Fall 2012

An Evaluation of the X10 Programming Language

Xiu Guo
San Jose State University

An Evaluation of the X10 Programming Language

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Masters of Science

By

Xiu Guo

Fall 2012

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

_____	_____
Dr. Robert Chun	Date
_____	_____
Dr. Sami Khuri	Date
_____	_____
Dr. Mark Stamp	Date

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank my advisor, Dr. Robert Chun, for providing his constant guidance and support throughout this project. I appreciate my committee members Dr. Sami Khuri and Dr. Mark Stamp for their time and suggestions.

Also, I would like to thank and offer my regards to all of those who supported me in any way for completing my Master's project in Computer Science at San Jose State University.

Abstract

As predicted by Moore's law, the number of transistors on a chip has been doubled approximately every two years. As miraculous as it sounds, for many years, the extra transistors have massively benefited the whole computer industry, by using the extra transistors to increase CPU clock speed, thus boosting performance.

However, due to heat wall and power constraints, the clock speed cannot be increased limitlessly. Hardware vendors now have to take another path other than increasing clock speed, which is to utilize the transistors to increase the number of processor cores on each chip.

This hardware structural change presents inevitable challenges to software structure, where single thread targeted software will not benefit from newer chips or may even suffer from lower clock speed.

The two fundamental challenges are:

1. How to deal with the stagnation of single core clock speed and cache memory.
2. How to utilize the additional power generated from more cores on a chip.

Most software programming languages nowadays have distributed computing support, such as C and Java [1]. Meanwhile, some new programming languages were invented from scratch just to take advantage of the more distributed hardware structures. The X10 Programming Language is one of them.

The goal of this project is to evaluate X10 in terms of performance, programmability and tool support.

Table of Contents

Index of Figures, Charts and Tables

An Evaluation of the X10 Programming Language

By Xiu Guo

1. Introduction

1.1 What is X10?

X10 is an open-source programming language developed to address the architectural challenge of multiple cores, hardware accelerators, clusters, and supercomputers by providing scalable performance in a productive manner. It is being developed by IBM Research, which roots X10 on a type-safe, class-based, object-oriented foundation.

The philosophy behind the new programming language, X10, is to make parallel programming easier to code and less-prone to deadlock and race condition. To achieve that, X10 embraces three principles: asynchrony, locality and atomicity [2]. Most of the other major programming languages now achieve their parallelism goal by adding additional libraries and APIs. However, because X10 was built from ground up with parallelism in mind, its native mode is to support asynchrony, locality and atomicity. Therefore, X10 might become a more effective alternative on distributed-computing due to its clearer goal. Its legitimacy as a useful alternative needs extensive experiments and evaluations.

1.2 Project Goal

The project will focus on whether X10 can utilize the divide-and-conquer concept well enough to serve as a pragmatic solution to parallelize certain algorithms and determine whether X10 is programmer-friendly enough to ease the difficulty when developing distributed software.

The performance evaluation will be based on a three-step process:

1. The collection of different divide-and-conquer problems.
2. Their implementation using Java threads and X10 (Java back-end) separately.
3. A comparison of the performance of above approaches.

From the results above, one can draw a conclusion on whether X10 can offer a speedup in divide-and-conquer problems over Java or at least ease the difficulty of programming parallel programs with little or no performance lost.

With quantitative evaluation results, this project can serve as a guide for developers on finding an optimal approach in terms of a balance of performance and ease of programmability.

2. Algorithms and Platforms

2.1 Divide-and-conquer algorithms

Divide-and-conquer algorithms are widely known and used to speed up computation by reducing a complex problem into a collection of smaller sub-problems in a recursive fashion. By combining results from sub-problems, the outcome of the original master problem can be produced. [3]

2.2 Why divide-and-conquer algorithms

Because of the nature of divide-and-conquer algorithms, they are well suited for running in parallel. When a problem is divided into smaller sub-problems, each sub-problem can be assigned to a single thread which will send the result back to the main thread upon completion.

Evaluating the use of divide-and-conquer algorithms can lead to a better understanding of the efficiency to spawn threads, the ability to prevent race condition and deadlock, and the cost involved due to overhead.

2.3 Parallel Processing Platforms

In reality, the hardware available and the algorithm used are usually restricted by financial and research resources. A better execution language platform sometimes will noticeably boost productivity even when the above resources stay the same. It is the exact goal of this project to find out whether X10 is a better execution platform. However, first, the comparison language platform needs to be described.

2.3.1 Java Virtual Machine Threading Models

Java is known as platform-independent; however, it is somehow platform-dependent if the Virtual Machine performance is taken into consideration. Although Java's operations are consistent to users among different platforms, the underlying platforms usually handle

the implementation differently, especially for features like concurrency and multi-threading. For scheduling multiple threads, Java utilizes two models: cooperative and preemptive threading.

For the cooperative threading model, threads can decide whether to give up their processor resources to other queued threads. This model is safe and easy to use for programmers. A programmer can access variables without having to worry about them changing between lines of code. However, performance of this model relies highly on how well written the code is. Poorly written code may have certain threads occupy some processor resources all the time which starve other threads.

For the preemptive threading model, threads run independently from each other. Only one thread at a time has focus, but focus can change from one line of code to the next. The switching between threads is under the control of the operating system [4]. This model is considered a better approach because no threads can monopolize the CPU resources. On the other hand, this model sometimes introduces unnecessary switches by the operating system, which is out of the programmer's control. That creates a nondeterministic environment where the programmer has no knowledge beforehand of which scheduling the threads will follow.

While both models have their pros and cons, it is generally a good idea to adapt to both models when designing a Java multi-thread program.

2.3.2 X10 Performance Model

X10 is an object-oriented language designed specifically to enable the productive programming of multi-core and multi-node computers. In addition to the expected core language features of any modern object-oriented language, it contains additional constructs of expressing fine grained concurrency and distributed computation. [5]

The design of X10's syntax has significant overlap with Java's type system, though differences do exist. For example, instead of using “int i” for declaring an integer in Java, X10 uses “i:Int”. In X10, variables are declared using either the keyword “val” or “var”. “val” is similar to a constant in Java, but it does not need to have a value when declared. It only has to be assigned some value at some point. Once it has a value, the value cannot be changed. “var” is like a normal variable where the values it holds are updated as needed.

These are examples that show the differences but do not indicate any performance consideration in the design of the language. Researchers of X10 did put significant thought into the design of the language so that it would be suitable for distributed computing.

2.3.3 X10 Type System

The type systems between Java and X10 differ in three ways, which are well intended to achieve X10's parallelism goal.

1. In addition to classes, X10 adds two additional kinds of fundamental structure: *function*s and *struct*s.
2. X10's generic type systems do not have the same erasure semantics as Java's generic types do. [5]
3. X10 includes constrained types, which enhances the ability to more precisely specify the acceptable values of a type by boolean expression.

