15-418 Final Project Report

Connor Mowry (cmowry), Wenqi Deng (wenqid)

December 14, 2024

1 Title

Parallelizing Dinic's Max-Flow Algorithm.

Connor Mowry (cmowry), Wenqi Deng (wenqid)

2 URL

`https://www.andrew.cmu.edu/user/cmowry/418-final-project-website`

3 Link to Git Repository

`https://github.com/wenqiden/parallelized_dinics`

4 Summary

This project focused on parallelizing Dinic's max-flow algorithm to improve its performance on both BFS and DFS phases. We leveraged OpenMP for shared memory systems and MPI for distributed memory systems, exploring various strategies to enhance scalability and efficiency.

5 Background

Dinic's max-flow algorithm is a well-known method in graph theory for solving maximum flow problems. It alternates between constructing a level graph using Breadth-First Search (BFS) and finding blocking flows using Depth-First Search (DFS). The BFS phase restricts DFS to only valid augmenting paths, improving computational efficiency.

Key Data Structures

- **Adjacency List/Matrix:** Represents the graph structure, where edges and vertices are stored.
- **Residual Graph:** Tracks remaining capacities on edges after flow is pushed.

Algorithm Overview

The algorithm operates in phases:

1. **Level Graph Construction (BFS):** This step identifies levels of the graph to limit DFS exploration to valid augmenting paths. BFS operates by traversing the graph layer by layer, marking vertices with their respective distances from the source.
2. **Blocking Flow Computation (DFS):** DFS is performed on the level graph to find augmenting paths and push flow along these paths until no further augmenting path exists.

Parallelization Potential

BFS:

- BFS is inherently data-parallel as vertices and edges can be processed independently, making it suitable for parallelization.
- Optimizing BFS involves balancing workload across threads or processes to avoid bottlenecks.

DFS:

- Unlike BFS, DFS phase in Dinic's algorithm suffers from dependencies due to its reliance on the state of residual capacities. Synchronization mechanisms (e.g., locks) are necessary to manage these dependencies.

Applications

Dinic's algorithm is widely used in fields such as:

- **Network Optimization:** Solving routing and bandwidth allocation problems.
- **Image Segmentation:** Identifying regions in images based on pixel connectivity.
- **Supply Chain Management:** Optimizing resource distribution in logistics networks.

Diagrams illustrating the BFS and DFS processes, as well as 1D and 2D decomposition strategies, will be included to enhance clarity and provide context.

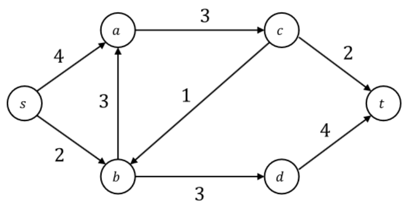


Figure 1: Max-Flow Example

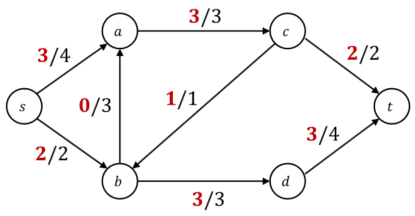


Figure 2: Solution to 1

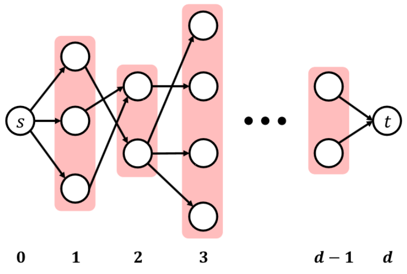


Figure 3: Example level graph produced by BFS

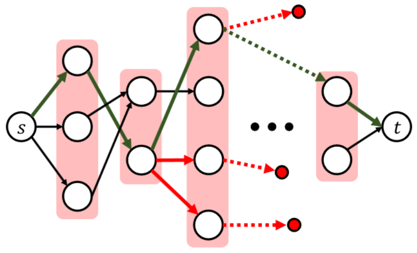


Figure 4: Example of DFS on the level graph

6 Approach

Sequential Implementation

We began by implementing Dinic's algorithm sequentially, closely following established pseudocode to ensure correctness. This implementation served as the baseline for all subsequent parallel versions.

OpenMP Implementations

1. **BFS Parallelization:**
   - **By Vertex:** Each thread processes a subset of vertices in the frontier, checking their neighbors and updating the next level. This approach is simple and works well for graphs with dense edges and relatively uniform vertex degrees.
   - **By Edge:** Threads iterate over edges in the frontier, which minimizes load imbalance by distributing work more evenly when vertex degrees vary widely. This method is particularly effective for Erdős–Rényi graphs as they have a large number of edges.

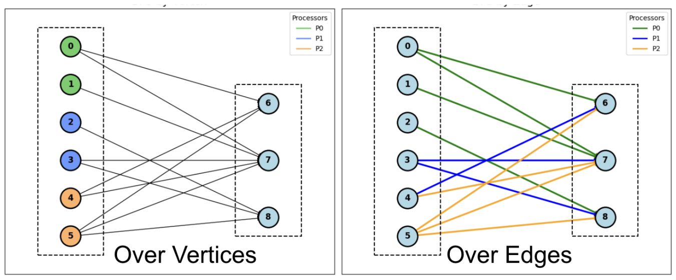


Figure 5: OpenMP BFS By Vertex vs By Edge

2. **DFS Parallelization:**
    - **Forward Lock:** Threads lock edges as they traverse paths, preventing conflicts during capacity updates. Locks are released after successfully pushing flow along a path.
    - **Reverse Lock:** After reaching the sink, threads lock edges along the path in reverse, compute the bottleneck capacity, and update residual capacities. This approach allows partial progress even if other threads modify capacities concurrently.

