Parallel Programming HW2 Mandelbrot Set

112065528 洪巧雲

HackMD (https://hackmd.io/@elliehung/SkjiDMbWyx)

Implementation

Pthread

- Pthread 的做法其實相對 MPI 以及 OpenMP 簡單很多，首先它不需要考量 rank 的分法，並且 default 的 process 數量為 1。程式一開始會根據 CPU 數量 (`num_cpus`)

```
cpu_set_t cpu_set;
sched_getaffinity(0, sizeof(cpu_set), &cpu_set);
int num_cpus = CPU_COUNT(&cpu_set);
```

去決定 threads 的數量。

```
/* create threads for mandelbrot set calculation */
for (int t = 0; t < num_threads; ++t) {
    thread_data[t].thread_id = t;
    thread_data[t].iters = iters;
    thread_data[t].left = left;
    thread_data[t].right = right;
    thread_data[t].lower = lower;
    thread_data[t].upper = upper;
    thread_data[t].width = width;
    thread_data[t].height = height;
    thread_data[t].image = image;
    thread_data[t].num_threads = num_threads;
    pthread_create(&threads[t], NULL, mandelbrot, (void*)&thread_data[t]);

}
```

- 再用 `pthread_create()` 去呼叫 mandelbrot sort，寫了一個 struct 把 `ThreadData` 弄在一起:

```c
typedef struct {
    int thread_id;
    int iters;
    double left;
    double right;
    double lower;
    double upper;
    int width;
    int height;
    int* image;
    int num_threads;
} ThreadData;
```

- 傳入 `void mandelbrot()` 的變數是 `(void*)&thread_data[t]` ，進入 `void mandelbrot()` 後，會是一條條 thread 去平行化完成計算。
- 每條 thread 中的計算量，我是由 `j: from thread_id to height` ，並且每次都增加 `j = j + num_threads` ，其中 `height` 為圖片總行數，每條 thread 所進行的 task 量均為 `height / num_threads` ，進入到每條 thread 後，會開始去計算 `i = 0 ~ width` ，一個個 pixel 去計算。

Hybrid

- 在理解完 Spec 之後，使用了跟 HW1 一樣的 partition method 去分每個 rank 所 input 的資料量:

```c
/* allocate memory for the local image slice */
int local_height = (rank == size - 1) ? (height / size + height % size) : (height / size);
int other_local_height = rank * (height / size);
int *local_image = (int *)malloc(width * local_height * sizeof(int));
assert(local_image);


int num_threads = omp_get_num_threads();
int thread_rank = omp_get_thread_num();
int end = local_height;

unsigned long long thread_start = thread_rank * (end / num_threads);
unsigned long long thread_end = (thread_rank == num_threads - 1) ? end : ((thread_rank + 1) * (end / num_threads));
```

- 我的方法是將 **height** 的資料平均分到每個 rank 中: `local_height = (rank == size - 1) ? (height / size + height % size) : (height / size)`
    - `local_height` 就是每個 rank 要處理的資料大小。
- 多出來的資料我把它們都放到最後一個 rank 中: `(rank == size - 1) ? local_height = (height / size + height % size)`
- 在進入 `#pragma omp parallel for schedule(dynamic, 1)` 標籤後，會開始使用 OpenMP 進行 threads 平行化，而這個部分的實作是把 `j = thread_start ~ thread_end` 去跑 for loop， `thread_start` 也就是這條 thread 的起始 index 會等於 `thread ID * 此條 thread 被分配到的資料量`

- 此條 thread 被分配到的資料量: `此 rank 被分配到的資料量: local_height / 總共有幾條 thread: numbers of threads`
- `thread_end`：為此條 thread 最後的 index。
- 接下來就會開始一個個 pixel 做計算。

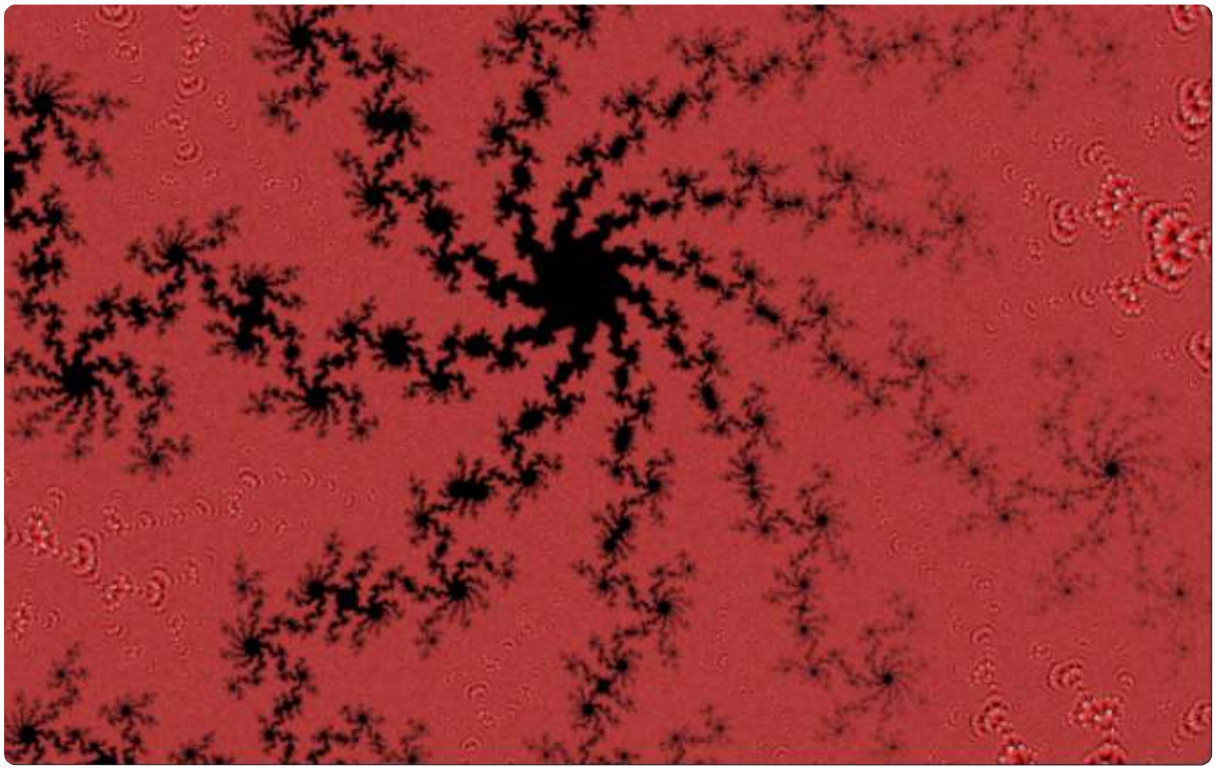
Experiment & Analysis

Intel Vtune

Profiling result:

- `./hw2_runVtune.sh -n4 -c6 hw2b out.png 10000 -0.5506164691618783 -0.5506164628264113 0.6273445437118131 0.6273445403522527 7680 4320`
- 使用最耗時的 testcase 做實驗: `strict34.txt`

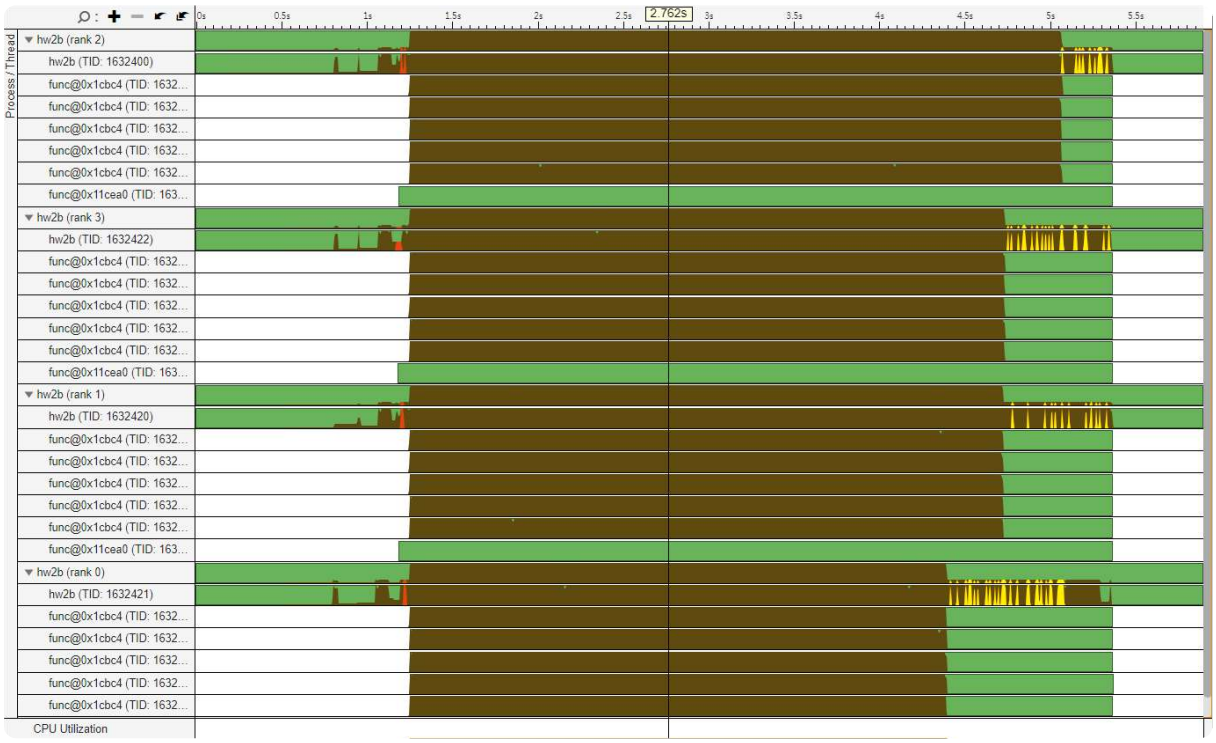
  `strict34.png`



`strict34.txt`： `10000 -0.5506164691618783 -0.5506164628264113 0.6273445437118131 0.6273445403522527 7680 4320`

- `-n4 -c6`：n 代表的是 Node，c 代表的是 threads 數量
- 不同實驗不同 n 與 c，會得出不同的結果

- 以下為 `Hybrid -n4 -c6 strict34.txt` 的執行結果:



可以看到每個 process 中的每條 thread 的執行都分布的很均勻，代表在 thread 中有 load balance。

NVIDIA Night Sight

- 在第一次使用 Night Sight 時，出現了 error message，結果發現是因為一個粗心的錯誤，module load 成 mpi 而不是 openmpi

```
pp24s094@nthu-master:~/hw2/sample$ ./hw2b_runNsight.sh -n4 -c6 hw2b out.png 10000 -0.5506164691618783 -0.5506164628264113 0.6273445437118131 0.6273445403522527 7680 4320
rm -f hw2seq hw2a hw2b hw2seq.o hw2a.o hw2b.o
mpicxx -lm -O3 -g3 -fopenmp -lm -lpng -lmpi -mavx512f -march=native    hw2b.cc  -lpng -o hw2b
+ cd nsight-result
+ mkdir -p hw2b_2024-10-26_20-38-44
+ cp ../hw2b.cc hw2b_2024-10-26_20-38-44/hw2b.cc
+ cp ../hw2b hw2b_2024-10-26_20-38-44/hw2b
+ srun -n 4 -c 6 ./wrapper.sh hw2b_2024-10-26_20-38-44 .././hw2b out.png 10000 -0.5506164691618783 -0.5506164628264113 0.6273445437118131 0.6273445403522527 7680 4320
Abort(610309) on node 3 (rank 3 in comm 0): Fatal error in PMPI_Comm_rank: Unknown error class, error stack:
PMPI_Comm_rank(106): MPI_Comm_rank(comm=0x0, rank=0x7ffc0c5e57a0) failed
PMPI_Comm_rank(63).: Invalid communicator
Abort(610309) on node 0 (rank 0 in comm 0): Fatal error in PMPI_Comm_rank: Unknown error class, error stack:
PMPI_Comm_rank(106): MPI_Comm_rank(comm=0x0, rank=0x7fffa76f0120) failed
PMPI_Comm_rank(63).: Invalid communicator
Abort(671698949) on node 1 (rank 1 in comm 0): Fatal error in PMPI_Comm_rank: Unknown error class, error stack:
PMPI_Comm_rank(106): MPI_Comm_rank(comm=0x0, rank=0x7fff33270020) failed
PMPI_Comm_rank(63).: Invalid communicator
Abort(470372357) on node 2 (rank 2 in comm 0): Fatal error in PMPI_Comm_rank: Unknown error class, error stack:
PMPI_Comm_rank(106): MPI_Comm_rank(comm=0x0, rank=0x7fffdd281e60) failed
PMPI_Comm_rank(63).: Invalid communicator
srun: Job step aborted: Waiting up to 32 seconds for job step to finish.
slurmstepd: error: *** STEP 245441.0 ON nthu-cn01 CANCELLED AT 2024-10-26T20:38:46 ***
srun: error: nthu-cn01: tasks 0-3: Killed
+ '[' 137 -ne 0 ']'
+ echo 'srun nsight failed, stopping the script.'
srun nsight failed, stopping the script.
+ exit 1
```

```
pp24s094@nthu-master:~/hw2/sample$ module list
Currently Loaded Modulefiles:
 1) mpi/latest   2) intel-vtune/2024   3) nvtx/3   4) nsys/2024.5.1

Key:
auto-loaded
```

- `module swap mpi openmpi` 後:

```
pp24s094@nthu-master:~/hw2/sample$ module swap mpi openmpi
Switching from mpi to openmpi/4.1.6
  Loading requirement: ucx/1.17.0
```

成功執行 Nsight。

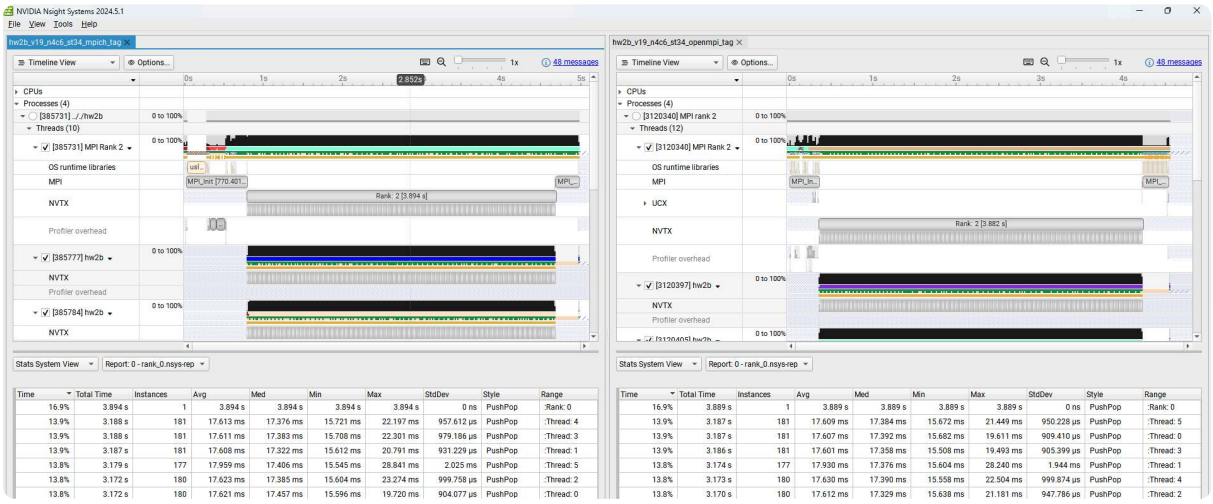
之後再跑了一次 `hw2b-judge` ，發現快了 26.47 seconds

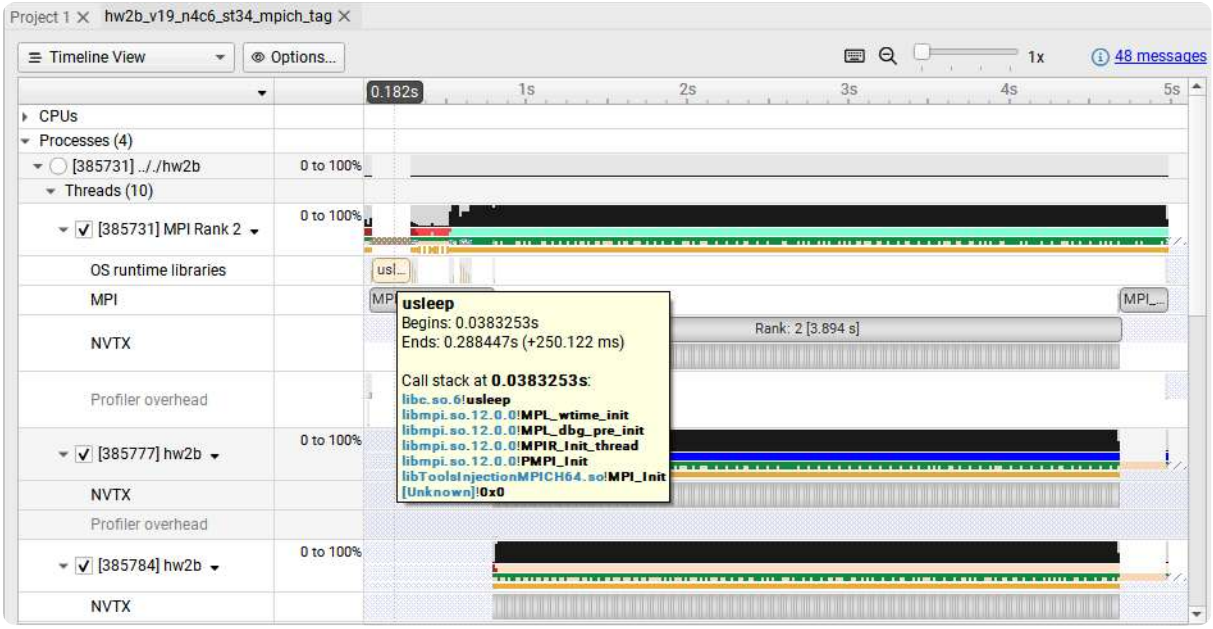
`mpi: {68 116.03} --> openmpi: {68 89.56}`

```
Removing temporary directory /share/judge_dir/.judge_exe.3904426292
Scoreboard: updated {68 116.03} --> {68 89.56}
```

