Graph Traversal Breadth First Search (BFS) Optimization

Chen-Yu Chang

EC527 Final Project

May 7th, 2021

Table of Contents

1. Abstract

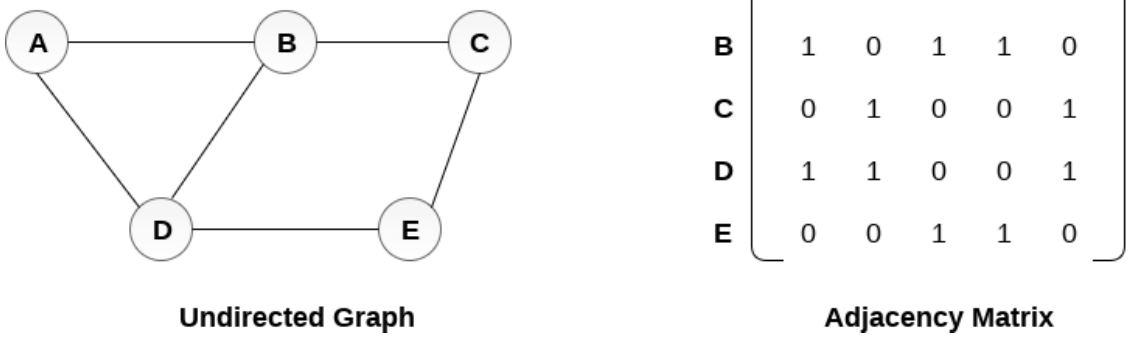
Breadth First Search is a significant implementation of graph processing. Besides from the ordinary solution for the application, such as shortest path and minimum spanning tree for unweighted graph, peer to peer networks, crawlers in search engines, and social networking websites, we discovered deeply in the optimization of breadth first search through parallelization, blocking, and other algorithms, such as hybrid algorithm and Dijkstra's algorithm. We also dug into the performance on CPU and GPU, comparing the efficiency of the different algorithms.

2. Graph Traversal

Graph traversal refers to the process of visiting (checking and/or updating) each vertex in a graph. Such traversals are classified by the order in which the vertices are visited. Compared to tree traversal, graph traversal sometimes requires some vertices being visited more than once, since it is not necessarily known before transitioning to a vertex that has already been explored. As graphs become more dense, this redundancy becomes more prevalent, causing computation time to increase and vice versa.

Thus, it is usually necessary to remember which vertices have already been explored by the algorithm, so that vertices are revisited as infrequently as possible. This may be accomplished by associating each vertex of the graph with color or visitation states during the traversal, which is then checked and updated as the algorithm visits each vertex. If the vertex has already been visited, it is ignored and the path is pursued no further; otherwise, the algorithm checks and updates the vertex and continues down its current path.

In simple words, graph traversal mainly goes through three procedures. Those graphs are converted into several types of graphs that help the computer to recognize its relationships, such as adjacency lists, adjacent matrix, and undirected graphs (Figure 1). Then, the connections between nodes should be discovered, which match from the parent nodes. Checking the visited state and recording it can help prevent it from repeating. Both the depth-first and breadth-first graph searches are adaptations of tree-based algorithms, distinguished primarily by the lack of a structurally determined root vertex and the addition of a data structure to record the traversal's visitation state.



Undirected Graph

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

Figure 1: Undirected Graph and Adjacency Matrix

3. Breadth First Search

Breadth First Search is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing of the same node again, use a boolean array which marks the node after it is processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.

Comparing Breadth First Search to Depth First Search, uses the opposite strategy of breadth-first search. Depth First Search explores the node branch as far as possible before being forced to backtrack and expand other nodes. Depth First Search can be implemented using a stack data structure, which follows the last-in-first-out (LIFO) method that the node that was inserted last will be visited first.

