HW3 - Standard Scaled Dot Product Attention with MPI

Team Information:
Group ID: Team 12
Team Member:

- 朱致伶 R14922040
- 李維哲 R14922134
- 江政諺 R14944041

說明: `green_function()`為 `MPI Instruction` , `orange_function()`為 `AVX512 Instruction`, `blue_function()`為自訂義函式

Q1. What optimization techniques do you apply?

在本次優化中，我們主要從三個方向去改善整體效能，**記憶體與資料配置**、**通訊與流程重疊**、**向量化與數值計算**。整體優化想法是：先用記憶體/資料切割把工作量平均、資料就近放置；再利用向量化與數值穩定最大化單一 rank 的計算量；最後以通訊優化與非阻塞管道把跨節點的等待時間藏進計算後面。下方為每個優化方向對應到的實際方式：

通訊與流程重疊（Distributed Communication & Pipelining）

- 兩階段全域歸約（`Iallreduce`：MAX → SUM）
- 非阻塞通訊 + Ping-Pong 緩衝（`Ibcast`/`Ireduce` + Ping-Pong Buffer）

向量化與數值計算（Vectorized Compute & Numerics）

- 資料型態向量化轉換（`cvt_d2f_avx512` / `cvt_f2d_avx512`）
- 向量化內積核心（`dot_avx512`）與 向量化 AXPY（`axpy_avx512`）
- 線上穩定 softmax（`online_softmax_attention`）

記憶體與資料配置（Memory & Data Locality）

- 資料切割與負載均衡（`owner_count` / `owner_disp`）
- 批量數值初始化 / 縮放（`memset_zero_scale` + Prefetch）

1. 分割策略：owner_count / owner_disp 與資料切割

將 **n**（K/V 的列數）按「**幾乎等分 + 前段補 1**」的方式分配給 size 個 rank，使每個 rank 的本地份量 `n_local` 接近平均，避免負載不均。

- `owner_count(n,size,rank)`:
  算出每個 `rank` 拿到的列數 = `n/size`，若 `rank < n % size` 則再加 1。
- `owner_disp(n,size,rank)`:

算出當前 `rank` 的起始列偏移 = 前面所有 rank 的 owner_count 加總。

2. **資料型態向量化轉換：cvt_d2f_avx512 / cvt_f2d_avx512**

在 `attention()` 計算中改用 float 以**提高 AVX512 單一指令的計算處理量**；寫回 result 再轉回 double，以保留精度**避免誤差累積**，可安全採用 **混合精度 (Mixed Precision)**，符合評分誤差 ±0.02 的容許範圍。

使用的 AVX512 基本單位：

向量型別	寬度	一次容納
__m512	512-bit	16 個 float
__m512d	512-bit	8 個 double
__m256	256-bit	8 個 float

Double 與 Float 型態雙向轉換的實作：

- `cvt_d2f_avx512(dst, src, n) & cvt_f2d_avx512(dst, src, n)`:
  - **大批次**：每回合處理 **32 筆** 以減少 loop overhead。
  - **中批次**：剩下可被 8 整除的餘量，單回合處理 8 筆。
  - **尾端**：逐一 scalar 轉換，保證正確性。

搭配 **AVX512 + loop unrolling**，「大批次→中批次→尾端」的階梯式批次化，常見於高效向量化 loop，能兼顧吞吐量與邊界處理的簡潔性。

3. **向量化核心算子：dot_avx512 與 axpy_avx512**

`dot_avx512(a,b,n)`： 計算內積，是 Attention 中 **QK^T** 中的基本單元，利用了 AVX512 的 **FMA (Fusion Multiply-Add)** 與 **mask 載入**。採用四個 accumulator，每回合處理 64 (4 x 16) 個浮點數；相比標準迴圈大幅降低指令次數與 cache misses。

- 使用 `_mm512_fmadd_ps` 指令累積結果；
- 結尾以 `_mm512_reduce_add_ps` 將四個 accumulator 的 partial sum 水平加總。

`axpy_avx512(y,x,alpha,n)`：`y += alpha * x`，**把某一列 V 的加權值累加到輸出貢獻**（softmax 權重乘上 V[j]）。使用相同 `_mm512_fmadd_ps` 操作，支援 64 元素展開與 mask 尾段處理。

4. Online softmax + batched：online_softmax_attention

採「**Online Max-Tracking**」演算法，以即時更新 `rmax` 和 `rsum`。

- 當新的分數超過現有最大值時，以 `corr = exp(old_rmax-new_rmax)` 修正既有累積；
- 即時計算 `exp(s-rmax)` 加權並累積至 contrib。

