Benchmarking and modeling OpenMPI collective operations on the ORFEO computing cluster

Andrea Esposito (mat. SM3600005)
University of Trieste - Scientific and Data Intensive Computing (SDIC)

High Performance Computing Final 2023/24

Abstract: This document reports procedures and results for Exercise 1 of the final assignment of the High Performance Computing course. The goal of this assignment's task is to gather data about timing of blocking collective operations implemented within OpenMPI using the well-known OSU benchmarking suite, on the ORFEO computing cluster and analyzing such data, along with the architecture of interest, in order to obtain a performance model for different collective operations which estimates running time for a given number of processes. The Broadcast and Barrier collective operations were analyzed in the context of this task, choosing the Linear (or Flat Tree) and Binary Tree algorithms for the first and the Linear and Recursive Doubling algorithms for the latter. As benchmarks were run with a varying number of processes allocated on CPU cores, CPU latency was also measured using the OSU benchmarking suite, in order to establish the communication overhead between processes run on different cores/nodes. The obtained models satisfyingly resemble the collected data for the different operations and algorithms, providing an explainable approximation in accordance to the architectural characteristics of the nodes on which the benchmarks were performed.

1 Introduction

Collective operations are implemented in OpenMPI in order to provide synchronization patterns and communication strategies among different processes in a distributed memory scenario for parallel programming. According to [1], roughly 80% of the execution time of MPI applications is spent in performing collective operations, whose optimization is therefore crucial for maximizing performance of distributed computation. Due to the variety of algorithms, topologies and methods encompassed by different collectives, optimizing them requires tailored effort for each specific case. Furthermore, the optimal implementation for each case depends on a number of influencing factors, such as the number of communicating processes, the physical topology of the system, the reciprocal distance between communicating nodes/cores, the latency of the communication mediums (NUMA, network or other), the message sizes and so on. Although blocking and non-blocking implementations of all collective operations exist within OpenMPI, only blocking routines were analyzed for the sake of this assignment. The implementation of collective operations is based on point-to-point blocking and non-blocking communication routines, i.e. *send* and *receive* operations which make use of shared buffers to exchange data among processes. The latency of such point-to-point operations was measured and considered as the main influencing factor for obtaining an estimation of the analyzed collectives. Furthermore, the topology and characteristics of the architecture on which the benchmarks were run were also taken into account.

1.1 Software

Benchmarks were performed to obtain baseline timing data for chosen algorithm of the selected collectives 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 time 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 benchmarks were run on Epyc nodes of the ORFEO computing cluster, based on the AMD 7H12 Rome architecture. Two fully reserved nodes were used during each run of the benchmarking suite, both for recording latency data and data about the broadcast/barrier collective operations. Each node has the topology shown in Figure 1.

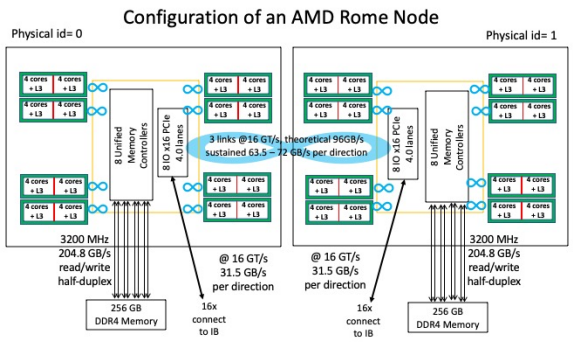

Figure 1: *AMD Epyc 7H12 Rome Node, source: [3]*

As shown, each node bears two sockets, each containing eight Core Complex Dies (CCDs). Each CCD is in turn comprised of two Core Complexes (CCXs), each of which

contains four physical cores sharing 16MB of L3 cache. Within every socket, four NUMA (Non-Uniform Memory Access) regions exist, containing two CCDs (eight cores) each. Cores in the same NUMA region share access to the same memory space. The complex topology of the Epyc architecture determines a differentiated latency among cores situated in regions closer or further from each other. For the sake of this task, memory access was not assessed nor taken into account as an influencing factor, as benchmarks were performed with a fixed and small message size.[1]

2 Experiment

Experimental trials were performed under fixed conditions, using the software and hardware setup described in 1.1 and 1.2. Mapping to *core* was specified when running benchmarks with `mpirun`. Considering all benchmarks were run on two fully reserved nodes, each bearing a total of 128 hardware cores, a grand total of 256 cores were available. All benchmarks were configured with 100 warm-up iterations (which are not considered when computing results) and 1000 iterations per run, whose results are averaged to obtain the final timing results.

