Converting A Graph Represented As An Adjacency List Into An Adjacency Matrix Using Multithreading

Maurely Acosta[1], Yair Camacho[2], Jorge Costa[3], and Malachi Williams[4]
School of Computing and Information Sciences, Florida International University
{macos101, ycama006, jcost035, mwill279}@fiu.edu

Abstract — An Adjacency matrix a n x n matrix used to represent a finite graph with n vertices. Each element of the matrix indicates whether pairs of vertices are adjacent. This is denoted by having a 1 if the vertices are adjacent or a 0 if the vertices are not adjacent. Our approach with multithreads was to compare the time complexity of three approaches: the linear approach, the multithreaded linear approach, and the brute force approach. Our findings showed that, although the multithreaded linear approach was faster than the multithreaded brute force approach, multithreading proved to be slower than non multithreaded linear approach in both cases. Thus, the sequential linear approach without multithreading was the fastest approach.

Keywords — *adjacency matrices; graph; multithreading*

I. INTRODUCTION

In computer science graph theory is the study of graphs. A graph is a collection of vertices and edges that connect pairs of vertices [1].
Graph theory is used in many research areas of computer science such as networking, data mining, and image capturing. For the purposes of this paper, we will be focusing on comparing the time complexities of three different approaches. Each approach will convert a graph represented as an adjacency list into an adjacency matrix.

An adjacency matrix is a matrix with rows and columns labeled by graph vertices, with a 1 or 0 in position ($v_{i,}$ v_j) according to whether $v_{i \text{ and }}$ v_j are adjacent or not [2].

Since each approach must iterate through all the edges each vertex is adjacent to, it is clear to see how as the size of the graph increases, so does the time complexity of each approach.
Fortunately, multithreading can improve the time complexity as much as logarithmic time.
For two of the three approaches, we will be using multithreading in, the programming language, C. One of the approaches will be

done in linear time without using multithreading as our control case.

Overall, as one of the first research works that employs converting an adjacency list graph to an adjacency matrix using multithreading in C, we show an original multithreaded implementation.

II. RELATED WORK

We have not come across any other work related to computing adjacency matrices from adjacency list graphs through multithreading, but uses of the adjacency matrix that are most noticeable are in fields such as data mining where graphs are used to summarize relationships between different data sets.

III. PROCESS

We attempted to solve this problem using three different approaches, a **brute force** approach, a **sequential linear** approach and a **multithreaded linear** approach. The goal of each approach is the same. We always start with a graph represented by an adjacency list and a matrix filled with zeros and conclude with an adjacency matrix.

A. Brute Force Approach

Our first algorithm employs a brute force approach, examining every possible adjacency near simultaneously using multithreading. The foundational method of our brute force approach is the check method. The check method takes two int values as parameters, each of which represents a vertex in the graph. These values are labeled ‘row’ and ‘column’ because they determine which spot in the adjacency matrix is being visited. If the two vertices are adjacent (determined by checking the adjacency list), the position indicated by ‘row’ and ‘column’ in the matrix is marked as 1, if not it is left at 0. Due to the way arguments must be passed when calling threads, the ‘row’ and ‘column’ int values are passed inside of a container structure. The code of the check method is as follows:

```
void * check(void *args)
{
    struct arg_struct *actual_args = args;
    // Get the vertices the current vertex is adjacent to.
    struct node* temp = graph->adjLists[actual_args->row];

    while(temp != NULL)// While there are adjacent vertices for the current vertex.
    {
        if(temp->vertex == actual_args->column)
        {
            *((matrix + actual_args->row*size) + actual_args->column) = 1;
        }
        temp = temp->next;
    }
    return NULL;
}
```

Our openThreads() method initializes our matrix and uses multithreading to open many instances of the check method. Using two nested for loops we iterate through the rows and columns of our matrix, calling a check method for each position. We employ multithreading by initiating a new thread for each instance of the check method that is called, like so:

The openThreads method concludes by waiting for each opened thread to finish before moving on.
Finally, the calculateTimeToGenerateMatrix method calls the openThreads method, times it, and prints the final adjacency matrix. The openThreads method is timed by measuring clock ticks, using the clock() method.

