COMS3008A Assignment – Report

Sayfullah Jumoorty, 2430888

June 5, 2023

1 Problem 1: Parallel Scan

In computer science, a scan operation, also known as a prefix sum operation, is a common parallel algorithm used to compute a cumulative sum or other associative operation on a sequence of elements. The result of the scan operation is a new sequence where each element represents the accumulation of values up to that point in the original sequence.

1.1 Test Data Generation

In order to get accurate results, it is important to have an efficient data generation technique for huge sets of random numbers, and also, an efficient storage type to quickly read the data between programs:

Input: An integer exponent n.

Output: A binary file containing a sequence of random integers.

1. Read the integer exponent n from the command-line arguments.
2. Check if the number of arguments is valid. If not, print an error message and exit.
3. Convert the exponent n to an integer.
4. Check if n is a non-negative number. If not, print an error message and exit.
5. Calculate the size of the array as 2^n. This determines the number of integers to be generated.
6. Allocate memory for an array of integers with the calculated size.
7. Create a random number generator using the Mersenne Twister engine, seeded with the current system clock time.
8. Define a uniform integer distribution that generates random numbers between a specified range.
9. Iterate from 0 to the size of the array, assigning each element a random number generated by the distribution.
10. Open a binary output file for writing.
11. Write the content of the array to the output file as a sequence of bytes.
12. Free the memory allocated for the array.
13. Exit the program with a success status code.

1.2 Serial Implementation

As we are tying to get the most optimal base line algorithm, we can hence do the following for the serial program:

- Read the input array from a binary file.
- Check if the size of the input array is a power of two.
- The prefix sum calculation is performed using the scan function. The function takes the input array (nums) and produces the prefix sum array (`prefix_sum`) using a simple for loop algorithm. Each element in the prefix sum array is computed by adding the corresponding element in the input array to the sum of all previous elements. This process can be seen in *Algorithm*1

The serial approach performs all these steps sequentially, without parallelization. It serves as a baseline for performance comparison and is suitable for situations where parallelization is not required or feasible due to hardware or software limitations. The serial approach can be used for small input sizes or when the overhead of parallelization outweighs the benefits.

Function `scan`($out : vector[int], in : vector[int]$)
  $N \leftarrow$ in.size();
  out.resize(N);
  if $N > 0$ **then**
    out$[0] \leftarrow$ in$[0]$;
  end
  for $i \leftarrow 1$ **to** $N-1$ **do**
    out$[i] \leftarrow$ in$[i]$ + out$[i-1]$;
  end

Algorithm 1: Scan function

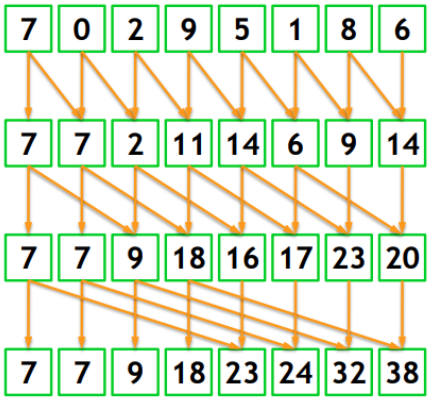

Figure 1: Process of Scan

1.3 OpenMP Implementation

In the OpenMP version of the code, several changes were made to parallelize the computation using OpenMP directives. This included using a scan method known as "Blelloch" [2]. The algorithm for scan operation in $Algoritm1$ is inherently sequential, as there is a loop carried dependence in the for loop. However, Blelloch 1990 gives an algorithm for calculating the scan operation in parallel. Here is a clear explanation of the approach, parallelization methods, correctness, and testing methods:

1. **Parallelization Approach:**
   - The code uses the OpenMP library to introduce parallelism and exploit multiple threads to speed up the computation.
   - Parallelization is applied to the "Up-sweep (reduce)" and "Down-sweep (scan)" phases of the Blelloch scan algorithm.
     - Perform the up-sweep (reduce) phase of the Blelloch scan algorithm:
       - Starting with a stride d of 1, iterate over the array.
       - Perform the reduction operation by adding each element at index $i+d-1$ to the element at index $i+2\cdot d-1$.
       - Repeat the process for increasing values of d until it reaches the array size.
       - Store the last value of the up-sweep phase.
     - Set the root of the binary tree to zero by modifying the last element of the array.
     - Perform the down-sweep (scan) phase of the Blelloch scan algorithm:
       - Starting with a stride d equal to half the array size, iterate over the array.
       - Perform the scan operation by swapping and accumulating the elements.
       - Repeat the process for decreasing values of d until it reaches 1.
     - Adjust the array size to accommodate the additional element and set the last element to the value obtained from the up-sweep phase.
     - Compute the prefix sum array.
2. **Parallelization Methods:**
   - The parallelization [5] is achieved using OpenMP directives, which control the distribution of work among multiple threads.
   - The following OpenMP directives are used in the code:
     - `#pragma omp parallel` creates a team of threads.
     - `#pragma omp for` distributes the loop iterations among the threads.
     - `#pragma omp single` ensures that a particular section of code is executed by only one thread.

