

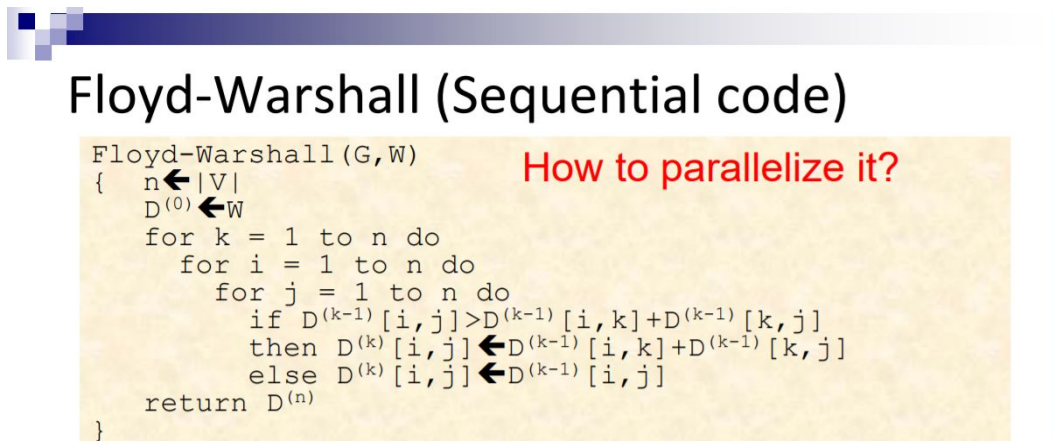
CS542200 Parallel Programming

Homework 3: Shortest Path (CPU version)

106030009 葉蓁

1. Implementation

- I use **Floyd-Warshall** algorithm, which has $O(N^3)$ time complexity in sequential implementation.
- For the Floyd-Warshall algorithm, simply implement the pseudo code in the lecture slide:

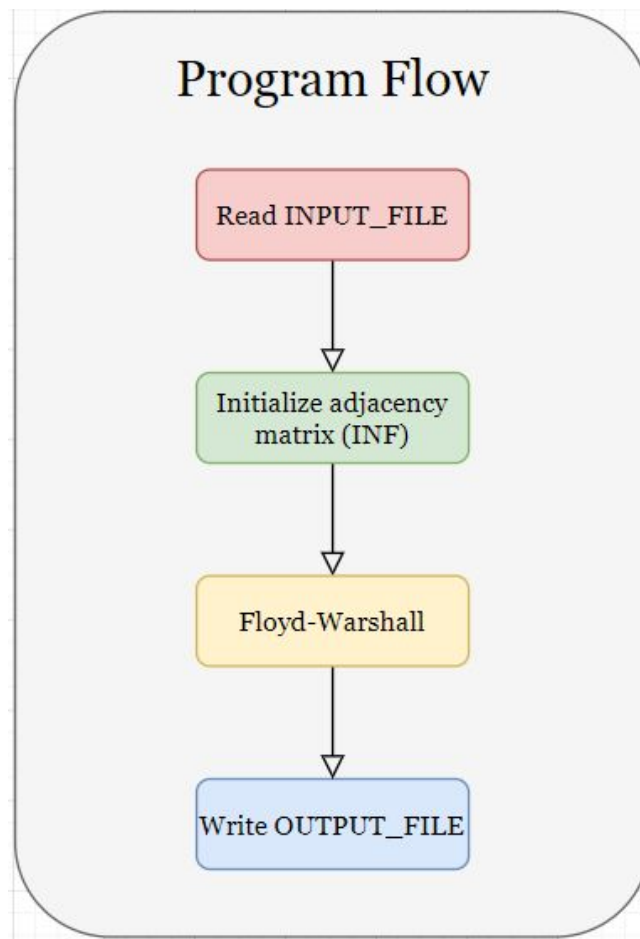


Floyd-Warshall (Sequential code)

```
Floyd-Warshall (G, W)
{
  n ← |V|
  D(0) ← W
  for k = 1 to n do
    for i = 1 to n do
      for j = 1 to n do
        if D(k-1) [i, j] > D(k-1) [i, k] + D(k-1) [k, j]
        then D(k) [i, j] ← D(k-1) [i, k] + D(k-1) [k, j]
        else D(k) [i, j] ← D(k-1) [i, j]
  return D(n)
}
```

How to parallelize it?

And use **OpenMP** to parallelize the two for-loops inside the k for-loop.



For reading input and writing output, use functions `fread()` and `fwrite()` in c library `<stdio.h>`.

I encountered several problems in my implementation. First, if I declare a two-dimensional adjacency matrix `adj[6000][6000]` in the main loop, I would get segmentation fault during srun. After moving this matrix “out of main loop” (i.e. change from local variables to global variables), the problem was solved. I guess it is because the stack of a program is not large enough to create `6000*6000*sizeof(int)` bytes.

I split my program into **3 parts: Handle input, Compute APSP, Writing output.**

(1) Handle Input

First, I use an integer array with 2 elements to store V and E (number of vertices and number of edges) in the first two integer of the input file.

```

// First read V and E.
fread(V_and_E, sizeof(int), 2, pFileRead);
#ifdef WTIME
Input_TotalTime += omp_get_wtime() - Input_starttime;
#endif
// printf("Numebr of Vertices: %d\n", V_and_E[0]);
// printf("Number of Edges: %d\n", V_and_E[1]);
num_of_vertices = V_and_E[0];
num_of_edges = V_and_E[1];

```

After knowing what V and E are, I use a dynamic allocated memory **input_buf** to store the edge information. Input_buf is an 1D array with 3*E elements. (src, dst, weight for each edge)

```

input_buf = (int*)malloc(sizeof(int)* (2 +3*num_of_edges) );
// read input file into buffer "input_buf"
// ===== NOTE !!! You haven't close the file, so start from the
// fread(input_buf, sizeof(int), BUFFER_SIZE, pFileRead );
#ifdef WTIME
Input_starttime = omp_get_wtime();|
#endif
fread(input_buf, sizeof(int), 3*(num_of_edges), pFileRead);
#ifdef WTIME
Input_TotalTime += omp_get_wtime() - Input_starttime;
#endif

