Parallel Breadth First Search without Shared Memory

Josh Long, Neel Verma, Santosh Paudel, Tomas Carino-Bazan

Ramapo College of New Jersey

December 21, 2017

Abstract

Various researchers have proposed parallel versions of the Breadth First Search algorithm. However, many of these parallel algorithms make the assumption that the nodes in their distributed system have access to shared memory. We present two variations of the Breadth First Search algorithm which address the case in which shared memory is nonexistent in a distributed system. We also provide a list of suggestions that can be implemented to improve our current Parallel Breadth First Search algorithms.

I. Introduction

We begin by introducing the Breadth First Search (BFS) algorithm. In its original version, it can be used traverse a graph[1] with V vertices and E edges in $O(V + E)$ time. While this time complexity is sufficient for the common man, graphs with billions of vertices and edges render the original BFS algorithm useless. Researchers, such as Buluç and Madduri [1], have proposed parallel variations of the original BFS algorithm to combat these shortcomings. Intrigued by these issues, our team has also proposed two variations of the original BFS algorithm to handle large graphs. Before discussing our methods, we will first describe our distributed system.

II. Setting Up a Distributed System

Our distributed system is a five-node cluster consisting of five Raspberry Pi's. The cluster has a master-slave structure with one master node and four slave nodes. The master node and a slave node are able to communicate by using MPICH, a message passing interface. While MPICH does facilitate node-to-node communication, node-to-node communication is far from efficient. We suspect that this may be due to each Raspberry Pi's sub-par I/O bus.

One important property of our cluster is that its nodes do not share memory. This is critical drawback of our distributed system as we are unable to keep a global list of which vertices have been visited. Our solution to this issue is to keep a list of which vertices have been visited on our master node. The master node can then relay the most recent status of a graph's vertices to the slave nodes when necessary. However, this method requires that more data be sent via an inefficient method. Moreover, this could lead to concurrency issues which could slow algorithm more than necessary. Thus, we propose two parallel BFS (PBFS) algorithms with the hopes of reducing message-passing latency.

III. Two Parallel Breadth First Search Algorithms

Our approach to minimize message-passing latency relies on analyzing the density d of the given graph. We present two PBFS algorithms: one for sparse graphs and one for dense graphs.

[1]The reader can assume that all graphs mentioned in this paper are directed.

i. A Sparse Graph PBFS Algorithm

We will first discuss the master node's role in this algorithm. The master node picks a starting vertex s from our graph G. The master node then iterates over the adjacency list corresponding to vertex s. While iterating over vertex u, the master node waits for one of the slave nodes to become available. Once a slave node is freed up, the master node tells said slave node that vertex u must be processed.

We will now discuss the slave nodes' role. After the master node tells the slave node to process vertex u, the slave node runs the original BFS algorithm starting from vertex u. The list of nodes that the slave node managed to visit during its execution is then sent to the master node.

This method was designed to traverse through sparse graphs. However, sparse graphs with certain structures may cause the slave nodes to visit vertices multiple times (since no global list of visited vertices is maintained). It should be noted that this algorithm works well for graphs with few Depth First Search cross edges. In particular, this algorithm would work well on sufficiently sparse tree graphs.

ii. A Dense Graph PBFS Algorithm

We will once again begin by presenting the master node's task. The master node picks a starting vertex s from our graph G. The master node also sets up an empty queue Q and an array A of size V whose values are all 0; the value of A_i will be 0 if and only if vertex i has not been visited. At this point, A_s is set equal to 1 and s is appended to Q. The master node then sorts the vertices in the vertex s's adjacency list in ascending order according to their degree. The vertices in s's adjacency list are then appended to Q and their corresponding value in A is set equal to 1. The master node then iterates over the adjacency list corresponding to vertex s. While iterating over vertex u, the master node waits for one of the slave nodes to become available. Once a slave node is freed up, the master node sends said slave node a message containing with the vertex u along with the array A.

We will now discuss the slave nodes' role. After receiving the master node's message, the slave node iterates over the adjacency list corresponding to vertex u. During this iteration, the slave node cross-references the list A to construct a pruned adjacency list P of vertices that are adjacent to vertex u that have not been visited yet. The list P is then sent to the master node.

Upon receiving the list P from a slave node, the master node iterates over the elements of P. The master node then checks the status of each element p of P. If $A_p = 1$, p is ignored. However, p is appended to the end of the queue Q and A_p is set equal to 1 if $A_p = 0$.

This algorithm relies heavily on message-passing, and does not guarantee that vertices will not be vertices will not be visited multiple times. As a result, it is no surprise that the run-time of this algorithm is volatile. We have not performed an in-depth analysis on this algorithm, but we heavily suspect that it runs best on dense graphs due to the use of the pruned adjacency lists.

IV. Future Work

The PBFS algorithms that we have proposed are still in their infant stage. As such, we propose a few goals that should be considered in future iterations of this project.

i. Shared Memory Issue

Our current algorithms are designed to address the fact that the nodes in our cluster do not share memory. We have considered a few workarounds this problem. One of these solutions require creating a file that all cluster nodes can read from and write to. However, we have not been able to think of a safeguard against concurrency problems.

ii. A Different Distributed System

It should be noted that the shared memory issue is inherent to our Raspberry Pi cluster. Thus, one solution to eliminate this problem is to get a distributed system that allows its nodes to share memory. In the process, the latency caused by message-passing may also improve.

iii. Load Balancing

The algorithm for traversing dense graphs uses a variation of the Shortest-Job-First algorithm to determine the order in which the source's neighbors will be processed first. This is done to increase the probability that the pruned adjacency lists generated by the slave nodes are as small as possible. In this future, different load balancing methods can be considered with the hopes of reducing the size of the pruned adjacency lists.

iv. Cycle Compression

We have mentioned that our sparse PBFS algorithm work well on sufficiently sparse tree graphs. One way to approximate a given graph G to a tree graph H is to compress a graph's cycle C to a vertex v. In particular, (v, u) will be an edge of H if and only if some vertex in the cycle C is adjacent to u. Similarly, (u, v) will be an edge of H if and only if u is adjacent to some vertex in the cycle C. We can then run the sparse PBFS algorithm on the graph H. By using this method, we may find that a graph can be traversed better by using our sparse PBFS algorithm instead of the dense PBFS algorithm.

V. Conclusion

The lack of shared memory between the nodes in our cluster is the main influence behind the design of our PBFS algorithms. While we did not anticipate this issue, we have come to terms with the inherent restriction that our Raspberry Pi cluster has placed on us. This limitation has introduced us to a relatively unexplored problem that we had not previously considered. Like with non-parallel graph traversal algorithms, we expect that there are a variety of non-shared-memory PBFS algorithms that take advantage of the structure of a given graph.

VI. References

1. Buluç, Aydin, and Kamesh Madduri. "Parallel Breadth-First Search on Distributed Memory Systems." *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*, 2011, doi:10.1145/2063384.2063471.