2.3.4 *Struct* in X10

In X10, *struct* is a different concept from that in C, though they share the same name. *Struct* is mainly designed to improve run time performance of X10 programs. First a simple example will demonstrate the necessity of *struct*.

Assuming class Point2D is a subclass of class Point, a method call is defined as:

```
public def doSomething(p:Point){...}
```

which takes in a Point as argument. Like Java, Point2D can be taken in where Point is expected because Point2D is one kind of Point. However, the cost of this inheritance approach is not minimized.

The value passed to doSomething() for p might be the subclass Point2D of Point, or it may be some other subclasses. One only knows for sure during runtime when the value actually gets passed. It is more costly because the class Point2D may have its own implementation of some methods, for instance toString(). Specific class information has to be determined before the right method gets called.

If there were no inheritance hierarchy, the compiler itself could already determine the correct code to call, which would completely eliminate the lookup cost during runtime. That is exactly where *struct* comes in to play.

However, using *struct* requires some care:

1. No *new* keyword is required when initializing a *struct* (unlike class).
2. For *struct*, s1==s2 means all their fields are equal, whereas for classes, c1==c2 means they reference the exact same piece of storage. In that sense, the "equal" sign is less contingent for *struct*.
3. There are no references to instances of a *struct*, because a *struct* contains all the fields within itself. It is neither a reference, nor does it require a reference.

2.3.5 Distribution in X10

X10 is a language designed for distributed computing, so the ability to scale computation into distributed systems is essential. X10 provides several necessary concepts to ensure that heavy-duty concurrency can be achieved.

1. Place: an address space in which activities (like threads in Java) may run.

- No two Places have any storage in common.
- An activity at one Place may refer directly to storage at another. [6]

2. At: to designate the Place for execution by 'at(p)' where p is a Place.
- For instance:

```
val result = at(p) doSomething();
```

will cause the runtime to pause the calling activity and go to Place *p*. Then it will call *doSomething()* and send the output back to this *Place* and assign it to the final result. After that, the paused activity will continue.

3. Async: used to spawn another activity without the need to wait for current activity to finish.

4. Finish: enclose a block of code to ensure that all activities inside such a block have finished before continuing outside the block.

2.3.6 X10's Race-Condition Prevention Mechanism

Besides having types like Double and Lon, X10 also has distributed computing specific types like *atomic*. For example:

```
atomic count += 1;
```

As self-explanatory as the declaration statement already is, the variable count can only be updated atomically, which means if one thread is updating the variable, all other threads will be locked out until the first thread finishes updating and releases the lock.
A better way to do it is using AtomicLong:

```
val count = new AtomicLong(0);
```

In this case, the count is declared as an *AtomicLong* type, which is more specific than *atomic*. Even though the runtime system can figure out the specific type of the variable *count* eventually in the first example, it always takes time to infer that information.

3. Performance Comparisons

X10 is in a stage where it borrows the runtime mechanism from either C++ or Java. That means X10 can be compiled into either Java byte codes or C++ binaries. Its higher level nature may indicate a possibility of performance loss. Whether X10's design concept is actually helping the performance should be tested with real world performance observations.

As of June 2011, X10's Java-backend is still significantly faster than the C++ -backend. [7] Therefore, the comparison will be X10's best performance that comes out of its Java-backend implementation versus its counterpart in original Java code. The X10 experiment is based on X10 version 2.2 specifications [8]. Due to constant modifications between X10 versions, programs written under version 2.2 are not compatible with compilers of early versions.

3.1 Presentation Format

The same presentation format will be applied on every implementation to ensure a clear comparison:

1. Algorithm description: a brief description of the nature of the algorithm and its run time characteristics.
2. Input and output: defines the type of input and expected type of output.
3. Java multi-threaded implementation with specified samples of input and corresponding output collected.
4. X10 implementation with same set of samples of input and corresponding output collected.

3.2 MergeSort

3.2.1 Algorithm Description

MergeSort is a well-known algorithm based on the divide-and-conquer concept. It divides the unsorted list into n sub-lists, each containing 1 element. Then it repeatedly merges sub-lists to produce new sub-lists until there is only 1 sub-list remaining. [9]

3.2.2 Input and Output

Input: an array of size n of unsorted elements

Output: a sorted array and the time spent sorting

3.2.3 Java Multi-Threaded Implementation

The multi-threaded implementation will adopt the same merge() operation when merging two sequences of sorted numbers. The difference from the sequential approach though is that more than one thread will be responsible for the sub-sequences, so that they can be executed at the same time.

A MergeRunnable class is defined to make sure each MergeRunnable is responsible for a certain sub-sequence.

```
class MergeRunnable implements Runnable{
        public void run(){
                // Do this runnable's share of merge sort
        }
}
```

By creating threads with different runnables, a sequence can be sorted after each thread has finished its task and merged together. A two threaded implementation with its running time being recorded is as follows:

```
//Lines needed for creating Java Threads can also be spared when coding X10
Thread thread1, thread2;
thread1 = new Thread(new MergeRunnable(Array8, 0, size/2, 4));
thread2 = new Thread(new MergeRunnable(Array8, size/2+1, size-1, 4));

startTime = System.currentTimeMillis();
thread1.start();
thread2.start();
try{
    thread1.join();
    thread2.join();
}
catch(InterruptedException ie){
    System.err.println(ie.toString());
}
```

For the above implementation, sequences of random numbers will be the input and the resulting running time elapsed will be recorded. The result is shown in Table 1 along with the result of X10 implementation.

3.2.4 X10 Implementation

X10 has its own way of declaring arrays, which will be evaluated more in later sections:

```
//Create arrays to be sorted
val arraySize = Int.parse(args(0));
val region = 1..arraySize;
var array:Array[Int] = new Array[Int](regionArray, (Point)=>0);
```

The algorithmic execution stays the same with its Java implementation counterpart to ensure the comparison is fair. The same timing mechanism is utilized to record its execution time as well.

```
startTime = t.milliTime();
finish{
    async m.mergeSort(Array2, 0, arraySize/2);
    async m.mergeSort(Array2, arraySize/2+1, arraySize-1);
}
m.merge(Array2, 0, arraySize/2, arraySize-1);
endTime = t.milliTime();
```

3.2.5 Result Comparison

Table 1 is the result based on different input sizes as well as both Java and X10 implementations:

SIZE	Java Sequential	Java Parallel x2	X10 Sequential	X10 Parallel x2
2500	6	6	135	40
5000	14	4	168	15
10000	23	9	178	21
20000	20	4	194	50
40000	20	10	239	92
80000	27	14	317	167
160000	43	28	528	349
320000	90	36	881	668
640000	153	82	1567	1320
1280000	273	136	2949	2511
2560000	629	260	5661	5322
5120000	1019	496	11446	10969

Table 1: Result on MergeSort using Java/X10 (In Milliseconds)

It is shown clearly, that in this case, X10 takes almost 10 times the amount of time to finish the same task, which is disappointing. However, some additional tweaks may be applied to better take advantage of X10's nature. A little tweak will reveal a subtle but crucial point when programming in the X10 language.

