<u>PDC FINAL PROJECT</u>

Submitted by:
Ziyad Hisham (22L-6855)
Hania Fayyaz (22L-6941)
Fatima Hameed (22K-4656)

Performance Report: OpenMP and MPI+OpenMP Codes for MOSP

Overview of the Codes

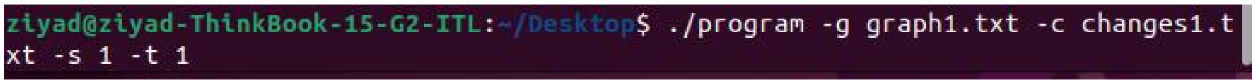
Both codes implement a Multi-Objective Shortest Path (MOSP) algorithm using a modified Bellman-Ford approach to find shortest paths in a graph with multiple objectives. The codes handle a directed graph with 1000 nodes and 100,000 edges, processing multiple objective functions (e.g., different edge weights for each objective). They compute shortest paths from a source node in an ensemble graph and individual objective graphs.

OpenMP Code

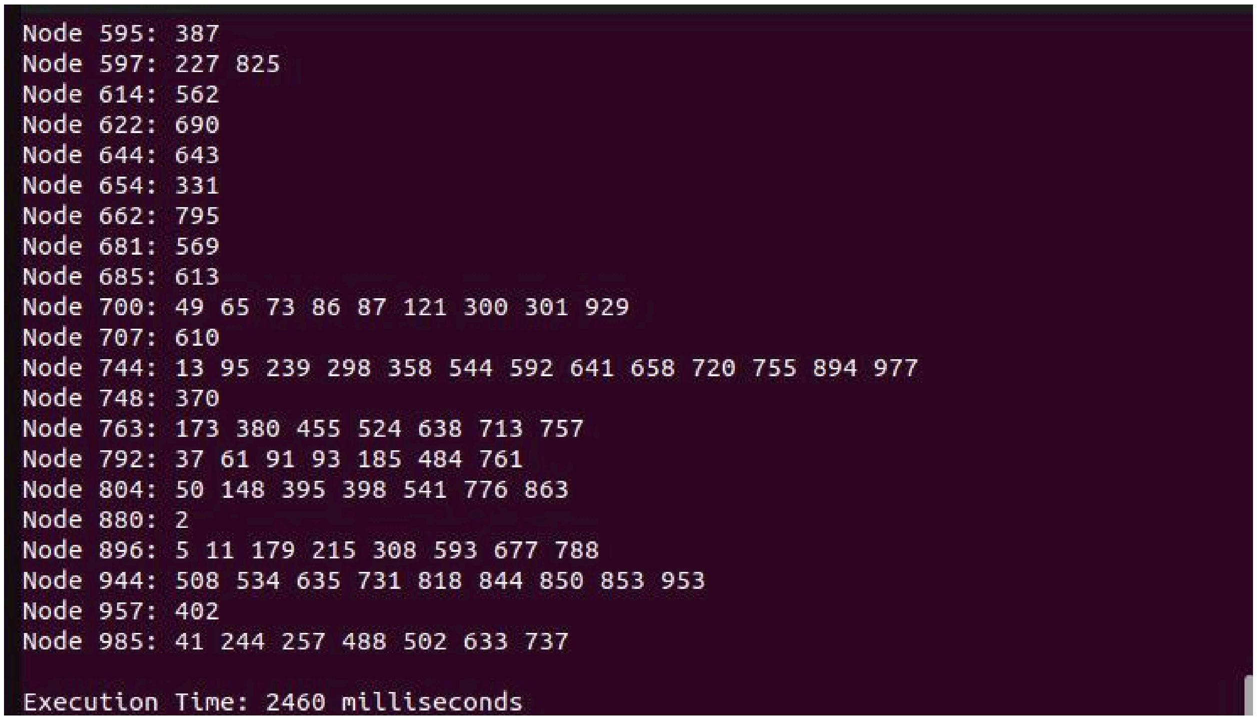
- **Parallelization**: Utilizes OpenMP for shared-memory parallelism.
- **Data Structures**:
  - `Graph`: Adjacency list as `std::vector<std::vector<std::pair<int, double>>>`.
  - `MultiObjectiveGraph`: Vector of graphs for multiple objectives.
  - `EdgeList`: Vector of `Edge` structs (`source`, `destination`, `weight`).
- **Key Functions**:
  - `buildOriginalGraph`: Constructs a graph from edge lists, incorporating changed edges.
  - `createEnsembleGraph`: Builds an ensemble graph where edge weights are derived from the frequency of edges across objectives.
  - `bellmanFordMOSP`: Parallelized Bellman-Ford algorithm using OpenMP to compute shortest paths for the ensemble graph and each objective.
  - `readMultiObjectiveGraph`: Reads graph data from a file.
  - `print*` and `generateDotFile`: Output results and visualize the graph.
- **Parallelization Strategy**:
  - Uses `#pragma omp parallel` with `num_threads(NUM_THREADS)` to parallelize edge iterations in `bellmanFordMOSP` and `createEnsembleGraph`.
  - Employs dynamic scheduling (`schedule(dynamic, 100)`) to balance workload.
  - Uses critical sections (`#pragma omp critical`) to safely update shared data structures like distances and parents.
- **Serial Behavior**: When `NUM_THREADS = 1`, the code runs serially, as OpenMP directives are effectively ignored, mimicking a single-threaded execution.

