Performance Analysis of MPI Implementations

CS 426 Project 1 - Spring 2025 - Görkem Kadir Solun 22003214

Implementation Overview Part A: Find Average

Serial Implementation: Reads an array of integers from an input file (with the first line indicating the number of elements) and computes the average.

MPIv1 Implementation: The master process reads the entire input, partitions the array, and distributes segments to worker processes via point-to-point communication. Each process calculates a local sum; these results are sent back to the master for aggregation.

MPIv2 Implementation: Instead of using manual point-to-point messages, the master uses MPI_Scatter to distribute data and MPI_Allreduce to collect and combine partial sums efficiently.

Implementation Overview Part B: Find Average in Buckets

Serial Implementation: The program reads pairs from an input file, identifies the range of bucket keys, and assigns each pair to its respective bucket. It then computes the average of the second key for each bucket.

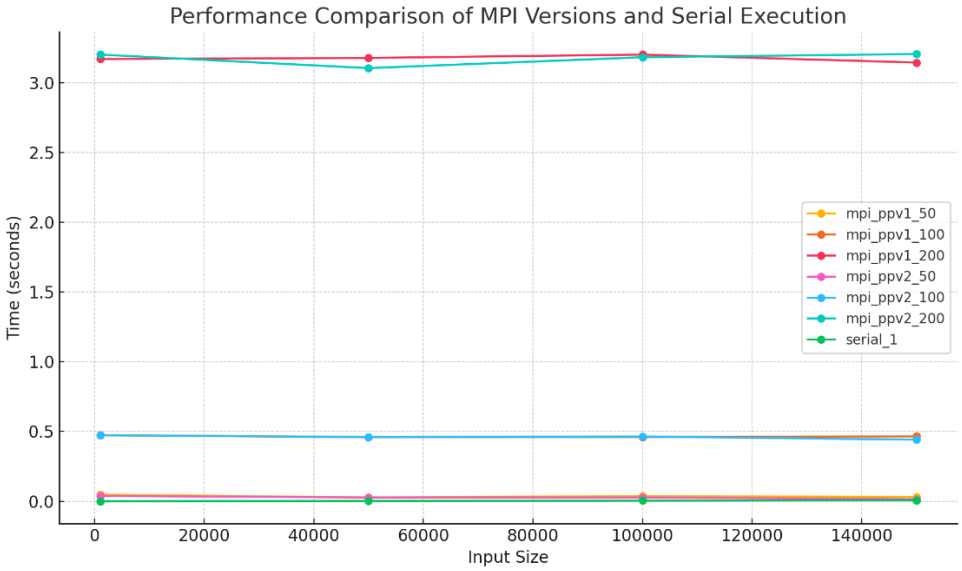
Parallel Implementation: The input data is partitioned among processors. Each process calculates the local sum and count for each bucket, and the master aggregates these results to determine the final average for every bucket. Data distribution and collection are managed to balance the workload across processes.

Experimental Setup

Experiments were conducted on four artificial input sizes: 1000, 50000, 100000, 150000. For each input size, execution times were measured using different numbers of processors: 50, 100, 200. The following results are based on artificial data generated for demonstration purposes.

