CSC505 - Final Project Report

Andy Sauerbrei (amsauerb@ncsu.edu)
AJ Bulthuis (ajbulthu@ncsu.edu)
Tyrone Wu (tkwu@ncsu.edu)

May 1, 2023

1 Introduction

In recent years, the field of High Performance Computing (HPC) has seen significant advancements in processor technology, resulting in increased performance capabilities. To fully leverage the potential of these systems, it is essential to develop sophisticated algorithms that can efficiently utilize the underlying hardware. In this paper, we aim to address this challenge by exploring the impact of a multi-thread model on runtime performance for identifying connected components. With connected components being a fundamental topic in graph algorithms, this study aims to experiment with the application of multi-threading on Breath-first Search and Label Propagation, and identify any potential benefits and/or limitations.

2 Experimental Design

2.1 Algorithms

As mentioned earlier, our algorithms will utilize multi-threading to facilitate the concurrency model. Although these algorithms are not primarily known as parallel algorithms, the following can be modified to allow to support concurrency:

- Breath First Search (BFS)
- Label Propagation

Additionally, the following keywords in our concurrency model should first be reviewed:

- `#spawn threads`: Spawn threads.
- `#divide for-loop among threads`: Distribute subset of the list among the threads.
- `#atomic operation`: Only one thread may access the memory location of the operation.
- `#thread barrier`: Wait for all threads to reach the point.
- `#single thread operation`: Operation should only be executed once.

2.1.1 Frontier-based BFS

For identifying connected components, breath-first search is a classic algorithm used to traverse a connected graph. In a traditional BFS, vertices are processed sequentially, with unexplored neighbors being added to a queue. The order in which vertices are marked is determined by the queue, and this functions as a "checklist" of vertices that have not been explored.

The concept of neighbor exploration in BFS can be extended to support concurrency by partitioning the exploration among multiple threads. Rather than sequentially exploring each neighbor in the queue, a list is used to maintain the next "frontier/layer" of unexplored vertices.

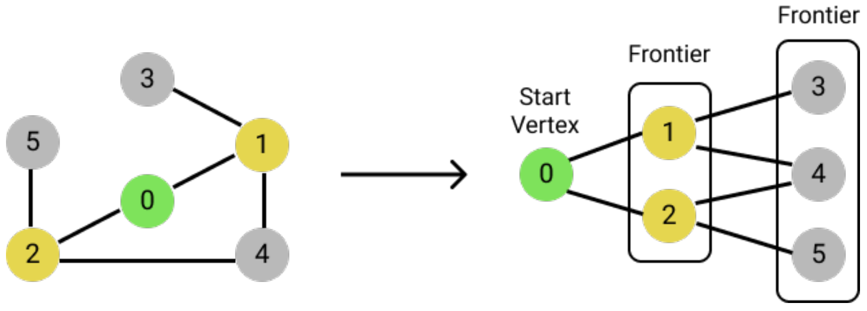

Figure 1: Visualization of frontiers

When the exploration of a “frontier” is divided among the threads, each thread will be assigned a subset of the vertices in the “frontier” list. The threads simultaneously visit the neighbors of their assign vertices, and update their own list of neighbors to explore next. Once a thread has finished exploring their assigned vertices, their list of “next unexplored vertices” will be merged to a globally visible “frontier” list. This process repeats by exploring the new “frontier” until all vertices have been explored.

One caveat of this design is that a race condition occurs when two vertices have an edge that point to the same neighbor. To avoid adding duplicate visits in the next frontier, an atomic scope (one thread access at a time) must be assigned to prevent multiple threads accessing the same unexplored vertex. By doing so, the threads can operate without interfering with each other’s exploration.

