HW2 - Mandelbrot Set

106062322 江岷錡

Implementation

本次實作，需完成 `pthread` 版本與 `hybrid` 版本，以下將分別講解各自實作：

Pthread

Global Variable & Basic Configuration

由於本次需實作一個大型 Mandelbrot image 計算，同時 pthread 只會在 single node 上執行，我開了一個 global variable `image` ，負責存放輸出的 image 。

```
image = (int*)malloc(width * height * sizeof(int));
```

接著，在原本的公式中，每次計算 x 值與 y 值時，都會重複出現 `((upper - lower) / height)` 及 `((right - left) / width)` 。本次實作上，將其用 `y0Offset` 、 `x0Offset` ，存成 global variable 的形式，以避免重複計算。

```
y0Offset = ((upper - lower) / height);
x0Offset = ((right - left) / width);
```

當相關變數都宣告完畢後，就開始依據幾個 `ncpus` ，進行 pthread 的 create 。為了讓 pthread 的執行能平行化同步進行，我一致執行 `pthread_create` 後，最終才執行 `pthread_join` 進行收尾。

```
for (int t=0;t<ncpus;t++) {
    ID[t] = t+1;
    pthread_create(&threads[t], NULL, calcPixelValue, &ID[t]);
}

for (int t=0;t<ncpus;t++) {
  pthread_join(threads[t], NULL);
}
```

Divide the data

本次實作的切割資料，使用了兩種方式：

1. 根據 `row` 切：資料透過 row 為單位進行 partition，每個 thread 各自取得 partition 後執行計算。

   實作方式為，讓每個 thread 的執行 function 都包在一個無窮 `while` 迴圈中，在迴圈執行第一步，會呼叫 `getHeightPosition` ，取得 `row_id` 。由於 `getHeightPosition` 會被多個 thread 呼叫，有 critical section 的問題，我使用 `mutex_lock` 進行保護。當所有 row 都遍歷後， `getHeightPosition` 就會回傳 `-1` ，促使 `while` 迴圈結束。

   ```
   while(true) {
     pthread_mutex_lock(&mutex);
     curHeight = getHeightPosition();
     pthread_mutex_unlock(&mutex);
     if (curHeight == -1) break;
     ...
   }
   ```

   在 `getHeightPosition` func 中，主要負責 row 的分配處理，每當有任一 thread 前來存取，就回傳現在的 row_idx ，並進行 id++ 動作。

   ```
   int curHeightIndex = 0;
   int getHeightPosition() {
       if (curHeightIndex < height) return curHeightIndex++;
       else return -1;
   }
   ```

2. 根據 `Batch Size` 切：一樣透過單個 Row 進行切分，但每次只回傳固定的 columns 個數。透過宣告 `BATCH_SIZE` ，強制每次只回傳固定大小，試圖降低單次 thread 的計算 Loading 。

   ```
   int curLowerWidth = -BATCH_SIZE;
   int curHigherWidth = 0;
   int curHeightIndex = 0;
   int getHeightPosition() {
       if (curHigherWidth + 1 > width) {
           curHeightIndex++;
           if (curHeightIndex >= height) return -1;
           curLowerWidth = 0;
           curHigherWidth = BATCH_SIZE;
           return curHeightIndex;
   ```

```
    } else {
        curLowerWidth += BATCH_SIZE;
        curHigherWidth = ((curHigherWidth + BATCH_SIZE) > width ? width : cur
HigherWidth + BATCH_SIZE);
        return curHeightIndex;
    }
}
```

此方法經過驗證後，發現 Batch Size 在 row 的寬度以下，沒有絕對最佳解。同時，當設定 `BATCH_SIZE` 後，反而會使執行時間變長，因此保留方法一的操作。（下方有實驗證明）

Compute the Image

根據助教提示，全部使用 Vectorizartion 進行實作。設計上，以 async 方法進行實作。簡單來說，由於 Vectorization 在 double 的環境計算下，最多單次只能有兩個 channel 排程執行計算。同時，單一個 pixel ，在迭代計算上也有 dependency 的疑慮，不能將其 vectorization 。因此，在我的設計中，vectorization 的兩個 channel 會永遠保持，並一定計算兩個不同的 pixel，每當任一一個 channel 計算的 pixel 結束後，就換下一個 pixel 計算丟入。以下為簡單範例：

