Graph Analytics on Shared Memory Multiprocessors

James Bickerstaff jjb169@pitt.edu

Fan Shen fas48@pitt.edu

Abstract

Graphs are versatile data structures that may be used to represent many different datasets, from computer networks, protein k-mers, circuitry, and much more. Depending on the size of a dataset, graphs may grow to be composed of millions of vertices and edges, creating a large memory footprint that will cost a single processor a large amount of time to analyze. Due to this large computational factor, this research paper pursues two important graph operations in an attempt to parallelize them and improve speedup on machines with shared memory and multiple processors available. These operations are minimum spanning trees and breadth-first search traversals, both of which have multiple applications depending on the data represented. OpenMP is utilized within the C++ language to create the programs for testing on a powerful research computing cluster, achieving peak speedups of approximately 2.25 and 3 for minimum spanning trees and breadth-first searches, respectively.

Keywords

OpenMP, graph, minimum spanning tree (MST), breadth-first search (BFS), Speedup, Parallel Efficiency

1 Introduction

A graph *G* may be defined as a collection of vertices *V* that are connected together by edges *E*, and may be classified as *undirected* or *directed* in nature. Undirected graphs contain unordered pairs of edges (source, destination), in which either vertex included may be seen as either the source or destination vertex. Directed graphs, however, make use of ordered pairs of edges (source, destination), in which the vertices are not interchangeable and represent a direct path from the source to destination [1]. These edges may also be classified as *weighted* or *unweighted*, represented as a tuple of (source, destination, weight). The weight of an edge is used to define the cost of traversing between the source and destination vertices, and may be a representation of distance, electrical resistance, or time, for instance [1]. The graphs analyzed within this paper are of undirected and weighted nature.

The importance of these graph structures is that they may represent numerous different kinds of data, including social networks, circuits, maps, and computer networks. These graphs may then be analyzed for the purpose of showing relationships between people and groups (e.g. Facebook), mapping of VLSI circuits [2], or creating a shortest path between a start and destination on a map (e.g. Google Maps). As these graphs grow larger and more complex in size and connections between vertices, the computational time to analyze them grows as well, as there are many branching paths that must be taken into account. This growing popularity of graphs, as well as their computational demands, are the primary motivations behind conducting

this research project, as we hope to exploit the available resources on parallel machines to accelerate common operations performed on the graph data structures.

2 Related Work

There have been various different research papers written in regards to accelerating and parallelizing pre-existing graph operations for both distributed and shared memory systems, as well as creating new algorithms to complete these functions with lower time complexities.

Two of such papers focus on Borůvka's algorithm for minimum spanning trees and were created by Chung and Condon [3], and Suryanarayana Durbhakula [4]. Chung and Condon sought to parallelize Borůvka's algorithm while targeting a distributed memory model that makes use of message passing. Their key contributions came from developing a new pointer jumping algorithm that is used to determine the root of an inputted tree among distributed data, and comparing the results of this implementation alongside other methods [3]. Durbhakula created two different parallel implementations based on Borůvka's algorithm for a shared memory multiprocessor and presented the results from both when compared to a serial baseline, showcasing runtime on various sized graphs. The difference between the two parallel implementations was in regards to how the cores coordinated working together and deciding on the cheapest edge for a component, making use of lock variables and atomic functions [4].

One more paper of interest with a focus on minimum spanning trees sought to parallelize the two other classical algorithms for the operation instead of Borůvka's: Prim's and Kruskals. Their implementations targeted distributed systems, utilizing MPI and analyzing graphs that do not fit on a single processor. The main parallelization behind Prim's algorithm was to determine cheapest edge candidates on each processor, then combine the results using all-to-one reduction and one-to-all broadcast [5]. For Kruskal's algorithm, each processor sorts the edges within their partition of the overall graph, creates a local minimum spanning tree, and then processors communicate to gradually connect their trees together into a final result [5].

There are several papers related to parallel breadth-first search using shared memory systems as well. As a common approach, breadth-first search typically uses a queue to store neighbors, which is not amenable to parallelism. One paper proposed by Leiserson and Schardl announced that the operation of the FIFO queue must be serial and this hinders the performance of parallelism, and they use a new data structure in place of the queue, called a 'bag'. A 'bag' is an unordered set used in their designed algorithm: Parallel BFS (PBFS), and it has a fixed size called backbone [6].

The reason why a 'bag' is used is because this kind of structure supports the parallel traversal, and it can be divided and merged. In PBFS, it defines a function to traverse each layer of the graph using the bag structure. After initialization, the PBFS can execute iteratively to deal with each layer. The result in the paper shows good parallelism, and with the increase of cores, the speedup improves as well.

Another idea of a parallel breadth-first search algorithm is proposed by Shivapuram et al(2017). In the paper, the design is to partition the graph between multiple compute nodes with necessary replication of boundary nodes across partitions and use both MPI and OpenMP to implement the algorithms [7]. MPI is used for communication between nodes and openMP is used to achieve parallelism in each node. When breadth-first search traverses the vertices at one level, the vertices of the next level are added to the queue frontier. The vertices of this

container that belong to different nodes will use MPI to send information to corresponding nodes. The processing of local-owned vertices can be optimized using openMP. When one processor receives vertices from a remote node, it will add them to the queue and continue to traverse. After each layer, an MPI Barrier is used to synchronize the processors. Throughout execution, a node will continue to process the locally owned vertices while waiting for other vertices to be received remotely.

3 Operations Implemented

For this paper we focused on parallelizing a minimum spanning tree algorithm, as well as a breadth-first search algorithm. All code was written in C++ and made use of OpenMP for multi-threading capabilities. Additionally, the graphs utilized are read in at the start of each program, and are stored in an adjacency list format. The following subsections will define these operations and how they are performed, as well as the approaches used for parallelization.