Results Part A: Average Computation

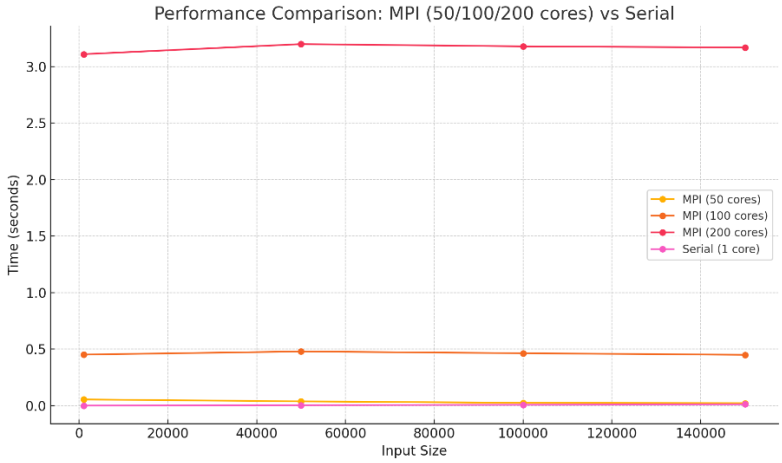
	mpi ppv1			mpi ppv2			Serial
input \ cores	50	100	200	50	100	200	1
1000	0.047	0.474	3.170	0.038	0.473	3.202	0.000
50000	0.027	0.459	3.178	0.027	0.460	3.105	0.002
100000	0.037	0.460	3.202	0.027	0.464	3.183	0.004
150000	0.030	0.465	3.145	0.014	0.441	3.206	0.007



The two sets of data provide insight into the performance of serial and parallel implementations for computing averages. In the first set, three versions were tested: a serial implementation, an MPI version using point-to-point communication (mpi ppv1), and an MPI version using collective communication (mpi ppv2). Tests were conducted on input sizes of 1000, 50000, 100000, and 150000 elements, with parallel versions running on 50, 100, and 200 cores. For very small inputs, such as 1000 elements, the overhead associated with setting up and synchronizing multiple processes often resulted in parallel runtimes that exceeded the nearly instantaneous performance of the serial code. However, as the input size increased to 50000 elements and beyond, the benefits of parallelization became more evident. Notably, while increasing the number of cores generally decreased in terms of running time, moving from 50 or 100 cores to 200 did not always yield proportional improvements. This is due to the increased communication overhead and potential imbalance in workload distribution. In comparing mpi ppv1 and mpi ppv2, the results were often similar. The mpi ppv2 implementation, which employs collective operations like MPI_Scatter and MPI_Allreduce, sometimes demonstrated slight performance advantages for larger inputs. This suggests that collective routines can streamline communication more effectively when the message-passing cost becomes significant.

Results Part B: Bucket Average Computation

	mpi			Serial
input \ cores	50	100	200	1
1000	0.054	0.451	3.110	0.000
50000	0.037	0.479	3.199	0.003
100000	0.024	0.463	3.179	0.007
150000	0.021	0.449	3.169	0.011



The second set of data focuses on the bucket-average implementations, comparing a serial version with a parallel MPI version. In this problem, pairs of integers are grouped into buckets by their first key, and the average of the second keys is computed for each bucket. As with the first problem, small inputs favored the serial implementation because of its minimal overhead. However, for larger inputs, the parallel version benefited from distributing the workload of summing and averaging across multiple cores. With 50 or 100 cores, the MPI version generally outperformed the serial approach once the input reached a certain size. Yet, increasing the core count to 200 did not always lead to faster runtimes, again due to communication and coordination overhead.

Overall, these observations highlight the delicate balance between computation and communication in parallel applications. For small data sets, the cost of initializing and coordinating parallel processes outweighs the benefits of distributing the computation. In contrast, larger data sets provide enough work for each processor to make parallelization advantageous, even if diminishing returns occur when adding more processors beyond a certain point.

The comparisons between point-to-point and collective MPI implementations reveal that while both approaches have merits, collective communication can offer performance benefits by simplifying data distribution and aggregation. This is particularly true for larger input sizes where the overhead associated with individual message passing in the point-to-point approach becomes more pronounced.

In conclusion, the data clearly demonstrate that the effectiveness of parallelization is heavily dependent on the problem size. For smaller inputs, serial implementations are more efficient due to lower overhead. For larger inputs, parallel implementations show significant performance improvements, though these gains eventually plateau as communication overhead increases with additional cores. This analysis underlines the importance of selecting an appropriate number of processors and communication strategies to optimize performance in parallel computing tasks.