PROJECT REPORT

for

CS-3006: Parallel and Distributed Computing

Prepared by Ahmed Murtaza Malik, i22-0985

FAST NUCES

6th May, 2025

Parallelization for Butterfly Counting Algorithms using OpenMP & MPI

Introduction

Butterfly counting is a fundamental graph problem that involves identifying the number of butterflies in a bipartite graph. This metric is critical for analyzing network structures in social networks, bioinformatics, and recommendation systems. The paper "Parallel Algorithms for Butterfly Computations" proposes efficient parallel strategies in their ParButterfly algorithm to tackle this problem at scale. This project implements the algorithm using a hybrid approach combining MPI (for distributed computing), OpenMP (for shared-memory parallelism), and METIS (for graph partitioning). The goal is to evaluate the scalability and performance of parallelization strategies on large real-world datasets.

- V1: Baseline serial implementation.
- V2: Multithreaded OpenMP version.
- V3: Distributed version utilizing METIS, MPI, and OpenMP.

The goal is to apply Parallel & Distributed Computing techniques and analyze the performance and scalability of the program on real-world large network datasets.

Datasets Used

Very few bipartite graphs were available online and most of them were either small for testing the scalability and performance of the application, or they had issues like ghost edges for which no nodes existed.

Hence a python script was used which generated a bipartite graph of a user-defined number of nodes and edges. For testing purposes, the following combinations of nodes and edges were used:
– 200 nodes, 500 edges
– 5000 nodes, 100000 edges
– 25000 nodes, 500000 edges
* The number of nodes mentioned here is the total number of nodes, half of this number was given to both sets in the bipartite graph.

Implementation

Implementation of the different versions followed these steps:

V1 (Serial)

– Used to provide a performance baseline.
– Wedge retrieval and grouping done.
– Global data structures consolidation.
– Degree based reordering.
– Two pass adjacency build
– Serially reads the dataset and counts the number of butterflies present in the dataset.

V2 (OpenMP)
– All of the above has been implemented alongside OpenMP for multithreading.
– Work distributed amongst threads and critical sections protected.

V3 (MPI + OpenMP)
– The root reads the bipartite graph and reorders U-nodes in descending order.
– Adjacency bitsets are built for U-nodes, where each bit represents a connection to an I-node.
– Root Broadcast critical data (U/I node lists, adjacency bitsets, sizes) to all processes.
– Each process receives a contiguous chunk of the U-nodes to process. (e.g 20 U-nodes per process for a launch of 4 processes with 80 U-nodes)
– Each process iterates over its assigned U-nodes and counts butterflies with all higher-indexed U-nodes to avoid duplicate counting.
– MPI Reduce is used to sum local butterfly counts into a global total on the root process.

Performance Results

The table below compares execution times (ms) for key operations across versions:

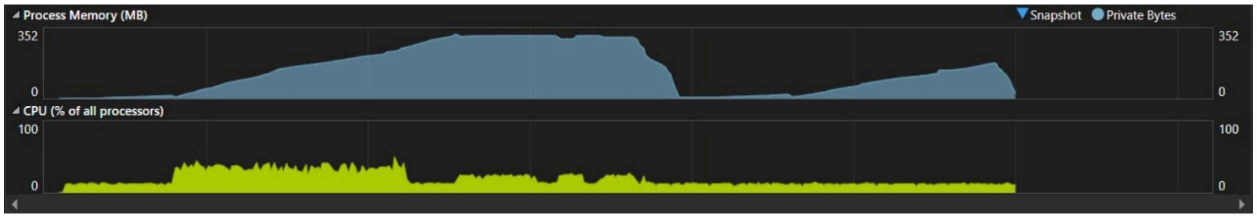
Dataset	Time: Serial	Time: OpenMP, 4 Threads	Time: OpenMP, 8 Threads	Speedup Achieved
Paper Dataset	0.6198s	0.0231s	0.01695s	-
200 nodes, 500 edges	0.062s	0.0199s	0.102s	3.11x
5k nodes, 100k edges	7.511s	6.35s	5.75s	1.30x
20k nodes, 1M edges	138s	107s	196s	1.29x

Times for varying MPI processes have also been compared here:

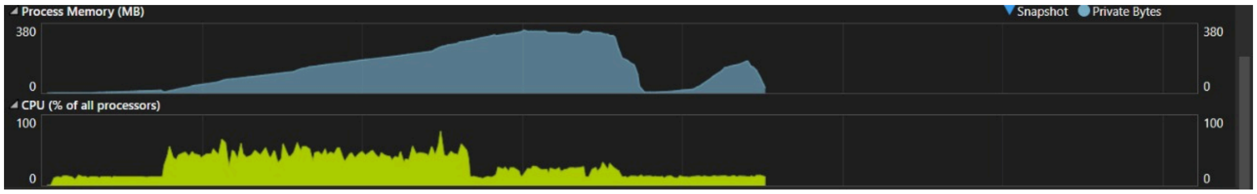
Dataset	Time: MPI 1 Process	Time: MPI 2 Processes	Time: MPI 3 Processes	Time: MPI 4 Processes
Paper Dataset	0.00982s	0.00904s	0.00926s	0.00932s
200 nodes, 500 edges	0.007504s	0.00529s	0.004446s	0.004059s
5k nodes, 100k edges	1.7639s	1.6972s	1.5451s	1.49801s
20k nodes, 1M edges	81.0292s	78.7506s	93.7702s	91.3416s

Application Profile & Analysis

Done using Microsoft Visual Studio BuiltIn Profiler, it shows up as



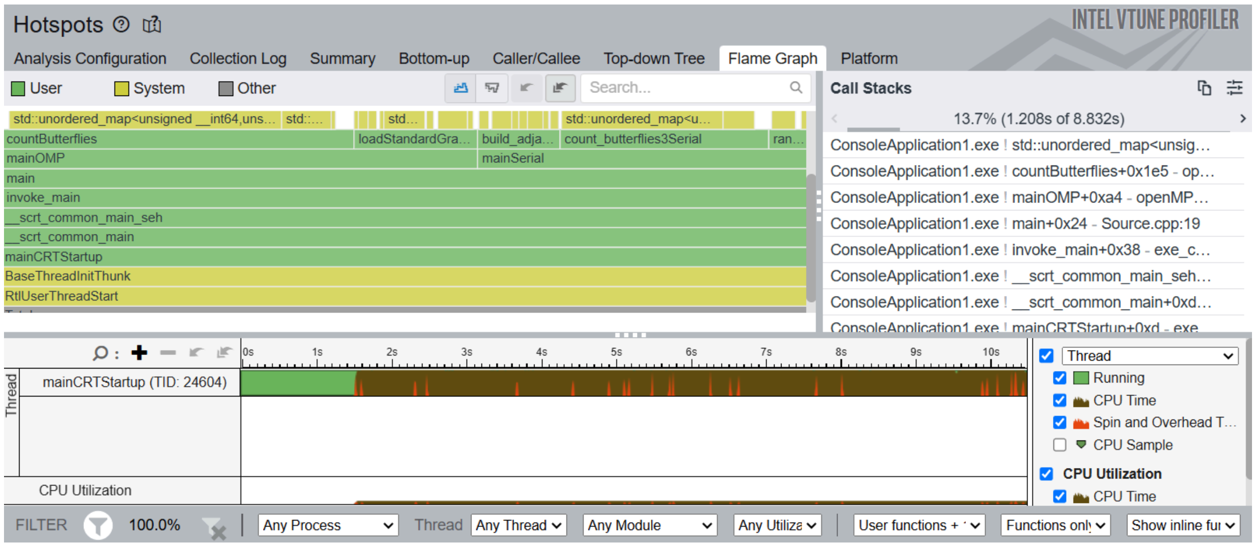
The picture above is for 4 threads.



