Go, F# and Erlang

Anders Järleberg
Sjöviksvägen 13
117 60 Stockholm

Kim Nilsson
Professorsslingan 10
114 17 Stockholm

May 20, 2012

Abstract

This report examines the three languages Erlang, F# and Go, which all have some form of inherent support for concurrency. The purpose was to determine the strengths of these three languages and in which situations they are suitable. We examine the performance and scaling of some parallel algorithms and compare it to a sequential version written in Java and discuss some factors relating to ease of implementation and maintainability.

We found that we could achieve a high degree of scaling on a multicore platform, with relatively simple methods. In most practical scenarios, however, it is likely not the performance and scaling that determines what language is the best choice, but instead some combination of features within the language itself as well as external factors such as availability of standard libraries or platform dependence.

Sammanfattning

Denna rapport undersöker de tre språken Erlang, F# och Go, som alla har någon form av stöd för parallell exekvering. Syftet med rapporten var att undersöka styrkorna hos dessa tre språk samt i vilka situationer de är lämpliga. Vi mäter prestanda och skalbarhet hos ett par olika parallella algoritmer och jämför dessa med en sekventiell version skriven i Java. Vi undersöker även faktorer rörande underhåll av koden.

Vi fann att vi kunde uppnå en hög skalbarhet på en flerkärnig plattform på ett relativt enkelt sätt. I de flesta praktiska situationerna är det dock sannolikt inte prestandan och skalbarheten som bestämmer vilket språk som är mest lämpligt, utan istället en kombination av språkets funktionalitet samt yttre faktorer som standardbibliotek eller plattformsberoende.

Contents

1 Introduction **6**

1.1 Purpose . . . 6

1.2 Problem statement . . . 6

1.3 Statement of collaboration . . . 7

2 Background **8**

2.1 The evolution of CPUs . . . 8

2.2 Multi-core programming . . . 8

2.3 Maintainability . . . 9

2.3.1 The code base . . . 9

2.3.2 The language itself . . . 10

2.3.3 Hardware support . . . 10

2.4 Languages . . . 10

2.4.1 Erlang . . . 10

2.4.2 Go . . . 11

2.4.3 F# . . . 12

2.4.4 Java . . . 12

3 Method **14**

3.1 Algorithms . . . 15

3.1.1 Matrix multiplication . . . 15

3.1.2 Finding prime numbers . . . 15

4 Results **17**

4.1 Matrix multiplication . . . 17

4.1.1 Erlang . . . 17

4.1.2 F# . . . 18

4.1.3 Java . . . 19

4.2 Prime number finder . . . 20

4.2.1 F# . . . 20

4.2.2 Go . . . 21

4.2.3 Java . . . 22

5 Discussion **23**

5.1 Performance and scaling . . . 23

5.2 Maintainability and ease of implementation . . . 24

6 Conclusion **26**

7 References **27**

A Code **29**
A.1 Erlang matrix multiplication implementation 29
A.1.1 matrix.erl 29
A.2 F# matrix multiplication implementation 31
A.2.1 Matrix.fs 31
A.3 Java matrix multiplication implementation 32
A.3.1 Matrix.java 32
A.3.2 Main.java 35
A.4 F# primes implementation 36
A.4.1 Program.fs 36
A.5 Go primes implementation 37
A.5.1 primes.go 37
A.6 Java primes implementation 38
A.6.1 PrimeTest.java 38

B Tables **39**
B.1 Matrix multiplication 39
B.1.1 Erlang 39
B.1.2 F# 39
B.1.3 Java 40
B.1.4 Java with -Xint flag 40
B.2 Prime number finder 40
B.2.1 F# 40
B.2.2 Go 42
B.2.3 Java 42
B.2.4 Java with -Xint flag 42

1 Introduction

As a civil engineer in computer science one is confronted by a multitude of different problems with a great amount of possible solutions. We are interested in when and why a certain method of tackling the problem is superior – whether it be the programming language, paradigm or something else – and what 'superior' actually means in the context. It could mean scalability, ease of implementation, performance, maintainability or something else. What identifies an ideal problem approach and what gains can be had versus others?

1.1 Purpose

The purpose of this report is to analyze the three programming languages from different angles on a couple of algorithms to, if possible, determine strengths and weaknesses. In the process, we hope to find some way of identifying when and why a given approach is best.

1.2 Problem statement

In effect, we wish to answer the rather naive question; "Which is the best programming language?" Of course, this is highly dependent of the situation, the problem and what the goals are. In most situations, however, there is some language that is better suited in terms of scalability, ease of implementation, performance, maintainability or some other factor than others.

The main questions we wish to answer are:

- What makes a language better than others in the aforementioned factors in a specific situation?
- What inherent qualities of the problem or the language makes it a better choice and why?
- What are the actual expected gains in a typical situation?
- Is the difference in the ease of implementation or maintainability worth it in terms of end-performance and scalability?

Most of these can not be objectively established. For example, the ease of implementing a certain algorithm is a subjective matter, based on the programmer's opinion, prior knowledge and skill. We will attempt to summarize the arguments for and against a given language, but our main focus will be the experimental results based on scalability and performance. In practice, this means we will be testing the performance of the virtual machines of the given languages and their ability to scale, rather than the actual language itself.

To determine the expected gains we will be comparing the performance of parallel algorithms implemented in these languages with a sequential version implemented in Java.

1.3 Statement of collaboration

The authors in this essay have divided the tasks between them. The matrix multiplication implementations in Erlang and Java were done by Kim Nilsson, while the prime number implementations in Go and Java were done by Anders Järleberg.

The F# implementation of matrix multiplication was based off Yin Zhu's implementation.[1]

The F# implementation of the prime number algorithm was based off the example given on Wikipedia.[2]

The text in this essay has also been divided between the authors, see table 1.

Table 1: Table showing what parts were written by whom. AJ stands for Anders Järleberg, KN stands for Kim Nilsson.

Section	Author(s)
1.x	Both
2.1	KN
2.2	Both
2.3	KN
2.4.1	KN
2.4.2-3	AJ
2.4.4	KN
3	AJ
3.1.1	KN
3.1.2	AJ
4.x	Both
5.1	AJ
5.2	Both
6	Both

2 Background

The nature of computing is changing. We have gone from terminals with central computers to single CPU workstations, to the current trend of cloud-based computing and multi-core CPU workstations. This brings along changes to the way programs are designed and developed. To make efficient use of the platforms at hand, it is important that the programs fit into the established framework. This involves a multi-core approach, but other factors that must be taken into account for future development include maintainability and ease of implementation.

There are a few different paradigms of programming. Historically, the most common paradigm is imperative programming. This involves telling the CPU exactly what instructions to perform in a sequence to get a result. Declarative programming involves stating what one wants, not necessarily how to get it. A subset of declarative programming is functional programming, which takes a more mathematical approach, with a focus on functions and recursion. Generally, a functional programming language is easier to use for parallel processes than an imperative one; we will see why later.

2.1 The evolution of CPUs

To make computer programs execute as fast as possible it is important to have a fast CPU. Generally speaking, the higher clock frequency the CPU has, the faster it can execute machine instructions and produce results. Therefore, we have since the dawn of the CPU, seen a steady increase of the clock frequency. In 1971 Intel released the Intel 4004 CPU with a clock frequency of at most 740kHz.[3] As of today, Intel's latest model has a clock frequency at around 4GHz – an increase in frequency by a factor of almost 5500.

To be able to function under a higher clock frequency the transistors in the processors must become smaller. The problem is that we can not make them infinitely small. As the size of the transistors come closer to the size of the actual atoms, effects such as quantum tunneling will prevent them from working as intended.[4] Another problem is that with the increase in frequency, the amount of power (and therefore heat) actually increases faster than the frequency gain.[5] In the long run, this is not a sustainable relation.

