Parallel Programming Homework 2 Mandelbrot Set

114062508 張熯尹

I. Implementation

Pthreads

A. Implementation Overview

本次 hw2a 的實作在單機上結合 **thread-level parallelism** 與 **instruction-level parallelism**，透過 **Pthreads** 與 **Intel SSE2** 指令集加速 Mandelbrot Set 的生成。

1. **Pthread**
   程式以多條 pthreads 執行，每個 thread 透過共享的任務指標 `next_row` 取得待處理的 row。為確保 thread-safe，我使用 `pthread_mutex_t` 保護此變數，在多執行緒環境下達成動態任務分配。這樣能讓較快完成工作的 thread 主動接手剩餘 rows，維持負載平衡。
2. **SSE2**
   在每個 thread 的內部運算中，使用 SSE2 的 `__m128d` 型別同時計算兩個 pixel，達成 2-lane 向量化。每次迭代同時計算兩個複數點的更新，顯著提升 CPU 浮點運算單元的利用率。

B. Pthreads Dynamic Scheduling

在這個版本中，我以 `pthread_mutex_t` 實作動態分配（dynamic scheduling）。
每個 thread 進入主迴圈時，先以互斥鎖保護共享變數：

```
pthread_mutex_lock(&next_row_mutex);
row = next_row++;
pthread_mutex_unlock(&next_row_mutex);
```

每次僅有一個 thread 能安全取得新的 row 索引並自動遞增，確保每個 row 只被處理一次。
完成一個 row 後，thread 會再次鎖定、取工，直到超出 `height` 為止。
這樣的機制相當於 OpenMP 的 `schedule(dynamic, 1)`，能在極低的同步開銷下達成良好的負載平衡。

C. SSE2 初步導入 — 雙 Pixel 並行

為了提升核心運算效率，我在每個 thread 內導入 **SSE2 向量化 (vectorization)**。
使用 `__m128d` 型別同時載入兩個 pixel 的 `(x, y)` 座標與中間變數，
讓 CPU 在單一指令下即可並行更新兩個複數點的迭代：

$$
\begin{aligned}
x' &= x^2 - y^2 + x_0 \\
y' &= 2xy + y_0
\end{aligned}
$$

在最初版本中，兩個 SIMD lane 被視為一組同步運算單元。
也就是說，只有當**兩個 pixel 都完成迭代**（達上限或 $|z|^2 \geq 4$）時，
才會載入下一組新像素。

這樣的設計雖然簡潔，且相較純 scalar 已帶來明顯加速，
但仍存在效率瓶頸——當其中一個 pixel 先收斂時，另一個 lane 仍需等待，
導致 SIMD 單元的利用率僅約 50%，產生「互等」現象。

D. SSE2（Dynamic 2-Pixel Pipeline）

為了進一步提升 SIMD 效率，我重新設計了 **動態 2-pixel 管線**。
在這個版本中：

- 每個 lane 獨立追蹤自己的迭代次數與 $|z|^2$；
- 當任一 lane 完成時（達 iters 上限或 $|z|^2 \geq 4$），
  立即將結果寫回 `image` 並從當前 row 中取出下一個 pixel 補進該 lane；
- 另一 lane 繼續運算，不需等待；
- 若兩 lane 同時完成，則同時補入兩顆新 pixels；
- 當列尾僅剩一個 pixel 時，改以 scalar 迴圈收尾。

此策略使 SIMD 單元幾乎全時滿載，
大幅提升計算密集區域的吞吐量，
同時避免了傳統 pair-wise 向量化的 idle 現象。

Hybrid (MPI + OpenMP)

A. Implementation Overview

`hw2b` 的實作採用三層平行結構：

1. **MPI 層** （inter-node）
   採用 *cyclic row distribution* 的策略，將 image rows 依 rank 平均分配。若共有 `P` 個 processes，則 rank `r` 負責 `row r, r + P, r + 2P, ...`。
2. **OpenMP 層** （intra-node）
   每個 rank 內部再以 OpenMP 進行 thread-level 平行化。最初版本使用 `#pragma omp for schedule(dynamic, 1)` 進行 row-level 動態排程
3. **SSE2 層** （instruction level）
   每個 thread 內部再以 Intel SSE2 向量化運算，一次同時計算 2 個 pixels （使用 `__m128d` ）。這是整份作業的核心優化。

B. Version 1 — MPI Cyclic Baseline (Runtime 487.20 s)

最初版本僅包含 MPI 層與 OpenMP 動態排程。
每個 rank 負責間隔的 rows （cyclic 切分），在 rank 內由 OpenMP threads 平行處理。
此版本結構簡單，執行正確，但效能瓶頸在於 Mandelbrot 影像中 row 間計算量差異極大。
當某些 threads 取到「高迭代區域（hot rows)」時，其他 threads 已空閒，造成 load imbalance。