3.1 Minimum Spanning Tree

A spanning tree of an undirected graph *G* may be defined as a subgraph of *G* that is a tree connecting all vertices within. This may be extended to weighted graphs in which the weight of the spanning tree is the sum of the weights of all edges. Combining these ideas, a minimum spanning tree (MST) for a weighted undirected graph *G* is a spanning tree with the smallest total weight achievable [1]. In the event that a graph is not fully connected (all vertices reachable from any vertex), a minimum spanning *forest* is produced instead of a minimum spanning *tree*. It should be noted that a graph may have multiple minimum spanning trees if the weight of two different spanning trees are both the smallest weight possible, but with different edges included. The number of edges within a minimum spanning tree will always be equal to the number of connected vertices minus one (*V -1*).

There are three different algorithms primarily used when solving for a minimum spanning tree. The first of which, published in 1926 by Otakar Borůvka, is known as Borůvka's algorithm [8]. This algorithm begins by analyzing one vertex at a time within a graph, keeping track of the lowest weighted edge. Once all vertices have been checked, these edges are added to the final resulting tree (each individual group of vertices connected together during the process is known as a "component" of the tree). If the minimum spanning tree is created after this first iteration, then the operation is finished. If it is not, the process begins again, analyzing each edge connected to a vertex, excluding those already added. The primary difference in all iterations after the first is that instead of taking the lowest weighted edge from a single vertex, the system must check the entire component that is a part of instead, taking the lowest weighted edge from the group of vertices and adding that to the overall result. An additional check that must be made is to ensure that the lowest weight edge observed does not connect two parts of the same component together. If the lowest weighted edge does so, it is thrown out and the next best option is selected. This process continues until there is a single component containing all possible vertices connected by V-1 edges, producing a spanning tree of minimum cost. Because this is the algorithm utilized within the research presented, an example will be displayed in Figure 1 on the following page to ensure understanding.

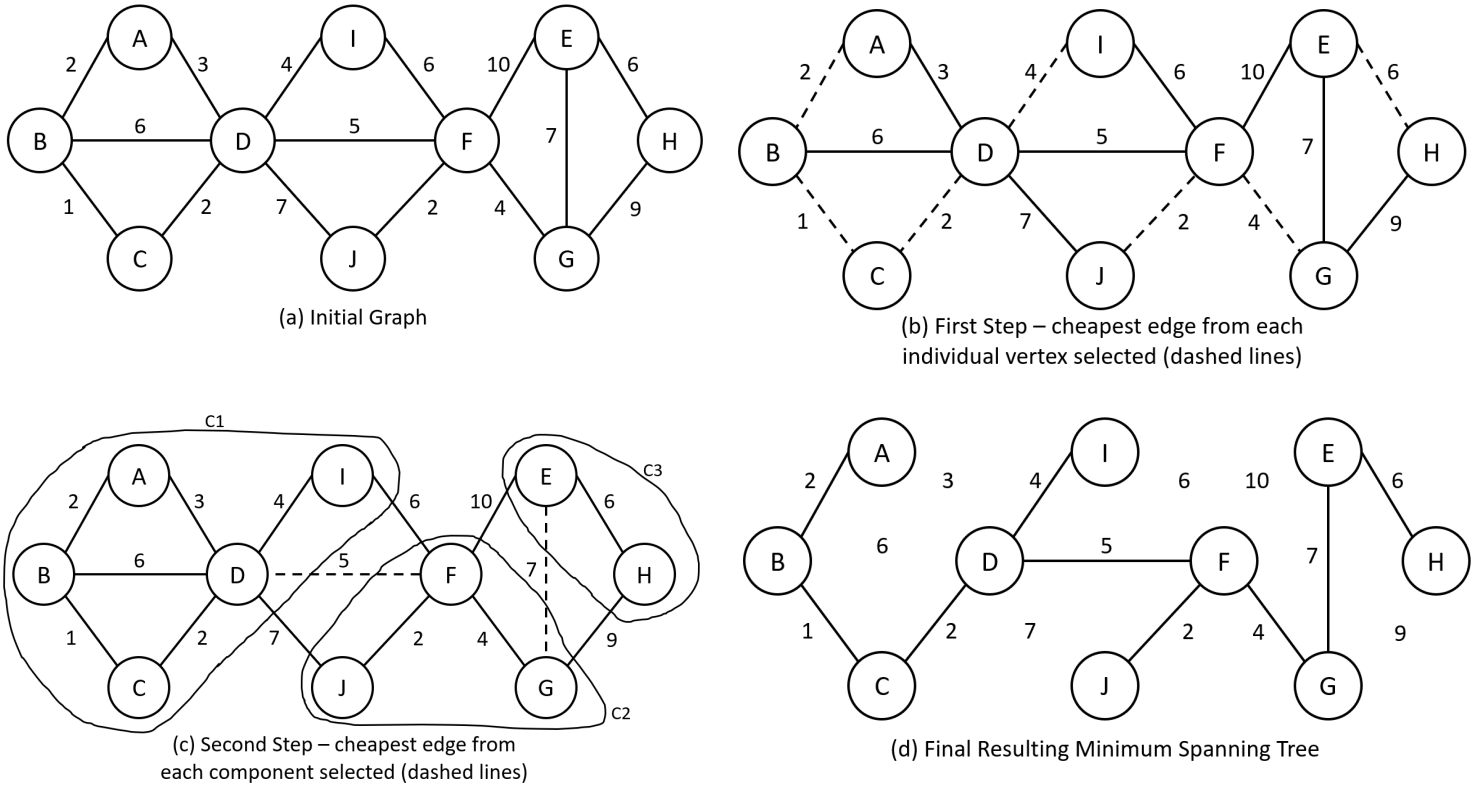


Figure 1. Example of Borůvka's Algorithm

Image (a) in Figure 1 above displays the initial graph structure that will be analyzed for the minimum spanning tree. Image (b) depicts the lowest weighted (cheapest) edges observed from each vertex in the graph, shown as dashed lines. These edges are then added to the final result, creating three separate components amongst the remaining vertices. These three components are seen in image (c), labeled as C1, C2, and C3. The cheapest edge from each component is then selected and once again represented as dashed lines. These edges are added to the final result, combining all three components into the minimum spanning tree as seen in image (d) above. The overall runtime of this algorithm is O(E * log V), where E is the number of edges, and V is the number of vertices in graph G.