```

To compute the all-pair shortest path problem, I need an adjacency matrix, so I create a dynamic allocated buffer **output_buf** and initialize it with INF (1073741823).

```

output_buf = (int*)malloc(sizeof(int)*num_of_vertices*num_of_vertices);

// ===== Intialize adjacency matrix to be INF =====

#pragma omp parallel for collapse(2) schedule(static)
for(int i=0; i< num_of_vertices; i++){
    // #pragma omp parallel for schedule(dynamic)
    for(int j=0; j< num_of_vertices; j++){
        if(i==j) output_buf[i*num_of_vertices+j] = 0;
        else output_buf[i*num_of_vertices + j] = INF;
        // if(j==print_boundary-1) std::cout<<adj_matrix[i][j]<<std::endl;
        // else std::cout<<adj_matrix[i][j];
    }
}

```

Then, fill in the correct field in output_buf according to the information from input_buf.

output_buf[src*num_of_vertices + dst] = weight

```

#pragma omp parallel for schedule(static)
for(int i=0; i < num_of_edges; i++){
    // adj_matrix[input_buf[2 + i*3]][input_buf[2 + i*3 +1]] = input_buf[2 + i*3 +2];
    // adj_matrix[input_buf[i*3]][input_buf[i*3 +1]] = input_buf[i*3 +2];
    output_buf[(input_buf[i*3])*num_of_vertices + input_buf[i*3 +1]] = input_buf[i*3 +2];
    // printf("Edge: %d src: %d dst: %d => weight: %d\n", i, input_buf[i*3], input_buf[i*3 +1], input_buf[i*3 +2]);
}
}

fclose(pFileRead);

```

(2) Compute APSP

After the output_buf is prepared, I can use the Floyd-Warshall algorithm.

Just implement the 3 for-loops and parallelize the latter 2 loops.

```

205
206     for( int k=0; k<num_of_vertices; k++){
207         #pragma omp parallel for schedule(dynamic, CHUNKSIZE) collapse(2)
208         for(int i=0; i<num_of_vertices; i++){
209             // #pragma omp parallel for schedule(dynamic)
210             for(int j=0; j<num_of_vertices; j++){
211                 if(i==j) continue;
212                 if( output_buf[i*num_of_vertices+j] > output_buf[i*num_of_vertices + k] + output_buf[k*num_of_vertices + j]){
213                     output_buf[i*num_of_vertices+j] = output_buf[i*num_of_vertices + k] + output_buf[k*num_of_vertices + j];
214                 }
215                 // else do nothing.
216             }
217         }
218     }
219 #endif
220

```

I use `#pragma omp for schedule(dynamic) collapse(2)` initially.

(3) Writing output

For writing the output answer to the output file (argv[2]), just use

```

return 1,
}else{
    fwrite(output_buf, sizeof(int), num_of_vertices*num_of_vertices, output_pFile);
    // fwrite(output_buf, sizeof(int), 100, output_pFile);
}

```

Then the answer would be written to the file output_pFile points to.

Optimization:

Initially, I cannot pass hw3-judge since it took more than 150 seconds to complete test cases c21.1 and c20.1

After checking for lots of time, I suddenly think of changing the schedule clause in `#pragma omp for`

Changing from `schedule(dynamic)` to `schedule(static)` :

從 `schedule(dynamic)` 換成 : `schedule(static)` 快了非常多！！

看來 `dynamic` scheduling internally 花的排程時間，在這個 task 中比起 `static` 去分配，花的還多。

這次的 task 不像 hw2 的 mandelbrot set, pixel之間運算時間可能差到非常多。

```
[pp20s35@apollo31 hw3]$ hw3-judge
Looking for hw3.cc: OK
Looking for Makefile: OK
Running: /usr/bin/make -C /home/pp20/pp20s35/.judge.157386432 hw3
make: Entering directory '/home/pp20/pp20s35/.judge.157386432'
g++ -O3 -lm -pthread -fopenmp -pthread -fopenmp hw3.cc -o hw3
make: Leaving directory '/home/pp20/pp20s35/.judge.157386432'
c02.1 0.12 accepted
c04.1 0.17 accepted
c01.1 0.22 accepted
c03.1 0.12 accepted
c06.1 0.17 accepted
c05.1 0.17 accepted
c09.1 0.12 accepted
c10.1 0.17 accepted
c07.1 0.12 accepted
c08.1 0.22 accepted
c13.1 0.37 accepted
c14.1 0.82 accepted
c12.1 0.27 accepted
c11.1 0.27 accepted
c15.1 0.62 accepted
c16.1 1.82 accepted
c17.1 3.82 accepted
c19.1 3.88 accepted
c18.1 8.79 accepted
c20.1 32.75 accepted
c21.1 32.90 accepted
Removing temporary directory /home/pp20/pp20s35/.judge.157386432
Scoreboard: updated {19 68.09} --> {21 87.87}
[pp20s35@apollo31 hw3]$
```

The running time of c20 and c21 become **faster by a factor of 5!**

接著繼續實驗：用 dynamic 但 chunk size 設大一點 (e.x.: 100)
schedule(dynamic, CHUNKSIZE=100)

單測 c21.1:

```
[pp20s35@apollo31 hw3]$ srun -n1 -c12 ./hw3 cases/c21.1 c21.1.out
Numbr of Vertices: 5000
Number of Edges: 10723117
Took 136.38969923 on Computation
Done writing output file.
Took 0.24261808 on Output
Work took 136.90419101 seconds
```

hw3-judge結果：

```
c13.1 0.32 accepted
c16.1 2.12 accepted
c17.1 5.48 accepted
c18.1 15.25 accepted
c19.1 5.53 accepted
c20.1 68.87 accepted
c21.1 69.32 accepted
Removing temporary directory /home/pp20/pp20s35/.judge.760552172
Scoreboard: not updating {21 87.87} -x-> {21 170.46}
[pp20s35@apollo31 hw3]$
```

效果還是比 static慢

把初始化 adjacency matrix 和 填寫adjacency matrix (根據INPUT_FILE的edge weight) 的for-loop部分改回static:

```
3688'
c02.1    0.12    accepted
c04.1    0.17    accepted
c03.1    0.07    accepted
c01.1    0.07    accepted
c06.1    0.17    accepted
c05.1    0.12    accepted
c07.1    0.17    accepted
c08.1    0.12    accepted
c10.1    0.32    accepted
c11.1    0.32    accepted
c09.1    0.17    accepted
c12.1    0.22    accepted
c14.1    0.82    accepted
c13.1    0.32    accepted
c15.1    0.57    accepted
c16.1    2.22    accepted
c17.1    5.43    accepted
c18.1    15.65   accepted
c19.1    5.48    accepted
c20.1    68.57   accepted
c21.1    68.72   accepted
Removing temporary directory /home/pp20/pp20s35/.judge.9
32133688
Scoreboard: not updating {21 87.87} -x-> {21 169.77}
[pp20s35@apollo31 hw3]$
```

