Final Report

Stacey Dale Robert Pomichter Shelby Menown
Hannah Moss Patrick Bauer

March 2021

1 Introduction

Blockchain is a technology which first emerged in the academic community in 1991 when Stuart Haber and W. Scott Stornetta conceived of a solution to cryptographically secure data. Specifically, Haber and Stornetta developed a chaining technique which maintained the integrity of records.

It was not until 2009 when a person or group known as Satoshi Nakamoto published the first whitepaper on blockchain; as it was implemented by Bitcoin, that blockchain technology became more widely recognized. When Nakamoto published the whitepaper, Blockchain was used as a ledger system for cryptocurrency. Nakamoto and Bitcoin were motivated by the idea of a completely decentralized form of currency where the complete history of transactions is distributed to everyone participating in the system. To this day, the focus of blockchain technology is primarily the decentralization of the system and the integrity of records[4].

Blockchain works by collating transactional records in structures referred to as blocks. Each block is constructed and appended to a chain of older blocks, which themselves were mined and appended to the chain previously. The block is constructed and then distributed to the network of computers referred to as nodes, where participants attempt to mine the block. After the block is mined successfully by one of the nodes in the network, the other nodes validate the newly mined block by verifying the block for correctness.

The process of mining, validating, and appending the block to the chain is at the heart of blockchain technology. Mining is a process which goes by other names depending on the consensus algorithm utilized in the technology. In the case of Bitcoin and other Proof-of-Work models, mining is a race between all nodes on the network to solve a computationally difficult problem. The first node to successfully solve the computationally taxing problem is said to mine the block.

Nodes on the network attempting to mine the block are referred to as miners. In the case of Bitcoin, miners are incentivized to mine blocks because a successfully mined block yields a reward in the form of cryptocurrency. The

complexity of the mathematical problem is periodically adjusted to ensure the blocks are not exclusively mined by one node on the network. This is the tenuous balance maintained between the blockchain and the miners. As the focus of blockchain technology is decentralization, it is imperative that no one node on the network obtains more than a majority share of all blocks in the chain[4].

There are many other consensus algorithms which all attempt to maintain the fundamental properties of the blockchain technology–integrity and decentralization. These other consensus algorithms include Proof-of-Stake, Proof-of-Burn, Proof-of-Capacity and many others which differ in terms of implementation and by approach to perceived threats on the blockchain[1]. The Proof-of-Work consensus algorithm is the primary focus of this project.

2 Overview

In this project, we implement an educational blockchain in three languages, C++, Java, and Haskell, first sequentially and then concurrently. We chose these three languages because each language uses threads differently. Due to the abstraction the JVM performs, Java threads can run differently depending on the underlying operating system. The user may not be aware of how the JVM is ensuring correct synchronization and concurrency. C++ is considered to have more efficient threads because they are not abstracted through any JVM. Haskell uses fine-grained threads, similarly to C++. However, Haskell has added efficiency in other mathematical calculations, thus making it a valid educational choice for a concurrent blockchain. Choosing an educational blockchain over common blockchain algorithms was an important part of our planning stage, ultimately allowing us to maintain our early decision to use the three languages discussed above to compare concurrent blockchain efficiency.

Initially, we expected to implement some form of Ethereum blockchain using concurrent smart contracts. This idea spawned from current discourse in the field of blockchain and concurrency. However, as we began to research Ethereum blockchain, with reference to the Ethereum whitepaper, we realized that 1) implementing Ethereum blockchain in general would be labor-intensive and 2) using our three chosen languages would not be the best approach to implementing Ethereum. Based on these two realizations, we were able to conclude that adding concurrency to Ethereum blockchain implemented in three different languages was far outside the scope of a semester-long, undergraduate project. Indeed, even becoming familiar with concurrency in Haskell and C++ was going to be a stretch for our team, since we started the semester only with a basic understanding of concurrency in Java.

