Term Project: Breadth-First Search Traversal

Kailee Madden

INTRODUCTION

Breadth-first search is a graph algorithm that is used to search a tree data structure. The algorithm starts at a given node and explores all nodes at the same current depth before moving on to searching the nodes at the next depth level. Essentially, BFS looks at the direct neighbors of a node and slowly expands the search from there.

This particular graph algorithm was a good candidate for parallelization because of this property of searching at the present depth before moving to the next depth. This means we can parallelize the search at each particular depth. In contrast, another common graph algorithm, depth-first search, is not a good candidate for parallelization because it explores as far as possible along each branch in the tree data structure before backtracking. This type of exploration is difficult to parallelize.

For the actual implementation of BFS, this project does not search for a particular node, but rather does a graph traversal, thus uses BFS to iterate through and visit every single node in the tree data structure.

There are two different types of BFS implementations that were used for this project. The classic sequential implementation and three attempted implementations in OpenMP. The OpenMP implementations are good examples of SIMD, with the data being a particular node (that was classified as a neighbor in the for loop) and checking if that node has been visited or not, then adding it to the queue as necessary. The BFS implementations will be described in more depth in the “Technical Approach” section. Beyond the BFS implementations, this project also required the development of a graph class in C++, so that the algorithms can take advantage of particular characteristics of the tree data structure that they are searching. The final two requirements that needed to be developed were graph test cases (five were developed, each of different size) and the batch scripts to run the parallelized BFS implementations while changing the number of threads, the NN variable, and the NM variable.

Initially, I also intended to develop an implementation in CUDA, however, the way in which I developed my graph class and test cases was quite inhibitive for using CUDA code. This is due to the reliance on maps based on strings. Memory on GPUs is extremely high latency, so passing a map from host to device would not only be complex and require a custom STL allocator, but also, it would not be worth the memory cost. Since my graph class was incompatible with developing GPU code, I decided to focus on the OpenMP implementations.

METHODOLOGY

Each of the BFS implementations that were developed for this project are discussed below. The discussions focus primarily on data structures and issues that arose during the parallelization process. The code files contain further comments explaining the purpose of each section. A general sequential pseudocode is first listed below, as it will be referenced in the implementation discussions.

Pseudocode:

```
Let G be a graph with nodes and s is our beginning node
Q is a queue (follows first-in first-out) and V is an array of booleans marking True/False if node has been visited

Q.enqueue(s) //insert beginning node into queue
V[s] = True //s has been visited

While Q is not empty:
        node = Q.dequeue() //remove the top element from the queue
        For all neighbors m of node in G:
                If V[m] is false: //have not visited the neighbor
                        Q.enqueue(m) //add to queue
                        V[m] = True //mark m as visited
```

Sequential Implementation:

The BFS algorithm leverages the queue data structure, which follows a "First-In, First-Out" protocol, to store each node that it needs to visit. The basic algorithm starts with a node and adds it to the queue, then while the queue is not empty, a search on all the node's neighbors is performed and they are added to the queue.

In order to store the actual graph nodes, an adjacency list was used. This is more efficient for iterating over nodes and adding new nodes, however, for large dense graphs this can cost much more memory. If the graph is sparse though, then adjacency lists can be better with regards to memory.

OpenMP Implementation:

Three OpenMP implementations were developed and tested. They are denoted “OMP1”, “OMP2”, and “OMP3”. Although initially the parallelization of the BFS algorithm appears to be fairly straightforward with only the inner for loop needing to be parallelized, there are actually a few more complications that arise.