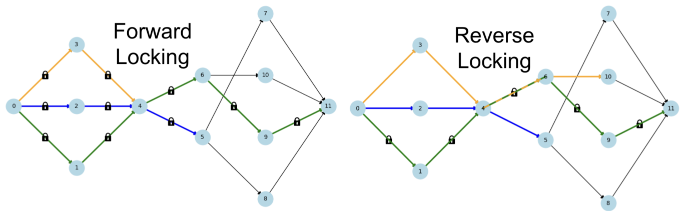


Figure 6: Forward Locking vs Reverse Locking

MPI Implementations

1. **BFS Parallelization**:
    - **1D Partition:** Vertices are distributed across processes, with each process handling a subset of the graph. Synchronization is achieved using MPI All-to-All to share information about the next frontier among processes.
        - **DFS Implementation Tradeoff:** Since processors need knowledge of flows from DFS phase to do parallel BFS, we have two options: either distribute the entire graph to all processors and let each run DFS independently, or distribute only the relevant

nodes to each processor, have the root processor perform the DFS, and then share the resulting flow with all processors.

(a) **Single-Processor DFS:** Running the DFS on one processor required scattering the residual capacity matrix to all processors after the DFS is completed. This introduced communication overhead but minimized memory usage.

(b) **All-Processor DFS:** Running the DFS redundantly on all processors eliminated the need for communication but required each processor to maintain a full copy of the graph, significantly increasing memory requirements. This might also increase memory access overhead.

After testing both approaches, we determined that the single-processor version of DFS was more efficient. This method reduced overall memory demands and maintained acceptable communication overhead, making it the preferred choice for our implementation, despite potential communication bottlenecks in dense graphs.

- **2D Partition:** The adjacency matrix is divided into blocks, assigning both rows (vertices) and columns (edges) to processes. Processes communicate only with those sharing a row or column, significantly reducing communication overhead compared to 1D partitioning. This strategy also balances memory usage more effectively across processes. See figure 7 for difference between 1D and 2D partitioning.

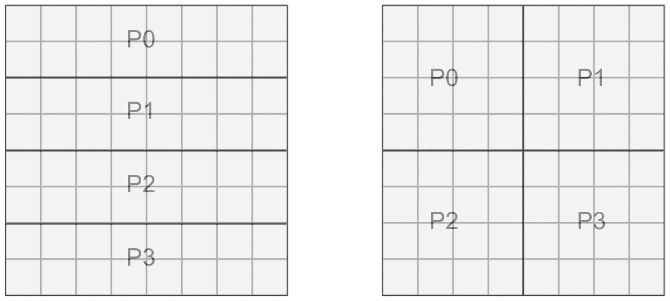


Figure 7: 1D vs 2D Graph Partitioning

2. **DFS Parallelization**: Due to the difficulty of overcoming data dependency in distributed memory systems and the lack of performance gains from parallel DFS using OpenMP over the sequential version, we decided to focus more on parallelizing BFS using MPI and run DFS in a single thread.

Optimizations

To improve performance, we applied several optimization techniques:

- **Task Granularity:** Adjusted granularity in edge-based BFS to balance thread workloads.
- **Reduction Operations:** Used reductions to merge results from threads efficiently.
- **Synchronization Minimization:** Minimized locking in DFS to reduce contention.
- **Communication Strategies:** Explored different MPI communication patterns to optimize frontier updates.

These implementations were iteratively tested and refined to achieve optimal performance for different graph types and sizes.

7 Challenge

The project faced several challenges in parallelizing both BFS and DFS phases of Dinic's algorithm. These challenges stemmed from the fundamental nature of the algorithm and the constraints of parallel computing.

BFS Challenges

- **Load Imbalance:** Graphs with uneven degree distributions, such as Barabási–Albert graphs, posed significant difficulties. To address this, we implemented two versions of BFS in OpenMP. The first parallelized over vertices in the frontier, while the second is parallelized over edges in the frontier to minimize load imbalance. The edge-based approach particularly helped balance workloads in graphs with high-degree vertices, but could potentially increase the scheduling and management overheads.
- **Communication Overhead in MPI:** In distributed memory systems, synchronizing frontier updates across processes incurred substantial communication costs, especially for large graphs. Reducing this overhead without compromising correctness was a critical challenge.

DFS Challenges

- **Data Dependencies:** DFS inherently relies on the state of residual capacities. Concurrent updates to these capacities by multiple threads or processes led to potential conflicts, requiring synchronization mechanisms.
- **Synchronization Costs:** Mechanisms such as edge locking introduced overhead, limiting scalability. Implementations like forward locking and reverse locking reduced conflicts but added their own performance tradeoffs.

MPI-Specific Challenges

- **1D vs. 2D Decomposition:** In MPI, we can parallelize BFS by partitioning parts of the adjacency matrix for different processes to handle. We can divide the graph by nodes (1D partition) or both nodes and edges within single nodes (2D partition). While 2D decomposition offered better load balancing and reduced memory footprints, its implementation required more complex communication patterns. Identifying the optimal decomposition strategy was non-trivial.

General Challenges

- **Graph Representation:** Choosing a graph representation that facilitated both efficient computation and effective parallelization was a recurring challenge. Adjacency lists were memory-efficient but required more complex access patterns, while adjacency matrices were simpler to parallelize but memory-intensive.

8 Results

We measured the computation time for calculating the max flow in networks of varying models and sizes, using both GHC and PSC machines. The time will be recorded in milliseconds. OOM stands for out of memory.

