Supra Containers: AppChains on Layer 1

Supra Research

October 2024

v2.0

Abstract

AppChains are blockchains dedicated to one or a group of related applications. Examples include DeFi Kingdoms [5] on Avalanche, Osmosis [15], and dYdX [7] on Cosmos. They provide sovereignty to the application creators and to its user community by allowing them to bring a custom token into circulation, set custom gas prices, permissioned deployment of smart contracts, etc. Polkadot Parachains, Cosmos Zones, Avalanche Subnets and Layer 2s are prominent AppChain solutions.

In the era of high throughput Layer 1 blockchains, we believe current solutions are too expensive, cumbersome and restrictive as an entire network of node operators and economic security needs to be provisioned and bootstrapped for an application. Inspired by AppChains, we offer *Supra Containers* on our high throughput, fast finality Supra Layer 1 blockchain. Supra Containers not only provide all the features of AppChains, but also solve the well known issues of fragmented liquidity owing to the lack of smart contract atomic composability across these zones and networks.

Compared to existing AppChain solutions, Supra Containers offer a better model for:

application creators by providing customization of gas tokens and gas prices, gated deployment of smart contracts, opening up auctioning markets for transaction fees, and most importantly Supra Containers enable cross-container composability and thus avoid liquidity fragmentation,

end consumers by providing a seamless access to multiple containers on a single platform and for services integrated across containers.

Thus, Supra Containers enables and facilitates application-based businesses and push the host Supra Layer 1 blockchain towards creating networking effects.

1 Introduction

Blockchains started off primarily with native currency transfers in the form of the Bitcoin network [12]. Soon Ethereum [8] introduced *smart contracts* capability and provided full Turing-complete on-chain programming, and it was heralded as Blockchain 2.0. This on-chain programming gave birth to transitioning many traditional financial instruments on to blockchains and provided the Decentralized Finance (DeFi) space with a clear technological advantage over legacy financial infrastructure. Though blockchains are being adopted in various applications like synthetic assets, forex swaps, supply chain management and validation, etc., DeFi has been the major driving force in the adoption of blockchains.

When an entire blockchain is reserved for an application or a group of applications that combination is called an **AppChain**, or an application specific chain. AppChains have emerged mainly because they grant sovereignty and provide flexibility to the dApp's creators as well as to its community, and have achieved product-market fit in DeFi, GameFi and DePIN. A community can employ their own bespoke business models, release their own tokens, create their own governance mechanisms, and set transaction fees to facilitate smooth and sustainable operations. In a sense, AppChains are becoming digital decentralized gated communities. Some examples of AppChains include Osmosis [15] and dYdX v4 [7] on Cosmos [13], DeFi Kingdoms [5] on Avalanche, and hundreds of Layer 2s on Ethereum.

Typically, the application creators float new tokens that are exclusively used for the purposes of interacting with the applications on their AppChains. Such currencies are

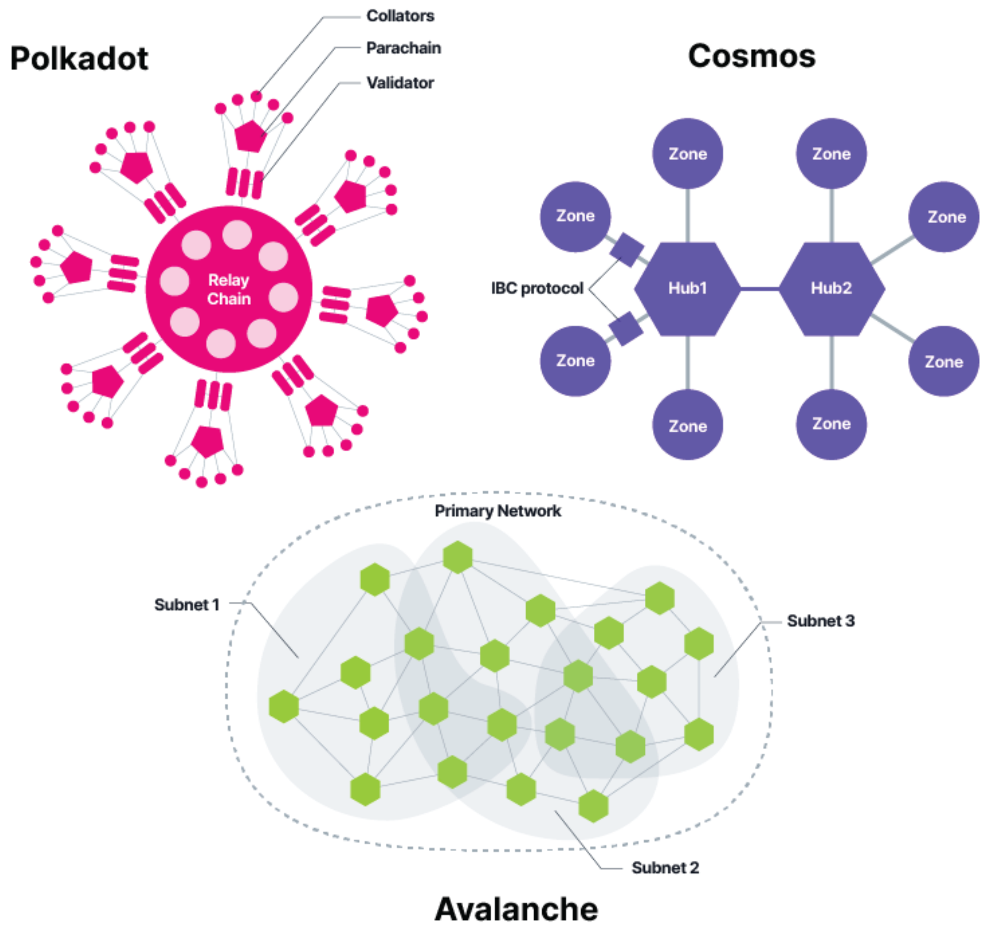

Figure 1 State-of-Art AppChain solutions

accepted for the payment of transaction fees as well as for governance. The holders of AppChain tokens become empowered similar to the holders of equity shares of a business.

