CSC 485c Project, All-Pairs Shortest Path

Aiden Bull, Gareth Marriott

February 2020

1 Introduction

Our project focuses on the all pairs shortest paths problem on directed graphs and optimising for large sparse data sets. The inspiration for the project was finding the solution to the all-pairs shortest path problem on Wikipedia. By this we mean treating Wikipedia pages as nodes and links from one internal page to another as directed edges. With this construction we aim to use a modified version of BFS to create a matrix that shows the shortest path from every page to every other page. As of January 23, 2020, the English Wikipedia has reached 6 million pages. To solve an all-pairs shortest path problem on a graph with over 6 million nodes will require heavy optimisations if we are to hope to achieve it.

2 Algorithm Description

As mentioned earlier, the problem we are trying to solve is the all-pairs shortest path problem. The first algorithm we explored for this problem was the Floyd-Warshall algorithm, which is a robust method for solving all-pairs shortest path. The Floyd-Warshall algorithm works by first initializing an adjacency matrix with the immediate edges between adjacent nodes. Then, it builds the shortest path for all pairs of nodes i to j by looping through every node on the graph k and seeing if each i-j path could be made shorter by replacing it with the path from i to k, then the path from k to j and replacing the path if so. The code for the main loop of Floyd-Warshall is shown in Figure 1.

Because of the loop described above, the Floyd-Warshall algorithm has time complexity $O(V^3)$, with V representing the number of nodes in the graph. In testing, this time complexity proved to be too poor as the running time of the program exploded long before we could reach a data size that would truly stress our memory capacity. Floyd-Warshall is a very robust algorithm that is capable of finding all-pairs shortest paths in graphs with positive and negative edge weights. But our initial goal of solving all-pairs shortest path on Wikipedia doesn't require weighted edges at all, as no link is more

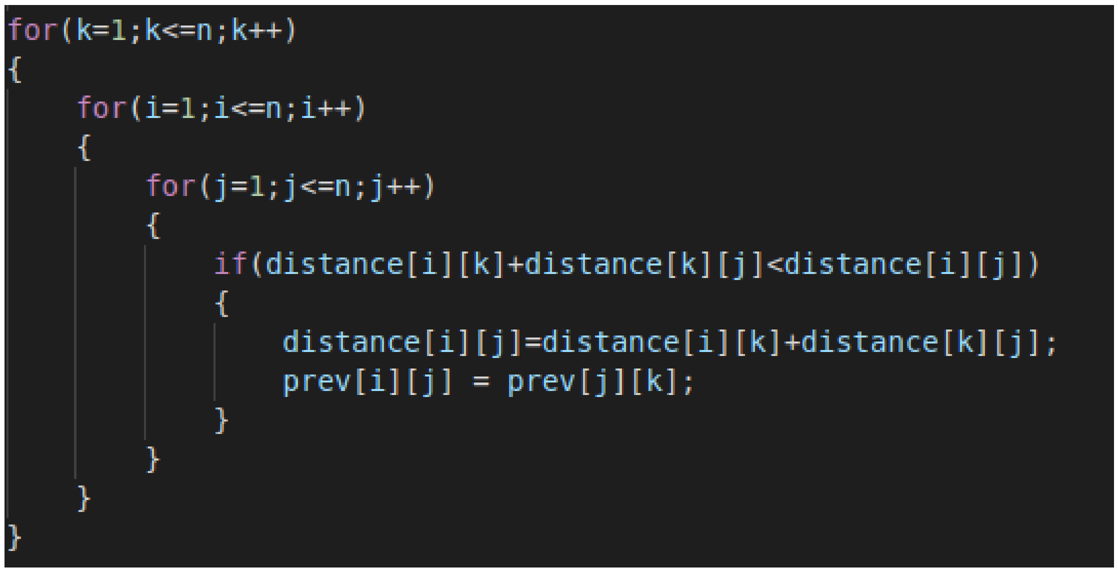
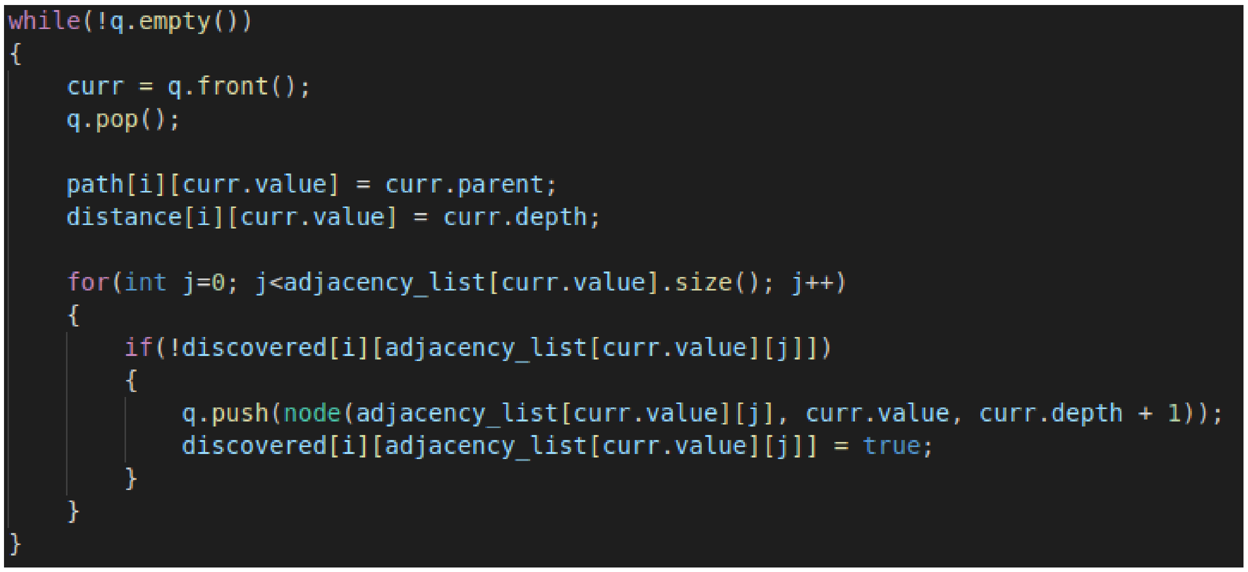

```
for(k=1;k<=n;k++)
{
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            if(distance[i][k]+distance[k][j]<distance[i][j])
            {
                distance[i][j]=distance[i][k]+distance[k][j];
                prev[i][j] = prev[j][k];
            }
        }
    }
}
```

