# Parallel Programming - Hw2 Report

**107062228 陳劭愷**



# Implementation

本次作業中，需要實作 Mandelbrot Set 的計算。分別在單節點利用 Pthread 以及多節點運用 MPI + OpenMP 來進行優化。

## Pthread Version

首先實作 Pthread 的版本，因為 Mandelbrot Set 每一個 height 的計算工作量都不太一樣，因此本來是實作平均分配 heights 給每一個 threads，後來改為每一個 thread 都共同存取 `cur_height` 變數，來得到現在要運算的高度是哪一個，以達到較好的 load balancing。因為需要共同存取 `cur_height` 並做加一，因此有透過 mutex lock 來避免 race condition。

```
int local_height;
pthread_mutex_lock(&mx);
local_height = cur_height++;
pthread_mutex_unlock(&mx);
```

接著先做一些運算方面的優化，包括：

1. 避免重複計算 `(upper - lower) / height` 以及 `(right - left) / width`。

2. 避免重複計算 `x * x`、 `y * y`、 `x * y`。

```
while (repeats < iters && length_squared < 4) {
  double xy = x * y;

  y = xy + xy + y0;
  x = xx - yy + x0;

  xx = x * x;
  yy = y * y;
  length_squared = xx + yy;
  ++repeats;
}
```

→ 674.45 seconds.

## Vectorization

嘗試助教上課所說到的 vectorization 來優化運算的部分。主要想法就是透過兩個 double 的同時運算來增加運算速度。Height 的部分依然是每個 thread 獨立的去分配，

```
int local_width = 0;
int repeats[2] = {0, 0};
int doing[2] = {-1, -1};
__m128d x0 = _mm_setzero_pd();
__m128d y0 = _mm_set_pd1(local_height * y_offset + lower);
__m128d x = _mm_setzero_pd();
__m128d y = _mm_setzero_pd();
__m128d xx = _mm_setzero_pd();
__m128d yy = _mm_setzero_pd();
__m128d length_squared = _mm_setzero_pd();
```

首先是先把原本使用到的變數，運算相關的都先變成 `__m128d` 的 Streaming SIMD Extensions 型態。接著一次拿出兩個 width 來做計算，因為每個 width 的計算時間可能差很多，所以實測發現如果等到兩個 widths 都計算完再換下一組，這樣效率不佳。因此我改成當有一個 width 運算完之後，就直接停止運算，將下一個 width 初始化到這些運算用的 `__m128d` 型態變數，再繼續運算。

其中新加的變數：`local_width` 紀錄現在運算到哪個 width。`doing` 紀錄現在計算的兩個 width 分別是哪兩個。

在一個 `while (true)` 的迴圈中：

1. 先檢查 `doing` 來確認有沒有正在運算的 width，沒有的話就抓下一個 width 來算並做初始化。

```
if (doing[0] == -1) {
  if (local_width == width) break;
  x0[0] = local_width * x_offset + left;
  x[0] = y[0] = xx[0] = yy[0] = length_squared[0] = 0;
  doing[0] = local_width;
  repeats[0] = 0;
  local_width++;
}
```

2. 將原本的計算改為兩個 width 一起運算，當一個 width 結束了就立刻停止。

```
while (repeats[0] < iters && repeats[1] < iters) {
  ++repeats[0];
  ++repeats[1];
  __m128d xy = _mm_mul_pd(x, y);

  y = _mm_add_pd(_mm_add_pd(xy, xy), y0);
  x = _mm_add_pd(_mm_sub_pd(xx, yy), x0);

  xx = _mm_mul_pd(x, x);
  yy = _mm_mul_pd(y, y);
  length_squared = _mm_add_pd(xx, yy);
  if (!(length_squared[0] < 4 && length_squared[1] < 4)) break;
}
```

3. 判斷是哪一個 width 計算完畢，把顏色寫到對應的位置上面。

```
if (!(repeats[0] < iters && length_squared[0] < 4)) {
  set_color(row + doing[0] * 3, repeats[0]);
  doing[0] = -1;
}

if (!(repeats[1] < iters && length_squared[1] < 4)) {
  set_color(row + doing[1] * 3, repeats[1]);
  doing[1] = -1;
}
```

最後離開 `while (true)` 迴圈後，可能還有尚未計算完畢的一個 width，因此補上剩餘的運算：

```
if (doing[0] != -1) {
  // do 0
  int& repeat = repeats[0];
  double _x0 = x0[0];
  double _y0 = y0[0];
  double _x = x[0];
  double _y = y[0];
  double _xx = xx[0];
  double _yy = yy[0];
  double _length_squared;
  while (repeat < iters) {
    ++repeat;
    double xy = _x * _y;

    _y = xy + xy + _y0;
    _x = _xx - _yy + _x0;

    _xx = _x * _x;
    _yy = _y * _y;
    _length_squared = _xx + _yy;
    if (!(_length_squared < 4)) break;
  }
  set_color(row + doing[0] * 3, repeat);
}
```

→ 399.15 seconds.

