HYBRID JAVA PARALLELIZER: A FRAMEWORK FOR PARALLELIZATION OF JAVA CODE

A. Iqbal and M.A. Khan

Department of Computer Science, Bahauddin Zakariya University Multan, Pakistan
Corresponding author's email: adeel.iqbal@hotmail.com

ABSTRACT: An enormous effort had been placed for converting sequential code to parallel in an automated way. In this regard, the frameworks like JOMP and JaMP, were proposed to facilitate Java programmers for parallelization of code. However, the programmer was still bound to provide directives to the compiler about possibly parallel portion of code and architectural specification in some predefined format. Moreover, these frameworks required source code as input, thereby constraining performance improvement subject to the availability of the source code. This paper proposed a framework called Hybrid Java Parallelizer (HJP) which was aimed at performance improvement through parallelization of Java code. It did not require source code for parallelization and was able to create threads according to the available cores. Experimentation was performed on the well-known matrix multiplication benchmark. Results showed that HJP achieved average speed-ups of 6.99, 3.54, and 6.98 times on machines having Intel Corei7, Core i5, and Xeon based processors, respectively.

Keywords: Code Parallelization, Java Programming, Multi-threading, Parallel Programming, Software Development.

(Received 02-05-2017 Accepted 28-12-2017)

INTRODUCTION

With the launch of first multi-core server by IBM in 2001(Sinharoy *et al.,* 2011) need for parallel applications has also increased potentially. On one hand, processor manufacturers are growing the number of processing units in a machine, and modern programming languages are providing extensive support for parallel execution of code in order to improve the performance. Consequently, many compilers, translators and libraries have been implemented to facilitate parallelization both in new languages and conventional languages like FORTRAN/C++ *etc.* For *Java* language, *bytecode* is executed by the Java Virtual Machine (JVM) (Lindholm *et al.,* 2015) and JVM makes it portable for a variety of devices ranging from embedded devices to powerful high performance servers. Around 5.5 billion devices are using JVM (Madhavarao and JayaRaju, 2013). A good performance on all the diverse systems can be achieved if the workload is evenly distributed among all the processing units, thereby requiring the parallelization of code. To exploit parallelism, *Java* supports threads which may use local memory and shared memory (Taboada *et al.,* 2013, Adve and Boehm, 2010).

Different frameworks such as JaMP (Veerasamy and Nasira, 2014), Pyjama (Giacaman and Sinnen, 2013), JOMP (Zhang *et al.,* 2015, Giacaman and Sinnen, 2013), JavaParty (Abbasi *et al.,* 2011), Titanium (Taboada *et al.,* 2013, Yelick, Graham *et al.,* 2011), Jackal (Ramos *et al.,* 2011), *and SPAR Java (*Zhang *et al.,* 2015) have been proposed and implemented. JOMP uses directives for parallelization. It's compiler is implemented in *Java* while using the *Java compiler compiler* (JavaCC) tool (Nagaveni and Raju, 2014, Viswanadha and Sankar, 2009). The directives are translated into the calls to functions from the runtime library which invokes *Java* threads for parallelization. *JaMP*, an implementation of *OpenMP* in *Java* includes a set of extensions similar to *OpenMP* (Veerasamy and Nasira, 2014). *Jackal* is an implementation for distributed shared memory environments. It is object-based and uses a cluster for parallel processing in *Java*. It performs distribution of workload for the cluster of workstations (Taboada *et al.,* 2013, Ramos *et al.,* 2011).

Bytecode decompilation is a major activity in the new proposed framework. However, while decompiling any *Java class* file, the decompilers can't regenerate the actual source code; instead they give an equivalent code (Grune, 2012, Nolan, 2004) .

MATERIALS AND METHODS

This paper proposed a prototype framework called HJP which worked in a shared memory model and enabled sequential *Java* code to become parallel by incorporating threads. The HJP framework taken as input the method names and optimized code in conformance with the underlying architecture. It

Worked in an automated manner and could also optimize *bytecode* without requiring the source code.

The JODE decompiler (Hoenicke, 2002).*class* file to a source file. It generated the source code in a

string which was used easily from any other *class*. For parsing *Java* code, *Javaparser* (Gesser, 2012), JavaCC parser generator.

