Improving Dijkstra's Algorithm using Parallelism

Lazaro Fernandez, Damian Niebler, Jessica Silva
Florida International University
11200 SW 8th St,
Miami, Florida 33199 USA

ABSTRACT

Throughout this paper, the focus will be on improving the efficiency of Dijkstra's algorithm to find the shortest path from point a to point b using parallelism. While the algorithm only works for non-negative lengths, in order to improve Dijkstra's algorithm, an adjacency matrix can be created which will contain the number of paths from each node and its weight; this will be pre-calculated by using an adjacency vertex of Y. Using parallelism, if the sum of x is less than the distance of y, the distance of Y will be updated by subtracting x from y. This simulation will assign the distance value to all vertices in an input graph, and through parallelism, check the neighboring nodes while updating the tables as the pointer moves from its current position to the next node. These instructions are then repeated until the final node is reached.

I. INTRODUCTION

Dijkstra's algorithm is one of various algorithms used to compute the single source shortest path (SSSP) for graphs that contain non-negative edges. The algorithm places each visited node into a table X of nodes, which have had their shortest path calculated. An unvisited node with the shortest path from X is selected, placed into X, and then the neighbors are updated with its new length. This all occurs, while a set containing the unvisited nodes are treated as a priority queue. There are numerous SSSP algorithms that use parallelism to implement Dijkstra's algorithm and run it within a reasonable amount of time on machines such as PRAM. In this paper, the algorithm will be improved by applying parallelism twice through CUDA based code to the inner loop. Parallelism will be applied once when checking the neighbor nodes and another when the table is updated.

There are three separate algorithms which were analyzed that are commonly used to calculate the SSSP, to find which of the three is the most efficient. Floyd-Warshall's algorithm has a time complexity of $O(N^3)$, it also allows the use of non-negative edges which is ideal when searching for the SSSP. The disadvantage of this however, is that it's slower compared to the other algorithms. As a result, applying parallelism to it does not have an effect on its time complexity when compared to the other algorithms. Bellman-Ford's time complexity is O(n*m) and is easier for parallelism to be applied. The drawback however, is that it has an asymptotical time complexity, which makes it less efficient. Lastly, Dijkstra's time complexity is based off of its implementation, therefore it could have a time complexity of $O(n^2)$, $O(n^2+m)$, O(m log n), or O(m + n log n). While it cannot handle non-negative numbers, it has the quickest time complexity and applying parallelism is easier, compared to the other algorithms.

In terms of code, the original algorithm is implemented using 2 level nested loops, an inner loop and an outer loop. The inner loop updates the length from X to all its neighbors, while the outer loop

goes through each node in the graph, and selects the node with the shortest path in respect to the first node in the graph. The focus of this paper will demonstrate the effect of applying parallelism to the inner loop of the algorithm, making it faster and more efficient.

The rest of this paper is organized as follows. Section 2 contains information on related works. Section 3 describes how Dijkstra's algorithm was parallelized through GPU. Section 4 discusses the results of the solution applied. And section 5, is the conclusion and it describes some possible future works.

II. Related Work

IS-IS (Intermediate System-to-Intermediate System Protocol) is a link state protocol feature that relates to the distance vector protocols that uses information based hearsay to create forwarding tables. The link state protocol enables quick convergence, large scalability, flexible protocol, and it has also been extended to include leading edge features for instance Multiprotocol Label Switching (MPLS) Traffic. Information in IS-IS is moved through link state protocol data units, and each IS or router discloses information relevant to itself and its links.

The basis for the route calculation in IS-IS is Dijkstra's algorithm, which calculates the shortest paths from a single source vertices to all the other vertices in a directed, weighted graph. The Cisco IOS implementation contained weights assigned to each branch of the tree in a configurable metric with spans of 2_{24} per individual link with 2_{32} from root to leaf. The information from the process is used to populate the forwarding table on the router.