- 因為這個粗心的錯誤去研究了一下 為甚麼這兩個 module 會差這麼多:

- 更換不同 module run Nsight:





- 發現使用 **Intel MPI** 的 Module 的實驗結果顯示它會有一段不短的 `usleep` 時間 (睡了 250 ms)，原因在上網查過後發現應該是因為它 usleep 的時間實際讓是在等待程式執行。

- **OpenMPI** 則是使用 `epoll_wait` 在等待 I/O 的時間，也是為甚麼它是占用 OS 比例最高的時間。

Time	Total Time	Num Calls	Avg	Med	Min	Max	StdDev	Name
96.1%	8.031 s	18	446.162 ms	2.926 ms	3.825 µs	4.164 s	1.246 s	epoll_wait

- **Intel mpi** 也有 `poll` 的時間:

Time	Total Time	Num Calls	Avg	Med	Min	Max	StdDev	Name
91.5%	4.350 s	49	88.782 ms	108.113 ms	27.117 ms	108.188 ms	27.087 ms	poll

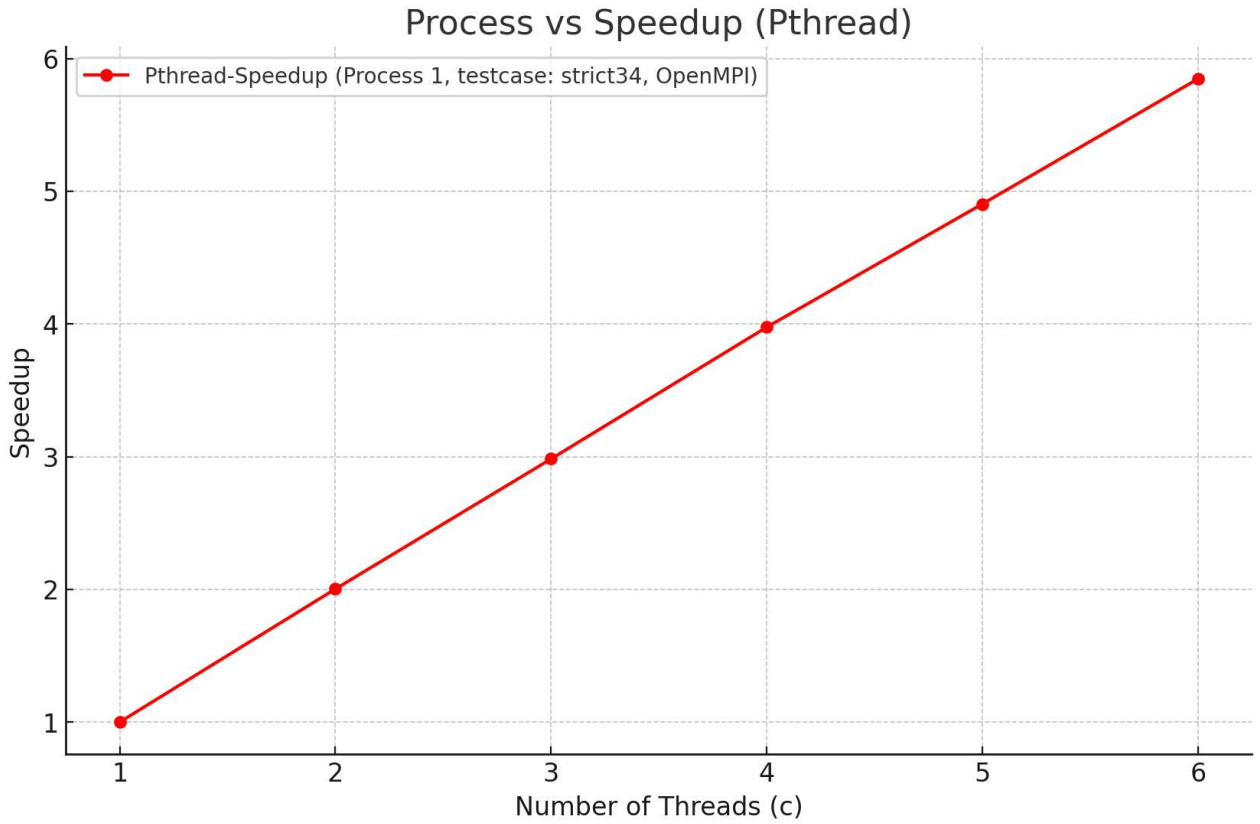
  也是佔 OS Runtime 中的最高比例。

Plots: Scalability & Load Balancing & Profile

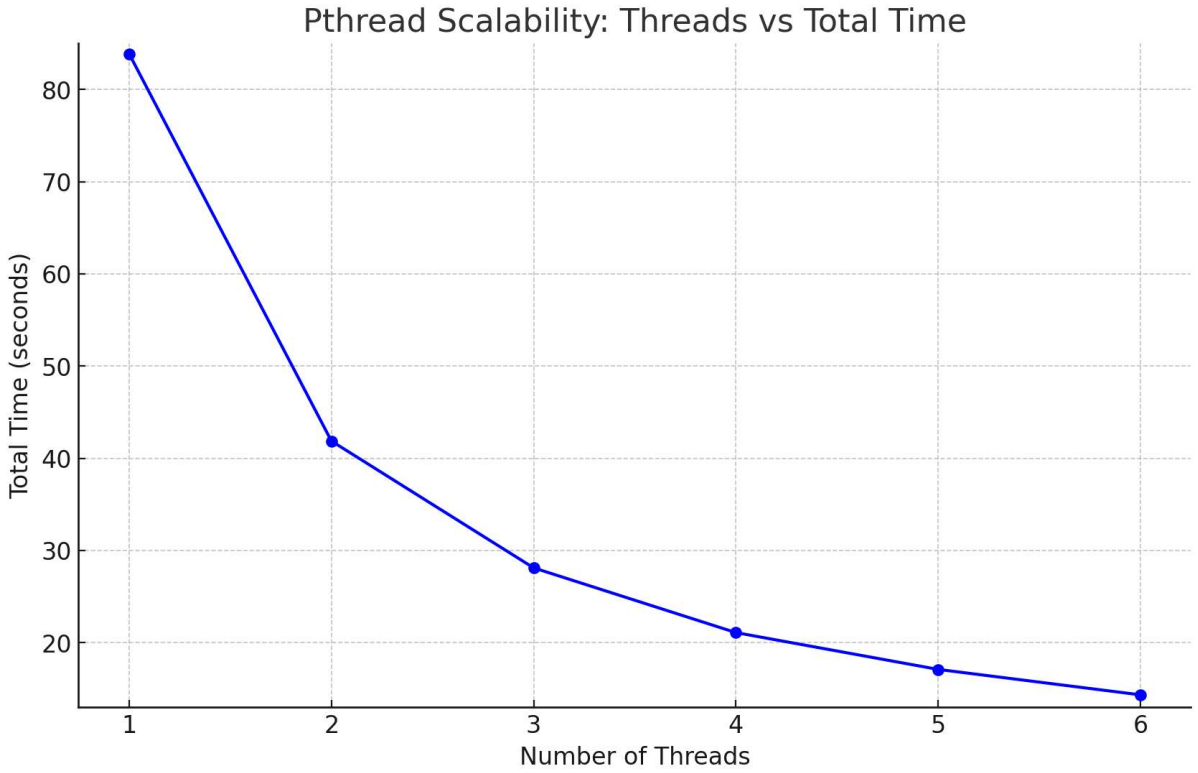
Pthread

Scalability

- 下圖為 Pthread 的 Speedup 情況，可以看到程式隨著 threads 數量越多，表現越好，Speedup 計算方式如下: `sequential time (T1) / multi-thread time`
  Pthread - speedup: threads (Process 1, testcase: strict34, OpenMPI)



- 隨著 thread 數量增加，執行時間越趨減少，故得知此程式有達到 Scalability。



Pthread NVTX 程式測試方法:

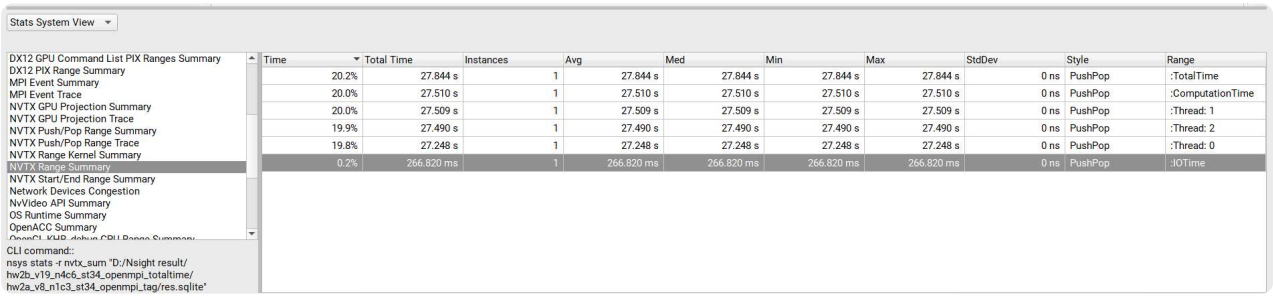
在程式中加入 `#include <nvtx3/nvToolsExt.h>` , `nvtxRangePush();` , `nvtxRangePop();`

- **Thread Time**:

```
void* mandelbrot(void* arg) {
    int thread_id = data->thread_id;
    iters = data->iters;
    double left = data->left;
    double right = data->right;
    double lower = data->lower;
    double upper = data->upper;
    width = data->width;
    height = data->height;
    image = data->image;
    int num_threads = data->num_threads;
    string str;
    stringstream ss;
    ss << "Thread: " << thread_id;
    str = ss.str();
    nvtxRangePush(str.c_str());

    for (int j = thread_id; j < height; j += num_threads) {
        double y0 = j * ((upper - lower) / height) + lower;
        __m512d x0_vec = _mm512_set1_pd(left), x_offset = _mm512_set1_pd((right - left) / width);
        int i = 0;
        for (; i < width - 7; i += 8)
        {
```

```
        for (; i < width; i++)
        {
            double x0 = left + i * ((right - left) / width);
            double x = 0.0;
            double y = 0.0;
            double length_squared = 0.0;
            int repeats = 0;
            while (length_squared < 4.0 && repeats < iters)
            {
                double temp = x * x - y * y + x0;
                y = 2.0 * x * y + y0;
                x = temp;
                length_squared = x * x + y * y;
                repeats++;
            }
            image[j * width + i] = repeats;
        }
    }
    nvtxRangePop(); // Thread
    pthread_exit(NULL);
}
```

如上圖程式碼所示，這樣在跑 Nsight 時，可以在 `Stats System View` 中點選 `NVTX Range Summary` ，即可看到每條 Thread 的執行時間了。



Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Style	Range
20.2%	27.844 s	1	27.844 s	27.844 s	27.844 s	27.844 s	0 ns	PushPop	:TotalTime
20.0%	27.510 s	1	27.510 s	27.510 s	27.510 s	27.510 s	0 ns	PushPop	:ComputationTime
20.0%	27.509 s	1	27.509 s	27.509 s	27.509 s	27.509 s	0 ns	PushPop	:Thread: 1
19.9%	27.490 s	1	27.490 s	27.490 s	27.490 s	27.490 s	0 ns	PushPop	:Thread: 2
19.8%	27.248 s	1	27.248 s	27.248 s	27.248 s	27.248 s	0 ns	PushPop	:Thread: 0
0.2%	266.820 ms	1	266.820 ms	266.820 ms	266.820 ms	266.820 ms	0 ns	PushPop	:IOTime

接著寫 `script` 把需要的資料提取出來輸出成 csv 檔案。

`hw2a_runNsight.sh`：

```
nsys stats -r nvtx_sum "$folderName/res.nsys-rep" --format csv --timeunit sec -o "$folderName/hw2a_v8_result.csv"
```

hw2a_runProcess.sh：

```
sample > $ hw2a_runProcess.sh
#!/bin/bash
for j in {1..6}; do
    ./hw2a_runNsight.sh -n1 -c$j -f "nsight-result/hw2a_v8_n1c${j}_st34_openmpi_tag" hw2a "out.png" 10000 \
    -0.5506164691618783 -0.5506164628264113 0.6273445437118131 0.6273445403522527 7680 4320
done
```

- **Computation Time**:

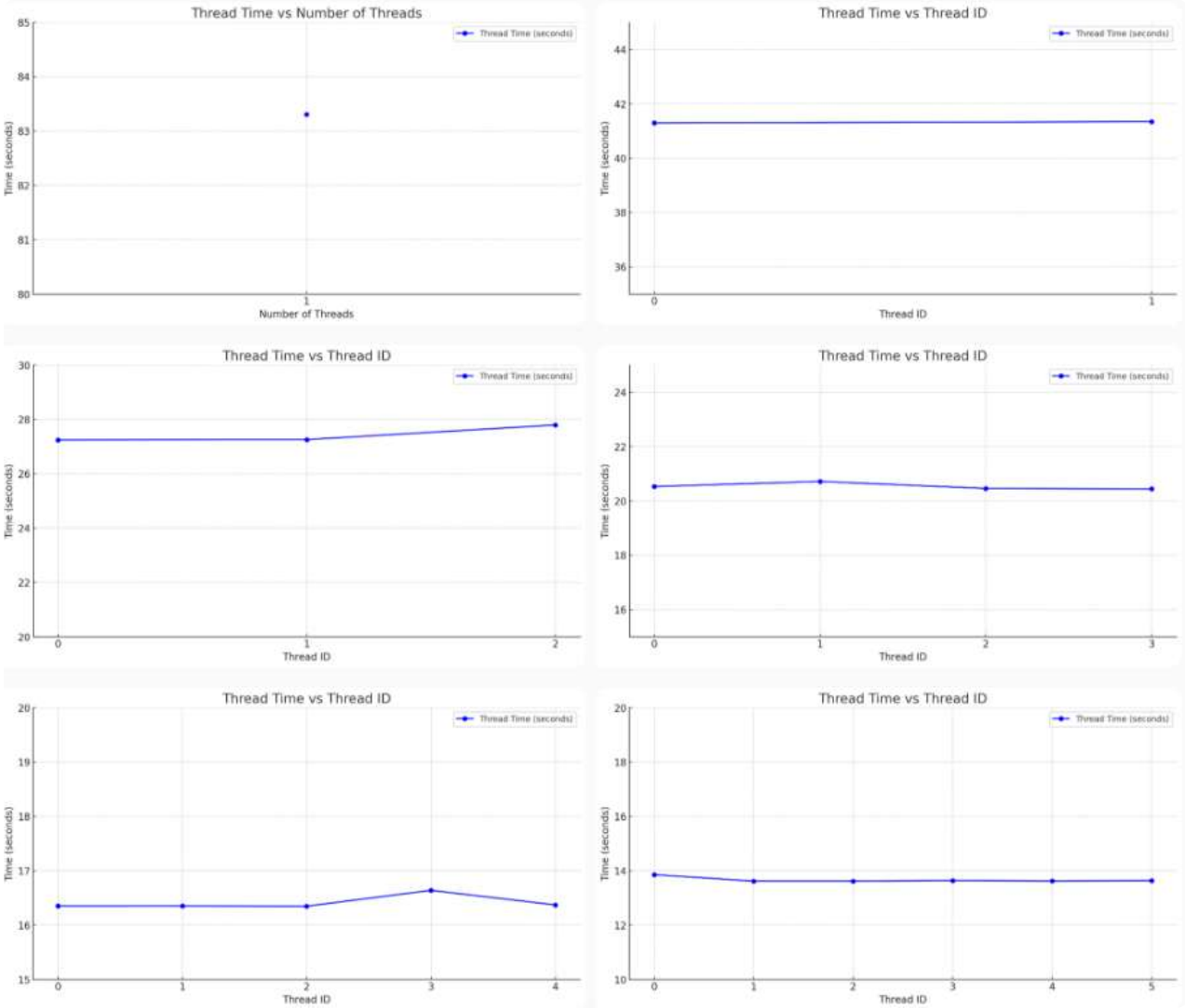
```
nvtxRangePush("ComputationTime");
/* create threads for mandelbrot set calculation */
for (int t = 0; t < num_threads; ++t) {
    thread_data[t].thread_id = t;
    thread_data[t].iters = iters;
    thread_data[t].left = left;
    thread_data[t].right = right;
    thread_data[t].lower = lower;
    thread_data[t].upper = upper;
    thread_data[t].width = width;
    thread_data[t].height = height;
    thread_data[t].image = image;
    thread_data[t].num_threads = num_threads;
    pthread_create(&threads[t], NULL, mandelbrot, (void*)&thread_data[t]);

}

/* join threads */
for (int t = 0; t < num_threads; ++t) {
    pthread_join(threads[t], NULL);
}
nvtxRangePop(); // ComputationTime
```

