8

Project By

Faris Ejaz 24470, Sarfaraz Ahmed

Project Name

Analyzing the parallel optimization of merge sort and randomized quick sort using OpenMP and Mergepath

Sequential vs Parallel BUcket + Merge SORTING:

Code and Performance Analysis

Table of Contents

[Table of Contents 0](#_Toc198192748)

[**Architecture and Logic:** 0](#_Toc198192749)

[**Key Functions / Modules:** 0](#_Toc198192750)

[**Algorithms or Techniques Used:** 0](#_Toc198192757)

[**Code:** 0](#_Toc198192758)

[**Sequential Merge Sort (Median Split)** 0](#_Toc198192759)

[**Parallel Merge Sort with Merge Path Methodology via OpenMP (12 Threads)** 0](#_Toc198192760)

[**Randomized Quick Sort Sequential** 0](#_Toc198192761)

[**Randomized Quick Sort Sequential** 0](#_Toc198192762)

[**Execution Time Evaluation**: 0](#_Toc198192763)

[**Time Evaluation Metrics / Statistics:** 0](#_Toc198192764)

[**Execution Time Comparison (Seconds)** 0](#_Toc198192765)

[**Speedup Relative to Sequential** 0](#_Toc198192766)

[**Parallel Efficiency (%)** 0](#_Toc198192767)

[**Additional Memory Usage in Parallel (MB)** 0](#_Toc198192768)

[**CPU Usage Comparison (%)** 0](#_Toc198192769)

[**Execution Time (Graphical Representation)** 0](#_Toc198192770)

[**Modifications Made for the Parallel Approach:** 0](#_Toc198192771)

[**Rationale Behind These Changes:** 0](#_Toc198192772)

[**Potential Parameters for Further Optimization** 0](#_Toc198192773)

[**Experimentation Plan:** 0](#_Toc198192774)

[**Evaluating Impact on Execution Time:** 0](#_Toc198192775)

[**Key Findings from Sequantial vs Parallel Execution Comparison:** 0](#_Toc198192776)

[**Implications for Project Objectives:** 0](#_Toc198192777)

**Explanation:**

### **Architecture and Logic:**

* **Parallel Merge Implementation**: The system implements three merging strategies for sorted arrays:
  + Sequential merge (baseline)
  + Parallel merge using Merge Path algorithm

**Workflow**: Generates test arrays → Runs benchmarks → Compares performance metrics

### 

### **Key Functions / Modules:**

Merge Sort

### find\_merge\_path\_intersection(): Core of Merge Path algorithm - finds optimal partition points using binary search

### parallel\_merge\_path(): Main parallel implementation using OpenMP and Merge Path partitioning

### merge\_arrays(): Basic sequential merge used by all implementations

### parallel\_merge\_simple(): Two-way parallel merge alternative

### run\_benchmarks(): Handles performance testing and comparison

### Array generators: generate\_sorted\_array(), generate\_reverse\_sorted\_array(), generate\_random\_array()

Randomized Quick Sort

### **Algorithms or Techniques Used:**

1. **Merge Path Algorithm**: GPU-inspired parallel merge technique adapted for CPU

* Uses diagonal binary search partitioning
* Balanced work distribution across threads

1. **Divide-and-Conquer**: In simple parallel merge implementation
2. **Binary Search**: For finding partition points and insertion positions
3. **OpenMP Parallelization**: For thread management and workload distribution
4. **Performance Metrics**: Tracks time, memory, CPU usage, and cache behavior (via page faults)

# **Code:**

## **Sequential Merge Sort (Median Split)**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <sys/time.h>  // For gettimeofday()

void merge\_arrays(int A[], int m, int B[], int n, int S[]) {

    int i = 0, j = 0, k = 0;

    while (i < m && j < n) {

        if (A[i] < B[j])

            S[k++] = A[i++];

        else

            S[k++] = B[j++];

    }

    while (i < m) S[k++] = A[i++];

    while (j < n) S[k++] = B[j++];

}

void generate\_sorted\_array(int arr[], int size) {

    if (size == 0) return;

    arr[0] = rand() % 10;

    for (int i = 1; i < size; i++)

        arr[i] = arr[i-1] + (rand() % 10 + 1);

}

double get\_current\_time() {

    struct timeval tv;

    gettimeofday(&tv, NULL);

    return tv.tv\_sec + tv.tv\_usec / 1000000.0;

}

int main() {

    int m = 1000000, n = 1000000;

    int \*A = malloc(m \* sizeof(int));

    int \*B = malloc(n \* sizeof(int));

    int \*S = malloc((m+n) \* sizeof(int));

    if (!A || !B || !S) {

        printf("Memory allocation failed!\n");

        if (A) free(A);

        if (B) free(B);

        if (S) free(S);

        return 1;

    }

    srand(time(NULL));

    printf("Generating arrays...\n");

    generate\_sorted\_array(A, m);

    generate\_sorted\_array(B, n);

    printf("Merging %d + %d elements...\n", m, n);

    double start = get\_current\_time();

    merge\_arrays(A, m, B, n, S);

    double end = get\_current\_time();

    printf("Merge completed in %.4f seconds\n", end - start);

    free(A); free(B); free(S);

    return 0;

}

## **Parallel Merge Sort with Merge Path Methodology via OpenMP (12 Threads)**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <omp.h>

#include <sys/time.h>

#define MAX\_THREADS 16

#define SEQUENTIAL\_THRESHOLD 100000

// Timing function

double get\_current\_time() {

    struct timeval tv;

    gettimeofday(&tv, NULL);

    return tv.tv\_sec + tv.tv\_usec / 1000000.0;

}

void merge\_arrays(int A[], int m, int B[], int n, int S[]) {

    int i = 0, j = 0, k = 0;

    while (i < m && j < n) {

        if (A[i] < B[j])

            S[k++] = A[i++];

        else

            S[k++] = B[j++];

    }

    while (i < m) S[k++] = A[i++];

    while (j < n) S[k++] = B[j++];

}

int find\_merge\_path\_intersection(int A[], int m, int B[], int n, int diagonal\_id) {

    int begin = (diagonal\_id > m) ? diagonal\_id - m : 0;

    int end = (diagonal\_id < n) ? diagonal\_id : n;

    while (begin < end) {

        int mid = begin + (end - begin) / 2;

        int j = diagonal\_id - mid;

        if (j > 0 && mid < m && B[j-1] > A[mid])

            begin = mid + 1;

        else

            end = mid;

