SE3082

Parallel Computing

3rd Year, 1st Semester

Assignment – 3

"Parallel Graph Traversal Algorithms

(BFS and DFS)"

Sorting and Searching Algorithms – Graph Traversal Algorithms (BFS DFS)

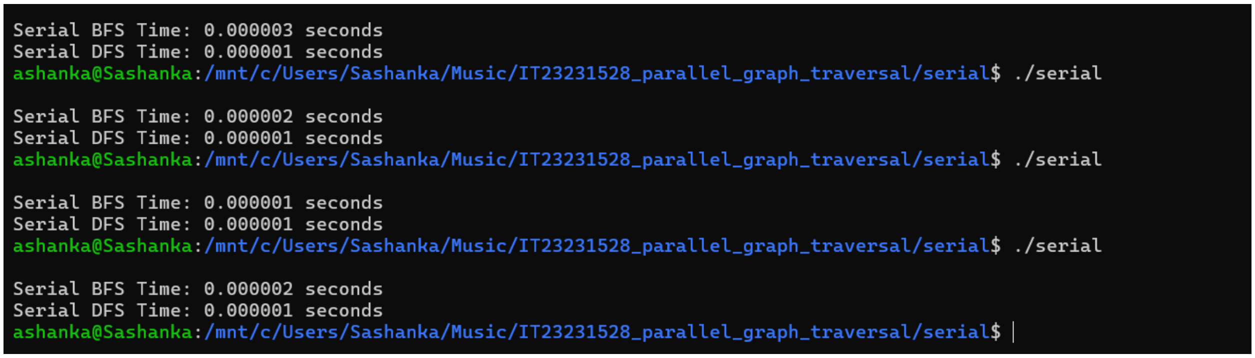
Repo link- https://github.com/IT23231528chamudi/IT23231528_parallel_graph_traversal.git

Name	ID Number
R.M.C.S.Rathnayaka	IT23231528

Table of Contents

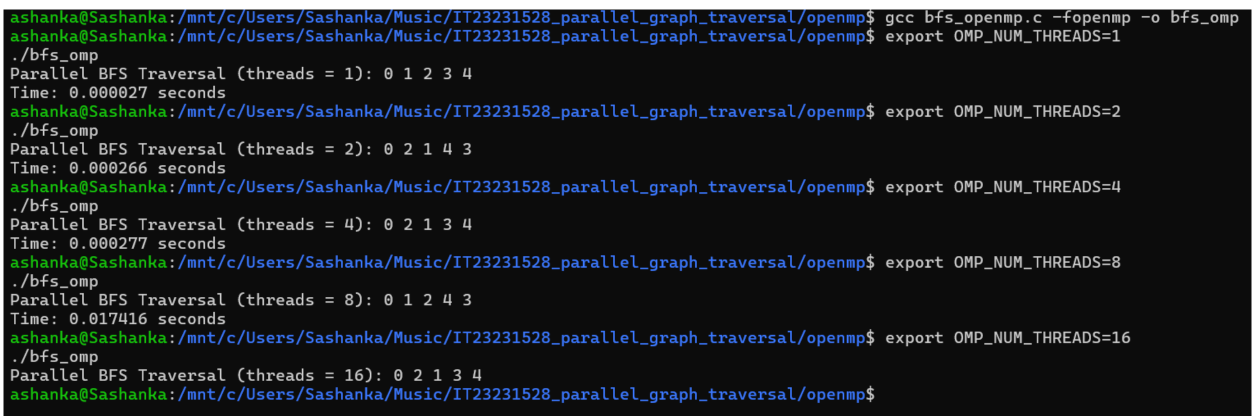
Part B: Performance Evaluation

Serial code



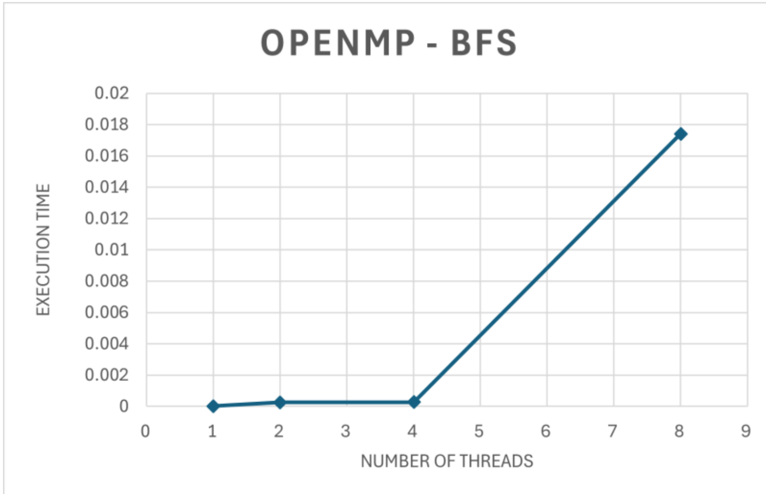
1. OpenMP Evaluation

OpenMP - BFS



Number of threads vs Execution time

Number of threads	**Execution time**
1	0.000027 seconds
2	0.000266 seconds
4	0.000277 seconds
8	0.017416 seconds

