Parallel Graph Algorithms

Yajur Ahuja (ya2109), Utkarsh Khandelwal (uk2051), Shuvadeep Saha (ss15592)

INTRODUCTION

A graph is a data structure consisting of vertices and edges. Edges represent the relationship between vertices. They have applications in modeling real-world networks, web graphs, networks in biology, social networks, unstructured meshes in scientific simulation, etc. In recent times due to the enormous availability of data, a graph can have billions and even trillions of edges. For example, Yahoo has approximately 6.6 billion edges, and Facebook has roughly 1 trillion edges. Due to their vast sizes, there are various algorithms to process graphs on distributed machines. However, with the recent advancement in computing hardware infrastructure today, even a single multicore processor machine can have more than a terabyte of memory that can support graphs with hundreds of billions of edges. Our project focuses on implementing some graph algorithms for shared-memory machines. We have implemented a shared memory version of Breadth-First Search (BFS), Bellman-Ford Algorithm (BF), and Connected Component Algorithm.

OBJECTIVE & METHODOLOGY

There are many different operations that could be performed on graphs, such as traversal, finding shortest paths, checking connectivity, and clustering. Many of these operations have been performed using algorithms developed by great computer scientists and mathematicians in the past few decades. These operations are quite expensive as they have to be performed on massive graphs. Hence decreasing the runtime of these operations for large graphs is essential. Our objective is to implement a few of the graph traversal and processing algorithms for shared-memory multicore processors. We implemented a software infrastructure as a framework and used it to implement different graph algorithms. We parallelized our code using the OpenMP library for C++. The main idea is that our frontier-based algorithms move forward parallelly, and race condition is taken care of using CAS or reduction technique.

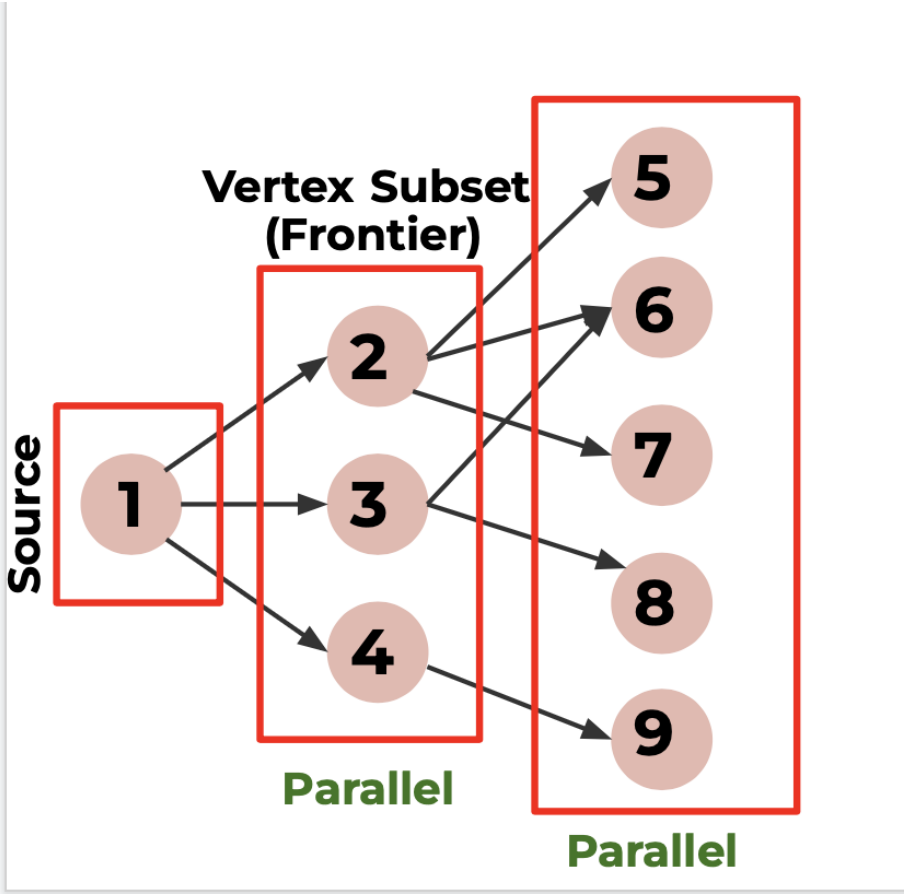

Figure 1. Parallel Implementation of frontier based algorithms

Software Framework Implementation

We studied the framework provided in the paper Shun and Blelloch (2013) for parallelization of graph algorithms. From the paper, we have implemented two fundamental building block functions - "EdgeMap" and "VertexMap" using C++17 in a parallel manner.