3.2.6 Optimizing X10 Implementation

The above comparison is disappointing on X10's part. The reasons can be listed after some investigation:

1. X10's Array class is more general than Java's. Array in X10 supports multi-dimensional arrays over arbitrary regions. It is very important to tell the X10 compiler statically that the Array is a 1-dimensional, dense, zero-based array in the above experiment.
2. When X10 code with tight loops over arrays is compiled, the '-O' option is very helpful in enhancing the performance of the loop.

With the above issues being taken care of, the X10 implementation of MergeSort is modified to the following:

```
//Create an array to be sorted.
val arraySize = Int.parse(args(0));
var Array1:Rail[Int] = new Rail[Int](arraySize);
var Array2:Rail[Int] = new Rail[Int](arraySize);
```

Rail[T] is a typedef for *Array*[T]{*rank==1, zeroBased, rect*}, where the rank of the array is 1 (one dimensional) and the base is 0 (first index starts from 0). Accordingly, all the indexes in the above implementation are updated to adapt that.

The result is shown in Table 2 and Table 3. In addition to the optimized X10 code, the thread number is scaled up to 8 this time.

Testing Machine: **CPU: AMD FX-8120 Eight-Core** **RAM: 8GB**

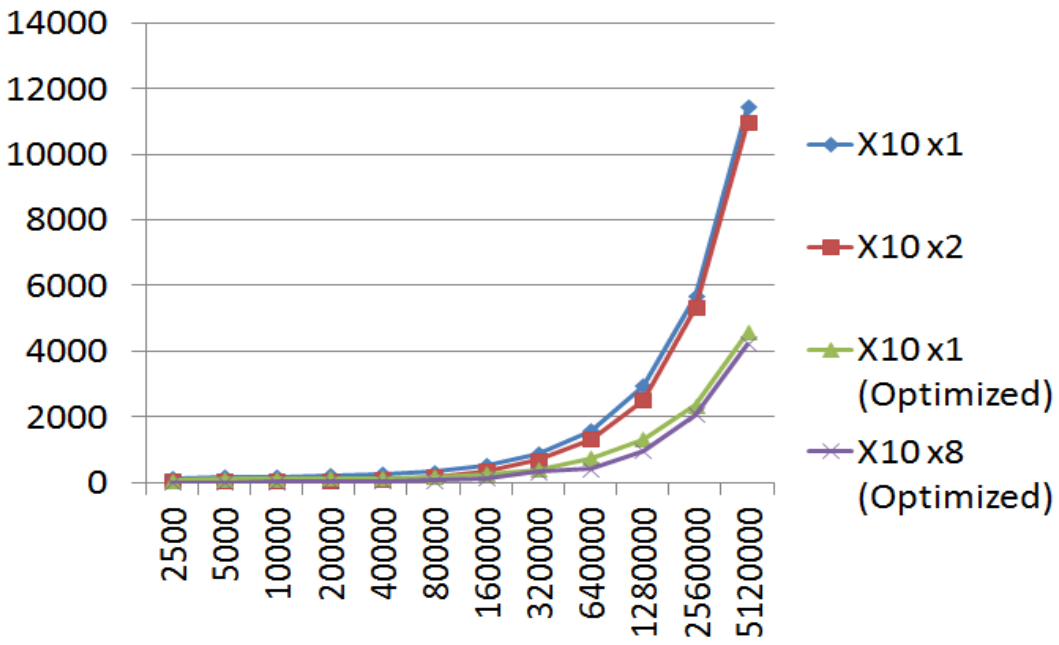
L2 Cache: 8MB **L3 Cache: 8GB**

SIZE	Java Sequential	Java Parallel x2	Java Parallel x4	Java Parallel x8
2500	6	6	7	6
5000	14	4	2	2
10000	23	9	12	10
20000	20	4	2	4
40000	20	10	7	6
80000	27	14	13	6
160000	43	28	12	18
320000	90	36	32	27
640000	153	82	52	57
1280000	273	136	91	95
2560000	629	260	166	125
5120000	1019	496	496	271

Table 2: Result of MergeSort using Java (In Milliseconds)

SIZE	X10 Sequential	X10 Parallel x2	X10 Parallel x4	X10 Parallel x8
2500	53	41	14	2
5000	94	16	6	4
10000	98	6	19	6
20000	110	18	25	21
40000	133	42	41	28
80000	170	70	77	54
160000	255	140	151	110
320000	397	274	271	314
640000	727	582	461	407
1280000	1301	970	990	965
2560000	2342	2006	2102	2066
5120000	4582	4365	4261	4200

Table 3: Optimized Result of MergeSort using X10 (In Milliseconds)

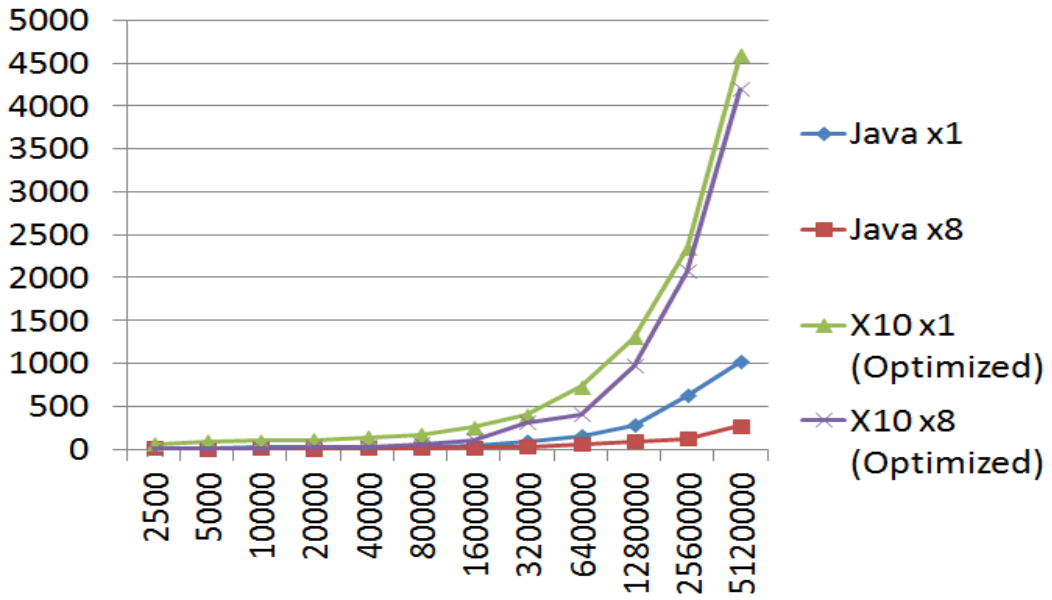


(Vertical Axis in miliseconds; Horizontal Axis represents number of input)

Chart 1: Comparison of X10 Performance Before and After Optimization

In comparison to non-optimized X10 code, the optimized code does have a significantly better result with 4365 milliseconds as opposed to 10969 milliseconds on the same size of 5,120,000 elements when using two threads.

