Parallel Programming HW2

106062230 徐嘉欣

Implementation

Pthread version

```
typedef struct{
    int num_thread;
    int thread_id;
    int *image;
    double left;
    double right;
    double upper;
    double lower;
    int height;
    int width;
    int iters;
}Arg;
```

首先先typedef了一個資料結構，負責當作argument傳進pthread裡面，傳入的參數包含**共有多少CPU core**、**thread id**、測資會輸入的參數、以及pthread算出的**數值寫入之處(image)**。整個程式中，main做的事非常簡單，就是設定好Arg這個資料型態的參數、創建pthread、再將參數傳給pthread，讓pthread去執行threadFunc，接著就是等待所有的pthread都做完後，再call write_png。最重要的其實是在執行threadFunc這個部分，故篇幅會著重在此。

```
Arg* arg = (Arg*) argument;
int num_thread = arg->num_thread;
int thread_id = arg->thread_id;
int *image = arg->image;
double left = arg->left;
double right = arg->right;
double upper = arg->upper;
double lower = arg->lower;
int height = arg->height;
int width = arg->width;
int iters = arg->iters;
```

每個thread在最一開始會先將收到的argument的各個數值先記錄成local variable的形式，因為argument是一個pointer，若不先轉換成local variable的話，之後要使用到這些數值時，等價於做**memory access**，會花費額外的時間，因此在最開始先轉成local variables。

```
double tmp = ((right - left) / width);
double x0[width];
double y0[height];
int j_x_width[height];
#pragma GCC ivdep
for (int i = 0; i < width; ++i)
    x0[i] = i * tmp + left;
tmp = ((upper - lower) / height);
for (int j = 0; j < height; ++j) {
    j_x_width[j] = j * width;
    y0[j] = j * tmp + lower;
}
```

因為每個pixel在計算x0與y0時，分別會用到(right - left) / width與(upper - lower) / height，如果每個pixel都要計算這些的話，重複計算到的部分就非常多，因此先將這兩個數值記錄在tmp中，再一一去計算對應width、height的x0、y0，之後的pixel只要讀取就可以了，不需要再重複計算。而j_x_width這個變數也是為了不要重複計算j*width的值，因此先統一計算好，供未來做使用。

分配pixel給thread的規則是 : 將總共height*width這麼多個pixel想成一維的陣列，每個thread會需要做now_pixel % num_threads == thread_id的pixel們，每個thread大約會需要計算height*width/num_threads這麼多個pixel，比較能分配均勻，不會有其中一部份的thread需要多計算一行，這樣分配頂多只會多計算1個pixel。舉例來說，若有5個thread，且總共有15001個pixel，則thread_id == 0的thread需要處理第0, 5, 10, 15, ..., 15000個pixel，比其他人都多處理一個pixel。

```
int total = height*width; // 所有的pixel數
int count = thread_id + num_thread; // 目前此thread計算到的pixel

// 目前正在計算的兩個pixel的height與width分別是多少
int index[2][2] = { {thread_id/width, thread_id%width},
                    {count/width, count%width} };

// 目前正在計算的兩個pixel的x0、y0分別是多少
double y0_[2] = {y0[index[0][0]], y0[index[1][0]]};
double x0_[2] = {x0[index[0][1]], x0[index[1][1]]};

// 目前正在計算的兩個pixel已個別進行多少次iteration
int repeats[2] = {0, 0};

// 目前正在計算的兩個pixel的運算過程中會使用到的數值
double x[2] = {0, 0};
double y[2] = {0, 0};
double xx[2] = {0, 0};
double yy[2] = {0, 0};
double length_squared[2] = {0, 0};

// 目前正在計算的兩個pixel的位置
int now[2] = {thread_id, count};
```

然而為了套用**Vectorization**、一次計算兩個pixel去加速程式，會需要一些額外的變數來記錄目前計算中的兩個pixel的一些數值。底下是要計算mandlebort set的部分，因為程式碼比較長，不確定如何切割做解釋會比較好，因此將解釋打在註解中說明。

```
while (count < total) { // 目前正在計算的兩個pixel都不得超出範圍

    // 在這兩個pixel沒有確認不屬於mandelbrot set前繼續做
    while (length_squared[0] < 4 && length_squared[1] < 4
           && repeats[0] < iters && repeats[1] < iters) {

        // 套用vectorization去同時計算兩個pixel
        #pragma GCC ivdep
        for (int k = 0; k < 2; ++k){
            y[k] = 2 * x[k] * y[k] + y0_[k];
            x[k] = xx[k] - yy[k] + x0_[k];
            xx[k] = x[k] * x[k];
            yy[k] = y[k] * y[k];
            length_squared[k] = xx[k] + yy[k];
        }
        ++repeats[0]; ++repeats[1];
    } // 套出迴圈代表至少已經有一個pixel計算完畢

    // 若是pixel0計算完畢的話
    if(length_squared[0] >= 4 || repeats[0] >= iters) {

        // 將repeat的次數寫入image中，因為大家寫入的位置都不同，因此不用mutex
        image[ j_x_width[ index[0][0] ] + index[0][1] ] = repeats[0];

        // 將pixel0改成下一個要計算的pixel
        count += num_thread;
        now[0] = count;
        index[0][0] = count/width;
        index[0][1] = count%width;
        y0_[0] = y0[ index[0][0] ];
        x0_[0] = x0[ index[0][1] ];
        // 將數值歸零給新的pixel0做使用
        repeats[0] = 0;
        x[0] = y[0] = xx[0] = yy[0] = length_squared[0] = 0;
    }
    // pixel1同理於pixel0，之所以不是用else if是因為兩個可能同時計算完成
    if(length_squared[1] >= 4 || repeats[1] >= iters) {
        image[ j_x_width[ index[1][0] ] + index[1][1] ] = repeats[1];
        count += num_thread;
        repeats[1] = 0;
        x[1] = y[1] = xx[1] = yy[1] = length_squared[1] = 0;
        index[1][0] = count/width;
        index[1][1] = count%width;
        y0_[1] = y0[ index[1][0] ];
```

```
        x0_[1] = x0[ index[1][1] ];
        now[1] = count;
    }
}
```

```
// 前面的while loop做完時代表最多只剩一個pixel還未做完
// 原因是因為前面一次做兩個pixel，若還有兩個pixel要做，則不會跳出while loop

// 若未做完的是pixel0，則將他補做完成
if(now[0] < total){
    while (length_squared[0] < 4 && repeats[0] < iters){
        y[0] = 2 * x[0] * y[0] + y0_[0];
        x[0] = xx[0] - yy[0] + x0_[0];
        xx[0] = x[0] * x[0];
        yy[0] = y[0] * y[0];
        length_squared[0] = xx[0] + yy[0];
        ++repeats[0];
    }
    image[ j_x_width[ index[0][0] ] + index[0][1] ] = repeats[0];
}

// 若未做完的是pixel1，也將他補做完成
if(now[1] < total) {
    while (length_squared[1] < 4 && repeats[1] < iters){
        y[1] = 2 * x[1] * y[1] + y0_[1];
        x[1] = xx[1] - yy[1] + x0_[1];
        xx[1] = x[1] * x[1];
        yy[1] = y[1] * y[1];
        length_squared[1] = xx[1] + yy[1];
        ++repeats[1];
    }
    image[ j_x_width[ index[1][0] ] + index[1][1] ] = repeats[1];
}
```