Thus, we faced a decision: discard our plan to use and compare concurrent blockchain across different languages or change the type of blockchain we would create. We chose the latter after finding a reasonable algorithm for a sequential blockchain in the C language. Our task then became translating the algorithm into each language to create single-threaded blockchains and implementing comparable concurrency across the three different blockchains. By

comparable, we mean using the most logically consistent algorithm we could implement across languages. Logical consistency is quintessential to our analysis; if one blockchain implements a more efficient concurrent algorithm, then the results cannot be compared to our expectations. If, however, the same concurrent algorithm is applied to each blockchain and one blockchain performs much better due to the types of threads it uses and the efficiency of the language, then we can analyze the results in accordance with our expectations.

Since we had decided to use an educational blockchain that does not implement Smart Contracts, we needed to determine which part of the blockchain we would make concurrent. Because the educational blockchain structure we chose uses a Proof-of-Work consensus similar to Bitcoin, we realized that we could use concurrency to solve the puzzle. In Bitcoin, this puzzle varies in difficulty with each run, with the time to solve the puzzle averaging ten minutes per solve. On any given puzzle, though, the difficulty could be greater or less than a ten minute solve. Testing the efficiency of concurrency on an ever-changing difficulty would be complicated, and the results would only be questionably reliable. Thus, our blockchain implementation does not change the difficulty of the puzzle across different tests. Additionally, each blockchain uses the same input file, so the only difference across tests is the timestamp. This is a reasonable difference that should not alter our results or subsequent analysis and conclusion.

Each blockchain implementation is merely academic. It does not support peer-to-peer networks. Rather, one machine creates the blockchain and adds blocks (nodes) with only one transaction (line from the text file). Our focus is 1) to test the efficiency of building the block chain without and with concurrency in one language and 2) to compare the level of efficiency in one language to others. Because of this, our actual code cannot be considered for implementation into current blockchain models. However, current discourse in blockchain is honing in on concurrency, and specifically how concurrency can be added into certain parts of the transaction process to improve efficiency. Our project is yet another exploration into how concurrency can fit into the blockchain process.

3 Research

We spent the first two weeks of our project conducting and combining research via a tool called Zotero. Upon choosing a far-too-broad topic, applications of concurrent blockchain, we dove into foundational research to answer a few broad questions: What is block chain, really, and What is the current discourse on concurrency in block chain?

These two questions led us to explore all aspects of blockchain. We first discovered "A survey of blockchain consensus algorithms performance evaluation criteria", a journal article by Seyed Bamakan, Amirhossein Motavali, and Alireza Bondarti published in Expert Systems with Applications [1]. This article not only provides an in-depth explanation of blockchain, but also details twelve types of consensus algorithms, including the algorithm we ultimately chose to implement: Proof of Work.

We continued to research blockchain, now focusing on concurrency. We found Laxmi Kadariya's thesis on concurrent block chain, "Concurrency in Blockchain Based Smartpool with Transactional Memory" [4]. In his thesis, Kadariya covers everything from an explanation of features and aspects of blockchain to a description of concurrent programming to provide a background for the proposal of a non-blocking concurrency mechanism for block verification within a decentralized mining pool.

Kadariya focuses primarily on Ethereum blockchain, which then led us to consider Zachary Painter, Victor Cook, Damian Dechev, and Pradheep Gayan's article "Parallel Hash-Mark-Set on the Ethereum Blockchain" [6]. This article provides a concurrent solution to the bottleneck caused by the sequential implementation of the Hash-Mark Set algorithm that allows peers to access a READ-UNCOMMITTED view of the blockchain network (that is, the network includes nodes that are not yet published). Because this article primarily looks at Ethereum blockchain, we began considering trying to implement the proposed concurrency algorithm.

We began researching Ethereum blockchain via the Ethereum Whitepapers [2]. The Whitepapers are thorough, not only providing plain language explanations of implementation, but also clear suggestions on how to implement. We noted quickly that 1) Solidity is a language made specifically for Ethereum, and thus if we actually planned to make a standalone Ethereum blockchain, we should use that language, and 2) implementing an Ethereum blockchain was a massive undertaking. We began researching other options for implementing concurrency in blockchain.