5. 全域規範化：兩階段 MPI_Iallreduce（MAX → SUM）

為確保所有 rank 的 softmax 數值一致且穩定，採用**兩階段全域歸約** `MPI_Iallreduce`：

- **全域最大值 Global Max**：以 `MPI_Iallreduce(..., MPI_MAX, ...)` 聚合各 rank 的 `lmax[b]` 得到 `gmax[b]`，再用 `corr = exp(lmax[b] - gmax[b])` 校正本地的 `lsum[b]` 與 `contrib[b]`。
- **全域總和 Global Sum**：接著以 `MPI_Iallreduce(..., MPI_SUM, ...)` 得到 `gsum[b]`，再令 `contrib[b] *= 1 / gsum[b]` 完成全域 softmax 規範化。

先取 MAX 再 SUM，能在統一的最大值基準下加總，避免數值誤差放大。

6. 批次與重疊：Ping-Pong Buffers + Prefetch + 非阻塞廣播/歸約

採用 **雙緩衝機制** 與 **非阻塞通訊 Non-Blocking Communication** (`MPI_Ibcast` / `MPI_Ireduce`)，將計算與通訊重疊：

- **Q buffer (q_bufs)**: 當前批次使用上一輪已廣播完成的 Q；同時以 `MPI_Ibcast` 非阻塞傳輸下一 Batch。
- **contrib 緩衝 (cb_bufs / rb_bufs)**: 當前批次計算完即發出 `MPI_Ireduce`，上一批的結果轉回 double 並寫入 result。
- **Prefetch:** 在 `online_softmax_attention` 中以 `_mm_prefetch` 提前載入下一列 `K_local`、`V_local`，降低 cache miss。

採用「**通訊/計算重疊 + 雙緩衝**」設計可將網路延遲隱藏於運算過程中，大幅提升整體平行效率。

7. 向量化初始化與縮放 (memset_zero_scale) + Prefetch

使用 AVX-512 指令將整段向量快速設為零或乘以常數。

- 若 `zero_first=true`，善用`_mm512_setzero_ps`, `_mm512_storeu_ps`。
- 否則以 `_mm512_mul_ps` 將原資料整段縮放。

同時在 `online_softmax_attention` 內部以 `_mm_prefetch` 提前載入下一筆 K、V，預熱 cache、降低記憶體延遲。

Q2. How do different optimization techniques influence performance?

此段落分析 Q1 提及的優化方式的效果，在 4 個計算節點、每節點 16 個 MPI Processes（共 64 Processes）的環境下，以分階段疊加的方式評估各項最佳化策略。

Baseline 為單純只使用 MPI 分割策略的 Attention 運算方式。由於部分技術在測試中是直接套用於 Baseline 上進行，因此在**單獨觀察時，加速效果未必顯著**；然而，這些技術能改善資料型態、計算吞吐與通訊行為，並在**整合後明顯提升整體效能**。

