Project Final Report:
Parallel Union-Find

Group members: Zhaowei Zhang, Eric Zhu

Summary

This project implemented and evaluated sequential, coarse-grained locking, fine-grained locking, and lock-free parallel Union-Find algorithms in C++ using OpenMP to compare their performance and scaling characteristics. Our primary deliverables are performance graphs comparing these implementations across datasets designed to vary in the number of operations, the level of data contention, and the ratio of different operation types (union, find, same-set). We conducted diverse testing and development on 8-core GHC machines, while scalability testing was performed on Pittsburgh Supercomputing Center (PSC) machines using up to 128 cores to analyze how each parallel strategy scales under different conditions and core counts.

Background

- **Algorithm:** The Union-Find (also known as Disjoint Set Union or DSU) data structure maintains a collection of disjoint sets. It efficiently supports two primary operations: union, which merges the sets containing two specified elements, and find, which determines the representative (or root) element of the set containing a given element.

  Standard optimizations are crucial for performance:

  - **Union by Rank/Size:** When merging sets, the root of the set with smaller rank (tree height) or size (number of elements) is made a child of the root of the larger set. This keeps the trees representing sets relatively shallow. In our implementation, we primarily use union by rank.
  - **Path Compression:** During a find operation, after finding the root, all nodes visited on the path from the starting element to the root are made direct children of the root. This dramatically flattens the trees over time, speeding up future find operations.

  We also implemented a sameset operation, which simply checks if two elements belong to the same set, typically by comparing the results of their find operations.

- **Key Data Structures:** The core data structure is a parent array (std::vector⟨int⟩ or std::vector⟨std::atomic⟨int⟩⟩), potentially with a special value or encoding for roots. An auxiliary rank array (std::vector⟨int⟩) stores ranks for the union-by-rank optimization. The lock-free version uses a single atomic array (A) encoding both parent and rank. Locking versions use standard vectors protected by std::recursive_mutex (coarse) or std::vector⟨std::mutex⟩ (fine).

- **Key Operations on Data structures:**

  - find(i): Read elements from the parent array, traversing the path from i to its root. Standard path compression also writes new parent pointers back to the parent array along the path.

    In our sequential and coarse-grained versions, this read-traverse-write process is fully protected by locks. In the lock-free version, path compression updates are attempted using atomic compare-and-swap operations within the traversal loop.

  - unionSets(i, j): Call find for i and j (reads). Reads ranks of the roots from the rank array. If roots differ, it writes to the parent array (to link one root to the other) and potentially writes to the rank array. These writes are protected appropriately in each version (global lock, fine-grained root locks, or atomic CAS).

  - sameSet(i, j): Primarily involves reads via the find operation.

  - processOperations(ops, results): Iterates through a list of Operation structs, calling the appropriate methods (unionSets, find, sameSet). When run in parallel, this function causes concurrent reads and writes to the shared parent and rank data structures. It writes results into the results vector.

- **Workload Breakdown & Parallelism Analysis:**

  - Dependencies: High potential for dependencies through modifications of the shared parent/rank structures by union and find (with path compression). Concurrent operations require synchronization (locks or atomics). The fine-grained version has slightly relaxed consistency during its find's path compression phase due to the lock-free writes.

- Amount/Type of Parallelism: Data parallelism over the list of operations in processOperations, distributed via #pragma omp parallel for
- Locality: Generally poor spatial locality due to potential pointer-chasing in find. Temporal locality can exist depending on access patterns and is aided by path compression.

Approach

- **Technologies Used:**
  - Language/APIs: The implementations were developed in C++20. Parallelism was achieved using OpenMP. For synchronization in the locking versions, we used std::mutex and std::recursive_mutex. The lock-free version relied heavily on std::atomic and associated compare-and-swap (CAS) operations (like compare_exchange_weak).
  - Build System: We used make with a Makefile and the g++ compiler.
  - Target Machines & Platforms: Development and testing were performed on Linux platforms. Specifically, we used 8-core GHC machines for general testing and 128-core PSC (Pittsburgh Supercomputing Center) machines for scalability analysis.
- **Mapping Problem to Parallel Hardware:**
  - Core Strategy: The parallelization targets the processOperations function, which handles a batch execution of Union-Find requests (UNION, FIND, SAMESET).
  - Work Distribution: We used OpenMP's work-sharing capabilities, specifically #pragma omp parallel for (with schedule(static) based on our code), to distribute the iterations of the loop over the input std::vector⟨Operation⟩ among the available OpenMP threads. Each thread executes a portion of the total operations.
  - Synchronization Mapping: The crucial difference between the approaches lies in how concurrent access to these shared structures by multiple threads is managed.

    In coarse-grained locking implementation, a single global std::recursive_mutex acts as a critical section around all accesses to the shared parent and rank arrays within the find, unionSets, and sameSet methods. This means only one thread can be actively

performing a Union-Find operation at any given moment, effectively serializing the core logic but parallelizing the loop overhead.

