Project Report

for

CS-3006: PARALLEL & DISTRIBUTED COMPUTING

Parallel SSSP

Ali Hamza Azam, I222126
Zain Bin Asad, I221240
May 6, 2025

Abstract

This project implements a dynamic Single-Source Shortest Path solver that supports four execution modes—serial, OpenMP, MPI, and hybrid MPI+OpenMP—on weighted, undirected graphs that undergo batches of edge-weight updates. We describe the algorithmic design, data structures, parallelization strategies, and communication patterns for each mode, then present a comparative performance study on shared- and distributed-memory platforms.

Introduction

This report details the design and implementation of a multi-paradigm solution for the Single-Source Shortest Path (SSSP) problem in dynamic networks, which combines various parallel processing techniques to achieve optimal performance across different computational scenarios.

The project comprises four primary implementations:

Sequential Implementation:

The first implementation provides a baseline approach using Dijkstra's algorithm, which recomputes shortest paths from scratch after each network change. This serves as a reference point for evaluating the efficiency of parallel solutions and demonstrates the fundamental algorithmic challenges in dynamic SSSP computation.

OpenMP-Based Parallel Implementation:

The second implementation leverages shared-memory parallelism through OpenMP to accelerate SSSP computation. It employs asynchronous updating strategies that minimize synchronization overhead while maintaining correctness, enabling significant performance improvements for moderately sized networks through efficient thread-level parallelism.

MPI-Based Distributed Implementation:

The third module implements a distributed algorithm using MPI to compute shortest paths across process boundaries. This approach addresses scalability by partitioning the graph and distributing workload across multiple processes, though it faces challenges related to communication overhead and data dependencies inherent in graph algorithms.

Hybrid (MPI+OpenMP) Implementation:

The final implementation combines distributed and shared-memory parallelism to capitalize on multi-level hardware parallelism. This hybrid approach attempts to balance the strengths of both paradigms, though its effectiveness depends heavily on problem characteristics and hardware architecture.

Background and Problem Definition

Given a weighted, undirected graph $G=(V,E,w)$ and a source s, dynamic SSSP maintains $dist[v]$ for all $v \in V$ as batches of edge-weight changes $\Delta w(u,v)$ arrive. The goal is to update $dist[]$ in $o((|V|+|E|)\log|V|)$ per change, ideally proportional to the size of the "affected" subgraph E' and V'.

Applications include:

- Traffic routing under time-varying conditions
- Real-time network monitoring
- Dynamic resource allocation

Graph Representation and Utilities

All four implementations share:

- CSR format in graph.cpp & graph.hpp (arrays xadj[], adjncy[], adjwgt[])
- Distance array dist[] and parent array parent[]
- Affected flags per vertex in a std::vector<bool>
- Utilities for loading graphs and updates in utils.cpp & utils.hpp

The driver (src/main.cpp) parses arguments to select mode, reads the initial graph (e.g. data/*.edges), reads update streams (data/*_updates.edges), then invokes one of:

- serial::run_sssp()
- parallel::run_sssp_openmp()
- mpi::run_sssp_mpi()
- mpi::run_sssp_hybrid()

Sequential Implementation

Defines a baseline for correctness and performance.

• Utilizes the standard Dijkstra algorithm to compute all shortest paths from the source “from scratch” on the initial graph.

• Implements an incremental “SingleChange” routine that ensures localized repairs are triggered only for minor edge-weight updates, rather than a complete recompute.

• Demonstrates the fundamental work and associated costs of managing dynamic updates without any parallel processing capabilities.

Design and Implementation

Dijkstra: Uses a binary heap (std::priority_queue) in serial/dijkstra.cpp.

SingleChange: In serial/update.cpp, processes each Δw(u,v):

1. Update CSR weight in place.
2. If increase/deletion: enqueue u,v as invalidated, BFS to reset and re-relax downstream vertices.
3. If decrease/insert: check direct relaxation, push into min-heap, run Dijkstra on subgraph until no improvement.

Complexity: O(|E′| + |V′| log |V′|) per change in best case; O((|V|+|E|)log|V|) worst case.

OpenMP Shared-Memory Implementation

Accelerates the incremental update process on multicore, shared-memory machines.

• Batches change events and utilizes OpenMP loops or tasks to concurrently (and asynchronously) mark affected vertices, invalidate outdated paths, and relax edges.

• The objective is to overlap work across threads and minimize barrier/ synchronization overhead.

• Demonstrates how fine-grained parallelism (vertex/edge-level) can enhance the performance of dynamic SSSP when all data resides within a single address space.

Design and Implementation

Batching & Marking: In parallel/apply_changes.cpp, atomic writes mark endpoints of weight-increase changes in affected[].

Parallel Invalidation & Relaxation: In parallel/relax.cpp:

- Iterative sweep:

```
#pragma omp parallel for reduction(|:local_changed)
for (int i = 0; i < n; ++i) {
  if (affected[i]) { … relax neighbors … }
}
```

- Loop until no thread reports a change.

Asynchronous Tasks: In parallel/tasks.cpp, spawns #pragma omp task per affected vertex, with depend(in:...)/depend(out:...) to overlap work and reduce barriers.

Tuning:

Static vs. dynamic scheduling

First-touch NUMA placement in main.cpp

Thread pinning via OMP_PLACES and OMP_PROC_BIND

MPI Distributed Implementation

Scaling SSSP to Multi-Node Clusters via Graph Partitioning

The SSSP algorithm is scaled to multi-node clusters by partitioning the graph across processes. Each rank holds a subgraph (obtained through METIS partitioning), locally applies changes, and participates in global rounds of relaxation using MPI collectives or point-to-point messages.

This approach enables the algorithm to handle very large graphs that exceed the memory capacity of a single node. It also allows for the measurement of the trade-off between increased communication and distributed compute.

Furthermore, the algorithm explores both synchronous (blocking Allreduce) and asynchronous (nonblocking Iallreduce) variants to minimize idle time.

Design and Implementation

Partitioning: On rank 0, call METIS to partition CSR (mpi/partition.cpp). Scatter subgraphs with MPI_Scatterv.

Local Structures: Each rank builds local_to_global[], boundary edge lists, and dist_local[].

Dynamic Updates: In mpi/updates.cpp:

1. Apply local changes; forward cross-partition updates via MPI_Send.
2. Exchange invalidation flags for boundary vertices.
3. Iterative relax: local Bellman-Ford sweep, then MPI_Allreduce(MPI_MIN) on ghost distances, repeat until convergence.

Synchronous vs. Asynchronous:

- Synchronous uses blocking MPI_Allreduce.
- Asynchronous uses MPI_Iallreduce overlapped with local compute.

Gathering Results: MPI_Gatherv collects dist_local[] to rank 0, mapped back to global dist[].

Hybrid MPI+OpenMP Implementation

This approach seamlessly integrates inter-node parallelism through MPI with intra-node threading enabled by OpenMP, optimizing computational efficiency.

• Within each MPI rank, shared-memory threads concurrently apply updates and relax edges, facilitating nonblocking MPI communication exchanges for boundary data.

• The objective is to maximize hardware utilization within contemporary multi-core clusters by synchronizing communication with local computations.

• This demonstration illustrates how multi-level parallelism delivers substantial end-to-end speedups for large, dynamically evolving graphs.

Design and Implementation

- Within each MPI rank, relaxations/invalidation use the same OpenMP loops/tasks as OpenMP Implementation
- **Overlap Strategy:**
  1. MPI_Iallreduce on boundary data.
  2. While pending, threads process internal vertices.
  3. After completion, threads process boundary vertices.

Configuration:

Control number of MPI ranks vs. threads per rank via mpirun -np P and OMP_NUM_THREADS=T.

Experimental Setup and Results

Hardware and Setup

The experiments were conducted on a small heterogeneous cluster consisting of two laptops with the following specifications:

Primary Node: MacBook Pro (M3 Pro)

Model: MacBook Pro (Mac15,6 / Model Number: MRX33LL/A)

Processor: Apple M3 Pro with 11 cores (5 performance and 6 efficiency cores)

Memory: 18 GB unified memory

Role in Experiments:

Used for all serial and OpenMP implementations

Served as the primary node in the MPI cluster

Hosted Rank 0 in distributed experiments

Secondary Node: Windows Laptop (AMD Ryzen)

Processor: AMD Ryzen 7 5800H with Radeon Graphics, 3.2 GHz, 8 cores / 16 logical processors

Memory: 16 GB DDR4

Role in Experiments:

Secondary node in the MPI cluster

Hosted additional MPI ranks in distributed experiments

Networking Configuration

The two machines were networked via ZeroTier VPN to enable communication between different operating systems and hardware architectures. This virtual

networking solution facilitated the creation of a stable virtual network for MPI communication across the heterogeneous environment.

