our study because it stresses several critical points of the programming languages performance such as: (a) the frequent access to the memory for operating on dynamic size array (massive use of the garbage collector) and (b) the frequent evaluation of logical conditions.

Although in the computer science domain many comparison studies has been proposed, we considered our test interesting because we implemented and executed the algorithm with a newer and improved version of the Java JDK. Indeed the current Java Virtual Machine (JVM) offers a new compiler, which greatly improves the performance of Java with respect to the older versions.

The paper is structured as follows. Section 2 reports about the Java related projects in robotics and presents a survey on the differences of performance of the two languages. Section 3 illustrates a performance comparison case study. We present a Java-based implementation of a mesh generation algorithm originally written in C++ and report several information about the execution time of both the Java and the C++ versions. Finally section 4 draws the relevant conclusion.

2 Java for Robotics

Java is an object oriented programming language and it was intended to serve as a new way to manage software complexity. It offers to its users a set of software libraries and specifications, which allow the designing and the deploying of cross-platform applications. Java is used in different application domains such as enterprise resource planning (ERP) and web servers (e.g. JSP). It is widely spread also on mobile phones and embedded devices. This section presents a set of robotics project developed with Java and a survey on several performance comparisons between Java and C++.

2.1 Robotics Java projects

During the 2011 Google I/O the researcher of Willow Garage and Google presented a new project that aims to develop a pure Java implementation of ROS [4]. By means of this project Google and Willow Garage aim to boost the development of advanced Android applications for robotics and easiness the access to the cloud computing for reducing the cost of the robotics hardware.

In [11] the integration of Matlab in a distributed behavioral robotics architecture is presented. The architecture is completely implemented in Java and leverages on the Jini platform for distributed object registration, lookup and remote method invocation. The Matlab integration is realized by means of JMatLink and allows the invocation of Matlab scripts and the access to the Matlab workspace as a distributed object. The authors present as case study a multi-robot mines detection. In [17] a team from Lund University demonstrated that it is feasible to develop a motion control system entirely in Java. They designed an application that takes a picture of a person and controls a pick and place robot in order to draw on a paper the result of the shooting. The software and the motion controller guarantee the respect of the real time constraints by means of Java RTS. In [16] a real-time control for a remote manipulator over a local area network or over internet is presented. The developers implemented both the control system and the teleoperation of the robot in Java. In [9] an autonomous motion planning system completely developed in Java is developed. The application allows the user to set up the working environment though a graphical interface and offers the functionalities of collision detection, obstacle avoidance, free-paths generation and selection of the shortest path. Finally in [14] an application for controlling robots through the World Wide Web is implemented. The software is designed for dealing with low bandwidth and high latency and allows the operator to control the robot from any computer connected to the web.

2.2 Java versus C++

One of the main differences between Java and C++ is that the first was born as an interpreted language while the second as a compiled language. Compiled languages are translated into machine code trough a compiler. This process generates a file that can be directly executed by the CPU. Interpreted languages are compiled in a platform independent language (bytecode), which can be executed only by means of an interpreter (e.g. JVM). Hence, programs written in C++ (compiled language) are platform dependent and must be compiled for every computing platform before the first execution. Java programs instead are translated into bytecode only once and can be used on different platforms but have to be interpreted at every execution (the JVM is platform dependent). For this reason interpreted languages are in general more flexible and portable than compiled languages but at the same time slower.

In order to improve the performance, in 1998 Java 1.2 was released with a new feature called Just-In-Time compiler (JIT)[1]. JIT is integrated into the JVM and it is in charge of translating the Java byte code into binary code. Each method is translated only when it is called for the first time. Thanks to this improvement the execution time decreases and the code is again portable.