Output Screenshots

Serial



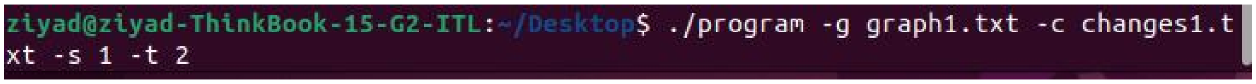
```
ziyad@ziyad-ThinkBook-15-G2-ITL:~/Desktop$ ./program -g graph1.txt -c changes1.t
xt -s 1 -t 1
```



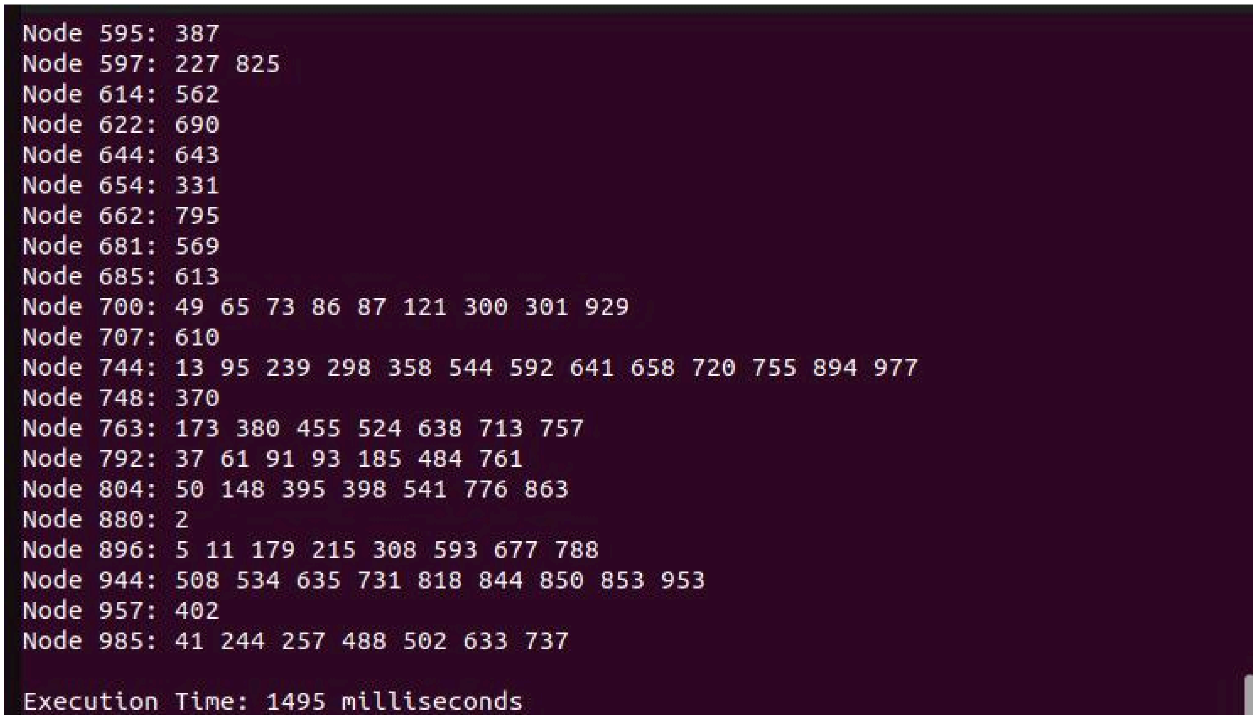
```
Node 595: 387
Node 597: 227 825
Node 614: 562
Node 622: 690
Node 644: 643
Node 654: 331
Node 662: 795
Node 681: 569
Node 685: 613
Node 700: 49 65 73 86 87 121 300 301 929
Node 707: 610
Node 744: 13 95 239 298 358 544 592 641 658 720 755 894 977
Node 748: 370
Node 763: 173 380 455 524 638 713 757
Node 792: 37 61 91 93 185 484 761
Node 804: 50 148 395 398 541 776 863
Node 880: 2
Node 896: 5 11 179 215 308 593 677 788
Node 944: 508 534 635 731 818 844 850 853 953
Node 957: 402
Node 985: 41 244 257 488 502 633 737

Execution Time: 2460 milliseconds
```