W this design, the following pseudo code for a parallel, frontier-based BFS can be outlined:

Algorithm 1 ParallelBFS

```
    Input: vertices - List[Vertex]                      ▷ Adjacency List of vertices
           startIndex - Integer                         ▷ Index of starting vertex
           levels - List[Integer]                       ▷ List of integers
 1: levels[startIdx] ← 0                                ▷ Mark start vertex as explored
 2: currentlevel ← 0                                    ▷ Track current level of frontier
 3: frontier ← { startIndex }                           ▷ Track current frontier of vertex indicies
 4: offset ← 0                                          ▷ Track the next offset for each thread to resume

 5: #spawn threads
 6: while frontier is not empty do

 7:    localNextFrontier ← {}                           ▷ Each thread maintains their own next frontier
 8:    #divide for-loop among threads
 9:    for vertexIndex in frontier do
                                                        ▷ Iterate neighbors of vertex
10:       incidentEdges ← vertices[vertexIndex].incidentEdges
11:       for destIndex in incidentEdges do

12:          if levels[destIndex] is -1 then            ▷ If neighbor has not been visited
13:             #atomic operation                       ▷ One thread at a time
14:                insert ← levels[destIndex]           ▷ Flag thread to add neighbor
15:                levels[destIndex] ← currentLevel + 1 ▷ Set neighbor as visited
                                 ▷ Flagged thread adds unexplored neighbor to next frontier
16:             if insert is -1 then
17:                insert destIndex to localNextFrontier
18:             end if
19:          end if

20:       end for
21:    end for

22:    #atomic operation                                ▷ One thread at a time
23:       localOffset ← offset                          ▷ Each thread tracks own insert position
24:       offset ← offset + size of localNextFrontier   ▷ Offset for next thread
25:    #thread barrier

26:    #single thread operation                         ▷ Only a single thread performs this operation
27:       resize frontier to offset    ▷ Size of frontier is total size of all thread’s local
    next frontier
28:    #thread barrier
```

```
                                          ▷ Threads insert their own local next frontier
    for i = 0 upto size of localNextFrontier - 1 do
        localIndex ← i + localOffset                ▷ Index of where thread should insert
        frontier[localIndex] ← localNextFrontier[i]
    end for
    #thread barrier

end while
```

Similar to a traditional BFS, the frontier-based variant maintains a list of "visited" vertices in the form of `levels`. Each element in the `levels` list indicates the "level" at which a vertex is explored. To begin with, all elements in the `levels` list are initialized to `-1` to represent all vertices as unexplored. Atomic operations are used in line 13 - 15 to prevent multiple threads from accessing and updating a given element in the `levels` list. This ensures that only one thread is flagged to add the particular vertex in the next frontier if multiple accesses to the same index occur at the same time.