Many comparisons between C, C++ and Java were documented in literature. From this point we call "*Java*" the version optimized with JIT and "*interpreted Java*" the original version. In [19] the execution times of C++ and Java are compared. The authors tested the execution of four sorting algorithms, two of $O(n^2)$ complexity (bubble sort and insertion sort) and two of $O\left(n \cdot log\left(n\right)\right)$ (recursive quick sort and heap sort), on four integer data sets of different sizes. The results demonstrated that C++ was much faster than pure interpreted Java (from 11 to 20 times) and only from 1.45 to 2.91 times faster than Java (version 1.3). In [6] a set of polynomial multiplications was computed and executed using the three languages. The results showed that Java completed the operations faster than standard C (mean of 21%) but in average 2.61 times slower than C++. In [7] the executions of the Linkpack benchmark were compared for Java

Paper	Test	OS	Java vs. C	Java vs. C++	JDK	C/C++ compiler
[19]	Sorting alg.	W	—	1.45 - 2.91	Sun 1.3	Borland v. 5.5
[6]	Polynomial mult.	S	0.79	2.61	Sun 1.2b5	Sun Workshop C 4.2
[7]	Linkpack bench.	W-S	2.25	—	Sun 1.2b4	—
[18]	Method call	W-S-L-M	—	1 clock slower	Please refer to the paper	
[12]	Int and float div.	W	—	~ 1	Sun 1.1.5	Visual C++ 5.0

Table 1. Results summary (OS: W=Windows, L=Linux, S=Solaris, M=MacOs)

and standard C. This benchmark was introduced by Jack Dongara and measures how fast a computer solves a dense N-by-N system of linear equations. The results showed that for a 1000 x 1000 system Java was 2.25 times slower than C. In [18] Ruolo evaluated the Java method call performance. Different tests with a different numbers of parameters showed that Java was only one clock cycle slower than C++. The same tests also highlighted that the time needed for allocating user defined objects on the heap was roughly equivalent. However C++ also uses the stack for allocating temporary object and in this case it was from 10 to 12 times faster than Java, which uses only the heap. Interesting conclusions were reported by Mangione [12]. He tested the repetitive execution of simple operations like integers and float divisions and showed that Java was as fast as C++. As summarized in table 1 all the papers report that, since the introduction of the Just-In-Time compiler, Java is only 1.45-2.91 times slower than C++ (Column OS: operating system, columns 4-5 execution time ratios).

Since these studies demonstrated how the execution of simple operations in Java is more or less as fast as in C++, one factor that could influence the total execution time of a Java program is the Garbage Collector (GC). However [10] showed that Java GC is as fast as a *malloc/free* operation in C++. In fact when a program executes a *malloc* operation, the allocator looks for an empty slot of the right size and returns a pointer to a random place in the memory. In Java instead the allocator use the bits of memory adjacent to the last bit it used. Hence it doesn't need to spend time looking for memory. So the amount of time used for the garbage collector is comparable to the amount of time that the allocator uses in C++ for finding free memory slots.

Finally other interesting results are documented in [15]. The same program was implemented by 40 different programmers in different languages (24 in Java, 11 in C++ and 5 in C). The experiment compared not only the performance of the languages but also the differences between the implementations in the same language (interpersonal differences). The results demonstrated that Java was 2 times slower than C++ and that the interpersonal differences were much larger than the average difference between Java and C++. That means a well written Java program could be as efficient as an average C++ program.

3 A performance comparison case Study: the Delaunay Triangulation

Visual sensors such as laser scanners acquire information on the environment geometry in form of a point cloud: a set of vertices in a 3D coordinate system.

Each one of these vertices corresponds to a point on the surface of one of the objects present in the environment. In order to reconstruct the surface of these objects the vertices have to be connected. This problem is called mesh generation and one of the possible solutions consists of the Delaunay triangulation [8].

Delaunay's algorithm connects the set of points in such a way to build a series of triangles which respect the following property: for all the set of points there is no point which lies inside the circumcircle of any triangle. The triangulation result is unique except if more than three vertices stand on the same circumference. In this case more than one solution exist.