Our research led us to simple blockchain implementations in C and Java [5, 3]. Both of these implementations are academic in nature; neither is interested in a peer-to-peer network of nodes, but rather in the method of creating a blockchain via hashing and mining. Both usea Proof-of-Work consensus. While neither implement concurrent blockchain, we were able to determine which parts of the process lend themselves to concurrency.

And thus, the real work began.

4 Expectations

The Proof-of-Work consensus algorithm forces nodes to find target hash values within a limited range. The narrower the range of the hashed value, the more computationally taxing the problem becomes. Since hash values generated by the SHA-256 cryptographic hashing algorithm vary drastically between even small changes made on the input value, nodes on the network must use a brute force approach to finding the target value[5].

The brute force approach involves iteratively generating hash values by changing the input value incrementally after each failed attempt using the nonce value. This approach is almost random, as no node on the network has the ability to determine which input value will successfully generate the target hash value. This process is the most laborious process in the blockchain and the

most time consuming[3].

By increasing the number of threads concurrently generating hash values, the expectation is naturally that there will be on average a decrease in the time between successfully mined blocks. This approach to increasing the rate by which blocks are mined is analogous to purchasing multiple raffle tickets. In other words, it is likely that someone who purchased 50 tickets when entering a raffle has a probabilistic advantage when compared to another individual who only purchased one ticket when entering the same raffle. However, due to the random nature of the raffle, and similarly the Proof-of-Work algorithm, it is still possible for someone who purchased fewer tickets to win the raffle. Likewise, a node with fewer working threads could still find the target hash before a node with more working threads.

Our expectation was to observe an overall increase in the rate of successfully mined blocks by a node with more threads than a node with fewer threads. We also expected that occasionally the node with fewer threads will successfully mine a block before a node with more threads. We were especially interested to observe the relationship between the number of worker threads and the overall rate of successfully mining blocks. Furthermore, across languages, we expected concurrent implementations in C++ and Haskell to perform better than the concurrent implementation in Java, since C++ and Haskell use low-level threads. Due to challenges with Haskell concurrency that we discuss further in Methodology, we decided to switch the third language implementation to Rust. Because none of us had ever used Rust before, we did not have clear expectations. However, because Rust does not abstract via the JVM, we expected it to perform better than the Java implementation.

5 Methodology

For each language implementation, we started with the single-threaded version. Each of us had varying levels of experience with and understanding of blockchain in general, so the single-threaded implementation gave us a chance to learn the underlying structures. As noted previously, we referenced Baeldung's Java implementation [3]. Each single-threaded implementation reads lines from an input text. Each line is placed in a block, hashed, mined, and, once validated, added to a blockchain.

Each implementation captures the mining time for each block and the total time of the blockchain creation. The mining times are then used to find the average block time. The final output of each implementation is `<total time> <average block time>`, where the total time and average block times are floats rounded to six decimal places.

When we converted each single-threaded implementation to a multi-threaded version, we changed the model so that each thread functions similarly to a node in an actual blockchain implementation. When given a transaction (an unmined block with no nonce), each thread begins mining the block. However, the threads do not do duplicate work. Instead, each thread gets a nonce from an atomic

incrementer and attempts to solve each puzzle with the nonce.

For each language, we encountered challenges. Some challenges seemed insurmountable; so much so that we had to scrap the implementation and try a new language.

5.1 Haskell

We initially chose Haskell based on a vague understanding that Haskell threads are low-level and efficient. However, after we chose our model for implementing concurrency in each blockchain version, we ran into a problem with Haskell concurrency. Haskell threads, it would seem, are only for use in IO functions. Our model, however, required us to make the mining process parallel, and the mining process is mathematical. We faced an impasse; we had already spent so much time working on the single-threaded implementation and devising a model for concurrency. Should we scrap the model?

Scrapping the model was highly undesirable to the team. After some research, we decided to change our third language implementation from Haskell to Rust. The positive aspect of this choice was that we could maintain the model we were using with C++ and Java. The negative aspect was that the student working on Haskell would now need to learn Rust and implement both the single and multi-threaded versions of the blockchain in a much abbreviated time span.

