Parallel Breadth First Search

Course: CSE 633 Parallel Algorithms

Instructor: Dr. Russ Miller

Presenter: Zhenyi Shen

University at Buffalo The State University of New York

Presentation Contents:

Breadth-First Search(BFS) Introduction

Why we need BFS

How Sequential BFS work

Why we need parallel BFS

Parallel BFS implementation

Execution time

Speedup

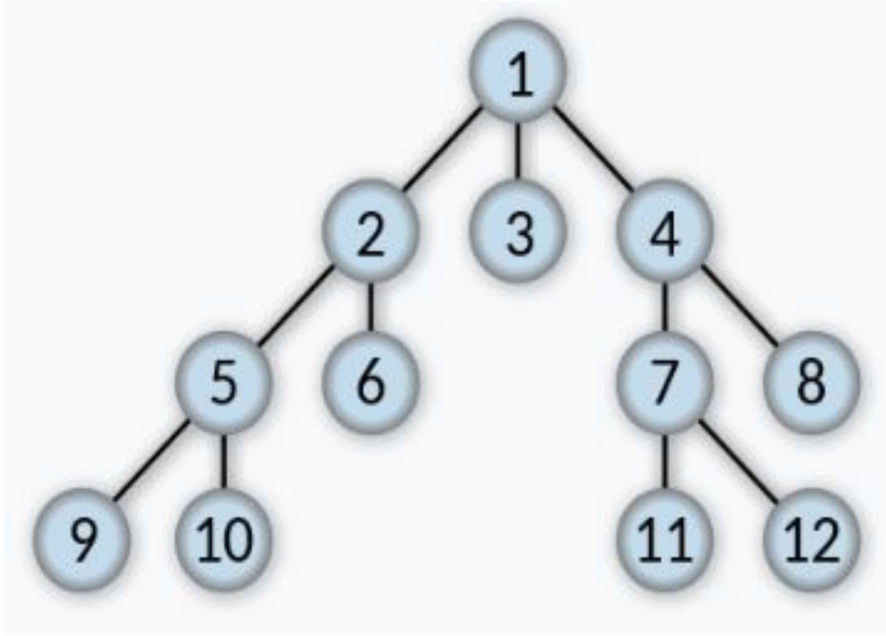
Conclusion

Reference

Breadth-First Search(BFS) Introduction

- Breadth First Search (aka BFS) algorithm is a graph traversal algorithm which is used to explore a tree or graph by visiting each node in breadth-first order

- BFS algorithm starts from a given node (we typically call it root node), and then visit all the nodes at the same level/height. If all nodes at level n are visited, it will then goes to level n+1



Refer: Breadth-first search - Wikipedia

Why we need BFS

- ❑ BFS can be used to **find the shortest path in an unweighted graph**, such as finding the shortest path between two locations for a navigation system
- ❑ BFS can be used to **build and analysis the social network as the fundamental algorithm**, such as finding the shortest path between two users in Facebook or Twitter
- ❑ BFS can be used to **solve puzzles**, such as Rubik's Cube game
- ❑ BFS can be used to **crawl web**, such as web page explore or search engines in Google

How Sequential BFS work

```
❑ BFS(G, src):                                  // G is a graph and src  is root node
    ❑ queue = Queue()                           // Create an empty queue to store the nodes to visit
    ❑ queue.enqueue(src)                        // Enqueue the root node R
    ❑ visited = set()                           // Create an empty set to store visited nodes
    ❑ visited.add(src)                          // Mark root node as visited
    ❑ while queue.isEmpty() == False:           // While loop until queue is empty
        ❑ u = queue.dequeue()                   // Dequeue the first node from the queue
        ❑ If u have neighbors:                  // If node u have neighbor nodes
            ❑ for neighbor in G.adjacency_list[u]:     // use loop to iterate each neighbor nodes of node u in graph G
                ❑ If neighbor not in node_visited:     // we only care about unvisited neighbor
                    ❑ visited.add(neighbor)            // Mark neighbor node as visited
                    ❑ queue.enqueue(neighbor)          // Enqueue the neighbor node
```

Continued - Sequential BFS

Input data: (G: Graph, src, goal)

:param G: graph with set of vertex, list of edge, and map of adjcenc_list

:param type: Graph

:param src: source node

:param type: Any

:param goal: target node

:param type: Any

Output data

:return: a boolean to show whether target node is find or not

:rtype: Boolean

Why we need parallel BFS

- **Better performance**: we can have multiple processors to process multiple nodes in parallel
- **Better scalability**: parallel BFS can work with large graphs more efficiently.
- **Better Flexibility**: there are various ways to implement parallel BFS, so parallel BFS can fit different environment requirements.