In this section a comparison between a C++ and a Java version of the Delaunay triangulation will be reported. We refactored in Java a C++ implementation coming from the OSG libraries[2]. The implementation of this triangulation algorithm is based on the Bowyer-Watson method, which works in the plane space. It iterates all the points of the cloud and for each one executes two main steps: identifying the triangles whose circumcircle contain the current analyzed point and then building a new set of triangles, which respect the Delaunay condition. This algorithm allows to process point clouds in 3D space but realizes only a triangulation in the plain space therefore the Z coordinate is ignored. It should be noted that the implemented algorithm does not provide a constrained Delaunay triangulation. For this reason, during the timing and the comparison of the computation time, we have excluded the constraints also in the OSG version.

Both the OSG and our implementations receive as input the point cloud in form of a collection of vertices. The OSG implementation defines a custom class, *Vec3Array*, which is a specialization of the class *MixinVector* (*MixinVector* allows inheritance to be used in order to easily emulate derivation from *std::vector* but without introducing undefined behaviour through violation of virtual destructor rules [3]). Hence, *Vec3Array* defines a vector of *Vec3* instances, which are triplets of float data types. Our implementation instead uses the Java *ArrayList*. We chose this collection because it is the fastest of all the collections provided by the Java framework for what regards the operations of inserting, iterating and sorting [20], and because its performance are comparable with the one of Java *Vector*. On the other side *ArrayList*, like C++ *std::vector*, is not as well efficient when it has to perform the operation of removing elements in random position. In order to better understand how much the overhead between Java and C++ is due to these data structures, we compared the performance of the *Vec3Array* and *ArrayList* collections. We executed a set of tests on the most used operations during the triangulation algorithm:

– Insertion. We executed 10000 and 100000 insertions of objects (instances of class that represent the 3D points) at the end of the 2 collections. We chose the values of 10000 and 100000 because they are the maximum orders of magnitude of the collection sizes used in the tests of the Delaunay algorithm.
– Removal. We executed the complete clearing of collections of 10000 and 100000 objects. We removed one element at time. In order to evaluate the performance in the worst case, the object at the head of the collection was chosen to be deleted during each iteration.

– Sorting. We invoked the sorting function on collections of 100 and 1000 points generated randomly. We chose these size values, which are lower with respect to the tests of the other operations, because in the Delaunay algorithm the sorting is always executed on little collections (see more details below).

Each test was executed 50 times and then the mean time was computed. We executed them on a 3.2 GHz Intel Pentium 4 processor with 1GB of RAM under Ubuntu 10.4 (OpenJDK Runtime Environment v. 1.6.0_20 and GCC v. 4.3.3). Results are reported in table 2 where times are expressed in milliseconds and regard the execution of all the n operations. *ArrayList* is faster than *Vec3Array* during the insert and the remove operations, whereas it takes much time to compute the sorting because of the used algorithm. Indeed the method for sorting Java collections uses a modified *merge-sort* algorithm [5], which offers guaranteed $O(n \cdot log(n))$ performance. The sorting algorithm provided by the C++ STL library instead uses the *introsort* algorithm whose worst case complexity is $O(n \cdot log(n))$.

	Insert		Remove		Sort	
N. of elements	10000	100000	10000	100000	100	1000
Java	1.30	6.33	46.67	5168.21	0.13	0.42
C++	1.51	11.27	275.82	27799	0.02	0.40
Java vs C++	0.86	0.56	0.17	0.19	6.5	1.05

Table 2. Times report - Collection comparison

We have also analyzed time required for the evaluation of logical conditions. Four tests were executed, taking into account the following logical conditions:

– Simple logical proposition ($var{=}{=}true$)
– Disequation ($a < b$). (Most evaluated condition in the case study, see eq. 1)
– Logical disjunction of two disequations ($(a < b)||(a > c)$)
– Logical conjunction of two disequations ($(a > b)\&\&(a < c)$)

Each evaluation was executed 10000 times and each test was repeated 50 times. Table 3 reports average times of the tests in milliseconds. We used a boolean variable (initialized false and its value was changed each execution ($var = !var$)) in the first test and float variables (initialized with a constant values) in the others. As can be seen, Java is always faster than C++, except for what regards the evaluation of simple logical proposition.