2.1 Latency

Latency tests were run specifying pairs of physical cores in order to obtain timings across different regions of the available CPUs. Six critical points were identified, inside each of which the value for latency can reasonably be considered constant. Such critical points represent latencies within the same CCX, CCD, NUMA region, socket, node and cluster. Latency values for such critical points are shown in Table 1.

Core	Latency [μs]
1	0.14
4	0.35
8	0.35
32	0.41
64	0.65
128	1.82

Table 1: *Latency values for communication of the given cores with Core #0*

It can be noted that communication between Core #0 and cores in the same CCD shares the same latency of communication between Core #0 and cores in the same NUMA region. Such critical point is therefore discarded in subsequent sections.

2.2 Broadcast

The broadcast operation was tested with the Chain and Binary Tree algorithms. As explained by [4], with the Chain algorithm each node serves as the root of a tree with only one child: 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* operations to each layer. In this latter case, the root only communicates with the first layer of the built tree (its left and right children), which in turn propagate the message to their children, until the leaves are reached.

Chain Broadcast Collected data about the Chain broadcast algorithm suggests that the algorithm scales linearly as the number of communicating processes increases. A favouring factor can be observed, which lowers the actual time with respect to a possible naive estimation based on a summation of latencies. Allegedly, this is to be attributed to the algorithm implementation being segmentation-based, which allows for a portion of the communication to be parallel, despite the intrinsically sequential nature of the Chain algorithm. The obtained model is the following.

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

where $L_{i,i+1}$ is the CPU latency in communication between core i and $i+1$ and $\gamma = 0.4$ is the aforementioned overlap factor. The model is compared with collected data in Figure 2.

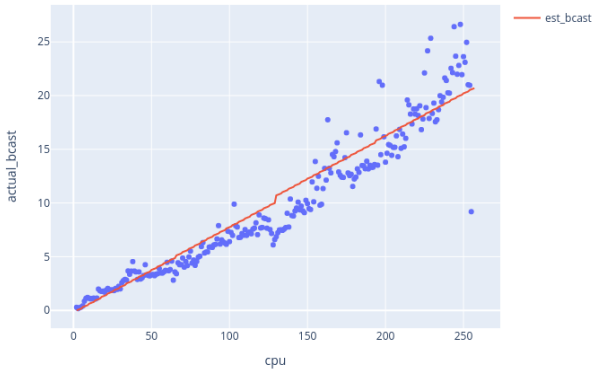

Figure 2: *Comparison of actual execution time (scattered) and estimated (line) execution time of the Broadcast collective using the Chain algorithm, fixed message size of* 2 `MPI_CHAR` *and from* 2 *to* 256 *processes*

Binary Tree Broadcast Collected data about the Binary Tree broadcast algorithm suggest a logarithmic increase, which is to be expected given the previously described implementation details. The tested model describe such collective estimates the execution time between P processes by creating a balanced binary tree with P nodes (i.e. all processes involved, from 0 to $P-1$). In order to then generate an actual estimate it traverses the tree keeping track of the maximum CPU latency[2] for the exchange of the message between each current node and its two children, to which the algorithm is applied recursively, updating the total. Only the maximum latency among the children is considered as the tree is actually traversed in parallel and the propagation of

[1] All benchmarks were instructed to exchange messages with size of 2 `MPI_CHAR` = 2 bytes, in order to render the communication time negligible. This causes all exchanged messages to fit in all caches, essentially avoiding memory access.

[2] the CPU latency is hereby accounted for twice, in order to consider the bidirectional exchange

the message is split between two processes. CPU latency between two communicating nodes in the tree is determined by considering their affinity, as explained in 2.1. The model is compared with collected data in Figure 3.

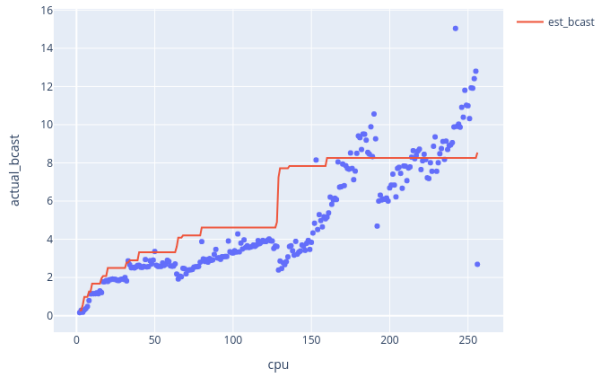

Figure 3: *Comparison of actual execution time (scattered) and estimated (line) execution time of the Broadcast collective using the Binary Tree algorithm, fixed message size of* 2 `MPI_CHAR` *and from* 2 *to* 256 *processes*

2.3 Barrier

The barrier operation was tested with the Linear and Double Ring algorithms. Such algorithms are explained by [5] and further details about their implementation were obtained by analyzing the source code of OpenMPI at [6].