Figure 1: Main loop of Floyd-Warshall

expensive to traverse than another. Because of this, we looked back at our options for algorithms and decided on a less robust, but more efficient algorithm.

The second algorithm we chose to explore was just running Breadth-First Search(BFS) on every node in the graph. In a graph with uniform positive edge weights, BFS will find the shortest path from a starting position to every other node it can reach. Because we can assume for our goal that each link could count as having an edge weight of one, BFS would find an optimal solution for our purposes. BFS works by first choosing a node as a starting point, and placing each node it's directly connected to into a FIFO queue. It then iteratively pops the next node x from the FIFO queue; updates the distance from the start to x; records the previous node in the shortest path from the start to x; and adds every undiscovered neighbouring node of x to the FIFO queue, marking those nodes as discovered.

BFS has a time complexity of $O(V + E)$, assuming an adjacency list is used to store neighbouring nodes, and not an adjacency matrix. For this notation, V represents the number of nodes in the graph and E represents the number of edges. This time complexity is for running BFS to solve shortest path on a single source. For all-pairs shortest path, we must run BFS with each node as a starting point. Because of this, our true time complexity is $O(V(V + E))$ or $O(V^2 + VE)$. Between the two terms V^2 and VE, typically VE is the larger term, as outside of extremely sparse graphs, there are often more edges than vertices for a graph.



```
while(!q.empty())
{
    curr = q.front();
    q.pop();

    path[i][curr.value] = curr.parent;
    distance[i][curr.value] = curr.depth;

    for(int j=0; j<adjacency_list[curr.value].size(); j++)
    {
        if(!discovered[i][adjacency_list[curr.value][j]])
        {
            q.push(node(adjacency_list[curr.value][j], curr.value, curr.depth + 1));
            discovered[i][adjacency_list[curr.value][j]] = true;
        }
    }
}
```

Figure 2: Main loop of BFS for a single source

Despite this, VE is always less than V^3, as a complete graph with directed edges will have $(V(V-1)) < V^2$ edges, and the number improves greatly as graphs become sparse. As our initial goal is to compute all-pairs shortest paths on the entirety of Wikipedia, our final graph will have 6 million vertices, but each vertex will have outgoing edges on the order of tens to hundreds. Because of this, an algorithm that benefits from sparse graphs will suit our goal well, and we believe BFS is a good choice. The code of the main loop of our baseline implementation of single-source BFS is shown in Figure 2.