沒什麼影響，看來 bottleneck不是這裡

嘗試更大的chunksize: 500

```

make: Leaving directory '/home/pp20/pp20s35/.judge.631966
636'
c04.1    0.07    accepted
c01.1    0.12    accepted
c02.1    0.07    accepted
c03.1    0.07    accepted
c07.1    0.17    accepted
c08.1    0.12    accepted
c06.1    0.07    accepted
c05.1    0.12    accepted
c09.1    0.12    accepted
c12.1    0.17    accepted
c13.1    0.17    accepted
c14.1    0.52    accepted
c15.1    0.42    accepted
c10.1    0.17    accepted
c11.1    0.17    accepted
c16.1    1.17    accepted
c17.1    2.92    accepted
c19.1    2.97    accepted
c18.1    7.98    accepted
c20.1    33.75    accepted
c21.1    34.38    accepted
Removing temporary directory /home/pp20/pp20s35/.judge.63
1966636
Scoreboard: updated {21 87.87} --> {21 85.68}
[pp20s35@apollo31 hw3]$ 

```

大部分的測資都加速了～唯獨最後兩筆稍微變慢
 比較 static 和 (dynamic, 500) 的結果：

6432'			c04.1	0.07	accepted
c02.1	0.12	accepted	c01.1	0.12	accepted
c04.1	0.17	accepted	c02.1	0.07	accepted
c01.1	0.22	accepted	c03.1	0.07	accepted
c03.1	0.12	accepted	c07.1	0.17	accepted
c06.1	0.17	accepted	c08.1	0.12	accepted
c05.1	0.17	accepted	c06.1	0.07	accepted
c09.1	0.12	accepted	c05.1	0.12	accepted
c10.1	0.17	accepted	c09.1	0.12	accepted
c07.1	0.12	accepted	c12.1	0.17	accepted
c08.1	0.22	accepted	c13.1	0.17	accepted
c13.1	0.37	accepted	c14.1	0.52	accepted
c14.1	0.82	accepted	c15.1	0.42	accepted
c12.1	0.27	accepted	c10.1	0.17	accepted
c11.1	0.27	accepted	c11.1	0.17	accepted
c15.1	0.62	accepted	c16.1	1.17	accepted
c16.1	1.82	accepted	c17.1	2.92	accepted
c17.1	3.82	accepted	c19.1	2.97	accepted
c19.1	3.88	accepted	c18.1	7.98	accepted
c18.1	8.79	accepted	c20.1	33.75	accepted
c20.1	32.75	accepted	c21.1	34.38	accepted
c21.1	32.90	accepted			
Removing temporary directory /home/pp20/pp20s35/.judge.1			Removing temporary directory /home/pp20/pp20s35/		
57386432			1966636		
Scoreboard: updated {19 68.09} --> {21 87.87}			Scoreboard: updated {21 87.87} --> {21 85.68}		
[pp20s35@apollo31 hw3]\$			[pp20s35@apollo31 hw3]\$		

繼續將 CHUNKSIZE 變大： 1500

又進步了快10秒！(從85.68 秒 -> 76.96 秒)

```
Running: /usr/bin/make -C /home/pp20/pp20s35/.judge.65057
8559 hw3
make: Entering directory '/home/pp20/pp20s35/.judge.65057
8559'
g++ -O3 -lm -pthread -fopenmp -pthread -fopenmp hw3.
cc -o hw3
make: Leaving directory '/home/pp20/pp20s35/.judge.650578
559'
c03.1      0.12      accepted
c04.1      0.22      accepted
c02.1      0.17      accepted
c01.1      0.17      accepted
c06.1      0.17      accepted
c05.1      0.22      accepted
c07.1      0.12      accepted
c08.1      0.12      accepted
c10.1      0.27      accepted
c09.1      0.22      accepted
c11.1      0.17      accepted
c12.1      0.17      accepted
c13.1      0.22      accepted
c15.1      0.42      accepted
c14.1      0.47      accepted
c16.1      1.12      accepted
c17.1      2.67      accepted
c18.1      7.08      accepted
c19.1      2.72      accepted
c20.1      29.91     accepted
c21.1      30.25     accepted
Removing temporary directory /home/pp20/pp20s35/.judge.65
0578559
Scoreboard: updated {21 85.68} --> {21 76.96}
[pp20s35@apollo31 hw3]$
```


CHUNK SIZE = 2500 :

```
make: Leaving directory '/home/pp20/pp20s35/.judge.713135368'
c04.1    0.12    accepted
c03.1    0.17    accepted
c02.1    0.17    accepted
c01.1    0.12    accepted
c07.1    0.12    accepted
c06.1    0.17    accepted
c08.1    0.17    accepted
c05.1    0.17    accepted
c11.1    0.22    accepted
c10.1    0.22    accepted
c12.1    0.22    accepted
c09.1    0.12    accepted
c13.1    0.22    accepted
c14.1    0.52    accepted
c15.1    0.32    accepted
c16.1    1.07    accepted
c17.1    2.67    accepted
c18.1    6.98    accepted
c19.1    2.67    accepted
c20.1    29.01    accepted
c21.1    29.47    accepted
Removing temporary directory /home/pp20/pp20s35/.judge.713135368
Scoreboard: updated {21 76.96} --> {21 74.87}
[pp20s35@apollo31 hw3]$
```

由以上實驗可知：現在的 bottleneck 是：最後兩筆測資。

先看看兩筆測資的資訊：

```
[pp20s35@apollo31 hw3]$ srun -n1 -c12 ./hw3 cases/c20.1 c20.1.out
Numebr of Vertices: 5000
Number of Edges: 7594839
Tooks 29.04499370 on Computation
Done writing output file.
Tooks 0.22905549 on Output
Work took 29.39635126 seconds
[pp20s35@apollo31 hw3]$
```

c20.1 有5000個vertices, 7594839條edges。

所以 APSP 用 Floyd-Warshall 演算法：要計算 5000^3 次方

其中 5000×5000 可以平行給 threads

嘗試將CHUNKSIZE 設的更大：6000

```
[pp20s35@apollo31 hw3]$ srun -n1 -c12 ./hw3 cases/c20.1 c20.1.out
Numebr of Vertices: 5000
Number of Edges: 7594839
Tooks 29.83010708 on Computation
Done writing output file.
Tooks 0.22176905 on Output
Work took 30.19789804 seconds
[pp20s35@apollo31 hw3]$ make clean
```

