Computational Performance of Java and C++ in Processing fMRI Datasets

Rodrigo Vivanco
Institute for Biodiagnostics
National Research Council Canada
435 Ellice Avenue, Winnipeg MB R3B 1Y6
Rodrigo.Vivanco@nrc.ca

Nicolino Pizzi
Institute for Biodiagnostics
National Research Council Canada
435 Ellice Avenue, Winnipeg MB R3B 1Y6
Nicolino.Pizzi@nrc.ca

ABSTRACT

Software systems for the analysis of image-based biomedical data, such as functional magnetic resonance imaging (fMRI), require a flexible data model, fast computational techniques and a graphical user interface. Object-oriented programming languages, such as Java and C++ facilitate software reuse and maintainability, and provide the foundations for the development of complex data analysis applications. This paper explores the advantages and disadvantages of using these two programming environments for scientific computation.

Keywords: Performance, fMRI data analysis, Java, C++.

1. INTRODUCTION

Java and C++ have advantages and disadvantages for the development of data analysis systems. C++ is well suited for computational and memory optimization while Java supports cross-platform development and has a rich set of reusable classes. For this study, the same data model and computational algorithms were implemented in C++ and Java. A graphical user interface was developed for Java but not for C++ since a different API would have to be used for every test platform. Comparisons were made in respect to computational performance and ease of development.

2. FMRI DATA MODEL

FMRI data consists of a set of images captured every few seconds. Over time, the intensity level of a given voxel will change if it is involved in the neural processing of a task. Analysis is performed along the time dimension of the data. For each voxel of interest a data vector is obtained by sampling each image at a given (x,y,z) coordinate. Since most of the data access is vector wise, Figure 1 shows how each voxel time course is stored contiguously in memory.

OOPSLA '02, November 4-8, 2002, Seattle, Washington, USA.
ACM 1-58113-626-9/02/0011.

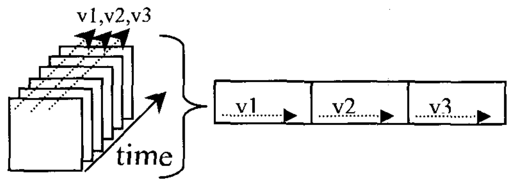

Figure 1. Contiguous data vectors in one large array.

3. IMPLEMENTATION

Procedural programming is similar in Java and C++ with respects to primitive types and looping constructs. The major differences that affect design and implementation are: C++ allows direct memory access (pointers), supports generic programming (templates) and multiple inheritance; Java is fully OO, everything must be encapsulated in a class, all objects are dynamically allocated in the heap, arrays are objects with run-time bounds checking and Java has automatic garbage collection.

4.1 C++

A generic data model using templates was implemented. Despite the difficulties in coding and debugging, templates are a trade-off that result in more maintainable code. Novice programmers found templates difficult to learn and use effectively. Also, templates are not yet fully supported by all C++ compilers. Pointer arithmetic was not used as extensively as first expected since the cost of incrementing loop counters and dereferencing pointers did not justify the added syntax complexity over using the [ ] operator.

As expected, it was not unusual to find memory leaks where the programmer forgot to **delete** a dynamic array or object. There are tools that assist in finding memory leaks but even then, it is still possible to make basic memory management mistakes. C++ has tried to address this problem by using automatic pointers, however, they only apply to local scope. It is recommended that developers implement their own "smart" reference counting pointer wrapper class to handle memory management.

4.2 Java

Compared to C++, Java is an easy object-oriented language to learn and use, the syntax is simple and consistent and novice programmers are more proficient and less likely to make programming errors. Java was designed as a safe, platform-neutral programming environment and does not allow direct memory access using pointers and has a performance penalty for bound checks.

Java compilers translate source code into Java machine byte code. A platform specific interpreter, known as a Java Virtual Machine (JVM), executes the Java byte code. Interpreted Java applications are inherently slower than C++ programs. Java minimizes memory leaks by providing garbage collection. However, the actual garbage collection mechanism is left up to the JVM implementation. It is still possible for objects to be remain in memory past their actual useful lifetime, for example, a static variable not set to **null** when it no longer needs an object. Garbage collection does not guarantee bug free programming.

