Modelling and benchmarking OpenMPI collective operation using Orfeo cluster

Author's Loletti Barbara (mat. SM3800010)
High Performance Computing final exam, a.a. 2023/24
University of Trieste, master degree in Data Science and Artificial Intelligence (DSAI)

Abstract: This documents reports the procedure used and the results obtained for the Exercise 1 of the final exam of High Performance Computing. The assigned task consists in collecting data about the timing of blocking collective operations implemented within OpenMPI and using a well-know OSU benchmark suite. The computations have been performed on the Orfeo cluster and the data that have been produced, together with the architecture of interest, are analyzed in order to obtain a performance model for the different collective operations which estimates the running time for a given number of processes. In the context of this task, the collective operations that have been analyzed are the Broadcast and the Gather ones, choosing for the first the Chain and Binary Tree algorithms and the Basic Linear and Binomial algorithms for the latter. As benchmarks were run with a varying number of processes allocated on CPU cores, the CPU latency is also measured using the OSU benchmarking suite. This step is crucial in order to establish the communication overhead between processes run on different cores/nodes. The final model that has been obtained satisfyingly resemble the collected data for the different operations and algorithms, providing an explainable approximation in accordance with the architectural characteristics of the nodes on which the benchmarks er performed.

1 Introduction

In the field of high-performance computing (HPC), efficient communication between processes is crucial for achieving optimal performance. Collective operations, which involve communication patterns where data is exchanged among multiple processes, are particularly important. This report focuses on modeling and benchmarking the performance of OpenMPI collective operations. OpenMPI is a widely used implementation of the Message Passing Interface (MPI) standard, providing a robust framework for developing parallel applications. Collective operations are essential in parallel computing as they facilitate efficient data exchange among processes, including operations like Broadcast, Gather, Scatter, and Reduce. In OpenMPI, these operations are optimized to minimize communication overhead and maximize data throughput. Understanding and modeling the performance of these operations helps in optimizing parallel applications and improving overall computational efficiency. By analyzing the performance of collective operations, we aim to develop a performance model that can predict the running time of these operations based on the number of processes involved.

1.1 Software

Benchmarks were performed to obtain baseline timing data for chosen algorithms of the selected collective operations and to gather information about CPU communication latency among cores with different affinity. The utilized benchmark suite is provided by OSU (Ohio State University) and is publicly available (source and documentation at [2]) and applicable to a number of different parallel computing protocols, implementations, and paradigms. Bindings exist for different programming languages as well. For the sake of this assignment, the `osu_bcast` and `osu_barrier` C programs were compiled and executed to gather timing information about the corresponding collective operations, specifying for each one the chosen algorithm(s) to be used during the tests. The `osu_latency` program was used to measure inter-process and inter-node communication latency, deemed relevant for deriving a parallel performance model. The executables were fed to the *mpirun* program, provided by the OpenMPI module (specifically, version 4.1.5 using GNU C compilers), which handles the allocation of processes on parallel hardware, specifying the number of processes and processor affinity details to be applied.

1.2 Hardware

All computations and benchmarks were performed on the Orfeo cluster, specifically using two THIN nodes. Each THIN node in the Orfeo cluster is equipped with multiple CPU cores, providing a suitable environment for testing the scalability and performance of collective operations. The THIN partition consists of 12 Intel nodes: two equipped with Xeon Gold 6154 processors and ten equipped with Xeon Gold 6126 processors. Each node features dual-socket configurations, with each socket hosting multiple CPU cores, and is interconnected via high-speed InfiniBand networking. This setup ensures low-latency communication between processes running on different nodes. The ten nodes with Intel Xeon Gold 6126 processors have a base clock speed of 2.6 GHz and a boost clock speed of 3.7 GHz, providing a total of 24 CPU cores (2x12) per node, along with 768 GiB of memory. The two nodes with Intel Xeon Gold 6154 processors have a base clock speed of 3.0 GHz and a boost clock speed of 3.7 GHz, providing a total of 36 CPU cores (2x18) per node, along with 1536 GiB of memory. All nodes are connected via a 100 Gb/s InfiniBand network and do not have GPUs. These nodes are based on the "Skylake" architecture. The memory configuration includes 768 GiB (12 x 64 GiB) of DDR4 RAM at 2666 MT/s. Each node's 12 CPU cores are arranged in a single NUMA node. The sockets are connected by an Ultra Path Interconnect (UPI) consisting of three links operating at either 10.4 GT/s or 9.6 GT/s. The performance characteristics of the Orfeo cluster, including CPU latency and inter-node communication overhead, were taken into account when developing the performance model

for the collective operations.