In fine-grained locking implementation, a std::vector⟨std::mutex⟩ provides a lock potentially associated with each element (though primarily used for roots). The unionSets operation locks only the mutexes corresponding to the roots of the sets being potentially merged (using ordered locking by index to prevent deadlocks). This allows multiple threads to operate concurrently if they are working on disjoint sets (sets with different roots). The find operation, however, performed its path compression step using potentially racy, non-locked writes.

In lock-free implementation, it avoids locks entirely. Concurrent access is managed using std::atomic operations on the combined parent/rank array (A). find uses atomic loads and CAS attempts for path compression. unionSets uses atomic loads and CAS within retry loops to safely find roots, compare ranks, and link trees even if other threads are simultaneously modifying related elements. Concurrency is maximized, with contention resolved via atomic hardware instructions and retries. Here are pseudocodes for the three main operations:

```
1  func FindInternal(u): (root_idx, root_val) {
2      p_val = AtomicLoad(A[u])
3      if IsRoot(p_val): return (u, p_val)
4      p_idx = p_val // p_val must be parent index here
5      (root_idx, root_val) = FindInternal(p_idx)
6      if p_idx != root_idx:
7          CAS(&A[u], p_val, root_idx)
8      return (root_idx, root_val)
9  }
10
11 func Union(a, b): boolean {
12     while (true) {
13         (root_a, _) = FindInternal(a)
14         (root_b, _) = FindInternal(b)
15         val_a = AtomicLoad(A[root_a])
16         val_b = AtomicLoad(A[root_b])
17         if !IsRoot(val_a) || !IsRoot(val_b): continue
18         if root_a == root_b: return false
19         rank_a = GetRank(val_a)
20         rank_b = GetRank(val_b)
21         child_root = root_a; child_val = val_a
22         parent_root = root_b; parent_val = val_b
23         parent_rank = rank_b // Rank of the potential final parent
24         if rank_a > rank_b || (rank_a == rank_b && root_a > root_b) {
25              child_root = root_b; child_val = val_b
26              parent_root = root_a; parent_val = val_a
```

```
27                parent_rank = rank_a // Rank of the potential final parent
28            }
29            if CAS(&A[child_root], child_val, parent_root) {
30                if rank_a == rank_b:
31                    new_parent_val = MakeRootVal(parent_rank + 1)
32                    // Best-effort rank update: OK if this CAS fails
33                    CAS(&A[parent_root], parent_val, new_parent_val)
34                return true
35            }
36        }
37    }
38
39    func SameSet(a, b): boolean {
40        while (true) {
41            (root_a, _) = FindInternal(a)
42            (root_b, _) = FindInternal(b)
43            if root_a == root_b: return true
44            val_at_root_a = AtomicLoad(A[root_a])
45            if IsRoot(val_at_root_a):
46                return false
47            continue
48        }
49    }
```

- **Optimization Process & Iterations:**

  The fundamental algorithms (find path logic, union by rank) were conceptually preserved. The main structural change was wrapping the execution of individual operations within the parallel processOperations framework. The core design for coarse-grained and fine-grained locking remains the same throughout the whole process. Changes of these two implementations are usually to align with the interface change of lock-free implementation.

  For the lock-free approach, our initial attempt (Version 1) utilized a 128-bit std::atomic⟨Node⟩ struct to store parent and rank pairs atomically, relying on the lock cmpxchg16b instruction. However, performance testing showed this version performed poorly, running even slower than the single-threaded baseline. After running profiling tools, we found out that instruction per cycle for this implementation is very low, indicating the possibility of severe contention. Therefore, we believe the issue is due to either the high intrinsic cost and low throughput of 128-bit atomic CAS operations, or the lack of true lock-free hardware support causing std::atomic⟨Node⟩ to use internal mutexes. This critical finding led us to redesign the lock-free implementation. This improved version uses std::atomic⟨int⟩ and encodes both parent pointer and rank information into the single integer[2]. This leverages highly efficient, native

word-size atomic instructions. While requiring helper functions for encoding/decoding, this approach proved successful, demonstrating good speedup and scalability in our tests on GHC and PSC machines by effectively utilizing hardware atomic primitives without the overhead or fallback issues of the 128-bit version. Throughout development, we also iteratively refined the interfaces across all versions for consistency.

Seeking further improvements based on lock-free literature, we then experimented with two additional optimizations built upon this successful atomic⟨int⟩ baseline:

- Direct store in find: We implemented a variant where the compare_exchange_weak (CAS) used for path compression within the find_internal function was replaced with a non-conditional, relaxed atomic store (A[u].store(root_idx, std::memory_order_relaxed)). We implement this relaxed optimization because we observe that in our large random dataset settings, contention is not that frequent. The error rate of CAS operation in certain conditions (when we have a large vertex set) can below 1%. And applying this optimization does not break our correctness test on final connectivity. Dan et al. note that path-compaction schemes—compression, halving, and splitting—are purely performance heuristics; they leave the algorithm's correctness intact [1] [3]. Consequently, the only read that must be fenced with a memory barrier is the one that tests whether a given node is currently a root.
- Immediate Parent Check (IPC): We implemented another variant that added a fast-path heuristic to the beginning of unionSets and sameSet. This check performs relaxed atomic loads of the immediate parents of the query elements (a and b). If the parents are identical indices (and not roots), the operation could potentially conclude early (returning false for union, true for sameSet under the assumption they likely belong to the same set) without invoking the full recursive find_internal traversals. We believe that it will work if the path is almost fully compressed after lots of Find invocations. So, similar to the previous optimization, this one is also believed to work better in certain situation.

