Homework 2: Mandelbrot Set

111062613 蔡鎮宇

1. Implementation

Pthread:

程式在一開始先知道能使用多少 thread 數，然後用一個 struct 把想要的參數傳給 Pthread_create，讓多個 thread 執行運算。
Load Balance:
使用 mutex lock 分配工作量，如果 thread i 獲得 lock，thread i 可以取用 x 個點出來運算，運算完後把資料寫在 image 的一維陣列上。其中的 x 是我們可以自行設定的值，在這邊我們不一次拿 1 row，而是用迴圈的方式獲得要運算的點，這樣可以避免剛好有 1 row 的運算量特別大導致某 thread 無法結束，而其他 thread 在等。
圖示為假設有 18 點要計算，3 個顏色代表 3 個 Process，一個 Process 一次取用 5 個點。

13	14	15	16	17	18
7	8	9	10	11	12
1	2	3	4	5	6

Vectorization:
把 sequential code 中的變數改成大小為 2 的 array，然後把 array load 進__m128d，接著在迴圈不停計算直到兩個點都達到終止條件。終止條件為複數的大小大於 2 或到達了 iteration 次數。因為上述的 x 值不一定是偶數（__m128d 可以放兩個 double 加速運算），因此除了用 vectorization 的運算外，還要再加上把最後一個點算完的手續。

Other efforts:

1. 有嘗試一次運算整 row 的寫法，但是效果明顯比上述的 Algo 來得慢。
2. 嘗試用不同 Flag 來編譯程式，最後是加上了-march=native -ffinite-math-only 會稍微快一點。
3. 嘗試用 Clang++編譯，但比 g++慢。

Hybrid:

Load Balance:
使用的是 Master-Slave 的模式，有一個 Master Process 負責把工作傳給其他 Slave。每一次傳 x row，接收 x row 的值，再把這 x row 寫在 image 的一維陣列上面。如果傳出 x row 會超出原本所要的計算總 row 數時，Master Process 會用 tag 的方式告知其他 Slave Process 終止。然後 Master Process 會把剩餘的 row 自行算完，最後再將 image 寫成 png 檔。
其中有幾個小技巧：

1. 寄出 x row 時，不需要真的用到 x * sizeof(int)的大小，只需要告訴 Slave 要計算第幾 row 就好，這樣 Slave 自然而然會知道要運算哪裡，如此可以省下記憶體空間。
2. 用 MPI_ANY_SOURCE 可以讓 Master Process 收到來自各個 Slave 的資料，用 status.MPI_SOURCE 的方式能知道是哪個 Slave 將資料送來，這樣 Master 也知道要回傳工作給誰

OpenMP:
每個 Slave 收到資料後，先把拿到的 row 去乘上圖片的寬度，以此來做為迴圈的長度，接著利用 dynamic、Chunk size 設成 1 的方式把要運算的點分給他能掌控的 thread 數量；如此一來只要某 thread 完成他的運算後就可以立刻去拿到下個點。
後來發現 Master Process 也能有多個 thread，因此也用 OpenMP 來一起運算。為了避免算到重複的 row，故將 row 的數字設成 critical section，使得 Master Process 傳給 Slave 的數字和 Master Process 運算用的 thread 要算的 row 不會重疊到。

Vectorization:
如同 Pthread 的 implementation。但因為這次運算的 thread 有 Master Process 的 thread 和 Slave Process 的 thread，所以變數的設置稍有不同。然後最後 Master 要處理的剩餘 row 數也會用 vectorization 加速。

Other Efforts:
1. 也有嘗試用 Pthread 創造出一個專門做分配工作的模式（已將 Master Process 能拿來運算的 thread 數減一），但成效沒有單純用 OpenMP 來得好。
2. 嘗試用不同 Flag 來編譯程式，最後是加上了-march=native 會稍微快一點。
3. 嘗試用 Clang++編譯，但比 g++慢。
4. 只要能用 vectorization 的地方都用上，但是會增加 code 的長度。
5. 因為有些迴圈會跑非常多次，因此 while loop 中的 operation 要嚴格控管，如果能減少乘法、load 和 store 的 operation，要盡可能減少。原本我把 loop 中的比大於 4 的行為全部用成 intrinsics 的比較方式，但後來發現沒有比多用一個長度為 2 的 loop 去比對每個 vector 中的 double 來得快。
6. 嘗試各種 chunk size 和 Master 要送出的 row 數，最後發現 chunk size = 1, row = 1 or 2，或得到較佳的結果。
7. 有嘗試用 static 的方法去切圖檔，而不是用 Master-Slave 的模式去分配工作，但這樣的效果不佳。