However, it is still significantly slower than Java's implementation, both sequential and parallel. Also, the parallel X10 code is noticeably faster than its sequential counterpart only until a certain size. With a 40000 elements array, the eight-thread parallel implementation takes 21.05% of what the sequential implementation takes to finish, while with a 5,120,000 input size, the number is 91.66%. With Java's 26.59% on the exact same input size, X10's parallel performance still has a long way to improve.



(Vertical Axis in miliseconds; Horizontal Axis represents number of input)

Chart 2: Comparison of performance between Java and X10

3.3 QuickSort

3.3.1 Algorithm Description

QuickSort, also known as "partition-exchange sort", is a comparison sort that requires O(n*log n*) comparisons on average and, in efficient implementations, is not a stable sort. QuickSort can be implemented with an in-place partitioning algorithm, so the entire sort can be done with only O(*log n*) additional space [10].

3.3.2 Input and Output

Input: an array of size n of unsorted elements

Output: a sorted array and the time spent sorting

3.3.3 Java Multi-Threaded Implementation

Similar to the implementation of MergeSort, each thread is responsible for a certain portion of the sequence, except that for QuickSort, no merging is required.

In QuickSorter.java, the actual quickSort method is defined:

```
public void quickSort(int p, int r){
        if(p<r){
            Splitter sp = new Splitter(array);
            int i = sp.split(p, r);
            quickSort(p, i-1);
            quickSort(i+1, r);
        }
}
```

QuickRunnable.java is created for Java's multi-thread implementation (X10 implementation can spare those lines of code because it does not need any *Runnable*s):

```
public QuickRunnable(int[] array, int p, int r){
    this.array = array;
    this.p = p;
    this.r = r;
}

public void run(){
    QuickSorter qs = new QuickSorter(array);
    qs.quickSort(p, r);
}
```

When the program is running, same timing mechanism is used:

```
QuickRunnable less = new QuickRunnable(parallelArray, 0, bound);
QuickRunnable more = new QuickRunnable(parallelArray, bound+1, size-1);
Thread lthread = new Thread(less);
Thread mthread = new Thread(more);

startTime = System.currentTimeMillis();
lthread.start();
mthread.start();
try{
    lthread.join();
    mthread.join();
}
catch(InterruptedException ie){
    ie.printStackTrace();
}
```

The result is shown in Table 4 along with the result of the X10 implementation.

3.3.4 X10 Implementation

With exactly the same algorithm, the X10 implementation will use *async* to spawn out threads. This time *Rail*[T], along with other optimizing techniques, are used:

```
if(p < r){
    var sp:Splitter = new Splitter();
    var i:Int = sp.split(array, p, r);
    if(thread == 1){
        quickSort(array, p, i-1);
        quickSort(array, i+1, r);
    }
    else{
        finish{
            async quickSort(array, p, i-1, thread/2);
            async quickSort(array, i+1, r, thread/2);
    }
}
```

3.3.5 Result Comparison

Table 4 is the comparison results based on different input sizes on multiple runs:

SIZE	Java Sequential	Java Parallel x2	Java Parallel x4	Java Parallel x8
2500	9	6	3	4
5000	13	11	6	7
10000	22	9	7	6
20000	28	7	2	2
40000	42	3	4	4
80000	41	6	5	7
160000	46	22	24	18
320000	88	42	46	32
640000	181	111	79	69
1280000	695	342	303	288
2560000	1982	1349	1212	1016
5120000	9241	5077	3449	2659

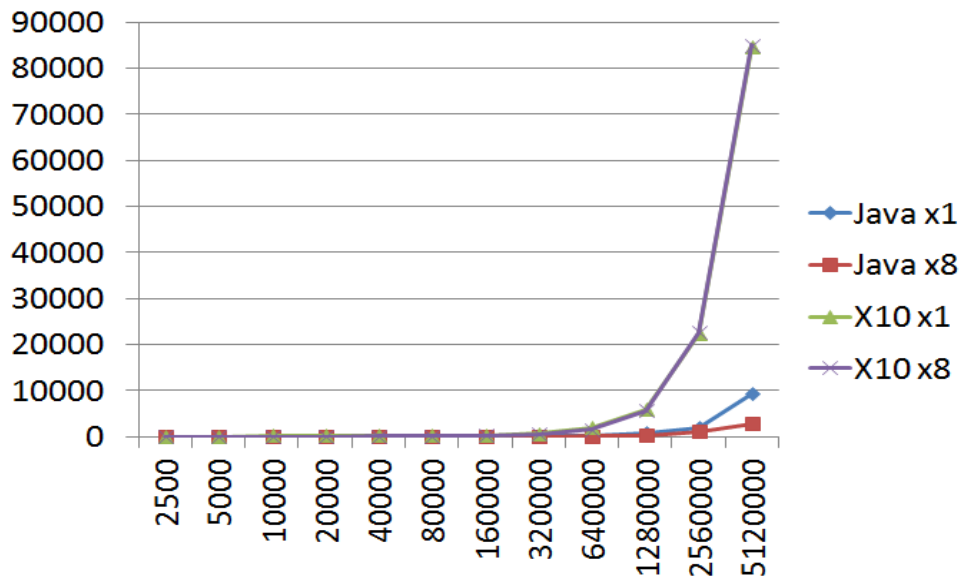
Table 4: Result of QuickSort based on Different Sizes using Java (In Milliseconds)

SIZE	X10 Sequential	X10 Parallel x2	X10 Parallel x4	X10 Parallel x8
2500	18	4	8	5
5000	25	7	20	3
10000	84	7	11	5
20000	86	10	17	8
40000	83	35	21	43
80000	110	69	57	70
160000	238	169	199	148
320000	638	535	575	454
640000	1848	1470	1499	1456
1280000	5869	5603	5655	5602
2560000	22193	22135	22033	22465
5120000	84469	84281	84753	85005

Table 5: Result of QuickSort based on Different Sizes using X10 (In Milliseconds)

