Foundations of HPC: Assignment 1

Alessandro Pierro
MSc in Data Science and Scientific Computing

Abstract

s The following is a report for the course "FoundationS of High Performance Computing" delivered by University of Trieste in 2021/2022. The instructions and more details on the assignment can be found on the `Foundations_of_HPC_2021` repository on GitHub.

1 MPI Programming

In this first section, the two MPI programming assignment will be analysed, both in terms of theoretical modeling and experimental performances. The related code is developed in C/C++, compiled with OpenMPI 4.1.1 + GNU 9.3.0 and executed on *Thin* nodes of the ORFEO cluster, using the Infiniband network.

1.1 Bidirectional Ring

Summary - A bidirectional ring is developed using a one-dimensional periodic virtual topology of P processors. On the first iteration, each processor sends two messages, one for each neighbour, having both TAG equals to the processor rank and a message consisting of a MPI_INT equals to `rank` (for the left neighbour) and `-rank` (for the right neighbour). On the successive iterations, each processor propagates the received messages in the two directions, respectively adding or subtracting its rank.

```
int TAG = rank;
int left_send = right_receive + rank;
int right_send = left_receive - rank;
```

After P iterations, each initial message is received back by its original sender processor and the program exits.

The `ring.cpp` program is executed on up to 24 processors of the same *Thin* node and the total wall time (averaged over 10^4 executions) is measured using `MPI_Wtime` utility. As shown in Figure 1, the walltime increases linearly with the number P of processors in the ring and, therefore, network performances can be modeled as follows:

$$walltime = \alpha + \beta \cdot P \tag{1}$$

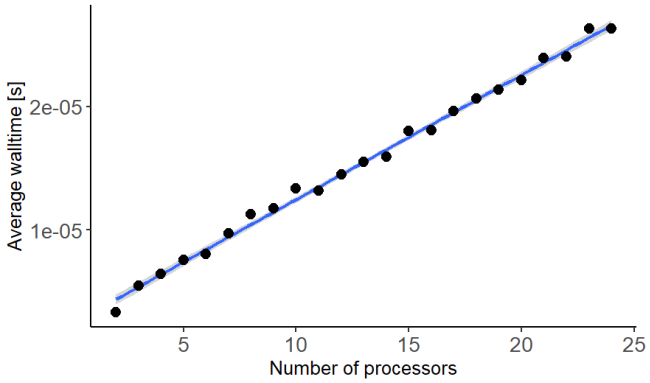

Figure 1: Average wall time [s] for the `ring.cpp` program as a function of the number of processors (up to 24) in the ring

Performing regression analysis in R, the following estimates of the model parameters are computed:

$$\hat{\alpha} = 2.35 \cdot 10^{-6} \pm 0.22 \cdot 10^{-6}$$
$$\hat{\beta} = 10.1 \cdot 10^{-7} \pm 0.15 \cdot 10^{-7}$$

The obtained model is also sounded from the theoretical point of view since the number of exchanged messages by each processor increases linearly with the number of processors in the ring; in fact, each processor sends exactly $2P$ messages per execution.

Testing the program on up to 48 processors (results shown in Figure 2), the linear model in Equation 1 starts to underestimate the average wall time and, in particular, a remarkable jump is observed for $P = 25$. This sudden increase in the average wall time can be explained keeping in mind that a Thin node on ORFEO has 24 physical cores and, therefore, for $P > 24$ the ring is extended over multiple nodes causing the additional overhead involved in communicating between different nodes.

In conclusion, the experimental results suggest that the average wall time increases super-linearly with the number of processors in the ring due to the increased complexity in the network structures each time a new node is added to the system. Supporting results can also be found in [SSM13] that, though using TCP/IP interconnect, modeled the performance of a ring network topology through simulation finding that

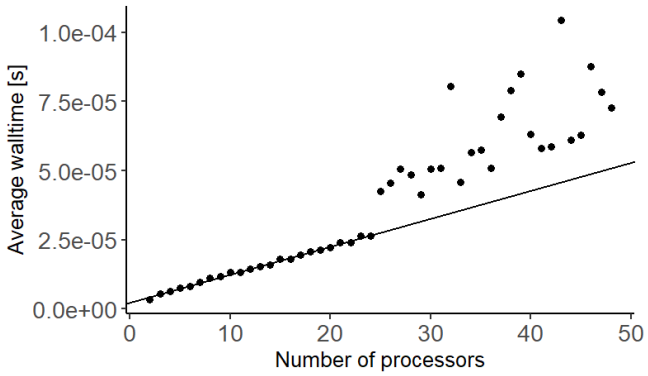

Figure 2: Average wall time [s] for the `ring.cpp` program as a function of the number of processors in the ring: for $P > 24$ the average wall time increases faster than predicted.

point-to-point communication performances deteriorate increasing the number of nodes in the network. So, even if the workload for each processors increases linearly, the increase in average wall time will be super-linear due to the increase in the communication overhead.

