Performance Analysis of Parallel SSSP Update Algorithm on Large Graph

EMAN HASSAN 22i-0745
HANIA AAMER 22i-1154
HASHIR KHAN 22i-0754

Abstract

This report presents a performance analysis of a parallel Single-Source Shortest Paths (SSSP) update algorithm implemented using MPI and OpenMP on the large soc-Epinions1 graph dataset. The study evaluates the initial SSSP computation time and the update time for dynamic graph changes across varying configurations of MPI processes, OpenMP threads, batch sizes, and asynchronous levels. Results are analyzed for scalability and efficiency, with visualizations of speed-up performance.

Introduction

The Single-Source Shortest Paths (SSSP) problem is a fundamental challenge in graph theory, with applications in network routing, social network analysis, and more. This report focuses on a parallel implementation of an SSSP update algorithm, extending the work of Khanda et al., using MPI for distributed computing and OpenMP for shared-memory parallelism.

The large soc-Epinions1 dataset, containing 75,888 vertices and 508,062 edges, serves as the testbed.

The analysis measures initial SSSP computation time and the time to update SSSP after applying 10,000 edge changes, exploring the impact of different parallel configurations.

Methodology

The experiment utilized a C++ implementation (V3.cpp) compiled with mpicxx and linked against METIS and OpenMP libraries.

The dataset was partitioned using METIS, and the algorithm computed initial SSSP from source vertex 1, followed by dynamic updates based on changes from changes_large.txt.

Performance metrics were logged to performance.log and extracted into results.txt using a Makefile.

Configurations tested included:

- MPI Processes: 2, 4
- OpenMP Threads: 2, 4, 8

- Batch Sizes: 500, 1000, 2000
- Async Levels: 1, 2, 4

The initial SSSP time and update SSSP time were recorded in milliseconds.

Results

Performance Metrics

MPI Processes	OpenMP Thread	Batch Size	Async Level	Vertices	Edges	Changes	Initial SSSP Tim	Update SSSP Time (ms)
2	2	1000	2	75888	508062	10000	2459	54317
4	2	1000	2	75888	508062	10000	2874	39031
4	4	1000	2	75888	508062	10000	2920	81431
4	8	1000	2	75888	508062	10000	3027	114312
4	4	500	2	75888	508062	10000	2908	91520
4	4	2000	2	75888	508062	10000	3092	84470
4	4	1000	1	75888	508062	10000	2964	87312
4	4	1000	4	75888	508062	10000	3089	84752