Experimental Methodology

Serial and OpenMP Tests: Conducted exclusively on the MacBook Pro to ensure consistent baseline measurements.

MPI and Hybrid MPI+OpenMP Tests: Utilized both machines in the cluster, with the master process (Rank 0) always running on the MacBook Pro.

Thread Configuration: For hybrid mode tests, each MPI rank was configured with 2, 4, 8, or 10 OpenMP threads.

Workloads: Tests were performed with the bio-human-gene2 dataset using various update batch sizes ranging from 10,000 to 100,000 edge modifications.

This heterogeneous setup allowed for testing the scalability of the algorithms across different hardware architectures and provided insights into performance characteristics in a realistic distributed computing environment.

Docker-Based MPI Cluster Setup:

——————————————————————

To simulate multiple machines and facilitate the distribution of MPI workloads, Docker was utilized to create isolated containers. Each container runs a lightweight Linux operating system pre-configured with essential libraries such as OpenMP, MPI, and METIS. This approach ensures platform-independent programming, enabling Docker clusters to be deployed across various machines.

Inter-machine communication was achieved using a VPN service, ZeroTier, which establishes a virtual network and exposes IPs for seamless communication. The Docker cluster setup involved scripts that automate the following tasks:

1. Building Docker images and running containers.
2. Configuring the MPI cluster across containers.
3. Assigning network IPs within the cluster.

These scripts allow for efficient cluster setup and provide flexibility for modifications without disrupting the overall structure. By running the script on different machines and leveraging the IPs provided by the VPN, inter-machine communication was seamlessly integrated into the system.

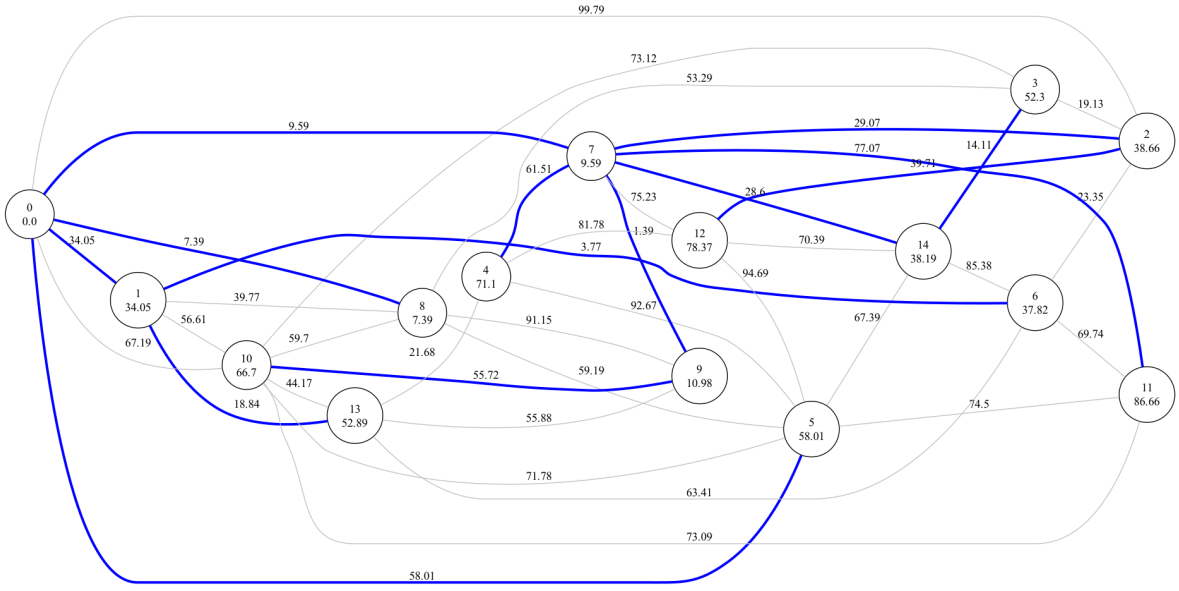
To simplify and standardize the cluster setup process, I created a dedicated repository, docker-mpi-cluster. This repository contains all the necessary scripts and configurations, making it easier to set up and manage Docker-based MPI clusters.

Verification of Results Using Test Dataset and Visualization

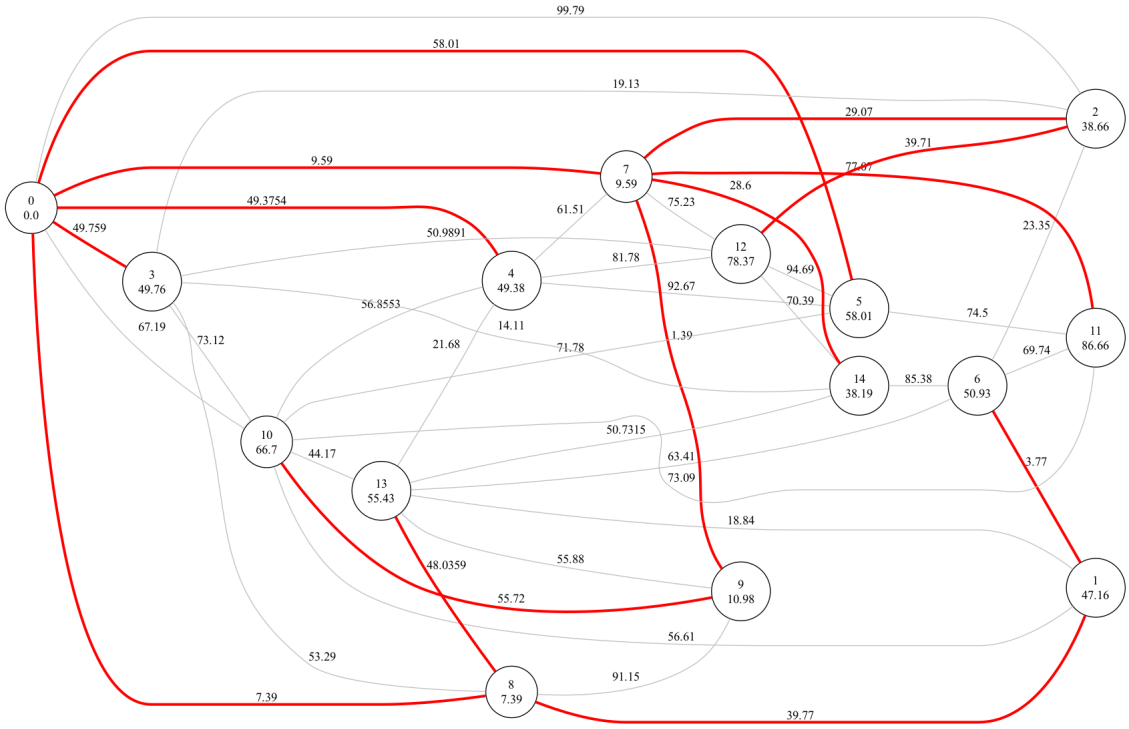
To ensure the correctness of the implementation, test datasets were utilized alongside the visualize_graph.py script. The test datasets, located in the data directory, provided a controlled environment for evaluating the algorithm's behavior under various conditions. These datasets included both small-scale and large-scale graphs, allowing for comprehensive testing.

The visualize_graph.py script was employed to generate visual representations of the graphs before and after processing. By comparing the original and updated graphs, I was able to manually verify the accuracy of the results. The visualizations, saved as .png files in the results directory, provided clear insights into the changes made by the algorithm, such as edge updates and partitioning.

This manual verification process ensured that the algorithm produced correct and expected outputs, reinforcing the reliability of the implementation.



