Parallel Graph Traversal

140050001
Mridul Sayana
Computer Science and Engineering

140050052
Abhijeet Kumar
Computer Science and Engineering

140050014
Shrey Kumar
Computer Science and Engineering

130050010
Suyash A Bhatkar
Computer Science and Engineering

***Abstract*—Essentially the title gives it away. We have taken the naive Breadth-First Search Algorithm and modified it to run parallel. This was done using OpenMP. We have also tried to do the same for another parallel traversal algorithm, namely, the Depth-First Algorithm. We have come to a conclusion that the implementation of BFS using multiprocessor runs much faster than the standard BFS.**

Keywords—Breadth-First Search, Depth-First Search, Parallel Programming, OpenMP.

I. Introduction

There are many graph traversal algorithms present in todays world. Some of them are:

1. Breadth First Search
2. Depth First Search
3. Dijkstra's shortest path algorithm
4. Bellman Ford
5. Kruskal's Minimum Spanning Tree algorithm
6. Prim's Algorithm, etc.

In our project, we looked into the Breadth-First Search and the Depth-First Search and used OpenMP to run them in parallel.

II. Algorithms

A. Breadth First Search

Suppose we have a given connected graph G = (V, E). V denotes the set of vertices and E denotes the set of edges in G. BFS explores the vertices and edges of a graph, beginning from a specified "starting vertex" that can be called the root vertex(level 0). It explores all the neighbouring nodes connected to the root. These neighbours nodes are placed in level(1). Similarly the algorithm then moves to these neighbouring nodes in level 1 and explores all their unexplored neighbouring nodes, and places these nodes in level 2.The algorithm continues till it explores last node of graph G.

BFS uses FIFO queue to decide which vertices to visit next. The queue starts out with only root (called 's') on it, with level[s] = 0. Then the general step is to take the front vertex v from the queue and visit all its neighbours. Any neighbour that hasn't yet been visited is added to the back of the queue and assigned a level one larger than LEVEL [v].

For example, in the following graph (Fig. 1), we start traversal from the root, that is vertex A. When we come to vertex A, we look for all adjacent vertices of it. B is also an adjacent vertex of 0. If we don't mark visited vertices, then B will be processed again and again and it will become a non-terminating process. A Breadth First Traversal of the following graph is **A, B, C, D, E, F, G, H, I, J.**

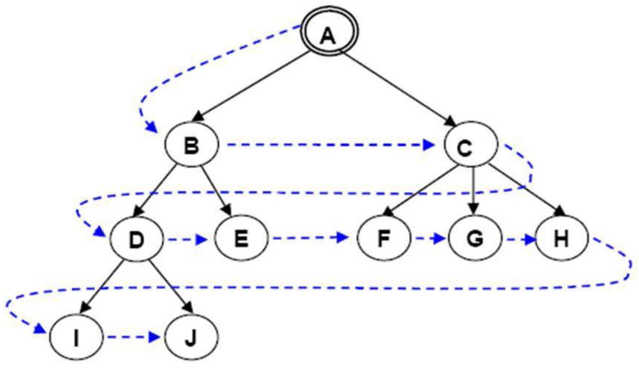

Figure 1: Breadth-First Search traversal

B. Depth First Search

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

For example, in the above graph, we start traversal from vertex A. We have already seen the BFS implementation of this graph. Now, suppose we start at vertex A. In case of DFS, when we come to vertex A, we look for all adjacent vertices of it. B and C is an adjacent vertex of A. We maintain a stack to ensure that we first visit the vertices with the maximum 'depth'. So, then A, B, D and I will be inside the stack with A at the bottom and I at the top. Then when we see that there is no neighbour of I left to visit, we backtrack to D by popping I from the stack and now D is on the top of the stack. Then we push J on the stack and then we visit the neighbour of J. Since, J has no neighbours, we pop J, backtrack to D and then to B and then visit E. And we continue this process until we have visited all the vertices. A Depth First Traversal of the following graph is **A, B, D, I, J, E, C, F, G, H.**

III. OpenMP

OpenMP is a standard parallel programming language used to develop parallel applications on shared memory machines. OpenMP is very suitable for designing parallel algorithms for regular applications where the amount of work is known apriori and therefore distribution of work among the threads can be done at compile time. In irregular applications, the load changes dynamically at runtime and distribution of work among the threads can only be done at runtime.

IV. Problem Statement

We want to study parallel graph traversal. Among the many known algorithms for this, we chose the Breadth First Search and Depth First Search. So, we have to connect the Sequential BFS and DFS into Parallel BFS and DFS to reduce Time Complexity in traversing the Graph.

