Thesis no:
URI: urn:nbn:se:bth-16548

Parallelism in Go and Java

A Comparison of Performance Using Matrix Multiplication

Tobias Andersson and Christoffer Brenden

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the bachelor degree in Software Engineering. The thesis is equivalent to 10 weeks of full time studies.

Contact Information:
Authors:
Tobias Andersson
Email: tobias_andersson1@hotmail.com

Christoffer Brenden
Email: christofferbrenden@hotmail.com

University advisor:
Dr. Fabian Fagerholm
Department of Software Engineering

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

ABSTRACT

This thesis makes a comparison between the performance of Go and Java using parallelized implementations of the Classic Matrix Multiplication Algorithm (CMMA). The comparison attempts to only use features for parallelization, goroutines for Go and threads for Java, while keeping other parts of the code as generic and comparable as possible to accurately measure the performance during parallelization.

In this report we ask the question of how programming languages compare in terms of multithreaded performance? In high-performance systems such as those designed for mathematical calculations or servers meant to handle requests from millions of users, multithreading and by extension performance are vital. We would like to find out if and how much of a difference the choice of programming language could benefit these systems in terms of parallelism and multithreading.

Another motivation is to analyze techniques and programming languages that have emerged that hide the complexity of handling multithreading and concurrency from the user, letting the user specify keywords or commands from which the language takes over and creates and manages the thread scheduling on its own. The Go language is one such example. Is this new technology an improvement over developers coding threads themselves or is the technology not quite there yet?

To these ends experiments were done with multithreaded matrix multiplication and was implemented using goroutines for Go and threads for Java and was performed with sets of 4096x4096 matrices. Background programs were limited and each set of calculations was then run multiple times to get average values for each calculation which were then finally compared to one another.

Results from the study showed that Go had ~32-35% better performance than Java between 1 and 4 threads, with the difference diminishing to ~2-5% at 8 to 16 threads. The difference however was believed to be mostly unrelated to parallelization as both languages maintained near identical performance scaling as the number of threads increased until the scaling flatlined for both languages at 8 threads and up. Java did continue to gain a slight increase going from 4 to 8 threads, but this was believed to be due to inefficient resource utilization on Java's part or due to Java having better utilization of hyper-threading than Go.

In conclusion, Go was found to be considerably faster than Java when going from the main thread and up to 4 threads. At 8 threads and onward Java and Go performed roughly equal. For performance difference between the number of threads in the languages themselves no noticeable performance increase or decrease was found when creating 1 thread versus running the matrix multiplication directly on the main thread for either of the two languages.

Coding multithreading in Go was found to be easier than in Java while providing greater to equal performance. Go just requires the 'go' keyword while Java requires thread creation and management. This would put Go in favor for those trying to avoid the complexity of multithreading while also seeking its benefits.

Keywords: Goroutines, Threads, Comparison, Matrices

CONTENTS

GLOSSARY

CMMA

Abbreviation for ‘Classic Matrix Multiplication Algorithm’. The algorithm is used because of its simple and similar implementation between programming languages.

Concurrency

Techniques that allow the execution of multiple tasks out-of-order or in partial order, as opposed to running them sequentially where each execution must finish before the next can start.

Parallelism

Techniques that allow the simultaneous execution of multiple tasks using multiple processing units.

Thread

A thread is a structure of values within a process that can be scheduled for execution. Several threads can exist within the same process where they share the process’ address space and resources, allowing the threads to also execute concurrently or in parallel.

Multithreading

Technique which allows a single program to be executed over multiple threads.

Hyperthreading

Technique available on Intel processors that allows 2 threads to run on, or share, a single processor core and its resources, theoretically increasing performance.

1. INTRODUCTION

Parallelism and multithreading has become an important area in software development as newer processors are made with an increasing number of processing-cores, cores that can also manage a greater number of threads than their predecessors. At the same time new programming languages are being developed or old ones are being improved with features to allow software to better utilize threading and parallelism. As such it would be a good idea to compare a number of these languages to ascertain how well they handle multithreaded processes.

Finding the languages that have the best performance when running multithreaded processes is important as it could help software developers when choosing the most beneficial languages for developing their systems. It is especially important in software development for high-performance systems such as those pursuing the next prime-number or doing advanced mathematical calculations where it is important to have good performance to reduce the time these calculations take to save both time and money. Time, as more advanced calculations can take years to complete and even minor improvements could see great returns. Money, as the reduced time would mean quicker access to the results and the resources could then be allocated elsewhere.

It is also an important subject for game programmers as a game with increased performance that utilizes the computer's hardware more efficiently becomes better optimized making the game more responsive and reducing display- and input-lag for its users.

1.1 Motivation

The demand for systems to better utilize parallelization has increased greatly in recent times due to hardware getting closer and closer to limitations with both physics and the materials being used, causing it to become increasingly difficult to get increased performance from each individual core in processors [1]. Newer processors are instead receiving an increased number of cores to meet the demand for increased speed and performance.
This has shifted the focus from getting the maximum amount of performance from a single core to also wanting to utilize multiple cores to run instructions in parallel for a single program with techniques such as multithreading. This demand is often seen in high-performance systems, often with a focus on mathematical calculations, where the purpose is to use parallelism to reduce the time that each calculation takes to save both time and money.

The shift to multithreaded programs does come with some caveats however [2] [3] as it opens the potential for bugs and problems not normally seen in sequential programs. Race hazards can appear as multiple threads try to access the same data at the same time, threads may need to be synchronized to ensure that calculations are done in the right order and ensuring that processors do not sit idle as threads wait for work are just some of the issues that need to be considered. Coding multithreaded programs can as such require specialized skill sets of the programmer and debugging these programs can be difficult [2], expensive and time consuming because of the greater number of ways that multiple threads can interact with each other over sequential programs. These factors present multiple hurdles to overcome when creating multithreaded programs and have created an increased demand for ways to simplify or automate these parts of development. In response there has been some interest [4] in compilers and programming languages that attempt to hide the complexity of multithreading from the user, letting the language do it automatically or by having the user specify simple commands or keywords that trigger multithreading and then take over from there.

A question then would be how efficient these techniques are compared to a programmer coding all the steps themselves. Should the task of coding multithreading be left to compilers or is the technology not quite there yet? We would hope to gain some insight into this by comparing cases in Java, where the programmer can choose whether they want to have control over the threads or letting high-level interfaces in the language handle the threads, and Go, that has chosen to hide multithreading away behind automation.

Finally, matrix multiplication is an important as well as common form of calculation in linear algebra, a field that is often used within both graphical, gaming and scientific calculations, it is important that these calculations are done as fast as possible to save time and money by reducing the time needed to complete each calculation. At the same time matrix multiplication has a fairly simple and similar implementation in most programming languages making it a good mathematical operation to test performance between different programming languages, as you then lower the risk of potential performance differences being caused by the implementation of the program rather than the performance of the languages themselves.

1.2 Scope

For the scope of this paper the choice was made to limit the analysis to the programming languages Go and Java. The method itself will be limited to performance comparisons between implementations of matrix multiplication algorithms in each language. However, the method or implementation should be general enough to be easily convertible for similar experiments comparing alternative languages.

Go was chosen as it is a relatively new language, created in 2009, that has both seen an increased popularity in recent years and has also made the list of being one of the most loved and wanted programming languages according to the annual developer survey conducted by Stack Overflow [5]. Go also has a big focus on concurrency and parallelism through the implementation of its own Goroutines instead of regular thread usage which makes the language of increased interest for the subject of this paper.

Java was chosen because it is an older language, created in 1995, that has seen extreme popularity and is used for all manner of systems with a spread over nearly all platforms. The hope is that Java would serve as a good benchmark for older languages as opposed to newer languages developed during the increasing focus on parallelism.

This paper also aims to investigate if this is an area worth further research in comparisons between programming languages or if resources could be focused on other areas that would provide greater benefits.

2. RESEARCH QUESTIONS

2.1 Goals

The goal of this study is to find out which of the chosen languages, Go and Java, performs best at multithreading and how big of a difference in performance there is between the chosen languages. Also, to find out whether one language is best at both multithreading and single threading or if one language is better at multithreading whereas the other is better at single threading.

2.2 Research Questions

2.2.1 RQ1 - Does the choice of programming language, between Go and Java, have any noticeable difference in performance in multithreading/parallelization?

Hypothesis: The choice of programming language will see an effect in the performance. If not due to increased performance in executing the code within the threads then with improved performance of the language's way of creating and handling threads. The difference itself will probably not be large due to the timescales that are operated on and the limited areas of improvement available therein. However, due to the timescales even a small difference could see great returns in the end. An estimate would be a reduction in time in favor of Go.

2.2.2 RQ2 - How does the implementation of parallelization differ between Go and Java?

There are several differences in the implementation of threads between Go and Java which this paper attempts to delve deeper into and provide a brief explanation on.

2.2.3 RQ3 - Does the same language perform better than the other for both multithreaded and single threaded programs, or is there a difference in which language performs best for multithreaded programs and which language performs best for single threaded programs?

Hypothesis: Go will probably perform better in both areas in part due to Go not relying on a JVM and Go's design focus on Goroutines improving thread efficiency. It would, however, be interesting if a different language performs better in each scenario.