Several other applications of Dijkstra's algorithm are relevant, even something as simple as browsing the internet, downloading content, or reading the news online. All of these daily activities have something in common, data transmission. While browsing the internet, data from servers over the world are transmitted to a user's IP address, which allow them to receive the data. This process uses a path, through a sequence of servers which allows data to show up on the user's device from the hosting server. The path that is used, is kept as short as possible to avoid delays, and is thus using in many cases Dijkstra's algorithm.

III. Parallelizing Dijkstra's through GPU

Two intuitive options for applying parallelism to Dijkstra's algorithm are available, to utilize either the inner loop or the outer loop of the algorithm. The solution that shall be described is when parallelism is applied to the inner loop. The sections of code below display Dijkstra's algorithm in CUDA code with and without parallelism. As shown in Algorithm 2, the updateDistance method was parallelized. This causes it to check the neighbors using multiple threads at the same time and updating the current path. This causes the algorithm's execution time to decrease as shown in Figure 2. Changing the for loop in lines 4-5 from Algorithm 1 to parallelizing with multiple threads in lines 11-15 from Algorithm 2.

Algorithm 1: Dijkstra's algorithm

```
int minDistance(int shortPath[], bool updater[]) {
   // Initialize min value
   int min = INT_MAX, min_index;

   for (int x = 0; x < V; x++)
     if (updater[x] == false && shortPath[x] <= min)
         min = shortPath[x], min_index = x;
   return min_index;
}//end of method
```

```
__global__ void updateDistance(bool* updater, long* matrix, int* shortPath, int u){

int i = blockIdx.x * blockDim.x + threadIdx.x;

// Update dist value of the adjacent vertices of the picked vertex.

        // Update dist[v] only if is not in sptSet, there is an edge from
        // u to v, and total weight of path from src to  v through u is
        // smaller than current value of dist[v]

        if(i<V){
  if ((!updater[i] && matrix[u*V+i] && shortPath[u] != INT_MAX &&
                    shortPath[u]+matrix[u*V+i] < shortPath[i]))
       shortPath[i] = shortPath[u] + matrix[u*V+i];

    }//end of if(big)
}//end of method



long* generateAdjMatrix(int count){

long* randoNumber = (long*)malloc(count*count*sizeof(long));

srand(time(NULL));

for(int i=0;i<count;i++){
        for(int j=0;j<count;j++){
                if(i !=j){
                        long randomResult = rand()%2;
                        randoNumber[(i*count)+j] = randomRes
                        randoNumber[(j*count)+i]= randomResu
                }
        }
}
        return randoNumber;
}//end of method

//count is the size of the matrix
```

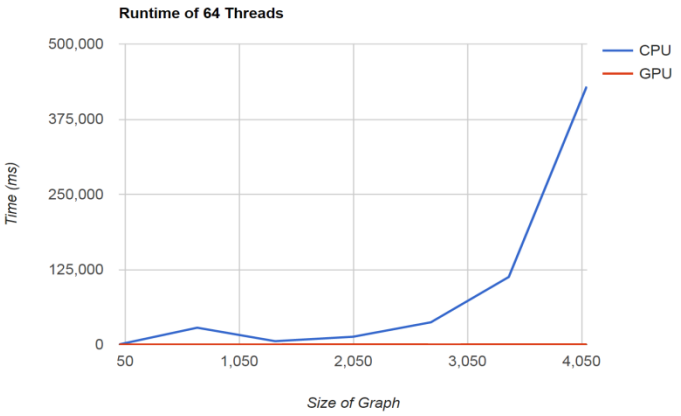
Algorithm 2: (above) Parallel implementation of Dijkstra's algorithm

IV. Results

By applying parallelism to Dijkstra's there are a number of changes that have been made when comparing the solution to the original algorithm. The time complexity is dependent on the number of vertices, if there are a small number of neighbors the code will only take a fraction of the total time complexity to calculate the shortest path. As shown in Algorithm 2, the use of multiple threads causes the algorithm's execution time to decrease as shown in Figure 2 when compared to the original algorithm in Algorithm 1.

