

1. Title & Identification

- **Title:** Homework 1 - Odd-Even Sort Implementation
- **Author:** 簡佩如
- **Student ID:** 112065525

2. Implementation

2.1 How do you handle an arbitrary number of input items and processes?

我透過 MPI 的 `MPI_Comm_size` 函數來獲取進程數量 `size`，並將輸入數據均分給各進程處理。每個進程處理的數量會根據總數量 `n` 和進程數量 `size` 自動計算，所以可以處理任意數量的輸入項目和進程。具體的分配方式是利用整數除法和餘數，先分配平均數量的元素，然後將餘數多出來的部分分配給前面的進程，使每個進程能夠獲得接近相等的數據來處理。這樣的分配方式能夠確保所有的數據項目都能夠被合理分配並處理。

2.2 How do you sort in your program?

排序的核心方法是使用奇偶排序 (Odd-Even Transposition Sort)，並搭配 Boost 庫中的 `Spreadsort` 演算法來對每個進程中的本地數據進行初始排序。首先，對各個進程中的本地數據進行初始的 `Spreadsort` 排序，然後利用奇偶排序方法進行進程間的交換和合併，以達成全域排序。奇偶排序的過程分成奇數和偶數階段，在每一個階段，進程會和它的相鄰進程進行數據交換和合併。若是偶數階段，奇數排名的進程會和它右邊的進程交換；若是奇數階段，則偶數排名的進程會和它左邊的進程交換。這樣的交換和合併過程能夠確保數據依次進行排序。

2.3 Other efforts you've made in your program.

1. **自訂合併函數：**不同於傳統的合併排序需要將每個進程與鄰居進行完整排序，我自訂了 `merge_arrays_max` 和 `merge_arrays_min` 函數，使得進程之間的排序複雜度從 $O(n + m)$ 降為 $O(n)$ 。這樣能有效減少合併的時間開銷。
2. **排序條件優化：**在排序過程中，加入了額外判斷條件，若左邊進程的最大值小於右邊進程的最小值，即表示該段已排序完成，無需進行排序操作。這樣的優化能避免不必要的排序，提高整體效

率。

3. **記憶體使用優化**：在合併過程中，利用 `std::swap` 函數交換指標，避免不必要的數據拷貝。這樣能有效減少記憶體佔用，同時提升程式執行效率。
4. **MPI 傳輸次數減少**：在數據交換過程中，原本透過兩次 `MPI_Sendrecv` 讓進程彼此交換數據與元素數量。經過效能分析發現 `MPI_Sendrecv` 消耗時間較多，因此調整為只傳送 `local_data`，而元素數量 `n` 則在每次進程中重新計算，減少了傳輸次數並提升了效能。
5. **預先宣告緩衝區**：將 `merged_data` 緩衝區於 `main` 函數中一次性宣告，作為各階段的合併緩衝區，避免在每次迴圈中重複進行 `malloc` 和 `free` 操作，降低了記憶體分配的負擔。
6. **使用 Boost Spreadsor 進行排序**：在進程內部原先使用 `qsort` 進行排序，後來改為使用 `boost::sort::spreadsor::spreadsor` 進行排序，能有效提升排序效率。

3. Experiment & Analysis

3.1 Methodology

SystemSpec：

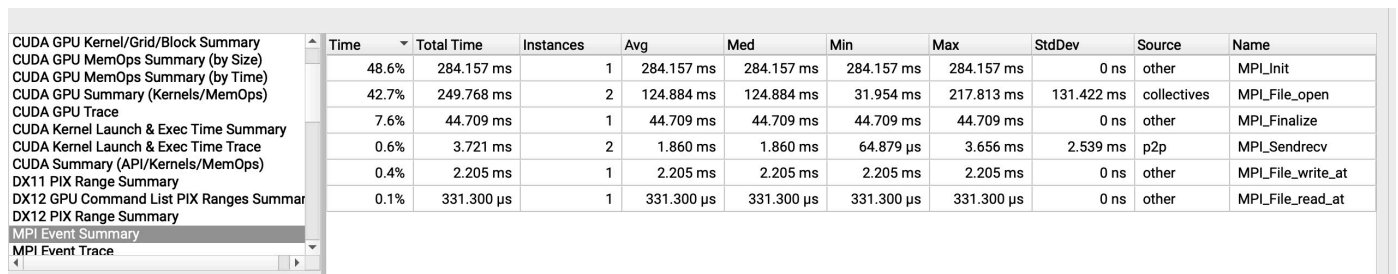
- 助教提供的 PP-Apollo 平台

Performance Metrics

在測量程式執行效能時，我透過 Nsight Systems 以及 MPI 函式進行資料收集和計算，具體方法如下：

1. 使用 Nsight Systems 進行資料收集：

針對每個執行 Rank，使用 Nsight Systems 擷取 MPI event summary，記錄了 I/O time（包含 `MPI_File_open`、`MPI_File_write_at`、`MPI_File_read_at`）以及 communication time（`MPI_Sendrecv`），我保留所有rank的這些資訊，以便後續分析，其圖表示意圖如下：



	Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Source	Name
CUDA GPU Kernel/Grid/Block Summary	48.6%	284.157 ms	1	284.157 ms	284.157 ms	284.157 ms	284.157 ms	0 ns	other	MPI_Init
CUDA GPU MemOps Summary (by Size)	42.7%	249.768 ms	2	124.884 ms	124.884 ms	31.954 ms	217.813 ms	131.422 ms	collectives	MPI_File_open
CUDA GPU MemOps Summary (by Time)	7.6%	44.709 ms	1	44.709 ms	44.709 ms	44.709 ms	44.709 ms	0 ns	other	MPI_Finalize
CUDA GPU Trace	0.6%	3.721 ms	2	1.860 ms	1.860 ms	64.879 μs	3.656 ms	2.539 ms	p2p	MPI_Sendrecv
CUDA Kernel Launch & Exec Time Summary	0.4%	2.205 ms	1	2.205 ms	2.205 ms	2.205 ms	2.205 ms	0 ns	other	MPI_File_write_at
CUDA Kernel Launch & Exec Time Trace	0.1%	331.300 μs	1	331.300 μs	331.300 μs	331.300 μs	331.300 μs	0 ns	other	MPI_File_read_at
CUDA Summary (API/Kernels/MemOps)										
DX11 PIX Range Summary										
DX12 GPU Command List PIX Ranges Summary										
DX12 PIX Range Summary										
MPI Event Summary										
MPI Event Trace										

2. 計算程式的總執行時間：

透過 `MPI_Wtime()` 函數來計算程式的總執行時間，並將該時間記錄到相應的 `.txt` 文件中。同時定義計算時間為：

computing time = 總執行時間 - (communication time + I/O time)

