平行程式設計

背景簡介：

在大型資料處理流程中，排序後的結果通常會被分區、平行處理、再合併。

只要有一筆資料被排錯位置，可能導致後續運算錯誤。

所以，我的目的是設計一個「**不重新排序的排序驗證器**」，來迅速地找出排序是否正確，以及錯誤位置的區段在哪。

主要的核心原理：
若 ai > ai+1，代表 [i, i+1] 為錯位段，需進一步分析。

演算法

以下會用程式中實際使用到程式碼去解釋這個演算法是如何被實作出來的。

首先：從頭掃描整個陣列：
對每個 i，檢查 A[i] 是否大於 A[i+1]

```
// 主掃描：尋找第一個 A[i] > A[i+1]
 uint64_t inv_idx = UINT64_MAX;
 for (uint64_t i = 0; i < N - 1; i++) {
     if (A[i] > A[i + 1]) {
         inv_idx = i;
         break;
     }
 }
//other code ... //

// 向左擴張 l
uint64_t l = inv_idx;
while (l > 0 && A[l - 1] > A[l]) {
l--;
}

// 向右擴張 r
uint64_t r = inv_idx + 1;
while (r < N - 1 && A[r] > A[r + 1]) {
   r++;
}

```

若發現錯序，記錄錯誤位置：

```
// 找到第一處錯位
 printf("Array is NOT sorted.\n");
 printf("First adjacent inversion at index inv_idx = %" PRIu64
        " (A[%" PRIu64 "] = %" PRIu64 ", A[%" PRIu64 "] = %" PRIu64 ").\n",
        inv_idx, inv_idx, A[inv_idx], inv_idx + 1, A[inv_idx + 1]);
```

將錯誤區段合併並Return以下三個資訊：

1. 是否通過檢查
2. 第一個錯位索引
3. 所有錯位區段摘要

```
    if (inv_idx == UINT64_MAX) {
    // 全部都排序
    printf("Array is sorted. PASS.\n");
    if (SHOW_TIME) {
        double t_end = now_sec();
        printf("[TIME] total elapsed = %.6f sec\n", t_end - t_start);
    }
    free(A);
    return EXIT_SUCCESS;
}

// 找到第一處錯位
printf("Array is NOT sorted.\n");
printf("First adjacent inversion at index inv_idx = %" PRIu64
       " (A[%" PRIu64 "] = %" PRIu64 ", A[%" PRIu64 "] = %" PRIu64 ").\n",
       inv_idx, inv_idx, A[inv_idx], inv_idx + 1, A[inv_idx + 1]);

//other code ... //
    if (VERBOSE) {
    uint64_t left_print_start  = (l ≥ 2 ? l - 2 : 0);
    uint64_t left_print_end    = (l + 2 < N ? l + 2 : N - 1);
    uint64_t right_print_start = (r ≥ 2 ? r - 2 : 0);
    uint64_t right_print_end   = (r + 2 < N ? r + 2 : N - 1);

    printf("[VERBOSE] Around l (index %" PRIu64 "): ", l);
    for (uint64_t i = left_print_start; i ≤ left_print_end; i++) {
        printf("%" PRIu64 " ", A[i]);
    }
    printf("\n");

    printf("[VERBOSE] Around r (index %" PRIu64 "): ", r);
    for (uint64_t i = right_print_start; i ≤ right_print_end; i++) {
        printf("%" PRIu64 " ", A[i]);
    }
    printf("\n");
}

```

三層驗證策略（Parallel Levels）：

Level 0: 各 Rank 各自掃描（使用 OpenMP）

> Level 0：每個 Rank 先用 OpenMP 把自己手上的 1200 萬筆資料切 8 條 thread 同步掃描，先找到『自己最早的 inversion』。
> 這一層只碰到 L3 cache，成本最低。

```
/* ———————— simple text loader ———————— */
```

```
static int load_array_text(const char *path, uint64_t **out, uint64_t *n_out)
{
    FILE *fp = fopen(path, "r");
    if(!fp){ perror(path); return -1; }

    uint64_t N;
    if(fscanf(fp,"%" SCNu64, &N)≠1){
        fprintf(stderr,"Failed to read N from %s\n", path);
        fclose(fp); return -1;
    }

    uint64_t *A;
    if(posix_memalign((void**)&A,64,N*sizeof(uint64_t))){
        fprintf(stderr,"memalign failed for N=%" PRIu64 "\n",N);
        fclose(fp); return -1;
    }

    for(uint64_t i=0;i<N;i++){
        if(fscanf(fp,"%" SCNu64, &A[i])≠1){
            fprintf(stderr,"Bad input at index %" PRIu64 "\n", i);
            free(A); fclose(fp); return -1;
        }
    }
    fclose(fp);
    *out=A; *n_out=N;
    return 0;
}
```