3.1 The implementation details

The two implementations compute the triangulation according to the same steps, which are described in the following list.

1. Initialization. The Initialization step consists of the setting up and the sorting of the input point cloud according to their coordinates. Then four new

	Prop.	Diseq.	Disj.	Conj.
N. of elements	10000	10000	10000	10000
Java	0.187	0.084	0.103	0.093
C++	0.039	0.262	0.452	0.290
Java vs C++	4.79	0.32	0.23	0.32

Table 3. Times report - Logical conditions evaluation comparison

points are inserted in order to surround the plain point cloud. These four points are used to build two main triangles (super-triangles), such that the plain point cloud lies inside their area. These triangles are stored in a collection, that we'll call *trianglesList*. The collection data structure was chosen accordingly to the operations that occur more often, indeed the *trianglesList* is subject to several iterations, insertions and removal. As shown in [20], *ArrayList* is the list of all the available lists in the Java framework that perform insertion, iteration and random access in the fastest way. Although removing objects from *ArrayList* requires a long time, insertions and iterations occur more often than remove operations; hence we decided to use *ArrayList* for implementing the *trianglesList*.

2. Iteration. During the iteration, each point is considered and is compared to the triangles contained in the *trianglesList*. First the condition 1 is checked ("*point*" stays for the current point and "*tri.circ* for the circumcircle of the current triangle).

$$point.X - tri.circ.X > tri.circ.radius \tag{1}$$

   – If it is true, the current triangle is removed from the *triangleList* and will not be more considered because the current point and also the following ones surely don't lie in the circumcircle of the current triangle (i.e. the triangle respects the Delaunay condition for all the points and it is part of the final mesh). This is guaranteed by the initial ordering of the points.
   – Otherwise, we must further investigate if the current point effectively lies in the circumcircle of the current triangle. In case it is true the Delaunay condition is not respected. Therefore the edges of the triangle are added to a specific *ArrayList* (called *edgeSet*) and the current triangle is deleted. Whereas, if the Delaunay condition is respected, the next triangle is considered. It has to be noted that the *edgeSet* collection has been implemented as an *ArrayList* because it is sorted many times during the triangulation algorithm. Hence, the usage of *Collections.sort* method and *ArrayList* is the Java solution that allows us to save time and increase performance in the best way. In our tests the maximum size of the *edgeSet* collection was never greater than 100.

When the whole *trianglesList* has been scanned, new triangles are constructed from the *edgeSet* collection and added to the *triangleList* (in our tests the maximum order of magnitude of this collection size is 10000). Note that if an edge is shared between two triangles that contain a point, then

the edge is not considered. The iteration proceeds until all points have been analyzed, except for the four points created during the initialization .
3. Completion. The four points introduced during the initialization step and triangles having vertices in common with these four points are deleted. If there are degenerate triangles (circumcircle radius equals to 0) they are eliminated too. Finally a return result is built in form of a Mesh.

Since the order of magnitude of the size of the list on which we perform more insertion is 10000 whereas the one of the collection on which we perform the sorting is 100 the time gained in Java for populating the first collection is lost for sorting the second one (see table 2). This suggests that the time spent for managing collections will be more or less the same for both the Java and C++ implementations. The evaluation of logical conditions instead seems to be not important from a performance point of view. In fact the time spent for evaluating conditions on 10000 iterations is much lower than the time spent for populating collections in more or less 100 iterations.

3.2 The Java HotSpot compilers

The current JVM offer a technology called HotSpot Compiler [13], which works better and faster than the pure JIT compiler. Rather than compiling each method at the first execution, the HotSpot runs the program using an interpreter for a while. During this time, in order to detect the most used and critical methods, the execution is analyzed. The collected information is then used to perform more intelligent optimizations and only the critical methods are actually compiled. This technique is called "*Adaptive Optimization*". It doesn't only produce better performance but it also reduces the overall compilation time. The adaptive optimization is continuously performed so that it adapts the performance to the users' needs.