- **I/O Time**:

```
nvtxRangePush("IOTime");
write_png();
nvtxRangePop();// IOTime
```

Load Balancing

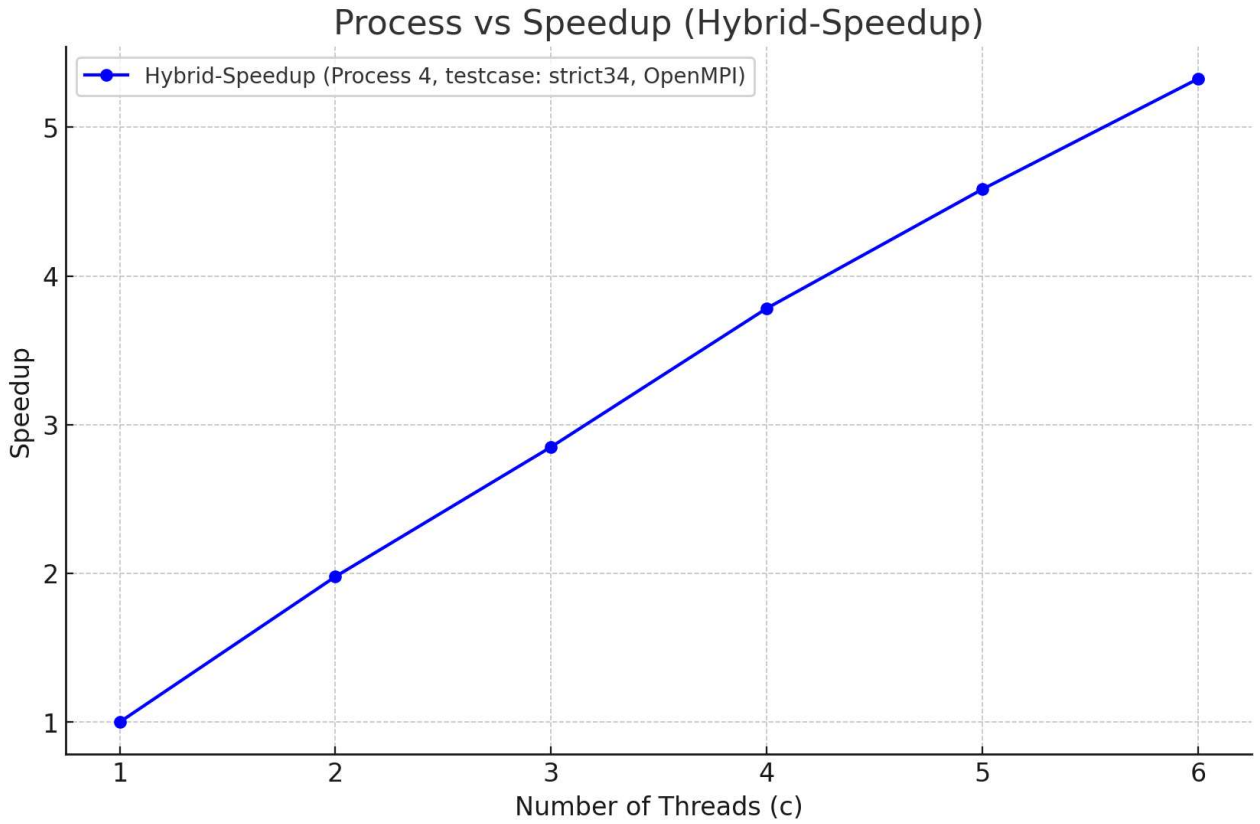
- 以下折線圖為 process = 1, threads = 1~6 的執行時間分析的折線圖，呈現的方式是每個 thread 的執行時間都差不多，故可以看出有達到 load balance。



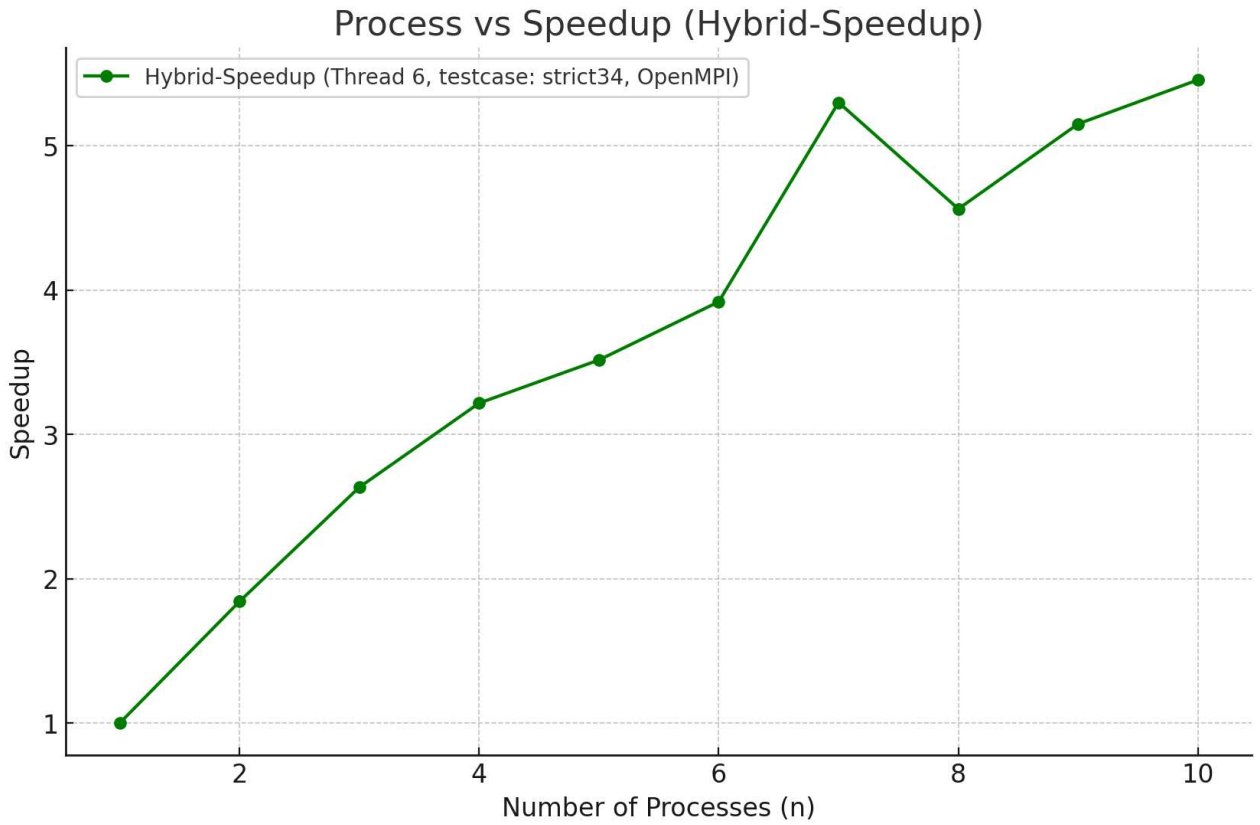
Hybrid: MPI + OpenMP

Scalability

- Speedup 折線圖是最好可以展現出程式是否符合 Scalability 的圖表，我的 speedup 算法是 `sequential time (T1) / multi-process or multi-thread time`
  Hybrid-Speedup: threads (Process 4, testcase: strict34, OpenMPI)

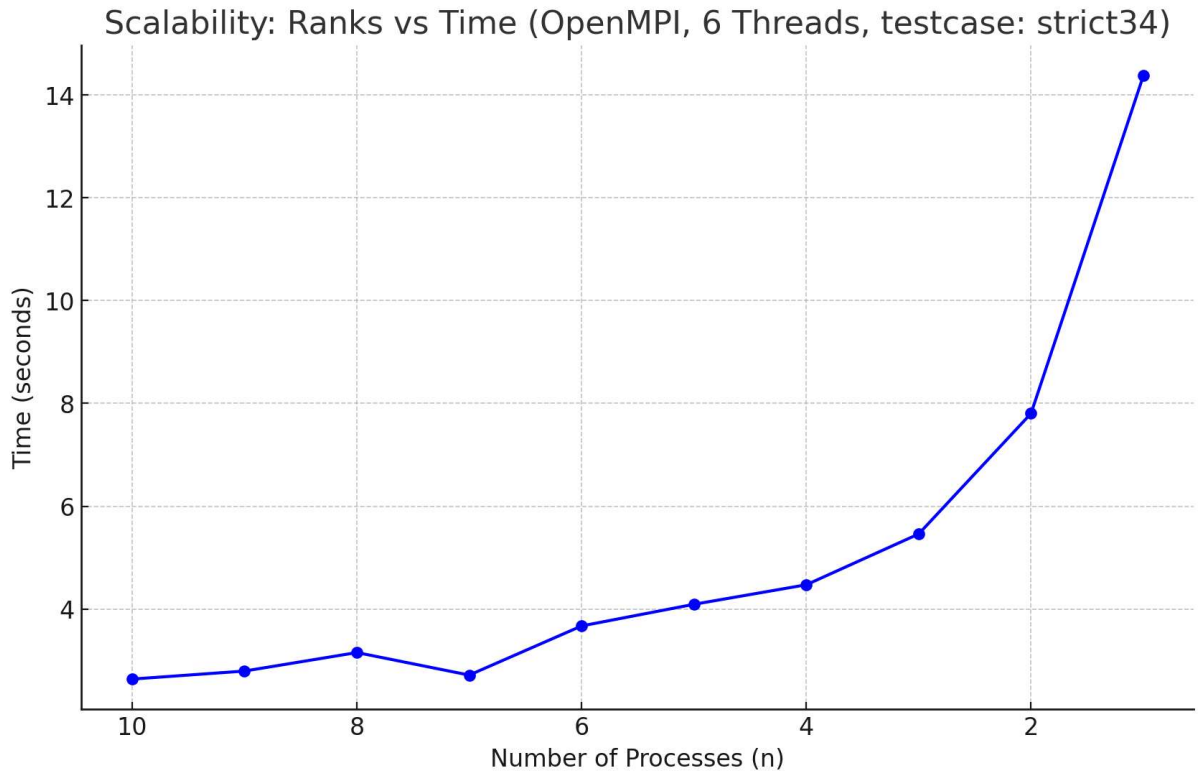


Hybrid-Speedup: ranks (thread 6, testcase: strict34, OpenMPI)



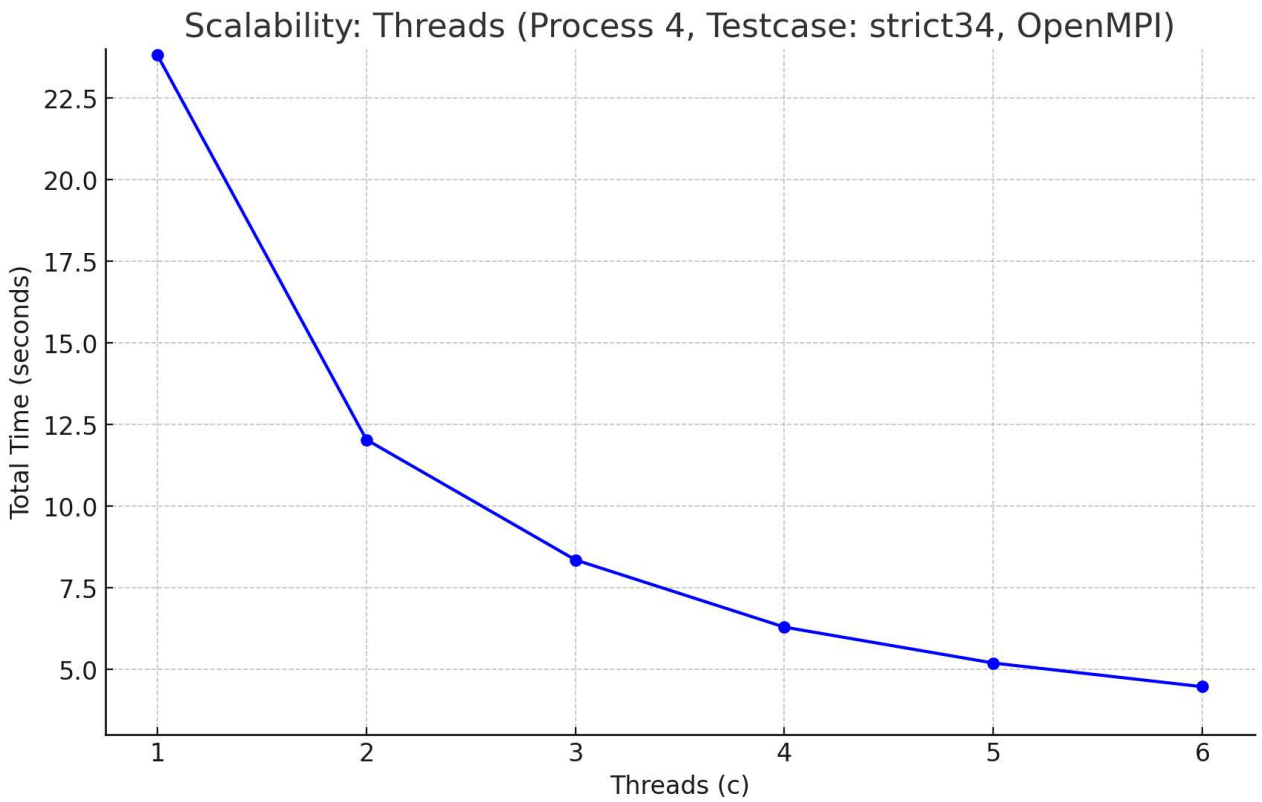
- 由以上 Speedup 折線圖可以看出程式隨著 Thread/Process 開越多的狀況下，效能成線性成長。

- Process 多開一些是有助於提升 Scalability 的，程式整體執行效能也比較好:



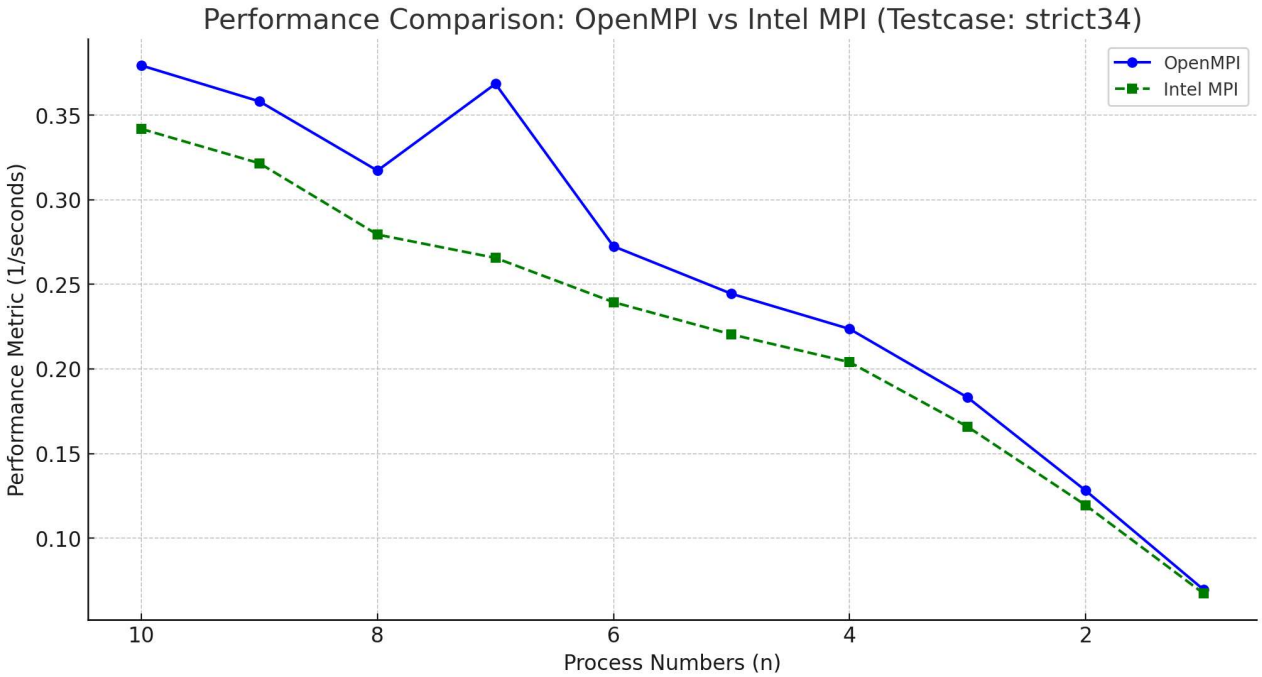
此折線圖為 Thread 相同 (c = 6)，但不同 Process 數量 (1-10) 下程式整體的執行時間比較。

- Thread 多開一些也有助於提升 Scalability 的，程式整體執行效能也比較好:



此折線圖為 Process 相同 (n = 4)，但不同 Threads 數量 (1-6) 下程式整體的執行時間比較。

- Performance Comparison: Openmpi vs Intel MPI (different process numbers, same thread numbers, testcase: strict34.txt)，下圖比較兩個不同 module 的 performance，y 軸為 `1 / second`：



Hybrid NVTX 程式測試方法:

一樣在程式中加入 `#include <nvtx3/nvToolsExt.h>`, `nvtxRangePush();`, `nvtxRangePop();`

- **Process Time (Rank Time)**:

Start:

```
/* allocate memory for the local image slice */
string str;
stringstream ss;
ss << "Rank: " << rank;
str = ss.str();
nvtxRangePush(str.c_str());
```

End:

```
/* gather all slices to rank 0 using MPI_Gatherv */
int *all_h = (int *)malloc(size * sizeof(int));
int *displs = (int *)malloc(size * sizeof(int));
image = NULL;
for (int i = 0; i < size; ++i)
{
    int new_local_height = (i == size - 1) ? SumModMul : heightMulSize;
    all_h[i] = new_local_height * width;
    displs[i] = i * heightMulSize * width;
}
if (rank == 0)
{
    image = (int *)malloc(width * height * sizeof(int));
    // assert(image);
}
nvtxRangePop();
```

- **Thread Time**:

Start:

```
#pragma omp parallel for schedule(dynamic, 1)
// #pragma omp parallel
for (int j = thread_start; j < thread_end; ++j)
{
    // char str[80];
    // sprintf(str, "");

    string str;
    stringstream ss;
    ss << "Thread: " << omp_get_thread_num();
    str = ss.str();
    nvtxRangePush(str.c_str());

    // nvtxRangePush("");
    double y0 = (j + other_local_height) * ((upper - lower) / height) + lower;
    __m512d x0_vec = _mm512_set1_pd(left), x_offset = _mm512_set1_pd((right - left) / width);
    int i = 0;
    for (; i < width - 7; i += 8)
    {

        // _mm512_setzero_pd
        __m512d x0 = _mm512_set_pd(
            i+7,
            i+6,
            i+5,
            i+4,
            i+3,
            i+2,
            i+1,
            i
```

End:

```
for (; i < width; i++)
{
    double x0 = left + i * ((right - left) / width), x = 0.0, y = 0.0, length_squared = 0.0;
    int repeats = 0;
    // bool mask = 1;
    // double xx=0, yy=0;
    while (length_squared < 4.0 && repeats < iters)
    {
        double temp = x*x - y*y + x0;
        // xy = x*y;
        y = 2.0 * x*y + y0;
        x = temp;
        length_squared = x*x + y*y;
        repeats++;
    }
    local_image[j * width + i] = repeats;
}
nvtxRangePop(); // Thread
```

- **Computation Time**:

Start:

```
nvtxRangePush("ComputationTime");
// nvtxRangePush("Rank");

int heightModSize = height % size;
int heightMulSize = height / size;
int SumModMul = heightMulSize + heightModSize;
int local_height = (rank == size - 1) ? SumModMul : heightMulSize;
int other_local_height = rank * heightMulSize;
int num_threads = omp_get_num_threads();
int thread_rank = omp_get_thread_num();
int end = local_height;
int *local_image = (int *)malloc(width * local_height * sizeof(int));
// assert(local_image);
int endNum_threads = end / num_threads;
unsigned long long thread_start = thread_rank * endNum_threads;
unsigned long long thread_end = (thread_rank == num_threads - 1) ? end : (thread_start + endNum_threads);
```

End:

```
    for (; i < width; i++)
    {
        double x0 = left + i * ((right - left) / width), x = 0.0, y = 0.0, length_squared = 0.0;
        int repeats = 0;
        // bool mask = 1;
        // double xx=0, yy=0;        (double)(4.0)
        while (length_squared < 4.0 && repeats < iters)
        {
            double temp = x*x - y*y + x0;
            // xy = x*y;
            y = 2.0 * x*y + y0;
            x = temp;
            length_squared = x*x + y*y;
            repeats++;
        }
        local_image[j * width + i] = repeats;
    }
    nvtxRangePop(); // Thread
}
nvtxRangePop(); // ComputationTime
```

- **I/O Time**:

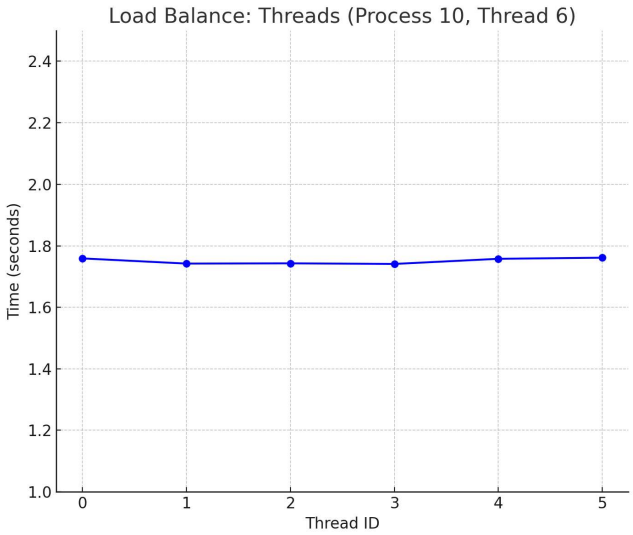
```
// /* draw and cleanup */
if (rank == 0)
{
    nvtxRangePush("IOTime");
    write_png();
    // free(image);
    nvtxRangePop(); // IOTime
}
```

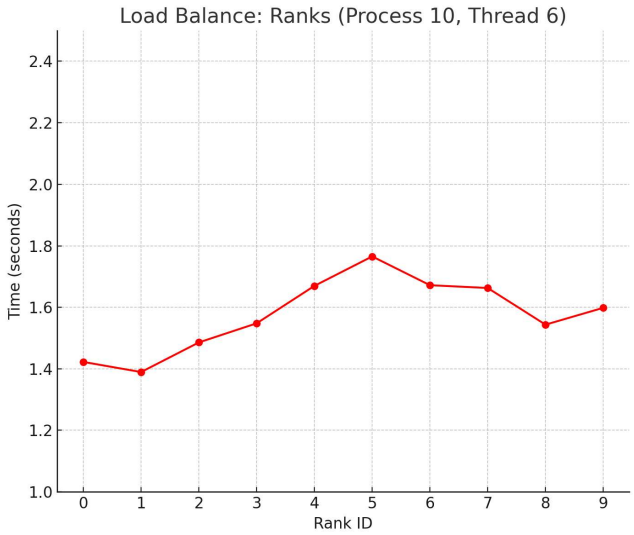
- **Communication Time**:

```
nvtxRangePush("CommunicationTime");
MPI_Gatherv(local_image, local_height * width, MPI_INT, image, all_h, displs, MPI_INT, 0, MPI_COMM_WORLD);
nvtxRangePop(); // CommunicationTime
```

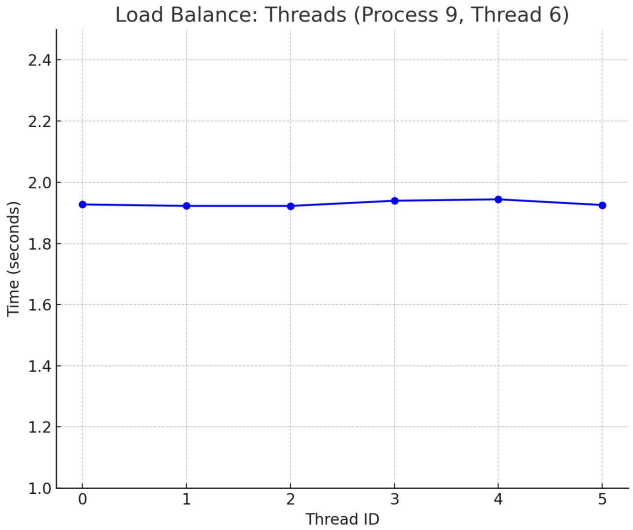
Load Balancing

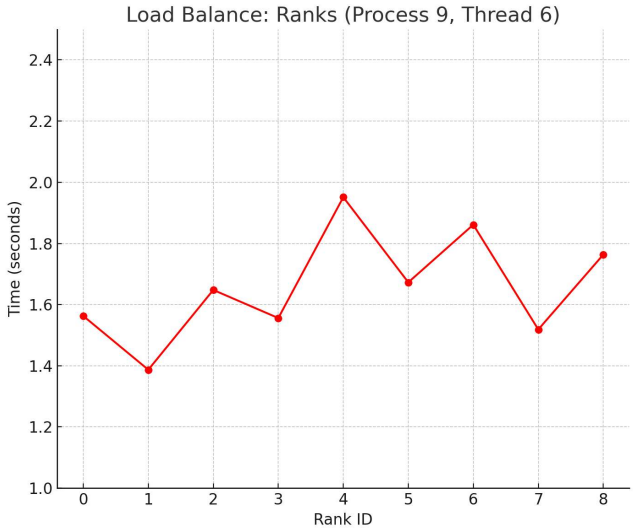
- 不同 Rank 數量以及不同 Thread 數量下，程式的平行化效能各有不同，越能達到 Load Balance 的程式效能越好，而在分配每個 rank 以及 thread 中有多少 input data 就會是重點。
- 以下為不同 Process 數量下 (Threads 數量相同都為 6)，不同 Thread 的執行時間折線圖。
- Process: 10, Thread: 6