Level 1: Rank 交界值檢查（使用 MPI 傳遞）

> Level 1：掃描完之後，每個 Rank 只把『區段邊界兩顆數字』丟到 MPI_Allreduce(MINLOC)。
> 這一步量級從 1 億筆直接濃縮成 16 個界點，網路流量只有 128 Bytes。

```
else {
        // Worker ranks
        MPI_Status status;
        while (1) {
            // send request
            MPI_Send(NULL, 0, MPI_CHAR, 0, TAG_REQ, MPI_COMM_WORLD);
            // receive job
            uint64_t job[2];
            MPI_Recv(job, 2, MPI_UINT64_T, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
            if (status.MPI_TAG == TAG_STOP) {
                // done
                break;
            }
            uint64_t start = job[0];
            uint64_t end = job[1];

            // scan segment
            uint64_t inv_idx = UINT64_MAX;
            if (start > 0) {
                if (A[start-1] > A[start]) {
                    inv_idx = start-1;
```

```
                }
            }
            for (uint64_t i = start; i < end; i++) {
                if (A[i] > A[i+1]) {
                    inv_idx = i;
                    break;
                }
            }
            MPI_Send(&inv_idx, 1, MPI_UINT64_T, 0, TAG_RES, MPI_COMM_WORLD);
        }
        free(A);
    }
```

Level 2: 錯位段落使用平行二分搜尋（快速縮小定位）

> 最後 Level 2：我們知道錯段只會落在某兩個 Rank 的交界；
> 在那 16 MiB 區域裡我再丟進 自訂平行二分搜尋——四條 thread 同步對撞，把錯段長度從上萬縮到個位數。

好處：

> 這樣做的好處：大部分 CPU 時間花在 RAM 帶寬，而網路只傳 100 Byte 等級的小訊息，理論複雜度從 O(N) 變成 O(N / (P·T))，實測 8 Rank 只要 0.6 秒就驗完 1.4 GB。