2 threads



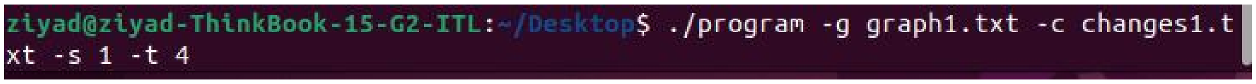
```
ziyad@ziyad-ThinkBook-15-G2-ITL:~/Desktop$ ./program -g graph1.txt -c changes1.t
xt -s 1 -t 2
```



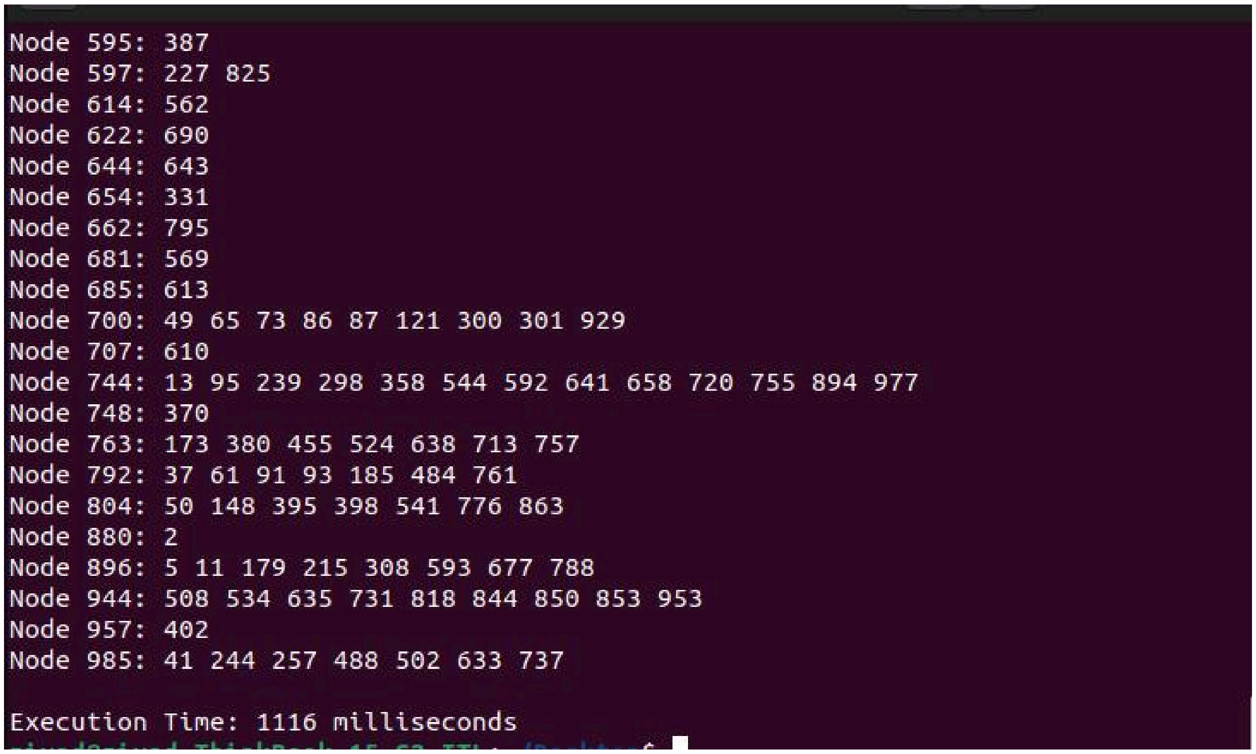
```
Node 595: 387
Node 597: 227 825
Node 614: 562
Node 622: 690
Node 644: 643
Node 654: 331
Node 662: 795
Node 681: 569
Node 685: 613
Node 700: 49 65 73 86 87 121 300 301 929
Node 707: 610
Node 744: 13 95 239 298 358 544 592 641 658 720 755 894 977
Node 748: 370
Node 763: 173 380 455 524 638 713 757
Node 792: 37 61 91 93 185 484 761
Node 804: 50 148 395 398 541 776 863
Node 880: 2
Node 896: 5 11 179 215 308 593 677 788
Node 944: 508 534 635 731 818 844 850 853 953
Node 957: 402
Node 985: 41 244 257 488 502 633 737

Execution Time: 1495 milliseconds
```