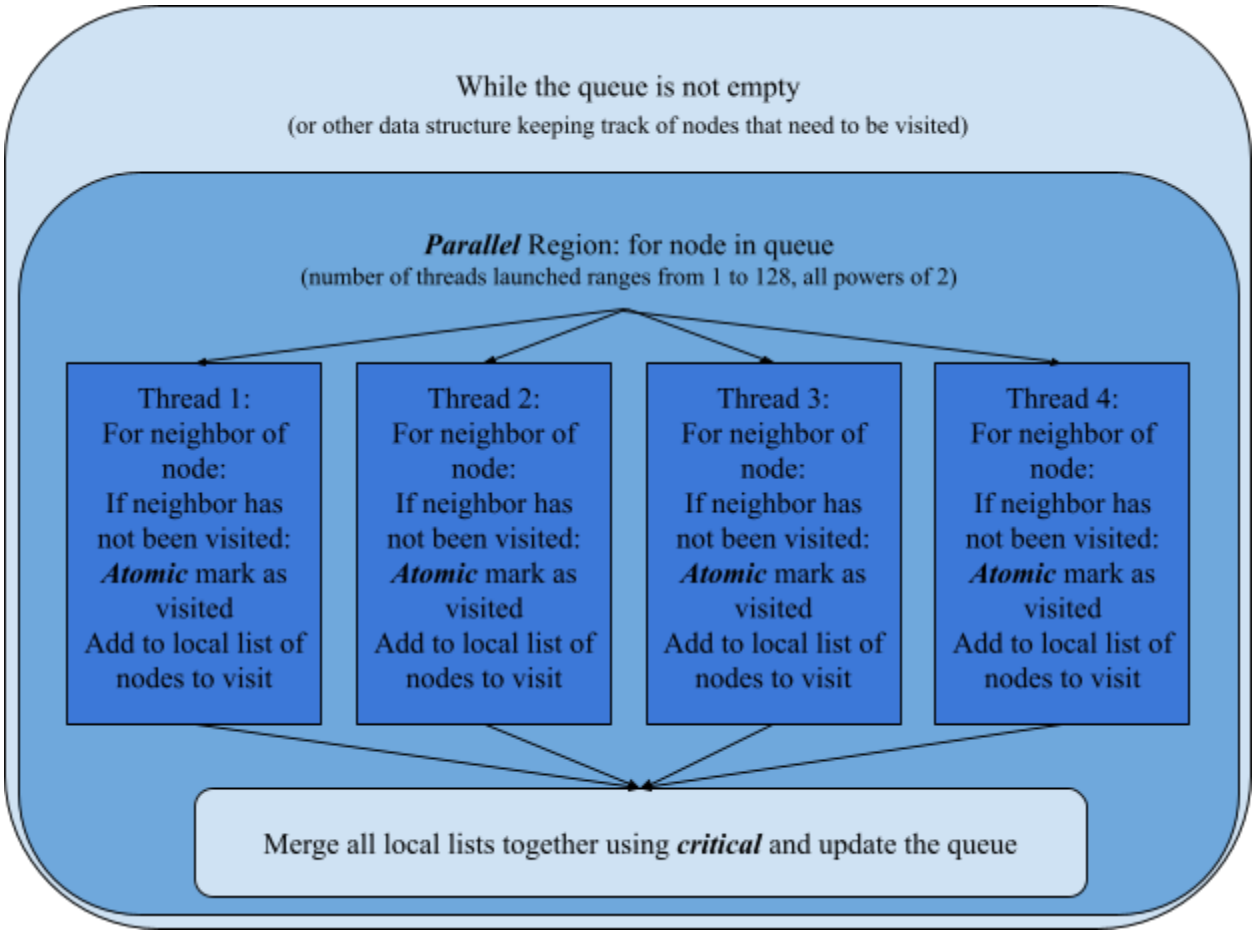
The first issue that arises is the actual implementation of the parallel for loop. When developing the code for this project, I started with the sequential implementation and thus developed a Graph class that utilized C++ lists. These are doubly linked lists and therefore have no random access memory. But OpenMP has limitations when it comes to for loops. One cannot use range based for loops: “for (auto value: mylist){}”. And within a for loop, only the following relational comparisons can be used: “<”, “<=”, “>”, “>=”. This means that iterators that use != cannot be used. There were two ways to address this issue that arose. I could rewrite my Graph class to utilize vectors instead of linked lists or I could do a different way of parallelizing my code. I implemented both options. In “OMP1” I used tasks so that I could still use an iterator in my for loop and in both “OMP2” and “OMP3” I leveraged the changes that I made to the Graph class in order to avoid this issue.

The second major issue that arises is multiple threads accessing a shared queue data structure simultaneously. In my initial naive approach, I locked my queue and my map of visited nodes in order to avoid this data race condition. However, this led to a major slowdown since the communication was slower than the computation. Thus, I did not use that method in any of my three final implementations, but rather, developed a new method where local lists of new nodes to visit were created on each thread and then those local lists were merged together at the end of the parallel region.