剛開始時，將 pixel1 與 pixel2 丟入 channel：

`[1, 2]`

過一段時間後， pixel2 先結束計算，將 pixel2 撤換成 pixel3 :

`[1, 3]`

過一段時間後， pixel3 先結束計算，將 pixel3 撤換成 pixel4 :

`[1, 4]`

過一段時間後， pixel1 先結束計算，將 pixel1 撤換成 pixel5 :

`[5, 4]`

以此方法，完成 async 的設計，也大幅增加計算效率。依據此概念後，以下將分別講解實作部分：

Load Vector Data

傳統上，load 通常需透過 `load_d` 相關的函式庫，但透過 assembly code 講解，該操作是進行 memory copy 的操作，理應較慢。我使用 `union` ，進行資料的處理：

```
union SsePacket {
    __m128d sseNum;
```

```
    double num[2];
};
```

Union 內的全部變數，會共享同樣的記憶體區間，這大幅度的加快我們的開發進程。當我們要 load double 時，只需要將 `num[0]` `num[1]` ，各自傳入相對應的變數，就可透過 `sseNum` 的提取，拿到 `__m128d` 型別的資料，進行 Vectorization 。

也因此，在 load __m128d 變數的過程中，就不必 call load 相關 API 。如下所示：

```
while(...) {
  if (finish[0] || finish[1]) {
      widthIdx++;
      if (finish[0]) {
          x0.num[0] =  widthIdx * x0Offset + left;
          x.num[0] = 0;
          y.num[0] = 0;
          length_square.num[0] = 0;
          repeats1 = 0;
          widthIdx1 = widthIdx;
          finish[0] = 0;
      } else if (finish[1]) {
          x0.num[1] =  widthIdx * x0Offset + left;
          x.num[1] = 0;
          y.num[1] = 0;
          length_square.num[1] = 0;
          repeats2 = 0;
          widthIdx2 = widthIdx;
          finish[1] = 0;
      }
  }
  ...
}
```

在此 while 迴圈中，每次執行時，都會檢查 channel 0 和 channel 1 是否計算完成。若是，則替換掉各自的 channel 初始值，讓其進行下一個 pixel 計算。在這些變數的設定過程中，以順便完成了 `__m128d` 的初始值導入。

Computation

```
while(...) {
  ...
  __m128d temp = _mm_add_pd(_mm_sub_pd(_mm_mul_pd(x.sseNum, x.sseNum), _mm_mul_pd
(y.sseNum, y.sseNum)), x0.sseNum);
  y.sseNum = _mm_add_pd(_mm_mul_pd(_mm_mul_pd(yMulSse, x.sseNum), y.sseNum), y0Ss
e);
  x.sseNum = temp;
```

```
  length_square.sseNum = _mm_add_pd(_mm_mul_pd(x.sseNum, x.sseNum),_mm_mul_pd(y.s
seNum, y.sseNum));
  ++repeats1;
  ++repeats2;
  ...
}
```

這部分依據原始的 code ，全部改成 vectorization 版本。也因為我們是以 vectorization 設計，會同時有兩個 channel 進行平行化計算。

```
while(!block[0] && !block[1]) {
  ...
  if (length_square.num[0] >= threshold || repeats1 >= iters) {
      if (!block[0] && finish[0] == 0) {
          image[curHeight * width + widthIdx1] = repeats1;
          finish[0] = 1;
      }
      if (widthIdx + 1 >= width) block[0] = 1;
  }
  if (length_square.num[1] >= threshold || repeats2 >= iters) {
      if (!block[1] && finish[1] == 0) {
          image[curHeight * width + widthIdx2] = repeats2;
          finish[1] = 1;
      }
      if (widthIdx + 1 >= width) block[1] = 1;
  }
  ...
}
```