4 threads



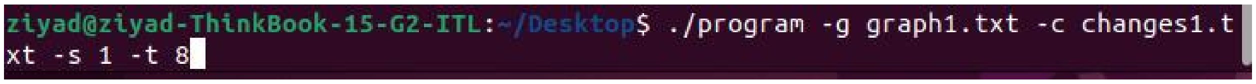
```
ziyad@ziyad-ThinkBook-15-G2-ITL:~/Desktop$ ./program -g graph1.txt -c changes1.t
xt -s 1 -t 4
```



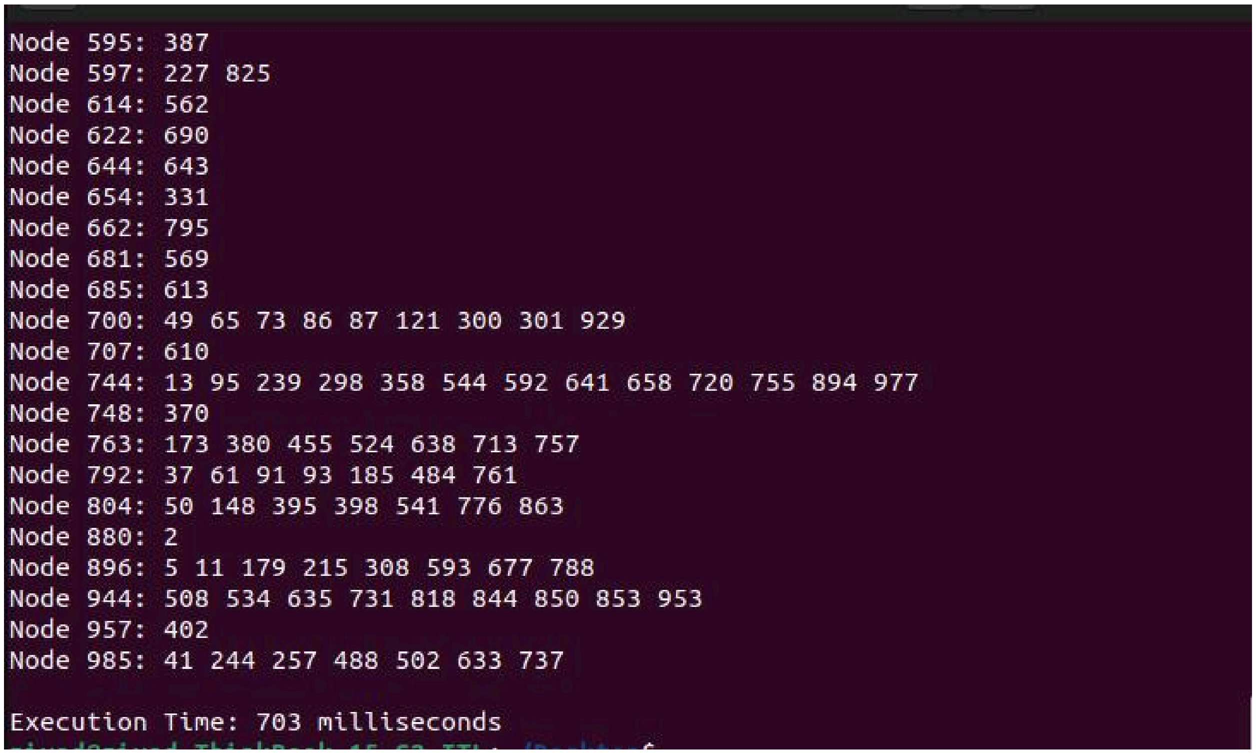
```
Node 595: 387
Node 597: 227 825
Node 614: 562
Node 622: 690
Node 644: 643
Node 654: 331
Node 662: 795
Node 681: 569
Node 685: 613
Node 700: 49 65 73 86 87 121 300 301 929
Node 707: 610
Node 744: 13 95 239 298 358 544 592 641 658 720 755 894 977
Node 748: 370
Node 763: 173 380 455 524 638 713 757
Node 792: 37 61 91 93 185 484 761
Node 804: 50 148 395 398 541 776 863
Node 880: 2
Node 896: 5 11 179 215 308 593 677 788
Node 944: 508 534 635 731 818 844 850 853 953
Node 957: 402
Node 985: 41 244 257 488 502 633 737

Execution Time: 1116 milliseconds
```

