CS542200 Parallel Programming: Homework 2 - Mandelbrot Set

Name: 簡佩如
Student ID: 112065525

1. Implementation

1.1 Pthread Version

在 `hw2a` 中，我利用多個 thread 和 SIMD 指令來平行化 Mandelbrot 集合的計算。每個 thread 使用 `curTask` 動態分配圖像的 row，並使用 AVX-512 SIMD 指令在 row 內平行處理 8 個像素。最終結果被各個 thread 分別計算後寫入共享的圖像陣列中，再輸出為 PNG 文件，加速了整體計算與輸出過程。

- **Implementation Approach**：
  - 使用 SIMD 指令與多 pthread 來平行化 Mandelbrot 集合的計算。
  - `mandelbrot_simd` 函式利用 AVX-512 指令同時處理 8 個像素，顯著減少每次迭代的計算時間。
  - 多 thread 處理不同的 row，提高平行度並加速整體效能。
- **Task Partitioning**：
  - 每個 thread 依據 row 進行分配，使用全域原子變數 `curTask` 來追蹤當前處理的 row。
  - `curTask.fetch_add(1)` 確保每次 row 分配是唯一的，避免競爭條件。
  - 每個 row 內以 8 像素為步進 (step) 分區，利用 SIMD 處理 8 個像素點，提升計算效率。
- **Performance Optimization**：
  - **SIMD 加速**：AVX-512 指令向量化計算，通過 mask 機制篩選像素，避免不必要的計算。
  - **多 thread**：創建多達 128 個 thread 處理 row，充分利用 CPU 資源。
  - **避免競爭**：使用 `std::atomic<int>` 保證每個 thread 的分配同步。
  - **PNG 壓縮設定**：將壓縮等級從 1 調整為 0，減少壓縮計算，提升 I/O 效率。
- **Other Efforts**：
  - 嘗試不依賴 CPU 可用數量而開啟 128 個 thread，但遇到以下問題：
    - 過多的 thread 切換增加 CPU 負擔。
    - 多餘的 thread 消耗系統資源，增加排程開銷。

1.2 Hybrid Version (MPI + OpenMP)

在 `hw2b` 中，我使用 MPI 和 OpenMP 的混合並行方法來計算 Mandelbrot 集合。首先，MPI 將圖像的 row 分配給不同的 process，每個 process 負責特定範圍的 row 計算；接著，process 內部使用 OpenMP 將每個 row 的像素計算分配到多個 thread 並行處理，並使用 AVX-512 指令優化計算速度。最終各 process 計算完成後，worker process 將結果傳送回 master process，合併並輸出最終 PNG 文件。

- **Implementation Approach**：

    - 使用 MPI 分配圖像 row 至各 process，process 內部通過 OpenMP 平行化 row 內計算。
    - MPI 負責跨 process 的 row 處理，OpenMP 則平行化 row 內的列計算，充分利用多核架構。
- **Task Partitioning**：
    - **process 間**：每個 process 根據 rank 計算並分配 row，以確保各 process 負載均衡。
    - **process 內**：利用 OpenMP 的 `#pragma omp parallel for schedule(dynamic)` 動態分配列，確保每個 thread 取得合適的工作負載。
- **Load Balancing Strategy**：
    - **AVX-512 SIMD 向量化**：使用 AVX-512 指令加速計算，每次操作 8 個浮點數，減少迭代次數。
    - **MPI 分散式計算**：通過 MPI 分配 row，提高多核利用率，適合大資料集。
    - **OpenMP row 內平行化**：OpenMP 動態分配列計算，進一步平行化，提升 row 內效能。
    - **PNG 壓縮設定**：將壓縮等級從 1 調整為 0，減少壓縮計算，提升 I/O 效率。
- **Other Efforts**：
    - 初期嘗試了 Master-Slave 模式，但會遇到 Master process 必須多次跟其他 process 通訊，進而成為瓶頸的問題。
    - **改進策略**：
        - 採用交錯與批量分配方式，讓 process 以交替的 row 進行計算，減少通信次數。
        - 我讓每個 process 一開始就領完所有需要計算的 row，盡量避免 process 之間的通訊，可以大幅提升效率。
        - 每個 process 在領取 row 的時候採取交替領取，可以避免有些 process 領到特別難計算的 rows。
        - 使用 RowData 結構與自訂 MPI 資料型態，將需要的資訊可以在最少的傳輸次數就完成，進一步提升 MPI 傳輸效率。
        - 這些改善方法大幅提升程式效率，大概提升20秒左右！

2. Experiment & Analysis

2.1 Experimental Setup

System Specifications：

實驗在 QTC 平台上執行

Performance Metrics：

Pthread Version

1. **Scalability**：
    - 使用 `gettimeofday()` 函數測量執行時間。
    - 編寫了下列腳本，測試單一 process 下不同 thread 數量的時間消耗，其中我測試的 thread 數量為 [1 2 4 8 16 32 64 84 96]
    - 腳本示例如下：

```bash
#!/bin/bash
threads_list=(1 2 4 8 16 32 64 84 96)

program="./hw2a"
output="out.png"
max_iter=734
minR=0.27483841838734274
maxR=0.4216774226409377
minI=0.5755165572756626
maxI=0.5039244805312306
width=1920
height=1080

> execution_times.txt

for threads in "${threads_list[@]}"; do
    echo "Running with ${threads} threads..."
    # Set the OMP_NUM_THREADS environment variable
    export OMP_NUM_THREADS=${threads}

    # Execute the program and write results to execution_times.txt
    srun -n1 -c${threads} ${program} ${output} ${max_iter} ${minR} ${maxR} ${minI} ${maxI} ${width}

    echo "Completed run for ${threads} threads."
done

echo "All experiments completed. Results saved in execution_times.txt."
```

- 利用 Python 程式解析 `.txt` 檔案，並輸出圖片進行視覺化分析。

2. **Load Balancing**：
   - 使用 `nsight systems` 和 `nvtx` 計算每個 thread 的運行時間。

```
int thread_id = sched_getcpu();
string range_name = "Thread Compute Start: Thread " + to_string(thread_id);
nvtxRangePush(range_name.c_str());
```

   - 編寫了 Python 程式碼以解析和視覺化 load balancing 結果。
   - 對於每筆測試資料，我使用 1 個 process 和 12 個 thread。
3. **Vtune**：
   - 使用 vtune 進行熱點分析。

Hybrid Version (MPI + OpenMP)

1. **Scalability**：
   - 使用 `gettimeofday()` 函數測量執行時間。
   - 編寫了下列腳本，測試不同 process 數量的時間消耗，其中我測試的 process 數量為 [1 2 4 8 16 32 48 96]
   - 腳本示例如下：

```bash
#!/bin/bash

# Define the list of process counts to test
process_list=(1 2 4 8 16 32 48 96)

# Define the program name and output file
program="./hw2b"
output="out.png"
max_iter=734
minR=0.27483841838734274
maxR=0.4216774226409377
minI=0.5755165572756626
maxI=0.5039244805312306
width=1920
height=1080

# Specify the result file name
result_file="execution_times_Fast10_2b.txt"

# Clear previous results
> $result_file

# Loop through the different process counts
for processes in "${process_list[@]}"; do
    echo "Running with ${processes} processes..."

    # Execute the program and capture the elapsed time in result file
    result=$(srun -n${processes} ${program} ${output} ${max_iter} ${minR} ${maxR} ${minI} ${maxI} $

    # Extract the elapsed time from the result
    elapsed_time=$(echo ${result} | awk '{print $3}')

    # Write process count and elapsed time to file
    echo "${processes}, ${elapsed_time}" >> $result_file

    echo "Completed run for ${processes} processes."
done

echo "All experiments completed. Results saved in $result_file."
```

- 利用 Python 程式解析 `.txt` 檔案，並輸出圖片進行擴展性分析。

2. **Load Balancing**：
   - 使用 `time` 計算每個 process 的運行時間，例如：

     ```bash
     srun -n16 time ./hw2b output.png 10000 -2 2 -2 2 800 800
     ```

   - 編寫腳本捕捉所有 rank 的運行時間：

```bash
#!/bin/bash

# Check if the number of arguments is correct
if [ "$#" -ne 10 ]; then
    echo "Usage: ./wrapper.sh <data_name> <program> <output.png> <maxIter> <minR> <maxR> <minI> <ma
    exit 1
fi

# Extract arguments
DATA_NAME=$1
PROGRAM=$2
OUTPUT=$3
MAX_ITER=$4
MIN_R=$5
MAX_R=$6
MIN_I=$7
MAX_I=$8
WIDTH=$9
HEIGHT=${10}

# Run the program and capture output
srun -n16 "$PROGRAM" "$OUTPUT" "$MAX_ITER" "$MIN_R" "$MAX_R" "$MIN_I" "$MAX_I" "$WIDTH" "$HEIGHT" >

echo "Execution times for all ranks saved in ${DATA_NAME}_times.txt"
```

- 使用 Python 程式碼解析並視覺化每個 process 的運行時間，分析負載平衡的有效性。
- 對於每筆測試資料，我使用固定 16 個 process 來進行測試。

3. **Vtune**：
   - 使用 vtune 進行熱點分析。

2.2 Results & Analysis

- **Test Case Descriptions**：
  我選擇了四組不同特性的測資，具體說明如下：
  i. **Fast10**
     - **Configuration**：
       `734 0.27483841838734274 0.4216774226409377 0.5755165572756626 0.5039244805312306 1920 1080`
     - **Description**：該組數據屬於小型測試數據集，迭代次數較少，圖像解析度為 1920 x 1080。
  ii. **Slow01**
     - **Configuration**：
       `174170376 -0.7894722222222222 -0.7825277777777778 0.145046875 0.148953125 2549 1439`
     - **Description**：該測試數據屬於中型數據集，具有較高的迭代次數和中等解析度，圖像尺寸為 2549 x 1439。
  iii. **Slow16**
     - **Configuration**：
       `26063 -0.28727240825213607 -0.2791226112721823 -0.6345413372717312 -0.6385148107626897 3840 2160`
     - **Description**：此測試數據也是中型測試集，具有適中的迭代次數和較高的解析度，圖像尺寸為 3840 x 2160。
  iv. **Strict34**
     - **Configuration**：
       `10000 -0.5506164691618783 -0.5506164628264113 0.6273445437118131 0.6273445403522527 7680 4320`