It is not easy to deploy other crypto-currencies that already have high adoption on each and every AppChains. However, there are work-arounds in the form of wrapped tokens but that do come with their own associated costs. So such difficulties also make way for an AppChain token to gain in value as and when their applications attract more transaction traffic. For example, DeFi Kingdoms AppChain has its own token JEWEL with its current value of 14 US cents (as on 21st September 2024).

Localized Fee Markets. Transactions targeting such an application expect that there is no influence. It is often seen in popular public blockchains, however, that a high volume of traffic targeting some contracts increases the transaction fees for all end consumers. Users and developers debate over whether this unfairly impacts certain dApps more than others. Localized fee markets are an appealing feature here, where the gas prices of transactions are expected to change in proportion only to the traffic attracted by its application rather than state congestion and smart contract congestion of unrelated applications.

The **state-of-art** solutions to AppChains include independent Layer 2s as well as Layer-2s built using modular frameworks like Celestia [3] or Layer-2 template stacks (e.g., OP stack of Optimism [14]), sidechains, Cosmos Zones, Polkadot Parachains, Avalanche Subnets, etc. Refer Figure 1. We observe that AppChains came into use before public blockchains achieved the extreme performance we see today. In the presence of a high throughput and fast finality Layer 1 blockchain, we believe AppChains can be accommodated and offered inside Layer 1s alone, rather than looking out for solutions in the Layer 2 space or disparate, external networks. This direction led us to the design of Supra Containers.

One of the major **shortcomings** of AppChains and Layer 2s is their inability to atomically compose with the services outside of their infrastructure. Consumers who use these AppChains have to bridge their assets to the respective AppChain first before starting to use its services. This bridging is an artifact of different networks hosting different data and is not entirely desirable as it **fragments liquidity**.

We recognize the growth in adoption of AppChains and appreciate the fundamental philosophy of AppChains as an *enabler* of businesses centered around blockchain dApps, and *empowerer* of decentralized communities. Inspired by these concepts, we studied and adapted the notion of AppChains on Supra Layer 1 blockchain, a high throughput and fast finality blockchain, as **Supra Containers**. This paper introduces Supra Containers and shows that they meet all the features of AppChains, and more importantly overcomes the limitation of fragmented liquidity through governance controlled atomic composability.

The rest of the paper is organized as follows. Section 2 details on the concept and features of Supra Containers. Section 3 expands on the benefits inherited by Supra Containers by Supra blockchain network. Section 4 presents how Supra Containers may be leveraged towards optimizing execution times via parallelized execution. Finally, Section 6 summarizes and concludes this paper.

2 Supra Containers

In this section, we define Supra Containers and expand on its position that they overcome a major limitation of AppChains of fragmented liquidity.

We notice that the on-chain component of a dApp (decentralized application) is just a group of smart contracts calling methods of each other. So we start off by modeling a **Supra Container** technically as a *package or bundle of smart contracts* typically owned or created by a single individual or an institution or a community[1].

Note that this bundling of smart contracts may not necessarily result in a partition of the blockchain state. It depends on the structure of storage. For example, on Ethereum Virtual Machine (EVM), since smart contracts have their own exclusive storage spaces, bundling of smart contracts partitions the EVM state into Container-states. However, this may not necessarily hold for the case of Aptos Move because there is no direct correlation between addresses of data elements and Move modules.

The Figure 2 illustrates three custom Containers – A, B and C, and the default public Supra Container. The default Container and Container B are public, meaning the smart contract deployments are not gated and anyone may deploy a smart contract here, as shown by the border less box. The arrows represent the method calls, they can be intra-Container as well as cross-Container.

[1] Supra Containers are not to be confused with Docker [6] Containers.

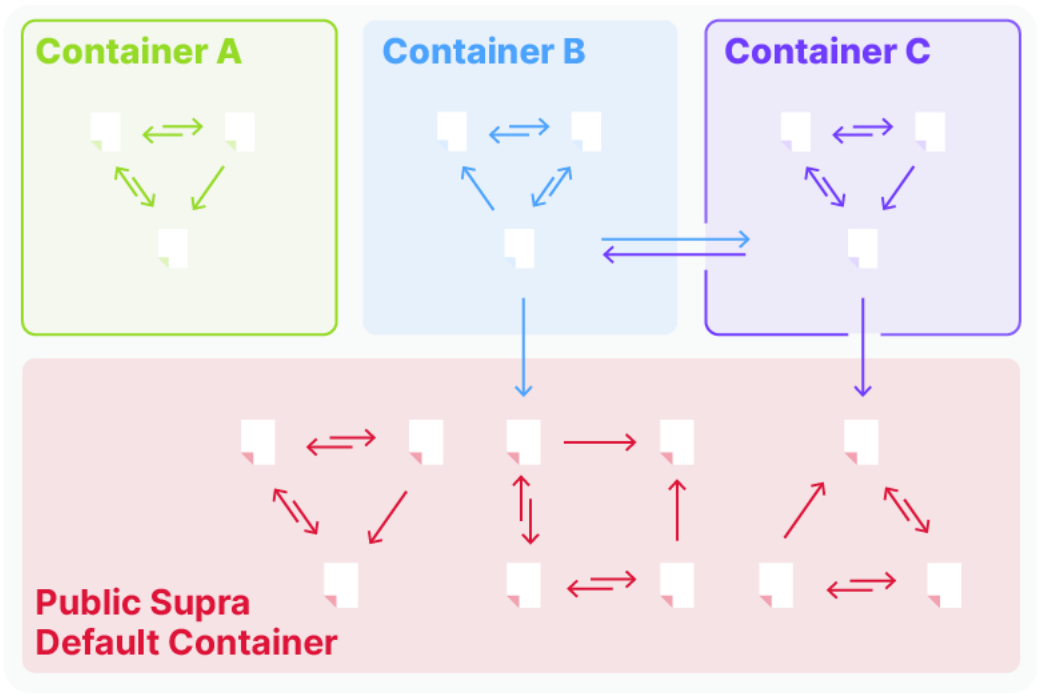

Figure 2 Supra Containers

2.1 Cross-Container Method Calls

It is likely that some Containers may not be fully self-sufficient, meaning that they may want to use some assets or some DeFi applications such as a lending module, or an AMM (Automated Market Maker), or some other on-chain component outside of their Container. It is also not practical to expect that there will be multiple deployments of the same asset or the same DeFi dApp on multiple Containers, and there is also the major issue of liquidity fragmentation. Hence cross-Container method calls become necessary. Though AppChains inherently have isolation by virtue of an exclusive network, Containers are not necessarily isolated constructs, and in fact we posit that interoperability between Containers may be highly desirably.

