Homework 2: Mandelbrot Set

107062120 姚潔恩

Implementation

I. Part A: Pthread

A. Overview

Part A 的基本架構非常直覺，首先我把 argument 以及 image 都放到 global，方便每一個 thread 都可以直接取用。接著在 main 中會用 pthread_create 讓 thread 平行的去處理各自負責的 work，最後在 join 回來。另外因為每個 thread 的工作是獨立的(不同 pixel)所以可以直接把結果寫進 global 變數 image，不用使用 mutex。

B. Partition the task

這次作業很強調 load balance，我採用的方法是很 naïve 的想法，就是將整體的 workload 按照 row 來切。然後和 OpenMP schedule dynamic 有點像，每個 thread 一次就是拿一個 row 去做。具體實作是使用一個 global 變數 row，代表現在工作進行到哪一列，而這個變數會用 mutex 保護以確保正確性。當一個 thread 要去拿新工作時，需要先取得 mutex lock，然後再把 row 的數值存到 local 變數 local_row 並把 row++，而 local_row 就是接下來 thread 要負責的計算的列。當一個 thread 讀取 row 時發現 row=height，這時就表示工作都做完了。

C. Vectorization

這次作業我有使用 vectorization 的技巧。我使用的是 Intel® SSE，其 128bit 的 vectorization 可以一次計算兩個 double，因此我的作法就是一次進行兩個 pixel 的運算。因為被 vectorize 的兩個 pixel 不一定會同時結束，因此我的作法是當其中一個 pixel 先結束，就會拿下一個 pixel 進來運算(當然也可能兩個 pixel 同時結束，那就會拿兩個新的 pixel 來算)。vectorization 計算迴圈的終止條件就是當其中一個 pixel 結束，要去拿下一個 pixel 的時候如果發現一整個 row 都結束了，就會跳出 vectorization 的迴圈。但這時要注意的是另外一個一起被 vectorize 的 pixel 是不是也同樣完成計算，如果沒有的話就會另外單獨完成那個 pixel。實作上有個小細節和助教在 lab 中的講解不同，就是我發現__m128d 這個 data type 可以把它看作一個 array，可以直接用 index 給值與取值。因此我就沒有使用 load/store 相關的 API，而是直接用[index]進行操作。

II. Part B: Hybrid

A. Overview

Part B 有多個 process 以及多個 thread，我的作法是先將 workload 平均分給每個

process。假設有 n 個 process，我的分法是依照 row 間隔的分，舉例來說就是 rank0 會拿到 row[0, 0+n, 0+2n, …]，rank1 會拿到 row[1, 1+n, 1+2n, …]，以此類推。接著在每個 process 中的作法就和 Part A 幾乎一樣，差別只在於我們不用自己做 scheduling，可以直接用 OpenMP 的 dynamic scheduler 做到 Part A 中一次拿一個 row 的效果。每個 process 做完以後再把結果傳到 rank0 並在 rank0 寫 png。

B. Communication

為了減少 communication 的次數，我的作法是等到每個 process 各自的工作全部做完以後再用一次 communication 傳給 rank0。因為每個 process 所負責的 pixel 是沒有重複，因此我可以直接使用 MPI_Reduce，將所有 process 的 image 陣列用 MPI_SUM 加起來並將結果存在 rank0 (我在 allocate image 的 memory 之後會把每一個 index 的值都初始化為零，這樣 process 沒有負責的 pixel 就會是零，因此 MPI_SUM 就會得到正確的結果)。

III. Other Optimization

A. Reuse calculation result：

觀察發現原本 sequential code 的 while 迴圈裡面 x/y 平方都重複算了很多次。稍微改寫一下迴圈就可以 reuse 上一個 iteration 就算過的 x/y 平方，以減少乘法的次數，也不用多 declare 一個 temp 變數。

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

B. Compiler optimize：

首先我找到-ffast-math 這個 flag，他會打開許多對 floating point 的運算優化的 flag，但是會犧牲一些準確度或是正確性。我發現搭配 g++的話在有一些測資會因為精準度問題出錯，但是如果改成 clang++的話就不會有問題，可能是兩個 compiler 對這個 flag 的優化實作不同。然而在 Part B 換到 mpicxx 的時候用 clang++編譯會有錯誤，因此在 Part B 中我使用 g++，並一一嘗試-ffast-math 所有打開的 flag，檢查是哪一個 flag 會造成精準度問題。最後發現只要不要打開-fassociative-math 就不會有精準度的問題。

