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

The dataset available at the ParButterfly papers' authors' GitHub, albeit very small, was used to confirm results. Due to its small size, we searched for larger datasets online.

However, 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
– 20000 nodes, 1000000 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, 2 Threads**	**Time: OpenMP, 4 Threads**	**Time: OpenMP, 8 Threads**	**Speedup Achieved**
Paper Dataset 800 nodes, less edges	0.6198s	0.4664s	0.0231s	0.01695s	36.67x
200 nodes, 500 edges	0.062s	0.01160s	0.0199s	0.102s	5.34x
5k nodes, 100k edges	7.511s	5.34762s	5.228s	5.75s	1.43x

20k nodes, 1M edges	138s	162.293s	107.245s	196.056s	1.29x

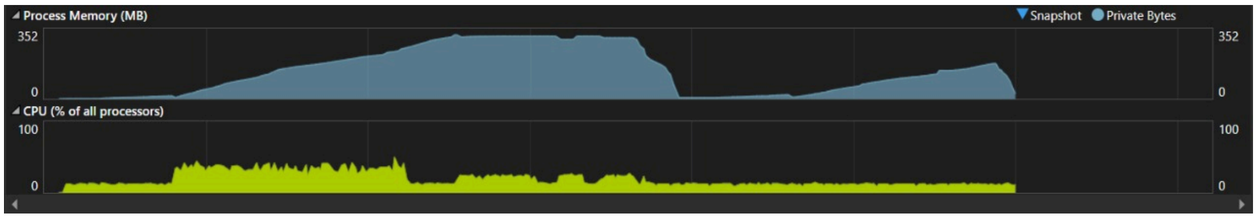
Times for varying MPI processes have also been compared here:

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

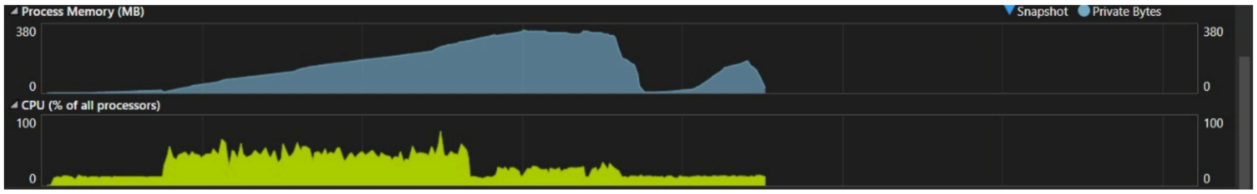
Application Profile & Analysis

Resource Utilization

Microsoft Visual Studio's built in performance profiler was used to get flame graphs signifying memory and CPU usage over the program's lifetime.



The picture above is for 4 threads.



The picture above is for 8 threads.

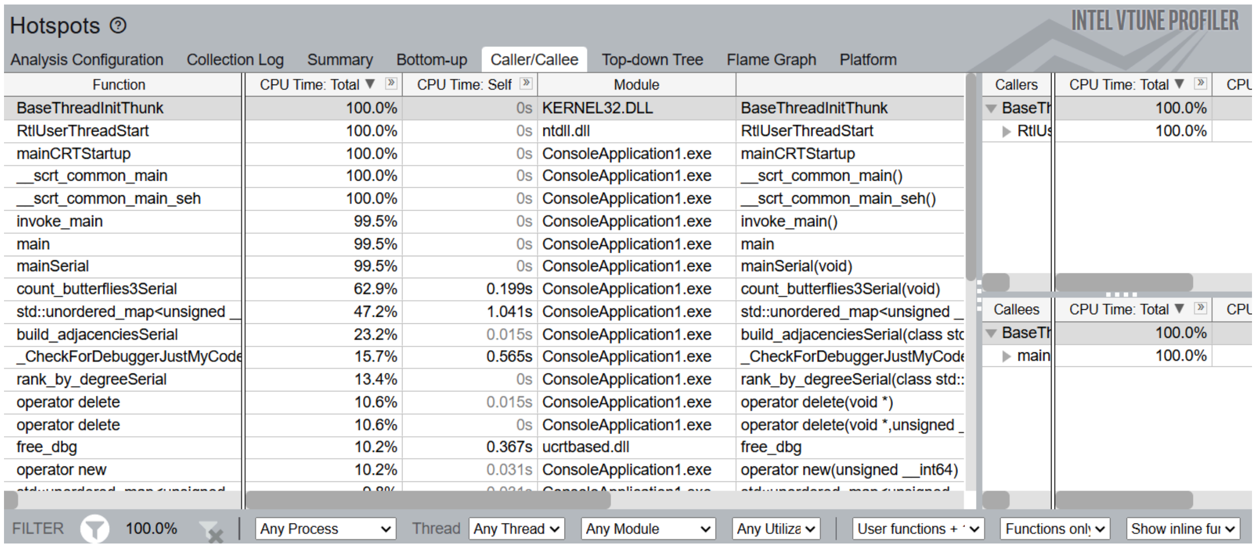
As this single process is running both serial and parallel versions of the code in the same run, the stark contrast produced in the process memory usage clearly shows

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.

