BTP Report

High Performance Big Data Analytics

Supervisor - Professor Dip Sankar Banerjee

Kshitij Jaiswal (B20CS028)

Anirudh Srikanth (B20CS006)

Overview -

BFS (Breadth First Search) is a graph search which explores a graph uniformly in all the directions. With unweighted edges, BFS can also be used to find minimum distance from source to all other vertices (Single Pair Shortest Path). This makes BFS an important precursor in Machine Learning Algorithms. Hence speeding up BFS using parallelized algorithms will improve the overall performance of the algorithm.

Big graphs (graphs from AWS, Meta etc.) are small world and scale free graphs. Small world graphs are the graphs with some upper bound on the diameter of the graph. Small world graphs exhibit high local clustering of nodes combined with short average path lengths, facilitating efficient information transfer. Scale-free graphs have a power-law degree distribution, meaning that a few nodes (hubs) have significantly more connections than others. Conventional BFS algorithms can be adapted to best fit the traversal needs for these graphs.

In this project, we have combined two approaches namely top-down (conventional) and bottom-up BFS. We have switched from one to the other based on certain conditions later discussed in the report. Combination of these approaches is called the hybrid BFS.

We have parallelized the hybrid BFS first using OpenMP and then using CUDA. We have compared the results from Sequential BFS, Hybrid BFS using OpenMP and Hybrid BFS using CUDA.

Brief on BFS approaches -

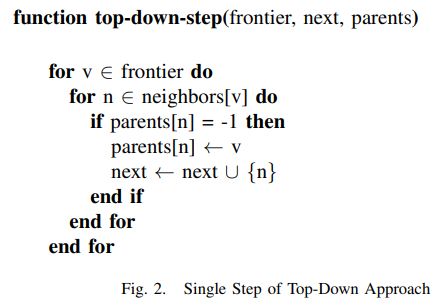
1. Top Down BFS

Top Down BFS visits the neighbor of every node in the current frontier and if the neighbor is unvisited then the neighbor is added to the next frontier. This is how BFS is implemented conventionally.

Now let’s imagine a point in traversal where the current frontier is very large such that almost all the nodes in the graph are neighbors of some node in the current frontier. Top Down BFS explores all the edges from the current frontier to the rest of the graph, this takes O(n+m) time where n is the number of nodes and m is the number of edges. For a graph with m ~ n2, running time becomes O(n2). However, this time can be reduced to O(n) using Bottom Up BFS.

Here O(n) is the traversal time of the current frontier and NOT the entire graph.

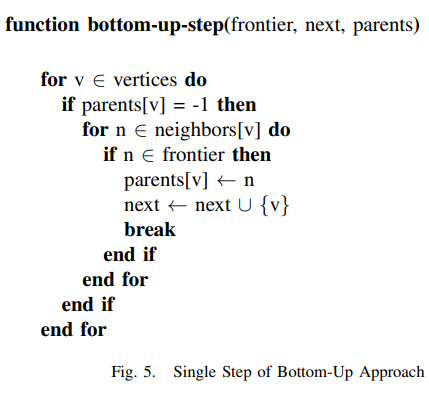
For the entire graph



1. Bottom Up BFS -

Bottom Up BFS is ideal when the frontier size is large. In Bottom Up BFS, we traverse all the nodes in the graph and check if for a given node its successor is in the current frontier. For this step, bottom up brings down the complexity of traversal of the current frontier from O(n+m) to O(n) reducing the time complexity for the traversal of the entire graph.

However, Bottom-Up BFS when run for the entire graph runs in O(n\*m) time.



To test our algorithms, we chose \*k-regular graphs.

For k-regular graphs, the upper bound on the diameter is n/k.

A hint for the proof of the same can be found [here](https://math.stackexchange.com/questions/351945/diameter-of-k-regular-graph).

Detailed proofs can be found [here](https://core.ac.uk/download/pdf/82105622.pdf).

