Parallel Programming HW2
Mandelbrot Set
ssh: pa128427359、程祥恩

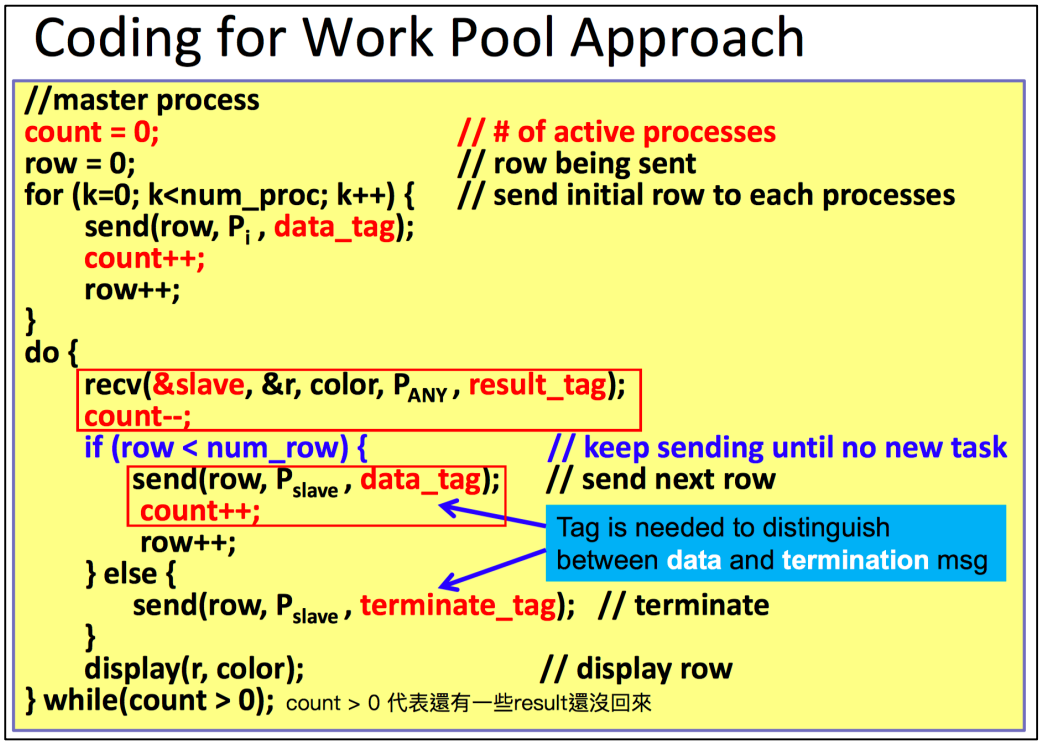
1. Implementation

i. MPI with Static Scheduling

把輸入的資訊橫著切，也就是說每一個 Process 所負責的寬度都一樣而高度不一樣。假設有 n 個 Process，首先要把輸入的 height 分成四份，接著修改一般版本的 Mandelbrot Set，把最外層的 for 迴圈（0 ~ height），改成每一個 Process 所得到的高度上下界（上界 p_height_lower = height * rank / size, 下界 p_height_upper = height * rank / size）進行運算，每個 Process 算出來值都存到各自的 image 陣列當中。最後，因為每一個 Process 所負責的 pixel 不一定相同，所以必須使用 MPI_Gatherv()來集合所有 Process 的 image 陣列，並存到 root 的 image_gather 陣列當中。

ii. MPI with Dynamic Scheduling

主要是參考課堂投影片的 pseudocode 去實作，使用 Work Pool Approach。會把 rank == 0 的機器當作 Master，不會實際去運算，因此在 n=2 時反而會因為過多無謂的傳送資料導致效率比 static 來得差。



Coding for Work Pool Approach

```
//slave process P ( i )
recv(&row, P_master , source_tag);   接收來自Master發配的任務
                                     如果收到的是termination，就結束這個process
while (source_tag == data_tag) {          // keep receiving new task
      c.imag = min_imag + (row * scale_image);
      for (x=0; x<640; x++) {
          c.real = min_real + (x * scale_real);
          color[x] = cal_pixel (c);       // compute color of a single row
      }
      send(i, row, color, P_master , result_tag); // send process id and results
      recv(&row, P_master , source_tag);
}
```

iii. OpenMP (Either Static or Dynamic Scheduling)