The Java Platform Standard Edition offers a JVM that comes with two compilers: the Client and the Server versions[1]. The Client compiler is the default one and it has been specially tuned to reduce the start-up time. It is designed for client environment, in particular for applications where there is not the need of continuous computation, for example a GUI. The Server compiler instead is designed for long-running server applications, where the operating speed is more important than the start-up time. This compiler offers an advanced adaptive optimizer and supports many of the optimizations offered by the C++ compilers.

Subsections 3.3 and 3.4 report the tests executed using the client compiler whereas the server compiler is used in the experiments of subsection 3.5.

3.3 Performance analysis

We executed the algorithm on five point clouds of different sizes: a semi-sphere, a floor, the Oxford Bunny and 2 terrains. Each point cloud was processed 50

[1] Users can specify the compiler by means of the options "-client" and "-server". The tests on the collections and logical conditions were executed with the client compiler

times and the execution time was measured; then the average, the standard deviation and the confidence intervals ($1 - \alpha = 0.95$) were computed. In the first experiment each triangulation corresponds to a single program invocation, so we executed the program 50 times per point cloud.

Table 4 reports the results of our test on the same PC presented before running Windows XP (Sun Java 6 v. 1.6.0_23 and C++ programs compiled with MinGw v. 3.82 and GCC v. 4.5.0). Mean time, standard deviation and confidence intervals (c_1 and c_2) are expressed in milliseconds. Java vs. C++ is the ratio between the average execution time.

		Sphere	Floor	Bunny	Terrain 1	Terrain 2
Number of vertices		642	10000	35947	66049	263169
Java	Mean	70.62	1458.8	3102.2	20344	132926
	Std Dev.	7.89	12.90	52.96	403.44	986.18
	c_1	68.38	1455.1	3087.1	20229	132646
	c_2	72.86	1462.5	3117.3	20459	133206
C++	Mean	7.50	298.13	886.25	3305.9	22908
	Std Dev	7.89	25.61	63.77	144.83	227.06
	c_1	5.26	290.85	868.13	3264.8	22844
	c_2	9.74	305.40	904.37	3347.1	22973
Java vs. C++		9.42	4.89	3.50	6.15	5.80

Table 4. Times report - Multiple invocation - Windows - Java Client

Referring to the table 4, the triangulation execution time obtained with the first point cloud (sphere) is not very truthful. Indeed Java version is 9.42 times slower than C++ and this value doesn't fit the ratios obtained with the other point clouds. In this case the execution time is very small and so the time required for compiling the code greatly influences the result. Note that the costs required by the compiler have a fixed part, which is the same for each point clouds. Hence the smaller is the point cloud, the greater is the influence of the compilation overhead on the execution time.

3.4 Single program invocation

In order to avoid the compilation overhead we decided to change our measuring strategy. We set up a second experiment, where 51 triangulations were measured in a single program invocation. This kind of experiment corresponds to a long run execution, where only the first invocation of the algorithm pays the compiler costs. We discarded the execution time of the first invocation and computed our statistics on the other 50 samples. Table 5 reports the results obtained under Windows and Ubuntu Lucid (both with the same Java and C++ versions presented before). As expected, the average execution times decrease for Java and remain more or less the same for C++.

Under Windows, without the compiler overhead, Java performance are always better than the results reported in table 4, but remain worse than the results

		Windows					Linux				
		Sphere	Floor	Bunny	Terr. 1	Terr. 2	Sphere	Floor	Bunny	Terr. 1	Terr. 2
Num. of vertices		642	10000	35947	66049	263169	642	10000	35947	66049	263169
Java	Mean	15.58	1157.2	2487.2	18108	115977	6.66	869.20	2111.3	13478	92301
	Std Dev.	0.50	18.82	20.57	80.53	399.65	1.48	40.83	80.94	106.06	383.57
	c_1	15.44	1151.9	2481.3	18085	115863	6.24	857.60	2088.3	13448	92191
	c_2	15.72	1162.6	2493.0	18130	116091	7.08	880.80	2134.3	13508	92410
C++	Mean	6.56	288.75	914.06	3226	23168	3.23	224.01	750.09	3333.4	24277
	Std Dev	7.79	7.89	11.92	110.36	605.89	0.84	2.35	7.41	62.90	437.73
	c_1	4.35	286.51	910.68	3195	22996	2.99	223.35	747.99	3315.5	24152
	c_2	8.78	290.99	917.45	3257	23340	3.47	224.68	752.20	3351.2	24401
Java vs. C++		2.37	4.01	2.72	5.61	5.01	2.06	3.88	2.81	4.04	3.80

