Parallel Programming hw3

tags: PP20

106062230 徐嘉欣

Implementation

Which algorithm do you choose?

我使用Blocked Floyd Warshell來實作這次的作業。

Describe your implementation.

主要是參考HW4-1的sequential版本來實作這次的作業，以下為程式碼講解：

```
FILE* file = fopen(infile, "rb");
fread(&n, sizeof(int), 1, file);
fread(&m, sizeof(int), 1, file);

for (int i = 0; i < n; ++i) {
    for (int j = 0; j < i; ++j) {
        Dist[i][j] = INF;
    }
    Dist[i][i] = 0;
    for (int j = i + 1; j < n; ++j) {
        Dist[i][j] = INF;
    }
}

int pair[3];
for (int i = 0; i < m; ++i) {
    fread(pair, sizeof(int), 3, file);
    Dist[pair[0]][pair[1]] = pair[2];
}
fclose(file);
```

首先是讀取input file的部分，會先將有多少個vertices和多少edges讀取進來，再來將紀錄距離的array先initialize，然後再一一讀取input file中點到點的距離。

```
/*
 * in the main function
int B = 64;
int round = ceil(n, B);
block_FW(B, round);
*/
void block_FW(int B, int round) {
```

```
    for (int r = 0; r < round; ++r) {

        /* Phase 1*/
        cal(B, r, r, r, 1, 1);

        /* Phase 2*/
        cal(B, r, r, 0, r, 1); // up
        cal(B, r, r, r + 1, round - r - 1, 1); // down
        cal(B, r, 0, r, 1, r); // left
        cal(B, r, r + 1, r, 1, round - r - 1); // right

        /* Phase 3*/
        cal(B, r, 0, 0, r, r); // upper-left
        cal(B, r, 0, r + 1, round - r - 1, r); // lower-left
        cal(B, r, r + 1, 0, r, round - r - 1); // upper-right
        cal(B, r, r + 1, r + 1, round - r - 1, round - r - 1); // lower-
right
    }
}
```

這邊設定的block size是64，round則為要處理多少次迴圈的次數，接著就會開始進行blocked floyd warshall去計算All-pairs shortest path。再每一個round中，會挑選第(r, r)個block當pivot，第一phase會先進行pivot的calculation，第二個phase則是進行pivot正上方、正左方、正下方、正右方的那些長條block們的calculation，接著第三個phase是pivot的左上、右上、左下、右下的block們的calculation。要做第二個phase必須先執行完第一個phase，第三個phase必須先做完第一、二個phase，第二個phase間的4個cal()之間沒有dependency，第三個phase間的4個cal()之間也沒有dependency。

```
void cal( int B, int Round, int block_start_x, int block_start_y, int
block_width, int block_height) {

    int block_end_x = block_start_x + block_height;
    int block_end_y = block_start_y + block_width;
    int Round_B = Round * B;
    int max_val = ( ( (Round + 1) * B ) > n )? n : (Round + 1) * B;

    switch(block_height){
        case 1:
        {
            int block_internal_start_x = block_start_x * B;
            int block_internal_end_x = (block_start_x + 1) * B;

            if (block_internal_end_x > n) block_internal_end_x = n;

            #pragma omp parallel for schedule(dynamic)
            for (int b_j = block_start_y; b_j < block_end_y; ++b_j) {

                int block_internal_start_y = b_j * B;
                int block_internal_end_y = (b_j + 1) * B;

                if (block_internal_end_y > n) block_internal_end_y = n;
```

```
                for (int k = Round_B; k < max_val; ++k) {
                    for (int i = block_internal_start_x; i <
block_internal_end_x; ++i) {
                        int dist_i_k = Dist[i][k];
                        for (int j = block_internal_start_y; j <
block_internal_end_y; ++j) {
                            int val = dist_i_k + Dist[k][j];
                            int disk_i_j = Dist[i][j];
                            Dist[i][j] = val * (val < disk_i_j) + disk_i_j
* (val >= disk_i_j);
                        }
                    }
                }
            }
            break;
        }
        default:
        {
            #pragma omp parallel
            {
                #pragma omp for schedule(dynamic)
                for (int b_i = block_start_x; b_i < block_end_x; ++b_i) {

                    int block_internal_start_x = b_i * B;
                    int block_internal_end_x = (b_i + 1) * B;

                    if (block_internal_end_x > n) block_internal_end_x =
n;

                    for (int b_j = block_start_y; b_j < block_end_y;
++b_j) {

                        int block_internal_start_y = b_j * B;
                        int block_internal_end_y = (b_j + 1) * B;

                        if (block_internal_end_y > n) block_internal_end_y
= n;

                        for (int k = Round_B; k < max_val; ++k) {
                            for (int i = block_internal_start_x; i <
block_internal_end_x; ++i) {
                                int dist_i_k = Dist[i][k];
                                for (int j = block_internal_start_y; j <
block_internal_end_y; ++j) {
                                    int val = dist_i_k + Dist[k][j];
                                    int disk_i_j = Dist[i][j];
                                    Dist[i][j] = val * (val < disk_i_j) +
disk_i_j * (val >= disk_i_j);
                                }
                            }
                        }
                    }
                }
```

```
                }
                break;
            }
        }
    }
```