- **Description**：此組數據為大型測試集中的極限配置，圖像解析度極高，達到 7680 x 4320。此測試數據通常需要較長時間完成運算，通常也是我跑最久的測資。

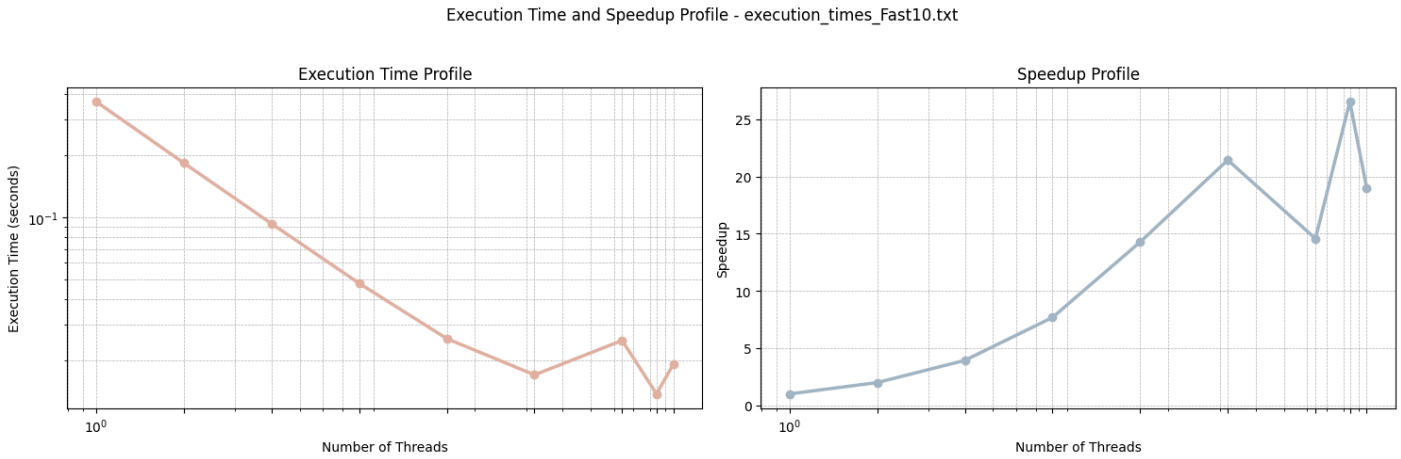
v. 希望透過實驗與分析這四組不同規模的測試數據，更深入了解程式的擴展性、負載平衡，以及在不同硬體配置下的性能表現。

2.2.1 Scalability

Pthread Version

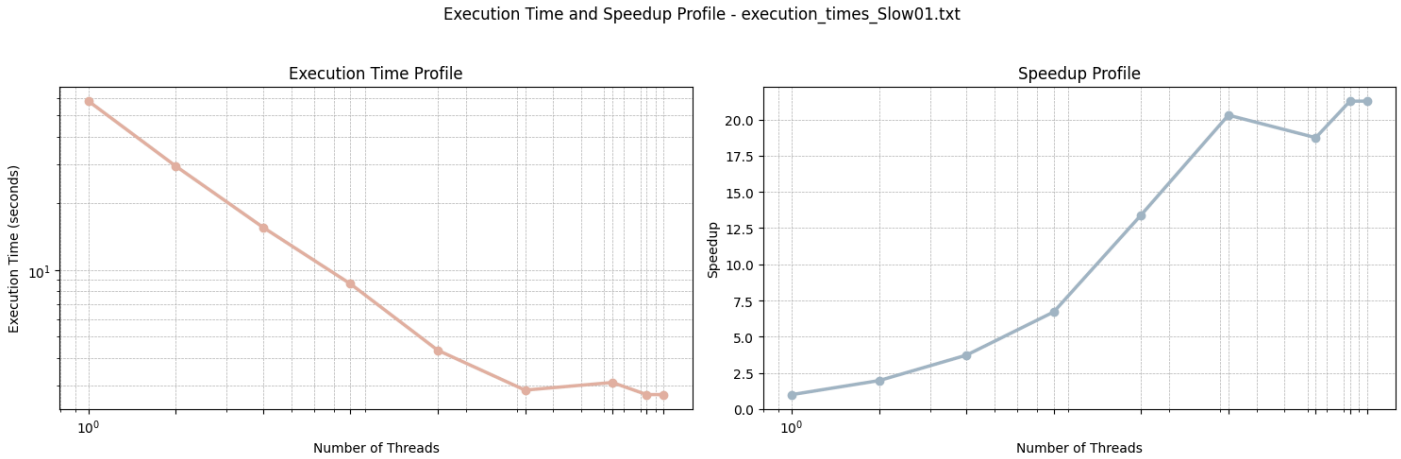
- **Scalability Plots**：

1. **Fast10 Execution Time and Speedup Profile**



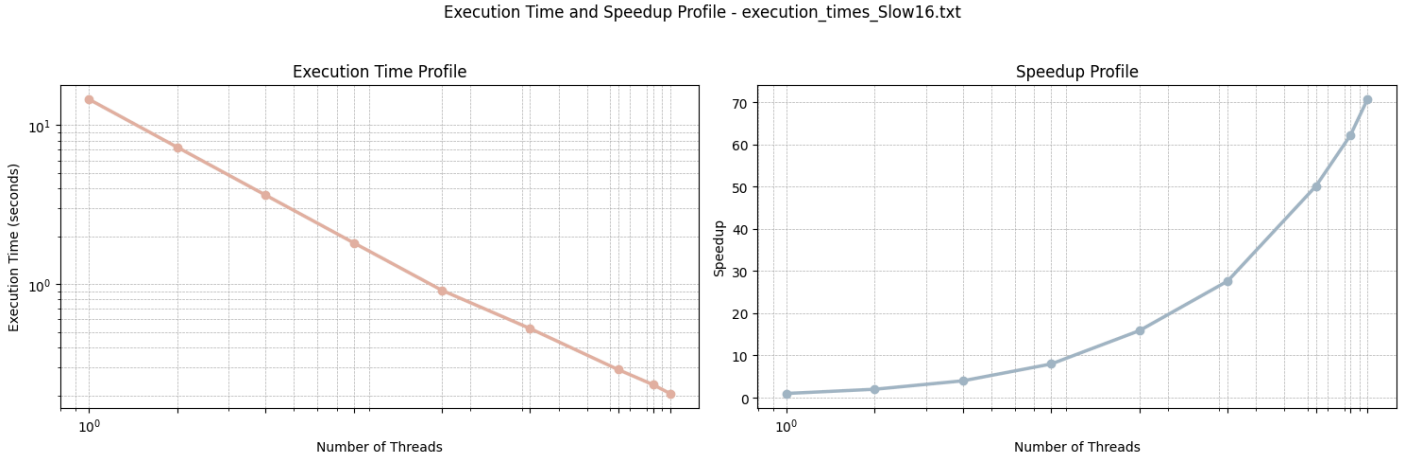
- **描述**：此圖展示了 `Fast10` 測試數據的執行時間和速度提升。在低執行緒數下執行時間下降，但在更高的執行緒數下，執行時間波動，這可能與該測試數據的規模較小有關，導致多執行緒的優勢不明顯。

2. **Slow01 Execution Time and Speedup Profile**



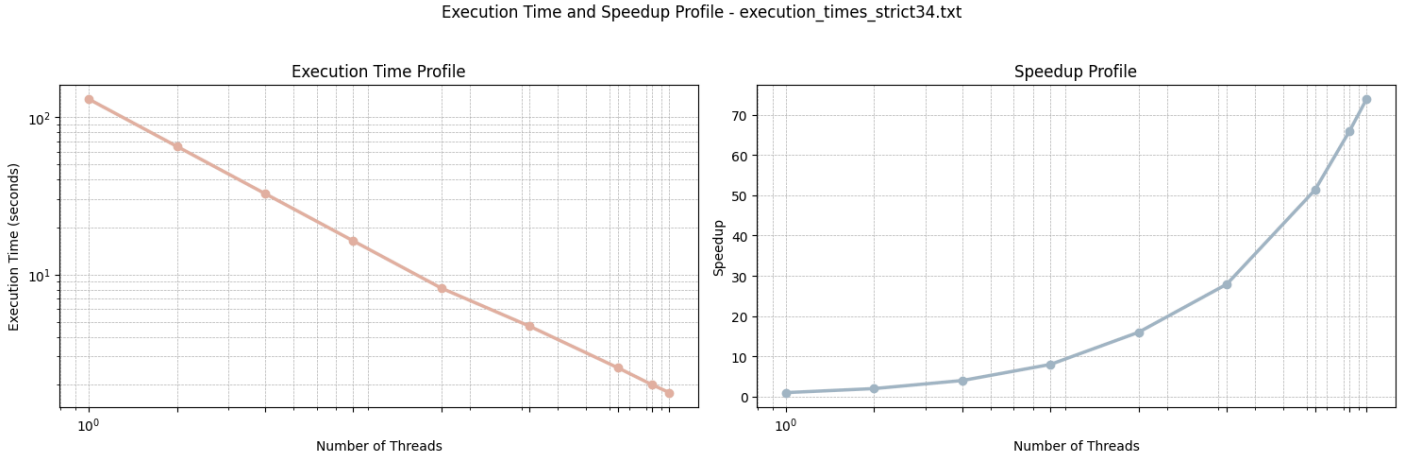
- **描述**：此圖展示了 `Slow01` 測試數據的執行時間和速度提升曲線。隨著執行緒數量增加，執行時間穩定下降，速度提升曲線表現出持續增長，表明該測試數據的規模適合多執行緒優化。

