Transaction Scheduling for Solana

Pranav Garimidi

July 27, 2022

1 Model

Solana faces the following scheduling problem in parallelizing the execution of transactions:

- We start by considering the scheduling problem for a single block with a fixed pool of transactions T [1]
- Each transaction i consists of
  - f_i, A transaction fee;
  - c_i, How many CU[2] the txn consumes;
  - R_i, a list of accounts it will read from;
  - W_i, a list of accounts it will write to;
- Each transaction i takes an amount of time t_i to process. t_i is unknown to the scheduler prior to actually processing the transaction but c_i acts as a correlated signal. For now we make the simplifying assumption $t_i = c_i$. We could also imagine a model such as $t_i = c_i + \epsilon_i$ where $\epsilon_i \sim \mathcal{N}(0, \sigma)$ for some $\sigma > 0$.
- There are k threads to schedule jobs on with each thread having a capacity C of maximum CU units the thread can schedule per block
- Let Q be the set of transactions scheduled before i on the thread i is assigned to. Then i starts processing at time $\sum_{j \in Q} t_j$ and finishes processing at time $t_i + \sum_{j \in Q} t_j$
- If transaction i writes to account a, then no other transaction j that either reads from a or writes to a can be processed at the same time i is being processed [3].
- We seek to maximize the total revenue that can be included in a block given these constraints [4]

While we start with this simplified model, we note the following technical difficulties not covered

[1] In reality this pool is dynamically updating with transactions arriving on the fly and transactions expiring after waiting long enough to be scheduled but we start by analyzing the static case

[2] CU is the equivalent of gas in Solana

[3] Note if both i and j are only reading from a and not writing to a then it is fine for them to be simultaneously scheduled across different threads

[4] Solana can't force any validator to use a scheduler so it's ideal that this scheduler is incentive compatible for validators to run

- The scheduler sending a single transaction at a time to each thread to be processed is inefficient. It is better engineering wise for batches of transactions to be sent to threads for completion. The implication being, given c_i as a noisy signal, it may not be optimal to wait for single transactions to finish processing to have exact knowledge of completion time before sending that thread more transactions.
- In order to maximize throughput, it is desirable that threads have a constant queue of transactions they can process instead of ever needing to wait for the scheduler to send them new transactions upon completion of old transactions
- In our model, the scheduler has a global view over the state of each of the threads at any given time, but in reality the scheduler only gains information when threads send the scheduler information
  - There is some overhead to sending this information so minimizing the amount of info needing to be sent is ideal
  - The current implementation idea in Solana is that a thread completes processing a batch of transactions and sends the completed batch back to the scheduler as a signal it has finished processing those transactions.

2 Old Scheduler Design

The old Solana design is to schedule transactions onto threads in an arbitrary manner to make sure every thread always has transactions to schedule. There is no structured sense in which threads prioritize avoiding conflicts between themselves or by which transactions are processed according to fee-priority. If a thread runs into read/write locks while it is processing a transaction, it simply retries later. By not thinking about conflicts when assigning transactions to threads, the threads would often stall when there was high contention. The new scheduler design looks to take these conflicts and fee prioritization into consideration allowing the threads to maintain ideal up-time while maximizing transaction fee revenue.

3 New Proposed Scheduler Design

The proposed design idea Solana's team has for their scheduler is to use a greedy approach w.r.t transaction fees to create batches of transactions that are then assigned to different threads. These batches are constructed with conflicts in mind so that when a batch is being constructed for a thread, none of the transactions in that batch will have conflicts with any transactions scheduled to be processed on any other thread. We formally outline this process below[5]

3.1 Setup

Every unprocessed transaction i is either scheduled, or pending. i is scheduled if i is in a batch that has been assigned to a thread. Every other transaction is pending and stored

