slug: parallel-builder title: Parallel block building authors: [robert] tags: [engineering] image: /img/posts/parallel-builder/conflict_nodes.png hide_table_of_contents: false description: We introduce a new approach to block building called parallel block building. Instead of treating all transactions as potentially conflicting - as traditional sequential building algorithms do - parallel block building recognizes that most transactions in a block are actually independent. When a user swaps ETH for USDC, it doesn't affect someone minting an NFT - so why process them sequentially?

In this blog post we introduce a new approach to block building: **parallel block building**. Instead of treating all transactions as potentially conflicting - as traditional sequential building algorithms do - parallel block building recognizes that most transactions in a block are actually independent. When a user swaps ETH for USDC, it doesn't affect someone minting an NFT - so why process them sequentially?

Parallel building doesn't. It identifies up front what groups of transactions truly conflict, and then processes these conflicts in parallel. Parallel processing gives us a few key benefits:

- **Faster building**: by isolating what transactions conflict we can process independent conflict groups simultaneously, cutting down on overall latency
- **Better ordering:** multiple building algorithms can be tried in parallel, and no single algorithm needs to be applied to the entire block
- **Resource maximization:** available computing power flexibly shifts between essential and speculative optimizations, ensuring resource usage is maximized
- **Pipelining**: key components operate independently and concurrently - from orderflow processing to conflict resolution to block assembly

This post describes our open source implementation and shares early results from testing: in production environments it's already building more valuable blocks 19% of the time and in backtesting it shows potential for up to 50%.

Let's dive into how it works.

Thank you to Robert Anessi for many ideas that inspired this design, as well as Daniel and Vitaly for their contributions.

Architecture

The above is our architecture at a high level

- **Conflict finding**: identifies and groups transactions with overlapping storage reads and writes, allowing us to separate groups of conflicting transactions for resolution.
- **Task generation**: creates tasks to resolve conflicts and helps to strategically manage resource utilization.
- **Conflict resolution**: a pool of workers with a shared task queue try to resolve conflicts in parallel.
- **Block assembly**: assembles the final block by combining the best resolved conflicts and independent transactions.

Below we'll walk through each component in more detail.

Conflict finding

The first step, conflict identification, isolates groups of transactions that may conflict based on shared storage reads and writes. Our implementation uses a `ConflictFinder` that quickly groups transactions by maintaining mappings between storage slots and transaction groups. For each transaction it identifies potential conflicts, and based on the number of conflicts it takes one of three paths:

- **No Conflict:** The transaction forms a new independent group.
- **Single Conflict:** The transaction joins an existing group.
- **Multiple Conflicts:** The transaction merges all conflicting groups into a single, larger group.

The result is a set of `ConflictGroup` structures, each containing its member transactions for resolution.

rust /// ConflictGroups describes set of conflicting orders. pub struct ConflictGroup { pub id: usize, pub orders: Arc<Vec<SimulatedOrder>>, pub conflicting_group_ids: Arc<HashSet<usize>>, }

Task generation

Once conflict groups are defined, the `ConflictTaskGenerator` generates prioritized tasks to resolve them. Rather than a single strategy, we explore multiple approaches concurrently, which provides several benefits:

1. **Many parallel strategies:** Different conflict groups can be optimized with different algorithms, so we run multiple algorithms simultaneously and take the best result per conflict group.
2. **Quick results + deep search:** Greed heuristics yield quick initial solutions, while in parallel we can run more intensive

algorithms to potentially find higher-value results later on.
3. **Resource maximization:** Worker threads should never sit idle. When high-priority task complete, workers can shift focus to lower-priority, speculative tasks.

Here's the structure that we use to define each task

```rust /// ConflictTask provides a task for resolving a [ConflictGroup] with a specific [Algorithm]. pub struct ConflictTask { pub group_idx: usize, pub algorithm: Algorithm, pub priority: TaskPriority, pub group: ConflictGroup, pub created_at: Instant, }

/// Defines a task's priority pub enum TaskPriority { Low = 0, Medium = 1, High = 2, }

/// Algorithm provides an algorithm for resolving a [ConflictGroup]. pub enum Algorithm { Greedy, // Max gas price, max profit ReverseGreedy, Length, AllPermutations, Random { seed: u64, count: usize }, } ```

This flexible approach allows us to try many algorithms and handle their prioritization effectively, balancing between quick and thorough results.

**Conflict resolution**

Conflict resolution leverages a pool of worker threads, each pulling from a shared task queue. Each thread executes the designated algorithm for its task in parallel, then writes the result to a shared "best results" store. A key optimization here is **simulation caching**: when evaluating different orderings of the same transactions, we reuse previously computed results for common sub-sequences, greatly reducing redundancy.

