A1: Parallel programming with OpenMP

Arib Ashhar Shamoon | 2025MCS2013

1. Design Decisions

1.1 updateDisplay()

Design Approach: The function employs a hybrid sequential-parallel approach. Order processing is performed sequentially to avoid race conditions when updating stock states. Snapshot generation is parallelized using OpenMP's dynamic scheduling, as each snapshot is independent once the state is captured **StockInfo** Structure is used to store each transaction.

Parallelization Strategy: After building all snapshot states sequentially, the generation of snapshot files is distributed across threads using `#pragma omp parallel for schedule(dynamic)`. Dynamic scheduling ensures load balancing, as different snapshots may have varying numbers of stocks to process and sort.

1.2 totalAmountTraded()

Design Approach: This function implements an parallel algorithm using OpenMP's reduction clause. Each thread independently decodes packets and computes trade amounts, with the runtime system handling the accumulation of partial sums.

Parallelization Strategy: The implementation uses `#pragma omp parallel for reduction(+:totalAmount) schedule(static)` with static scheduling for optimal cache performance. Since all operations are independent and uniform in cost, static scheduling minimizes thread synchronization overhead.

1.3 printOrderStats()

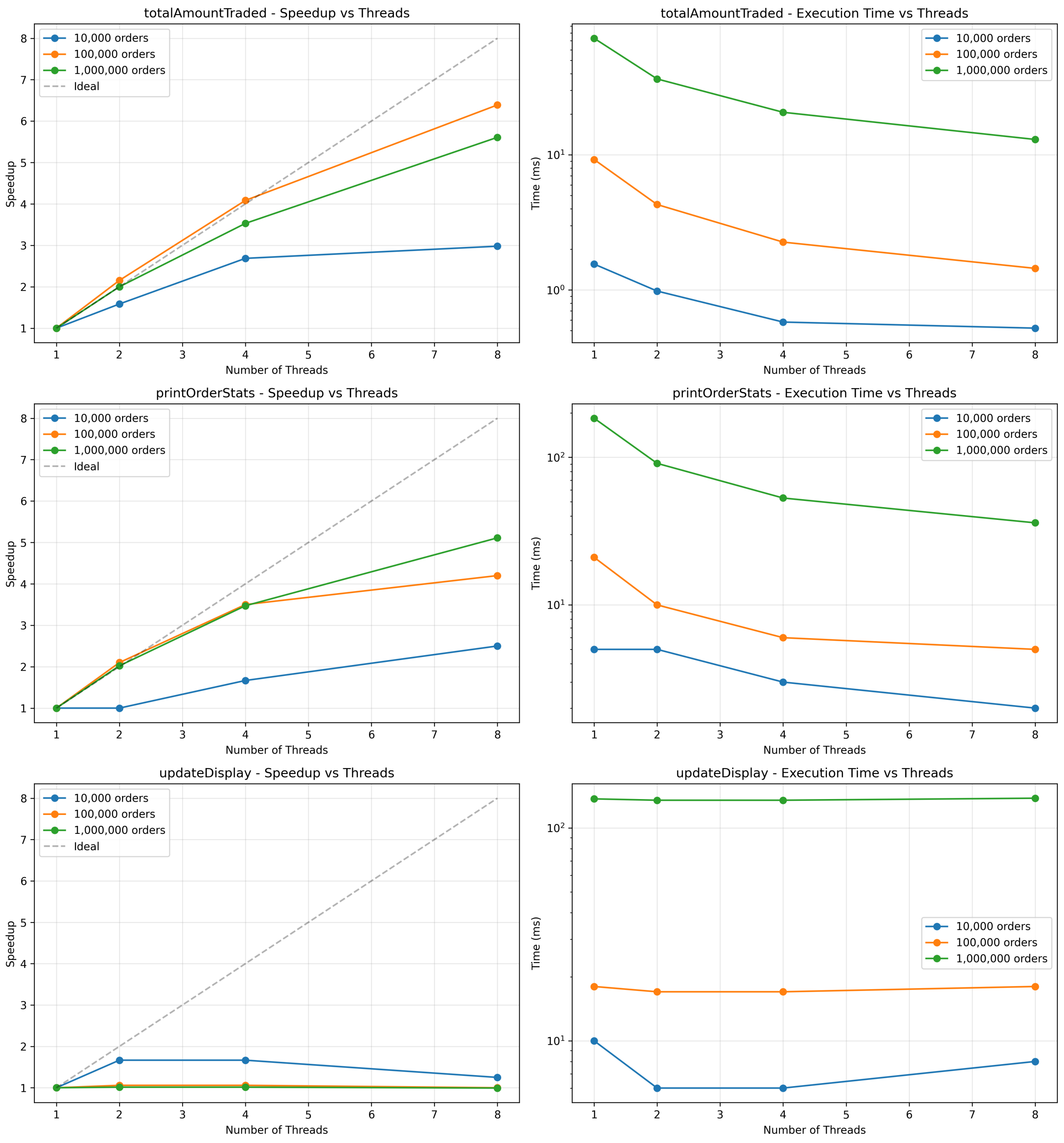
Design Approach: This method implementation uses thread-local storage. Each thread maintains a private map of stock statistics in a structure **StockStats**, processing a disjoint subset of orders. After parallel processing, thread-local results are merged sequentially.

Parallelization Strategy: We allocate t thread-local maps (where t is the number of threads) and use `#pragma omp for schedule(static)` to partition work. Static scheduling provides better cache locality for sequential data access. The merge phase combines statistics using min/max operations for sell/buy values and summation for totals.

2. Performance Analysis

Experimental Setup: Benchmarks performed on: Apple M4, 10 cores with OpenMP. Test cases: 10K, 100K, and 1M orders with varying unique stock IDs.

- **totalAmountTraded():** Achieves 5.61x speedup at 8 threads.
- **printOrderStats():** Achieves 5.11x speedup at 8 threads.
- **updateDisplay():** Achieves 0.99x speedup at 8 threads.



(a) benchmark results

Figure 1: Speedup vs. number of threads for three parallel functions. Tests conducted with varying packet sizes (10K, 100K, 1M orders). Dashed line represents ideal linear speedup.