5-2016

Parallel Computing in Java

Muqeet Mohammed Ali
St. Cloud State University, mdmuqeetali@gmail.com

Parallel Computing in Java

by

Muqeet Mohammed Ali

Starred Paper

Submitted to the Graduate Faculty

of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree

Master of Science

in Computer Science

May, 2016

Abstract

The Java programming language and environment is inspiring new research activities in many areas of computing, of which parallel computing is one of the major interests. Parallel techniques are themselves finding new uses in cluster computing systems. Although there are excellent software tools for scheduling, monitoring and message-based programming on parallel clusters, these systems are not yet well integrated and do not provide very high-level parallel programming support.

This research presents a number of issues which are considered to be key to the suitability of Java for HPC (High Performance Computing) applications and then explore the support for concurrency in the current Java 1.8 specification. We further present various relatively recent parallel Java models which support HPC for both shared and distributed memory programming paradigms. Finally, we attempt to evaluate the performance of discussed Java HPC models by comparing the same with the relative traditional native C implementations, where appropriate. The analysis of the results suggest that Java can achieve near similar performance to natively compiled languages, both for sequential and parallel applications, thus making it a viable alternative for HPC programming.

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank Dr. Donald O. Hamnes and my committee members Dr. Dennis C. Guster and Dr. Jie Hu Meichsner for their time and cooperation, which led to the successful completion of my starred paper. I would also like to extend my thanks to Dr. Ramnath Sarnath for his support throughout my masters program, and the Department of Computer Science, SCSU, for their help in providing the necessary resources.

I am also grateful to my friends (Wajahat Ali, Shoeb Uddin Mohammed, Mukhtar Mohammed, Younus Raza, Zubair Mohammed, Yousuf Mohammed, Hatim Mohammed, Abubucker Siddiqui) for their support and motivation during my initial days of the masters program.

I wish to thank my siblings Mukarram Ali Mohammed and Ruqia Maihveen for the continuous encouragement, support, love, and care they provided throughout my life. Am also thankful to my fiancé Sumiah Fatima, without whose understanding and motivation this research wouldn’t have been completed.

Lastly, I would like to express my deepest gratitude to my parents, Ahmed Ali Mohammed and Bilquees Begum. Their love, care and motivation at every step of my life defined the person I am today. They inspire me every day and were the strength that brought my Master’s journey to its culmination and to them I dedicate this paper.

Table of Contents

List of Tables

List of Figures

List of Graphs

List of Equation

Chapter I

INTRODUCTION

Parallel computing is a form of computation in which many calculations are carried out simultaneously. It operates on the principle that large problems can often be divided into smaller ones, which are then solved concurrently [1]. In recent years, parallel computing has been widely adopted in domains that need massive computational power, such as graphics, animation, data mining and informatics. Traditionally, software has been written for serial computation that involves [1]:

- Running on a single computer having a single Central Processing Unit (CPU);
- Splitting a problem into a discrete series of instructions.
- Sequentially executing instructions.
- Executing only one instruction at any moment in time.

There are limits to serial computing. Significant constraints for building ever faster serial computers include both physical and practical reasons [1]:

- Size limitations: New processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be.
- Transmission speeds: The speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light (30 cm/nanosecond) and the transmission limit of copper wire (9 cm/nanosecond). Increasing speeds require increasing nearness of processing elements.

- Economic limitations: It is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same performance is less expensive.

On the other hand, parallel computing uses multiple computer resources simultaneously to solve complex computational problems that involve [2]:

- Utilizing multiple CPUs.
- Splitting a problem into discrete parts that can be solved concurrently.
- Further breaking down of each part to a series of instructions.
- Each part executing instructions simultaneously on different CPUs.

Why use parallel computing? The main reasons include:

- Speed-up computations: If a problem is split into parts and each part is computed simultaneously on different processors then significant speed ups can be achieved.
- Solve larger problems: Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.
- Provide concurrency: A single compute resource can only do one thing at a time. Multiple computing resources can be doing many things simultaneously.
- Use of non-local resources: Using computing resources on a wide area network when local compute resources are scarce.

Historically, parallel computing has been considered to be “the high end of computing,” and has been used to model difficult scientific and engineering problems found in the real world. Some fields include [2]:

- Atmosphere, Earth, Environment
- Physics—applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
- Bioscience, Biotechnology, Genetics
- Chemistry, Molecular Sciences
- Geology, Seismology
- Mechanical Engineering—from prosthetics to spacecraft
- Electrical Engineering, Circuit Design, Microelectronics
- Computer Science, Mathematics

Motivation of the Study

Most of the parallel applications are generally written in C or FORTRAN but as Java has become one of the most popular languages in the IT industry with its “write once, run anywhere” feature and powerful support from open source organizations, a Java based parallel computing framework has become essential [3]. The out of the box networking and multithreading support, continuous Java Virtual Machine (JVM) performance improvements and support for programming multi-core clusters also contribute for the increasing interest in Java for High Performance Computing (HPC).

We begin this paper by discussing the potential of Java as a HPC language, present various parallel programming issues that hinder its adoption for HPC and explore the support for concurrency in the current Java 1.8 specification. We then proceed by presenting various relatively recent parallel Java models which support HPC for both shared and distributed memory programming paradigms. Finally, we attempt to evaluate the performance of discussed

Java HPC models by comparing the same with the relative traditional native C implementations, where appropriate.

Objectives of the Study

The objectives of this research are to:

- Present the issues that limit the possibilities for compiler optimization for parallel programming support in Java.
- Survey and investigate relatively recent models and environments proposed for HPC parallel programming in Java. The models include support for both shared and distributed memory architectures. These are listed below:
  a. Java Threads [4]
  b. Java Sockets [5]
  c. RMI [6]
  d. mpiJava [7]
  e. MPJ Express [8]
  f. FastMPJ [9]
- Compare the performances of C and an appropriate subset of the above listed parallel programming models in Java by implementing a computational problem. The performance metrics that can be used are elapsed time and speed up. The implementation details are explained below:
  1. Implement a basic sequential algorithm for the matrix multiplication problem in C and Java and capture results. Implement the described parallel algorithm using threads in Java on a single processor and capture the results.

2. Gain understanding of how all/most of the models provide parallelism in Java by implementing the described matrix multiplication parallel algorithm on a multi-processor system (cluster or multicore SMP or both) and capture the results.
3. If time permits, implement the algorithm in C using MPI and capture the results.
4. Deduce conclusions from as many of the following comparisons as possible:
   a. Sequential implementation in C vs. Sequential implementation in Java (Elapsed time).
   b. Parallel implementation in Java using threads vs. Sequential implementation in Java (Speed up).
   c. Parallel implementation in C using MPI vs. Sequential implementation in C (Speed up).
   d. Compare the performance of the various programming models in Java on a multi-processor system (Elapsed time/Speed up for SMP and cluster).
   e. Parallel implementation in C using MPI vs. best performing Java model on a multi-processor system (Elapsed time/Speed up for SMP and cluster).

Chapter II

JAVA AS A PARALLEL PROGRAMMING LANGUAGE

Introduction

M. Ashworth [10] states that the Java language has attracted considerable interest in the popular and technical literature and there are now propositions that Java may be able to meet the needs of the HPC (High Performance Computing) community where other languages have failed. Java clearly has many advantages. Apart from being simple, object-oriented, portable, robust and extensible, a thread based execution model makes it certainly suitable for parallel platforms. Java source is compiled to class files containing machine-independent byte-code, which is like machine code in form but is not specific to any particular hardware. In this byte-code form, the Java class file can be easily transmitted across a network from the server to a Java-enabled client where it is interpreted by the Java Virtual Machine (JVM). This indicates that a Java program can run unchanged on any machine on which a JVM has been written.

Also, recent deployments of HPC infrastructures are significantly increasing the number of installed cores in order to meet the ever increasing computational power demand. The importance of multithreading and parallelism competences is reinforced by this current trend to multi-core clusters [11]. With multithreading support, out of the box networking capabilities, features to take advantage of shared, distributed and hybrid memory models, Java becomes a natural choice for the development of parallel applications. Java can be used to achieve both intra node (shared memory) and inter node (distributed memory in clusters) by simply utilizing threads, Sockets, RMI and other networking support features. However, as Ashworth states [10], even though the performance gap between Java and native languages is usually small for sequential applications, it can be particularly high for parallel applications when depending on inefficient communication libraries, which has hindered Java adoption for HPC. Specific improvements have been proposed to try to improve the performance of Java. On most platforms it is possible to compile Java class files with a Just-in-Time (JIT) compiler. The byte codes are no longer directly interpreted, but firstly compiled on-the fly into machine code just before execution. Native Java compilers which compile directly to machine code perform exceedingly well where strict platform independence is not required. However, even native compilers struggle matching the performance of traditional languages like C, C++ and Fortran, because there are certain features of Java which limit compiler optimization possibilities, which in turn, limits the parallel programming support in Java, these are discussed below.

Parallel Programming Issues in Java

Because of its thread based implementation model, the Java programming language can be easily extended to parallel platforms. It also has the abilities needed for supporting high-performance computations. However, since Java is relatively new, it lacks the extensive scientific support of other languages like C, C++ and FORTRAN. A number of issues which limited the support of Java for HPC applications, were identified by the Java Grande Forum Panel (JGFP) in [12], the most important of which were categorized into numerical issues and concurrency issues.

Numerical Issues

The JGFP in [12] has concentrated on five critical areas where improvements to the Java language are needed: floating-point arithmetic, complex arithmetic, multidimensional arrays, lightweight classes and operator overloading.

- Floating-point Arithmetic: The strict floating point semantics of the Java language, which is used to ensure accuracy and precise reproducibility across a wide range of platforms, results in poor performance. For example, to make use of the processor's fused multiply-add operation, it is forbidden to rearrange operations using associativity. The HPC requirements for floating point computations varies from one application to another, but in most cases developers require a stable developing environment and the ability to utilize the peak performance of the floating point hardware by selecting the most appropriate optimizations. Hence to improve performance optimization, the JGFP proposals allow for a strict floating point declaration along with some alternative options that may facilitate rearranging of certain operations.
- Complex Arithmetic: There is no support for complex arithmetic in Java. To implement complex arithmetic we need to create a complex class whose object's behavior is different from other primitive type numbers. This difference in behavior is against the readability of scientific codes that states that the complex numbers should be used in exactly the same way as any other primitive type. It also states that there should be no compromise in the speed of computation as well. To achieve this, complex number classes can be implemented in Java on top of the existing primitive types, but the object overhead of complex methods makes them unacceptably inefficient. Furthermore, readability and ease of code maintenance is much more difficult to attain, if the syntax and semantics of complex number implementation is significantly different from that of the primitive types.

- Multidimensional Arrays: Most of the computations in the HPC problems are represented using multidimensional arrays. To improve the performance it is necessary for the compiler to easily optimize the operations on multidimensional arrays. The developer often exploits the knowledge of how multidimensional arrays are stored and organized in the memory by writing efficient code to assist compiler optimization. Multi-dimensional arrays in Java are implemented as arrays of arrays. There is no requirement in Java for the elements of a row of a multi-dimensional array to be stored contiguously in memory, making the efficient use of cache locality dependent on the particular JVM. At first it may seem an effective solution but unfortunately, this way of storing arrays leads to jagged arrays, in which all rows are not of the same length, aliasing between rows and changes in shape. Since the compiler has no information about possible shape changes and aliasing, it reduces performance by generating additional loads and stores.
- Lightweight Classes: Lightweight classes facilitate creation of new Java objects. Using this feature many significant features, including complex numbers, can be implemented as classes. As discussed above this would suffer from poor performance when normal Java classes are used. The issue here is how to implement lightweight classes without making many changes to the JVM and at the same time reducing the possibility of acceptance.
- Operator Overloading: Operator overloading is essential to enable additional numeric types to be used in a way which is easy to develop, understand and maintain. Different operators have different implementations depending on their arguments. It

also allows user-defined types. The addition of two complex numbers may be expressed as ‘x + y’ just as with any primitive type. Addition of two operands with user-defined type would usually have to be expressed using an explicit method such as ‘Complex.sum(x,y)’, but this would make scientific codes virtually unreadable. Some technique should be designed to allow overloading of the arithmetic, comparison, assignment and subscripting operators.

Concurrency Issues

The JGFP in [12] further states that the Java language has several built-in mechanisms which allow the parallelism in scientific programs to be exploited. Threads and concurrency constructs are well-suited to shared memory computers, but not to large-scale distributed memory machines. Although sockets and the Remote Methods Invocation (RMI) interface allow network programming, they are too client/server oriented to be suitable for scientific programming. Codes based on RMI would potentially underperform compared to platform-specific implementations of standard communication libraries like the Message Passing Interface (MPI).

The performance and capabilities of RMI are regarded as being key to exploiting Java for high performance parallel and distributed computing by JGFP. Currently RMI is based on the client/server model and communicates over TCP/IP networks, which places limitations on performance especially for closely coupled systems, such as distributed memory multi-processors and workstation clusters, commonly used for HPC applications. Many scientific applications have fine-grained parallelism making fast remote method invocations with low latency and high bandwidth essential.

Chapter III

SHARED MEMORY PROGRAMMING IN JAVA

Introduction

A typical shared memory multi core architecture is shown below in Figure 3.1, where multiple cores share the same memory space. Each core is essentially a processor (shown as CPU in the figure). Applications designed for shared memory environments may make use of Java threads or Open Message Passing Implementations (MPI) for achieving parallelism. The use of Java threads for parallel programming is quite extended due to its high performance, although it is a rather low-level option for HPC. In terms of complexity and programming effort, the use of threads especially for large applications, requires a significant and careful programming effort. Issues like thread contention, deadlocks, shared data access and race conditions are common fallacies. Finally, in order to relieve programmers from the low-level details of threads programming, many concurrency utilities such as thread pools, tasks, blocking queues, and low-level high-performance primitives for advanced concurrent programming were introduced in the Java 1.5 specification. This support has been continuously enhanced in following specifications, ever since. Below, we discuss the current concurrency support in the latest Java 1.8 specification.

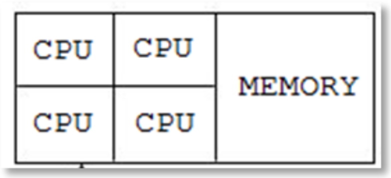
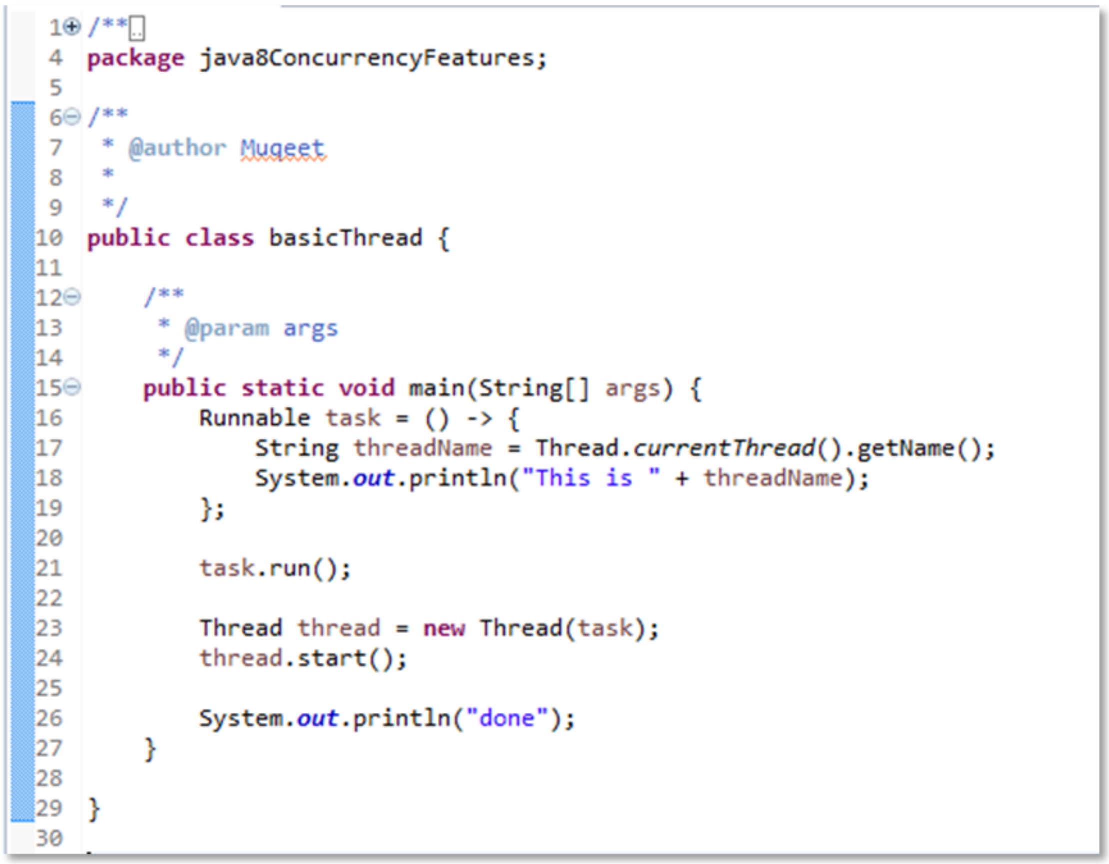


Figure 3.1: A typical shared memory multi core architecture.

Java Threads

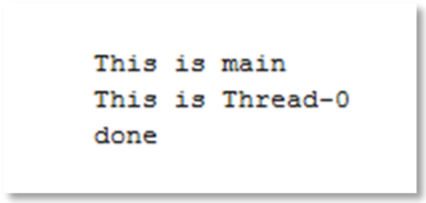
Java threads are a natural choice for parallel programming as the thread class is part of the standard Java libraries since the Java 1.0 specification. As stated in [4] and [13], threads are lightweight processes that exist within a process and share its resources like memory, files etc. This results in using fewer resources for its creation compared to a typical process creation, hence enabling efficient communication. Traditionally, concurrent execution was achieved by a multi-threaded process, on a single processor, by switching the processor execution resources between threads. With recent JVMs, in a shared memory multi-threaded process, each thread can run on a separate processor at the same time resulting in parallel execution [13]. This is due to the fact that current JVMs implement threads on top of native OS threads, which allows them to be scheduled on different processors to achieve parallel execution. However, creation and termination of threads is expensive, so, instead of creating a new thread for each parallel task, maintaining a fixed pool of threads for the entire duration of the program and queuing up parallel tasks often results in more efficiency. To further optimize performance, it is also essential to avoid thread context switch overhead by ensuring that the number of threads are either less or ideally match the number of available processors [4].

Java threads are created either by extending the ‘java.lang.Thread’ class or by implementing the ‘java.lang.Runnable’ interface. The code to be processed by the thread, frequently referred to as task, is usually defined before the actual thread creation. A sample thread creation and execution is illustrated in Figure 3.2 below, based on example in [14]. Lambda expressions, introduced as part of Java 1.8 specification, are used to print out the current thread’s name below. The code executes the runnable task directly on main thread first before starting a new thread. The result of the below code snippet cannot be predicted due to concurrency, the runnable can be called before or after printing “done.” This unpredictability is illustrated below in the Figures 3.3 and 3.4 where multiple executions of the snippet results in different output to the console.