這邊以block為單位，一個一個block去進行floyd warshall，因為有些在處理列(行)時只會有一列(一行)，所以在最開始時先判斷`block_height`是否等於1，若是等於1的話就不要對b_i這層for loop進行paralize，改對b_j這個for loop進行平行，若不這麼做的話，就會在某些情況時只有一個thread在做事，其他thread都沒有拿到東西做。

我們可看到最內層的三個for loop(i.e, for (int **k = Round * B** ....)開始)，可看到k = Round * B，而在block_FW中，Round這個值會在[0, # blocks in a column - 1]中做更動，所以當整個block_FW做完後，其實k等價於在[0, # vertices - 1]中作更動，也就是找尋過所有的路徑了。並且因為使用if的話無法做vectorization，因此改成`val * compare + Dist[i][j] * (1 - compare)`的形式去判斷現在是否有更短的路徑出現。

在此之中，我盡量將會重複用到的變數提前到for loop外，避免重複計算的部分，或多或少能減少一些計算的時間。

```
void output(char* outFileName) {
    FILE* outfile = fopen(outFileName, "w");
    for (int i = 0; i < n; ++i) {
        fwrite(Dist[i], sizeof(int), n, outfile);
    }
    fclose(outfile);
}
```

最後再將計算完的資料一行一行寫到output file中，並且因為每一條邊的權重最多是1000，最多經過6000個點才能到達目的地，也就是最多最多計算出來的路徑長是6000000，也比INF(1073741823)的值小，因此不用擔心最終結果會超過INF，直接將一整行寫入即可。

```
CC = gcc
CXX = clang++
CXXFLAGS = -O3 -fopenmp -march=native
CFLAGS = -O3 -lm -fopenmp
TARGETS = hw3

.PHONY: all
all: $(TARGETS)

.PHONY: clean
clean:
    rm -f $(TARGETS)
```

這次的作業我不是使用原先Makefile中的`g++`去compile，改成使用`clang++`進行編譯，發現編譯後執行檔的速度比使用g++編譯的還快上不少。

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

先令block size = B。

```
for (int k = Round * B; k < max_val; ++k) {
    for (int i = block_internal_start_x; i < block_internal_end_x; ++i) {
        for (int j = block_internal_start_y; j < block_internal_end_y;
++j) {
            int val = Dist[i][k] + Dist[k][j];
            bool compare = val < Dist[i][j];
            Dist[i][j] = val * compare + Dist[i][j] * (1 - compare);
        }
    }
}
```

在這個for loop中會花費O(B^3)的時間，而每個block都需要進行這個運算，因此在一個round中會花費O((B^3) * (V/B)^2)。

一共要進行ceil(V/B)個round，因此計算blocked floyd warshall的時間是O(V^3)，再考慮有P個CPU cores，因此時間複雜度變為O((V^3) / P)。

讀寫檔案花費的時間複雜度都是O(V^2)，因此計算的時間複雜度是**O((V^3) / P)**。

How did you design & generate your test case?

```
int k = 0;
for (int i = 0; i < n; i++){
    for (int j = 0; j < n; j++){
        if(i != j){
            int a = rand();
            if( a % 3 != 0 ) {
                int b = ( rand() % 1000) + 1;
                (p[k]).start = i;
                (p[k]).end = j;
                (p[k]).weight = b;
                k++;
            }

        }
    }
}
m = k;
std::random_shuffle(p, p+m);
```

首先我先設定一個變數k來記錄之後會要輸出多少的edges，接著透過雙層for loop去進行testcase的製造，在i!=j時，會有1/3的機率那一條路徑是無法通行的，而剩下2/3的機率中，weight透過random去取權重，設定到