Due to these factors, we have in recent years already started to see a stagnation in clock frequency speeds.[6] Instead it has become more common to increase the number of actual CPU cores in processors. These so called multi-core processors can execute several machine instructions at the same time – which gives rise to true parallelism.

2.2 Multi-core programming

Having more than one CPU core raises a fundamental problem to programmers if you want to make your programs run as efficiently as possible. In order to be able to use several cores at the same time, you must design your program, if at all possible, in such a way that certain tasks of the program are independent of each other, so that they are concurrent. In general this is quite hard to do, especially since most of the commonly used programming languages follow an imperative programming paradigm. Today, many programs using imperative languages are written in a way that promotes sequential execution of the code. This results in a high dependency between different sets of instructions which typically can only make use of a single core.

However, there are other languages that, due to their design, remedy this to some degree. Programming languages that follow the declarative programming paradigm work a bit differently. Since the programmer does not specify exactly what sequence of instructions must be performed, the compiler is free to divide mutually independent instructions which can be executed simultaneously on different threads.

In addition, functional programming does not involve any side effects by design, which makes parallelization even easier due to a small amount of dependency outside of functions.

2.3 Maintainability

Performance is not everything. This is true for a lot of programs. For example it does not matter if it takes 10 or 15 ms to generate a diagram in your text editor. Rather than spending resources on optimizing the code, the software developer might focus on getting releases out on the market faster or producing a more stable release with additional features instead.

2.3.1 The code base

A lot of today's projects have a huge code base. It is not uncommon to have millions of lines of code. Therefore it is very important to keep the code base in a good condition to make it easy to maintain and to make further developments. There are a few things a program language can provide to aid the programmer in making the code more easily maintained. For example, object oriented programming encourages the programmer to design the program in a way that keeps cohesion high and coupling low. This can greatly improve the maintainability of the code.[7] Some languages such as Java follow the object oriented paradigm very thoroughly.

Even if you have good programmers you will eventually end up with bugs in your code. In most cases this is fine. All you have to do is find the bug, write a software patch and update your software. But in certain systems this can pose a real problem. Some critical systems must have close to a 100% uptime and a reboot of the system might not be accepted. These systems must be able to get updated software without shutting down any services – this is called *hot-swapping*. Some languages provide support for hot-swapping code. This can be very useful as it allows programmers to patch running software with bug fixes without the need to shut it down or even restart the program.

2.3.2 The language itself

It is most likely easier and cheaper to find a skilled programmer in a popular programming language than an exotic one. This is very important because you can not maintain something you do not understand. This becomes an even more important factor as time goes by. If it is easy to acquire skilled Erlang programmers today - what about in 5, 10 or even 20 years?

Another important language specific factor is the user base. Is the language community active? This might prove to be very important because an active community will make the programming language thrive and develop further. It also greatly aids the learning process if there is an active community. The number of Google hits a language results in might not seem too important at first glance but it might prove to be very important when you are stuck working on a difficult problem.

Different languages come with different standard libraries. Some are very well developed, such as Java's which is maintained and frequently updated by the Java community. Other programming language might have a more sparse standard library putting a heavier burden on the programmers themselves. This can increase the time and money spent in having to manually code what is otherwise available in other languages' standard libraries.

2.3.3 Hardware support

As hardware continues to develop and we move our code to new platforms we take for granted that our code will be able to run well on our new platform. In most cases this requires a port of a virtual machine or a compiler to the new system in order for the code to run. An old and forgotten language with a dead community might not get this port rendering the software obsolete for new platforms.

2.4 Languages

In the following sections we will present the languages and give some background and history for them.

2.4.1 Erlang

Erlang is a functional programming language and it is, since 1998, an open source project.[8] The development of Erlang began in 1986 at Ericsson by Joe Armstrong. Erlang is compiled to byte code which runs on a virtual machine. It is also possible to compile to native code which in some cases will give a substantial performance gain. It is dynamically, strongly typed. The syntax of Erlang is greatly influenced by Prolog.[9]

The goal with Erlang was to create a programming language suitable for creating highly parallel, distributed and fault-tolerant systems. It tries to do so by implementing a technique called message passing. This means that in Erlang, you can spawn lightweight processes that are completely independent of each other and the only form of communication that can exist between two or more processes is by sending each other messages.[10]

Since different processes do not share memory with each other programmers do not have to worry about dead-locks and race conditions when it comes to making programs run efficient on multi-core CPUs. This is what often makes Erlang a good match for problems that are of a parallel nature. Another important feature of Erlang is that it is possible to hot-swap code during runtime. This can be crucial for server applications that require a high uptime.

The interest in Erlang has grown lately.[11] The reason for this is probably due to its message passing style and how it simplifies concurrent programming. Erlang is not very common in the industry but can be seen occasionally, especially when it comes to networking hardware such as routers. For example, a very successful ATM switch from Ericsson runs Erlang and as a proof of its stability it has an uptime of 99.999999999 (nine nines).[12]

2.4.2 Go

Go is an open source programming language developed by Google Inc, initially designed in 2007.[13] It is an attempt to mix the ease of an interpreted language with dynamic types combined with the efficiency of a compiled language with static types. Go provides inherent support for concurrency, garbage collection, and promotes a multi-core approach.[13] Compilation is fast and Go provides easier dependency management than, for example, C. The designers behind Go felt that current languages were lacking in several of these areas, and often required sacrifices to be made to get more than one of these features. They wanted to make development be faster and targeting multi-core systems to be an inherent part of the design of the language itself.[13]

Go has no type hierarchy and no class inheritance. The designers say this makes the language more lightweight and easier to use by not having to explicitly state relationships between types and making the syntax easier to read.[13]

Go uses something they call *goroutines*, which are basically like separate functions that can be executed in parallel. They communicate through *channels*. These can be buffered or unbuffered, and separate goroutines can send or receive messages on the channels, which allows the programmer to easily define synchronization or locks between functions without needing to know the exact low-level workings of the execution.[14]

Goroutines may not necessarily end up running in parallel on separate threads, but can be switched between during execution based on messages sent or received over channels. If, for example, a goroutine is blocked (perhaps waiting for a system call to finish), other goroutines that are running in parallel can be moved to another thread and continue execution so they are not unnecessarily blocked by the first routine. At this point in the language's development, if one wants parallel execution, the programmer has to specify how many CPU cores may be used with a call to the built-in package function `runtime.GOMAXPROCS(int)`.[13]

2.4.3 F#

F# is a multi-paradigm programming language designed by Microsoft – it has elements of functional, imperative and object oriented paradigms.[15] It fully supports the .NET Framework and is included in Visual Studio 2010. F#'s first version came out in 2005, but it is a variant of an older functional programming language known as ML.

F# is strongly typed and has type inference, so the programmer does not need to specify types for variables or parameters unless one wants to – the compiler will otherwise do this automatically.[15] By default, variables are immutable and the language is of a functional nature (no side effects, all functions return values, etc.), but also offers mutable variables and imperative programming (changing variables, ignoring return values, etc.) as well as object oriented approaches. Since F# targets the .NET Framework, it has full support for .NET objects and modules, so it can be used to make Windows Forms and similar programs in addition to running from a command line or an interpreter.[15]

The language has inherent support for structures like tuples (two or more values coupled together), lists and types. It also features pattern matching, curried functions, function composition and lambda functions, much like other functional programming languages.[15]

One interesting feature of F# is the *async* keyword. It allows the programmer to define tasks that can be performed asynchronously, such that they can be performed without having to wait for the instructions to finish before continuing. One can set up a list of asynchronous functions and let them run in parallel by using the `Async.Parallel` function.[16] During run-time, these functions will be evaluated on different threads. Clearly, F# is suitable for programs that make heavy use of parallel execution.

2.4.4 Java

The first version of the Java programming language was released around the year 1995.[17] It was developed by Sun Microsystems. Java is an imperative, multi-paradigm language. It is static and strongly typed. Java's syntax looks a lot like the syntax of C++. This was an important design decision so that it would be easy for C++ programmers to start using Java.