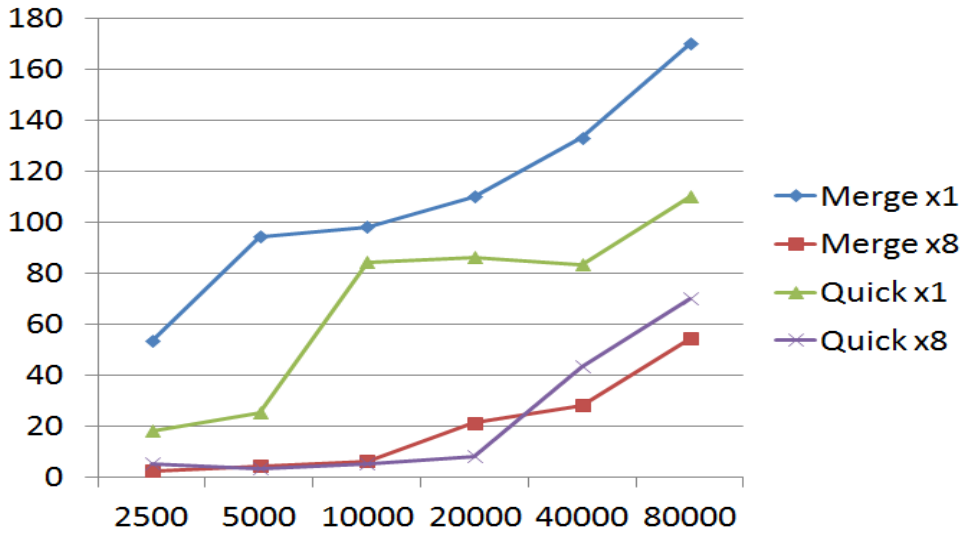
X10 has a similar performance as in the merge sort test case, which is one tenth of Java's performance. One interesting observation is that both programs' optimal input size for performance is between 80000 and 160000. This is due to the machine's CPU cache and memory where the AMD FX8120 has eight cores with 8MB L2 cache and 8MB L3 cache for each core. After the cache is filled up when taking in a large size of input, the speed will be noticeably slower when RAM kicks in.

Chart 3 and Chart 4 illustrate this difference clearly. With a 5,120,000-element array, X10's sequential and parallel implementations yield almost no different outcome; on the other hand, with fewer than 80,000 elements, the parallel version is truly faster.



(Vertical Axis in miliseconds; Horizontal Axis represents number of input)

Chart 3: Comparison of performance between Java and X10



(Vertical Axis in miliseconds; Horizontal Axis represents number of input)

Chart 4: Comparison of performance of X10 with Smaller Input Size

Even with faster parallel performance, which is expected, X10's performance on shared memory is still not comparable to Java's.

3.4 Strassen Matrix Multiplication

3.4.1 Algorithm Description

Strassen introduced an algorithm in 1969 to multiply M x M matrices, which has a lower complexity than the traditional $O(M^3)$ [11]. The algorithm is presented in Figure 1. It demonstrates the scheme for the product of two 2 x 2 matrices. This scheme involves seven multiplications and 18 additions instead of the usual eight multiplications and four additions for two 2 x 2 matrices.

Input : $Matrices\, A, B$

Phase 1

$$T_1 = A_{11} + A_{22} \qquad T_6 = B_{11} + B_{22}$$
$$T_2 = A_{21} + A_{22} \qquad T_7 = B_{12} - B_{22}$$
$$T_3 = A_{11} + A_{12} \qquad T_8 = B_{21} - B_{11}$$
$$T_4 = A_{21} - A_{11} \qquad T_9 = B_{11} + B_{12}$$
$$T_5 = A_{12} - A_{22} \qquad T_{10} = B_{21} + B_{22}$$

Phase 2

$$Q_1 = T_1 * T_6 \qquad Q_5 = T_3 * B_{22}$$
$$Q_2 = T_2 * B_{11} \qquad Q_6 = T_4 * T_9$$
$$Q_3 = A_{11} * T_7 \qquad Q_7 = T_5 * T_{10}$$
$$Q_4 = A_{22} * T_8$$

Phase 3

$$U_1 = Q_1 + Q_4 \qquad U_2 = Q_5 - Q_7$$
$$U_3 = Q_3 + Q_1 \qquad U_4 = Q_2 - Q_6$$
$$C_{11} = U_1 - U_2 \qquad C_{12} = Q_3 + Q_5$$
$$C_{21} = Q_2 + Q_4 \qquad C_{22} = U_3 - U_4$$

Output : $C = (C_{ij})$

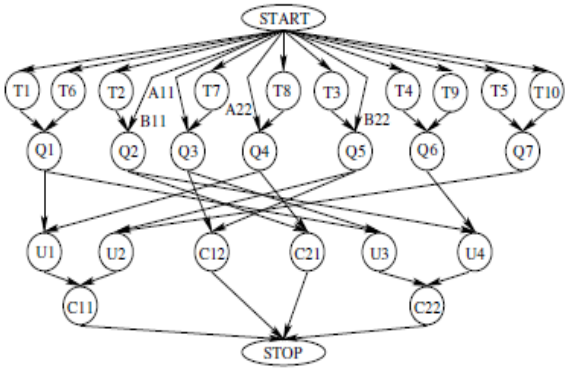


Figure 1: Strassen's Matrix Multiplication Algorithm

From a parallel processing point of view, the above calculations can be parallelized whenever no dependency of execution exists. For example, $T_1 = A_{11} + A_{22}$ and $T_6 = B_{11} + B_{22}$ are two executions that have no dependency upon one another. However, any calculations from different phases are subject to the dependency barrier. $Q_1 = T_1 * T_6$ has to wait until T_1 and T_6 get their values after Phase 1.

With that in mind, programs that parallelize the calculation of the product of two 16 x 16 matrices are written to compare between Java and X10.

3.4.2 Java Multi-Threaded Implementation

With Java, Threads and Runnables are used again for consistency throughout the experiment. For an eight-thread implementation, eight Runnables have to be constructed to get the job done.

```
MatrixRunnable mr1 = new MatrixRunnable(x,0,0,a,size);
MatrixRunnable mr2 = new MatrixRunnable(x,0,1,b,size);
MatrixRunnable mr3 = new MatrixRunnable(x,1,0,c,size);
MatrixRunnable mr4 = new MatrixRunnable(x,1,1,d,size);
MatrixRunnable mr5 = new MatrixRunnable(y,0,0,e,size);
MatrixRunnable mr6 = new MatrixRunnable(y,0,1,f,size);
MatrixRunnable mr7 = new MatrixRunnable(y,1,0,g,size);
MatrixRunnable mr8 = new MatrixRunnable(y,1,1,h,size);
```

Each thread is doing its own share of calculation in each phase of the Strassen Matrix Multiplication.

The result of Java's performance is shown in Table 6 along with X10's performance result.