2. Experiment & Analysis

i.Methodology

(a). SystemSpec

都是使用課程提供的 apollo 進行測試。

(b). Performance Metrics

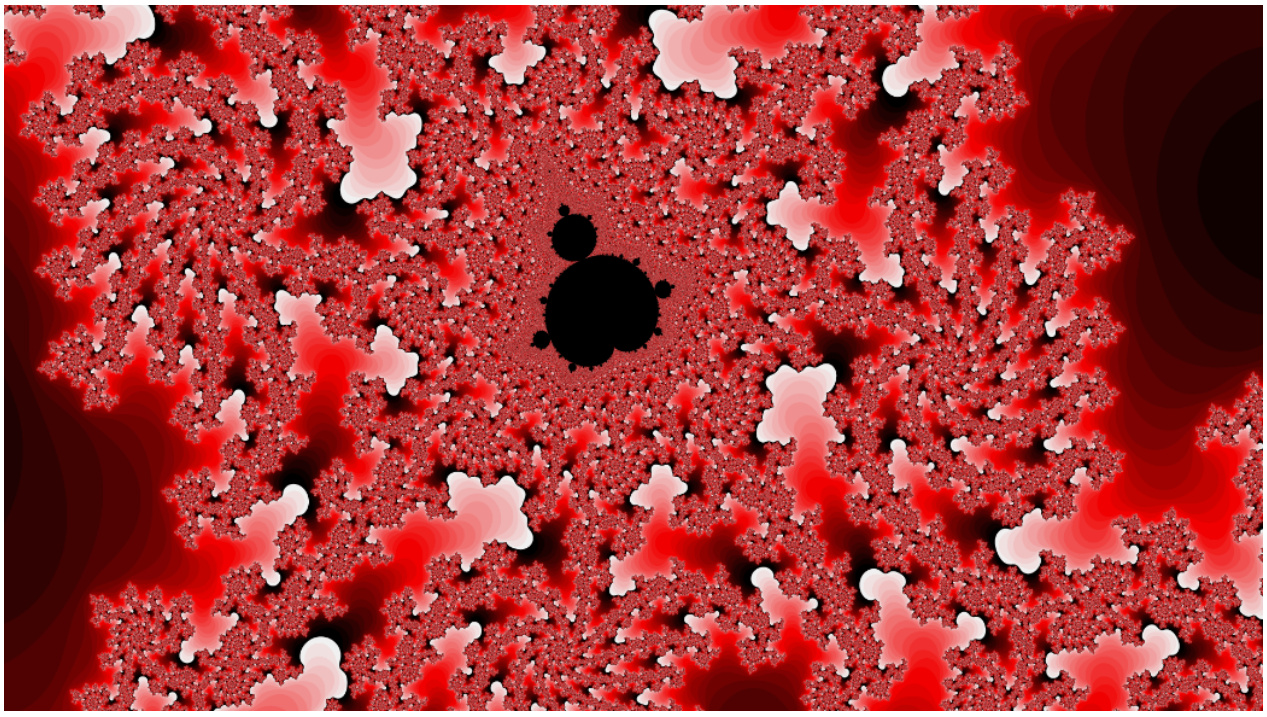
對於 Pthread version，使用的是<time.h>中的 clock_gettime()和 struct timespec，timespec 中有 tv_sec 和 tv_nsec，提供秒數和 nanosecond 的資訊。我們在 pthread_create()後 call 的函式的一開頭就使用 clock_gettime()，程式的結尾也用，可以計算出完成一個 thread 的總時間。

Hybrid version 使用的是 MPI_Wtime()來進行計算。Slave Process 會把 MPI_Send、MPI_Recv 等函式包住，Computation Time 就是把總時間扣掉溝通時間；而對於 Master Process，會對 Communication 的 Thread 計算他的計算時間，對 Communication Thread 計算他的溝通，平行化的階段結束後，會再計算 I/O 的時間。而 I/O 時間並不會被涵蓋在本次實驗的 Speedup 中，因為我們要觀察的是平行化、vectorization 的效果。

ii. Plots: Scalability & Load Balancing & Profile

- Experimental Method :
    - Testcase Description:

      這次選擇的是 strict28，因為可以看出它的分布很不均勻，有些地方很黑，有些地方是白色，因為我們 Master-Slave 的工作模式是送 row 出去，希望可以透過實驗發現會不會有 Load Balance 不均的問題。



    - Parallel Configurations:

      Pthread:

因為是 Sinlge-node 的方式，所以只會有同個 node 和 single Process 來做平行實驗。

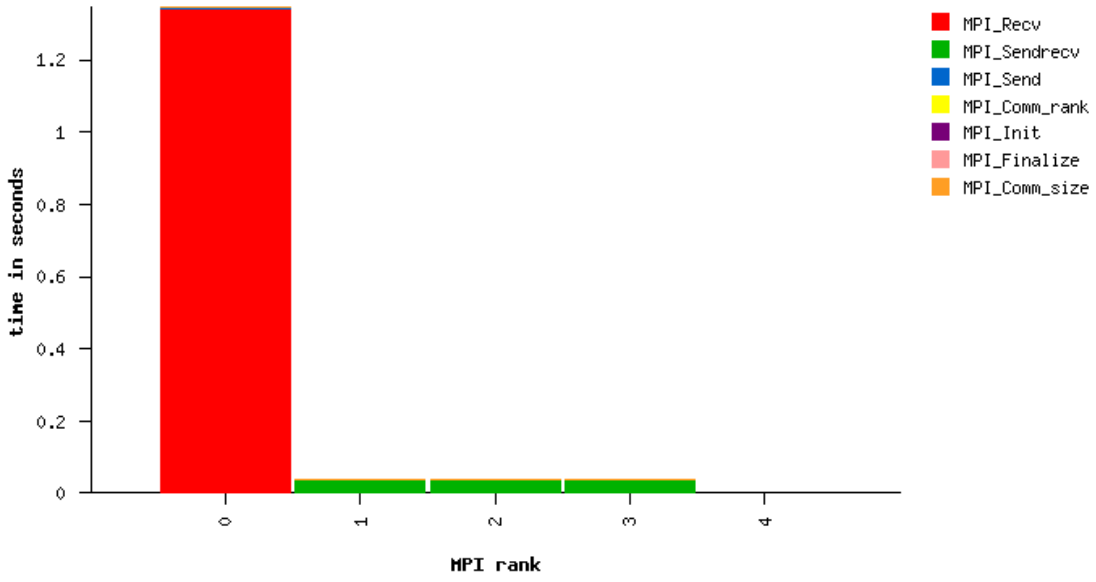
Hybrid:

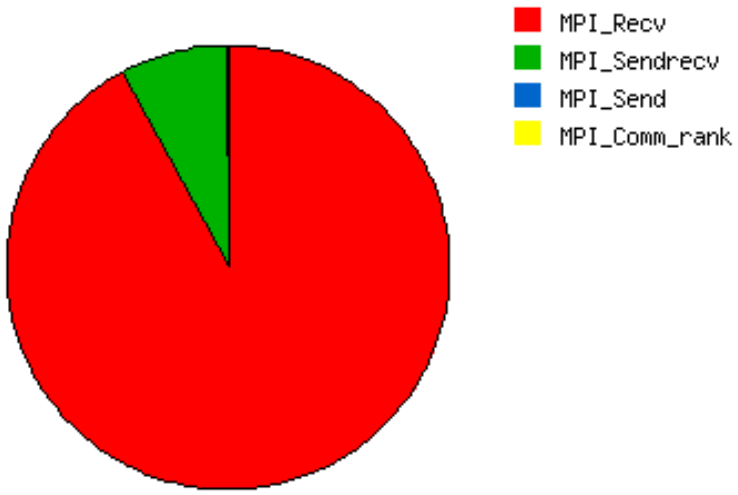
由於是 Master-Slave 的模式，所以必須要有 2 個 Process 或是至少 2 個 thread 才能計算，因此實驗會以 2 個 Process 為主，根據給予不同的 thread 來測量他的 Speedup。另外，也會加上多個 Process、Node 的實驗來看是否對溝通有影響。

- Performance Measurement:
    - Use a profiler (like IPM) for performance analysis.

IPM Profiler 提供不同 Process 所花的 MPI 時間。由圖可知 strict28 這個 testcase 有 4 個 Process 可以使用。而在 Master-Slave 模式中，rank 0 是 Master，因此花了最多時間在做 communication，且主要是以 MPI_Recv 去收集其他 Process 的成果。而其他 Process 則是將自己運算出的結果傳送給 Master 後等待 Master 給予工作，故花很多時間在 MPI_Sendrecv。從分布上也能看出每個 Slave Process 的工作量是很平均的。

可以看出最主要的時間會是在 MPI_Recv，因為 Master 接收一個 array，每個 Slave Process 要傳送一個 array 和接收一個變數。





- Provide basic metrics like execution time

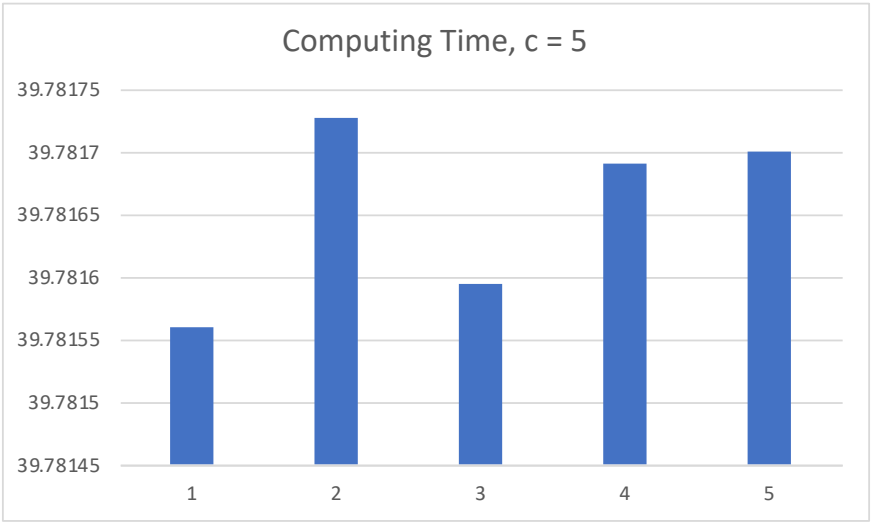
  在 Pthread version 中，會計算的是每個 thread 所花的總時間。