Table 5. Times report - Single invocation - Java Client

discussed in section 2. Indeed in our tests the execution time ratio between Java and C++ goes from 2.37 to 5.61 against the range 1.45-2.91 reported in table 1. One possible reason is that the triangulation process requires an intensive use of the memory and probably the C++ version leverages the possibility of store temporary objects on the stacks, which is much faster [18].

The table shows that under Ubuntu Java is more efficient than under Windows. Indeed the ratio range goes from 2.06 to 4.04. It should be noted that we executed the tests with both the OpenJDK and the Sun JDK. However in the paper we consider only the first one because the results are almost the same.

Another consideration can be done on the relation between the point clouds sizes and the execution time ratios. The value of the performance ratio doesn't show a linear trending, hence we can assert that for this algorithm there is no correlation between the performance ratio and the input size.

3.5 JVM Server option

Table 6 report the results obtained using the Server compiler. We executed the tests on the same machine used previously with the same configurations. Of course, only the Java version was tested and the C++ rows report the results of the previous experiments. Despite we are aware of the existence of the optimizations provided by GCC, we didn't work on them. Indeed the default *release* configuration of the OSG libraries is tuned in order to offer the best performance.

The results shows that under Windows the server compiler considerably reduces the average execution time of complex operations. In particular the execution times decrease 3-4 times with respect to the client compiler. The first cloud represents the unique exception. Indeed, in that case the execution is too short and so most of the iterations are executed without optimization. This is the typical case in which the larger start-up time required by the server compiler is not compensated. Regarding the other point clouds, the ratio range goes from 1.09 to 1.51 and so Java is nearly equivalent to C++.

The results demonstrates that also under Ubuntu the server compiler significantly improves the performance. The same considerations reported above could

		Windows					Linux				
		Sphere	Floor	Bunny	Terr. 1	Terr. 2	Sphere	Floor	Bunny	Terr. 1	Terr. 2
Number of vertices		642	10000	35947	66049	263169	642	10000	35947	66049	263169
Java	Mean	24.96	356.26	999.02	4873.1	30320	20.40	428.5	1253.1	4778.9	29368
	Std Dev.	15.45	20.16	20.40	42.80	230.28	19.58	20.66	56.64	172.29	250.81
	c_1	20.57	350.53	993.22	4861.0	30255	14.84	422.63	1237.0	4729.9	29297
	c_2	29.35	361.99	1004.8	4885.3	30385	25.96	434.37	1269.2	4827.8	29440
C++	Mean	6.56	288.75	914.06	3225.9	23168	3.23	224.01	750.09	3333.4	24277
Java vs. C++		3.80	1.23	1.09	1.51	1.31	6.32	1.91	1.67	1.43	1.21

Table 6. Times report - Single invocation - Java Server

be applied to the results obtained with the first cloud. Referring to the other clouds the ratio range goes from 1.21 to 1.91 and the average execution times are from 2 to 3 times better than the results obtained with the client compiler.

4 Conclusions and future works

In this paper we have described our work on the evaluation of the performance of Java with respect to C++ in robotics applications. The results obtained with the Client compiler, which works better for short-running applications, have shown that Java is from 2.72 to 5.61 times slower than C++. Using the Server compiler, which is best tuned for long-running applications, have instead demonstrated that Java is from 1.09 to 1.91 times slower. These results show that the performance of Java are now better with respect to the tests previously documented in literature and demonstrate that the use of the Server compiler for long run application greatly reduces the execution time.

