X10 vs Java: Concurrency Constructs and Performance

Anh Trinh
San Jose State University

X10 vs Java:

Concurrency Constructs and Performance

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

Of the Requirement for the Degree

Master of Science

by
Anh Trinh

November 2011

SAN JOSE STATE UNIVERSITY

The Undersigned Project Committee Approves the Project Titled

X10 VS JAVA: CONCURRENCY CONSTRUCTS AND PERFORMANCE

by

Anh Trinh

APPROVED FOR THE DEPARMENT OF COMPUTER SCIENCE

Dr. Robert Chun	Department of Computer Science	Date
Dr. Soon Tee Teoh	Department of Computer Science	Date
Mr. Peter Tran	NASA Ames Research Center	Date

ABSTRACT

To avoid overheating the chip, chip designers have switched to multi-cores. While multi-core CPUs reserve instruction-level parallelism features that help existing applications run as if they were running under single core, applications do not reach speeds two or four times faster. Instead of relying on compiler and hardware to figure out parallelism in source code, software developers now must control parallelism explicitly in their programs. Many programming languages and libraries, such as Java, C# .NET, and OpenMP, are trying to help programmers by providing rich concurrency API. X10 is the new experimental language from IBM Research, which has been under development since 2004 targeting multi-core programming ranging from multi-cores single machine to cluster. This project examines the X10 parallel constructs, compares its usability with the Java language, the OpenMP library, and then compares the performance between X10 and Java language.

ACKNOWLEDGEMENT

I would like to thank Dr. Robert Chun for his patience and support throughout this project. I would also like to thank Dr. Mark Stamp for stepping in as my adviser at the last minute, and Dr. Soon Tee Teoh for accepting to be a committee member, and last but not least, I thank Peter Tran for looking over my work and James Le for reviewing my writing.

Table of Contents

List of Table and Figures

1. Introduction

Moore's law states that the number of transistors on a chip will double about every two years, and this statement has proven to be true thus far [8]. As the number of transistors increase, CPU clock speed rises as well. Current clock speed has increased to a level where it overheats, which has become a major issue [14, 15]. To achieve better efficiency without violating Moore's law, chip designers have switched to multi-cores. While multi-core CPUs reserve an instruction-level parallelism (ILP) feature that helps existing applications run as if they were running under single core, applications do not reach speeds of two times (2X) faster under dual-cores or four times (4X) faster under quad-cores [8]. The reason is that applications were designed and programmed in a single-threaded, sequential process fashion.

To maximize all cores, programmers must think, design, and program in a parallel model [3, 9]. Programmers are no longer able to enjoy the implicit parallelism of the ILP to maximize core performance. Now they must control the parallelism explicitly. Programmers must do the dirty work of taking care of deadlocks and race conditions. Many programming languages and libraries such as Java, C# .NET, and OpenMP, are trying to help programmers by providing rich concurrency API. X10 is the new experimental language from IBM Research, which has been under development since 2004. The main focus of the language is parallel programming. X10 claims that by using its concurrency constructs, X10 programs are guaranteed to be free of deadlocks and race conditions [4, 24]. This paper briefly touches on parallel computing architecture, but its main focus is on comparing concurrent constructs that deal with deadlock and race conditions between Java language, OpenMP library, and the X10 language, then comparing performance between Java and X10 on a single local machine.

2. Parallel computing

2.1. What is parallel computing?

Parallel computing is a method of computation in which many calculations happen simultaneously [1]. The principle of parallel computing is to divide a large problem into small, computable sub-problems and to have them computed concurrently to reduce runtime. Sub-problems can be distributed among computing resources on a network, or among cores within CPU die, or a combination of both where each computing resource contains multi-cores CPU. In computer science, parallel computing often deals with system hardware and software issues related to concurrent computations [2].

2.2. Parallel computing architecture

2.2.1. Macro architecture

Macro architecture is parallel computing architecture that deals with large-scale systems. The architecture is concerned with how to assemble individual computing nodes together to perform massive work [25]. The following are typical macro parallel computing architectures:

- A cluster is a group of computers linked together through a high-speed local area network (LAN). Clusters are usually used in high-performance calculations and to increase availability [6].

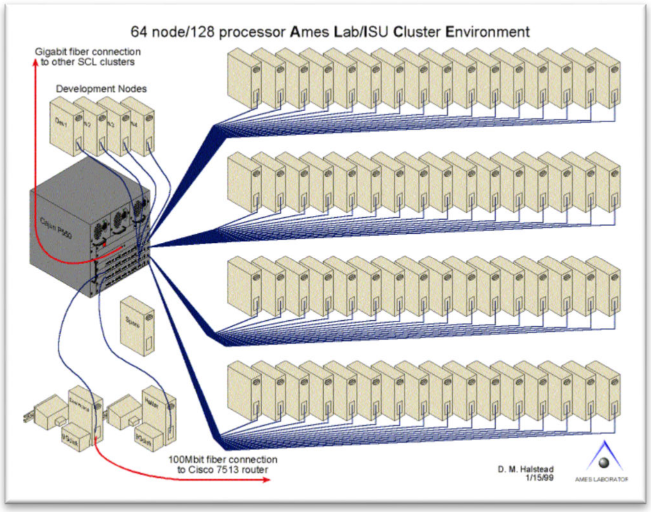

64 node/128 processor Ames Lab/ISU Cluster Environment
Gigabit fiber connection to other SCL clusters
Development Nodes
Cajun P550
100Mbit fiber connection to Cisco 7513 router
D. M. Halstead 1/15/99
AMES LABORATORY

Figure 1: Examples of cluster computers [19]

- Grid computing (e.g. NASA Columbia Supercomputer with 10,240-CPU SGI Altix supercluster, with Intel Itanium 2 CPUs) can be thought of as a distributed system for big tasks that require massive computation and a great number of files. Grid combines computers from different administrative domains to solve a single task. Grid components may be dispersed geographically and more loosely coupled. A grid uses middleware to divide a big program into smaller pieces and to distribute among nodes [7].

Figure 2: Grid computing example [16]

- Cloud distributed computing (e.g. Amazon Elastic Compute Cloud (EC2), Microsoft Azure, etc.) describes remote services, such as computation, software, data access, and storage, that end-user need not be aware of any physical location or the configuration of the system that delivers the services. This is similar to the concept of an electrical grid where end-consumers use resources without needing to understand the component devices in the electrical grid [8].

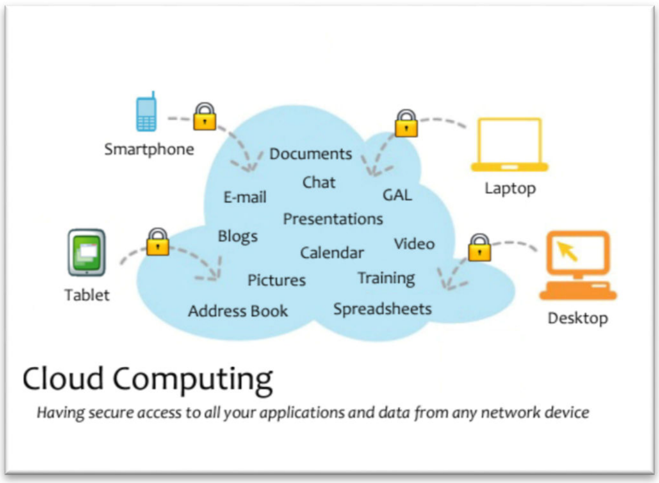


Figure 3: Cloud services, applications, and access devices

2.2.2. Micro architecture

While macro architecture deals many computing nodes in a network, micro parallel computing architecture is concerned with how to put parallelism into each computing node at the transistor level [25]. Multi-core is the new era in personal computing, since pushing clock speed at such high levels also dramatically increases the heat within the CPU dies and chassis. To have greater efficiency without violating Moore's law, chip designers have switched to more cores. Multi-core is the new way to solve the problem by putting many computing processors on one single die [8].

2.3. Parallel computing classification (Flynn's taxonomy)

There are four parallel computing platforms spanning across two-dimensional matrix. The first dimension is instruction centric, focusing on how many instruction streams a computer architecture may process at one single point in time. The second is data centric and is concerned with how many data streams can be processed at a single point in time. The classification in the table below is known as Flynn's taxonomy [8].

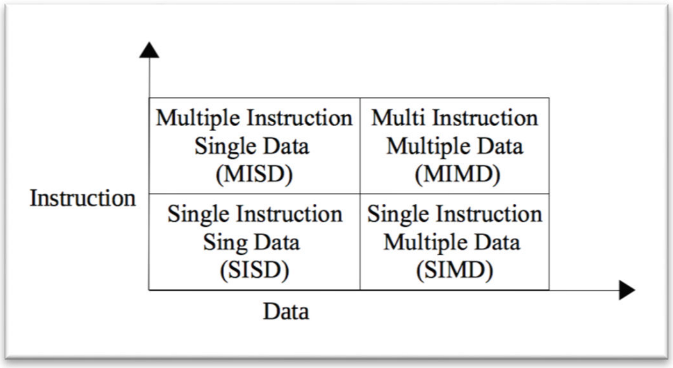

Instruction
Multiple Instruction Single Data (MISD)
Multi Instruction Multiple Data (MIMD)
Single Instruction Sing Data (SISD)
Single Instruction Multiple Data (SIMD)
Data

Figure 4: Flynn's taxonomy

- Single Instruction Single Data (SISD) is a traditional sequential platform where there is no parallelism in hardware. Only one data stream is processed per clock cycle and instruction execution is completed in serial fashion.
- Multiple Instructions Single Data (MISD) is a computer platform that has the ability to process a single data stream with multiple instruction streams
- Single Instruction Multiple Data (SIMD) is a computer platform in which one single instruction stream can process multiple data streams. Almost all computers are SIMD machines.
- Multiple Instructions Multiple Data (MIMD) is a computer platform with the capacity to execute multiple instructions streams on different independent data streams. MIMD is the most common parallel computing platform today and is exhibited in technology such as Intel multi-core processors.

3. Parallelism with hardware solution

Instruction Level Parallelism (ILP) is processor and compiler techniques that speed up program execution via parallel execution of individual RISC-based operations, such as memory load and stores, integer additions, and floating-point multiplications. Although operations are executed in parallel, there is only a single thread of execution. The processors-compiler system hands a single program, written for a sequential processor, from which it extracts the parallelism automatically. The rationale behind this feature is that parallelism becomes transparent in the system. Programmers do not need to worry about parallelizing the program. Instead they can write code in sequential and let compiler and processors find parallelism in the program. Therefore, when computers get newer and faster processors, programs tend to be faster [10].

3.1. Pipelining (intra-instruction parallelism)

Pipelining is one method of injecting parallelism, or concurrency, into computer programs at the hardware level. It refers to a segmentation of an instruction into phases or stages, which are executed by dedicated, independent units. A RISC instruction is divided into multiple stages [Instruction]=>[IF | ID | OF | EX] (IF: Instruction Fetch; ID: Instruction Decode; OF: Operand Fetch, EX: Execute). Mimicking an industrial assembly line, RISC-based consecutive instructions can be executed at the same time in an overlapping fashion. In short, pipelining is a technique that splits a repeated sequential instruction into stages where each stage can be executed efficiently and concurrently by dedicated autonomous modules. [11]

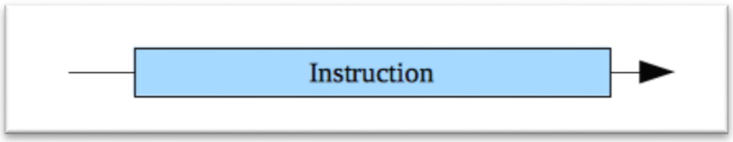


Figure 5: Execute non-pipelined instructions one-by-one in sequential fashion

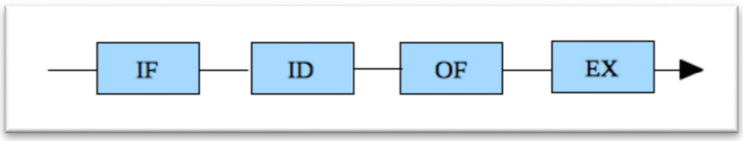
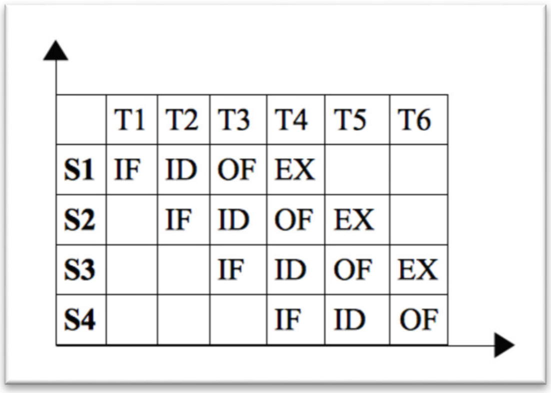


Figure 6: Pipeline instruction is divided into 4 phases



	T1	T2	T3	T4	T5	T6
S1	IF	ID	OF	EX		
S2		IF	ID	OF	EX	
S3			IF	ID	OF	EX
S4				IF	ID	OF

Figure 7: Multiple pipelined instruction execution, each overlaps each other at different phases

Multiple pipelined instructions are executed at the same time but at different stages. Instruction execution overlaps each other at different phases. Since each phase is an independent process, each instruction can execute concurrently, assuming no data dependency between them as depicted in Figure 7.

3.2. Multi-function units (inter-instruction parallelism)

All ILP processors have multiple functional units that can execute multiple operations in parallel. Independent instructions are executed in parallel even though a sequential program was given to the hardware. Superscalar processors perform resource allocation and data dependency between program instructions. If instructions are independent from each other and there are available functional units, sequential instructions can be executed in parallel. Superscalar processor will make sure multiple instructions do not try to use the same functional unit at the same time.

Example: A, B, I, T, X, and Y are simple integers

1: A = I * T

2: B = X * Y

3: I = A + B

If there are two addition units and two multiply units available in a processor and they are not in use, instructions 1 and 2 can be executed in parallel since there are two multiply units. Furthermore, instructions 1 and 2 do not have any data conflict. They are independent of each other.

3.3. VLIW (Very Long Instruction Word)

Using ILP to inject transparent parallelism increases hardware complexity because hardware must perform all conditional branch prediction, out-of-order execution, and other ILP operations. VLIW tries to achieve all of ILP features with reduced hardware complexity. Instructions have variable length. Each instruction may pack many operations and send those operations to processors for execution. For fully utilizing VLIW processors, compilers must be smart enough to figure all dependencies, to ensure no resource hazards, and to analyze and find parallelism in code [10].

3.4. Hyper-Threading

A physical processor is made up of different resources, such as CPU registers, interrupt controller registers, caches, buses, execution units, and branch prediction logic. A thread, on the other hand, needs only an architectural state. Therefore, a logical processor can be created by duplicating architectural space. This is called simultaneous multi-threading, or Intel's Hyper Threading Technology. The extra hardware allows software to look at a single core as if there were two. Multiple threads can be scheduled to logical processors as they would be on multiprocessors. Since there is only one physical core, threads must go interleaving. One process may stall and allow the other to proceed. With extra hardware, processors accelerate context switching between threads to achieve faster speed [23].

3.5. Multi-core

As the number of transistors increase to obey Moore's law, the CPU clock frequencies have become faster. However, this also causes heat to increase greatly. Intel engineers stated that if microprocessor keeps increase clock frequency, its temperature may reach the sun surface level [12, 25]. So ditching the idea of further raising frequency speeds, engineers opted to add more cores. By going with a multi-core solution, CPUs lose heat and gain parallelism and power efficiency. With each core having ILP capability and its own architecture state – register, interrupt logic, caches, buses, executions units, and branch prediction – programs are no longer executed in interleaving fashion. Two cores can execute two programs simultaneously. Better yet, each core also has Hyper-Threading capability.

4. Why have parallelisms in software?

At the micro architecture level, chip designers solve the heat problem without violating Moore's law with the multi-cores solution. Yet, the new trend also has its own problems. To take advantage of the new multi-cores chips, programs must utilize all cores, which means programs must be executed on different cores in parallel, otherwise it is the same as running on single core machine. The idea of running the same source code programs faster on next year's hardware is no longer valid [26]. Hardware solutions are not feasible for software development. Parallelism now is exposed directly to software developers, which means software developers must explicitly design and code the program in parallel thinking [9]. Now, programmers must take full control of the parallelism in their programs.

5. Problems in parallel computing

In parallel programming, deadlocks and race conditions are the most common issues. Between the two, race conditions are difficult to detect and solve because different runtime environments can yield different results. At later program execution, race conditions may not occur. In computer science, a deadlock is a situation where two or more processes are waiting for each other to release their resource. If there are more than two, processes will create a circular chain of waiting [8]. The situation can be illustrated by a traffic jam, as in the figure below. In programming, an example of 3 processes, P1, P2, and P3, denoted as circles and 3 resources, R1, R2, and R3, denoted as squares, each of the processes need to have at least two resources to complete a task as in Figure 9. Solid lines represent ownership and dotted lines represent resource requests. In this scenario, a resource allocation circle is formed, which causes deadlock where each process holds and waits for available resources.

Figure 8: Traffic jam illustration (http://chake.chinatefl.com/images/szintersection.jpg)

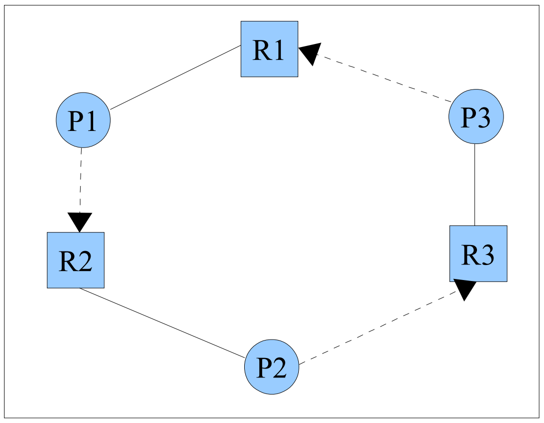


Figure 9: Resource process deadlock diagram

A race condition is a situation where multi processes execute code out-of-order that produces incorrect data [8, 13]. This bug is hard to detect and avoid because different runtime environments yield different results. One of the classic problems is the sequence number where two or more processes trying to get unique sequence numbers. However, if those processes try to get the number at the exact same time, this results in the two processes getting the same number.

6. Java and OpenMP solutions

6.1. Java implementation

To ensure that there are no race conditions, the Java language provides two basic synchronization approaches. By supplying keyword `synchronized`, synchronized methods and statements are guaranteed to be executed atomically. Java 5.0/6.0 SE provides many useful common concurrent APIs within the `java.util.concurrent` package. The package contains thread-safe concurrent collection such as `BlockingQueue`, `ArrayBlockingQueue`, `ConcurrentHashMap`, `ConcurrentLinkedQueue`, and other concurrency utilities for task/thread scheduling. The `java.util.concurrent.atomic` provides a small set of classes that support lock-free, thread-safe, atomic operations on single variable, such as `AtomicBoolean`, `AtomicInteger`, `AtomicIntegerArray,` and others. There is also the Java keyword `volatile` used for multi-threaded Java programming so that multiple threads access the same memory space and not from each respective thread's local variable cache.

The following Java program demonstrates the classic example of race condition in the form of the banking problem. The problem shows that if two people withdraw and deposit using the same account at the same time, the account balance shows different numbers at different runtimes due to memory consistency errors.

