BINANCE RESEARCH

Technical Deep Dive: Parallel Execution

December 2023

Derek Ho

Table of Contents

1 Key Takeaways

- Parallel computing can increase throughput of blockchains by utilizing computing resources more efficiently.
- Traditional blockchains operate in series, but a number of newer projects support parallel computing. While there are several ways to do this, this report focuses on techniques to parallelize the execution step.
- Software design for parallel computing has many pitfalls that can lead to critical bugs. Several mechanisms exist to address these challenges, with pros and cons. We discuss three mechanisms and example projects[1] that use them:
  - Actor model (e.g. Vara): solves critical issues, but has computation overhead, may miss opportunities to parallelize, and can be difficult for developers to reason about or debug if the program is complex.
  - Memory locks (e.g. Solana, Sui and Sei): gives developers finer-grained control of the program, potentially increasing throughput. However, it is vulnerable to critical bugs that can be hard to identify, such as deadlocks.
  - Software transactional memory ("STM") (e.g. Aptos and Monad): often achieves high parallelism without risk of the critical bugs seen in the memory lock model. Simpler for decentralized application ("dApp") developers, as added complexity is dealt with by the base layer. This makes implementation potentially complex for base layer devs. Also, throughput can be worse than serial computation in some cases.
- Projects can also use single instruction multiple data ("SIMD") processing, which improves speed of certain calculations, e.g. cryptographic verifications.

[1] Understanding these different technologies gives us insight into different projects, but we need to consider them with other factors to form conclusions about the projects. These example projects were chosen simply as examples to illustrate the points made in this report.

2 Overview

2.1 Introduction

The recent surge in Solana's price has piqued interest in its distinctive design choices and technologies, especially in comparison to the frequently discussed Ethereum. There are certain areas where Ethereum has historically stumbled, but Solana has excelled. An example is Sealevel, a feature that allows the Solana Virtual Machine ("SVM") to compute programs (normally referred to as "smart contract"s on other blockchains) in parallel. This enables Solana to achieve significantly higher throughput than a conventional Ethereum Virtual Machine ("EVM").

Another noteworthy aspect is Solana's extensive use of the Rust language, not only at its core layer but also for program development. This approach assists in mitigating specific vulnerabilities and enhancing system performance.

In this report, we'll concentrate on parallel execution. We will delve into its technical aspects and obstacles that software developers must overcome. We will explore various strategies for dealing with these challenges, their advantages and disadvantages, and examples of different blockchains that utilize them for parallel processing.

This report marks the inaugural issue of our Technical Series — a deeper dive into the nuts and bolts of various blockchain technologies than our usual coverage. Our goal is to help you unravel the intricacies of certain technologies and techniques used in the crypto space.

You may run across an equation or two, along with snippets of code. For those with a tech-savvy streak, this deeper dive may prove particularly engaging. However, we made it a point to write this report in an accessible way to anyone in the crypto space, regardless of their background. We hope you find this report useful and insightful.

2.2 Why parallel?

Executing in parallel can improve throughput by utilizing hardware more efficiently.

Parallel execution employs simultaneous computation of multiple tasks on two or more processing units. The common perception is that parallel execution enhances throughput. But, why should that be the case?

First of all, parallel computation isn't always faster — it can sometimes be slower. It is *normally* faster than serial computation because:

1. Hardware with multiple slower cores is easier to build and require less energy to operate than a single fast core
2. Processes in a program can often run in parallel

Firstly, creating and operating a fast single core processor is much more difficult and expensive compared to multiple slower cores. This is why processor clock speeds have remained mostly stagnant for 2 decades, while we see a proliferation in the quantity of cores. A fast single core will consume more energy because it requires higher voltage, and face other issues such as limitations with cache and memory.

Secondly, programs don't automatically benefit from multiple cores, leading to our next point. Most programs can be parallelized to a certain degree. **The greater the degree of parallelization** (as opposed to being serial), **the more a program can theoretically leverage multi-core hardware**. However, writing **parallel software can be challenging and carries an increased risk of bugs**. Therefore, the advantage of higher throughput by switching to multi-core systems should be analyzed in light of these aspects.

2.3 Which operations can we parallelize?

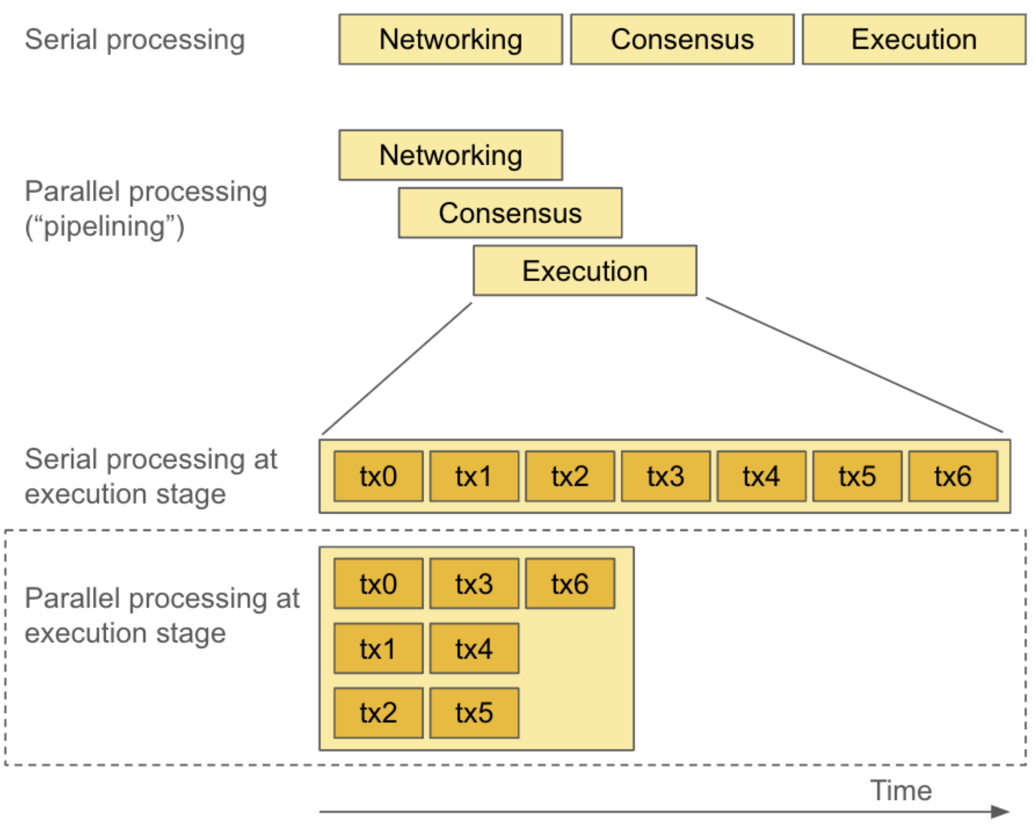
There are many opportunities to parallelize, including overlapping the consensus and execution stages. In this report, we focus on methods to parallelize the execution stage

One way to parallelize is to overlap ("pipeline") different validator operation stages. For example, proof-of-stake ("POS") validators need to share ("gossip") transaction information, elect a block producer, execute the proposed block, vote on the block, and carry out several other operations. Conventionally, POS chains perform these tasks in series. Newer blockchains are exploring methods to process some of these steps in parallel.

Protocols can also use methods such as directed acyclic graphs ("DAG"s) for its consensus, which can be applied to proof of work or proof of stake systems. Sui, for example, utilizes DAG for its consensus.