Major limitation of AppChain. We argue that AppChains have caused the untenable fragmentation of liquidity owing to the difficulty in composing with the services available outside of their respective infrastructure. Users must first bridge their assets and funds to an AppChain, and only then use its services before transferring assets back to the underlying settlement network. We believe this to be a significant limitation of AppChains that results in a confusing user experience.

Unified liquidity. We find that, not only can most features of AppChains be offered using our notion of Supra Containers, but also this significant limitation of non-composability of services across AppChain contracts can be solved with Containers on Supra Layer 1, as they exist upon Supra's shared infrastructure. Thus the cross-Container method calls facilitate unified liquidity. Here, a consumer may hold one account and still be able to avail the services of multiple Containers in a seamless atomic fashion.

2.2 Permissioned Deployment of Smart Contracts

Container administrators are provided with the power to gate the deployment of smart contracts in their Containers through any form of decentralized governance structure they see fit. So, at the time of deploying a smart contract in a Container, the administrator may validate that the cross Container method calls conform to its access policy. They may include any other checks / conditions that are deemed required from the perspective of the respective community's governance structures. This feature facilitates custom governance mechanisms, similar to that of any AppChain or Layer 2, while not requiring the Container administrator to bootstrap node operators and economic security.

2.3 Custom Gas Token and Custom Gas Fees

We recognize that Container administrators will be the primary drivers of attracting interest and traffic to their respective Containers, doing their own marketing and deploying their own incentivization plans. So Container owners need to be rewarded for the successful consumption of their service. Hence we allow the gas fees to be set by the Container administrators. We also credit the gas fees of the transactions targeting a Container to that Container administrator's account. Of course, the host Layer 1 network must also be compensated because the resources and the whole technology stack is actually built and provided by it. So we find that the best approach to satisfy both the Container owners and the network is for the Container owners to receive the fees from the transactions they generate, and then make Container owners pay to the network its share of execution consumption.

Profit and gas sponsorship. This allows the Container owners to set a base gas price (bgp) for their transactions to any amount as long as they are happy with the traffic they are getting. As long as the Container's bgp is more than the bgp of Supra on its global traffic, the Container makes a profit. On the opposite side, a Container could set up a promotional period wherein they deliberately set lower fees or zero fees so that they attract transaction traffic. As the Container owner still has to pay the host Supra network for the transactions, essentially they are sponsoring the users for submitting transactions targeting the respective Container. Further work on $\$SUPRA$ gas futures in the form of auctions are being further explored. In such a case, Container administrators may be able to lock in a fixed price in advance for their gas consumption, thereby further limiting gas fluctuations due to global traffic changes.

Gas fees for cross-Container transactions. Because cross-Container composability is actually encouraged, the number of Containers having modules with cross-Container method calls is expected to be significant. A pertinent question then is: What is the gas token for a transaction accessing many Containers? The options are as follows:

1. User packs the appropriate amount in the custom gas token of each of the Containers to its transaction, so that each accessed Container is paid in its custom gas token for the amount of resources used in that Container.
2. User packs only $\$SUPRA$, the global gas token, and each of the accessed Containers are paid in $\$SUPRA$ for the amount of resources used in that Container.

We choose option 2 so that the user experience is not complicated in packing multiple gas tokens to a cross-Container transaction.

2.4 Execution Space Auctions

Since AppChains are run on a separate secondary network, transaction fees compete only with the traffic on that network. This shields transaction fees from the rise and fall of the global gas fees of disparate AppChain networks.

We observe that all transactions are competing for block space. When there is a high volume of transactions, the transaction fees naturally increase to disincentivize users from submitting additional transactions. It is well known that if a transaction is assured a place in the block, then there is no impact on the global rate of incoming transactions. Hence, a natural way to provide this protection to the transactions targeting a Container is to reserve block space quotas for Containers. Note that the transaction fees may still increase based on the traffic inside Containers. Hence the term **localized fee market**.

The enforcement of block space quotas entails that the network validators know about these quotas as well as respect and validate them at the time of block creation and block validation. Such an enforcement is difficult and impacts performance especially in the setting of Supra where the data dissemination, ordering and execution are decoupled. Because of this decoupling it is not clear at the time of creation of a batch in which block this batch gets placed. Also to reserve a constant amount of block space for a Container in every block, say for a certain period, appears not practical as the pattern of traffic targeting a dApp or a Container may not remain consistently constant across every block. Because of such variation in the container traffic it is likely that block space reservation may result in underutilization of network capacity. For all these reasons it is appropriate to consider execution space for a longer length than blocks.

Our novel work on the economics of commoditising these Container-based execution spaces is presented in [20]. We introduce an entity called *execution space lessee*, who bid and reserve execution spaces for specific Containers for a specific length of time by predicting Container-wise transaction traffic. We argue that this execution space auctioning is a win-win for all the parties involved: Supra – the host network, the Container owners, the execution space lesees and the users alike. The readers are referred to our paper [20] for more details.

We provide the Container administrators the ability to bring in a new or an established fungible asset to be the gas or the transaction fee token for a Container. We argue that this coupling of the value of the services offered by a Container with its transaction fee token helps accrue a greater value to the token and distribute value to the token holders, thus empowering the community behind the Container. We refer to [20] for more details on the benefits of the custom gas token.

2.5 Sovereignty

More generally, a community governing a Container gains sovereignty. The multi-signature administrators represent this community. This community may act as an owner, guardian or caretaker of its Container in various respects. Therefore, the Supra Container framework gives the ability to regulate the deployment and upgrading of smart contracts in its Container. Virtually any form of governance will be possible; it's up to the Container's community.