The program in Figure 10 will run as following: `Depositor` will perform `deposit()` operation 100 times and `Withdrawer` will perform `withdraw()` operation on the `Account` 100 times with the amount of $10 each. The `Account` will have a $10,000 initial balance. Each object runs on its own thread concurrently. We assume that the account balance should have the initial amount of $10,000 when the program finishes. The program runs repeatedly 500 times as in Figure 11. Most of the results return $10,000 which is correct, but there is at least one runtime that does not give $10,000.

```
import java.io.*;
class Banking{
        static class Account{
            private double balance = 10000;

            public Account(){}
            public double deposit(double amount){
                      balance -= (amount < balance) ? amount : 0.00 ;
                      return balance;
            }
            public double withdraw(double amount){
                      balance += amount;
                      return balance;
            }
            public double check(){
                      return balance;
            }
    }

    static class Withdrawer implements Runnable {
            String name = "Withdrawer";
            Account account;
            public Withdrawer(Account account){
                      this.account = account;
            }
            public void run(){
                      for(int i=0; i<100; i++){
                double newBalance = account.deposit(10);
                      //Do do something with new balance code
                      }
                      System.out.println(name + " finishes");
            }
    }
    static class Depositor implements Runnable{
            String name = "Depositor";
            Account account;
            public Depositor(Account account){
                      this.account = account;
            }
            public void run(){
                      for(int i = 0; i < 100; i++){
                        double newBalance = account.withdraw(10);
                      //Do do something with new balance code
                      }
                      System.out.println(name + " finishes");
            }
    }

    public static void main(String[] args){
            Account account = new Account();
            Thread t1 = new Thread( new Withdrawer(account));
            Thread t2 = new Thread( new Depositor(account));
            try{
                      t1.start();
                      t2.start();

                      t1.join();
                      t2.join();
            }catch(InterruptedException e){
                      e.printStackTrace();
            }
            System.out.println("Balance: " + account.check());
    }
}
```

Figure 10: Race condition of the banking problem, Depositor deposits to the Account while Withdrawer withdraws from the account. Both perform their respective actions at the same time.

```
sh-3.2$ ./runjava.sh Banking 10
 ...
 Withdrawer finishes Depositor finishes Balance: 10000.0
 Depositor finishes Withdrawer finishes Balance: 9000.0
 Withdrawer finishes Depositor finishes Balance: 10000.0
 Withdrawer finishes Depositor finishes Balance: 10000.0
 Withdrawer finishes Depositor finishes Balance: 10000.0
 ...
```

Figure 11: Output shows that race condition causes incorrect balance

When forcing a thread to sleep for a random number between 0-10 of milliseconds before performing `withdraw()` or `deposit()` at line 6 of Figure 12, 13, it increases the number of wrong results. When each thread sleeps for a random of number milliseconds within a small range, it increases the chance that `Depositor` and `Withdrawer` perform their actions at the same time. The updated value from one operation does not reflect the other operation.

```
...
 public void run(){
     java.util.Random random = new java.util.Random();
     for(int i=0; i<100; i++){
         try{
             Thread.sleep(random.nextInt(10));       //Line 6: Sleep
         }catch(Exception e){
             e.printStackTrace();
         }
         double newBalance = account.deposit(10);
     }
     System.out.println(name + " finishes");
 }
 ...
```

Figure 12: Modify Depositor object making it sleep for a random number between 0-10 milliseconds before performing deposit

```
...
 public void run(){
     java.util.Random random = new java.util.Random();
     for(int i=0; i<100; i++){
         try{
             Thread.sleep(random.nextInt(10));       //Line 6:Sleep for random ms
         }catch(Exception e){
             e.printStackTrace();
         }
         double newBalance = account.withdraw(10);
     }
     System.out.println(name + " finishes");
 }
 ...
```

Figure 13: Modify Withdrawer object making it sleep for a random number between 0-10 milliseconds before performing withdraw

```
sh-3.2$ ./runjava.sh Banking 10
Depositor finishes Withdrawer finishes Balance: 10000.0
Withdrawer finishes Depositor finishes Balance: 9990.0
Depositor finishes Withdrawer finishes Balance: 10000.0
Withdrawer finishes Depositor finishes Balance: 9960.0
Withdrawer finishes Depositor finishes Balance: 9990.0
Depositor finishes Withdrawer finishes Balance: 10000.0
Depositor finishes Withdrawer finishes Balance: 9990.0
Depositor finishes Withdrawer finishes Balance: 9990.0
Depositor finishes Withdrawer finishes Balance: 9990.0
Withdrawer finishes Depositor finishes Balance: 9990.0
```

Figure 14: Output after sleep statement added; the number of wrong results increases

Looking at the banking problem, we are attempting to use Java Atomic variable because every operation is trying to modify the `balance` variable. However, Java only provides `AtomicInteger` in `java.util.concurrent.atomic` package; it does not offer `AtomicDouble`. Instead, we use regular Java Intrinsic Lock. We create synchronized block with `lock` variable as type `Object` and have `synchronized(lock)` statement surrounding the updating statements. Whenever a thread A wants to execute the block, that thread acquires `lock` and releases after it finishes. If `lock` is acquired or used by some different threads, thread A will wait until `lock` is released before it continues. The modification to the program is at lines 7 and 14, as shown in Figure 15.

```
static class Account{
        private double balance = 10000;
        private Object lock = new Object();
        public Account(){}

        public double deposit(double amount){
            synchronized(lock){                                      //Line 7
                balance -= (amount < balance) ? amount : 0.00 ;
            }
            return balance;
        }

        public double withdraw(double amount){
            synchronized(lock){                                      //Line 14
                balance += amount;
            }
            return balance;
        }

        public double check(){
            return balance;
        }
  }
```

Figure 15: Account object with locking mechanism; only thread with lock can perform action

On the other hand, we can even simplify the solution by adding a synchronized method instead of a synchronized block as shown at lines 6 and 11 in Figure 16. Both solutions give the same result.

```
static class Account{
        private double balance = 10000;

        public Account(){}

        public synchronized double deposit(double amount){        //Line 6
            balance -= (amount < balance) ? amount : 0.00 ;
            return balance;
        }

        public synchronized double withdraw(double amount){       //Line 11
            balance += amount;
            return balance;
        }

        public synchronized double check(){
            return balance;
        }
  }
```

Figure 16: Java synchronized methods

Another classic race condition example is the sequence number generator. It is a typical situation where two or more clients request a unique number from a server. It is critical that the server will not give the same number to more than one client.

The program in Figure 17 tries to demonstrate the race condition in the code and is implemented as follows: There is one `SequenceNumberGenerator` object that is responsible for spitting out the next number in the sequence; multiple `Requester` objects will run on different threads trying to acquire a unique number. In this experiment, there are two requesters and each will ask for a unique number for 1000 times. Each time, a thread will sleep for a random number between 0-10 milliseconds before asking for the next number. Because there are two threads and each requests 1000 numbers each, the last request should have value of 2000. The running output shows that 10 different runtimes return different incorrect results in Figure 18.

```
import java.io.*;

 class SequenceNumber{
     static class SequenceNumberGenerator{
         private int num = 0;
         public SequenceNumberGenerator(int initial){
             num = initial;
         }
         public int getNext(){
             return ++num;
         }
         public int currentValue(){
             return num;
         }
     }

     static class Requester implements Runnable{
         private SequenceNumberGenerator sng;
         private String name;
         public Requester(String name, SequenceNumberGenerator sng){
             this.name = name;
             this.sng = sng;
         }

         public void run(){
             java.util.Random random = new java.util.Random();
             try {
                 for(int i=0; i<1000; i++){
                     Thread.sleep(random.nextInt(10));
                     int value = sng.getNext();
                 }
             } catch (InterruptedException e) {
             } finally {
                 System.out.println(this.name + " : " + sng.currentValue());
             }
         }
     }

     public static void main(String[] args){
         SequenceNumberGenerator sng = new SequenceNumberGenerator(0);

         Thread t1 = new Thread( new Requester("A", sng) );
         Thread t2 = new Thread( new Requester("B", sng) );
         t1.start();
         t2.start();
     }
 }
```

Figure 17: Java implementation of sequence number generator

```
sh-3.2$ ./runjava.sh SequenceNumber 10
 A : 1941 B : 1977
 B : 1959 A : 1982
 A : 1946 B : 1989
 A : 1958 B : 1990
 B : 1942 A : 1995
 B : 1956 A : 1982
 B : 1932 A : 1979
 A : 1968 B : 1982
 B : 1978 A : 1985
 A : 1974 B : 1986
```

Figure 18: Output of the sequence generation program running 10 times

One solution is to make the integer increment in atomic operation. To do that, we have to add `lock` object as described above, and have a `synchronized(lock)` statement surrounding `num++` to ensure atomicity. An alternative and quicker way is to use an `AtomicInteger` object, which includes useful atomic methods, one of which is `incrementAndGet()`. `AtomicInteger` object fits perfectly to the problem. The program output shows 50 different runtimes giving 50 of the exact same results, which proves that this solution is correct. In Figure 19, each line in the output has 2 numbers, A:1974 and B:2000, meaning A's last requested value is 1974 and B's last requested value is 2000. In this program, we only care about the last request, which should be 2000 every time.

```
static class SequenceNumberGenerator{
     private AtomicInteger num = new AtomicInteger(0);                     //Line 2
     public SequenceNumberGenerator(AtomicInteger initial){
         num = initial;
     }

     public int getNext(){ return num.incrementAndGet();}

     public int currentValue(){ return num.get(); }
 }
```

Figure 19: Sequence number generator uses AtomicInteger instead of Integer

```
sh-3.2$ ./runjava.sh SequenceAtomicNumber 50
 A : 1994 B : 2000
 A : 1957 B : 2000
 B : 1984 A : 2000
 B : 1980 A : 2000
 A : 1975 B : 2000
 B : 1985 A : 2000
 A : 1969 B : 2000
 B : 1956 A : 2000
 B : 1939 A : 2000
 B : 1989 A : 2000
 B : 1994 A : 2000
 A : 1996 B : 2000
 A : 1981 B : 2000
 A : 1958 B : 2000
 A : 1987 B : 2000
 B : 1969 A : 2000
 B : 1980 A : 2000
 B : 1969 A : 2000
 B : 1964 A : 2000
 B : 1953 A : 2000
 B : 1968 A : 2000
 A : 1992 B : 2000
 A : 1976 B : 2000
 B : 1976 A : 2000
 A : 1945 B : 2000
 B : 1957 A : 2000
 A : 1983 B : 2000
 B : 1980 A : 2000
```

```
A : 1988 B : 2000
A : 1996 B : 2000
A : 1960 B : 2000
A : 1994 B : 2000
A : 1972 B : 2000
B : 1982 A : 2000
A : 1988 B : 2000
B : 1966 A : 2000
A : 1945 B : 2000
B : 1998 A : 2000
B : 1998 A : 2000
A : 1965 B : 2000
A : 1961 B : 2000
A : 1990 B : 2000
B : 1978 A : 2000
A : 1986 B : 2000
A : 1992 B : 2000
B : 1995 A : 2000
A : 1991 B : 2000
A : 1997 B : 2000
B : 1971 A : 2000
A : 1974 B : 2000
```

Figure 20: Output of sequence number generator using AtomicInteger implementation.

The next program extends the bank example. In this scenario, there are two people and each has an individual account. Simultaneously, each will transfer money from his own account to the other with an amount of $100. To transfer funds, a person must acquire his own account first, then he will try to acquire the other. After he gets hold of two accounts, he can withdraw an amount from his account and deposit that amount to the other account. The deadlock occurs when two people try to transfer at the exact same time. Each transferor locks his account and keeps waiting until the others are free, but none of them release their own locks. Therefore, both will wait indefinitely in this “hold and wait” scenario, as Java code demonstrates in Figure 21.

```java
import java.io.*;
 class BankDeadlock{
    static class Account{
        private int accountNumber = 0;
        private double balance = 0;

        public Account(int accountNumber, double initialAmount){
            this.accountNumber = accountNumber;
            this.balance = initialAmount;
        }
        public double withdraw(double amount){
            balance -= amount;
            return balance;
        }
        public double deposit(double amount){
            balance += amount;
            return balance;
        }
        public double checkBalance(){
            return balance;
        }
    }

    static class Transferrer implements Runnable{
        private String name = "";
        private Account myAccount;
        private Account otherAccount;
        public Transferrer(String name, Account myAccount, Account otherAccount){
            this.name = name;
            this.myAccount = myAccount;
            this.otherAccount = otherAccount;
        }

        public void run(){
            System.out.println(name + " waits to lock my account");
            synchronized(myAccount){
                System.out.println(name + " locks my account");
                System.out.println(name + " waits to lock other account");
                synchronized(otherAccount){
                    System.out.println(name + " locks other account\nStart transfer");
                    myAccount.withdraw(100);
                    otherAccount.deposit(100);
                }
            }

            System.out.println(name + " finished transfer. My account: $" +
              myAccount.checkBalance() + " Other: $" + otherAccount.checkBalance());
        }
    }

    public static void main(String[] args){
        Account A = new Account(100, 10000);
        Account B = new Account(101, 10000);
        Thread t1 = new Thread( new Transferrer("A", A, B) );
        Thread t2 = new Thread( new Transferrer("B", B, A) );
        t1.start();
        t2.start();
    }
 }
```

Figure 21: Extended banking example, Transferrer A and B objects simultaneous transfer money from his account to the other. If A, B happens to transfer at the same time, a deadlock occurs due to locking resources

As for deadlocks, Java has no silver bullet to solve the problem. Instead, it is up to developers to manage these issues. As stated by Oaks and Wong [17], "Deadlock between threads competing for the same set of locks is the hardest problem to solve in any threaded program. It's a hard enough problem, in fact, that it cannot be solved in the general case. Instead, we try to offer a good understanding of deadlock and some guidelines on how to prevent it. Preventing deadlock is completely the responsibility of the developer. The Java virtual machine does not do deadlock prevention or deadlock detection on your behalf." To solve this problem, we impose a hierarchical resource allocation. Resources have to be allocated in a certain order. In this banking example, the solution is to use checksums of the two accounts. The account with a smaller hash-sum value will be acquired first, then the other, as in Figure 22.

```
static class Transferrer implements Runnable{
        private String name = "";
        private Account myAccount;
        private Account otherAccount;
        private Account account1, account2;
        public Transferrer(String name, Account myAccount, Account otherAccount){
            this.name = name;
            this.myAccount = myAccount;
            this.otherAccount = otherAccount;
        }

        public void run(){
            int myAccountHashValue = System.identityHashCode(myAccount);
            int otherAccountHashValue = System.identityHashCode(otherAccount);

            if( myAccountHashValue < otherAccountHashValue ){
                account1 = myAccount;
                account2 = otherAccount;
            }else if( myAccountHashValue > otherAccountHashValue ){
                account1 = otherAccount;
                account2 = myAccount;
            }else{
            }

            System.out.println(name + " waits to lock " + account1.getAccountNumber());
            synchronized(account1){
                System.out.println(name + " locks " + account1.getAccountNumber());
                System.out.println(name + " waits to lock " + account2.getAccountNumber());
                synchronized(account2 ){
                    System.out.println(name + " locks " + account2.getAccountNumber() +
                                                                    "\nStart transfer");
                    myAccount.withdraw(100);
                    otherAccount.deposit(100);
                }
            }

            System.out.println(name + " finished transfer. My account: $" +
                        myAccount.checkBalance() + " Other: $" + otherAccount.checkBalance());
        }
 }
```

Figure 22: Hierarchical resource allocation – whichever account has a smaller hash value will be acquired first

6.2. OpenMP Implementation

OpenMP is an API that supports SMP programming in C/C++ and FORTRAN. OpenMP operates on a fork/join model. Programmers observe source codes and analyze any region that can run parallel, then they can insert OpenMP directives to tell the compiler that those regions can be divided and each be worked on independently. The main thread forks child threads to perform concurrent tasks. In the end, all the slave threads join back to the main thread to finish final computation.

Similar to other multi-thread programming languages and libraries, OpenMP is still susceptible to race condition and deadlock. Even though OpenMP has several safety nets to help avoiding race condition, it does not prevent programs from getting this kind of error. The OpenMP library is relatively small, so the learning curve is relatively flat. The most important aspect of developing OpenMP programs is knowing how and when to set variables as shared or private.

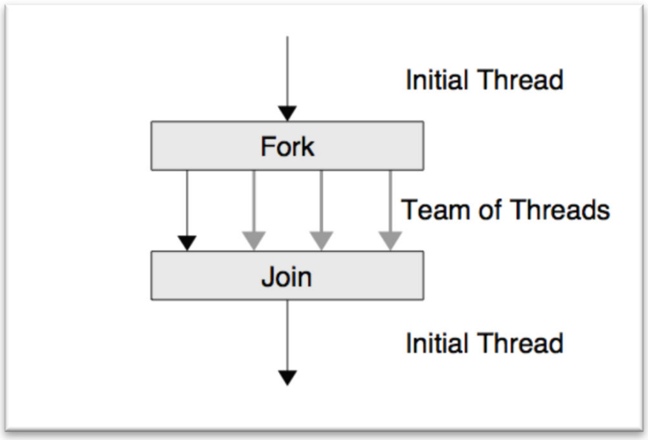


Figure 23: The fork/join programming model in OpenMP – the program starts as a single thread. A team of threads will be forked at the parallel region and join back at the end.

By default, variables are shared, except for loop index variables. If developers want to a variable to be declared private within each thread, they must declare those variables using the `private` directive. The program in Figure 24 demonstrates how OpenMP library deals with race conditions using shared and private variables. The program will generate 100 random numbers and put them in an array. This array simulates scores for 10 students where each has 10 scores. The elements from [0-9] represent the scores of the first student; elements [10-19] represent the scores of the second student; and subsequence intervals are for subsequent students respectively.

The program will calculate averages for all students. First, the program will run only with one thread. Later, it will add OpenMP directives to make it multi-threaded. At line 28 in Figure 24, we tell the compiler that we want to run the loop in parallel. Each `i` index value tells which student's scores should be calculated. For example, if a thread happens to calculate the average of the fifth student, which `i` has value of 4, it must iterate and sum all scores from [40-49] and compute the average.

```
#include <stdio.h>
#include <omp.h>
#include <time.h>

void randomize_array(int array[]);
void print_array(int array[]);

int main(int argc, char* argv[]){
        int tnum=0;

        if(argc != 2){
                printf("Usage %s number_of_thread\n", argv[0]);
                return 1;
        }

        tnum=atoi(argv[1]);
        if(tnum==0){
                printf("Number of thread should be a number\n");
                return 2;
        }

        omp_set_num_threads(tnum);

        int array[100]={0};
        int i=0,j=0,index=0,total=0;
        randomize_array(array);
        printf("-----\n");

        #pragma omp parallel for                                                                   //28
        for(i=0;i<10;i++){                                                                         //29
                total=0                                                                            //30
                for(j=0;j<10;j++){                                                                 //31
                        index=i*10+j;                                                              //32
                        total+=array[index];                                                       //33
                        printf("%2d ", array[index]);                                              //34
                }                                                                                  //35
                printf("= %2.2f  [%d]\n",total/10.0, omp_get_thread_num());                        //36
        }                                                                                          //37
}

/* Randomize array elements */
void randomize_array(int array[]){
        int i=0,j=0,total=0;
        unsigned int iseed = (unsigned int)time(NULL);
        srand (iseed);
        for(i=0;i<10;i++){
                total=0;
                for(j=0;j<10;j++){
                        array[i*10+j]=rand()%100;
                        total+=array[i*10+j];
                        printf("%2d ", array[i*10+j]);
                }
                printf(" AVE = %2.2f\n",total/10.0);
        }
}

/* Print out the array */
void print_array(int array[]){
        int i=0,j=0;
        for(i=0;i<100;i++){
                printf("%2d ", array[i]);
                if(i%10==9)
                        printf("\n");
        }
}
```

Figure 24: Averaging students' scores program; randomize_array() will generate random student scores and pre-computed averages. The actual computation codes are within lines 28-37

```
$./ompGroupAve 1
78 48 84  7 23 25 23 35 27 29  AVE = 37.90
 0 32 21  4 72 92 56 69 21 94  AVE = 46.10
89 88 38 43 38 17 20 22 11 63  AVE = 42.90
82 89 63 18 97 86 95 72 73 23  AVE = 69.80
 1 74  7 74 30 31 66 87  1 39  AVE = 41.00
33 42 80 72 86 18 89 58 92  0  AVE = 57.00
21 26 41 36 45 90 22 40 63 95  AVE = 47.90
15 16 21 74 91 52  6  9 91 59  AVE = 43.40
49 24 53 29 96 39 99 37 49 44  AVE = 51.90
89 22 70 31 58 67 21 80 60 36  AVE = 53.40
-----
78 48 84  7 23 25 23 35 27 29 = 37.90  [0]
 0 32 21  4 72 92 56 69 21 94 = 46.10  [0]
89 88 38 43 38 17 20 22 11 63 = 42.90  [0]
82 89 63 18 97 86 95 72 73 23 = 69.80  [0]
 1 74  7 74 30 31 66 87  1 39 = 41.00  [0]
33 42 80 72 86 18 89 58 92  0 = 57.00  [0]
21 26 41 36 45 90 22 40 63 95 = 47.90  [0]
15 16 21 74 91 52  6  9 91 59 = 43.40  [0]
49 24 53 29 96 39 99 37 49 44 = 51.90  [0]
89 22 70 31 58 67 21 80 60 36 = 53.40  [0]
```

Figure 25: Output of the program running in single thread. The first part of output is the pre-computed average during random scores generation. The second part is the actual computation output.

The first part of the program output is in a single-threaded environment. The first part of the output is the randomly generated number with the pre-computed average, which will be used to check the accuracy of later experiments. The second part of the output is the actual computation. The number in the bracket indicates which thread is used to compute averages by using the `omp_get_thread_num()` function. The first output shows that computation is correct by checking the result against the pre-computed result manually.