Parallelization can also be applied to the execution of transactions themselves. Nodes need to carry out a sequence of transactions to ascertain state transitions. Parallelizing this process could boost efficiency and execution speed. Our discussion here mainly focuses on this aspect—parallelizing transaction execution.

Figure 1: Parallel computing comes in many forms. This report focuses on parallel computing during the execution stage



Source: Binance Research

3 Under the hood

3.1 Different ways to parallelize execution

Sharding *reduces workload on a given client,* task parallelism *utilizes multiple cores for different tasks, and* data parallelism *improves performance of certain types of operations, such as cryptographic validation.*

Let's look at a non-exhaustive list of approaches different blockchains have used for parallel execution.

- **Sharding:** split execution work between validators
- **Task parallelism:** process non-dependent transactions in parallel using multiple cores
- **Data parallelism:** process similar instructions on different data simultaneously (SIMD)

(Execution) Sharding

The blockchain is split ("sharded") into multiple parts, and transactions related to each shard is processed by the assigned nodes separately. Sharding may be considered a parallel processing technique from a network perspective. From an individual node's perspective, they may still be processing transactions in series. However, a given node processes a fraction of all transactions submitted to the network, resulting in lower workload and increased throughput.

TON is an example of a chain that implements this. Ethereum planned to do this at one point, before opting for the roll-up centric design. Note that Ethereum's current plan, Danksharding, is a slightly different concept, as it shards transaction *data*, without splitting execution workload.

We won't deep dive into this here.

Task parallelism

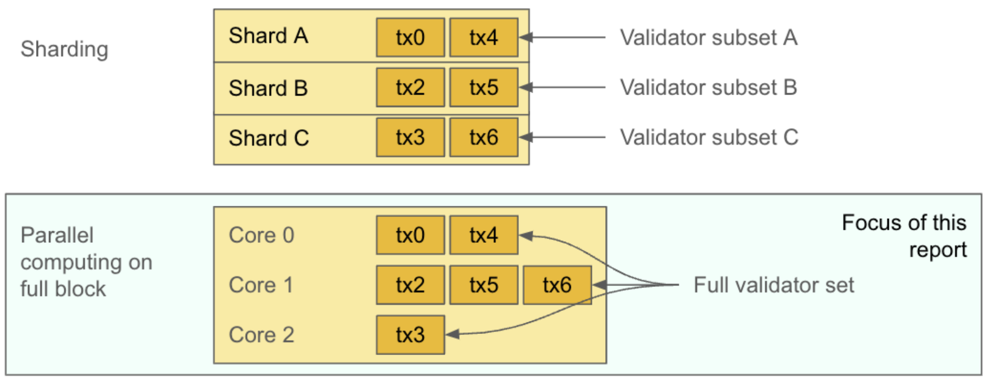
An execution client executes multiple transactions simultaneously, using multiple processing units. For example, if a node's hardware has 16 cores, it may be able to compute 16 transactions simultaneously, in batches.

Solana falls under this category. Solana has no shards, so an **execution client needs to execute all transactions in a block**. Execution is quicker because nodes can **utilize its hardware more efficiently** by taking advantage of multiple cores.

Data parallelism (Single instruction multiple data, or SIMD)

Task parallelism computes multiple instructions on multiple data ("MIMD"). On the other hand, data parallelism (the way we define it in this report) performs a single instruction on multiple data ("SIMD")[2]. SIMD operates at a low level, mostly for manipulating vectors and matrices, which are arrays of numbers. This is useful for many types of operations, notably cryptographic hashing and verification.

Figure 2: We deep dive into parallel computing a full block (task and data parallelism), rather than sharding



Source: Binance Research

3.2 Task parallelism

How parallelizable is a program?

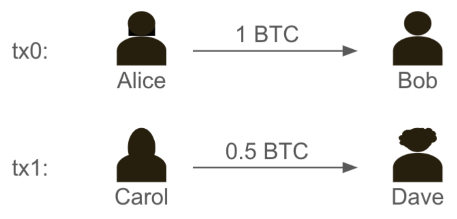
Tasks which are independent can be computed in parallel, and result in the same outcome had they been computed in series. The program execution speed is limited by the longest chain of dependent tasks.

Parallelism can increase throughput, but it comes with challenges. One of the first challenges is to determine if two tasks can be computed in parallel. Let's introduce a concept called ***independence***: Task A and Task B can be executed in parallel if they are independent.

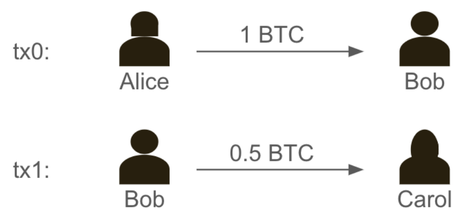
[2] Not to be confused with Solana IMprovement Docs (also called SIMD)

Cases when tasks are dependent

Case 1:



Case 2:

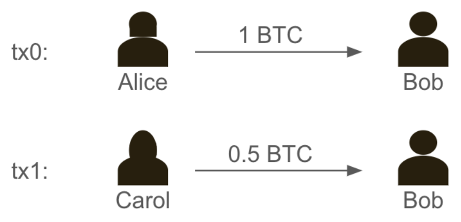


Most people would (correctly) guess that transactions ("txs") in Case 1 are independent, but not in Case 2. Imagine if there are two CPU cores, each core handling one transaction. In Case 1, the second transaction can happen regardless of the outcome of the first, and vice versa. One transaction can fail or we can change the sequence of transactions, and the final outcome will still be the same.

This is not true in Case 2. If Bob has no BTC initially, the outcome of tx1 depends on what happens in tx0.

It may be tempting to conclude that two transactions are parallelizable if the sequence does not matter. In other words, if we can flip tx0 and tx1 and get the same outcome, are the transactions independent? The answer is no. **Even when their order doesn't matter, two transactions can still be dependent**. Consider this case:

Case 3:



In Case 3, the sequence of transactions doesn't affect the outcome. tx1 can happen before tx0. However, these two transactions are still dependent. Imagine these two processes ("threads") running concurrently:

Figure 2: An example of a *race condition*, a type of software bug

		BTC balance		
Thread 0	**Thread 1**	**Alice**	**Carol**	**Bob**
Initial state		2	2	0
Read Alice's balance (2BTC)				
	Read Carol's balance (2BTC)			
Read Bob's balance (0BTC)				
	Read Bob's balance (0BTC)			
Update balances (Alice: 1BTC, Bob: 1BTC)				
	Update balances (Carol: 1.5BTC, Bob: 0.5BTC)			
Write new balance to memory		1	2	1
	Write new balance to memory	1	1.5	**0.5**

Source: Binance Research

Bob was supposed to have 1.5 BTC, but instead, he has only 0.5 BTC. If Thread 0 had completed later, Bob would have had 1 BTC, which is still not right. As we can see, the result is unpredictable because different components can interact in various ways, depending on when they happen. This turns out to be a significant issue.

Conditions for (in-)dependence

Two tasks are ***dependent*** if they fulfill one of these conditions:

- **The output of one task writes to the input of the other** — e.g. Alice transfers ETH to Bob (output), and Bob (input) transfers ETH to Carol.
- **The output of multiple tasks write to the same memory location -** e.g. both Alice and Carol transfer ETH to Bob (output).