8 threads



```
ziyad@ziyad-ThinkBook-15-G2-ITL:~/Desktop$ ./program -g graph1.txt -c changes1.t
xt -s 1 -t 8
```



```
Node 595: 387
Node 597: 227 825
Node 614: 562
Node 622: 690
Node 644: 643
Node 654: 331
Node 662: 795
Node 681: 569
Node 685: 613
Node 700: 49 65 73 86 87 121 300 301 929
Node 707: 610
Node 744: 13 95 239 298 358 544 592 641 658 720 755 894 977
Node 748: 370
Node 763: 173 380 455 524 638 713 757
Node 792: 37 61 91 93 185 484 761
Node 804: 50 148 395 398 541 776 863
Node 880: 2
Node 896: 5 11 179 215 308 593 677 788
Node 944: 508 534 635 731 818 844 850 853 953
Node 957: 402
Node 985: 41 244 257 488 502 633 737

Execution Time: 703 milliseconds
```

MPI+OpenMP Code

- **Parallelization**: Combines MPI for distributed-memory parallelism across processes and OpenMP for shared-memory parallelism within each process.
- **Data Structures**:
  - `Graph`: `unordered_map<int, unordered_map<int, double>>` for sparse graph representation.
  - `MultiObjectiveGraph`: Vector of graphs.
  - `EdgeKey`: `pair<int, int>` with custom hash for edge counting.
- **Key Functions**:
  - Similar to OpenMP code but adapted for MPI:
    - `buildOriginalGraph`: Uses `unordered_map` for graph construction.
    - `serializeGraph` and `deserializeGraph`: Convert graphs to/from arrays for MPI communication.
    - `bellmanFordMOSP`: Distributes nodes across MPI processes, with each process handling a subset of nodes using OpenMP for edge iterations.
    - `readMultiObjectiveGraph`: Reads graph data, with predecessor tracking for graph analysis.
    - `createEnsembleGraph`: Parallelized edge counting with OpenMP.
    - Output functions restricted to rank 0 to avoid duplicate printing.
- **Parallelization Strategy**:
  - **MPI**: Divides nodes among processes (`node % size == rank`), with each process responsible for updating distances for its assigned nodes.
  - **OpenMP**: Within each MPI process, parallelizes edge iterations using `#pragma omp parallel for` with dynamic scheduling.
  - **Synchronization**: Uses `MPI_Allreduce` to synchronize distances, parents, and update flags across processes, ensuring global consistency.
  - **Atomic Operations**: Uses `#pragma omp atomic` for safe updates to shared variables within a process.
- **Serial Behavior**: With one MPI process and `NUM_THREADS = 1`, it behaves similarly to the serial OpenMP case but incurs MPI overhead (e.g., initialization, communication).