Below is the result of the program running with 2 threads, 4 threads, and 8 threads, consecutively. The result shows that single-threaded program output is different than the multi-threaded program. Disregarding the output format, we can see that the averages are different between runs with a different number of threads. The calculations are off for all rows.

```
ant@hp-m8100y:~/school/cs297$ ./ompGroupAve 2
 80 48 94 91  2 82 29 76 46 24  AVE = 57.20
 43 12 61 74 26 81 36  1 29 97  AVE = 46.00
 17 21 76 35 30 44 35 59 45 53  AVE = 41.50
 34 25 53 29 68 55 11 49 83  9  AVE = 41.60
 73 79 74 86 53  0 19 42 53 49  AVE = 52.80
 39 70 70 15 58 52 60 93 63  5  AVE = 52.50
 98 98 82  3 79  2 10 90 51 94  AVE = 60.70
 51 25 73 25 63 26 77 35 68 30  AVE = 47.30
 84 60 53  6 27 63 10 39  8 74  AVE = 42.40
 96  6 72 78 61 51 32 71 93 84  AVE = 64.40
 -----
 80 48 94 91  2 82 29 76 46 24 = 53.10  [0]
 43 12 61 74 26 81 39 93 63  5 36 = 49.40  [1]
 98 98 82  3 79  2 10 90 = 49.40  [0]
 17 21 76 35 30 44 35 51 59 51 53 = 42.10  [0]
 34 25 53 29 68 55 11 49 = 42.10  [1]
 51 25 73 25 63 26 77 83 83  9 35 = 46.70  [0]
 73 79 74 86 53  0 19 42 = 46.70  [1]
 84 60 53  6 27 63 10 53 53 49 39 = 44.40  [0]
 = 44.40  [1]
 96  6 72 78 61 51 32 71 93 84 = 64.40  [1]
 ant@hp-m8100y:~/school/cs297$ ./ompGroupAve 4
 30 34 71 67 77 81 90 10 32 27  AVE = 51.90
 23 77  0 93 16 96 11 20 49 55  AVE = 44.00
 68 96 65 22 25 18 72  1 81 53  AVE = 50.10
 56 11 39 79 78 68 60 20 78 45  AVE = 53.40
 47 54 22 99 47 38 48 58 11 49  AVE = 47.30
 13 79 45 78  1 22 97 73 24 30  AVE = 46.20
 78 32 41 69 11 19 38 24 91 68  AVE = 47.10
 69 91 22 43 90 70 34 38 28 45  AVE = 53.00
 88 94 24 85 24 77  8 73 51 84  AVE = 60.80
  3 81 16 44 51 79 15 41  3 59  AVE = 39.20
 -----
 30 34 71 67 77 81 90 10 32 27 = 54.80  [0]
  3 11 39 79 78 68 60  3  3 59 = 44.00  [3]
 78 = 44.00  [2]
 69 91 22 43 23 96 11 20 49 55 = 54.60  [0]
 68 96 65 22 25 90 18 34 28  1 = 40.20  [0]
 20 = 40.20  [1]
 47 54 22 99 47 38 48 45 28 45 = 48.60  [2]
 88 94 24 85 24 77  8 73 58 51 49 = 57.30  [2]
 = 57.30  [1]
 13 79 45 78  1 22 97 73 24 30 = 46.20  [1]
 ant@hp-m8100y:~/school/cs297$ ./ompGroupAve 8
 65 86 84 57 80 39 95 21 15 85  AVE = 62.70
  9 19  2 86 57 83 46 85 99 16  AVE = 50.20
 13 50 26 75 15 96 59 33 47 57  AVE = 47.10
 83 64 95 67 21 75 58 68 97 25  AVE = 65.30
  6 58 45  8 96  2 92 43 39 43  AVE = 43.20
 11 53 93 89 28 60 38 40 94 85  AVE = 59.10
 97 77 49 93 96 23 20  7 43 69  AVE = 57.40
 84 49 28 29 58 24 84  2 19 23  AVE = 40.00
 45 31 76 91 72 57 51 10 97 97  AVE = 62.70
 95 46 26 97 39 75 72 12 34 15  AVE = 51.10
 -----
 13 97 77 77 49 75 96 65 86 84 57 80  6 45 92 10 39 39 20 = 62.90  [2]
 11 53 93 89 28 60 38 97 96 97 97 = 60.60  [4]
 95 46 26 97 39 75 72 12 40 = 62.90  [0]
  9 19  2 86 57 83 46 85 = 60.60  [1]
 83 64 95 67 21 75 58 68 97 25 = 65.30  [1]
 99 = 65.30  [0]
 = 62.90  [3]
 84 49 28 34 29 39 24 72  2 34 23 = 38.40  [3]
 = 38.40  [4]
 85 = 38.40  [2]
```

Figure 26: All averages from running with 2, 4, 8 threads are different compared to the pre-computed values

The problem is caused by shared variables `total`, `index`, `j`, and because they are shared by all threads, updated values on one thread affects the other. Although loop variables are private by default, inner nested loop variables are not. So, a good idea is to make `total`, `index`, `j` to be private explicitly. There are several changes in the second revision of the program. Beside output formatting changes for readability, the most important change is the omp pragma that sets `i`, `j`, `total`, `index` to be private at line 28 in Figure 27. By making `i`, `j`, `total`, `index` private, each thread can access array through index variable without interfering with other threads' index value. With this setting, each thread can fully iterate 10 consecutive score correctly.

```
#include <stdio.h>
#include <omp.h>
#include <time.h>

void randomize_array(int array[]);
void print_array(int array[]);

int main(int argc, char* argv[]){
       int tnum=0;

       if(argc != 2){
               printf("Usage %s number_of_thread\n", argv[0]);
               return 1;
       }
       tnum=atoi(argv[1]);
       if(tnum==0){
               printf("Number of thread should be a number\n");
               return 2;
       }
       printf("------------\nNumber of cores: %d\n------------\n",omp_get_num_procs());
       omp_set_num_threads(tnum);

       int array[100]={0};
       int i=0, j=0, index=0, total=0;
       randomize_array(array);
       printf("------------\n");
                                                                                   //27
       #pragma omp parallel for private(i,j,total,index)                           //28
       for(i=0;i<10;i++){                                                          //29
               total=0;                                                            //30
               for(j=0;j<10;j++){
                       index=i*10+j;
                       total+=array[index];
               }
               printf("AVERAGE = %2.2f [TheadID: %d]\n",total/10.0, omp_get_thread_num());
       }

}
/*
 * Randomize array elements
 */
void randomize_array(int array[]){
       int i=0,j=0,total=0;
       unsigned int iseed = (unsigned int)time(NULL);
       srand (iseed);
       for(i=0;i<10;i++){
               total=0;
               for(j=0;j<10;j++){
                       array[i*10+j]=rand()%100;
                       total+=array[i*10+j];
                       printf("%2d ", array[i*10+j]);
               }
               printf(" AVE = %2.2f\n",total/10.0);
       }
}

/*
 * Print out the array
 */
void print_array(int array[]){
       int i=0,j=0;
       for(i=0;i<100;i++){
               printf("%2d ", array[i]);
               if(i%10==9)
                       printf("\n");
       }
}
```

Figure 27: Add OpenMP directive to specify i, j, total, index as private at line 28

```
ant@antux:~/school/cs297$ ./ompPPGroupAvg 4
------------
Number of cores: 2
------------
72  3 25 43 32 46 16 48 88 93  AVE = 46.60
55 88 16 65 55 41 98  0 32 38  AVE = 48.80
 0  5 97 31 45 38 76  5 92 14  AVE = 40.30
56 65 69 82  8 53 28 76  1 16  AVE = 45.40
69  9 56 38 74 12 31 24 12 63  AVE = 38.80
14 65 20 11 96 18  2 72 23 94  AVE = 41.50
38 31 59  8 13 20 61 94 96 63  AVE = 48.30
62 18 24 19  8 50 83 39 74 47  AVE = 42.40
54 40 12 75  3 60 93  5 33 68  AVE = 44.30
 0 71 99 59 31 65 31 93 59 80  AVE = 58.80
------------
AVERAGE = 46.60  [ThreadID: 0]
AVERAGE = 48.80  [ThreadID: 0]
AVERAGE = 40.30  [ThreadID: 0]
AVERAGE = 58.80  [ThreadID: 3]
AVERAGE = 48.30  [ThreadID: 2]
AVERAGE = 42.40  [ThreadID: 2]
AVERAGE = 44.30  [ThreadID: 2]
AVERAGE = 45.40  [ThreadID: 1]
AVERAGE = 38.80  [ThreadID: 1]
AVERAGE = 41.50  [ThreadID: 1]
ant@antux:~/school/cs297$ ./ompPPGroupAvg 8
------------
Number of cores: 2
------------
53 88 37 63 39 37 50 17 15  3  AVE = 40.20
96 19 27 75 54 83 51 68 37  4  AVE = 51.40
91 47 35 99 96 90 17 89 96 23  AVE = 68.30
89  1 63 26 17  2 16 67 72 83  AVE = 43.60
22 20 54  1 95  8 85 98 28 74  AVE = 48.50
55 71 21 90 71 69 32 40 58 80  AVE = 58.70
15 99 34 78 26 51 33 94 70  5  AVE = 50.50
77 93 25 32 94 72 92 31 22 73  AVE = 61.10
 5 77 44 78 19 67 47  3 60 57  AVE = 45.70
84 75 57 18  6 35 21 39 81 91  AVE = 50.70
------------
AVERAGE = 68.30  [ThreadID: 1]
AVERAGE = 43.60  [ThreadID: 1]
AVERAGE = 48.50  [ThreadID: 2]
AVERAGE = 58.70  [ThreadID: 2]
AVERAGE = 50.50  [ThreadID: 3]
AVERAGE = 61.10  [ThreadID: 3]
AVERAGE = 45.70  [ThreadID: 4]
AVERAGE = 50.70  [ThreadID: 4]
AVERAGE = 40.20  [ThreadID: 0]
AVERAGE = 51.40  [ThreadID: 0]
ant@antux:~/school/cs297$
```

Figure 28: Output of OpenMP program after making variables (i, j, total, index) private

The next program is the sequence number generator. The program will generate sequence numbers 10,000 times in a loop, and we tell the compiler that we want to run the loop in parallel with a different number of threads from user input. In this program, threads will race each other to get the next sequence number using OpenMP threads. One or more threads may get the same consecutive number if they happen to perform increment operation at the same time.

```
#include <stdio.h>
#include <omp.h>
#include <time.h>

int main(int argc, char* argv[]){
        int tnum=0;

        if(argc != 2){
                printf("Usage %s number_of_thread\n", argv[0]);
                return 1;
        }

        tnum=atoi(argv[1]);
        if(tnum==0){
                printf("Number of thread should be a number\n");
                return 2;
        }

        printf("------------\nNumber of cores: %d\n------------\n",omp_get_num_procs());
        omp_set_num_threads(tnum);

        int seq = 0,i=0;

        #pragma omp parallel for
        for(i=0; i<10000; i++){
                seq++;
        }
        printf("%d\n",seq);
}
```

Figure 29: OpenMP implementation of sequence number generator

The program output in Figure 30 shows it running in 2, 4, and 8 threads, respectively. The expected final number should be 10,000. However, the result shows different value for the last request. Race condition causes multiple threads getting the same number; therefore, the number increases incorrectly.

```
ant@antux:~/school/cs297$ ./ompSeqNum 2
------------
Number of cores: 2
------------
9695
ant@antux:~/school/cs297$ ./ompSeqNum 4
------------
Number of cores: 2
------------
8446
ant@antux:~/school/cs297$ ./ompSeqNum 8
------------
Number of cores: 2
------------
8913
ant@antux:~/school/cs297$
```

Figure 30: Race condition causes incorrect and inconsistent results

Making `seq++` to be an atomic statement by inserting `#pragma omp atomic`, the atomic statement can only be executed by only one thread at a time. The atomic directive was added to the program at line 25 in Figure 31.

```
 #include <stdio.h>
 #include <omp.h>
 #include <time.h>


 int main(int argc, char* argv[]){
       int tnum=0;

       if(argc != 2){
              printf("Usage %s number_of_thread\n", argv[0]);
              return 1;
       }
       tnum=atoi(argv[1]);
       if(tnum==0){
              printf("Number of thread should be a number\n");
              return 2;
       }
       printf("------------\nNumber of cores: %d\n------------\n",omp_get_num_procs());
       omp_set_num_threads(tnum);

       int seq = 0,i=0;

       #pragma omp parallel for
       for(i=0; i<10000; i++){                                                   //Line 24
              #pragma omp atomic                                                 //Line 25
              seq++;                                                             //Line 26
       }                                                                         //Line 27
       printf("%d\n",seq);
 }
```

Figure 31: Making seq++ to be executed atomically by adding '#pragma omp atomic'

```
ANHs-MacBook-Pro:cs297 anhtrinh$ ./ompSeqNumAtomic 2
 ------------
 Number of cores: 2
 ------------
 10000
 ANHs-MacBook-Pro:cs297 anhtrinh$ ./ompSeqNumAtomic 4
 ------------
 Number of cores: 2
 ------------
 10000
 ANHs-MacBook-Pro:cs297 anhtrinh$ ./ompSeqNumAtomic 8
 ------------
 Number of cores: 2
 ------------
 10000
```

Figure 32: Results after adding atomic directives to seq++

7. X10 Proposed Solution

X10 is the recent experimental language under development at IBM Research since 2004 in collaboration with several academic partners. X10 is a member of Partitioned Global Address Space (PGAS). Claiming it is a type-safe, parallel language, X10 aims at parallel system with multi-core SMP nodes interconnected in scalable cluster configurations. X10 means ten-fold productivity increases. It is designed for the modern multi-core era and for traditional clustered architectures. X10 supports execution across multiple address spaces, also global object model. Even though X10 has been under development for years, it is still in the early stages, and is still considered unstable. Library support, interoperability, IDE, compiler performance, and runtime performance of certain X10 idioms have their known issues. Besides claiming itself as type-safe, X10 also guarantees that it will never have a logical deadlock with `async, finish, atomic` operations [18, 24].

X10 compiler has 2 modes. One is Java-backend and the other is C++-backend. As shown in the Figure 33, the X10 compiler will take in X10 source code and parse it into Abstract Syntax Tree, and then generate C++ or Java output files per user choice. The output files are then passed to a post-compiler to generate executable/class files as the final result of the compilation process. These final output files can either execute in JVM (Java) or natively (C++).

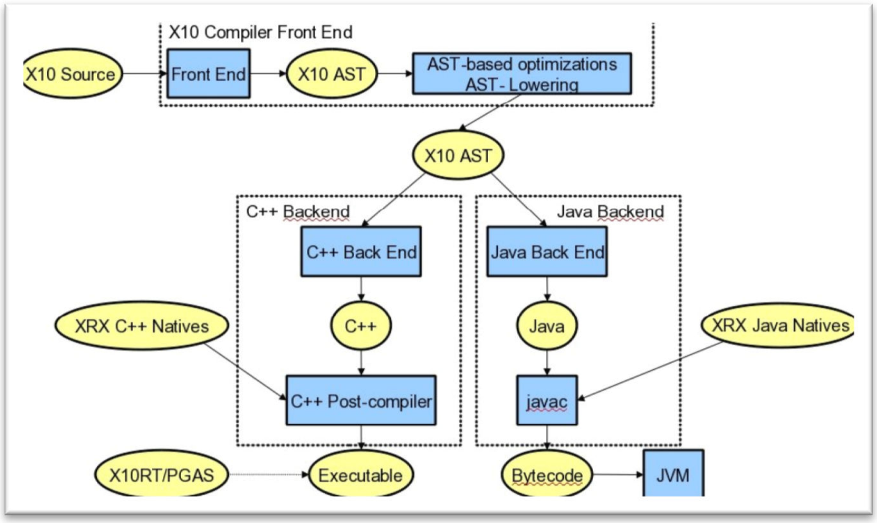


Figure 33: X10 compiling architecture

7.1. Concurrency constructs

X10 has a limited number of concurrency constructs to ease multi-threaded program's development. The followings are the most basic concurrency constructs that a multi-thread program would use in X10.

- `async`: Spawning an activity

By executing `async(s)`, the program will spawn an activity – another word for a thread in X10 language – and execute statement `s` in parallel. While executing `s` in parallel, main will continue and proceed with the next code statement [4].

In the sample HelloWorld.x10 above, the program is compiled with Java-backend. The main thread executes the first `println()` at line 6, then it spawns another thread executing `println(` at line 7, then continues without waiting for the spawning activity to finish.

```
import x10.io.*;
import x10.util.*;

class HelloWorld{
      public static def main(args:Array[String](1)){
              x10.io.Console.OUT.println("Hello World 1");             //6
              async{                                                   //7
                      x10.io.Console.OUT.println("Hello World 2");     //8
                      x10.io.Console.OUT.println("Goodbye 2");         //9
              }                                                        //10
              x10.io.Console.OUT.println("Goodbye 1");
      }
}
```

Figure 34: HelloWorld program; the main activity spawn new activity to do println()

```
ant@antux:~/school/cs297/x10$ x10c HelloWorld.x10
ant@antux:~/school/cs297/x10$ x10 HelloWorld
Hello World 1
Goodbye 1
Hello World 2
Goodbye 2
```

Figure 35: HelloWorld program output

- `finish`

The statement `finish S` converts global termination to local termination. An activity `A` executes `finish S` by executing `S` and then it waits for all activities spawned by `S` to terminate [4].

A revised version of HelloWorld.x10 in Figure 36 with `finish` has different execution flow. Main executes first `println()` then it spawns another activity to executes other `println()` in the `async` block. Meanwhile, main waits for the spawning activity to finish before it can proceed and execute the rest of the codes.

```
import x10.io.*;
import x10.util.*;

class HelloWorld{
        public static def main(args:Array[String](1)){
                finish{
                        x10.io.Console.OUT.println("Hello World 1");
                        async{
                                x10.io.Console.OUT.println("Hello World 2");
                                x10.io.Console.OUT.println("Goodbye 2");
                        }
                }
                x10.io.Console.OUT.println("Goodbye 1");                                    //13
        }
}
```

Figure 36: Modified HelloWorld making main activity to wait for all spawning thread to finish before it can execute line 13

```
ant@antux:~/school/cs297/x10$ x10c HelloWorld.x10
ant@antux:~/school/cs297/x10$ x10 HelloWorld
Hello World 1
Hello World 2
Goodbye 2
Goodbye 1
```

Figure 37: HelloWorld program output with finish statement

- `atomic:` Atomic Blocks

X10 developers believe locks are low-level construct and error-prone synchronization mechanism, which makes it very easy for a program to cause race conditions and deadlocks [24]. X10's atomic blocks provide a high-level mechanism to obtain mutual exclusion. Programmers can use atomic blocks to ensure shared data are maintained even though the data are being accessed by multiple processes simultaneously [4].

The code snippet in Figure 38 will execute loops. For each loop iteration will spawn an activity. Each activity will race to execute line 7, which is the important line of the program. By using atomic, X10 ensures that statements in atomic block will be executable atomically without any interference.

```
import x10.io.*;
import x10.util.*;
class AtomicBlock{
        public static def main(args:Array[String](1)){
                var v:int = 0;
                finish for([i] in 1..4) async{
                        atomic{                                         //7
                                v = i;                                  //8
                                x10.io.Console.OUT.println(i + " " + v);      //9
                        }                                               //10
                }
        }
}
```

Figure 38: X10 program tests atomic block

```
ant@antux:~/school/cs297/x10$ x10c AtomicBlock.x10
 ant@antux:~/school/cs297/x10$ x10 AtomicBlock
 1 1
 2 2
 3 3
 4 4
```

Figure 39: Testing atomic block program output

- `when:` Conditional Atomic Blocks

Conditional atomic block allows an activity to wait for certain conditions to be met before executing later code statements. However, conditional atomic block is not a guarantee if the conditions hold intermittently [4].

7.2. X10 code examples

The following programs in this section demonstrate the usage of X10 concurrency constructs by re-implementing above programming examples in the Java implementations and OpenMP implementations sections. All the programs will be compiled and run using Java-backend.

A banking problem, where multiple threads are trying to deposit or to withdraw from an account at the same time, demonstrates race condition handling in X10. The program has `Holder` class, deposit and withdraw activities through `Account` API, which are `deposit(amount:int)` and `withdraw(amount:int)`. Within each method, add and subtract operations are set to atomic statement as in line 11 and 15, as in Figure 40, to ensure that only one process access functions one at a time.

```
import x10.io.Console;
public class Banking{
        public static class Account{
                var id:int;
                var balance:int;
                public def this(id:int, balance:int){
                        this.id = id;
                        this.balance = balance;
                }
                public def withdraw(amount:int):int{
                        atomic balance -= amount;                                   //11
                        return balance;                                             //12
                }                                                                   //13
                public def deposit(amount:int):int{                                 //14
                        atomic balance += amount;                                   //15
                        return balance;
                }
                public def balance():int{ return balance;}
        }

        public static class Holder{
                var name:String;
                var acct:Account;

                public def this(name:String, acct:Account){
                        this.name = name;
                        this.acct = acct;
                }
                public def deposit(amount:int){ var balance:int =this.acct.deposit(amount);}
                public def withdraw(amount:int){var balance:int =this.acct.withdraw(amount);}
        }
        public static def main(args:Array[String](1)){
                var acct:Account = new Account(1000, 100000);
                var family:Holder = new Holder("Family", acct);
                Console.OUT.println("Initial Balance: 100000");
                Console.OUT.println("-----------------");
                Console.OUT.println("4 threads each will add $1 for 100000 times (total
$100000) to account balance.");
                finish for(1..4) async{
                        for(1..100000){ family.deposit(1); }
                }
                Console.OUT.println("Balance: " + acct.balance());
                Console.OUT.println("-----------------");
                Console.OUT.println("4 threads each will subtract $1 for 100000 times (total
$100000) to account balance.");
                finish for(1..4)async{
                        for(1..100000){family.withdraw(1);  }
                }
                Console.OUT.println("Balance: " + acct.balance());
                Console.OUT.println("-----------------");
                Console.OUT.println("2 threads P1 will withdraw $1 for 1000 times, P2 will
deposit $1 for 1000 times");
                var p1:Holder = new Holder("P1", acct);
                var p2:Holder = new Holder("P2", acct);
                finish{
                        async for(1..1000) p1.withdraw(1);
                        async for(1..1000) p2.deposit(1);
                }
                Console.OUT.println("Balance: " + acct.balance());
        }
  }
```

Figure 40: Banking example – multiple threads access one account simultaneously. Each thread will deposit $1 to the account 10,000 times

```
ant@antux:~/school/cs297/x10$ x10c Banking.x10
ant@antux:~/school/cs297/x10$ x10 Banking
Initial Balance: 100000
-----------------
4 thread each will add $1 for 100000 times (total $100000) to account balance.
Balance: 500000
-----------------
4 thread each will subtract $1 for 100000 times (total $100000) to account balance.
Balance: 100000
-----------------
2 thread P1 will withdraw $1 for 1000 times, P2 will deposit $1 for 1000 times
Balance: 100000
```

Figure 41: Banking example program output for various scenarios

Extending the banking example, the program below simulates money transferring between 2 accounts and 2 account owners. Each account initially has $1,000,000 as its balance. Simultaneously, each owner will transfer $500,000 from his account to the other account. The program also utilizes X10 atomic blocks, but this time we set it as method level instead of individual statement. The first two lines of the output show the initial balance of the accounts; the last two lines show the balance after the program terminates. Both parts of the output give the same result.

```
import x10.io.*;
import x10.io.Console;
import x10.util.*;

class BanksTransactions{
	public static class Account{
		var balance:int = 1000000;
		var accid:int = 0;

		public def this(accid:int){
			this.accid = accid;
		}

		public atomic def deposit(amount:int){                        //Line 11
			this.balance += amount;
		}

		public atomic def withdraw(amount:int){                       //Line 15
			this.balance -= amount;
		}

		public atomic def balance(){
			return this.balance;
		}
	}

	public static  class Owner{
		var myacct: Account;
		var uracct: Account;
		var name: String;

		public def this(myacct: Account, uracct:Account, name:String){
			this.myacct = myacct;
			this.uracct = uracct;
			this.name = name;
		}

		public def transfer(amount: int){
			this.myacct.withdraw(amount);
			this.uracct.deposit(amount);
		}
	}

	public static def main(args:Array[String](1)){
		var account1:Account = new Account(1);
		var account2:Account = new Account(2);
		Console.OUT.println("Acct1: " + account1.balance());
		Console.OUT.println("Acct2: " + account2.balance());

		finish{
			val owner1:Owner = new Owner(account1, account2, "A");
			val owner2:Owner = new Owner(account2, account1, "B");
			async{
				for(1..500000){
					owner1.transfer(1);
				}
			}
			async{
				for(1..500000){
					owner2.transfer(1);
				}
			}
		}
		Console.OUT.println("Acct1: " + account1.balance());
		Console.OUT.println("Acct2: " + account2.balance());
	}
`
```