For the math-inclined, a more formal statement is Bernstein's conditions. For two tasks i and j, where M(i) are memory locations i modifies, and R(i) are memory locations that i reads, i and j are *independent* if:

$$M(i) \cap M(j) = M(i) \cap R(j) = R(i) \cap M(j) = \emptyset$$

These math symbols effectively say the same thing as our two bullet points above.

Back to our question: how parallelizable is a program? It is determined by how dependent our tasks are. **The program is held back by the longest chain of dependent tasks**, which needs to be executed in series.

Blockchain transactions often don't depend heavily on each other, so they can run in parallel to a large extent. This provides a lot of opportunities to speed up the execution by working on multiple transactions simultaneously.

Notably, many cryptographic schemes require fully sequential computation. It's what makes them secure. For example, the computation below hashes a value 1000 times, and has to be run sequentially.

$$c_i = SHA256(c_{i-1}),\ for\ i = 0, 1, \dots , 1000$$

These types of computations are common in blockchain. For example, the *Fiat-Shamir*, which we might use as part of zero-knowledge ("zk") proof generation, requires iterative hashing. Although such cryptographic computations need to be performed sequentially, if several of such computations happen in separate independent transactions, we can still parallelize them

Additionally, on-chain cryptographic verifications (e.g. signature verification) are very common, and these often benefit from parallel computing.

Issues with task parallelism and methods to deal with them

Recall that task parallelism refers to multiple instructions on multiple data (MIMD) processes, the ability to execute multiple different tasks simultaneously. While this can increase throughput, it also opens a can of worms due to concurrency issues.

> Loosely speaking, parallel computing refers to having more than one processor core running at the same time. Concurrent computing, a closely related term, refers to a system where multiple parts can run in an out-of-order fashion, without affecting the outcome. In our case, concurrency enables parallel computing. If it doesn't immediately make sense, don't worry. Many use the two terms interchangeably, and it will work for our purposes too.

Concurrent programs are prone to software bugs. A non-exhaustive list includes:

- race conditions, as we saw earlier
- reentrancy issues
- deadlocks, where two or more tasks wait for one another indefinitely, and a related condition called livelocks
- non-composability, where a software component may be correct on its own, but have critical bugs if used in a larger program

- × priority inversion, a condition where low priority task may indirectly block a higher priority task due to intricacies on how a scheduler works
- × resource starvation, where some tasks consume more than its fair share of resources, starving other tasks of the resources they need

The first four are critical bugs. Priority inversion and resource starvation may be critical depending on the situation.

Additionally, concurrent programs are **harder for developers to manage**. This is because their design is more complex, and it's more difficult to figure out and fix issues.

However, if we can address these challenges, the advantages of parallel computing can exceed the difficulties. Let's explore some methods to manage this (known as "concurrency control") and take a look at some example projects that use them:

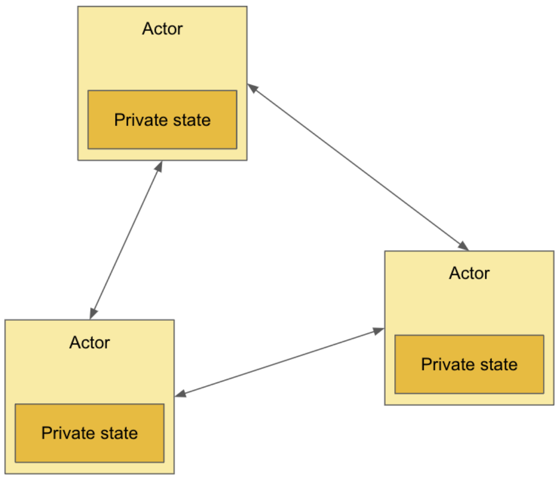
- **Message passing model**:
  - **Actor model** (used by Vara)
- **Shared memory model**, which can be further categorized into:
  - **Memory locks** such as mutual exclusion (mutex) (used by Solana, Sui and Sei)
  - Optimistic concurrency control or **software transactional memory** (OCC/STM) (used by Aptos and Monad)

Actor model (e.g. Vara)

> *Solves many critical problems with concurrency. However, the global state cannot be directly observed, which makes it harder for the dApp developer to reason about and debug. There may also be missed opportunities to parallelize*

The actor model assigns different *actors* their ***private state***. An actor can directly modify its private state, but only indirectly modify another actor's state by passing a **message**. When an actor receives a message, it decides how to handle it, potentially modifying its own state in response to the message.

Figure 3: Actors have private states that they have exclusive access to. Actors can also send or receive messages from other actors.



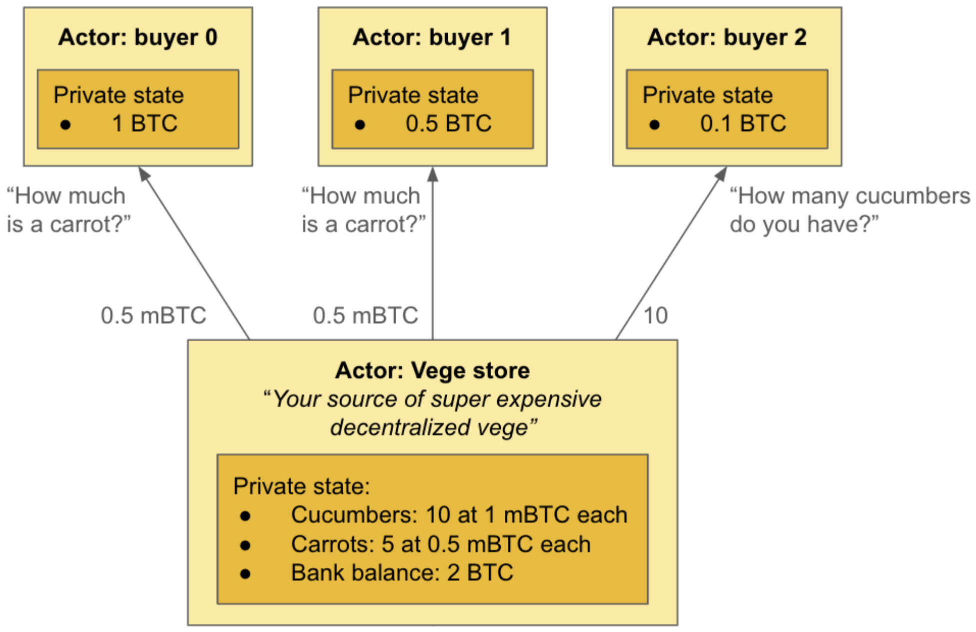
Source: Binance Research

How does this enable parallelism? Each actor can process their own operations sequentially, while the program (with many actors) assigns a processing thread to each actor. Since actors can only access their own data, which is unique, there cannot be situations where multiple actors are reading or writing to the same memory location. Therefore, assigning a thread to each actor enables parallel computing, without risk of data dependency issues.

CosmWasm (which is a smart contract framework, not a blockchain) also uses the actor model. It uses it to avoid reentrancy vulnerabilities on smart contracts, one of the biggest smart contract vulnerabilities on Ethereum Virtual Machines (EVMs). It can potentially be used to enable parallel computing as well, but CosmWasm does not support this yet, as of writing.

