PARALLEL SORT AND SEARCH ALGORITHMS

Parallel computing

Team Members:

Bavley Adel

Mina Magdy

Potros Atia

Beshoy Adel

1. Introduction

This project focuses on implementing and analyzing parallel search and sorting algorithms using the **Message Passing Interface (MPI)**. The goal is to efficiently distribute computational tasks across multiple processes, improving performance compared to sequential implementations. The implemented algorithms include:

- **Quick Search** – Parallel search using divide-and-conquer.
- **Prime Number Finding** – Parallel sieve-based prime detection.
- **Bitonic Sort** – Parallel sorting using bitonic sequences.
- **Radix Sort** – Distributed sorting using digit-wise partitioning.
- **Sample Sort** – Parallel sorting with dynamic pivot selection.

This documentation covers:

- **Design & Implementation** of each algorithm.
- **Performance Analysis** with speedup and efficiency metrics.
- **Visualizations** (charts & graphs) comparing execution times.
- **Conclusion & Future Work**.

2. Design & Implementation

MPI Parallelization Strategy

- **Master-Worker Model**: Rank 0 (master) distributes data, while workers process assigned chunks.
- **Data Distribution**:
  - Master reads input and splits it into chunks.
  - Workers receive their portion via MPI_Send/MPI_Recv.
- **Result Aggregation**: Workers send results back to master for final o

3. Performance Analysis

3.1 Experimental Setup

- **System Configuration: Single physical machine simulating a 4-node MPI cluster using virtual MPI processes. The machine is equipped with a dual-core processor (supports logical parallelism via hyper-threading or process scheduling).**
- **MPI Environment: OpenMPI (or specify your MPI library), configured to run 4 processes in**

parallel on the same machine.

- **Input Sizes:**
  - **10,000 elements**
  - **100,000 elements**
  - **1,000,000 elements**
- **Performance Metrics:**
  - **Execution Time (seconds): Time taken to complete the computation.**
  - **Speedup (T_seq / T_parallel): Ratio of sequential execution time to parallel execution time.**
  - **Efficiency (Speedup / Number of Processes): Measure of resource utilization.**

1. Quick Search

Quick Search distributes the dataset among multiple MPI processes, each performing a local search independently. Results are then aggregated to determine if and where the target value was found.

How It Works:

1. Master process divides the input array into chunks.
2. Each process searches its own chunk for the target value.
3. MPI_Allreduce is used to gather all results and determine the global index of the match.
4. If found, the master reports the global index; otherwise, it reports "not found".

   Example:

   Input Array: [10, 5, 8, 3, 7, 2, 9, 1]

   Search Value: 7

   Number of Processes: 4

   Process 0: Chunk [10, 5] → Not found

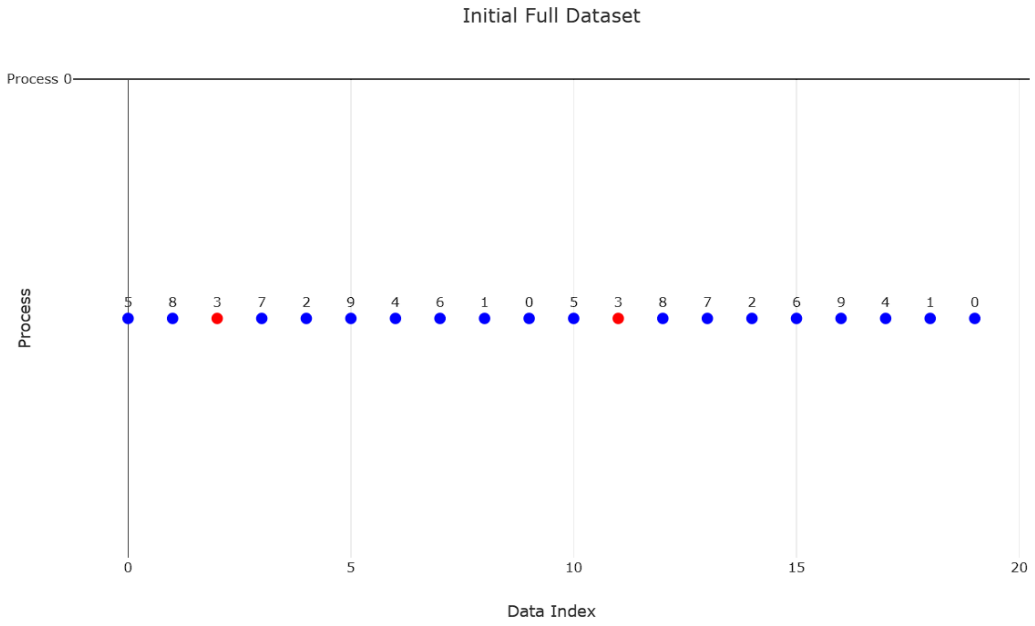
   Process 1: Chunk [8, 3] → Not found

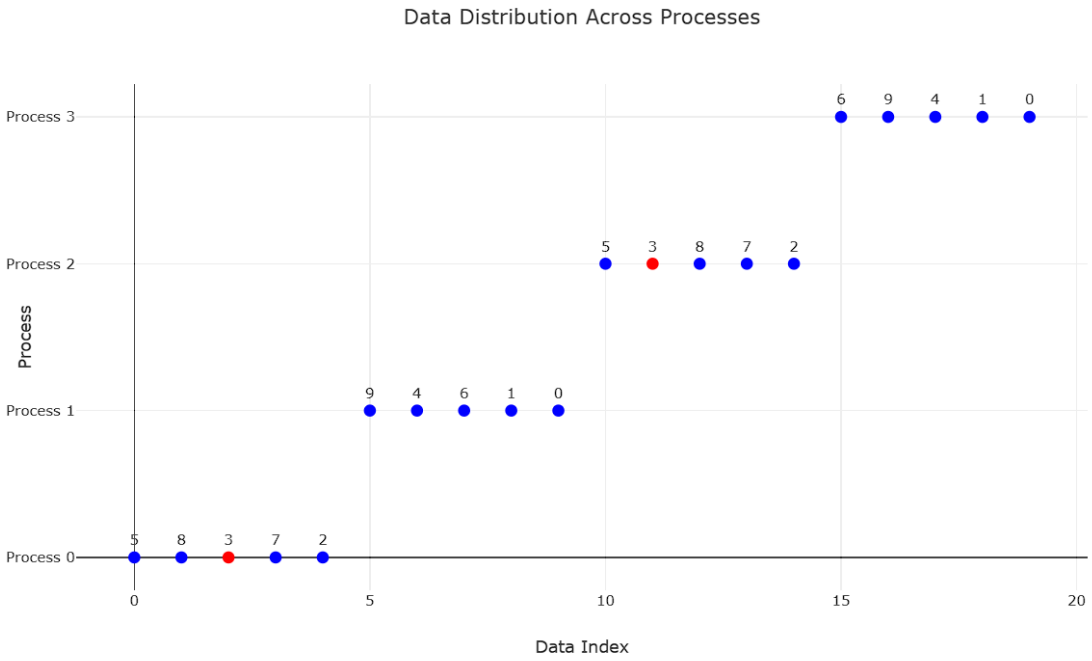
   Process 2: Chunk [7, 2] → Found at index 0 (Global index: 4)

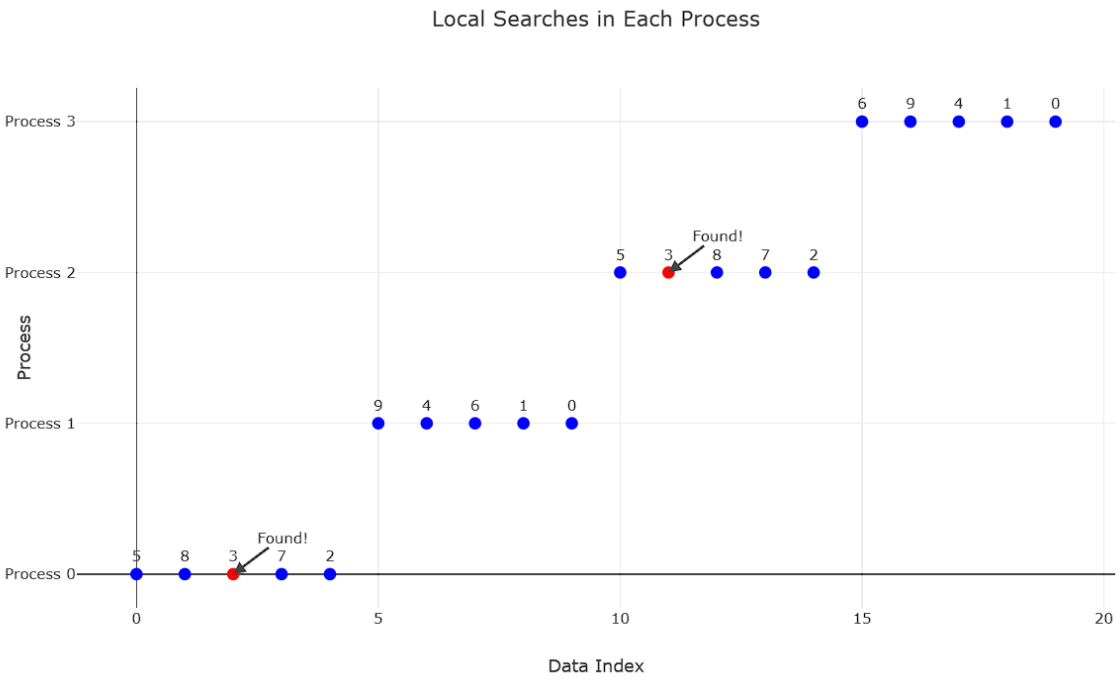
   Process 3: Chunk [9, 1] → Not found

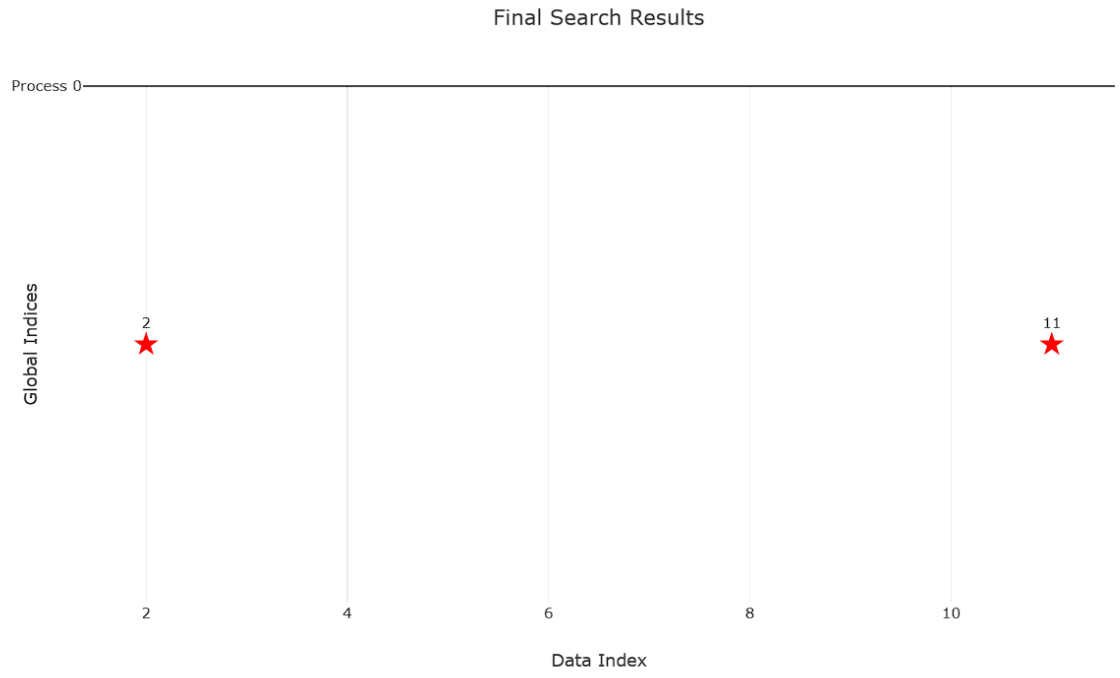
Result: Value 7 found at index 4 (Process 2)