The input for our all-pairs shortest path BFS implementation would be a text file, storing an adjacency matrix, and the output would be a distance matrix between nodes, and a path matrix from which the shortest path from some pair of nodes i to j could be inferred. The input matrix is an NxN matrix in which each row i represents one node, and each column value in that row j could be either a 1, to signify an existing edge from i to j, or a -1 to signify no edge. The path matrix at location (i, j) stores the most recent node travelled in the shortest path from i to j. To make the path matrix easier to read from users, the function *print_path(i, j)* has been added which prints the shortest path from i to j, if a path exists. An NxN ”discovered” matrix is also used to keep track of whether a node has been discovered in an execution of single-source BFS, and an adjacency list is used to store the input from the adjacency matrix text file.

3 Experiment Setup

Despite the motivation of the project being the Wikipedia database we wanted more control of the size and sparsity of our data sets. We created a adjacency matrix generator that takes 3 parameters: the number of nodes in the graph, the inverse of the chance that a possible edge exists(ie. parameter of 4 would give any given edge a 1/4% chance of appearing), and representation we want the non-edges to have(-1 or 2). The matrix generation at every possible edge evaluates the statement

```
    if (rand() % max_rand + 1 <= 1)
```

creating an edge if the statement is true and having no edge if the statement is false.(Where max_rand is the inverse of the chance and possible edge exists) In terms of benchmarking we used perf's built in run time tracker to calculate the speed of each execution. Our main test consisted of testing both the baseline and our implementation on a total of 16 matrix with sizes from 1000x1000 to 4000x4000 and four different sparsity levels 1/2, 1/4, 1/10, and 1/100. Our main test is in perf_testing.sh (part of perf_testing.sh below)

```
./create_matrix 1000 2 1 > 1000-2matrix.txt
./create_matrix 1000 4 1 > 1000-4matrix.txt
...
./create_matrix 4000 100 1 > 4000-100matrix.txt
...
perf stat -d -d -d ./$1 1000-2matrix.txt
perf stat -d -d -d ./$1 1000-4matrix.txt
...
perf stat -d -d -d ./$1 4000-1000matrix.txt
```

4 Overall Comparison

Once we implemented the loop unrolling we achieved an on average 21.19% speed increase.(Figure 6) The speed increase seems quite constant with the increasing size of the data set leading us to believe that the code is not hitting any sort of cache or memory walls.(See Figures 4-5)

Google Spreadsheet link

5 Diagnostics

From Floyd-Warshall to each implementation of BFS, using perf to profile each of our programs revealed that they were all stalling on roughly 40% of cycles. These programs were tested on a superscalar processor, so depite the high stall rate, we still displayed a fairly good instructions per cycle count of around 2. These stalled cycles are listed as "front end stalled cycles", so its likely our bottleneck is in the front end, and the back end is spending a significant amount of time waiting for instructions to be delivered to it. This is supported by the low number of L1 cache misses reported by perf and the high (sometimes 400% and above) instruction translation look-aside buffer miss rate. Unfortunately we couldn't get perf to display percentage of L1 instruction cache misses.