While the actor model solves many issues with concurrency, it is not perfect. A single actor processes its tasks sequentially, which can miss opportunities to parallelize. Let's see an example, using a hypothetical decentralized vegetable store serving several customers:

Figure 4: Three actors (the buyers) querying an actor's (vege store's) private state by exchanging messages. In a basic actor model, the vege store above needs to process each message one by one, although each operation is independent



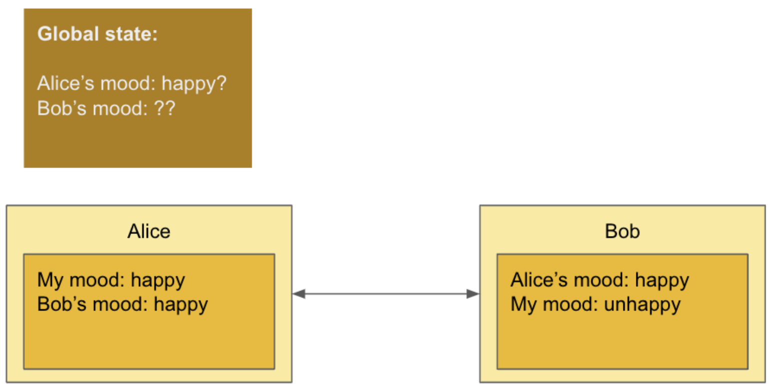
Source: Binance Research

Despite the vegetable store actor processing messages sequentially, **our independence conditions suggest they could be handled concurrently.** Multiple processes can read the same datapoint in parallel. Is there a more effective model that allows the store to disclose its private state like a public price list?[3]

Also, there's no single location to observe the system's global state. A programmer might examine each private state to understand the bigger picture, but it can sometimes be complex. Interwoven private states can generate a **global state that's not instantly visible**, making system behavior prediction challenging.

[3] This can actually be done with some tweaks. For example, we can improve our basic actor model by defining some values as immutable (such as the price) and some as mutable (such as the number of items and balances). Immutable values can be safely read directly by any actor because there is no possibility of a process modifying it. However, implementing this increases complexity.

Figure 5: It's a bad idea to have duplicate states if they are meant to be global, but this can happen in a poorly implemented program using the actor model. Alice thinks Bob is happy, but Bob is actually unhappy. The outcome of an interaction is unpredictable. If we try to piece together the global state, we'll realize it is either inconsistent or more complicated than we initially thought.



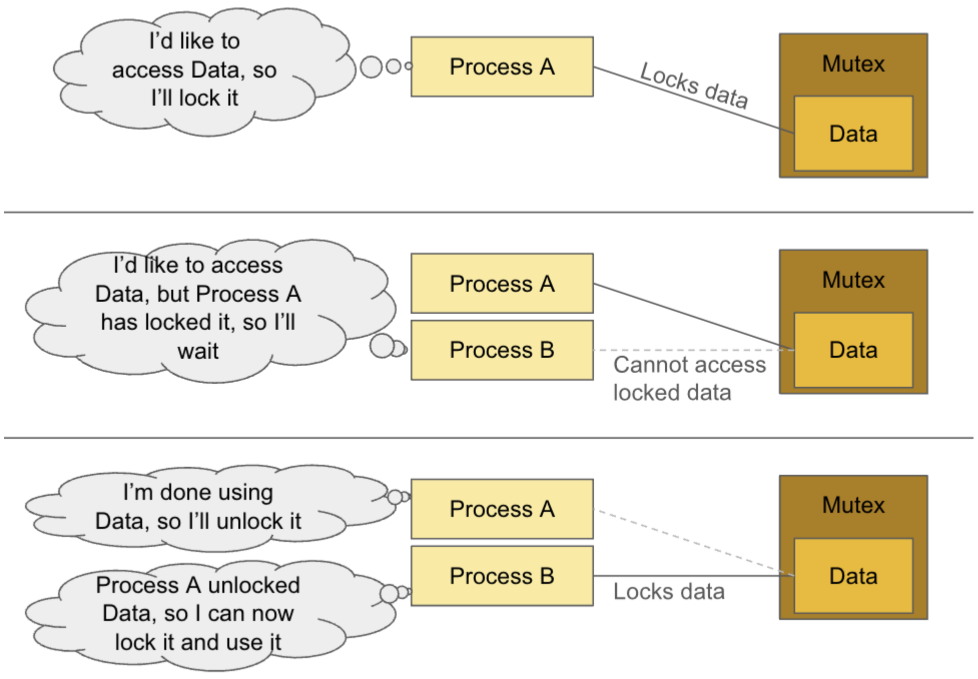
Source: Binance Research

Memory locks (e.g. Solana, Sui and Sei)

> *Achieves high concurrency if implemented correctly. May be easier for dApp developers to reason about. Has many pitfalls that lead to critical software bugs, but many (not all) are mitigated by using a language with strong thread-safety (eg: Rust).*

In the memory lock model, processes have access to all memory locations (i.e. data) by default. If a process wishes to access some data, it 'locks' that memory location before starting its process. Once the process is complete, it unlocks that memory location to release it, so other processes can use it again.

Figure 6: In a mutex lock, every process wishing to access memory will first lock it. Another process wishing to access the data will wait until it is unlocked, before locking it again and using the data.



I'd like to access Data, so I'll lock it
Process A
Locks data
Mutex
Data
I'd like to access Data, but Process A has locked it, so I'll wait
Process A
Process B
Cannot access locked data
Mutex
Data
I'm done using Data, so I'll unlock it
Process A unlocked Data, so I can now lock it and use it
Process A
Process B
Locks data
Mutex
Data

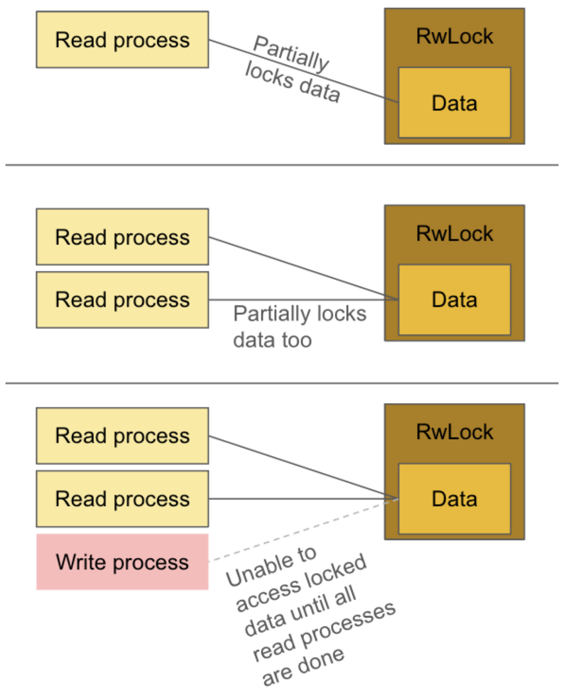
Source: Binance Research

But we can do better. Recall the conditions for independence. Another way of stating it is:

Multiple tasks can access the same memory location only if none of them modifies its contents

We can have a mechanism that checks every process wishing to access memory for whether it intends to read or modify the data. The mechanism disallows a process from accessing the data if the rules of independence are violated. It effectively has two types of keys, read-only keys and a write (modify) key.

