Programming Languages Final Report

Pranav Raghavan

Winter 2020

1 Introduction

More's Law says that the number of transistors on a microchip doubles every year in the early days of computing. Unfortunately, we have hit the limit on the number of transistors we can put on microchips. Augmenting More's Law to be about *speed up* instead makes it so that we can use all the cores of the CPU to make computers faster every year. So it is up to the programmer to bring about asynchronous components to increase speed up. For example, a Linux kernel runs 10x as fast when provided with ten cores. Linear or Exponential speedup is the end goal of multithreading. Non-determinism is the reason why multithreading can be so problematic. The Operating System is in charge of scheduling the threads in languages that do not use cooperative threads like Go. Just by looking at the code, it would be hard to tell which lines from the different threads will get executed first, and for this reason, there are deadlocks, race conditions, and starvation that come up in applications that are in production.

About 80% of bugs that come up in production systems have no solution for them because of the non-determinism that multithreading causes. [1] There are "Heisenbugs" (a parallel to the Observer's effect in physics) in which the act of trying to observe the bug (say through a debugger) would affect the outcome of the program and not manifest itself anymore. Moreover, a race condition might happen after a million perfect executions of the program - this is how programs become non-deterministic when they are multithreaded. In systems that cannot afford any downtime, they will have to do a reboot or a partial reboot just to get the system to work again [2]. Generally, these systems are in C, like most systems, and these issues are as much a fault of language as of the hardware itself. "Safer" languages such as rust boast features like ownership of memory, reference-counted objects, and so on that make it hard to introduce these types of bugs. Any piece of code that gets through "rustc" the rust compiler is threadsafe. Unless they are using a lot of "unsafe" syntax, in which case they are defeating the purpose of the language.

These asynchronous features need to be as simple as possible to create the best chance for programmers to implement their ideas in the **simplest** way they can. Simplicity can be subjective, so we will define it as a way to get from a single-threaded solution to a multithreaded solution quickly. One of Go's strengths is that it does not require much knowledge about systems to write multithreaded code in Go because it is all optimized to give the best performance. The C and C++ compilers are a lot slower than Go, faster compile time is the main reason why google transitioned away from C and C++ so that they can expedite the build cycle.

The languages that we evaluated in this project are C, C++, Go, and Rust. C and C++ are slightly older languages that have a way of multithreading that is the accepted standard. For decades people have been writing multithreaded code the "c way." The newer languages like Go and Rust do aim to make this multithreading of their algorithms easier (but in different ways) using innovative techniques. Go will make building the system more straightforward, but it might not have all the safety that Rust offered for threads or even the security that it offers. Rust will make building the system harder, but much debugging will become a lot easier since race conditions and deadlocks are not even possible. At the same time, it will suffer from performance overhead due to the added security.

The testing framework is a python script that uses the NumPy package (which does multithreading to reduce latency). Python is a pervasive language in the industry, but it has been used more to create prototypes and not for a fast, highly parallel algorithm. The speed of the interpreter and the inability to use parallel threads of execution are the biggest reasons for that.

2 Algorithm

Matrix Multiplication is a well-known problem in multithreading because there is not a very elegant way to solve it. The reason for this is that this problem stresses the cache. By the definition of matrix multiplication, elements in different parts of the matrix data structure have to be read and written. This algorithm tests the cache of the machine so it will not work as fast with machines that have less of it. In a powerful enough machine, having many parallel threads of execution makes the program run exponentially faster.

1. Dot product - which is just to take the dot product of each row and column vector to form a resultant matrix.
2. Strassen's algorithm - which uses divide and conquers to solve this problem, thereby exploits cache coherency a lot better than the simplistic approach of using the dot product of the row and column vectors.
3. Transpose before multiplication is another way to use row-wise iteration through the matrices.

The multiplication function takes in two matrices and gives out a resultant matrix that has the same number of rows as the first matrix and the same number of columns as the second. The reason that we can reason about these matrices as the first and second one is that matrix multiplication is not associative like integer multiplication. If the number of columns of the first matrix is not the same as the number of rows of the second matrix, matrix multiplication is not possible.

