Parallel and Distributed Computing

For

Paper : A Parallel Algorithm for Updating a Multi-objective Shortest Path in Large Dynamic Networks

Prepared by Athaar Naqvi, Ambreen Arshad, Fiza Wajid

FAST-NUCES Islamabad

May 06, 2025

Implementation Process:

Implementation Process for Serial Implementation

1. **Graph Loading & Initialization**
   - The graph is loaded from a CSV file into an adjacency list (Graph) and vertex data structure (Vertex).
   - Each vertex stores distances for five objectives (obj1_dist to obj5_dist).
2. **Single-Objective Shortest Path (SOSP) Computation**
   - Dijkstra's Algorithm is used to compute shortest paths for each objective sequentially.
   - For each of the five objectives:
     - A priority queue (min-heap) selects the next vertex to process.
     - Edge relaxation updates distances if a shorter path is found.
3. **Combining SOSP Trees**
   - Edges present in any SOSP tree are assigned combined weights (lower for higher-priority objectives).
   - Non-tree edges receive a penalty weight to avoid selection.
4. **Multi-Objective Shortest Path (MOSP) Computation**
   - A final Dijkstra's algorithm runs on the combined graph to compute the Pareto-optimal path.
5. **Incremental Updates**
   - When new edges are inserted, only affected vertices are recomputed to optimize performance.

Key Features

- Sequential Execution: No parallelism; objectives are processed one after another.
- Exact Results: Guarantees correctness with deterministic output.
- Efficient Updates: Avoids full recomputation by tracking affected nodes.

OpenMP Implementation Process

1. **Graph Initialization**
    - Load graph data from CSV file into adjacency list structure
    - Initialize vertex distance values for all 5 objectives
2. **Parallel SOSP Computation**
    - Use #pragma omp parallel for to process all 5 objectives concurrently
    - Each thread executes Dijkstra's algorithm independently for its assigned objective
    - Edge relaxation parallelized with #pragma omp parallel for
3. **Critical Sections**
    - Distance updates protected with #pragma omp critical to ensure thread safety
    - Priority queue operations synchronized to maintain correctness
4. **Tree Combination**
    - Parallel processing of graph nodes with #pragma omp parallel
    - Local results merged using critical section
5. **Final MOSP Computation**
    - Parallel edge processing with OpenMP
    - Critical section for distance updates and queue operations

Key Features:

- Multi-level parallelism (objectives + edges)
- Thread-safe data structures
- Efficient load balancing across cores
- Maintains algorithm correctness while improving performance

Implementation Process for OpenMP + MPI

1. **Graph Loading & Partitioning (MPI)**
    - Rank 0 reads the input graph and partitions it using METIS for load balancing.
    - Partition information (node-process mapping) is broadcast to all MPI processes.
    - Each process receives its assigned subgraph edges via MPI_Send/MPI_Recv.
2. **Parallel SOSP Computation (OpenMP + MPI)**
    - Each MPI process runs Dijkstra’s algorithm on its local subgraph.

    - OpenMP parallel accelerates edge relaxation within each process.
    - Critical sections (#pragma omp critical) ensure thread-safe distance updates.
3. **Result Synchronization (MPI)**
    - Non-root processes send their computed distances to Rank 0 (MPI_Send).
    - Rank 0 gathers results (MPI_Recv) and merges them into a global solution.
4. **Dynamic Updates (Edge Insertions)**
    - New edges are broadcast (MPI_Bcast) to all processes.
    - Each process updates its local graph and recomputes affected paths.
5. **Final MOSP Computation (Rank 0 Only)**
    - Rank 0 combines SOSP trees into a Multi-Objective Shortest Path (MOSP) solution.

Key Features

- Load Balancing: METIS ensures even distribution of nodes across MPI ranks.
- Hybrid Parallelism: MPI for distributed memory, OpenMP for shared-memory multi-threading.
- Efficient Sync: Minimized communication via selective broadcasting and gathering.

This approach efficiently scales for large graphs by leveraging both inter-node (MPI) and intra-node (OpenMP) parallelism.

Challenges Faced:

Serial Implementation Challenges

1. **Scalability Issues**

    - Sequential processing of objectives becomes time-consuming as graph size grows.