5.2 Rust

Because we had already used a consistent single-threaded blockchain model across three languages, building the single-threaded version in Rust was really about learning Rust. After the single-threaded version was complete, however, we still had to determine the best way to implement concurrency.

One of the strengths of Rust–ownership–also proved to be one of the biggest challenges of multi-threading. Creating an atomic counter and accessing it from the threads was much more difficult in Rust. Also, the Rust mutex lock can easily result in a deadlock when multiple threads try to acquire it at the same time. Thus, the student realized that, unless the threads are known to all start asynchronously (something that is very hard to know), she had to use a try-lock() and a back-off method to ensure that the threads made progress in the work. Ultimately, the student who worked on the Rust blockchain notes that the implementation likely does not use Rust to its full capacity since she studied both the language and concurrency within the language in an abbreviated time frame. However, even with the comparatively slower execution times, she did use concepts from class to speed up the multi-threaded implementation (specifically with the use of Mutex locks, backoff protocol, and atomic incrementers).

5.3 C++

Applying concurrency to the C++ single-threaded blockchain was logically simple. The single-threaded block chain has a nonce variable for each block that is a part of the data to be hashed. If an attempt to mine the block does not result in a hash with the desired prefix, then the nonce is increased and the block is mined again. The multi-threaded block chain has a shared variable for the nonce. Each thread gets the nonce and increments it using a lock to prevent other threads from accessing the nonce at the same time. Once an individual thread finds the desired hash with prefix, main is alerted that a hash is found, the nonce used is saved to the block along with the hash, and outputs from all other threads are ignored. The other threads exit normally once they have finished with their current nonce.

The student's lack of understanding of how threads were implemented in C++ was a barrier to implementing the multi-threaded block chain at first. Different versions of C++ have different implementations of threads. Older versions use pthreads, which have a complex interface but can be configured to run very efficiently. The latest versions of C++ have a standard class called Thread that implements pthreads. This class provides a clear interface, but loses the same customization. Our implementation of the C++ block chain uses C++ 11, which supports the Thread class.

5.4 Java

Several attempts were made to implement a multi-threaded blockchain using the Java programming language. Firstly, an attempt was made to utilize the ExecutorService Java Development Kit (JDK) API. A pool of eight threads were instantiated and managed by the ExecutorService and then, in a for loop, a task using the Callable interface was submitted to the ExecutorService. The task implementing the Callable interface allowed for each thread to calculate a hash value based on a unique nonce value which incremented with the counter variable of the for loop. Essentially, each iteration of the for loop instantiated a task with a nonce value; submitted the instantiated task to the ExecutorService; and captured the result returned by the thread via the Callable interface using an object implementing the Future interface. If the returned value successfully mined the block (i.e., the hash value's prefix matched the prefix string) then the blocks nonce value was updated and the next block was mined in the same fashion.

Unfortunately, this first attempt did not truly allow for concurrency because each iteration of the for loop attempted to get the promised value from the Future before the next iteration of the for loop. As later discovered, this created a bottleneck where each thread executed tasks sequentially due to the blocking property of the get() method used to retrieve results from Future objects.

To overcome this hurdle an attempt to refine this implementation was made by instantiating an array list of Future objects to store hash values produced by threads. The isDone() method was used to check whether the Future objects

in the array list contained a value. If the array list did not contain a value to be checked then the for loop continued to issue tasks to threads. If the array list contained a value to be checked then the the values were inspected until a suitable value was found. If no suitable value was found then the for loop continued in this fashion until one was found.

Once again, this did not perform as well as expected. Although this time threads produced candidate values in parallel, threads also did nothing after the accumulation of a large number of values. Essentially, the time needed to check or consume all of the produced values interrupted the ability to issue tasks to idle threads.

The final implementation shifted the focus from returning values via the Callable and Future interfaces to instead updating a shared variable when a thread mined a block. Threads concurrently produced values and when a suitable value was produced the shared variable was updated such that other threads essentially discarded their results. This meant threads would continually check the status of the shared variable and created quite a bit of contention.