3.4.3 X10 Implementation

Again, X10 implementation adapts the same algorithm with specific X10 programming language optimization:

```
finish{
    async submatrix(x,0,0,a,size);
    async submatrix(x,0,1,b,size);
    async submatrix(x,1,0,c,size);
    async submatrix(x,1,1,d,size);
    async submatrix(y,0,0,e,size);
    async submatrix(y,0,1,f,size);
    async submatrix(y,1,0,g,size);
    async submatrix(y,1,1,h,size);
}
```

Since the thread-spawning has to happen several times through the program. X10's "Runnable Free" syntax saves quite a few lines of code.

3.4.4 Result Comparison

Table 6 is the comparison results based on random generated matrices on multiple runs:

Run #	Java Sequential	Java x8	X10 Sequential	X10 x8
1	231	158	377	294
2	198	131	396	309
3	207	146	298	265
4	232	143	325	287
5	209	142	366	298
6	204	179	372	291
7	208	152	309	259
8	223	142	361	278
9	233	135	332	266
10	202	155	351	284
Average	214.7	148.3	348.7	283.1

Table 6: Result Comparison of Strassen's Matrix Multiplication (In Milliseconds)

With 16 x 16 matrices multiplication, the eight-thread parallel implementation of Java takes 69.07% of what the sequential implementation takes to finish, while X10 takes 81.19%. The parallel implementation is not significantly faster because only part of the algorithm is parallelizable; however, X10 still shows less impressive results compared to Java.

3.5 π Calculation

Another performance criterion is how efficiently each thread accesses shared atomic variables. [12] A comparison is conducted using the algorithm to compute **π**.

3.5.1 Monte Carlo Method

The Monte Carlo method was introduced in the 1940s. [13] It uses probability distribution as the means to compute **π**. For example, given a square of 1.0 x 1.0, one quarter of a circle is inscribed within, as shown in Figure 2. If objects are uniformly scattered within the square, the ratio of the number of objects in blue (inside the circle) and the number of objects in white (outside the circle) should equal π/4.

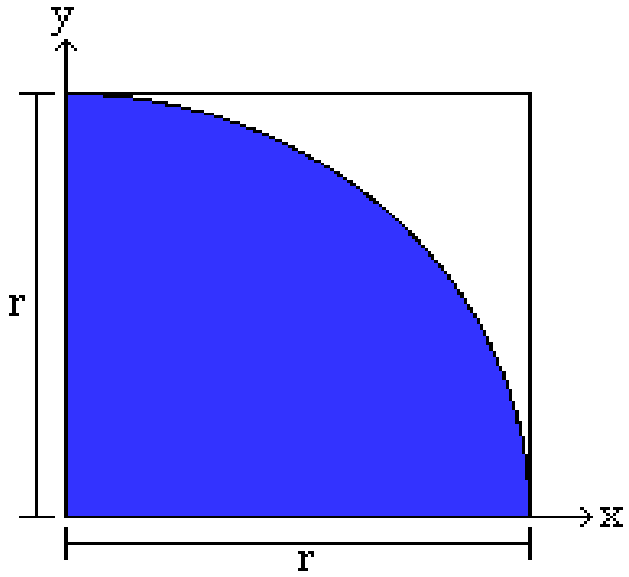


Figure 2: Monte Carlo method on π calculation

Because the Monte Carlo method is essentially about the count of objects, programs can be written to test the performance of the efficiency of how the count is updated. With that in mind, both Java and X10 implementations do not use local counts; instead, every

update of the shared count happens after one thread determines whether or not the object is inside the circle.

3.5.2 Java Multi-Threaded Implementation

Java's implementation creates four threads, each being forced to access the shared variable *count*:

```
public static class PiRunnable implements Runnable{
    private int n;

    public PiRunnable(int n){
        this.n = n;
    }

    @Override
    public void run() {
        Random ranGen = new Random();
        float x, y;
        for (int j=0; j<n; j++){
            x = ranGen.nextFloat();
            y = ranGen.nextFloat();
            if (inCircle(x, y))
                count.incrementAndGet();
        }
    }

    public boolean inCircle(float x, float y){
        return x*x+y*y <= 1.0 ? true : false;
    }
}
```

The result is shown in Table 7 along with X10's performance result.

3.5.3 X10 Implementation

X10 adopts the exact same implementation to coerce each thread to compete for one shared variable:

```
finish for(var k:Int = 0; k<4; k++){
    val r = new Random();
    async {
        var x:Float, y:Float;
        for (var l:Int=0; l<nPerThread; l++){
            x = r.nextFloat();
            y = r.nextFloat();
            if (x*x+y*y <= 1.0)
                count.incrementAndGet();
        }
    }
}
```

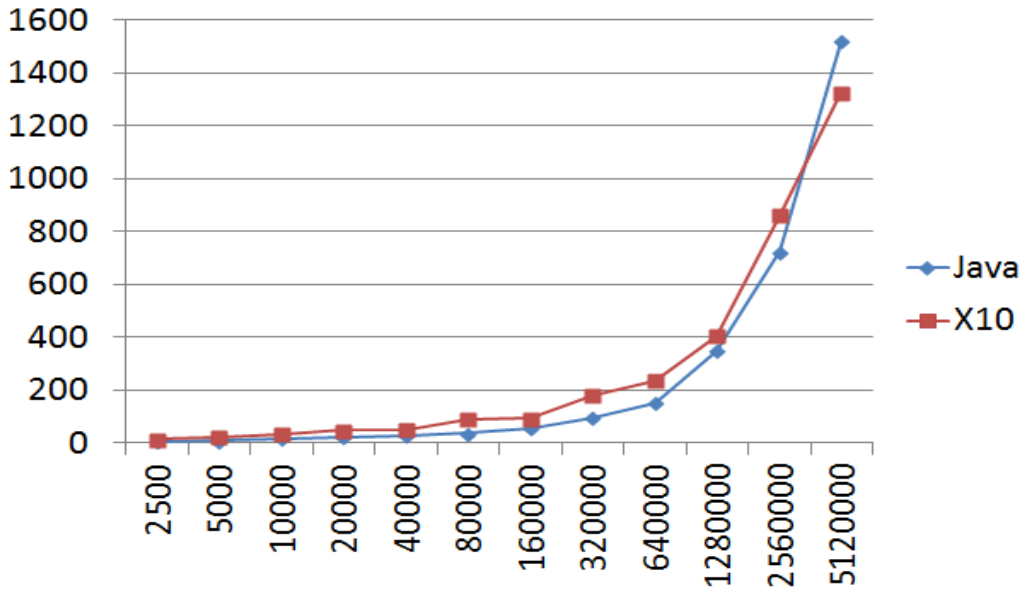
3.5.4 Result comparison

Table 7 is the comparison results based on different input sizes on multiple runs:

Size of Domain	Java	X10
2500	8	16
5000	9	23
10000	18	35
20000	23	49
40000	30	53
80000	38	91
160000	58	95
320000	98	182
640000	154	238
1280000	349	405
2560000	724	862
5120000	1523	1329

Table 7: Result Comparison of π calculation (In Milliseconds)

X10's performance shows a slight edge this time when the input size is very large.



