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.

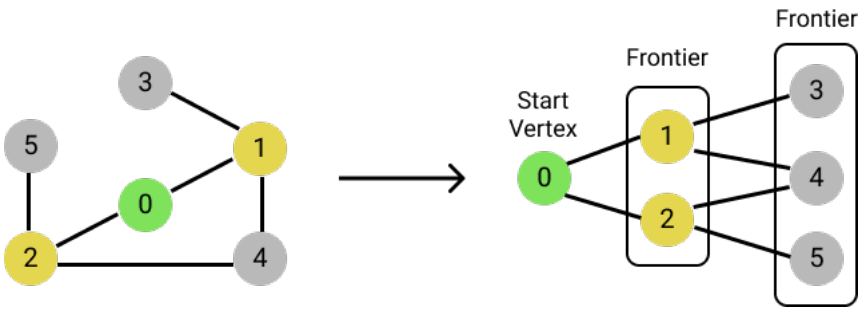

Figure 1: Visualization of frontiers

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
29:     for i = 0 upto size of localNextFrontier - 1 do
30:         localIndex ← i + localOffset                ▷ Index of where thread should insert
31:         frontier[localIndex] ← localNextFrontier[i]
32:     end for
33:     #thread barrier

34: 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.

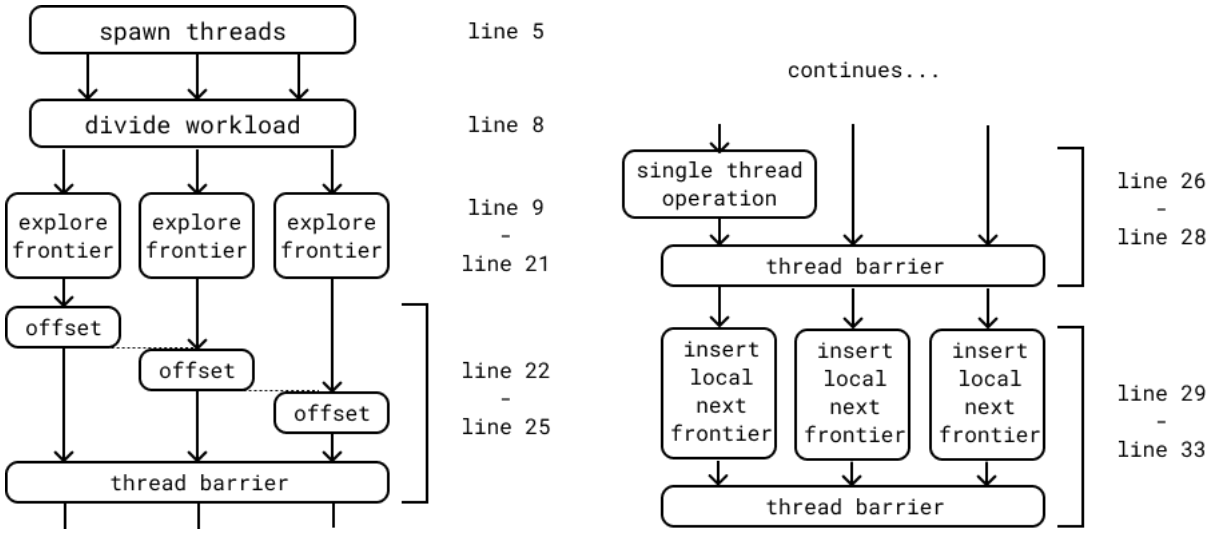

Figure 2: Multi-thread model of Frontier-Based BFS

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

Traditionally, Label Propagation is used for identified communities in a graph and the algorithm works in around four steps. Firstly, it will give all vertices in the graph a unique label. Secondly, it will randomize the order of vertices. Thirdly, it will look at each vertex and set its label equal to the majority of label of that vertex's neighbors. Fourthly, it will repeat this process for every vertex until a certain minimum number of changes are met. This will end up with the graph having a number of groups of closely related vertices identified by a particular label.