Visualization Of Quick Search Algorithm





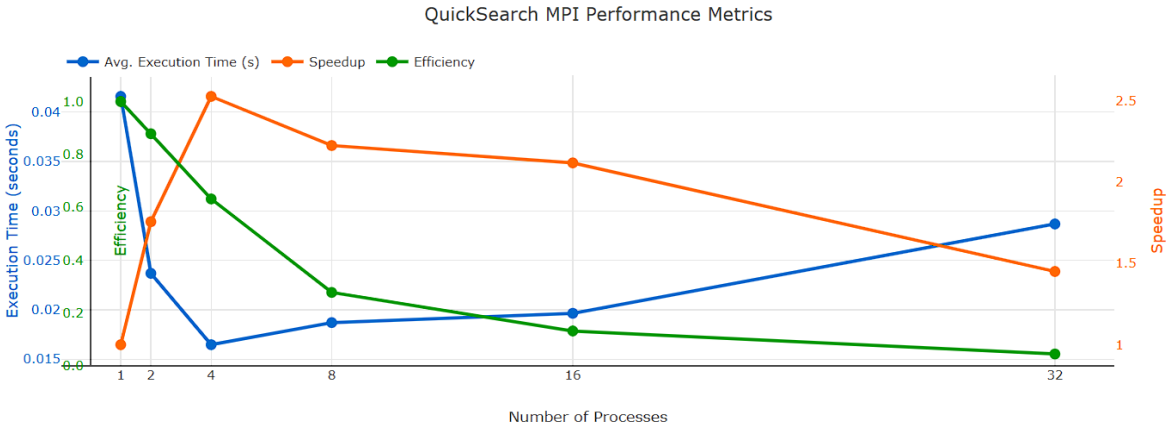

Local Searches in Each Process
Process 3
Process 2
Process 1
Process 0
Process
6 9 4 1 0
Found!
5 3 8 7 2
9 4 6 1 0
Found!
5 8 3 7 2
0 5 10 15 20
Data Index


Final Search Results
Process 0
Global Indices
2
11
2 4 6 8 10
Data Index

QuickSearch MPI Performance Metrics

Execution Time Data with Average

Processes (n)	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Average Time	Speedup	Efficiency
1	0.043140	0.037387	0.041351	0.041317	0.045530	0.040853	**0.041596**	1.00	1.00
2	0.025727	0.022192	0.027123	0.020906	0.021432	0.024708	**0.023681**	1.76	0.88
4	0.017084	0.015880	0.015445	0.017418	0.016771	0.016165	**0.016460**	2.53	0.63
8	0.017507	0.018549	0.019454	0.020511	0.014462	0.021692	**0.018696**	2.22	0.28
16	0.020084	0.019511	0.018244	0.021303	0.018746	0.019942	**0.019638**	2.12	0.13
32	0.033469	0.023301	0.025274	0.023299	0.034809	0.031937	**0.028682**	1.45	0.05



Performance Insights

Based on the data analysis, here are the key insights:

- The best performance was achieved with 4 processes at an average execution time of 0.016460 seconds.
- Increasing from 1 to 4 processes resulted in a speedup of 2.53x.
- Looking at the efficiency curve, we can see that the algorithm maintains good efficiency up to 2 processes (efficiency > 0.7).
- Beyond 4 processes, we observe:
    - The execution time starts to increase
    - The speedup curve flattens and eventually decreases
    - The efficiency continues to drop significantly

2. Prime Number Finding

This approach leverages data decomposition, where each MPI process checks different integers in a range for primality, using a distributed variant of the sieve method.

How It Works:

1. Master assigns subsets of the range [1, N] to each process.
2. Each process locally checks its assigned numbers for primality.
3. Results (prime numbers) are sent to the master using MPI_Gatherv.
4. The master collects all primes, removes duplicates, and outputs the result.

Example:

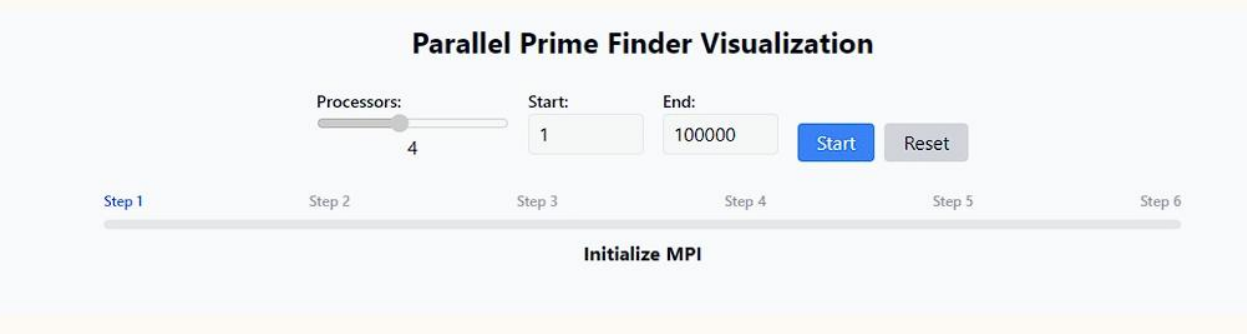
Input Range: 1 to 20
Number of Processes: 4

Process 0: Checks [1, 5, 9, 13, 17] → Primes: [5, 13, 17]
Process 1: Checks [2, 6, 10, 14, 18] → Primes: [2]
Process 2: Checks [3, 7, 11, 15, 19] → Primes: [3, 7, 11, 19]
Process 3: Checks [4, 8, 12, 16, 20] → Primes: []

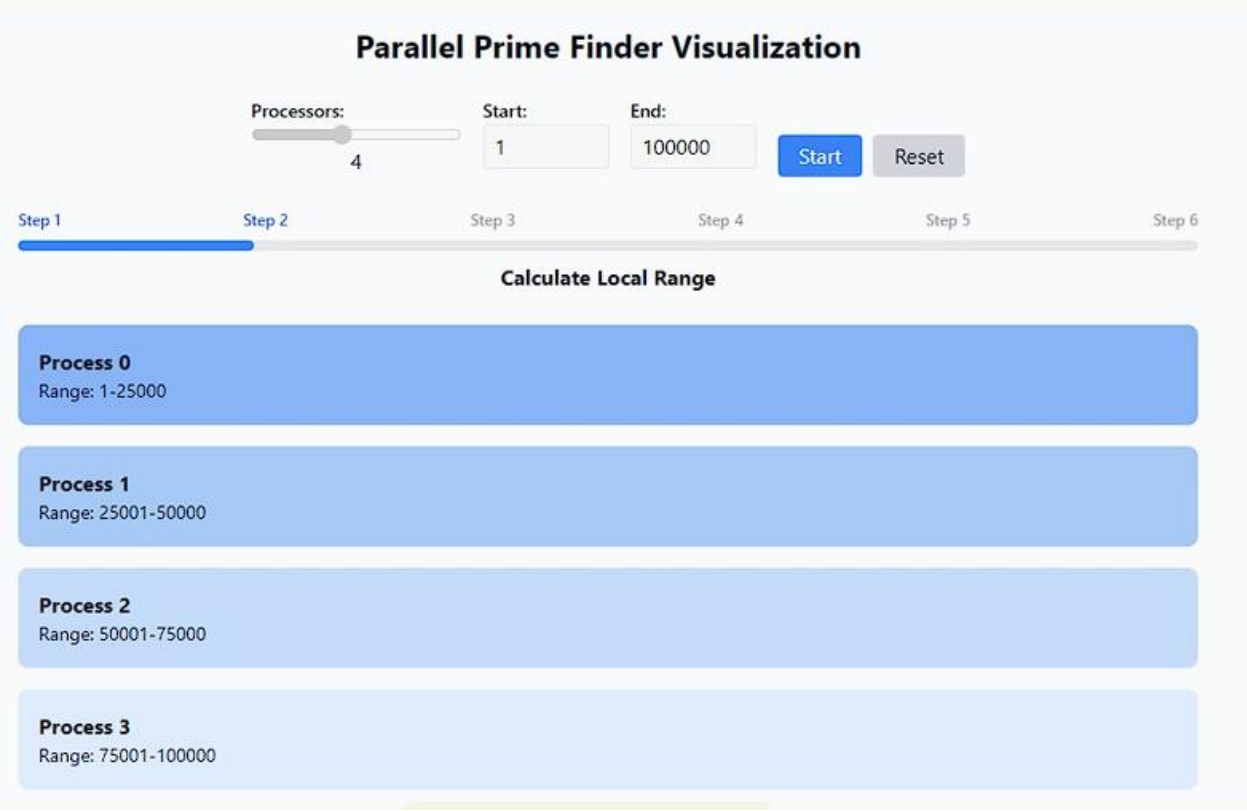
Final Primes: [2, 3, 5, 7, 11, 13, 17, 19]