The next algorithm that is popular for finding minimum spanning trees is Prim's algorithm. Prim's algorithm begins by arbitrarily picking a starting vertex (the root), observing all edges attached, and picking the one with the lowest weight possible to add to the final result. These two vertices may then be considered as a single component moving forward, similarly to Borůvka's algorithm. On the second step, all edges connected to the root and the newly added vertex are observed, and the lowest weighted edge (not connecting a component to itself) is then determined and added to the resulting tree. This process continues, adding a single vertex at a time to the growing component, until all vertices are included within the final result [1]. This algorithm is known to run in O(V^2) time, or O(E * log V) time with optimizations.

The third minimum spanning tree algorithm that will be reviewed is Kruskal's algorithm. This algorithm begins by sorting all edges within the entire graph by weight- from least to greatest. Following this, the lowest weighted edge will be taken from the list, checked if its addition will create a cycle (a cycle exists if a vertex may traverse through its neighbors and end

up reaching itself- i.e. if an edge is added between two vertices within the same component), and if not, it will then be added to the result. This process of taking the shortest edge, checking it, and adding it to the result is then repeated until there are (V-1) edges in the tree, which means that the minimum spanning tree has been produced [9]. The runtime for this algorithm may be defined as O(E * log V), identical to that of Prim's with optimizations.