2 The experiment

Experimental trials have been performed under fixed conditions using the hardware and software setups described in 1.1 and 1.2. Mapping to *core* was specified when running benchmarks using *mpirun*. Considering that each benchmark was run on two fully reserved node, each with 24 cores available with a grand total of 48 cores available. All benchmark were configured with 100 warm-up iterations

2.1 Latency estimation

Latency tests were run specifying pairs of physical cores in order to obtain timing across different regions of the available CPUs. Six critical points were identified, inside each of which the value for latency can be considered reasonably constant. Latency values for such critical points are shown in Table 1.

Core	Latency (μs)
1	0.10
4	0.20
8	0.30
16	0.50
24	0.80
48	2.0

Table 1: *Latency for communication of the given core with core #0*

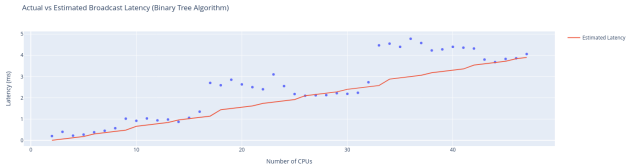
2.2 Broadcast

The Broadcast operation has been tested using the Chain and the Binary Tree algorithms. As reported in [1], in the Chain algorithm each node serves as the root of a tree with only one child which is also its successor node. On the other hand, the Binary Tree algorithm, as the name implies, builds a balanced binary tree in order to propagate the *send* operation to each layer. In the latter case, the root only communicates with the first layer of the built tree, so its left and right children. After, the nodes of this layer propagate the message to their children nodes, ensuring that the leaves will be reached. height=0.6

Chain Broadcast The data that have been collected about the Chain Broadcast algorithm shows a that both the actual and the estimated latency for the algorithm show a general increasing trend with the number of CPUs. Initially, the model aligns reasonably well with the actual measurements. However, as the number of CPUs increases beyond 15, a significant divergence is observed and the actual latency values begin to scatter more and show a higher variance compared to the estimated linear progression. The model that has been used is

$$T(P) = \gamma \cdot \sum_{i=0}^{P-1} L_{i,i+1} \tag{1}$$

where $L_{i,i+1}$ represents the CPU latency between the node i and the node $i+1$, $\gamma = 0.6$ is the overlap factor. The discrepancy we're witnessing is likely due to factors such as network contention, where increased communication traffic causes delays, and memory access patterns, where accessing data across different memory regions introduces additional latency. For smaller systems, the model's accuracy is acceptable, but for larger systems, it becomes less reliable, indicating the need for a more sophisticated model that incorporates non-linear terms and additional influencing factors for precise latency prediction. The model compared with the collected data is shown in Figure 2.



Binary Tree Broadcast In this case we're investigating the actual versus estimated broadcast latencies for a range of CPUs on THIN nodes. The estimation model used here considers the hierarchical structure of the binary tree, incorporating latency values between different regions of the architecture. By looking at the image portraying the comparison between the model chosen to represent the expected latency and the actual data we got, we can see how the actual broadcast latencies have more or less the same values as the expected one. But when the CPUs starts to be around 15, the real latency starts to skyrocket to then drastically drop and take values way below the ones for the expected latency. This same behaviour repeaters itself after the 30 CPUs mark. This actual latency occasional divergence from the estimated linear trend indicates that while the model captures the overall trend, it fails to account for the complexities and variability in the real system's performance. The observed discrepancies can be attributed to network contention, increased communication overhead, and non-uniform memory access (NUMA) effects, which are not fully captured by the simple linear model. For better accuracy, especially in larger systems, a more refined model that includes these factors is necessary. The model is compared with the collected data reported in Figure 3.

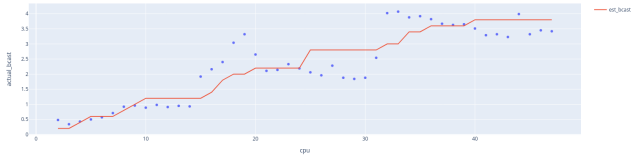


Figure 1: *Comparison between actual execution time (crosses) and estimated (line) execution time of the Broadcast collective using the Binary Tree algorithm, from 2 to 48 processes*