The *Hybrid Java Parallelizer* (HJP) framework was aimed at performance improvement through maximum utilization of resources available on the machines while working in a shared memory model. It parallelizes the *Java bytecode* without requiring the *source code*. This section described the working mechanism and the implementation of the proposed framework.

The HJP framework worked in two phases: decompilation and parsing. The decompilation was referred to as transforming machine-readable object code back to source code.

The architecture of the HJP framework was depicted in Figure 1. The framework took as input the file and method name whose code needed to be parallelized.

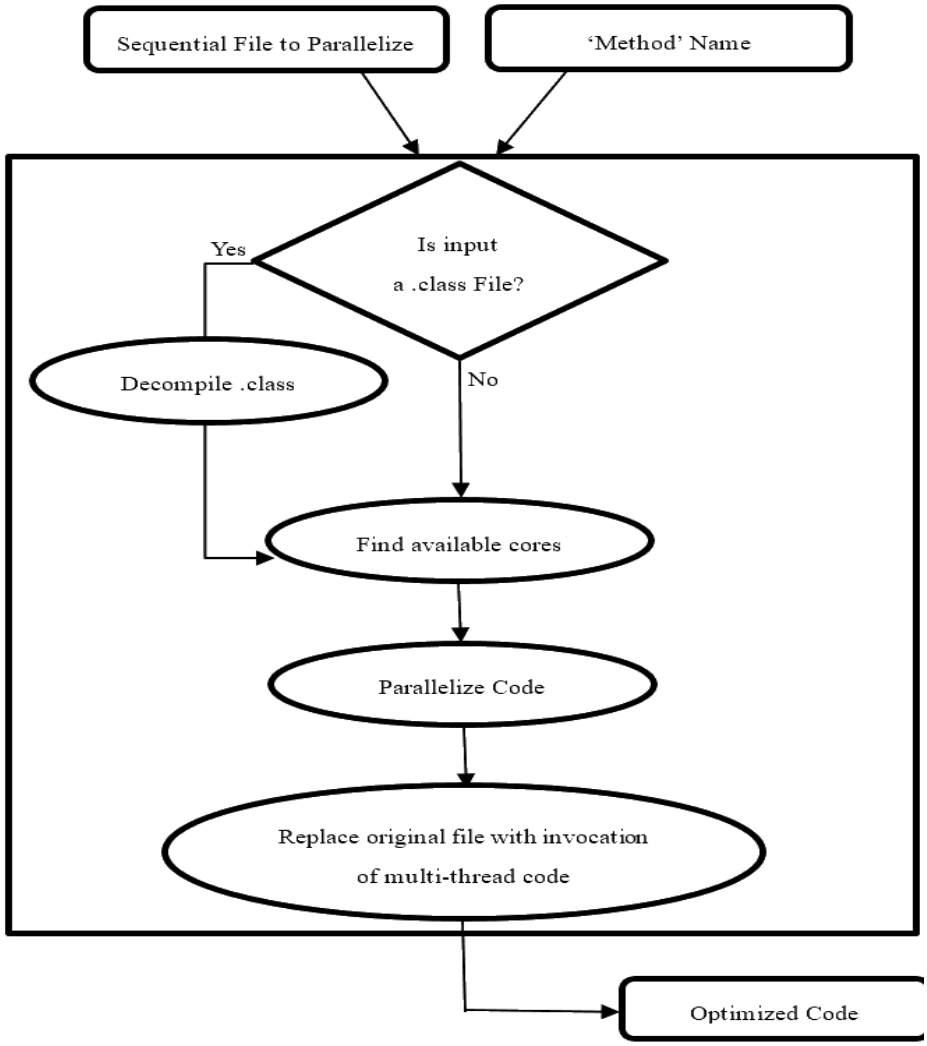

Figure-1: HJP Framework Architecture

If the input was a *.class* file, the code would first be decompiled before proceeding for parallelization. Subsequently, the number of cores existing in the system was found. This was followed by parallelization of code that was accomplished by creating multi-threaded code for the method. In the current prototype implementation, the parser searched for only the *for*-loop and distributed its workload among the cores. In the last step, the code was further modified to invoke the multi-threaded code instead of the original sequential code.

