MTR130458

MITRE TECHNICAL REPORT

MITRE

Mixing C and Java™ for High Performance Computing

Nazario Irizarry, Jr.

September 2013

Bedford, MA

This page intentionally left blank.

Abstract

Performance computing software, especially high performance embedded computing (HPEC) software, is typically developed in C for processing as well as infrastructure functionality. Infrastructure includes communications, processor/core allocation, task management, job scheduling, fault detection, fault handling, and logging. C and optimized assembler language libraries are used to achieve the highest processing performance relative to limitations on size, weight, and power dissipation. There has been a trend to move away from dedicated logic and specialty programmable processors to commodity-based processors. As a result, it is now possible to examine more robust software options. Many of these functions, especially infrastructure functions, can be implemented quickly and safely utilizing Java™ frameworks but some doubt that the performance can be satisfactory. Java frameworks have extensive open source and vendor support, and Java's platform independence reduces the need to redevelop functions as hardware and operating systems evolve.

Tests show that the performance of Oracle® Java™ 7 Standard Edition (on Intel processors) can equal that of C for some problems if dynamic object creation is used judiciously and the application can afford a brief start-up and warm-up time. Java can invoke native libraries very quickly. With some of the available bridging technologies it can natively allocate data and even extend its garbage collection discipline to such data. Even though there is some additional Java overhead for invoking and utilizing graphics processing units, in tests it was able to utilize the graphical processing unit (GPU) at the same rate as native C code when the data was allocated natively.

Java compute-grid frameworks offer features such as auto-discovery, auto-failover, inter-node data synchronization, automatic data serialization, multi-node work queuing, active load monitoring, adaptive load balancing, and, dynamic grid resizing that can save significant developer time if such features are needed. GridGain was easy to setup and use and has many options for work distribution and handling node failures. Akka and Storm, performed well relative to the high-performance community's Message Passing Interface (MPI) even though such a comparison is unfair due to the significant differences between MPI framework features and the Java framework features. On a 10 gigabit Ethernet cluster, Akka and Storm achieved 10,000 asynchronous round-trip exchanges per second, which is more than adequate for many applications.

Scala and Python also are investigated briefly to understand how they perform relative to each other and relative to Java. Scala is newer, more concise, and equal in performance to Java. Python is older, does not perform as well, but has extensive native library support and a concise syntax.

Hybrid Java/C applications make sense and can be designed to perform well. This opens the door to intricate high performance applications in which developer productivity via Java can be traded against the ultimate compute speed attainable with C and native customized libraries.

This page intentionally left blank.

Executive Summary

Developers of performance computing software, especially high performance embedded computing (HPEC) software, develop code not just for computational processing but also for management and infrastructure tasks such as communication, processor/core allocation, task management, job scheduling, fault detection, fault handling, and logging. HPEC software development has required extensive software development to get the highest performance relative to limitations on size, weight, and power dissipation. In the past, these limitations often drove the choice between dedicated logic and programmable processors.

There have been significant improvements in processing capabilities relative to these limitations as processor feature sizes have shrunk over time and additional processor cores have become available in commodity HPEC processor designs—resulting in the trend to move away from dedicated logic and specialty programmable processors to commodity-based processors. As the hardware and run-time executives of the sensor platform are updated, such software is often discarded and new versions are redesigned and re-implemented. As a result of this additional processing capability, it is now possible to examine more robust software options for HPEC—ranging from fuller-featured operating systems to Java™. Java frameworks support portability between multiple hardware and operating system platforms. Utilizing Java frameworks would reduce redesign and re-implementation. There exist numerous commercial and open source Java frameworks that provide hardware-independent ready solutions to multi-processor infrastructure needs.

There are those who question whether a Java application can provide the requisite performance. The goal is not to replace C and optimized assembler libraries but to mix Java, C, and the libraries. A hybrid application allows the design engineer to exploit each component's strengths with the expected overall benefit of Java's enhanced code safety, maintainability, and expedited development. Investigations were performed to characterize what performance one might obtain from Java. The goal was to be able to design a hybrid application backed by engineering trade-offs based on quantitative data. This study has collected data on an initial set of characteristics.

The following aspects were investigated:

- How well does Java perform on various computation algorithms?
- What is the overhead of the Java Runtime?
- How do the various technologies by which Java can invoke native C compare in performance and ease of use? What is the performance impact of data in Java memory versus data in C memory?
- How would one implement concurrent multi-core computation in Java and how effective is that relative to C?
- How would one employ the processing power of graphical processing units (GPUs) from Java? How does the performance compare to doing that from C?
- Given that Java performance is dependent on the Java Virtual Machine (JVM), is the performance of other JVM languages similar to Java's?

Four types of algorithms that were initially implemented in C were converted to Java—matrix-multiply, Fast Fourier Transforms (FFTs), linear feedback shift register (LFSR) sequence

generation, and, red-black tree sorting. These exercise the processor in different ways. The matrix multiply has nested loops with small compute bodies and sequential array indexes. The FFT has bigger loop bodies and non-sequential array indexes. The LFSR sequence requires integer-based bit masking and Boolean operations. The red-black tree sorting is highly recursive and requires significant dynamic object creation and cleanup.

The testing was performed using Oracle® Java 7 Standard Edition on an Intel four-core x86 system running the Linux operating system. All test data was collected after the application was allowed to exercise the processing paths because Java execution is faster after warm up. Once warmed the Java matrix multiply ran faster than the C version that was compiled with a GNU compiler at its highest level of optimization. The Java FFT performance was 20 to 40 percent slower than C's. This is likely due to the more complex and non-sequential use of array indexes as compared with the matrix multiplication. On the LFSR sequence generation the Java implementation ran 30 percent faster than the one in C.

These first three tests require no memory allocation for computation. In contrast, the red-black tree sorting test requires heavy use of memory allocation to allocate space for keys and values to be inserted into the tree. The entire tree and its contents are disposed between iterations, requiring the garbage collection system to work to recover unused memory. For small and moderate allocations Java was a little faster for insertion and a little slower than C for retrieval. The faster insertion speed degraded to poorer as the allocation size increased.

The fact that sometimes the Java computation is faster and sometimes slower suggests that:

- Hardware architecture, such as cache size, instruction pipeline depths, number and throughput of floating point versus integer arithmetic logic units may impact these findings.
- The HPEC designer should test the application kernels to really know what the interaction of the code and the target hardware will be. It should not be assumed that C code would always be faster.
- Even if the heavy computation is done in C, incidental computation might be handled by Java. In the end the bigger consideration may be whether the data is located in Java or on the native side and how many times it has to move back and forth.

The findings confirm that the overall performance of Java can be comparable to C's—after Java warm-up. Typically, performance computing systems repeatedly run fixed application software in a production mode. Consequently, the time for the Java warm-up will have a negligible impact as the software runs for extended periods of time (hours, days, or even weeks).

The Java application will always utilize multiple threads for runtime maintenance even if the user code does not start multiple threads. In a multicore machine the maintenance work can have little impact on the main work thread(s) if the main threads are not sharing the same cores as the maintenance threads. A test was designed to measure the maintenance load. FFT and red-black tree tests were instrumented to measure work completion time as the number of processor cores was varied. From this it was confirmed that when there is no dynamic memory allocation there is no measurable overhead. On the other hand, the red-black tree test was intentionally run with very limited memory to force about seven garbage collector operations per second. The impact was a 15 percent decrease in compute throughput. A well-tuned application would have significantly less garbage collector activity. If the garbage collector activity only occurs every few seconds then the impact should be propritionally less. It is reasonable to expect less than a 1

percent impact. Note that a different JVM should result in a different overhead. The developer should measure the target platform.

In a hybrid application, Java could be used for communication, processor/core allocation, task management, work sequencing, fault detection, fault handling, and logging. Calling from Java into native computation code is key. Using the low-level Java Native Interface (JNI) is the fastest, but also the most tedious and error-prone, way to do this. Java Native Access (JNA) is the slowest of the callout mechanisms, handling a few hundred invocations per second. JNA is easy to use and is platform independent. If callout is required, thousands or tens-of thousands of times per second, then BridJ and the Simplified Wrapper and Interface Generator (SWIG) are the likely choices. They provide similar performance. BridJ is easier to use (in the investigators opinion) and can be used without external native libraries on various hardware and operating system platforms. SWIG is more work to use, though it generates C code that allows a SWIG-based native library to be compiled for any target platform.

Regarding computational concurrency, Java does not have a simple syntax for concurrent looping. C has OpenMP, that provides a fairly concise syntax. Testing on a four-core machine with various workload splits and various loading showed that the best parallel computing techniques fell shy of C's performance by 4 to 5 percent. The fastest Java concurrency mechanisms were not the standard ones built into the Java 7 standard edition. The third-party libraries, Javolution and Parallel Java, had the best concurrency for the pure-compute test used here. This does not diminish the value of the built-in concurrency mechanisms. They have features for scheduled, periodic, delayed, and recursive tasking that are necessary for many types of tasks. Testing also showed that sometimes there was erratic performance on Linux in which adding more processor cores did not reduce the overall compute time. However, this was mitigated by running with Linux round-robin scheduling.

GPUs can be effectively invoked from Java. The approach investigated here utilized a Java library that invokes an underlying native OpenCL interface to the GPU. Thus a Java application can never be faster than a C application that can utilize the native OpenCL interface directly. However, the goal was to test performance given this constraint.

There are only a few Java interface libraries for OpenCL. JavaCL was used here. It was developed by the same open-source developer that developed the BridJ callout mechanism mentioned above. A test using GPU-optimized FFT kernels originally developed by Apple, Inc. showed that it took Java code about 2 microseconds longer to set up and invoke a compute kernel than C code. The C code required about 9 microseconds. However, since GPU kernel computation proceeds in parallel with kernel setup, the net impact on GPU FFT throughput was typically less than 2 percent. Note that with these kinds of setup times GPU interactions exceeding thousands per second are reasonable.

Five Java compute-grid frameworks, Parallel Java, Java Parallel Processing Framework, GridGain, Storm, and, Akka were tested on a 10-gigabyte Ethernet cluster. Collectively the frameworks offer features such as auto-discovery, auto-failover, inter-node data synchronization, automatic data serialization, multi-node work queuing, active load monitoring and adaptive load balancing that can save significant developer time. They were compared with each other and to the high performance community's Message Passing Interface (MPI), which is the workhorse for performance computing.

With greater flexibility and features, the more sophisticated frameworks were not expected to perform as well as MPI. Indeed only MPI and low-level Java socket input/output (I/O) could

handle small data sizes at rates of 10,000 synchronous round-trip exchanges per second. On the other hand Akka and Storm achieved 10,000 asynchronous round-trip exchanges per second and outperformed MPI in some cases. GridGain was the easiest to setup and use, in the investigators opinion. The tradeoffs between throughput, features, productivity, and life-cycle costs are complex and must be addressed for each application.

Other JVM languages were tested to understand how their performance compares to Java and C. Python is used in the performance-computing world for scriptablity, conciseness, and ease of use in part due to untyped variable declarations. Python has various implementations such as CPython, which is interpreted, PyPy which is compiled, and Jython which runs on the JVM. The Jython 2.7.0 performance was a little faster than CPython 2.6.6 for the memory structures (red-black tree) test, though about 30 percent slower on pure computation (matrix multiply). Jython was a bit faster than CPython at calling out to native code. CPython would not typically be used for computation but neither would Jython. PyPy 1.9 was 120 times faster than Jython on the matrix multiply but one third as fast as Java 7.

Scala is a newer language that runs on the JVM. It is scriptable and concise. It has strong type safety. Variable declarations are easier to write than in Java. It also has strong features for functional coding including the ability to declare operators (such as add, multiply) for user-defined data types (such as vectors and matrixes). This is potentially of interest to the performance developer. Testing showed that Scala has the performance of Java. However, integer for-loops need to be replaced with while-loops because Scala's "for-based" looping mechanism does not currently optimize as well as it could for simple integer loops. This problem is expected to be fixed in the forthcoming Scala 2.11. On the 4-byte red-black tree insertion and retrieval tests, Scala performed better than Java due to a unique language feature that allows generalized algorithms to be optimized for primitive data types.

Performance always needs to be tested and statistics need to be gathered to compare and evaluate alternatives. The benchmark code developed during this study can be used to evaluate Java and operating system platforms other than the Oracle Java 7 standard edition and Linux combination used here. The benchmarks have been released as open source at http://jcompbmarks.sourceforge.net. The timing utilities developed here have been made available at http://jtimeandprofile.sourceforge.net. A tool to simplify calling from C to Java has been released at http://cshimtojava.sourceforge.net.

The overall conclusion is that though Java start-up time is longer than C's, long-running applications can reasonably mix Java and C code and expect good performance, provided a few basic design guidelines are followed:

- Allocate data vectors, matrices, and images on the native side to avoid the overhead of Java-to-native conversion if repeated native processing will access the data. For most of today's commodity platforms, BridJ provides an easy-to-use way to do this that performs well.
- Design interactions between the two at an appropriate level of method-call granularity. Thousands of calls per second can be implemented effectively but a design that requires too much interaction should be questioned.
- Minimize dynamic memory allocation by preallocating working data structures and buffers. Keeping Java's garbage collector quiet is key. Beware of how strings and

iterators are used because they can inadvertently result in much dynamic memory allocation if used carelessly.

- If running long, parallel, compute tasks in Java, beware if fully loading all cores on the CPU for sustained periods. Performance can degrade unless real-time scheduling is used.
- Allow for Java warm-up time. Design the code to allow key code paths to be exercised as part of start-up.

Java, and the JVM, can be part of the high performance tool chest for applications that run long enough to allow Java 7 Hotspot warm-up optimizations to "kick in." A hybrid Java/C development can exploit the productivity and code safety of Java and the speed of C to build solutions that perform well, make good use of scarce development dollars, and leverage extensive Java resources.

Acknowledgments

Thanks to David Koester and Brian Sroka for their help reviewing data and findings and for sharing their experience and perspectives on high-performance embedded computing. Thanks to Amy Lee Kazura with her assistance in implementing some of the timing and profiling libraries. Thanks to Evy MacDade whose early research of related research helped focus this work. Thanks to Claire Cragin who reviewed this report most diligently.

Table of Contents

List of Figures

List of Tables

Code Listings

This page intentionally left blank.

1 Introduction

1.1 Motivation and Intended Audience

This work is intended for developers of high performance code, especially those of high performance embedded computing (HPEC) applications. HPEC can cover a range from smaller read-only memory (ROM)-based devices to larger multi-board, disk-based, turn-key systems. This study is focused on the latter.

HPEC software includes not just computational processing but also management and infrastructure software for communication, processor/core allocation, task management, job scheduling, fault detection, fault handling, and logging. HPEC software development has required extensive software development to get the highest performance relative to limitations on size, weight, and power dissipation. In the past, these limitations often drove the choice between dedicated logic or programmable processors.

There have been significant improvements in processing capabilities relative to these limitations as processor feature sizes have shrunk over time and additional processor cores have become available in commodity processor designs—resulting in the trend to move away from dedicated logic and specialty programmable processors to commodity-based processors. As the hardware and run-time executives of sensor platform are updated, such software is often discarded and new versions are redesigned and re-implemented. As a result of this additional processing capability, it is now possible to apply more robust software options for HPEC—ranging from fuller-featured operating systems to Java™. Java frameworks support portability between multiple hardware and operating system platforms. Utilizing Java frameworks would reduce redesign and re-implementation. There exist numerous commercial and open source Java frameworks that provide hardware-independent, ready solutions to multi-processor infrastructure needs. Java's potential for fewer defects, enhanced code safety, and code reuse spanning an Internet-scale repository of frameworks, libraries, and tools cannot be dismissed. There is also a large talent pool of Java developers that could potentially be tapped for development.

The premise of this report is that it is possible to mix Java and C (and optimized native libraries) and maintain good performance. The question is how to mix these? Does Java perform well by itself? Are there algorithms that are best implemented in one or the other? If Java calls C or native libraries what is the overhead? What are the best ways to code Java? How much of a performance "hit" should one expect due to dynamic memory allocation and the garbage collector's automatic cleanup?

To mix Java and C, the design engineer will want to make design trade-offs based on initial quantitative expectations of what those trade-offs might mean to performance. This report attempts to quantify some of these expectations by testing Oracle® Hotspot™-based Java in different ways and comparing the results to those for equivalent C code. This report also documents coding patterns that lead to good performing Java code.

Today's processor systems usually include graphics processing units (GPU) that can be invoked from the main central processing unit (CPU). Again this report provides the results of quantitative experiments to determine how much Java overhead one might expect when using GPUs.

Embedded turn-key systems often utilize more than one processing node to handle the load and to provide reserve and/or failover capability. This report looks at the capabilities of Java

compute-grid frameworks to understand what they have to offer and what kind of performance one might obtain.

The point of this report is not to displace C but to address the still persisting notion that Java is summarily slow. A crafted mix of Java, C and native libraries can exploit the productivity and code safety of Java as well as the speed of C to build solutions that perform well and make good use of increasingly scarce development dollars and available Java-talent.

1.2 About High Performance Embedded Computing Systems

Embedded systems are characterized by running only a few applications that are well known at design time—in a turnkey production mode. As production systems, they repetitively run accredited applications, and end users are not permitted to reprogram or change them. They have fixed run-time requirements or at least defined bounds on input rates, processing load, and output rates. They are embedded in the sense that they are collocated with or are a part of a larger mission-oriented system or sensor. Embedded systems can be small, such as inside a cell-phone, or they can consist of multiple processor nodes in multiple racks of equipment. This study is more concerned with the moderate sized systems.

1.3 Why Java?

Java was designed by Sun Microsystems (now Oracle) starting in the early 1990's as a way to develop software for small embedded devices that were connected to each other. In 1995 Java was publically released and quickly captured the attention of the developer community. By comparison with C, Java offers:

- similar language syntax for flow of control
- leak-free memory management with automatic garbage collection, buffer overflow/underflow safety
- an object structure. The community has accepted objects as a key way to work with complex data structures and large modular software designs.
- an intrinsic threading model with language constructs for cross-thread synchronization and locking
- standard support for network communication and data structure serialization that is processor and operating system independent
- standard lengths for numerical types
- package, class, file, variable naming conventions that have been well accepted and generally implemented by the developer community
- a processor and operating system independent byte-code format that facilitates code build, deployment, and movement across development, test, deployment, and distributed systems (including heterogeneous compute systems)
- availability of many tools for development (including free).

The last point has deep reaching benefits for some projects because the developer does not have to be "on" the exact target hardware for much of the development. HPEC hardware can often be specialized for form factor and power needs. The ability to develop and test the code on almost any machine can free many projects from the bottleneck imposed by a limited number of specially configured systems.

1.4 Why Not Java?

Java application start-up can be slower than that of other languages for reasons discussed in the body of this report. On the other hand, high performance codes are often long running. The longer it runs, the less significant start-up time is.

Java jitter leads to variable execution times. Vendors of real-time Java systems have Java compilers and run-time environments to address this. The typical desktop Java (perhaps the one used for development) may have variable execution time but the target run-time environment can have more predictable execution times. The author's experience, however, is that while some real-time Java systems can achieve significant predictability, they often do so at the expense of overall speed.

Java is more succinct than C largely due to the absence of the C include files, which are typically used to define structures and method signatures. Java is less succinct than other languages such as Ruby, Python, and Scala. Languages such as MATLAB®[1] are much more succinct for many types of numerical processing.

Succinctness is desirable but so is code safety and these are often at odds with one another. At least one study[2] shows that the defect rate in code is more dependent on the number of lines of code written by an individual programmer than on the language. The author's experience with Scala suggests that the same function can be coded with about three-quarters of the number of lines of code. In one experiment[3] Ruby also required about three-quarters the amount of code, though Python required approximately half of the amount of code.

1.5 Study Goal

This study is not a comprehensive comparison of languages or frameworks. The focus is on showing that Java (and a little more generally the Java Virtual Machine (JVM)) can have a good role in performance processing. The goal is to understand how well by quantifying performance of different types of code. With a quantitative understanding, the developer can then make better engineering trade-offs as to how to mix Java and C. The next section starts the analysis looking at Java performance on different algorithms.

---

[1] MATLAB is a trademark of The MathWorks, Inc.

[2] Hatton, Les. "Why is the defect density curve U-shaped with component size?" for IEEE Software, April 27, 1996. <http://www.leshatton.org/wp-content/uploads/2012/01/Ubend_IS697.pdf.>

[3] Nene, Dhananjay. "Performance Comparison – C++ / Java / Python / Ruby/ Jython / JRuby / Groovy." <http://blog.dhananjaynene.com/2008/07/performance-comparison-c-java-python-ruby-jython-jruby-groovy/>

This page intentionally left blank.

2 Java on One Node

This section presents the findings of several tests performed in C and Java, using "identical" code. A four-core system 64-bit system described below was used. The tests have been made available on the Internet so that readers may test their own platforms.[4]

The next section describes how the tests designed for this study were prepared.

2.1 Approach

This study designed benchmarks that:

- test the same implementations in Java and C
- collect performance metrics to try to characterize what performs best in which language
- compare alternatives where they exist.

Much time was spent to make sure that tests were equivalently implemented in the two languages to avoid skewing the results implementation differences. If one compared an algorithm that used a Java `HashMap` with a "similar" C program that used a standard template library `hash_map`, one would be left wondering about the impact of implementation differences and whether they were significant.

Since Java is object oriented and has garbage collection and C is not, it is easier to take a code implemented in C and translate it to Java than the other way around. C data types, structs, and methods can be translated reasonably directly to Java. This study searched the open source for algorithms implemented in C and converted them into Java. Some minor changes were made. For example, an integer flag in C being used as a true/false condition is more naturally implemented as a Boolean variable in Java.

Embedded compute applications often run modally for a long time. Once input sources and output sinks are identified, a processing flow is often reapplied at a certain rate continuously for a long time. Long-term steady-state throughput over hours is more important than start-up time and initialization. The tests and metrics collected in the benchmarks in this study focus on post-warm up Java behavior.

Java and C have nanosecond resolution timing application programming interfaces (API)—though the hardware platform may not necessarily have clock sources with nanosecond accuracy. It can be tedious to interact with these APIs, when one is looking to repeatedly compute timing statistics for elapsed time, average time, and standard deviation of time. For this reason, a higher-level timing utility was developed so that when the C code was translated to Java, the timing instrumentation could also easily be translated.

2.2 About the Test Platform

The results presented throughout this study were run on a four-core Intel Core 2 Quad Q9650 @ 2.99 Gigahertz. The operating system was Linux with kernel version 2.6.32-220.17.1.el6.x86_64. The C code is compiled with option "-O3" using GCC version 4.4.6. The machine is run without a graphical desktop to minimize interference. The test platform had a

[4] "Java/C Comparative Benchmarks," <http://jcompbmarks.sourceforge.net>

clock granularity of 34 nanoseconds. The sample-to-sample jitter measured by C code was approximately 18 nanoseconds.

Java is run under Oracle's Java SE 7—*Java(TM) SE Runtime Environment (build 1.7.0_02-b13); Java HotSpot(TM) 64-Bit Server VM (build 22.0-b10, mixed mode)*. Other versions and vendors' JVMs were selectively used to verify the consistency of the observations although those results are not reported here. For most tests, the following Java command line flags were used:

```
-Xms500m -Xmx1500m -Xincgc -XX:+UseCompressedOops -XX:+UseNUMA –server
```

Since there are several Java vendors, one must be careful to caveat the findings of this report as applying to Oracle Java 7. Other vendors' JVMs will have different execution characteristics especially with regard to the trade-offs between jitter, speed, and garbage collector predictability/behavior.

2.3 Java Initialization and Warming

It is widely known that Java Standard Edition start-up is slow. To be fair, it is recognized that there are vendors that have ways to precompile and significantly reduce start-up time[5]. However, the following discussion will be limited to the Oracle, Java 7 Standard Edition.

The first execution of code under Standard Edition is slow in comparison to future executions for several reasons. One reason is due to class loading. Java classes are loaded one-by-one as needed at run time. In C, entire code sections pulled together by the linker are loaded as a unit. In Java, individual class files are loaded as needed, which can require more disk input/output (I/O). The specification additionally requires various code validations and verifications as classes are loaded.

A second reason for start-up slowness is due to static initialization. The compiled Java code, which is platform independent, is not a memory image that can be simply loaded into memory. A memory image is platform dependent. Thus, initializations, such as static integer variables that are initially non-zero, actually require code to be executed to reach their initial value.

After loading, execution is monitored by an adaptive just-in-time (JIT) compiler—Hotspot[6]. Hotspot actively gathers execution statistics while the code is running. Hotspot may decide that code that is initially running in an interpreted fashion should be recompiled in memory into native code for better performance. Method calls that can be safely in-lined[7] can have their code moved into other methods to avoid the overhead of putting arguments on the stack and doing

---

[5] Some JVM's such as IBM®'s J9 have options to improve initial load time. They use ahead-of-time compilers (AOT) and they use compiled class caches to reduce the need to run code in interpreted mode. Though they speed up initial execution time, the code can still get faster with time if they also use active profiling and runtime code optimization and in-memory recompilation.

6 "Java SE HotSpot at a Glance." Oracle Technology Network. <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html>

7 Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. 2001. A dynamic optimization framework for a Java just-in-time (JIT) compiler. In Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '01). ACM, New York, NY, USA, 180-195. DOI=10.1145/504282.504296 <http://doi.acm.org/10.1145/504282.504296>

method invocations. There are other[8] optimizations as well. For this reason, JVMs employing Hotspot will usually run faster after the code has been running for a while.

The phrase "Java warming" captures the fact that Java runs faster beyond the loading and initial processing stages. In this study, the statistics collected and presented are either post warm-up or both the non-warmed and warmed results are presented. Warm-up time can be tolerated for applications that run for a long time if subsequent performance is good.

The effect of warming is illustrated in Figure 2-1. The figure shows the execution time for sequential iterations of a pure memory-based algorithm that has no I/O. In each iteration an empty red-black tree is created and filled using a series of key-value insertions. A series of gets are also executed and timed. The first iteration takes so long that it was not plotted to avoid "saturating" the vertical axis. Iterations 2 to 26 take from 0.16 to 0.3 microseconds (µs). At iterations 27, 46, 94, and 301 this changes to 0.08, 0.22, 0.020, and 0.010 µs respectively. The final speed is much faster than the earlier speeds. Table 2-1 shows the computed speedup factors, depending on which intervals one is comparing to the final.

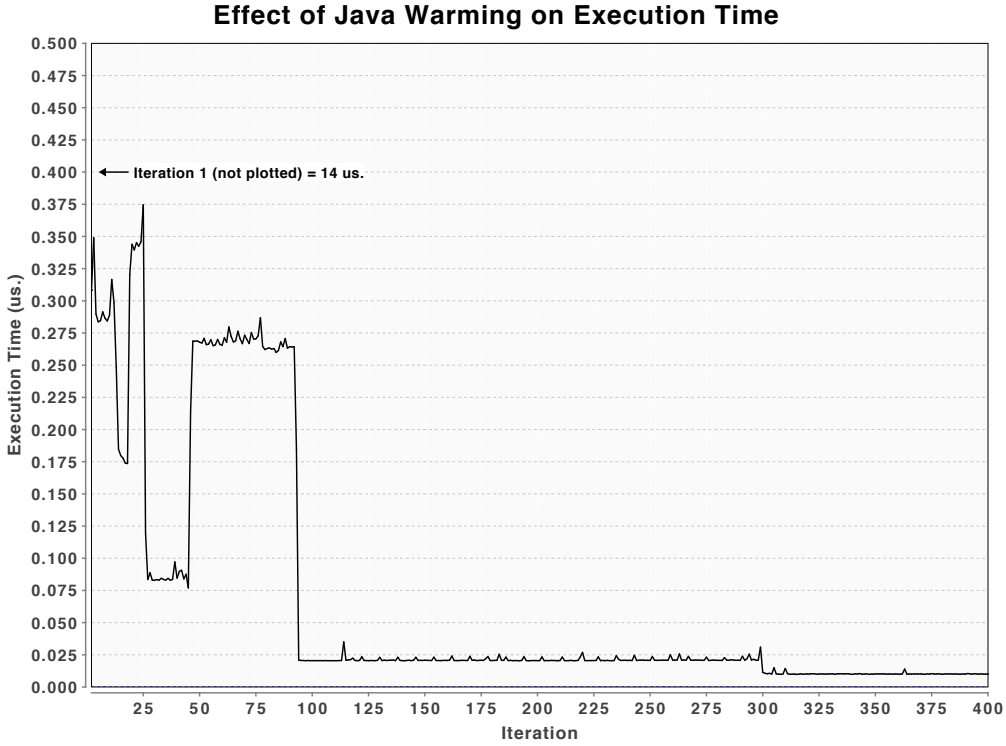

Figure 2-1. Java Warming Improves Speed

8 "The Java HotSpot Performance Engine Architecture." Oracle Technology Network. <http://www.oracle.com/technetwork/java/whitepaper-135217.html#method>

Table 2-1. Relative Speedup Due to Warming

Reference Iteration	Speedup Relative to the Reference
First iteration (includes class loading on the first call of the work method)	1400:1
9	30:1
27	8:1
60	27:1
150	2:1

As a final note, the Java Runtime API includes a class, `java.lang.Compiler`, with methods to direct the Runtime to compile the specified classes. However, implementation of the behavior is optional. JVMs may implement these as "no-ops." Oracle Java 7, Standard Edition, in particular seems to implement these as no-ops.

2.4 Reasons for Jitter

A native C application running in a non real-time operating system will be subject to clock interrupts, I/O interrupts, other operating system interrupts, scheduling irregularities, and more. Real-time operating systems have facilities for suppressing those while the application threads are running and/or ensuring that interrupts are serviced by cores that are not dedicated to the application. Note that the Linux kernel has ways to restrict the general system to certain cores and ways to attach specific processes to certain cores.[9]

Java applications are subject to the same sources plus additional ones. As described above, dynamic class loading and initialization will cause initial loads to take a long time. Code running under interpreted mode can have initial jitter even between successive executions of the same code.[10] Code compiled by the JIT can change speed as increasing optimization is applied to frequently used code. There are real-time Java environments that will forgo run-time tuning but these also forgo the ultimate speed potential so they can achieve execution time repeatability. Finally, work by the garbage collector may require the executing thread to temporarily halt while the garbage collector moves data or if the garbage collector is competing with the current thread for needed CPU time.

9 See the documentation on the "isolcpus" kernel boot parameter at http://www.kernel.org/doc/Documentation/kernel-parameters.txt, which can exclude cores from general scheduling (but not interrupt handling). See also the main pages on "taskset."

10 "Controlling Runtime Jitter." Oracle/Sun Java Real-Time System 2.1. <http://docs.oracle.com/javase/realtime/doc_2.1/release/JavaRTSTechInfo.html#jitter-runtime>

Many of the tests presented below collect Java and C timing jitter statistics. The data are provided as a potential benefit to the developer of real-time software[11]. Many high-performance codes are concerned more with throughput than they are with jitter. The real-time high performance developer will likely be interested also in scheduling jitter, though that is best tested in the context of a real-time operating system, which was not used in these tests.

2.5 Iterative Computation – Matrix Multiply

A benchmark was designed to test execution of a compute intensive algorithm in C and Java. A matrix multiplication test was written. The test used a single array to hold the matrix coefficients as is commonly done in vector and matrix software packages. This avoids multi-dimension indirection.

The same matrix-multiplication kernel was implemented in the two languages. The Java matrix multiply kernel is shown in Listing 2-1. The C kernel is nearly identical except that the syntax of the structure references are, for example, `a->data[indexA]` instead of `a.data[indexA]`.

Listing 2-1. Java Matrix Multiply Kernel

```
/**
 * c = a * b
 */
public static void mult( Matrix a , Matrix b , Matrix c ) {
     int aIndexStart = 0;
     int cIndex = 0;

     for( int i = 0; i < a.nRows; i++ ) {
         for( int j = 0; j < b.nCols; j++ ) {
             double total = 0;

             int indexA = aIndexStart;
             int indexB = j;
             int end = indexA + b.nRows;
             while( indexA < end ) {
                 total += a.data[indexA] * b.data[indexB];
                 indexA += 1;
                 indexB += b.nCols;
             }

             c.data[cIndex++] = total;
         }
         aIndexStart += a.nCols;
     }
 }
```

The time to multiply two NxN matrices was timed for N ranging from 10 to 88. Statistics were collected for 10,000 matrix multiplies. Figure 2-2 compares C and Java matrix-multiply performance. The Java timings were collected after "warming."

11 Jensen, E. Douglas. "Real-Time Overview/Overview of Fundamental Real-Time Concepts and Terms." <http://www.real-time.org/time-constraints-2>

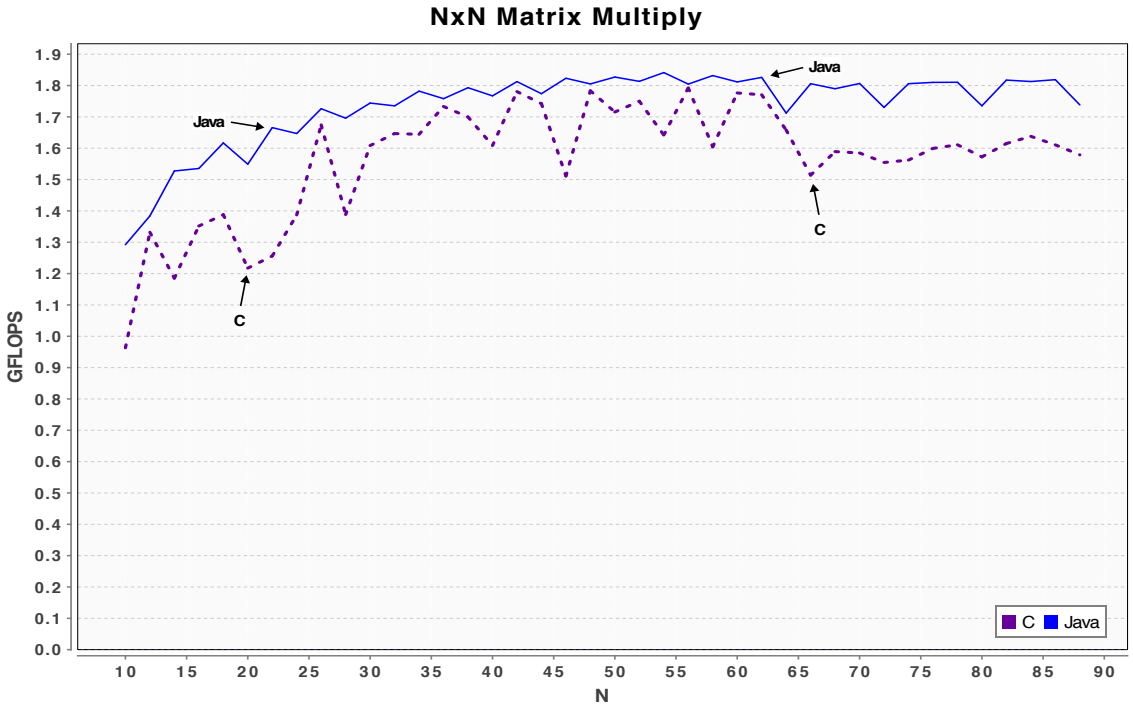

Figure 2-2. Matrix Multiplication Throughput, C and Java

Over the range $40 \leq N \leq 60$ the Java compute throughput exceeded C by a typical 3 percent. Over the range $75 \leq N \leq 88$ this went to about 10 percent. Figure 2-3 shows that execution time jitter, measured as the standard deviation of the execution times, was comparable. The ratio of jitter to the mean execution time, shown in Figure 2-4 is also very close for C and Java.

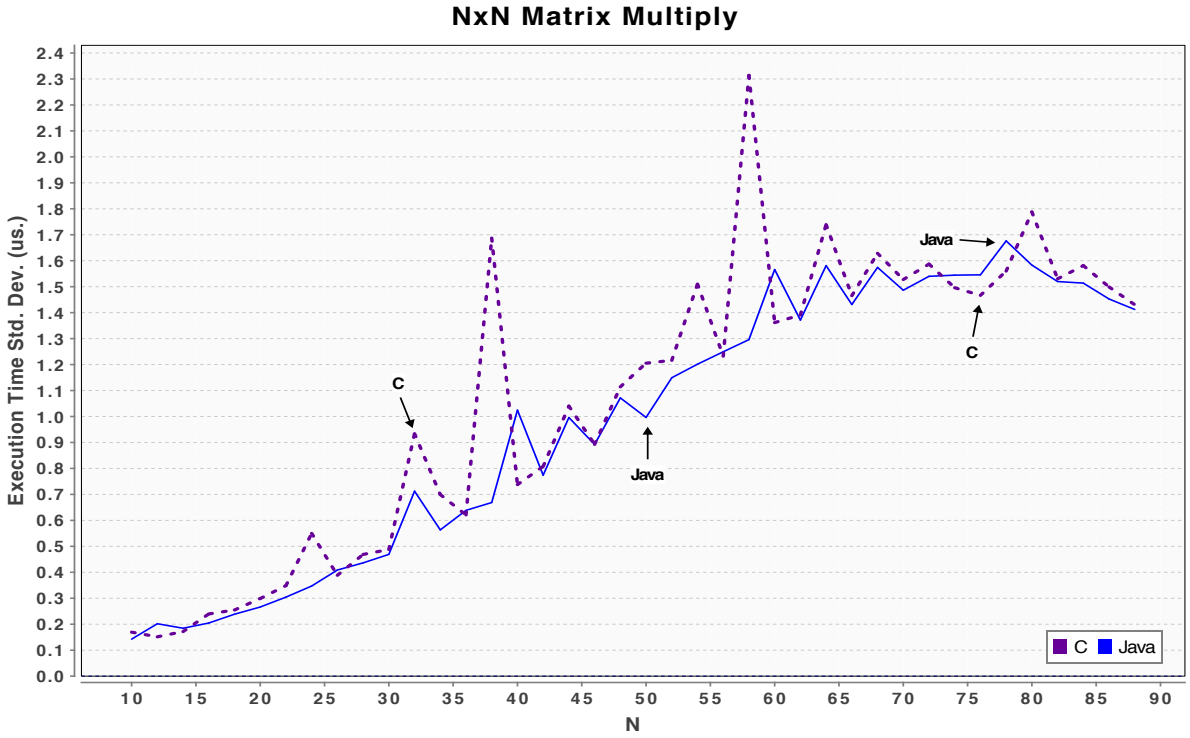

Figure 2-3. Matrix Multiplication Execution Time Jitter, C and Java

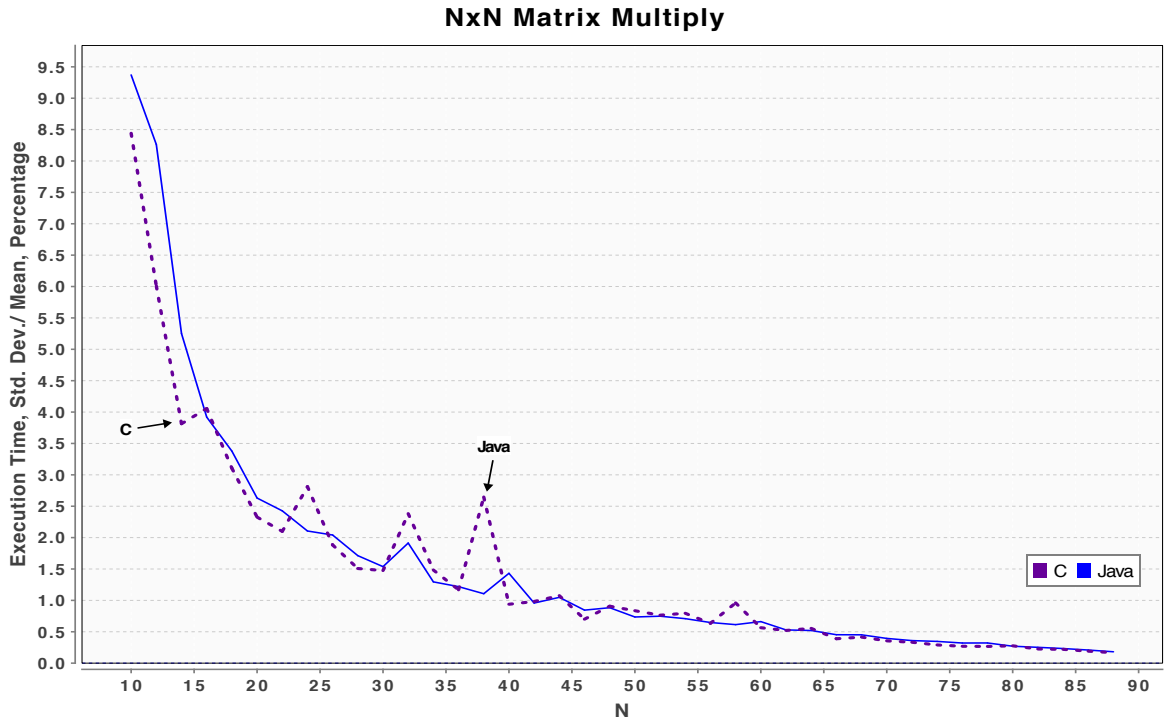

Figure 2-4 Matrix Multiplication Time, Jitter-to-Mean Percentage, C and Java

The slightly better performance of Java over C was verified on different Intel-based machines (Mac Quad-Core i7 with hypertheading[12]). The slight Java edge needs to be placed in perspective. Performance details will change with different hardware, L1 and L2 cache sizes and speeds, etc. However, one can say that in the very least the Java performance was comparable to C's. One can hypothesize that the characteristics of the code allowed Hotspot to do a very good job optimizing the Java code. This is a pure compute algorithm with the following characteristics:

- small code kernel
- loops with bounds that are fixed or easily determined by the compiler
- array indexes with simple increments with some of the increments being constant within some of the loops
- no memory allocation/deallocation within the compute kernels.