這樣可以得到每個測試情境下的計算時間，作為效能指標之一。

3. 腳本自動化測試流程：

我針對不同的測試資料集撰寫了相對應的腳本，以下為使用 $n = 12345$ 的範例腳本，具體步驟包括：

- 確認 `wrapper.sh` 腳本為可執行狀態，並逐一執行不同 n 值的測試，每個測試皆進行 profiling。
- 使用 `nsys stats` 指令批量導出每個 Rank 的 MPI 事件摘要 CSV 檔案。
- 執行每個測試後，將計算所得的 `MPI_Wtime()` 時間結果記錄在對應的 `.txt` 文件中，供後續分析使用。

```
# Ensure wrapper.sh is executable
chmod +x wrapper.sh

# Run 12 tests concurrently, each using wrapper.sh for profiling
for n in {1..12}; do
    srun -N1 -n$n ./wrapper.sh ./hw1 12345 testcases/14.in output/14_output_${n}.out
done

# Wait for all tests to complete
wait

# Export MPI Event Summary CSV for all ranks
for n in {1..12}; do
    output_dir="/home/pp24/pp24s092/hw1/nsys_reports_${n}_12345"
    for file in ${output_dir}/rank_*.sqlite; do
        nsys stats -r mpi_event_sum --format csv "${file}" -o "${file%.sqlite}_mpi.csv"
    done
    srun -N1 -n$n ./hw1 12345 testcases/14.in output/14_output_single_${n}.out > "output/14_output_${n}.out"
done
```

4. Python 分析與圖表生成

使用 Python 進行數據解析，主要包括資料讀取、計算與時間配置圖表生成，以便全面分析程式的通訊、I/O 和計算時間隨進程數量的變化趨勢：

• 資料讀取與處理：

從每個 rank 生成的 `mpi_event_summary.csv` 文件中讀取 MPI 事件的執行時間，並進行合併計算。具體操作包括將 `MPI_File_open`、`MPI_File_write_at`、`MPI_File_read_at` 標記為 I/O 時間，`MPI_Sendrecv` 標記為通訊時間。對於每個不同的 process 數量，計算各 Rank 的通訊與 I/O 時間的總和後取平均，以獲得該配置下的整體通訊與 I/O 時間。最終結果經轉換後以秒為單位，用於進一步分析。

- **計算與添加計算時間：**

計算時間的定義為總執行時間減去通訊和 I/O 時間的總和，並記錄為「計算時間」，合併至資料中，以便全面的時間分布分析。

- **時間配置圖與加速圖生成：**

使用 matplotlib 庫以並排圖表的方式呈現不同進程數下的時間配置和加速效果。

- **時間配置圖：**生成堆疊條形圖，將 I/O、通訊與計算時間的貢獻比例視覺化。此圖表展示了隨著進程數量增加，各時間組成的分佈變化，有助於識別性能瓶頸和優化方向。
- **加速圖：**以基準時間（單一進程的執行時間）為基準，展示隨進程數量增加的加速比，視覺化不同進程配置下的效能提升。

3.2 Plots and Analysis**

3.2.1 Experimental Method:

- Test Case Description:

Explain the test data and its sizes you've chosen.

在測試中，我選擇了四個測資 `testcases05`、`testcases14`、`testcases25` 和 `testcases34`，其資料量大小分別為 100、12345、400009 和 536869888。

以下是選擇的理由：

- 資料規模的多樣性：**這些測試案例涵蓋了從小型數據集（100）到極大型數據集（536869888）的範圍。透過不同規模的資料，我可以觀察程式在處理不同負載下的效能表現，進而了解隨著數據量增加，計算、通訊和 I/O 時間的變化趨勢。
- 極限性能測試：**選擇最大的資料集 `testcases34`（ $n = 536869888$ ）則是為了進行極限性能測試，檢驗系統在處理高負載時的瓶頸，並觀察程式在極大數據量下的穩定性和可擴展性表現，或許可以作為性能分析和未來優化的參考。

- Parallel Configurations:

Describe the number of processes used, and how nodes and cores are distributed.

對於不同大小的資料集，分別有種實驗配置：

- 在單一個 node 上，觀察從 1 到 12 個 process 的表現。
- 在固定 12 個 process 的情況下，觀察從 1 到 8 個 node 的表現。

3.2.2 Performance Measurement:

- Use a profiler (such as `perf`, `nsys`, `mpiP`) for performance analysis.
- Provide basic metrics like execution time, communication time, IO time, etc.

1. **Profiler 使用與資料導出：**

- 我主要使用 Nsight Systems 進行效能分析，並透過以下指令將分析結果匯出為 CSV 格式：

```
nsys stats -r mpi_event_sum --format csv "${file}" -o "${file%.sqlite}_mpi_event"
```

- 這樣的 CSV 格式匯出結果便於後續使用 Python 進行數據處理和分析，以更好地識別程式效能中的潛在瓶頸。

2. 分析指標：

- 分析中包含了執行時間 (execution time)、通訊時間 (communication time)、I/O 時間 (IO time) 以及計算時間 (computing time) 等指標。
- 這些指標提供了程式在不同並行配置下的效能表現，並有助於深入了解程式在不同進程與節點配置下的資源消耗，特別是通訊和 I/O 的開銷比例。

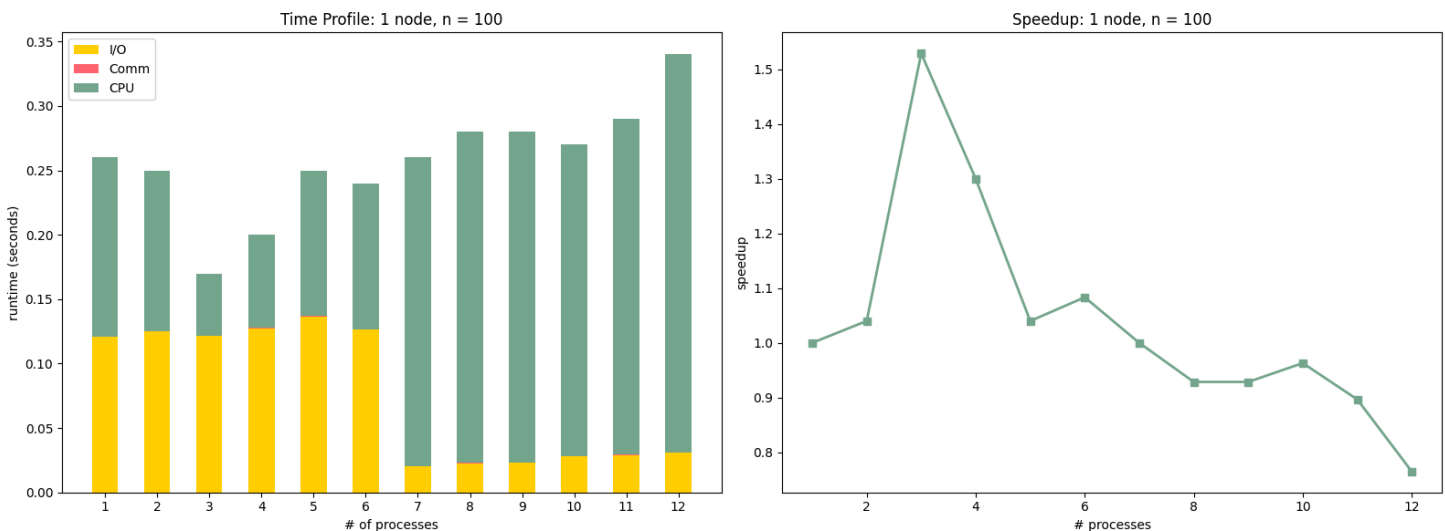
3. 資料呈現：

- 將不同配置下的指標數據以圖表形式呈現，例如每個配置的執行時間和通訊時間佔比，便於直觀比較不同情境下的效能差異。

3.2.3 Analysis of Results:

單一節點上不同資料量對 1 至 12 個進程的效能分析

(a) 在單一個 node 上，觀察從 1 到 12 個 process 的表現: n = 100



Analysis of Results :