When we parallelize bottom up or top down, the parallel algorithm runs for a number of frontier times and each call takes at most max(degi) iterations where degi is the degree of node i.

Parallel Running time = number of frontiers \* max(degi)

= diameter of the graph \* max(degi)

= (n/k) \* k = n

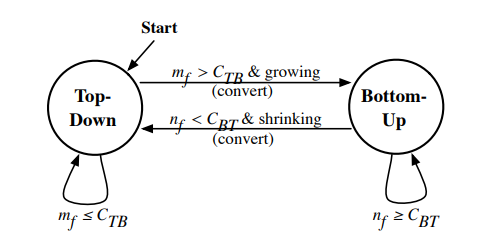
Therefore, parallel versions of bottom up and top down both run in O(n).

|  | Sequential | Parallel |
| --- | --- | --- |
| Top Down | O(n+m) | O(n) |
| Bottom Up | O(n\*m) | O(n) |

\* A k-regular graph is a graph in which each vertex has exactly k neighbors or connections, resulting in a uniform degree distribution across all nodes.

1. Hybrid BFS -

Hybrid BFS combines bottom up and top down approaches and switches back and forth based on certain conditions. Here are the conditions –



Here mf are the number of edges from the nodes in the current frontier to the nodes NOT in the current frontier (Cut from S to V/{S} where S is the set of vertices of in the current frontier and V is the set of all the vertices in the graph).

nf are the number of nodes in the current frontier.

where CBT = n/β and CTB = mu/α .

mu is the sum of degrees of unvisited vertices.

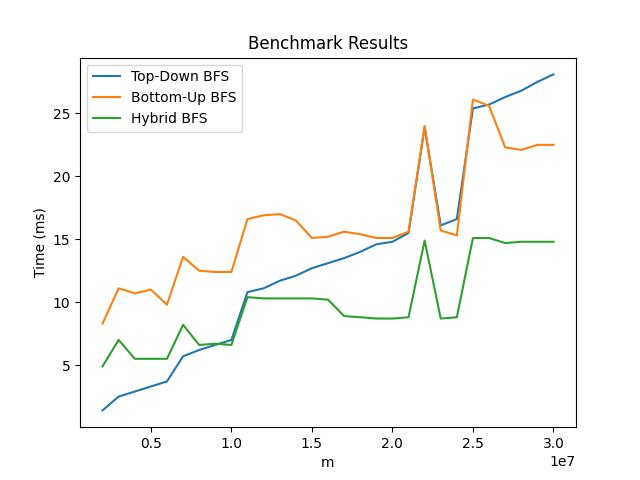
α and β are constants determined for optimal performance in [this](https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6468458) paper. We have used these values α = 14 and β = 24.

These conditions are best suited to imply growth and shrinkage of the frontier while performing BFS.

Parallelization using OpenMP -

OpenMP (Open Multi-Processing) is an application programming interface (API) that facilitates parallel programming in shared-memory systems. It provides a set of compiler directives and library routines for expressing parallelism in programs written in C, C++, and Fortran. With OpenMP, we can easily parallelize sequential code by adding directives to it, indicating where parallel execution should occur.

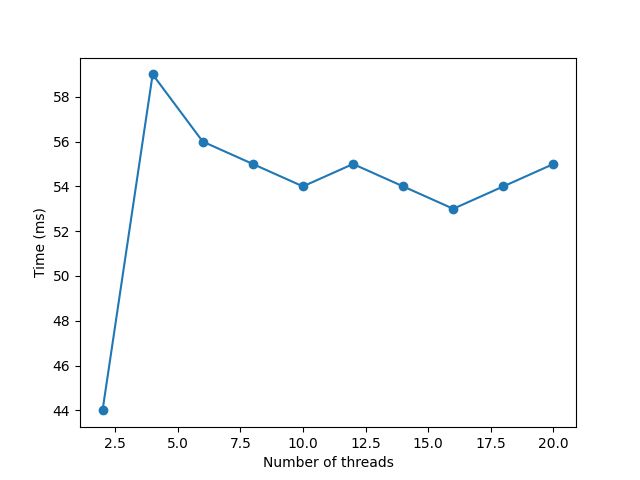
We implemented hybrid BFS on OpenMP with a maximum thread limit of 20 threads. OpenMP showed improved performance as compared to sequential BFS. for 60 M edges, OpenMP performs 4.35 times faster than traditional BFS



The above graph shows the running time of graphs of various sizes when run with top down, bottom up and hybrid approaches using OpenMP.