Linear Collected data about the Linear barrier algorithm suggests that the algorithm scales linearly as the number of communicating processes increases. The actual implementation is based on a root node waiting for messages from all other nodes with non-blocking *read* operations and then responding afterwards with *send* calls performed sequentially; all other nodes (in parallel) perform a blocking *send* towards the root and wait for a response with a blocking *receive*. The following is a derived model for resembling data about the linear barrier operation.

$$T(P) = 2 \cdot L_P + \gamma \cdot \sum_{0}^{L-1} L_P$$

where L_P is the CPU latency associated with process P and γ is an overlapping factor which accounts for the parallelized *send* and *receive* operations. The model is compared with collected data in Figure 4.

Double Ring Collected data about the Double Ring barrier algorithm shows a baseline similarity to the Linear algorithm, suggesting that the algorithm scales linearly as the number of communicating processes increases. It is expected that an approximation of this algorithm can be obtained in a very similar fashion to the Linear algorithm for the barrier collective, as the implementation bears the same linear complexity. The main difference resides in the communicating nodes, which in this case are (mostly) neighbouring nodes, as the message is propagated to the right with carryover. A feasible

Figure 4: *Comparison of actual execution time (scattered) and estimated (line) execution time of the Barrier collective using the Linear algorithm, fixed message size of* 2 `MPI_CHAR` *and from* 2 *to* 256 *processes*

model was obtained by summing CPU latency values of the communicating nodes, determined (as before) as explained in 2.1. The resulting model visibly overestimates the communication time within the same node ($P \leq 128$), probably due to the possibility of nodes of *exiting* the ring under certain conditions, which was not taken into account. Message segmentation is not considered as an influencing factor, as the barrier collective exchanges zero-sized messages. The model is compared with collected data in Figure 5.

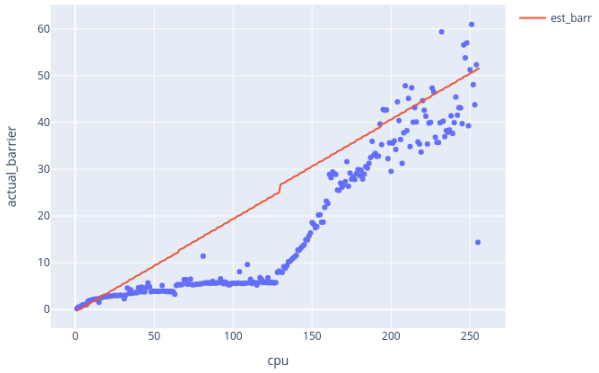

Figure 5: *Comparison of actual execution time (scattered) and estimated (line) execution time of the Barrier collective using the Double Ring algorithm, fixed message size of* 2 `MPI_CHAR` *and from* 2 *to* 256 *processes*

3 Conclusion

In conclusion, although the analyzed collectives and algorithms bear a seemingly simple implementation with a well-known complexity, available data and knowledge did not always allow for satisfying naive models to be derived. Certain implementation details and optimization patterns, among with more in-depth caveats of the complex architecture on which the tests were run, along with unpredictable factors (such as congestion and computational/transfer load

at the time data was gathered) make for a greater challenge when trying to model the performance of intrinsically hybrid (serial and parallel) procedures. Nonetheless, the generated naive models always highlight the leading scaling trend in the underlying data, allowing to visualize the role of CPU latency and its variability according to core affinity, when running collective MPI operations.

References

[1] R. Rabenseifner, "Automatic profiling of mpi applications with hardware performance counters," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, J. Dongarra, E. Luque, and T. Margalef, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 35–42.

[2] O. Benchmarks, *Http://mvapich.cse.ohio-state.edu/benchmarks/*, 2023.

[3] A. R. Processors, *Https://www.nas.nasa.gov/hecc/support/kb/amd-rome-processors*$_6$58.*html*, 2022.

[4] E. Nuriyev, J.-A. Rico-Gallego, and A. Lastovetsky, "Model-based selection of optimal mpi broadcast algorithms for multi-core clusters," *Journal of Parallel and Distributed Computing*, vol. 165, pp. 1–16, 2022.

[5] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. Fagg, E. Gabriel, and J. Dongarra, "Performance analysis of mpi collective operations," in *19th IEEE International Parallel and Distributed Processing Symposium*, 2005, 8 pp.-.

[6] OpenMPI, *Https://github.com/open-mpi/ompi/blob/main/ompi/mca/coll/base/coll*$_b$*ase*$_b$*arrier.c*, 2023.