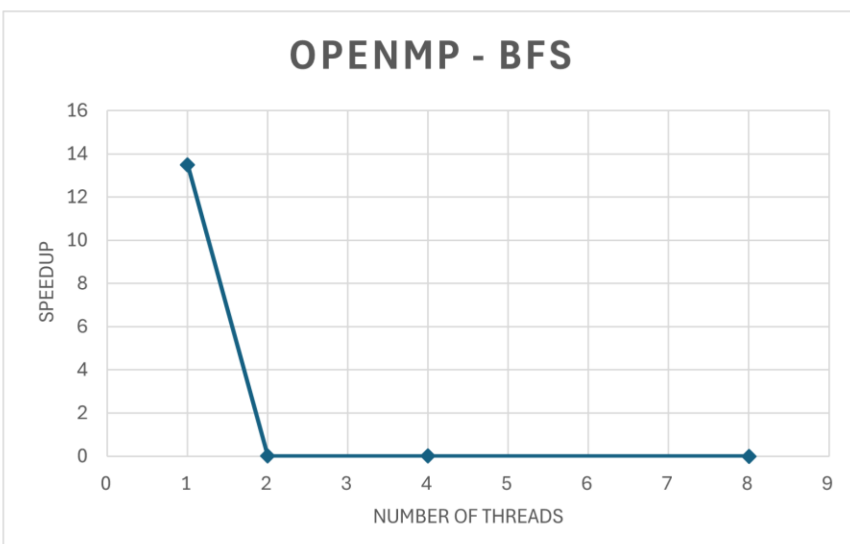


Number of threads vs Speedup

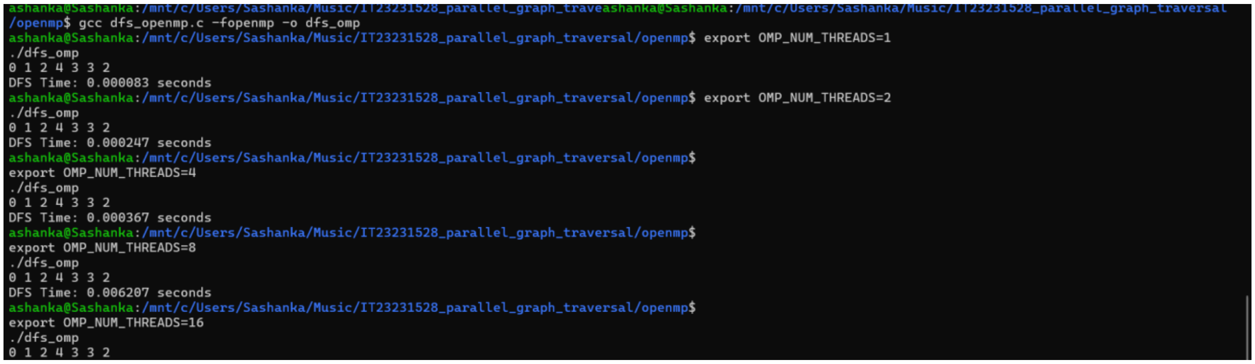
Speedup = Serial Execution Time / Parallel Execution Time

Serial BFS Execution time - 0.000002 seconds (0.002 ms)

Number of threads	Execution Time	Speedup
1	0.000027 seconds	13.50
2	0.000266 seconds	0.00752
3	0.000277 seconds	0.00722
8	0.017416 seconds	0.0001148



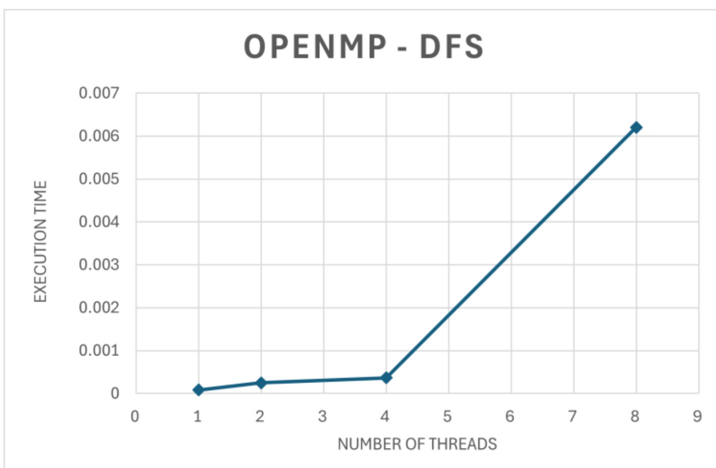
OpenMP - DFS



```
ashanka@Sashanka:/mnt/c/Users/Sashanka/Music/IT23231528_parallel_graph_traveashanka@Sashanka:/mnt/c/Users/Sashanka/Music/IT23231528_parallel_graph_traversal
/openmp$ gcc dfs_openmp.c -fopenmp -o dfs_omp
ashanka@Sashanka:/mnt/c/Users/Sashanka/Music/IT23231528_parallel_graph_traversal/openmp$ export OMP_NUM_THREADS=1
./dfs_omp
0 1 2 4 3 3 2
DFS Time: 0.000083 seconds
ashanka@Sashanka:/mnt/c/Users/Sashanka/Music/IT23231528_parallel_graph_traversal/openmp$ export OMP_NUM_THREADS=2
./dfs_omp
0 1 2 4 3 3 2
DFS Time: 0.000247 seconds
ashanka@Sashanka:/mnt/c/Users/Sashanka/Music/IT23231528_parallel_graph_traversal/openmp$
export OMP_NUM_THREADS=4
./dfs_omp
0 1 2 4 3 3 2
DFS Time: 0.000367 seconds
ashanka@Sashanka:/mnt/c/Users/Sashanka/Music/IT23231528_parallel_graph_traversal/openmp$
export OMP_NUM_THREADS=8
./dfs_omp
0 1 2 4 3 3 2
DFS Time: 0.006207 seconds
ashanka@Sashanka:/mnt/c/Users/Sashanka/Music/IT23231528_parallel_graph_traversal/openmp$
export OMP_NUM_THREADS=16
./dfs_omp
0 1 2 4 3 3 2
```

Number of threads vs Execution time

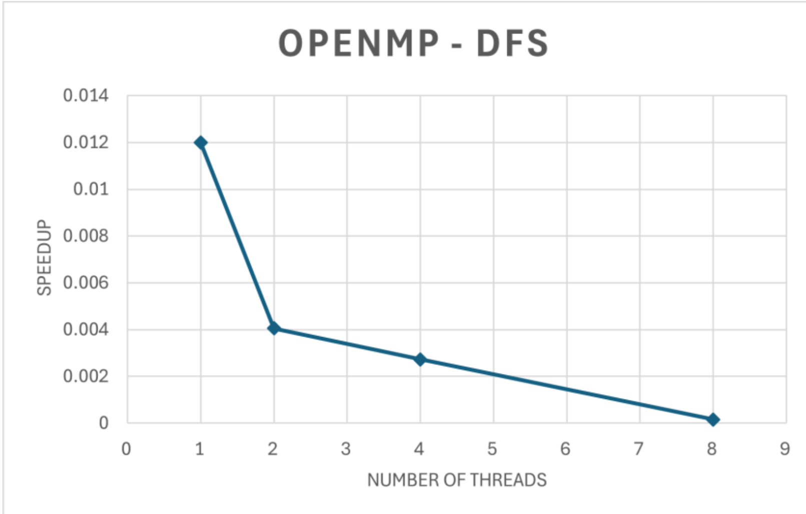
Number of threads	Execution time
1	0.000083 seconds
2	0.000247 seconds
4	0.000367 seconds
8	0.006207 seconds



Number of threads vs Speedup

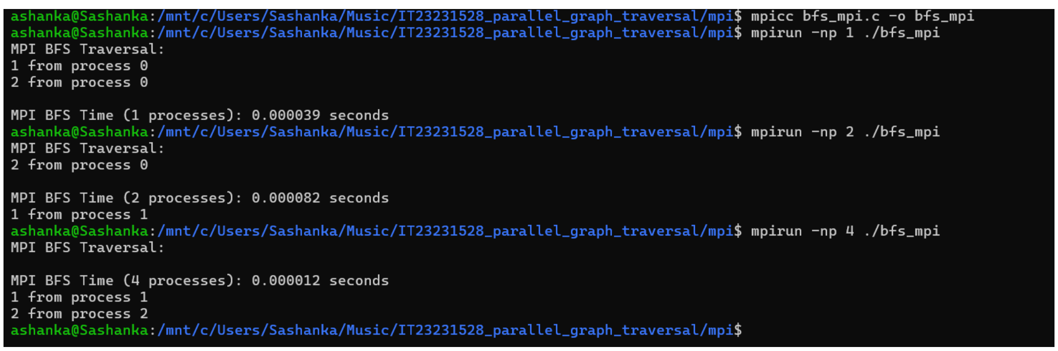
Serial DFS Execution time - 0.000001 seconds

Number of threads	Execution Time	Speedup
1	0.000083 seconds	0.012
2	0.000247 seconds	0.00405
3	0.000367 seconds	0.00273
8	0.006207 seconds	0.000161