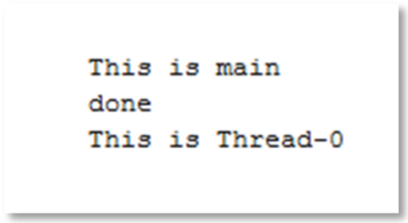
```java
/**
package java8ConcurrencyFeatures;

/**
 * @author Muqeet
 *
 */
public class basicThread {

    /**
     * @param args
     */
    public static void main(String[] args) {
        Runnable task = () -> {
            String threadName = Thread.currentThread().getName();
            System.out.println("This is " + threadName);
        };

        task.run();

        Thread thread = new Thread(task);
        thread.start();

        System.out.println("done");
    }

}
```

Figure 3.2: Thread Spawning Code Snippet



```
This is main
This is Thread-0
done
```

Figure 3.3: Console Output When Runnable Is Invoked Before Printing “Done”



```
This is main
done
This is Thread-0
```

Figure 3.4: Console Output When Runnable Is Invoked After Printing “Done”

This non-deterministic ordering quickly becomes error prone with complex implementations that involve synchronization of multiple threads. To account for mutual exclusion and synchronization over complex pieces of codes are common issues for plain old threads. While this methodology results in thread safe code, it usually limits parallelism that is induced by the exclusion and synchronization schemes built into the code, causing long periods of exclusion for parallel execution [15]. As with any computing cases, such complex coding schemes with low level OS primitives opens up doors to manual errors as developers tend to spend more time focusing on thread synchronization rather than the parallel computation problem at hand. Ideally, developers should be able to make use of efficient high level libraries that handle thread scheduling issues and facilitate a simple framework to support parallelism. Java 1.5 specification, introduced this ability by adding the package ‘java.util.concurrent’ which contains many useful classes for concurrent programming. Since then the concurrent package has been enhanced gradually with every new release, even the recent Java 1.8 specification release facilitates concurrency by providing new classes and improvements. Next, we discuss some highlights of the Java 1.8 specification concurrency package below.

Executors

[16] [17] describes executors as a high level service that are capable of running asynchronous tasks and typically manage a pool of threads, so we do not have to create new threads explicitly. They enable encapsulated access to thread pools, thus decoupling thread management and creation from the rest of the application. This greatly reduces the developer effort to manage thread pools, synchronization and scheduling problems allowing more focus on the actual parallel computation problem. An illustration of task delegation to an Executor Service is shown in Figure 3.5 below.

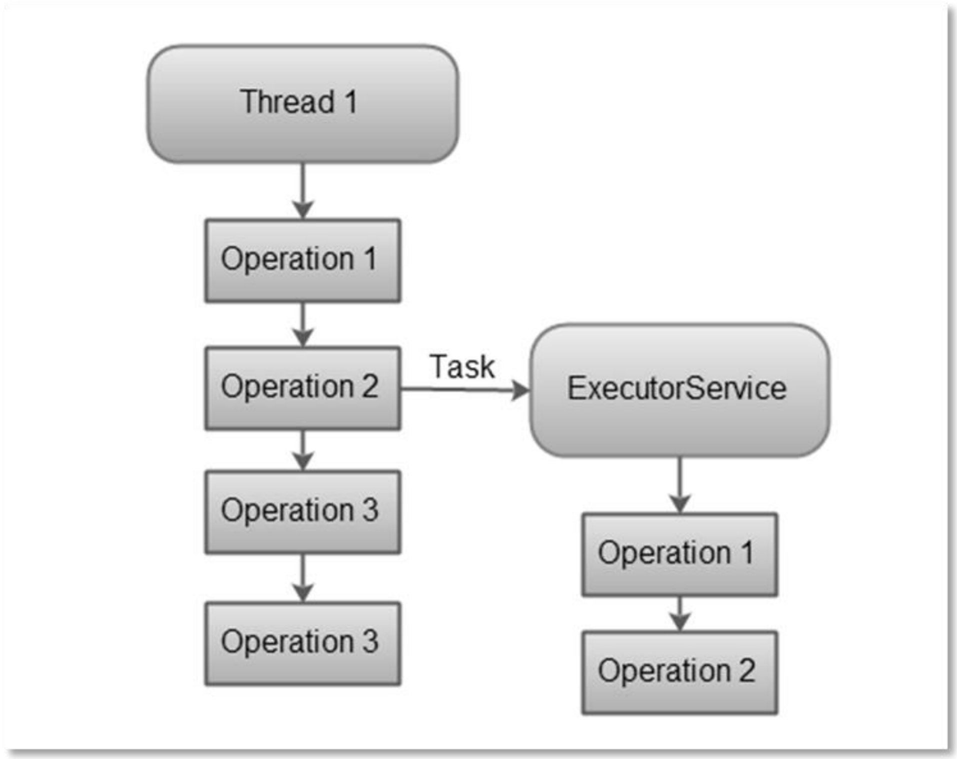
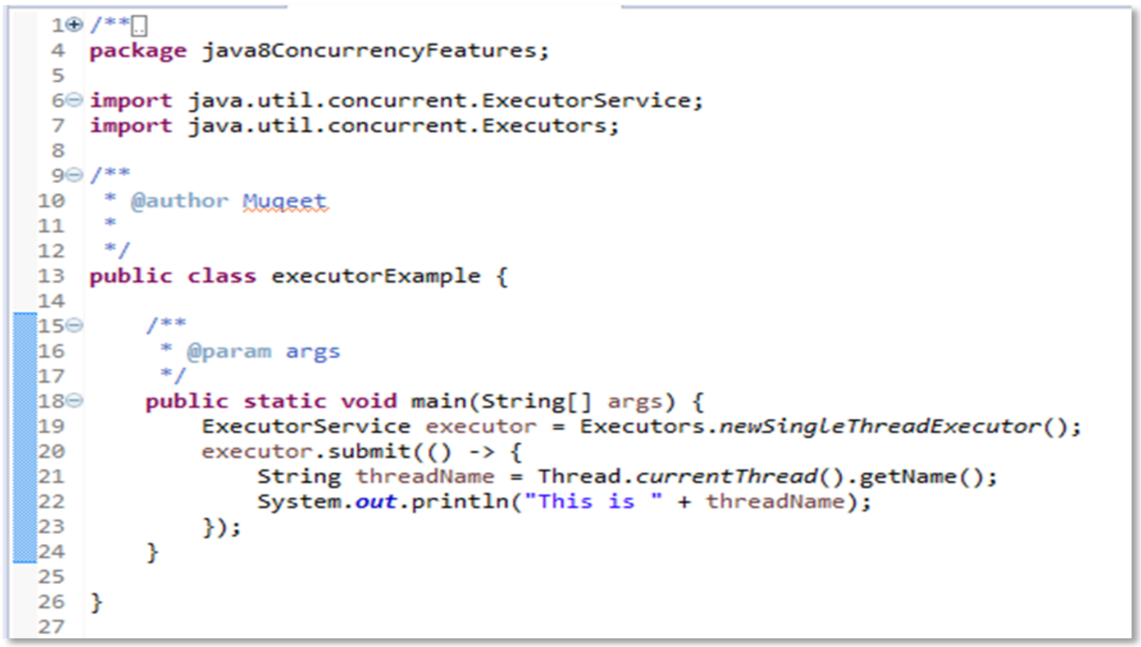


Figure 3.5: Thread Delegating Task to an Executor Service

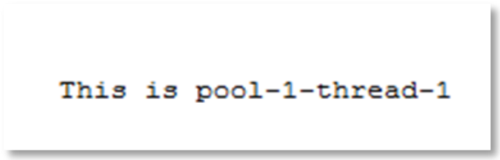
Since an executor service manages a pool of threads which are reused under the hood, we can get away with as many concurrent tasks as needed for the application execution with a single execution service. Executors class provides various factory methods for creating different executor services [17]. The previous thread example (shown in Figure 1) implemented using executor service with an executor of single thread pool size and the corresponding console output is illustrated in Figure 3.6 and 3.7 below, based on example in [18].

Although the output looks similar to the previous example, there is a significant difference. The Java process for executors never exits unless explicitly stopped, until then it keeps listening for new tasks. shutdownNow() and shutdown() methods are provided by the ExecutorService to interrupt all executing tasks and shut down the executor either immediately or by waiting for currently running tasks to finish, respectively.



```java
/**
package java8ConcurrencyFeatures;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * @author Muqeet
 *
 */
public class executorExample {

    /**
     * @param args
     */
    public static void main(String[] args) {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        executor.submit(() -> {
            String threadName = Thread.currentThread().getName();
            System.out.println("This is " + threadName);
        });
    }

}
```

Figure 3.6: Executor Service Example



```
This is pool-1-thread-1
```

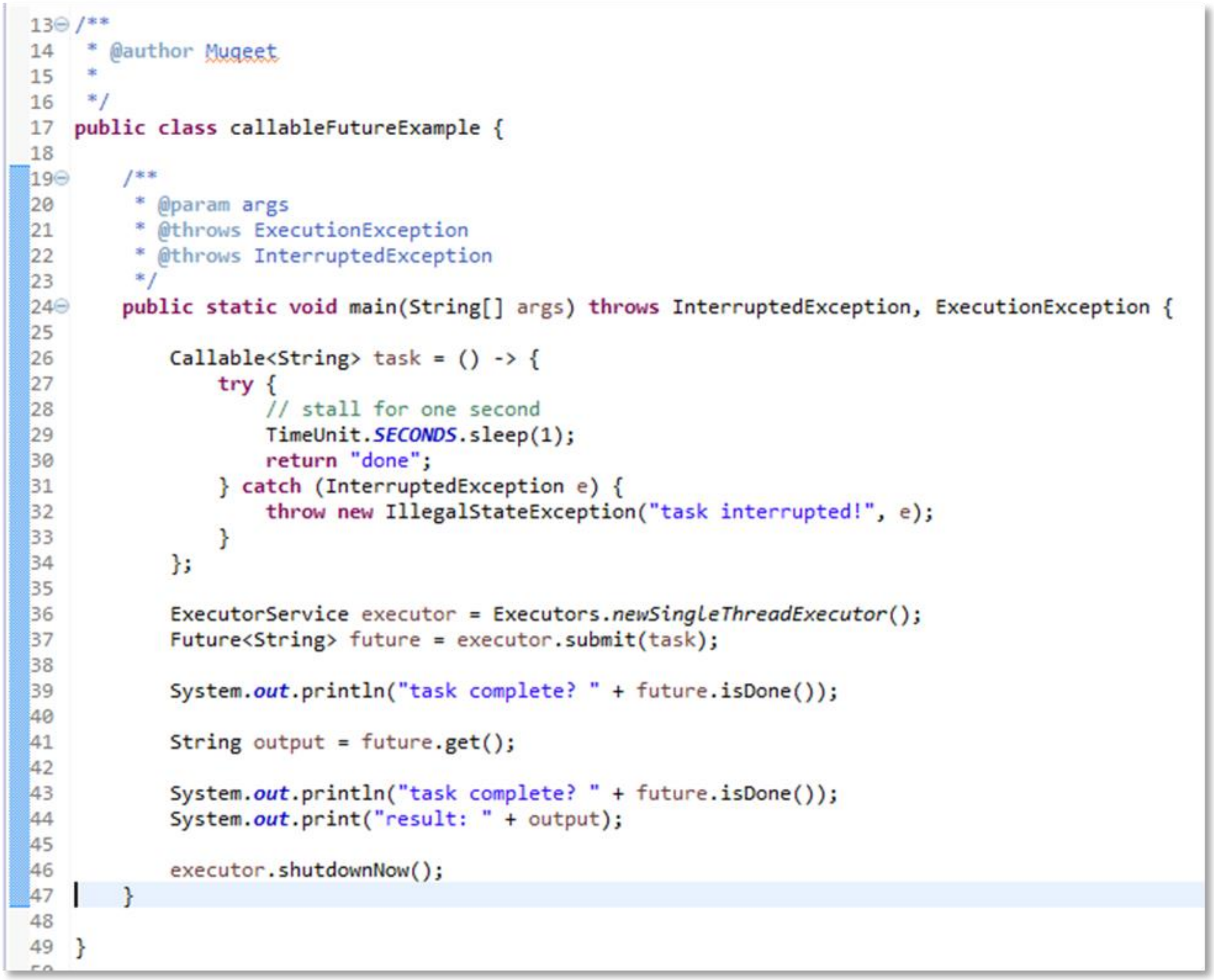
Figure 3.7: Console Output for Executor Service Example

Callables and Futures

Callables [19] is a functional interface that was introduced in the Java 1.5 specification to complement the existing Runnable interface. Unlike Runnable, it is usually utilized to submit a task to a thread or a thread pool for asynchronous execution to either return a result or throw an exception. Callable represents an asynchronous computation, whose value can be accessed via Future [20] object. All the code which needs to be executed asynchronously goes into call() method. Callable can be used along with lambda expression in Java 1.8 specification, since it is also a single abstract method type.

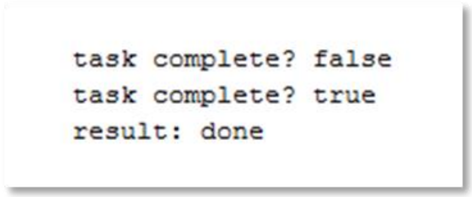
When a Callable is passed to the thread pool, it chooses one thread and executes the Callable to return a Future object in order to hold the result of computation once complete. The get() method of Future can be then used to return the result of the computation or block if the computation is not complete. An overloaded get() method with timeout is also available to avoid indefinite blocking. Future also allows to cancel the task if it's not started, or interrupt if it's already started. Classes like Integer, String etc. can be wrapped using both Callable and Future.

Figure 3.8 below, based on example in [20], illustrates a code extract that depicts the submission of Callable that returns a string, to an Executor and retrieval of the result via a Future object. In the code snippet below, after submitting the Callable to the executor we check if the Future has finished execution via isDone() method which is least likely since the above callable sleeps for one second before returning the string value. Invoking the get() method blocks the current thread and waits until the Callable completes before returning the actual result ”done.” Now the Future is finally done and the following result, shown in Figure 3.9 below, is displayed on the console.



```java
/**
 * @author Muqeet
 *
 */
public class callableFutureExample {

    /**
     * @param args
     * @throws ExecutionException
     * @throws InterruptedException
     */
    public static void main(String[] args) throws InterruptedException, ExecutionException {

        Callable<String> task = () -> {
            try {
                // stall for one second
                TimeUnit.SECONDS.sleep(1);
                return "done";
            } catch (InterruptedException e) {
                throw new IllegalStateException("task interrupted!", e);
            }
        };

        ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<String> future = executor.submit(task);

        System.out.println("task complete? " + future.isDone());

        String output = future.get();

        System.out.println("task complete? " + future.isDone());
        System.out.print("result: " + output);

        executor.shutdownNow();
    }

}
```

Figure 3.8: Code Snippet for Callable and Future



```
task complete? false
task complete? true
result: done
```

Figure 3.9: Console Output for Callable and Future Code Snippet

Locks

Java is a multi-threaded language where multiple threads run in parallel to complete program execution, hence synchronization plays a vital role. Advanced design and coding is required when accessing shared mutable objects concurrently from multiple threads to avoid inconsistent behavior. Fortunately, Java supports thread level synchronization using the keyword ‘synchronized.’ This feature can be used to avoid race conditions when accessing critical sections of the code. Java internally manages synchronization through monitors. Each object in Java is associated with a monitor, which can be locked or unlocked by a thread [21]. Only one thread at a time can obtain a lock on a monitor, all the other threads attempting that monitor are blocked until they obtain the lock. The lock obtained by the synchronized keyword is re-entrant meaning a thread can obtain the same lock multiple times without running into deadlocks. The *synchronized* keyword, however, does not allow separate read and write locks, thereby limiting concurrent reads and potentially limiting scalability. Also, it does not provide any means in the API to timeout or interrupt a thread waiting for a lock, which could lead to starvation or deadlocks if synchronization was implemented incorrectly. To allow more fine grained control of the critical section and flexible structuring of the application, explicit locks were introduced in the Java 1.5 specification and then improved upon in the following releases. Apart from having the general capabilities of timeouts and interrupts, multiple variations of lock implementation are available, these are presented below [22]:

Reentrant Lock

Reentrant Lock [23] is a mutual exclusion lock with the same basic behavior as the implicit monitors accessed via the synchronized keyword but with extended capabilities. It is owned by the last thread that acquired the lock and has not yet unlocked it. If the current thread owns the lock then the method returns immediately else the lock is acquired if it is not acquired by any other thread.

Apart from the ability to be interrupted and timeouts, this lock provides a couple of enhancements over the traditional synchronized keyword. First, the constructor accepts a fairness boolean parameter, which if set to true will yield the lock to the threads that are long waiting and guarantees less starvation. However, this setting may result in slower execution times than the default settings. Second, the tryLock() method avoids thread wait by only granting the lock only if available at the invocation time. If the previous fairness setting is enabled on the lock then tryLock() method can be used to immediately acquire the lock, if available, disregarding the other fairness parameter for other waiting threads.

ReadWrite Lock

ReadWrite Lock [24] specifies another lock provided since the Java 1.5 specification, which is able to acquire locks for both read and write access. The main motivation for read-write locks is based on the fact that mutable variables can be read concurrently and safely as long as no other process is writing to this variable. This implies that the read lock can be held simultaneously by multiple threads as long as no other threads hold the write lock. This often results in greater concurrency for accessing shared data when compared to mutual exclusion lock, which can only be fully realized on a multi-processor in cases where reads are more frequent than writes.

Stamped Lock

Stamped Lock [25] is a new capability based lock introduced in Java 1.8 specification. It has three modes for controlling read/write access. Stamped locks return a long value, often referred to as a stamp, which can be used to check the lock validity or to release a lock. Stamped locks do not implement reentrant characteristics. A new stamp is returned for each invocation of the stamped lock API. Sometimes, the response is blocked if no lock is available, even if the same thread already holds a lock. This calls for close attention while coding to avoid running into deadlocks. Apart from read and write, a new lock mode called optimistic locking is also supported by stamped lock. An optimistic read lock is acquired by calling tryOptimisticRead() which always returns a stamp without blocking the current thread, irrespective of the lock being actually available. If there's already a write lock active then the returned stamp equals zero. validate(long) method returns true if the lock has not been acquired in write mode since obtaining a given stamp. It is stated in [25] that stamped lock can be thought of as an extremely weak version of a read lock that can be broken by a writer thread at any time. Using optimistic mode for short read only code segments often reduces contention and improves throughput. However, values read using optimistic mode may be inconsistent so it is used only when the data representation is clearly understood to repeatedly invoke validate() method, if needed, to check consistency.

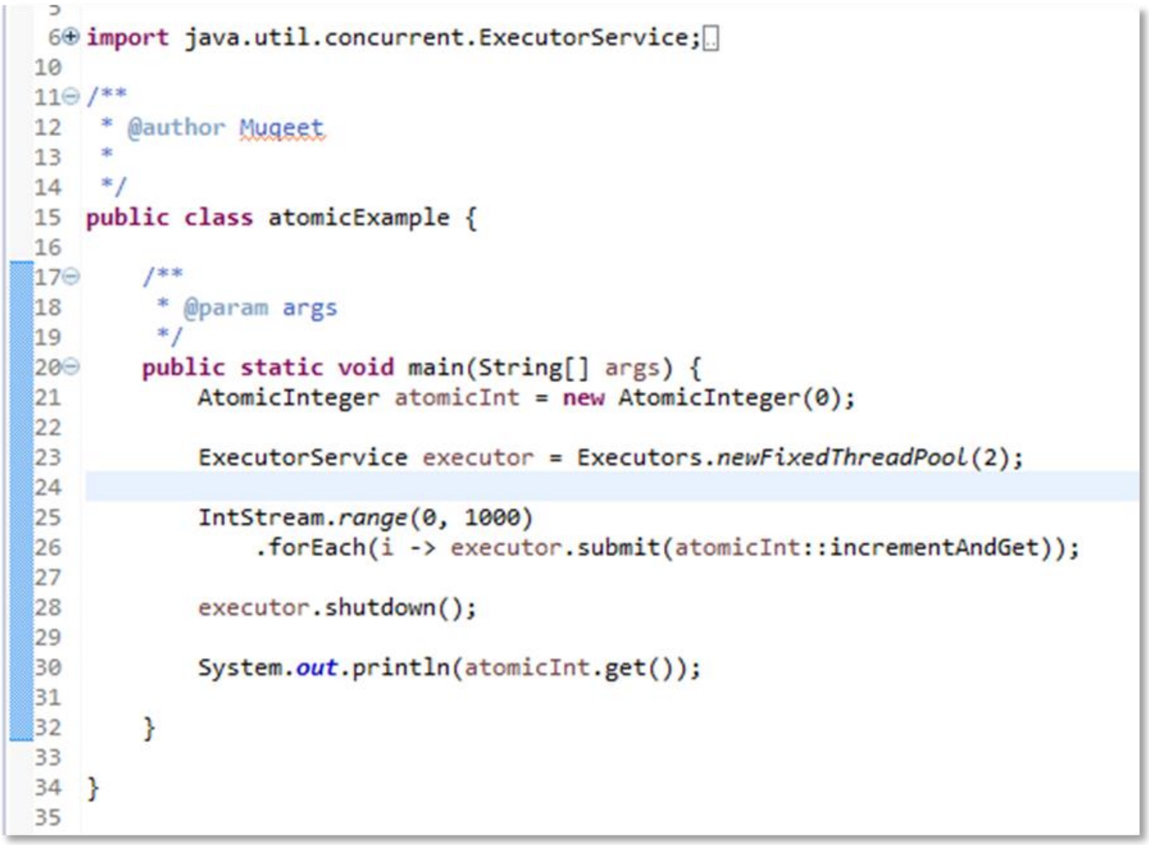
Another important feature supported by Stamped lock class is the lock mode upgrade feature. Sometimes it's desirable to convert a read lock into a write lock, if possible, without unlocking and locking again. Stamped Lock provides methods like tryConvertToReadLock(), tryConvertToWriteLock() and tryOptimisticRead() to achieve the same.