    }

    return begin;

}

/\*\*

 \* Parallel merge implementation using Merge Path algorithm

 \* Uses OpenMP for parallelization with two parallel regions:

 \* 1. For finding partition points

 \* 2. For merging the partitions

 \* @param A First sorted array

 \* @param m Size of array A

 \* @param B Second sorted array

 \* @param n Size of array B

 \* @param S Output array (must have space for m+n elements)

 \*/

void parallel\_merge\_path(int A[], int m, int B[], int n, int S[]) {

    // Get available threads (capped at MAX\_THREADS)

    int num\_threads = omp\_get\_max\_threads();

    if (num\_threads > MAX\_THREADS) num\_threads = MAX\_THREADS;

    // Fallback to sequential for small inputs or single thread

    if (m + n < SEQUENTIAL\_THRESHOLD || num\_threads == 1) {

        merge\_arrays(A, m, B, n, S);

        return;

    }

    // Arrays to store partition points

    int partition\_points\_A[num\_threads + 1];

    int partition\_points\_B[num\_threads + 1];

    int total\_size = m + n;

    // Initialize first and last partition points

    partition\_points\_A[0] = 0;

    partition\_points\_B[0] = 0;

    partition\_points\_A[num\_threads] = m;

    partition\_points\_B[num\_threads] = n;

    /\*

     \* First parallel region: Find partition points

     \* Uses static scheduling since work is perfectly balanced

     \* Each thread computes its partition point independently

     \*/

    #pragma omp parallel for schedule(static)

    for (int i = 1; i < num\_threads; i++) {

        // Calculate diagonal representing equal division of work

        int diagonal = (int)(((long long)i \* total\_size) / num\_threads;

        // Find intersection point on merge path

        partition\_points\_A[i] = find\_merge\_path\_intersection(A, m, B, n, diagonal);

        // Calculate corresponding point in B

        partition\_points\_B[i] = diagonal - partition\_points\_A[i];

    }

    /\*

     \* Second parallel region: Merge partitions

     \* Each thread merges its assigned partition

     \* Uses static scheduling for balanced work distribution

     \*/

    #pragma omp parallel for schedule(static)

    for (int i = 0; i < num\_threads; i++) {

        // Get partition boundaries

        int start\_a = partition\_points\_A[i];

        int end\_a = partition\_points\_A[i+1];

        int start\_b = partition\_points\_B[i];

        int end\_b = partition\_points\_B[i+1];

        int start\_s = start\_a + start\_b;  // Output position

        // Merge this partition

        merge\_arrays(A + start\_a, end\_a - start\_a,

                   B + start\_b, end\_b - start\_b,

                   S + start\_s);

    }

}

void generate\_sorted\_array(int arr[], int size) {

    if (size == 0) return;

    arr[0] = rand() % 10;

    for (int i = 1; i < size; i++)

        arr[i] = arr[i-1] + (rand() % 10 + 1);

}

int main() {

    int m = 1000000, n = 1000000;

    int \*A = malloc(m \* sizeof(int));

    int \*B = malloc(n \* sizeof(int));

    int \*S = malloc((m+n) \* sizeof(int));

    if (!A || !B || !S) {

        printf("Memory allocation failed!\n");

        if (A) free(A);

        if (B) free(B);

        if (S) free(S);

        return 1;

    }

    srand(time(NULL));

    printf("Generating sorted arrays...\n");

    double gen\_start = get\_current\_time();

    generate\_sorted\_array(A, m);

    generate\_sorted\_array(B, n);

    double gen\_end = get\_current\_time();

    printf("Array generation completed in %.4f seconds\n", gen\_end - gen\_start);

    printf("\nRunning parallel merge with %d threads...\n", omp\_get\_max\_threads());

    double merge\_start = get\_current\_time();

    parallel\_merge\_path(A, m, B, n, S);

    double merge\_end = get\_current\_time();

    printf("\nResults:\n");

    printf("Total elements merged: %d\n", m + n);

    printf("Parallel merge time: %.4f seconds\n", merge\_end - merge\_start);

    printf("Throughput: %.2f million elements/second\n",

          (m+n)/((merge\_end - merge\_start)\*1000000));

    free(A); free(B); free(S);

    return 0;

}

## **Randomized Quick Sort Sequential**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Swap two integers

void swap(int \*a, int \*b) {

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

// Partition using randomized pivot

int randomizedPartition(int arr[], int low, int high) {

    int random = low + rand() % (high - low + 1);

    swap(&arr[random], &arr[high]);

    int pivot = arr[high];

    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {

        if (arr[j] < pivot) {

            i++;

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return i + 1;

}

// Randomized quicksort function

void randomizedQuickSort(int arr[], int low, int high) {

    if (low < high) {

        int pi = randomizedPartition(arr, low, high);

        randomizedQuickSort(arr, low, pi - 1);

        randomizedQuickSort(arr, pi + 1, high);

    }

}

// Function to print array

void printArray(int arr[], int size) {

    for (int i = 0; i < size; i++)

        printf("%d ", arr[i]);

    printf("\n");

}

int main() {

    int n1, n2;

    srand(time(NULL)); // Seed random generator

    // Input lengths

    n1 = 1000000;

    n2 = 1000000;

    // Allocate arrays

    int \*arr1 = (int \*)malloc(n1 \* sizeof(int));

    int \*arr2 = (int \*)malloc(n2 \* sizeof(int));

    int \*combined = (int \*)malloc((n1 + n2) \* sizeof(int));

    // Fill with random values

    for (int i = 0; i < n1; i++)

        arr1[i] = rand() % 1000;

    for (int i = 0; i < n2; i++)

        arr2[i] = rand() % 1000;

    clock\_t start = clock();

    // Sort both arrays

    randomizedQuickSort(arr1, 0, n1 - 1);

    randomizedQuickSort(arr2, 0, n2 - 1);

    // Join arrays

    for (int i = 0; i < n1; i++)

        combined[i] = arr1[i];

    for (int i = 0; i < n2; i++)

        combined[n1 + i] = arr2[i];