Figure 42: BanksTransactions – two people, each has his own account – each simultaneously transfers money from his account to the other

```
ant@antux:~/school/cs297/x10$ x10c BanksTransactions.x10
ant@antux:~/school/cs297/x10$ x10 BanksTransactions
Acct1: 1000000
Acct2: 1000000
Acct1: 1000000
Acct2: 1000000
```

Figure 43: First 2 lines of output show the balance of 2 accounts before transferring. Last 2 lines show the balance after transferring, which is correct because they transfer to each other with the same amount

Duplicating the sequence number generation like in Java and OpenMP, two versions of the X10 program are implemented. One is using AtomicInteger, which is similar to the Java version, and the other uses atomic block. The main thread of the program will spawn four threads. Each will try to get 1000 sequence number simultaneously. If there is no race condition, the value of the last request should be 4000.

```
import x10.io.Console;
import x10.util.concurrent.AtomicInteger;

class AtomicSequenceInteger{
      public static class Generator{
             var seq:AtomicInteger=new AtomicInteger();
             public def this(start:int){
                    seq.set(start);
             }

             public def next(){
                    return seq.getAndIncrement();
             }

             public def current(){
                    return seq.get()-1;
             }
      }

      public static def main(args:Array[String](1)){
             val gen = new Generator(1);
             Console.OUT.println("4 Threads each will try to get 1000 sequence number.");
             finish for(1..4) async{
                    for(1..1000){
                           gen.next();
                    }
             }
             Console.OUT.println(gen.current());
      }
}
```

Figure 44: AtomicSequenceInteger – 4 threads request unique numbers in parallel. Each will ask for 1000 times. The implementation uses AtomicInteger object as solution.

```
ant@antux:~/school/cs297/x10$ x10c AtomicSequenceInteger.x10
ant@antux:~/school/cs297/x10$ x10 AtomicSequenceInteger
4 Threads each will try to get 1000 sequence number.
4000
```

Figure 45: AtomicSequenceInteger program output – indicating the last request having value of 4000

By implementing using atomic block, the `SequenceNumber` mimics the `AtomicInteger` object behavior.

```
import x10.io.Console;
class SequenceNumber{
	public static class Generator{
		var seq:int=0;
		public def this(start:int){
			seq = start;
		}

		public atomic def next(){
			return seq++;
		}

		public def current(){
			return seq-1;
		}
	}

	public static def main(args:Array[String](1)){
		val gen = new Generator(1);
		Console.OUT.println("4 Threads each will try to get 1000 sequence   number.");
		finish for(1..4) async{
			for(1..1000){
				gen.next();
			}
		}
		Console.OUT.println(gen.current());
	}
}
```

Figure 46: X10 implementation of sequence number generator

The final program is the consumer/producer problem. The program `Producer` will read an input and put each character to `Buffer`. `Buffer` is a bounded `ArrayList` with max size of 10 elements. With conditional atomic block, producer can only put characters when buffer is not full. Similarly, consumer can only take one character if the queue is not empty. When producer reaches the end of the file, it will put '\003', indicating that it finished reading the input file and will terminate. When consumer sees '\003', it will also terminate. `Consumer` will fetch the character from `Buffer` and output that into a file. The final result is the replication of the input file. The goal of this program is to test the conditional atomic construct in X10.

```
 import x10.io.*;
 import x10.util.*;
 class ConsumerProducerFile{
        public static class Consumer{
                var buffer:Buffer;
                public def this(buffer:Buffer){ this.buffer = buffer; }
                public def consume(item:Char){
                        if(item == '\003'){ return; }   //HIT END OF FILE INDICATOR
                        val filepath = "ConsumerProducerFile-Copy.x10";
                        var file:File = new File(filepath);
                        var buf:ArrayList[Char] = new ArrayList[Char]();
                        try{
                                if( file.exists() ){
                                        var fr:FileReader = new FileReader(file);
                                        while(fr.available() > 0){
                                                var char:Char = fr.readChar();
                                                buf.add(char);
                                        }
                                        fr.close();
                                }
                                var fw:FileWriter = new FileWriter(file);
                                for(char in buf){ fw.writeChar(char); }
                                fw.writeChar(item);
                                fw.close();
                        }catch(e:Exception){ e.printStackTrace();       }
                }
                public def start(){
                        var item:Char;
                        do{
                                item = this.buffer.removeItem();
                                consume(item);
                        }while(item!='\003');
                        x10.io.Console.OUT.println("Consumer finished.");
                }
        }
         public static class Producer{
                var buffer:Buffer;
                public def this(buffer:Buffer){ this.buffer = buffer; }
                public def start(){
                        var fr:FileReader=new FileReader(new Fle("ConsumerProducerFile.x10"));
                        while(fr.available() > 0){
                                val item:Char = fr.readChar();
                                this.buffer.addItem(item);
                        }
                        this.buffer.addItem('\003');
                        fr.close();
                        x10.io.Console.OUT.println("Producer finished.");
                }
        }
         public static class Buffer{
                var ptr:int=0, size:int;
                var queue:ArrayList[Char];
                public def this(size:int){
                        this.size = size;
                        queue = new ArrayList[Char](size);
                }
                public def addItem(item:Char){ when( queue.size()<size ) queue.add(item); }
                public def removeItem(){ when( queue.size()>0 ) return queue.removeFirst(); }
                public atomic def size(){ return queue.size(); }
        }
         public static def main(arg:Array[String](1)){
                val buffer = new Buffer(10);
                val con = new Consumer(buffer);
                val pro = new Producer(buffer);
                finish{
                        async pro.start();
                        async con.start();
                }
        }
}
```

Figure 47: X10 implementation of consumer/producer problem

```
ant@antux:~/school/cs297/x10$ ls -l ConsumerProducerF*.x10
-rw-r--r-- 1 ant ant 2102 2011-05-25 03:02 ConsumerProducerFile-Copy.x10
-rw-r--r-- 1 ant ant 2102 2011-05-25 01:23 ConsumerProducerFile.x10
ant@antux:~/school/cs297/x10$ diff ConsumerProducerFile.x10 ConsumerProducerFile-Copy.x10
ant@antux:~/school/cs297/x10$
```

Figure 48: Verify consumer/producer program by running 'ls -l' and 'diff' commands

Because the X10 IO package does not provide a way to open a file in append mode, a workaround is implemented. Appending is simulated by opening the file, reading every character into an `ArrayList`, appending the newly read from the `Buffer` queue to the end of `ArrayList`. Then the consumer will write the `ArrayList` of characters back to the file with "-Copy" in the file name. With bounded memory, consumer can only take an item in the `Buffer` when the queue is not empty. Likewise, producer can only put an item into the queue if the queue is not full. Within the class Buffer, two underlined statements in Figure 47 demonstrate the usage of X10 conditional atomic block. Comparing the input and the output of the program by executing 'ls -l' and 'diff', the two files are the same.

Conditional atomic block in X10 is implemented as spinlock. To prove the claim of `when(c)` not to guarantee if c holds intermittently, we write a simple routine. The program contains a `number` variable initialized with 0. Main will spawn two threads: one increases the value of `number` through a loop; the second will wait and check if the number reaches 5, then print out the value. The program runs for several iterations and the results are not the same. Sometimes the program terminates; other times it just hangs. By checking the system task monitor, we can see the CPU spins 100%. So, programmers must use `when(condition)` with care.

```
class TestWhen{
        public static def main(Array[String](1)){
                var number:int=0;
                finish{
                        async{
                                when( number==5 )
                                        Console.OUT.println(number);
                        }
                        async{
                                for(i int 0..10)
                                        number++;
                        }
                }
        }
 }
```

Figure 49: Testing 'when' construct in X10, 'when(c)' fails when 'number++' is instantaneous

8. Performance

For further study of the X10 programming language, this section examines runtime performance of X10 and Java on a single local machine. Since X10 is a “higher-level” language than Java and C++, X10 performance is considered optimal if Java-backend performance is comparatively equal with core Java performance. Hopefully, Native-backend compilation will surpass Java.

To assume X10 has Java and C++ comparable performance is a bit optimistic because X10 will likely generate extra codes adding overhead that an ordinary optimized program would not. Therefore, it is expected that X10 Java-backend will run slower than Java. In addition, X10 runtime environment is an extension of Java runtime environment (JRE); the runtime environment has an impact on Java-backend programs. At the same time, Native-backend may generate extra overhead in C++ code, but Native-backend is expected to run faster than Java because C++ object code does not need to run through a virtual machine.

In this experiment, three basic algorithms were used for performance benchmarking: quick sort algorithm, matrix multiplication, and AES encryption. Each algorithm will run with three different data sets, and each data set will run 25 times and be measured in milliseconds (ms).

All test cases will use the same environment as follows:

- Operating System: Linux Ubuntu version 11.11 (32-bits)
- CPU: Quad-cores 2.4GHz (No hyper-threading)
- Memory: 4GB RAM
- Java version: 6.0SE
- Gcc 4.6

8.1. Quick sort

Quick sort is an O(nlog n) sorting algorithm, which was developed by Tony Hoare. Using a divide-and-conquer approach, the algorithm picks a pivot and rearranges elements into two regions. Larger or equal elements than pivot will stay on right half and smaller will stay on the left half. This process is preferred as partition. The algorithm then recursively repeats the operation for right and left regions until all items are sorted [5, 22].

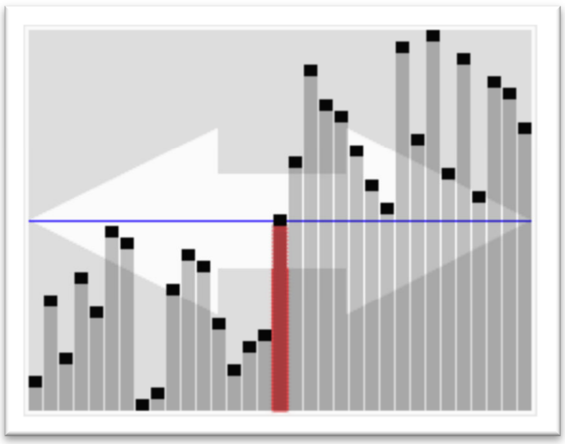


Figure 50: Quick sort partitions [22]

In a sequential version of quick sort, the algorithm will pick one side after first partitioning, splitting that half into another 2 sub regions, and repeat the operation until the smallest region containing 2 elements is sorted. After the two regions on the left furthest side are sorted, the algorithm will work on the sibling regions and work back up. The recursive algorithm will use call stack heavily to keep track of function calls and return.

In a parallel version, sub regions are stacked for later processing; the program will spawn new threads to operate on sub regions in parallel fashion. When all threads finish processing, all sub regions should be sorted. If a system can create an unlimited number of processes, a large divide-and-conquer problem is solved in a short time. In reality, a system always has limitations.

In this experiment, quick sort java-implementation has a small, limited number of processes. The main thread first partitions the original input list, then it splits the list into 2 sub regions and spawns 2 processes where each thread will operate on one region in parallel with the other. Again, each sub region will split and spawn another 2 processes, which will create 4 more processes. The 4 newly created processes will operate as single-threaded quick sort versions on the subset elements. The X10 quick sort program will have total of 7 processes as illustrated below.

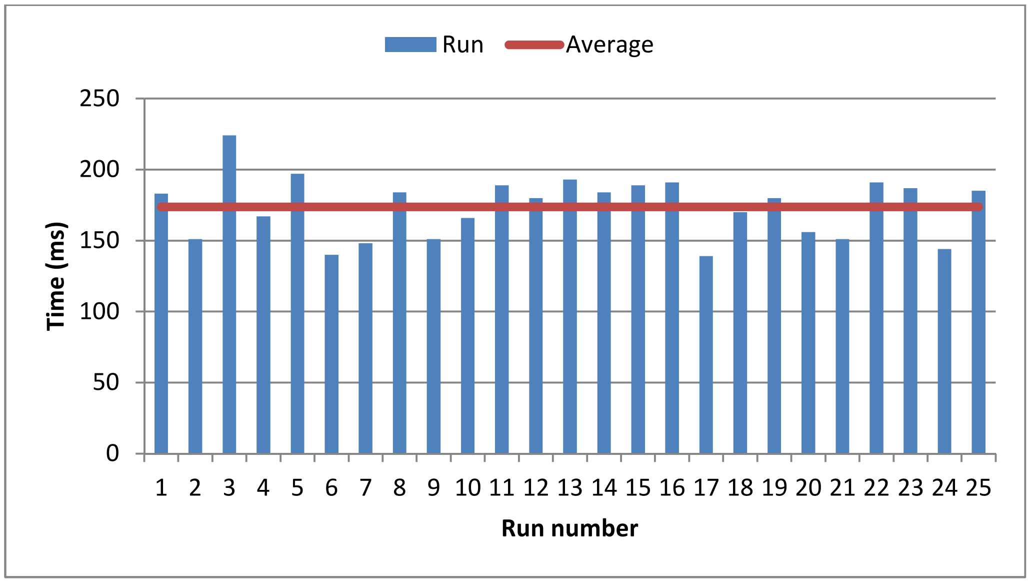
15, 5, 11, 0, 1, 4, 2, 8, 10, 3, 9, 13, 7, 6, 12, 14				1st
6, 5, 7, 0, 1, 4, 2, 3		10, 8, 9, 13, 11, 15, 12, 14		2nd
A[...]	B[...]	C[...]	D[...]	3rd

There are three test cases for sorting random numbers for each language and compilation mode. Each test case will sort 1, 10, and 100 million of random-generated numbers ranging from 0 to 2^{31}-1 (max value of integer). Normally, a system always has many OS processes running in the background, so putting a timer in the program may not get accurate elapse time. Therefore, this experiment will run the program multiple times, then take the mean of all results. Because this experiment is about comparing performance between two languages, the conclusion can be drawn as long as both programs run under the same environment.

8.1.1. Java

The following figures illustrate the runtime results of quick sort using Java implementation. They show the runtime results of sorting 1, 10, and 100 million random numbers 25 times.

The Figure 51 shows the program taking 139 ms at minimum and 224 ms at maximum to sort. On average, the program takes 173.6 ms to finish sorting 1 million numbers. (For more details, see Appendix A.)



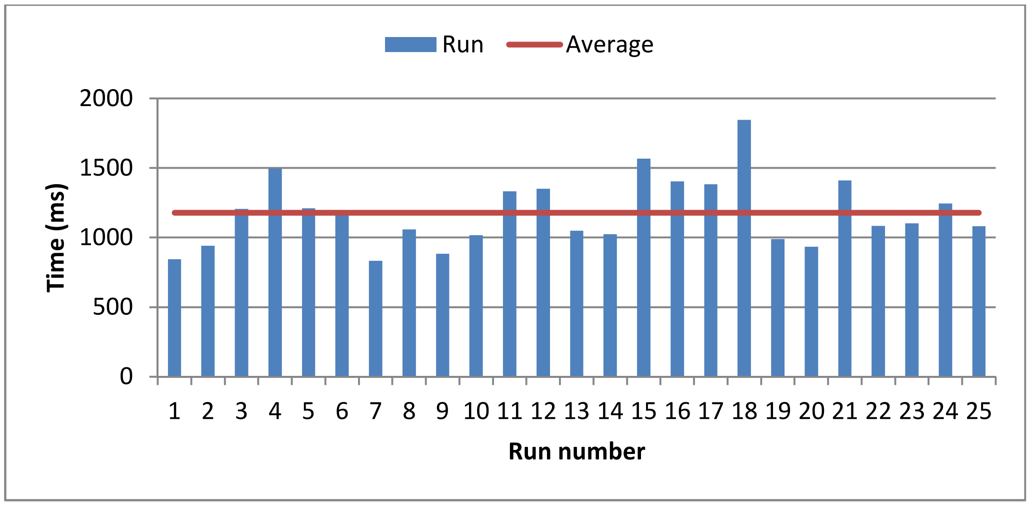
Min: 139

Max: 224

Mean: 173.6

Figure 51: Java quick sort - sorting 1 million numbers

Figure 52 shows the runtime result of sorting 10 million numbers using Java implementation. The results show at minimum the program taking 833 ms, and at maximum taking 1846 ms to finish. On average, it takes 1179.16 ms. (For more details, see Appendix A.)



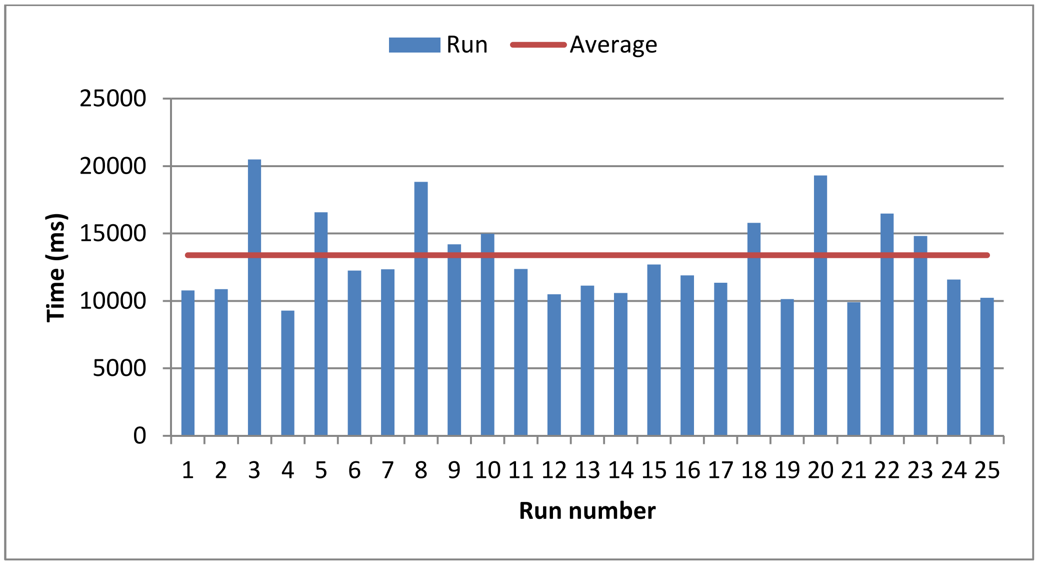
Min 833

Max: 1846

Mean: 1179.16

Figure 52: Java quick sort - sorting 10 million numbers

Figure 53 shows the program taking 9282 ms at minimum, and taking 20502 ms at maximum to sort. On average, the program takes about 13380.26 ms to finish sorting. (For more details, see Appendix A.)



Min: 9282

Max: 20502

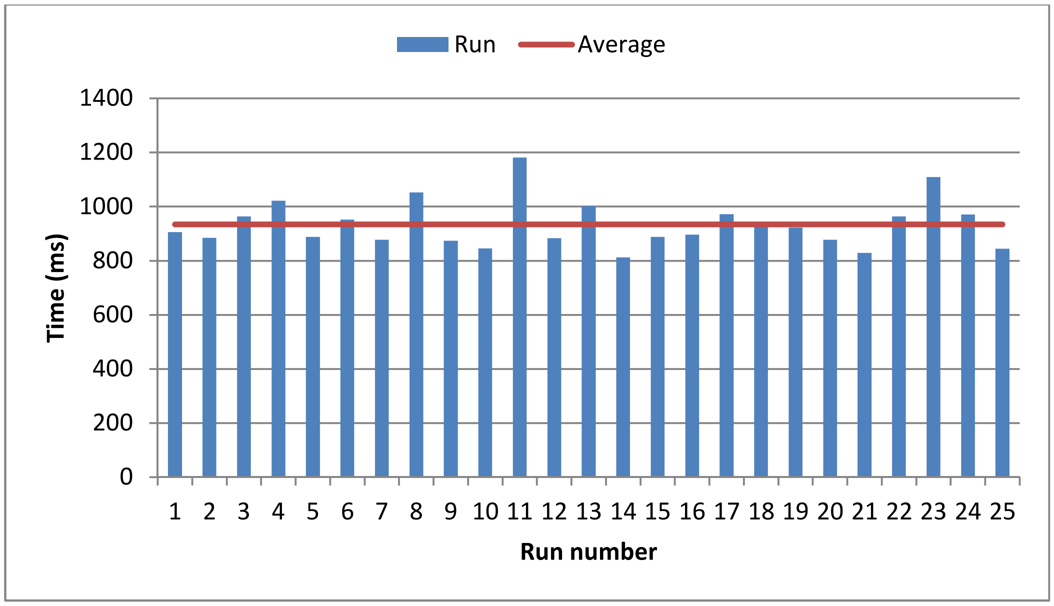
Mean: 13380.26

Figure 53: Java quick sort - sorting 100 million numbers

8.1.2. X10 Java-backend

The following figures illustrate the runtime results of quick sort using X10 implementation with Java-backend compilation. This experiment uses X10 implementation from a tutorial found at X10's official website [27].

Figure 54 shows the program taking 813 ms at minimum, and 1182 ms at maximum. On average, the program takes 924.52 ms to finish based on 25 runs. (For more details, see Appendix A.)