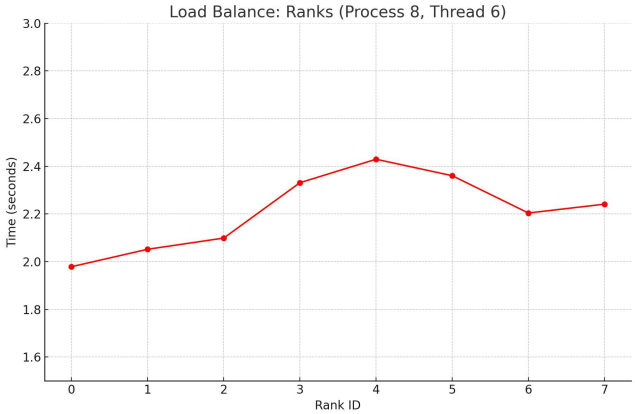


- Process: 9, Thread: 6

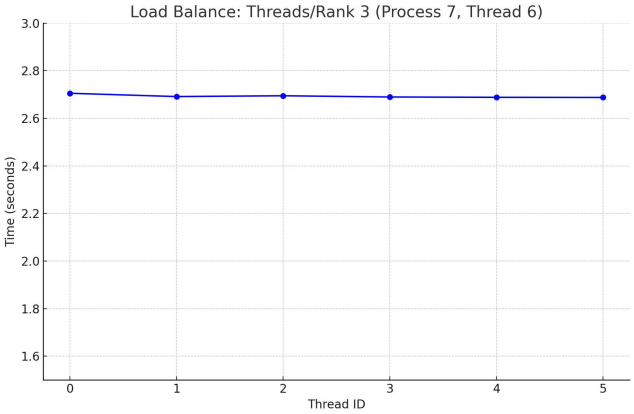


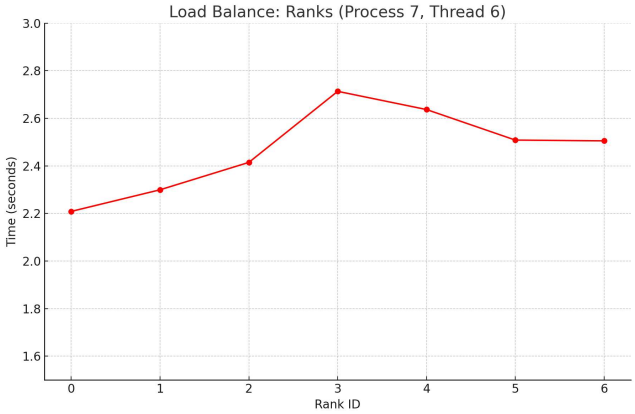


- Process: 8, Thread: 6

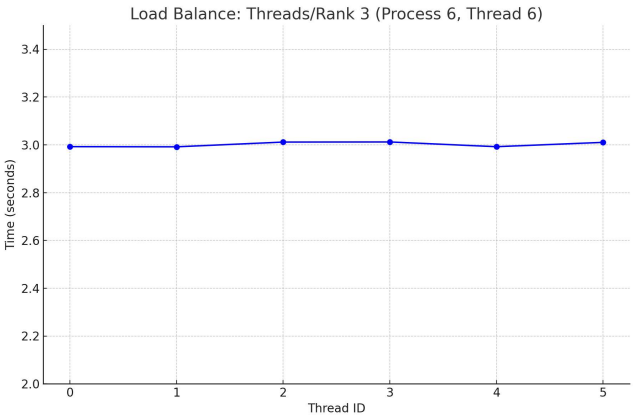


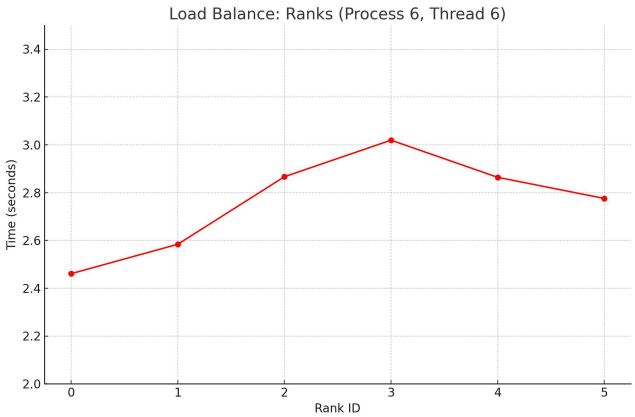
- Process: 7, Thread: 6



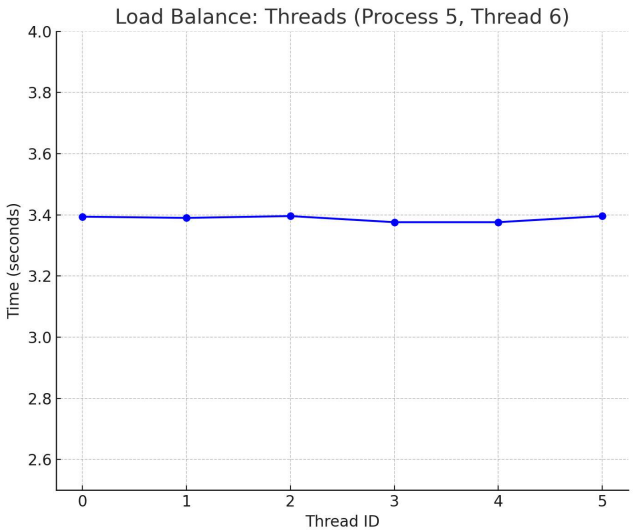


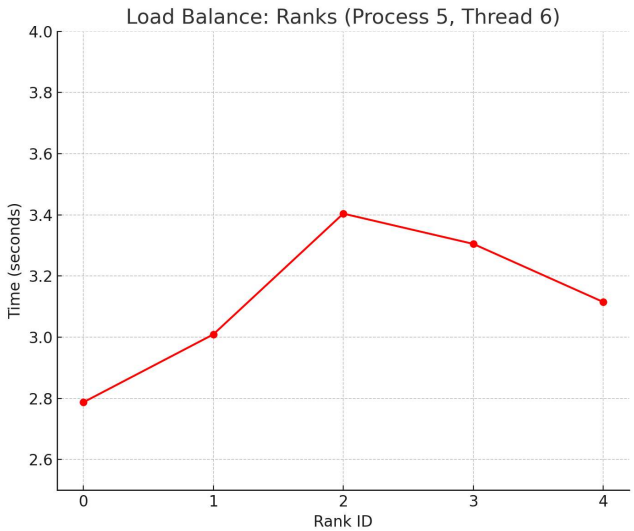
- Process: 6, Thread:6



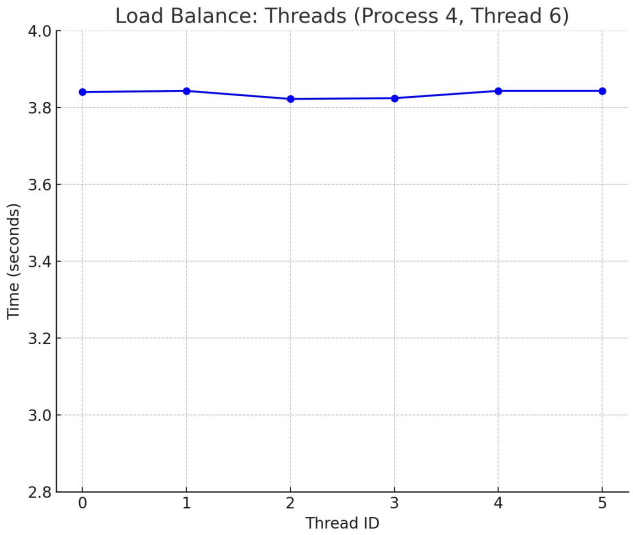


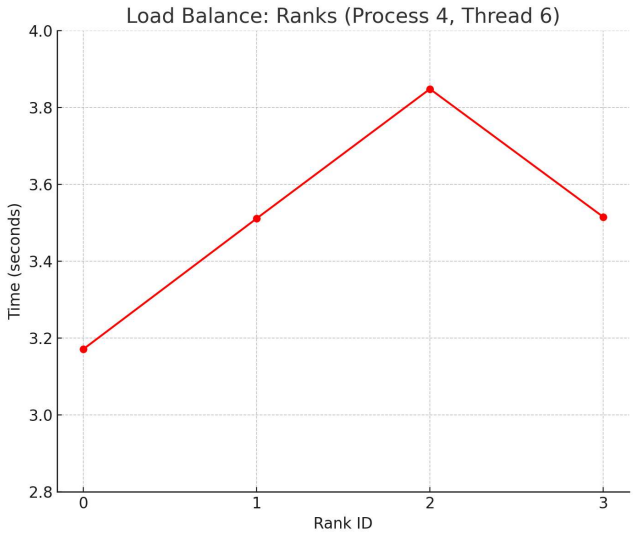
- Process: 5, Thread: 6



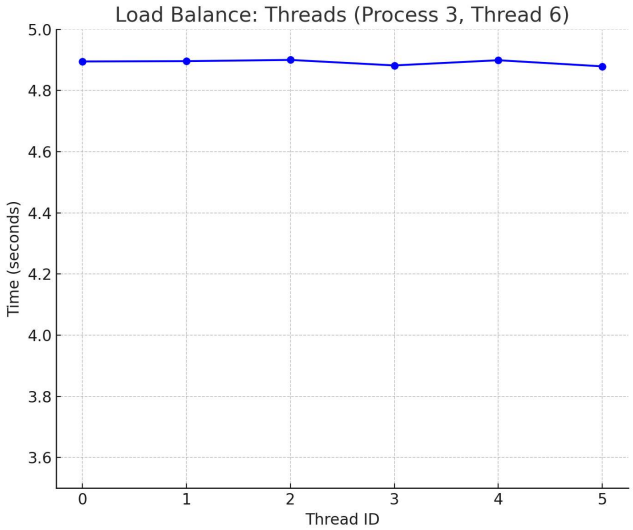


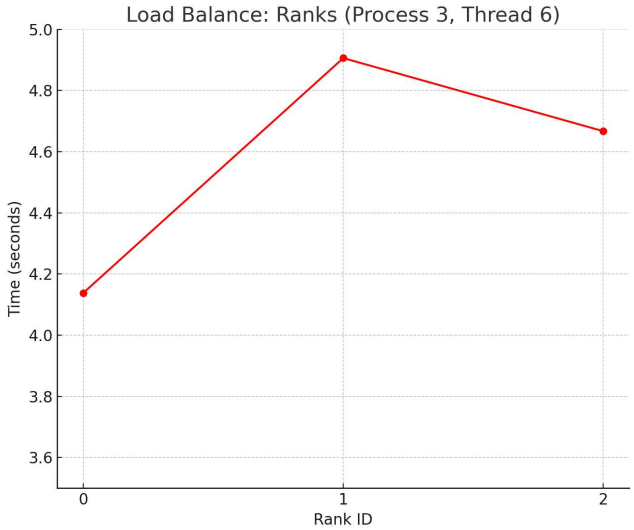
- Process: 4, Thread: 6



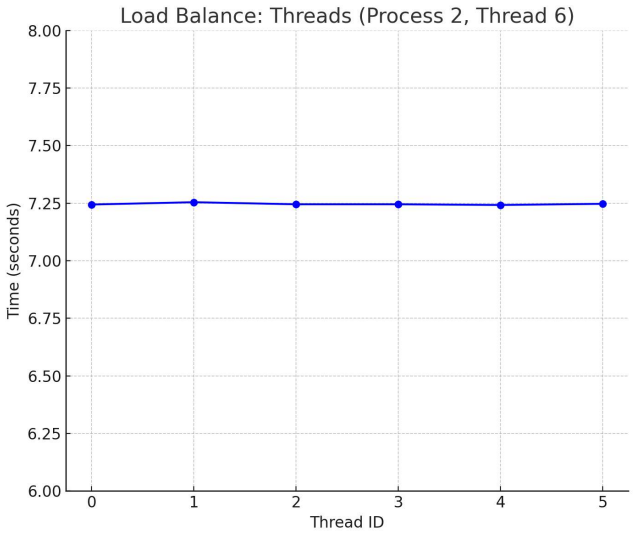


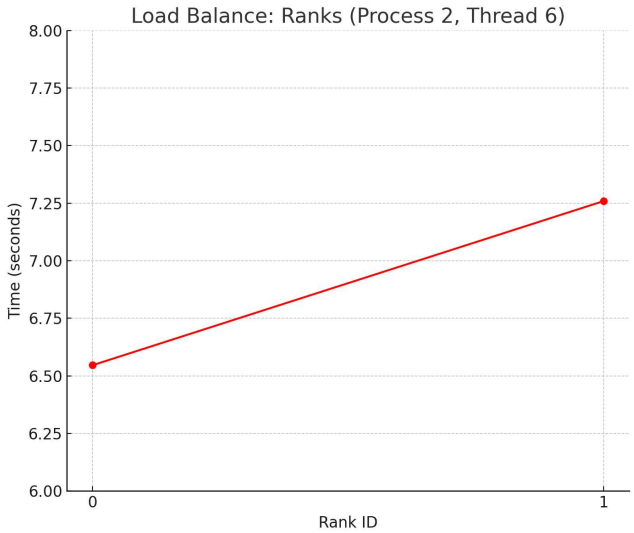
- Process: 3, Thread: 6



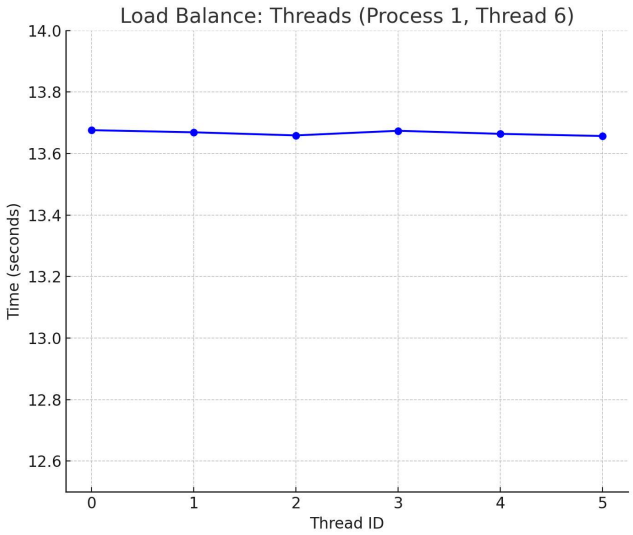


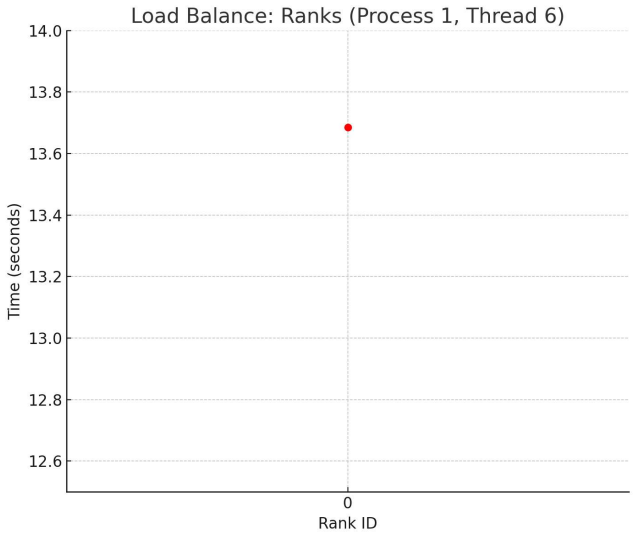
- Process: 2, Thread: 6





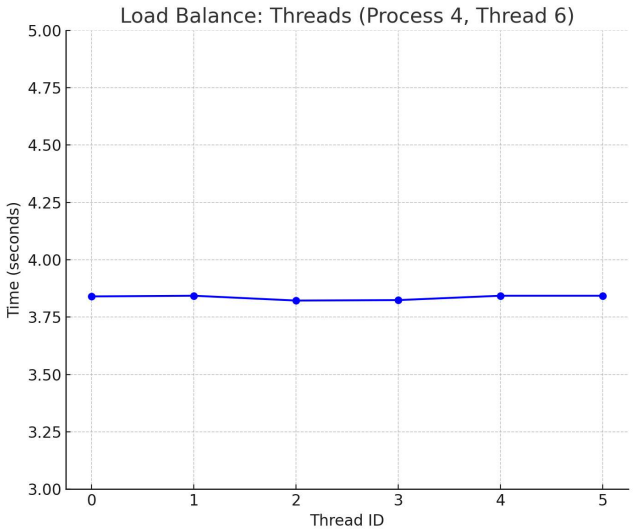
- Process: 1, Thread: 6

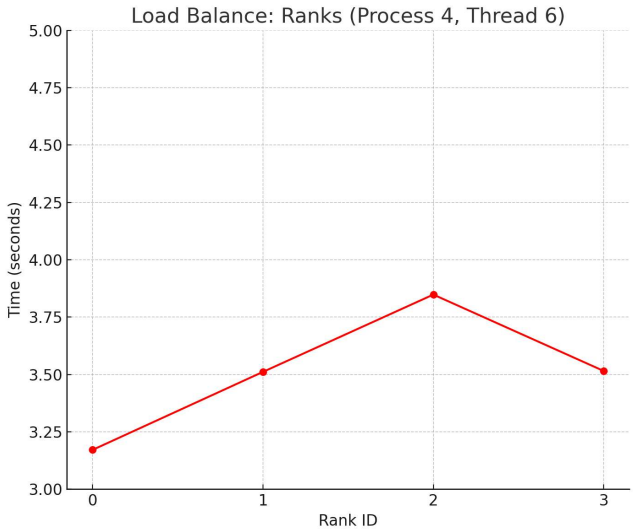




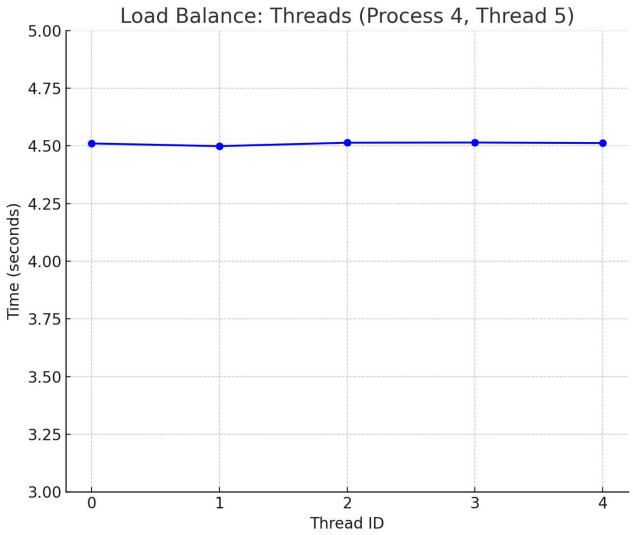
可以發現在各個 Process 中每個 Thread 執行時間都差不多，故有達到 Load Balance。

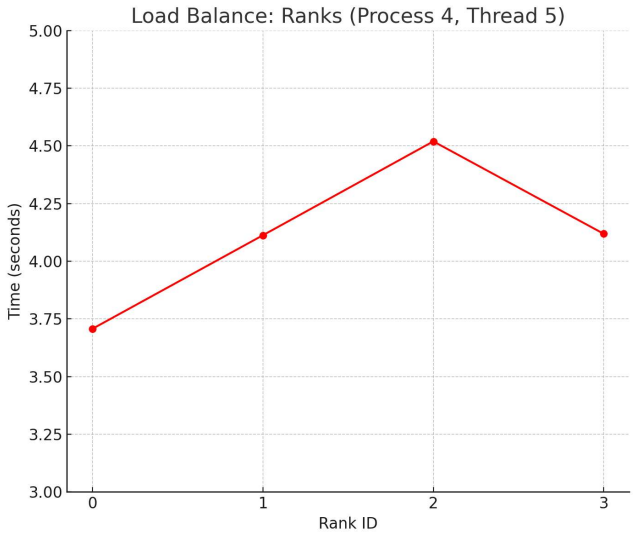
- 以下為不同 Thread 數量下 (Process 數量固定都為 4)，不同 Thread 的執行時間折線圖。
- Process: 4, Thread: 6