Java was designed with the following five principles in mind.[18]

- It should be "simple, object-oriented and familiar"
- It should be "robust and secure"
- It should be "architecture-neutral and portable"
- It should execute with "high performance"
- It should be "interpreted, threaded, and dynamic"

When a Java program gets compiled the source code is transformed into bytecode. The bytecode is the same for all computer architectures and it gets executed by a Java Virtual Machine, JVM. This makes the Java language very portable. You never have to port your program to another platform, you only need to port the JVM and this has already been done to most platforms. This was emphasized by Java's slogan: "Write once, run anywhere".[18]

Another core feature of Java is its automatic garbage collector. The garbage collector is responsible for deallocating memory that is no longer in use. This means that programmers do not have to manually deallocate memory which often causes memory leaks and other program malfunctions. But Java's garbage collector is also one of the downsides with Java. Even if it makes program easier to maintain and write, the garbage collector will cause a small overhead in execution time and may cause programs to run slower.[18]

3 Method

We began by identifying a couple of different algorithms that can be easily parallelized and that we believe would be able to showcase the language's characteristics. The exact algorithms we used and why we used them will be discussed in 3.1.

The actual tests to produce our results were done on a few different computers due to various requirements in the programming languages. All computers that we used to test our implementations had a multi-core CPU of 4 cores. We started by performing the test with all cores active on a fairly large problem size and made sure the programs fully utilized the given amount of cores and took note of the time it seemed to take on average. We then decreased the amount of active cores (all the way down to a single core) and timed the results there. Setting the amount of cores the implementation would use was done with the Linux command *taskset* or through the *Set affinity* option in the Windows Task Manager. We also used the System Monitor or Task Manager to verify that all the cores were being used. These results were then compared to each other as well as the Java implementation which did not make use of any parallelization techniques.

Some implementations were tested on a Windows computer and some on a Linux computer. It is possible that these differ somehow in how they handle parallel execution, so the absolute runtimes in the implementations may not be completely comparable. However, the relative times between different runs of the same implementations are still interesting as they show how well the performance scales with cores and the size of the problem instance.

All of the implemented algorithms perform some sort of calculation, for example finding a prime number or finding the product of two matrices. Since we are mostly interested in how well the program performs, the results of the calculation are not particularly relevant to us and are not printed to the console. This is because we know that the printing process will be sequential and it is not relevant to the performance of the algorithm itself.

To compare the scalability of the implemented algorithms, we will compare their *efficiency*. This is defined as

$$E_p = \frac{T_1}{pT_p},$$

where T_1 is the time required for the sequential algorithm and T_p is the time required for a parallel algorithm with p CPUs.[19] The efficiency will be a number between 0 and 1, which can be considered to measure how much of the processor time is actually used to solve the problem versus communication and synchronization between threads. An efficiency of 1 means the entirety of the processor time is used to solve the problem – a sequential algorithm therefore always has an efficiency of 1. As p increases, the efficiency typically drops for the parallel algorithm unless it has *linear speedup* (which means that an algorithm becomes twice as fast when the amount of CPUs are doubled).[19] Linear speedup is the ideal result as it means the algorithm will continue to scale and can be made as fast as possible simply by adding more cores.

3.1 Algorithms

3.1.1 Matrix multiplication

One of the implemented algorithms is a matrix multiplication algorithm. We chose a matrix multiplication algorithm because we can easily exploit the fact that different operations of the algorithm are completely independent of each other. This allows us to make these operations execute concurrently, since they do not rely on the other's results.

The product one gets by multiplying two matrices is another matrix. The cells in the resulting matrix is the dot product of the the corresponding row from the first factor (the left factor) and the corresponding column from the last factor (the right one). Note that matrix multiplication is not commutative.

For example: if we would like to multiply matrix A and B; the cell in the upper left corner would be the dot product of the first row of A and the first column of B. The cell to the right of the one in the upper left corner can be seen as the red circle in the figure.

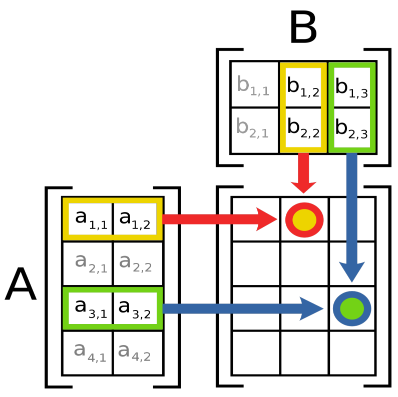

Figure 1: [20] The resulting cell is based on the corresponding row and column in the original matrices.

Each cell in the resulting matrix is completely independent of all the other cells. All the information you need to calculate the result of a cell is the corresponding row and column from the two factors. This is what makes matrix multiplication easy to parallelize.

3.1.2 Finding prime numbers

Prime numbers do not follow any kind of known sequence or system and it is relatively difficult to check whether a given number is prime or not. A well-known (though ineffective) algorithm for finding prime numbers is to successively check each number and see if it has any integer divisors (that is, integers $k > 1$ such that when the given number is divided by k it results in a remainder of 0 – in other words, the number can be written as k multiplied by some integer $\neq 1$). Obviously, if there are no divisors with zero remainder for this number, it is a prime. As one may realize, when testing divisors for a number n one only needs to test divisors between 2 up to $\sqrt{n}$. If $n = ab$ where a and b are $\neq 1$ then either a or b is at most $\sqrt{n}$.

Thus, we have the algorithm given in figure 2.

```
//begin is the first number to test for primality
//number is the amount of numbers to test
primes(begin, number) :=
        end := begin+number
        while begin < end do
                prime := true
                for i := 2 ... int(sqrt(begin)) do
                        if begin % i = 0 then
                                prime := false
                if prime = true then
                        print "%i is a prime", begin
                begin++
```

Figure 2: An algorithm for a simple prime number finder.

Given that this algorithm does not make use of any previously calculated results, there is no dependency on the order in which one tests for primes within the given range. In other words, each number can be tested for primality completely separately and parallel from every other. This means that this simple algorithm is good for testing parallelization.

4 Results

4.1 Matrix multiplication

4.1.1 Erlang

The Erlang implementation of matrix multiplication did not seem to scale very well, see the results in figure 3. At the end, when adding additional cores, going from 2 to 4 CPU cores does not present much gain in the time taken. This is also obvious when we look at the relative efficiency, as shown in table 2. The Erlang implementation, when given 4 CPU cores, only makes use of roughly half of the available processing power to perform the calculations.

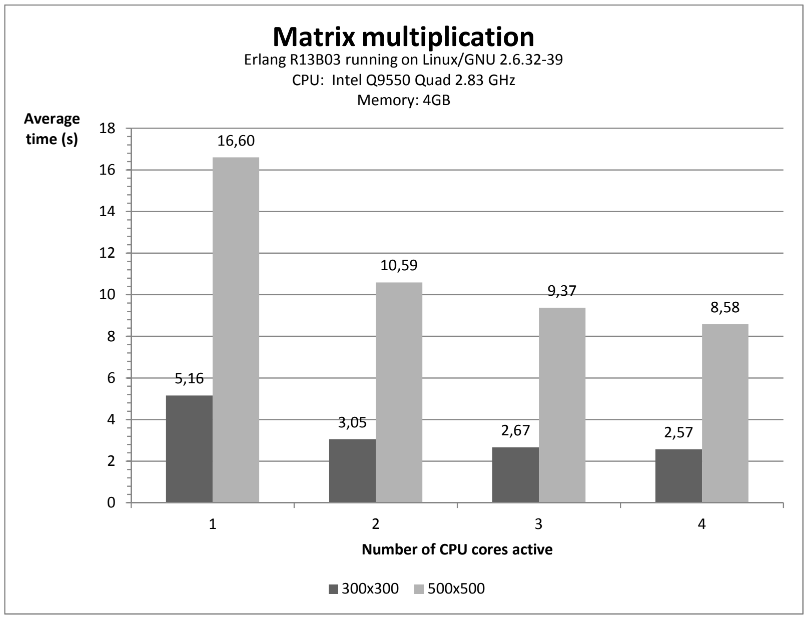

Figure 3: Results from the matrix multiplication implemented in Erlang, tested with different numbers of cores active.