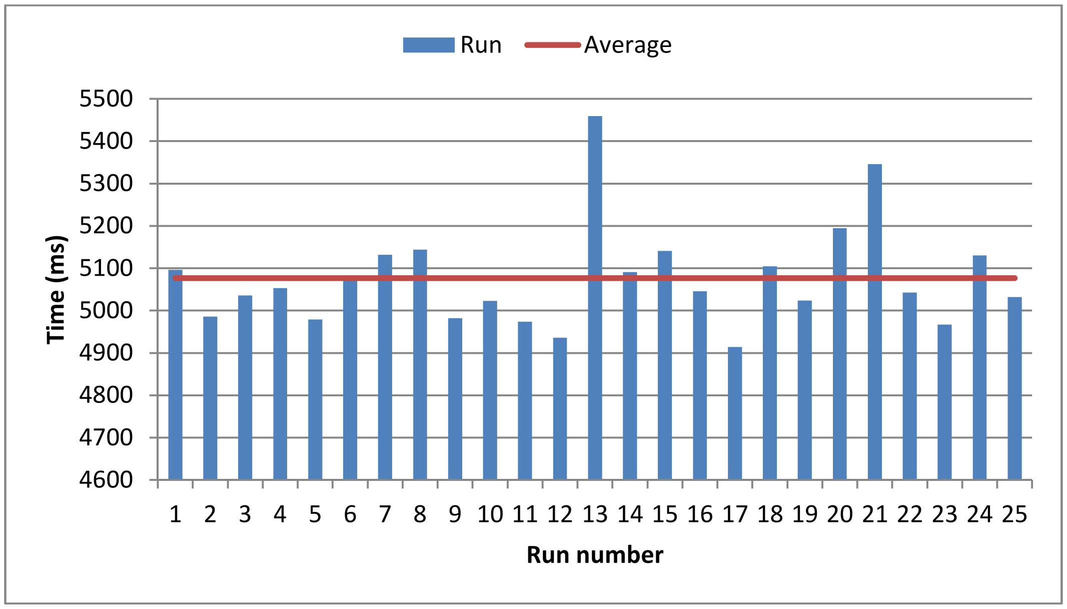
Min: 813

Max: 1182

Mean: 934.52

Figure 54: X10 Java-backend - sorting 1 million numbers

Figure 55 shows the runtimes of sorting 10 million numbers. The program takes at least 4914 ms, and at maximum 5459 ms to finish sorting. On average, the program takes 5076.32 ms to sort. (For more details, see Appendix A.)



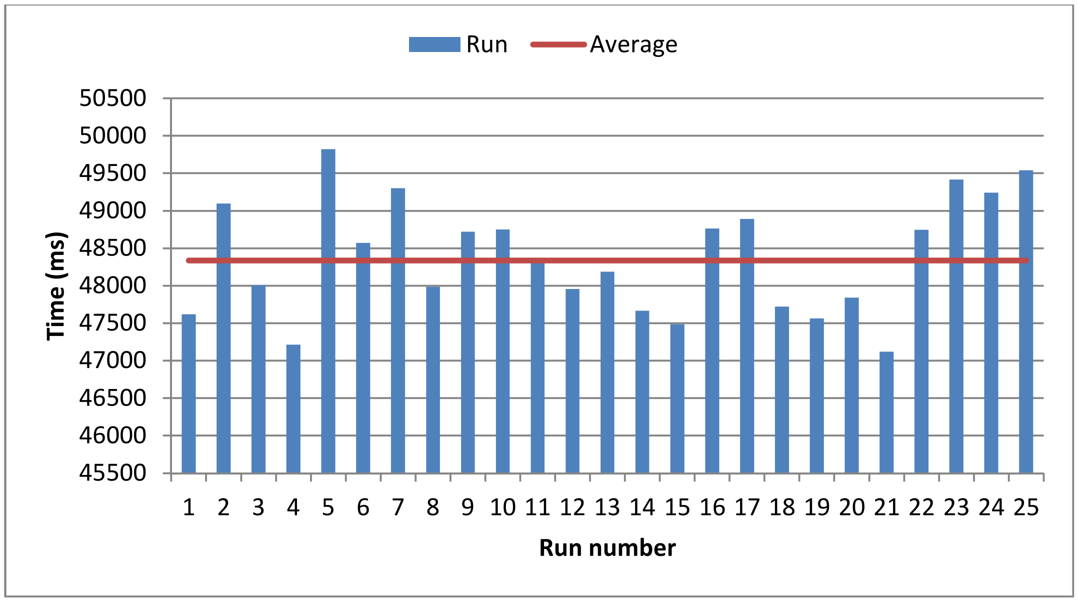
Min: 4914

Max: 5459

Mean: 5076.32

Figure 55: X10 Java-backend - sorting 10 million numbers

Figure 56 shows the runtimes of sorting 100 million numbers. The program takes 47120 ms at the minimum and 49822 ms at the maximum to finish. On average, the program takes 48335.21 ms to finish sorting. (For more details, see Appendix A.)



Min: 47120

Max: 49822

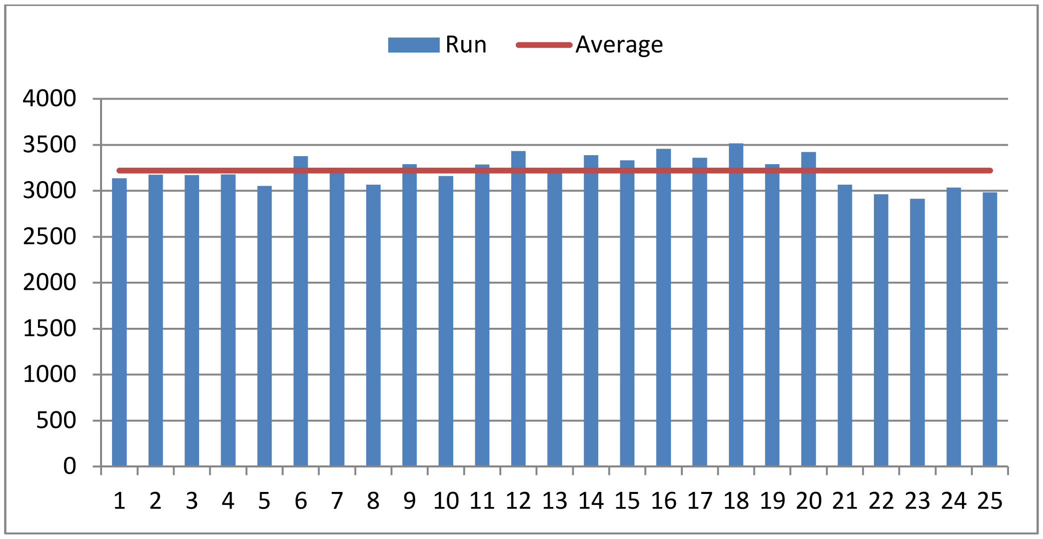
Mean: 48335.21

Figure 56: X10 Java-backend - sort 100 million numbers

8.1.3. X10 C++-backend

The following figures illustrate the runtime results of quick sort using X10 implementation with C++-backend compilation. This experiment also uses the same X10 implementation from a tutorial found at X10's official website, except that the program is compiled using C++-backend.

Figure 57 shows the runtime results of sorting 1 million numbers using X10 C++-backend. The program takes 2912 ms at minimum, and 3518 ms at maximum to finish sorting. On average, the program takes 3220.92 ms. Looking at the graph, the runtime is more uniform compared to other Java implementation and Java-backend, but it is slower. (For more details, see Appendix A.)



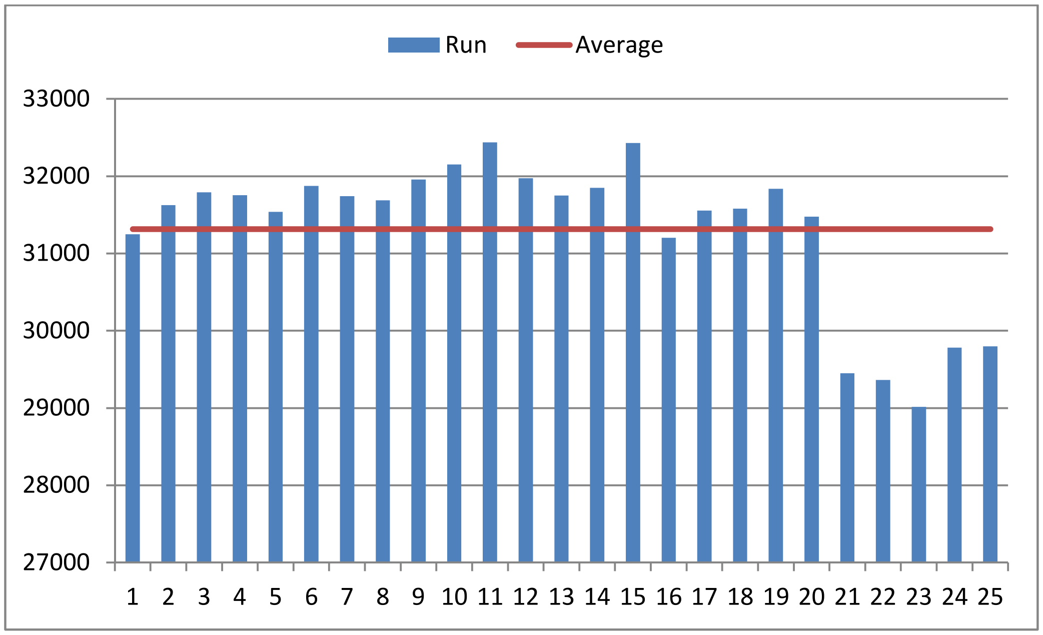
Min: 2912

Max: 3518

Mean: 3220.92

Figure 57: X10 C++-backend – sorting 1 million numbers

Figure 58 shows the runtime result of sorting 10 million numbers. The program takes 32436 ms at the maximum and 29016 ms at the minimum. On average, the X10 C++-backend takes 31314.8 ms to finish sorting. The last 5 runs seem to be faster the first 20, yet it is much slower compared to Java implementation or Java-backend. (For more details, see Appendix A.)



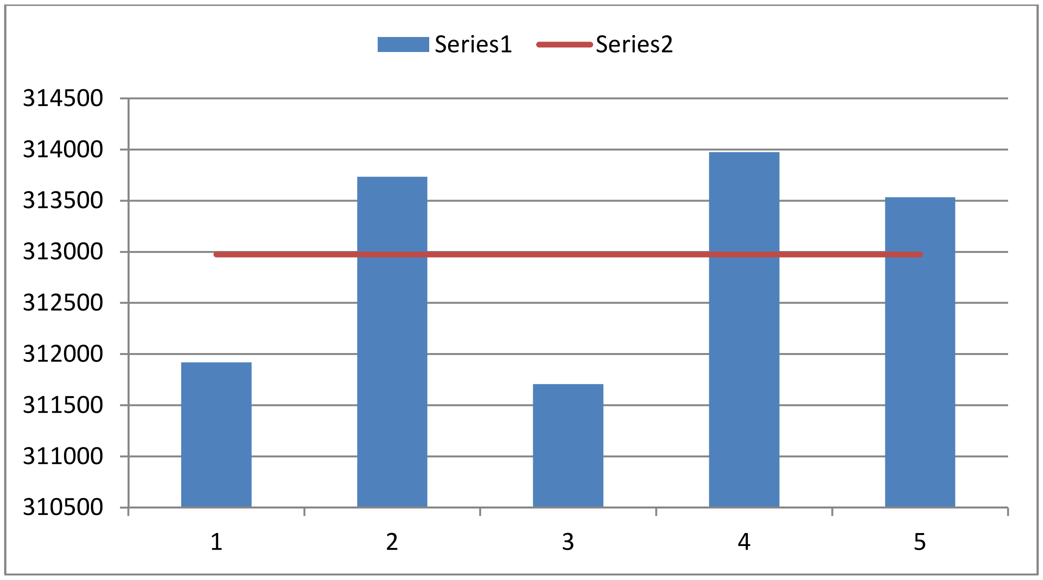
Max: 32436

Min: 29016

Mean: 31314.8

Figure 58: X10 C++-backend – sorting 10 million numbers

Figure 59 shows the runtime result of sorting 100 million numbers for 5 times. The program takes 313976 ms at the maximum and 311708 ms at the minimum. On average, it takes 312974.4 ms to finish sorting. The experiment is stopped after 5 times because each run takes more than 5 minutes. Even with 5 results, it is sufficient to say that X10 C++-backend is very slow. (For more details, see Appendix A.)



Max: 313976

Min: 311708

Mean: 312974.4

Figure 59: X10 C++-backend – sorting 100 million numbers

8.1.4. Quick sort summary

In Figure 60, the chart shows the averages of all runs from both languages and test cases. The first three columns are the average of sorting 1 million numbers where Java takes 173.6 milliseconds; X10 Java-backend takes 934.52 milliseconds; and X10 C++-backend takes 3220.92 milliseconds. The next three columns are averages sorting 10 million numbers, and the last three columns are averages sorting 100 million numbers.

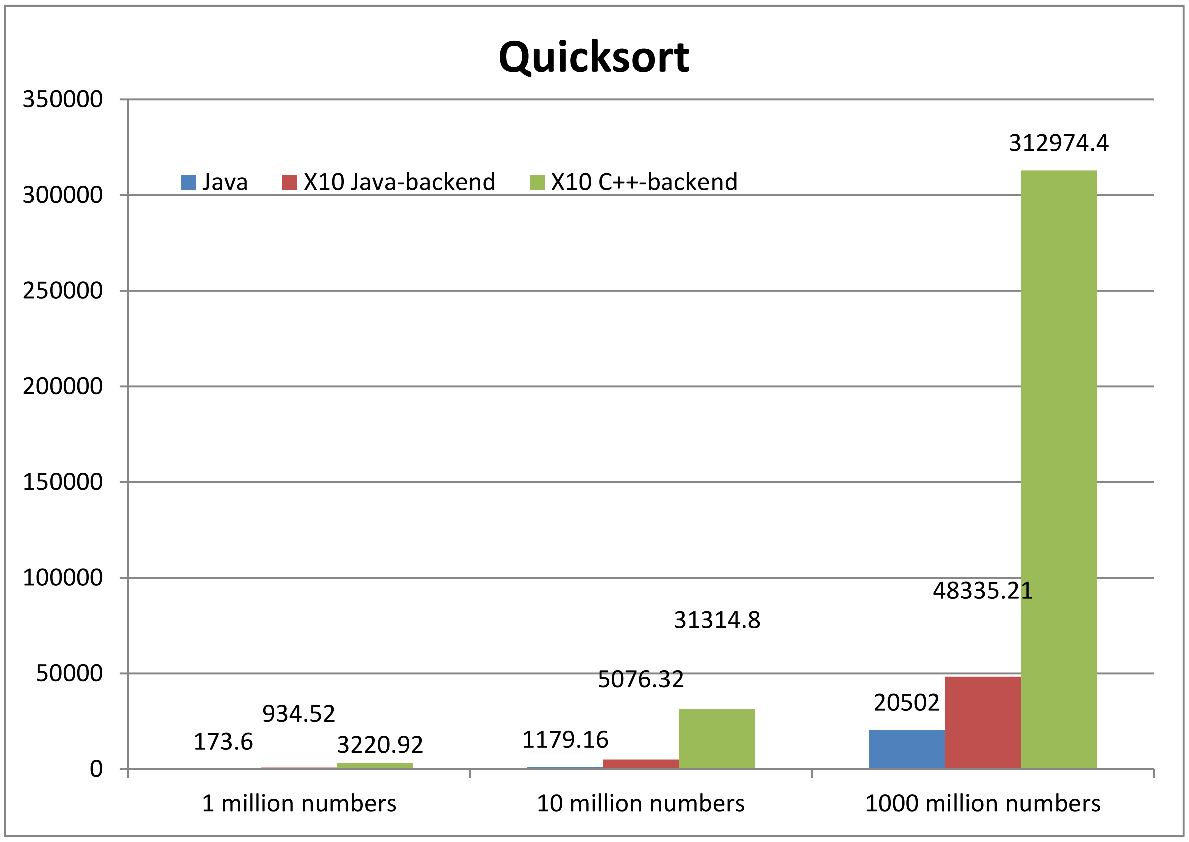


Figure 60: Quick sort comparison of all test cases of both languages and compilation modes

According to Figure 60, Java takes the shortest time for all cases to finish sorting. Meanwhile, X10 C++-backend takes the longest time for all test cases. On the other hand, Java-backend performs better when data size increases. For 1 million numbers, X10 Java-backend is more than 4X slower compared to ordinary Java, less than 4X for 10 million numbers, and less than 3X for 100 million numbers.

The CPU histogram in Figure 61 and Figure 62 shows that all four cores actively work proves that X10 utilizes all cores in both compilation modes. Looking at memory usage, X10 uses more memory than Java: 1.4GB for X10 Java-backend, 727.4 MB for X10 C++-backend, and 692.9MB for Java as Figure 61, Figure 62, and Figure 63. However, X10 performance is nothing close to the ordinary Java version.

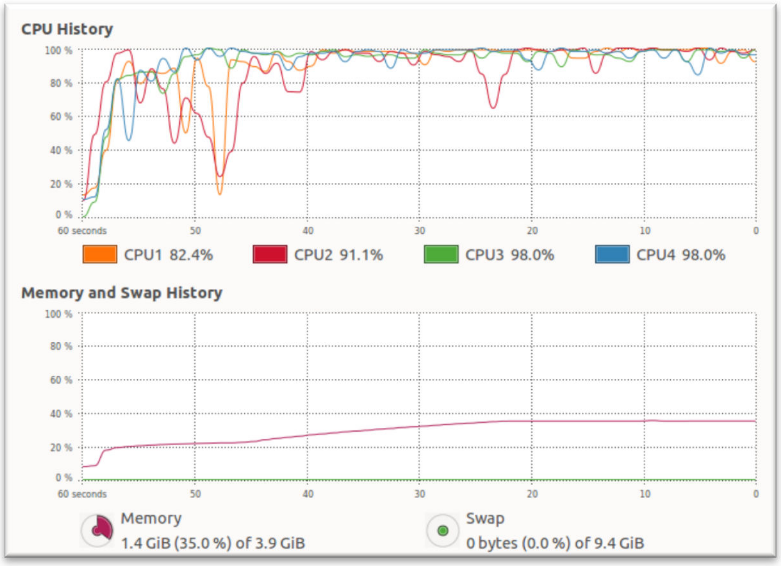


Figure 61: X10 Java-backend quick sort CPU usage

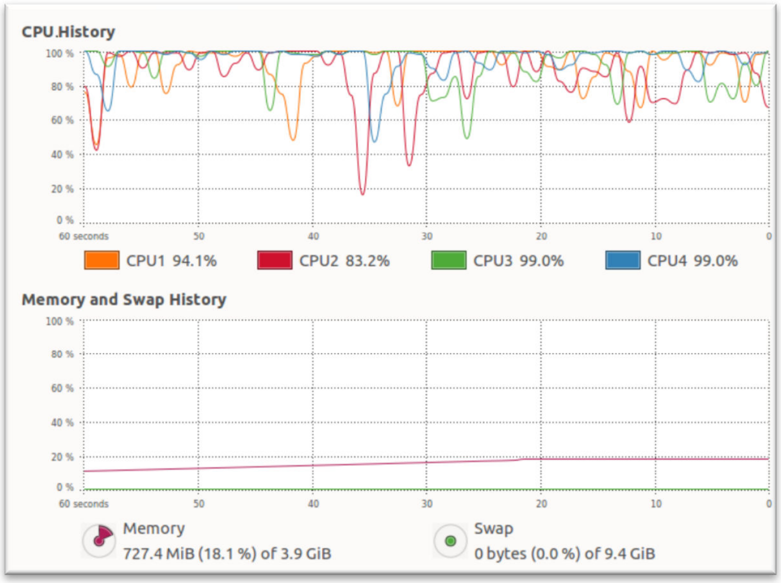


Figure 62: X10 C++-backend quicksort CPU usage

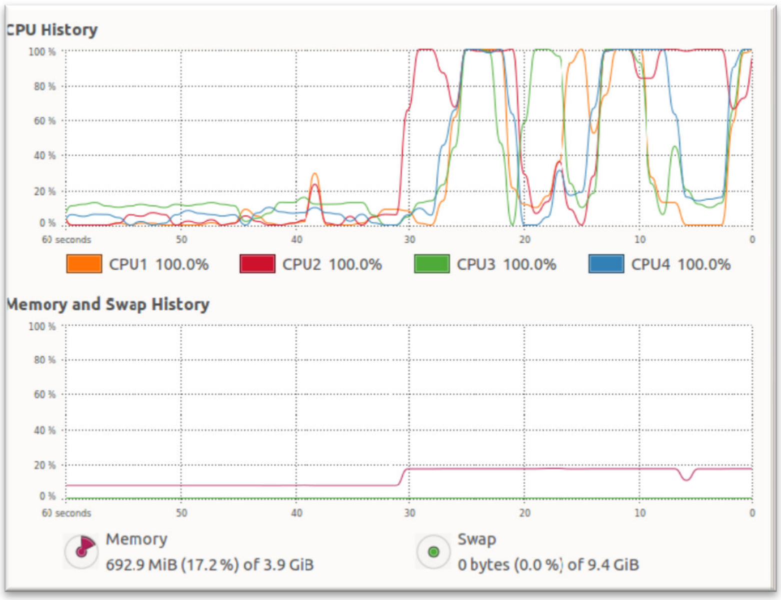


Figure 63: Java quick sort CPU usage

8.2. Matrix multiplication

To continue with benchmarking, this study also measures the performance of matrix multiplication using naive matrix multiplication algorithm. Naive matrix multiplication algorithm is the most basic way to multiply matrices. Elements in the resulting matrix C are the dot products of rows from matrix A and columns from matrix B, as in Figure 64. Unlike quick sort, which compares and then swaps elements, matrix multiplication uses simple mathematical operations.

