CSC 485c Project, All-Pairs Shortest Path, Term Report

Aiden Bull, Gareth Marriott

April 2020

Introduction

For the past three months we've been looking into methods for optimizing a solution to the all-pairs shortest path (APSP) problem. Over those three months we first decided on which algorithm to work with and developed a baseline. We then optimised that algorithm for single core computing, then multicore computing, then finally for GPGPU computing. In this paper we will describe the process and results from our project. GitHub Link

Problem and Baseline Algorithm

The problem we chose to tackle was the all-pairs shortest path problem. In particular, we were aiming to build an algorithm that would compute APSP between Wikipedia articles. What we mean by this is for our graph, Wikipedia pages would be our nodes, and hyperlinks to other pages would be outgoing directional edges. We at first looked at solving the problem using the robust Floyd-Warshall algorithm. We then realized that because we decided to work on this Wikipedia version of APSP, we weren't aiming to solve the problem on a general graph, but rather on an unweighted directional graph. Allowing the assumptions that edges are directional and unweighted meant that we could choose a less robust but more efficient algorithm. The algorithm we ended up choosing was to simply run Breadth-first Search (BFS)[1] on every node and combine the outputs. This algorithm has a competitive time complexity when graphs are sparse, which makes it well suited to our task.

Our baseline algorithm simply works by running single source BFS in a loop starting from each node. Our algorithm made use of a few data structures: an adjacency list created from an input text file, a distance matrix keeping track of how many links each destination is from each source, a path matrix to store the shortest path for each node pair, and a discovered matrix to keep track of which destination nodes are discovered by which source nodes. Each of these data structures were stored using 2D vectors. An image of our BFS baseline code is shown in figure 1.

Changes Made to Algorithm

Throughout the term, we submitted three versions of our algorithm. In each submission we took our code from the previous submission and expanded it to another computing environment. In our first submission we analyzed our baseline code using a

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

    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 1: Baseline BFS code

profiling tool, then attempted to address the bottlenecks we found. In our second submission, we took our optimized single threaded program and expanded it to take advantage of a multi-core system. Finally with our third submission, we took the algorithm we had up to that point, and expanded it to take advantage of GPU hardware to get even greater speed ups.

For our first submission, we began by analyzing the performance statistics of our program using the Linux profiling tool, perf. In testing our program we found that perf was reporting a high amount of front-end stalled cycles (40%). Upon later testing with Intel's VTune profiler these statistics were found to be incorrect; our program was actually experiencing more back end stalls. We didn't figure that out until the second submission, however, and so we assumed that front-end stalls were our major bottleneck. By referring to the Intel top-down analysis method[1] and viewing more perf stats we came to the conclusion that our program was likely stalling due to instruction cache misses or ITLB misses. We were uncertain how to approach this bottleneck as most of our lessons up to that point had focused on improving data cache miss rates and improving performance on superscalar processors. Despite this we still attempted to apply some methods from class.

We applied two optimization techniques to the baseline code. The first was to attempt to improve instruction level parallelism by using a vector of FIFO queues instead of a single queue and modifying the outer loop so subsequent BFS iterations would use different queues. This change would break dependency chains between instructions, essentially allowing outer loops to be run in parallel and ideally assisting instruction prefetching. Perhaps unsurprisingly, this change was unsuccessful in improving performance. This is likely due to the size and complexity of the outer loop. The second optimization we applied was to apply loop unrolling to the for loop that pushes neighbours of the current node to the FIFO queue. This change was successful in improving performance of our program. After applying this change, we saw a 10-20% decrease in run time across our tests. An image showing the loop unrolling modified code is shown in figure 2.