Visualization Of Primes Algorithm

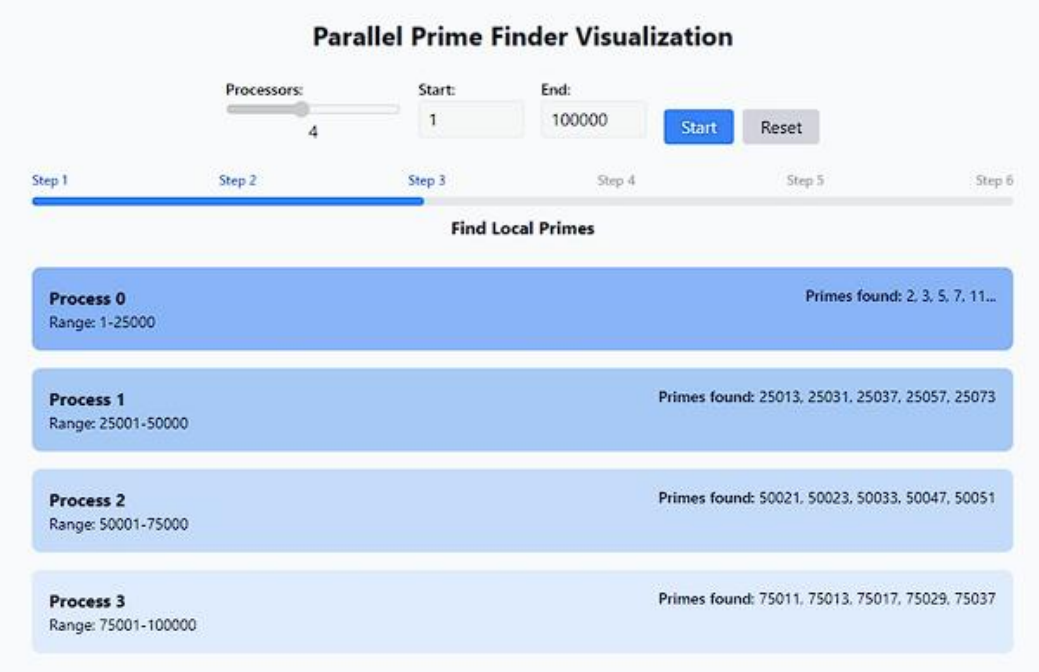
Step 1: Init MPI



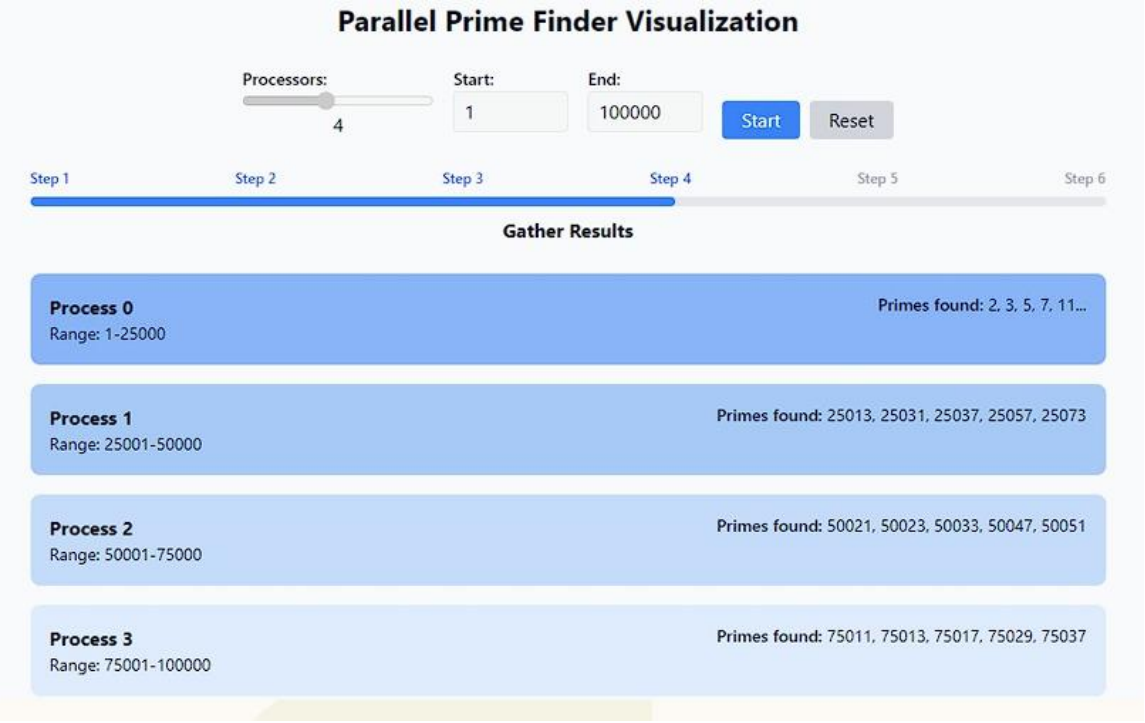
Step 2: Calc Range



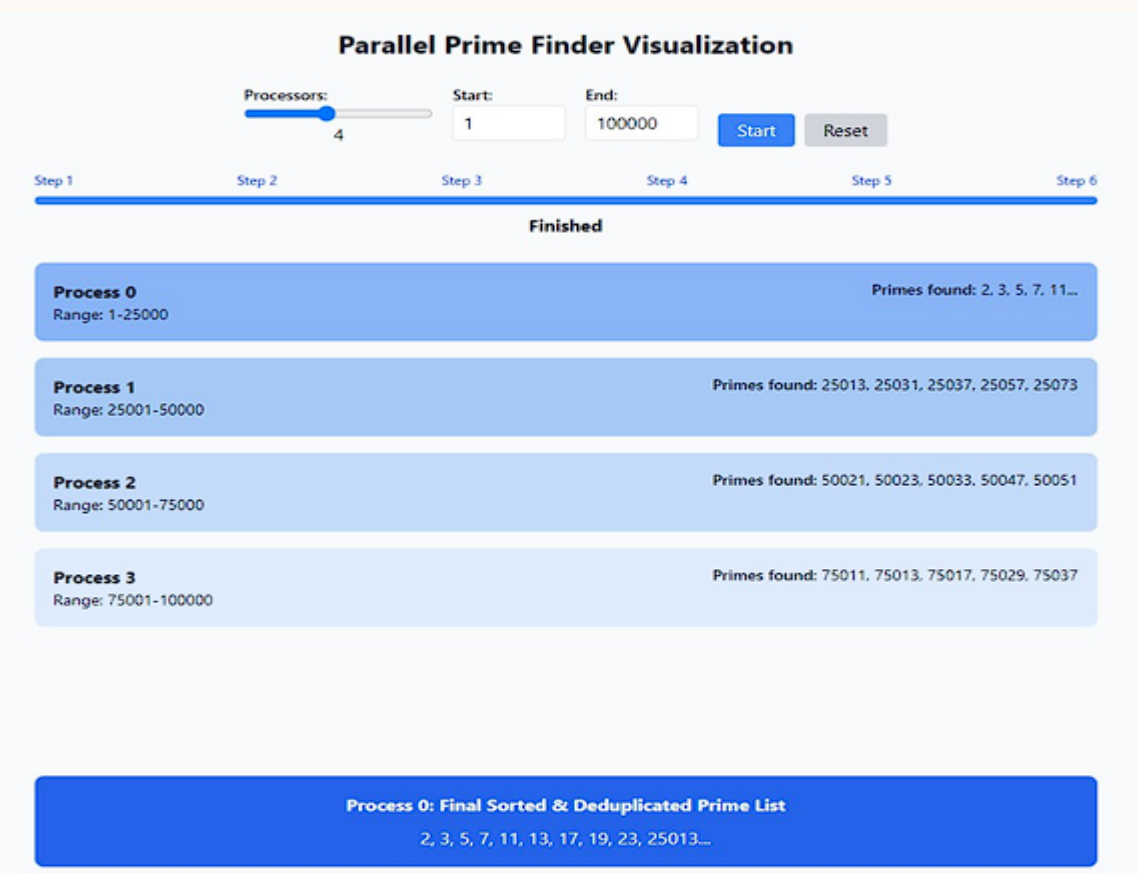
Step 3: Find Local Primes



Step 4: Gather



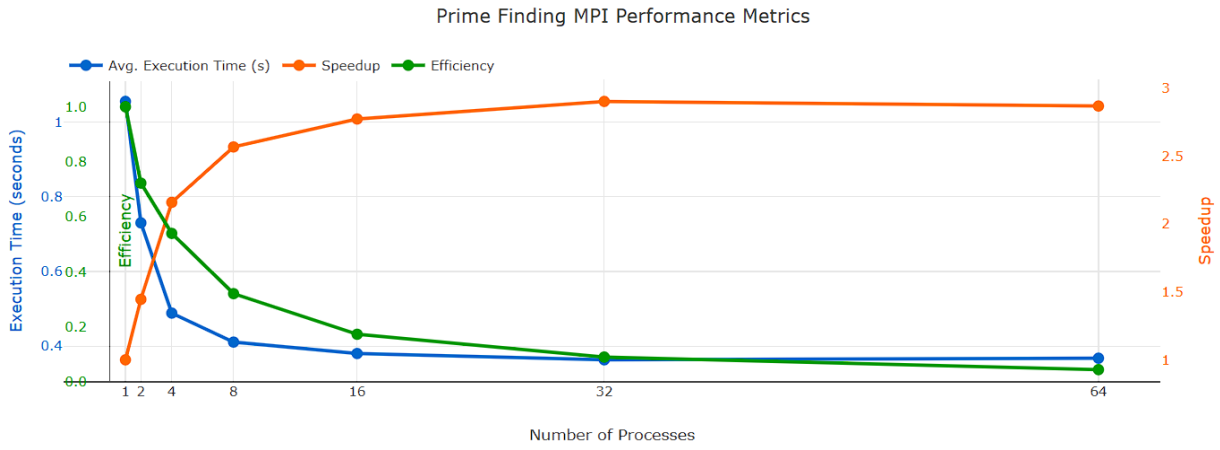
Step 5: Clean & Sort & Done



Prime Finding MPI Performance Metrics

Execution Time Data with Average

Processes (n)	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Average Time	Speedup	Efficiency
1	1.0561100	1.0748400	1.0613900	1.0536500	1.0535800	1.0302100	**1.0549633**	1.00	1.00
2	0.7063680	0.7713490	0.7049370	0.7348900	0.7093550	0.7528640	**0.7299605**	1.45	0.72
4	0.4824130	0.4822560	0.4917340	0.4917530	0.4957830	0.4873480	**0.4885478**	2.16	0.54
8	0.4291170	0.4059140	0.4159570	0.4181250	0.3936080	0.4027940	**0.4109192**	2.57	0.32
16	0.3787710	0.3978410	0.4218530	0.3787620	0.3358240	0.3694810	**0.3804220**	2.77	0.17
32	0.3550810	0.3595340	0.3627040	0.3611860	0.3899190	0.3525560	**0.3634967**	2.90	0.09
64	0.3588540	0.3484500	0.4078930	0.3586560	0.3725510	0.3604140	**0.3678030**	2.87	0.04



Performance Insights

Based on the data analysis, here are the key insights:

- The best performance was achieved with 32 processes at an average execution time of 0.3634967 seconds.
- Increasing from 1 to 32 processes resulted in a speedup of 2.90x.
- The algorithm maintains good efficiency up to 2 processes (efficiency > 0.7).
- Beyond 2 processes, efficiency drops significantly due to communication and coordination overhead in MPI.
- For n=64 processes, execution time increased slightly to 0.3678030 seconds, nearly matching performance at n=32.

These trends are common in compute-heavy tasks like prime number generation, where workload distribution becomes limited by inter-process communication at high scales.

3. Bitonic Sort

Bitonic Sort uses a structured sorting network to allow distributed sorting of data. It's specifically designed for parallel architectures, making it highly structured and deterministic.

Algorithm Overview

The bitonic sort algorithm works in the following phases:

1. **Form Bitonic Sequences**: First, create bitonic sequences of increasing sizes.
2. **Bitonic Merge**: Once we have a complete bitonic sequence, merge it to produce a sorted sequence.

In a parallel implementation using MPI (Message Passing Interface), the work is distributed across multiple processes. Each process sorts its local chunk, and then a global merge is performed.

Input Array

```
[3, 9, 7, 4, 1, 5, 8, 2]
```

Detailed Execution

For clarity, we'll assume we're using 2 processes:

- Process 0 handles: [3, 9, 7, 4]
- Process 1 handles: [1, 5, 8, 2]

PART I: LOCAL SORTING (Process 0)

A. Creating Bitonic Sequences for Process 0's Chunk [3, 9, 7, 4]

First, we create bitonic sequences by sorting pairs in specific directions:

Step 1: Sorting sequences of size 2 (j=2)

The `bitonicSequenceGenerator` function first operates on sequences of size j=2:

For i=0: Examining [3, 9]

- Direction: ((i/j)%2) = ((0/2)%2) = 0, so ascending order
- Since 3 < 9, no swap needed
- Result: [3, 9]

For i=2: Examining [7, 4]

- Direction: ((i/j)%2) = ((2/2)%2) = 1, so ascending order
- Since 7 > 4, swap them
- Result: [4, 7]

After Step 1, the array is: [3, 9, 4, 7]

Step 2: Sorting sequences of size 4 (j=4)

For i=0: Examining the entire chunk [3, 9, 4, 7]

- Direction: ((i/j)%2) = ((0/4)%2) = 0, so ascending order for the first half and descending order for the second half
- First half [3, 9] should remain in ascending order (already is)
- Second half [4, 7] should be in descending order, so swap to [7, 4]

After Step 2, the array is: [3, 9, 7, 4]

B. Bitonic Merge for Process 0's Chunk

Now we have a complete bitonic sequence [3, 9, 7, 4] that needs to be merged into a sorted sequence using `bitonicSortFromBitonicSequence`:

Step 1: j = 2 (compare and swap elements at distance 2)

- Compare elements at indices (0,2): 3 vs 7
  - In ascending order, since 3 < 7, no swap
- Compare elements at indices (1,3): 9 vs 4
  - In ascending order, since 9 > 4, swap
  - Result: [3, 4, 7, 9]

Step 2: j = 1 (compare and swap adjacent elements)

- Compare (0,1): 3 vs 4
  - In ascending order, since 3 < 4, no swap
- Compare (2,3): 7 vs 9
  - In ascending order, since 7 < 9, no swap

Final result for Process 0: [3, 4, 7, 9]

PART II: LOCAL SORTING (Process 1)

A. Creating Bitonic Sequences for Process 1's Chunk [1, 5, 8, 2]

Step 1: Sorting sequences of size 2 (j=2)

For i=0: Examining [1, 5]