Parallel BFS Implementation

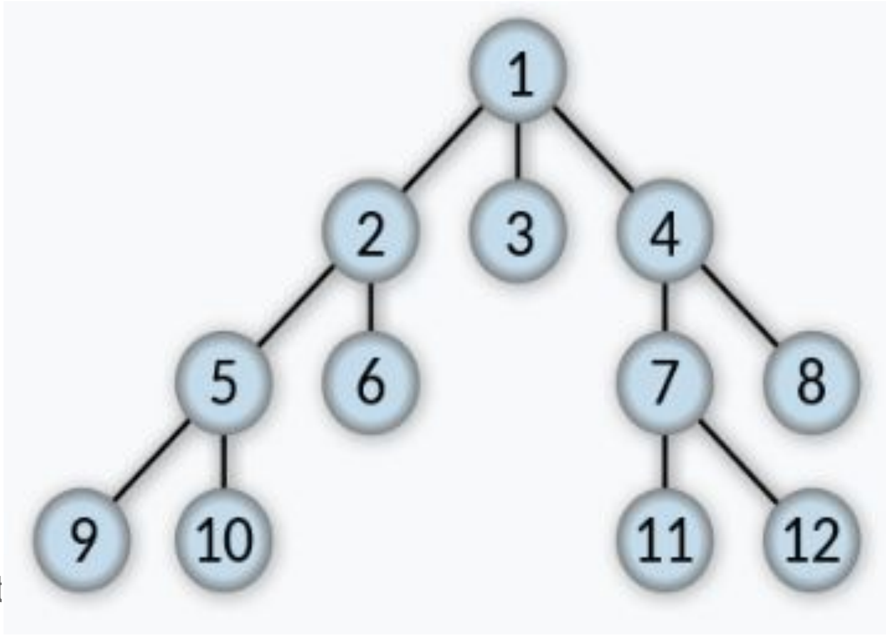
Parallel BFS is similar with sequential BFS, the difference is:

1. Use rank 0 broadcast each level to other ranks
2. Once other ranks received broadcast, they will handle part of nodes in that level
3. Once other ranks have processed data, rank 0 will gather those data and do step 1 again
4. Once there is no data to broadcast or find the target node, we done.

Continued - Parallel BFS Implementation

Assume our Graph is right graph, src is 1, goal is 6.

1. Rank 0 get [2,3,4] as data from (Graph, src)
2. Rank 0 broadcast [2,3,4], and set its data as None
3. Rank 1 handle 2 in [2,3,4], so its data is [5,6]
4. Rank 2 handle 3 in [2,3,4], so its data is None
5. Rank 3 handle 4 in [2,3,4], so its data is [7,8]
6. After Rank 0 gather data, its data is [None, [5,6], None, [7,8]]
7. Rank 0 find 6, we done (if we say goal is 10, its dat will convert into [5,6,7,8] and do step2 - step7 until data = [] or find the goal