當計算後發現超過 `threshold (Mandelbrot limit)` ，或 repeats 次數超過，則停止計算，將 pixel 值填入。值得注意的是，當該 row ，只剩下最後一個 pixel 計算時，會 label 上 `block[i]` 為 0 ，迫使跳出 while 迴圈，讓剩餘該 pixel 還未計算完的部分，用原本的方式，完成最後剩餘的 pixel 的計算，如下所示：

```
if (!finish[0]) {
    while (repeats1 < iters && length_square.num[0] < 4) {
        double temp = x.num[0] * x.num[0] - y.num[0] * y.num[0] + x0.num[0];
        y.num[0] = 2 * x.num[0] * y.num[0] + y0;
        x.num[0] = temp;
        length_square.num[0] = x.num[0] * x.num[0] + y.num[0] * y.num[0];
        ++repeats1;
    }
    image[curHeight * width + widthIdx1] = repeats1;
}
```

當全數跑完後，則寫回 image ，完成本次處理。

Hybrid

在 hybrid 版本中，需使用 OpenMP + MPI 。首先，在切分 data 的處理上，仍是依據目前的 MPI Size `N`，進行 row data 切分。

- 當總體資料小於 row `K` 數量時，則每人平均分一個 row ，超過的後續會被丟棄。
- 當總體資料大於 row `K` 數量時，先將 `N 除以 K` ，得出的商 `s` 與餘數 `m` 。接著，讓 `m` 以前的每個 process 多分一筆資料，即可完美均分資料。

```
int curHeightNum, avgHeight, modHeight = 0;
if (rank >= height) {
    avgHeight = 1;
} else {
    modHeight = height%mpiSize;
    avgHeight = height/mpiSize;
    curHeightNum = (rank < modHeight) ? avgHeight + 1 : avgHeight;
}
```

由於目前會有跨 node 的計算，已無法在共享 image global variable 。透過每個 Node 創造自己的 `curImage` ，儲存各自的計算結果。

```
int* curImage = (int *)malloc(width*curHeightNum * sizeof(int));
```

接續，為了有效平行化計算，我先在最外層包上 `parallel num_threads(ncpus)` ，並在內層套上 `#pragma omp for schedule(dynamic)` ，平行化 for 迴圈的計算。

切分資料上，對任一 node 的每次計算時，我試過兩種方法：

1. 直接依序分配定量的 row 給各 Node ： 根據上方分配的算法， `rank-0` 會拿到 `0 -> s-1` `rank-1` 會拿到 `s -> 2s-1` ，此方法較慢。
2. 非連續性的分配 row 給各 Node：迭代都以 `rank` `rank+N` `rank+2N` ... 的方式進行迭代，設定目前計算的 row。舉例來說， `rank-0` 會拿到 `0, 0+N, 0+2N...` 、 `rank-1` 會拿到 `1, 1+N, 1+2N...` 。此方法的理論背景為，通常 loading 較大的工作，會出現在附近的 row ，因次跳 row 的分配，會降低計算壓力。

```
#pragma omp parallel num_threads(ncpus)
{
    #pragma omp for schedule(dynamic)
    for (int heightIdx=0; heightIdx < curHeightNum; heightIdx++) {
        int curHeight = rank + mpiSize*heightIdx;
        int curImageIndex = heightIdx*width;
        ...
    }
}
```

接著，在內部實作中，與上方 pthread 講解相同，以 async 方式進行排程。

當計算完成後，會進行 `MPI_Gatherv` 的總和儲存。用 MPI_Gatherv 的原因在於，每個 Node 的總計算大小皆不同，`MPI_Gatherv` 可客製化每個 Node 回傳數量，讓整體運作更有彈性。

```
int* revcount = (int*)malloc(mpiSize*sizeof(int));
int* displs = (int*)malloc(mpiSize*sizeof(int));
displs[0] = 0;
for(int i=0; i<mpiSize; i++){
  if (i < modHeight){
    revcount[i] = (avgHeight+1)*width;
    if(i+1 < mpiSize) displs[i+1] = displs[i] + revcount[i];
  }
  else {
    revcount[i] = avgHeight*width;
    if(i+1 < mpiSize) displs[i+1] = displs[i] + revcount[i];
  }
}
MPI_Gatherv(curImage, curHeightNum*width, MPI_INT, image, revcount, displs, MPI_I
NT, 0, MPI_COMM_WORLD);
```