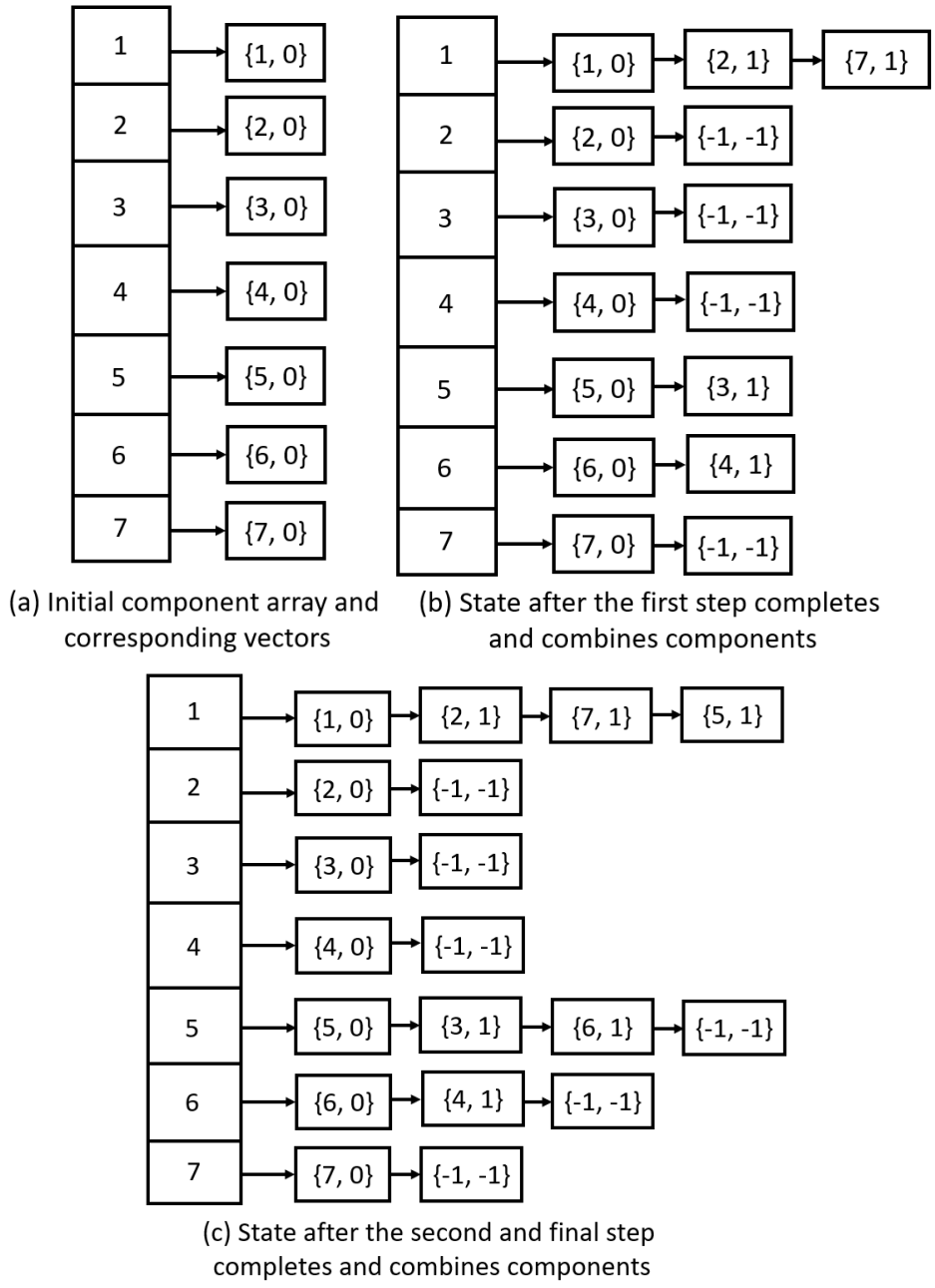
3.1.1 Parallel Approach

As mentioned in the previous section, Borůvka's algorithm was picked to parallelize the minimum spanning tree optimization. This is due to several factors regarding both Borůvka's and the other two algorithms described above. First, Borůvka's algorithm is inherently parallel in that there is no singular starting root, and each vertex (or component) may pick out their cheapest edge available at the same time, then once all have finished they may be added to the result. Prim's algorithm, however, picks a singular starting root and grows outwards from that one vertex, which makes it difficult to parallelize. It may still be parallelized to an extent, however, through each processor finding the minimum for its assigned range of vertices and performing all-to-one reduction to create a final answer for that current iteration [1]. Kruskal's algorithm is similar in that the operation of picking a cheapest edge is generally sequential and difficult to parallelize; there have been attempts to do so, however. One method is to parallelize the sorting required at the start of the operation, then add them sequentially. A second approach, as done in [5], is to partition the graph and find local minimum spanning trees on each partition, combining them once all processors complete their task. Thus, due to its parallel nature and potential for speedup, Borůvka's algorithm was chosen.

Three main approaches were implemented and tested to achieve a speedup over the serial baseline for Borůvka's algorithm. The first approach makes use of an additional data structure containing the different components within a graph, represented as an array of vectors storing integer numbers. An array index is created for each vertex within the graph, and the vector that corresponds with it holds the values of vertices that are contained within the same component, rooted at that index (ex: components[5] = {5, 1, 2, 3} represents a component rooted at vertex 5, that includes vertices 5, 1, 2, and 3). Any value that is not a root holds the value of "-1" and is ignored by a processor, which then moves on to the next index for evaluation. Once the lowest weighted edges are picked for each component, the edges are added to the resulting minimum spanning tree, and the component vectors are combined based on which root has a higher "rank", which is just a value representing component size. This process is then repeated until there is a single component remaining that holds all connected vertices. The work is scheduled and distributed by using the OpenMP "guided" method, giving each core responsibility over several component groups to start, and gradually decreasing the number handed out until all iterations are completed. This ensured that no two cores would be working on the same component or creating data races and memory access errors with one another. The overall performance of this design, however, was slower than the serial baseline by several seconds, even when operating on 16 cores. This is more than likely due to the uneven distribution of work as the algorithm progresses, in addition to the time it takes for the component vectors to be combined, as that process must be done sequentially. It must be noted that this additional 'components' data structure does not represent the final resulting graph,

rather it is used in the creation of the minimum spanning tree and is a means to organize and distribute the work between cores.

The second approach taken to parallelize Borůvka's algorithm is an extension/modification to the first method, attempting to optimize away the need for all of the component vector combinations. This implementation uses the same underlying component data structure- represented as an array of vectors, however, the vectors now store pairs of values instead of single integers. Each pair contains two integers: one representing the vertex that's been added to the component (similar to the original implementation), and another that states if the given vertex was previously a root and had vertices attached to it. The second integer value may be a "0", "-1", or "1". If the value observed is a "0", that vertex's edges are observed to see if they have the lowest weight for the component, which follows the same behavior as the previous implementation. If the value is a "-1", the core observing it will either ignore traversing the vector, or understand it as the end of the traversal and end the task. If the value is a "1", the processor checks the corresponding vertex's array index and traverses the vector to find out what vertices are attached to the component. This process trickles through each connection and continues traversing until an end condition is observed ({-1, -1}), at which time the system will begin to return up the function call stack and produce the lowest weight edge observed. Once a component is added to another, a pair of {-1, -1} is added to the end of the smaller component to signify that it has been absorbed into a larger piece. This allows a core to ignore traversing the given index and move onto the next candidate, as it has become the responsibility of the larger component's owner. This entire process is handled in a recursive manner, which adds complexity, but ensures that when two components are combined, only two additions are made to the component vectors, as opposed to the numerous needed by the previous implementation. A brief example is shown in Figure 2 on the next page to illustrate the process.



(a) Initial component array and corresponding vectors

(b) State after the first step completes and combines components

(c) State after the second and final step completes and combines components

Figure 2. Example of second Borůvka Implementation

Image (a) in FIgure 2 above displays the components array for a graph of 7 nodes, each pointing to the values contained within their respective vectors. Initially, each vertex contains only that index's value alongside a 0, telling the core responsible to observe the edges for that given vertex and check for a lowest weight value. Following the first step in the algorithm, the components 2 and 7 are combined with 1, resulting in {-1, -1} pairs being added to the vectors of 2 and 7. Component 3 is added to 5, and 4 is added to 6, once again adding {-1, -1} pairs to their vectors. The resulting vector array may be observed in image (b) above. When a core begins to check for the cheapest edges of a component in the second step, beginning at vertex 1, for example, the process follows as such: a core enters the vector of vertex 1, sees the pair {1, 0}, and reads all edges for vertex 1. Next, it observes pair {2, 1}, noting the "1" as the second

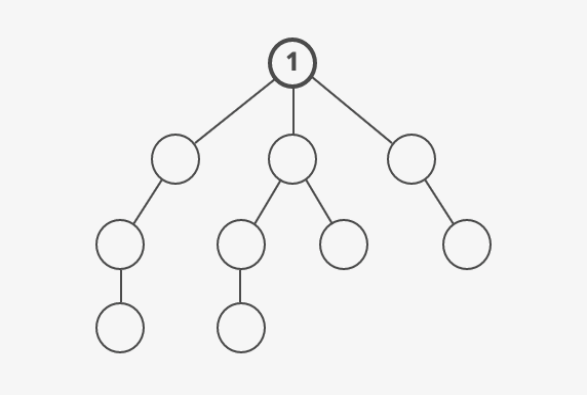
value, and visits index 2 to check its vector. The core finds the pair {2, 0}, and read the edges for this vertex. {-1, -1} is seen next within the next vector location, informing the core to return to the previous index, 1 in this case, and the pair {7, 1} is now observed. In a similar fashion, index 7 is visited and {7, 0} is observed, reading the edges for this vertex and comparing them for the lowest weight. The pair {-1, -1} is found next, returning the process once more to vertex 1. Following this, there are no more pairs to check, and thus, the edge with the lowest weight for the component rooted at vertex 1 has been found. One more step is taken to complete the minimum spanning tree process, adding component 6 to 5, and 5 to 1, resulting in the component array found in image (c) above. As noted above, this resulting data structure is not the minimum spanning tree, but is a means to organize and distribute the work needed to complete the Borůvka's algorithm process. Unfortunately, this implementation also suffered from slower runtimes when compared to the serial baseline, and due to the complexity, the reason for it was unable to be found before another solution had to be pursued due to time constraints.