1. 左圖：時間分佈 (Time Profile)

- 隨著 process 數量增加，總執行時間沒有顯著縮短，反而在某些情況下有所增加，這表示盲目增加 process 未必會有效提升效能。
- I/O 時間在 process 數量較少時佔了相當大的比例，隨著 process 增加，I/O 時間相對穩定或略有下降。
- 通訊時間 (Comm) 幾乎不可見，可能是因為在小資料集的情況下，跨 process 排序的次數較少，因為很快就可以排序完成，導致通訊時間不明顯。

- 計算時間隨 process 增加後顯得更為顯著，這可能是因為每個 process 還是需要執行 MPI_Comm_rank 和 MPI_Comm_size 等查詢操作的時間。由於每個 process 負責的資料量小，這些管理開銷顯得較為突出，導致計算時間比例變高。

2. 右圖：加速因子 (Speedup Factor)

- 當使用 2 至 3 個 process 時，加速效果稍有提升，但在 4 個 process 後，加速因子開始下降，並在 7 個 process 時降至 1 以下，表示效能反而變差。
- 這表明在小資料集下，增加 process 數量會引入額外的開銷，使得通訊和管理操作（例如 MPI 查詢）佔據更多的時間，導致整體效能下降。

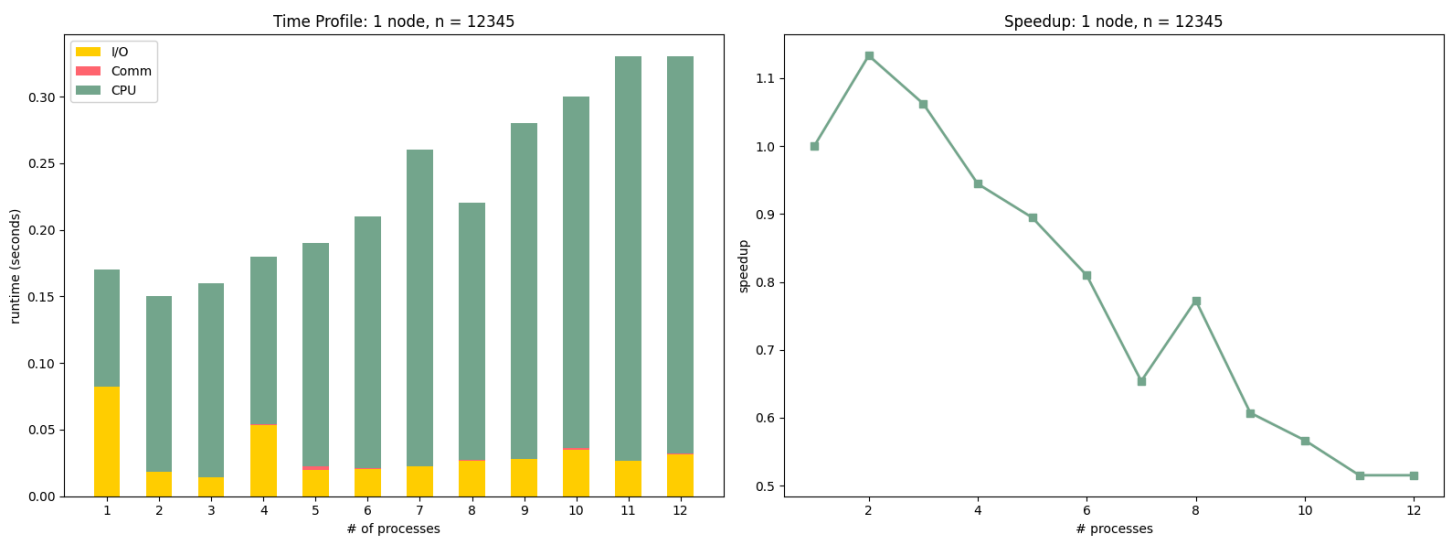
3. Identify Performance Bottlenecks

- **管理開銷佔比較高**：在小資料集下，MPI 的管理操作如 MPI_Comm_rank 和 MPI_Comm_size 對整體執行時間有顯著影響。由於資料量小，排序計算時間的比例被壓縮，而 MPI 管理開銷無法隨之減少，因此在多進程情況下顯得突出。
- **通訊需求不足**：由於資料量極小，每個 process 分得的資料量極少，process 間幾乎無需頻繁交換資料，導致通訊時間極低，但計算時間被大量的 MPI 管理開銷掩蓋。
- **I/O 開銷**：I/O 的時間在各種配置下保持穩定，說明在小資料集下，I/O 並不是主要的效能瓶頸，因為平行化 I/O 的需求沒有隨著 process 數量增加而顯著上升。

4. 分析

- 若希望提高效能，可以選擇更大的資料集來分攤 MPI 管理開銷，讓每個 process 負責更多的計算工作。這樣能降低管理操作的相對影響，使得並行化效果更加顯著。
- 在小資料集下不建議增加過多的 process，因為效能瓶頸主要來自 MPI 管理開銷和較少的通訊需求，增加 process 數量無法帶來實質的效能提升。在這樣的情況下，建議將 process 控制在 2 至 4 個範圍內，避免因管理開銷過高而導致效能下降。

(b) 在單一個 node 上，觀察從 1 到 12 個: $n = 12345$



Analysis of Results :

1. 時間分佈 (Time Profile)

- 對於 $n = 12345$ 而言，隨著 process 數量增加，總執行時間並未顯著縮短，並在超過 4 個 process 後反而比只有 1 個 process 高。這種情況與 $n = 100$ 相似，在兩種資料量下，增加 process 並未帶來明顯的效能提升。這是因為隨著 process 數量增加，管理與同步成本變高，抵消了平行計算的效益。
- 與 $n = 100$ 不同的是， $n = 12345$ 的 I/O 時間佔比較小且穩定。
- 通訊時間 (Comm) 在 $n = 12345$ 的情況下略微顯現，尤其是 process 超過 5 個時，可以看到通訊需求增加的趨勢。相比之下， $n = 100$ 幾乎不需要跨 process 通訊，因此通訊時間不明顯。

2. 加速因子 (Speedup Factor)