Initially Top Down performs the best but for larger graphs it is outperformed by Hybrid.

Hybrid BFS always performs better than Bottom Up BFS.



The above graph shows the performance of a graph with 2M nodes and 12M edges with varying number of threads. The number of threads were varied in the range [2, 20] with a step of 2.

Performance time decreases with an increasing number of threads.

Parallelization using CUDA –

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA for their Graphics Processing Units (GPUs). It enables developers to harness the computational power of GPUs for general-purpose computing tasks beyond graphics rendering. CUDA allows programmers to write parallel code using a C-like programming language and extend it with GPU-specific primitives.

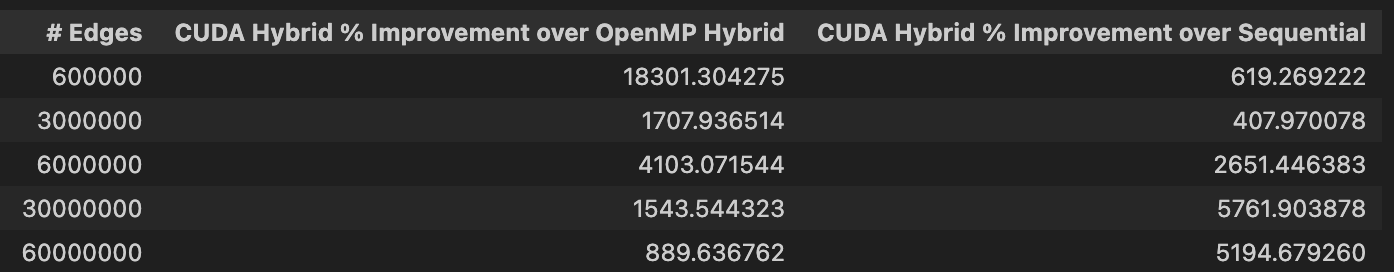
One of the important benefits of CUDA over OpenMP is that it uses \_\_syncthreads() to handle contention while concurrently executing. The \_\_syncthreads() method is a lot easier to use than the methods available in OpenMP for the same purposes.

In CUDA, we had 231 blocks, each block had 210 threads. Hence the total number of threads were 241 ~ 2.2 X 1012 threads. Therefore, we had sufficient threads to assign each thread to one node.

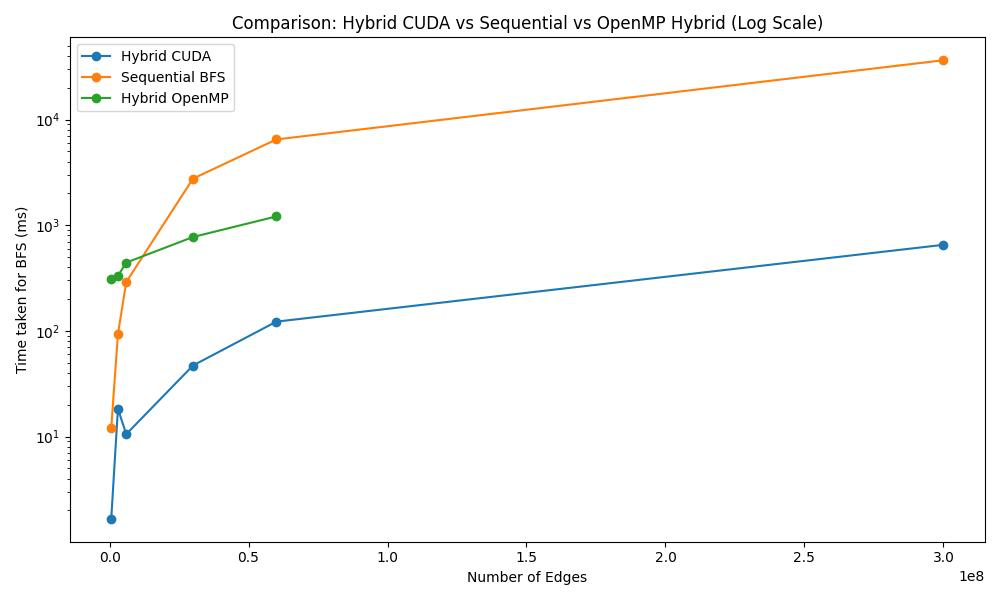
Hence our hybrid DFS implementation in CUDA has threads equal to the number of vertices in the graph.