在計算 Mandelbrot Set 的外層 for 迴圈之前，加上

#pragma omp parallel num_threads(num_threads) shared(image)

#pragma omp for schedule(dynamic)

當每個 thread 要把計算結果存回 image 時，必須要對應到每一個 thread 所分配到的區塊的 index，也就是 image[j * width + i] = repeats，最後得到的 image 就會是正確答案。

iv. Hybrid parallelism – MPI + OpenMP

首先把輸入的資訊橫著切，接著對每個 Process 分配到的高度，同樣加上

#pragma omp parallel num_threads(num_threads) shared(image)

#pragma omp for schedule(dynamic)

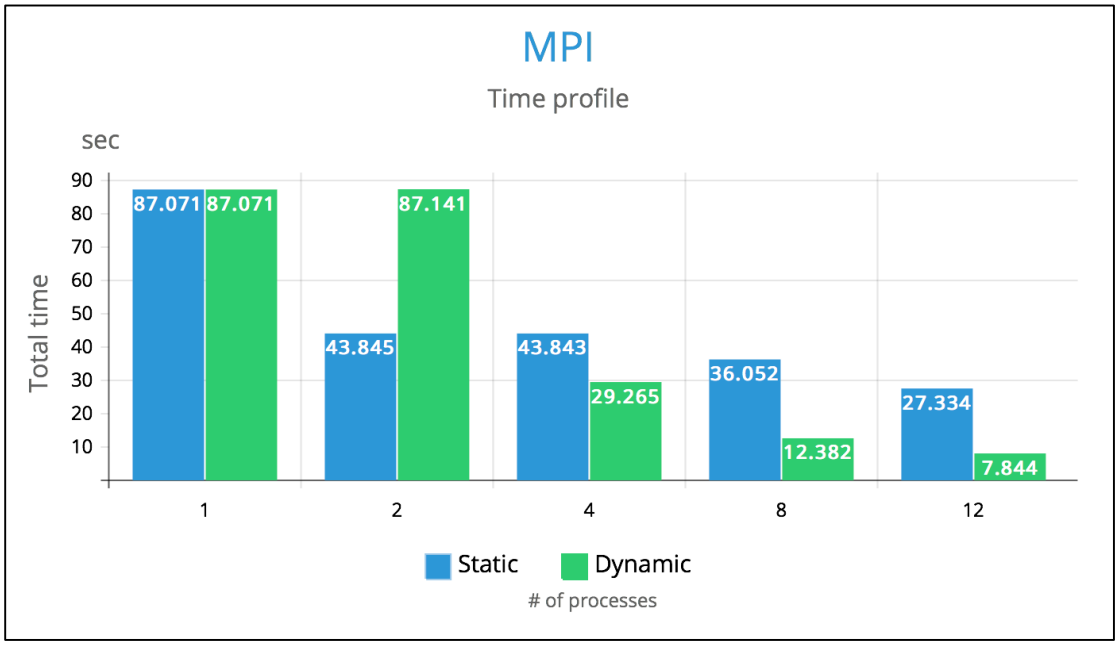
去執行 Mandelbrot Set，但每個 thread 要把計算結果存回 image 時，必須要這樣寫 image[j * width + i - count] = repeats，其中 count 是每一個 process 在最後集合成 image_gather 時，所負責的 index offset。