Atomic Variables

The package ‘java.concurrent.atomic’ [26], added in Java 1.5 specification, provides many useful classes to carry out atomic operations. An operation is atomic when we can safely perform the operation in parallel on multiple threads without using the synchronized keyword or locks. Internally, the atomic classes make heavy use of compare-and-swap (CAS), an atomic instruction directly supported by most modern CPUs [27]. Those instructions usually are much faster than synchronization via locks. So it’s generally advisable to prefer atomic classes over locks in cases where we just have to change a single mutable variable concurrently. Figure 3.10, below illustrates an example of an Integer Atomic variable derived from [28] [29]. It is evident from the example that by using an AtomicInteger instead of an Integer, the number can be incremented concurrently in a thread-safe manner without synchronizing the access to the variable. The incrementAndGet() method atomically increments the current value by one [30], so it can be safely called from multiple threads concurrently. Finally, the shutdown method is invoked to attempt a controlled shutdown of the executor service. The console output of the code snippet is shown below in Figure 3.11.

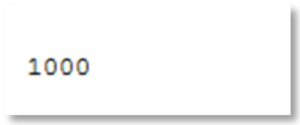
ConcurrentMap

The interface ConcurrentMap [31] defined in ‘java.util.concurrent.ConcurrentMap’ extends the map interface and defines one of the most useful concurrent collection types which is capable of handling concurrent access to it. TheConcurrentHashMap is an implementation available in the concurrency package that is very similar to the ‘java.util.HashTable’ class, except that the former offers better concurrency. For instance, ConcurrentHashMap does not lock the map for reading. Additionally, ConcurrentHashMap does not lock the entire Map when writing, it only internally locks the part of the map that is being written to.



```
import java.util.concurrent.ExecutorService;

/**
 * @author Muqeet
 *
 */
public class atomicExample {

    /**
     * @param args
     */
    public static void main(String[] args) {
        AtomicInteger atomicInt = new AtomicInteger(0);

        ExecutorService executor = Executors.newFixedThreadPool(2);

        IntStream.range(0, 1000)
            .forEach(i -> executor.submit(atomicInt::incrementAndGet));

        executor.shutdown();

        System.out.println(atomicInt.get());

    }

}
```

Figure 3.10: Atomic Integer Code Snippet



```
1000
```

Figure 3.11: Console Output for Atomic Integer Code Snippet

With Java 1.8 specification, ConcurrentHashMap [32] has been further enhanced with new methods to perform parallel operations upon the map. Like parallel streams, these methods use a special ForkJoinPool available via ForkJoinPool.commonPool(). This pool uses a preset parallelism which depends on the number of available cores. Three kinds of parallel operations were added in latest Java 1.8 specification, these are listed below [32]:

- forEach—Used to perform a given operation on each element of the map.
- Search—Used to return the first not null result of applying the given function on each element.
- Reduce—Used to accumulate the result of applying the given reduction function on each element.

All of the above operations are defined in the API as functions with the following four variations for the input arguments - keys, values, entries and key-value pair arguments. These operations also accept an argument called parallelismThreshold that indicates the minimum collection size for the operation to be executed in parallel [32]. For instance, if we pass a threshold of 100 and the actual size of the map is 99 then the operation will be executed sequentially on a single thread.

Chapter IV

DISTRIBUTED MEMORY PROGRAMMING IN JAVA

Introduction

With the recent availability of high-performance low-cost networking components, it is now becoming practical and cost-effective to assemble a network of workstations and/or PCs into a single distributed-memory computing resource. Applications running on workstation clusters built-up over local area networks can make use of the high percentage of idle time on under-utilized desktop machines. Such systems are increasingly available because of the decrease in prices of processors and the high-bandwidth links to connect them. Figure 4.1, below shows a typical multi core distributed memory architecture. The communication network in the figure could be a local area network such as an Ethernet, or a wide area network such as the Internet. Programming parallel and distributed systems requires a different set of tools and techniques than that required by the traditional sequential applications. Some of the prevalent Java based distributed memory programming models are discussed below.

Java Sockets

Sockets are a low-level programming interface for network communication, which allows exchanging streams of data between applications [5]. Due to the socket implementations on almost every network protocol and the extended API, the socket API can be considered as the standard low level communication layer. Therefore, sockets are used frequently for implementing the lowest level of network communication in Java.

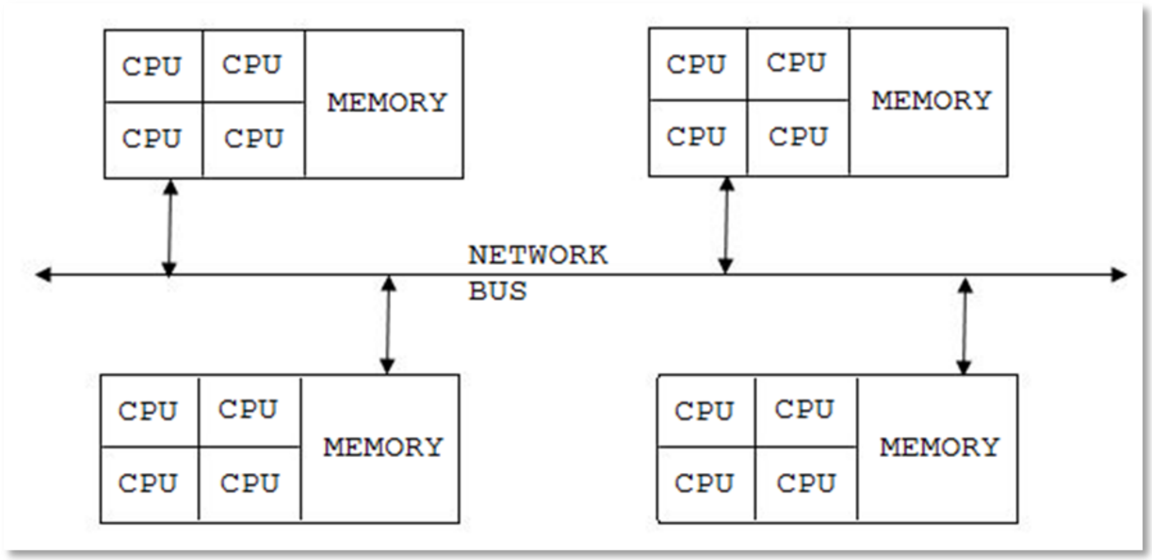


Figure 4.1: A Distributed Multicore Memory Architecture

Java has two main socket implementations, the widely extended Java IO sockets, and Java NIO (New I/O) sockets which provide scalable non-blocking communication support. However, both implementations do not provide high speed network support nor HPC tailoring. Their implementation on top of the JVM sockets library limits their performance benefits.

Taboada demonstrated in [33], Java sockets usually lack efficient high speed network support and has to resort to inefficient TCP/IP emulations for full networking support. Furthermore, he also introduced a new high performing socket implementation, known as Java Fast Sockets (JFS) in [33] that attempts to provide an efficient Java communication middleware for high performance clusters to support parallel and distributed Java applications. He proved experimentally that the JFS reduced socket processing CPU load up to 60% compared to traditional sockets, which was a significant improvement. This implementation can also be used by the middleware developers to implement JFS based high level communication APIs like message passing and RMI implementations to further enhance the overall performance of the distributed application.

Java RMI

Java Remote Method Invocation (RMI) [6] [34] is an API which supports seamless remote method invocation on objects in different JVMs across networks. RMI provides a simple and direct model for distributed computation with Java objects. These objects can be new Java objects, or can be simple Java wrappers around an existing API. Simply stated, RMI attempts to make the designing of a distributed application as simple and straightforward as designing a non-distributed application. Sending data across networks or translation of data into objects is abstracted by the RMI API providing a simple interface to the programmer. This abstraction helps in avoiding writing low level protocols to handle data reads from the network. Also, since RMI is Java based, it brings the strengths of Java safety and portability to distributed computing. Using the standard Java native interface (JNI), RMI can connect to existing and legacy systems in non-Java languages as well.

RMI has evolved and now includes several enhancements for security and transport mechanisms [35]. However, both RMI and Java object serialization as currently implemented are significantly computationally expensive. The optimization of the RMI protocol has been the goal of several projects, such as KaRMI [36], RMIX [37], Ibis RMI [38] and Manta [39]. Each of them attempted to improve RMI performance by using different protocols but still fell short by either not providing a standard API or inferior performance over large data sets.

Another research was conducted in 2007 by Toboada, who demonstrated in [40] an efficient implementation of the RMI protocol using direct high performance sockets library (JFS)

[33] on a cluster. This implementation was fully transparent to the user and is interoperable with other systems, hence negating the need for source code modifications. Other optimizations like performing native array serialization, reducing class annotations also helped in significantly reducing the overhead associated with RMI calls, especially in high speed networks. Another Java based GRID middleware, named ProActive [41] was introduced in 2009 by Amedro and Taboada. Its communication layer was built on top of the RMI protocol. The research demonstrated that by substituting the default socket based RMI implementation with an efficient high performing sockets implementation, the performance of ProActive could be improved significantly.

Even though research like [36] [37] [38] and [39] improve the performance of the native Java RMI implementation, due to inherent remote machine involvement, RMI is subjected to increased latency even if the nodes in the cluster are connected via a high speed network. Also, since RMI requires creation of additional threads to manage remote requests for data, context switching and synchronization incurs additional overhead to exchange data between multiple JVMs. Moreover, issues like network failures and security are left with the programmers to be dealt with. At its core, RMI applications are based on a client and server model which is not suitable for high performance scientific applications, particularly, in applications that use parallel decomposition techniques. Instead of invoking methods on remote data, these parallel applications require passing of data between processes to be processed locally [40].

The use of non-standard APIs for alternate better implementations and the insufficient overhead reductions, still significantly larger than socket latencies, have restricted RMI applicability. Therefore, although Java communication middleware used to be based on RMI, current Java communication libraries use sockets due to their lower overhead [33]. The use of low level API for sockets requires higher programming effort, but allows for higher throughput, which is the key in HPC.

Message Passing

Message passing Interface (MPI) [42] [43] is a well-known established standard for parallel programming for languages compiled to native code. Due to its origins as a C/Fortran library, the design of MPI is distinctly non-object-oriented and unfamiliar to the Java environment. With the popularity of Java, there have been several research efforts to provide MPI for Java. A message passing like interface for Java has the advantage of being easily recognizable to the HPC community, with the least learning overhead. Furthermore, MPI has various benefits, for instance, its backing for collective communication and its ability to run on both shared and distributed memory systems. Because of this distributed support, message passing for Java offers the potential to develop a parallel application to run within both a single and multiple JVM environments.

The Message Passing standard implementations in Java (MPJ) have been realized by either: (a) using Java RMI (all Java implementation); or (b) wrapping an underlying native messaging library like MPI through Java Native Interface (JNI); or (c) using Java sockets. Each implementation type caters to specific situations and setups, but does exhibit associated trade-offs. Taboada demonstrates in [40], the use of Java RMI (for an only Java approach) as the foundation for MPJ libraries, ensures portability. However, it might not be the most efficient solution in the presence of high speed communication hardware, as RMI is unable to take advantage of such infrastructure. The use of JNI, on the other hand, allows for efficient use of high performance communication networks using native libraries, but has portability problems. The use of Java sockets requires a substantial programming effort, especially in order to provide scalable solutions, but it significantly outperforms RMI-based communication libraries. Although most of the Java communication middleware is based on RMI, MPJ libraries looking for efficient communication seemed to have followed the latter two approaches.

There are two main specifications of MPJ: the mpiJava API [44] and the MPJ API [45]. Both specifications are largely defined by the same group of researchers and are quite similar, differing mainly in the naming conventions. The mpiJava specification is followed by the majority of the current available MPJ implementations and is effectively the standard MPJ specification. There have been many projects that have implemented some MPJ specification. Many of these projects were short lived and are no longer being actively worked on or supported. After the modular component approach was introduced by Baker in 1999 [46], several message passing libraries were developed around it. With this approach, the entire communication system is composed of layers. At each layer there might be several alternative components available for use, these components are generally knows as devices. This design allowed users to select specific devices at particular layers to meet some specific requirements. For instance, there might be an all Java device offering portability and another JNI based device that offers better performance, available at a particular layer. Based on the requirements, the user can then make a selection from the available devices for that layer.

Apart from MPJ implementations, native MPI implementations such as MPICH [47] and OpenMPI [48] also contain a low level communication device layer that allows new devices to be plugged in as support for new networks is added or removed. In this research, we discuss two of the most recently proposed MPJ implementations based on modular component approach [46]—MPJ Express [8] and Fast MPJ [9]. These are currently both being actively developed and supported. They both implement the same MPJ specification of mpiJava API and provide good performance and portability.

MPJ Express

The mpiJava library [7] implements the mpiJava specification. It was introduced by the same group who worked on the specification itself. For several years this was the most widely used MPJ implementation. It was a thin wrapper implementation on top of an underlying native MPI library, with reduced JNI overhead. Although it offered high performance, its dependency on JNI introduced portability issues. Additionally, it was not thread-safe, thus unable to take advantage of multi-core systems through multithreading. As a result of these drawbacks, the mpiJava project was superseded by the development of MPJ Express.

MPJ Express [8] is a thread-safe, message-passing implementation of the mpiJava specification, completely written in Java. Since, its thread safe it supports the highest level of multithreading support defined in the MPI specification. It is based on a modular design that includes a pluggable architecture of communication devices. The topmost layer provides the full API of the library. The next two layers contain the collective and point-to-point communication primitives. The point-to-point primitives are implemented on top of the MPJ device layer, namely mpjdev. The mpjdev layer in turn supports two implementations, one of which is a 100% Java implementation, namely xdev and the other is a *native* implementation built on top of a native MPI implementation designed to elevate features of native MPI libraries, when needed. The pure Java implementation of mpjdev currently relies on three implementations for the xdex device, present at the lowest level of its stack: *niodev* over Java NIO sockets, *mxdev* using JNI allowing support for Myrinet network and *smpdev* to be used in shared memory environments. A combination of *smpdev* and *niodev* can be used to support execution of parallel Java programs in the hybrid configuration using multicore and cluster configurations respectively. This combination is available to use out of the box, using *hybdev* cluster configuration. This design of xdev device layer allows MPJ Express to implement support for new communication libraries with relative ease, when needed. Furthermore, these implementations allow MPJ Express to be used in both shared and distributed memory programming models. Also, this project is the most active in terms of adoption by the HPC community, presence in academic and production environments, and available documentation. In this paper, we capture results for the following four configurations by using out of the box communication device drivers from MPJ Express: multicore (*smpdev*), *niodev*, *hybdev* and *native*.

FastMPJ

FastMPJ [9] is a high performance and scalable message passing implementation of the mpiJava specification completely written in Java, with a modular design. FastMPJ supports several high speed networks and makes a number of devices available at different levels. Most significantly it includes the xxdev device layer. xxdev is an extension of the xdev API present in MPJ Express. Unlike xdev, xxdev API supports the direct transmission of any serializable object without using buffers. Also, apart from being simple and concise, it supports both blocking and non-blocking point to point communication. Higher level functions such as collective communication functions are implemented at higher levels in the FastMPJ stack. This allows the new xxdev devices to be implemented with relatively little effort and negates the need to make other changes within the application.

FastMPJ provides scalable collective functions with alternative algorithms for each collective primitive. Based on the message size and the characteristics of the HPC environment, it can automatically select the best algorithm to get the best possible performance at runtime. Due to this feature, its primitives are identified as topology aware. As stated in [9], more stable runtime framework, higher performance and scalability help make FastMPJ rival the performance of other MPI libraries, including MPJ Express. Unfortunately FastMPJ is a commercial offering and could not be implemented in this research for comparative study. However, Taboada demonstrated in [9] that by using an efficient implementation of the communication middleware, FastMPJ was able to rival native MPI performance and scalability, even outperforming it in some scenarios. This is a significant improvement over other traditional MPJ implementations and greatly reduces the gap between Java and native languages in HPC applications. Implementation details, test results and research can be found in [9].

Chapter V

PROBLEM DESCRIPTION: MATRIX MULTIPLICATION

Introduction

Matrix multiplication is one of the most central operations performed in scientific computing and pattern recognition [49] [50] [51]. Simply stated, multiplication of large matrices requires a significant amount of computation time as its complexity is O (n^3), where n denotes the matrix dimension. It can be scaled for a wide range of performance, because the computation grows with the order of n^3 for matrices of order n. More efficient sequential matrix multiplication algorithms have been implemented. Strassen's algorithm, devised by Volker Strassen in 1969 [52], reduces complexity to O ($n^{2.807}$). Even with these improvements performance is limited.

Formal Definition

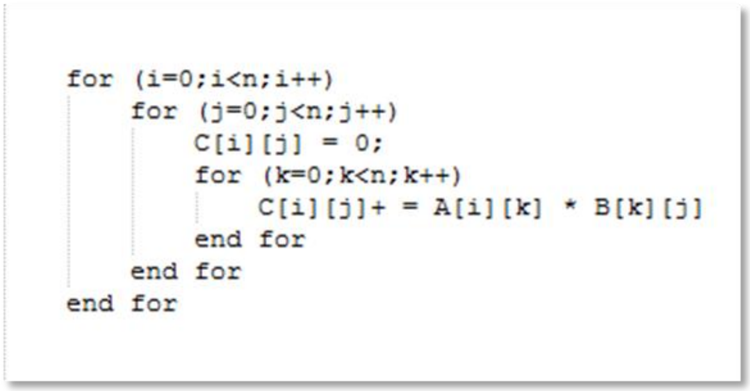
Given a matrix A (n x n) n rows and n columns, where each of its elements is denoted by A_{ij} with $0 \le i < n$ and $0 \le j < n$, and a matrix B($n \times n$) of n rows and n columns, where each of its elements is denoted by B_{ij} with $0 \le i < n$, and $0 \le j < n$, the matrix C resulting from the operation of multiplication of matrices A and B, $C = A \times B$, is such that each of its elements is denoted as C_{ij} with $0 \le i < n$ and $0 \le j < n$, and is calculated by the formula shown in Figure 5.1 below.

