CS542200 Parallel Programming

Homework 3: All-Pairs Shortest Path (CPU)

109065517 林佳縈

1. Implementation

In this section, I'd like to briefly introduce how I solve "All-pairs shortest path CPU version". First of all, I'd point out which algorithm I chose; thus, I'll go through my implementation process. Last, I'll touch on the part about how I designed and generated my own testing case.

a. Algorithm

To solve the problem, I chose the Floyd-Warshall algorithm. The sequential version of APSP pseudocode could be written as follow, where

A: an adjacency matrix of the input graph

n: number of vertices in that graph

```
func Floyd_All_Pairs_Shortest_Path(A){
        D(0) = A;
        for k:=1 to n do
            for i:=1 to n do
                for j:=1 to n do
                    di,j(k) := min(di,j(k-1), di,k(k-1) + dk,j(k-1))
}
```

b. Implementation

The implementation is all based on the sequential Floyd-Warshall algorithm, but for further parallelization, I chose OpenMP.

As the most outer loop k stands for the iteration times, which seems not parallelizable, I plan to parallelize either the second or the third loop. So this is how my structure eventually looks like:

```
for(int k=0; k<V; k++)
    #pragma omp parallel for num_threads(omp_get_max_threads())
     for(int i=0;i<V;i++)
        for(int j=0;j<V;j++)
            if(dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j] = dist[i][k] + dist[k][j];
```

c. Time Complexity

Time Complexity = $O(V^3/P)$,

V = number of vertices
P = number of CPUs

d. Testing Case Generation

As the complexity is greatly influenced by number of nodes, I plan to have my testing cases with max nodes: 5999 and max edges: 5999*5998. I generate edges with random number within the range of [0 – (2^{30}-1)].

e. Other Optimization

For other optimization that I've performed during the implementation, I mainly focus on optimization on File I/O as I found that I/O could be a bottleneck for testing cases with large number of nodes and edges.
Therefore, for Input optimization, I adjusted the original methods: fstream with mmap; while for output optimization, fstream is thus adjusted with FILE fwrite.

Input Optimization

The original version with fstream, reading each character one by one.

```
std::ifstream f(argv[1]);
f.read((char*)&V, sizeof V);
f.read((char*)&E, sizeof E);
for (int i = 0; i < E; i++){
        f.read((char*)&p1, sizeof(int));
        f.read((char*)&p2, sizeof(int));
        f.read((char*)&dis, sizeof(int));
}
```

The adjusted version with mmap: read directly a memory section to an 1D array.

```
result = mmap(0,len,PROT_READ,MAP_FILE|MAP_PRIVATE,fd,0);
```

After some experiments, I found that there is significant time reduce by using **mmap**. For further SpeedUp analysis, I will discuss more in detail in Experiment and Analysis part.

Output Optimization

The original version using fstream to write result into file, which requires 2 nested for loop:

```
std::ofstream out(argv[2]);
for (int i = 0; i < V; ++i)
    for (int j = 0; j < V; ++j)
        out.write((char*)&dist[i][j], sizeof(int));
```

The adjusted FILE fwrite, which requires only one for loop, with one row written to file each time:

```
FILE *fp = fopen( argv[2] , "wb" );
for (int i = 0; i < V; ++i)
     fwrite(dist[i], sizeof(int), V, fp);
```