3. **Up-sweep (Reduce) Phase:**
   - The up-sweep phase is parallelized using the OpenMP directive `#pragma omp for`.
   - The loop that performs the reduction operation is divided among multiple threads, with each thread handling a subset of the iterations.
   - The `reduction(+ : last)` clause is used to perform a reduction operation on the `last` variable, ensuring that each thread contributes its local sum to the final value.

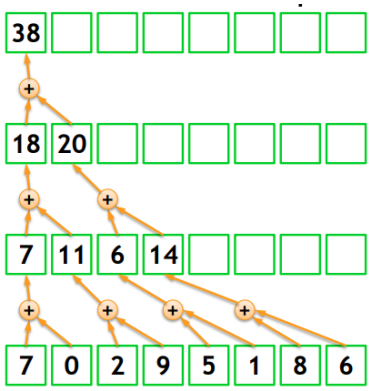

Figure 2: Up-sweep (Reduce) Phase

4. **Down-sweep (Scan) Phase:**
   - The down-sweep phase is parallelized using the OpenMP directive `#pragma omp parallel for`.
   - Similar to the up-sweep phase, the loop is divided among multiple threads, allowing them to work on different subsets of the iterations.

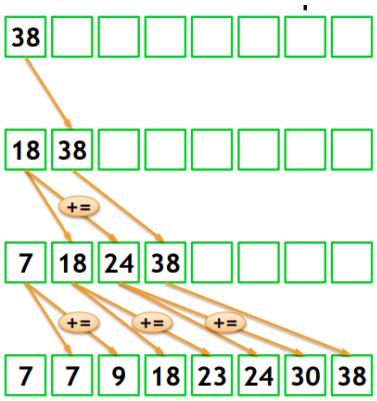

Figure 3: Down-sweep (Scan) Phase

```
Function blelloch_scan(nums : vector[int])
    n ← nums.size();
    // Up-sweep (reduce) phase
    last ← 0;
    #pragma omp parallel default(none) shared(nums, n) reduction(+ :  last);
    { for d ← 1 to n − 1 step 2 ∗ d do
        #pragma omp for;
        for i ← 0 to n − 1 step 2 ∗ d do
            nums[i + 2 ∗ d − 1] += nums[i + d − 1];
        end
    end
    #pragma omp single;
    last ← nums[n − 1];
    }
    // Set root to 0
    nums[n − 1] ← 0;
    // Down-sweep (scan) phase
    for d ← n/2 to 1 step d/2 do
        #pragma omp parallel for;
        for i ← 0 to n − 1 step 2 ∗ d do
            t ← nums[i + d − 1];
            nums[i + d − 1] ← nums[i + 2 ∗ d − 1];
            nums[i + 2 ∗ d − 1] += t;
        end
    end
    // Increase the size of the array by 1 and set the last element to the last value
       of the up-sweep phase
    nums.resize(n + 1);
    nums[n] ← last;
    // Remove the first element of the array
    nums.erase(nums.begin());
```

Algorithm 2: Blelloch scan function

5. **Correctness:**
   - The correctness of the prefix sum computation is ensured by validating the result against a temporary array using the `validatePrefixSum` function.
   - The function checks if the prefix sum values in the final array match the expected values based on the original array.
   - If a mismatch is found, the program terminates with an error message indicating that the prefix sum is invalid.
6. **Testing Methods:**
   - The code includes a `checkIfPowerOfTwo` function to verify if the size of the input array is a power of two.
   - This check is important because the Blelloch scan algorithm assumes a power-of-two input size.
   - If the size is not a power of two, the program terminates with an error message.

1.4 Graph Comparison - OpenMP

The scalability can be evaluated by varying the input size or the number of threads (t) and observing the impact on execution time. In the provided results, the execution time decreases as the number of processing elements increases (t=2 to t=8). This suggests that the code exhibits reasonable scalability, as distributing the workload among more threads leads to faster execution times.