- **Testing** & **Benchmarking**

A rigorous testing and benchmarking framework was essential for validating our parallel

Union-Find implementations and evaluating their performance characteristics. This framework encompassed dataset generation, correctness testing, and performance benchmarking.

Dataset Generation:

We developed a Python script to programmatically generate input workloads, allowing precise control over various parameters critical to Union-Find performance. The script creates text files containing a sequence of operations (UNION=0, FIND=1, SAMESET=2), preceded by a header specifying the number of elements (n_elements) and the number of operations (n_operations). Key controllable parameters included:

- Size: n_elements and n_operations
- Operation Mix: The target ratios for FIND operations (–find-ratio) and SAMESET operations (–sameset-ratio, specified as a fraction of the non-FIND operations)
- Contention Model: The script supports three distinct modes to simulate varying levels and patterns of concurrent access:
  1. Uniform: Elements for operations are chosen uniformly at random (using –contention-level 0), representing workloads with low contention
  2. Focused: A specific element (–hot-element) is targeted with a probability determined by –contention-level (0.0 for low focus, approaching 1.0 for high focus), while other elements are chosen uniformly. This models scenarios with access hotspots
  3. Extreme: Activated by –extreme-contention, this mode forces all operations to involve only elements 0 and 1, maximizing contention on this minimalsubset
- Reproducibility: A –seed argument allows for generating identical datasets across runs.

This generator enabled us to create a wide range of input scenarios to thoroughly test and benchmark the different implementations.

Correctness Testing:

We used a C++ test harness (test_parallel_correctness.cpp) to verify that parallel implementations produce results equivalent to the sequential baseline. The key point we are checking is the final connectivity. So, we verify the correctness by comparing the final state of the

disjoint sets. The program will call the find method on every element for both the serial and parallel structures. It then iterates through all unique pairs of elements (a, b), asserting that serial.find(a) == serial.find(b) if and only if parallel.find(a) == parallel.find(b). This ensures that, despite concurrency, the parallel algorithms maintain the same final equivalence relations between elements as the deterministic serial version. We acknowledge that this test focuses on final state consistency and does not compare the results of intermediate FIND or SAMESET operations returned during the concurrent execution.

Performance Benchmarking:

A dedicated C++ benchmarking program (benchmark.cpp) was used to measure execution time and analyze scalability. It allows selection of the implementation (serial, coarse, fine, lock-free, lock-free_plain, lock-free_ipc), the input operations file, the number of repetitions (num_runs), and the number of OpenMP threads. A warm-up run is performed to mitigate initialization costs and cache effects. Multiple timed runs are then executed. For each run, a fresh instance of the selected Union-Find structure is created and initialized before timing the operation. Moreover, the program records the duration of each run and calculates the minimum, maximum, average time, and standard deviation, providing insights into performance variability. Throughput (operations/second) and speedup (serial time / parallel time) are derived from these timings. It also provided the perf command for cache and ipc statistics. This comprehensive testing and benchmarking strategy allowed us to confidently verify the correctness of our parallel implementations and thoroughly analyze their performance trade-offs under various conditions relevant to Union-Find workloads.

Results

This section details the performance evaluation of the implemented sequential and parallel Union-Find algorithms. We analyze their execution time, throughput, and scalability under various workloads and thread counts, comparing the effectiveness of the different parallelization strategies (coarse-grained locking, fine-grained locking, and lock-free variants).

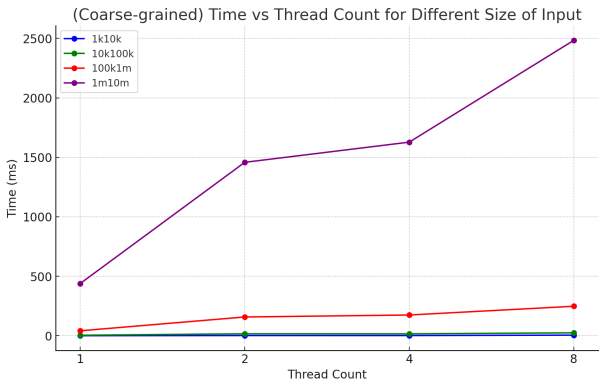
Our primary goal was to achieve significant speedup over the optimized sequential baseline using parallel execution, particularly with the fine-grained and lock-free approaches, and to understand how different factors like contention and operation mix affect performance.

