Combining Scala with C++ for Efficient Scientific Computation in the Context of ScalaLab

Stergios Papadimitriou, Lefteris Moussiades

***Abstract*—ScalaLab is a MATLAB-like environment for the Java Virtual Machine (JVM). ScalaLab is based on the Scala programming language. It utilizes an extensive set of Java and Scala scientific libraries and also has access to many native C/C++ scientific libraries by using mainly the Java Native Interface (JNI). The performance of the JVM platform is continuously improved at a fast pace. Today JVM can effectively support demanding high-performance computing and scales well on multicore platforms. However, sometimes optimized native C/C++ code can yield even better performance. That code can exploit the peculiarities of the hardware architecture and of special parallel hardware, as for example Graphical Processing Units. The present work reports some of the experience that we gained with experiments with both JITed JVM code and native code. We compare some aspects of Scala and C++ that concern the requirements of scientific computing. The article describes how ScalaLab tries to combine the best features of the Java Virtual Machine with those of the C/C++ technology, in order to implement an effective scientific computing environment.**

***Index Terms*—Java, scientific computation, JIT Compilation, scripting, numerical performance**

I. Introduction

The recently introduced ScalaLab [5] scientific programming environment for the Java Virtual Machine (JVM) leverages the statically-typed Scala object-oriented/functional language [3]. It provides MATLAB-like syntax that is used to construct scripts that are then compiled by ScalaLab for execution on the JVM. The Scala language supports the implementation of simple, coherent and efficient MATLAB-like interfaces for many Java scientific libraries. These interfaces are compiled within the core of ScalaLab. ScalaLab is an open-source project and can be obtained from http://code.google.com/p/scalalab/ . It can be installed easily. The only prerequisite is the installation of the Java 8 (or newer) runtime (which is free). We supply scripts for launching these systems for all the major platforms. The general high-level architecture of ScalaLab is described in [5]. The JVM provides a flexible environemnt for the implementation of a MATLAB like user friendly scientific programming environment. Also, Scala provides both the powerful Scala interpreter that leads to a scripting execution and elaborate mechanisms for building new syntax. However, scientific computation demands high performance. This is the main reason that Fortran is still a popular scientific programming language. The recent advance in Just-In-Time (JIT) compilation allows execution speed of bytecodes near or sometimes even better than native code.

But, sometimes the flexibility offered by C/C++ to utilize low-level features for optimization and to access specialized hardware is important. The paper supports the claim that the flexibility of the Scala programming language combined with the robustness of the Java Virtual Machine, are sufficient for building on top of them a capable scientific programming environment. However, for some basic numerical analysis operations that tend to be computationally demanding, as for example the Singular Valued Decomposition (SVD) and eigenvalue computations, native C / C++ libraries can offer a notable speedup. The native library that we most exploit, the GNU Scientific Library is a portable and efficient C scientific library. We can access its functionality by using a JNI based interface.

The paper proceeds as follows: Section II demonstrates how some advanced characteristics of ScalaLab are implemented on top of the corresponding features of the Scala language. Section III discusses some aspects of JVM vs C/C++ native code issues. Section IV summarizes on how we exploited the javacpp open source project in order to interface the GNU Scientific Library (GSL) in ScalaLab. Section V elaborates on some performance related issues. Finally, we conclude our paper and presents some directions for future work.

II. Advanced characteristics of the ScalaLab environment

This section demonstrates how some advanced characteristics of the ScalaLab are implemented on top of thecorresponding features of the Scala language.

Global function workspace

Scientific programming environments demand a global namespace of functions. Scala does not have the concept of globally visible methods; every method must be contained in an object or a class. A global function namespace, can be implemented easily with static imports. In Scala objects are imported since these objects encapsulate the static imports. Therefore, the automatic import of static methods provides the appearance of the existence of global methods. For example, the plot method appears to be available globally since we import it from the object *scalaSci.plot.plot*. Scala also offers the ability to define apply methods for the companion objects of classes. If a class implements the apply method, an instance of the class can essentially be 'executed' like a function as the instance name followed by a list of arguments in parentheses. When the apply method is implemented by a class, a method doesn't need to be imported into the 'global' namespace, it is only necessary to import the class itself.

The Scala Interpreter

Manuscript received Dec 05, 2015; revised Jan 11, 2016.