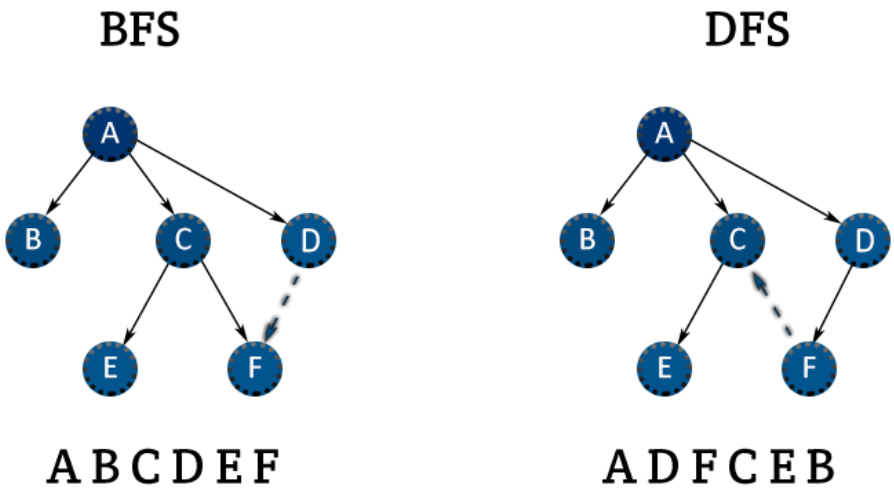


Figure 2: Comparison of BFS and DFS

3.1 Serial Method

For the serial method, we can use a queue to store the node and mark it as visited until all of the current node's neighbours that are directly connected to it are marked as visited, too. The queue follows the First In First Out (FIFO) queuing method, and therefore, the neighbors of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on. For further explanation, we can elaborate the process of the following graph (Figure 3). For example, the graph contains the elements of a, b, c, d, e, f, g. We start from the source node a and enqueue it. We mark a as visited node and dequeue it, but inserting the child nodes into the queue, which are b and c. The queue is not empty and since the order goes from b then c. Thus, we dequeue b and insert with its child nodes d and e. While all of the other nodes work as the same procedure.

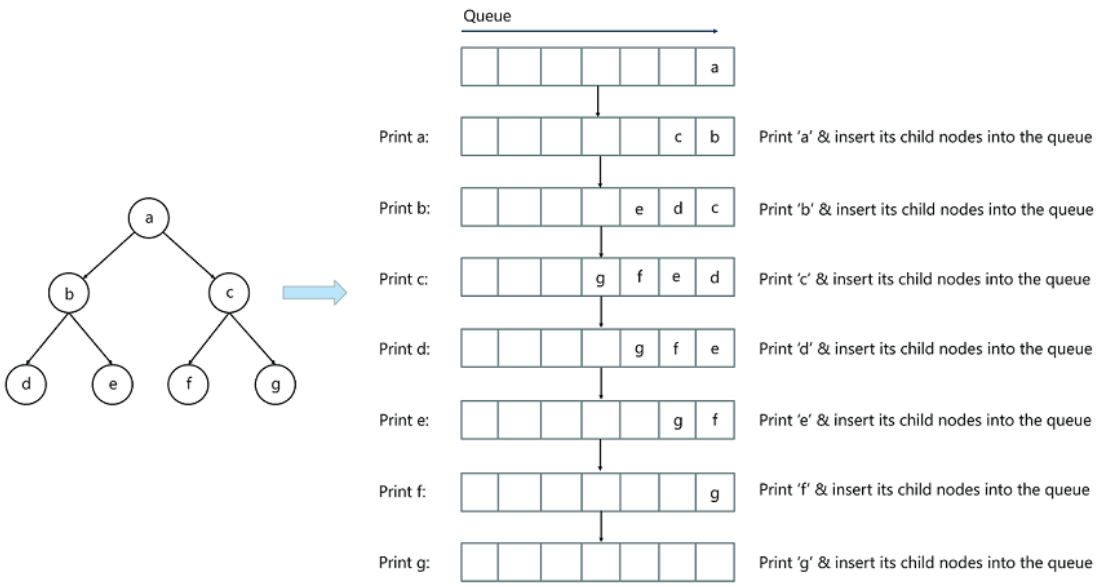


Figure 3: Serial Method Elaboration (Queue)

The key of using a queue is when the node is visited, we add to the queue. After going to the next level, dequeue the previous node and add the new ones. In this pseudocode, we mark the visited nodes with color. The color white represents the nodes that are not visited, while the color black represents the nodes visited. For gray, it indicates the nodes that are about to be visited.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

Figure 4: BFS Serial Pseudocode

One of the challenges we face when using queues to store the data is that we need to set a high capacity for the queue if we want to operate breadth first search with a large number of nodes. Furthermore, in a huge graph, breadth first search will operate very slowly since all the other levels of nodes have to wait for the previous nodes to be visited. Memory access is another constraint we faced that the memory space is $O(V^2)$, storing the data including parents, visited, queue, and graph arrays.