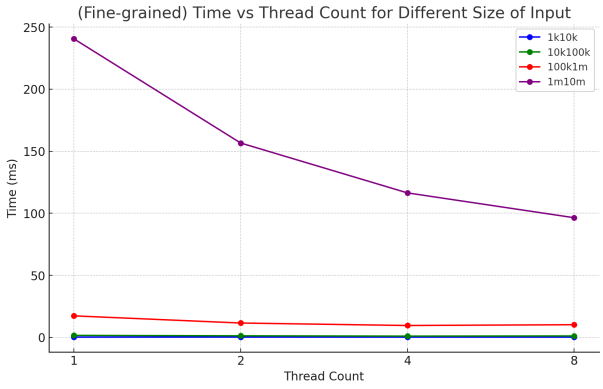
Experimental Setup:

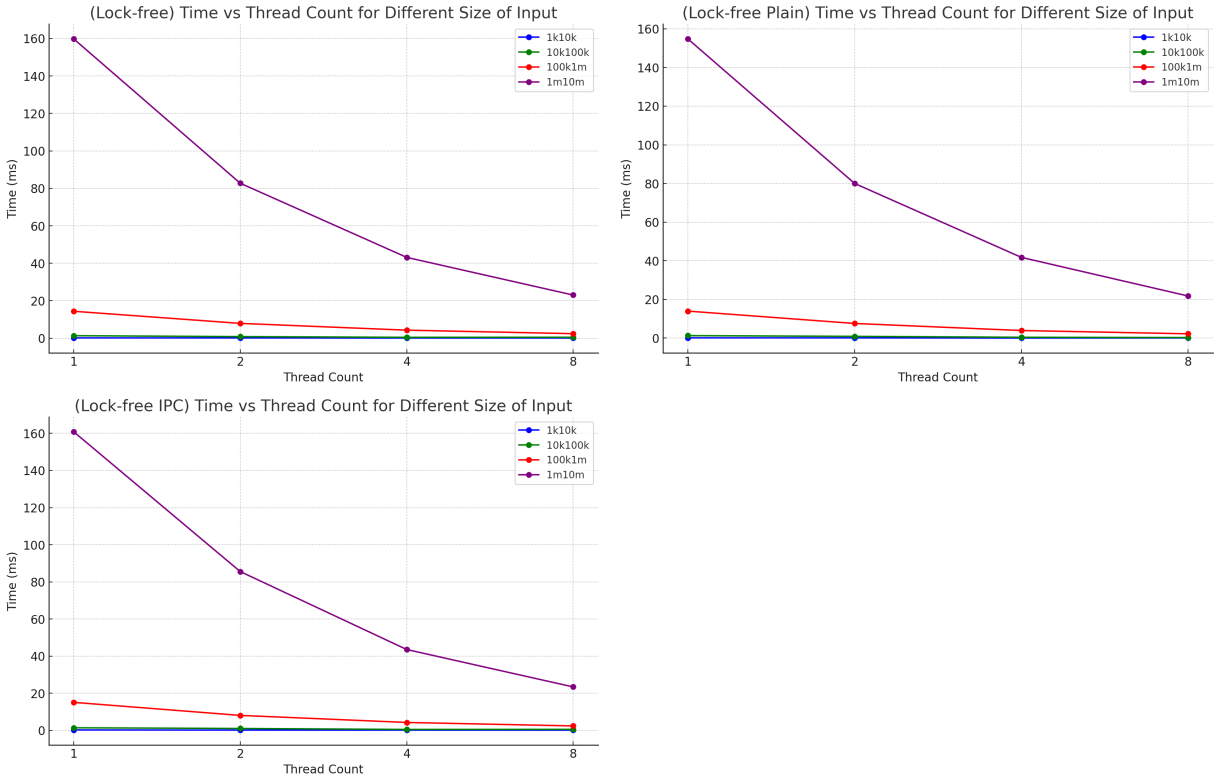
- Input Datasets: Workloads were generated using our custom Python script. We systematically varied[4]:
  - Number of elements: 1k, 10k, 100k, 1M.
  - Number of operations: 10k, 100k, 1M, 10M
  - Operation Mix: Controlled via –find-ratio and –sameset-ratio.
  - Contention: Use all three modes we specified earlier. We found out that this does not change the performance of lock-free implementation too much. So, we will not show graphs on this dimension.
- Benchmarking Procedure: The benchmark.cpp program performed multiple timed runs (10 in our testing) for each configuration, creating a fresh data structure instance per run and measuring the processOperations time. A warm-up run preceded timed measurements. Average, min, max, and standard deviation were calculated.
- Baseline: In general, the baseline in our project is the sequential implementation of union find. However, in some situations, like scalability settings, the baseline becomes the same instance with only one thread.