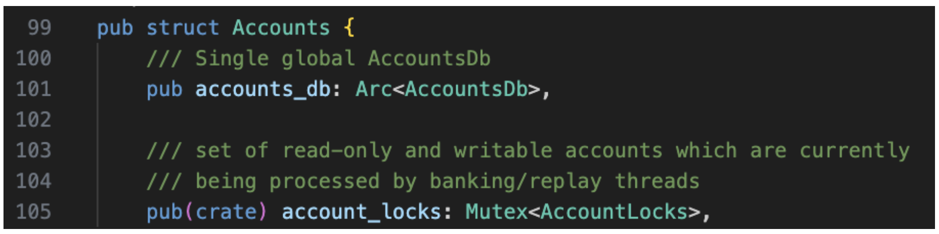
Figure 7: We can have a locking mechanism which allows either a) multiple read processes or b) one write process to access the data



Source: Binance Research

Let's have a look at Solana's code. In Solana, memory locations are called *accounts*. An account can contain data, and transactions may need to lock one or more accounts in order to access the data inside. Below is one part of the code that keeps track of the accounts that have been locked.

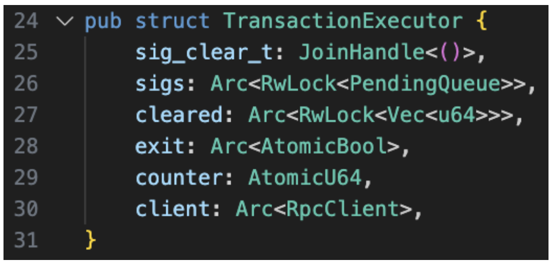
Figure 8: Solana data structure that contains information on account locks. This data structure itself uses Mutex (line 105) which ensures that only 1 thread at a time can access the list of account locks.



```
 99   pub struct Accounts {
100       /// Single global AccountsDb
101       pub accounts_db: Arc<AccountsDb>,
102
103       /// set of read-only and writable accounts which are currently
104       /// being processed by banking/replay threads
105       pub(crate) account_locks: Mutex<AccountLocks>,
```

Source: Solana repository commit c3323c0. Solana > accounts-db > src > accounts.rs

Figure 9: Usage of RwLock<> in Solana's code, which is similar to Mutex but allows multiple read operations to happen concurrently

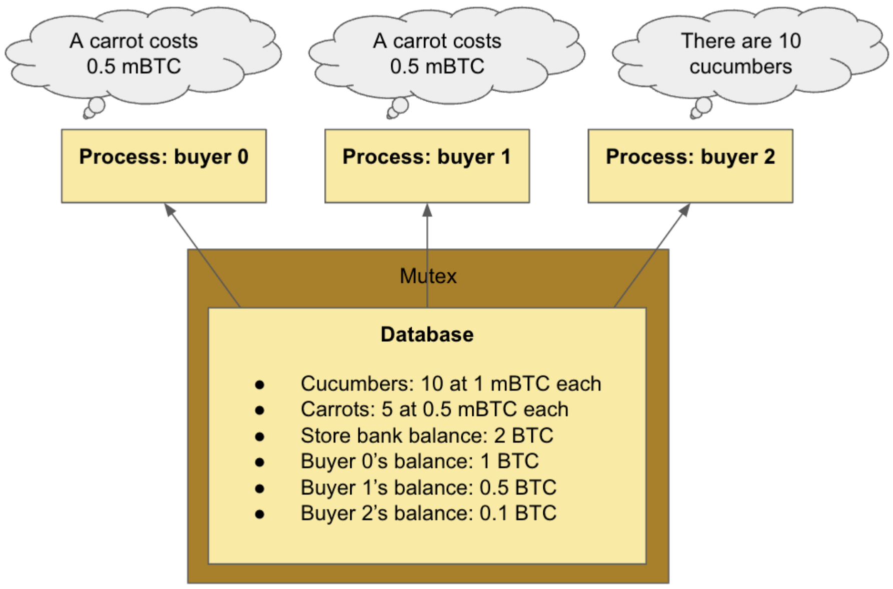


```
pub struct TransactionExecutor {
    sig_clear_t: JoinHandle<()>,
    sigs: Arc<RwLock<PendingQueue>>,
    cleared: Arc<RwLock<Vec<u64>>>,
    exit: Arc<AtomicBool>,
    counter: AtomicU64,
    client: Arc<RpcClient>,
}
```

Source: Solana repository commit c3323c0. Solana > client > src > transaction_executor.rs

Back to our vege store illustration. With memory locks, the three buyer's queries can be performed in parallel, because they do not modify the data.

Figure 10: Multiple read operations are allowed, so these three buyers can read the database concurrently. It is as if the store owner has a board displaying prices and current inventory levels. Multiple people can read the board at the same time.



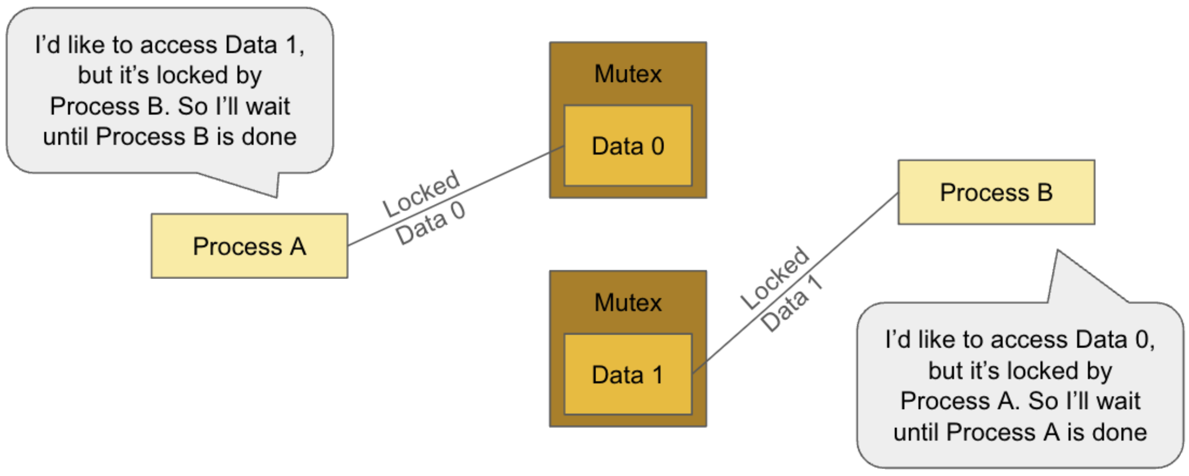
Source: Binance Research

While it sounds conceptually simple, memory locks are complicated to work with and error-prone. In fact, some strongly discourage using this model. One of Go's (or Golang, the programming language) slogan is "*Don't communicate by sharing memory; share memory by communicating.*"

Using memory locks is doing precisely what Go says you shouldn't do (note that memory locks are available in Go despite this slogan). The reason is **there are many pitfalls when writing code with memory locks**. It's easy to mis-coordinate the locking process. A process could lock a data point, complete its process, and forget to unlock it. Or the process is terminated before it can unlock a data point. Tasks that need to access that memory location are now locked out indefinitely.

Deadlocks are another pitfall, and can be **difficult to identify**. A deadlock happens when two processes need at least two locks, but have each acquired one. They will wait on each other indefinitely, because neither can acquire the other lock it needs.

Figure 11: A deadlock. Process A and B wait for each other forever.


I'd like to access Data 1, but it's locked by Process B. So I'll wait until Process B is done
Mutex
Data 0
Locked Data 0
Process A
Process B
Mutex
Data 1
Locked Data 1
I'd like to access Data 0, but it's locked by Process A. So I'll wait until Process A is done

Source: Binance Research

Data can also be corrupted, called *mutex poisoning* in the Rust community, when a thread holding a lock aborts ('panics') due to an error. Unlike our example earlier of a lock never being returned, Rust has safety mechanisms to ensure a lock is returned even if a process terminates. However, the contents of the data may be corrupted, as the thread may have partially modified the data before crashing.

