Blockchain System for Scalability Study

Aaron Mills 260849570
Aidan Jackson 260924686
Sami Ait Ouahmane 260908610
Roey Wine 260848818

ECSE 458 - DP 09

Project Advisor: Professor Christophe Dubach

April 2023

Abstract

Blockchain technology allows for secure transactions between two parties without the need for additional costs, security concerns, and lack of privacy associated with an overseeing institution. With its growing popularity, blockchain systems' safety, reliability, and scalability are increasingly important.

The motivation for this project stems from the misnomer that blockchain systems are not scalable. It analyzes the scalability of blockchain systems with respect to computational resources by discussing the results of various performance tests on a custom, minimalist blockchain system.

The system in question uses cryptographically verifiable transactions, blocks, and the ability to create nodes (peers) in a decentralized network. Nodes can connect to others in the network, transmit transactions, and create blocks. Before the network was part of the system, there was a simulation to mock a larger network, including generators that periodically created transactions and blocks.

The study investigated the performance of fundamental operations in this blockchain system, including verifying transactions and validating blocks. In addition, it analyzed the impact of a network on the system and used parallelism to improve performance.

Overall, the study has shown that the minimalist blockchain system is scalable. This was done by testing the system using a suite of performance tests to verify that the fundamental operations achieve linear or sub-linear performance with respect to system size.

Contents

Notation

The following list describes acronyms and short forms of words or phrases used in the text of the paper and in the diagrams contained within the paper.

- Unspent Transaction Output Data Structure: UTXO
- Transaction: Tx
- Hash of transaction n: hashn
- Hash of the hash of transaction n concatenated with the hash of transaction m: hashnm
- Transaction input: TxIn
- Transaction output: TxOut
- Transaction identifier: TxId
- Most recently added block to the blockchain : Head
- Local Area Network: LAN
- Transmission Control Protocol: TCP
- User Datagram Protocol: UDP

Chapter 1

Introduction

1.1 Why Study Blockchain?

Throughout the world, millions of transactions occur daily. The ability to securely and reliably perform transactions at a low cost is fundamental to society. In traditional transaction systems, a central authority facilitates the transaction, ensures that the parties possess the currency they claim to own, settles any disputes, and prevents double spending (spending the same currency on multiple transactions). However, the fact that this third party is necessary for transactions brings about additional costs, security concerns, and an inherent lack of privacy.

Blockchain technology provides a decentralized electronic cash transfer system that solves the double spending problem without needing a third party. This is significant, as the lack of an overseeing financial institution means that transactions are both cheaper and can be private. In addition, blockchain systems are designed to be secure when a majority of the computing power contributing to the system is from honest peers, which is practically the case for any scaled system.

This project aims to study the scalability of blockchain systems with respect to CPU computation. The intention is to investigate the misnomer that blockchain systems are not scalable. Scalability means the system's functions are expected to operate with linear or sub-linear time complexities. Testing was done on a custom minimalist system created for this project. The analysis is from the perspective of the peer (a user in the network) interacting with the surrounding network. This means only the throughput of operations and functions internal to the peer are considered, not other factors such as network latency.

1.1.1 Applications of Blockchain

There are several applications of blockchain systems. One example is its use in supply chain transparency by maintaining an immutable, distributed ledger containing information about the supply chain. Another example is the use of blockchain in Internet of Things (IoT) systems. IoT devices can use a blockchain to share data with one another without the need for central control. The main application explored in this project is cryptocurrency, described in detail in the next section.

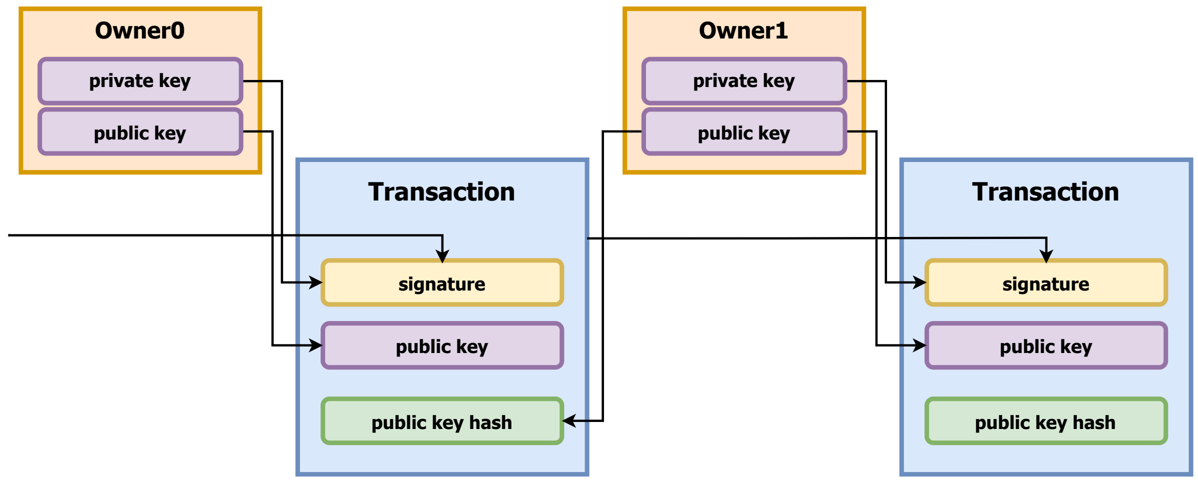


Figure 1.1: Diagram demonstrating the cryptographic keys provided by each party in creating a transaction. ‘Owner0’ sends a transaction to ‘Owner1’ who then creates a new transaction.

1.2 Background

1.2.1 What is Bitcoin?

Bitcoin is a peer-to-peer electronic cash system that first demonstrated the power of blockchain technology in a paper released by Satoshi Nakamoto in 2008 [3]. The system designed in this project is a much-simplified version of a blockchain based on the paradigm presented by Bitcoin.

1.2.2 What are Transactions?

The role of transactions in the system is to represent some form of value and implement the ability for this value to be transferred. In other words, they are the entity that describes a transfer of value from one party to multiple others. In a blockchain system, control over the value described in transactions is dictated by cryptography. Possessing a cryptographic key associated with some transaction value allows the creation of a new transaction that transfers that value.

The sender (existing controller of some value) must obtain a public key from each receiving party to create a new transaction. Each receiving party will create a new public-private key pair and send the public key to the sender. The sender will then create a script for each of these public keys. This script ensures that only the person with the private key associated with the public key used to create the script will control the corresponding value [4]. Once the transaction is created, the sender broadcasts it over the network. The process is depicted in Figure 1.1.

In reality, transactions comprise multiple ‘transaction inputs’ and multiple ‘transaction outputs’. The transaction outputs contain the value, and the transaction inputs refer to transaction outputs of previous transactions whose value is to be spent.

Unique to blockchain systems, transactions must be included in a structure called a ‘block’. To confirm the transaction, its block must be included in the official history of valid blocks (the blockchain). Blocks will be discussed further in Section 1.2.4.

1.2.3 An Important Data Structure: The UTXO

The Unspent Transaction Output, or UTXO, is a data structure that stores the detail about all of the value in the system that can be spent on future transactions in the form of transaction outputs.

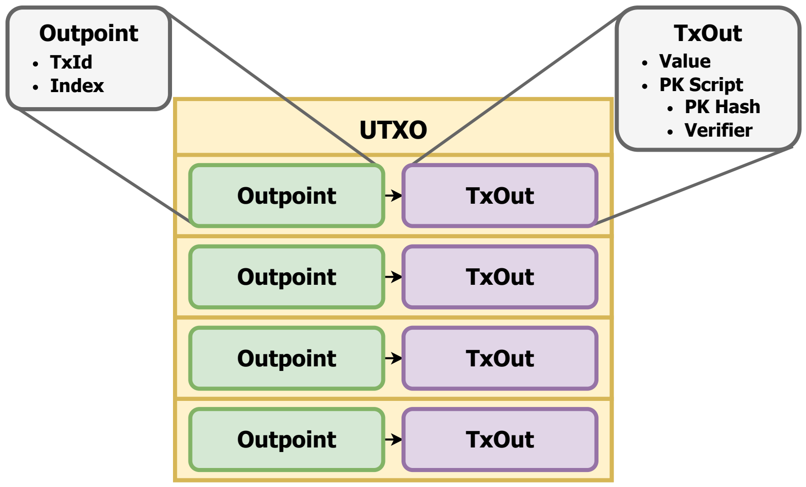


Figure 1.2: Diagram showing the content of the UTXO data structure. It shows a mapping from outpoints (which will be in future TxIns) to unspent TxOuts that can be used for future transactions.

Transaction inputs have an internal structure called an outpoint that specifies exactly which past transaction output it refers to, as seen in Figure 1.2. The UTXO is fundamental to creating new transactions and verifying the system's integrity. In particular, the UTXO keeps track of whether an output has been spent and updates by removing the spent outputs and adding the new unspent outputs when a transaction is created. The UTXO contains all the data needed for creation and verification to be readily available.