3. **Slow16 Execution Time and Speedup Profile**



- **描述**： `Slow16` 測試數據展示了較為穩定的執行時間下降和速度提升趨勢。隨著執行緒數量的增加，執行時間顯著減少，速度提升曲線呈現接近線性增長，這表明程式在該測試數據下的擴展性非常理想。

4. **Strict34 Execution Time and Speedup Profile**



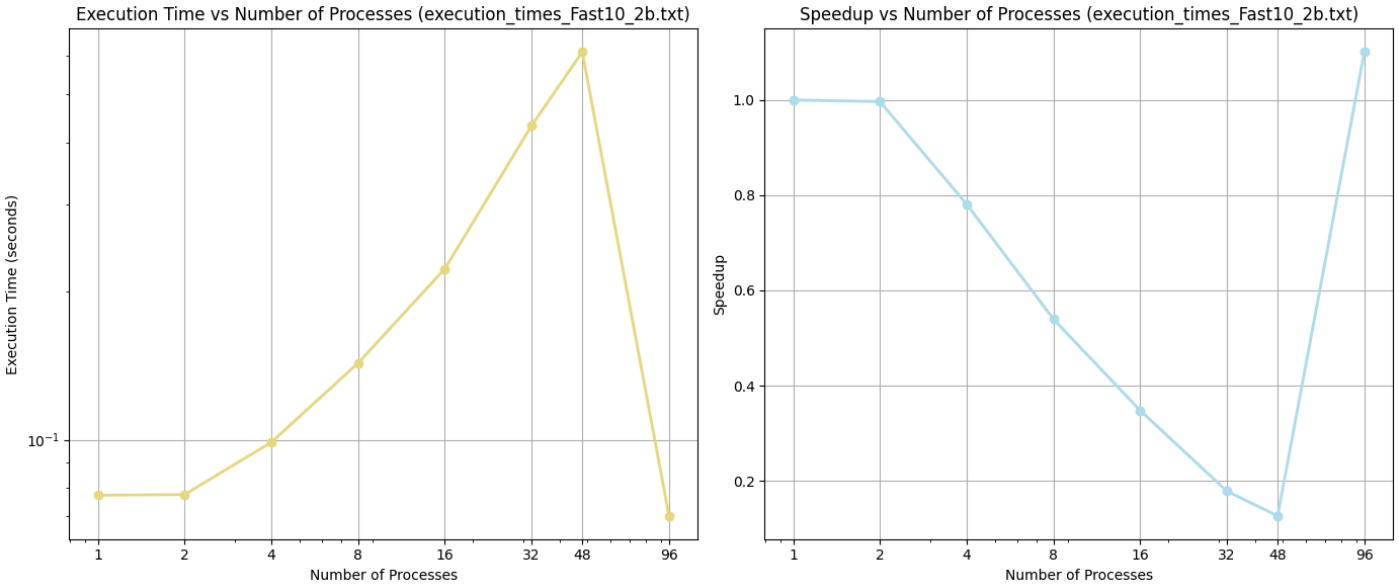
- **描述**：此圖展示了 `Strict34` 測試數據在不同執行緒數量下的執行時間和速度提升。從圖中可以看到，執行時間隨著執行緒數量增加而呈現指數下降趨勢，而速度提升也相應地呈現逐漸增長，反映了程式良好的擴展性。

- **Scalability Analysis**
  隨著測試數據計算量的增大，Pthread 版本在多執行緒下的可擴展性逐漸提升。透過比較不同測試數據的執行時間與速度提升圖，我們可以觀察到，在大型數據集下，多執行緒的優勢更為明顯。這是因為資料量越大，單一執行緒的運行時間越長，增加執行緒數量的收益也越高，從而降低了運算瓶頸並提高了程式的效率。
- **Execution Time Profile**
  執行時間的圖表展示了，隨著執行緒數量增加，執行時間顯著減少。特別是在較大數據集上，例如 `Strict34` 和 `Slow16` 測試數據，隨著執行緒數量的增長，執行時間呈現穩定且顯著的下降趨勢。這表明程式能夠有效利用多執行緒處理大量計算任務，從而縮短總處理時間。
- **Speedup Profile**
  速度提升圖表顯示，隨著執行緒數量增加，速度提升曲線逐漸上升。在大型數據集（例如 `Strict34`）下，速度提升更為顯著，幾乎達到了理想的線性增長。這是因為計算負載的增加讓每個執行緒有足夠的工作量，減少了多執行緒間的同步開銷和閒置時間。在較小的數據集（例如 `Fast10`）中，速度提升幅度則略有波動，可能是因為計算量較少，且多執行緒的開銷相對增大，削弱了多執行緒的效果。

Hybrid Version (MPI + OpenMP)

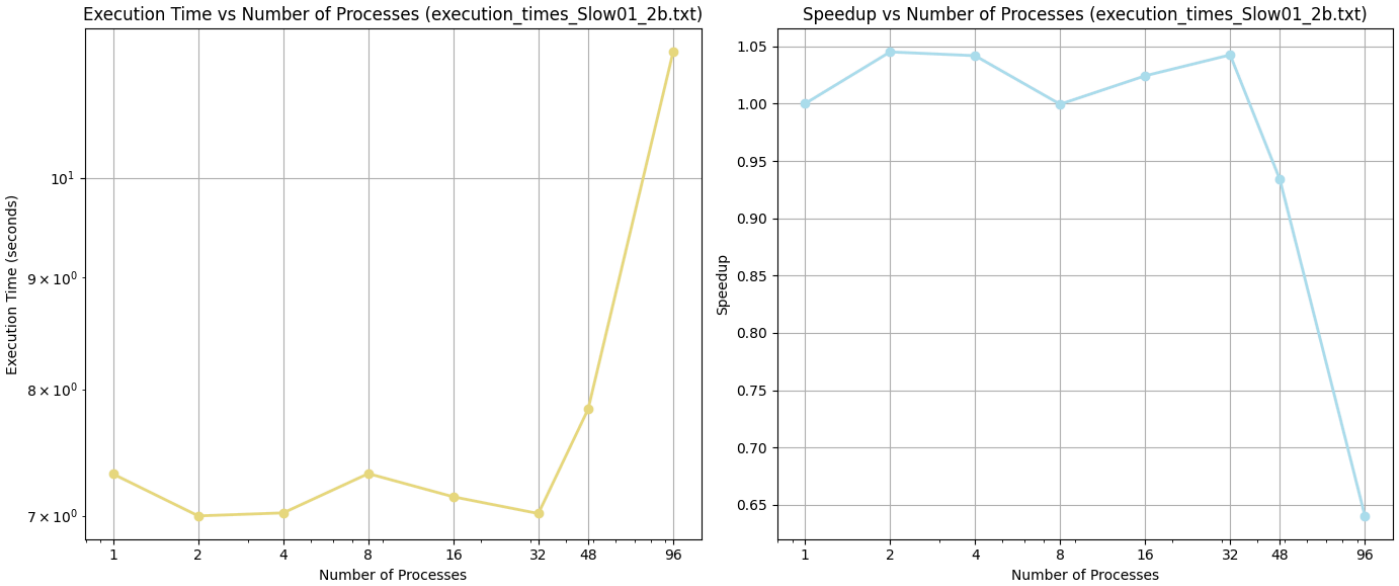
- **Scalability Plots**：

1. **Fast10 Execution Time and Speedup Profile**



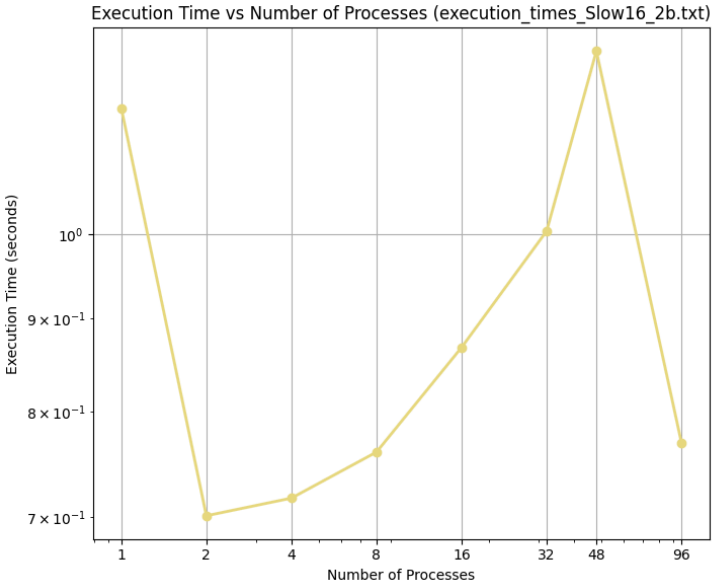
- **描述**：在 `Fast10` 測試數據中，隨著進程數量增加，執行時間最初有所下降，但在更高進程數量下執行時間波動，這可能是由於數據集較小，平行化開銷超過了效能收益，限制了可擴展性。

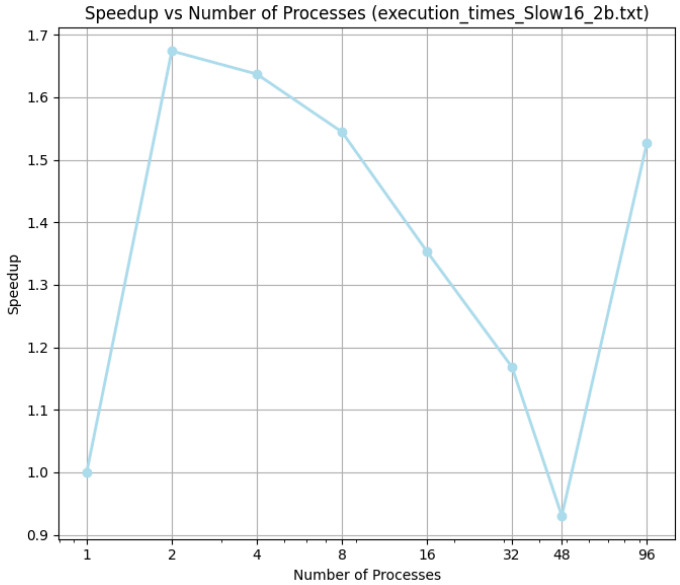
2. **Slow01 Execution Time and Speedup Profile**



