A Highly Parallel Implementation of Merge Sort Measures on Multicore Systems

Team Four (4)

Abstract

Sorting large datasets is a fundamental problem in computer science and requires significant computational resources, especially for larger input sizes. Parallel processing offers a natural solution to reduce execution times by distributing the workload across multiple processing units. In this paper, we focus on parallelizing the Merge Sort algorithm using two popular parallel programming models: Pthread and MPI. Pthread is used for shared-memory architectures, while MPI is employed for distributed-memory systems. We present parallel implementations of Merge Sort for both architectures and evaluate their performance using various dataset sizes. Our experimental results demonstrate the efficiency of these parallel solutions, achieving significant speedups and scalability. This work provides insights into the comparative performance and applicability of Pthread and MPI for parallel sorting tasks on multicore systems

Keywords:

Merge Sort, parallel sorting, parallelization, Pthread, Message Passing Interface (MPI), speedup, multicore architectures.

1. Introduction

Sorting algorithms are fundamental in computer science, underpinning numerous applications in data processing, scientific computing, and large-scale information retrieval systems. As data sizes continue to grow exponentially, the computational cost of sorting becomes a significant challenge, particularly when processing large datasets in real-time applications. Parallel computing has emerged as a practical solution to address these challenges by exploiting modern multicore and distributed-memory architectures to reduce execution time.

Among sorting algorithms, merge sort is particularly well-suited for parallelization due to its divide-and-conquer structure, which allows the sorting of subarrays independently. By leveraging this property, significant performance improvements can be achieved using parallel computing techniques. In this work, we investigate parallel implementations of merge sort using two prominent approaches: Pthreads, which operates on shared-memory systems, and MPI (Message Passing Interface), designed for distributed-memory systems. These two paradigms represent the core models of parallel computation and provide an opportunity to evaluate their relative efficiency, scalability, and suitability for sorting large datasets.

Previous research has extensively explored the parallelization of sorting algorithms., achieving notable performance gains. However, merge sort continues to be a competitive choice due to its stability and predictable time complexity of O(n log n). Parallel merge sort implementations have demonstrated significant speedups over their sequential counterparts, particularly for high-volume datasets, but the choice of parallelization framework plays a critical role in determining overall performance.

In this paper, we focus on evaluating and comparing the performance of Pthreads and MPI implementations of parallel merge sort. While Pthreads excels in scenarios where threads share a common memory space, MPI is designed to handle distributed datasets across multiple nodes. These

differing architectures provide an opportunity to study how system design influences performance and scalability in parallel sorting.

Our contributions include the following:

1. A detailed implementation of parallel merge sort using Pthreads for shared-memory architectures.
2. A corresponding implementation of parallel merge sort using MPI for distributed-memory architectures.
3. A comparative analysis of the two approaches based on experimental results, highlighting the trade-offs in execution time, speedup, and scalability for varying input sizes.

The remainder of this paper is organized as follows. Section 2 provides an overview of the merge sort algorithm and its suitability for parallelization. Section 3 outlines the design and implementation details for Pthreads and MPI. Section 4 presents a performance analysis of the proposed implementations, and Section 5 discusses the experimental results. Finally, Section 6 concludes the paper and offers insights for future work.

2. provides an overview of the merge sort algorithm and its suitability for parallelization

Merge sort is a comparison-based sorting algorithm that follows the divide-and-conquer paradigm. It recursively divides an input array into smaller subarrays, sorts these subarrays, and then merges them back together to produce a fully sorted array. This process ensures a stable sorting mechanism with a predictable time complexity of O(nlogn).

2.1 Merge Sort Algorithm

The merge sort algorithm operates in three main steps:

1. **Divide:** The input array is divided into two equal (or nearly equal) halves. This process is repeated recursively until each subarray contains only one element.
2. **Conquer:** Each of the subarrays is sorted independently.
3. **Combine:** The sorted subarrays are merged back together to form a single sorted array.

The merge step is the core of the algorithm, where two sorted arrays are combined into one. This is done by comparing elements from each subarray and placing the smaller element into a temporary array, ensuring that the result remains sorted.

2.2 Suitability for Parallelization

Merge sort is inherently parallelizable due to its recursive nature and independent subproblems. Each division creates smaller subarrays that can be sorted concurrently, making it well-suited for execution on multicore and distributed systems. The merge step, while dependent on the completion of the sorting phase, can also be parallelized for larger subarrays.