Overflow error - In an initial implementation of `ring.cpp`, in order to respect the specification asking that each message should have a `TAG` proportional to the sender rank, the messages were propagated with a TAG equals to the product of the received one with the rank of the new processor (instead of using only the new sender `rank`). However, testing the program on more than 12 processors resulted in a fatal error. Investigating the code, it turned out that, with 13 processors, two of the messages of the final iteration would contain a `TAG` equals to 13! that, being greater than $2^{31} - 1$, is interpreted as a negative number due to *overflow*. Since `TAG` cannot contain a negative `MPI_INT`, the program throws an exception and exits.

1.2 Parallel Sum of 3D Tensors

Summary - A parallel program for the sum of two 3D tensors of `MPI_DOUBLE` is developed, using only the MPI collectives operations discussed during the lectures. The program accepts in input the dimensions of the tensor (n_x, n_y, n_z) and the number of processors $(x_{dim}, y_{dim}, z_{dim})$ for each dimension of the virtual topology. Then it generates on `processor 0` two random tensor of the specified dimension, scatters them on all the available processors, performs the parallel sum and gathers the resulting tensor back on `processor 0`. The performance of the program are evaluated using 24 cores on a Thin node and compared across all the admissible virtual topologies for the following dimensions of the tensor:

- 2400 x 100 x 100
- 1200 x 200 x 100
- 800 x 300 x 100

A tensor $T \in \mathcal{R}^{n_x} \times \mathcal{R}^{n_y} \times \mathcal{R}^{n_z}$ is represented in `sum3Dmatrix.c` as a contiguous array A of dimension $n_x \cdot n_y \cdot n_z$. Following the convention seen during the lectures, the conversion between the formal structure and the computational representation is performed as:

$$T(i,j,k) = A\big[(k \cdot n_x \cdot n_y) + (j \cdot n_x) + i\big] \qquad (2)$$

In order to distribute such a tensor between the different processors in the network, a `MPI_Scatter` operation is performed. However, as pointed out during the class discussion, `MPI_Scatter` does not depends on the topology of the network and bases the distribution only on the rank of each processors. That is, independently from the chosen virtual topology, the `MPI_Scatter` operation divides the array A equally across the different nodes.

To better explain why this is imprecise, let's consider a tensor T as pictured in Figure 3 and a topology composed of two processors. Accordingly to Equation 2, this tensor can be represented as an array $A = \{0, ..., 7\}$.

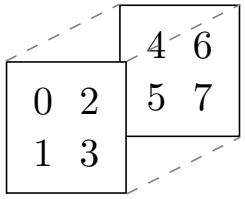

Figure 3: Representation of a 3D tensor where the element (i, j, k) is the coefficient on the i-th row, j-th column and k-th matrix.

Let's now consider two differnt domain decompositions: a 1D domain decomposition along the x axis and one along the y axis. The corresponding virtual topologies would have the following dimensions:

$$dim_{\text{1D along x}} = (2, 1, 1)$$
$$dim_{\text{1D along y}} = (1, 2, 1)$$

According to the theory of domain decomposition, the correct data distribution for these two examples should be as presented in Figure 4. As said earlier however, due to the inner working of `MPI_Scatter`, the obtained decomposition would be the same, independently of the specified virtual topology, assigning $\{0, ..., 3\}$ (first half of the array) to the first processor and $\{4, ..., 7\}$ (second half of the array) to the second processor.

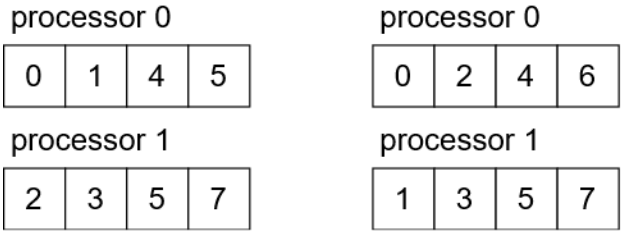

Figure 4: Correct data distributions for the tensor in Figure 3: for the 1D domain decomposition along x on the left and along y on the right.

