# Multithreading in HFT: Building Ultra-Low-Latency Trading Systems

Quant Insider

## Contents

# 1 Executive Summary

High-frequency trading (HFT) systems operate in a domain where nanoseconds determine competitive advantage. Building such systems is a complex interplay between software architecture, hardware-aware programming, and deep understanding of concurrency. This document explores multithreading in the context of HFT — not just the usage of `std::thread`, `std::future`, and `std::promise`, but also the underlying CPU mechanics, memory models, synchronization tradeoffs, and hardware topology that shape performance at the microsecond scale.

This paper aims to provide a comprehensive deep dive into:

- System-level concurrency challenges in ultra-low-latency trading.
- Design principles for multithreaded architectures.
- Best practices for using C++ concurrency primitives.
- Hardware-level insights: cache, NUMA, hyper-threading.
- Performance benchmarks, tuning techniques, and common pitfalls.

# 2 System Design Challenges

Designing multithreaded trading systems introduces unique challenges that go far beyond writing concurrent code:

1. **Determinism vs. Parallelism:** Low latency often demands deterministic execution paths. Uncontrolled concurrency introduces jitter and latency spikes.
2. **Synchronization Overhead:** Locks, atomic operations, and memory fences can add significant latency overhead if used naively.
3. **NUMA and Cache Effects:** Memory locality plays a critical role. Remote memory access can be 3–4x slower than local L1 cache hits.
4. **Thread Scheduling:** OS schedulers are not latency-aware. Without explicit control, context switches and migrations introduce unpredictable delays.

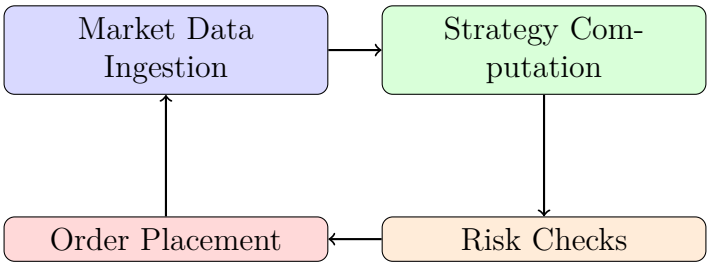


**Figure 1:** Simplified multithreaded HFT pipeline.

Each stage above can be implemented as a separate thread or set of threads. The main challenge is ensuring that data flows between them with minimal synchronization cost.

# 3 Low-Latency Concurrency Principles

In a high-performance trading engine, concurrency is not just about parallelism. It's about eliminating performance killers such as:

- **Lock contention:** Mutexes can cause threads to block and add microseconds of latency.
- **False sharing:** When threads modify independent variables on the same cache line, invalidations slow the system.
- **Context switches:** Even a single switch can cost thousands of cycles.
- **NUMA penalties:** Accessing memory on a remote node can cost 200–300 ns compared to 4 ns for an L1 cache hit.

**Best Practices:**

1. Pin threads to cores to avoid context migration.
2. Use lock-free data structures and atomic primitives where possible.
3. Keep hot data on the same NUMA node as the thread processing it.
4. Align and pad shared data structures to avoid false sharing.

# 4 Implementation Details: C++ Concurrency Primitives

## 4.1 std::thread

The fundamental building block for multithreading in C++ is `std::thread`. It represents a single thread of execution.

```
#include <iostream>
#include <thread>

void process_market_data() {
    std::cout << "Processing␣market␣data..." << std::endl;
}

int main() {
    std::thread t1(process_market_data);
    t1.join(); // Ensures thread completes before exit
    return 0;
}
```

**Best Practices:**

- Use `join()` or `detach()` on all threads to prevent resource leaks.
- Use thread affinity APIs (`pthread_setaffinity_np`) to pin threads.