  在 hybrid version 中，會進行每個 Process Computation 時間的比較。而 I/O 的時間基本上是固定的，所以就不在圖表上呈現。

- Load Balancing

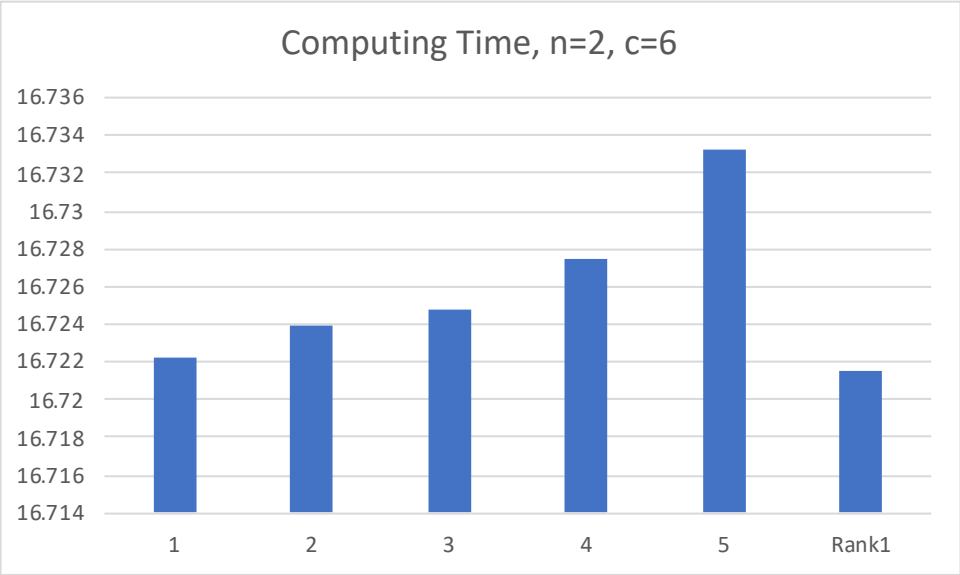
  Pthread:

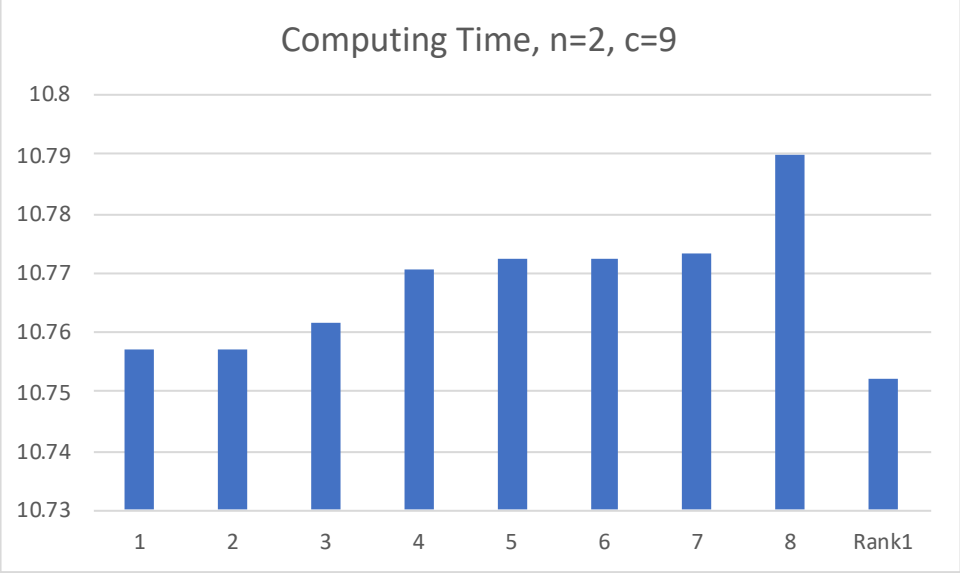
  如果出現工作量不均的情況，每個 Thread 的總時間會有明顯落差，讓整個工作最後卡在某幾個 thread 上面。從下圖可以看出每個 thread 的工作時間差距非常小。



Hybrid:

X 軸數字表示的是 Master 的 thread number，Rank 是其他 Process 的 Rank。不論是 Master Process 的 thread 還是其他 Slave Process，工作時間非常接近，差距介於 0.04 秒之間，故可以得出 Load Balancing 正常的結論。

