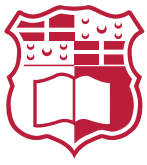
Improving Shard Allocation in Account-Based Blockchain State Sharding via Label Propagation Algorithm

Neville Grech

Supervisor: Prof. John Abela

June 2025

Submitted in partial fulfilment of the requirements
for the degree of Bachelor of Science in Information Technology (Honours)
(Software Development).



L-Università ta' Malta
Faculty of Information &
Communication Technology

Abstract

Blockchain systems face a critical scalability challenge, as consensus protocols limit transaction throughput. State sharding, which partitions the global state into smaller shards that process transactions in parallel, offers a promising solution. However, its effectiveness depends heavily on the account-to-shard allocation strategy.

The primary objective is to minimise cross-shard transactions, which increase latency and communication overhead, while maintaining a balanced transaction workload across shards to prevent bottlenecks. Shard allocation is modelled as a balanced graph partitioning problem, where vertices represent accounts and weighted edges reflect transaction volume. As this problem is NP-hard, a graph-based heuristic called Constrained Label Propagation Algorithm (CLPA) is used to assign accounts to shards based on transaction patterns. The project aims to improve shard allocation by investigating and enhancing the CLPA.

Several key enhancements to CLPA are proposed and implemented. These include a novel shard score function with a revised penalty term, an early stopping mechanism based on convergence, a memory-based label voting system to stabilise decisions, and a deterministic parallel execution strategy that improves solution quality through concurrent exploration.

Extensive evaluation was conducted using a real-world Ethereum dataset consisting of the first three million transactions. The proposed algorithm achieved a 21.2% reduction in average partitioning fitness compared to the original CLPA, a metric reflecting the balance between cross-shard communication and workload distribution. Additionally, it reduced mean epoch runtime by 10.5%, making it more suitable for time-sensitive blockchain environments. Performance gains increased with shard count, highlighting the improved scalability of the solution. These results show the approach effectively improves shard allocation.

In conclusion, this work shows that careful redesign of score functions and the use of parallel execution can significantly enhance label propagation-based shard allocation methods. The results validate this improved approach and contribute to the foundation for future advancements in scalable blockchain system design.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. John Abela, for his guidance throughout the course of my project. His consistent encouragement and inspiration were instrumental in helping me complete this work.

A word of appreciation also goes out to my parents and siblings for their unwavering support, and my grandma Maria for her encouragement.

Finally, I would also like to thank my friend Stella for always being there for me.

Contents

List of Figures

List of Tables

List of Abbreviations

CLPA Constrained label propagation algorithm.

LPA Label propagation algorithm.

UTXO Unspent transaction output.

Glossary of Symbols

α Weighting factor in the objective function.

β Penalty weight coefficient used in the shard score function.

$C(\boldsymbol{x})$ Number of cross-shard transactions in partition $\boldsymbol{x}$.

$D(\boldsymbol{x})$ Workload imbalance index for partition $\boldsymbol{x}$.

$F(\boldsymbol{x})$ Fitness score function combining cross-shard cost and workload imbalance.

$[K]$ Set of all shard indices, where K is the total number of shards.

ρ Maximum number of label switches allowed for a vertex in CLPA.

τ Maximum number of iterations allowed in CLPA.

W_k Workload of shard k.

$\boldsymbol{x}$ Partitioning solution, representing the account-to-shard mapping.

1 Introduction

1.1 Motivation

Blockchain technology has emerged as a revolutionary decentralised ledger system that facilitates secure, transparent, and tamper-resistant transactions across multiple participants. Its applications extend beyond cryptocurrencies, encompassing finance, supply chain management, healthcare, and other fields. However, a critical challenge facing blockchain networks is scalability - how to efficiently process an increasing number of transactions while maintaining decentralisation and security [1].

Among the various proposed solutions, state sharding has gained significant attention as a promising approach to enhancing blockchain throughput. State sharding partitions the blockchain state into smaller, more manageable partitions called shards that can process transactions in parallel, thereby improving efficiency [2]. However, the effectiveness of this technique is contingent upon an optimal account-to-shard allocation strategy, which must balance certain objectives.

This project explores enhancements to shard allocation by building on the Constrained Label Propagation Algorithm (CLPA) and rigorously benchmarks the proposed modifications against the original approach.

1.2 Aims and objectives

This project aims to address the challenge of improving shard allocation in account-based blockchain state sharding. Specifically, it explores the application of variations of the Label Propagation Algorithm (LPA) for community detection. This means grouping blockchain accounts into communities based on how they interact with each other through transactions.

The goal is to develop an efficient partitioning strategy that minimises the number of transactions occurring between accounts assigned to different communities, as these kinds of transactions typically incur higher processing costs and communication overhead in a sharded blockchain system. By keeping frequently interacting accounts within the same shard, the system can reduce latency and improve throughput. At the same time, the strategy must ensure that the overall transaction workload is evenly distributed across all shards to prevent any single shard from becoming a performance bottleneck. This balance is crucial for maintaining the scalability and efficiency of the blockchain network [3, 4].

To achieve this, the project incorporates several enhancements to the traditional LPA framework, including a redesigned shard score function, a convergence-based

stopping criterion, and a memory-based voting mechanism that stabilises label decisions. The proposed algorithms are also deterministic rather than stochastic, leading to more stable outcomes. These enhancements aim to produce higher-quality shard assignments in significantly less time.

1.3 Approach

This project strives to achieve the defined objectives by adopting a graph-based approach where blockchain accounts and their recent transaction history are represented as a state graph.

In fact, the shard allocation problem can be modelled as a type of balanced graph partitioning problem. Since such problems are known to be NP-hard, finding optimal solutions is computationally infeasible for large graphs. To address this, the research leverages LPA for community detection within the graph. LPA is chosen due to its computational efficiency and near-linear time complexity, making it a practical solution for large-scale blockchain systems.

Improvements and refinements over existing label propagation algorithms used in blockchain shard allocation are explored and evaluated, particularly in terms of the quality of the resulting partition, the algorithm's runtime performance, determinism, and efficiency. These dimensions together guide the development of a more practical, effective partitioning strategy for real-world blockchain systems.

Furthermore, the project introduces a parallelised version of the CLPA algorithm. Multiple runs of the algorithm are executed concurrently, enabling the exploration of diverse partitioning outcomes for each epoch. The most optimal result is selected from these candidates, improving consistency and partition quality.

The proposed methods are evaluated using a real-world dataset of Ethereum transactions, divided into epochs to simulate different transaction arrival rates. This ensures that the algorithms are tested under realistic blockchain conditions. Extensive experimentation was conducted following a scientific approach: hypotheses were formed, tested through controlled experiments, and iteratively refined to improve results.

The methodology is developed under the following assumptions, which are justified in later chapters:

- The blockchain follows an account-based model, rather than an Unspent Transaction Output (UTXO) model.
- The state graph structure remains relatively stable between epochs, allowing the use of past transaction patterns for future shard assignments.

- Accounts are limited to a single shard at any given epoch, avoiding the complexities of account replication across shards.

1.4 Overview of the report

The remainder of this report is organised as follows:

- Chapter 2 (Background) provides key concepts in blockchain architecture, scalability challenges, and the shard allocation problem.
- Chapter 3 (Literature Review) surveys existing approaches to blockchain sharding and graph partitioning, with a focus on LPA and its applicability to shard allocation.
- Chapter 4 (Methodology and Implementation) outlines the methodology used for developing and evaluating the proposed algorithms. It includes a detailed explanation of how the algorithms were implemented and how the experiments were set up to assess their performance.
- Chapter 5 (Results and Evaluation) presents the experimental results, comparing the proposed methods with a baseline algorithm that has previously been applied to the account-to-shard allocation problem in blockchain systems. Particular attention is given to partitioning fitness, execution time, and scalability.
- Chapter 6 (Future Work and Conclusions) outlines potential extensions to the study, focusing on promising directions that emerged but were not fully explored, and then summarises the key findings and contributions of the project.

2 Background

2.1 Blockchain

Blockchain is a decentralised, distributed ledger technology that records transactions across multiple computing devices securely and transparently. It was first conceptualised in the original Bitcoin whitepaper by Satoshi Nakamoto in 2008 [5]. Blockchain ensures security and trust through cryptographic hashing and consensus mechanisms, eliminating the need for a central authority. Another defining feature is its immutability, meaning that once data is recorded on the blockchain, it cannot be altered without consensus from the network. This ensures a tamper-resistant and reliable transaction history [1].

2.1.1 Blockchain data models

Blockchain systems typically use one of two data models: the UTXO model (used by Bitcoin) and the account-based model (used by Ethereum) [6]. The UTXO model treats transactions as chains of inputs and outputs, each consuming unspent outputs from previous transactions and creating new ones [7–9]. This structure enables parallel processing but requires complex UTXO tracking.

In contrast, the account-based model records balances directly in accounts, similar to traditional banking. Transactions update account balances by debiting senders and crediting recipients, simplifying state management [7–9]. As specified in the Ethereum yellow paper, an external transaction (one initiated by an externally owned account) always has exactly one sender and one recipient. However, in the case of smart contract creation, there is no recipient but rather an address for the new smart contract is created [10].

2.1.2 The blockchain scalability problem

The use of distributed consensus protocols such as Proof-of-Work and Proof-of-Stake to sequentially create and finalise blocks in blockchain systems ensures security, consistency and trust. However, this sequential process prevents the blockchain system from scaling well [8, 11]. In fact, the goal of improving the throughput, in transactions per second, that the blockchain system can handle is a prominent open problem in the field of blockchain [2, 7, 11–17].

The issue of scalability limits the potential of blockchain systems as viable solutions to real world scenarios [18]. Yet, efforts to increase throughput must be mindful of the blockchain trilemma. This refers to the balance a blockchain system

must keep between decentralisation, security, and scalability [17, 19, 20], as depicted in Figure 2.1. It has been proven that a blockchain cannot achieve the highest levels of these three properties at the same time [19].

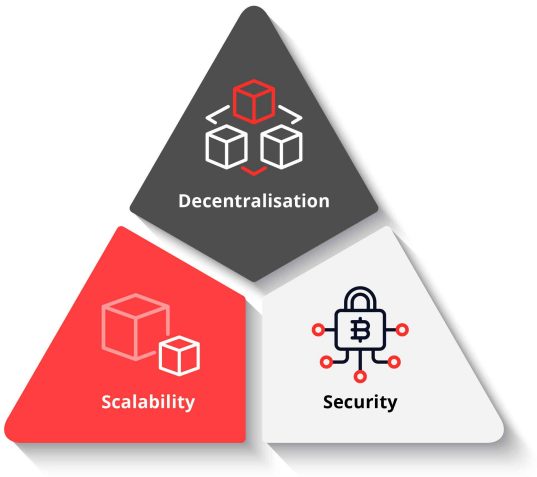


Figure 2.1 The blockchain trilemma illustrating the trade-off. Reproduced from [21].

2.1.3 Blockchain sharding

Sharding is an on-chain solution that has the potential to improve blockchain scalability while not sacrificing much from the other two properties of the blockchain trilemma [3, 17]. Sometimes referred to as horizontal fragmentation [22], sharding is a scalability solution commonly applied for databases and cloud infrastructure [23, 24]. It involves splitting a network or dataset into smaller, more manageable parts called shards. Blockchain shards can process transactions in parallel [3]. Therefore, a given node on the blockchain is not required to process every single transaction.

There exist three blockchain sharding techniques, namely network sharding, transaction sharding and state sharding [2, 3, 24]. Network sharding partitions the blockchain network into smaller groups, called shards or committees. Each node processes a specific subset of transactions assigned to its shard. Network sharding is often regarded as the foundation for the other forms of blockchain sharding [2, 3].

Transaction sharding assigns different transactions to different shards [2, 24] but still requires each node to store the full ledger state. As the ledger grows, this can become infeasible for some nodes, potentially leading to undesirable centralisation. State sharding addresses this by requiring each node to store only part of the ledger [15]. However, it introduces challenges such as ensuring atomicity for cross-shard transactions to prevent double-spending [8]. Additionally, assigning accounts to shards remains a complex problem, making state sharding an active area of research [3, 14, 15].

2.2 The shard allocation problem

This project focuses on the account-to-shard allocation strategy that must be in place for the implementation of blockchain state sharding. Before delving deeper into the problem, some important terms must be well defined. Firstly, the workload of shards. This term is widely used to refer to the number of transactions, both intra-shard and inter-shard, that each shard processes in a given period or epoch [3, 4, 7, 15, 18]. In turn, an epoch is defined as the "fixed period in which the blockchain operates" [18].

At a high level, an epoch typically consists of the consensus phase and the account reconfiguration phase, as shown in Figure 2.2. During the consensus phase each shard separately processes the transactions it was assigned according to the accounts it holds, and consensus is reached on the blocks proposed. This is followed by the reconfiguration stage where some accounts are shifted from one shard to another as determined by a reallocation process [3, 18]. After reconfiguration, the process is restarted with the next epoch commencing operations on the new partition results.

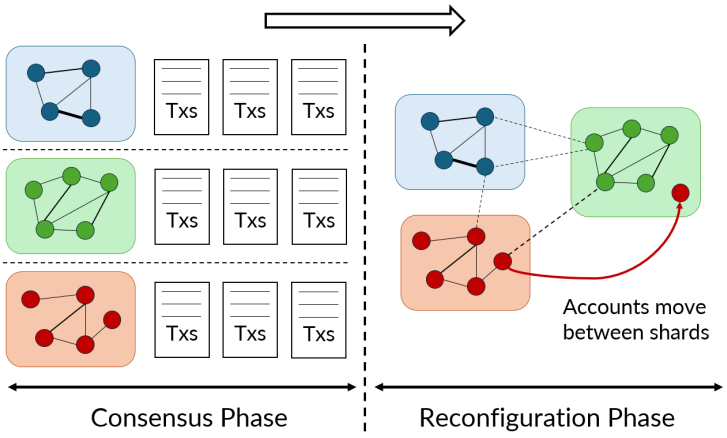


Figure 2.2 The life cycle phases of an epoch in a sharded blockchain.

A key factor is the presence of cross-shard transactions, which are transactions between two accounts that belong to different shards. They require more effort in terms of network resources, as they involve inter-shard communication and coordination overhead to ensure consistency and atomicity across shards [4, 7, 13, 15, 18]. Additionally, they introduce higher transaction confirmation latency due to the need for multiple shards to verify and process the transaction before finalising it [3, 6, 7, 15, 25]. Therefore, it is clear that cross-shard transactions must be minimised.

2.2.1 The hot shard problem

The other aspect of blockchain shard allocation is the problem often referred to as the 'hot shard' problem [18]. It occurs due to the power-law distribution observed with

accounts generating transactions. This distribution implies that a disproportionately large number of transactions on the blockchain are generated by a small set of accounts, while most accounts are responsible for only a few transactions [18]. This skewed distribution is illustrated in Figure 2.3, where the log-log plot shows that the highest ranked (by activity) accounts generate the majority of transactions.

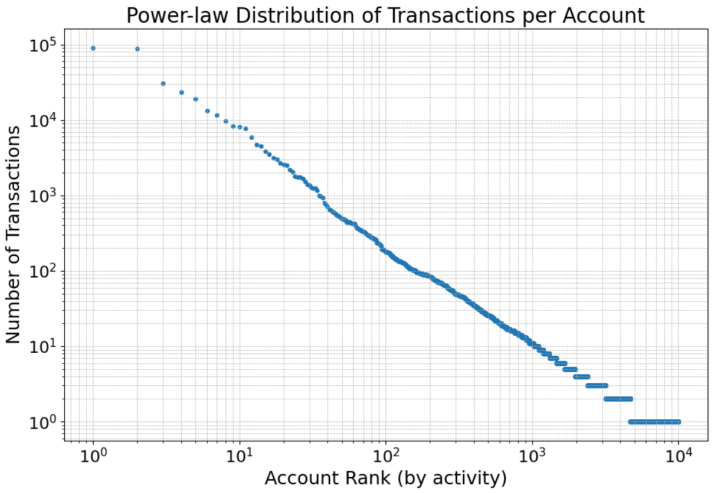


Figure 2.3 Log-log plot showing the power-law distribution of transactions per account.

This imbalance in the workload of shards results in certain shards becoming overloaded (referred to as hot shards), while others remain underutilised. This results in significantly longer transaction confirmation times, as the overloaded shard struggles to process an excessive number of transactions [3]. On the other hand, underutilised shards naturally waste computing resources by remaining idle [7].

2.2.2 Objectives of a shard allocation strategy

Accounts should be allocated to shards in a way to reduce the number of cross-shard transactions while maintaining balanced shard workloads. However, these two objectives may compete, creating a trade-off where minimising cross-shard transactions can lead to workload imbalance, while ensuring balanced workloads may increase cross-shard transactions [4, 15, 26]. An optimal allocation strategy must carefully navigate this trade-off. This can be achieved using methods such as heuristic algorithms, machine learning approaches and graph-partitioning methods [26].

2.2.3 Graph partitioning and community detection methods

This project approaches the shard allocation problem through graph partitioning, using a state graph representation common in the literature [3, 7, 15, 18, 25]. The state graph is an undirected graph where each account is a vertex, and each edge represents a transaction between two accounts during an epoch.