It is interesting to compare this with an algorithm in which these conditions are not as simple. The next section does this.

12 "Intel Hyper-Threading Technology." Intel Corp. <http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html>

2.6 Iterative Computation – Fast Fourier Transform

Fast Fourier Transform (FFT) code was used to compare Java and C on a compute algorithm of greater complexity than the matrix multiply of the previous section. Since the goal was to compare identical implementations, simple code that could be easily translated to multiple languages was chosen. An existing un-optimized FFT implementation from Columbia University was used and adapted to the testing. It was originally implemented in C[13] and was translated to Java. It uses complex data and implements an in-place, radix-2-decimation algorithm.

FFT transforms for data lengths from 2^6 (64) to 2^{18} (262,144) were computed. For every data length, thousands of transforms are timed to collect execution time statistics. Java and C performance is shown in Figure 2-5. Jitter is shown in Figure 2-6. For those curious and familiar with FFTs the figure also shows the results of optimized, single-threaded, Fastest Fourier Transform in the West (FFTW)[14] code on the same hardware. The FFTW helps establish the limits of the hardware but should not be compared otherwise. From the figure, it is clear that neither the C or Java FFT implementation is as efficient as FFTW.

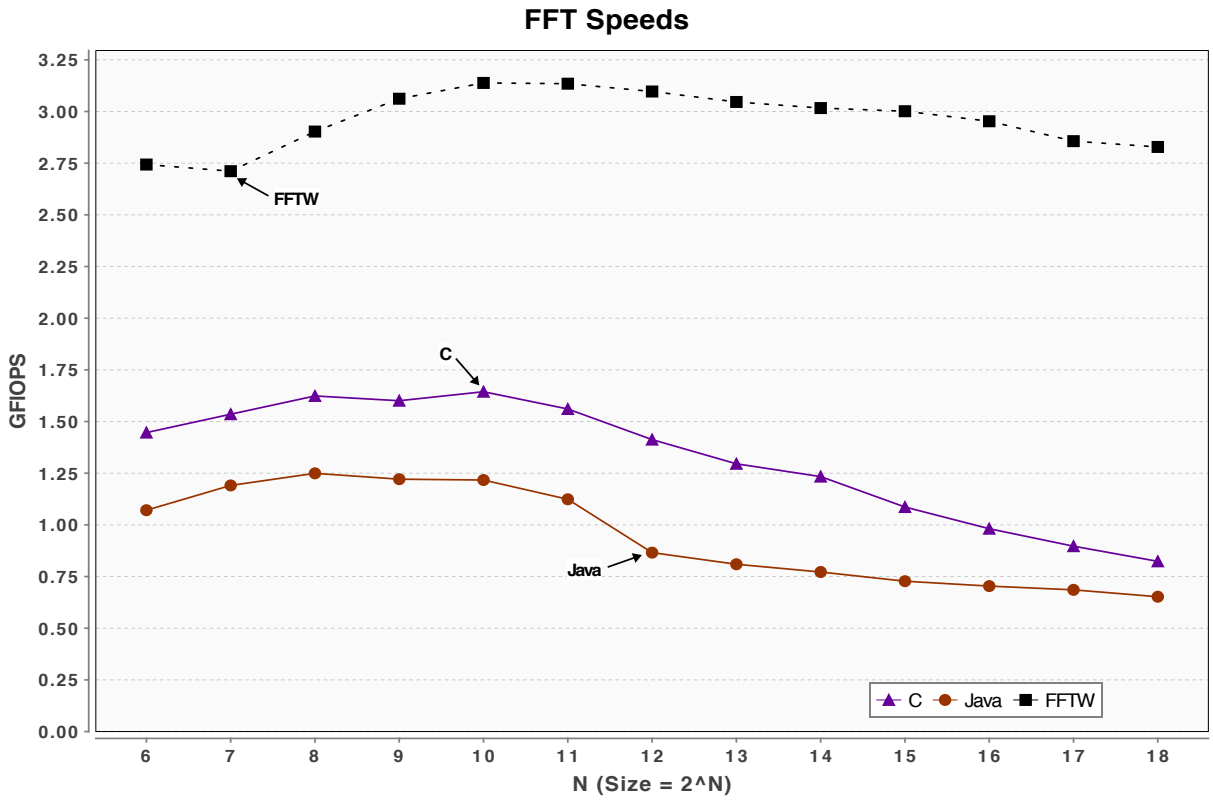

Figure 2-5. Comparison of C and Java FFT Speeds

13 "FFT.java." MEAPsoft/Columbia University <http://www.ee.columbia.edu/~ronw/code/MEAPsoft/doc/html/FFT_8java-source.html>

14 "FFTW." <http://www.fftw.org/>

The Java code executes at between 60 to 85 percent of the speed of the C code. This seems inconsistent with the results of the matrix multiply benchmark of the previous section. However, if one looks at the source code, in Listing 2-2, the characteristics of the code are different than that of the matrix-multiply:

- The code kernel is longer.
- There are more array indexes used.
- Array index increments are not sequential making them more difficult to predict by code analysis done by the Runtime.
- Non-sequential array data access
- Loop bounds are more complex functions of outer loops.
- There are more array lookups.

One of Java's safety features is that all array indexes are checked to make sure they are in range. In simple loops[15] the compiler can analyze index bounds and check them once, before the loop is executed. If the bounds cannot be determined, then Java must check each array index, which can carry a significant performance penalty. However, due to resource limitations, it was not verified how much a factor this was here.

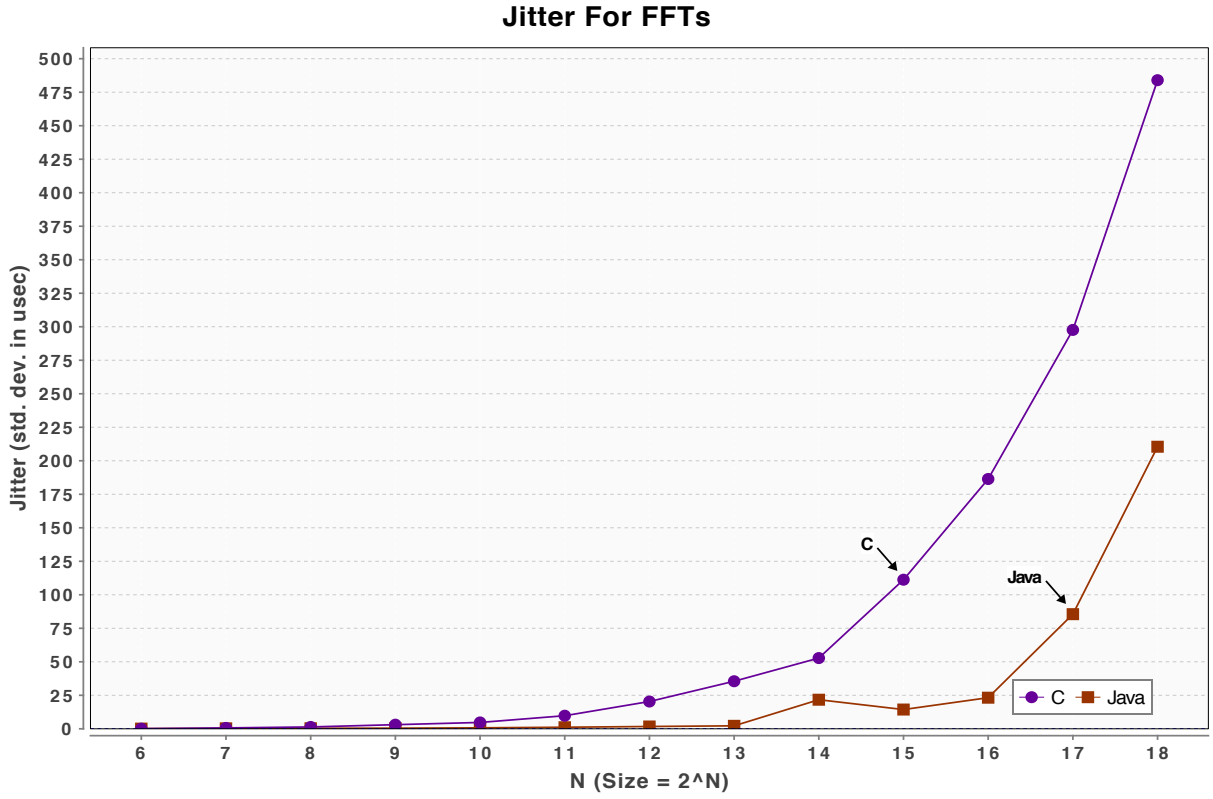

Figure 2-6. Comparison of C and Java FFT Jitter

15 Würthinger, Wimmer, Mössenböck. "Array Bounds Check Elimination for the Java HotSpot™ Client Compiler." Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, pp. 125–133.

PPPJ 2007, September 5–7, 2007, Lisboa, Portugal. <http://ssw.jku.at/Research/Papers/Wuerthinger07/Wuerthinger07.pdf>

Listing 2-2. Java FFT Kernel

```
public void fft(double[] x, double[] y) {
    { // Bit-reverse
        int j = 0;
        int n2 = n/2;
        for (int i=1; i < n - 1; i++) {
            int n1 = n2;
            while ( j >= n1 ) {
                j = j - n1;
                n1 = n1/2;
            }
            j = j + n1;

            if (i < j) {
                double t1 = x[i];
                x[i] = x[j];
                x[j] = t1;
                t1 = y[i];
                y[i] = y[j];
                y[j] = t1;
            }
        }
    }

    { // Butterflies
        int n1 = 0;
        int n2 = 1;

        for (int i=0; i < m; i++) {
            n1 = n2;
            n2 = n2 + n2;
            int a = 0;

            for (int j=0; j < n1; j++) {
                final double c = cos[a];
                final double s = sin[a];
                a +=  1 << (m-i-1);

                for (int k=j; k < n; k=k+n2) {
                    final double t1 = c*x[k+n1] - s*y[k+n1];
                    final double t2 = s*x[k+n1] + c*y[k+n1];
                    x[k+n1] = x[k] - t1;
                    y[k+n1] = y[k] - t2;
                    x[k] = x[k] + t1;
                    y[k] = y[k] + t2;
                }
            }
        }
    }
}
```

2.7 Bit Twiddling

The previous two sections presented quantitative comparisons of Java and C on pure-compute algorithms working from memory. This section looks at bit-twiddling in Java and C without explicit use of low-level bit-set and bit-clearing machine instructions. This means using And,

Or, Exclusive-Or, and Negation operations that operate on integer variables in order to test or change the state of a single bit.

A bit-twiddling-intensive benchmark was designed based on generating linear feedback shift-register[16] (LFSR) sequences and well known polynomials[17] [18]. On each iteration of the test, integer sequences derived from polynomials of orders 3 to 19 are generated. These correspond to sequence lengths from 7 to 524,287. The array to hold the sequence is preallocated so that the benchmark timing is not affected by memory allocation.

Figure 2-7 shows the results. The height of the bar indicates the average run time. The standard deviation is overlaid over the bars. The Java version runs almost 30 percent faster. As a related study, the code was modified to see how much of the measured times were due to the various loops that wrap the actual bit-twiddling kernel. Listing 2-3 shows the contents of the `fill()` method that actually computes the LFSR values while Listing 2-4 shows an alternative "trivial" `fill()` method that skips the shifts, ANDs, and XORs. Figure 2-8 compares the performance for "trivial" filling. The Java code is 4 percent faster for the trivial fill.

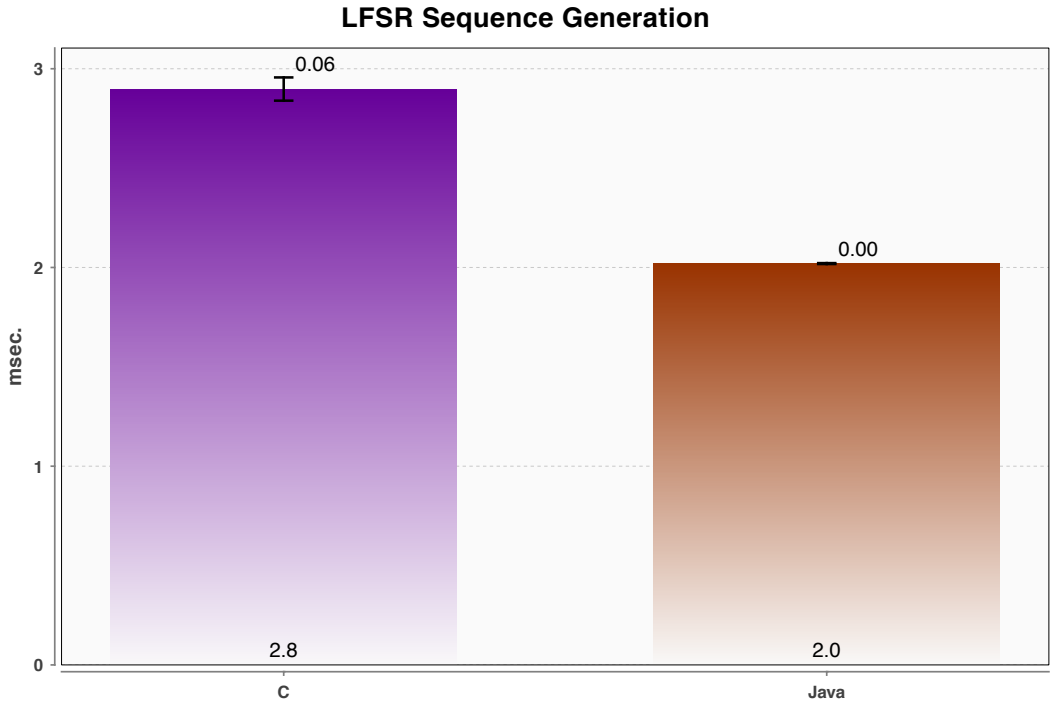

Figure 2-7. LFSR Sequence Generation Comparison

16 "Linear feedback shift register." Wikipedia. <http://en.wikipedia.org/wiki/Linear_feedback_shift_register>

17 Alfke, Peter, "Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators." XILINX Inc. <http://www.xilinx.com/support/documentation/application_notes/xapp052.pdf>

18 "Linear Feedback Shift Registers." New Wave Instruments. <http://www.newwaveinstruments.com/resources/articles/m_sequence_linear_feedback_shift_register_lfsr.htm>

Listing 2-3. Kernel for Generating Linear Recursive Sequences

```
private void fill() {
    int poly = 0;
    int[] taps = coefs[order];

    for (int i : taps) {
        poly |= (1 << (i-1));
    }

    int lfsr = 1;
    int i = 0;

    do {
        if ((lfsr & 1) == 1){
            lfsr = (lfsr >>> 1) ;
            lfsr ^= poly;
        }
        else {
            lfsr = (lfsr >>> 1);
        }
        values[i] = lfsr;
        i += 1;
    }
    while(i < length);
}
```

Listing 2-4. Kernel for Simple Filling

```
private void fill() {
    int i = 0;
    do {
        values[i] = i;
        i += 1;
    }
    while(i < length);
}
```

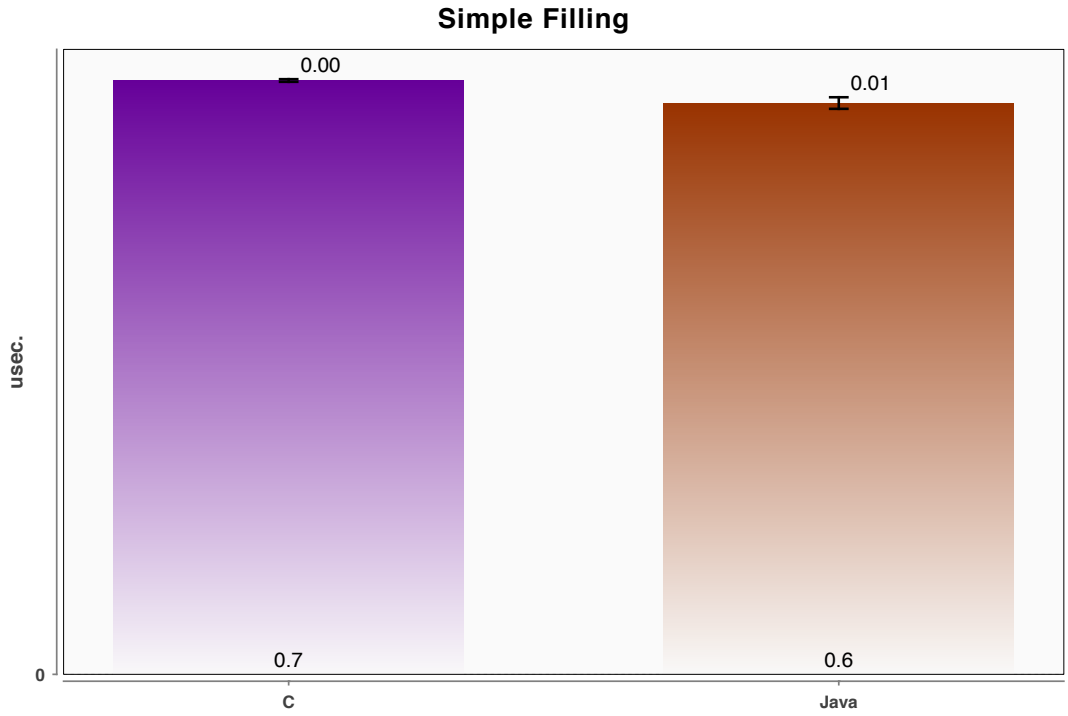

Figure 2-8. Trivial Sequence Generation Comparison

The greater improvement of the LFSR kernel suggests that the gain measured for the Java implementation is actually due to improved execution of the bit-twiddling parts of the code, not just the test-jig's looping code. Of course, it should be possible to match and exceed Java's performance in this test with tuned C code. But the fact that "generic" Java code can match C's performance is significant. One should not conclude that Java is always faster in this kind of code. On different hardware, the actual relative performance will depend on the detailed interaction of the CPU's integer logic units, address pre-fetch logic, and memory speeds. This investigator's conclusion is only that the Java performance is comparable.

2.8 In-Memory Data Structure Creation

A benchmark was designed to compare C and Java on an algorithm that involved creating memory structures and "walking" through the structures repeatedly. The algorithm chosen was a red-black tree structure. The red-black tree is a binary search tree that maintains a mapping of keys to values that are sorted by key. As keys are added the old structure must be traversed to find the right place to insert new key-value pairs. Occasionally portions of the structure must be rebalanced.

An existing C implementation of a red-black tree by Emin Martinian[19] was chosen. The C code was written to handle keys and values of any type and length. To accommodate keys of any type and length, the algorithm allows the user to provide a comparator function that matches the key type. For this test 4-byte integer keys were used. The algorithm was translated as directly as possible to Java. A test driver was written to pseudo-randomly generate keys and use them for

19 Martinian, Emin, "Red-Black Tree C Code." <http://web.mit.edu/~emin/www.old/source_code/red_black_tree/index.html>

insertion and retrieval. On a given run, the values have a fixed length that can be changed from one run to the next. Lengths of 4 bytes, 512 bytes, and 2048 bytes were used on different runs. On retrieve, the test uses the same sequence of pseudo-random keys. This means that all fetched keys are known to be present in the tree.

The results of the insertion tests are shown in Figure 2-9. There are three C insertion timings shown. The first includes the time to allocate the tree, the keys, and the data. The second adds to this the time to free the tree memory between repeated trials. With Java one cannot control when garbage collection is taking place so the Java performance line includes memory reclamation. The third uses `calloc` instead of `malloc` to take into account the time to clear the content before use. Java always clears the content of data as a side-effect of allocation. Collectively, the three timings help to compare C to the equivalent Java timing.

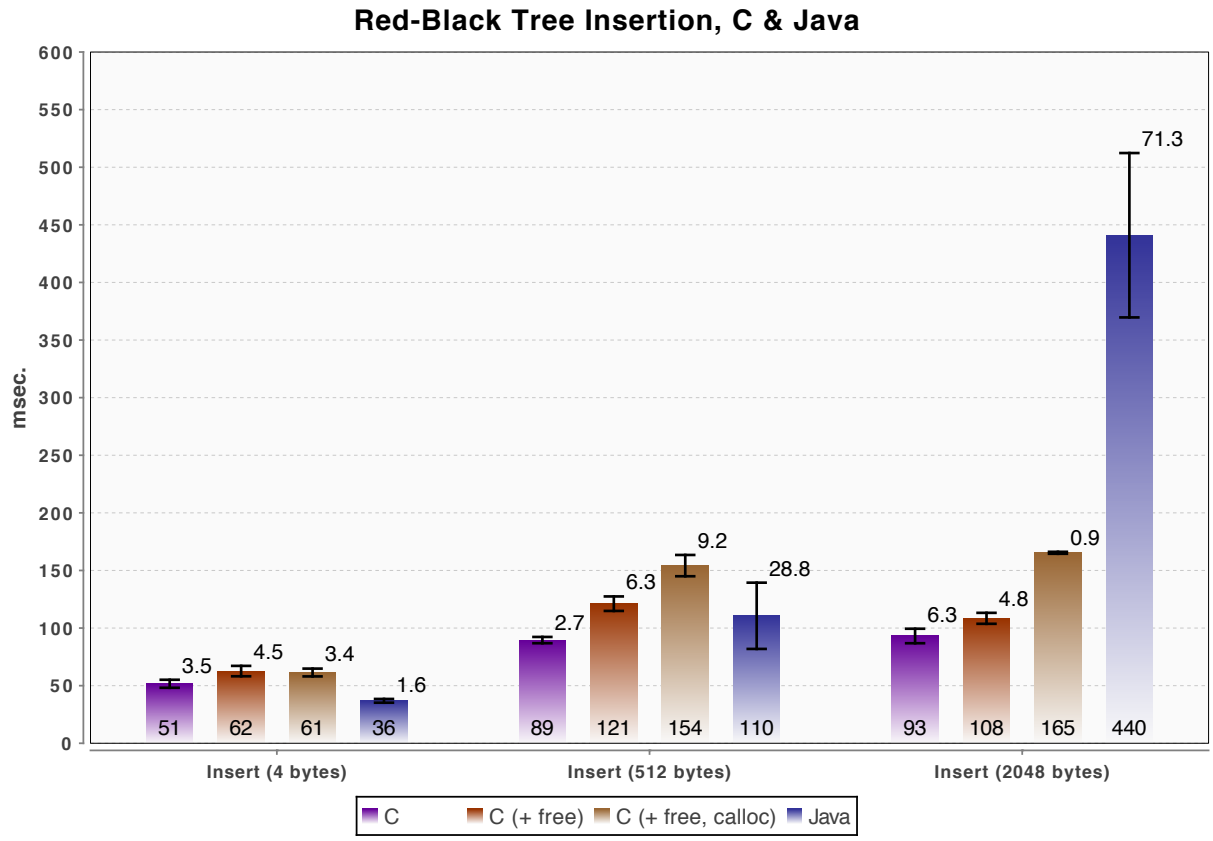

Figure 2-9. Insertion Into a Red-Black Tree

The "values" that are inserted into the tree are integer arrays with a size fixed for each run of the test. One can see that Java performance is comparable to C's for the 4 and 512-byte cases. For the 2048-byte case the Java implementation performs poorly. Since each test run uses identical sequences of keys, the difference in the Java execution time seems to be due to a length-dependent overhead for memory allocation and reclamation.

Comparing the insertion test "C +free" time to the Java time shows that Java was 42 percent faster with 4-byte values, 1 percent faster with 512-byte values, and three times slower with 2048-byte values.

The results of the retrieval tests are shown in Figure 2-10. Interestingly, Java was about 5 percent slower for the two smaller value sizes, but 18 percent faster for the large value size.

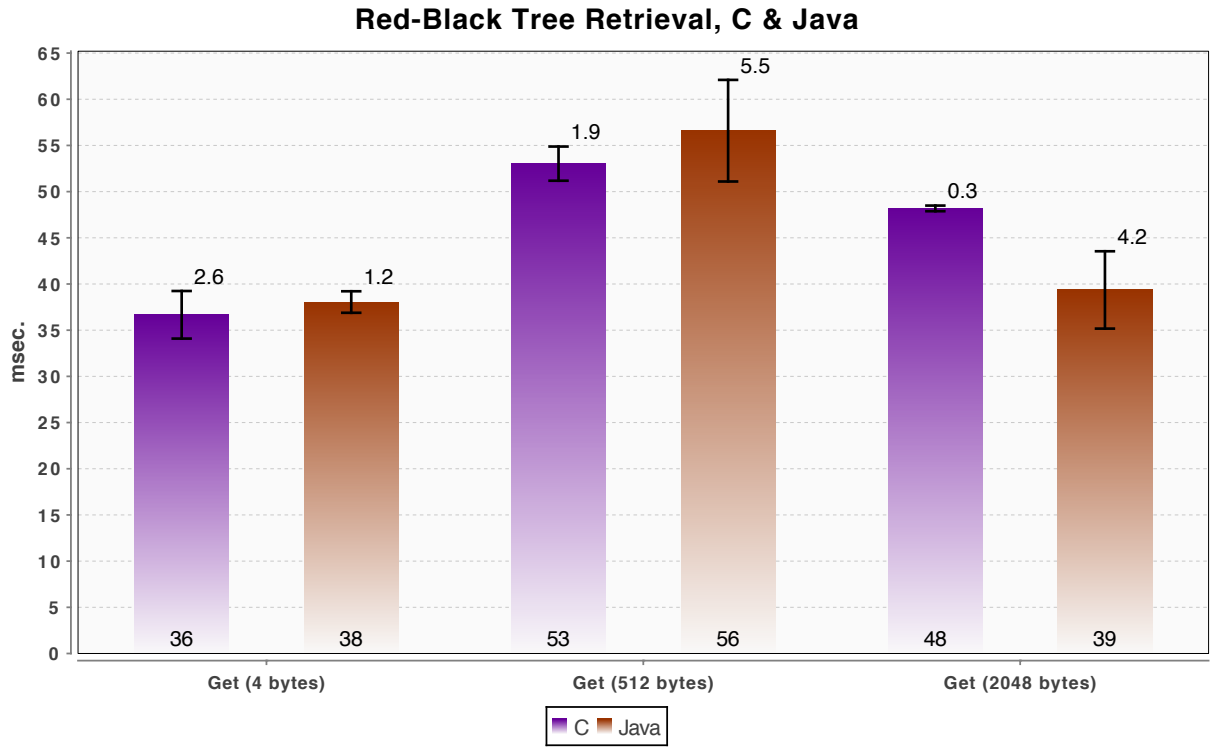

Figure 2-10. Retrieval From a Red-Black Tree

2.9 Java Overhead

This section attempts to quantify how much processor power is consumed by the JVM and how much of that overhead is due to garbage collection. This can be done for any specific case but one cannot generalize the findings because so much depends on what the application is doing. Notwithstanding the caution, tests were designed for two cases. One case has no dynamically allocated memory. The other case has a lot of dynamically allocated and freed memory and forces a lot of garbage collector activity by picking small Java heap sizes to force frequent garbage collection.

This report will not try to describe all the Java memory and garbage collector options or quantify their effects. There are many sources with information on tuning the garbage collector.[20] [21] [22] [23] [24]

---

20 (Java 6) "Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning." Oracle Technology Network. <http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html>

21 (Java 7) "The Garbage-First Garbage Collector." Oracle Technology Network. <http://www.oracle.com/technetwork/java/javase/tech/g1-intro-jsp-135488.html>

22 (Java 7) Bruno, Eric J., "G1: Java's Garbage First Garbage Collector," <http://www.drdobbs.com/jvm/g1-javas-garbage-first-garbage-collector/219401061>

23 Jiva, Azeem, 2009, "Easy Ways to do Java Garbage Collection Tuning." AMD Developer Central. <http://developer.amd.com/Resources/documentation/articles/pages/4EasyWaystodoJavaGarbageCollectionTuning.aspx>

24 Carr, Sean. "Adventures in Java Garbage Collection Tuning." <http://blog.rapleaf.com/dev/2011/12/07/adventures-in-java-garbage-collection-tuning/>

Also, note that command-line tuning flags[25] [26] [27] [28] depend on the JVM provider. Furthermore, algorithms used by the garbage collectors can change significantly as each new major version (such as Java 7) is released. Of course, the best way to reduce garbage collector impact is to avoid creating garbage when possible.

In a Java application that does not have a graphical user interface, the application's thread(s) and the JVM garbage collector are the biggest users of processor time. There are other JVM threads but they consume little time. If there are sufficient CPU cores then much of the garbage collector's work will not interfere with the work of the main threads. If one only measures the performance of the main thread(s), one does not get a true measurement of the overhead.

This study measures overhead as the percentage in increased time to complete a compute workload. In the tests developed here, the number of threads used to perform the Java work is a command line parameter. The number of cores used by the Java application can also be limited at run time by using Linux's "taskset" command.

Performance was measured for a no-memory allocation case and for a "heavy" memory allocation/deallocation case. The no-memory allocation test performed 120,000 iterations of an 8192-point FFT using the pure-Java FFTs used in other tests above. The heavy memory allocation performed 400 iterations of the red-black tree insertion test. For that case, a Java heap size was chosen (though experimentation) to result in about 7.5 garbage collector events per second. Note that the insert test generates garbage both by discarding duplicate keys and by completely discarding the full tree after every iteration.

The results are shown in Table 2-2. There was a small and insignificant difference between the two and three-thread versions of the no-memory test. Without memory allocation (and without a graphical user interface), Java overhead was zero. The memory intensive test suggests a 15 percent overhead.

---

25 "Java HotSpot VM Options." Oracle Technology Network. <http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html#G1Options>

26 "JVM command-line options." IBM User Guide for Java v7 on Linux. <http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/index.jsp?topic=%2Fcom.ibm.java.lnx.70.doc%2Fdiag%2Fappendixes%2Fcmdline%2Fcommands_jvm.html>

27 Bailey, Gracie, Taylor. "Garbage collection in WebSphere Application Server V8, Part 1: Generational as the New Default Policy." IBM developerWorks. Jun 2011. <http://www.ibm.com/developerworks/websphere/techjournal/1106_bailey/1106_bailey.html>

28 Biron, Benjamin, and Ryan Sciampacone. "Real-time Java, Part 4: Real-time Garbage Collection." IBM developerWorks. May 2007. <http://www.ibm.com/developerworks/java/library/j-rtj4/>

Table 2-2. JVM Overhead in Memory Allocation Intensive Test

Case (All use two Compute Threads)	Memory Intensive Test	No Memory Test
2 Cores	178 sec. (15%)	155.79 sec. (0%)
3 Cores	155 sec.	155.70 sec.

2.10 Summary of Java On a Single Core

Four types of algorithms that were initially implemented in C were converted to Java. The matrix multiply has nested loops with small compute bodies and sequential array indexes. The FFT has medium sized loops, but non-sequential array indexes. The LFSR sequence requires integer-based bit masking and Boolean operations. The red-black tree sorting is highly recursive and requires significant dynamic object creation and cleanup.

Once warmed, the Java matrix multiply ran a little faster than the C version that was compiled with a GNU compiler at its highest level of optimization. The Java FFT performance was 20 to 40 percent slower than C's. This may be due to the more complex and non-sequential use of array indexes as compared with the matrix multiplication. The Java version of the LFSR sequence generation ran 30 percent faster than the C version.

The first three tests required no memory allocation for computation. In contrast, the red-black tree-sorting test required heavy use of memory allocation to allocate space for keys and values to be inserted into the tree. Between iterations, the entire tree and its contents are disposed of requiring the garbage collection system to work to recover unused memory. In this test, Java was a little faster for insertion and a little slower than C for retrieval. There also seemed to be object-size dependence. The faster insertion speed degraded to poorer as the allocation size increased.

The fact that sometimes the Java is faster and sometimes slower suggests that:

- Hardware architecture, such as cache size, instruction pipeline depths, number and throughput of floating point versus integer arithmetic logic units can significantly impact these findings.
- The designer should test the application kernels to really know what yields the best performance for the application. On the other hand, it should not be assumed that C code will always be the best.
- Even if the heavy computation is done in C, incidental computation might be handled by Java. In the end, the bigger consideration may be whether the data is located in Java or on the native side.

Overall, these findings show that performance of Java is comparable to C's—after Java warm-up. This opens up a lot of options for mixing Java and C.

This page intentionally left blank.

3 Java Parallelism

A compute solution running on a multicore machine will want to concurrently utilize multiple cores when possible. Java has provided standard language constructs and library support for this since Java 1.0. However, the developer trying to implement a "simple" parallel for-loop will find that there does not exist a simple syntax for doing this in Java like the one that exists in C using OpenMP™ [29]. OpenMP "extends" the C language by utilizing pragma statements that allow OpenMP-aware compilers to distribute the for-loop workload among the available cores. The pragma statements look like C comments. Compilers that are not OpenMP capable will compile the for-loops normally within a single thread.

The Java Specification Request (JSR) 166 [30] will simplify parallel execution of code blocks in Java 8. The JSR does not address parallel loops. Instead it provides changes that allow collections of various sorts to parallelize themselves and apply functions to their members.

The Java 7 developer has a few alternatives for parallelism. This study compares the performance of three Java 7 alternatives for parallelism. The study also compares the performance of two additional third-party, open source alternatives.

3.1 Methodology for Testing Parallelism

The concept of a parallel for-loop was used to design a simple workload to execute in a parallelized manner. The goal was to determine how efficiently Java can distribute a pure compute workload among the available CPU cores. A compute workload that takes a certain amount of time on a single core would ideally take one-fourth the amount of time on four cores. In reality, there is a small time penalty for breaking up a compute task into multiple independent cores. There is also a penalty for waiting for cores to complete and then aggregating the results. Small workloads may not be worth breaking up for this reason. Large workloads may only approach the ideal gain asymptotically.

A variably sized workload was designed so that the time to complete the work with different levels of parallel execution could be measured. The work can be performed in-line or divided on up to four threads on the test system. The workload is shown in Listing 3-1. Each test is performed multiple times. Every iteration executes four operations—loop increment, loop test, division, and accumulation.

---

29 OpenMP is a registered trademark of the OpenMP Architecture Review Board. See the OpenMP specification at <http://openmp.org/wp/openmp-specifications/>

30 "JSR 166: Concurrency Utilities." Java Community Process. <http://jcp.org/en/jsr/detail?id=166>

Listing 3-1. Worker Task

```
static double sumStartLength(int startIndex, int length){
double cum = 0;
final int endIndex = startIndex+length;
for (double i = startIndex+1; i<=endIndex; i++){
cum += 1.0/i;
}
return cum;
}
```

The sections below present the results of these tests. On the Linux system used in these tests, there appeared to be adverse interactions between the application threads, management threads, and the operating system scheduling of threads leading to "thread-thrashing" and irregular execution time. 4 to 6 provide examples of this.

4 is an example in which, as the workload increases, there are "spikes" of poor performance. Figure 3-2 is an example showing overall improved performance with additional threads, but an irregular fluctuation in the execution time as the workload increases. Finally, Figure 3-3 shows reduced execution time for one and two threads as expected but then increased execution time with three threads. The four-thread performance line is not much better than the two-thread line either.

There were various techniques tested for concurrency (described in the following section). The thrashing was not consistent. Sometimes a technique "afflicted" with poor behavior under one version of Linux would be okay under a different version of the Linux kernel. In the end, the investigator found that the best way to get consistent behavior was to run using Linux round-robin scheduling at real-time priorities, even when the total workload kept the CPU cores less than fully busy.