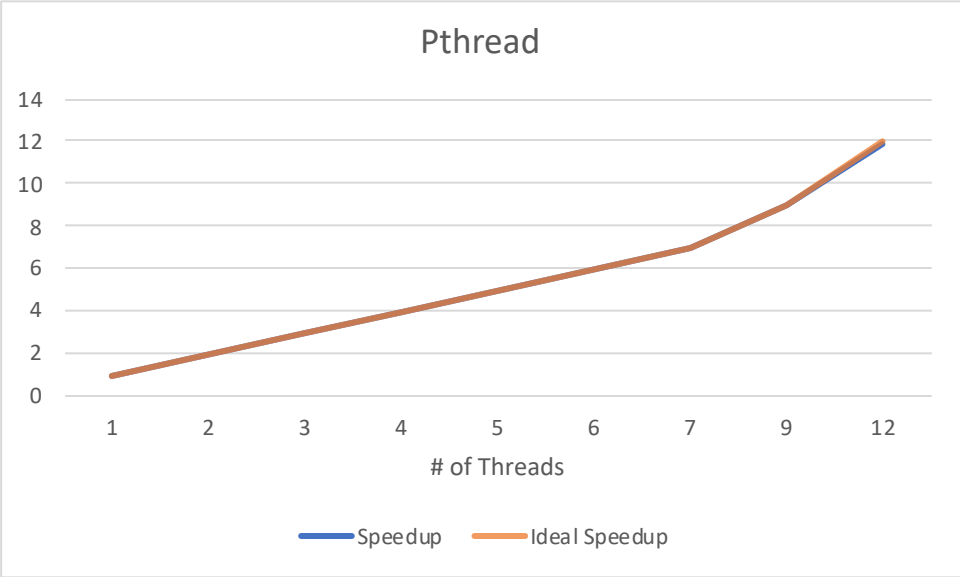




- Strong Scalability

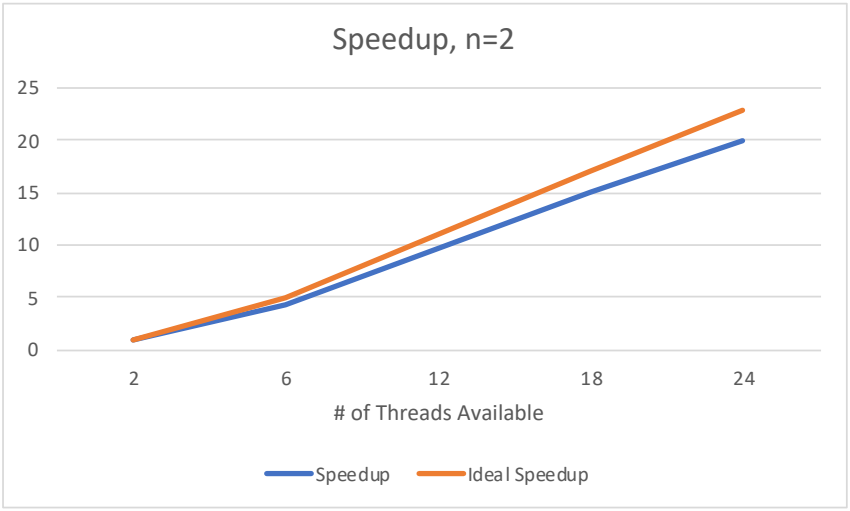
Pthread:

從折線圖和表格可以看出幾乎是完美的 Speedup，兩條線幾乎疊在一起，Process 數相當於 Speedup。因此可以發現就算有 mutex lock，還是不太影響整體平行化的表現。