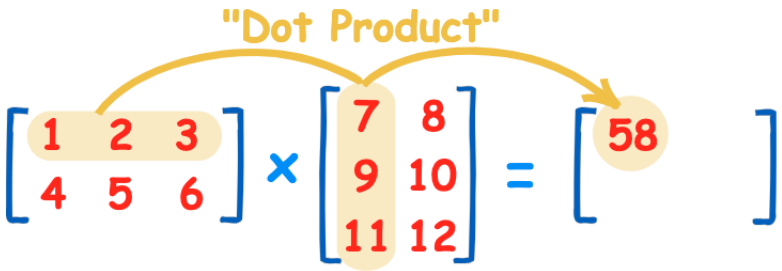

Figure 1: The dot product is the basic unit of computation in matrix multiplication. Computing all the dot products in a row is the function that gets called serially by each of the cores (on the rows that the boss thread assigned them

The basic unit of computation in this algorithm is a dot product of a row vector and a column vector. The dot product of the first row from the first matrix and the first column of the second matrix gives the first element in the first row of the matrix. Some possible approaches to multithreading this algorithm might be

1. dividing the number of dot products by the number of cores.
2. dividing the number of rows by the number of cores to divide the workload evenly.

This evaluation uses the second approach of splitting the number of rows by the number of cores available because of the added complexity of splitting starting row multiplication in the middle of the row. The running time of this algorithm is O (n^3) (assuming we have equal rows and cols) because each dot product is O(n), and we have to do n (rows) x n (cols) number of dot products. When paralleled, this program does a lot better than O(n^3) since many calculations can happen at the same time.

The parallel version of the program from figure 2 shows how this algorithm works. The main thread spawns as many threads as available on the system and then solves the smallest division of the number of rows.

3 C

C language is one of the oldest languages in the industry. It is still used for the majority of systems even though it has "unsafe" features. C has been the most used language for a long time. There is a lot of legacy code out there. The Linux

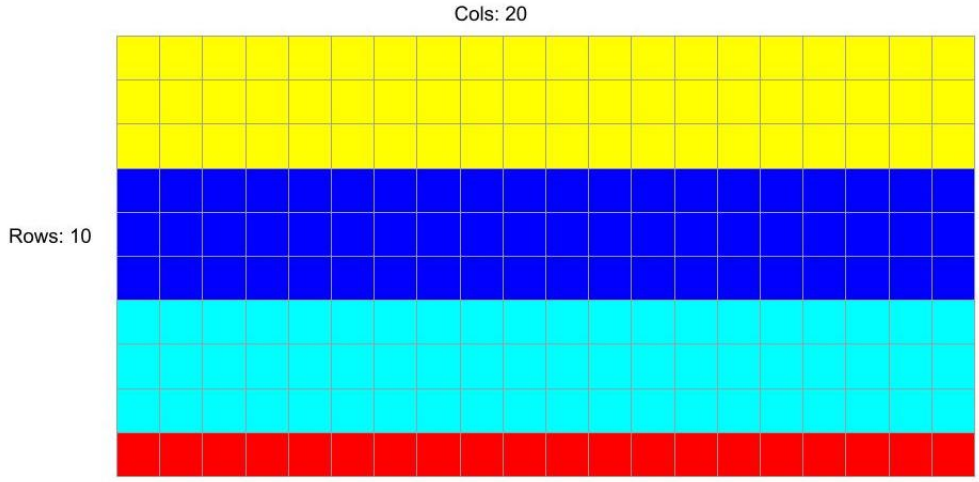

Figure 2: Approach 2, dividing the rows by the number of cores. If there are four cores on a machine, the main thread will hand off the first three rows to the first core, the next three rows to the second core, the third three rows to the third core and the main thread will solve the last row since it spawned three threads, it does fewer dot products so as the even out the work that the threads are doing.

kernel is in C as well, and it would be arduous to convert that to another language. The language has existed for a long time, and many systems are written in it, which means it would be easier to find help when something goes wrong. It is easier to find someone else who dealt with a similar issue making it perfect for building ”reliable” systems. Software engineers are under the illusion that it is reliable because of all the preexisting legacy code. The phrase, *A known devil is better than an unknown angel* applies perfectly to this aspect of using C over a newer language. The days of C domination are coming to an end. Software engineers will realize that true reliability comes from thread-safety alone.

