INF236 - Assignment 2

Kristian Sørdal

March 23, 2024

Contents

1 Introduction

In this assignment, we are experimenting with two different strategies for paralellizing breadth first search. There will be two different parallel algorithms implemented.

Breadth first search or just BFS, is a classical graph exploration algorithm, where the input is given as a graph $G(V, E)$, and some starting node $n \in V$. We can then use BFS to figure out various properties about the graph, including (but not limited to) finding the shortest path between the starting node and some other node, the path used to get from one node to another node, and the amount of nodes that are reachable from a given node, and consequently whether or not the graph is connected.

BFS explores the graph layer by layer, which means that from the starting node, all of the neighbours of this node are added to a queue, and the distance d to these nodes will be equal to 1 plus the distance from the starting node. This process is repeated for these nodes neighbours once again, and so on until there are no more nodes to explore (the queue is empty). We also need to keep track of which nodes we have seen, and if we encounter a node we have already seen, we need to check if we have found a shorter distance to this node, and update that nodes distance accordingly.

2 Parallel BFS

Two different verions of parallel BFS have been implemented for this assignment, with two different approaches to the parallelization.

2.1 First implementation

When parallelizing BFS, we can observe that nodes are found layer by layer. So given a start node with distance 0, we will discover all its neighbours. These nodes will have distance 1 from the start node, their neighbours again will have distance 2, and so on. This lends itself nicely to paralellizing. If we maintain all nodes found in a given layer in some queue, we can distribute this queue amongst the different threads, using the OpenMP construct `#pragma omp for`. Then, every thread can work locally on discovering neighbours of these nodes, and maintaining these found neighbours in a private list. We don't care about which parent a given node points to, as long as the distance from the start node is correct. This means that the first thread that discoveres a node without a parent pointer, will set the parent pointer of that node, even if some other node in the same layer is also a neighbour of this node.

After all nodes in the current layer have been discovered, we can for each thread, copy the private discovered list back into the global layer queue, and continue on, until there are no more nodes to discover.

To ensure we copy the nodes back into the global queue in the correct order, we need to keep track of how many nodes each thread has discovered, and where in the global queue they should be placed. We can do this by using the T array, where each thread will store the amount of nodes they have discovered in their private list. After this, we can calculate the offset for each thread using a prefix sum on the T array, and copy the nodes back into the global queue. The indices for each thread is given by $[\mathit{offset}, \mathit{offset} + \mathit{numDiscovered})$.

An outline of this approach is given below. Note that this routine is called from within a parallel environment, where the parameters to the function are shared between all threads, and variables declared inside the function are private to each thread.

Algorithm 1: Parallel BFS - First approach

Input : *G(V,E)* - Graph, *dist* - Distance Array, p - Parent Array, S - Global Queue, T - Temporary global queue
Output: *dist* and p, populated with distances and parent of all nodes

```
layerSize ← 1
numDiscovered ← 0
discovered ← list of size n
p[1] ← 0
dist[1] ← 0
S[0] ← 0
while layerSize ≠ 0 do
    barrier
    for i = 0 to layerSize do in parallel
        for v ∈ neighbours(S[i]) do
            if p[v] == -1 then
                dist[v] ← dist[S[i]] + 1
                p[v] ← S[i]
                discovered[numDiscovered++] ← v
    T[tid] ← numDiscovered
    barrier
    layerSize ← T[0]
    offset ← 0
    for i = 1 to numThreads do
        if i == tid then
            offset ← layerSize
        layerSize += T[i]
    if numDiscovered > 0 then
        S[offset : numDiscovered] = discovered
        numDiscovered ← 0
```

2.2 Second implementation

For the second implementation, the strategy is as follows. First, we will perform BFS sequentially until some condition is reached. This condition can for example be: *perform sequential BFS until the layer discovered contains* $n \cdot \#threads$ *nodes.*

After this, we distribute the newly discovered nodes into a local queue for each thread as evenly as possible. Then each thread will continue discovering all nodes with distance at most k away from the originally distributed layer.