2.2.4 RQ4 - Is there any difference in performance for the chosen languages between running matrix multiplication without any threads and creating a new thread to run the matrix multiplication in?

Hypothesis: Due to Java programs running on a virtual machine, it does not have direct access to the operating system's thread functionality, but instead must call native methods which in turn spawns threads in the operating systems. This increases the overhead of creating a new thread in Java, which could make a main-thread program noticeably faster than a program which only spawns one thread.

Go spawns threads and runs the main function in a Goroutines by default however which makes the outcome a bit harder to predict. A preferable scenario would be that the difference would be minimal between the no-thread and new-thread test as the scheduler would pick up on the situation and run it as normal.

2.3 Expected Outcome

The expected outcome of this study is to find a difference in performance between each programming language in both sequential and multithreaded programs, and whether the difference is big enough to justify choosing one language over the other for software development. This study will therefore provide beneficial information to software developers as it may help them decide which programming language to use for high performance systems.

3. RESEARCH METHOD

RQ1, RQ4 and RQ4 will be answered through performance experimentation where data will be collected and compared to each other to answer the questions.

RQ2 will be answered through studying documentation and literature concerning the implementation of parallelization in Java and Go.

3.1 Literature research

The method of searching for references consists of searching for past thesis reports and scientific articles, publications and literature in Google Scholar, DIVA and Google which include any form or combination of the words “Multithreaded”, “Parallelization”, “Performance measuring”. Also “Java” and “Go/Golang” for references connected to our chosen programming languages, and “Matrix multiplication” for references connected to the implementation of our experiment.

In cases where the references consist of a gathering of information such as literature combining several subjects or similar reports only those chapters or sections deemed relevant to, or can strengthen the reasoning within the topics of this report need be considered.

For references used in describing the general concepts of the report, such as descriptions of each of the languages used, the number of total references need not exceed 2 or 3 as these are not the main subjects of the report and the need to strengthen their validity is deemed to be of lesser relevance.

Finally, results which reference events or experiments should preferably be firsthand or source material, although secondhand material with well detailed and proper methods and references will be considered in case of a lack of firsthand results.

3.2 Experiment

To test the difference in performance between Go and Java, we implemented matrix multiplication in both languages. The reason we chose to implement matrix multiplication is because this algorithm is fairly simple to implement in each language and is an important and common mathematical operation in linear algebra. Note that much of this chapter references matrix multiplication, which if you are unfamiliar with will be explained more in detail in Chapter 4.

For the experiment a modern laptop and operating system was chosen (see Table 1) and Go and Java are run in up-to-date versions (see Table 2).

Computer model	MacBook Pro 2015
CPU	Quad-core, 8-thread, Intel i7-4870HQ
Operating System	MacOS High Sierra 10.13.4
RAM	16GB

Table 1: Hardware used in the study.

Go	v1.9.4
Java	v9.0.1
JVM	Java HotSpot 64-Bit v9.0.1

Table 2: Software versions used in the study.

The programs will be run using the command 'nice' with the flag '-n -20' which gives the process the highest possible CPU-priority which reduces the chance of other processes interfering with the experiment. The Go program will be run with the garbage collector turned off, however Java does not allow for the garbage collector to be turned off, so the Java program will instead be run with the flags '-Xms8g -Xmx8g' which gives the JVM initial access to 8 GB of RAM which greatly reduces chance of the garbage collector from running as it only runs when the JVM runs out of RAM [6].

The data used in the experiment is a 4096x4096 matrix which is pre-generated with random values of between 0 and 1000. The same matrix is used in both programs to remove the possibility that the data which is used could be favorable for one of the languages which could cause faulty differences in performance between the languages. At the start of both programs the 4096x4096 matrix will be read from file before the timekeeping starts as this study will not be testing the performance of reading from file. The timekeeping will start after both languages have finished splitting up the matrix for each process and before each thread is created and started, and then the timekeeping will stop once each thread has finished their matrix multiplication and returned to the main thread.

Before each performance test run, there will be a test run, which will not be recorded, that will be used as a cold start-up to remove the possibility of the cores and the RAM being cold and clearing the RAM and cache memory from data which could affect the performance results.

The tests will first be run in the main thread without any multithreading to get a base value without parallelization and then the tests will be run with multithreading, at different numbers of threads. The amount of threads will increase exponentially, starting with 1 and 2 threads to get base values, then with 4 threads to test each of the cores, 8 threads to test each of the threads available to the hardware through Intel's Hyper-Threading Technology, and lastly with 16 threads to see if there is any benefit gained by exceeding the number of threads available. Another reason for the increases by factors of 2 is to maintain efficient splitting of the matrices for the CPU, reducing the chance any of them will be idle. This process will be run 7 times for each language and then the largest and smallest value will be removed and the average value for each test result will be calculated to minimize the risk of unexpected factors that could interfere with the results.

At first the tests were run using the Go program and a Java program which used the 'Thread' class for multithreading. However, we later found a second way of multithreading for Java which involved using an 'ExecutorService', a high-level feature included in Java version 5.0+ of creating and handling threads [7], which would be interesting to find out whether performance would be affected. Therefore, the tests were rerun using the Go program and both the Java program, which used the 'Thread' class, and another Java program which used the 'ExecutorService'.

The implementation of the multithreaded matrix multiplication used in this study is in the form of the following pseudocode:

```
    matrices = splitMatrixes(matrixA, nrOfThreads);
    FOR threadID = 0 to nrOfThreads
        createThread(threadID, matrices[threadID]);
    ENDFOR
    startTimeKeeping();
    FOR threadID = 0 to nrOfThreads
        /* start threads and run matrix multiplication in each
thread */
        startThread(threadID);
    ENDFOR
    waitForThreads(nrOfThreads);
    endTimeKeeping();
```

Code 1: Pseudocode of how matrix multiplication is split up between the threads and how the threads are created and run.

```
    matrixMultiplication(matrixA, matrixB, matrixC) {
        FOR i = 0 to matrixA.size()
            FOR k = 0 to matrixB.size()
                FOR j = 0 to matrixB.size()
                    matrixC[i][j] += matrixA[i][k] * matrixB[k][j];
                ENDFOR
            ENDFOR
        ENDFOR
    }
```

Code 2: Pseudocode of the matrix multiplication algorithm used in this study.

The implementation of this pseudocode for each of the two languages is included in the annexes of this thesis.

4. LITERATURE REVIEW

Here we provide a summary of the information gathered from the literature study, the sources consist mostly of academic papers and theses to summarize results on previous research on the subject, as well as official Go and Java documentation directly from their respective website to find specific information about the two programming languages.

Studies comparing Java and Go, especially in terms of multithreading, were not copious. Instead, studies that compared Java and Go in general and in cases included more than just Go and Java or in one case used similar languages, such as Scala in place of Java, were used.

Up-to-date documentation about Go was quite difficult to come by, this might be since Go is a new programming language which is continuously updated, thus much of the documentation on Go found was often outdated or in the form of a developer's blog posts with few references other than the open source code repository for Go itself. While documentation for Java on the other hand was more plentiful and easier to come by, as Java is an older language than Go and has been well documented from the very beginning.

4.1 A focus on multicore processors

For decades the computer industry could rely on Moore's law to provide increased performance for computers with single-core processors. The number of transistors able to fit on chips could keep up with predictions and increased exponentially roughly every 2 years and performance along with them.
More recently however developments started slowing down as the difficulty and costs being encountered in developing the next generation of processors rapidly increased [1] [8]. As the size of transistors grew smaller, problems appeared such as running up against the physical limitations of the materials being used, increased heat buildup and increased power consumption and leakage and dissipation which limited the clock frequencies of single-core processors from exceeding 4GHz [8]. These developments forced the industry to start looking elsewhere if they wanted to meet the ever-increasing demands on chip performance.

One way of increasing performance that manufacturers turned to was to use processors that combined multiple processing units, or cores, inside a single chip, known as a multi-core processor. Rather than increase the speed that the work could be completed in sequence the idea was to instead split the work into multiple parts and assign each of them between the cores and calculate them separately but in parallel.
This would create the increase in performance as even though each individual task was completed in the same amount of time it was now possible to complete two, four or even more tasks in the same timeframe as just the one.

Simply changing processors to use multiple cores was not enough to provide the additional benefit however. The tasks were not divided or scheduled between the cores automatically by the hardware, instead it was left up to the software developers to design their software in such a way as to effectively divide the work between the cores [8]. This came with greatly increased complexity when trying to prevent cores from running idle while waiting for work and wasting potential processing power or managing the shared memory of the processor so that multiple cores were not trying to work on the same data at the same time.

At the time of this writing although the problem is being worked upon there are currently few ways of doing parallel programming automatically. The Go language attempts to alleviate the problem with its goroutines [9], hiding some of the complexity of thread creation and assignment between special commands and a built-in scheduler that handles the

dividing of the work between processors. Support for synchronization also exists but maintains some complexity.

4.2 Java

The Java project was started in 1991 by a team of engineers from the company Sun Microsystems called the 'Green Team' but the project was not launched until 1995 when the developer team announced that the Netscape Navigator browser would start using the Java technology [10]. Java was originally designed with five design goals in mind [11].