反而變慢，改用跟 # vertices 一樣：CHUNK SIZE = 5000

先單測 c20.1

```
[pp20s35@apollo31 hw3]$ srun -n1 -c12 ./hw3 cases/c20.1 c20.1.out
Numebr of Vertices: 5000
Number of Edges: 7594839
Tooks 28.90041659 on Computation
Done writing output file.
Tooks 0.21925567 on Output
Work took 29.24704043 seconds
[pp20s35@apollo31 hw3]$
```

單測 c21.1 的測資：

```
[pp20s35@apollo31 hw3]$ srun -n1 -c12 ./hw3 cases/c21.1 c21.1.out
Numebr of Vertices: 5000
Number of Edges: 10723117
Tooks 29.11975848 on Computation
Done writing output file.
Tooks 0.21507691 on Output
Work took 29.50744024 seconds
```

改用 chunk size == 5000: 測 c21.1

```
[pp20s35@apollo31 hw3]$ srun -n1 -c12 ./hw3 cases/c21.1 c21.1.out
Numebr of Vertices: 5000
Number of Edges: 10723117
Tooks 29.26930933 on Computation
Done writing output file.
Tooks 0.21334663 on Output
Work took 29.66429206 seconds
```

竟然比 用6000 變慢0.1秒呀～

直接挑戰 hw3-judge:

```

make: Leaving directory '/home/pp20/pp20s35/.judge.514554740'
c03.1    0.17    accepted
c04.1    0.17    accepted
c02.1    0.12    accepted
c01.1    0.17    accepted
c08.1    0.12    accepted
c05.1    0.17    accepted
c06.1    0.17    accepted
c07.1    0.12    accepted
c09.1    0.22    accepted
c12.1    0.17    accepted
c10.1    0.17    accepted
c11.1    0.17    accepted
c13.1    0.22    accepted
c14.1    0.47    accepted
c15.1    0.27    accepted
c16.1    1.12    accepted
c17.1    2.52    accepted
c18.1    7.08    accepted
c19.1    2.67    accepted
c20.1    29.45    accepted
c21.1    29.67    accepted
Removing temporary directory /home/pp20/pp20s35/.judge.514554740
Scoreboard: not updating {21 74.87} -x-> {21 75.36}
[pp20s35@apollo31 hw3]$ 

```

竟然變慢 XD

看來 tune CHUNKSIZE 的優化差不多到極限了，把 CHUNKSIZE 設越大，越接近 static，但還能比 static 快，就是設在一個剛好的位置，讓 Thread 可以有一點自由度 (先執行完的 Thread 可以去拿下一個 task)

11/28 update:

```

c11.1    0.37    accepted
c13.1    0.27    accepted
c12.1    0.37    accepted
c15.1    0.22    accepted
c16.1    0.92    accepted
c10.1    0.17    accepted
c14.1    0.37    accepted
c17.1    2.17    accepted
c18.1    5.88    accepted
c19.1    2.07    accepted
c20.1    25.87    accepted
c21.1    25.02    accepted
Removing temporary directory /home/pp20/pp20s35/.judge.803959192
Scoreboard: updated {21 71.46} --> {21 65.19}
[pp20s35@apollo31 hw3]$ make clean && make

```

不知為何，不確定中間修改了什麼 (因為一直改)
把全部的 schedule 換成 static，突然加速到 65 了！


```

c12.1    0.37    accepted
c10.1    0.17    accepted
c11.1    0.17    accepted
c15.1    0.42    accepted
c14.1    0.52    accepted
c13.1    0.17    accepted
c16.1    0.82    accepted
c18.1    5.33    accepted
c17.1    2.12    accepted
c19.1    2.07    accepted
c20.1    21.81   accepted
c21.1    21.87   accepted
Removing temporary directory /home/pp20/pp20s35/.judge.089052396
Scoreboard: updated {21 65.19} --> {21 57.46}
[pp20s35@apollo31 hw3]$ make clean && make

```

把 APSP 部份的 collapse(2) 拿掉，換成兩層for各別寫 #pragma omp parallel for schedule(static) 加速到 57 !?

```

c03.1    0.22    accepted
c01.1    0.17    accepted
c04.1    0.07    accepted
c02.1    0.17    accepted
c05.1    0.27    accepted
c06.1    0.22    accepted
c08.1    0.12    accepted
c07.1    0.27    accepted
c09.1    0.32    accepted
c10.1    0.27    accepted
c11.1    0.17    accepted
c12.1    0.27    accepted
c13.1    0.27    accepted
c14.1    0.57    accepted
c15.1    0.27    accepted
c16.1    0.87    accepted
c17.1    1.97    accepted
c18.1    5.33    accepted
c19.1    1.97    accepted
c20.1    21.77   accepted
c21.1    21.82   accepted
Removing temporary directory /home/pp20/pp20s35/.judge.719141354
Scoreboard: updated {21 57.46} --> {21 57.32}
[pp20s35@apollo31 hw3]$ 

```

把初始化 matrix 用的 collapse 也拿掉，加速些微 XD

把 if(i==j) continue;拿掉：快超級多！

從 57.32 => 47.11 秒

其中的原因百思不得其解，猜測是因為使用branch (if)指令，pipeline cpu 設計會想要先猜會跳到if後面，但如果branch結果錯誤，就必須跳回來反而多花時間。


```

make: Entering directory '/home/pp20/pp20s35/.judge.017528962'
g++ -O3 -lm -pthread -fopenmp hw3.cc -o hw3
make: Leaving directory '/home/pp20/pp20s35/.judge.017528962'
c03.1 0.17 accepted
c01.1 0.12 accepted
c04.1 0.12 accepted
c02.1 0.22 accepted
c07.1 0.27 accepted
c06.1 0.22 accepted
c08.1 0.12 accepted
c05.1 0.07 accepted
c09.1 0.37 accepted
c11.1 0.32 accepted
c12.1 0.27 accepted
c10.1 0.17 accepted
c13.1 0.42 accepted
c15.1 0.47 accepted
c14.1 0.47 accepted
c16.1 0.77 accepted
c19.1 1.72 accepted
c18.1 4.44 accepted
c17.1 1.62 accepted
c20.1 16.96 accepted
c21.1 17.86 accepted
Removing temporary directory /home/pp20/pp20s35/.judge.017528962
Scoreboard: updated {21 57.32} --> {21 47.11}
[pp20s35@panel031 ~]$