Figure 2.4 illustrates a simple partition of the blockchain into three shards, distinguishing intra-shard and cross-shard transactions. Edge weights, represented by varying thickness, indicate the number of transactions between accounts during a given epoch. This transforms the shard allocation problem into a balanced graph partitioning problem, enabling the application of standard graph partitioning and community detection techniques.

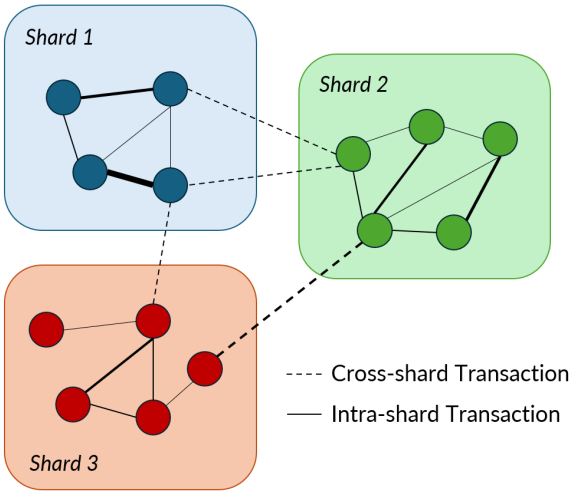


Figure 2.4 Partitioning example of a simple state graph.

Community detection is a fundamental method used to analyse a graph's structure and uncover groups within it. A community consists of nodes that have strong internal connections while maintaining relatively weak links to nodes outside the group [4, 15]. Community detection is a widely studied technique for uncovering hidden structures in complex networks [27, 28]. It can effectively guide graph partitioning.

2.2.4 The balanced graph partitioning problem

The 'k-way balanced graph partitioning' problem involves dividing a graph into k partitions while maintaining a roughly equal load across partitions and minimising inter-partition communication [29].

There are two different strategies for dividing a graph into smaller components or partitions, namely: edge-cut partitioning and vertex-cut partitioning. Edge-cut partitioning assigns vertices to specific partitions, focusing on minimising the number of cut edges when dividing a graph into subgraphs. It aims to keep highly connected nodes together while reducing inter-community connections. In contrast, vertex-cut partitioning splits a graph by cutting through nodes rather than edges [30, 31]. This results in vertices being replicated across multiple partitions to reduce the number of cut edges.

3 Literature Review

This chapter explores the evolution of blockchain sharding protocols, comparing the use of different shard allocation strategies and partitioning algorithms. Emphasis is placed on community detection techniques, particularly LPA variations which address the objectives of the blockchain sharding allocation problem.

3.1 Blockchain sharding systems

3.1.1 Blockchain scalability techniques

Various techniques have been proposed to address blockchain scalability. Sidechains connect multiple blockchains to enable asset transfers and improve interoperability [24]. Directed Acyclic Graphs offer an alternative to linear chains [6, 7, 24], though they are generally considered less secure [2]. Off-chain solutions like Lightning Networks and rollups improve scalability [6, 7] but may compromise decentralisation [2]. Of all the proposed approaches, sharding is widely regarded as the most promising for improving blockchain scalability [2, 16, 17].

Elastico [23] is viewed as the first successfully implemented protocol for a secure, permissionless sharded blockchain, where every shard processes transactions in parallel [3]. However, Elastico applies transaction sharding, which still requires each node to store the entire state ledger, as opposed to state sharding. In contrast, the Ethereum 2.0 roadmap explores state sharding as a more advanced approach - aiming to improve transaction throughput [32]. This is shown in Figure 3.1, where the *Beacon Chain* coordinates multiple shard chains that process transactions and maintain separate portions of the global state in parallel.

3.1.2 UTXO vs. account-based models in sharded blockchains

The UTXO model was often preferred in the past when designing blockchain sharding protocols, as evidenced by the large collection of works that utilise it: Elastico [23], OmniLedger [34], RapidChain [35], CycLedger [36], RepChain [37], Optchain [38] and SSChain [13]. The main reason behind this preference is that it is easier to implement cross-shard transactions using this model [2, 3].

Nonetheless, the account-based model is believed to have more potential than the UTXO model since it supports smart contract execution [7, 15]. For this reason, the account-based model is adopted by a number of recently proposed sharded blockchains such as: Monoxide [8], BrokerChain [3], Transformers [15] and TxAllo [4].

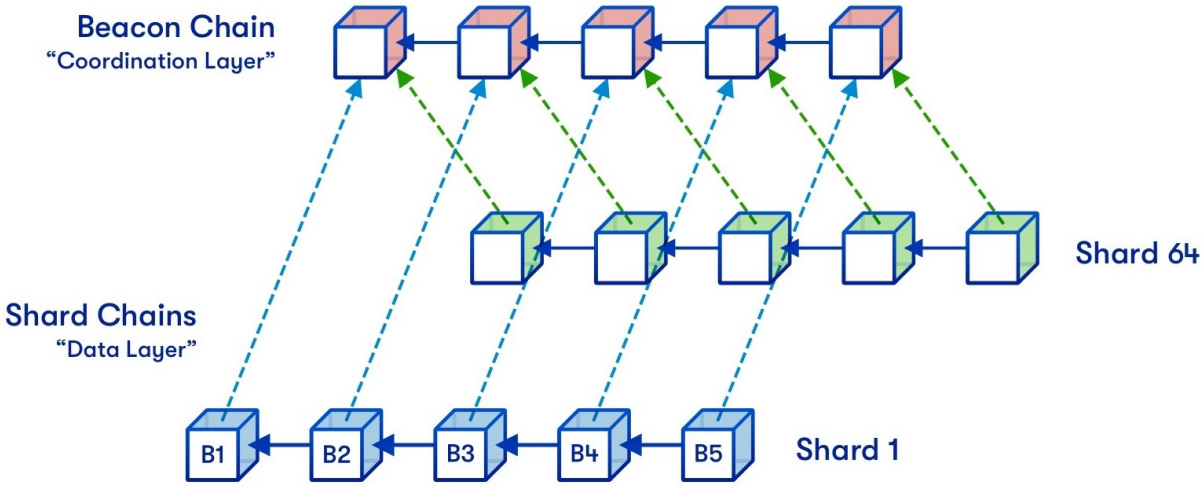


Figure 3.1 Illustration of Ethereum 2.0's proposed sharding architecture, where multiple shard chains process transactions in parallel. Reproduced from [33].

Based on this reasoning, this project adopts the account-based model.

3.1.3 Account-to-shard allocation strategies

A simple option for an account-to-shard allocation strategy is to allow users to choose which shard their account will belong to and to move between shards [6]. This avoids the overhead of running a system-managed allocation mechanism [39] and obviously eliminates the need to design a complex partitioning strategy altogether.

When users can move between shards, they tend to minimise fees by migrating to less loaded shards or those containing frequent transaction partners. This behaviour, known as 'selfish' load balancing, generally avoids extreme workload imbalance [6]. However, shards with popular accounts (such as decentralised finance accounts) can become overloaded as many users move their accounts to those shards. N. Okanami et al. demonstrate improved results over selfish load balancing using a meta-heuristic shard allocation method: simulated annealing [6].

Another option is to randomly assign accounts to shards. This enhances security since malicious nodes cannot predict which shard they will be assigned but results in a high variation in the workloads of each shard. This occurs since a shard may be assigned multiple highly active accounts, while another may be assigned less active accounts [18]. A number of blockchain sharding systems including Elastico [23], Omniledger [34] and RapidChain [35] perform shard allocation using some protocol with an element of randomness [25].

Monoxide [8] assigns accounts to shards based on specific bits of the account address, making the process deterministic. This is beneficial because if partitioning

were non-deterministic, operators in different shards would need to communicate to reach consensus on the result [4]. A deterministic algorithm ensures all operators derive the same partition from the same input, eliminating coordination overhead.

The problem with these types of sharding schemes is that a large proportion of cross-shard transactions arises since these strategies do not consider the relationships between accounts in terms of transaction activity. Since the accounts in the state graph exhibit community-like interaction patterns, the network is referred to as a social network [15, 40]. An example of such a social network is illustrated in Figure 3.2. It can be observed that the nodes form densely connected clusters. This characteristic of blockchain networks renders community detection a more effective and principled approach for partitioning.

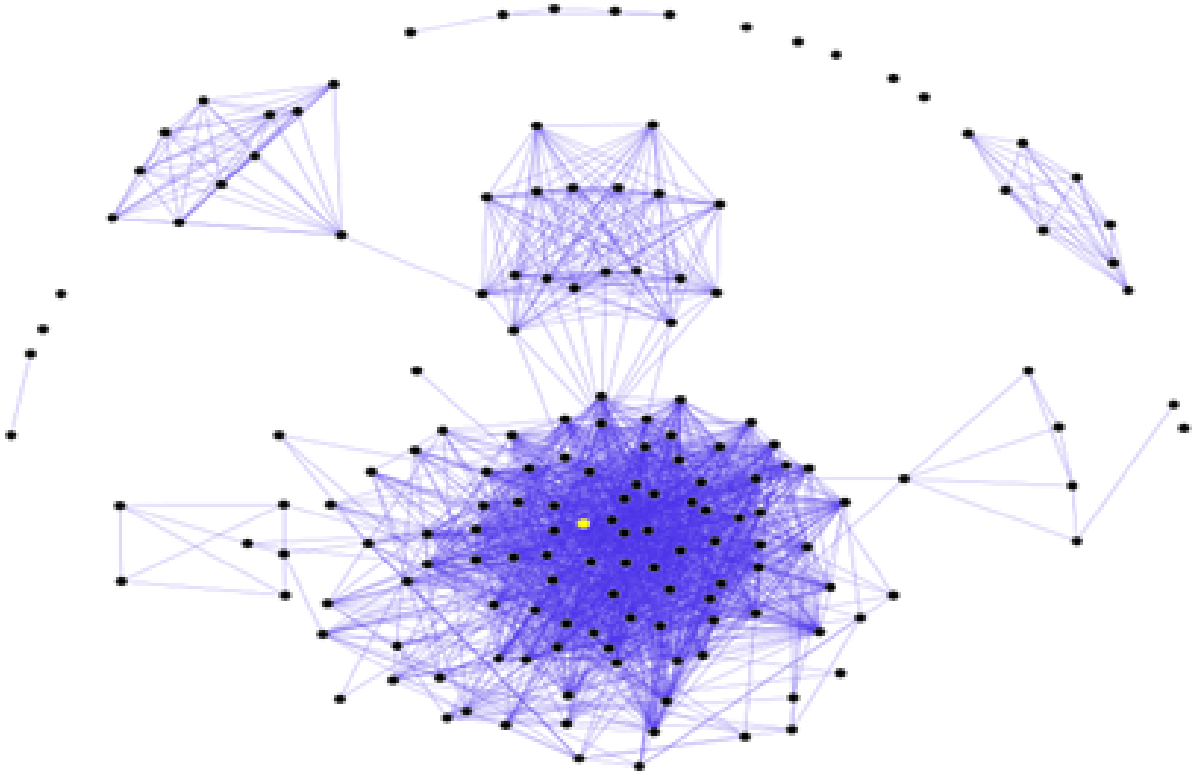


Figure 3.2 An example of a social network with densely connected clusters. Reproduced from [41].

It has been established that accounts in the state graph frequently interact with the same counterparts across epochs. This makes analysing past transaction data useful for predicting future behaviour. For this reason, many studies [3, 4, 6, 7, 15, 18] use account interaction history, typically from the most recent epoch, to guide shard allocation in the next epoch. Okanami et al. argue that for effective load balancing, the interaction data used must be recent rather than outdated, making the latest epoch a good compromise between accuracy and efficiency [6].

3.1.4 Incremental partitioning techniques

A desirable characteristic for the partitioning algorithm is that it is adaptive or incremental. This refers to the capability of the partitioning method to build upon the previous partitioning results to make improvements and adapt to the new data, rather than starting from scratch [4]. In the case of Tx-Allo, the adaptive algorithm 'A-TxAllo' can be run more frequently, for a lesser expense in terms of computing resources and time, than the global algorithm 'G-TxAllo' [4]. A Hidden Markov Model, applied by Xi et al. to detect community structures from a graph of the network, also adopts the adaptivity principle by fine-tuning the sharding structure based on the previous allocation [16].

3.2 Shard allocation as an optimisation problem

N. Okanami et al. [6] provide a formulation of the optimisation problem of effective account to shard allocation. They reason that transaction fees correlate with shard load, while acknowledging that their exact cost is usually unpredictable [6]. Nonetheless, the rationale is that optimising shard load distribution reduces overall transaction fees, since no shard becomes overloaded. The authors go on to call the problem the *Maximum Load Minimisation Problem* and state that it is NP-hard, since the *Task Assignment Problem* is known to be NP-hard and can be reduced to it in polynomial time [6].

Wang et al. [39] model the allocation problem as an optimisation task, with the goal of maximising overall system throughput. However, the proposed system called Orbit is different from the other sharding allocation mechanisms considered since it involves a dynamic account allocation strategy. It does not rely on historical transaction data but reacts in real time by analysing the pending transaction pool [39].

3.2.1 An intuitive objective function

The Transformers paper [15] also formulates an optimisation problem and goes on to present an intuitive Objective Function (3.1) for shard allocation.

$$\min F(\boldsymbol{x}) = \alpha \cdot C(\boldsymbol{x}) + (1 - \alpha) \cdot D(\boldsymbol{x}) \tag{3.1}$$

The function (3.1) is defined in terms of several expressions introduced throughout the paper. The partitioning solution, denoted by $\boldsymbol{x}$, serves as the input to the objective function. The number of induced cross-shard transactions is represented by $C(\boldsymbol{x})$, while $D(\boldsymbol{x})$ denotes the workload imbalance index. The latter is a metric that

quantifies how unevenly the workload is distributed across shards.

The idea behind the objective function is to minimise the fitness function $F(\boldsymbol{x})$, ensuring an optimal trade-off between reducing cross-shard transaction workload and maintaining workload balance across shards. The parameter α, which lies between zero and one, acts as a weighting coefficient that determines the relative importance of these two objectives [15].

3.2.2 NP-hardness of the shard allocation problem

Li et al. acknowledge that finding the optimal solution where the objective function (3.1) is minimised is an NP-hard problem [15]. They support this claim by referring to the following three similar works on graph partitioning:

- a paper which formalises the 'application-driven partition problem' and claims that it is NP-hard as can be verified by reduction from the 'set partition problem' [42].
- a paper which claims that: "bisecting a graph into two equal parts with as few crossing edges as possible" is known to be NP-hard [43].
- a paper [44] (that laid the foundation for another paper that introduced a popular partitioning algorithm called Metis [45] by the same authors) that makes the claim that the graph partitioning problem is NP-complete and therefore also NP-hard.

However, none of these problems exactly match the balanced graph partitioning formulation relevant to blockchain sharding. To further support their NP-hardness claim, Li et al. [15] refer to *The Design of Approximation Algorithms* [46], which discusses related problems. One is the 'multiway cut problem', involving k-way separation of predefined nodes, which is an assumption not present in blockchain sharding. Another problem is 'recursive balanced cuts', which uses iterative binary splits rather than dividing the graph into k subsets simultaneously [46].

Therefore, the references cited by [15] do not provide conclusive evidence that the balanced graph partitioning problem is NP-hard. Additionally, other papers mentioned earlier also claim that optimal multi-objective sharding, framed as a balanced graph partitioning problem, is NP-hard [6, 7, 18]. However, they do not cite sources that explicitly prove this.

An exception is the paper which introduces a graph partitioning algorithm called *High Degree Replicated First* and claims that both the edge-cut and vertex-cut versions of the problem are NP-hard [30]. This paper cites a source that proves the k-way balanced graph partitioning problem is indeed NP-hard, as often claimed. Specifically, it refers to the paper *Balanced Graph Partitioning* [29] which establishes that the problem

is NP-hard for $k \geq 3$. This is proven via a reduction from the strongly NP-complete '3-Partition problem'. These results provide solid and satisfactory proof of the NP-hardness of the balanced graph partitioning problem.

3.3 Graph partitioning methods applied to shard allocation

This section reviews two graph partitioning algorithms that have been used in prior work on shard allocation. LPA variants are excluded, as they are discussed separately in the following section and include the algorithm used as a baseline in this project.

3.3.1 Louvain algorithm

The Louvain algorithm is a popular, edge-cut community detection algorithm. Although it has been in use since its introduction in 2008 [47], it remains one of the most widely regarded and effective community detection algorithms due to its efficiency and strong modularity optimization [27, 28]. Remarkably, its time complexity is only linear [27].

The Louvain algorithm alone does not account for workload balance, a key objective in sharding. Paper [25] applies Louvain for community detection in blockchain sharding and achieves only 20% cross-shard transactions but does not address workload distribution. GPChain [18] and TxAllo [4] improve on this by incorporating Louvain alongside additional steps to balance workload. For a given number of shards, GPChain achieves 5% higher throughput than BrokerChain and double that of Monoxide [18].