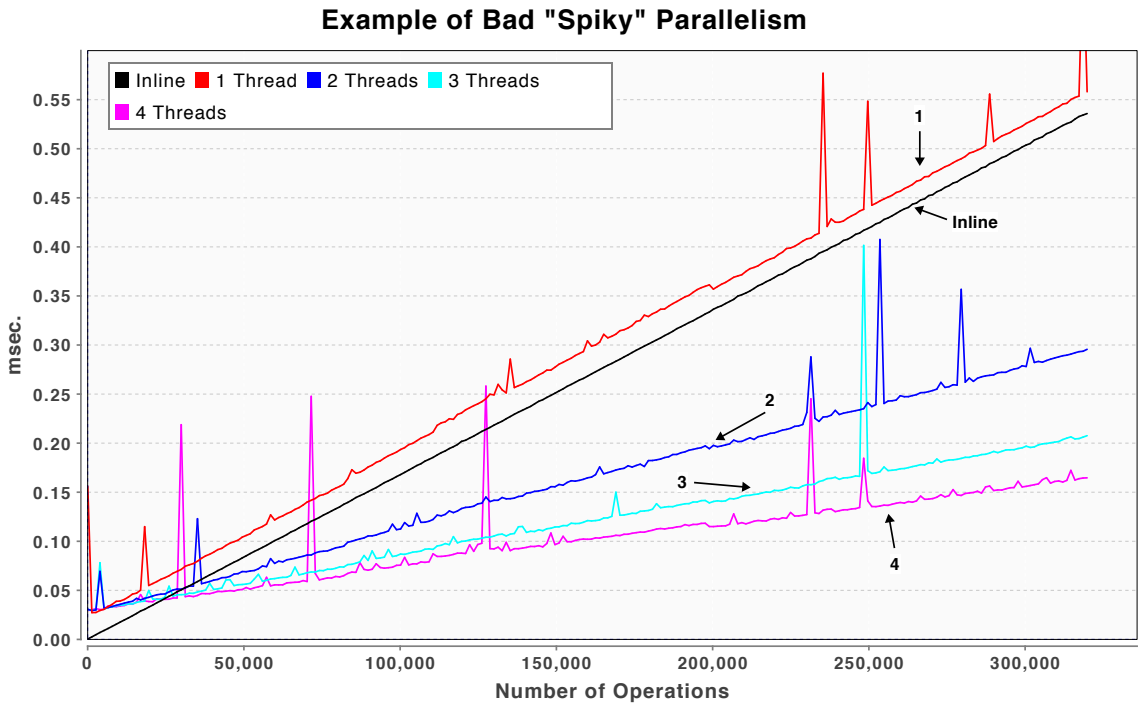

Figure 3-1. Poor Parallelism Behavior Example 1

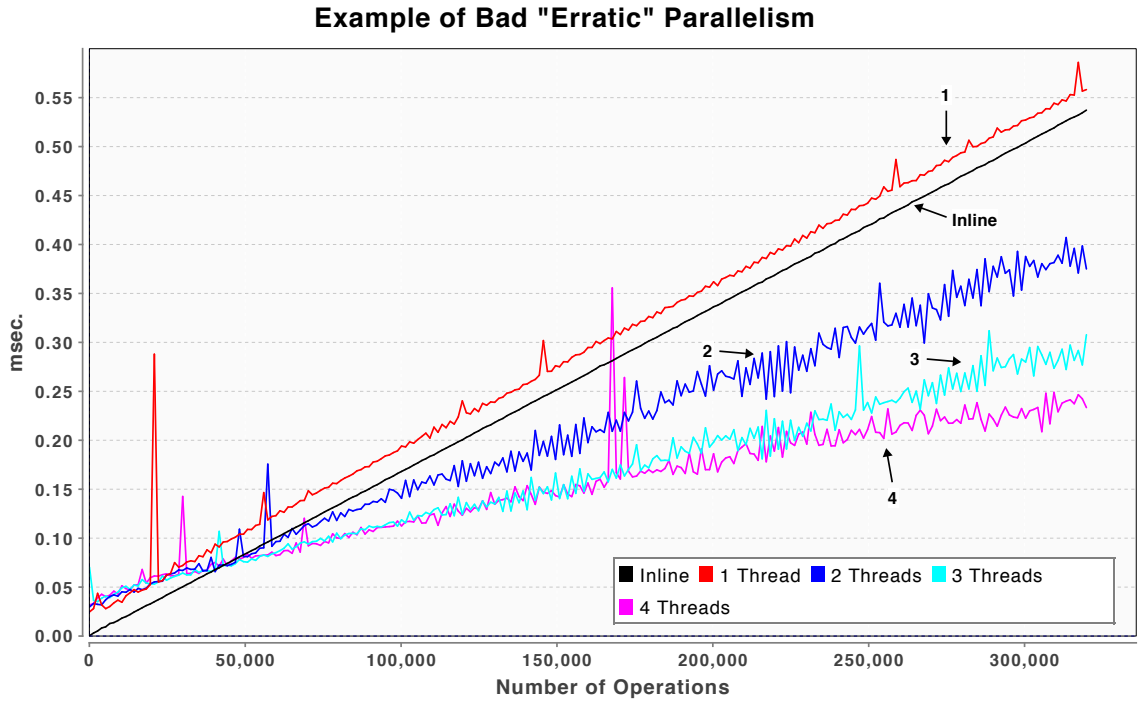

Figure 3-2. Poor Parallelism Behavior Example 2

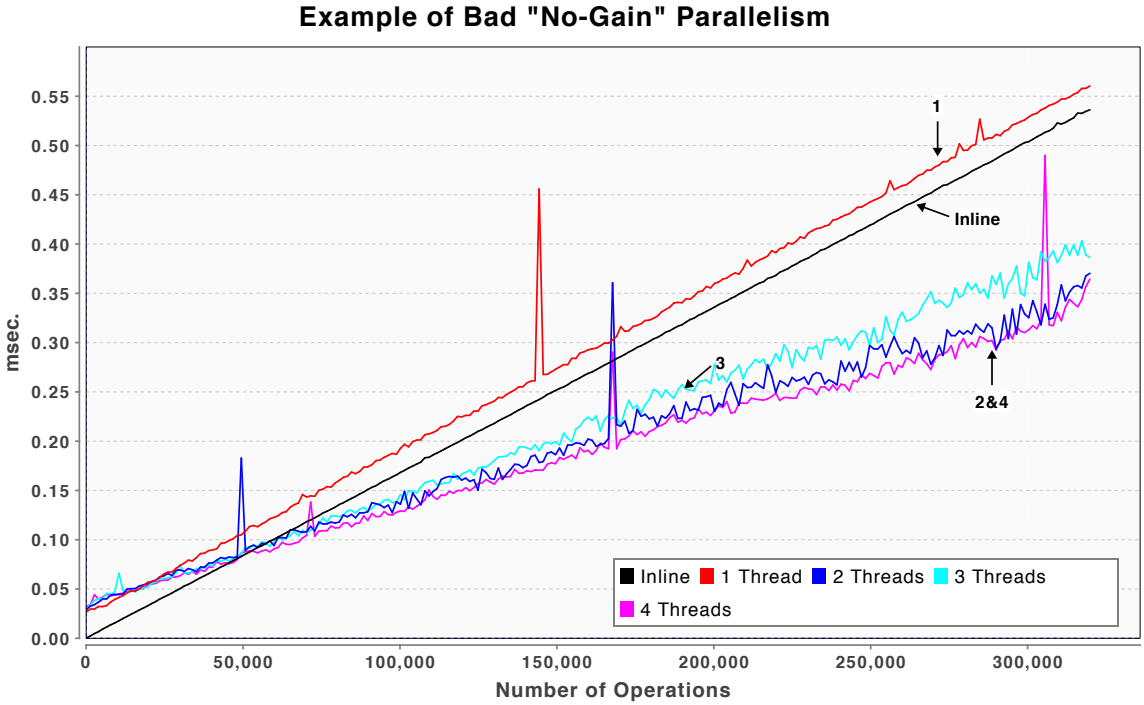

Figure 3-3. Poor Parallelism Behavior Example 3

3.2 Parallelism at "Half" Load With Linux Real-Time Scheduling

3.2.1 Parallelism with Timers

The `java.util.Timer` and the `java.util.TimerTask` can be used for concurrency. These have been available since Java 1. The example in Listing 3-2 shows one way to break up a "for-loop" onto multiple timer threads using a `CountDownLatch` to coordinate on work completion. The example:

- uses as many Timers as there are available processors
- creates a `CountDownLatch` used to wait for all threads to complete
- declares `final` outer loop variables that are referenced within the inner `Runnable` so the value can be accessed from a different thread
- signals when each inner thread is complete
- has the outer thread wait for the inner thread completion.

Listing 3-2. Example of Parallel Execution Using Timer and CountDownLatch

```
import java.util.Timer;
import java.util.TimerTask;
import java.util.concurrent.CountDownLatch;

class TimerExample {

    final int nThreads = Runtime.getRuntime().availableProcessors();
    private Timer[] timers;

    /*
     * Create the Timers
     */
    private void init(){
        timers = new Timer[nThreads];
        for (int n = 0; n < n; n++){
            timers[n] = new Timer("Timer"+n,true);
        }

    }

    void work() throws InterruptedException {
        int startIndex = 0;
        int loopLength = 1000;
        // this increment is not exactly correct, but ok for illustration
        final int increment = loopLength/nThreads ;

        final CountDownLatch latch = new CountDownLatch(nThreads);
        for (int j = 0; j < nThreads; j++) {
            // Can't access the non-final field inside the inner class.
            final int _start = startIndex;

            timers[j].schedule(new TimerTask()  {
                @Override
                public void run()  {
                    doWorkFragment(_start, increment);
                    latch.countDown();
                }
            }, 0);
            startIndex += increment;
        }
        latch.await();
    }

    void doWorkFragment(int startIndex, int numberOfIterations) {
        // work done here
    }

}
```

The approach is cumbersome due to the details that must be coded. One detail not shown in the example is ensuring that the initial count for the `CountDownLatch` is correct even if there are runtime errors that might prevent one of the inner `Runnable` instances from being created.

In the example, a timer thread is used for every fragment of work to be performed. It is not necessary to have as many timer threads as there are available processors. The invoking thread will be idle while waiting for the worker threads to complete. It should be more efficient if the main thread handles the final work fragment, avoiding overhead for dispatching and waiting on

the additional thread. Thus, for a desired parallelism of N only, N-1 timer threads need to be used. However, the more efficient code (not shown) is also a little more complex.

Figure 3-4 and Figure 3-5 plot the execution time versus the workload size using the N-1 and the N timer strategy respectively. There is not a significant difference between these except that with the N timer plot one can clearly see the overhead of parallelism as the separation between the in-line and the 1-thread curves. However one would not use either scheme unless one was interested in at least two-way parallelism. Table 3-1 lists some of these crossover points.

The jitter plots are shown in Figure 3-6 and Figure 3-7. There are small, though, insignificant differences.

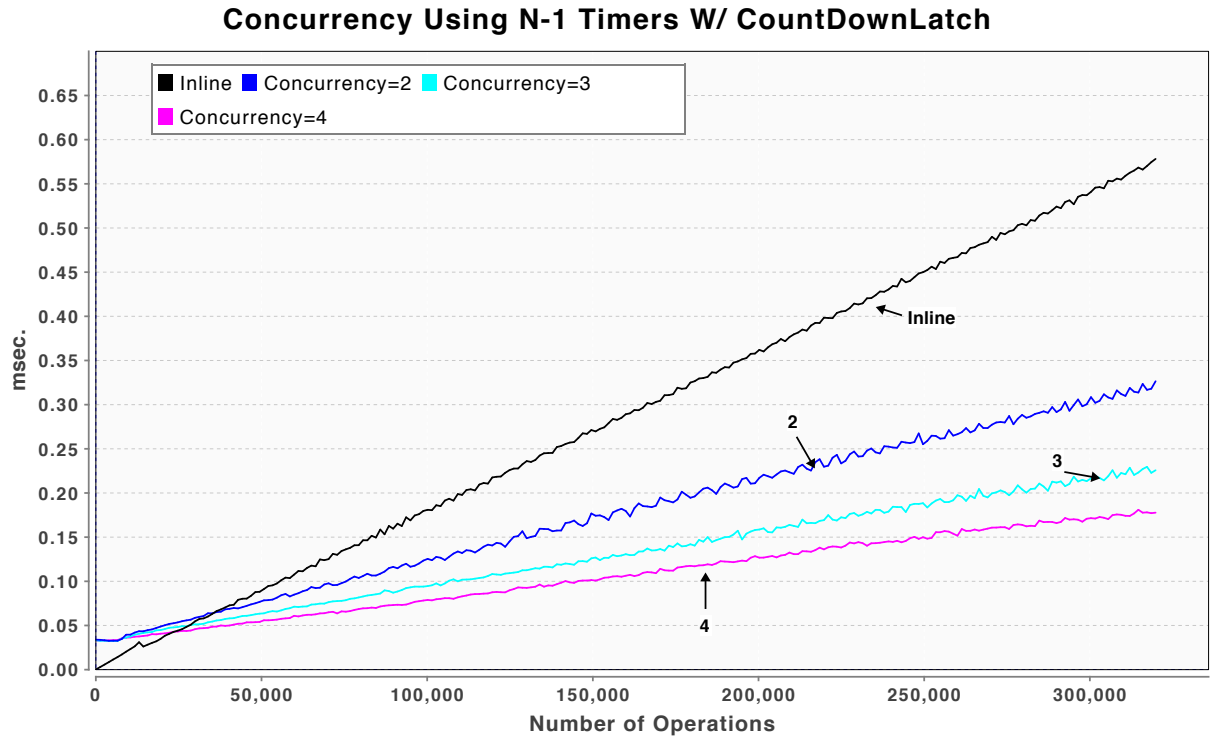

Figure 3-4. The TimerTask with CountDownLatch and N-1 Threads

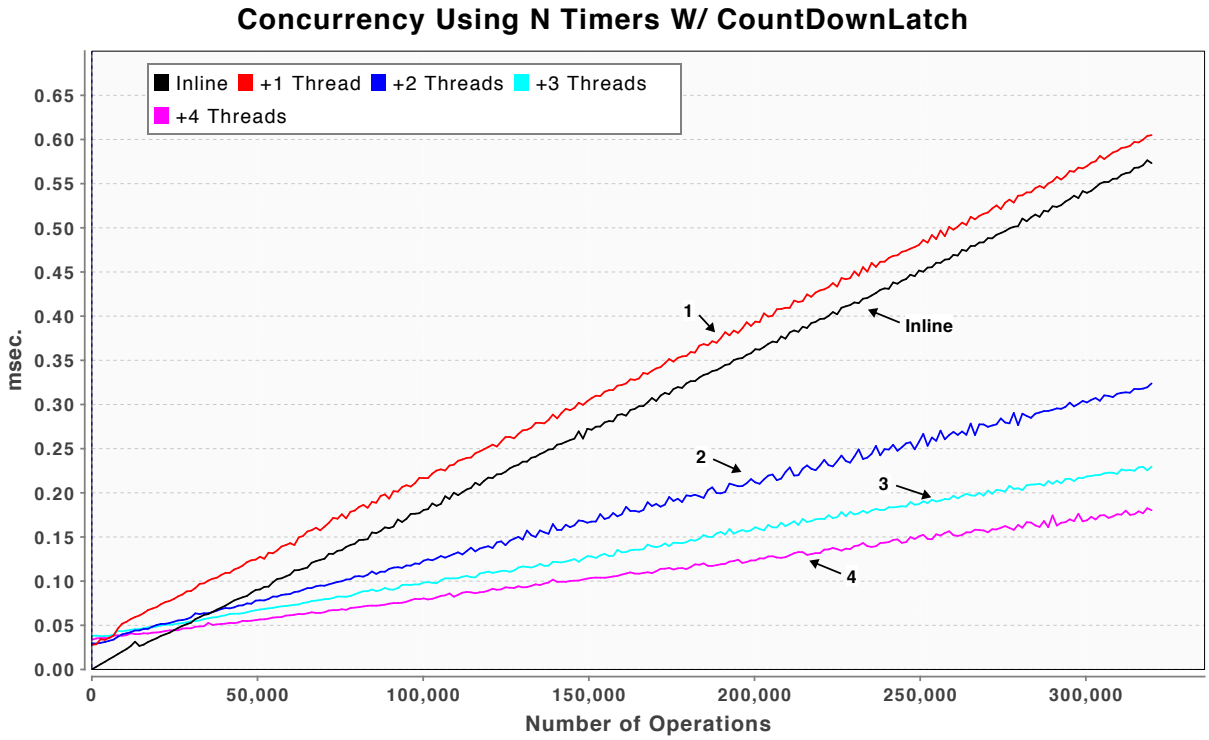

Figure 3-5. The TimerTask with CountDownLatch and N Threads

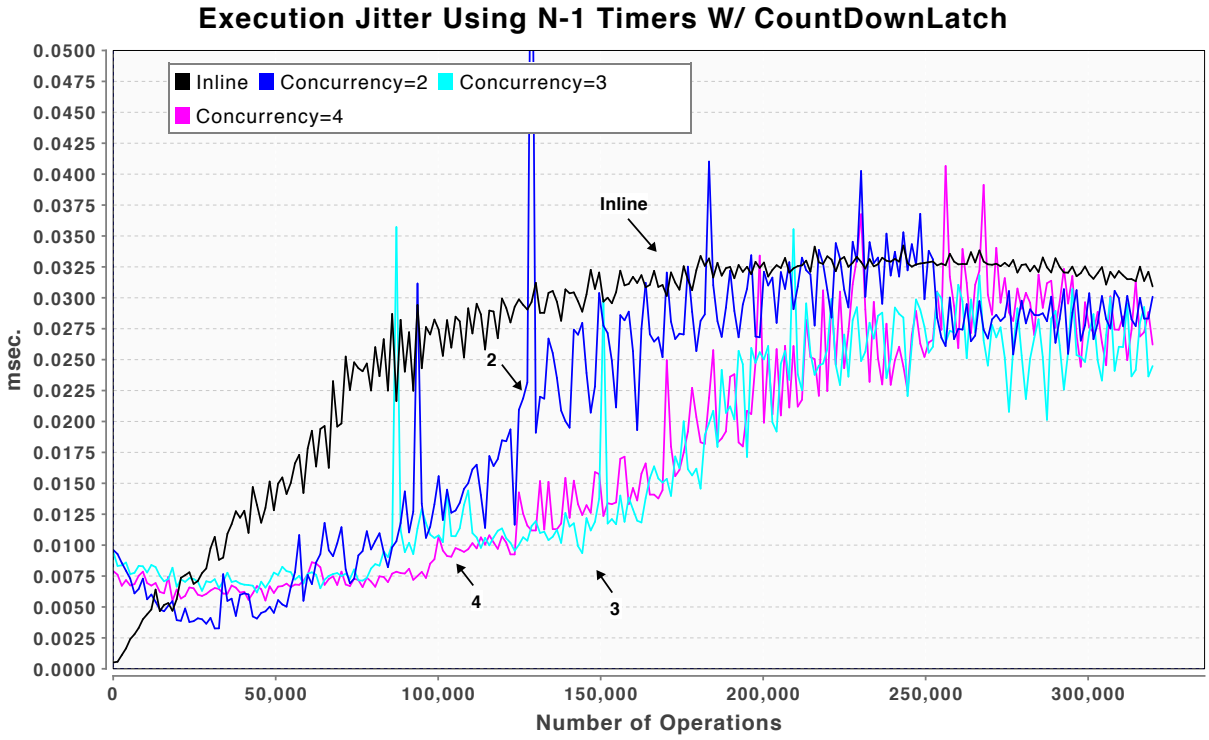

Figure 3-6. Execution Jitter Using N-1-Timer Based Concurrency

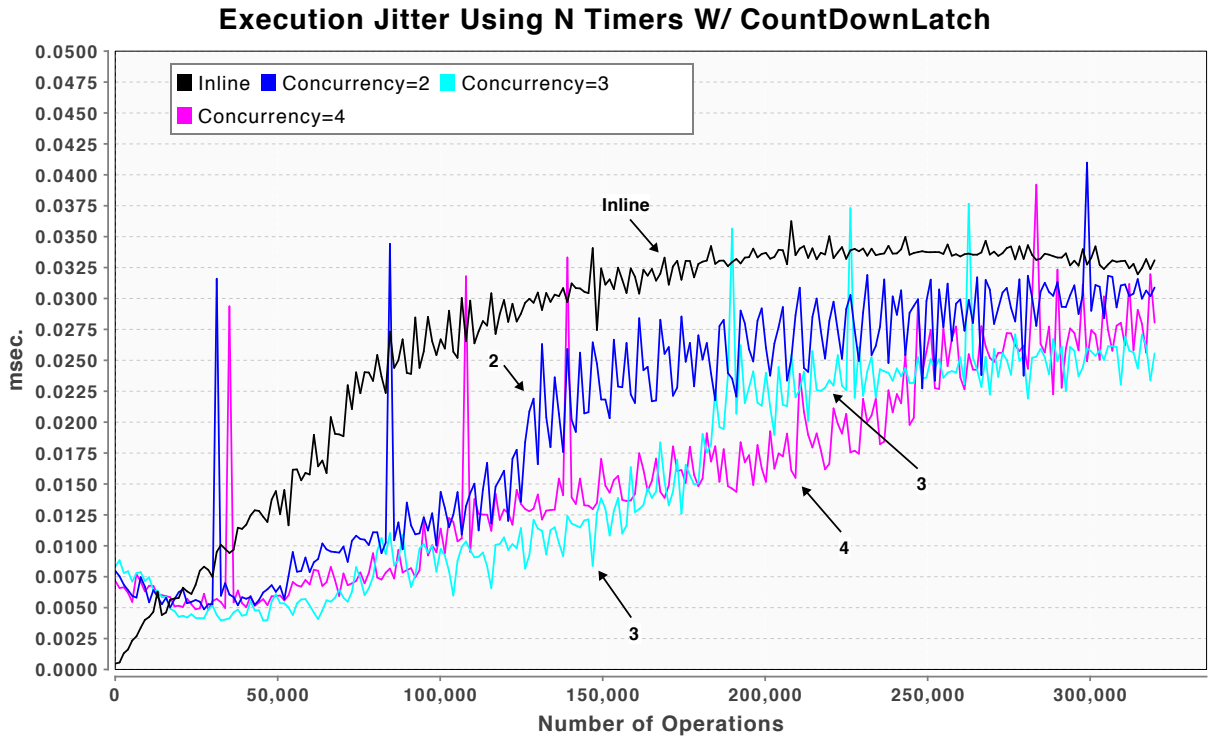

Figure 3-7. Execution Jitter Using N-Timer Based Concurrency

3.2.2 Parallelism Utilizing ExecutorService Threads

Another option for distributing work among multiple cores is to use the `java.util.concurrent.ExecutorService`. By comparison with the `java.util.Timer` the executor service is a little easier to use because the `invokeAll()` method provides an easier way to start and wait for all tasks to finish. An example is shown in Listing 3-3. There is also the option to submit work fragments individually using the `submit()` method and use `CountDownLatch` to wait for completion. This was tested but found to be slower than using the `invokeAll()` method.

Listing 3-3. Example of Parallel Execution Using the ExecutorService

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class ExecServiceExample {

    final int nThreads = Runtime.getRuntime().availableProcessors();
    final ExecutorService threads = Executors.newFixedThreadPool( nThreads );
    final Collection<Callable<Object>> tasks = new ArrayList<Callable<Object>>(nThreads);

    void work() throws InterruptedException, BrokenBarrierException {

        int startIndex = 0;
        int loopLength = 1000;
        // this increment is not exactly correct, but ok for illustration
        final int increment = loopLength/nThreads ;
        tasks.clear();
        for (int j = 0; j < nThreads; j++) {
            // Can't access the non-final field inside the inner class.
            final int _start = startIndex;
            tasks.add(new Callable<Object>()  {
                @Override
                public Object call()  {
                    doWorkFragment(_start, increment);
                    return null;
                }
            });
            startIndex += increment;
        }
        threads.invokeAll(tasks);
    }

    void doWorkFragment(int startIndex, int numberOfIterations) {
        // work done here
    }
}
```

Figure 3-8 shows the results of using the `ExecutorService` with `invokeAll()`. Figure 3-9 shows the execution jitter. Table 3-1 shows the normalized execution times. Comparing this to the timer-based concurrency shows that the timers have slightly less scheduling overhead. The point where the two, three, and four-way concurrency lines cross the inline-line is representative of the overhead.

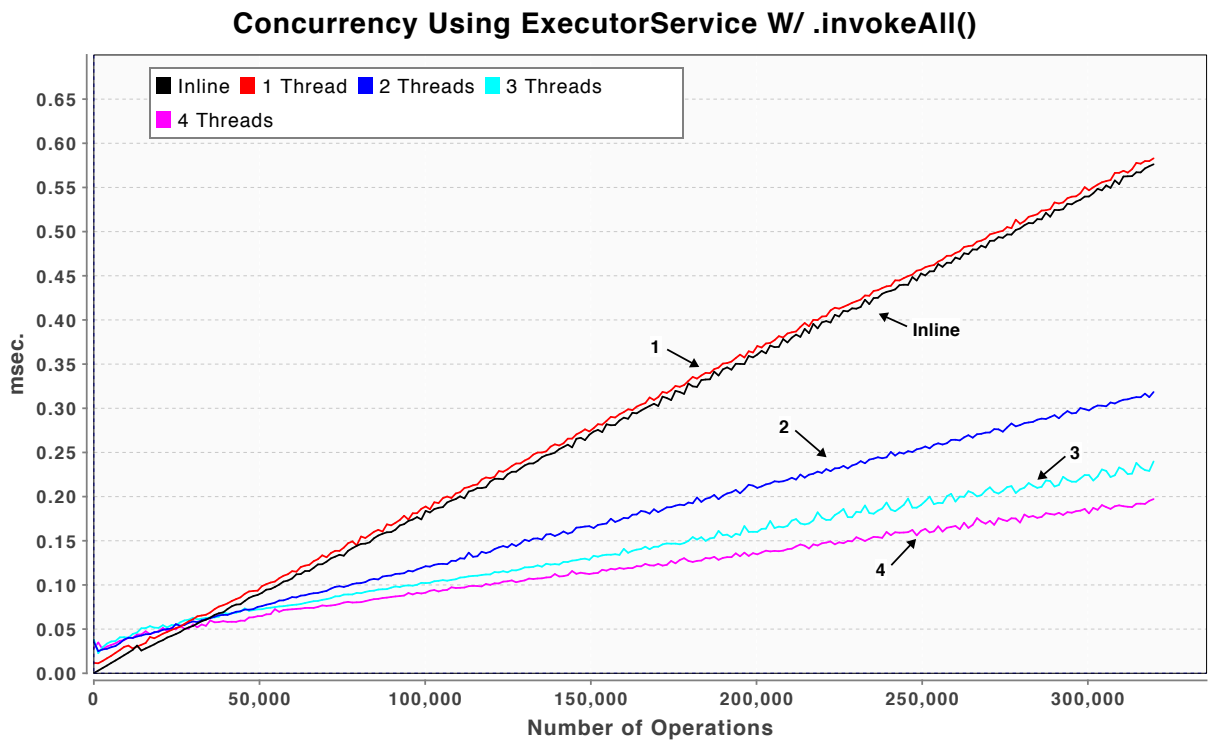

Figure 3-8. Concurrency Using the ExecutorService

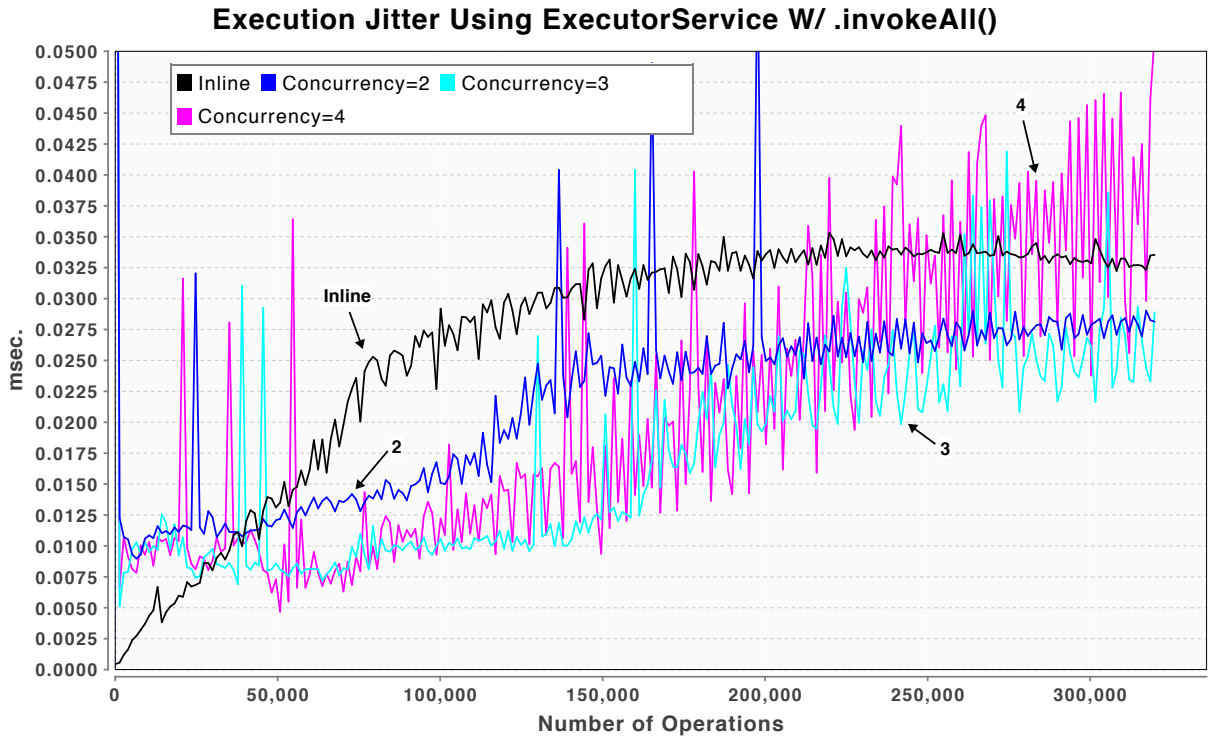

Figure 3-9. Execution Jitter Using the ExecutorService

3.2.3 Parallelism Using Java 7 Fork/Join

Java 7 has added support for parallelism that has similarities to the fork/join directives in Unix. However, the Unix fork/join results in creating Unix processes while the Java fork/join does not. In Java 7, the fork/join mechanism creates lightweight tasks that are run on a `ForkJoinPool`. Listing 3-4 is an example of how this works.

The `ForkJoinPool` parallelism performance plots are shown in Figure 3-10 and Figure 3-11. Table 3-1 lists the normalized execution times. Overall, the performance is roughly comparable to that of the `ExecutorService` though there is a slightly higher overhead leading to slightly greater execution time for the four-thread line. However, the fork/join mechanism provides a great deal of flexibility that is not easily achieved with the `ExecutorService`. The `ForkJoinPool` may be overkill when parallelizing for-loops but it is well suited to more complex "divide-and-conquer" compute algorithms.

One note regarding `ForkJoinPool` is that some have found that it does not scale well on machines with many cores[31]. Doug Lea has noted the problem and published scaling updates to the pool beyond the Java 7 implementation[32]. These are available as JSR-166y. That version of the `ForkJoinPool` was tested but not found to offer significant performance improvements for concurrency for only a few cores. For this reason only data for the standard Java 7 `ForkJoinPool` is presented below.

[31] Scalability of Fork Join Pool, < http://letitcrash.com/post/17607272336/scalability-of-fork-join-pool >

[32] Doug Lea, ForkJoin updates, < http://cs.oswego.edu/pipermail/concurrency-interest/2012-January/008987.html >

Listing 3-4. Example of Parallel Execution Using Java 7 Fork/Join

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

class ForkJoinExample {

    int nThreads = Runtime.getRuntime().availableProcessors();
    ForkJoinPool forkJoinPool = new ForkJoinPool(nThreads);
    List<RecursiveAction> forks = new ArrayList<RecursiveAction>(nThreads);

    void work(){
        int startIndex = 0;
        int totalIterations = 1000;
        // this increment is not exactly correct, but ok for illustration
        final int increment = totalIterations/nThreads ;
        forkJoinPool.invoke(new RecursiveAction()  {
            int baseIndex = 0;
            @Override
            protected void compute() {
                forks.clear();
                for (int j = 0; j < nThreads; j++) {
                    final int startIndex = baseIndex;
                    RecursiveAction subComputation = new RecursiveAction(){
                        @Override
                        protected void compute() {
                            doWorkFragment(startIndex, increment);
                        };
                    };
                    forks.add(subComputation);
                    baseIndex += increment;
                }
                invokeAll(forks);
            }
        });

    }

    void doWorkFragment(int startIndex, int numberOfIterations) {
        // work done here
    }
}
```

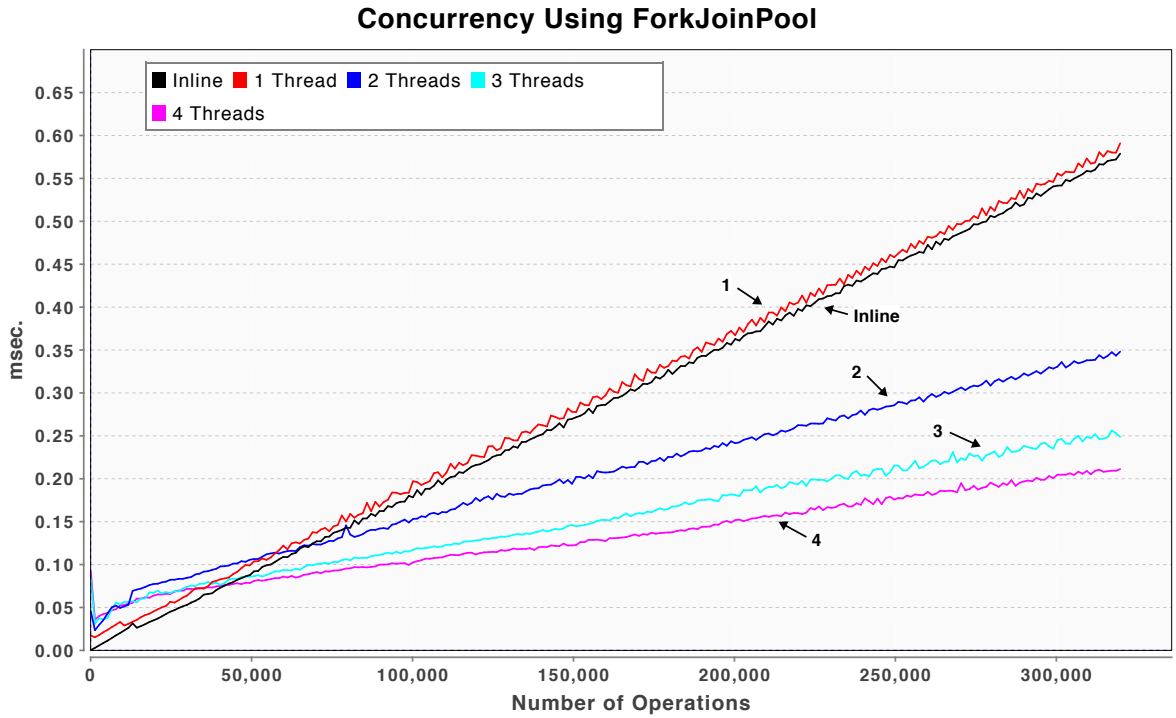

Figure 3-10. Concurrency Using the ForkJoinPool

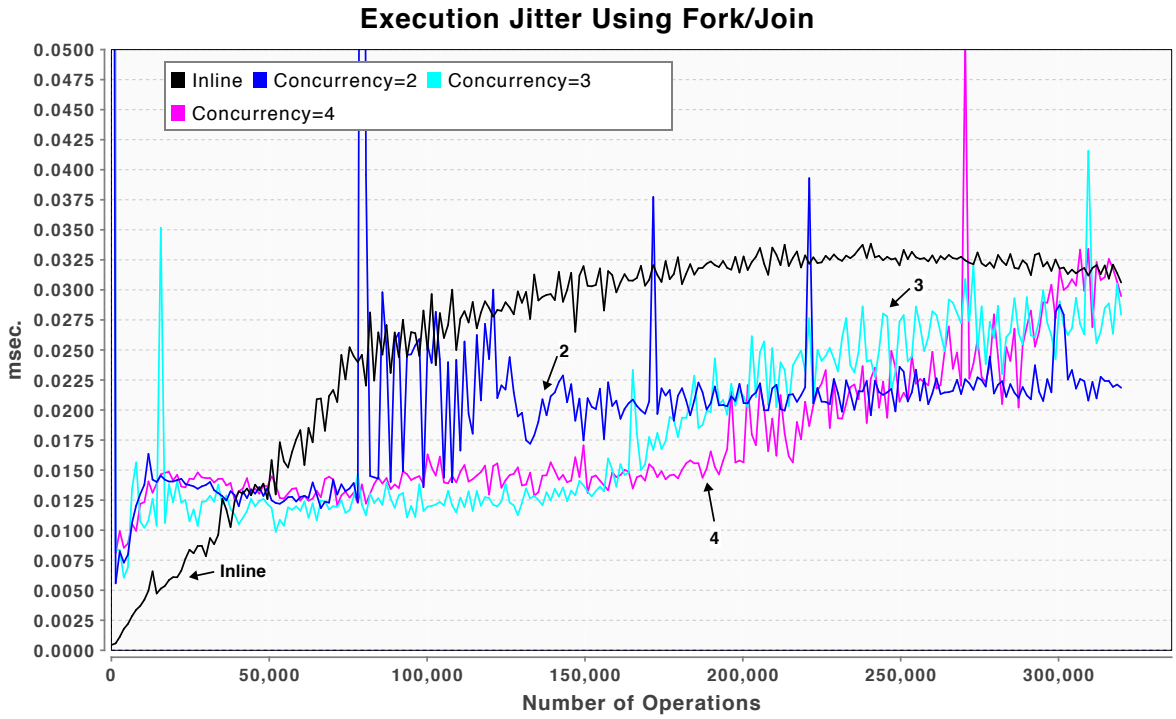

Figure 3-11. Execution Jitter Using ForkJoinPool Concurrency

3.2.4 Parallelism Using the Parallel Java Library

The Parallel Java[33] (PJ) library was developed as an all-Java library for parallel programming on a single node or on multiple nodes. PJ was developed by Alan Kaminsky and Luke McOmber at the Rochester Institute of Technology.

PJ provides a number of classes that simplify the syntax and legibility of parallel code. An example for executing a parallel for-loop is shown in Listing 3-5. There are a variety of constructs with similarities to concepts found in OpenMP™. There are `ParallelTeam`, `ParallelRegions`, `ParallelSections`, `ParallelForLoops`, and `ParallelIterations`. See the PJ documentation for their descriptions. Performance is shown in Figure 3-12 and jitter is shown in Figure 3-13. Normalized execution time is listed in Table 3-1. Of all the concurrency mechanisms described so far, this has the best performance. This also requires the fewest lines of code.

Listing 3-5. Example of Parallel Execution Using the Parallel Java Library

```
import edu.rit.pj.IntegerForLoop;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelTeam;

class ParallelJavaExample {

    ParallelTeam parThreads =
            new ParallelTeam(Runtime.getRuntime().availableProcessors());

    int loopIndexBegin = 0;
    int loopIndexEnd = 99;

    void work() throws Exception {

        parThreads.execute(new ParallelRegion() {
            @Override
            public void run() throws Exception {
                execute(loopIndexBegin, loopIndexEnd, new IntegerForLoop() {
                    @Override
                    public void run(int first, int last) {
                        // work for a range of loop indexes done here
                    }
                });
            }
        });
    }
}
```

[33] Kaminsky, Alan. "Parallel Java Library." Rochester Institute of Technology. <http://www.cs.rit.edu/~ark/pj.shtml>

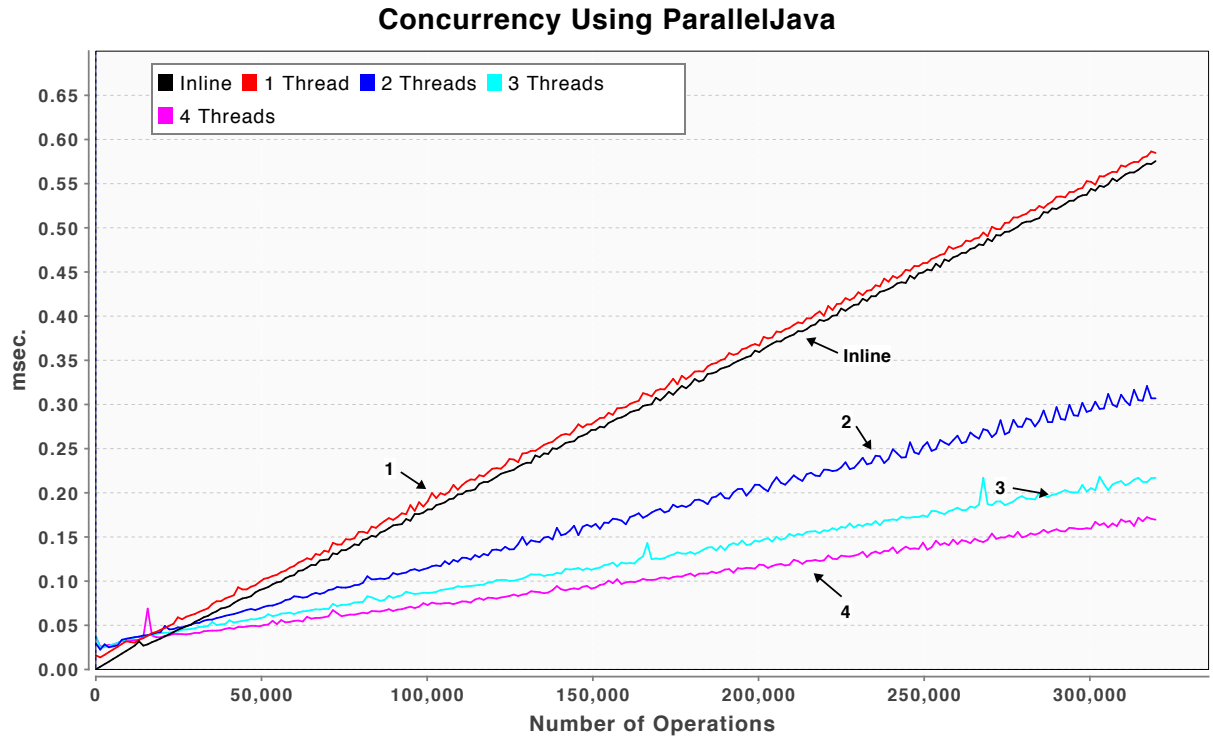

Figure 3-12. Concurrency Using the Parallel Java Library

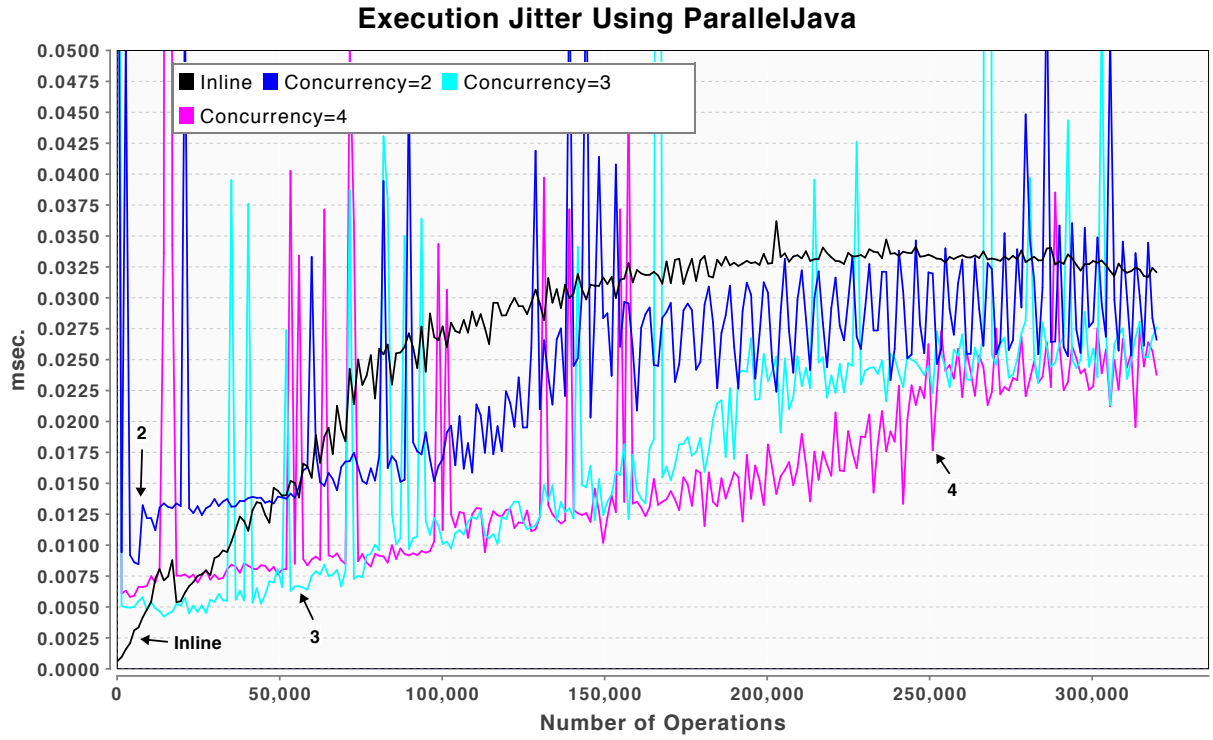

Figure 3-13. Execution Jitter Using the Parallel Java Library

3.2.5 Parallelism Using the Javolution Library

The Javolution[34] library is an open-source library developed for real-time and embedded systems. It was developed by Jean-Marie Dautelle to provide deterministic execution times. It has facilities for concurrent execution of code blocks and facilities to facilitate interfacing with C. It also provides Java structures such as Maps and Lists that were designed to minimize memory allocation, which should reduce garbage collector interruption.