The divide-and-conquer structure makes merge sort adaptable to both shared-memory and distributed-memory parallel architectures:

- **Shared-Memory Parallelism:** In a shared-memory model, threads can independently sort different subarrays and merge them back together, using synchronization mechanisms to manage shared data. This can be implemented efficiently using libraries like Pthreads.
- **Distributed-Memory Parallelism:** In a distributed-memory model, subarrays can be distributed

across different nodes in a cluster. Each node sorts its local data and then communicates with other nodes to perform the merge step. This model is effectively handled by MPI, which provides mechanisms for data exchange between nodes.

2.3 Challenges in Parallelizing Merge Sort

Despite its suitability for parallelization, merge sort has challenges that must be addressed:

- **Load Balancing:** Ensuring equal distribution of work among threads or nodes is critical for achieving high efficiency. Unbalanced workloads can lead to idle cores or nodes.
- **Synchronization Overhead:** In shared-memory systems, threads must coordinate access to shared data, which can introduce delays.
- **Communication Overhead:** In distributed-memory systems, the merge step requires data exchange between nodes, which can become a bottleneck for large datasets.

2.4 Use Cases

Parallel merge sort is especially beneficial for sorting large datasets where sequential algorithms may be too slow. Applications include:

- Large-scale data processing (e.g., sorting logs, databases).
- Scientific computing requiring efficient sorting of numerical data.
- Real-time systems where sorting must meet strict time constraints.

In the next section, we discuss the design and implementation of parallel merge sort using Pthreads and MPI, exploring how these architectures address the challenges mentio

3.The Parallel Implementation

In this section, we discuss two parallel implementations of the merge sort algorithm on multicore systems, utilizing Pthread and Message Passing Interface (MPI). These implementations are designed to handle large datasets efficiently by leveraging parallelism.

3.1 Pthread Implementation

The Pthread-based implementation focuses on parallelizing the sorting phase of the merge sort algorithm. The input array is divided into smaller subarrays, with each thread responsible for sorting a specific portion of the array independently. This ensures that the computational workload is distributed among the available threads.

Initialization:

The input array is shared between the threads, ensuring that all threads have access to the data they need to process. The array of size n is divided into p independent chunks, where each thread handles approximately n/p elements.

Parallel Sorting:

Each thread sorts its assigned subarray using the sequential merge sort algorithm. Once all threads have completed their local sorting, the subarrays are merged iteratively to produce the final sorted array. Algorithm 1 illustrates the parallelization of this process.

Algorithm 1: Parallel Merge Sort with Pthreads

1. Divide the input array of size n into p chunks.
2. Assign each chunk to a thread for local sorting using sequential merge sort.
3. Synchronize threads to ensure that all local sorting is complete.
4. Iteratively merge the sorted subarrays:
    - In the first iteration, merge pairs of adjacent sorted subarrays.
    - Continue merging until a single sorted array remains.
    -

Implementation Details:

- **Thread Creation:** Threads are created using the Pthread library, with each thread receiving a range of indices corresponding to its assigned chunk.
- **Synchronization:** A barrier or join operation is used to ensure all threads complete their sorting phase before proceeding to the merge phase.
- **Load Balancing:** Equal division of the array among threads minimizes idle time and ensures efficient utilization of computational resources.

The Pthread implementation is well-suited for shared-memory systems, as it leverages the ability of threads to access the same memory space. By minimizing inter-thread communication and focusing on local computation, this approach achieves significant speedup compared to the sequential implementation.

Algorithm 1: Parallelization of Step 1+2+3+4
Input :array[n],thread ID,n, thread num
Output :sorted array[n]
Chunk_size= n/thread num;

```
for (int i = 0; i < N; i++) {

        cin >> myarr[i];

    }
```

Divide the work among the threads –step (2)

```
    for (int i = 0; i < numThreads; i++) {

        args[i].low = index;

        args[i].high = index + chunkSize - 1;

        if (remainder > 0) {

            args[i].high++;

            remainder--;

        }

        args[i].id = i;

        index = args[i].high + 1;
```

/*cration for phreads*/ ------step(2)

```
if (pthread_create(&myThreads[i], NULL, threadMergeSort, (void*)&args[i]) != 0) {

            cout << "Error creating thread " << i << endl;

            return 1;

        }
```