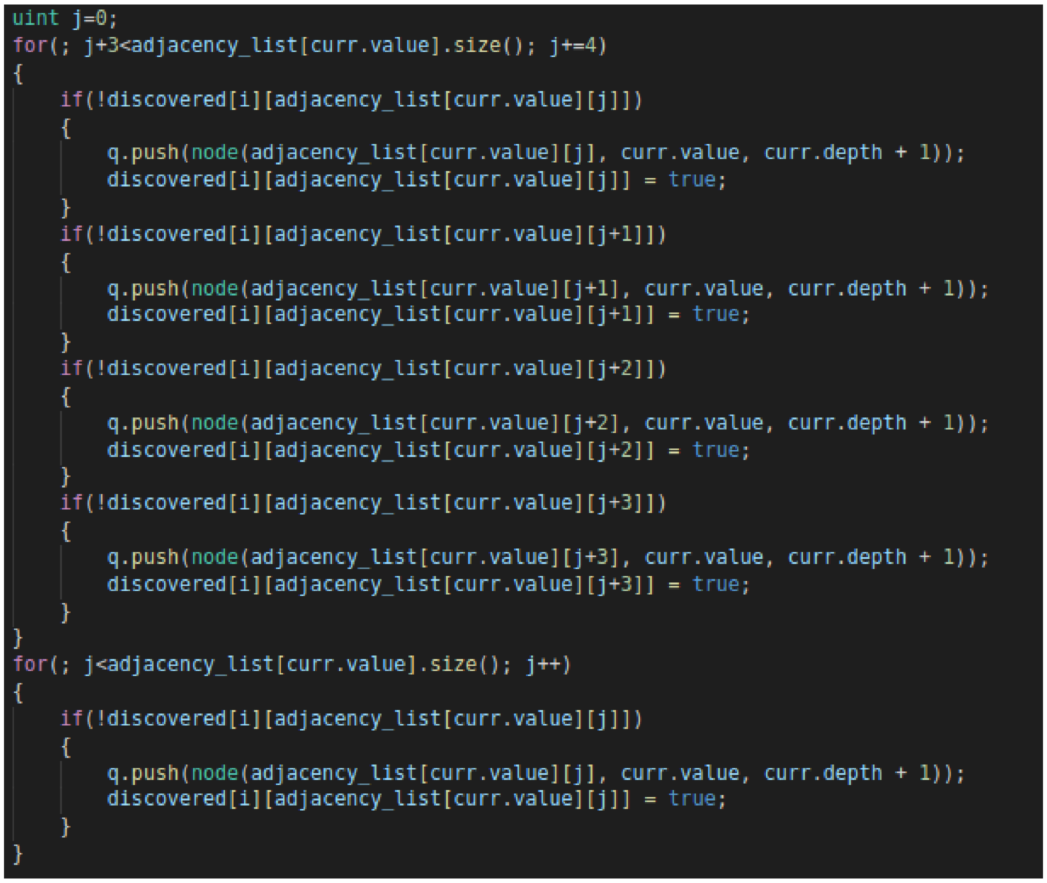
6 Single-Threaded optimisation

For this interim report, we were unfortunately unable to implement many single threaded optimisations. Due to a few reasons including lack of organization and confusion regarding due dates, we were only able to implement two optimisations that were covered in class. The first optimisation was an attempt to introduce instruction level parallelism by using multiple FIFO queues to allow us to potentially run single-source BFS on multiple starting points simultaneously. We did this by making a vector of queues, incrementing the outer loop of our all-pairs BFS implementation by the length of our queue vector, and adding an inner loop that would each run single-source BFS using a different source vector and a different queue. This optimisation should achieve instruction level parallelism because any changes made to our distance, path, and discovered matrices in an execution of single-source BFS is restricted to the row corresponding to that source. Despite this, the optimisation wasn't very effective at increasing the efficiency of our program, and in most cases, the final running time was actually worse than our baseline. This may be because the loop that we attempted to parallelize was too long or complex for the compiler to recognize the potential parallelism. A screen capture showing the main loop of our multiple FIFO queue BFS implementation is included in the appendices (Figure 9).

The second optimisation we implemented was to unroll the loop that pushes the neighbours of the current node onto the FIFO queue. We unrolled the expensive inner loop of single-source BFS to ideally reduce for loop condition branching and expose parallelism for out-of-order instruction execution. We implemented this by incrementing the loop counter by 4 instead of 1, and manually writing out the instructions 4 times in the body of the loop. If the condition of the for loop would become false before the next 4 iterations were completed, the loop would break out early and a second for loop would complete the remaining iterations. This change actually resulted in an improved running time across the board. With this modification, the programs would often run 10-20% faster than the baseline implementation. A screenshot showing the unrolled loop is displayed in the appendices in Figure 9.

