

# Parallel Programming

## Homework 2 Report: Mandelbort Set

108062605 呂宸漢

### A. Implementation

pthread(without vectorization) : hw2a\_basic.c

1. 利用 `sched_getaffinity(...)` 取得可以用的 core 數，再以 core 數決定 thread 數，因為若 thread 數大於 core 數時，CPU 必須做 context switching 使 thread 達到 logical parallel，可是 context switching 需要時間，且對於 core 而言是 sequential 在執行不同的 thread 而已；若 thread 數小於 core 數則會有 core idle，造成運算資源的浪費。所以當 thread 數等於 core 數時，就已經完全利用所有的運算資源了。
2. 依照所輸入的引數 allocate 一個圖片需要的 memory 空間，並將該 memory 空間宣告成 global，讓每個 thread 都可以 access。
3. 利用 `pthread_create(...)` 建立 thread 並執行 mandelbort set 運算。我的 partition 的分法(Figure 1)是用 thread id 決定該 thread 要計算的 x 軸方向的 pixel，也就是每個 thread 都要計算每個 row，不過不用計算整個 row，而是計算 row 內的特定 pixel 而已。在分配 pixel 時則是以 round robin 的方式分配，就 load balance 而言，雖然是以 round robin 的方式分配，以機率性的方式讓 thread 平攤 row 內的計算，可是仍然會有一些 thread 先做完，其他 thread 還沒做完的 unbalance 情況。

若以 thread pool 實作，則可以讓已經做完的 thread 做其他 thread 還沒完成的工作，可是當每個 pixel 運算皆很快完成時，在負責 manage thread pool 的 thread 就會變成 bottleneck，會拖慢整體的時間。因此我用 static 的分配方法分配 pixel，雖然可能 unbalance，可是分配速度較快。

30	31	32	33	34	35	36	37	38	39
20	21	22	23	24	25	26	27	28	29
10	11	12	13	14	15	16	17	18	19
0	1	2	3	4	5	6	7	8	9

Figure 1. 每一格為 1 pixel，數字表示 pixel 編號，顏色表示不同 thread

4. 以 `pthread_join(...)` 當作 barrier 確定每個 thread 都做完運算，再寫入圖片。寫入圖片的方式與 sequential version 的寫入相同。

pthread(with vectorization) : hw2a.c

1. 利用 `sched_getaffinity(...)` 取得可以用的 core 數，再以 core 數決定 thread 數，因為若 thread 數大於 core 數時，CPU 必須做 context switching 使 thread 達到 logical parallel，可是 context switching 需要時間，且對於 core 而言是 sequential 在執行不同的 thread 而已；若 thread 數小於 core 數則會有 core idle，造成運算資源的浪費。所以當 thread 數等於 core 數時，就已經完全利用所有的運算資源了。
2. 依照所輸入的引數 allocate 一個圖片需要的 memory 空間，並將該 memory 空間宣告成 global，讓每個 thread 都可以 access。
3. 利用 `pthread_create(...)` 建立 thread 並執行 mandelbort set 運算。我的 partition 的分法(Figure 2)是用 thread id 決定該 thread 要計算的 x 軸方向的 pixel，也就是每個 thread 都要計算每個 row，不過不用計算整個 row，而是計算 row 內的特定 pixel 而已。在分配 pixel 時則是以 round robin 的方式分配，就 load balance 而言，雖然也是以 round robin 的方式分配，以機率性的方式讓 thread 平攤 row 內的計算，可是仍然會有一些 thread 先做完，其他 thread 還沒做完的 unbalance 情況。

30	31	32	33	34	35	36	37	38	39
20	21	22	23	24	25	26	27	28	29
10	11	12	13	14	15	16	17	18	19
0	1	2	3	4	5	6	7	8	9