6 Results

Once the blockcchains across C++, Java, and Rust were finished, we each ran them with the test script. We combined our test results to produce a data set that we could then visualize with Python.

We used Python to interpret the results and visualize our data. After our initial interpretation, we could see that there was some speedup with the multi-threaded implementations for all languages on the larger prefixes. However, because the Rust and Java were slower overall (Rust in particular) and we were dealing with large times with drastic variance due to the difference in power across machines, we knew we needed a clearer interpretation of the results.

Based on our results, we could see that, indeed, C++ did perform the best overall. The times were consistently faster than Java and Rust. Also, the C++ implementation saw a sharp increase in speed with the addition of multi-threading: a 188.17% increase at prefix 4. Its highest speedup for prefix 4 occurred on 10 blocks with a speedup of 218.01%.

Java also saw an average increase, although the change was far humbler at 3.36%. The Java implementation had its highest speedup of 9.31% at prefix 4 on 10 blocks and its lowest speedup of -0.16% at prefix 4 on 1000 blocks. Java was also much slower than the C++ version to begin with. Like the Java implementation, the Rust implementation was much slower than the C++ blockchain, both single- and multi-threaded versions.

The Rust blockchain had an average increase total execution time increase of 3.07% for prefix 4. Interesting to note is that, at 1000 blocks at prefix 4, the Rust implementation had a speedup of 21.09% and at 100 blocks a speedup of 26.96%. Unfortunately, the Rust implementation had a negative speedup for 10 blocks: -38.83%.

Across block size, prefix size, and multi- or single-threaded implementation,

C++ Multi-Threaded Total Execution Time

	prefix=0	prefix=1	prefix=2	prefix=3	prefix=4
blocks=10	8.00720	8.27860	18.79320	145.21860	2055.45680
block=100	65.65780	61.04220	135.50780	1307.15080	23752.80800
blocks=1000	1016.15660	946.27660	1685.79600	14381.04000	219998.28000

Individual Times (Expand Collapsed Rows to View)

HM	9.49700	10.07600	11.55900	53.25900	814.73700
	136.33400	109.68700	115.46700	528.69300	13826.30000
	1934.00000	1456.53000	1489.88000	8092.53000	111684.00000
PB	1.49200	1.57100	6.04300	44.81100	1588.08000
	12.41200	14.85700	40.98300	617.95400	12119.50000
	215.63600	236.66600	491.70900	6321.91000	103910.00000
RP	11.61200	11.87100	37.14600	233.25700	3462.78000
	66.34900	53.98700	211.97200	2007.88000	46251.20000
	813.37600	858.99800	2239.75000	23080.10000	377081.00000
SD	9.13500	9.42800	30.27200	346.55900	3827.83000
	48.71800	59.82500	231.26700	2906.56000	39782.60000
	1360.17000	1399.48000	3320.66000	29722.60000	435971.00000
SM	8.30000	8.44700	8.94600	48.20700	583.85700
	64.47600	66.85500	77.85000	474.66700	6784.44000
	757.60100	779.70900	886.98100	4688.06000	71345.40000

C++ Multi-Threaded Avg. Block Execution Time

	prefix=0	prefix=1	prefix=2	prefix=3	prefix=4
blocks=10	0.57990	0.57852	1.54638	13.78912	165.60576
block=100	0.62640	0.58546	1.32422	12.87596	235.06830
blocks=1000	1.01258	0.94295	1.68209	14.37942	219.94070

Individual Times (Expand Collapsed Rows to View)