Programs designed with memory locks are also normally **not composable**—a software module could work properly on its own, but fail when used as part of a larger program. A programmer may have written code in a way that ensures no deadlocks happen within the module, but when interacting with other modules that access the same data, a deadlock may be created. Also, different modules may use different mutex logic—one using a guard and one using a more manual approach—which can lead to mis-coordination in handling the memory locks. These situations are likely to result in critical bugs.

With all these issues, why is anyone using memory locks at all? One important factor could be because of **Rust** (the programming language), which **allows programmers to write code with memory locks in a safer way than was possible with previous languages** such as Java and C#. Incidentally, both Solana and Sui are written in Rust[4]. Sei is a Cosmos

[4] Deadlocks are still a potential issue when programming with Rust.

chain, written in Go, but its parallelization module is written in Rust. With the safety afforded by Rust, these projects can benefit from the advantages of using memory locks.

Figure 12: Rust's mascot (left) and Go's mascot (right) arguing. If you don't get the joke: Rust says Go is only good at the actor model. Go responds by poking fun at Rust's notoriously strict compiler. Rust's compiler enables memory locks to be used relatively safely, but it also causes many new Rust programmers to spend more time "fighting" with compiler errors than actually writing code. Rust hits back by saying that because of the compiler, Rust doesn't have data races (a type of race condition, a critical bug we demonstrated earlier), unlike Go.

All you do is act
At least I don't fight with my compiler all day and move slowly
I don't race

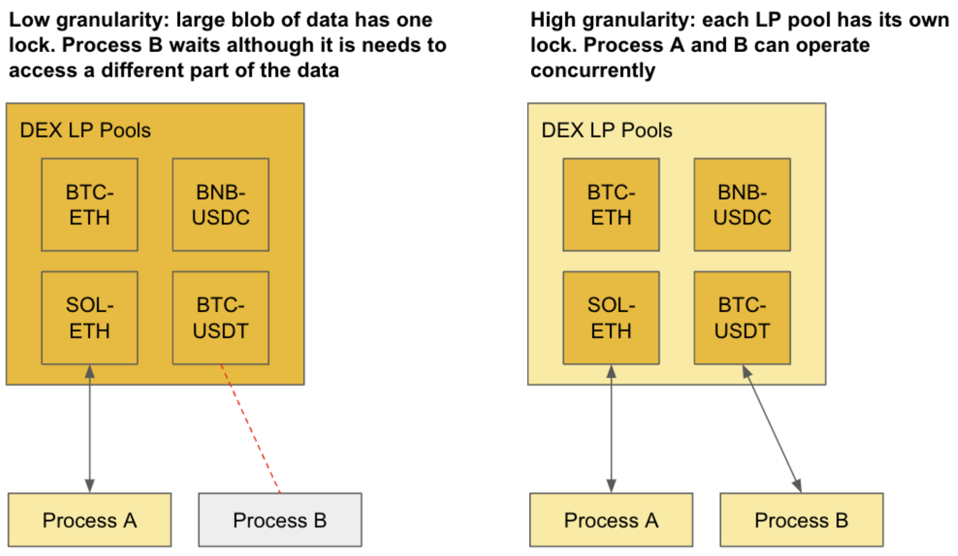
Source: Binance Research

Memory locks normally offer **finer-grained control** than the actor model, making it easier for developers to optimize an application. A program using the memory lock model may also be easier to reason about than one using the actor model, especially in simple cases. In complex programs, this point is debatable.

When designing a program with memory locks, there is a **tradeoff in the level of granularity**. Highly granular locks enable greater parallelism, because it avoids locking parts of data that don't need to be locked. But high granularity also results in more overhead due to the greater complexity of managing locks, and a higher risk of creating deadlocks, as it is more complex for programmers to reason about.

Solana programs (aka smart contracts) are encouraged to split their state across several accounts, which increases granularity. For example, a decentralized exchange ("DEX") can store information on all its liquidity pools ("LPs") in a single account. However, this may be non-optimal, as transactions wishing to access this information will lock the entire blob of data, preventing others from accessing it. On the other end of the spectrum, the program could have a separate account for each LP pool, which increases parallelism but at the cost of greater overhead and risks.

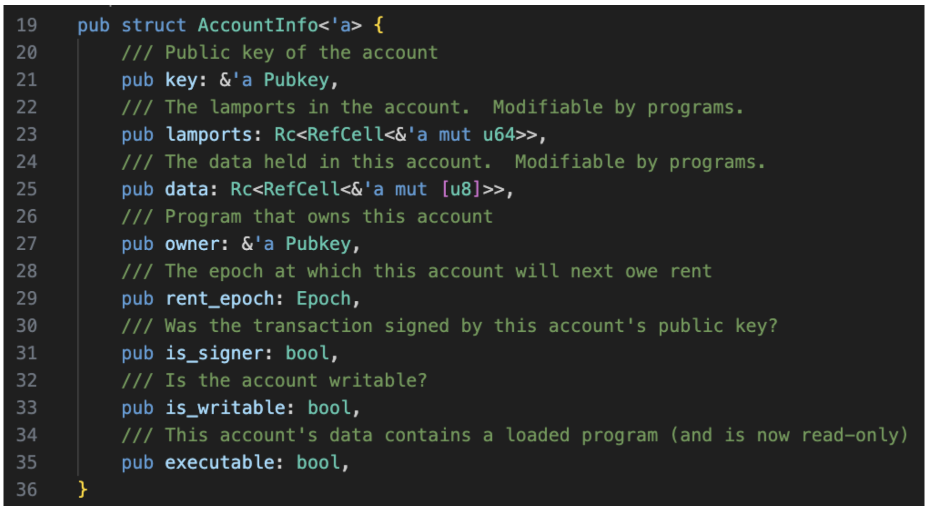
Figure 13: Higher granularity improves parallelism because of lower contention (conflicts)...but increases computation overhead and risk of creating deadlocks


Low granularity: large blob of data has one lock. Process B waits although it is needs to access a different part of the data
DEX LP Pools
BTC-ETH
BNB-USDC
SOL-ETH
BTC-USDT
Process A
Process B
High granularity: each LP pool has its own lock. Process A and B can operate concurrently
DEX LP Pools
BTC-ETH
BNB-USDC
SOL-ETH
BTC-USDT
Process A
Process B

Source: Binance Research

Similar to the actor model, handling memory locks requires **extra work** from the software programmer. For example on Solana, transactions need to **explicitly declare the accounts (i.e. memory locations) they will access**. The Solana VM then uses this information to lock data associated with those accounts.