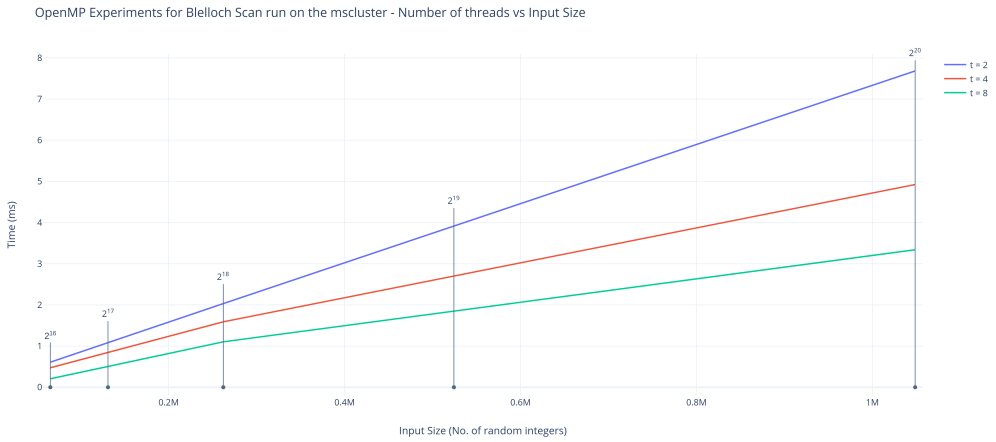

Figure 4: OpenMP Comparisons for blelloch scan

From the graph, we can see a direct relation to increasing the number of threads for different input sizes results in a speedup, and that speedup gets significantly better as the input size grows.

1.5 MPI Implementation

In the MPI [4] implementation of the code, we know solve the computation using the Message Passing Interface (MPI) library.

1. **Parallelization Approach:**
   - The code uses the MPI library to distribute the work among multiple processes and enable communication between them.
   - Parallelization is achieved by dividing the input data and computations among different MPI processes.
2. **Parallelization Methods:**
   - The MPI library provides various functions for process management, communication, and collective operations.
   - The code utilizes MPI functions such as `MPI_Init`, `MPI_Comm_size`, `MPI_Comm_rank`, `MPI_Bcast`, `MPI_Scatter`, `MPI_Gather`, and `MPI_Finalize` to achieve parallelization and coordination among processes.
3. **Differences from Serial and OpenMP Approaches:**

- Unlike the serial approach, where the computation is performed sequentially on a single thread, and the OpenMP approach, which parallelizes the computation using multiple threads within a shared memory environment, the MPI approach distributes the computation across multiple processes running on separate memory spaces.
- In the MPI implementation, each process works on a subset of the input data and performs computations independently. Communication between processes is necessary to exchange data and synchronize the results.
- MPI allows scaling the computation across multiple machines, making it suitable for distributed memory systems, clusters, or supercomputers.

4. **Approach to Solving the Problems:**
   - The input data is read from a binary file and stored in a vector on the root process (rank 0).
   - The size of each chunk of data is calculated on the root process and broadcasted to all other processes using `MPI_Bcast`.
   - The input data is then scattered among the processes using `MPI_Scatter`.
   - Each process performs the prefix sum computation on its local input data using the Blelloch scan algorithm, similar to the serial and OpenMP approaches.
   - Global sums are gathered from each process to the root process using `MPI_Gather`.
   - The root process performs the Blelloch scan on the gathered global sums.
   - Each process receives a single global sum and adds it to its local input data.
   - Finally, the modified local input data is gathered from each process to the root process, resulting in the final prefix sum array.
5. **Correctness:**
   - The correctness of the prefix sum computation is ensured by validating the result against the original input data using the `validatePrefixSum` function.
   - The function checks if the prefix sum values in the final array match the expected values based on the input data.
   - If any mismatch is found, an error message is printed, and the program is aborted using `MPI_Abort`.
6. **Testing Methods:**
   - The code measures the time taken to perform the prefix sum computation using `MPI_Wtime`.
   - The time is printed and saved to a CSV file for analysis.
   - Additionally, the correctness of the prefix sum is validated using `validatePrefixSum`.
   - The testing methods can include running the code on different input sizes, comparing the results with the sequential and OpenMP implementations, and evaluating the performance scalability with an increasing number of MPI processes.

1.6 Performance Evaluation

To evaluate the performance of the baseline (serial) version compared to the parallel version and test the scalability by varying the number of processing elements, we can analyze the provided results. We can a generalized discussion of the performance evaluation, in which results were obtained by running the different implementations on the mscluster using 4 nodes with 4 processes each, and with an input size of 2^n random integers:

Comparison of Performance (Speedup) - Serial vs. OpenMP

Table 1: Comparison of Performance (Speedup) - Serial vs. OpenMP

Description	Time (ms)	Speedup
Serial Time	18.9288 ms	–
Parallel Time	7.36385 ms	2.57

The table above compares the execution time of the serial version and the parallel version of the code. The "Serial Time" represents the time taken by the code in the serial execution mode. The "Parallel Time" represents the time taken by the code when parallelized. The "Speedup" column indicates the speedup achieved by the parallel version compared to the serial version. In this case, the parallel version achieved a speedup of 2.57, indicating that the parallel implementation is 2.57 times faster than the serial implementation.