Figure 2. 每一格為 1 pixel，數字表示 pixel 編號，顏色表示不同 thread，

最下面紅色框的為 packing 示意圖，淺色的有 packing、深色的沒有

由上圖可以看出，thread 0 必須 sequential 計算所有藍色的 pixel，thread 1 必須 sequential 計算所有紅色的 pixel，以此類推。為了使 thread 內也可以加速，我用 vectorization 將 thread 要計算的 pixel 兩兩 packing 成一組，讓計算時間長的 pixel 可以 overlap 計算時間短的 pixel，不過只限於同一個 row 內的 pixel。由於在 coding 跨 row 的 packing 時發生了一點問題，所以只在同一個 row 上 packing，沒有被 packing 的 pixel 數最多為  $O(t * h)$  ( $t$ : thread number,  $h$ : picture's height)。若 height 很大且 thread 數很多的話，將會損失不少可以平行的時間。

若以 thread pool 實作，則可以讓已經做完的 thread 做其他 thread 還沒完成的工作，而 pixel 的 packing 方式也會不一樣，可是當每個 pixel 運算皆很快完成時，在負責 manage thread pool 的 thread 就會變成 bottleneck，會拖慢整體的時間。因此我用仍然使用 static 的分配方法分配 pixel，雖然可能 unbalance，可是分配速度較快，packing 方式也比較簡單。

4. 以 `pthread_join(...)` 當作 barrier 確定每個 thread 都做完運算，再寫入圖片。寫入圖片的方式與 sequential version 的寫入相同。

hybrid(without vectorization) : hw2b\_basic.c

1. 呼叫 `MPI_Comm_size(...)`取得當前建立的 process 數(T)，並讀取引數取得圖片的 height(H)，利用兩者決定如何分配及處理資料。
  - i.  $H < T$  :  
利用 `MPI_Comm_group(...)`建立 `MPI_COMM_WORLD` 的 group，並用 `MPI_Group_range_incl(...)`在 `MPI_COMM_WORLD` 的 group 中蒐集需要的 process，再搭配 `MPI_Comm_create(...)`建立一個新的 communication，如此一來沒有在新 communication 中的 process 就會顯示 `MPI_COMM_NULL`，即使呼叫了 `MPI_Finalize()`後，也不會影響新的 communication。
  - ii.  $H = T$  :  
一個 process 分配一個 row。
  - iii.  $H > T$  :  
以 round robin 的方式分配 row 給不同 process。如此一來 process 間最多只差 1 row，機率性的使每個 process 間的 loading 較平均，雖然與 master-slave 的分法相比仍然 unbalance，可是分配的速度較快，而且 process 間的傳遞次數較少，不用擔心傳輸時間的問題。
2. 以分配到 row 數最多的 process 為基準 allocate 一個可以裝下  $row * width$  個 pixel 的 pixel array，用來儲存 process 的計算結果，再 allocate 一個可以裝下  $row * width * rank$  個 pixel 的 image array，用來儲存收回的資料。
3. 在分配完 row 後，就可以利用 openMP 對 column 做 parallel，也就是分配 row 裡面的 pixel 給不同 thread 執行(Figure 3)，在 openMP 內，我採用 dynamic 的 schedule 並設 chunk size 為 1，讓 openMP 動態分配 pixel 給不同的 thread 增加 parallel 的 performance。

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39

Figure 3. 每一格為 1 pixel，數字表示 pixel 編號，顏色表示不同 process，

由於 process 內的 thread 是動態分配的，在此就不標示 thread

4. 利用 `MPI_Gather(...)`收回各個 process 的計算結果給 rank 0，讓 rank 0 將資料轉換成圖片。寫入圖片的方式與 sequential version 的寫入大致相同，由於一開始是用 round robin 的方式分配 row 給 process，因此收回後的資料需要重組，在 `png_write_row(...)`前，必須先換算下一行資料的位址，才不會輸出成錯的圖片。

hybrid(with vectorization) : hw2b.c

1. 呼叫 `MPI_Comm_size(...)`取得當前建立的 process 數(T)，並讀取引數取得圖片的 height(H)，利用兩者決定如何分配及處理資料。

iv.  $H < T$  :

利用 `MPI_Comm_group(...)`建立 `MPI_COMM_WORLD` 的 group，並用 `MPI_Group_range_incl(...)`在 `MPI_COMM_WORLD` 的 group 中蒐集需要的 process，再搭配 `MPI_Comm_create(...)`建立一個新的 communication，如此一來沒有在新 communication 中的 process 就會顯示 `MPI_COMM_NULL`，即使呼叫了 `MPI_Finalize()`後，也不會影響新的 communication。

v.  $H = T$  :