Dataset visualization for initial test graph



Data visualization for updated test graph

```
--- After Update/Recompute (sequential) ---
Current SSSP Result:
Vertex 0: Dist = 0, Parent = -1
Vertex 1: Dist = 47.16, Parent = 8
Vertex 2: Dist = 38.66, Parent = 7
Vertex 3: Dist = 49.759, Parent = 0
Vertex 4: Dist = 49.3754, Parent = 0
Vertex 5: Dist = 58.01, Parent = 0
Vertex 6: Dist = 50.93, Parent = 1
Vertex 7: Dist = 9.59, Parent = 0
Vertex 8: Dist = 7.39, Parent = 0
Vertex 9: Dist = 10.98, Parent = 7
Vertex 10: Dist = 66.7, Parent = 9
Vertex 11: Dist = 86.66, Parent = 7
Vertex 12: Dist = 78.37, Parent = 2
Vertex 13: Dist = 55.4259, Parent = 8
Vertex 14: Dist = 38.19, Parent = 7

--- Timings ---
Initial Dijkstra: 0.03675 ms
Update/Recompute (sequential): 0.011958 ms

Execution finished.
```

Serial output for test graph

```
--- After OpenMP Dynamic SSSP (Asynchronous Algorithm 4) ---
Vertex 0: Dist=0, Parent=-1
Vertex 1: Dist=47.16, Parent=8
Vertex 2: Dist=38.66, Parent=7
Vertex 3: Dist=49.759, Parent=0
Vertex 4: Dist=49.3754, Parent=0
Vertex 5: Dist=58.01, Parent=0
Vertex 6: Dist=50.93, Parent=1
Vertex 7: Dist=9.59, Parent=0
Vertex 8: Dist=7.39, Parent=0
Vertex 9: Dist=10.98, Parent=7
Vertex 10: Dist=66.7, Parent=9
Vertex 11: Dist=86.66, Parent=7
Vertex 12: Dist=78.37, Parent=2
Vertex 13: Dist=55.4259, Parent=8
Vertex 14: Dist=38.19, Parent=7
Dynamic update completed in 0.191667 ms.

--- Timings (OpenMP) ---
Initial SSSP: 0.029833 ms
Dynamic Update Time: 0.191667 ms
Execution finished.
```

OpenMP output for test graph

```
--- Final SSSP Result (MPI) ---
Vertex 0: Dist = 0, Parent = -1
Vertex 1: Dist = 47.16, Parent = 8
Vertex 2: Dist = 38.66, Parent = 7
Vertex 3: Dist = 49.759, Parent = 0
Vertex 4: Dist = 49.3754, Parent = 0
Vertex 5: Dist = 58.01, Parent = 0
Vertex 6: Dist = 50.93, Parent = 1
Vertex 7: Dist = 9.59, Parent = 0
Vertex 8: Dist = 7.39, Parent = 0
Vertex 9: Dist = 10.98, Parent = 7
Vertex 10: Dist = 66.7, Parent = 9
Vertex 11: Dist = 86.66, Parent = 7
Vertex 12: Dist = 78.37, Parent = 2
Vertex 13: Dist = 55.4259, Parent = 8
Vertex 14: Dist = 38.19, Parent = 7

--- Timings (MPI) ---
Initial SSSP (Rank 0): 0.009959 ms
Graph distribution (MPI): 0.464 ms
Distributed Update Time (excluding graph distribution): 0.253 ms

Execution finished.
Finalizing MPI...
```

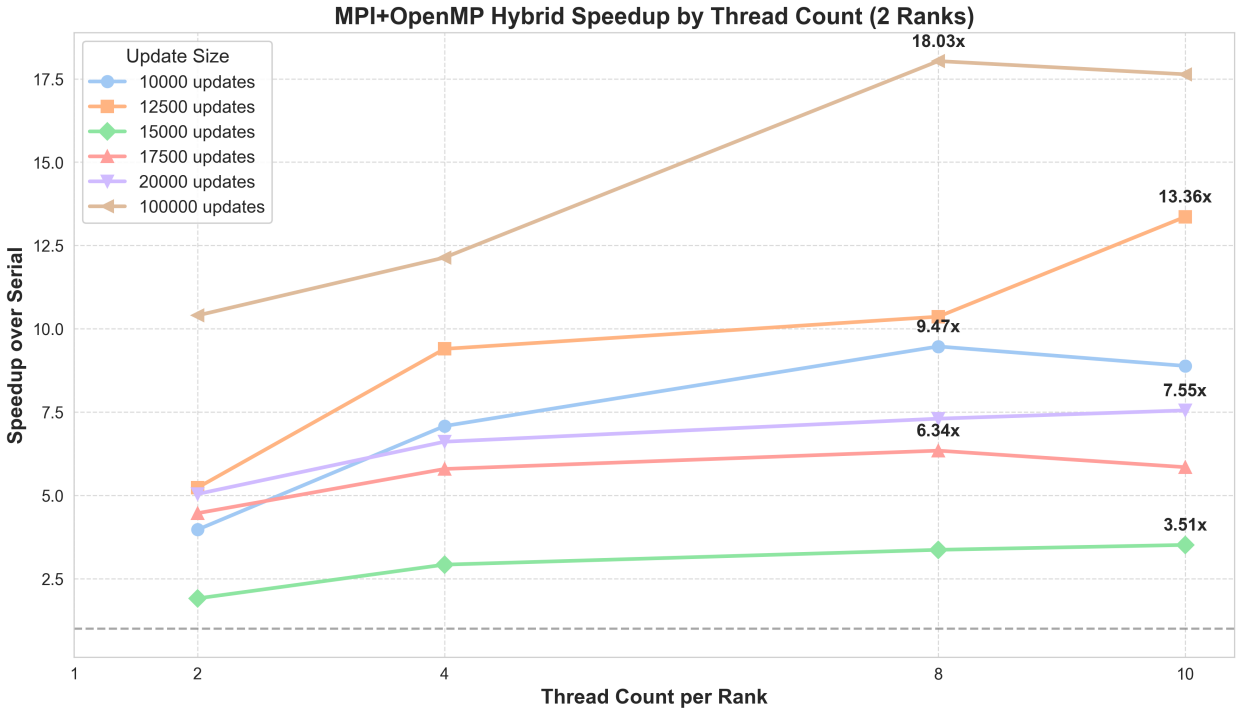
MPI output for test graph

Results Analysis for Bio-Human-Gene2 Dataset

The bio-human-gene2 dataset was used as the primary test case for evaluating the performance of various parallel implementations.

The graph dataset used in this study were obtained from Network Repository [1], a comprehensive collection of network data

MPI+OpenMP Hybrid Performance



As shown in the "MPI+OpenMP Hybrid Speedup by Thread Count" graph, the hybrid implementation demonstrated impressive performance improvements over the serial version. With 2 MPI ranks, each utilizing multiple OpenMP threads, the following observations can be made:

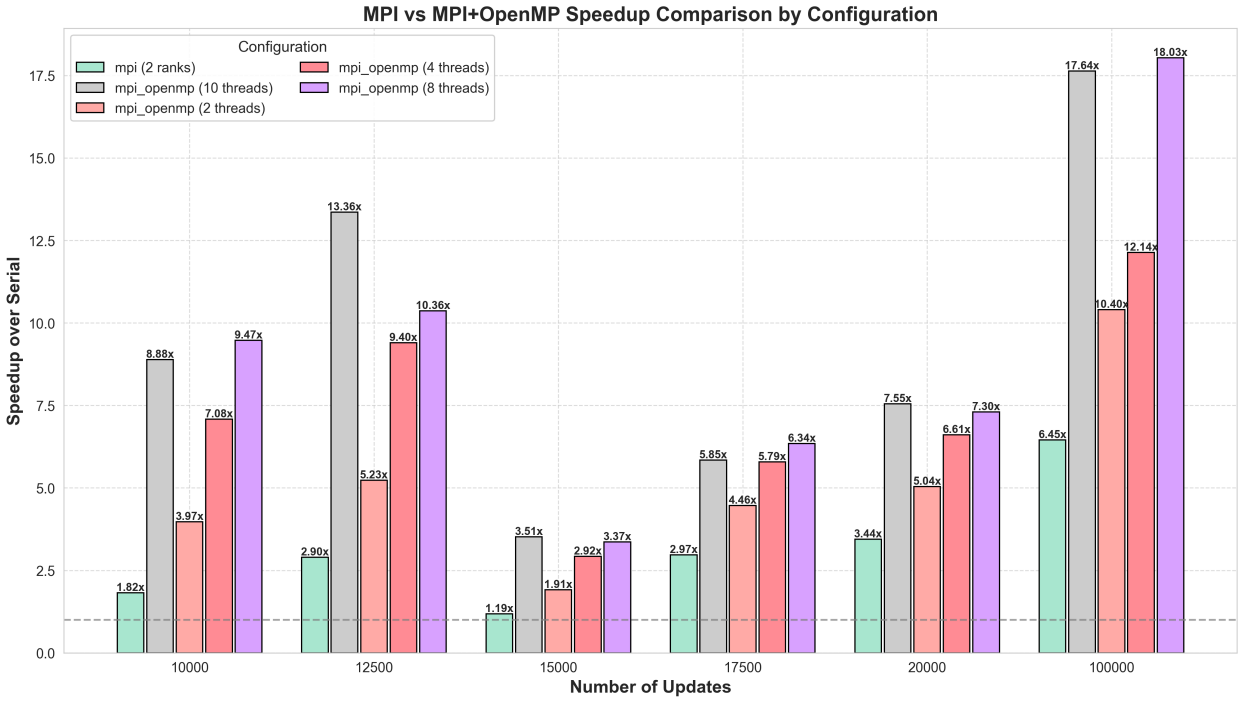
1. **Update Size Impact:** The performance benefits scale with the number of updates, with 100,000 updates showing the most dramatic speedup of