Comparison of Performance (Speedup) - Serial vs. MPI Version

Table 2: Comparison of Performance (Speedup) - Serial vs. MPI Version

Description	Time (ms)	Speedup
Serial Time	18.9288 ms	–
MPI Time (np=4)	8.47074 ms	2.23

The table above compares the execution time of the serial version and the MPI version of the code. The "Serial Time" represents the time taken by the code in the serial execution mode. The "MPI Time (np=4)" represents the time taken by the code when parallelized using MPI with 4 processing elements. The "Speedup" column indicates the speedup achieved by the MPI version compared to the serial version. In this case, the MPI version achieved a speedup of 2.23, indicating that the MPI implementation with 4 processing elements is 2.23 times faster than the serial implementation.

Testing Scalability - Varying the Input Size or Number of Processing Elements

Table 3: Testing Scalability - Varying the Input Size or Number of Processing Elements

Description	Time (ms)	Speedup
2^{16}	1.80664 ms	1.03
2^{18}	4.30736 ms	1.74
2^{20}	8.47074 ms	2.23

The table above shows the execution times and speedup for different input sizes or number of processing elements. In this case, the input size was varied while keeping the number of processing elements fixed at 4. The "Speedup" column indicates the speedup achieved by the parallel MPI version compared to the serial version. As the input size increased from 2^{16} to 2^{20}, the speedup also increased from 1.03 to 2.23, indicating that the code exhibits good scalability.

Overall, the parallel version (using either OpenMP or MPI) outperforms the serial version in terms of speedup, demonstrating the benefits of parallelization. The MPI version shows improvements compared to the serial version, indicating

the effectiveness of MPI parallelization. Furthermore, the code exhibits good scalability by achieving faster execution times as the input size increases. Further scalability analysis can be performed by varying the input size or number of processing elements in future evaluations.

1.7 Graph Comparison - MPI

The scalability can be evaluated by varying the input size or the number of processing elements (np) and observing the impact on execution time. In the provided results, the execution time decreases as the number of processing elements increases (np=2 to np=4). This suggests that the code exhibits reasonable scalability, as distributing the workload among more processes leads to faster execution times.

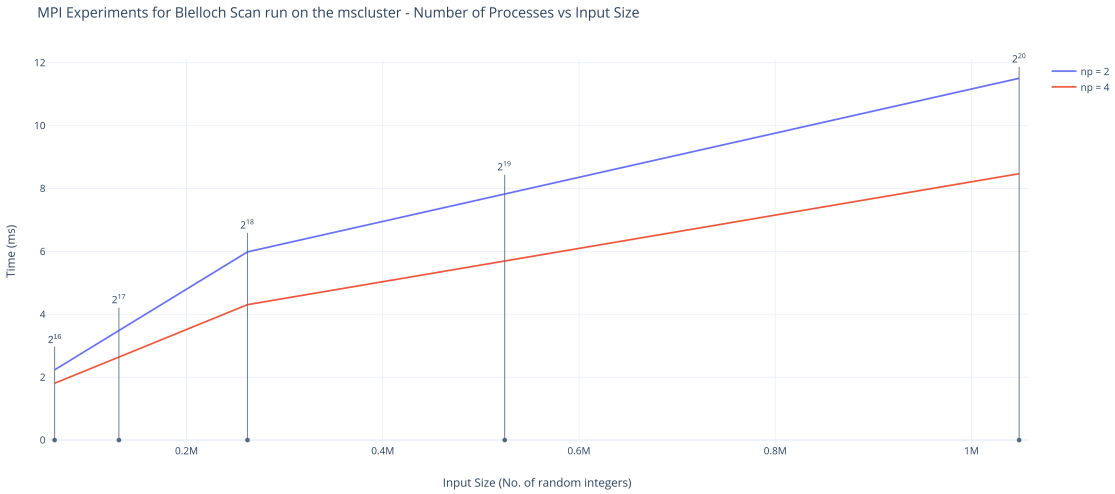

Figure 5: MPI Comparisons for blelloch scan

From the graph, we can see a direct relation to increasing the number of processes for different input sizes results in a speedup, and that speedup gets significantly better as the input size grows.

2 Problem 2: Parallel Bitonic Sort

- The **bitonic sort** algorithm is based on the concept of a **sorting network**.
- This algorithm is particularly **well-suited for parallel processing**, making it a popular choice for sorting on **Graphics Processing Units (GPUs)**.
- It **efficiently sorts** a given list or array of elements in ascending or descending order.

2.1 Test Data Generation

The method mentioned in 1.1 is also used for generation of data for the various bitonic approaches, in order to get valuable and meaningful results.