另外，因為double乘double的乘法還是屬於比較花費時間的計算，希望能少去一些重複計算的部分，下面的程式碼中可以看到，上方的是原先的版本，下方的是優化後的版本，在原先的版本中重複計算了2次的x*x與y*y，因此我使用另外的變數xx與yy去記錄x*x與y*y，實測下來的結果大約能使花費的時間減少一成多左右。

```
/*  original version */
int repeats = 0;
double x = 0;
double y = 0;
double length_squared = 0;
while (repeats < iters && length_squared < 4) {
    double temp = x * x - y * y + x0;
    y = 2 * x * y + y0;
```

```
    x = temp;
    length_squared = x * x + y * y;
    ++repeats;
}

/* optimized version */
int repeats = 0;
double x = 0;
double y = 0;
double xx = 0;
double yy = 0;
double length_squared = 0;
while (length_squared < 4 && repeats < iters) {
    y = 2 * x * y + y0;
    x = xx - yy + x0;
    xx = x * x;
    yy = y * y;
    length_squared = xx + yy;
    ++repeats;
}
```

Hybrid version

```
int revcount[size]; // 記錄每個process會回傳的array大小
int displs[size] = {0}; /* 記錄每個process回傳回來的開始位置
                           之後會使用MPI_Gatherv去搜集數據 */

int h = height / size; // 先均勻分配行數給各個process
int remain = height % size;  // 剩下沒辦法均勻分配的行數
revcount[0] = (remain == 0)? h * width : ( h + 1 ) * width;

for(int i = 1; i < size; ++i){
    revcount[i] = (i < remain)? ( h + 1 ) * width : h * width;
    displs[i] = displs[i-1] + revcount[i-1];
}

int part_height = (rank < remain)? h + 1 : h; // 記錄自己要計算pixel的有幾行

// allocate空間去儲存運算的結果
int *part_image = (int*) malloc (width * part_height * sizeof(int));
```

```
double tmp_y = ((upper - lower) / height);
double tmp_x = ((right - left) / width);
double x[width];
#pragma GCC ivdep
```

```
for (int i = 0; i < width; ++i)
    x[i] = i * tmp_x + left;
```

和pthread version類似，因為會有一些重複計算的部分，因此先將他們計算好，之後就只需要讀值就好了。接著下面的程式碼是在解釋利用openMP去計算mandlebrot set的部分，因為程式碼比較長，不知道如何切割比較好說明，因此直接將說明打在註解中。

```
#pragma omp parallel num_threads(num_threads)
{
    #pragma omp for schedule(dynamic)
    // 每個process都只做j%size==rank的行數
    for (int j = rank; j < height; j += size) {
        double y0 = j * tmp_y + lower;

        // 計算在這一行開始前已經計算了多少個pixel
        int prev_pixel = (j / size) * width;
        int i; // 之所以放在for外面定義i是因為後面可能會用到

        /* 這邊的for loop跟pthread version的實作方式有些不同，
           是每兩個相鄰的pixel去做vectorization，若有其中一個
           pixel先做完，會先將剩下的那個pixel做完後，再往下找兩
           相鄰的pixel去做vectorization，原因是pthread方式的
           版本也有套用到hybrid version試過，但效能比這個差一些 */
        for (i = 1; i < width; i+=2) {
            double x0[2] = {x[i-1], x[i]};

            // 等等要計算的兩個pixel會使用到的參數
            int repeats = 0;
            double x[2] = {0};
            double y[2] = {0};
            double xx[2] = {0};
            double yy[2] = {0};
            double length_squared[2] = {0};

            /* 執行while的條件是兩個pixel的length_squared都沒有超過4
               且兩個pixel一起跑iteraion的次數也沒有超過iters這麼多次 */
            while ( length_squared[0] < 4
                    && length_squared[1] < 4 && repeats < iters) {
                for (int k = 0; k < 2; ++k){
                    y[k] = 2 * x[k] * y[k] + y0;
                    x[k] = xx[k] - yy[k] + x0[k];
                    xx[k] = x[k] * x[k];
                    yy[k] = y[k] * y[k];
                    length_squared[k] = xx[k] + yy[k];
                }
                ++repeats;
            }

            // 若repeats < iters代表有人的length_squared超過4了
```

```
        if(repeats < iters){
            /* 若pixel0還沒計算完的話，先將pixel1 repeat的
               次數記錄起來，再將剛才還沒有做完的pixel0做完  */
            if(length_squared[0] < 4) {
                part_image[prev_pixel + i] = repeats;
                while (length_squared[0] < 4 && repeats < iters) {
                    y[0] = 2 * x[0] * y[0] + y0;
                    x[0] = xx[0] - yy[0] + x0[0];
                    xx[0] = x[0] * x[0];
                    yy[0] = y[0] * y[0];
                    length_squared[0] = xx[0] + yy[0];
                    ++repeats;
                }
                part_image[prev_pixel + i - 1] = repeats;
            }
            /* 反之則先將pixel0 repeat的次數記錄
               起來，再將剛才沒有做完的pixel1做完  */
            else {
                part_image[prev_pixel + i - 1] = repeats;

                while (length_squared[1] < 4 && repeats < iters) {
                    y[1] = 2 * x[1] * y[1] + y0;
                    x[1] = xx[1] - yy[1] + x0[1];
                    xx[1] = x[1] * x[1];
                    yy[1] = y[1] * y[1];
                    length_squared[1] = xx[1] + yy[1];
                    ++repeats;
                }

                part_image[prev_pixel + i] = repeats;
            }
        }

        // 代表兩個pixel做了iters這麼多次後仍屬於mandelbrot set
        else{
            part_image[prev_pixel + i] = repeats;
            part_image[prev_pixel + i - 1] = repeats;
        }
    }

    // 跳出while loop，還是有可能有未做完的單個的pixel還要處理
    for (i = i-1; i < width; ++i) {
        double x0 = x[i];

        int repeats = 0;
        double x = 0;
        double y = 0;
        double xx = 0;
        double yy = 0;
        double length_squared = 0;
```

```
            while (length_squared < 4 && repeats < iters) {
                y = 2 * x * y + y0;
                x = xx - yy + x0;
                xx = x * x;
                yy = y * y;
                length_squared = xx + yy;
                ++repeats;
            }

            part_image[prev_pixel + i] = repeats;
        }
    }
}


/* allocate memory for gathered image */
// gather_image是拿來收集各個process計算出來的結果
int* gather_image = (int*)malloc(width * height * sizeof(int));
assert(gather_image);

/* gather all the part image results */
/* 因為每個prcoess計算的pixel數不同，因此不能單純使用MPI_Gather收集大家的結果。
   這邊改成使用MPI_Gatherv去收集，和MPI_Gather不同處在於要額外計算每個process
   傳送的array大小和這些array要從gather_image的哪裡開始放，可以注意到的是若一開
   始各個process分配的方法愈複雜，收集回來的結果要做整理也會變得複雜，要做好取捨。   */
MPI_Gatherv(part_image, revcount[rank], MPI_INT, gather_image, revcount,
                                    displs, MPI_INT, 0, MPI_COMM_WORLD);

/* change the position to the right position */
// 因為是gather到rank 0的process
if(rank == 0){

    int* image = (int*)malloc(width * height * sizeof(int));
    #pragma omp parallel num_threads(num_threads)
    {
        #pragma omp for schedule(dynamic)

        // 首先按照process的rank數去做整理，從rank 0回傳的數值開始
        for(int i = 0; i < size ; ++i){

            // 計算這個rank的process要計算多少行
            int part_h = (i < remain)? h + 1 : h;
            for(int j = i, now_h = 0; now_h < part_h; j += size, now_h++){
                // 因為當初各個process是跳著{size}行去做，因此這邊也要跳著{size}行去放
                int prev_pixel_0 = width * j;
                int prev_pixel_1 = displs[i] + now_h * width;

                // 將整行的資料搬遷過去
                #pragma GCC ivdep
                for(int k = 0; k < width; k++){
                    image[prev_pixel_0 + k] = gather_image[prev_pixel_1 + k];
```

```
                }
            }
        }
    }

    /* draw and cleanup */
    // 最後再寫入png file中
    write_png(filename, iters, width, height, image);
    free(image);
}
```