Table 2: The efficiency of the Erlang implementation of matrix multiplication.

Size	E_2	E_3	E_4
300x300	0.85	0.65	0.50
500x500	0.78	0.59	0.48

4.1.2 F#

The F# implementation of matrix multiplication is, at first, much slower than the Erlang one. However, once 4 CPU cores are active, the F# implementation speeds up considerably (though it is still slightly slower than the Erlang one). In fact, our test results give an efficiency of over 1.0 for the F# implementation with 4 cores at 300x300. This is very surprising. Our hypothesis is that the sequential runtime (which is, in fact, the parallel implementation executed with only one core) is not running at full efficiency as the creation of threads and synchronization takes up processor time unnecessarily for one core. This makes the speedup appear more significant than it would be in practice.

The reason why the F# implementation performs relatively poorly before 4 CPU cores are active could possibly be that it has a lot of overhead which is especially apparent with only a single core.

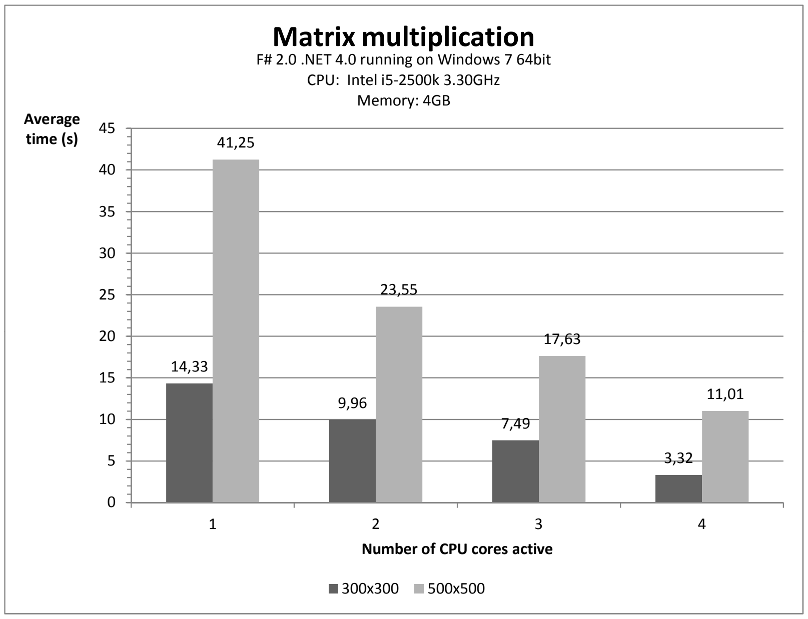

Figure 4: Results from the matrix multiplication implemented in F#.

Table 3: The efficiency of the F# implementation of matrix multiplication. See section 4.1.2 for discussion.

Size	E_2	E_3	E_4
300x300	0.72	0.64	1.08*
500x500	0.88	0.78	0.94

4.1.3 Java

The Java implementation of the matrix multiplication algorithm showed no signs of performance gain running on a multi-core CPU. The application only took advantage of one core. Our first tests showed that Java performed much faster than the other implementations, which implied that the JVM was using some form of optimization that allowed it to run in parallel. To make sure that the Java implementation was not making use of any parallel optimizations, we ran the program using the Java flag `-Xint` which does not compile to native code and instead executes all bytecode. Thus, none of JVM optimizations are done.

As seen in table 4, Java performed much worse than the other implementations and did not scale at all with additional cores.

Size	Average time (1 core)	Average time (2 cores)
300x300	9.528 ms	9.541 ms
500x500	44.166 ms	45.060 ms

Table 4: Results from the Java implementation of the matrix multiplication algorithm. These results were tested on Java JRE6 running on Windows 7 64bit. CPU: Intel Q6600 2.4GHz. Memory: 4 GB.

4.2 Prime number finder

4.2.1 F#

The F# implementation of the prime number finder scaled very well, as seen in figure 5. See also the efficiency values given in table 5. The efficiency is very close to 100% (linear speedup) throughout, and does not appear to drop considerably with an increasing amount of cores.

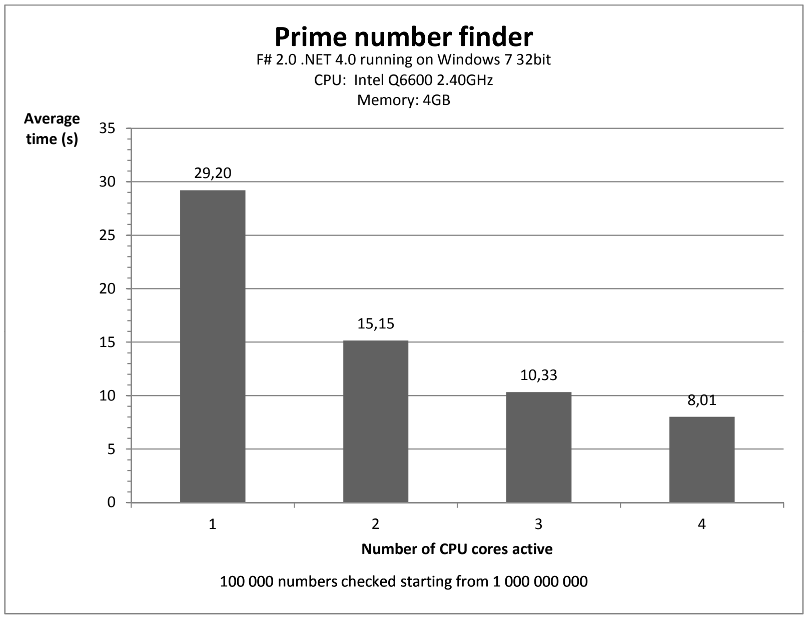

Figure 5: Results from the prime number finder implemented in F#.

Table 5: The efficiency of the F# implementation of the prime number finder.

E_2	E_3	E_4
0.96	0.94	0.91

4.2.2 Go

The Go implementation's results (see figure 6) are very similar to those of the F# implementation. Go did however take slightly longer in all cases, but it is only a difference of a few seconds at most. The scaling is exceptional, as illustrated by the efficiency shown in table 6. It is incredibly close to linear speedup throughout - doubling the amount of cores will almost halve the runtime.

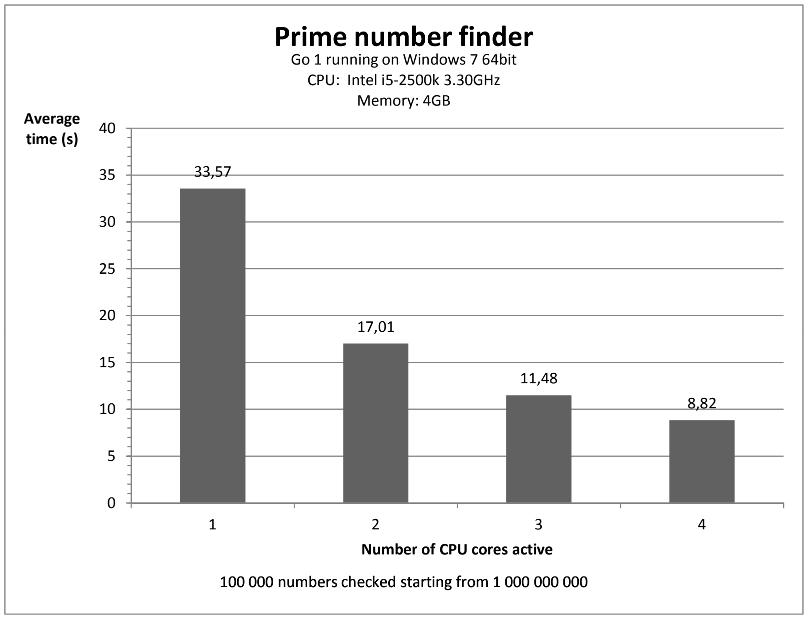

Figure 6: Results from the prime number finder implemented in Go.

