hw2 report

109062320 朱季葳

Implementation

Optimize Sequential Code

首先在原本的sequential code裡面可以發現一些不必要的計算或者空間儲存，而這個部分我做了以下幾點優化

1. 減少重複計算:在這裡我把for loop裡面會重複用到的部分都額外拉出來做計算，像是x和y的平方就額外加上 `x_seq`，`y_seq` 來避免每次iteration都重複計算好幾次。
2. 減少不必要的iteration，像是在 `write_png` 裡面就跑了一個雙層迴圈然後把先前在 `mandelbrot set` 那邊處理好的image裡的值取出來然後再處理過並寫入row，但是在這裡可以發現我們其實並不一定需要image，並且在這裡的row是不斷被重複使用的，於是我在一開始就開一個新的陣列並取名為 `rows` ，讓最後在寫的時候每條row是獨立的，然後在 `mandelbrot set` 那邊就直接把 `repeats` 的值寫到每條row裡面相對應的位置，而這裡有一點值得注意的就是寫到row的時候index要反過來，如此一來在write的時候就可以把雙層迴圈改為單層。

```
int main(){
    //...
    rows = (png_bytep*)malloc(height * sizeof(png_bytep));
    row_size = 3 * width * sizeof(png_byte);
    //...
}
void mandelbrot(){
    for (int j = 0; j < height; ++j) {
        double y0 = j * YCoeff + lower;
        rows[(height - 1 - j)] = (png_bytep)malloc(row_size);
        memset(rows[(height - 1 - j)], 0, row_size);
        for (int i = 0; i < width; ++i) {
            // ...
            write_color(repeats, rows[(height - 1 - j)] + i * 3);
        }
    }
}
void write_png(const char* filename, int iters, int width, int height){
    //...
    for (int y = 0; y < height; ++y) {
        png_write_row(png_ptr, rows[y]);
        free(rows[y]);
    }
}
```

Vectorize

在這個部分我使用 `SSE intrinsic` 去進行Vectorize的實作，而我在這裡把每兩兩一組的column綁在一起做計算，使得在inner for loop的每次iteration都可以對兩個pixel做計算，舉例來說就是把i=0, i=1的計算合併，且因為一個 `__m128d` 的vector裡面可以放入兩個double，所以在後面所有的運算我都可以在initialize vector過後，把原本的sequential code改為使用intrinsic來做vectorize的計算，而因為每個pixel的工作量可能會不一樣，所以只有在兩個pixel的工作都還沒做完的時候才使用intrinsic才做計算，且因為這個vector是fixed size的，所以可以把它視為是一個array，並且在只有一個pixel需要繼續做的情況下用index進行操作。然後因為總column的數量(width)並不一定是偶數，所以有再特別針對這樣的狀況拉出來獨立處理。

Partition

PTHREAD

在這裡我使用 `Mutex lock` 去讓每個thread去搶 `global row` 並對該row的pixels去進行計算，然後把原本 `mandelbrot set` 那邊計算pixel的outer for loop改為while(true)，並在 `global row` 等於height的時候(代表所有pixel都算完了)break掉這個loop。

HYBRID

在這裡我原本是使用很naive的做法把所有的row均分給所有process，就和第一次作業的實作方式是一模一樣的，然後最後再用 `MPI_Gatherv` 把所有的local row做完的pixels merge到一個global row去，但是因為一張圖片裡面每個row的工作量可能會有很大的差距，所以這樣的做法並不能有很好的load balancing(針對這個部分我在**Hybrid_exp**這邊有再特別做實驗並針對這個部分做說明)，所以後來改為輪流分配的方式去分工作給每個process，例如process0負責 `row0, size, 2size... for size=# of process` ，並和最一開始的做法一樣merge到global row然後在rank0的時候寫出到png，在這裡要特別注意的一點就是因為每個process處理的部分是不連續的，所以我另外寫的一個mapping的array並於寫資料的時候根據mapping到的部分進行寫出。