Hybrid version之所以沒有像pthread version一樣以pixel為單位去分配(可以更均勻地分配工作)，理由其實上面也有提到一些，因為**分配的方式愈複雜，整理的時候就會愈困難**，並且因為pthread時不需要太在乎data dependency的問題(因為每個thread處理的資料不同，因此迴圈內就算有一些data dependency也不會影響太大)，而若hybrid版的迴圈中有data dependency，使用openMP、pragma輕易地去分配給各個CPU core，反而會使平行度大打折扣，因此hybrid的版本反而我採取了比較單純的想法，目前也正在構思是否能夠有更好的做法。

Experiment & Analysis

System Spec

使用apollo進行實驗測量。

Methodology::Performance Metrics

Pthread version

```
void* threadFunc(void* argument)
{
    std::chrono::steady_clock::time_point t1 = std::chrono::steady_clock::now();
    /*
     Computing mandlebrot set here...
    */
    std::chrono::steady_clock::time_point t2 = std::chrono::steady_clock::now();
    std::cout << "[Thread " << thread_id << "] took " <<
std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count() << "us.\n";
    return NULL;
}
```

我在每個thread執行mandlebrot set的計算前，使用`std::chrono::steady_clock`取得當下的時間，完成mandlebrot set的計算後，再使用一次`std::chrono::steady_clock`獲取當下的時間，相減之後就能得到這個thread運算mandlebrot set的時間，如此一來就能獲得所有thread的運算時間，以檢查是否有**load balance**。

並且使用`srun -n1 -c $t ./hw2a out.png $iter $x0 $x1 $y0 $y1 $w $h`的形式去測量整個程式執行的時間。

結果如下圖所示:

```
experiments > slurm-2544428.out
  1   -n1 -c6 strict36.txt pthread version
  2   [Thread 5] took 27010168us.
  3   [Thread 3] took 27030007us.
  4   [Thread 4] took 27030333us.
  5   [Thread 2] took 27062072us.
  6   [Thread 1] took 27095022us.
  7   [Thread 0] took 27113845us.
  8   Writing png file took 2485272us.
  9   164.58user 0.05system 0:29.62elapsed 555%CPU (0avgtext+0avgdata 133968maxresident)k
 10   0inputs+34976outputs (0major+45776minor)pagefaults 0swaps
 11
```

Hybrid version

```
/* comm time */

start_comm = MPI_Wtime();
/* gather all the part image results */
MPI_Gatherv(part_image, revcount[rank], MPI_INT, gather_image, revcount, displs,
MPI_INT, 0, MPI_COMM_WORLD);
total_comm += MPI_Wtime() - start_comm;

/* thread computing time */

#pragma omp parallel
{
    #pragma omp for schedule(dynamic)
    for (int j = rank; j < height; j += size) {
        start_thread_time[omp_get_thread_num()] = omp_get_wtime();
        /*
         compute mandlebrot set here ...
        */
        total_thread_time[omp_get_thread_num()] += omp_get_wtime() -
start_thread_time[omp_get_thread_num()];
    }
}
```

我在開始mandlebrot set的計算前，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，完成這一行的mandlebrot set的計算後，再使用一次`std::chrono::steady_clock`獲取當下的時間，並扣除當初開始計算這一行時取得的時間，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**。

並且使用`srun -n $procs -c $t ./hw2a out.png $iter $x0 $x1 $y0 $y1 $w $h`的形式去測量整個程式執行的時間。

結果如下圖所示:

experiments > slurm-2560194.out

```
-n3 -c4 strict34.txt hybrid version
[Process 1] comm time = 0.076649s
[Process 1 thread 0] thread time = 46.054874s
[Process 1 thread 1] thread time = 46.073421s
[Process 1 thread 2] thread time = 46.070854s
[Process 1 thread 3] thread time = 46.075279s
[Process 2] comm time = 0.044199s
[Process 2 thread 0] thread time = 46.132834s
[Process 2 thread 1] thread time = 46.068124s
[Process 2 thread 2] thread time = 46.079736s
[Process 2 thread 3] thread time = 46.074920s
[Arrange image data took 26470us.
Writing png file took 3654780us.
[Process 0] comm time = 0.085682s
[Process 0 thread 0] thread time = 46.093488s
[Process 0 thread 1] thread time = 46.069261s
[Process 0 thread 2] thread time = 46.074747s
[Process 0 thread 3] thread time = 46.057845s
187.84user 0.10system 0:50.40elapsed 372%CPU (0avgtext+0avgdata 109964maxresident)k
0inputs+0outputs (0major+28376minor)pagefaults 0swaps
187.95user 0.08system 0:50.41elapsed 373%CPU (0avgtext+0avgdata 130868maxresident)k
0inputs+0outputs (0major+33263minor)pagefaults 0swaps
187.81user 0.28system 0:50.43elapsed 372%CPU (0avgtext+0avgdata 406064maxresident)k
0inputs+76008outputs (0major+92298minor)pagefaults 0swaps

```

從上圖的結果有個小發現，發現**其實最後整理image data時所花費的時間很短**，不到0.1秒，所以之後可以考慮把資料打得更散去做更好的load balance。

Plots: Scalability & Load Balancing

Pthread version

- 表(一) testcase strict36 **with vectorization**

# CPU core	execute time (second)	average thread time (second)	各個thread time的標準差 (second)	speed up
1	90.56	88.129167	0	1
2	46.58	44.0976095	0.07300523962	1.944182052
3	31.89	29.44207867	0.03178399604	2.839761681
4	24.53	22.06963825	0.01522834012	3.691805952
5	**25.22**	19.7198536	**2.774184631**	3.590800952
6	17.37	14.73107217	0.03170632273	5.213586644
7	15.11	12.65423686	0.01061250957	5.993381866