1.2.4 What are Blocks?

Aside from transactions, blocks are the most important component of blockchain systems. The role of blocks in the system is to collect broadcasted transactions and create a single entity that can then be added to the blockchain. When a block gets added as the new head of the blockchain, all the transactions inside the block are then 'confirmed' as part of the ledger. In addition to the transactions, each block contains a 'block header' with important metadata. In particular, it includes a 'hash' of the previous block in the chain (creating a link) and the 'Merkle root' which will be discussed in section 1.2.5. A 'hash' is produced by a hashing function that takes some object as input and returns some fixed-length combination of bytes as output, called the hash. The result of a hash function should be a completely random unique identifier. Blocks containing the block's hash that came before it forms an ordering. If an earlier block is tampered with and modified, its hash changes. This change will propagate forward, adjusting the hash of successive blocks and making it clear to everyone in the network that the chain has been tampered with. The structure of a block can be seen in Figure 1.3.

1.2.5 Understanding the Merkle Tree

The Merkle tree is a binary tree composed of hashes, as seen in Figure 1.4. The leaf nodes are the hashes of a given set of transactions. The rest of the tree is formed by taking two nodes in the layer below and hashing their concatenation. This is done for each layer until just a single hash remains. This final hash, called the Merkle root, uniquely identifies the transactions used to create the tree (i.e., any other combination of transactions would yield a different Merkle root).

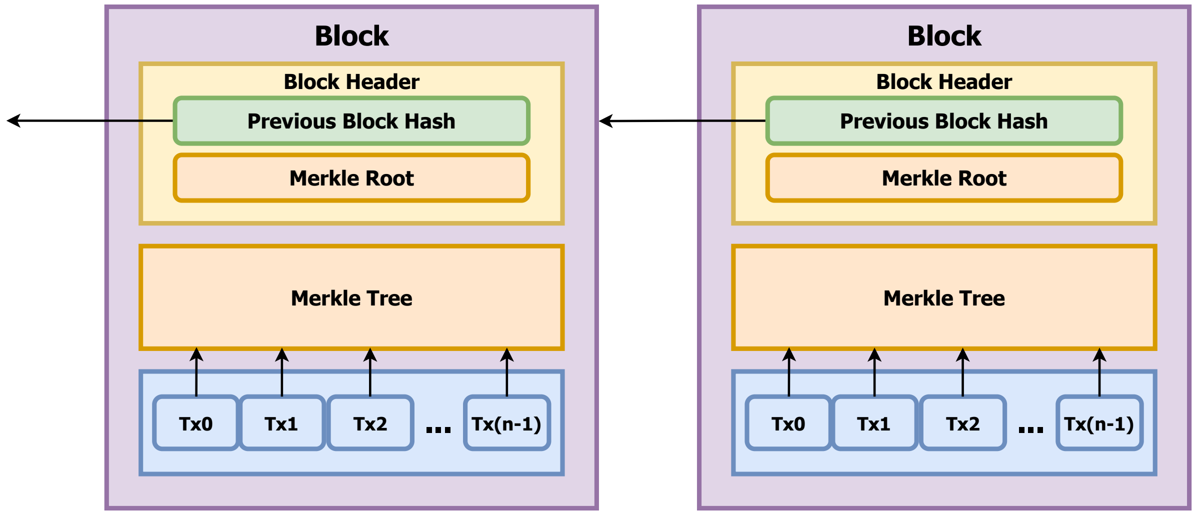


Figure 1.3: Diagram showing the composition of blocks, including the block header, Merkle tree, and transactions. In addition, it shows the link between successive transactions with the hash of the previous block included in a block's header.

The Merkle root enables constant time comparison between two blocks by comparing the header content. The Merkle tree can also efficiently verify that a transaction is in a specific tree and, therefore, in a specific block.

1.2.6 What are Peer-to-Peer Networks?

While the implementation of the system has focused on the internal operations of the peers, in reality, the peers are part of a larger network. Peers in the blockchain system must communicate in a decentralized manner meaning that the classic 'client-server' architecture cannot be used. Instead, a 'peer-to-peer' (P2P) networking architecture must be used. In P2P networks, peers act as both servers and clients, simultaneously requesting services for themselves and fulfilling services for other peers [5].

The motivation for implementing a P2P network is to obtain a more accurate understanding of the bottlenecks of blockchain systems, more precisely, whether the peer's functionality or current networking capabilities act as the bottleneck.

1.2.7 How Can Parallelization be Utilized for Increased Performance?

Parallelization has been utilized in multiple critical sections of the system to achieve better performance. The attainable theoretical performance improvement can be characterized by Amdahl's law, which conveys the theoretical speedup of a program given the parallelizable fraction of the sequential program.

$$speedup = \frac{1}{(1-F)+\frac{F}{n}} \tag{1.1}$$

In the equation above, F is the fraction of the program that can be parallelized (a number between 0 and 1), and n is the number of processors or threads used in parallelization [7]. This equation gives a theoretical upper bound for speedup results and will be compared with the empirical results obtained in section 3.2.3.

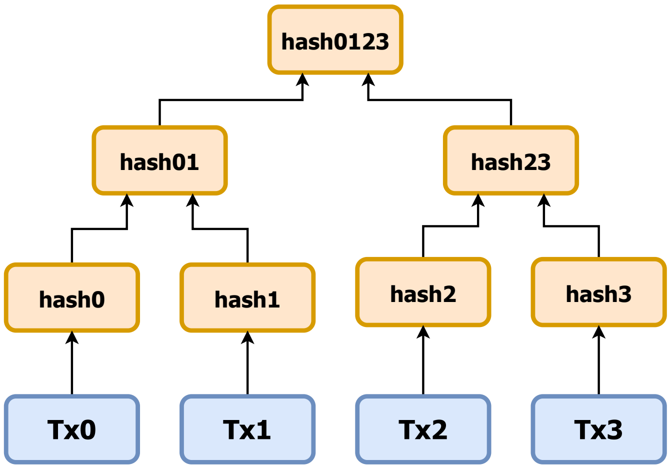


Figure 1.4: Diagram showing the structure of a Merkle tree for a collection of 4 transactions. The transactions are first hashed, and then the tree is built bottom-up.

1.3 Requirements

1.3.1 Scalability of the System

This project aims to study the scalability of blockchain systems, but what does it mean for such a system to be scalable? The key resources involved in the scalability of software systems include computational complexity (speed), space complexity (storage), and power consumption. This project focuses on scalability with respect to computation resources (CPU) rather than storage or power.

For a decentralized system such as blockchain to be scalable, the fundamental functions that the system performs are required to operate asymptotically better than or equivalent to a linear function (i.e., $O(n)$) as the network grows. When new peers join the network, they bring along more computational resources in addition to more computational requirements. The specific functionality of interest is the transaction throughput (transactions verified per second) from the peer's perspective. In blockchain systems, the ability of the network's peers to process transactions and verify their validity is commonplace and of great importance to the system's success. If this is not scalable, the system becomes unusable as more peers join the network. This differs from transaction confirmation since a transaction is only confirmed after it is sealed in a block, and that block has had a chain built on top of it.

1.3.2 Functionality of the System

For transactions to be functional, the system is expected to determine whether transactions are valid given a system state (i.e., the UTXO). When a valid transaction is created, the system is expected to update its state by updating the UTXO. When an invalid transaction is provided, the system is expected to determine that the transaction is invalid, declare why it is invalid, and should not update the system state based on the information contained within that transaction. With the introduction of networking, the system is expected to be able to broadcast transactions between peers on the network and effectively handle incoming transactions from the network. When transactions are received from the network, they are buffered before being processed. If this buffer continues to grow due to the peer's inability to process the transactions fast enough, then the peer is deemed ineffective at handling incoming transactions. On the surface, this seems like a speed issue.

Still, in reality, it is a functionality issue because if the buffer continues to grow, the time between transaction creation and transaction verification becomes larger over time, which is impractical.

For blocks to be functional, the creation of blocks is required to include the following: the collection of transactions for the block must be formed into a Merkle tree, the block header must be created containing the hash of the block at the head of the chain and the Merkle root, and each of the transactions needs to be verified. Once these stages have passed, the block can be created and broadcast to the system. With the introduction of the network, the mining peers need to verify incoming transactions from the network effectively, form them into blocks, and broadcast these blocks to the network so the blockchain can be built. As mentioned above, transaction processing must be fast enough to stabilize the transaction buffer. Similarly, the creation of blocks must also be fast enough.