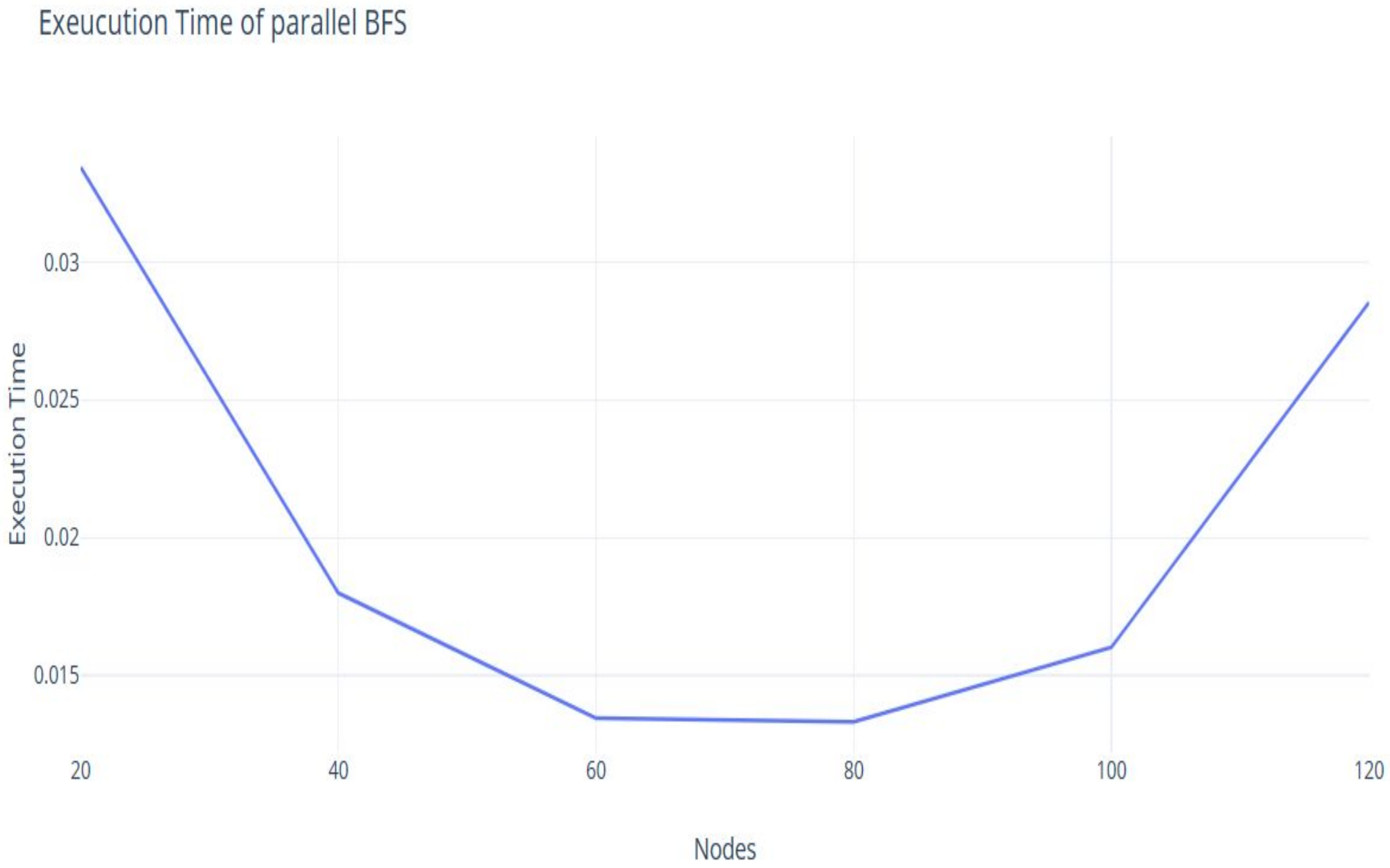


Refer: Breadth-first search - Wikipedia

Execution time

Vertices: 500

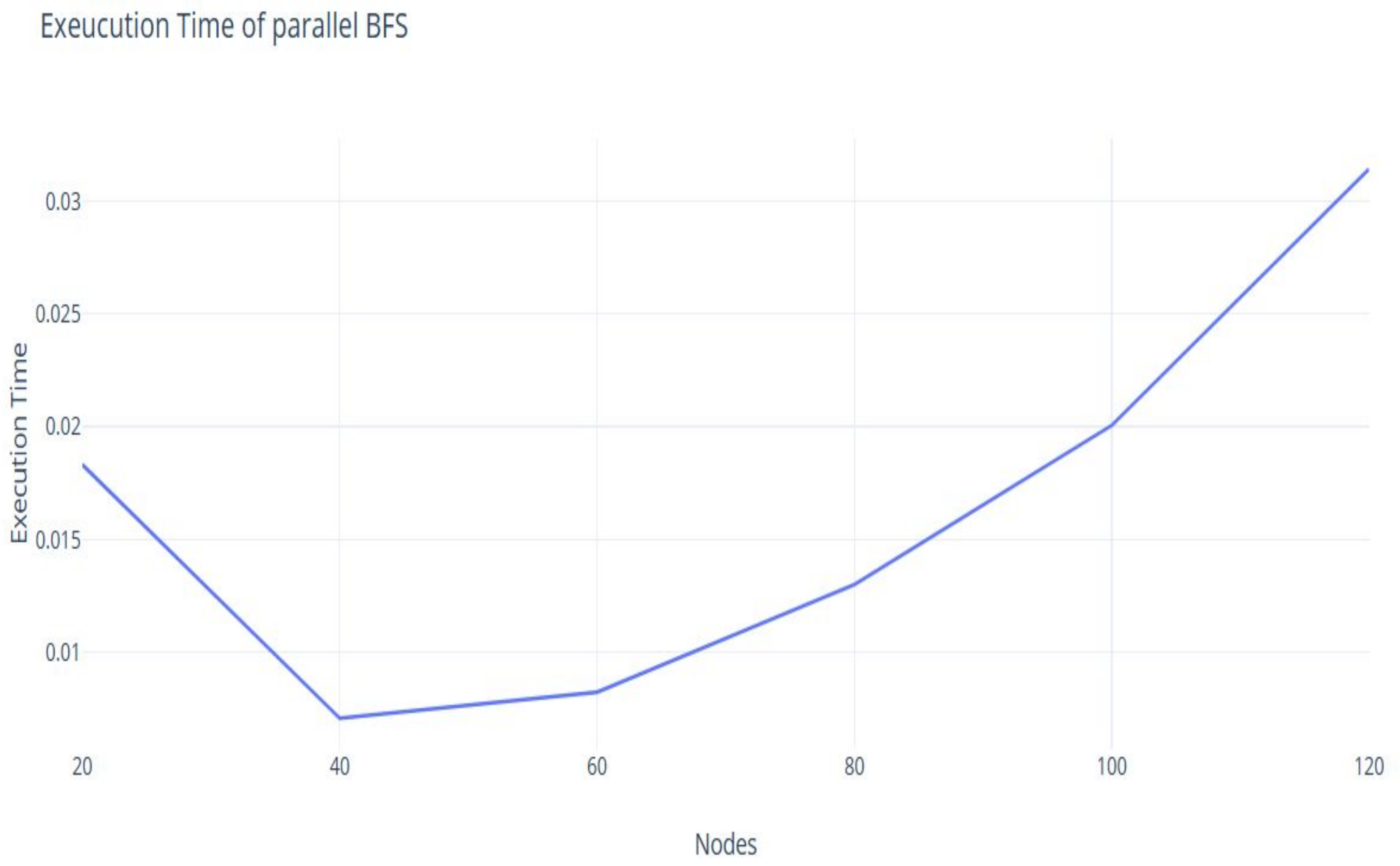