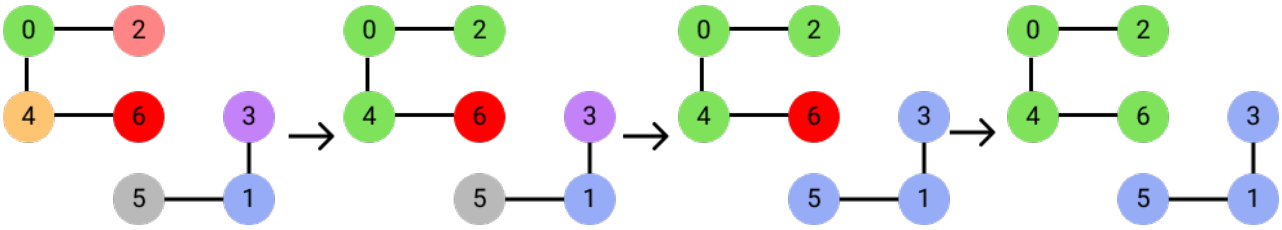

Figure 3: Visualization of Label/Color Propagation

Our variation of Label Propagation is of course being used to identify connected components, so we made a few changes. Firstly, instead of setting each vertex to match the number of a majority of its neighbors, our version will choose the lowest value label of its neighbors (See Figure 3). Secondly, instead of just changing the label of the vertex we are analyzing the neighbors of we change the labels of the neighbors as well. Thirdly, we will run the algorithm until no changes are made. Finally, we have an additional loop after the main loop to cont the unique number of labels so that we know how many connected components we have.

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
```

It should be noted that the additional loop mentioned previously to count the number of unique labels was not placed in the above pseudo code. Additionally, figure 4 bellow will better describe how to for loop is dividing the work amongst the threads denoted in lines 5-6.

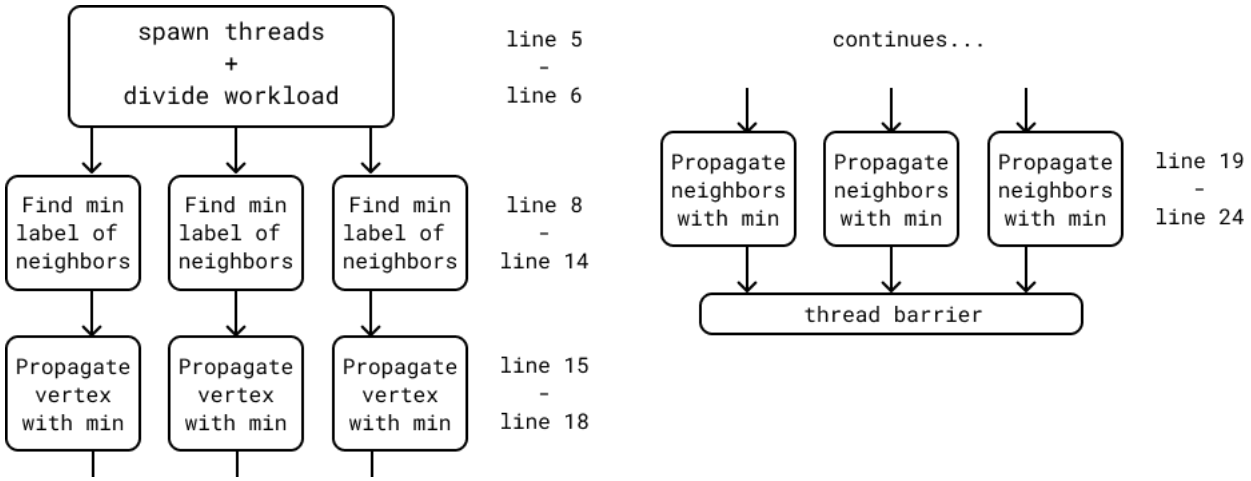

Figure 4: Multi-thread model of Label Propagation

2.2 Hypothesis

Hypothesis 1: Given that there is a upfront cost for allocating threads, we anticipate that a smaller number of threads will out perform larger quantities of threads when the graphs are smaller and sparser.