最終，由 `rank=0` ，進行圖片的輸出。由於在前置作業中，我透過跳 row 的方式進行資料劃分，導致 Gatherv 合成的圖片仍有誤差。故在此步驟，需將合成的圖片重新組織統整。

```
if(rank==0) {
    int* ansImage = (int*)malloc(imagesize * sizeof(int));
    int index = 0;
    for(int i=0;i<curHeightNum;i++){
        int realHeightIdx = 0;
        int jumpOffset = curHeightNum;
        for(int j=i; j<height && index < imagesize; j+=jumpOffset){
            int w0 = j*width;
            for(int w=0;w<width;w++){
                ansImage[index] = image[w0 + w];
                index++;
```

```
            }
            if(realHeightIdx < modHeight) jumpOffset = curHeightNum;
            else jumpOffset = avgHeight;
            realHeightIdx++;
        }
    }
    write_png(filename, iters, width, height, ansImage);
    free(ansImage);
}
```

Experiment & Analysis

Methodology

System Spec

使用學校提供的 Apollo Cluster。

Performance Metrics

本次使用 `clock_gettime` 進行採樣 time。由於用此 function 可達成 nanosecond 級的採樣，得以更細緻的進行表現差異的分析。

一個標準的 `clock_gettime` 範例如下：

```
struct timespec start, end, temp;
double time_used;
clock_gettime(CLOCK_MONOTONIC, &start);

....  something to be measured  ...

clock_gettime(CLOCK_MONOTONIC, &end);
if(end.tv_nsec < start.tv_nsec){
  output = ((end.tv_sec - start.tv_sec -1)+(nano+end.tv_nsec-start.tv_nsec)/nan
o);
} else {
  output = ((end.tv_sec - start.tv_sec)+(end.tv_nsec-start.tv_nsec)/nano);
}
time_used = temp.tv_sec + (double) temp.tv_nsec / 1000000000.0;

// Get what we want to evaluate
printf("%f second\n", time_used);
```

Scalability & Load Balancing

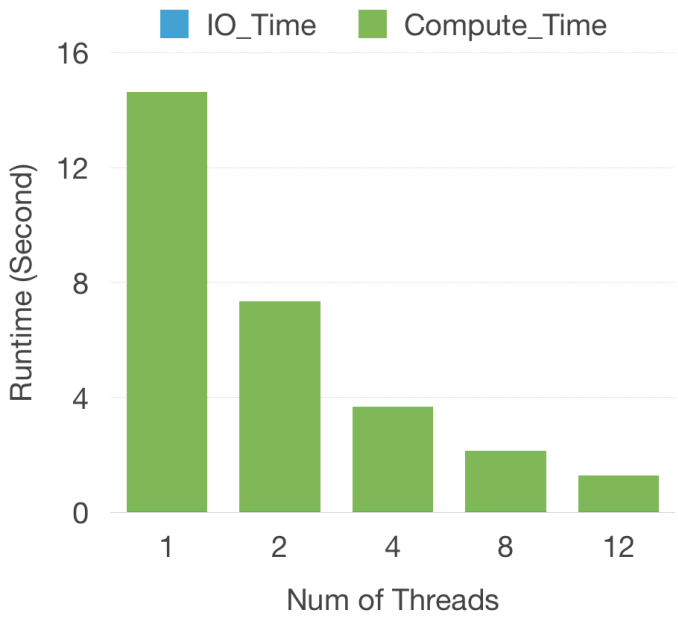
a(Pthread): Strong Scalability & Speedup Factor

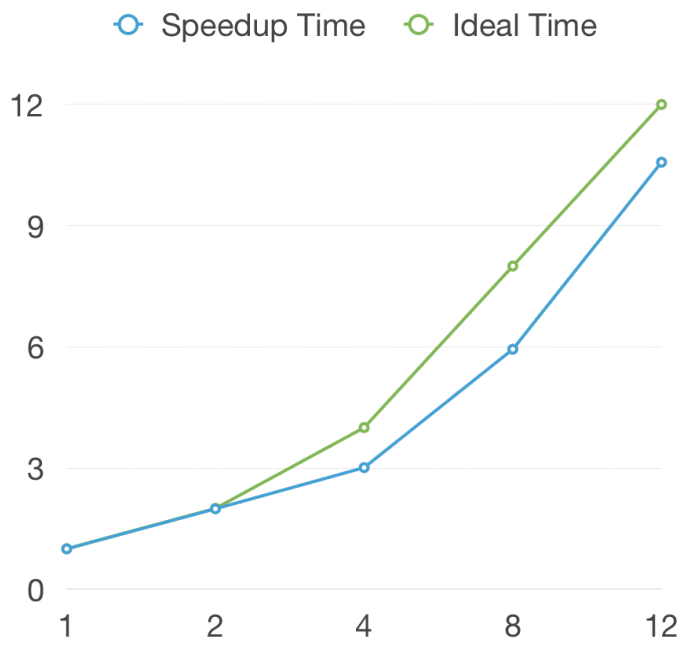
Time Profile(Pthread - Single Node)