Nodes: 20 - 120



Execution time

Vertices: 1000

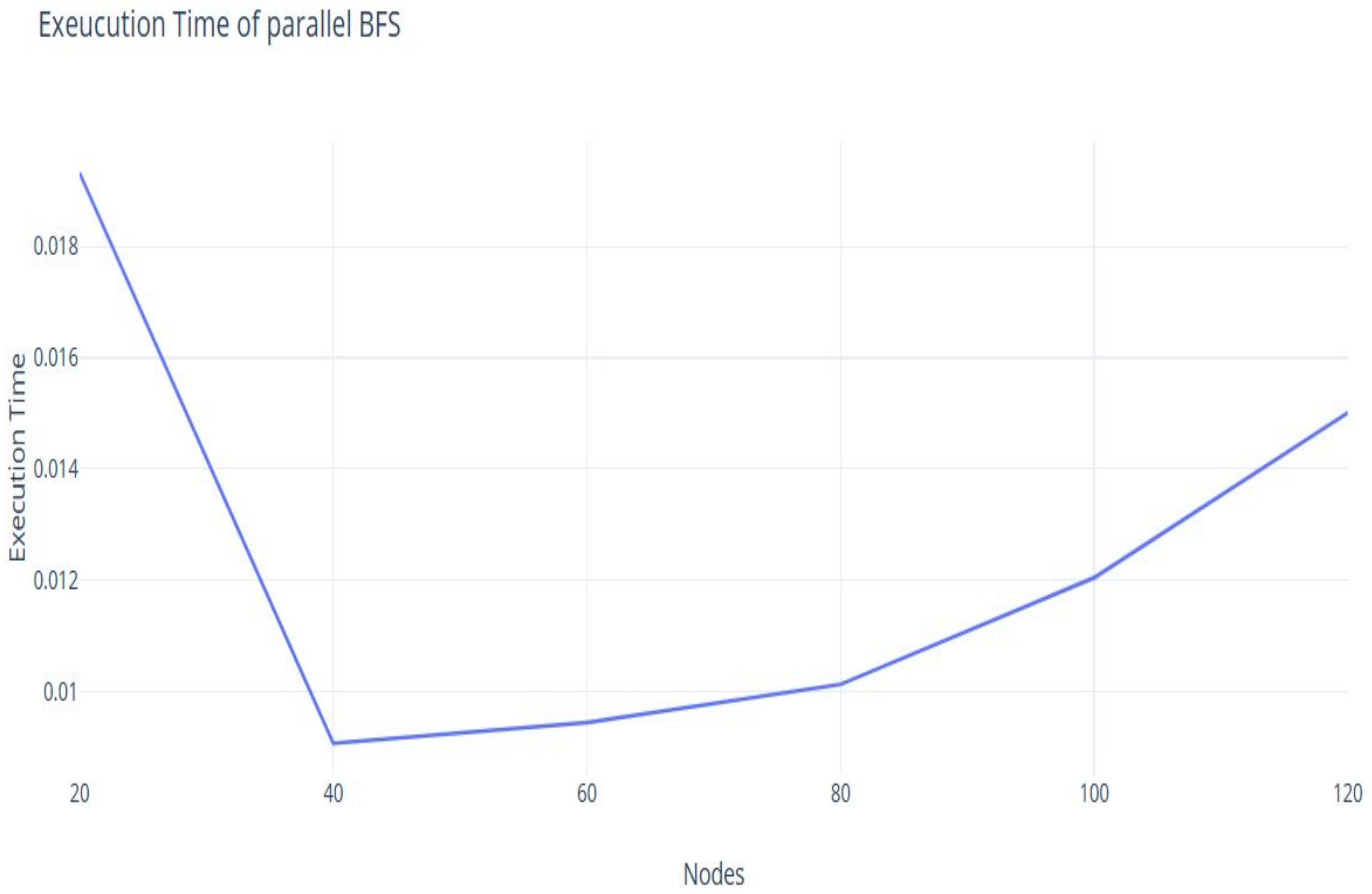
Nodes: 20 - 120



Execution time

Vertices: 1500