3.2 Hybrid Method

Breadth First Search is a significant building block of many graph algorithms. Starting from the source node, frontier expands outward and visits all the vertices on the same level. ***Top-Down*** method checks all of the neighbors whether they are visited or not. If not, they will be added to the frontier and marked visited by setting its parent variable. Top-Down method is suitable for small-world and scale-free networks, which have a low effective diameter. This method is more efficient on small frontiers. In other words, it will loop the current frontier and its neighbors. If there are any nodes not visited, it will be added to the next frontier. Top-Down method shows its performance of O(E), while E represents the number of edges in the graph.

```
function top-down-step(vertices, frontier, next, parents)
    for v ∈ frontier do
        for n ∈ neighbors[v] do
            if parents[n] = -1 then
                parents[n] ← v
                next ← next ∪ {n}
            end if
        end for
    end for
```

Figure 5: Top-Down Algorithm

When the frontier is large, operating breadth first search reversely might improve the efficiency. Doing reverse search, which is called ***Bottom-Up*** Algorithm, it finds the node's parents that are not in the current frontier. Instead of making each vertex in the frontier attempt to become the parent of the neighbors, each vertex will find its own parent among the neighbors. By operating Bottom-Up, once one vertex finds its own parent, we do not have to check the others. The Bottom-Up method shows its performance of O(NV+E), where N represents the number of neighbors, V represents the number of vertices, and E represents the number of edges.

```
function bottom-up-step(vertices, frontier, next, parents)
    for v ∈ vertices do
      if parents[v] = -1 then
        for n ∈ neighbors[v] do
          if n ∈ frontier then
            parents[v] ← n
            next ← next ∪ {v}
            break
          end if
        end for
      end if
    end for
```

Figure 6: Bottom-Up Algorithm

The combination of Top-Down method and Bottom-Up method improves the overall efficiency of breadth first search algorithm. Hybrid Algorithm determines when to operate Up-Down method and when to choose Bottom-Up method. When the frontier is large, the Bottom-Up will be utilized as the best approach, while Top-Down method will instead perform its worst.

```
function HYBRID-BFS(vertices, source)
    frontier ← {source}
    next ← {}
    parents ← [-1,-1,···,-1]
    while frontier ≠ {} do
        if next-direction() = top-down then
            top-down-step (vertices, frontier, next, parents)
        else
            bottom-up-step (vertices, frontier, next, parents)
        end if
        frontier ← next
        next ← {}
    end while
    return parents
end function
```

Figure 7: Hybrid Algorithm

To control the situation when to use bottom-top and top-down algorithms, we initialized a heuristic based on the number of edges to check from the frontier, the number of vertices in the frontier, and the number of edges to check from the unvisited frontier. We use two parameters, α and β, to tune the algorithms. When the number of edges to check from the frontier is smaller than the unvisited edges to check dividing by α or when the number of vertices in the frontier is smaller than vertices dividing by β, the hybrid algorithm will choose to operate top-down algorithm. On the other hand, when the number of edges to check from the frontier is larger than the unvisited edges to check

dividing by α or when the number of vertices in the frontier is larger than vertices dividing by β, the algorithm chooses to operate bottom-up method.

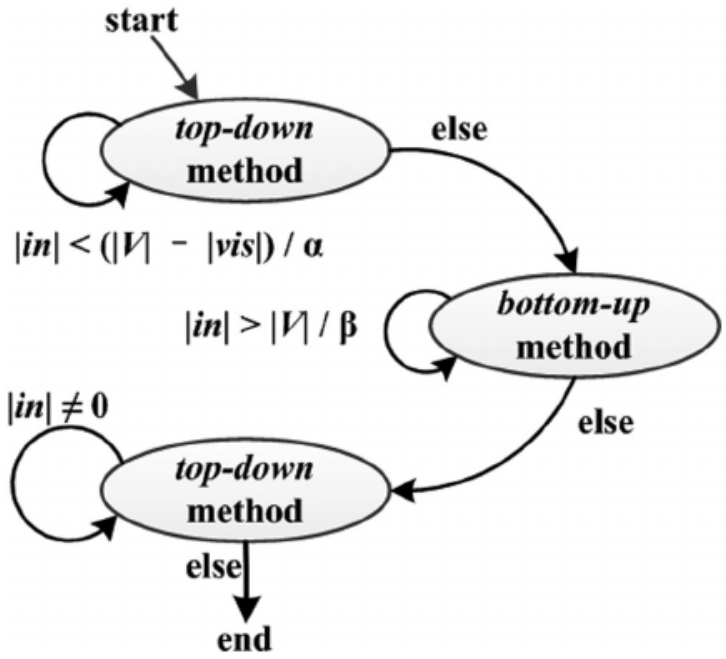


Figure 8: State Machine of Hybrid Algorithm

For tuning α and β, we first tune α before tuning β since α has a greater influence on the decision of which algorithm to use. Going through many values of α in a wide range, breadth first search seems to be optimized when α is set to 1024 when we test for 32768 nodes in a graph. α's usage is to determine when the frontier is too large and compensate for the finish of the bottom-up algorithm. Then, we start to tune β. While β is not too influential compared to α, the change of β does not increase the efficiency of breadth first search remarkably. However, we still tuned the value of β to 16 with the test of 32768 nodes. β's usage is to check whether the frontier is in the appropriate size and compensate for the finish of the top-down algorithm.

