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 approached 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.

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.

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, 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 = 1to numThreads do
        if i == tid then
            offset ← layerSize
        layerSize += T[i]

    barrierCopy localQueue to S starting at offset barrier
    Redistribute S evenly into local queues
```

3 Experiments

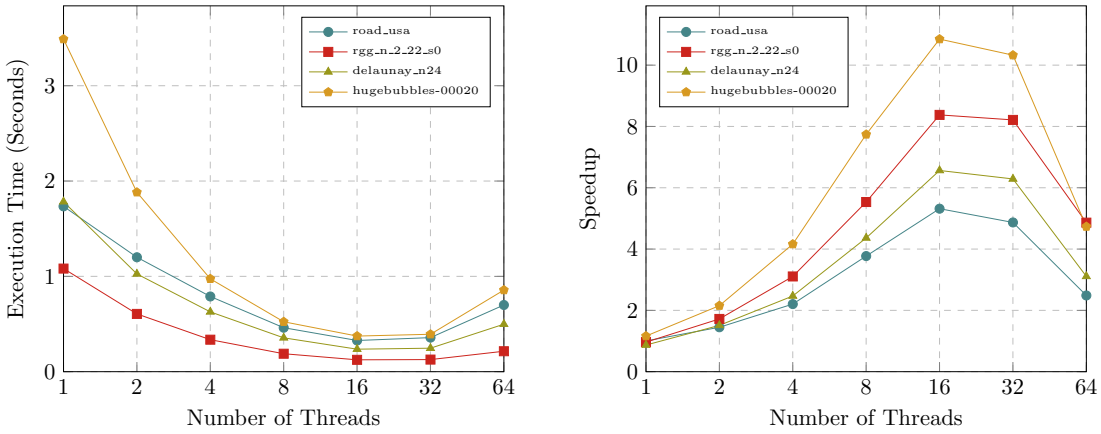

Figure 1: Execution time and speedup of parallel BFS