1.3.3 Constraints

There are fewer physical constraints for this project than one involving a hardware product. Nevertheless, several constraints still restricted the project in many different areas. Some constraints were obvious in a project, such as a lack of time. The team only had eight months to produce a high-quality minimalist blockchain system and test as many computation time-related features as possible. All the team members were taking other classes during this project, so the workload from those classes prevented spending larger amounts of time on the project. Other constraints more specific to this project include the limited number of machines available to the team members. Testing the scalability of a network of machines with respect to computational time should ideally include using a large number of machines. There was a limitation since the team owned just five machines collectively and did not implement LAN-LAN communication. Organizing other students to assist in testing was unreasonable as it would require everyone to be at the same location for a long period of time.

Chapter 2

Design

2.1 System Overview

A control flow diagram describing the system startup procedure can be seen in Figure 2.2.

2.1.1 Selecting a Programming Language

The first significant design decision was to select a programming language for development. Since the goal was to study the scalability of blockchain systems, the speed of the language was a crucial factor. Some high-performing languages, including C and C++, were considered, but there was concern that working with C/C++ may lead to unnecessary issues due to manual memory allocation and learning difficulty. Python was also considered, given its ease of use, the massive collection of available libraries, and the entire team already has experience with it. It was not chosen due to its poor performance compared to other languages.

Ultimately, Rust was selected as the programming language for this project. Rust is a system-level language emphasizing speed, memory safety, and concurrency handling. In Rust, the programmer does not manually allocate and free memory, and there is no garbage collection. Instead, the programmer must abide by certain rules while programming, ensuring a memory-safe program.

2.1.2 The Importance of an Iterative Approach

An iterative approach to designing, implementing, and documenting all parts of the system was followed throughout the project. The decision to follow this methodology was made at the very beginning of the project. Given the time constraints, this approach has been core to creating a functional system. In particular, it has allowed for efficient molding of the development schedule around the hectic schedules of taking full course loads. Two key components of this approach were splitting tasks into small components to allow for better distribution among the team and trying to limit dependencies between different developers to avoid the situation where one team member is waiting for another.

2.1.3 The Interactive Shell

Early into the development, it became clear that there needed to be an easy way to interact with the system, namely a shell. This feature allows for easier testing and a simple user interface. From the interactive shell, the user can start a software simulation, save the system's state to a JSON file,

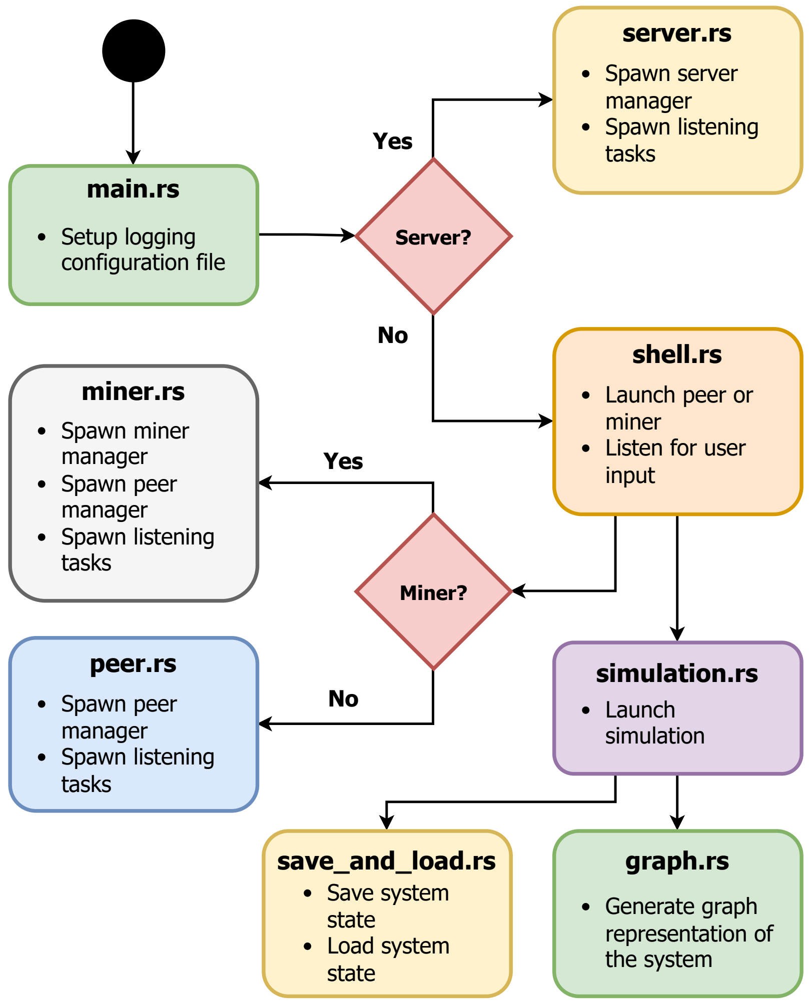


Figure 2.1: Control flow diagram for the system upon startup.

create a visual graph of the current blockchain (an example is shown in Appendix A), broadcast a transaction, and more.

2.1.4 The Design of Transactions

Transactions are a fundamental data structure representing value and facilitating its transfer. Internally, a transaction is subdivided into Transaction Inputs (TxIns) and Transaction Outputs (TxOuts). TxIns have an attribute called an outpoint that links it to the TxOut of a previous transaction and a signature script. This connection is shown in Figure 2.3. TxOuts contain a value representing its worth in the system and a public key script that, combined with the signature script, forms cryptographic proof to verify that the creator of the new transaction indeed possesses the authority to spend the value in the TxOut. The ownership of a TxOut is specified at creation time in the form of a public-private key pair. Only the holder of that private key has the ability to use that output (i.e., spend the value). Verifying a transaction is the most fundamental operation of the blockchain system. Much effort was spent improving the transaction verification performance throughput of the system.

Three requirements must hold for the transaction to be considered valid. First, a transaction's

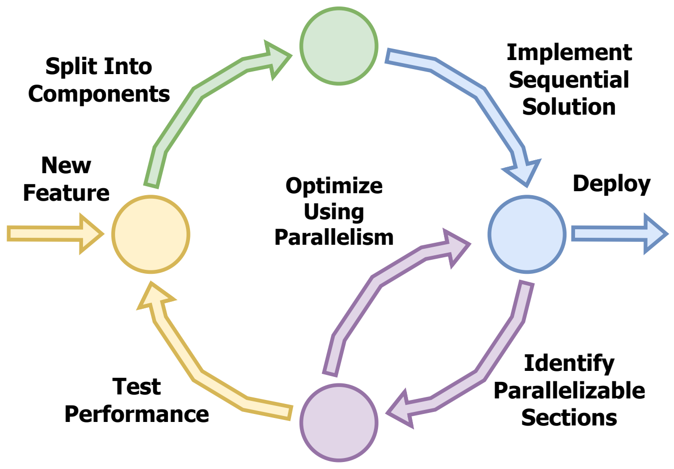

Split Into Components
Implement Sequential Solution
New Feature
Optimize Using Parallelism
Deploy
Identify Parallelizable Sections
Test Performance

Figure 2.2: Design cycle diagram depicting the procedure for implementing new features. Features are first split into smaller components and implemented as fast as possible. Performance improvement through parallelization or other means is then considered, and the implementation cycle is repeated with the new design.

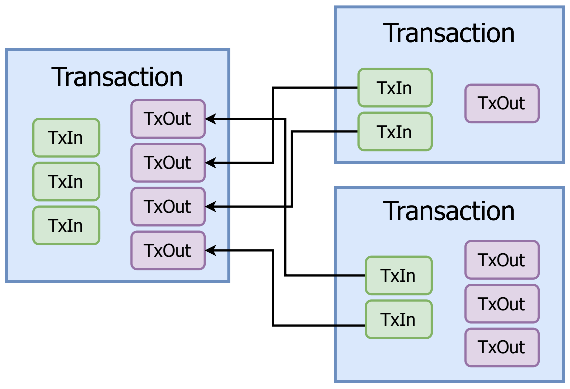


Figure 2.3: Diagram showing the breakdown of transactions into inputs and outputs and an example of the interconnection between transactions.

TxIns must point to existing and unspent TxOuts of previous transactions. This includes inputs within the same transaction. No two TxIns can refer to the same unspent TxOut from a previous transaction (called double-spending). Keeping track of the system's state in this regard is the job of the UTXO data structure. Second, the sum of the value in the new TxOuts must be less than or equal to the sum in the TxOuts that the TxIns point to. This condition ensures that the value spent must come from somewhere (a previous transaction). Effectively, value conservation needs to be ensured in the system. Lastly, the data in a TxIn's signature script must be valid according to the corresponding TxOut's public key script. This is called signature verification and proves cryptographically that the creator of the transaction is the holder of the private key associated with that TxOut, and thus has the authority to spend its value. Out of these three steps, signature verification was found to be the most expensive computationally, taking up 62% of the total verification time. Therefore, the focus has been on making this step more efficient to increase performance.