3.3.2 Metis algorithm

Several studies [3, 48, 49] use the Metis graph-partitioning heuristic [45] to divide the state graph into non-overlapping partitions. Metis is effective for large-scale graphs and aims to minimise inter-partition edge cuts.

However, Zhang et al. argue that Metis is not well-suited for blockchain sharding because it does not address the trade-off between workload balance and cross-shard transactions [4]. The Transformers paper supports this view, noting that Metis "does not consider the workload balance among shards" [15]. In practice, Metis as used in BrokerChain, is outperformed by GPChain in terms of workload balancing [18].

3.4 Label propagation algorithm

Label propagation was initially introduced in 2002 [50] as an effective approach for inferring missing labels in graph data within a semi-supervised learning framework. This same concept was later adapted for community detection through the LPA, a computationally efficient method for identifying community structures in large-scale networks [51]. The algorithm is also known as the *Raghavan, Albert and Kumara* algorithm [52] after the authors of paper [51].

3.4.1 Original LPA for community detection

LPA starts by assigning each node a unique label and iteratively updates the label of each node to match the most frequent label among its neighbours. This process continues until a consensus is reached, leading to the emergence of densely connected groups forming communities.

This process is illustrated in Figure 3.3, which captures a moment during an iteration of LPA. On the left, each node is assigned a label, represented by different colours. When the algorithm reaches the central node, it examines its neighbours to determine the most frequent label. On the right-hand side, the central node has updated its label to black, reflecting the most common label among its neighbours.

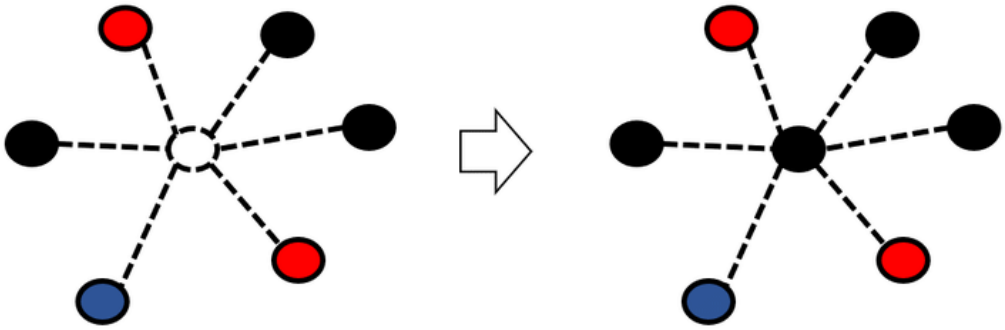


Figure 3.3 A node changing its label during an LPA iteration. Reproduced from [53].

For a more rigorous presentation, the pseudocode for the original LPA is given in Algorithm 1. It closely follows the original community detection formulation described in [51], with comments included for additional clarification. It is worth highlighting that unlike other community detection algorithms such as k-means clustering or spectral partitioning [27], LPA operates without prior knowledge of the number or sizes of communities and does not require the optimisation of a predefined objective function [51].

Algorithm 1 Original LPA

Input: Graph $G = (V, E)$ with vertices V and edges E
Output: Communities formed by grouping vertices with the same labels

for each $v \in V$ **do**
    Assign $L_v = v$ ▷ Each vertex gets a unique label
end for

$t \leftarrow 0$
stop $\leftarrow$ **false** ▷ Initialise stopping condition
while stop = **false do**
    Shuffle the vertices in a random order X
    for each $v \in X$ **do**
        Count occurrences of each label among neighbours of v
        $C_{\text{max}} \leftarrow$ labels with the highest occurrence
        if $|C_{\text{max}}| = 1$ **then**
            Assign $L_v \leftarrow$ the single most frequent label
        else
            Select a random label from C_{max}
            Assign $L_v \leftarrow$ the selected label
        end if
    end for
    ▷ Check stopping condition

    stop $\leftarrow$ **true**
    for each $v \in V$ **do**
        if L_v is not the most frequent label among its neighbours **then**
            stop $\leftarrow$ **false**
            break ▷ Label change needed — algorithm has not converged
        end if
    end for
    $t \leftarrow t + 1$
end while

Significant research has been conducted to enhance the robustness and stability of LPA for both overlapping and disjoint community detection, leading to various improvements. Nonetheless, ongoing efforts continue to refine and optimise its performance for more reliable community detection [52].

3.4.2 Variations of LPA

Many refined versions of the LPA have been proposed, but they may not be optimal for the specific problem of blockchain shard allocation. One such variant, often referred to as MemLPA, enhances LPA by allowing each node to retain past label information during community detection. Instead of relying solely on the current labels of neighbouring nodes, MemLPA uses a memory-based decision rule to select the most frequently observed label over multiple iterations. This approach helps prevent the common issue in standard LPA of forming a single dominant community and significantly improves detection accuracy [54].

Another variation of LPA, proposed by Xie and Szymanski [55], introduces a method to improve the runtime efficiency of the algorithm by classifying nodes as either active or passive based on their neighbourhood. A node is considered active if there is a possibility that it will change its label in the next iteration. Otherwise, it is marked as passive.

During each iteration, only active nodes are updated, significantly reducing unnecessary computations. After each update, the status of the node and its neighbours is re-evaluated to keep the list of active nodes updated [55]. An overview of this process is illustrated in the flowchart in Figure 3.4, highlighting the updating of the ‘active nodes list’. As clearly shown, the iterations t stop if the active nodes list becomes empty. Note the use of the word ‘passive’ as the opposite of active, since the term ‘inactive’ is sometimes used in the literature and in this work to refer to vertices with no transactions in a given epoch.

While there are many LPA variations like the ones just discussed, limited work exists on LPA variants specifically applied to blockchain sharding. Some of these will now be examined.

In the Estuary sharding protocol, a customised LPA variant is employed to optimise user state distribution across shards. It models shards as overlapping communities and assigns users based on transaction behaviour using belonging coefficients. By updating these iteratively, the LPA variant called Community Overlap Propagation Algorithm for Sharding reduces cross-shard transactions and balances the processing load [56].

Another example is the so-called Modified LPA, which is used in the blockchain system ALB-Chain [57]. Unlike the other works referenced in this project, it is

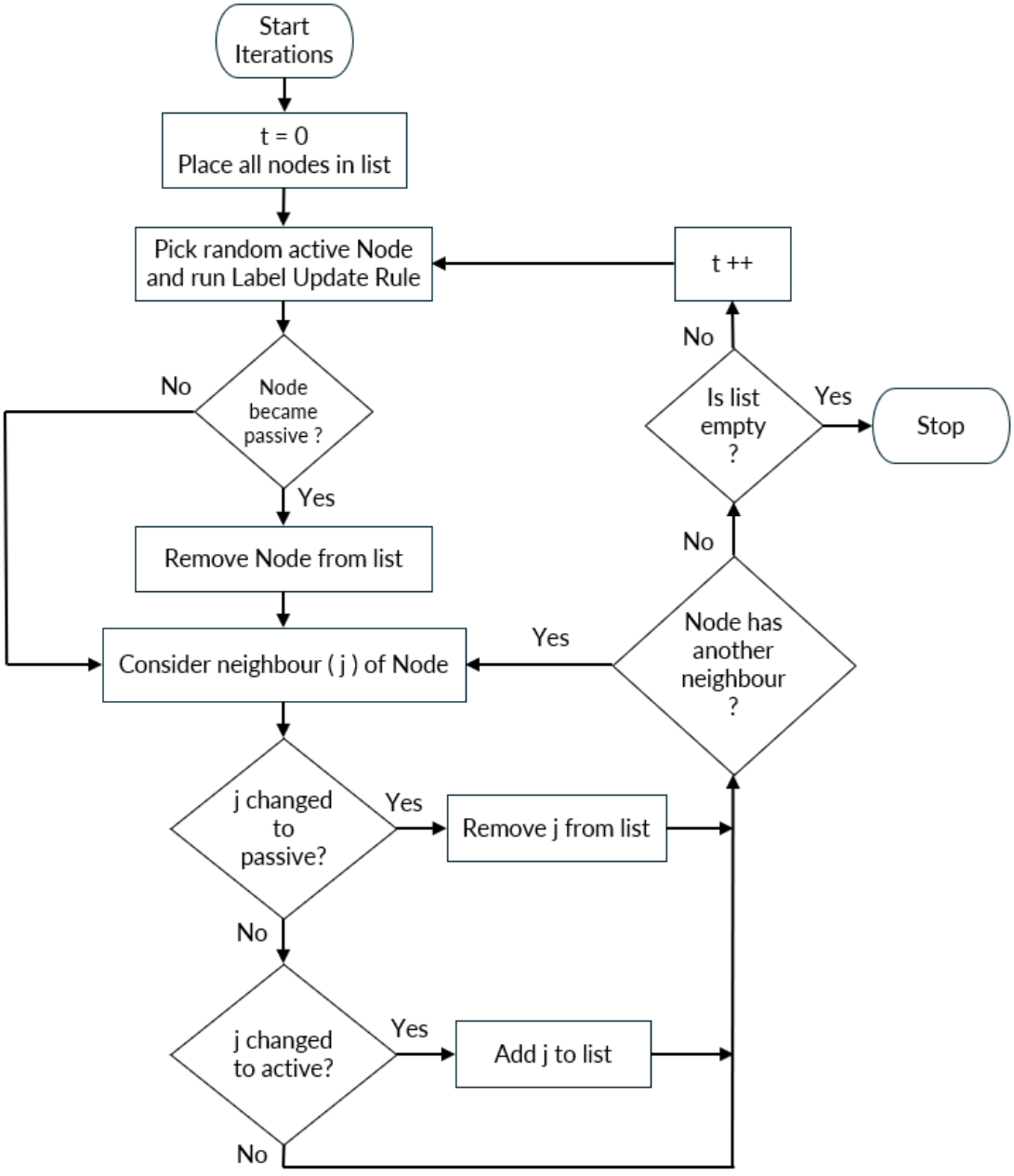


Figure 3.4 The flowchart depicts the logic driving the approach of the active nodes list.

important to note that this paper has not yet been published in a peer-reviewed journal. ALB-Chain combines Modified LPA with a greedy update mechanism. Modified LPA uses a score function to assign scores to each shard, guiding the allocation of users. However, this implementation differs from others because it introduces a novel type of replica accounts designed to eliminate the inefficiencies associated with cross-shard transactions [57].

3.4.3 Constrained label propagation algorithm

CLPA reformulates the traditional LPA as an optimisation problem. This refinement introduces a revised objective function that incorporates constraints to discourage suboptimal solutions [7]. The idea of propagating labels under constraints was introduced in [58], but the term CLPA was coined later in [59].

The Transformers protocol [15] applies CLPA to the blockchain sharding problem, guided by the objective function described in Section 3.2.1. This objective is optimised using the score function illustrated in Figure 3.5. From this point onwards, the first term on the right-hand side of the score function will be referred to as the association strength component and the second term as the penalty term.

The score function implements the constraints of low cross-shard transactions and workload balance, to assign a score to each shard with respect to a vertex. The vertex label is then assigned the label of the shard that obtains the highest score. Notably, the score function for a vertex is only defined for shards connected to the vertex via its neighbours, as unconnected shards are irrelevant [15]. Transformers [15], along with Monoxide [8], TxAllo [4], and Orbit [39], adopt a model where each account is assigned to a single shard. As a result, CLPA applies an edge-cut approach to partitioning.

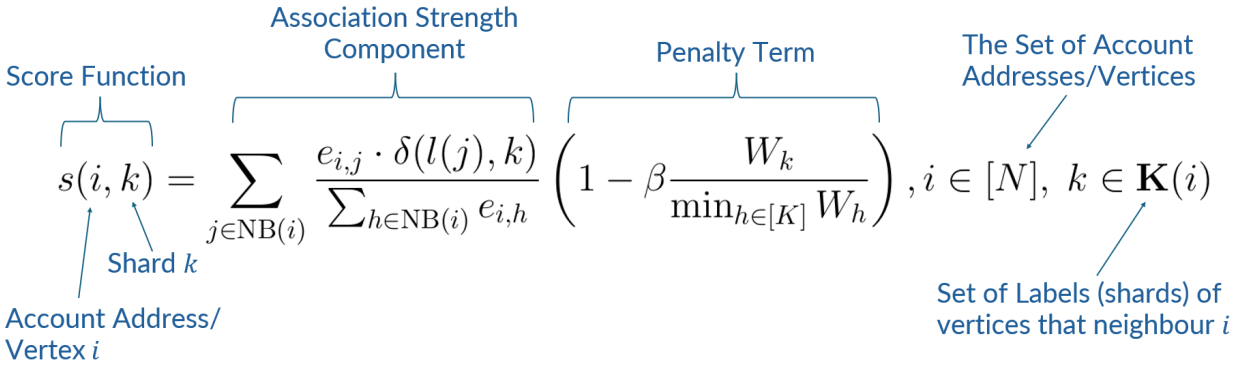


Figure 3.5 A high-level breakdown of the main components of the CLPA score function.

During iterations of CLPA, a vertex may oscillate between two shards and hence prevent the algorithm from terminating. The authors state that this is the reason

behind the setting of τ, which represents the number of iterations to be executed. Furthermore, it is made clear that the algorithm continues running for all τ iterations "no matter whether it reaches a convergence or not" [15]. As a result, the time complexity of CLPA is $O(N \cdot \tau)$, where N is the number of vertices. Another input to the algorithm ρ, represents the maximum number of times that a vertex can shift from one shard to another [15].

The CLPA presented in [15] and detailed in this section serves as the baseline for evaluating the performance of the LPA variant proposed in this project.

3.4.4 Fast shard allocation with LPA

The simplicity and efficiency of LPA make it particularly well-suited for large networks, as it operates in near-linear time [51]. LPA and its variants are widely recognised in the literature for being computationally efficient while still producing high-quality community detection results [27, 28]. CLPA adopts an incremental approach (see Section 3.1.4) from one epoch to the next. This is because the algorithm builds on the previous epoch's results during the initialisation phase by setting the labels of nodes to their shard from the previous epoch [15].

This is beneficial in the context of blockchain sharding, where efficient execution of the shard allocation algorithm is a critical requirement. The runtime speed directly affects how frequently the allocation can be updated. Faster execution enables more frequent updates, which in turn allows the system to better adapt to evolving transaction patterns, thereby improving the quality of the partitioning solution over time [4]. The importance of low runtime is further emphasised by the authors of GPChain [18] and ContribChain [60], who highlight the reduced execution time of their shard allocation algorithms as a key advantage over competing approaches.

3.5 Summary

The literature review covered key shard allocation strategies, including LPA-based approaches. It also reviewed how different blockchain systems efficiently handle scalability and workload distribution. In particular, the review highlighted how graph partitioning methods like Louvain, Metis, and various LPA variants have been applied to the sharding problem, each with different trade-offs between accuracy, efficiency, and workload balance.

4 Methodology and Implementation

This chapter details the implementation of modified versions of LPA for blockchain state sharding and justifies the design decisions with support from the literature. The source code can be found here: `https://github.com/nev19/shardinglpa`.

4.1 CLPA baseline and initial observations

The CLPA baseline was implemented exactly as described in paper [15]. This process provided a valuable opportunity to thoroughly understand the inner workings of the algorithm. Both the baseline and the newly proposed LPA were coded using Go (version 1.23.3) to maintain consistency across the experiments. Go was mainly chosen due to its high execution speed. This is important due to the time-sensitive nature of blockchain sharding as explained in Section 3.4.4. Additionally, Go offers strong support for parallelism, a feature that becomes especially relevant later in the project.

4.1.1 Baseline implementation assumptions

Certain assumptions were necessary where the CLPA [15] description lacked specific details. In such cases, the original LPA paper [51] was used as a reference where applicable. For instance, since CLPA does not specify vertex traversal order, it is assumed to follow LPA's stochastic approach by using a different random order each iteration. Similarly, in the absence of tie-breaking rules, ties are assumed to be resolved randomly, as in LPA.

Finally, the CLPA paper [15] does not specify rules for initialising labels of vertices when they first appear in the state graph, which represents a new account address on the blockchain. Therefore, it is assumed that each new vertex is randomly assigned an initial label from the set of known shards.

4.1.2 The label update mode

CLPA [15] updates labels asynchronously, meaning that vertex labels in an iteration are updated according to the labels of its already updated and not yet updated neighbours in the present iteration. Generally, asynchronous mode is preferred over synchronous mode, since the latter is prone to oscillations [51]. Nonetheless, the synchronous mode was implemented by the function *ClpaIterationSync* from the *paperclpa* package. This function updates labels based on the labels of neighbours from the previous iteration. Although simple, this implementation represents the first attempt to enhance the CLPA baseline, and its effects are evaluated in Section 5.2.

4.2 The score function

The score function used by CLPA [15] was carefully studied and scrutinised for possible improvements, since it is the crux of the algorithm. This function is shown in Figure 4.1, where key components are annotated for clarity. The previously mentioned association strength component, represents the magnitude of the association between an account address i and a shard k. Meanwhile, the penalty term is responsible for applying a penalty to the score of shard k with respect to account address/vertex i. More specifically, the expression $\beta \frac{W_k}{\min_{h \in [K]} W_h}$ represents the penalty.