Iteration Size: 100000

Strong Scalibility

	1	2	4	8	12
IO_Time	0.034229	0.034561	0.034192	0.034882	0.034628
Compute_Time	14.596764	7.315050	3.639862	2.102350	1.261233





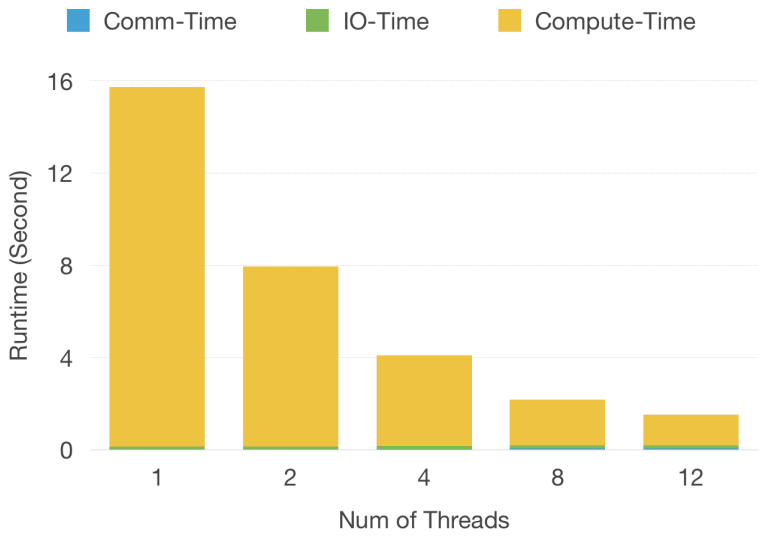
從實驗數據推測， `計算時間(Computation Time)` 為最主要的優化項目。當我們增加越多 threads 時，整體 speedup 的效率成大幅正相關，表示我們不但有效的分配運算資源，進行整張 image 的計算，loadbalancing 也處理妥當。Speedup 圖中，也可發現我們的增長幅度，大致趨向最佳狀態。

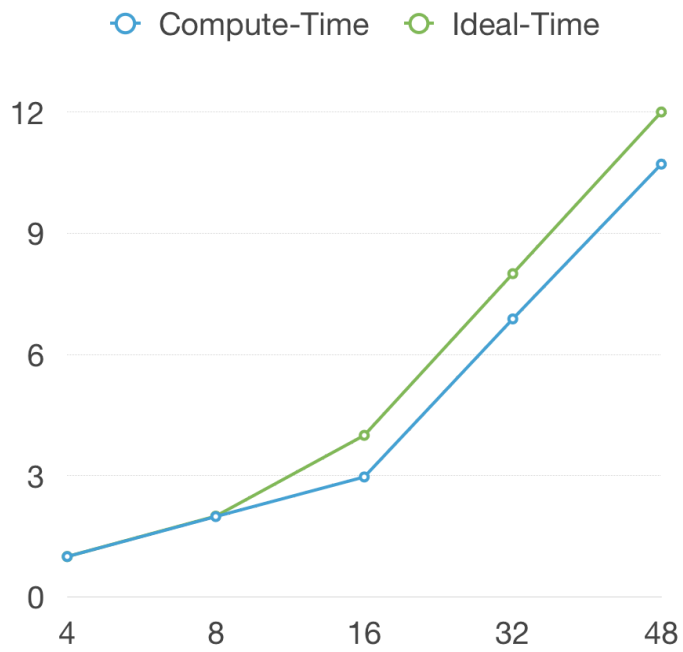
b(Hybrid): Strong Scalability & Speedup Factor