2.2 Serial Implementation

In order to solve the problem of sorting in a bitonic [1] manner, we can break it apart and think of it as sections:

- **Bitonic Sequence Validation**:
  - The input sequence is checked to determine if it is a valid bitonic sequence.
  - A bitonic sequence is defined as a sequence of numbers that first increases and then decreases (or vice versa).
  - The program compares each element in the sequence with its adjacent element and checks if the order is consistent with the desired direction.
  - If the order is violated at any point, an error message is displayed, indicating that the sequence is not a valid bitonic sequence.
- **Power of Two Check**:
  - The size of the input vector is checked to ensure that it is a power of two.
  - A power of two is a number that can be expressed as 2^n, where n is a non-negative integer.
  - This check is performed to ensure that the sorting algorithm works correctly.
  - If the size of the input vector is not a power of two, it indicates an invalid input, and the program exits.

- **Bitonic Merge**:
  - The bitonic merge operation is a key step in the Bitonic Sort algorithm.
  - It merges two sorted sequences in a bitonic manner, resulting in a sequence that is either strictly increasing or strictly decreasing.
  - The algorithm compares the elements at corresponding positions in both sequences and swaps them if they violate the desired order.
  - This process is repeated recursively until the entire sequence is merged.

```
Function bitonicMerge(arr : vector[int], low : int, cnt : int, direction : bool)
    if cnt > 1 then
        k ← cnt/2;
        for i ← low to low + k − 1 do
            if direction = (arr[i] > arr[i + k]) then
                swap(arr[i], arr[i + k]);
            end
        end
        bitonicMerge (arr, low, k, direction);
        bitonicMerge (arr, low + k, k, direction);
    end
```

Algorithm 3: Bitonic merge function

- **Bitonic Sort**:
  - The Bitonic Sort algorithm is based on the concept of bitonic sequences and bitonic merge.
  - It recursively divides the input sequence into two halves, sorts each half independently in a bitonic manner, and then performs bitonic merge operations to merge the sorted halves.
  - This process is repeated until the entire sequence is sorted in the desired order.

```
Function bitonicSort(numbers : vector[int], left : size_t, size : size_t)
    groupSize ← 2;
    itr ← log2(size);
    increasing ← true;
    for i ← 0 to itr − 1 do
        for j ← 0 to size − 1 step groupSize do
            increasing ← ((j/groupSize) mod 2 = 0);
            bitonicMerge(numbers, left + j, groupSize, increasing);
        end
        groupSize ← groupSize × 2;
    end
```

Algorithm 4: Bitonic sort function

- **Testing and Timing**:
  - The main function of the program handles the input and output operations.
  - It reads a sequence of numbers from a file, performs the bitonic sort algorithm on the sequence, and measures the execution time.
  - The sorted sequence is validated to ensure that it is a valid bitonic sequence.
  - The execution time is recorded and saved to a file for further analysis.

The correctness of the approach can be evaluated through rigorous testing, including unit tests, integration tests, boundary tests, and performance tests. These tests cover various scenarios and inputs to ensure that the code functions correctly and produces accurate results.

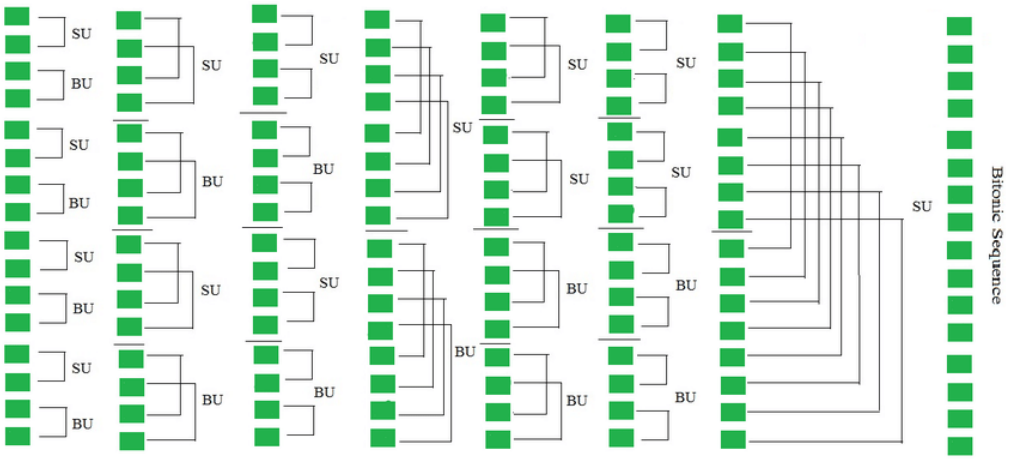

Figure 6: Bitonic Sequence

2.3 OpenMP Implementation

Parallelization Approach:

The parallelization [5] approach in the code aims to improve the performance of the bitonic sort algorithm by utilizing parallel execution of sorting operations. Parallelization involves dividing the workload among multiple threads to execute tasks concurrently. This approach takes advantage of the parallel processing capabilities of modern multi-core or multi-processor systems to speed up the sorting process.