Other Optimization

因為floating point的計算很吃precision，所以在這裡的話compiler有提供一些flag可以再對這個部分進行優化，而我使用的是clang的 `-ffast-math` 去進行加速，而因為如果直接使用這個flag的話，在 `hybrid` 的實作會有precision不夠高的問題，然後clang官網有寫到 `-ffast-math` 等同於其他數個flag合起來，所以我就把一個一個貼上去試，最後得到以下flags是可以確定不會嚴重hurt到precision且能夠有效加速程式的：

```
-fopenmp
-fno-math-errno
-ffinite-math-only
-freciprocal-math
-fno-signed-zeros
-fno-trapping-math
-fno-rounding-math
-ffp-contract=fast
```

Experiment & Analysis

Methodology

SystemSpec

課程所提供的Apollo cluster

Performance Metrics

pthread的部分我使用 `time.h` 裡面的 `clock_gettime` 在各處做時間標記(像是每個thread所需要的runtime就在進入 `mandelbrot` 這個function之後埋一個start，並在 `pthread_exit` 之前再埋一個end然後跟start)然後減去消耗的時間，而在pthread這邊因為只有對 `mandelbrot set` 實作的部分進行平行化，剩下的I/O time都一樣是write png，而每次需要寫的資料量都是一樣的，所以在這個部分就沒有特別去計算，因為不論多少thread，I/O時間都會是一樣的。
hybrid的部分我使用 `Nvidia Nsight System` 去測量I/O time, communication time還有elapsed time(從 `MPI_Init` 開始直到 `MPI_Finalize`)。

Plots: Scalability & Load Balancing & Profile

Experimental Method

Test Case Description:在testcase的部分我選擇strict26作為主要的實驗測資，因為他的runtime不至於到太短也不至於到太長，比較方便用來進行大量的實驗。