In terms of security as well, this language is not perfect. Many security exploits rely on C’s use of memory. Using Rust will give these modules high security since these memory locations are owned by only one function at a time because it has bounds checking to ensure that the user of a block of memory is the ”owner.” Rust data structures are not mutable or even readable without changing the rust code itself. A double-free leads to memory corruption, which is a way that hackers exploit a system. It is impossible to write a double-free because of memory ownership - the compiler will just not let users do it. However, this safety comes at a price performance.

A reoccurring theme in systems design is that performance and security are at odds with one another. Since all the threads can share the addresses of variables in the program at the same time, there is less copying and sending messages between threads. Sending messages is a fast way for threads to communicate, but the ”C way” is to use the memory itself to enable this communication. There are many benefits to this approach, especially when running on a lower amount of computation power. An embedded device, for example, probably does not have enough memory to run Rust or Go or support their compilers. However, in general, Go is more minimalistic and faster than C for algorithms that do not require external dependencies, and it has the tools for finding race conditions.

The general idea in this multithreaded implementation is the same as the algorithm shown above, split the rows by the number of cores and then spawn similarly sized tasks in as many logical cores as the computer offers.

4 C++

This language supports the object-oriented model, and that can make it convenient to have ”nouns” as the classes, and the actions that those nouns are capable of are the member functions. Object-Oriented Programming makes the core organization easy. In languages like Go and C, the file itself acts like a class, and importing other files allows usage of code implemented in other files. Organization of code is necessary when working with a large system because

debugging involves looking at many aspects of the code, including other components.

There are endless multi-threading primitives in this language, and the C programmers might use p_thread and mutexes from C to get the same output, and that is fine because C++ is a superset of C. Generally, most of these C++ primitives utilize a C mutex in the background as well. A mutex ensures that only one of the threads has access to the critical section of the code at a time. It is standard programming practice to minimize the number of multiple mutable writes. The ”thread” **object**, which gets created from calling the constructor of the class, is the only multi-threading primitive that was needed just like how p_thread_create was the only one needed in the C version. In C++, users can pass in member functions to the thread making it more user-friendly than p_thread.

The algorithm is unchanged for C++. The only difference is the use of classes and thread objects. C does not let users pass in many arguments to the p_thread function. Generally, programmers create structs with the addresses of all the memory that it has access to, so it requires users to be more explicit just like C, but for C++, references to objects are all that it takes. It also has no guarantees for thread safety or security features like bound checking like Rust does so it can perform better. In terms of pure performance, however, we can see from figure 6 that a single-threaded C++ program performs the worst. From figure 5, we can see that it does perform better than Rust, but it completely lacks tools for checking race conditions. Doing less will always make an algorithm run faster. If complicated data structures are not involved, then Go and C solutions are going to be faster than C++ because they are minimalist, and that adds a lot of performance gain. Most software needs to use external dependencies, and those dependencies need to be fast as well to achieve high performance.

C++ is known for its optimizations. The vector in C++ will dynamically allocate more memory as and when it is needed and, therefore, will use as little memory as needed. The sort function in C++ will use a hybrid sorting algorithm using parts of Quick-sort, Heap-sort, and Insertion sort to give the best performance. Internally, the implementation of p_thread and thread are very similar. For this reason, we see higher speedup in C++ and C, as seen in figure 3 because C++ has an optimal thread spawning procedure.