Parallelization Methods:

1. **Parallel For Loop:** The main parallelization method used in the code is the `#pragma omp parallel for` directive. It parallelizes the outer loop in the `bitonicSort` function, which iterates over different group sizes. By adding this directive, the iterations of the loop are distributed among multiple threads, allowing them to work on different parts of the sequence concurrently. This parallelization method effectively divides the workload and reduces the overall execution time.
2. **Private Variables:** To ensure data integrity and prevent data races, the `private(increasing)` clause is added to the parallel for loop directive. This makes each thread have its private copy of the `increasing` variable, which is used within the loop. By making the variable private to each thread, conflicts that may occur during concurrent execution are avoided.

Function `bitonicSort`(*numbers* : *vector*[*int*], *left* : *size_t*, *size* : *size_t*)
- groupSize ← 2;
- itr ← $\log_2$(size);
- increasing ← true;
- **for** $i \leftarrow 0$ **to** $itr - 1$ **do**
  - #pragma omp parallel for private(increasing);
  - **for** $j \leftarrow 0$ **to** $size - 1$ *step groupSize* **do**
    - increasing ← $((j/\text{groupSize}) \bmod 2 = 0)$;
    - bitonicMerge(numbers, left + j, groupSize, increasing);
  - **end**
  - groupSize ← groupSize × 2;
- **end**

Algorithm 5: OpenMP Bitonic sort function

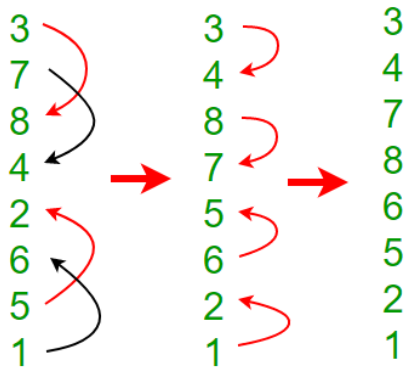

Figure 7: Sort Method

Correctness and Testing Methods:

To ensure the correctness of the parallelized approach, rigorous testing methods can be employed. This includes:

1. **Functional Testing:** Test the sorting algorithm with various inputs, including different bitonic sequences, to verify that the parallelized code produces the expected sorted output. This testing should cover both ascending and descending order sequences.
2. **Validation Testing:** Validate the sorted sequence to ensure that it is a valid bitonic sequence. Compare the sorted sequence with the original input sequence and check if it satisfies the bitonic property (either strictly increasing or strictly decreasing). This validation step ensures the correctness of the sorting algorithm.
3. **Boundary Testing:** Test the parallel code with edge cases and boundary conditions, such as small input sizes, large input sizes, and extreme values. This helps identify any potential issues or limitations of the parallelization approach.
4. **Performance Testing:** Measure the execution time of the parallelized code with different input sizes and compare it with the sequential version. Performance testing allows for evaluating the speedup achieved by parallel execution and assessing the efficiency of the parallelization methods used.

By conducting these testing methods, the correctness of the parallelized approach can be thoroughly evaluated, and any issues or bugs can be identified and addressed. It is crucial to ensure that the parallel code produces accurate results while achieving improved performance.

2.4 Graph Comparison - OpenMP

The scalability can be evaluated by varying the input size or the number of threads (t) and observing the impact on execution time. In the provided results, the execution time decreases as the number of processing elements increases (t=2 to t=8). This suggests that the code exhibits reasonable scalability, as distributing the workload among more threads leads to faster execution times.

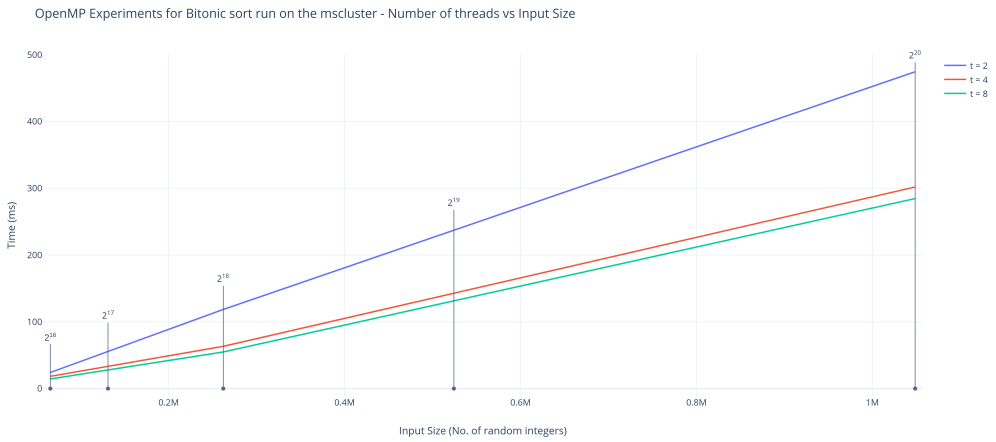

Figure 8: OpenMP Comparisons for bitonic sort

From the graph, we can see a direct relation to increasing the number of threads for different input sizes results in a speedup, and that speedup gets significantly better as the input size grows.