Implementing the UTXO

The Unspent Transaction Output (UTXO) data structure allows for transaction verification and facilitates the updating of the system when a transaction occurs. To do so, the UTXO maintains a structure containing all the unspent TxOuts in the system. It must be quick to access and inexpensive to expand. Based on these requirements, a HashMap data structure is a simple yet effective way to obtain fast, amortized O(1) lookup (assuming it is properly designed) and extend to large sizes. In the chosen HashMap implementation, collisions are resolved using open addressing with quadratic probing based on the C++ 'Swisstable' hash map implementation [8]. Resolution through open addressing tends to provide faster lookup time than resolution through chaining but has the downside of a hash map of a given size being limited in the number of entries it can hold [9]. In the case of the HashMap having no available slots, the only solution is to extend the table size.

When a transaction is created, the spent TxOuts are removed from the UTXO, and those created are added. Peers in the network validate the system's state by continuously updating the UTXO and ensuring new transactions meet the requirements of transaction verification.

Creating New Transactions

A sequence of steps and multiple parties are involved in creating a new transaction. First, the sender collects the unspent transaction outputs (TxOuts) they control to transfer their value to the receivers. The sender then creates and signs the messages from the previous transaction's content to create the signature scripts. These prove they control the selected unspent TxOuts and can thus spend them. Once the outpoints that refer to these unspent TxOuts are created, the transaction inputs (TxIns) are established. Then, the sender must obtain the hash of public keys controlled by the receivers of the new TxOuts (there is a one-to-one relationship with the public key hash to TxOut). Possessing the private key associated with the public key hash embedded in the newly created TxOuts will allow that person to spend its value in the future. After creating the transaction, the sender can broadcast it to the network.

'Signing' Transactions and Verifying Signatures

Cryptographically signing the content of transactions and verifying who signed the transaction is essential for the system's integrity. In the system, the message that is signed is the concatenation of the transaction hash and output index of the previous transaction and the public key hash of the new controller of the value. This message is hashed using the SHA256 hash function and signed using the private key. The signer can be verified with the associated public key. The verification process results in a boolean value that indicates whether the public key is the correct one associated with the private key used to sign the message.

2.1.5 Block Composition

Blocks are another fundamental data structure in the blockchain system created for this study. They house three main components: the block header, Merkle tree, and transactions. Many transactions per block are needed for a scalable system. Since the information about the ledger needs to be propagated across the whole network, it would be impractical to do this on a transaction-by-transaction basis.

Designing the Merkle Tree

When designing the Merkle tree, a decision that needed to be made was whether to use a pointer-based or array-based implementation. In general, pointer-based implementations are easier to work with. However, a more space-efficient array-based implementation was chosen since the Merkle trees are of a fixed size and need not be modified. Since all transactions need to be hashed to create the Merkle tree, this is done in parallel to save time.

Creating New Blocks

To create a new block in the system, a miner collects a list of unconfirmed and unverified transactions. First, the set of transactions is fully verified using parallelized verification (discussed in section 2.3) to ensure the transactions are valid. Second, the Merkle tree is built using parallelized techniques. The Merkle root is then copied into the block's header, and the block's hash currently at the head of the chain is added as the previous hash. The block is then propagated across the network.

2.2 Networking

Significant effort in the project revolved around researching networking theory, discussing different potential techniques, implementing a chosen design, and iterating to improve performance.

The first major decision that needed to be made regarding networking was whether to use TCP or UDP for communication. The key differences between the two are that with TCP, a connection between the two devices is explicitly set up before data transmission occurs, meaning that delivery of the data is guaranteed. UDP provides a 'best-effort' approach, meaning data is sent without an explicit connection and the sender receives no information on whether the data was properly delivered.

Both TCP and UDP are viable options for P2P networks, but TCP was chosen for the system due to the reliability and service guarantees that UDP does not provide. This comes at the cost of the time needed to create the explicit connection between two peers [6].

2.2.1 What is a Peer Server? The Peer 'Launch' Process

To facilitate P2P, peers need to be informed about the IP addresses and listening ports of other peers on the network (i.e., their neighbours). The 'peer servers' are special peers on the network that allow new peers to discover neighbouring peers and provide them with unique identifiers. Besides distributing and maintaining peer IDs, a peer server also maintains tables mapping IDs to their IP addresses and IP addresses to the listening ports. The network's peers can find a peer server through (deliberately) hard-coded sockets in the software.

Suppose a user boots up the system for the first time. The only peer in the network they know is some peer server. First, they contact the peer server requesting a unique identifier, which is promptly returned. Then they request information about neighbouring nodes. In this request, the peer includes information about its own listening ports so that the peer server can update its tables. The peer server will send back tables containing the contact information of neighbouring peers. Upon reception, the peer updates its personal tables, and then the peer information is persisted.

When the user boots up the system later, the existing peer information is loaded, and then the request for active neighbours is sent to the peer server.

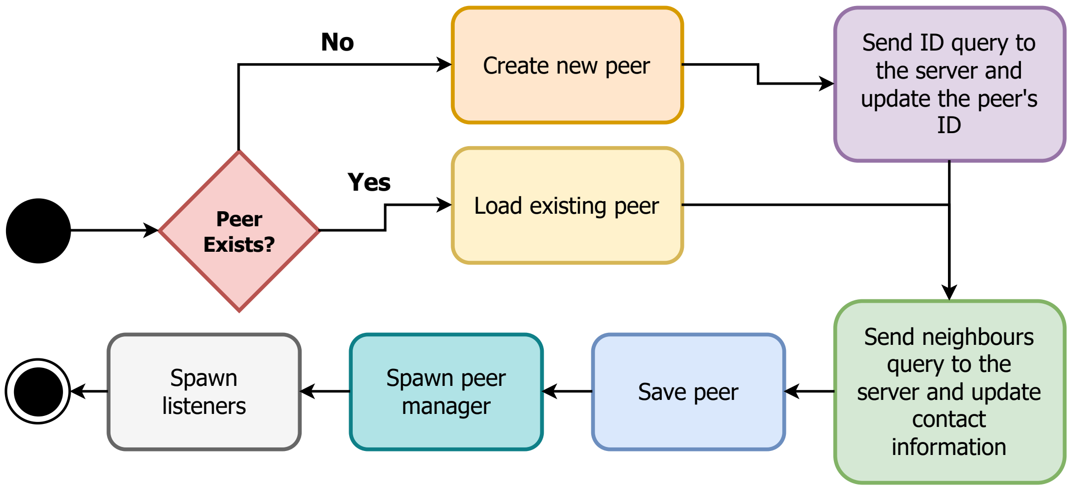

No
Yes
Peer Exists?
Create new peer
Load existing peer
Send ID query to the server and update the peer's ID
Send neighbours query to the server and update contact information
Save peer
Spawn peer manager
Spawn listeners

Figure 2.4: Control flow diagram describing the launch procedure for a peer.

The final steps in the process are spawning the ‘peer manager’, listening on the ports that it informed the peer server it would be listening on, and returning control to the main thread so that the user can input commands. A control flow diagram depicting the entire process can be seen in Figure 2.4. The peer manager processes all the incoming requests to the peer, whether they involve reading or modifying peer information. These requests can come from different threads simultaneously. The peer manager ensures that there are no race conditions (when simultaneous access of shared resources causes incorrectness) nor propagation of incorrect or incomplete data. The managers are further discussed in section 2.2.2.

2.2.2 Managers: Ensuring the Integrity of the System

Each of the three types of peers: the regular peer, the peer server, and the mining peer have their own ‘managers’ that handles requests. Some of these requests come from the listeners through queries from other peers, and some come from the user through the shell. For a thread to access the peer to read or modify, it must send the appropriate command to the manager. The list of commands supported by the managers can be found in Appendix B.

The manager for the server is the most limited. It is only able to respond to an ID query or a neighbours query. The regular and mining peer managers have much more functionality and are similar to one another. The mining peer runs two managers, one regular peer manager and one miner manager, to handle block creation. The miner manager forwards commands to the regular manager for all other requests.

2.2.3 How are Incoming Transactions Handled?

This section will overview how new transactions are processed when a peer receives them over the network. This process is slightly different for a mining peer.

To process incoming transactions effectively, a pipeline was created with three stages, as seen in Figure 2.5. The first stage involves collecting the transactions, the second is verifying them, and the third is updating the data structures.