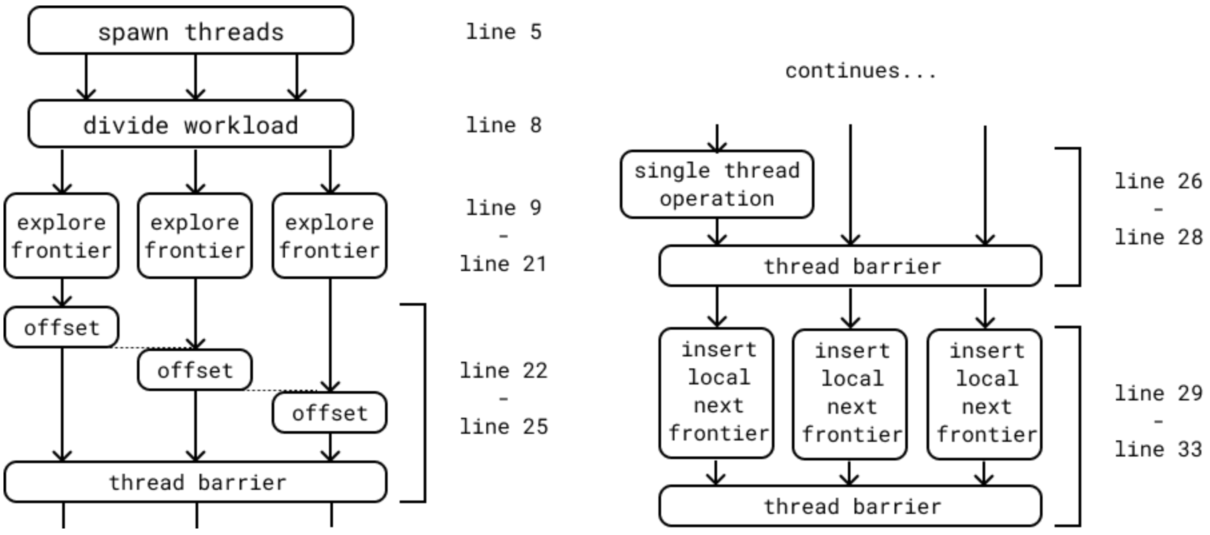

Figure 2: Multi-thread model of Frontier-Based BFS

The functionality of lines 22 - 33 involves each thread combining their own `localNextFrontier` into `frontier` for the next wave of exploration. As each thread records their `localNextFrontier`, the `localOffset` is used to determine where a thread should insert their `localNextFrontier` in `frontier`.

Since the underlying algorithm still operates in a BFS fashion, the time complexity for a single threaded, frontier-based BFS is still $O(V+E)$. With additional threads, the time complexity becomes hard to determine since the incident edges to a vertex is highly variable on the structure of the graph itself.

2.1.2 Label Propagation

General description of LP...

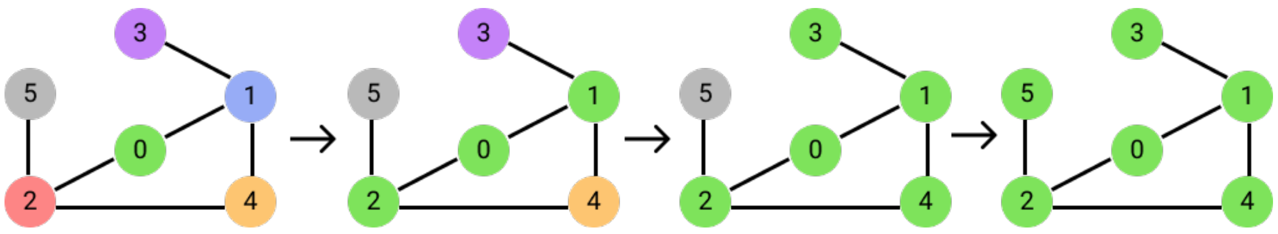

Figure 3: Visualization of Label/Color Propagation

Algorithm 2 LabelPropagation

```
Input: vertices - List[Vertex]                        ▷ Adjacency List of vertices
       N - Integer                                    ▷ Number of vertices in the graph
1: labels ← { 0, ···, N-1 }                           ▷ List filled from 0 to N-1
2: change ← false                                     ▷ Track if a change was made
3: do                                                 ▷ Propagate label of each vertex through the graph
4:    change ← false                                  ▷ Reset to false for each stage of propagation
5:    #spawn threads
6:    #divide for-loop among threads
7:    for i = 0 upto N-1 do                           ▷ Iterate vertices
8:       min ← labels[i]                              ▷ Track min value to propagate
9:       incidentEdges ← vertices[i].incidentEdges
10:      for neighborIndex in incidentEdges do        ▷ Iterate neighbors of vertex
11:         if labels[neighborIndex] is less than min then   ▷ Get min label of neighbors
12:            min ← labels[neighborIndex]
13:         end if
14:      end for

15:      if labels[i] not equals min then             ▷ Check if change is needed
16:         labels[i] ← min                           ▷ Propagate vertex label to new min
17:         change ← true                             ▷ Flag change has occured
18:      end if

19:      for neighborIndex in incidentEdges do        ▷ Iterate neighbors of vertex again
                                                      ▷ Propagate min label to neighbors
20:         if labels[neighborIndex] is greater than min then
21:            labels[i] ← min
22:            change ← true                          ▷ Flag change has occured
23:         end if
24:      end for
25:   end for
26: while change is true
```