紀錄path的變數p中，最後再將p去做random_shuffle，使測資不要剛好是按照順序讀取的，可以增加一些cache miss的可能性。

使用random的原因是因為有時候random出來的測資反而算是偏難的，再加上測資大又要自己去設計的話會比較困難一些，因此採用random的方式。

Experiment & Analysis

System Spec

使用apollo做實驗測量。

Performance Metrics

首先，我使用`srun time -N1 -n1 -cCPUS ./hw3 INPUTFILE OUTPUT`的方式去測整個程式跑的時間。

```
std::chrono::steady_clock::time_point t1 =
std::chrono::steady_clock::now();
FILE* file = fopen(infile, "rb");
fread(&n, sizeof(int), 1, file);
fread(&m, sizeof(int), 1, file);
std::chrono::steady_clock::time_point t2 =
std::chrono::steady_clock::now();

for (int i = 0; i < n; ++i) {
    for (int j = 0; j < i; ++j) {
        Dist[i][j] = INF;
    }
    Dist[i][i] = 0;
    for (int j = i + 1; j < n; ++j) {
        Dist[i][j] = INF;
    }
}
std::chrono::steady_clock::time_point t3 =
std::chrono::steady_clock::now();

int pair[3];
for (int i = 0; i < m; ++i) {
    fread(pair, sizeof(int), 3, file);
    Dist[pair[0]][pair[1]] = pair[2];
}
fclose(file);

std::chrono::steady_clock::time_point t4 =
std::chrono::steady_clock::now();
std::cout << "Reading file took " <<
std::chrono::duration_cast<std::chrono::microseconds>(t4 - t3 + t2 -
t1).count() << "us.\n";
```

```
int main(int argc, char** argv) {
    // some computation before ...
    std::chrono::steady_clock::time_point t1 =
std::chrono::steady_clock::now();
    output(argv[2]);
    std::chrono::steady_clock::time_point t2 =
std::chrono::steady_clock::now();
    std::cout << "Writing file took " <<
std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count() <<
"us.\n";
}

void output(char* outFileName) {
    FILE* outfile = fopen(outFileName, "w");
    for (int i = 0; i < n; ++i) {
        fwrite(Dist[i], sizeof(int), n, outfile);
    }
    fclose(outfile);
}
```

再來使用上面兩個方格中的方式，去計算input、output所需要的時間，最後再用總時間扣除處理input、output的時間，計算出computation的時間。

```
void cal( int B, int Round, int block_start_x, int block_start_y, int
block_width, int block_height) {
    int block_end_x = block_start_x + block_height;
    int block_end_y = block_start_y + block_width;
    #pragma omp parallel
    {
        #pragma omp for schedule(dynamic)
        for (int b_i = block_start_x; b_i < block_end_x; ++b_i) {
            start_thread_time[omp_get_thread_num()] = omp_get_wtime();
            /*
             Computing here...
            */
            total_thread_time[omp_get_thread_num()] += omp_get_wtime() -
start_thread_time[omp_get_thread_num()];
        }
    }
}
```

我在開始進行computing前，使用`omp_get_thread_num()`取得當下計算這一行的thread的thread id，並且透過`omp_get_wtime()`獲取當時的時間儲存在`start_thread_time`中，完成目前分內的計算後，再使用一次`omp_get_thread_num()`獲取當下的時間，並扣除當初開始計算時取得的時間，將其累加在`total_thread_time`中，如此一來就能獲得所有thread的運算時間，以檢查是否有**load balance**。

下圖為範例:

```
slurm-2759440.out
  1   # CPU core = 7
  2   Reading file took 899423us.
  3   Writing file took 245970us.
  4   [Thread 0] thread time = 26.511926s
  5   [Thread 1] thread time = 26.496865s
  6   [Thread 2] thread time = 26.584244s
  7   [Thread 3] thread time = 26.552621s
  8   [Thread 4] thread time = 26.572968s
  9   [Thread 5] thread time = 26.581831s
 10   [Thread 6] thread time = 26.649924s
 11   192.70user 0.72system 0:28.80elapsed 671%CPU (0avgtext+0avgdata 121196maxresident)k
 12   0inputs+195320outputs (0major+205622minor)pagefaults 0swaps
```