2.3 Gather

The gather operation is a fundamental collective communication operation in high-performance computing environments, particularly within the Message Passing Interface (MPI) framework. Its primary purpose is to collect data from multiple processes and assemble it at a designated root process. During a gather operation, each participating process

sends a block of data to the root process. The root process then receives these data blocks and concatenates them into a single, contiguous block of data. This operation is crucial for tasks that require data aggregation from various sources, such as in parallel computing scenarios where distributed computations must be consolidated for further processing or analysis. To achieve this, different algorithms can be employed, with two common ones being the basic linear gather and the binomial gather. The basic linear gather algorithm is straightforward: each process sends its data directly to the root process, which collects the data sequentially. While easy to implement, this method can become inefficient as the number of processes increases, leading to a communication bottleneck at the root process. In contrast, the Binomial Gather algorithm improves scalability by organizing the processes in a binomial tree structure. In this method, data gathering happens in a logarithmic number of steps relative to the number of processes. Each process gathers data from a subset of other processes and forwards the aggregated data up the tree. This reduces the communication load on the root process and balances the workload more evenly across all participating processes, making it more efficient for larger systems.

Basic Linear Gather The Basic Linear Gather algorithm operates by having each participating process send its data directly to the root process. This straightforward approach results in a linear increase in latency as the number of CPUs increases, since each process independently communicates with the root. The estimated latencies, represented by the red line, follow a linear model calculated as the sum of the latencies between each process and the root process. This model is formulated as follows

$$T(P) = \sum_{i=0}^{P-1} L_{i,0} \tag{2}$$

where *T(P)* is the total time for the gather operation with P processes and $L_(i;0)$ represents the latency between the process i and the root process. On the other hand the actual latencies (blue dots) initially align reasonably well with the model, but beyond a certain point, there is a noticeable divergence. As shown in Figure 3, after a certain point the latency starts to be less than the expected one, resulting in an unexpected behaviour for the linear model.

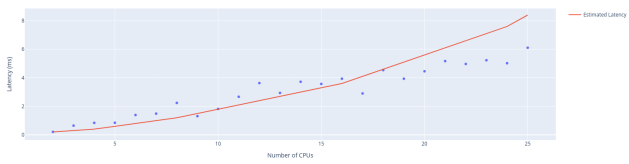


Figure 2: *Comparison between actual execution time (crosses) and estimated (line) execution time of the Gather collective using the Basic Linear algorithm, from 2 to 48 processes*

Binomial Gather In this case the estimated latency, represented by the red line, calculated based on the structure of the binomial tree, can be formulated as

$$T(P) = \sum_{i=0}^{\log_2 P} \sum_{j=0}^{\frac{P}{2^i}-1} L_{i,i+2^i} \tag{3}$$

where *T(P)* is the total time for the gather operation with P processes and $L_(i;i+2^i)$ represents the latency between the process i and $i+2^i$ in the binomial tree structure. Looking at the plot, the actual latencies (blue dots), when compared with the expected values, show a global significant divergence. Starting with a low number of CPUs the actual lateny values can be percieved as goodly aligned with the expected value but, as the number of CPUs increases this difference becomes greater and greater. As in the cases of Broadcast algorithms, this can be due to to several real-world factors not fully captured by the model, such as network contention and varying memory access patterns. The binomial gather algorithm's hierarchical structure helps to mitigate some of these issues, resulting in better scalability and efficiency compared to the basic linear gather. However, the complexities of real-world network and memory behaviors still introduce variability in the actual latency measurements. Figure 4 shows the actual latencies and the expected one.

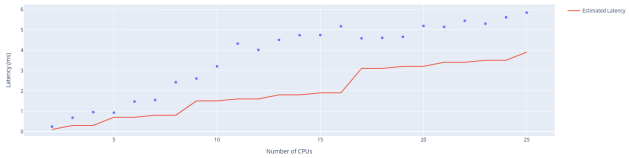


Figure 3: *Comparison between actual execution time (crosses) and estimated (line) execution time of the Gather collective using the Binomial algorithm, from 2 to 48 processes*

3 Conclusion

The analysis of collective operations using the Chain and Binary Tree algorithms for Broadcast and the Basic Linear and Binomial algorithms for Gather highlighted the challenges in accurately modeling performance on THIN nodes. While the naive models captured general trends, actual latencies often deviated due to factors like network contention and memory access patterns. The Chain Broadcast and Basic Linear Gather algorithms showed initial alignment with the models but diverged at higher CPU counts. The Binomial Gather algorithm, though more efficient, also displayed significant discrepancies. These results underscore the need for more sophisticated models that account for real-world complexities to better predict performance in high-performance computing environments.