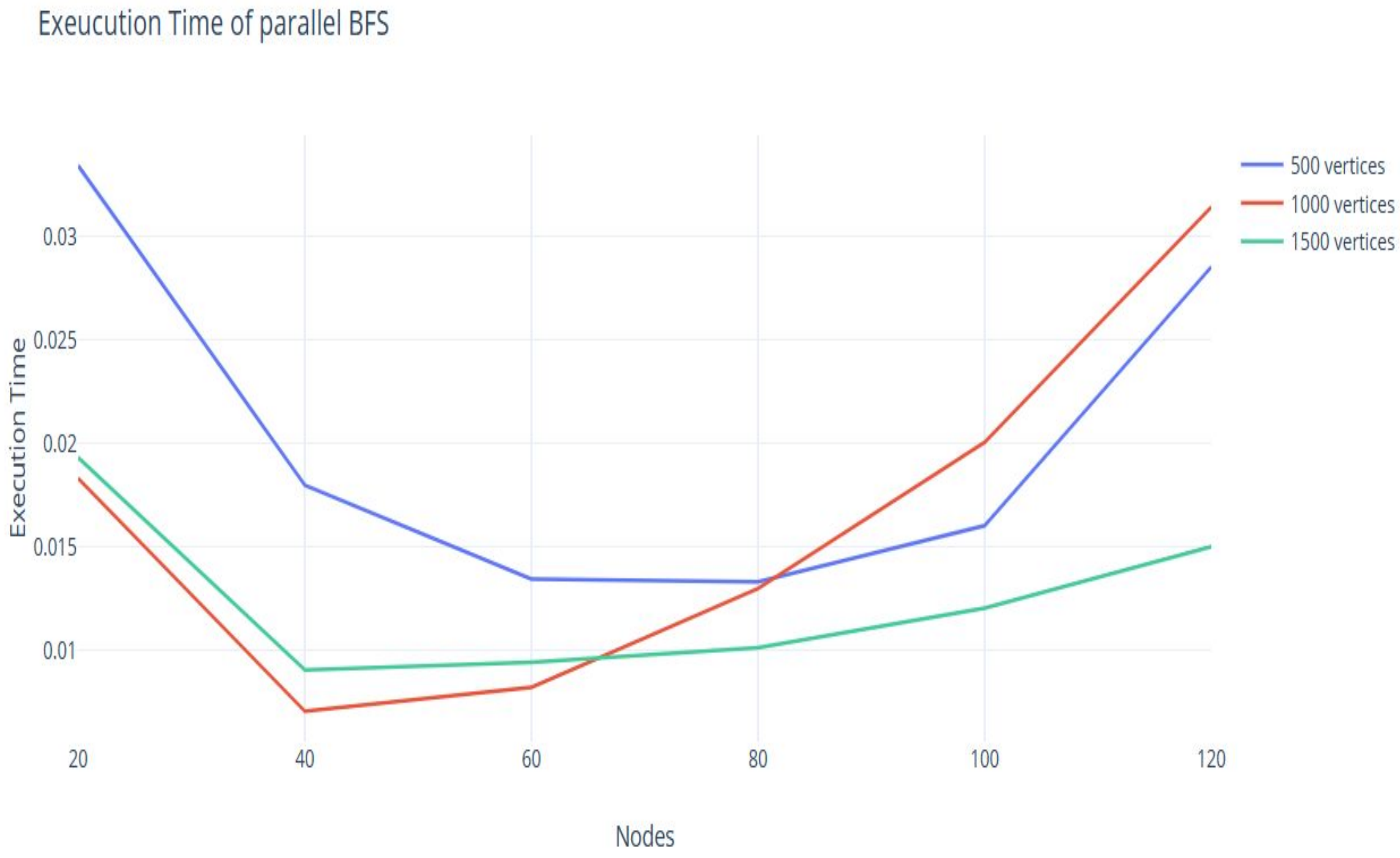
Nodes: 20 - 120



Execution time

Vertices: 500, 1000, 1500

Nodes: 20 - 120



Scaling Concept

Scaling Concept: Scaling means the relative between the performance of a parallel and serial implementation

Definition of Scaling: Speedup Factor S(p)