2. Experiment & Analysis

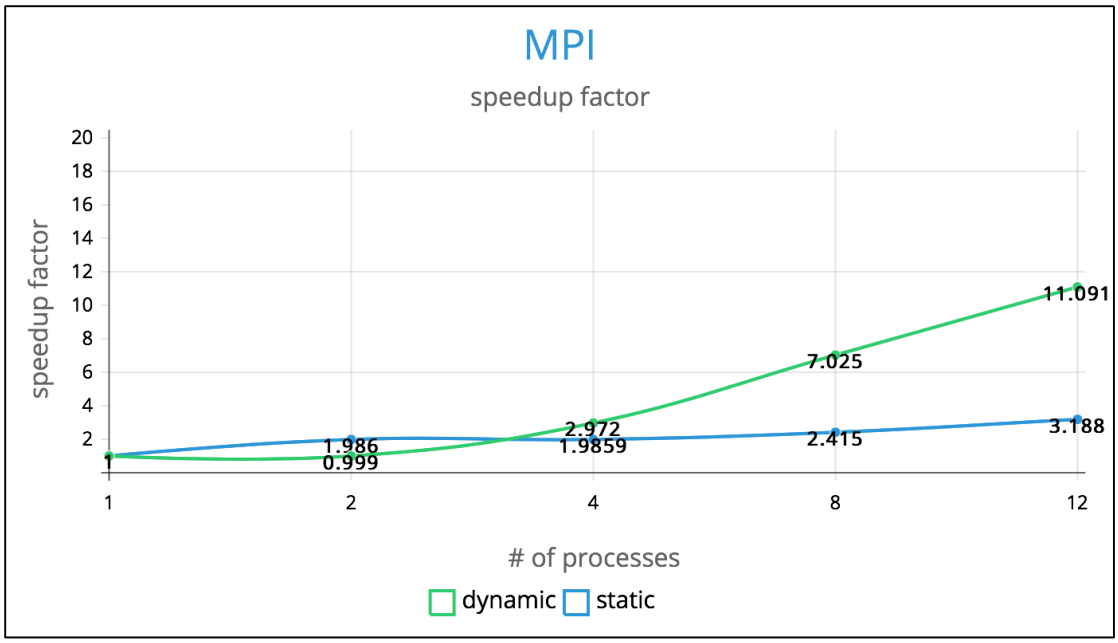
Problem size: (left, right, lower, upper) = (-2, 2, -2, 2), pixel = (1000 x 1000)

i. Time profile and speedup factor

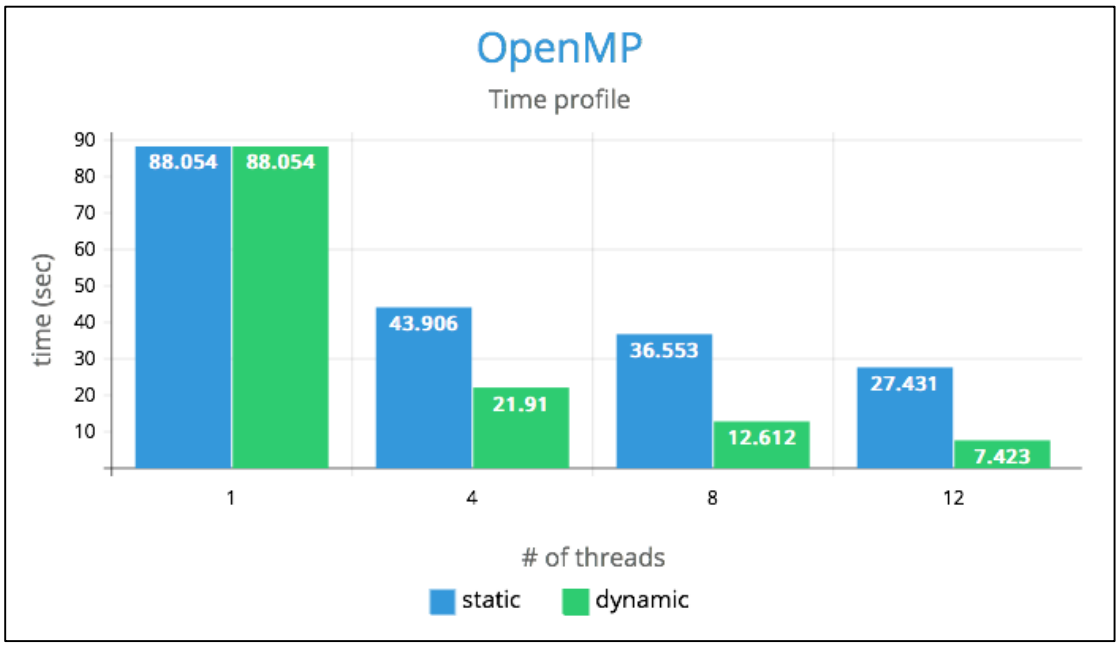
a. MPI with Static Scheduling & Dynamic Scheduling