2.5 MPI Implementation

Approach:

The MPI (Message Passing Interface) implementation aims to parallelize the bitonic sort algorithm by using a distributed computing approach. Unlike the serial and OpenMP versions, which utilized shared memory and threads within a single machine, the MPI implementation utilizes multiple processes running on different machines or nodes within a cluster. Each process has its own memory space and executes a portion of the sorting algorithm independently. The processes communicate with each other by passing messages to exchange data and perform collective operations.

Parallelization Methods:

1. **Process Initialization:** The MPI library is initialized using the `MPI_Init` function, which assigns a unique rank to each process and determines the total number of processes in the MPI communicator. The `MPI_Comm_rank` and `MPI_Comm_size` functions are used to obtain the rank and size of the MPI communicator, respectively.
2. **Data Distribution:** The input data is divided among the processes using the `MPI_Scatter` function. The root process broadcasts the entire input array to all other processes using the `MPI_Bcast` function. Each process receives a portion of the input array, which it will independently sort.
3. **Bitonic Sort:** Each process performs the bitonic sort algorithm on its local portion of the array using the `bitonic_sort` function. The local sorting is similar to the serial version, where each process sorts its portion of the array independently. This step is performed in parallel by all processes.
4. **Bitonic Merge:** After the local sorting is completed, the processes perform a bitonic merge operation using the `bitonic_merge` function. The merge operation is similar to the serial and OpenMP versions, where elements from different processes are compared and exchanged to create a globally sorted array. This step is performed in parallel by all processes.
5. **Data Gathering:** The sorted subarrays from each process are gathered and merged by the root process using the `MPI_Gather` function. The root process receives the sorted subarrays from all other processes and merges them into a single globally sorted array.

Correctness and Testing Methods:

To ensure the correctness of the MPI implementation, similar testing methods as the serial and OpenMP versions can be employed.

By conducting these testing methods, the correctness of the MPI implementation can be thoroughly evaluated, and any issues or bugs can be identified and addressed. It is crucial to ensure that the parallel code produces accurate results while achieving improved performance in a distributed computing environment.

2.6 Performance Evaluation

Comparison of Performance (Speedup) - Serial vs. OpenMP

To compare the performance between the baseline (serial) and parallel versions of the code, we can calculate the speedup. Speedup measures the improvement in execution time achieved by parallelization compared to the serial version. The following table shows the execution times for the serial and parallel versions, along with the calculated speedup:

Table 4: Comparison of Performance (Speedup) - Serial vs. OpenMP

Description	Time (ms)	Speedup
Serial Time	1020.11 ms	–
Parallel Time	252.126 ms	4.04

The speedup is calculated as the ratio of the serial time to the parallel time:

$$\text{Speedup (Serial vs. Parallel)} = \frac{\text{Serial Time}}{\text{Parallel Time}} = \frac{1020.11 \text{ ms}}{252.126 \text{ ms}} = 4.04$$

This indicates that the parallel version is approximately 4.04 times faster than the serial version.

Comparison of Performance (Speedup) - Serial vs. MPI Version

Similarly, we can compare the performance between the serial version and the MPI version with 4 processing elements. The following table presents the execution times and speedup for this comparison:

Table 5: Comparison of Performance (Speedup) - Serial vs. MPI Version

Description	Time (ms)	Speedup
Serial Time	1020.11 ms	–
MPI Time (np=4)	152.6 ms	6.68

The speedup is calculated as the ratio of the serial time to the MPI time:

$$\text{Speedup (Serial vs. MPI)} = \frac{\text{Serial Time}}{\text{MPI Time (np=4)}} = \frac{1020.11 \text{ ms}}{152.6 \text{ ms}} = 6.68$$

This indicates that the MPI [4] version achieves approximately 6.68 times better performance than the serial version.

Testing Scalability - Varying the Input Size or Number of Processing Elements

To test the scalability of the algorithm, the input size or the number of processing elements can be varied.

Scalability refers to the ability of a parallel algorithm to maintain or improve its performance as the problem size or the number of processing elements increases. In this case, the scalability can be evaluated by analyzing the execution times for different input sizes while keeping the number of processing elements constant (e.g., np=4).