For our second submission we expanded our optimised single-threaded solution to take advantage of multi core parallelism. The way we did this took heavy advantage of our choice of baseline algorithm. We initially chose BFS for our baseline algorithm because of its competitive asymptotic time complexity, but the algorithm also lends itself very well to parallelism. Because each iteration of the outer loop of APSP BFS essentially runs a full execution of single-source BFS, Parallelism could be easily achieved by dividing outer loop iterations between cores. Following this idea, we achieved the majority of our parallelism by dividing outer loop iterations between threads using a pragma omp parallel for loop. We then only needed to worry about race conditions. Of the matrix data structures we use, the path, distance, and discovered matrix are all written to in an iteration of single-source BFS and as such all have potential to cause race conditions. However, each of these matrices are only written to and read from at the first dimension index corresponding to the outer BFS loop iteration. An exception to this is the adjacency list, but that data structure is never written do after preprocessing, and as such can not be a source of race conditions. In addition to parallelising BFS, we also parallelised the initialization of every matrix data structure except for the adjacency list, which was being initialized using getline(), and as such could not be made parallel.

The third and final submission required us to take advantage of GPU hardware. We achieved this by parallelising BFS in a similar way to submission 2, but this time across GPU cores rather than CPU cores. We did this by using Nvidia's CUDA model. To parallelise BFS, we had to convert it to a CUDA kernel function. Doing this required us to copy memory from host to device and vice versa, and also to convert the BFS code from C++ to C. Converting the code to C importantly meant converting C++ data structures to a C usable format. In particular this meant that we required a replacement for the vector data structures and the FIFO queue. To replace the vector data structures, we first converted them into a 1D row-order array of the same type, then manually indexed the array

```
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 2: Neighbour-pushing loop with loop unrolling applied.

in the kernel. To replace the FIFO queue, we allocated an array of nodes in the kernel and kept track of the head and tail via two integers. The main loop of the BFS kernel code is displayed in figure 3.

An additional change was implemented since the third submission for the purpose of this report. A few of the earlier submissions took input in the form of an adjacency matrix instead of an adjacency list. We switched to adjacency list input as it can be processed much faster than an adjacency matrix. For this report all tests were run with adjacency list input to make tests more fair.

Results

We implemented two different modes of generating adjacency lists throughout the three months each with its pro's and con's. Our first method was to assume a node had a given chance of an edge pointing to another node in the graph and generating the graph by running this random chance for each possible edge. This gave us our sparseness factor which was; for any possible edge that edge has a

$$\frac{1}{sparseness\_factor}$$

chance of existing. This was a binomial approach of generating edges and when approximated to nor-

```
//idx == threadIdx.x + blockIdx.x * blockDim.x
for(int j=0; j<dev_adjacency_size[idx]; j++)
{
    queue[tail++] = node{dev_adjacency_list[dev_adjacency_offset[idx]+j], idx, 1};
    dev_discovered[idx*n + dev_adjacency_list[dev_adjacency_offset[idx]+j]] = true;
}

while(head != tail)
{
    curr = queue[head++];

    dev_path[idx*n + curr.value] = curr.parent;
    dev_distance[idx*n + curr.value] = curr.depth;

