Performance Analysis and Optimization Report

Latency Benchmarking

Metrics Measured:

1. **Order Placement Latency:**
   - Observed Latency: ~800-1200 ms
   - Occasional Spikes: Up to 1500 ms
2. **Market Data Processing Latency:**
   - Observed Latency: ~800-1200 ms
   - Covers fetching orderbook and positions
3. **WebSocket Message Propagation Delay:**
   - Observed Delay: ~0-1 ms
   - Occasional Spikes: Up to 2 ms
   - Context: Both client and server are running on the same computer
4. **End-to-End Trading Loop Latency (Cancel/Modify Operations):**
   - Observed Latency: Similar to order placement and market data processing, ~800-1200 ms

---

Optimization Requirements and Implementations

1. Memory Management

- **Analysis:** Memory usage was initially ~15,000 KB, with occasional peaks to 15,200 KB.
- **Optimizations:**
  - Retained similar memory usage when switching to simdjson for improved CPU efficiency.
  - Replaced std::vector with std::unordered_set for managing subscriptions. This reduced memory overhead, particularly when the WebSocket client subscribes to multiple symbols.
  - Used auto and const keywords where applicable, reducing unnecessary memory allocations and improving code clarity.

2. Network Communication

- **Analysis:** The primary bottleneck for network communication is latency in order placement and market data processing, heavily influenced by internet speed.
- **Optimizations:**

- Implemented asynchronous sending using websocketpp::lib::asio::post, ensuring non-blocking message propagation and better scalability.
- Optimized WebSocket message handling by leveraging efficient JSON parsing and construction techniques.

3. Data Structure Selection

- **Analysis:** Managing client subscriptions required a scalable and efficient data structure to handle frequent modifications.
- **Optimizations:**
    - Replaced std::vector with std::unordered_set for storing subscribers. This change improved lookup, insertion, and deletion times, particularly as the number of symbols and subscribers increased.

4. Thread Management

- **Analysis:** With increasing subscribers and tasks, thread management was crucial to ensure responsiveness and low latency.
- **Optimizations:**
    - Used a thread pool with 4 threads (matching the number of CPU cores) for concurrent task handling.
    - Moved WebSocket server operations to a background thread using the thread pool to prevent blocking the main thread.
    - Periodic updates (e.g., orderbook updates) were offloaded to the thread pool for asynchronous execution.

5. CPU Optimization

- **Analysis:** The initial CPU usage was measured at 0.023%.
- **Optimizations:**
    - Integrated simdjson for parsing JSON, reducing CPU utilization to 0.01% while maintaining memory efficiency.
    - Improved computational efficiency by optimizing loops and leveraging std::shared_mutex for thread-safe access to shared resources (e.g., m_subscribers).

Benchmarking Methodology

1. **Tools Used:**
    - Self-implemented functions and manual timing for latency measurements.
    - System resource monitors for CPU and memory usage.
2. **Procedure:**

- Benchmarked the server by running client simulations for order placement, market data fetching, and trading loops.
- Measured latency for WebSocket message propagation under varied subscription loads.
- Collected resource usage data during peak operation.

3. **Baseline Metrics:**
   - Established initial benchmarks before applying optimizations for comparison.
4. **Testing Environment:**
   - Single machine setup with client and server running locally to isolate code-level optimizations from network latency.

Before/After Performance Metrics

Metric	Before Optimization	After Optimization
Order Placement Latency	~800-1200 ms, spikes 1500 ms	Similar
Market Data Processing Latency	~800-1200 ms	Similar
WebSocket Propagation Delay	0-1 ms, spikes to 2 ms	Similar
Memory Usage	~15,000 KB, peaks 15,200 KB	Similar, slightly reduced for high subscriptions
CPU Usage	0.023%	0.01%

Note: The insignificant change in latencies is primarily due to the internet speed being the bottleneck. Additionally, sub-optimal decisions made during the initial design phase of the system have contributed to this issue.

Justification for Optimization Choices

1. **simdjson Integration:**
   - Reduced CPU usage significantly during JSON parsing without impacting memory usage.
   - Justified for handling high-frequency WebSocket updates.
2. **std::unordered_set for Subscriptions:**
   - Reduced memory usage for high subscription scenarios.
   - Improved performance of insertion, deletion, and lookup operations.

3. **Thread Pool and Async Sending:**
    - Leveraged multithreading for scalability, reducing blocking operations and improving throughput.
    - Justified by the need to handle concurrent client requests and periodic updates efficiently.
4. **std::shared_mutex:**
    - Ensured thread-safe access to shared data while minimizing locking overhead.
    - Vital for maintaining low latency under concurrent operations.

Discussion of Potential Further Improvements

1. **Improve JSON Serialization:**
    - Explore custom lightweight serialization to reduce both CPU and memory overhead further.
2. **Dynamic Thread Pool Sizing:**
    - Adjust thread pool size dynamically based on workload to optimize resource utilization.
3. **Parallel Market Data Processing:**
    - Process market data for different symbols concurrently, reducing overall latency for fetching large datasets.
4. **WebSocket Compression:**
    - Implement WebSocket compression (e.g., permessage-deflate) to reduce message size and improve throughput.
5. **Advanced Network Techniques:**
    - Introduce connection pooling and keep-alive mechanisms to further optimize network communication.