Minor description/clarification of pseudocode

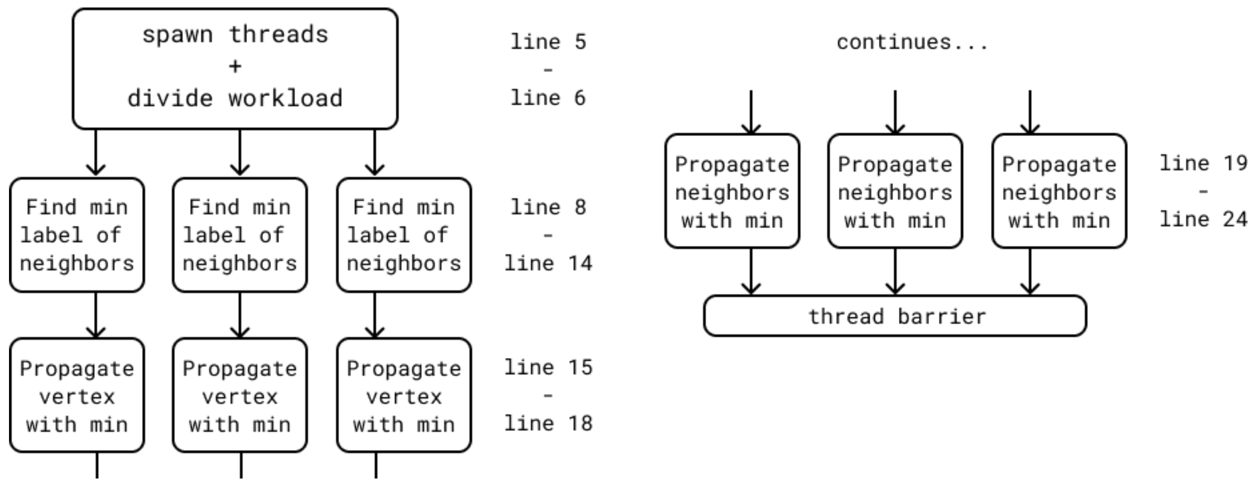
spawn threads + divide workload
line 5 - line 6
Find min label of neighbors
line 8 - line 14
Propagate vertex with min
line 15 - line 18
continues...
Propagate neighbors with min
line 19 - line 24
thread barrier

Figure 4: Multi-thread model of Label Propagation

add time complexity analysis?

2.2 Hypothesis

Hypothesis 1: Given that
Hypothesis 2:

2.3 Methodology

For our dataset, the following

3 Implementation Description

3.1 Data Structures

Our graph data structure uses an adjacency list to store the vertex itself, as well as a list of incident edges, which are

3.2 Experimental Environment

3.2.1 Compilation Specifications

Our entire implementation of the program was done in `C++` with the following compilation specification:

- `C++` Standard: `C++ 20`
- Compiler: `clang++`
  - Version: `Debian clang version 11.0.1-2`
  - Target: `x86_64-pc-linux-gnu`

3.2.2 Hardware:

The experiment was conducted inside a Linux (Debian) Docker container on a Windows machine with the image:
`mcr.microsoft.com/devcontainers/cpp:latest`:

Hardware specification (including the container):

- System Type: 64-bit OS, x64-based processor
- Operating System:
  - Linux (Debian) Container: Linux version 5.4.72-microsoft-standard-WSL2 (gcc version 8.2.0 (GCC))
  - PC: Windows 10 Education Version 21H2 (OS Build 19044.2604)
- CPU:
  - Model Name: AMD Ryzen 5 2600X Six-Core Processor
  - Base Speed (MHz): 3.60 MHz
  - Cores: 6 cores
  - Logical Processors: 12 processors
    - Processor Speed (MHz): 3.60 MHz
    - Cache Size (KB): 512 KB
- Memory:
  - Total Physical Memory (GB): 16 GB
  - Total Virtual Memory (GB): 26 GB
  - RAM Speed (MHz): 2667 MHz
  - RAM Type: DDR4