On GHC machines:

1. Small Erdős–Rényi Graph (n=128, e=15481)

	1	2	4	8
Sequential	0.208	NA	NA	NA
OpenMP BFS vertex	0.214	0.238	0.262	5.48
OpenMP BFS edge	0.408	0.403	0.301	0.289
OpenMP DFS forward lock	0.352	0.398	0.337	0.393
OpenMP DFS reverse lock	3.40	2.41	1.81	1.66
MPI BFS 1D Non-distributed	8.41	0.745	0.47	0.586
MPI BFS 1D Distributed	1.58	0.859	1.50	0.585
MPI BFS 2D Distributed	7.56	5.01	0.824	0.638

2. Small Barabási–Albert Graph (n=128, e=1694)

	1	2	4	8
Sequential	0.103	NA	NA	NA
OpenMP BFS vertex	0.153	0.311	0.576	5.62
OpenMP BFS edge	1.13	0.234	1.08	3.84
OpenMP DFS forward lock	0.033	0.249	0.450	0.147
OpenMP DFS reverse lock	0.148	0.218	1.01	5.53
MPI BFS 1D Non-distributed	0.553	0.598	0.173	0.381
MPI BFS 1D Distributed	0.559	0.113	0.723	0.258
MPI BFS 2D Distributed	1.49	1.38	0.294	0.414

3. Small Grid Graph (n=121, e=440)

	1	2	4	8
Sequential	0.051	NA	NA	NA
OpenMP BFS vertex	0.156	0.534	1.74	7.12
OpenMP BFS edge	1.36	1.92	3.76	0.793
OpenMP DFS forward lock	0.090	0.296	0.787	0.218
OpenMP DFS reverse lock	0.366	1.02	2.00	0.495
MPI BFS 1D Non-distributed	0.955	1.64	0.487	1.94
MPI BFS 1D Distributed	0.926	0.334	2.449	0.881
MPI BFS 2D Distributed	4.42	5.21	4.93	1.41

4. Medium Erdős–Rényi Graph (n=2048, e=3983314)

	1	2	4	8
Sequential	67.8	NA	NA	NA
OpenMP BFS vertex	66.2	51.9	48.3	47.4
OpenMP BFS edge	124.1	71.1	44.5	31.5
OpenMP DFS forward lock	160.5	116.0	92.4	83.6
OpenMP DFS reverse lock	1676.7	1674.6	1201.6	1116.1
MPI BFS 1D Non-distributed	4608.9	1291.4	408.7	192.8
MPI BFS 1D Distributed	4582.9	1281.6	396.9	154.9
MPI BFS 2D Distributed	429.3	248.9	160.0	104.5

5. Medium Barabási–Albert Graph (n=2048, e=400670)

	1	2	4	8
Sequential	4.61	NA	NA	NA
OpenMP BFS vertex	4.62	4.56	5.69	3.84
OpenMP BFS edge	59.0	37.2	22.4	16.1
OpenMP DFS forward lock	11.2	8.89	7.51	7.16
OpenMP DFS reverse lock	55.5	39.4	31.9	30.4
MPI BFS 1D Non-distributed	136.8	76.1	41.4	34.8
MPI BFS 1D Distributed	130.4	70.6	39.4	24.6
MPI BFS 2D Distributed	49.8	36.1	28.6	24.5

6. Medium Grid Graph (n=2025, e=7920)

	1	2	4	8
Sequential	0.045	NA	NA	NA
OpenMP BFS vertex	0.133	0.396	0.415	0.702
OpenMP BFS edge	28.4	16.4	10.4	7.30
OpenMP DFS forward lock	0.128	0.216	0.222	0.255
OpenMP DFS reverse lock	0.644	0.577	0.538	0.621
MPI BFS 1D Non-distributed	1.47	1.67	0.491	1.59
MPI BFS 1D Distributed	1.52	0.348	0.376	0.757
MPI BFS 2D Distributed	13.9	11.9	8.10	6.71

7. Large Erdős–Rényi Graph (n=4096, e=15934812)

	1	2	4	8
Sequential	232.5	NA	NA	NA
OpenMP BFS vertex	231.2	183.1	173.9	167.9
OpenMP BFS edge	388.1	228.3	147.1	103.8
OpenMP DFS forward lock	608.6	416.3	318.1	281.1
OpenMP DFS reverse lock	7560	6613	6442	6804
MPI BFS 1D Non-distributed	26928	7129	2097	861.1
MPI BFS 1D Distributed	26852	7119	2064	718.2
MPI BFS 2D Distributed	1423	768.9	515.5	362.5

8. Large Barabási–Albert Graph (n=4096, e=1595310)

	1	2	4	8
Sequential	22.6	NA	NA	NA
OpenMP BFS vertex	22.8	18.3	17.1	17.2
OpenMP BFS edge	335.5	190.8	116.5	77.0
OpenMP DFS forward lock	44.8	37.2	32.5	32.2
OpenMP DFS reverse lock	304.7	246.0	189.5	188.3
MPI BFS 1D Non-distributed	1479.4	767.2	418.1	246.1
MPI BFS 1D Distributed	1472.8	761.2	409.0	228.9
MPI BFS 2D Distributed	269.8	186.0	142.0	114.5

9. Large Grid Graph (n=4096, e=16128)

	1	2	4	8
Sequential	0.07	NA	NA	NA
OpenMP BFS vertex	0.193	0.430	0.478	0.636
OpenMP BFS edge	114.7	66.0	41.3	28.3
OpenMP DFS forward lock	0.140	0.229	0.178	0.217
OpenMP DFS reverse lock	0.836	0.827	0.917	0.983
MPI BFS 1D Non-distributed	2.94	2.54	0.527	2.38
MPI BFS 1D Distributed	2.987	0.519	0.523	0.888
MPI BFS 2D Distributed	53.5	39.9	30.1	23.9