The third implementation created is the simplest among the three ideas pursued, and produced the fastest parallel runtimes which are those displayed within the experimental results section below. This design functions primarily by splitting up all vertices within the graph amongst the cores available (e.g. core 0 “owns” vertices 1-5000, core 1 “owns” 5001-10000, etc.), allowing each to check for the lowest weighted edges at the same time, combining the results as the program runs. The OpenMP scheduling method utilized for this implementation is “dynamic, 5000”, as many different configurations had been thoroughly tested, and the fastest runtimes were produced using this setting. This allows for cores that are running faster to pick up more work and improve runtime, whereas in a static scheduling situation this would not be possible. It also aids with load balancing, as in the case where a vertex may have a huge number of edges attached and others have significantly less, the cores handling sparse vertices may simply take more work to make up for it. An additional change made is that in the baseline algorithm, a single edge observed may be added as cheapest for two different components at the same time if the criteria was met, improving the overall runtime. For this implementation, however, that option had to be sacrificed, as it was causing many data races and memory access errors during runtime. The parallel execution of this algorithm is able to make up for this sacrifice, however, providing speedup over the baseline. Once the lowest weighted edges have been selected for each component, the program returns back to serial execution in order to add the edges to the final resulting graph and ensure the correct result is produced.

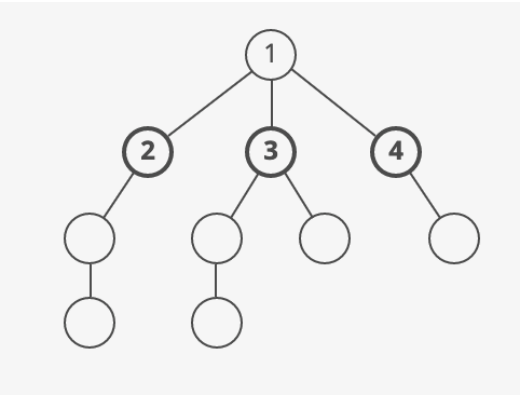
3.2 Breadth-First Search

Breadth-first search is an approach to traverse the graph from a selected node and explore the neighbor nodes by traversing the graph layerwise. If we extend the breadth-first serach algorithm in a weighted graph, the algorithm may not work, since the path with the fewest edges may not be the shortest path if the edges it contains are expensive. Basically, we start breadth-first search at the root, usually called node 0, and the algorithm will explore the nodes in the current depth before moving to the next depth level. As a commonly used algorithm, breadth-first search has a very long history. Breadth-first search was firstly proposed by Konrad Zuse in 1945 to find connected components of graphs in his Ph.D thesis on Plankalkül programming language. However, the idea was not published until the 1970s [10]. In

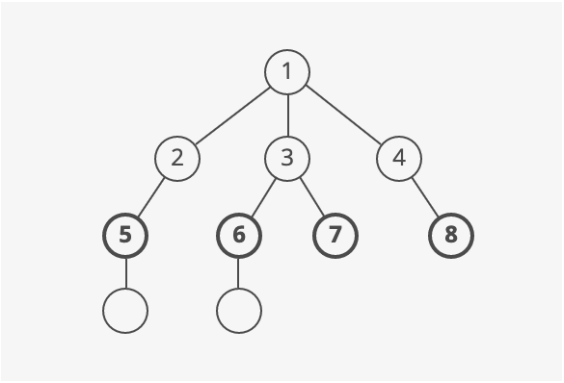
this time period, Edward F.Moore reinvented it in 1959 to determine the shortest path out of a maze. C.Y.Lee developed the algorithm in a wire routing scenario.
Here is a simple example of using breadth-first search to traverse a tree.



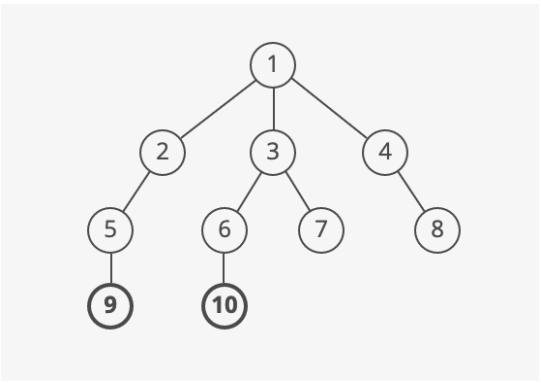
Step 1: Starting with the root



Step 2: Visit the children of the root



Step 3: Visit the next depth level



Step 4: Continue until the end is reached

Figure 3. Example of Breadth-First Traversal

In the algorithm, a FIFO queue is typically used in order to achieve the result. Suppose we need to traverse the whole graph, It usually has four steps:
Step1: Starting with the root- in the previous example this is node 1, we enqueue node 1 to the queue;
Step2: Node 1 is then marked as visited;
Step3: If the queue is not empty, this step is taken. The next node is removed from the front of the queue, and checked whether its neighbors have been visited or not. In the previous example, the front node is 1. The neighbors of node 1 are added to the queue and marked as visited.
Step4: Loop step 3 until the queue is empty, and once this is true, all nodes in the graph have been visited.
Below is an illustration of this process:
Begin the breadth-first search algorithm on the previous example, the queue status is:

1									

Dequeue node 1, and enqueue its neighbors node 2, node 3, node 4.