Hypothesis 2: Considering that Label Propagation consists of simpler operations, in comparison the BFS, at the cost of a larger number of iterations, we anticipate that Label Propagation will outperform BFS on smaller, sparser graphs.

2.3 Methodology

For our dataset, we use the following parameters for our input:

- 1 Connected Component with 102,000 vertices
- 3 Connected Components each with 34,000 vertices
- 6 Connected Components each with 17,000 vertices

The edges of each component $(102,000; 34,000; 17,000)$ start from the minimum possible edges $(V-1)$ and increment in an exponential fashion by s steps with $V * 3^s$.

3 Implementation Description

3.1 Experimental Environment

3.1.1 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`

3.1.2 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

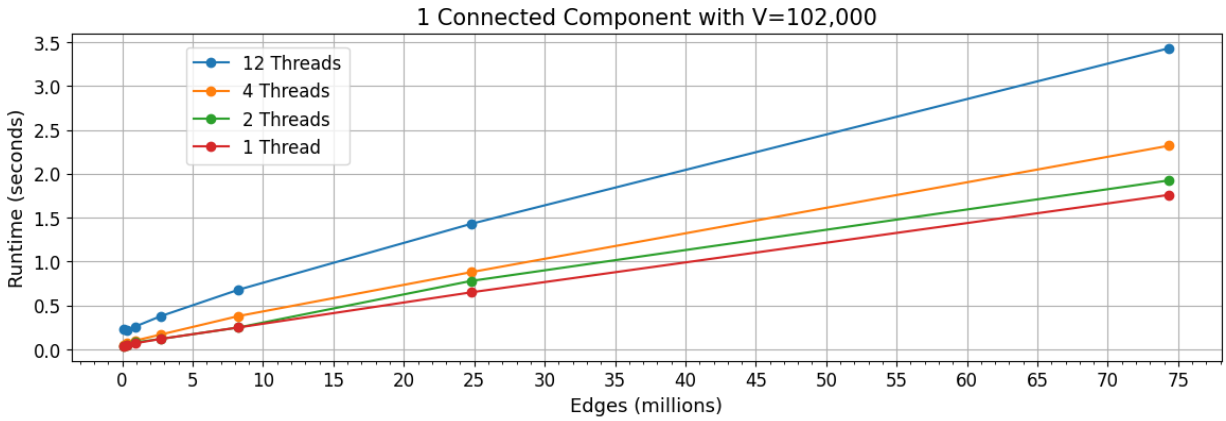

Figure 5: Frontier BFS Results on 1 Connected Component

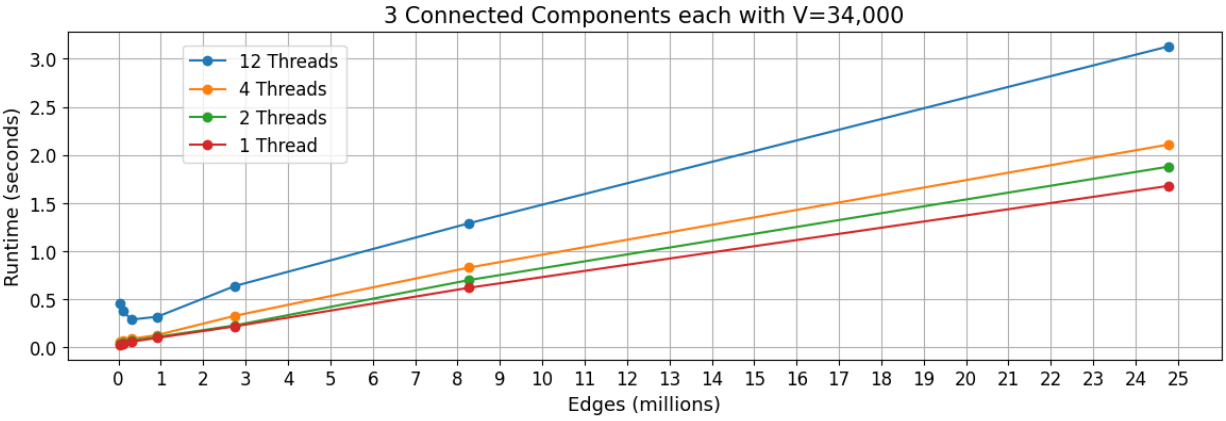

Figure 6: Frontier BFS Results on 3 Connected Components

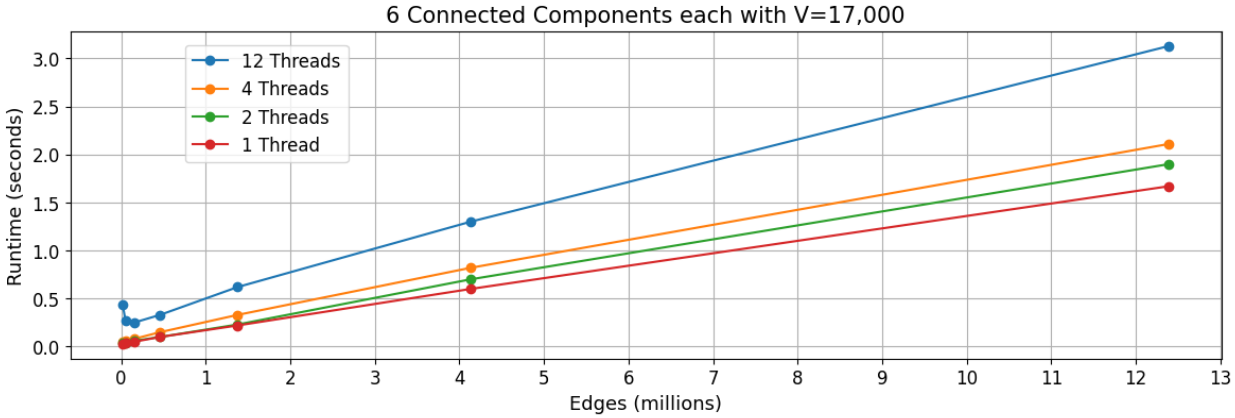

Figure 7: Frontier BFS Results on 6 Connected Components

Contrary to what we assumed, more threads caused a significant drop in both sparse and dense graphs regardless of the number of connected components present in the graph. We anticipated this to be the case for very small and very sparse graphs as each thread may not have very much work to do in these instances as the frontier may be rather small. If each thread didn't have too much work to do, the upfront cost of actually allocating and deleting threads may be greater than any performance gains. With this line of thinking, we assumed large and dense graphs would provide plenty of work for threads, and this would allow for performance improvements greater than the upfront costs of creating the threads. However, this was not the case.

In general, we have two main thoughts on why we didn't see the anticipated performance increases on larger, denser graphs. The first case would be that we may not have configured OpenMP properly with our hardware, but it seemed to be allocating thread properly in task-manager. The second case would be that the atomic operations and single thread operations for merging each of the thread's frontiers together may be too costly to justify any performance increase we are getting out of allocating additional threads; this operation only gets more complex as the number of threads increases.