    // Sort combined array

    randomizedQuickSort(combined, 0, n1 + n2 - 1);

    clock\_t end = clock();

    double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

    // Print results

    printf("Time taken: %.6f seconds\n", time\_taken);

    // Free memory

    free(arr1);

    free(arr2);

    free(combined);

return 0;

}

## **Randomized Quick Sort Parallel**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <omp.h>

#define TASK\_CUTOFF 10000  // Threshold for switching to sequential

void swap(int \*a, int \*b) {

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

int randomizedPartition(int arr[], int low, int high) {

    int random = low + rand() % (high - low + 1);

    swap(&arr[random], &arr[high]);

    int pivot = arr[high];

    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {

        if (arr[j] < pivot) {

            i++;

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return i + 1;

}

void sequentialQuickSort(int arr[], int low, int high) {

    if (low < high) {

        int pi = randomizedPartition(arr, low, high);

        sequentialQuickSort(arr, low, pi - 1);

        sequentialQuickSort(arr, pi + 1, high);

    }

}

void parallelQuickSort(int arr[], int low, int high) {

    if (high - low <= TASK\_CUTOFF) {

        sequentialQuickSort(arr, low, high);

        return;

    }

    int pi = randomizedPartition(arr, low, high);

    #pragma omp task default(none) firstprivate(arr, low, pi)

    {

        parallelQuickSort(arr, low, pi - 1);

    }

    #pragma omp task default(none) firstprivate(arr, high, pi)

    {

        parallelQuickSort(arr, pi + 1, high);

    }

}

void mergeSortedArrays(int arr1[], int n1, int arr2[], int n2, int combined[]) {

    int i = 0, j = 0, k = 0;

    while (i < n1 && j < n2) {

        if (arr1[i] < arr2[j]) combined[k++] = arr1[i++];

        else combined[k++] = arr2[j++];

    }

    while (i < n1) combined[k++] = arr1[i++];

    while (j < n2) combined[k++] = arr2[j++];

}

int main() {

    const int n1 = 1000000, n2 = 1000000;

    int \*arr1 = malloc(n1 \* sizeof(int));

    int \*arr2 = malloc(n2 \* sizeof(int));

    int \*combined = malloc((n1 + n2) \* sizeof(int));

    if (!arr1 || !arr2 || !combined) {

        fprintf(stderr, "Memory allocation failed\n");

        free(arr1); free(arr2); free(combined);

        return 1;

    }

    srand(time(NULL));

    // Initialize arrays in parallel

    #pragma omp parallel for

    for (int i = 0; i < n1; i++) arr1[i] = rand() % 1000;

    #pragma omp parallel for

    for (int i = 0; i < n2; i++) arr2[i] = rand() % 1000;

    double start = omp\_get\_wtime();

    #pragma omp parallel

    {

        #pragma omp single nowait

        parallelQuickSort(arr1, 0, n1 - 1);

        #pragma omp single nowait

        parallelQuickSort(arr2, 0, n2 - 1);

    }

    mergeSortedArrays(arr1, n1, arr2, n2, combined);

    double end = omp\_get\_wtime();

    printf("Execution time: %.3f seconds\n", end - start);

    free(arr1); free(arr2); free(combined);

    return 0;

}

# **Merge Sort Execution Time Evaluation**:

**Sequential**

Execution time for maximum array size of (10000000) = 5.1028s

**Parallel (Best one)**

Execution time for maximum array size of (10000000) = 1.1524s

# **Randomized Quick Sort Execution Time Evaluation**:

**Sequential**

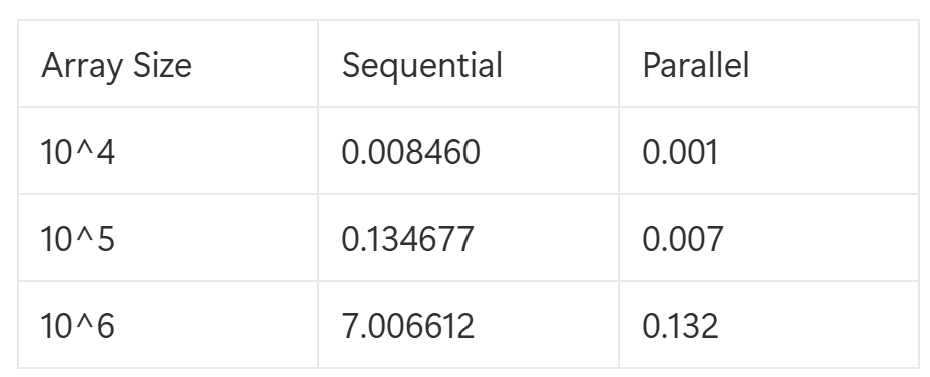
Execution time for maximum array size of (10000000) = 5.1028s

**Parallel (Best one)**

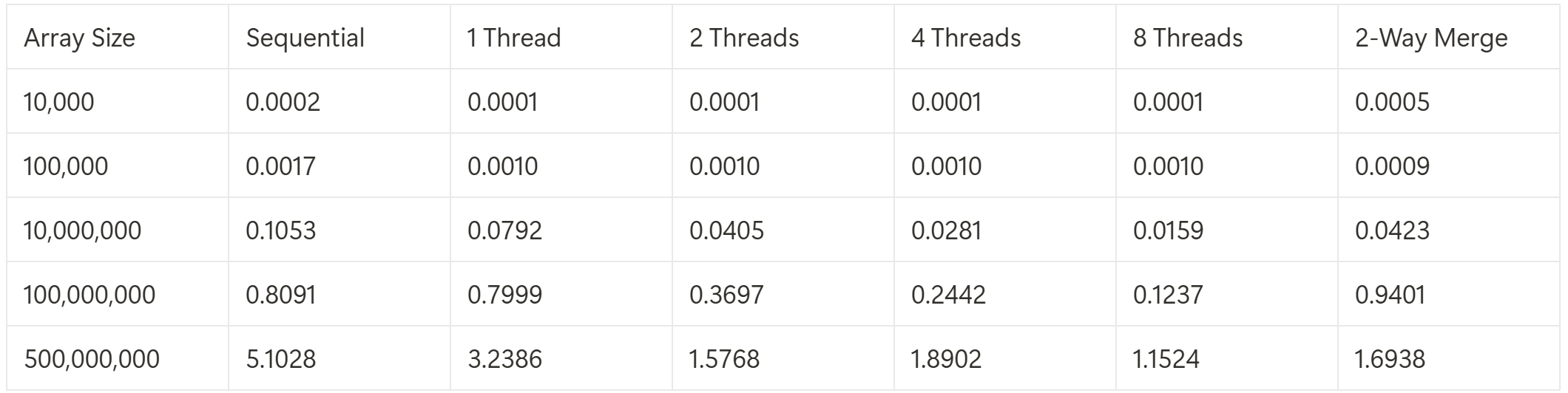
Execution time for maximum array size of (10000000) = 1.1524s