HM	0.80030	0.85390	0.93540	4.89670	80.71170
	1.33348	1.07502	1.13396	5.24523	137.92800
	1.92797	1.45264	1.48697	8.09198	111.48400
PB	0.11770	0.12840	0.55870	4.36390	133.77400
	0.12126	0.14480	0.40577	6.14365	119.35600
	0.21515	0.23595	0.49034	6.32519	103.96100
RP	0.55420	0.45180	2.97010	22.03810	308.18700
	0.58467	0.47592	2.04630	19.40020	456.78200
	0.80712	0.85132	2.23343	23.07230	376.72800
SD	0.73200	0.76760	2.51230	33.23800	250.48000
	0.46146	0.58102	2.27667	28.88370	393.72000
	1.35678	1.39688	3.31502	29.72240	436.12700
SM	0.69530	0.69090	0.75540	4.40890	54.87610
	0.63112	0.65055	0.75839	4.70701	67.55550
	0.75586	0.77798	0.88469	4.68521	71.40350

C++ Single-Threaded Total Execution Time

	prefix=0	prefix=1	prefix=2	prefix=3	prefix=4
blocks=10	1.68060	3.54560	29.00660	544.89500	6536.62007
block=100	6.95000	24.94300	256.75200	3825.45615	60463.06250
blocks=1000	325.35919	414.41940	2717.83662	39884.70234	642280.96563

Individual Times (Expand Collapsed Rows to View)

HM	1.03900	1.02000	16.83200	219.07600	4120.01611
	5.62600	17.47100	148.23801	2026.75098	35270.83594
	463.46399	296.76999	1522.09399	23117.29883	320211.18750
PB	0.15800	1.07000	15.00900	257.64499	2199.20093
	1.95500	7.62400	98.53100	1472.99194	21134.38867
	113.97600	157.22301	1051.88501	15238.80859	254539.10938
RP	5.52500	8.94900	43.21000	813.64301	7906.00586
	14.82700	38.00900	415.26901	4495.05908	104305.35938
	394.79001	612.57098	4199.88818	61131.49609	987114.25000
SD	1.20500	5.85700	56.90500	1225.94202	14413.48047
	9.72000	51.77600	485.40799	9430.09180	110897.51563
	533.59497	819.25403	5538.60693	81104.32031	##############
SM	0.47600	0.83200	13.07700	208.16901	4044.39697
	2.62200	9.83500	136.31400	1702.38696	30707.21289
	120.97100	186.27901	1276.70898	18831.58789	313135.53125

C++ Single-Threaded Avg. Block Execution Time

	prefix=0	prefix=1	prefix=2	prefix=3	prefix=4
blocks=10	0.03392	0.18756	2.60994	51.86666	602.24947
block=100	0.05040	0.23305	2.53190	38.08384	595.61577
blocks=1000	0.32294	0.41182	2.71400	39.90899	642.14207

Individual Times (Expand Collapsed Rows to View)

HM	0.01770	0.06130	1.60510	21.61490	372.10980
	0.04672	0.16593	1.46728	19.99582	347.90350
	0.45862	0.29461	1.52142	23.13460	319.72211
PB	0.01090	0.09330	1.48770	22.69360	219.59891
	0.01885	0.07544	0.95799	14.72062	207.31525
	0.11381	0.15711	1.05246	15.25044	254.67986
RP	0.03810	0.22220	3.44550	76.61080	787.67883
	0.07071	0.31768	4.04745	44.40312	1025.98059
	0.38951	0.60367	4.18384	61.16800	986.29999
SD	0.08920	0.51460	5.24250	117.87399	1273.36231
	0.09391	0.51213	4.83267	94.28694	1093.79639
	0.53304	0.81883	5.54096	81.18067	1336.63965
SM	0.01370	0.04640	1.26890	20.54000	358.49750
	0.02179	0.09408	1.35412	17.01268	303.08313
	0.11972	0.18490	1.27132	18.81123	313.36877

Figure 1: The preliminary results of our tests–the first visualization with Python. We created three of these: one for C++, one for Java, and one for Rust.

C++ invariably lead. Although we initially predicted that this would be a race between Haskell and C++, given that our languages changed, C++ was our expected winner. Java and Rust both had a set number of threads, which we believed would contribute to slower increase from a multi-threaded implementation.