## MPI + OpenMP

一開始我的想法是將圖對高度平均的切割給每一個 Node，在使用 OpenMP 對原本使用 pthread 平行話的部分（選取要跑哪一個 height 的部分）進行平行化。

所以首先先計算出要運算的上下高度是多少：

```
int low_h = rank * (height / size);
int high_h = (rank == size - 1) ? height : (rank + 1) * (height / size);
```

再來，因為每一個節點會各自負責一些 rows 的運算，並且最後我的實作是把所有節點的運算都送到 Node 0 統整再寫入 png 檔，因此需要送出的 rows 是一個二維的陣列，但是 MPI 的 API 都只能對一維陣列使用，所以我採用先將一維陣列做 memory allocate 得到一段連續的記憶體位置後，再將這些連續的記憶體位置分配給二維陣列使用：

```
int local_h = high_h - low_h;
int cols = 3 * width;
png_bytep _rows = (png_bytep)malloc(local_h * cols * sizeof(png_byte));
png_bytep* rows = (png_bytep*)malloc(local_h * sizeof(png_bytep));
for (int i = 0; i < local_h; i++)
  rows[i] = &_rows[cols * i];
```

接著就可以使用 OpenMP 做原本的運算：

```
#pragma omp parallel num_threads(num_cpus)
{
  #pragma omp for schedule(dynamic)
  for (int h = low_h; h < high_h; ++h) {
    // ...
  }
}
```

對於 schedule 的部分，因為每一個 height 的工作量都不太一樣，因此這次沒有選擇 static 而是選擇 dynamic 的方式來使用，如此一來先運算完的就可以先繼續下一輪運算。

運算後，使用 `MPI_Gatherv` 來收集所有的 rows，根據 API 的定義先計算出必要的兩個 array：

- **recvcounts:** integer array (of length group size) containing the number of elements that are received from each process (significant only at root)
- **displs:** integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root)

```
int* all_h;
int* displs;
if (rank == 0) {
  // calculate every node's local_h

  all_h = (int*)malloc(size * sizeof(int));
  displs = (int*)malloc(size * sizeof(int));
  for (int i = 0; i < size; i++) {
    int lh = i * (height / size);
    int hh = (i == size - 1) ? height : (i + 1) * (height / size);
    all_h[i] = (hh - lh) * cols;
    displs[i] = lh * cols;
  }
}
```

再來就是進行接收與傳輸，接收的一維陣列宣告方式與上述提到的一樣：

```
MPI_Gatherv(_rows, local_h * cols, MPI_UNSIGNED_CHAR, _all_rows, all_h, displs, MPI_UNSIGNED_CHAR, 0
, MPI_COMM_WORLD);
```

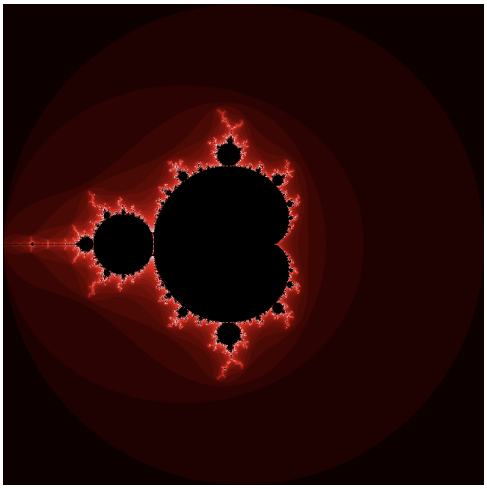
最後 root 節點進行寫檔結束。

→ 474.02 seconds.

### Load balanced

實作完 MPI + OpenMP 版本後，速度竟然比原本還要慢。才想到如果使用 height 直接平均分配的話，可能會因為圖形的特性使得不 load balanced。

因為以圖形來看，相鄰的 height 運算的次數應該是差不多的，因此改使用 height 輪流拿取的策略，也就是 node 0 計算 $0, n, 2 \times n, ...$，node 1 計算 $1, n + 1, 2 \times n + 1, ...$，其中 $n$ 為節點數量。



改為 load balanced 版本後，因為 `MPI_Gatherv` 收到的 rows 就會變成 height 是不連續的了，因此做了一個 mapping 函數將真正的 height 對應到他的接收順序（圖中的 `j` 迴圈即為按照順序接收到的 height，再倒過來即為寫入 png 的順序）：

```
mapping = (int*)malloc(height * sizeof(int));
int h = 0;
for (int i = 0; i < size; i++) {
  for (int j = i; j < height; j += size) {
    mapping[height - j - 1] = h++;
  }
}
```

最後果然提升了不少速度。

→ 316.54 seconds.