- 在 $n = 12345$ 時，2 個 process 能夠提升加速因子至約 1.1，顯示了較佳的平行效率。然而，隨著 process 增加，加速因子開始下降。這與 $n = 100$ 的情況類似，兩者在 process 數量超過 4 個後，加速效果都比只有 1 個 process 時差。

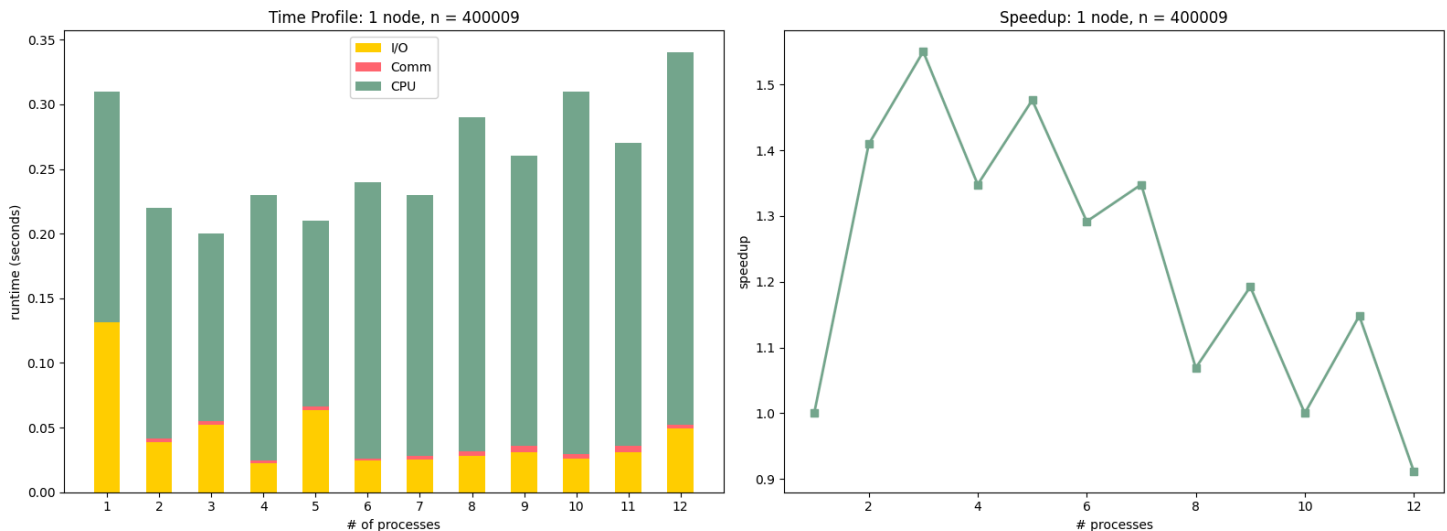
3. Identify Performance Bottlenecks

- **通訊需求**：隨著資料量增大， $n = 12345$ 下的跨 process 通訊需求開始顯現，特別是超過 5 個 process 時，通訊時間逐漸上升，這顯示出較大資料集在並行環境中通訊的重要性。相較於此， $n = 100$ 的通訊需求非常低，通訊開銷在小資料集下幾乎可以忽略。
- **I/O 開銷**：在 $n = 12345$ 的情況下，I/O 的佔比跟 $n = 100$ 差不多，沒有因為資料量變大而變大，所以不算是 bottlenecks。

4. 分析

- $n = 12345$ 在 2 到 3 個 process 下能夠獲得一定的加速效果(比只有一個 process 快)，這與 $n = 100$ 沒有明顯不同，可能因為資料還不夠大。
- 在 $n = 12345$ 下，通訊需求開始顯現，而 $n = 100$ 則幾乎無通訊需求。這意味著隨著資料量增大，通訊時間可能逐漸成為平行計算中不可忽視的因素。
- 對於小資料量 $n = 12345$ ，2-3 個 process 仍能帶來一定的效能提升，但更高的 process 數量可能導致管理和通訊開銷增多，反而拖累效能。

(c) 在單一個 node 上，觀察從 1 到 12 個: $n = 400009$



Analysis of Results :

1. 時間分佈 (Time Profile)

- 在 $n = 400009$ 大資料集下，隨著 process 數量增加，總執行時間在 2 至 3 個 process 時顯著減少，說明平行計算在此資料量下帶來了更明顯的效能提升。然而，在超過 4 個 process 後，執行時間開始增加，說明進一步增加 process 的效果有限。
- I/O 時間在多數 process 配置下持續降低、相對穩定但仍占小比例，跟前面 $n = 100$ 以及 $n = 12345$ 時差不多。
- 通訊時間 (Comm) 在 $n = 400009$ 的情況下相對穩定並且略顯增加，這表明跨 process 通訊在大資料集下具有一定影響力。相比之下， $n = 100$ 幾乎無通訊需求。

2. 加速因子 (Speedup Factor)

- 在 $n = 400009$ 的情況下，加速因子在 3 個 process 時達到接近 1.5 的峰值，顯示出此配置下平行效果最佳。
- 當 process 超過 8 個後，幾乎等於沒加速，甚至在有所下降，表明增加 process 數量並未繼續帶來效能提升，可能是因為通訊和管理開銷開始超過計算的增益。但相比 $n = 100$ 和 $n = 12345$ ， $n = 400009$ 的加速因子，這邊在多 process 有比較顯著的效能提升。