    for(int j=0; j<dev_adjacency_size[curr.value]; j++)
    {
        if(!dev_discovered[idx*n + dev_adjacency_list[dev_adjacency_offset[curr.value]+j]])
        {
            queue[tail++] = node{dev_adjacency_list[dev_adjacency_offset[curr.value]+j], curr.value, curr.depth + 1};
            dev_discovered[idx*n + dev_adjacency_list[dev_adjacency_offset[curr.value]+j]] = true;
        }
    }
}
free(queue);
```

Figure 3: CUDA kernel BFS loop.

mal gave us a distribution of:

$$normal(\frac{n}{sparseness\_factor}, \frac{n(sparseness\_factor - 1)}{sparseness\_factor^2})$$

this would give for example a graph with n = 5000 and sparseness factor = 100 a mean of 50 and SD of 499.5. This method served us well enough when using small size graphs but as we expanded to larger data sets our data began to not represent the data set of Wikipedia well. We therefore implemented a new adjacency list generator that directly created edges of our graph based off of a normal distribution. This had the benefit of removing the link between the size of the data set and the mean and standard distribution. We therefore had to data mine a subset of the Wikipedia database to determine a representative mean and standard distribution. We used a sample of 286 articles which gave us a mean of 114 and a standard deviation of 103. This mean and standard deviation was used for all tests involving the normal distribution. It is important to note both the binomial and normal distribution are bounded by zero and n. This affects the two distributions differently. The binomial distribution will have a clustering at zero edges, where the normal distribution recalculates the number of edges if it falls out of the bounds. Therefore all edges that would fall below zero in the normal distribution are spread throughout the data set. All of our data is available on a Google spreadsheet.