一個 process 分配一個 row。

vi.  $H > T$  :

以 round robin 的方式分配 row 給不同 process。如此一來 process 間最多只差 1 row，機率性的使每個 process 間的 loading 較平均。

2. 以分配到 row 數最多的 process 為基準 allocate 一個可以裝下  $row * width$  個 pixel 的 pixel array，用來儲存 process 的計算結果，再 allocate 一個可以裝下  $row * width * rank$  個 pixel 的 image array，用來儲存收回的資料。
3. 在分配完 row 後，就可以利用 openMP 對 column 做 parallel，也就是分配 row 裡面的 pixel 給不同 thread 執行(Figure 4)，在 openMP 內，我採用 dynamic 的 schedule 並設 chunk size 為 1，讓 openMP 動態分配 pixel 給不同的 thread 增加 parallel 的 performance。

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39

Figure 4. 每一格為 1 pixel，數字表示 pixel 編號，顏色表示不同 process，

由於 process 內的 thread 是動態分配的，在此就不標示 thread，

上面紅色框的為 packing 示意圖

由上圖可以看出，雖然 process 內可以動態分配 thread，可是每個 thread 一次只能計算一個 pixel。為了使 thread 內也可以加速，我用 vectorization 將相鄰的 pixel 兩兩 packing 成一組，讓計算時間長的 pixel 可以 overlap 計算時間短的 pixel，由於只有對 column 做 openMP 的 parallel，因此只限於同一個 row 內的 pixel。

4. 利用 `MPI_Gather(...)`收回各個 process 的計算結果給 rank 0，讓 rank 0 將資料轉換成圖片。寫入圖片的方式與 sequential version 的寫入大致相同，由於一開始是用 round robin 的方式分配 row 給 process，因此收回後的資料需要重組，在 `png_write_row(...)`前，必須先換算下一行資料的位址。

## B. Experiment & Analysis

### i. Methodology

#### a. System Spec

所有程式皆在課程所提供的 cluster 上進行測試。

#### b. Performance Metrics

##### 1. pthread(without vectorization)

在測量 pthread 時，我原本使用 time.h 的 clock() 進行測量，結果每次測量的時間似乎都很不一樣，尤其是當 thread 數變多時，測出來的時間往往差了好幾倍，後來上網查後發現，clock() 如果用在 thread 裡面可以正確測量，可是用在 thread 外，因為不同 thread 使用的 core 都不一樣，所以 clock() 測出的基準也不一樣，導致誤差超級大。因此我改用 linux 內的 sys/time.h 內的 gettimeofday(...) 記錄時間，他會將時間記錄在 timeval 內，並可以用 timeval 變數內的 tv\_sec 及 tv\_usec 計算出準確的時間。我用這個測量整的程式的 runtime 及每個 thread 的 computation time，再將 computation time 平均，即可得到正確的 computation time，再取出每個 thread 的 time 就可以看出 load balance 的程度。

##### 2. pthread(with vectorization)

與 pthread(without vectorization) 相同。

##### 3. hybrid(without vectorization)

在 MPI\_Init(...) 後與 MPI\_Finalize() 前加上 MPI\_Wtime() 後將兩個值相減可以得到整的程式的 runtime。

而測量 computation time 就比較麻煩，因為 hybrid 版本的 thread 是用 openMP 自動產生的，因此需要用 omp\_get\_wtime() 加在 thread 要計算的程式的前後再相減即可得到 thread 所花的時間，不過因為 thread 是 dynamic 分配的，所以需要先取得 thread 的 id 才可以將 thread 自己的 computation time 全部記錄出來。

由於不同 node 計算的 row 數雖然差不多，可是計算的難易度可能也不同，因此也需要記錄不同的 node 計算完自己分配到的 row 的總時間，及自己內部 thread 計算的個別時間，全部加以平均才可以找到準確的 computation time。