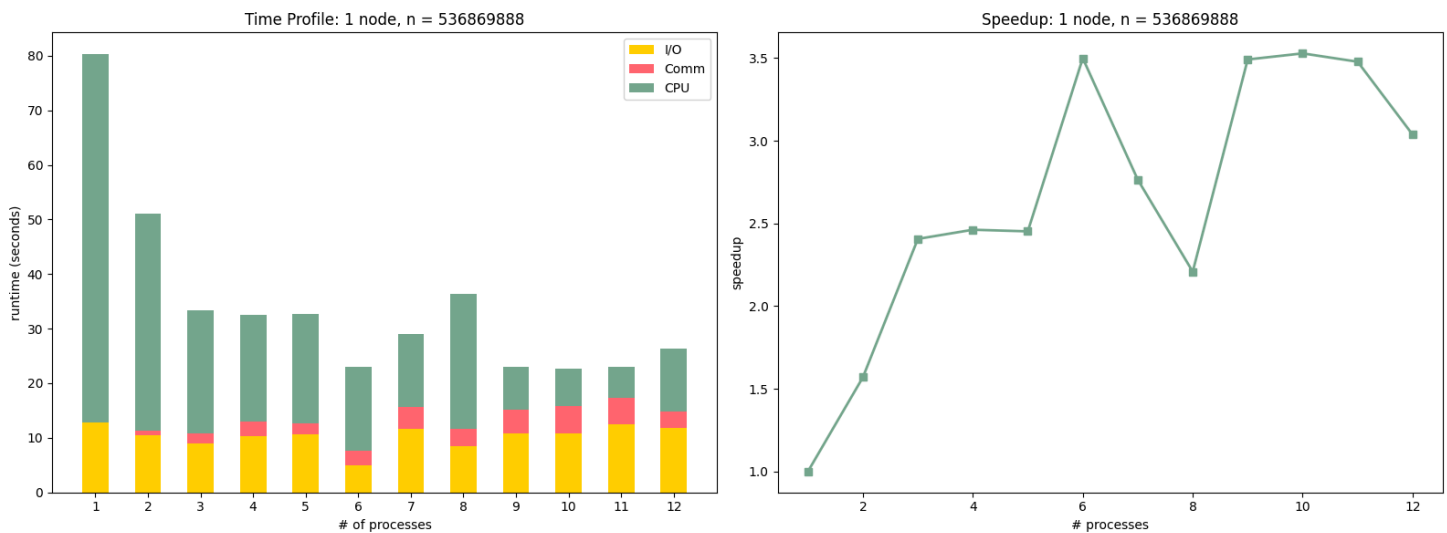
3. Identify Performance Bottlenecks

- 管理與通訊開銷的比例：**隨著資料量增加，MPI 的管理開銷有被分攤的趨勢，這在 $n = 400009$ 下比較明顯，使得更多的 process 可以有效分擔計算負擔，因此在 process 數量增加後效能有所提升。
- 通訊需求的增強：** $n = 400009$ 下，通訊需求更為顯著，但仍在可接受範圍內。相比之下， $n = 100$ 幾乎無通訊需求，而 $n = 12345$ 則僅在超過 5 個 process 時出現少量通訊需求。這表明隨資料量增加，跨 process 通訊開始成為影響效能的重要因素之一。
- I/O 開銷：**在 $n = 400009$ 下，I/O 開銷到目前為止都跟前幾次小型資料都差不多。

4. 分析

- **效能提升的範圍**：對於 $n = 400009$ ，並行計算在 3 個 process 下能夠顯著提升效能，在 6 至 8 個 process 配置下仍然比只有 1 個 process 時好。而 $n = 100$ 和 $n = 12345$ 的情況中，增加 process 數量的效果明顯遞減。
- **最佳 process 配置**：在 $n = 400009$ 下，3 個 process 是較為理想的配置範圍，可以在不增加過多管理和通訊開銷的情況下提升效能。而在較小資料量（如 $n = 100$ 和 $n = 12345$ ）下，建議 process 數量控制在 2 至 4 個。
- 隨資料量進一步增大，通訊開銷可能成為更大的瓶頸。若資料量再增加，可以考慮更有效的通訊模式，看起來還可以再嘗試更大的資料集！

(d) 在單一個 node 上，觀察從 1 到 12 個: $n = 536869888$



Analysis of Results :

1. 時間分佈 (Time Profile)

- 對於這個極大資料集，隨著 process 數量增加，執行時間顯著下降，顯示了平行計算所帶來的效能提升都比之前的實驗都好，說明在大資料量下平行計算更能發揮其優勢！
- I/O 和通訊的時間也有比前幾次實驗的資料多，但是佔比一樣穩定，隨著 process 增加，I/O 分攤效果顯著，I/O 的佔比大致相同。
- 通訊時間 (Comm) 在此資料集下隨 process 增加而變得顯著，但仍然維持在可以接受的範圍內。雖然在極大資料集下通訊需求增加，但整體效能依然受益於平行處理的優勢。

2. 加速因子 (Speedup Factor)

- 在 $n = 536869888$ 的情況下，加速因子隨 process 增加顯著提升，尤其在 6 個 process 時達到峰值，接近 3.5 倍的加速效果。這表明在大資料量下，平行效能大幅提升，且增加 process 數量能有效減少執行時間。
- 當 process 超過 6 個後，加速因子有所波動，但仍能維持在接近 2 至 3 之間，顯示在極大資料量下，即使增加 process 數量，效能依然穩定，這與較小資料集形成鮮明對比。在 $n =$

100 和 $n = 12345$ 時，加速因子迅速下降，而在 $n = 400009$ 時雖然能保持穩定，但並未顯示出如此大幅的提升。

3. Identify Performance Bottlenecks

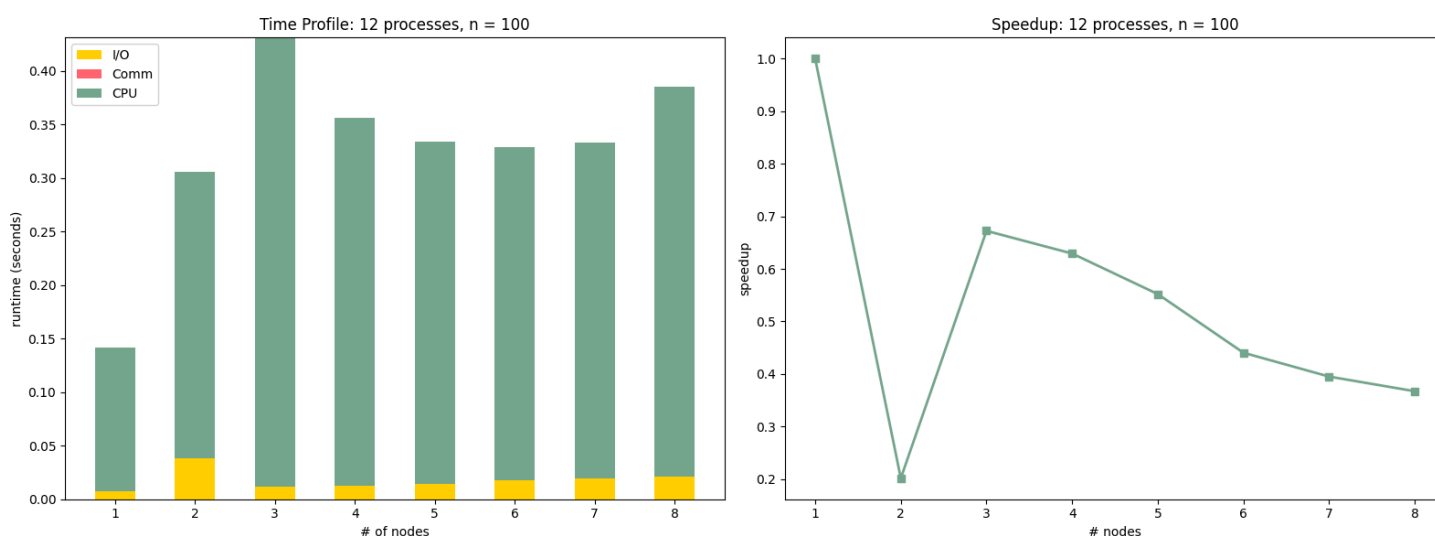
- **計算與管理開銷的分攤**：在這麼大的資料集下，MPI 管理開銷相對於計算開銷變得極小，因此可以輕鬆分攤這些管理開銷。相比之下，在 $n = 100$ 和 $n = 12345$ 下，管理開銷比例較大，而在 $n = 400009$ 時也有所減少，但不及此極大資料集的分攤效果。
- **通訊需求的影響**：對於 $n = 536869888$ ，通訊需求開始顯著增加，可能會成為主要瓶頸。對比在小資料集 $n = 100$ 和 $n = 12345$ 中，通訊需求極低甚至可以忽略，而在中等資料集 $n = 400009$ 中，通訊需求顯現但未成為瓶頸。
- **I/O 開銷**：在極大資料集下，I/O 開銷在不同數量 process 的情況下都差不多，顯示出平行計算在極大資料集中的優勢。