2. Experiment & Analysis

a. System Spec

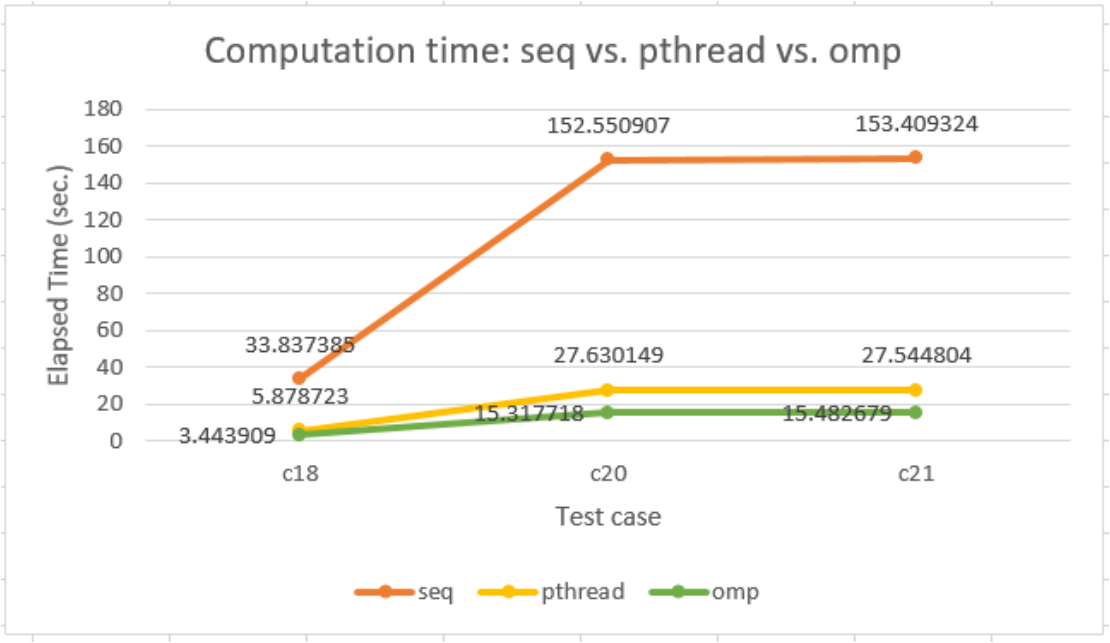
During the entire implementation and testing phases, I performed them both on

Apollo.

b. Strong Scalability

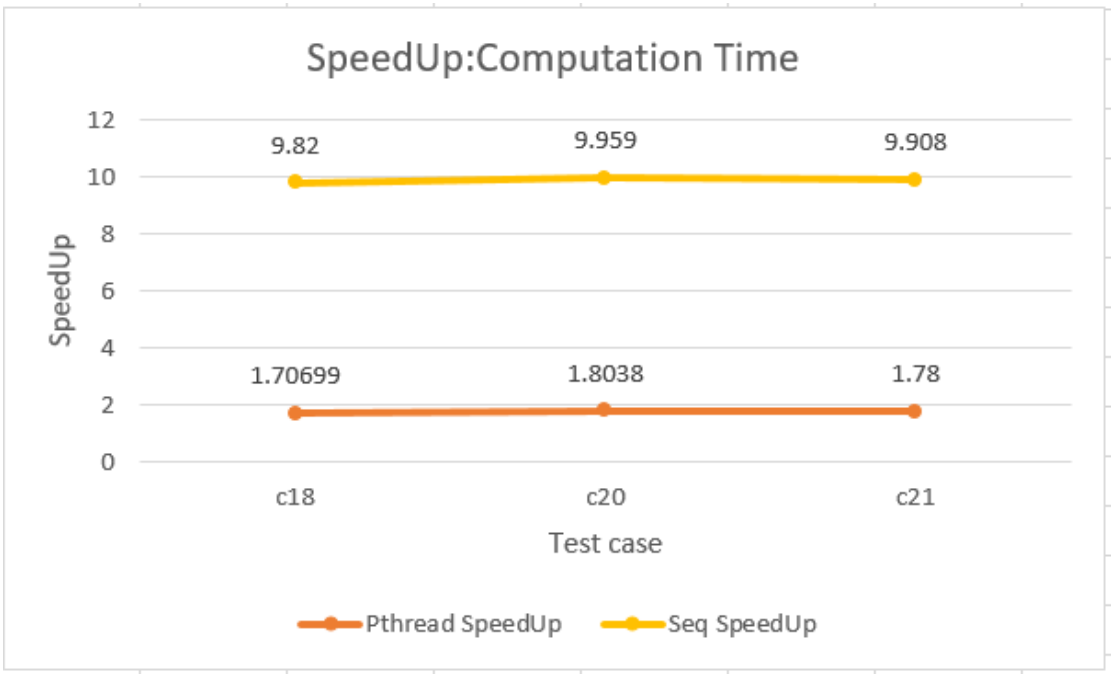
For scalability experiments, I use testing cases c18.1, c20.1, c21.1 to test my scalability. To clearly demonstrate the scalability, I compare my final submit version, which is implemented by OpenMP, with sequential and Pthread version. Experiments are divided into 3 different parts:

- **Computation Time:**



▲ 2b.1: Computation time

According to graph 2b.1, we can simply tell that implementation by openmp has the best performance among all the other implementation methods. To clarify, let's see the speedup evaluation.