##### 4. hybrid(with vectorization)

與 hybrid(without vectorization) 相同。

##### 5. Other

測試數據：iter: 100000, x0: -2, x1: 2, y0: -2, y1: 2, w: 800, h: 800

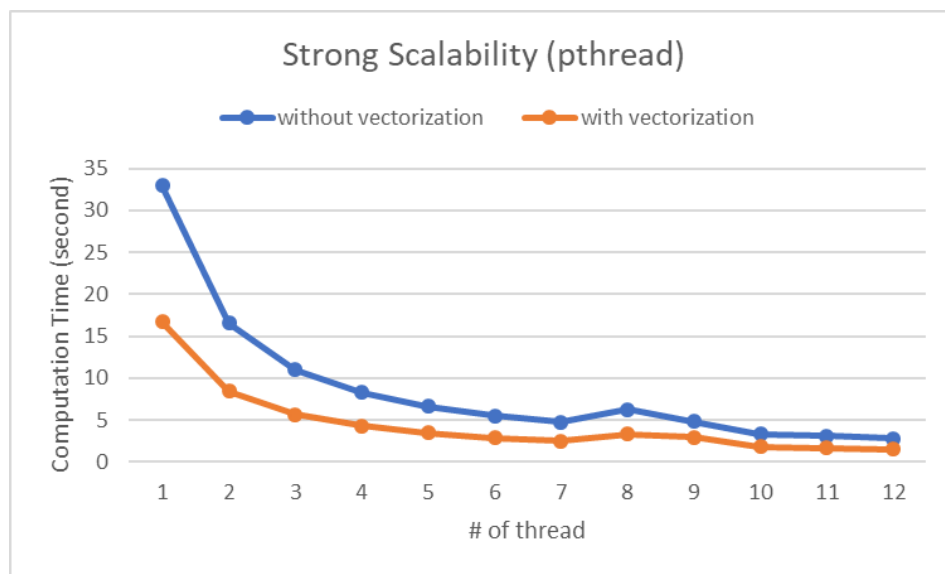
由於每次測試程式時，cluster 的狀況及 runtime 皆不相同，因此我將每種測試的組合都測試 3 次，並將 3 次所得到的時間取平均，降低不同狀態的 cluster 對實驗數據的影響。

## ii. Plots: Speedup Factor & Time Profile

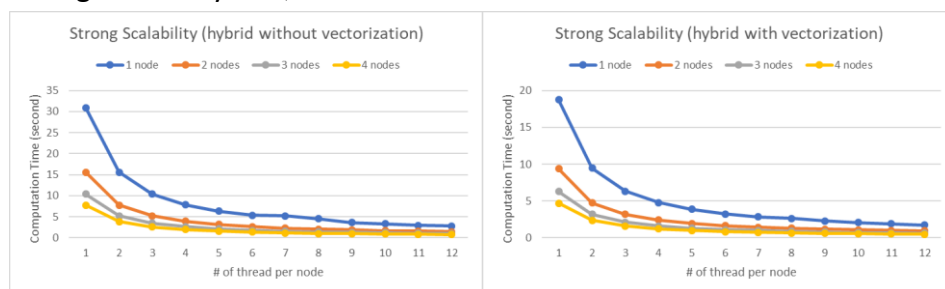
由於寫入圖片的方式都是 sequential 的寫入，無論多少 thread 或 node 所花費的時間都差不多，因此以下比較皆只看 computation time，不考慮 I/O 的時間。

### 1. Strong Scalability

由下圖可見，在 pthread 版本中 with vectorization(以下稱為 advance 版)的 computation time 皆比 without vectorization(以下稱為 basic 版)的長，且 advance 版的 computation time 幾乎都是 basic 版的一半，可以看出 vectorization 真的提升了兩倍的速度，不過在 thread 數為 8 及 9 的時候，可能是 loading 不平均的關係，在 computation time 略顯上升，不過以整體而言，兩者的 strong scalability 都不錯。