More importantly, it **has** the tools to create data structures a hash table, set vector, queue, stack, and many more. These data structures are already optimized, giving systems programmers the freedom to focus on the real problem. In C, one might have to build a hashtable from scratch if they want to use it. Obviously will not perform as well as a hashtable in C++ or Rust, for example. If one chooses to use the ”Boost” package, they can use data structures that are not in the Standard Template Library. If complicated data structures are not in use, C might do a lot better because it does not do as much.

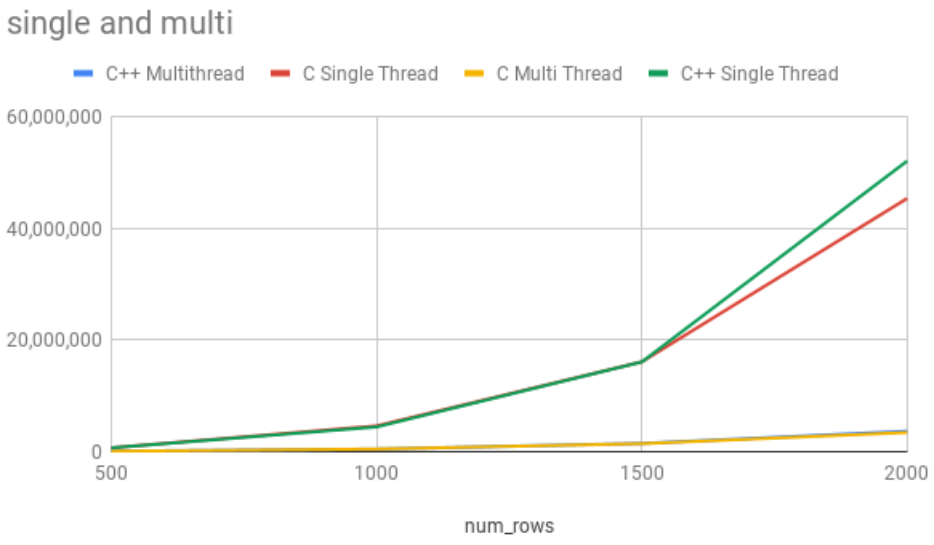

Figure 3: Comparison between C and C++ running times in multithreaded and single threaded mode.

5 Go

The Go Language uses the idea of cooperative scheduling and cooperative threads instead of the native threads that Java or C use.[3] The operating system offers threads natively, but the scheduling is done over in kernel. The Go scheduler, which schedules Go routines, will maintain a queue of the threads and will **not context switch** away from the thread until the thread relinquishes control or ends. Go routines give programmers the ability to start a lot more threads than they could in a language that has native OS threads. Generally, it is the programmer's job to come up with the most optimal parallelization of the algorithm. Go routines allow the user only to specify a smaller unit of computation (row of dot products in this solution). The Go scheduler runs the go routines optimally to get the same performance as coming up with the optimal solution themselves. This idea is great for writing smaller programs like this one. For a server or data center where performance matters a lot, programmers would prefer traditional threads and optimizing the program themselves.

For the Go language, even though the programmer does not need to specify the perfect parallelization of the program, the performance would be better if they did. This solution is comparable *as is* to the other languages. For this reason, Go is an "easy" language for multithreading. However, to optimize the solution, its performance is similar to the other languages.

At compile-time, Go makes no checks for race conditions and will allow something like **multiple mutable access** without warning just like C. At runtime, they might find out that they did this and then they will run "go test-race" which will tell them to use a lock on a particular Rust would not allow code like this to compile in the first place which is suitable for large systems.

From figure 5 we can see that Go is the fastest (single and multithread) in this evaluation, and it is also the most straightforward language to use. The only problem with Go is that it does not have the same guarantees at compile-time, so it might be harder to catch race conditions. Thankfully some tools can find them.

6 Rust

Rust is a fascinating programming language. It runs close to the hardware, meaning that it is perfect for building systems (especially multithreaded systems). In this program as well, Rust performed the best in a single-threaded setting, as seen in 1. Being **explicit** is one of the main design principles behind Rust. It can be even more explicit than C, which is a big reason why people cannot get over the learning curve of the language.