Hotspot Analysis (Serial Code)

To identify potential bottlenecks and compute intensive areas in the program, hotspot analysis using Intel Vtune was performed for the 5k nodes, 100k edges data and the 20k nodes, 1M edges data.

For 5k Nodes, 100k Edges:



Hotspots

INTEL VTUNE PROFILER

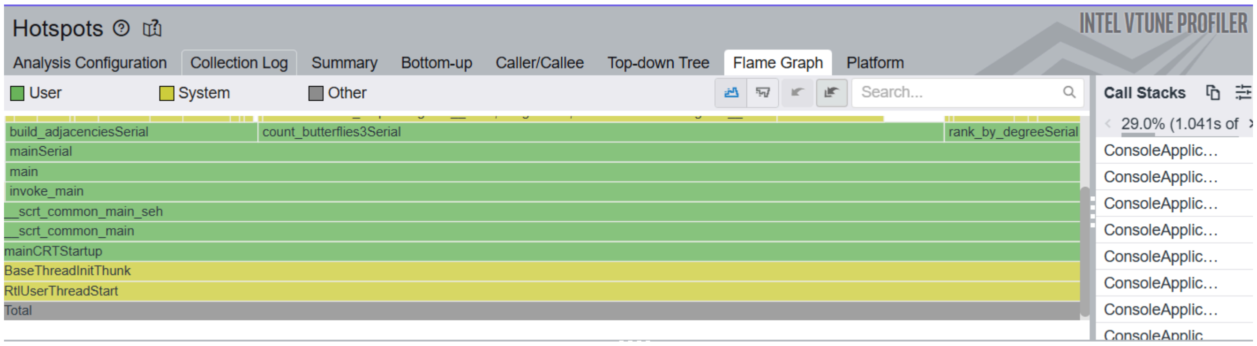
Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Flame Graph | Platform

Function	CPU Time: Total ▼	CPU Time: Self	Module	
BaseThreadInitThunk	100.0%	0s	KERNEL32.DLL	BaseThreadInitThunk
RtlUserThreadStart	100.0%	0s	ntdll.dll	RtlUserThreadStart
mainCRTStartup	100.0%	0s	ConsoleApplication1.exe	mainCRTStartup
__scrt_common_main	100.0%	0s	ConsoleApplication1.exe	__scrt_common_main()
__scrt_common_main_seh	100.0%	0s	ConsoleApplication1.exe	__scrt_common_main_seh()
invoke_main	99.5%	0s	ConsoleApplication1.exe	invoke_main()
main	99.5%	0s	ConsoleApplication1.exe	main
mainSerial	99.5%	0s	ConsoleApplication1.exe	mainSerial(void)
count_butterflies3Serial	62.9%	0.199s	ConsoleApplication1.exe	count_butterflies3Serial(void)
std::unordered_map<unsigned _	47.2%	1.041s	ConsoleApplication1.exe	std::unordered_map<unsigned _
build_adjacenciesSerial	23.2%	0.015s	ConsoleApplication1.exe	build_adjacenciesSerial(class stc
_CheckForDebuggerJustMyCode	15.7%	0.565s	ConsoleApplication1.exe	_CheckForDebuggerJustMyCode
rank_by_degreeSerial	13.4%	0s	ConsoleApplication1.exe	rank_by_degreeSerial(class std::
operator delete	10.6%	0.015s	ConsoleApplication1.exe	operator delete(void *)
operator delete	10.6%	0s	ConsoleApplication1.exe	operator delete(void *,unsigned _
free_dbg	10.2%	0.367s	ucrtbased.dll	free_dbg
operator new	10.2%	0.031s	ConsoleApplication1.exe	operator new(unsigned __int64)