4 Experimental Results

4.1 Frontier-based BFS

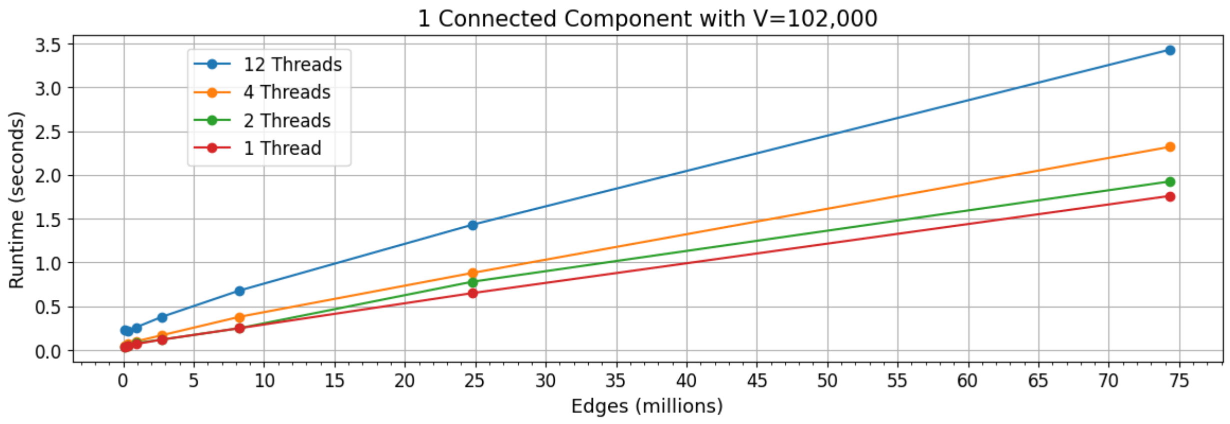

Figure 5: Frontier BFS Results on 1 Connected Component

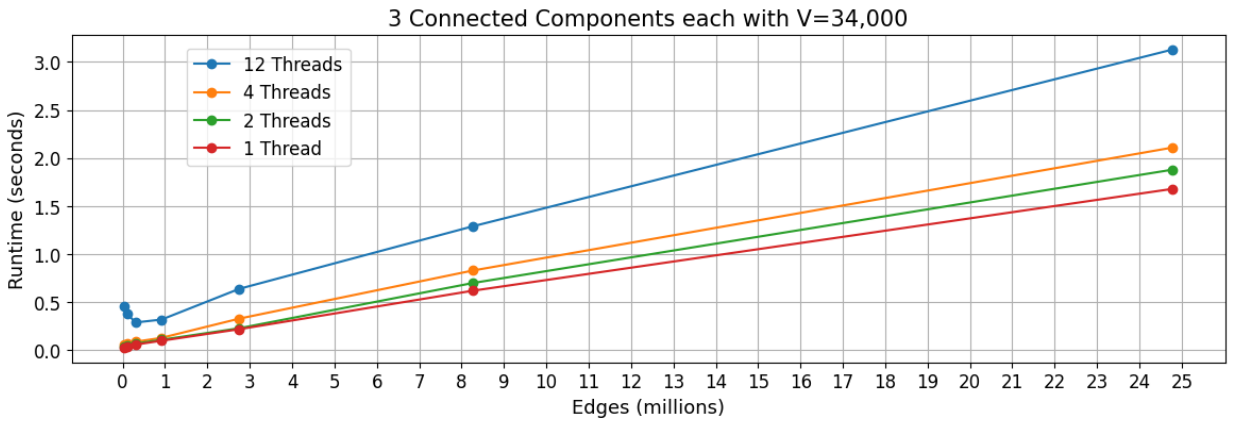

Figure 6: Frontier BFS Results on 3 Connected Components