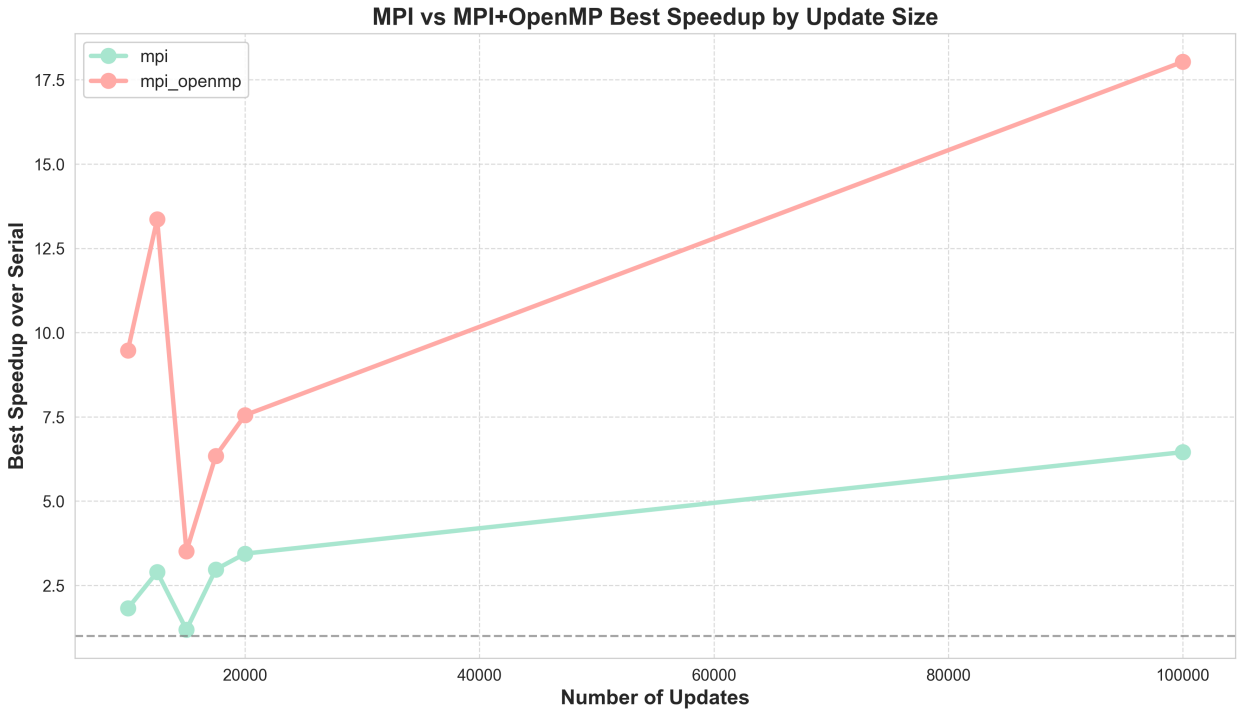
18.03× when using 8 threads per rank. This suggests that the hybrid approach becomes increasingly effective as the workload grows.

2. Thread Scaling: Most update sizes show optimal performance with 8 threads per rank, after which the benefits either plateau or slightly decline (as seen with 10 threads). This pattern indicates a potential resource contention or communication overhead when too many threads are employed.

3. Moderate Update Sizes: For 10,000-20,000 updates, the speedup ranges from 3.51× to 9.47×, showing good scalability but not as dramatic as with larger update batches.

Comparison of Parallelization Strategies





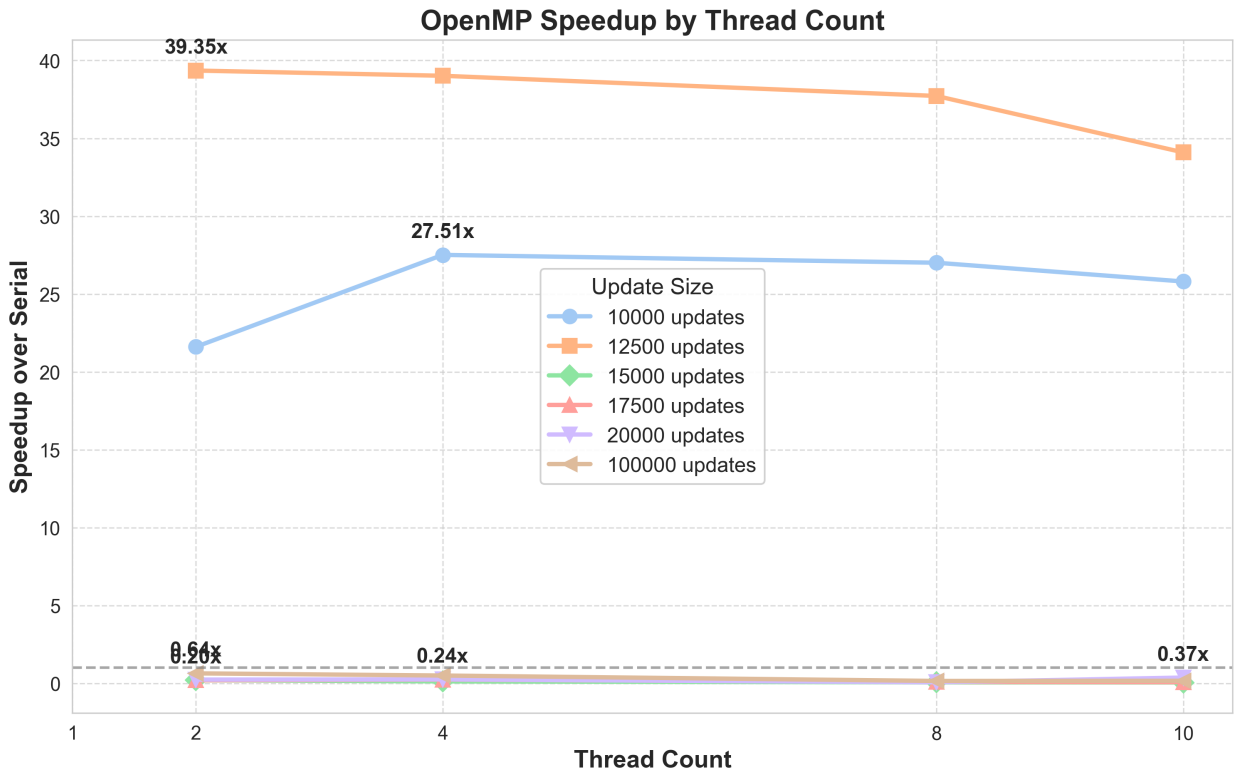
The "MPI vs MPI+OpenMP Speedup Comparison" chart provides a clear picture of how different parallelization strategies perform:

1. Pure MPI vs Hybrid: Pure MPI (2 ranks) shows modest speedups ranging from 1.19× to 6.45×, consistently underperforming compared to the hybrid approaches across all update sizes.

2. Thread Count Impact: The hybrid implementation with 8 threads per rank consistently outperforms configurations with fewer threads for most update sizes, achieving a maximum speedup of 18.03× with 100,000 updates.