Since the below test cases exceed the AFS quota on GHC machines, we did benchmarking on PSC machines:

1. Extreme Erdős–Rényi Graph (n=8192, e=63743938)

	1	2	4	8	16	32	64	128
Sequential	1284.5	NA	NA	NA	NA	NA	NA	NA
OpenMP BFS vertex	1348.2	931.3	793.8	733.3	793.4	810.5	823.8	854.8
OpenMP BFS edge	2025.3	1413.8	1223.9	817.4	879.7	954.8	771.3	783.1
OpenMP DFS forward lock	3535.5	2460.7	1841.3	1573.2	1893.9	1839.9	1791.6	1762.7
OpenMP DFS reverse lock	47954	34909	31844	27034	25403	86543	86407	149029
MPI BFS 1D Non-distributed	326551	84177	22876	7115	2971	2097	4251	OOM
MPI BFS 1D Distributed	325071	83085	21983	6502	2390	1288	1132	1162.6
MPI BFS 2D Distributed	10258	5478	3442	2185	1583	1174	1250	1644.4

2. Extreme Barabási–Albert Graph (n=8192, e=6381240)

	1	2	4	8	16	32	64	128
Sequential	160.1	NA	NA	NA	NA	NA	NA	NA
OpenMP BFS vertex	158.6	121.5	103.0	100.1	124.4	149.5	151.4	120.4
OpenMP BFS edge	1753.4	1383.5	986.8	612.3	591.2	611.8	400.2	426.4
OpenMP DFS forward lock	347.2	270.4	226.3	210.6	259.4	303.2	297.5	221.5
OpenMP DFS reverse lock	3247.1	2640.3	2675.7	3345.8	3106.4	5328.0	9336.0	16240
MPI BFS 1D Non-distributed	18028	9089	4690	2545	1443	649.8	483.2	419.7
MPI BFS 1D Distributed	17826	8926	4561	2427	1334	529.2	273.5	226.5
MPI BFS 2D Distributed	1809	1306	976.8	821.7	704.1	652.6	657.9	656.9

3. Extreme Grid Graph (n=8190, e=32398)

	1	2	4	8	16	32	64	128
Sequential	0.453	NA	NA	NA	NA	NA	NA	NA
OpenMP BFS vertex	0.735	2.509	2.55	0.828	2.31	8.83	9.27	407.6
OpenMP BFS edge	772.3	826.2	494.5	313.2	334.7	339.3	227.5	368.6
OpenMP DFS forward lock	1.62	2.94	3.53	3.77	4.34	5.89	6.78	53.5
OpenMP DFS reverse lock	5.83	8.53	10.2	11.2	15.6	16.8	16.7	38.6
MPI BFS 1D Non-distributed	2.86	2.48	2.05	2.74	3.54	6.51	15.8	31.1
MPI BFS 1D Distributed	2.90	3.78	4.64	7.45	12.3	25.5	15.4	31.5
MPI BFS 2D Distributed	479.7	354.9	296.04	277.8	263.3	264.6	283.2	309.8

4. Impossible Erdős–Rényi Graph (n=12040, e=99609033)

	1	2	4	8	16	32	64	128
Sequential	1764.9	NA	NA	NA	NA	NA	NA	NA
OpenMP BFS vertex	1825.5	1391.2	1194.6	1114.2	1168.2	1204.2	1195.7	1490.0
OpenMP BFS edge	2546.9	1775.5	1564.7	1060.5	1105.2	1325.7	1124.0	1041.7
OpenMP DFS forward lock	5272.2	3586.0	2567.6	2033.0	2214.6	2219.8	2063.5	5729.1
OpenMP DFS reverse lock	80849	57050	51330	59283.9	42305	121861	143007	170746
MPI BFS 1D Non-distributed	478337	122531	33182	10394	4432	5350	OOM	OOM
MPI BFS 1D Distributed	475758	120788	31946	9508	3515	1951	1748	1677.9
MPI BFS 2D Distributed	12342	6700	4323	2859	2138	1668	1716	1586.9

5. Impossible Barabási–Albert Graph (n=12040, e=9961472)

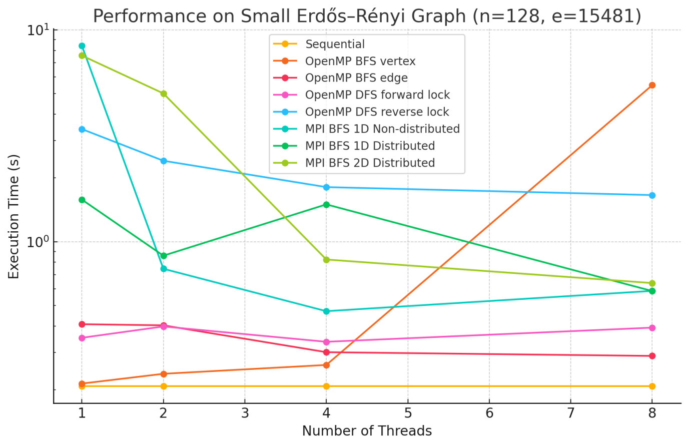
	1	2	4	8	16	32	64	128
Sequential	250.1	NA	NA	NA	NA	NA	NA	NA
OpenMP BFS vertex	241.0	183.3	151.2	149.1	171.5	201.1	200.2	172.2
OpenMP BFS edge	2774.0	2371.1	1732.0	934.1	933.0	946.5	626.4	499.2
OpenMP DFS forward lock	574.0	409.0	342.1	309.7	387.1	425.6	451.9	299.3
OpenMP DFS reverse lock	5254.4	4784.1	4456.0	4732.6	5173.3	10112	8030.3	18067.8
MPI BFS 1D Non-distributed	34844	17476	8974.4	4835.9	2652.5	1095.4	727.1	562.7
MPI BFS 1D Distributed	34647	17229	8775.56	4593.7	2493.6	956.1	455.4	320.6
MPI BFS 2D Distributed	2838.0	1944.3	1492.4	1273.7	1101.2	1022.1	1024.9	1029.2