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} . B_{kj}$$

Equation 5.1: Matrix Multiplication for Each Element of Resultant Matrix C

Sequential Algorithm

In this paper we used a basic sequential implementation shown in Figure 5.1 for both C and Java with a time complexity of O (n^3).



```
for (i=0;i<n;i++)
    for (j=0;j<n;j++)
        C[i][j] = 0;
        for (k=0;k<n;k++)
            C[i][j]+ = A[i][k] * B[k][j]
        end for
    end for
end for
```

Figure 5.1: Sequential Matrix Multiplication Algorithm

Simple Parallel Algorithm

A parallel implementation of matrix multiplication offers another alternative to increase performance [53] [54] [55]. Matrix multiplication is composed of basic computations which makes it simple and appropriate means for evaluating the performance of the Java support for parallelism vs. the native C implementation. Generally stated, parallel processing is based on numerous processors working collectively to accomplish a task. The basic idea is to partition the computation into smaller tasks and distribute them among the processors. This approach reduces computation time by a maximum factor of the number of processors available [53]. In this paper, we implement matrix multiplication across both shared memory and distributed memory systems.

Shared Memory Algorithm

In shared memory systems, processors share a common memory and data is easily accessible to all processors. Consider two square matrices A and B with dimensions n×n, that are to be multiplied in parallel using p processors. All the processors will have access to all the rows and columns of matrices A and B. For simplicity, let us assume that the rows of matrix A can be equally divided across all processors. To achieve parallelism, we then simply allocate an equal set of rows of matrix A across each processor and then multiply it with the entire matrix B to compute row vectors of resultant product matrix C [54]. This is illustrated in the Figure 5.2 below, for n=4 and p=2. In the figure below, processor P0 multiplies the first two rows of matrix A with entire matrix B to yield first two rows of matrix C. Similarly, processor P1 computes the remaining rows of matrix C in parallel by multiplying remaining two rows of matrix A with matrix B.

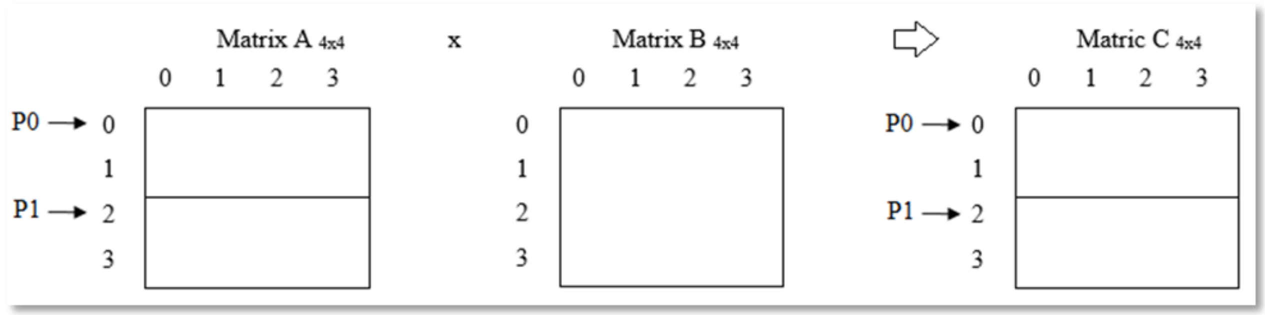


Figure 5.2: Shared Memory Implementation of Matrix Multiplication

Distributed Memory Algorithm

In distributed memory systems, each processor has only local memory, and the data is exchanged as messages between processors. Implementing a matrix multiplication algorithm across multiple processors, typically involves two types of processes namely, manager process and worker process [55]. Generally speaking, the manager process issues commands to be performed by the worker processes. The manager process is also responsible for the issuing of matrix multiplication requests, partitioning of the input matrices, transmission of the sub-matrices, reception of the results, and time keeping duties. The worker process accepts matrix multiplication commands from the manager process, performs all of the actual matrix calculations, and returns the results in the form of sub-matrices, to the manager process.

Consider two square matrices A and B with dimensions n×n, that are to be multiplied in parallel using p processors. In this implementation, it is assumed that only one process, typically the manager process, has access to both matrices A and B. In the first step, the manager process splits matrix A into 'p-1' partitions and scatters them across all the available worker processes along with the entire matrix B. Once the worker process receives row partition of matrix A and the entire matrix B from the manager process, it then performs matrix multiplication to compute an output sub-matrix. Finally, each worker process returns its output sub-matrix to the manager process, which then gathers the partial results to form the resulting product matrix C. This algorithm [54] is illustrated in the below steps, for n=8 and p=4.

Step 1: The manager process (P0) scatters matrix A into 'p-1' (=3) partitions. The size of the row partition is equal to n/(p-1) among all workers, in cases where n%(p-1) = 0. In other cases, where n%(p-1) <> 0, the extra rows are balanced out by passing an additional row among worker processes whose ranks are less than or equal to n%(p-1). Rank is the unique id assigned to a process in the MPI implementation. In this particular case, n/(p-1) = 2 and n%(p-1) = 2, which results in worker processes with ranks 1 (P1) and 2 (P2) being assigned an extra row of matrix A along with the usual set of n/(p-1) rows. This is illustrated in Figure 5.3 below.

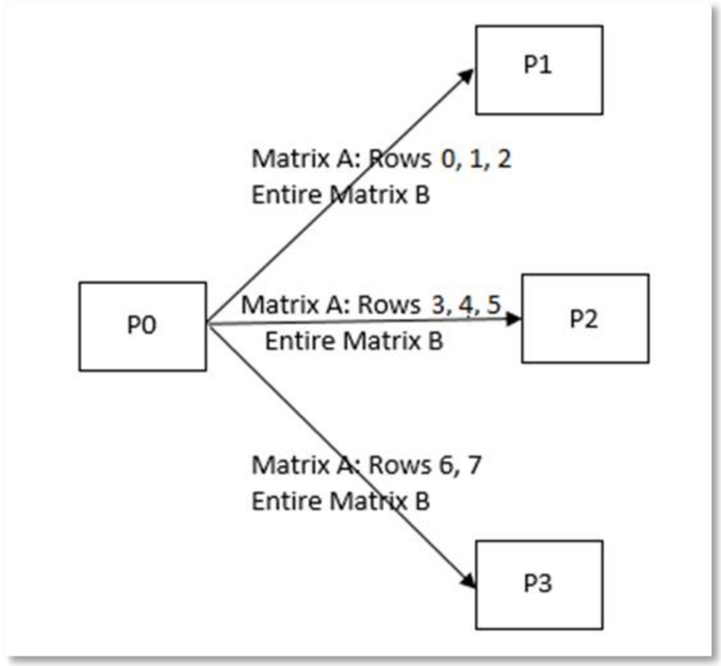
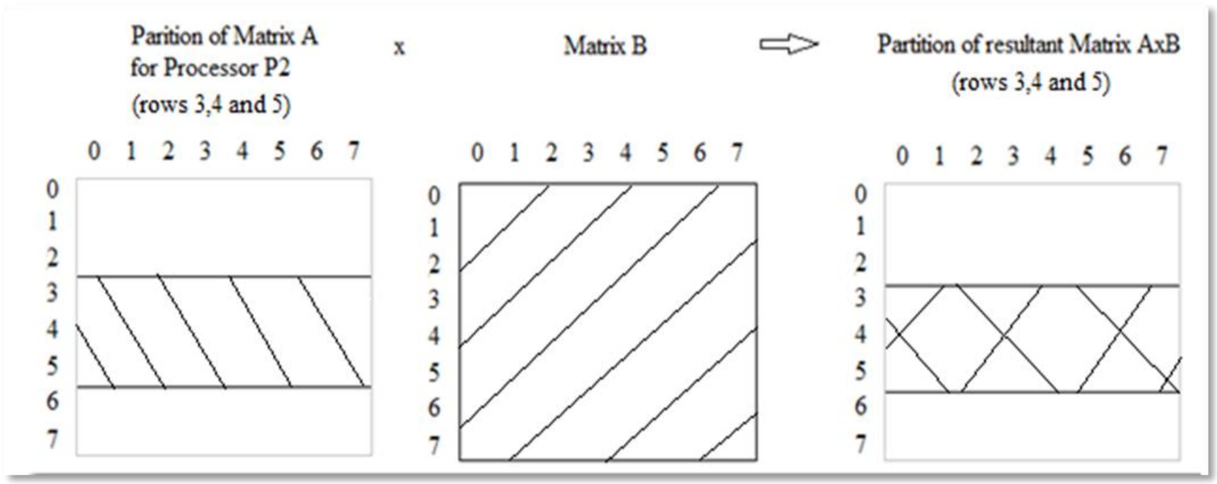


Figure 5.3: Distributed Memory Implementation—Asynchronous Data Broadcasting

Step 2: Each worker process then performs distributed matrix multiplication of the received row partition of matrix A and entire matrix B. Figure 5.4 below, illustrates distributed matrix multiplication for Processor P2 which multiples rows 3, 4, and 5 of matrix A with entire matrix B to yield partial result of rows 3, 4, and 5 for resultant matrix AxB.



Parition of Matrix A for Processor P2 (rows 3,4 and 5)
x
Matrix B
Partition of resultant Matrix AxB (rows 3,4 and 5)
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7

Figure 5.4: Distributed Memory Implementation–Distributed Matrix Multiplication for P2

Step3: Each worker process asynchronously sends the partial result computed in step 2 to the manager process. The manager process then gathers them to complete the final resultant matrix AxB. This is illustrated in Figure 5.5 below

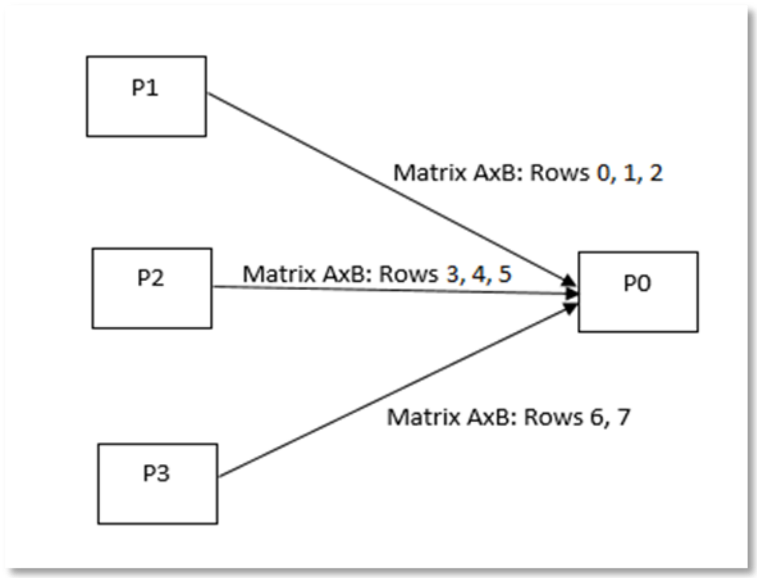


Figure 5.5: Distributed Memory Implementation–Manager Process Gathering Partial Results

In the next chapter, we present the captured results for performance measures like elapsed time and speed up, evaluate results and draw conclusions by comparing appropriate models for both serial and parallel realizations of the above described algorithms.

Chapter VI

COMPARISON AND TEST RESULTS

Performance Metrics

In this paper we use metrics like Elapsed time and Speed up to capture the performance of the various matrix multiplication implementations.

Elapsed Time

Elapsed time is the amount of time that passes from the start of an event to its finish. A sequential algorithm is usually evaluated in terms of its execution time, expressed as a function of the size of its input [56]. In this paper we calculate elapsed time by using clock() and System.currentTimeMillis() methods, for C and Java respectively. The elapsed time is captured as follows:

- For serial implementations, elapsed time is captured around the outermost for loop, shown in Figure 5.1 of Chapter V.
- For parallel implementation, the elapsed time includes the following steps of the simple parallel algorithm described in Chapter V.
    a) Time taken for the master process to split matrix A into (p-1) partitions in a p processor/core system.
    b) Time taken for the master process to send above row partitions of A and complete matrix B to all the (p-1) worker processes asynchronously.
    c) Time taken by each worker process to receive the relevant row partitions of matrix A and entire matrix B asynchronously.

d) Time taken by each worker process to perform its computation and send back its partial result to the master process asynchronously.

e) Time taken by the master process to gather all partial results from the worker processes and merge them into the resultant matrix.

Speed Up

Speed up is a measure that captures the relative benefit of solving a problem in parallel [56]. Speed up of a parallel algorithm is defined as $S_p = T_s/T_p$, where T_s is the algorithm execution time when executed serially, and T_p is the algorithm execution time using p processors. Theoretically, the maximum speed up that can be achieved by a parallel computer with p identical processors working concurrently on a single problem is p.

Experiment Strategy

Matrix Population

Since the input variations may vary the results across multiple runs or other programming languages, we kept the matrix population strategy simple and consistent in order to be able to reproduce it exactly each time. This was achieved by initializing both matrices A and B with the sum of its indices, that is A[i][j] = B[i][j] = i+j. This strategy allowed us to obtain more computation focused results.

Matrix Memory Optimization

Communicating two dimensional arrays using MPJ Express is severely hampered by the performance of Java serialization [8]. To get better comparisons we decided to flatten the two dimensional arrays onto a one dimensional array and communicate using native data types. For uniformity, this is done across the board for both C and Java implementations. This means the element at location ij of matrix A with size N, will be accessed as A[(i*N)+j] instead of A[i][j].

Result Capture Strategy

All the results for elapsed times are recorded as average of five consecutive runs to minimize any unrelated interferences. Also, in an attempt to ensure best run times for Java programs, the methods System.gc() and System.runFinalization() are exclusively invoked to run the garbage collector and finalize methods of discarded objects before entering compute intensive section of the code. Also all the derived speed ups were rounded off to three decimal places.

Overall Experimental Configuration

Hardware Setup

We used a Virtual Private Cluster (VPC) of three identical amazon EC2 c4.2xlarge [57] instances/virtual machines, specification are as follows:

- 64 bit Amazon Linux AMI 2016.03.0 with amazon virtualization layer, named HVM.
- Each node consisted of 8 vCPUs. Each vCPU was a hyperthread of an Intel Xeon E5-2666 v3 (Haswell) 2.9 GHz processor core.
- 15 GB memory
- Secondary storage of 16 GB was configured with a dedicated throughput of 1000 Mbps.
- Also, the instances were placed in the same placement group which enabled low latency, full bisection 10 Gbps bandwidth between instances.

More details on Amazon EC2 configurations can be found in [57] and [58].

Compiler Versions

- C compiler – 64 bit GCC version 4.8.3 20140911 (Red Hat 4.8.3-9)
- Java compiler – 64 bit Oracle javac 1.8.0_45
- MPJ Express v0_44
- Open MPI 1.6.4

Communication Directives

In this paper, the following communication directives were used to achieve inter process communication in parallel implementations, where applicable.

C—Open MPI implementation

- MPI_Send [59] - blocking send operation.
- MPI_Recv [59] - blocking receive operation.

Java—MPJ Express implementations

The equivalent of above directives were used in MPJ Express Java implementations

- MPI.COMM_WORLD.Send [60] – blocking send operation.
- MPI.COMM_WORLD.Recv [60] – blocking receive operation.

Serial Implementations

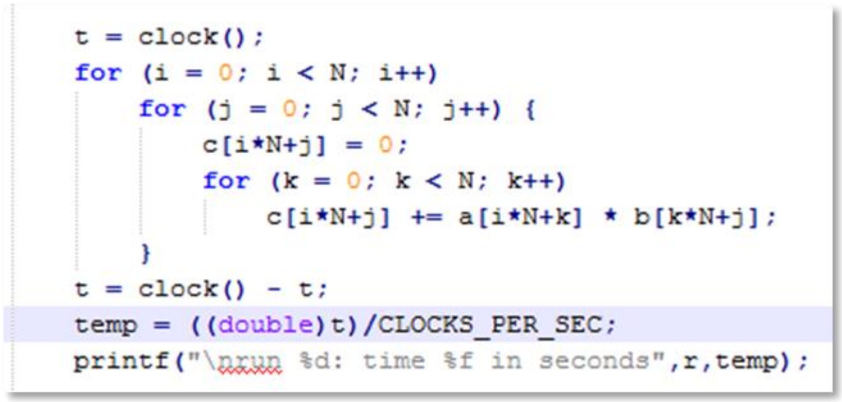
Experimental Configuration

Hardware: A single EC2 instance (c4 2xlarge) with 8 vCPUs and 15 GB RAM was used to carry out below serial implementations.

Compiler Version:

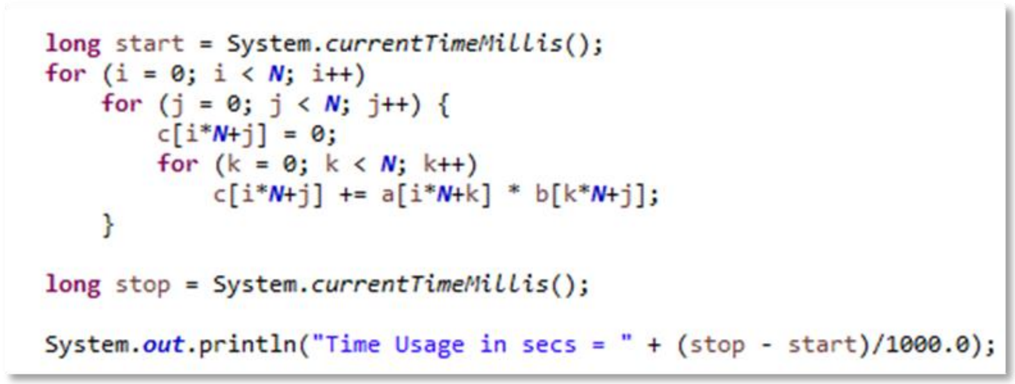
- GCC version 4.8.3 20140911 (Red Hat 4.8.3-9)
- javac 1.8.0_45

Execution details: Both C and Java variants implement the sequential algorithm described in Chapter V on page 46. Figures 6.1 and 6.2 below, illustrate the code snippets for the realizations of the sequential algorithm for C and Java respectively.



```
t = clock();
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
        c[i*N+j] = 0;
        for (k = 0; k < N; k++)
            c[i*N+j] += a[i*N+k] * b[k*N+j];
    }
t = clock() - t;
temp = ((double)t)/CLOCKS_PER_SEC;
printf("\nrun %d: time %f in seconds",r,temp);
```

Figure 6.1: Code Snippet for Serial Matrix Multiplication Implementation in C



```
long start = System.currentTimeMillis();
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
        c[i*N+j] = 0;
        for (k = 0; k < N; k++)
            c[i*N+j] += a[i*N+k] * b[k*N+j];
    }

long stop = System.currentTimeMillis();

System.out.println("Time Usage in secs = " + (stop - start)/1000.0);
```

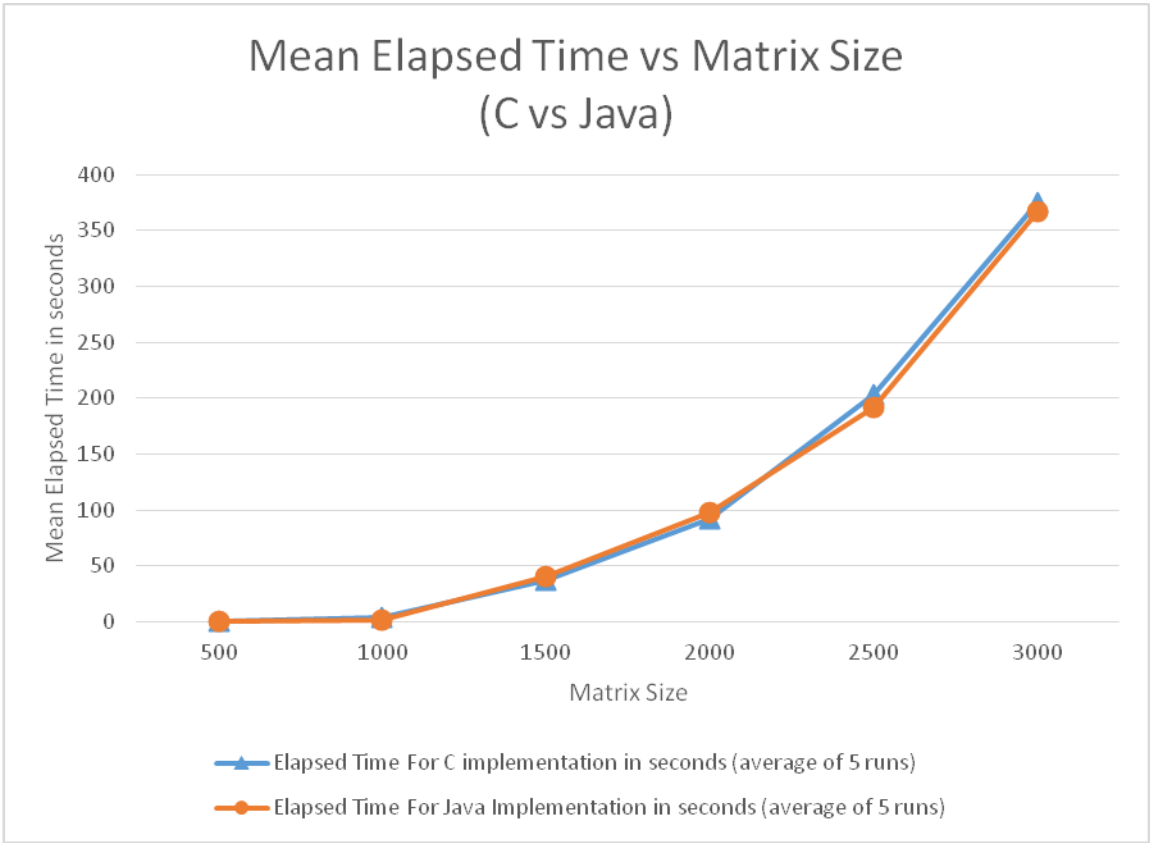
Figure 6.2: Code Snippet for Serial Matrix Multiplication Implementation in Java

Result capture: All the elapsed times below are in seconds and were taken over an average of five consecutive runs. Also, the standard deviation was rounded off to three decimal places.

