

Homework 1 Odd–Even Sort Report

B114062508 張燦尹

Implementation

資料分配策略

在平行程式裡，第一步是「把資料平均分給每個 process」。如果資料量 N 剛好能被 process 數 P 整除，那麼每個 process 負責 N/P 筆資料即可。但現實通常不會那麼剛好，所以我的做法是把多出來的餘數分配給前幾個 process。這樣一來，每個 process 分到的工作量差距不會超過 1。

另一個需要處理的情況是「process 太多」。假設我們只要處理 100 筆資料，卻開了 200 個 process，後面一半的 process 根本就沒有資料。如果還讓它們參與排序，只會增加不必要的通訊和同步。因此我在程式裡用 `MPI_Comm_split` 把這些 $\text{rank} \geq N$ 的 process 剔除，直接退出，避免資源浪費。

最後，每個 process 必須知道「自己要從哪裡開始讀檔」。為此我計算了 offset，讓每個 process 可以直接用 `MPI_File_read_at` 去讀取屬於自己的區段，並在最後用 `MPI_File_write_at` 把排序好的結果寫回。這樣避免了集中到 rank 0 再分發的瓶頸，I/O 的效率更高。

平行 I/O

在讀取與寫入階段，我採用了 **MPI-IO 的平行存取**，讓每個 process 都直接讀寫自己負責的資料區段。這種方式避免了集中式 I/O 的瓶頸，也能隨著 process 數量的增加，讓總體的讀寫時間下降。不過，這並不是沒有代價的。當 process 數量過多時，所有人同時存取磁碟反而會產生激烈的競爭，使得 I/O 效率下降。這也是為什麼在實驗結果中，我們會看到 I/O 時間在少量 process 時下降，但在多核情況下又被拉高的原因

Local Sort

每個 process 在獲得自己的區段資料之後，首先會進行本地排序。在這裡，我測試了多種排序方式。最初使用的是 `std::sort`，它在大部分情況下表現不錯。但後來我發現，

Boost 提供的 `spreadsort` 對於浮點數排序特別高效，因此我將其作為預設選項。從效能實驗來看，`spreadsort` 在大資料下比 `std::sort` 或 `std::stable_sort` 快了許多，這個優勢在平行化後仍然相當明顯。

Odd–Even Sort

每一輪排序分為兩個相位：**Even phase** 與 **Odd phase**。在 Even phase 裡，所有偶數編號的 process 會與右邊的鄰居交換資料；而在 Odd phase 裡，則輪到奇數編號的 process 與右鄰交換。這樣交錯執行，就能確保整個排序過程持續推進。

為了降低不必要的通訊開銷，我在每個相位開始前，並不會直接整段交換資料，而是先交換彼此的邊界值（最小值與最大值），來去判斷是否有重疊或是合併的需求。如果邊界比較後發現彼此的資料範圍已經完全分開，那麼這一輪的兩個 process 就不需要傳輸完整的資料，能省下大量的時間。

當確定需要合併時，才進行完整的交換。合併的方式並不是直接整合，而是透過 rank 較小的 process 保留「較小的一半」元素，rank 較大的 process 保留「較大的一半」。這樣做能保證合併後兩個分段仍然保持排序正確，並且符合全域的順序要求。

每一輪相位結束後，所有 process 會透過一次 MPI_Allreduce 確認這一輪是否有發生交換。如果整輪都沒有任何資料被合併或修改，就代表整個全域的資料已經完全有序，可以提前結束迴圈。這個 early-stop 機制讓程式能夠在資料分布已經有序的情況下快速收斂，而不必硬跑滿所有迴圈次數。

Experiment & Analysis

Methodology

System Spec

- 環境：NTHU Apollo HPC (Intel Xeon, 256 GB RAM, Lustre FS, InfiniBand)。
- 測資：第 35 筆 testcase (536,869,888 筆浮點數)，確保運行時間足夠觀察 scalability。

Performance Metrics

我透過 `MPI_Wtime()` 量測各區段的時間，並在 root (rank 0) 使用 `MPI_Reduce(..., MPI_MAX, ...)` 取每個區段的最大值，反映整體執行時間 (makespan)，因為最慢的 process 會決定程式完成時間。具體定義如下：

- **I/O**

- `max_read`：每個 process 讀取自己區段資料 (`MPI_File_read_at`) 的耗時。
- `max_write`：每個 process 寫回輸出 (`MPI_File_write_at`) 的耗時。
- `I0 = max_read + max_write`。

- **CPU**

- `max_sort`：本地排序 `spreadsor::spreadsor` 的時間。
- `max_merge`：Odd-Even Sort 階段進行跨 rank 合併的時間。
- `CPU = max_sort + max_merge`。

- **Communication**

- `max_comm_p`：交換邊界值 (min/max) 進行探測 (probe) 的時間。
- `max_comm_d`：確定需要交換後，進行整段資料傳輸的時間。
- `Comm = max_comm_p + max_comm_d`。