C. Version 2 — SSE2 初步導入 (Runtime 385.04 s)

接著在每個 OpenMP thread 的計算核心導入 **SSE2 向量化**。
使用 `__m128d` 一次載入兩個 pixel 的 `(x, y)` 值，同步更新：

```
x' = x² − y² + x₀
y' = 2xy + y₀
```

這樣的設計能同時計算兩個複數點，整體效能較 scalar 版本明顯提升。不過，若其中一個 pixel 提早收斂，另一個仍需等待，SIMD 單元利用率約僅 50%。

D. Version 3 — SSE2 動態通道管理 (Runtime 327.56 s)

每個 lane 獨立追蹤自身迭代狀態，當任一 pixel 完成（達上限或 $|z|^2 \geq 4$）時，立即將結果寫回並從當前 row 補入新的像素，另一 lane 可持續運算。
若兩者同時結束則同時補入新資料，最後僅剩單一像素時以 scalar 方式收尾。

這樣的動態補位機制讓 SIMD 單元幾乎全時運作，有效提升吞吐量並消除閒置現象，整體效能較前一版本再提升約 15%。

E. Communication and I/O Strategy

所有 ranks 各自建立 `local_image` 陣列並初始化為 0。
當所有像素計算完成後，使用
`MPI_Reduce(local_image, final_image, …, MPI_SUM, 0, MPI_COMM_WORLD)`
將結果彙整至 rank 0。

由於每個 rank 僅寫入自己負責的 pixels ，其他位置維持 0，
因此 `MPI_SUM` 可正確合併整張影像。
最終僅由 rank 0 呼叫 `write_png()` 輸出結果，避免 I/O 競爭。

F. Optimization Summary

Version	主要變更	OMP 策略	SIMD	分工 策略	Runtime (s)
hw2b-487s	MPI Cyclic Baseline	`omp for dynamic,1`	—	列級 cyclic	487.20
hw2b-sse2-385s	SSE2 (靜態 2-lane)	`omp for dynamic,1`	SSE2	列級 cyclic	385.04
hw2b-sse2-dyn-328s	SSE2 動態管線	`omp for dynamic,1`	SSE2	列級 cyclic	327.56

II.Experiment & Analysis

A. Methodology

System Spec

- **環境**：NTHU Apollo HPC
- **測資**： strict24.txt `10000 -2 1 -1 1 2048 2048`

Performance Metrics

在 **pthread 版本（HW2a）** 中，我使用 C++ 的 `<chrono>` 標頭提供的 `std::chrono::steady_clock` 來量測程式的 computation time。
具體作法是：