Upon reception, transactions are stored in a temporary buffer in preparation for processing. Eventually, this buffer will fill beyond a threshold, and verification is invoked. The buffer is emptied, and a new thread is spawned to verify the transactions. To prevent any integrity issues with

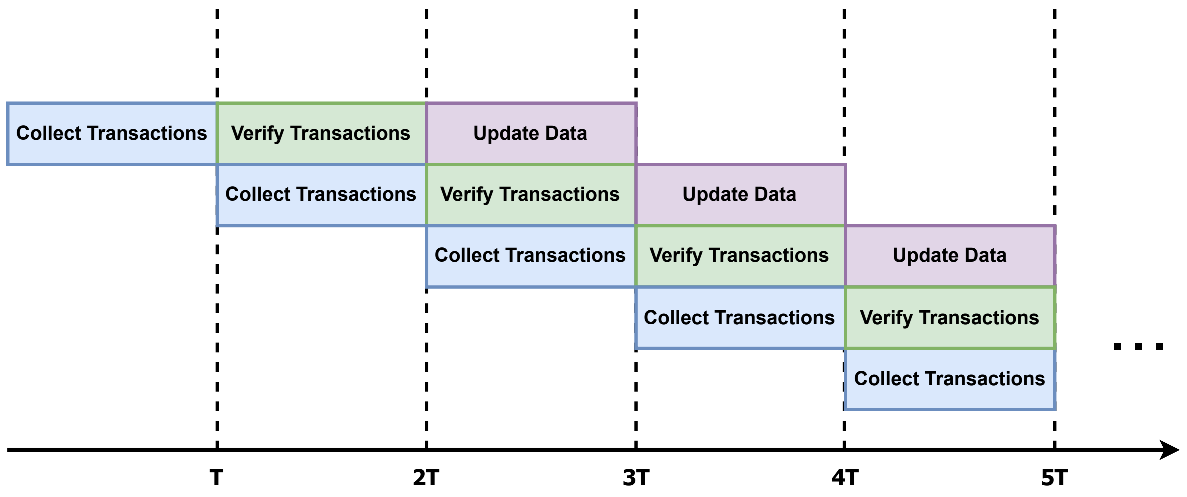


Figure 2.5: Diagram depicting the pipeline for handling transactions the peer receives from the network where 'T' is the time it takes to complete the longest stage.

shared resources, mutexes are used. While the verification thread runs, the main thread returns to collecting transactions into the buffer. After the verification, another thread is spawned to update the data structure that holds the verified transactions.

If the buffer reaches its limit before the verification thread is ready for another batch, the buffer simply continues to fill, and the verification thread only spawns when it is ready.

The mining peer has a similar process, but it is slightly different since it uses the transactions to create blocks. Instead of only verifying the transactions in the second stage, it goes through the entire process of creating a new block. Then, the thread for the third stage is spawned, where the miner updates the verified transactions and broadcasts the block.

2.3 Where was Parallelization Utilized?

There are two locations where parallelization was utilized to obtain very large performance increases. These were chosen after a manual profiling of functions in the system that indicated that a large portion of the CPU time was spent in sections that are parallelizable (this profiling is shown in Section 3.2.2). These are the portions where the largest speedup can be achieved according to Amdahl's law. The first and most important of the two tasks discussed is transaction verification, in particular, the third step of transaction verification: signature verification. The second task discussed is the hashing of transactions before creating the Merkle trees.

Signature verification accounts for approximately 62% of the total running time of transaction verification (the exact percentage depends on the machine itself). This step is parallelizable (unlike the first two steps) because it lacks dependency, meaning that each signature, message, and public key tuple required to perform a signature verification are independent of all the other signatures being verified for a given transaction. Parallelization is done by splitting a list of these tuples into equally sized batches, each being verified sequentially simultaneously.

Before the Merkle tree can be created from a collection of transactions, the hashes of those transactions are required. Since they do not depend on one another, computing the hashes can be done in parallel.

One of the challenges to sort out was how many threads were needed to obtain optimal performance. According to Amdahl's law, the larger the number of processes used, the greater the theoretical speedup with diminishing returns for each additional thread spawned. However, in reality, each additional thread introduces overhead, and machines are limited with respect to the number of processes they can run in parallel. To resolve this issue, intensive testing was run on each of the 5 machines owned by the team. Two of these machines have 2 cores, one of them has 4 cores, and the last two have 8 cores. Unsurprisingly, the machines with more cores obtained a larger speedup than those with fewer. However, the more important information learned from this experiment was that for each machine to maximize its individual performance for parallelized transaction verification, it needed to spawn the same number of threads for which its processor has cores. The data supporting this can be seen in Appendix C.

Chapter 3

Evaluation

3.1 Qualitative Results

Several unit tests were created for essential components in the system: the UTXO, blocks, message signing, transaction verification, and Merkle tree creation. The unit test suite was designed such that there is confidence in the correctness of the components with a minimal number of tests. In addition, these tests assisted in finding bugs in the system. A system simulation provided further confidence in the system's functionality.

3.2 Quantitative Results

Experimental Setup

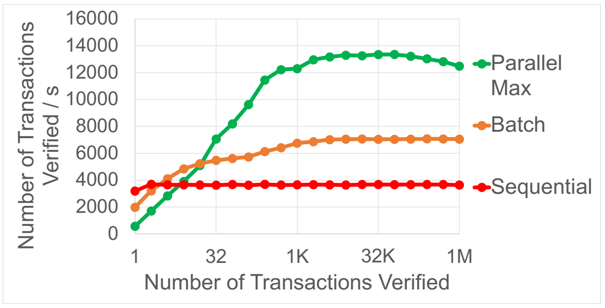
All performance tests were performed on the same machine for consistency, and all measurements were made using Rust's timers. The tests were also run in the same location to ensure consistent network performance, and the machine was always plugged in to prevent a decrease in performance when on battery power. The tests were run on an AMD Ryzen 7 5800HS CPU with Radeon Graphics and a 3.20GHz processing frequency. The CPU has 8 cores and 16 threads. The machine has 16GB of Random Access Memory (RAM) with 15.4GB of usable RAM. The operating system is 64-bit Windows, and a solid-state drive is used for secondary storage.

All the graphs in this section have a logarithmic x-axis. The few graphs with a logarithmic y-axis have a title starting with 'logarithmic'.

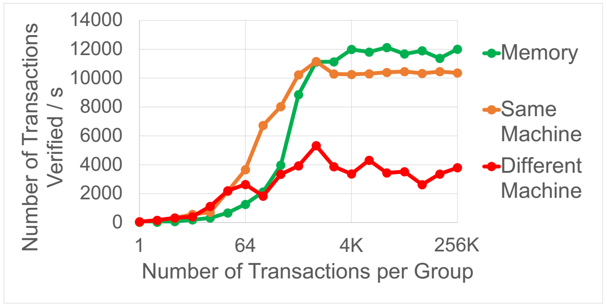
3.2.1 Transaction Throughput

Goal: Transaction verification throughput was first measured independently from any networking or blocks. The throughput was compared using three different methods of verification: sequential verification using the Elliptic Curve Digital Signature Algorithm (ecdsa) library, batch verification using the ed25519_dalek library (an optimization of ecdsa), and parallel verification [10]. Parallel verification is built on top of the batch verification code and was implemented with the optimal number of threads.

Afterward, transaction verification throughput in the context of the networking pipeline in the system was measured. The dependent variable was the number of transactions that needed to be collected in the pool before spawning the verification thread. The throughput, as measured by the receiver, was compared using three different methods of sending transactions: directly from memory, from the receiver itself, and a different machine (on the same LAN).



(a) Graph representing median throughput for isolated transaction verification (transactions are not in a block nor sent through a network). The different lines represent the different techniques used to verify the transactions.



(b) Graph representing median throughput for transactions across a network (transactions are not in a block). The different lines represent the different sources of the transactions received by the machine used for testing.

Figure 3.1: Graphs representing throughput of transaction verification. Raw verification time can be found in Appendix D.

Procedure:

1. The UTXO is initialized with a single TxOut linked to the TxIn of a first transaction. From there, valid transactions are created with one input and output starting at $n = 2^0$ transactions with increasing powers of 2 for each iteration.
2. The time for transaction verification is measured for the current quantity of transactions.
3. The previous two steps are repeated for ten total runs for each value of n, and the median verification time is used to get the throughput (number of transactions verified per second).

Discussion of Results: Figure 3.1a shows that sequential verification of isolated transactions yielded an approximately constant throughput of 4K transactions verified per second. This suggests that sequential verification scales linearly with respect to computation time. Batch verification throughput increases steadily until approximately 4K transactions, where the throughput stabilizes and yields an improvement of approximately 1.8 times compared to sequential verification. This demonstrates that batch verification performs best when verifying at least 4K transactions. After this point, it appears to scale linearly with respect to computation time. This better performance for larger batches is expected because the library used for batch verification performs optimizations for larger batches. Parallel verification throughput increases rapidly until approximately 1K transactions, at which point it stabilizes, peaks, and eventually starts decreasing steadily at approximately 33K transactions. At its maximum, parallel verification shows an improvement in throughput of approximately 3.5 times compared to sequential verification. The decrease in throughput for very large transactions suggests that the overhead of dealing with more transactions for each thread becomes significant. Since this negatively affects scalability, alternative methods are needed to ensure steady throughput to verify large quantities of transactions. However, this experiment shows that much greater performance can be achieved through parallelization, demonstrating that the cost of verifying transactions scales linearly.