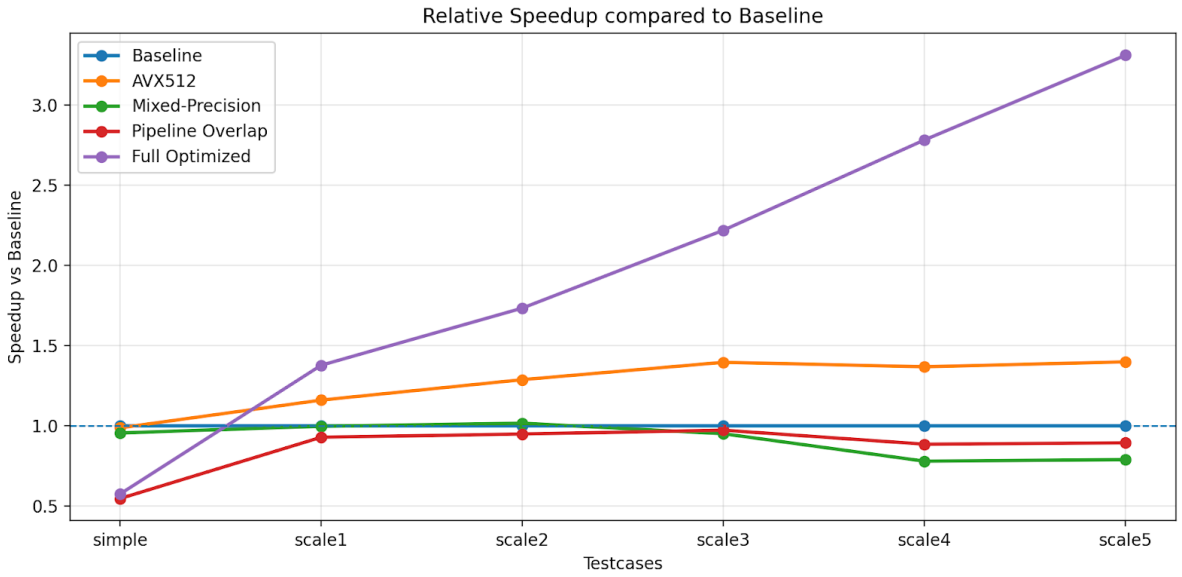

圖 1. 相對於 Baseline 的 Speedup Ratio（4 Nodes × 16 Processes/Node）

Testcase	AVX512	Mixed-Precision	Pipeline Overlap	Full Optimized
Simple	0.99×	0.96×	0.55×	0.58×
Scale1	1.16×	1.00×	0.93×	1.38×
Scale2	1.29×	1.02×	0.95×	1.73×
Scale3	1.40×	0.95×	0.97×	2.22×
Scale4	1.37×	0.78×	0.89×	2.78×
Scale5	1.40×	0.79×	0.89×	3.31×

Table 1. 相對於 Baseline 的 Speedup Ratio（4 Nodes × 16 Processes/Node）

- **AVX512 + Loop Unrolling**

透過 AVX512 向量指令提升計算量，並配合迴圈展開減少分支與索引開銷，使 CPU 管線維持高飽和度。屬於**計算核心層級**的可持續加速手段。結果顯示，在 scale1 ~ scale5 中均維持 **1.16× 至 1.40×** 的穩定加速；即便在最大規模（scale5）下也能持續提升，顯示此方法主要受限於記憶體頻寬，而非通訊瓶頸。

- **Mixed Precision（double→float）**

單獨測試時效能略有波動（約 0.95× ~ 1.02×），顯示轉換成本與浮點單位的差異抵銷了部分效益；但此方法顯著降低 K/V 矩陣的傳輸量與快取壓力， 結合 AVX512 與 Overlap 的整合貢獻關鍵。

- **Pipeline Overlap（計算/通訊重疊）**

利用非阻塞 MPI 通訊（MPI_Ibcast、MPI_Ireduce）及 Ping-Pong 雙緩衝，使通訊與 Softmax 計算並行進行。在小規模（simple、scale1）時因通訊比例低，效能僅約 0.55× ~ 0.93×，效果有限；但隨著規模擴大至 scale3 以上，**重疊效應開始顯現，延遲被計算階段隱藏**，最終穩定於 0.97× 左右。

- **Full Optimization（整合所有技巧）**

最終版本整合 AVX512、Mixed-Precision 與 Pipeline Overlap 等技術，同時優化：

  - 計算端 ：AVX512 向量化 + 混合精度減負載
  - 傳輸端 ：自適應 Bcast/Scatterv 分發 + Overlap 隱藏通訊延遲

效能呈現**隨規模遞增的線性加速趨勢**，從 simple 的 0.58× 逐步提升至 scale5 的 3.31×，在最大測資下明顯優於任一單獨技術。顯示跨層整合能同時發揮計算吞吐與通訊隱藏的綜效，達到整體系統最佳化。

結論：AVX512 提供穩定的計算加速，Mixed-Precision 降低傳輸與快取壓力，Pipeline Overlap 隱藏通訊延遲，三者整合後可於多節點 MPI 環境中實現超過 3× 的整體效能提升。

Q3. How does matrix size influence performance?

