UNIVERSITÀ DEGLI STUDI DI TRIESTE

EXERCISE 1: COMPARE DIFFERENT OPENMPI ALGORITHMS FOR COLLECTIVE OPERATIONS

STUDENT

FU ZIYANG
SM3800011

MSc in Data Science & Artificial Intelligence
Foundations of Artificial Intelligence and Machine Learning

TRIESTE, 21ST FEBRAURY 2024

Contents

Contents 1

1 Introduction 2

2 Architecture 3

2.1 EPYC node 3

2.2 Performance model 4

3 Broadcast 6

3.0.1 Linear algorithm 6

3.0.2 Binary Tree algorithm 8

4 Barrier 9

4.0.1 Performance model 9

5 Conclusions 12

1
Introduction

The purpose of this exercise is to evaluate the performance of different collective communication algorithms implemented in the openMPI library, particularly focusing on the **broadcast** operation and the additional collective operation **barrier**.
The evaluation is conducted using the OSU benchmark suite, a well-established tool for assessing MPI performance.

In this exercise, we aim to understand the impact of varying the number of processes, keeping the message size fixed to 2, on the latency of default openMPI implementations for these collective operations. Additionally, we investigate the alternative algorithms **linear** and **binary tree** provided for the broadcast operation.

The ultimate purpose of this assignment is to understand/infer the performance model behind the selected algorithms, taking into consideration the architecture on which they are being executed.

The optimal implementation of a collective for a given system depends on many factors, including for example, physical topology of the system, the number of processes executed, message sizes, and the location of the root node.

In the next section the first of the previously listed performance influencing agent will be examined.

2 Architecture

This section of the report is designed to provide an overview over the topology on which the performance metrics have been collected: the **EPYC** partition, consisting of the 8 nodes equipped with two **AMD EPYC 7H12 cpus**, with the **Rome** architecture.

2.1 EPYC node

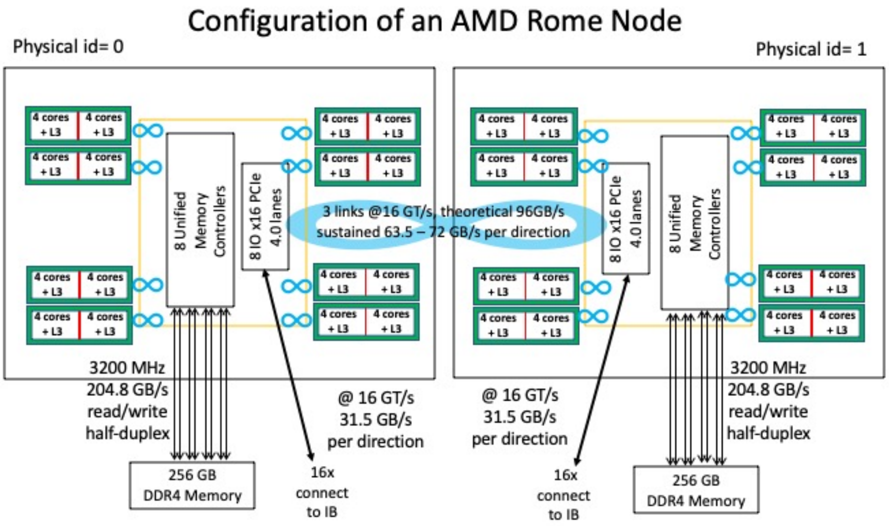


Figure 2.1: *simplified configuration of an EPYC Rome node with two sockets* Nasa, 2022

The processor presents the following hierarchy:

- **CPU**: a CPU core has private L1I, L1D, and L2 caches;

- **CCX**: a core complex includes four cores and a common L3 cache of 16 MB. Different CCXs do not share L3;
- **CCD**: a core complex die includes two CCXs and an Infinity Link to the I/O die (IOD). The CCDs connect to memory, I/O, and each other through the IOD;
- **Socket**: a socket includes eight CCDs (total of 64 cores) and a link to the network interface controller (NIC);
- **Node**: a node includes two sockets and a network interface controller (NIC)

In summary, there are **64 cores per socket** and **128 cores per node**.

2.2 Performance model

To accomplish the ultimate goal of the assignment (a simple performance model), it is essential to estimate the latency of point-to-point communication routines, since collectives operations are built on top of them.