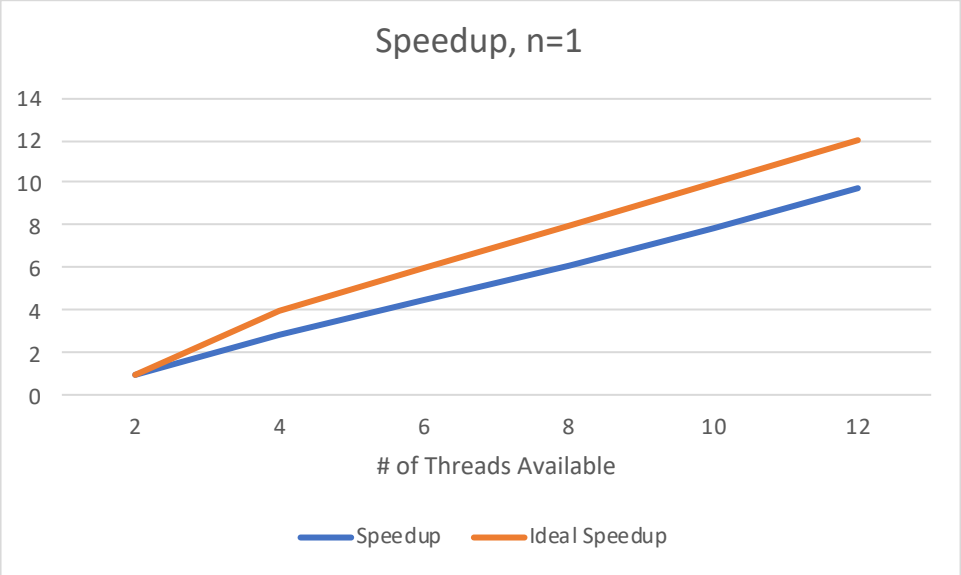


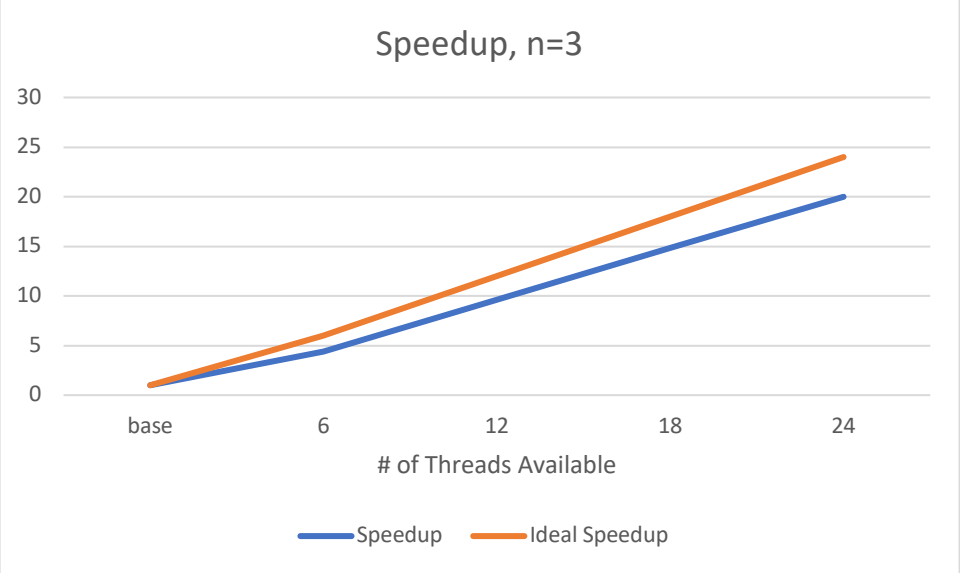
Thread =	1	2	3	4	5	6	7	9	12
Speedup	1.00	2.00	3.00	4.00	4.99	5.99	6.98	8.94	11.93
Ideal Speedup	1	2	3	4	5	6	7	9	12

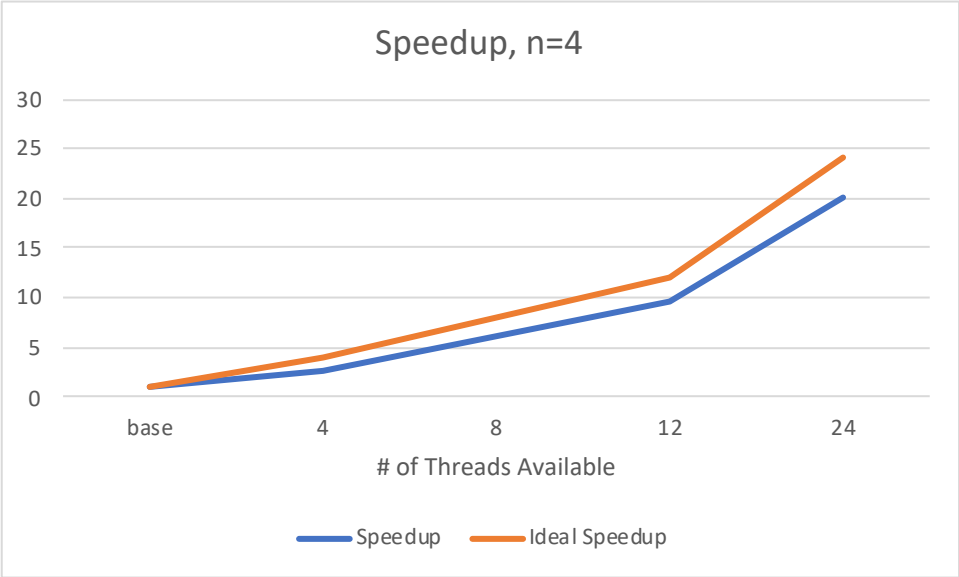
Hybrid:
當我們固定 Process 數為 2，改變每個 Process 能用使用的 thread 數時，可以發現 Speedup 並沒有像 Pthread 那樣完美上升。
要注意的是 Speedup = 1 時，總 Thread 數是 2，因為在 Master-Slave 的情況下要有一個 Slave Process 擁有 1 個 CPU 來進行運算。所以 Thread 數減去 1，才會是有在進行運算的實際 Thread 數。
根據 Speedup 公式：Execution time (舊) / Execution time (新)，因為我們要觀察的是使用 MPI、OpenMP 和 SSE 的 Speedup，所以 Execution Time 只包含 Computing time 和 Communication time，I/O 並不包含在其中。



- Strong Scalability with Process number = 1, 3 and 4
  在不同的 Process 數，固定總 Thread 數的情況下，Speedup 的情形和 Process 數等於 2 時相同，沒有太多變化。
  由於當 Process 數大於 2 時，無法設定小於 3 個 Process 數的執行方式，所以 base 是用 Process 數等於 2（base）的時間去計算總時間，以作為 Speedup 公式的分子。