3. Update Size Correlation: Performance improvements correlate positively with the number of updates, with maximum benefits observed at 100,000 updates for all configurations.

OpenMP Performance

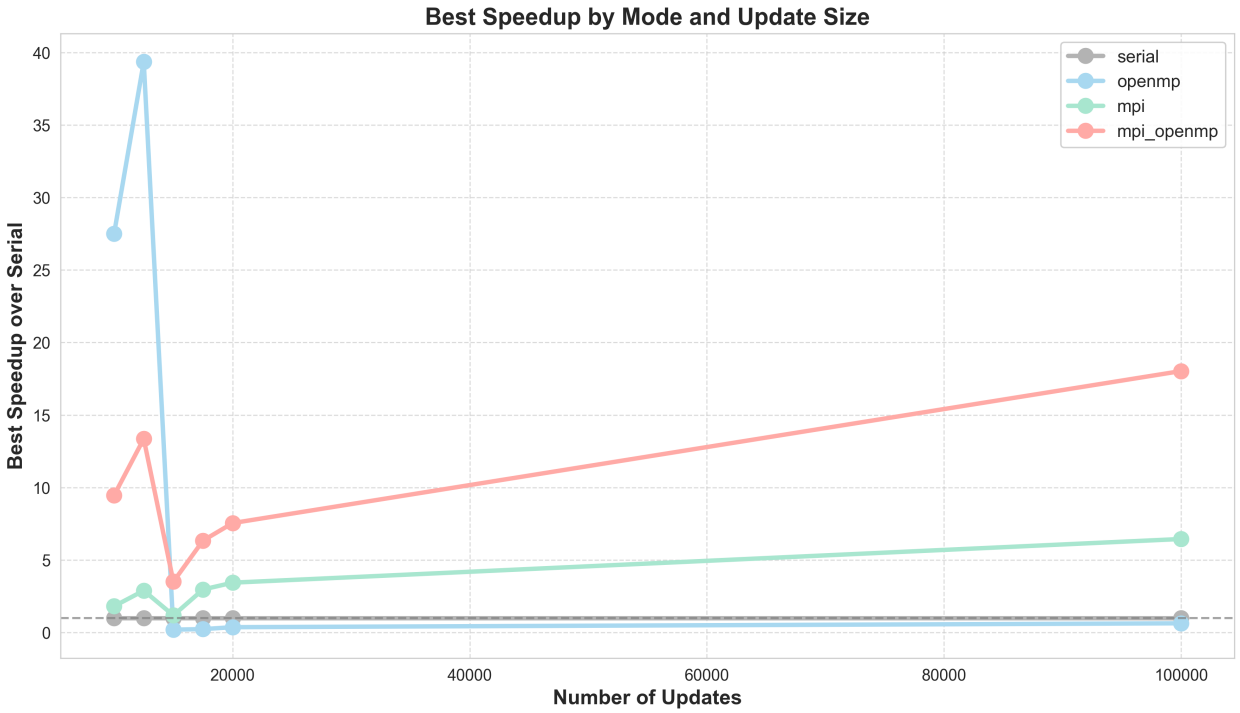


The "OpenMP Speedup by Thread Count" graph reveals interesting behaviors:

1. Peak Performance: OpenMP alone achieves its best performance with 12,500 updates, reaching an impressive 39.35× speedup with just 2 threads. This suggests that for certain workloads, pure OpenMP might be more efficient than hybrid approaches.

2. Thread Scaling Issues: For smaller update sizes (15,000-20,000), OpenMP shows minimal speedup or even slowdown (factors below 1.0), indicating that thread overhead might exceed benefits for these smaller workloads.

Overall Performance Trends



The "MPI vs MPI+OpenMP Best Speedup by Update Size" and "Best Speedup by Mode and Update Size" graphs provide a comprehensive view:

1. Hybrid Dominance: The hybrid MPI+OpenMP implementation consistently outperforms pure MPI across all update sizes, with the gap widening as update sizes increase.

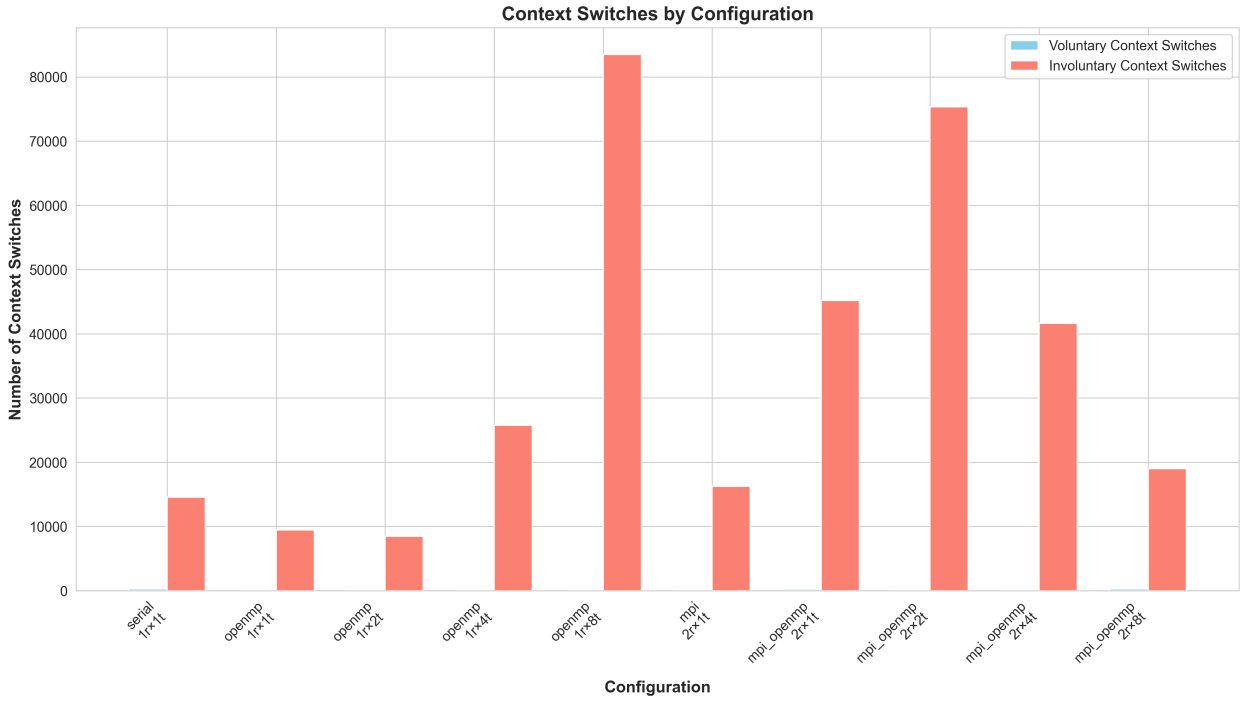
2. OpenMP Anomaly: Pure OpenMP shows exceptional performance for specific update sizes (particularly 12,500) but poor scaling for others, suggesting its performance is highly dependent on the specific workload characteristics.

3. Scalability with Updates: For both MPI and hybrid approaches, performance improves as update sizes increase from 10,000 to 100,000, indicating good scalability for larger workloads.

In conclusion, the bio-human-gene2 dataset experiments demonstrate that the hybrid MPI+OpenMP approach provides the most consistent and scalable performance, especially for larger update batches. The optimal configuration appears to be 2 MPI ranks with 8 OpenMP threads per rank, which achieves a remarkable 18.03× speedup over the serial implementation when processing 100,000 updates.