```

最終程式碼如下：

```

154     for( int k=0; k<num_of_vertices; k++){
155         // #pragma omp parallel for schedule(static) collapse(2)
156         #pragma omp parallel for schedule(static)
157         for(int i=0; i<num_of_vertices; i++){
158             #pragma omp parallel for schedule(static)
159             for(int j=0; j<num_of_vertices; j++){
160                 // if(i==j) continue;
161                 if( output_buf[i*num_of_vertices+j] > output_buf[i*num_of_vertices + k] + output_buf[k*num_of_vertices + j]){
162                     output_buf[i*num_of_vertices+j] = output_buf[i*num_of_vertices + k] + output_buf[k*num_of_vertices + j];
163                 }
164                 // else do nothing.
165             }
166         }
167     }
168 #endif

```

幾個重點：

- (1) 把 schedule(dynamic, chunksize) 換成 schedule(static)
- (2) 把 collapse(2) 拿掉，自己在兩層for都各寫一次
- (3) 把 if(i==j) continue; 的判斷拿掉，快10秒！（超多）

從原本 ranking 30, 71 秒多

進步到 ranking 12, 47.11秒！

超感動啦！

pp20s35	12	21	47.11
---------	----	----	-------

關注bottleneck的兩筆測資（20, 21）：附圖是17~21 測資的時間(sec)

1.62	4.44	1.72	16.96	17.86
------	------	------	-------	-------

時間也縮小到 16, 17秒！（原本是29~30秒左右，幾乎快兩倍）

c. What is the time complexity of your algorithm, in terms of number of vertices V , number of edges E , number of CPUs P ?

(1) For reading input file: $O(2+3 \cdot E)$ -> 後來得知不需要考慮 I/O time

(2) Initialize adjacency matrix : $O(V^2 / P)$

(3) Memory copy from input_buf to output_buf (fill in the adjacency matrix) : $O(E / P)$

(4) Parallelize Floyd-Warshall algorithm: $O(V \cdot (V^2 / P))$

(5) Writing output file: $O(V^2)$ -> 後來得知不需要考慮 I/O time

Total time complexity : $O(V^2 / P) + O(E / P) + O(V \cdot (V^2 / P))$

$= O((V^2 + E + V^3) / P) = \mathbf{O((V^3 + E) / P)}$

其中 $(V^3 + E) / P$ 是可以利用 Thread 平行之處；

$O(E + V^2)$ 是 fread() / fwrite() 的時間(因為 sequentially 所以沒有平行)

但實際測出：花最多時間的還是 Computation 部分： $O((V^3 + E) / P)$

p.s. 黃色處：後來得知不需要考慮 I/O time

d. How did you design & generate your test case?

How do I verify the answer is correct?

First, when generating input files, I run the “sequential version of Floyd-Warshall algorithm” simultaneously to solve the All-pairs shortest path problem, and output a sample answer at my_testcases/inputXX.out. For example:

**./gen_input 500 1000 my_testcases/input02.in
my_testcases/input02.out**

The sample answer “input 02.out” would be generated.

```
[pp20s35@apollo31 hw3]$ ./gen_input 500 1000 my_testcases/input02.in my_testcases/input02.out
v: 500
e: 1000
Numebr of Vertices: 500
Number of Edges: 1000
Tooks 0.21985360 on Computation
Done writing output file.
Tooks 0.00048145 on Output
Work took 0.22084679 seconds
```

Second, I run my parallel version (hw3.o) to solve my_testcases/input02.in

```
[pp20s35@apollo31 hw3]$ srun -c12 ./hw3 my_testcases/input02.in input02.out
Numebr of Vertices: 500
Number of Edges: 1000
Tooks 0.12963173 on Computation
Done writing output file.
Tooks 0.00426003 on Output
Work took 0.14564406 seconds
```

Finally, I compare the two file using ./Verification_two_files

./Verification_twoFiles my_testcases/input02.out input02.out

The result is below:

```
[pp20s35@apollo31 hw3]$ ./Verification_twoFiles my_testcases/input02.out input02.out
Files are equal
[pp20s35@apollo31 hw3]$
```

2. Experiments

a. System Spec

I use *apollo* server as my computation platform.

b. Strong Scalability

Strong Scalability (v.s. sequential time)

```
[pp20s35@apollo31 hw3]$ sbatch hw3
hw3      hw3.cc      hw3exp.sh  hw3_seq.cc
[pp20s35@apollo31 hw3]$ sbatch hw3exp.sh
Submitted batch job 2718938
[pp20s35@apollo31 hw3]$ sbatch hw3exp.sh
Submitted batch job 2718943
[pp20s35@apollo31 hw3]$ sbatch hw3exp.sh
Submitted batch job 2718945
[pp20s35@apollo31 hw3]$ sbatch hw3exp.sh
Submitted batch job 2718947
[pp20s35@apollo31 hw3]$ sbatch hw3exp.sh
Submitted batch job 2718954
[pp20s35@apollo31 hw3]$ sbatch hw3exp.sh
Submitted batch job 2718955
[pp20s35@apollo31 hw3]$ sbatch hw3exp.sh
Submitted batch job 2718957
[pp20s35@apollo31 hw3]$ ./Verification_twoFiles c21_12.out cases/c21.1.out
Files are equal
[pp20s35@apollo31 hw3]$ ./Verification_twoFiles c21_8.out cases/c21.1.out
Files are equal
[pp20s35@apollo31 hw3]$ ./Verification_twoFiles c21_10.out cases/c21.1.out
Files are equal
[pp20s35@apollo31 hw3]$ ./Verification_twoFiles c21_6.out cases/c21.1.out
Files are equal
[pp20s35@apollo31 hw3]$ ./Verification_twoFiles c21_4.out cases/c21.1.out
Files are equal
[pp20s35@apollo31 hw3]$ ./Verification_twoFiles c21_2.out cases/c21.1.out
Files are equal
[pp20s35@apollo31 hw3]$ ./Verification_twoFiles c21_1.out cases/c21.1.out
Files are equal
```

First, write a script file (hw3exp.sh) and use sbatch to hand in my jobs with different numbers of threads (-cX) to apollo platform.

