Homework 2: Mandelbrot Set

108062313 黃允暘

Implementation

以下會根據 pthread 及 hybrid 版本來講解：

Pthread

1. partition the task:

為了要有效利用 pthread 完成平行話，首先要定義出執行工作的基本單位。我將一個 row 的計算當作是一個計算的基本單位。定義出基本單位之後，接下來就可以分配給 threads 去執行了。考慮到每個 row 的計算量都不一樣，為了避免 threads 互相等待的情況，我在這邊採用 dynamic load balancing 的策略。

實際作法是設置一個變數 finished_row，這個變數可以記錄目前執行到哪個 row。當 thread 要拿一個 row 來計算時，他會拿 finished_row 的下一個 row，並把 finished_row++。這樣一來，如果有 thread 比較快完成工作的話，就可以立刻再拿一個 row 來執行。不過，因為可能有多個 threads 同時要讀取並更改 finished_row 的值，所以在這邊會用 mutex 區隔出一個 critical section，並在裡面對 finished_row 做讀寫。

```
pthread_mutex_lock(&mutex);
if ( finished_row < height){
    cur_row=finished_row;
    finished_row++;
}
else{
    cur_row=-1;
}
pthread_mutex_unlock(&mutex);
```

2. calculate pixel

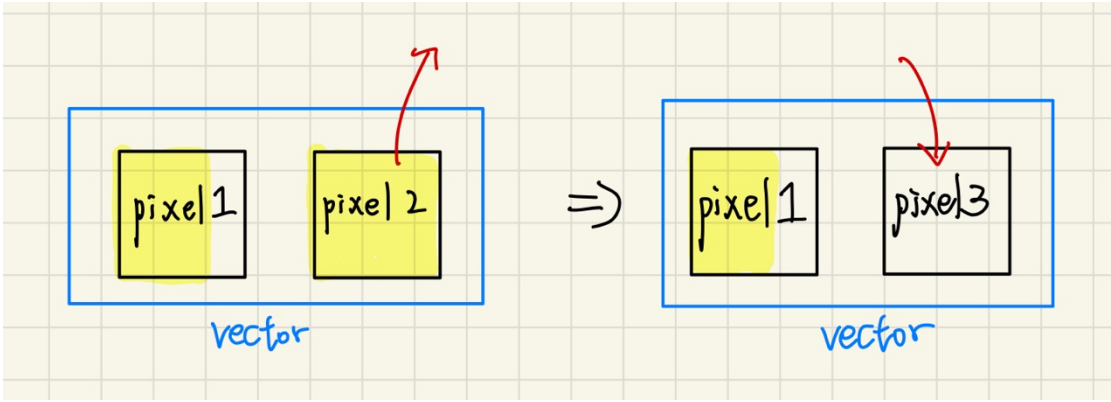
(a) vectorization

當每個 thread 都拿到一個 row 時，接下來就是要計算在 row 中的每個 pixel 的值。我在這邊一開沒有做任何的優化，就是直接 for 迴圈遍歷整個 row，結果好時非常久。因此我後來利用了 vectorization 的技術去實作。因為 SSE2 支援 128bits 的 vectorize，而一個 double 為 64bits，所以在這邊可以一次執行兩個數字的計算，也就是可以同時計算兩個 pixel。為了要能夠利用 SSE2 座計算，要將變數設為__m128d 的型態。

(b) load pixel into vector

不過每個 pixel 所需要的計算時間也不同，我在這邊為了節省時間，將

這兩個數字的計算完全獨立開來。也就是說，在 vector 中任一個 pixel 計算完後，會立刻將下一個 pixel 丟進來，如下圖。



而將一個新的 pixel 丟進 vector 時，要將一些計算 pixel 的參數值做更新，也要更新 pixel 的座標(row, col)。

```
if(finish[0]){
    finished_col++;
    cur_col1 = finished_col;
    x0_vec[0] = cur_col1 * x0base + left;
    x_vec[0] = 0;
    y_vec[0] = 0;
    finish[0] = 0;
    repeats[0]=0;
}
```

不過需要注意的是，因為 vector 裡面一定要有兩個 pixel，且不能是同一個 pixel，因為會有 dependency 的問題。所以當剩下最後一個 pixel 得時候，就不能用 vector 的方式去計算了，需要用原本 double 的方式計算最後一個 pixel 的值。

(c) computing vector

這部分就只是單純的把範例 code 轉換成 vector 的版本。

```
__m128d temp = _mm_add_pd(_mm_sub_pd(_mm_mul_pd(x_vec, x_vec), _mm_mul_pd(y_vec, y_vec)), x0_vec);
y_vec = _mm_add_pd(_mm_mul_pd(_mm_mul_pd(x_vec, constant_2), y_vec), y0_vec);
x_vec = temp;
length_squared_vec = _mm_add_pd(_mm_mul_pd(x_vec, x_vec),_mm_mul_pd(y_vec, y_vec));
repeats[0]+=1;
repeats[1]+=1;
```