2	3	4							

Dequeue node 2, and enqueue its neighbors node 5.

3	4	5							

Dequeue node 3, and enqueue its neighbors node 2, node 3, node 4.

4	5	6	7						

Dequeue node 4, and enqueue its neighbors node 7, node 8.

5	6	7	8						

The second level now has been visited, moving to the next level. Dequeue node 5, and enqueue its neighbor node 9.

6	7	8	9						

Dequeue node 6, and enqueue its neighbor node 10.

7	8	9	10						

Dequeue node 7, however, there are no new nodes enqueued, this process continues until the queue is empty.

8	9	10							

9	10								

10									

For breadth-first search, the algorithm ‘s runtime is proportional to $O(|V|+|E|)$, where $n = |V|$ is the number of vertices and $|E|$ is the number of edges of a graph $G=(V,E)$. The algorithm’s best-case and worst-cast performances are equal [10]. In the worst case, BFS has to explore all of the edges in the connected component in order to reach every vertex in the connected component [11]. The space complexity is $O(|V|)$, this is in addition to the required space for the graph itself. The complexity can be varied depending on the representation of the graph by implementing the algorithm.

In real-life, breadth-first search is useful in many scenarios. For example, a primary usage of the BFS algorithm is for finding the shortest path between the entry point and the exit point (without weighting) [6]. In communication, P2P network(Peer to peer network), Breadth-first search can be used to find all neighbors nodes.In social networks, we can use the algorithm to find people within a selected distance from another person. The initial breadth-first search is used to find out a path for a maze game, breadth-first search can begin at the gate and traverse all possible paths and figure out which route we can reach the end.

3.2.1 Parallel Approach

As the inspiration of Sivapuram's paper. I can divide the graph by the number of cores and assign each beginning vertex with a different index to each core. Suppose we have N vertices in the graph, and C cores are used to parallelize the operation. We also can use the FIFO queue to store vertices and a visited array to check and change the status of each vertex. Initially, we need to initialize an array to store the status of nodes in a graph. We divide our graph into different parts by index, and each core is assigned a different beginning node. In this case, the graph was divided evenly, so the beginning nodes for each core is ((N/C)*core_id), and then the parallel regions are introduced. At first, we will push the root into the queue and pop the front element from the head of the queue, and mark this node as visited. Then we check the status of the neighbors of this node; if they are not visited, we need to push those nodes into the queue and mark the nodes as visited. After all of the cores private queues become empty, that means all nodes have been visited.
Here is simple 2 cores(core0 and core1)diagram:

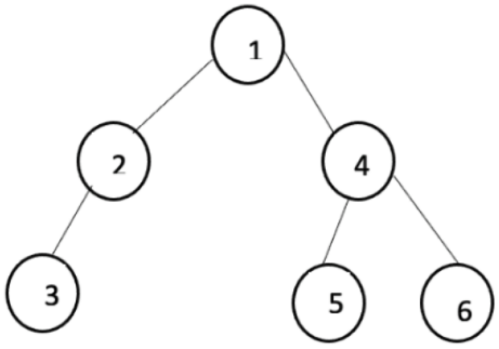


Figure 4. Graph example for Parallel BFS

By assigning nodes to 2 cores, core0 begins at the root (node1), and core1 begins to traverse from node 4. As before, we enqueue the neighbors, so first we will check the visited statuses for each core's neighbors. Core 0 will enqueue node 2 instead of node 2 and node 4. Ultimately, core 0 will go through 1->2->3, and core 1 will go through 4->5->6 in a similar way.
Below is an example of this process, with the elements of the queues in each step:
Beginning the breadth-first search algorithm on the previous example, the queue status is as follows:

Core0: Core1:

1			4		

Dequeue node 1 from core 0 and mark node 1 as visited, then enqueue its neighbor node 2. At the same time, dequeue node 4 from core 1 and mark node 4 as visited, then enqueue its neighbors node 5 and node 6.

Core0: Core1:

2			5	6	

Dequeue node 2 from core 0 and mark node 2 as visited, and enqueue its neighbor node 3. At the same time, dequeue node 5 from core 1 and mark node 5 as visited. There are no further neighbors enqueued for this core.

Core0: Core1:

3			6		

Following the next step, the queue will be empty, and the breadth-first algorithm has finished its traversal.

Using this process, we can theoretically achieve 2 times speedup running on 2 cores compared with the traditional breadth-first search algorithm. If we continue to increase the number of the cores we will expect to achieve a higher speedup.

:

4 Experimental Results

This section will display the results for each graph operation performed, comparing a serial baseline version of the algorithms against the implemented parallel versions as described in the Operations Implemented section above. Primary measurement metrics for evaluating the system are serial baseline vs parallel runtimes, in addition to the overall speedup produced (serial runtime / parallel runtime) and parallel efficiency (speedup / number of cores) achieved. The testbed utilized for this research is the University of Pittsburgh Center for Research Computing (CRC), which comprises numerous different computing clusters. The shared multiprocessor cluster (SMP) in particular is the one in which all of the measurements have been made, and is equipped with an Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz processor, 192 GB RAM, and two solid state drives of sizes 256GB and 500GB. This allowed the tests to be run and measured on core counts of 16, 8, 4, and 2. Because CRC is a shared computing resource for the university, runtimes may have been impacted due to high utilization among all available nodes. This is able to be alleviated to an extent, however, by programming the code to execute 10 times in a row during a single job submission, taking an average of the runtimes as a final result. If the outputs produced appeared to be skewed or inaccurate, the test would be run again until a more rational result was produced. An additional point to note is that all configurations have been compiled with “-O3” optimizations.