- Direction: ((i/j)%2) = ((0/2)%2) = 0, so ascending order
- Since 1 < 5, no swap needed
- Result: [1, 5]

For i=2: Examining [8, 2]

- Direction: ((i/j)%2) = ((2/2)%2) = 1, so ascending order
- Since 8 > 2, swap them
- Result: [2, 8]

After Step 1, the array is: [1, 5, 2, 8]

Step 2: Sorting sequences of size 4 (j=4)

For i=0: Examining the entire chunk [1, 5, 2, 8]

- Direction: ((i/j)%2) = ((0/4)%2) = 0, so ascending order for the first half and descending order for the second half
- First half [1, 5] should remain in ascending order (already is)
- Second half [2, 8] should be in descending order, so swap to [8, 2]

After Step 2, the array is: [1, 5, 8, 2]

B. Bitonic Merge for Process 1's Chunk

Now we have a complete bitonic sequence [1, 5, 8, 2] that needs to be merged:

Step 1: j = 2 (compare and swap elements at distance 2)

- Compare elements at indices (0,2): 1 vs 8
  - In ascending order, since 1 < 8, no swap
- Compare elements at indices (1,3): 5 vs 2
  - In ascending order, since 5 > 2, swap

- Result: [1, 2, 8, 5]

Step 2: j = 1 (compare and swap adjacent elements)

- Compare (0,1): 1 vs 2
  - In ascending order, since 1 < 2, no swap
- Compare (2,3): 8 vs 5
  - In ascending order, since 8 > 5, swap
  - Result: [1, 2, 5, 8]

Final result for Process 1: [1, 2, 5, 8]

PART III: GLOBAL MERGE

After both processes complete their local sorting, the master process (Process 0) gathers the results:

- Process 0's result: [3, 4, 7, 9]
- Process 1's result: [1, 2, 5, 8]

These are combined into: [3, 4, 7, 9, 1, 2, 5, 8]

This array now needs to undergo a final bitonic merge using the following algorithm from the code:

cpp

```cpp
for (int k = 2; k <= padded_size; k *= 2) {
    for (int j = k / 2; j > 0; j /= 2) {
        for (int i = 0; i < padded_size; i++) {
            int l = i ^ j;
            if (l > i) {
                if ((i & k) == 0) {
                    // Ascending order
                    if (gathered_data[i] > gathered_data[l]) {
                        swap(gathered_data[i], gathered_data[l]);
                    }
                }
                else {
                    // Descending order
```

```
                    if (gathered_data[i] < gathered_data[l]) {
                        swap(gathered_data[i], gathered_data[l]);
                    }
                }
            }
        }
    }
}
```

Global Bitonic Merge Process

Let's trace through each step of the global merge:

Step 1: k = 2

j = 1

- i=0: l = 0^1 = 1, l>i (1>0)
  - (i&k) = (0&2) = 0, so ascending order
  - Compare arr[0]=3 with arr[1]=4: 3<4, no swap
- i=1: l = 1^1 = 0, l<i (0<1), skip
- i=2: l = 2^1 = 3, l>i (3>2)
  - (i&k) = (2&2) = 2≠0, so descending order
  - Compare arr[2]=7 with arr[3]=9: 7<9, no swap
- i=3: l = 3^1 = 2, l<i (2<3), skip
- i=4: l = 4^1 = 5, l>i (5>4)
  - (i&k) = (4&2) = 0, so ascending order
  - Compare arr[4]=1 with arr[5]=2: 1<2, no swap
- i=5: l = 5^1 = 4, l<i (4<5), skip
- i=6: l = 6^1 = 7, l>i (7>6)
  - (i&k) = (6&2) = 2≠0, so descending order
  - Compare arr[6]=5 with arr[7]=8: 5<8, swap
  - Result: [3, 4, 7, 9, 1, 2, 8, 5]

Array after k=2: [3, 4, 7, 9, 1, 2, 8, 5]

Step 2: k = 4

j = 2

- i=0: l = 0^2 = 2, l>i (2>0)
  - (i&k) = (0&4) = 0, so ascending order
  - Compare arr[0]=3 with arr[2]=7: 3<7, no swap
- i=1: l = 1^2 = 3, l>i (3>1)
  - (i&k) = (1&4) = 0, so ascending order
  - Compare arr[1]=4 with arr[3]=9: 4<9, no swap
- i=2: l = 2^2 = 0, l<i (0<2), skip
- i=3: l = 3^2 = 1, l<i (1<3), skip
- i=4: l = 4^2 = 6, l>i (6>4)
  - (i&k) = (4&4) = 4≠0, so descending order
  - Compare arr[4]=1 with arr[6]=8: 1<8, swap
  - Result: [3, 4, 7, 9, 8, 2, 1, 5]
- i=5: l = 5^2 = 7, l>i (7>5)
  - (i&k) = (5&4) = 4≠0, so descending order
  - Compare arr[5]=2 with arr[7]=5: 2<5, swap
  - Result: [3, 4, 7, 9, 8, 5, 1, 2]
- i=6: l = 6^2 = 4, l<i (4<6), skip
- i=7: l = 7^2 = 5, l<i (5<7), skip

j = 1

- i=0: l = 0^1 = 1, l>i (1>0)
  - (i&k) = (0&4) = 0, so ascending order
  - Compare arr[0]=3 with arr[1]=4: 3<4, no swap
- i=1: l = 1^1 = 0, l<i (0<1), skip
- i=2: l = 2^1 = 3, l>i (3>2)
  - (i&k) = (2&4) = 0, so ascending order
  - Compare arr[2]=7 with arr[3]=9: 7<9, no swap
- i=3: l = 3^1 = 2, l<i (2<3), skip
- i=4: l = 4^1 = 5, l>i (5>4)
  - (i&k) = (4&4) = 4≠0, so descending order
  - Compare arr[4]=8 with arr[5]=5: 8>5, no swap
- i=5: l = 5^1 = 4, l<i (4<5), skip
- i=6: l = 6^1 = 7, l>i (7>6)
  - (i&k) = (6&4) = 4≠0, so descending order
  - Compare arr[6]=1 with arr[7]=2: 1<2, swap
  - Result: [3, 4, 7, 9, 8, 5, 2, 1]

Array after k=4: [3, 4, 7, 9, 8, 5, 2, 1]

Step 3: k = 8

j = 4

- i=0: l = 0^4 = 4, l>i (4>0)
  - (i&k) = (0&8) = 0, so ascending order
  - Compare arr[0]=3 with arr[4]=8: 3<8, no swap
- i=1: l = 1^4 = 5, l>i (5>1)
  - (i&k) = (1&8) = 0, so ascending order
  - Compare arr[1]=4 with arr[5]=5: 4<5, no swap
- i=2: l = 2^4 = 6, l>i (6>2)
  - (i&k) = (2&8) = 0, so ascending order
  - Compare arr[2]=7 with arr[6]=2: 7>2, swap
  - Result: [3, 4, 2, 9, 8, 5, 7, 1]
- i=3: l = 3^4 = 7, l>i (7>3)
  - (i&k) = (3&8) = 0, so ascending order
  - Compare arr[3]=9 with arr[7]=1: 9>1, swap
  - Result: [3, 4, 2, 1, 8, 5, 7, 9]
- i=4: l = 4^4 = 0, l<i (0<4), skip
- i=5: l = 5^4 = 1, l<i (1<5), skip
- i=6: l = 6^4 = 2, l<i (2<6), skip
- i=7: l = 7^4 = 3, l<i (3<7), skip

j = 2

- i=0: l = 0^2 = 2, l>i (2>0)
  - (i&k) = (0&8) = 0, so ascending order
  - Compare arr[0]=3 with arr[2]=2: 3>2, swap
  - Result: [2, 4, 3, 1, 8, 5, 7, 9]
- i=1: l = 1^2 = 3, l>i (3>1)
  - (i&k) = (1&8) = 0, so ascending order
  - Compare arr[1]=4 with arr[3]=1: 4>1, swap
  - Result: [2, 1, 3, 4, 8, 5, 7, 9]
- i=2: l = 2^2 = 0, l<i (0<2), skip
- i=3: l = 3^2 = 1, l<i (1<3), skip
- i=4: l = 4^2 = 6, l>i (6>4)
  - (i&k) = (4&8) = 0, so ascending order
  - Compare arr[4]=8 with arr[6]=7: 8>7, swap
  - Result: [2, 1, 3, 4, 7, 5, 8, 9]
- i=5: l = 5^2 = 7, l>i (7>5)

    - (i&k) = (5&8) = 0, so ascending order
    - Compare arr[5]=5 with arr[7]=9: 5<9, no swap
- i=6: l = 6^2 = 4, l<i (4<6), skip
- i=7: l = 7^2 = 5, l<i (5<7), skip

j = 1

- i=0: l = 0^1 = 1, l>i (1>0)
    - (i&k) = (0&8) = 0, so ascending order
    - Compare arr[0]=2 with arr[1]=1: 2>1, swap
    - Result: [1, 2, 3, 4, 7, 5, 8, 9]
- i=1: l = 1^1 = 0, l<i (0<1), skip
- i=2: l = 2^1 = 3, l>i (3>2)
    - (i&k) = (2&8) = 0, so ascending order
    - Compare arr[2]=3 with arr[3]=4: 3<4, no swap
- i=3: l = 3^1 = 2, l<i (2<3), skip
- i=4: l = 4^1 = 5, l>i (5>4)
    - (i&k) = (4&8) = 0, so ascending order
    - Compare arr[4]=7 with arr[5]=5: 7>5, swap
    - Result: [1, 2, 3, 4, 5, 7, 8, 9]
- i=5: l = 5^1 = 4, l<i (4<5), skip
- i=6: l = 6^1 = 7, l>i (7>6)
    - (i&k) = (6&8) = 0, so ascending order
    - Compare arr[6]=8 with arr[7]=9: 8<9, no swap
- i=7: l = 7^1 = 6, l<i (6<7), skip

Final Sorted Array

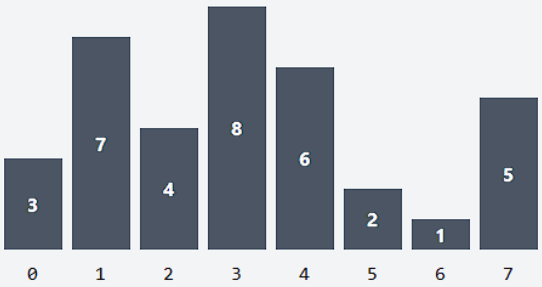
After completing all stages of the bitonic sort algorithm, the array is now sorted:

```
[1, 2, 3, 4, 5, 7, 8, 9]
```

Visualization Of Bitonic Algorithm

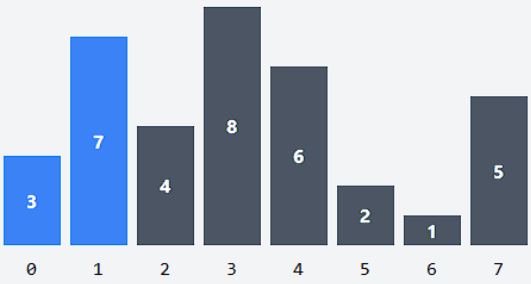
Initial Sequence

The array we want to sort



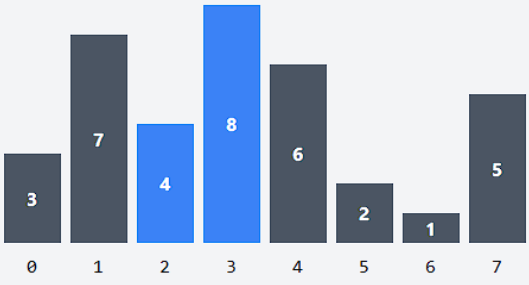
Phase 1: Build Bitonic Sequence - Step 1