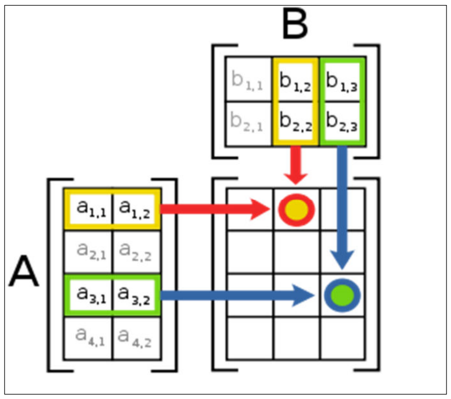


Figure 64: Matrix multiplication using naive algorithm [20]

In a multithreaded version, matrix A will be divided into 4 smaller matrices where each little matrix A_n will multiply with matrix B in parallel as Figure 65. For example, when matrix multiplying with 500x500 matrices, matrix A will be divided into 4 smaller matrices at a size of 124x500 where each piece multiplies with 500x500 matrix B. Each individual smaller matrix multiplication should not have any data dependency on other. Because they do not have any data dependency, dot product results can be put into only one result matrix without worrying about the other overwriting the same element index.

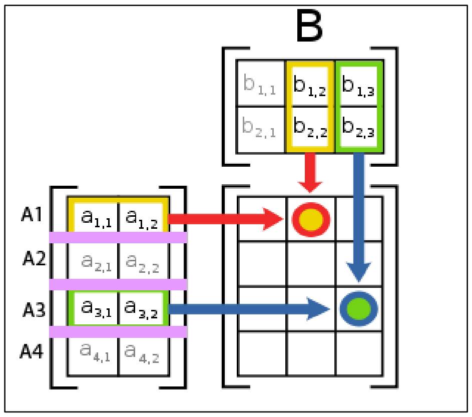
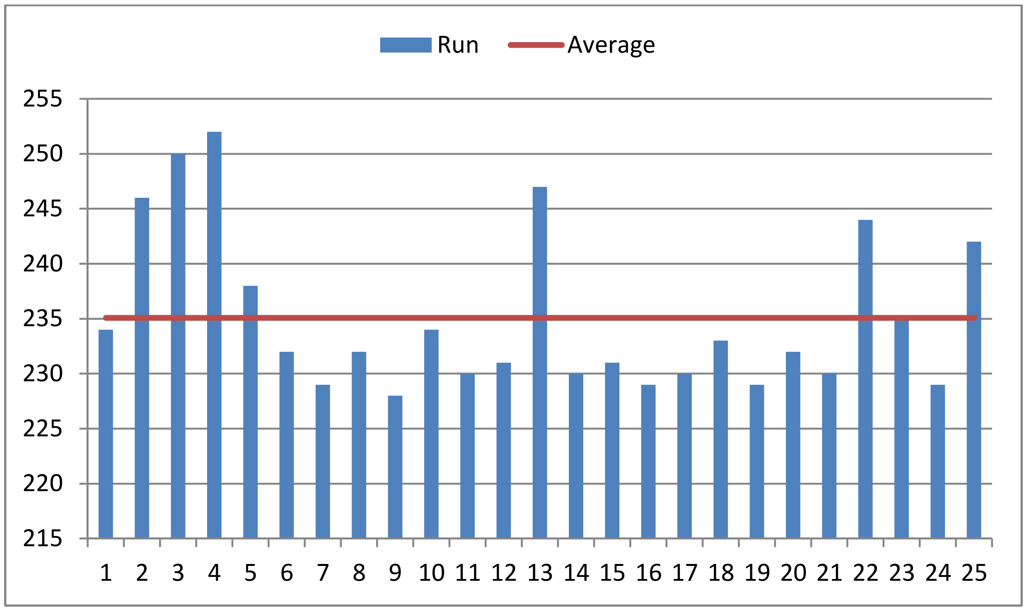


Figure 65: Matrix A is divided into 4 smaller matrices by purple lines; each sub matrix can do multiplication with matrix B in parallel

8.2.1. Java

The following figures illustrate the runtimes of matrix multiplication using Java implementation. Figure 66 shows the runtime of matrix size 500x500. The program at minimum takes 228 ms, and at maximum takes 252 ms to finish operation. On average, it takes 235.08 ms to finish. (For more details, see Appendix A.)



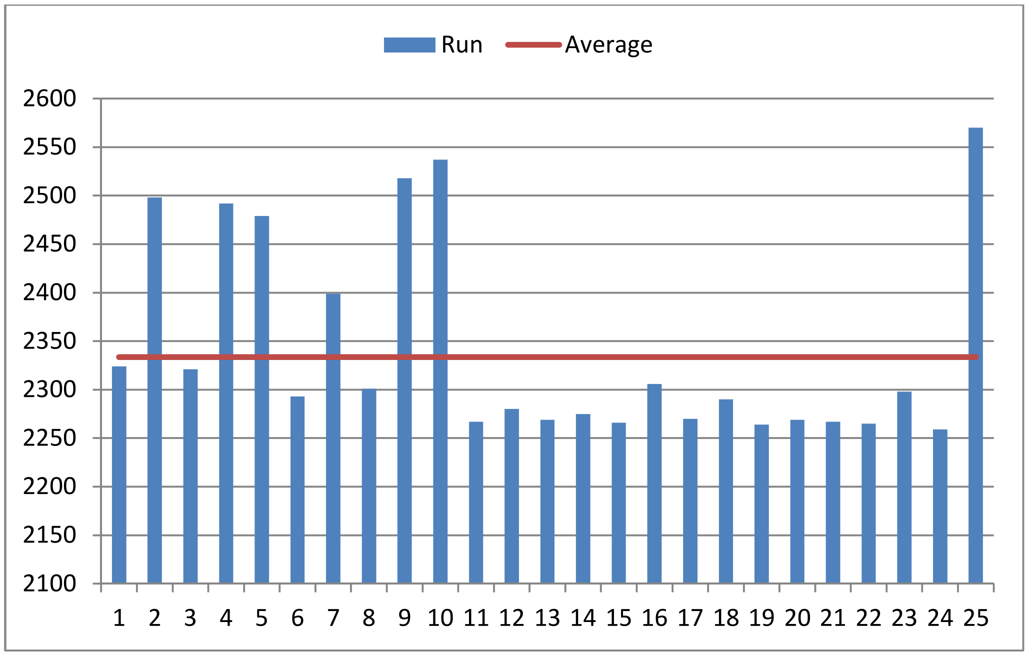
Min: 228

Max: 252

Mean: 235.08

Figure 66: Java matrix multiplication 500x500 matrices

Figure 67 shows the runtime of matrix size 1000x1000. The program at minimum takes 2259 ms, and at maximum takes 2537 ms to finish operation. On average, it takes 2333.63 ms to finish. (For more details, see Appendix A.)



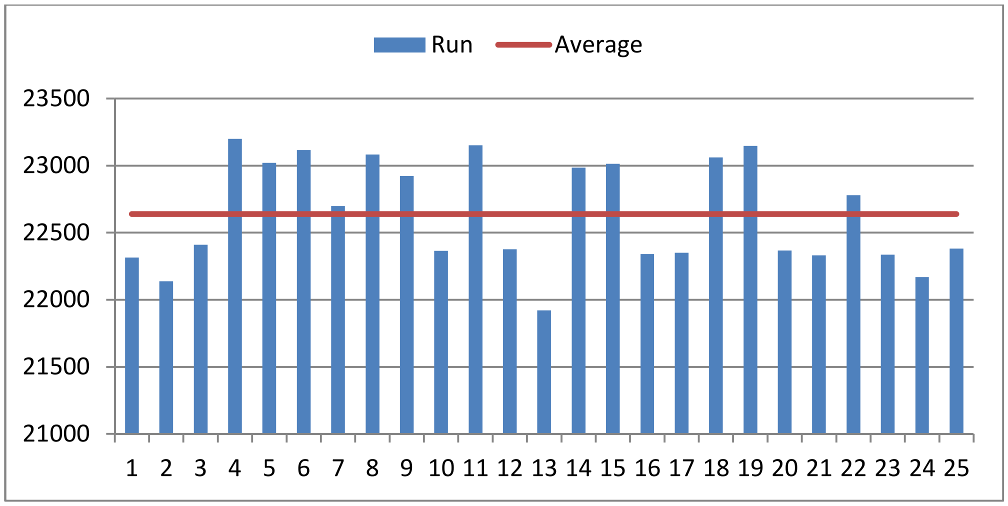
Min: 2259

Max: 2537

Mean: 2333.63

Figure 67: Java matrix multiplication 1000x1000 matrices

Figure 68 shows the runtime of matrix size 2000x2000. The program at minimum takes 21921 ms, and at maximum takes 23200 ms to finish operation. On average, it takes 22639.56 ms to finish. (For more details, see Appendix A.)



Min: 21921

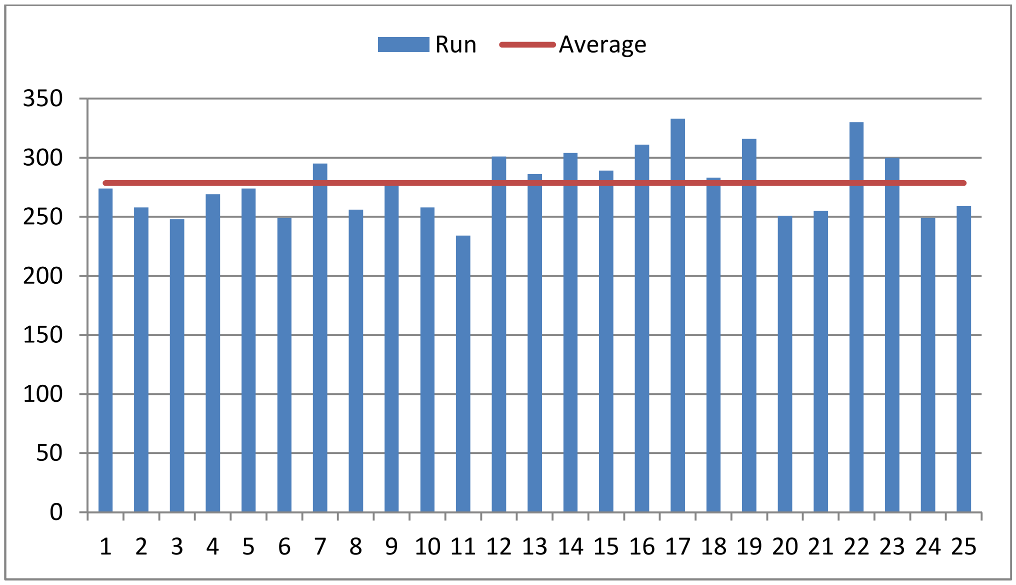
Max: 23200

Mean: 22639.56

Figure 68: Java matrix multiplication 2000x2000 matrices

8.2.2. X10 Java-backend

The following figures illustrate the runtimes of matrix multiplication using X10 implementation with Java-backend compilation. Figure 69 shows the runtime of matrix size 500x500. The program at minimum takes 228 ms, and at maximum takes 252 ms to finish operation. On average, it takes 235.08 ms to finish. (For more details, see Appendix A.)



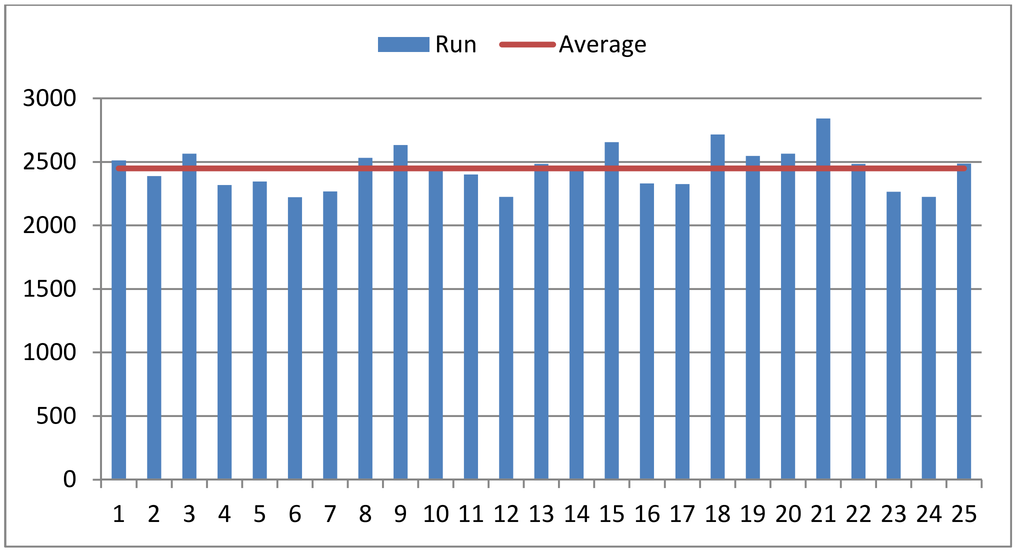
Min: 234

Max: 333

Mean: 278.44

Figure 69: X10 Java-backend matrix multiplicaiton 500x500

Figure 70 shows the runtime of matrix size 1000x1000. The program at minimum takes 2223 ms, and at maximum takes 2843 ms to finish operation. On average, it takes 2449.12 ms to finish. (For more details, see Appendix A.)



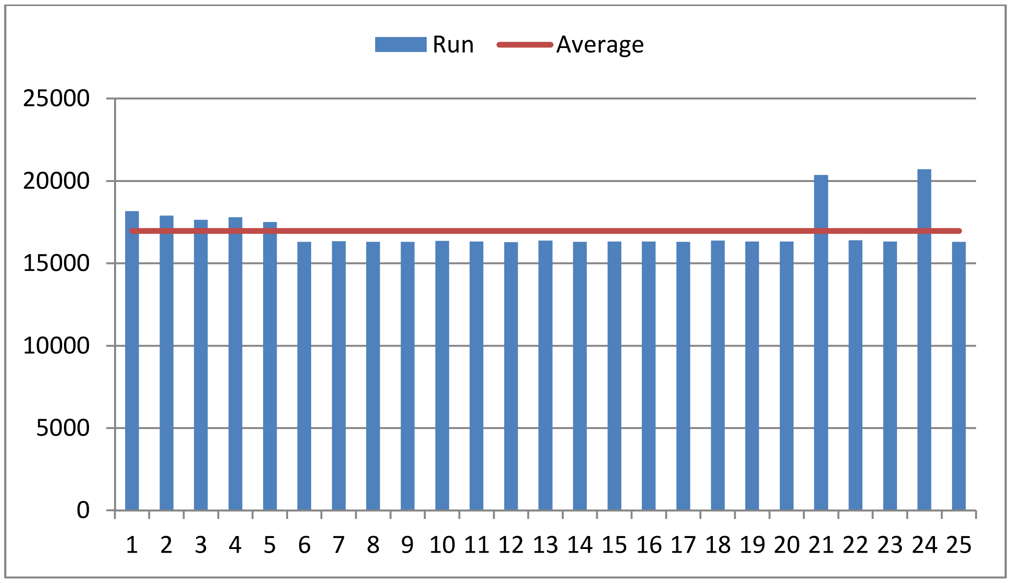
Min: 2223

Max: 2843

Mean: 2449.12

Figure 70: X10 Java-backend matrix multiplication 1000x1000 matrices

Figure 71 shows the runtime of matrix size 2000x2000. The program at minimum takes 16288 ms, and at maximum takes 20720 ms to finish. On average, it takes 16959.6 ms to finish. (For more details, see Appendix A.)



Min: 16288

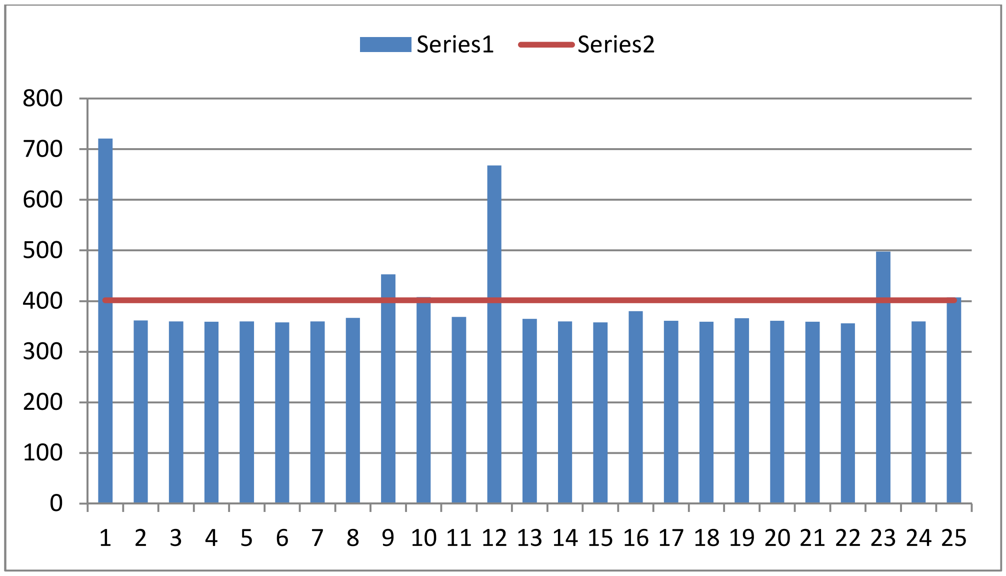
Max: 20720

Mean: 16959.6

Figure 71: X10 Java-backend matrix multiplication 2000x2000 matrices

8.2.3. X10 C++-backend

The following figures illustrate the runtimes of matrix multiplication using X10 implementation with C++-backend compilation. Figure 72 shows the runtime of matrix size 500x500. The program at minimum takes 356 ms, and at maximum takes 721 ms to finish operation. On average, it takes 401.4 ms to finish. (For more details, see Appendix A.)



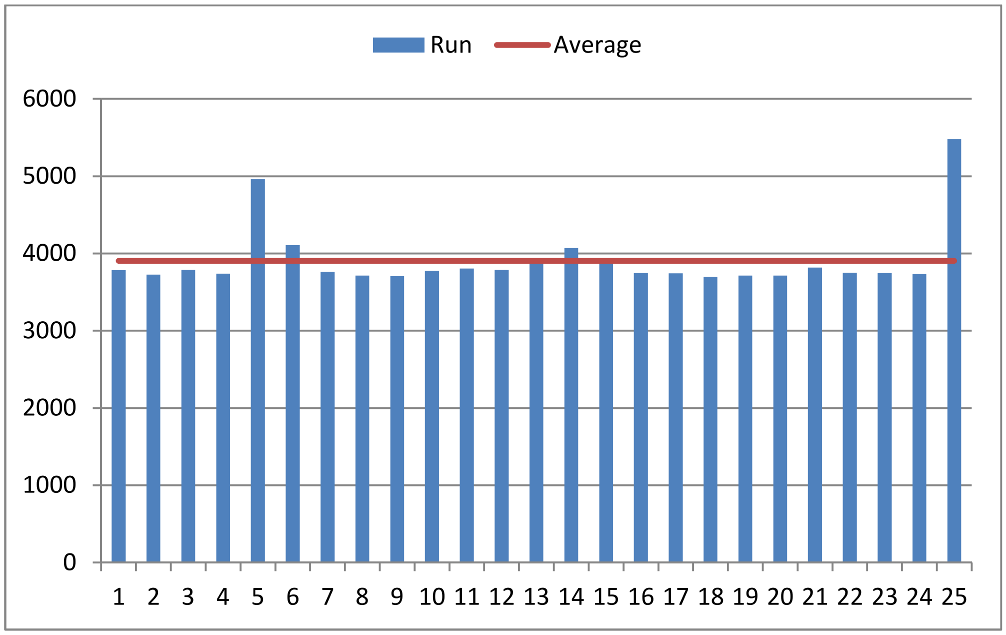
Min: 356

Max: 721

Mean: 401.4

Figure 72: X10 C++-backend matrix multiplication 500x500 matrices

Figure 73 shows the runtime of matrix size 1000x1000. The program at minimum takes 3696 ms, and at maximum takes 5479 ms to finish operation. On average, it takes 3906.12 ms to finish. (For more details, see Appendix A.)



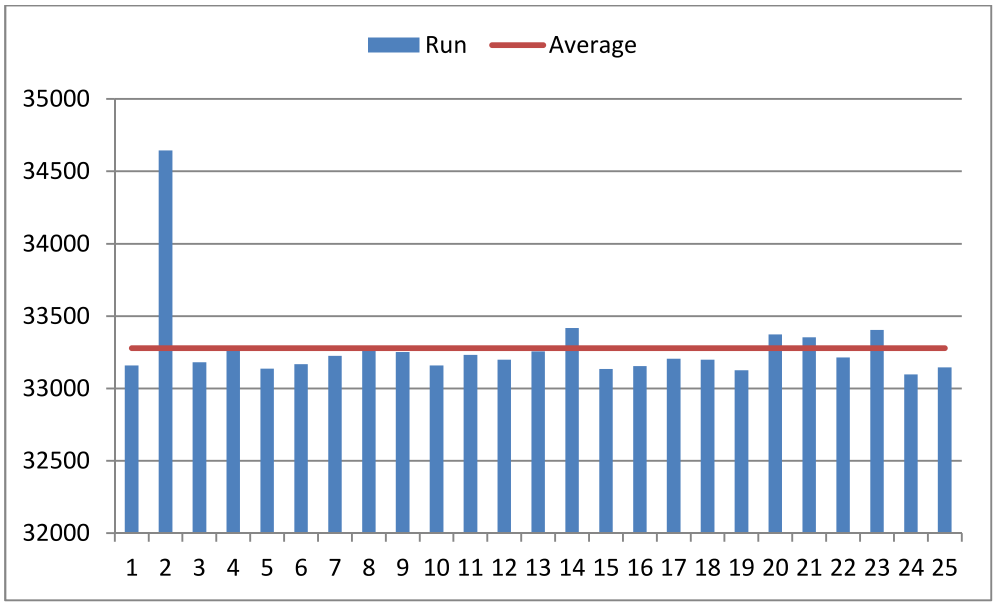
Min: 3696

Max: 5479

Mean: 3906.12

Figure 73: X10 C++-backend matrix multiplication 1000x1000 matrices

Figure 73 shows the runtime of matrix size 2000x2000. The program at minimum takes 33098 ms, and at maximum takes 34644 ms to finish operation. On average, it takes 33278.88 ms to finish. (For more details, see Appendix A.)