In the original algorithm, while size of graph increases, the runtime of results takes far longer compared to the code running on the GPU. Not much change is even visible when running graphs as large as 4,000 nodes. While Dijkstra's can take up to 40 seconds to compute results, a more efficient algorithm can compute the same results to as low as .7ms. This proves that it's efficient to apply parallelism to Dijkstra's algorithm when comparing the two algorithm's execution time; therefore it was possible to improve this algorithm using parallelism.

(Figure 1)



(Figure 2)

	64 Threads				1024 Size of Graph	
Size of Graph	**Time CPU**	**Time GPU**		**# of Threads**	**Time CPU**	**Time GPU**
64	300 ms	79 ms		64	32278 ms	84 ms
128	2800 ms	72 ms		128	36979 ms	90 ms
256	5586 ms	75 ms		256	37000 ms	86 ms
512	12942 ms	238 ms		512	36604 ms	90 ms
1024	37018 ms	84 ms		1024	31641 ms	93 ms
2048	112395 ms	99 ms		2048	34159 ms	74 ms
4096	429148 ms	104 ms		4096	38878 ms	77 ms

V. Conclusion and Future Works

This algorithm could be used in truck companies that transport food from city A to city B. To maintain the food fresh and on-time, the truck must have an optimal path with the shortest time traveled. Based on this improvement on Dijkstra algorithm, we can easily obtain the best results and get the efficient route on every trip.

In this paper we have shown how parallelism was applied to Dijkstra's algorithm and how it is efficient when compared to the original algorithm. This has been shown by timing how quickly it takes to execute the algorithm with large amounts of nodes as shown in Figure 2 with code written in CUDA as shown in Algorithm 2. With all of this information the result was that Dijkstra's algorithm can be improved by using multi threads.

References:

1. Anastopoulos, Nikos, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. "Early Experiences on Accelerating Dijkstra's Algorithm Using Transactional Memory." *Academia.edu*. National Technical University of Athens School of Electrical and Computer Engineering Computing Systems Laboratory, n.d. Web. 19 Apr. 2016.
2. Bertsekas, Dimitri P., and Robert G. Gallager. *Data Networks*. 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1992. Print.
3. "Intermediate System-to-Intermediate System Protocol [IP Routing]."*Cisco*. N.p., n.d. Web. 21 Apr. 2016.
4. Mohammad, Asad, and Vikram Garg. "Comparative Analysis of Floyd Warshall and Dijkstra's Algorithm Using Opencl." *International Journal of Computer Applications IJCA* 128.17 (2015): 4-6. International Journal of Computer Applications (0975 – 8887), Oct. 2015. Web. 19 Apr. 2016.
5. Singh, Dhirendra Pratap, and Nilay Khare. "Parallel Implementation of the Single Source Shortest Path Algorithm on CPU–GPU Based Hybrid System." *Academia.edu*. (IJCSIS) International Journal of Computer Science and Information Security,, Sept. 2013. Web. 21 Apr. 2016.
6. Singh, Dhirendra Pratap, Nilay Khare, and Akhtar Rasool. "Efficient Parallel Implementation of Single Source Shortest Path Algorithm on GPU Using CUDA." *Efficient Parallel Implementation of Single Source Shortest Path Algorithm on GPU Using CUDA* (n.d.): n. pag. *Http://www.ripublication.com*. International Journal of Applied Engineering Research ISSN 0973-4562 Volume 11, Number 4 (2016) Pp 2560-2567 © Research India Publications. Web. 18 Apr. 2016.
7. Thao, Nguyen Hoang, Nguyen Gia Duy, and Ngo Viet Hoang Hiep. "Classical Algorithms on Graphs." *Classical Algorithms on Graphs* (2011-2013): n. pag. *Http://dept-info.labri.fr/*. University of Bordeaux 1, 18 June 2012. Web. 19 Apr. 2016.