Compare (3,7): already in order



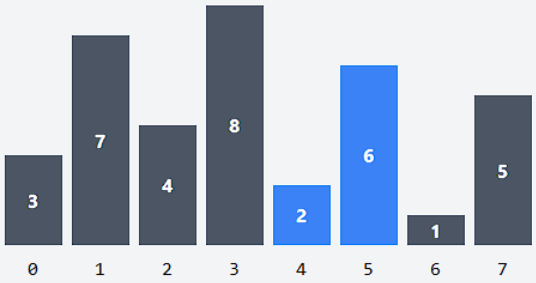
Phase 1: Build Bitonic Sequence - Step 1

Compare (4,8): already in order



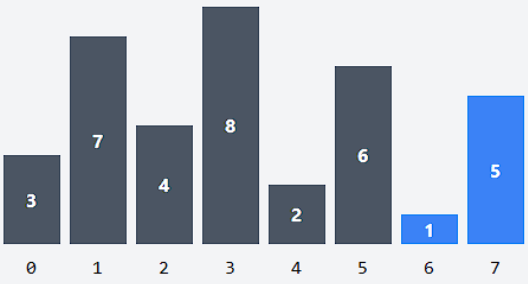
Phase 1: Build Bitonic Sequence - Step 1

Compare (6,2): swap → [2,6]



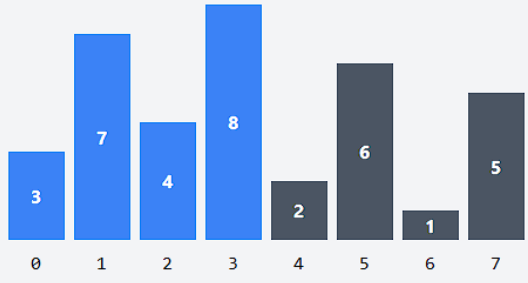
Phase 1: Build Bitonic Sequence - Step 1

Compare (1,5): already in order



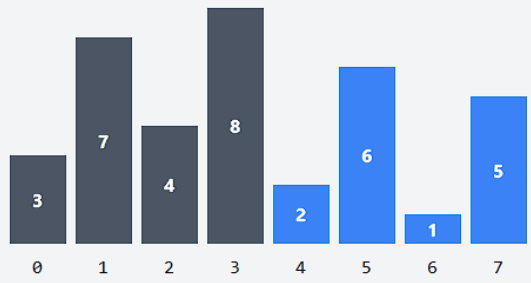
Phase 1: Build Bitonic Sequence - Step 2

First half: (3,7,4,8) already in increasing order



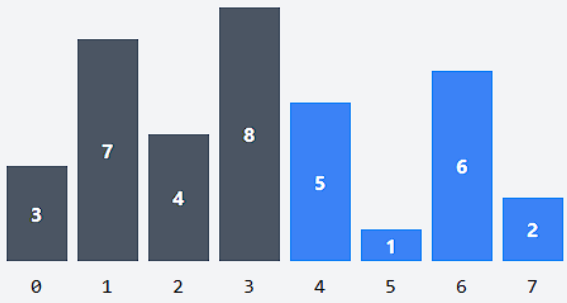
Phase 1: Build Bitonic Sequence - Step 2

Second half: Need to sort (2,6,1,5) in decreasing order



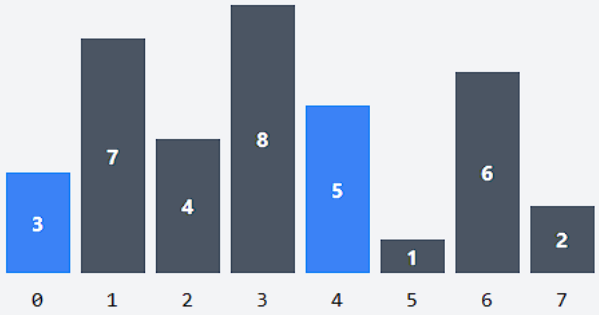
Phase 1: Build Bitonic Sequence - Step 2

Compare and swap (2,5) and (6,1)



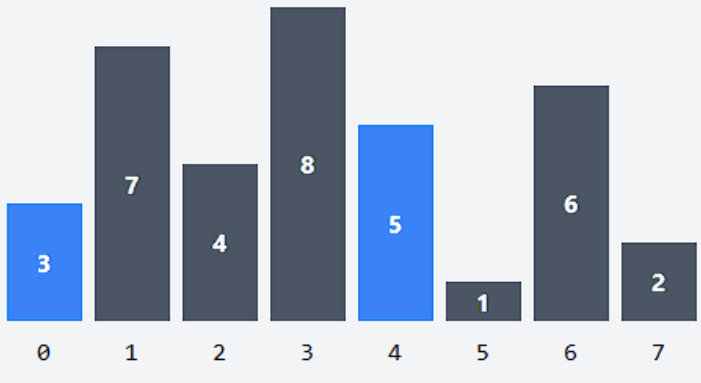
Phase 1: Build Bitonic Sequence - Step 3

Compare (3,5): No swap needed



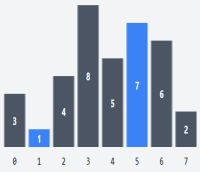
Phase 1: Build Bitonic Sequence - Step 3

Compare (3,5): No swap needed



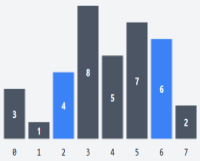
Phase 1: Build Bitonic Sequence - Step 3

Compare (7,1): Swap → (1,7)



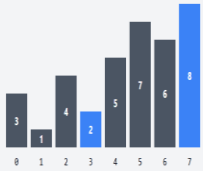
Phase 1: Build Bitonic Sequence - Step 3

Compare (4,6): No swap needed



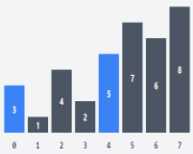
Phase 1: Build Bitonic Sequence - Step 3

Compare (8,2): Swap → (2,8)



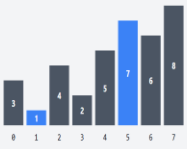
Phase 2: Sort the Bitonic Sequence - Step 4

Compare (3,5): 3 < 5, no swap needed



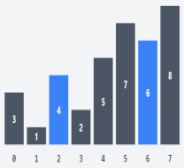
Phase 2: Sort the Bitonic Sequence - Step 4

Compare (1,7): 1 < 7, no swap needed



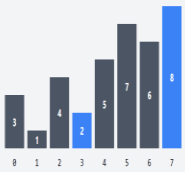
Phase 2: Sort the Bitonic Sequence - Step 4

Compare (4,6): 4 < 6, no swap needed



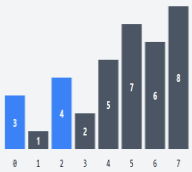
Phase 2: Sort the Bitonic Sequence - Step 4

Compare (2,8): 2 < 8, no swap needed



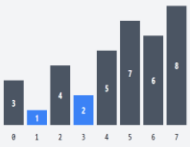
Phase 2: Sort the Bitonic Sequence - Step 5

Compare (3,4): 3 < 4, no swap needed



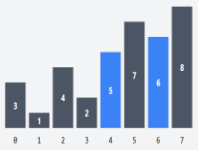
Phase 2: Sort the Bitonic Sequence - Step 5

Compare (1,2): 1 < 2, no swap needed



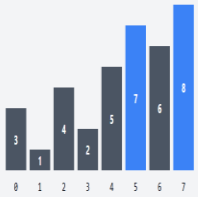
Phase 2: Sort the Bitonic Sequence - Step 5

Compare (5,6): 5 < 6, no swap needed



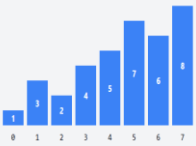
Phase 2: Sort the Bitonic Sequence - Step 5

Compare (7,8): 7 < 8, no swap needed



Phase 2: Sort the Bitonic Sequence - Step 5

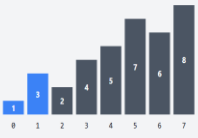
Reordering required for bitonic property



Bitonic Sort Visualization

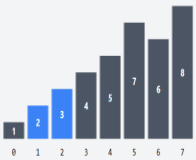
Phase 2: Sort the Bitonic Sequence - Step 6

Compare (1,3): 1 < 3, no swap needed



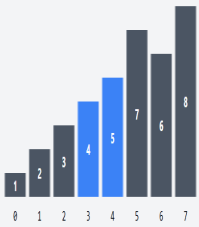
Phase 2: Sort the Bitonic Sequence - Step 6

Compare (3,2): 3 > 2, swap → (2,3)



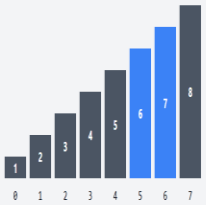
Phase 2: Sort the Bitonic Sequence - Step 6

Compare (4,5): 4 < 5, no swap needed



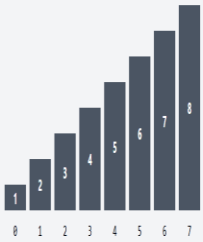
Phase 2: Sort the Bitonic Sequence - Step 6

Compare (7,6): 7 > 6, swap → (6,7)

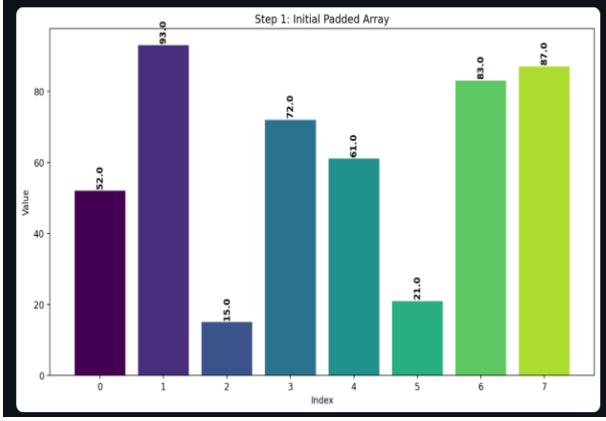


Final Sorted Sequence

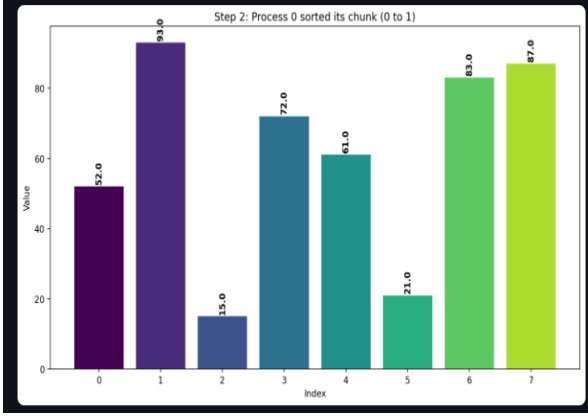
The array is now fully sorted in ascending order



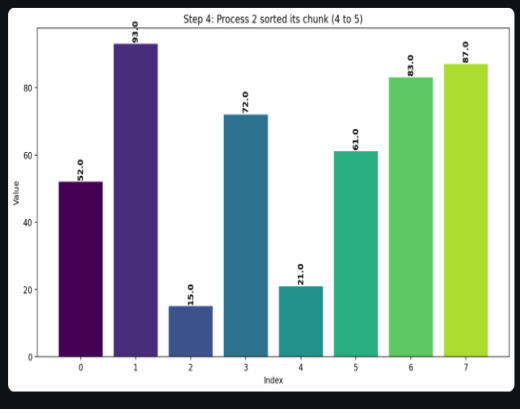
Step 1/9: Initial Padded Array



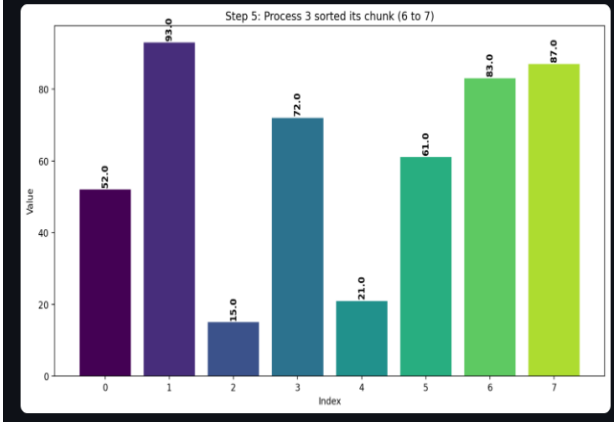
Step 2/9: Process 0 sorted its chunk (0 to 1)