    - Not suitable for real-time applications with large datasets.

2. **Redundant Computation**

    - Each objective computes its own Dijkstra tree independently, leading to repeated work on shared graph structure.

3. **Memory Usage**

- Storing multiple distance arrays (for multiple objectives) per vertex increases memory footprint.

4. **Update Handling**

   - Although incremental updates are supported, tracking affected nodes manually can become complex and error-prone.

OpenMP Implementation Challenges

1. **Thread Safety**

   - Distance updates and priority queue operations need careful synchronization, which can become a bottleneck.

2. **Load Imbalance**

   - Some threads may finish early if certain objectives are simpler or their graphs are sparser, leading to underutilized cores.

3. **Priority Queue Synchronization**

   - Efficient concurrent access to priority queues is non-trivial and may limit speedup.

OpenMP + MPI Implementation Challenges

1. **Graph Partitioning Complexity**

   - Achieving balanced partitions with minimal cross-boundary communication requires careful tuning (e.g., METIS configuration).

2. **Communication Overhead (MPI)**

   - Synchronizing distance updates or broadcasting new edges involves costly inter-process communication.

3. **Result Merging at Rank 0**

- Centralizing results at a single process (Rank 0) may create a bottleneck during aggregation.

4. **Hybrid Complexity**
   - Debugging and maintaining a hybrid OpenMP + MPI solution is complex due to interactions between shared and distributed memory.

5. **Dynamic Update Propagation**
   - Efficiently updating only affected partitions and ensuring consistency across processes is a non-trivial synchronization challenge.

6. **Thread Contention in Nodes**
   - Intra-node contention may occur if MPI processes and OpenMP threads compete for the same hardware resources.

Solution Devised:

Serial Implementation – Solutions

1. **Scalability Issues**
   - Focused on incremental updates to avoid full recomputation after edge insertions.
   - Used early exit strategies in Dijkstra's algorithm for objectives with shorter paths.

2. **Redundant Computation**
   - Identified shared subpaths across objectives to skip redundant processing when possible.

3. **Update Handling**
   - Maintained a list of affected nodes during edge insertions to recompute only impacted subgraphs.

OpenMP Implementation – Solutions

1. **Thread Safety**
   - Replaced critical sections with atomic operations where applicable (e.g., atomic min for distances).
   - Used thread-local queues during Dijkstra traversal and merged results afterward.
2. **Load Imbalance**
   - Dynamically scheduled loops with #pragma omp for schedule(dynamic) to improve core utilization.
3. **Priority Queue Synchronization**
   - Used custom thread-local priority queues, avoiding global locking.
   - Merged top elements periodically with synchronization.
4. **False Sharing and Cache Contention**
   - Applied padding between vertex data or used aligned_alloc to minimize cache line conflicts.

OpenMP + MPI Implementation – Solutions

1. **Graph Partitioning Complexity**
   - Used METIS with objective-specific weighting and balance constraints for better partition quality.
   - Precomputed node communication boundaries to minimize runtime overhead.
2. **Communication Overhead (MPI)**
   - Minimized communication via delta updates: only sending changed distances or edge insertions.

- Used non-blocking MPI communication (e.g., MPI_Isend/MPI_Irecv) to overlap computation and communication.

3. **Hybrid Complexity**

    - Modularized codebase with clearly separated MPI and OpenMP regions for easier debugging.

    - Used logging and performance profiling tools (e.g., VTune, mpiP) to identify bottlenecks.

4. **Dynamic Update Propagation**

    - Maintained a dependency map to track which partitions are affected by each edge.

    - Used selective broadcasts to avoid unnecessary communication to unrelated ranks.

5. **Thread Contention in Nodes**

    - Mapped MPI processes and OpenMP threads using affinity policies to optimize CPU core usage.