Callers	CPU Time: Total ▼	CPU
▼ BaseTh	100.0%	
▶ RtlUs	100.0%	

Callees	CPU Time: Total ▼	CPU
▼ BaseTh	100.0%	
▶ main	100.0%	

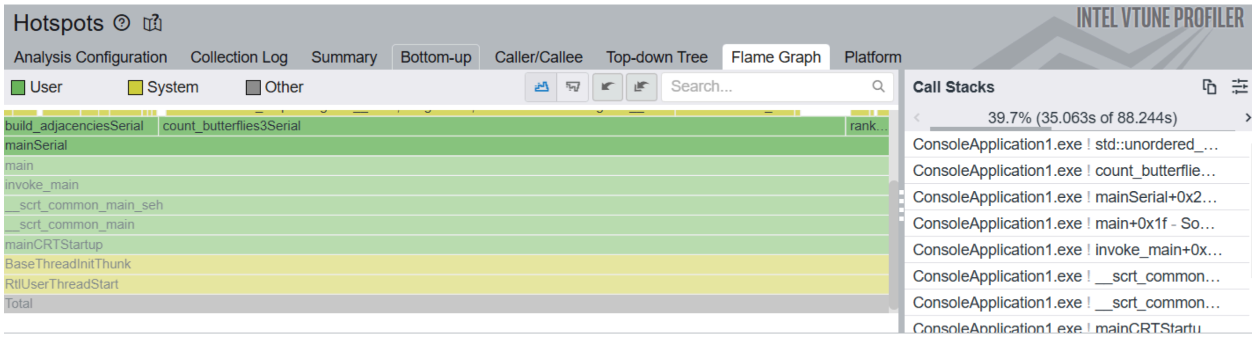
FILTER 100.0% | Any Process | Thread Any Thread | Any Module | Any Utiliz | User functions + | Functions only | Show inline fu



As observed in the screenshots, most of the CPU time was spent in building adjacencies and counting the butterflies.

62.9% of total time was spent in counting butterflies.
23.2% of total time was spent in building adjacencies.

For 5k Nodes, 100k Edges:


Hotspots

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Flame Graph | Platform

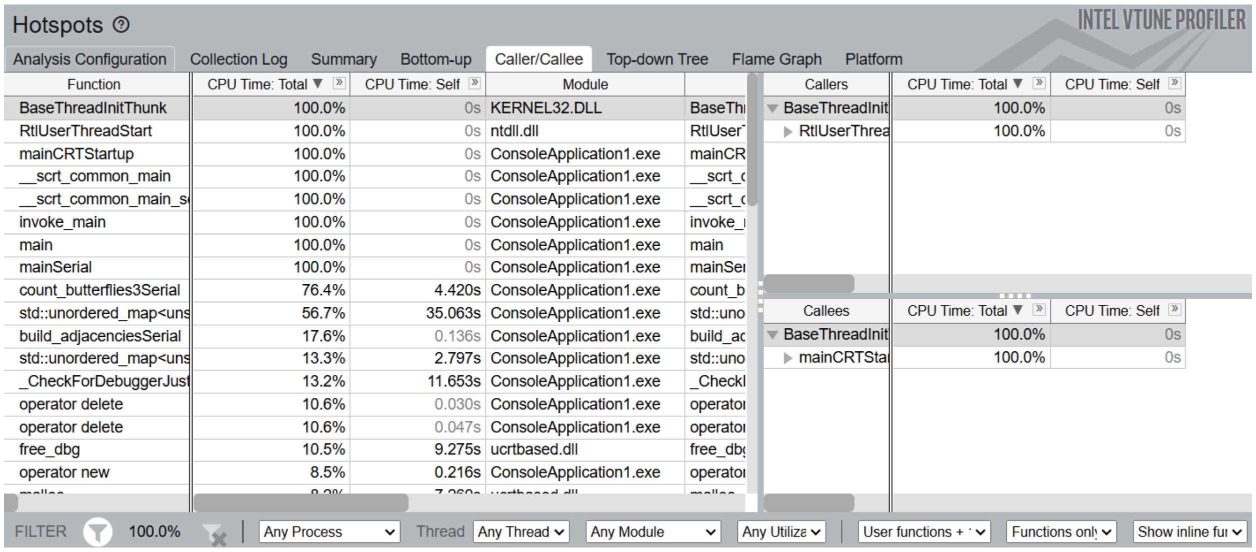
🟩 User 🟨 System ▨ Other

```
build_adjacenciesSerial | count_butterflies3Serial | rank...
mainSerial
main
invoke_main
__scrt_common_main_seh
__scrt_common_main
mainCRTStartup
BaseThreadInitThunk
RtlUserThreadStart
Total
```