TABLE 6.1

C vs. JAVA SERIAL IMPLEMENTATION COMPARISON

Matrix Size (N)	**C serial implementation**		**Java serial implementation**	
	Mean Elapsed Time (in secs)	Standard Deviation	Mean Elapsed Time (in secs)	Standard Deviation
500	0.45200	0.004	0.1972	0.002
1000	3.776000	0.005	1.5296	0.004
1500	37.248000	0.064	40.0986	0.074
2000	92.280000	0.071	97.3964	0.062
2500	202.710000	0.078	192.0288	0.082
3000	375.216000	0.078	367.1946	0.091



Graph 6.1: C vs. Java Serial Implementation Comparison

Observation: It appears from the graph above, there are no significant differences between the elapsed times of C and Java serial implementations for matrix multiplication.

Parallel Implementations

In this section we implement parallel algorithms described in Chapter V for both shared and distributed memory architectures. The source code for MPI based implementations in C and Java, can be found in the Appendix A and B respectively. Since, MPI supports both multicore and cluster configurations the same source code was re-used for running both shared and distributed memory use cases. The target execution configuration (multicore or cluster) is passed on to the compiled object at runtime, which then employs appropriate communication to achieve the result. The configuration used and implementation specifics for each use case is described in the relevant sections below.

Shared Memory Parallel (SMP) Implementations

In this section, we implement the shared memory algorithm described in Chapter V on pages 46 through 47. We first present results of the MPI based C implementation using Open MPI, followed by two Java implementations: a) Java Threads and b) MPI based Java implementation using multicore configuration of MPJ Express described in Message Passing section of Chapter IV, and then finally compare all the implementations. The final comparison will help us in summarizing the performance of C vs. Java for the presented parallel implementations in shared memory systems.

C SMP Implementation–Open MPI

Experimental Configuration

Hardware: A single EC2 instance (c4 2xlarge) with 8 vCPUs and 15 GB RAM was used to carry out below MPI based implementation in C using Open MPI.

Compiler Version:

- GCC version 4.8.3 20140911 (Red Hat 4.8.3-9)
- Open MPI 1.6.4

Execution details: The source code can be found in the Appendix A. Below steps describe the sample steps needed to capture the results for a 3000x3000 matrix multiplication with 8 processes using the source code.

- Modify the macro value N (matrix dimension) to 3000 in source code, if not already.
- Compile the source code, say named as ‘mm_mpi.c’, by using mpicc as follows

mpicc –std=gnu99 –o mm mm_mpi.c

- Execute the above compiled object in shared memory configuration as follows with 8 processes: *mpirun –np 8 mm*

  mpicc [59] – Open MPI C wrapper compiler.

  mpirun [59] – Used to execute serial and parallel jobs in Open MPI.

Result capture: All the elapsed times below are in seconds and were taken over an average of five consecutive runs. Also the speed ups and standard deviations are rounded off to three decimal places.

TABLE 6.2

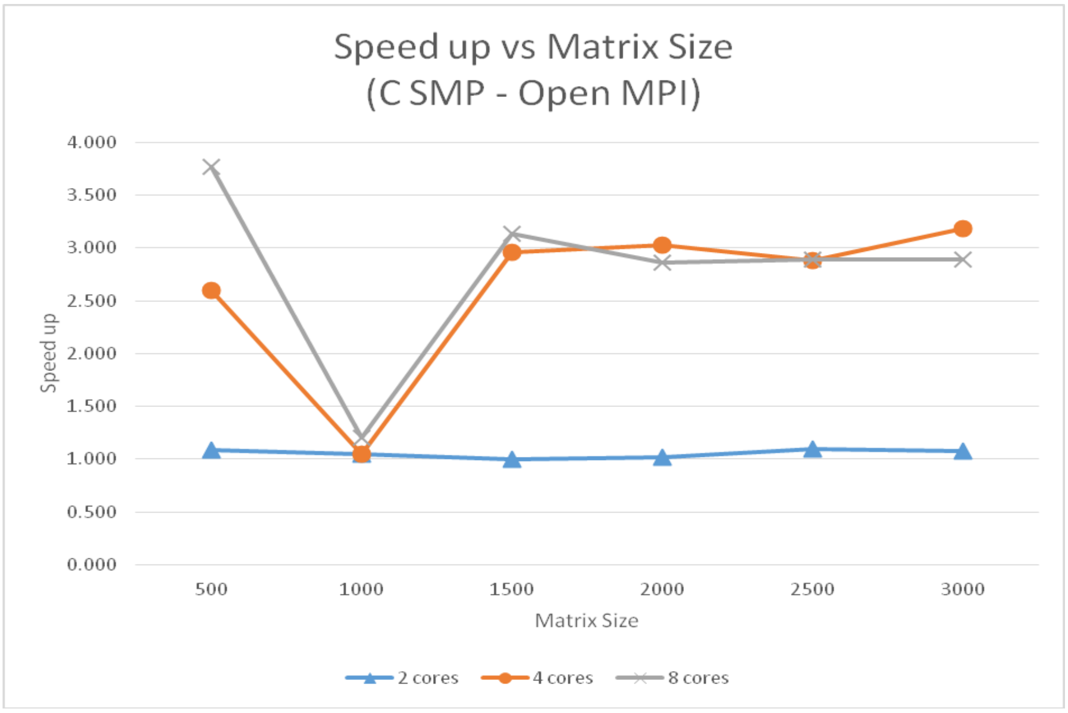
C SMP IMPLEMENTATION—OPEN MPI: MEAN ELAPSED TIME AND STANDARD DEVIATION

Matrix Size (N)	**2 cores**		**4 cores**		**8 cores**	
	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation
500	0.418	0.010	0.174	0.053	0.12	0.000
1000	3.618	0.063	3.618	2.966	3.146	0.037
1500	37.424	0.215	12.588	0.091	11.898	0.062
2000	90.32	0.361	30.472	0.174	32.238	0.937
2500	185.148	1.664	70.264	9.535	70.052	0.490
3000	349.104	2.703	117.928	2.279	130.006	0.925

TABLE 6.3

C SMP IMPLEMENTATION—OPEN MPI: SPEED UP

Matrix Size (N)	**Serial Implementation Mean Elapsed Time (T_s)**	**Speed up (T_s/T_p)**		
		2 cores	4 cores	8 cores
500	0.45200	1.081	2.598	3.767
1000	3.776000	1.044	1.044	1.200
1500	37.248000	0.995	2.959	3.131
2000	92.280000	1.022	3.028	2.862
2500	202.710000	1.095	2.885	2.894
3000	375.216000	1.075	3.182	2.886



Graph 6.2: C SMP Implementation-Open MPI: Speed up vs. Matrix Size

Observation: From the graph above it appears that the speed up is gradually increasing as more cores are added, except for the case of matrix size 1000 with 4 and 8 cores. It might be due to the fact that the communication overhead among the processes was far greater than the time taken for actual multiplication itself, resulting in the above pattern.

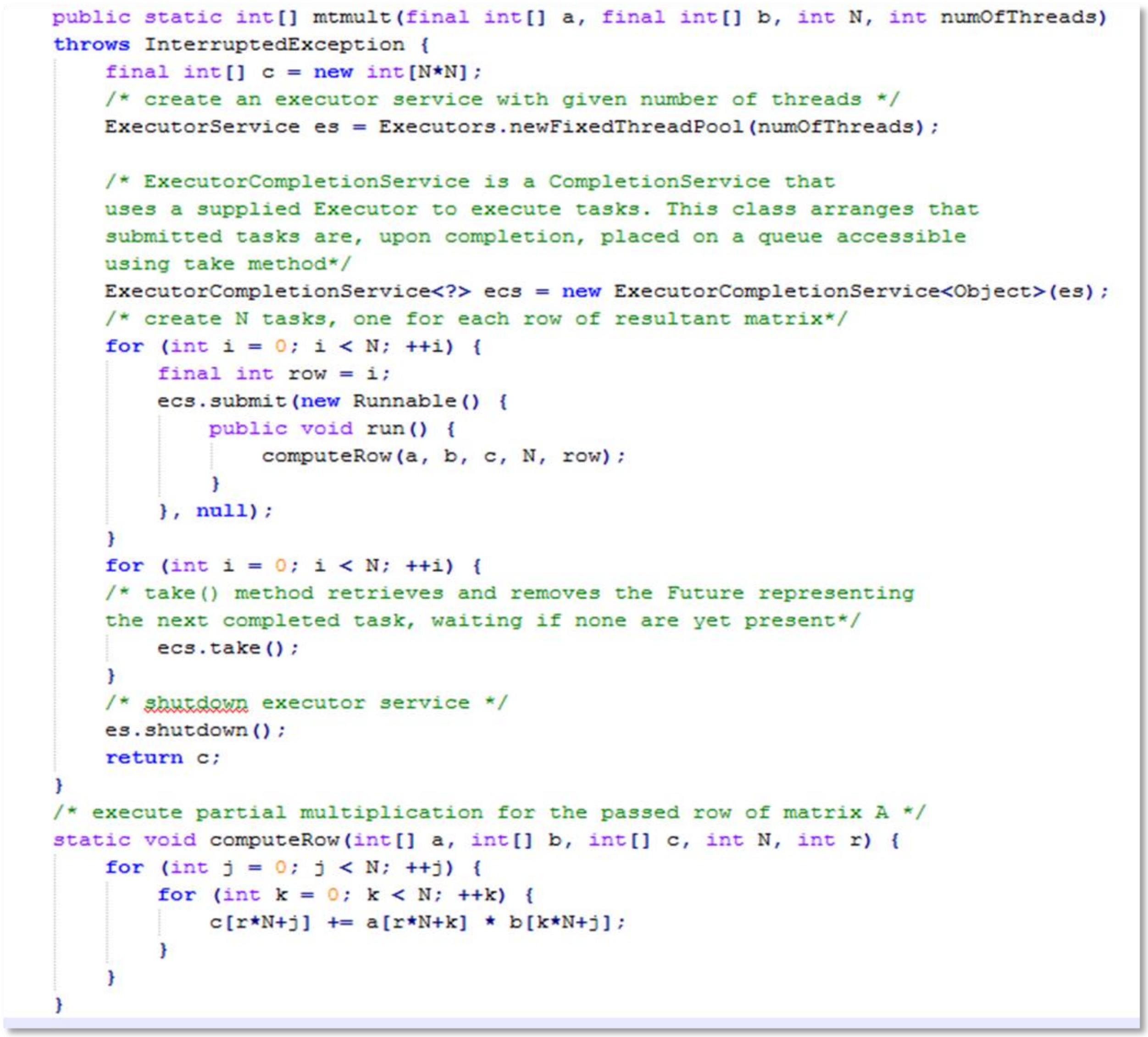
Java SMP Implementation 1–Java Threads

Experimental Configuration

Hardware: A single EC2 instance (c4 2xlarge) with 8 vCPUs, 15 GB RAM was used to carry out below Java Threads implementation.

Compiler version: javac 1.8.0_45

Execution details: In this implementation, we use ExecutorService [18] to create a fixed thread pool for the specified number of threads. This executor service is then passed on to the ExecutorCompletionService [61] to execute partial matrix multiplication tasks. The relevant code snippet is illustrated in Figure 6.3 below. Each row of the resultant matrix is computed by performing partial matrix multiplication of the related row of matrix ‘a’ and entire matrix ‘b’, this is done by using the computeRow() method shown in the code snippet. On a high level, this is achieved by creating one task per row of the resultant matrix and submitting it to the executor service. The executor service then executes these tasks using the fixed thread pool, created earlier. We then wait until all the tasks complete execution using task() method. On completion of all tasks the resultant matrix will be completely filled with partial results. For instance, to multiply two 4x4 matrices using two threads, the below code creates 4 tasks, one for each row of resultant matrix, and then submits each of them to the executor service which in turn distributes the submitted tasks among its pool of two threads. Both threads, on completion of each task, fill in the related partial results for the resultant matrix.



```
public static int[] mtmult(final int[] a, final int[] b, int N, int numOfThreads)
throws InterruptedException {
    final int[] c = new int[N*N];
    /* create an executor service with given number of threads */
    ExecutorService es = Executors.newFixedThreadPool(numOfThreads);

    /* ExecutorCompletionService is a CompletionService that
    uses a supplied Executor to execute tasks. This class arranges that
    submitted tasks are, upon completion, placed on a queue accessible
    using take method*/
    ExecutorCompletionService<?> ecs = new ExecutorCompletionService<Object>(es);
    /* create N tasks, one for each row of resultant matrix*/
    for (int i = 0; i < N; ++i) {
        final int row = i;
        ecs.submit(new Runnable() {
            public void run() {
                computeRow(a, b, c, N, row);
            }
        }, null);
    }
    for (int i = 0; i < N; ++i) {
    /* take() method retrieves and removes the Future representing
    the next completed task, waiting if none are yet present*/
        ecs.take();
    }
    /* shutdown executor service */
    es.shutdown();
    return c;
}
/* execute partial multiplication for the passed row of matrix A */
static void computeRow(int[] a, int[] b, int[] c, int N, int r) {
    for (int j = 0; j < N; ++j) {
        for (int k = 0; k < N; ++k) {
            c[r*N+j] += a[r*N+k] * b[k*N+j];
        }
    }
}
```

Figure 6.3: Code Snippet for SMP Matrix Multiplication Using Java Threads

Result capture: All the times below are in seconds and were taken over an average of five consecutive runs. Also the speed ups and standard deviations are rounded off to three decimal places.

TABLE 6.4

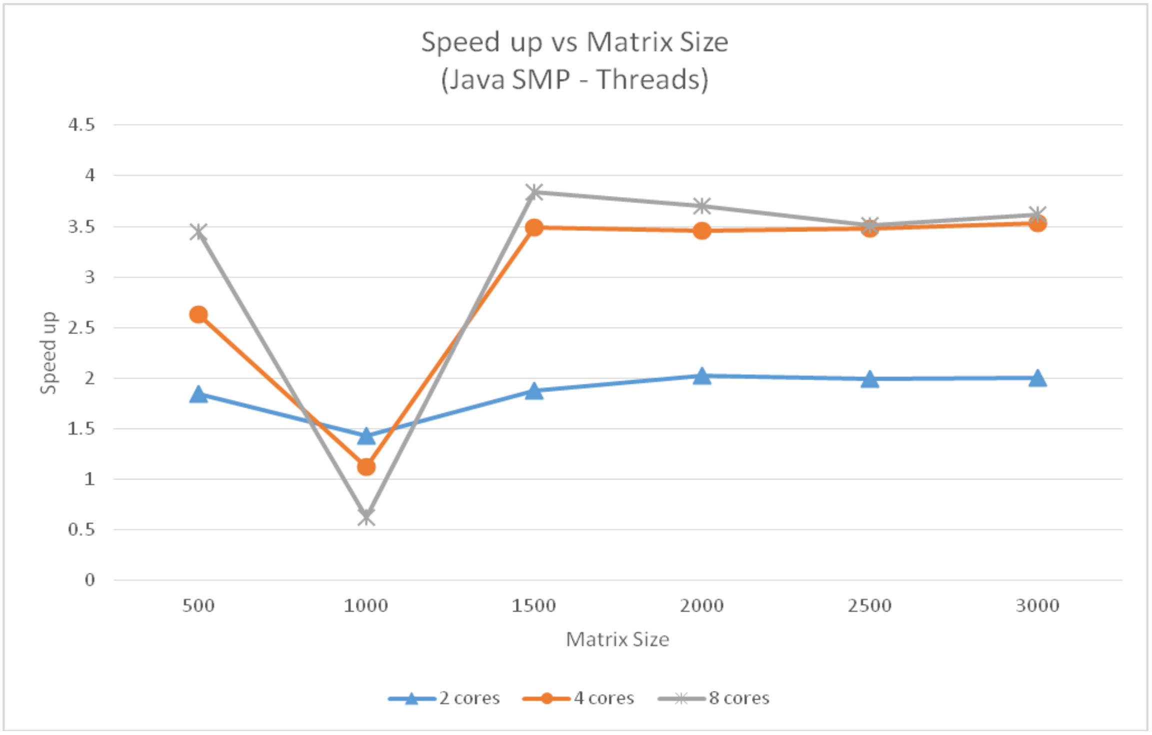
JAVA SMP IMPLEMENTATION—JAVA THREADS: MEAN ELAPSED TIME AND STANDARD DEVIATION

Matrix Size (N)	**2 cores**		**4 cores**		**8 cores**	
	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation
500	0.1068	0.003	0.075	0.253	0.0572	0.060
1000	1.0674	0.063	1.3664	2.766	2.4732	0.035
1500	21.4136	0.215	11.491	0.157	10.4494	0.532
2000	48.1028	0.361	28.155	0.474	26.2938	0.967
2500	96.2288	1.362	55.1692	8.323	54.6918	0.890
3000	183.8028	2.112	104.0308	2.288	101.5042	1.623

TABLE 6.5

JAVA SMP IMPLEMENTATION–JAVA THREADS: SPEED UP

Matrix Size (N)	**Serial Implementation Mean Elapsed Time (T_s)**	**Speed up (T_s/T_p)**		
		2 cores	4 cores	8 cores
500	0.1972	1.846	2.629	3.448
1000	1.5296	1.433	1.119	0.618
1500	40.0986	1.873	3.490	3.837
2000	97.3964	2.025	3.459	3.704
2500	192.0288	1.996	3.481	3.511
3000	367.1946	1.998	3.530	3.618



Graph 6.3: Java SMP Implementation—Java Threads

Observation: From the graph above it appears that the speed up is gradually increasing as more cores are added, except for the case of matrix size 1000 with 4 and 8 cores. It might be due to the fact that the communication overhead among the cores was far greater than the time taken for actual multiplication itself, resulting in the above pattern.

Java SMP Implementation 2–MPJ Express Multicore Configuration

Experimental Configuration

Hardware: A single EC2 instance (c4 2xlarge) with 8 vCPUs, 15 GB RAM was used to carry out below Java implementation using MPJ Express multicore configuration.