4.2 Label Propagation

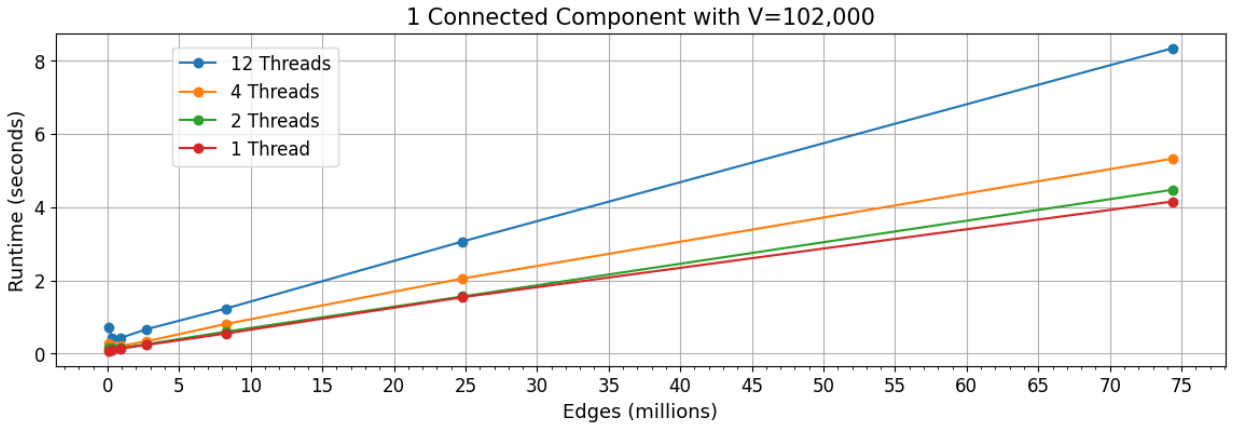

Figure 8: Label Propagation on 1 Connected Component

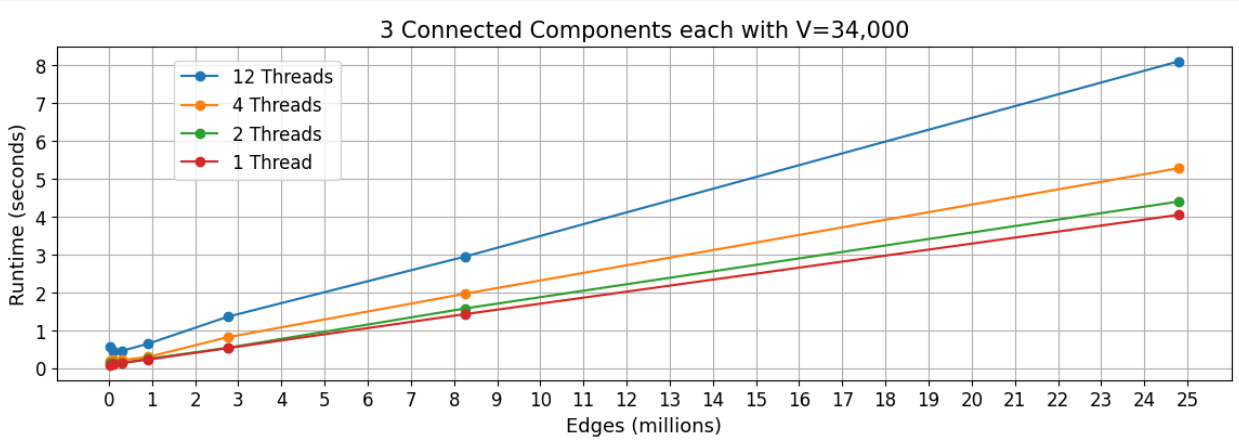

Figure 9: Label Propagation on 3 Connected Components

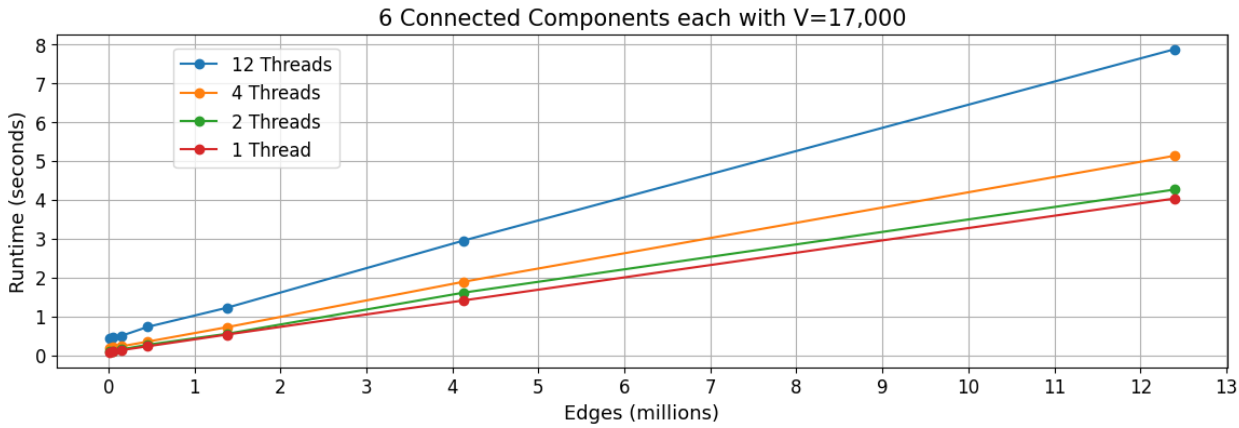

Figure 10: Label Propagation on 6 Connected Components

Like our results regarding BFS, the results for label propagation ran contrary to our assumptions in the same manner; an increased number of threads had a negative impact on performance for both smaller, sparse graphs as well as with larger, dense graphs. However, our run times for the lowest two thread quantities, one and two, seemed to be somewhat comparable until we moved on to denser graphs; this was most prominent in the smallest number of connected components (Figure 8).

We believe that these results can be explained by at least one of two reasons. The first reason is that OpenMP might not have been configured properly for our hardware, although threads seemed to be allocated properly when looking in task manager. The second reason is that there may have been some false sharing present when each thread is reading from and updating the array holding the label for each vertex, but it was implemented in this way to hopefully outweigh the potential cost of giving each thread its own list of labels that would need to be merged and redistributed to each thread—a potential issue we identified with our BFS implementation's performance.

4.3 Comparison

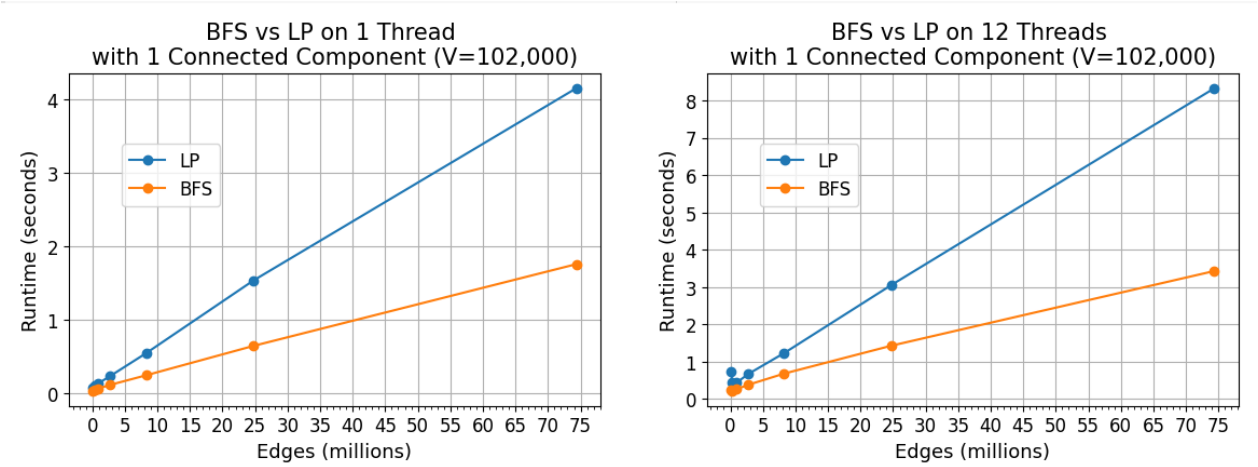