Call Stacks

39.7% (35.063s of 88.244s)

```
ConsoleApplication1.exe ! std::unordered_...
ConsoleApplication1.exe ! count_butterflie...
ConsoleApplication1.exe ! mainSerial+0x2...
ConsoleApplication1.exe ! main+0x1f - So...
ConsoleApplication1.exe ! invoke_main+0x...
ConsoleApplication1.exe ! __scrt_common...
ConsoleApplication1.exe ! __scrt_common...
ConsoleApplication1.exe ! mainCRTStartu
```


Hotspots

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Flame Graph | Platform

Function	CPU Time: Total	CPU Time: Self	Module	
BaseThreadInitThunk	100.0%	0s	KERNEL32.DLL	BaseThı
RtlUserThreadStart	100.0%	0s	ntdll.dll	RtlUser
mainCRTStartup	100.0%	0s	ConsoleApplication1.exe	mainCR
__scrt_common_main	100.0%	0s	ConsoleApplication1.exe	__scrt_
__scrt_common_main_s	100.0%	0s	ConsoleApplication1.exe	__scrt_
invoke_main	100.0%	0s	ConsoleApplication1.exe	invoke_
main	100.0%	0s	ConsoleApplication1.exe	main
mainSerial	100.0%	0s	ConsoleApplication1.exe	mainSeı
count_butterflies3Serial	76.4%	4.420s	ConsoleApplication1.exe	count_b
std::unordered_map<uns	56.7%	35.063s	ConsoleApplication1.exe	std::uno
build_adjacenciesSerial	17.6%	0.136s	ConsoleApplication1.exe	build_ac
std::unordered_map<uns	13.3%	2.797s	ConsoleApplication1.exe	std::uno
_CheckForDebuggerJust	13.2%	11.653s	ConsoleApplication1.exe	_Checkl
operator delete	10.6%	0.030s	ConsoleApplication1.exe	operatoı
operator delete	10.6%	0.047s	ConsoleApplication1.exe	operatoı
free_dbg	10.5%	9.275s	ucrtbased.dll	free_dbı
operator new	8.5%	0.216s	ConsoleApplication1.exe	operatoı

Callers	CPU Time: Total	CPU Time: Self
BaseThreadInit	100.0%	0s
RtlUserThrea	100.0%	0s

Callees	CPU Time: Total	CPU Time: Self
BaseThreadInit	100.0%	0s
mainCRTSta	100.0%	0s

FILTER 100.0% | Any Process | Thread Any Thread | Any Module | Any Utiliza | User functions + | Functions only | Show inline fu

It was observed with the increase of data that the main compute intensive function is the butterfly counting, and as nodes and edges are increased, building adjacencies takes less time and counting butterflies takes more time, as shown by the following percentages:

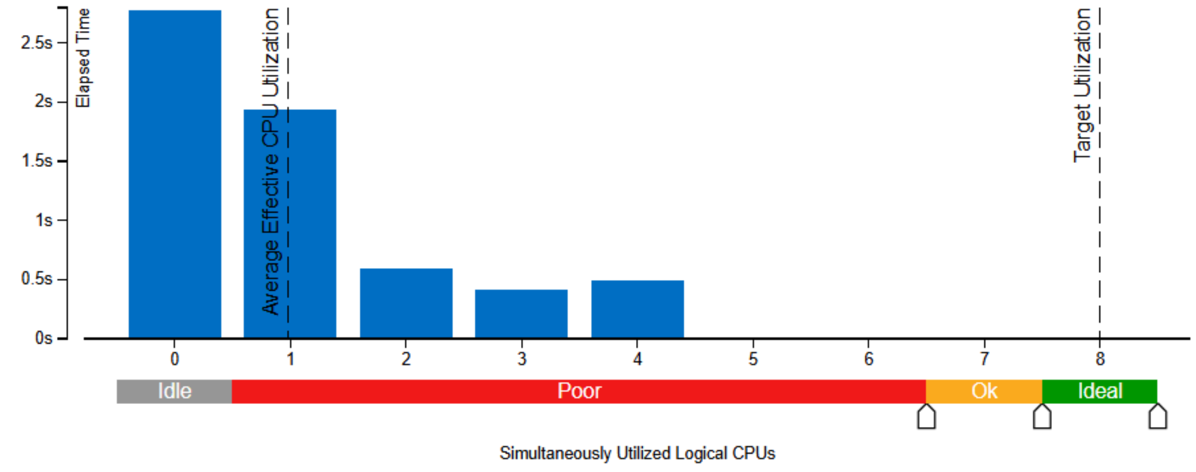
76.4% time was spent counting butterflies: A 13.5% increase from the smaller dataset.
17.6% time was spent building adjacencies: A 5.6% decrease from the smaller dataset.