Stergios Papadimitriou and Lefteris Moussiades are with the Technology Education Institute of East Macedonia and Thraki, Dept of Informatics and Computer Engineering, Agios Loukas, 65404 Kavala,GREECE, emails: sterg@teiemt.gr, lmous@teiemt.gr

ISBN: 978-988-19253-0-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

An essential component of Scala's scripting framework is the Scala interpreter. This is an essential component for a scientific programming environment. It is based n compiled scripting and thus scripts run at full speed. Since compiled C/C++ programs are executed as operating system processes, there is not an easy way to keep computed variables from an executed C program for later runs.

The overall approach of the Scala interpreter is based on initially compiling the requested code. A Java classloader and Java reflection are then used to run the compiled bytecode and access its results.

The scheme implemented in the Scala interpreter extracts the whole visible (i.e. context of public variables) computation state as the interpreter's context. ScalaLab binds both data and code objects automatically to this context. ScalaLab runs Java bytecode, that can be manipulated as data variables, i.e. we can store a code object that implements a function to a variable. It is clearly much more difficult to handle natively compiled chunks of C/C++ code. Although this is more powerful it imposes difficulties in retrieving the previous context when we create a new Scala interpreter. In that case it is necessary to replay the commands in order to restore the environment. However, the restoration of the user environment in ScalaLab is performed fastly; the user does not notice the delay of the few Scala compiler runs. Restoration of the computation context also is a somewhat rare operation.

A single compiler instance is used in the Scala's interpreter to accumulate all successfully interpreted Scala code. The interpretation of a piece of code is performed by generating a new object that includes that piece of code and has public members to export all variables defined by that code. To accommodate user expressions that are constructed from variables or methods defined in previous statements, import statements are used.

It becomes evident that an effective approach for detecting the variables that a piece of code defines is required, in order to import their values at the global workspace. The Scala interpreter utilizes the Scala parser to accomplish the non-trivial task of analyzing variable visibility. The Scala parser is also utilized, in order to detect which variables are used by a code snippet. The values of these variables are then requested from previously executed code. Such type of retrieval seems very difficult or even impossible for natively executed C/C++ code. It is unnecessary to request variables from the interpreter that do not appear in a new code snippet, since these values will not change. For example, if prevVar = 5.5, is not used at our new script, the interpreter does not request the value of prevVar.

The Scala interpreter keeps a list of names of identifiers bound with a var or val declaration (boundVarNames) and also a list of function identifiers defined with a def declaration (boundDefNames). In addition we can retrieve the last source code fragment of the program that has been interpreted (lastLineInterpreted).

We keep a symbol table of the current ScalaSci bound variables. This task is used to graphically display the current work context to the user. We have to keep this external table synchronized with the internal variables binding that the interpreter keeps (i.e. boundVarValNames). The current value of each variable is retrieved from the interpreter by issuing a simple command with the name of the variable.

If an identifier is being redefined as a function then it is removed from the variable binding. This is necessary since the namespace of variables and functions in Scala is common [4] and thus the identifier is hidden by the function definition.

III. Discussion of JVM vs C/C++ native code issues

Recently, the gap between Java and C++ performance has been narrowed. Many author studies [10], [11], [12], [13], [14] have shown that Java can achieve similar performance to natively compiled languages such as C++. Taboada et al. [11] claims that with exploiting the Just-in-Time (JIT) compiler Java obtains native performance from Java bytecode. The JIT (Just-In- Time) compiler can significantly speed up the execution of Java applications. The JIT, which is an integral part of the JVM takes the bytecodes and compiles them into native code before execution. Since Java is a dynamic language, the JIT compiles methods on a method-by-method basis just before they are called. If the same method is called many times or if the method contains loops with many repetitions the effect of re-execution of the native code can dramatically change the performance of Java code [13].

An important advantage of JIT, is that the compilation can be optimized to the targeted CPU and the operating system model where the application runs. JIT compilers are able to collect statistics about how code actually runs in the environment it is in, and they can rearrange and recompile for optimum performance.

Many researchers argue that the main reasons JIT optimization are less effective than C++ are:

1) Java is dynamically safe; it ensures programs do not violate the semantics or allow direct access to untrusted memory. Dynamic type tests must be frequently performed.
2) The Java language allocates all objects on the heap, in contrast to C++, where many objects are stack allocated. This means that object allocation rates are much higher for the Java language than for C++.
3) In the Java language, most method invocations are virtual, and are more frequently used than in C++.
4) Java technology-based programs can change on the fly due to the ability to perform dynamic loading of classes. Oracle develops the Java Hotspot compiler that combines interpreted evaluation with adaptive JITing of the hot spots. Then it focuses the attention of a global native code optimizer on the hotspot. This allows avoiding compilation of infrequently used code. This hot spot monitoring is continued dynamically as the program runs, so that it literally adapts its performance on the fly to the users needs.