Figure 11: BFS vs LP on 1 Connected Component

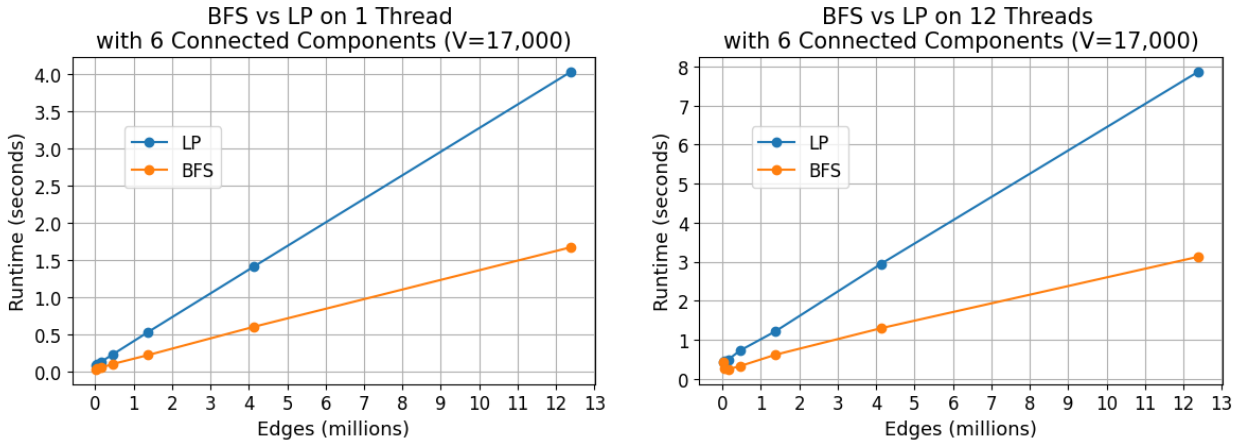

Figure 12: BFS vs LP on 6 Connected Components

Across the board, label propagation was outperformed by BFS, and the gap in performance grew as the graphs became increasingly dense. We assumed that label propagation may perform better with the smaller, sparse graphs than BFS, as its algorithm is simpler, requiring less complex operations, but needing to iterate through every vertex in the graph for every iteration. Granted, the gap in performance was smallest with smaller, sparser graphs.

We believe that label propagation may have performed more comparably, or at least have been compared more fairly, to our BFS implementation if we had implemented each thread in label propagation to have each its own list of labels that would be merged after every iteration to avoid false sharing, as we did with the BFS implementation. Additionally, label propagation likely had this worse performance in the instance of identifying connected components as we need to iterate through the list of vertices an additional time to identify unique labels to count the total number of components, something which BFS didn't need to worry about.

5 Conclusions and Future Work

With this project we have learned that there are more nuances associated with multi-threaded algorithms then previously anticipated as we more or less though we could insert a threaded portion in the algorithms at the point in which the greatest number of operation needed to be preformed (exploring each node in the frontier for BFS and iterating through all of the vertices for LB). Most of our issues seem to be oriented around both false sharing as well as the cost of merging lists that may not exist in a sequential algorithm.

In regards to the viability of detecting connected components using both BFS and Label Propagation using multi-threading, we believe that Label Propagation may have more promise or room for optimization. Unlike BFS, the entire process of Label Propagation can be multi-threaded, as BFS relies on fully exploring everything off of a single vertex before looking for a new vertex from the overall list to explore whilst Label Propagation will always iterate through the entire set of vertices. However, our initial experiments did indicate that BFS did outperform, Label Propagation.

5.1 Future Work

In terms of future experiments or refinements upon this experiment we believe the following are the most pertinent:

- Implement Label Propagation to give each thread its own list of labels.
- Implement Label Propagation with spacing in the array to try an counter false sharing.
- Experiment with OpenMS to better understand how it is operating with our testing hardware.
- Implement BFS with a single shared frontier list with spacing in order to fight against false sharing.
- See how Label Propagation performs if we have it just change the starting vertex to the new label as oppossed to all it neighbors to the new label, since some threads might end up doing the same work as other threads with the current implementation.