Report for Assignment 1 of Foundations of High Performance Computing 2021/2022

Marco Celoria[1,*]
[1] *Master in High Performance Computing SISSA/ICTP*
(Dated: December 30, 2021)

I. SECTION 1

A. Ring

In the first part of Section 1 of the assignment we have to implement a MPI program using N processors on a ring. We actually wrote the program in C, **section1/ring.c**. To compile the code on ORFEO, navigate to the directory and enter:

⩾ module load openmpi-4.1.1+gnu-9.3.0

⩾ mpicc ring.c -o ring.x

while to run the code:

⩾ mpirun -np N ./ring.x

or

⩾ mpirun -np N ./ring.x n_iterations

In the second case, the code performs n_iterations and measures the average time for one cycle. By default n_iterations = 1. For example, entering

⩾ mpirun -np 4 ./ring.x 10

the output (on login node) is

I am process 0 and I have received 8 messages. My final messages have tags 0, 0; values -6, 6
walltime on processor 0 after 10 iterations 0.00023979. Average: 0.00002398
I am process 1 and I have received 8 messages. My final messages have tags 10, 10; values -6, 6
walltime on processor 1 after 10 iterations 0.00026223. Average: 0.00002622
I am process 2 and I have received 8 messages. My final messages have tags 20, 20; values -6, 6
walltime on processor 2 after 10 iterations 0.00026877. Average: 0.00002688
I am process 3 and I have received 8 messages. My final messages have tags 30, 30; values -6, 6
walltime on processor 3 after 10 iterations 0.00017639. Average: 0.00001764

Eventually, to have better statistics and stable results we considered $n\_iterations = 1000000$.
Actually, data of this Section were obtained with

⩾ mpirun -np N –map-by core ./ring.x 1000000 (pinning different cores for THIN nodes)

⩾ mpirun -np N –map-by hwthread ./ring.x 1000000 (pinning different threads for GPU nodes)

The ring consists of a simple 1D topology where each processor has a left and right neighbour, and there are two possible strategies: we may implement a dedicated a virtual topology or not. Since the topology of the ring is a simple 1D one, we decided to keep it simple and use the 1D topology naturally inherited by MPI_COMM_WORLD. Another possible choice is related to the routines for sending and receiving the messages. To get better performances the code uses the $MPI\_Sendrecv$ routine. Such performances are described in Figure 1 for two runs, one on two THIN nodes

*Electronic address: celoria.marco@gmail.com

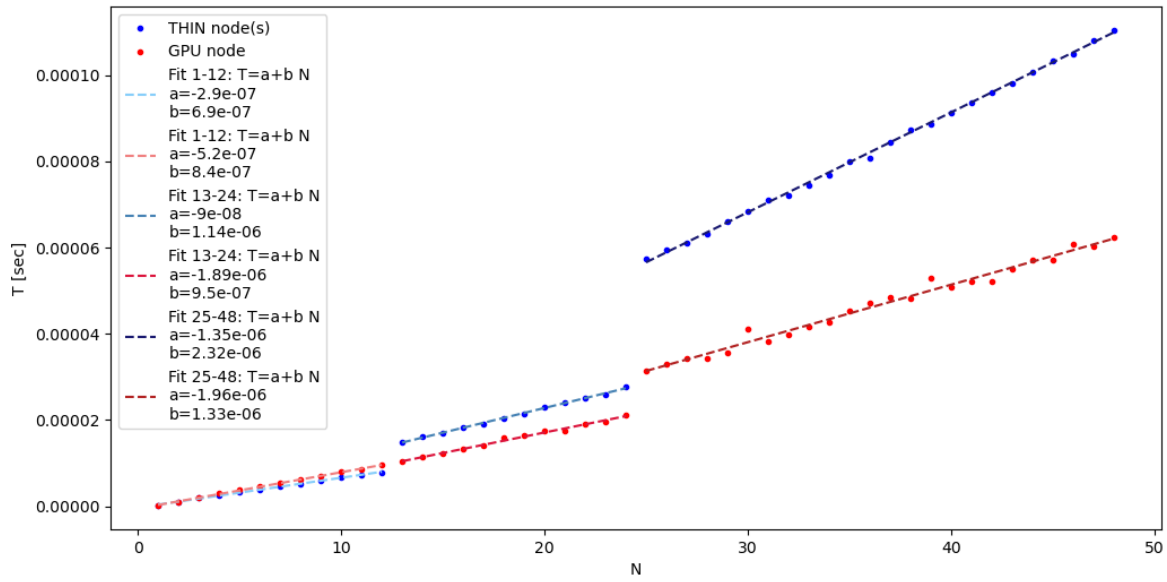

FIG. 1: The average time (in sec) for a cycle on the y-axis as a function of the number of processes N on the x-axis.

and the other on a GPU node with hyper-threading.
We fitted the first 12 processes ($N = 1, \ldots, 12$), i.e. the number of processes that can fit on a socket, with a linear $f(N) = a + b \cdot N$ function using python (specifically using the numpy.polynomial.polynomial.polyfit module).
Then, we fitted the processes $N = 13, \ldots, 24$, i.e. the number of processes that can fit on a THIN node or on one socket on GPU node with hyper-threading.
Finally the processes $N = 25, \ldots, 48$, i.e. the number that can fit on two THIN nodes or one GPU node with hyper-threading.
As one can see, we obtain a three-fold linear scaling (the three-piece-wise linear fit seems very accurate). The possible reason is that the message passing is latency dominated (we are just passing integers), so we can assume that there is a constant fixed amount of time T_ℓ to pass each message. Therefore, considering N processes, we have to pass $2 \cdot N$ messages and we have $T_{N+1} = T_N + 2T_\ell = a + b \cdot N + 2T_\ell = a + b \cdot (N + 1)$. So $2T_\ell = b$ and