The call to HJP framework was made through the command line interface as well as from *Java* code. To parallelize code using HJP, the following command line arguments were required.

java HJPjava Filename methodName [className]

The first argument to HJP was the *.java* file name and the second argument was the name of the method in which the *for-loop* was parallelized. The last optional argument would have been used when the *class* name was different from the file name, i.e. *non-public*

classes. The method name can't have a *return type* and could not be receiving arguments. Moreover, the *inner* classes were not supported. The HJP framework could be invoked from within some other *class*. To accomplish that, HJP provided an API as given follows:

public static int parallelize (String fileName, String methodName)

public static int parallelize (String fileName, String className, String methodName)

The first variant of the *parallelize* method assumed the *class* to be public, whereas the second variant supported parallelization for *non-public classes* as well. For step-wise invocation of parallelization by HJP, two classes *CHandle* and *MHandle* were used, representing respectively the handles for the *class* and the method to be optimized. The major steps of parallelization were supported through the API given below:

public static CHandle getCHandle (String fileName, String className)

public static MHandle getMHandle (CHandle cH, String methodName)

The method *getCHandle* returned an integer number as a handle that was used to identify the *class* whose code needed to be modified. Similarly, the method *MHandle* returned a handle to the method that would be modified and transformed for parallel execution.

public static int createBackup (CHandle cH)

The method *createBackup* was used to create a backup file of the actual file in order to avoid any data loss in case of error.

public static int getAvailableCores()

The method *getAvailableCores* returned the number of cores required for creating threads and the method *cDecompile* was used to decompile the *.class* file to a *.java* source code file.

public static int cDecompile (CHandle cH)

public static int transform (MHandle mH)

Using the *transform* method, the code first determined the number of cores and then extended the *class* with the *Thread class*. The actual method was then renamed with "run" in order to make several instances execute concurrently. A new method having the same name as the original method was added which contains code for creation and invocation of the threads (using the Thread's *start* method). The entire modified file may then be compiled and executed to benefit from multi-threaded execution.

The current HJP prototype was limited to parallelization of a single outer most *for-Loop* representing the number of iterations of some computation. Consequently, no code dependencies were analyzed as each of iteration was supposed to contain independent work.

For experimental testing of the HJP performance analysis, the following machines were used.

i. Machine-A: Intel Core i7-2600M CPU, 3.40 GHz with 4GB RAM
ii. Machine-B: Intel Core i5-5200U CPU, 2.20 GHz with 4GB RAM
iii. Machine-C: Intel Xeon CPU E5606, 2.13Ghz (2 Processors) with 8GB RAM

Machine-A comprised an Intel Core i7 based processor with 4 physical cores (8 logical cores). Similarly, Machine-B comprised an Intel Core i5 based processor with 2 physical cores (4 logical cores). Machine-C comprised an Intel Xeon processor with 2 physical processors (8 logical cores).

Matrix Multiplication benchmark, considered to be a major high performance benchmark was used for testing. A simple matrix multiplication problem with square matrices of input sizes 500 X 500, 1000 X 1000, 1500 X 1500, …., 6000 X 6000 (rows X columns respectively) , represented as M1,M2,M3, …. , M12, respectively, was executed on all of the above mentioned machines. The execution time was measured in milliseconds. This paper also presented the speedup results obtained by the HJP optimized code over the original code.

RESULTS AND DISCUSSION

The performance results on Machine-A were depicted (in logarithmic scale) in Fig.2. The results showed that the HJP generated code performed better for values starting from matrix size of 1500 X 1500 and the performance gained with an increase in the size of input matrix. For Machine-A, the HJP generated code achieved the minimum, average and maximum speed-ups of 3.23, 6.99, and 7.69 times, respectively.

The performance results on Machine-B were depicted (in logarithmic scale) in Figure 3. The results show that the HJP generated code performed better for values starting from matrix size of 2000 X 2000 and the performance gained with an increase in the size of input matrix. For Machine-B, the HJP generated code achieved the minimum, average and maximum speed-ups of 2.06, 3.54, and 3.89 times, respectively.

As shown in Figure 4, the HJP generated code on Machine-C as well performed better for values starting from matrix size of 500 X 500 and the performance gained with an increase in the size of input matrix. For Machine-C, the HJP generated code achieved the minimum, average and maximum speed-ups of 3.04, 6.98, and 7.71 times, respectively.