IV. Aims Of This Project

Our aim in this project is to experiment with OpenMP and reduce the time complexity for the traversal of the Graph. We chose BFS and DFS to fulfil our mission. Our vision is to use OpenMP API in Sequential BFS to make it Parallel BFS. Similarly we want to do the same with DFS.

V. Sequential BFS

It is safe to say that when we are trying to study parallelism, we cannot rely on recursion to help us. That is because of the recursion stack that is maintained by the CPU. We cannot successfully traverse the graph parallel because we won't know how to divide the input between the threads. So we use the iterative method to do the Breadth First Search

Now, we are using an iterative BFS, so we can't just pass off the list of neighbours of the nodes of the current level to another function call and expect the CPU to take care of it. So, in place of this, we store the nodes of the current level in one array and the neighbours of the nodes in this array are stored in the second array in the order in which they appear in the first array. See Figure 2 for reference. Here A, B and C are nodes in the level 1 and Neighbours of A, if not visited, occupy the first part of Level 2. Neighbours of B if not visited occupy the next few positions followed by neighbours of C and so on. After all the nodes in this array are completed, this array is discarded and contents of level 2 copied into level 1. The unvisited neighbours of these nodes are stored in the level 2 array. This process is repeated till we finish visiting all the nodes.

Cycles are already taken care of because we only visit a node if it is not already visited. In case of disconnected graphs, we check if all the nodes given have been visited so far when there are no more neighbours available. Any nodes not visited so far have to be disconnected from the graph we explored, so we start with such a node and explore this component until every node has been visited.

VI. Parallel BFS

The main concept of parallel BFS is finding out which nodes will be in the next level, that is, the neighbours of nodes in the current level, in that order, and then distributing the work of visiting those nodes equally to every thread. So, if there are 'n' nodes to be explored and 't' threads, each thread visits 'n/t' nodes. The distribution of this work is always static and hence can be easily done by just using an OMP for loop which divides the iterations equally. The only part threads have to take care of is that when a thread tries to work on the next level without waiting for every thread to find all the neighbours of the current level, OMP won't know the correct amount of nodes in the next level, so that thread will get less work compared to the other threads which start working on that level later which isn't optimal.

This is why we use a critical section so that every thread has to wait for all threads to write down the neighbours it found before moving on to the next level. We implement the Parallel BFS in this way.

VII. Testing And Results

We generated graphs using the code that is submitted with the report. It was a roller coaster ride for selecting the perfect input for testing. After experimenting a lot, the best input that we found was a graph(included with report) with 10000 nodes, each with an out-degree of 1000. The timing data is represented in the form of a graph in Figure 3. The corresponding speed-up is represented in Figure 4

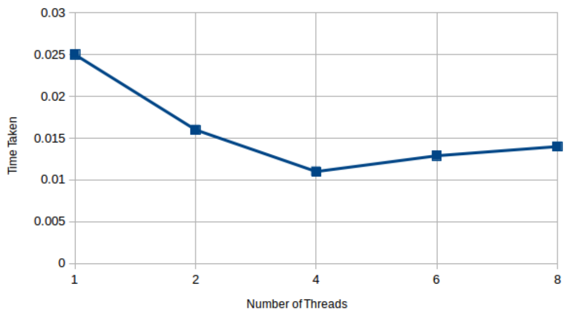

Fig 3. Timing Study WRT Number of Threads

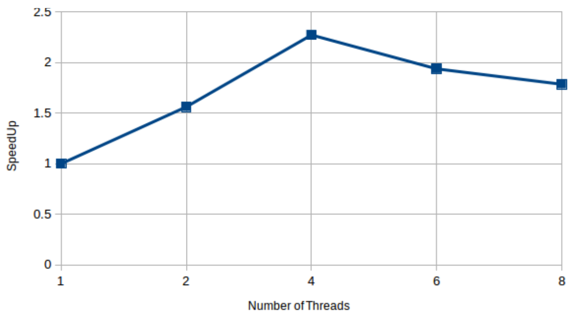

Fig 4. Speedup Corresponding to Fig 3.

While testing, we also noticed that the speedups don't depend on just number of nodes or the number of neighbours of each node, but a combination of both. So, even if we use 100,000 nodes, we get no speedup compared to the sequential BFS if we set maximum out-degree of 20. We think it might be due to the fact that we divide the breadth of the graph at a given level to different threads, so we might need a big enough level breadth to actually get a speedup compared to sequential BFS. But, if we just use a large number as the number of neighbours, it still won't speedup much if there aren't enough nodes in the graph, so we need to have a balance between the two.