The first design goal was that Java was designed to be simple, object oriented and familiar [11]. This design goal was worked towards through simplifying complex environments while still being efficient through being an object-oriented language, and to include various tested libraries which include functionalities that can be used by programmers to simplify development of Java applications. Also, Java was designed to have the "look and feel" of C++ to create a familiarity between the two languages as this would make it easier for C++ programmers to migrate over to using Java.

The second design goal was that Java was designed to be robust and secure which was worked towards through including certain features in the language to increase the languages reliability and security [11]. These features are, for example, automated garbage collection, which prevents memory leaks in the software and simplifies memory management for the programmer, and by including built-in security features which prevents attacks such as buffer overflow attacks from happening.

The third design goal was that Java was built to be architecture neutral and portable through virtualization by using Java Virtual Machines, JVM [11]. These JVM run a certain type of bytecode which the Java Compiler generates which is architecture neutral and can run on any JVM on any platform. These virtual machines can be implemented according to the JVM specification for any new architecture if the architecture meets certain basic requirements, such as multithreading.

The fourth design goal was that Java was designed to have high performance, which was worked towards by having the automated garbage collector running on a low-priority background thread to reduce the possibility of both the main thread and the automated garbage collector accessing the same memory location at the same time [11].

The fifth design goal was that Java was designed to be interpreted, threaded and dynamic [11]. This was worked towards through supporting the use of multithreading and by providing a class, 'Thread', and an interface, 'Runnable', for developers to use to create multithreaded programs. Java also provides the use of condition locks and monitors for multithreading through the run-time system, and by implementing the functionalities in the Java libraries as 'thread safe' in order for the functionalities to be run safely without conflict between multiple threads.

There are two different low-level ways of creating threads in Java, each with their own benefit and disadvantage [12]. One way of creating a thread in Java is to create a class which extends the 'Thread' class, this gives the programmer increased control of the lifecycle of the thread, however since Java only allows single inheritance, this prevents the thread from being able to extend other base classes. The other way of creating a thread in Java is to create a class which implements the 'Runnable' interface, this allows for the class to be able to extend other base classes while still being run as a new thread, however this does not give the programmer control of the actual thread itself.

Java versions 5.0+ also provides high-level features for concurrency to simplify creating and handling threads [7]. One of these high-level features are ‘Executors’ which provide an API for creating and handling threads through using reusable threads in a thread pool to reduce the overhead of creating new threads each time a new task is to be executed in a separate thread.

4.3 Go

Go is an open source language that was made by R. Griesemer, R. Pike and K. Thompson at Google and version 1.0 was released in 2012 [13]. The language was in part created to handle criticisms of other languages at the time and their ability to meet the demands of the developing software industry [13], particularly in having to choose between a language either being easy to use or being either efficient to compile or to execute, rather than combining it all into one language.
Go tries to combine the simplicity of dynamically typed languages like python, fast compilation times, garbage collection and reduced complexity when developing for concurrency while also allowing development to be quick and efficient [13].

There are multiple implementation choices worth noting about Go in terms of parallelism and multithreading:

For concurrency and parallelism Go uses an implementation of threads called Goroutines [9] that are designed for ease-of-use and efficiency when implementing concurrency. It uses a built-in scheduler for handling threads behind the scenes and tries to hide much of the complexity of usual thread management from the user in favor of simply specifying what functions to run and prefacing them with the ‘Go’ command.
In other languages implementations of threads are often linked to operating system threads and have large, fixed stack sizes (up to 2 MB, OS dependent) [14]. This large memory size limits the number of threads able to be run concurrently. Switches between threads then also involve the hardware scheduler and take full context switches which are expensive and time consuming.

Go also uses operating system threads behind the scenes [15]. The way it does this is that normal operating system threads are automatically created or parked/unparked by the Go runtime each time a goroutine is about to be started, to ensure that each goroutine always has a ready thread that it can be scheduled to by the Go scheduler. An important distinction is that switches of the goroutines onto the OS threads are not handled by the operating system, instead goroutines are so called “Green” or “lightweight” threads [15], threads that are handled by user-level mechanisms, as opposed to Kernel or Native threads handled by the OS, and for Go this means that switches are instead handled by the Go scheduler.

As for the scheduler itself it is designed as a ‘work stealing’ scheduler [15] where each processor is given a queue of work items to compute. If a processor depletes its queue it can steal work items from the queues of other processors to continue working. This is to ensure that processors do not remain idle and that work is spread out around them.
Go’s Goroutines are designed to be lightweight with a small overhead and a stack size that is around 2KB on creation [14]. The stack can grow as needed and the initially small size makes it possible to have several hundred thousand Goroutines at the same time.

Also, similar to languages like Java, Go has a built in automatic garbage collector [16] used for managing and reclaiming unused memory. In software garbage collectors have a reputation of ‘Stopping the world’, where they pause execution of everything else while they do their garbage collection sweeps. This can of course be problematic for very time-sensitive

programs such as those doing precise measurements, and unlike in some other languages that allow you to choose between several different garbage collectors with varying strengths and weaknesses Go does not allow you to switch between different implementations, instead it defaults to using what is known as a 'concurrent mark and sweep' collector based on ideas by Dijkstra in 1978 [16].
While the garbage collector cannot be changed it can fortunately be disabled or set to be controlled by the developer to ensure it does not run at inopportune times.

Something interesting to note about Go is that most of the research [17] [18] [19] that was analyzed and mentioned working with Go or compared it to other languages remarked on Go being a pleasantly easy and efficient language to work with. This would suggest that the developers' goals of creating a language with a focus on being easy to use were potentially successful.

One of the papers also mentioned potential issues with Go's slices. Slices are references to sections of arrays in Go and are designed to be used in favor of and in much the same way as arrays to provide increased flexibility.
Cases were found [19] where slice of slice, a 2d array type version of a slice, compared to simple slice performance for Go was skewed in favor of simple slices, nearly halving performance by comparison in the earlier case.
This may be of importance during the course of this report as 2d arrays, and thereby slice of slices for Go, are a common representation of matrices which are used for the experiments.

4.4 Matrix multiplication and implementations

Matrix multiplication is the linear algebraic operation, $C = AB$, which multiplies two matrices A, of size $n \times m$, and B, of size $m \times p$, and produces a result matrix C of size $n \times p$ [20]. The mathematical formula of matrix multiplication is written as following:

$$C_{ik} = \sum_{j=1}^{m} A_{ij} B_{jk}$$

for all possible values of i and k where $i \in \{1,\ldots,n\}$ and $k \in \{1,\ldots,p\}$.

Matrix multiplication is an important yet simple mathematical operation in linear algebra and the classic matrix multiplication algorithm(CMMA) can easily be implemented in any programming language [21]. However, there are additional different and more complex implementations for matrix multiplication, created to optimize the matrix multiplication operation, where some of these implementations give a significant increase in performance in comparison with the CMMA.

The performance of CMMA can be written using the Big O notation where it runs in O(N^3) time, where N is the dimension of the matrix, while another matrix multiplication algorithm(MMA), written by Francois Le Gall, runs in O(N^2.3728639) [21]. This may not seem like a significant difference in time, however when converting the time complexities to the amount of floating point operations required on

A[N][N] • B[N][N] =C[N][N]

where

N = 32768

then the CMMA requires ~35 184 372 088 832 number of floating point operations, whereas the Francois Le Gall MMA requires ~51 826 053 965 number of floating point operations. This means that the Francois Le Gall MMA requires ~679 times less number of operations than the CMMA (as ~35 184 372 088 832 divided by ~51 826 053 965 equals ~679).

Unfortunately, Le Gall's MMA is only theoretical and has never been implemented due to its complexity which makes it impractical to both implement and use in real world systems. Also, since the goal of this study is to compare performance between programming languages and not to compare performance between algorithms, the choice of algorithm has no impact on the result if the chosen algorithm is implemented equally in both languages.

The implementation for CMMA is very simple as it only requires 4 lines of code in a C style programming language to implement the core structure of the algorithm [21]:

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

Code 3: C style implementation of the CMMA.

There are both advantages and disadvantages of using the CMMA, where the advantages are, for example, its simplicity in regards of implementation and portability in any programming language [21], as it only requires 4 lines of code to have the core structure of the algorithm, and being easy to optimize by using multithreading and parallelization. However, the disadvantages for the CMMA is that of its performance [21] as it has poor performance for matrices larger than 2048x2048 in comparison to other implementations of matrix multiplication.

4.5 Language comparisons and analysis

Below follows a short summary of papers that were found on language comparisons, either between Java and Go directly or between either language and similar languages.

Previous research has found differences between Go and Java both in compilation times and runtime performance. A comparison between Go 1.2 and Java 1.7.0_45 found that Go compiles 3 times faster than Java, has fewer lines of code and performs better than Java in parallel computation tests [18]. Java on the other hand performed better in the sequential case. These initial findings would make it seem that Go has an advantage in favor of Java for multithreading.

Go 1.2	**Java 1.7.0_45**
Compiles 3x faster	Better performance sequentially
Fewer lines of code	
Better performance in parallel	

Table 3: Comparison results between Go 1.2 and Java 1.7.0_45 [18].