# Experiment & Analysis

## Methodology

### System Spec

學校提供的 Apollo Cluster。

### Performance Metrics

使用 `std::chrono::steady_clock()` 在 IO、通訊以及運算的前後都埋一個時間戳記，使用這些時間戳記的差總和以毫秒為單位的總 IO、通訊與運算時間。

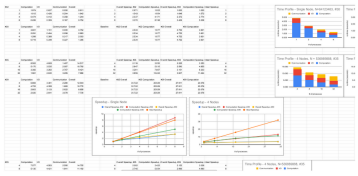
```
#include <chrono>

typedef std::chrono::steady_clock::time_point tp;

void start_span(tp& start_time) {
  start_time = std::chrono::steady_clock::now();
}

void end_span(tp& start_time, int& total) {
  total += std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::steady_clock
::now() - start_time).count();
}
```

## Plots: Scalability & Load Balancing

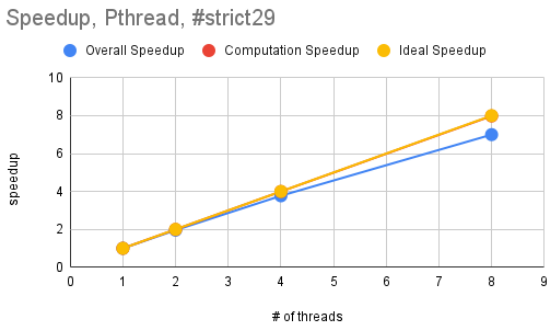
平行程式

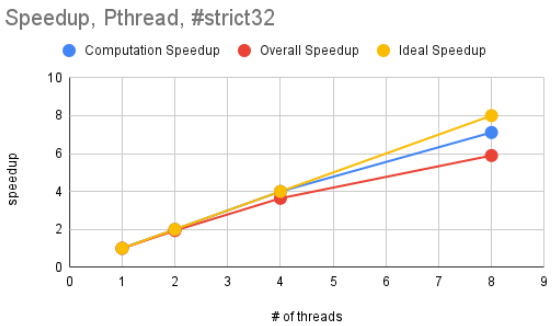
HW1 #32,Computation,I/O,Communication,Overall,Overall Speedup,
#32,Computation Speedup, #32,Overall Speedup, #30,Computation Speedup,
#30,Ideal Speedup 1,1.974,0.637,0.000,2.611,1,0.971,1.002,1.000,1.000,1

https://docs.google.com/spreadsheets/d/1gf9imf_HyQbHsUWwDIjr9HkK7K0ou1fX7357jjRJxfw/edit?usp=sharing



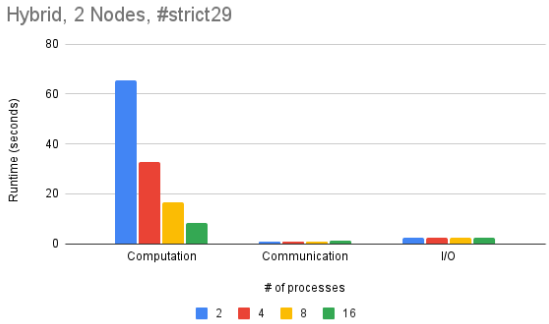
### Pthread + Vectorization

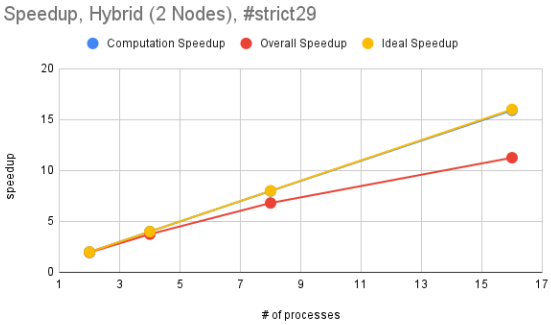


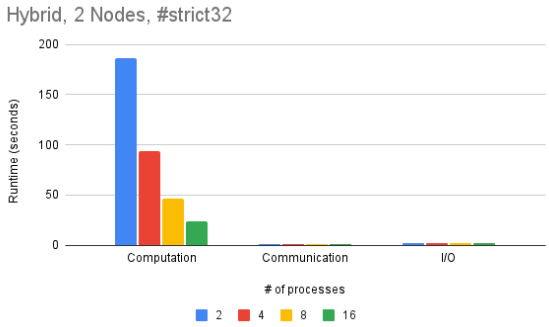



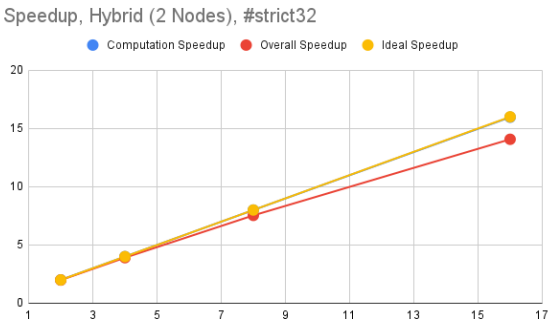

在 Pthread 版本中，可以看到 computation speedup 已經趨近於 ideal 達到 linear 的成長，但是由於 I/O time 並沒有變快，因此 overall speedup 是隨著 threads 的數量上升而趨緩的。

