Parallel Programming

Hw2 Report

110062208 林書辰

1. Implementation

i. pthread version

在pthread version中，我先用sched_getaffinity得到core的數量，我開的thread數量就等於core的數量。

接下來在分配資源給threads的地方我以高度1 pixel的row為基本單位進行分配，每個row輪流分配給每一個threads，也就是thread 0會拿到第0, threads_num, 2*threads_num...以此類推的row。理所當然的如果row的數量沒有辦法整除，假設會剩餘remain_size個row，他們就會被分配到前面remain_size個threads。在pthread傳入的參數中，只有image是用指標傳送，其他都當作local variable在傳。

另外，我還有使用SSE vectorization。這個部分我是對每個row做vectorization，row上面每兩個pixels進行vectorization，因為在判斷while終止條件的時候需要兩個pixels都終止，所以我用done[2]紀錄是否兩個pixel都已達到終止條件。如果每個row有奇數個pixel，需要單獨計算最後一個pixel，這裡我是單獨計算row上面的第一個pixel。

ii. hybrid version

在MPI的部分，我跟在pthread版本一樣以高度1 pixel的row為基本單位分配給每個rank，分配的方式也都一樣，然後我在本地開了一個new_image來存每個rank算完的數值。最後用MPI_Gather把所有算完的new_image存到在rank 0 的image中。因為row不是按照順序分配給processes的，所以在write_png的地方也有做一些改動，就是讓他在計算的時候維持原本pixel的順序。

在openMP的不分我是對inner loop做parallel，也就是對每一個row內部的計算做平行，我是用schedule(dynamic, CHUNCKSIZE)，CHUNCKSIZE是5。嘗試過更高的CHUNCKSIZE但都沒有更好的表現。

SSE vectorization的部分跟pthread一樣我都是對row上面的pixels跑vectorization，每兩個pixels一起，和pthread version的沒有太大的區別，也是利用done[2]來記錄兩個pixel分別達到終止條件了沒，最後如果一個row有奇數個pixels的話就先把第一個拿出來單獨算，其他的每兩個一起算。

2. Experiment & Analysis

i Methodology

(a) **System spec**

(i) 利用課堂上的cluster進行測試

(b) **Performance metrics**

Pthread version: 計時的函式我是用clock_gettime來記錄start_time, end_time, 我在thread function的頭跟尾使用並記錄start_time, end_time, 最後再把他們相減就可以得到每個threads的時間。程式的execution time則是記錄整個程式跑的時間。

Hybrid version: 我利用MPI_Wtime()來記錄start_time, end_time, 最後也是相減得到需要的時間, 在這裡我也有嘗試IPM, mpiP來看MPI_Gather所執行的時間。不過主要是利用MPI_Wtime來計算CPU時間。在紀錄時間的時候有把write_png分開計算, 因為只有rank 0會跑到, 所以CPU時間我只有紀錄到MPI_Gather之前。

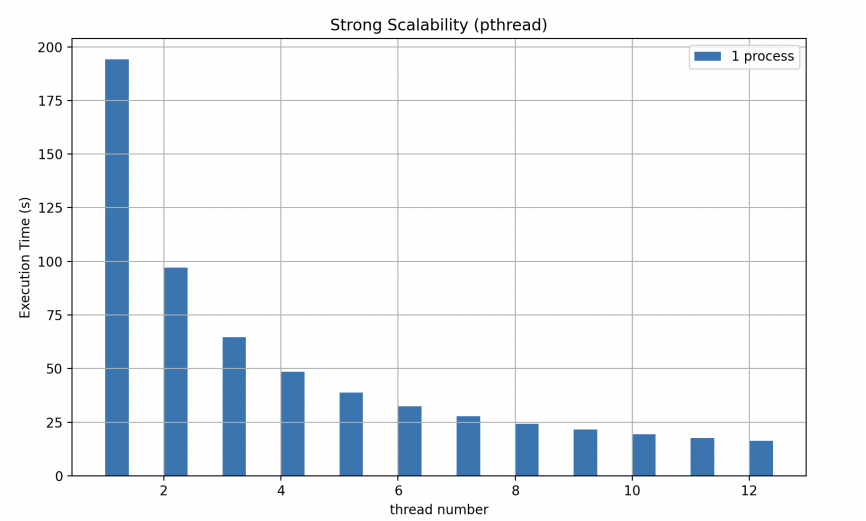
ii Plots: Scalability & load balancing & Profile