An example using Parallel Java is shown in Listing 3-6. Benchmark performance is shown in Figure 3-14 and jitter in Figure 3-15. Normalized execution time is shown in Table 3-1. In this test, Javolution exhibited the smallest two-way crossover point making it better suited than alternative approaches for parallelizing small workloads.

Listing 3-6. Example of Parallel Execution Using the Javolution Library

```
import javolution.context.ConcurrentContext;

public class JavolutionExample {

    void work(final int nThreads, final int length) throws InterruptedException {
        // This increment may be off a little for
        // the last thread but is good enough for this example
        final int increment = length/nThreads;
        ConcurrentContext.enter();
        try {
            int baseIndex = 0;
            for (int j = 0; j < nThreads; j++) {
                final int startIndex = baseIndex;
                ConcurrentContext.execute( new Runnable(){
                    @Override
                    public void run() {
                        doWorkFragment(startIndex, increment);
                    };
                });
                baseIndex += increment;
            }
        }
        finally {
            ConcurrentContext.exit();
        }
    }

    void doWorkFragment(int startIndex, int numberOfIterations) {
        // work done here
    }
}
```

[34] "Javolution | The Java Solution for Real-Time and Embedded Systems." <http://javolution.org/>

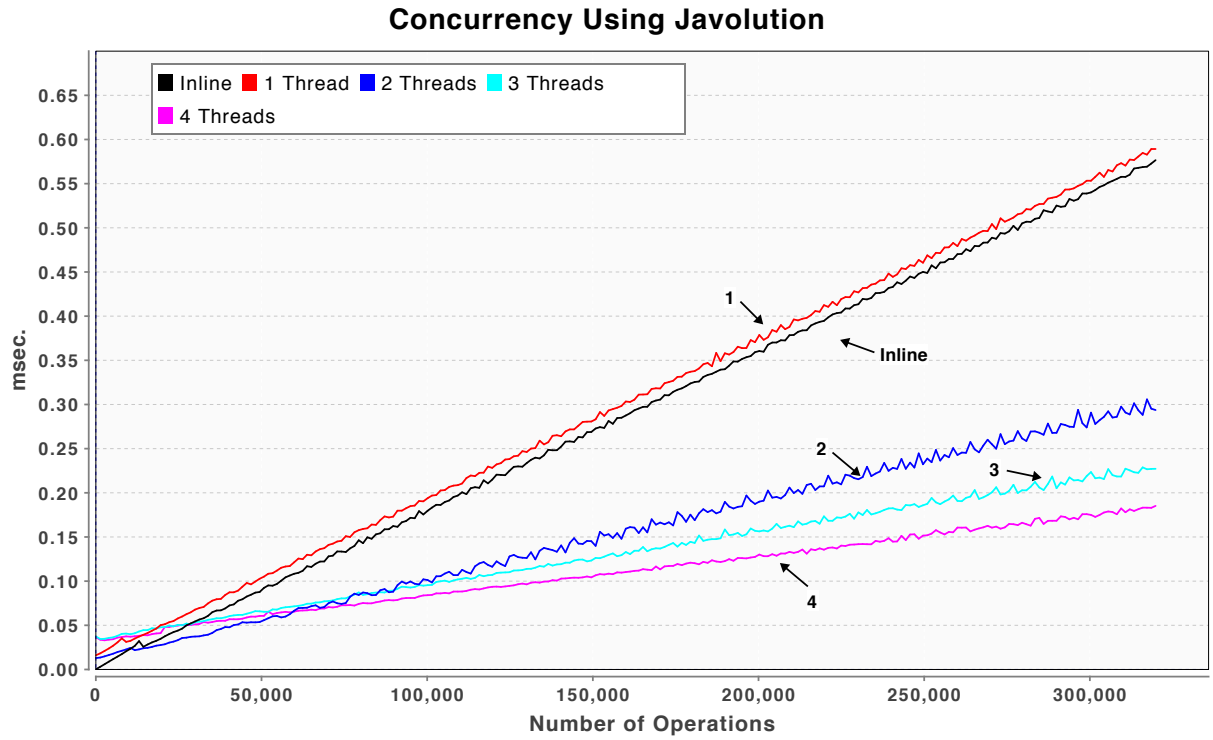

Figure 3-14. Concurrency Using the Javolution Library

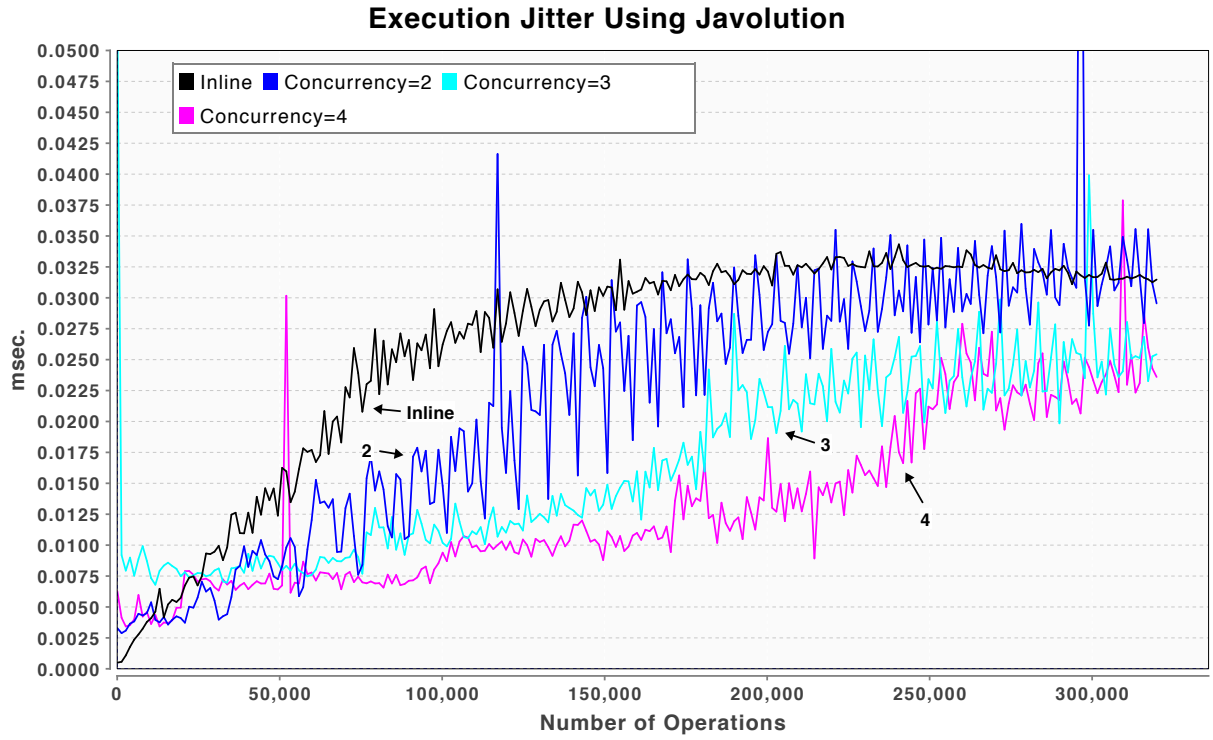

Figure 3-15. Execution Jitter Using the Javolution Library

3.2.6 C Parallelism Using OpenMP

For comparison, Figure 3-16 and Figure 3-17 show the performance and jitter of C code. The approach is the same, namely running at approximately half load by yielding one millisecond of time every millisecond and running the process at a Linux real-time level of fifty. Interestingly, the no-concurrency jitter is bigger than Java's. The two, three, and four-way concurrency jitter is much less—approximately 0.002 milliseconds (ms) for C versus approximately 0.0325 ms for Java.

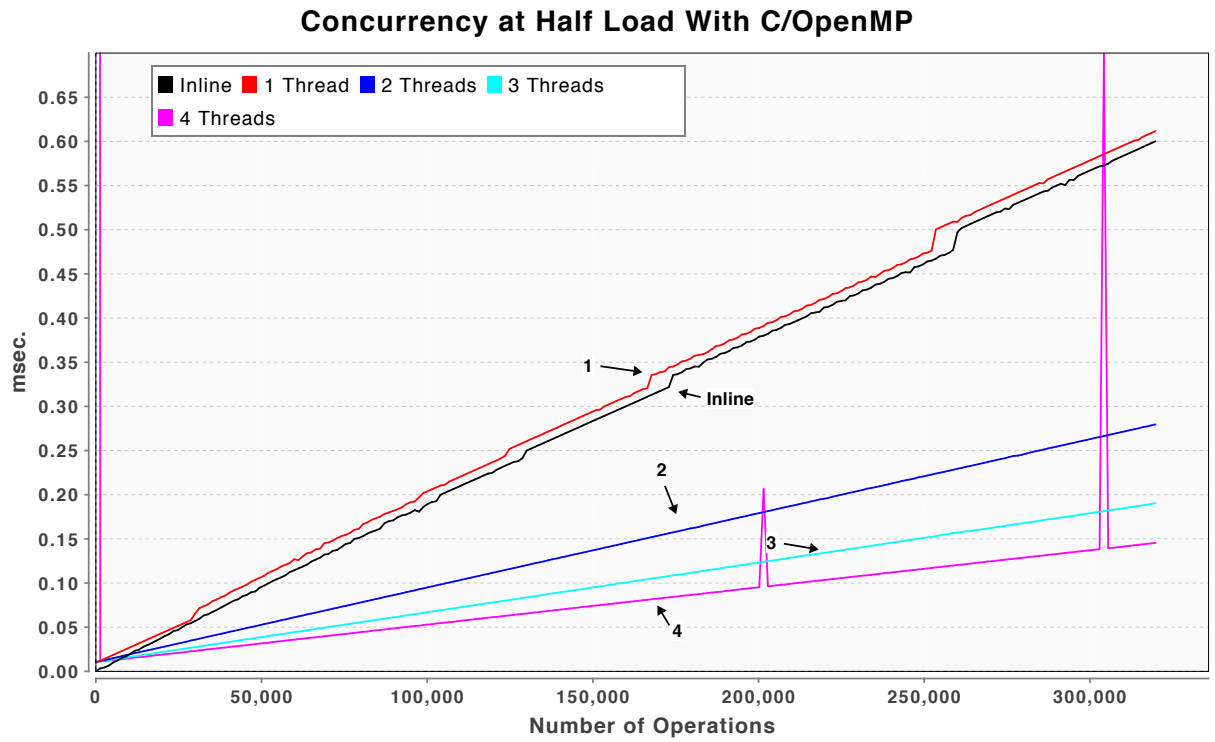

Figure 3-16. Concurrency Using C with OpenMP

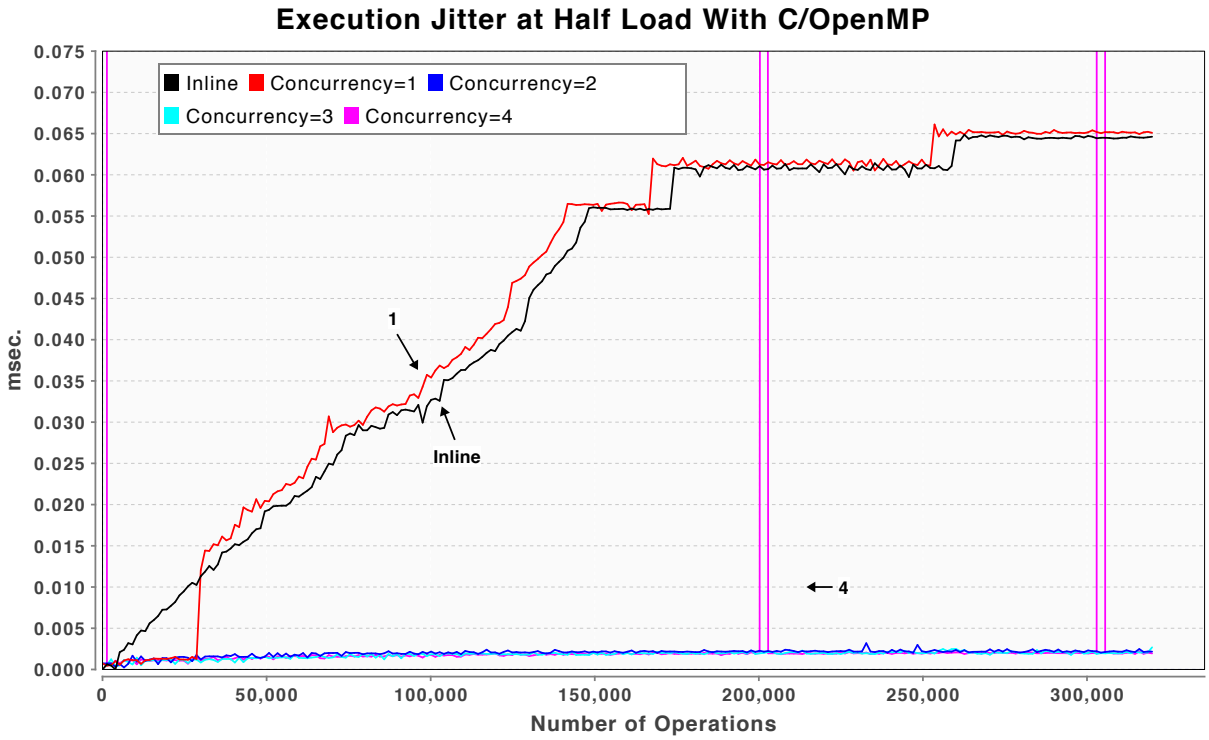

Figure 3-17. Execution Jitter for C and OpenMP

Table 3-1. Normalized Parallelization Time Comparisons at "Half" Loading

Case	2-Way Crossover	3-Way Crossover	4-Way Crossover	2-Way Load Split	3-Way Load Split	4-Way Load Split
N-1-Timer	35,600	28,300	23,600	55%	39%	31%
N-Timer	36,200	30,400	24,700	55%	39%	32%
ExecutorService	34,400	35,000	27,700	54%	40%	34%
ForkJoinPool	68,500	47,900	42,100	60%	44%	37%
ParallelJava	28,600	23,500	21,200	53%	38%	30%
Javolution	10,600	28,100	27,500	51%	39%	32%
C	10,400	7,800	7,200	47%	32%	25%

3.3 Parallelizing at "Full" Loading with Real-Time Scheduling

In the previous section, the benchmark code explicitly yielded for 1 millisecond after approximately every millisecond of continuous computing—resulting in approximately 50 percent loading. In this section the compute tasks are run at 100 percent loading. To help the

thread-thrashing problems described earlier, the Java tests are run using a Linux real-time priority of fifty.

Figure 3-18 shows the performance lines for a two-way workload for all the techniques. Since the differences are hard to see graphically the legend lists the approaches in the order of improved performance. The fork-join pool and executor service resulted in the poorest performance. Parallel Java had the best performance.

Figure 3-19 shows the results for the three-way load split. Again the fork-join pool and executor service have the worst performance and Parallel Java has the best. Figure 3-20 shows the four-way load split.

None of the Java concurrent computation techniques does as well as C in utilizing the available cores.

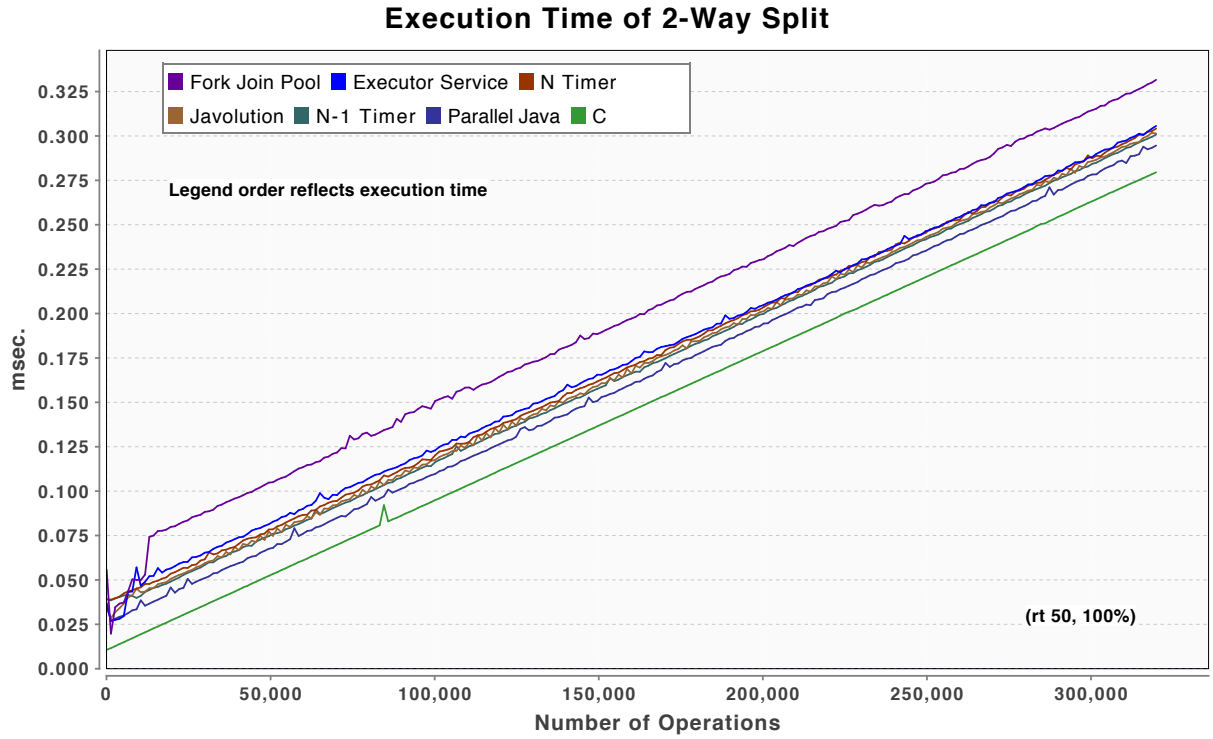

Figure 3-18. Performance Curves for a Two-Way Load Split at Real-time Priority

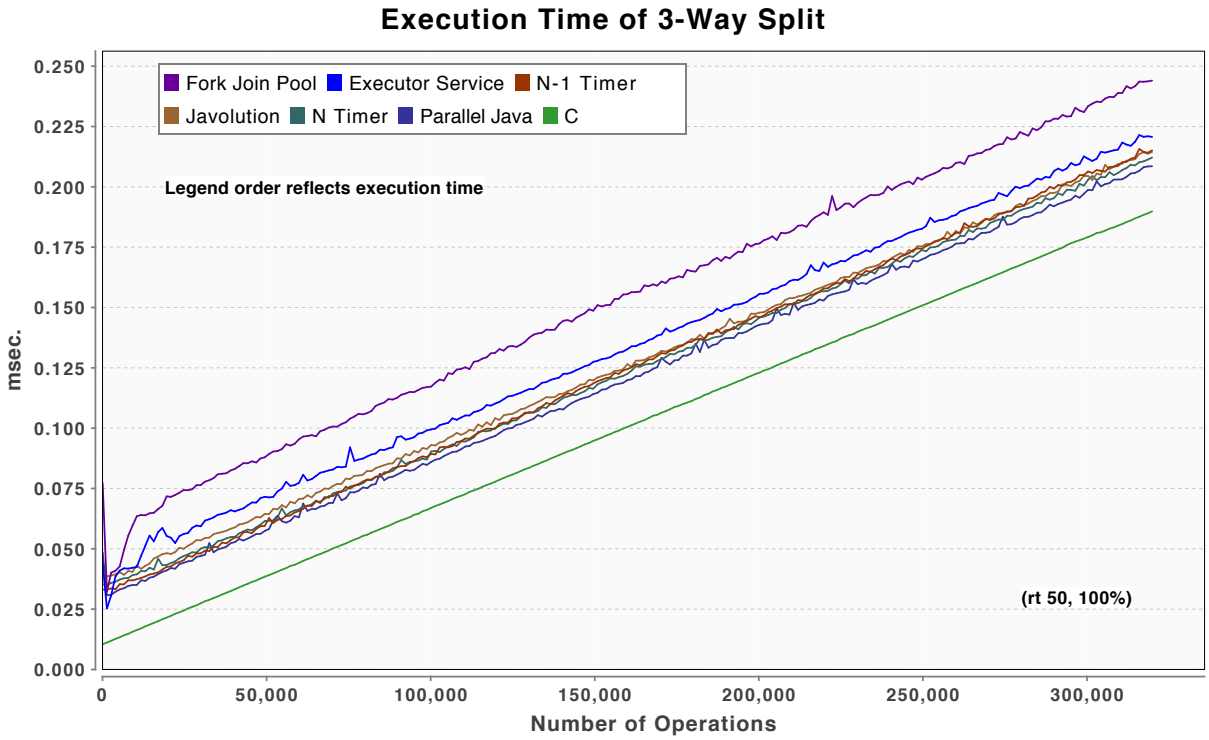

Figure 3-19. Performance Curves for a Three-Way Load Split at Real-time Priority

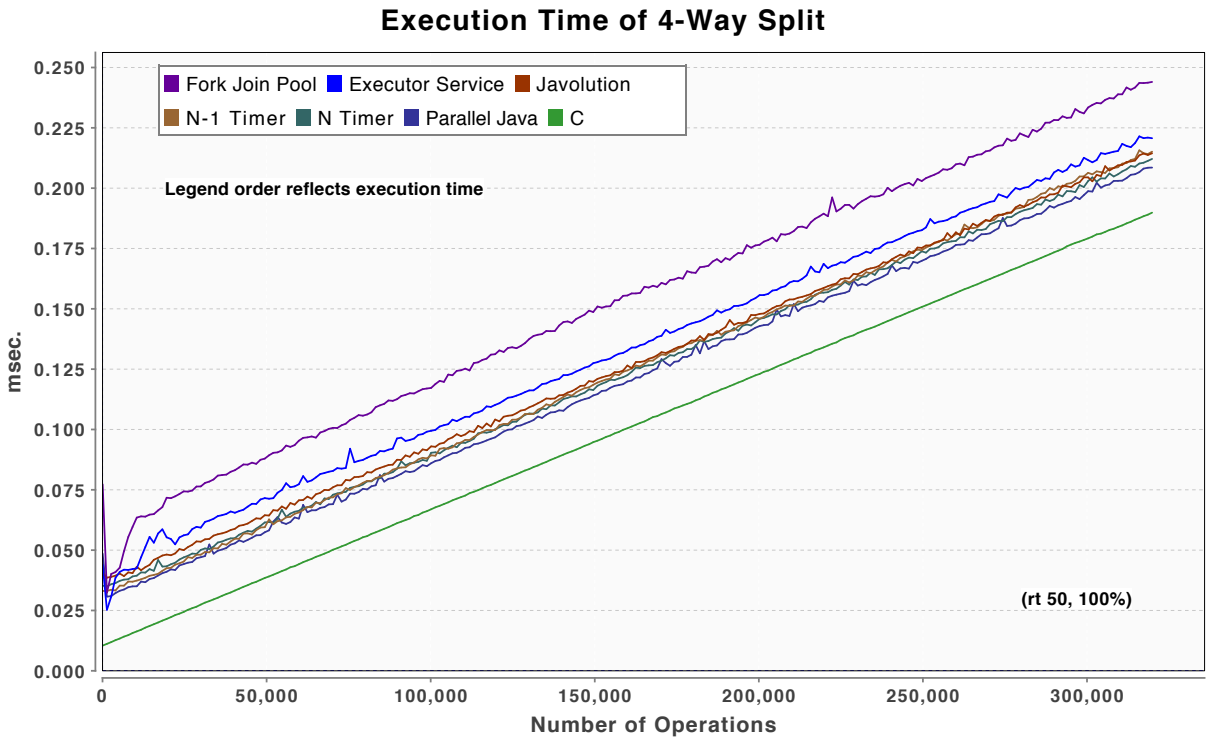

Figure 3-20. Performance Curves for a Four-Way Load Split at Real-time Priority

Table 3-2 numerically summarizes the findings. Figure 3-21 to Figure 3-23 show execution jitter. Comparing the N timer and N-1 timer performance show mixed results. One does better with the two-way parallelism, the other better with the four-way parallelism.

The ExecutorService has better performance and lower jitter than the ForkJoinPool. Parallel Java exhibits the best parallelism consistently and it exhibits good jitter behavior.

Comparing C with the Java results one sees that the best two-way Java parallel load splits fall about 3 percent short of C's. This difference goes up to 4 percent with a four-way load split.

Table 3-2. Normalized Parallelization Time Comparisons with Full Loading

Case	2-Way Crossover	3-Way Crossover	4-Way Crossover	2-Way Load Split	3-Way Load Split	4-Way Load Split
N-1-Timer	39,500	28,600	29,500	56%	40%	35%
N-Timer	43,900	29,400	27,500	57%	40%	31%
ExecutorService	47,700	39,300	30,900	56%	41%	33%
ForkJoinPool	79,300	54,200	46,200	62%	46%	36%
ParallelJava	30,800	26,800	25,200	55%	39%	31%
Javolution	40,000	32,500	30,300	56%	40%	32%
C	13,000	10,400	9,100	52%	35%	27%

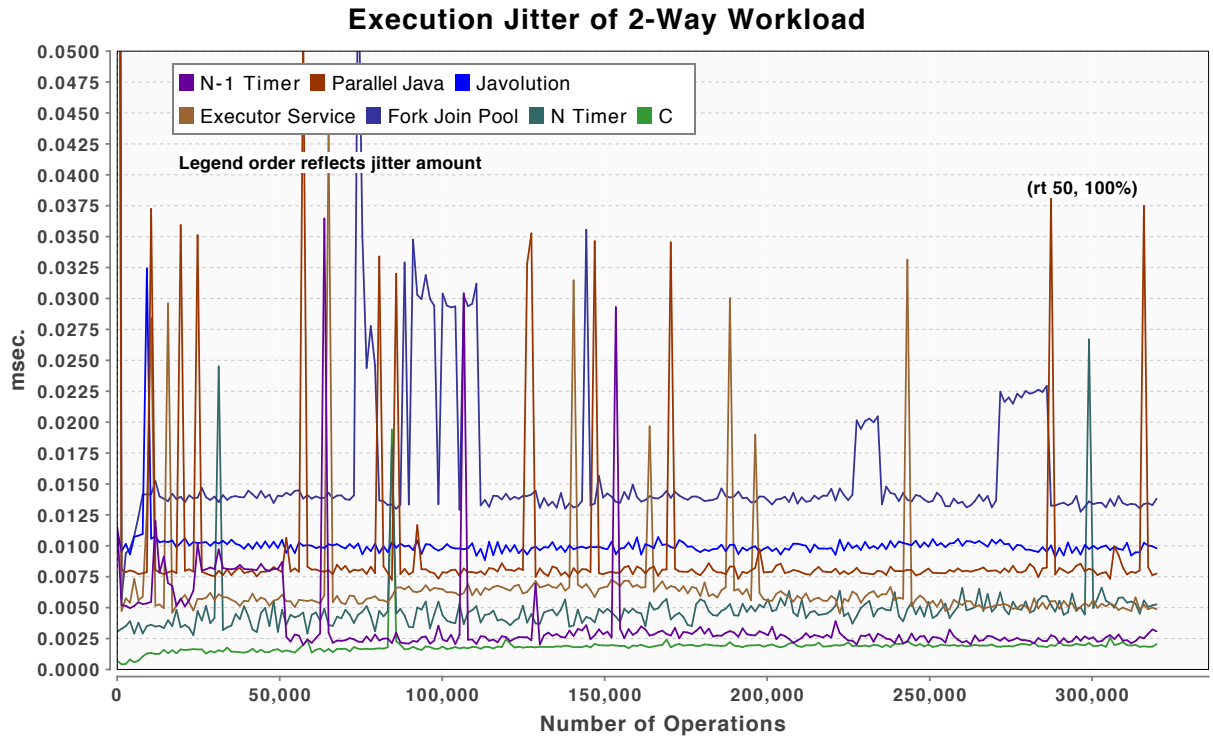

Figure 3-21. 2-Way Execution Jitter at Full Loading

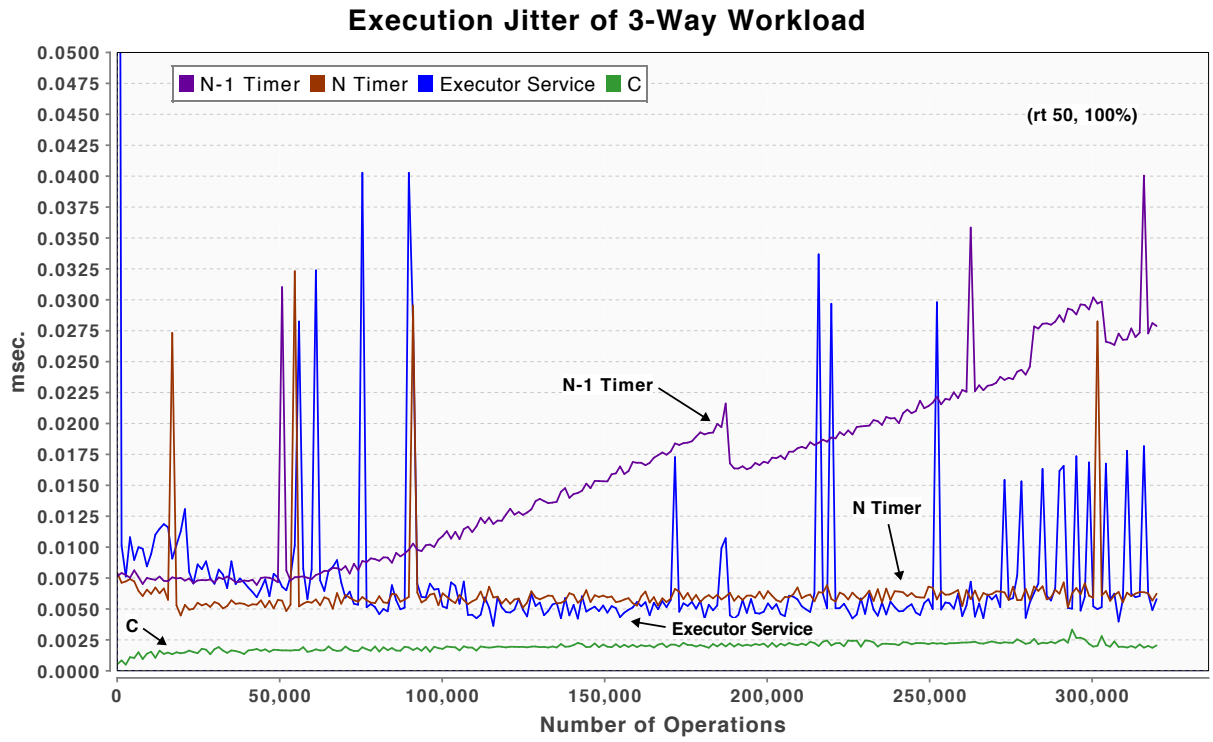

Figure 3-22. 3-Way Execution Jitter at Full Loading (1 of 2)

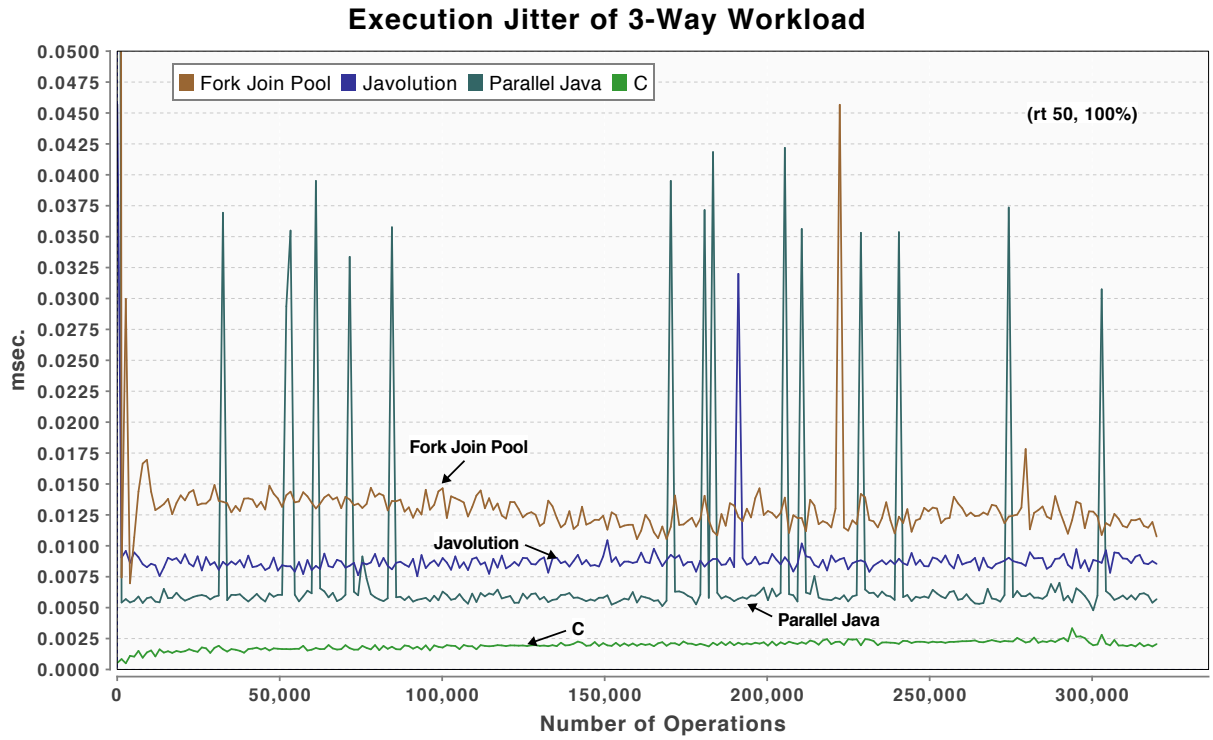

Figure 3-23. 3-Way Execution Jitter at Full Loading (2 of 2)

3.4 A Summary of Java Parallelism for Numerical Computation

Tests were performed on a four-core Linux system to see how effectively Java could parallelize a simple workload. Linux can run applications using default scheduling but also have provisions to run applications using real-time scheduling at real-time priorities.

Various Java utility classes were tested to see how they performed at 50 percent loading and at 100 percent loading. With default Linux scheduling, there were various cases of "thread-thrashing," which led to erratic thread execution times. In one case, the thread-thrashing was severe with the effect that adding a third or fourth core provided no computing gain. It is possible that the problem is related to the way the tests repeatedly launch multiple workloads "simultaneously" that also complete simultaneously. However, the study did not have the resources to investigate if this was the root of the problem. Note, that the C version of the tests did not have this problem.

The way that Java thread-thrashing was completely eliminated was to schedule the application to run using Linux round-robin scheduling at real-time priority.

At 50 percent loading, a C four-way load split was able to achieve the "perfect" execution-time reduction of 25 percent. At 100 percent loading, this increased to 27 percent. The best Java parallelization approach was 4 to 5 percent less effective.

Five mechanisms for Java parallelism were compared. The three standard mechanisms were the `Timer`, `ExecutorService`, and the `ForkJoinPool`. The two third-party mechanisms were the use of the ParallelJava library and the Javolution library. ParallelJava obtained the best throughput consistently. Javolution and a `Timer`-based approach were next in overall performance. The `ForkJoinPool` obtained the poorest throughput consistently.

The absence of language features to simplify parallel for-loops iteration will frustrate some with a traditional computing background. Parallel Java has the most concise way to do this. On the other hand, this investigation focused very narrowly on for-loop-based numerical computation. The reader may decide that one of the other methods is more appropriate for other types of concurrency.

There is a key lesson here. Given that there are a variety of concurrency techniques available to the Java developer, the developer should instrument their application and compare the parallelized and unparallelized versions of their application to ensure they are obtaining the expected behavior.

This page intentionally left blank.

4 Java Invoking Native Code

When mixing Java and C, the designer would most likely use C for "number crunching" and to perform tasks that require low execution jitter. Java is likely to be used to perform other tasks such as logging, external application communications, inter-node coordination, and data persistence. Benchmarks were created to better understand how to most quickly invoke C from Java and to gauge the relative difficulties of the options. Here the focus is primarily on numerical data in arrays. This study does not attempt to investigate the use of structures or strings although most of alternatives investigated here support these.

Java provides a low-level API to make calls from Java to C and vice-versa. This is the Java Native Interface[35] (JNI). Using JNI directly can be cumbersome because it can take multiple JNI service invocations to access array arguments. Integer arguments are passed directly, however.

If one starts with a C API, the method signatures would have to be modified to accept calls directly from Java. It is usually easier to write a proxy that accepts the Java calls, locks any Java arrays (so the garbage collector does not move the data), copies the arrays if needed, and then invokes the actual C function. When the C function returns, modified arrays have to be written back and unlocked.

This is the fastest way to access Java data, though it is awkward. The JNI proxy code is C code that must be compiled and linked for the target machine and operating system. Thus callouts via JNI are not platform independent.

There are higher level options for Java-to-C invocation. Most have a platform dependence, though one achieves platform independence for a significant set of popular operating systems and target hardware.

The higher level options typically hide the JNI details and seek to provide both a simple Java interface and a simple C interface. They typically build a native proxy class statically, as a developer compile-time action, or dynamically by building "glue" on-the-fly. Four such options are Java Native Access (JNA), BridJ, Simplified Wrapper and Interface Generator (SWIG), and HawtJNI.

From the description at the JNA repository site[36]:

> "JNA provides Java programs easy access to native shared libraries (DLLs on Windows) without writing anything but Java code—no JNI or native code is required. […] Access is dynamic at run time without code generation. […] The Java call looks just like it does in native code. […] The JNA library uses a small native library stub to dynamically invoke native code. The developer uses a Java interface to describe functions and structures in the target native library. […] While some attention is paid to performance, correctness and ease of use take priority."