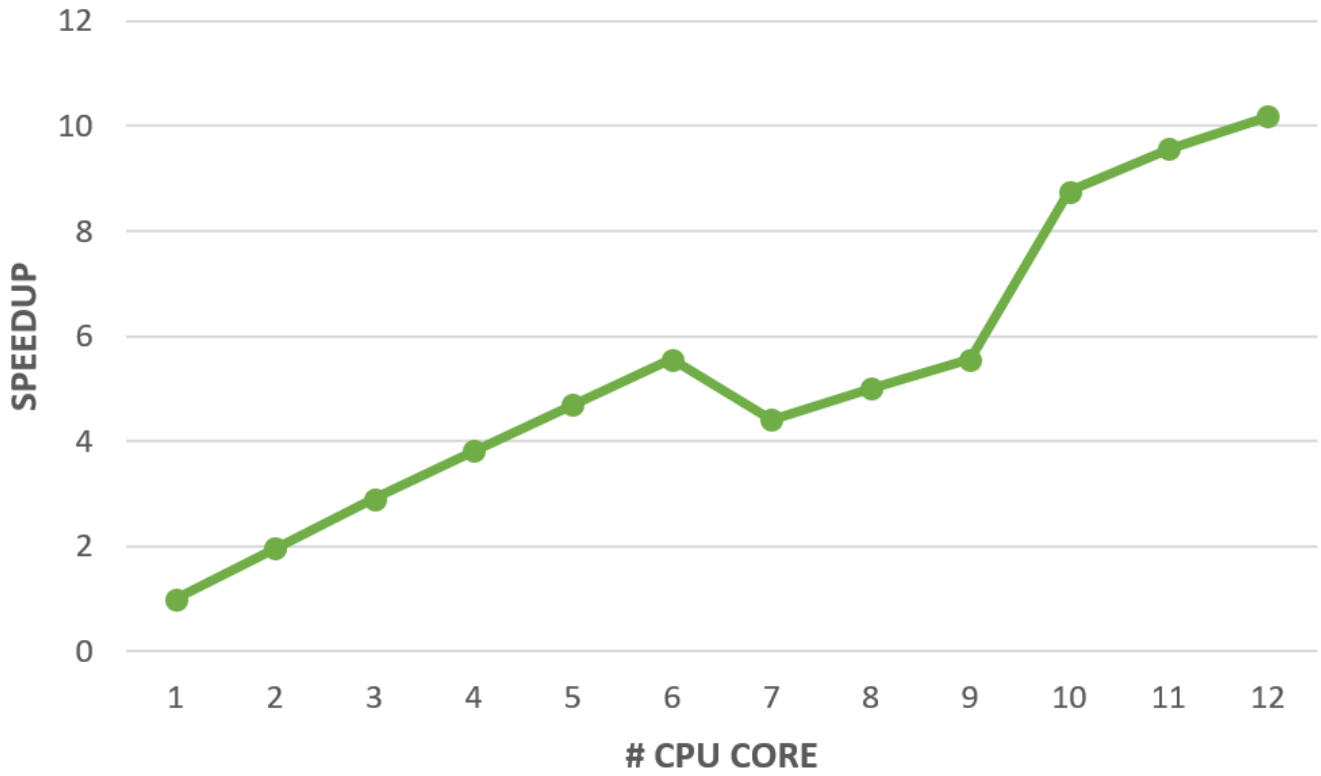
# CPU core	execute time (second)	average thread time (second)	各個thread time的標準差 (second)	speed up
8	**19.22**	12.53751113	**2.557510859**	4.711758585
9	**17.29**	10.95788044	**2.174216462**	5.237709659
10	11.44	8.8732108	0.04261600589	7.916083916
11	10.59	8.071788818	0.01423065009	8.551463645
12	9.90	7.409835667	0.047575851	9.147474747

- 表(二) testcase strict36 **without vectorization**

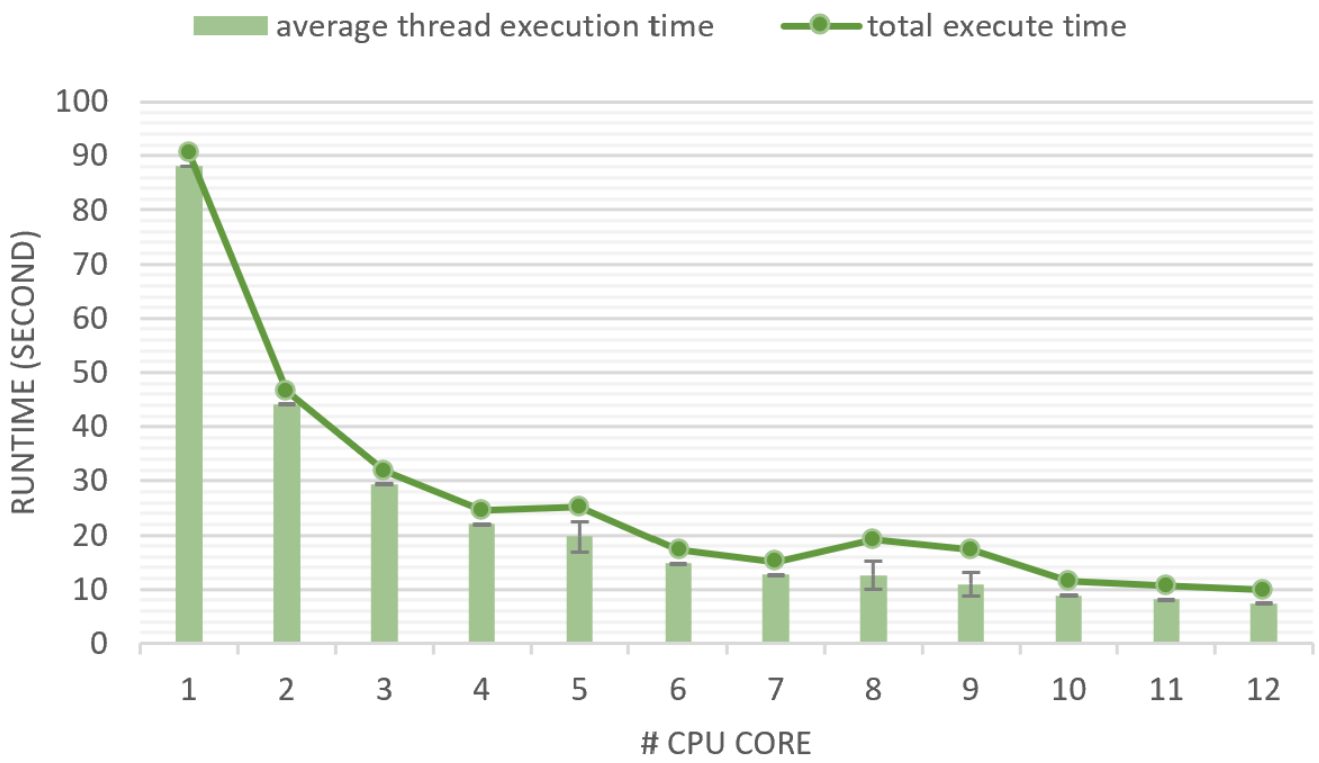
# CPU core	execute time (second)	average thread time (second)	各個thread time的標準差 (second)	speed up
1	164.71	162.287874	0	1
2	83.70	81.174974	0.0659433642	1.96786141
3	56.68	54.14839033	0.06306088297	2.905963303
4	43.06	40.57961275	0.0482305643	3.825127729
5	35.03	32.4964628	0.04013892302	4.70196974
6	29.62	27.05690783	0.04083059609	5.56076975
7	**37.30**	26.53449857	**5.681986398**	4.415817694
8	**32.93**	22.85579675	**4.675690763**	5.001822047
9	**29.61**	20.08948044	**3.991152541**	5.562647754
10	18.81	16.2612644	0.04524450017	8.756512493
11	17.23	14.78413209	0.01829217095	9.559489263
12	16.18	13.55554508	0.04258228826	10.17985167