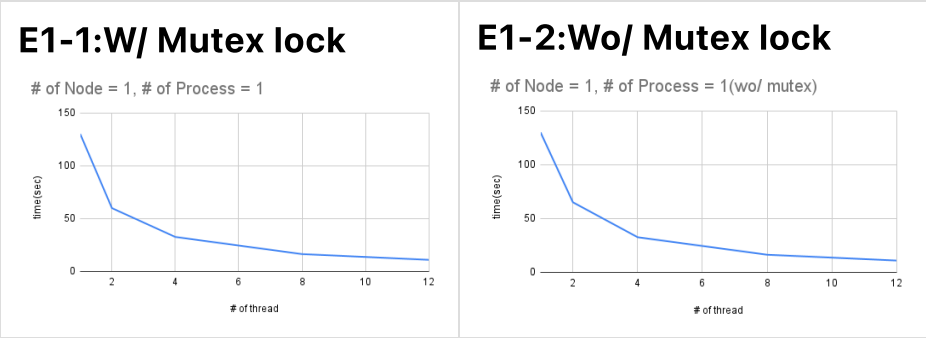
PERFORMANCE MEASUREMENT

ANALYSIS OF RESULTS

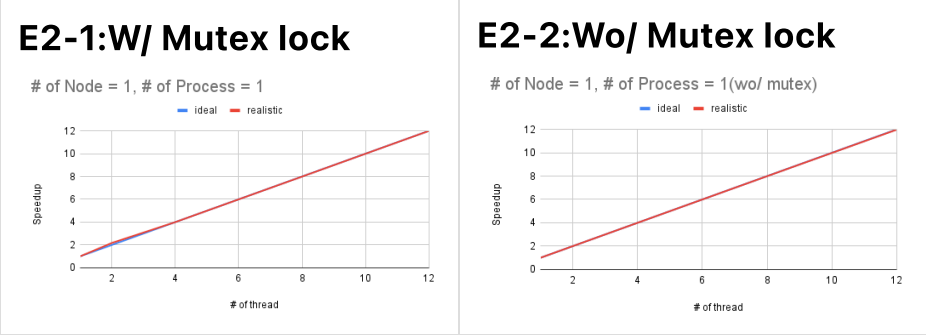
Pthread

在這裡因為實測出來single Node和Multi-Node的環境下，用相同的cpu數量測出來的實驗結果是差不多的，其中的差異大概只有Multi-Node會有Node之間的溝通時間而使得要花費的時間較多，所以在下面只放single Node的環境下測出來的實驗結果。
而以下將針對兩種針對load balancing的實作方法，分別是用Mutex lock以及用tid的數字來做均分所呈現出來的結果進行比較。

- Runtime

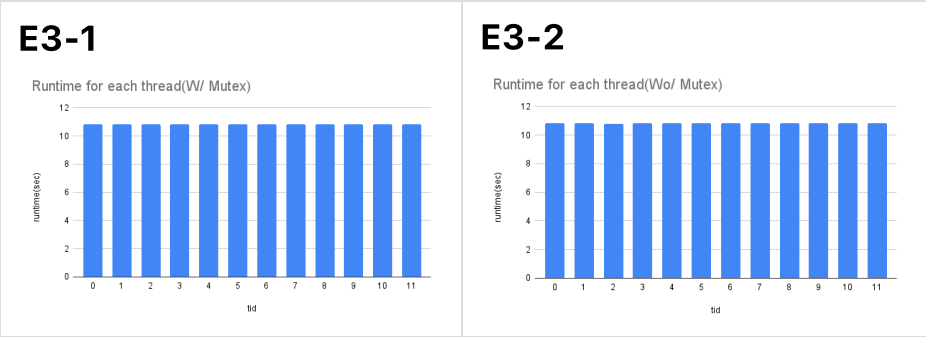


- Speedup



從speedup的部分可以看到無論是使用Mutex lock讓所有thread去搶 `global_row` 還是讓所有thread輪流拿取一段連續的row number(例如thread0負責row0, thread1負責row1...)的實作方式，實驗出來的scalability都很好，甚至已經與ideal的狀況完全貼合了，而我推測這是因為thread之間可以存取共同的share memory，而每個pixel要寫入的部分都是獨立的不會互相干擾，然後也不太需要communication time，所以就只剩下comutation的部分，然後平均的分散給每個thread之後就很容易的可以達到ideal的狀況。

- Load balancing(with #of thread = 12)



在load balancing的部分，首先是搶mutex lock的部分，因為只要有哪個thread拿到的row的work load比較少，他就可以在做完之後馬上又去搶mutex lock，然後再繼續工作，而拿到work load較重的row的thread就繼續把它做完，這樣一來就可以避免掉有某幾個thread拿到的工作量都很大，而有些thread卻沒事做的狀況，並有效的達到load balancing。而在平均分散的實作方法中，因為較為相近的row會拿到的工作量也比較相近，所以透過平均分散給每個thread可以很有效的讓每個thread都拿到差不多的工作量，因而達到load balancing。