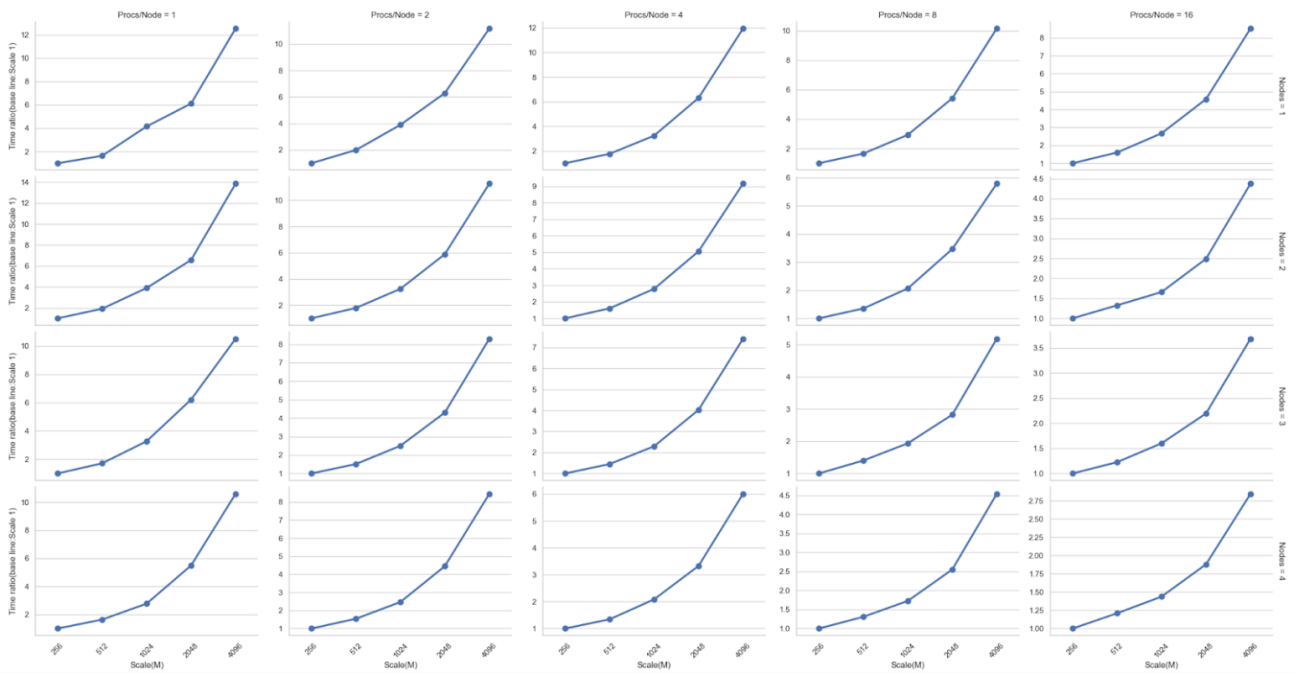

圖 2. 不同矩陣大小、節點數與每節點 Process 數下的執行時間變化

（顯示在不同節點數與每節點 process 數組合下，執行時間隨矩陣大小變化的趨勢）

整體趨勢為**執行時間隨著矩陣規模（Scale）增加而上升**，符合計算複雜度 $O(m \times n \times dk)$的理論預期。成長幅度並沒有像想像中的成比例增加，而是**呈非線性成長現象**，時間成長幅度大於線性，尤其在 scale4、scale5 之後曲線明顯上升，顯示效能受到快取與記憶體瓶頸影響。可能原因包括：

- 計算資料量超出 CPU cache 容量，造成頻繁快取未命中（cache miss）
- 大型矩陣導致 MPI 通訊資料量增加，網路延遲成為主要瓶頸

同時，效能瓶頸隨著矩陣大小增加而轉移，小矩陣以計算開銷為主（compute-bound），大矩陣則逐漸轉為記憶體與通訊受限（memory / communication-bound）。 在平行化程式中(procs/node 和 node 數量增加的情況)下，可觀察到**最大與最小測資間的執行時間差距顯著縮小**。這反映了兩種可能情況：

- 小型矩陣下的通訊負擔過高：當 process 數過多時，通訊成本佔比增加，導致小測資執行時間反而上升；
- 大型矩陣的良好可擴展性：在大測資中，**計算負載足以攤平通訊延遲**，平行化能有效提升效能並縮短時間。

Q4. How do the number of processes influence parallelization performance?

從圖三可觀察到，在固定 4 個節點、Scale5（最大測資）下，整體呈現良好的 Strong Scaling 趨勢。當 process 數增加時，執行時間持續下降，但加速比 (Speedup) 並非無限線性增長，呈現「**前期接近理想、後期遞減趨緩**」的典型平行化行為。在大規模資料下可展現良好的加速比，但對小規模問題若過度平行化，則會因通訊與同步開銷導致效率反而下降。結合圖二去綜合分析。