- 圖(一) Speedup **with vectorization**

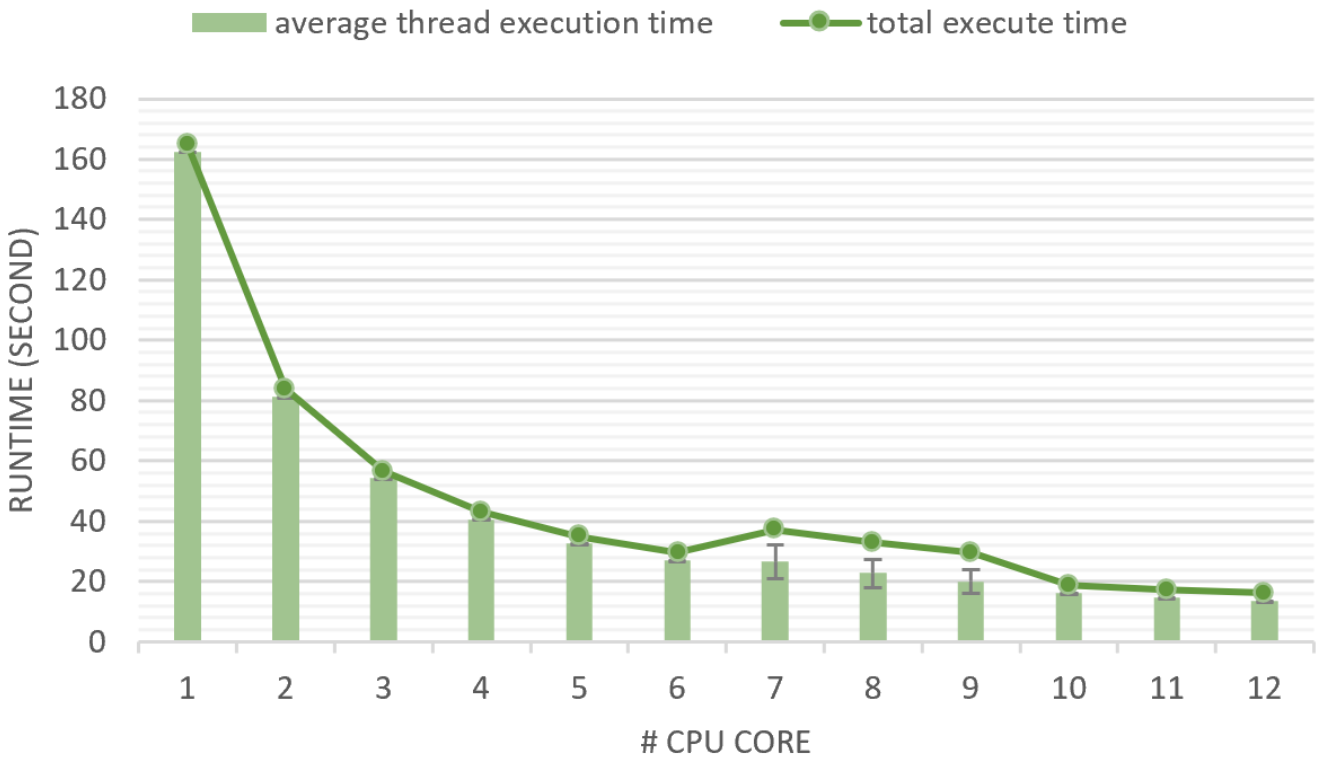
- 圖(二) Speedup **without vectorization**



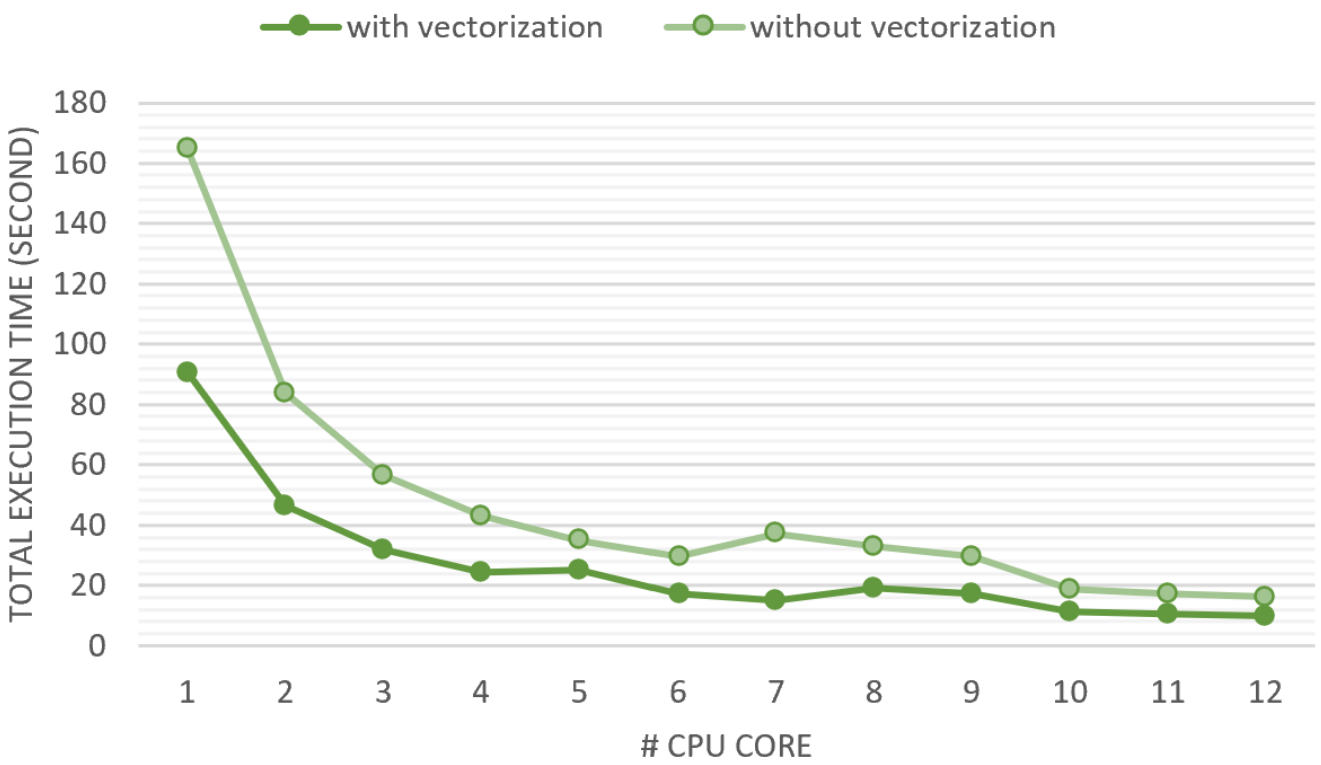
- 圖(三) Total execution time and thread execution time **with vectorization**



- 圖(四) Total execution time and thread execution time **without vectorization**



- 圖(五) Comparasion of total execution time with and without vectorization



Hybrid version

以下的實驗都是使用testcase strict34，**並將iters從1,0000改成10,0000，更能清楚地判斷結果**。

- 表(A) run on one node with vectorization

# CPU core per process	execute time (second)	average thread time (second)	各個thread time的標準差 (second)	average comm time	speedup
1	557.54	553.252468	0	0.076417	1
2	280.94	276.5777335	0.071903567	0.100973	1.984551862
3	188.77	184.4016953	0.019778107	0.099532	2.953541347
4	144.63	138.2326035	0.043710256	0.102765	3.854940192
5	114.98	110.604447	0.037990032	0.09837	4.84901722
6	103.45	98.96463617	0.041132751	0.121783	5.237576327
7	94.22	89.76287571	0.039520344	0.107439	5.917427298
8	81.35	76.92368413	0.021908293	0.096674	6.853595575
9	68.34	63.94805578	0.029678665	0.106822	8.158326017
10	59.77	55.3037244	0.031755076	0.094084	9.328091016
11	54.73	50.26636727	0.046377891	0.148638	10.18710031