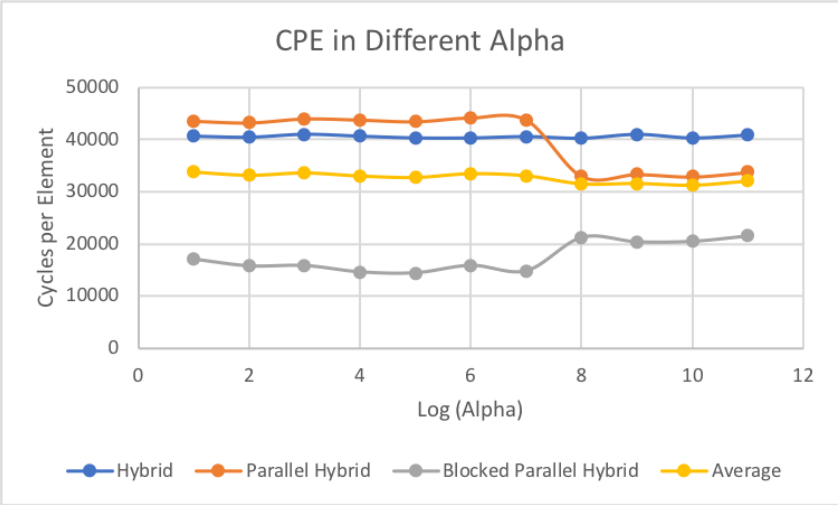


Figure 9: CPE for different number of α

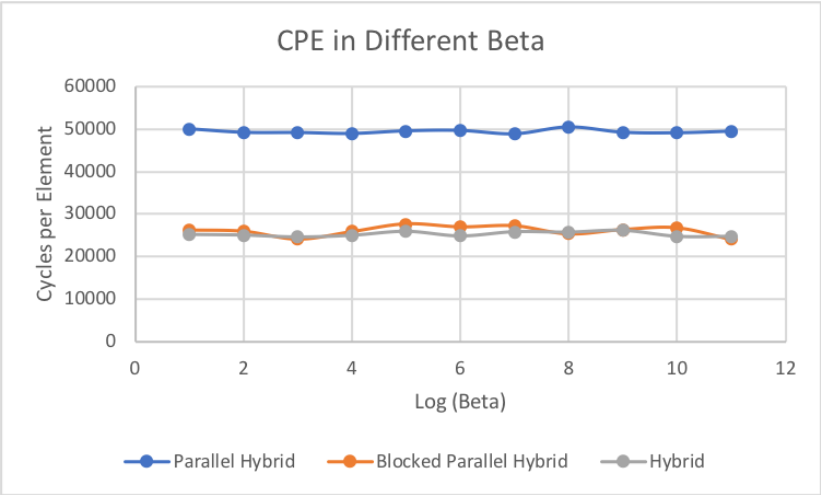


Figure 10: CPE for different number of β

Overall, comparing the performance of the serial method with the hybrid algorithm, the efficiency of the hybrid algorithm is much higher than the serial code. As shown in Figure 11, as the number of nodes increases, the hybrid algorithm seems to decrease the number of iterations better than the serial method. For a small number of nodes, serial code performs better since the simple way will be faster for smaller graphs.