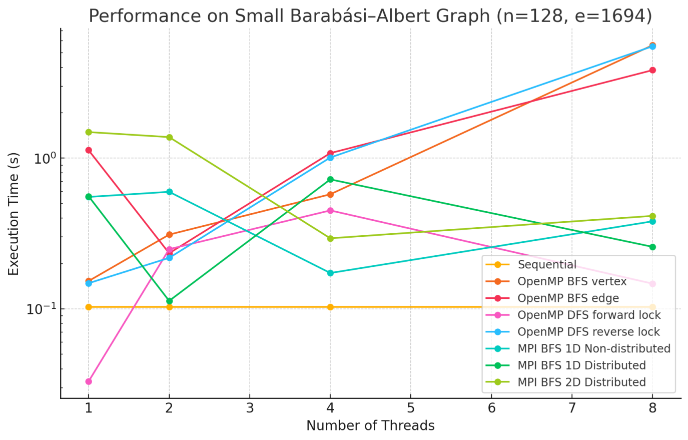
6. Impossible Grid Graph (n=10201, e=40400)

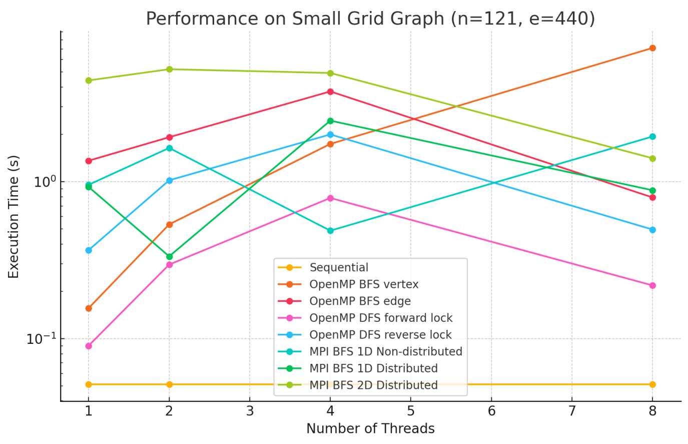
	1	2	4	8	16	32	64	128
Sequential	0.630	NA	NA	NA	NA	NA	NA	NA
OpenMP BFS vertex	0.914	6.58	4.21	4.65	2.57	8.68	5.57	181.4
OpenMP BFS edge	1152.7	884.8	853.5	515.0	485.0	498.8	326.3	395.95
OpenMP DFS forward lock	2.17	3.97	5.87	6.31	5.79	7.42	8.00	55.0
OpenMP DFS reverse lock	6.80	10.5	15.1	20.0	21.4	21.6	13.45	58.2
MPI BFS 1D Non-distributed	3.73	3.42	2.57	3.21	4.01	7.37	16.3	37.9
MPI BFS 1D Distributed	3.82	4.54	5.10	7.90	12.9	26.0	16.2	33.2
MPI BFS 2D Distributed	719.3	526.6	430.5	422.3	428.6	441.2	417.2	447.5