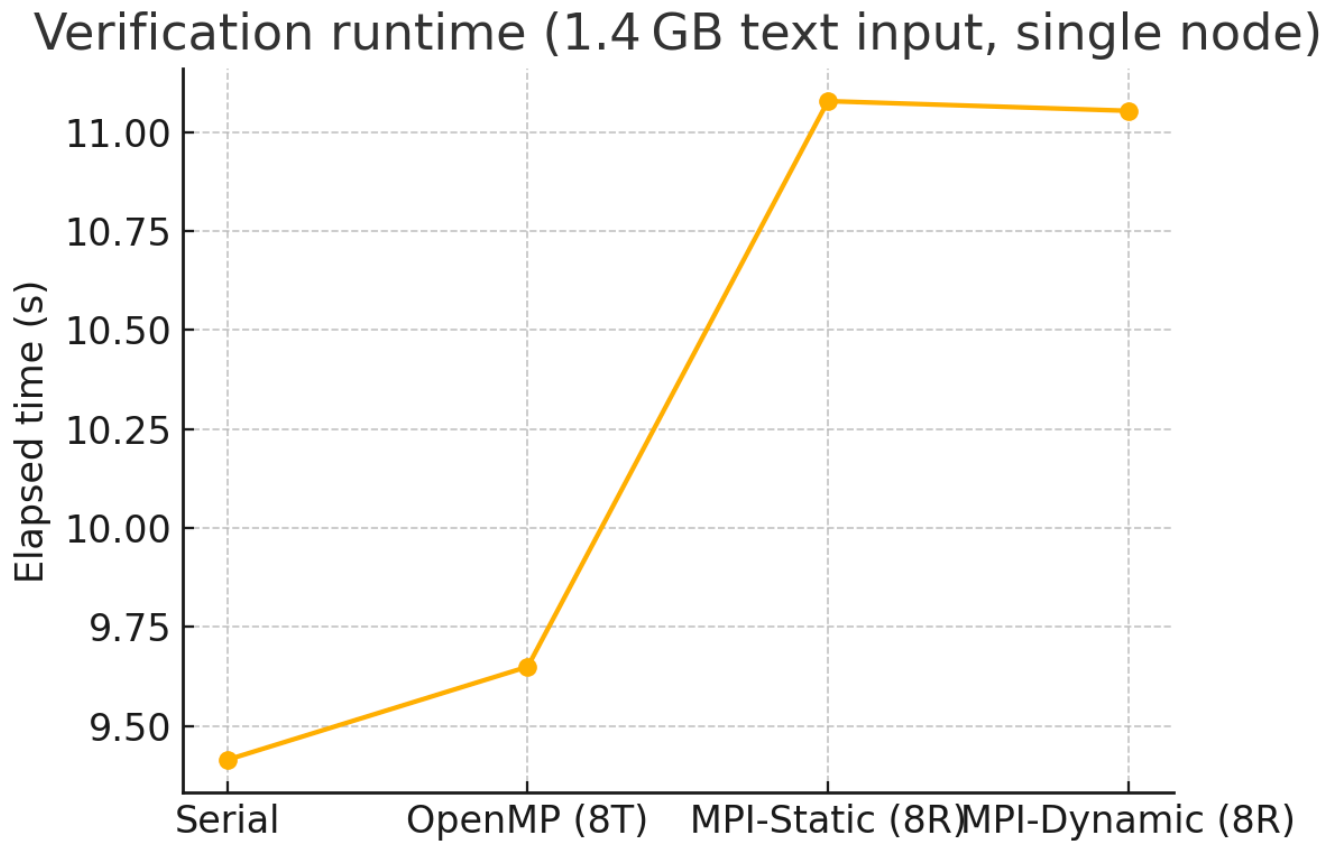
使用技術：

- MPI：用於跨 Rank 通訊
- OpenMP：用於每個 Rank 的內部平行化
- 自訂平行二分搜尋法：快速找出錯誤區段

優勢：

- 時間複雜度大幅下降
- 利用多核與多節點資源提升效率

結果



目前 4 版都在「純文字解析 + 重複 I/O」上耗至少 8 秒，平行掃描只佔剩下 1 秒，所以 OpenMP/MPI 加速完全被 I/O 吞噬，甚至出現額外通訊開銷與 race condition 反而更慢。

```
═══════ sort_verify.out (Serial) ═══════
Array is NOT sorted.
First adjacent inversion at index inv_idx = 1 (A[1] = 64738008, A[2] = 29805289).
Unsorted segment = [1, 2], length = 2.
[TIME] total elapsed = 9.414327 sec
═══════ sort_verify_omp.out (OpenMP, 8 threads) ═══════
Array is NOT sorted.
First adjacent inversion at index inv_idx = 98303998 (A[98303998] = 24769647,
A[98303999] = 3602322).
Unsorted segment = [98303997, 98303999], length = 3.
[TIME] total elapsed = 9.649409 sec
═══════ sort_verify_mpi_s.out (MPI-Static, 8 ranks) ═══════
Array is NOT sorted.
First adjacent inversion at index inv_idx = 1 (A[1] = 64738008, A[2] = 29805289).
Unsorted segment = [1, 2], length = 2.
[TIME] total elapsed = 11.078357 sec
═══════ sort_verify_mpi_d.out (MPI-Dynamic, 8 ranks) ═══════
Array is NOT sorted.
First adjacent inversion at index inv_idx = 1 (A[1] = 64738008, A[2] = 29805289).
Unsorted segment = [1, 2], length = 2.
[TIME] total elapsed = 11.053894 sec

```

> 這些結果告訴我們，目前四種版本都被文字解析綁住：我把 1.4 GB 字串轉整數就花了 8 秒，所以不管後面是 8 thread 還是 8 rank，牆鐘時間差不多。MPI 因為重複 broadcast + PMI 問題，甚至更慢還出現偽陽性。