- **Experimental Method**
  - **Test Data:** 我是利用strict35來當作測試資料, 因為他執行的時間不會太短, 我相信這樣比較容易看出區別。strict35.txt是
    iteration: 10000 **x0 :** -0.2931209325179713
    x1: -0.2741427339562606 **y0:** -0.6337125743279389
    y1: -0.6429654881215695 **width:** 7680 **height:** 4320
  - **Parallel Configurations:** 在pthread version中, 我用一個process測試1到12個core的平行度, thread開的數量與core一樣。在hybrid version中, 我測試了1 到 12個processes, 每個processes有3個cores的情況。

- **Performance Measurement**
  - Pthread version: 沒有使用profiler, 相對的我自己利用clock_gettime來計算花費的時間, 並用python畫圖。
  - Hybrid version: 我有利用IPM來看每個processes執行MPI_Gather所花的時間, 主要的CPU時間則是利用MPI_Wtime(), 因為rank 0寫入資料沒有平行, 所以自己用MPI_Wtime()測時間可以避開這個部分。

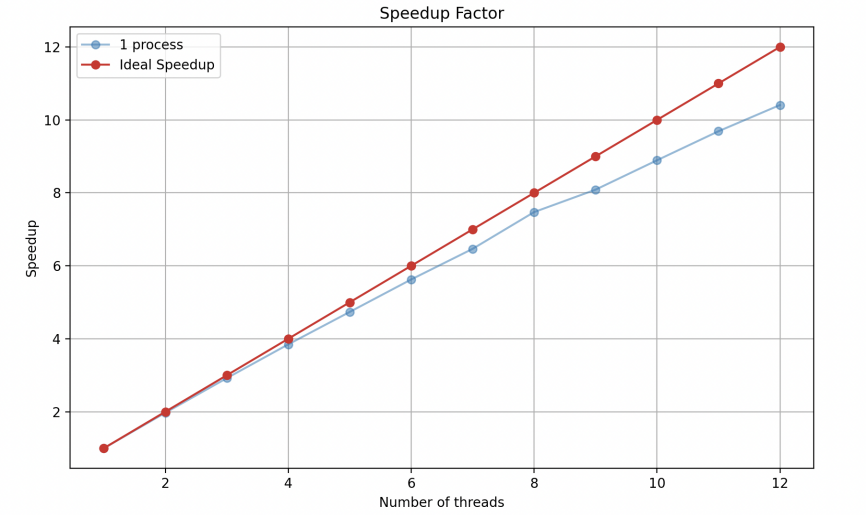
- **Analysis of Results**
  - **pthread version:**