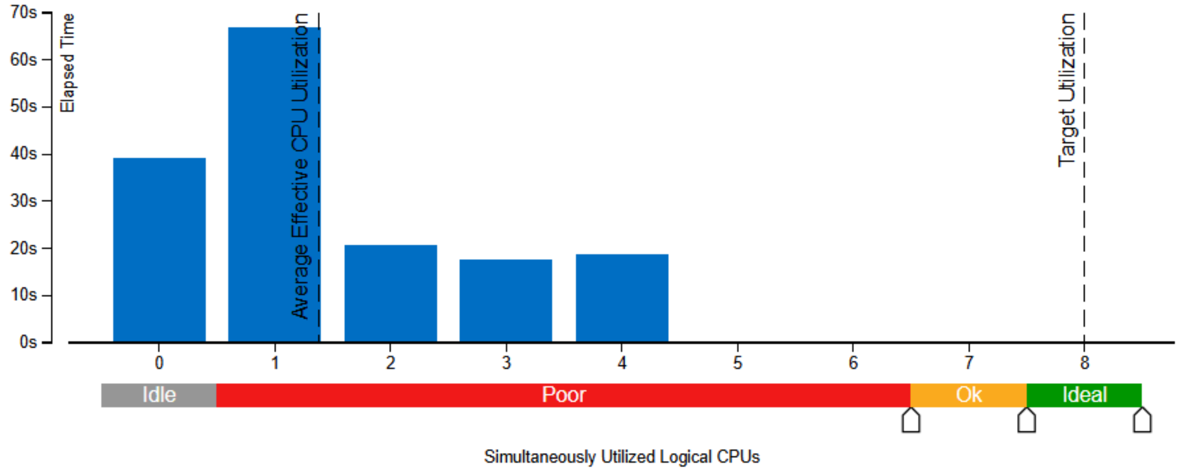
Thus it is prevalent that increasing the data size affects butterfly counting time adversely and when we transition from serial code to multithreaded code, it is this function that we must look to parallelize.

Threading Analysis (OpenMP)

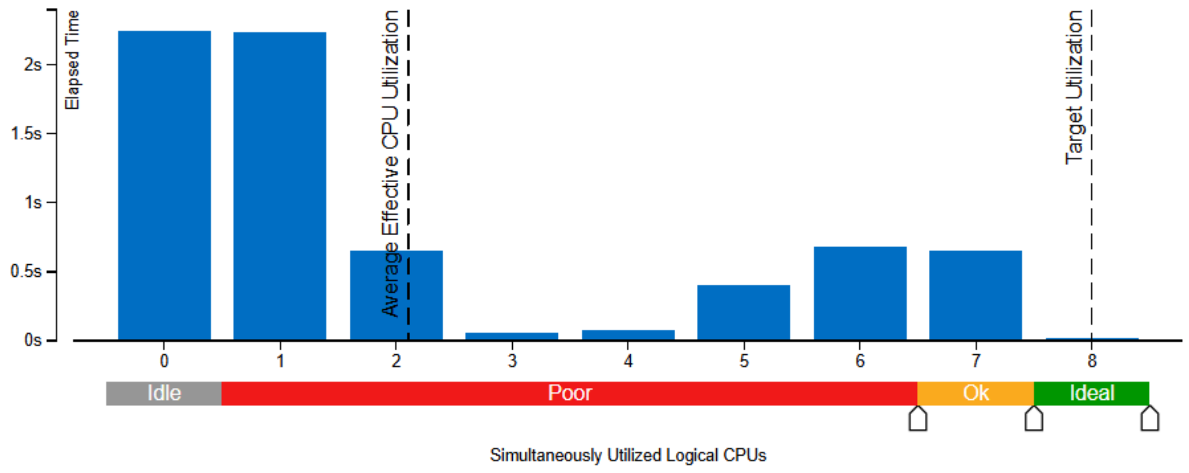
To evaluate the threading performance of the application, the program was tested with datasets of 5k nodes and 20k nodes, using 4 threads and 8 threads, respectively. The performance profiler tools provided visual outputs, including CPU utilization histograms per thread and a Gantt chart depicting thread activity over time.