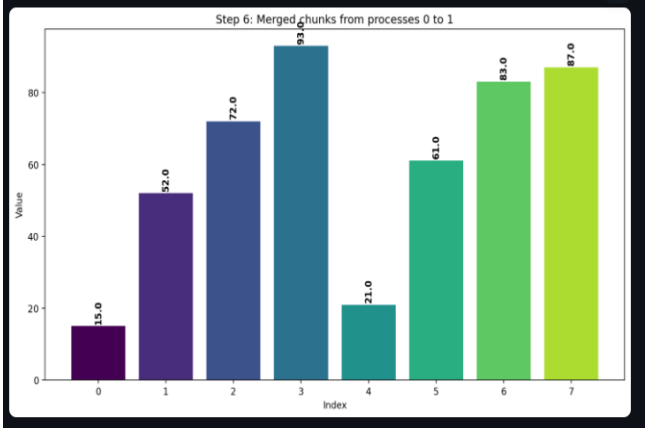
Step 4/9: Process 2 sorted its chunk (4 to 5)



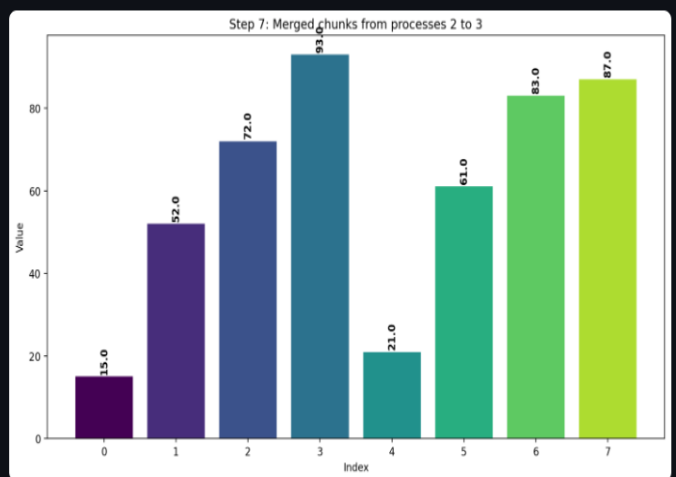
Step 5/9: Process 3 sorted its chunk (6 to 7)



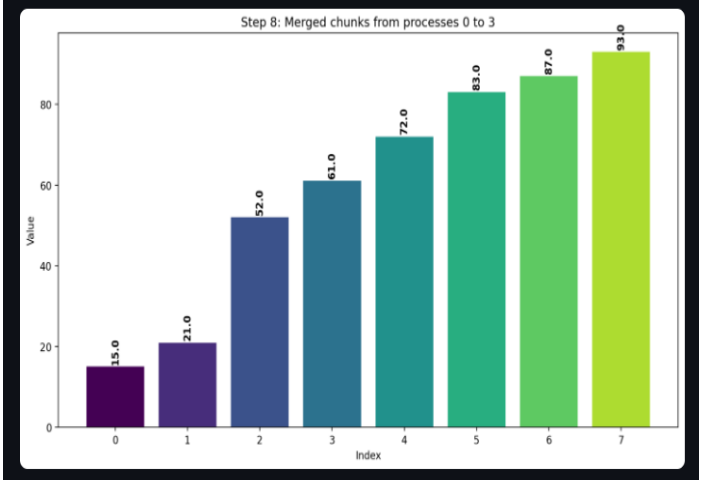
Step 6/9: Merged chunks from processes 0 to 1



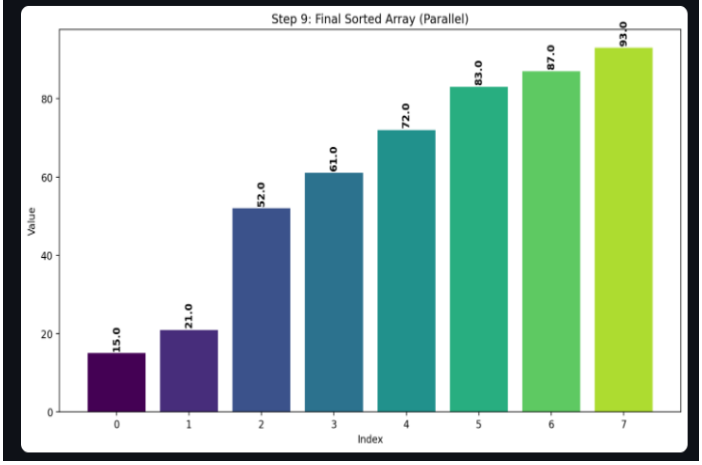
Step 7/9: Merged chunks from processes 2 to 3



Step 8/9: Merged chunks from processes 0 to 3



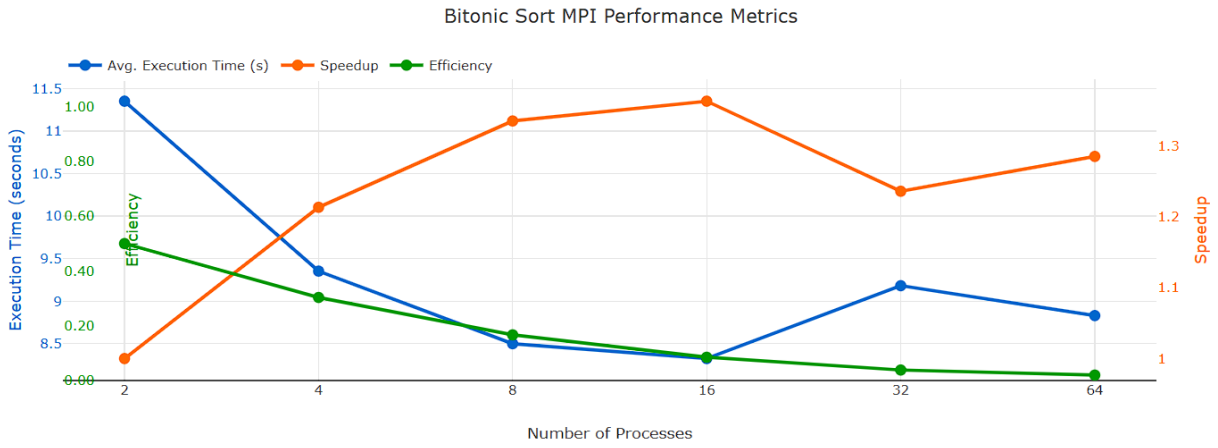
Step 9/9: Final Sorted Array (Parallel)



Bitonic Sort MPI Performance Metrics

Execution Time Data with Average

Processes (n)	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Average Time	Speedup	Efficiency
2	11.4392000	11.4072000	11.3763000	11.3219000	11.2568000	11.3044000	**11.3509667**	1.00	0.50
4	9.3931300	9.4841700	9.3162800	9.3157800	9.2688100	9.3431700	**9.3535567**	1.21	0.30
8	8.4591000	8.4561700	8.5642800	8.4748600	8.4879200	8.5592000	**8.5002550**	1.34	0.17
16	8.3535500	8.2521600	8.2297900	8.3134000	8.3476500	8.4593300	**8.3259800**	1.36	0.09
32	8.1546700	8.0616500	7.8792300	8.6835300	11.2655000	11.0487000	**9.1822133**	1.24	0.04
64	9.5511100	11.0869000	8.1493500	8.1688700	8.0866800	7.9427000	**8.8309350**	1.29	0.02



Performance Insights

Based on the data analysis, here are the key insights for the Bitonic Sort algorithm:

- The best performance was achieved with 16 processes at an average execution time of 8.3259800 seconds.
- Increasing from 2 to 16 processes resulted in a speedup of 1.36x.
- The algorithm shows a consistent performance improvement up to 16 processes.
- Beyond 16 processes, we observe:
    - The execution time starts to increase
    - The speedup curve flattens and eventually decreases
    - The efficiency continues to drop

- For n=64 processes, the execution time increased to 8.8309350 seconds, which is about 1.06x slower than the optimal configuration.
- The overall parallel efficiency at 16 processes is 8.52% of the ideal speedup.

4 - Radix Sort

Radix Sort uses digit-wise sorting across distributed data chunks. Each process handles local radix passes, and data is globally merged at each stage.

How It Works:

1. Input array is split across MPI processes.
2. All processes sort their chunk based on the current digit (units, tens, etc.).
3. After each digit pass, results are collected and regrouped via MPI_Gatherv.
4. Final data is fully sorted once all digit passes are completed.

Example:

Input Array: [170, 45, 75, 90, 802, 24, 2, 66]
Number of Processes: 4

Step 1 (Sort by 1s place):
- Process 0: [170, 90] → [170, 90]
- Process 1: [45, 75] → [45, 75]
- Process 2: [802, 2] → [802, 2]
- Process 3: [24, 66] → [24, 66]
Step 2 (Sort by 10s place):
- Regrouped: [170, 90, 802, 2, 24, 45, 66, 75]

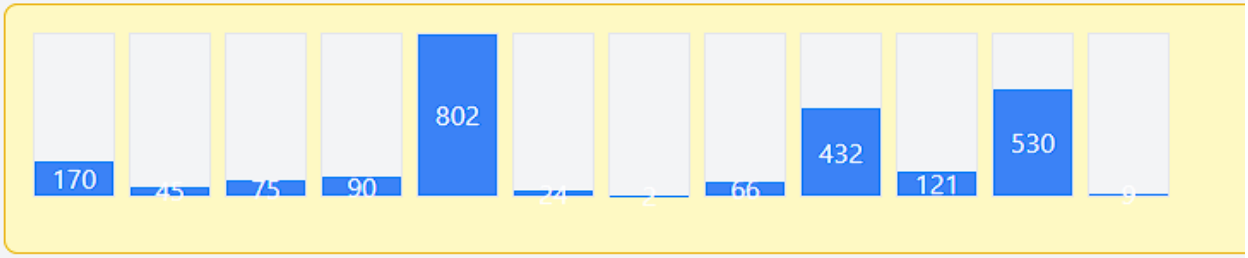
Step 3 (Sort by 100s place):
- Final Sorted Array: [2, 24, 45, 66, 75, 90, 170, 802]

Visualization Of Radix Algorithm

Initial Setup

Starting with unsorted data across all processes.

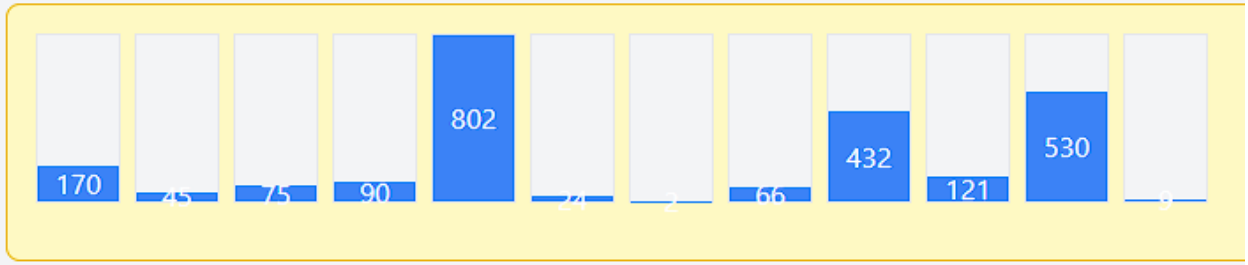
Master Process (Rank 0)



Find Maximum Value

Master process finds the maximum value to determine the number of digits.