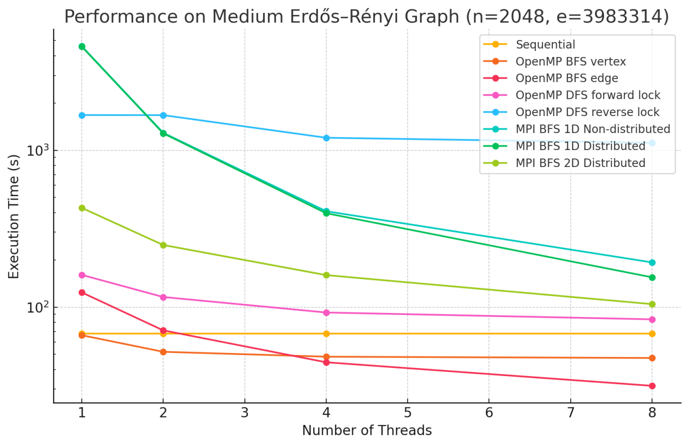
9 Performance Graphs

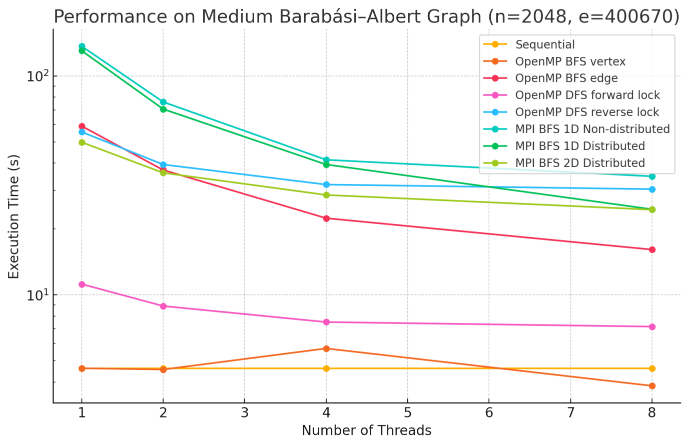
GHC Benchmarks

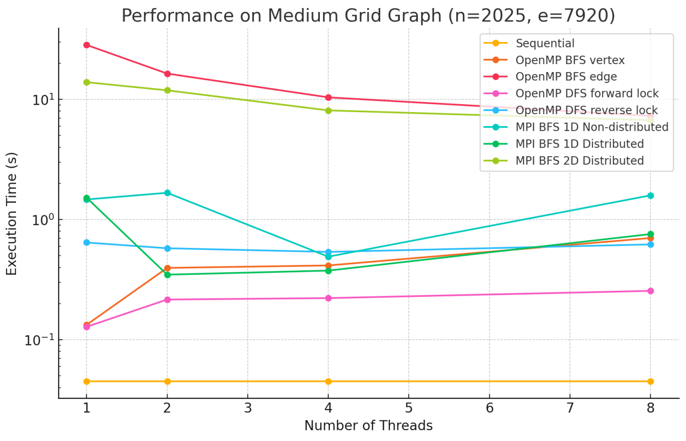


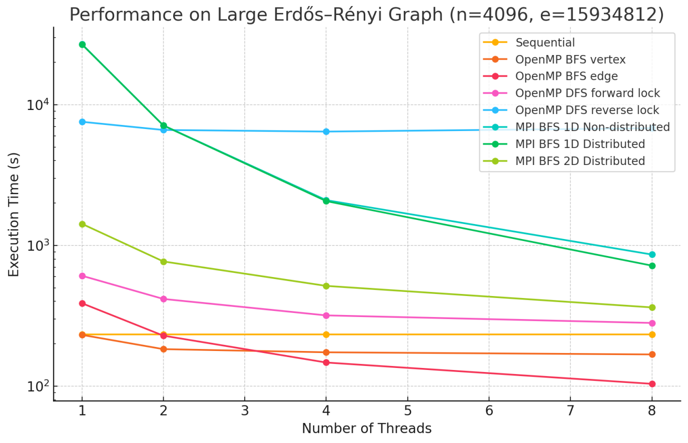


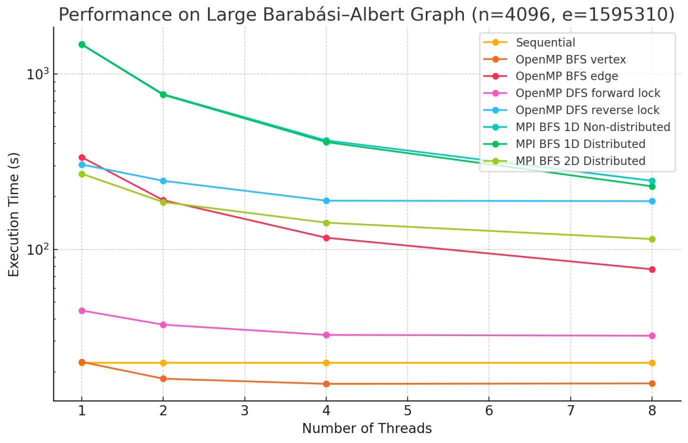

Performance on Small Grid Graph (n=121, e=440)
Execution Time (s)
Number of Threads
Sequential
OpenMP BFS vertex
OpenMP BFS edge
OpenMP DFS forward lock
OpenMP DFS reverse lock
MPI BFS 1D Non-distributed
MPI BFS 1D Distributed
MPI BFS 2D Distributed

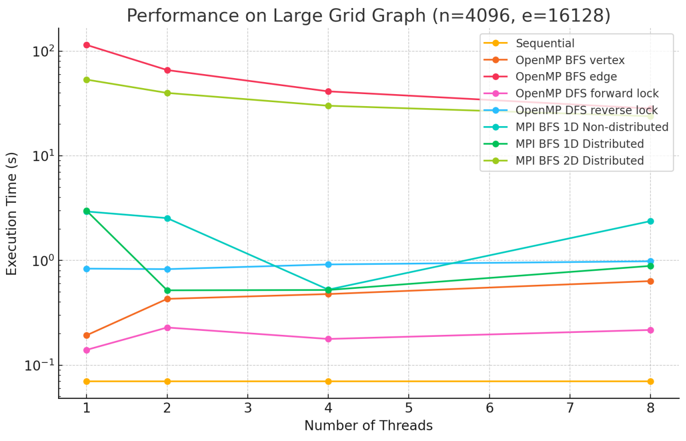

Performance on Medium Erdős–Rényi Graph (n=2048, e=3983314)
Execution Time (s)
Number of Threads
Sequential
OpenMP BFS vertex
OpenMP BFS edge
OpenMP DFS forward lock
OpenMP DFS reverse lock
MPI BFS 1D Non-distributed
MPI BFS 1D Distributed
MPI BFS 2D Distributed


Performance on Medium Barabási–Albert Graph (n=2048, e=400670)
Execution Time (s)
Number of Threads
Sequential
OpenMP BFS vertex
OpenMP BFS edge
OpenMP DFS forward lock
OpenMP DFS reverse lock
MPI BFS 1D Non-distributed
MPI BFS 1D Distributed
MPI BFS 2D Distributed

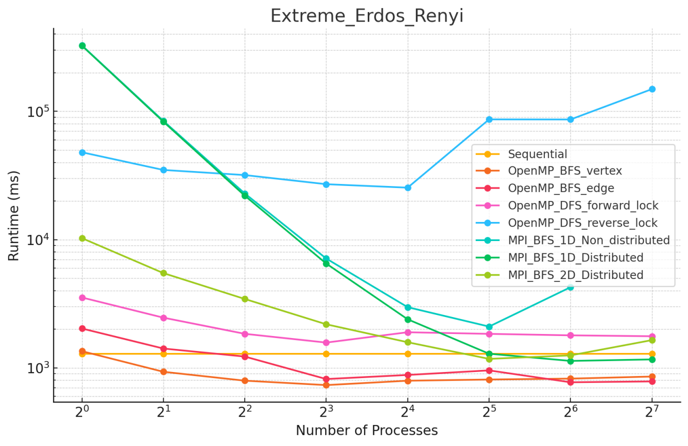

Performance on Medium Grid Graph (n=2025, e=7920)
Execution Time (s)
Number of Threads
Sequential
OpenMP BFS vertex
OpenMP BFS edge
OpenMP DFS forward lock
OpenMP DFS reverse lock
MPI BFS 1D Non-distributed
MPI BFS 1D Distributed
MPI BFS 2D Distributed