At the application layer, however, the Container administrators may always implement and enforce an access control policy of their choice. It may place an eligibility criteria for any user to participate and use the services offered by the Container. These criteria could be in the form of a possession of an asset (for example, possession of an NFT related to a real-world asset - RWA), a deposit in a certain vault, or some attributes related to the user (for example, above 18 years of age), etc. Container administrators may implement a desired policy on cross-container method calls typically approved by its community that are checked at the time of module publishing.

We believe that this tight coupling of the value of the services offered in a Container with the ability to govern the deployment of smart contracts, charge their users in a custom gas token at a custom rate, auction its execution spaces in a predictive market, as well as leverage the features, services and the performance offered by the underlying host blockchain network – Supra, makes Supra Containers a very appealing proposition for the developers, institutions, as well as the user community.

3 Benefits inherited from the Supra Platform

We now list the prominent benefits that a Supra Container begets by virtue of being hosted on the Supra blockchain network.

Security. Containers are hosted on Supra's Layer 1 blockchain, meaning they share the security guarantees of the Supra network. Supra's network is a Proof of Stake network with validators posting stake in $*SUPRA*, giving the economic security to the network. Please refer to the Supra Technology Overview whitepaper [23] for more details on the security aspects of the network. There is no exclusive set of validators responsible for Containers, but all Containers share the same validators set from the Supra network, thus inheriting the whole of the economic security of the Supra network.

Performance. Supra's blockchain network is built on the Moonshot consensus protocol - a high throughput, low latency, novel consensus protocol. Supra's *tribes and clans* model [23] yields an elastic network where we can add clans to meet any required throughput demands. In general, we have endeavored to make all the components of Supra network highly performant. So, Containers naturally inherit all these performance benefits.

Vertically-integrated and cross-chain services. As detailed in the Supra Technology Overview whitepaper [23], all the services hosted on the Supra network such as DORA [4] for Oracle feeds, VRF [10] for on-chain randomness, zero-block delay Automation [21], and Supra's IntraLayer [22] for cross-chain services become available for all applications deployed in a Container. Any dApp that deploys on Supra natively has access to these value-added services that do not typically come with AppChains and Layer 2s, further making Supra Containers an attractive alternative.

DevX. Towards making strides in developer experience (DevX), Supra is working to enable the simple plug-and-play creation of Containers. Developers will be presented with options of which smart contract applications their Containers should be initialized with. Developers may choose a lending protocol X from an array of choices, and another Automated Market Maker Y from another set, and build a dApp in their own exclusive Container. Since Supra also vertically integrates a multitude of oracle services directly into its infrastructure, a single development framework and single token, $*SUPRA*, is required to access our built-in oracle price feeds, on-chain randomness, automation service, and cross-chain communication protocols.

4 Leveraging Containers for Parallel Execution

In this section we investigate the benefits of grouping smart contracts into Containers towards optimizing the execution time.

As detailed in our Supra Technology Overview whitepaper [23], the Supra network is amenable to state and execution sharding. Supra network will have validator-committees called *clans* hosting state-shards and each clan will execute transactions related to only the hosted state-shard. In this network architecture, a cross-Container transaction may be of two kinds: intra-shard or cross-shard. Intra-shard cross-Container transactions pose no challenge except for the approaches executing Container transactions in parallel. Executing a cross-shard cross-Container transaction boils down to an intra-network bridging problem. Here we employ a *family* of nodes (set of nodes which guarantees the presence of at least one honest node) from the Supra network to run Supra's HyperNova [17] interoperability solution to execute such transactions. As long as a single virtual machine is not hosted on more than one clan, atomic cross-shard transactions won't be necessary. For the rest of this paper, we scope cross-Container transactions to be of intra-shard in nature.

Consider only the intra-Container transactions in a context where Containers result in a partition of blockchain state. For example on EVM, as each smart contract has its own storage, naturally Containers have their exclusive state space. Then the intra-Container transactions targeting one Container does not conflict with the transactions of other Containers. Consequently such intra-Container transactions targeting different containers may be executed on different cores and hence reduce the execution times.

In such a context, from the perspective of executing transactions, Containers may be seen as an access specification. So, we posit this study amidst the spectrum of input specifications of storage locations that are read by or written to by the transactions. On one extreme end, transactions come with no specifications – we call it the *access set oblivious* model, e.g., Ethereum [8] and Aptos [1] transactions. On the other extreme end, transactions come with full specification of read and write sets – we call it the *access set aware* model, e.g., Solana [24] and Sui [19]. It is well noted by the community that the transaction sizes bloat and occupy more block space with full specifications. We find the specifications provided from the Container abstraction to be in the middle of these extremes. As they do not add any extra information to the transactions themselves, they don't bloat the transactions burdening the network bandwidth to disseminate the transaction data.

Note that the cross-Container transactions cannot be so batched as intra-Container transactions to execute them in parallel. However, if we distinguish transactions at the structural level itself, as intra-Container and cross-Container, then all the cross-Container transactions may be batched together and any parallel execution approaches such as software transactional memory (STM) techniques or dependency graph techniques may be applied for any batch of cross-Container transactions.

In the rest of the paper, we show that the execution time benefits yielding from the intra-Container transactions is significant and is comparable with the full specification model, and hence this abstraction is a secure and efficient optimization.

We now present *Supra-work-sharing queue* (*Supra-WSQ*) scheduler, a simple algorithm that leverages the Container abstraction to achieve maximal parallelism. This algorithm works analogously to the following real-world scenario. Consider that there are many clerks and a stream of incoming clients who require a clerk's signature on a document. The clients form a queue such that whoever is in the front of the queue walks to any available clerk. The clerk gets busy working through the document provided by the client and concludes the session with the client by either signing the document or not. The clerks are never idle as the next client approaches a clerk as soon as the current client finishes his session. This protocol is depicted in Figure 3.