- **Total**

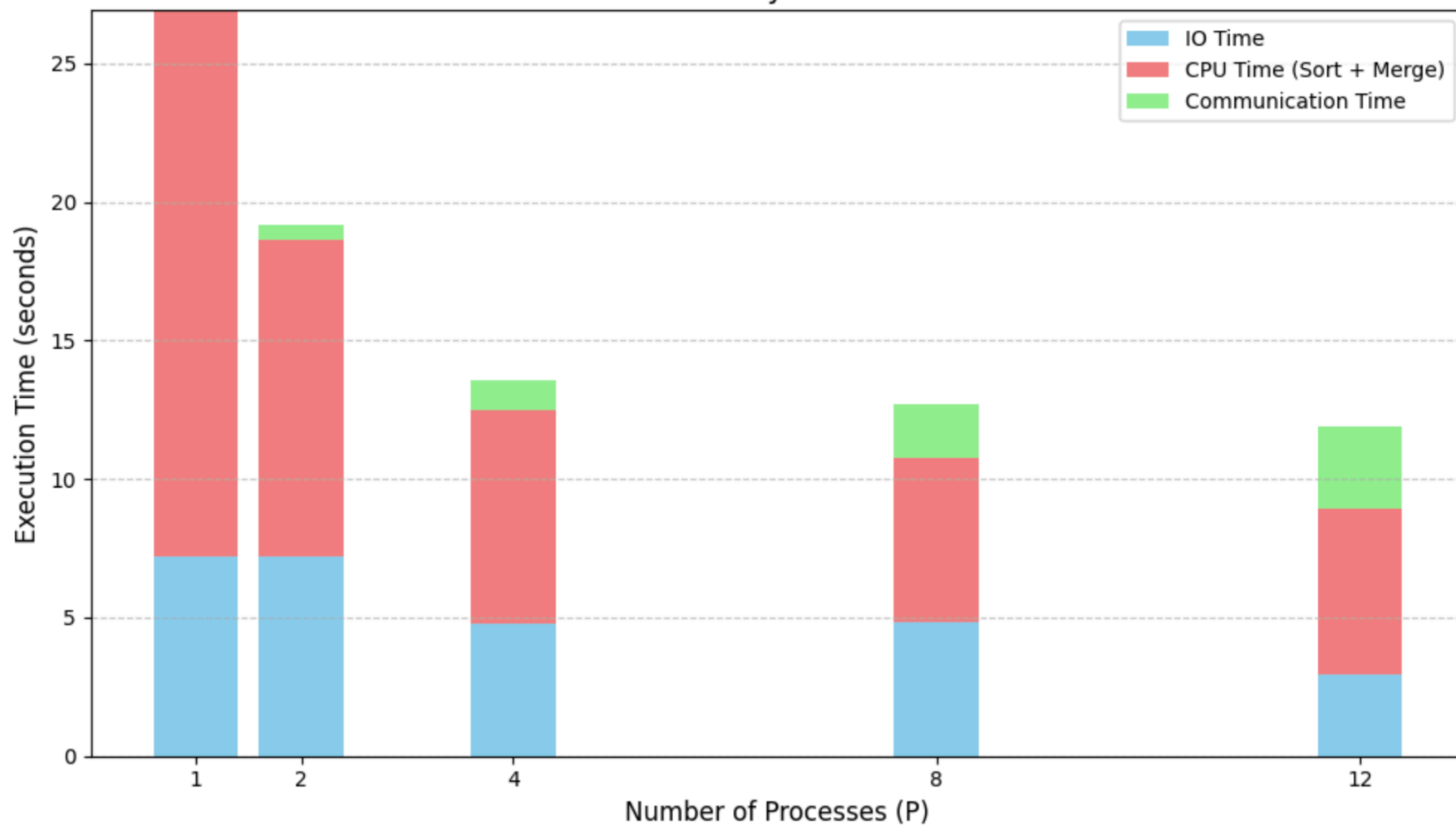
- 整體程式執行時間：`max_total`。
-

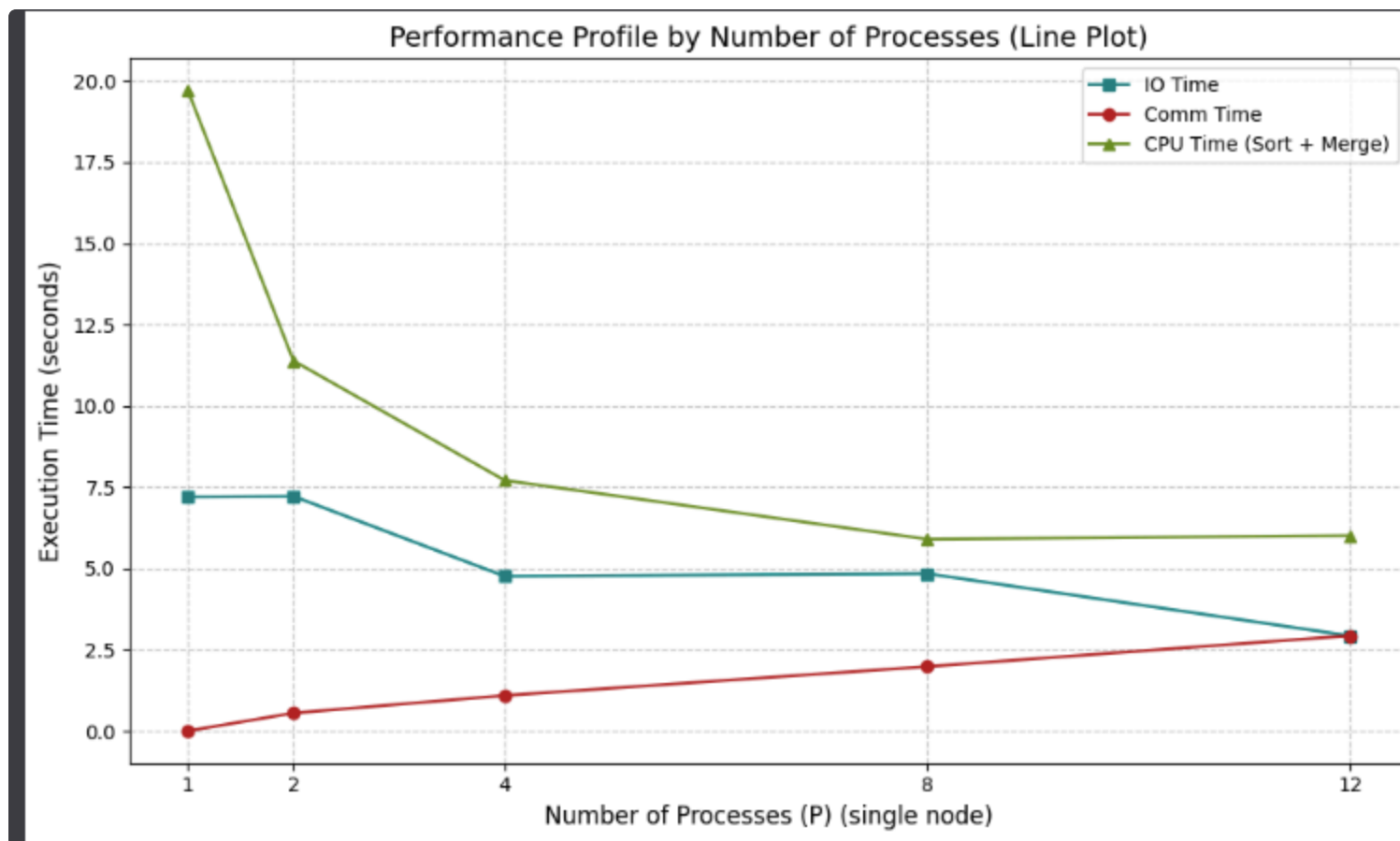
Plots: Speedup Factor & Profile

Time Profile

SINGLE NODE

Performance Profile by Number of Processes