Another, similar comparison was done of Go 1.4.2 and Scala 2.11.6 [19]. This is interesting because Scala is based on Java and shares the usage of a JVM. It concluded that Go had worse runtime performance than Scala [19]. However, the result was concluded to be because of the usage of and higher efficiency of Scala's futures. While Java also has a version of futures the purpose of this report was to keep the test cases between the languages as equal as possible. The inclusion of futures could disturb that goal and will as such not be used.

Go 1.4.2	Scala 2.11.6
	Better performance with the use of Futures

Table 4: Comparison results between Go 1.4.2 and Scala 2.11.6 [19].

A larger research project compared C++, Java, Go and Scala where the code size, compilation time, binary size, memory footprint and performance were compared between the four languages using a sequential loop recognition algorithm [22]. The conclusion was that Scala had the least amount lines of code, Go had the fastest compilation time, Java had the smallest binary file and C++ had the least amount of memory usage and had the best performance out of the four languages. Though looking only at Java and Go for comparison, it showed that Go had lesser amount lines of code, faster compilation time and less memory usage than Java, but Java had a smaller binary file and had better performance than Go.

C++	Java	Go	Scala
Least amount of memory usage	Smallest binary file	Fastest compilation time	Smallest code size
Best performance			

Table 5: Comparison results between C++, Java, Scala and Go [22]. Versions not specified in the report.

Go	Java
Smaller code size	Smaller binary file
Faster compilation time	Better performance
Less memory usage	

Table 6: Comparison results between Go and Java [22]. Versions not specified in the report.

While many of these findings were not directly applicable to the purpose of this report they do provide an idea of what to expect on how each of the languages would compare to one another while testing, and could also be used for further discussion should the results diverge.

5. RESULTS

This section contains a formatted summary of the data gathered from the tests along with short descriptions for each section. For more information on the experiment itself see section 3, Research Method.

Something to note, in the experimentation run there was no noticeable performance difference between the Java program which used the 'Thread' class and the Java program which used the 'ExecutorService', hence why both Java lines are almost identical.

These are the results gained from the experiments:

5.1 Experimentation results

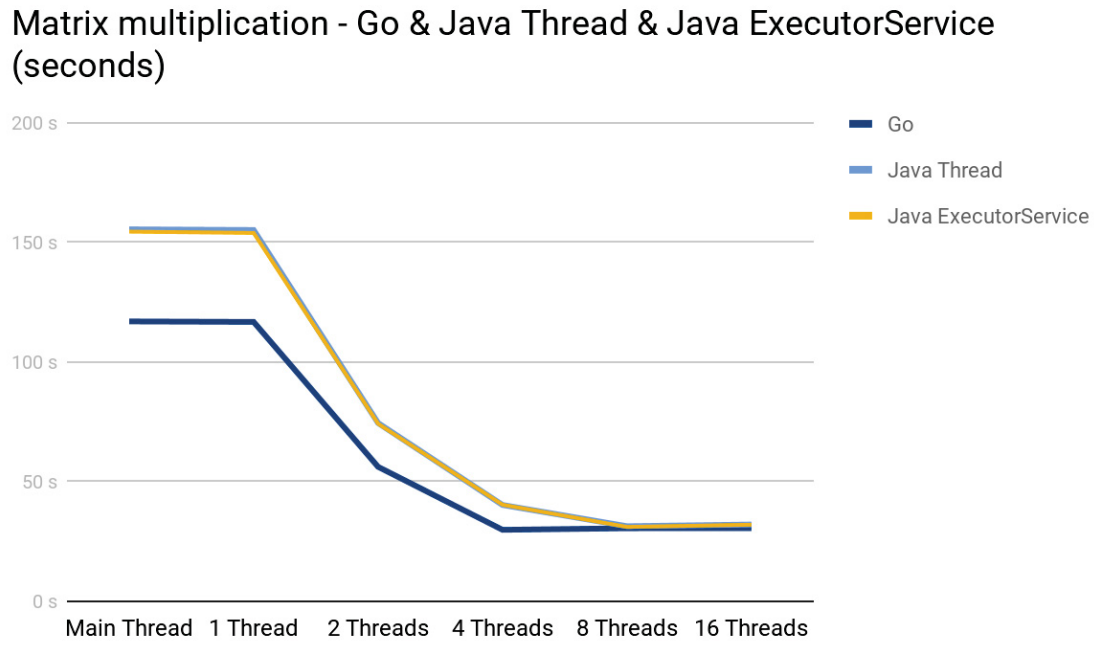

Figure 1: Performance comparison between Go and Java, with both Java Thread implementation and using Java ExecutorService, measuring time taken to multiply 4096x4096 matrices using different amount of threads.

	Go	**Java Thread**	**Java ExecutorService**
Sequential	117,01 s	155,51 s	154,54 s
1 Thread	116,80 s	155,22 s	154,00 s
2 Threads	56,23 s	74,54 s	74,24 s
4 Threads	29,91 s	40,29 s	40,39 s
8 Threads	30,55 s	31,42 s	31,12 s
16 Threads	30,60 s	32,12 s	32,05 s

Table 7: Results from performance comparison in Figure 1(in seconds).

The results show a significant advantage for Go in the cases of no parallelization, however this advantage gradually decreases when the number of threads increases to the point of there being barely any difference between performance of Go and Java for 8 and 16 threads.

Java starts off slower than Go even with no threads used despite the similarity of the calculations being made. A few potential reasons:

- The code for both languages is in fact not as equivalent as hoped.
- The code is equivalent but Java is somehow slower than Go, either at the specific calculations used for the tests or because of Java's usage of a JVM.

The difference between running the programs on the main thread and on 1 thread for both languages is so slight, around 1%, that the reason for the differences might not be linked to whether the program is run on the main thread or on 1 thread, but instead might be due to some performance deviation during the running of the experiments.

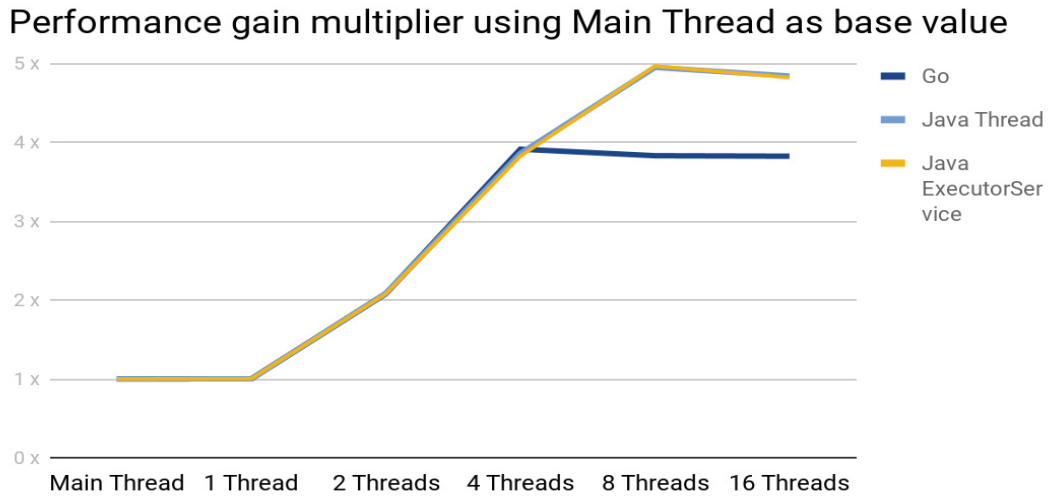

Figure 2: Performance comparison between Go and Java, with both Java Thread implementation and using Java ExecutorService, showing performance gain multiplier of multithreading using each language's Main Thread as base value.

When increasing the number of threads to 1, 2 and 4, the performance gain multiplier for each language is very similar to each other, however at 4 threads Go reaches its highest performance, as Go even has a reduction in performance at 8 and 16 threads, which may be connected to the Go scheduler and the way it handles the assigning of threads. However, Java continues to see an increase in performance up to 8 threads, with a slight reduction in performance at 16 threads.

The peak performance is at 4 threads for Go and at 8 threads for Java. At 8 threads the performance between both Java & Go is almost identical. Which means that if a high-performance system has access to 8 or more threads, then there is no reason for choosing one language over the other based on performance alone.

These findings coincide with previous research on older versions of the languages [18] where it was found that Go outperforms Java in parallel tasks. However, these findings also differ from previous research which show that Java outperforms Go in sequential tasks [18] [22].

6. DISCUSSION

The following section contains an analysis of the data gathered from the previous sections where we attempt to link the data to each of the research questions posed in section 2.

As the difference between the two Java programs are very minimalistic, the comparisons between Go and Java will only use the data gathered from the Java program which uses the Thread class.

6.1 Research Questions

6.1.1 RQ1 - Does the choice of programming language, between Go and Java, have any noticeable difference in performance in multithreading/parallelization?