Table 6: The efficiency of the Go implementation of the prime number finder.

E_2	E_3	E_4
0.99	0.98	0.95

4.2.3 Java

Once again, the Java implementation is not parallel, and is executed with the `-Xint` flag set to prevent optimizations that would give misleading results. As seen in table 7, the Java implementation was much slower than the other implementations.

Average time (1 core)	Average time (2 cores)
141.199 ms	143.861 ms

Table 7: Results from the Java implementation of the prime number finder. These results were tested on Java JRE6 running on Windows 7 64bit. CPU: Intel Q6600 2.4GHz. Memory: 4 GB.

5 Discussion

It is hard to compare programming languages with each other and decide which one is best. It can easily become a matter of opinion and personal taste rather than hard facts. To make our report as unbiased as possible we have tried to put most of our effort on comparing the languages in such ways that are easy to quantify such as performance measuring of the virtual machine and how the performance scales when increasing the amount of processors.

It is important, however, to point out that our results are highly dependent on how we implemented our algorithms (see appendix A). It also depends on how well external components such as the Java and Erlang virtual machines, operating systems, etc., perform, and does not solely rely on the programming language at hand. Another factor is that we were not able to run all of the tests on the same system, so they may not be directly comparable with each other, but they still illustrate the scaling.

5.1 Performance and scaling

We were not expecting the Erlang implementation to scale as poorly as it did. Erlang is known for its ability to scale, but with four cores active, the efficiency of the Erlang implementation was only about 50%, compared to Go and F#, which both had an efficiency above 90% with four cores. The reason for this could be our implementation of the chosen algorithm.

Java did not scale, but was slightly faster than F# with one core. We also saw that when Java was not running in interpreted mode, it was able to optimize so efficiently that it executed in only a few seconds (see table B.1.3). This is an interesting observation and shows that the JVM does a lot of optimizations that are otherwise mostly invisible to the programmer, and can outperform some languages (or rather, their virtual machine) when executed with one core. The reason for including Java is mainly to compare the parallel results with a single-core implementation, which is why we had to run it in interpreted-only mode so that it would not give misleading results. This gives a somewhat unfair picture of Java's performance.

Go and F# are both fairly new languages and may still benefit from further optimizations of their virtual machine, though they both scaled exceptionally well. Go, in particular, just released its first version (Go 1) on March 28th 2012 and is still young, as evidenced by the programmer needing to specify how many cores the program may use.

Erlang, on the other hand, is relatively old. Perhaps our measurements are a result of our implementations as mentioned before, or it could be that the Erlang designers did not prioritize performance as much as fault tolerance and parallel execution in the design of the Erlang virtual machine.

Most likely it is some combination of the above-mentioned factors that gave us our results, though it is also important to mention that, in practice, one may find very different results from these since situations typically are not as clear-cut as they were in our tests. There are other activities like reading from the hard drive, waiting for a network signal or some other form of IO that can affect the performance much more than the scaling of the algorithms involved.

We think it is important to put these results in the context of the established developing trend toward more and more CPU cores as well as the potential of cloud computing. Given this development, it is reasonable to assume that the languages that promote parallel programming and a multi-core approach will improve even further from a performance standpoint in the future. While Java certainly has support for parallel programming, it does not come at such ease as it does in, for example, Go, F#, and Erlang. The latter is also well-known for its fault tolerance and ability to hot-swap code – there is more to a language than the performance.

5.2 Maintainability and ease of implementation

When it comes to comparing maintainability there are certain factors that one must take into account. One important factor is how popular the language is; as mentioned earlier, it can be easier to find a skilled programmer in Java than in Erlang, for example. The community around a language can also make a big difference; the amount of libraries and support available can be of major importance.

Another factor worth mentioning is that most programmers have a background in C-influenced languages. This could make it more difficult for a typical programmer to transition into a way of programming other than imperative programming. A language with a syntax similar to C could have an advantage over a language influenced by, say, Prolog.

Regardless of the language or paradigm chosen, a small project is fairly easy to maintain. When the number of collaborators on a project increases, and the scope of the project is larger than a simple application, it is easy to get lost without a good structure on the code. A proper object-oriented implementation can help maintain the structure and makes the code more viable for change. This is part of why object-oriented languages have been so popular in recent years.[21]

Erlang is a language where we did not see much performance gain, but the way Erlang is designed makes it easy to make use of a network to balance load. The uptime is demonstrably very high and Erlang's support for hot-swapping and development for distributed networks is something that the other languages do not offer to the same degree. These features can be of great value in some fields.

Another factor is the portability of the code. F# requires the .NET Framework, which is only available under Windows. Go compiles into platform-specific executables but has support for Mac OS X, Linux and Windows, while Erlang runs in a virtual machine, much like Java. It can be valuable to have a codebase that can easily be compiled to another operating system, or even to compile it once and have it run anywhere like Java. If it is important to support many platforms, it can be easier to use a language that has full support for these.

As mentioned before, the current trend toward more CPU cores over faster clock speed means languages that have inherent support for parallelization is going to be more important in the future. This is another factor to the maintainability; if it is easy to develop software that scales well and makes efficient use of the CPU power available, it is likely going to stay relevant.

6 Conclusion

It is apparent that it is possible to achieve fairly good performance scaling with relatively simple code. We did not have any previous experience with these languages (Erlang, F# and Go) but we were able to make implementations of simple algorithms that clearly scaled well with an increasing amount of cores.

It was interesting to see how much more effective the Java optimizations made all of the Java implementations. This shows of how much importance the virtual machine and compiler can be.

We are also of the opinion that performance is not everything. The ease at which we were able to implement parallel algorithms should be commended, and there are a great many features in these languages that are worth an additional look, such as Erlang's fault tolerance, F#'s .NET platform integration and Go's goroutines.

In our problem statement, we stated that we wished to answer the question of "which is the best programming language?" There is no real answer to this question, and from our results and research, it is obvious that every language has something to offer. If you want a system to be highly fault tolerant and able to divide load over a network, maybe Erlang is the right language for the job. On the other hand, if you want to develop for a Windows environment and use the large library available in the .NET Framework coupled with a very powerful language with support for parallelization, look into F#. If it is important to have a high degree of synchronization between different threads and a good scalability, Go might be the language you want. Of course, it is hard to go wrong with Java – while it may not have the same ease of parallel programming, it has a very large standard library and community, and can still be very competitive in terms of performance.

Our conclusion is that the differences in languages are difficult to quantify. The performance and scaling is only one part of what makes the strengths of the language, and the practical differences between these three were relatively small. Instead, it seems to be the factors surrounding the language that will determine how suitable it is for a specific task.

7 References

[1] Zhu Y. How to: implement parallel matrix multiplication in F# [homepage on the Internet]. c2010 [cited 2012 Apr 11]. Available from: `http://msdn.microsoft.com/en-us/library/hh304369.aspx`

[2] Wikipedia. F# (programming language) [homepage on the Internet]. No date [updated 2012 Mar 31; cited 2012 Apr 11]. Available from: `http://en.wikipedia.org/wiki/F_Sharp_%28programming_language%29#Examples`

[3] Intel. Single Chip 4bit 4004 CPU datasheet [homepage on the Internet]. No date [cited 2012 Apr 4]. Available from: `http://www.intel.com/Assets/PDF/DataSheet/4004_datasheet.pdf`

[4] Levi A. Applied quantum mechanics. Cambridge: Cambridge University Press, 2003; p. 144.