▲2b.2: Computation SpeedUp evaluation

From graph 2b.2, how openmp is faster than other methods is calculated as following:

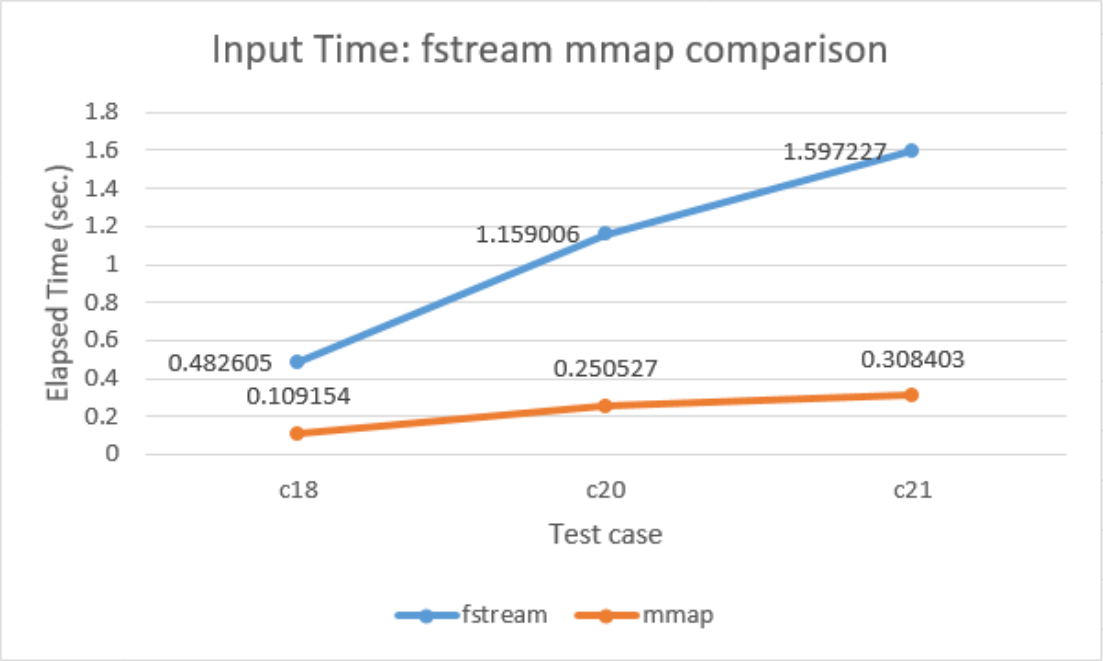
Pthread SpeedUp = pthread computation time / omp computation time

Seq SpeedUp = seq computation time / omp computation time

So again, openmp wins with relatively less computation time.

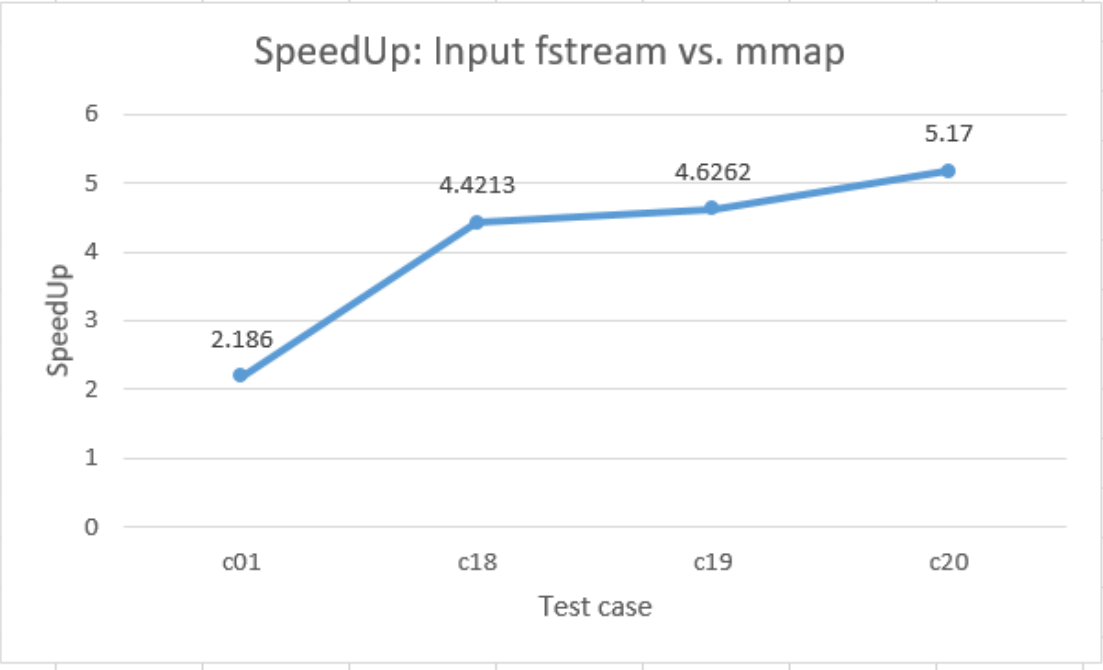
- **Input Time:**

  After feeding in several testing cases with huge amount of nodes/edges, I found that the time for file reading is also a part that I should pay attention to, which seems to take quite a long time doing file reading. Therefore, I adjusted the original way to ***mmap.*** By directly mapping a section of memory, time for file reading is speed upped with around 5x.