Based on the experiment in this study, the choice of programming language has a noticeable difference in runtime performance for 1, 2 and 4 threads with Go being ~32-35% faster than Java. This then diminishes in the case of 8 and 16 threads to roughly a ~2-5% difference in performance but still in favor of Go. Additionally, both versions of Java tested had nearly identical results and the use of ExecutorService only showed a <1% improvement in performance.

Regrettably, as the goal of the study was to measure parallelism, due to the scaling of the performance increase as new threads are added being nearly identical between the languages the differences in performance observed during the tests is most likely not due to parallelism, but due to base differences between the languages themselves.

Another observation is that Java initially performs worse than Go with the difference remaining largely consistent going up to 4 threads, where Go reaches its peak. Java then sees an additional gain in performance at 8 to 16 threads. 8 threads would be when the number of threads exceeds the number of physical cores available in our testing equipment and should be scheduled to logical cores using Intel's hyperthreading.
Intel's Hyper-Threading Technology allows for each physical core to be seen as two logical cores instead, this means that the two logical cores will have their own architecture state but will share the physical execution units with each other [23]. This is done to maximize resource utilization by allowing two threads to run in parallel on the same core, but this is mainly meant for utilization of otherwise unused physical execution units on the physical core. Since both threads are performing the same forms of calculations there would be no free resources for the second thread to utilize and the performance increase would be minimal, as is seen with Go.

Why does Java continue to see improvement above 4 allocated threads yet Go does not?
Is Java or Go inefficient in this case?
This difference remained consistent when using a thread pool for Java so merely the existence of a scheduler or pool cannot be the cause. Is Go's scheduler more efficient?

If CPUs are already being used to 100% by a program an expected result of attempting to allocate more threads than there are CPUs would be a decrease in performance as the threads are inefficiently competing for resources. This might explain the worsening performance between Go at 4 goroutines and 8 to 16.

For Java the case might be the inverse, CPUs are not used at 100% or are stuck waiting. In this case allocating more threads than the number of CPUs might increase performance as

more threads are scheduled and are put to work when an existing thread is set to a waiting state.

Another possible explanation could be Java running in a JVM while Go does not. It would be interesting to see if there were ways to close the gap for Java through using different settings for the JVM.

	Performance increase of Go in comparison to Java Thread
Main Thread	32.9%
1 Thread	32.9%
2 Threads	32.6%
4 Threads	34.7%
8 Threads	2.8%
16 Threads	5.0%

Table 8: Performance increase of Go in comparison to Java using the Thread class.

	Performance difference for Go
Main thread to 1 thread	+0.2%
1 thread to 2 threads	+107.7%
2 threads to 4 threads	+88.0%
4 threads to 8 threads	-2.1%
8 threads to 16 threads	-0.2%

Table 9: Performance difference of experiment for Go as the number of threads increases.

	Performance difference for Java Thread
Main thread to 1 thread	+0.2%
1 thread to 2 threads	+108.2%
2 threads to 4 threads	+85.0%
4 threads to 8 threads	+28.2%
8 threads to 16 threads	-2.2%

Table 10: Performance difference of experiment for Java using the Thread class as the number of threads increases.

	Performance difference for Java ExecutorService
Main thread to 1 thread	+0.4%
1 thread to 2 threads	+107.4%
2 threads to 4 threads	+83.8%
4 threads to 8 threads	+29.8%
8 threads to 16 threads	-2.9%

Table 11: Performance difference of experiment for Java using the ExecutorService implementation as the number of threads increases.

6.1.2 RQ2 - How does the implementation of parallelization differ between Go and Java?

Based on information gathered during the literature study in section 4 of this report the implementations are as follows:

On program startup Go creates an operating system thread which it then sets the main goroutine, a so called green thread, to run on. Go then allows the user to create or assign functions to additional goroutines that are then scheduled automatically onto the OS threads by the Go scheduler. If the scheduler detects that there are no free threads to handoff a newly created goroutine to, or if a thread is about to block in some way it creates a new thread or unpark an existing thread for the handoff. The user is then allowed limited handling of the goroutines such as synchronizing their output with wait-groups or intercommunication with channels, but the handling and scheduling to the OS threads is almost entirely handled by the Go scheduler, hidden away from the user.

Java however does not create OS threads on program startup, except for the main thread, so to create additional threads for a Java program to achieve parallelization, the user has a few options on how to create the threads, both low-level and high-level solutions.
The low-level solutions are by either creating a class which extends the ‘Thread’ class or by creating a class which implements the ‘Runnable’ interface. The high-level solutions provided by the Java concurrency package, available by default in Java version 5.0+, are by using pre-implemented classes which implements the ‘Executors’ interface. One of these implementations is the ‘ExecutorService’ which allows the user to create reusable threads in a thread pool which the user then may supply tasks, in the form of implementations of the ‘Runnable’ or the ‘Callable’ interface, to be handled and executed by the ‘ExecutorService’. This thread pool will then in turn increase performance as the overhead of creating a new thread for each task will be removed since the tasks will reuse threads which are already created by the ‘ExecutorService’.

6.1.3 RQ3 - Does the same language perform better than the other for both multithreaded and single threaded programs, or is there a difference in which language performs best for multithreaded programs and which language performs best for single threaded programs?

Based on our experiment we found that Go performs best for both multithreaded and single threaded programs. This is consistent with previous research on the topic [18] that also found Go to perform better than Java for multithreading. However, this also differs from previous research that found Java to perform better than Go for sequential tasks [18] [22].

The reason for the differing results is difficult to ascertain as the first research report does not include examples of the code used for the calculations, making comparisons difficult, and the second does not specify the version used for Go and Java. With a publication date of 2011 however that would put the Go version to a version prior to Go 1, released in March 2012. This could make the differences attributable to improvements made over the different versions, but no clear conclusion on this area can be made from current data.

6.1.4 RQ4 - Is there any difference in performance for the chosen languages between running matrix multiplication without any threads and creating a new thread to run the matrix multiplication in?

There is only a very slight difference between running Go and Java on their main thread and with 1 threads respectively, less than a 1% difference. The difference is so slight it most likely comes down to rounding in the calculation times themselves, as Go rounds 3 digits earlier than Java, and not having enough test data to form a conclusive average.

On startup both Go and Java create and run in a thread by default. Swapping between threads has some overhead and context switching that would cause a slight slowdown compared to not switching. However, as our measurements only start after the calculating thread has been started this slowdown would not be noticeable. Therefore, in our case running on their main thread or with 1 thread would largely be the same.

6.2 Limitations and validity threats

The largest threat to the experiment's validity is the fact that we were not able to run our experiments in single-user mode as Java could not be run in single-user mode. This meant that we had to run the experiments in multi-user mode in which other processes may disturb our experiment and therefore give faulty results. However, to limit the impact that other processes may have had on our experiments and results, we took some precautions by increasing the amount of times that the tests were run, shutting down as many processes that could be shut down, as well as disabling WIFI and Bluetooth.

We also had to limit ourselves to using only one algorithm for testing due to time limitations, this may also be a validity threat as matrix multiplication might favor one language over the other even though the implementation is equal in both languages.

We also limited ourselves to only using one system which could be a threat as running these experiments on different operating systems or different hardware, such as an AMD processor, may produce different results.

7. CONCLUSION

We performed a comparison between Go and Java using parallelized versions of matrix multiplication using sets of 4096x4096 matrices, in Go using goroutines and in Java using threads. We then measured the differences in time to completion of the calculations and made an analysis of the results. The goal being to investigate which language had the best performance during parallel tasks and if languages that handle parallelism in an automatic fashion would be competitive with languages where the developer creates and manages threads.

Java was the developer controlled language and has the user create operating system threads before they can be used and starting them when needed. The scheduling of the threads however is handled by the operating system.

Go on the other hand, being the language with the automated parallelism, uses Goroutines, threads that are scheduled by the runtime, and uses an automatic scheduler that assigns the Goroutines to hidden operating system threads that are created by the runtime as needed. Go gives the developer very few ways of interacting with the operating system threads in any way.

Based on the results the choice of programming language has a noticeable difference in performance. For 1, 2 and 4 threads Go had ~32-35% better performance than Java. This number then diminished as the number of threads increased, and at 8-16 threads the difference diminished to roughly a ~2-5% difference in performance. However, these results were most likely due to base differences with the language and not parallelism, as the performance scaling with each added thread was nearly identical between both languages.

Tests were also run to see if running the program on the main thread or creating and switching to a new thread would have any performance difference, but the performance remained the same.

If the program is to be run sequentially or on 1, 2 or 4 threads then the difference could be considerable enough to motivate choosing Go over Java for performance dependent programs. However, when the amount of threads is increased to 8 or 16 the choice becomes less clear based on performance alone. Languages with more automatic ways of parallelization were also found to at least be competitive with developer controlled parallelism and as such could be a potential alternative.

Finally, from a usability standpoint using threads in Go was found to be easier than in Java. While there were some additional requirements such as assigning the goroutines to wait groups to synchronize them, simply writing ‘go’ before a function was much simpler than creating and assigning threads in Java. This could put Go ahead of Java for those seeking the benefits of multithreading by wanting to avoid the complexity.