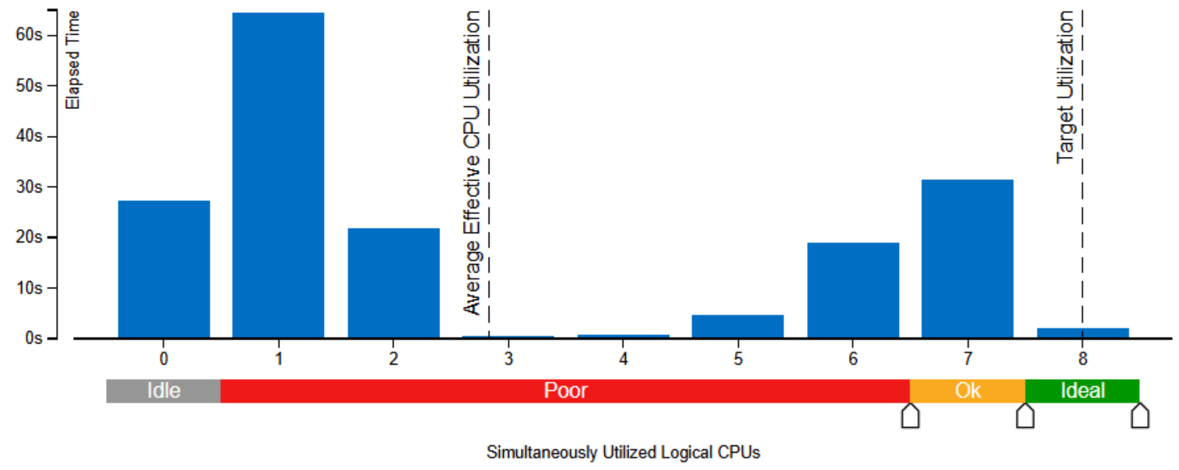
CPU Utilization:
These are the histograms that show 4 threads' usage for the smaller and larger data respectively.





These are the histograms that show 8 threads' usage for the smaller and larger data respectively.





The histograms show that with an increase in the number of threads, the CPU utilization becomes more evenly distributed among the available cores up until a certain number of threads, owing to context switching or overhead bottlenecks.

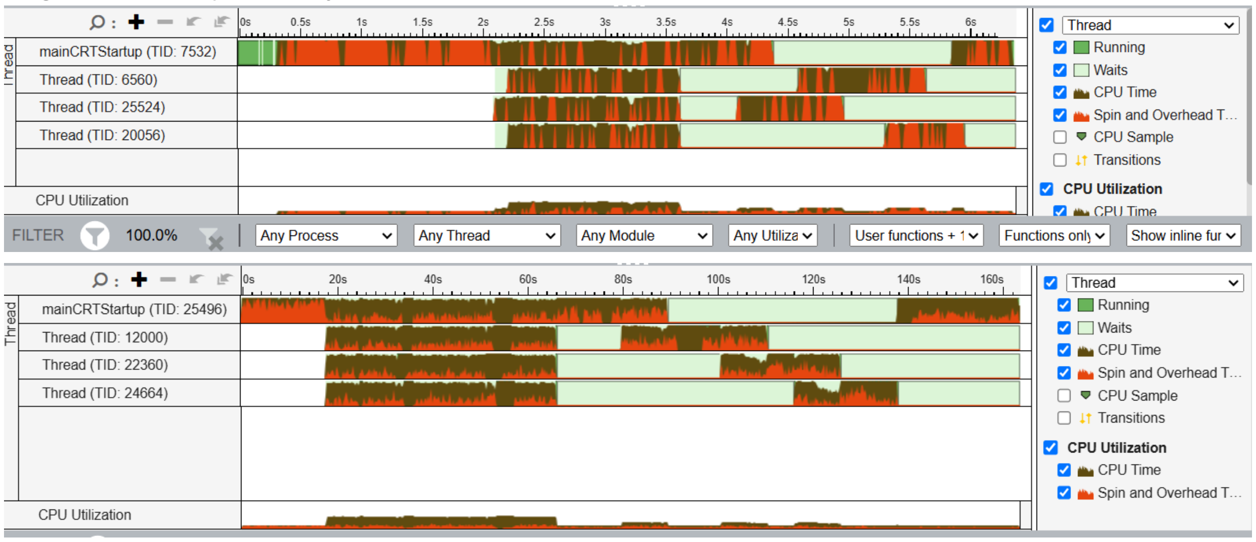
– In the case of 4 threads, the workload is reasonably divided, although out of the 4 available threads, one thread utilized more time than the other 3.

– In contrast, the 8-thread execution shows utilization across all threads during parallel segments. However, the increased thread count resulted in utilization times that became less uniform and more sporadic.