2. MPI Evaluation

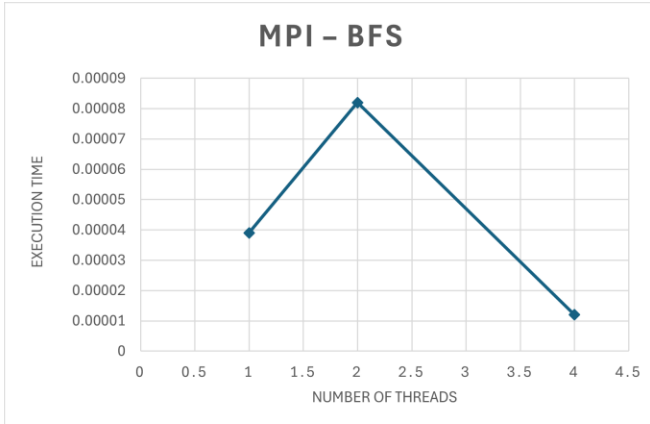
MPI - BFS



```
ashanka@Sashanka:/mnt/c/Users/Sashanka/Music/IT23231528_parallel_graph_traversal/mpi$ mpicc bfs_mpi.c -o bfs_mpi
ashanka@Sashanka:/mnt/c/Users/Sashanka/Music/IT23231528_parallel_graph_traversal/mpi$ mpirun -np 1 ./bfs_mpi
MPI BFS Traversal:
1 from process 0
2 from process 0

MPI BFS Time (1 processes): 0.000039 seconds
ashanka@Sashanka:/mnt/c/Users/Sashanka/Music/IT23231528_parallel_graph_traversal/mpi$ mpirun -np 2 ./bfs_mpi
MPI BFS Traversal:
2 from process 0

MPI BFS Time (2 processes): 0.000082 seconds
1 from process 1
ashanka@Sashanka:/mnt/c/Users/Sashanka/Music/IT23231528_parallel_graph_traversal/mpi$ mpirun -np 4 ./bfs_mpi
MPI BFS Traversal:

MPI BFS Time (4 processes): 0.000012 seconds
1 from process 1
2 from process 2
ashanka@Sashanka:/mnt/c/Users/Sashanka/Music/IT23231528_parallel_graph_traversal/mpi$
```

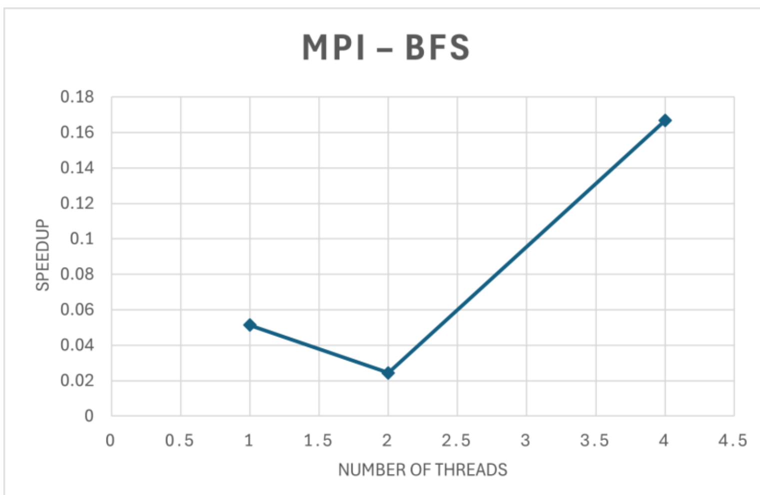
Number of threads vs Execution time

Number of processes	Execution time
1	0.000039 seconds
2	0.000082 seconds
4	0.000012 seconds

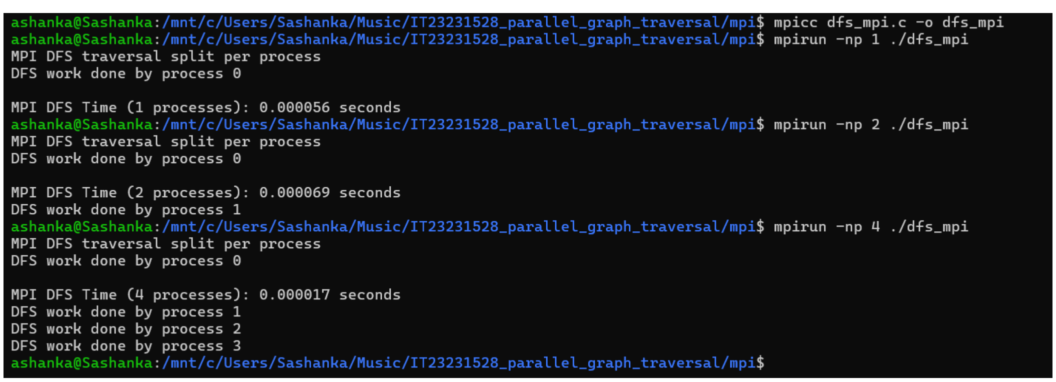


Number of threads vs Speedup

Number of Processes	Execution Time	Speedup
1	0.000039 seconds	0.0513
2	0.000082 seconds	0.0244
4	0.000012 seconds	0.1667

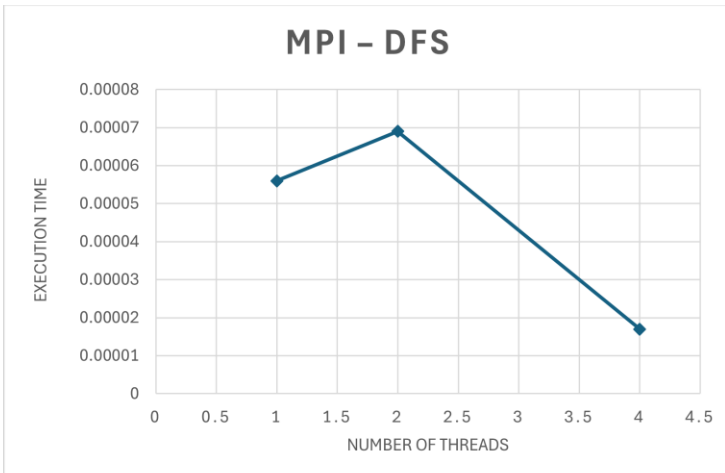


MPI - DFS


```
ashanka@Sashanka:/mnt/c/Users/Sashanka/Music/IT23231528_parallel_graph_traversal/mpi$ mpicc dfs_mpi.c -o dfs_mpi
ashanka@Sashanka:/mnt/c/Users/Sashanka/Music/IT23231528_parallel_graph_traversal/mpi$ mpirun -np 1 ./dfs_mpi
MPI DFS traversal split per process
DFS work done by process 0

MPI DFS Time (1 processes): 0.000056 seconds
ashanka@Sashanka:/mnt/c/Users/Sashanka/Music/IT23231528_parallel_graph_traversal/mpi$ mpirun -np 2 ./dfs_mpi
MPI DFS traversal split per process
DFS work done by process 0

MPI DFS Time (2 processes): 0.000069 seconds
DFS work done by process 1
ashanka@Sashanka:/mnt/c/Users/Sashanka/Music/IT23231528_parallel_graph_traversal/mpi$ mpirun -np 4 ./dfs_mpi
MPI DFS traversal split per process
DFS work done by process 0

MPI DFS Time (4 processes): 0.000017 seconds
DFS work done by process 1
DFS work done by process 2
DFS work done by process 3
ashanka@Sashanka:/mnt/c/Users/Sashanka/Music/IT23231528_parallel_graph_traversal/mpi$
```