The following table shows the execution times for different input sizes:

Description	Time (ms)	Speedup
2^{16}	19.375 ms	2.40
2^{18}	47.6291 ms	4.62
2^{20}	152.6 ms ms ms	6.68

In summary, the performance evaluation reveals that the parallel version of the code achieves significant speedup compared to the serial version. Additionally, the MPI version with 4 processing elements achieves even better speedup. The algorithm demonstrates scalability as the input size increases, resulting in reduced execution times.

2.7 Graph Comparison - MPI

To further evaluate scalability, additional measurements can be taken by varying the input size and analyzing the execution times for different input sizes and number of processing elements. This can help determine if the code scales efficiently as the problem size or resources increase

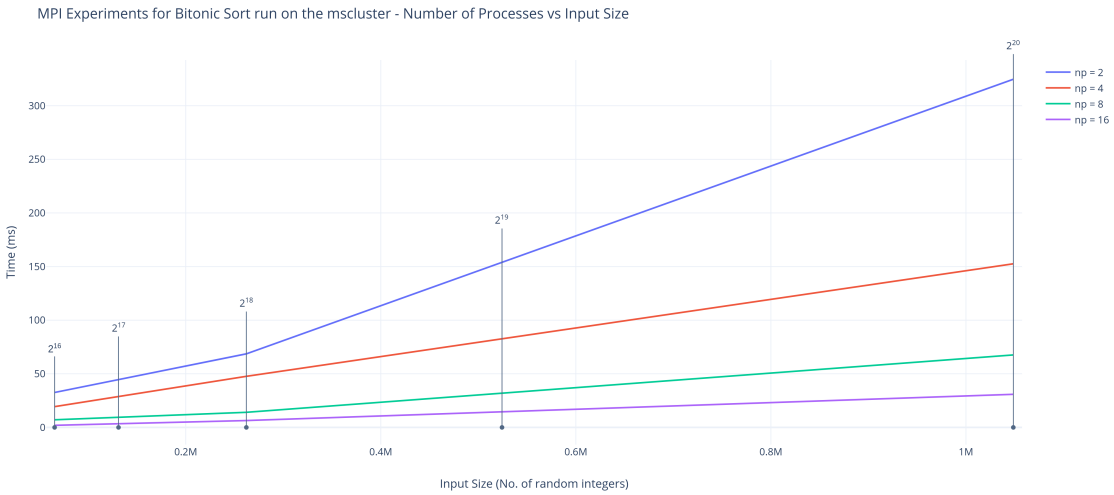

Figure 9: MPI Comparisons for bitonic sort

From the graph, we can see a direct relation to increasing the number of processes for different input sizes results in a speedup, and that speedup gets significantly better as the input size grows.

3 Conclusion

Using OpenMP [3] and MPI [6] can significantly improve the performance of parallel algorithms like scan and bitonic sort compared to their serial counterparts. By leveraging multiple threads or processes, these parallel programming models allow for efficient utilization of modern multi-core processors and distributed computing systems.

OpenMP enables parallelism at the thread level, where the computation is divided among multiple threads that execute concurrently. By using OpenMP directives, such as parallel for loops or task-based parallelism, the workload can be distributed across threads, leading to improved execution times. Adjusting the number of threads in OpenMP allows for fine-grained control over the level of parallelism and can result in noticeable speed-ups.

On the other hand, MPI allows for parallelism at the process level, enabling computations to be distributed across multiple nodes in a distributed memory system. With MPI, data is explicitly exchanged between processes, facilitating efficient communication and coordination. By carefully orchestrating the communication patterns and workload distribution, significant speed-ups can be achieved in parallel algorithms.

When applying OpenMP and MPI to scan and bitonic sort algorithms, the speed-up is consistently observed by increasing the number of threads or processes. As the workload is divided among more computational units, the overall computation time decreases due to parallel execution and efficient resource utilization. By effectively balancing the workload and minimizing communication overhead, OpenMP and MPI allow for scalability, enabling larger problem sizes to be tackled efficiently.

In conclusion, the combination of OpenMP and MPI, along with adjusting the number of threads or processes, offers a powerful approach to improve the performance of parallel algorithms like scan and bitonic sort. Leveraging the parallelism provided by these programming models can lead to significant speed-ups, enabling efficient processing of large-scale data and computationally intensive tasks.

References

[1] Kenyon F Bitonic. Sorting networks and their applications. *Proceedings of the IEEE*, 68(3):366–377, 1980.

[2] Guy E Blelloch. Prefix sums and their applications. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 330–334. IEEE, 1990.

[3] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2008.

[4] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. The MIT Press, 1999.

[5] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2003.

[6] James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Elsevier, 2013.