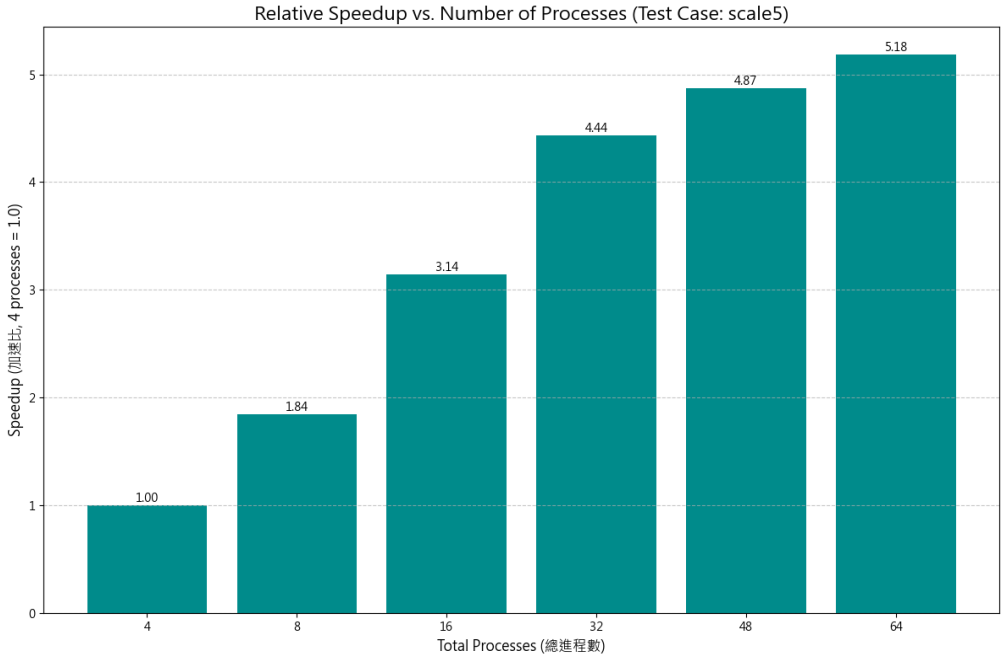

圖 3. 不同 Process 數對平行化效能 (Strong Scaling) 的影響（固定 4 Nodes, Scale5）

初期階段：良好的線性擴展性（4 → 8 procs/node）

一開始通訊壅塞較低且 CPU 資源未充分利用，當 process 數從 4 增加至 8 時，加速比達 1.84×，**幾乎接近理想的 2×**。此階段的計算量仍足以攤平：

- MPI 啟動與同步開銷；
- 通訊延遲與封包累積效應。

顯示此時程式的 **平行效率 (Parallel Efficiency) 仍維持高水準**，CPU 資源被充分利用，**計算/通訊比 (Compute-to-Communication Ratio) 高**。

後期階段：加速比趨緩（32 → 48 → 64 procs/node）

隨著 process 數持續增加，每個 process 分配到的資料量減少，導致：

- 計算時間下降幅度有限；
- 通訊與同步時間無法完全隱藏；
- Amdahl's Law 所揭示的「非平行化部分」開始主導效能上限。

觀察到 speedup 僅從 4.44 → 4.87 → 5.18，增幅明顯減緩。此現象顯示程式已**進入通訊受限區域 (Communication-bound region)，平行效率開始下降**。

另外，從圖二觀察，可以發現高 Process 數（16）在不同矩陣大小的效能優化有不同結果。

- 對大型矩陣仍具**明顯加速效果**（時間約減半）;
- 但對小型矩陣反而因 **MPI 通訊比例過高而效能下降**（Over-parallelization Overhead）。

Q5. How do the number of nodes influence parallelization performance?

我們保持總 Process 數固定為 16 不變，僅改變 Process 在節點之間的分布方式，藉此說明「跨節點通訊」對效能的影響。如圖四所示，當所有 16 個 Processes 都配置在同一節點時（**1 node, 16 procs/node**），整體效能最佳。我們將此作為基準。然而，當相同的 Process 數分散至**兩個節點（2 nodes, 8 procs/node）**時，效能下降至 0.926；進一步分散至**四個節點（4 nodes, 4 procs/node）**後，效能降至 0.909。