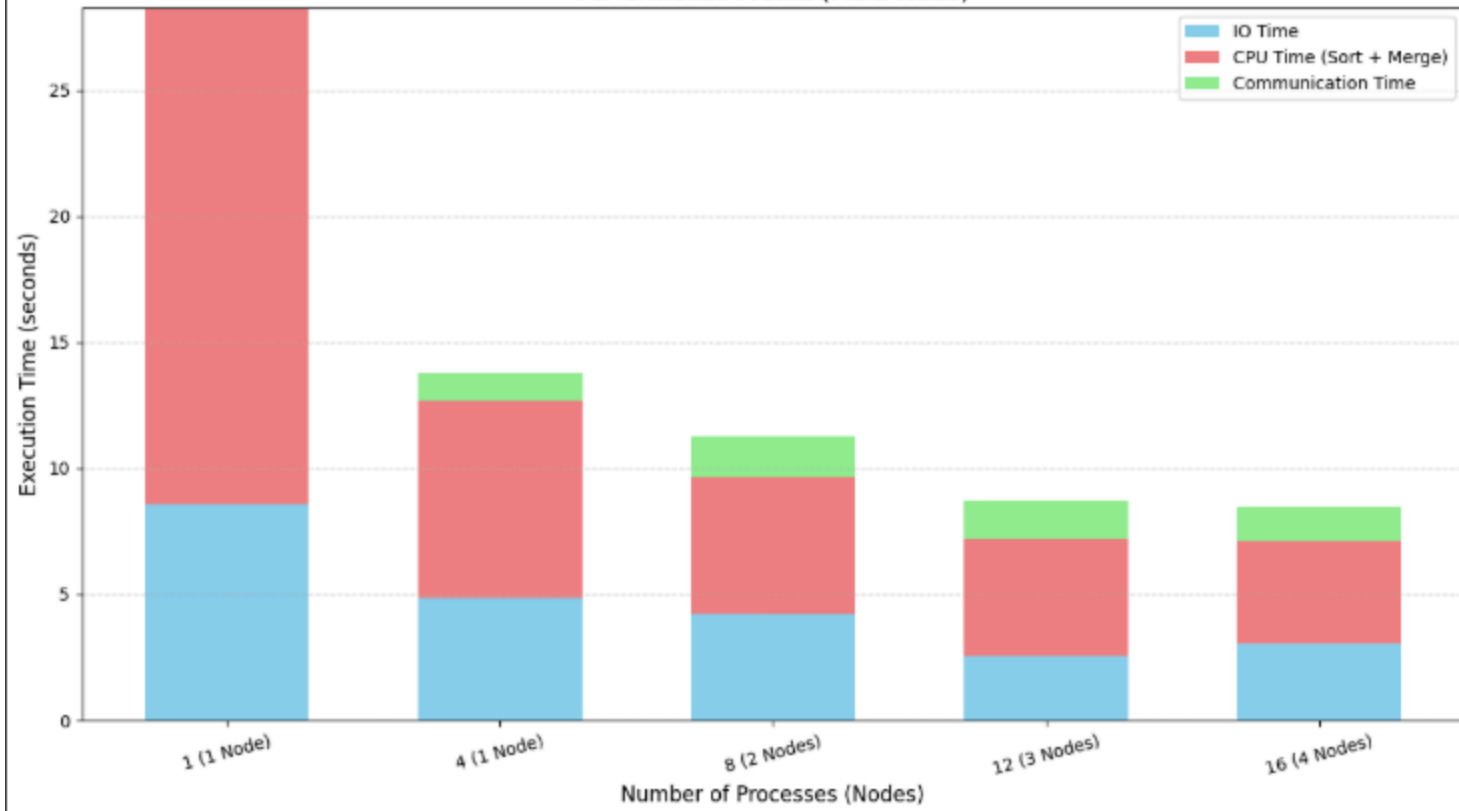


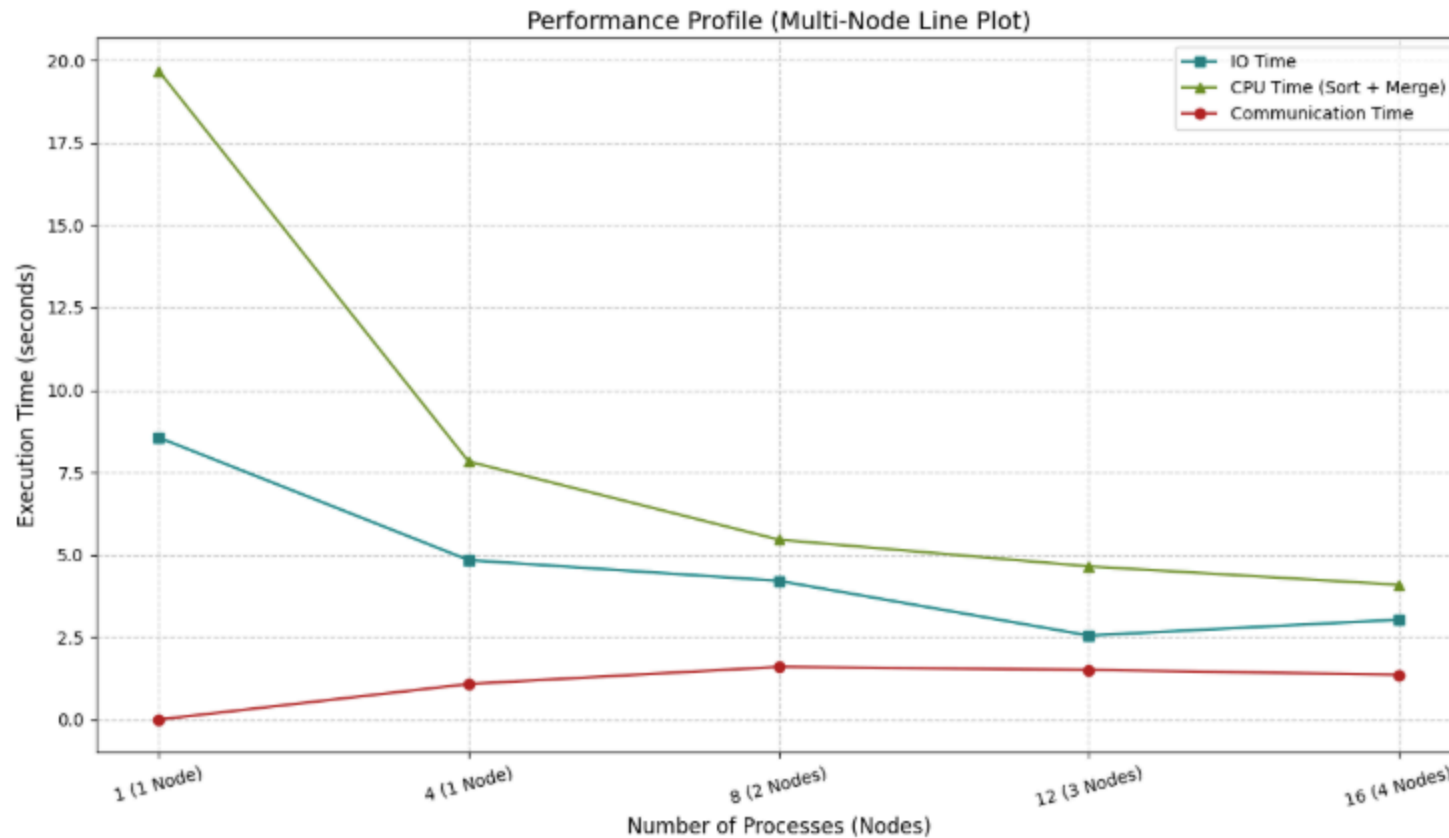
所有實驗皆在單節點環境執行，並逐次增加 process 數量（1、2、4、8、12）。在單節點下，CPU 時間隨著 processes 增加快速下降，因為隨著 process 數目增加每個 process 所需處理的資料量就成反比的減少，因此可以減少 CPU time。IO time 也有隨著 process 數量變多而遞減，原因是 process 可以平行的做 IO 所以使得 IO time 減少。Communication time 一開始會隨著 processes 數增加而成長，並且在實驗中呈現持續上升的趨勢。這是因為 odd-even sort 本質上需要鄰居間的頻繁交換，當 processes 越多，整體需要的交換次數也隨之增加，即使每次交換的資料量不大，通訊延遲與同步