8. FUTURE WORK

Due to time and resource limitations there were numerous areas which this study was unable to touch further upon, areas which could be interesting to delve into for future studies. For instance, whereas this study was only able to use one set of configurations Java has several settings available in its JVM that could be altered to provide different results. These could give Java an edge over Go and might be worth testing for. Also, future versions of each language like Java 11 are in development as of this writing and Go is rumored to have possible performance improvements available to it that could cause significant changes.

Finally, as this study was limited to comparison with matrix multiplication other areas of each language could be worth investigating to get a more complete picture of how they match up against one another.

9. REFERENCES

[1] D. Geer, "Chip makers turn to multicore processors". Computer Volume: 38, Issue: 5, pp 11-13, May 2005. IEEE

[2] J. Ousterhout, "Why threads are a bad idea (for most purposes)", 1996, 1996 USENIX conference

[3] S.P. Midkiff, D.A. Padua, Issues in the compile-time optimization of parallel programs, 1990
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.48.2337&rep=rep1&type=pdf
Visited: 8 June 2018

[4] A. Nanda, Automatic Parallelism, Oracle Magazine 2010
https://blogs.oracle.com/oraclemagazine/automating-parallelism
Visited: 8 June 2018

[5] Stack Overflow Annual Developer Survey
https://insights.stackoverflow.com/survey/
Visited: 7 June 2018

[6] Sun Java System Application Server 9.1 Performance Tuning Guide - Managing Memory and Garbage Collection
https://docs.oracle.com/cd/E19159-01/819-3681/6n5srlhqf/index.html
Visited: 7 June 2018

[7] Oracle Java Documentation - High Level Concurrency Objects.
https://docs.oracle.com/javase/tutorial/essential/concurrency/highlevel.html
Visited: 9 May 2018

[8] J Parkhurst, J Darringer, B Grundmann, From Single Core to Multi-Core: Preparing for a new exponential, 2006
http://home.deib.polimi.it/sami/architetture/multicore.pdf
Visited: 20 April 2018

[9] The Go Programming language FAQ - Goroutines
https://golang.org/doc/faq#goroutines
Visited: 20 April 2018

[10] Oracle - The History of Java Technology.
http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html
Visited: 13 April 2018

[11] Oracle - The Java Language Environment.
http://www.oracle.com/technetwork/java/intro-141325.html
Visited: 13 April 2018

[12] R. Buyya, "Object oriented programming with Java", Chapter 14, pp 367-368, 2009. Tata McGraw Hill Education Private Limited

[13] The Go Programming language FAQ - Origins
https://golang.org/doc/faq#Origins
Visited: 13 April 2018

[14] Alan A. A. Donovan; Brian W. Kernighan, "The Go Programming Language", ISBN:0134190440, Chapter 9.8.1, 2015. Addison-Wesley Professional

[15] Go scheduler source code, with comments
https://github.com/golang/go/blob/master/src/runtime/proc.go
Visited: 22 May 2018

[16] The Go Blog: Go GC: Prioritizing latency and simplicity,
https://blog.golang.org/go15gc
Visited: 13 April 2018

[17] P. Tang, Multi-Core Parallel Programming in Go, 2010
https://pdfs.semanticscholar.org/7679/bd13f5987da282b0662a0c41a4d6dd6f2165.pdf
Visited: 11 March 2018

[18] N. Togashi, V. Klyuev, Concurrency in Go and Java: Performance Analysis. *4th IEEE International Conference on Science and Technology*, **2014**
https://ieeexplore.ieee.org/document/6920368/
Visited: 2 March 2018

[19] C. Johnell, Parallel programming in Go and Scala: A performance comparison, 2015
http://www.diva-portal.org/smash/get/diva2:824741/FULLTEXT03.pdf
Visited: 13 April 2018

[20] E. Weisstein, Matrix Multiplication:
http://mathworld.wolfram.com/MatrixMultiplication.html
Visited: 27 April 2018

[21] S. Kostrov, Performance of Classic Matrix Multiplication Algorithm on Intel Xeon Phi Processor System, *Intel Developer zone,* **2017**
https://software.intel.com/en-us/articles/performance-of-classic-matrix-multiplication-algorithm-on-intel-xeon-phi-processor-system
Visited: 3 March 2018

[22] R. Hundt, Loop Recognition in C++/Java/Go/Scala, 2011
https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/37122.pdf
Visited: 27 April 2018

[23] D. T. Marr et al., Hyper-Threading Technology Architecture and Microarchitecture
https://www.cs.sfu.ca/%7Efedorova/Teaching/CMPT886/Spring2007/papers/hyper-threading.pdf
Visited: 7 June 2018

10. ANNEXES

10.1 MatrixMultiGo.go

```
package main

import (
    "bufio"
    "fmt"
    "log"
    "os"
    "runtime"
    "runtime/debug"
    "strconv"
    "strings"
    "sync"
    "time"
)

func multiply(A [][]int, B [][]int) [][]int {
    sizeA := len(A)
    sizeB := len(B)
    n := make([][]int, sizeA)
    for i := range n {
        n[i] = make([]int, sizeB)
    }
    for i := 0; i < sizeA; i++ {
        for k := 0; k < sizeB; k++ {
            temp := A[i][k]
            for j := 0; j < sizeB; j++ {
                n[i][j] += temp * B[k][j]
            }
        }
    }
    return n
}

func splitMatrix(nrOfThreads int, matrix [][]int) (matrixes
[][][]int) {
    splitter := len(matrix) / nrOfThreads
    for i := 0; i < nrOfThreads; i++ {
        matrixes = append(matrixes,
matrix[splitter*i:(splitter*(i+1))])
    }
    return
}

func multiplyStuff(finalMatrix *[][][]int, matrix1 [][]int,
matrix2 [][]int, i int) {
    (*finalMatrix)[i] = multiply(matrix1, matrix2)
}

func readFile(filePath string) (matrix1 [][]int, matrix2 [][]int)
{
    file, err := os.Open(filePath)
    if err != nil {
        log.Fatal(err)
    }
    defer file.Close()
```

```
    var temp []int
    matrixNr := 1
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        words := strings.Fields(scanner.Text())
        if len(words) != 0 {
            for _, element := range words {
                i, err := strconv.Atoi(element)
                if err != nil {
                    log.Fatal(err)
                }
                temp = append(temp, i)
            }
            if matrixNr == 1 {
                matrix1 = append(matrix1, temp)
            } else {
                matrix2 = append(matrix2, temp)
            }
            temp = nil
        } else {
            matrixNr = 2
        }

    }

    if err := scanner.Err(); err != nil {
        log.Fatal(err)
    }
    return
}

func main() {
    file := os.Args[1]
    nrOfThreads, err := strconv.Atoi(os.Args[2])
    if err != nil {
        log.Fatal("USAGE: " + os.Args[0] + " <file>
<nrOfThreads>")
    }

    debug.SetGCPercent(-1)

    if nrOfThreads <= 0 {
        runtime.GOMAXPROCS(1)
    } else if nrOfThreads >= 16 {
        runtime.GOMAXPROCS(8)
    } else {
        runtime.GOMAXPROCS(nrOfThreads)
    }

    var wg sync.WaitGroup
    finishedMatrix := make([][][]int, nrOfThreads)

    matrix1, matrix2 := readFile(file)

    if len(matrix1) != len(matrix2) || (nrOfThreads != 0 &&
len(matrix1)%nrOfThreads != 0) {
        log.Fatal("USAGE: " + os.Args[0] + " <file>
<nrOfThreads>")
    }
```

```
    var start int64

    if nrOfThreads == 0 {
        start = time.Now().UnixNano()
        multiply(matrix1, matrix2)
    } else {
        matrixes := splitMatrix(nrOfThreads, matrix1)

        start = time.Now().UnixNano()
        for i := 0; i < nrOfThreads; i++ {
            wg.Add(1)
            go func(index int) {
                defer wg.Done()
                multiplyStuff(&finishedMatrix, matrixes[index],
matrix2, index)
            }(i)
        }

        wg.Wait()
    }

    end := time.Now().UnixNano()
    fmt.Printf("Execution took %d ns\n", (end - start))
    runtime.GC()
}
```

10.2 GoTest.sh

```
#! /bin/sh
COUNTER=0
MAX=7

go build MatrixMultiGo.go

> result.txt

echo "Cold Running tests"
nice -n -20 ./MatrixMultiGo 4096.in 16 >> /dev/null

echo "Running tests for no multithreading..."
echo "### No Multithreading" >> result.txt
while [ $COUNTER -lt $MAX ]
do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 ./MatrixMultiGo 4096.in 0 >> result.txt
done

COUNTER=0
echo "Running tests for 1 thread..."
echo "### Threading with 1 thread" >> result.txt
while [ $COUNTER -lt $MAX ]
do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 ./MatrixMultiGo 4096.in 1 >> result.txt
done

COUNTER=0
echo "Running tests for 2 threads..."
echo "### Multithreading with 2 threads" >> result.txt
while [ $COUNTER -lt $MAX ]
do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 ./MatrixMultiGo 4096.in 2 >> result.txt
done

COUNTER=0
echo "Running tests for 4 threads..."
echo "### Multithreading with 4 threads" >> result.txt
while [ $COUNTER -lt $MAX ]
do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 ./MatrixMultiGo 4096.in 4 >> result.txt
done

COUNTER=0
echo "Running tests for 8 threads..."
echo "### Multithreading with 8 threads" >> result.txt
while [ $COUNTER -lt $MAX ]
do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 ./MatrixMultiGo 4096.in 8 >> result.txt
```

```
done

COUNTER=0
echo "Running tests for 16 threads..."
echo "### Multithreading with 16 threads" >> result.txt
while [ $COUNTER -lt $MAX ]
do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 ./MatrixMultiGo 4096.in 16 >> result.txt
done

echo "Done";
```