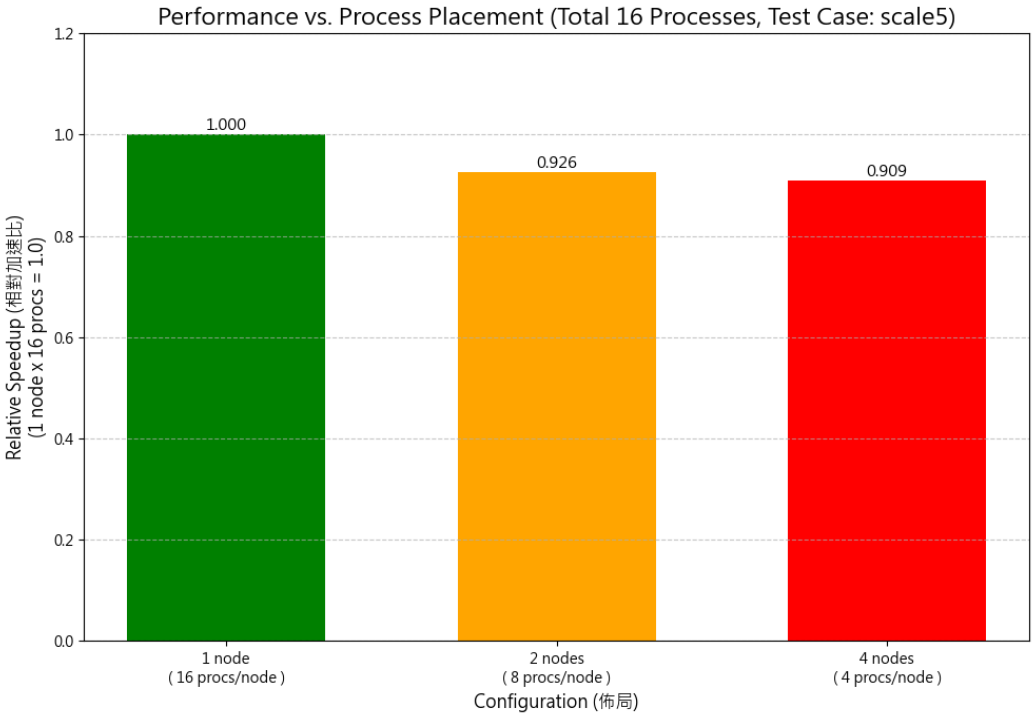

圖 4. 不同節點分佈 (Process Placement) 對平行效能的影響（固定 16 Processes，測試案例：scale5）

此結果顯示，在**總計算量與 Process 數相同的條件下，節點數越多、跨節點通訊越頻繁**，性能便越受影響。其原因在於跨節點通訊（特別是 `MPI_Bcast`, `MPI_Allreduce`, `MPI_Reduce` 等 collective operations），其延遲與頻寬皆遠高於同節點內共享記憶體通訊。因此，當 Process 分布越分散，這些**通訊延遲便成為主要的效能瓶頸**，導致整體速度下降。此結果清楚突顯了通訊成本在平行化運算中的主導影響。

在我們了解到通訊時間會大幅影響總時間的基礎下，再來觀察以下的數據，**利用保持一樣的總 Processes 數，改變 Process 的分布**，就可以更進一步的發現通訊時間的影響，以及跨節點通訊是一筆極大的開銷。

Q6. Speedup of parallelized code compared to the serial one.