4.1 Graphs Analyzed

The graphs analyzed with the minimum spanning tree and breadth-first search algorithms described are undirected and weighted in nature. They have been produced by the MAWI (Measurement and Analysis on the WIDE Internet) group [12], and were created from packet trace data from the WIDE network backbone that is maintained by this same group. The graphs have been obtained from the GraphChallenge [13] and SuiteSparse Matrix Collection [14] websites, which host a large variety of different graphs for the purpose of conducting research. Five graphs were used of varying sizes (in ascending order according to size): “mawi_201512020030.mtx”, “mawi_201512020130.mtx”, “mawi_201512020330.mtx”, “mawi_201512020130.mtx (newer/larger)”, and “mawi_201512020330.mtx (newer/larger)”. Making use of different sized graphs allows us to analyze the performance scalability of our implementations, and whether or not they are effective as the number of vertices and/or edges increases. The reason for using the graphs “mawi_201512020130.mtx (newer/larger)” and “mawi_201512020330.mtx (newer/larger)” is because it allows us to strictly measure the performance when increasing the edge count within a graph, while keeping the vertex count the same as their smaller counterparts of the same name. The exact details regarding each graph’s details may be seen within Table 1 below. An additional important detail regarding the graphs is that they are not fully connected, with approximately 5% of each graph being disconnected from the larger component. Thus, when analyzing them, the code will pick the larger piece of the graph and choose the root as the vertex with the highest count of edges connected to it.

Name of graph	Memory Size	Vertex Count	Edge Count
mawi_201512020030.mtx	1.32GB	68863315	71707480
mawi_201512020130.mtx	2.53G	128568730	135117420
mawi_201512020330.mtx	4.79GB	226196185	240023949
mawi_201512020130.tsv (newer/larger)	5.06GB	128568730	270234840
mawi_201512020330.tsv (newer/larger)	9.58GB	226196185	480047894

Table 1. Characteristics of Each Graph Tested

4.2 Minimum Spanning Tree

The results of running the parallel Borůvka’s algorithm are organized and displayed within Table 2 two pages down. To ensure the correct minimum spanning tree had been produced by the parallel program, the number of edges added to the tree, as well as the total weight, had been calculated after the operation was completed. These values were then compared against those produced by the serial baseline.

It can be seen that a speedup over the baseline is achieved for all configurations of cores (16, 8, 4, 2) for all graphs tested. The max speedup achieved is 2.29 for 16 cores when

analyzing the “mawi_201512020330.mtx” graph. When compared to the larger version of this graph with more edges, the speedup drops by 0.03, although this may also be the result of contention while running on CRC, rather than an actual drop in performance, and additional testing while utilization is low may provide clarity. 16 cores on graph “mawi_201512020330.mtx” produces a speedup of 1.91, and is improved upon by 0.05 when its larger counterpart is analyzed. This contradiction between speedups makes it uncertain how well the system scales with total edges, and additional datasets may be needed to reach a conclusive result. It may be clearly seen, however, that as the number of vertices within a graph increases, as does the peak speedup achieved.

While the system does achieve speedup over the baseline, the parallel efficiency is quite poor for most configurations. The peak is nearly 66% for 2 cores on the “mawi_201512020130.tsv (newer/larger)” dataset, while reaching a minimum of 12% on 16 cores for almost all graphs. This may possibly be improved upon by making use of more clever graph partitioning methods or ways of distributing data among cores, as there may still be instances in which a single core processes very dense vertices (those with many edges), while others have very sparse vertices (those with few edges). The approach implemented does avoid contention between cores when accessing the main graph adjacency list, however, ensuring that there is minimal slowdown due to memory access of the same location or similar issues.

(Table located on next page so that all results may be seen together)

Name of Graph	Baseline Runtime (seconds)	Cores	Parallel Runtime (seconds)	Speedup	Parallel Efficiency
mawi_201512020030.mtx	12.17	16	6.53	1.86	0.12
		8	6.70	1.82	0.23
		4	8.32	1.46	0.37
		2	10.86	1.12	0.56
mawi_201512020130.mtx	22.42	16	11.75	1.91	0.12
		8	13.40	1.67	0.21
		4	13.99	1.60	0.40
		2	19.03	1.18	0.59
mawi_201512020330.mtx	45.41	16	19.79	2.29	0.14
		8	24.47	1.86	0.23
		4	28.12	1.61	0.40
		2	38.67	1.17	0.59
mawi_201512020130.tsv (newer/larger)	24.35	16	12.43	1.96	0.12
		8	13.94	1.75	0.22
		4	15.77	1.54	0.39
		2	18.47	1.32	0.66
mawi_201512020330.tsv (newer/larger)	51.26	16	22.67	2.26	0.14
		8	25.96	1.97	0.25
		4	31.74	1.62	0.40
		2	44.14	1.16	0.58

Table 2. Minimum Spanning Tree Results

4.3 Breadth-First Search

The results of the parallelized breadth-first search are displayed in Table 3 on the following page. Once again, to ensure correctness, the results of parallel execution were compared against the serial baseline. However, instead of counting edges and weight totals, the number of visited vertices was used to measure accuracy.

It may be seen that for this operation speedups have been achieved over the serial baseline for all graphs, with all configurations of cores, much like the result of the minimum spanning tree operation. Breadth-first search, however, achieved up to 3x speedup on the smaller graphs tested, while maintaining a minimum of approximately 2.5x speedup as their sizes increased. It may be worth experimenting further with even larger graphs, composed of many more vertices and edges, to observe if the speedup continues on a downward trend, or perhaps scales better or worse depending on the sparseness or density of graphs. An additional point that is interesting to note is that in all trials, the speedup for cores are close together in range, all hovering around similar values. This may be due to an uneven distribution of the traversal and poor partitioning, so it would be worth looking into “smarter” ways to distribute work in future research. One possible avenue to attempt is to take all vertices with the most outgoing edges, and begin traversals from there, as opposed to set strides apart in the numerical value of vertices.