Rust is a "thread-safe" language meaning that if the code compiles, then it will not end up having race-conditions or deadlocks. Thread safety is the reason why it can be slightly harder to write code in Rust. The truth of the matter is that thread safety is not trivial, and it requires the programmer to know the implication of the code they write at a system level. C++ lets us have multiple mutable access from different threads. Multiple mutable access increases the chances of race conditions, which are incredibly hard to debug. In Rust, they have to make an explicit decision to have atomic reference-counted (ARC) pointers if they want to pass variables into threads. More importantly, they would have to make sure that it follows ownership rules, which would ensure that they do not get race conditions. As mentioned in the introduction, 80 % of bugs in the industry are from deadlocks and race conditions and have no fix. In Rust, this would not happen; programmers will not be able to write race conditions into their code unless they use "unsafe" syntax. Generally, Rust is safe to use for large servers and data centers. In an extensive system like a data center or a server that has high availability like nine nines (99.9999999% or downtime 31.56 milliseconds per year) or even two nines (99% or 3.65 days per year), it would not be smart at all to risk deadlocks or data races.

The use of external programming tools like high-speed data structures is something that Rust has. As programmers, we have to be able to leverage the work of others to make our algorithms better. C++ has many tools, too, and gets compared with Rust a lot. For C++ programmers, features of the language like ownership might seem like a limiting factor in building applications fast, but in reality, the compiler comes in and saves the day for most errors that people write. Thankfully the compiler messages are a lot better than C++ template errors. These errors are infamous for being multiple pages long and even have lines that are too long. In addition to the compiler being useful, it saves users from data races and other huge bugs that people include in their code every day.

One example of its uses in the industry is Mozilla; they invented the language and were able to use multiple threads to enhance the user experience of Firefox users. Web and game developers can take advantage of thread

safety in games and web applications, which makes them a lot faster than JavaScript. Everyone seems to agree that the modern features of the language make it ideal for doing anything. In research, they have evaluated Rust as kernel extensions [4], containers [5], file system drivers, and network drivers [6] and thousands of other examples. In general, Rust has not only performed better, but it is more trusted because it does not suffer from the same thread safety issues that C other languages have.