## 4.2 std::future and std::promise

These abstractions provide a safe and efficient way to pass results between threads asynchronously.

```
#include <future>
#include <iostream>

int compute_signal() {
    return 42; // Simulated signal calculation
}

int main() {
    std::future<int> result = std::async(std::launch::async, compute_signal
        ↪ );
    std::cout << "Signal: " << result.get() << std::endl;
}
```

`std::promise` is useful for explicit communication between threads:

```
#include <promise>
#include <numeric>
#include <vector>
#include <thread>

void accumulate(std::vector<int>::iterator first,
                std::vector<int>::iterator last,
                std::promise<int> prom) {
    prom.set_value(std::accumulate(first, last, 0));
}

int main() {
    std::vector<int> data = {1, 2, 3, 4, 5};
    std::promise<int> prom;
    std::future<int> fut = prom.get_future();
    std::thread t(accumulate, data.begin(), data.end(), std::move(prom));
    std::cout << "Sum: " << fut.get() << std::endl;
    t.join();
}
```

# 5 Hardware-Level Considerations

## 5.1 Cache Hierarchies

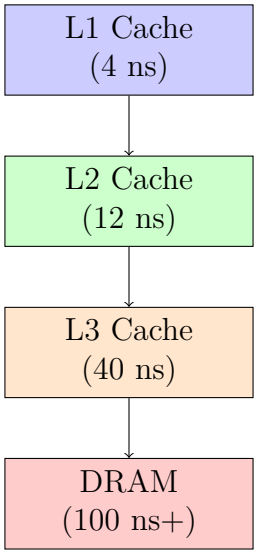


**Figure 2:** Memory access latency hierarchy.

Cache locality is one of the most critical factors in HFT. Ensuring that hot data structures remain in L1 or L2 drastically reduces latency.

## 5.2 NUMA Considerations

On multi-socket systems, each CPU socket has local memory. Accessing memory from another socket is significantly slower.

**Best Practice:** Use `numactl` or `hwloc` to control memory placement and bind threads to specific NUMA nodes.

## 5.3 Hyper-Threading

While hyper-threading increases throughput, it can add unpredictable latency due to shared execution units. For deterministic latency-sensitive threads (e.g., order routing), consider disabling hyper-threading on critical cores.

# 6 Performance Benchmarks & Tradeoffs

## 6.1 Lock Contention Costs

| Technique | Time for 500M increments |
|---|---|
| Single Thread (no lock) | 300 ms |
| Single Thread (volatile) | 4700 ms |
| Single Thread (Atomic) | 5700 ms |
| Single Thread (Mutex) | 10000 ms |
| Two Threads (Atomic) | 30000 ms |
| Two Threads (Mutex) | 224000 ms |

**Insight:** Locks scale poorly under contention. Lock-free designs can improve performance by orders of magnitude.

### 6.2 Thread Pinning vs. OS Scheduling

Pinned threads can reduce jitter by up to 40%. Unpinned threads risk context switches, cache invalidations, and migrations across NUMA domains.

## 7 Deployment & Debugging Notes

**Deployment Recommendations:**

- Isolate latency-critical threads on dedicated cores.
- Use `taskset` or `cset` for CPU affinity in production.
- Disable power-saving states (C-states) that introduce wake-up latency.

**Debugging Tools:**

- `perf`: For profiling CPU usage and context switches.
- `htop` with thread view: Monitor thread scheduling and CPU affinity.
- `numactl --hardware`: Inspect NUMA topology.

## 8 Final Thoughts

Multithreading in HFT is not merely a programming exercise — it is an art that blends low-level system understanding with careful software engineering. The difference between a well-engineered multithreaded architecture and a naive one can be tens of microseconds — a lifetime in trading.

Building ultra-low-latency trading systems requires:

- Deep understanding of CPU architecture and memory hierarchies.
- Conscious tradeoffs between throughput and latency.
- Use of lock-free and cache-friendly data structures.
- Tight control over thread placement, scheduling, and synchronization.

The future of HFT lies at the intersection of software concurrency and hardware-aware design. Mastering multithreading is a crucial step toward achieving deterministic, ultra-fast trading systems.