[5] IBM. What's this multi-core computing really? [homepage on the Internet]. No date [cited 2012 Apr 4]. Available from: `http://www-03.ibm.com/systems/resources/pwrsysperf_WhatIsMulticoreP7.pdf`

[6] Ross P. Why CPU frequency stalled [homepage on the Internet]. c2008 [cited 2012 Apr 3]. Available from: `http://spectrum.ieee.org/computing/hardware/why-cpu-frequency-stalled`

[7] Henry S, Humpfrey M, Lewis J. Evaluation of the maintainability of object-oriented software [homepage on the Internet]. No date [cited 2012 Apr 7]. Available from: `http://eprints.cs.vt.edu/archive/00000214/01/TR-90-32.pdf`

[8] erlang.org [homepage on the Internet]. c2011 [cited 2012 Apr 10]. Available from: `http://www.erlang.org/about.html`

[9] History of Erlang [homepage on the Internet]. c2011 [cited 2012 Apr 10]. Available from: `http://www.erlang.org/course/history.html`

[10] erlang.org. Concurrent programming [homepage on the Internet]. c2012 [cited 2012 Apr 10]. Available from: `http://www.erlang.org/doc/getting_started/conc_prog.html`

[11] Aloi R. The growth of Erlang: the Stack Overflow case study [homepage on the Internet]. c2010 [updated 2010 Dec 11, cited 2012 Apr 12]. Available from: `http://aloiroberto.wordpress.com/2010/12/11/the-growth-of-erlang-the-stack-overflow-case-study/`

[12] Armstrong J. Making reliable distributed systems in the presence of software errors [homepage on the Internet]. c2003 [cited 2012 Apr 4]; p. 191. Available from: `http://www.sics.se/~joe/thesis/armstrong_thesis_2003.pdf`

[13] golang.org. FAQ - The Go Programming Language [homepage on the Internet]. No date [cited 2012 Apr 10]. Available from: `http://golang.org/doc/go_faq.html`

[14] golang.org. The Go Memory Model [homepage on the Internet]. c2012 [updated 2012 Mar 6; cited 2012 Apr 10]. Available from: `http://golang.org/ref/mem`

[15] MSDN. Visual F# [homepage on the Internet]. No date [cited 2012 Apr 10]. Available from: `http://msdn.microsoft.com/en-us/library/dd233154.aspx`

[16] Syme D. Introducing F# asynchronous workflows [homepage on the Internet]. c2007 [updated 2007 Oct 10; cited 2012 Apr 10]. Available from: `http://blogs.msdn.com/b/dsyme/archive/2007/10/11/introducing-f-asynchronous-workflows.aspx`

[17] The history of Java technology [homepage on the Internet]. No date [cited 2012 Apr 10]. Available from: `http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html`

[18] The Java language environment [homepage on the Internet]. c1997 [cited 2012 Apr 10]. Available from: `http://java.sun.com/docs/white/langenv/Intro.doc2.html`

[19] Hurley S. Factors that limit speedup [homepage on the Internet]. c1994 [cited 2012 Apr 10]. Available from: `http://www.cs.cf.ac.uk/Parallel/Year2/section7.html`

[20] Wikimedia Commons [image on the Internet]. 2010 October [cited 2012 Apr 105]. Available from: `http://en.wikipedia.org/wiki/File:Matrix_multiplication_diagram_2.svg`

[21] Canfora G, Cimitile A. Software maintenance. Benevento, University of Sannio; 2000 [cited 2012 Apr 11]. Faculty of Engineering at Benevento, University of Sannio, Italy. Available from: `ftp://cs.pitt.edu/chang/handbook/02.pdf`

A Code

A.1 Erlang matrix multiplication implementation

A.1.1 matrix.erl

```
-module(matrix).
-export([dot_product_worker/1, transpose/1, multiply/2,
        spawn_workers/2, scatter/6, scatter_columns/6,
        combine/3, generate_partial_matrix/4, matrix_sum/2,
        random_matrix/2,vector_sum/2, row_worker/1, dot_product/2]).

%% Multiply matrix A with B.
%% Constraint: COLS = ROWS
multiply([], _) -> [];
multiply(A, B) ->
        Bt = transpose(B),
        CELLS = length(A)*length(A),
        ACCUM = generate_partial_matrix(0, length(A), length(A), CELLS),
        %%io:format("Calling combine.~n"),
        COMBINER = spawn(matrix,combine,[length(A), length(A), ACCUM]),
        PIDS = spawn_workers(length(A), COMBINER),
        scatter(A, Bt, PIDS, length(A), length(A), CELLS).


%% This method scatters the dot product operation (row dotted column)
%% out to row_workers
%%
%% Function parameters are
%% MatrixA, MatrixB, Worker PIDs, Rows left to scatter, Number of columns, Number of cells
scatter([], _, [], _, _, _) -> io:format("Done spawning workers.~n");
scatter([MAH | MAT], MB, PIDS, ROWSTOGO, COLS, CELLS) ->
        ROWINDEX = trunc(math:sqrt(CELLS)) - (ROWSTOGO - 1),
        %%io:format("Generating worker for row ~w. ~w rows to go.~n", [ROWINDEX, ROWSTOGO]),
        PID = hd(PIDS),
        PID ! {MAH, MB, ROWINDEX},
        UNUSED_PIDS = lists:sublist(PIDS, 2, length(PIDS)),
        scatter(MAT, MB, UNUSED_PIDS, ROWSTOGO - 1, COLS, CELLS).

row_worker(SenderID) ->
        receive
                        {ROW, MB, ROWINDEX} ->
                                        RES = lists:map(fun(X) -> dot_product(ROW,X) end, MB),
                                        SenderID ! {RES, ROWINDEX}
        end.


%% This is a helper function for the scatter function.
%% It spawns a process for each column for a specific row.
scatter_columns(_, [], _, _, _, _) ->
```

```
        [];
scatter_columns(ROW, [COL|T], [PID|REST], ROWINDEX, COLSTOGO, CELLS) ->
        COLINDEX = trunc(math:sqrt(CELLS)) - (COLSTOGO - 1),
        PID ! {ROW, COL, ROWINDEX, COLINDEX},
        scatter_columns(ROW, T, REST, ROWINDEX, COLSTOGO - 1, CELLS).

%% Transpose matrix
transpose([[]|_]) -> [];
transpose(M) ->
          [lists:map(fun hd/1, M) | transpose(lists:map(fun tl/1, M))].

%% Spawn dot_product_worker processes.
spawn_workers(0, _) -> [];
spawn_workers(NUM, COMBINER) ->
        %%io:format("Number of workers left to spawn: ~w.~n", [NUM]),
        [spawn(matrix, row_worker, [COMBINER]) | spawn_workers(NUM-1, COMBINER)].

dot_product(VecA,VecB) ->
        %%io:format("dot_product received ~w and ~w~n", [VecA, VecB]),
        lists:sum(lists:zipwith(fun(X,Y) -> X*Y end, VecA, VecB)).

%% Workers run this function.
%% It waits for two vectors and two indexes. Then sends the result and
%% the corresponding indexes back.
dot_product_worker(SenderID) ->
        receive
                {VecA, VecB, Row, Col} ->
                        RESULT = lists:sum(lists:zipwith(fun(X,Y) -> X*Y end, VecA, VecB)),
                        %%io:format("Worker PID: ~w | VecA:~w VecB:~w --- VecR:"
                        %%                 "~w~nRow: ~w Col: ~w Ordered by: ~w~n","
                        %%                   [self(), VecA, VecB, RESULT, Row, Col, SenderID]),
                        SenderID ! {RESULT, Row, Col}
                        %% Work work!
                        %%dot_product_worker(SenderID)
        end.

%% Combines the partial results from the workers.
combine(_, 0, ACCUM) -> io:format("Done!"); %% ~n~w~n",[ACCUM]);
combine(ROWS, ROWS_TO_GO, ACCUM) ->
        receive
                {ROW, ROWINDEX} ->
                        %%io:format("Combine: Received ~w, index is ~w.~n", [ROW, ROWINDEX]),
                        %%io:format("Combine: Rows to go: ~w~n", [ROWS_TO_GO]),
                        combine(ROWS, ROWS_TO_GO-1, ACCUM)
        end.

generate_partial_matrix(VAL, Row, Col, CELLS) ->
        ROWS = trunc(math:sqrt(CELLS)),
        lists:map(fun(_) -> generate_zero_row(ROWS) end, lists:seq(1,Row-1))
        ++ [generate_filled_row(ROWS, VAL, Col)] ++
        lists:map(fun(_) -> generate_zero_row(ROWS) end, lists:seq(Row+1,ROWS)).

generate_zero_row(COLS) ->
        lists:map(fun(_)->0 end, lists:seq(1, COLS)).

generate_filled_row(COLS, VAL, Col) ->
        lists:map(fun(_)->0 end, lists:seq(1,Col-1))
        ++ [VAL] ++
        lists:map(fun(_)->0 end, lists:seq(Col+1,COLS)).

matrix_sum([],[]) -> [];
matrix_sum(MA, MB) ->
        %%io:format("HA: ~w TA:~w~n", [HA,TA]),
        %%io:format("HB: ~w TB:~w~n", [HB,TB]),
        lists:zipwith(fun(RA,RB) -> lists:zipwith(fun(X,Y)->X+Y end, RA,RB) end, MA, MB).

vector_sum([],[]) -> [];
vector_sum([HA|TA], [HB|TB]) ->
        [HA+HB | vector_sum(TA,TB)].

random_matrix(Size, MaxVal) ->
          random:seed(),
          lists:map(
                  fun(X) ->
                          lists:map(
                                  fun(Y) ->
                                          case Y of
                                                  X -> 0;
                                                  _ -> random:uniform(MaxVal)
                                                  end
                                          end,
                                          lists:seq(1,Size))
                                  end,
                                  lists:seq(1,Size)).
```