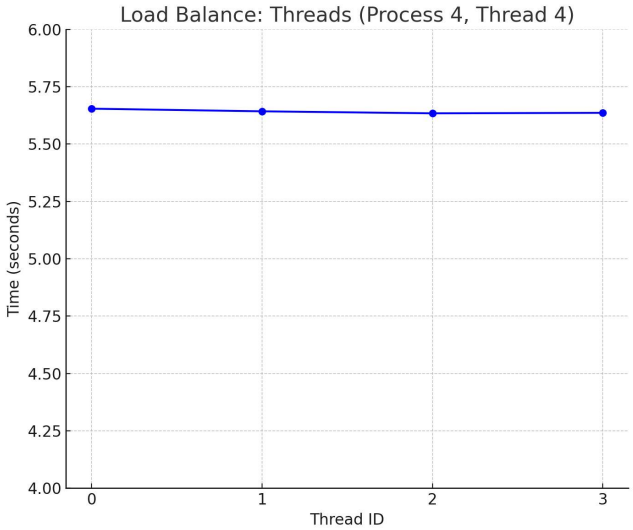


- Process: 4, Thread: 5

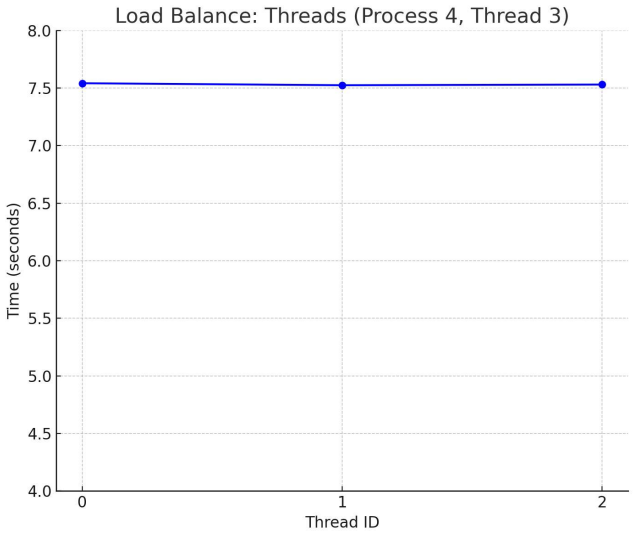


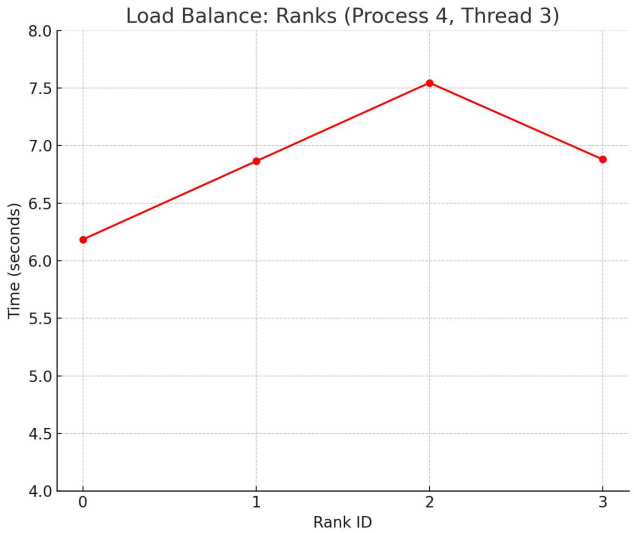


- Process: 4, Thread: 4

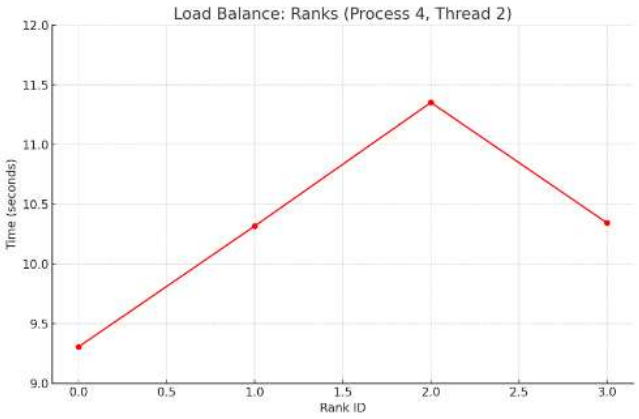


- Process: 4, Thread: 3

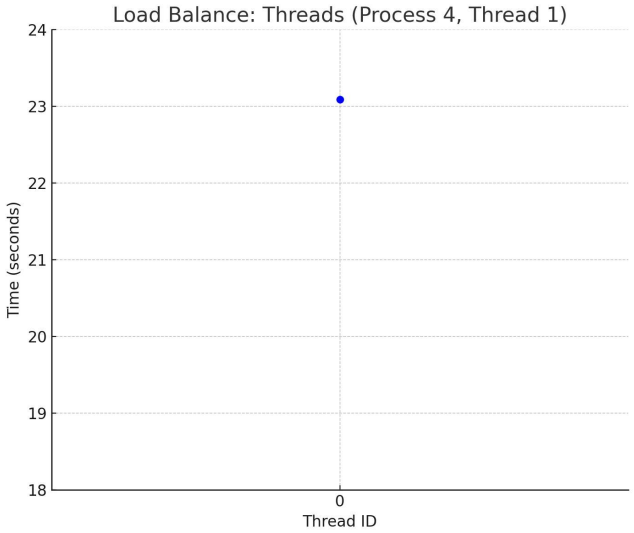


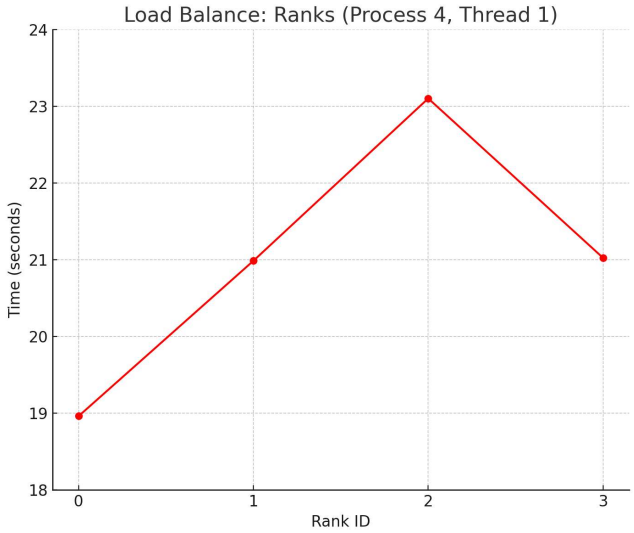


- Process: 4, Thread: 2



- Process: 4, Thread: 1

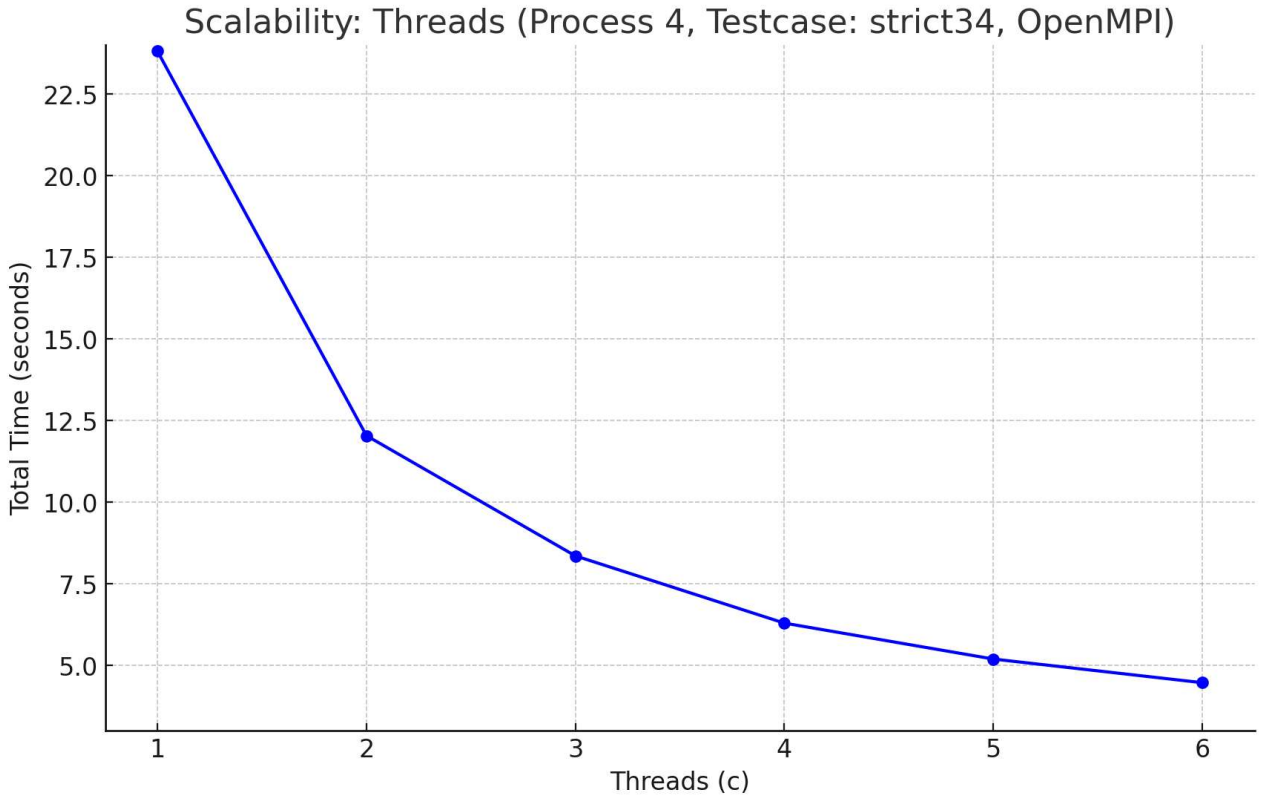




> 可以由折線圖發現每個 Thread 依舊執行時間差不多，但隨著 Thread 數量越少，Rank 之間執行時間差距越大:

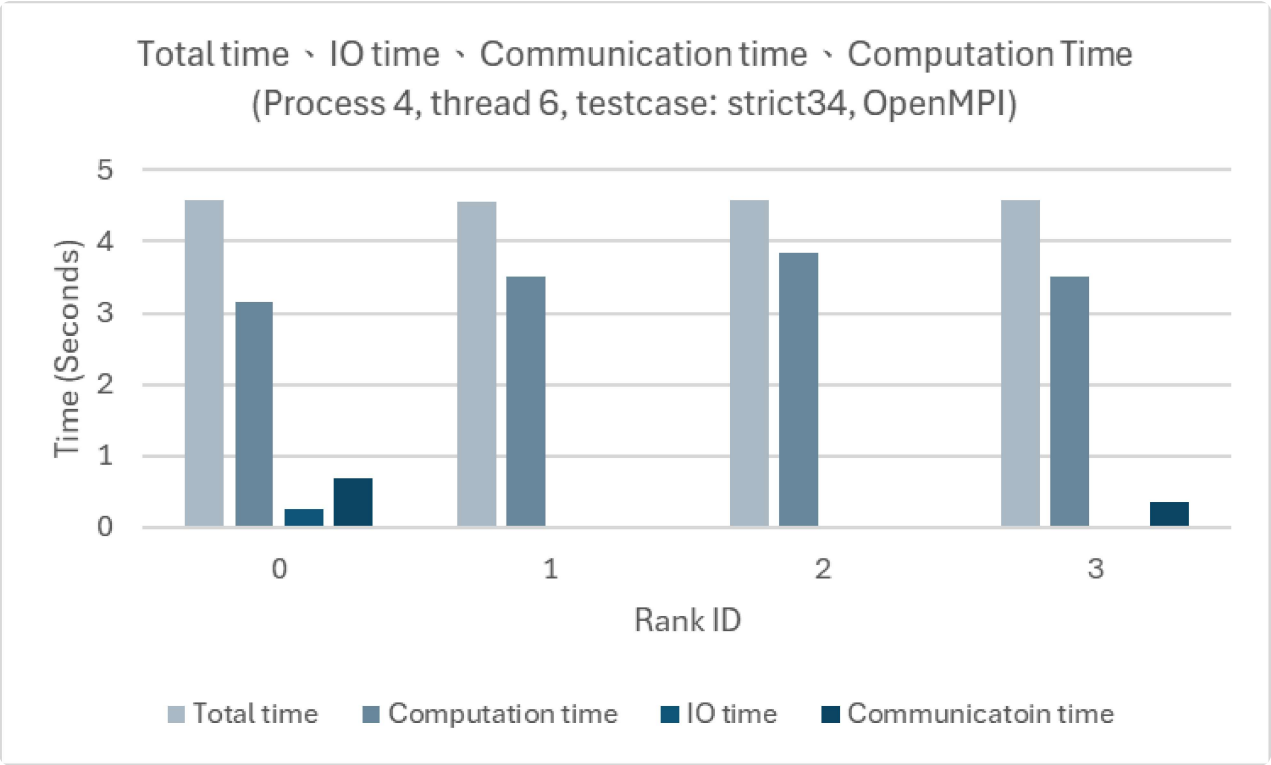
Process/Thread	**4/6**	**4/1**
(Max - min) rank execution time(seconds)	0.677	4.1395

- 由 Process: 4, Thread: 1 中的數據，找出 Rank 中執行時間的 Max value (at rank 2: 23.1011 seconds)，減去 min value (at rank 0: 18.9616 seconds)，相減得出 4.1395 秒。
- Process: 4, Thread: 6 中的數據，找出 Rank 中執行時間的 Max value (at rank 2: 3.848
  seconds)，減去 min value (at rank 0: 3.171 seconds)，相減得出 0.677 秒。
- 由此可知，Thread 多開一些是有助於 load balance 的，程式整體執行效能也比較好(Scalability):



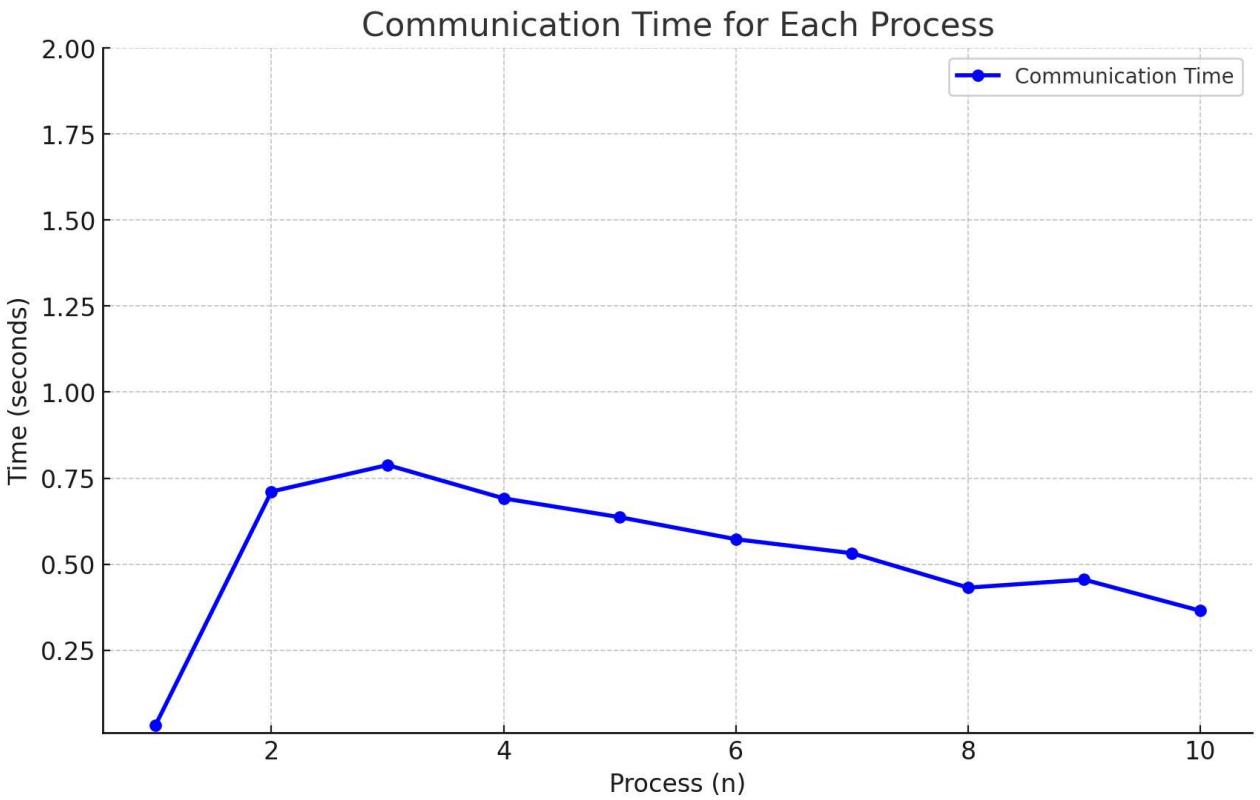
此折線圖為 Process 相同 (n = 4)，但不同 Threads 數量下程式整體的執行時間比較。

- Total time、IO time、Communication time、Computation Time (Process 4, thread 6, testcase: strict34, OpenMPI)

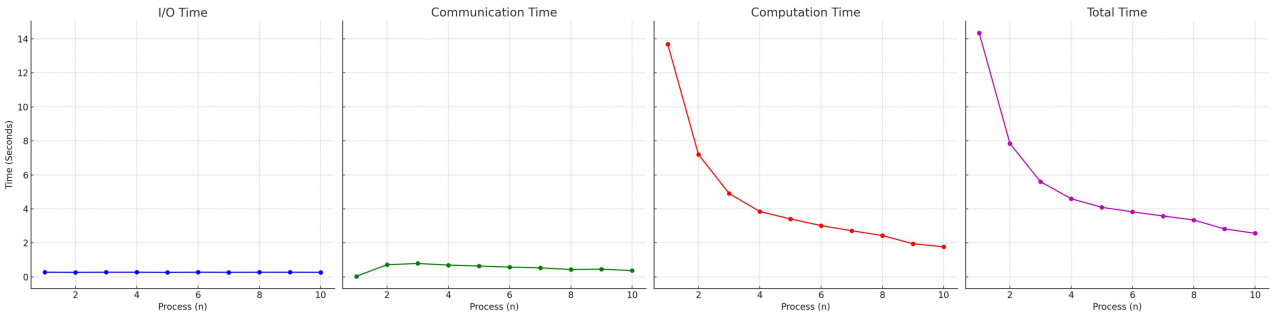


由此柱狀圖可以發現 rank 0 因為需要負責 call write_png() 故 IO time 都集中在 rank 0。

- Commucation Time 隨著 Process 的增加而增加:
  (Threads 固定在 6 個)
- 因為當 process = 1 時，因為只有一個 process 故不需要 process 與 process 之間的溝通，所以其時間接近 0。

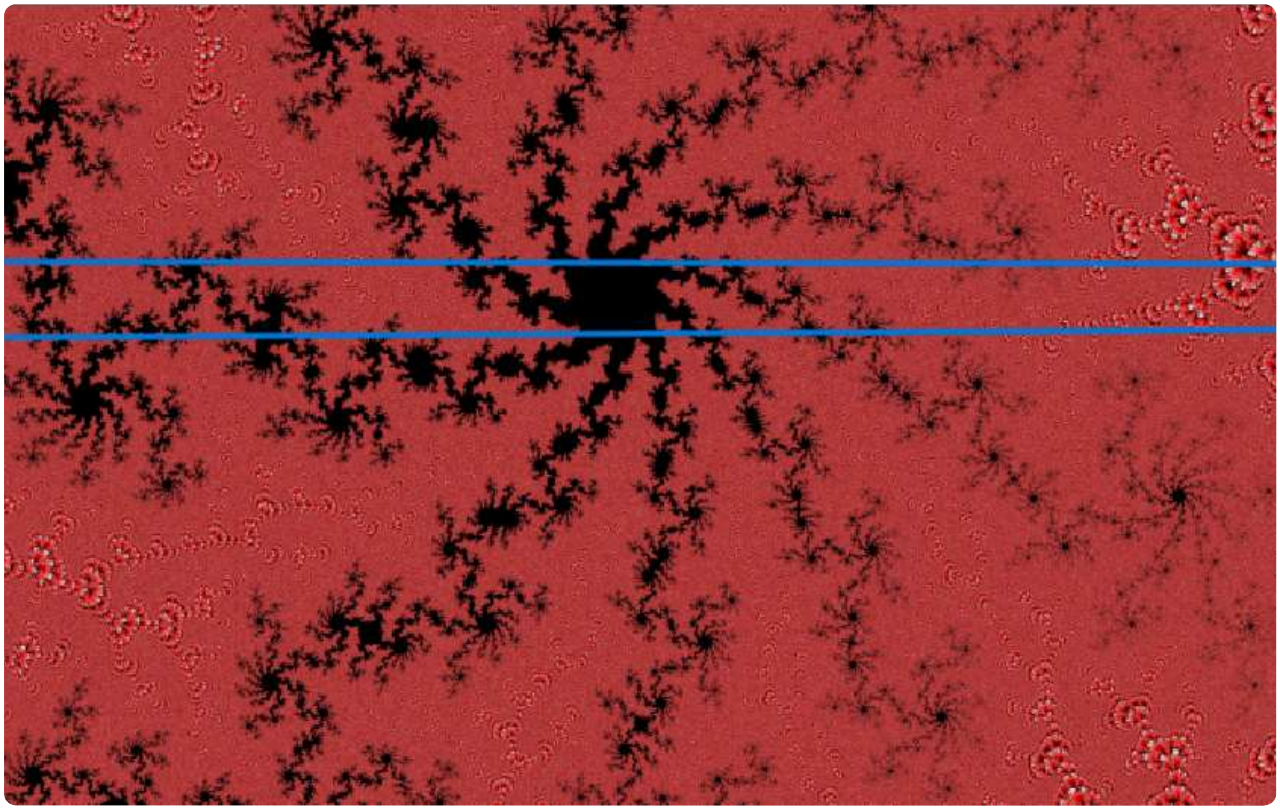


- 下圖為 Process 由 1 增加到 10 的 Communication time 折線圖:



Experience & Conclusion

- 實驗的部分選擇的是最耗時的 testcase: strict34.txt:



- 其中此圖片的 input 資訊為:

```
iter: 10000
x0: -0.5506164691618783
x1: -0.5506164628264113
y0: 0.6273445437118131
y1: 0.6273445403522527
width: 7680
height: 4320
```

- 可以發現在我畫藍色直線區間內，黑色的部分最多，這也解釋了為甚麼後面的數據畫出來的折線圖中，rank 2 or rank 3 都是花費最多時間的 rank。
- 從 computation time 折線圖中也可以發現 rank 2 or rank 3 最花時間。

Optimization

SIMD

- `#include <immintrin.h>`
- 將程式中的乘法、加法都做了向量化優化 (Vectorization)，向量化之所以能優化程式是因為它是讓多個數字一起做運算，也是一種 parallel computing。我的程式使用的是 AVX-512 `__m512d` ，這個 512 是指 512 bits，一個 double 是 64 bits，故 `__m512d` 一次可以做 8 個 double 的運算，這也是為甚麼使用了 vectorization 後能節省這麼多執行時間。