B. Linear Approaches

The linear approach functions by checking each the adjacencies of each vertex in the order they appear in the adjacency list. This results in the adjacency matrix being filled row by row. This is done by the createMatrix method. The createMatrix method is passed the first node in the list of adjacencies to a particular vertex. Then it cycles through the list, marking the positions in the matrix where there is an adjacency with a 1. When the method concludes it has filled in one row of the adjacency matrix. In other words, it has recorded all the adjacencies of one vertex. The loop that populates the row looks
like this:

```
vertices the current vertex is adjacent to.
* row_in_matrix = graph->adjLists[actual_args->adjListNode];


in_matrix)


tex = row_in_matrix->vertex;

" %d ->", vertex);

ix + actual_args->adjListNode*vertices) + vertex) = 1;


matrix = row_in_matrix->next;
```

Note that, like the brute force approach, we pass our arguments through a container

```
// Let us create n (where n is the number of vertices) threads.
int i;
for (i = 0; i < vertices; i++){

  struct arg_struct *args = malloc(sizeof *args); //struct for passing matrix position
  args->adjListNode = i;

    error = pthread_create(&thread_id[i], NULL, &createMatrix, args);

    if (error != 0){
        printf("\nThread can't be created :[%s]", strerror(error));
    }

}
```

structure.
The createMatrix method is utilized in two different ways by our two different linear approaches. In the **sequential linear approach**, the createMatrix method is called for each row of the adjacency matrix one at a time. When one process terminates, the other begins. The **multithreaded linear approach** also calls the createMatrix method for each row of the adjacency matrix, but uses multithreading to call them simultaneously. The loop that calls the createMatrix method for the multithreaded approach is as follows:

Like the brute force approach, the linear approach uses a calculateTimeToGenerateMatrix method which times the execution of the code that generates the matrix using the clock() method.

IV. RESULTS

Table 1. Time Taken To Generate The Matrix For Each Approach

	Linear Sequential	Linear Multithreaded	Brute Force
20 vertices, each with degree vertex of 4	0.148 ms	1.69 ms	14.68 ms
4 vertices, each with degree vertex of 20	0.098 ms	0.58 ms	0.57 ms
20 vertices, each with degree vertex of 20	0.43 ms	2.82 ms	15.60 ms
4 vertices, each with degree vertex o 4	0.025 ms	0.25 ms	0.44 ms

In our project, we used 4 different graphs represented as an adjacency list data structure into an adjacency matrix to compare each approach's execution time. Our prediction was that the linear multithreading approach was going to have the fastest execution time for each graph compared to the other two. However, the overall results showed that the linear sequential approach had the fastest execution time for each graph. From our results we realized that perhaps having multiple threads running was less efficient than having a single thread completing a task. To figure out why multithreading approaches were slower in execution time than a sequential approach is very complex to analyze, but our hypothesis is that having multiple threads running can be very inefficient, because of context switching. Another reason is when all threads need to access and share the same memory space, thereby some threads running into each other making all the other threads have to wait.

The analysis we found in our results was that the Brute Force Multithreading approach was very slow compared to the other two approaches when the graph contained a lot of nodes, and the difference in execution times were very far apart compared to Brute Force. That's because it meant creating a lot more threads when the graph contained a lot more nodes, but when the graph had much fewer nodes, the difference between execution times for Brute Force and the other two approaches were not very far apart anymore, since it didn't require Brute Force to create and have a large amount of threads running.

When comparing one approach to itself in each graph, for example, Linear Sequential, the algorithm was a lot slower when the graph contained a lot of nodes, and was quicker when the graphs contained a lot fewer nodes. The same can be said for the other two approaches when comparing the execution times to themselves respectively.

Our last analysis from our results was analyzing the effect of incident edges per vertex in a graph. The effect of incident edges per vertex in a graph only matters when using graphs that have the same number of nodes; otherwise, the results will not correlate effectively. For example, when comparing two graphs both containing 4 vertices, and one graph has 4 incident edges for each vertex, and the other has 20 incident edges for each vertex, the graph with the fewer incident edges was faster to run that the graph with vertices that contained more incident edges. The reason for that is self explanatory, the execution time is faster, because there are a lot less incident edges to traverse per head node in an adjacency list.

V. CONCLUSIONS

Our conclusion, is that having more threads running compared to having less threads running is slower when performing a task. In our project that meant, the more nodes a graph contained the more threads that had to be created and running to perform a task for both Linear Multithreading and Brute Force Multithreading, making them the two slowest algorithms when converting an adjacency list to an adjacency matrix. Therefore, signifying that multithreading will not always necessarily make a task quicker to perform, sometimes it's more efficient if a single thread is running in sequential order. The reasons why multithreading may perform slower requires more research and thorough analysis which was beyond the reach of this project, but our hypothesis was that the more threads meant an increase in context switching or the wait time to access the same memory space made multithreading slow down.

VI. REFERENCES

[1] N. Biggs, E. Lloyd, and R. Wilson, "Graph Theory,1736-1936, Oxford University Press, 1986.

[2] Weisstein, Eric W. "Adjacency Matrix." From *MathWorld*--A Wolfram Web Resource.http://mathworld.wolfram.com/AdjacencyMatrix.html