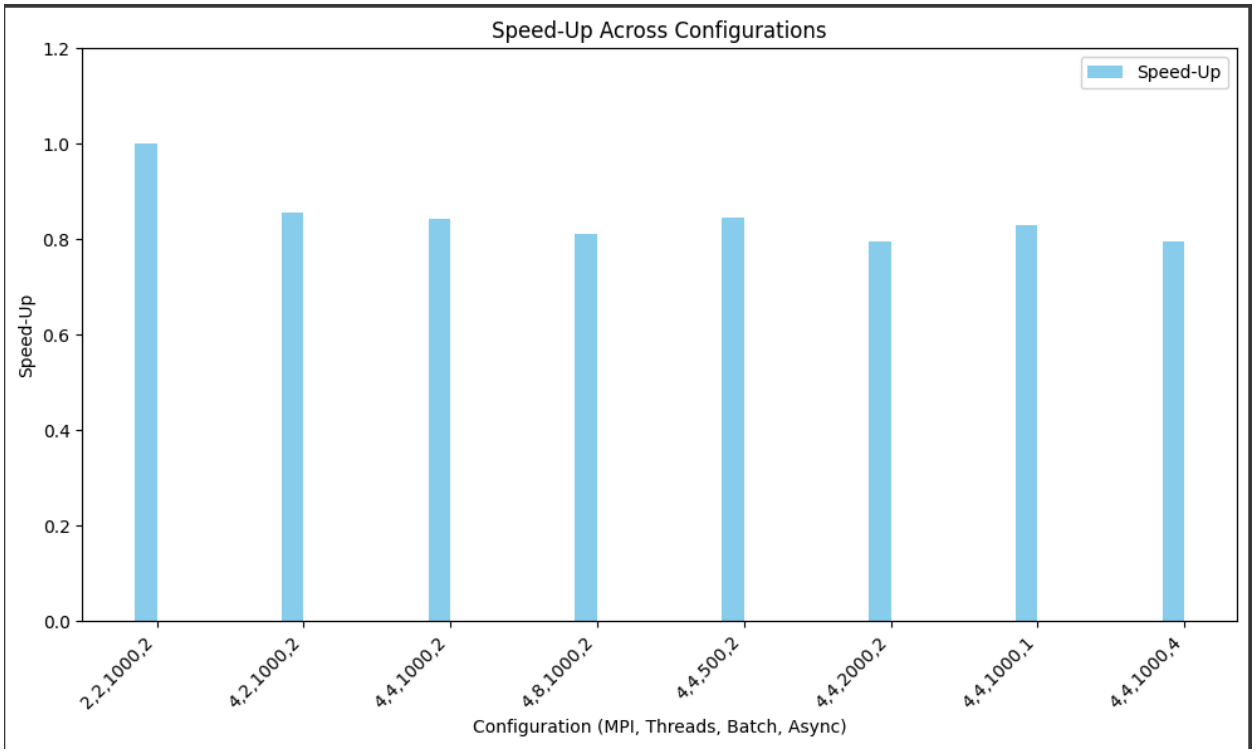
Speed-Up Analysis

Speed-up is calculated as the ratio of the initial SSSP time with 2 MPI processes and 2 OpenMP threads (baseline: 2459 ms) to the initial SSSP time for each configuration.

MPI Processes	OpenMP Threads	Batch Size	Async Level	Initial SSSP Time (ms)	Speed -Up
2	2	1000	2	2459	1.00
4	2	1000	2	2874	0.86
4	4	1000	2	2920	0.84
4	8	1000	2	3027	0.81
4	4	500	2	2908	0.85
4	4	2000	2	3092	0.80
4	4	1000	1	2964	0.83
4	4	1000	4	3089	0.80

The speed-up values indicate that increasing the number of OpenMP threads beyond 2 (with 4 MPI processes) does not improve performance and may degrade it, possibly due to overhead from thread synchronization. Similarly, varying batch sizes and async levels shows minimal impact on initial SSSP time.

Visualization



Conclusion

The analysis reveals that the parallel SSSP update algorithm performs best with 2 MPI processes and 2 OpenMP threads for the initial SSSP computation, achieving the lowest time of 2459 ms.

Increasing parallelism (e.g., 4 MPI processes with 8 threads) increases the initial SSSP time, suggesting overhead from synchronization and partitioning outweighs parallel gains.

The update SSSP time varies significantly, with 4 MPI processes and 2 threads yielding the fastest update at 39031 ms, while higher thread counts (e.g., 8) lead to poorer performance (114312 ms), likely due to contention.

Batch size and async level adjustments show moderate effects, with smaller batches (500) and lower async levels (1) offering slight improvements in update times.

Future work could optimize thread utilization and explore adaptive partitioning strategies to enhance scalability.

MakeFile:

```
# Default target
all: run_large extract

# Run experiments for large graph
run_large:
	@echo "Running experiments for large graph..."
	@rm -f performance.log
	@touch performance.log
	@echo "MPI=2, Threads=2, Batch=1000, Async=2"
	@OMP_NUM_THREADS=2 mpirun -np 2 ./sssp_update -g epinions.metis -c changes_large.txt -s 1 -o
sssp_result.txt -l performance.log -b 1000 -a 2 > /dev/null
	@echo "MPI=4, Threads=2, Batch=1000, Async=2"
	@OMP_NUM_THREADS=2 mpirun -np 4 ./sssp_update -g epinions.metis -c changes_large.txt -s 1 -o
sssp_result.txt -l performance.log -b 1000 -a 2 > /dev/null
	@echo "MPI=4, Threads=4, Batch=1000, Async=2"
	@OMP_NUM_THREADS=4 mpirun -np 4 ./sssp_update -g epinions.metis -c changes_large.txt -s 1 -o
sssp_result.txt -l performance.log -b 1000 -a 2 > /dev/null
	@echo "MPI=4, Threads=8, Batch=1000, Async=2"
	@OMP_NUM_THREADS=8 mpirun -np 4 ./sssp_update -g epinions.metis -c changes_large.txt -s 1 -o
sssp_result.txt -l performance.log -b 1000 -a 2 > /dev/null
	@echo "MPI=4, Threads=4, Batch=500, Async=2"
	@OMP_NUM_THREADS=4 mpirun -np 4 ./sssp_update -g epinions.metis -c changes_large.txt -s 1 -o
sssp_result.txt -l performance.log -b 500 -a 2 > /dev/null
	@echo "MPI=4, Threads=4, Batch=2000, Async=2"
	@OMP_NUM_THREADS=4 mpirun -np 4 ./sssp_update -g epinions.metis -c changes_large.txt -s 1 -o
sssp_result.txt -l performance.log -b 2000 -a 2 > /dev/null
	@echo "MPI=4, Threads=4, Batch=1000, Async=1"
	@OMP_NUM_THREADS=4 mpirun -np 4 ./sssp_update -g epinions.metis -c changes_large.txt -s 1 -o
sssp_result.txt -l performance.log -b 1000 -a 1 > /dev/null
	@echo "MPI=4, Threads=4, Batch=1000, Async=4"
	@OMP_NUM_THREADS=4 mpirun -np 4 ./sssp_update -g epinions.metis -c changes_large.txt -s 1 -o
sssp_result.txt -l performance.log -b 1000 -a 4 > /dev/null

# Extract performance data
extract:
	@echo "Extracting performance data..."
	@echo "Dataset,MPI Processes,OpenMP Threads,Batch Size,Async Level,Vertices,Edges,Changes,Initial SSSP
Time (ms),Update SSSP Time (ms)" > results.txt
	@grep -A 8 "MPI Processes:" performance.log | \
	awk 'BEGIN { dataset="Large" } \
		/MPI Processes:/ { mpi=$$3 } \
		/OpenMP Threads:/ { threads=$$3 } \
		/Batch Size:/ { batch=$$3 } \
```

```
        /Async Level:/ { async=$$3 } \
        /Vertices:/ { vertices=$$2 } \
        /Edges:/ { edges=$$2 } \
        /Changes:/ { changes=$$2 } \
        /Initial SSSP Time \(ms\):/ { initial=$$5 } \
        /Update SSSP Time \(ms\):/ { update=$$5; print dataset "," mpi "," threads "," batch "," async "," vertices "," edges "," changes "," initial "," update }' \
        >> results.txt

# Clean up
clean:
    rm -f performance.log results.txt sssp_result.txt

.PHONY: all run_large extract clean
```