Strong scalability & Time profile

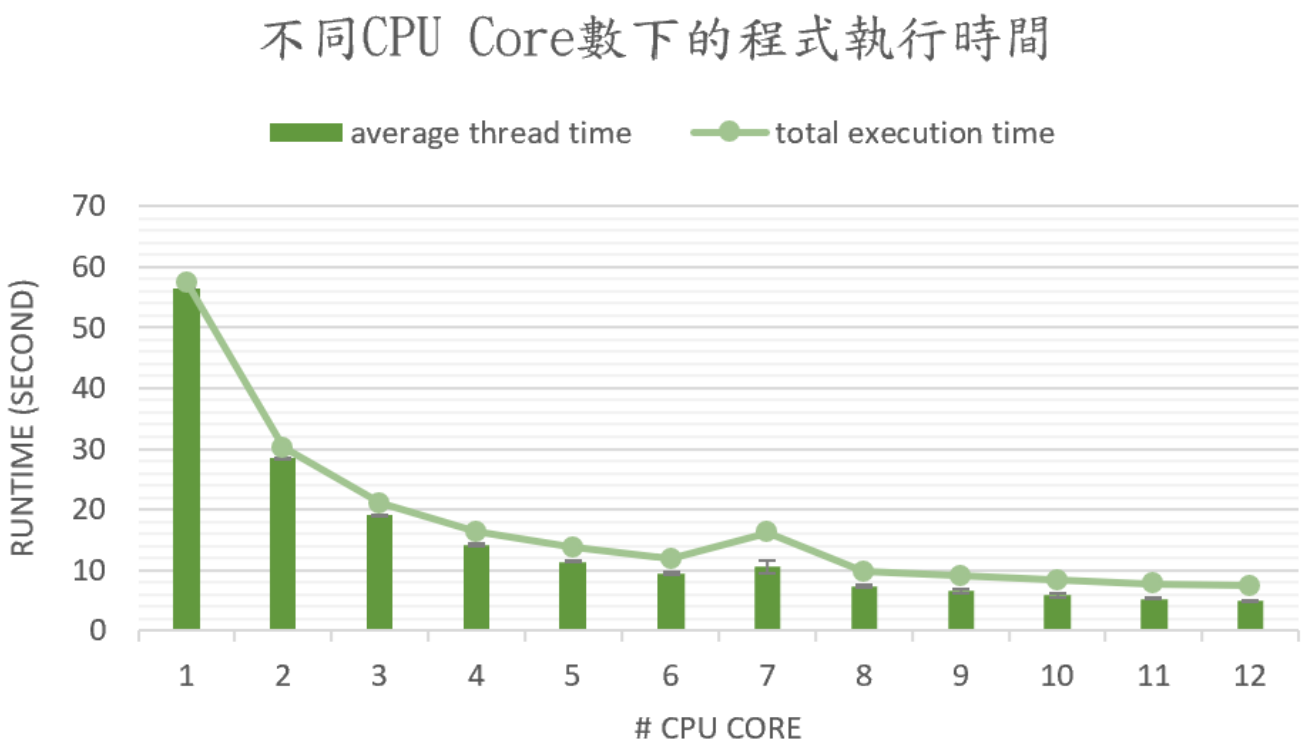
run testcase c21.1 with different number of CPU cores