(Vertical Axis in miliseconds; Horizontal Axis represents size of domain)

Chart 5: Comparison of performance of π calculation

On an input size of 1,000,000, where the atomic variable *count* gets updated for hundreds of thousands of time, X10's performance is significantly better:

Run #	Java	X10
1	4179	1960
2	3879	2123
3	4221	2052
4	4087	2050
5	4107	2037
6	3703	2005
7	4371	1923
8	4186	1988
9	3922	2068
10	4108	2000
Average	4076.3	2020.6

Table 8: Result Comparison of π calculation on input size of 1,000,000 (In Milliseconds)

As shown in Table 8, X10 takes 50.43% less time than Java to calculate π. Given the fact that X10 did poorly on almost all previous experiments, the better efficiency of updating the atomic value is very likely the reason X10 does well with π calculation in this case.

4. Programmability Comparison

Some of X10's programming and syntax differences were mentioned in earlier sections to illustrate X10's idealism towards distributed computing. This section will focus solely on the ease of programmability of Java and X10 in its present format.

4.1 Spawning and Synchronizing Threads

For concurrency programming, spawning and synchronizing threads is the portion of code most frequently written to support the distributed-computing purpose.

For Java, there are two ways to do this:

1. Use a *Runnable* object:

The *Runnable* interface defines a single method, *run()*, meant to contain the code executed in the thread. [14] With this approach, the runnable object has to be passed in as an argument to construct a *Thread* object. For example:

```
public class SampleRunnable implements Runnable{
        public void run(){
                //Do the task
        }
}
Thread t = new Thread(SampleRunnable);
t.start();
```

2. Subclass *Thread*:

The *Thread* class itself implements *Runnable*, though its *run()* method does nothing. An application can subclass the *Thread* class, providing its own implementation of *run()*. For example:

```
public class SampleThread extends Thread{
```

```
        public void run(){
                //Override default behavior and do the task
        }
}
(new SampleThread).start();
```

For Java, synchronizing threads would come after spawning and finishing the tasks:

```
try{
        thread1.join();
        thread2.join();
}
catch(InterruptedException ie){
        System.err.println(ie.toString());
}
```

Those few lines of code usually cannot be spared if one wants to create threads as a way to finish some tasks concurrently in Java.

For X10, spawning and synchronizing happens altogether with the following syntax:

```
finish{
        async{
                //Thread 1 execution
        }
        async{
                //Thread 2 execution
        }
        async{
                //Thread 3 execution
        }
}
// All threads are synced and joined, main thread ready to go....
```

It is clear that X10's syntax is much simpler. No *Thread*s or *Runnable*s need to be explicitly created, and no method needs to be overridden like in Java. Inside each *async* block, all the lines are normal statements as if they are from a sequential program. After all threads finish their execution, no *try{} catch{}* blocks are needed. With *finish{}* enclosing the parallel portion, all the threads will be synchronized after they exit the *finish{}* block.

4.2 Functions

Functions are used extensively in many programming languages. A function takes a set of inputs; does some calculations on the inputs; and then returns a set of results.

In Java, a function is declared as following:

```
public int calculateAnswer(int arg1, String arg2, double arg3){
   //do the calculation here
}
```

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, and a body between braces.

In X10, the basic syntax for functions is:

```
(arg1Type, arg2Type, ...) => returnType
```

For example:

```
var computeSum: (a:Array[Float](1)) => Float;
```

The value of the variable of computeSum -- or in other words, the body of the function computeSum -- will be a method that takes a singly-indexed array of Floats and returns a Float.

The function body may be a block like Java's function body. For instance, to compute an integer's square value, one can use addition repeatedly:

```
val square: (Int) => Int
        = (n:Int) => {
                var result: Int = 0;
                for (var i=0; i<n; i++)
                        result += n;
                return result;
        }
```

The declaration alone, (Int)=>Int, is neat; however, a definition of an X10's function has redundant syntax as both (Int)=>Int in the first line and (n:Int)=>{} in the rest of the lines are present.

The redundant syntax may have special purposes, one of them is to make sure that the function can be taken as an object.

4.2.1 Function as an Object

It is sometimes possible to need a function as an argument for another function's parameter list.

In Java, a common pattern would be to wrap a function body within an interface. For example, the interface Callable can achieve that purpose. Any class that implements Callable will have to override its call():

```
public T funcToBePassed(){
        //do something
}
public void funcTakesAnotherFunc(Callable<T> func){
        //do something
}
```

When calling the *funcTakeAnotherFunc*(), using an anonymous class will achieve the goal:

```
funcTakesAnotherFunc(new Callable<T>(){
        public T call(){
                return funcToBePassed();
        }
});
```

In X10, because a function has already been assigned to a named variable, the above task, which requires more care for Java, would be trivial for X10:

```
val r = new Random();
val rand = ()=>r.nextDouble();
val calculationUsingRandomNumber = calculateSomething(rand);
```

where *rand* in the above example is a function that takes no argument and returns a random number each time it is called.

4.3 Array

In Java, a one-dimensional array's declaration and assignment is as trivial as possible:

```
int[] anArray = new int[10];
anArray[0] = 100;
...
```

The case for two-dimensional arrays is not too much more complicated than that for one-dimensional arrays:

```
int[][] rank2Array = new int[10][15];
rank2Array[0][0] = 100;
...
```

or if the assignment has some rules, then the syntax should be as follows:

```
for(int i=0; i<10; i++){
        for(int j=0; j<15; j++)
                //assignment
}
```

However, 3-tuples, 4-tuples and even more tupled arrays will be complicated to initialize and present.

In X10, the array system introduces the concept of Points and Regions. A Point corresponds to an element in the array that is of the rank of n (n-tuples), while Regions are the domains within which the Point can define.

For one-dimensional arrays, the syntax of X10 is a bit cumbersome to use considering the simple nature of such arrays.

```
val region = 1...15; //assume an array of size 15
var anArray:Array[Int](1) = new Array[Int](1)(region, (Point)=>0);
```

For two dimensional arrays, the scale of the matrix is defined with Region:

```
var region = (1...10)*(1...15);
var anArray = new Array[Int](region, (Point)=>0);
```

A *for* loop can also be used for the assignment of the array.

```
for(int i=0; i<10; i++){
        for(int j=0; j<15; j++)
                //assignment
}
```

In fact, the syntax of both languages is similar in terms of complexity for *Array*s. X10 proposes the Region and Point approach for performance consideration. However, as shown in the MergeSort example, the Point implementation right now is still quite slow, especially for dense arrays. For dense arrays, better written code in terms of performance is as follows:

```
var anArray = new Rail[Int](15);
```

This is essentially a declaration of a Java-like array.