Algorithm 1 EdgeMap

```
1:  procedure EDGEMAP(G,U,F,C)
2:      Out = {}
3:      parfor each v ∈ U do
4:          parfor ngh ∈ N+(v) do
5:              if C(ngh) == 1 and F(v,ngh) == 1 then
6:                  Add ngh to Out
7:          end parfor
8:      end parfor
9:      Remove duplicates from Out
10:     return Out
```

EdgeMap function takes the following argument and returns a vertex subset.

- G: Graph that we are working on

- U: Vertex Subset that we are currently working on
- F: function object that applies function F to all edges with source vertex in U and target vertex satisfying C. This is having parallelized implementation.
- C: function object that takes vertex as a input and returns a boolean.This is having parallelized implementation.

Algorithm 2 VertexMap

```
1: procedure VERTEXMAP(U,F)
2:     Out = {}
3:     parfor each u ∈ U do
4:         if F(u) == 1 then
5:             Add u to Out
6:     end parfor
7:     return Out
```

VertexMap function takes a vertex subset U and function object F as argument and returns a vertex subset.

Function objects F and C are different for different algorithms, and so these two arguments control which algorithm we are implementing using this framework. We will later show how to use these two EdgeMap and VertexMap functions to write different graph algorithms. Our implementation source code is provided in the following GitHub repository: [Source Code]

Breadth First Search (BFS) Implementation

For implementing BFS, we will be using F as UPDATE(vertex,vertex) and C as COND(vertex) functions.

Algorithm 3 Breadth First Search

```
1: Parents ={-1,-1,...,-1}                          ▷ Initialized all to -1
2: procedure UPDATE(s,d)
3:     return (CAS(Parenrts[d],-1,s))

4: procedure COND(i)
5:     return (Parenrts[i],-1)

6: procedure BFS(G,r)                               ▷ r is the root
7:     Parents[r] = r
8:     Frontier = {r}                               ▷ Vertex Subset initialized to contain only r
9:     while SIZE(Frontier) ≠ 0 do
10:        Frontier = EDGEMAP(G,Frontier,UPDATE,COND)
```

Bellman Ford Implementation

For implementing Bellman Ford, we will be using F as BFUPDATE(vertex,vertex) in EdgeMap function while F as as BFRESET(vertex) in VertexMap function. Also, function C in EdgeMap will always return true for Bellman Ford.

Algorithm 4 Bellman Ford

```
1:  SP ={∞,∞,...,∞}            ▷ Initialized all to ∞ Visited = {0,0,...,0} ▷ Initialized all to 0
2:
3:  procedure BFUPDATE(s,d,edgeWeight)
4:      if (WriteMinSP[d,SP[s] + edgeWeight) then
5:          return CAS(Visited[d],0,1)
6:      else
7:          return 0

8:  procedure BFRESET(i)
9:      Visited[i] = 0
10:     return 1

11: procedure BELLMAN-FORD(G,r)                                   ▷ r is the root
12:     SP[r] = 0
13:     Frontier = {r}                          ▷ Vetex Subset initialized to contain just r
14:     round = 0
15:     while SIZE(Frontier) ≠ 0 and round < |V| do
16:         round = round + 1
17:         Frontier = EDGEMAP(G,Frontier,BFUPDATE,C_true)
18:         Frontier = VERTEXMAP(Frontier,BFRESET)
19:     if round == |V| then
20:         return "negative-weight cycle"
21:     else
22:         return SP
```

Connected Component Implementation

For implementing Connected Components, we will be using F as CCUPDATE(vertex,vertex) in EdgeMap and F as COPY(vertex) in VertexMap function. Also, similar to Bellman Ford *C* function will always return ture in this case.