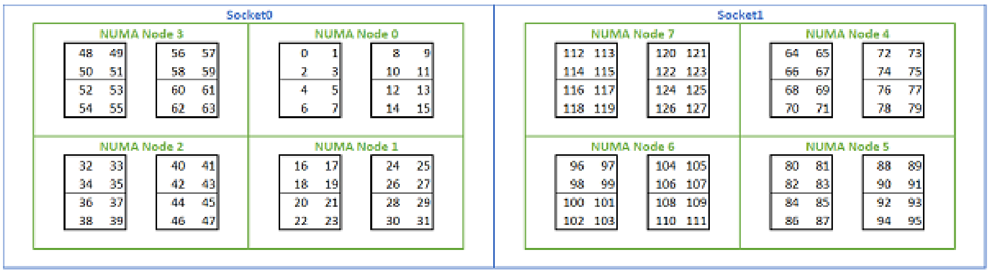
"Point-to-point" refers to the direct exchange of messages between individual processes. It involves the transmission of data between exactly two processes: a sender and a receiver.

Specifically, we need to explore the latency among cores located in different regions of the processor. By taking two EPYC nodes, and measuring this metric with `mpirun -np 2 -cpu-list 0,int`, the following latencies have been detected:

Table 2.1: *Table showing the latencies between different processor regions*

Processor region	Latency
Same CCX	0.14 μs
Same CCD	0.31 μs
Same socket	0.44 μs
Same node	0.66 μs
Different nodes	1.82 μs

Couple more specifications: within the same socket the CCDs connect to each other through the IOD and internal connectivity in ORFEO is handled by a 100 Gbit/s Infiniband link for Remote Direct Memory Access (RDMA) between the compute nodes.



ORFEO, 2024

Figure 2.2: *simplified configuration of an EPYC Rome node, highlighting the NUMA grouping*

From the above figure 2.2 we can see that two contiguous CCDs share the same NUMA, so we can reasonably expect that the latency between the cores 0-15 being lower than the one between 0-30.

Additionally the latencies were measured without writing any *rankfile*, so the reported latency between cores located in different nodes was extrapolated from the official ORFEO documentation (ORFEO, 2024).

[1]

[1] For the sake of this task, since we have fixed the message size to 2 bytes, differences in memory access was not assessed nor taken into account as an influencing factor.

3

Broadcast

Broadcast is **collective operation**, that are implemented in OpenMPI with the purpose to provide synchronization patterns and communication strategies among different processes in a distributed memory scenario for parallel programming.

The rationale behind the extensive research in this field is due to the fact that roughly 80% of the execution time of MPI applications is spent in performing collective operations (Rabenseifner, 1999), thus optimization is key in maximizing the performance of distributed computation.

3.0.1 Linear algorithm

In the linear algorithm a root process sends an individual message to all the processes in the communicator.

Developing performance models is challenging due to the myriad factors that need to be taken into consideration. Thanks to the simplicity of the linear broadcast algorithm and to the **mapping policy** `map-by core`, that allows us to map the processes to the available hardware cores, the following **naive model** has been developed:

$$T_{linear}(P) = \textstyle\sum_{i=1}^{P-1}(T_{p2p} \cdot P + \frac{L(p)}{2}) \cdot 2$$

where:

- T_{linear} is the bidirectional communication time between two nodes;
- P is the number of processes;

- L is the estimated latency between the core 0 and the core $P-1$; from now on L follows the estimations listed in the table 2.1;
- T_{p2p} is the point-to-point communication time, estimated as

$$\frac{T_{linear}(256)-(2 \cdot L(256)))}{255}.$$

This model simply computes the total communication time as the sum of P point to point bilateral communication times.

The estimated results are shown in the figure 3.1

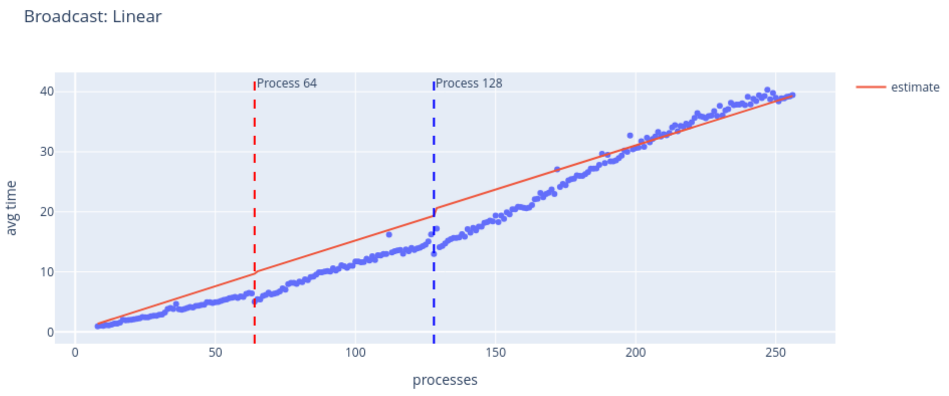


Figure 3.1: *Naive model estimate of the linear broadcast algorithm*

We can see that the model tends to underestimate the time needed to complete $P-1$ communications and we might trace this underestimation due to the perfect parallel non-blocking send assumption.