- **描述**： `Slow01` 測試數據隨著進程數量增加，執行時間穩定減少，加速比曲線也相對穩定增長，顯示該數據集適合平行處理。少數波動可能與進程間的通信開銷有關。

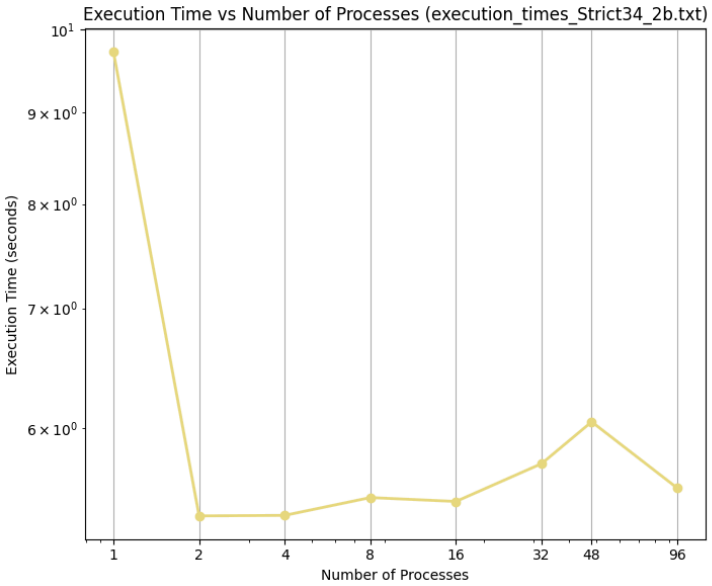
3. **Slow16 Execution Time and Speedup Profile**

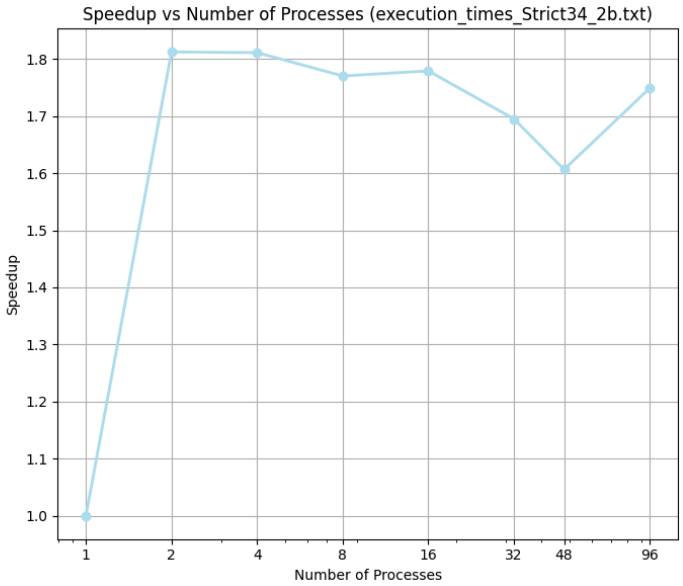




- **描述**： `Slow16` 測試數據顯示出隨著進程增加，執行時間持續下降，加速比接近線性增長，表現出優秀的可擴展性。數據集較大，讓進程間的任務分配更有效，減少了同步開銷。

4. **Strict34 Execution Time and Speedup Profile**





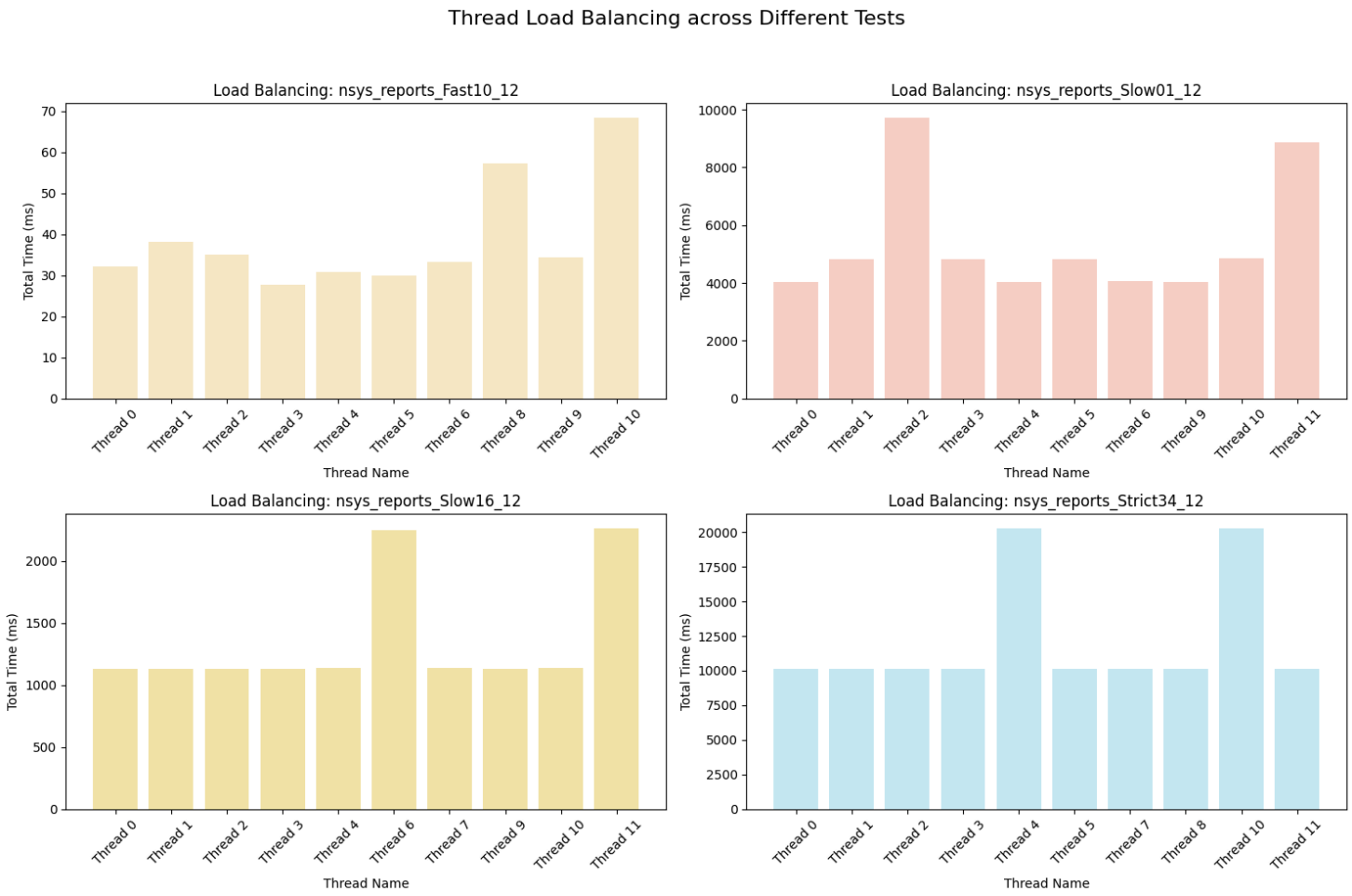
- **描述**： `Strict34` 測試數據作為大型測試案例，顯示隨著進程數量增加，執行時間顯著下降，加速比穩定上升。數據規模大，能夠充分利用 MPI 和 OpenMP，達到接近線性的加速效果，反映出高可擴展性。

- **Scalability Analysis**：
  Hybrid 版本（MPI + OpenMP）展示出較高的可擴展性，特別是在大型數據集下。隨著數據集增大（如 `Slow16` 和 `Strict34`），增加進程數量持續減少執行時間，並有效提高加速比，因為工作負載能被高效分配。然而在小型數據集（如 `Fast10`）中，由於平行化開銷的影響，可擴展性受到限制，說明這種混合模式特別適合於大規模計算。
- **Execution Time Profile**：
  執行時間圖表顯示隨著進程數量增加，執行時間顯著減少，特別是在大數據集上。這說明隨著計算需求增加，Hybrid 模式利用進程（MPI）和執行緒（OpenMP）進行任務分配，有效縮短總處理時間，顯示出該模式在大量計算任務下的優勢。
- **Speedup Profile**：
  加速比圖表顯示，隨著進程數量增加，較大數據集的加速效果更為顯著。在小數據集（如 `Fast10`）中，加速比曲線呈現波動，這是由於計算量較少，且平行化開銷影響較大，削弱了加速效果。但在大數據集（如 `Strict34`）中，加速比接近線性增長，反映出計算負載足夠，進程間同步開銷較低，使得 Hybrid 模式發揮出色的效能。