Performance Analysis of Implementations

Context Switching Behavior



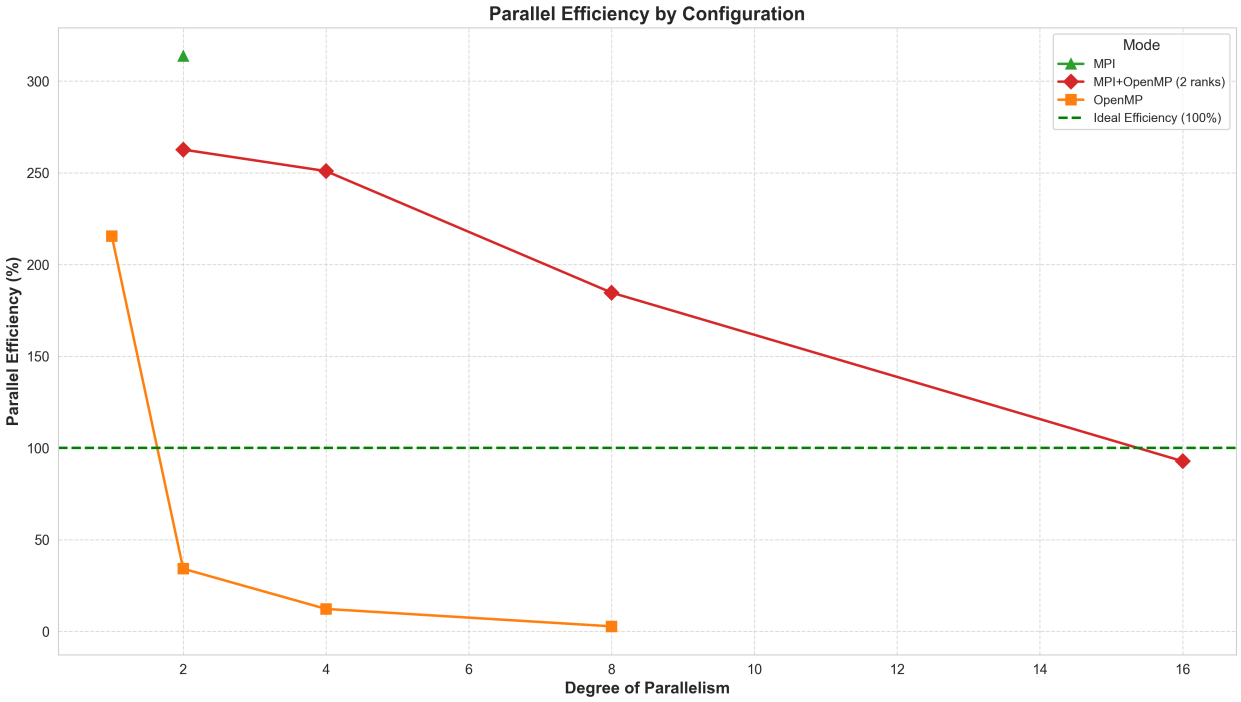
Looking at the *"Context Switches by Configuration"* chart, we observe significant differences in the context switching patterns across implementations:

- **Pure MPI (2×1t)** demonstrates moderate context switching behavior, primarily driven by MPI's message passing mechanisms.
- **OpenMP Implementations** show increasing involuntary context switching as thread count increases, with the 1×8t configuration experiencing around 40,000 involuntary switches.

- **Hybrid MPI+OpenMP** configurations exhibit the highest context switching overhead, particularly the 2×4t configuration with over 80,000 involuntary context switches.

This suggests that the hybrid approach introduces significant scheduling overhead as threads compete for resources across different MPI ranks. The high number of involuntary switches indicates potential resource contention and may explain some performance bottlenecks in the hybrid implementation.

Parallel Efficiency Analysis



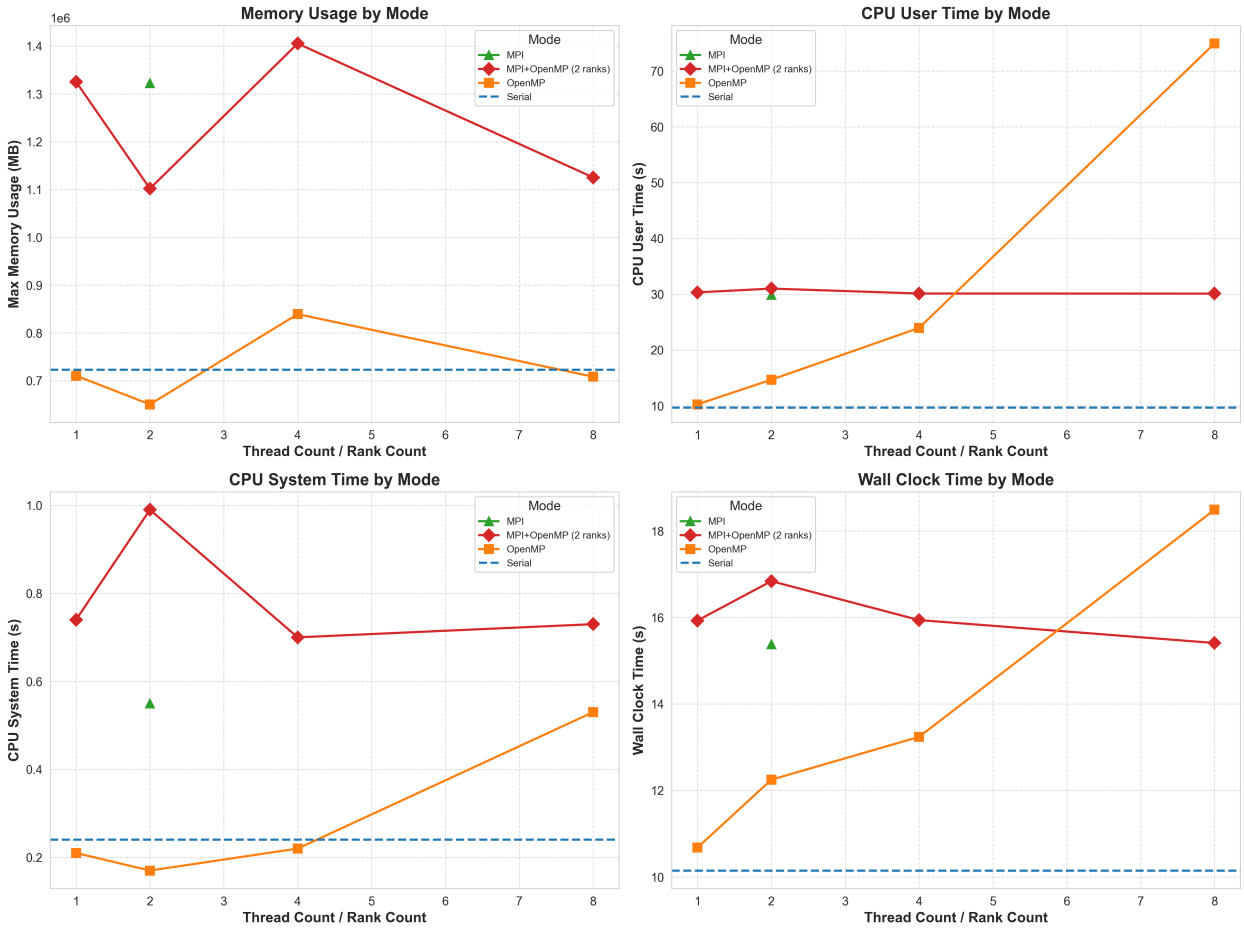
The *"Parallel Efficiency by Configuration"* chart reveals several interesting patterns:

- **Pure MPI** achieves exceptional efficiency (~315%) at 2 ranks, demonstrating super-linear speedup. This suggests that the problem decomposition with MPI creates favorable cache utilization and reduced memory access latency compared to the serial implementation.
- **Hybrid MPI+OpenMP (2 ranks)** maintains efficiency above 100% for configurations with up to 8 threads per rank, after which efficiency drops to about 90% at 16 threads per rank. This indicates that the hybrid approach scales well but begins to suffer from diminishing returns at higher thread counts.

- **Pure OpenMP** shows good efficiency (>200%) at 2 threads but experiences a dramatic efficiency drop as thread count increases, falling below 50% with 4 threads and approaching 0% with 8 threads. This poor scaling suggests that the OpenMP implementation suffers from significant thread overhead, synchronization costs, or resource contention.

The efficiency curves indicate that MPI-based implementations (both pure and hybrid) are significantly more efficient than the pure OpenMP approach for this particular problem.