開銷仍會累積成為主要的瓶頸。這也是為什麼在 8 與 12 個 processes 的情況下，雖然 CPU 計算時間已經接近極小值，但總執行時間的改善幅度有限。

MULTI NODE

Performance Profile (Multi-Node)





在多節點環境下，我的實驗分別使用了 1、4、8、12、16 個 processes，並固定每個節點配置 4 個 processes（即 `-mincpus=4`）。這樣的設計可以逐步增加節點數量，從單節點一路擴展到四個節點，藉此觀察跨節點後效能變化的趨勢。

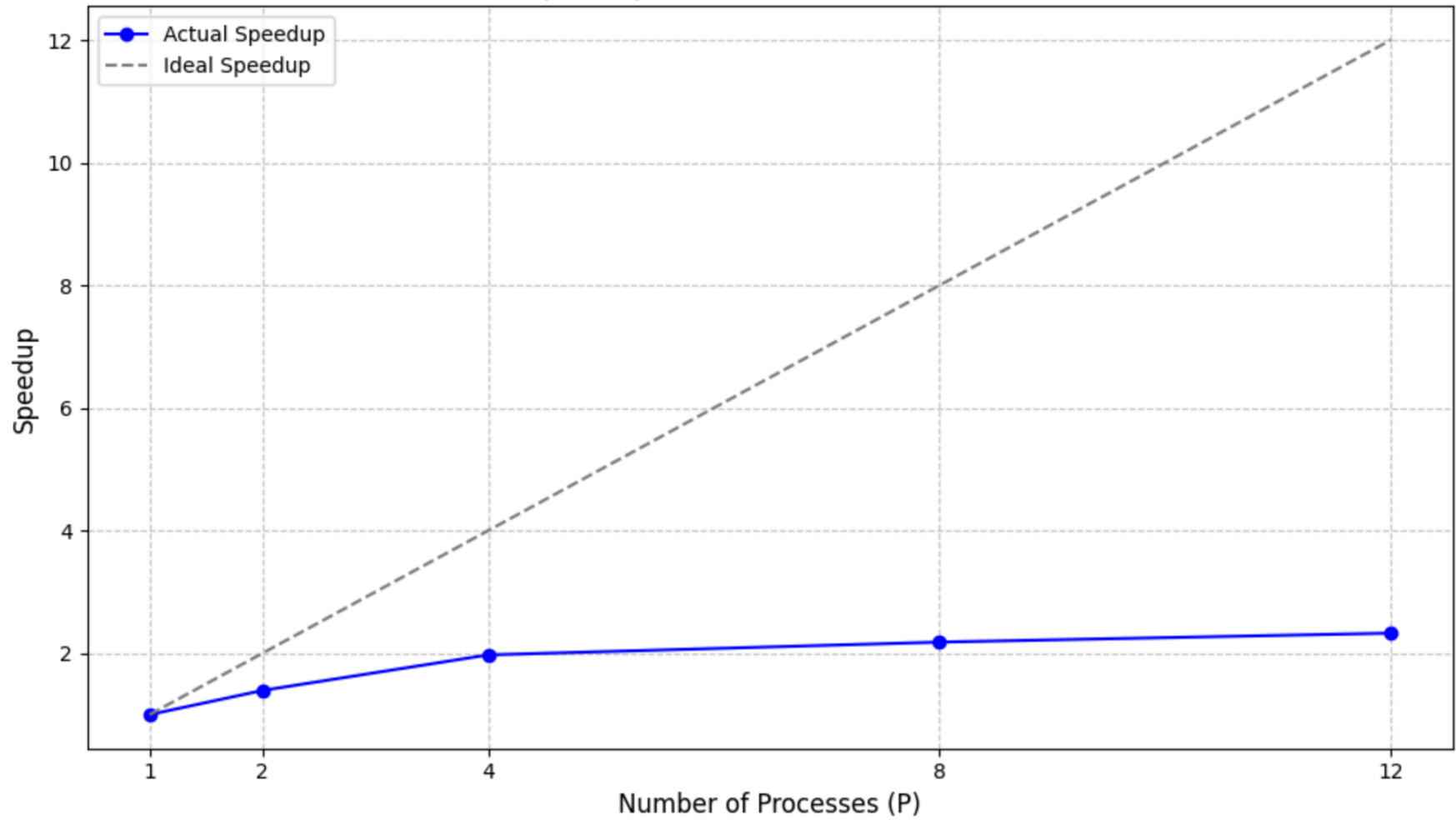
IO 在多 node 下沒有隨著 process 數量持續下降，反而波動，這可能是跨節點並行 I/O 的 overhead。CPU 大幅下降到 16 proc 只剩下 4s。Comm 從 8 procs 開始顯著上升，但 12、16 procs 時有下降趨勢，顯示 MPI Odd-Even Sort 的交換次數上限被碰到