2.2.2 Load Balancing

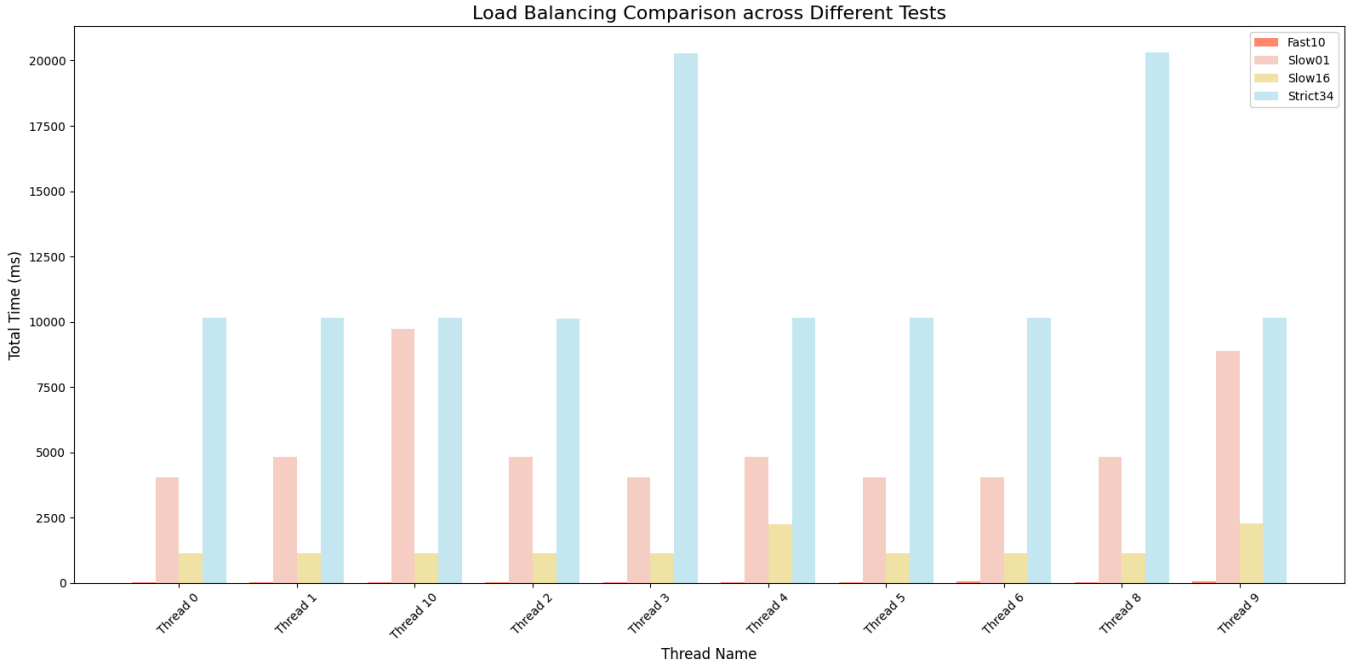
- **Load Balancing Plots**：展示負載平衡的圖表，分析不同配置下的負載分配。
- **Analysis**：討論負載平衡的效果，並指出特定配置下的平衡性差異。

Pthread Version



1. **Fast10**
   - **描述**：在 `Fast10` 測試數據下，各執行緒的總運行時間差異較大，特別是部分執行緒（例如 Thread 6 和 Thread 9）有較高的負載。這可能是由於 `Fast10` 的計算量較少，導致負載在不同執行緒之間分配不均，影響了整體的平衡性。
2. **Slow01**
   - **描述**： `Slow01` 測試數據的負載平衡較 `Fast10` 更均勻，但仍然可以觀察到負載差異，特別是在 Thread 2 和 Thread 11 上負載顯著增加。這表明，在中等規模的測試數據下，負載分配仍需進一步優化，以達到更理想的平衡效果。
3. **Slow16**
   - **描述**：在 `Slow16` 測試數據下，各執行緒的負載分佈相對穩定，且總運行時間較為接近，顯示了良好的負載平衡。隨著測試數據規模的增加，Pthread 版本的負載分配效果逐漸顯現出更高的穩定性。
4. **Strict34**
   - **描述**： `Strict34` 是一組大型測試數據，圖表顯示所有執行緒的總運行時間非常接近，負載分配幾乎完全均勻，顯示出非常理想的負載平衡效果。在大規模數據下，多執行緒的分配效率顯著提升。

- **Load Balancing Comparison across Different Tests**



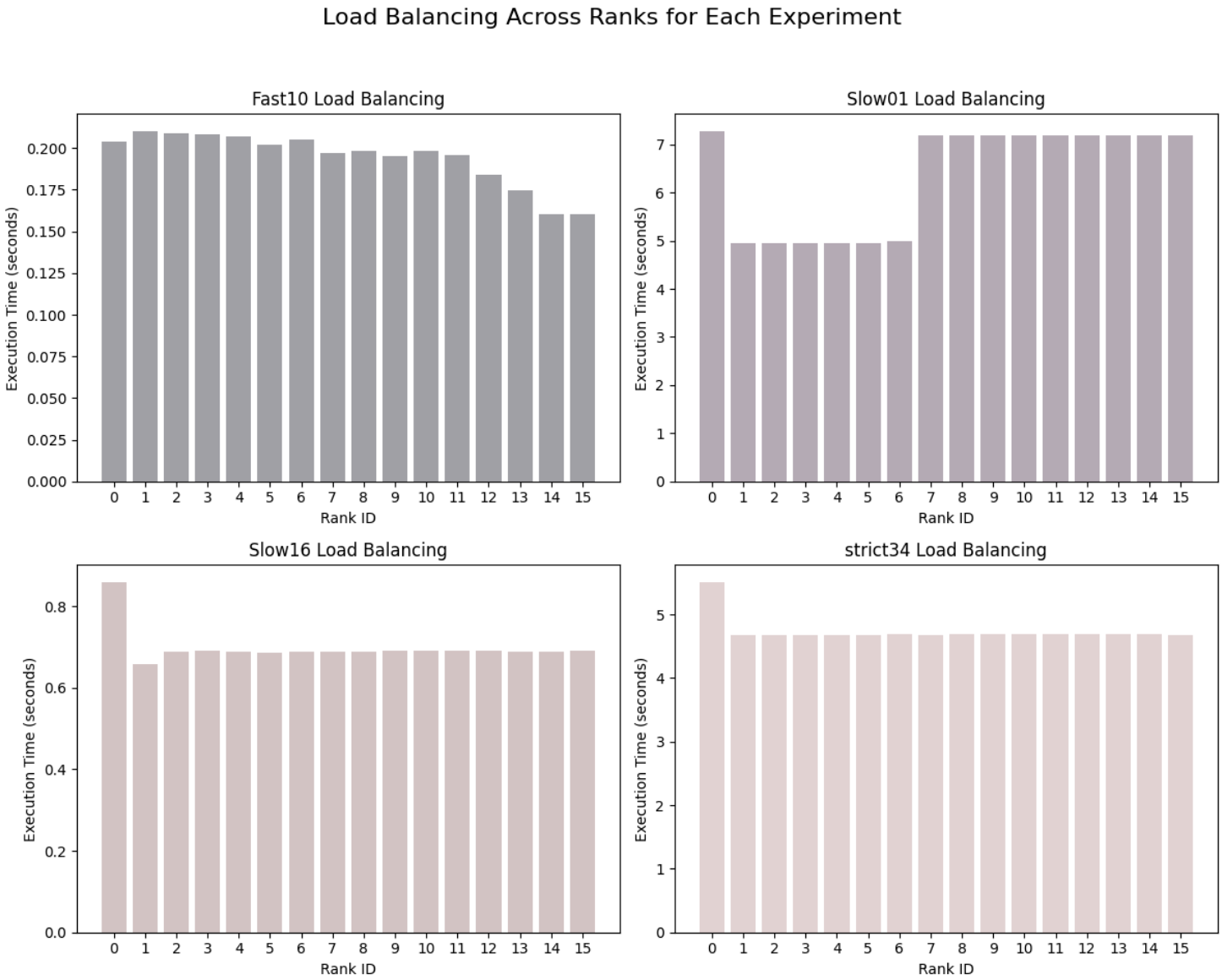
- **描述**：該圖展示了四組測試數據（Fast10、Slow01、Slow16 和 Strict34）在相同執行緒配置下的負載平衡情況。可以觀察到，隨著測試數據規模的增加，各執行緒之間的負載逐漸趨於均衡，特別是在 `Strict34` 上負載平衡效果最佳，而在 `Fast10` 上負載波動較大，顯示出小型數據集在負載分配方面的挑戰。

- **Analysis**

  在不同測試數據和執行緒配置下，Pthread 版本的負載平衡效果有所不同。隨著數據集規模增大，負載在各執行緒間的分佈更加均勻，這是因為在較大規模的數據下，分配給每個執行緒的工作量足夠大，從而減少了負載波動。另一方面，在較小的數據集（如 `Fast10`）上，由於整體計算量少，負載分配不均的情況更為明顯。