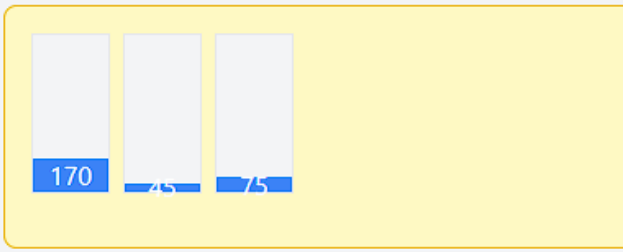
Master Process (Rank 0)



Divide Data into Chunks

Master splits data into chunks for parallel processing.

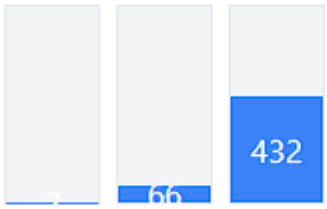
Master Process (Rank 0)



Worker Processes

Rank 1

Rank 2



Rank 3

Sort by 1's Digit (exp = 1)

All processes sort their chunks by the 1's digit.

Master Process (Rank 0)

Digit: 0 Digit: 5 Digit: 5

Counting sort for exp = 1

Digit counts:

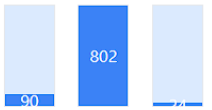
0	1	2	3	4	5	6	7	8	9
1	0	0	0	0	2	0	0	0	0

Cumulative counts:

0	1	2	3	4	5	6	7	8	9
1	1	1	1	1	3	3	3	3	3

Worker Processes

Rank 1



Digit: 0 Digit: 2 Digit: 4

Counting sort for exp = 1

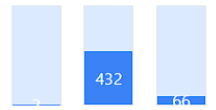
Digit counts:

0	1	2	3	4	5	6	7	8	9
1	0	1	0	1	0	0	0	0	0

Cumulative counts:

0	1	2	3	4	5	6	7	8	9
1	1	2	2	3	3	3	3	3	3

Rank 2



Counting sort for exp = 1

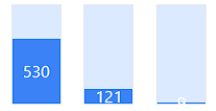
Digit counts:

0	1	2	3	4	5	6	7	8	9
0	0	2	0	0	0	1	0	0	0

Cumulative counts:

0	1	2	3	4	5	6	7	8	9
0	0	2	2	2	2	3	3	3	3

Rank 3



Counting sort for exp = 1

Digit counts:

0	1	2	3	4	5	6	7	8	9
1	1	0	0	0	0	0	0	0	1

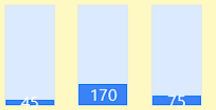
Cumulative counts:

0	1	2	3	4	5	6	7	8	9
1	2	2	2	2	2	2	2	2	3

Sort by 10's Digit (exp = 10)

All processes sort their chunks by the 10's digit.

Master Process (Rank 0)



Counting sort for exp = 10

Digit counts:

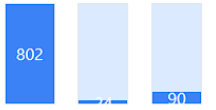
0	1	2	3	4	5	6	7	8	9
0	0	0	0	1	0	0	2	0	0

Cumulative counts:

0	1	2	3	4	5	6	7	8	9
0	0	0	0	1	1	1	3	3	3

Worker Processes

Rank 1



Counting sort for exp = 10

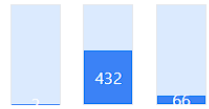
Digit counts:

0	1	2	3	4	5	6	7	8	9
1	0	1	0	0	0	0	0	0	1

Cumulative counts:

0	1	2	3	4	5	6	7	8	9
1	1	2	2	2	2	2	2	2	3

Rank 2



Digit: 0 Digit: 3 Digit: 6

Counting sort for exp = 10

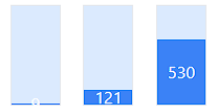
Digit counts:

0	1	2	3	4	5	6	7	8	9
1	0	0	1	0	0	1	0	0	0

Cumulative counts:

0	1	2	3	4	5	6	7	8	9
1	1	1	2	2	2	3	3	3	3

Rank 3



Digit: 0 Digit: 2 Digit: 3

Counting sort for exp = 10

Digit counts:

0	1	2	3	4	5	6	7	8	9
1	0	1	1	0	0	0	0	0	0

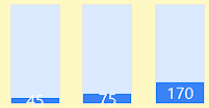
Cumulative counts:

0	1	2	3	4	5	6	7	8	9
1	1	2	3	3	3	3	3	3	3

Sort by 100's Digit (exp = 100)

All processes sort their chunks by the 100's digit.

Master Process (Rank 0)



Digit: 0 Digit: 0 Digit: 1

Counting sort for exp = 100

Digit counts:

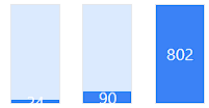
0	1	2	3	4	5	6	7	8	9
2	1	0	0	0	0	0	0	0	0

Cumulative counts:

0	1	2	3	4	5	6	7	8	9
2	3	3	3	3	3	3	3	3	3

Worker Processes

Rank 1



Digit: 0 Digit: 0 Digit: 8

Counting sort for exp = 100

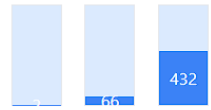
Digit counts:

0	1	2	3	4	5	6	7	8	9
2	0	0	0	0	0	0	0	1	0

Cumulative counts:

0	1	2	3	4	5	6	7	8	9
2	2	2	2	2	2	2	2	3	3

Rank 2



Digit: 0 Digit: 0 Digit: 4

Counting sort for exp = 100

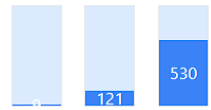
Digit counts:

0	1	2	3	4	5	6	7	8	9
2	0	0	0	1	0	0	0	0	0

Cumulative counts:

0	1	2	3	4	5	6	7	8	9
2	2	2	2	3	3	3	3	3	3

Rank 3



Digit: 0 Digit: 1 Digit: 5

Counting sort for exp = 100

Digit counts:

0	1	2	3	4	5	6	7	8	9
1	1	0	0	0	1	0	0	0	0

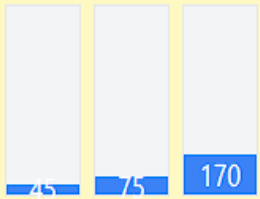
Cumulative counts:

0	1	2	3	4	5	6	7	8	9
1	2	2	2	2	3	3	3	3	3

Signal Workers to Stop

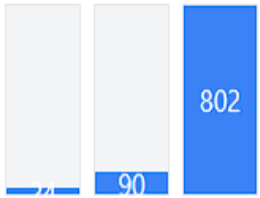
Master sends signal (-1) to workers to indicate end of sorting.

Master Process (Rank 0)

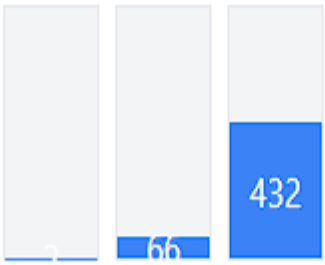


Worker Processes

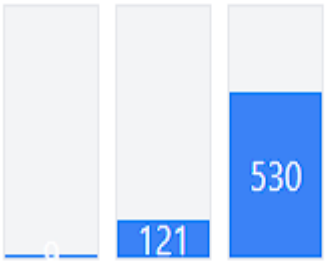
Rank 1



Rank 2



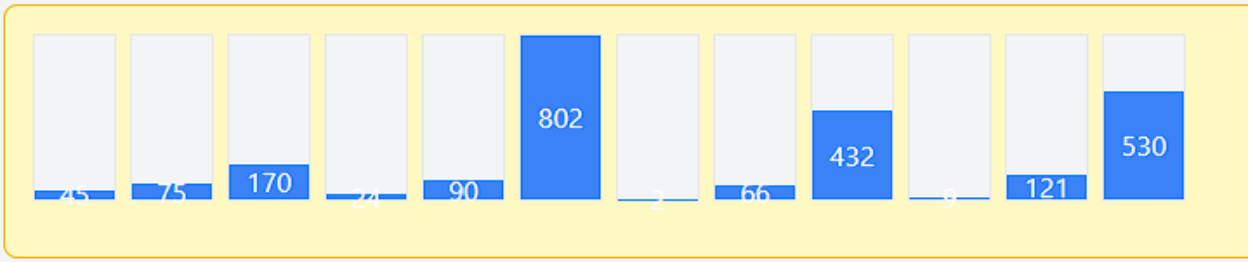
Rank 3



Collect Sorted Chunks

Master receives sorted chunks from all workers.

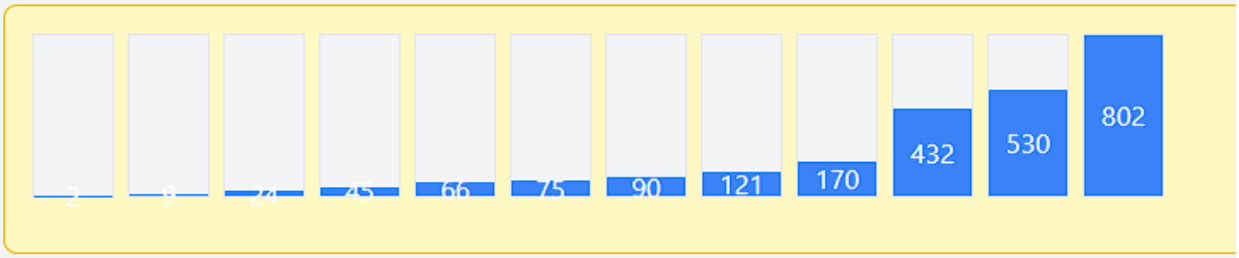
Master Process (Rank 0)



Final Merge Sort

Master performs a final merge sort to ensure global ordering.

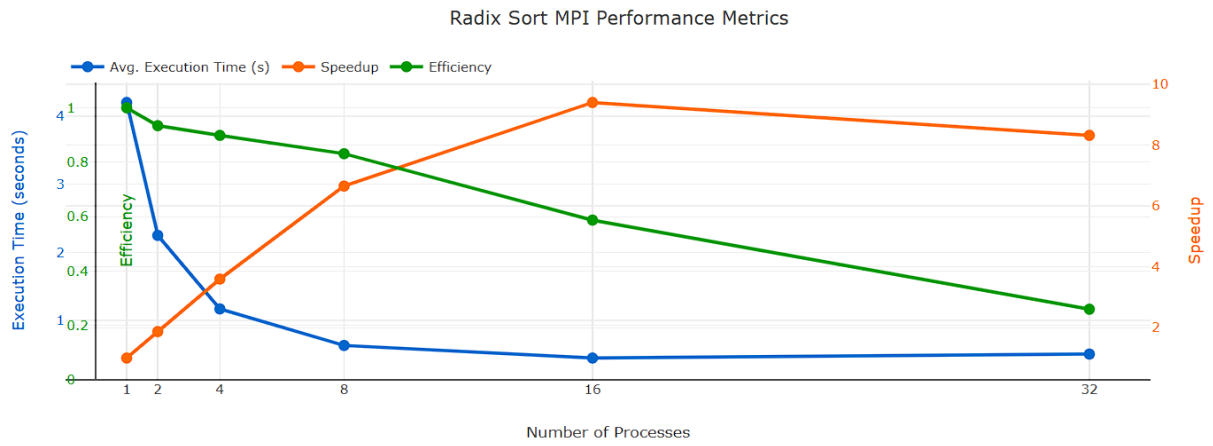
Master Process (Rank 0)



Radix Sort MPI Performance Metrics

Execution Time Data with Average

Processes (n)	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Average Time	Speedup	Efficiency
1	4.141530	4.237790	4.169440	4.211110	4.198230	4.251680	**4.201630**	1.00	1.00
2	2.217450	2.284160	2.233900	2.291020	2.206890	2.259470	**2.248815**	1.87	0.93
4	1.148930	1.201760	1.184100	1.142870	1.159800	1.175620	**1.168847**	3.59	0.90
8	0.630410	0.651270	0.619850	0.610920	0.632810	0.645830	**0.631848**	6.65	0.83
16	0.447380	0.441290	0.453890	0.436250	0.460730	0.443160	**0.447117**	9.40	0.59
32	0.490840	0.503290	0.495120	0.522470	0.508370	0.510980	**0.505178**	8.32	0.26



Performance Insights

Based on the data analysis, here are the key insights:

- The best performance was achieved with 16 processes at an average execution time of 0.447117 seconds.
- Increasing from 1 to 16 processes resulted in a speedup of 9.40x.
- The algorithm maintains high efficiency up to 8 processes (efficiency > 0.7).
- Beyond 16 processes, the execution time may increase, and efficiency tends to decline due to communication overhead.