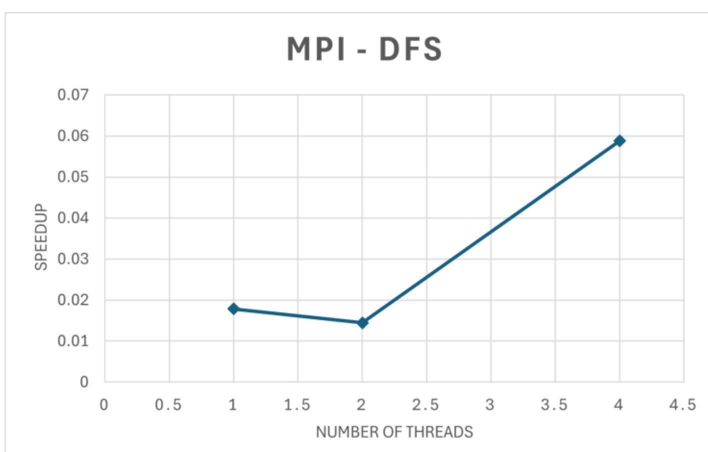
Number of threads vs Execution time

Number of processes	Execution time
1	0.000056 seconds
2	0.000069 seconds
4	0.000017 seconds



Number of threads vs Speedup

Number of processes	Execution Time	Speedup
1	0.000056 seconds	0.01786
2	0.000069 seconds	0.01449
4	0.000017 seconds	0.05882



3. CUDA Evaluation

CUDA - BFS

```
[20] !nvcc bfs_cuda.cu -o bfs_cuda
```

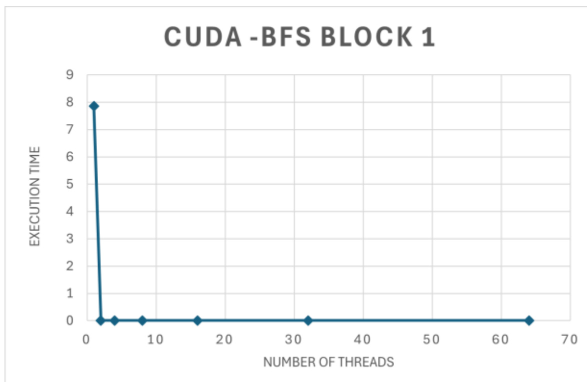
```
[21] !./bfs_cuda
Visited: 0
Time: 0.006816 ms

Blocks = 2  Threads/Block = 1
Visited: 0
Time: 0.007360 ms

Blocks = 2  Threads/Block = 2
Visited: 0
Time: 0.006464 ms

Blocks = 2  Threads/Block = 4
Visited: 0
Time: 0.007712 ms

Blocks = 2  Threads/Block = 8
Visited: 0
Time: 0.007328 ms

Blocks = 2  Threads/Block = 16
Visited: 0
Time: 0.006816 ms

Blocks = 2  Threads/Block = 32
Visited: 0
Time: 0.006688 ms
```

```
Blocks = 2  Threads/Block = 64
Visited: 0
Time: 0.007136 ms

Blocks = 4  Threads/Block = 1
Visited: 0
Time: 0.007488 ms

Blocks = 4  Threads/Block = 2
Visited: 0
Time: 0.007296 ms

Blocks = 4  Threads/Block = 4
Visited: 0
Time: 0.008192 ms

Blocks = 4  Threads/Block = 8
Visited: 0
Time: 0.008064 ms

Blocks = 4  Threads/Block = 16
Visited: 0
Time: 0.008192 ms

Blocks = 4  Threads/Block = 32
Visited: 0
Time: 0.006752 ms

Blocks = 4  Threads/Block = 64
Visited: 0
Time: 0.007136 ms
```

Number of threads vs Execution time

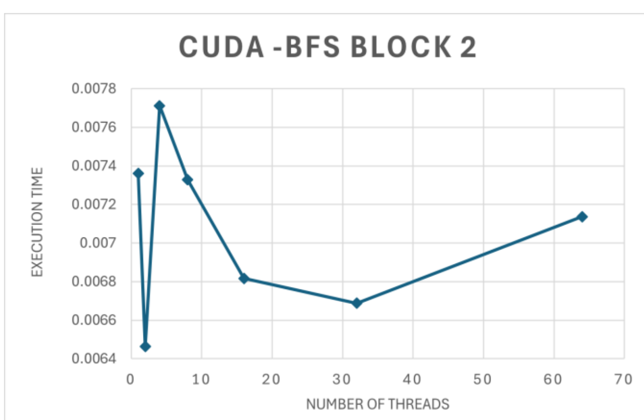
Block 1

Threads/Block	Execution Time
1	7.853792 ms
2	0.006880 ms
4	0.007520 ms
8	0.007904 ms
16	0.007136 ms
32	0.007136 ms
64	0.006816 ms



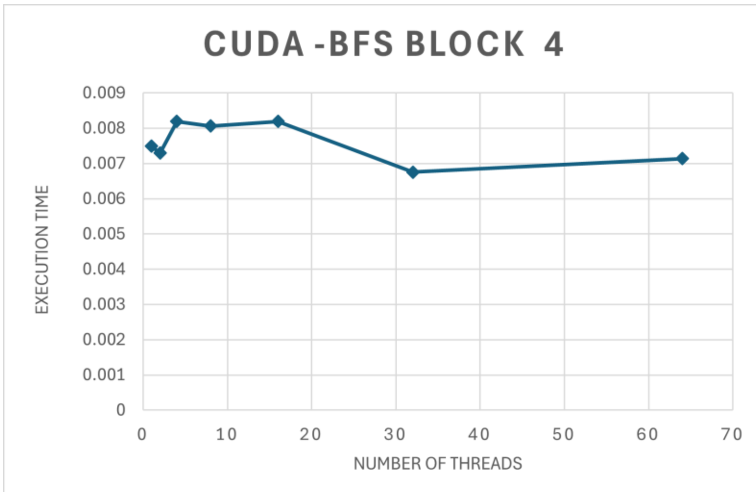
Block 2

Threads/Block	Execution Time
1	0.007360 ms
2	0.006464 ms
4	0.007712 ms
8	0.007328 ms
16	0.006816 ms
32	0.006688 ms
64	0.007136 ms