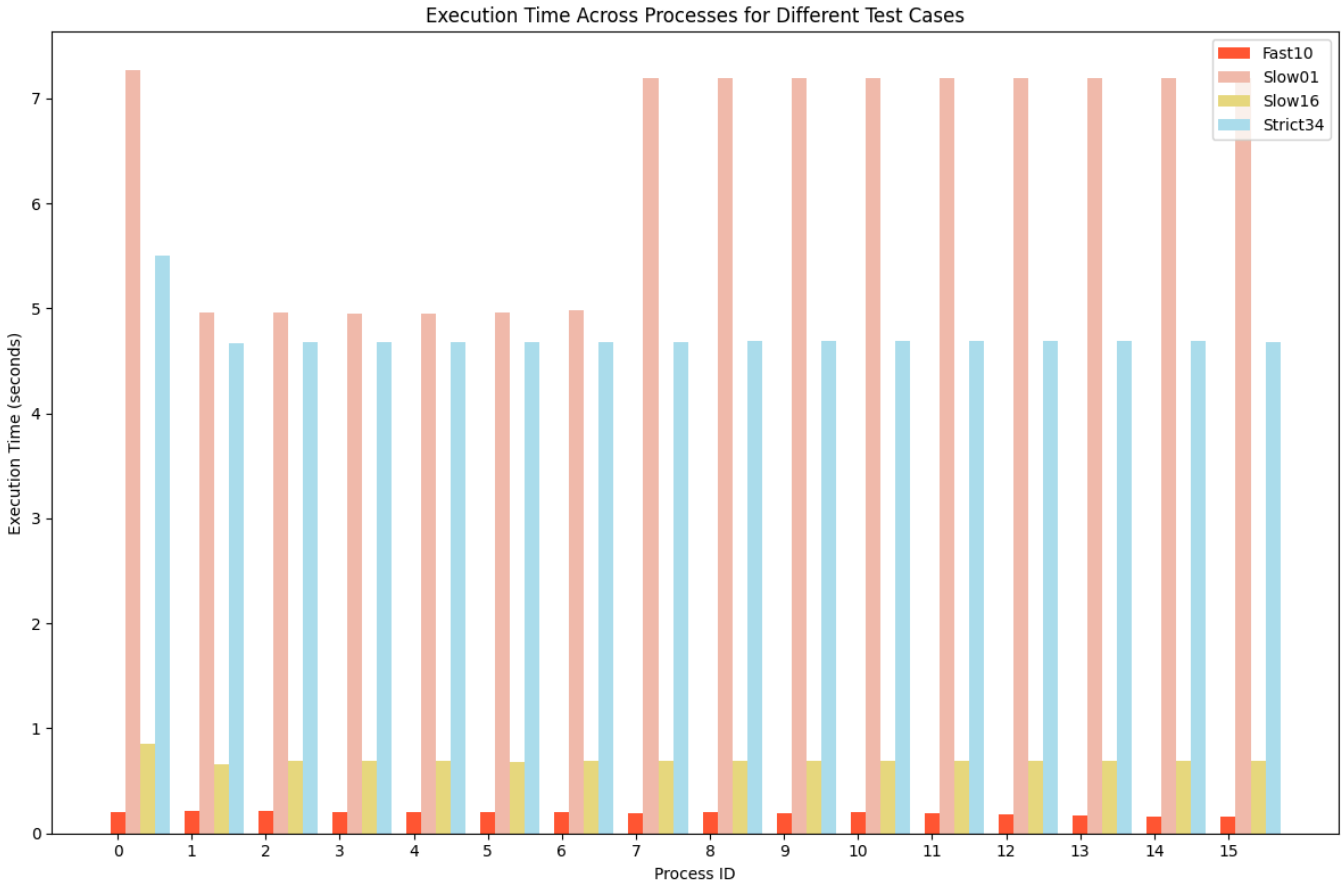
Hybrid Version (MPI + OpenMP)

2.2.2 Load Balancing (Hybrid Version - MPI + OpenMP)



1. **Fast10 Load Balancing**
   - **描述**：在 `Fast10` 測試數據下，各 Rank 的執行時間分布較為均勻，但隨著 Rank ID 的增加，執行時間略微下降。這可能是因為 `Fast10` 的計算量較小，進程之間的負載分配在小數據集上更容易達成平衡。
2. **Slow01 Load Balancing**
   - **描述**：在 `Slow01` 測試數據下，Rank 0 明顯有較高的負載，而其餘 Rank 的負載分布較均勻。這表明在中等規模數據下，某些進程（如 Rank 0）可能承擔了更多的初始計算負荷，導致負載不平衡。
3. **Slow16 Load Balancing**
   - **描述**：在 `Slow16` 測試數據中，各 Rank 的執行時間較為接近，負載分布更均勻。這表明隨著數據集規模的增加，Hybrid 版本能夠更好地分配負載，使進程之間的負載平衡達到較高水平。
4. **Strict34 Load Balancing**
   - **描述**：在 `Strict34` 測試數據中，各 Rank 的負載非常均勻，顯示了非常理想的負載平衡效果。這反映了在大規模數據下，Hybrid 版本的 MPI + OpenMP 能夠充分發揮進程間協同的優勢，使得進程負載更均衡。

- **Load Balancing Comparison across Ranks**



- **描述**：該圖展示了四組測試數據（Fast10、Slow01、Slow16 和 Strict34）在相同進程配置下的負載平衡情況。隨著測試數據規模的增加，各進程之間的負載逐漸趨於均衡，特別是在 `Strict34` 上負載平衡效果最佳，而在 `Fast10` 上負載波動較大，顯示出小型數據集在負載分配方面的挑戰。

- **Analysis**

  在不同測試數據和進程配置下，Hybrid 版本（MPI + OpenMP）的負載平衡效果有所不同。隨著數據集規模增大，進程間的負載分布更加均勻，這是因為在較大規模的數據下，分配給每個進程的工作量足夠大，從而減少了負載波動。在較小的數據集（如 `Fast10`）中，負載分配不均的情況更為明顯，顯示出 Hybrid 版本在小數據集上仍有優化空間。

2.2.3 Hotspots Analysis

Pthread Version

Top Hotspots

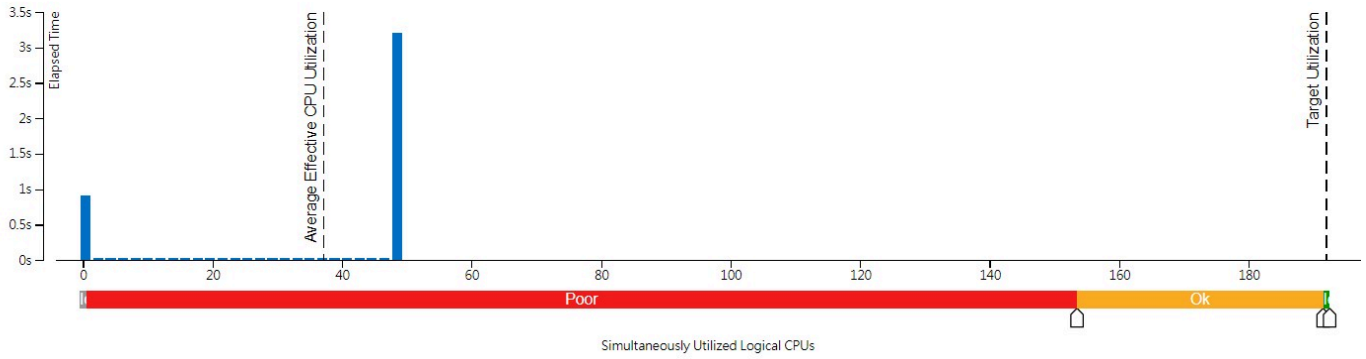
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

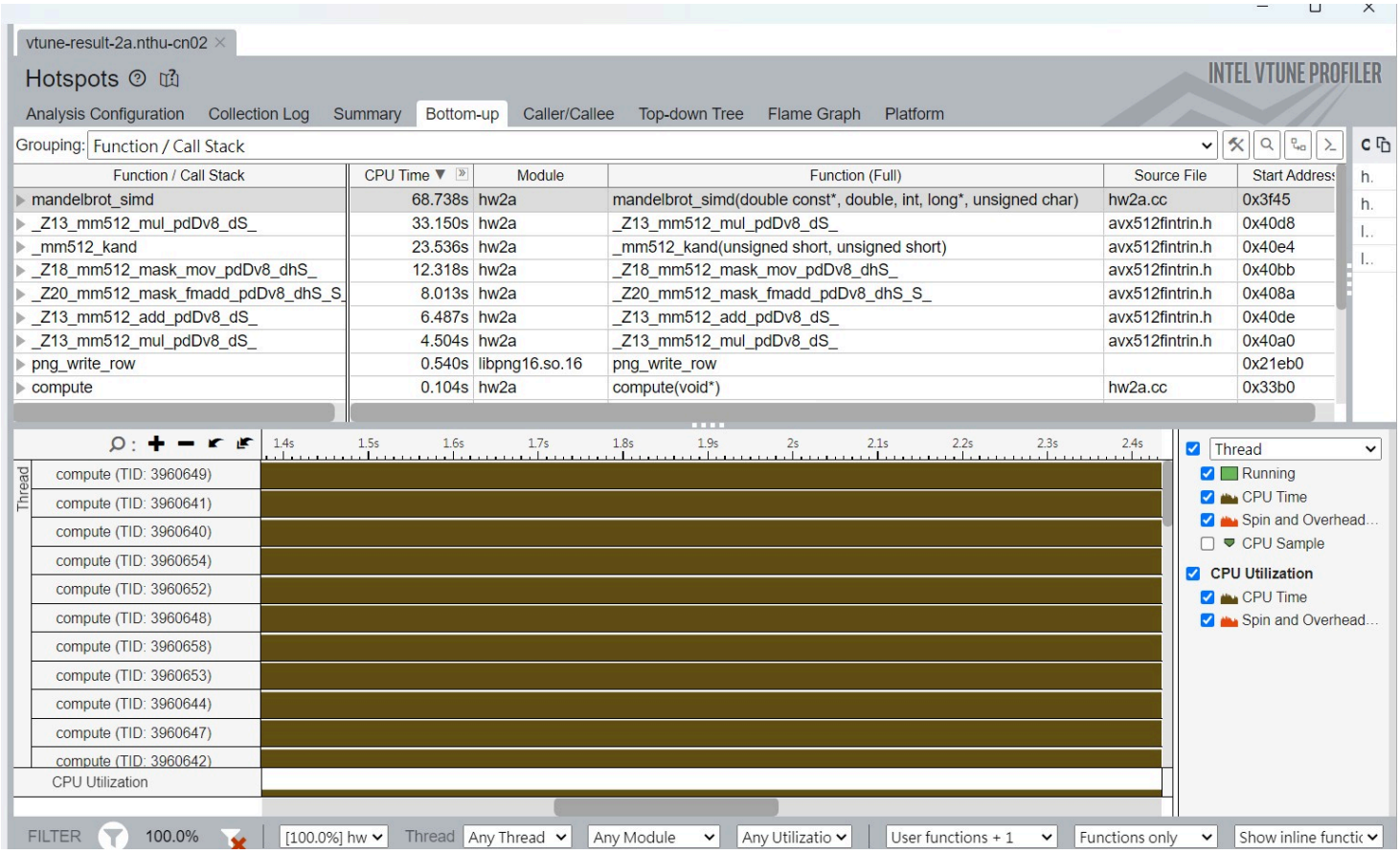
Function	Module	CPU Time	% of CPU Time
mandelbrot_simd	hw2a	68.738s	43.6%
_Z13_mm512_mul_pdDv8_dS_	hw2a	33.150s	21.0%
_mm512_kand	hw2a	23.536s	14.9%
_Z18_mm512_mask_mov_pdDv8_dhS_	hw2a	12.318s	7.8%
_Z20_mm512_mask_fmadd_pdDv8_dhS_S_	hw2a	8.013s	5.1%
[Others]	N/A*	11.851s	7.5%

**N/A is applied to non-summable metrics.*

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.





