Exam 1

Riley Kopp

18 October 2020

Program Description:

Floyd-Warshall Algorithm:

This program uses the Floyd-Warshall shortest path algorithm to find the shortest path between any points i and j in a given weighted directional graph G. Link to Wiki

Algorithm 1: Floyd Warshall Algorithm

```
Result: A matrix containing the shortest path between each set of points.
Adj = the adjacentcy matrix for the graph;
n = the number of nodes in the graph;
k = 0;
i = 0;
j = 0;
dist = Adj;
for k < n do
    for i < n do
        for j < n do
            if dist[i][j] > dist[i][k] + dist[k][j] then
                dist[i][j] = dist[i][k] + dist[k][j];
            end
        end
    end
end
```

To parallelize this algorithm, the trivial solution of parallelizing the outer loop was not an option due to the value at i, j being referenced in multiple threads at the same time which would cause a data race. Instead, the solution is to parallelize the i loop.

Code Break Down:

Organization of Code:

The code is broken down into 3 folders. The folders `serial`, `omp`, and `cuda` contain the floyd warshall algorithm implemented in the respective parallelization library. `main.cu` parses command line arguments, loads the graph based on the arguments and runs the specified algorithms.

Libraries Used:

There were no libraries used beyond the expected (math, time, stdio, stdlib, omp...)

How to Use the program:

As stated earlier, the program is designed such that one executable can run OMP, Cuda and a serial version of the algorithm.

To compile the program simply run the command:

```
$ make
```

Then to run program, type one of the following:

```
# Run all 3 programs on a known graph and check the solution.
./run_exam_1 test_1.txt

# Run a single algorithm on a known graph and check the solution.
./run_exam_1 -<s|o|c> test_1.txt

# Run a single algorithm on a random graph.
./run_exam_1 -<s|o|c> -r <graph size>
```

Testing and Verification

In order to verify the code, we can supply the executable with a text file that contains a graph along with the 'solved' graph that the program will compare it to. See Appendix A for an example file.

Data

Table 1: Parallel Speeds

Nodes	CUDA	OpenMP	Serial
500	118	393	431
1000	419	2805	3375
1500	1680	9709	1141
2000	4344	22731	27372
2500	8250	44484	54148
3000	14140	77135	93250

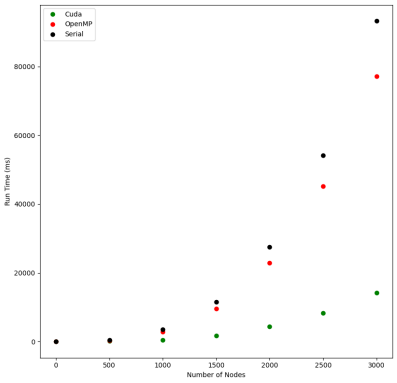

Figure 1: Algorithm Comparison

Analysis

Using the equation: $S = t_s/t_p$.

Table 2: Speedup

Nodes	CUDA	OpenMP
500	3.65	1.09
1000	8.05	1.2
1500	6.79	1.17
2000	6.3	1.2
2500	6.56	1.21
3000	6.59	1.2

Using the equation $E = t_s/(p * t_p)$.

Table 3: Efficiency

Nodes	CUDA	OpenMP
500	.73	13.7
1000	.80	15.04
1500	.45	14.69
2000	.31	15.02
2500	.26	15.2
3000	.21	15.1

Using the equation $SF = (1/S_p - 1/p)/(1 - 1/p)$.

Table 4: Serial Fraction

Nodes	CUDA	OpenMP
500	.27	.90
1000	.12	.81
1500	.14	.83
2000	.15	.8
2500	.15	.79
3000	.15	.8

Scalablity:
Since we have experimentally determined that there is 80% of the OMP code that is serial we can use Amdahls law ($S = 1/((t_p/p) + t_s)$) to say that our scalablity cannot be more than a 1.25 times speedup.

In CUDA, however, our serial fraction is .15 meaning that our scalablity cannot be more than a speed of up 6.66 times.

Both of these numbers can be confirmed based on the values in table 2.

A Example Test File

Test files are formatted as follows. The first line contains the number of nodes in the graph, we will call that value n. The next n lines should contain the adjacency matrix, followed immediately by the solution matrix in the next n lines.

```
6
0 2 5 inf inf inf
inf 0 7 1 inf 8
inf inf 0 4 inf inf
inf inf inf 0 3 inf
inf inf 2 inf 0 3
inf 5 inf 2 4 0
0 2 5 3 6 9
inf 0 6 1 4 7
inf 15 0 4 7 10
inf 11 5 0 3 6
inf 8 2 5 0 3
inf 5 6 2 4 0
```

B Submission

Within the tarball submitted you will find the code written for the project along with a test graph and a make file.