Recall that the Container abstraction requires transactions to target a single Container. We term an ordered list of transactions targeting a single Container as a *batch of Container transactions*, or simply a batch, when the context is clear. Now we apply the aforementioned real-world scenario to our execution model by mapping clerks to CPU cores and threads, and the clients to the batches of Container transactions.

We present the pseudocode of Supra-work-sharing queue scheduler in Algorithm 1. The

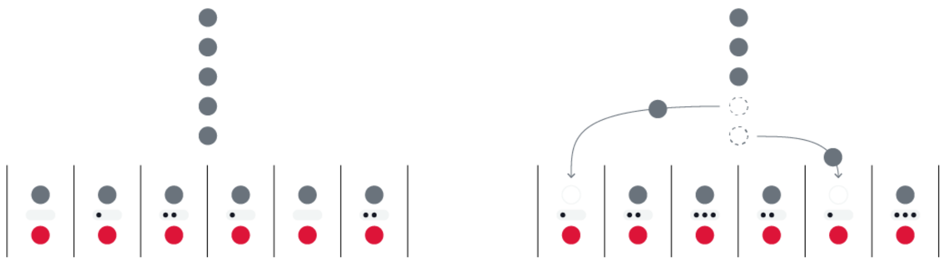

Figure 3 Supra-Work-Sharing Queue (Supra-WSQ) Algorithm

procedure *init* sets all the cores (note that clerks are analogous to cores) to the available state. *Supra-work-sharing-queue* is the main procedure. It takes the batch which is in the front of the batch queue (given by $head(batches)$) and assigns it to an available core (given by $getAvailableIndex(cores)$). Note that 'getAvailableIndex' needs to be an atomic operation. If no cores or threads are available, the algorithm waits until at least one becomes available. Then, the assigned core initializes the execution of its batch. Upon completion of the execution, the core becomes available for execution of subsequent batches in the queue. Because the Supra-work-sharing queue algorithm uses multiple cores for the sequential execution of batches, it is faster than executing all the batches sequentially on a single core when multiple batches from different Containers are to be executed.

Consider an approach in which we utilize a scheduling algorithm or other technique that utilizes all the cores to execute a single batch. After the execution of a batch, the next batch is also executed in a similar fashion. Since we do not preclude the presence of conflicts in the execution of transactions inside a batch, these techniques naturally and necessarily need to account for such conflicts and facilitate their resolution using *aborts* and *retries*.

As there are no aborts and retries in the Supra-work-sharing queue algorithm, it executes even faster than these techniques in the presence of many conflicts among transactions. The important thing to note is that these techniques we are not leveraging the specification given by the Container , so it is not truly *apples-to-apples* comparison. Nevertheless, because the Supra-work-sharing queue algorithm leverages this specification based on Container abstraction, it executes faster.

Algorithm 1 Supra-Work-Sharing Queue Scheduler for Containerized Batch Execution.

```
Procedure init():
    forall i: cores[i] = available
Procedure Supra-work-sharing queue(cores[], batches[]):
    while batches not empty do
        execute(head(batches), getAvailableIndex(cores));
Procedure execute(b, ci):
    // The core cores[ci] starts executing the batch b.
    cores[ci] = busy;
Procedure complete(ci):
    b = batchBeingWorked(ci);
    remove(b, batches);
    cores[ci] = available; // The core cores[ci] finished executing b and is available.
Procedure getAvailableIndex(cores):
    wait if no ci: cores[ci] = available;
    return ci: cores[ci] = available;
```

To reiterate, it is not the superiority of the Supra-work-sharing queue algorithm that gives an advantage in execution times, but rather the facility of the specification given by the Container abstraction.

5 Parallel Execution of Containerized Transactions - Experiments

In this section we study the execution latency and execution throughput of the aforementioned Supra-work-sharing queue (Supra-WSQ) approach by executing on multiple cores.

As discussed in Section 4, Containerization groups transactions into different batches, each of which is executed sequentially, while batches of different Containers are executed in parallel. The sequential execution of transactions within each batch will overcome the abort and re-execution overhead of speculative or optimistic execution. Hence, we can expect that batching will outperform speculative execution when the workload is inherently sequential due to high contention, meaning transactions inside each batch are sequential by nature and form a chain of conflicts. As a future work, we plan to leverage container specification in conjunction with other parallel execution approaches like STM.

5.1 Experimental Setup and Implementation

We ran our experiments on a single-socket AMD machine consisting of 32 cores, 64 vCPUs, 128 GB of RAM, and running the Ubuntu 18.04 operating system with 100 GB of SSD.

We design these experiments to evaluate Containerized parallel execution with state-of-the-art approaches like Aptos's Block-STM [9], Solana's SeaLevel [25], and Ethereum's baseline sequential execution [8]. The experiments are performed on the benchmarking setup [2] of the Block-STM simulating in-memory (no access to persistent storage) read and write operations on Move-VM.

The input test workloads for all three approaches are the same. The Supra-work-sharing queue approach processes the transactions by grouping into batches and executing them in parallel across batches. The transactions from those batches are concatenated to form a list of transactions (preset serialization order) and given as input to Block-STM and SeaLevel.

5.2 Construction of Synthetic Tests

We now discuss how workloads are constructed and grouped into multiple non-conflicting batches.

Synthetic workload means the workload of simulated or constructed transactions. We analyzed historical transactions (≈30 million) from Solana's Mainnet to better understand the account access patterns in order to construct a synthetic workload. We observe that transactions have a tail distribution, meaning that the majority of accesses in a block are made by a few accounts that are responsible for the majority of conflicts (read-write, write-write, and write-read), while other accounts have just a few accesses.

We construct a block to be made up of several batches. A batch is a group of transactions, and each transaction operates (reads and writes) on keys randomly chosen (from the batch-assigned keys) following a log-normal distribution [11]. Here, we considered four parameters to generate the workloads: operations per transaction, transactions in the block, batches in the block, and the number of keys per batch. These parameters are interrelated, and changing them can increase or decrease conflicts in the block.