4. 分析

- **效能最佳範圍**：在 $n = 536869888$ 的情況下，2 至 7 個 process 下能獲得顯著的效能提升。
- **與較小資料集的對比**：相比 $n = 100$ 、 $n = 12345$ 和 $n = 400009$ 的情況，極大資料集下平行計算的加速效果最為顯著。
- 在更大規模的計算中，可以考慮更多 process 來分攤開銷，或使用更優化的通訊模式以進一步提升效能。

固定 12 個 process 不同資料量對 1 至 8 個 node 的效能分析

(a) 12 個 process 在不同 node 下的表現： $n = 100$



Analysis of Results :

1. **Time Profile**：在 1 個節點的情況下，運行時間最短。當節點增加到 2 或更多時，運行時間增加。
2. **Speedup Analysis**：速度提升呈現負增長，尤其是在 2 節點的情況下加速比降低至接近 0.2。

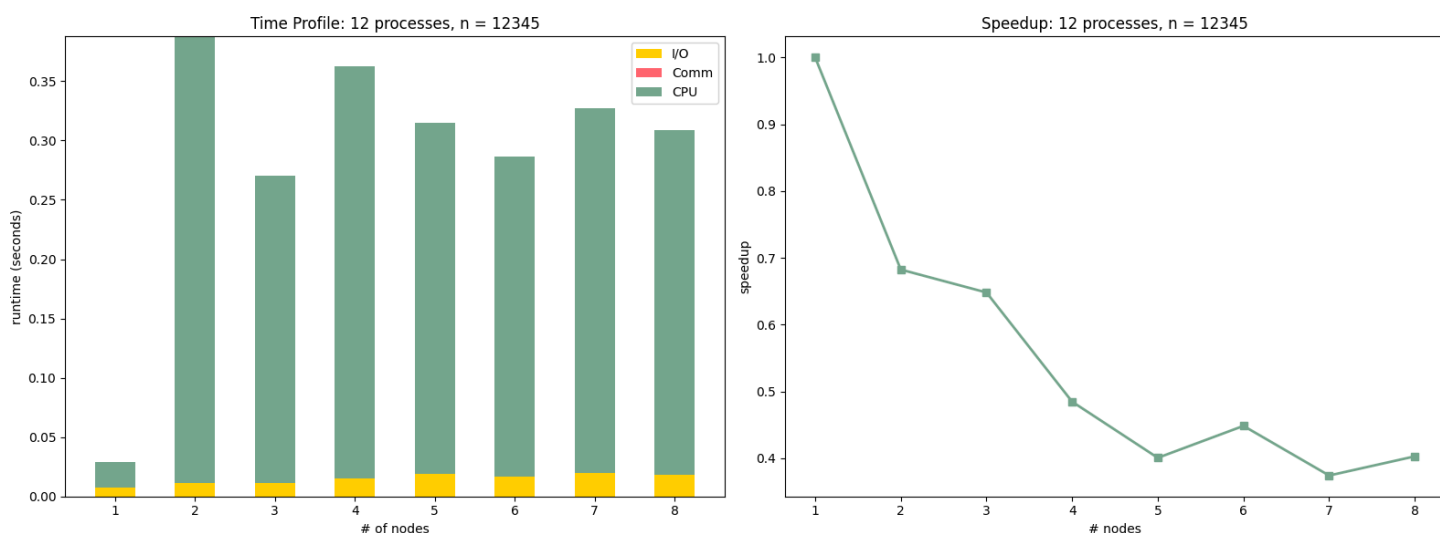
3. Identify Performance Bottlenecks :

- **主要瓶頸**：跨節點間通訊開銷過大。由於資料量小，計算時間非常短，而多節點之間的通訊成本相對較高，導致加速比不僅沒有提高，反而下降。
- **解決方法**：在這樣的小資料量下，應避免多節點部署，或者採用更高效的通訊方式，甚至可以考慮單節點處理，以減少通訊的負面影響。

4. 分析

- 在小資料量情況下，通訊開銷顯著影響性能，多節點部署反而降低了整體效能。由於通訊時間遠高於計算時間，導致加速比急劇下降，適合單節點運行來避免不必要的通訊成本。

(b) 12 個 process 在不同 node 下的表現： n = 12345



Analysis of Results :

1. **Time Profile**：增加資料量至 12345 後，1 節點依然具有最佳性能。
2. **Speedup Analysis**：隨著節點數增加，加速比急劇下降，在節點數 3 之後，速度基本維持在低於 0.5 的水平，說明跨節點的通訊開銷在這一資料量下仍然是主要瓶頸。

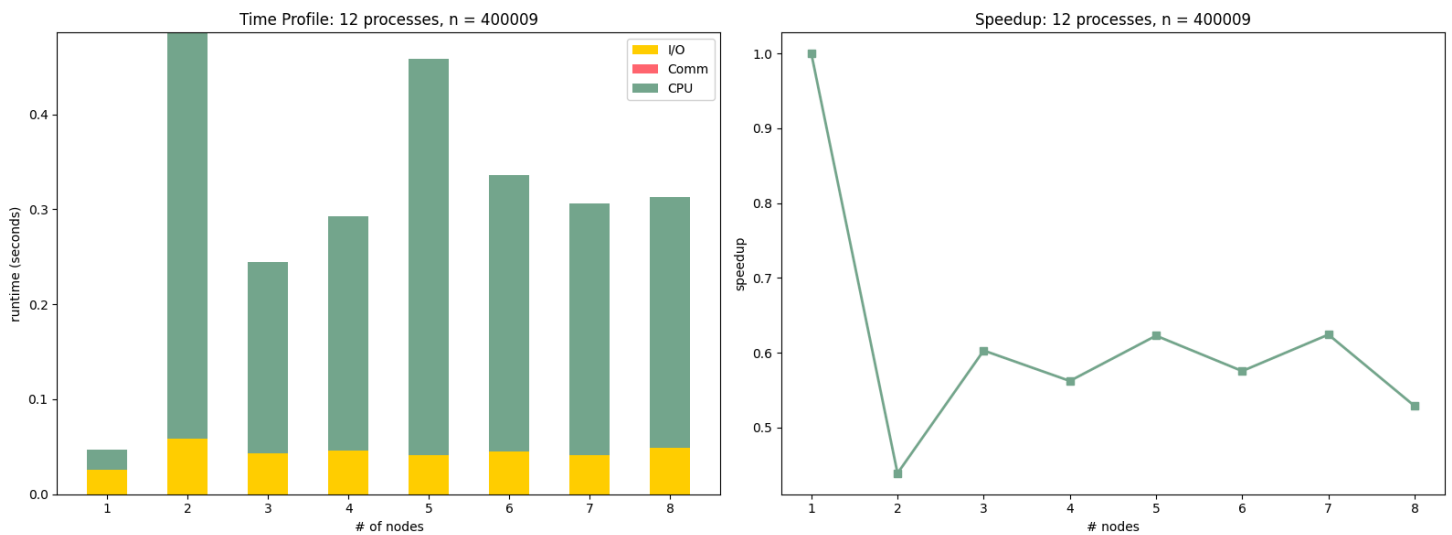
3. Identify Performance Bottlenecks :