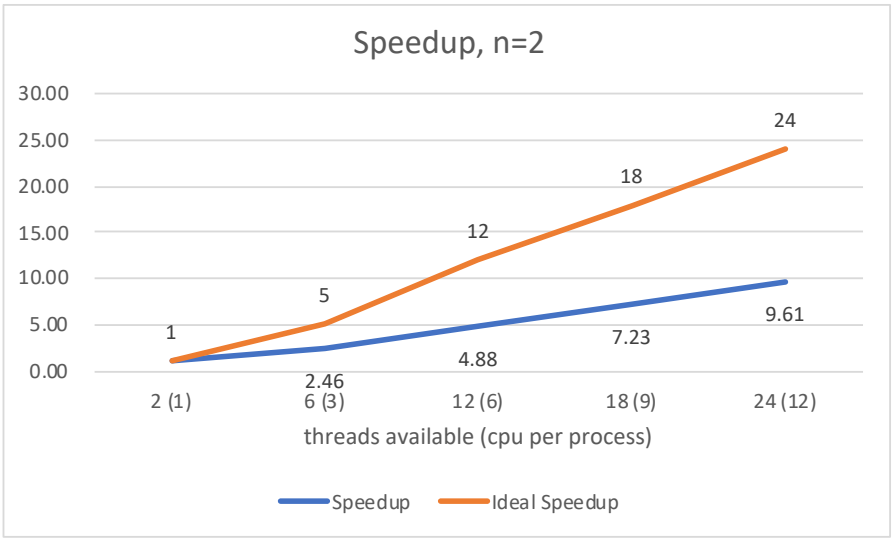


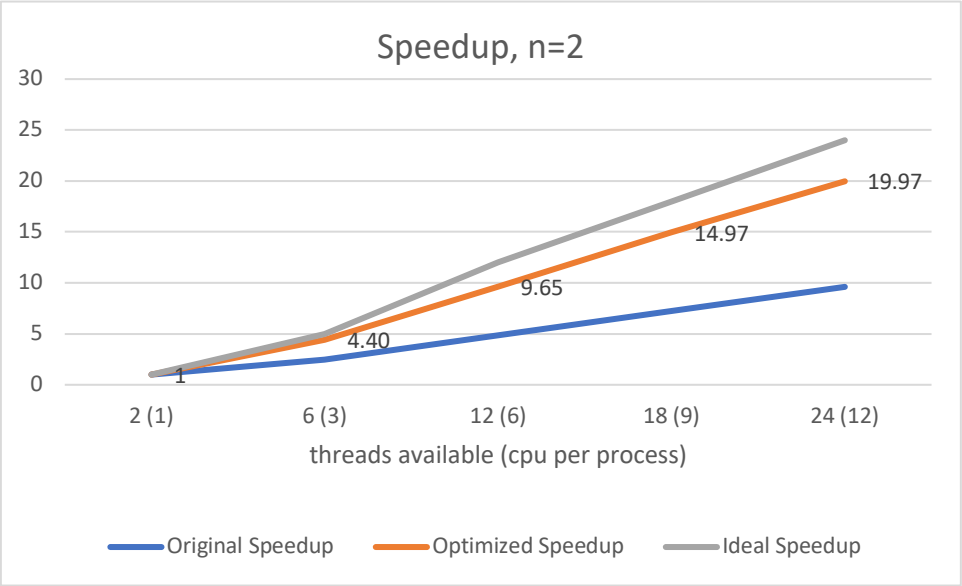




- Optimized Strategies:

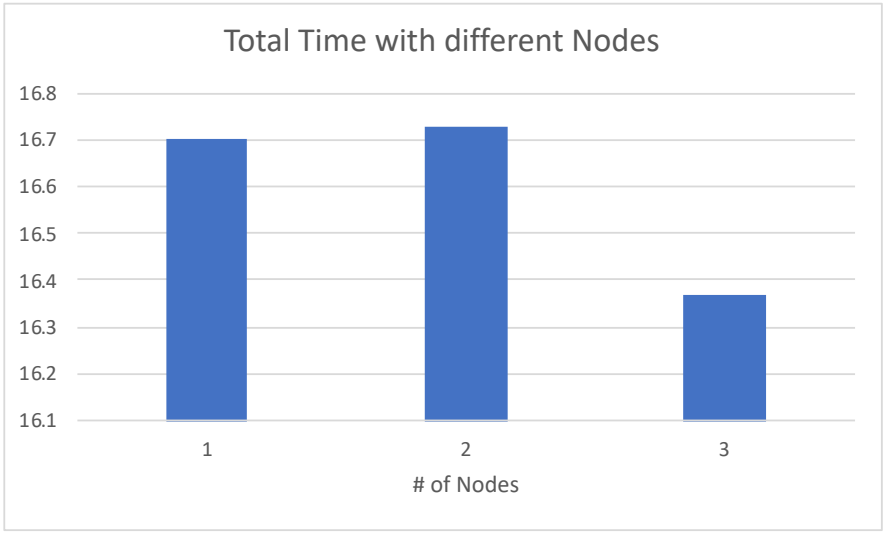
  從時間分配上，可以發現程式在傳送資料、接收資料所花的時間並不多，因此可以知道作業 2 是個 CPU-Bound 的任務，因此如果想要增加 Speedup，最重要的方式是增加平行度。而我有提出 3 個方法：1. 使用 OpenMP（最基本的）。2. 使用 Vectorization，讓每一個 thread 的工作量變成 1x ~ 2x。 3. 讓 Master 在做溝通的同時，也有其他 thread 在運算。
  下圖（上）為 Strategies 3 的成果，可以看到在只有 2 個 Process 時，如果不使用 optimized 方法，當 n=2, c=3 時，我們只有 Slave Process 的 3 個 cpu 可以用，因此 Speedup 頂多只有 3，再扣除一些成本使得 Speedup 從理想上的 5 降到了 2.46。從下圖（下）可以看出經過優化，橘色的線明顯比藍色線更貼近理想上的 Speedup，這是因為我們有充分運用到硬體給予的資源。