Figure 14: Transactions on Solana need to declare the memory locations ("accounts") that it will read or modify. This code defines the format of this information. The runtime uses this information to decide which transactions to run in parallel.



```rust
pub struct AccountInfo<'a> {
    /// Public key of the account
    pub key: &'a Pubkey,
    /// The lamports in the account.  Modifiable by programs.
    pub lamports: Rc<RefCell<&'a mut u64>>,
    /// The data held in this account.  Modifiable by programs.
    pub data: Rc<RefCell<&'a mut [u8]>>,
    /// Program that owns this account
    pub owner: &'a Pubkey,
    /// The epoch at which this account will next owe rent
    pub rent_epoch: Epoch,
    /// Was the transaction signed by this account's public key?
    pub is_signer: bool,
    /// Is the account writable?
    pub is_writable: bool,
    /// This account's data contains a loaded program (and is now read-only)
    pub executable: bool,
}
```

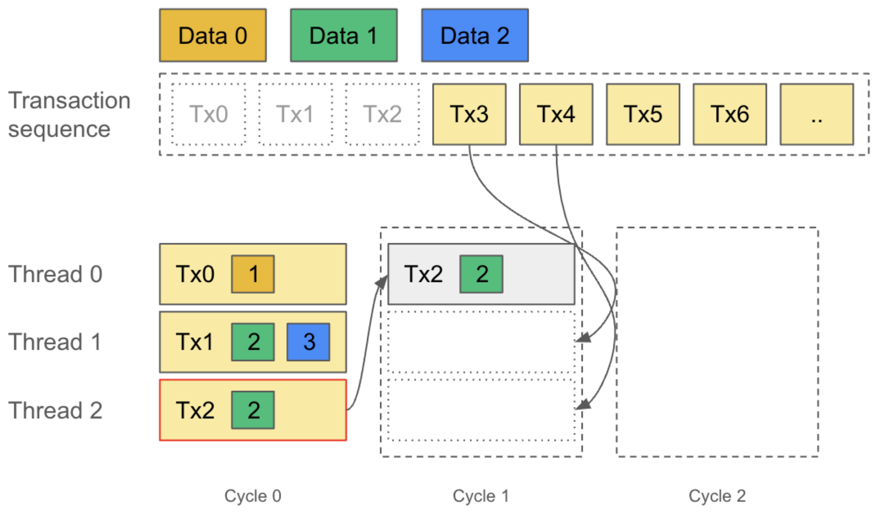
Source: Solana repository commit c3323c0. Solana > sdk > program > source > account_info.rs

Software transactional memory (e.g. Aptos and Monad)

STM normally achieves high parallelism without the risks of deadlocks. dApp developers do not need to deal with increased complexity, but base layer implementation can be difficult. Throughput can be low if there are many dependencies, as transactions may retry many times before succeeding.

Software transactional memory ("STM") takes an **optimistic approach** to shared memory. It runs many threads in parallel, disregarding whether they are independent or not. Once done, it checks for potential conflicts. The highest priority task in any conflict is committed, while the others are aborted and rerun ("retry") in the next cycle. This happens until all processes are executed.

Figure 15: Simplified illustration of Aptos' STM design: Tx0, Tx1 and Tx2 are taken from the queue and run optimistically in parallel. In Cycle 0, Tx0 modifies Data 0, Tx1 modifies Data 2 and 3, and Tx2 modifies Data 2. After the cycle, Tx1 and Tx2 notice that they had both modified Data 2 (a conflict). Because Tx1 is earlier in the sequence of transactions, Tx1 commits and Tx2 aborts. Tx2 retries in the next cycle.



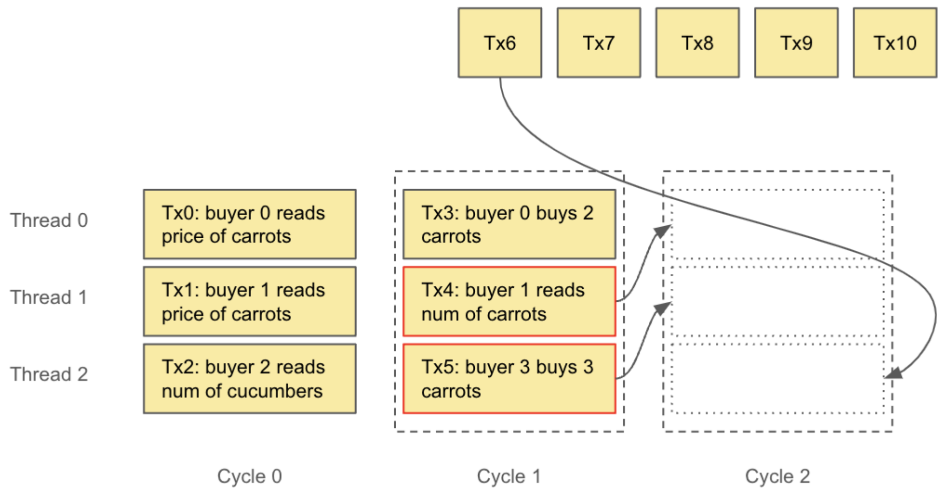
Source: Binance Research

There are many different designs of STM. Aptos' STM design uses a multi-versioned data structure to detect and handle conflicts. When transactions modify any value, the system logs the change and attaches a new version number, while keeping track of the historical values. All STM implementations require some form of log like this. **The log takes place in memory before being committed, which can lead to high memory requirements**.

Let's return to our vege store illustration. If we were to use STM, it would look something like this:

Figure 16: using STM on the vege store. After Cycle 0, all three threads note that there had been no conflict, so all three transactions are committed. In Cycle 1, Tx4 and Tx5

see that the information they read or modified (number of carrots) had been changed by Tx3 in the same cycle. Since Tx3 is ordered before Tx4 and Tx5 in the sequence of transactions, Tx3 is committed, and Tx4 and Tx5 abort. Tx4 and Tx5 retry in Cycle 2.


Tx6
Tx7
Tx8
Tx9
Tx10
Thread 0
Thread 1
Thread 2
Tx0: buyer 0 reads price of carrots
Tx1: buyer 1 reads price of carrots
Tx2: buyer 2 reads num of cucumbers
Tx3: buyer 0 buys 2 carrots
Tx4: buyer 1 reads num of carrots
Tx5: buyer 3 buys 3 carrots
Cycle 0
Cycle 1
Cycle 2

Source: Binance Research

At first glance, STM's concept of aborting and retrying many times may seem wasteful. Also, the **chance of conflict increases with the number of threads**. If we run 100 threads, we are more likely to have conflicts than if we run 5 threads. In a program with many dependencies, a task can abort many times before finally committing. It also has computation overhead of managing the process. This can ultimately result in lower performance than simple serial execution.

Nevertheless, **STM results in higher throughput in many cases**. It generally achieves comparable parallelism to a memory lock model with high granularity, but without the risk of deadlocks. Also, there are clever ways to design the system so that an aborted task does not need to be recomputed from scratch, or to detect potential conflicts earlier in a cycle for tasks that are being retried.

It is also easier on the dApp developer. The **dApp developer can write a smart contract as if it is a serial program**, yet benefit from parallel computation. However, the base layer (e.g. Aptos' VM) needs to handle the complexity of concurrency and parallelizing the execution, **making STM complex for the base layer developers** (e.g. the Aptos core team writing the code for the blockchain). STM is also an area of active research, so some aspects of the model may not be fully understood yet.

3.3 Data parallelism (SIMD)

SIMD improves computation speed for certain types of operations, such as cryptographic verifications.

Data parallelism, in this report, refers to operations with a single instruction on multiple data (SIMD). *Instructions* are quite basic computing units, such as addition and multiplication. It turns out that this is useful when dealing with vectors or matrices. SIMD allows us to perform vector or matrix operations very quickly.

Figure 17: In this example, we encrypt our secret data {1.5, 3.2} by multiplying it with the encryption key matrix. To decrypt, we multiply the inverse of this matrix with the encrypted data. If you remember matrix multiplications, you will see that we retrieve the original secret data. Real-world ciphers such as Advanced Encryption Standard (AES) perform similar matrix multiplications as part of a larger, much more sophisticated algorithm.