Min: 33098

Max: 34644

Mean: 33278.88

Figure 74: X10 C++-backend matrix multiplication 2000x2000 matrices

8.2.4. Matrix multiplication summary

In Figure 75, the first three columns show averages of multiplying 500x500 matrices; the second set of columns show averages for 1000x1000 matrices; and the last three show averages runtime for 2000x2000 matrices. The runtime performance is different than quick sort performance. X10 is not as much slower than Java. In contrast, X10 Java-backend performs even faster than ordinary Java implementation with large matrices. Looking at the matrix size 500x500 and 1000x1000, Java-backend catches up in terms of speed. For 2000x2000, Java-backend takes less time to finish. Even though, C++-backend is still slower than Java for all these test cases, C++-backend closes the performance gap. For large, less data-dependent problems, X10 tends to perform well. Unlike quick sort, compare-and-swap operations are slow.

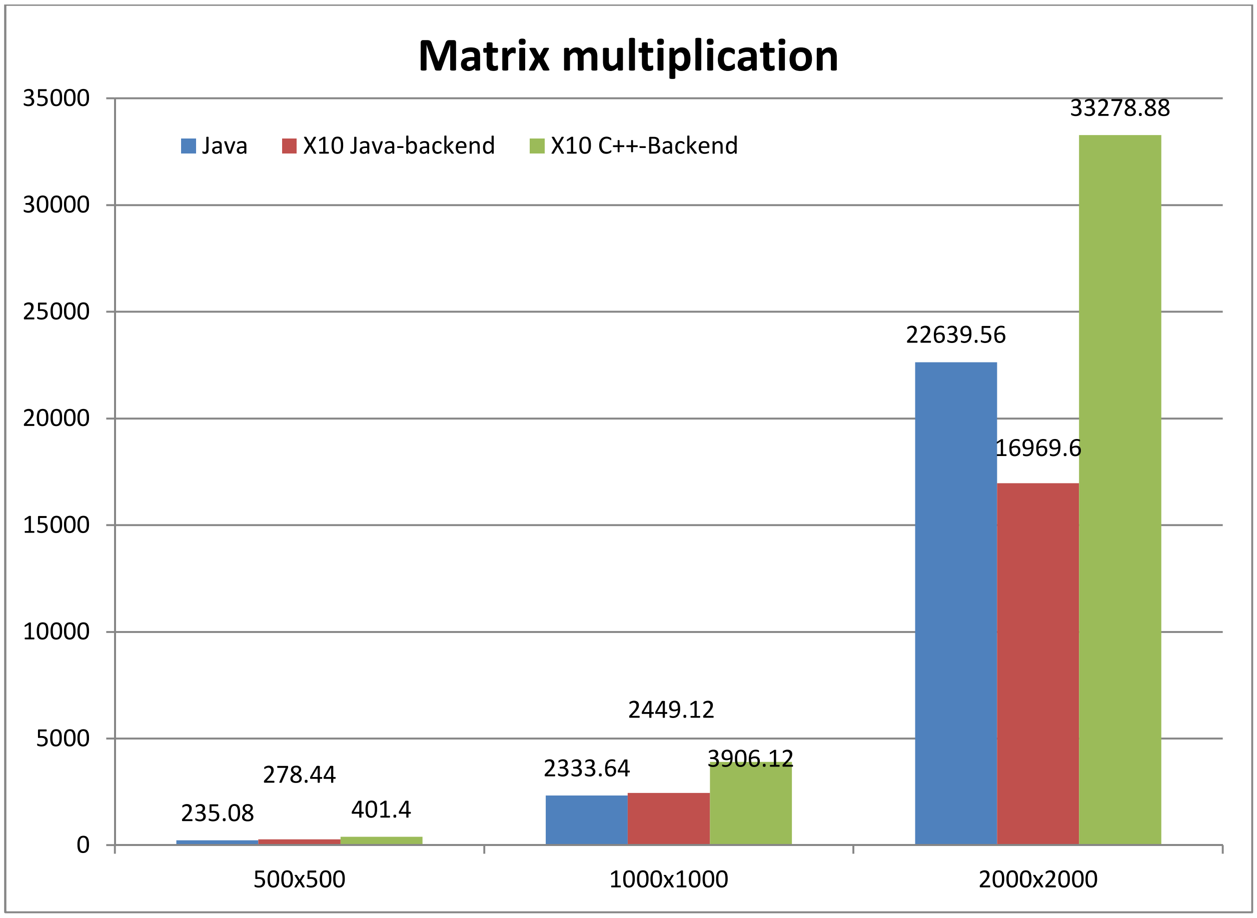


Figure 75: Matrix multiplication Java/X10

8.3. Advanced Encryption Standard (AES)

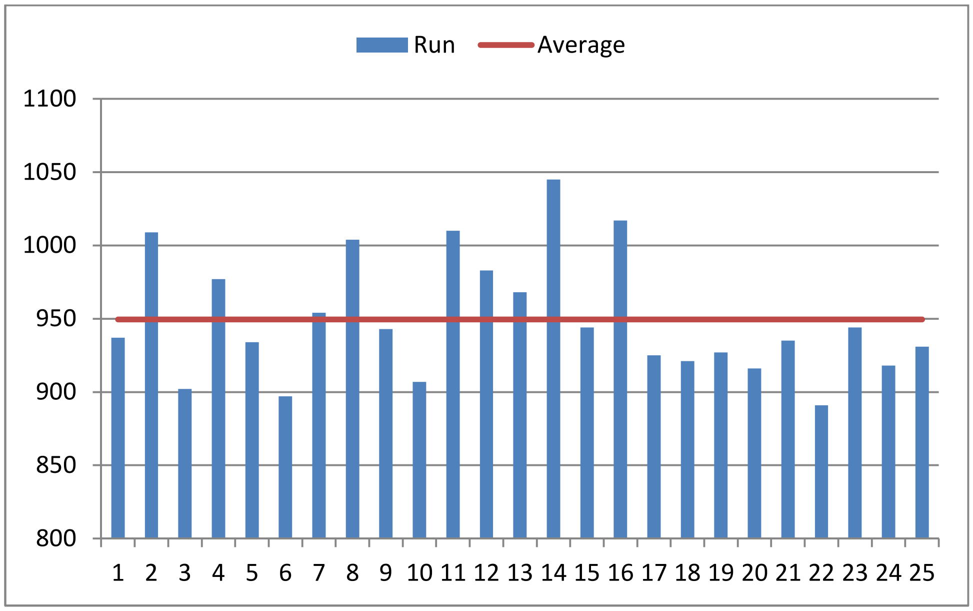
Unlike the previous two benchmarks, which are compare-then-swap operations for sorting and simple mathematics for matrix multiplication, AES encryption benchmark involves lower-level operations, such as switching, shifting bits and bytes. This experiment implements AES ECB mode using 256 bits key size. Because AES algorithm is widely known and there are many open-source implementations in various programming languages available online, the experiment uses a Java implementation by Popa Tiberiu [21]. The approach included any existing implementation in Java, porting it to X10 language, and then benchmarking the run time performance.

The parallel approach is to split the input file into 4 chunks as long as each chunk is the correct size. In ECB mode, current encryption block is not related to previous block or next block, which makes AES parallelization implementation easier. Using an object-oriented approach, an AES object initialization takes plain text chunk, key, and index offset from the original plain text as parameters. Once all AES objects are ready, it will start encrypting the plain text input. When all of them finish encrypting their assigned chunks, the main process takes output from all AES objects and assembles them back in to one. Times are captured when processes start and finish encryption operations. Any I/O operations, such as file assembly and file saving, will be ignored.

There will be 3 test cases where each will have different input file sizes: 2.1 Mbytes, 10.6 Mbytes, and 39.9 Mbytes. Similar to other algorithm test cases, each runs 25 times.

8.3.1. Java

The following figures illustrate the runtime results of AES encryptions using Java implementation. Figure 76 shows the runtimes of encrypting 2.1 Mbytes file. The Java program takes 891 ms at minimum and 1045 ms at maximum to finish encryption. On average, it takes 949.56 ms for encryption. (For more details, see Appendix A.)



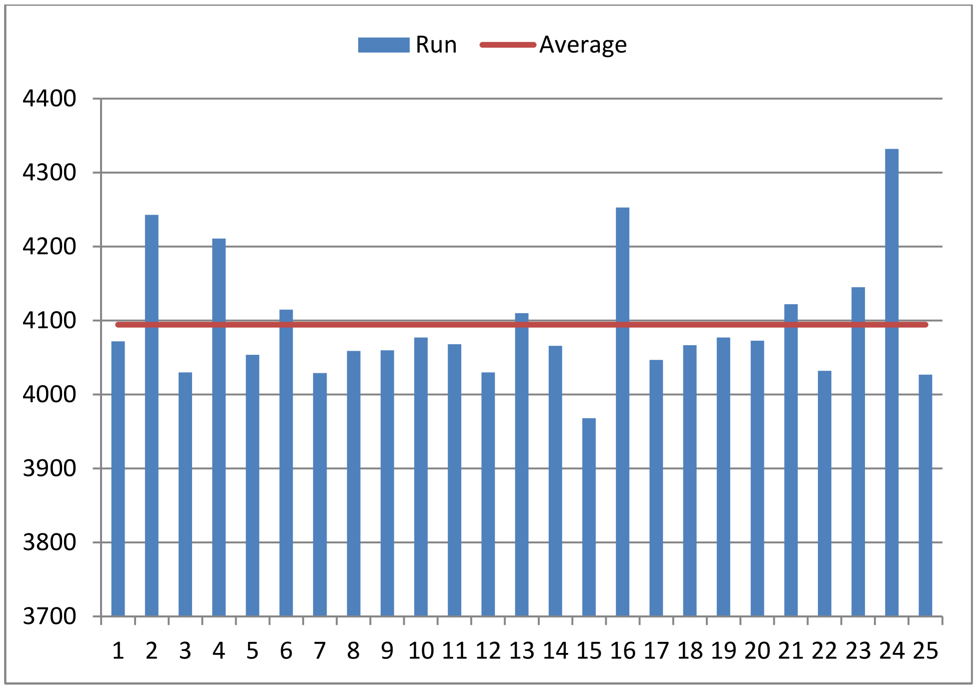
Min: 891

Max: 1045

Mean: 949.56

Figure 76: Java - encrypt 2.1 Mbytes file

Figure 77 shows the runtimes of encrypting a 10.6 Mbytes file. The Java program takes 3968 ms at minimum and 4332 ms at maximum to finish encryption. On average, it takes 4094.58 ms to encrypt. (For more details, see Appendix A.)



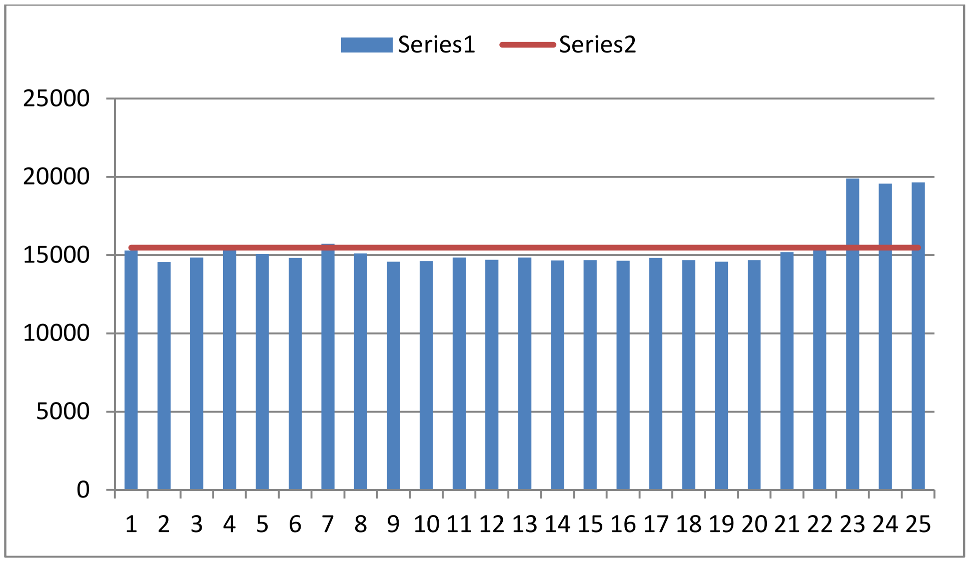
Min: 3968

Max: 4332

Mean: 4094.68

Figure 77: Java - encrypt 10.6 Mbytes file

Figure 78 shows the runtimes of encrypting a 39.9 Mbytes file. The program takes 14569 ms at minimum and 19885 ms at maximum to finish encryption. On average, it takes 15478.04 ms for all processes to finish. (For more details, see Appendix A.)



Min: 14569

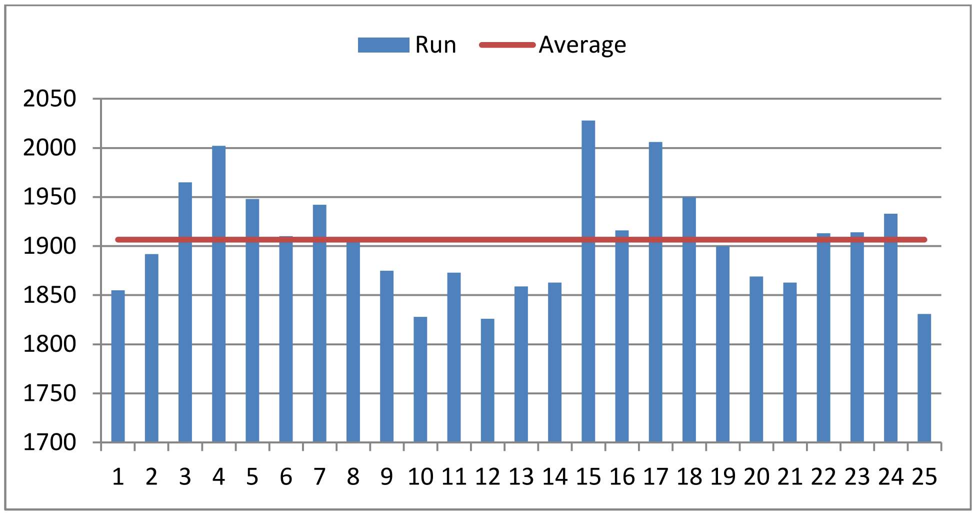
Max: 19885

Mean: 15478.04

Figure 78: Java - encrypt 39.9 Mbytes file

8.3.2. X10 Java-backend

The following figures illustrate the runtime results of AES encryption using X10 implementation with Java-backend compilation. Figure 79 shows the runtimes of encrypting a 2.1 Mbytes file 25 times. The program takes 1826 ms at minimum and 2028 ms at maximum. On average, it takes 1906.64 ms to encrypt. (For more details, see Appendix A.)



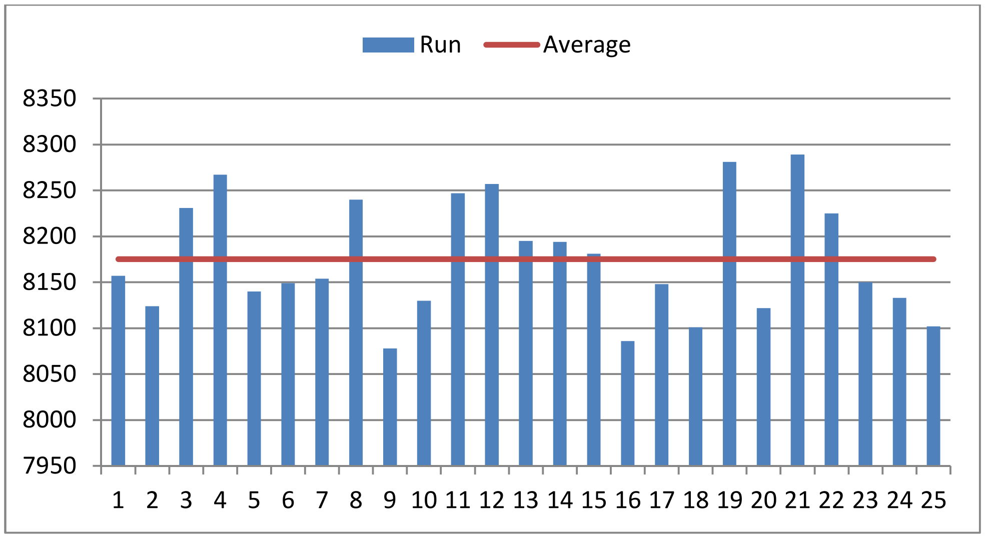
Min: 1826

Max: 2028

Mean: 1906.64

Figure 79: X10 Java-backend encrypts 2.1Mbytes file

Figure 80 shows the runtimes of encrypting a 10.6 Mbytes file 25 times. The program takes 8078 ms at minimum and 8289 ms at maximum. On average, it takes 8175.24 ms to encrypt. (For more details, see Appendix A.)



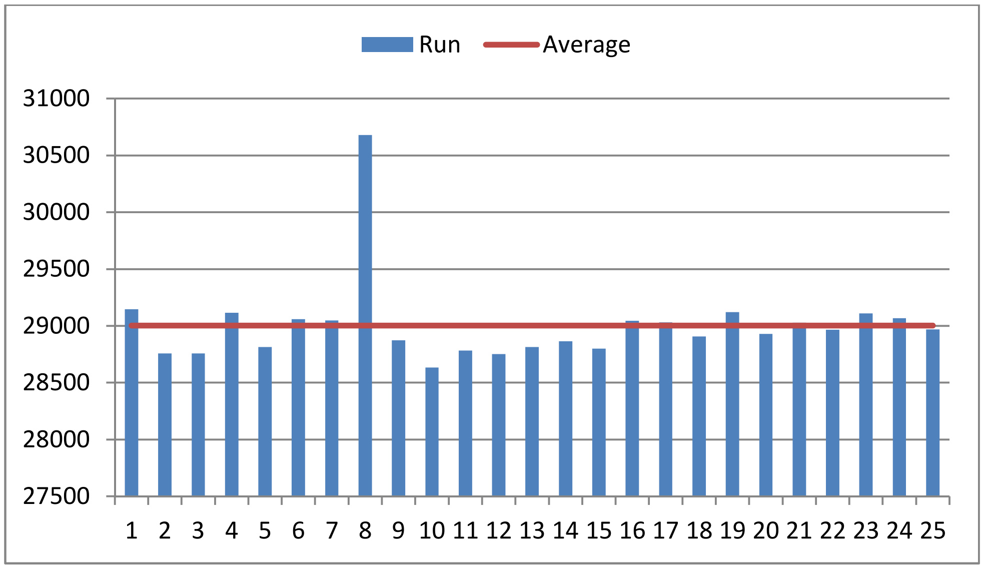
Min: 8078

Max: 8289

Mean: 8175.24

Figure 80: X10 Java-backend encrypts 10.6 Mbytes file

Figure 81 shows the runtimes of encrypting a 39.9 Mbytes file 25 times. The program takes 28635 ms at minimum and 30678 ms at maximum to finish encrypting. On average, it takes 29003.4 ms for all processes to finish. (For more details, see Appendix A.)



Min: 28635

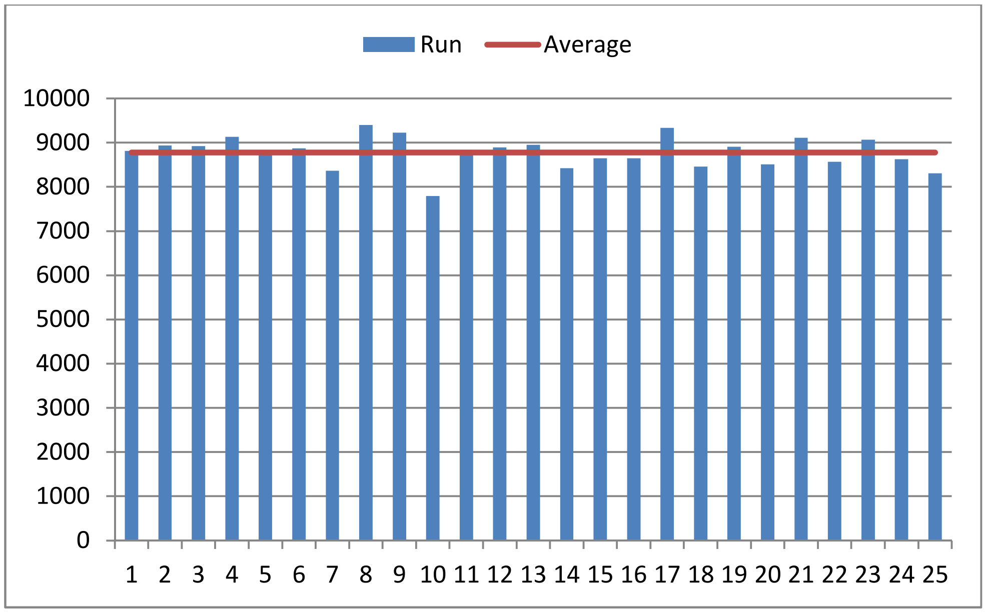
Max: 30678

Mean: 29003.4

Figure 81: X10 Java-backend encrypts 39.9 Mbytes file

8.3.3. X10 C++-backend

The following figures illustrate the runtime results of AES encryption using X10 implementation with C++-backend compilation. Figure 82 shows the runtimes of encrypting a 2.1 Mbytes file 25 times. The program takes 7795 ms at minimum and 9401 ms at maximum to finish encryption. On average, it takes 8780.96 ms for all processes to finish. (For more details, see Appendix A.)