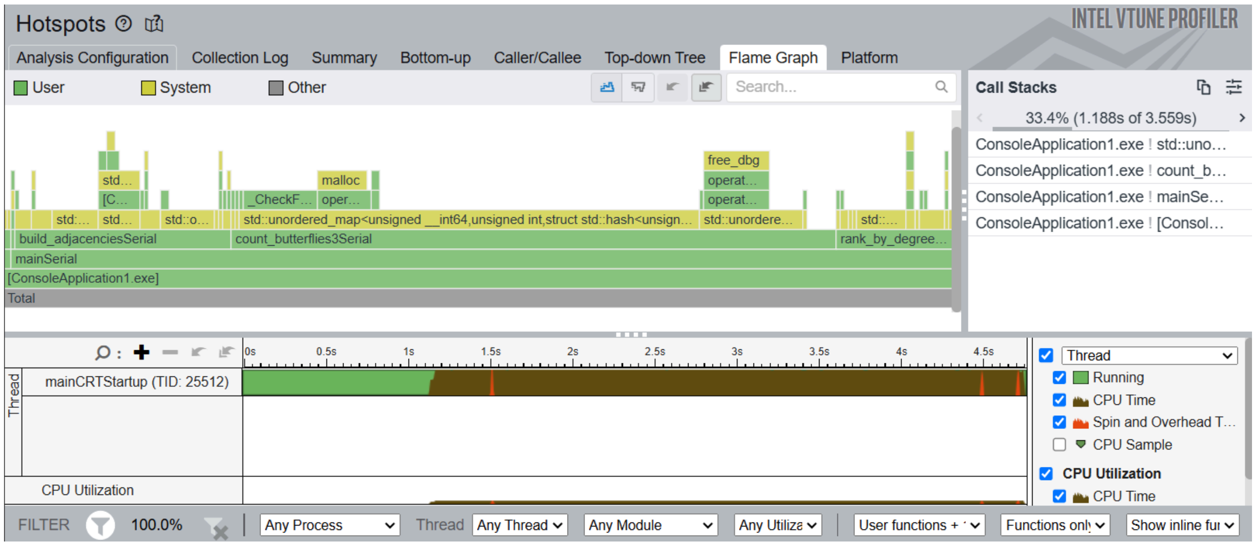
The picture above is for 8 threads.

As this single process is running both serial and parallel versions of the code in the same run, the stark contrast produced in the process memory usage clearly shows the impact this has on the resources of the computer.
CPU usage increases in more threads and so does Process Memory due to its own overhead.

Some profiling hotspot analysis results for OpenMP and Serial implementations with Vtune are given below as well:

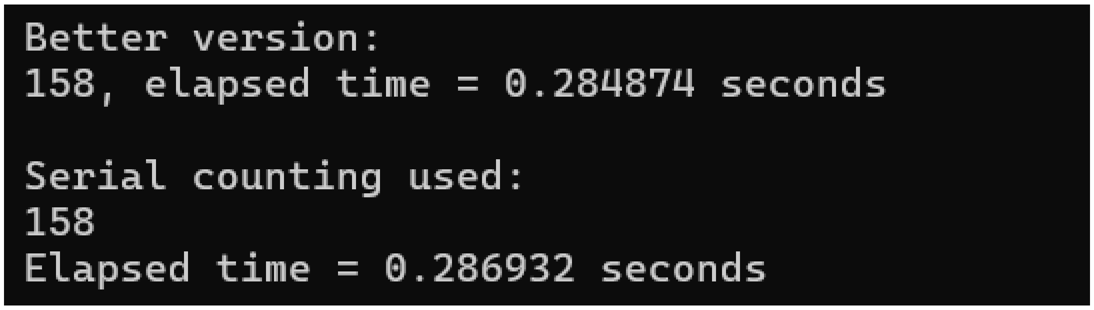


```
13.7% (1.208s of 8.832s)
ConsoleApplication1.exe ! std::unordered_map<unsig…
ConsoleApplication1.exe ! countButterflies+0x1e5 - op…
ConsoleApplication1.exe ! mainOMP+0xa4 - openMP…
ConsoleApplication1.exe ! main+0x24 - Source.cpp:19
ConsoleApplication1.exe ! invoke_main+0x38 - exe_c…
ConsoleApplication1.exe ! __scrt_common_main_seh…
ConsoleApplication1.exe ! __scrt_common_main+0xd…
ConsoleApplication1.exe ! mainCRTStartup+0xd - exe
```