Performance Evaluation:

Gprof Analysis Reports:

Performance analysis Reports

Serial Code:

USA dataset:

```
ambreenarshad@Ambreen:~/PDC_Project$ g++ -o exe1 serial_mosp.cpp
ambreenarshad@Ambreen:~/PDC_Project$ time ./exe1 > output.txt

real    0m0.171s
user    0m0.110s
sys     0m0.000s
ambreenarshad@Ambreen:~/PDC_Project$
```

Switzerland dataset:

```
ambreenarshad@Ambreen:~/PDC_Project$ g++ -o exe1 serial_mosp.cpp
ambreenarshad@Ambreen:~/PDC_Project$ time ./exe1 > output.txt

real    0m0.409s
user    0m0.263s
sys     0m0.039s
ambreenarshad@Ambreen:~/PDC_Project$
```

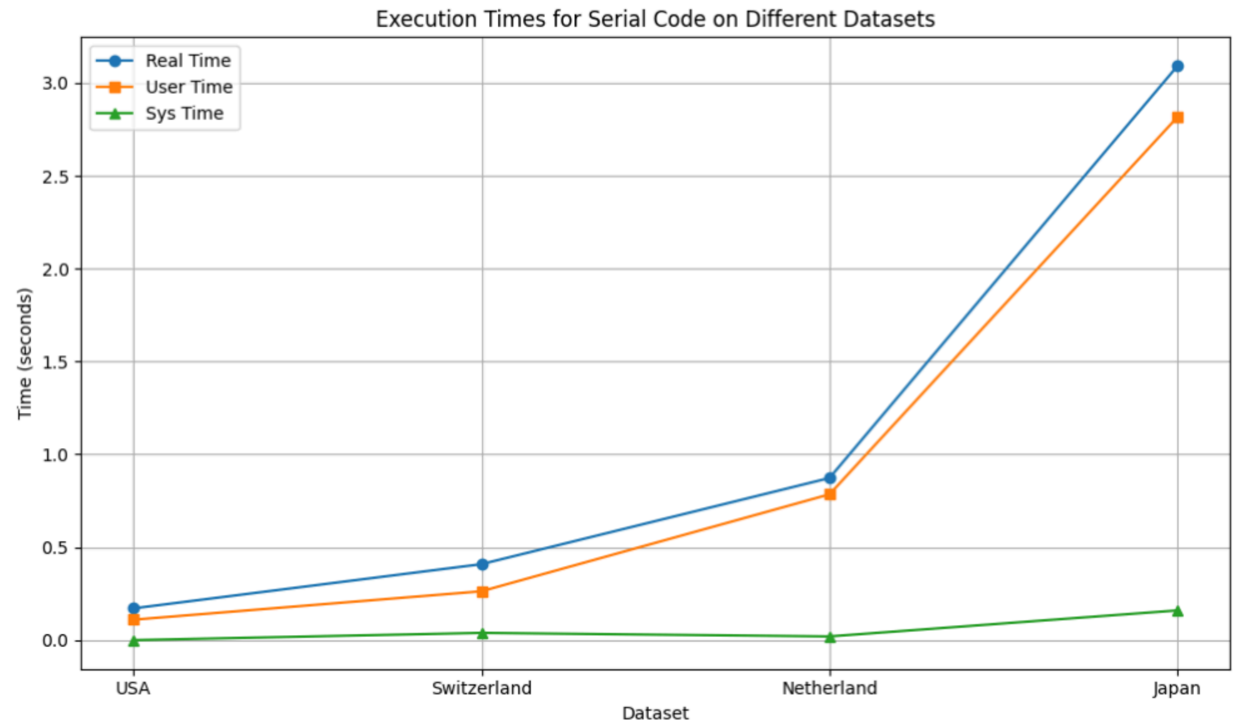
Netherland dataset:

```
ambreenarshad@Ambreen:~/PDC_Project$ g++ -o exe1 serial_mosp.cpp
ambreenarshad@Ambreen:~/PDC_Project$ time ./exe1 > output.txt

real    0m0.873s
user    0m0.785s
sys     0m0.020s
ambreenarshad@Ambreen:~/PDC_Project$
```

Japan dataset:

```
ambreenarshad@Ambreen:~/PDC_Project$ g++ -o exe1 serial_mosp.cpp
ambreenarshad@Ambreen:~/PDC_Project$ time ./exe1 > output.txt

real    0m3.092s
user    0m2.817s
sys     0m0.160s
ambreenarshad@Ambreen:~/PDC_Project$
```