1. **Top Hotspots 分析**
   - 從熱點分析圖可以看到， `mandelbrot_simd` 函數是 Pthread 版本的主要運算瓶頸，佔據了 43.6% 的 CPU 時間。這一部分主要處理每個像素的計算，利用 SIMD 指令加速了多像素的並行處理。
   - 其他高消耗的指令包括 `_Z13_mm512_mul_pdDv8_dS_` (21.0%) 和 `_mm512_kand` (14.9%)，它們負責處理複數計算和像素篩選，分別使用了 AVX-512 的向量操作。
   - **優化建議**：可以進一步精簡 `mandelbrot_simd` 函數的向量化操作，減少重複計算和不必要的分支，以期降低這些指令的 CPU 時間佔比。
2. **Effective CPU Utilization**
   - CPU 利用率圖表顯示，大多數時間只有一部分 CPU 核心處於活動狀態，且在多數時候的利用率低於目標水平，顯示在低負載區（紅色）。
   - **分析**：這可能由於多執行緒負載不均，導致部分執行緒閒置或工作不飽和。在小型數據集下，這一現象更為明顯。
3. **Thread CPU Time 分析**
   - 從 CPU 利用時間圖可以看到，各執行緒的運行時間分佈較為集中，但存在部分執行緒的運行時間稍長。這可能是由於負載分配不均，或某些操作需要額外的 CPU 時間。

Hybrid Version (MPI + OpenMP)

Top Hotspots

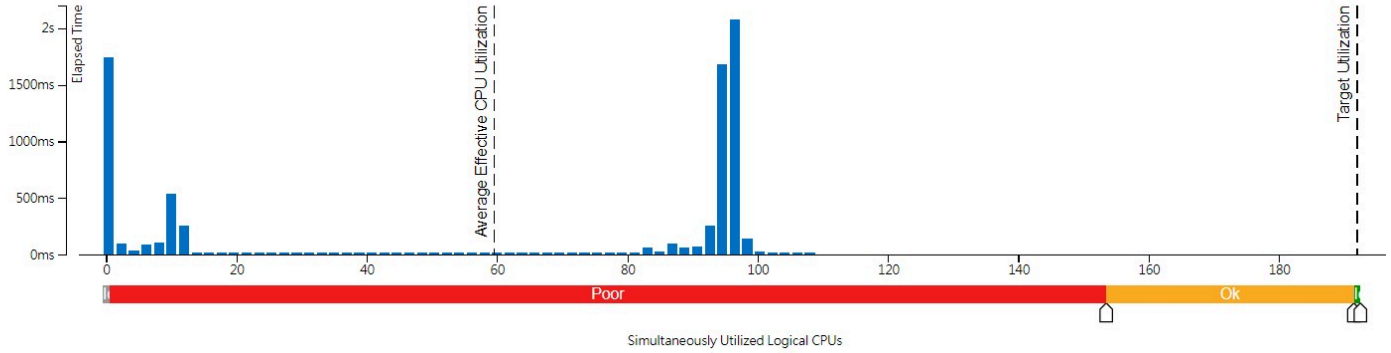
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

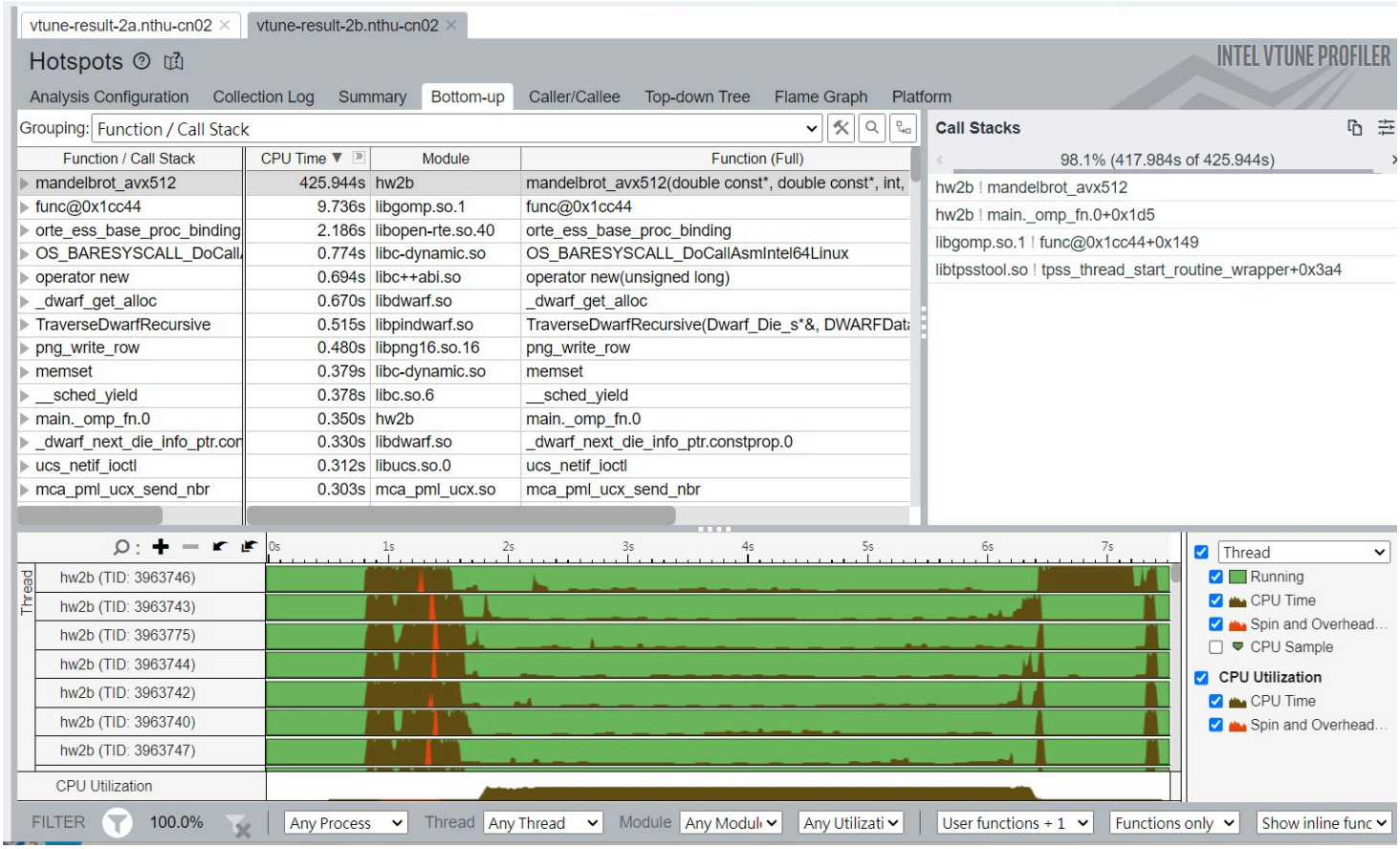
Function	Module	CPU Time	% of CPU Time
mandelbrot_avx512	hw2b	425.944s	94.9%
func@0x1cc44	libgomp.so.1	9.736s	2.2%
orte_ess_base_proc_binding	libopen-rte.so.40	2.186s	0.5%
OS_BARESYSCALL_DoCallAsmIntel64Linux	libc-dynamic.so	0.774s	0.2%
operator new	libc++abi.so	0.694s	0.2%
[Others]	N/A*	9.286s	2.1%

**N/A is applied to non-summable metrics.*

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.





1. **Top Hotspots 分析**
   - 在 Hybrid 版本中， `mandelbrot_avx512` 函數佔據了 94.9% 的 CPU 時間，表明該函數在 Hybrid 版本中的運算佔比更高。這是因為此函數同時被多個 MPI 進程和 OpenMP 執行緒使用。
   - 其他較高消耗的函數如 `func@0x1cc44` 和 `orte_ess_base_proc_binding` ，這些函數主要負責 OpenMP 和 MPI 的內部管理和進程綁定，佔比相對較小。
2. **Effective CPU Utilization**
   - CPU 利用率圖表顯示，大部分時間 CPU 核心都能夠充分利用，但仍存在部分低利用區域（紅色）。
   - **分析**：這可能是因為不同 MPI 進程間的通信開銷導致的短暫閒置。較大的數據集讓進程能夠更持續地執行，但在同步和傳輸期間 CPU 利用率略有下降。
3. **Thread CPU Time 分析**
   - 從 CPU 利用圖可以觀察到，不同進程間的執行時間分佈較為穩定，但某些區段顯示高 CPU 使用率和同步開銷。

2.2.4 Optimization Analysis

- **Optimization Strategies**：根據分析結果提出潛在的優化策略。
  透過這些圖表，我們可以發現 Pthread 版本的主要瓶頸在於 SIMD 指令的執行，而 Hybrid 版本則受到 MPI 進程間通信的影響。優化策略應該針對各版本的特點調整，例如動態負載分配、減少進程間同步，以及優化主要熱點函數的 SIMD 操作，以提高 CPU 資源的利用率並降低整體執行時間。

2.2.4 Optimization Analysis

- **Optimization Strategies**：

1. **Pthread 版本**：
   - **SIMD 優化**：主要瓶頸在於 `mandelbrot_simd` 函數，佔據了 43.6% 的 CPU 時間。可以通過減少冗餘的 SIMD 操作，合併一些 AVX-512 指令來優化此函數，以降低執行時間。此外，透過減少函數內的分支操作，可以進一步簡化