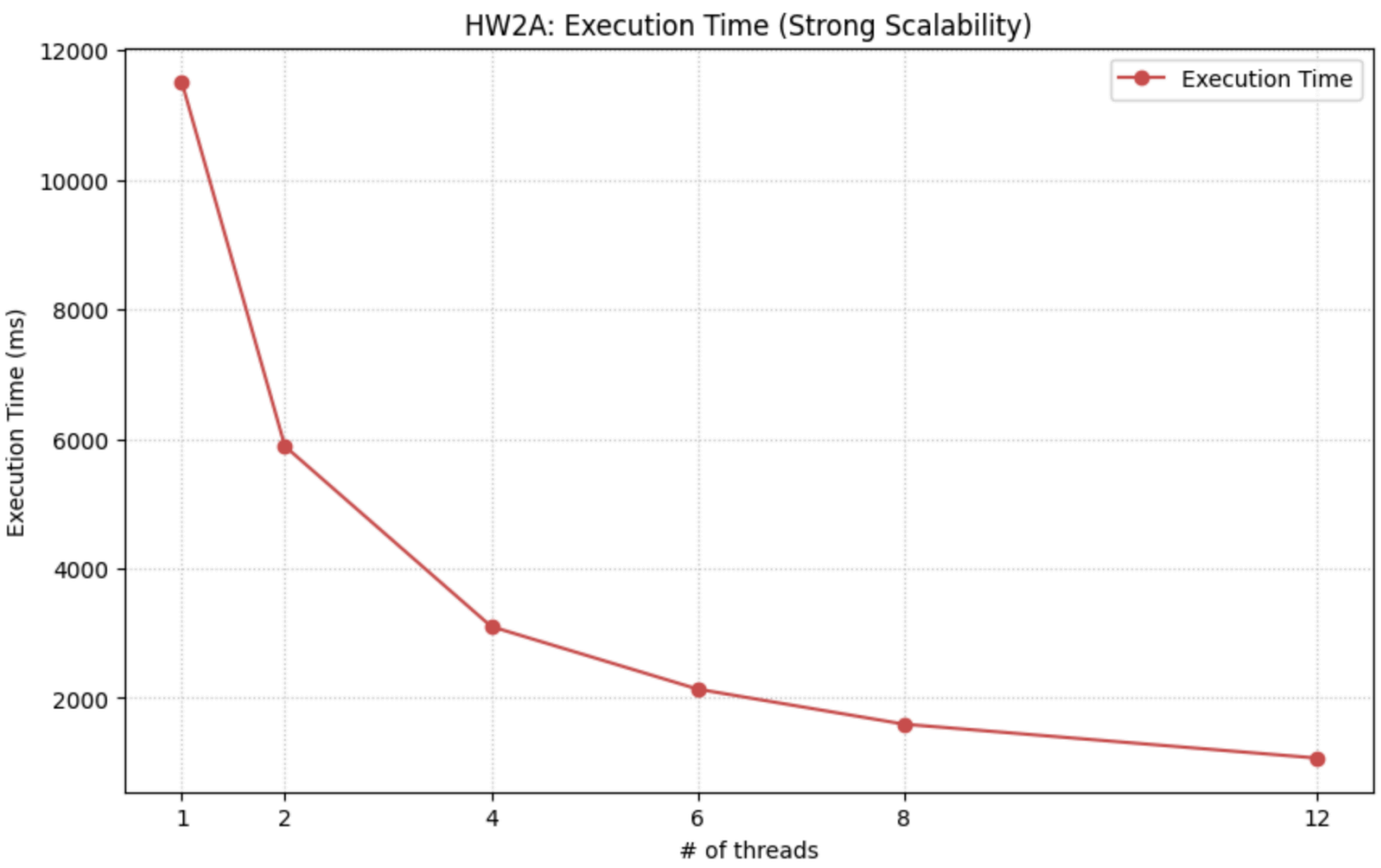
1. **計時起始點 (`start`)**：在 `main` 函式中，於 `pthread_create` 迴圈開始、即將創建第一個 worker thread 之前記錄。
2. **計時結束點 (`end`)**：在 `main` 函式中，當 `pthread_join` 迴圈完成、即所有 worker thread 均已執行完畢並返回後記錄。
最終的「Computation Time」即為 `end` 與 `start` 兩時間點的差值。我們將此差值轉換為毫秒 (ms) 單位，並以此數據作為分析 strong scalability 的基礎。