Strong Scalability



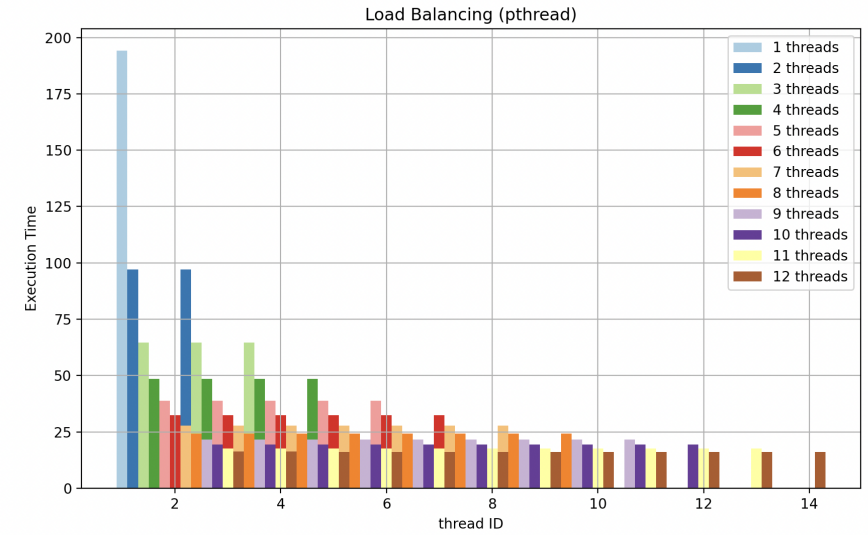
我測試了在一個process上開1到12個CPU，可以看到執行時間是有隨CPU數量增加下降的，我認為這樣的Strong Scalability是不錯的，因為到最後雖然有趨緩但還是看得出CPU數量對平行化的影響。

Speedup Factor



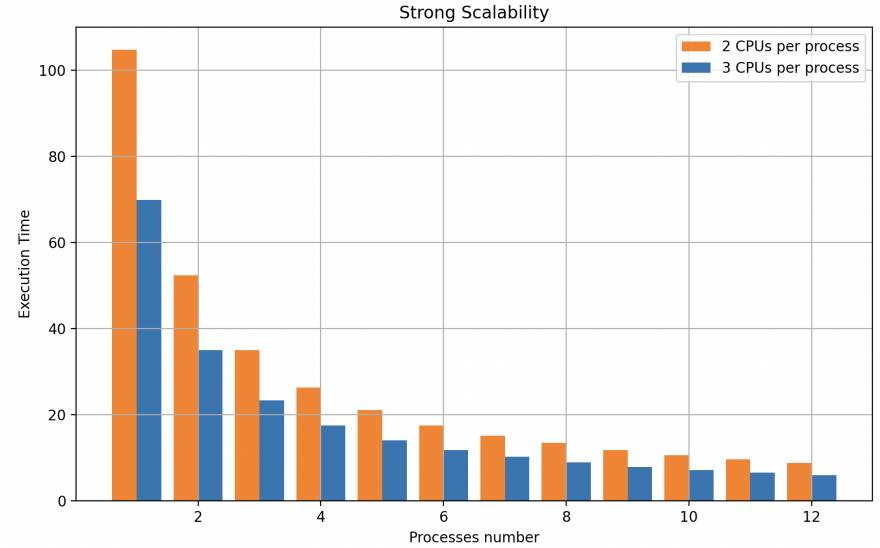
紅色的線是ideal speedup，可以看到我的程式的speedup並沒有偏離太多，但用越多的threads可能就會體現write_png沒辦法被平行的缺點，所以speedup沒辦法跟上ideal speedup，但整體來說是很好的。