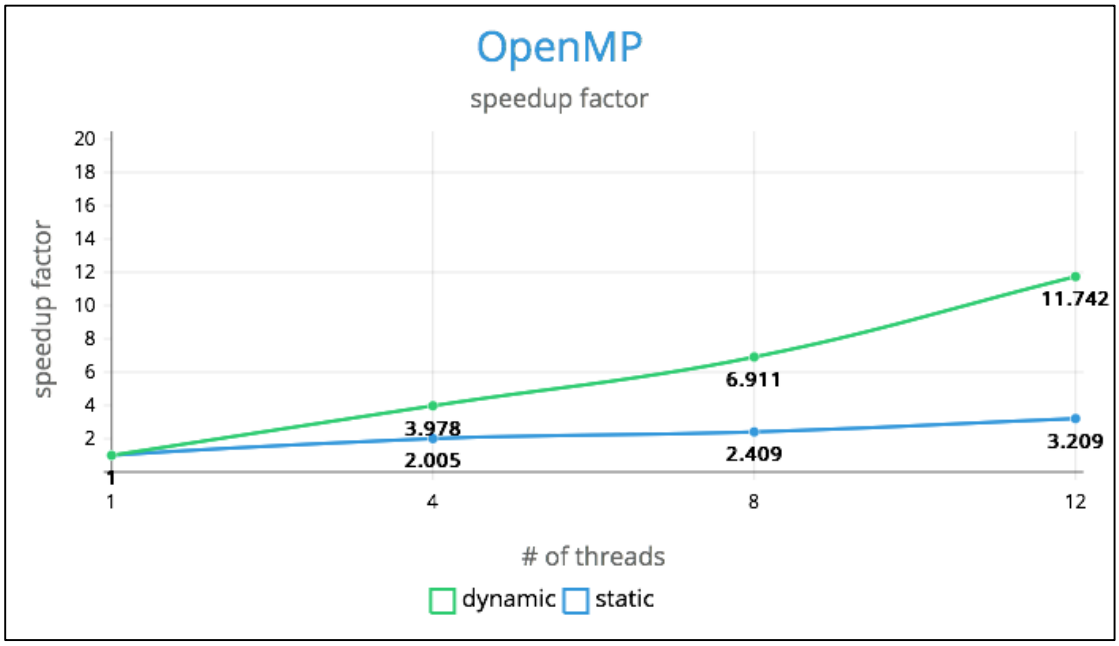


# of processes Static/Dynamic	1	2	4	8	12
I/O time (sec)	0.076				
Comm time (sec)	0 / 0	0.013 / 0.021	0.020 / 0.037	0.023 / 0.054	0.036 / 0.068
CPU time (sec)	86.995 / 86.995	43.766 / 87.044	43.749 / 29.152	36.403 / 12.252	27.292 / 7.7