Experiment & Analysis

I. Methodology

A. System Spec

實驗環境使用課程提供的 server。實驗的測資使用 strict21。用最原始的 sequential code 跑需要約 266 秒。

B. Performance Metrics

實驗的計時方式是使用 clock_gettime。量測的區間從程式一開始直到 write_png 之前，也就是 write_png 的時間不會計入。在 Part A 的部分我只有量測 Total time，在 Part B 的部分我有量測 Communication time (有使用 MPI 的資料 communication routines 的區塊)以及 CPU time (Total time – Communication time)。另外在 Part B 中每個 process 都會各自計算時間，而我 report 的數據是 rank0 的 process 測出來的時間。最後為了取得更穩定的實驗數據，若沒有額外說明則呈現於圖表上的時間都是跑三次實驗的平均。

II. Plots & Discussion

A. Sequential Code Optimization

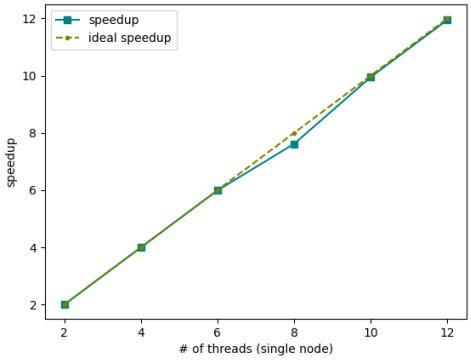
我的實作中有使用一些優化如 vectorization、改寫迴圈的邏輯以減少一些多餘的 instruction、compiler 的優化。這些優化對於 sequential code 也同樣適用。因此可以發現我的 sequential code 跑出來的速度相比原版的 sequential code 有約 2.3 倍的 speedup。

原版的 sequential code	優化的 sequential code	speedup
266.58	114.547402	2.3272461474

註：此優化的數據是 Part A 的結果。使用的 compiler 是 clang++並加上-ffast-math 的 flag。Part B 使用 g++以及其他的 flag 編譯出來的速度會比較慢(141.470585)。

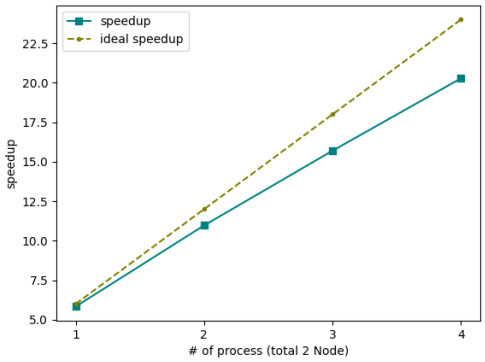
B. Scalability

1. Part A: Pthread

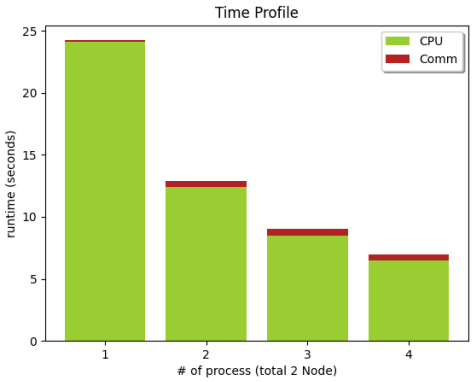


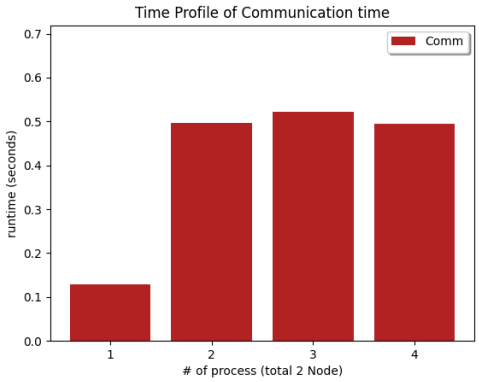
從以上實驗可以看出來 Part A 的 scalability 非常的好，在 single node 的情況下實驗結果幾乎完美的貼合理想的 speedup 曲線(也就是多幾個 thread 速度就快幾倍)。推測是因為這次的 task 算是一種 Embarrassingly Parallel 的工作，每一個 pixel 的工作都是獨立的，本來就很適合平行化。再加上使用 pthread 沒有 communication 的 overhead，因此在 loading 很平衡的情況下，就可以達到接近完美的 speedup。