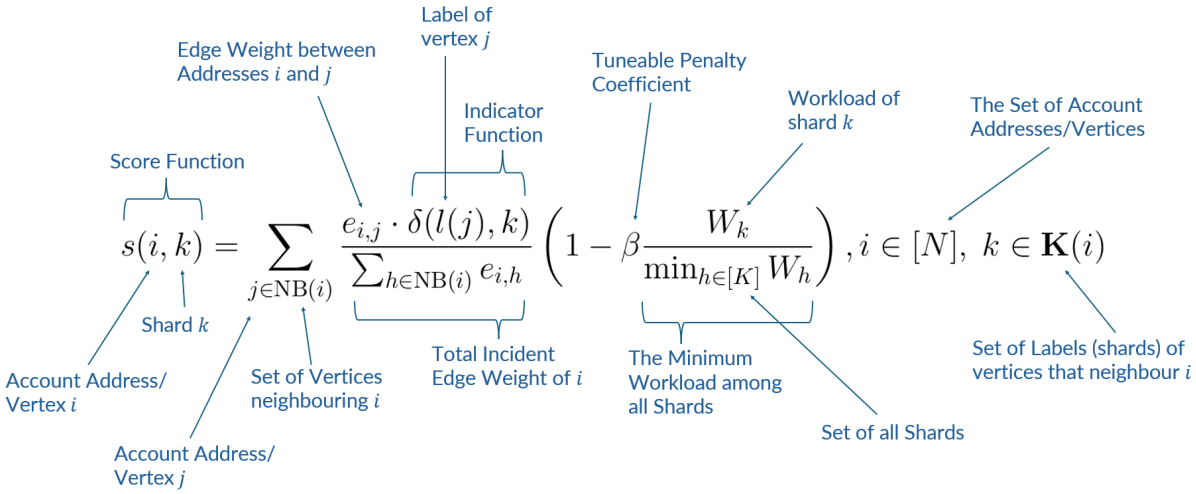


Figure 4.1 The score function used in CLPA with detailed annotation.

4.2.1 The ineffective normalisation term

This section deals with the denominator of the association strength component, labelled in Figure 4.1 as 'Total incident Edge Weight of i'. Presumably, the authors of the paper [15] included this term as a means of normalising the score by ensuring that the relative weight of transactions between i and its neighbours from a particular shard is considered in proportion to the total transactions of node i with all its neighbours. However, since scores are not compared across vertices, this normalisation is unnecessary for shard assignment. This hypothesis was formally proven correct in Appendix A. Therefore, the algorithm can become marginally more efficient by omitting the normalisation, as is done in the improved implementation proposed later in this project.

4.2.2 The penalty term problem

While examining how scores are calculated at each iteration of the CLPA, it was observed that the penalty term of the score function can sometimes result in a negative value. Before explaining why this is problematic, it is important to first note that the association strength component always equates to a positive number.

This is evident from the following: the association strength component sums over all neighbours j of vertex i, adding the edge weight between i and j if j is in shard k, and zero otherwise. This is then divided by the total edge weight of i, which is always positive, since an inactive vertex (one with no transactions in an epoch) would not be included in CLPA. Additionally, there is guaranteed to be at least one neighbour with a positive edge weight, as the score function only applies to shards with assigned neighbours (see Section 3.4.3). Since both numerator and denominator are positive, their ratio - the association strength component - is also positive. The following example illustrates why a negative penalty term can be problematic.

Figure 4.2 shows a simple state graph and the partitioning solution x. There are only a few accounts and transactions, and just four shards. The workload of each shard is shown. Suppose a CLPA iteration starts from vertex *c*, and α and β are set to 0.5, as per the settings in [15]. Using the values from Figure 4.2 and the formulae from [15], the fitness for the partitioning solution x evaluates to 10.375. The full calculation is provided in Appendix B.

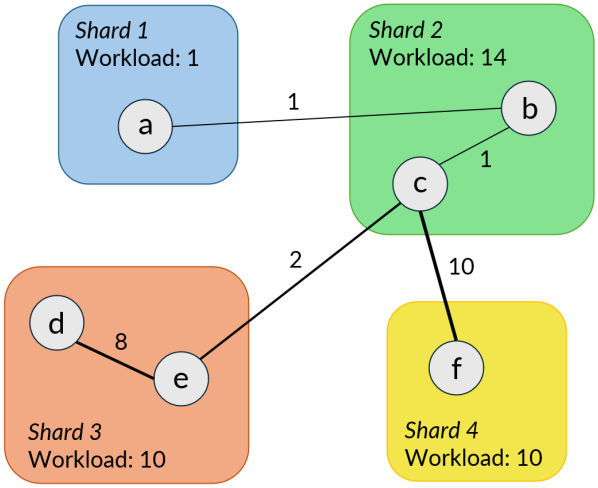


Figure 4.2 A simple example of the state graph being partitioned into 4 shards by CLPA.

Moreover, the minimum shard workload is that of *shard 1* with a workload of 1 and the total edge weight associated with vertex *c* is 13. Given these values, the shard

scores for vertex *c* are computed as follows:

$$
\begin{aligned}
s(c,1) &= \text{undefined, since } c \text{ has no neighbour which is in } \textit{shard 1} \\
s(c,2) &= \frac{(1\cdot 1)+(2\cdot 0)+(10\cdot 0)}{13}\cdot\left(1-0.5\cdot\frac{14}{1}\right) = -0.4615 \\
s(c,3) &= \frac{(1\cdot 0)+(2\cdot 1)+(10\cdot 0)}{13}\cdot\left(1-0.5\cdot\frac{10}{1}\right) = -0.6154 \\
s(c,4) &= \frac{(1\cdot 0)+(2\cdot 0)+(10\cdot 1)}{13}\cdot\left(1-0.5\cdot\frac{10}{1}\right) = -3.0769
\end{aligned}
$$

Therefore, *shard 2* obtained the greatest score of -0.4615 and so vertex *c* remains in that shard. Furthermore, the fitness of the solution remains the same, since no vertex moved between shards. However, upon closer inspection, it becomes clear that assigning vertex *c* to *shard 4* would have been better. This is because *Shard 4* has a lower workload of 10 compared to 14 of *shard 2*, and it also has a stronger connection with vertex *c*. In particular, the total edge weight between *c* and the vertices in *shard 2* is only 1. On the other hand, the total edge weight between *c* and the vertices in *shard 4* is 10, meaning the connection with *shard 4* is ten times stronger. Finally, if vertex *c* moved to shard 4, the fitness would drastically improve from 10.375 to 5.25.

The problem can be traced to the penalty term evaluating to a negative number. When this happens, a shard that has a stronger connection with a vertex is more heavily penalised. This is because the association strength term is larger in such a case and so when it is multiplied by the penalty term and becomes negative, the score is effectively smaller. This occurs in the example above, where *shard 3* and *shard 4* have the same workloads and vertex *c* is more strongly associated with *shard 4*. Nonetheless, *shard 3* obtains a greater score of −0.6154 compared to −3.0769 of *shard 4*.

Indeed, the suboptimal shard is selected whenever all connected shards get a negative score. This occurs when each connected shard's workload exceeds twice the minimum shard workload. This is illustrated in Figure 4.3, which shows how the penalty term varies with the workload of the shard being considered (W_k) relative to the minimum shard workload ($\min_{h\in[K]} W_h$, where $[K]$ is the set of all shards). Firstly, by the definition of minimum, W_k cannot be less than $\min_{h\in[K]} W_h$. Secondly, there is no limit for how small the value of the penalty term can be; it depends on how large W_k is compared to $\min_{h\in[K]} W_h$. Finally, it is noted that all these observations are made under the assumption that β is set to 0.5, as specified in [15].

The penalty term was found to be problematic in other scenarios. Say, a shard gets a positive score due to its penalty term falling in the positive range which is the middle section in Figure 4.3. Any shard with a slightly greater workload has its penalty term fall just about in the negative range, and so loses against the positive score shard, even though its association strength component may be one thousand times greater.

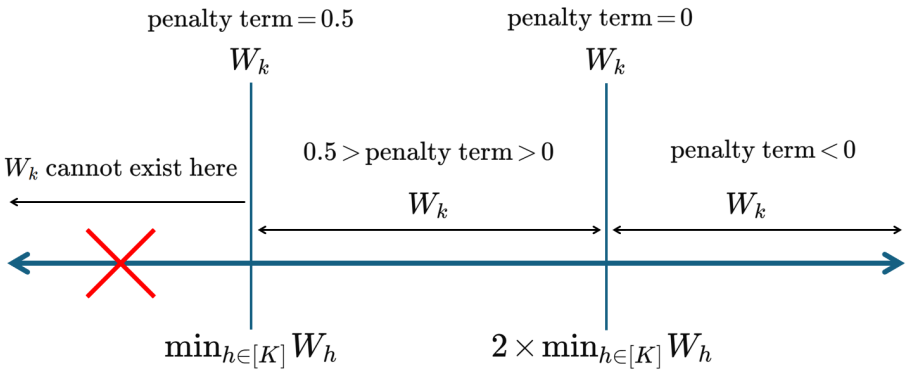


Figure 4.3 Penalty term behaviour relative to shard and minimum shard workload.

To avoid these scenarios, the new score function shown in Figure 4.4 is proposed, with a penalty term that never results in a negative number. Apart from being more intuitive, the new score function ensures that the penalty is bounded between zero and one, providing smoother scaling. The shard workload in the penalty term is normalised relative to the global range. A small constant ϵ prevents division by zero, in the rare case that the minimum and maximum shard workloads are equal.

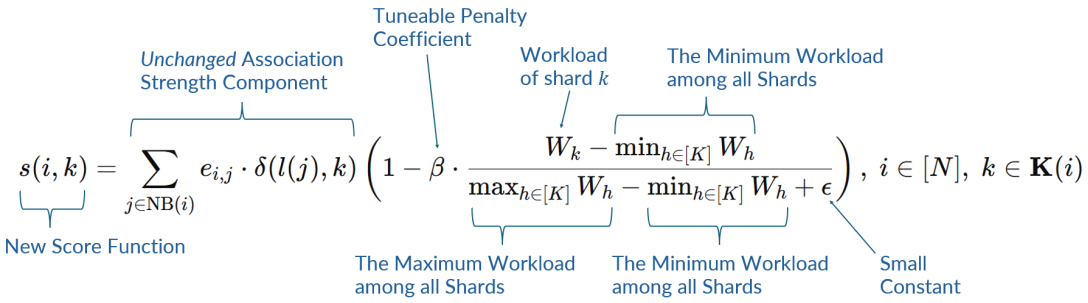


Figure 4.4 The new score function with detailed annotation.

4.3 Parallelisation of CLPA

4.3.1 Motivation for parallelisation

The CLPA [15] baseline generates different partitioning solutions across multiple runs on the same epoch, even with identical settings, due to the stochastic factors discussed in Section 4.1.1. This observation motivated the development of a parallelised version of the CLPA, where for each epoch, multiple instances are run in parallel. For each run: the new account addresses/vertices in that epoch are assigned different shards, the vertices are visited by the algorithm in different orders even in

corresponding iterations, and ties between shards with equal scores are broken differently. This enables thorough exploration of the solution space.

4.3.2 Parallel execution strategy

The setup for consecutive epochs is shown in Figure C.1. This is a custom parallel execution diagram, created to illustrate the parallel processing of partitioning solutions across epochs. In this diagram, the graph with the best fitness among all those generated in parallel is selected as the winning graph for the next epoch. The rest of the graphs are discarded. This selection is performed by the function *GetBestGraph* from *parallel_utils.go* in the *shared* package. In this case, the graph with the best fitness was *graph 3* produced by *goroutine 3* using *seed c*. In a real-world scenario, the blockchain would be sharded according to that best partitioning solution.

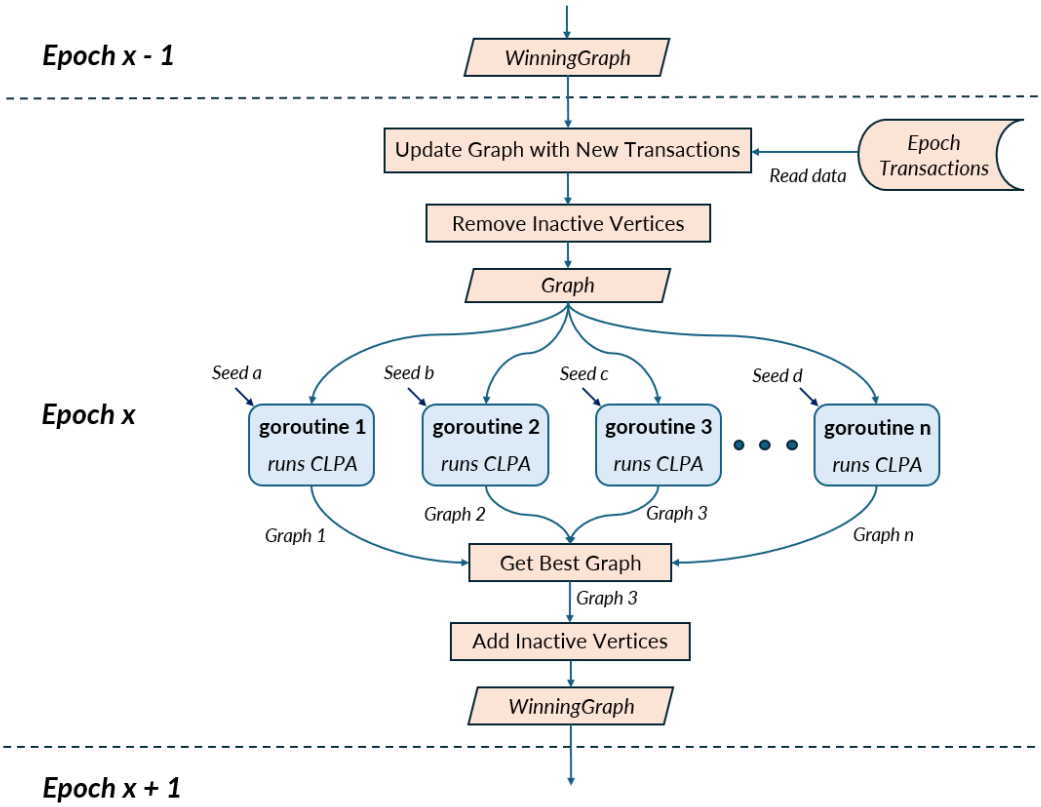


Figure 4.5 Parallel execution of CLPA across epochs, selecting the best graph.

The Figure C.1 also captures the optimisations implemented to minimise the overhead introduced by parallelisation. For instance, preparing the state graph for partitioning requires reading the epoch's transactions from storage, followed by updating the graph by creating edges for the transactions and adding any new vertices. This entire process is performed by the function *UpdateGraphFromRows* and executed only once per epoch, since the result is deterministic. Also, the inactive vertices are

removed before the parallel runs, since that action is also deterministic. Then, they are added back exclusively to the winning graph.

To ensure determinism and reproducibility, the random seeds used for each parallel run of the algorithm are obtained from a file in the *mylpa* package. This effort for reproducibility was made due to the principles outlined in Section 3.1.3. The random seeds were obtained from the website *random.org* [61] and screenshots of the random number generation process can be found in Appendix C.

4.3.3 Parallelisation implementation details in Go

The parallel process is implemented using lightweight threads called goroutines, managed by the Go runtime. With sufficient system threads, multiple goroutines can run efficiently in parallel. Coordination is handled using WaitGroups, which block execution until all goroutines complete, and a buffered channel, which safely collects their results. The *GOMAXPROCS* runtime parameter, set by default to the number of logical CPU cores, defines how many OS threads can execute Go code concurrently [62].

The parallel CLPA was run with the *-race* flag to confirm it is free of race conditions, ensuring proper synchronisation of shared data [62]. Each goroutine works on a deep copy of the state graph, allowing safe, independent updates to vertex fields and shard workloads.

4.4 The final version of LPA with all upgrades

The final version of LPA proposed in this project builds upon the deterministic, parallelised version by integrating the improved penalty term (described earlier), along with early stopping and a memory-based voting mechanism, both detailed in this section. This final version of the algorithm is implemented in the *mylpa* package. Finally, details of the 'active nodes list' optimisation (see Section 3.4.2) are in Appendix D as part of the development journey, as it was not included in the final implementation.

4.4.1 Early stopping and function dispatching

The evaluation of the new penalty term, which can be found in Section 5.3, led to the implementation of a convergence-based stopping criterion. In the context of partitioning the graph, convergence is defined as two consecutive iterations where no vertex switches shards. The function *RunClpaConvergenceStop* implemented in the *paperclpa* package runs CLPA and terminates upon convergence. The code snippet in Listing 4.1 shows the logic that runs after each iteration to check for convergence.

```
// Set flag to show convergence occurred unless a label changes
converged := true

// Iterate through all vertices and unset flag if any label changed
for id, vertex := range graph.Vertices {
  if oldLabels[id] != vertex.Label {
    converged = false
    break
  }
}
```

Listing 4.1 Go convergence check.

This early stopping behaviour is different from that exhibited by the baseline's *RunClpaPaper* function which runs for the complete number of iterations, irrespective of convergence, as discussed in Section 3.4.3.