Performance Results & Analysis:

Time vs Number of threads:







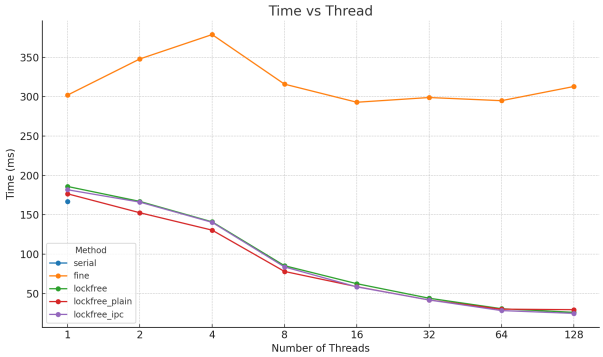
Analysis:

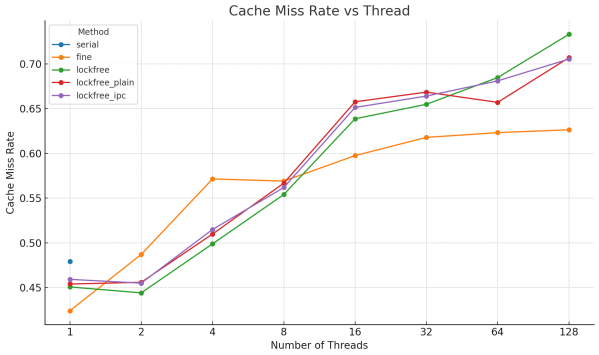
The performance evaluation clearly demonstrates the limitations of traditional locking strategies for parallelizing Union-Find operations. The coarse-grained implementation exhibited poor performance, with execution time increasing significantly as threads were added, especially for larger datasets. This indicates severe contention on the single global lock, rendering parallel execution ineffective. The fine-grained approach offered improvement over coarse-grained, showing some initial speedup (particularly from 1 to 2 threads), but its scalability quickly diminished. This suggests that while allowing more concurrency, contention on root locks and the overhead of acquiring/releasing multiple locks still impose significant bottlenecks, preventing effective scaling to higher thread counts like 8.

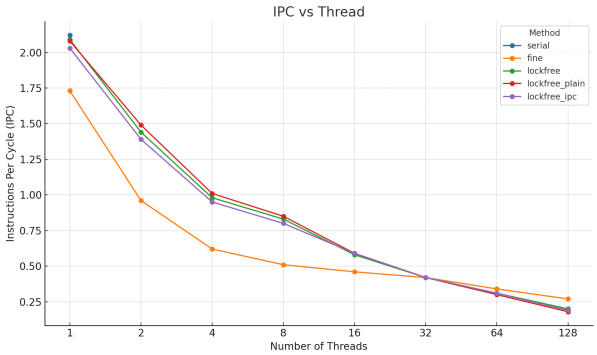
In contrast, the lock-free implementations consistently outperformed the locking strategies across all tested input sizes and thread counts up to 8. The baseline lock-free version, using std::atomic⟨int⟩ and CAS-based path compression, showed good scalability, with execution time decreasing steadily as threads increased. This highlights the benefit of avoiding blocking synchronization and leveraging efficient hardware atomic primitives. Interestingly, the further optimizations attempted – replacing path compression CAS with plain atomic writes (Lock-free Plain) and adding the Immediate Parent

Check heuristic (Lock-free IPC) – resulted in performance nearly identical to the baseline lock-free version in these tests with slight increase. This suggests that for these workloads and up to 8 threads, the potential overhead reduced by these specific micro-optimizations was negligible compared to other factors like the fundamental cost of atomic operations, cache coherence effects, or potential retries within the core unionSets logic. Overall, the results strongly favor the lock-free approach using native atomics as the most effective parallelization strategy for this problem on multi-core CPUs.

Scalability:







Analysis:

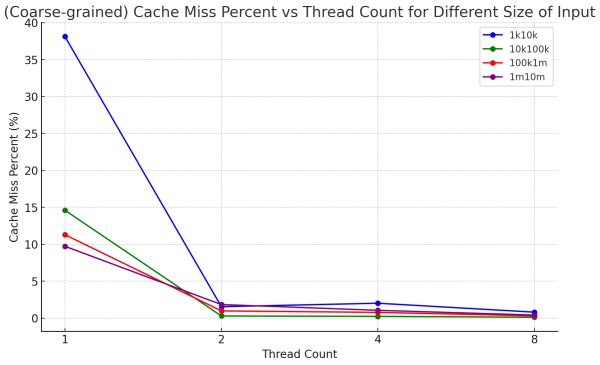
The coarse-grained locking method exhibits very poor scalability. As more threads are introduced, contention for a single global lock severely limits performance gains and degrades efficiency at higher thread counts. Thus, we did not show its stats to avoid huge difference in y-axis, which could diminish the differences between the rest approaches.

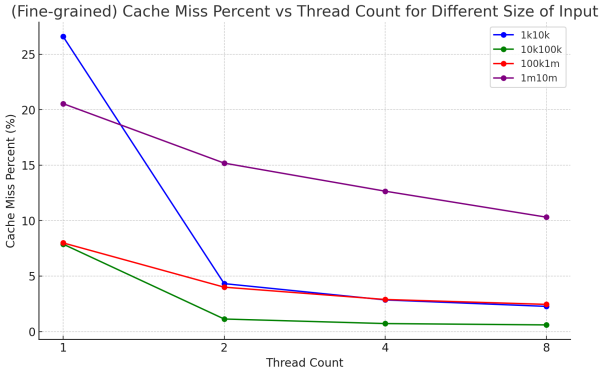
Fine-grained locking does not scale for this problem beyond a very small number of threads due to lock contention and overhead, confirmed by high execution times, high cache miss rates, and low IPC.