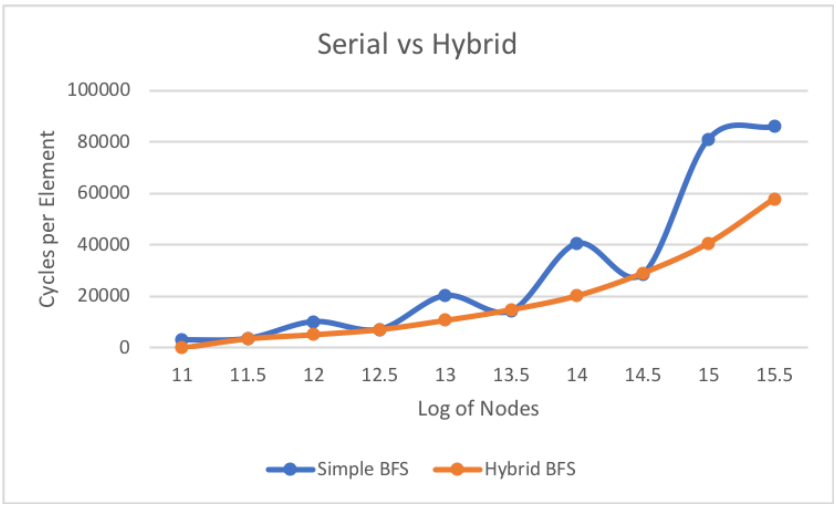


Figure 11: CPE Comparison of Serial and Hybrid Algorithms

3.3 Parallelization

For parallelizing breadth first search with serial codes, we can share the memory for faster speed. Shared memory provides higher memory-bandwidth and lower latency since all processors share the memory together, and all of them have direct access to it. Thus, we don't need to program message passing processes, which is necessary for distributed memory to get data from remote local memory. Therefore, the overhead of messages is avoided. We need to store the current frontier and the next vertex frontier and make the next vertex frontier be switched to the current frontier at the last of each step. These two data can be held in every processing entity(such as thread) which supports data locality but needs extra load balancing mechanisms. For example, in Figure 12, we see an adjacent node from S, which is A, B, C. We find A and enqueue it. Then, we enqueue and mark visited B that is adjacent from S. Finally, the next adjacent node C is visited and enqueued. These tasks are all implemented at the same time on different threads, which successfully synchronize the breadth first search.

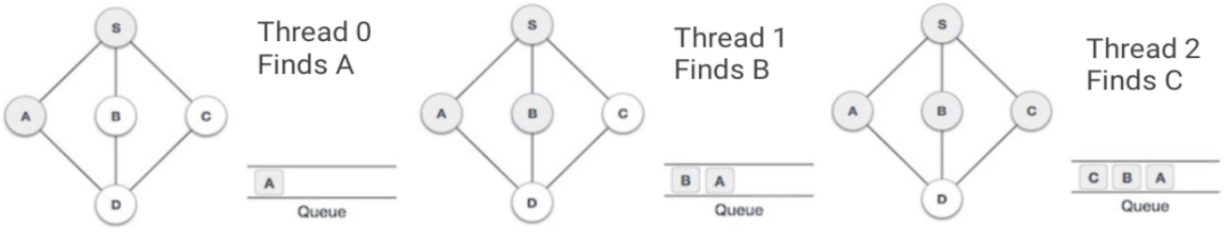


Figure 12: Example of Parallelization of Breadth First Search

The heads of the queue are all shared to all the threads, and the threads will split into several neighbors to visit like Figure 12. The queue here is locked and needs to wait for the next update of the queue; otherwise, it stays in rest. From the data performance, we found that the performance of parallelization improves. However, the serial implementation is quite unstable, which sometimes performs better than the parallel implementation. This is because the parallelized codes will spend more time on waiting for updates.

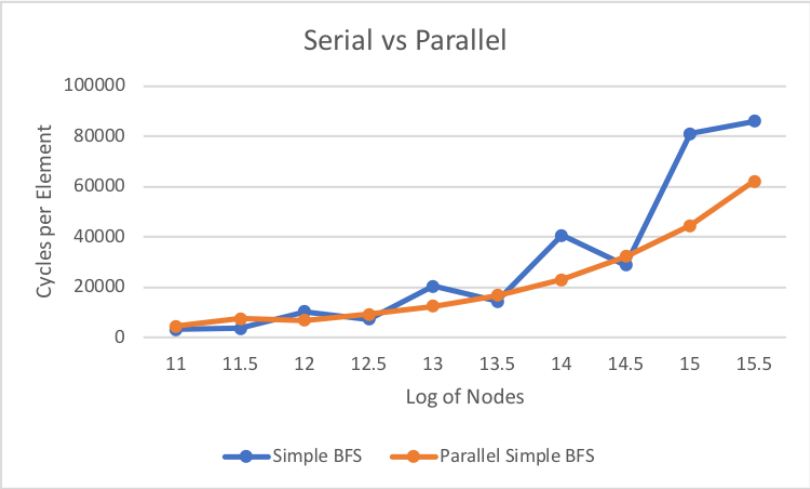


Figure 13: Serial vs Parallel

Furthermore, maintaining the strength of choosing the more efficient way of searching, the hybrid algorithm was improved on the level of synchronization. We set barriers on each level; for example, we implement the root node in thread 0. Then, we set a barrier to enter the next level that thread 0 and 1 are going to synchronize operating the next frontier at the same time. A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier. Once all subgroups have done their synchronizations, the first thread in each subgroup enters the second level for further synchronization. The threads in the next group form new subgroups of k threads and synchronize within groups, sending out one thread in each subgroup to next level and so on.