Bridj[37] is a library that:

---

35 "Java Native Interface Specification." Oracle Java SE Documentation. <http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html>

36 "Java Native Access (JNA)." <https://github.com/twall/jna#readme>

37 "BridJ: Let Java & Scala Call C, C++, Objective-C, C#..." <http://code.google.com/p/bridj/>

> "Lets you call C, C++ and Objective-C libraries from Java as if you were writing native code. It is a recent alternative to JNA. With the help of *[the tool]* JNAerator[38] , it provides clean and unambiguous mappings to C/C++ code and a very straightforward and highly typed API to manipulate pointers, classes, structs, unions..."

With JNAerator, a Java .jar file can be generated that has all the necessary glue logic prebuilt and can include the native library inside for great ease of use. With Bridj, Java code can be used to allocate structures in native memory, eliminating the need to convert Java data into C data with the associated overhead. Also, Bridj creates proxy Java objects that will clean up the C-side data when the Java objects are reclaimed. Thus, Bridj allows the developer to leverage the Java garbage collector to keep C data properly garbage collected as well. The code that JNAerator generates is pure-Java, which means that the resulting Java proxy classes will run on any target machine and operating system that Bridj already runs on.

SWIG[39]:

> "Is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. SWIG is used with different types of target languages including common scripting languages such as Perl, PHP, Python, Tcl and Ruby *[and Java]* … SWIG is typically used to parse C/C++ interfaces and generate the 'glue code' required for the above target languages to call into the C/C++ code."

To generate this glue code requires that the developer write a SWIG-specific interface file that identifies a variety of structure mapping options and identifies the API that is to be exposed. SWIG generates proxy Java and C code. The C code must be compiled and linked for the target machine and operating system.

Finally, HawtJNI[40]:

> "HawtJNI is a code generator that produces the JNI code needed to implement Java native methods. It is based on the jnigen code generator that is part of the SWT Tools project which is used to generate all the JNI code which powers the eclipse platform."

HawtJNI lets the developer write a Java source class with methods that match the target C methods and it generates the required JNI code proxy. For those that like Maven[41], it is well integrated into Maven.

4.1 Generating Bridge Code for Java to C Invocation

Each of the bridging tools/libraries has its own way to create the bridge code to C. For the experiments performed here, the targeted C code is shown in Listing 4-1 and Listing 4-2. The code is shown in two listings for clarity.

---

[38] "JNAerator - Native C / C++ / Objective-C Libraries Come to Java !" <http://code.google.com/p/jnaerator/>

[39] "SWIG." <http://www.swig.org/>

[40] "HawtJNI." <http://fusesource.com/forge/sites/hawtjni/>

[41] "Apache Maven Project." <http://maven.apache.org/>

Listing 4-1. C Test Methods with Integer Arguments

```
int sfInt1 (int i1){
   return i1;
}

int sfInt2 (int i1, int i2){
   return i1;
}

int sfInt5 (int i1, int i2, int i3, int i4, int i5){
   return i1;
}

int sfInt10 (int i1, int i2, int i3, int i4, int i5, int i6, int i7, int i8, int i9, int i10){
   return i1;
}
```

Listing 4-2. C Test Methods with Array Arguments

```
// Access one value from the array
double saDouble1 (const double a1[]){
   return a1[0];
}

// Access one value from one array. Ignore the second array
double saDouble2 (const double a1[], const double a2[], int n){
   return a1[0];
}

// Modify the contents of the second array, by copying from the first array
double saDouble2rw (const double a1[], double a2[], int n){
   int i;
   for (i = 0; i<n; i++){
       a2[i] = a1[i];
   }
   return a1[0];
}
```

4.1.1 Java Native Interface

The JNI API and specification[42] are part of the documentation for the Java Standard Edition. To use JNI directly the developer has to either:

- rewrite the targeted method signatures following JNI conventions, or
- write a proxy class in C that follows the JNI conventions and calls the desired C API.

Then this C code is compiled, linked, and distributed as a platform-specific native library.

The JNI code for the tests performed here is shown in Listing 4-3 and Listing 4-4. JNI adds two arguments to all method calls, for the `JNIEnv` and the `jclass` arguments. These allow the C code to make callbacks into Java to access metadata and data from the Java side.

---

42 "Java Native Interface Specification." Oracle Java SE Documentation, <http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html>

Listing 4-3. Manually Created JNI Code for Test Methods with Integer Arguments

```
JNIEXPORT jint JNICALL Java_sfInt1
 (JNIEnv * env, jclass cls, jint i1)
{
   return i1;
}

JNIEXPORT jint JNICALL Java_sfInt5
 (JNIEnv * env, jclass cls, jint i1, jint i2, jint i3, jint i4, jint j5)
{
   return i1;
}

JNIEXPORT jint JNICALL Java_sfInt10
 (JNIEnv * env, jclass cls, jint i1, jint i2, jint i3,
   jint i4, jint j5, jint i6, jint i7, jint i8, jint i9, jint i10)
{
   return i1;
}
```

Listing 4-4. C Manually Created JNI Code for Test Methods with Array Arguments

```
JNIEXPORT jdouble JNICALL Java_saDouble1
  (JNIEnv * env, jclass cls, jdoubleArray _a1)
 {
    jdouble firstElement;
    jdouble *a1 = (*env)->GetDoubleArrayElements(env, _a1, 0);
    firstElement = a1[0];
    (*env)->ReleaseDoubleArrayElements(env, _a1, a1, JNI_ABORT); //no changes, don't copy back
    return firstElement;
 }

 JNIEXPORT jdouble JNICALL Java_saDouble2
  (JNIEnv * env, jclass cls, jdoubleArray _a1, jdoubleArray _a2, jint n)
 {
    jdouble firstElement;
    jdouble *a1 = (*env)->GetDoubleArrayElements(env, _a1, 0);
    jdouble *a2 = (*env)->GetDoubleArrayElements(env, _a2, 0);
    firstElement = a1[0];
    (*env)->ReleaseDoubleArrayElements(env, _a2, a2, JNI_ABORT); //no changes, don't copy back
    (*env)->ReleaseDoubleArrayElements(env, _a1, a1, JNI_ABORT); //no changes, don't copy back
    return firstElement;
 }

 JNIEXPORT jdouble JNICALL Java_saDouble2rw
 (JNIEnv * env, jclass cls, jdoubleArray _a1, jdoubleArray _a2, jint n)
 {
    jdouble *a1 = (*env)->GetDoubleArrayElements(env, _a1, 0);
    jdouble *a2 = (*env)->GetDoubleArrayElements(env, _a2, 0);
    int i;
    for (i = 0; i<n; i++){
        a2[i] = a1[i];
    }
    double firstElement = a1[0];
    (*env)->ReleaseDoubleArrayElements(env, _a2, a2, 0); // 0= copy back and free
    (*env)->ReleaseDoubleArrayElements(env, _a1, a1, JNI_ABORT); //no changes, don't copy back
    return firstElement;
}
```

4.1.2 Java Native Access

Java Native Access is much easier to use than JNI. The developer has to know the method signatures for the called C methods and then write a Java class that matches the API. The matching Java class for the tests here is shown in Listing 4-5. Note that the JNAerator[43] tool can also be used to generate the Java class from the C .h file for JNA. For the JNA tests done here, the class was written by hand.

With this class defined, it is processed at run time by the JNA library to dynamically generate the required native glue code. The developer does not have to build and distribute a platform-specific native library. However, the price paid for this type of platform independence is speed, as will be seen below.

---

[43] See jnaerator, http://code.google.com/p/jnaerator/wiki/Documentation. Using JNAerator with JNA requires using the "`-runtime JNA`" command line option.

Listing 4-5. Java JNA Class

```
import com.sun.jna.Native;

public class WorkersJNA {
   public static native int sfInt1 (int i1);

   public static native int sfInt2 (int i1, int i2);

   public static native int sfInt5 (int i1, int i2, int i3, int i4, int i5);

   public static native int sfInt10 (int i1, int i2, int i3, int i4, int i5,
                                     int i6, int i7, int i8, int i9, int i10);

   public static native double saDouble1 (double[] a1);

   public static native double saDouble2 (double[] a1, double[] a2, int n);

   public static native double saDouble2rw (double[] a1, double[] a2, int n);

   static {
       String name="WorkersJNA";
       try {
           Native.register("WorkersJNA");
           System.out.println("Native library "+name+" loaded ");
       } catch (UnsatisfiedLinkError e) {
           System.err.println(name+" native library failed to load. \n" + e);
           System.exit(1);
       }
   }
}
```

4.1.3 BridJ

Aside from BridJ and JNA, the bridging technologies investigated here require the developer to build a platform-specific native library. BridJ attempts to keep the developer from building a platform-specific native library by having BridJ internals that are platform specific though the users API will be generic. The developed code will be platform independent within the set of platforms already supported by BridJ.

The developer produces a Java class that matches the target C API either manually or using the JNAerator tool[44] to process the C .h file and produce the matching Java class. BridJ allows array (or struct) data to be created on either the Java side or the native side. The native data can be allocated and a Java proxy object instantiated, for example:

```
Pointer<Double> double1 = Pointer.allocateDoubles(n);
```
(Java code)

When the Java array proxy gets garbage collected, the corresponding native data will also be released by BridJ.

With native allocation, there are two ways to invoke the target C API. The first is to simply pass the object itself, for example:

```
saDouble1(double1)
```

[44] Using JNAerator with BridJ requires using the "`-runtime BridJ`" command -line option.

The second way is to pass the array's native address, for example:

```
saDouble1(double1.getPeer())
```

The first requires that the BridJ runtime library call back to Java to get the actual address of the native array. In the second, the call to `getPeer()` retrieves the array address and passes it directly avoiding the extra call back on each invocation.

Listing 4-6 shows the Java class that expects proxy arguments while Listing 4-7 shows the Java class that accepts native array addresses. The developer's C code is the same in both cases. It is BridJ that uses slightly different runtime glue code.

JNAerator can be used to automatically process the original C .h file and produce an initial Java API class, although that class will use proxy array arguments. JNAerator can also conveniently build a Java .jar file that bundles several things into one .jar file:

- the C native library
- the Java proxy class (using proxy arrays)
- all required BridJ support classes.

Listing 4-6. BridJ Proxy Class with Proxy Array Arguments

```
import org.bridj.BridJ;
import org.bridj.CRuntime;
import org.bridj.Pointer;
import org.bridj.ann.Library;
import org.bridj.ann.Runtime;

@Library("Workers")
@Runtime(CRuntime.class)
public class WorkersLibrary {
   static {
       BridJ.register();
   }

   public static native int sfInt1(int i1);
   public static native int sfInt2(int i1, int i2);
   public static native int sfInt5(int i1, int i2, int i3, int i4, int i5);
   public static native int sfInt10(int i1, int i2, int i3, int i4, int i5,
                                    int i6, int i7, int i8, int i9, int i10);

   public static native double saDouble1(Pointer<Double > a1);
   public static native double saDouble2(Pointer<Double > a1, Pointer<Double > a2, int n);
   public static native double saDouble2rw(Pointer<Double > a1, Pointer<Double > a2, int n);
}
```

Listing 4-7. BridJ Proxy Class with Array Addresses

```
import org.bridj.BridJ;
import org.bridj.CRuntime;
import org.bridj.ann.Library;
import org.bridj.ann.Ptr;
import org.bridj.ann.Runtime;

@Library("Workers")
@Runtime(CRuntime.class)
public class WorkersBridJCAOptimized {
    static {
        BridJ.register();
    }

    public static native int sfInt1(int i1);
    public static native int sfInt2(int i1, int i2);
    public static native int sfInt5(int i1, int i2, int i3, int i4, int i5);
    public static native int sfInt10(int i1, int i2, int i3, int i4, int i5,
                                     int i6, int i7, int i8, int i9, int i10);

    public static native double saDouble1(@Ptr long a1);
    public static native double saDouble2(@Ptr long a1, @Ptr long a2, int n);
    public static native double saDouble2rw(@Ptr long a1, @Ptr long a2, int n);
}
```

4.1.4 Simplified Wrapper and Interface Generator

SWIG is designed for starting with C and C++ code and then generating interface and proxy code for potentially many different languages. To do this, the developer writes a separate SWIG interface file that identifies the methods that are to be exposed. Examples are shown in Listing 4-8 and Listing 4-9.

When working with Java, arrays can be allocated either on the Java-side or on the C-side. When on the C-side, SWIG generates creator/destructor methods to create the arrays. Unlike BridJ, the default generated code does not clean up the native array memory when the Java proxy classes are reclaimed by the garbage collector. The SWIG documentation describes ways to simplify this though it is not as easy as with BridJ. The developer also has to include code to load the required runtime as part of the SWIG .i file.

SWIG is very powerful. It allows the developer to customize how it generates C code to handle different data types on input and output. If arrays are on the Java side, there is no out-of-the-box way to indicate that arrays are input-only or output-only.

Listing 4-8. Example SWIG Interface File Using Java-side Arrays

```
%module WorkersJaSWIG
%include "arrays_java.i"
%{
#include "Workers.h"
%}

%pragma(java) jniclasscode=%{
 static {
   String name="WorkersJaSWIGJNI";
   try {
     System.loadLibrary(name);
     System.out.println("Native library "+name+" loaded ");
   } catch (UnsatisfiedLinkError e) {
     System.err.println(name+" native library failed to load. \n" + e);
     System.exit(1);
   }
 }
%}

int sfInt1 (int i1);
int sfInt2 (int i1, int i2);
int sfInt5 (int i1, int i2, int i3, int i4, int i5);
int sfInt10 (int i1, int i2, int i3, int i4, int i5, int i6, int i7, int i8, int i9, int i10);
double saDouble1 (const double a1[]);
double saDouble2 (const double a1[], const double a2[], int n);
double saDouble2rw (const double a1[], double a2[], int n);
```

Listing 4-9. Example SWIG Interface File Using C-side Arrays

```
%module WorkersCaSWIG
%include "carrays.i"
%array_class(double, DoubleArray);

%{
#include "Workers.h"
%}

%pragma(java) jniclasscode=%{
 static {
   String name="WorkersCaSWIGJNI";
   try {
     System.loadLibrary(name);
     System.out.println("Native library "+name+" loaded ");
   } catch (UnsatisfiedLinkError e) {
     System.err.println(name+" native library failed to load. \n" + e);
     System.exit(1);
   }
 }
%}

int sfInt1 (int i1);
int sfInt2 (int i1, int i2);
int sfInt5 (int i1, int i2, int i3, int i4, int i5);
int sfInt10 (int i1, int i2, int i3, int i4, int i5, int i6, int i7, int i8, int i9, int i10);
double saDouble1 (const double a1[]);
double saDouble2 (const double a1[], const double a2[], int n);
```

```
double saDouble2rw (const double a1[], double a2[], int n);
```

4.1.5 HawtJNI

HawtJNI provides a Maven-friendly way to create a platform-dependent library and C proxy class. One of the strengths of HawtJNI is the ability to use Java annotations to provide hints to the HawtJNI code generator for mapping structures, tagging pointer types, identify input-only, and output-only arguments, etc. Listing 4-10 shows some of these annotations in use.

Listing 4-10. Example HawtJNI Annotated Java API

```
import static org.fusesource.hawtjni.runtime.ArgFlag.NO_OUT;
import org.fusesource.hawtjni.runtime.JniArg;
import org.fusesource.hawtjni.runtime.JniClass;
import org.fusesource.hawtjni.runtime.Library;

@JniClass
public class WorkersHawtJNI {
    public static native int sfInt1 (int i1);

    public static native int sfInt2 (int i1, int i2);

    public static native int sfInt5 (int i1, int i2, int i3, int i4, int i5);

    public static native int sfInt10 (int i1, int i2, int i3, int i4, int i5,
                                      int i6, int i7, int i8, int i9, int i10);

    public static native double saDouble1 (@JniArg(flags={NO_OUT}) double[] a1);

    public static native double saDouble2 (@JniArg(flags={NO_OUT}) double[] a1,
                                           @JniArg(flags={NO_OUT}) double[] a2, int n);

    public static native double saDouble2rw (@JniArg(flags={NO_OUT}) double[] a1,
                                                                     double[] a2, int n);

    static {
        String name="WorkersHawt";
        Library LIBRARY = new Library(name, WorkersHawtJNI.class);
        try {
            LIBRARY.load();
            System.out.println("Native library "+name+" loaded ");
        } catch (UnsatisfiedLinkError e) {
            System.err.println(name+" native library failed to load. \n" + e);
            System.exit(1);
        }
    }
}
```

4.2 Test Results

C methods having one, two, five, and ten integer arguments and one and two arrays of double values were used to test the speed:

- for calling methods with integer arguments

- for calling methods with array data
- with which Java-side data is converted to and from the C methods as needed.

The time to invoke methods with integer arguments is primarily the time to set up arguments on the stack. The time to invoke the methods with array arguments includes the time to access and convert Java data to C data if the data was allocated on the Java side. Most of the methods only return a single double value but the method `saDouble2rw` also copies the contents of one array to the other. If the data originated on the Java side, then that data must be copied and converted back to Java.

There are seven test cases examined here, exercising the various Java-to-C calling technologies. These are listed in Table 4-1. There are BridJ and SWIG tests to test native allocations and Java-side allocations. For the others, the data is on the Java side. Since it takes time to move, pin-down, and/or convert data from Java to C and vice-versa, one expects the two cases with native data to be the fastest.

In the table there is a column that indicates which of the test cases provide developer control over whether data is input-only. Intrinsically, the native-data cases provide control because the developer's C code is in full control of which arrays are read and written. For Java data, the BridJ data proxy gives the user control over when to put or when to get data from the native side. HawtJNI has Java annotations to indicate non-output and non-input.

Table 4-1. Array-Focused Comparison of Java Native Callout Options

Test Case	Array Data is on the Java Side	Array Data is Native	Approach Supports Input-Only Array Arguments
JNI	√		√
JNA	√		
BridJ-Ja (Java arrays)	√		√
BridJ-Ca (Native arrays)		√	√
SWIG-Ja (Java arrays)	√		Note 1
SWIG-Ca (Native arrays)		√	√
HawtJNI	√		√

Note 1: SWIG is very flexible and could allow hand-implemented input-only or output only arrays with sufficient work. It does not support this "out-of-the-box."

Figure 4-1 graphs the time to call methods with simple integer arguments. The time increases as the number of arguments increase, as expected. The manual JNI implementation is the fastest, while the JNA implementation is the slowest. The next fastest way to invoke methods with simple, non-array arguments is SWIG. BridJ is very fast with a few arguments (four or fewer) but

slows down beyond that. BridJ is still under development and the developer is planning to extend the performance "knee" to sixteen arguments. The BridJ build used here was dated July 18, 2012.

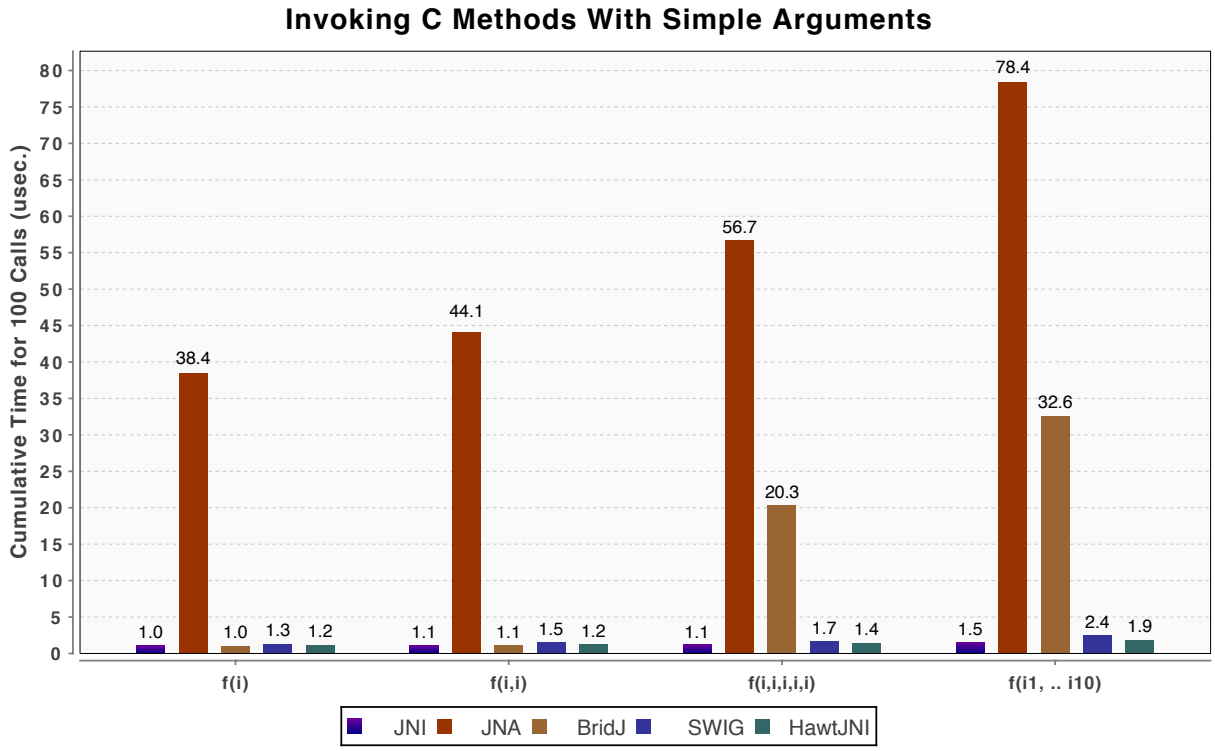

Figure 4-1. Invoking Native Methods with Simple Integer Arguments

The next set of timings, in Figure 4-2, are for calling methods with input-only double array arguments of length 200. These are the methods `saDouble1()` and `saDouble2()` from Listing 4-2. As expected, the two fastest cases are the ones where the data is allocated natively.

When the array data is on the Java side, BridJ has the fastest invocation times. SWIG is slower because it assumes the arrays are input and output. Thus, it converts the data back into Java even when not needed.

With array data on the native side, BridJ has a slight edge on SWIG in this test. The BridJ proxy was specified using the `getPeer()` approach to pass the array's native address, which is a mechanism similar to the one that SWIG uses.

A separate comparison, in Figure 4-3, compares BridJ's two ways of passing arrays. There is a big speed difference between the two approaches that may or may not be significant depending on how much time the body of the method takes.

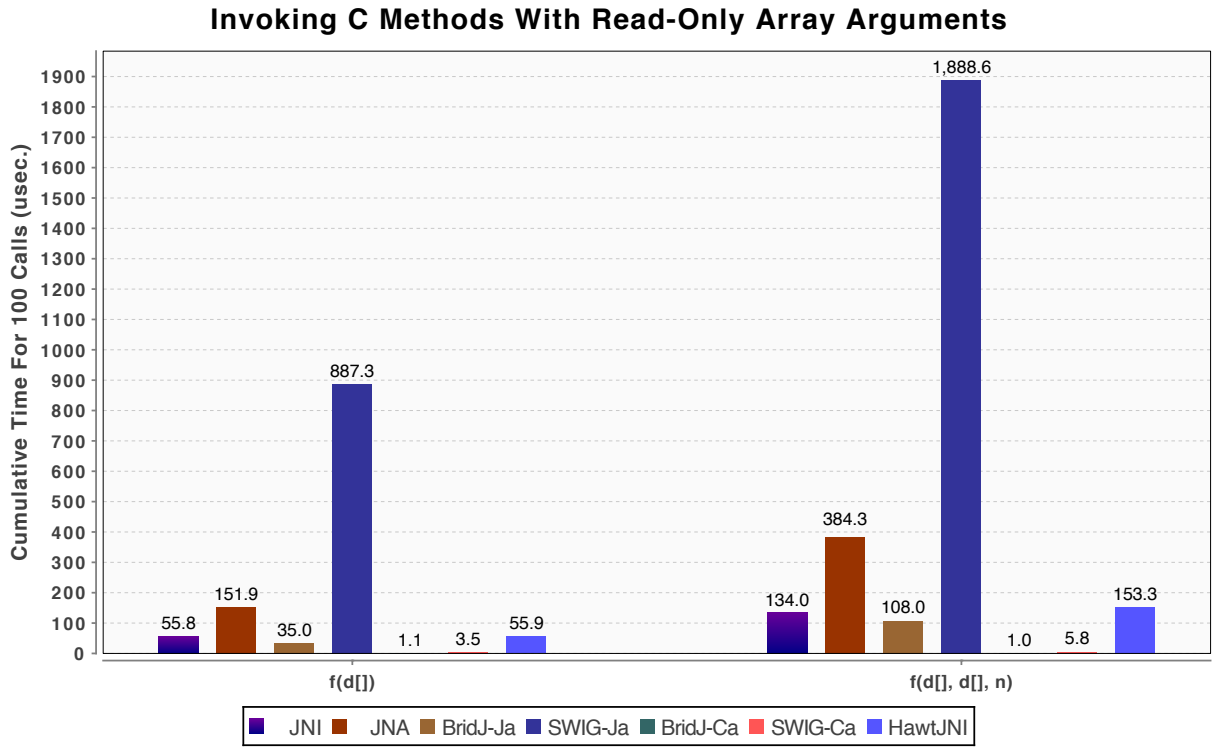

Figure 4-2. Invoking Native Methods with Array Arguments

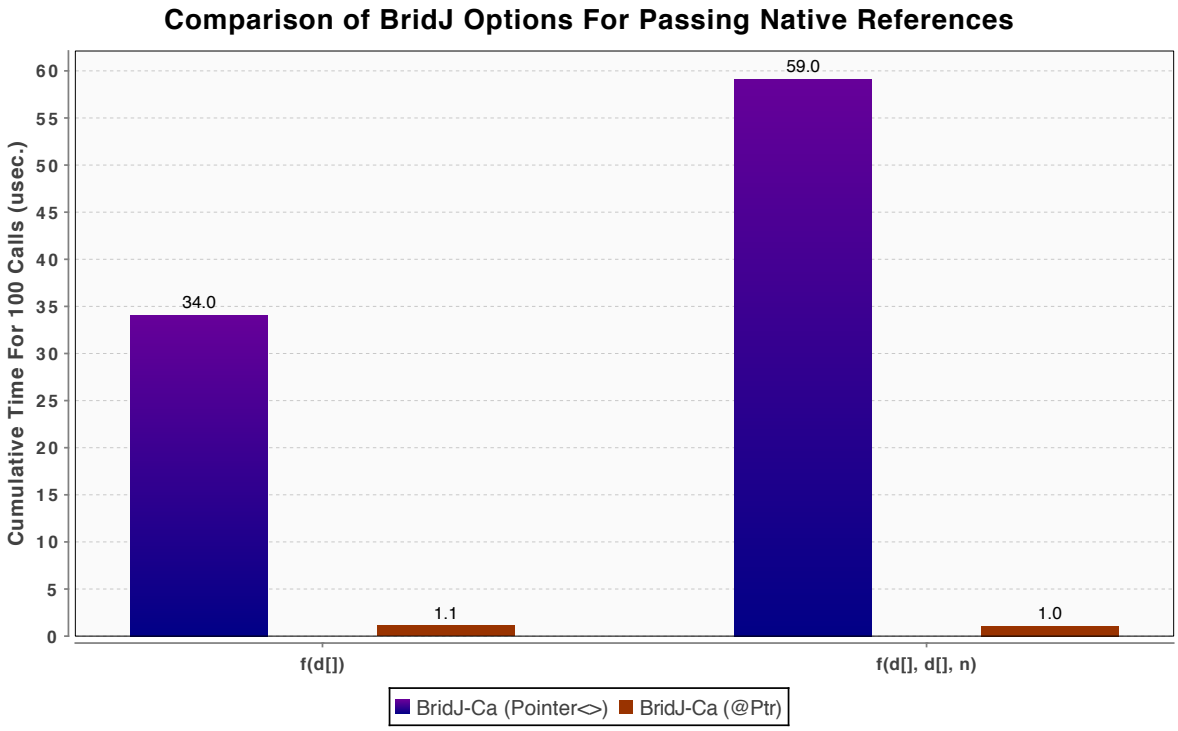

Figure 4-3. BridJ's @Ptr Array Passing Is Fast

The final set of times are shown in Figure 4-4. These times are for the method that copies one array to the other and returns the modified array. The times shown include both the additional work of copying one array to another and the time to copy the data back to Java. Some of the technologies such as JNA do this automatically. For others, the copy-back was done by extra code.

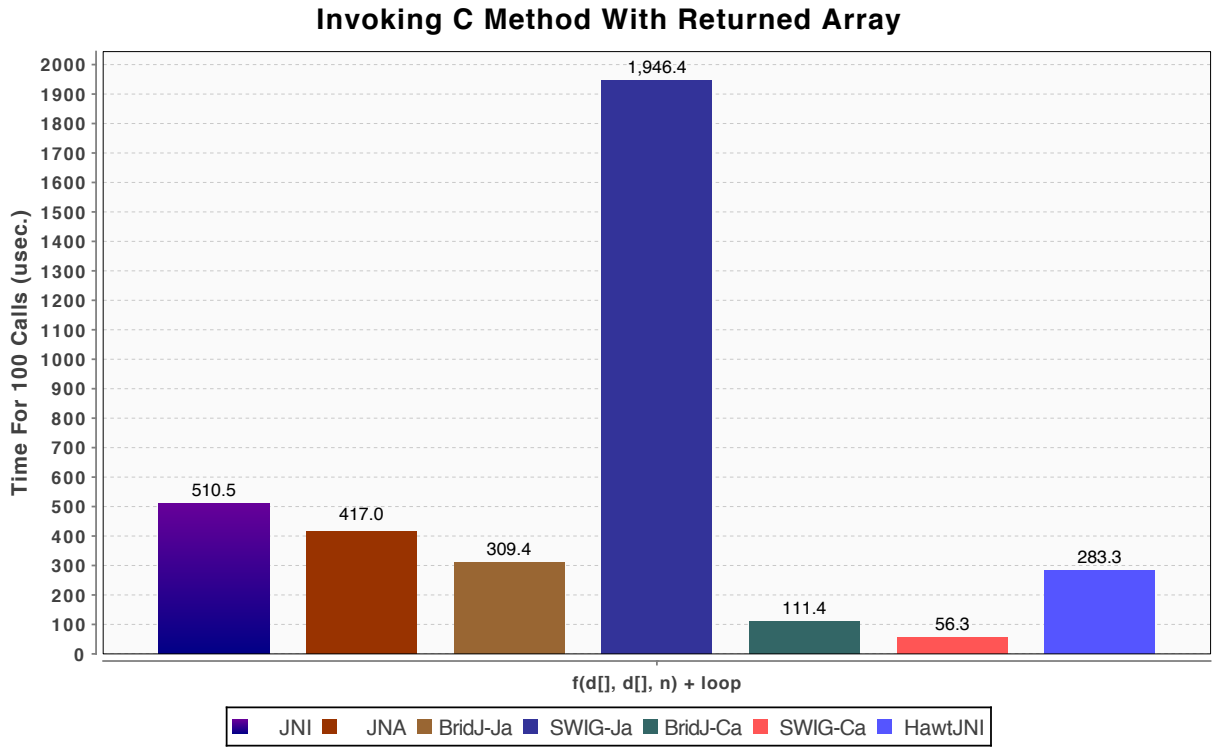

Figure 4-4. Invoking Native Methods That Modify an Array

4.3 Java Calling Java, C Calling C

The previous section presented timings for calling from Java to C. The question may arise as to how quickly Java can call Java and C can call C. A simple benchmark was designed to measure this. The problem with the benchmark was that a Hotspot-based Java inlines simple method bodies. This makes it difficult to design a simple test to compare the two call times. Figure 4-5 shows the results. The figure shows that the Java execution time (due to presumed in-lining) is approximately one-sixth that of C's. This underscores the value of one of Hotspot's optimizations.

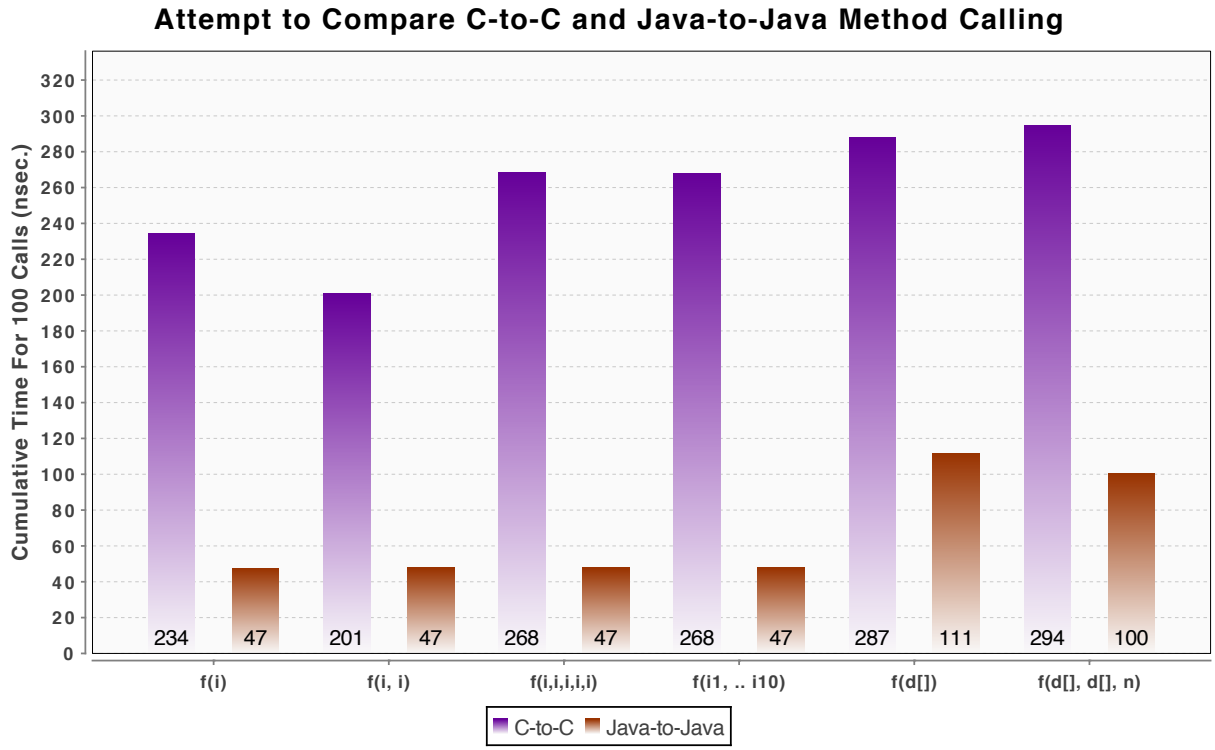

Figure 4-5. C-to-C and Java-to-Java Call Times

4.4 Calling from C to Java

This section has assumed that Java would be used as the manager and controller of compute methods that were implemented in C. It may be necessary to have the main program written in C and invoke Java as a "side-kick" that handles selected tasks. The developer can start a JVM from C using existing JNI methods. They can also invoke Java code from C using JNI.

There are a few resources for how C code can be written to do low-level JNI including Oracle's Invocation API documentation[45] and IBM® developerWorks® tutorial[46]. There were no tools, however, to simplify this (such as the way that BridJ simplifies calling C from Java). A tool was written to help start the JVM and to instantiate static public Java methods with simple arguments. The CShimToJava[47] tool has been released to the public domain for the benefit of others.

4.5 Summary of Tools and Libraries for Calling from Java to C

Five tools with their corresponding libraries were tested for making calls from Java to C; JNI, JNA, BridJ, SWIG, and HawtJNI.

---

[45] Oracle, The Invocation API, < http://docs.oracle.com/javase/6/docs/technotes/guides/jni/spec/invocation.html >

[46] IBM developerWorks, Java programming with JNI, < http://www.ibm.com/developerworks/java/tutorials/j-jni/section3.html >

[47] C Shim To Java, < http://cshimtojava.sourceforge.net >

The fastest way to call C from Java is to manually code a JNI proxy class that accepts the Java call and mediates the call to the C method. JNA is well known to be slow and the findings here confirm that. HawtJNI performs better than JNA but not as well as BridJ and SWIG.

BridJ and SWIG are the fastest overall. BridJ is faster and is platform independent within the platforms supported by BridJ. SWIG is slower, and more difficult to learn and work with, but yields interfaces that can be compiled for any platform that has C. SWIG, however, has great potential for customization, which means that with sufficient work it could ultimately be faster than BridJ.

If an application is being developed on a platform supported by BridJ, the best strategy may be to generate the BridJ-based glue using JNAerator. If call times are too slow and if the data (either arrays or structs) is native, then the developer can modify the Java proxy to utilize BridJ's @Ptr mechanism.

If the platform is not supported by BridJ then there are two options. If the data is primarily array data, JNI can be used "by rote" which yields the ultimate in speed. If there are C structures involved, then SWIG facilitates working with those.

Table 4-2 summaries the pros and cons of these technologies along with this investigator's assessment of ease of use for working with array-based data.

Table 4-2. Comparison of Bridging Tools

	Pros	Cons	Investigators Ranking of Ease of Use (1 = Easiest)
Manual JNI	• Gives complete control over what data is read-only and what is read/write	• C method arguments must be modified to implement JNI conventions or a proxy class must be manually written. • JNI code must be recompiled and linked for every different target platform and O/S.	4
JNA	• Easiest to use • Platform independent deployment	• Slowest	1
Bridj	• Allows data to be allocated natively. • Utilizes existing .h files to generate a Java proxy class to the native API. • Cleans up native data when the Java proxy is reclaimed. • Fastest out-of-the-box behavior on a variety of tests.	• The automated way to generate the proxy API class with does not generate the fastest way to pass array data • Supports a fixed set of operating systems and hardware platforms	2
SWIG	• Allows data to be allocated natively. • Can generate interfaces for many languages in addition to Java • Very fast	• Does not include C-side memory cleanup out-of-the-box. • Requires a separate SWIG-unique interface definition file • Generated C code must be compiled and linked for every different target platform and O/S.	5

This page intentionally left blank.

5 Java, Graphics Processing Units, and OpenCL™

GPUs offer significant speed benefits for many types of computations. OpenCL[48] provides a versatile way to program a variety of vendor GPUs on a variety of hardware and operating system platforms. The specification for versions 1.0, 1.1, and 1.2 of the API can be found at the Khronos Group's web site.[49] The power and availability of GPUs suggest that the GPU is not an option but is instead a critical part of the high performance computing toolbox.

OpenCL is not the only way to use GPUs but it is a key way. The reader may be interested in investigating projects such as Aparapi[50] and Rootbeer[51] that simplify or eliminate explicit GPU kernel creation and invocation. These were not tested or evaluated in this study. This study is based on a more traditional use of OpenCL.

There are several Java libraries that provide access to OpenCL. Some just implement the C OpenCL API leaving the Java developer to worry about memory pointer details. Others provide greater abstraction and greater type safety. Table 5-1 and Table 5-2 identify four Java alternatives. The tables provides a brief summary of key aspects of each alternative.

Several of the alternatives use `java.nio` (NIO) direct data buffers to hold array data for OpenCL kernels natively. NIO does not provide a direct way to release native memory once it has been allocated. NIO buffers were designed so that the garbage collector could eventually reclaim the memory when there were no more references to them. However, since the garbage collector tends to wait until the Java heap is mostly full and since native buffer memory is not on the heap, it may take a while for the garbage collector to reclaim unreferenced NIO buffers.

There appear to be ways to work around the NIO buffer deallocation problem[52] by making direct calls into non-public Java classes. These may not work on all Java implementations. If NIO buffer storage is the only option for array data then the best approach is to allocate all the required buffers once and then reuse them throughout the life of the application. This is a good idea anyway to reduce memory fragmentation and to minimize garbage collector work.

---

[48] OpenCL is a trademark of Apple, Inc. and is used under license by Khronos Group.

[49] "OpenCL - The open standard for parallel programming of heterogeneous systems." <http://www.khronos.org/opencl/>

[50] Aparapi converts Java bytecode to OpenCL at run time and executes it on the GPU. See <http://code.google.com/p/aparapi/>.

[51] Rootbeer is a graphics processor unit (GPU) compiler that analyzes standard Java bytecode and produces Compute Unified Device Architecture (CUDA)-based GPU code. See <https://github.com/pcpratts/rootbeer1/>.

[52] For ways to work around the inability to free NIO direct buffers see <http://stackoverflow.com/questions/8462200/examples-of-forcing-freeing-of-native-memory-direct-bytebuffer-has-allocated-us>.