[5]The scheduler as outlined is logically equivalent to the planned implementation but certain implementation details pertaining to data structures/how certain operations are done are omitted or may differ from how they are described here. For detailed implementation details look at https://github.com/solana-labs/solana/pull/26362 (implementation around how transactions are blocked thread by thread has not been done yet)

in a list P ordered in decreasing fashion according to the ratio f_i/c_i. Additionally if i is pending, it can be in one of three states, unconstrained, uniquely constrained, or blocked. i is unconstrained if it is eligible to be scheduled on any thread, i is uniquely constrained if there is one thread it can be scheduled on, and i is blocked if there are no threads i can be scheduled on. i will be unconstrained when i has no conflicts across any currently scheduled transactions, i will be uniquely constrained if it has conflicts only with transactions scheduled on a single thread, and i will be blocked if it has conflicting transactions scheduled on two or more distinct threads. We define the state i is in via a variable s_i where $s_i = 0$ if it is unconstrained, $s_i = a$ if i is uniquely constrained to be scheduled on thread a, and $s_i = -1$ if i is blocked. The idea here is that if i only has conflicts on a single thread, then it can safely be scheduled to that thread since transactions are executed serially across a single thread.

We describe the scheduler for the offline model as described above, so P is initialized to include all the transactions in T. We also define an approximate CU per batch limit G. G can be imagined to be a small but not trivial fraction of C.

3.2 Batch Creation

With this setting, we define the subroutine $BuildBatch(P, a, G)$ to greedily build a batch of transactions with total CU approximately G from transaction list P for thread a as follows:

1. The batch B is initialized to the empty set.
2. The scheduler iterates through P in order examining each transaction i
   (a) If $s_i = 0$ or $s_i = a$, the scheduler adds i to B and removes i from P. Now the scheduler iterates through every other transaction $i' \in P$. If i' conflicts with i (formally if $W_i \cap (W_{i'} \cup R_{i'}) \neq \emptyset$ or $R_i \cap W_{i'} \neq \emptyset$) then if $s_{i'} = 0$ update $s_{i'} \leftarrow a$ otherwise if $s_{i'} \neq a$ then update $s_{i'} \leftarrow -1$. If $s_{i'} = a$ then it can be left as is.
   (b) If adding i to B causes B's CU to pass G then break
3. Return B

3.3 Scheduler Algorithm

Using the batch creation subroutine, the scheduler runs as follows.

1. Each thread a is initialized to have an empty queue of batches
2. For each $a \in [k]$, run $BuildBatch(P, a, G)$ twice and send the 2 batches to thread a[6]
3. When a thread a finishes processing a batch B of transactions, a removes B from its queue and sends the scheduler a completion signal
   (a) Upon receiving a completion signal for B from a thread a, the scheduler iterates through P and for each transaction $i \in P$, updates s_i accordingly to the state i should be in with the conflicts from B on thread a now gone. This is done efficiently by keeping track of which scheduled transactions are blocking which

[6]Once a thread receives a batch, it starts processing immediately and continues processing until it runs out of transactions.

pending transactions on certain threads [7] so when all the transactions i blocking transaction i' from being scheduled are processed, i' can become eligible again to be scheduled.

(b) The scheduler then calls $BuildBatch(P, a, G)$ to build a new batch B'

(c) If processing B' would not push thread a past processing more than C CU, then the scheduler sends B' to a's batch queue.

4. The scheduler continuously listens for completion signals for threads until a block is produced

In the model where T is dynamically updating , when a transaction i arrives, i is added to P and the scheduler checks which currently scheduled transactions i would have conflicts for to initialize s_i.

3.4 Analysis

Intuitively the worst case performance of this greedy approach is $\frac{1}{k}$ when all the c_i are small with respect to C. The worst case happens when there is one set of transactions that conflict with every other transaction but only have marginally higher ratios of f_i/c_i. Then all these transactions get greedily scheduled on the same thread and block any other transactions from being scheduled on any of the other threads. In general this suggest the problem with the greedy approach is when transactions that have marginally higher fees but many conflicts are scheduled with higher priority than similar fee transactions with very few conflicts.

[7] Some more detailed implementation details in https://github.com/solana-labs/solana/pull/26362 although details around keeping thread by thread conflicts not currently described