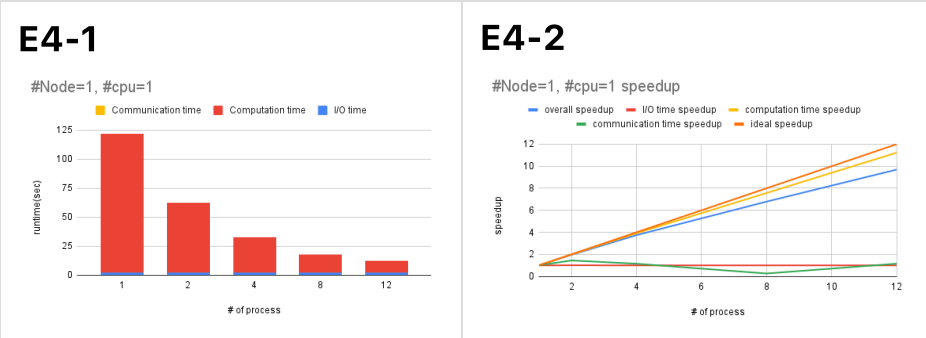
Hybrid

以下將進行兩種實驗

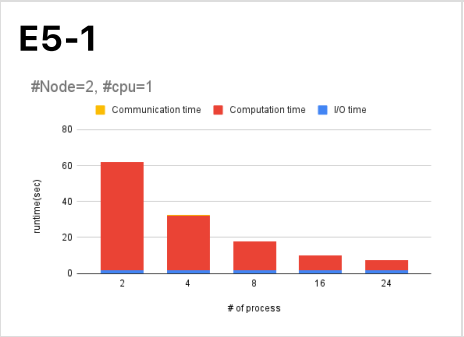
1. 針對最終版本在single node,Multi-node，不同process的環境下所產生的實驗結果
2. 針對兩種實作資料分配的方法進行在Single node, 不同process的環境下產生的實驗結果

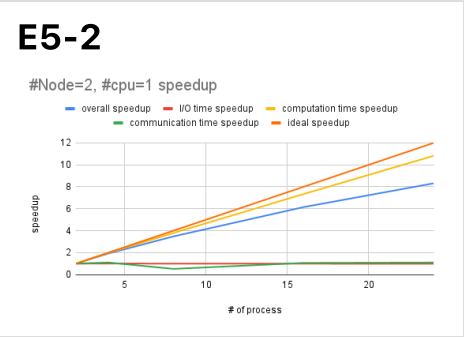
Runtime & Speedup

- Single node final version



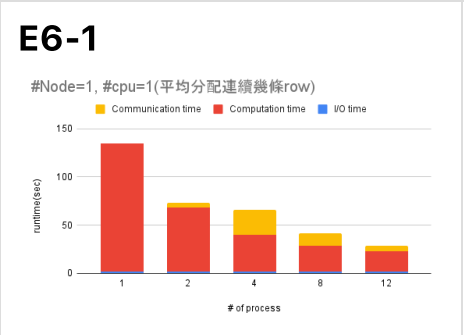
- Multi-node final version

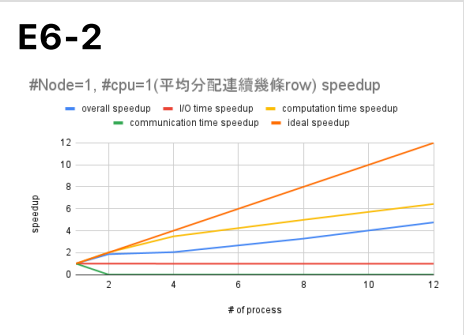




從以上圖表可以看到隨著process數量增多，runtime都能有顯著的減少，而computation time的部分是非常貼近ideal的，但是我猜測可能是因為中間還有call到一些MPI的API所以會有一些overhead，因為如果去看詳細的數據的話可以發現跟ideal的時間其實只差2秒以內而已，所以computation time才沒有辦法完全貼合ideal的狀況。而overall的部分我認為則是受到I/O time以及communication time影響，因為process之間要溝通還是會有一定的overhead，然後再加上在我的實作中是到rank0才把data一次寫出來，所以在I/O的部分就完全沒有做到平行化的部分，因而導致overall的scalability並沒有像pthread那樣好。此外，可以透過實驗數據看到single node和multi-node的環境下做出來的實驗結果其實非常接近，主要的差異大概就只有multi-node可能會有node之間的溝通時間而使得communication time那邊沒有辦法有很好的scalability。