CUDA showed significant improvements in the BFS traversal times. For a graph with 30 M edges, CUDA was 15.43 times faster than OpenMP and 57.62 times faster than sequential BFS.

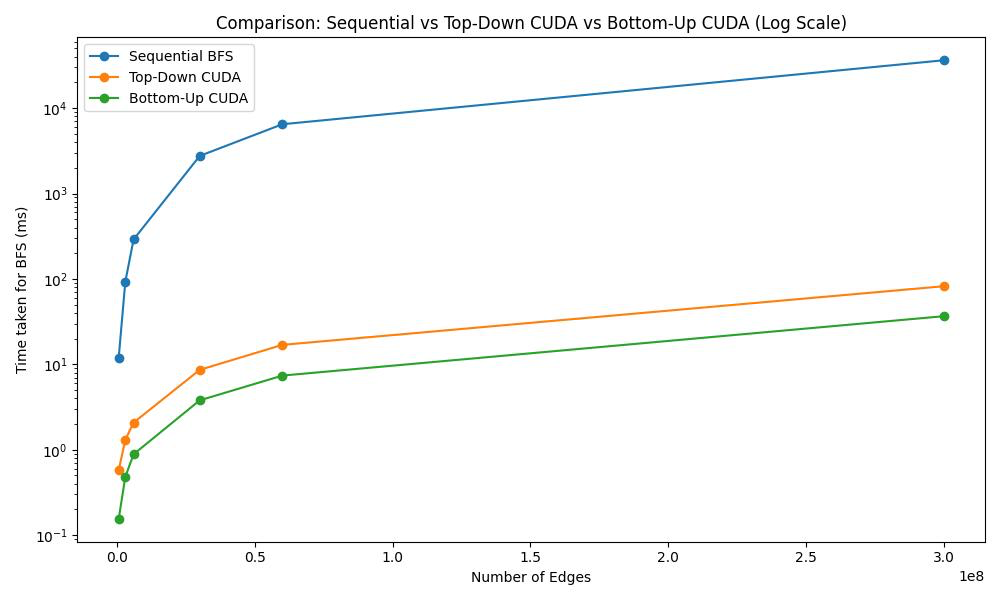
Results for CUDA -



The above table shows the percentage improvements for CUDA from Sequential and OpenMP Hybrid BFS implementations. These values were obtained after averaging out runtimes for 10 iterations. Maximum percentage standard deviation in the runtime was ±2% for OpenMP. There were no significant deviations for CUDA and sequential BFS.



Hybrid OpenMP could not be run for large graphs because of limited heap memory allocation.



Bottom Up CUDA even outperforms Top Down CUDA because of less memory contentions.

The Github Repository of our implementations can be found here.

<https://github.com/yeetholmes619/Parallel-BFS/tree/main>

References –

1. <https://core.ac.uk/download/pdf/82105622.pdf>
2. <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6468458>
3. <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8334786&tag=1>
4. <https://math.stackexchange.com/questions/351945/diameter-of-k-regular-graph>