7 Evidence to Support Optimisation Claims

Figures 4 and 5 display running time differences between our baseline implementation and our loop unrolling implementation. The different colours of the lines represent how many nodes V were in the graph being tested, with the corresponding numbers being displayed at the top of the figures. The horizontal positions of the points in the figures represents the sparsity of each graph, specifically the likelihood that any possible edge exists. As an example, the green line represents a graph with 4000 nodes, and the 1/100 point of the green line means that each of the 4000 nodes had approximately $4000 * (1/100)$ outgoing edges, in this case roughly 40 per node. As shown by the figures, the implementation with loop-unrolling showed a consistent speed up over the baseline implementation. Looking at the perf stats for the loop unrolling and baseline implementations, we can see that the loop unrolling implementation of BFS has consistently lower instruction counts, branch counts, and loads from caches and TLBs. The amount of reduced instructions and branches seem to be consistent with the reduced running time which likely indicates that they were the cause of the speed up. Looking at the number of front end stalled cycles we can see that the values haven't improved from the baseline to the new implementation, which seems to indicate that we were unsuccessful at targeting our bottleneck. There does appear to be a pattern of iTLB load miss rate being lower in the loop unrolling implementation than the baseline, but the variation in values is too high to be able to make any claims for certain, and it only seems to stabilize as we approach denser graphs with higher node counts.



```
uint j=0;
for(; j+3<adjacency_list[curr.value].size(); j+=4)
{
    if(!discovered[i][adjacency_list[curr.value][j]])
    {
        q.push(node(adjacency_list[curr.value][j], curr.value, curr.depth + 1));
        discovered[i][adjacency_list[curr.value][j]] = true;
    }
    if(!discovered[i][adjacency_list[curr.value][j+1]])
    {
        q.push(node(adjacency_list[curr.value][j+1], curr.value, curr.depth + 1));
        discovered[i][adjacency_list[curr.value][j+1]] = true;
    }
    if(!discovered[i][adjacency_list[curr.value][j+2]])
    {
        q.push(node(adjacency_list[curr.value][j+2], curr.value, curr.depth + 1));
        discovered[i][adjacency_list[curr.value][j+2]] = true;
    }
    if(!discovered[i][adjacency_list[curr.value][j+3]])
    {
        q.push(node(adjacency_list[curr.value][j+3], curr.value, curr.depth + 1));
        discovered[i][adjacency_list[curr.value][j+3]] = true;
    }
}
for(; j<adjacency_list[curr.value].size(); j++)
{
    if(!discovered[i][adjacency_list[curr.value][j]])
    {
        q.push(node(adjacency_list[curr.value][j], curr.value, curr.depth + 1));
        discovered[i][adjacency_list[curr.value][j]] = true;
    }
}
```

Figure 3: Unrolled inner loop of single-source BFS

8 Appendix

Here is a link to our GitHub repo for the first interim report.