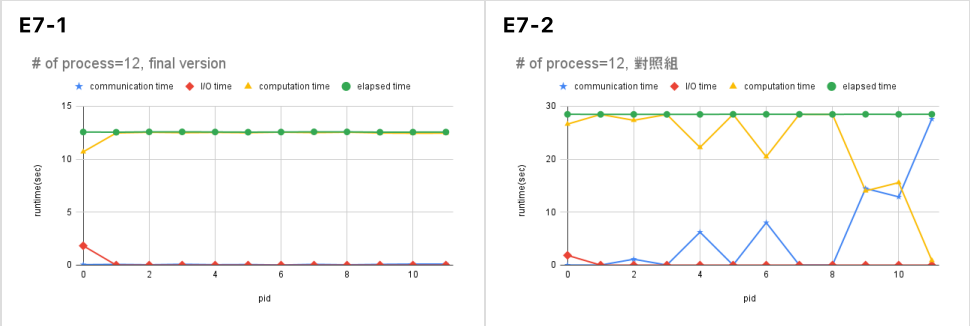
- Single node的環境下，平均分配連續幾條row給每個process的實作方式所產生的實驗結果(以下將統稱為對照組)





在這裡可以很明顯的看到communication time大幅的上升了，因為在對照組的實作方法是像HW1的時候一樣每個process分配一段連續的工作，但是這樣一來就會導致每個process的工作量差距很大，再加上我使用 `MPI_Gatherv` 去把所有local的rows組起來然後最後再寫出來，而這個是blocking的API，所以process之間互等又需要花上一大段的時間而導致無論是在performance還是在scalability的部分都不如實驗組(final version)。

Load Balance



在這裡可以看到final version的load balancing比對照組好上非常多，尤其在communication time以及computation time的部分，因為如果load balancing做不好的話就會導致在同個時間點下，有一些process是沒有事情可以做的，而這樣無疑是一種浪費計算資源的實作方法，而communication time的部分同樣的因為每個process的工作量不同再加上使用blocking的API，所以就會有很多工作量較少的process需要等待那些工作量多的process做完之後才能統一在rank0的時候寫出。

Optimization Strategies

從**Analysis-of-Results**的實驗結果可以看到load balancing對於performance真的有很重大的影響，尤其在hybrid的部分，我原本是使用對照組的實作方式，在改成後來較為load balancing的版本之後，整體的performance上升了非常多。此外，透過實驗的結果可以看到I/O time和communication time是主要的bottleneck，在這個部分因為我覺得如果可以把communication拿掉然後讓各個process自己把自己負責的pixels直接寫到png的話應該能夠很顯著的使performance上升，不過code可能會變得很複雜。

Discussion

SCALABILITY

在scalability的部分，正如我在**Optimization-Strategies**所提到的，我的實作還是會有I/O和communication的bottleneck，而如果可以針對寫出資料的部分再進行平行化的話，效能應該會有很顯著的提升，至於computation的部分我認為scalability已經接近ideal了，如果要再進行優化的話我覺得應該可以針對一些有branch的部分單獨拉出來做，像是如果是if的話就把他的condition那邊再做平行化，而有一些

intrinsic像是RISC-V V extension就可以把branch的條件弄成mask然後每種Arithmetic也都有支援masked的version，如果是改為這樣的實作方式的話可能可以更貼近ideal。

LOAD BALANCE

在load balance的部分，我覺得在我的實作方式中，pthread那邊用mutex lock去搶工作的部分應該無論是在哪一種類型的data都能夠有不錯的performance的。而hybrid那邊final version的做法在這次的作業可能有不錯的load balancing，但是這並不能當作general solution。

Experience & Conclusion

在這次作業中，我深深的體會到了load balance的重要性，同時在optimize sequential code的時候，我也發覺到其實很多data都是可以獨立做計算的，然後突然理解到為什麼GPU一開始是發明來進行圖片運算的，因為那些圖片的計算還有寫出的部分其實都不太會有data dependency的問題，所以就可以透過GPU加速高解析度的計算，這是我以前都沒有想到過的事情。另外在這次作業我覺得最大的困難大概就是array的index會不小心取到範圍外而導致segmentation fault的問題了，在這個部分真的要非常的細心，幸好有address sanitizer不然我不知道要debug de多久。