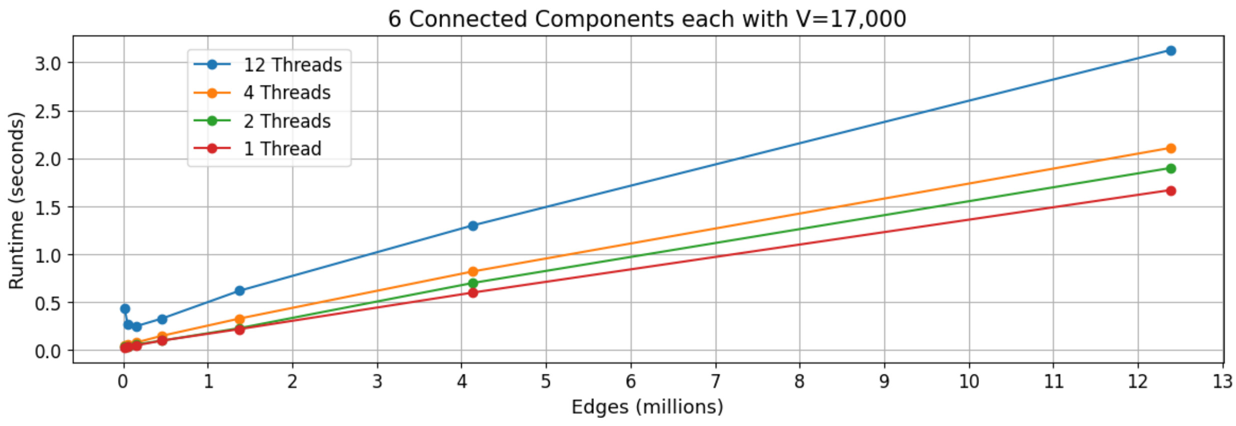

Figure 7: Frontier BFS Results on 6 Connected Components

4.2 Label Propagation

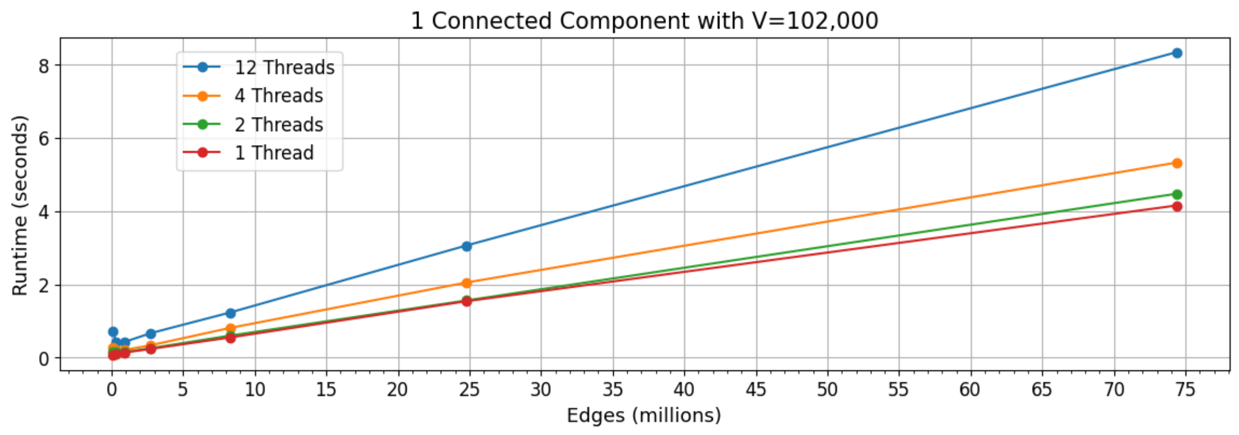

Figure 8: Label Propagation on 1 Connected Component