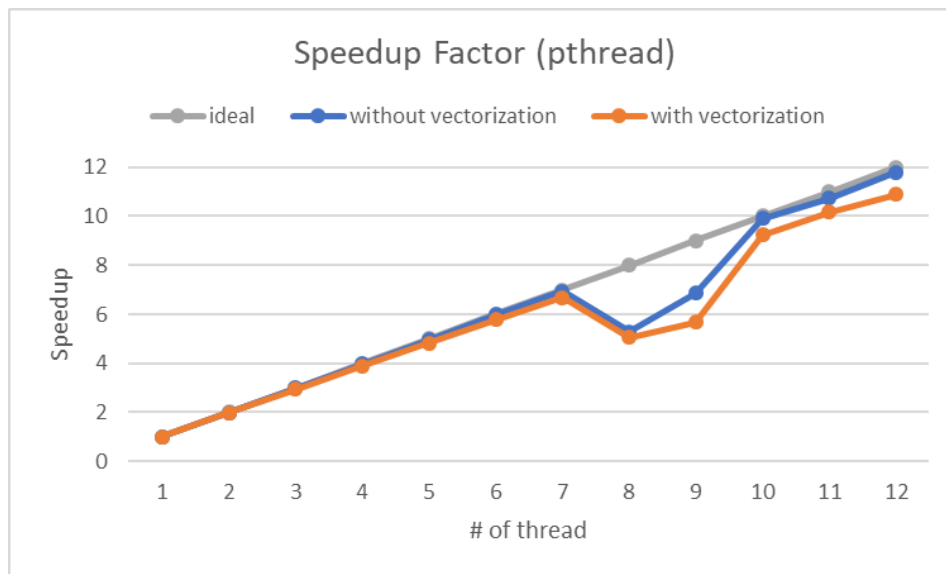
由下圖可見，在 hybrid 版本中，無論是 without vectorization(以下稱為 basic 版)或是 with vectorization(以下稱為 advance 版)，computation time 皆隨著 node 數增加而減少，而且 advance 版的 computation time 也都比 basic 版的更低，可以看出 vectorization 真的有提升運算的速度，不過與 pthread 版一樣，1 node 在 8 個 thread 時 computation time 似乎有變高一些，也間接表明不是當時 cluster 的問題，而是當時的 loading 不平均，不過以整體而言，兩者在任意 node 數的情況下，strong scalability 也都不錯。



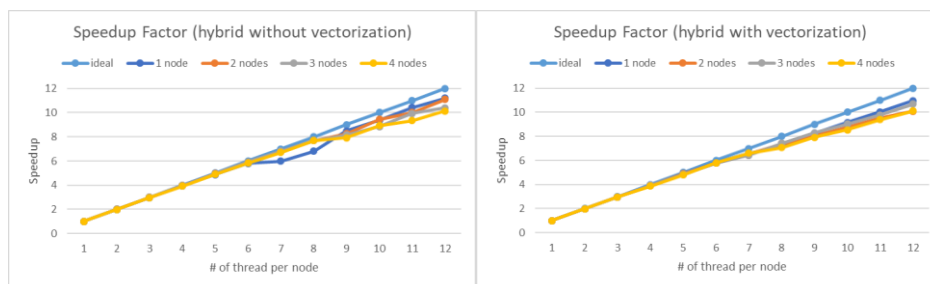


## 2. Speedup Factor

由下圖可見，在 pthread 版本中 without vectorization(以下稱為 basic 版)與 with vectorization(以下稱為 advance 版)在 7 個 thread 以下的 speedup 幾乎與理想狀況(ideal)相同，不過在 8 個 thread 及 9 個 thread 時，無論是 basic 版或是 advance 版的 speedup 皆偏離理想狀況許多，在大於 10 個 thread 時又與理想狀況相同，我認為是 8 個 thread 及 9 個 thread 在分配 row 時，雖然分配的數量差距皆不超過 1，但是他們的計算難度可能相差很多，以至於即使有 overlap 其他 pixel 的計算時間，可是仍然無法使每個 thread 的計算量平均，導致有些 thread 多累了整體的 runtime。