Load Balancing



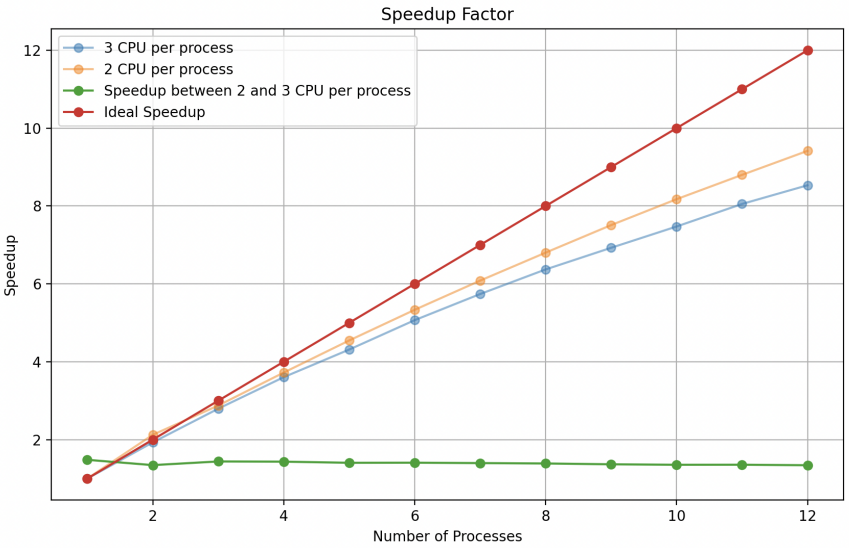
這裡我也是用一個process然後測試不同數量的threads下，每個thread花了多久的時間在跑thread function，沒有測threads function以外的，主要是想看資料的分配有沒有平均。圖上每個顏色對應到不同的threads數量的執行結果，可以看到不管是用幾個threads，他們的時間分配都很平均。不管用幾個threads都有達到好的load balancing。後面的長條圖跑到14是因為前面的佔太多所以跑到那裡去，其實並沒有到14個。

- **hybrid version:**
  Strong Scalability



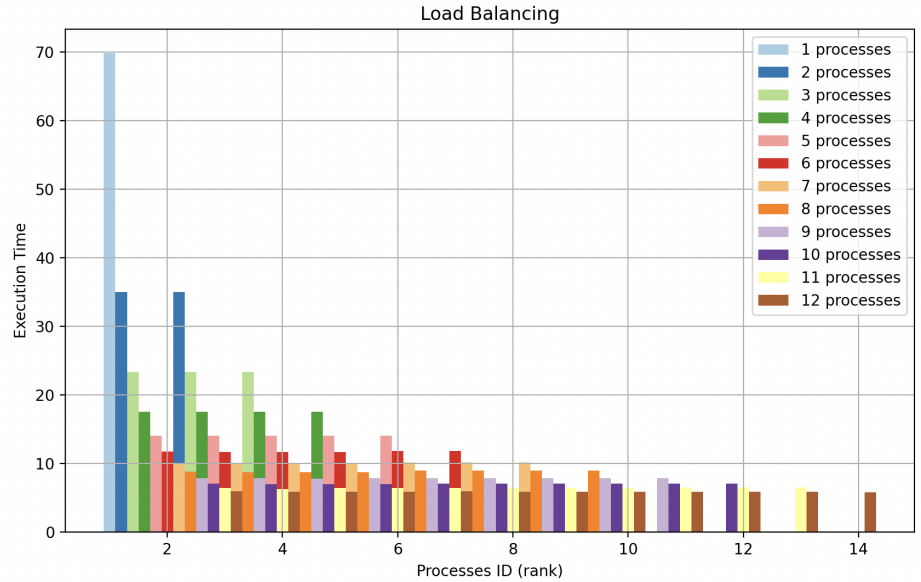
我分別測試了在每個process上跑2或3個CPUs，調整process數量的strong scalability。可以看到使用越多的process或者用越多的CPU per process都有效的加速的程式的運算。