The main compiler optimizations according to Oracle are: deep inlining and inlining of potentially virtual calls, fast instanceof/checkcast, range check elimination, loop unrolling, feedback-directed optimizations the Java virtual machine profiles the program execution in the interpreter before compiling the Java bytecode to optimized machine code and the profiling information is used later by the compiler to more aggressively and optimistically optimize the code in certain situations. This allows to Java applications to run in at similar or greater speed than C++ programs.

ISBN: 978-988-19253-0-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

IV. The Java - C++ interfacing framework of the javacpp project

ScalaLab uses extensively the JavaCPP project that provides efficient access to native C++ inside Java. It is a framework developed by an open source project, https://github.com/bytedeco/javacpp

JavaCPP maps naturally and efficiently many common features afforded by the C++ language and often considered problematic. Some of such features are: overloaded operators, class and function templates, callbacks through function pointers, function objects (i.e. functors), virtual functions and member function pointers, nested struct definitions, variable length arguments, nested namespaces, large data structures containing arbitrary cycles, virtual and multiple inheritance, anonymous unions, bit fields, exceptions and destructors with garbage collection.

JavaCP consists of a parser that parses the C/C++ code and constructs automatically the appropriate Java classes and C/C++ interfacing code. Finally, the code can be compiled with the javacpp Java library. The final result is a Java class that provides a large fraction of the functionality of the native library, to the JVM accessible by means of the JNI interface. To implement native methods JavaCPP generates appropriate code for JNI, and passes it to the C++ compiler to build a native library.

Native libraries can be constructed with JavaCPP in three steps:

1) the first step is to compile our Java class that uses native methods with javac (as usually), using also javacpp.jar on our classpath.
2) the next step invokes as a Java program the functionality that JavaCPP provides for automatically building the appropriate native library.
3) finally, we can run the Java application with the native library support as usually, with the only additional requirement of placing javacpp.jar at the class-path. We exploited the javacpp framework in order to interface the GNU scientific library (GSL) routines with ScalaLab. We present some examples of their use in a Web wiki:
http://code.google.com/p/scalalab/wiki/GNUScientificLibraryInScalaLab

Although the interface of the GSL routines is not as straightforward as generally is for the pure Java libraries, some basic numerical routines as Singular Valued Decomposition (SVD) and eigenvalue computations perform significantly faster for large matrices. For example we can obtain a speedup of 2 to 3 times for matrices of sizes of more than a million elements. Therefore, we can afford some more complexity in order to gain that notable computational speed improvement.

V. Performance

The Scala language is statically-typed and therefore Scala code can theoretically be compiled to bytecode that runs as fast as Java, sometimes a bit faster sometimes a bit slower, depending on the situation. However, for the advanced features of Scala such as pattern matching, trait inheritance and type parameters, it is difficult to optimize their compilation. The Scala language developers concentrate on these issues and improve the performance of the Scala compiler with each new version of the language.

We would note that the performance of the recent version of MATLAB (2012b) has been impressively improved, while SciLab has been improved also but not so much.

Benchmarking

In order to access the performance of the C++ and Scala platforms, a variety of mathematical computation algorithms will be examined. These will include matrix computations, Fast Fourier Transforms (FFT), eigen decomposition of a matrix and singular value decomposition of a matrix.

Fast Fourier Transform benchmark

The Fast Fourier Transform (FFT) benchmark is performed in ScalaLab using implementations of the FFT from various libraries. Of these libraries, the Oregon DSP library provides the best performance. Close in performance to this library is the JTransforms (https://sites.google.com/site/piotrwendykier/software/jtransforms) library. Since JTransforms is multithreaded, it will accordingly perform more efficiently with more robust machines (e.g. having 8 or 32 cores, instead of only 4). The tutorial FFT implementation of the classic Numerical Recipes book [1] (with the C/C++ code translated to Java) was also observed to achieve reasonable performance in ScalaLab. Interestingly, it was observed that the Oregon DSP and JTransforms FFT routines are nearly as fast as the optimized built-in FFT of MATLAB. We should note that the reported differences in benchmarks are stable, e.g. the relative differences are about the same on different computers, and individual runs show small variations at the results. Contributing to the small variability is that we perform explicitly garbage collection before any benchmark run.