- 和 n = 100 時差不多。
- **主要瓶頸**：通訊開銷過大。由於資料量小，計算時間非常短，而多節點之間的通訊成本相對較高，導致加速比不僅沒有提高，反而下降。
- **解決方法**：在這樣的小資料量下，應避免多節點部署，或者採用更高效的通訊方式，甚至可以考慮單節點處理，以減少通訊的負面影響。

4. 分析

- 資料量增加至 12345 時，性能仍然受到通訊開銷的限制。多節點的加速比持續降低，顯示資料量不足以有效發揮平行計算的優勢。在這樣的資料量下保持較少的節點數，避免過多的通訊負擔。

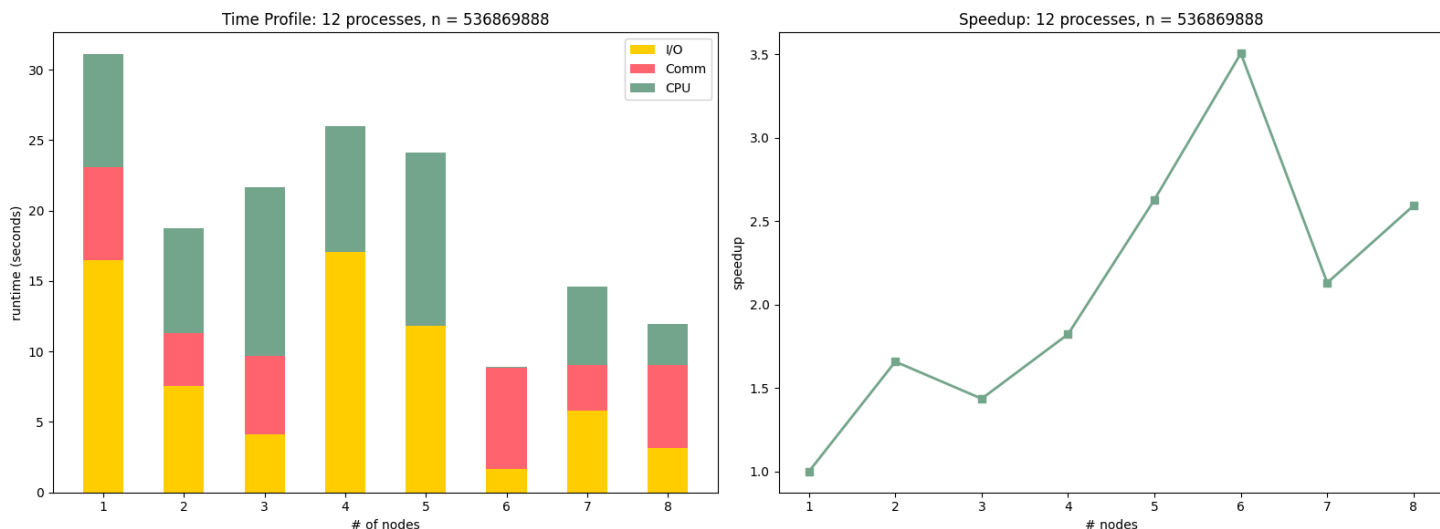
(c) 12 個 process 在不同 node 下的表現：n = 400009



Analysis of Results :

- Time Profile**：當資料量達到 400009 時，跨節點的通訊開銷仍舊無法被有效分攤，因此超過兩個節點都會變慢。
- Speedup Analysis**：跟前面幾次實驗有差不多的結論：跨節點通訊開銷仍舊是成為主要限制因素。
- Identify Performance Bottlenecks**：
 - **主要瓶頸**：通訊開銷。儘管資料量增加帶來一定的計算負載，但通訊開銷還是導致擴展效果有限。
 - **解決方法**：在這樣的資料量下，應避免多節點部署，或者採用更高效的通訊方式，甚至可以考慮單節點處理，以減少通訊的負面影響。
- 分析**
 - 中等資料量情況下，情況跟小資料差不多，還是不要有太多節點比較好。

(d) 12 個 process 在不同 node 下的表現：n = 536869888



Analysis of Results :

- Time Profile**：在這樣大規模的資料量下，1 節點的計算時間最長，隨著節點數增加，IO 和通訊的開銷變得更加明顯，但總運行時間隨節點數的增加有所減少。
- Speedup Analysis**：加速比在 6 個節點時達到最大值，約為 3.5，這顯示此時的負載均衡和計算效率最佳。然而，節點增加到 7 和 8 時，加速比有所下降，這可能是由於過多的通訊和同步導致效率下降。
- Identify Performance Bottlenecks**：
 - 主要瓶頸**：計算和通訊均占有重要比例。在增加節點時，可以有效地分擔計算負載，相比之前的都有所改善。這邊的瓶頸已經不再只是跨節點的通訊開銷，還有 I/O 的開銷。
 - 解決方法**：可嘗試減少通訊次數，例如將部分局部計算延遲到所有計算完成後再做一次同步。
- 分析**：
 - 在大資料量情況下，增加節點有效地提升了計算效率，尤其在 6 個節點時加速效果最佳，達到加速比約 3.5。但節點增加到 7 或 8 後，加速比下降。適當選擇節點數並優化通訊模式可以達到更好的擴展性和平衡效能。

結論

從分析結果可以看出，在處理不同大小的資料量時，節點數量和通訊模式對於性能的影響是顯著的。對於小資料量，應減少節點以避免過多的通訊開銷；對於大資料量，則應在節點和通訊開銷之間尋找最佳平衡點。在優化方面，可以針對不同場景選擇合適的通訊模式和 IO 策略來進一步提升平行效率。

4. Optimization Strategies

Based on the analysis results, propose potential optimization strategies.

If optimizations have been implemented, provide a comparison of performance before and after the enhancements.