A.2 F# matrix multiplication implementation

The F# implementation of matrix multiplication was based off Yin Zhu's implementation.[1]

A.2.1 Matrix.fs

```
module Matrix


let randomMatrix =
    let rnd = new System.Random()
    Array2D.init 500 500 (fun _ _ -> rnd.NextDouble())

let matrixMultiply (a:float[,]) (b:float[,]) =
    let rowsA, colsA = Array2D.length1 a, Array2D.length2 a
    let rowsB, colsB = Array2D.length1 b, Array2D.length2 b
    let result = Array2D.create rowsA colsB 0.0
    [ for i in 0 .. rowsA - 1 ->
        async {
            for j in 0 .. colsB - 1 do
                for k in 0 .. colsA - 1 do
                    result.[i,j] <- result.[i,j] + a.[i,k] * b.[k,j]
        } ]
    |> Async.Parallel
    |> Async.RunSynchronously
    |> ignore
    result

open System


let t = System.Diagnostics.Stopwatch.StartNew()

ignore (Console.ReadKey false)
ignore (matrixMultiply randomMatrix randomMatrix)
ignore (printf "Elapsed time: %d ms" t.ElapsedMilliseconds)
ignore (Console.ReadKey false)
```

A.3 Java matrix multiplication implementation

A.3.1 Matrix.java

```
/**
 * This class reprents a matrix.
 * It has a two attributes, numRows and numCols. The cells in this
 * matrix is of the type integer. The dimension of this matrix is 0-indexed.
 * <bold>Note</bold> that the dimensions must be quadratic.
 */
import java.util.Random;

public class Matrix {

        private int numCols;
        private int numRows;
        private int[][] data;

        /**
         * Default constructor.
         * Creates a new instance of a matrix.
         * @param numRows - Number of rows this matrix has.
         * @param numCols - Number of columns this matrix has.
         * @return Matrix - The matrix created.
         */
        public Matrix(int numRows, int numCols) {
                        if (numRows != numCols) {
                                        System.err.println(">Error! Dimension must be "+
                                                          +"quadratic dimensions!");
                                        System.exit(1);
                        }

                        this.numCols = numCols;
                        this.numRows = numRows;

                        data = new int[numRows][numCols];
        }

        /**
         * Multiplies two matrices returning the result as a Matrix.
         * @param Matrix - Left operand in the matrix multiplication.
         * @param Matrix - Right operand in the matrix multiplication.
         * @return Matrix - The product of the two given matrices.
         */
        public static Matrix MatrixMultiplication(Matrix m1, Matrix m2) {
                Matrix res = null;

                if ( m1.getNumCols() != m2.getNumRows()) {
                                System.err.println(">Error! Dimensions dont " +
                                                + " match, can't multiply.");
                                System.exit(1);
                }

                res = new Matrix(m1.getNumRows(), m2.getNumCols());

                int val = 0;
                for(int i = 0; i < m1.getNumRows(); i++) { // aRow
                        for(int j = 0; j < m2.getNumCols(); j++) { // bColumn
                                for(int k = 0; k < m1.getNumCols(); k++) { // aColumn
                                        val += m1.getValue(i,k) * m2.getValue(k,j);
                                }
                                res.setValue(i,j,val);
                                val = 0;
                        }
                }

                return res;
        }

        /**
         * Sets the value of a specific cell in the matrix.
         * Note that the dimensions are 0-indexed.
         * @param row - The row of the value to set.
         * @param col - The column of the value to set.
         * @param value - Value to assign.
         */
        public void setValue(int row, int col, int value) {

                // Validate array boundaries.
                if (row >= 0 && row < numRows && col >= 0 && col < numCols) {
                                data[row][col] = value;
                                return;
                }

                System.err.println("Error! setValue got incorrect boundaries.\n" +
                                        "row = " + row + " this.numRows = " + this.numRows + "\n" +
                                        "col = " + col + " this.numCols = " + this.numCols);
                return;
        }

        /**
         * Returns the value of the matrix in position requested.
         * Note that the dimensions are 0-indexed.
         * @param row - The row of the requested value.
         * @param col - The column of the requested value.
         */
        public int getValue(int row, int col) {

                        // Validate array boundaries.
                        if (row >= 0 && row < numRows && col >= 0 && col < numCols) {
                                        return data[row][col];
                        }

                        System.err.println(">Error! getValue got incorrect boundaries.\n" +
                                        "row = " + row + " this.numRows = " + this.numRows + "\n" +
                                        "col = " + col + " this.numCols = " + this.numCols);
                        return -1;
        }

        /**
         * Returns a string representation of the matrix object.
         */
        public String toString() {
                StringBuilder sb = new StringBuilder();

                for(int row = 0; row < numRows; row++) {
                        for(int col = 0; col < numCols; col++) {
                                        sb.append(data[row][col] + " ");
                        }
                        sb.append("\n");
                }

                return sb.toString();
        }
```

```
        /**
         * Returns the number of rows this matrix has.
         * @return int - Number of rows in the matrix.
         */
        public int getNumRows() {
                        return numRows;
        }

        /**
         * Returns the number of columns this matrix has.
         * @return int - Number of column in the matrix.
         */
        public int getNumCols() {
                        return numCols;
        }

        /**
         * This method generates a filled matrix.
         * This can be handy when testing.
         * @param int - The number of rows in the generated matrix.
         * @param int - The number of columns in the generated matrix.
         * @param int - The maximum value of the cells in the generated matrix.
         */
        public static Matrix generateMatrix(int rows, int cols, int maxRand) {
                Matrix ret = new Matrix(rows, cols);
                Random rand = new Random(rows*cols*maxRand);

                for(int i = 0; i < rows; i++) {
                        for(int j = 0; j < cols; j++) {
                                        ret.setValue(i,j, rand.nextInt(maxRand));
                        }
                }

                return ret;
        }
}
```