```rust /// This isn't the actual code, but its demonstrative of the core process self.thread_pool.spawn(move || { while !cancellation_token.is_cancelled() { if let Some(task) = task_queue.pop() { // Process task and find best ordering let result = process_task(task); result_sender.send(result); } } });

...

/// Collects and manages the best results for each group in a concurrent environment. pub struct ResultsAggregator { result_receiver: std_mpsc::Receiver<(GroupId, (ResolutionResult, ConflictGroup))>, best_results: Arc, }

/// Shared best results instance pub struct BestResults { pub data: DashMap, pub version: AtomicU64, } ```

The `ResultsAggregator` monitors results send by workers and keeps only the highest-value solution per conflict.

**Block assembly**

After conflict resolution, the `BlockBuildingResultAssembler` combines the best solutions from each conflict group with independent transactions to create a final block. This step is straightforward: since conflicts have been resolved, the assembly process involves minimal ordering constraints and we can simply merge groups together.

## Results

Testing the parallel builder has yielded promising results.

**Backtesting in a high MEV environment**

Across 2000 blocks on a high MEV day (where complex ordered blocks are more likely) in backtesting the parallel builder builds better blocks than just greedy algorithms **50**% of the time, producing 40 additional ETH of value. Interestingly, most of this additional value was concentrated in a small number of blocks: the top 20 blocks yielded 25 additional ETH and the top 75 yielded 35 ETH.

A similar distribution can be observed if we look at the top blocks by % increase in block value, with a heavy concentration in a small % of blocks and a sharp drop off in value.

This pattern suggests that a small number of blocks benefit significantly from more complex ordering, while the majority of blocks see little to modest benefit.

Finally, we observed room for our implementation to improve as well: some transactions are being included by the greedy algorithms that aren't by the parallel builder. In theory this shouldn't happen, so this clearly indicates a bug somewhere.

**Backtesting in a low MEV environment**

Across 2000 blocks in backtesting in a low MEV environment, 60% of blocks the parallel builder produced better results than either greedy algorithm. However, there was very little additional value from these improvements. At best there was 0.05 additional ETH in a block, but the median block with additional value was less than 0.005 ETH more profitable. These

results indicate that a greedy algorithm is sufficient when there is less activity, which makes intuitive sense.

**Results from a live production (low MEV) environment**

In the first week of production deployment - a relatively low MEV time period - the parallel builder built better blocks 19% of the time. For a small portion of blocks it was much more profitable: in 3% of blocks the parallel builder was 100% more valuable than the next best algorithm! This roughly matches what we saw in backtesting where better merging disproportionally impacts a small number of blocks.

However, the fact that the parallel builder is only producing better blocks 20% of the time instead of 50-60% suggests that the live version has room for improvement. Intuition suggests that this is because of the latency overhead of finding conflicts and async communication, but more analysis is needed.

Overall we think these results show great promise for this model of block building, especially as more complicated and purpose built merging algorithms are implemented.

## Future work

There are many exciting future lines of work on top of this architecture:

- **Outsourcing merging:** Allowing external searchers to handle conflict resolution, leveraging a permissionless structure similar to bottom-of-block searchers in TDX. We are in the early stages of designing this system. Please reach out to @astarinmymind on Telegram if you are interested in it.
- **New algorithms**: The parallel architecture allows for advanced strategies like application-specific merging rules (e.g., DEX-specific rules) and exhaustive algorithms that weren't feasible with sequential processing.
- **Improved conflict identification:** Improving the ConflictFinder to provide finer-grained data on transaction dependencies could reduce the search space for optimal ordering.
- **Better cancellation handling**: Improve the handling of cancellations in conflict identification, as well as how that cascades to other parts of the system (pending tasks, best results store, etc).
- **Block caching:** Similar to simulation caches, caching entire blocks for ultra-fast new blocks when non-conflicting order flows are updated.
- **Handling variable execution**: transactions that change their execution in different places in a block are challenging to handle because they make proper conflict identification hard. We can improve how we identify and deal with these cases.

## Join us

- The code is open source if you want to take a look or to contribute.
- We're hiring staff level engineers for the builder to work on similar industry leading ideas. Feel free to reach out if you're interested and want to learn more :)
- We have a bunch of other engineering and product positions open
- If you are interested in running a merging algorithm on private flow, similar to searching in TDX, please reach out to @astarinmymind on Telegram.