Table 5-1. Java-Based OpenCL Libraries, JOCL and Java OpenCL

Library	Summary
JOCL (jocl.org)	• http://www.jocl.org/downloads/downloads.html • OpenCL 1.1 • Implements the C OpenCL interface almost verbatim via a `org.jocl.CL` object that is directly backed by JNI. • The developer must check error return codes as with native OpenCL. • The developer stores array data using `java.nio` direct/native buffers. An `org.jocl.Pointer` object then wraps the `nio` buffer data and is used in the API to OpenCL (for example to pass to the `clSetKernelArg()` method)3. • Platform binaries available for Win 64, Win 32, Mac x86_64, Linux x86, Linux x86_64.
Java OpenCL (jogamp.org)	• http://jogamp.org/jocl/www/ • OpenCL 1.1 • Implements an object-oriented abstraction of OpenCL that may simplify usage for developers not already familiar with OpenCL. Kernels are invoked via the `com.jogamp.opencl.  CLCommandQueue` class using methods that are named a little differently than in the OpenCL spec. • Checks OpenCL return error codes. Invokes a `com.jogamp.opencl CLErrorHandler` that is coded and registered at run time by the developer. • Has a parameterized `com.jogamp.opencl.CLBuffer` object that represents an OpenCL `cl_mem` buffer. A `CLBuffer` can be based on any of the `java.nio` buffer types, `ByteBuffer`, `DoubleBuffer`, …, `etc`. The `CLBuffer` can move data from indirect NIO buffers to the native side. It can also use direct NIO buffers. The `CLBuffer` can be released but does not result in immediately freeing the associated native memory other than as is normally done by NIO. • Platform-dependent versions for android, Linux x64, Linux ARMv7, Linux i586, MacOS X, Solaris x64, Solaris 586, Windows x64, Windows i586

Table 5-2. Java-Based OpenCL Libraries, JavaCL and LWJGL

Library	Summary
JavaCL and OpenCL4Java (nativeLibs4Java)	• OpenCL 1.2 • JavaCL is an abstraction tier built over OpenCL4Java which contains the low-level bindings that closely match the OpenCL C API. • Kernel execution is handled via the `com.nativelibs4java.opencl.CLKernel` class. This class has multiple `setArgs()` overloads to facilitate passing scalar values and isolate the developer from memory pointer ambiguities. • Automatically checks error return codes and converts them to Java exceptions • Adds thread safety if running on OpenCL 1.0 • No platform-specific OpenCL native libraries though BridJ has platform-specific native libraries for MacOS X (x86, x64, ppc), Linux (x64, x86), Solaris x86, Win (32, 64), Android
The Lightweight Java Game Library (LWJGL)	• http://www.lwjgl.org/ • OpenCL 1.2 • OpenCL support is part of a larger library targeted to game developers on small (J2ME) devices. OpenCL support is found in the org.lwjgl.opencl package. • Setting kernel arguments is done via the org.lwjgl.opencl.CLKernel class. This class has multiple setArgs() overloads to facilitate passing scalar values and isolate the developer from memory pointer ambiguities. Queuing a kernel is done via one of the classes; CL10, CL11, CL12. These implement the constants and methods of the corresponding OpenCL specifications 1.0, 1.1, and 1.2. • The developer must check error return codes as with native OpenCL. • Platform dependent JNI native libraries for Linux(x86, x64), MacOS X(x86, x64), Solaris (x86, x64), Windows (32,64)

JavaCL, from the nativeLibs4Java project, was chosen for testing. JavaCL has a private mechanism for allocating and de-allocating native storage as needed based on the capabilities of BridJ. BridJ provides a general ability to interface with C structures, not just arrays. This capability could make it ideal for use in a mixed Java-C environment that includes GPU's as well as other types of native access. JavaCL also checks OpenCL return error status and converts them to Java exceptions when errors occur. This allows the Java developer to use try-catch blocks, which may be more appropriate for multi-scoped error catching.

There was a concern about the performance impact of additions such as the conversion of errors to exceptions. A test was designed to compare the rates at which Java (using JavaCL) and C can invoke kernels for FFTs using a variety of data sizes. Optimized FFT kernels developed by

Apple, Inc.[53] were used as the basis of the C test. The test code was translated as directly as possible into Java for the Java test. (The converted code is available as part of the benchmark code released to the public domain[4]).

FFT processing can be decomposed as a chain of processing stages. Each stage is further decomposed as a set of transformations that can be performed in parallel on subsets of the data. The number of steps in the chain and the architecture of each subset of the processing can be optimized based on the data size, whether the data is processed in-place or out-of-place, whether the real and imaginary parts are in separate arrays or interleaved, etc. Apple's C code generates and sequences the OpenCL kernel code. The C-based and Java-based tests generate the same OpenCL kernels and sequence them in the same way. There should be no difference in the execution time of the FFT kernel but the rate at which the Java code can set up kernel arguments, compute kernel work group sizes, and, queue kernel executions needed testing.

The key performance metrics in the test are the per-FFT setup time and the net FFT throughput. These were measured for data sizes ranging from 64 to 524,288 complex points. For each data size the throughput was measured for two cases. In the synchronous case, the code waits at the end of each FFT chain (by invoking `clFinish()`). In the asynchronous case, there is no wait and all the thousands of FFT iterations are queued as quickly as possible.

Figure 5-1 compares the effective FFT throughput in giga-flops. JavaCL is used for the Java test. The C version is only a little faster than the Java version in a few cases. At N=11 the C async case is 9 percent faster. At N=16 the C sync case is 2 percent faster. This plot shows the combined effect of the CPU code, the GPU throughput, and the FFT optimization strategy used for the different lengths. It does not provide insight on what the differences are due to.

At the larger data sizes, the total processing time is limited by the time that the GPU computes. At smaller data sizes the total processing time is throttled by the time to queue many OpenCL operations for small data lengths. This is shown in Figure 5-2 for the synchronous case. For the small data sizes, Java using JavaCL, spends about 5 percent more time on setup relative to the total processing time.

[53] "OpenCL_FFT." Mac Developer Library, <http://developer.apple.com/library/mac/#samplecode/OpenCL_FFT/Introduction/Intro.html>

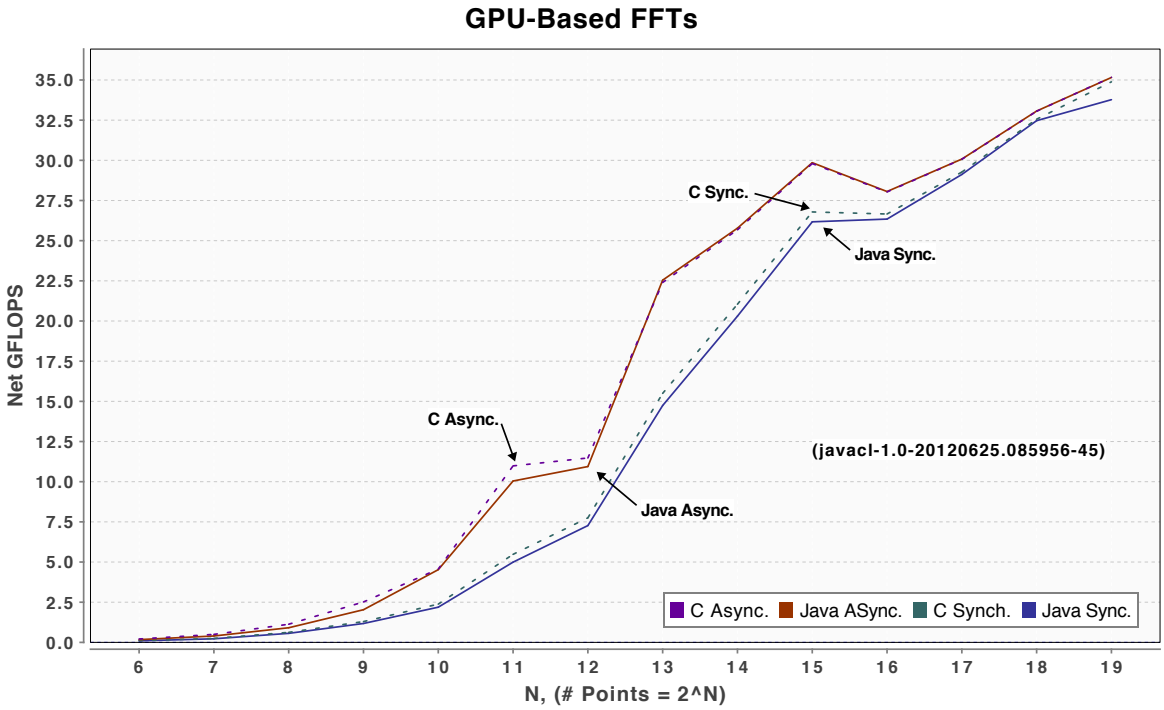

Figure 5-1. Throughput of GPU-Based FFTs

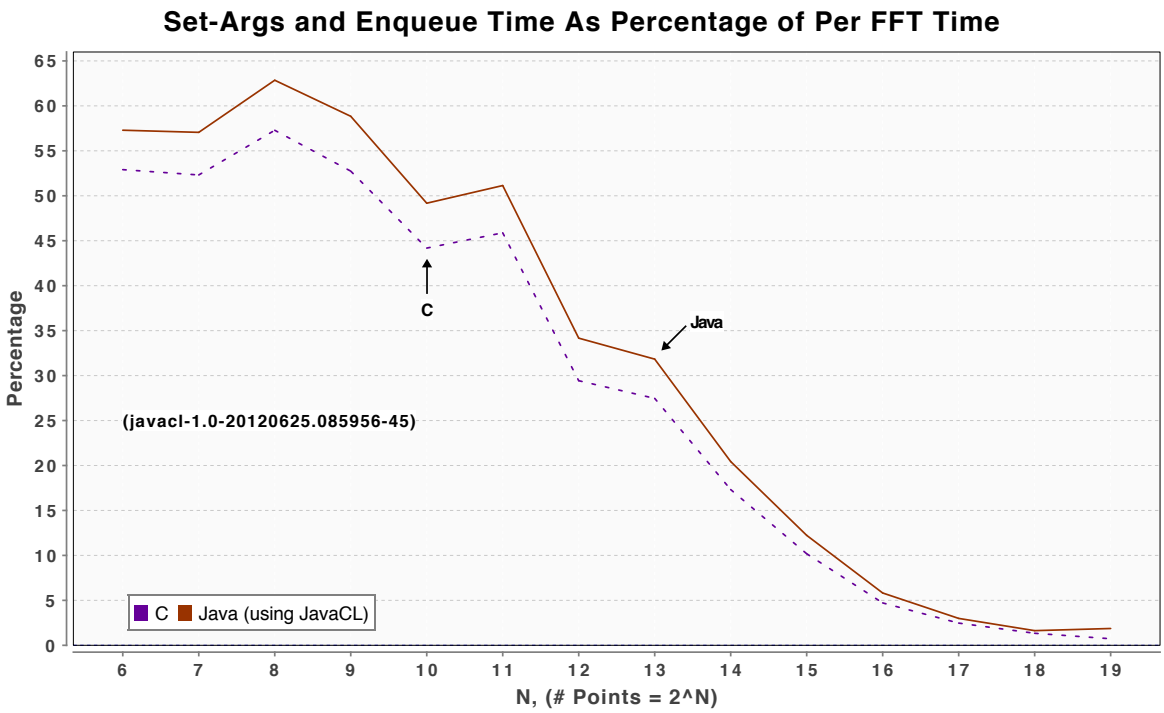

Figure 5-2. Setup Time as a Percentage of FFT Time

Looking at the absolute synchronous setup and enqueue time for the FFT kernels produces the plot of Figure 5-3. This plot shows the time to set kernel arguments, compute workgroup sizes,

and queue the kernel chain for each iteration of the test. It shows that it takes about 2 µs more for Java than for C to set up and enqueue each kernel. Since the Java OpenCL code must invoke the native/C underpinning, these 2 µs are the overhead of the JavaCL library tier and its callout overhead.

This is the final comparison. For this Linux kernel, on this hardware, and with this GPU one would be limited to approximately 105,000 kernel enqueues per second from C. With Java, one would be limited to approximately 87,000 kernel enqueues per second. Only hundreds of kernel enqueues per second are required for many types of problems.

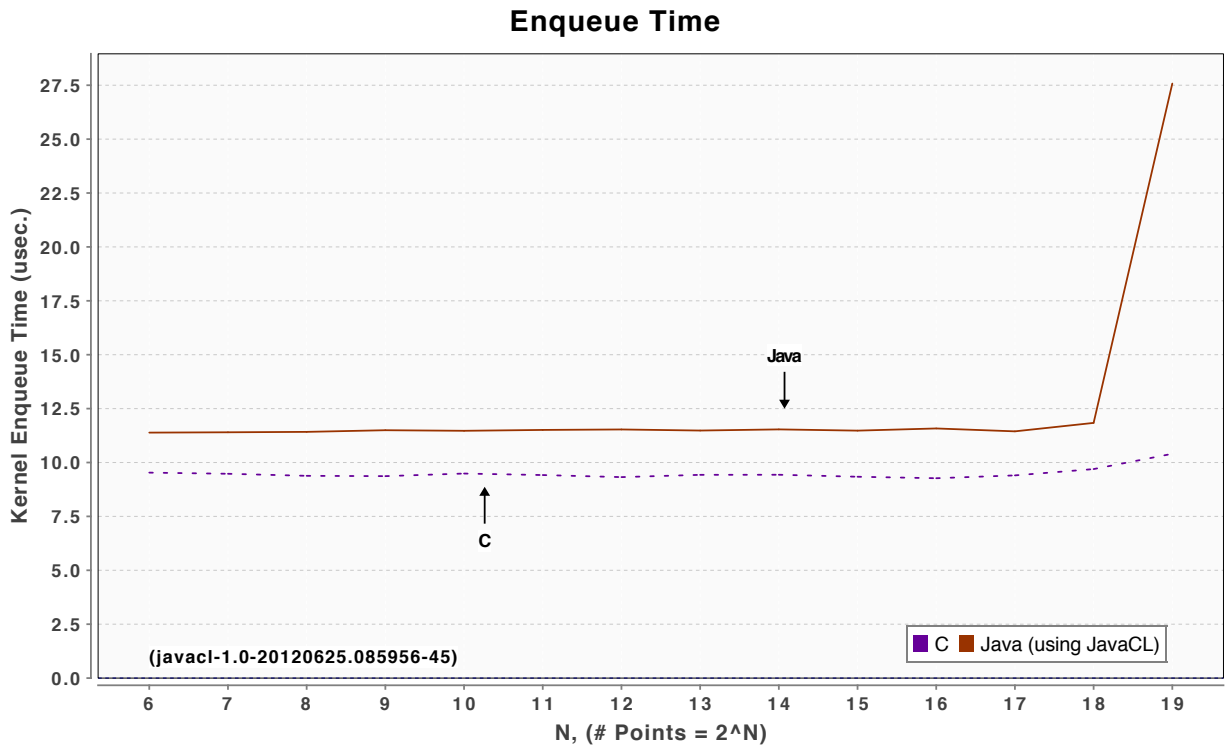

Figure 5-3. Enqueue Time Using JavaCL

6 Java Frameworks for Multi-Node Work

Code that handles inter-node communication can be tedious and error-prone to write. There are various Java frameworks that can be used in an compute cluster to simplify this. This section investigates several of these and presents performance results. Results are compared with the Message Passing Interface (MPI) which is the "workhorse" of high computing.

6.1 Comparing Grid Computing Frameworks Capabilities in the Abstract

For the purpose of this report the term "grid computing" is used to describe the use of tightly coupled compute nodes working together on a problem. Both large-scale compute problems such as weather prediction and smaller HPEC-scale problems have the need to serialize, buffer, transmit, verify delivery, and receive, re-buffer, and de-serialize data. When comparing different compute grid frameworks there will be differences in how physical assets are identified, how messages and work are managed, and how communication links are utilized. Figure 6-1 depicts these as areas for comparison.

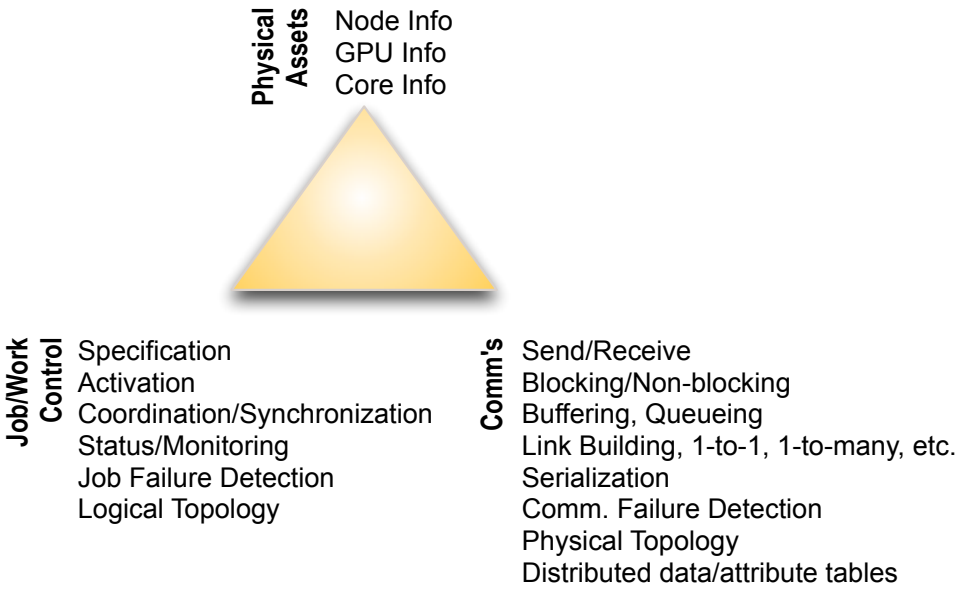

Figure 6-1. Considerations When Comparing Compute Grid Frameworks

Physical assets refers to how a framework provides information about the processors and other hardware characteristics of a node. It is often necessary to know the capabilities of the physical assets to make best use of the cluster. For example, a particular node may not contain a GPU. Such a node might not be used for high performance calculations in favor of the faster node with a GPU. The implementer can always build custom structures that describe the operating system, CPU, and GPU environments, and then use the communication facilities to convey that to other nodes. However, the greatest developer productivity is achieved if the framework already collects and propagates that information and allows the developer to add meta-data as needed.

Communications refers to the mechanisms and models provided by the framework to exchange information between the nodes. At the lowest level this includes the ability to send and receive a sequence of bytes. Other considerations include:

- Whether synchronous send/receive is supported. Synchronous communications allow data to be "copied" directly to/from application data buffers but requires the sender and receiver to block until communication is complete.
- Whether asynchronous send/receive is supported. Here sender and receiver do not wait but must make additional framework calls to test communication status. The greatest flexibility is often achieved if a separate communication thread is used, but then additional queues and buffers need to be used. A framework that provides asynchronous communication as well as the needed queuing and buffering will save the user much development time.
- Whether there is flexibility to implement all manner of one-to-one, one-to-many, many-to-one send/receive, or many-to-many communications
- Physical topology tells the application how nodes are interconnected while link building allows the application to specify a logical topology for how nodes and/or processes within a node are logically interconnected. Frameworks that support logical topology building also usually provide buffering for data and automatically handle the details of point-to-point, one-to-many, and many-to-one send/receive. This can also be a great productivity boost.
- Serialization refers to how application data, arrays, and structures are converted into a sequence of bytes for transmission. The easiest serialization is when the framework can determine how structures are organized to do this automatically without special coding by the developer. This is possible in Java, but is not possible in C (using standard C-language mechanisms).
- Communication failure detection in the simplest form allows the sender to detect that a communication has failed. In a more sophisticated framework the failure might be automatically reattempted, or rerouted to another available node.
- Distributed data and attribute tables allow the developer to insert and retrieve key/value pairs that are distributed to all nodes in the cluster. If the framework provides this, it saves the developer from writing code to send, sort, order, synchronize, and update key/value pairs along avoiding the pitfalls and race-conditions of such code.

Job/work control refers to facilities the framework may provide for concurrent execution of code in a distributed fashion. This includes:

- How jobs are specified and invoked. Each job will, in general, have some metadata that identifies a job name or id, what the inputs are, and may also include what the outputs are. One form of this is simply remote method invocation that allows code on a remote node to be invoked. The methods arguments are serialized, sent to the remote node, the method is activated, and method results are serialized and returned to the caller.
- Coordination/synchronization. A more sophisticated form of job control will provide a local queue onto which job descriptors (or job objects) are placed. Jobs on the local queue are then distributed to other nodes and executed. Framework facilities may allow conditions to be set up such that one job start might require other job completions as a precondition.

- Job status and monitoring. The ability to inspect job queues to determine what work is pending may allow data throttling and adaptive job distribution. This is not as important for applications that have a fixed input and output rate, but it is very useful for applications whose processing load may depend on the data content.
- Job failure detection. Similar to communication failure detection, this may simply mean tracking job completion with and indicator of whether the job completed normally or abnormally. A more sophisticated framework might also automatically retry the job on a different node.
- Logical topology. Since the framework works with some sort of abstraction of work, the logical topology is where the framework provides an easy way to specify that the output of one job becomes the input to one or more jobs. The timesaving here is that the framework that supports this will automatically configure the communication paths that are required between work units freeing the developer from this tedious coding.

The previous paragraphs have outlined attributes for analyzing frameworks. One additional consideration is to what degree advanced framework features are manual or automated or to what degree, they are statically or dynamically configured. For example, topology might be described statically, by configuration files that are read by a framework at runtime, or it may be programmatically specified by the application as part of application initialization, or it may be heuristically determined by the framework in response to application and network load.

6.2 Five Java Compute Grid Frameworks

Five Java frameworks were chosen. The frameworks have different models of work, data, the compute grid, and the communications. They also represent both open-source and commercial offerings. The frameworks are:

- PJ[33]
- Java Parallel Processing Framework[54] (JPPF)
- GridGain[55]
- Storm[56]
- Akka[57]

Additionally, MPI[58] was chosen as a performance baseline. MPI has been developed over many years and is the defacto workhorse for high performance computing. It has optimized support for basic cluster communication over both Ethernet and low latency communications such as Infiniband[59].

---

[54] Java Parallel Processing Framework, < http://www.jppf.org >

[55] GridGain, < http://www.gridgain.com >

[56] Storm, < http://storm-project.net >

[57] Akka, < http://akka.io >

[58] MPI, < http://www.mcs.anl.gov/research/projects/mpi/index.htm >

[59] Infiniband Trade Association - < http://www.infinibandta.org >

6.2.1 Message Passing Interface

MPI focuses on reliable message transmission among a number of nodes. MPI 2.2 was used in these studies. MPI programs are launched from the command-line in which the user specifies how many copies of the process should be launched. The copies are launched using a grid topology specification file that identifies candidate nodes as well as how many processes may be launched at each node. From the point of view of MPI all nodes can connect to all other nodes.

The same application is launched at all nodes. An MPI executive communicates with a daemon or job controller running on each node to instantiate the application copies at that node. Each instance is identified by a unique integer. This leaves it to the developer to write code to detect the id and decide which ids should handle which tasks and how they should communication with each other using those ids. Synchronous and asynchronous communications are supported. The developer must write any necessary queuing and buffering. The developer must specify how structures are serialized though MPI then uses that programmatic specification to do the serialization and deserialization as needed. There is no notion of a job. There are attributes that can be associated with different contexts. Facilities exist for synchronizing process flow across multiple nodes. There is no automated behavior to handle communication failures. There are commercial and open-source implementations of MPI that are available. OpenMPI 1.6.4 was used here.

6.2.2 Parallel Java

PJ is an all-Java framework for cluster computing. Parts of PJ have many similarities to MPI. PJ is distributed under a General Public License (GPL). The distribution includes a cluster communication API that is very similar to MPI. It also includes functionality for in-process multi-threaded programming that is similar in concept to OpenMP and was partially discussed in Section 3.2.4. Finally, PJ includes a variety of other numerical support functions and utilities that are used at RIT.

PJ applications are also usually launched from the command-line. There is a scheduler process running on a node that communicates with Secure Shell (SSH) daemons running on other nodes to start the required number of instances of the application on the other nodes. The detailed startup interactions are a little different than those of MPI though once all of the instances of the application are running the developer must handle many of the same software design issues related to buffers and queues that face the MPI developer. Serialization is a bit more capable than MPI because Java objects can be serialized with much less work due to general support in Java for object serialization.

PJ is fairly easy to set up and use although the framework had problems maintaining connections between nodes when large data sizes were being exchanged. Version 20120620 was used here.

6.2.3 Java Parallel Processing Framework

While MPI and PJ have basic facilities for cluster communication based on message exchange, JPPF has different model. In JPPF, work is distributed as jobs and tasks. A job is a list of tasks that may be executed in parallel. A task is a Java object that is either `Runnable`, `Callable`, or a subclass of `JPPFTask`. A task may also be designated as a particular method on an object that is annotated with the JPPF annotation `JPPFRunnable`. A task has inputs and a result that is serialized and available upon task completion.

Unlike MPI and PJ the entire application is not run at each node. Instead only the tasks are run when needed on selected nodes. The JPPF model of the grid is as one big queue that accepts tasks. The topology of the grid is less important. Though the applications classes can be installed on all nodes, JPPF can dynamically serialize and deploy the classes when running. This simplifies development and makes it easier to configure elastic clusters with nodes that can join or leave the cluster dynamically.

Jobs can be submitted synchronously or asynchronously. JPPF has some tolerance to failures because if a node becomes unavailable it will not be used for subsequent task execution. Furthermore, each job has an expiration time that could be used to detect work that has already been dispatched but has failed to complete. JPPF also uses heartbeats to detect when nodes fail and can automatically re-queue a task onto a different node than the one that failed.

The JPPF deployment architecture has at least one JPPF server and one JPPF compute node. Nodes get their work from a server and there may be multiple nodes connected to the same server. There may be multiple servers that are connected to each other which will distribute work between them. The topology is configured by a configuration file and can be a mix of predefined connections and automatically discovered connections. Predefined connections use Transmission Control Protocol/Internet Protocol (TCP/IP). Discovered connections use User Datagram Protocol [UDP] multicast.

When the main application runs, it makes a connection to a server and is then able to submit work to the grid. Regardless of the connection relationships between servers and compute nodes, a client sees the entire grid as a simple queue. Jobs are distributed from the queue using one of several available load-balancing algorithms. It is also possible to write a custom load-balancing algorithm.

JPPF is available as open-source under an Apache 2 license. Version 3.3.4 was tested in this investigation.

6.2.4 GridGain

GridGain is a commercial product that evolved from an earlier open-source development by the same principal developers. It is available for three different types of grids—compute grids, data grids, and map-reduce grids. The compute grid version was used for this study.

GridGain is a very flexible framework that allows work to be distributed to an abstract grid. The grid is abstracted to a greater degree than what JPPF does. Grid nodes are all logical peers to each other and have attributes that can be used when work is dispatched. User code can attach attributes to nodes though some grid node attributes are automatically added by the system including environment variables, JVM Runtime properties, and, operating system name/machine architecture/version.

Node attributes are also dynamically updated with over 50 metrics including:

- average CPU load
- average number of waiting jobs
- garbage collection CPU load
- amount of file system free space
- memory usage metrics

- communication message counts
- available number of CPUs.

In GridGain a task contains jobs. This is opposite to JPPF in which jobs contain tasks. A task is an object that can be created as a subclass of one of several predefined base classes. GridGain tasks are submitted to the entire grid or a subset of the grid. A grid subset can be created by the client application by using node attributes to sub-select candidate nodes.

Work can also be submitted as a byproduct of calling methods that have been compiled with the `Gridify` annotation. This is very attractive for development and maintenance of more intricate applications because method calls are easy to write and understand. The framework takes over the complexities of argument and return value serialization as well as those of dispatch to the grid.

The previous frameworks utilized daemons to instantiate the application or its worker processes at various nodes. With GridGain an instance of GridGain is run at each network node. Each instance participates in the grid and can run tasks directly—additional operating system (OS) processes are not required. Client code can be run as part of one node's GridGain instance, or the GridGain instance can be started as part of the client code. The client code accesses the entire grid via its local GridGain instance. Furthermore, it is possible to have client code at any GridGain node—each client can invoke distributed services executed among all other nodes. This is very attractive for fault tolerant computing because there does not need to be an explicit master node. The user code could detect when the current active master node fails and then can assume those responsibilities at another already active node. The algorithm for this is, of course, up to the user.

GridGain can autodeploy code to any node that needs it. Job arguments and return results are auto-serialized. The user can write custom serializers if desired. There are various types of load-balancing available including ones that factor in a node's actual CPU load into the distribution algorithm. The work distribution algorithm can also be customized. It is also possible to customize the handler for failures although handlers are already provided that will resubmit work to a different node if one fails.

GridGain was the easiest to install and use of all the frameworks tested. As a commercial product it also has the most extensive documentation. Version 4.3.1e was used for testing.

6.2.5 Storm

Storm is focused on distributed streaming computation. In this model, client work is decomposed into several small computation stages that are interconnected. The input stages are called "spouts" and the others are called "bolts." The client application is implemented as a data-flow network with these spouts and bolts interconnected in a topology that is submitted as part of a small application. Data are represented as n-tuples. The n-tuple components are composed of serializable Java objects and can be simple or as complex as needed.

The I/O relationships between the spouts and bolts define a topology that is specified by a "starting" application that submits the topology to special framework daemon named "Nimbus." Nimbus then communicates with the Storm compute node daemons (supervisors) that will instantiate sprouts and bolts as needed. The sprouts and bolts are instantiated as new OS processes with socket connections to each other.

One unique aspect of Storm relative to the other frameworks investigated is that the framework can easily create multiple instances of the same spouts and bolts to implement parallel processing. Stream tuples that are directed to a parallelized bolt will be automatically "stripped" across the bolt instances to distribute the load.

Storm spouts and bolts can be written with or without tuple delivery acknowledgment. Failures can cause tuples to be resent although this is done by user code upon detection of missing acknowledgments. Spouts and bolts should maintain no state and be capable of being quickly restarted.

Storm is early in its development life. There is an assumption that all the processing nodes are equally capable when parallelized bolts are instantiated. This leaves it to the developer to write specialized schedulers to handle more processor-aware assignment of bolt instances. There is no single work queue onto which work can be queued. Any one-of-a-kind processing has to be done in response to a "trigger" tuple sent to a node.

Storm is open-source and is available under an Eclipse Public License. Storm is relatively new and lacks some of the "polish" of the older frameworks. Storm runs on the JVM but is written in Java and Clojure. It requires the use of Apache Zookeeper[60], ZeroMQ[62], and the Java bindings for ZeroMQ, all of which must be installed and configured separately. Storm version 0.8.2 was used for testing.

6.2.6 Akka

Akka presents a more evolved message distribution model known as an actor[61] model. With MPI and PJ, messages are sent to compute elements based on an integer node id. With actors, messages are sent to code objects that are identified symbolically. A reference to an actor is looked up using a symbolic identifier assigned by the developer. The reference is used for subsequent delivery. Unlike MPI and PJ's integer id, the actor's symbolic identifier does not depend on the grid size. The referencing mechanism is used whether the actor is local to the sending process or remote. This allows an application to be written initially as a number of local inter-communicating actors that can later be easily instantiated at different nodes.

Message sending is asynchronous. Any serializable Java object (including tuples of Java objects) can be sent as a message. Actors may maintain state and may change their behavior as needed. Unlike Storm, the I/O relationships between the actors are not captured in a separate static topology—each sender determines the receiver for every message when it is sent.

Akka supports clustered operation though the user has to write some glue code to recognize cluster events such as nodes joining or leaving the cluster. Akka supports message distribution to parallel instances of actors on different nodes. The framework provides predefined actors known as routers that are very useful for distributing messages to parallel instances of actors. However, using these takes a little more developer work and understanding than non-router distribution.

---

[60] Apache Zookeeper, < http://zookeeper.apache.org >

[61] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A universal modular ACTOR formalism for artificial intelligence. In Proceedings of the 3rd international joint conference on Artificial intelligence (IJCAI'73). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235-245

Akka has various types of routers including an adaptive load balancing router that uses runtime CPU metrics to determine which workers to route messages to. The metrics include memory utilization, load average, CPU utilization, or mixes of these. Version 2.1.2 was used for testing.

6.3 Feature Comparisons

The previous section provided an introductory description for each framework. Table 6-1 and Table 6-2 compare aspects of these frameworks that are of interest to those designing multi-node compute applications.

Architecture refers to the relationships between nodes. In some of the frameworks there is a master node, for example, that may launch other nodes or take a special role in I/O. Node and worker topologies may be fixed for the duration of the application or they may support ad hoc entry/exit of compute nodes. In the latter case, the worker code is typically distributed dynamically among whatever physical nodes are available.

Distribution model refers to whether messages or work is distributed. Of course, to distribute work the framework has to move data but the details of the messaging are handled by the framework. In the case of Storm, data tuples are explicitly output from spouts and bolts but the target of the tuples is not known by the spout or the bolt. The target has been identified by the topology that was submitted to the framework for execution. In contrast, in Akka, the sending actor explicitly identifies the actor that will receive the message.

Frameworks may or may not start separate OS processes to perform work. GridGain and Akka are the only ones that do not do this.

Load balancing refers to a framework's ability to distribute work across available nodes. Simple algorithms will just distribute the load in a round-robin fashion to available resources. More sophisticated algorithms will factor in past execution time to predict the loading impact of current distribution decisions. Other algorithms will also factor in the node's current CPU and memory loading. JPPF, GridGain, and Akka all provide various forms of load balancing as well as provisions for user provided load balancing algorithms.

Recovery from node failure describes whether the framework facilitates recovery from node failure without imposing much work on the developer to detect and take corrective action. MPI and PJ are at a disadvantage because their integer id-based process identification makes it hard to easily recover when a specific id is down, requiring the user code to do some intricate reassignment of the work to other nodes.

Table 6-1. Framework Feature Comparison (1 of 2)

Feature	MPI	PJ	JPPF	GridGain	Storm	Akka
Version Used	OpenMPI 1.6.4	20120620	3.3.4	4.3.1e	0.8.2	2.1.2
Architecture	Fixed peers; Fixed grid	Master/ slave; Fixed grid	Master/ slave; Dynamic topology	Dynamic number of node peers	Dynamic number of node peers	Dynamic number of node peers
Distribution model	Message passing	Message passing	Job/task submission	Task/job submission; Distributed remote function and method invocation	Data tuples w/ implicit message targets	Actor w/ explicit message targets
Starts OS processes for "workers"	Yes	Yes	No	No	Yes	No
Load balancing	No	No	Yes	Yes	Message "striping"	Yes
Recovery from node failure	No	No	Yes	Yes	Yes	Yes
Inter-node communication	TCP/IP*; RDMA; Other	TCP/IP; UDP/IP	TCP/IP; UDP/IP	TCP/IP; UDP/IP	ZeroMQ[62]	TCP/IP; ZeroMQ possible
Serialization	Descriptors provided by application	Automatic	Automatic	Automatic	Automatic	Automatic

[62] ZeroMQ is an implementation of Pragmatic General Multicast over UDP/IP. See, The Simplest Way to Connect Pieces – zeromq < http://zeromq.org >

Inter-node communication identifies the key protocols used for communication. MPI can use remote direct memory access[63] (RDMA). RDMA allows MPI to exploit features of network interface cards that let one node to directly place information into another's memory. ZeroMQ is an API and protocol that facilitates message-based communications.

Serialization refers to the need to take client data, arrays, and data structures and assemble them into sequential bytes for transmission. With MPI the structure of the data has to be provided declaratively by the application itself. With Java the data is an object which can be serialized automatically if it follows some basic conventions.

Unit of execution refers to how code is transferred and executed at the multiple nodes. In MPI and PJ the entire application is run at each node and it is up to the application itself to decide what to do at each node. The other frameworks provide more fine-grained portions of the code for activation. JPPF, GridGain, and Storm have the ability to transfer the necessary code to the nodes at runtime.

Shared and distributed work queuing is where processes on any node can queue additional work to be not just executed, but also distributed across the grid nodes. Without framework support for this, the developer must implement queues that can be written to/from any node, code to take work from the queues and distribute them to the worker nodes, and code to send the results back to the node that originated the work request.

Inter-task or inter-process synchronization across nodes is where there are mechanisms so that code on one node can synchronize with code on another node. One example of this is a barrier that requires multiple codes to reach a certain point before any of them can continue.

Work distribution based on node attributes is where a task or job can be "sent" to a node based on an attribute assigned to the node. This is useful when the node has special I/O capability or compute capability that needs to be utilized to do the work.

Message distribution based on data attributes is where data is sent to a node based on the data itself. This is useful, for example, for data analytics which may benefit by having all data reductions for a type of data performed at the same node.

Implements for/join semantics across nodes refers to the ability to start execution on one node, start (fork) execution of multiple code sections on other nodes, then when the other nodes complete, execution continues on the original node (or on only one of the other nodes). Direct framework support saves developer work and errors. On the other hand, while this is very handy for some types of algorithms it only makes sense for the job and task oriented frameworks.

Shared synchronized data between nodes allows multiple nodes to have the same view of data even after modifications are made. Regardless of the order that data is modified in, from different nodes all nodes see the same order of data changes.

[63] RDMA Consortium, < http://www.rdmaconsortium.org >

Table 6-2. Framework Feature Comparison (2 of 2)

Feature	MPI	PJ	JPPF	GridGain	Storm	Akka
Unit of execution	Entire replicated application	Entire replicated application	Individual jobs and tasks that can be deployed dynamically	Individual jobs and tasks that can be deployed dynamically	Individual code spouts and bolts that are deployed dynamically	Actor objects that are deployed statically
Shared and distributed work queuing across nodes	N/A	N/A	Yes	Yes	As a response to stripped messaging	Yes but some "glue" required
Work distribution based on node attributes	N/A	N/A	No	Yes	N/A	Almost - Akka recognizes "node roles" but user code has to use them
Message distribution or data partitioning based on data attributes	No; requires hand coding	No; requires hand coding	No; requires hand coding	Yes	Yes	Almost – Could be implemented with a custom router
Implements fork/join semantics across nodes	N/A	N/A	Yes with the Fork/Join add-on	Yes	N/A	N/A
Shared synchronized data between nodes	Using shared file mechanism	No	Yes with a DataProvider	Yes, several mechanisms provided	No	No
Inter-task or inter-process synchronization across nodes	Yes	No	No	Yes	No	No

This has been a quick overview of these five JVM frameworks. Four of the frameworks, JPPF, GridGain, Storm, and, Akka free the developer from low level message "massaging" such as serialization, buffering, and queuing. They also have significant pre-build capabilities for load balancing, concurrency, and failure recovery. Those knowledgeable with any of these frameworks may feel that these descriptions have been too simplistic. However, the goal was to guide others who don't have experience with these frameworks and who need an initial impression of what the differences are.

6.4 Framework Performance

One expects that frameworks with lots of features will not perform as well as smaller frameworks with fewer features but where more attention has been given to performance. This section investigates framework performance. It measures two simple metrics—latency and throughput.

A simple ping-pong test was used to measure latency and throughput. Both a synchronous and an asynchronous version of the test were created for each framework. In the synchronous test, Node A sends a fixed size message to Node B and waits for a response. Upon receipt of a message from Node A, Node B returns a message of the same size back to Node A. This is done repeatedly. In the asynchronous test, Node A sends multiple messages to Node B but does not wait for a response. Node B responds to each message as it receives it.

The asynchronous test requires data to be queued by the first node because the rate at which data is produced is faster than the rate at which it is transmitted. Thus, the asynchronous test provides data about queuing and buffering capabilities of the framework. MPI and PJ do not provide queuing and buffering facilities beyond those that might be part of the underlying communications software stack. Therefore, the investigator added round-robin queuing and buffering as part of the MPI and PJ benchmark code. This was not needed for the other frameworks.

Each experimental run exchanges a fixed amount of data varying in length from approximately 80 bytes to approximately 650 kilobytes.. With MPI, the buffer size can be controlled because in C one has direct control over the data being sent. With Java, the data is contained in a Java object. The object is shown in Listing 6-1. Serializing a Java object adds descriptive metadata to what is being sent so that it can be safely reconstructed at the remote node. Thus, the actual byte-count transmitted is greater than the size of the user's data. For the object shown in Listing 6-1 the size is 182 bytes of metadata plus 36 bytes for the non-array fields plus the number of bytes for the array data. Some of the frameworks use standard Java serialization though others have framework-specific serializers. Some of the frameworks allow the user to create more efficient serializers. There may be even more overhead because for a task/job oriented framework such as JPPF the job/task object must be serialized in addition to the user data.