In addition, we expected the Java implementation to be slower overall, given that the language is abstracted from the OS via the JVM. Thus, while we suspect there are ways to speed up the Java implementation further, we also understand the limitations of an abstracted language, especially where concurrency is concerned.

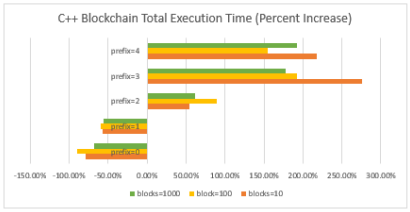
Rust, on the other hand, is lauded as a fast, efficient, and safe language for single- and multi-threaded programs. We believe that the drastically slower times arise not from the limitations of Rust but rather from our limited experience of Rust synchronization mechanisms. Thus, we feel certain that with more research, we could implement a faster, more efficient version of Rust, both multi- and single-threaded.

C++ Blockchain Total Execution Time (Percent Increase)

	prefix=0	prefix=1	prefix=2	prefix=3	prefix=4
blocks=10	-79.01%	-57.17%	54.35%	275.22%	218.01%
block=100	-89.41%	-59.14%	89.47%	192.66%	154.55%
blocks=1000	-67.98%	-56.21%	61.22%	177.34%	191.95%
Average	-78.80%	-57.50%	68.35%	215.07%	188.17%

C++ Blockchain Avg. Block Execution Time (Percent Increase)

	prefix=0	prefix=1	prefix=2	prefix=3	prefix=4
blocks=10	-94.15%	-67.58%	68.78%	276.14%	263.66%
block=100	-91.95%	-60.19%	91.20%	195.77%	153.38%
blocks=1000	-68.11%	-56.33%	61.35%	177.54%	191.96%
Average	-84.74%	-61.37%	73.77%	216.49%	203.00%



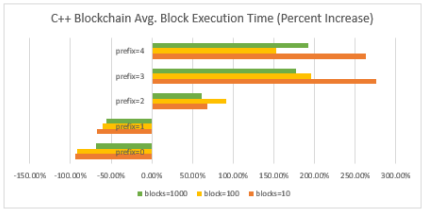


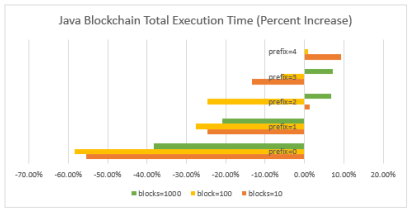
Figure 2: The percentage increase of the C++ multi-threaded implementation (compared to the single-threaded implementation)

Java Blockchain Total Execution Time (Percent Increase)

	prefix=0	prefix=1	prefix=2	prefix=3	prefix=4
blocks=10	-55.43%	-24.71%	1.38%	-13.34%	9.31%
block=100	-58.45%	-27.68%	-24.58%	-5.76%	0.92%
blocks=1000	-38.39%	-20.92%	6.78%	7.12%	-0.16%
Average	-50.76%	-24.44%	-5.47%	-3.99%	3.36%

Java Blockchain Avg. Block Execution Time (Percent Increase)

	prefix=0	prefix=1	prefix=2	prefix=3	prefix=4
blocks=10	-55.43%	-24.71%	1.38%	-13.34%	9.31%
block=100	-58.45%	-27.68%	-24.58%	-5.76%	0.92%
blocks=1000	-38.39%	-20.92%	6.78%	7.12%	-0.16%
Average	-50.76%	-24.44%	-5.47%	-3.99%	3.36%



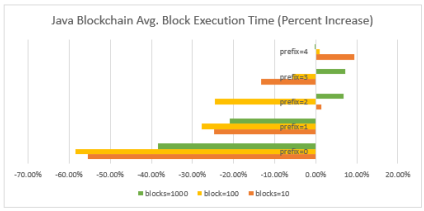


Figure 3: The percentage increase of the Java multi-threaded implementation (compared to the single-threaded implementation)

7 Conclusion

The overhead involved in orchestrating and monitoring the threads is expensive, to a degree. It is not surprising, then, that at the lower prefixes the single-threaded versions of the blockchain performed better or the same as the multi-threaded versions. However, as the prefixes get larger, we did see an overall decrease in the average time per block and total execution time.