# CPU core per process	execute time (second)	average thread time (second)	各個thread time的標準差 (second)	average comm time	speedup
12	50.47	46.06915017	0.008475932	0.101885	11.04695859

- 表(B) run on three nodes with vectorization

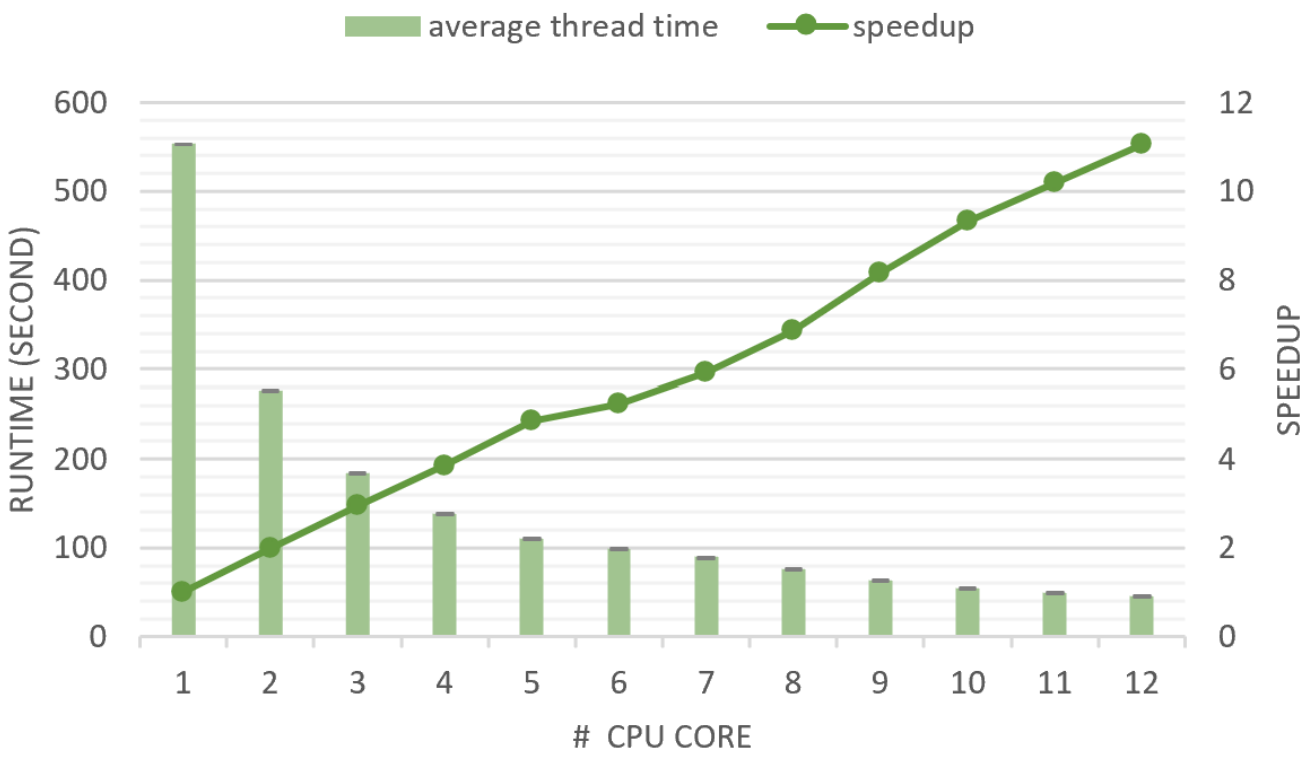
# CPU core per process	execute time (second)	average of each process' average thread time (second)	standard deviation of each process' average thread time (second)	average comm time	speedup
1	188.77	184.3853947	0.107969545	0.099804	2.953541347
2	96.54	92.1615615	0.002138561	0.0770863	5.775222706
3	65.8	61.44612567	0.010068691	0.06849	8.47325228
4	50.44	46.07180083	0.005293622	0.0774063	11.05352895
5	41.28	36.88417573	0.009596001	0.114941	13.50629845
6	35.18	30.73096728	0.009169804	0.1321806	15.84820921
7	**41.36**	33.35215805	**3.185690951**	**3.6630996**	**13.48017408**
8	27.59	23.05686583	0.009166882	0.1429013	20.20804639
9	**27.53**	21.45865207	**1.392401645**	**1.736467**	**20.25208863**
10	23.04	18.4761975	0.044096703	0.1811703	24.19878472
11	21.31	16.77450694	0.008927713	0.1491936	26.16330361
12	19.91	15.37914592	0.007411152	0.142103	28.00301356

- 表(C) 固定每個process的CPU core數並跑在不同數量的process上

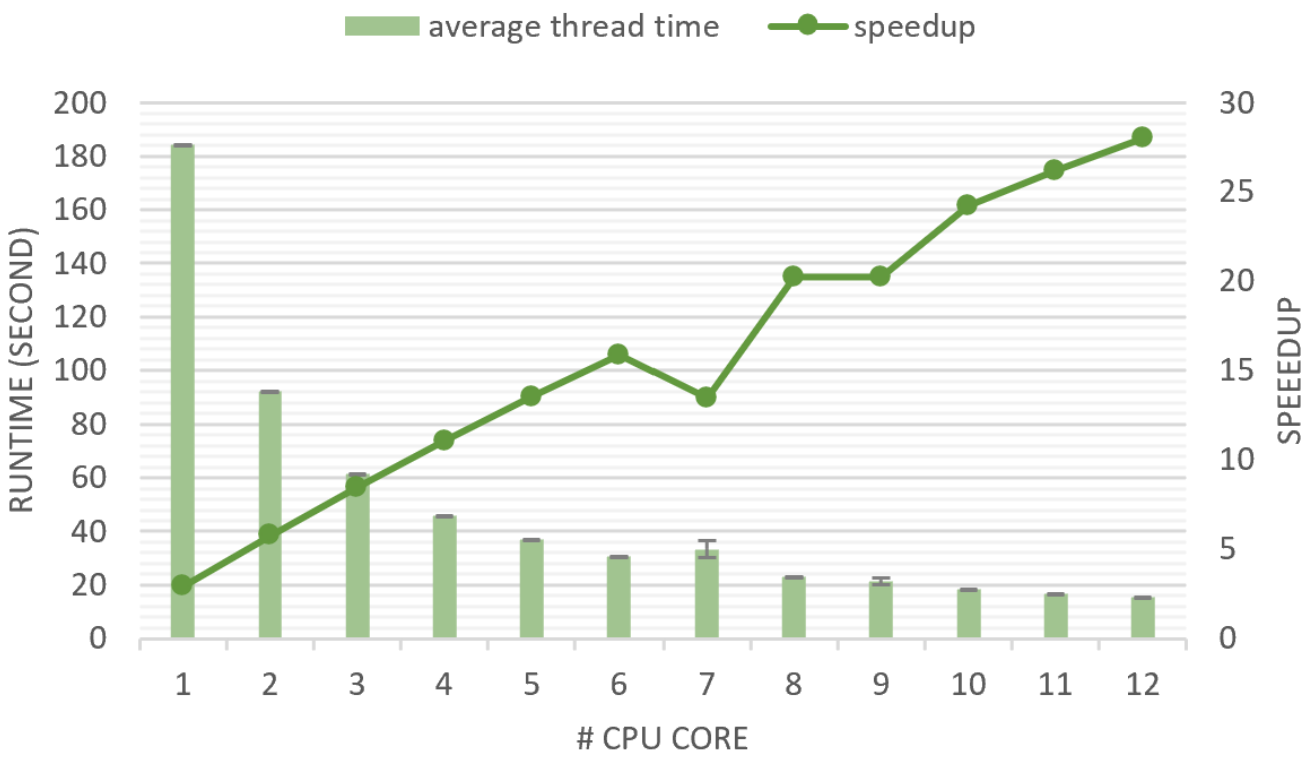
# process	execute time (second)	average thread time (second)	各個thread time的標準差 (second)	average comm time	speedup
1	557.21	552.872002	0	0.076902	1
2	281.01	276.563785	0.176683357	0.1647155	1.982883171
3	188.67	184.2856747	0.011784401	0.082708333	2.953357715
4	142.75	138.300317	0.075711732	0.06694875	3.903397548
5	115.1	110.5702346	0.01876067	0.0703684	4.8410947
6	96.79	92.191728	0.043739902	0.077934333	5.756896374
7	83.78	78.98981314	0.067555904	0.201808857	6.65087133
8	73.83	69.1280385	0.013825653	0.0622435	7.547203034