在 **Hybrid 版本 (HW2B)** 中，我們同樣使用 `std::chrono::steady_clock`

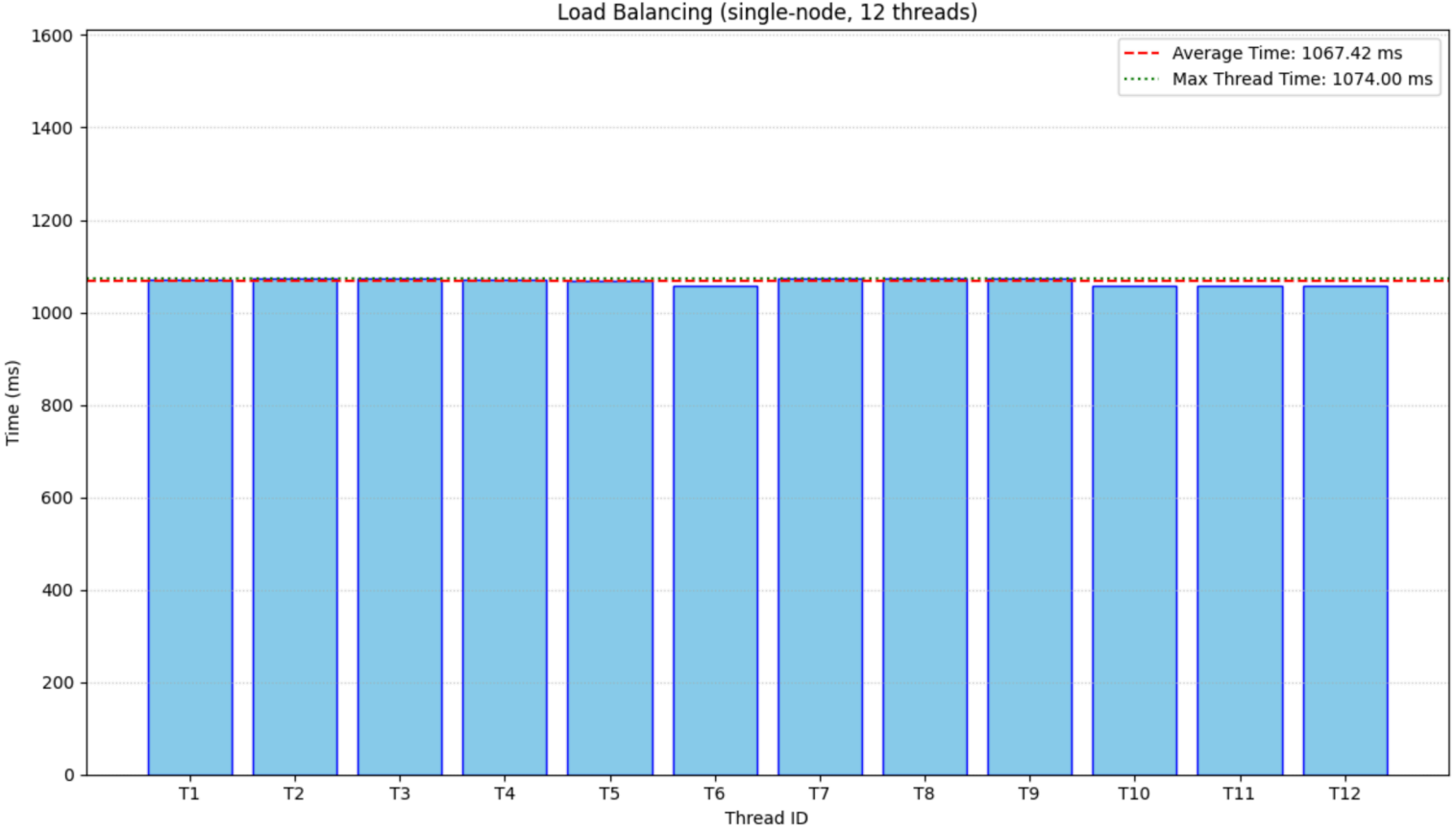
- **Computation Time (ms):** 此指標量測 MPI Process 執行 `OpenMP` 平行迴圈（`#pragma omp parallel for`）的區段。
  - **計時起始點 (`compute_start`)**：在 OMP 迴圈開始前。
  - **計時結束點 (`compute_end`)**：在 OMP 迴圈結束後。
- **Communication Time (ms):** 此指標量測 `MPI_Reduce` 操作的通訊開銷。
  - **計時起始點 (`comm_start`)**：在 `MPI_Reduce` 呼叫之前。
  - **計時結束點 (`comm_end`)**：在 `MPI_Reduce` 呼叫完成後。
- **Intra-Process Thread Time (ms):** 為了量測 `OpenMP` 內部的 load balance（即「thread 之間」的差異），我們採用了與 `HW2A` 類似的作法。在 `OpenMP` 平行區塊中，每一條 thread（共 6 條）使用獨立的計時器來記錄各自的 Busy Time。
- **Sync (Wait) Time (ms):** 為了量測 **MPI Process 之間**的負載不均（Inter-Process Load Imbalance），我們在 `MPI_Reduce` 之前插入了一個 `MPI_Barrier`。
  - **計時起始點 (`sync_start`)**：在 `MPI_Barrier` 之前。
  - **計時結束點 (`sync_end`)**：在 `MPI_Barrier` 之後。
  - 此時間代表「快的 Process 等待慢的 Process」所花費的閒置時間。

B. Plots & Discussion

Pthread


HW2A: Execution Time (Strong Scalability)
Execution Time
Execution Time (ms)
12000
10000
8000
6000
4000
2000
1
2
4
6
8
12
# of threads


HW2A: Thread Scaling Speedup (800x800, 10k Iters)
speedup
ideal speedup
speedup
12
10
8
6
4
2
0
1
2
4
6
8
12
# of threads (single node)