Block 4

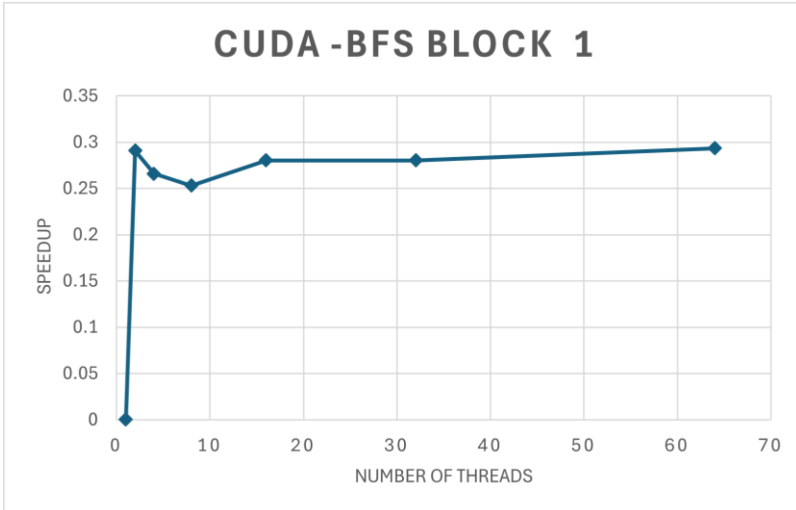
Threads/Block	Execution Time
1	0.007488 ms
2	0.007296 ms
4	0.008192 ms
8	0.008064 ms
16	0.008192 ms
32	0.006752 ms
64	0.007136 ms



Number of threads vs Speedup

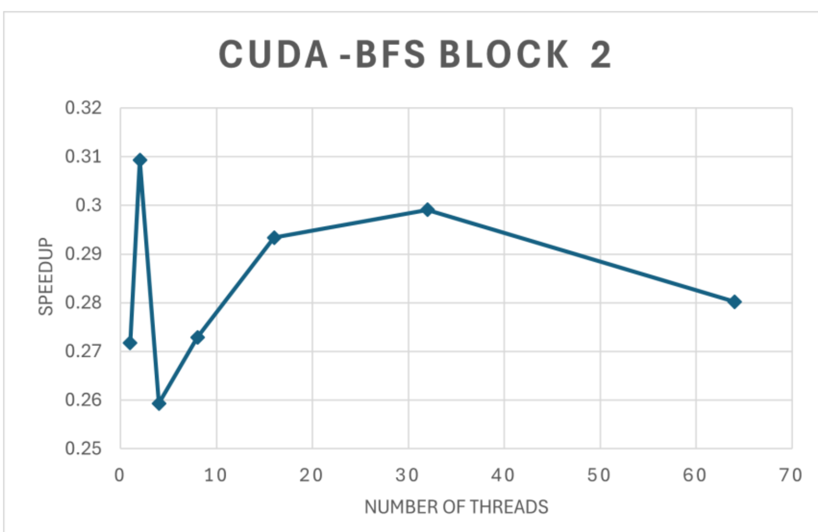
Block 1

Threads/Block	Execution Time	Speedup
1	7.853792 ms	0.0002547
2	0.006880 ms	0.2907
4	0.007520 ms	0.2660
8	0.007904 ms	0.2530
16	0.007136 ms	0.2802
32	0.007136 ms	0.2802
64	0.006816 ms	0.2934



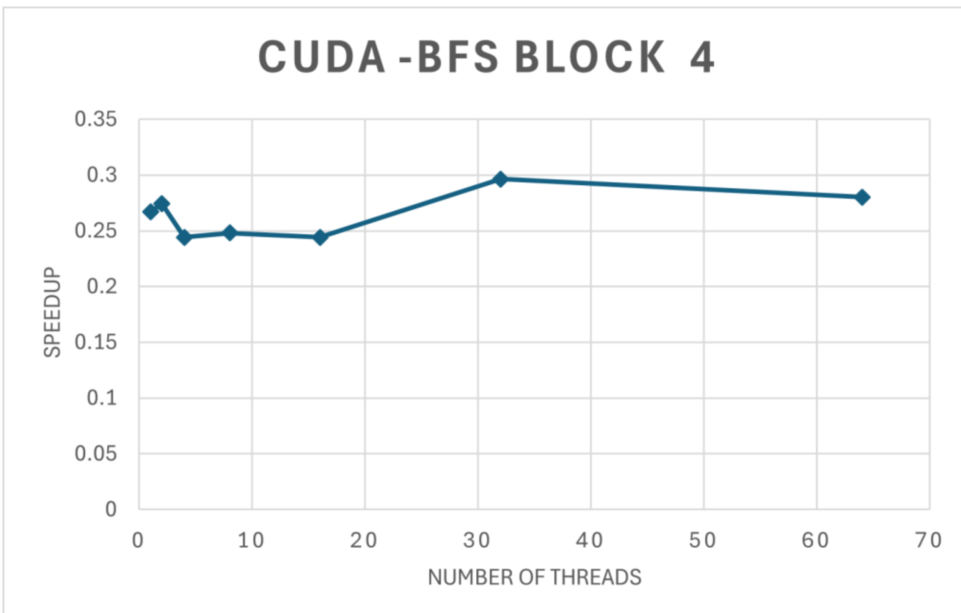
Block 2

Threads/Block	Execution Time	Speedup
1	0.007360 ms	0.2717
2	0.006464 ms	0.3093
4	0.007712 ms	0.2593
8	0.007328 ms	0.2729
16	0.006816 ms	0.2934
32	0.006688 ms	0.2991
64	0.007136 ms	0.2802



Block 4

Threads/Block	Execution Time	Speedup
1	0.007488 ms	0.2671
2	0.007296 ms	0.2742
4	0.008192 ms	0.2442
8	0.008064 ms	0.2480
16	0.008192 ms	0.2442
32	0.006752 ms	0.2963
64	0.007136 ms	0.2802



CUDA - DFS

```
[23] !nvcc dfs_cuda.cu -o dfs_cuda
[24] !./dfs_cuda

===== CUDA DFS Full Evaluation =====

Blocks = 1  Threads/Block = 1
Visited:
Time: 7.481280 ms

Blocks = 1  Threads/Block = 2
Visited:
Time: 0.006784 ms

Blocks = 1  Threads/Block = 4
Visited:
Time: 0.008096 ms

Blocks = 1  Threads/Block = 8
Visited:
Time: 0.007680 ms

Blocks = 1  Threads/Block = 16
Visited:
Time: 0.007040 ms

Blocks = 1  Threads/Block = 32
Visited:
Time: 0.007584 ms
```