2. Part B: hybrid



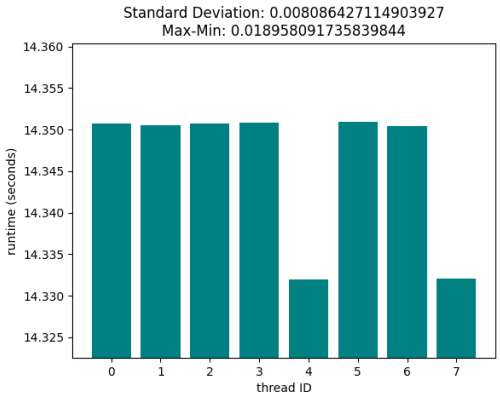
Part B 的實驗為固定一個 process 有 6 個 core，然後 process 數從一開始逐步增加(node 數都是兩個)。從實驗結果可以看出 Part B 的 scalability 也不錯，但是沒有 Part A 中那麼好的 scalability，主要原因是使用 MPI 的架構會產生 communication 的 overhead。從下圖可以看出在 process 數大於一的情況下，就會需要比較多的 communication time。因此除了 process 數為 1 的實驗 speedup 還可以貼近完美，其他多 process 實驗的 speedup 就會比理想值低一些。

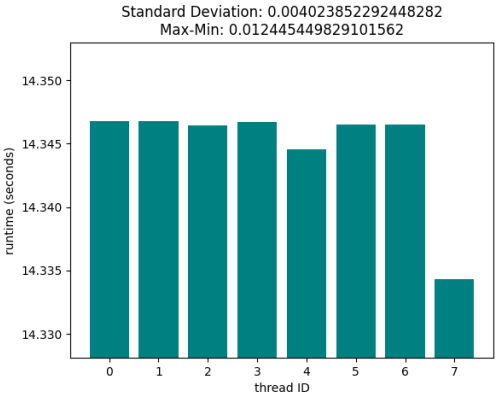




C. Load Balancing

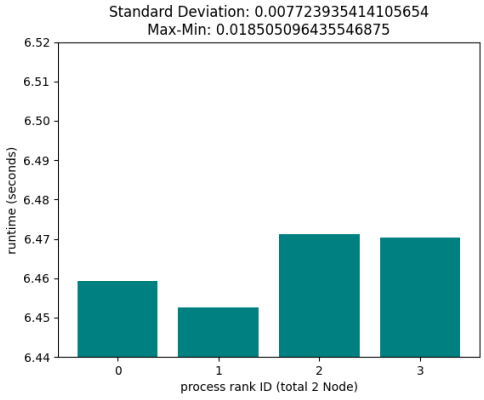
1. Part A: Pthread

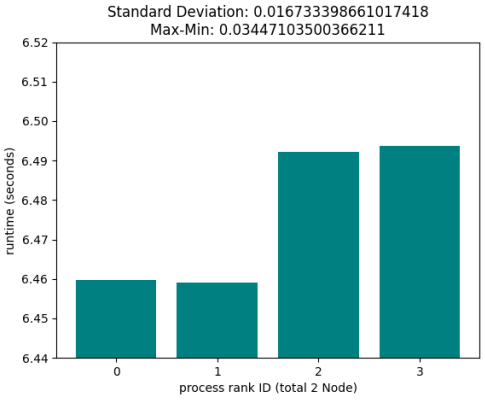




以上實驗結果為跑了兩次八個 thread 的實驗，並分別計算每一個 thread 各自執行的時間。從上圖可以發現 load balance 相當的好，每個 thread 的執行時間非常相近，兩次實驗八個 thread 執行時間的標準差都小於 0.01 秒，最久的時間和最短的時間也非常相近。這證明雖然只是簡單的用 dynamic 的方式一次拿一個 row 還是有很好的 load balance 效果。

2. Part B: hybrid





以上實驗結果為跑了兩次 4 個 process 的實驗(node 數皆為 2)，並分別計算每一個 process 各自執行的時間。可以看到這邊的 load balance 表現依然優異。最長的時間和最短的時間也是非常相近。

Conclusion

這次的作業有許多優化的地方，但是很特別的是大多數的優化都是對 sequential code 的優化，優化完後竟然可以有高達 2.3 倍的 speedup。對於平行的 load balance 處理反而沒有特別的效果，就如實驗呈現的直接用最 naïve 的方法用 row 做 dynamic 的分配就可以有不錯的效果。所有優化中又屬 vectorization 最為麻煩。因為每個 pixel 各自的計算迴圈是有 dependency 的，一開始我還想不太到要怎麼

使用 vectorization。後來才想到可以一次計算兩個 pixel，最一開始的寫法是將一個 row 內的 pixel 兩兩一組計算，後來才又想到一個 pixel 結束就可以馬上拿下一個 pixel 進來計算。雖然過程很麻煩，但是看著速度越來越快也是很有成就感。