Then, after the jobs are completed, use ./Verification_twoFiles to make sure the answers are corrected.

(The same as the sample answer provided by TAs.)

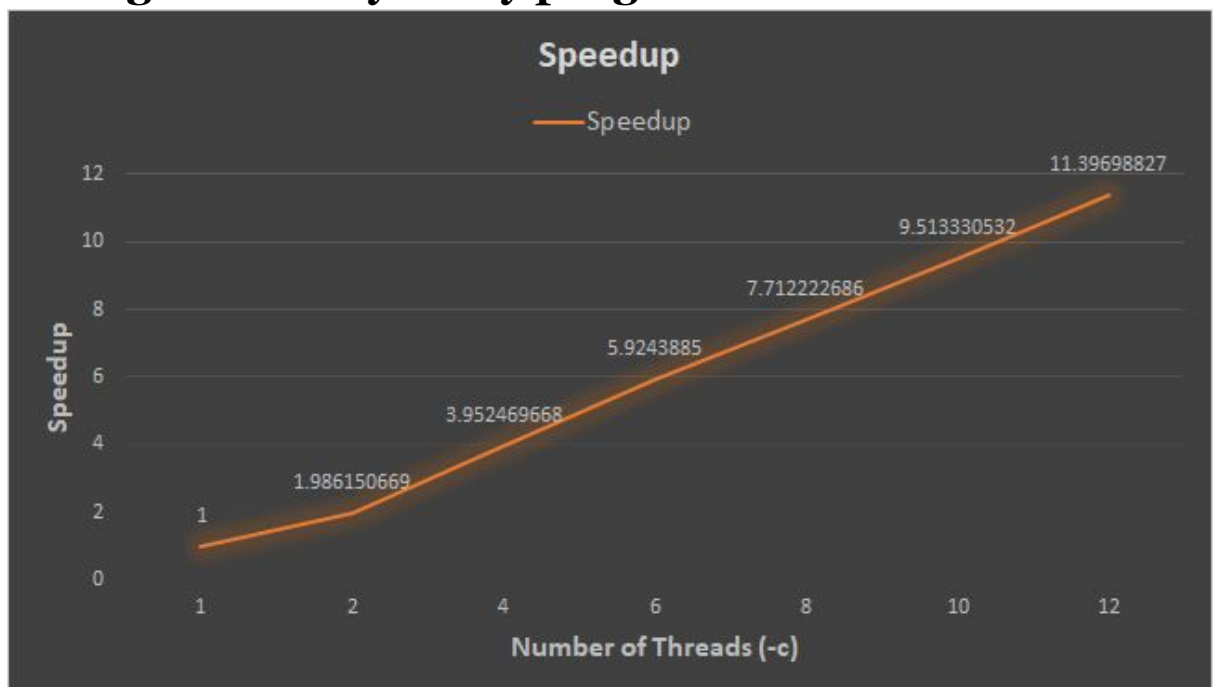
Finally, their running time and other information can be checked in the output files from slurm-xxxxxxx.out.

For example, see the case with 12 threads:

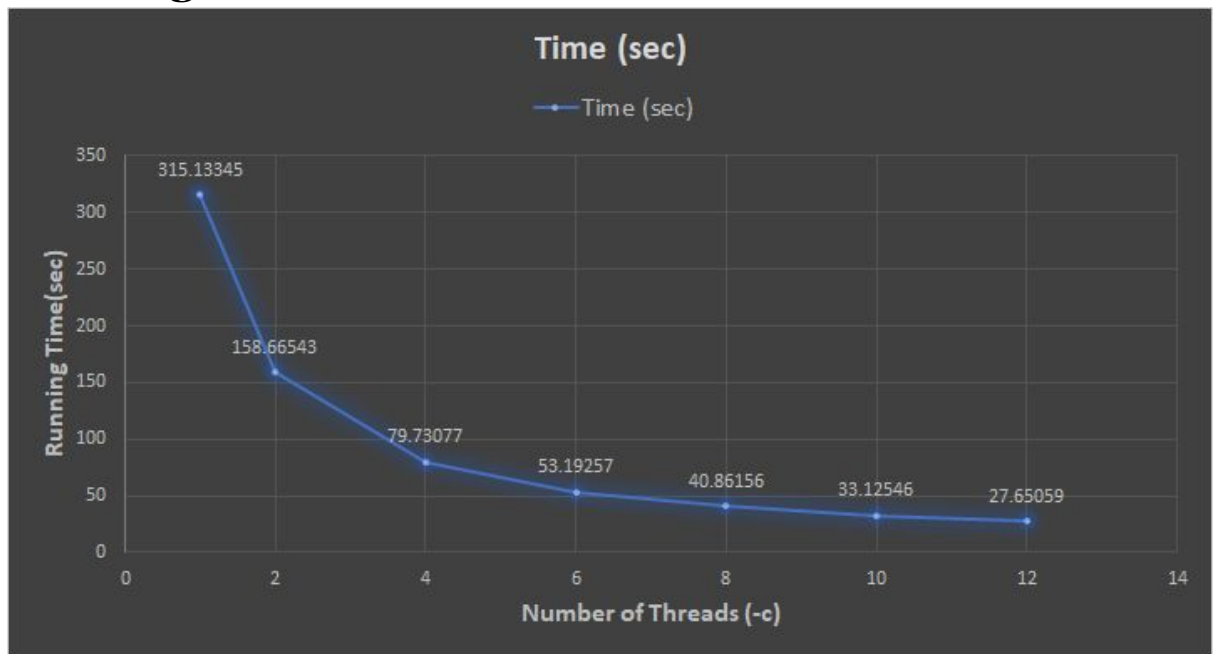
```
Welcome  exp.sh  hw3exp.sh  slurm-2718938.out X
hw3 > slurm-2718938.out
1  Numebr of Vertices: 5000
2  Number of Edges: 10723117
3  Took 27.31931710 on Computation
4  Done writing output file.
5  Took 0.18692449 on Output
6  Work took 27.65059176 seconds
7
```

As you can see, it took about 27.65 seconds to finish job 2718938. Furthermore, the computation time of solving the APSP problem cost about 27.32 (98.8% of running time)