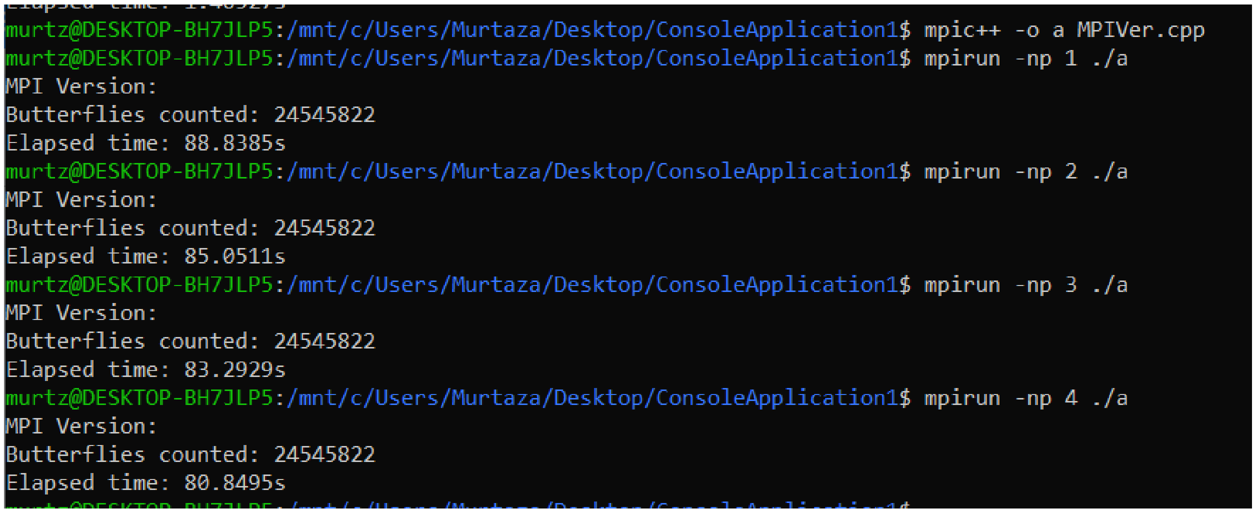
```
33.4% (1.188s of 3.559s)
ConsoleApplication1.exe ! std::uno…
ConsoleApplication1.exe ! count_b…
ConsoleApplication1.exe ! mainSe…
ConsoleApplication1.exe ! [Consol…
```

Example Output Results



```
Better version:
158, elapsed time = 0.284874 seconds

Serial counting used:
158
Elapsed time = 0.286932 seconds
```



```
murtz@DESKTOP-BH7JLP5:/mnt/c/Users/Murtaza/Desktop/ConsoleApplication1$ mpic++ -o a MPIVer.cpp
murtz@DESKTOP-BH7JLP5:/mnt/c/Users/Murtaza/Desktop/ConsoleApplication1$ mpirun -np 1 ./a
MPI Version:
Butterflies counted: 24545822
Elapsed time: 88.8385s
murtz@DESKTOP-BH7JLP5:/mnt/c/Users/Murtaza/Desktop/ConsoleApplication1$ mpirun -np 2 ./a
MPI Version:
Butterflies counted: 24545822
Elapsed time: 85.0511s
murtz@DESKTOP-BH7JLP5:/mnt/c/Users/Murtaza/Desktop/ConsoleApplication1$ mpirun -np 3 ./a
MPI Version:
Butterflies counted: 24545822
Elapsed time: 83.2929s
murtz@DESKTOP-BH7JLP5:/mnt/c/Users/Murtaza/Desktop/ConsoleApplication1$ mpirun -np 4 ./a
MPI Version:
Butterflies counted: 24545822
Elapsed time: 80.8495s
```

Team Contributions

Individual contributions are documented in the GitHub repository (linked in the report). Commit history reflects the following contributions:

Ahmed Murtaza: Dataset generation, graph reading, MPI workload distribution etc.
Ahmed Hannan: Serial and OpenMP implementations of the algorithm

Conclusion

Serial vs. OpenMP:

Small Datasets (e.g., Paper Dataset, 200 nodes): OpenMP significantly accelerates performance, achieving up to a 3.11x speedup with 4 threads. However, using 8 threads for the 200-node dataset caused a slowdown (0.102s vs. serial 0.062s), likely due to thread management overhead.

Medium/Large Datasets (5k–20k nodes): OpenMP provided modest speedups (1.29–1.30x) with 4–8 threads, but scalability diminished for larger graphs. The 20k-node dataset saw worse performance with 8 threads (196s vs. 107s with 4 threads), indicating resource contention or memory bandwidth limitations.

MPI + OpenMP Performance:

Small/Medium Datasets: MPI demonstrated strong scalability, with execution times improving as processes increased (e.g., 200-node dataset: 4.8x faster with 4 processes vs. serial). For the 5k-node dataset, 4 processes reduced runtime by 15% compared to 1 process.

Large Dataset (20k nodes): MPI performance degraded with >2 processes (91s for 4 processes vs. 78s for 2), suggesting communication overhead or load imbalance dominate at scale.

Combining MPI (distributed memory) with OpenMP (shared memory) optimized performance for large graphs.

GitHub Repository

The GitHub repository can be found at the following link: Repo

---