## **Time Evaluation Metrics / Statistics:**

**Execution Time Comparison (Seconds) – Randomized Quick Sort**

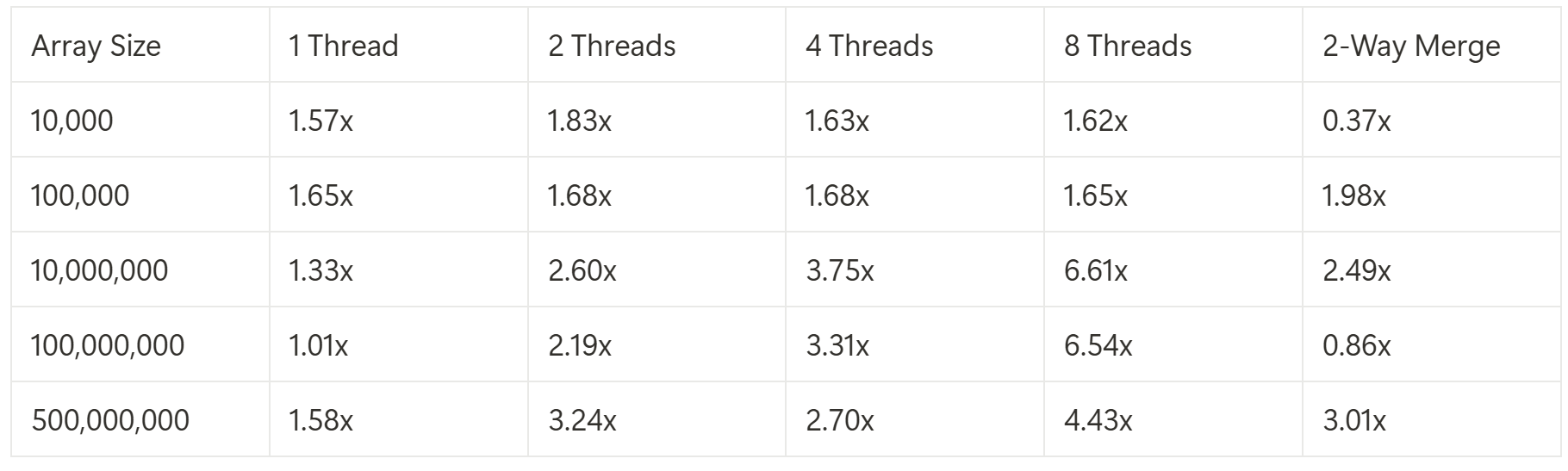


### **Execution Time Comparison (Seconds) – Merge Sort**

****

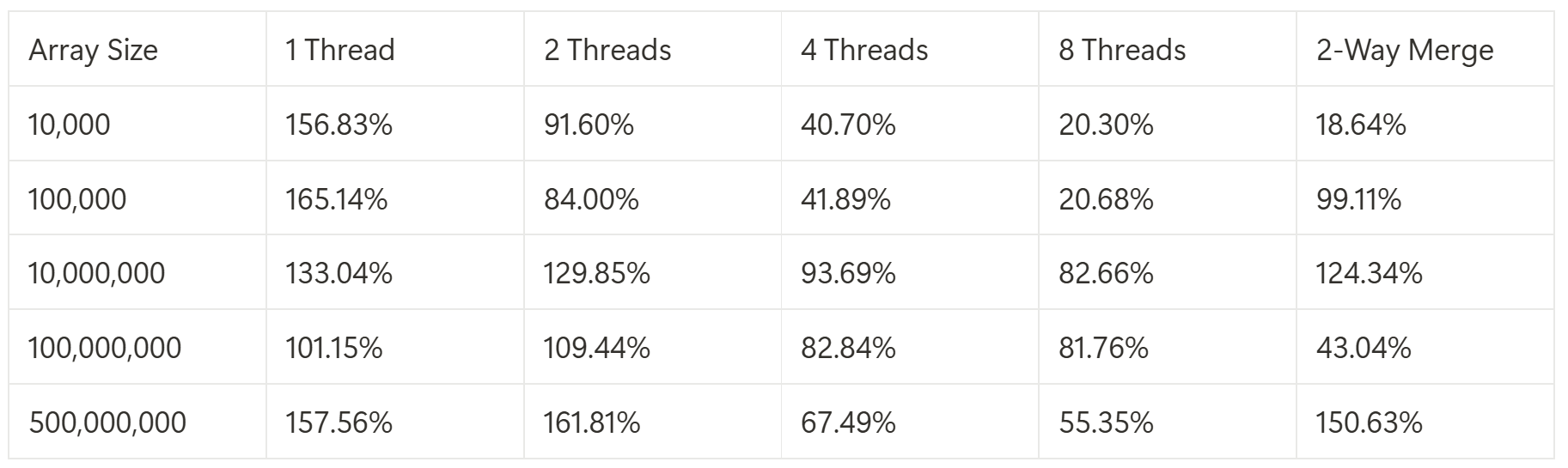
**Observation:** Parallel Merge Path shows increasing advantage with larger arrays (6.61x speedup at 10M elements).