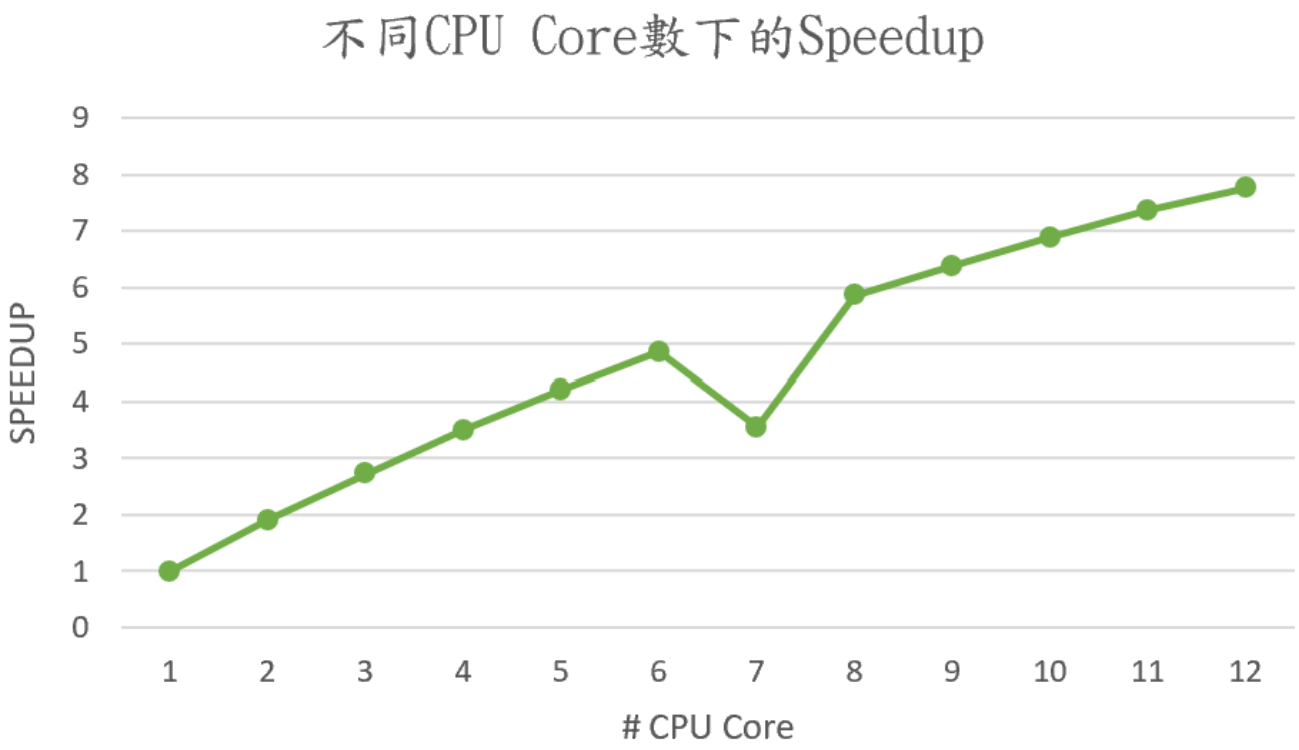
因為資料都放在同一個表格，會使表格變得太胖，因此拆成兩個表格放置。

# CPU core	total time (second)	computation time (second)	Reading file time (second)	Writing file time (second)
1	57.49	56.469	0.802406	0.218395
2	30.23	29.165	0.823854	0.240907
3	21.06	19.992	0.830524	0.237467
4	16.40	15.367	0.801152	0.231406
5	13.69	12.649	0.804857	0.235761
6	11.84	10.818	0.801948	0.220498
7	16.21	15.182	0.799767	0.228467
8	9.81	8.782	0.800557	0.227673
9	9.02	7.984	0.800198	0.235922
10	8.35	7.313	0.804636	0.232834
11	7.80	6.773	0.801046	0.226241
12	7.41	6.369	0.805362	0.235671

# CPU core	total time (second)	Average thread time (second)	Thread time標準差 (second)	speedup
1	57.49	56.410464	0	1
2	30.23	28.39245	0.183883118	1.901753225
3	21.06	18.96703067	0.157863657	2.729819563

# CPU core	total time (second)	Average thread time (second)	Thread time標準差 (second)	speedup
4	16.40	14.212721	0.187058542	3.505487805
5	13.69	11.4070234	0.131695856	4.199415632
6	11.84	9.5493345	0.225064332	4.855574324
7	16.21	10.58460386	**1.061511891**	3.546576188
8	9.81	7.3298005	0.317303613	5.860346585
9	9.02	6.529765111	0.268552758	6.373614191
10	8.35	5.850785	0.251766684	6.88502994
11	7.80	5.307520909	0.108886866	7.370512821
12	7.41	4.883583667	0.141354872	7.758434548