預設為 4 個 process ， 並在各自設定 thread 值。

Strong Scalibility - HW2b

	4	8	16	32	48
Comm-Time	0.006122	0.006865	0.045293	0.059293	0.062511
IO-Time	0.132052	0.133300	0.132024	0.132121	0.131330
Compute-Time	15.579105	7.799472	3.920940	1.976719	1.329615





從實驗數據推測，`計算時間(Computation Time)` 仍為最主要的優化項目。當我們增加越多 threads 時，整體 speedup 的效率成大幅正相關，loadbalacing 也進行了良好處理。

Load Balancing

在 Load Balancing 實驗中，我透過檢查每個計算單位（ thread for hw2a and node for hw2b ）的計算時間，查看我們的分配效率。共有三個 Metric 可供檢查：

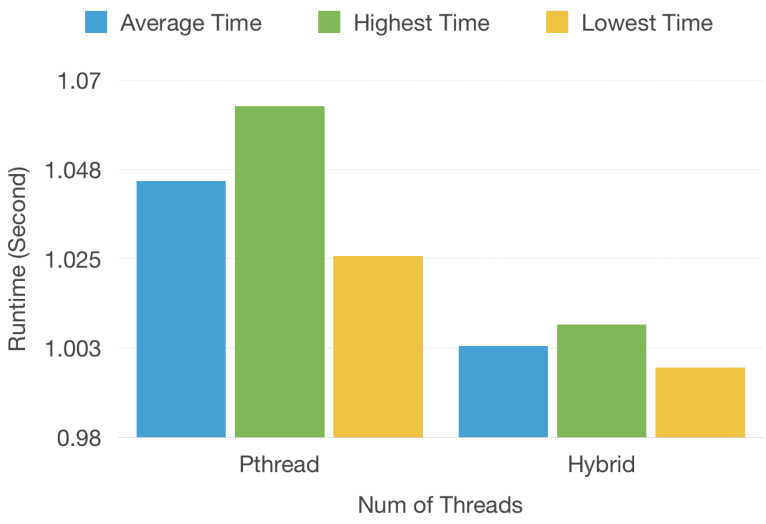
1. `Average Time`：平均每個計算單位的執行時間

2. `Highest Time`：每個計算單位中，最高的執行時間
3. `Lowest Time` ：每個計算單位中，最低的執行時間

合理上，最好的 Load Balancing ，應該要讓 `Highest Time` 和 `Lowest Time` ，與 `Average Time` 更接近。我固定開設 8 thread ，檢查分配效率：

Load Balancing

	Pthread	Hybrid
Average Time	1.044574	1.003030
Highest Time	1.063515	1.008401
Lowest Time	1.025633	0.997658