▲ 2b.3: Input Time Comparison between using ***fstream*** and ***mmap***.

With c18, c20, c21 stands for different testing cases contain relatively huge amount of input data, we can easily tell from the graph that by adjusting the way the program deals with file input, we can thus significantly reduce the input time spent.



▲2b.4: SpeedUp graph representing speedup by adjusting fstream with mmap.

To more clearly demonstrate the speedup, I take also one testing case c01

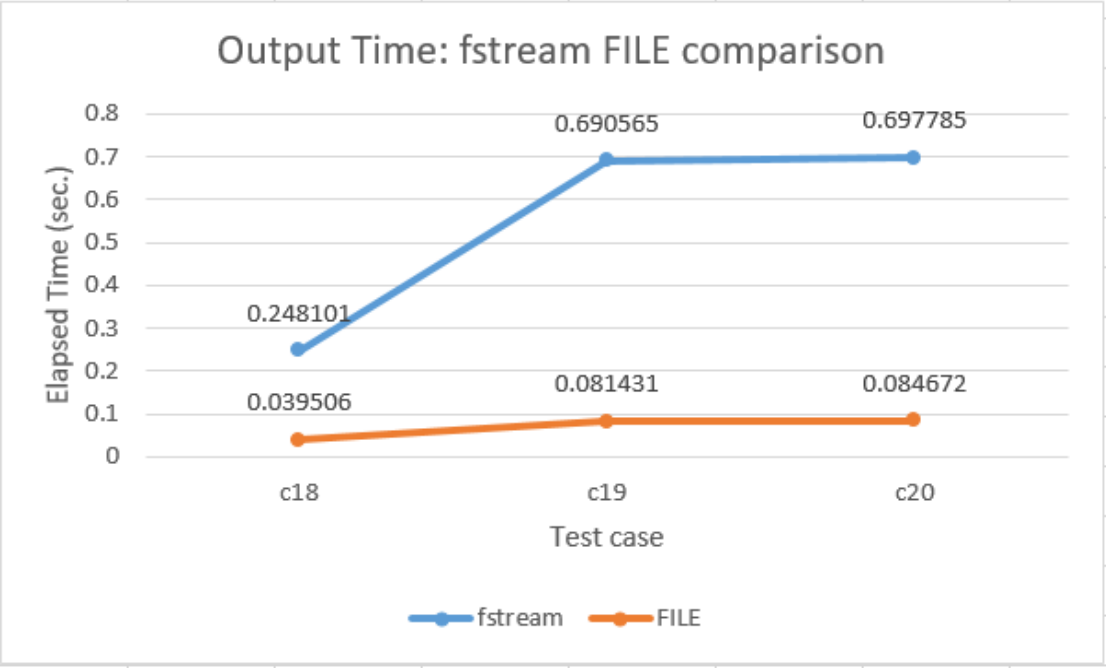
with small amount of nodes and edges (5, 10) for comparison. This is how I calculate speedup:

SpeedUp(S) = fstream file reading time / mmap file reading time

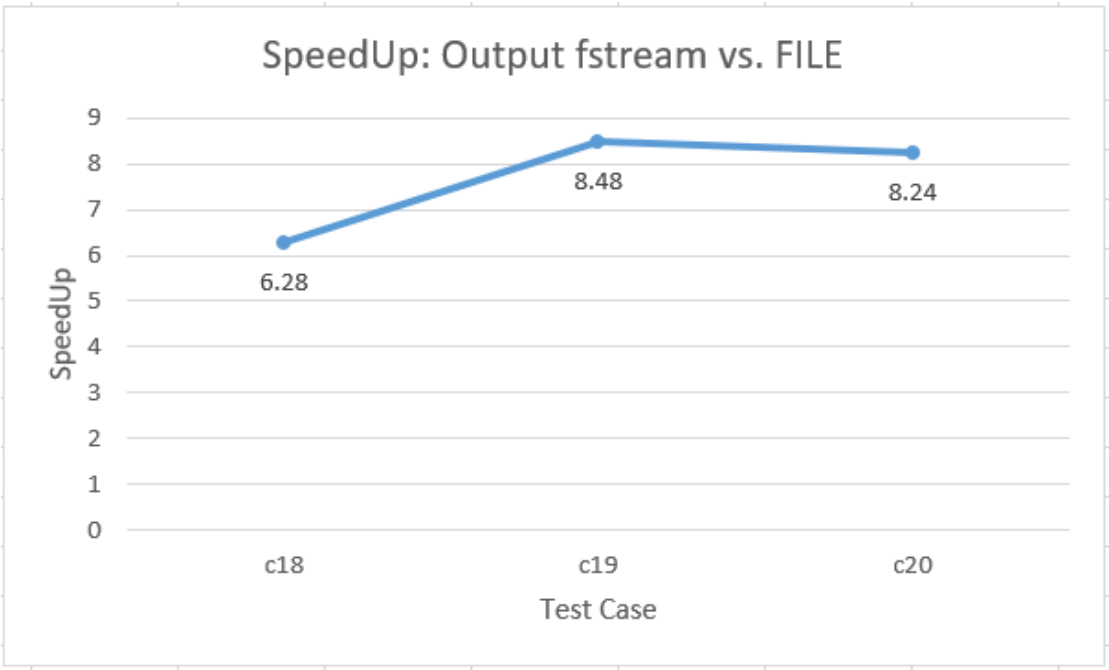
According to graph, we can tell that for small amount of edges and nodes, there is already a speedup between mmap and fstream, while for cases with large amount of data, fast file reading plays quite a major role in overall IO speedup.

- **Output Time:**

Compare to fast file reading, fast file writing is even more important since it accounts for larger proportion of the whole execution time than file reading. Again, after several trials and errors, I found that FILE's fwrite is quite faster than all the other file writing methods, even faster mmap.



▲2b.5: Output time comparison between fstream and FILE fwrite.



▲2b.6: SpeedUp graph representing file writing speedup by simply adjusting output file method from fstream to FILE fwrite.

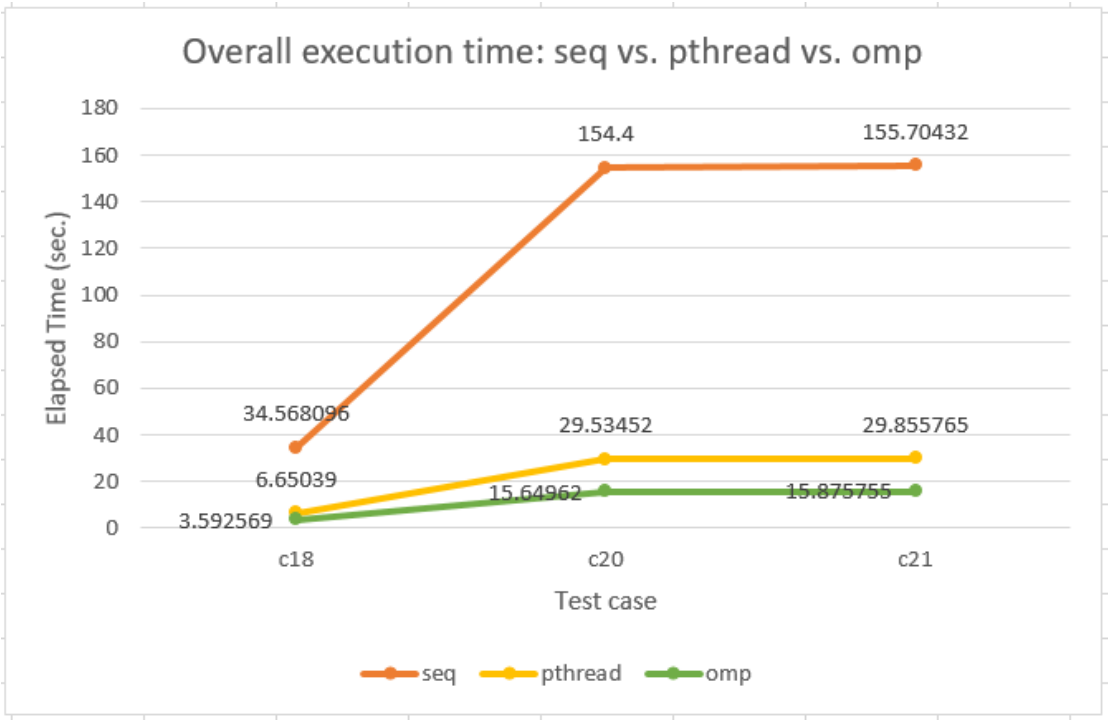
This is how I calculate speedup level:

SpeedUp(S) = file writing time by fstream / file writing time by FILE

The speedup is siginificant!