Some languages performed phenomenally. C++ scored high marks on speed up, and also saw faster speeds for both single and multi-threaded implementations compared to Java and Rust. But although Java and Rust were slower than C++, they also saw improvement in speed as the prefix got higher. In an actual Bitcoin (or similar) blockchain, speeding up the mining process would not really help the miners, since the algorithm attempts to average out the mining times to 10 minutes. However, for our purposes, adding concurrency to the blockchain helped us to test, prove, and refute our preconceptions of the implementation and limitation of concurrency across Rust, C++, Java, and Haskell.

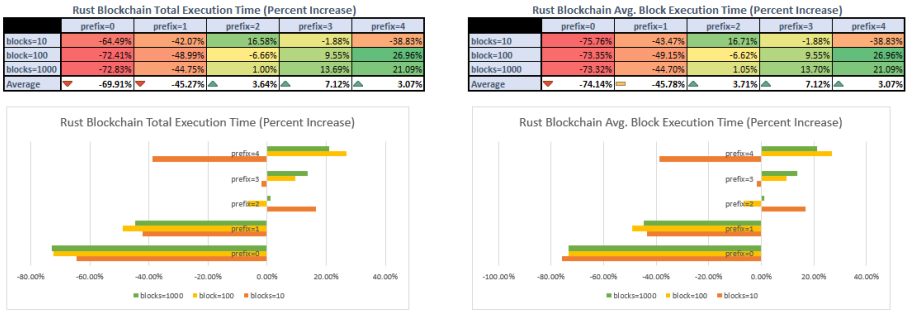


Figure 4: The percentage increase of the Rust multi-threaded implementation (compared to the single-threaded implementation)

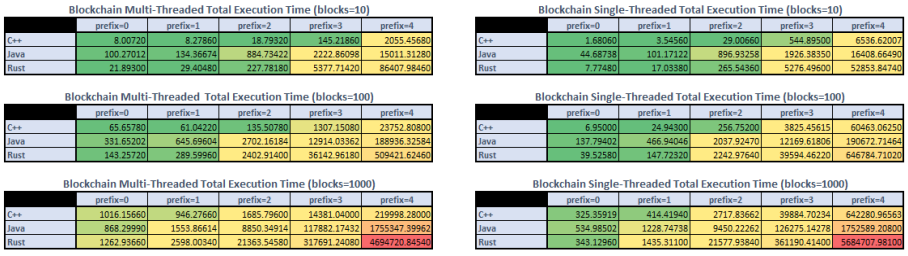


Figure 5: A comparison across block sizes and prefixes of the C++, Java, and Rust implementations

There are multiple ways to extend this project. Anyone who is interested in applying what we have learned and adding to that knowledge might choose an extension from the following list:

1. Conduct further research on Java and Rust concurrency to find the fastest concurrent implementation for this blockchain model. Challenge: Make the Java and Rust times comparable to the C++ version.

2. Use concurrency to simulate a p2p network, with each thread functioning as a node which mines blocks and validates blocks mined by other threads.

We invite future students to use one or both of these suggestions to research and implement concurrency in later projects.

References

[1] Seyed Mojtaba Hosseini Bamakan, Amirhossein Motavali, and Alireza Babaei Bondarti. A survey of blockchain consensus algorithms performance evaluation criteria. *Expert Systems with Applications*, 154:113–385, September 2020.

[2] Vitalik Buterin. Ethereum whitepaper, 2013.

[3] Kumar Ch and rakant. Implementing a Simple Blockchain in Java | Baeldung, September 2019.

[4] L Kadariya. Concurrency in Blockchain Based Smartpool with Transactional memory. 2018.

[5] J Meiners. Write your Own Proof-of-Work Blockchain.

[6] Z. Painter, P. K. Gayam, V. Cook, and D. Dechev. Parallel Hash-Mark-Set on the Ethereum Blockchain. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–5, May 2020.