The Wikipedia data was scraped using a python scraper and the Beautiful soup library. The get_all_pages.py file would scrape the Wikipedia page reference for every page link and then using the scrap.py file download links for a subset of the pages. Once a subset was downloaded we used get_avg_links.py to get the number of links on each page and therefore mean and standard deviation of

the set of pages.

Our final three implementations correspond to the interim reports submitted and are all four implementations run on the Azure server using shell scripts. Our baseline is an implementation of the BFS algorithm. The loop unrolling implementation focuses on loop unrolling because that was most successful in achieving speed increases. The multi core implementation uses omp multi threading to split workload amongst 6 cores. Finally our GPU implementation utilizes CUDA to run on the azure GPU.

Each interim report saw speed ups from the last and due to the inherent parallelisability of the problem we chose we saw the most significant speed ups when implementing the 6 core CPU solution and the GPU solution. The three implementations and the baseline are plotted on graphs with there respective inputs (figures 5, 6, 7, 8).

Analysis

Our single-threaded loop unrolling implementation saw a 14% increase in speed over the baseline. The loop that loop unrolling was applied to is expected to take up a majority of a BFS iterations runtime, so helping to reduce branching from the for loop could explain the speed increase. Additionally it could be that loop unrolling helped with instruction prefetching making it easier for superscalar processors to make better use of their ports.

Due to our algorithm choice we struggled to see non parallel speedups but saw great speedups when it came to multi core CPU and GPU implementations. We saw very modest increases in speed when implementing loop unrolling but saw an average of 5.7x speed up when going from baseline to multi core CPU and 30x speed up from baseline to GPU(When measured on the normal distribution graphs, figure 4). An important note is the GPU implementation is slower then the multi core CPU implementation at small data sets due to increased time required to set up the GPU but quickly surpasses the multi core CPU implementation. The speed reached by the GPU implementation at 30k nodes is 82x faster and we see no reason this speed up couldn't continue to increase.