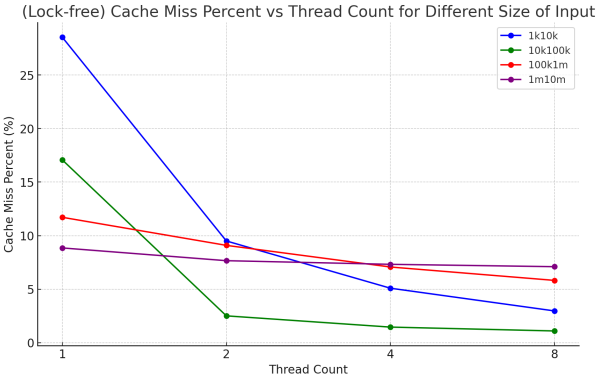
Lock-free implementations scale reasonably well up to 16-32 threads but show diminishing returns

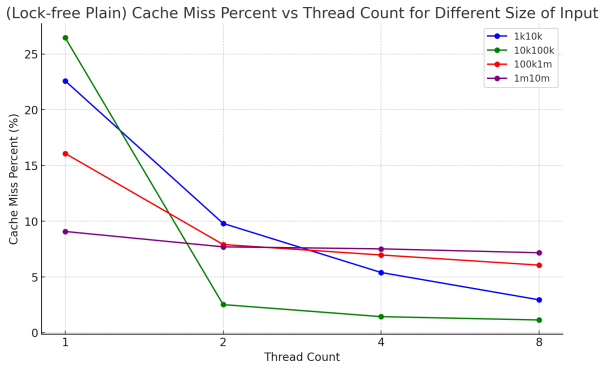
at higher core counts (64-128). The primary limiters appear to be memory system bottlenecks. The increasing cache miss rate directly points to the cost of maintaining coherence for the shared A array under frequent atomic updates from many cores. Increased cache misses lead to pipeline stalls while waiting for data, reflected in the decreasing IPC. While non-blocking, high contention on specific elements (roots) likely leads to increased CAS failures and retries in unionSets causing more cycles.

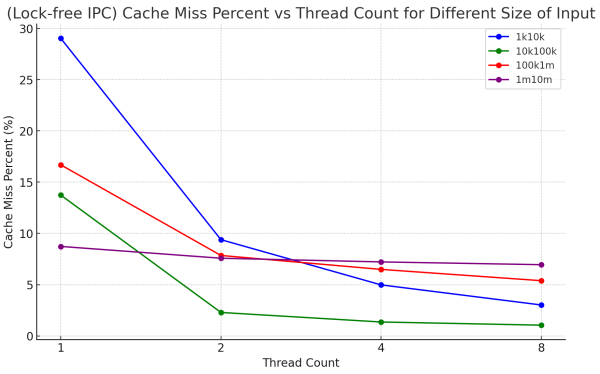
Cache Miss Rate vs Number of Thread:











Analysis:

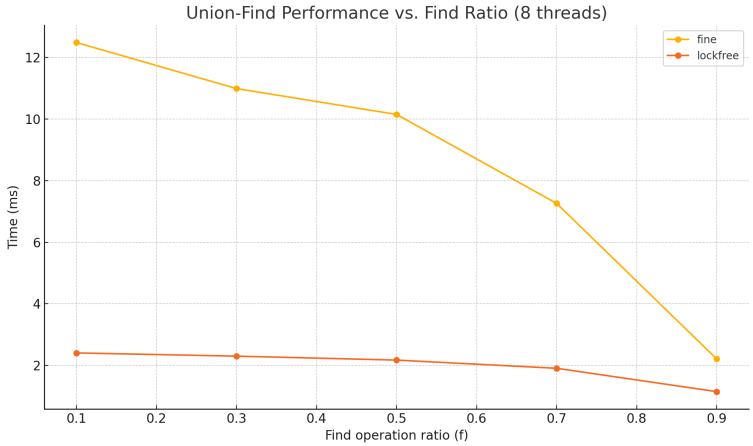
We shall analyze combined with the cache miss rate in the previous section.

The cache miss rate plots reveal distinct behaviors between the locking and lock-free implementa-

tions. All methods exhibit a notably high cache miss percentage when run with a single thread, likely dominated by initial compulsory or capacity misses for the given dataset sizes. This rate drops significantly when moving from 1 to 2 threads across all implementations. For the coarse-grained and fine-grained locking versions, the cache miss rate remains relatively low and stable between 2 and 8 threads. This might seem counter-intuitive given their poor time performance, but it likely reflects the serialized nature of their execution; the coarse lock allows only one thread active access, reducing concurrent cache invalidations, while the fine-grained lock contention might still limit the number of truly simultaneous updates to shared cache lines.