Compiler version:

- javac 1.8.0_45

Execution details: The source code can be found in the Appendix B. Below commands are executed in order to compile and run the source code, say named as MM_MPI.java, in multicore configuration and capture the results for a 3000x3000 matrix multiplication with 8 processes.

Compile: *javac -cp .:$MPJ_HOME/lib/mpj.jar MM_MPI.java*

Run: *mpjrun.sh –np 8 MM_MPI 3000*

'mpjrun.sh' is a wrapper script that is used to execute the compiled objects in MPJ Express.

Result capture: All the times below are in seconds and were taken over an average of five consecutive runs. Also the speed ups and standard deviations are rounded off to three decimal places.

TABLE 6.6

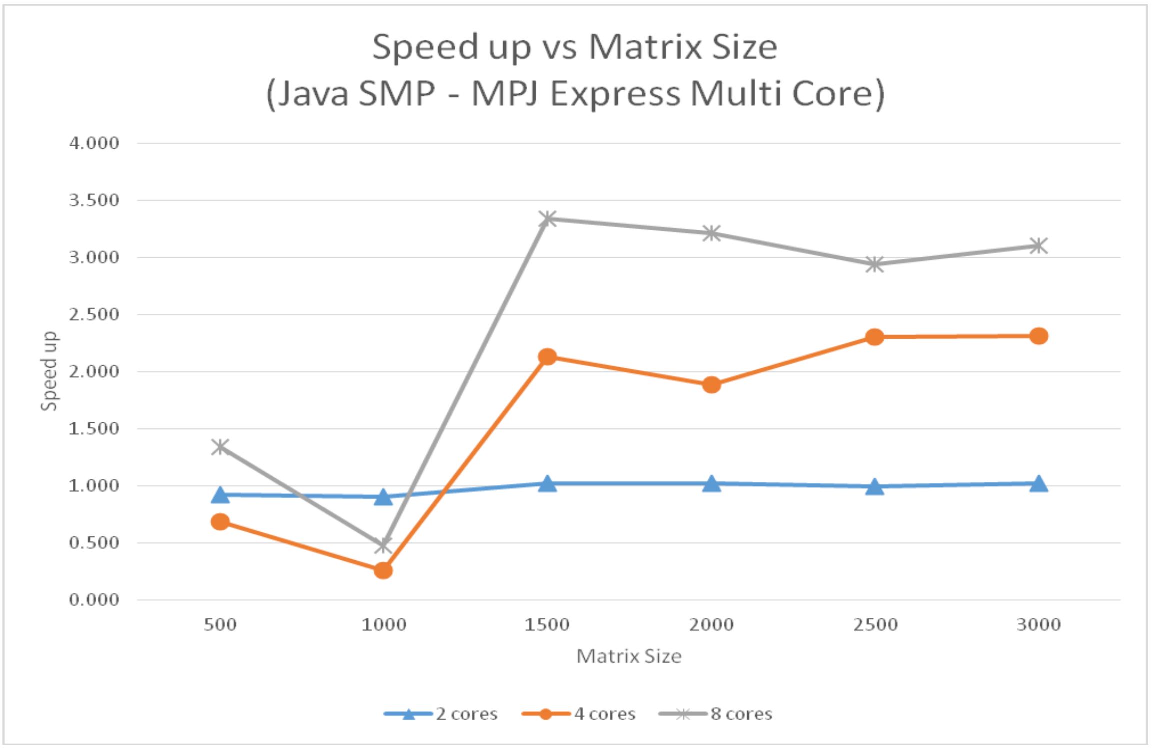
JAVA SMP-MPJ EXPRESS MULTICORE CONFIGURATION: MEAN ELAPSED TIME AND STANDARD DEVIATION

Matrix Size (N)	**2 cores**		**4 cores**		**8 cores**	
	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation
500	0.2136	0.004	0.2882	0.342	0.147	0.007
1000	1.6886	0.024	6.0412	2.697	3.1962	0.016
1500	39.1774	0.105	18.8722	6.851	12.0134	0.140
2000	95.1776	0.412	51.6572	15.962	30.295	1.660
2500	194.1812	0.618	83.4154	23.848	65.279	2.954
3000	359.717	3.986	158.9412	57.002	118.444	5.456

TABLE 6.7

JAVA SMP-MPJ EXPRESS MULTICORE CONFIGURATION: SPEED UP

Matrix Size (N)	**Serial Implementation Mean Elapsed Time (T_s)**	**Speed up (T_s/T_p)**		
		2 cores	4 cores	8 cores
500	0.1972	0.923	0.684	1.341
1000	1.5296	0.906	0.253	0.479
1500	40.0986	1.024	2.125	3.338
2000	97.3964	1.023	1.885	3.215
2500	192.0288	0.989	2.302	2.942
3000	367.1946	1.021	2.310	3.100



Graph 6.4: Java SMP Implementation–MPJ Express Multicore Configuration

Observation: From the graph above it appears that the speed up is gradually increasing as more cores are added, except for one case of matrix size 1000 with 4 and 8 cores. It might be due to the fact that the communication overhead among the cores was far greater than the time taken for actual multiplication itself, resulting in the above pattern.

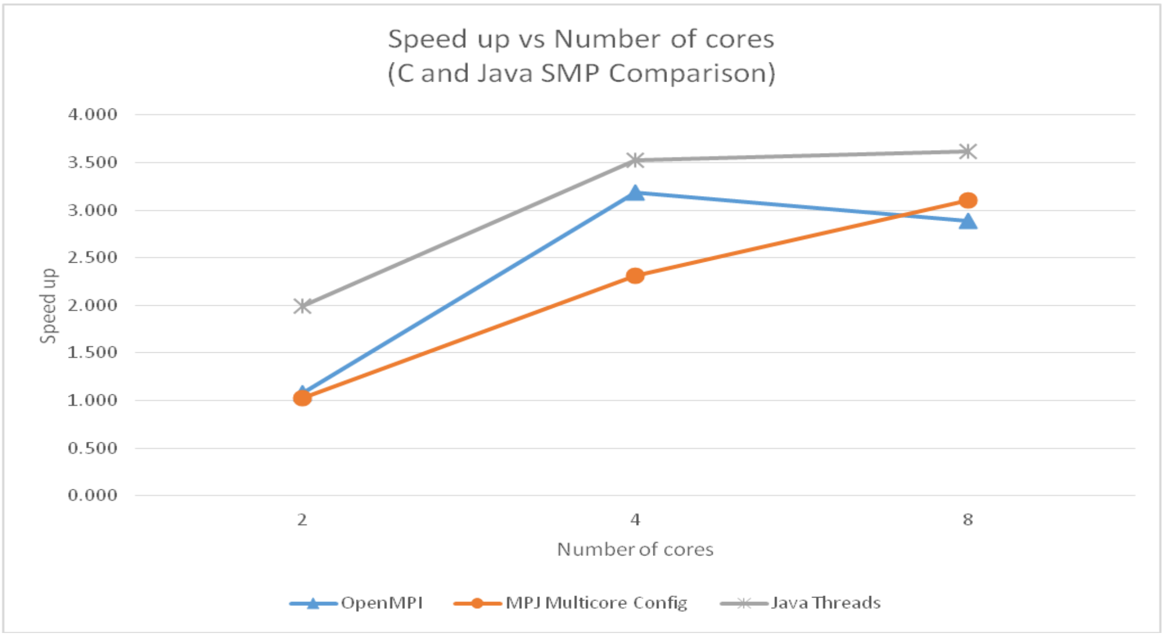
Comparison of C and Java Shared Memory Implementations

This comparison summarizes all the above SMP implemenations of both C and Java. The below graph is obtained by plotting the speed ups of a 3000x3000 matrix multiplication of each SMP implementation across number of cores.

TABLE 6.8

C vs. JAVA SMP IMPLEMENTATION COMPARISON

SMP Implementation	**Speed Up**		
	2 cores	**4 cores**	**8 cores**
Open MPI–C	1.087	3.215	7.323
MPJ Express Multicore Configuration—Java	1.01	2.965	6.532
Java Threads	1.008	2.999	6.074



Graph 6.5: C vs. Java SMP Implementation Comparison

Observation: Java Threads seem to have a better speed up followed by Open MPI - C implementation and then MPJ Express multicore Java implementation.

Distributed Memory Parallel (DMP) Implementations

In this section, we implement the distributed memory algorithm described in Chapter V on pages 47 through 51. We first present results of the MPI based C implementation using Open MPI in cluster configuration, followed by three Java implementations using the following cluster configurations of MPJ Express, described in Message Passing section of Chapter IV: a) niodev b) hybdev and c) native, and then finally compare all the implementations. The final comparison will help us in summarizing the performance of C vs Java for the presented parallel implementations in distributed memory systems.

Pre-Requisites for Cluster Configuration

- All the nodes in the cluster should be reachable via ssh from each other.

- A file with all the hostnames of the cluster should exist. This file will be used by MPI based implementations to invoke processes on the listed nodes in the cluster. We assume this file is named as 'machines' for all the below implementations.

C DMP Implementation–Open MPI

Experimental Configuration

Hardware: A Virtual Private Cluster (VPC) of three EC2 instance (c4 2xlarge) with 8 vCPUs, 15 GB RAM was used to carry out below distributed C implementation using Open MPI.

Compiler version:

- GCC version 4.8.3 20140911 (Red Hat 4.8.3-9)
- Open MPI 1.6.4

Execution detail: The source code can be found in the Appendix A. Below steps describe the sample steps needed to capture the results for a 3000x3000 matrix multiplication with 8 processes using the source code in Appendix.

- Modify the macro value N (matrix dimension) to 3000 in source code, if not already.
- Compile the source code, say named as 'mm_mpi.c', by using mpicc as follows

  mpicc –std=gnu99 –o mm mm_mpi.c
- Execute the above compiled object in cluster configuration with 8 processes, using mpirun as follows

  mpirun –machinefile machines –np 8 mm

mpicc [59]–Open MPI C wrapper compiler.

mpirun [59]–Used to execute serial and parallel jobs in Open MPI.

Note: When running code in cluster configuration using above command, MPI decides the best configuration to run the program. It may or may not use all the required cores of the same machine in cases when number of processes is less than number of cores.

Result capture: All the elapsed times below are in seconds and were taken over an average of five consecutive runs. Also the speed ups and standard deviations are rounded off to three decimal places.

TABLE 6.9a

C DMP IMPLEMENTATION—OPEN MPI: MEAN ELAPSED TIME AND STANDARD DEVIATION

Matrix Size (N)	**2 cores**		**4 cores**		**8 cores**	
	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation
500	0.42	0.000	0.15	0.000	0.082	0.015
1000	3.618	0.027	1.4	0.365	0.578	0.010
1500	37.29	0.561	12.864	0.703	5.642	0.160
2000	88.98	0.308	30.592	0.975	20.572	7.972
2500	180.734	0.964	62.288	0.731	27.372	0.968
3000	345.324	3.917	116.69	0.759	51.24	1.662

TABLE 6.9b

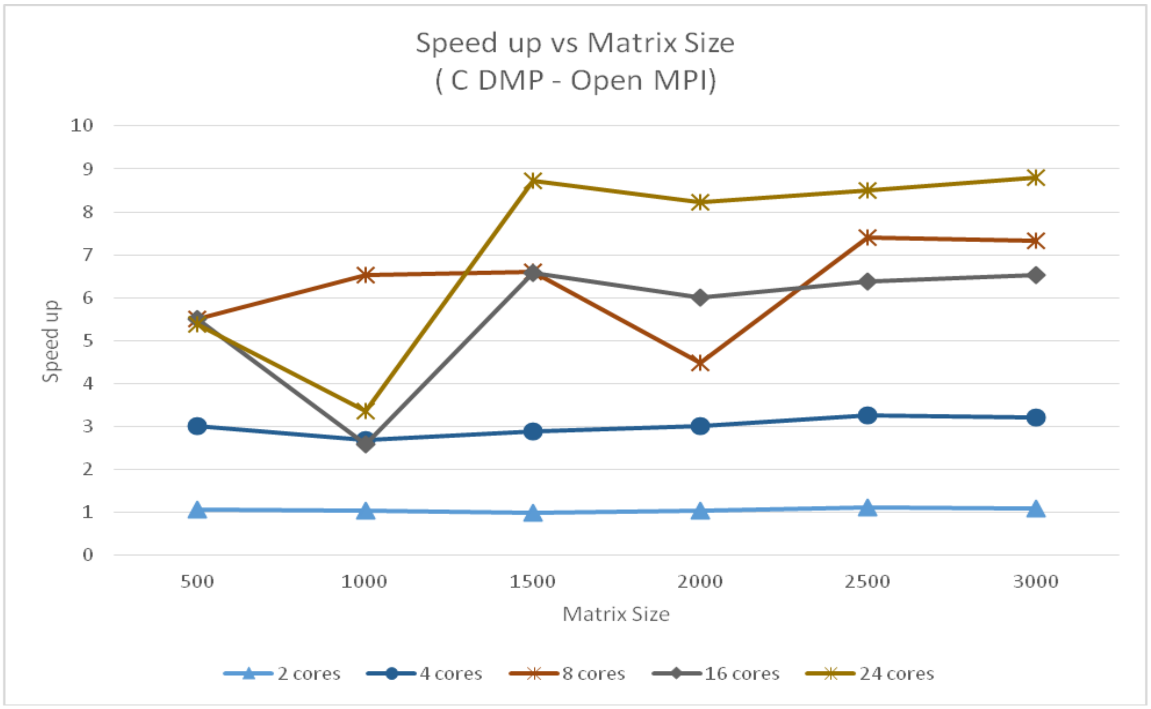
C DMP IMPLEMENTATION—OPEN MPI: MEAN ELAPSED TIME AND STANDARD DEVIATION

Matrix Size (N)	**16 cores**		**24 cores**	
	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation
500	0.082	0.004	0.084	0.005
1000	1.458	0.031	1.126	0.037
1500	5.66	0.083	4.27	0.131
2000	15.336	0.197	11.202	0.152
2500	31.812	0.713	23.868	0.432
3000	57.406	5.372	42.608	0.364

TABLE 6.10

C DMP IMPLEMENTATION—OPEN MPI: SPEED UP

Matrix Size (N)	**Speed Up**				
	2 cores	4 cores	8 cores	16 cores	24 cores
500	1.076	3.013	5.512	5.512	5.381
1000	1.044	2.697	6.533	2.590	3.353
1500	0.999	2.896	6.602	6.581	8.723
2000	1.037	3.016	4.486	6.017	8.238
2500	1.122	3.254	7.406	6.372	8.493
3000	1.087	3.215	7.323	6.536	8.806



Graph 6.6: C DMP Implementation–Open MPI

Observation: For 3000x3000 matrix size, we were able to achieve 3x speed up (around 9) when compared to the speed up (around 3) of C SMP Open MPI implementation by using all 24 cores available in the cluster.

Java DMP Implementation 1—MPJ Express *niodev* Cluster Configuration

Experimental Configuration

Hardware: A Virtual Private Cluster (VPC) of three EC2 instance (c4 2xlarge) with 8 vCPUs, 15 GB RAM was used to carry out below distributed Java implementation using MPJ Express *niodev* cluster configuration.

Compiler version:

- MPJ Express v0_44

Result capture: All the elapsed times below are in seconds and were taken over an average of five consecutive runs. Also the speed ups and standard deviations are rounded off to three decimal places.

Execution details: In this particular implementation we use the Java New I/O (NIO) device driver provided by MPJ Express, known as *niodev* [8]. This was described in Message Passing section of Chapter IV. This device uses Ethernet based interconnect for message passing. The source code can be found in the Appendix B. Below commands are executed in order to compile and run the source code, say named as MM_MPI.java, in *niodev* cluster configuration and capture the results for a 3000x3000 matrix multiplication with 8 processes.

a) Compile: *javac -cp .:$MPJ_HOME/lib/mpj.jar MM_MPI.java*
b) If not done already, then start daemons used by MPJ Express to figure out nodes in the cluster by using: m*pjboot machines*
c) Run: *mpjrun.sh –dev niodev –np 8 MM_MPI 3000*

   'mpjrun.sh' is a wrapper script that is used to execute the compiled objects in MPJ Express.

Note: When running code in cluster configuration using above commands, MPI decides the best configuration to run the program. It may or may not use all the required cores of the same machine in cases when number of processes is less than number of cores.

TABLE 6.11a

JAVA DMP - MPJ EXPRESS *niodev* IMPLEMENTATION: MEAN ELAPSED TIME AND STANDARD DEVIATION

Matrix Size (N)	**2 cores**		**4 cores**		**8 cores**	
	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation
500	0.2152	0.002	0.1144	0.023	0.093	0.008
1000	1.5066	0.367	0.6328	0.008	1.1398	1.073
1500	39.4348	0.091	13.634	0.305	6.601	1.066
2000	95.405	0.536	32.1186	0.173	14.1892	0.271
2500	193.7314	1.232	66.5742	0.815	31.895	2.490
3000	363.4274	2.258	123.8556	1.018	56.2118	3.443

TABLE 6.11b

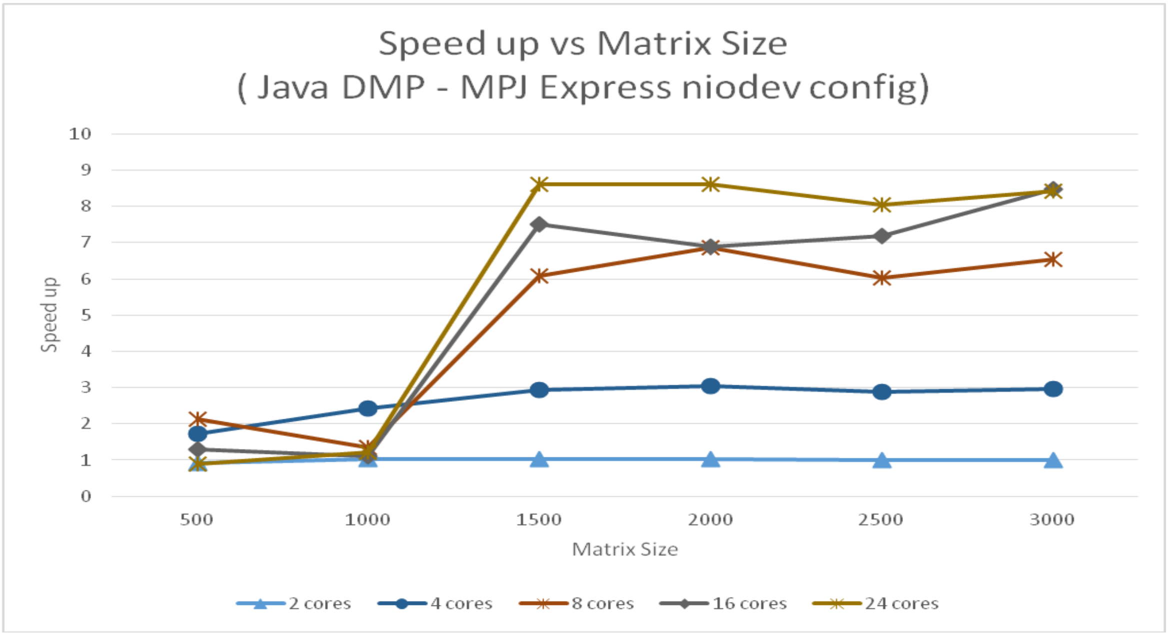
JAVA DMP—MPJ EXPRESS *niodev* IMPLEMENTATION: MEAN ELAPSED TIME AND STANDARD DEVIATION

Matrix Size (N)	**16 cores**		**24 cores**	
	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation
500	0.1528	0.009	0.2196	0.011
1000	1.398	0.209	1.2588	0.109
1500	5.3502	0.388	4.6634	0.072
2000	14.1396	1.447	11.3306	0.162
2500	26.7378	3.048	23.8426	0.409
3000	43.381	2.618	43.6262	0.312

TABLE 6.12

JAVA DMP—MPJ EXPRESS *niodev* IMPLEMENTATION: SPEED UP

Matrix Size (N)	**Speed Up**				
	2 cores	4 cores	8 cores	16 cores	24 cores
500	0.916	1.724	2.120	1.291	0.898
1000	1.015	2.417	1.342	1.094	1.215
1500	1.017	2.941	6.075	7.495	8.599
2000	1.021	3.032	6.864	6.888	8.596
2500	0.991	2.884	6.021	7.182	8.054
3000	1.010	2.965	6.532	8.464	8.417



Graph 6.7: Java DMP Implementation 1—MPJ Express *niodev* Cluster Configuration

Observation: We were able to achieve 3x speed up (around 9) when compared to the speed up (around 3) of Java SMP MPJ Express multicore implementation by using all 24 cores available in the cluster.