Native code optimizations

In order to test the JVM performance vs native code performance, an implementation of SVD is used [see http://code.google.com/p/scalalab/wiki/ScalaLabVsNativeC] Both the Microsoft's cl compiler of Visual Studio on Windows 8 64-bit and the gcc compiler running on Linux 64-bit were used. ScalaLab is based on the Java runtime version: 1.7.0_25 and Scala 2.11 M7. It has been observed that ScalaLab performs better than unoptimized C and are even close to optimized C code when performing matrix calculations. Table 1 shows some results.

VI. Conclusions

The paper has discussed some aspects of scientific computing in the Scala programming language, a modern object-functional language for the Java Virtual Machine. Scala is exploited with the ScalaLab environment that presents to the user a MATLAB like user-friendly environment for performing scientific explorations. The flexible syntax of Scala, its direct and elegant interface with the plenty high-quality Java scientific libraries and its capable scripting interpreter form the basis for a powerful scientific environment.

Also, the speed of the Scala language with a statically typed design and an advanced and elaborate compiler com-

ISBN: 978-988-19253-0-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

TABLE I
SVDPERFORMANCE: JAVA VS NATIVE CODE

Matrix Size	Optimized C	ScalaLab	Unoptimized C
200X200	0.08	0.15	0.34
200X300	0.17	0.2	0.61
300X300	0.34	0.58	1.23
500X600	3.75	5.06	8.13
900X1000	35.4	51.3	53.3

pares well to the speed of bytecode compiled with the javac Java compiler, sometimes a little slower sometimes a bit faster. We discussed some of the reasons that Java bytecodes today compete the speed of native optimized C/C++ code and we have presented some benchmark results that support this claim. However, there exist circumstances that native C/C++ code can perform better. We presented some of the experience we gained using the GSL scientific library that is implemented in C. We utilized the javacpp open source project in order to implement the corresponding C to Java interface. Future work can improve on the native interfaces of ScalaLab, and can possibly utilize even more efficient C/C++ native routines, to complement the computational potential of the JVM.

REFERENCES

[1] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Numerical Recipes in C++, The Art of Scientific Computing, Second Edition, Cambridge University Press, 2002
[2] Stergios Papadimitriou, Konstantinos Terzidis, Seferina Mavroudi, Spiridon Likothanasis, Scientific Scripting for the Java Platform with jLab, IEEE Computing in Science and Engineering (CISE), July/August 2009, Vol. 11, No 4, pp. 50-60
[3] Martin Odersky, Lex Spoon, Bill Venners, Programming in Scala, Artima eds, 2008
[4] Venkat Subramaniam, Programming Scala Tackle Multicore Complexity on the Java Virtual Machine, Pragmatic Bookself 2009
[5] Stergios Papadimitriou, Konstantinos Terzidis, Seferina Mavroudi, Spiridon Likothanasis, ScalaLab: an effective scientific programming environment for the Java Platform based on the Scala object-functional language, IEEE Computing in Science and Engineering (CISE), Vol. 13, No 5, 2011, p. 43-55
[6] E. Anderson, Z. Bai, C. Birschof, S. Blackford, J. Demmel, J. Dongarra, J.Du Croz, A. Greenbaum, S. Hammarling, A. Mckenney, D. Sorensen, LAPACK Users' Guide, SIAM, Third Edition, 1999
[7] Gilles Dubochet, On Embedding domain-specific languages with user-friendly syntax, In Proceedings of the 1st Workshop on Domain Specific Program Development, pages 19-22, 2006
[8] Gilles Dubochet, Embedded Domain-Specific Languages using Libraries and Dynamic Metaprogramming, PhD Thesis, EPFL, Suise, 2011
[9] Timothy A. Davis, Direct Methods for Sparse Linear Systems, SIAM publishing, 2006
[10] Nikishkov, Y, Nikishkov, G & Savchenko, V 2003 Comparison of C and Java performance in finite element computations, Computers and Structures, vol. 81, no. 24, pp. 2401-2408.
[11] Taboada, GL, Ramos, S, Exposito, RR, Tourino, J & Doallo, R 2013 Java in the High Performance Computing arena: Research, practice and experience, Science of Computer Programming, vol. 78, no. 5, pp. 425.
[12] Lewis, J.P, Neumann, U, 2004 Performance of Java versus C++, University of Southern California, ¡http://scribblethink.org/Computer/javaCbenchmark.html¿
[13] Oracle n.d., The Java HotSpot Performance Engine Architecture¡http://www.oracle.com/technetwork/java/whitepaper-135217.html¿
[14] Oancea, B, Rosca, IG, Andrei, T & Iacob, AI 2011 Evaluating Java performance for linear algebra numerical computations, Procedia Computer Science, vol. 3, pp. 474-478.

ISBN: 978-988-19253-0-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)