Experimental results of Machine-A, Machine-B, and Machine-C showed that HJP was a powerful tool which utilized the possible available resources. A summarized view of the results on Machine-A, Machine-B, and Machine-C were shown in Table 1.

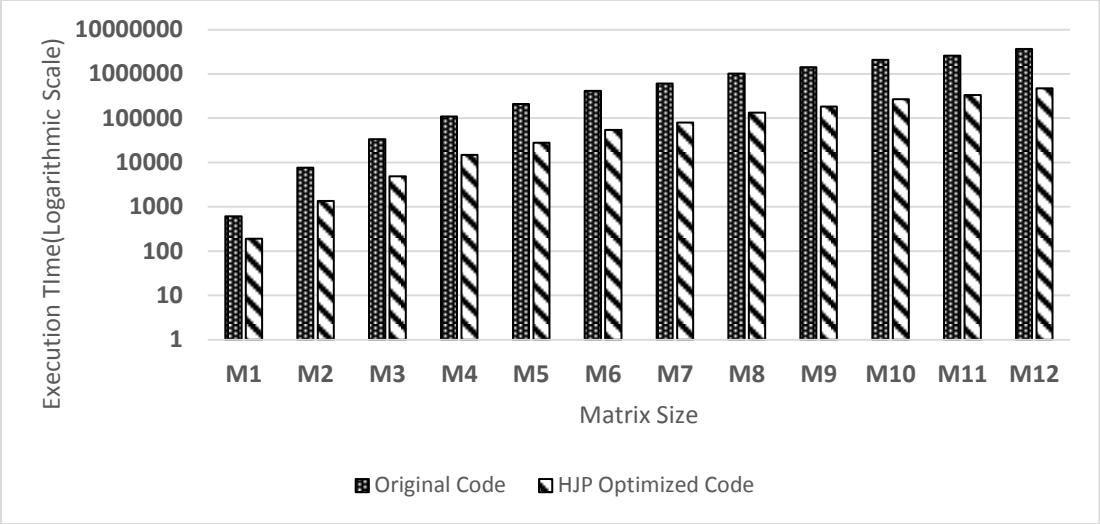

Figure-2: Machine-A Code Execution Time Graph

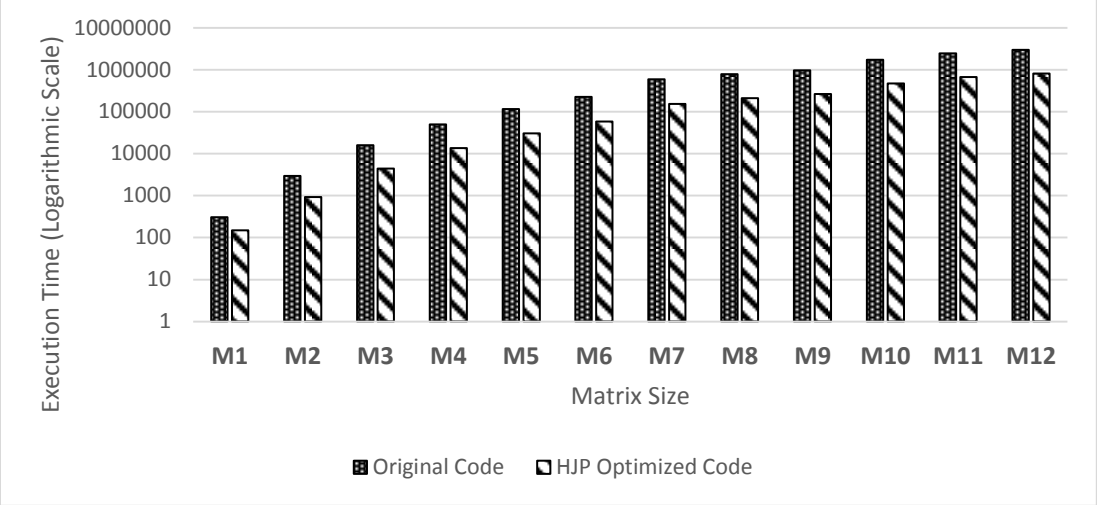

Figure-3: Machine-B Code Execution Time Graph

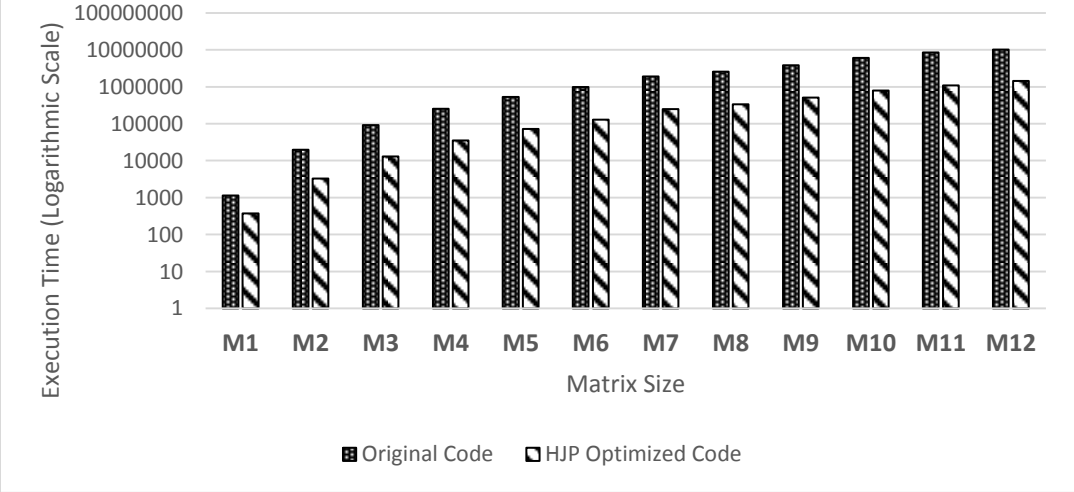

Figure-4: Machine-C Code Execution Time Graph

Table-1: Summarized View of Results (No. Of Times Performance Increased).

	Minimum Speed Up	Average Speed Up	Maximum Speed Up
Machine-A	3.23	6.99	7.69
Machine-B	2.06	3.54	3.89
Machine-C	3.04	6.98	7.71

The minimum speedup of 2.06 times was obtained on Machine-B, whereas, the maximum speed-up of 7.71 times was obtained on Machine-C. The results also showed that the parallel code execution performance was not good for smaller sizes of input matrices such as 500X500 and 1000X1000, because the overhead of thread execution was large and the total execution time was very small.

The significant gain in improvement after parallelization of code was obtained due to the maximum utilization of processors, which manifested the effectiveness of the HJP framework for parallelization of code. In comparison with other frameworks, the HJP framework supported the transformation of code for the compiled *bytecode*, thereby making it prominent over other parallelization frameworks. Yalagi and Apte (2015) examined his sample database on different number of processors and reported speed-up of around 3.5 on 7 processors and speed-up of 1.8 on 4 processors. The JOMP is an implementation of OpenMP in Java, Zhang *et al*. (2015) examined the Pi calculation benchmark on Intel I3 using different threads and reported speed-up of around 1.35 times while using the 6 cores available. Klemm *et al.