Further details on all three implementations can be found in the comments within the cpp files.

METHODOLOGY: *PARALLELIZATION FLOWCHART*

Below is a flowchart representing the parallelization that occurs in the OpenMP code. Each OpenMP implementation follows the basic process below, but varies in the way that it launches its threads and what data structures are used.



RESULTS

The first chart in this section provides the time it took for my sequential code to run for each of the test cases I developed. As the number of nodes in the graph grew, so did the amount of time to complete a BFS traversal, as expected. At 10,000 nodes, the sequential code became too large for memory allocation, so was seeded with a much lower number than the OpenMP code (.1 vs .9). The seed was for determining, randomly, which nodes would be connected. So by seeding with a smaller number, the sequential code has significantly fewer node connections and is a much sparser graph overall than the OpenMP implementations.

Sequential

Nodes Visited	Time
5	4.50E-05
52	0.000214
160	0.004239
1000	1.21718
10000	22.3565

The three charts below provide the details on how quickly each of the five test cases ran, with thread amounts ranging from 1 to 128, for each of the OpenMP implementations. It is important to note that the 1,000 and 10,000 node test cases involved some randomization, so the times are only comparable within a single implementation and not across implementations. The other three test cases were exactly the same across all implementations, and thus are comparable.

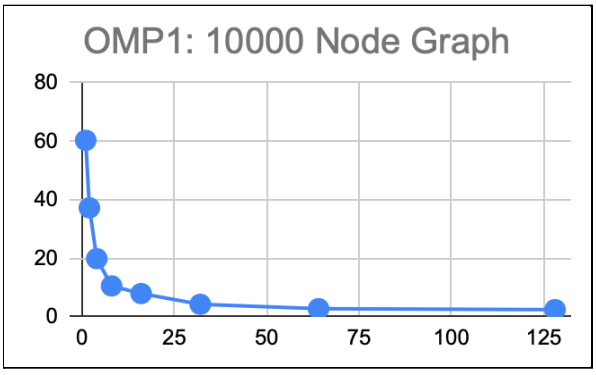
OMP 1									
Nodes Visited									
5	Threads	1	2	4	8	16	32	64	128
	Time	0.003738	0.003614	0.003831	0.006072	0.00734	0.036221	0.108698	0.156779
52	Threads	1	2	4	8	16	32	64	128
	Time	0.003618	0.003637	0.004145	0.004956	0.017078	0.048636	0.099428	0.172375
160	Threads	1	2	4	8	16	32	64	128
	Time	0.005475	0.00864	0.00466	0.004913	0.013236	0.032752	0.136282	0.171181
1000	Threads	1	2	4	8	16	32	64	128
	Time	0.516474	0.26506	0.134923	0.073735	0.0429	0.078354	0.120155	0.310282
10000	Threads	1	2	4	8	16	32	64	128
	Time	60.2686	37.219	19.849	10.5346	7.92648	4.2684	2.76343	2.42944

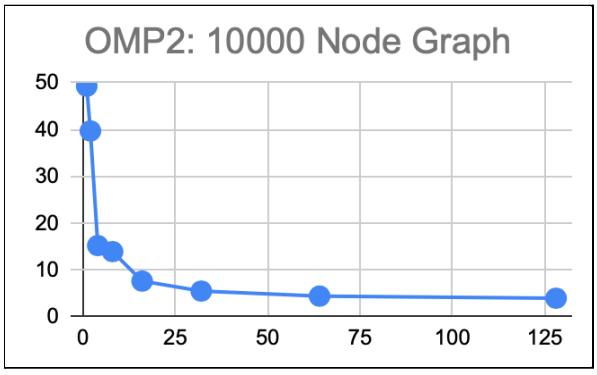
OMP 2									
Nodes Visited									
5	Threads	1	2	4	8	16	32	64	128
	Time	0.003283	0.003839	0.003984	0.004323	0.005611	0.036239	0.090762	0.173219
52	Threads	1	2	4	8	16	32	64	128
	Time	0.003478	0.003901	0.003957	0.005372	0.010561	0.019948	0.078243	0.12841
160	Threads	1	2	4	8	16	32	64	128