Hybrid

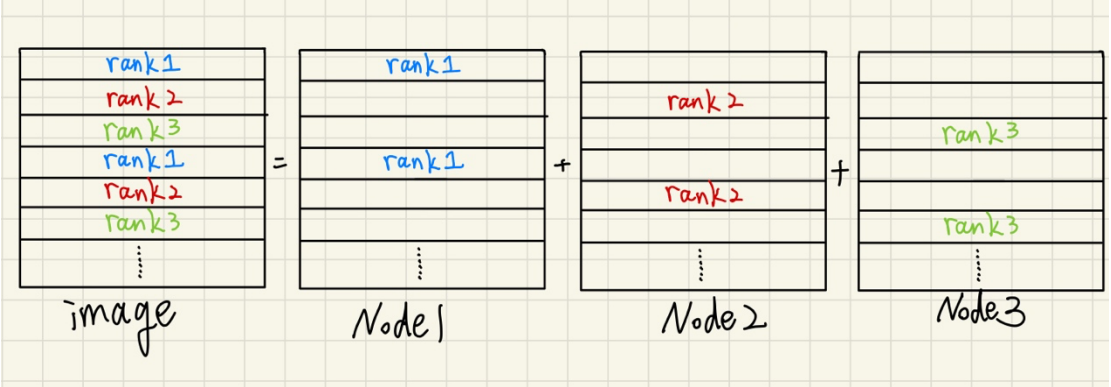
我在 Hybrid 部分的實作，計算方面基本是跟 pthread 部分是一樣的，因此在這邊會著重講解如何以 MPI 及 OpenMP 達到平行化。

1. MPI:

在這個部分一樣是以一個 row 為計算單位。不過就是用 static loading 的方式去做 loading，會依照 rank 的值去分配需要計算的 row。而考慮到不

同的 row 的計算量可能會相差很多，因此在這邊會讓每個 Node 用穿插的方式去計算每個 row，盡量讓每個 Node 的計算工作相同。另外，每個 Node 都會有自己的 local_image，而 Node 只需要計算他們被分配到的 row 就可以了，如下圖。

```
for(int row=rank; row<height; row+=size){
    cal_pixel(row);
}
```



而當每個 local_image 都計算完之後，就可以利用 MPI_Reduce()，將每個 Node 的 local_image 加起來，這樣就會是一張完整的 image 了。

```
MPI_Reduce(local_image, image, image_size, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

2. **OpenMP:**

這部分相對簡單，我主要就是將程式當中可以用 for 迴圈執行，且迴圈內執行的內容沒有 dependency 的地方，用 OpenMP 平行化。至於 schedule 的部分，考慮到每個 row 的季算量不同，採用 dynamic 的方式，並將 chunk 設為 5。而 num_threads 就設為 processors 的數量。我在這邊發現 chunk 的值對 Performance 有蠻大的影響，因此在這邊是了蠻多值，最後定在 5。

```
#pragma omp parallel num_threads(ncpus)
{
    #pragma omp for schedule(dynamic, 5)
    for(int row=rank; row<height; row+=size){
        cal_pixel(row);
    }
}
```

Experiment & Analysis

1. **Methodology**

(a) System spec: 學校的系統環境

- 1 login node (apollo31) (200%CPU max)
- 18 compute nodes (1200% CPU max)
- Use `squeue` to view SLURM usage
- Cluster monitor: http://apollo.cs.nthu.edu.tw/monitor
- 48GB disk space per user

(b) Performance Metrics:

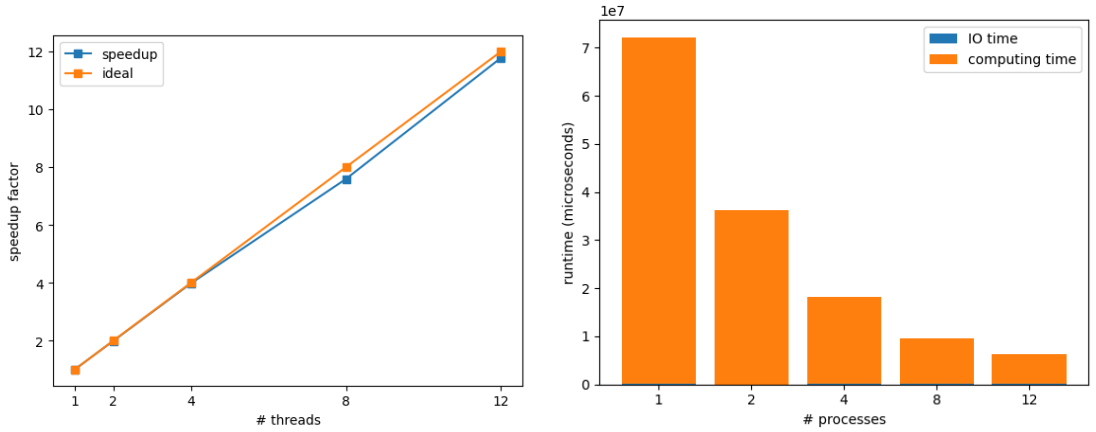
我利用 std::chrono::steady_clock 去計算程式執行的時間，計算的方式如下圖。