# process	execute time (second)	average thread time (second)	各個thread time的標準差 (second)	average comm time	speedup
9	66.23	61.45838533	0.023752015	0.052891889	8.413256832
10	60.18	55.3215629	0.013409435	0.0723937	9.259056165
11	55.24	50.28565482	0.058785741	0.087627636	10.08707458
12	51.13	46.09643683	0.013100856	0.07592875	10.8979073

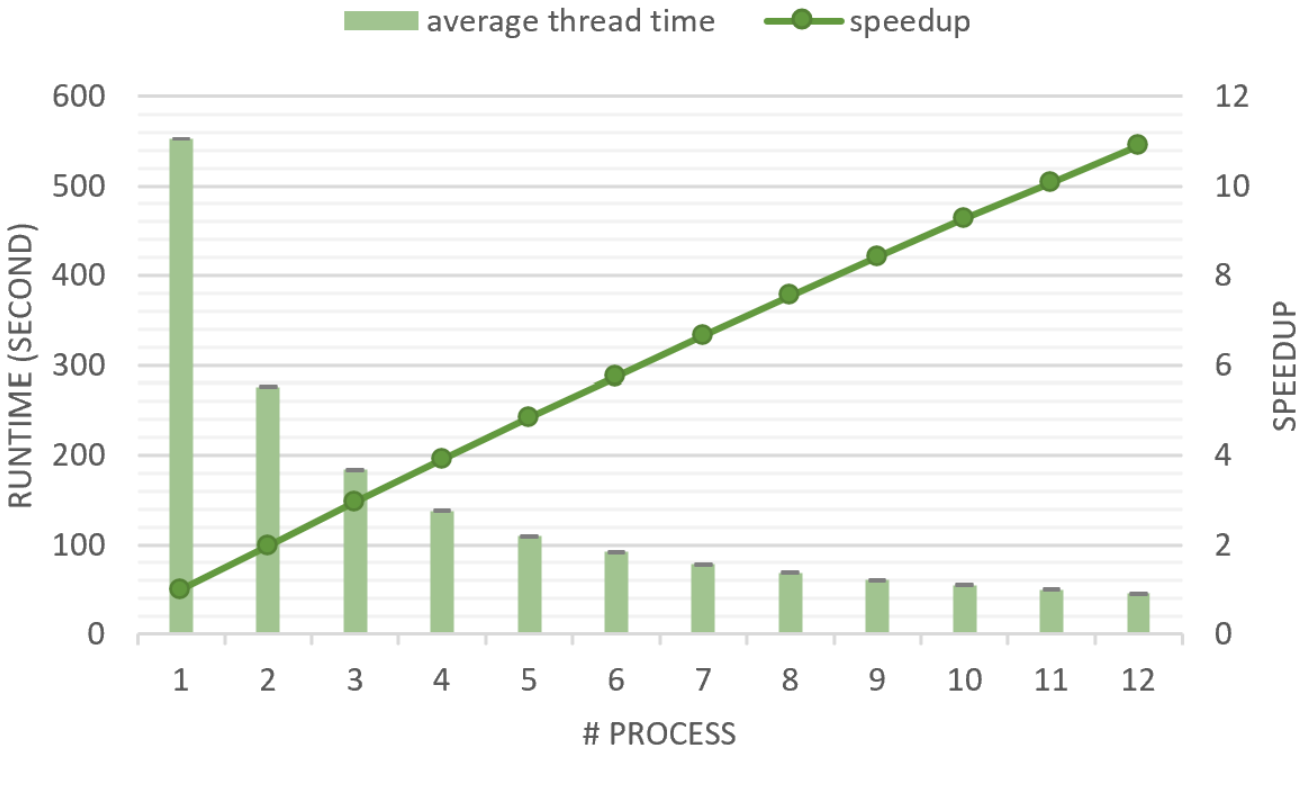
- 圖(A) 當只有在一個node上執行時，不同CPU core數的平均thread execution time與speedup (with vectorization)



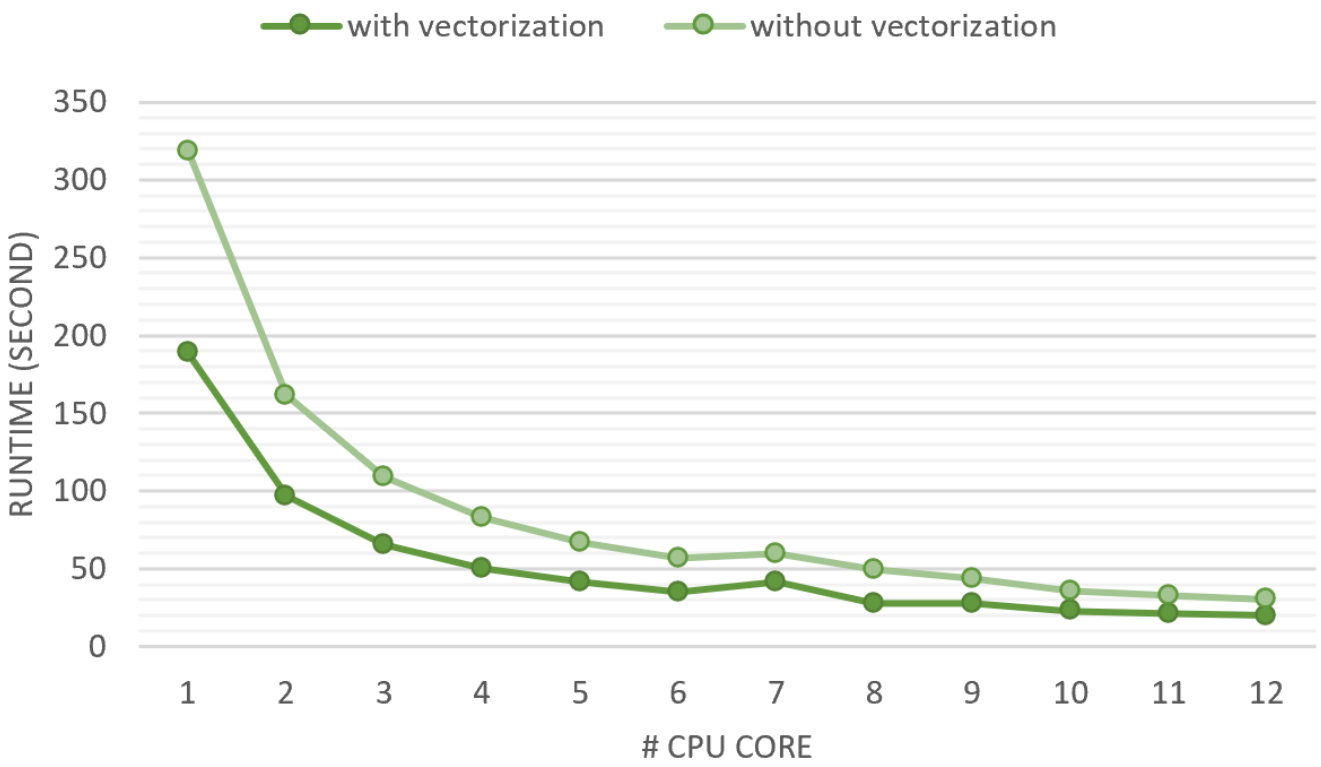
- 圖(B) 當在三個node上執行時，不同CPU core數的平均thread execution time與speedup (with vectorization)