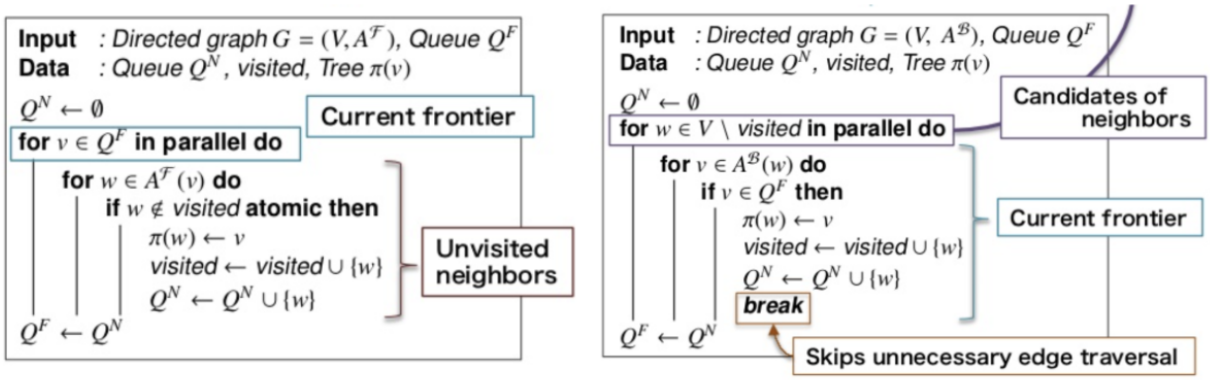


Figure 14: Parallel Hybrid Algorithm

3.4 Blocked Parallelization

Blocking is a well-known optimization technique for improving the effectiveness of memory hierarchies. Instead of operating on entire rows or columns of an array, blocked algorithms operate on submatrices or blocks, so that data loaded into the faster levels of the memory hierarchy are reused. For different values we set for the block size, the efficiency of breadth first search also changes. To decrease the temporal locality, we tested a range of block size from 2 to over 1000, and we found that as the block size increases, the cycles per iteration decrease. Specifically, when the block size is set to 1024 for 32768 nodes, the speed appears to be the fastest.

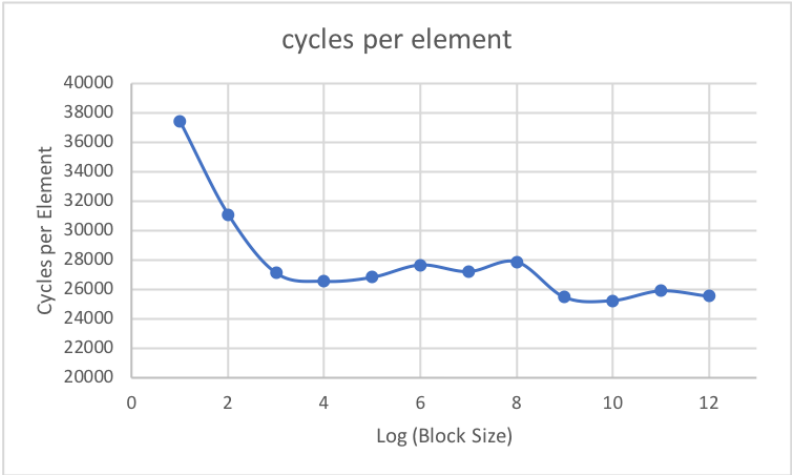


Figure 15: Parallel Blocked Hybrid Algorithm

Comparing the hybrid, parallel hybrid, and parallel blocked hybrid algorithms, we discovered that for breadth first search, both multi-threads and parallelization are efficient optimization for a large number of nodes.

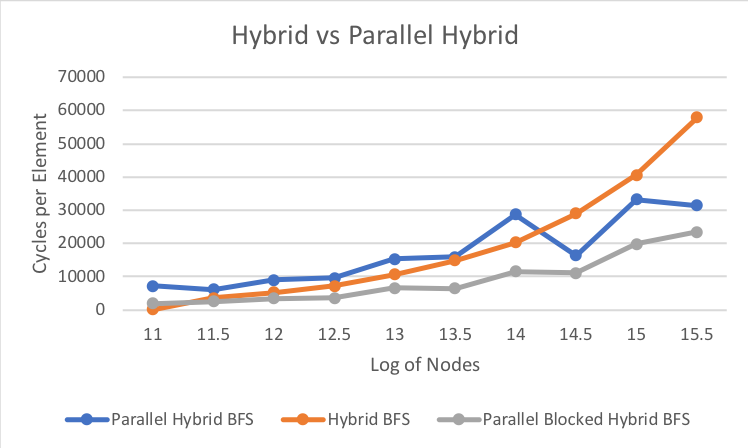


Figure 16: Comparison of Hybrid, Parallel Hybrid, and Parallel Blocked Hybrid Algorithms

In conclusion, the common methods for optimization tend to perform well on breadth first search, while the larger the graph is, the more obviously efficient the operations will be. According to the data we gathered, the ranking of the different algorithms from slowest to fastest is serial code, parallel code, hybrid, parallel hybrid, and parallel blocked hybrid approach.