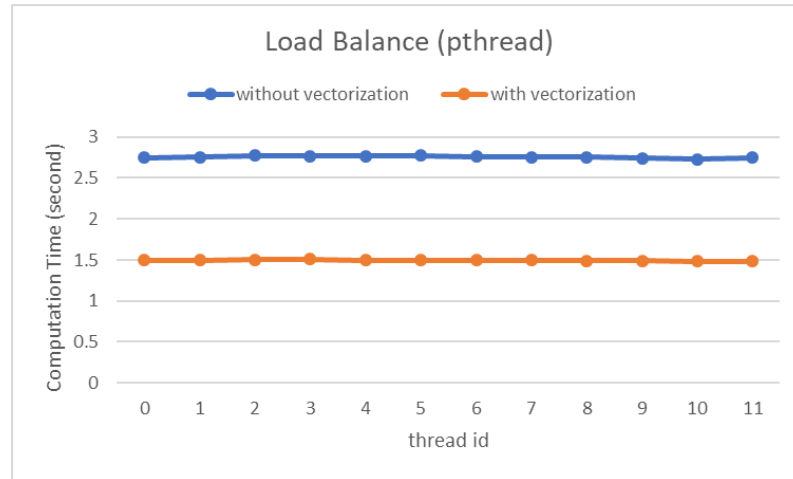


由下圖可見，在 hybrid 版本中 without vectorization(以下稱為 basic 版)與 with vectorization(以下稱為 advance 版)在 6 個 thread 以下的 speedup 幾乎與理想狀況(ideal)相同，不過 basic 版的 1 node 在 7 個 thread 及 8 個 thread 時，偏離理想狀況較多，在 9 個 thread 以上則又回歸原本的趨勢了。值得注意的是，在 basic 版與 advance 版的曲線可以看出，當 node 數變多且 thread 數變多時，兩者的 speedup 皆變差了，我認為也是因為不同 node 雖然分配的 row 數相去不遠，可是計算難度不同，導致先計算完的 node 要等待尚未完成的 node，拖慢了整體的 runtime。

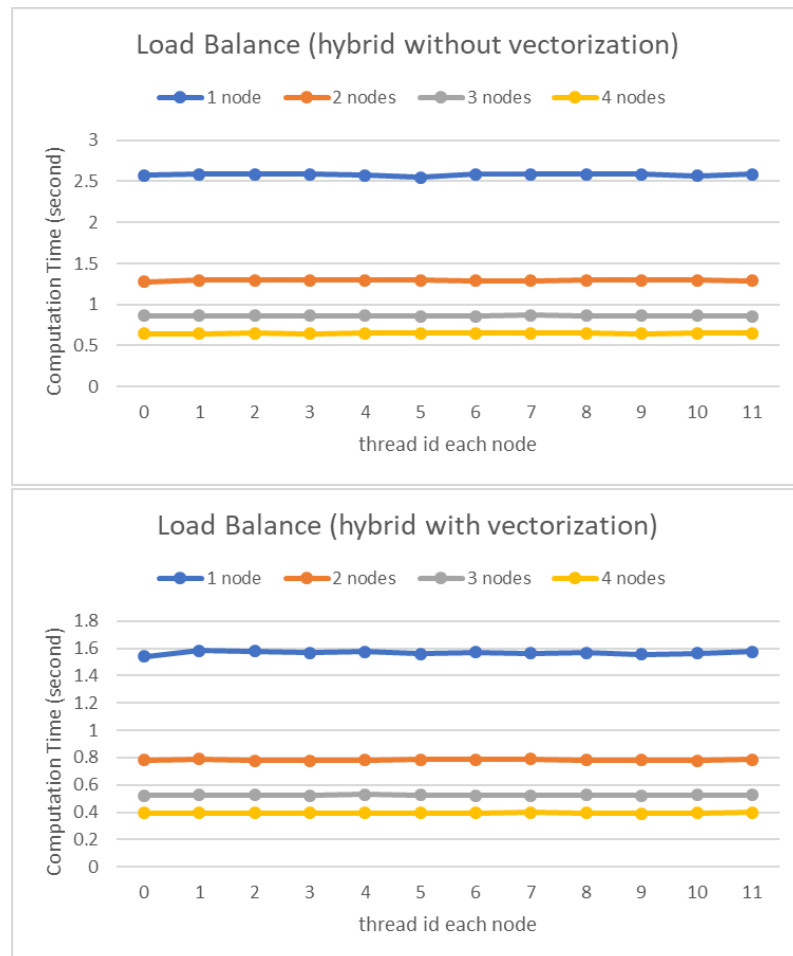


### 3. Load Balance

由下圖可見，12 thread 的 pthread 版中 without vectorization 和 with vectorization 的各個 thread 的執行時間近乎相同，表示每個 thread 的 loading 很平均。