It seems obvious that the problem highlighted so far makes the exercise trivial. This is confirmed by the results of the timing of `sum3Dmatrix.c` on a tensor of dimension $(2400, 100, 100)$ using different virtual topologies. As summarised in Table 1, the average wall time shows not significant variation across the different virtual topologies, consistently with the explained functioning of `MPI_Scatter`.

n_x	n_y	n_z	Time $[s]$
24	1	1	$3.568 \cdot 10^{-3}$
1	24	1	$3.526 \cdot 10^{-3}$
1	1	24	$3.502 \cdot 10^{-3}$
12	2	1	$3.501 \cdot 10^{-3}$
12	1	2	$3.461 \cdot 10^{-3}$
1	12	2	$3.430 \cdot 10^{-3}$
2	12	1	$3.550 \cdot 10^{-3}$
1	2	12	$3.535 \cdot 10^{-3}$
2	1	12	$3.486 \cdot 10^{-3}$
6	4	1	$3.510 \cdot 10^{-3}$
6	1	4	$3.570 \cdot 10^{-3}$
1	6	4	$3.560 \cdot 10^{-3}$
4	6	1	$3.557 \cdot 10^{-3}$
1	4	6	$3.528 \cdot 10^{-3}$
4	1	6	$3.544 \cdot 10^{-3}$
3	1	8	$3.530 \cdot 10^{-3}$
3	8	1	$3.496 \cdot 10^{-3}$
1	3	8	$3.506 \cdot 10^{-3}$
8	3	1	$3.516 \cdot 10^{-3}$
1	8	3	$3.499 \cdot 10^{-3}$
8	1	3	$3.529 \cdot 10^{-3}$
6	2	2	$3.464 \cdot 10^{-3}$
2	6	2	$3.505 \cdot 10^{-3}$
2	2	6	$3.475 \cdot 10^{-3}$
3	2	4	$3.574 \cdot 10^{-3}$
3	4	2	$3.576 \cdot 10^{-3}$
2	3	4	$3.529 \cdot 10^{-3}$
4	3	2	$3.534 \cdot 10^{-3}$
2	4	3	$3.516 \cdot 10^{-3}$
4	2	3	$3.558 \cdot 10^{-3}$

Table 1: Average execution time of `sum3Dmatrix.c` for a tensor of dimension $(2400, 100, 100)$ for all possible virtual topologies with 24 processors (on a Thin node).

2 Intel PingPong MPI Benchmark

Summary - In this section, the network performance of point-to-point communication will be analysed on the ORFEO cluster, using the `PingPong` test from Intel MPI Benchmark. The following factors have been taken into account: the kind of internal interconnect (Infiniband or TCP/IP), the mapping of the communicating processors (by core, socket or node), the use of Intel Hyper-Threading Technology and the used compiler (OpenMPI 4.0.3 or Intel(R) MPI Library 2019.9).

As covered during the lecture, a basic model for the performances of a network is the so-called **latency/bandwidth model**:

$$T_{comm} = \lambda + \frac{size}{b_{network}} \tag{3}$$

In Equation 3, the time taken for a point-to-point communication over a network can be estimated using the network latency λ, the size of the message $size$ in MB and the bandwidth of the network $b_{network}$ in MB/s.

Using Intel MPI PingPong Benchmarck, the network parameters λ and $b_{network}$ are estimated for the following combinations of node type, interconnect and compiler:

- Thin nodes + Gigabit Ethernet + OpenMPI 4.1.1
- Thin nodes + InfiniBand + OpenMPI 4.1.1
- Thin nodes + InfiniBand + Intel MPI Version 2019.9
- GPU nodes + InfiniBand + OpenMPI 4.1.1

Mapping	Latency s	Bandwidth MB/s
by core	$6.04 \cdot 10^{-6}$	4428
by socket	$9.61 \cdot 10^{-6}$	2123
by node	$23.15 \cdot 10^{-6}$	1890

Table 2: Results of least squares regression for the network performance model on Thin nodes using OpenMPI Version 4.1 compiler and Gigabit Ethernet interconnect.

Mapping	Latency s	Bandwidth MB/s
by core	$0.95 \cdot 10^{-6}$	6445
by socket	$1.92 \cdot 10^{-6}$	5689
by node	$3.38 \cdot 10^{-6}$	12160

Table 3: Results of least squares regression for the network performance model on Thin nodes using OpenMPI Version 4.1 compiler and InfiniBand interconnect.