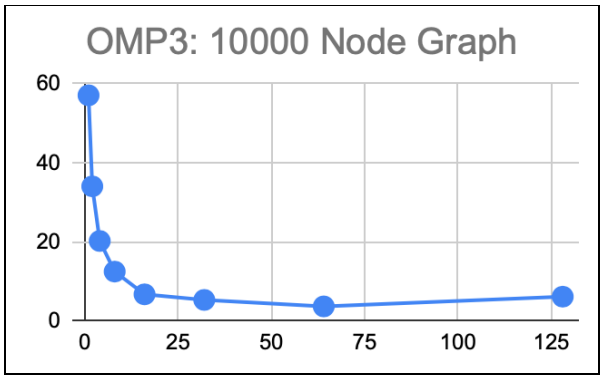
	Time	0.006187	0.005012	0.004857	0.005111	0.007056	0.045647	0.06825	0.115556
1000	Threads	1	2	4	8	16	32	64	128
	Time	0.413586	0.21532	0.112213	0.06014	0.064301	0.080448	0.154806	0.225264
10000	Threads	1	2	4	8	16	32	64	128
	Time	49.2205	39.6652	15.2025	13.9066	7.61769	5.48621	4.40971	3.96234

OMP 3									
Nodes Visited									
5	Threads	1	2	4	8	16	32	64	128
	Time	0.003795	0.003316	0.003761	0.004311	0.013971	0.033193	0.064383	0.112578
52	Threads	1	2	4	8	16	32	64	128
	Time	0.003937	0.003343	0.003738	0.004529	0.006002	0.043076	0.06648	0.104499
160	Threads	1	2	4	8	16	32	64	128
	Time	0.00503	0.004774	0.004231	0.004596	0.006052	0.017169	0.071906	0.079554
1000	Threads	1	2	4	8	16	32	64	128
	Time	0.417947	0.215875	0.112969	0.060239	0.038818	0.080915	0.11584	0.195274
10000	Threads	1	2	4	8	16	32	64	128
	Time	57.0443	34.0438	20.2392	12.5084	6.76248	5.34626	3.70688	6.16714

As shown in the charts, the expected parallel speedup was not fully seen until 10,000 nodes were used. The graphs for 10,000 nodes, and the varying thread amounts, are shown below. Only these graphs are shown below, because while some parallel speedup was demonstrated in other test cases, it was limited to less than 32 threads. Graphs for each of the test cases can be found in the PDF file “PerformanceAnalysis.pdf”. From the graphs below, it is evident that “OMP1” had the optimal parallel speedup.







According to the relative speedup equation, $Speedup(P) = \frac{T(1)}{T(P)}$, we can analyze the speedup for the 10,000 node test case shown above. The speedups are as follows:

- OMP1 has a speedup for 128 threads of $\frac{60.2686}{2.42944} = 24.8$
- OMP2 has a speedup for 128 threads of $\frac{49.2205}{3.96234} = 12.4$
- OMP3 has a speedup for 128 threads of $\frac{57.0443}{6.16714} = 9.2$

From these speedups, we can extrapolate that the "OMP1" implementation is the quickest of the OpenMP BFS implementations. However, when looking more closely at the charts of all times, it is clear that each of the implementations has relatively good speedup and there is no single implementation that is optimal for all test cases.

CONCLUSIONS

Based on the observations in the previous section, it is possible to extrapolate and draw a few conclusions about which implementation is best for various scenarios.

The first conclusion is that a sequential implementation is easier and just as quick, or potentially quicker if many threads are launched, when the number of nodes is relatively small. This is true for both sparse and dense graphs. Once the number of nodes reached 1,000 though, this was no longer the case and a clear parallel advantage can be seen (as shown in the graphs from the previous section).

The second conclusion is that the classic BFS algorithm is not as easily parallelized as it initially seems. There are actually two main BFS algorithms, a top-down approach and a bottom-up approach. The top-down approach was the algorithm that I implemented. However, a bottom-up approach can eliminate some of the extra checks of nodes that have already been visited.

Overall, there is a clear benefit to using a parallelized BFS algorithm, but this benefit is only shown once the graph has become sufficiently large.

References:

Scott Beamer, Aydin Buluc, Krste Asanovic, and David Patterson, *Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search*.