/*for syncronization*/ -----step(3)

```
    // Wait for all threads to finish

    for (int i = 0; i < numThreads; i++) {

        if (pthread_join(myThreads[i], NULL) != 0) {
```

```
            cout << "Error joining thread " << i << endl;
            return 1;
        }
    }

// Merge Sort function to divide the work ---------step(4)
void mergeSort(vector<int>& arr, int low, int high) {
    if (low >= high) return;
    int mid = (high + low) / 2;

    mergeSort(arr, low, mid);
    mergeSort(arr, mid + 1, high);
    merge(arr, low, mid, high);
}
// Merge function that combines two sorted halves
-----step(4)
void merge(vector<int>& arr, int low, int mid, int high) {
    vector<int> temp(arr.begin() + low, arr.begin() + high + 1);
    int i = low, j = mid + 1, k = low;

    while (i <= mid && j <= high) {
        if (arr[i] <= arr[j]) {
            temp[k - low] = arr[i++];
        } else {
            temp[k - low] = arr[j++];
        }
        k++; }
```

```
    while (i <= mid) {
        temp[k - low] = arr[i++];
        k++;
    }

    while (j <= high) {
        temp[k - low] = arr[j++];
        k++;
    }

    for (int z = low; z <= high; z++) {
        arr[z] = temp[z - low];
    }
}
```

MPI Implementation

The MPI-based implementation of merge sort is designed for distributed-memory systems. Unlike the Pthread approach, where threads share the same memory space, MPI relies on multiple processes communicating via message passing. This makes MPI particularly suitable for distributed computing environments where memory is spread across multiple nodes or machines.

Initialization:

The input array is distributed across the available processes. The master process divides the array into ppp chunks, where p represents the total number of processes involved in the computation. Each process is responsible for sorting a specific chunk of the array. In this stage, the master process uses the MPI_Scatter function to distribute the data chunks to all processes.

Parallel Sorting:

Once each process receives its chunk of data, it independently sorts its portion of the array using the standard sequential merge sort algorithm. This phase runs entirely in parallel across the processes.

Merge Phase:

After sorting the individual chunks, a parallel merging phase begins. In the MPI implementation, this phase requires processes to communicate and merge their sorted subarrays. MPI provides various communication functions like MPI_Send and MPI_Recv or MPI_GATHER to exchange data between processes. The merging process typically follows a logarithmic pattern, where processes pair up and merge their sorted subarrays in multiple iterations.

Algorithm 2: Parallel Merge Sort with MPI

1. Scatter Phase:
    - The master process divides the input array into p chunks and uses MPI_Scatter to distribute the chunks to each of the pp processes.
2. Local Sorting Phase:
    - Each process sorts its assigned subarray using the sequential merge sort algorithm.
3. Merging Phase:
    - Pairs of processes merge their sorted subarrays. Initially, each process is responsible for one subarray. As merging progresses, the number of active processes is reduced in each iteration by halving the number of processes that are merging.
4. Gather Phase:
    - After all merges are complete, the results are gathered back to the master process. This is done using MPI_Gather, where the sorted subarrays are sent back from all processes to the master process.

Implementation Details:

- Data Distribution:
    - MPI_Scatter ensures an even distribution of data across processes. This avoids load imbalance and ensures that each process performs roughly the same amount of work.
- Communication:
    - A tree-based communication pattern can help minimize the overhead associated with multiple rounds of communication.

- Synchronization:
    - The synchronization in MPI is handled by the communication operations themselves. Since each process merges its subarrays in parallel, synchronization is implicitly ensured as processes wait for their peers to finish sending and receiving data.
- Scalability:
    - The MPI approach can scale well with large datasets and a high number of processes. Each additional process adds more computational power, which results in better performance for large arrays. However, communication overhead can become significant for smaller arrays, and the benefit of parallelism might diminish.

Performance Considerations:
The MPI implementation offers significant performance improvements for large datasets compared to the sequential version of merge sort. However, it is important to note that for smaller arrays, the overhead of communication between processes may counteract the benefits of parallelization. For larger arrays, the MPI implementation can achieve near-**linear speedup** as the number of processes increases, making it a powerful option for high-performance computing environments.