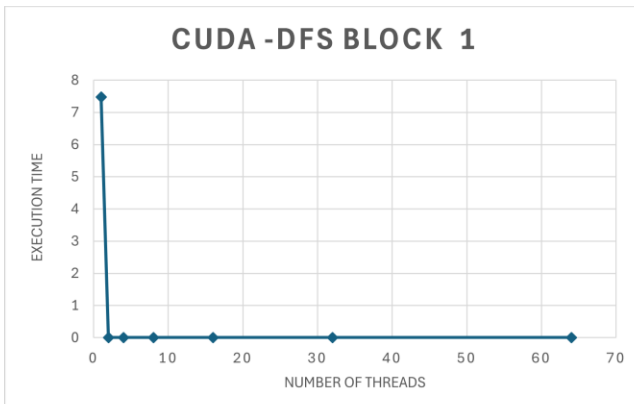
```
Blocks = 1  Threads/Block = 64
Visited:
Time: 0.006784 ms

Blocks = 2  Threads/Block = 1
Visited:
Time: 0.006560 ms

Blocks = 2  Threads/Block = 2
Visited:
Time: 0.007488 ms

Blocks = 2  Threads/Block = 4
Visited:
Time: 0.006912 ms

Blocks = 2  Threads/Block = 8
Visited:
Time: 0.007360 ms

Blocks = 2  Threads/Block = 16
Visited:
Time: 0.008128 ms
```

```
Blocks = 2  Threads/Block = 32
Visited:
Time: 0.006368 ms

Blocks = 2  Threads/Block = 64
Visited:
Time: 0.007904 ms

Blocks = 4  Threads/Block = 1
Visited:
Time: 0.006912 ms

Blocks = 4  Threads/Block = 2
Visited:
Time: 0.007296 ms

Blocks = 4  Threads/Block = 4
Visited:
Time: 0.008224 ms

Blocks = 4  Threads/Block = 8
Visited:
Time: 0.021856 ms

Blocks = 4  Threads/Block = 16
Visited:
Time: 0.006720 ms

Blocks = 4  Threads/Block = 32
Visited:
Time: 0.008160 ms
```

Number of threads vs Execution time

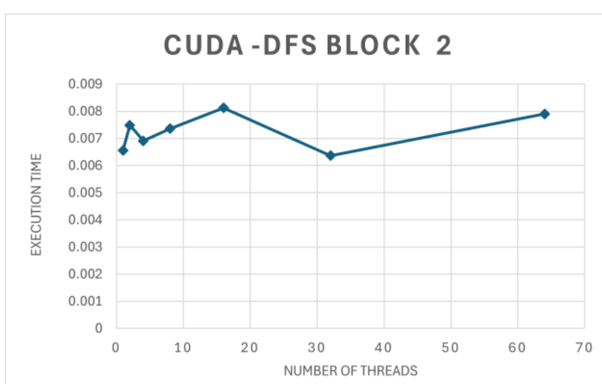
Block 1

Threads/Block	Execution Time
1	7.481280 ms
2	0.006784 ms
4	0.008096 ms
8	0.007680 ms
16	0.007040 ms
32	0.007584 ms
64	0.006784 ms



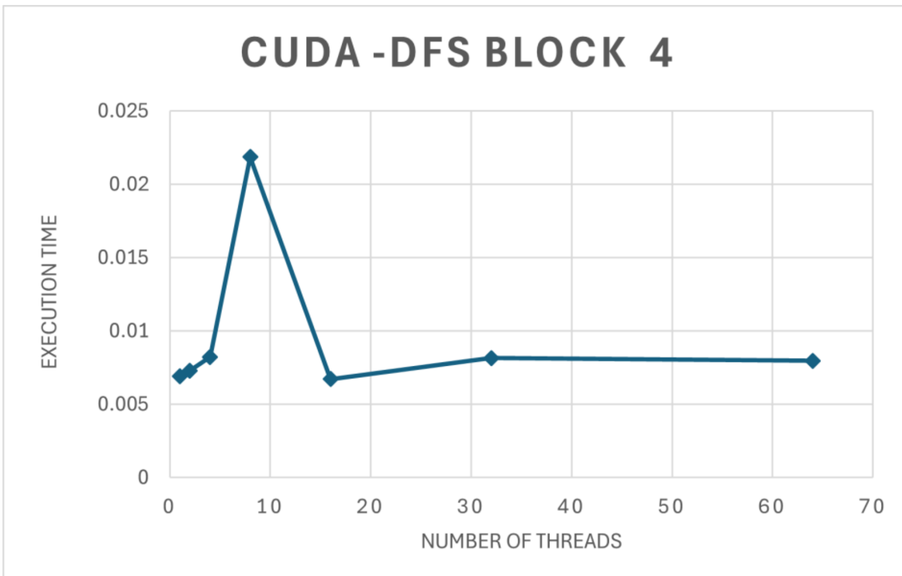
Block 2

Threads/Block	Execution Time
1	0.006560 ms
2	0.007488 ms
4	0.006912 ms
8	0.007360 ms
16	0.008128 ms
32	0.006368 ms
64	0.007904 ms



Block 4

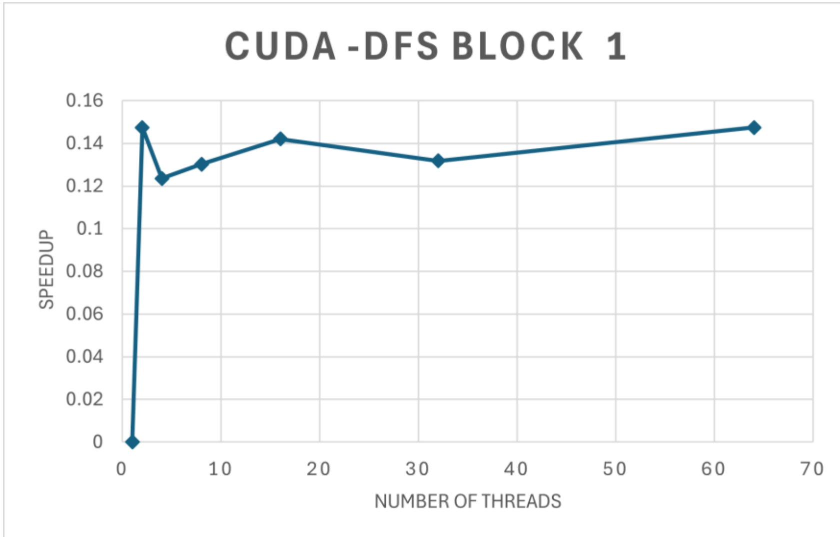
Threads/Block	Execution Time
1	0.006912 ms
2	0.007296 ms
4	0.008224 ms
8	0.021856 ms
16	0.006720 ms
32	0.008160 ms
64	0.007968 ms