8.1 A - Figures

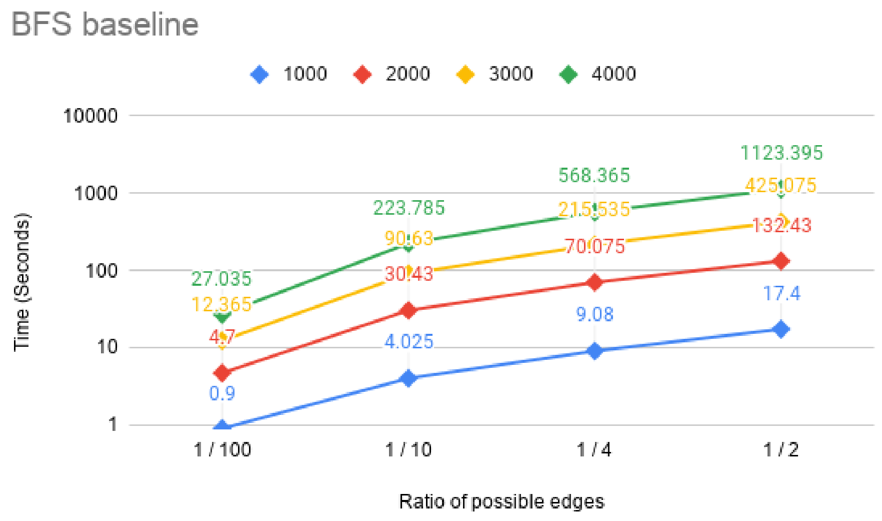


Figure 4: BFS Baseline speeds for different data sets

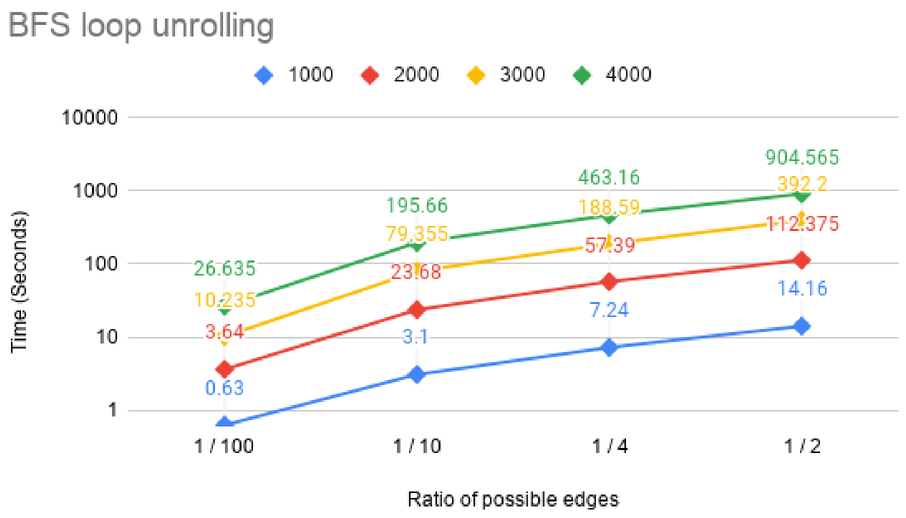
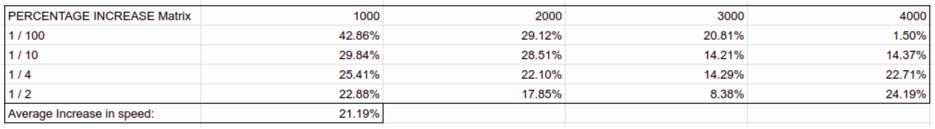
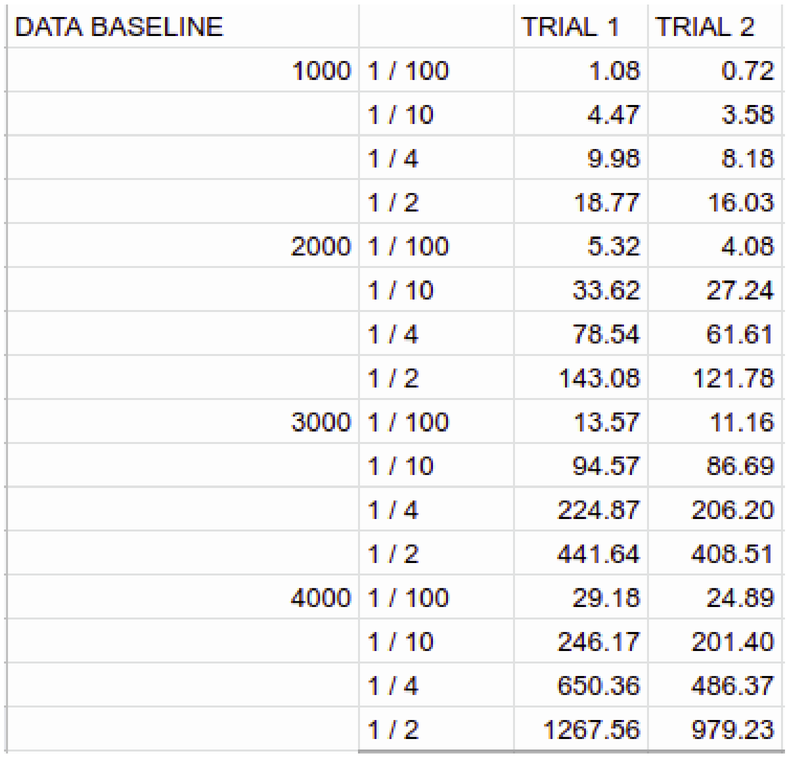