- More Experiment – Node

  接著我們針對不同的 Node 數進行實驗，觀察如果固定 Process (n)和 CPU (c)的數字，時間會不會有變化。
  我選擇了參數 -n4 -c3，代表 4 個 Process，每個 Process 有 3 個 CPU，一共 12 個 thread 可以運用，選擇此參數是因為要有至少且最多 3 個 Process 才能讓我們的實驗進行到 1~3 個 Node，CPU 選 3 是因為不希望有太多的 CPU 進行運算而使得總時間太短，讓我們看不出 Communication Time 有沒有造成影響。
  由實驗結果下圖看出 Node 數量不會是我們關注的重點，總花費時間不會因為 Node 數增加而有明顯提升。就算程式放在不同的 Node 上面跑，網路不會是 Bottleneck，不會對時間有影響。因此更可以確認我們的作業是 CPU-bound。



- 未完成的 Optimized Strategies

  根據實驗我們可以看出 CPU 的使用率是加速的最佳路徑，溝通不是問題，使用 Vectorization，能讓每一個 thread 的工作量最多變成 2x。目前的狀況是當 vector 中某一個 element 完成時，他必須等到另一個 element 算完，這樣就會浪費掉我們要的 Speedup。如果可以充分使用到 vectorization 的話，我認為效能可以再提升。

  還有一個是寫成 png 的過程應該可以被平行化。這次作業我沒有很仔細研究 write_png 這個 function 是如何把原本的一維陣列寫成 png 檔，但看起來也是一個 row 一個 row 的方式去進行，I/O 會固定吃掉一些時間，雖然沒有很多，但也是可以優化的地方。

3. Discussion

(a). Compare and discuss the scalability of your implementations.

除了溝通的成本之外，因為每次 Slave Process 接收到工作後都還要再把工作用 Schedule 的方式分配給它擁有的 Threads，這個時間沒有被計算到，但是會明顯地影響了 Speedup 無法跟上理想 Speeup。且由於我們用的是 Dynamic 的模式，這個 overhead 會比 static 多。可是這是必要的，因為每個 pixel 的計算量非常不同，如果用 static 的話會導致某些 thread 很快做完手上的工作發呆，其他 thread 持續工作，使得 Slave Process 無法向 Master 要求更多計算量。反觀 Pthread 因為已經把 Thread 生出來了，再讓 Thread 去搶工作，這以實驗結果而言，是成本較低的方法。

這次的 Scalability 我認為是不錯的，從圖表來看雖然不是完美，但也很明顯地跟著理想線的趨勢，而 I/O 的時間其實不會佔到太高比例，所以就算加上去 Scalability 應該還是不錯。

(b). Compare and discuss the Load Balance of your implementations.

在本次作業我使用的是 Master-Slave 的方法，OpenMP 的 Schedule 是使用 Dynamic，chunk size = 2 的方法。

我認為 Master-Slave 是較佳的方法，因為我也有實作 Static 的切割方法，把圖片平均分配給 Process 去運算，最後再用 MPI_Gather 把資料收集起來。其中切割的方法是用跳著切的做法，因為從圖片可以看出有些黑色區域非常集中，如果剛好被某個 Process 收到，那其他 Process 肯定要去等它做完。所以假設今天有 5 row，3 個 Process，Process 1 要算的是 row 1 和 row 4；Process 2 要算 row 2 和 row 5。但實作下來的效能還是比 Dynamic 差滿多的。

而 Master-Slave 的溝通成本並不會太高，根據 IPM 的圖也可以看到每個 Process 的 MPI_Sendrecv 都花差不多的時間，因此 Load Balance 還算正常。

OpenMP 的 Schedule 如(a)章節所述，在嘗試過 chunk size、guided、static 等各種選項後，Dynamic 的效果是最好的，這不難理解，因為只要 thread 完成自己的工作就可以再拿到，這樣是有效率的。

4. Experience & Conclusion

Master-Slave 是實驗出來較好的 Load Balance 方法，而 OpenMP 和 Vectorization 可以進一步加速運算。如果能用到 AVX 的 256 位元運算的話，相信速度可以再提升。

在這個作業嘗試了非常多種的做法，從 static partition、Master-Slave、Scheduling，還有 vectorization。其中 vectorization 花了不少時間在做優化，一開始我很想把所有 operation、變數都弄成__mm128d，但不知道為什麼，用一個迴圈去比較 vector 中的 2 個 element 的效能就是比較好。此外，做這些測試時，一定要確保其他 operation 都沒有冗餘，尤其是減少 load、store，會佔超多時間。

還有我原本希望 vectorization 可以有辦法當其中一個 element 完成時，就去拿新的 pixel 近來運算，到作業截止前我都還沒有寫出一個 AC 的版本，真的很困難。而且這樣要設置 critical section，也要增加許多 if-else 的 statement，好像時間會大幅增加，所以可能還是要回歸到簡化 while loop 中的複雜度，增加 branch 應該會讓平行化的效果減弱。

這次作業讓我學到更多平行程式的技巧，也更能了解以前作業系統學到的 pthread 還有 Load Balance 的概念，雖然效能排名不是最前面，但還是覺得收穫滿滿。