To support multiple implementations of the same function, the *ClpaCall* function type was defined in the *paperclpa* package. The two previously described functions both implement this type. Using function dispatching, each algorithm variant can call the appropriate implementation without additional conditional logic, allowing flexible convergence strategies. This follows idiomatic Go practices [62], minimising runtime overhead. Additionally, the *ScoringPenalty* and *ClpaIterationMode* function types abstract the penalty term and iteration update mode, respectively.

4.4.2 Memory voting mechanism

The final version of LPA employs a memory voting mechanism to stabilise shard assignments and reduce label oscillations. Each vertex maintains a *LabelVotes* map that records the number of votes it has accumulated for each shard throughout successive iterations. In each iteration, a vertex updates its vote tally but only changes its shard label if another shard gains a significant voting advantage.

Figure 4.6 presents a finite state machine that models the behaviour of a vertex during the CLPA process under the memory voting mechanism. The starting state corresponds to the start of the CLPA process, where each vertex enters the *Initialised* state which represents the assignment of its initial label. Following this, the vertex transitions into the *Voting* state, where the scores of all shards with respect to the vertex are computed. The states *Stable*, *Moved*, and *Frozen* are treated as final states, representing the possible conditions in which a vertex may reside when the CLPA algorithm terminates - either due to convergence, iteration limit, or update threshold.

The voting margin for a vertex to switch labels was set to require at least one additional vote, based on preliminary testing that showed this provided the best balance of responsiveness and stability. Additionally, the implementation enforces a

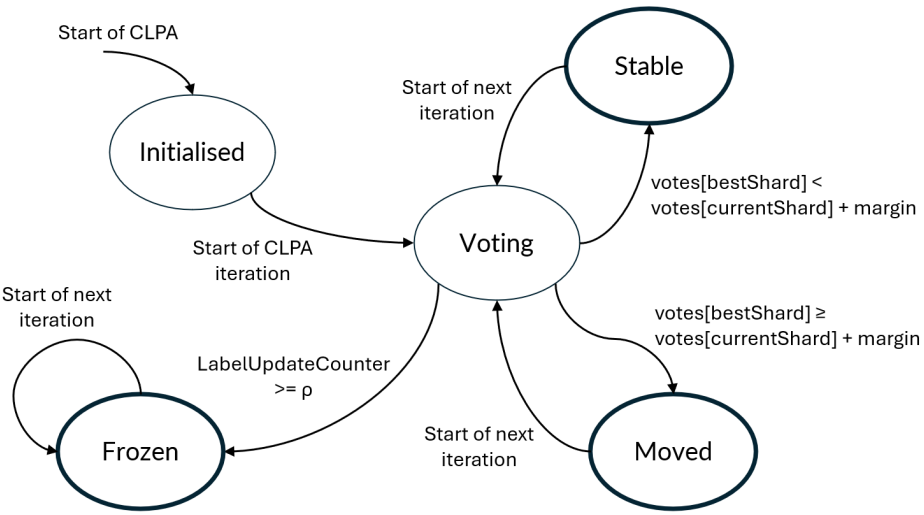


Figure 4.6 Finite state machine of vertex behaviour in CLPA with memory voting.

minimum of five iterations, to prevent premature termination due to early fluctuations. This approach allows vertex decisions to be influenced by accumulated evidence over time, rather than being overly reactive to momentary score fluctuations.

4.5 Dataset

The dataset used consists of the first three million transactions on Ethereum. This range was selected to match the dataset used in [15], allowing for a fair comparison. The first two million blocks were downloaded in comma separated value format from Xblock [63]. Then, the three million transactions were extracted from the dataset. To simulate low and high transaction arrival rates, the dataset was split into 30 epochs of 100,000 transactions (low rate) and 12 epochs of 250,000 transactions (high rate), respectively. This mirrors the arrival rates used in the simulations of paper [15].

Figure 4.7 shows a snapshot of sample transactions from the dataset. The sole transaction in block 46,402 has its *to* field set to None; however, the *toCreate* column contains a valid address, indicating smart contract creation. In such cases, *toCreate* was used in place of *to*. For the LPA implementation, only the *blockNumber*, *timestamp*, *from*, and *to* columns were retained, as they were the only relevant fields. Summary statistics are included in Appendix E for dataset insight. Notably, the implementation accounts for transactions where the sender and receiver are the same address.

blockNumber	timestamp	transactionHash	from	to	toCreate
46401	1438922533	0x78e67bdd9e0bf	0xaf880fc756	0x543807d0af	None
46402	1438922535	0x6c929e1c3d860	0xa1e4380a3b	None	0x9a049f5d18c239efaa258af9f3e7002949a977a0
46403	1438922543	0x8f798ca2400ea	0xf86a3ea807	0x7cd05927fe	None

Figure 4.7 Sample Ethereum transactions highlighting key fields used in CLPA.

5 Results and Evaluation

The test suites carried out are discussed in this chapter, along with visualisations. All tests are available in the *tests* package of the source code, with each suite organised into its own subpackage. Effort was made to write the test code in an efficient and reusable style, especially for the final test suite which has many tests.

All hyperparameters were set to the values indicated in [15], specifically: α, β = 0.5; τ = 100; ρ = 50; number of shards = 8, which was the most commonly used value in the original evaluation. This practice is followed for all subsequent tests, unless a change is explicitly mentioned. All test suites involve running some number of LPA variants for each epoch in the dataset, as if the algorithms were performing the partitioning of the blockchain. The results of each test suite were saved as comma separated value files within the same package.

5.1 Software and hardware specifications

While the tests themselves were implemented in Go, Python (v3.9.21) was employed for post-test data evaluation, taking advantage of its robust capabilities in data analysis and visualisation. All analyses were performed within a Jupyter Notebook environment. All tests were conducted on a machine equipped with an Intel i5-11600K CPU and 16 GB of 2666Mhz DDR4 RAM. The only exception was the final three-part test, which required a machine with a higher thread count due to its parallelisation demands. For this test, a system with an Intel i9-7960X 16-core, 32 thread CPU and 64GB of 3600Mhz DDR4 RAM was used.

5.2 Testing the label update mode

A simple test was set up in the *updatemode* package to test the performance of CLPA when vertex labels are updated asynchronously instead of synchronously. The partitioning solutions generated in asynchronous mode achieved a mean fitness score of 21,151 over twenty test runs across all epochs. In contrast, the synchronous mode achieved a mean fitness score of 51,863, rounded to the nearest whole number. It is very clear that asynchronous mode is superior.

Table 5.1 Mini test suite results.

	Original Score Function		New Score Function	
	8 shards	16 shards	8 shards	16 shards
Mean iterations until converge	82.3	95.2	13.1	15.7
Non-convergence rate (%)	74.0	86.0	22.7	25.3

5.3 Testing of score function with new penalty term

The new score function was first assessed against the one in [15] in a mini test suite called *Paper Penalty vs New Penalty (mini)* which can be found in the *penalty* package inside *tests*. The mini test suite is made up of just two tests, the first with 8 shards and the second with 16, both using the low transaction arrival rate. The test suite was only repeated five times, since it quickly became evident that the algorithm using the new score function converged in much fewer iterations than the original one. Specifically, using the new score function, the epochs converged within 16% of the iterations needed by the original score function, for both 8 and 16 shards. The results are shown in Table 5.1, where the non-convergence rate of the new score function is shown to be much lower. Note that the 'Mean iterations until convergence' is computed over those epochs that did converge.

It is important to note that both algorithms were allowed to run for all one hundred iterations, but the iteration at which convergence first occurred — if it occurred — was simply recorded. Based on this insight, the convergence behaviour of CLPA was further analysed to assess whether introducing an early stopping mechanism could be beneficial.

5.3.1 Testing convergence behaviour

The convergence test suite uses the low transaction arrival rate dataset and was repeated fifty times. The hypothesis was that, once convergence is reached, further iterations do not improve the partitioning fitness. To test this, CLPA was run for 500 iterations per run, recording whether there were label changes and the fitness at each step.

As an aside, the test suite revealed that both score functions exhibit non-monotonic behaviour. Figure 5.1 shows the fitness at each iteration for a representative epoch using the new score function. In this case, convergence occurs a few iterations after the 300^{th} iteration. The graph illustrates that fitness can fluctuate during iteration, an observation relevant to the discussion in Section 6.1.

In epochs where convergence was reached using the new score function, the

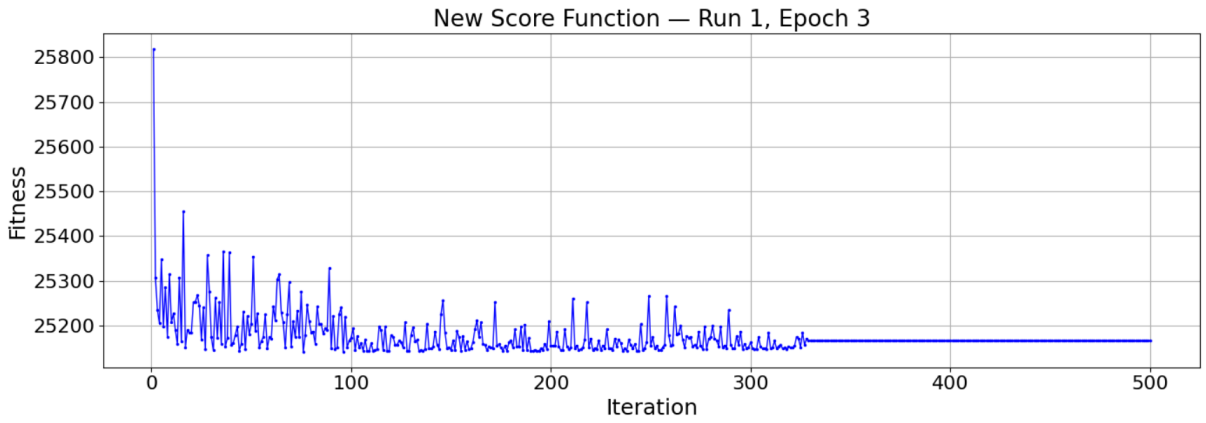


Figure 5.1 Fitness across iterations for a given epoch using the new score function.

partitioning fitness remained stable thereafter, supporting the original hypothesis. Interestingly, in a few rare cases, some vertices continued switching labels post-convergence without affecting fitness. This was likely due to the random tie-breaking of shard scores. The fitness was not affected, possibly due to the label changes being two vertices oscillating between shards. Since this had no impact on fitness, its cause was not further investigated. The original score function also displayed this behaviour in a few instances. However, in one epoch (epoch 20 of run 19): convergence was reached, some vertices continued changing labels, and fitness improved very slightly. As a result, the hypothesis is technically rejected for the original score function.

In light of these findings, the modified CLPA using the new score function is considered safe to terminate upon convergence, with confidence that continuing iterations would yield virtually no gains in partitioning quality.

5.3.2 Full evaluation of the original vs new score function

Building on insights from the convergence test suite, the *Paper Penalty vs New Penalty* test suite was run to compare both score functions in depth. The experiment was repeated fifty times to ensure statistically robust results. The version using the new score function also includes early stopping upon convergence. While this does not affect the final fitness, it enables a meaningful runtime comparison with the original algorithm from [15], which lacks this optimisation. As previous results showed faster convergence with the new score function, the benefit of early stopping is expected to be substantial.

As was previously mentioned, β is a tuneable penalty coefficient, that can take values from zero to one. This means it controls the emphasis placed on reducing cross-shard transaction workload versus maintaining balanced workloads across shards. Since this evaluation compares two score functions where β plays a key role,

the test suite varied β (0.1, 0.3, 0.5, 0.7, 0.9) while keeping α fixed at 0.5 to ensure consistent weighting of objectives in the fitness calculation. Varying β allows for a fair comparison, as each score function may have its own inherent biases.

The test suite was run with shard counts of 8, 16, and 24, which are commonly used in prior work [3, 15, 18]. While higher shard counts could offer deeper insights into scalability, they significantly increase execution time. Even with these settings, the full suite required over 31 hours to complete. The tests involving 24 shards alone consumed over 14 hours, accounting for nearly half of the total runtime. These timing results, recorded in *test_times.csv*, highlight the practical limitations imposed by project time constraints.

Figure 5.2 compares the two score functions across varying β values and shard counts. At 8 shards, the original function outperforms the new one in most cases, except when β = 0.9. However, at 16 and 24 shards, this trend reverses, with the new function consistently yielding better results across all β values. The y-axes are fixed to allow for direct visual comparison. As shard count increases, the average fitness score also rises for both functions, which is likely due to the increased number of cross-shard transactions. Notably, the performance gap between the two functions widens with higher shard counts, indicating that the new score function scales more effectively with larger numbers of shards.

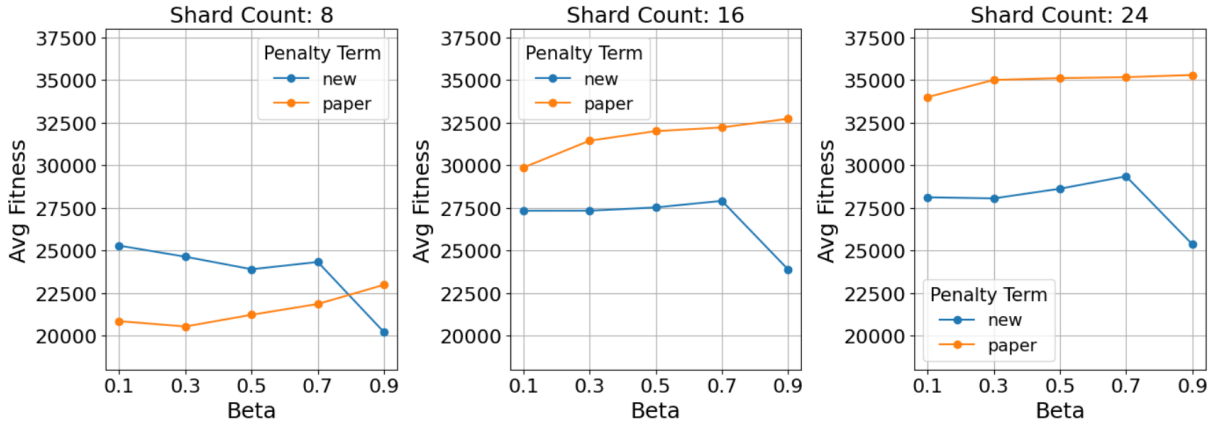


Figure 5.2 Comparison of average fitness between the original and new score functions across varying β values and shard counts.

As the experiment focuses on the internal mechanics of the score function, attention is now directed towards the two underlying objectives that inform the fitness calculation: cross-shard transactions and workload imbalance. The results, summarised in Table 5.2, report the mean values of these objectives across the entire test suite. The new score function results in higher workload imbalance, rising from 15,658 to 34,929, meaning the value obtained by the original is 44.8% that of the new score function. However, the reduction in cross-shard workload is more substantial, dropping from 43,048 to 17,307, meaning the value obtained by the new score function is only 40.2%

Table 5.2 Mean objectives under original and new score functions.

	Original score function	New score function
Mean workload imbalance	15,658	34,929
Mean cross-shard workload	43,048	17,307

of that of the original. This indicates that the proportional increase in imbalance is outweighed by the greater proportional reduction in cross-shard communication.

These results indicate that the modified function more effectively reduces cross-shard communication, consistent with the intended impact of the earlier corrective adjustment, which refined how the strength of association between vertices and shards is represented. However, the higher workload imbalance may be a consequence of the normalisation applied to the penalty term, which could diminish its effectiveness in discouraging uneven workload distribution. Nonetheless, as shown in Figure 5.3, the new score function consistently achieves better (lower) overall fitness values, particularly at high β values, indicating superior partitioning outcomes.

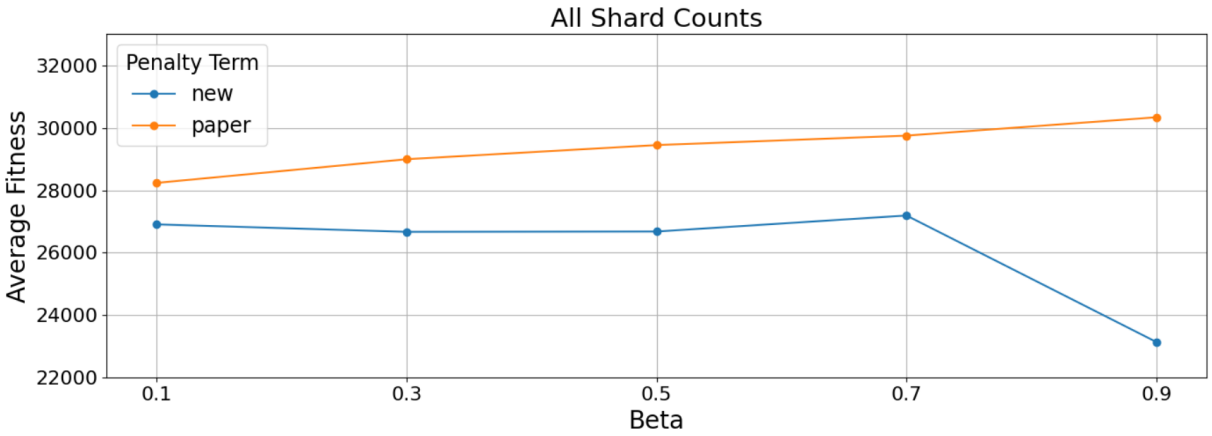


Figure 5.3 Overall average fitness between the original and new score functions.