Mapping	Latency s	Bandwidth MB/s
by core	$0.57 \cdot 10^{-6}$	4207
by socket	$0.67 \cdot 10^{-6}$	5490
by node	$2.34 \cdot 10^{-6}$	12239

Table 4: Results of least squares regression for the network performance model on Thin nodes using Intel MPI Version 2019.9 compiler and InfiniBand interconnect.

Mapping	Latency s	Bandwidth MB/s
by core	$0.40 \cdot 10^{-6}$	6089
by socket	$0.88 \cdot 10^{-6}$	4852
by node	$2.15 \cdot 10^{-6}$	12160

Table 5: Results of least squares regression for the network performance model on GPU nodes using OpenMPI Version 4.1 compiler and InfiniBand interconnect.

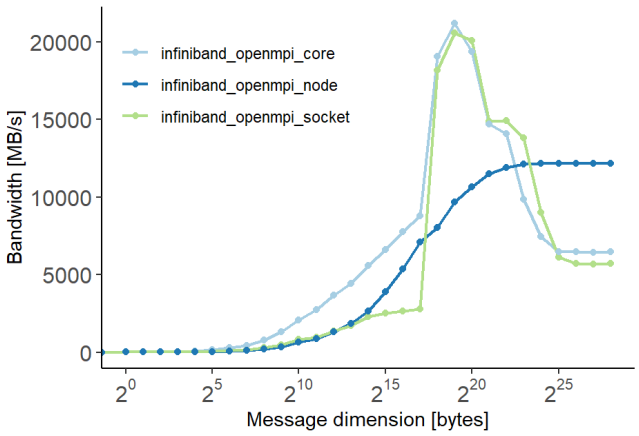

Figure 5: **Impact of processors mapping** - Comparison of network bandwidth $[MB/s]$ on ORFEO using InfiniBand, the Open-MPI compiler and mapping the two processors by node, socket or core.

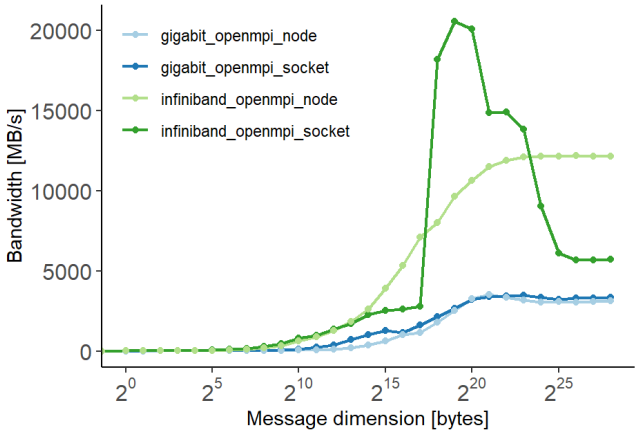

Figure 6: **Impact of interconnect** - Comparison of network bandwidth $[MB/s]$ on ORFEO using InfiniBand or Gigabit Ethernet and mapping the two processors by socket or by core.

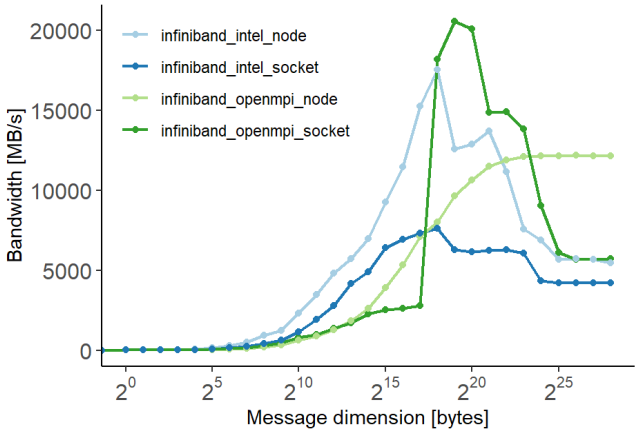

Figure 7: **Impact of compiler** - Comparison of network bandwidth $[MB/s]$ on ORFEO using InfiniBand or Gigabit Ethernet and mapping the two processors by socket or by core.

3 3D Jacobi Solver

Summary - The Jacobi solver is a stencil-based iterative method for solving numerically differential equations. In this section, its performance characteristics are analysed for solving the following diffusion equation on a 3D domain.

$$\frac{\partial \Phi}{\partial t} = \Delta \Phi$$

The code `Jacobi_MPI_vectormode.F` is a parallel implementation of the method and is provided by Georg Hager and Gerhard Wellein as part of their book on HPC. [HW10] The program is compiled with OpenMPI Version 4.1 and executed on multiple Thin and GPU nodes, with GPU nodes having HyperThreading enabled.