Figure 5: BFS Loop unrolling speeds for different data sets



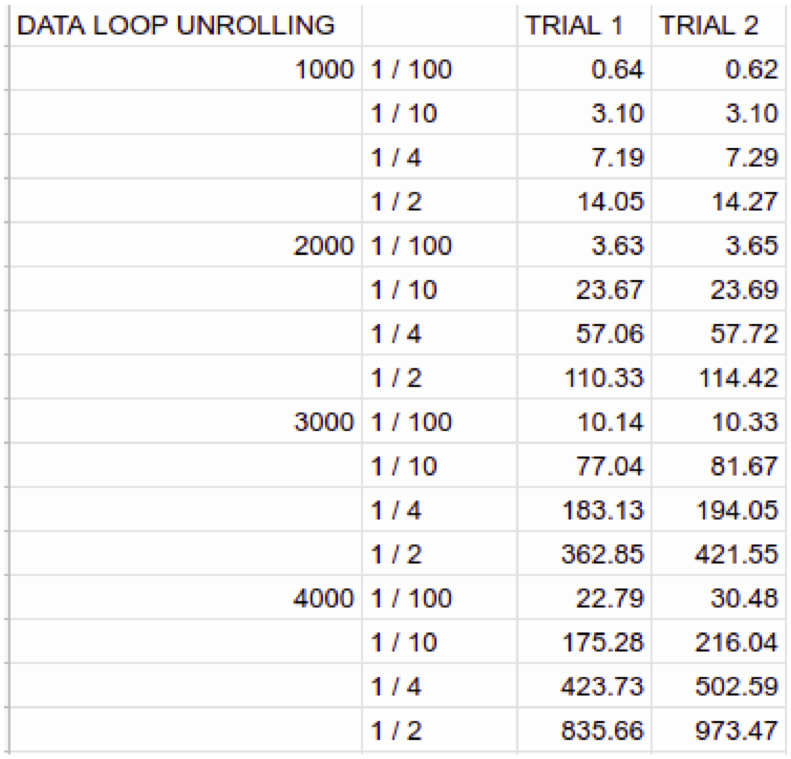
PERCENTAGE INCREASE Matrix	1000	2000	3000	4000
1 / 100	42.86%	29.12%	20.81%	1.50%
1 / 10	29.84%	28.51%	14.21%	14.37%
1 / 4	25.41%	22.10%	14.29%	22.71%
1 / 2	22.88%	17.85%	8.38%	24.19%
Average Increase in speed:	21.19%			