While improvements in fitness are significant, the most substantial advantage of the new score function lies in its convergence efficiency. Across all shard counts and β values, the average time required to compute a partition for one epoch using the original setup described in [15] is 3.25 seconds. In contrast, the new score function setup completes in just 1.75 seconds on average. This represents a 46.2% reduction in runtime. As shown in Figure 5.4, the convergence distribution clearly favours the new score function, with significantly more runs being completed in fewer iterations and substantially fewer runs failing to converge.

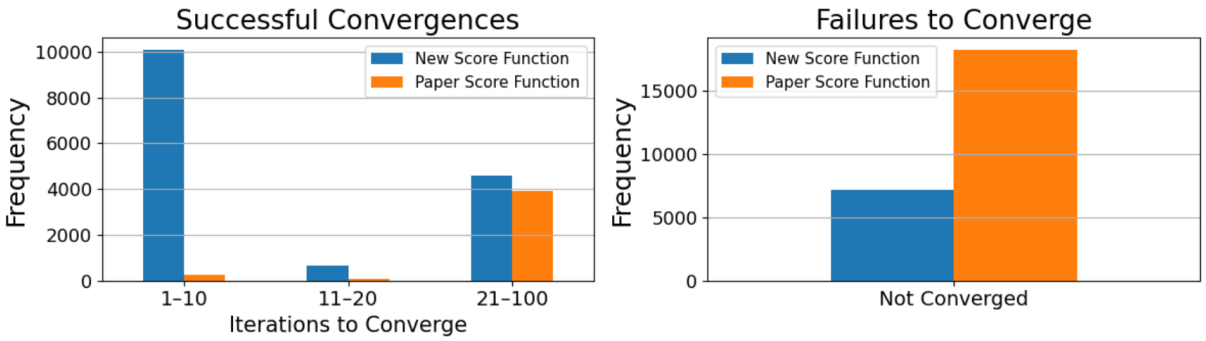


Figure 5.4 Comparison of convergence rates for original and new score functions.

5.4 Three-part comparison of LPA versions

The final evaluation is a three-part comparison between the baseline CLPA exactly as described in [15], the parallelised version of CLPA, and the final version incorporating all the modifications and improvements proposed in this project. The suite can be found in the *threepart* package. The parallel and final versions use corresponding seeds across runs to allow for direct comparison.

The test suite was run thirty times with different seeds to balance statistical robustness with practical time constraints, as in total the suite took just over 119 hours to complete (see *test_times.csv*). This duration reflects the scope of the evaluation: three algorithms tested across thirty scenarios combining various β values, transaction arrival rates (low/high), and shard counts (8, 16, 24).

5.4.1 Goroutines and execution time results

Each test launched a number of goroutines equal to half the available threads. This corresponds to 16 on the 32-thread machine described in Section 5.1. This was done to balance concurrency with system performance. According to the *Effective Go* guide [64], goroutines are multiplexed onto a pool of OS threads, allowing for lightweight concurrency. Limiting the number of concurrent goroutines helps prevent CPU saturation and excessive context switching, while giving the Go scheduler and garbage collector room to operate efficiently.

Using the previously described settings, the Parallel CLPA took 29.2% longer per epoch than the original CLPA, based on average runtimes across all runs. This value was calculated from the average epoch runtimes of the algorithms, which are shown in Table 5.3, rounded to two decimal places. This increase is expected, as 16 graph solutions are being generated in parallel rather than just one. The added time can be attributed to overheads such as synchronisation costs and increased memory usage.

Execution time increased even further when the number of goroutines was set

Table 5.3 Mean epoch runtimes under different algorithm versions.

	Paper CLPA	Parallel CLPA	Final LPA
Mean epoch runtime	7.16	9.25	6.41

to the number of available threads. This experiment is not part of the test suite in package *threepart* but is detailed in Appendix F. In summary, using 32 threads instead of 16 yielded only a marginal reduction in fitness, while significantly increasing execution time. The trade-off was deemed too costly for the limited benefit, which is why the final test suite described here spawned only 16 goroutines at a time.

5.4.2 Illustrating the benefits of parallel CLPA

Figure 5.5 shows the fitness progression over epochs for test 16, run 1, which by chance was configured with a high transaction arrival rate and a β value of 0.1. The boxplots represent the distribution of fitness values produced by the 16 parallel runs of CLPA at each epoch, showing the minimum, first quartile, median, third quartile, and maximum fitness. Overlaid on top as an orange line is the fitness of the single-threaded Paper CLPA for the same test.

The visual contrast between the boxplots and the orange line highlights the benefit of parallelisation. In nearly every epoch, the fitness achieved by the Paper CLPA is higher (worse) than the minimum fitness found among the 16 parallel runs. In some cases, such as epoch 6, the Paper CLPA's result even aligns with the maximum fitness of the parallel set, further illustrating the relative inefficiency of relying on a single run.

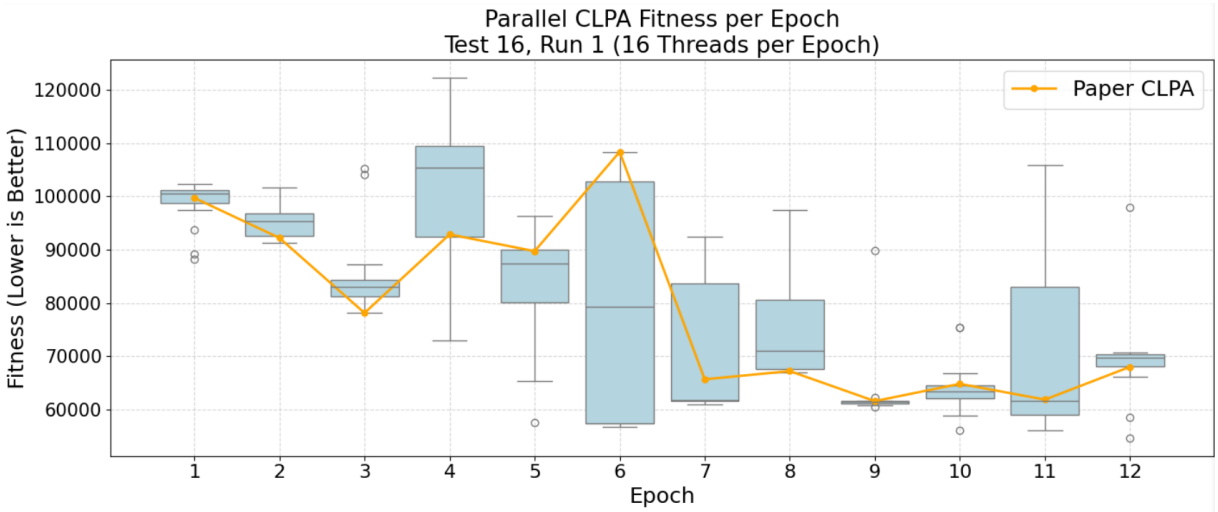


Figure 5.5 Fitness progression across epochs for parallel and single-threaded CLPA.

5.4.3 Final evaluation results and performance comparison

Figure 5.6 shows two bar charts comparing the average fitness achieved by the three algorithms across various configurations. Each chart represents a different transaction arrival rate: low and high. In all cases, lower fitness indicates better performance.

The Final LPA consistently outperforms the other algorithms. The only exceptions occur when the number of shards is 8, where the Parallel CLPA slightly outperforms the others, and under high transaction rates at 8 shards, where the original CLPA very slightly outperforms Final LPA. However, the performance advantage of the Final LPA becomes increasingly clear at higher shard counts.

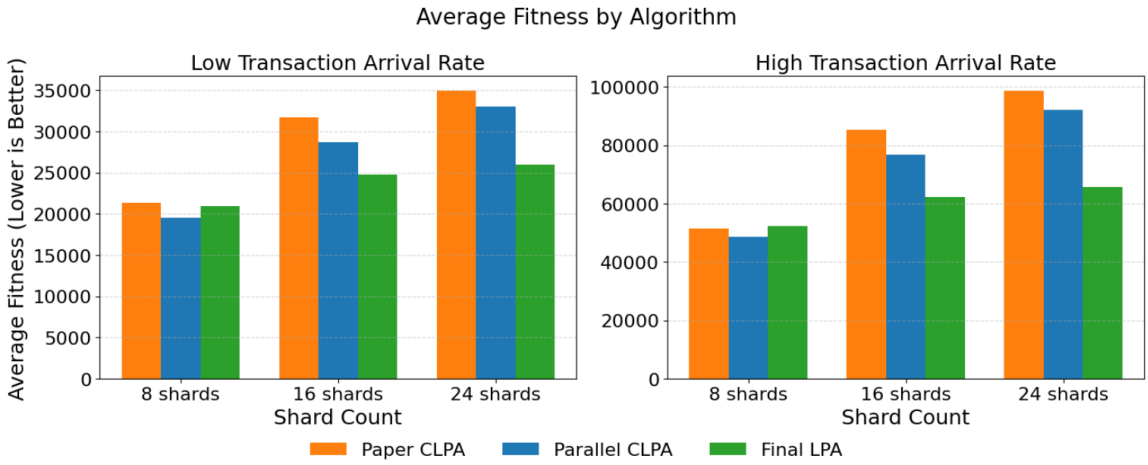


Figure 5.6 Average fitness of three algorithms under low/high transaction arrival rates.

As shown in the dumbbell chart in Figure 5.7, the final algorithm achieves a 21.2% reduction in average fitness compared to the original CLPA. Perhaps most impressively, the final algorithm not only produces better partitions but also reduces mean epoch runtime by 10.5%, as evidenced by the 6.41-second mean epoch time in Table 5.3. Therefore, while parallelisation introduces some overhead in terms of timing, this is outweighed by improvements from the new score function.

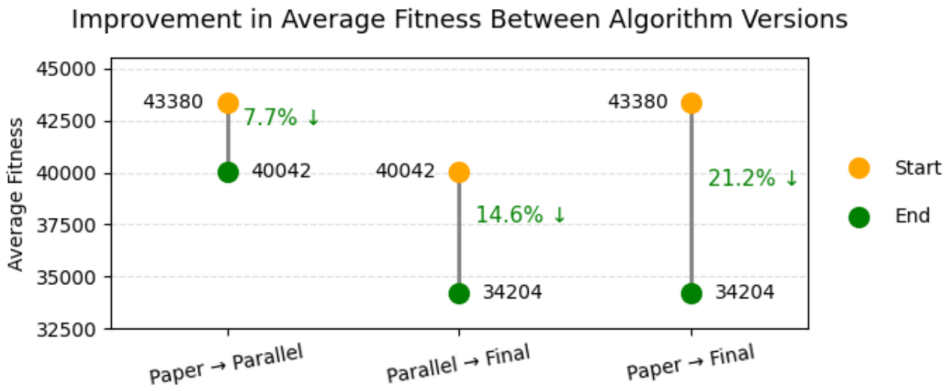


Figure 5.7 Average fitness improvements across algorithm versions.

6 Future Work and Conclusions

6.1 Future work

While significant improvements were achieved in this project, there remains considerable scope for further exploration. This section outlines several avenues that were considered but not fully pursued, primarily due to time constraints of the project. Testing certain hypotheses required experiments that run for days, making further exploration impractical within the project timeline.

Firstly, one potential enhancement involves retaining the partitioning solutions from the final few iterations of an epoch and selecting the one with the best fitness value. This would allow the algorithm to return the best recent state rather than relying solely on the final iteration. Implementing this would require additional memory to store intermediate graphs. The idea arose from the *Convergence of CLPA with Paper Penalty and New Penalty* test, which revealed that the CLPA is not monotonic. In some epochs, fitness was better (lower) at earlier iterations than at the final one, indicating that convergence on the final state might not always yield the best partitioning result.

Another potential line of investigation inspired from the convergence tests involves understanding why some vertices continue switching shard labels even when the overall fitness remains unchanged, thereby preventing convergence. The evaluation provided plausible explanations for this phenomenon, but these remain speculative. A deeper analysis of why that behaviour occurred could yield insights for improving algorithm stability or convergence guarantees.

To further improve execution time, the parallel algorithm could be adjusted to stop once a few goroutines have completed, instead of waiting for all of them to finish. This would significantly speed up processing, as the current setup is delayed even if one goroutine takes all one hundred iterations. While this might slightly reduce the quality of results, it offers a practical trade-off between speed and partitioning fitness.

Moreover, investigating higher shard counts would be valuable, as this project was limited to a maximum of 24 due to time constraints. Results showed that the final LPA version outperformed the baseline more as shard count increased, suggesting better scalability. Further testing could confirm if this trend continues.

Finally, a promising direction for future work is the use of machine learning to guide shard allocation, particularly through graph-based models like Graph Neural Networks. Unlike heuristic methods like LPA, these models can learn from historical transaction data to predict shard assignments that reduce cross-shard interactions and balance workloads. This approach could adapt to evolving patterns and improve performance over time, especially in dynamic blockchain environments.

6.2 Conclusions

This project set out to enhance shard allocation strategies in account-based blockchain systems by building upon the CLPA and addressing its core limitations. The ultimate goal was to develop a more efficient, stable, and scalable algorithm for assigning accounts to shards in a way that minimises cross-shard transactions and balances workload across the network. Through the careful design and rigorous evaluation of several enhancements including: a novel shard score function, parallel execution, convergence-based stopping criteria, and a memory-based label voting mechanism; the project successfully achieved this aim.

One of the most significant accomplishments of this work is the 21.2% improvement in average partitioning fitness achieved by the final version of the proposed algorithm over the original CLPA described in Transformers [15]. This metric captures the combined goal of reducing cross-shard workload while maintaining balance, and the improvement indicates a substantial advancement in the effectiveness of the shard allocation strategy. Just as noteworthy, the final algorithm achieved this better quality result while also reducing mean epoch runtime by 10.5%.

The introduction of a new penalty term in the score function was pivotal in achieving these results. It addressed a key flaw in the original formulation, where penalties could become negative and unintentionally favour less suitable shard assignments, hindering fitness and converging much slower. Another part of the score function redesign was the removal of an ineffective normalisation term. This further simplified computation without affecting accuracy. The revised score function was more intuitive and ensured a smoother relationship between shard workloads and penalty impact. This led to improved convergence behaviour, and during experimentation, the new score function mostly outperformed the original across the tested parameter configurations.

Another core innovation was the parallelisation of CLPA, which tackled the randomness inherent in the baseline algorithm by running multiple variants concurrently and selecting the best partitioning solution. Though this approach introduced a 29.2% increase in execution time compared to the baseline when run with 16 goroutines, it paid dividends in partition quality. The final version of the proposed algorithm included early stopping and memory-based voting. It was even able to outperform this parallel version in both speed and fitness in most scenarios, especially at higher shard counts where scalability is most needed.

The project was grounded in scientific rigour, following a cycle of hypothesis formation, testing, and refinement. Detailed convergence tests showed that the CLPA algorithm is not monotonic; fitness can fluctuate across iterations. The project also uncovered and analysed unexpected behaviours, such as non-convergence due to label

oscillations without fitness degradation. These observations informed design choices, like memory-based label voting to improve stability.

Despite these accomplishments, there were scenarios where the final version of the algorithm did not outperform the baseline. Notably, when the number of shards was set to 8, the baseline sometimes achieved slightly better fitness. Nonetheless, the parallel version of CLPA still beat the baseline. While this reinforces the value of the improvements for real-world, high-volume blockchain systems, it also reflects a balanced and honest evaluation of the work's limitations.

In summary, this project delivered on its primary aim: to significantly improve the performance and practicality of shard allocation strategies in account-based blockchain sharding using enhanced LPA-based methods. The final algorithm is faster, deterministic, and yields higher-quality shard assignments. Along the way, the project led to a deeper understanding of the nuanced behaviour of label propagation techniques in transaction networks, uncovered meaningful insights into score function design, and proposed a scalable architecture for parallel experimentation. These contributions serve as a foundation for further research and practical deployment in scalable blockchain systems.

References

[1] K. Croman et al., "On scaling decentralized blockchains," in *Financial Cryptography and Data Security*, J. Clark, S. Meiklejohn, P. Y. Ryan, D. Wallach, M. Brenner, and K. Rohloff, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 106–125, ISBN: 978-3-662-53357-4. DOI: 10.1007/978-3-662-53357-4_8.

[2] Y. Li, J. Wang, and H. Zhang, "A survey of state-of-the-art sharding blockchains: Models, components, and attack surfaces," *Journal of Network and Computer Applications*, vol. 217, p. 103 686, 2023, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2023.103686. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804523001054.

[3] H. Huang et al., "Brokerchain: A cross-shard blockchain protocol for account/balance-based state sharding," in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, 2022, pp. 1968–1977. DOI: 10.1109/INFOCOM48880.2022.9796859.