Then, we put all nodes in this layer into the global queue, as we did in the first implementation, and redistribute the nodes amongst the ranks again, as evenly as possible. This is repeated until there are no more nodes to discover. A general outline of how this algorithm is implemented is given below.

Algorithm 2: Parallel BFS - Second approach

Input : *G(V,E)* - Graph, *dist* - Distance Array, p - Parent Array, S - Global Queue, T - Temporary global queue
Output: *dist* and p, populated with distances and parent of all nodes

```
stepsPerRound ← 4
threadID ← index of the current thread
layerSize ← 1
localLayerSize ← 0
localQueue ← list of size n
discovered ← list of size n
numDiscovered ← 0

p[1] ← 0
dist[1] ← 0
S[0] ← 0

Perform sequential BFS until #threads nodes are found, store found nodes at last layer in S,
 and layer size in T[0]
layerSize ← T[0]

Distribute nodes in S to local queues

while layerSize ≠ 0 do
    for i = 0 to stepsPerRound do
        barrier
        for j = 0 to localLayerSize do
            for v ∈ neighbours(S[j]) do
                if p[v] == -1 then
                    dist[v] ← dist[S[j]] + 1
                    p[v] ← S[j]
                    discovered[numDiscovered++] ← v
        Swap pointers of localQueue and discovered
        localLayerSize ← numDiscovered
        numDiscovered ← 0
    T[tid] ← localLayerSize
    barrier
    layerSize ← T[0]
    offset ← 0
    for i = 1 to numThreads do
        if i == tid then
            offset ← layerSize
        layerSize += T[i]

    barrier
    Copy localQueue to S starting at offset
    barrier

    Redistribute S evenly into local queues
```

3 Experiments

Graph	Sequential Time (s)
road_usa	1.739677
rgg_n_2_22_s0	1.042576
delaunay_n24	1.549098
hugebubbles-00020	4.052872

Table 1: Sequential Execution Times

3.1 PBFS

The following results are from running the first implementation of parallelized BFS on the provided graphs. Please note that no more than 64 threads were used, as using any more than this yielded poorer results, with 80 threads resulting in a ten fold decrease in execution time.

Threads	Parallel Time	Speedup
1	1.734203	1.00
2	1.200486	1.45
4	0.788950	2.20
8	0.461280	3.77
16	0.327205	5.31
32	0.357365	4.87
64	0.699687	2.48

Table 2: road_usa

Threads	Parallel Time	Speedup
1	1.081684	0.96
2	0.606074	1.72
4	0.335648	3.10
8	0.188359	5.54
16	0.124455	8.38
32	0.126939	8.22
64	0.214437	4.86

Table 3: rgg_n_2_22_s0

Threads	Parallel Time	Speedup
1	1.783102	0.87
2	1.025448	1.51
4	0.627425	2.47
8	0.355146	4.36
16	0.236014	6.56
32	0.246382	6.28
64	0.498225	3.11

Table 4: delaunay_n24

Threads	Parallel Time	Speedup
1	3.489007	1.16
2	1.882132	2.15
4	0.974213	4.16
8	0.523627	7.75
16	0.373559	10.85
32	0.392521	10.33
64	0.856313	4.73