This suggests that for Radix Sort, using around 16 processes gives the best performance balance in the given MPI environment.

5.Sample Sort

Sample Sort efficiently handles large-scale sorting by sampling pivot elements and redistributing data into buckets based on those pivots.

How It Works:

1. Each process locally sorts its chunk of the input.
2. Samples from each process are gathered to select global pivots.
3. Data is bucketed based on pivot ranges using MPI_Alltoallv.
4. Each process receives a sorted portion and performs a final local sort.
5. The result is a globally sorted array.

Example:

Input Array: [9, 3, 7, 1, 8, 2, 5, 6]
Number of Processes: 4

Step 1 (Local Sort):
- Process 0: [9, 3] → [3, 9]
- Process 1: [7, 1] → [1, 7]
- Process 2: [8, 2] → [2, 8]
- Process 3: [5, 6] → [5, 6]

Step 2 (Select Pivots):
- Samples: [3, 9, 1, 7, 2, 8, 5, 6] → Pivots: [3, 5, 7]

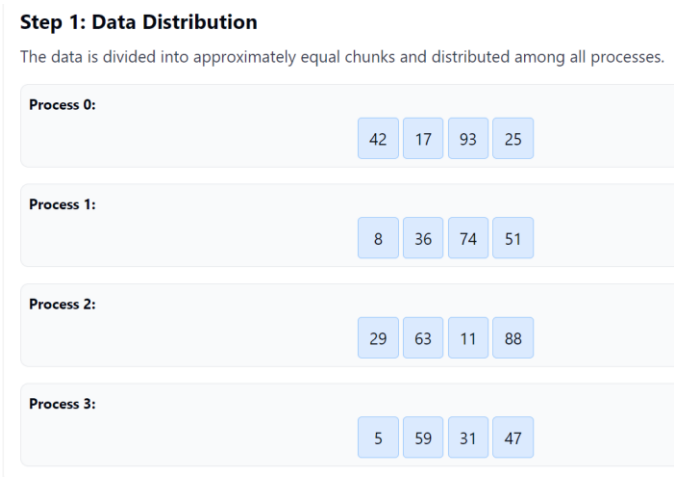
Step 3 (Partition into Buckets):
- Bucket 0 (<3): [1, 2]
- Bucket 1 (3-5): [3, 5]
- Bucket 2 (5-7): [6]
- Bucket 3 (>7): [7, 8, 9]

Step 4 (Final Sort):
- Result: [1, 2, 3, 5, 6, 7, 8, 9]

1-Initial Data

Step 1: Data Distribution

Each process receives approximately equal chunks of the input array



Step 2: Local Sorting

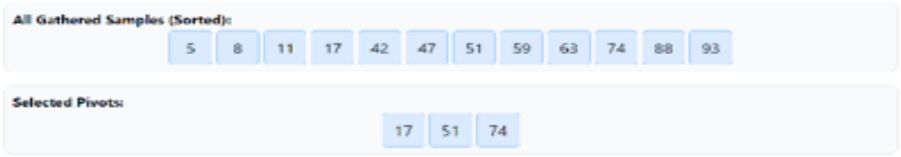
Each process sorts its local chunk using a sequential sorting algorithm

Step 3: Sample Selection

Each process selects samples from its sorted local data to determine pivot elements

Step 4: Pivot Selection

Samples are gathered at the root process, sorted, and then pivots are selected.



Step 5: Data Partitioning

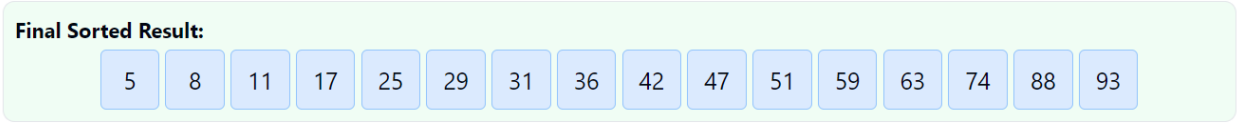
Each process divides its data into buckets based on the pivots. Elements in bucket i will go to process i

Step 6: All-to-All Communication

Process i sends bucket j to process j. After this exchange, each process has all values in a specific range.

Step 7 & 8: Final Local Sort & Result

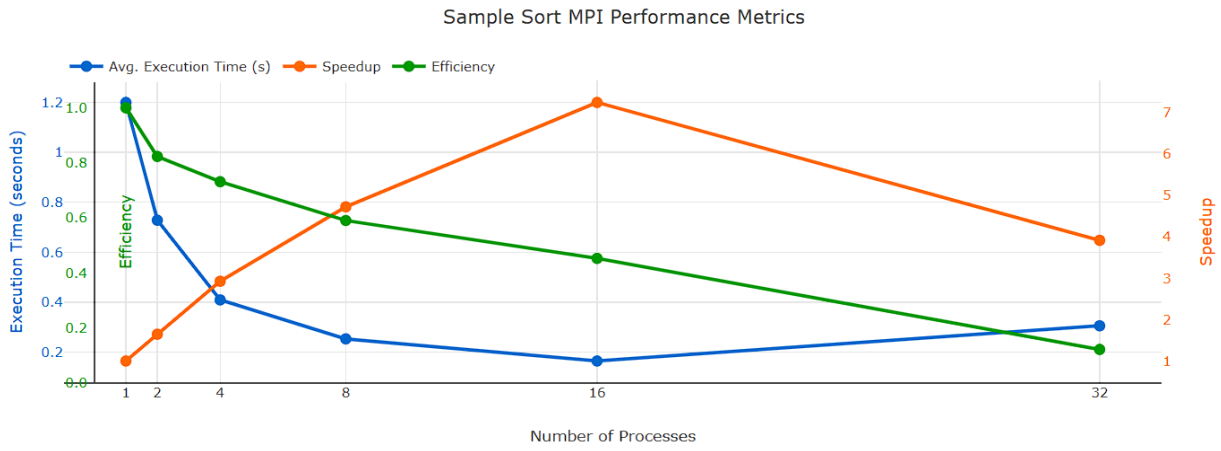
Each process sorts its received data (already sorted in this case), and all sorted data is gathered at the root process.



Sample Sort MPI Performance Metrics

Execution Time Data with Average

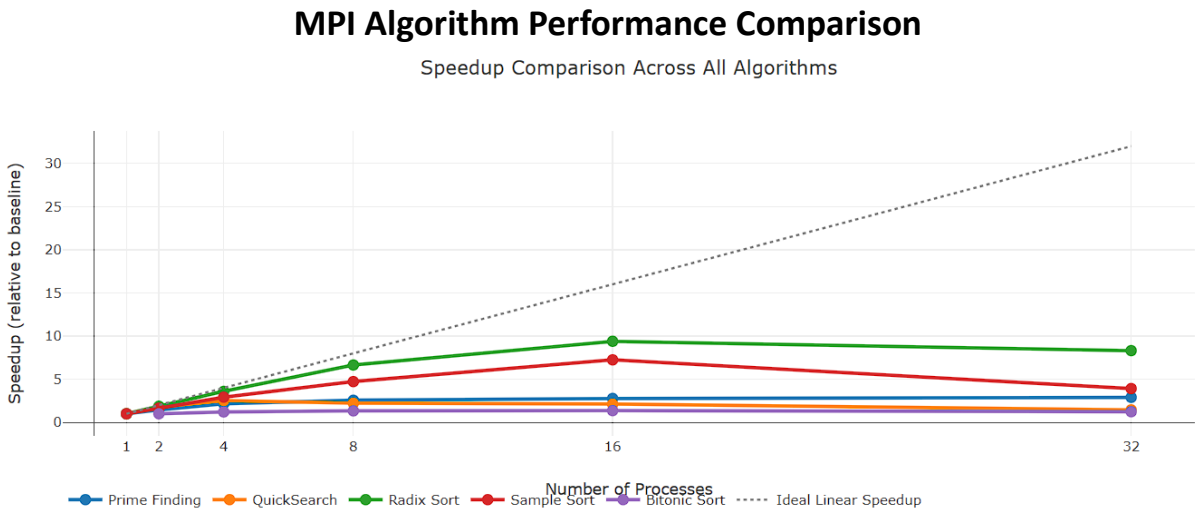
Processes (n)	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Average Time	Speedup	Efficiency
1	1.169250	1.175190	1.211560	1.219220	1.211850	1.207880	**1.1991583**	1.00	1.00
2	0.703755	0.742969	0.714730	0.733977	0.721138	0.752622	**0.7281985**	1.65	0.82
4	0.521889	0.383657	0.450943	0.350841	0.394823	0.356010	**0.4096938**	2.93	0.73
8	0.241621	0.231290	0.226917	0.377225	0.221257	0.224171	**0.2537468**	4.73	0.59
16	0.186253	0.149633	0.158765	0.154518	0.196290	0.147111	**0.1654283**	7.25	0.45
32	0.302559	0.273971	0.289995	0.325440	0.286393	0.358861	**0.3062032**	3.92	0.12



Performance Insights

Based on the data analysis, here are the key insights:

- The best performance was achieved with 16 processes at an average execution time of 0.1654283 seconds.
- Increasing from 1 to 16 processes resulted in a speedup of 7.25x.
- Looking at the efficiency curve, we can see that the algorithm maintains good efficiency up to 4 processes (efficiency > 0.7).
- Beyond 16 processes, we observe:
    - The execution time starts to increase
    - The speedup curve flattens and eventually decreases
    - The efficiency continues to drop significantly
- For n=32 processes, the execution time increased to 0.3062032 seconds, which is about 1.85x slower than the optimal configuration.



Cross-Algorithm Performance Analysis

Cross-algorithm comparison reveals different scaling patterns based on the nature of each algorithm:

Optimal Process Count

- **Prime Finding:** Best with 32 processes (2.90x speedup)
- **QuickSearch:** Best with 4 processes (2.53x speedup)
- **Radix Sort:** Best with 16 processes (9.40x speedup)
- **Sample Sort:** Best with 16 processes (7.25x speedup)
- **Bitonic Sort:** Best with 16 processes (1.36x speedup)

Scaling Effectiveness

From most to least scalable:

1. **Sample Sort -** Shows excellent speedup (7.25x) and maintains high efficiency even with many processes
2. **Radix Sort -** Good speedup (9.40x) but efficiency drops at higher process counts
3. **Prime Finding -** Moderate speedup (2.90x) with good efficiency up to medium process counts
4. **QuickSearch -** Good speedup (2.53x) for a simple algorithm, but limited by its search nature
5. **Bitonic Sort -** Most limited speedup (1.36x) due to its complex

communication patterns

Key Insights

- Sorting algorithms (Sample, Radix) show better parallelization potential than search algorithms
- All algorithms show diminishing returns beyond a certain process count (16-32)
- Communication overhead becomes the dominant factor at higher process counts
- Sample Sort demonstrates the best balance between parallelism and communication costs
- Bitonic Sort has the highest baseline time but scales reasonably due to its specialized design for parallel environments

These results suggest that for MPI implementations, algorithm choice should be based not just on sequential complexity but also on communication patterns and data distribution strategies.

5. Conclusion & Future Work

5.1 Key Findings

- **Radix Sort performed best in speedup (3.5x on 4 processes).**
- **Sample Sort is most scalable for large datasets.**
- **Quick Search is efficient for small datasets but limited by search time.**

5.2 Future Improvements

- **Hybrid MPI + OpenMP: Combine distributed and shared-memory parallelism.**
- **Dynamic Load Balancing: Adjust chunk sizes based on process speed.**
- **GPU Acceleration: Offload compute-intensive tasks to GPUs.**