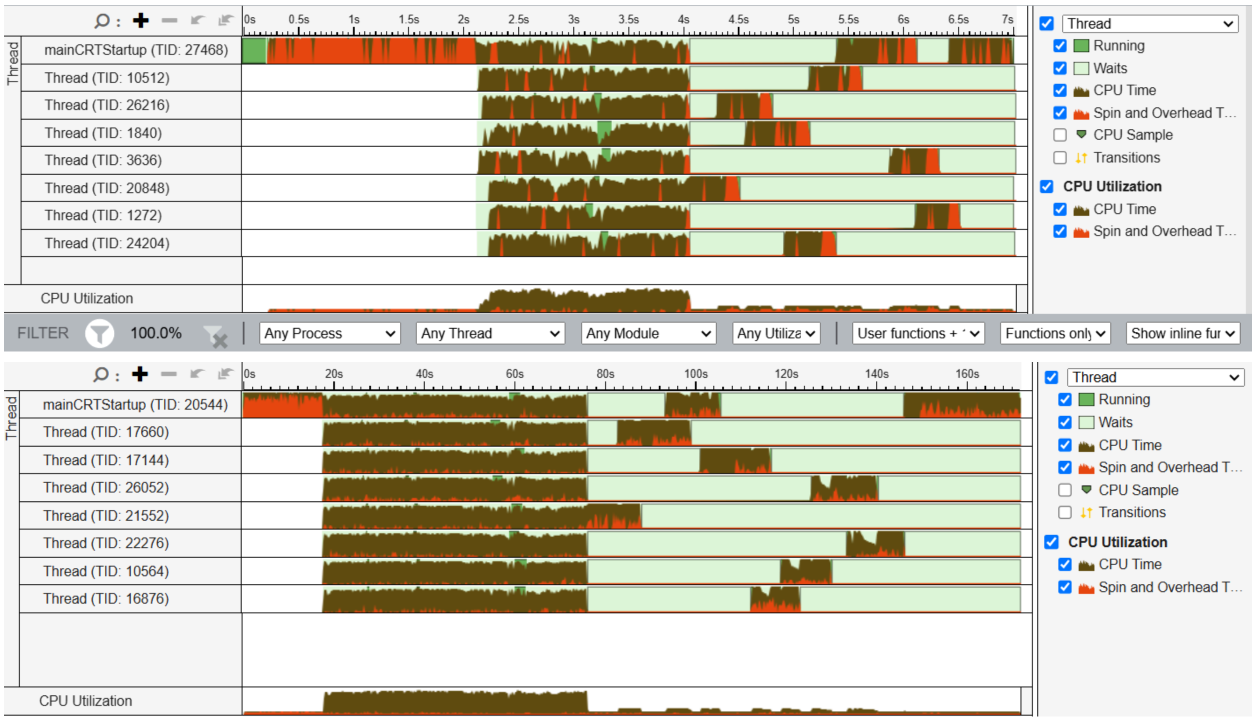
– The causes of this non-uniformity become more clear when we perform thread activity analysis, which will be done below.

Thread Activity (Gantt Charts):
The Gantt charts illustrate the thread-level execution timeline, providing insight into thread scheduling and workload balance.

These are the Gantt charts showing thread activity with time for the smaller and larger data respectively, for 4 threads:



These are the Gantt charts showing thread activity with time for the smaller and larger data respectively, for 8 threads:



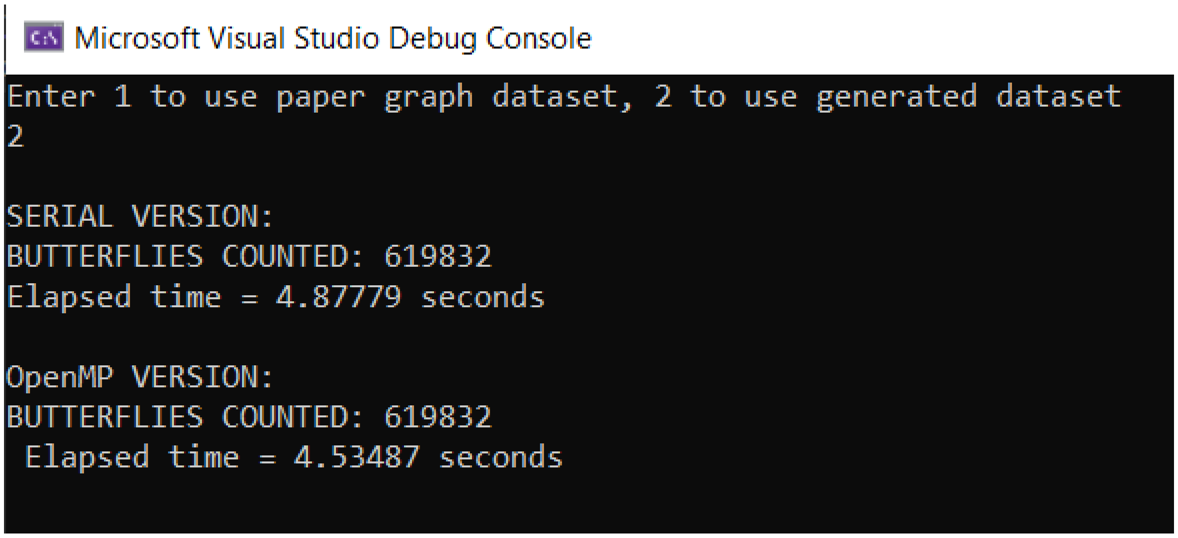
Gantt charts reveal that:
– In the 4-thread runs, the larger per-thread workload results in longer, more consistent parallel sections, minimizing synchronization delays and thread idling. This led to less overall execution time despite the slightly higher overhead.