Figure 3.1b shows how the throughput of verifying transactions using the networking pipeline increases until approximately 1K transactions per group. Receiving transactions from either memory or the same machine shows steeper increases until around this point. When receiving transactions from memory, the throughput stabilizes at approximately 12K transactions per second, and the throughput from the same machine stabilizes at approximately 10K transactions per second. The throughput is expected to be higher when receiving from memory because the program does not have to handle any networking infrastructure (i.e., no TCP connection is established). The throughput of receiving transactions from a different machine does not stabilize as much but is approximately 4K after 512 transactions per group. This curve's erratic behaviour indicates that the network's available bandwidth was a limiting factor in the test since peaks correspond to points where the bandwidth was high while troughs correspond to points where the bandwidth was low. The throughput is lower than the other curves as the data has to travel physically across the Wi-Fi network (unlike the other cases).

Future Considerations: The introduction of GPUs (discussed in depth in section 4.1.1) can possibly result in larger throughput improvements. This would allow for further parallelization due to the significantly greater number of cores compared to the CPU used for the existing tests.

There are several ways that testing the networking pipeline can be expanded. The verification throughput when receiving transactions from a single machine was tested; however, it would be valuable to test the effect of sending transactions from many machines simultaneously. The results of this test may help explain the erratic behaviour of the networking pipeline. This test was not performed due to time constraints.

Another valuable test is the effects of sending transactions between machines on different LANs. This would require the setup of LAN-LAN communication as described in section 4.1.3.

3.2.2 Validating a Block

Goal: The amount of time required to validate one block was measured in this test. Validation consists of three checks: detecting whether or not a fork occurred, checking that the Merkle tree of the block is consistent with that derived from the transactions, and verifying the transactions in the block. The time was measured for these three checks individually, and so was the total time. The test intends to analyze which section requires the most time and whether the timing is linear with respect to block size. It is also important that the total time is similar to the sum for the three sections. Lastly, the speedup between the parallelized and sequential implementations is shown.

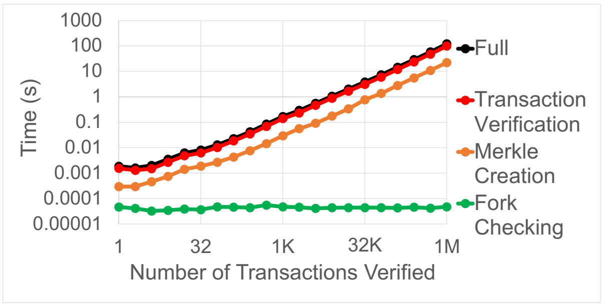
Procedure:

1. The number of transactions in one block is varied from $n = 2^0$ transactions to $n = 2^{20}$ transactions with increasing powers of 2 for each iteration.

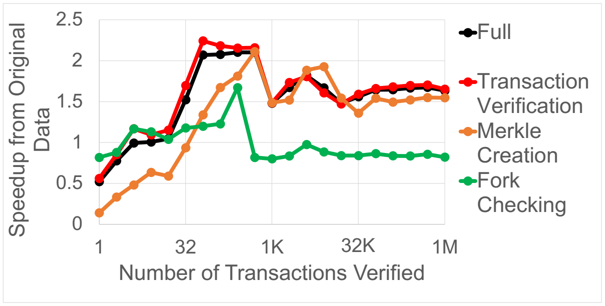
2. The blockchain is initialized with a genesis block (used to begin the chain), and then the block created with the specified number of transactions is added.

3. The three conditions mentioned above are checked, and timing is measured.

4. The three previous steps are repeated for ten total runs for each value of n and the median is used in the results.



(a) Logarithmic graph representing parallel median single block verification time.



(b) Graph representing current speedup over the basic approach of non-parallelized median single block verification. Raw computation times using sequential implementations can be found in Appendix E.

Figure 3.2: Graphs representing median single block verification.

Discussion of Results: Figure 3.2a demonstrates that checking for a fork occurs in constant time and is insignificant compared to the other tasks. Creating the Merkle tree shows linear asymptotic behaviour which is expected since creating a new Merkle tree from the incoming block's transactions is also $O(n)$. The transaction verification is more than ten times slower than Merkle tree creation and thus takes up a large majority of the block verification time. It is also done in linear time with respect to the number of transactions. The total block verification time is consistent with the sum of the three checks as expected which increases the confidence that there is no significant part missing from the analysis.

Figure 3.2b shows the speedup between the sequential and improved parallel approaches. As the number of transactions increases, Merkle tree creation and transaction verification speed settles around 1.6, and fork detection slows slightly to around 0.9. The large burst before 512 transactions is likely due to data being initially stored in the CPU cache. The improvements made to Merkle tree creations and transactions verification discussed earlier manifest here with an improvement in total block validation time.

Future Considerations: With a system that includes competing blockchains, fork checking will be much more complicated, as discussed in 4.1.2. It would be valuable to determine how that would affect the results of this test.

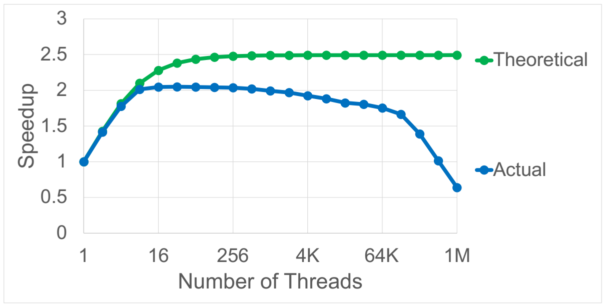
3.2.3 Speedup

Goal: The speedup of parallelizing using various numbers of threads when performing parallelized tasks was tested. Specifically, the speedup of transaction verification and Merkle tree creation was measured and these results were compared to the theoretical speedup achievable according to Amdahl's law as described in section 1.2.7.

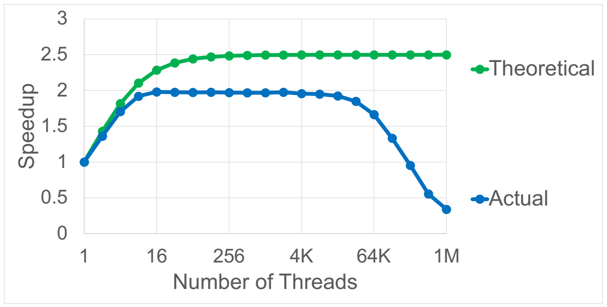
Procedure:

1. Parallelized tasks were run using $n = 2^0$ threads to $n = 2^{20}$ threads with increasing powers of 2 for each iteration.

2. The speedup was obtained by dividing the runtime by the time it took to complete the task using just one thread.



(a) Graph representing speedup of transaction verification using the parallel verification technique.



(b) Graph representing speedup of Merkle tree creation using the parallel technique.

Figure 3.3: Graphs representing speedup of parallelized tasks. Each task processed 2^{20} transactions. The theoretical speedup calculated using Amdahl's law is compared to the actual results.

3. The previous two steps were repeated for ten total runs and the median of their results was used.

Discussion of Results: For the graphs in Figure 3.3, the measured speedup nearly matches the calculated theoretical speedup for 1, 2, and 4 threads. Transaction verification speedup at 8 threads also matches. The measured speedups stabilize at approximately 2 while the theoretical speedup continues to increase steadily and stabilizes at approximately 2.5. The measured speedups do not increase significantly beyond 8 threads as the machine used for testing has just 8 cores. Additional threads do not provide more true parallelization, only more threads running concurrently, which can actually hinder performance. In fact, both measured speedup curves begin decreasing when many threads are used. This is likely due to increased overhead with more threads being spawned. Each thread has less work to do in comparison to its overhead. Transaction verification speedup begins decreasing steadily at approximately 1K threads and decreases more rapidly at approximately 128K threads. Merkle tree creation speedup decreases rapidly at approximately 32K threads.

Future Considerations: The results from this section show that CPU parallelization has only managed to extract a fraction of the available speedup. Introducing GPUs with many cores can allow the measured curves to cling to the theoretical bound for longer and reach closer to the maximal 2.5 speedup multiplier. Additionally, the success of parallelization means that looking for more areas in the system to apply parallelization in the future can result in an even more efficient system computationally.

Chapter 4

Review and Conclusion

4.1 Future Plans Given More Time

4.1.1 Leveraging GPU Acceleration for Transaction Verification

This semester, the implementation of networking required a large amount of focus. Due to this, leveraging GPU acceleration for improved performance quickly became a task that would not be possible by the end of the semester. Since the purpose of this project was to study the scalability of the blockchain system with respect to computation time, investigating the additional speedup that can be achieved through GPU acceleration would make for a more conclusive investigation with respect to parallelism. Since the testing shows that the speedup of parallelizable tasks is quite close to the theoretical maximum speedup shown in Figure 3.3 with only CPUs, it would be interesting to see how close the data could get to the theoretical maximum by utilizing GPUs.