Algorithm 5 Connected Components

```
1:  IDs ={0,1,...,|V|−1}                                  ▷ Initialized such that IDs[i] = i
2:  prevIDs ={0,1,...,|V|−1}                              ▷ Initialized such that prevIDs[i] = i

3:  procedure CCUPDATE(s,d)
4:      origID = IDs[d]
5:      if (WriteMinIDs[d,IDs[s]) then
6:          return origID == prevIDs[d]
7:      return 0

8:  procedure COPY(i)
9:      prevIDs[i] == IDs[i]
10:     return 1

11: procedure CC(G)
12:     Frontier = {0,1,...,|V|−1}                        ▷ Vetex Subset initialized to V
13:     while SIZE(Frontier) ≠ 0 do
14:         Frontier = VERTEXMAP(Frontier,COPY)
15:         Frontier = EDGEMAP(G,Frontier,CCUPDATE,C_true)
16:     return IDs
```

TECHNIQUES & CHALLENGES

We use OpenMP to parallelize the Framework functions VertexMap and EdgeMap. These functions operate on the frontier, which is a subset of vertices. As we saw above, we can operate on the vertices of the frontier in parallel. We used **OpenMP for** on the for loop traversing the frontier vertex subset.

Even though the working on these vertices by different threads is independent, we access and modify shared memory arrays/vectors when operating on the neighbors of the frontier vertex subset. Hence we need some way of synchronization while dealing with shared memory. We faced a challenge at two places, which required providing some synchronization.

1. **Challenge**: For each of the graph algorithms, we have a shared array, e.g., Parents array for Breadth First Search, Components ID array for Connected Components, Shortest Paths for Bellman-Ford. As different threads will access and modify them, there is always of a chance of a data race.
   Solution: We use Compare and Swap (CAS) to update these arrays and are able to guarantee synchronization in accessing shared arrays. We implemented this using the inbuilt GCC compiler function __sync_bool_compare_and_swap, which is an atomic instruction. We also have a writeMin function which is a modified version of the CAS function atomically updating the variable to the minimum of two values. For this, also we use the same atomic instruction.

2. **Challenge**: After each iteration, we output a new frontier to work on the next iteration. Each thread is appending data to the output array/vector, and hence we need to make the appending of data to the array a critical section. We use openMP critical to do this. It provides synchronization, but OpenMP critical is very slow and doesn't provide any efficiency. So using this technique, we did not see much speed up with multi-threading.

Solution: We decided to use a reduction technique here to avoid using critical altogether. We created private arrays for each thread. Each thread appends vertices to its own private array, and then we combine the arrays together. We first find the size of each private array of the threads and then allocate the exact index for each of the threads to update the values from their own private arrays. Here we avoid any kind of critical section and also notice a performance improvement.

RESULTS AND OBSERVATIONS

We tested our data on quite a few different graph inputs, starting with a small size of 100 nodes and edges to a large graph with about a Million nodes and edges. Below are the results of a [Graph] with 1 Million nodes and 15 Million edges. We ran three algorithms that we had parallelized. We ran the algorithms with a different number of threads (powers of 2) and noted their running times. The Bellman-Ford and Breadth First Search algorithms running times depend on the source node we choose. To understand how our algorithm was doing, we chose 100 random start nodes and then calculated their average running times for each of the thread sizes we used. We then found the speed up by comparing the avg parallel time to the avg sequential time over a set of random source nodes/vertices. Below are the graphs of the results we got for each of the algorithms, Bellman Ford fig2, Breadth First Search fig3, Connected Components fig4