Java DMP Implementation 2—MPJ Express *hybdev* Cluster Configuration

Experimental Configuration

Hardware: A Virtual Private Cluster (VPC) of three EC2 instance (c4 2xlarge) with 8 vCPUs, 15 GB RAM was used to carry out below distributed Java implementation using MPJ Express *hybdev* cluster configuration.

Compiler version:

- MPJ Express v0_44

Result capture: All the elapsed times below are in seconds and were taken over an average of five consecutive runs. Also the speed ups and standard deviations are rounded off to three decimal places.

Execution details: In this implementation we use the *hybdev* device driver, which is used to execute parallel java applications on clusters of multi-core machines [8]. This was described in Message Passing section of Chapter IV. This device uses both multicore configuration (SMP) and *niodev* cluster configuration of MPJ Express for intra node communications and only *niodev* cluster configuration for inter-node communication. The source code can be found in the Appendix B. Below commands are executed in order to compile and run the source code, say named as MM_MPI.java, in *hybdev* cluster configuration and capture the results for a 3000x3000 matrix multiplication with 8 processes.

a) Compile: *javac -cp .:$MPJ_HOME/lib/mpj.jar MM_MPI.java*
b) If not done already, then start daemons used by MPJ Express to figure out nodes in the cluster by using: m*pjboot machines*
c) Run: *mpjrun.sh –dev hybdev –np 8 MM_MPI 3000*

‘mpjrun.sh’ is a wrapper script that is used to execute the compiled objects in MPJ Express.

Note: When running code in cluster configuration using above commands, MPI decides the best configuration to run the program. It may or may not use all the required cores of the same machine in cases when number of processes is less than number of cores.

TABLE 6.13a

JAVA DMP—MPJ EXPRESS *hybdev* IMPLEMENTATION: MEAN ELAPSED TIME AND STANDARD DEVIATION

Matrix Size (N)	**2 cores**		**4 cores**		**8 cores**	
	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation
500	0.2184	0.002	0.0976	0.002	0.0878	0.010
1000	1.6954	0.025	0.6192	0.013	2.0714	1.393
1500	39.4574	0.045	13.6918	0.021	8.439	3.075
2000	95.6644	0.226	32.4606	0.240	21.8438	8.144
2500	191.9478	1.152	67.4758	0.786	35.8412	4.454
3000	364.1952	3.398	122.428	1.708	60.449	11.454

TABLE 6.13b

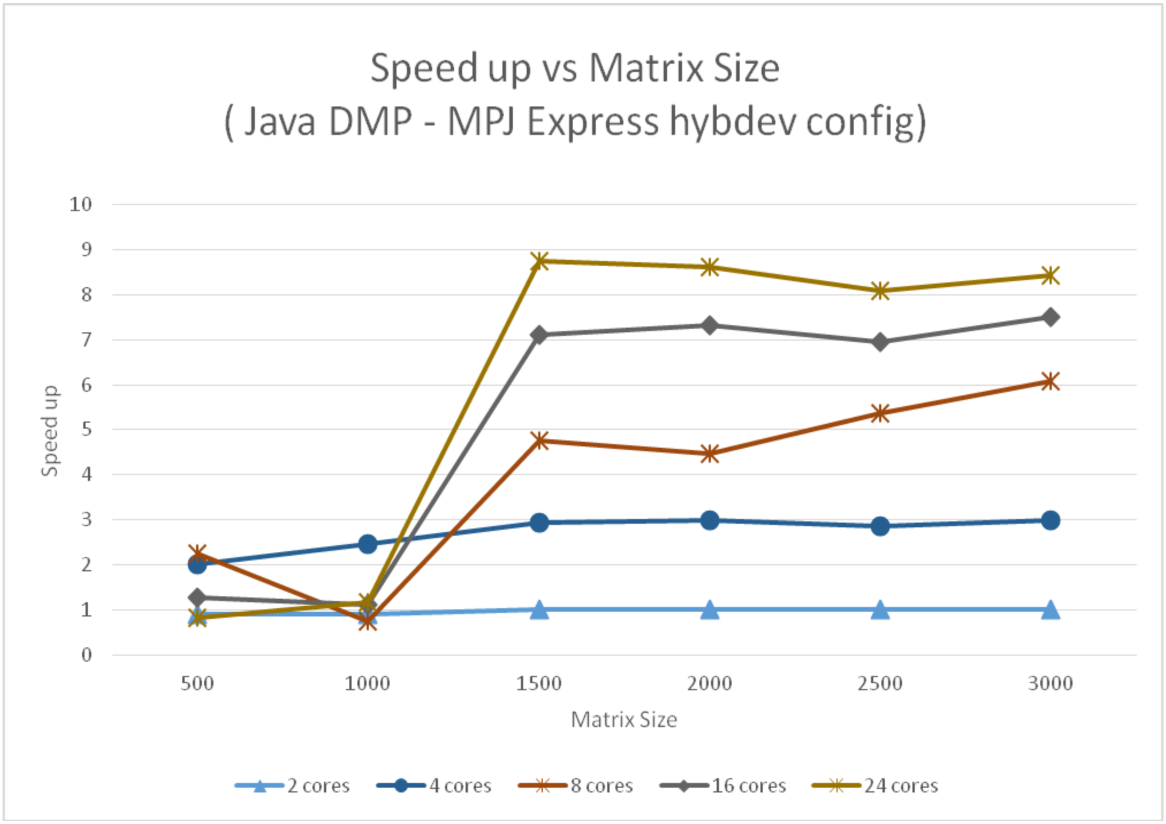
JAVA DMP—MPJ EXPRESS *hybdev* IMPLEMENTATION: MEAN ELAPSED TIME AND STANDARD DEVIATION

Matrix Size (N)	**16 cores**		**24 cores**	
	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation
500	0.155	0.018	0.2372	0.005
1000	1.3834	0.314	1.3142	0.080
1500	5.6426	0.049	4.5804	0.070
2000	13.277	1.879	11.2816	0.217
2500	27.5876	3.789	23.6964	0.210
3000	48.9448	6.691	43.5074	0.269

TABLE 6.14

JAVA DMP—MPJ EXPRESS *hybdev* IMPLEMENTATION: SPEED UP

Matrix Size (N)	**Speed Up**				
	2 cores	4 cores	8 cores	16 cores	24 cores
500	0.903	2.020	2.246	1.272	0.831
1000	0.902	2.470	0.738	1.106	1.164
1500	1.016	2.929	4.752	7.106	8.754
2000	1.018	3.000	4.459	7.336	8.633
2500	1.000	2.846	5.358	6.961	8.104
3000	1.008	2.999	6.074	7.502	8.440



Graph 6.8: Java DMP Implementation 2—MPJ Express *hybdev* Cluster Configuration

Observation: We were able to achieve 3x speed up (around 9) when compared to the speed up (around 3) of Java SMP MPJ Express multicore implementation by using all 24 cores available in the cluster.

Java DMP Implementation 3—MPJ Express *native* Cluster Configuration

Experimental Configuration

Hardware: A Virtual Private Cluster (VPC) of three EC2 instance (c4 2xlarge) with 8 vCPUs, 15 GB RAM was used to carry out below distributed Java implementation using MPJ Express *native* cluster configuration.

Compiler version:

- MPJ Express v0_44
- Open MPI 1.6.4

Result capture: All the elapsed times below are in seconds and were taken over an average of five consecutive runs. Also the speed ups and standard deviations are rounded off to three decimal places.

Execution details: In this implementation we use the *native* device driver of MPJ Express, which is used to execute parallel java applications using a native MPI implementation for communication [8]. This was described in Message Passing section of Chapter IV. This is achieved by using a JNI wrapper library. This device also helps in elevating messaging logic, efficient collection algorithms and new interconnects of underlying MPI native library, whenever they are released. The source code can be found in the Appendix B. Below commands are executed in order to compile and run the source code, say named as MM_MPI.java, in *native* cluster configuration and capture the results for a 3000x3000 matrix multiplication with 8 processes.

a) Compile: *javac -cp .:$MPJ_HOME/lib/mpj.jar MM_MPI.java*
b) If not done already, then start daemons used by MPJ Express to figure out nodes in the cluster by using: m*pjboot machines*
c) Run: *mpjrun.sh –dev native –np 8 MM_MPI 3000*

   ‘mpjrun.sh’ is a wrapper script that is used to execute the compiled objects in MPJ Express.

Note: When running code in cluster configuration using above commands, MPI decides the best configuration to run the program. It may or may not use all the required cores of the same machine in cases when number of processes are less than number of cores.

TABLE 6.15a

JAVA DMP—MPJ EXPRESS *native* IMPLEMENTATION: MEAN ELAPSED TIME AND STANDARD DEVIATION

Matrix Size (N)	**2 cores**		**4 cores**		**8 cores**	
	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation
500	0.2224	0.002	0.0996	0.003	0.0914	0.011
1000	1.7082	0.035	0.6802	0.074	0.55	0.379
1500	39.485	0.087	13.675	0.184	6.5218	0.742
2000	95.59	0.262	32.736	0.363	15.3872	2.431
2500	192.812	2.016	66.8442	0.942	38.4086	14.820
3000	364.9928	3.385	124.6632	3.212	65.4312	15.593

TABLE 6.15b

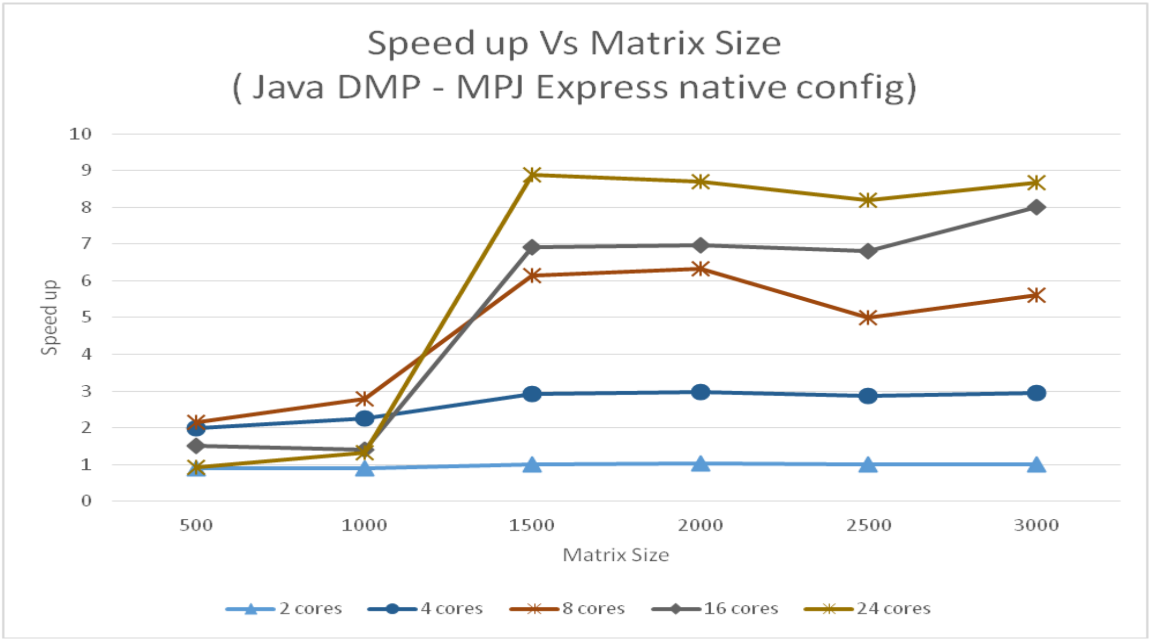
JAVA DMP—MPJ EXPRESS *native* IMPLEMENTATION: MEAN ELAPSED TIME AND STANDARD DEVIATION

Matrix Size (N)	**16 cores**		**24 cores**	
	Mean Elapsed Time (T_p)	Standard Deviation	Mean Elapsed Time (T_p)	Standard Deviation
500	0.1304	0.006	0.2162	0.026
1000	1.0906	0.248	1.1668	0.043
1500	5.802	0.124	4.5146	0.102
2000	13.9896	1.435	11.1792	0.298
2500	28.1658	2.778	23.413	0.588
3000	45.859	4.243	42.2744	0.543

TABLE 6.16

JAVA DMP—MPJ EXPRESS *native* IMPLEMENTATION: SPEED UP

Matrix Size (N)	**Speed Up**				
	2 cores	4 cores	8 cores	16 cores	24 cores
500	0.887	1.980	2.158	1.512	0.912
1000	0.895	2.249	2.781	1.403	1.311
1500	1.016	2.932	6.148	6.911	8.882
2000	1.019	2.975	6.330	6.962	8.712
2500	0.996	2.873	5.000	6.818	8.202
3000	1.006	2.945	5.612	8.007	8.686



Graph 6.9: Java DMP Implementation 3—MPJ Express *native* Cluster Configuration

Observation: We were able to achieve 3x speed up (around 9) when compared to the speed up (around 3) of Java SMP MPJ Express multicore implementation by using all 24 cores available in the cluster.

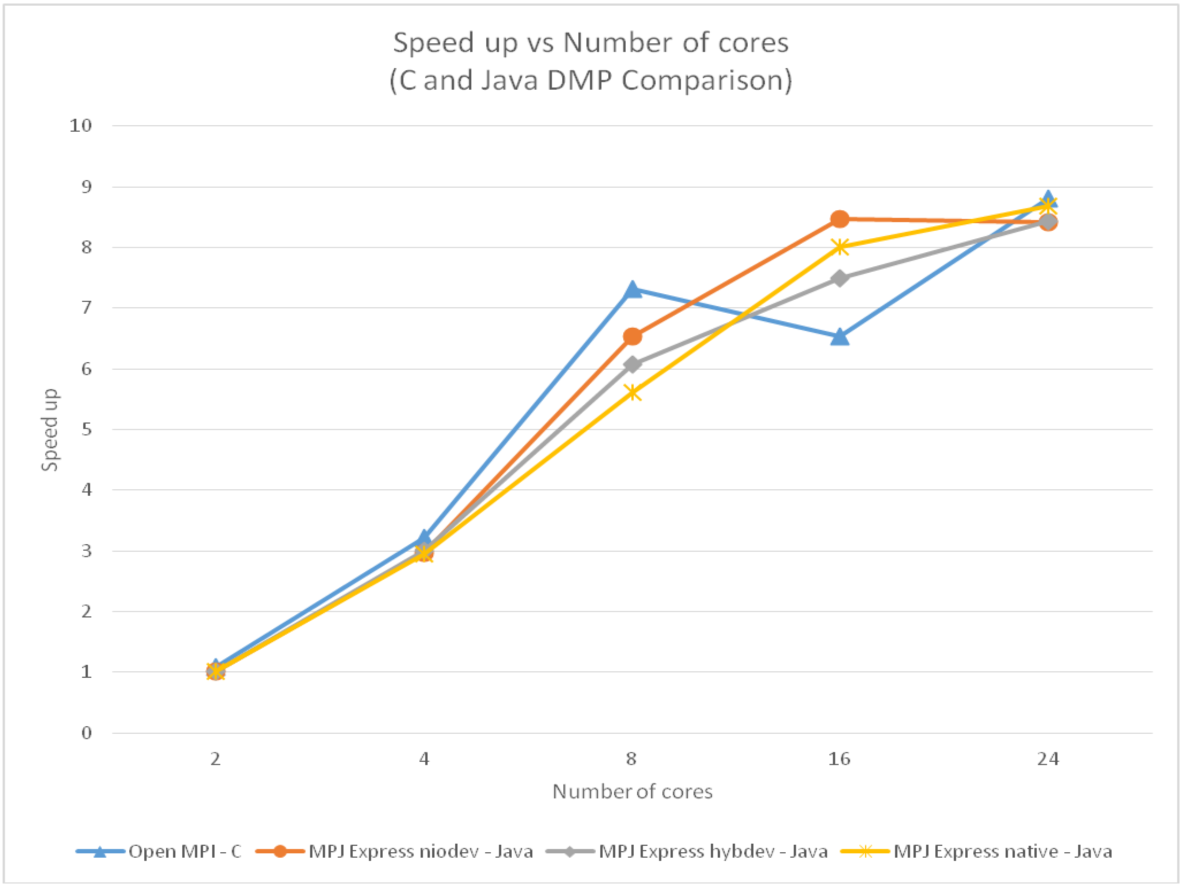
Comparison of C and Java Distributed Memory Implementations

This comparison summarizes all the above DMP implemenations of both C and Java. The below graph is obtained by plotting the speed ups of a 3000x3000 matrix multiplication of each DMP implementation across number of cores.

TABLE 6.17

C vs. JAVA DMP IMPLEMENTATION COMPARISON

Cluster Implementation	**Speed Up**				
	2 cores	4 cores	8 cores	16 cores	24 cores
Open MPI – C	1.087	3.215	7.323	6.536	8.806
MPJ Express *niodev* - Java	1.01	2.965	6.532	8.464	8.417
MPJ Express *hybdev* - Java	1.008	2.999	6.074	7.502	8.44
MPJ Express *native* – Java	1.006	2.945	5.612	8.007	8.686



Graph 6.10: C vs. Java DMP Implementation Comparison

Observation: From the above graph, it appears that all the implementations are really close to each other in terms of speed ups achieved from cluster configuration. Open MPI – C implementation appears to be, almost always, slightly better than rest. Following Open MPI – C implementation very closely, *niodev* implementation of MPJ Express seems to be next with *native* and *hybdev* implementations of the same being slowest.

Chapter VII

CONCLUSION

In this paper, we discussed the current state of Java for HPC, both for shared and distributed memory programming models, focusing mostly on current projects. These projects are the result of the continuous interest in the use of Java for HPC, from when it was put forth by the Java Grande Forum in 1998 [12]. This paper also presented the performance evaluation of current Java HPC solutions and research developments on shared memory environments and distributed multi-core hybrid clusters. The analysis of the results suggest that Java can achieve near similar performance to natively compiled languages, both for sequential and parallel applications, thus making it a viable alternative for HPC programming. In fact, the performance overhead that Java may impose is a reasonable trade-off for the appealing features that this language provides for parallel programming multi-core architectures. Furthermore, the recent advances in the efficient support of Java communications on shared memory and low-latency networks are bridging the performance gap between Java and more traditional HPC languages. Finally, the active research efforts in this area are expected to bring in new developments that will continue increasing the benefits of the adoption of Java for HPC.

Future Work

The current IT technologies have a strong need for scaling up the high-performance analysis to large-scale datasets. In the past few years, with continuous improvements in JVM performance, Java gained popularity in processing “big data” mostly with Apache big data stack [62][63][64]–a collection of open source frameworks dealing with enormous volumes of data, which includes several popular systems such as Hadoop, Hadoop Distributed File System

(HDFS), and Apache Spark [65]. This increasing interest in Java for big data, is also demonstrated by the recent efforts of Open MPI, one of the main open source MPI projects, that has announced its Java interface motivated by a request from the Hadoop community. This Java support could benefit from a large community of users and developers, and from the highly optimized MPI support of Open MPI. Cheptsov in [62], has explored the feasibility of executing big data applications with the traditional HPC infrastructure. For potential future work, this research can possibly be extended to analyze performance of Java based Grid frameworks like Apache Spark [65], Apache Storm [66] or Akka [67] etc. to expose the bottlenecks and identify potential improvements in JVM to further improve efficiency and support for big data and HPC through these platforms.