Speedup Factor



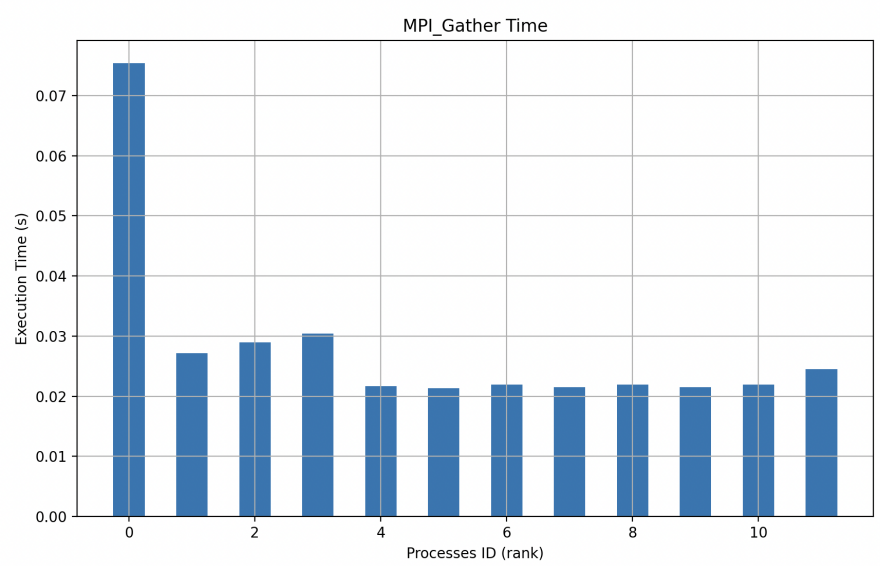
藍色還有橘色的線分別是在每個process 2CPU還有3CPU的情況下，process增加的speedup，可以看到因為write_png會佔掉時間，而且在越多processes，也就是總執行時間越少的情況會讓speedup掉得越明顯。然後每個process增加一個CPU大約可以加速1.5倍(綠色的線)，在不同的process數量下看起來都可以有這樣的表現。

Load Balancing



在這裡我把不同process數量的load balancing用條狀圖畫了出來，每個顏色代表的是不同process數量執行的結果。可以看到不管

用幾個processes，balance的狀況看起來都很好，這裡我只有考慮CPU tim，因為rank 0寫入會花較多時間，我覺得那段code沒有在平行的code裡面所以就沒有算入。後面的長條圖跑到14是因為前面的佔太多所以跑到那裡去，其實並沒有到14個。

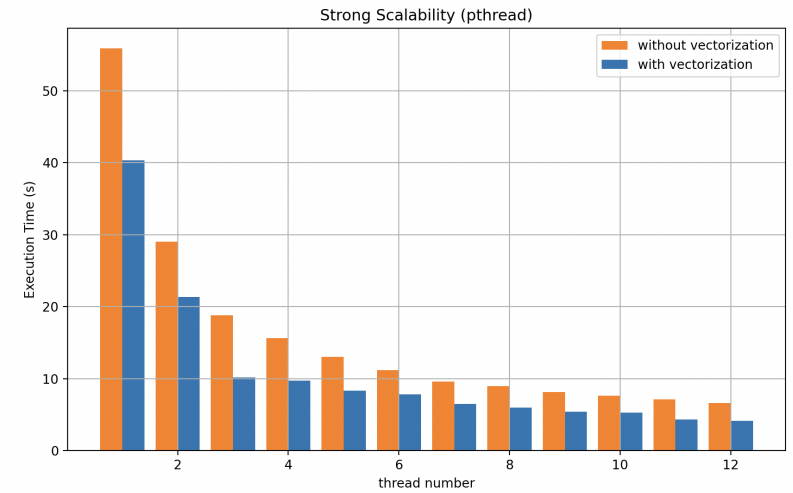


我也有利用mpiP profile紀錄了每個rank在執行MPI_Gather花了多少時間，可以看到雖然rank 0 雖然花了較多的時間，但是他跟花最少時間的rank也只差了0.05秒左右，因為用12個processes跑這筆測資也只需要約6秒的時間(除了write_png)，所以可以看到MPI_Gather相對花了很少的時間，所以可以說明他的load balancing表現是很好的。