10.3 Java_ExecutorService.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.ArrayList;

public class Java_ExecutorService {
    public List<ArrayList<ArrayList<Integer>>> read(String
filename) {
        ArrayList<ArrayList<Integer>> A = new
ArrayList<ArrayList<Integer>>();
        ArrayList<ArrayList<Integer>> B = new
ArrayList<ArrayList<Integer>>();

        String thisLine;

        try {
            BufferedReader br = new BufferedReader(new
FileReader(filename));

            while ((thisLine = br.readLine()) != null) {
                if (thisLine.trim().equals("")) {
                    break;
                } else {
                    ArrayList<Integer> line = new
ArrayList<Integer>();
                    String[] lineArray = thisLine.split("\t");
                    for (String number : lineArray) {
                        line.add(Integer.parseInt(number));
                    }
                    A.add(line);
                }
            }

            while ((thisLine = br.readLine()) != null) {
                ArrayList<Integer> line = new
ArrayList<Integer>();
                String[] lineArray = thisLine.split("\t");
                for (String number : lineArray) {
                    line.add(Integer.parseInt(number));
                }
                B.add(line);
            }
            br.close();
        } catch (IOException e) {
            System.err.println("Error: " + e);
        }

        List<ArrayList<ArrayList<Integer>>> res = new
LinkedList<ArrayList<ArrayList<Integer>>>();
        res.add(A);
        res.add(B);
        return res;
    }
```

```
    public int[][]
matrixMultiplication(ArrayList<ArrayList<Integer>> A,
ArrayList<ArrayList<Integer>> B, int m, int n) {
        int[][] C = new int[m][n];

        for (int i = 0; i < m; i++) {
            for (int k = 0; k < n; k++) {
                int temp = A.get(i).get(k);
                for (int j = 0; j < n; j++) {
                    C[i][j] += temp * B.get(k).get(j);
                }
            }
        }

        return C;
    }

    public ArrayList<ArrayList<ArrayList<Integer>>>
splitMatrix(ArrayList<ArrayList<Integer>> A, int nrOfThreads) {
        int n = A.size();
        int m = n / nrOfThreads;
        ArrayList<ArrayList<ArrayList<Integer>>> B = new
ArrayList<ArrayList<ArrayList<Integer>>>();

        for (int i = 0; i < nrOfThreads; i++){
            B.add(new ArrayList<ArrayList<Integer>>(A.subList(i*m,
(i+1)*m)));
        }
        return B;
    }

    public static void main(String[] args) {
        String filename = "";
        int nrOfThreads = 0;
        if (args.length < 2) {
            System.out.println(
                    "Missing filename and/or number of
threads\nUSAGE: java Java_ExecutorService <file> <nrOfThreads>");
            System.exit(1);
        } else {
            filename = args[0];
            nrOfThreads = Integer.parseInt(args[1]);
            Java_ExecutorService java_ExecutorService = new
Java_ExecutorService();
            java_ExecutorService.start(filename, nrOfThreads);
        }

    }

    public void start(String filename, int nrOfThreads) {
        List<ArrayList<ArrayList<Integer>>> matrices =
read(filename);
        ArrayList<ArrayList<Integer>> A = matrices.get(0);
        ArrayList<ArrayList<Integer>> B = matrices.get(1);

        if (nrOfThreads <= 0) {
            int n = A.size();
            long startTime = System.nanoTime();
            int[][] C = matrixMultiplication(A, B, n, n);
```

```
            long endTime = System.nanoTime();

            System.out.println("Execution took " + (endTime -
startTime) + " ns");
        } else {
            if (A.size() % nrOfThreads != 0) {
                System.out.println("Size of matrix is not
divisible by the supplied number of threads");
                System.exit(1);
            }
            ArrayList<int[][]> result = new ArrayList<int[][]>();
            int[][] empty = new int[][]{{}};

            for(int i = 0; i < nrOfThreads; i++){
                result.add(empty);
            }

            ArrayList<ArrayList<ArrayList<Integer>>>
workerMatrices = splitMatrix(A, nrOfThreads);

            long startTime = System.nanoTime();

            ExecutorService ex =
Executors.newFixedThreadPool(nrOfThreads);
            for (int i = 0; i < nrOfThreads; i++) {
                ex.execute(new Worker(workerMatrices.get(i), B, i,
result));
            }
            ex.shutdown();

            try {
                ex.awaitTermination(10L, TimeUnit.MINUTES);
            } catch(Exception e){
                System.err.println(e);
            }

            long endTime = System.nanoTime();

            System.out.println("Execution took " + (endTime -
startTime) + " ns");
        }
    }

    class Worker implements Runnable {
        private ArrayList<ArrayList<Integer>> A;
        private ArrayList<ArrayList<Integer>> B;
        private int index;
        private ArrayList<int[][]> result;
        private int m;
        private int n;

        public Worker(ArrayList<ArrayList<Integer>> A,
ArrayList<ArrayList<Integer>> B, int index,
                ArrayList<int[][]> result) {
            this.A = A;
            this.B = B;
            this.index = index;
            this.result = result;
            this.m = A.size();
            this.n = B.size();
```

```
        }

        @Override
        public void run() {
            this.result.set(this.index,
matrixMultiplication(this.A, this.B, this.m, this.n));
        }

    }
}
```

10.4 Java_ExecutorServiceTest.sh

```
#! /bin/sh
COUNTER=0
MAX=7

javac Java_ExecutorService.java

> result.txt

echo "Cold Running tests"
nice -n -20 java -Xms8g -Xmx8g Java_ExecutorService 4096.in 16 >>
/dev/null

echo "Running tests for no multithreading..."
echo "### No Multithreading" >> result.txt
while [ $COUNTER -lt $MAX ]; do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 java -Xms8g -Xmx8g Java_ExecutorService 4096.in
0 >> result.txt
done

COUNTER=0
echo "Running tests for 1 thread..."
echo "### Threading with 1 thread" >> result.txt
while [ $COUNTER -lt $MAX ]; do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 java -Xms8g -Xmx8g Java_ExecutorService 4096.in
1 >> result.txt
done

COUNTER=0
echo "Running tests for 2 threads..."
echo "### Multithreading with 2 threads" >> result.txt
while [ $COUNTER -lt $MAX ]; do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 java -Xms8g -Xmx8g Java_ExecutorService 4096.in
2 >> result.txt
done

COUNTER=0
echo "Running tests for 4 threads..."
echo "### Multithreading with 4 threads" >> result.txt
while [ $COUNTER -lt $MAX ]; do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 java -Xms8g -Xmx8g Java_ExecutorService 4096.in
4 >> result.txt
done

COUNTER=0
echo "Running tests for 8 threads..."
echo "### Multithreading with 8 threads" >> result.txt
while [ $COUNTER -lt $MAX ]; do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 java -Xms8g -Xmx8g Java_ExecutorService 4096.in
```

```
8 >> result.txt
done

COUNTER=0
echo "Running tests for 16 threads..."
echo "### Multithreading with 16 threads" >> result.txt
while [ $COUNTER -lt $MAX ]; do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 java -Xms8g -Xmx8g Java_ExecutorService 4096.in
16 >> result.txt
done