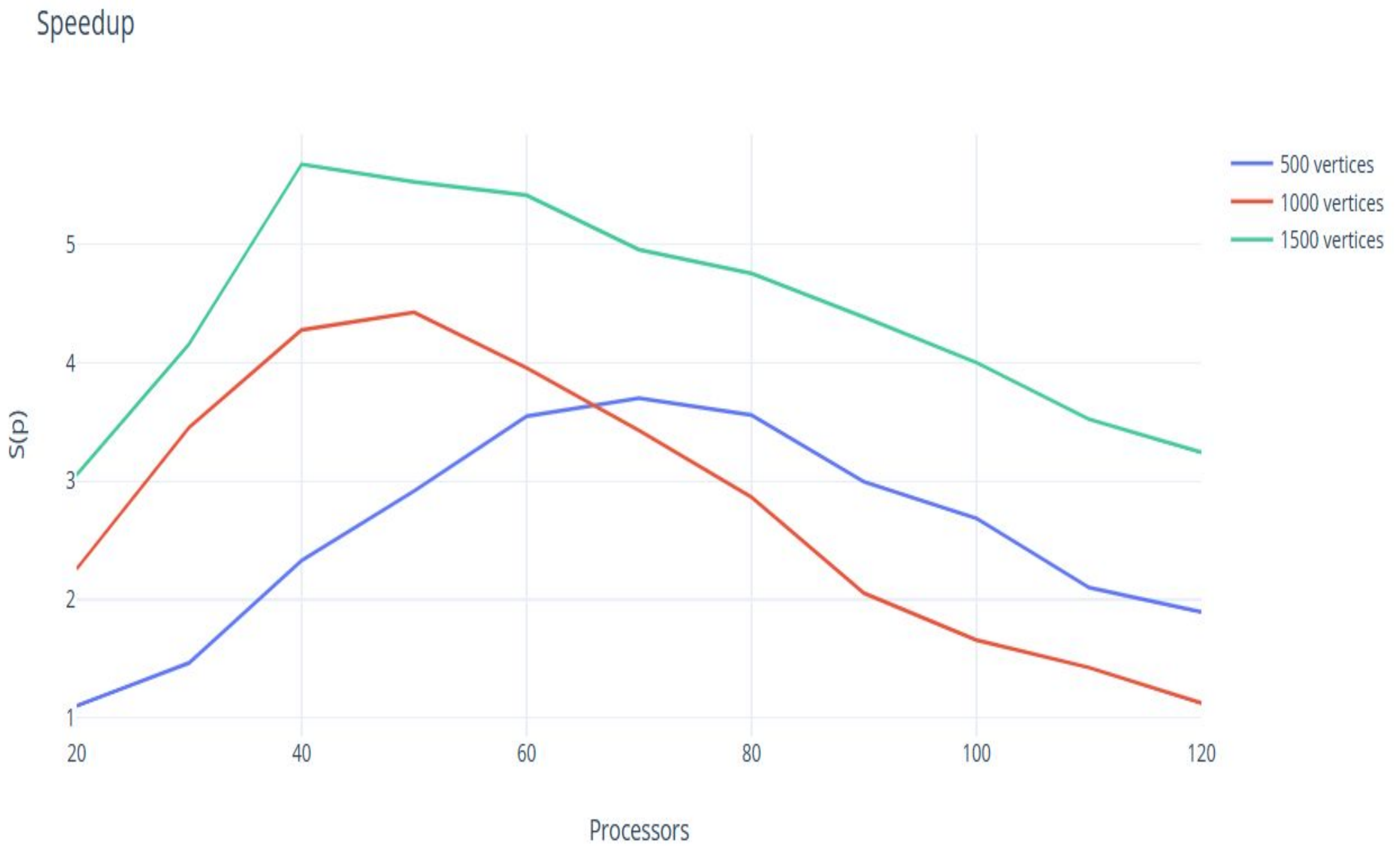
Equation of Scaling:

$$S(p) = \frac{\textit{Sequential execution time (using optimal implementation)}}{\textit{Parallel execution time using } p \textit{ processors}}$$

Speedup

Vertices: 500, 1000, 1500

Nodes: 120



Conclusion

From graphs, we can see that parallel BFS algorithm worked more efficiently for larger vertices

For the trends of Speedup graph, it showed that parallel BFS algorithm worked efficiently under a certain number of processors. However, with the increasing number of processors, the efficiency of the parallel algorithm decrease.

Since larger vertices related with larger speedup value, it looks like parallel BFS algorithm works more better than sequential BFS algorithm for input size of large vertices.

View the maximum point of Speedup graph, with the increasement of the vertices, we observe the position of maximum point moves to left, which means we only need less processors to fit the maximum speedup point if the vertices be large.

University at Buffalo The State University of New York

Reference

- https://en.wikipedia.org/wiki/Parallel_breadth-first_search
- Center for Computational Research (buffalo.edu)
- Tutorials, Workshops and Training Documents : Center for Computational Research (freshdesk.com)
- Parallel Computing - MATLAB & Simulink (mathworks.com)
- Academic Compute Partitions Hardware Specs - Center for Computational Research - University at Buffalo
- A Work-Efficient Parallel Breadth-First Search Algorithm (or How to Cope with the Nondeterminism of Reducer Hyperobjects) (mit.edu)

Thank you