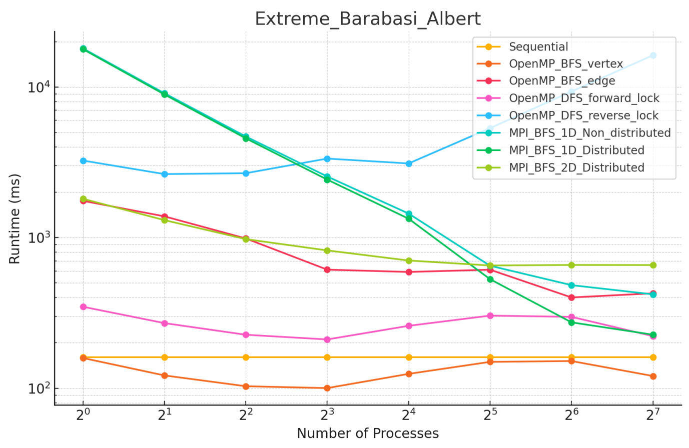

Performance on Large Erdős–Rényi Graph (n=4096, e=15934812)
Execution Time (s)
Number of Threads
Sequential
OpenMP BFS vertex
OpenMP BFS edge
OpenMP DFS forward lock
OpenMP DFS reverse lock
MPI BFS 1D Non-distributed
MPI BFS 1D Distributed
MPI BFS 2D Distributed

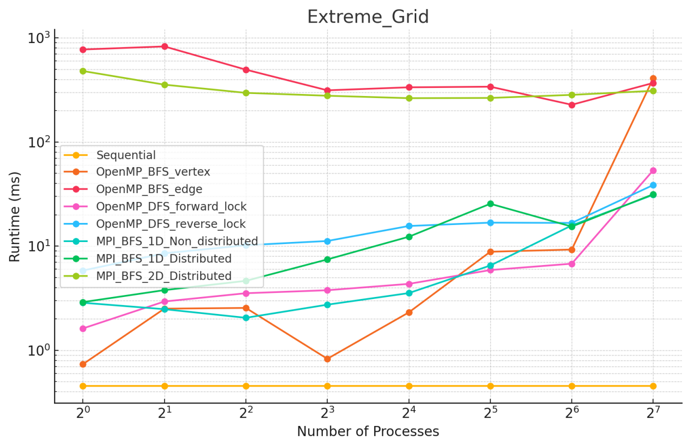

Performance on Large Barabási–Albert Graph (n=4096, e=1595310)
Execution Time (s)
Number of Threads
Sequential
OpenMP BFS vertex
OpenMP BFS edge
OpenMP DFS forward lock
OpenMP DFS reverse lock
MPI BFS 1D Non-distributed
MPI BFS 1D Distributed
MPI BFS 2D Distributed