[4] Y. Zhang, S. Pan, and J. Yu, "Txallo: Dynamic transaction allocation in sharded blockchain systems," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 721–733. DOI: 10.1109/ICDE55515.2023.00390.

[5] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, Accessed: January 2025, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf.

[6] N. Okanami, R. Nakamura, and T. Nishide, "Load balancing for sharded blockchains," in *Financial Cryptography and Data Security: FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers*, Kota Kinabalu, Malaysia: Springer-Verlag, 2020, pp. 512–524, ISBN: 978-3-030-54454-6. DOI: 10.1007/978-3-030-54455-3_36. [Online]. Available: https://doi.org/10.1007/978-3-030-54455-3_36.

[7] S. Xu et al., "A sharding scheme based on graph partitioning algorithm for public blockchain," *Computer Modeling in Engineering & Sciences*, vol. 139, no. 3, pp. 3311–3327, 2024, ISSN: 1526-1506. DOI: 10.32604/cmes.2023.046164. [Online]. Available: http://www.techscience.com/CMES/v139n3/55633.

[8] J. Wang and H. Wang, "Monoxide: Scale out blockchains with asynchronous consensus zones," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA: USENIX Association, Feb. 2019, pp. 95–112, ISBN: 978-1-931971-49-2. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/wang-jiaping.

[9] Z. Hong, S. Guo, and P. Li, "Scaling blockchain via layered sharding," *IEEE Journal on Selected Areas in Communications*, vol. 40, no. 12, pp. 3575–3588, 2022. DOI: 10.1109/JSAC.2022.3213350.

[10] G. Wood, *Ethereum: A secure decentralised generalised transaction ledger*, Shanghai Version efc5f9a – 2025-02-04, 2025. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf.

[11] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19, Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 123–140, ISBN: 9781450356435. DOI: 10.1145/3299869.3319889. [Online]. Available: https://doi.org/10.1145/3299869.3319889.

[12] C. T. Nguyen, D. T. Hoang, D. N. Nguyen, Y. Xiao, D. Niyato, and E. Dutkiewicz, "Metashard: A novel sharding blockchain platform for metaverse applications," *IEEE Transactions on Mobile Computing*, vol. 23, no. 5, pp. 4348–4361, 2024. DOI: 10.1109/TMC.2023.3290955.

[13] H. Chen and Y. Wang, "Sschain: A full sharding protocol for public blockchain without data migration overhead," *Pervasive and Mobile Computing*, vol. 59, p. 101 055, 2019, ISSN: 1574-1192. DOI: https://doi.org/10.1016/j.pmcj.2019.101055. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574119218306370.

[14] X. Wang, W. Wang, Y. Zeng, T. Yang, and C. Zheng, "A state sharding model on the blockchain," *Cluster Computing*, vol. 25, no. 3, pp. 1969–1979, 2022, ISSN: 1573-7543. DOI: 10.1007/s10586-022-03578-3. [Online]. Available: https://doi.org/10.1007/s10586-022-03578-3.

[15] C. Li et al., "Achieving scalability and load balance across blockchain shards for state sharding," in *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*, 2022, pp. 284–294. DOI: 10.1109/SRDS55811.2022.00034.

[16] J. Xi et al., "A blockchain dynamic sharding scheme based on hidden markov model in collaborative iot," *IEEE Internet of Things Journal*, vol. 10, no. 16, pp. 14 896–14 907, 2023. DOI: 10.1109/JIOT.2023.3294234.

[17] Y. Liu et al., "Building blocks of sharding blockchain systems: Concepts, approaches, and open problems," *Computer Science Review*, vol. 46, p. 100 513, 2022, ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2022.100513. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013722000478.

[18] H. Han, S. Chen, Z. Xu, X. Dong, and W. Tian, “Gpchain: Optimizing cross-shard transactions and load imbalance in sharded blockchain networks,” in *Internet of Things – ICIOT 2023*, K. Ye and L.-J. Zhang, Eds., Cham: Springer Nature Switzerland, 2024, pp. 31–46, ISBN: 978-3-031-51734-1.

[19] S. Mssassi and A. Abou El Kalam, “The blockchain trilemma: A formal proof of the inherent trade-offs among decentralization, security, and scalability,” *Applied Sciences*, vol. 15, no. 1, 2025, ISSN: 2076-3417. DOI: 10.3390/app15010019. [Online]. Available: https://www.mdpi.com/2076-3417/15/1/19.

[20] Y. Fu et al., “Quantifying the blockchain trilemma: A comparative analysis of algorand, ethereum 2.0, and beyond,” in *2024 IEEE International Conference on Metaverse Computing, Networking, and Applications (MetaCom)*, 2024, pp. 97–104. DOI: 10.1109/MetaCom62920.2024.00028.

[21] J. Muthemba, *Understanding the blockchain trilemma: Balancing scalability, security, and decentralization*, Accessed: January 2025, 2023. [Online]. Available: https://medium.com/@jordanmuthemba/understanding-the-blockchain-trilemma-balancing-scalability-security-and-decentralization-1e763c261bf3.

[22] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*, 4th ed. Cham: Springer, 2020, ISBN: 978-3-030-26252-5. DOI: 10.1007/978-3-030-26253-2.

[23] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, “A secure sharding protocol for open blockchains,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16, Vienna, Austria: Association for Computing Machinery, 2016, pp. 17–30, ISBN: 9781450341394. DOI: 10.1145/2976749.2978389. [Online]. Available: https://doi.org/10.1145/2976749.2978389.

[24] X. Cai et al., “A sharding scheme-based many-objective optimization algorithm for enhancing security in blockchain-enabled industrial internet of things,” *IEEE Transactions on Industrial Informatics*, vol. 17, no. 11, pp. 7650–7658, 2021. DOI: 10.1109/TII.2021.3051607.

[25] Z. Zhang et al., “A community detection-based blockchain sharding scheme,” in *Blockchain – ICBC 2022*, S. Chen, R. K. Shyamasundar, and L.-J. Zhang, Eds., Cham: Springer Nature Switzerland, 2022, pp. 78–91, ISBN: 978-3-031-23495-8. DOI: 10.1007/978-3-031-23495-8_6.

[26] P. Li et al., “Spring: Improving the throughput of sharding blockchain via deep reinforcement learning based state placement,” in *Proceedings of the ACM Web Conference 2024*, ser. WWW ’24, Singapore, Singapore: Association for Computing Machinery, 2024, pp. 2836–2846, ISBN: 9798400701719. DOI:

10.1145/3589334.3645386. [Online]. Available: https://doi.org/10.1145/3589334.3645386.

[27] S. Souravlas, A. Sifaleras, M. Tsintogianni, and S. Katsavounis, "A classification of community detection methods in social networks: A survey," *International Journal of General Systems*, vol. 50, pp. 63–91, Jan. 2021. DOI: 10.1080/03081079.2020.1863394.

[28] J. Li et al., "A comprehensive review of community detection in graphs," *Neurocomputing*, vol. 600, p. 128 169, 2024, ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2024.128169. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231224009408.

[29] K. Andreev and H. Racke, "Balanced graph partitioning," *Theory of Computing Systems*, vol. 39, no. 6, pp. 929–939, 2006, ISSN: 1433-0490. DOI: 10.1007/s00224-006-1350-7. [Online]. Available: https://doi.org/10.1007/s00224-006-1350-7.

[30] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "Hdrf: Stream-based partitioning for power-law graphs," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, ser. CIKM '15, Melbourne, Australia: Association for Computing Machinery, 2015, pp. 243–252, ISBN: 9781450337946. DOI: 10.1145/2806416.2806424. [Online]. Available: https://doi.org/10.1145/2806416.2806424.

[31] M. Kim and K. S. Candan, "Sbv-cut: Vertex-cut based graph partitioning using structural balance vertices," *Data & Knowledge Engineering*, vol. 72, pp. 285–303, 2012, ISSN: 0169-023X. DOI: https://doi.org/10.1016/j.datak.2011.11.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169023X11001480.

[32] V. Buterin, "Ethereum: Platform review – opportunities and challenges for private and consortium blockchains," R3, Tech. Rep., 2016, Accessed: January 2025. [Online]. Available: https://www.smallake.kr/wp-content/uploads/2016/06/314477721-Ethereum-Platform-Review-Opportunities-and-Challenges-for-Private-and-Consortium-Blockchains.pdf.

[33] V. Buterin, *A simple explanation of ethereum sharding*, Accessed: January 2025, 2021. [Online]. Available: https://vitalik.eth.limo/general/2021/04/07/sharding.html.

[34] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 583–598. DOI: 10.1109/SP.2018.000-5.

[35] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: Association for Computing Machinery, 2018, pp. 931–948, ISBN: 9781450356930. DOI: 10.1145/3243734.3243853. [Online]. Available: https://doi.org/10.1145/3243734.3243853.

[36] M. Zhang, J. Li, Z. Chen, H. Chen, and X. Deng, "Cycledger: A scalable and secure parallel protocol for distributed ledger via sharding," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 358–367. DOI: 10.1109/IPDPS47924.2020.00045.

[37] C. Huang et al., "Repchain: A reputation-based secure, fast, and high incentive blockchain system via sharding," *IEEE Internet of Things Journal*, vol. 8, no. 6, pp. 4291–4304, 2021. DOI: 10.1109/JIOT.2020.3028449.

[38] L. N. Nguyen, T. D. T. Nguyen, T. N. Dinh, and M. T. Thai, "Optchain: Optimal transactions placement for scalable blockchain sharding," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 525–535. DOI: 10.1109/ICDCS.2019.00059.

[39] X. Wang, B. Li, L. Jia, and Y. Sun, "Orbit: A dynamic account allocation mechanism in sharding blockchain system," in *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*, 2024, pp. 333–344. DOI: 10.1109/ICDCS60910.2024.00039.

[40] X. Ao, Y. Liu, Z. Qin, Y. Sun, and Q. He, "Temporal high-order proximity aware behavior analysis on ethereum," *World Wide Web*, vol. 24, no. 5, pp. 1565–1585, 2021, ISSN: 1573-1413. DOI: 10.1007/s11280-021-00875-6. [Online]. Available: https://doi.org/10.1007/s11280-021-00875-6.

[41] Wikimedia Commons contributors, *Social network diagram (large) [svg image]*, Accessed: January 2025, 2008. [Online]. Available: https://commons.wikimedia.org/wiki/File:Social_Network_Diagram_%28large%29.svg.

[42] W. Fan et al., "Application driven graph partitioning," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20, Portland, OR, USA: Association for Computing Machinery, 2020,

pp. 1765–1779, ISBN: 9781450367356. DOI: 10.1145/3318464.3389745. [Online]. Available: https://doi.org/10.1145/3318464.3389745.

[43] J. Ugander and L. Backstrom, "Balanced label propagation for partitioning massive graphs," in *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, ser. WSDM '13, Rome, Italy: Association for Computing Machinery, 2013, pp. 507–516, ISBN: 9781450318693. DOI: 10.1145/2433396.2433461. [Online]. Available: https://doi.org/10.1145/2433396.2433461.

[44] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998. DOI: 10.1137/S1064827595287997. eprint: https://doi.org/10.1137/S1064827595287997. [Online]. Available: https://doi.org/10.1137/S1064827595287997.

[45] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," Department of Computer Science, University of Minnesota, Tech. Rep. TR 98-017, 1998, Accessed: January 2025. [Online]. Available: https://conservancy.umn.edu/items/2f610239-590c-45c0-bcd6-321036aaad56.

[46] D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*. Cambridge University Press, 2011, ISBN: 9780521195270. [Online]. Available: https://www.cambridge.org/core/books/design-of-approximation-algorithms/88E0AEAEFF2382681A103EEA572B83C6.

[47] V. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics Theory and Experiment*, vol. 2008, Apr. 2008. DOI: 10.1088/1742-5468/2008/10/P10008.

[48] Z. Zhen, X. Wang, H. Lin, S. Garg, P. Kumar, and M. S. Hossain, "A dynamic state sharding blockchain architecture for scalable and secure crowdsourcing systems," *Journal of Network and Computer Applications*, vol. 222, p. 103 785, 2024, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2023.103785. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804523002047.

[49] A. Mizrahi and O. Rottenstreich, "Blockchain state sharding with space-aware representations," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1571–1583, 2021. DOI: 10.1109/TNSM.2020.3031355.

[50] X. Zhu and Z. Ghahramani, "Learning from labeled and unlabeled data with label propagation," Center for Automated Learning and Discovery, Carnegie Mellon University, Tech. Rep. CMU-CALD-02-107, 2002. [Online]. Available: https://mlg.eng.cam.ac.uk/zoubin/papers/CMU-CALD-02-107.pdf.

[51] N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical review. E, Statistical, nonlinear, and soft matter physics*, vol. 76, p. 036 106, Oct. 2007. DOI: 10.1103/PhysRevE.76.036106.

[52] I. Ben El Kouni, W. Karoui, and L. B. Romdhane, "Node importance based label propagation algorithm for overlapping community detection in networks," *Expert Systems with Applications*, vol. 162, p. 113 020, 2020, ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2019.113020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417419307377.

[53] ResearchGate, *An example of the label propagation algorithm [figure]*, Accessed: January 2025, 2020. [Online]. Available: https://www.researchgate.net/figure/An-example-of-the-Label-Propagation-Algorithm_fig2_340627329.

[54] A. M. Fiscarelli, M. R. Brust, G. Danoy, and P. Bouvry, "Local memory boosts label propagation for community detection," *Applied Network Science*, vol. 4, no. 1, p. 95, 2019, ISSN: 2364-8228. DOI: 10.1007/s41109-019-0210-8. [Online]. Available: https://doi.org/10.1007/s41109-019-0210-8.

[55] J. Xie and B. K. Szymanski, "Community detection using a neighborhood strength driven label propagation algorithm," in *2011 IEEE Network Science Workshop*, 2011, pp. 188–195. DOI: 10.1109/NSW.2011.6004645.

[56] L. Jia, Y. Liu, K. Wang, and Y. Sun, "Estuary: A low cross-shard blockchain sharding protocol based on state splitting," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 3, pp. 405–420, 2024. DOI: 10.1109/TPDS.2024.3351632.

[57] A. Wu and C. Hua, *Alb-chain:achieving scalability and load balance blockchain sharding*, Jul. 2024. DOI: 10.21203/rs.3.rs-4823614/v1.

[58] M. J. Barber and J. W. Clark, "Detecting network communities by propagating labels under constraints," *Phys. Rev. E*, vol. 80, p. 026 129, 2 Aug. 2009. DOI: 10.1103/PhysRevE.80.026129. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.80.026129.

[59] J. H. Chin and K. Ratnavelu, "Detecting community structure by using a constrained label propagation algorithm," *PLOS ONE*, vol. 11, no. 5, e0155320, 2016, ISSN: 1932-6203. DOI: 10.1371/journal.pone.0155320. [Online]. Available: https://doi.org/10.1371/journal.pone.0155320.

[60] X. Huang et al., “Contribchain: A stress-balanced blockchain sharding protocol with node contribution awareness,” in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Accepted for presentation at IEEE INFOCOM 2025, 2025. [Online]. Available: https://arxiv.org/abs/2505.06899.

[61] Haahr, Mads, *Random.org: True random number service*, Accessed: January 2025, 2025. [Online]. Available: https://www.random.org/.

[62] The Go Authors, *The go programming language documentation*, Accessed: January 2025, 2025. [Online]. Available: https://go.dev/doc/.

[63] P. Zheng, Z. Zheng, J. Wu, and H.-N. Dai, “Xblock-eth: Extracting and exploring blockchain data from ethereum,” *IEEE Open Journal of the Computer Society*, vol. 1, pp. 95–106, 2020. DOI: 10.1109/OJCS.2020.2990458.

[64] The Go Authors, *Effective go*, Accessed: January 2025, 2024. [Online]. Available: https://go.dev/doc/effective_go.

Appendix A Proof of Ineffective Normalisation Term

Let $a_1, a_2, \ldots, a_n$ be a finite sequence of strictly positive real numbers, and suppose these values are ordered such that

$$a_{\pi(1)} < a_{\pi(2)} < \cdots < a_{\pi(n)}$$

for some permutation π of the index set $\{1, 2, \ldots, n\}$. Let $x > 0$ be an arbitrary fixed positive constant. Consider the transformed sequence obtained by dividing each element by x:

$$\frac{a_{\pi(1)}}{x}, \frac{a_{\pi(2)}}{x}, \ldots, \frac{a_{\pi(n)}}{x}.$$

The aim is to show that the ordering of the original sequence is preserved under this transformation, i.e.,

$$\frac{a_{\pi(1)}}{x} < \frac{a_{\pi(2)}}{x} < \cdots < \frac{a_{\pi(n)}}{x}.$$

Since $x > 0$, the operation $t \mapsto \frac{t}{x}$ is strictly increasing for all $t > 0$. Therefore, for any pair $i, j \in \{1, \ldots, n\}$ such that $a_i < a_j$, it follows that

$$\frac{a_i}{x} < \frac{a_j}{x}.$$

Consequently, the relative ordering of the elements remains unchanged after division by x. This implies that any `argmax` or shard selection based on these scores will yield the same result before and after such a normalisation.