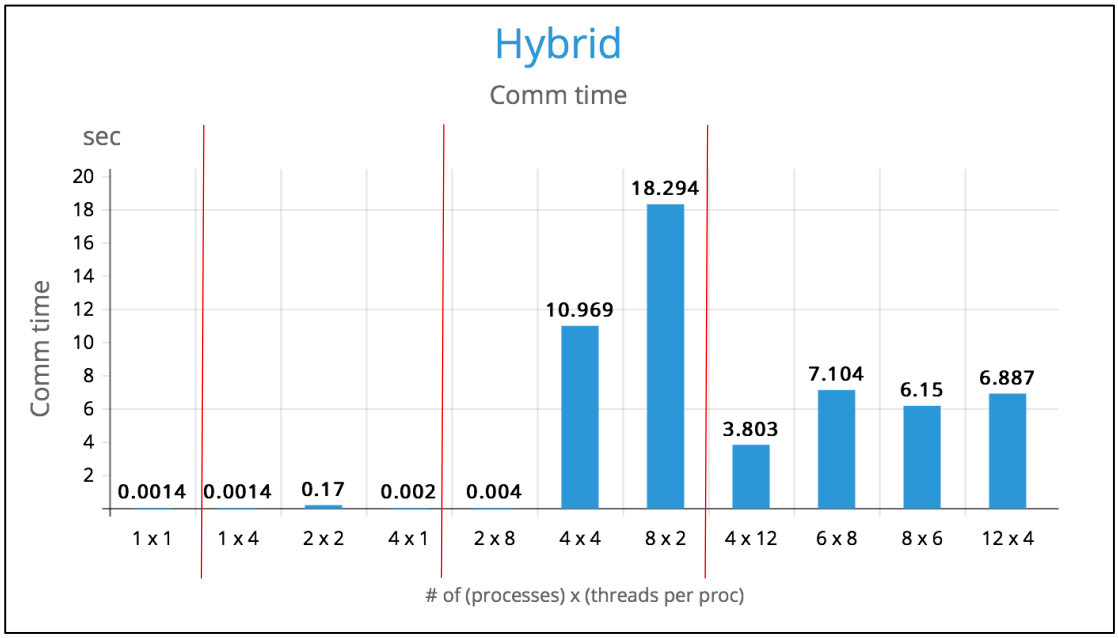

MPI
speedup factor
speedup factor
20
18
16
14
12
10
8
6
4
2
1
0.999
1.986
1.9859
2.972
7.025
2.415
11.091
3.188
1
2
4
8
12
# of processes
dynamic
static

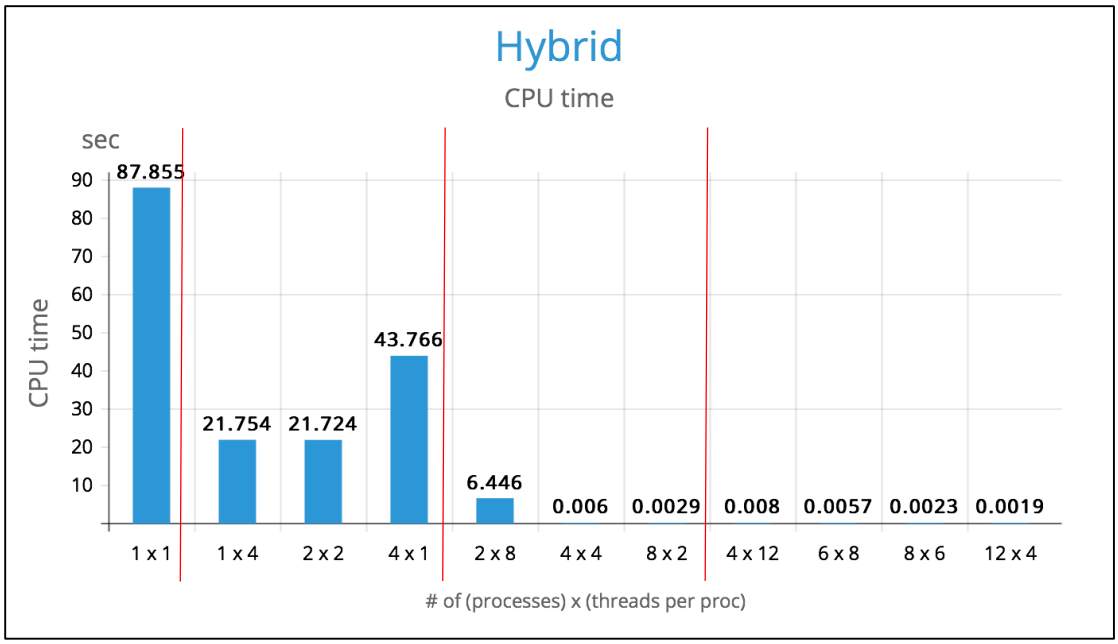
b. OpenMP (Either Static or Dynamic Scheduling)