The parallel efficiency for the parallel breadth-first search sees an improvement over that experienced by minimum spanning tree, but still leaves some to be desired for core counts 8 and 16. One curious point observed while testing is the superlinear speedup and parallel efficiency experienced when using 2 cores on all configurations. Multiple trials were ran in order to ensure these timings listed below were accurate, and the baseline was also tweaked to ensure that the parallel version did not have any “unfair advantages”. The primary explanation we could come up with for these occurrences is that the multiprocessor execution is simply able to make better use of the cache size available (2x normal), providing for fewer cache misses and fetches to lower memory, in addition to many more cache hits. Further testing and experimenting with both serial and parallel implementations may be needed to come to a conclusive explanation, however.

Name of Graph	Baseline Runtime (seconds)	Cores	Parallel Runtime (seconds)	Speedup	Parallel Efficiency
mawi_201512020030.mtx	2.12	16	0.77	2.76	0.17
		8	0.72	2.96	0.37
		4	0.70	3.02	0.76
		2	0.69	3.08	1.54
mawi_201512020130.mtx	4.02	16	1.34	2.99	0.19
		8	1.31	3.08	0.39
		4	1.35	2.97	0.74
		2	1.31	3.07	1.53
mawi_201512020330.mtx	6.84	16	2.27	3.01	0.19
		8	2.28	2.991	0.37
		4	2.30	2.98	0.74
		2	2.48	2.76	1.38
mawi_201512020130.tsv (newer/larger)	4.46	16	1.80	2.48	0.15
		8	1.69	2.63	0.33
		4	1.70	2.62	0.66
		2	1.73	2.58	1.29
mawi_201512020330.tsv (newer/larger)	8.16	16	2.99	2.73	0.17
		8	3.04	2.68	0.34
		4	3.21	2.54	0.64
		2	2.99	2.72	1.36

Table 3. Breadth-First Search Results

5 Conclusions and Future Work

Throughout this paper we have showcased the minimum spanning tree and breadth-first search operations, as well as our attempts to parallelize and improve their performance. Available algorithms to complete these tasks have been analyzed for parallel potential and chosen appropriately, picking the ones we believe to have the most upside. The various attempts at parallelization have been noted as well, employing different underlying data structures and OpenMP strategies to obtain the best results. We were able to successfully meet our goals and achieve speedup on all graph datasets utilized, despite having parallel efficiencies that point to underutilization and call for better distribution of the work. Thus, we have shown that it is worthwhile to make use of parallel architectures, namely shared memory multiprocessors, in order to accelerate operations done on the highly flexible and popular graph data structures.

We feel that this research has just scratched the surface, however, and have different avenues for future work in mind that may improve the performance beyond that achieved within this paper. First, if better partitioning schemes may be created for the graph operations presented, not only could performance be improved on a single machine, but a distributed system may be leveraged that makes use of a tool such as MPI to coordinate traversals and minimum spanning tree building over a much larger network. In addition, because the results shown in section 4 above display increasing speedup as the number of total vertices increases, finding or generating much larger graphs and processing those would be an interesting experiment. Combining ideas and tests such as these may yield large performance gains over a serial baseline, and allow systems to overcome the largely memory bound operations presented by graphs through even distribution of work and optimized communications between processors.

All code is available for viewing at: https://github.com/jjb169/ECE2166_Project

6 References

[1] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to parallel computing, Second edition*. Addison-Wesley, 2003.

[2] H. Zhou, N. Shenoy, and W. Nicholls, “Efficient minimum spanning tree construction without Delaunay triangulation,” *Proceedings of the 2001 conference on Asia South Pacific design automation - ASP-DAC '01*, Feb. 2001.

[3] S. Chung and A. Condon, “Parallel implementation of Bouvka's minimum spanning tree algorithm,” *Proceedings of International Conference on Parallel Processing*, Apr. 1996.

[4] S. M. Durbhakula, “Parallel Minimum Spanning Tree Algorithms and Evaluation,” May 2020.

[5] V. Lončar, S. Škrbić, and A. Balaž, “Parallelization of minimum spanning tree algorithms using distributed memory architectures,” *Transactions on Engineering Technologies*, pp. 543–554, 2014.

[6]C.E. Leiserson and T.B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In Proc. 22nd ACM Symp. on Parallism in Algorithms and Architectures (SPAA ’10), pages 303–314, June 2010.

[7]Shivapuram, Ashwin, Somay Jain, Chirag Majithia and Virinchi Srinivas ashwinsl. “A Parallel Hybrid Framework for Graph Processing.” (2017).

[8] J. Nešetřil, E. Milková, and H. Nešetřilová, “Otakar Borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history,” *Discrete Mathematics*, vol. 233, no. 1-3, pp. 3–36, 2001.
[9] C. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009.
[10] Buluc, Aydin, Beamer, Scott, Madduri, Kamesh, Asanovic, Krste, and Patterson, David. 2017. "Distributed-Memory Breadth-First Search on Massive Graphs. IN: Parallel Graph Algorithms". United States. https://www.osti.gov/servlets/purl/1398519.
[11] S. Beamer, A. Buluç, K. Asanovic and D. Patterson, "Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search," 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, 2013, pp. 1618-1627, doi: 10.1109/IPDPSW.2013.159.
[12] Kenjiro Cho, Koushirou Mitsuya and Akira Kato. "Traffic Data Repository at the WIDE Project". USENIX 2000 FREENIX Track, San Diego, CA, June 2000.
[13] *GraphChallenge*. [Online]. Available: https://graphchallenge.mit.edu/. [Accessed: Mar-2022].
[14] “Suitesparse Matrix Collection formerly the University of Florida Sparse Matrix Collection,” *SuiteSparse Matrix Collection*. [Online]. Available: https://sparse.tamu.edu/. [Accessed: Mar-2022].