Parallel Programming
HW2: Mandelbort Set

111065510 黃韋慈

A. Implementation

A.a Overall

這次的作業要分別使用 Pthread 與 OpenMP+MPI 實作出 Mandelbort Set，因此以下會根據兩種版本分別進行探討。

A.b Pthread

A.b.1.1 取得可用的 CPU 數

透過 sched_getaffinity()取得可用的 CPU 數，並以 CPU 數決定 thread 數。

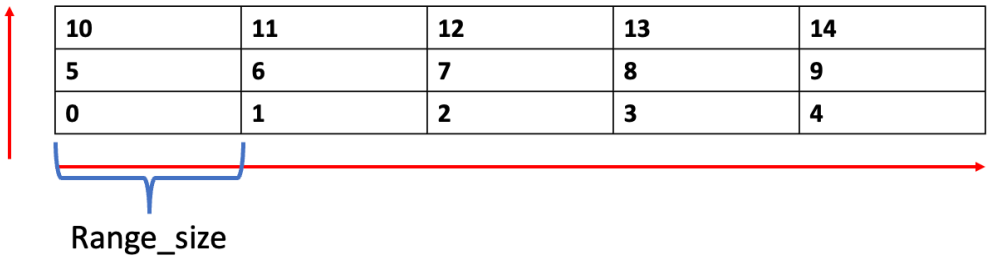
A.b.1.2 配置要輸出的圖片的記憶體

根據輸入的參數決定輸出圖片的記憶體大小，並且由於是 Pthread 每個 thread 都可以存取，因此直接配置要輸出的大小即可。

A.b.1.3 執行 Mandelbort Set 的計算

透過 pthread_create()建立 thread 並執行 Mandelbort Set 計算。

A.b.1.3.1 執行 partition 方法



圖一

考量到只有一個 Node 多個 thread 的情況及 thread 工作的特性，因此決定採用這種類似動態分配的分配方法。我的分配方法簡單來說就是如圖一所呈現，會先由左至右的分配，再由下依序往上，因為只有一個 Node 因此不太需要考慮到要切

高度，全部都以 rang_size 為單位處理即可．也就是說在每個 thread 每次進行 Mandelbort Set 的計算前，都會先去取得這次要處理的 workload，而因為 thread 會共用記憶體，因此這邊靠 pthread_mutex_lock 去確保不會有 thread 拿取重複的 workload，而每次 thread 都會拿取一開始設定好的量當作這次的 workload，當列的方向取完後，便會往上一個並退至最左邊，一直重複以上動作直到紀錄目前取用的高度的指標超過最後要輸出圖片的高度，便表示所有的工作皆已完成．

A.b.1.3.2　使用 Vectorization 進行加速

而為了能夠進行加速，我使用 lab 所教的方法進行．而因為我會先做完橫軸的部分，因此透過 Vectorization 將 thread 要計算的 pixel 兩兩當成一組進行(_m128d 可存兩個 double)，而因為要同時紀錄現在進行到哪裡，因此有額外使用幾個 array 分別去紀錄像是 repeat、橫軸目前進行到的位置等等，以便後續進行運算．這邊比較特別的是在實作過程中，因為 length_squared 需要同時參與 Mandelbort Set 的計算，又要當作迴圈判斷的指標，因此這邊試過許多方法如_mm_storeu_pd、_mm_cvtsd_ff，希望能將使用_128d 進行計算的 length_squared 轉成 double 方式以進行迴圈判斷，但一直無法成功，後來發現使用 c++中的 union 資料型別可以解決此問題，因為存在 union 中的資料會共享變數，因此就可以透過在其中分別宣告_128d、double 型態以直接取用所需要的數．

A.b.1.3.3　計算 Mandelbort Set

這邊分別透過 while 迴圈去控制 thread 是否有完成它所取的 workload，並且如同 sequential code 一般以 Vectorization 的方式進行運算，這邊需要注意到因為一次處理兩個值，因此 repeat 與將結果存入 image 的動作都需要做兩次．而為了使

整體計算更快（也是發現只做到這裡會有許多 testcase 超過時間），因此這邊如同在 HW1 所採用的排序方法類似，若是有其中一個先行計算完畢，那接續便再透過 for 迴圈依序檢查，先準備下一個要參與計算的值，包括移動目前橫軸進行的位置、歸零 repeat 等等一開始計算前所做的動作，而其中因為要一次處理兩個值並進行判斷，因此分別使用_mm_loadl_pd、_mm_loadh_pd 去分別將 x、y、length_squared 的值清零．最後再如同 lab 所教會再處理剩下的值，便完成計算．