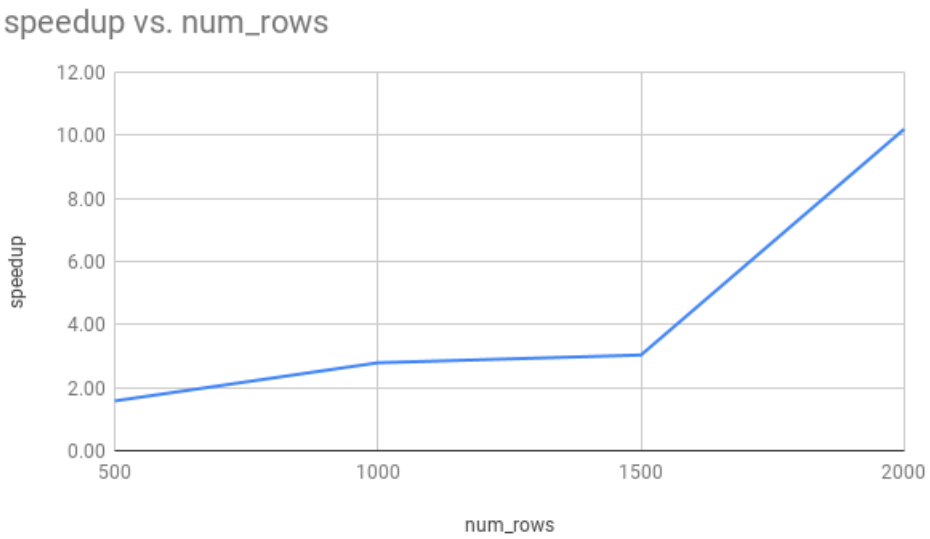

Figure 4: Rust Speedup Graph

With today's computers and data centers, it will most likely run almost as fast and, most importantly, **safer** if written in Rust instead of Go, C, or C++. Rust uses much memory, and when we ran experiments on an 8 GB RAM mac book pro, it was slower across the board for multithreaded tests. A couple of years or decades ago, programmers might have explicitly turned these safety features off if it were available because **performance** used to be the most critical thing. In figure 6, in a massive parallel task, the speedup from using Rust is respectable with 10.19 x in comparison to the unsafe languages, which can get a speedup of 12 x to 14 x. However, in the lower regions, it just does not perform as well because of the ownership rules of Rust that prevent programmers from writing the *most* efficient solution.

7 Testing Framework

The testing framework for this project is important in the evaluation of the speed up. The testing framework performs several tests (450) correctness tests, where it calls the multithreaded version of the algorithm to verify the correctness of the algorithm and cross verifies with the answer received from NumPy. It also has performance tests that run on a square matrix in order to keep the performance testing simple. The performance tests run matrix multiplication on square matrices of size 500, 1000, 1500, and 2000.

Timing a command itself will not give the exact time that it took to execute the multiplication algorithm because 1) there OS latency in spawning threads 2) input of matrices through stdin does not count in the latency of the multiplication. Each of the solutions prints to stdout the latency of calling the multiply function and getting the response. This includes time to allocate the resultant matrix and write to it from the different cores. The script also passes a command-line argument to the ./matrix executable to indicate whether to use just one thread for evaluation or ALL threads available on the system.

The program runs the single-threaded version of each of the four solutions and then runs the multithreaded version

of the program in order to calculate speedup. The speedup is given by

$$Speedup = \frac{Single\ Threaded\ Latency}{Multi - threaded\ Latency} \tag{1}$$

All the measurements are in microseconds in order to get a precise reading.

8 Results

The four programs ran on Hummingbird, which a compute cluster dedicated to UC Santa Cruz. The configurations on this system are listed below. Most saliently, it has 32 CPUs and two threads per core, so a total of 64 logical cores for the algorithm to execute on. Also, the system has a much more powerful cache. Since matrix multiplication is demanding of the cache, the evaluation of this algorithm in a weaker machine will be meaningless. We were able to isolate the latency to just multithreading and not some of the other factors like cache.