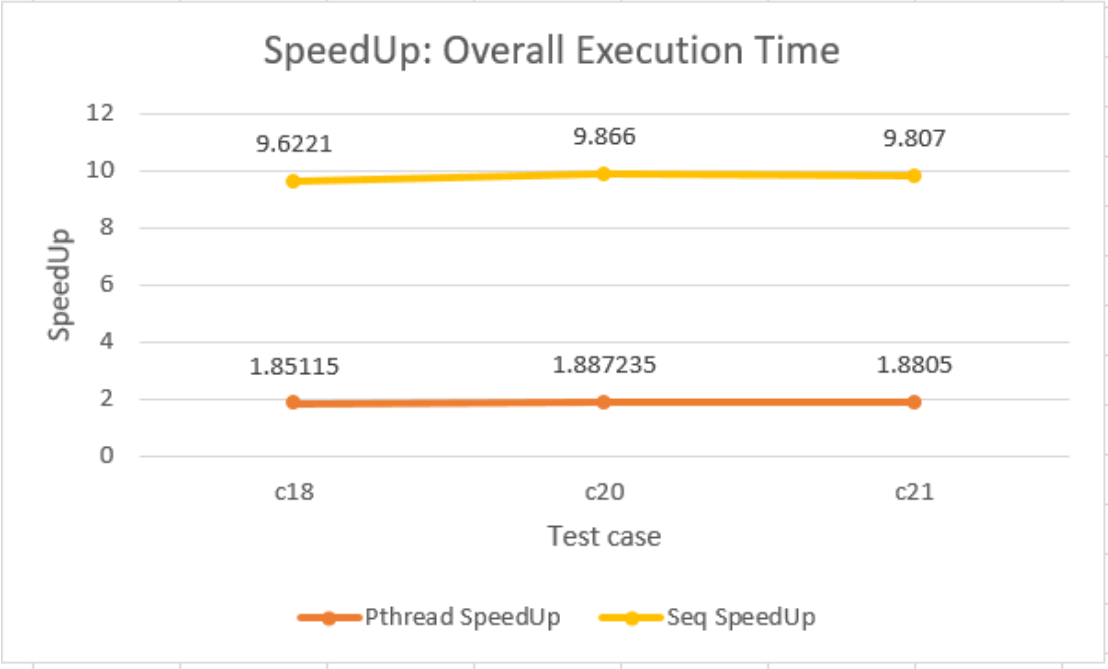
- **Overall Execution Time:**

Last part I'd like to discuss about the overall execution time. To demonstrate the execution result by OpenMP is way faster, I'd compare results from OpenMP with results by pthread and sequential code. For testing cases, still, I picked c18.1, c20.1, c21.1, denoted as c18, c20, and c21 as they contain large volume of nodes and edges.



▲2b.7: Overall execution time by sequential, pthread and openmp version.

From the graph up above, we can clearly tell that OpenMP's performance is way better than the other two methods.



▲2b.8: Overall speedup for execution time.

Graph 2b.8 represents the overall speedup for the total execution time. I compare the openmp version with pthread and sequential version respectively. The orange line denotes how openmp is faster than pthread while the yellow one stands for how openmp is faster than sequential version.

c. Time Profile

To do time profiling, I chose the time measurement function **omp_get_wtime()** supported by <omp.h> library to measure the elapsed time. The experiment environment is as listed:

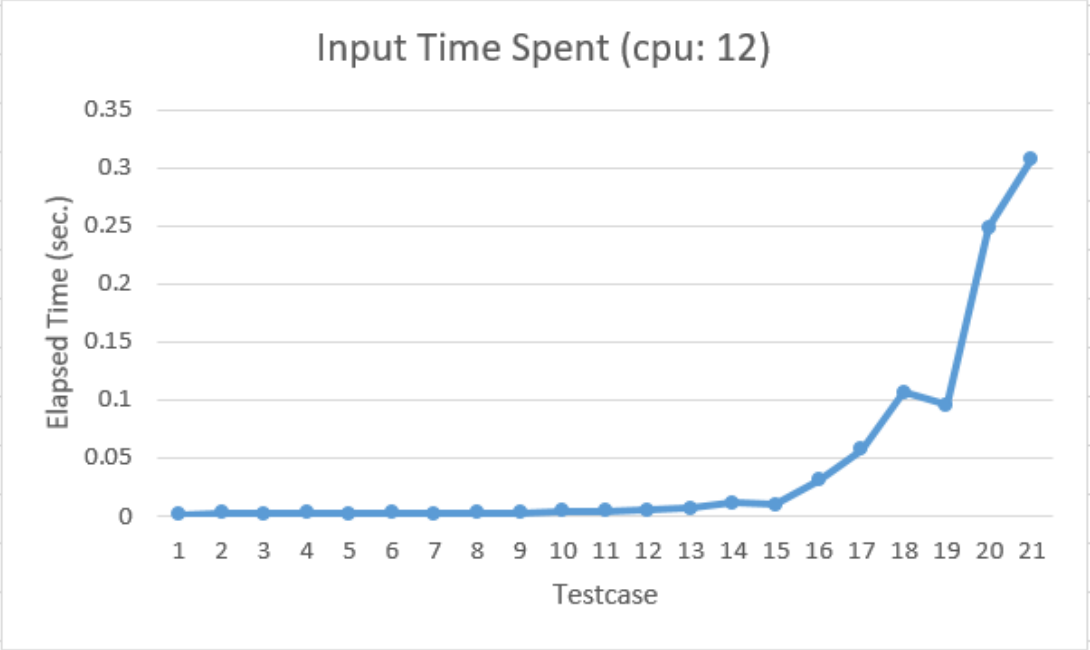
number of CPUs: 12

*testing cases: ~/pp20/share/cases/**

To measure my time, I use total 21 testing cases provided for this homework under the *~/pp20/share* folder with every testing case 12 CPUs assigned during its execution.

Input Time Measurement

First, let's see the result of input time measurement.



▲ 2c.1: Time measurement of input time that was spent for each testing case. Numbers on x-axis stand for testing cases name, 1 is c01.1, 2 is c02.2, etc.

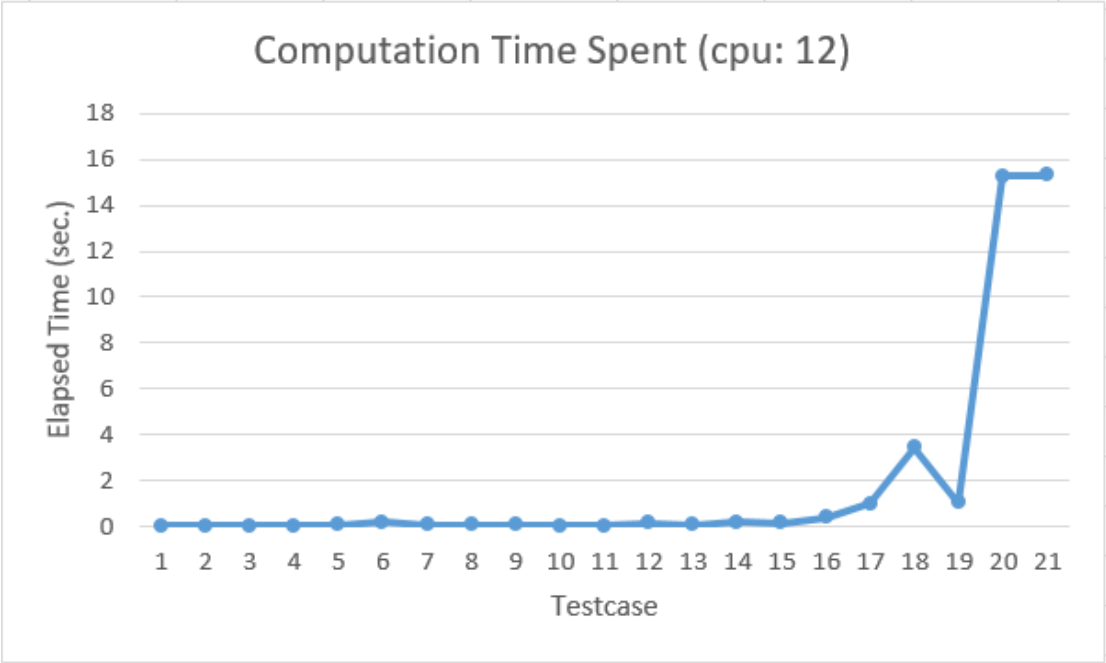
The graph up above shows the time which is spent for each testing case. Basically, the time varies along with the data volume it has to consume. For example, we can tell that for testing cases 18, 20, 21, there are significant growths on time that is spent.