echo "Done";
```

10.5 Java_Thread.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.LinkedList;
import java.util.List;
import java.util.ArrayList;

public class Java_Thread {
    public List<ArrayList<ArrayList<Integer>>> read(String
filename) {
        ArrayList<ArrayList<Integer>> A = new
ArrayList<ArrayList<Integer>>();
        ArrayList<ArrayList<Integer>> B = new
ArrayList<ArrayList<Integer>>();

        String thisLine;

        try {
            BufferedReader br = new BufferedReader(new
FileReader(filename));

            while ((thisLine = br.readLine()) != null) {
                if (thisLine.trim().equals("")) {
                    break;
                } else {
                    ArrayList<Integer> line = new
ArrayList<Integer>();
                    String[] lineArray = thisLine.split("\t");
                    for (String number : lineArray) {
                        line.add(Integer.parseInt(number));
                    }
                    A.add(line);
                }
            }

            while ((thisLine = br.readLine()) != null) {
                ArrayList<Integer> line = new
ArrayList<Integer>();
                String[] lineArray = thisLine.split("\t");
                for (String number : lineArray) {
                    line.add(Integer.parseInt(number));
                }
                B.add(line);
            }
            br.close();
        } catch (IOException e) {
            System.err.println("Error: " + e);
        }

        List<ArrayList<ArrayList<Integer>>> res = new
LinkedList<ArrayList<ArrayList<Integer>>>();
        res.add(A);
        res.add(B);
        return res;
    }

    public int[][]
matrixMultiplication(ArrayList<ArrayList<Integer>> A,
```

```
ArrayList<ArrayList<Integer>> B, int m, int n) {
        int[][] C = new int[m][n];

        for (int i = 0; i < m; i++) {
            for (int k = 0; k < n; k++) {
                int temp = A.get(i).get(k);
                for (int j = 0; j < n; j++) {
                    C[i][j] += temp * B.get(k).get(j);
                }
            }
        }

        return C;
    }

    public ArrayList<ArrayList<ArrayList<Integer>>>
splitMatrix(ArrayList<ArrayList<Integer>> A, int nrOfThreads) {
        int n = A.size();
        int m = n / nrOfThreads;
        ArrayList<ArrayList<ArrayList<Integer>>> B = new
ArrayList<ArrayList<ArrayList<Integer>>>();

        for (int i = 0; i < nrOfThreads; i++){
            B.add(new ArrayList<ArrayList<Integer>>(A.subList(i*m,
(i+1)*m)));
        }
        return B;
    }

    public static void main(String[] args) {
        String filename = "";
        int nrOfThreads = 0;
        if (args.length < 2) {
            System.out.println(
                    "Missing filename and/or number of
threads\nUSAGE: java Java_Thread <file> <nrOfThreads>");
            System.exit(1);
        } else {
            filename = args[0];
            nrOfThreads = Integer.parseInt(args[1]);
            Java_Thread java_Thread = new Java_Thread();
            java_Thread.start(filename, nrOfThreads);
        }

    }

    public void start(String filename, int nrOfThreads) {
        List<ArrayList<ArrayList<Integer>>> matrices =
read(filename);
        ArrayList<ArrayList<Integer>> A = matrices.get(0);
        ArrayList<ArrayList<Integer>> B = matrices.get(1);

        if (nrOfThreads <= 0) {
            int n = A.size();
            long startTime = System.nanoTime();
            int[][] C = matrixMultiplication(A, B, n, n);
            long endTime = System.nanoTime();

            System.out.println("Execution took " + (endTime -
startTime) + " ns");
```

```
        } else {
            if (A.size() % nrOfThreads != 0) {
                System.out.println("Size of matrix is not
divisible by the supplied number of threads");
                System.exit(1);
            }
            ArrayList<Worker> threads = new ArrayList<Worker>();
            ArrayList<int[][]> result = new ArrayList<int[][]>();
            int[][] empty = new int[][]{{}};

            for (int i = 0; i < nrOfThreads; i++) {
                result.add(empty);
            }

            ArrayList<ArrayList<ArrayList<Integer>>>
workerMatrices = splitMatrix(A, nrOfThreads);

            long startTime = System.nanoTime();

            for (int i = 0; i < nrOfThreads; i++) {
                threads.add(new Worker(workerMatrices.get(i), B,
i, result));
                threads.get(i).start();
            }

            for (int i = 0; i < nrOfThreads; i++) {
                try {
                    threads.get(i).join();
                } catch (Exception e) {
                    System.err.println(e);
                }
            }
            long endTime = System.nanoTime();

            System.out.println("Execution took " + (endTime -
startTime) + " ns");
        }
    }

    class Worker extends Thread {
        private ArrayList<ArrayList<Integer>> A;
        private ArrayList<ArrayList<Integer>> B;
        private int index;
        private ArrayList<int[][]> result;
        private int m;
        private int n;

        public Worker(ArrayList<ArrayList<Integer>> A,
ArrayList<ArrayList<Integer>> B, int index,
                ArrayList<int[][]> result) {
            this.A = A;
            this.B = B;
            this.index = index;
            this.result = result;
            this.m = A.size();
            this.n = B.size();
        }

        @Override
        public void run() {
```

```
            this.result.set(this.index,
matrixMultiplication(this.A, this.B, this.m, this.n));
        }

    }
}
```

10.6 Java_ThreadTest.sh

```
#! /bin/sh
COUNTER=0
MAX=7

javac Java_Thread.java

> result.txt

echo "Cold Running tests"
nice -n -20 java -Xms8g -Xmx8g Java_Thread 4096.in 16 >> /dev/null

echo "Running tests for no multithreading..."
echo "### No Multithreading" >> result.txt
while [ $COUNTER -lt $MAX ]; do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 java -Xms8g -Xmx8g Java_Thread 4096.in 0 >>
result.txt
done

COUNTER=0
echo "Running tests for 1 thread..."
echo "### Threading with 1 thread" >> result.txt
while [ $COUNTER -lt $MAX ]; do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 java -Xms8g -Xmx8g Java_Thread 4096.in 1 >>
result.txt
done

COUNTER=0
echo "Running tests for 2 threads..."
echo "### Multithreading with 2 threads" >> result.txt
while [ $COUNTER -lt $MAX ]; do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 java -Xms8g -Xmx8g Java_Thread 4096.in 2 >>
result.txt
done

COUNTER=0
echo "Running tests for 4 threads..."
echo "### Multithreading with 4 threads" >> result.txt
while [ $COUNTER -lt $MAX ]; do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 java -Xms8g -Xmx8g Java_Thread 4096.in 4 >>
result.txt
done

COUNTER=0
echo "Running tests for 8 threads..."
echo "### Multithreading with 8 threads" >> result.txt
while [ $COUNTER -lt $MAX ]; do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 java -Xms8g -Xmx8g Java_Thread 4096.in 8 >>
result.txt
```

```
done

COUNTER=0
echo "Running tests for 16 threads..."
echo "### Multithreading with 16 threads" >> result.txt
while [ $COUNTER -lt $MAX ]; do
    ((COUNTER++))
    echo "Running test #$COUNTER"
    nice -n -20 java -Xms8g -Xmx8g Java_Thread 4096.in 16 >>
result.txt
done

echo "Done";
```

10.7 Experiment Result (Go, Java Thread & Java ExecutorService)

10.7.1 Go

Run #	No multithreading	Threading with 1 thread	Multithreading with 2 threads
1	117386663759 ns	116342154427 ns	56351459720 ns
2	116869812642 ns	117001348145 ns	55591390001 ns
3	116945982579 ns	117372533903 ns	58199278808 ns
4	117291574008 ns	116298482703 ns	57142427820 ns
5	117239937175 ns	116901127505 ns	54882398184 ns
6	116716089284 ns	116924806034 ns	53977385604 ns
7	116505379916 ns	116838537369 ns	57205524098 ns

Run #	Multithreading with 4 threads	Multithreading with 8 threads	Multithreading with 16 threads
1	28745760965 ns	30452495837 ns	30572978761 ns
2	29556234853 ns	30519239341 ns	30691757796 ns
3	29900788960 ns	30426248255 ns	30664999515 ns
4	29928625356 ns	30629706746 ns	30689749990 ns
5	30074020510 ns	30690917309 ns	30494402262 ns
6	30070320498 ns	30557729941 ns	30588729964 ns
7	30245953787 ns	30597237885 ns	30484792492 ns

10.7.2 Java Thread

Run #	No multithreading	Threading with 1 thread	Multithreading with 2 threads
1	153235687993 ns	154722276854 ns	74272389261 ns
2	155218701861 ns	158113749453 ns	73616763448 ns
3	156083961348 ns	153149213072 ns	75644627191 ns
4	154232137936 ns	156571564089 ns	75318337596 ns
5	155301373103 ns	153613754413 ns	73413441564 ns
6	156723408026 ns	158066272150 ns	75262168852 ns
7	158075028178 ns	150735984581 ns	74244357896 ns

Run #	Multithreading with 4 threads	Multithreading with 8 threads	Multithreading with 16 threads
1	40050975293 ns	30935060718 ns	31890776084 ns
2	40568793861 ns	31187704696 ns	32761088411 ns
3	40472880652 ns	31434171650 ns	31822386201 ns
4	40246881320 ns	31664135978 ns	31437059349 ns
5	40676387385 ns	31698970789 ns	33205605819 ns
6	40084958497 ns	31616412969 ns	32176735658 ns
7	40078214332 ns	31198305209 ns	31965180172 ns

10.7.3 Java ExecutorService

Run #	No multithreading	Threading with 1 thread	Multithreading with 2 threads
1	154665066821 ns	157431416820 ns	73986056606 ns
2	151545789468 ns	154229262749 ns	73472301691 ns
3	157295496567 ns	152879349637 ns	73969552042 ns
4	153627276117 ns	153761700849 ns	74769194219 ns
5	155669428569 ns	154839469812 ns	74814259303 ns
6	153714880818 ns	154165982791 ns	75539163980 ns
7	155058309513 ns	153020538471 ns	73700290867 ns

Run #	Multithreading with 4 threads	Multithreading with 8 threads	Multithreading with 16 threads
1	39726430506 ns	30989557012 ns	32025318217 ns
2	40152512322 ns	31066688300 ns	31344335489 ns
3	40403826129 ns	30883675669 ns	32996530102 ns
4	40520034044 ns	31162636138 ns	31838169988 ns
5	40430191833 ns	31471491878 ns	32687284441 ns
6	40605274198 ns	31592427799 ns	31399603563 ns
7	40464690042 ns	30904093502 ns	32320430090 ns