In addition to the fact that now the performance are not so different with respect to C++, we also have to consider that Java offers a set of interesting features. *Portability*: Java is designed to be platform independent and so Java software is very portable. The low level data types such as integer and float are fully defined in Java specification and aren't platform dependent. *Reusability*: Java comes by default with a lot of common libraries for several purposes. It is also easy to deploy and reuse the developed libraries without sharing source or header files or requiring a specific compiler. *Maintainability*: Java is designed to forbid common bugs such dangling pointers, casting errors, out-of-bounds array, stack overflows, segmentation faults and uninitialized variables.

In conclusion, the results obtained with the server compiler and these important features suggest that Java can be considered a valid alternative to C++. We plan to executes new experiments in order to further confirm this thesis. The tests will regards the communication with external devices (USB, RS-232, . . .) and the execution of multi-thread programs.

5 ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. FP7-ICT-231940-BRICS (Best Practice in Robotics). The authors would like to thank all the partners of the project for their valuable comments.

References

1. Just in time compiler. `http://en.wikipedia.org/wiki/JIT_compiler`.
2. Open Scene Graph. `http://www.openscenegraph.org`.
3. Open Scene Graph API reference. `http://www.openscenegraph.org/documentation/OpenSceneGraphReferenceDocs`.
4. Rosjava - An implementation of ROS in pure Java with Android support. `http://code.google.com/p/rosjava/`.
5. Specifications of java.util.collections. `http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html`.
6. L. Bernardin, B. Char, and E. Kaltofen. Symbolic computation in Java: an appraisement. In *Proceedings of the 1999 Int. symposium on Symbolic and algebraic computation*, pages 237–244. ACM, 1999.
7. J. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. A methodology for benchmarking Java Grande applications. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 81–88. ACM, 1999.
8. B. Delaunay. Sur la sphere vide. *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk*, 7:793–800, 1934.
9. A. Elnagar and L. Lulu. A global path planning Java-based system for autonomous mobile robots. *Science of Computer Programming*, 53(1):107–122, 2004.
10. J. Lewis and U. Neumann. Performance of Java versus C++. *Computer Graphics and Immersive Technology Lab, University of Southern California, Jan*, 2003.
11. M. Long, A. Gage, R. Murphy, and K. Valavanis. Application of the distributed field robot architecture to a simulated demining task. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE Int. Conference on*. IEEE, 2005.
12. C. Mangione. Performance tests show java as fast as c++. *JavaWorld*, 1998.
13. S. Meloan. The Java HotSpot (tm) Perfomance Engine: An In-Depth Look. *Article on Suns Java Developer Connection site*, 1999.
14. F. Monteiro, P. Rocha, P. Menezes, A. Silva, and J. Dias. Teleoperating a mobile robot. A solution based on JAVA language. In *Industrial Electronics, 1997. ISIE'97., Proceedings of the IEEE Int. Symposium on*, volume 1. IEEE, 2002.
15. L. Prechelt et al. Comparing Java vs. C/C++ efficiency differences to interpersonal differences. *Communications of the ACM*, 42(10):109–112, 1999.
16. F. Raimondi, L. Ciancimino, and M. Melluso. Real-time remote control of a robot manipulator using java and client-server architecture. In *Proceedings of the 7th Int. Conference on Automatic Control, Modeling and Simulation*, 2005.
17. S. Robertz, R. Henriksson, K. Nilsson, A. Blomdell, and I. Tarasov. Using real-time Java for industrial robot control. In *Proceedings of the 5th Int. workshop on Java technologies for real-time and embedded systems*, pages 104–110. ACM, 2007.
18. M. Roulo. Accelerate your Java apps. *Java World*, 1998.
19. S. Spw, S. Wentworth, and D. Langan. Performance evaluation: Java vs c++. In *39th Annual ACM Southeast Regional Conference*. Citeseer, March 16-17 2001.
20. S. Wilson and J. Kesselman. *JavaTM Platform Performance - Chapter 8*. Sun Microsystems, 2001. `http://java.sun.com/docs/books/performance/1st_edition/html/JPAlgorithms.fm.html`.