Figure 6: Percentage increase in speed



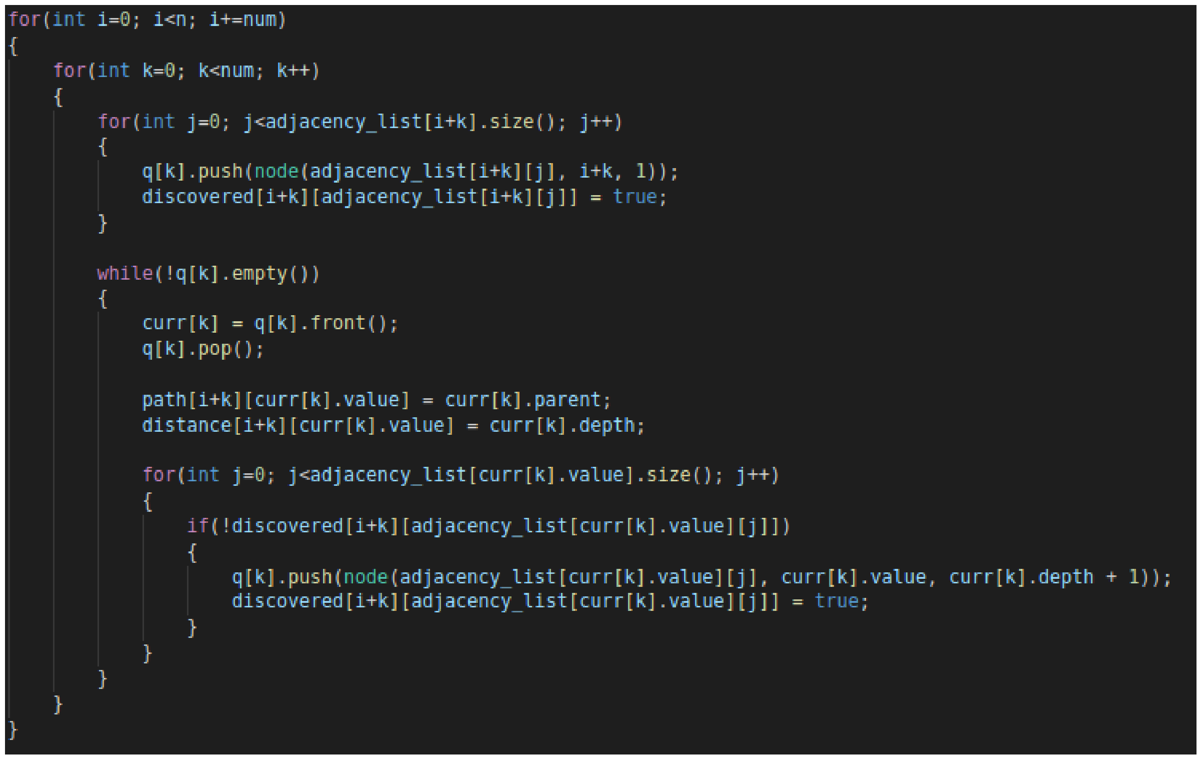
DATA BASELINE		TRIAL 1	TRIAL 2
1000	1 / 100	1.08	0.72
	1 / 10	4.47	3.58
	1 / 4	9.98	8.18
	1 / 2	18.77	16.03
2000	1 / 100	5.32	4.08
	1 / 10	33.62	27.24
	1 / 4	78.54	61.61
	1 / 2	143.08	121.78
3000	1 / 100	13.57	11.16
	1 / 10	94.57	86.69
	1 / 4	224.87	206.20
	1 / 2	441.64	408.51
4000	1 / 100	29.18	24.89
	1 / 10	246.17	201.40
	1 / 4	650.36	486.37
	1 / 2	1267.56	979.23

Figure 7: baseline data



DATA LOOP UNROLLING		TRIAL 1	TRIAL 2
1000	1 / 100	0.64	0.62
	1 / 10	3.10	3.10
	1 / 4	7.19	7.29
	1 / 2	14.05	14.27
2000	1 / 100	3.63	3.65
	1 / 10	23.67	23.69
	1 / 4	57.06	57.72
	1 / 2	110.33	114.42
3000	1 / 100	10.14	10.33
	1 / 10	77.04	81.67
	1 / 4	183.13	194.05
	1 / 2	362.85	421.55
4000	1 / 100	22.79	30.48
	1 / 10	175.28	216.04
	1 / 4	423.73	502.59
	1 / 2	835.66	973.47

Figure 8: loop unrolling data



```
for(int i=0; i<n; i+=num)
{
    for(int k=0; k<num; k++)
    {
        for(int j=0; j<adjacency_list[i+k].size(); j++)
        {
            q[k].push(node(adjacency_list[i+k][j], i+k, 1));
            discovered[i+k][adjacency_list[i+k][j]] = true;
        }

        while(!q[k].empty())
        {
            curr[k] = q[k].front();
            q[k].pop();

            path[i+k][curr[k].value] = curr[k].parent;
            distance[i+k][curr[k].value] = curr[k].depth;

            for(int j=0; j<adjacency_list[curr[k].value].size(); j++)
            {
                if(!discovered[i+k][adjacency_list[curr[k].value][j]])
                {
                    q[k].push(node(adjacency_list[curr[k].value][j], curr[k].value, curr[k].depth + 1));
                    discovered[i+k][adjacency_list[curr[k].value][j]] = true;
                }
            }
        }
    }
}
```

Figure 9: Code snippet of the main loop of all-pairs shortest path BFS using multiple FIFO queues