Table 5: hugebubbles-00020

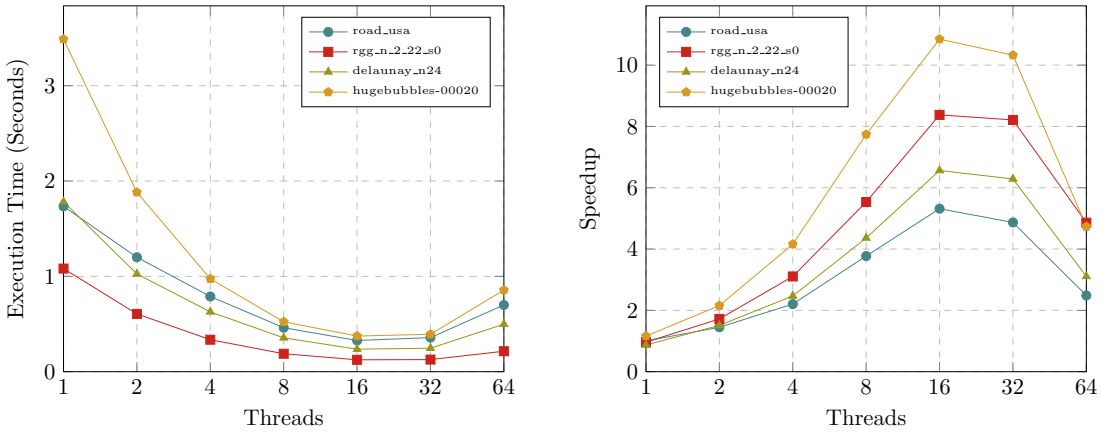

Figure 1: Plots of execution time and speedup of parallel BFS

3.2 ABFS

For the following results, please note that there were some issues with my implementation of this algorithm. Although all vertices were discovered, some vertices did not have the correct distance from the root vertex upon termination of the algorithm. This was due to some issue in the initial sequential part of this algorithm, as the issue was eliminated when the initial sequential part was omitted. Below is a table demonstrating the accuracy of the algorithm for various values of k. The accuracy was calculated by $\left(1 - \frac{\text{nodes w/ wrong distance}}{\text{num nonzeros}}\right) \cdot 100$

T / k	2	4	8	16
1	100%	100%	100%	100%
2	100%	87.08%	100%	100%
4	100%	100%	100%	100%
8	100%	100%	100%	100%
16	92.7%	92.7%	92.7%	92.7%
32	92.7%	92.7%	92.7%	92.7%
64	99.9%	99.9%	99.9%	99.9%

Table 6: Accuracy of ABFS on road_usa

T / k	2	4	8	16
1	100%	100%	100%	100%
2	100%	100%	100%	100%
4	100%	100%	100%	100%
8	100%	100%	100%	100%
16	99.9%	99.9%	99.9%	100%
32	100%	100%	99.9%	100%
64	100%	100%	99.9%	99.9%

Table 7: Accuracy of ABFS on rgg_n_2_22_s0

T / k	2	4	8	16
1	100%	100%	100%	100%
2	100%	100%	100%	100%
4	100%	99.9%	99.9%	99.9%
8	99.9%	100%	100%	99.9%
16	100%	99.9%	99.9%	99.9%
32	99.9%	99.9%	99.9%	99.9%
64	100%	99.9%	99.9%	100%

Table 8: Accuracy of ABFS on delaunay_n24

T / k	2	4	8	16
1	100%	100%	100%	100%
2	100%	80.8%	100%	100%
4	100%	100%	99.9%	100%
8	100%	100%	100%	100%
16	99.9%	99.9%	99.9%	100%
32	99.9%	99.9%	99.9%	99.9%
64	99.9%	99.9%	99.9%	99.9%

Table 9: Accuracy of ABFS on hugebubbles

Threads	Parallel Time	Speedup
1	1.598203	1.09
2	1.375186	1.26
4	0.763902	2.27
8	0.304963	5.70
16	0.265961	6.54
32	0.250951	6.93
64	0.312814	5.56

Table 10: road_usa $k = 8$

Threads	Parallel Time	Speedup
1	0.872044	1.20
2	0.570689	1.83
4	0.358203	2.91
8	0.204515	5.10
16	0.147258	7.08
32	0.171226	6.09
64	0.204075	5.11

Table 11: rgg_n_2_22_s0, $k = 8$

Threads	Parallel Time	Speedup
1	1.408174	1.10
2	0.994790	1.56
4	0.670101	2.31
8	0.348662	4.44
16	0.210787	7.34
32	0.154256	10.04
64	0.188006	8.23

Table 12: delaunay_n24, $k = 8$

Threads	Parallel Time	Speedup
1	3.461700	1.17
2	1.913450	2.12
4	1.027452	3.94
8	0.531675	7.62
16	0.327152	12.39
32	0.243940	16.63
64	0.285755	14.19