Since our tests were done on 6 cores we can project our results onto a 16 core machine. From our speed ups from baseline table it is clear that the speed ups achieved by the 6 core CPU implementation are not influenced by the set size so it is reasonable to assume our data is not running into a memory wall. Therefore it is safe to assume with six cores achieving almost a 6x speed up that sixteen cores would achieve well over a 10x speed up.

Our code shows a clear ability to be paralleled as evident in our VTune profiling results. Our single threaded implementation resulted in 30% of cycles being core bound stalls and our 4 thread implementation reduced this to 13% (figures 9, 10).

How to Reproduce Results

To reproduce our results navigate to the Final_Report folder and run one of 4 shell scripts:

Size	Loop Unrolling	CPU	GPU
1000	15.33%	565.22%	170.80%
2000	13.51%	570.22%	759.91%
3000	11.76%	576.97%	1341.68%
4000	11.39%	565.03%	1925.52%
5000	12.58%	514.08%	2529.58%
6000	13.39%	587.84%	2975.86%
7000	12.10%	574.23%	3359.37%
8000	14.47%	590.18%	3321.71%
9000	14.49%	593.31%	3553.03%
10000	14.70%	593.52%	3708.49%
20000	13.49%	574.95%	5022.88%
30000		582.80%	8212.03%
	13.38%	574.03%	3073.41%

Figure 4: Speed ups from baseline ((baseline / tested) - 1)

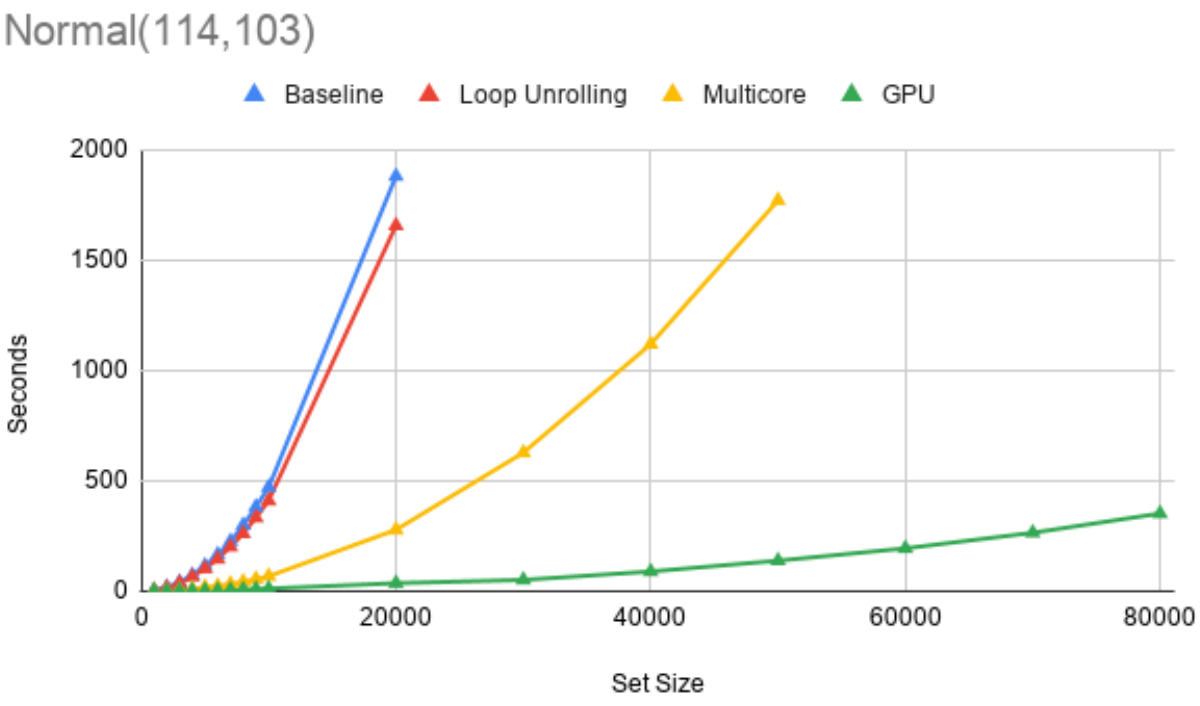

Figure 5: Speed of implementations tested on normal distribution

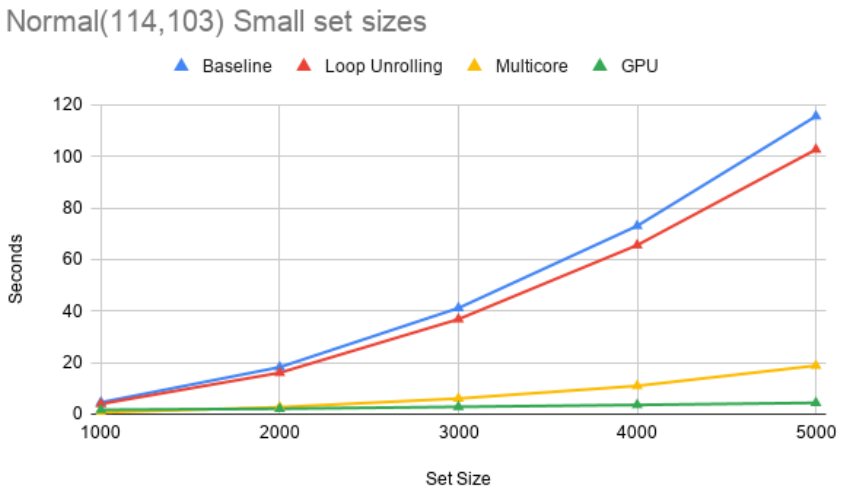

Figure 6: Speed of implementations tested on normal distribution(zoomed in on small set sizes)

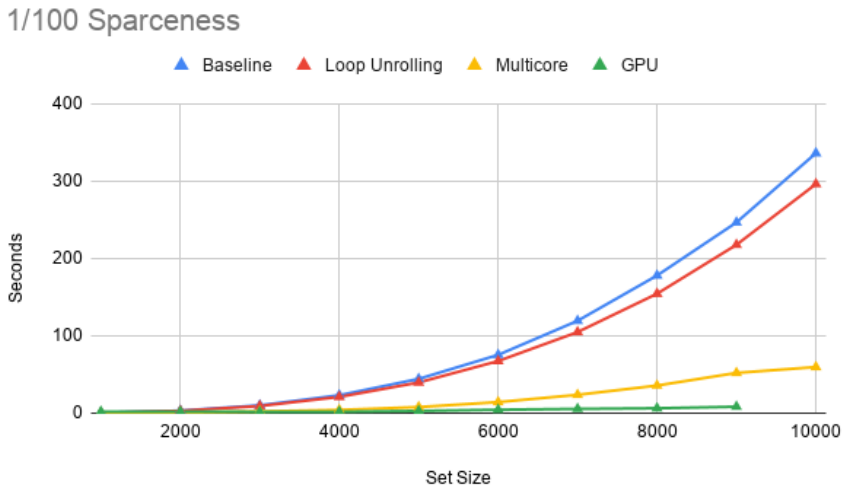

Figure 7: Speed of implementations tested on graph with spareness factor 100

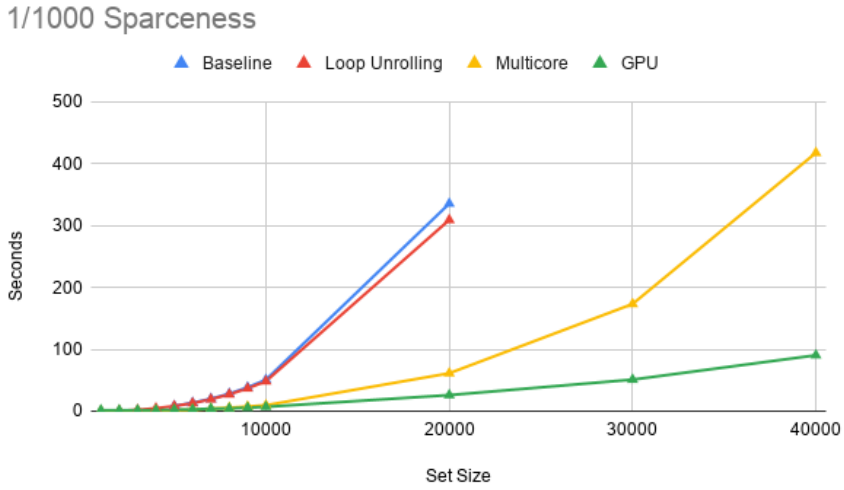

Figure 8: Speed of implementations tested on graph with spareness factor 1000

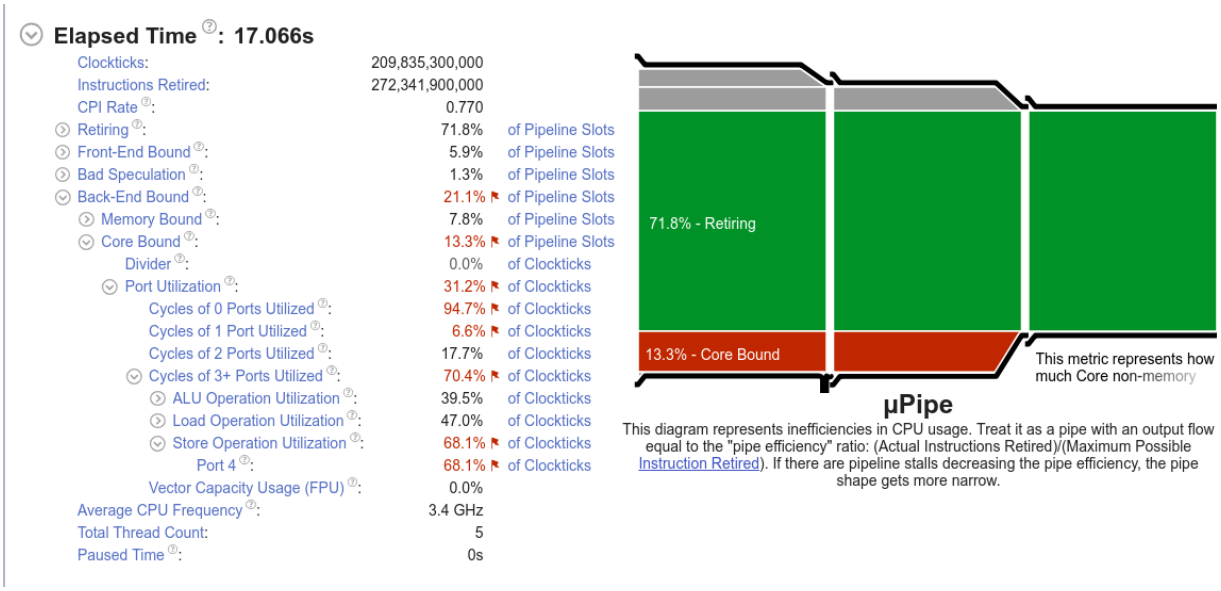

Figure 9: VTune results for 4 core implementation