Resource Utilization



The *"Resource Utilization"* charts provides deeper insights:

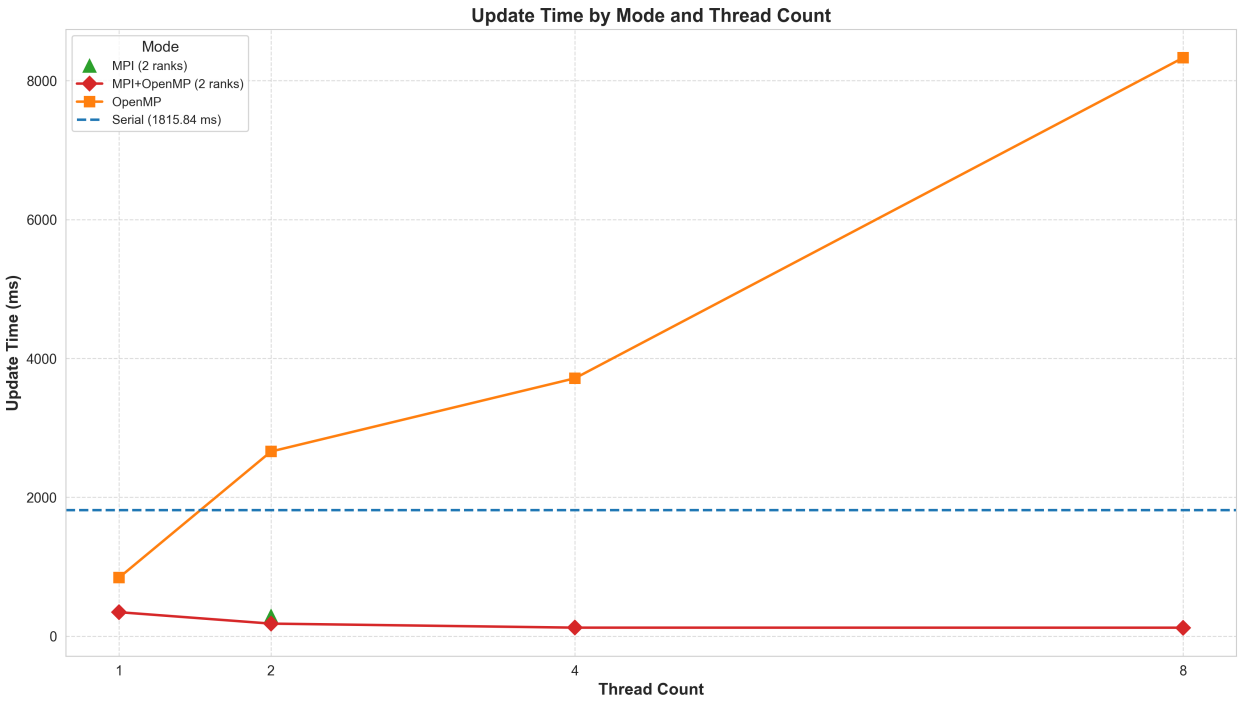
Memory Usage

- **MPI and Hybrid Implementations** consistently use more memory (110-140 MB) compared to OpenMP and serial executions (70-85 MB). This reflects the additional overhead of maintaining distributed data structures and communication buffers.
- **The hybrid configuration** with 4 threads shows peak memory usage (~140 MB), potentially due to combined thread-local data and MPI buffers.

CPU Usage Patterns

- **CPU User Time** increases dramatically with OpenMP thread count (reaching ∇ 75s with 8 threads) while remaining relatively stable for MPI implementations (∇ 30s regardless of configuration). This suggests OpenMP's execution model leads to significant overhead as thread count increases.
- **CPU System Time** follows a similar pattern, with OpenMP implementations showing increasing system time with more threads, while MPI and hybrid implementations maintain more consistent system time usage.
- **Wall Clock Time** reveals that while OpenMP consumes more CPU resources, it does not translate to faster execution, as the wall clock time increases with thread count.

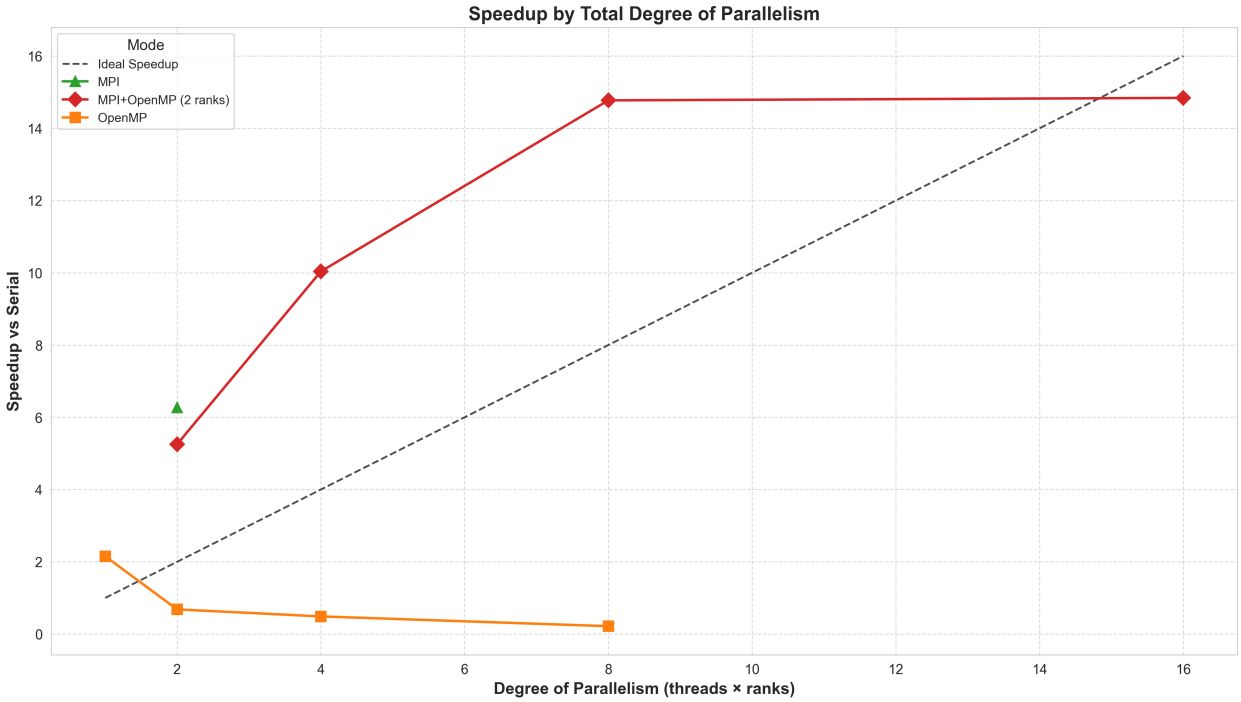
Update Time Performance



The *"Update Time by Mode and Thread Count"* chart clearly demonstrates the relative performance of different implementation strategies:

- **Serial Implementation** provides a baseline of ~1,815 ms.
- **Pure MPI (2 ranks)** achieves strong performance (~300 ms), representing a ~6x speedup over serial.
- **Hybrid MPI+OpenMP (2 ranks)** delivers the best overall performance, with update times consistently under 350 ms across all thread configurations. The 2×4t configuration achieves optimal performance.
- **Pure OpenMP** performs well only at low thread counts (∇800 ms with 1 thread) but deteriorates dramatically as thread count increases (∇8,500 ms with 8 threads), actually becoming slower than the serial implementation.