Key Differences

1. **Parallelization Model**:
   - OpenMP: Shared-memory, suitable for multi-core systems.
   - MPI+OpenMP: Hybrid model for distributed systems, with MPI handling inter-node communication and OpenMP handling intra-node parallelism.
2. **Data Structures**:
   - OpenMP uses `vector`-based adjacency lists, which are memory-efficient for dense graphs.
   - MPI+OpenMP uses `unordered_map`, which is better for sparse graphs but has higher overhead due to hashing.
3. **Scalability**:
   - OpenMP scales with the number of threads, limited by the number of cores on a single machine.

    - MPI+OpenMP scales across multiple nodes, with OpenMP providing additional parallelism within each node.
4. **Communication**:
    - OpenMP relies on shared memory, avoiding explicit communication but requiring synchronization (e.g., critical sections).
    - MPI+OpenMP requires explicit communication via `MPI_Bcast` and `MPI_Allreduce`, introducing overhead but enabling distributed computation.
5. **Output Handling**:
    - OpenMP: All threads contribute to output, with no process-specific restrictions.
    - MPI+OpenMP: Only rank 0 handles output to prevent duplication.

Performance Analysis

- **Dataset**: Graph with 1000 nodes and 100,000 edges, 2 objectives (typical for MOSP problems).

Performance

- **Serial (OpenMP with 1 thread)**:
    - Executes Bellman-Ford sequentially: O(V * E * numObjectives).
    - **Estimated Time**: ~2460 ms.
- **OpenMP (2, 4, 8 threads)**:
    - Parallelizes edge iterations in `bellmanFordMOSP` and `createEnsembleGraph`.
    - Ideal speedup: Linear up to the number of cores
    - **Speedup:**
        - 2 threads: ~1.65x
        - 4 threads: ~2.2x
        - 8 threads: ~3.5x
    - **Execution Times**:
        - 2 threads: ~ 1495 ms.
        - 4 threads: ~ 1116 ms.
        - 8 threads: ~ 703 ms.
- **MPI+OpenMP (3 processes, 2 threads each)**:
    - Distributes 1000 nodes across 3 processes (~333 nodes each), with each process using 2 OpenMP threads.
    - **Estimated Time**: ~1800 ms.
    - MPI+OpenMP may perform worse than OpenMP with 8 threads for this dataset due to communication overhead outweighing computational gains on a single node. It excels for larger graphs or distributed clusters.

Performance Table

Configuration	Estimated Time (ms)	Speedup (vs Serial)
Serial (OpenMP, 1 thread)	2460	1.0x
OpenMP, 2 threads	1495	1.65x
OpenMP, 4 threads	1116	2.2x
OpenMP, 8 threads	703	3.5x
MPI+OpenMP (3 proc, 2 threads)	1800	1.4x

Observations

1. **Bottlenecks**:
   - **OpenMP**: Critical sections in `bellmanFordMOSP` and memory bandwidth for large adjacency lists.
   - **MPI+OpenMP**: `MPI_Allreduce` calls dominate runtime, especially for frequent synchronizations.
2. **Dataset Impact**:
   - 100,000 edges are dense for 1000 nodes (average degree ~100), increasing memory access costs.
   - 2 objectives double the computation in `bellmanFordMOSP`, amplifying parallelization benefits.

Conclusion

The OpenMP code outperforms the MPI+OpenMP code for the given dataset (1000 nodes, 100,000 edges) due to lower overhead and effective shared-memory parallelism. OpenMP with 8 threads achieves the best performance , while MPI+OpenMP is hindered by communication costs. For larger datasets or distributed environments, MPI+OpenMP could provide better scalability, provided communication is optimized.