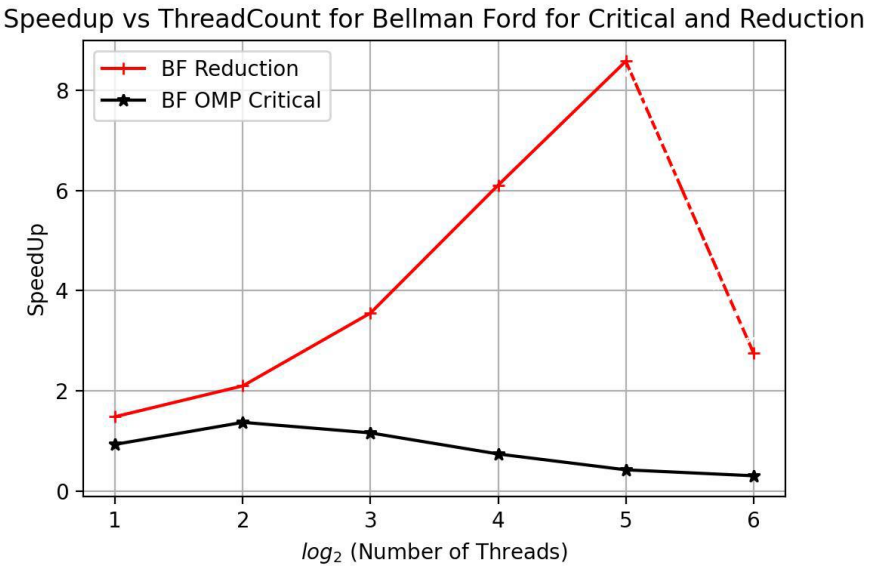

Figure 2. Bellman Ford

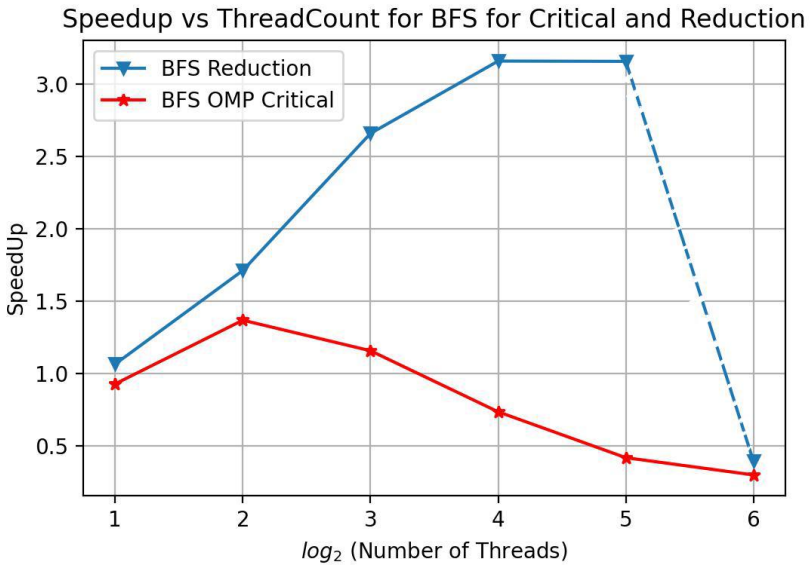

Figure 3. Breadth First Search

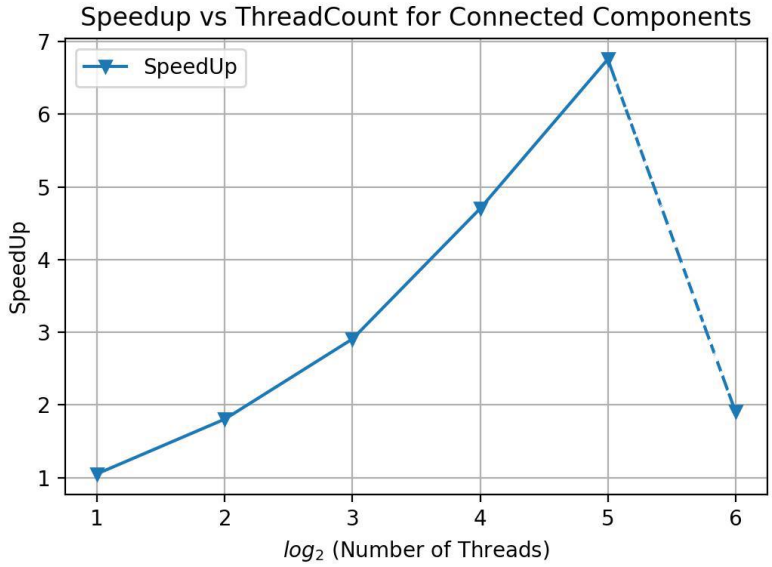

Figure 4. Connected Components

Observations

- The graph shows that with the increase in the number of threads, we see an increase in the speed up, which is a sign of efficient parallelism.
- We do, however, observe that after a threshold, we see a drop in performance which could have been due to the overhead of thread creation. In the case of this graph, we find that to be at 32 threads. Now we expect that with an increase in the number of edges and having a denser graph, we should find an optimal number of threads to increase to more than 32 and give better performance.

- For smaller graphs, there is significant overhead in parallelization, and therefore, sequential versions of algorithms should be preferred.
- We tested Bellman Ford against the implementation based on Bellman ford in CLRS and found it to be really slow for large graphs. This shows that we definitely require some form of parallelization, even for mid-sized graphs which have a size in Millions.
- The graph represents that openMP critical is inefficient, and the reduction gives much better performance.

FURTHER IMPROVEMENTS

- Code Optimization using profiling
- Try with different parallelization libraries
- Can be combined with distributed system algorithms to process huge graphs

REFERENCES

Shun, J. and Blelloch, G. E. (2013). Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 135–146, New York, NY, USA. Association for Computing Machinery.