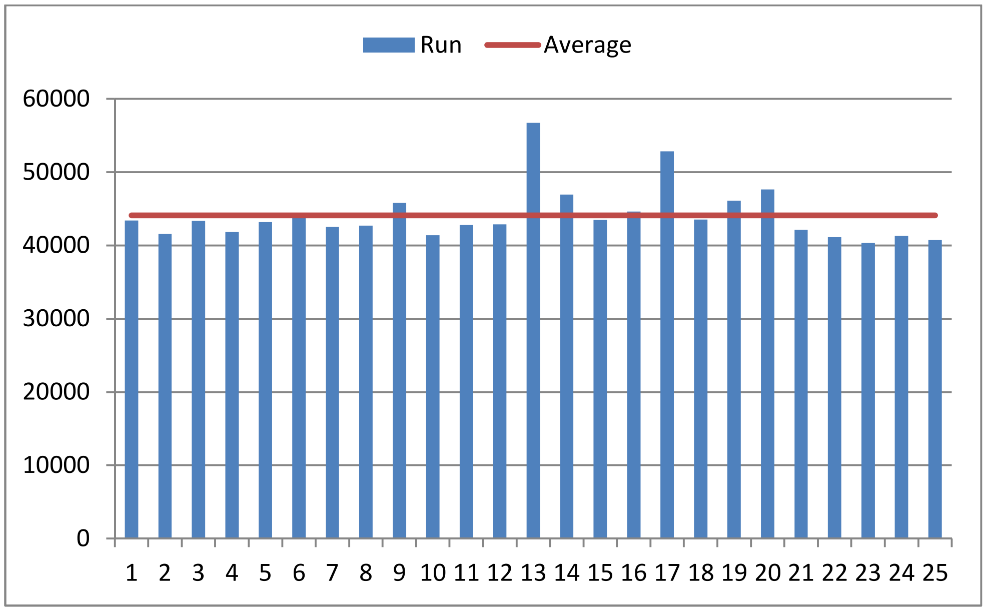
Min: 7795

Max: 9401

Mean: 8780.96

Figure 82: X10 C++-backend - encrypt 2.1Mbytes file

Figure 83 shows the runtimes of encrypting a 10.6 Mbytes file 25 times. The program takes 40355 ms at minimum and 56747 ms at maximum to finish. On average, it takes 44111.56 ms for all processes to finish. (For more details, see Appendix A.)



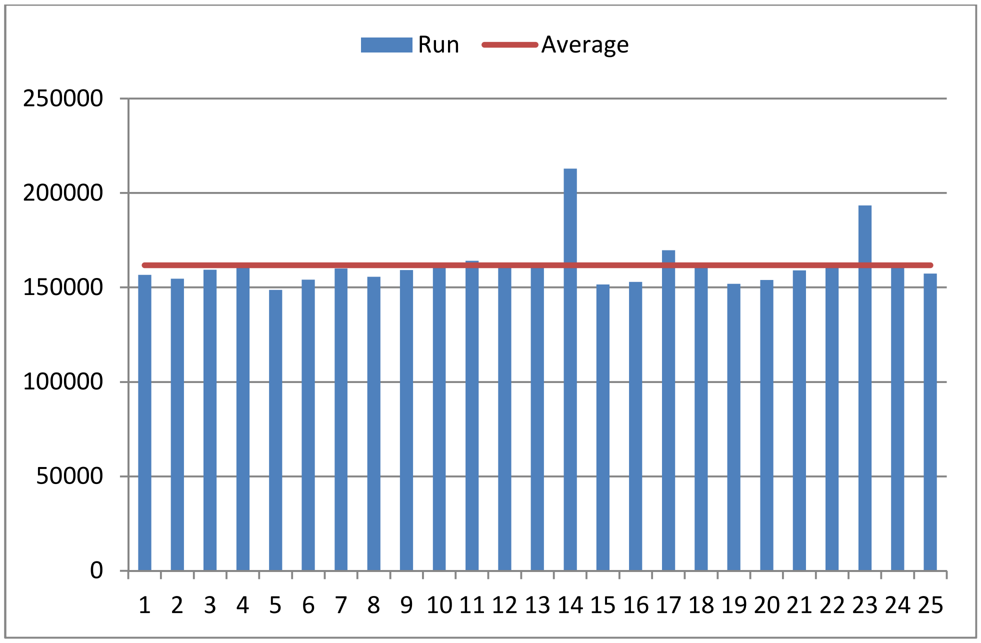
Min: 40355

Max: 56747

Mean: 44111.56

Figure 83: X10 C++-backend - encrypt 10.6 Mbytes file

Figure 83 shows the runtimes of encrypting a 39.9 Mbytes file 25 times. The program takes 148668 ms at minimum and 212953 ms at maximum to finish. On average, it takes 161767.5 ms for all processes to finish. (For more details, see Appendix A.)



Min: 148668

Max: 212953

Mean: 161767.5

Figure 84: X10 C++-backend - encrypt 39.9 Mbytes file

8.3.4. AES Summary

In this case, Java again outperforms X10 in either compilation method. The ordinary Java version runs twice as fast X10 Java-backend and nearly 10 times as fast for the C++-backend. For 2.1Mb file input, C++-backend runs less than 10 times slower than Java runtime, as file size increases the runtime ratio changes.

- 2.1 Mb: C++-backend/Java = 9.25
- 10.6 Mb: C++-backend/Java = 10.77
- 39.9 Mb: C++-backend/Java = 10.45

Looking at the last column in the graph, C++-backend takes more than 10 times longer compared to Java implementation. In this case, 10-times productivity does not mean 10-times faster.

X10 Java-backend seems to keep constants runtimes ratio or little better as file size increases comparing to regular Java program.

- 2.1 Mb: Java-backend/Java = 2.008
- 10.6 Mb: Java-backend/Java = 1.997
- 39.9 Mb: Java-backend/Java = 1.874

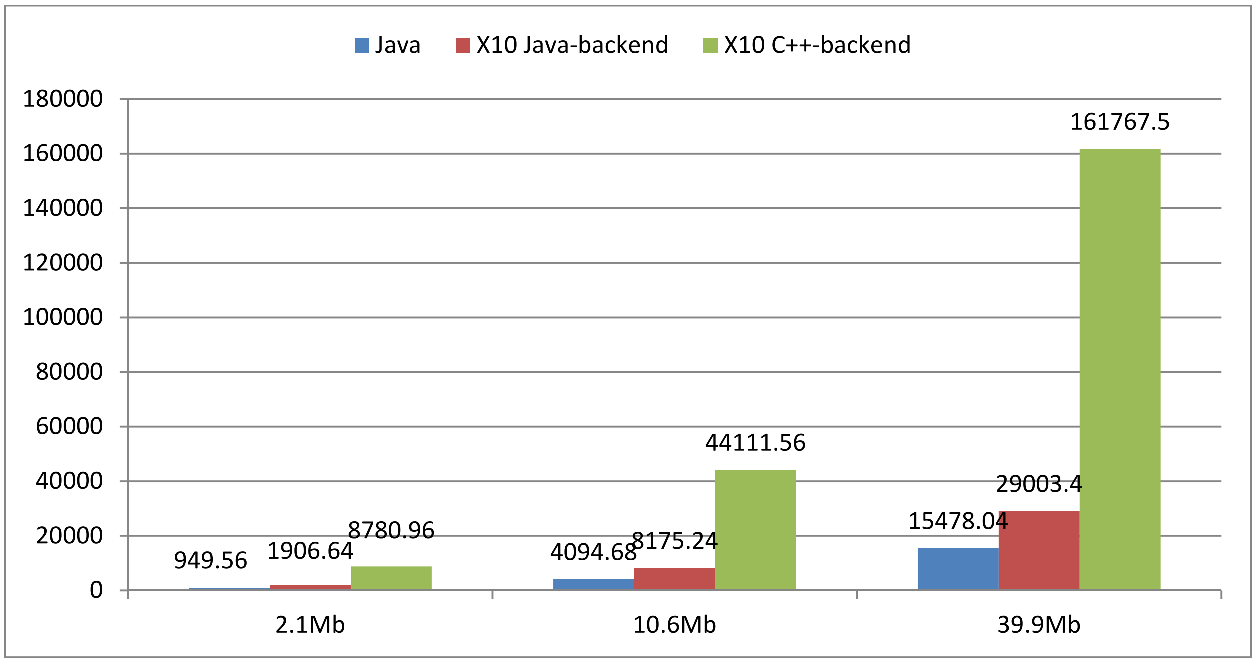


Figure 85: AES Java/X10

9. Conclusion and future work

X10 programming syntax is very similar to C++ and Java, proving that the creators of this language aimed to reduce the learning curve associated with learning a new programming language. Since X10 supports OOP, Java and C++ programmers can pick up the language easily within a short timeframe. Compared with OpenMP, I was having trouble inserting correct directives into the programs and understanding the OpenMP directives. By eliminating lock, X10 programmers no longer have an option to think about locking, which is a good move because it is one less thing to worry. Now, programmers can focus on how to solve parallelism with atomic blocks and other concurrency constructs. With X10's limited concurrency constructs, the X10 language is easier to develop multi-threaded programs than Java.

Even though X10 is more stable in later versions, it is not as mature as other languages. Looking at X10 libraries, they are not rich as Java in terms of documentation and available functions. Despite X10 being around for years, it is still considered a new programming language and was adopted less by the developers' community. Programming in X10 sometimes is frustrating because there are limited resources to look to for help. Unlike Java or C/C++, there are not many forums or blogs about any programming questions or issues, and only limited "X10 language" related resources will come up in a Google or Bing search. Most of the results are from the X10 language website. Most active resources are mailing lists. Questions related to X10 are usually answered within 24 hours by X10 developer team members, but users may not be able to find solutions to their questions. Besides mailing lists, programmers will rely heavily on X10 language specification and library API documentation, which is very similar to the Java API documentation format using Javadocs standard formats. For developing a multi-threaded programming, X10 provides simple API for manipulating thread concurrency.

X10 seems to be more stable in version 2.0 or later. Looking at some earlier versions of X10 research papers, X10 language syntax was very different; it was very much like Java syntax. Therefore, any old existing code would not be compiled in later versions. As X10 progressively improves and changes, there are risks of working codes from previous versions being unable to be compiled in the latest version. These backward compatibility issues are a normal risk when developing with experimental languages.

After going through these programming exercises, it is clear that X10 has its own advantages and disadvantages, including:

- Pros:
    - Handles type-safe asynchrony and atomic constructs to support multi-threaded environments.
    - Mirrors and supports popular object-oriented programming languages like Java and C++ language bindings for a faster learning curve and adoption.
    - Open-source supported community lead by IBM and major universities and research organization world-wide.
    - Simple, concrete concurrency constructs.
- Cons:
    - Still in early development and adoption phases and not mainstream yet.
    - API still premature and lacks some key I/O features, such as appending to a file.
    - Poor performance, as single local machine benchmarks show.

X10 performance is not as good as Java or C++ on multi-core local machine, in exchange X10 gives programmers easy ways to develop multi-threaded program. On a single machine, X10 may not have luck against Java, but X10 may have potential with cluster or multi-network nodes, because X10 provides `Place`, `DistArray` for multiple-host environments. One good study would compare performance between distributed computation in Java and X10.

In a previous AES benchmark, several tweaks were applied. The X10 language has real concept of multi-dimensional array unlike other languages treating multi-dimensional array as array of array. Therefore, the way to declare an array in X10 dictates how array access becomes optimized. By replacing `Rail[Byte]` with `Array[Byte]`, current AES performance boosts by 25%. However, 25% more is nowhere close to Java. It would be good to see how X10 generates Java/C++ code, and then analyze to see what can be optimized. Another advanced study would be applying artificial intelligence to an X10 compiler to scan program source code and decide which type of array access should be used.

10. References

1. http://en.wikipedia.org/wiki/Parallel_computing
2. http://www.intel.com/pressroom/kits/upcrc/parallelcomputing_backgrounder.pdf
3. Mattson, T., & Wrinn, M. (2008). Parallel programming: can we PLEASE get it right this time? In *Proceedings of the 45th annual Design Automation Conference* (DAC '08). New York, NY: ACM. doi: 10.1145/1391469.1391474 http://doi.acm.org/10.1145/1391469.1391474
4. http://x10-lang.org/
5. Cormen, T. (2001). *Introduction to Algorithms*. Cambridge, MA: MIT Press.
6. http://en.wikipedia.org/wiki/Computer_cluster
7. http://en.wikipedia.org/wiki/Grid_computing
8. Akhter, S. (2006). *Multi-core programming : increasing performance through software multi-threading*. Hillsboro, OR: Intel Press.
9. , Arvind; August, D., Pingali, K., Chiou, D., Sendag, R., & Yi, J.J. (2010). Programming multicores: Do applications programmers need to write explicitly parallel programs? *Micro, IEEE*, *30*(3), 19-33.
   doi: 10.1109/MM.2010.54
   URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5506935&isnumber=55069 26
10. Rau, B.R., & Fisher, J.A. (2003). Instruction-level parallelism. In A. Ralston, E.D. Reilly, & D. Hemmendinger (Eds.), *Encyclopedia of Computer Science* (4th ed.) (833-837). Chichester, UK: John Wiley and Sons Ltd.
11. Ramamoorthy, C.V. & Li, H.F. (1977). Pipeline Architecture. ACM Comput. Surv. 9, 1 (March 1977), 61-102. DOI=10.1145/356683.356687 http://doi.acm.org/10.1145/356683.356687
12. http://spectrum.ieee.org/semiconductors/processors/multicore-cpus-processor-proliferation
13. http://developers.sun.com/solaris/articles/raceconditions.html
14. Ross, P.E. (2008). Why CPU Frequency Stalled. *Spectrum, IEEE*, *45*(4), 72.
    doi: 10.1109/MSPEC.2008.4476447
    URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4476447&isnumber=44764 22
15. Chapman, M. (2005). *The benefits of dual-core processors in high-performance computing*. IBM Systems and Technology Group URL: http://www-07.ibm.com/servers/eserver/includes/content/opteron/pdf/XSW01277USEN.PDF
16. http://heprc.phys.uvic.ca/sites/heprc.phys.uvic.ca/
17. Oaks, S. (2004). *Java threads*. Beijing Farnham: O'Reilly.

18. Philippe, C., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., & Sarkar, V. (2005). X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not, 40*(10), 519-538. DOI=10.1145/1103845.1094852 http://doi.acm.org/10.1145/1103845.1094852

19. http://www.scl.ameslab.gov/Projects/parallel_computing/cluster_examples.html

20. http://en.wikipedia.org/wiki/Matrix_multiplication

21. http://n3vrax.wordpress.com/2011/08/14/aesrijndael-java-implementation/

22. http://en.wikipedia.org/wiki/Quicksort

23. Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., & Upton, M. (2002). Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, *6*(1), 1-12. Retrieved from http://www.mendeley.com/research/hyperthreading-technology-architecture-and-microarchitecture/

24. Kemal, E., Saraswat , V., & Sarkar, V. X10: An experimental language for high productivity programming of scalable systems (extended abstract). In Workshop on Productivity and Performance in High-End Computing (P-PHEC), February 200

25. http://www.sjsu.edu/people/robert.chun/courses/CS286Fall2011/s1/PP%20is%20a%20Hardware%20Solution.pdf

26. http://www.gotw.ca/publications/concurrency-ddj.htm

27. http://x10-lang.org/documentation/tutorials/sc2010-tutorial.html

Appendix A: Data

Run number	1 million numbers	10 million numbers	100 million numbers
1	183	844	10778
2	151	941	10867
3	224	1206	20502
4	167	1497	9282
5	197	1211	16578
6	140	1184	12250
7	148	833	12344
8	184	1058	18824
9	151	884	14198
10	166	1017	14993
11	189	1332	12379
12	180	1350	10490
13	193	1050	11139
14	184	1023	10585
15	189	1567	12701
16	191	1403	11883
17	139	1383	11347
18	170	1846	15795
19	180	990	10129
20	156	934	19307
21	151	1412	9898
22	191	1084	16473
23	187	1103	14821
24	144	1245	11596
25	185	1082	10232

Table 1: Java quicksort runtime

Run number	1 million numbers	10 million numbers	100 million numbers
1	906	5096	47621
2	885	4986	49096
3	964	5036	48008
4	1022	5053	47213
5	888	4979	49822
6	952	5074	48572
7	877	5132	49301
8	1053	5144	47986
9	874	4982	48722
10	846	5023	48751
11	1182	4974	48347
12	884	4936	47958
13	1002	5459	48189
14	813	5091	47665
15	888	5141	47486
16	897	5046	48765
17	972	4914	48892
18	940	5105	47723
19	923	5024	47562
20	878	5195	47843
21	829	5346	47120
22	964	5043	48748
23	1109	4967	49415
24	971	5130	49240
25	844	5032	49540

Table 2: X10 Java-backend - quicksort runtime

Run number	1 million numbers	10 million numbers	100 million numbers
1	3136	31250	311919
2	3176	31624	313735
3	3173	31792	311708
4	3180	31753	313976
5	3053	31539	313534
6	3377	31875	
7	3249	31741	
8	3067	31689	
9	3291	31956	
10	3162	32153	
11	3287	32436	
12	3432	31974	
13	3205	31751	
14	3388	31848	
15	3333	32429	
16	3456	31204	
17	3361	31555	
18	3518	31580	
19	3291	31838	
20	3422	31476	
21	3068	29450	
22	2964	29361	
23	2912	29016	
24	3037	29783	
25	2985	29797	

Table 3: X10 C++-backend quick sort runtime

Run number	500x500	1000x1000	2000x2000
1	234	2324	22316
2	246	2498	22139
3	250	2321	22411
4	252	2492	23200
5	238	2479	23021
6	232	2293	23116
7	229	2399	22698
8	232	2301	23084
9	228	2518	22924
10	234	2537	22365
11	230	2267	23152
12	231	2280	22377
13	247	2269	21921
14	230	2275	22986
15	231	2266	23014
16	229	2306	22341
17	230	2270	22351
18	233	2290	23061
19	229	2264	23148
20	232	2269	22367
21	230	2267	22331
22	244	2265	22779
23	235	2298	22337
24	229	2259	22169
25	242	2570	22381

Table 4: Java matrix multiplication runtime

Run number	500x500	1000x1000	2000x2000
1	274	2513	18176
2	258	2388	17888
3	248	2566	17637
4	269	2318	17805
5	274	2345	17504
6	249	2223	16306
7	295	2267	16340
8	256	2533	16309
9	279	2632	16296
10	258	2459	16364
11	234	2400	16325
12	301	2225	16288
13	286	2484	16375
14	304	2429	16307
15	289	2656	16324
16	311	2331	16315
17	333	2326	16296
18	283	2716	16378
19	316	2548	16327
20	251	2564	16331
21	255	2843	20356
22	330	2484	16397
23	300	2266	16325
24	249	2226	20720
25	259	2486	16301

Table 5: X10 Java-backend matrix multiplication runtime

Run number	500x500	1000x1000	2000x2000
1	721	3783	33160
2	362	3726	34644
3	360	3789	33182
4	359	3740	33263
5	360	4960	33137
6	358	4106	33167
7	360	3765	33225
8	367	3714	33267
9	453	3707	33253
10	408	3776	33159
11	369	3806	33232
12	668	3789	33198
13	365	3892	33257
14	360	4072	33418
15	358	3884	33134
16	380	3747	33154
17	361	3744	33206
18	359	3696	33200
19	366	3714	33125
20	361	3713	33373
21	359	3818	33353
22	356	3751	33215
23	498	3748	33405
24	360	3734	33098
25	407	5479	33147

Table 6: X10 C++-backend matrix multiplication runtime

Run number	2.1Mb	10.6Mb	39.9Mb
1	937	4072	15287
2	1009	4243	14569
3	902	4030	14840
4	977	4211	15383
5	934	4054	15064
6	897	4115	14818
7	954	4029	15717
8	1004	4059	15114
9	943	4060	14578
10	907	4077	14622
11	1010	4068	14842
12	983	4030	14695
13	968	4110	14848
14	1045	4066	14654
15	944	3968	14676
16	1017	4253	14633
17	925	4047	14823
18	921	4067	14691
19	927	4077	14572
20	916	4073	14688
21	935	4122	15202
22	891	4032	15521
23	944	4145	19885
24	918	4332	19573
25	931	4027	19656

Table 7: Java AES encryption runtime

Run number	2.1Mb	10.6Mb	39.9Mb
1	1855	8157	29148
2	1892	8124	28758
3	1965	8231	28759
4	2002	8267	29117
5	1948	8140	28814
6	1910	8149	29060
7	1942	8154	29049
8	1905	8240	30678
9	1875	8078	28874
10	1828	8130	28635
11	1873	8247	28783
12	1826	8257	28753
13	1859	8195	28813
14	1863	8194	28866
15	2028	8181	28800
16	1916	8086	29044
17	2006	8148	29032
18	1950	8101	28907
19	1900	8281	29122
20	1869	8122	28930
21	1863	8289	29029
22	1913	8225	28966
23	1914	8150	29109
24	1933	8133	29069
25	1831	8102	28970

Table 8: X10 Java-backend AES encryption runtime

Run number	Run number	2.1Mb	10.6Mb
1	8813	43412	156748
2	8934	41554	154687
3	8922	43361	159324
4	9134	41828	161178
5	8759	43179	148668
6	8869	43847	154097
7	8369	42510	160107
8	9401	42687	155669
9	9228	45812	159137
10	7795	41385	160576
11	8835	42774	164165
12	8892	42880	160338
13	8955	56747	163077
14	8421	46936	212953
15	8648	43483	151674
16	8651	44635	152937
17	9335	52848	169783
18	8458	43522	161624
19	8912	46103	151929
20	8507	47644	153930
21	9112	42135	159085
22	8569	41115	161266
23	9068	40355	193453
24	8627	41297	160426
25	8310	40740	157356

Table 9: X10 C++-backend AES encryption runtime