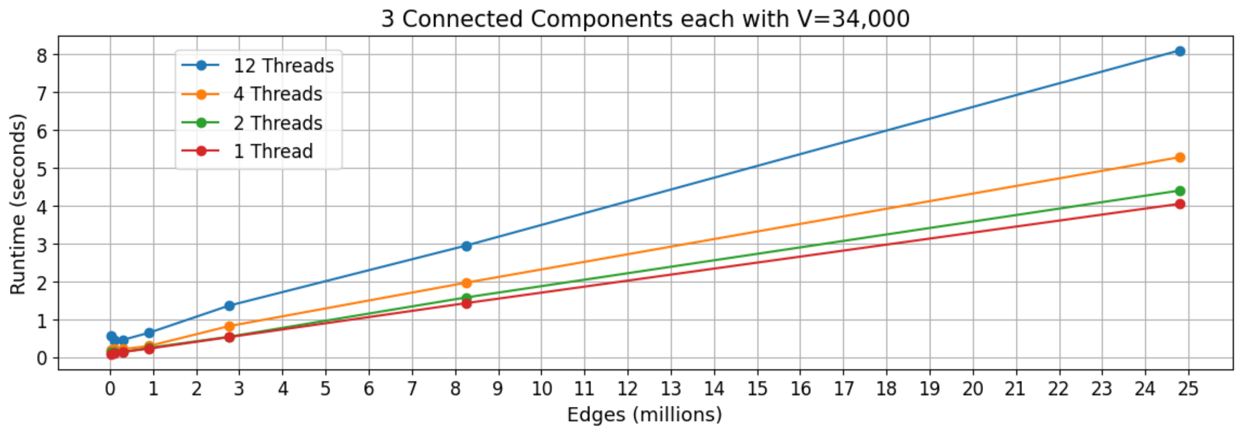

Figure 9: Label Propagation on 3 Connected Components

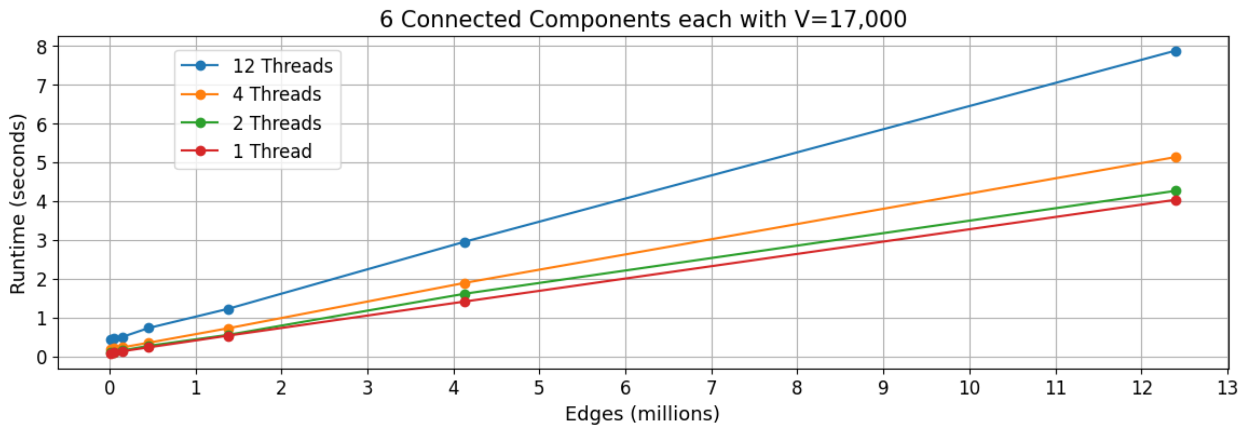

Figure 10: Label Propagation on 6 Connected Components

4.3 Comparison