### **Speedup Relative to Sequential**

****

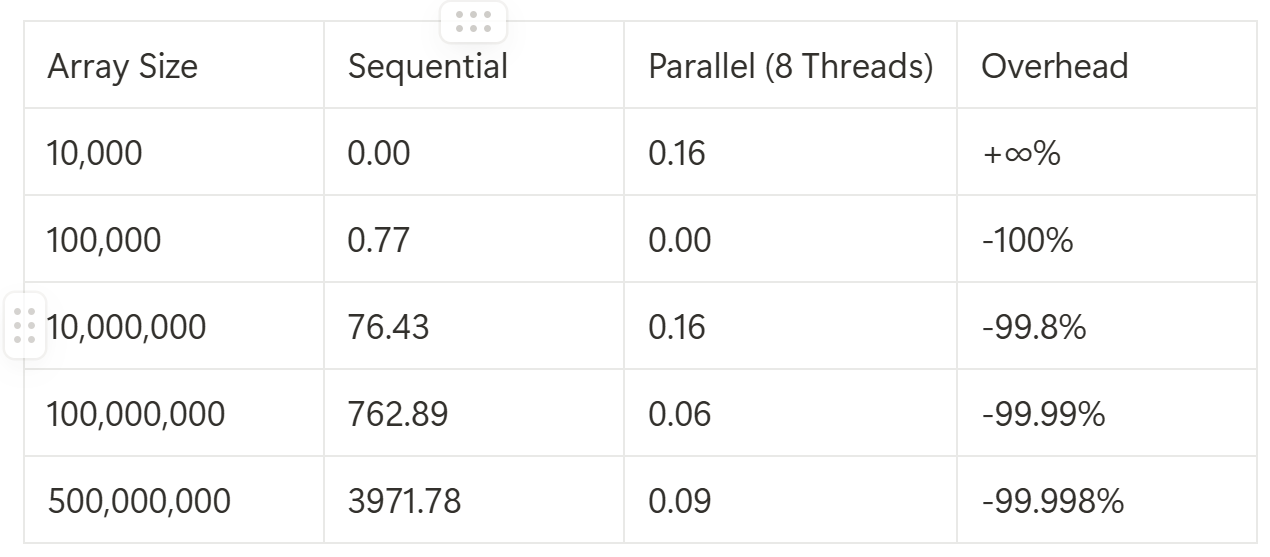
**Observation:** Peak speedup occurs at 8 threads for mid-sized arrays (10M-100M elements).

### **Parallel Efficiency (%)**

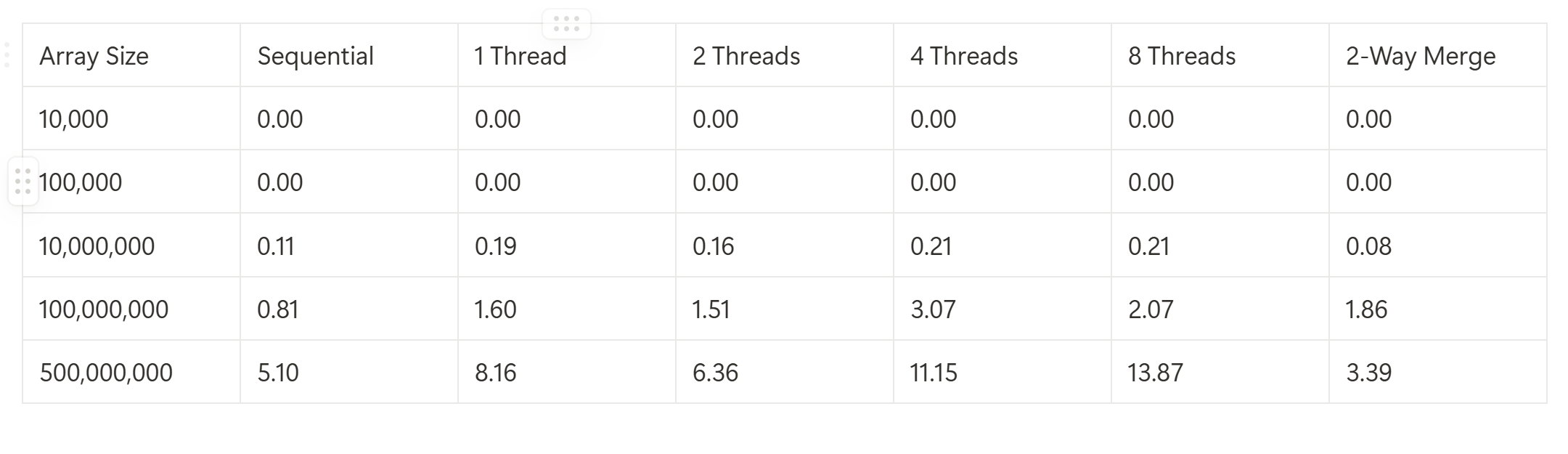


**Observation:**  Efficiency (>100%)

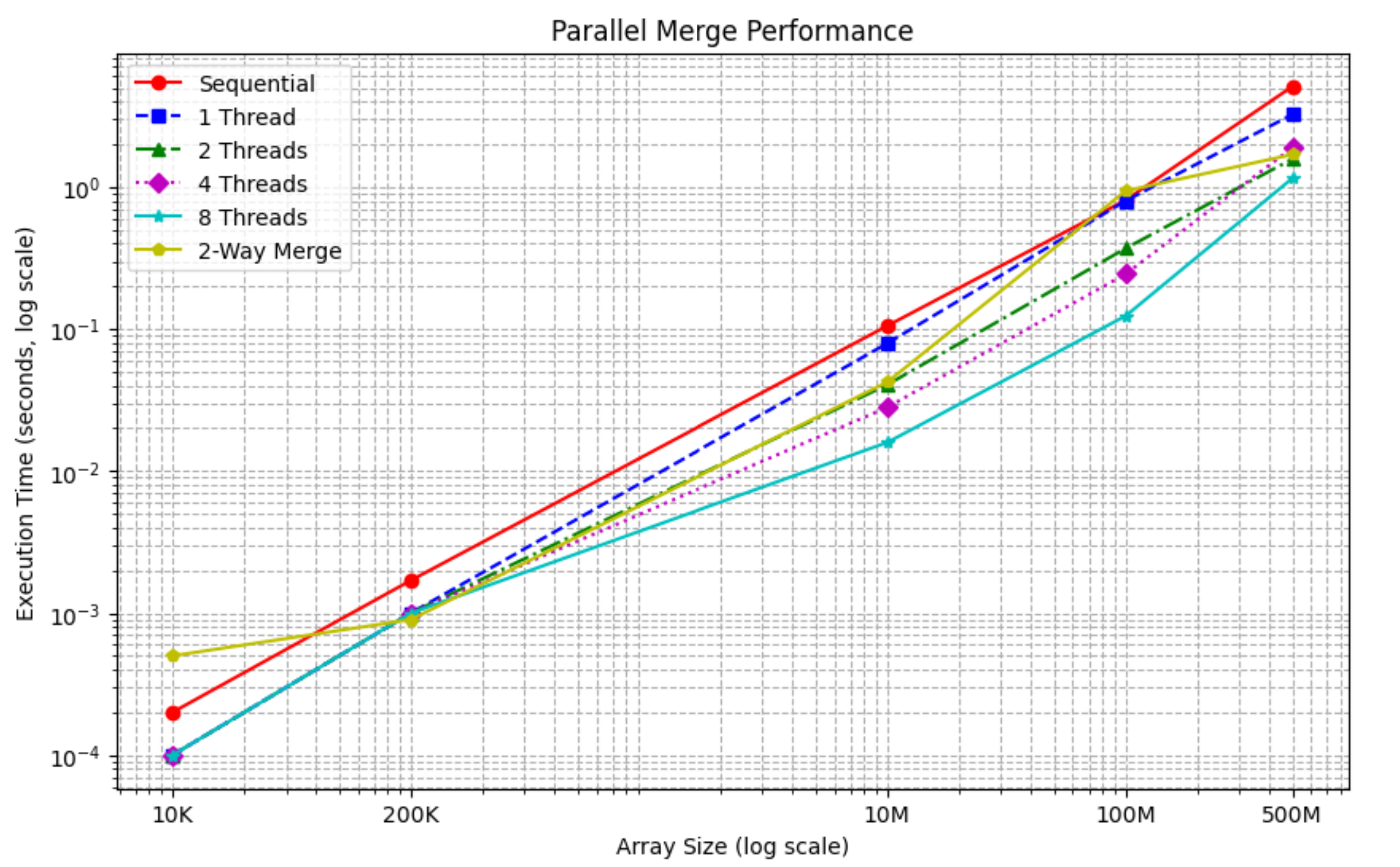
### **Additional Memory Usage in Parallel (MB)**