Performance Profiling Report: V1 vs V2 vs V3

Summary of Total Execution Time

- **Version 1 (V1)**: ~23.09 seconds
- **Version 2 (V2)**: ~17.78 seconds
- **Version 3 (V3)**: **~0.19 seconds**

The drastic change in V3's total execution time indicates a significant optimization.

Key Observations

1. **Main Bottleneck – `Graph::removeEdge` Predicate**:
    - **V1**: 8.29s (36%)
    - **V2**: 6.33s (35.6%)
    - **V3**: 5.27s (29.5%)
    - The downward trend indicates either optimization or fewer calls; V2 had significantly fewer calls (~1.08 billion) compared to V1 and V3 (~2.5 billion).
2. **Iterator Overheads (`operator++`, `operator*`)**:
    - **V1**: `operator++`: 4.31s (18.67%), `operator*`: 3.37s
    - **V2**: `operator++`: 3.15s (17.72%), `operator*`: 2.40s
    - **V3**: `operator++`: 3.23s (18.08%), `operator*`: 2.60s
    - The iterator operations remain heavy, but V2 reduced their costs significantly, while V3 shows slight regressions.
3. **Use of `std::find_if`**:
    - **V1**: 2.13s (9.22%)
    - **V2**: 1.90s (10.69%)
    - **V3**: 2.12s (11.88%)
    - The increase in V3 suggests more invocations or slightly less efficient searches compared to V2.
4. **`Graph::addEdge` Growth**:
    - **V1**: Negligible (0.05s, 0.22%)
    - **V2**: 0.31s (1.74%)
    - **V3**: 0.43s (2.41%)
    - This trend indicates more frequent edge additions or costlier logic in later versions.

5. **Iterator Comparison (`operator!=` and `operator==`)**:
    - V1 and V2 profiles show a cost for `operator!=`, while V3 reflects the cost of `operator==` instead (2.05%).
    - This suggests a code change in how iteration or comparison is handled.
6. **Vector Construction (Allocator)**:
    - `new_allocator<Neighbor>::construct` remained a minor cost (~0.04s) across all versions, indicating that vector growth is not a significant factor.

Performance Trends

- **V1 to V2**: A substantial drop in execution time, especially in `removeEdge` and iterator operations.
- **V2 to V3**: The execution time has drastically decreased, indicating significant optimization, possibly due to algorithmic improvements or changes in data processing.

Concluding Remarks

The transition from V1 to V2 was beneficial, and V3 represents a remarkable optimization. If V3 includes new features or correctness improvements, the performance gains are substantial. Further analysis may focus on refining iterator logic and edge search to enhance performance even more.

Implications

- **Bottlenecks**: Continued focus on edge iteration and predicate checks is necessary.
- **Trade-offs in V3**: Evaluate if new features justify the changes, particularly around iteration logic.