A usual pattern we saw was the decrease in time taken from sequential to Number of Threads = 2 and from Number of Threads = 2 to 4 . But, when we used 6 threads or higher, it usually took more or around the same time than with 4 threads. OMP depends on the number of processors. It parallelises the threads, one on each processor. So even if we tell it to use 6 threads, it can only use 4 since number of processors is 4. So we get speedup during runtime until 4 threads and then runtime increases due to the fact that the only 4 threads at a time can run on the CPU and any extra threads will have to wait in queue. This was a positive experiment with our theories matching our results.

VIII. Sequential DFS

A stack is a very important part of a Depth First Traversal. A DFS cannot work without it. A recursive implementation of DFS is pretty trivial, so we implemented an iterative version with a stack, in the C++ language. Our DFS algorithm takes in a node and starts pushing all the neighbours of this node in the stack. Intuitively the program counter travels to the farthest depth possible from the starting point. Then, in the next iteration, the first element in the stack is popped and the process repeated. So, the first element is popped, intuitively thinking, the program counter backtracks and then again goes to the maximum depth possible. This process is repeated until all the nodes are visited.

As mentioned earlier, we can see that the stack is a very important foundation on which the Depth First Traversal works. But when it comes to Parallel Depth First Traversal, this same stack becomes our biggest foe. We will talk about this in the next section.

IX. Parallelisation Of DFS

We parallelise Depth First Search by distributing the work to be done among a number of processors. Each processor searches a disjoint part of the search space in a Depth First fashion. When a processor finishes searching its part of the search space it tries to get the unsearched part of the search space from other processors. When a solution path (i.e. a path from the initial node to the goal node) is found, all of them quit. If the search space is finite and has no goal nodes then eventually all the processors would run out of work and the (parallel) search will terminate. We assume that at the start of each iteration all the search space is assigned to one processor and other processors are given null spaces. From then on the search space is dynamically divided and distributed among various processors. Since each processor searches the space in a depth first manner, the (part of) depth first state-space tree to be searched is easily represented by a stack. The depth of the stack is the depth of the currently explored node, each level of stack keeps track of unvisited alternatives. Each processor maintains its own local stack on which it executes DFS. When its local stack is empty, the processor tries to get some of the stack from another processor.

But one problem arises here. Subgraphs can be of very different in size so it is difficult to estimate the size of a subtree rooted at a node. Dynamic load balancing is required. We need to distribute work between the threads properly. Because, the total time will be the maximum time taken by some thread.

We can try to do this dynamically, or statically. Thinking statically, each subgraph can have different sizes and there's no

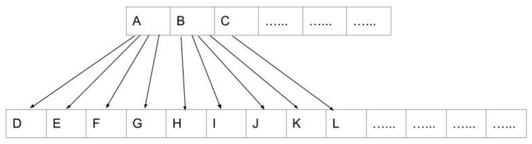

Fig 2. Working of our DFS

clear way of knowing it without exploring it all the way so assigning any subgraph to a thread is something that can't be trusted, because it being optimal can only be verified after they have finished their work, at which point it has already done all the work. Dynamic methods give us a hope. The idea is that once some thread finishes working, it can request some other heavily loaded thread for work, so that it wont stay idle. But there are again some problems with this method.

The sequential version of DFS maintains a stack. So theoretically speaking, every single thread will maintain a stack. Now, the main question lies in combining these multiple stacks in the right order. The second thing is that these many stacks exist, that means that they were divided in some way! So, who does this division? Its the main thread! How does the main thread divide the stack into these many separate stacks?

X. Conclusion

We have studied about graph traversal algorithms. One of them is BFS that is breadth first search algorithm and we have implemented a parallel approach to the sequential breadth first search. We experimented with some input graphs and then came to the result that on parallel processing with our proposed algorithm the graph is traversed much faster. We have also studied that by using different number Of threads, different timings of traversing the graph is noted. To achieve the above we have used OpenMP with C. We also studied about the parallel version of DFS and how to implement it. We saw and discussed a lot of ideas but due to time constraints, couldn't implement them as such. We came to a conclusion that, while we can parallelise DFS partly, parallelising DFS fully is a fairly daunting task and would require immense time and research.

XI. References

1. ftp://cs.indiana.edu/pub/techreports/TR250.pdf
2. https://www-users.cs.umn.edu/~kumar/papers/ijpp2.ps
3. Task Level Parallelization of Irregular Computations using OpenMP 3.0 by Eid Albalawi
4. https://mspace.lib.umanitoba.ca/bitstream/handle/1993/23321/Albalawi_Eid.pdf?sequence=1
5. https://www.programiz.com/dsa/graph-dfs
6. http://www-users.cs.umn.edu/~karypis/parbook/Lectures/AG/chap11_slides.pdf