In the context of the score function used in the CLPA algorithm, this property confirms that dividing the association strength component by the total incident edge weight of a vertex (a constant specific to that vertex and common to all candidate shards) has no effect on the outcome of the label propagation. Since shard assignment decisions depend only on the relative ordering of shard scores for each vertex, this normalisation term can be omitted without altering the result. Hence, it is formally redundant.

Appendix B Fitness Calculation

This appendix provides the working to calculate the fitness using $F(\boldsymbol{x})$.

The fitness function is defined as a weighted sum of two components: $C(\boldsymbol{x})$, which quantifies the cross-shard communication cost, and $D(\boldsymbol{x})$, which measures the load imbalance across shards. Hence, both $C(\boldsymbol{x})$ and $D(\boldsymbol{x})$ must be explicitly calculated.

As mentioned before: $\alpha = 0.5$ and $(1 - \alpha) = 0.5$.

$$F(\boldsymbol{x}) \;=\; \alpha \cdot C(\boldsymbol{x}) + (1-\alpha) \cdot D(\boldsymbol{x})$$

$$C(\boldsymbol{x}) \;=\; \textstyle\sum_{\text{cross-shard edges } (i,j)} e_{i,j} \;=\; 1 + 2 + 10 \;=\; 13$$

$$D(\boldsymbol{x}) \;=\; \max_{k \in [K]} \left| L_k(\boldsymbol{x}) - \hat{L}(\boldsymbol{x}) \right|$$

To calculate $D(\boldsymbol{x})$, the load on each shard must be identified:

$$L_1(\boldsymbol{x}) \;=\; 1$$

$$L_2(\boldsymbol{x}) \;=\; 14$$

$$L_3(\boldsymbol{x}) \;=\; 10$$

$$L_4(\boldsymbol{x}) \;=\; 10$$

Now, $\hat{L}(\boldsymbol{x}) \;=\; \frac{1+14+10+10}{4} = \frac{35}{4} \;=\; 8.75$, and so:

$$|L_1 - \hat{L}(\boldsymbol{x})| \;=\; |1 - 8.75| \;=\; 7.75\,,$$

$$|L_2 - \hat{L}(\boldsymbol{x})| \;=\; |14 - 8.75| \;=\; 5.25\,,$$

$$|L_3 - \hat{L}(\boldsymbol{x})| \;=\; |10 - 8.75| \;=\; 1.25\,,$$

$$|L_4 - \hat{L}(\boldsymbol{x})| \;=\; |10 - 8.75| \;=\; 1.25$$

Therefore, $D(\boldsymbol{x}) \;=\; \max(7.75, 5.25, 1.25, 1.25) \;=\; 7.75$

And finally, $F(\boldsymbol{x}) \;=\; 0.5 \cdot C(\boldsymbol{x}) + 0.5 \cdot D(\boldsymbol{x}) \;=\; 0.5 \cdot 13 + 0.5 \cdot 7.75 \;=\; \boxed{10.375}$

Appendix C Random Seeds Generation

A total of fifty thousand random seeds were generated using the website *random.org* to ensure sufficient coverage for all test suites. These seeds were saved in a file named *seeds.csv*, which is included in the *mylpa* package to support reproducibility across experimental runs. A screenshot of the generation settings used on the website is shown in Figure C.1.

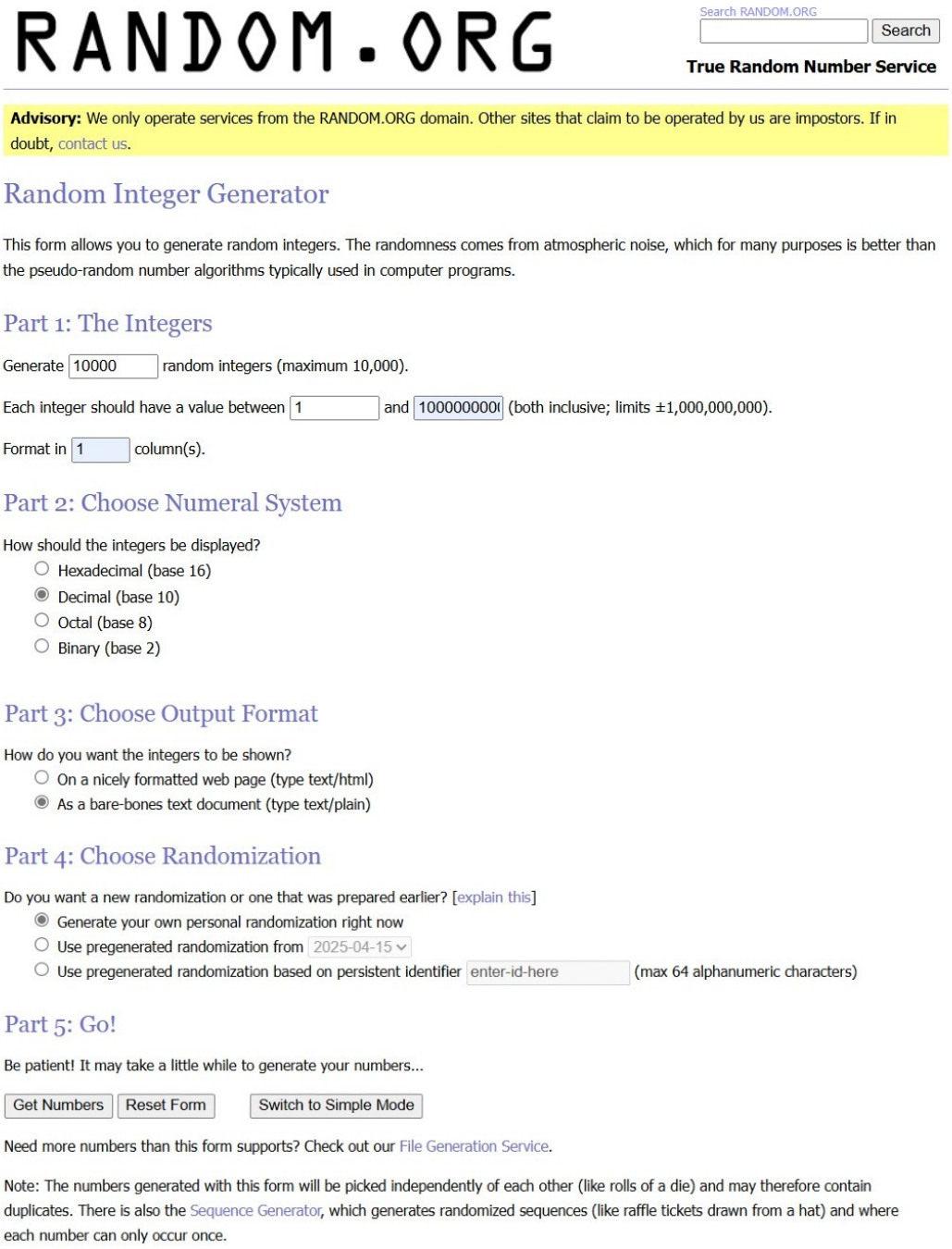


Figure C.1 Screenshot of settings for random generation of seeds.

Appendix D Active Nodes List Optimisation

The 'active nodes list' approach is illustrated in Listing D.1. The *runClpa* and *clpaIteration* functions were modified to implement this optimisation, as clearly indicated by the in-line comments. The supporting function *isPassive*, also shown in the listing, was responsible for identifying passive vertices, meaning those whose label matched all their neighbours and thus would not change label.

While the idea was to reduce computation by skipping such vertices, it turned out that the time saved by avoiding updates to passive nodes was offset by the overhead required to continually track and re-evaluate node passivity. In the specific context of this project's blockchain dataset, this resulted in no significant runtime improvement. Consequently, the optimisation was excluded from the final version proposed in this work.

```
// The CLPA function
func runClpa(alpha float64, beta float64, tau int, rho int, graph *shared.Graph, randomGen *rand.Rand, seed int64) *shared.EpochResult {

  // Map to store vertices that may change their label on the next run
  pendingVertices := make(map[string]struct{})

  // Add each vertex to the pending list initially
  for id := range graph.Vertices {
    pendingVertices[id] = struct{}{}
  }

  // Ensure all vertices have initialised LabelVotes
  for _, vertex := range graph.Vertices {
    if vertex.LabelVotes == nil {
      vertex.LabelVotes = make(map[int]int)
      vertex.LabelVotes[vertex.Label] = 1 // Give an initial vote for current label
    }
  }

  convergenceIter := -1 // Default value showing no convergence

  minIterations := 5 // Run at least 5 iterations before checking convergence

  // Carry out CLPA iterations
  for iter := 0; iter < tau; iter++ {

    // create a map with all old labels - meaning labels of vertices before current CLPA iteration
```

```
28      oldLabels := make(map[string]int)
29      for id, vertex := range graph.Vertices {
30        oldLabels[id] = vertex.Label
31      }
32
33      // Perform an iteration of CLPA while keeping track of which vertices
    are pending
34      clpaIteration(graph, beta, randomGen, rho, pendingVertices)
35
36      // CLPA iterations should stop once convergence is reached
37
38      // Set flag to show convergencence occurred unless a label changes
39      converged := true
40
41      // Iterate through all vertices
42      for id, vertex := range graph.Vertices {
43
44        // Check if any vertex changed its label, and adjust flag if so
45        if oldLabels[id] != vertex.Label {
46          converged = false
47          break
48        }
49      }
50
51      if converged && iter+1 >= minIterations {
52        convergenceIter = iter + 1
53        break
54      }
55    }
56
57    // Calculate the workload imbalance, number of cross shard transactions
    and fitness of the partitioning
58    workloadImbalance, crossShardWorkload, fitness := shared.
   CalculateFitness(graph, alpha)
59
60    // Return the results of the epoch
61    return &shared.EpochResult{
62      Seed:               seed,
63      Fitness:            fitness,
64      WorkloadImbalance:  workloadImbalance,
65      CrossShardWorkload: crossShardWorkload,
66      ConvergenceIter:    convergenceIter,
67    }
68
69  }
70
71
```

```
72  // The function that performs an iteration through all vertices and
     assigns shards
73  func clpaIteration(graph *shared.Graph, beta float64, randomGen *rand.
     Rand, rho int, pendingVertices map[string]struct{}) {
74
75    // Get a random order to use for this CLPA iteration
76    sortedVertices := setVerticesOrder(graph, randomGen)
77
78    // Iterate through each vertex in some order
79    for _, vertex := range sortedVertices {
80
81      // Skip this vertex if it is not in the pending list (non-passive)
82      if _, isPending := pendingVertices[vertex.ID]; !isPending {
83        continue
84      }
85
86      // Calculate the score of shards with respect to current vertex
87      scores := calculateScores(graph, vertex, beta)
88
89      // Get the ID of the best shard with respect to current vertex
90      bestShard := getBestShard(scores, randomGen)
91
92      // Instead of moving immediately, add a vote
93      vertex.LabelVotes[bestShard]++
94
95      // Find the label with the most votes
96      winningShard, maxVotes := vertex.Label, vertex.LabelVotes[vertex.
    Label]
97      for shard, votes := range vertex.LabelVotes {
98        if votes > maxVotes {
99          winningShard = shard
100         maxVotes = votes
101       }
102     }
103
104     // If winning shard is different and has enough dominance, then move
105     voteMargin := 1 // need at least 1 more vote than current label
106     if winningShard != vertex.Label && (vertex.LabelVotes[winningShard]-
    vertex.LabelVotes[vertex.Label] >= voteMargin) {
107       moveVertex(graph, vertex, winningShard, rho)
108     }
109
110     // Check if vertex has become passive, if so delete from
    pendingVertices, otherwise add it
111     if isPassive(vertex, graph) {
112       delete(pendingVertices, vertex.ID)
113     } else {
```

```
114         pendingVertices[vertex.ID] = struct{}{}
115       }
116
117       // Check if neighbours of current vertex have become passive
118       for neighborID := range vertex.Edges {
119         neighbor := graph.Vertices[neighborID]
120
121         // If neighbouring vertex has become passive, delete it from the
      pending vertices if it is there
122         // Else, add it to the pending vertices (it is overwritten if it is
       present already)
123         if isPassive(neighbor, graph) {
124           delete(pendingVertices, neighborID)
125         } else {
126           pendingVertices[neighborID] = struct{}{}
127         }
128       }
129     }
130   }
131
132
133   // Function to check if vertex is passive
134   func isPassive(vertex *shared.Vertex, graph *shared.Graph) bool {
135     for neighborID := range vertex.Edges {
136       if graph.Vertices[neighborID].Label != vertex.Label {
137         return false
138       }
139     }
140     return true
141   }
```

Listing D.1 CLPA algorithm implementation

Appendix E Dataset Summary Statistics

The tables provide summary statistics for each epoch in the two transaction arrival rate scenarios used during testing. Table E.1 presents metrics for the high transaction arrival rate configuration, consisting of 12 epochs with 250,000 transactions each. Table E.2 corresponds to the low transaction arrival rate configuration, where the dataset was split into 30 epochs of 100,000 transactions each.

For each epoch, the number of total vertices, active and inactive vertices, total number of edges, and the number of self-loops are reported. An active vertex is one that participated in at least one transaction during the epoch, whereas inactive vertices were present in the graph but had no transaction activity. The edges represent distinct account-to-account interactions, while self-loops occur when a transaction involves the same sender and receiver. These statistics offer insight into the evolving structure and density of the state graph across epochs under different transaction volumes.

Table E.1 Summary statistics per epoch for high transaction arrival rate.

Epoch	Vertices	Active Vertices	Inactive Vertices	Edges	SelfLoops
1	15467	15467	0	27973	918
2	23318	10910	12408	15793	24
3	29025	9132	19893	15082	1914
4	32635	6784	25851	11328	17
5	37476	9081	28395	14113	24
6	49736	18381	31355	26419	65
7	64534	24068	40466	34480	103
8	80761	26164	54597	37071	1306
9	104492	37019	67473	51822	115
10	119111	30152	88959	45204	97
11	128434	24340	104094	37256	51
12	139494	28199	111295	43511	63

Table E.2 Summary statistics per epoch for low transaction arrival rate.

Epoch	Vertices	Active Vertices	Inactive Vertices	Edges	SelfLoops
1	11701	11701	0	20591	764
2	14475	5193	9282	8544	147
3	16342	4579	11763	7138	8
4	18579	4525	14054	6793	10
5	23318	7432	15886	10306	13
6	25968	5420	20548	9760	1902
7	27972	4868	23104	7212	4
8	30265	5194	25071	8084	16
9	31830	4646	27184	7448	5
10	32635	3643	28992	5503	4
11	34073	4679	29394	6977	8
12	36165	5764	30401	8850	6
13	39177	7673	31504	10942	20
14	44287	10262	34025	14411	13
15	49736	11104	38632	15400	42
16	53514	9966	43548	13821	45
17	61957	16804	45153	22930	38
18	67358	12926	54432	18139	30
19	73307	14076	59231	20465	1273
20	80761	15728	65033	20736	23
21	91213	20802	70411	28867	32
22	100960	20638	80322	27429	47
23	108167	19569	88598	26660	52
24	114238	18167	96071	26331	49
25	119111	17178	101933	25242	32
26	123485	16621	106864	24705	28
27	126787	15097	111690	22171	19
28	130739	17126	113613	24869	14
29	136029	19556	116473	28721	37
30	139494	16912	122582	25329	16

Appendix F Threads Test Suite

The *Half vs All Threads* test suite, found in the *threads* package, includes eight tests that vary the transaction arrival rate (low or high) and the number of shards (8 or 16). All tests ran on the 32-thread machine. Half used 16 goroutines and the other half used 32. Each test compares the CLPA baseline with the parallel version proposed in this project. The suite was repeated 50 times and took just under 36 hours to complete.

Table F.1 shows that the (single threaded) baseline CLPA achieved an average fitness of 38324.1. The parallel version with the full number of threads improved this to 34907.9, which is about 8.9% better. Launching as many goroutines as half the number of threads gave a fitness of 35308.8, or about 7.9% better than baseline. The full-thread version performs slightly better, but the difference between full and half threads is only around 1.1%.

Table F.1 Average fitness values for different thread configurations.

Thread Configuration	Average Fitness
Single thread (Baseline CLPA)	38324.1
goroutines = Full number of threads (Parallel CLPA)	34907.9
goroutines = Half number of threads (Parallel CLPA)	35308.8

However, the slight performance improvement has a price in runtime. Table F.2 presents the average runtime performance of the baseline and parallel version implementations under the two different thread configuration. The table includes the mean execution time for each implementation as well as the ratio of the parallel version time to the baseline time, which provides a clear measure of relative performance.

From the reported ratios, namely 1.82 for the full thread configuration and 1.10 for the half thread configuration, it is clear that using a number of goroutines equal to the full number of threads results in much slower performance.

Table F.2 Average runtimes and ratios for full and half thread configurations.

	Average Epoch Baseline Time (single thread)	Average Epoch Parallel Version Time	Parallel/Baseline Ratio
goroutines = Full threads	6.27	11.43	1.82
goroutines = Half threads	6.23	6.86	1.10

It was concluded that the marginal reduction in fitness came at the cost of significantly increased execution time. The trade-off was deemed too costly for the limited benefit.