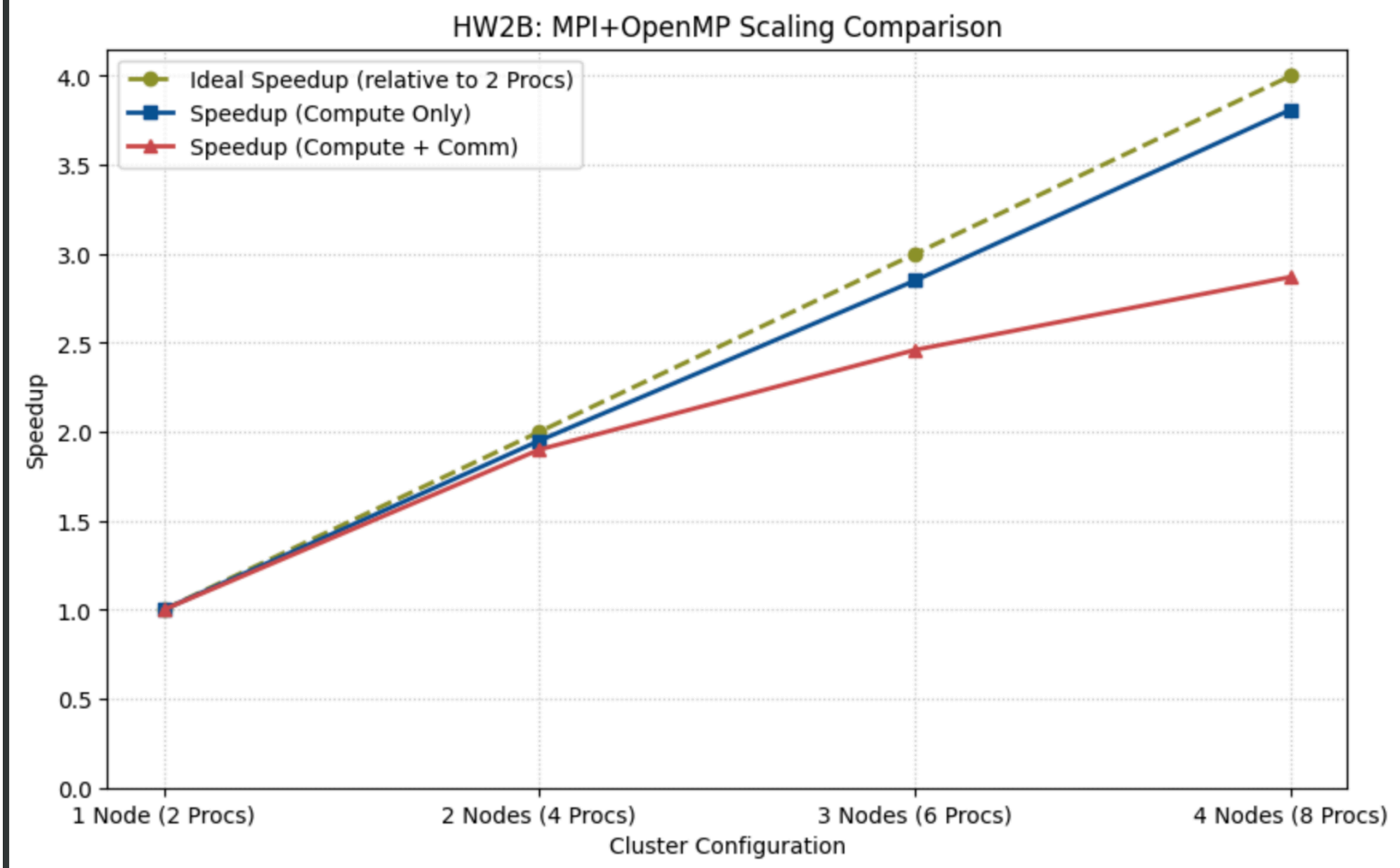


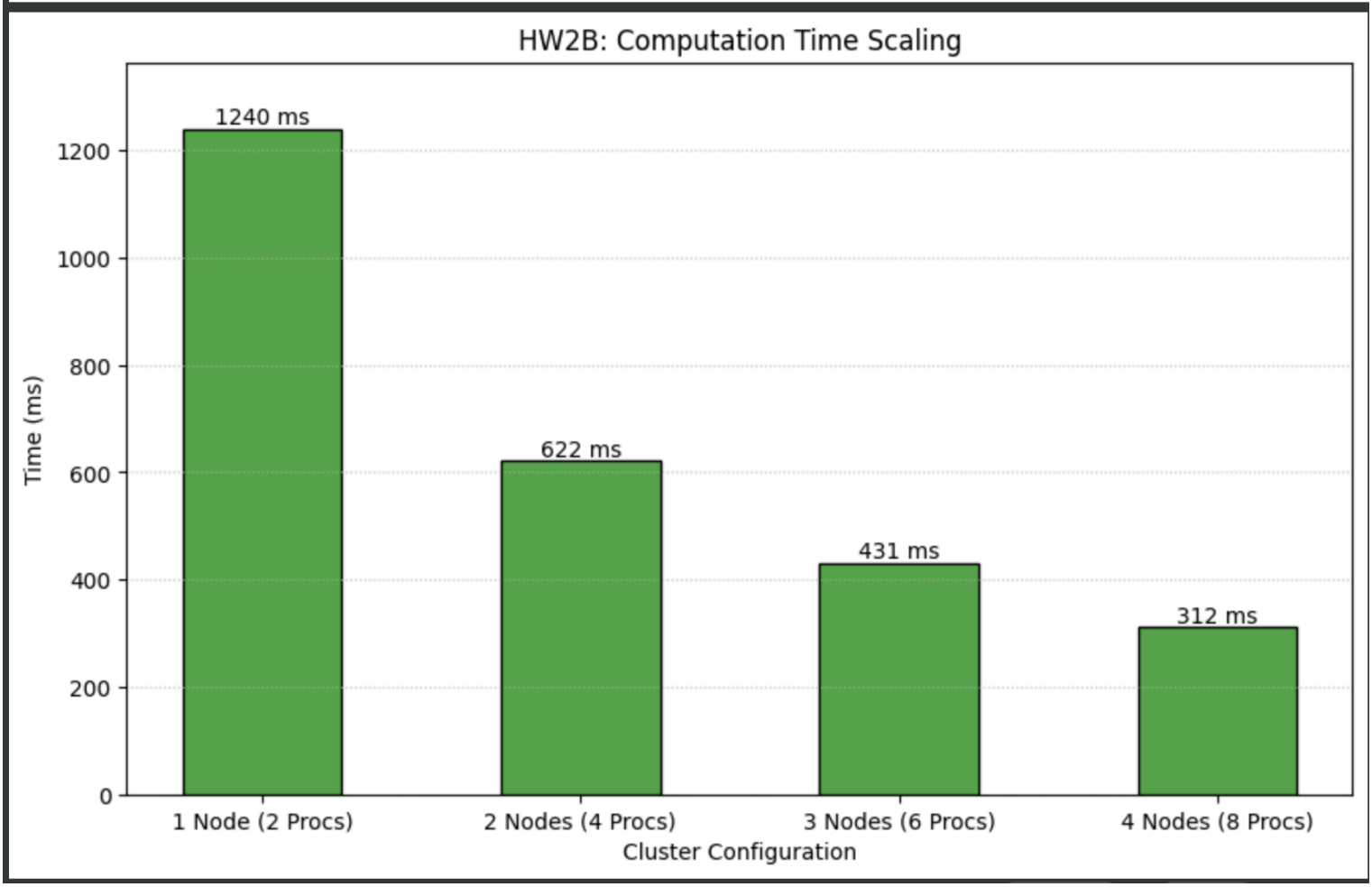
Discussion

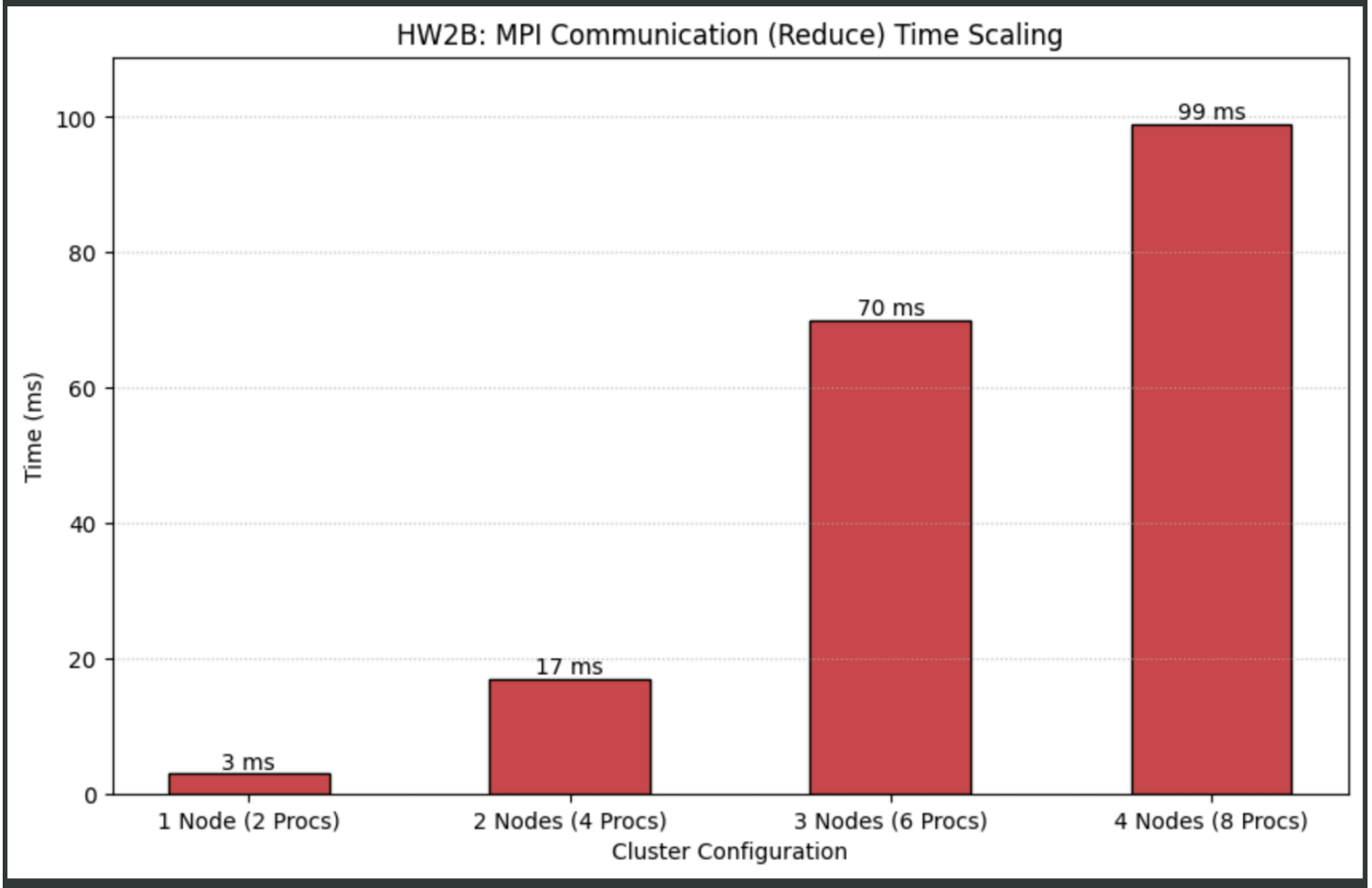
在 pthread 版本的實驗中，整體的平行化效果相當理想。從強可擴展性圖來看，隨著 thread 數量增加，執行時間幾乎呈現線性下降，12 條 thread 時可達到 10.71 倍的加速（效率約 89%）。這樣的結果主要來自 Mandelbrot Set 本身高度獨立的特性，各 row 之間沒有任何資料依賴，也幾乎沒有通訊或 I/O 的開銷，使得整個運算非常適合平行化。