1. 小資料量 (100 和 12345)

- **主要瓶頸**：不論是跨節點還是跨進程，通訊開銷過大，導致加速比隨節點數增加反而下降。
- **優化策略**：
 - **減少節點數量**：在小資料量下，應避免使用多個節點，以降低通訊的負擔。保持在 1 至 2 個節點範圍內可減少通訊成本。

2. 中等資料量 (400009)

- **主要瓶頸**：跨節點跟跨進程的通訊開銷依然限制擴展效果。
- **優化策略**：
 - **非阻塞通訊**：使用非阻塞的 MPI 通訊方式來減少通訊等待，進一步優化計算與通訊的重疊。
 - **已實施優化**：
 - **自訂合併函數**：不同於傳統的合併排序，使用 `merge_arrays_max` 和 `merge_arrays_min` 函數來減少合併的時間複雜度，讓每個進程都有可以使用的函數，而不用都等待 `merge_arrays`。
 - **減少 MPI 傳輸次數**：通過減少 `MPI_Sendrecv` 的次數，只傳輸本地數據而不是每次都傳送元素數量，顯著減少通訊開銷。
 - 優化後使用 `hw1-judge` 快了 20 秒左右。

3. 大資料量 (536869888)

- **主要瓶頸**：通訊和I/O的的開銷。
- **優化策略**：
 - **適當選擇節點數量**：根據測試結果，6 個節點時性能最佳。繼續增加節點會導致通訊開銷過高，因此在大資料量情況下應限制節點數。
 - **已實施優化**：
 - **預先宣告緩衝區**：在 `main` 函數中一次性宣告合併緩衝區，避免每次合併操作重複進行內存分配，顯著降低內存管理的開銷。
 - **使用 Boost Spreadsort 排序**：原先使用 `qsort`，後來改為 `boost::sort::spreadsort::spreadsort`。
 - 優化後使用 `hw1-judge` 快了 10 秒左右。

總結

根據不同資料量大小，需針對性地調整節點數和通訊策略：

- 中小型資料量下應避免多節點和進程以降低通訊負擔。
- 大資料量可增加進程或節點，但需在通訊模式和負載平衡上進行更深入的優化，以獲得最佳效能。

5. Discussion

1. Compare I/O, CPU, Network Performance

- **I/O :**
 - I/O 的運行時間比例在小資料量 ($n=100$ 和 $n=12345$) 下相對較小，而隨著資料量的增加 (尤其是 $n=536869888$)，I/O 的占比顯著上升。
 - 在大資料量下，I/O 成為了一個重要的性能瓶頸，尤其是在多節點情況下，由於需要頻繁地進行資料讀寫，I/O 的延遲影響了整體性能。
- **CPU :**
 - 在所有資料量的測試中，CPU 的運行時間占比相對穩定，在中小型資料量的情況下，CPU 占用大部分計算時間。
 - 在 $n=536869888$ 時，隨著節點增加，CPU 占比不再是最大的，並不是主要的性能瓶頸。
- **通訊 (Network Performance) :**
 - 小資料量 ($n=100$ 和 $n=12345$) 下，進程之間不需要太頻繁進行資料交換，所以 `MPI_Sendrecv` 不是主要瓶頸，反而是執行 `MPI_Comm_rank` 和 `MPI_Comm_size` 等查詢操作的這些管理開銷顯得較為突出，導致計算時間比例變高。
 - 大資料量下，跨節點的管理開銷降低，因為隨著計算負載的增加，`MPI_Sendrecv` 開始成為主要的瓶頸。
- **瓶頸辨識 :**
 - 在小資料量情況下，跨節點的管理開銷是主要瓶頸，因為計算負載不足以抵消多節點之間的通訊開銷。
 - 在大資料量情況下，I/O 和通訊 (`MPI_Sendrecv`) 是主要的瓶頸，尤其在高節點數的情況下，I/O 的頻繁操作和通訊延遲顯著影響性能。
- **優化建議 :**
 - **減少通訊次數 :** 在小資料量時，考慮使用較少的節點和進程，甚至不需要開平行。資料量大的時候盡量降低 `MPI_Sendrecv` 的次數。
 - **合併 I/O 操作 :** 在大資料量下，考慮合併多個 I/O 操作以減少頻繁的讀寫，或者使用更高效的文件系統。

2. Compare Scalability

- **Scalability Analysis :**
 - 從各組實驗結果的加速比圖中可以看出，程式的擴展性在不同資料量下表現不同：
 - **中小型資料量 (n=100, 12345, n=400009) :** 擴展性非常有限，隨著節點數增加，加速比反而下降，這表明程式並沒有很好的擴展性。
 - **大資料量 (n=536869888) :** 在 6 個節點時，程式的擴展性表現最好，加速比達到 3.5，顯示在此時的負載均衡效果良好，但對於更多節點，隨著通訊和 I/O 開銷的增加，加速比有所下降。
- **Does the Program Scale Well? :**
 - 在小資料量下，程式並不能很好地擴展，因為通訊成本超過了計算的增益。
 - 在大資料量情況下，程式在一定範圍內能達到不錯的擴展性，但隨著節點過多，通訊和 I/O 成為主要瓶頸，但擴展性也還是不錯整體趨勢是更優化的。
- **Achieving Better Scalability :**
 - **負載平衡 :** 應確保每個節點的計算負載均衡分配，減少因負載不均導致的某些節點閒置或過載。
 - **減少通訊和同步 :** 採用非阻塞通訊方式或者減少同步次數，進一步優化計算與通訊重疊，提高效率。
 - **提升 I/O 效率 :** 通過更高效的文件系統和 IO 合併策略來降低大資料量下 I/O 對於擴展的限制。

總結

通過分析不同資料量下的性能表現，可以看出程式的擴展性在大資料量情況下相對較好，但還是受限於通訊和 I/O 的開銷，不能盲目的無限增加。針對不同大小的資料，策略性的地減少通訊、優化 I/O，以及更有效的負載平衡策略，是提升程式擴展性的關鍵。

6. Experiences & Conclusion

在這次作業中，我深入了解了平行程式設計，特別是使用 MPI 實作平行排序演算法的複雜性。這次經驗不僅讓我學習了奇偶排序的理論基礎，還讓我理解了如何在實際環境中進行效能優化。

1. 學到的內容 :

- 實作平行奇偶排序讓我更清楚地理解了進程間通訊和資料同步的概念。學會了奇偶階段的通訊模式，這幫助我更深入了解平行排序的運作原理。
- 使用 Nsight Systems 和其他剖析工具，讓我了解到效能指標（如計算時間、通訊時間和 I/O 時間）在識別效能瓶頸中的重要性。能夠視覺化並解讀效能數據，讓我能夠根據數據進行優化決策。

2. 遇到的挑戰：

- **實驗耗時且錯誤代價高：**由於實驗過程需要大量的運算時間，一旦某些設定或邏輯出錯，就可能導致整個實驗結果不正確，必須重新執行所有步驟。此外，我在完成實驗後總覺得還有優化空間，想讓實驗結果更加完善，因此反覆進行調整，進一步增加了時間成本。
- **平行程式撰寫快速，但優化耗時：**撰寫基本的平行程式相對容易，但要達到更高的效能和更佳的擴展性，卻花費了大量時間進行思考和測試。如何減少通訊開銷、提升資源利用率，以及找到最佳的參數配置，這些都需要不斷嘗試不同的優化方法並觀察效能變化，這使得優化過程比撰寫程式本身更具挑戰性。

3. 反饋與建議：

- 第一次學到如何用平行程式來加速程式，從一開始的300多秒加速到94秒，非常有成就感。