4.3 Java Native Interface

Java provides the means to link a Java application to natively compiled C/C++ code via the Java Native Interface (JNI). Performance enhancement was a common motivation for using JNI. In scientific computing a new rationale is to incorporate legacy code. JNI adds another level of coding complexity resulting in code that is harder to understand, debug and maintain. With JNI Java's platform independence is compromised and its built-in safety measures can be circumvented.

5. BENCHMARKS

Where possible, the exact same procedural code was used for both Java and C++. All benchmarks were executed on a Pentium III 700 MHz processor, with 512MB of memory running under Linux Red Hat 7.2. Two C++ compilers were used, the Portland Group compiler geared towards distributed and parallel applications (pgCC), and the GNU compiler (gcc). Four Java compilers and JVMs were tested. IBM's Java kit 1.3, Sun's JDK 1.2, 1.3 and 1.4 beta. IBM 1.3, JDK 1.3 and 1.4 are Just-In-Time (JIT) interpreters. Java bytecode is interpreted and also compiled as the application runs. Sun's JDK 1.3 and 1.4 uses HotSpot which tries to optimize the Java program as it is running. All code was compiled with the highest optimization levels available.

5.2 SciMark 2

SciMark 2 is a composite benchmark developed by the National Institute of Standards and Technology that tests floating point operations of 5 computational kernels. Similar source code is provided for Java and ANSII C. With C++ the small test suite ran at 140 Mflops, while with Java it ranged from 120 MFlops (IBM 1.3) to 22 MFlops (JDK 1.2). For the large test suite C++ executed at 50 MFlops and Java between 37 and 10 MFlops.

5.3 Algorithms

Two algorithms were tested to simulate fMRI data analysis, correlation and fuzzy clustering. Correlation on 262,144 data vectors, each with 70 elements, showed that C++ was 3 times faster than Java. However, for Java to process 35 MB of data under 1 second is not considered slow for fMRI analysis.

Fuzzy clustering was tested with 32,768 data vectors, each with 42 elements. The IBM 1.3 JVM runs as fast as native C++ for this particular algorithm at 29 seconds. Tests indicated that calculating the power function was the bottle neck. Many Java API methods are implemented in native code for performance reasons and it appears that IBM's implementation of the Math power function is faster than the pgCC math library.

6. DISCUSSION

An important aspect of scientific computing is performance. Many factors affect application execution speed, ranging from platform hardware, operating system, algorithms and the programming language used. The most elegant and flexible approach for improving scientific computation performance is to use efficient algorithms and coding methods.

When Java was first introduced in 1995, JVM technology was very slow when compared to compiled C++ programs, mostly due to the fact that Java programs were fully interpreted. The benchmarks have shown that Java technology has vastly improved performance in the last few years.

JNI is a viable option for improving performance but at a cost to application development. Maintaining JNI code is difficult and time consuming, often resulting in portability problems. If many legacy algorithms are available in C/C++, it would be advisable to develop the entire application in C++ instead of Java with JNI.

Java and C++ are object- oriented programming languages that promote modular, maintainable code. Java is easier to use than C++ leading to more robust code and less development time. However, tests show that C++ generally outperforms Java. With the advent of JIT virtual machines, Java performance can at times be at par with C++. However, for raw computational performance Java will be at a disadvantage if programs have to be executed by an interpreter.

7. ACKNOWLEDGEMENTS

The Natural Sciences and Engineering Council (NSERC) is gratefully acknowledged for its support of this investigation.

8. REFERENCES

[1] G. Phipps, "Comparing observed bug and productivity rates for Java and C++," *Software — Practice and Experience*, vol. 29, pp. 345–358, 1999.
[2] SciMark 2.0, http://math.nist.gov/scimark2/
[3] Stroustrup B., *The C++ programming language*, Reading, Mass.: Addison-Wesley, 1997.
[4] Meyers S, *Effective C++, 50 Specific Ways to Improve Your Programs and Designs*, Reading, Mass.: Addison-Wesley, 1992.
[5] Gosling J, Joy B, Steele G, *The Java Language Specification*. Reading, Mass.: Addison-Wesley, 1996
[6] Wilson S, Kesselman J, *Java Platform Performance, Strategies and Tactics*. Boston: Addison-Wesley, 2000