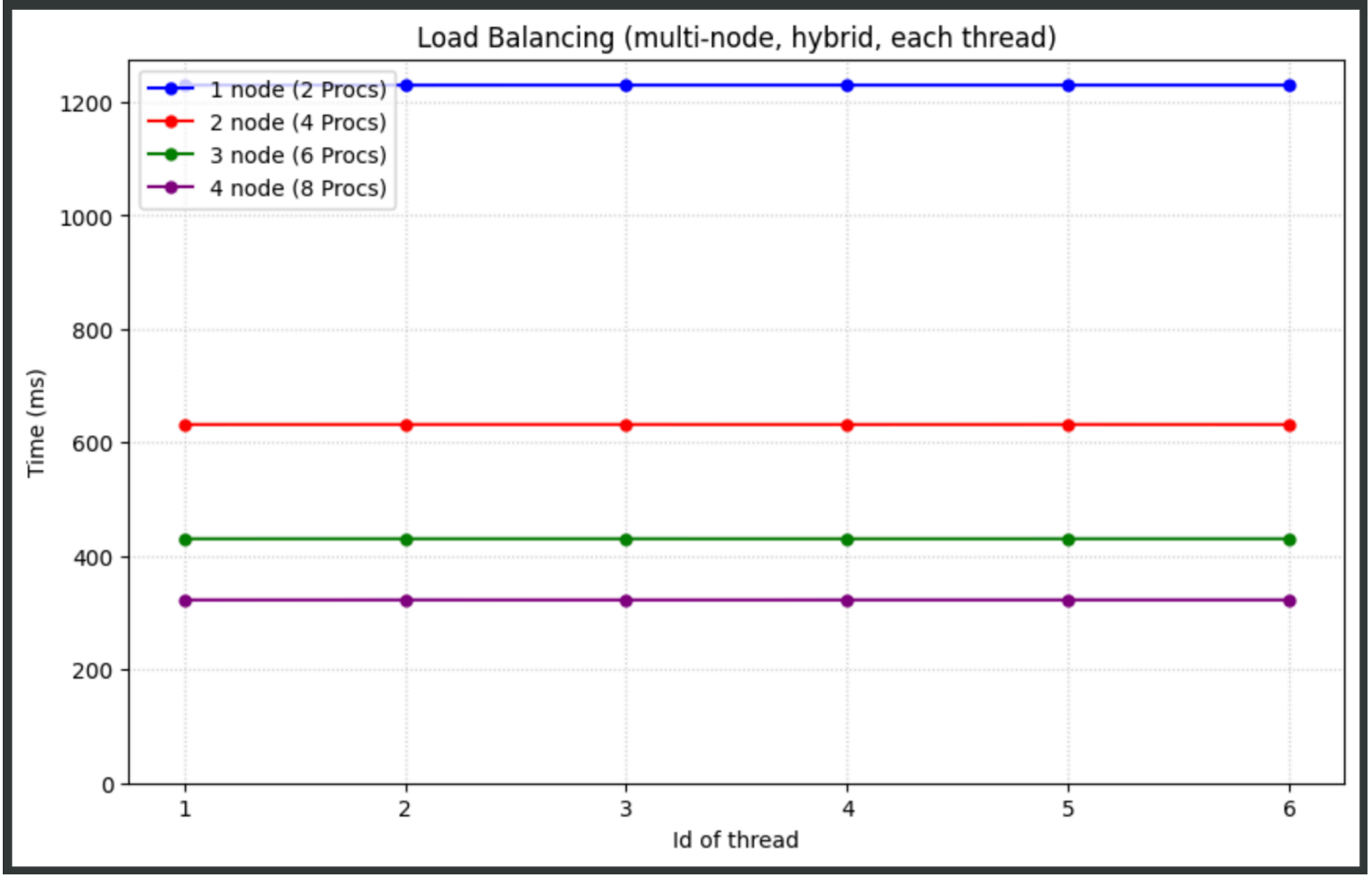
在程式設計上，我以 `pthread_mutex_t` 保護共享變數 `next_row`，讓每個 thread 能安全地從全域計數器中取得下一個待處理的 row。這樣的動態取工機制開銷極小，但能大幅改善負載分配。實驗中可觀察到，12 條 thread 的執行時間幾乎重疊，最長與最短僅相差約 17 毫秒，顯示工作分配極為平均。此設計有效避免靜態切分下常見的閒置問題，使所有 threads 幾乎全時忙碌，平行效率接近理論上限。