Encryption

$$\overbrace{\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}}^{\text{Encryption key}}\overbrace{\begin{pmatrix}1.5\\ 3.2\end{pmatrix}}^{\text{Data}} = \overbrace{\begin{pmatrix}7.9\\ 17.3\end{pmatrix}}^{\text{Encrypted data}}$$

Decryption

$$\overbrace{\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}^{-1}}^{\text{Inverse encryption key}}\overbrace{\begin{pmatrix}7.9\\ 17.3\end{pmatrix}}^{\text{Encrypted data}} = \begin{pmatrix}-2 & 1\\ 1.5 & -0.5\end{pmatrix}\begin{pmatrix}7.9\\ 17.3\end{pmatrix} = \overbrace{\begin{pmatrix}1.5\\ 3.2\end{pmatrix}}^{\text{Decrypted data}}$$

Source: Binance Research

This is not a math lesson, so we won't dive into this. The thing to note is that **vectors and matrices are very useful in many areas of computer science, such as processing large amounts of data and performing cryptographic operations**. The basic example above shows how matrices can be used for encryption and decryption using an encryption key.

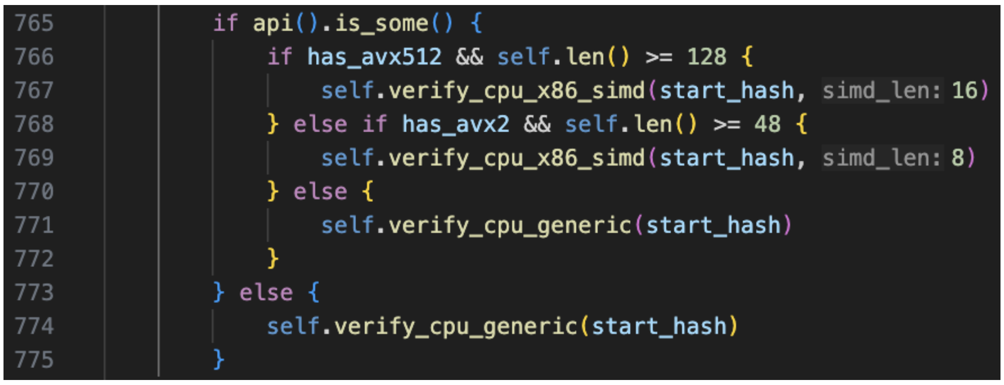
SIMD can be done using CPUs or GPUs. Many CPUs today have 64-bit processors. This means that in each clock cycle, a processor can operate on 64-bit data. Some processors, such as AVX-2 and AVX-512, expand this register to 256 or 512 bits. With a 256 bit register, the processor can simultaneously operate on four 64-bit numbers, for example.

GPUs can also perform SIMD operations, where a single task scheduler can assign similar instructions to many processing units. With several thousand cores, GPUs are performant in cases where a single instruction is used across a very large dataset.

Vector and matrix operations involve a lot of repeated additions and multiplications on arrays of numbers. SIMD can speed up these operations significantly.

Let's look at Solana's code, showing the use of the AVX processor.

Figure 18: Code snippet performing cryptographic verification during Solana's Proof of History (PoH). Depending on the hardware being used, Solana will use different algorithms for verification. For example, if it detects an AVX-512 processor, it will verify using SIMD with 16 data points per cycle (lines 766 and 767).



```
765            if api().is_some() {
766                if has_avx512 && self.len() >= 128 {
767                    self.verify_cpu_x86_simd(start_hash, simd_len: 16)
768                } else if has_avx2 && self.len() >= 48 {
769                    self.verify_cpu_x86_simd(start_hash, simd_len: 8)
770                } else {
771                    self.verify_cpu_generic(start_hash)
772                }
773            } else {
774                self.verify_cpu_generic(start_hash)
775            }
```

Source: Solana repository commit c3323c0. Solana > entry > src > entry.rs

4 Wrapping up

Let's recap what we have covered. Parallel processing can increase throughput (or reduce cost) by utilizing resources more efficiently. The execution stage is one of several areas we can parallelize, but it presents issues that software developers must overcome. Incorrectly handling these issues results in critical software bugs. We discussed a few solutions, which is summarized below, including their pros and cons:

Figure 19: Pros and cons of approaches discussed in this report

		Pros	Cons	Example projects discussed in this report
Sharding		Node operators do not need to have hardware that support parallel processing	Often trades off security or latency for throughput, rather than using hardware more efficiently	Used by TON, and was the original Ethereum 2.0 vision
Message-passing model	**Actor model**	Solves most concurrency issues	Harder to reason about, and may miss parallelism opportunities	Vara
Shared memory model	**Memory locks / Mutex**	Finer-grained control, allowing for more optimization. Potentially lower complexity burden on dApp developers	Vulnerable to some critical issues, notably deadlocks and non-composability, even when using a thread-safe program like Rust	Solana, Sui, Sei
	Software Transactional Memory (STM)	Solves critical concurrency issues. Often achieves high parallelism. No additional complexity for dApp developers.	Complexity burden on base layer developers. Active area of research, so potentially less well-understood.	Aptos, Monad
Data parallelism	**SIMD**	Significant speed-up on certain operations, mostly those that use matrices, which includes many cryptographic functions	Only works for certain operations, i.e. those that have the same instructions on many data points	Solana

Source: Binance Research

Will Ethereum incorporate parallel processing? It is **unlikely** in the near future, considering it is pursuing a rollup-centric model. Nevertheless, there is promising potential for EVMs equipped with parallel processing capabilities (e.g. Monad) and rollups that use parallel processing at the sequencer level and settle on Ethereum (e.g. Eclipse). There is also an emerging trend allowing Solidity to be used for smart contracts on blockchains equipped with parallel processing. For instance in Solana, while Rust is the main language, Solidity can also be used by utilizing the Solang compiler.

In the wider crypto-ecosystem, parallel processing is being adopted by many new projects. We've discussed a few here, highlighting Solana due to recent interest. Parallel processing is key to improving throughput and is a selling point for many projects.

References

https://docs.solana.com/validator/runtime

https://github.com/solana-labs/solana

https://arxiv.org/abs/2203.06871

https://aptos.dev/papers/whitepaper.pdf

https://docs.sui.io/research

https://docs.monad.xyz/technical-discussion/execution/parallel-execution

https://docs.sei.io/advanced/parallelism

https://vara-network.io/

https://docs.cosmwasm.com/docs/

New Binance Research Reports

Exploring Tokenomics Models and Developments

An overview of developments in Tokenomics

A Primer on On-Chain Gaming

An introduction to on-chain gaming

Are We Entering a Bull Market? Top 10 Narratives to Follow

The top 10 narratives to follow as we go through the next few months

Monthly Market Insights - December 2023

A summary of the most important market developments, interesting charts and upcoming events

About Binance Research

Binance Research is the research arm of Binance, the world's leading cryptocurrency exchange. The team is committed to delivering objective, independent, and comprehensive analysis and aims to be the thought leader in the crypto space. Our analysts publish insightful thought pieces regularly on topics related but not limited to, the crypto ecosystem, blockchain technologies, and the latest market themes.

Derek Ho

Protocol Specialist

Derek is a Protocol Specialist at Binance, working on protocol and security with various teams. He enjoys reading whitepapers, understanding mathematical formulas, and analyzing blockchain code. He holds an engineering degree from Cambridge University.

Resources

Read more here

Share your feedback here