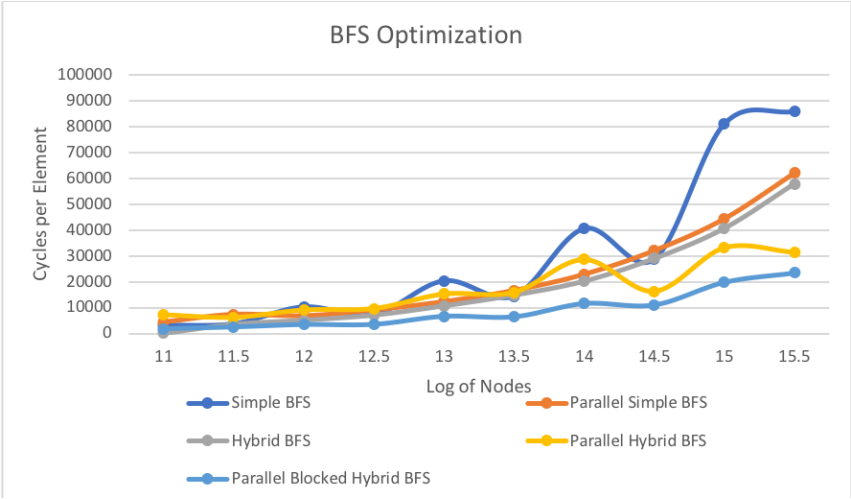


Figure 17: Comparison of all optimization algorithms

4. GPU

Breadth-first search (BFS) is a core primitive for graph traversal and a basis for many higher-level graph analysis algorithms. It is also representative of a class of parallel computations whose memory accesses and work distribution are both irregular and data-dependent. Recent work has demonstrated the plausibility of GPU sparse graph traversal, but has tended to focus on asymptotically inefficient algorithms that perform poorly on graphs with non-trivial diameter. We are particularly interested in the

implementation of the optimization of Dijkstra's Algorithm for the purpose of completing breadth first search.

4.1 Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path between two given nodes, but a more common variant fixes a single node as the source node and finds shortest paths from the source to all other nodes in the graph. We maintain two sets: one set contains vertices included in the shortest path tree, while the other set includes vertices not yet included in the shortest path tree. At every step of the algorithm, we find a vertex which is in the other set and has a minimum distance from the source.

Dijkstra's algorithm uses this idea to come up with a greedy approach. In each step, we choose the node with the shortest path. We fix this cost and add this node's neighbors to the queue. Therefore, the queue must be able to order the nodes inside it based on the smallest cost. We can consider using a priority queue to achieve this. The following figure shows the difference that Dijkstra's uses a priority queue data structure to keep track of the frontier of unvisited nodes. Breadth-first search uses a regular queue data structure. Operations on a priority queue are O(log n). Operations on a regular queue are O(1). Therefore, the use of a regular queue in BFS is made possible by all edge weights being 1, which makes the regular queue effectively behave as a priority queue.

	BFS	**Dijkstra**
Main Concept	Visit nodes level by level based on the closest to the source	In each step, visit the node with the lowest cost
Optimality	Gives an optimal solution for unweighted graphs or weighted ones with equal weights	Gives an optimal solution for both weighted and unweighted graphs
Queue Type	Simple queue	Priority queue
Time Complexity	$O(V + E)$	$O(V + E(logV))$

Figure 18: Difference between BFS and Dijkstra' Algorithm

To implement Dijkstra's algorithm as breadth-first search, we can consider breadth-first search is just Dijkstra's algorithm with all edge weights equal to 1. We initialize all node weights to infinity except for the source node and mark the source node as settled after this point. Then, we choose a node which is relaxed and settled and to relax the outgoing edges of each settled node. Finally we mark all the settled nodes.

4.2 Optimization

To optimize the breadth first search of Dijkstra's Algorithm, we initialize the values of the node weights to infinity except for the source node which is set to 0. We then mark the source node to be settled after that and all other nodes unsettled. Next, we choose the node with minimum node weight whose edges are to be relaxed. Finally, we relax all the edges of a node.