```
auto compute_start = std::chrono::steady_clock::now();
{…
auto compute_end = std::chrono::steady_clock::now();
double ComputeTime = std::chrono::duration_cast<std::chrono::microseconds>(compute_end-compute_start).count();
```

2. Strong Scalability & Load Balancing

Pthread

(a) Speedup Factor & Time Profile




```
54564 -0.34499 -0.34501 -0.61249 -0.61251 800 800
```

Input parameters

從 speedup factor 的圖表上可以看到，當 thread 的數量增多的時候，speedup factor 也有隨之上升。且隨著 threads 持續增加，speedup 的程度也幾乎與 ideal 的線條重疊。因此可以了解到，程式的 scalability 非常好。我另外也把 time profile 給畫出來。可以看到 IO 幾乎沒有站任何時間。因此程式的 bottleneck 及為 computing time。而 computing time，也隨著 threads 的數量成比例下降，代表我的程式有很好利用 threads 來做運算。

(b) Load Balancing

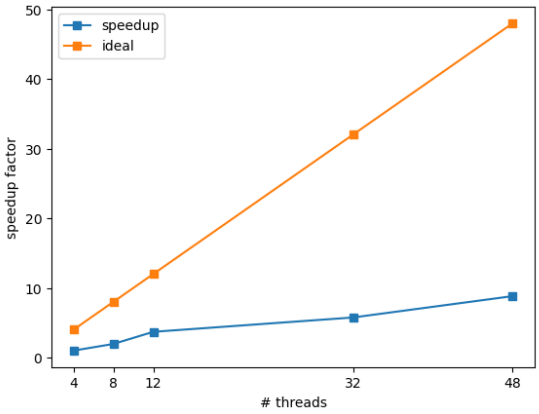
	p=1	P=2	p=4	p=8	p=12
Average time (microseconds)	72132763	36136066	18063279	9194396	6044488
Diff (microseconds)		57694	70456	25836	51667
ratio		0.15%	0.39%	0.28%	0.85%

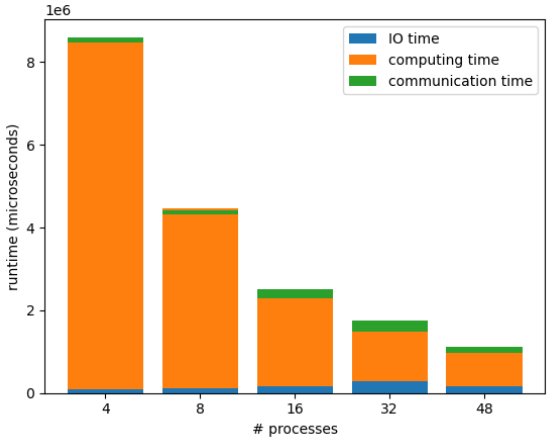
圖表中的 diff 代表該次任務中，執行速度最快的 threads 與執行速度最慢的 thread 的差距。可以看到 diff 在平均執行時間的佔比都不到 0.1%，代表程式有很好的 load balancing。因此我認為 dynamic load balancing 在這項任務中的效果很好。(Average time 代表全部 threads 的平均執行時間，ratio 代表 Average time / Diff)

Hybrid

(a) Speedup Factor & Time Profile

固定為 4 個 Node，並調整 threads 的數量。






```
10000 -0.34499 -0.34501 -0.61249 -0.61251 3199 3199
```

Input parameters

可以看到在 hybrid 中，speedup 的程度並沒有像 pthread 版本一樣那麼好，我認為這是因為多了 communication 的 tradeoff 緣故。因此我也將 Time Profile 畫處來，並發現 communication time 確實有隨著 processes 上升而上升的趨勢。

(b) Load Balancing

固定為 4 個 threads，並調整 Node 的數量，以測試 Node 之間的 load balancing。

	N=1 p=4	N=2 p=4	N=3 p=4	N=4 p=4
Average time (microseconds)	8524488	4428995	3002319	2293828
diff		64577	20845	97833
ratio		1.45%	0.69%	4.26%

我在 hybrid 用的是 static load balancing，會讓每個 Node 用穿插的方式去計算圖片的每個 row，盡量讓每個 Node 的計算工作相同。不過 ratio 的值普遍比 thread 版本的還要高，代表 thread 版本的 load balancing 做得比較好。推次是因為在這邊採取 static load balancing，儘管有設計一些做法希望能讓每個 Node 的工作量相同，不過還是比不過 dynamic load balancing。

Conclusion

我在這一次作業花了很多時間在 vectorization 上面，有部分原因是因為不熟悉 SSE 的寫法，另一個部分是想要將 vector 內的 pixel 獨立處理，因此花了蠻多時間，不過 vectorization 確實使程式的表現大幅度提升。相較於 Hw2a 寫的很久，Hw2b 一下就寫完了，因為基本的架構都在 Hw2a 完成了。不過 Hw2b 的 scoreboard 表現卻比 Hw2a 在 scoreboard 上的表現好很多。這次的作業使我更了解 pthread、 MPI 及 OpenMP。