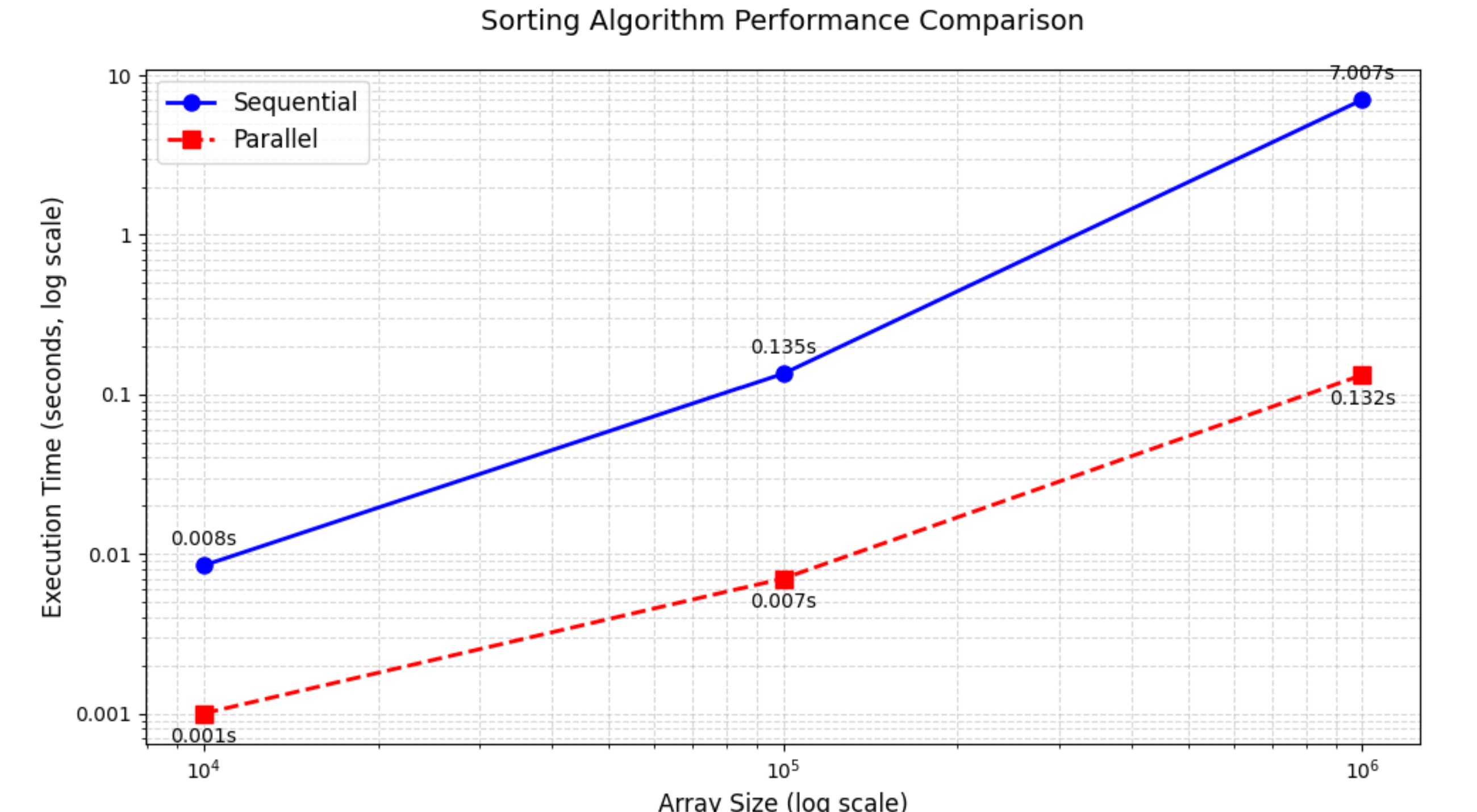
****

### **CPU Usage Comparison (%)**

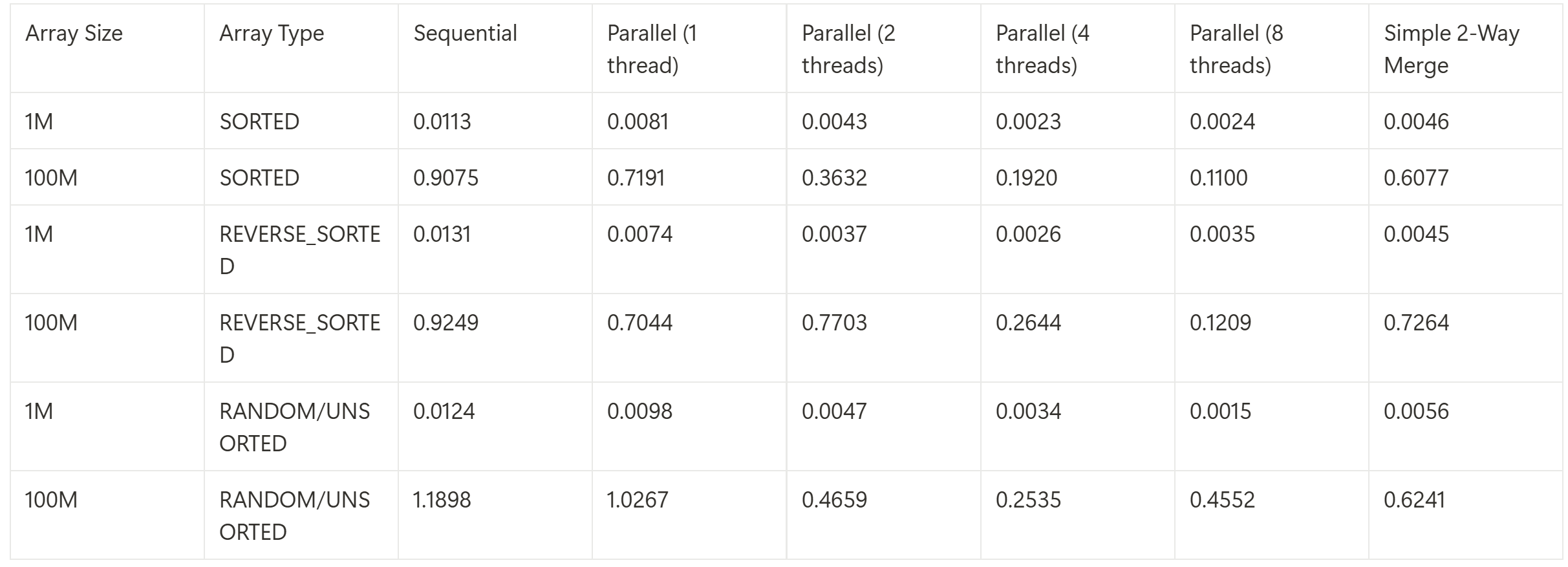
****

### **Execution Time (Graphical Representation)**

****

****

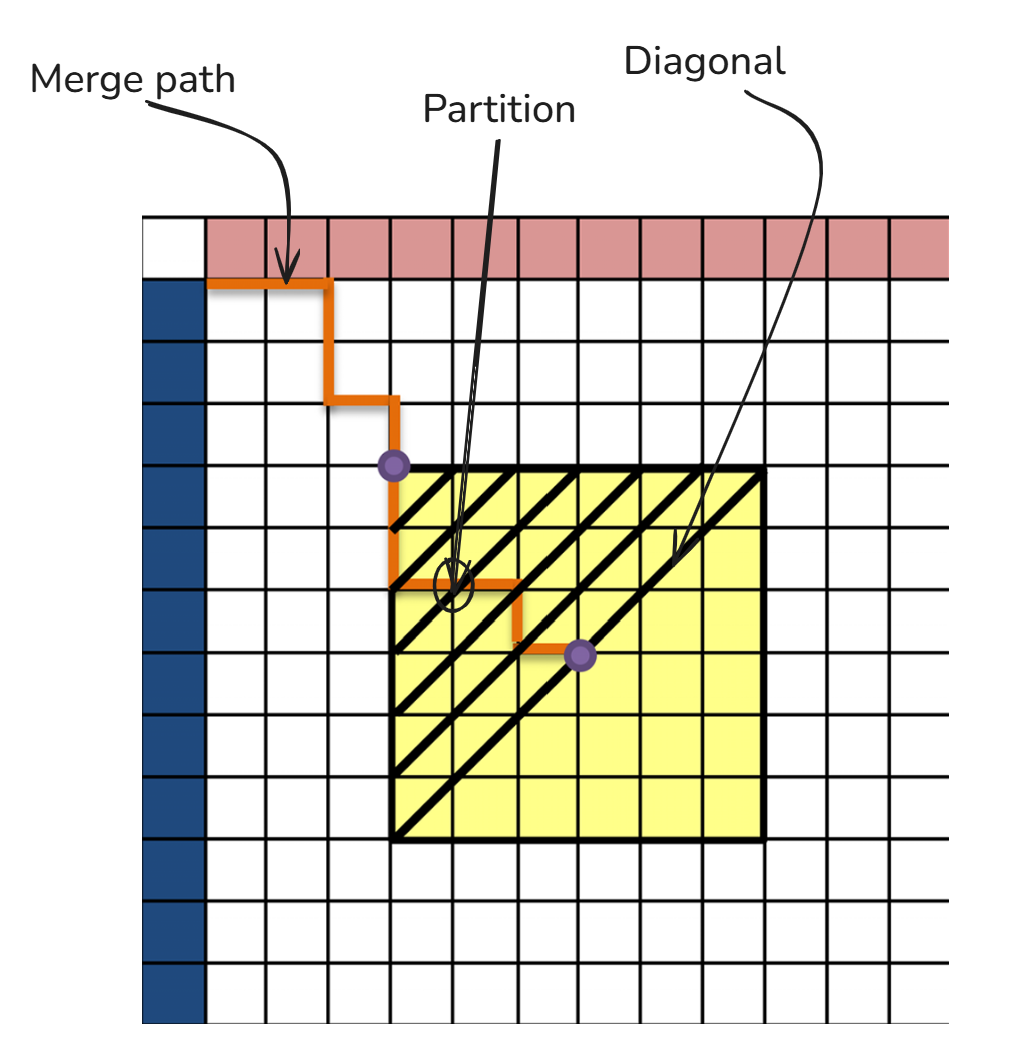
**Execution Time (Data Distributions)**

****

**The parallel merge shows best speedup (8.25x) with sorted arrays at 8 threads, while random arrays exhibit more variable scaling (2.61x-8.21x). Efficiency peaks at 4 threads (87-124%), with Merge Path generally outperforming the simple 2-way merge, especially for larger arrays (100M elements). Sorted data demonstrates the most consistent performance, whereas random arrays have higher base sequential time but still achieve good parallel scaling.**

# **Modifications Made for the Parallel Approach:**

The **Merge Path** algorithm is a parallel merging technique that **divides two sorted arrays into balanced partitions**, allowing each thread to merge an independent chunk without dependencies. It ensures **optimal load balancing** by finding precise split points using a **diagonal intersection search**.