處理流程，提升 SIMD 的運算效率。這個部分我已經盡量優化了，此份實驗就是優化的版本！

- **最佳化執行緒數量**：過多的 thread 會增加 CPU 開銷，特別是在小型數據集上。因此，可以根據工作負載大小動態調整執行緒數量，以提升效率。這個部分我已經盡量優化了，此份實驗就是優化的版本！
- **動態負載平衡**：或許使用任務隊列進行更動態的負載分配，取代靜態的行分配策略，這樣可以緩解小型數據集（如 Fast10）中的執行緒負載不均問題。

2. **Hybrid 版本 (MPI + OpenMP)**：
   - **減少 MPI 同步開銷**：MPI 進程之間的通信開銷影響了性能，特別是在大數據集上。優化通信可以通過批量傳輸數據或減少進程間的同步頻率，從而減少這些開銷。這個部分我已經盡量優化了，此份實驗就是優化的版本！
   - **優化 Rank-Thread 協作**：確保每個 Rank 都有均衡的工作負載，透過調整 OpenMP 動態排程來維持每個進程內部的執行緒負載平衡。這個部分我已經盡量優化了，此份實驗就是優化的版本！
   - **精簡熱點函數**：對於 `mandelbrot_avx512` 等函數，可以考慮減少 MPI 調用頻率或最小化傳輸的數據量，這樣可以提升進程間的通信效率，降低各 Rank 間的等待時間，進一步提升 CPU 資源的利用率。這個部分我已經盡量優化了，此份實驗就是優化的版本！

3. Discussion

3.1 Scalability Comparison

- **Pthread**：Pthread 版本顯示出不錯的可擴展性，特別是在較大規模的測試資料下隨著執行緒數量增加，執行時間有顯著減少。這表明 Pthread 能夠有效利用多執行緒處理大規模的工作負載。
- **Hybrid**：Hybrid 版本在不同進程和執行緒組合下，展現了比 Pthread 更強的可擴展性。隨著進程和執行緒數的增加，Hybrid 版本能夠有效分配工作量至不同的處理單元，達到更高的並行效能。

2. Synchronization and Communication Overheads

- **Pthread**：在 Pthread 中，主要的開銷來自於多執行緒之間的同步開銷。在小型數據集下，這些同步開銷顯得特別顯著，導致速度提升曲線波動。然而，隨著數據量增加，這些同步開銷被大量計算掩蓋，使得多執行緒的效能提升更加穩定。
- **Hybrid**：Hybrid 版本額外加入 MPI 通訊開銷，這在進程之間的數據交換中尤為明顯。當使用較多進程時，通訊延遲會使得效能提升趨於飽和，甚至在某些情況下出現效率下降。然而，適當的進程和執行緒搭配能有效降低這類開銷。

3. Load Balancing

- **Pthread**：從負載平衡圖來看，Pthread 版本在不同執行緒上的負載分布較為均勻，特別是在大數據集下，負載能夠更均勻地分散至各個執行緒中。這種均衡的負載分配有助於最大化多執行緒的效能。
- **Hybrid**：Hybrid 版本的負載分配相對較為複雜。因為不同進程負責不同部分的工作，且不同進程內部又含有多執行緒，因此負載分配的平衡性會受到進程和執行緒配置的影響。從圖中可以看出，某些測試資料在 Hybrid 配置下，負載分配出現輕微不均，這可能是由於進程間的工作量分配不均導致的。

4. Efficiency of Resource Utilization

- **Pthread**：Pthread 在單一處理器架構上能有效利用多核資源，適合在單一進程上處理大規模數據。當執行緒數增加時，CPU 資源的利用率相對提升。然而，當資源利用到達飽和狀態後，進一步增加執行緒數無法顯著提升效能。
- **Hybrid**：Hybrid 組合了 MPI 和 Pthread，能夠在多進程架構中同時利用多進程和多執行緒資源。這種架構在大規模計算下具備高效能和彈性，但隨著資源的增加，Hybrid 版本在跨進程的通訊同步上會產生額外開銷，因此在資源利用效率上要

根據不同的數據規模做出合適的配置選擇。

Summary

總的來說，Pthread 版本在單進程多執行緒的環境中擁有不錯的可擴展性和均勻的負載分配，而 Hybrid 版本則是在特定條件下才具有更高的可擴展性，因為在通訊同步上會有較高開銷。因此，雖然針對大規模計算，Hybrid 版本在適當配置下能達到更好的效能，但需權衡通訊開銷。

3.2 Load Balancing Comparison

1. Load Distribution

- **Pthread 版本**：圖表顯示在不同數據集下，Pthread 版本中的執行時間在各執行緒之間大致均勻分佈，這表示各執行緒的計算負載分配較為平均。然而，某些特定執行緒的執行時間略有差異，可能是因為不同計算任務間的負載不完全相同。
- **Hybrid 版本**：Hybrid 版本結合了 MPI 和 OpenMP，圖表顯示負載分配較為集中。Hybrid 版本的負載在不同 process 間的平衡性受限於 MPI 溝通開銷與 OpenMP 的負載分配機制，使得不同執行緒的執行時間差異較 Pthread 更為明顯。

2. Execution Efficiency and Resource Utilization

- **Pthread 版本**：在 Pthread 版本中，各執行緒間的計算負載分配相對均勻，且系統不涉及網路或多進程通信，因此執行效率主要受到 CPU 資源的利用影響。當數據集增大時，各執行緒的執行時間更為接近，顯示出良好的 CPU 資源利用。
- **Hybrid 版本**：在 Hybrid 版本中，MPI 用於分散計算任務，並藉由 OpenMP 加速每個 MPI process 內部的計算。然而，由於 MPI 的進程間需要進行通信，並且不同進程間的數據傳遞開銷較高，因此在負載分佈上更易出現執行緒間的負載差異。

3. Communication Overhead and Synchronization Impact

- **Pthread 版本**：Pthread 無需進行跨進程通信，僅依賴於 CPU 的共享記憶體架構來同步數據，因此通信開銷較低。這使得 Pthread 版本在負載平衡方面有更高的效率，尤其在執行緒數量少的情況下。
- **Hybrid 版本**：Hybrid 版本需考量 MPI 進程間的同步和通信開銷，尤其在不同進程間。這種開銷會導致部分進程的等待時間，進而影響整體的負載平衡性。

4. Load Balancing Effectiveness

- **Pthread 版本**：在負載分配均勻的情況下，Pthread 版本可以達到較好的負載平衡效果，尤其在進程多執行緒的架構中。
- **Hybrid 版本**：Hybrid 版本中的負載平衡效果較 Pthread 稍差，但它能夠利用多進程來提升計算資源的利用。然而，由於通信與同步開銷，在負載平衡上難以達到 Pthread 版本的水準。

Summary

Hybrid 版本藉由 MPI 和 OpenMP 的結合，使其在多進程計算資源的利用上更具彈性，但通信開銷導致的負載分配不均可能影響效率。相比之下，Pthread 版本則因為無需跨進程通信，能夠在單進程內達到較高的負載平衡性。

4. Experience & Conclusion

4.1 Key Takeaways

- **可擴展性與效能分析**：透過本次作業，我對平行計算中的可擴展性與效能分析有了更深入的理解。學會如何評估不同執行緒模型（Pthread vs. Hybrid）對執行時間、速度提升以及負載平衡的影響。這次經驗讓我能辨識出影響效能的關鍵因素，尤其是多進程環境下的通訊開銷與負載分配。
- **MPI與OpenMP的混合平行模式**：此次作業讓我體會到混合平行模式的複雜性與優勢。透過整合分散式記憶體（MPI）與共享記憶體（OpenMP）模型，能在多進程上提升計算效能，但也遇到跨進程通訊的挑戰。
- **實務優化經驗**：本次作業強調了優化策略的重要性。藉由分析優化前後的效能差異，我了解到小幅度的程式調整，如減少通訊頻率或調整負載平衡，能對可擴展性與效能產生顯著影響。

4.2 Challenges & Solutions

- **挑戰 1：跨進程的負載平衡**
  在Hybrid版本中，因為MPI通訊開銷和各執行緒工作量的不同，達成有效的跨進程負載平衡具有挑戰性。**解決方案**：我嘗試了不同的rows分配策略，並透過減少不必要的數據傳輸來優化通訊，這樣的調整減輕了同步延遲並改善了執行緒之間的負載分配。
- **挑戰 2：管理通訊開銷**
  尤其是在大型數據集的情況下，進程之間的通訊延遲影響了效能。**解決方案**：我通過減少MPI通訊頻率來優化，每個進程進行更多計算後才進行數據傳輸，以達到較好的計算與通訊平衡，這有效地提高了運行效率。
- **挑戰 3：使用剖析工具進行分析**
  初期使用剖析工具收集執行時間和速度提升的資訊相對複雜，尤其需要解釋多種度量的細節。**解決方案**：我集中分析特定度量，例如每個執行緒的執行時間和總通訊時間，從中辨識瓶頸並進行程式調整。這讓我了解到剖析工具在優化平行程式中的重要性。
- **挑戰 4：使用各種方法進行實驗進**
  我花了大量時間搜集資料並整理成有意義的數據，雖然有python跟其他的腳本可以同時處理大量資料，但一個小地方錯了實驗就要重做，因此要謹慎分析每個步驟使否存在問題。

4.3 Feedback

- **本次作業的收穫**：這次作業讓我獲得了實務上的寶貴經驗，從程式設計到效能分析，再到針對性的優化，讓我深入了解了平行程式設計的複雜性與挑戰。我學會了如何有效利用 Pthread 和 Hybrid 模式來提升程式效能，並掌握了剖析工具在優化過程中的應用。
- **未來的應用與改進**：這次作業的實作經驗，為我未來處理大型資料集和高效能計算奠定了基礎。在未來的應用中，我希望能進一步提升平行計算模型的靈活性，例如根據數據集的特性自動調整執行緒和進程的配置。此外，我認為深入學習更多的剖析工具和分析方法，能讓我更全面地理解程式的性能瓶頸，並制定更精確的優化策略。