Listing 6-1. Java Object Used In Ping-Pong Tests

```
public class RunInfo implements Serializable {
    private static final long serialVersionUID = 9478L;
    public int jobId;
    public long creationTimeNanos;
    public long remoteReceiptTimeNanos;
    public long intervalNanos;
    public long durationNanos; // unused but present
    public long[] data;
}
```

In the benchmarks the message sent from the first node is tagged with the time in nanoseconds just before the framework call to send the data. When the second node receives the data it adds the time of receipt in nanoseconds to the message and returns the same message. Latency can then be determined by comparing the two times.

Special care was taken regarding the use of the server's clock. Nanosecond time on machines is typically relative to machine startup. Millisecond time is usually available as an absolute time. The two can be carefully combined to get an absolute nanosecond time that can be compared between machines. Also, standard clock hardware can drift significantly over the course of minutes. Calibration software was developed to measure relative server time as part of each test run. The calibration software used is the JNodeToNodeClockSkew tool that has been released[64] to the public domain. The technique was able to resolve server clock differences repeatedly on the test hardware to approximately 2 µs.

Testing was performed on a 4-node cluster of Intel Xeon E5620s connected with a dedicated 10 gigabit Ethernet switch. A test to measure the "best" case latency was performed in C and Java. The best Java latency was measured using the JNodeToNodeClockSkew tool. The best C latency was measured using the same algorithm as used by the JNodeToNodeClockSkew tool but implemented using MPI. Table 6-3 compares measured end-to-end software latencies. Java exhibited an additional 13 ms latency. Though it may be possible to implement a faster Java test, the goal here is to have a reference minimum with which to compare the Java grid-framework latencies.

Table 6-3. Measured Best Case Latency

Language	Latency
Java (TCP sockets)	33-41 us
C (using MPI)	20-27 us

Figure 6-2 graphs the one-way latencies for each framework using the framework provided send/receive APIs. One can see that there is a significant amount of Java framework latency above and the best case. Akka exhibited the smallest latency.

[64] Poor Man's Timing and Profiling, < http://jtimeandprofile.sourceforge.net >

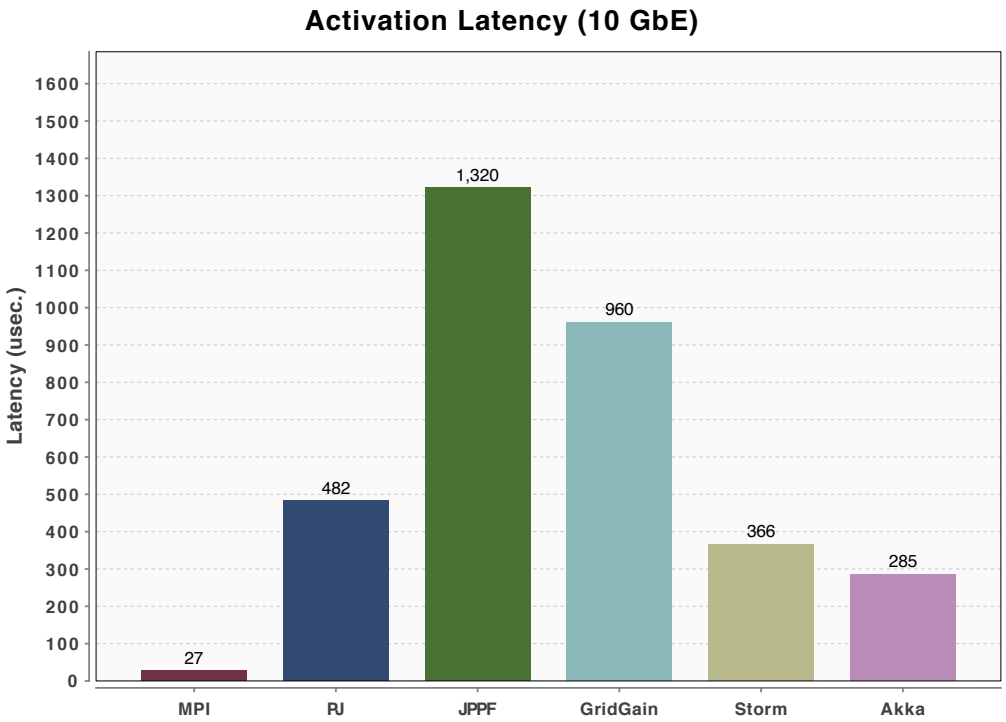

Figure 6-2. Comparison of Framework Latency Over 10 GbE

Figure 6-3 plots synchronous activation rate as a function of message size. The message size is the size of the user data. As expected the highest message rate is achieved by MPI. JPPF has the lowest rate. The best synchronous Java performer, Akka, was about a factor of 11 slower than MPI for small data sizes. For large data sizes, where the Java serialization overhead is less significant, Akka has 70 percent of the rate of MPI.

Figure 6-3 also plots the number of message exchanges that were achieved using just low level Java TCP socket I/O. This size is the true message length in bytes because for that curve there is no serialization. It might seem odd to compare this no-overhead raw-byte-count curve with the post-serialization Java curves. However, the purpose is to show the capabilities of the various options as seen by the end user who has little control over serialization overhead unless the user writes their own serializer. The Java socket I/O curve also shows that in this synchronous ping-pong test Java itself is not the limitation to throughput. The lower performance of the Java frameworks is likely due to a combination of two factors. One factor is the additional work performed by each framework to implement each frameworks features. The second factor is less efficient I/O design.

Figure 6-4 plots the asynchronous activation rate. These curves should be higher than their synchronous counterparts because new messages or task activations can be queued before the previous have completed. Again Akka has the best Java rate and JPPF has the lowest. There are two MPI curves shown. The MPI-A1 test maintains a single outstanding send buffer. The MPI-A4 test maintains 4 buffers that are used in a round-robin fashion to allow 4 outstanding activations.

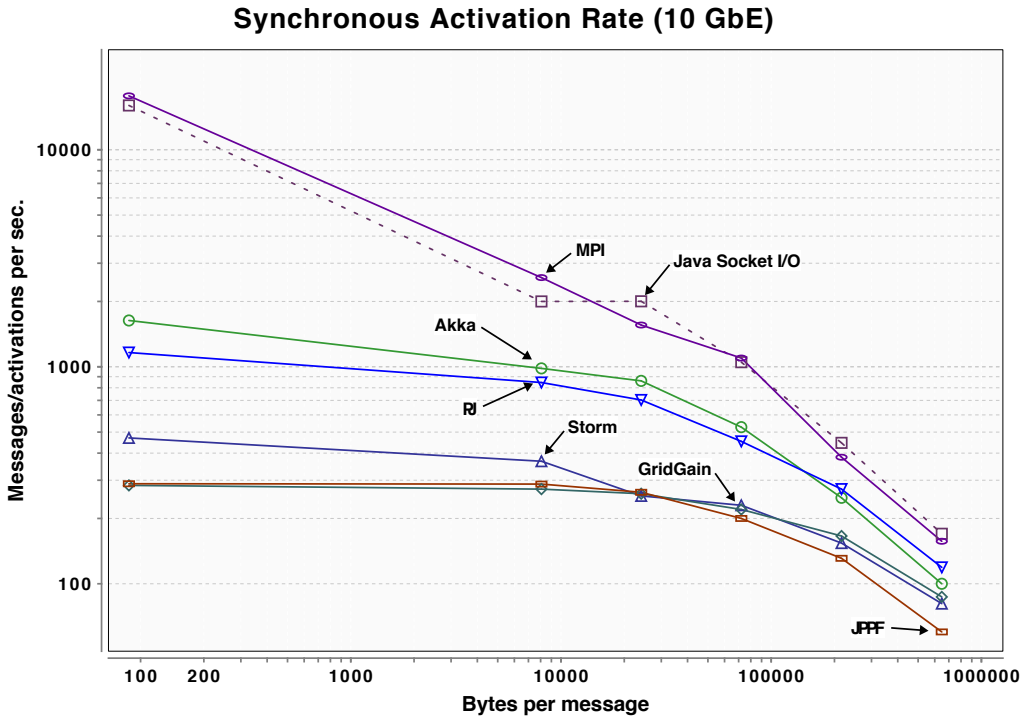

Figure 6-3. Synchronous Activation Rate Over 10 GbE

For small messages, Akka's asynchronous activation rate was one-fourth and one-seventh slower than MPI, depending on the MPI curve. However, it was 30 and 40 percent faster for large data sizes. The reason for this may be due to structural differences in the tests. In Akka an actor can only respond to an incoming message if it is not busy sending a message. Thus, it was necessary to implement Akka message sending in a distinct thread so that the actor could asynchronously handle incoming messages. The MPI implementation maintained a fixed number of outstanding sends but serviced both send and receive in the same thread.

Figure 6-5 plots synchronous throughput and Figure 6-6 plots asynchronous throughput. Since throughput is just the activation rate multiplied by the data size the relative performances shown are the same as for the activation rate curves.

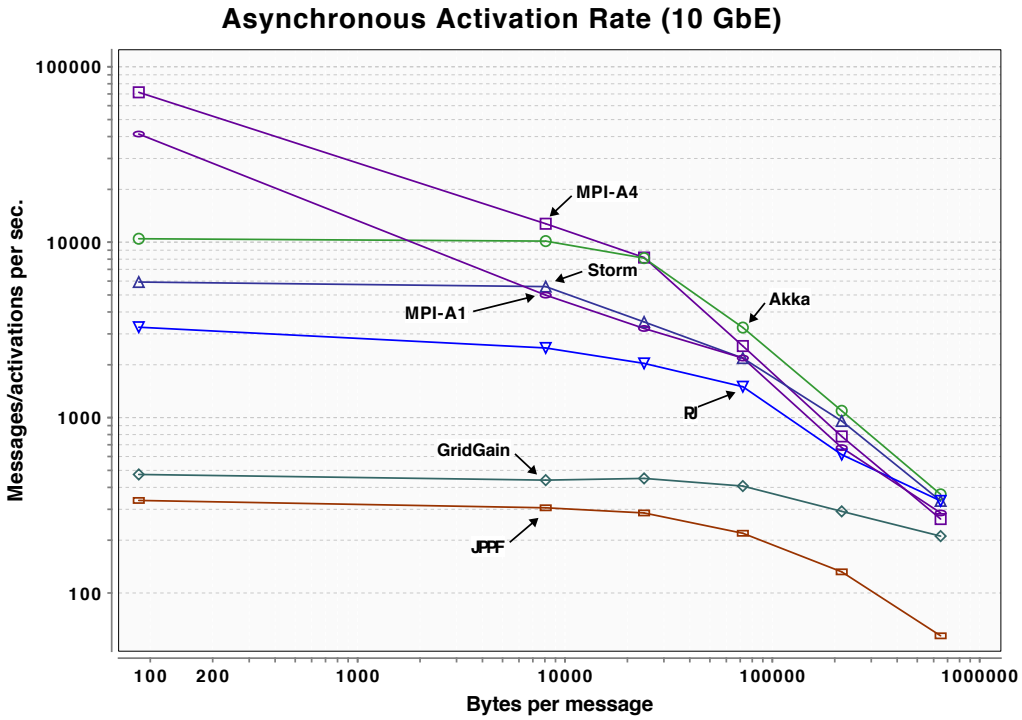

Figure 6-4. Asynchronous Activation Rate Over 10 GbE

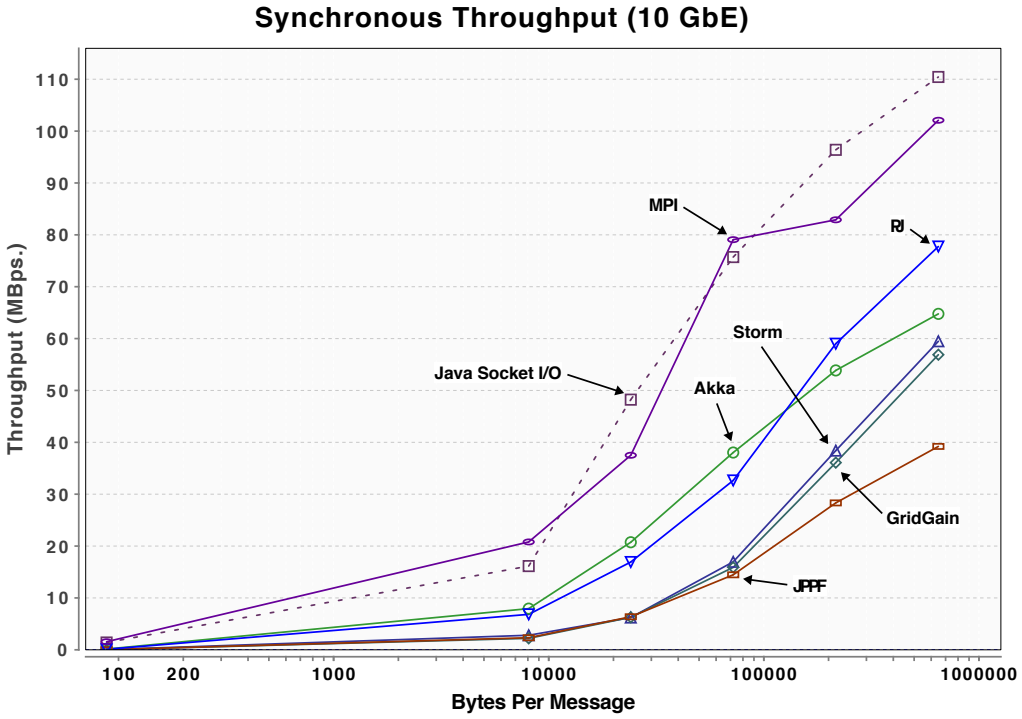

Figure 6-5. Synchronous Throughput Over 10 GbE

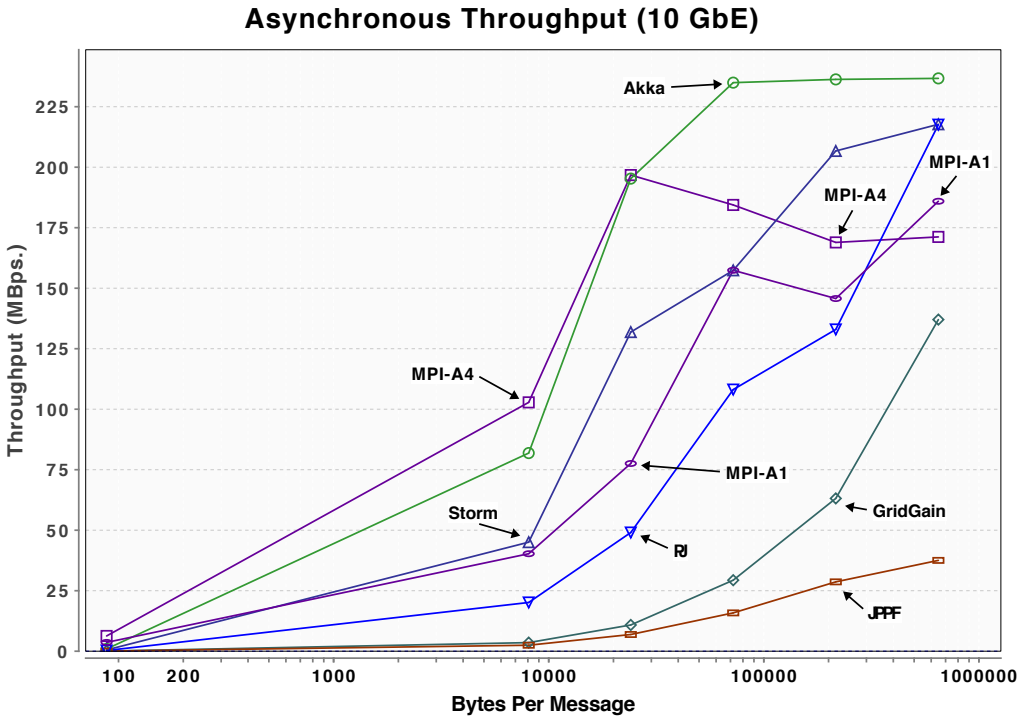

Figure 6-6. Asynchronous Throughput Over 10 GbE

6.5 Java Framework Summary and Observations

This study was motivated by the desire to more quickly develop intricate high performance and HPEC applications including multi-node applications. In this section we examined 5 Java compute-grid frameworks to understand capabilities, architectural models, and expected performance. PJ[33], JPPF[54], GridGain[55], Storm[56], and, Akka[57] were tested on a 10 giga-byte Ethernet cluster. The frameworks have different architectures, deployment models, ease of setup, and, ease of use. There are different options for remote method invocation, distributed task and job queuing, multimode synchronization, actor communication, data flow processing, load distribution, and failure recovery. GridGain has the most built-in features and was the easiest to setup and use for distributed method invocation though it was not the fastest. The actor and data-flow models of Akka and Storm may be more "natural" for some types of problems.

It was expected that performance would be less than that of the high performance community's Message Passing Interface[58] (MPI) since several of the Java frameworks offer greater capability and flexiblity than MPI which often comes at the expense of lower performance. None of the Java frameworks performed comparably to MPI for small message sizes. Java serialization overhead increases the data payload to exchange messages and invoke remote methods. If the data size is just a few hundred bytes long then the developer who needs more performance will want to write a custom serializer to avoid the default serialization overhead.

However, with larger data sizes both Akka and Storm were found to sometimes exceed one, or both, of the MPI implementations. This was likely due to differences in the detailed implementation of the test codes. These must be different due to significantly different API's and interaction models. If this is the case, it is significant that a design difference may account for the better Java performance. When one reaches the point where design differences are key then one

has passed the point where the answer to the performance question it is simply about whether the framework is a C or Java framework.

Also, the plain MPI performance may be misleading for a more intricate application. If advanced queuing and inter-node communication features are required the developer's code will add to the MPI per-message-overhead. This will decrease performance.

A performance application often has numerical performance requirements. Some classes of applications require job recurrence times that are on the order of 100 jobs per second. The tests show that most of the frameworks can readily support that rate. At 1000 jobs per second three of the Java frameworks can support the rate for moderate data sizes.

Regarding language and JVM capabilities, testing showed that Java socket I/O latency (via the `java.io` package) was about 13 microseconds higher than MPI's 27-microsecond latency on the 10 gigabit Ethernet network. However, JVM socket overhead was not the main part of framework latencies which were much higher than that. The observed Java framework latencies, ranged from 236 to 1320 microseconds. Though we could not analyze why, we must assume that the large framework latencies are due to some combination of the grid/data models, framework software architecture and perhaps un-optimized design implementation. For example, with Storm, data must be "wrapped" as tuples. Tuples facilitate filtering and data grouping, but recognizing them and manipulating them also adds to transmission latency.

There are other aspects of multi-node performance that were not explored in this investigation. Factors for future investigation include:

- network load – how much overhead a grid framework requires to support features such as auto-discovery of peer node participants and to communicate node status
- memory load – how much processor memory is required when a framework runs
- scalability with increased number of nodes – how effectively additional nodes can be used to distribute the load and how busy nodes can be maintained for busy applications
- time to detect failure – how quickly a failed node can be detected so that work can be reassigned or messages can be rerouted.

7 Writing Java Code for Performance

This section describes coding practices to help get the best performance from Java applications. The practices are based on the investigator's experience and those of his peers.

7.1 Minimize Memory Allocation within Repeated Processing Paths

A memory intensive test described in a previous section found a 15 percent overhead due to garbage collector activity on that test. Other tests show increased execution jitter due to memory allocation/deallocation. One clear way to improve Java performance for Oracle's Java 7 Standard Edition is to minimize memory allocation, especially on repeated processing paths.

Garbage collection is a great benefit but its convenience and transparency can lead developers to overuse dynamic memory allocation. The garbage collector has been the focus of continued development for many years not just by Oracle/Sun, but also by vendors of other JVMs. Oracle's Java 7 garbage collector, G1[65], is the latest in the Oracle/Sun line of improvements.

On the positive side, the developer is freed of keeping track of unused memory structures. On the negative side, the garbage collector is an additional thread that must periodically run to recover the memory of unused structures. The scanning takes time that detracts from the main processes. The times when the garbage collector run are unpredictable from the point of view of the application, contributing to execution jitter.

Note that simple data structures allocated for use in a limited scope of code may not cause extra garbage collector work. If it is determined at run time that an object instantiation is not used outside of the scope[66], the object may be allocated on the stack. If the object is simple enough, it may not even be created[67]. In Java 6, this escape analysis was off by default, but in Java 7 it is enabled by default.

7.2 Preallocate Working Buffers and Structures

For non real-time Java the best thing to do is to give the garbage collector as little work to do as possible. Preallocating working buffers and structures helps accomplish this. This includes double buffers used to allow separate threads to move data and process simultaneously.

Since preallocation tends to "break" object encapsulation, it is not typically done in Java code. Under the object philosophy, an object manages its own allocation of working structures without regard for reuse, relying on the managed memory mechanism to clean up. Thus, "performance" Java coding has to take exception to this design practice in favor of the more efficient practice of preallocation.

---

[65] The Garbage-First Garbage Collector, <http://www.oracle.com/technetwork/java/javase/tech/g1-intro-jsp-135488.html>.

[66] [Choi99] Jong-Deok Choi, Manish Gupta, Mauricio Seffano, Vugranam C. Sreedhar, Sam Midkiff, "Escape Analysis for Java," Procedings of ACM SIGPLAN OOPSLA Conference, November 1, 1999

[67] Goetz, Brian, "Java theory and practice: Urban performance legends, revisited," Sept. 2005, <http://www.ibm.com/developerworks/java/library/j-jtp09275/index.html>.

7.3 Use StringBuilder for Repeated Concatenation

Some Java applications slow down because of excessive use of the `String` class for parsing and concatenation. Java strings are immutable. Concatenating two strings means that a third string must be created and data must be copied from the two constituent strings.

The Java `StringBuilder` class will internally allocate character arrays and reallocate bigger ones if required. Appending to an existing instance simply uses preallocated space within the object's internal array. The instance of the `StringBuilder` can also be reset to zero length and reused avoiding the overhead of memory allocation and potentially additional garbage collector work.

It is best to estimate how big the `StringBuilder` should be when it is instantiated to reduce intermediate reallocation of internal arrays. Note that the `StringBuffer` class is similar but offers thread-synchronized method calls that may add overhead.

7.4 Use SLF4J For Good Logging Performance and Simple Syntax

One common source of inadvertent object allocation is in debug statements. Consider the following snippet using the Log4J[68] logging library:

```
for (i = 0; i<10000; i++){
   cum += i;
   log.debug("Iteration "+i+" cumulative == "+cum);
}
```

This form of the debug method takes one argument, forcing the developer to concatenate to build the proper information for debugging. If debug statements are not being logged, concatenation is wasted work. On the other hand, using SLF4J[69] avoids the concatenation. With SLF4J this can now be written as:

```
for (i = 0; i<10000; i++){
   cum += i;
   log.debug("Iteration {} cumulative == {}",i, cum);
}
```

If there are many debug statements for unlogged information, the time savings to deployed code can be significant. A simple experiment was devised to measure this. Logging was disabled and 10,000 statements were logged as in the code fragment of the previous paragraph. Log4J (with the concatenations) required 3.5 ms while SLF4J required 1.3 ms. Of course, if console logging is enabled, the time to actually log a string usually swamps these numbers.

For the sake of completeness, it should be mentioned that there are two other logging options available: Apache commons logging[70] and Java standard logging[71]. A detailed comparison of the logging packages is beyond the scope of this study. Note, that Apache Commons logging also "suffers" from the problem of having only one argument in its debug method, forcing the

---

[68] <http://logging.apache.org/log4j/>

[69] <http://www.slf4j.org/>

[70] <http://commons.apache.org/logging/>

[71] Java standard logging is part of the standard J2SE distribution and can be found in package java.util.logging.

developer to have unneeded string concatenations. Java standard logging, does not have this problem but has method arguments that are more awkward to use (in the author's opinion). This tends to cause developer to not use it. With built-in logging, the equivalent code would look like the following:

```
for (i = 0; i<10000; i++){
   cum += i;
   log.log(Level.FINE, "Iteration {} cumulative == {}",
               new Object[]{i,cum});
}
```

This avoids the string concatenation but requires creation of an array of objects, which still leads to inefficiency. The time to execute the above loop with logging disabled was 2.5 ms.

SLF4J has the added advantage that it is a thin tier on top of either Log4j, Java logging, or Commons logging. The developer has the advantage of coding using a convenient API with good performance while "hedging their bet" on the underlying logging infrastructure.

7.5 Minimize Iterator Creation to Reduce Jitter and Garbage Collector Activity

The `java.util.Iterator` interface is widely used in Java's collection classes. However, creating an iterator requires object instantiation and future garbage collection. Though the developer cannot change the design of the standard Java libraries the developer may be able to reduce iterator creation for their own collection classes if:

- Only a single thread will iterate over the collection.
- The collection contents do not change over the course of the iteration.

If these conditions are met, then there are various ways to reduce iterator instantiation. One way is to have the collection class instantiate an iterator only if the contents have changed. The iterator implementation can have a way to reset the iterator before the next use. This reduces the number of iterators that are created if the collection changes slowly.

If there are multiple threads accessing a collection, it is possible to create an iterator-per-thread using the `java.lang.ThreadLocal` facility.

Finally, there is at least one library developed to minimize iterator instantiation; Javolution[72]. Developers concerned with jitter should look to see if Javolution's collection classes are suitable.

7.6 Consider Alternative Collection Classes

The standard collection classes are based on objects. Maps and sorted lists require comparators that handle the appropriate object types. When the keys or list contents are primitive types, the collections can be specialized for faster insertion, search, and removal. There are alternative

[72] See Javolution, http://javolution.org/

collection classes that have been developed. The reader should be aware of Fastutil[73], Apache commons primitives[74], Guava[75], Trove[76], and Apache commons collections[77]. There are others.

7.7 Inheritance and Use of the Final Keyword

Invoking subclass methods in Java is efficient. It might seem that the deeper the inheritance chain is the less efficient method invocation would be. This is not the case. Java method lookup involves a single indexed lookup in an internal table.

> Method lookup and invocation time is constant relative to the inheritance depth.

We expected and confirmed that using the "final" keyword would provide a hint to the Runtime that calls to a final method could be optimized. A test was devised to measure the time to call methods of a parent class with various children. The class is shown in Listing 7-1.

Listing 7-1. Parent Class

```
public class Parent {
    static int sa;
    static final public int m4(){ return sa; }
    static final public int m5(Parent p){ return p.a; }

    int a;
    public int m1(){ return a; }
    public int m2(){ return a; }
    final public int m3(){ return a; }
}
```

In the class, `m1()` is not final; `m3()` is final. The test code instantiates subclasses but invokes methods via references typed as `Parent`—such that the Runtime cannot tell that the non-final class has not been overridden. After suitable warming, 100,000,000 invocations of `m1()` took 314 ms. The same number of invocations of m3() required 100 ms—three times as fast. In highly recursive algorithms with small method bodies, method call times can be a significant part of total runtime. From the experiment, one can generalize the following:

> Mark methods in parent classes that will never be overwritten using the "final" keyword.

The time to invoke the final static method `m4()` 100,000,000 times was very fast—5.3 ms. The JVM probably in-lined the method.

> Class methods that don't require access to instance fields should be marked "static" for potential speed savings.

---

[73] See fastutil, <http://fastutil.dsi.unimi.it/

[74] See Apache commons primitives, <http://commons.apache.org/primitives/>

[75] See Guava, <https://code.google.com/p/guava-libraries/>

[76] See Trove, <http://trove.starlight-systems.com/>

[77] See Apache commons collections, <http://commons.apache.org/collections/>

7.8 Avoid Allocating Java-Side Data If It Is Repeatedly Passed to Native Libraries

Converting Java data so it is natively accessible by C libraries takes time. If there is a sequence of native processing operations it is better to use BridJ or SWIG to allocate buffer space natively to avoid this conversion time. Of course, there are many considerations here such as which data should be allocated natively, which should be allocated in Java, which should be duplicate, when to copy it back and forth, etc. But these are application-specific considerations that have to be addressed specifically.

7.9 Use Lazy Instantiation and Deferred Initialization If Application Startup Needs Expediting

Application initialization will generally be longer for Java than for C as discussed in a previous section. Different vendors have different ways to expedite loading including class ahead-of-time (AOT) compilation and caching. The developer can code classes to do minimal state initialization at class loading and instantiation. These can be deferred until the first time a class is used, though that then adds to the time for the first execution. Even the instantiation of singleton classes might be deferred until first needed.

7.10 Cache Read-Only References to Instance and Class Fields

When accessing read-only class and instance fields from a method, it is generally faster to retrieve them once within the method then use the local method variable throughout the rest of the method. This also applies within inner classes where access to a final variable is really access to an anonymous classes instance variable.

7.11 Alternatives to Conditional Compilation

A standard though underutilized Java feature is the use of assertions. In Java, an assertion is a Boolean expression that is expected to return true. The expression can even be a function invocation that returns a Boolean result. Java assertions are disabled by default. If they are disabled, the condition of the assertion is never actually evaluated. When disabled, assertions have a very small run time impact.

In C, one might accomplish selective use of development-time assertions using C macros. But that requires recompilation for deployment to remove them from the executable code. Using Java assertions for development can help detect problems sooner in development and contribute to development productivity.

Use Java assertions for development with little performance impact in deployment.

Another helpful mechanism is the use of `static final Boolean` variables (set to false) to bracket out code that might be used for development-only. The compiler will not generate code if the variable is used as the condition in an if-then statement.

Use “static final boolean” variables to conditionally compile/elide development-only code.

7.12 "Warm" Key Code Paths

Applications running on JVMs that use JIT compiler optimizations will typically get faster after a while. A satisfactory number of iterations depends on the code details especially when nested loops and multiple levels of method calls are involved. The best thing is to structure the application to allow separate warming of key code paths, then time the warming after a certain number of iterations.

This works well for "server" style applications that start and then wait for an external event to trigger processing or a response. This is also appropriate for applications that need to achieve a certain throughput before the application is fully "up."

7.13 Share the Central Processing Unit Cores

One lesson learned the hard way by this investigator was the unpredictability of a fully loaded processor. If an application is running multiple compute-bound threads, it is easy to demand too much of the available cores. In some of the initial tests on parallelism, a four-way load split performed more poorly than a three-way load split even when not running at real-time priority. Most likely not only was the operating system starved, but also the garbage collector was starved.

There are two approaches that can be taken. One is to reserve one core for operating system use. The other is to intentionally add monitoring code to compute heavy tasks such that if computations have been going for too long the code will intentionally yield time to other threads.

Reserving an entire core for the operating system does not completely solve the problem because it leaves the garbage collector competing with compute threads on the remaining cores. Furthermore, the operating system most likely will not need all the horsepower of an entire core. This means that compute bound threads may have no choice to but implement the second option.

Of course, usually there is some form of I/O involved, and compute-bound tasks are usually broken up by I/O. But it is worth keeping this issue in mind because unless the application is instrumented to measure execution times and unless comparisons are performed under different load-splitting alternatives, the developer may not realize that performance is worse than it should be.

8 A Note on Real-Time Java and Safety-Critical Java

Most of the work in this study was done using Oracle Java 7 Standard Edition on x86_64 hardware. Different JVMs will have different performance. Real-time JVMs implement JSR 1, "Real-time Specification for Java" (RTSJ)[78]. The specification was designed to "enable the creation, verification, analysis, execution, and management of Java threads whose correctness conditions include timeliness constraints (also known as real-time threads)."

There are approximately 70 objects or interface additions within the `javax.realtime` package. They provide support for allocating object in memory scopes that are not subject to garbage collector activity. They also provide mechanisms for creating and running threads that run at real-time priorities. This makes it possible to write code that will not be interrupted by the garbage collector. Furthermore, while timer and scheduler implementation in standard Java provide "approximate" timing implementations of real-time Java, real-time operating systems provide much more precise timing and scheduling.

There are also vendors that provide JVM environments that allow the final Java code to be streamlined for burning into ROM devices. For those JVMs, there is a focus on tools that can generate executable images that have the application code and the minimum amount of Java library objects for the application to work.

Real-time and ROMable JVMs can have comparable speeds to the Hotspot-based JVM used in this study. But some can also be much slower. This study did not attempt to characterize the performance of such JVM's.

One reason for reduced speed is that to reduce jitter they might reduce or eliminate JIT compilation. Having a JIT "optimize" running code would change its execution time. Another reason is that they might also increase the overhead work associated with memory allocation to more predictably detect unused structures and trigger memory reclamation. Deterministic garbage collection results in more predictable code with repeatable behavior.

Safety-critical Java[79] (SCJ) is a "J2ME™ capability, based on the RTSJ (JSR-1), containing minimal features necessary for safety-critical systems capable of certification, e.g., DO-178B." Services include start-up, concurrency, scheduling, synchronization, memory management, timer management, and interrupt processing. It has a region-based memory allocation scheme instead of one general heap. Regions form a stack of scopes that are created programmatically and can be reclaimed when they fall out of scope. There is no garbage collector operating on these regions, hence reduced jitter.

Some of the early SCJ implementations convert Java to C. This means the code cannot benefit from JIT optimizations and will also likely run slower than Oracle Java 7, Standard Edition.

The approach taken by vendors of RTSJ and SCJ JVMs will vary. Though this study did not attempt to characterize the performance of such systems, the benchmarks and approaches used in this investigation should help characterize such platforms.

---

[78] See "JSR 1: Real-time Specification for Java," July 2006, <http://jcp.org/en/jsr/detail?id=1>

[79] See "JSR 302: Safety Critical Java™ Technology," January 2011, <http://jcp.org/en/jsr/detail?id=302>

This page intentionally left blank.

9 Conclusions

This study sought to show that Java 7 Standard Edition and C could be mixed in a hybrid high performance computing application. The motivation was to offload the less compute intensive work to Java where development is usually faster and safer and where there are intricate infrastructure and grid-computing frameworks.

A hybrid application would likely use Java for communication, logging, resource scheduling, etc., and leave the heavy computation to C-based code. Nonetheless, the study explored Java compute performance to get comfortable with Java performance overall. The study compared several libraries and approaches for invoking native C code from Java. It collected metrics showing how the choice of Java-side or native-side data limited callout rates. It investigated third-party libraries to facilitate native memory allocation and management. It also measured the overhead of Java background threads. It used FFT code optimized for execution in GPUs to compare C and Java utilization of GPUs. The study also looked beyond the boundary of a single node and investigated the characteristics and performance of 5 Java compute-grid frameworks. Finally, in the appendices, data was presented on the performance of Scala, a modern, concise language that runs on the JVM, and Jython, a version of Python suitable for scripting on the JVM.

The investigation on Java-based computation showed that for some computations Oracle Java 7 Standard Edition was as fast as GCC compiled C code. All of the benchmark codes began as existing C code that was translated to Java. Bit-twiddling and matrix multiplication were faster in Java. Allocating, manipulating, and accessing the red-black tree data structures was also faster when the size of the data structures allocated and inserted into the tree was less than a few hundred bytes. It seems that the speed of dynamic memory allocation was better for small allocation sizes than for larger ones. Performing FFTs, using the non-optimized, radix-two kernel, was faster in C. It is hypothesized that the Java compiler and JIT runtime optimizations do a good job with small loops and simple array indexes. FFT “butterflies” have multiple array indexes with and intricate sequence of array indexes.

The investigation on concurrent computation showed that the fastest libraries and APIs for parallelized “for-loop” computation are third-party libraries. The Parallel Java library provided the best performance followed by the Javolution library. They fell 3 to 5 percent short of C’s performance on the same tests. The investigation also revealed that there can be adverse interactions between Java thread scheduling and Linux default scheduling that can be detected by measuring application execution time. Should this occur the problem can be mitigated by running the Java application using Linux round-robin real-time scheduling.

The investigation on Java overhead showed that in the absence of garbage collector work, there was no measurable JVM overhead for computation and in-memory structure manipulation. On the other hand, a poorly tuned, memory allocation/deallocation-intensive test with approximately seven garbage collector activations per second caused a 15 percent decrease in compute throughput. In general a well-designed and well-tuned application would have much fewer garbage collector activations and almost imperceptible garbage collector activity.

A mixed Java-C compute application may want to put the graphics processing units, GPUs, under control of Java. The investigation compared the ability of Java and C to invoke a GPU. Data was allocated natively using BridJ. JavaCL was used to access the underlying OpenCL service, which provided access to GPU functions. Java took about 2 microseconds longer than C’s 10 microseconds to set up and queue an OpenCL kernel computation. But that did not result in a 20

percent decrease in GPU throughput because kernel setup and queuing proceeds in parallel with GPU execution. There was no more than a 2 percent difference in the effective GPU throughput for most data sizes between 64 and 524,288.

Regarding invoking native libraries, the investigations showed that when native data is used SWIG and BridJ can perform comparably to hand coded JNI (which is the low level Java callout mechanism). BridJ, which is supported on many platforms, makes it very easy to allocate and manage the native data. This includes garbage collection of the native data when it is no longer referenced by the Java code. SWIG is powerful but has more complexity and requires more learning. However, if the target platform is not one of the supported ones, SWIG may be then next best choice.

Five Java compute-grid frameworks, [33]PJ, JPPF, GridGain, Storm, and, Akka were tested on a 10-gigabyte Ethernet cluster. Collectively the frameworks offer features such as auto-discovery, auto-failover, inter-node data synchronization, automatic data serialization, multi-node work queuing, active load monitoring and adaptive load balancing that can save significant developer time. They were compared with each other and to MPI, which is the workhorse for performance computing.

With greater flexibility and features, the more sophisticated frameworks were not expected to perform as well as MPI. Indeed only MPI and low-level Java socket I/O could handle small data sizes at rates of 10,000 synchronous round-trip exchanges per second. On the other hand Akka and Storm achieved 10,000 asynchronous round-trip exchanges per second. For many applications with lower requirmenets, other Java frameworks, such as GridGain and JPPF, could be used as well. Even a C framework offering enhanced flexibility and features would be expected to have poorer performance than MPI.

Two other investigations were conducted and are presented in the appendices. One looked at Scala because some have thought it might be a successor to Java. It is potentially attractive for embedded high performance computing because it is more concise but only if it retains Java's throughput. On matrix-multiply and FFTs the investigation found that for-loops need to be implemented as while-loops because of increased overhead in Scala's for-based looping statements.

The second language investigation presented in the appendices is the use of JPython for scripting. JPython was found to have performance more comparable to CPython than to Java or C. It was faster to call out to native libraries than CPython though slower at numerical computation. PyPy, a faster compiled implementation of Python was found to be slower than Java by a factor of three for matrix multiply.

Java, and the JVM, can be part of the high performance tool chest for applications that run long enough to allow Java 7 Hotspot warm-up optimizations to "kick in." Dynamic memory allocation should be minimized by preallocating working storage. Other practices for good performance were also discussed in the body of this study.

A crafted mix of Java and C can exploit the productivity and code safety of Java as well as the speed of C to build solutions that perform well and make good use of increasingly scarce development dollars and available Java-talent. It is hoped that the developer will find this report and its findings helpful in making quantitative trade-offs leading to good performing, hybrid applications.

The benchmarks developed here have been made available as open-source at http://jcompbmarks.sourceforge.net. Systematic analysis of performance requires easy-to-use facilities for timing code sections and accumulating the simple statistics. The timing utilities developed here for C, Java, and Python feature a similar API and have proved invaluable in these studies. They have been made available at http://jtimeandprofile.sourceforge.net. Finally, a tool to facilitate invoking Java from C has been released at http://cshimtojava.sourceforge.net.

This page intentionally left blank.

Appendix A Scala

Scala is a relatively new language first released in 2003. It can run on a Java Virtual Machine (JVM) though it features a more powerful and more concise language syntax. It retains an object design but brings many powerful lessons from the world of functional programming to coexist with the object design. It has strict type checking but does not require the many type declarations that make Java feel "bloated." Since it runs on the JVM and is concise, the question arose as to whether it might retain sufficient performance to be considered as an even more attractive addition to the high performance toolbox than Java might be.

Scala can run on the JVM and easily use Java's libraries. Much work has been put into making it easy to mix Java and Scala collection types and iterators within an application. Scala has power, flexibility, and the ability to define type-aware operators that are somewhat like operator overloading. Scala's power comes at the cost of a bit more language intricacy that requires more learning than one might want.