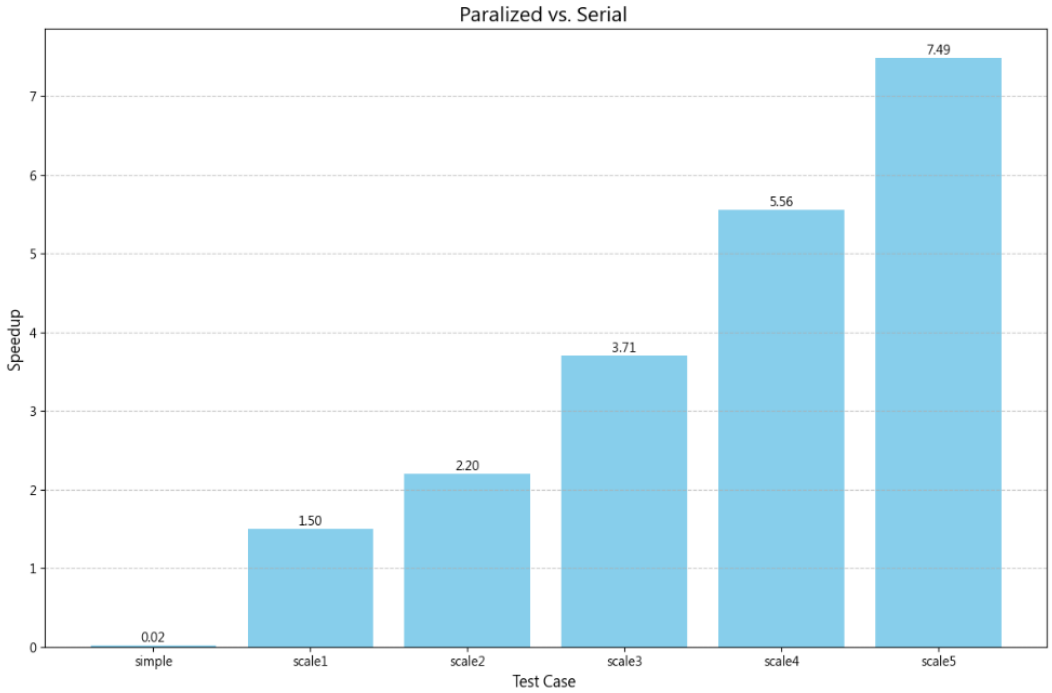

圖 5. 平行化程式相較於序列程式的加速比

我們比較了序列版本與在 4 個節點、共 64 個 MPI Processes 下的平行化版本之執行效能。從圖五中可以明顯觀察到，加速比 (Speedup) 與測試案例的規模 (Scale) 之間呈現強烈的正相關。

- **小型測資（simple ~ scale1）:**

Speedup 幾乎無法體現，甚至在最小測資下僅約 0.02×，代表平行化版本比序列程式更慢。這是因為在小規模問題中，**通訊開銷 (communication overhead) 遠大於計算時間**，MPI 的資料分派與同步成本掩蓋了任何潛在的計算加速；當平行程式仍在等待資料交換時，序列程式往往已經完成整個運算。

- **中大型測資（scale2 ~ scale4）:**

隨著矩陣規模增大，計算量成為主導因子，**通訊延遲開始被計算時間所隱藏 (latency hiding)**。**Speedup 由 2.20×（scale2）提升至 5.56×（scale4）**，顯示平行化策略逐漸發揮效益。此階段的 Compute-to-Communication Ratio 顯著提高，MPI 通訊不再是主要瓶頸。

- **最大測資（scale5）:**

當問題規模進一步擴大，**CPU 的運算負載足以完全攤平通訊延遲**，整體 **Speedup 提升至 7.49×**，展現良好的**強尺度擴展性 (Strong Scaling)**。表示在大型問題下，平行化能有效利用多節點的總運算資源，**接近理想的線性加速**。

Q7. Scalability Analysis

針對 Weak Scalability Analysis:

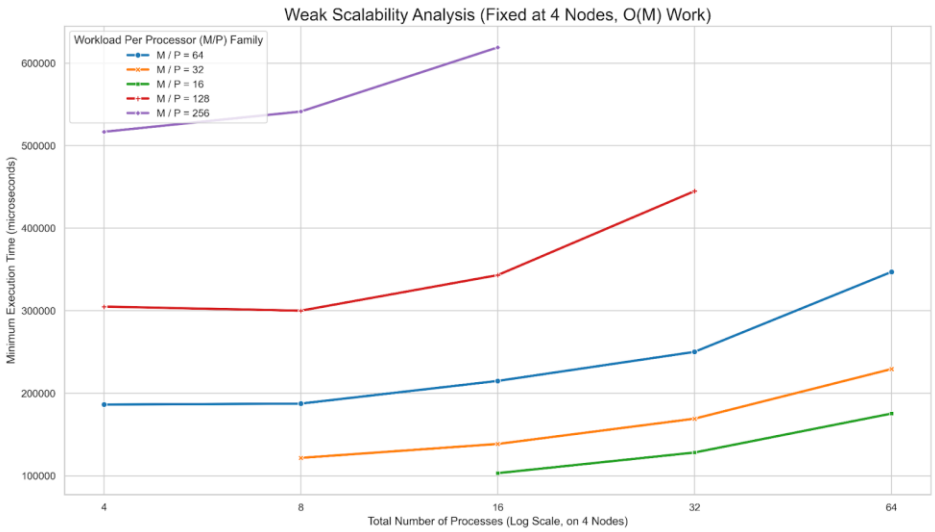

圖 6. Weak Scalability Analysis

圖 6 中**同一條線中的每一個點代表著每一個 process 都有著相同的工作量**，而理想的 Weak scalability 線段應該是要呈現水平線。由此圖可以了解到，當**總 Processes 數由 4 到 8** 時，每條線都趨近於理想的水平線，代表有著**良好的 Weak scalability**，整體延遲與吞吐量表現最平滑、效率最佳。但**當 process 數量逐漸增加時，完成工作的時間逐漸上升**並且斜率逐漸增大，我們推斷可能的原因是分配出去的 process 數變多了，等待資料返回 host 的時間也會逐漸增大，並且由於總 process 數是乘冪次成長，這也反映了斜率逐漸增大的原因。

針對 Strong Scalability Analysis:

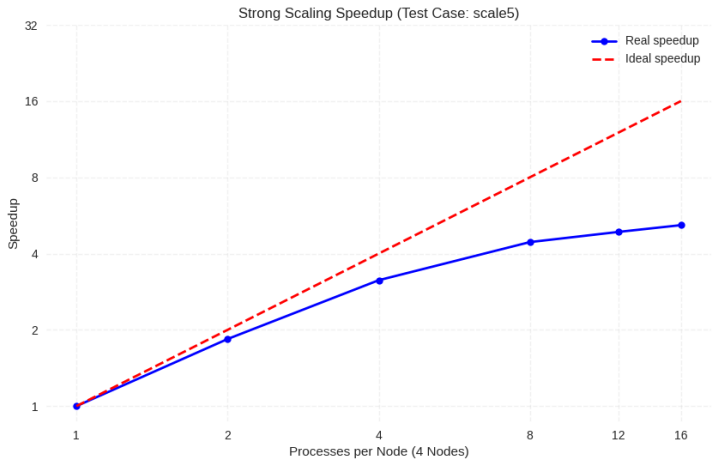

圖 7. Strong Scalability Analysis

我們針對固定問題大小（Test Case: scale*5*）進行 Strong Scalability 測試，並在 4 個 Nodes 的前提下逐步增加 Processes per Node (procs/node) 數量，描繪其 Speedup 與理想線性加速的差異。如圖七所示，紅色虛線代表理想加速情況，而藍色實線為實際測得的加速效果。

在較低的並行度（1 → 2 → 4 procs/node）區間中，**實際 Speedup 曲線與理想線性趨勢相近**，顯示此階段中程式計算部分（包含向量化的 dot_avx512、axpy_avx512 以及 online softmax kernels）能**有效地隨 procs/node 平行分工**，使工作負載縮小能直接反映在執行時間的縮短上，展現出良好的可擴展性。

然而，當 Process 數持續增加至 8 → 12 → 16 時，Speedup 明顯開始偏離理想線性成長，**曲線趨向平緩**。此現象反映了通訊同步成本在高並行度下的佔比上升：隨著每個 Process 分到的資料量變小，計算時間下降程度比跨節點通訊時間更快，導致**整體效能開始受到網路延遲與全域同步所限制**。

因此，整體 Strong Scalability 行為可分為兩個區段：

- 中低並行度 (1-4 **procs/node**)：計算成本主導 → 加速效果良好（接近線性）
- 高並行度 (8-16 **procs/node**)：通訊成本主導 → 加速曲線趨於飽和

Q8. Anything You Want to Share

自適應資料分發：Bcast vs Scatterv 閥值策略

在多節點 Attention 中，K 與 V 需由 root 分發至所有節點。我們起初設計為：

- MPI_Bcast 廣播 Q（所有節點皆需完整 Q）；
- MPI_Scatterv 分散 K / V（每節點僅需部分片段）。

此做法理論上可減少傳輸量，但實測結果顯示在數 MB ~ 數十 MB 資料規模下，Scatterv 的啟動延遲反而更高。

經過實驗觀察與分析，發現造成**效能差異的關鍵在於 MPI 的內部傳輸拓樸建立機制：**

方法	技術特性	實測結果
Scatterv	每次呼叫需重建多對多傳輸拓樸與偏移表	初始化成本高，延遲偏大
Bcast	在 MPI_Init 時已建立樹狀通訊拓樸	呼叫時可直接使用，延遲低且穩定

結論：中小型資料下 Bcast 具明顯優勢，啟動延遲更低、傳輸更穩定。

根據觀察結論，為兼顧效能與記憶體使用，本程式採**自適應分發策略 (Adaptive Strategy)**，根據 K/V 總資料量動態切換通訊模式 (此 64 MB 閥值經多組 scale 測試驗證，可平衡延遲與峰值記憶體負載)：

條件	使用方法	適用情境
< 64 MB	`MPI_Bcast`	小資料，啟動成本低、延遲最小
≥ 64 MB	`MPI_Scatterv`	大資料，節省記憶體與頻寬

(1) 小資料：`MPI_Bcast` 模式

1. **Root** 將 `K`、`V` (double) 轉為 float。
2. 以 `MPI_Bcast(Kf)`、`MPI_Bcast(Vf)` 廣播完整矩陣。
3. 各 rank 依 `owner_disp()` 擷取所屬區段到 `K_local`、`V_local`。利用既有樹狀通訊結構，延遲最小。

(2) 大資料：`MPI_Scatterv` 模式

1. **Root** 轉型後依 `owner_count()`、`owner_disp()` 準備偏移表。
2. 以 `MPI_Scatterv` 將對應片段直接發送至各 rank。每節點僅接收必要資料，節省記憶體。