If we look into the data the program was fed in, we can see that compared to those testing cases such as c01.1 or c02.2, those with small amount of nodes and edges, for testing cases c18.1(denoted in the graph as 18), c20.1, c21.1, they contain huge amount of nodes and edges, for example, in c18.1, there are 3000

nodes with 3483319 edges; and for case c20.1, the amount of nodes is 5000 with edges 7594839. Therefore, the more nodes, edges could lead to more input time spent.

Computation Time Measurement

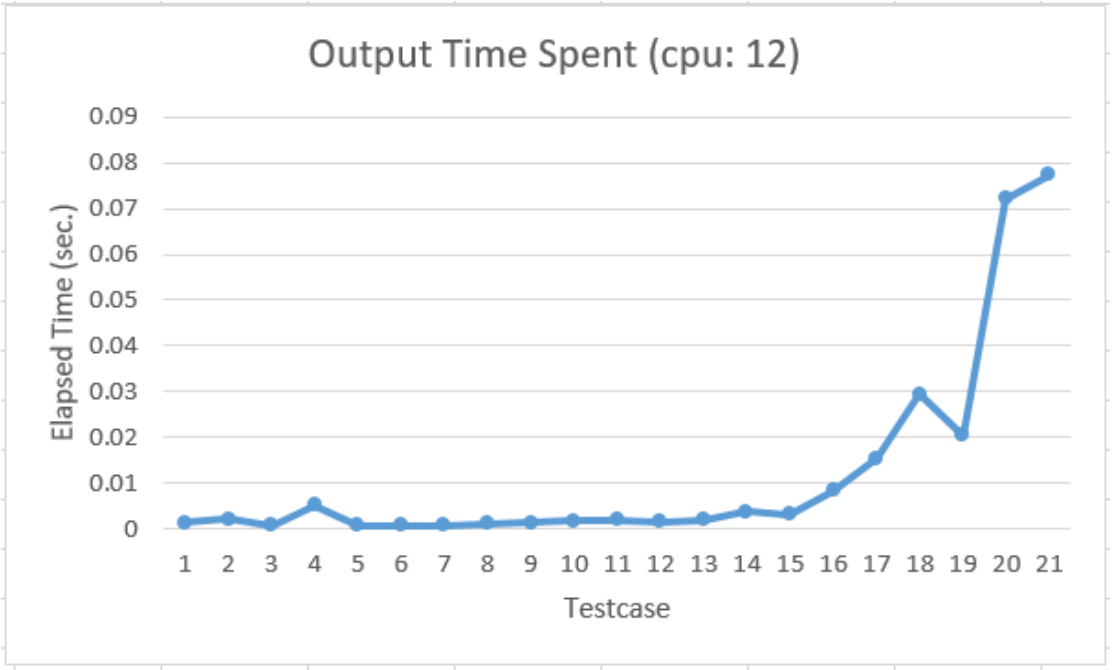
Secondly, let's see the result of computation time measurement.



▲ 2c.2: Time measurement of computation time that is spent for each testing case. Numbers on x-axis stand for testing case name.

Computation time is probably the main factor that determines the execution time for those cases with huge amount of nodes and edges. For testing cases c01.1 to c17.1 and c19.1, the amount of nodes is all under 2100, while for cases 18, 20 and 21, nodes are 3000, 5000, and 5000 respectively. So obviously, nodes and edges determine the execution time as there should be more iterations in the for loop.

Output Time Measurement



▲ 2c.3: Output time measurement.

Write file is another bottleneck for execution time. The larger the output result 2D matrix is, the longer the time for file writing would be spent. After couples of experiments, this result is the best case using C's FILE instead of <fstream>. By using FILE instead of fstream, I could lower the execution time for file writing down within 1 second. It is quite a significant progress since fstream normally take up to 1.0 second for the last two testing cases for writing files.

3. Experience & Conclusion

During the entire implementation, I felt like I'm learning the previously learnt parallelization methods once again. At first, after the sequential version, I tried to implement with Pthread, which I encountered a serious bottleneck in that thread function. Later on, after knowing calculations such as mod or division require huge computation resources and time, I did some modification on the adjacency matrix and parallelization part. Undoubtedly, there's a huge progress on the last two cases' execution time. Yet, the outcome is still not that ideal. So I started to do some optimization on IO but found that even if IO is accelerated to certain level, bottleneck is still the three nested for loops. Therefore, I tried to implement with OpenMP.

Magically, with only one statement, the execution time is wildly reduced.