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz
Stepping:              4
CPU MHz:               896.191
CPU max MHz:           3000.0000
CPU min MHz:           800.0000
BogoMIPS:              4200.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              11264K
NUMA node0 CPU(s):     0-7,16-23
NUMA node1 CPU(s):     8-15,24-31
```

In conclusion, it is still difficult to say what the best language is because it depends on the use case. Go is excellent for smaller parallel scripts and larger ones as long as there is a testing framework that runs tools like ”go test-race” to ensure that there are no multiple accesses or race conditions. Since Go prioritizes performance, Rust will not perform as well but will have a lot more guarantees. In highly available systems, it might be a lot better that uses Rust. C is the only option for a lot of embedded systems, and the performance is excellent. C++ and Rust are great options if the availability of tools is essential. C++ is a lot more unsafe but performs better.

In terms of multithreaded performance, figure 5 shows that Go is the fastest followed by C, then C++, and finally, Rust. It is impossible to tell C, C++, and Go at a lower number of rows and columns. Rust is not the fastest, but it is the solution for highly available systems.

In terms of single-threaded performance, the fastest solution is still Go, followed by Rust, which tells us that these languages are close to the hardware as well. C is faster than C++ for single-threaded programs so. It would be preferred if the hardware only supports one thread.

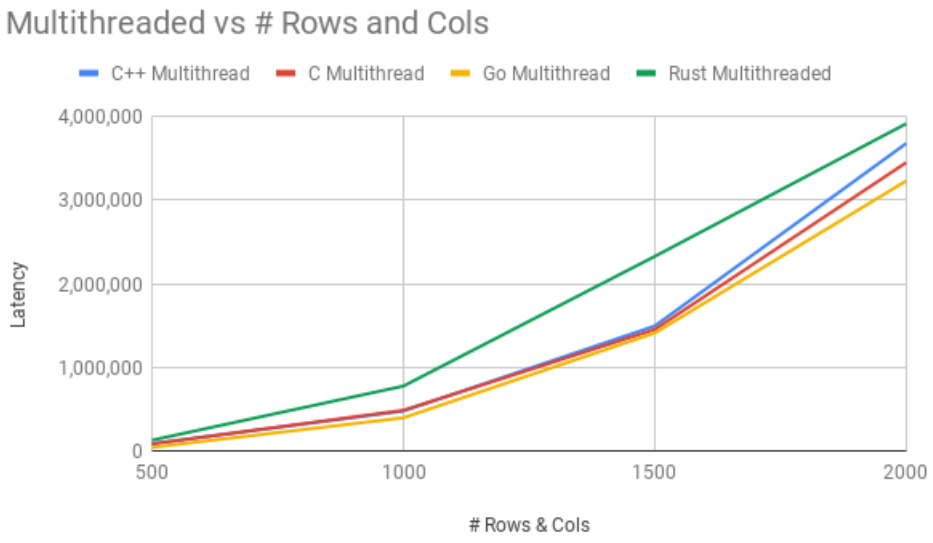

Figure 5: Comparison of the runtimes of multithreaded matrix multiplication in Go, C, C++ and Rust.

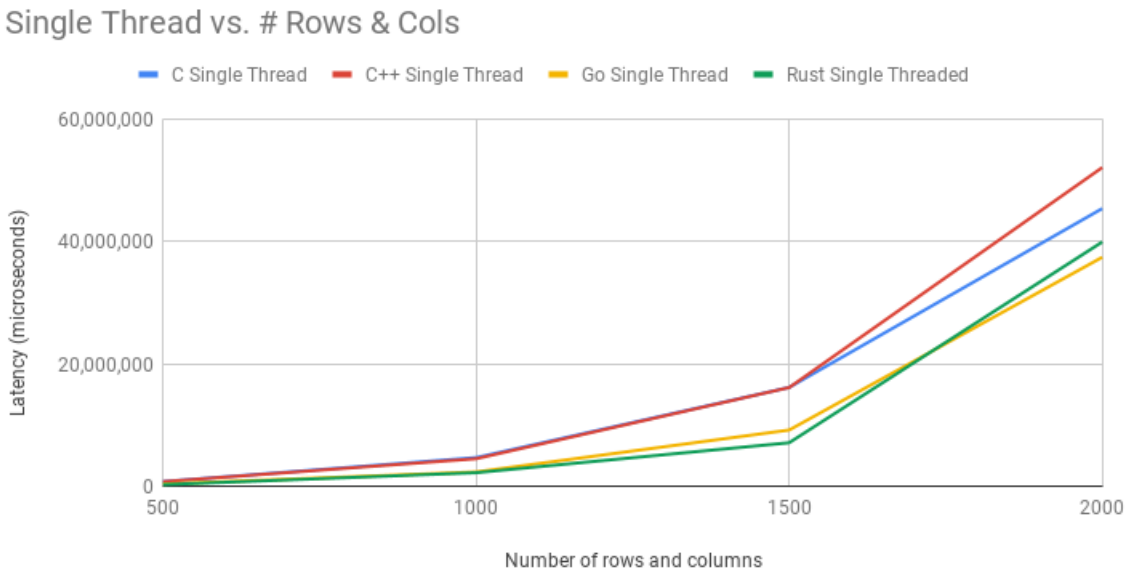

Figure 6: Comparison of the runtimes of single threaded matrix multiplication in Go, C, C++ and Rust.

References

[Woo03] Alan P Wood. “Software reliability from the customer view”. In: *Computer* 36.8 (2003), pp. 37–42.