* (2007) examined Lattice-Boltzmann Method Benchmark and reported speed-up of around 6.1 on their proposed JaMP framework using 8 cores. Giacaman and Sinnen (2013) in their proposed framework named *Pyjama* used series Benchmark, MonteCarlo Benchmark and RayTracer Benchmark. The performance of Pyjama was remarkably good and obtained speed-up of around 2.8, 3.5 and 4 on 4, 6 and 8 cores respectively on MonteCarlo Benchmark and almost similar results were reported on others as well. Ali and Khan (2016) worked on XML parallel compression and reported speed-up of average 2 on different machines.

Hybrid Java Parallelizer (HJP) was intended for improvement through parallelization in a shared memory model. The HJP framework performed code transformation from the source code (*.java* file) as well as the *bytecode* (*.class* file). If the given input was a *bytecode*, the decompilation was performed first and an equivalent source file (*.java*) was generated. Subsequent to the generation of source code, the HJP framework performed optimization of code for the specified region and generated an equivalent parallel code. The parallel code generated by HJP contained threads which were executed in parallel. The number of threads depended upon the number of cores available in the machine on which the HJP was initiated. This functionality differentiated HJP from all the existing frameworks including JaMP (Veerasamy and Nasira, 2014), JOMP (Zhang *et al.*, 2015, Giacaman and Sinnen, 2013), and HPJava (Carpenter and Fox, 2003) etc. that required the source code to be available as well as the programmer's directive describing the parallelism in terms of the processing units. In order to provide portability for diverse architectures, the HJP framework itself had been implemented in Java.