- Naturally, the number of conflicts in a batch increases as we decrease the number of keys. Further, increasing the number of operations per transaction increases the number of conflicts in a batch. We construct the *high conflict workload* (W_h) by allocating just 10 keys per batch and performing 20 operations per transaction.We include 10 transactions per batch and 100 such batches in a block. This results in a workload with high-conflicts (more conflicts) pushing the batch towards sequential execution. The transactions in every batch operate by accessing a key space of size 10^3 randomly (following the log-normal distribution).
- When 10^3 keys are allocated per batch with 2 operations per transaction we find that the conflicts reduce, and so we generate a *moderate conflict workload* (W_m). Here we include 10 transactions per batch and 100 such batches in a block.
- When we increase the key space allotted per batch to 10^5 with a uniform random key distribution for transactions to access them, the number of conflicts decreases further. We construct the *low conflict workload* (W_l), by fixing 2 operations per transaction, 10 transactions per batch, and 100 such batches in the block, where each batch is allocated 10^5 keys.

All the above mentioned workloads are succinctly represented in Table 1.

5.3 Assumptions

Our experiments and evaluation are under the following assumptions:

A1 There are more batches in the block than the number of cores available to start with.

A2 The batches contain transactions targeted for a single Container, meaning there are no cross-Container transactions. Cross-Container transactions have to be dealt separately.

A3 The workload is homogeneous, meaning batches contain an equal number of transactions.

5.4 Evaluation

For each workload (from Table 1), we execute blocks for Supra-work-sharing queue (Containerized batching), Aptos's Block-STM, Solana's SeaLevel, and sequential execution. The sequential execution serves as a baseline. The histogram plots in Figure 4 − Figure 6 show an average execution *throughput (tps)* on the Y-axis, while the number of threads varies on the X-axis from 2 to 32. The experiments were carried out 52 times; the first 2 executions are left as warm-up runs, and each data point in the histograms is an average of 50 executions.

The histogram plots for the average tps on high conflict workload W_h can be seen in Figure 4. As shown, Supra-work-sharing queue outperforms as the number of threads increases since there is no speculative execution overhead of abort and re-execution. Due to increased contention at the shared queue, there is a decline in performance for Supra-work-sharing queue at 32 threads. Although throughput for both Block-STM and SeaLevel increases with threads, Block-STM performance falls short of Supra-work-sharing queue due to increased aborts and re-execution (re-validation) overhead; similarly, in SeaLevel, there

Table 1 Synthetic workload

Workload	Oprs/Tx	Txs/Block	Batches/Block	Keys/Batch
High Conflict (W_h)	**20**	1000	100	1000
Moderate Conflict (W_m)	**2**	1000	100	1000
Low Conflict (W_l)	2	1000	100	**100000**

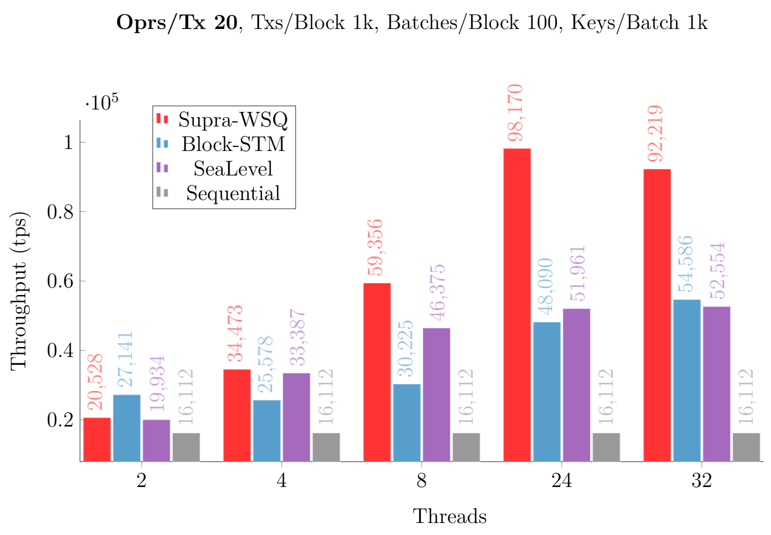

Figure 4 Synthetic workload W_h: high conflicts within each batch

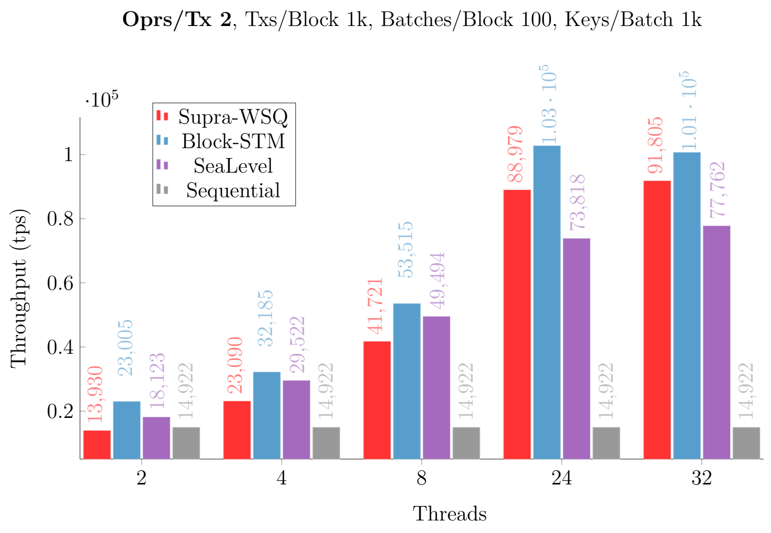

Figure 5 Synthetic workload W_m: moderate conflicts within each batch

will be more iterations with fewer transactions per iteration. The optimal throughput for Supra-work-sharing queue is $\approx 98k$ tps at 24 threads. Block-STM and SeaLevel reach a peak throughput of $\approx 54k$ and $\approx 52k$ tps at 32 threads, respectively.

The workload W_m has fewer conflicts than W_h. Figure 5 shows that all three of the parallel execution approaches outperform sequential execution in this case and perform neck-to-neck with one another. The execution overhead in Block-STM is moderate because there aren't many conflicts; likewise, SeaLevel will have a moderate number of iterations with multiple transactions. Block-STM performs more effectively with an optimal tps of $\approx 102k$ at 24 threads. At 32 threads, the maximum throughput of Supra-work-sharing queue is $\approx 91k$, while SeaLevel reaches $\approx 77k$ tps.