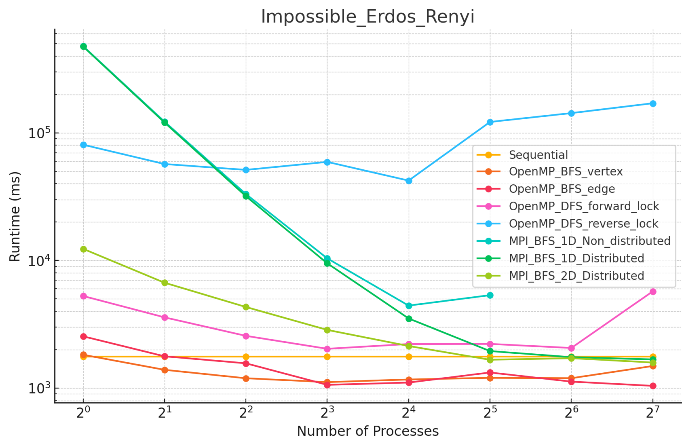


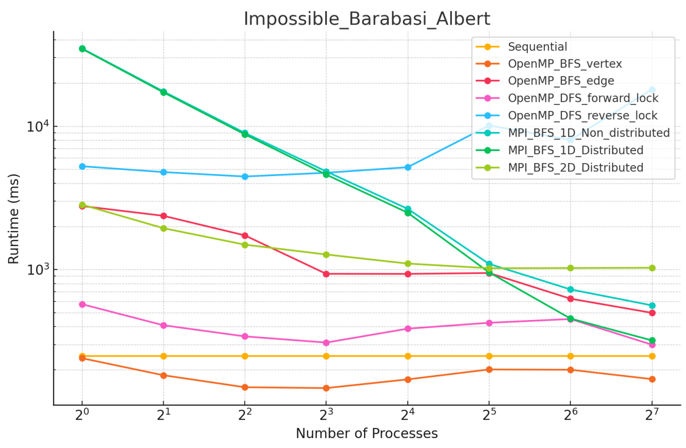
PSC Benchmarks

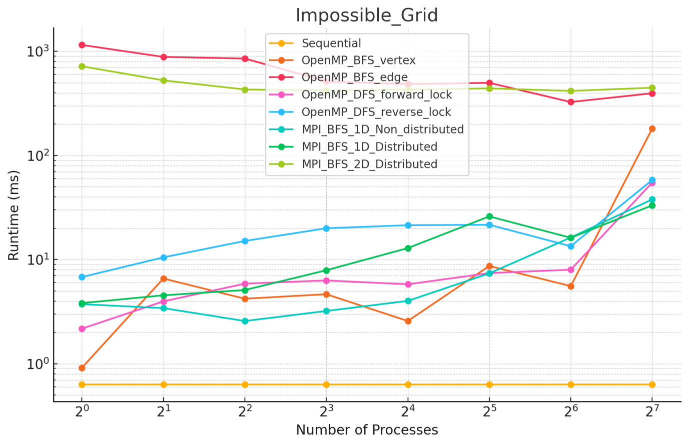



Extreme_Barabasi_Albert
Runtime (ms)
Number of Processes
Sequential
OpenMP_BFS_vertex
OpenMP_BFS_edge
OpenMP_DFS_forward_lock
OpenMP_DFS_reverse_lock
MPI_BFS_1D_Non_distributed
MPI_BFS_1D_Distributed
MPI_BFS_2D_Distributed


Extreme_Grid
Runtime (ms)
Number of Processes
Sequential
OpenMP_BFS_vertex
OpenMP_BFS_edge
OpenMP_DFS_forward_lock
OpenMP_DFS_reverse_lock
MPI_BFS_1D_Non_distributed
MPI_BFS_1D_Distributed
MPI_BFS_2D_Distributed


Impossible_Erdos_Renyi
Runtime (ms)
Number of Processes
Sequential
OpenMP_BFS_vertex
OpenMP_BFS_edge
OpenMP_DFS_forward_lock
OpenMP_DFS_reverse_lock
MPI_BFS_1D_Non_distributed
MPI_BFS_1D_Distributed
MPI_BFS_2D_Distributed





10 Discussion

From the benchmarking results, we observed that:

1. **OpenMP BFS**

    - In large and dense networks with more edges, parallelizing BFS in OpenMP shows measurable performance improvements against sequential implementation that increase with the number of threads up to 16. However, as the number of threads exceeds 16, it's hard to achieve further speedup. This is likely due to the limited number of parallelizable tasks available, causing increased overhead from thread management and synchronization, which reduces the effectiveness of additional threads.

- In most test cases, parallelizing by vertex yields better performance than parallelizing by edge. However, in the case of the Erdős–Rényi networks, parallelizing by edge often outperforms the vertex-based approach. This may be because the ER networks we tested have a higher edge density compared to other networks, making the adjacency matrix representation we use in the edge-based approach more efficient for these types of graphs.

2. **OpenMP DFS**

   - The speedup from parallelizing the DFS phase compared to the sequential version was minimal in most test cases, which could be caused by the high contention, synchronization, and overhead of managing locks. However, there are noticeable performance improvements when running with more threads versus a single thread in large and dense networks.
   - For most of the test cases, especially when there is a large number of edges, the reverse lock approach performs worse than the forward lock approach. This might be due to situations where some threads make progress that does not contribute meaningfully to the overall solution. For example, when another thread saturates one of the edges in the paths other threads are currently exploring. These unproductive efforts waste computational resources and add synchronization overhead.

3. **MPI BFS**

   - Overall, the MPI implementation of BFS parallelization performs worse than the OpenMP version, likely due to the high communication overhead involved in synchronizing nodes at the next level across processes.
   - In our 1D MPI implementation, running DFS redundantly on all processors avoids communication overhead but requires each process to maintain a copy of the entire graph. This can lead to out-of-memory errors for large test cases or high process counts. Conversely, running DFS on the root process reduces memory usage but introduces communication overhead, as residual capacities must be scattered after each DFS.
   - In large and dense graphs, when using a large number of processes, both 1D and 2D partitioning can outperform sequential implementations. This is primarily due to the efficient distribution of nodes and edges across processes.
   - 1D and 2D partitioning do not perform well with a small number of processes because the workload is less evenly distributed, and each

process may handle a large portion of the graph, leading to higher memory usage and reduced parallel efficiency. However, the performance grows as the number of processes increases, distributing the graph more evenly and reducing the per-process workload.

- Among MPI-based parallel BFS implementations, 2D partitioning often outperforms 1D partitioning on large and dense graphs when the number of processes is small, while 1D partitioning works better for sparse graphs. This is potentially due to different graph representations we used (adjacency matrix for 2D and adjacency list for 1D), leading to 2D partitioning having higher utilization with dense graphs.

Approaches	**Settings**	**Effects**
Sequential	Sequential	Faster in small and sparse graphs
By Vertex	OpenMP BFS	Faster than sequential in large and dense graphs
By Edge	OpenMP BFS	Faster than by vertex in very dense graphs
Forward Locking	OpenMP DFS	Slower than sequential, but exhibits speedup compared to single thread
Reverse Locking	OpenMP DFS	Slower than forward locking
1D Partition	MPI BFS	Slower than OpenMP BFS; faster than 2D in sparse graphs
2D Partition	MPI BFS	Slower than OpenMP BFS; faster than 1D in dense graphs

Table 1: Summary of Approaches and Results

11 Work Distribution

The work is evenly distributed, with 50% assigned to Connor Mowry and 50% to Wenqi Deng. Both of us collaborated on implementing the code. Besides that, Connor primarily focused on algorithm design, while Wenqi concentrated on generating tests and benchmarking.

Reference

1. https://hps.vi4io.org/_media/teaching/summer_term_2023/pchpc-student/jonas_hafermas_zoya_masih_report.pdf
2. https://www.cs.cmu.edu/~15451-f24/lectures/lecture11-flow1.pdf
3. https://www.cs.cmu.edu/~15451-f24/lectures/lecture12-flow2.pdf
4. https://en.wikipedia.org/wiki/Parallel_breadth-first_search