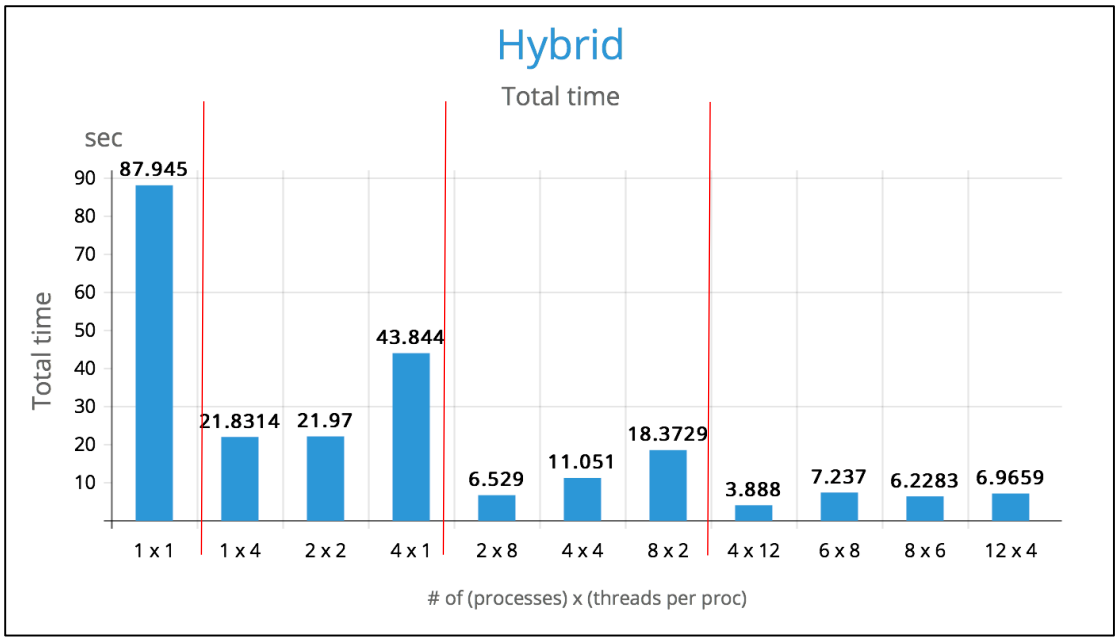


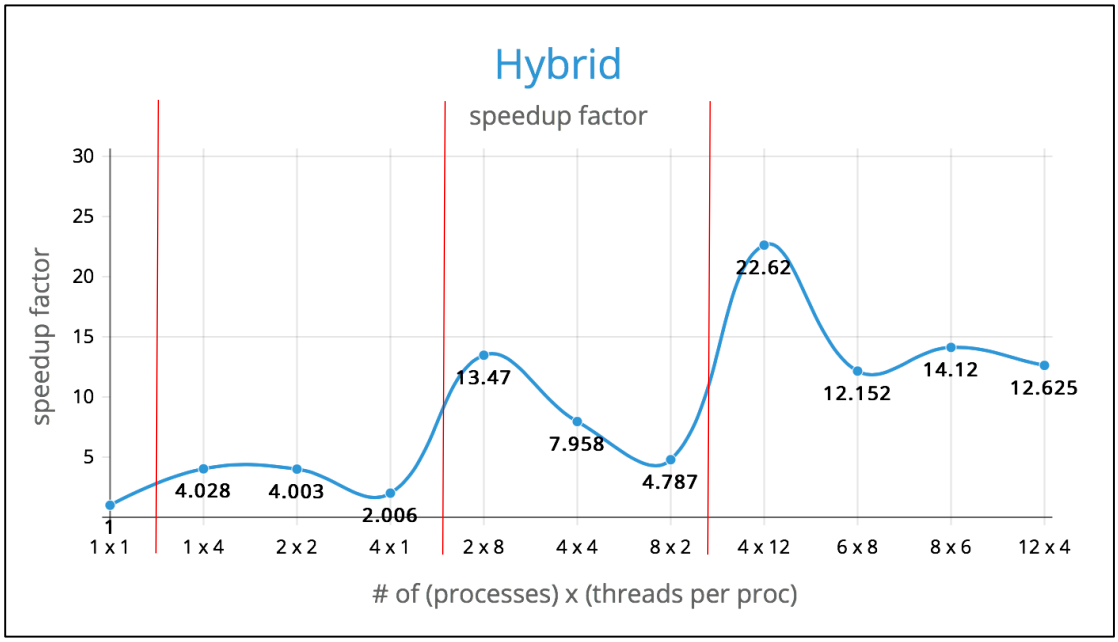


c. Hybrid parallelism – MPI + OpenMP






Hybrid
Total time
sec
Total time
90
80
70
60
50
40
30
20
10
87.945
21.8314
21.97
43.844
6.529
11.051
18.3729
3.888
7.237
6.2283
6.9659
1 x 1
1 x 4
2 x 2
4 x 1
2 x 8
4 x 4
8 x 2
4 x 12
6 x 8
8 x 6
12 x 4
of (processes) x (threads per proc)


Hybrid
speedup factor
speedup factor
30
25
20
15
10
5
1
4.028
4.003
2.006
13.47
7.958
4.787
22.62
12.152
14.12
12.625
1 x 1
1 x 4
2 x 2
4 x 1
2 x 8
4 x 4
8 x 2
4 x 12
6 x 8
8 x 6
12 x 4
of (processes) x (threads per proc)

ii. Discussion

a. MPI with Static Scheduling

在 n = 2 時擁有良好的 speedup，推測原因是 1000x1000 的 Mandelbrot Set 圖形上半部和下半部的是對稱的，所以剛好具備很棒的 Load Balance。一旦 n > 2，每一個 Process 處理的運算量就會開始有很大的差異，導致花費很多時間在等待對方完成，Load Balance 和 Scalability 急劇下降。

b. MPI with Dynamic Scheduling

和 MPI Static 相反，在 n=2 時由於其中一台 process 是 Master，所以實際上只有一台 process 在計算，反而和 n=1 的效能差不多，不過一旦 n 的數量增加，效果就會遠遠優於 MPI Static，在 speedup factor 方面雖不如 Dynamic OpenMP，但仍然有超水準的演出。