A.b.1.4 確定每個 thread 做完運算

透過 pthread_join()確定每個 thread 都做完運算．

A.b.1.5 寫入圖片

而因為在 thread 執行的過程中共享記憶體，因此這部分如同 sequential code 直接寫入圖片即可，不需額外再做其他處理．

A.c Hybrid

由於會有不同的 process 參與因此需要透過 MPI 進行，且需要分配高度以進行．

A.c.1.處理 Process 與 Height 分配

A.c.1.1Height< Process

這邊如同 HW1 一樣透過 MPI_Comm_group()建立 MPI_COMM_WORLD 的 group，並在其中使用 MPI_Group_range_incl()在 MPI_COMM_WORLD 的 group 取用需要的 process，以處理不會用到的 process，再搭配 MPI_Comm_create()建立一個新的 communication，而沒有在新 communication 中的 process 便會顯示 MPI_COMM_NULL．如此最後 Process 的數便會與 Height 一樣，就會到 A.c.1.2 的情況．

A.c.1.2Height= Process

一個 process 處理某一高度的橫列．

A.c.1.3Height> Process

這邊也如同 HW1 分配方法一樣，若是 Height> Process，表示一個 Process 會處理多個列．因此這邊透過 height/process 的方式計算每個 process 平均要處理的列數，而剩餘的數則從第一個依序分配給其他的 process 進行處理，也就是說如果 rank id 小於剩餘的數的 process 會再多處理一筆資料，如此便能將剩餘的數都處理完畢，處理的資料量最多也只會差到一列．

A.c.2.配置不同 process 所要處理的 pixel 的記憶體

由於每個 process 要處理的資料量不同，因此這邊會根據上述計算的列數配置屬於各個 process 的記憶體大小．

A.c.3.執行 Mandelbort Set 的計算

A.c.3.1.1　執行 partition 方法

由於會有多個 process 參與計算因此這邊不能如同 hw2a 的方法，都動態的取用 workload，因為後續組裝資料會很麻煩．因此我們需要對高度進行處理，也就是說每個 process 會被分到不同的列，而在那列中 thread 還是如同 hw2a 的方式一般會取用一定的量當作這次的 workload，而在這裡一開始採用的是順序取用的方式，也就是 rank 0 的 process 會順序處理它所需要處理的列數，但後續發現這樣在最後 rank 0 收回資料並重新組裝的時候會有很大的困難，因此改用跳著取用的方式以進行．也就是說 rank0 會處理 0、0+Process Size、0+Process Size*2…..的列，而在那列中就如同 hw2a 方式一樣，當橫向被取用完畢就會再往要處理的下一列走，這邊為了防止 thread 拿取到重複的資料，透過 omp_set_lock 進行鎖住．

A.c.3.1.2　使用 Vectorization 進行加速

這邊與 hw2a 的方式基本相同．

A.c.3.1.3　計算 Mandelbort Set（存入資料的位置）

這部分主要也與 hw2a 的方式基本相同．主要是差在由於現在是由各個 process 一起進行處理，因此在一開始便有宣告放置屬於這個 process 記憶體的大小，因此在存放資料時需要將 y 高度的部分去除以 size 數，以還原這個值在本地 process 所存的位置．

A.c.3.1.4　收回各個 process 計算結果給 rank 0

而由於每個 process 所處理的資料數不相同，因此這邊使用 MPI_Gatherv() 以將資料收集到 rank 0，這邊需要特別注意因為 MPI_Gatherv()方法會需要多帶入不同的參數，像是 recvcounts、displs 等等，recvcounts 主要用來記錄每個 process 收回的量，而 displs 則紀錄最後資料收回來後的起始位置，因此在使用 MPI_Gatherv()收回資料前需要先算好 recvcounts 與 displs 的值，再進行收集．

A.c.3.1.5　rank 0 處理收到的資料並寫入圖片

在 rank 0 收集完資料後，由於收集到的資料是分別由各個 process 來的，因此如果想要使用 sequential code 中寫入圖片的方法，需要重新將資料重組，這邊我透過剛剛所設定的 displs 以取的不同 process 的起始值再使用變數分別取用那個 process 中的 thread 所處理的工作的結果，最後再將收到的集合 array 中取值給最後要寫入圖片的 array 以完成資料的重組．

B. Experiment & Analysis

B.a Methodology

B.a.1.System Spec

所有程式皆在課堂上所提供的設備進行．

B.a.2.Performance Metrics

B.a.2.1 Pthread

測量 pthread 時，我透過<sys/time.h>中的 gettimeofday()以記錄時間，他會將時間記錄在 timeval 內，最後使用其中的 tv_sec 及 tv_usec 分別計算出每個 thread 所處理的時間(computation time)。最後再將 computation time 進行平均，以得到較為精準的 computation time．最後在觀察 Load balance 時，分別取出在 12 threads 一同處理時每個 thread 所要處理的時間就可以看出 Load balance 的程度。