4.4 Comparison based on lines of code

To have a quantitative comparison of the syntax complexity between Java and X10, it is helpful to count the lines of code each programming language needs to accomplish the same task. It needs to be noted that the comparison is merely based on a few examples, so it is more of an evaluation than a definitive answer as to which language is more simplistic in syntax.

Table 9 shows the comparison based on the examples in section 3:

Program	Java (Full)	X10 (Full)	Java (Exclude Declaration)	X10 (Exclude Declaration)
Mergesort	161	99	116	71
Quicksort	173	110	130	82
Strassen	256	160	173	127
π Calculation	56	34	29	15

Table 9: Comparison of Syntax Complexity (In Number of Lines of Code)

It is shown that X10 needs fewer lines of code to execute the same task in all the test cases above. It certainly indicates the simplicity of X10's distributed-computing syntax to some degree.

5. Tool Support

As a programming language that has been around since 1995 [15], Java is one of the most popular programming languages in the software industry. Therefore, the tool support for Java is state of the art. Many major IDEs support Java in their native mode, for example Eclipse, NetBeans and IntelliJ. They provide editors for writing and editing programs, a syntax checker to statically check the program before compilation, and debuggers for locating logic errors [16].

On the other hand, as a newer programming language, X10 has a less-evolved tool support system [17]. As of now, X10DT is the only comprehensive development tool that supports X10's syntax and runtime debugging.

X10DT is built on top of Eclipse similar to JDT for Java. It allows programmers to edit, build, and launch the program. Help pages are also integrated into X10DT for X10 language help and X10DT usage help.

The IBM Parallel Debugger for X10 Programming [18] is integrated with X10DT for debugging purposes. It can assist the programmers to display X10 variables during runtime; set breakpoints and enable operations, such as step into, step over, pause, and resume. Although it sounds quite similar to what JDT debugger does for Java programming, the X10 debugger is much more rough-edged.

Even with X10DT, of which the primary goal is to support X10 development, some X10 specific functionalities are still missing. For example, "autocomplete" is a popular and, arguably, a must-have feature in Java development, but it is not in X10DT yet. Also, with X10DT's debugging tool, it is hard to navigate to the erroneous lines with its less detailed console information.

These limitations do affect the productivity when developing X10 programs. However, it is reasonable to expect the tool support of X10 to improve over time to solve all these issues.

6. Conclusion

X10 is a well thought-out programming language aiming to be at least a distributed-computing alternative to Java. Overall, its syntax is traditional with its object-oriented feature. Programmers with Java or C++ programming background will find the syntax familiar and easy to acquire. Beyond that, X10 has a very simplistic distributed computing syntax, which serves its purpose as a concurrent language very well.

Furthermore, X10 introduces some original features and concepts in order to define and handle the new problems encountered with distributed computing. The idea might be good, but the execution is not there yet.

To sum it up, X10, as a new language aiming at being a distributed-computing alternative, has the following advantages and disadvantages:

Advantages:

1. It builds from the ground up with the distributed-computing concept in mind.
2. It is well designed to get rid of some complexity related to distributed-computing programming.
3. It can be compiled into both Java bytecode and C++ binaries, which gives itself a broader platform.
4. Its syntax is similar to the most popular programming languages like Java and C++, so it will be relatively easy to adopt once it is matured.

Disadvantages:

1. Its performance in most aspects is still very poor compared to a sophisticated language like Java.
2. Its tool support is very poor with a minimum static syntax checker available and few debugging tools.

3. X10 had been changed back and forth quite a bit between version 1.x and 2.x. The same code from version 1.x does not run when adopting a 2.x version compiler.

It can be noted that all of X10's disadvantages are due to the fact that it is a very new programming language. After it matures with time, it should have no problem to be adopted considering its programmer friendly nature and structural distributed computing emphasis.

Reference

[1] Programming languages for Distributed Computing Systems. By Henri E. Bal, Jennifer G. Steiner and Andrew S. Tanenbaum. ACM Computing Surveys, Vol 21, No. 3, September 1989.

[2] X10: Performance and Productivity at Scale. *x10-lang.org*, October 2012.

[3] Evaluating Skandium's Divide-and-Conquer Skeleton. By Panagiotis Tsogkas. School of Informatics, University of Edinburgh.

[4] Preemptive and Cooperative Thread Models. SoftVelocity Inc., 2007.

[5] A Performance Model for X10 application. By David Grove, Olivier Tardieu, David Cunningham, Ben Herta, Igor Peshansky and Vijay Saraswat. IBM Research.

[6] An Introduction To Programming With X10. By Jonathan Brezin, Stephen J. Fink, Bard Bloom and Cal Swart. IBM Research, December 2010.

[7]Compiling X10 to Java. by Mikio Takeuchi, Yuki Makino, Kiyokuni Kawachiya, Hiroshi Horii, Toyotaro Suzumura, Toshio Suganuma, and Tamiya Onodera. ACM SIGPLAN 2011 X10 Workshop, June 2011.

[8] X10 Language Specification Version 2.2. By Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove, July 2012.

[9]Merge sort Wikipedia. http://en.wikipedia.org/wiki/Merge_sort, Retrieved October 25, 2012.

[10]Quicksort Wikipedia. http://en.wikipedia.org/wiki/Quick_sort, Retrieved October 25, 2012.

[11] Gaussian Elimination is not Optimal. By V. Strassen. Numerische Mathematik 1969, 14(3):354.

[12] Automatic Skeleton-Driven Performance Optimizations for Transactional Memory. By Luis Fabricio Wanderley Goes. Institute of Computing Systems Architecture, School of Informatics, University of Edinburgh, 2012.

[13] Monte Carlo method Wikipedia. http://en.wikipedia.org/wiki/Monte_Carlo_method, Retrieved October 26, 2012.

[14] The Java Tutorials. By Oracle. http://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html, Retrieved October 11, 2012.

[15] The History of Java Technology. http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html, Retrieved October 26, 2012.

[16] Java for Programmers: Deitel Developer Series. By Paul Deitel. Deitel & Associates, Inc., April 18, 2011.

[17] An Overview of the X10 Programming Language and X10 Development Tools. By Evelyn Duesterwald, Emmanuel Geay, Vijay Saraswat and David Grove. IBM Research.

[18] IBM Parallel Debugger for X10 Programming. IBM Developer Works. https://www.ibm.com/developerworks/mydeveloperworks/groups/service/html/communityview?communityUuid=e3fb8100-3ee3-4402-bf67-bf66b29797ea, Retrieved November 2, 2012.