Algorithm 1 GPU implementation of Dijkstra's algorithm. CUDA kernels are delimited by <<< ... >>> .

```
1: <<<initialize>>> (U, F, δ);          //Initialization
2: while (Δ ≠ ∞) do
3:    <<<relax>>> (U, F, δ);            //Edge relaxation
4:    Δ =<<<minimum>>> (U, δ);          //Settlement step_1
5:    <<<update>>> (U, F, δ, Δ);        //Settlement step_2
6: end while
```

Algorithm 2 Pseudo-code of a CUDA thread in *relax kernel*.

```
1: tid = thread.Id;
2: if (F[tid] == TRUE) then
3:    for all j successor of tid do
4:       if (U[j] == TRUE) then
5:          BEGIN ATOMIC REGION
6:             δ[j] = min{δ[j], δ[tid] + w(tid, j)};
7:          END ATOMIC REGION
8:       end if
9:    end for
10: end if
```

Algorithm 3 Pseudo-code of a CUDA thread in *update kernel*.

```
1: tid = thread.Id;
2: F[tid]= FALSE;
3: if (U[tid]==TRUE and δ[tid] <= Δ) then
4:    U[tid]= FALSE;
5:    F[tid]= TRUE;
6: end if
```

Figure 19: Optimization of Dijkstra's Algorithm for BFS

4.3 Comparison

By comparing the data of the unoptimized Dijkstra's algorithm and the optimized one, we found that the optimization is quite remarkably efficient. For the serial Dijkstra's algorithm, when the number of vertices increases, the time it takes increases. However, the optimized code does not have influence on the runtime as the number of vertices increases under the fixed number of 4000 nodes and the weight of each edge of 5.

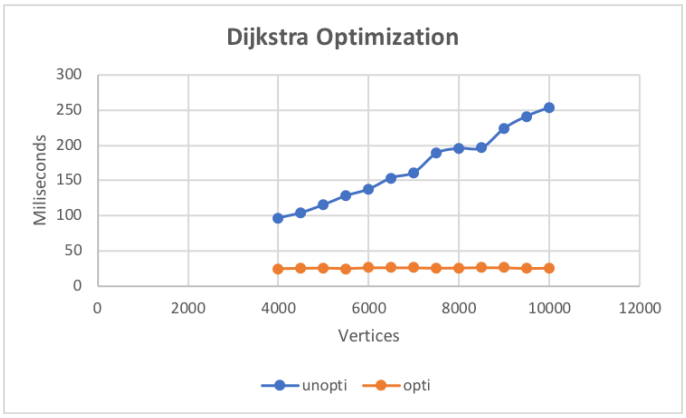


Figure 20: Comparison of Optimization in Dijkstra's Algorithm

We also compared the serial code of breadth first search and the parallelized breadth first search on GPU. In our data collection, we found that on GPU, it took 3.81 seconds to execute the breadth first search for the serial method. Instead, when it runs in parallel, it took 0.02636 seconds to finish executing the parallel approach, which is about 144.554 times faster. Overall, running on GPU is also faster than running on CPU.

5. Closing

Performing breadth first search with a different algorithm, the hybrid algorithm, improves its efficiency. The bottom-up technique is advantageous when the search is on a large connected component of low-effective diameter, as then the frontier will include a substantial fraction of the total vertices. The top-down approach works well for the beginning and end of such a search, where the frontier is a small fraction of the total vertices. With the optimization of parallelization with threads and blocking, the efficient algorithm has become more effective if we tune the parameters correctly. On GPU, we endeavored to operate Dijkstra's algorithm to breadth first search by setting all the edges weight to 1. It turns out that working with this algorithm also executed well and increased the speed compared to the serial codes. Coding both on CPU and GPU, we also found that GPU was several times faster. This work presents an algorithmic innovation to accelerate the processing of more difficult-to-parallelize BFS.

6. Reference

Graph Traversal: https://en.wikipedia.org/wiki/Graph_traversal

Figure 1: https://www.javatpoint.com/graph-representation

Breadth First Search: https://en.wikipedia.org/wiki/Breadth-first_search

Breadth First Search: https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

BFS vs DFS: https://open4tech.com/bfs-vs-dfs/

Figure 2: https://open4tech.com/bfs-vs-dfs/

Figure 3: https://www.edureka.co/blog/breadth-first-search-algorithm/

Figure 4: https://rusyasoft.github.io/algorithms/2019/05/20/cormen-graphs-bfs/

Scott Beamer, Krste Asanović, David Patterson. Direction-Optimizing Breadth First Search, EECS Department, University of California, Berkeley

Figure 5: http://www.scottbeamer.net/pubs/beamer-sc2012.pdf

Figure 6: http://www.scottbeamer.net/pubs/beamer-sc2012.pdf

Figure 7: https://link.springer.com/article/10.1007/s41019-016-0024-y

Figure 8: https://www.researchgate.net/figure/The-state-machine-of-the-hybrid-BFS-algorithm_fig1_260824238

Parallel Breadth First Search: https://en.wikipedia.org/wiki/Parallel_breadth-first_search

Dijkstra's Algorithm: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

Difference between Dijkstra's Algorithm and Breadth First Search: https://www.baeldung.com/cs/graph-algorithms-bfs-dijkstra

Lalinthip Tangjittaweechai. Parallel Shortest Path Algorithms for Graphics Processing Units, Asian Institute of Technology School of Engineering and Technology