B.a.2.2 Hybrid

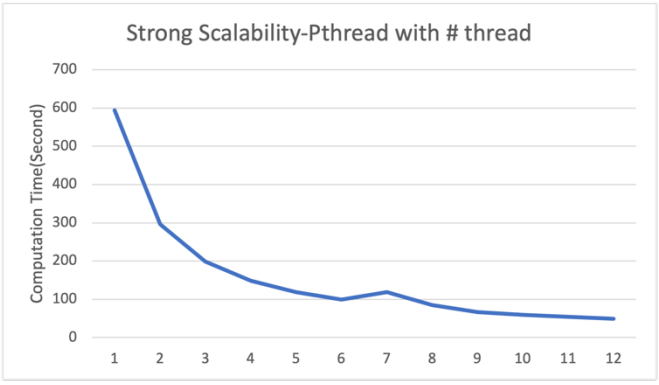
分別在 MPI_Init()後與 MPI_Finalize()前加上 MPI_Wtime()並相減，便可以得到整的程式的 runtime。

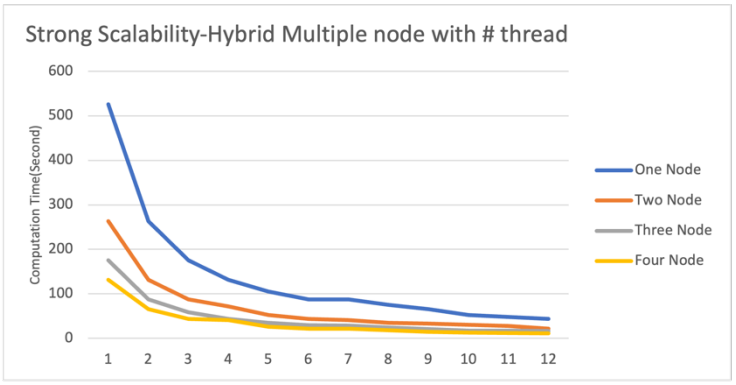
測量每個 thread 的 computation time 是在要計算的程式的前後加 omp_get_wtime()，再進行相減就可以得到 thread 的 Computation time，最後再依照製圖需要加以平均以得到較為精準的 Computation time。而在製作 Load Balance 時便是將不同 process 相同 thread id 的值加以平均得出．

B.a.3.Plot: Strong Scalability & Speedup Factor & Load Balance

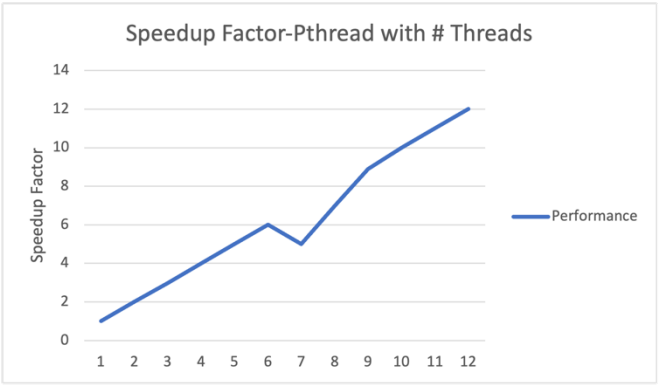
由於寫入圖片的方式都如同 sequential code 方式一樣，因此這邊不會將 I/O time 一起納入討論．這邊使用 test case 中的 strict34.txt 進行以下測驗，選擇此 test case 的原因在於當執行 judge 指令時可以發現此 test case 耗時最久，因此應該能較為清楚的表現數量不同的 thread 與 process 間的差距．