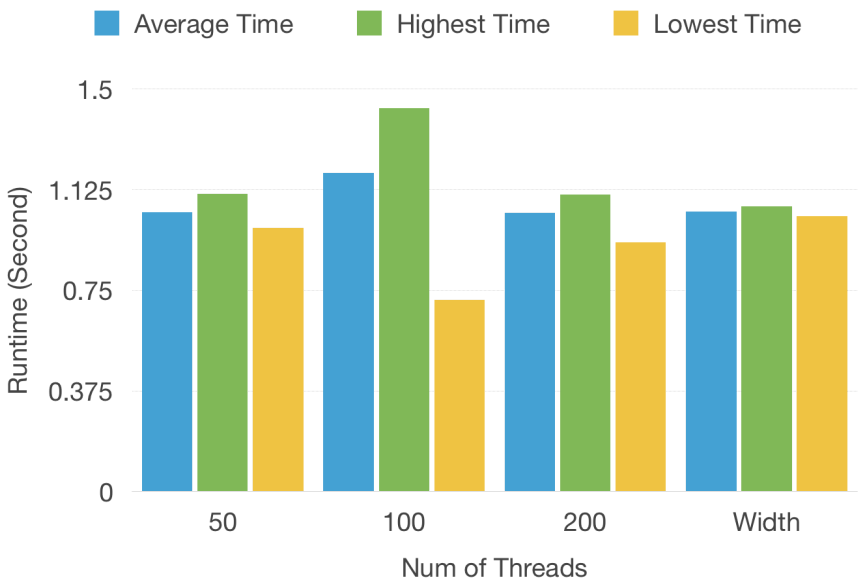


從圖中可見，Pthread 版本中，看似照著 width 分，理論上無法到最佳的 Load Balancing ，但實驗結果發現仍相當平均，約上下差只差 0.02 秒。個人推測主要原因，是因為我使用 async 方式計算，導致切分長度不會是太大問題。我透過執行第二個實驗，驗證這個想法。另外， Hybrid 版本中，幾乎每個 Node 被分配到的計算壓

力都相同，使其有較好的計算效率。反映在 scoreboard 上，也得到很好的 ranking 。

Pthread - Specific Version

	50	100	200	Width
Average Time	1.041063	1.188591	1.038055	1.044574
Highest Time	1.110094	1.429514	1.108181	1.063515
Lowest Time	0.981875	0.715266	0.928674	1.025633



這個實驗中， `50` 、 `100` 、 `200` 及 `Width` ，分別代表單次不同的 `BATCH_SIZE` 。數據中可見，從 50 → 100 → 200 → Width 的過程中，反而 100 的 `BATCH_SIZE` 執行的最差，沒有一個趨勢上的呈現，哪種方向算法可以得到最佳 Load Balancing。我認為這更加證明，在 async 的方法撰寫下， Load Balancing 的大小並不是最主要的 Bottleneck ，只要 BATCH_SIZE 不要設定超過 Width，基本上效能上不會差別太大。

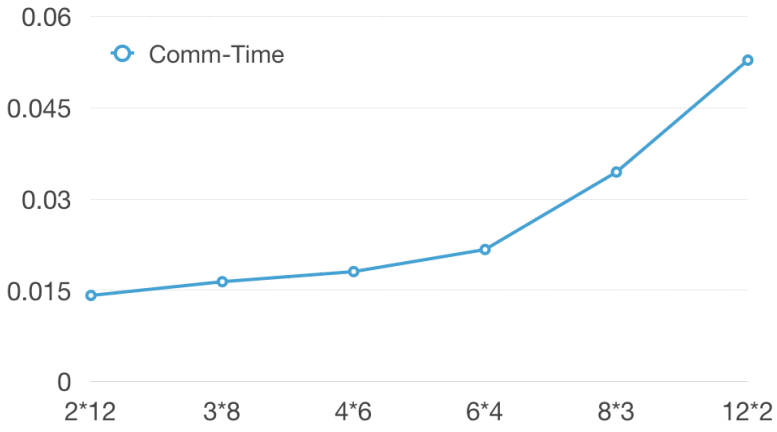
Other Experiment

本次實作 hybrid 時，發現不同的 proc \ threads 分配方法，會讓最終結果產生差異。舉例來說，當我們需要有 24 threads 進行實作時，可以有 `2 process * 12 threads` 分配方法，也有 `12 process * 2 threads` 的分配方法。透過實驗，試圖找出最好的分配方式：

圖示： `2*12` 代表 2 proc 與每個 proc 有 12 threads，以此類推。

Hybrid Experiment

	2*12	3*8	4*6	6*4	8*3	12*2
Comm-Time	0.014162	0.016424	0.018081	0.021713	0.034446	0.052829



實驗中可見，當我們想產生足夠的 threads 執行時，最好的方法應是 `用少一點 process 與 單 process 用多 threads`，可以避免 communication time 的 exponential 成長。

Conclusion

本次實作，在 Vectorization 上花費的時間，真的相當多。從發現 load double ，和助教的講義寫的方向完全不一樣，到套用 Union、用 async 全面改寫 ，Debug ，整體花費時間相當多。但成就上，也從最初的第一版 764s ，進步到最後 400s ，基本上將近到 2 倍速度。在整理數據時，用數據化的方式展現優化表現，又發現一些可以再改進的方向，多做了一些優化。同時，整體優化的過程也曠日費時，有時直覺想像

中可優化的實作，數據上不一定也能呈現優化，如 `切 row balancing` 反而表現更好，或 `相鄰的 row 計算量` 其實差不多，所以跳 row 分配反而更好。