Also, due to the unavailability of actual physical servers, this research was conducted on virtual machines. An extension of this research may be to conduct the same research on physical hardware and confirm the results. Another possibility might be to extend this research to include the UC Berkeley parallel implementation of Java, named Titanium [68] which was developed to support HPC computing needs on large scale multiprocessors and distributed memory clusters.

References

[1] A. Kaminsky, "Building Parallel Programs: SMPs, Clusters, and Java," *Cengage Course Technology*, 2010, ISBN 1-4239-0198-3.

[2] B. Barney, "Introduction to Parallel Computing." *Lawrence Livermore National Laboratory*. Available: https://computing.llnl.gov/tutorials/parallel_comp/

[3] X. He, "Thesis: Improved Parallel Java Cluster Middleware," *Rochester Institute of Technology,* June 2010.

[4] Oracle Java Specification Documentation: *Understanding Basic Multithreading Concepts*. Available: https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032e/index.html

[5] Oracle Java Specification Documentation: *All About Sockets*. Available: https://docs.oracle.com/javase/tutorial/networking/sockets/index.html

[6] Sun Microsystems, Inc. "Java Remote Method Invocation Specification," in *Technical report, Sun Microsystems, Inc.*, October 1998.

[7] M. Baker, B. Carpenter, G. Fox, S. Hoon Ko, and S. Lim. "mpiJava: An Object-Oriented Java interface to MPI," in *Proceedings International Workshop on Java for Parallel and Distributed Computing,* April 1999.

[8] M. Baker, B. Carpenter, and A. Shafi, "MPJ Express: Towards Thread Safe Java HPC," in *IEEE International Conference on Cluster Computing*, September 2006.

[9] G. L. Taboada, J. Touriño, and R. Doallo, "FastMPJ: a scalable and efficient Java message-passing library," in *The Journal of Supercomputing,* April 2012.

[10] M. Ashworth, "New Language on the Block: Java for High-Performance Computing?," in *ERCIM News Number 36*, January 1999.

[11] J. Dongarra, D. Gannon, G. Fox, and K. Kennedy, "The Impact of Multicore on Computational Science Software," in *CTWatch Quarterly*, February 2007.

[12] Java Grande Forum, "Making Java Work for High-End Computing," in *Java Grande Forum Panel, Supercomputing*, November 1998.

[13] Oracle Java Specification Documentation: *Processes and Threads.* Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html

[14] M. Williams, “Java SE 8: Lambda Quick Start—Runnable Lambda.” Available: http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html#section2

[15] J. Ponge, “Fork and Join: Java Can Excel at Painless Parallel Programming Too!.” *In Oracle Technetwork*, July 2011. Available: http://www.oracle.com/technetwork/articles/java/fork-join-422606.html

[16] Oracle Java Specification Documentation: *Executors*. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html

[17] Oracle Java Specification Documentation: *Executors JDK*. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html

[18] Oracle Java Specification Documentation: *Interface ExecutorService.* Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html

[19] Oracle Java Specification Documentation: *Callable*. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html

[20] Oracle Java Specification Documentation: *Future*. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html

[21] Oracle Java Specification Documentation: *Threads and Locks*. Available: https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html

[22] Oracle Java Specification Documentation: *Locks*. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html

[23] Oracle Java Specification Documentation: *ReentrantLock*. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html

[24] Oracle Java Specification Documentation: *ReadWriteLock*. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReadWriteLock.html

[25] Oracle Java Specification Documentation: *StampedLock*. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html

[26] Oracle Java Specification Documentation: *Atomic*. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html

[27] B. Goetz, “Java theory and practice: Going atomic,” *In IBM Developer Works*, November 2004.

[28] Oracle Java Tutorials: *Atomic Variables*. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/atomicvars.html

[29] Oracle Java Specification Documentation: *Interface IntStream*. Available: https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html#range-int-int-

[30] Oracle Java Specification Documentation: *Atomic Integer*. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html#incrementAndGet--

[31] Oracle Java Specification Documentation: *ConcurrentMap.* Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.html

[32] Oracle Java Specification Documentation: *ConcurrentHashMap*. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html

[33] G. L. Taboada, J. Touri~no, and R. Doallo, “Java Fast Sockets: Enabling High-speed Java Communications on High Performance Clusters,” *In Computer Communications*, November 2008.

[34] Oracle Java Specification Documentation: “Java Remote Method Invocation, Distributed Computing for Java,” in *Oracle Technetwork.* Available: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html#15

[35] Oracle Java Specification Documentation: *An Overview of RMI Applications*. Available: https://docs.oracle.com/javase/tutorial/rmi/overview.html

[36] M. Philippsen, B. Haumacher, and C. Nester, “More Efficient Serialization and RMI for Java,” in *Concurrency: Practice & Experience*, May 2000.

[37] D. Kurzyniec, T. Wrzosek, V. Sunderam, and A. Slominski. “RMIX: a multiprotocol RMI framework for Java,” in *Parallel and Distributed Processing Symposium,* April 2003.

[38] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. “Ibis: an Efficient Java-based Grid Programming Environment,” in *Joint ACM Java Grande - ISCOPE Conference,* November 2002.

[39] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman, “Efficient Java RMI for Parallel Programming,” in *ACM Transactions on Programming Languages and Systems (TOPLAS),* November 2001.

[40] G. L. Taboada, C. Teijeiro, and J, Tourino, “High Performance Java Remote Method Invocation for Parallel Computing on Clusters,” in *Computers and Communications, ISCC 12th IEEE Symposium*, July 2007.

[41] B. Amedro, D. Caromel, F. Huet Guillermo, and L. Taboada, “ProActive: Using a Java Middleware for HPC Design, Implementation and Benchmarks,” in *International Journal of Computers and Communications*, May 2009.

[42] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dougarra, “MPI: The Complete Reference*”* in *MIT Press Cambridge*, 1995, ISBN 0-262-69215-5.

[43] The Message Passing Interface (MPI) standard. Available: http://www.mcs.anl.gov/research/projects/mpi/

[44] B. Carpenter, G. C. Fox, S-H, Ko, and L. Sang, “mpiJava 1.2: API Specification,” in *Northeast Parallel Architecture Center*, 1999.

[45] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox, “MPJ: MPI-like message passing for Java,” in *Concurency and Computation Practice and Experience*, September 2000.

[46] M. A. Baker and D. B Carpenter, “mpiJava: An Object-Oriented Java interface to MPI,” in *the 1st Java Workshop at the 13th IPPS & 10th SPDP Conference*, April 1999.

[47] W. Gropp, E. L. Lusk, and A. Skjellum, “The MPICH Implementation of MPI,” in *Using MPI:Portable Parallel Programming with the Message Passing Interface, MIT Press*, May 1999.

[48] E. Gabriel, G. E. Fagg, and G. Bosilca, “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation,” in *Proceedings of 11th European PVM/MPI User Group Meeting*, September 2004.

[49] N. H. F. Beebem, “High-Performance Matrix Multiplication,” in *USI Report No. 3*, November 1990.

[50] K. Goto, and R. A. van de Geijn, “Anatomy of High-Performance Matrix Multiplication, “ in *ACM Transactions on Mathematical Software*, May 2008.

[51] J. A. Gunnels, G. M. Henry, and R. A. van de Geijn, “A Family of High-Performance Matrix Multiplication Algorithms,” in *Computational Science*, July 2001.

[52] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, “Introduction to Algorithms - Chapter 28, Strassen’s algorithm for matrix multiplication, in *MIT Press and McGraw-Hill*, 2001, ISBN 0-262-03293-7.

[53] G. H. Golub and C. F. Van Loan, “Matrix Computations,” *The John Hopkins University Press*, 1996, ISBN: 0-8018-5414-8.

[54] R. M. Piedra, “A Parallel Approach for Matrix Multiplication on the TMS320C4x DSP,” in *Texas Instruments DSP Application Report*, 1996.

[55] A. Gupta and V. Kumar, “Scalability of Parallel Algorithms for Matrix Multiplication,” in *International Conference on Parallel Processing*, Aug 1993.

[56] S. Sahni and V. Thanvantri, “Parallel Computing: Performance Metrics and Models,” 1995.

[57] Amazon EC2 instance types - c4.2xlarge. Available: https://aws.amazon.com/ec2/instance-types/

[58] Amazon EC2, Virtual Server Hosting. Available: https://aws.amazon.com/ec2/

[59] Open MPI v1.6.4 documentation: *mpicc, mpirun, Send and Recv*. Available: https://www.open-mpi.org/doc/v1.6/

[60] MPJ Express v0.44 Java Docs: *Send and Recv*. Available: http://mpj-express.org/docs/javadocs/index.html

[61] Oracle Java Specification Documentation: *ExecutorCompletionService*. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorCompletionService.html

[62] A. Cheptsov, “HPC in Big Data Age: An Evaluation Report for Java-Based Data-Intensive Applications Implemented with Hadoop and OpenMPI,” in *Proceeding EuroMPI/ASIA '14 Proceedings of the 21st European MPI Users Group Meeting*, September 2014.

[63] G. Fox, “Returning to Java Grande: High Performance Architecture for Big Data,” in *International Advanced Research Workshop on High Performance Computing*, July 2014.

[64] S. Jha, J. Qiu, A. Luckow, P. Mantha and G. C. Fox, “A Tale of Two Data-Intensive Paradigms: Applications, Abstractions, and Architectures,” in *Big Data IEEE International Congress*, July 2014.

[65] M. Zaharia, M. Chowdhury, and S. Shenker, “Spark: cluster computing with working sets.” *In HotCloud'10,”* in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, June 2010.

[66] Apache Storm. Available: http://storm.apache.org/

[67] J. Bonér, "Introducing Akka—simpler scalability, fault-tolerance, concurrency & remoting through actors." January 2010. Available: http://jonasboner.com/introducing-akka/ and http://akka.io/

[68] S. Yelick, M. Pike, K. Liblit, G. Hilfinger, G. Graham, and A. Colella, "Titanium: A High-Performance Java Dialect," in *Workshop on Java for High-Performance Network Computing*, February 1998.

Appendices

Appendix A

Open MPI-C Implementation of Parallel Matrix Multiplication Algorithm

```
/* Open MPI – C implementation of matrix multiplication */

#include “mpi.h”
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define MASTER 0                    /* taskid of first task */
#define FROM_MASTER 1         /* setting a message type */
#define FROM_WORKER 2         /* setting a message type */

#define N 3000            /* matrix dimension, this is changed to capture result from 500 to 3000 */

#include <time.h>

void print_matrix(int *A)
{
   for (int i=0; i<N; i++) {
    printf(“\n\t| “);
    for (int j=0; j<N; j++)
      printf(“%2d ,” *((A+i*N) + j));
    printf(“|”);
              }
  }

static int a[N*N],           /* matrix A to be multiplied */
                    b[N*N],          /* matrix B to be multiplied */
                    c[N*N];          /* result matrix C */

int main (int argc, char *argv[])
{
          char hostname[1024];
          gethostname(hostname, 1024); /* get the hostname */

          int       numtasks,            /* number of tasks in partition */
                    taskid,              /* a task identifier */
                    numworkers,          /* number of worker tasks */
                    source,              /* task id of message source */
                    dest,                /* task id of message destination */
                    mtype,               /* message type */
```

```
        rows,               /* rows of matrix A sent to each worker */
        averow, extra,offset,  /* used to determine rows sent to each worker */
        i, j, k;         /* loop variables */

MPI_Status status;

clock_t t;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
numworkers = numtasks-1;

//printf("debug: Hello world from process %d of %d on %s\n," taskid, numtasks, hostname);
if (taskid == MASTER)
{
        /* populate matrices A and B */
        for (i=0; i<N; i++){
                for (j=0; j<N; j++){
                        a[(i*N)+j]= i+j;
                        b[(i*N)+j]= i+j;
                }
        }

         /* Calculate row partitions of matrix A */
         averow = N/numworkers;
         extra = N%numworkers;
         offset = 0;
         mtype = FROM_MASTER;

         /* start the clock */
         t = clock();

        /* Send row partitions of A and entire matrix B to the worker processes */
         for (dest=1; dest<=numworkers; dest++){
                rows = (dest <= extra) ? averow+1 : averow;
                //printf("debug: Sending %d rows to task %d offset=%d\n,"rows,dest,offset);
                MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
                MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
                MPI_Send(&a[offset*N], rows*N, MPI_INT, dest, mtype,
                                MPI_COMM_WORLD);
                MPI_Send(&b, N*N, MPI_INT, dest, mtype, MPI_COMM_WORLD);
                offset = offset + rows;
         }

         /* Receive results from worker processes into resultant matrix C */
         mtype = FROM_WORKER;
```

```
            for (i=1; i<=numworkers; i++)
            {
                    source = i;
                    MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
                    MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
                    MPI_Recv(&c[offset*N], rows*N, MPI_INT, source, mtype,
                                    MPI_COMM_WORLD, &status);
                    //printf("debug: Received results from task %d\n,"source);
            }

           /* all workers returned results - calculate elapsed time */
            t = clock() - t;
             printf("master %d, with %d processes cluster - time in seconds for %dx%d = %f\n,"
                                    taskid,numtasks,N,N,((double)t)/CLOCKS_PER_SEC);

             printf("\n****************************************\n");

            /* debug: validate resultant matrix */
           //print_matrix((int*)c,N);

           }
   /**************************** worker task ************************************/
    if (taskid > MASTER)
    {
            /* Receive relevant row partitions of matrix A and entire matrix B */
            mtype = FROM_MASTER;
            MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
            MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
            MPI_Recv(&a, rows*N, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
            MPI_Recv(&b, N*N, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);

            /* perform partial multiplication */
            for (k=0; k<N; k++)
                    for (i=0; i<rows; i++)
                    {
                            c[(i*N)+k] = 0.0;
                            for (j=0; j<N; j++)
                              c[(i*N)+k] = c[(i*N)+k] + a[(i*N)+j] * b[(j*N)+k];
                    }

            /* Send the partial result to master process */
            mtype = FROM_WORKER;
            MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
            MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
            MPI_Send(&c, rows*N, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```

Appendix B

MPJ Express Implementation of Parallel Matrix Multiplication Algorithm

```
/* MPJ Express implementation of parallel matrix multiplication algorithm */

import mpi.*;
public class MM_MPI {
        public static void main(String[] args) {

                // for multicore and niodev
                int N = Integer.parseInt(args[3]);

                // for native
                //int N = Integer.parseInt(args[0]);

                // for hybdev
                //int N = Integer.parseInt(args[8]);

                //debug: to get the argument location of main.
                // for(String s : args)     System.out.println(“>>>“+s);

                int MASTER = 0;
                int FROM_MASTER = 1;
                int FROM_WORKER = 2;
                int numtasks,   /* number of tasks in partition */
                taskid,         /* A task identifier */
                        numworkers,     /* number of worker tasks */
                source,         /* task id of message source */
                dest,           /* task id of message destination */
                nbytes,         /* number of bytes in message */
                mtype,          /* message type */
                intsize,  /* size of an integer in bytes */
                dbsize,         /* size of A double float in bytes */
                i, j, k,  /* loop variables */
                        averow, extra,
                count;

                int[] A = new int[N*N];   /* matrix A to be multiplied */
                int[] B = new int[N*N];         /* matrix B to be multiplied */
                int[] C = new int[N*N];         /* result matrix C */

                int[] offset = new int[1];
                int[] rows  = new int[1];   /* rows of matrix A sent to each worker */
```

```
MPI.Init(args);

taskid = MPI.COMM_WORLD.Rank();
numtasks = MPI.COMM_WORLD.Size();
numworkers = numtasks - 1;


/* *************** Master Task ****************** */
if(taskid == MASTER) {
        /* attempt garbage collection */
        runGC();

        /* populate matrices A and B */
        for(i = 0; i < N; i++) {
                for(j = 0; j < N; j++) {
                        A[(i*N)+j] = i+j;
                        B[(i*N)+j] = i+j;
                }
        }

        /* Calculate row partitions of matrix A */
        averow = N/numworkers;
        extra = N%numworkers;
        offset[0] = 0;
        mtype = FROM_MASTER;

        /* start the clock */
        long start = System.currentTimeMillis();

        /* Send row partitions of A and entire matrix B to the worker processes */
        for(dest = 1; dest <= numworkers; dest++) {
                rows[0] = (dest <= extra) ? averow+1 : averow;
                //System.out.printf("debug: Sending %d rows to task %d
                offset=%d\n,"rows[0],dest,offset[0]);
                MPI.COMM_WORLD.Send(offset, 0, 1, MPI.INT, dest, mtype);
                MPI.COMM_WORLD.Send(rows, 0, 1, MPI.INT, dest, mtype);
                count = rows[0] * N;
                MPI.COMM_WORLD.Send(A, (offset[0]*N), count, MPI.INT, dest,
                mtype);
                count = N*N;
                MPI.COMM_WORLD.Send(B, 0, count, MPI.INT, dest, mtype);
                offset[0] = offset[0] + rows[0];
        }

        /* Receive results from worker processes into resultant matrix C */
```

```
		mtype = FROM_WORKER;
		for(i = 1; i <= numworkers; i++) {
			source = i;
			MPI.COMM_WORLD.Recv(offset, 0, 1, MPI.INT, source, mtype);
			MPI.COMM_WORLD.Recv(rows, 0, 1, MPI.INT, source, mtype);
			count = rows[0] * N;
			MPI.COMM_WORLD.Recv(C, offset[0]*N, count, MPI.INT, source,
			mtype);
			//System.out.printf("debug: Received results from task %d\n,"source);
		}

		/* all workers returned results - calculate elapsed time */
		long stop = System.currentTimeMillis();
		System.out.printf("\nmaster %d with %d processes for %dx%d - time in seconds
		= %f\n,"taskid,numtasks,N,N,(stop - start)/1000.0);
		System.out.println("\n*********************************");

		/* debug: validate resultant matrix */
		//print_matrix(taskid,C,N);
	}

	/* ***************** worker task ******************** */
	if(taskid > MASTER) {
		/* Receive relevant row partitions of matrix A and entire matrix B */
		mtype = FROM_MASTER;
		source = MASTER;
		MPI.COMM_WORLD.Recv(offset, 0, 1, MPI.INT, source, mtype);
		MPI.COMM_WORLD.Recv(rows, 0, 1, MPI.INT, source, mtype);
		count = rows[0] * N;
		MPI.COMM_WORLD.Recv(A, 0, count, MPI.INT, source, mtype);
		count = N * N;
		MPI.COMM_WORLD.Recv(B, 0, count, MPI.INT, source, mtype);

		/* perform partial multiplication */
		for(i = 0; i < rows[0]; i++) {
			for(k = 0; k < N; k++) {
				C[(i*N)+k] = 0;
				for(j = 0; j < N; j++) {
					C[(i*N)+k] = C[(i*N)+k] + A[(i*N)+j] * B[(j*N)+k];
				}
			}
		}

		/* Send the partial result to master process */
		mtype = FROM_WORKER;
```

```
                        MPI.COMM_WORLD.Send(offset, 0, 1, MPI.INT, MASTER, mtype);
                        MPI.COMM_WORLD.Send(rows, 0, 1, MPI.INT, MASTER, mtype);
                        MPI.COMM_WORLD.Send(C, 0, rows[0]*N, MPI.INT, MASTER, mtype);
                }

                MPI.Finalize();
        }

        static void print_matrix(int rank, int[] A, int N) {
                 int i, j = 0;
                 System.out.println("rank = "+rank);
                 for (i=0; i<N; i++) {
                                System.out.printf("\n\t| ");
                         for (j=0; j<N; j++)
                                 System.out.printf("%2d ,"A[(i*N)+j]);
                         System.out.printf("|");
                 }
        }

        static void runGC(){
                System.runFinalization();
                System.gc();
                System.gc();
                System.out.println("gc complete");
        }

}
```