A2 Report

Overview:

This report is a comprehensive analysis of the performance and scalability of an MPI-based program developed for sorting small integers in a distributed memory environment. The objective is to evaluate the scalability and efficiency of the implemented solution across different processor counts and problem sizes.

Improvement in the code

The earlier code, which was submitted in A2Q1, all the numbers were collected on the first processor which could lead to memory related issues. In the new code the issue has been fixed, and all the work is properly divided into all the processors.

What is the Function?

The implemented MPI program sorts an array of small integers distributed across multiple processors. The array is partitioned among p processors, each counting occurrences of integers within a defined range. A global reduction aggregates these counts, ensuring equal keys are assigned to the same processor while maintaining order.

The core of this implementation is the *isort* function, which uses non-blocking communication to distribute and collect keys. Each processor creates a local bucket for its integers and participates in a collective operation to form a global bucket. This approach efficiently determines the distribution of elements and constructs the sorted output for each processor in a distributed memory environment.

Code Complexity:

Let n denote the total number of elements distributed across p processors. The key steps are:

Local Counting: Each process iterates over its subset of approximately n/p elements to build a local frequency bucket. This operation is O(n/p) per process.

Global Reduction: The MPI_Allreduce call operates on a fixed-size array of length RANGE (a constant, 65536) and uses a tree-based algorithm. This results in O(log p) time complexity for this step.

Bucket Assignment and Local Output Generation: Assigning ranges of values to each process and then generating the locally sorted subset is again proportional to n/p per process. The assignment step is O(1) since it depends only on the fixed RANGE, and producing the final sorted subset is O(n/p).

Combining these steps, the dominant per-process costs are O(n/p) for local operations and O(log p) for the collective communication. The total time complexity is approximately O (n/p + log p).

Parallel Code Complexity: O(n/p + log p)

Sequential Code Complexity: O(n)

Experimental Setup:

Hardware and Software Requirements

To replicate the readings please use the below mentioned specifications:

- **CPU:** Intel(R) Xeon(R) Gold 6230
- **RAM:** 128GB

- **Compiler:** mpicxx
- **Compiler Command:** `mpicxx -std=c++17 -O2   a2.cpp   -o a2`

Input Parameters

The input size n (representing the number of integers to sort) and the number of tasks per node were adjusted to evaluate the scalability of the implemented sorting algorithm. Several distinct values of n were tested, ranging from 10 billion to 160 billion. Specifically, n values of 10 billion, 20 billion, 40 billion, 80 billion, and 160 billion were used. The number of tasks per node varied from 1 to 16, while the number of nodes was fixed at 8.

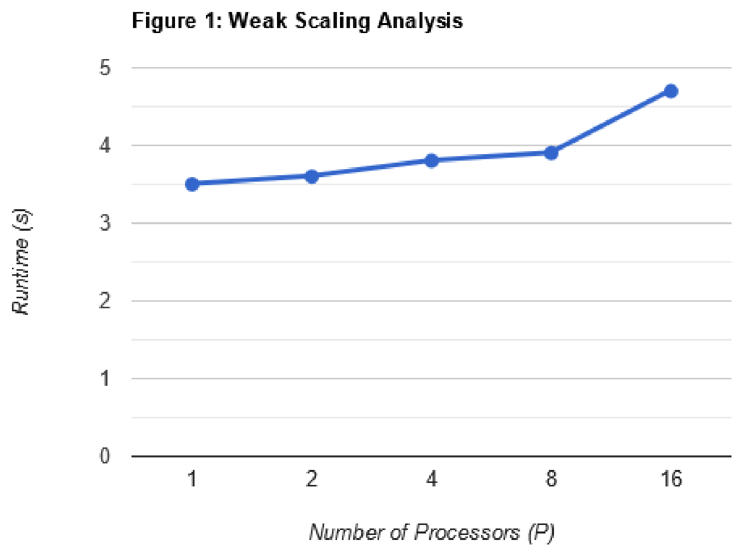
Experiment Results

The table below shows the result of the experiment conducted. These values can now be assessed to understand the scalability and performance of the code with different problem sizes(n) and number of tasks per node. The execution times, reported in seconds, represent the average of three independent runs, ensuring statistical reliability.

Table showing Tasks per Node, Input value n and Runtime(s) when nodes = 8			
Tasks per node	Total input size (n)	Runtime (seconds)	Efficiency
1	10,000,000,000	3.547	-
2	20,000,000,000	3.598	98.6%
4	40,000,000,000	3.792	93.5%
8	80,000,000,000	3.990	88.9%
16	160,000,000,000	4.732	74.9%

Weak Scaling Analysis:

The sorting algorithm demonstrated strong weak scaling efficiency for smaller task counts, maintaining over 90% efficiency up to 4 tasks per node. However, efficiency declined at higher task counts due to the increasing overhead. Future optimizations could focus on reducing communication and synchronization costs to improve performance at higher scales.



Conclusion

The MPI-based sorting algorithm demonstrated strong weak scaling performance, maintaining over 90% efficiency for up to 4 tasks per node. However, efficiency declined to 74.9% at 16 tasks per node, primarily due to increased communication and synchronization overheads. The algorithm's complexity, dominated by O(n/p) for local operations and O(logp) for communication, scales well for large problem sizes. Overall, the program exhibits robust scalability, making it suitable for distributed memory sorting tasks.