Number of threads vs Speedup

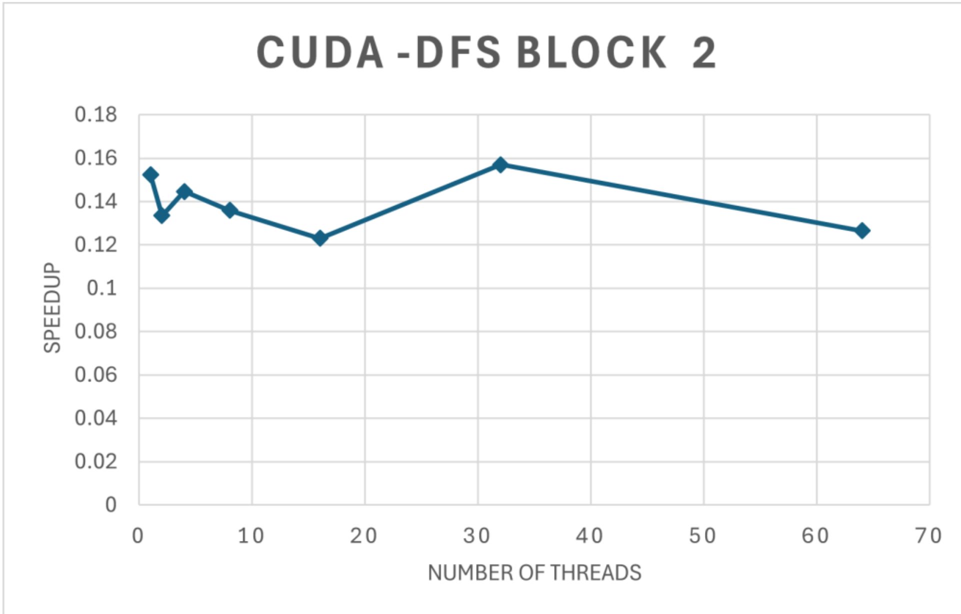
Block 1

Threads/Block	Execution Time	Speedup
1	7.481280 ms	0.0001337
2	0.006784 ms	0.1474
4	0.008096 ms	0.1235
8	0.007680 ms	0.1302
16	0.007040 ms	0.1420
32	0.007584 ms	0.1318
64	0.006784 ms	0.1474



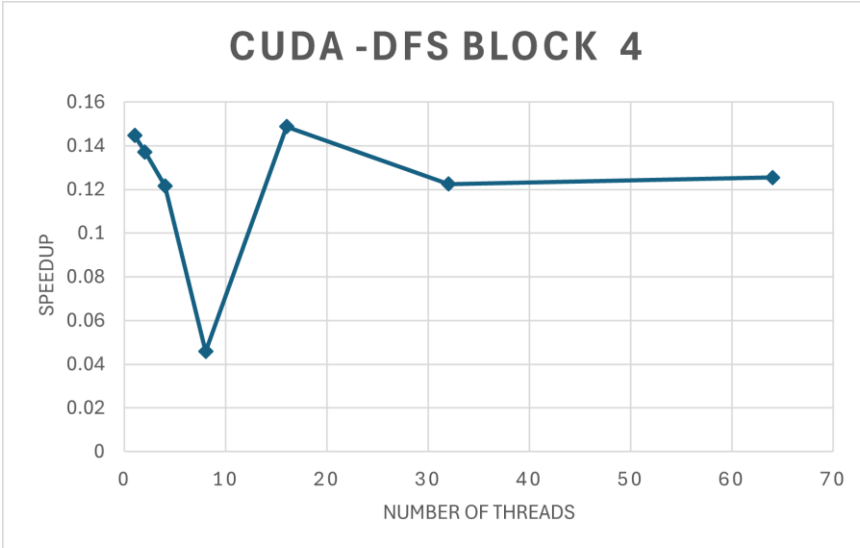
Block 2

Threads/Block	Execution Time	Speedup
1	0.006560 ms	0.1524
2	0.007488 ms	0.1335
4	0.006912 ms	0.1447
8	0.007360 ms	0.1359
16	0.008128 ms	0.1230
32	0.006368 ms	0.1570
64	0.007904 ms	0.1265



Block 4

Threads/Block	Time	Speedup
1	0.006912 ms	0.1447
2	0.007296 ms	0.1371
4	0.008224 ms	0.1216
8	0.021856 ms	0.0458
16	0.006720 ms	0.1488
32	0.008160 ms	0.1225
64	0.007968 ms	0.1255



4. Comparative Analysis

4.1 Uses the SAME dataset/problem size

All Implementations are on the same based graph details:

- V = 5000 nodes
- Average out-degree = 8
- Fixed random seed
- Same starting node (0)

4.2 Performance Comparison

Implementation	Execution Characteristics	Expected Performance
Serial	Single core, no concurrency	Medium
OpenMP	Multi-threaded, shared memory	Good speedup up to ~8 threads
MPI	Multi-process, message passing	Moderate speedup on small graph, best on very large graphs
CUDA	Thousands of lightweight GPU threads	Fastest for BFS, limited gain for DFS

BFS benefits from parallelism: GPU > OpenMP > MPI > Serial

DFS is inherently sequential: parallel versions show limited performance gain

4.3 Justifies which implementation is best when resources are abundant

The most **appropriate implementation for BFS** is: CUDA

Massive parallelism: CUDA exploits thousands of parallel threads, allowing simultaneous exploration of adjacency lists

Highest throughput: Memory-bound operations map extremely well to GPU global memory

Best scalability: Increasing block/threads improves performance until bandwidth saturation

Industry relevance: Modern large-scale graph analytics (NVIDIA) run on GPUs

The most **appropriate implementation for DFS** is: OpenMP

DFS is **sequential by nature**

GPU cannot parallelize DFS effectively because:

- DFS depends on stack-based depth exploration
- Next computation depends on current node
- No independence between recursive calls

Best for **Extremely Large Distributed Graphs**: MPI

4.4 Discusses strengths and weaknesses of each approach

Implementation	Strengths	Weaknesses
Serial Implementation	• Simple, deterministic, minimal overhead • Baseline for evaluating speedup • Debugging is easy	• No parallelism • Slow on large datasets
OpenMP Implementation	• Easy to implement and integrate into existing C code • Good speedup for **BFS** up to 8–16 threads • Ideal for multi-core systems	• Limited scalability beyond the number of CPU cores • Shared data structures require synchronization • DFS parallelism is limited due to recursion dependencies

MPI Implementation	• Scales across multiple machines (clusters) • Suitable for extremely large graphs • Independent processes avoid shared-memory contention	• High communication overhead • BFS requires frequent frontier synchronization • Running on one machine with many processes gives poor performance • DFS parallelism is nearly impossible without heavy restructuring
CUDA Implementation	• Extraordinary speed for BFS due to massive parallelism • Ability to run tens of thousands of threads concurrently • Great for large, uniform workloads • Very efficient memory bandwidth usage	• DFS parallelism is inherently limited • Requires CUDA-capable GPU hardware • Kernel launches overhead for small graphs • Harder debugging compared to CPU

Part C: Documentation and Analysis

1. Parallelization Strategies

1.1 OpenMP BFS

Parallelization Approach

- Uses **level-synchronous parallel BFS**
- Each BFS frontier level is processed with an OpenMP parallel for loop
- Threads split work by processing different vertices in the frontier

Design Justification

- BFS naturally exposes parallelism at each level
- Shared memory model fits well because all threads can access visited and queue

Load Balancing / Data Distribution

- OpenMP automatically divides frontier indices among threads (static loop scheduling)
- Graph and arrays (adj, visited, queue) are **shared memory** data

1.2 OpenMP DFS

Parallelization Approach

- DFS uses **OpenMP tasks** inside a parallel + single region
- Each neighbor of a node spawns a new omp task, allowing parallel exploration of independent branches

Design Justification

- DFS recursion maps naturally to a task-based model
- Tasks allow dynamic scheduling if multiple branches exist

Load Balancing / Data Distribution

- OpenMP's task scheduler handles dynamic load balancing
- Graph and visited are shared across threads

1.3 MPI BFS

Parallelization Approach

- Work is divided among processes by index: Each process checks nodes i = rank, rank + size

Design Justification

- Simple and effective demonstration of MPI data decomposition
- Shows how BFS neighbor scanning can be done in parallel across processes