- **Optimization Strategies**
  - **vectorization**

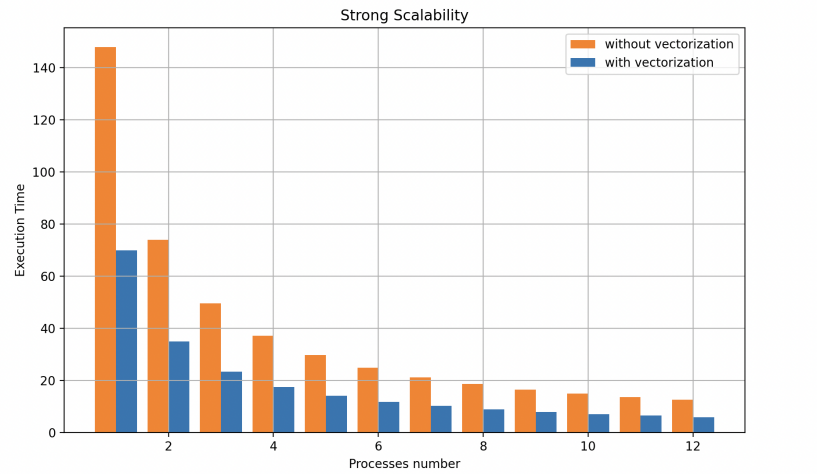
    利用SSE vectorization加速部分loop的運算，雖然上面討論的都是已經使用vectorization的code，但我想在這裡討論一下有沒有用vectorization的差別，比較他們的執行時間。

    Pthread version



我測了有沒有vectorization對於不同數量的thread的執行時間。因為原本的用的資料 strict35在一個process一個CPU並且沒有vectorization會超過cluster的時間限制，所以我找了strict10當作另一個測試的資料，**iteration:** 10000 **x0:** -0.3421054598064634 **x1:** -0.2373971478909443 **y0:** -0.6373595233099365 **y1:** -0.6884105743609876 **width:** 7680 **height:** 4320。可以明顯看到vectorization加速了很多。並且在不同threads的情況下都能有顯著的加速。

Hybrid version



hybrid version我就維持了原本的資料進行測試，每個process維持3個CPU，可以看到沒有使用vectorization的話，所花的時間和加速過後的差距是很明顯的。

iii. Discussion

- Strong Scalability

  可以看到不管是pthread version還是hybrid version，Strong Scalability都是不錯的。因為我是以row為單位去分配計算範圍的，所以可以看到這樣的分配方式對於mandelbrot set是還不錯的，更好的方式或許是動態的分配，但我沒有想到具體要怎麼在計算的時候分配資源所以沒有實現。關於vectorization的部分，因為我是把row上面的pixel兩個兩個算，因此兩個pixel的while執行次數差是加速的關鍵，我有想過維持每次都兩個pixel計算，當一個結束馬上接下一個進來，但實作的結果有點不理想，還沒找到原因。因為我感覺這樣的話overlap的時間會更多，但也有可能頻繁的更動sse vector會增加計算資源。

- Load Balancing

  雖然這筆資料測試的結果顯示我的程式有還不錯的load balancing，但有幾筆測資我實際測的時候會有一兩個rank花比較少的時間跑完，但我覺得要判edge case的話會太多，所以並沒有多做處理。我認為更好的解決方法就是像我上面所說的那樣用動態分配資源的方式，這樣能解決一些edge case的問題，但有沒有辦法在增加動態分配的計算的情況下維持原本的加速就不得而知了。

4. Experience & Conclusion

我認為完成基本的程式還算輕鬆，在寫lab2的時候就有練習過pthread, openMP, 上次的作業也花滿多時間在MPI上面，所以還算順利。因為一開始像上次一樣分資料是直接thread 0或rank 0直接拿前面task_size筆資料，完全沒有考慮到load balancing的問題，後面幾經修改才讓load balance比較好。後面的vectorization是我第一次接觸，摸索了滿多時間，也在intel intrisics guided裡面逛了很久才找到自己要的函式。思考要怎麼用vectorization也算是滿有趣的，平行出來的結果也不錯。但我發現scoreboard上有人跑得很快，想了很久也試了很多不同的方法，像是上網查有甚麼奇怪的flag可以加速或者改變資料分配的方法，但我感覺都大同小異，找不到其他人跑這麼快的原因，希望之後能有分享環節讓大家知道他們有甚麼小技巧讓他們的程式可以再更進一步的加速。