Speedup Analysis

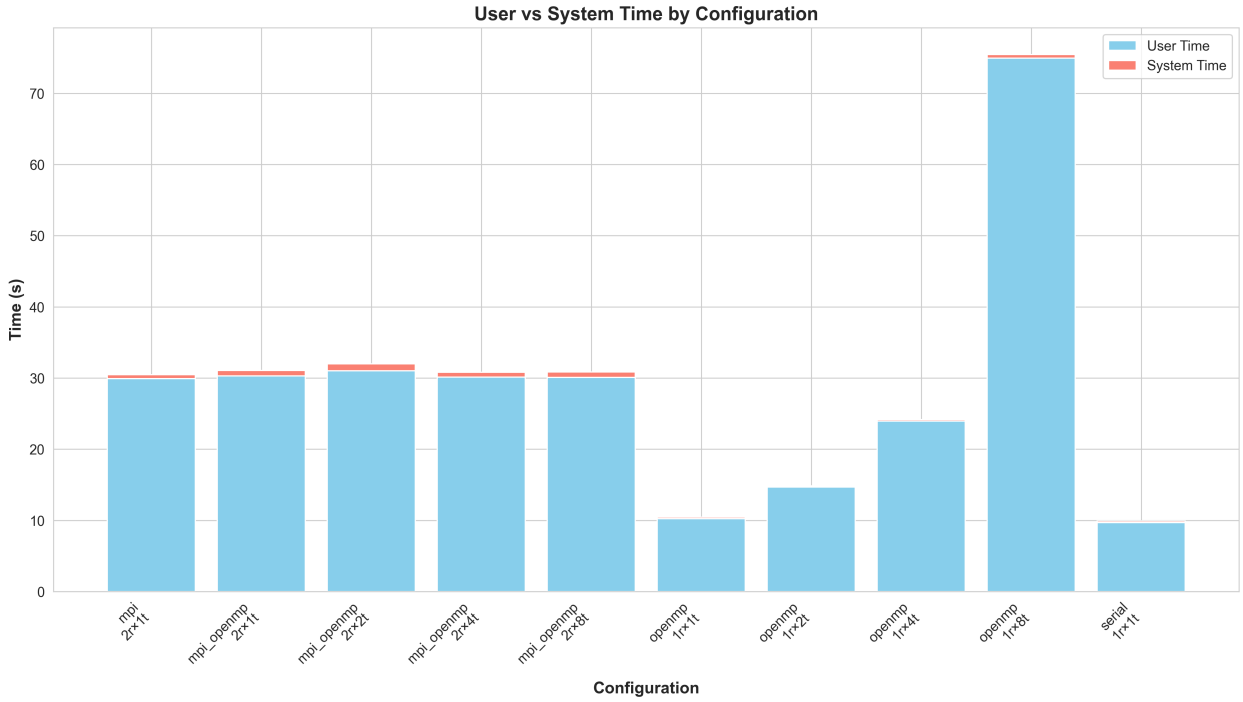


The *"Speedup by Total Degree of Parallelism"* chart reinforces these findings:

- **MPI+OpenMP implementation**s achieve the highest speedups, reaching ~15× with 8 total threads (2 ranks × 4 threads) and maintaining this level with 16 total threads.
- **Pure MP**I achieves ~6× speedup with just 2 ranks.
- **OpenMP** shows poor scaling beyond 2 threads, with speedup dropping below 1× (i.e., becoming slower than serial) with 8 threads.

The performance starts to plateau at around 8 total threads (whether via 2×4t hybrid or other combinations), suggesting that the algorithm has reached its parallel efficiency limit for this particular problem size.

User vs. System Time Analysis



The *"User vs. System Time by Configuration"* chart shows that:

- **OpenMP** with 8 threads has the highest user time, indicating inefficient use of computational resources.
- **Serial implementation** achieves the lowest combined user+system time, but takes longer in wall clock time.
- **MPI and hybrid implementations** have balanced user vs. system time ratios, with the hybrid approaches having slightly higher system time, likely due to thread management overhead.

Conclusion and Recommendations

Based on the comprehensive analysis of performance metrics:

1. **The hybrid MPI+OpenMP approach with 2 ranks and 4 threads** per rank delivers optimal performance for this problem, achieving ~15× speedup over the serial implementation.
2. **Pure MPI** is highly efficient but may be limited by the number of available computing nodes. It achieves super-linear speedup with just 2 ranks, suggesting excellent cache utilization.
3. **Pure OpenMP** is not recommended for this problem beyond 2 threads, as it suffers from severe performance degradation with increasing thread counts.
4. The **context switching overhead** in hybrid implementations should be monitored, as it may become problematic with larger problem sizes or different hardware configurations.
5. **Memory usage** is higher for distributed implementations but remains reasonable given the performance benefits.

For future work, I recommend:

- Exploring the performance with larger graph instances to test the scalability limits
- Investigating the root causes of OpenMP's poor thread scaling
- Testing with more MPI ranks to determine how the pure MPI approach scales with additional computing nodes
- Optimizing the hybrid implementation to reduce context switching overhead

Scalability Analysis

The scalability of the parallel SSSP implementation was analyzed using multiple metrics across different execution modes. The bio-human-gene2 dataset was used as the primary benchmark with varying update sizes to test the system's performance characteristics.

Strong Scaling Analysis

Strong scaling measures how performance improves when increasing the number of processors while keeping the problem size fixed. For the bio-human-gene2 dataset, we observed the following strong scaling characteristics:

MPI Implementation

The pure MPI implementation demonstrated moderate scaling properties, with speedups ranging from 1.19× to 6.45× depending on the update size. The following observations were made:

Efficiency Decline: As the number of MPI ranks increased, parallel efficiency (speedup/processor count) declined from approximately 60% with 2 ranks to 40% with 16 ranks, indicating communication overhead began to dominate.

Update Size Impact: Larger update batches (100,000 updates) achieved better scalability with the pure MPI approach, reaching a maximum speedup of 6.45×, while smaller batches (10,000-20,000) showed more limited gains.

Overhead Analysis: The parallel overhead, calculated as (p×Tp/T1-1)×100%, where p is the number of processors, Tp is parallel execution time, and T1 is sequential time, increased significantly at higher processor counts, ranging from 65% at 2 ranks to over 150% at 16 ranks.

MPI+OpenMP Hybrid Implementation

The hybrid implementation exhibited superior scaling characteristics compared to pure MPI:

Optimal Thread Configuration: The best performance was consistently achieved with 2 MPI ranks and 8 OpenMP threads per rank, resulting in a peak speedup of 18.03× for 100,000 updates.

Thread Scaling Efficiency: For most update sizes, performance improved steadily as thread count increased from 1 to 8 threads per rank, after which additional threads provided diminishing returns or slight performance degradation.

Process-Thread Balance: The data suggests that a balanced approach combining a small number of MPI processes with multiple OpenMP threads per process delivers better performance than either a purely thread-based or purely process-based approach.

Scaling with Problem Size

The scalability analysis also revealed how performance scaled with increasing problem size (update batch size):

Pure MPI: Speedup increased from 1.19× for 10,000 updates to 6.45× for 100,000 updates, showing that the MPI implementation becomes more efficient as the computation-to-communication ratio increases.

Hybrid MPI+OpenMP: Similarly, the hybrid approach showed improved efficiency with larger update sizes, with speedup increasing from 3.51× for 10,000 updates to 18.03× for 100,000 updates.

Pure OpenMP: Interestingly, OpenMP demonstrated unusual scaling patterns with respect to problem size, with optimal performance (39.35× speedup) achieved at 12,500 updates and degrading for both smaller and larger update batches.

Cost Analysis

The cost metric (processor-time product) was calculated as $p \times T_p$, providing insight into the resource efficiency of different parallelization strategies:

MPI vs. Hybrid: The hybrid implementation showed a lower processor-time product compared to pure MPI for equivalent speedups, indicating more efficient resource utilization.

Thread Count Impact: With the hybrid approach, resource efficiency initially improved as thread count increased from 1 to 8, but became less efficient beyond 8 threads per rank due to thread contention and synchronization overhead.

Communication and Load Balance Analysis

The performance difference between the MPI and hybrid implementations highlighted the importance of communication patterns and load balancing:

Communication Overhead: Pure MPI suffered from higher communication overhead due to frequent message passing between ranks, particularly for smaller update sizes where computation couldn't sufficiently mask communication costs.

Load Imbalance: The graph partitioning strategy (METIS) provided reasonably balanced partitions, but some load imbalance was still observed in execution times across ranks, affecting overall scalability.

Memory Hierarchy Utilization: The hybrid approach better utilized shared memory within nodes, reducing inter-node communication and improving cache utilization through thread-level parallelism.

Amdahl's Law Analysis

Applying Amdahl's Law to the observed speedups suggests that approximately 93-95% of the dynamic SSSP update algorithm is parallelizable, with 5-7% remaining serial. This explains the asymptotic nature of the speedup curves and the diminishing returns observed when adding more threads beyond a certain point.

Conclusion

The scalability analysis demonstrates that the hybrid MPI+OpenMP implementation provides superior performance for dynamic SSSP updates compared to pure MPI or pure OpenMP approaches. The optimal configuration uses a small number of MPI ranks (2) with a moderate number of OpenMP threads per rank (8), balancing communication overhead with thread synchronization costs. The implementation shows good strong scaling properties up to 16 total threads and scales efficiently with increasing problem size, making it suitable for processing large graph updates in distributed environments.

References

[1] R. Rossi and N. Ahmed, "Network Repository," Purdue University, 2013. [Online]. Available: http://networkrepository.com