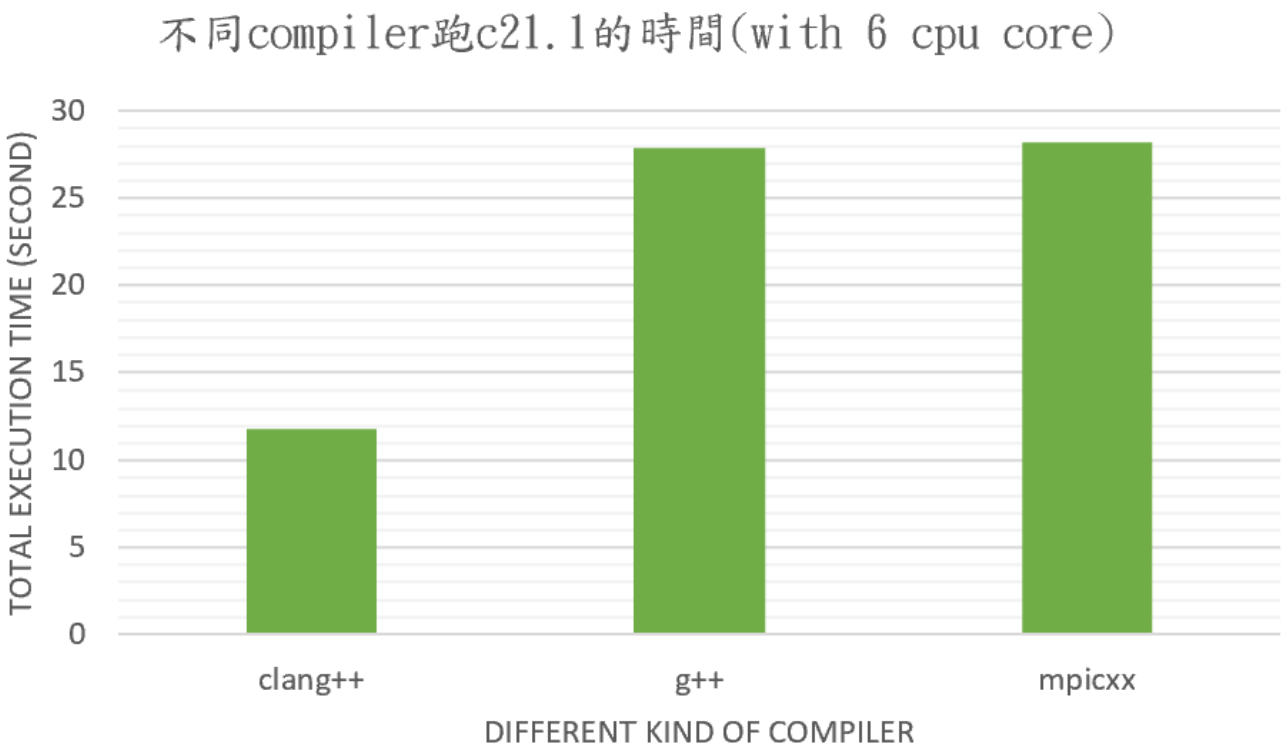


可以看到執行時間大致上是有隨著CPU core數的增加，而逐漸遞減，但因為有些情況的**load balance**沒有處理得很好，如CPU core數是7的時候，可看到總共7個thread的thread time標準差大於1，代表其實各個thread執行的時間稍微有些差距，有些thread比較輕鬆，有些thread比較勞累，致使效能沒有變好，還因此變差了。

而speedup的部分也大致是linear speedup，雖然沒辦法達到每增加一倍的cpu core時執行時間就減少一半，但加速的也差不多與cpu core數成正比。

我相信還有更好地分配工作方式，或者是還有能減少運算的部分，希望在之後學習到更多東西的我能想到優化此處的更多方法。

Different kind of Compiler



因為自己在寫作業卡關不知道該如何優化時，一時好奇改用了不同的compiler去編譯看看，發現對效能的影響還蠻大的，原先也想使用icpc測看看，但發現apollo上沒有icpc，因此只有使用`clang++`、`g++`、`mpicxx`去試看看，發現在測資較大時，`clang++`比`g++`與`mpicxx`快上了2倍以上。

Experience & conclusion

What have you learned from this homework?

在這次作業中，我學習到了Blocked Floyd Warshall這個方法，以前只有學過Floyd Warshall，這次更進一步地去優化Floyd Warshall，感覺到自己演算法有進步一小點XD。

這次最開始使用vectorization去進行優化的時候非常失敗，跑hw3-judge時的時間甚至變長了1.5倍，讓我一度非常失望，想放棄優化這次的作業，幸好後來還是有將vectorization做起來，讓程式有變快一些(~1.5倍)，感覺自己的vectorization又更進步了一些。

最後，還有學到不同編譯器(clang++, g++, ...)對執行速度的影響，之前用hw3-judge的時間卡在47s好多天的時候，想了很多方法都沒有一個明確將程式優化的方向，當時一時興起，想說把`g++`改成`clang++`看看，結果較大測資竟然快了大約2倍以上，讓我非常震驚，於是我又好奇嘗試`mpicxx`去編譯看看，才了解原來不同的編譯器對效能的影響也不算小。