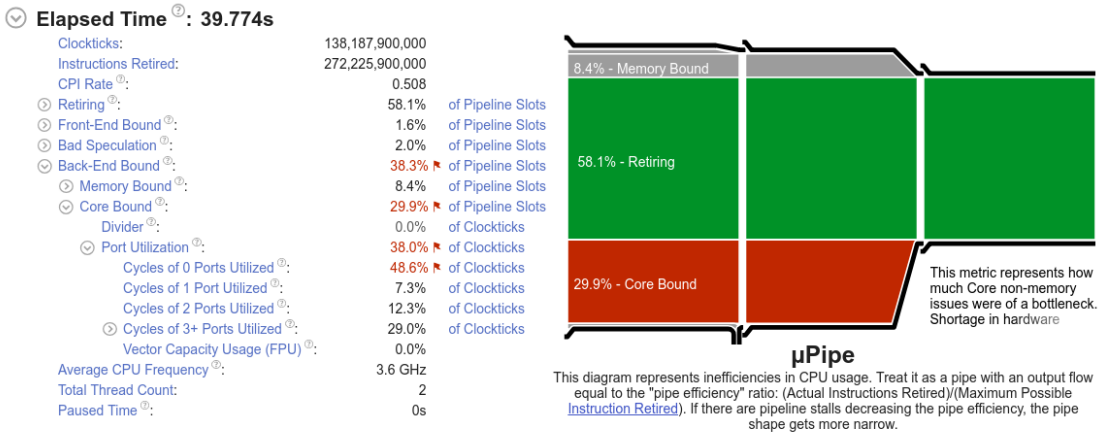

Figure 10: VTune results for 1 core implementation

1. sh remote_testing.sh $\langle file\_to\_be\_tested \rangle$ $\langle number\_of\_threads \rangle \langle file\_matrix\_generator \rangle$
2. sh $remote\_testing\_cuda.sh$ $\langle file\_to\_be\_tested \rangle$
3. sh $remote\_testing\_distribution.sh$ $\langle file\_to\_be\_tested \rangle \langle number\_of\_threads \rangle \langle mean \rangle \langle SD \rangle$
4. sh $remote\_testing\_distribution.sh$ $\langle file\_to\_be\_tested \rangle \langle mean \rangle \langle SD \rangle$

GitHubLink

Conclusion

Over the course of this project, we found success in making our code more effective at making use of available hardware. Our single threaded implementation found less success than our multithreaded implementations, but each were able to achieve a speedup over the last. It was helpful having a real world problem to aim for as we solved the more general APSP on unweighted undirected graphs. Through this project we learned valuable skills for taking advantage of modern hardware in different computing environments. These skills are becoming increasingly essential for any programmer hoping to take full advantage of their computer.

References

[1] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, 2019.

Appendix: Group Contributions

0.1 Aiden

Aiden worked on roughly half of the coding, focusing more on the BFS implementations. For reporting he focused more on writing the problem and implementation descriptions.

0.2 Gareth

Gareth worked on roughly half the code focusing on the matrix creation in cpp and Wikipedia web scraping in Python. Gareth also wrote the sections on the report regarding results of the testing and data and handled the testing and data input, compilation and presentation for all the reports.

0.3 Additional comments

A large amount of code writing was done using pair programming to help with solving difficult problems. Both partners found the pairing very productive and solved problems effectively together.