[Can+04] George Candea, Shinichi Kawamoto, Yuichi Fujiki, et al. “Microreboot — A Technique for Cheap Recovery”. In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*. OSDI’04. San Francisco, CA: USENIX Association, 2004, p. 3.

[AP09] Martin Abadi and Gordon Plotkin. “A model of cooperative threads”. In: *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2009, pp. 29–40.

[Emm+19] Paul Emmerich, Simon Ellmann, Fabian Bonk, et al. “The Case for Writing Network Drivers in High-Level Programming Languages”. In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2019, pp. 1–13.

[Tha+18] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, et al. “Cntr: Lightweight {OS} Containers”. In: *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 2018, pp. 199–212.

[Ell18] Simon Ellmann. “Writing Network Drivers in Rust”. PhD thesis. B. Sc. Thesis. Technical University of Munich, 2018.

9 Tables and Figures

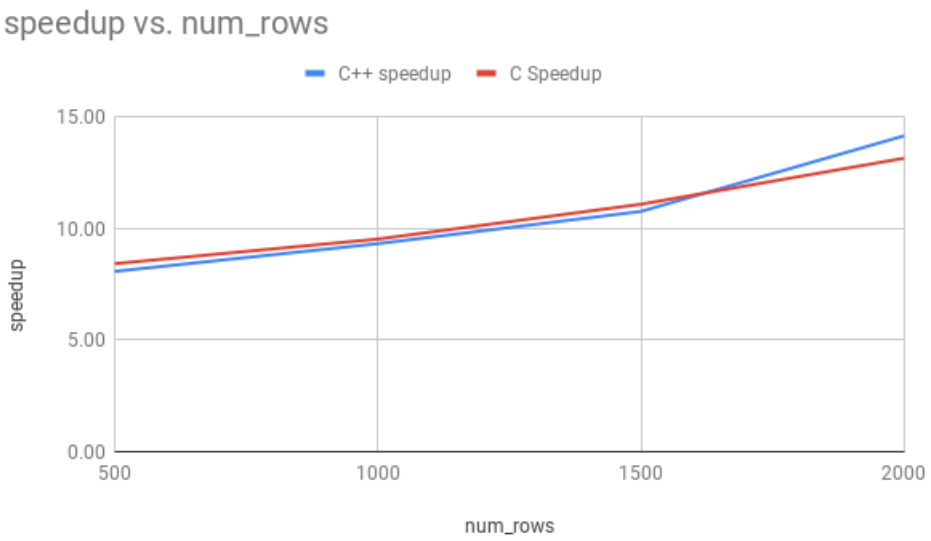

Figure 7: Comparison of the speedup when using

# Rows and Cols	Single Thread (micro-sec)	Multi-thread (micro-sec)	Speedup (Times [x])
500	202,733	128,398	1.58
1000	2,164,110	776,173	2.79
1500	7,065,586	2,329,101	3.03
2000	39,941,235	3,917,738	10.19

Table 1: Speedup of Rust Solution

# Rows and Cols	Single Thread (micro-sec)	Multi-thread (micro-sec)	Speedup (Times [x])
500	306,400	43,188	7.09
1000	2,286,432	396,275	5.77
1500	9,145,617	1,411,208	6.48
2000	37,440,922	3,232,615	11.58

Table 2: Speedup of Go Solution

# Rows and Cols	Single Thread (micro-sec)	Multi-thread (micro-sec)	Speedup (Times [x])
500	709,749	84,201	8.43
1000	4,637,056	486,468	9.53
1500	16,125,318	1,453,539	11.09
2000	45,416,937	3,452,005	13.16

Table 3: Speed up of C solution

# Rows and Cols	Single Thread (micro-sec)	Multi-thread (micro-sec)	Speedup (Times [x])
500	695,825	86,085	8.08
1000	4,449,394	477,067	9.33
1500	16,108,048	1,495,355	10.77
2000	52,133,567	3,681,372	14.16

Table 4: Speed up of C++ solution