- 圖(C) 固定每個process的CPU core數，跑在不同數量的process上的平均thread execution time與speedup (with vectorization)



- 圖(D) 當在三個node上執行時使用不同的CPU core數，比較有無使用vectorization的execution time



Discussion

Pthread version

從表(一)、圖(一)、圖(三)這兩個**有使用**vectorization的版本，可看到在CPU Cores是5、7、8時，整體的執行時間不減反增，speedup也大打折扣，這是因為當初在撰寫程式時，每個thread所需跑的行數大約都是固定的(約height/num_thread行)，然而**每行所需處理的時間卻不是固定的**，也就是有些行數是比較吃力的，在沒有處理好load balance的情況下，反而導致了執行時間的增加。而其他情況下每個thread實際執行的時間是差不多的(標準差都小於0.1)，代表每個thread的load是比較有達到balance的，所以speedup較符合預期。

而表(二)、圖(二)、圖(四)這兩個**沒有使用**vectorization的版本，也可以看出類似的結果，同樣因為每個thread所需跑的行數大約都是固定的(約height/num_thread行)，沒有考慮到實際上每行的困難程度不同，所以在CPU Cores是7、8、9時，導致了執行時間的speedup的不理想。

從圖(一)、圖(二)可看到兩種版本的scalability，除去沒有load balance的那幾點的話，可以發現**大致上speedup是與CPU core數成正比的**，沒辦法達到沒辦法達到每增加一倍的CPU core數就使得執行時間減半的原因是**有一部份的工作是比較不能平行處理的**，如最後寫成png file時，因為是一行易行的寫入，若是平行去執行則無法確保寫入的順序，又或者在處理每一行資料時欲開啟多個thread去跑那一行的資料，也不一定會使效能提升，因為創建pthread也需要花時間。這一部分的工作難以平行處理，因此想達到沒辦法達到每增加一倍的CPU core數就使得執行時間減半是有點困難的。

圖(五)是有無使用vectorization的execution time比較，可發現有使用vectorization的執行時間幾乎為沒有使用vectorization的一半，理由非常簡單，同時跑兩個pixel的時間大約就會比一次只跑一個pixel的快上兩倍左右。

在處理這些數據的時候，不難發現**scalability與load balance息息相關**，如果能確保每個thread負擔的運算是差不多的，而mandlebrot set又幾乎沒有data dependency的問題，所以增加一倍的CPU core，效能就會將近增加一

倍，然而**若load balance沒有做好，則可能會發生**如前面提到的**增加了CPU core卻使執行時間變得更長**。

Hybrid version

透過表(A)與圖(A)，我們可以從不同CPU core數下的thread time標準差，觀察出其實**各個thread之間是有達成load balance的**(標準差都小於0.1)，達成負載平衡的方法是透過`#pragma omp for schedule(dynamic)`去分配工作。並且也因為有達到load balance，運算過程中又沒有什麼data dependency，致使在**process數只有1個的時候，scalability非常地好**。

從表(B)&圖(B)則可以發現到一件很有趣的事，因為process數固定是2個，且分配工作的方式是讓process去運算`height % num_process == rank`的那些行，因此這兩個**process個別需要計算的行數與工作量不會跟著CPU core數的改變而有所差異**，所以若是在兩個process都只有1個CPU core可以使用時達成load balance，理論上在CPU core數等於2, 3, 4, ..., 12時都該要可以達成load balance，因為工作量是相同的，並且處理mandlebrot set的各個pixel之間也沒有data dependency的關係，理論上是不會影響到各個process間負擔重量的問題，然而從表(B)與圖(B)卻可以發現在CPU core數為7和9時，會發生效能相較於CPU core數為6和8時，效能不增反減的情況，猜測是因為**Hyper-Threading**的原因，將六核心模擬成了十二核心，導致在CPU core數為7時，會先用3核心模擬成6個CPU core，最後的1個CPU core可能是與其他program共用一核心。在研究數據時還發現有時候明明process的CPU core數是7個，執行的時間卻與CPU core數是6個時相近，有時候又會比較快，推測有時候可能是別的program佔用了該核心，以至於7個CPU core數卻只能使用6個。CPU core數是9個時推測也是這樣的情況。而CPU core數是11時為何沒有發生類似的情況，推測是因為當有process索取了11個CPU core數，apollo可能就會直接assign一整台電腦給他，就不會有其他program來跟單獨多出來的那個CPU core搶奪核心使用權。

而當process數增加到兩個、三個或以上時，因為每個process分配到的行數會是固定的(各個process會去計算`height % num_process == rank`的那些行，**並沒有考慮到每一行的困難程度**，這樣並無法保證在任何情況下都能達到load balance，然而出乎意料的是這樣的分配方式似乎能達到不錯程度的load balance，從表(C)與圖(C)可看到不同process中的每個thread所執行的時間幾乎相同，speedup也幾乎是linear程度的speedup，應該算是scalability與load balance都還算不錯的。

從圖(D)可觀察出有無使用vectorization的執行時間差異，有使用vectorization的時間大約為沒有使用的1/2 ~ 2/3左右，提升了非常多的效能，與pthread版本中的圖(五)相似，理由非常簡單，同時跑兩個pixel的時間大約就會比一次只跑一個pixel的快上兩倍左右。

Experience & Conclusion

透過這次的作業，我學習到如何使用多個thread去有效加速整體程式，對於mutex的使用更加得心應手，以前學習到關於mutex的知識時，都是紙上談兵，並沒有實際去操作，透過撰寫pthread version，讓我對mutex更加上手了；而這次同時也是我第一次寫openMP的程式，再次感嘆compiler的強大，只要寫幾個pragma，就可以使程式簡單地開啟多個thread同時跑。在撰寫hybrid version時，也發現static/dynamic分配對整體效能的影響真的很大，只有一個字的不同，卻對整體效能影響甚大，雖然openMP貌似很簡單，但其實也是個大哉問。

開始寫這次作業的時候，其實一路上都還算順利，大約花一個晚上就能把兩個版本的程式碼都寫出來，但是當時跑的時間真的超久，大約是助教的整整2倍多，研究了很久的vectorization總算成功了，但就算成功套用後，還是比助教的慢上一些，想了很久也沒有figure out到底還能如何優化，可能是自己的程式中還有一些多餘的運算，希望之後能夠陸續發現還有哪裡能改善。透過這次作業與上次作業，真實地感受到自己的coding能力有在進步，覺得非常開心，我會在剩下的半學期繼續學習、奮鬥，期許在之後的課堂中能更精進自己。