Conversely, the three lock-free variants (baseline, plain-write, IPC), while showing the initial drop from 1 to 2 threads, tend to exhibit a stabilizing or even slightly increasing cache miss rate as the thread count grows from 2 to 8, especially for larger problem sizes. This pattern suggests that as more threads concurrently access and modify the shared atomic array A using non-blocking operations (like CAS), the overhead of maintaining cache coherence across cores becomes more pronounced. Increased concurrent writes lead to more cache line invalidations and transfers between cores, resulting in a higher percentage of memory accesses missing the local cache. The near-identical cache behavior across the three lock-free variants further indicates that this cache coherence traffic, inherent to frequent atomic updates on shared data, is a more significant factor than the specific micro-optimizations related to path compression or immediate parent checks in this regime.

Time vs. Find ratio:



Analysis:

This graph illustrates the impact of operation mix on the performance of the fine-grained and

lock-free implementations when executed with 8 threads. The x-axis represents the ratio of FIND operations in the workload, meaning a lower ratio implies a higher proportion of UNION (we set sameset ratio to 0). The lock-free implementation consistently outperforms the fine-grained version across the entire spectrum of find ratios, maintaining significantly lower execution times.

Both implementations show improved performance (lower execution time) as the ratio of FIND operations increases. However, the effect is far more dramatic for the fine-grained locking version. Its execution time drops sharply from over 12ms at a 0.1 FIND ratio to around 2ms at a 0.9 FIND ratio. This indicates that the fine-grained implementation is particularly sensitive to the frequency of modifying operations. High rates of UNIONs likely lead to frequent contention for root locks, causing significant synchronization overhead and limiting performance. As FIND operations become dominant, lock contention decreases, allowing for better performance.

The lock-free implementation also benefits from a higher FIND ratio, but its performance curve is much flatter, decreasing moderately from roughly 2.4ms to 1.2ms. While UNION operations in the lock-free code are still more complex than FINDs (involving CAS loops and potential retries), the absence of blocking locks mitigates the performance degradation seen in the fine-grained version under high UNION loads. The consistent speed advantage of the lock-free approach across all operation mixes underscores its effectiveness in handling concurrent access compared to fine-grained locking, though both benefit when the workload shifts towards less contentious read-mostly operations.

Analysis of Speedup Limiters:

Based on the data presented in the graphs, we can analyze the factors limiting speedup for each parallel implementation:

- Coarse-Grained Locking: The core limitation factors are synchronization overhead and lack of parallelism. From the graph in Time vs Number of threads section, it clearly shows negative scaling (increasing time with more threads) for larger inputs. This is the classic symptom of a single global lock bottleneck. Only one thread can execute within the critical section (the core Union-Find logic) at a time, serializing execution. While the cache miss rate (from the cache miss rate section) appears low after the initial drop, this is in fact misleading. It's low because threads are mostly idle waiting for the lock, not because memory access is efficient

during parallel execution. The lack of parallelism imposed by the lock is the fundamental barrier.

- Fine-Grained Locking: The primary limiters are synchronization overhead and lock contention. Although performing better than coarse-grained initially, the fine-grained version exhibits poor scalability in our scalability section, with performance saturating or degrading quickly beyond 2-4 threads. We also notice that its cache miss rate increasing significantly with threads, ending highest among all methods. This points to high costs associated with acquiring/releasing multiple locks and, more importantly, contention when multiple threads attempt to lock the same root node simultaneously (especially likely with high UNION ratios or focused workloads). This contention leads to threads stalling (low IPC) and significant cache coherence traffic as locked cache lines are invalidated and transferred between cores.

- Lock-Free Implementations (Baseline, Plain Write, IPC): The primary limiters are communication overhead (cache coherence) and atomic operation costs. These versions consistently show the best performance and scalability. However, their speedup is not perfectly linear and flattens significantly beyond 16-32 threads according to our graph in scalability section. The key indicators are the steadily increasing cache miss rate and decreasing IPC as thread count scales. This pattern strongly suggests that the limitation is not blocking synchronization but rather the overhead inherent in the non-blocking approach at scale. Frequent atomic operations (especially Compare-and-Swap in unionSets) on the shared data array (A) generate substantial cache coherence traffic across cores. As more cores participate, the probability increases that a core needs data recently modified by another, leading to cache misses and pipeline stalls (lower IPC) while waiting for data transfer via the coherence protocol. While atomics avoid blocking, the communication overhead associated with maintaining memory consistency and the intrinsic cost of atomic operations become the bottleneck at high core counts. The near-identical performance of the three lock-free variants further suggests these fundamental memory system and atomic costs are the dominant limiters, rather than the specific micro-optimizations attempted.

Deep Analysis

To understand the bottlenecks in the baseline lock-free implementation at higher core counts, we

ran Intel VTune Profiler (hotspots and threading analysis) on the PSC machine. We tested with 1 million elements and 10 million operations (30% FIND, 50% SAMESET of remaining) at both 32 and 128 threads.

At 32 threads: The profile showed a mix of bottlenecks. OpenMP barrier waits (due to load imbalance) were significant (12-14% of CPU time), but substantial time was also spent within the core lock-free algorithm functions (processOperations, unionSets, atomic loads). Performance was limited by both load imbalance and the inherent costs of the concurrent algorithm.