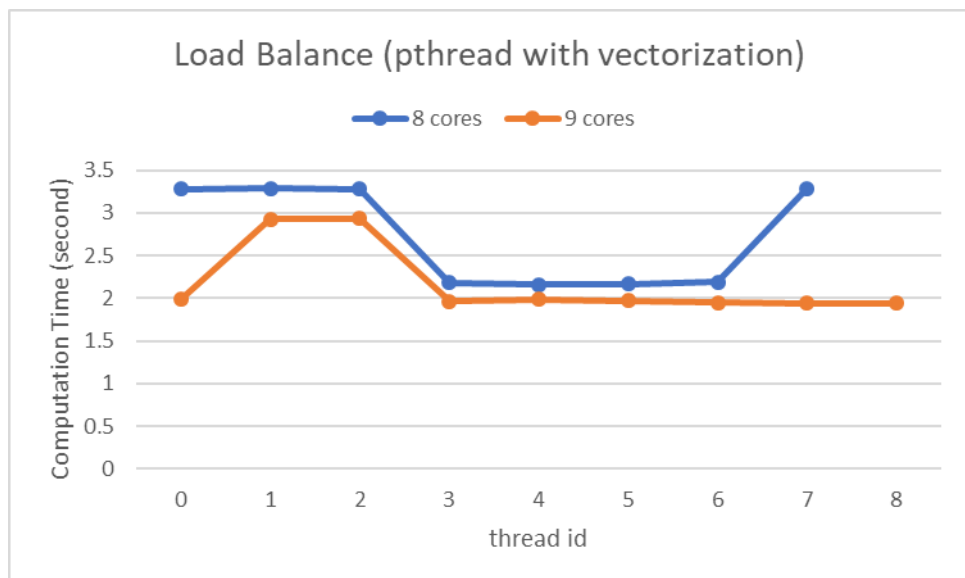
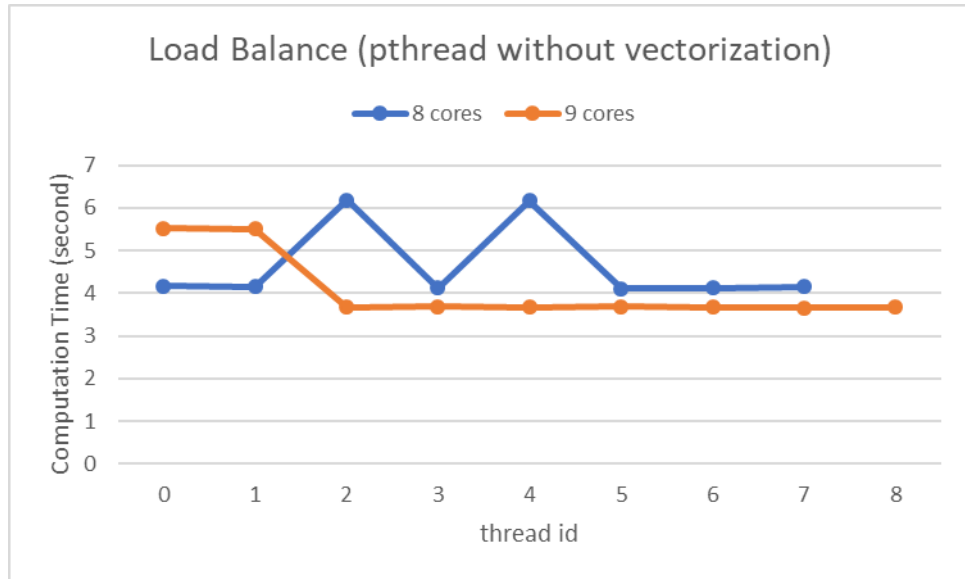
由下圖可見，12 thread 的 hybride 版中 without vectorization 和 with vectorization 的各個 thread 的執行時間近乎相同，表示每個 thread 的 loading 很平均。