However, Scala is also quite concise without sacrificing strong type-checking. The investigator's experience with Scala suggests that the same Java function can be coded with about three-quarters of the number of lines of code.

The first experiment with Scala was an implementation of the red-black tree insertion test described in Section 2.8. In Scala, everything can be treated as an object including numbers. The red-black tree algorithm used in this report was based on generic objects making this easy for translation to Scala. But the language also has a provision that allows specialized/faster code to be generated if certain classes are instantiated for primitive types instead of objects. This creates multiple versions of the generic class (such as the red-black tree) but should provide better performance. Thus, the red-black tree tests for Scala were run both ways, with specialization and without.

Figure A-1 compares the Java insert and get tests for four-byte content with a Scala version and a Scala version with the @specialized annotation on the keys of the red-black tree and red-black node. The unspecialized Scala implementation was a little slower than the Java version, though still faster than the C version documented earlier in this report. The specialized version did a little better than the Java version.

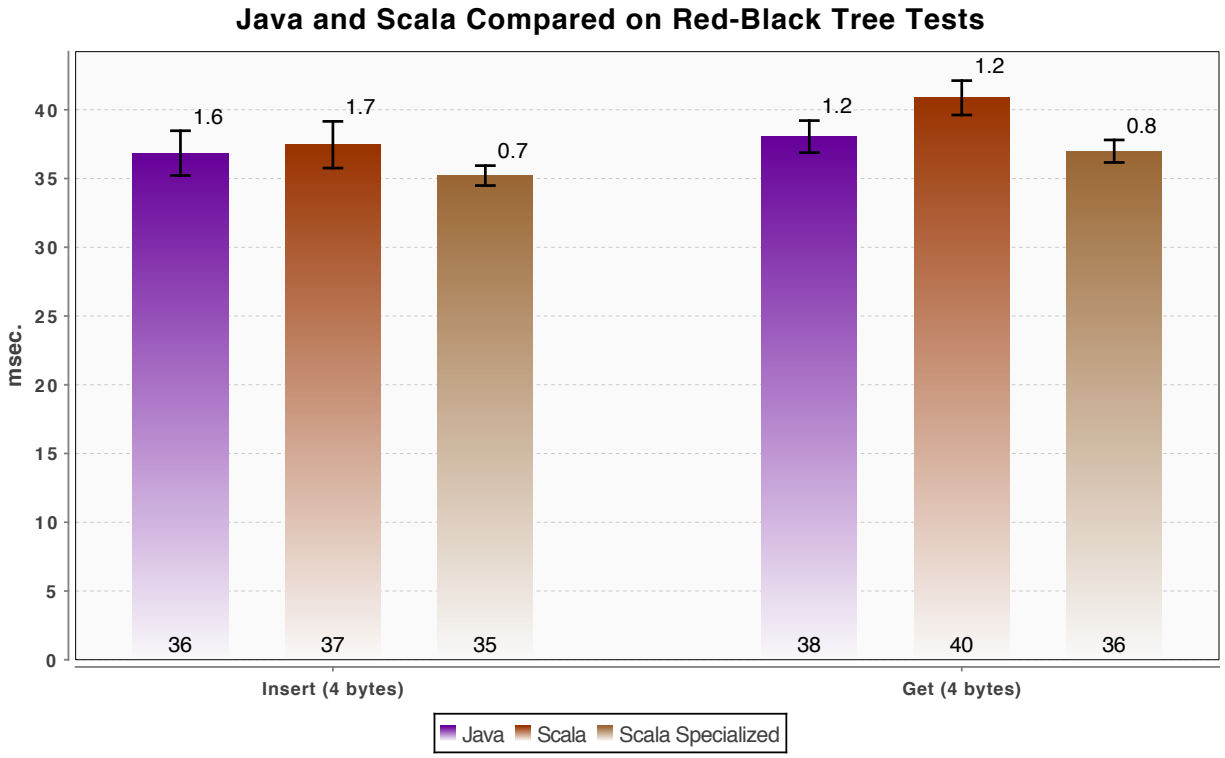

Figure A-1. Java and Scala Red-Black Insert and Get Performance

The matrix multiply test requires three nested loops. This makes a good test to understand the performance of Scala loops. Scala's for-comprehensions are actually "syntactic sugar" for higher-order methods. The for-comprehension is a general construct that can be used for iteration over a variety of types and can include filtering of iterated types as well. This results in a very compact syntax for a sequence of more complex operations, but it has the downside effect that a simple sequential integer-based iteration is not translated as a simple byte-code loop.

There has been much discussion of this issue.[80,81] The compiler, as of Scala 2.10.2, still generates code with the slower for-comprehension for integer-based looping. However, there is a Scala compiler addition targeted for Scala 2.11 that will perform the desired optimization.[82]

Figure A-2 compares the Java matrix multiply test described in the body of this report to its Scala counterpart. For Scala, a test was run using a for-comprehension and another by converting the loop into a while-loop. One can see that the for-comprehension had poor performance for $N < 18$. On the other hand, the Scala while-loop-based implementation performed nearly the same as the Java implementation.

---

[80] See two of many discussions at <http://stackoverflow.com/questions/6146182/how-to-optimize-for-comprehensions-and-loops-in-scala> and <https://groups.google.com/forum/#!topic/scala-user/hq20TXLvRJg>.

[81] See the ticket, <https://issues.scala-lang.org/browse/SI-1338>.

[82] "ScalaCLPlugin," < http://code.google.com/p/scalacl/wiki/ScalaCLPlugin >

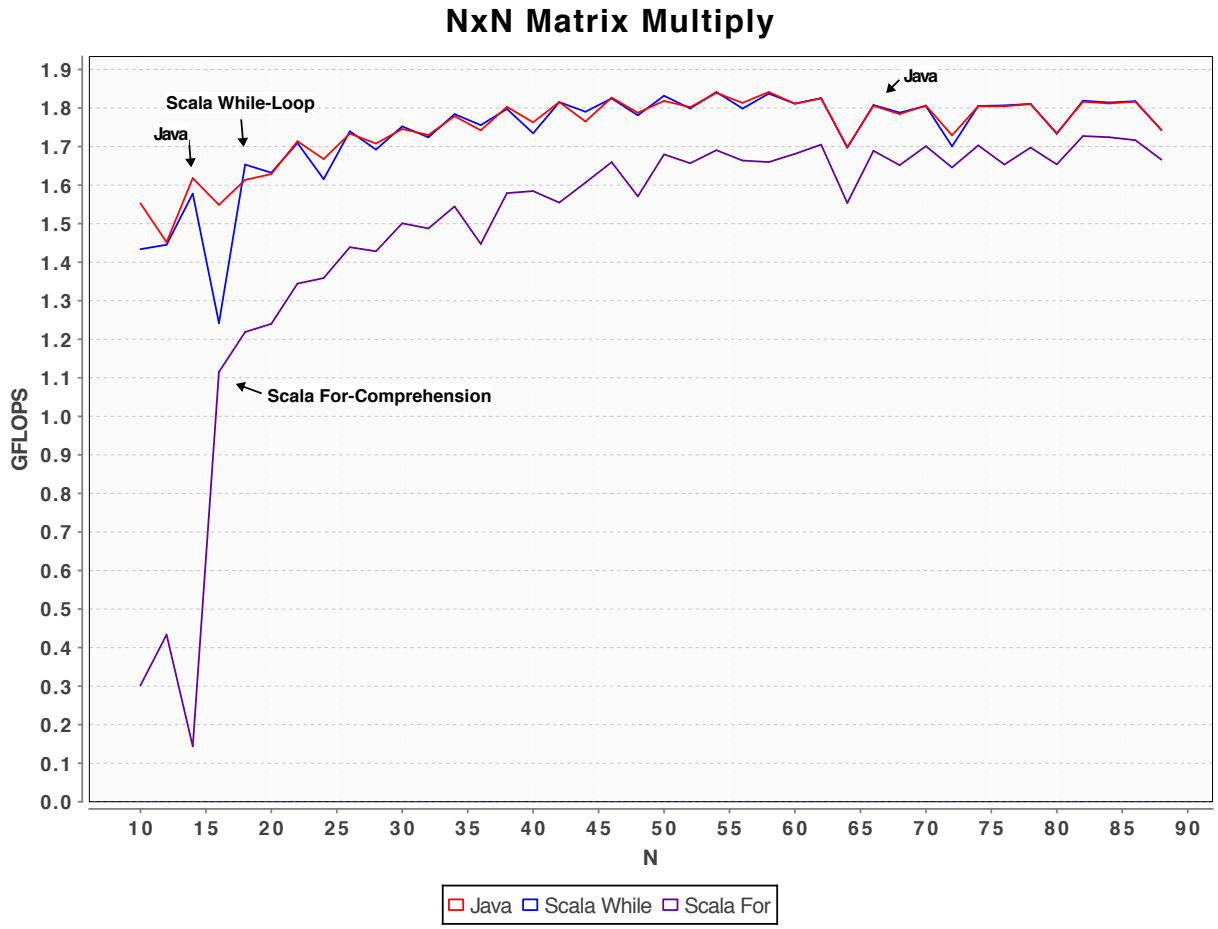

Figure A-2. Matrix Multiplication Comparing Java and Scala

The final test performed was to compare Scala's speed on the FFTs to Java's. One version of the FFT was implemented using the for-comprehension and another with a while loop. Figure A-3 shows the results. The Scala for-comprehension iteration executes quite slowly, between 13 to 34 percent of the Java code. The Scala while-loop implementation executes at the same speed as its Java counterpart other than at one anomalous point.

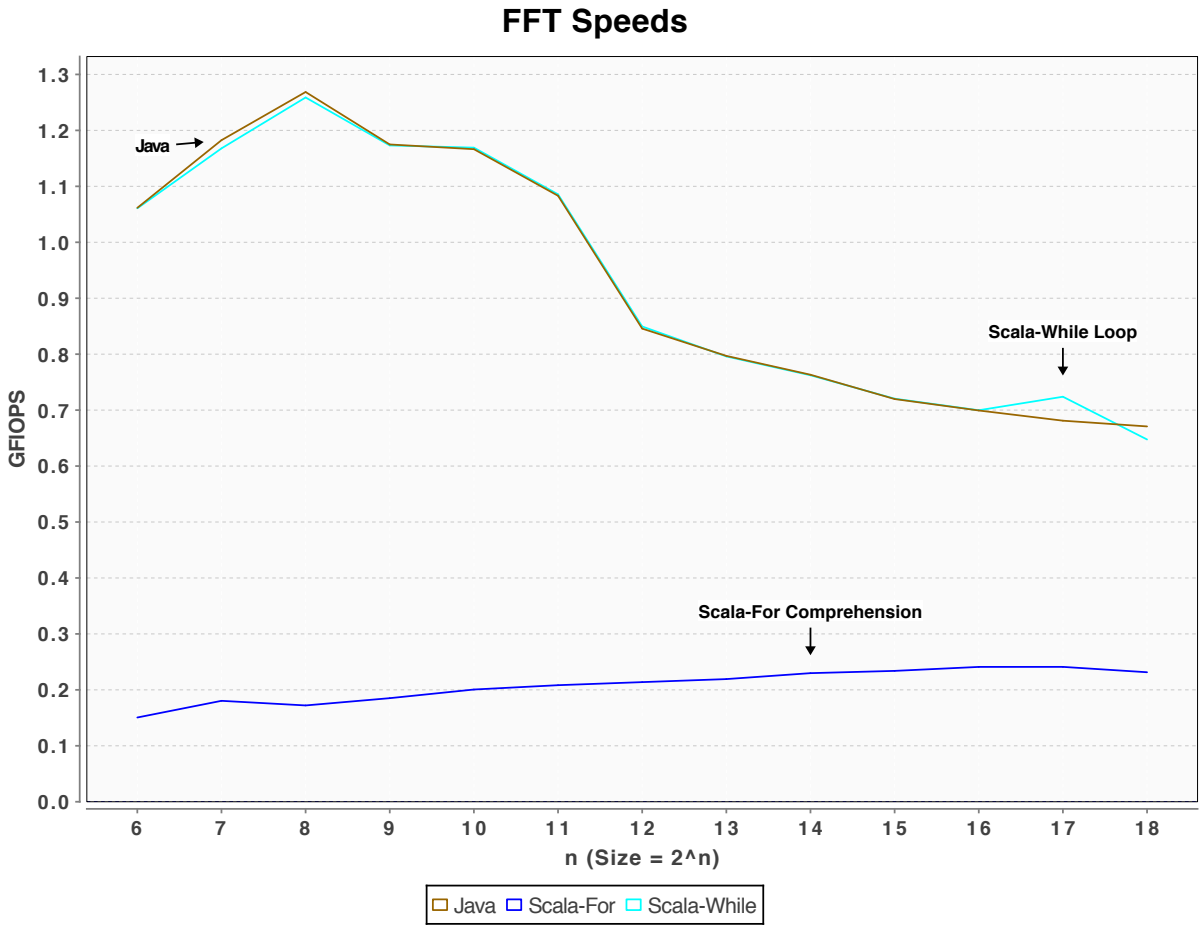

Figure A-3. Comparison of Java and Scala FFT

Based on these tests this investigator's opinion is that Scala can be used in the performance environment provided that the ScalaCL plugin is used to expedite the speed of simple integer-based for-comprehension loops. Much performance code relies on existing libraries and algorithms that have their roots in old Fortran code with for-loops. Converting these to while-loops by hand would be too burdensome and error prone. Scala's conciseness compared to Java is a great win for the JVM.

Appendix B JPython

Jython is an implementation of Python that runs on the Java Virtual Machine (JVM). Python has existed since the 1980s. It is popular in the scientific computing and performance computing communities. It features a more concise syntax than Java due in part to a lack of type checking. It has garbage collection. It has object support though the developer has to manage more of the mechanics of subclassing. It makes it easy to call out to external native C libraries and create language "extensions" based on native libraries. This makes Python an easy-to-use as a scripting language for binding and sequencing native functions.

CPython is an interpreted Python that is relatively slow when compared to compiled code. PyPy is a fast implementation of Python that includes a just-in-time (JIT) compiler that also exhibits speedup behavior after a few iterations of code.

The question arose as to how some of the versions of Python compared to each other and to Java on the algorithms used in the body of this report. The goal was for quantitative comparisons that would be of potential benefit to the designer making implementation trade-offs. The following are some "quick" comparative performance investigations.

The first comparison is performance on the matrix-multiply test. Python developers typically use the NumPy package. NumPy does much of the work in native libraries and would not be a reasonable indication of relative language performance. Instead, the test implemented the same matrix multiply code used in the body of this report for C and Java in pure Python. Python's array package was used for matrix storage on all the Python tests. All the tests first pre-exercise the code before collecting timings.

The results are summarized in Table B-1, that compares throughput ratios. The raw throughputs, in giga-flops, are shown in Figure B-1 and Figure B-2. Jython exhibited the slowest throughput on this test with about 70 percent of CPython's throughput. PyPy was significantly faster than CPython but slower than Java by a factor of three. Recall from the body of this report that warmed Java was consistently a little faster than C on Oracle Java 7 Standard Edition.

Table B-1. Relative Matrix Multiply Performance for Java, Jython, CPython, and PyPy

Throughput Ratio	Value
CPython/Jython	1.4
PyPy/CPython	120
Java/PyPy	3

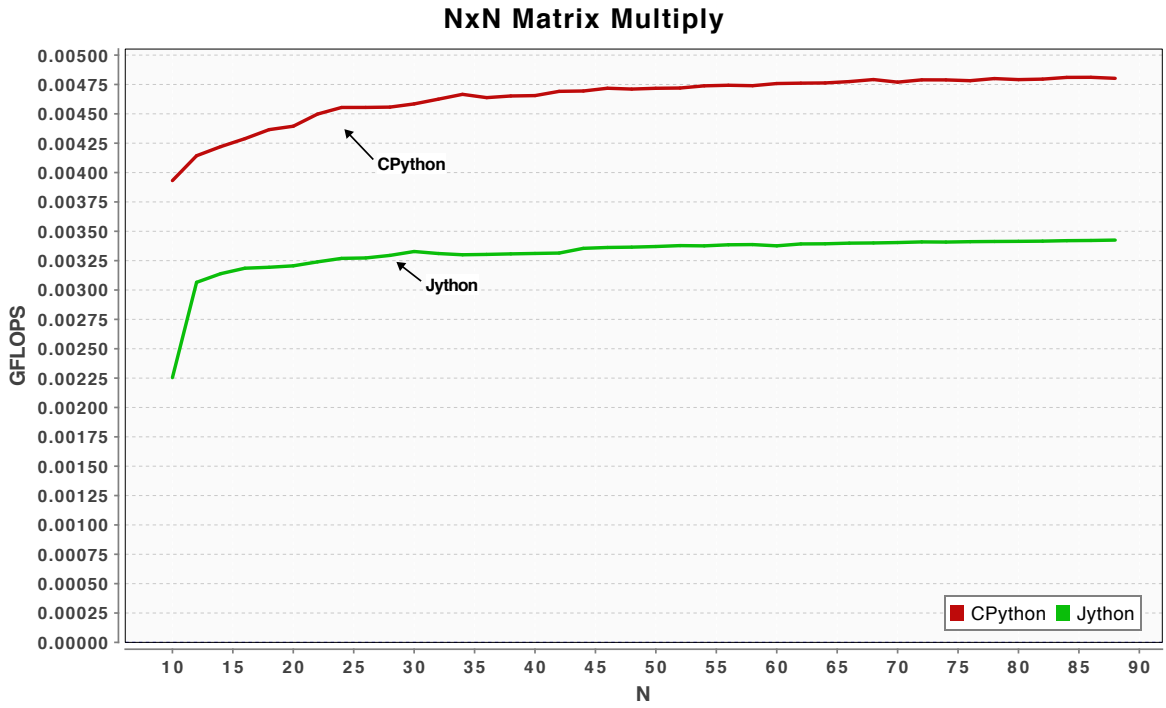

Figure B-1. CPython and Jython Matrix Multiply (without NumPy)

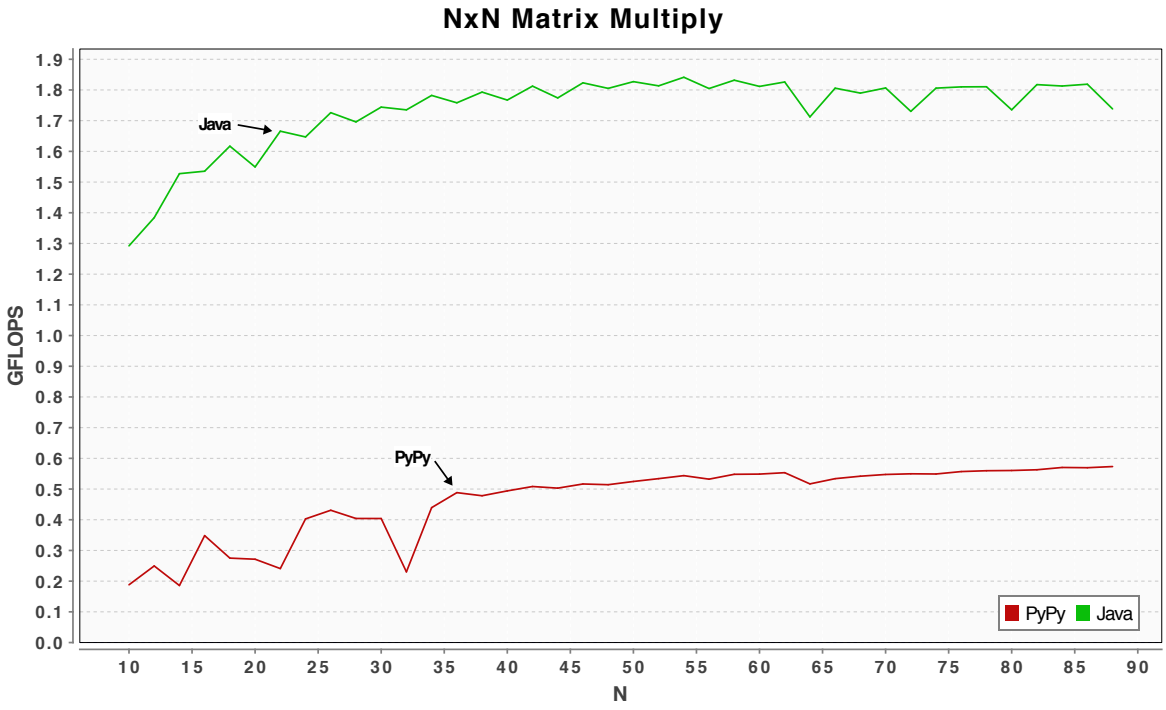

Figure B-2. PyPy and Java Matrix Multiply (without NumPy)

The matrix-multiply test focuses on computation. The second test was a repeat of the red-black tree test described in the body of the report but implemented in Python. This test is a non-

compute test of memory allocation and in-memory structure building. Again, Java, Jython, CPython, and PyPy implementations were compared for how quickly they could finish a fixed number of inserts and a fixed number of gets to/from the red-black tree.

The results are shown in Figure B-3 and Figure B-4. Jython and CPython performed comparably. One was slightly faster on inserts, the other slightly faster on retrieval. PyPy was faster than CPython by a factor of about fourteen. Java was the fastest. It was faster than PyPy by a factor of three on retrieval and by a factor of six on insertion of simple integer content.

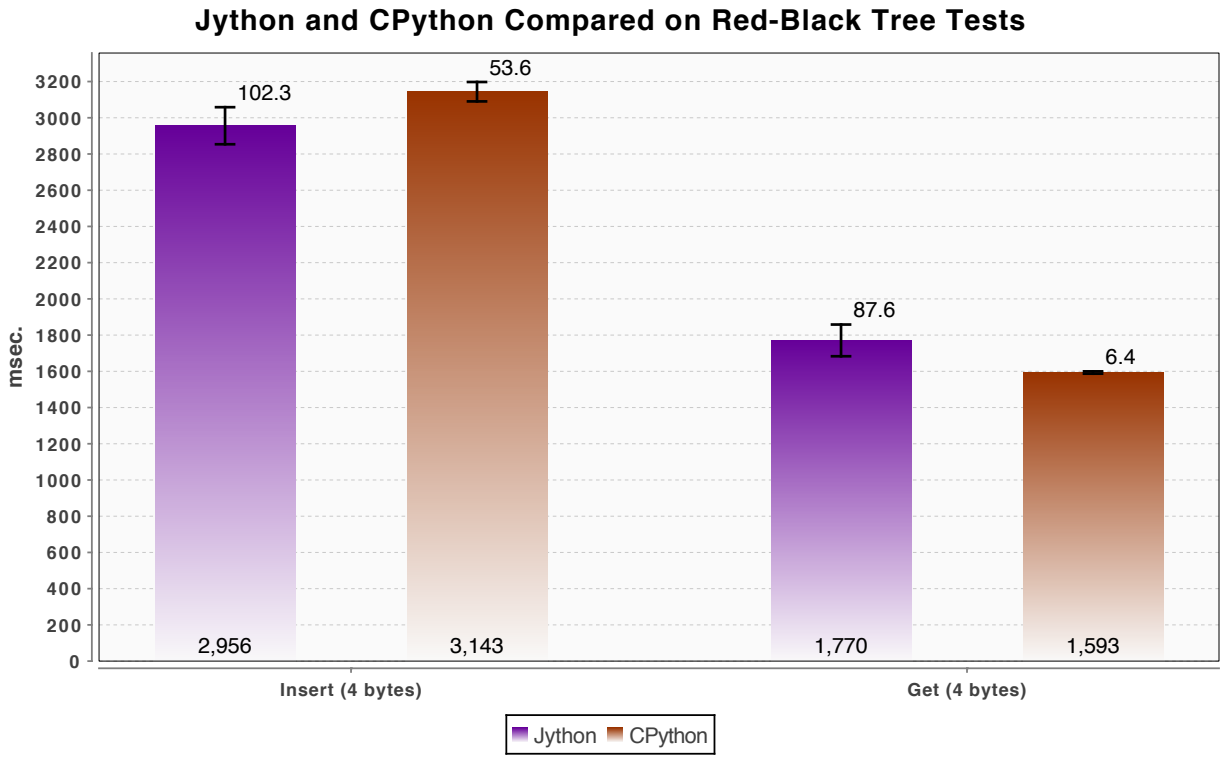

Figure B-3. CPython and Jython Red-Black Tree Test

A final investigation compared the time to invoke native libraries from Jython and CPython. Python has an extension application programming interface (API) that is conceptually similar to Java Native Interface (JNI). An extension was hand-coded to call into the native library using this API for CPython. Jython, on the other hand, can already import classes and make method calls to Java libraries. Reusing the JNI glue developed for the studies performed previously was easy.

One note before presenting the findings is that in the previous study with Java invoking C there were several options for how to pass Java array data or whether to create native data under the control of Java code. Since all those options are still available, this test did not repeat that. Instead this test focused on the call times to methods with simple integer arguments since those can be extrapolated to predict performance with the various array-passing schemes.

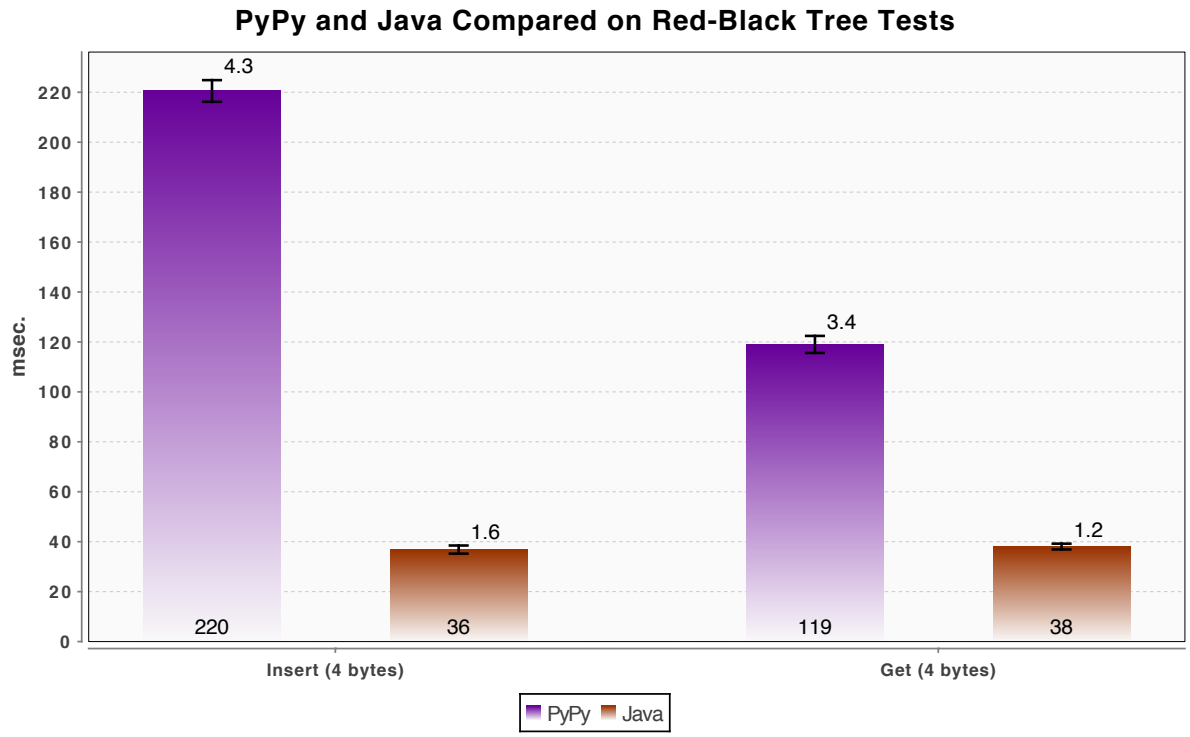

Figure B-4. PyPy and Java Red-Black Tree Test

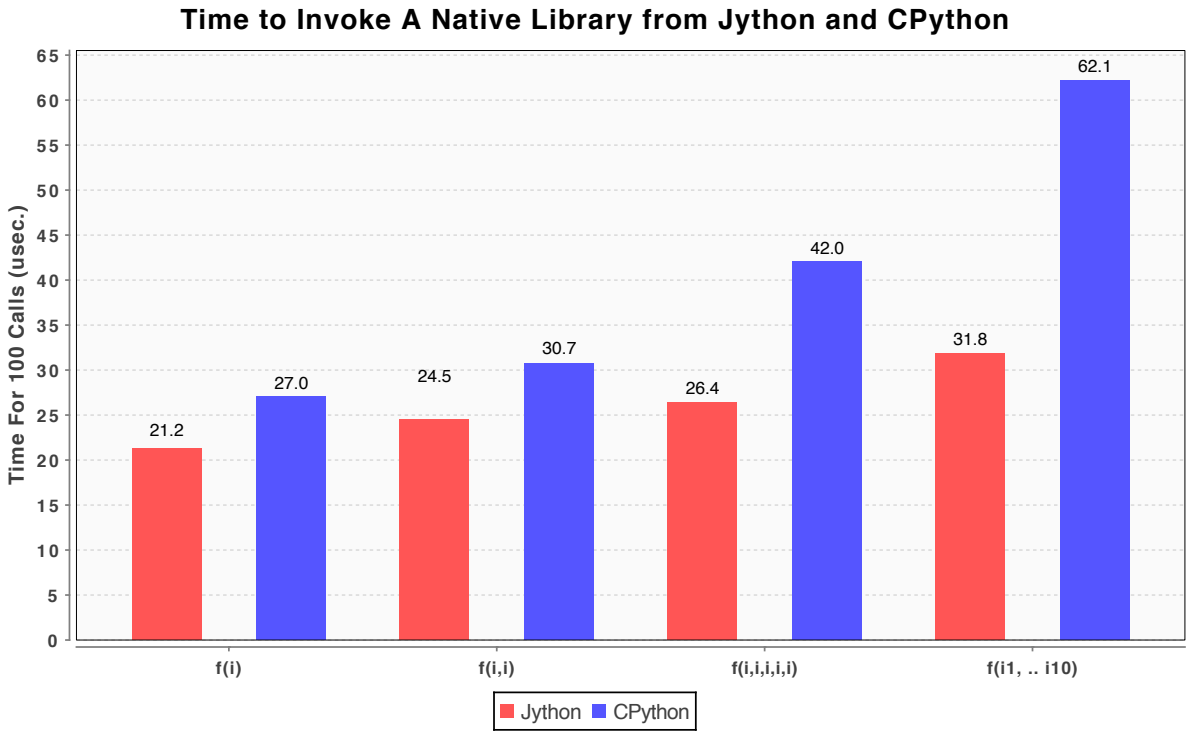

Figure B-5. CPython and Jython Times to Invoke Native Library Methods

The results of the test are shown in Figure B-6. For a few arguments it could be said that the times are in the same "ballpark." However, there is an increasingly disproportionate difference as

the number of arguments increases. The reason for this is probably because in the CPython extension API all of the arguments are objects that must be accessed as an interpreted call-back of sorts by the native glue. On the other hand, with JNI each integer (or address argument) is passed directly, requiring no further callback.

The difference in time is insignificant for many applications if only a few thousand calls are made per second. For some applications the difference may be more important.

It may be helpful to put these callout times into perspective by comparing them to the time for CPython to call CPython and Jython to call Jython. Here Jython is about twice as fast.

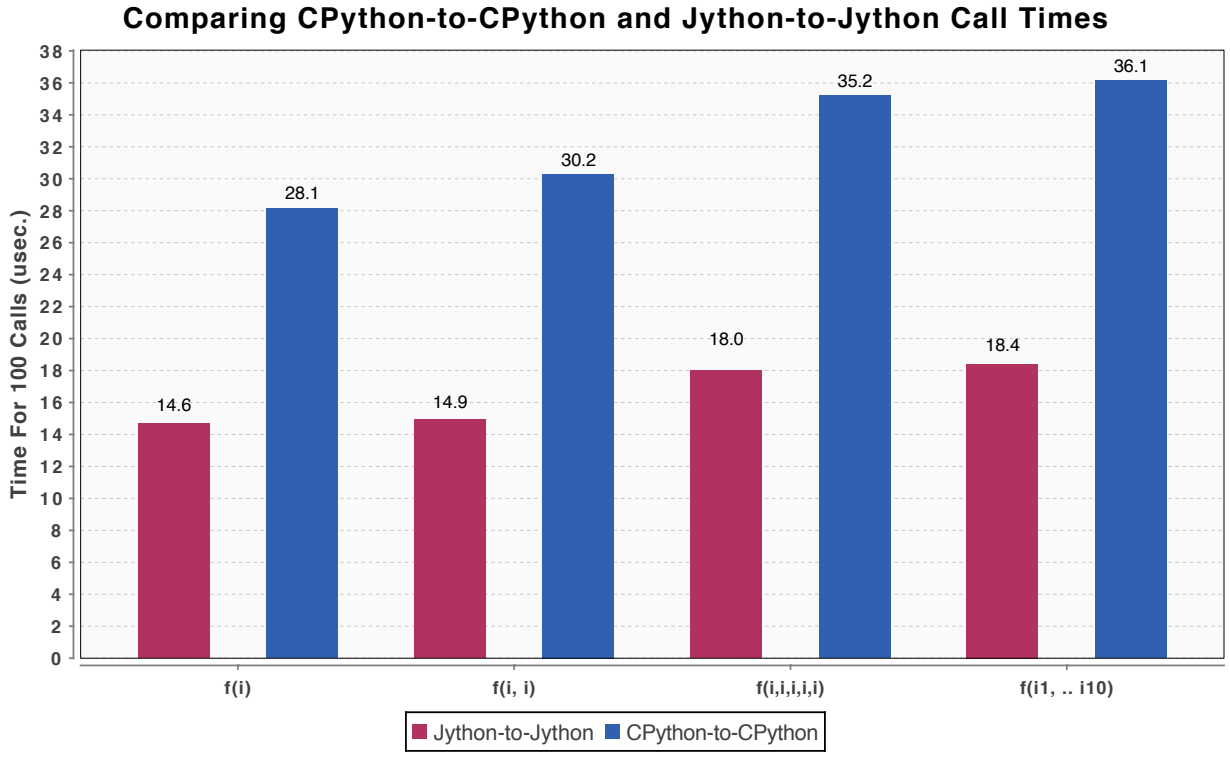

Figure B-6. CPython and Jython Call Times

This page intentionally left blank.

Appendix C Libraries for Numerical Computing In Java

This is a list of Java libraries and benchmarks for numerical computing that were discovered in the course of this study. They are listed here as a starting point for those who are interested. The performance of some of these are compared at http://ojalgo.org/performance_ejml.html and http://java.dzone.com/announcements/introduction-efficient-java. Note also that there is a tool for evaluating Java linear algebra libraries, Java Matrix Benchmark (JMatBench) at http://code.google.com/p/java-matrix-benchmark/.

Table C-1. Java Libraries for Numerical Computing (1 of 2)

Name	Summary & URL
Apache Commons Math	• Math and statistics utilities including linear regression, least-squares fitting, root-finding, solving linear equations, solving ordinary differential Equations, etc. • http://commons.apache.org/math/apidocs/index.html
COLT	• Libraries for high performance scientific and technical computing developed at CRN. Linear Algebra, Multi-dimensional arrays, Statistics, Histogramming, Monte Carlo Simulation, etc. • http://acs.lbl.gov/software/colt/
Efficient Java Matrix Library (EJML)	• Linear algebra library for manipulating dense matrices • http://code.google.com/p/efficient-java-matrix-library/
JAMA	• Linear algebra • http://math.nist.gov/javanumerics/jama/
JAMPACK	• Linear algebra • ftp://math.nist.gov/pub/Jampack/Jampack/AboutJampack.html
jblas	• Linear algebra library for Java; based on BLAS and LAPACK; uses native libraries • http://jblas.org/

Table C-2. Java Libraries for Numerical Computing (2 of 2)

Name	Summary & URL
JLAPACK	• BLAS and LAPACK machine translated from Fortran source • http://icl.cs.utk.edu/f2j/ - Also home of the f2j translator which translates Fortran to Java
JScience	• Units of measurement, coordinate representation/transformation, operations and representation of mathematical fields, sets, rings, vector spaces, linear algebra, symbolic manipulation and equation solving, physics constants and models, monetary conversion, etc. • http://jscience.org/
Linear algebra library for Java (la4j)	• Sparse and dense matrix library • http://code.google.com/p/la4j/
Matrix-Toolkits-Java (MTJ)	• Matrix data structures, linear solvers, least squares methods, eigenvalue and singular value decompositions • Based on BLAS and LAPACK for its dense and structured sparse computations, and on the Templates project for sparse operations • http://code.google.com/p/matrix-toolkits-java/
ojAlgo	• Linear algebra, linear Programming, quadratic programming, mixed integer programming, modern portfolio theory, random number distributions, • http://ojalgo.org/
Universal Java Matrix Package (UJMP)	• Sparse and dense matrices, as well as linear algebra calculations such as matrix decomposition, inverse, multiply, mean, correlation, standard deviation, etc. • http://sourceforge.net/projects/ujmp/

Appendix D CPU Loading for Tested Frameworks

Data on central processing unit (CPU) loading was collected for the synchronous and asynchronous framework tests presented in Section 6.4. Tables D-1 and D-2 show the data. The percentage shown is the total percentage of CPU loading between the two nodes. The two numbers in parenthesis are the percentages at each node respectively. The data was collected in the hope that it would provide some insight into the CPU time "cost" of running the framework.

It is difficult to generalize conclusions from the findings. MPI displays pretty similar loading for different data sizes in both synchronous and asynchronous tests. The similarity between the synchronous and asynchronous tests can be explained by the only difference being that the asynchronous version of the test had a round-robin buffer system that allowed up to 4 outstanding buffers to be in transit. The majority of the test is spent transmitting or receiving the next (albeit delayed) buffer. On the other hand the Parallel Java test used the same round-robin buffering scheme and it does not exhibit consistent behavior.

Storm exhibits very low CPU loading for synchronous data exchange compared even to MPI. Its load increased a lot with the asynchronous tests.

Other than JPPF, the Java frameworks used much less CPU than MPI for the synchronous tests but higher CPU than MPI for the asynchronous tests.

The findings have been presented here for future reference.

Table D-1. Percentage Processor Load For Synchronous Tests

Framework	Data Size		
	88	8072	648072
MPI	51% (26/25)	55% (31/24)	51% (26/25)
PJ	24% (12/12)	29% (13/16)	17% (7/10)
JPPF	39% (26/13)	77% (56/21)	35% (12/23)
GridGain	33% (21/12)	24% (16/8)	21% (11/10)
Storm	18% (13/4)	19% (13/6)	22% (10/12)
Akka	29% (13/16)	27% (13/14)	15% (10/12)

* 100 percent = 1 core

Table D-2. Percentage Processor Load For Asynchronous Tests

Framework	Data Size		
	88	8072	648072
MPI	51% (25/26)	49% (25/24)	47% (24/23)
PJ	78% (37/41)	70% (32/38)	31% (13/18)
JPPF	66% (51/15)	70% (51/19)	73% (24/49)
GridGain	77% (38/39)	118% (48/46)	72% (34/38)
Storm	89% (46/43)	134% (62/72)	102% (83/22)
Akka	94% (48/46)	94% (48/46)	74% (34/38)

* 100 percent = 1 core

Appendix E Abbreviations and Acronyms

AOT	Ahead of Time
API	Application Programming Interface
CPU	Central Processing Unit
FFT	Fast Fourier Transform
FFTW	Fast Fourier Transform West
GPL	General Public License
GPU	Graphics Processing Unit
HPEC	High Performance Embedded Computing
I/O	Input/Output
IP	Internet Protocol
JIT	Just in Time
JNA	Java Native Access
JNI	Java Native Interface
JPPF	Java Parallel Processing Framework
JSR	Java Specification Request
JVM	Java Virtual Machine
LFSR	Linear Feedback Shift Register
MPI	Message Passing Interface
ms	Millisecond
ns	Nanosecond
OS	Operating System
PJ	Parallel Java
RDMA	Remote Direct Memory Access

ROM	Read-only Memory
RTSJ	Real-time Specification for Java
SCJ	Safety-critical Java
SSH	Secure Shell
SWIG	Simplified Wrapper and Interface Generator
TCP	Transmission Control Protocol
UDP	User Data Protocol
µs	Microsecond