c. OpenMP (Either Static or Dynamic Scheduling)

Static: 和 MPI with Static Schedule 差不多，因為是靜態分配工作項目，導致每一條 Thread 的工作量不同，花費很多時間在等待對方完成，因此 Scalability 和 Load Balance 皆不理想。

Dynamic: 因為是動態分配工作，所以每一條 Thread 的工作量都大致相同，加上由於所有 Thread 都在同一台機器裡執行，所以具備極低的 Communication time。在 Scalability 和 Load Balance 的表現上都很優秀。

d. Hybrid parallelism – MPI + OpenMP

在總 Thread 數量相同的情況下（例如 4x12 和 12x4），動用到越少 Process（也就是每一個 Process 的 Thread 越多）代表著越高的 Speedup，也擁有不錯的 Load Balance 和 Scalability，在選擇 Process 和 Thread 的分配上，應該盡量選擇 Thread 較多而 Process 較少的設置，能夠大幅減少 Communication time，同時又不至於過度拖慢 CPU time。

3. Conclusion

在這次作業中我們發現，如果沒辦法事先平分每個 Process 的 Load Balance，那就應該使用動態分配工作給每一個 Process，否則的話就會像本題的實驗結果一樣：使用靜態分配使得每個 Process 的工作量差異極大，導致工作量少的 Process 花費大量時間等待其他 Process 執行完畢。當 Process 數目越多，Speedup 就成長得越慢，甚至停滯不前。一旦使用的動態分配，整個系統將會擁有良好的 Scalability 和 Load Balance，即便 Process 數量很多，Speedup 也可以維持在一定的水準，若想進一步提升效能，則可以適度地在每個 Process 中多開幾條 Threads，這麼一來即可大幅降低 Communication time，同時也不會過度影響 CPU time。