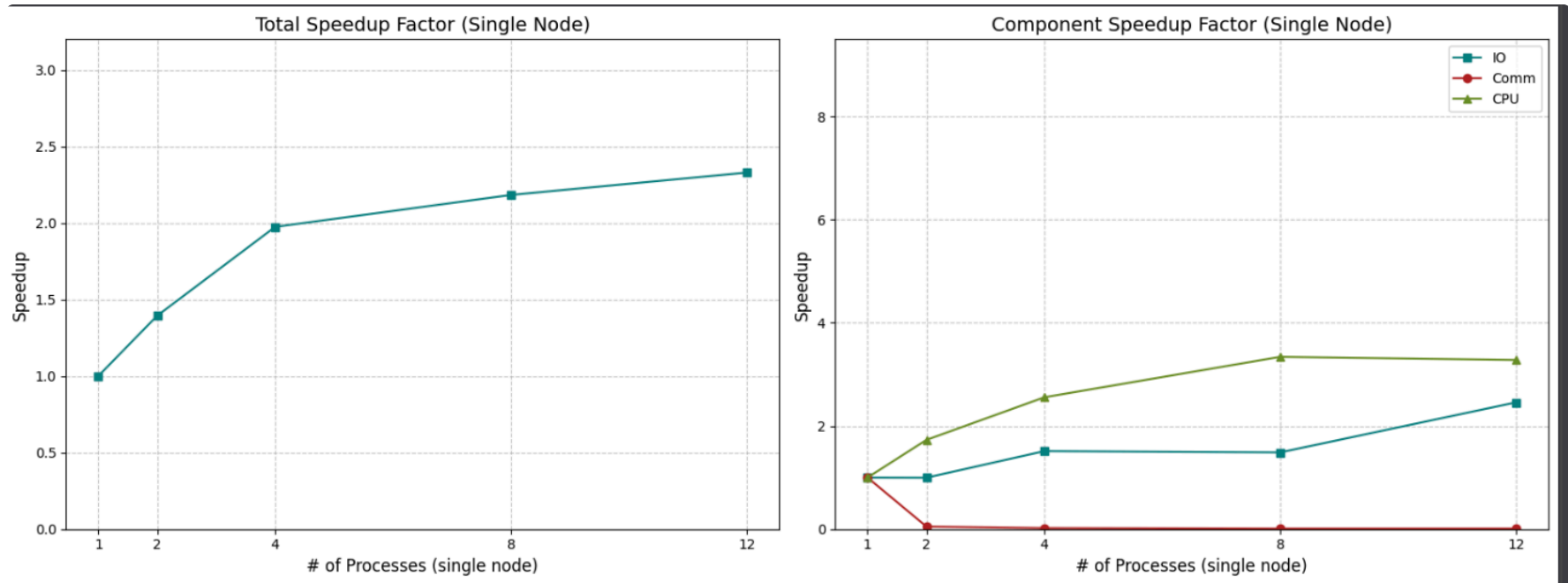
整體來說，Multi-Node 的 scaling 效果仍然存在，但相較於單節點，效能改善幅度受到通訊影響而趨緩。這反映了 Odd-Even Sort 在多節點環境下的結構性限制：它需要頻繁的鄰居交換，而跨節點的交換代價遠高於同一節點內的交換。