Strong Scalability of my program:



Running Time:



Time Table :

N	n	Speedup
1	1	
Number of Threads (-c)	Time (sec)	Speedup
1	315.13345	1
2	158.66543	1.986150669
4	79.73077	3.952469668
6	53.19257	5.9243885
8	40.86156	7.712222686
10	33.12546	9.513330532
12	27.65059	11.39698827

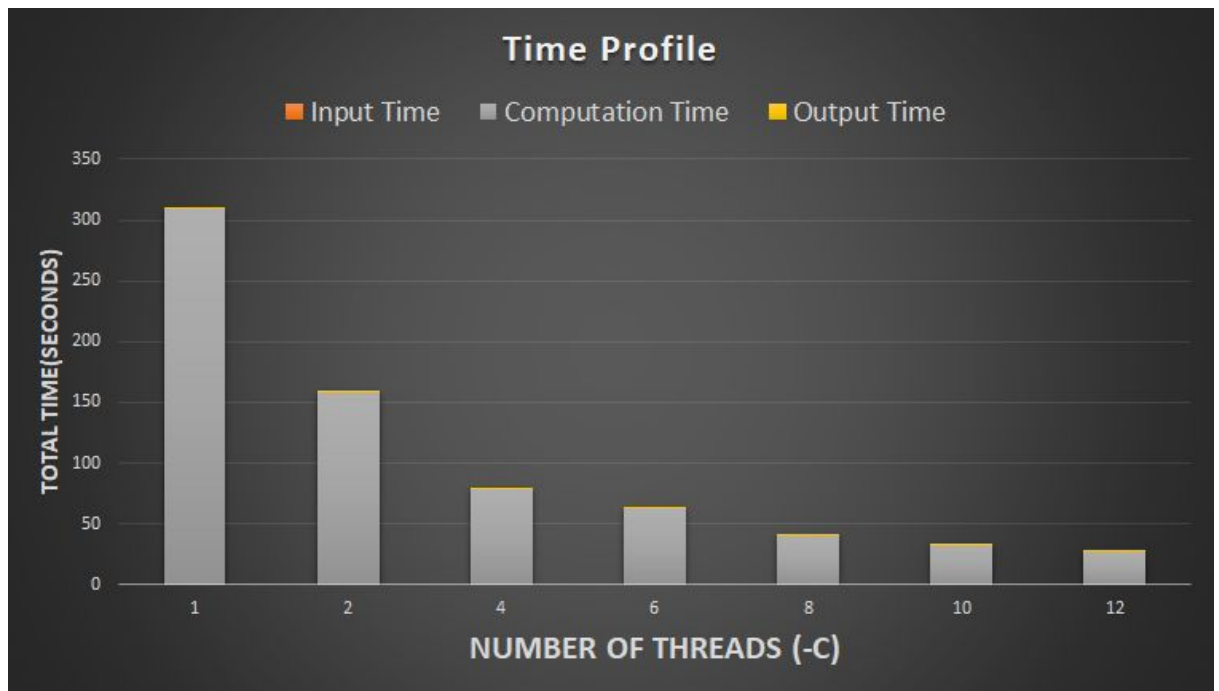
c. Time Profile

(Input / Computation / Output time and ratio)

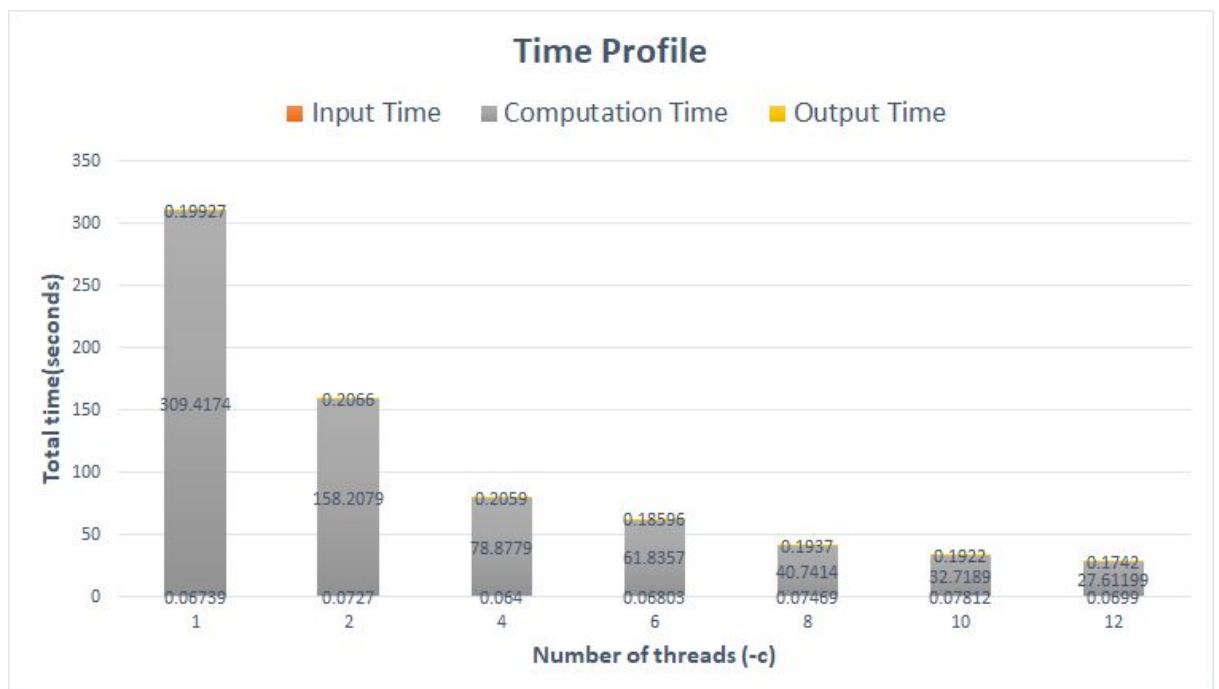
```
hw3 > cat slurm-2721498.out
1 Input Time 0.06994492
2 Output Time 0.17415346
3 Computation Time 27.61198566 seconds
4 Total Time 27.85608404 seconds
5
```

Table:

Time Profile	Total Time	Input Time	Computation Time	Output Time
Number of Threads (-c)	sec	sec	sec	sec
1	309.68403	0.06739	309.4174	0.19927
2	158.4872	0.0727	158.2079	0.2066
4	79.1478	0.064	78.8779	0.2059
6	62.0897	0.06803	61.8357	0.18596
8	41.0098	0.07469	40.7414	0.1937
10	32.9892	0.07812	32.7189	0.1922
12	27.8561	0.0699	27.61199	0.1742

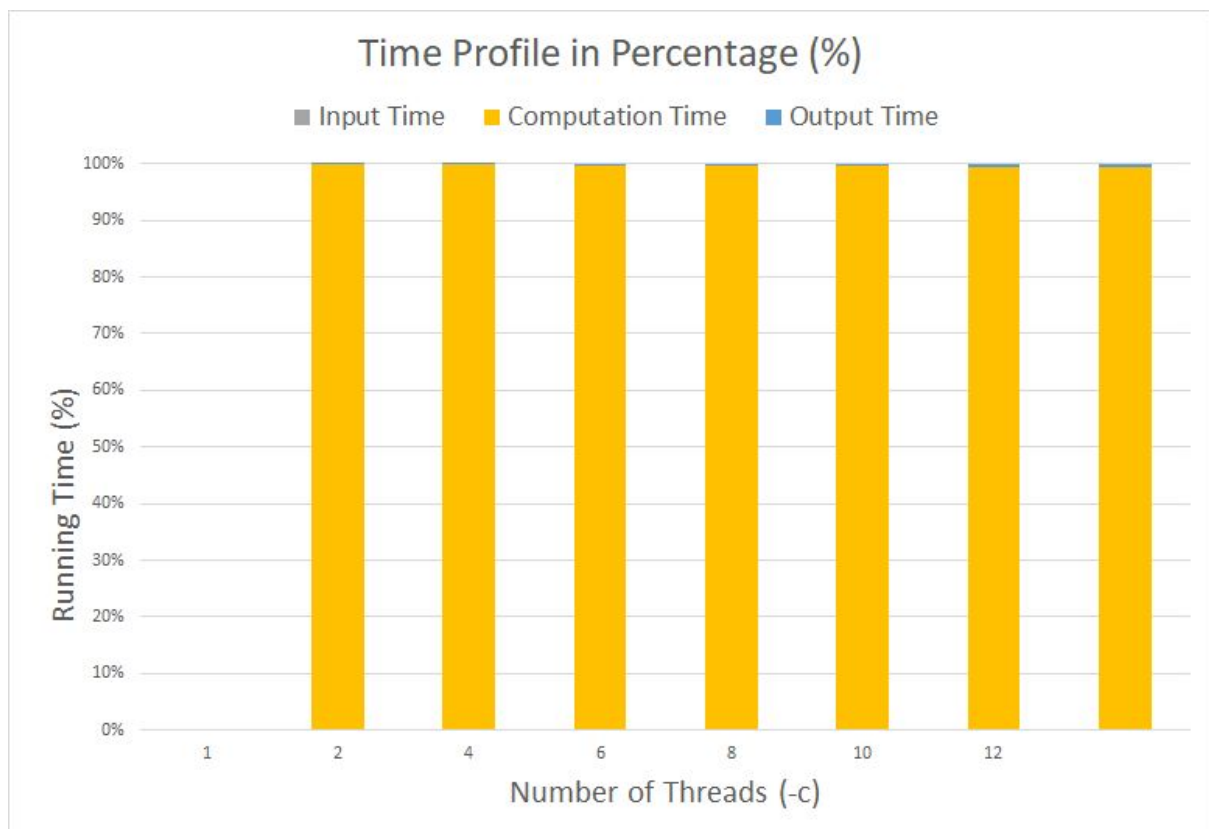


As you can see, since about **99% running time is for Computation**, we can hardly see Input Time and Output Time. Below is the diagram with “Number of Time(sec)” on each bar.



We can see the table and figure in percentage (%) :

Time Profile	Total Time	Input Time	Computation Time	Output Time
Number of Threads (-c)	sec	%	%	%
1	309.68403	0.02176089	99.91390257	0.064346231
2	158.4872	0.045871212	99.82377126	0.13035753
4	79.1478	0.080861376	99.65899242	0.260146207
6	62.0897	0.109567287	99.59091444	0.299502172
8	41.0098	0.182127199	99.34552229	0.472326127
10	32.9892	0.236804772	99.18064094	0.582614916
12	27.8561	0.250932471	99.12367489	0.625356744



We can see very little blue and gray parts at the top and bottom of each bar. For the table above you can observe that the Computation section took about 99% of the total running time.

The **bottleneck** is obviously the **computation** of the APSP problem.

3. Experience & conclusion

a. What have you learned from this homework?

Through this homework, I learned how to:

1. Solve the All-Pairs Shortest Path problem in parallel.
2. Use OpenMP to parallelize programs with SIMD property.
(Where data has no dependency.)
3. Sometimes static scheduling is faster than dynamic scheduling.
4. Generated input cases of All-Pairs Shortest Path problem.

b. Feedback (optional)

I have thought of trying different algorithms such as: Johnson's algorithm.

However, it says that only when the input size is small Johnson's algorithm would be faster than Floyd-Warshall.

I also thought maybe I could try dijkstra with V iterations (Each vertex takes turn being the starting point S)

But dijkstra's time complexity: $O()$

It would be :

3. Dijkstra's Algorithm :

- 只要Graph中的edge沒有**negative weight**，即使有**cycle**，亦可使用。
- 時間複雜度，根據實現**Min-Priority Queue**之資料結構將有所不同：
 - 若使用普通矩陣(array)，需要 $O(E + V^2)$ ；
 - 若使用**Binary Heap**，需要 $O((E + V) \log V)$ ；
 - 若使用**Fibonacci Heap**，只需要 $O(E + V \log V)$ 。

reference : <http://alrightchiu.github.io/SecondRound/shortest-pathintrojian-jie.html>

If we use Fibonacci Heap and get $O(E + V \log V)$

Now since we have V vertices, we do V times.

$V * O(E + V \log V) = O(V(E + V \log V)) = O(V^2 \log V + V * E)$

Maybe this would be faster than $O(V^3)$! Determine on the number of edges (E)

If a dense graph : $E = V^2$, it would be worse than $O(V^3)$

Finally, maybe I could try vectorization (4 integers at a time)

But the if-else statement would have to be revised and designed into a both concise and efficient logic to make significant accelerations.