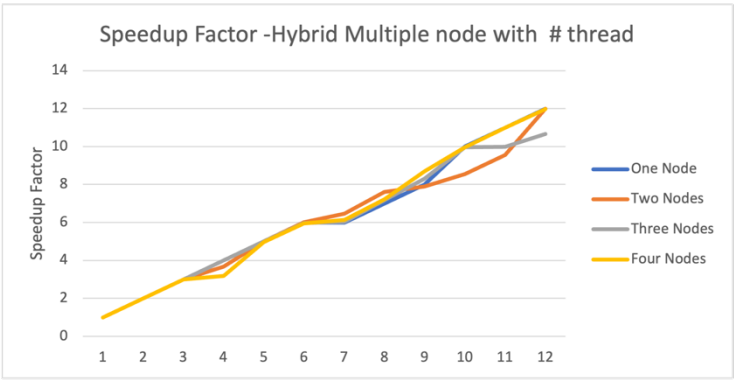
B.a.3.1 Strong Scalability



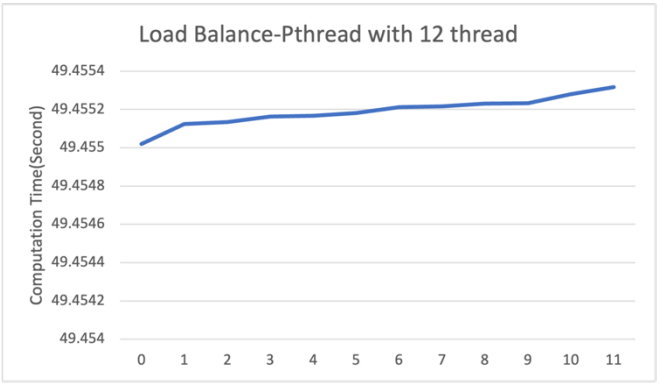


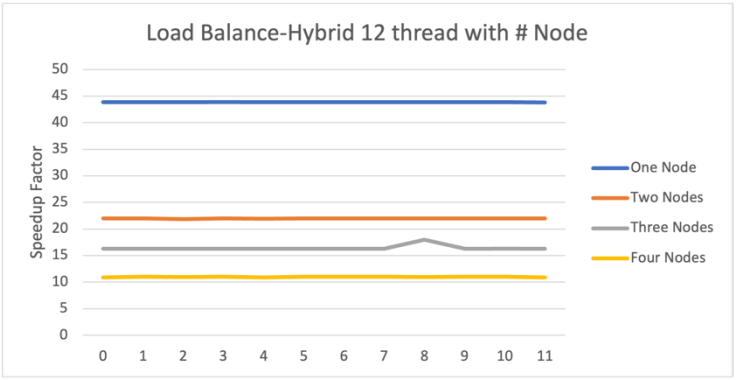
B.a.3.2 Speedup Factor





B.a.3.3 Load Balance





B.a.4.Discussion

B.a.4.1 Pthread

B.a.4.1.1 Strong Scalability

可以清楚看到當 thread 數量增加時，所耗的 Computation time 從原本需要近快 600 秒下降到約 50 幾秒的程度，雖然在 thread 7 時有些微的上升，但整體的 Strong Scalability 還是十分不錯。

B.a.4.1.2 Speedup Factor

可以看到當在 thread 7 時，表現下降許多，由於在 pthread 版本中採用的是動態分配的方式，因此判斷有可能是每個 thread 的 loading 不太相同所造

成的．

B.a.4.1.3 Load Balance

可以看到雖然 Load Balance 似乎沒有想像中平滑，但那是因為間隔都差距在 0.0004 之間，基本上已經趨近於完全相同了，表示每個 loading 都十分平均．

B.a.4.2 Hybrid

B.a.4.2.1 Strong Scalability

可以觀察到 Computation time 隨著 process 數增加下降也跟著降低，雖然在一個 node 時 thread 7 時也有些微的上升，但隨著 node 數的增加其實都趨於正常，整體 Strong Scalability 還是十分不錯的．

B.a.4.2.2 Speedup Factor

可以觀察到整體的 Speedup Factor 都不錯，尤其是當使用 4 個 node 時更是趨近於理想狀態，雖然在後續 2、3 個 node 的情況下沒有如 4 個 node 時表現良好，但依然呈現正常的的上升，在某些時候的下降可以也是因為雖然每個 process 處理的資料量最多相差一列，但每列所執行的難度可能會有所差距所造成的．

B.a.4.2.3 Load Balance

可以看到不論是在有多少 node 的情況下，loading 都十分的平均，只有在有 3 個 node 的情況下稍微上升一些，原因可能也是因為處理的難易度有差所造成的．

C. Experiences/Conclusion

雖然其實我還是不太了解 Mandelbort Set 算法的意義，但也因為這樣我能更專注於思考要如何將原本既定的 sequential code 改成不同的方式進行．在一開始都很粗糙的學著寫 lab 的方式去改 code，但後來發現這次的作業真正的意義是要讓我們思考要如何使用 Vectorization，透過 Vectorization 其實更能讓我了解到有關硬體的應用，像是_m128d 裏面可以存兩個 double 等等的內容都是以前從未想過的，也能更讓我了解 c++的使用，像是原本只了解基礎的 struct 資料結構型態，但因為這次程式的需求以及 Vectorization 取用功能的失敗，讓我思考是不是其實可以從原本程式語言的角度去換一種寫法改用 union，這些都對我影響十分巨大，因為從前不論是寫專案其實都偏向找尋現有 library 去達成目的，漸漸忘記思考其實能夠往基礎思考透過語言的特性去達成．