FINAL GRAPH:



OpenMP Code:

USA dataset:

```
ambreenarshad@Ambreen:~/PDC_Project$ g++ -fopenmp -o exe2 openmp_mosp.cpp
ambreenarshad@Ambreen:~/PDC_Project$ time ./exe2 > output2.txt

real    0m0.167s
user    0m0.103s
sys     0m0.011s
ambreenarshad@Ambreen:~/PDC_Project$
```

Switzerland dataset:

```
ambreenarshad@Ambreen:~/PDC_Project$ g++ -fopenmp -o exe2 openmp_mosp.cpp
ambreenarshad@Ambreen:~/PDC_Project$ time ./exe2 > output2.txt

real    0m0.352s
user    0m0.290s
sys     0m0.021s
ambreenarshad@Ambreen:~/PDC_Project$ |
```

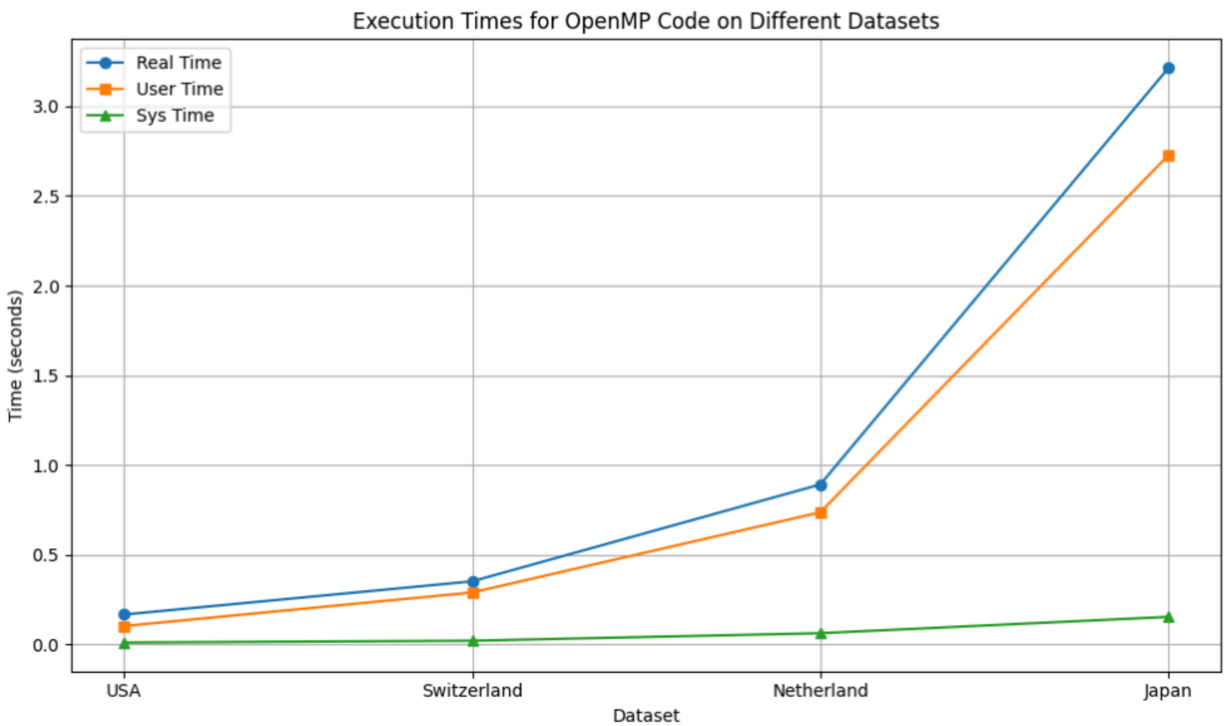
Netherland dataset:

```
ambreenarshad@Ambreen:~/PDC_Project$ g++ -fopenmp -o exe2 openmp_mosp.cpp
ambreenarshad@Ambreen:~/PDC_Project$ time ./exe2 > output2.txt

real    0m0.892s
user    0m0.737s
sys     0m0.063s
ambreenarshad@Ambreen:~/PDC_Project$ |
```