Speed Factor

SINGLE NODE

Speedup vs. Number of Processes



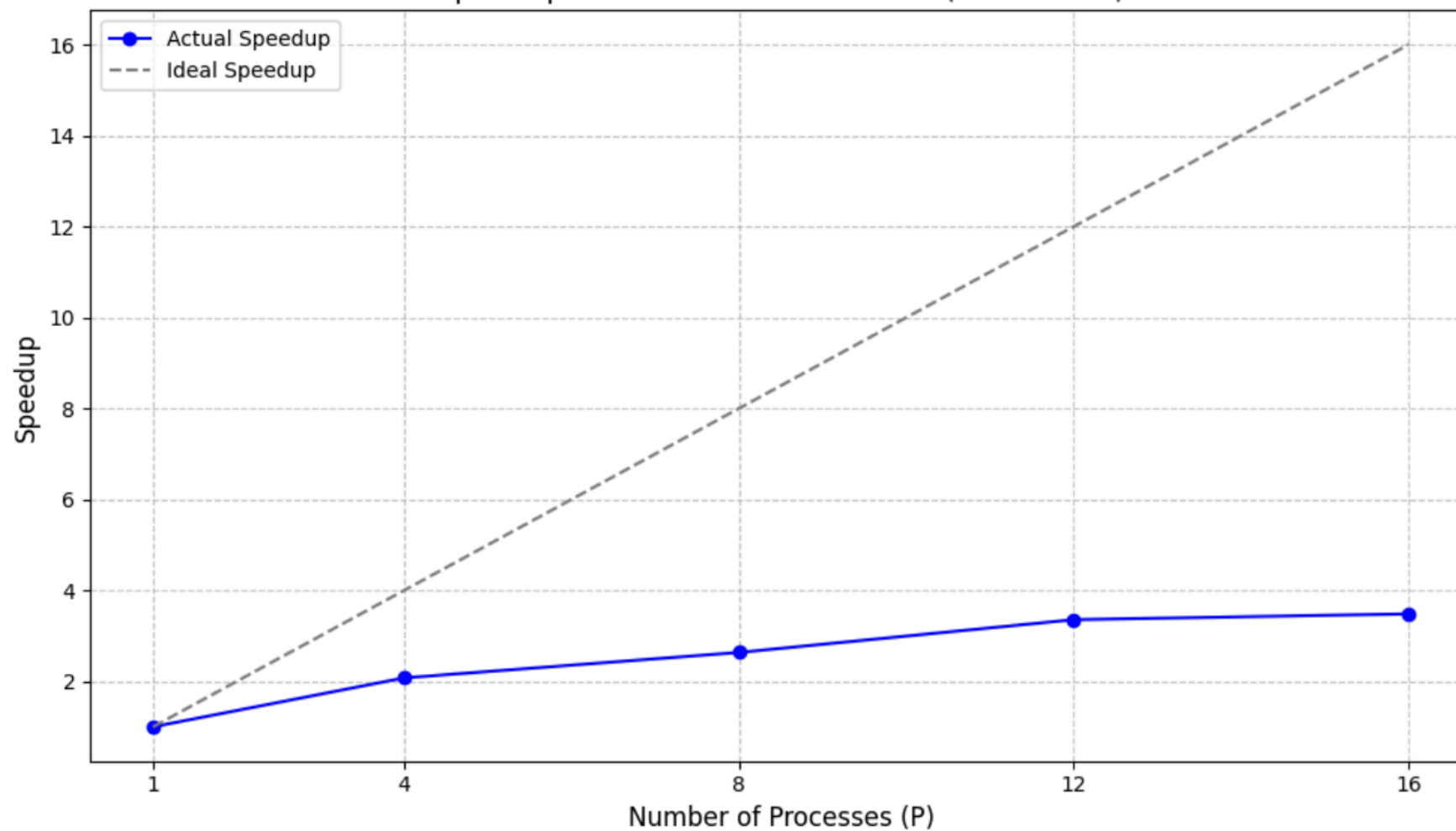


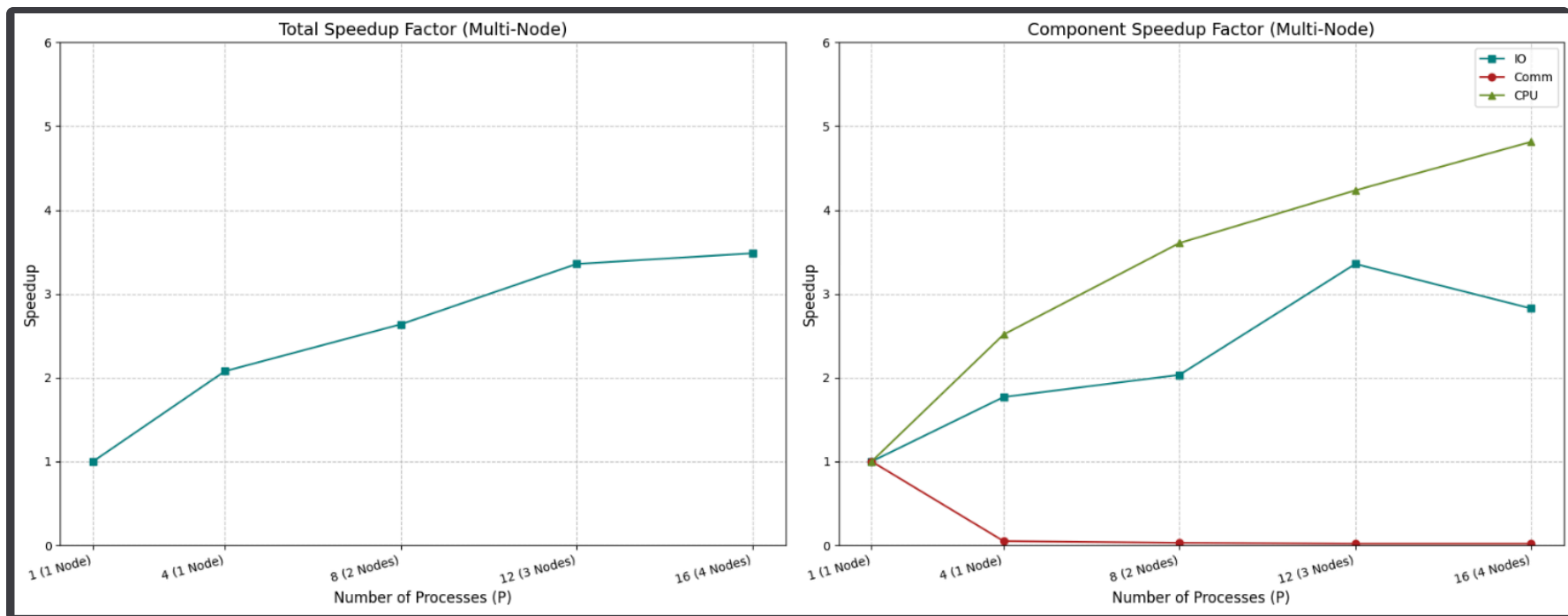
在單節點的情況下，隨著 process 數量的增加，程式的加速比呈現正向成長，但成長幅度逐漸趨緩。從 1 process 到 12 processes，speedup 從基準的 1 倍提升到約 2.33 倍，顯示平行化確實能帶來效能改善。不過，實際曲線並未接近理想的線性加速，原因在於 Odd-Even Sort 本質上需要頻繁的鄰居交換，當 process 數增加時，通訊與同步的比例逐漸提高，使得 CPU 計算的下降效益被抵消。這也是為什麼在 8 與 12 processes 的實驗中，雖然 local sort 的時間已經接近最小值，但 speedup 的增幅明顯放緩。

MULTINODE

NUMBER OF PROCESSES (NODES)

Speedup vs. Number of Processes (Multi-Node)





在多節點的情況下，加速效果雖然持續存在，但受到網路通訊的限制更為明顯。從 1 process 到 16 processes，總時間從 29 秒降到 8.4 秒左右，加速比約為 3.48。雖然比單節點的 2.33 倍更好，但與理想的 16 倍相差甚遠。這是因為當 process 跨越節點後，每一次交換都必須經過網路堆疊，造成更高的延遲與同步成本。尤其在 Odd-Even Sort 需要頻繁交換的情況下，跨節點的溝通變成主要瓶頸。因此，雖然多節點能持續降低執行時間，但 speedup 曲線在高 process 數量時變得更加平緩

Discussion