As future work, the HJP framework would be extended to support multiple loops together with a limited dependence analysis in order to parallelize code accordingly.

Conclusion: Proposed HJP framework has the capability to decompile and then parallelize the java class files which distinguish it from its predecessors. Although it has some limitations during the experimentation phase.

REFERENCES

Abbasi, M.J., S. Koupaeimalek , S. Khachateryan and M.S.A. Lattif (2011). Java based high performance of parallel processing application with JavaParty. Int. J. Comput. Sci. Telecom. 2(9): 22-25.

Adve, S.V. and H.J. Boehm (2010). Memory models: a case for rethinking parallel languages and hardware. Commun. ACM. 53(8): 90-101.

Ali, M. and M.A. Khan (2016). Efficient parallel compression and decompression for large XML files. Int. Arab J. Inf. Technol., 13(4): 403-408.

Carpenter, B. and G. Fox (2003). Hpjava: A data parallel programming alternative. Comput. Sci. Eng. 5(3): 60-64.

Madhavarao, M.E. and C. JayaRaju (2013). Data sharing in the cloud using distributed accountability. Int. J. Adv. Res. Comput. Eng. Tech. 2(6): 1939-1944.

Gesser, J.V. (2012). "Javaparser–Java 1.5 Parser and AST." Retrieved June 30, 2013, from http://code.google.com/p/javaparser.

Giacaman, N. and O. Sinnen (2013). Pyjama: OpenMP-like implementation for Java, with GUI extensions. 2013 Int. Workshop PMAM, ACM. 43-52.

Grune, D. (2012). Modern compiler design. Springer Sci. Bus. Media. 382 p.

Hoenicke, J. (2002). "JODE." Retrieved 1-Feb-2017, from http://jode.sourceforge.net/.

Klemm, M., M. Bezold, R. Veldema and M. Philippsen (2007). JaMP: an implementation of OpenMP for a Java DSM. Concurr. Comp-Pract. E. 19(18): 2333-2352.

Lindholm, T., F. Yellin, G. Bracha and A. Buckley (2014). The Java Virtual Machine Specification. 8th Ed. Addison-Wesley. 2 p.

Nagaveni, V. and G. Raju (2014). Parallelizing Quicksort Algorithm using JOMP and its performance analysis with OpenMP. Int. J. Eng. Sci. Tech. 6(5): 254.

Nolan, G. (2004). Decompiling Java. A press. 2 p.

Ramos, S., G.L. Taboada, J. Tourino and R. Doallo (2011). Scalable java communication middleware for hybrid shared/distributed memory architectures. IEEE 13th Int. Conf. HPCC. 221-228.

Sinharoy, B., R. Kalla, W.J. Starke, H.Q. Le, R. Cargnoni, J.V. Norstrand, B.J. Ronchetti, J. Stuecheli, J. Leenstra and G.L. Guthrie (2011). IBM POWER7 multicore server processor. IBM J. Res. Dev. 55(3): 1-1.

Taboada, G.L., S. Ramos, R.R. Expósito, J. Touriño and R. Doallo (2013). Java in the High Performance Computing arena: Research, practice and experience. Sci. Comput. Program. 78(5): 425-444.

Veerasamy, B.D. and D.G. Nasira (2014). Overall Aspects of Java Native Threads on Win32 Platform. 2nd Int. Conf. ERCICA. 667-675.

Viswanadha, S. and S. Sankar (2009). "Java compiler compiler (JavaCC): The Java parser generator." Retrieved March 20, 2013, from https://javacc.dev.java.net/.

Yalagi, P.S. and S.S. Apte (2015). Parallelization of Annotated Java code in a Distributed Network. Int. J. Comput. Appl. Technol. 110(7): 6-9.

Yelick, K., S.L. Graham, P. Hilfinger, D. Bonachea, J. Su, A. Kamil, K. Datta, P. Colella and T. Wen (2011). Titanium. 1st Ed. Encycl. Parallel Comput. Springer. 2049 p.

Zhang, H., X. Wang, J. Cao and C. Zhu (2015). Parallel computing of shared memory multiprocessors based on JOMP. Int. Conf. MEIC. 1510-1514.