Hybrid


HW2B: MPI+OpenMP Scaling Comparison
Ideal Speedup (relative to 2 Procs)
Speedup (Compute Only)
Speedup (Compute + Comm)
Speedup
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
1 Node (2 Procs)
2 Nodes (4 Procs)
3 Nodes (6 Procs)
4 Nodes (8 Procs)
Cluster Configuration


HW2B: Computation Time Scaling
Time (ms)
0
200
400
600
800
1000
1200
1240 ms
622 ms
431 ms
312 ms
1 Node (2 Procs)
2 Nodes (4 Procs)
3 Nodes (6 Procs)
4 Nodes (8 Procs)
Cluster Configuration





Discussion

在 hybrid（MPI + OpenMP）版本中，整體的平行化表現相當穩定。從 scalability 圖可以看到，若僅觀察純計算時間（深藍色線），在 4 個節點（8 個 process）時相較於 2-process baseline 已達到 3.81 倍的加速，幾乎貼近理論極限，顯示 MPI 的跨節點分工與 OpenMP 的節點內平行化協調得當，能有效發揮多層次平行架構的優勢。

然而，若將通訊時間（紅色線）納入考量，加速比則下降至 2.87 倍，顯示隨著 process 數量增加，MPI 的溝通成本逐漸成為主要瓶頸，導致整體 scalability 偏離理想線。這一現象與 Amdahl's Law 相符，反映了非平行部分（如 MPI 初始化與資料整合）的限制。

在負載平衡方面，可以分成兩個層級來看：
在 thread 層級（intra-node），OpenMP 採用與 pthread 版本相同的 `schedule(dynamic, 1)` 排程策略，使各 thread 的忙碌時間幾乎一致（差異小於 0.1ms），展現出極佳的平衡性。
在 process 層級（inter-node），從 MPI_Barrier 的同步延遲（4-node 時約 8ms）可推測出節點間的分工仍算平均。不過，目前採用靜態分配方式（`for (int row = rank; row < height; row += size)`），在均勻負載的測資下運作良好，但若面對計算密度差異較大的影像，可能出現部分節點提早結束、其他節點仍在運算的情況。未來若能導入動態工作分配機制，應能進一步提升系統的穩定性與效率。

III. Experience & Conclusion

在這次作業中，我學到許多關於多層平行程式設計的實作經驗。
最初的挑戰來自於如何正確使用 **Pthreads**、**OpenMP** 以及 **MPI** 這三種不同層級的平行模型。
我特別花了不少時間去理解 **mutex** 的運作方式，
像是在多個 threads 同時存取共享變數時，如何避免 race condition、
如何在鎖的開銷與動態分配之間取得平衡。

另一個讓我印象最深的部分是 **SSE2（Single Instruction, Multiple Data）** 的實作。
這個階段常常出現一些難以預期的錯誤，例如 Wrong Answer 或程式卡住不動。
後來我發現問題主要出在 SIMD 管線的設計——
一開始兩個運算通道並不是能夠「動態補位」，導致當其中一個 pixel 提早收斂時，
另一個仍被迫等待，整體吞吐量下降。
在修正成可持續載入新資料的動態版本後，效能才真正達到預期。

如果還能繼續改進的話，我會想在 **MPI 層** 嘗試導入動態工作分配機制，
讓不同節點可以根據進度自動取得剩餘的 rows，
以進一步減少跨節點的不平衡與通訊延遲。

整體來說，這份作業讓我第一次完整體驗了從 thread-level 到 instruction-level 的多層平行化，
也讓我實際感受到當理論與實作結合、效能接近理想線性加速時的成就感。
特別是這次的 Mandelbrot Set 屬於高度可平行化的純計算任務，
在優化完成後能看到效能幾乎貼近理論極限，
這讓我更有信心面對之後更複雜的平行系統開發。