Algorithm 2: Parallelization MPI of Step 1+2+3+4

Input :array[n],thread ID,n, thread num
Output :sorted array[n]
Chunk_size= n/thread num;

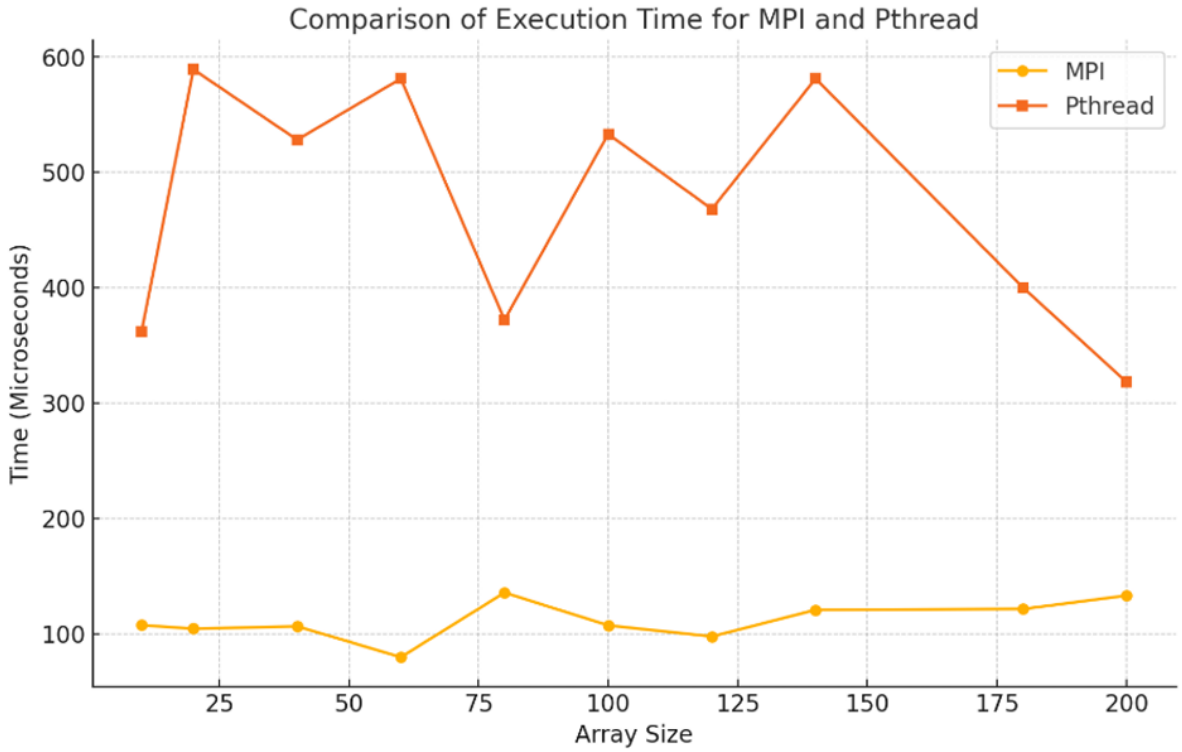
```
MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);
        MPI_Comm_size(MPI_COMM_WORLD, &nump);

/********** Divide the array in equal-sized chunks **********/
        int size = n/nump;
/********** Send each subarray to each process **********/
        int *sub_array = malloc(size * sizeof(int));
        MPI_Scatter(original_array, size, MPI_INT, sub_array, size, MPI_INT, 0, MPI_COMM_WORLD);


/********** Perform the mergesort on each process **********/
        int *tmp_array = malloc(size * sizeof(int));
        mergeSort(sub_array, tmp_array, 0, (size - 1));

/**GATHER THE SORTED Aryyas**********/
MPI_Gather(sub_array, size, MPI_INT, sorted, size, MPI_INT, 0, MPI_COMM_WORLD);
```

4.Performance Result:



6. Conclusions

In this paper, we compare two parallel implementations of the merge sort algorithm using Pthread and MPI on multicore systems.
Through performance analysis and experimental results, we evaluate the effectiveness of each parallelization strategy.
The results indicate that the MPI implementation outperforms the Pthread implementation in our case. This can be attributed to the inherent design of MPI, which efficiently handles parallelism in distributed memory systems. While MPI involves communication between processes, the communication overhead is relatively minimal compared to the computation time for large datasets, allowing for better scalability.

In contrast, while the Pthread implementation is generally efficient for shared memory systems, the synchronization and memory access patterns can create bottlenecks when the data size increases, leading to less effective performance scaling in comparison to MPI.

The experimental results show that both implementations provide near-linear speedup for medium data sizes. However, the MPI approach demonstrated superior performance due to its ability to efficiently distribute the data across multiple processes, reducing the time needed for sorting larger datasets.
In conclusion, the MPI implementation proves to be more effective for larger-scale merge sort tasks, as it achieves better performance through distributed memory handling and efficient parallelism. The Pthread implementation, while still useful for smaller-scale parallelization, exhibits limitations in scalability when compared to MPI. Both parallelization strategies are highly effective but cater to different use cases, with MPI being more suitable for larger datasets requiring better scalability