#### 4. Other

在此測試 pthread 版本中 without vectorization 版與 with vectorization 版皆不 balance 的 case，也就是 8 thread 與 9 thread 的情況。在分別列出各自 thread 的執行時間後發現，最長時間的 thread 比最短時間的 thread 多了將近兩秒，就是這個原因導致這些 case 的 runtime 原本應該比前面 case 短的，結果卻與理想狀況完全相反，不但沒有加速，甚至比之前的 case 更長，因此在 pthread 內，不是所有的 thread 數皆可以 load balance。



### iii. Discussion

綜合以上測試結果與圖表顯示，pthread 版本與 hybrid 版本的 without vectorization 及 with vectorization 的 strong scalability 與 speedup factor 的趨勢雷同，以 load balance 而言，hybrid 版本則較 pthread 版本佳，因次以 general case 而言，hybrid with vectorization 版有較好的 performance。

#### 1. pthread without vectorization :

由於我 pthread 版本沒有實作 thread pool，因此 thread 沒有辦法動態分配 pixel 給先做完的 thread，導致 thread 的執行時間不一，有的提早結束，有的卻還有一堆 pixel 要計算，正是這樣的原因，導致有些 case 沒辦法與理想的 speedup 一樣，幾個 thread 就加速幾倍，runtime 會被時間最長的 thread 所制約，導致 case 加速不上去。

#### 2. pthread with vectorization :

而這個版本與 pthread 雷同，只是加了 vectorization 將兩個 pixel 同時計算，理想狀況下耗時長的 pixel 可以 overlap 耗時短的 pixel，只要兩者不相差太多，則可以達到兩倍的加速效果；若兩者相差較大，則 overlap 的時間就會變少，可以加速的空間就會被限制。

#### 3. hybrid without vectorization :

由於我 hybrid 版本是用 static 的方式分配 row 給不同的 node，因此可以將 pixel 分得更細，由之前的圖可以看出，當 node 數變多後，雖然 speedup 會遠離理想曲線，可是因為 pixel 切的夠均勻，反而比較不會有像 pthread 般明顯的 unbalance 出現，我認為在 node 數多且 thread 數多時，上面測試結果會顯示 speedup 偏離理想曲線是因為測試 case 太小了，導致有 extra time 或是 communication time 影響了 speedup 的結果，若是加大 test case 的 size 可能可以得到更接近理想曲線的數據，若與 pthread 相比，我認為 hybrid 的 performance 更佳。

#### 4. hybrid with vectorization :

而這個版本與 hybrid 雷同，是綜合 hybrid without vectorization 跟 pthread with vectorization 的版本。

值得討論的是在做 vectorization 的方式，因為 vectorization 是利用特殊的暫存器做運算，因此在寫的時候需要用組合語言的思維去架構，並減少 memory 的存取，盡量讓運算、比較及判斷皆在暫存器完成，因為在暫存器的運算比在 memory 快很多，因此要找許多不同的 function call 組合成自己需要的功能，才能發揮 vectorization 的最大效益。

### C. Experiences / Conclusion

其實這次作業如果沒有要用 vectorization 的話很快就可以寫完了，從 sequential 版改成 pthread 版及 hybrid 版就很像在 lab 把改 pi 的方式很像，只需要半小時到一小時就可以改好一個可以執行且 performance 不差的版本，比較困難的是利用 intel 及伺服器上有限的 function call 做 vectorization，雖然在 intel 的參考網頁可以找到很多 function call 可以使用，可是實際上伺服器可以支援的不多，必須一直想新的方式或是組合建構出自己需要的 function，還必須回到組合語言的思維去寫，盡量讓所有資料保持在暫存器內，避免 memory 存取以節省與 memory 傳輸的 overhead，在比較的時候也是在暫存器內比較，避免把值取出來到 memory 比較，就是因為這些小問題，我花了很多天在建構程式邏輯，查詢可以用的 function 並寫成 vectorization，在寫出來後真的感動，雖然花了很多時間，不過真的是值得的。

這次我也發現判斷式內的判斷順序也很重要，如果擺錯地方可能會讓執行次數與時間變得更久，我這次就有因為變數順序，讓一些 case 從原本 200 多秒變成只剩 20 幾秒，由此可見，放對判斷式內的判斷順序是很重要的，也很謝謝助教教我們 vectorization 的使用及有範例程式可以參考，真的對寫作業的時候幫助很大。