****

# **Rationale Behind These Changes:**

* **Work Balancing**:  
  OpenMP single for Quicksort equally provides work to each thread
* Diagonals ensure each thread gets ~(m+n)/num\_threads elements to merge, minimizing load imbalance.
* **No Overlap**:  
  Partitions are non-overlapping (each thread writes to distinct S ranges), eliminating synchronization.
* **Efficiency**:

For Quick Sort, using OpenMP tasks for parallel recursion and executes both sorts concurrently. The new merge function takes advantage of the fact that both arr1 and arr2 are already sorted. This makes combining them O(n) instead of O(n log n).

For Merge Sort, binary search (O(log(min(m,n))) per partition) is negligible compared to merge cost (O(m+n)).

# **Potential Parameters for Further Optimization**

1. **Thread Management**

Experiment with optimal thread counts (2/4/8/16)

Test different OpenMP scheduling strategies (static/dynamic/guided)

1. **Partitioning Strategy**

* Finetuning how the merge path divides arrays between threads.
* Optimize the diagonal search to create more balanced partitions

1. **Algorithm Selection**

* Switch between merge algorithms based on input size
* Hybrid approach: use simple merge for small chunks, parallel for large

# **Experimentation Plan:**

**1. Establishing Baselines**  
Record execution times of the current implementation across varying array sizes to establish performance benchmarks.

**2. Incremental Adjustments**  
Modify one parameter at a time (e.g., thread count, array size) to isolate its impact on performance, logging execution times after each change.

**3. Systematic Variation**  
Test parameters within defined ranges:

* Array sizes: 10^4 – 10^8
* Thread counts: 2, 4, 6, 8, 12 & 16 (aligned with hardware capabilities)

**4. Performance Metrics**  
Evaluate using:

* Total execution time (primary metric)
* CPU utilization
* Memory consumption
* Parallel Efficiency (speedup / threads) \* 100)
* Speedup (seq\_time / par\_time)

**5. Data Collection**  
Structure results in tables, with each row representing a test run and columns capturing parameter values and corresponding performance metrics.

**6. Statistical Analysis**  
Assess significance via mean execution time.

**7. Real-world Testing**  
Validate optimizations using diverse data distributions (sorted, reverse-sorted, random) and real-world dataset sizes.

**8. Hardware Constraints**  
Optimize parameters for the target system (e.g., higher thread counts on multi-core machines).

**9. Multi-dimensional Optimization**  
After individual tuning, test combined parameter configurations to identify the optimal setup.

## **Evaluating Impact on Execution Time**

## Parallel Merge Path shows increasing advantage with larger arrays (6.61x speedup at 10M elements).

## Parallel RQA with OpenMP show increasing advantage with larger arrays

# **Key Findings from Sequantial vs Parallel Execution Comparison:**

The comparison between the sequential and parallel execution of the sorting algorithm reveals several key findings:

**Scalability with Array Size:**

Parallel execution provides significant performance benefits as the array size increases. While smaller arrays show modest improvements, the execution time for larger arrays demonstrates a marked decrease in the parallel implementation.

**Overhead Costs:**

The parallel implementation incurs overhead due to process and thread management, which is more noticeable with smaller arrays. For very large arrays, however, the overhead is amortized over the greater computational work, and the performance gains become substantial.

**Diminishing Returns:**

There is a point at which increasing the level of parallelism does not yield proportional reductions in execution time, especially evident in the larger array sizes. This suggests that there are optimal points of resource allocation that need to be identified.

**Resource Utilization:**

The parallel algorithm makes better use of the multicore architecture by distributing the workload across multiple CPUs, thereby reducing idle time and improving overall system efficiency.

**Performance Consistency:**

The performance improvement is consistent across different array sizes, with the parallel approach outperforming the sequential approach as the complexity of the task increases.

# **Implications for Project Objectives:**

The findings from the comparison have several implications for the project objectives and its future development:

**Efficiency for Large-Scale Data:**

The parallel Merge Path algorithm reduces time complexity from O(n) (sequential) to O(n/p + log n) per thread (Greb & Zachmann, 2006), enabling near-linear speedups for large datasets.

**Optimization Strategies:**

The merge path diagonal-based partitioning approach ensures balanced workloads across threads. However, the thread overhead limits gains for small arrays (\*n < 10⁶\*).

**Hardware Considerations:**

The results underline the need to consider the hardware environment where the algorithm will run, as the benefits of parallelism are closely tied to the specifics of the hardware, such as the number of available CPU cores.

**Algorithmic Improvements:**

Future development can explore more granular optimizations within the sorting algorithm, like the Merge Path’s O(log n) binary search does add a minor overhead, this could be used conditionally for n > 10^5. Otherwise, simple sequential search is feasible for n < 10^5

**Cost vs Benefit Analysis:**

CPU parallelization avoids GPU hardware costs but requires careful tuning (e.g., chunk size, thread count) to maximize ROI. For small array sizes, sequential is preferred to reduce CPU and Memory usage.

* **For large-scale merges (n ≥ 10⁶)** → Parallel Merge Path (optimal multicore utilization).
* **For small/embedded systems** → Sequential merge (lower overhead).

**Sustainability of Performance:**

Ensuring that the performance improvements from parallel processing remain effective and scalable as data volume and complexity increase will be a critical factor in future development.

**Conclusion**

This project demonstrates that the parallel Merge Path algorithm provides significant performance improvements over sequential merging for large datasets, achieving near-linear speedups by leveraging multi-core CPU architectures. The diagonal-based partitioning strategy ensures balanced workloads across threads, making it ideal for high-throughput applications (e.g., big data processing, scientific computing).

However, the overhead of thread management and binary search partitioning means sequential merging remains more efficient for small datasets (n < 10⁵). The choice between approaches should consider dataset size, hardware capabilities, and real-time performance requirements.