### Hybrid + Load balance



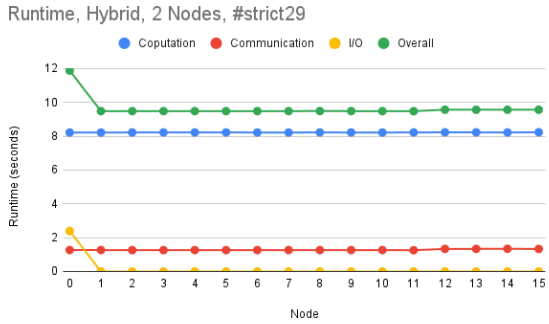



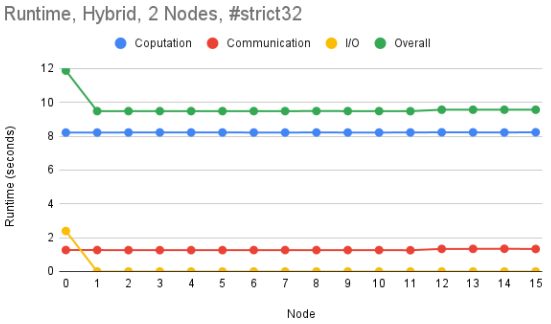




在 Hybrid 版本中，兩組測資隨著 processes 的數量變多：

- I/O 的時間花費基本上不變，因為只有一個 Node 會進行 I/O，並且寫入的檔案大小不變。
- Communication 的花費時間逐漸上升，這是因為需要參與溝通的節點變多了，需要更多時間來做初始化。
- Computation 的時間逐步下降，基本上有貼近 linear 的下降。

因此最後的 Speedup 來說，只看 computation speedup 的話，就算是多節點也可以做到 linear speedup。而 overall speedup 雖然被 communication 拖累，但是事實上 speedup 是很不錯（16 個 processes 時上圖有 11 倍，下圖有 14 倍）。




另外來看 load balance 的狀況，可以看到我將每個節點的 computation time、communication time 與 I/O time 畫成圖表，發現 computation time 與 communication 都是非常平均的。

# Discussion

### Scalability

這次的實驗 scalability 比起 hw1 時好很多。對於 pthread 版本，使用更多核心的 CPU 來工作大幅的提升了運算的速度，再加上 vectorization 的優化後，可以更加充分的利用 CPU 的運算核心，直接達到 linear 的 speedup。而對於 hybrid 版本，使用多個節點雖然使得 overall speedup 稍微下降，但是總體來說還是可以達到很好的效果，推測比起 hw1 時好的原因在於使用 MPI 溝通的數量很少，並且只有在運算完的最後才需要做一次溝通，因此 computation speedup 基本上沒有被影響。另外因為負載均衡，大大的降低了互相等待的時間，最後只需要將算完的結果送到節點 0 即可，因此也不需要花太多的時間（比起運算時間來說，這次的 communication time 是非常少的）。

### Load Balance

這次的 load balance 也是非常的平均，每個節點的運算時間基本上一樣，都只有 < 0.02 秒的差別。但是對於節點 0 來說，需要多負責一個寫檔的動作，所以都會比其他節點還晚結束。

## Experiences & Conclusion

這次作業最重要的就是學到 vectorization 的技巧，以前都不知道有這種加速運算的方式，非常感謝助教在 lab 時的補充，受益良多。

遇到比較大的困難是在收集 vectorization 的資料上，似乎網路上沒有很多這部分的教學，因此都是看著文件中的函數一個一個找到底有哪一些可以使用的。

## References


https://stackoverflow.com/questions/12495467/how-to-store-the-contents-of-a-m128d-simd-vector-as-doubles-without-accessing

https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#ig_expand=4269,4314,6082,4276,4274,6085,6137,3140,186,130,1213&techs=SSE,

https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#ig_expand=3222,3222&text=__m128d

https://wdv4758h.github.io/notes/simd/algorithm.html

https://docs.microsoft.com/zh-tw/cpp/parallel/openmp/openmp-simd?view=msvc-170

https://www.univ-orleans.fr/lifo/Members/Sylvain.Jubertie/doc/SIMD/html/group__loadstoreops.html#gaa79a73543322f8

https://docs.microsoft.com/zh-tw/cpp/parallel/openmp/d-using-the-schedule-clause?view=msvc-170