– With 8 threads, the workload gets fragmented into smaller portions. While overhead per thread is lower, this fragmentation introduces frequent synchronization points, increased idle periods, and greater imbalance, resulting in longer overall execution time.

– Interestingly, as the dataset grows from 5k to 20k nodes, thread overhead tends to decrease. This is likely because larger datasets offer longer and more substantial parallel regions, giving threads more consistent work to do and reducing the impact of coordination and scheduling overhead.

4 threads provide better performance under current conditions, but larger datasets may help unlock the potential of higher thread counts, provided synchronization is further optimized. Due to hardware restrictions, exponentially larger datasets (>10M Edges) were not tested as they would take a significant amount of time to test.

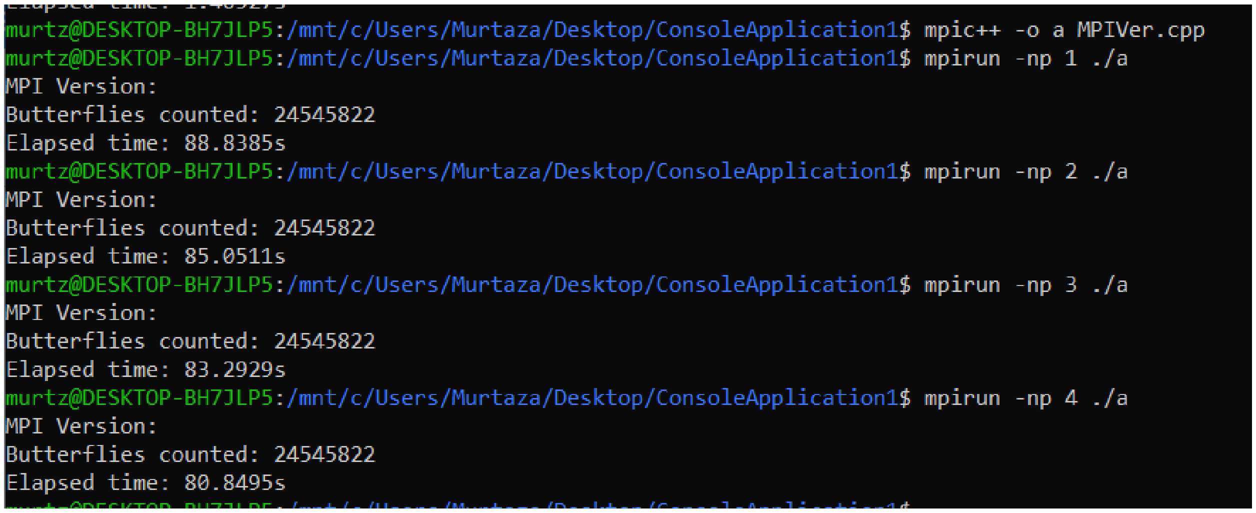
Example Output Results



```
Enter 1 to use paper graph dataset, 2 to use generated dataset
2

SERIAL VERSION:
BUTTERFLIES COUNTED: 619832
Elapsed time = 4.87779 seconds

OpenMP VERSION:
BUTTERFLIES COUNTED: 619832
 Elapsed time = 4.53487 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

Report was a joint effort.

Conclusion

Serial vs. OpenMP:

Small Datasets (e.g., Paper Dataset, 200 nodes): OpenMP significantly accelerates performance, achieving up to a 5.34x 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 (1.29–1.43x) 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