在單節點環境中，程式的效能變化可以很明顯地拆解為三個部分。CPU 時間隨著 process 數量增加而快速下降：從單 process 的 19.71 秒一路減少到 8 process 的 5.90 秒，證明了 local sort 的平行化確實有效。但在 12 process 時，CPU 卻微幅上升到 6.01 秒，這代表資料切分過小時，排序與合併的額外開銷開始抵銷效益。I/O 在 1 到 4 個 process 之間下降幅度最大 (7.20 → 4.76 秒)，但之後幾乎持平，顯示磁碟競爭成為瓶頸。至於通訊，從 2 process 開始就持續增加，在 12 process 時達到 2.93 秒，與 I/O 時間幾乎持平，說明在單節點下的限制來自同步交換的累積成本。這也是為什麼總時間在 8 和 12 process 之間的改善幅度有限。

在多節點實驗中，總時間下降得更明顯：從 1 process 的 29.37 秒降到 16 process 的 8.43 秒。CPU 部分幾乎呈線性下降，從 19.69 秒降到 4.09 秒，這是 speedup 的主要來源。然而，I/O 並不像單節點那樣隨著 process 數穩定下降，反而在 8 到 16 process 間出現震盪 (4.21 → 2.55 → 3.03 秒)，這與跨節點的 MPI-IO 有關，平行存取帶來了同步與網路干擾，導致效能不穩。通訊時間在 8 process 時達到最高 1.60 秒，但之後反而下降到 1.36 秒，這與 Odd-Even Sort 的演算法特性有關：每個 process 只和鄰居交換，當跨節點 process 數再增加，單次交換的資料量反而更小，抵銷了一部分網路延遲

Experiences / Conclusion

這次作業讓我體會到效能優化的複雜性。有時候小小的改動就能縮短時間，但照理論設計的優化反而會讓程式更慢。我曾嘗試改寫 merge，結果比原本簡單直覺的版本還慢，這讓我意識到最簡單的方法有時就是最好的。

最大的困難是難以理解為什麼別人的程式能比我快一半，但這也提醒我效能優化需要大量實驗與驗證，而不是只靠直覺。透過這次作業，我除了熟悉了平行化設計與 MPI 的使用方式，也學到效能分析與調整的方法。