4.1.2 Maintaining Competing Chains

As seen in Figure 3.2, the time taken to check for forks is relatively constant. This is due to two design decisions: the way fork detection is tested and the lack of maintaining multiple chains in the system. In a practical system, multiple competing blockchains must be maintained by each peer until they are sure which one contains the true ledger. The process of fork detection typically becomes much more complicated in this case because there are many chains to consider, and the new block could theoretically be a continuation of more than one of the chains. This process would greatly increase the time taken by the fork-check stage of block validation. Maintaining multiple chains would be worth investigating with respect to the scalability tests, as it brings a unique set of challenges and room for creative optimizations.

4.1.3 Networking: LAN-LAN Communication and UDP

Networking makes up a large portion of a blockchain system, and therefore it is important to test the system with multiple forms of network communication. As seen in Figure 3.1b, performance tests were run for transaction verification with communication between two peers within the same LAN. There is still one test remaining that is a natural extension of this: transaction verification between two peers in two separate LANs. This test would provide further insight into the bottlenecks of the system by allowing for further investigation into the effects of the network on the system with respect to performance. Earlier, it was established that TCP was selected as the networking protocol for the system rather than UDP. However, UDP is a viable choice with potential speed improvements over TCP. It would be an interesting study to compare two similar systems where one uses TCP and the other uses UDP to see the performance difference due to networking protocol.

4.1.4 Scripting Language: Creation of Smart Contracts

A useful feature of a blockchain system is the ability to allow for flexible and intelligent transaction constraints. These constraints dictate that additional conditions must be met beyond the associated cryptographic key pairs before the transaction's value can be spent. This is done through the use of a scripting language to allow for the description and evaluation of conditions in the form of an executable script.

The implementation of this feature immediately realizes a trade-off between the capability of the system and its performance. With the introduction of a scripting language, the system would be more capable in terms of how transactions can be designed; however, verifying these transactions would take more time. It would be interesting to analyze just how much performance would be impacted as a consequence.

4.2 Impact on Society and Environment

The system that we developed throughout the project uses networking to enable multiple parties to send transactions to each other. Many computers operating at the same time require lots of computational power, which may negatively impact the environment. An increase in the energy requirement worldwide would almost certainly lead to an increase in non-renewable resource consumption as the share of global electricity generation from renewable sources was only 20.5% as of 2018 [11]. Nevertheless, the performance of our system was significantly improved using parallelization and pipelining. These improvements reduce the amount of computation required to process and send requests.

Traditional banking involves the use of cash (which is made from non-renewables) to facilitate transactions between different parties. The cost of cash includes production, transportation, refining, printing, and distribution. Expensive resources have been used to produce cash, such as water, electricity, and fuel. It was demonstrated that the overall annual environmental cost of cash notes ($12.9 billion USD) far exceeds the annual environmental cost of Bitcoin ($1.3 billion USD). [14]

Concerning the risk and safety of blockchain systems, one potential concern is that users have no one who can help them when a customer loses a private key (which acts like a password), making them unable to access their wallets. As part of our system, we have developed a simple technique to make it easy for users to maintain their collection of keys. Another risk of blockchain is that the transaction information of users is stored in a public ledger which can be viewed by anyone with an internet connection [12]. If a user were to convert their digital currency into physical cash on an online platform and their information from the platform was leaked, then anyone would be able to trace their transactions using the public ledger. This scenario is becoming increasingly common as law enforcement agencies are able to trace precisely where digital currency is coming from [15].

Our blockchain system can offer many benefits to society. The use of cryptography increases trust between different stakeholders in the blockchain as it ensures the privacy of users. This enables different parties to participate in activities involving transactions that would usually require a middleman. Second, blockchain's decentralized structure means that no single entity is exclusively in charge of the system, which eliminates reliance on designated authorities. Decentralization also enhances resource distribution such that services are provided with better performance and consistency when compared to a centralized system[12].

4.3 Report on Teamwork

Aaron	Aidan	Roey	Sami
Transaction Scheme	Validator	Transaction Generator	Message Encoding
Blocks and Merkle Tree	Interactive Shell	DOT Graphing	Creating Txs From CLI
UTXO	MIMO Transactions	Fork Detection	Cryptographic Function
Transaction Verification	Block Validation Testing	Simulation Creation	Logging
Block Generator	Save and Load	Multiple Ports per Peer	Consistent UTXO
Save and Load	Managers	Network Infrastructure	Signing Transactions
Parallel Tx Verification	Peer Server	Tx Handling Pipeline	Unit Testing
Networking Infrastructure	Network Architecture	Preparing Networking Tests	Performance Testing
Peer Server	Fork Detection	Managers	Report
Peer Save and Launch	Peer Launch	Performance Testing	
Managers	Code Refactoring	Report	
Tx Handling Pipeline	Simulation		
Parallel Merkle Hashing	Report		
Miner			
Report			

Table 4.1: Team Members' Main Contributions

Our team was very well organized and collaborated well together. We had a standard meeting schedule, which we adhered to throughout the project. Teammates had support from one another when working on their individual tasks. This support came as assisting in problem-solving, debugging, and decision-making.

4.4 Conclusion

Throughout the two semesters, our team has completed many important features such as:

- Command Line Interface.
- Live Simulation of a Blockchain System with Transaction and Block Creation.
- Peer-to-peer Networking.
- Parallel Batch Transaction Verification.
- Transaction Processing Pipeline.
- Peer, Miner, and Server Information Managers.

Our team set out to justify our thought that blockchain systems are scalable. As a result of the work our team has done to produce and test the system defined above, we are convinced that blockchain technologies are in fact scalable. This project accomplished the goal set out by the study and showed that blockchain systems' operations meet an $O(n)$ time complexity with respect to computation time. Given that blockchain systems are scalable, we hope that their adoption into many industries and technologies is quickly explored.

Due to the vast array of different sub-fields involved in developing a blockchain system from scratch, each member of our group has learned many valuable skills. These include proper software testing procedures, basic cryptography, and peer-to-peer networking. Knowledge of these skills will allow us to be more confident in tackling future endeavors which require these specific topics. Other non-technical skills such as public speaking, time management, and conflict resolution will prove useful as we progress with our future careers.

Bibliography

[1] "IBM Supply Chain Intelligence Suite - blockchain transparent supply," IBM. [Online]. Available: https://www.ibm.com/products/supply-chain-intelligence-suite/blockchain-transparent-supply.

[2] "What is IOT with blockchain? - IBM Blockchain," IBM. [Online]. Available: https://www.ibm.com/topics/blockchain-iot.

[3] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." [Online]. Available: https://bitcoin.org/bitcoin.pdf.

[4] "Bicoin Developer Guide: Transactions," Bitcoin. [Online]. Available: https://developer.bitcoin.org/devguide/transactions.html.

[5] X. Shen, H. Yu, J. Buford, and M. Akon, Handbook of peer-to-peer networking. Springer.

[6] A. S. Tanenbaum, Computer Networks: Problem Solutions. Upper Saddle River: Prentice-Hall International, 1996.

[7] M. Hill and M. Marty, Amdahl's Law in the Multicore Era. [Online]. Available: https://research.cs.wisc.edu/multifacet/amdahl/.

[8] "Abseil open-source foundational code," Abseil. [Online]. Available: https://abseil.io/about/design/swisstables.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to algorithms. Cambridge (Mass.): MIT Press, 2009.

[10] Dalek-Cryptography, "Dalek-cryptography/ED25519-dalek: Fast and efficient ed25519 signing and verification in rust.," GitHub. [Online]. Available: https://github.com/dalek-cryptography/ed25519-dalek.

[11] *2018 Renewable Energy Data Book*, US Department of Energy, 2018.

[12] A. M. Antonopoulos, *Mastering bitcoin*, 1st ed. Sebastopol, CA: O'Reilly Media, Inc, 2010.