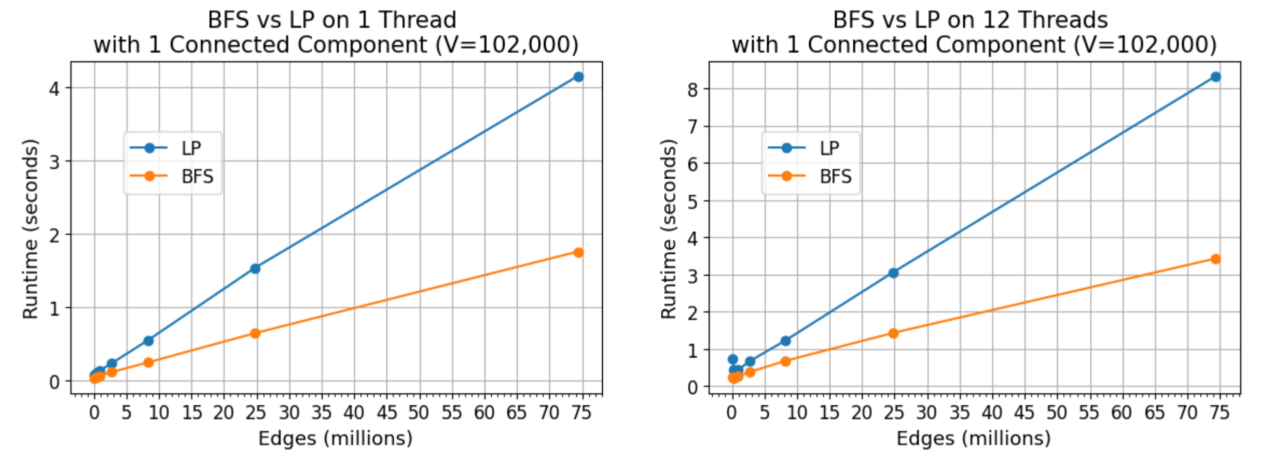

Figure 11: BFS vs LP on 1 Connected Component

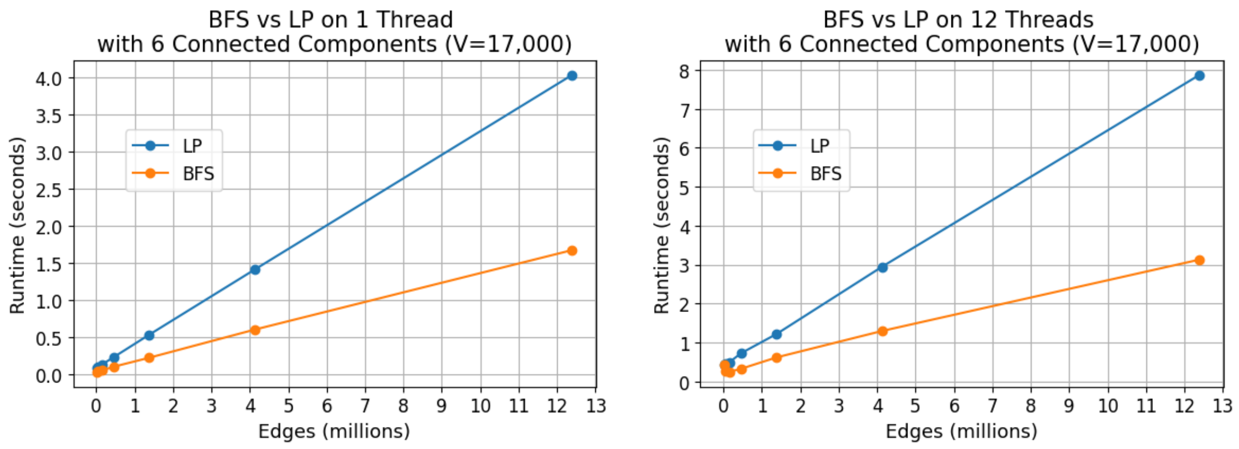

Figure 12: BFS vs LP on 6 Connected Components

5 Conclusions and Future Work

5.1 Future Work