Figure 6 shows the histogram plot for W_l with low conflicts in the batches. The plots demonstrate how the performance of Block-STM, Supra-work-sharing queue, and SeaLevel

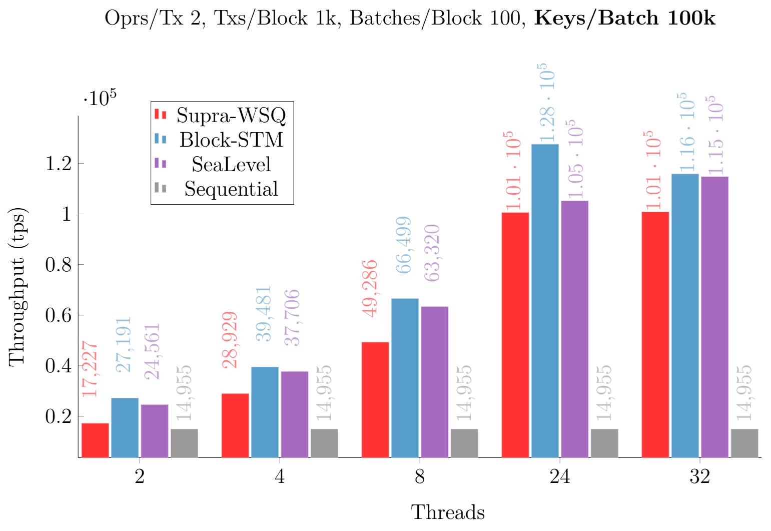

Figure 6 Synthetic workload W_l: low conflicts within each batch

increases with the number of threads, peaking at $\approx 127k$, $\approx 100k$, and $\approx 114k$, respectively. All three approaches have the maximum throughput because blocks have high concurrency in this workload, in other words, less contention. Block-STM outperforms other approaches as it utilizes the threads in a more balanced way than Supra-work-sharing queue and SeaLevel.

Since there are no aborts and re-executions in Containerized batching, in the presence of many conflicts among batch transactions (as we have seen in Figure 4), it outperforms other techniques. Moreover, when there are moderate (Figure 5) or fewer conflicts (Figure 6), it matches with the performance of other parallel execution approaches. The proposed Supra-work-sharing queue approach achieves $\approx 6\times$ speedup over sequential execution under all synthetic workloads.

5.5 Testing on Real-World Transactions

We downloaded $\approx$30 million historical transactions from publicly available 15k blocks (205465000 – 205480000) from Solana [24] using the `getBlock()` JSON-RPC Solana API of Quicknode [16]. We chose Solana because transactions are made up of account access information, an array of accounts to read from (*read-set*) or write to (*write-set*) [18]. This information aids in the construction of the batches for containerized execution. The real-world block transactions show heterogeneity in the workload, meaning when transactions are batched they are not necessarily of the same sizes. Transactions may also not all have the same number of reads and writes unlike in our synthetic tests.

We show the result of two experiments here: one on a randomly selected block - Block number 205465004 of Solana blockchain, another on a block constructed by concatenating consecutive 50 blocks from Block number 205465000 to 205465049 of Solana blockchain. The Table 2 show more details of these tests. We remove the Solana voting transactions from these blocks. The containerized batches are constructed by placing conflicting transactions in different batches. For any two transactions i and j if either: $\textit{read-set}_i \cap \textit{write-set}_j$ or $\textit{read-set}_j \cap \textit{write-set}_i$ or $\textit{write-set}_i \cap \textit{write-set}_j$ is non-empty then they are placed in different batches.

The histogram plots for the throughput in average tps on block 205465004 from Solana

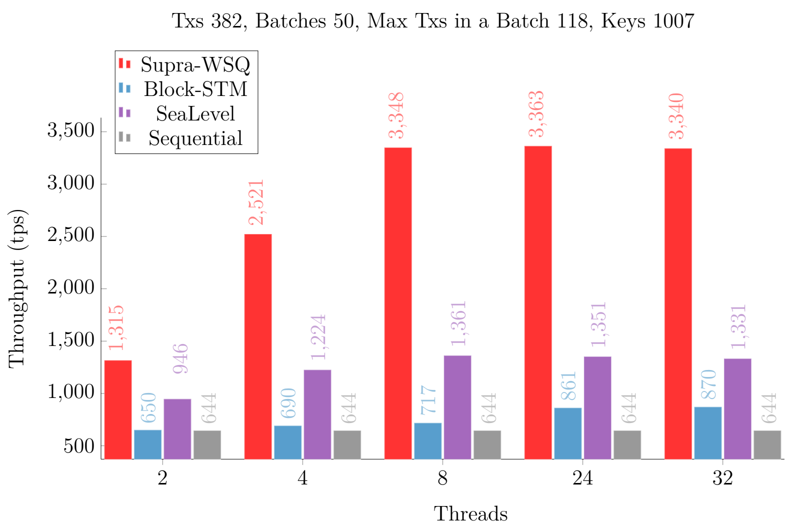

Figure 7 Historical workload W$_{ht}$: Block 205465004

are shown in Figure 7. As shown, the containerized abstraction proves more valuable as the number of threads increases since there is no speculative execution overhead of abort and re-execution. Although throughput for both Block-STM and SeaLevel increases with threads, the Block-STM performance falls short of Supra-work-sharing queue due to increased aborts and re-execution (re-validation) overhead; similarly, in SeaLevel, there will be more iterations with fewer transactions. The average throughput for Supra-work-sharing queue is $\approx 3k$ tps. Block-STM and SeaLevel reach a peak throughput of $\approx 1k$ and ≈ 800 tps, respectively. Similarly, on the concatenated Solana blocks: from 205465000 to 205465049, the performance trends remain the same for Supra-work-sharing queue and Block-STM, reach a peak throughput of $\approx 3.4k$ and ≈ 642 tps, respectively, as shown in Figure 8. However, the SeaLevel performance has seen upward trends and achieves an optimal throughput of $\approx 1.5k$ tps, due to the increased number of transactions per iteration.