Additionally the model does not capture the changes in the slope that can be noted from the figure 3.1: the measured datapoints can be segmented into 3 parts, each presenting different slopes:

- the region from **0 to 64**: this identifies processes located in the same socket of the root, thus presenting the least steep slope;
- the region from **65 to 128**: this identifies the processes mapped to the other socket of the node, with respect to the location of the root;
- the region from **129 to 256**: this identifies the processes mapped to a different node than the root process, indeed presenting the steepest slope.

3.0.2 Binary Tree algorithm

In the binary tree algorithm the messages sent from the root traverse the tree starting from the root itself and going towards the leaf nodes through intermediate nodes.

To model the performances of this algorithm we have made use of an already documented model from Pjesivac-Grbovic et al., 2005: **the LogP** model.

$$T_{LogP}(P) = \text{ceil}(log_2(P+1)-1)*(L*o(P))$$

We took a naive in estimating the o parameter, which is the parallelization factor, representing the increase in the cost of $P-1$ overlapping non-blocking send operations by the root process, by looking at the datapoints pattern: we intended to estimate this parameter as the slope of the datapoints pattern, but a change in the latter starting from 150 made us modelling two separates o: one from 0 up to 150 processes and one from 150 up to 256 processes.

Its estimates are shown below:

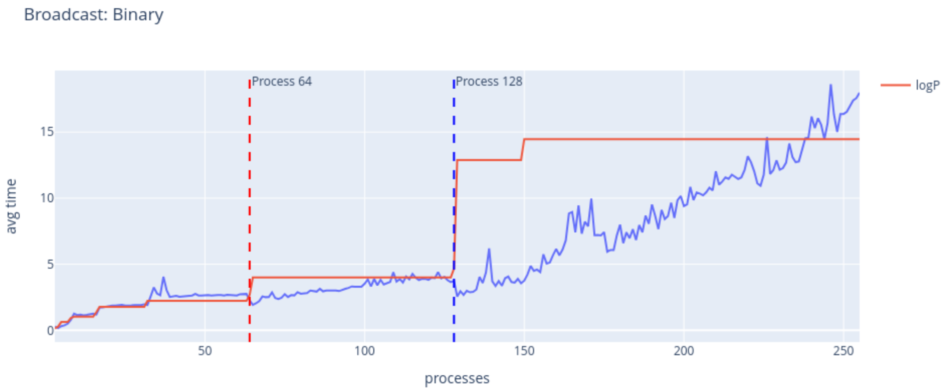


Figure 3.2: *Naive model estimate of the linear broadcast algorithm*

The model mapped quite well up to the 128th processs, after which the results start to show important discrepanciens.

4

Barrier

Barrier is a collective operation used to synchronize a group of processes. It is used to ensure that all processes in a group reach a designated point in their execution before any of them proceed further. It's called a "barrier" because it acts like one, forcing processes to wait until all processes have reached the same point before any of them are allowed to continue.

Specifically the **linear** algorithm of this collective operation was taken into examination: all the nodes report to a pre-selected root and once every node has done the reporting, the root sends a releasing message to all the other nodes involved in the communicator. A node can leave the barrier only after the reception of the pre-said message.

4.0.1 Performance model

To infer a performance model for this algorithm, we will refer again to Pjesivac-Grbovic et al., 2005, that provided us already tested models to estimate the performance of this collective operation.

In particular the first selected model has been the **Hockney**:

$$T_{linear}(P) = (P - 1) \cdot \alpha$$

where P is the number of processes and α represents the latency per message.

Experiments have proven that Hockney heavily overestimates the needed time, as the below figure 4.1 shows:

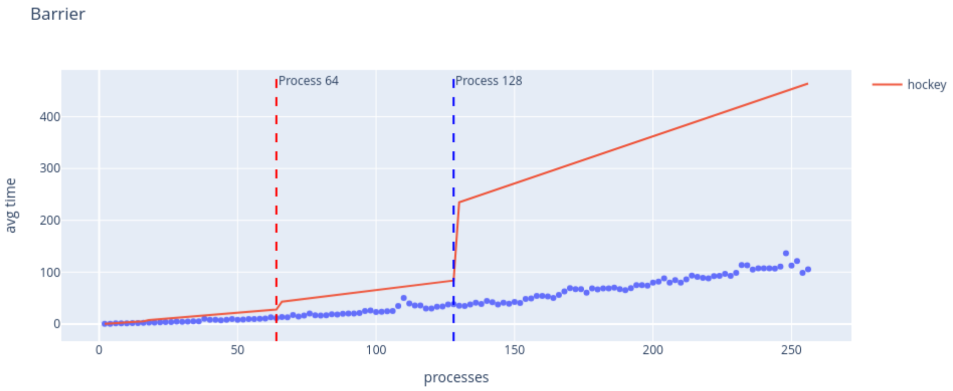


Figure 4.1: *Hockney model estimate of the communication time*

The measured data do not present the sharp jum in the regions where the root and not-root process are respectively placed in different sockets (after the 64th process) and in different nodes (after the 128th process).
We assumed this behaviour to be present with the collected data explained in section two (2).

Given the poor results of the Hockney model we decided to experiment with another one: the **LogP**.
It suites our case since it assumes that only constant and small-sized messages are communicated between processes and models the communication time as $L + 2o$, where L is the latency due to cores distance, and o is the parallelization factor, representing the increase in the cost of $P - 1$ overlapping non-blocking receive operations by the root process.

The *LogP* models the linear barrier as follows:

$$T_{linear}(P) = (P - 2) + 2 \cdot (L + 2 \cdot o)$$

The estimated performances are shown in the below figure 4.2:

o has been estimated as the slope of the line traversing the datapoints.

The results is slightly better, since now processor latencies are added to the number of processes, however the model still implies, as the Hockney

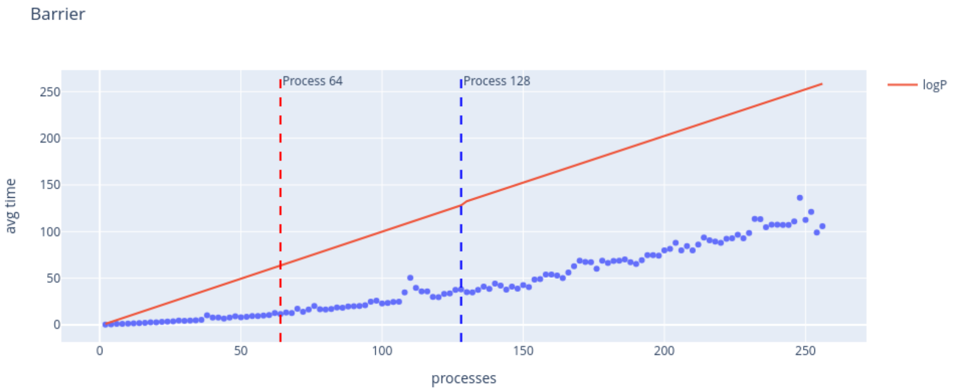


Figure 4.2: *LogP model estimate of the communication time*

model does, a perfect serial execution of the reporting stage of the non-root processes, whereas the measured datapoints suggest us otherwise. This led to overestimate the time needed by the algorithm to complete P processes.

5

Conclusions

In our examination of broadcast and barrier operations on the ORFEO cluster, we discovered the nuanced interplay between communication patterns, hardware configurations, and algorithm selections within the OpenMPI framework. Through experimentation and modeling efforts, we uncovered intricate latency dynamics, and tried to shed light on discrepancies between actual performance and simplified performance models.

Our analysis underscored the significance of accounting for various factors, such as cluster topology, fluctuations in point-to-point communication latencies, and the impacts of scaling, in constructing precise performance models.

The assignment findings emphasize the complexity inherent in optimizing communication performance in parallel computing environments, necessitating a comprehensive understanding of the underlying system dynamics.

Bibliography

Nasa (2022). *Website Title*. Accessed on Month Day, Year. URL: `https://www.nas.nasa.gov/hecc/support/kb/amd-rome-processors_658.html`.

ORFEO (2024). *Website Title*. Accessed on Month Day, Year. URL: `https://orfeo-doc.areasciencepark.it/examples/MPI-communication/#more-on-core-mapping`.

Pjesivac-Grbovic, J. et al. (2005). "Performance analysis of MPI collective operations". In: *19th IEEE International Parallel and Distributed Processing Symposium*, 8 pp.-. DOI: `10.1109/IPDPS.2005.335`.

Rabenseifner, Rolf (1999). "Automatic profiling of MPI applications with hardware performance counters". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 6th European PVM/MPI Users' Group Meeting Barcelona, Spain, September 26–29, 1999 Proceedings 6*. Springer, pp. 35–42.