As discussed in [HW10], a performance model for the 3D Jacobi solver can be described assuming a network communication approximated with a latency/bandwidth model. Said $\mathbf{n} = (n_x, n_y, n_z)$ the vector containing the dimensions of the virtual topology and L the lateral dimension of the cubic subdomains used by the solver, the performance can be written as:

$$P(K, \mathbf{n}) = \frac{n_x n_y n_z L^3}{T_S(L) + T_C(L, \mathbf{n})} \tag{4}$$

where:

$$T_C(L, \mathbf{n}) = \frac{c(L, \mathbf{n})}{b_{network}} + k\lambda \tag{5}$$

Here, λ and $b_{network}$ are the network characteristics evaluated in the previous chapter while $c(L, \mathbf{n})$ is the maximum bidirectional data volume transferred over a node's network link. Based on the Cartesian domain decomposition, this can be written as:

$$c(L, \mathbf{n}) = 2k \cdot 8 \cdot L^2 \tag{6}$$

where k is the number of dimensions in the virtual topology that have more than 1 processor.

Based on the described model, experimental evaluation of various combinations of settings is conduced on ORFEO. and the results are summarized in the tables in the following page. In the last column of each table, P_{pred} is the predicted performance based on Equation 4 and the results from the previous Section.

References

[HW10] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. 2010.

[SSM13] Harmeet Singh, Sukhjeet Singh, and R. Malhotra. "Modeling, Evaluation and Analysis of Ring Topology for Computer Applications Using Simulation". In: (2013).

N	(n_x, n_y, n_z)	$2k$	$c(L,n)$	P	P_{pred}
1	$(1,1,1)$	0	0.00	85.70	85.70
4	$(2,2,1)$	4	92.16	447.46	442.96
8	$(2,2,2)$	6	138.24	891.49	851.03
12	$(3,2,2)$	6	138.24	1319.71	1245.44

Table 6: Performance results of Jacobi solver on a single Thin node with `--map-by core` using InfiniBand interconnect.

N	(n_x, n_y, n_z)	$2k$	$c(L,n)$	P	P_{pred}
1	$(1,1,1)$	0	0.00	85.70	85.70
4	$(2,2,1)$	4	92.16	451.76	450.14
8	$(2,2,2)$	6	138.24	892.25	885.03
12	$(3,2,2)$	6	138.24	1335.73	1335.35

Table 7: Performance results of Jacobi solver on a single Thin node with `--map-by socket` using InfiniBand interconnect.

N	(n_x, n_y, n_z)	$2k$	$c(L,n)$	P	P_{pred}
1	$(1,1,1)$	0	0.00	78.29	78.29
4	$(2,2,1)$	4	92.16	309.56	307.40
8	$(2,2,2)$	6	138.24	583.45	581.51
12	$(3,2,2)$	6	138.24	846.25	846.55

Table 8: Performance results of Jacobi solver on a single GPU node with `--map-by core` using InfiniBand interconnect.

N	(n_x, n_y, n_z)	$2k$	$c(L,n)$	P	P_{pred}
1	$(1,1,1)$	0	0.00	78.29	78.29
4	$(2,2,1)$	4	92.16	310.52	308.38
8	$(2,2,2)$	6	138.24	611.05	600.04
12	$(3,2,2)$	6	138.24	896.64	891.14

Table 9: Performance results of Jacobi solver on a single GPU node with `--map-by socket` using InfiniBand interconnect.

N	(n_x, n_y, n_z)	$2k$	$c(L,n)$	P	P_{pred}
1	$(1,1,1)$	0	0.00	113.39	113.39
12	$(3,2,2)$	6	138.24	1326.06	1301.41
24	$(4,3,2)$	6	138.24	2653.43	2641.91
48	$(4,4,3)$	6	138.24	5094.46	5011.00

Table 10: Performance results of Jacobi solver on two Thin nodes with `--map-by node` using InfiniBand interconnect.

N	(n_x, n_y, n_z)	$2k$	$c(L,n)$	P	P_{pred}
1	$(1,1,1)$	0	0.00	78.29	78.29
12	$(3,2,2)$	6	138.24	938.07	895.31
24	$(4,3,2)$	6	138.24	1877.15	1685.41
48	$(4,4,3)$	6	138.24	2505.45	2485.05

Table 11: Performance results of Jacobi solver on two GPU nodes with `--map-by node` using InfiniBand interconnect.