At 128 threads (Static Scheduling): The profile changed dramatically. OpenMP barrier functions became the dominant hotspots (28% combined), while the application code's relative contribution decreased. Effective CPU utilization was very low (11-14%), with high spin times (27-33%). This indicates severe load imbalance, where threads finished work quickly and spent most of their time spinning at the loop barrier.

At 128 threads (Dynamic Scheduling, Chunk 1000): Switching to dynamic scheduling reduced the barrier wait time (14% from 28%) and increased the relative time spent in application code. Spin time decreased (19% from 30%), showing improved load balance. However, significant spin time remained, indicating that even with dynamic scheduling, keeping 128 cores fully utilized was challenging for this workload size.

Nonetheless, while VTune profiling indicated that switching from static to dynamic scheduling successfully reduced the time threads spent spinning at OpenMP barriers (lowering the percentage of CPU time in barrier wait), we observed that this did not translate into a significant improvement in the overall benchmark execution time (wall-clock time). This suggests a performance trade-off or the exposure of underlying bottlenecks. The reduction in barrier wait time confirms that dynamic scheduling achieved better load balancing compared to static. However, the lack of corresponding wall-clock improvement implies that either the overhead introduced by the dynamic scheduling mechanism itself offset the gains from reduced waiting, or, more likely, that mitigating the load imbalance bottleneck simply allowed the next major limiter to dominate. Based on the VTune hotspots (significant time still spent in atomic loads and unionSets) and the scalability/cache miss rate graphs (showing performance flattening and rising cache misses at high thread counts), this next limiter is still memory system contention.

Therefore, in order to further improve our program, we need to focus on both work load balancing

and memory contention. Simply trying to solve one of them did not yield expected result as we did.

Future Work

Given all these benchmarks we conducted and analysis we carried out, our future work will be focusing on finding a good work balancing strategy and coping with the cache coherence issue through padding.

Conclusion

This project successfully implemented and evaluated several parallelization strategies for the Union-Find data structure using C++ and OpenMP, including coarse-grained locking, fine-grained locking, and multiple lock-free variants. Our goal was to understand the performance characteristics and scalability limitations of these different approaches on modern multi-core processors.

Our experimental results clearly demonstrated the superiority of the lock-free approach over traditional locking methods for this problem. The coarse-grained implementation suffered from severe serialization due to its single global lock, exhibiting negative scaling. The fine-grained version offered some improvement but was ultimately limited by lock contention and synchronization overhead, scaling poorly beyond a few threads, particularly under workloads with frequent modifying operations (unions).

The lock-free implementation, specifically the baseline version with parent/rank encoding, provided significantly better performance and initial scalability. However, its speedup was not linear and exhibited diminishing returns at higher core counts. Analysis using performance graphs and VTune profiling revealed that its scalability is primarily limited by memory system effects (cache coherence traffic resulting from frequent atomic operations) and the intrinsic cost of atomics up to moderate core counts (16-32). At very high core counts (128), the bottleneck shifted towards load imbalance within the OpenMP parallel loop, leading to substantial time spent spin-waiting at barriers and low effective CPU utilization, even when using dynamic scheduling. Further micro-optimizations explored for the lock-free version (plain write path compaction, immediate parent check) yielded negligible performance benefits in our tests, suggesting the aforementioned factors were more dominant.

Overall, lock-free techniques provide the most promising path for parallelizing Union-Find on multi-core CPUs, effectively mitigating lock contention. However, achieving efficient scaling to very high

core counts requires careful consideration of memory system interactions and ensuring sufficient, well-balanced workloads to overcome parallel runtime overheads. The irregular memory access patterns inherent in Union-Find also affirm that CPU-based parallelism, rather than GPU acceleration, remains the more suitable target for this algorithm.

Work Distribution

- Sequential union find (Zhaowei Zhang)
- Coarse-grained union find (Zhaowei Zhang)
- Fine-grained union find (Eric Zhu)
- Lock Free union find (Zhaowei Zhang)
- Lock Free Optimization (Zhaowei Zhang)
- Benchmarking (Eric Zhu, Zhaowei Zhang)
- Data collection and graph generation (Eric Zhu)
- Poster (Eriz Zhu, Zhaowei Zhang)
- Report (Zhaowei Zhang, Eric Zhu)

Contribution split: 60 % Zhaowei, 40 % Eric. The distribution of credit should be the same as the contribution distribution.

References

[1] Dan Alistarh, Alexander Fedorov, and Nikita Koval. In search of the fastest concurrent union-find algorithm, 2019.

[2] Richard J. Anderson and Heather Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, STOC '91, page 370–380, New York, NY, USA, 1991. Association for Computing Machinery.

[3] Alexander Fedorov, Diba Hashemi, Giorgi Nadiradze, and Dan Alistarh. Provably-efficient and internally-deterministic parallel union-find. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '23, page 261–271. ACM, June 2023.

[4] Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Work-efficient parallel union-find, 2018-02-25.