A.3.2 Main.java

```
/**
 * This class holds the main method.
 */
import java.io.*;
public class Main {

        public static void main(String[] args) {
                try {
                        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
                        br.readLine();
                        } catch (Exception e) {
                }

                long start = System.currentTimeMillis();
                Matrix m1 = Matrix.generateMatrix(500, 500, 500);
                Matrix m2 = Matrix.generateMatrix(500, 500, 500);

                Matrix m3 = Matrix.MatrixMultiplication(m1,m2);
                long end = System.currentTimeMillis();
                System.out.println("Done! It took " + (end - start) + "ms");
                //System.out.println("m3: \n" + m1);
        }

}
```

A.4 F# primes implementation

The F# implementation of the prime number algorithm was based off the example given on Wikipedia.[2]

A.4.1 Program.fs

```
(* Async workflows sample (parallel CPU and I/O tasks) *)

module Program

(* A very naive prime number detector *)
let isPrime (n:int) =
   let bound = int (System.Math.Sqrt(float n))
   seq {2 .. bound} |> Seq.exists (fun x -> n % x = 0) |> not

(* We are using async workflows *)
let primeAsync n =
    async { return (n, isPrime n) }

(* Return primes between m and n using multiple threads *)
let primes m n =
    seq {m .. n}
        |> Seq.map primeAsync
        |> Async.Parallel
        |> Async.RunSynchronously
        |> Array.filter snd
        |> Array.map fst

open System

(* Run a test *)
ignore (Console.ReadKey false)
```

```
primes 1000000000 1000100000
    |> Array.iter (printfn "%d")
ignore (Console.ReadKey false)
```

A.5 Go primes implementation

A.5.1 primes.go

```
package main

import (
        "fmt"
        "runtime"
        "math"
        "time"
)

const NCPU = 1

func checkPrimes(begin int, number int, c chan int) {
        end := begin+number

        for ; begin < end; begin++ {
                prime := true
                for i := 2; i < int(math.Sqrt(float64(begin))); i++ {
                        if begin % i == 0 {
                                prime = false
                        }
                }
                if prime == true {
                        //fmt.Printf("%d  ", begin)
                }
        }
        c <- 1
}

func main() {
        runtime.GOMAXPROCS(NCPU)

        begin := 1000000000
        number := 100000

        c := make(chan int)

        t0 := time.Now()

        for i := 0; i < NCPU; i++ {
                go checkPrimes(begin + i*(number/NCPU), number/NCPU, c)
        }
        for i := 0; i < NCPU; i++ {
                <-c
        }

        t1 := time.Now()
        fmt.Printf("Duration: %v\n", t1.Sub(t0))

}
```

A.6 Java primes implementation

A.6.1 PrimeTest.java

```
import java.io.IOException;
import java.util.Date;


public class PrimeTest {

        /**
         * @param args
         * @throws IOException
         */
        public static void main(String[] args) throws IOException {
                int begin = 1000000000;
                int number = 100000;
                int end = begin+number;
                boolean prime = true;

                System.in.read();
                long t0 = (new Date()).getTime();

                while (begin < end) {
                        prime = true;
                        for (int i = 2; i <= (int)(Math.sqrt(begin)); i++) {
                                if (begin % i == 0) {
                                        prime = false;
                                }
                        }
                        //if (prime) System.out.println(begin + " is a prime");
                        begin++;
                }

                long t1 = (new Date()).getTime();
                System.out.println("Time elapsed: " + (t1 - t0) + " ms");
                System.in.read();

        }

}
```

B Tables

All units in tables are given in seconds unless stated otherwise.

B.1 Matrix multiplication

B.1.1 Erlang

Test results from Erlang R13B03 running on Linux/GNU 2.6.32-39. CPU: Intel Q9550 Quad 2.83GHz. Memory: 4GB.

Table 8: Test results from Erlang matrix multiplication.

Size	1 core	2 cores	3 cores	4 cores
300x300	5.155	3.049	2.752	2.713
	5.157	3.055	2.729	2.516
	5.159	3.047	2.692	2.529
	5.159	3.053	2.487	2.527
Average	5.158	3.051	2.665	2.571
500x500	16.596	10.630	9.722	8.878
	16.617	10.643	9.662	8.047
	16.592	10.407	8.327	8.506
	16.592	10.687	9.785	8.897
Average	16.599	10.592	9.374	8.582

B.1.2 F#

Test results from F# 2.0 .NET 4.0 running on Windows 7 64bit. CPU: Intel i5-2500k 3.30GHz. Memory: 4GB.

Table 9: Test results from F# matrix multiplication.

Size	1 core	2 cores	3 cores	4 cores
300x300	14.608	10.065	7.217	3.094
	13.645	10.029	7.375	3.016
	13.851	10.359	7.902	3.343
	15.211	9.394	7.457	3.843
Average	14.329	9.962	7.488	3.324
500x500	41.381	25.623	18.383	11.115
	41.445	23.261	16.747	10.657
	40.987	22.571	18.454	11.399
	41.194	22.741	16.931	10.874
Average	41.252	23.549	17.629	11.011

B.1.3 Java

Test results from Java JRE6 running on Windows 7 64bit. CPU: Intel Q6600 2.4GHz. Memory: 4GB.

Table 10: Test results from Java matrix multiplication.

Size	1 core	2 cores
300x300	0.461	0.481
	0.458	0.471
	0.465	0.493
	0.461	0.481
Average	0.461	0.481
500x500	2.184	2.210
	2.177	2.183
	2.161	2.178
	2.210	2.211
Average	2.183	2.195

B.1.4 Java with -Xint flag

Test results from Java JRE6 running on Windows 7 64bit. CPU: Intel Q6600 2.4GHz. Memory: 4GB.

Table 11: Test results from Java matrix multiplication.

Size	1 core	2 cores
300x300	9.489	9.815
	9.583	9.550
	9.651	9.473
	9.389	9.329
Average	9.528	9.541
500x500	43.663	46.962
	44.987	44.132
	44.994	44.832
	43.021	44.312
Average	44.166	45.060

B.2 Prime number finder

B.2.1 F#

Test results from F# 2.0 .NET 4.0 running on Windows 7 32bit. CPU: Intel Q6600 2.40GHz. Memory: 4GB. 100000 numbers tested, starting from

1000000000.

Table 12: Test results from F# prime number finder.

	1 core	2 cores	3 cores	4 cores
	29.256	15.092	10.404	8.024
	29.169	15.234	10.107	8.106
	29.195	15.138	10.414	7.968
	29.192	15.154	10.395	7.958
Average	29.203	15.155	10.330	8.014

B.2.2 Go

Test results from Go 1 running on Windows 7 64bit. CPU: Intel i5-2500k 3.30GHz. Memory: 4GB. 100000 numbers tested, starting from 1000000000.

Table 13: Test results from Go prime number finder.

	1 core	2 cores	3 cores	4 cores
	33.565	17.017	11.481	8.824
	33.583	17.003	11.473	8.817
	33.560	17.005	11.480	8.830
	33.567	17.029	11.480	8.823
Average	33.569	17.014	11.479	8.824

B.2.3 Java

Test results from Java JRE6 running on Windows 7 64bit. CPU: Intel Q6600 2.4GHz. Memory: 4GB. 100000 numbers tested, starting from 1000000000.

Table 14: Test results from Java prime number finder.

	1 core	2 cores
	2.695	2.710
	2.660	2.642
	2.665	2.632
	2.665	2.524
Average	2.671	2.627

B.2.4 Java with -Xint flag

Test results from Java JRE6 running on Windows 7 64bit. CPU: Intel Q6600 2.4GHz. Memory: 4GB. 100000 numbers tested, starting from 1000000000.

Table 15: Test results from Java prime number finder.

	1 core	2 cores
	141.593	150.659
	141.711	142.022
	141.012	141.444
	140.480	141.321
Average	141.199	143.861