- 加上它支援很多不同的運算，例如: `_mm512_add_pd` 、 `_mm512_mul_pd` 、 `_mm512_sub_pd` ,...,etc.
- 在向量化程式中，我還使用到了 `mask` ，這個我用來判斷程式是否需要進入迴圈中，原程式需要去判斷 `repeats >= iters` 就跳出迴圈 terminate，使用 `__mmask8 mask` 可以讓它一次判斷 8 個數字，並將狀態存到 `mask` 中: 當 `repeats < iters` mask 設為 1，若 `mask != 0` 則繼續進入 while loop 做計算，
- 為了節省 while loop 的計算成本，將原本的

```
while (repeats < iters && length_squared < 4) {
```

這兩個條件一併在迴圈中用一條 `&` bit operation 去做判斷

```
mask = _mm256_cmplt_epi32_mask(repeats, iter_vec) & _mm512_cmplt_pd_mask(length_squared, four);
```

存到 mask 當中。

> Before & After

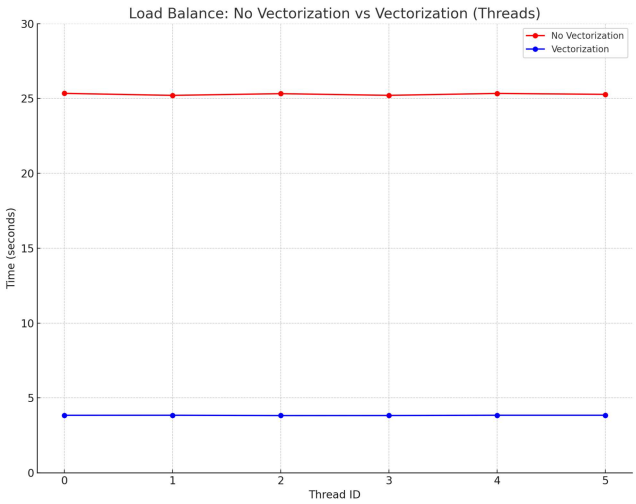
Hybrid

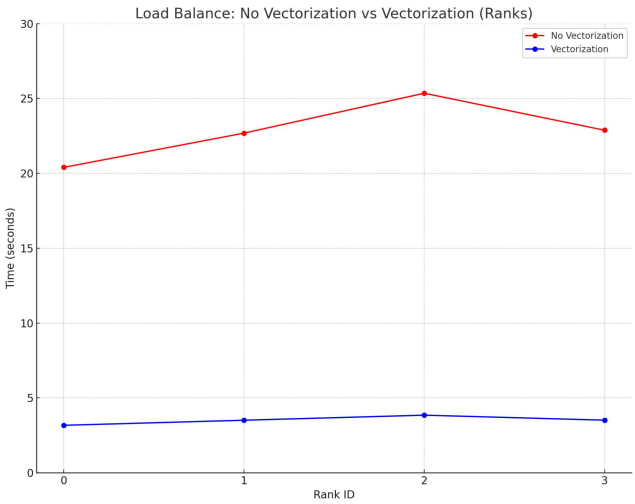
- No Vectorization & Vectorization: Computation Time Optimization

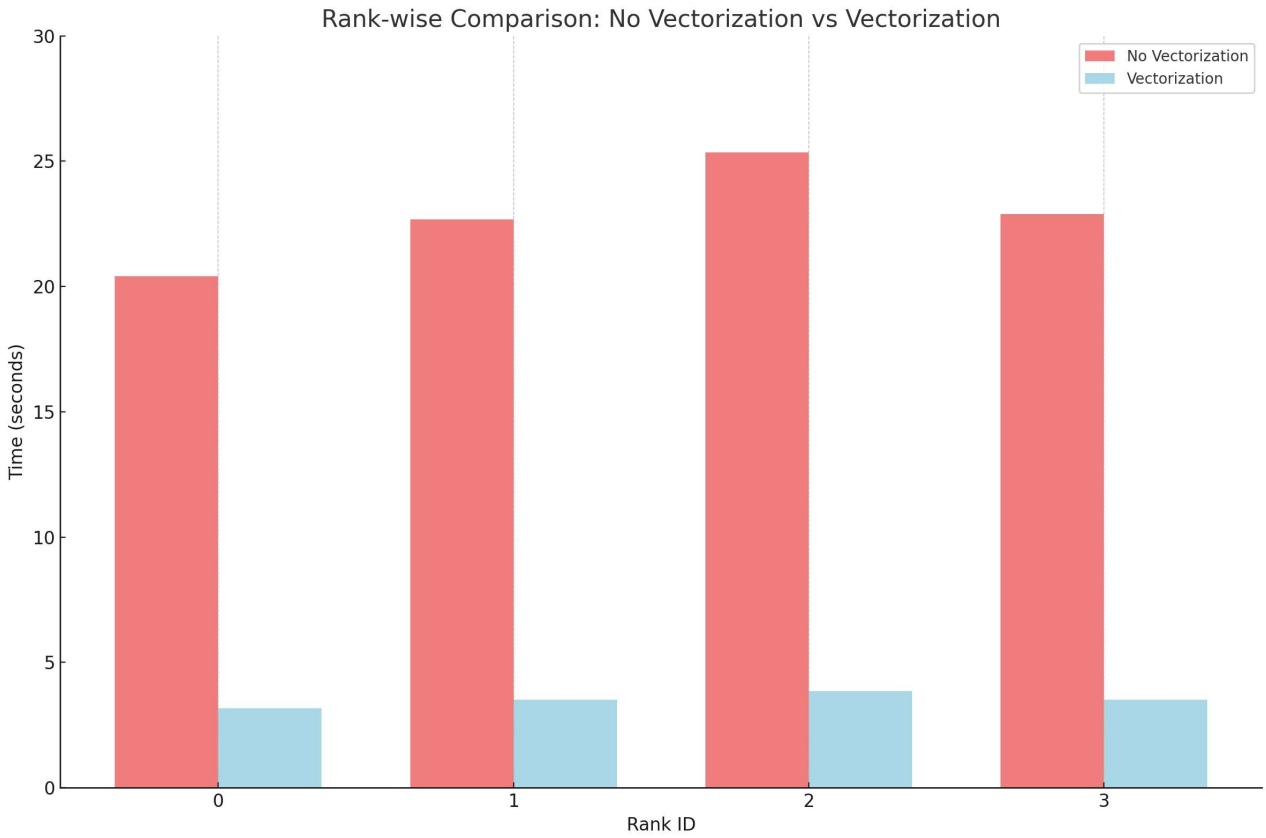
```
#pragma omp parallel for schedule(dynamic, 1)
for (int j = thread_start; j < thread_end; ++j)
{
    double y0 = (j + other_local_height) * ((upper - lower) / height) + lower;
    __m512d x0_vec = _mm512_set1_pd(left), x_offset = _mm512_set1_pd((right - left) / width);
    int i = 0;
    for (; i < width - 7; i += 8)
    {
        // _mm512_setzero_pd
        __m512d x0 = _mm512_set_pd(
            i+7,
            i+6,
            i+5,
            i+4,
            i+3,
            i+2,
            i+1,
            i
        );
        x0 = _mm512_add_pd(x0_vec, _mm512_mul_pd(x0, x_offset));
        __m512d x = _mm512_setzero_pd(), y = _mm512_setzero_pd(), length_squared = _mm512_setzero_pd(), two = _mm512_set1_pd(2.0), four = _mm512_set1_pd(4.0);
        __m256i one = _mm256_set1_epi32(1.0), iter_vec = _mm256_set1_epi32(iters), repeats = _mm256_setzero_si256(); // int: 32 bits * 8 = 256
        __mmask8 mask = 0xFF;
        __m512d xx = _mm512_mul_pd(x, x), yy = _mm512_mul_pd(y, y);
        // _mm256_cmplt_pd_mask(repeats, iters) // repeats < iters: set to 1
        while (mask)
        {
            // __m512d temp = _mm512_mul_pd(xySum, xySum); // x * x - y * y
            __m512d temp = _mm512_add_pd(_mm512_sub_pd(xx, yy), x0);  // x * x - y * y
            y = _mm512_add_pd(_mm512_mul_pd(two, _mm512_mul_pd(x, y)), _mm512_set1_pd(y0)); // 2 * x * y + y0
            x = temp;
            xx = _mm512_mul_pd(x, x), yy = _mm512_mul_pd(y, y);
            length_squared = _mm512_add_pd(xx, yy); // x * x + y * y
            repeats = _mm256_mask_add_epi32(repeats, mask, repeats, one); // repeats ++;
            // mask = _mm256_cmplt_epi32_mask(_mm256_mask_add_epi32(repeats, mask, repeats, one), iter_vec) & _mm512_cmplt_pd_mask(length_squared, four);
            mask = _mm256_cmplt_epi32_mask(repeats, iter_vec) & _mm512_cmplt_pd_mask(length_squared, four);
        }

        // _mm256_storeu_epi32(&local_image[j * width + i], repeats);
        // Store the iteration counts for the 8 elements
        int temp[8];
        _mm256_store_epi32(temp, repeats);
```

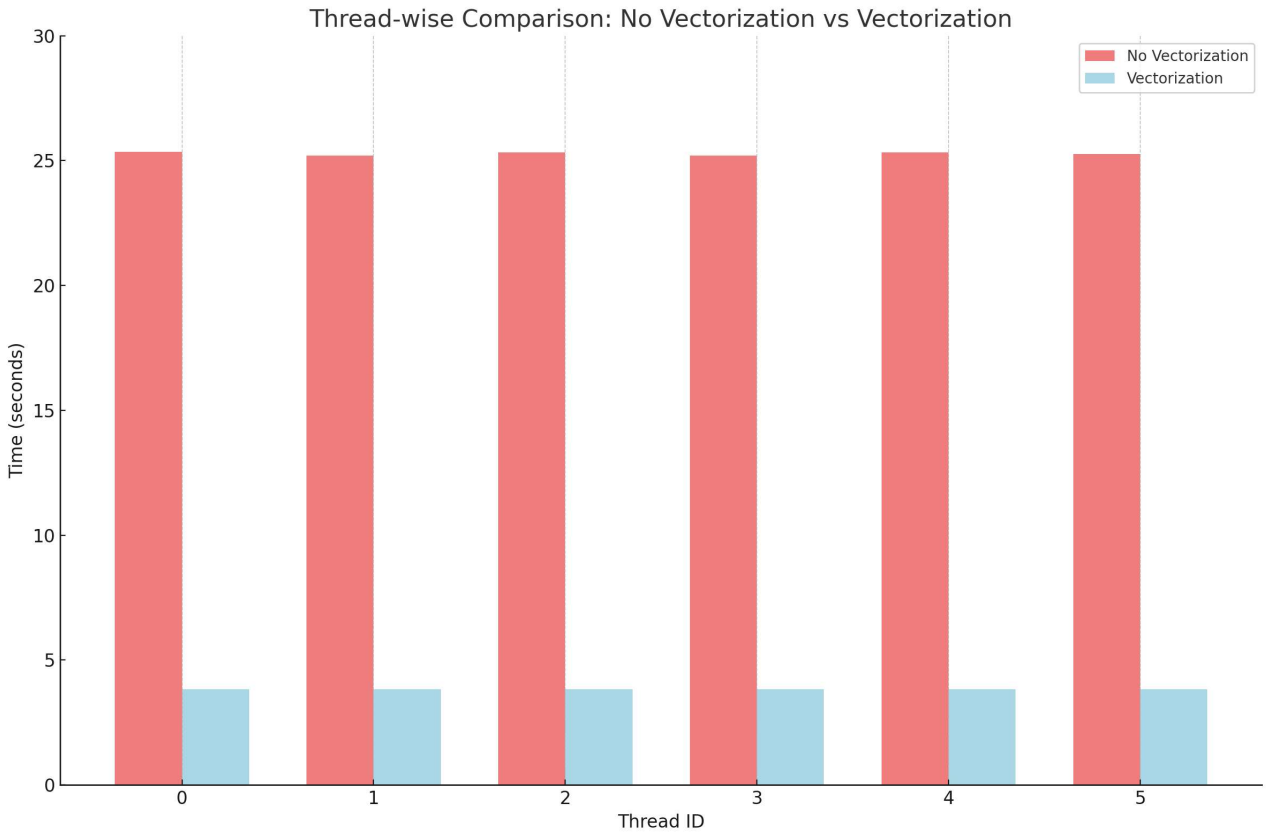
```
    for (; i < width; i++)
    {
        double x0 = left + i * ((right - left) / width), x = 0.0, y = 0.0, length_squared = 0.0;
        int repeats = 0;
        // bool mask = 1;
        // double xx=0, yy=0;
        while (length_squared < 4.0 && repeats < iters)
        {
            double temp = x*x - y*y + x0;
            // xy = x*y;
            y = 2.0 * x*y + y0;
            x = temp;
            length_squared = x*x + y*y;
            repeats++;
        }
        local_image[j * width + i] = repeats;
    }
}
```

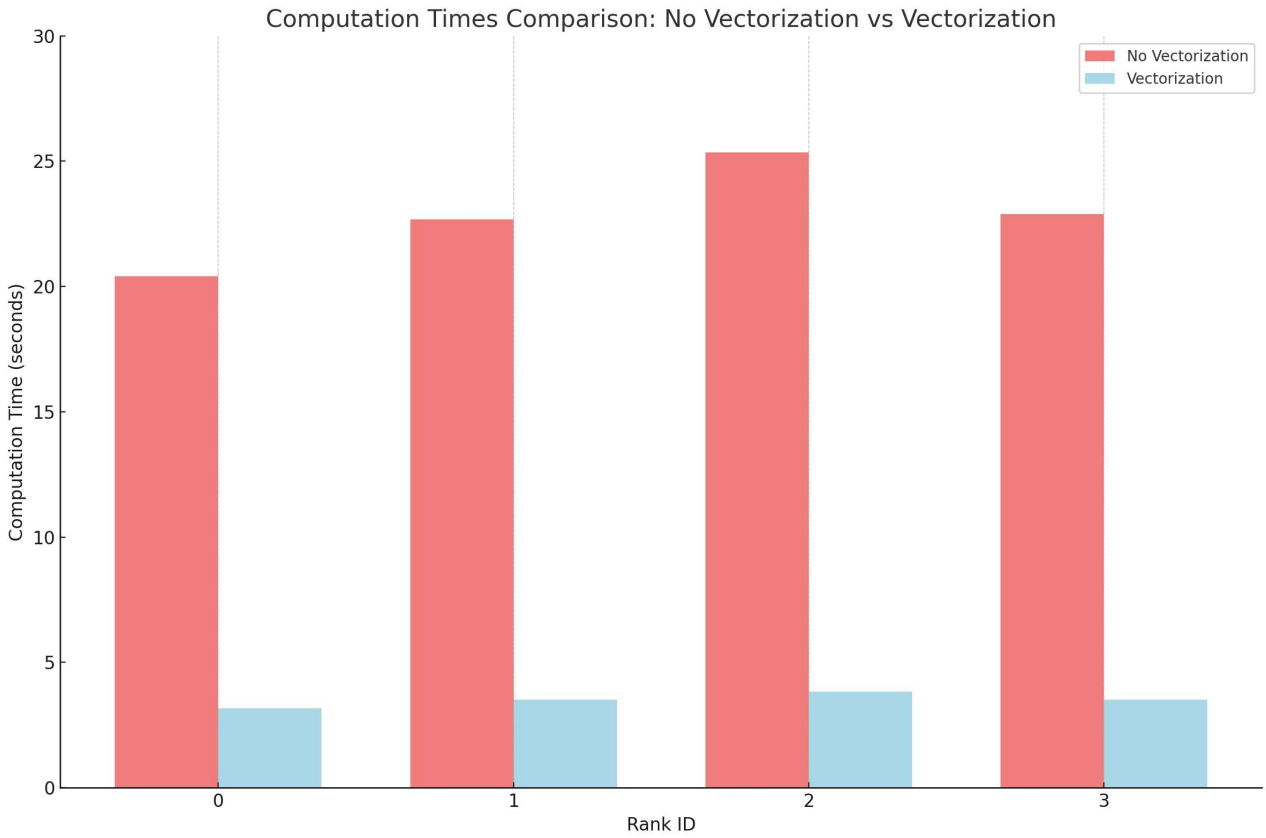
- 由以下圖表可以發現，沒向量化的程式多了非常多執行時間，尤其在 rank 2 時的執行時間尤其多，原因是 rank 2 的 computation time 最多故影響最大。