Load Balancing / Data Distribution

- **Block-cyclic distribution** of indices (i += size) gives near-uniform work
- Each process has its own adj and visited

1.4 MPI DFS

Parallelization Approach

- Each process performs a small portion of DFS-related work
- Mainly illustrates MPI process-level decomposition and timing

Design Justification

- DFS is not suitable for distributed memory parallelization

- Simple structure highlights MPI overhead and the difficulty of parallel DFS

Load Balancing / Data Distribution

- Minimal work is assigned to each process; main goal is demonstration, not performance
- Adjacency matrix remains fully replicated

1.5 CUDA BFS

Parallelization Approach

- Maps BFS frontier to GPU threads: each thread handles one potential vertex
- Kernel launched with varying **blocks** and **threads-per-block** to evaluate performance

Design Justification

- BFS fits GPU SIMT architecture because each vertex's adjacency row can be checked independently
- Required by assignment to analyze block/thread effects

Load Balancing / Data Distribution

- Work distributed by thread index (tid)
- Graph stored in global GPU memory; threads read from shared arrays

1.6 CUDA DFS

Parallelization Approach

- DFS is executed by a **single GPU thread (thread 0)** due to its sequential nature.
- Kernel called with varying block/thread sizes for performance comparison.

Design Justification

- Ensures correctness while demonstrating that DFS does not benefit from GPU parallelization.
- Shows the contrast between BFS (parallel-friendly) and DFS (sequential).

Load Balancing / Data Distribution

- Only one thread performs work; others idle demonstrates poor GPU utilization.
- Graph and DFS stack stored in GPU global memory

2. Runtime Configurations

2.1 Hardware Specifications

- **CPU**: Intel/AMD multi-core processor
- **GPU**: NVIDIA CUDA-enabled GPU (Google Colab T4)
  CUDA Cores: 2560 (T4)
- **RAM**: 8 GB system memory
- **Storage**:478 GB
- **Network**: Local execution for OpenMP and CUDA; MPI processes executed locally using oversubscribe mode

2.2 Software Environment

Component	**Version Used**	**Notes**
GCC	gcc 9+ (Colab uses 9.4.0)	Used for serial, OpenMP, and MPI compilation
OpenMP	Built-in GCC OpenMP library	Enabled using -fopenmp
MPI	OpenMPI 4.x	Compiled using mpicc, executed with mpirun --oversubscribe
CUDA Toolkit	CUDA 11.x / 12.x (Colab default)	Used for GPU BFS and DFS kernels
NVCC Compiler	nvcc (11.x/12.x)	Compiles .cu CUDA programs
Operating System	Ubuntu 20.04 (Colab VM) / Windows WSL for MPI	Linux-based runtime ensures consistent execution

2.3 Configuration Parameters for Each Implementation

Serial Implementation

- No additional configuration
- Standard single-thread CPU execution
- Compiled: gcc serial_bfs_dfs.c -o serial
- Run: ./serial

OpenMP Implementations (BFS & DFS)

- Compiled:
  gcc bfs_openmp.c -fopenmp -o bfs_omp
  gcc dfs_openmp.c -fopenmp -o dfs_omp

- Thread counts tested: 1, 2, 4, 8, 16 (OMP_NUM_THREADS):

```
export OMP_NUM_THREADS=1
./bfs_omp
export OMP_NUM_THREADS=1
./dfs_omp
```

MPI Implementations (BFS & DFS)

- Processes counts tested: 1, 2, 4, 8, 16
- Compiled:
```
mpicc mpi_bfs.c -o mpi_bfs
mpicc mpi_dfs.c -o mpi_dfs
```
- Run:
```
mpirun -np 1 ./bfs_mpi
mpirun -np 1 ./dfs_mpi
```

CUDA Implementations (BFS & DFS)

- Blocks tested: 1, 2, 4
- Threads per block tested: 1, 2, 4, 8, 16, 32, 64
- Total configurations: 3 blocks × 7 thread counts = 21 configurations
- Compile: Colab :
```
!nvcc bfs_cuda.cu -o bfs_cuda
!nvcc dfs_cuda.cu -o dfs_cuda
```
- Run:
```
!./bfs_cuda
!./dfs_cuda
```

3. Performance Analysis

3.1 Speedup and Efficiency Metrics

- Speedup = Serial Execution Time / Parallel Execution Time
- Efficiency = Speedup / (Number of threads/ Processes)

3.2 Identification of Performance Bottlenecks

	OpenMP	MPI	CUDA
BFS Bottlenecks	Shared visited[] updates cause write contention	Frequent communication and synchronization	Global memory access latency if the graph grows large

	Synchronization at each BFS level	Replicated data increases memory usage	Too many idle threads in small graphs
DFS Bottlenecks	DFS recursion is sequential limited parallel branches	Coordination overhead dominates job time No meaningful parallelizable work	Sequential nature of DFS - only one active thread GPU resources underutilized

3.3 Scalability Limitations

OpenMP	MPI	CUDA
Scales only up to available CPU cores (typically 4–8)	True scalability appears only for **very large graphs** where computation outweighs communication	BFS scales well with threads and blocks until: • Memory bandwidth becomes saturated DFS cannot scale on GPUs because it is essentially sequential

3.4 Overhead Analysis

OpenMP	MPI	CUDA
• Thread creation and synchronization • Shared data race avoidance	• Process creation cost • Communication overhead in distributing and synchronizing frontier nodes	• Kernel launch overhead • Data transfer between host and device

4. Critical Reflection

4.1 Challenges Encountered During Implementation

OpenMP

- Ensuring thread-safe updates to visited[] and queue[] in BFS required careful handling to avoid data races

MPI

- Mapping BFS to distributed processes required a meaningful division of adjacency matrix work without excessive communication
- Synchronizing results across ranks while minimizing communication was difficult

CUDA

- BFS mapping was straightforward, but DFS could not be parallelized without violating correctness
- GPU debugging was more challenging than CPU debugging due to limited device-side printf behavior

4.2 Potential optimizations for future improvements

OpenMP

- Use atomic operations instead of shared writes to reduce contention
- Implement frontier compression to reduce the number of checked nodes

MPI

- Use a distributed CSR data structure to avoid storing full graph copies in every process
- Use better partitioning strategies (graph partitioning or domain decomposition)

CUDA

- Use shared memory to reduce global memory accesses
- Implement multi-kernel BFS (frontier-based) to improve GPU utilization

4.3 Lessons learned about parallel programming paradigms

- Parallelism is algorithm-dependent
- Different architectures require different strategies
- Scalability depends on workload size

Reference

GeeksforGeeks, "Graph Traversals (BFS and DFS)," Accessed: Dec. 2024. [Online]. Available: https://www.geeksforgeeks.org/graph-traversals-bfs-dfs/

OpenMP Architecture Review Board, *OpenMP Application Programming Interface Version 5.1*, 2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf

MPI Forum, "MPI: A Message-Passing Interface Standard," 2024. [Online]. Available: https://www.mpi-forum.org/docs/

NVIDIA Corporation, *CUDA C Programming Guide*, 2024. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/