Japan dataset:

```
ambreenarshad@Ambreen:~/PDC_Project$ g++ -fopenmp -o exe2 openmp_mosp.cpp
ambreenarshad@Ambreen:~/PDC_Project$ time ./exe2 > output2.txt

real    0m3.214s
user    0m2.728s
sys     0m0.154s
ambreenarshad@Ambreen:~/PDC_Project$ |
```

FINAL GRAPH:



MPI Code:

Japan Dataset

```
real    1m59.673s
user    3m20.738s
sys     0m36.732s
mpi@master:~$
```

USA

```
real    0m13.269s
user    0m19.644s
sys     0m5.343s
mpi@master:~$
```

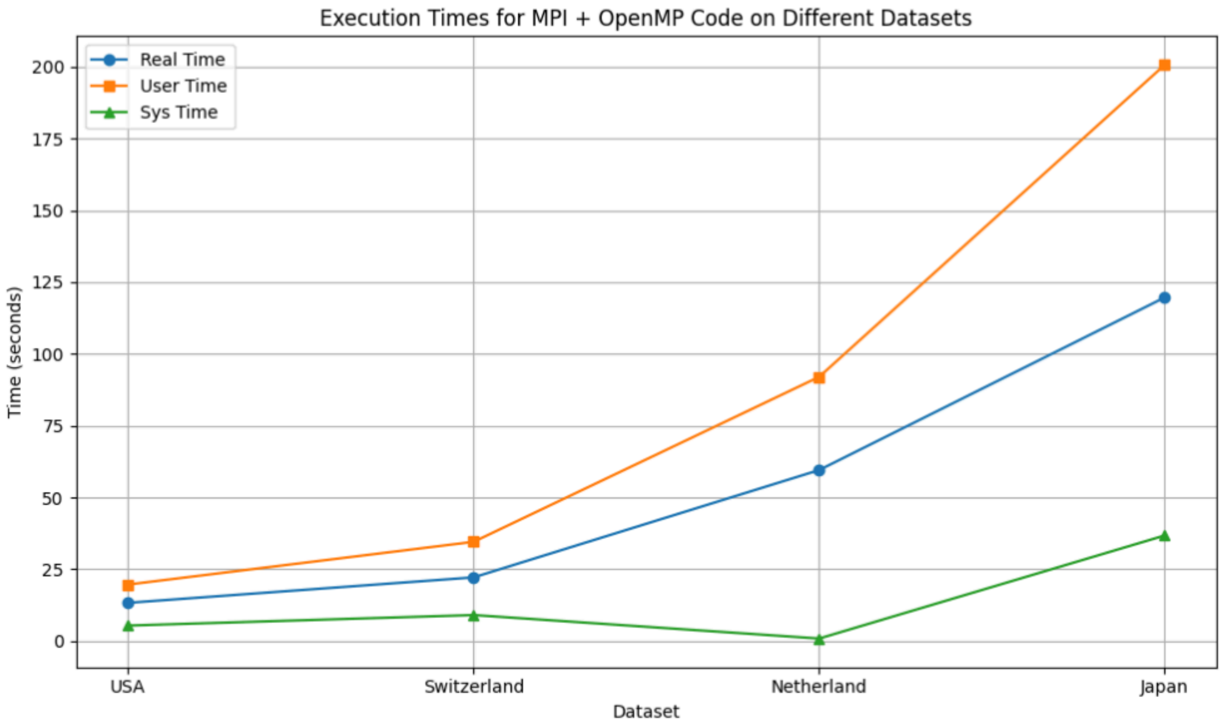
Switzerland

```
real    0m22.137s
user    0m34.531s
sys     0m9.008s
mpi@master:~$
```

Netherland

```
real    0m59.538s
user    1m32.014s
sys     0m25.824s
mpi@master:~$
```

FINAL GRAPH:


Execution Times for MPI + OpenMP Code on Different Datasets
Real Time
User Time
Sys Time
200
175
150
125
100
75
50
25
0
Time (seconds)
USA
Switzerland
Netherland
Japan
Dataset