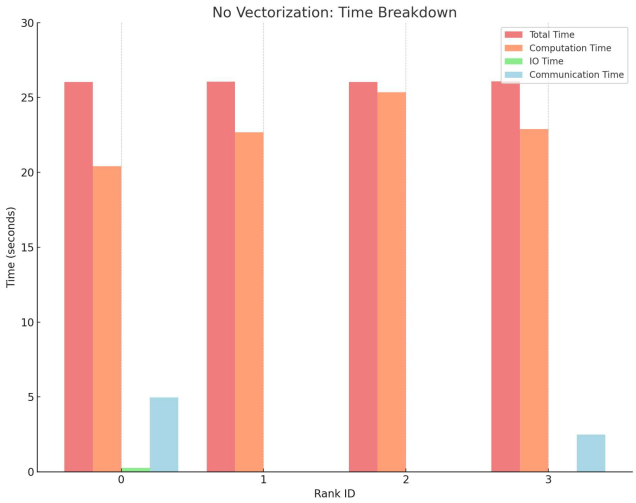


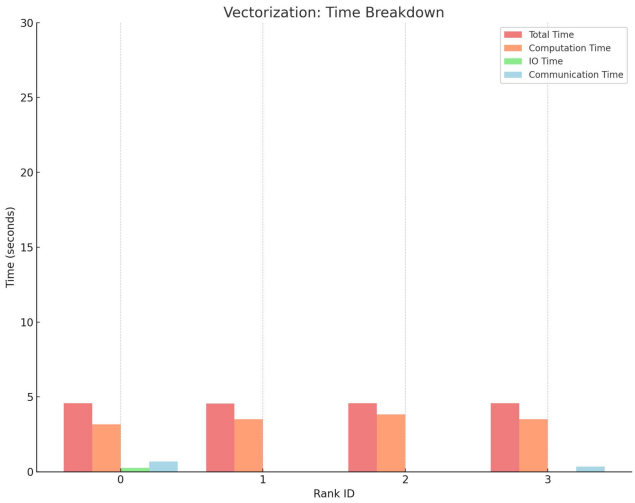

Rank-wise Comparison: No Vectorization vs Vectorization
No Vectorization
Vectorization
Time (seconds)
0
5
10
15
20
25
30
0
1
2
3
Rank ID


Thread-wise Comparison: No Vectorization vs Vectorization
No Vectorization
Vectorization
Time (seconds)
0
5
10
15
20
25
30
0
1
2
3
4
5
Thread ID



下圖為兩者的 Total、Communication、I/O、Computation time 的柱狀圖:





Other Optimization

`write_png()`

- 在 write_png 當中，有發現兩處可以改的參數

1. `png_set_compression_level(png_ptr, 0)`：由 1 改成 0，這個改動讓程式快了好幾秒，壓縮等級改為 0，這就表示不會進行壓縮（僅儲存原始數據），因此省略了壓縮過程。
2. `png_set_filter(png_ptr, 0, PNG_FILTER_NONE)`：由 `PNG_NO_FILTERS` 改為 `PNG_FILTER_NONE` ，也讓程式快了好幾秒，原因是改為 `PNG_FILTER_NONE` 後明確告

訴 libpng 不要使用任何過濾，這會在某些實作中更有效率。因為在某些情況下，PNG_NO_FILTERS 可能默認會讓程式根據數據來判斷是否需要使用某些過濾方法，使用 `PNG_FILTER_NONE` 避免了這種自動判斷的 overhead，加速寫入過程。

3. 除此之外，我將 `write_png` 中不需要傳入參數做呼叫，原因是傳入參數做 void function 的呼叫也會耗費一些時間，我將 原本需要傳入的部分當作 global variable

```
const char* filename;
int iters, height, width;
int* image;
void write_png()
{
```

。

4. 將 mod + 乘法運算

```
color[1] = color[2] = p % 16 * 16;
```

改為 bit operation:

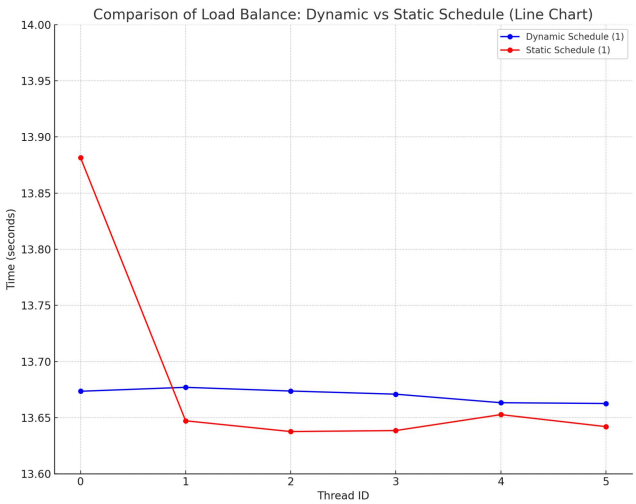
```
color[0] = (p&16) ? 240 :(p & 15) << 4;
color[1] = color[2] = (p&16) ? (p & 15) << 4 : color[1];
```

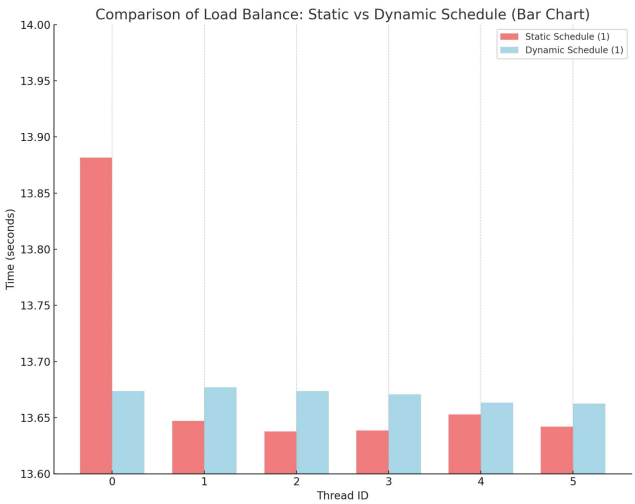
Load Balance

Hybrid

- OpenMP:

> Before & After





- #pragma omp parallel for schedule(dynamic, 1)
  - 有 thread 先做完會先去搶還沒做的工作，Load Balance會比較好

- #pragma omp parallel for schedule(static, 1)
    - 每個 thread 會執行同樣數量的工作，Load Balance 比較差

Pthread & MPI & OpenMP

- Pthread 相對於 MPI 的優勢在於 Pthread 可以透過 `pthread_create` 、 `pthread_exit` 、 `pthread_join` 控制
- OpenMP 與 Pthread 的區別在於編譯方式的不同，OpenMP 需加入編譯器處理指令 `#pragma` ，在 create thread 後的工作會需要 compiler 來完成。Pthread 是一個資料庫，所有的 parallel program 會需要 user 手動去 call `pthread_create()` 、 `pthread_exit()` 、 `pthread_join()` 等等 API 才會創建 thread。
- MPI 則是給予程式能夠進行跨越 Node or Process 之間的一個傳送資料的方法，而 pthread 與 OpenMP 的 thread 與 thread 之間的資料為共享的，故只需傳送指針就能做到資料共享與溝通。
- 由上面實驗結果可知， `hw2a-Pthread` 中無 Communication Time，因為它 default Process 數量為 1。在 `hw2b-Hybrid` 中，Communication Time 來自 `MPI_Gatherv()` ，這一行主要在收集不同 process 之間的資料，故會需要花很多 Communication Time。
- 我認為這份作業最難的部分有兩者:

1. 透過分每個 process 中的 task 量去做到 load balance
2. 去思考如何做向量化

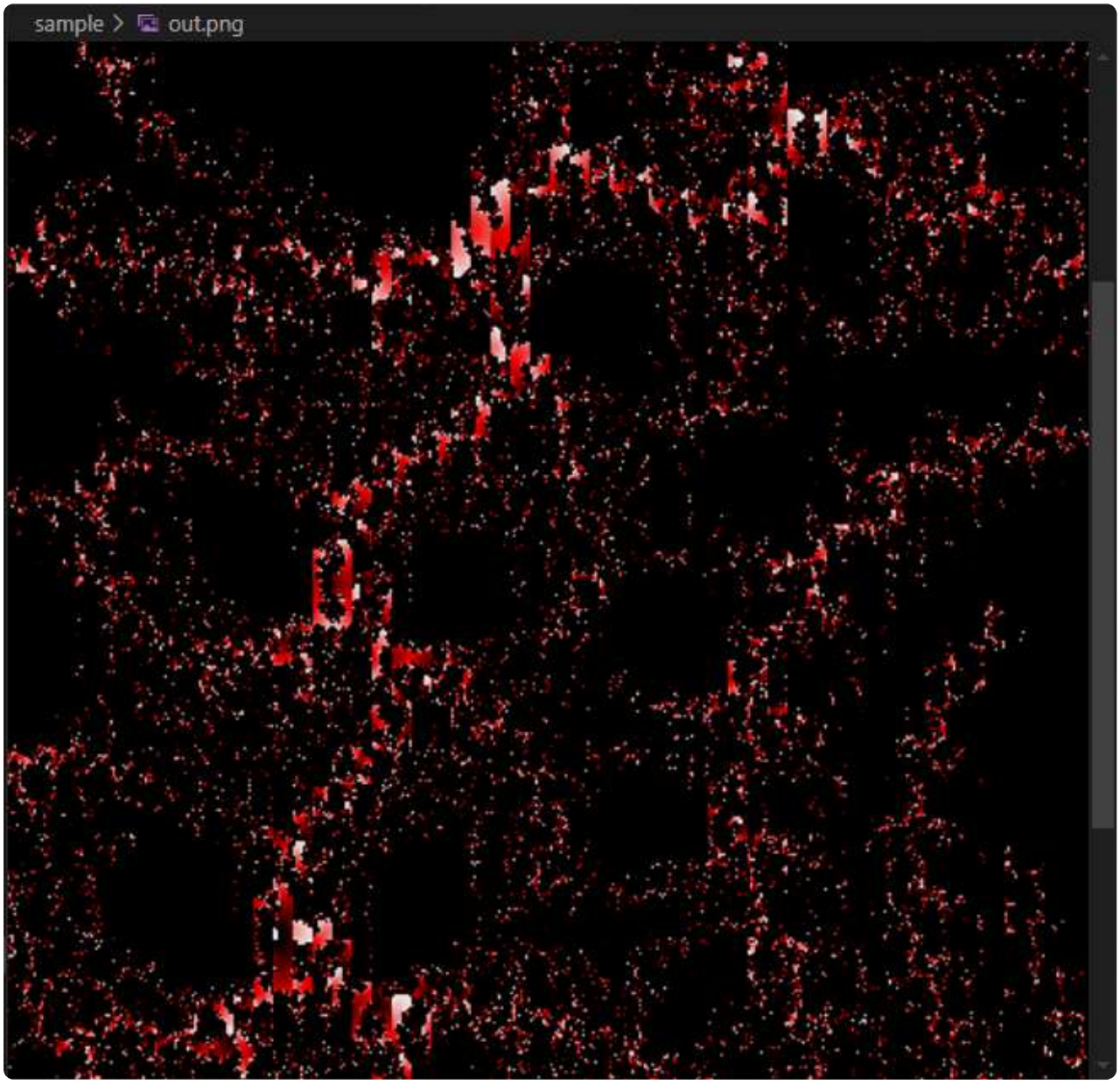
- 在分配 process 的過程，會需要去考量 testcase 的 input，除了閱讀 spec 以外，我還有去查 Mandelbrot Set 的計算方法是甚麼，在大致了解以上先備知識後，實作過程有遇到某些測資一直出現 wrong answer 的 result，在使用 `out,png` 將圖片輸出時才發現兩個嚴重的問題:

1. 有 row 沒跑到 -> 這個部份我當時發現是因為我分每個 process 的 task 量沒考慮好，因為我的分法是當 `height / size` 有餘數時，會將餘數全部丟到最後一個 process，此時在最後一個 process 做 `MPI_Gatherv` 時，如果沒去處理 `all_h` 及 `displs` 這兩個變數的話會出現 error，原因是因為 `all_h` 應該要去計算各個 process 的數據量，而後面這個在 call `MPI_Gatherv` 之前的 for loop 就是在處理這兩個 `int` type 的 array :

```
for (int i = 0; i < size; ++i)
{
    int new_local_height = (i == size - 1) ? SumModMul : heightMulSize;
    all_h[i] = new_local_height * width;
    displs[i] = i * heightMulSize * width;
}
```

- 在 for loop 中，也要像前面一樣去判斷是否為最後一個 Process，是的話會需要重新 assign 數據量(第 i 個 process) `all_h[i] = new_local_height * width` (這個 process 處理的 pixel 數量)
- 數據位移 position: `displs[i] = i * heightMulSize * width` 這個變數主要是去記錄這個 process 在 recieve buffer 中的起始位置。
- 故如果沒有在這邊加上這行 `int new_local_height = (i == size - 1) ? SumModMul : heightMulSize;` 則會產生 process 少跑到數據的狀況。

2. 第二個問題出在發現在向量化後，圖片中每 8 個 pixels 左右顛倒了:



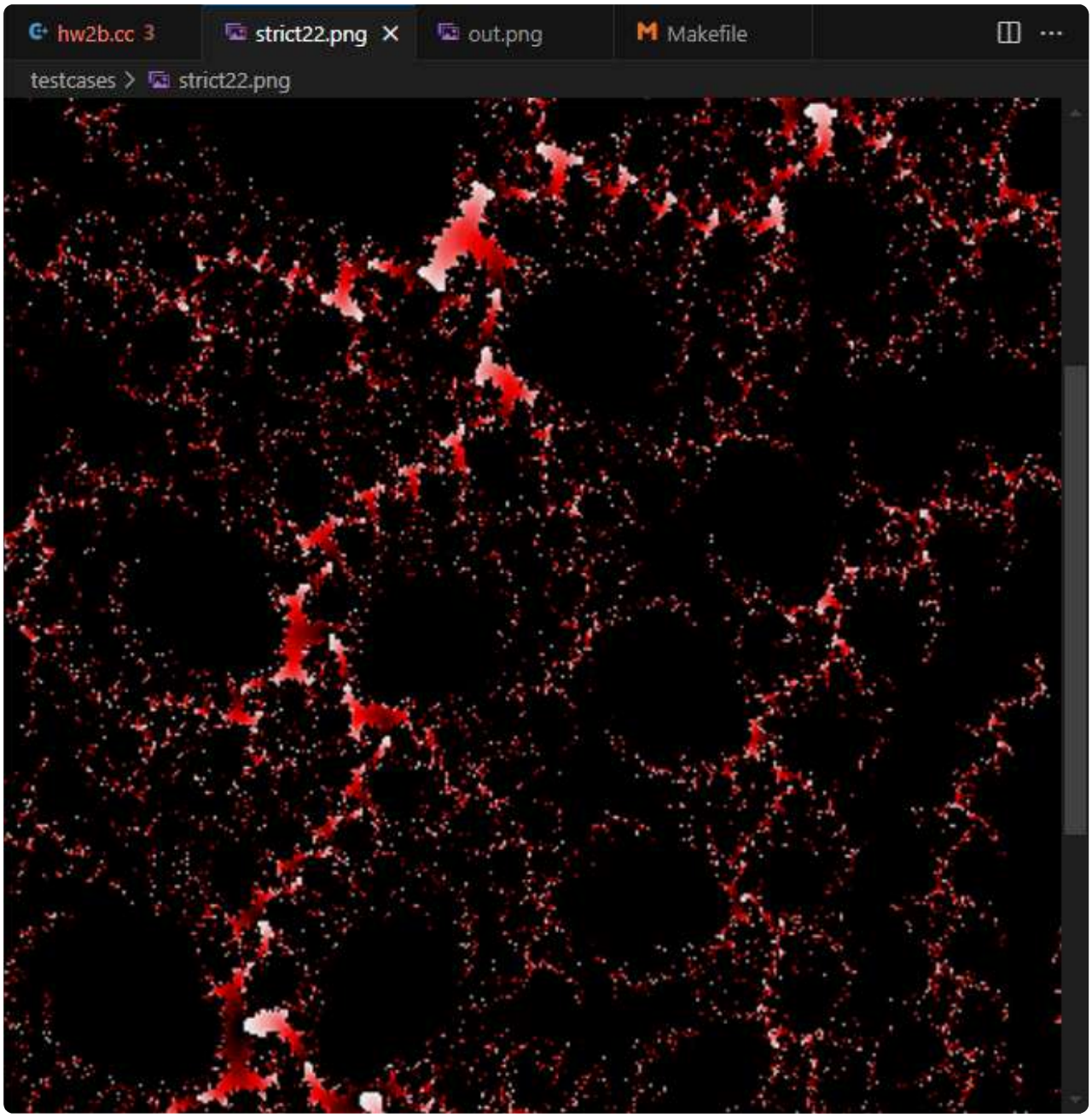
原因是 `__m512d` 存資料的方式顛倒 MSB -> LSB， 而下面將 data 從 `__m256i` 中的 `repeats` store 進 `temp array` 中，會是 LSB -> MSB:

```
int temp[8];
_mm256_store_epi32(temp, repeats);
for (int k = 0; k < 8; ++k)
{
    local_image[j * width + i + k] = temp[k];
}
```

- 故要將一開始的 `__mm512d x0` 中的資料反著存:

```
__m512d x0 = _mm512_set_pd(
    i+7,
    i+6,
    i+5,
    i+4,
    i+3,
    i+2,
    i+1,
    i
);
```

- 得出正確的 `out.png` 圖片:



Conclusion

這份作業其實對於熟悉平行程式的工具起到了很大的幫助，一開始的時候對於很多平行化程式的工具都非常的不熟，但在做完這份作業後才發現這些工具真的很好用，尤其是利用 Nsight 做實驗後，視覺化所有 process 與 thread 才發現原來我的程式在某些方面有很多改善的空間，原本認為平行程式是一個很抽象的東西，因為需要去思考每個 process 與 thread 中要分配多少工作量，一開始只能靠憑空想像程式平行化之後的效能，直到使用 Nsight 並且把數據圖表化才一目了然自己的程式效能原來長這樣，我認為在每個 thread 中的 load balance 表現是不錯的，但我的程式 bottle neck 出在每個 rank 中的工作量不均，我認為在接下來的作業我會想一個新的工作分配方法去讓 process 跟 process 之間的工作量能更 load balance 一點。