[13] J. L. Hennessy, D. A. Patterson, "Trends in Technology" in *Computer Architecture*, 5th ed. Burlington, (Mass.), USA: Morgan Kaupmann, 2012, pp. 17

[14] M. Donfang, E. Flood, "How Green is the Greenback? An Analysis of the Environmental Costs of Cash in the United States" [Online]. Available: https://sites.tufts.edu/digitalplanet/how-green-is-the-greenback-an-analysis-of-the-environmental-costs-of-cash-in-the-united-states/

[15] R. McMillan, "The U.S. Cracked a $3.4 Billion Crypto Heist—and Bitcoin's Anonymity" [Online]. Available: https://www.wsj.com/amp/articles/bitcoin-blockchain-hacking-arrests-93a4cb29

Chapter 5

Appendix

Appendix A

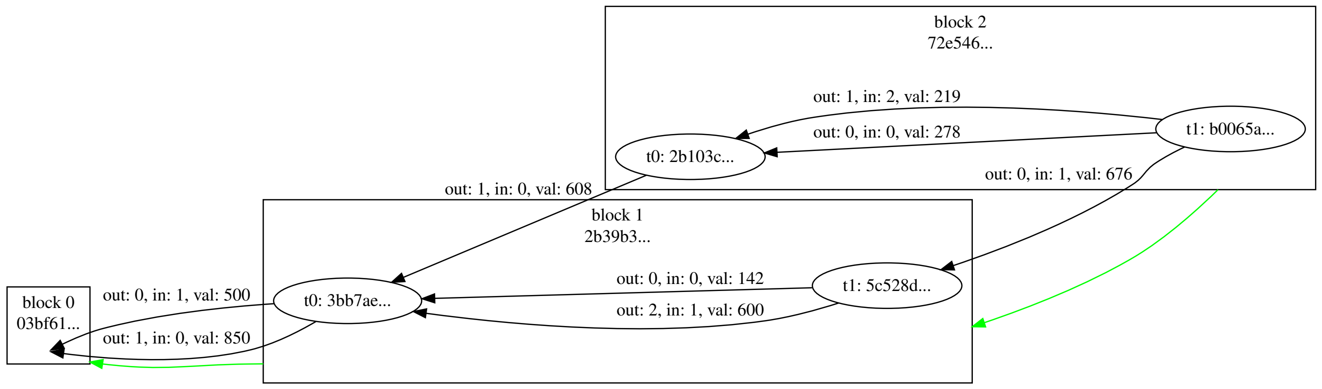


Figure 5.1: A DOT graph generated from the simulation. The rectangles represent blocks, and the ovals within them are transactions in those blocks. The black arrows represent the TxIns pointing to TxOuts that they refer to in previous transactions. The green arrows are the links between successive blocks.

Appendix B

A list of all commands that can be sent across the network:

- **ID Query:** Query from a peer to the peer server to obtain a unique identifier.
- **Neighbours Query:** Query from a peer to another peer to obtain a map of IDs to IP addresses and a map from IP addresses to ports.
- **Get ID:** Returns the ID of the peer to the caller.
- **Get Ports:** Returns the ports that the peer listens on to the caller.
- **Get IP Map:** Returns the map from peer IDs to IP addresses to the caller.
- **Get Ports Map:** Returns the map from IP addresses to listening ports to the caller.
- **Get All:** Returns the peer ID, ports, IP map, and ports map to the caller.
- **Get Block Information:** Returns the hash of the head block of the blockchain, the peer ID, and both maps to the caller.
- **Set Transaction:** Handles an incoming transaction from another peer.
- **Set Block:** Handles an incoming block from another peer.
- **Set Neighbours:** Handles the result obtained from this peer making a Neighbours Query.
- **Set Block Download:** Handles the result obtained from this peer making a Block Download Query.

Appendix C

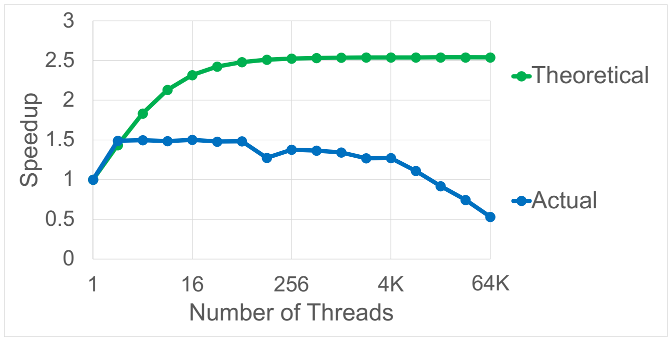


Figure 5.2: Graph representing speedup of parallelized transaction verification using a machine with two cores. Each task processed 2^{16} transactions. The theoretical speedup calculated using Amdahl's law is compared to the actual results.

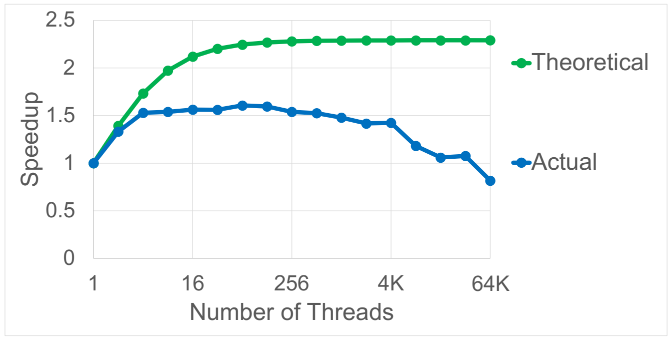


Figure 5.3: Graph representing speedup of parallelized transaction verification using a machine with four cores. Each task processed 2^{16} transactions. The theoretical speedup calculated using Amdahl's law is compared to the actual results.

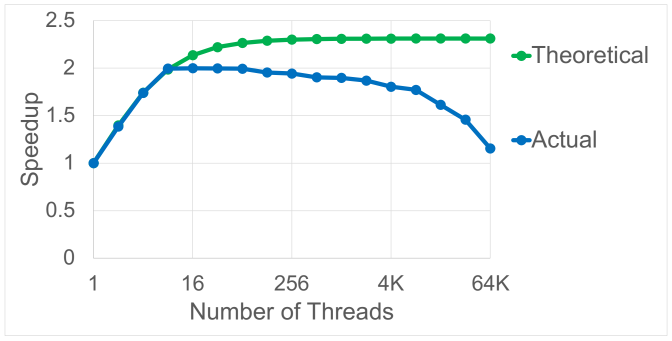


Figure 5.4: Graph representing speedup of parallelized transaction verification using a machine with eight cores. Each task processed 2^{16} transactions. The theoretical speedup calculated using Amdahl's law is compared to the actual results.

Appendix D

Number of Txs	Sequential	Batch	Parallel Min
1	313.5	503.5	1759
2	541.5	623.5	1174
4	1096	974	1412.5
8	2189	1651	2046
16	4385	3060.5	3145
32	8826	5845	4528
64	17395.5	11386	7813.5
128	35292	22338.5	13305
256	69393	41767.5	22363.5
512	140645.5	79855	41893.5
1024	280265.5	151652	83259.5
2048	557976.5	298079.5	157996.5
4096	1118766	583423	310950.5
8192	2250890	1162276	616139
16384	4456664	2319647	1235339
32768	8913459	4652026	2454437
65536	17846371	9303474	4906158
131072	35742612	18589635	9914041
262144	71306157	37047841	20119180
524288	1.43E+08	74197776	40877559
1048576	2.88E+08	1.49E+08	83999878

Table 5.1: Raw median isolated transaction verification times in microseconds. Columns 2 - 4 represent the different techniques used to verify the transactions. The parallel column is the data that corresponds to the thread that produced a minimum median value.

Number of Txs	Memory	Same Machine	Different Machine
1	51330	20739	15973.5
2	58000	12447	12310
4	52746	12463	12260.5
8	39836	13881.5	19769
16	50193	22378.5	14225.5
32	47291.5	14815	14503.5
64	50369.5	17483.5	24185.5
128	60134.5	19023	69653
256	64402.5	31877.5	76495
512	57835	50015	130339
1024	92085.5	91866.5	192298.5
2048	184042.5	199116.5	528021.5
4096	341761	399426.5	1215148
8192	693786.5	794971.5	1895734
16384	1351811	1576119	4763779
32768	2807604	3133262	9297388
65536	5510454	6346491	24936140
131072	11527864	12541757	38939925
262144	21840602	25302874	68955795

Table 5.2: Raw median transaction verification times across the network in microseconds. Columns 2 - 4 represent the different sources of the transactions received by the machine used for testing.

Appendix E

Txs per Block	Fork Checking	Merkle Creation	Transaction Verification	Full
1	38	41	863.5	968
2	36	99	1105	1240.5
4	38	222.5	1746	1989
8	39	473	2934	3559
16	40	836.5	5541.5	6511
32	43	1756	10555	12326
64	57	3573.5	22830	26632.5
128	56.5	7198	40424.5	47424.5
256	73.5	13893	74244	88369
512	45	30336.5	148003	178142.5
1024	38	43559.5	207112.5	250464
2048	38	85344	403743	489641
4096	40	172193	834851.5	1007097.5
8192	38.5	337145	1420463.5	1760361
16384	37	518113.5	2458693.5	2982018
32768	37	1039992.5	4946843	5988712
65536	38	2085798	9910518.5	11997244
131072	36	4185671.5	19924720.5	24108761
262144	38	8393759.5	40090241.5	48487192
524288	36	16919274	80835961.5	97755850
1048576	39	34154069	164026474	198232433

Table 5.3: Raw times to complete the steps in block validation using sequential implementations in microseconds.