In summary, the analysis of real-world transactions of the Solana blocks shows that the container abstraction provides more value in parallel execution than existing state-of-art parallelization techniques.

6 Conclusion

Inspired by AppChains, we introduced Supra containers and showed that it overcomes the main limitation of AppChains, which is fragmented liquidity.

Coupled with the features of gated deployment of smart contracts, custom gas token and gas pricing, execution space auctioning, and of atomic composability we posit Supra

Table 2 Solana Historical non-voting transaction workload (W$_{ht}$)

Block Number	Oprs/Txs				No. of Batches	Maximum Txs in a Batch	Total Txs	Keys
	Average		Maximum					
	Reads	Writes	Reads	Writes				
205465004	5	7	23	37	50	118	382	1007
205465000 - 205465049	6	7	34	41	171	1321	5626	5354

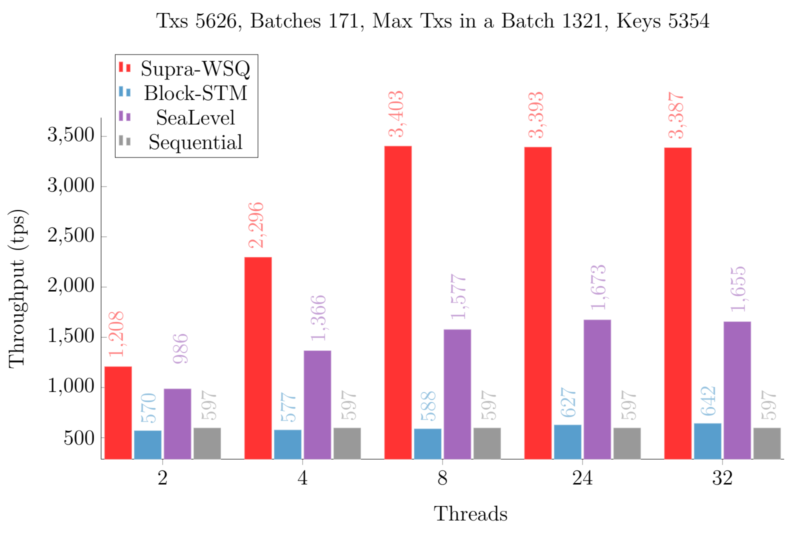

Figure 8 Historical workload W_{ht}: Block 205465000 - 205465049

containers as an appealing proposition for developers, institutions and user communities. The host Supra network also boosts this value proposition of Supra Containers with its high performance and vertical integration of varied services.

We observe that the Container abstraction technique may be used as access specification that can be leveraged towards executing transactions in parallel. Through our experiments we demonstrate that such access specification gives a significant improvement in performance comparable to known parallel execution techniques and in some scenarios outperforms them too.

References

1 The aptos blockchain: Safe, scalable, and upgradeable web3 infrastructure. `https://aptos.dev/assets/files/Aptos-Whitepaper-47099b4b907b432f81fc0effd34f3b6a.pdf`, August 2022.

2 Block-stm implementation. `https://github.com/danielxiangzl/Block-STM`. [Online: accessed 15 January 2024].

3 Celestia. `https://celestia.org/`. [Online: accessed 25 September 2024].

4 Prasanth Chakka, Saurabh Joshi, Aniket Kate, Joshua Tobkin, and David Yang. Oracle agreement: From an honest super majority to simple majority. In *43rd IEEE International Conference on Distributed Computing Systems, ICDCS 2023, Hong Kong, July 18-21, 2023*, pages 714–725. IEEE, 2023. `doi:10.1109/ICDCS57875.2023.00025`.

5 Defi kingdoms. `https://defikingdoms.com/`. [Online: accessed 3 September 2024].

6 Docker. `https://www.docker.com/`. [Online: accessed 25 September 2024].

7 dydx v4. `https://dydx.exchange/blog/dydx-chain`. [Online: accessed 3 September 2024].

8 Ethereum (ETH): open-source blockchain-based distributed computing platform. `https://www.ethereum.org/`. [Online: accessed 15 January 2024].

9 Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 232–244, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3572848.3577524`.

10 Aniket Kate, Easwar Vivek Mangipudi, Siva Maradana, and Pratyay Mukherjee. Flexirand: Output private (distributed) vrfs and application to blockchains. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1776–1790, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3576915.3616601.

11 Log-normal distribution. https://en.wikipedia.org/wiki/Log-normal_distribution. [Online: accessed 13 Jun 2024].

12 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://www.bibsonomy.org/bibtex/23db66df0fc9fa2b5033f096a901f1c36/ngnn, 2009.

13 Cosmos Network. Cosmos network documentation, 2024. [Online: accessed 30 September 2024]. URL: https://docs.cosmos.network/.

14 Optimism stack. https://docs.optimism.io/builders/chain-operators/tutorials/create-l2-rollup. [Online: accessed 25 September 2024].

15 Osmosis dex. https://osmosis.zone/. [Online: accessed 25 September 2024].

16 Quicknode: Solana rpc. https://www.quicknode.com/docs/solana. [Online: accessed 10 January 2024].

17 Supra Research. Hypernova: Efficient, trustless cross-chain solution. Technical report, https://supra.com, 08 2023. URL: https://supra.com/docs/Supra-HyperNova-Whitepaper.pdf.

18 Solana documentation. https://docs.solana.com/. [Online: accessed 10 January 2024].

19 Sui documentation: Discover the power of sui through examples, guides, and concepts. https://docs.sui.io. [Online: accessed 10 January 2024].

20 Commoditized block space. [Upcoming, as of 25 September 2024].

21 Supra zero block delay native automation. [Upcoming, as of 25 September 2024].

22 Supra's defi intralayer. [Upcoming, as of 25 September 2024].

23 Supra technology - an overview. [Upcoming, as of 25 September 2024].

24 Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper*, 2018.

25 Anatoly Yakovenko. Sealevel - parallel processing thousands of smart contracts. https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192, September 2019. [Online: accessed 23 January 2024].