Table 13: hugebubbles-00020, $k = 8$

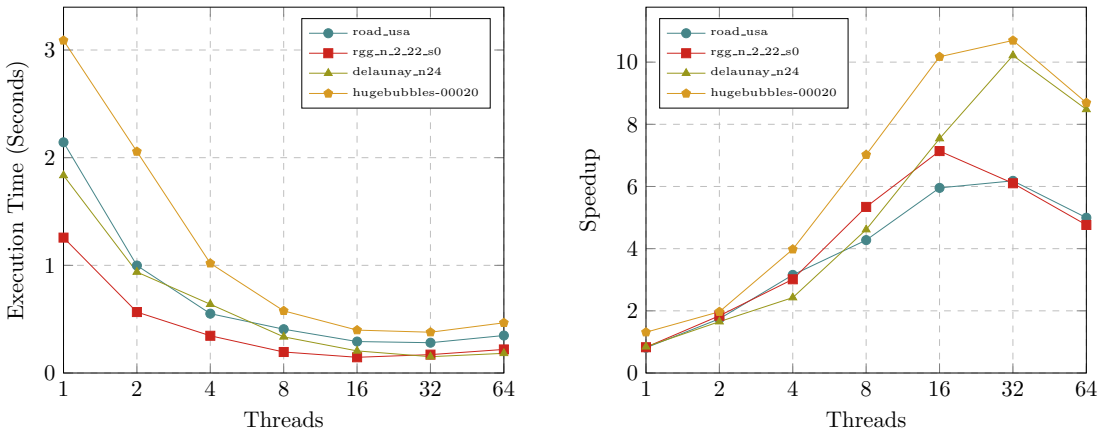

Figure 2: Plots of execution time and speedup of alternative BFS, with $k = 2$

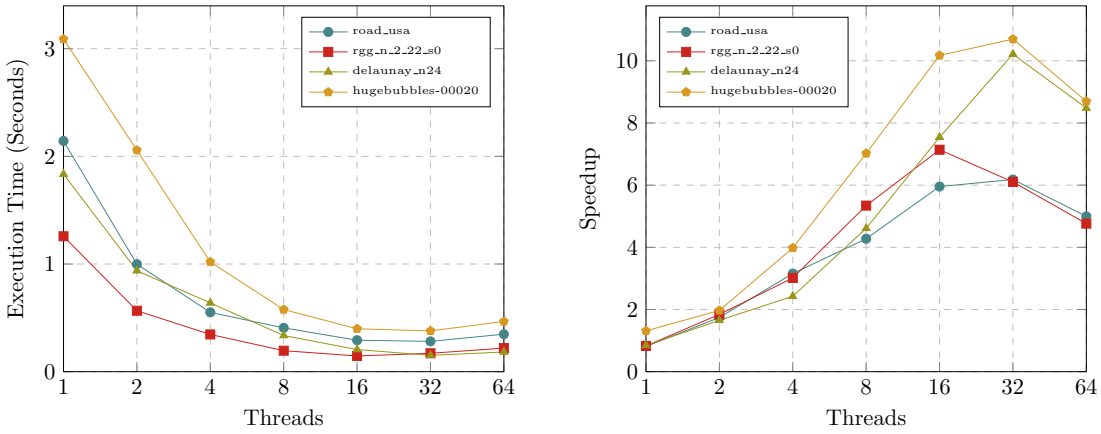

Figure 3: Plots of execution time and speedup of alternative BFS, with $k = 4$

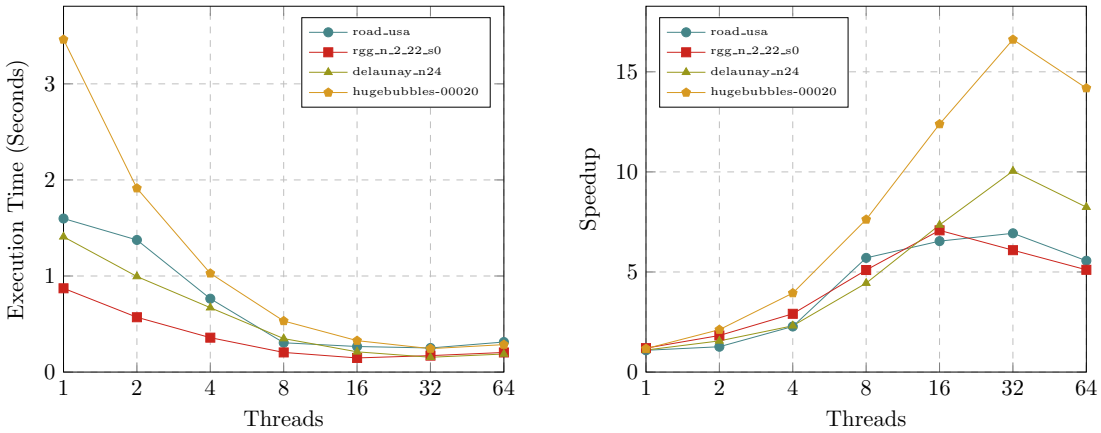

Figure 4: Plots of execution time and speedup of alternative BFS, with $k = 8$

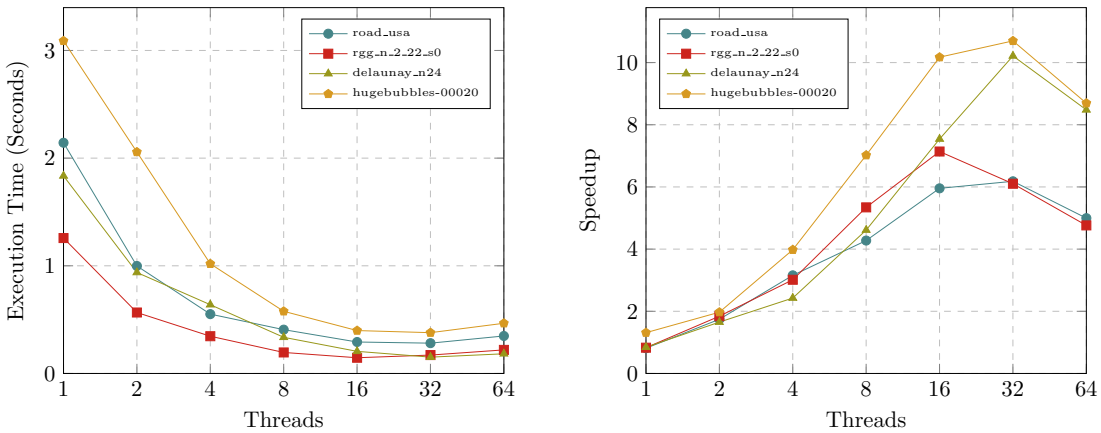

Figure 5: Plots of execution time and speedup of alternative BFS, with $k = 16$

4 Conclusion

As we can see from the tables and plots, it is possible to gain decently high speedup when parallelizing BFS. Depending on the graph, we can get a speedup > 10, with as little as 16 threads. This ended up being higher than expected going into this assignment.

For the alternative BFS, I do believe all results should be taken with a grain of salt, as there are some bugs in the algorithm, which introduces errors into the computation of the distances from the root nodes. However, we can see that for the cases where the results are correct, we get speedup on par with that of the PBFS implementation. Sometimes even exceeding the PBFS implementation, as evident by running it on the hugebubbles graph with $k = 8$, where we achieve a speedup of 16.63. Although, not all distances are correct, as it yields a 99.9% accuracy. As far as the optimal value of k, we can see that when $k = 8$, we get the best results. This is probably due to a good balance between the barriers present in the code when regathering the local queues and redistributing, and the speed obtained from exploring more than one layer at a time.

I wish that the final implementation of ABFS would be more correct, but after hours and hours of debugging, I could not locate and fix the bug which caused the inaccuracies.