$$\begin{cases} T_\ell = \frac{b}{2} \simeq 0.34\mu s & \text{THIN intra-socket} \\ T_\ell = \frac{b}{2} \simeq 0.42\mu s & \text{GPU intra-socket} \\ T_\ell = \frac{b}{2} \simeq 0.57\mu s & \text{THIN intra-node inter-socket} \\ T_\ell = \frac{b}{2} \simeq 0.47\mu s & \text{GPU intra-socket hyper-threading} \\ T_\ell = \frac{b}{2} \simeq 1.16\mu s & \text{THIN inter-node} \\ T_\ell = \frac{b}{2} \simeq 0.66\mu s & \text{GPU intra-node inter-socket hyper-threading} \end{cases} \tag{1}$$

which is more or less consistent (same order of magnitude) with the latency obtained from IMB-MPI1 PingPong of Section 2 (differences might be due to some overhead).

B. Sum 3D matrices

In the second part of Section 1 of the assignment we have to implement a simple 3D matrix-matrix addition in parallel using a 1D, 2D and 3D distribution of data using virtual topology and study its scalability in terms of communication within a single THIN node. Moreover, we have to use just collective operations to communicate among MPI processes. Program should accept as input the sizes of the matrices and should then allocate the matrices and initialize them using double precision random numbers.
We actually wrote the program in C, **section1/sum3Dmatrix.c**, that can be found in the related directory. To

compile the code on ORFEO, navigate to the directory and enter

$\gg$ module load openmpi-4.1.1+gnu-9.3.0

$\gg$ mpicc sum3Dmatrix.c -o sum3Dmatrix.x

There are several ways to run the code, the simplest way on P processors is

$\gg$ mpirun -np P ./sum3Dmatrix.x N_x N_y N_z

for matrices of size $N_x \times N_y \times N_z$. In other words, the program needs three positive integers representing the sizes of the matrices otherwise it throws an error. In this way, the code automatically chooses the virtual topology.
However, one can specify the virtual topology by

$\gg$ mpirun -np P ./sum3Dmatrix.x N_x N_y N_z p_x p_y p_z

The product of the three topology indexes must be equal to the number of processes, otherwise the program exits, i.e. $p_x \times p_y \times p_z = P$.
Finally, the code implements two algorithms, more precisely the program can distribute the elements of the matrices among the processors in two possible ways, which can be chosen by the user by means of

$\gg$ mpirun -np P ./sum3Dmatrix.x N_x N_y N_z p_x p_y p_z a

with $a = 0$ or $a \neq 0$. By default $a = 0$.
Let us explain how these two algorithms work with an example.
Suppose, we define on one processor (say rank 0) three 5×5 matrices A and B (initialized with random doubles) and C. We want to distribute the 25 elements of A and B among 4 processors, i.e. we scatter into A^i_L and B^i_L local matrices ($i = 1, 2, 3, 4$). Then we perform the local sums $C^i_L = A^i_L + B^i_L$, and gather the elements of the C^i_L matrices into C defined on rank 0. There are several ways to distribute the 25 elements, for example

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 & a_5 \\ b_1 & b_2 & b_3 & b_4 & b_5 \\ c_1 & c_2 & c_3 & c_4 & c_5 \\ d_1 & d_2 & d_3 & d_4 & d_5 \\ e_1 & e_2 & e_3 & e_4 & e_5 \end{pmatrix} \Longrightarrow \begin{pmatrix} a_1 & a_2 & a_3 & a_4 & a_5 \\ b_1 & b_2 & b_3 & b_4 & b_5 \\ c_1 & c_2 & c_3 & c_4 & c_5 \\ d_1 & d_2 & d_3 & d_4 & d_5 \\ e_1 & e_2 & e_3 & e_4 & e_5 \end{pmatrix} \text{ or } \left[\begin{pmatrix} a_1 & a_2 & a_3 & a_4 & a_5 \\ b_1 & b_2 & b_3 & b_4 & b_5 \\ c_1 & c_2 & c_3 & c_4 & c_5 \\ d_1 & d_2 & d_3 & d_4 & d_5 \\ e_1 & e_2 & e_3 & e_4 & e_5 \end{pmatrix} \text{ or } \begin{pmatrix} a_1 & a_2 & a_3 & a_4 & a_5 \\ b_1 & b_2 & b_3 & b_4 & b_5 \\ c_1 & c_2 & c_3 & c_4 & c_5 \\ d_1 & d_2 & d_3 & d_4 & d_5 \\ e_1 & e_2 & e_3 & e_4 & e_5 \end{pmatrix} \right] \tag{2}$$

Namely:

1. In the first splitting, we can assign 7 elements to the first processor and 6 to the other three processors. This democratic approach (corresponding to $a = 0$) divides the load in the most balanced way and it does not depend on the virtual topology.
2. A second possibility would be to assign 10 elements to the first processor and 5 elements to the others (corresponding to $a \neq 0$ with a 1D virtual topology).
3. Finally, we can assign 9 elements to the first processor, 6 elements to the second and third processors and 4 to the forth (corresponding to $a \neq 0$ with a 2D virtual topology).

Note, however, that we store the 3D matrices as vectors, i.e.

a_1	a_2	a_3	a_4	a_5	b_1	b_2	b_3	b_4	b_5	c_1	c_2	c_3	c_4	c_5	d_1	d_2	d_3	d_4	d_5	e_1	e_2	e_3	e_4	e_5

(3)

So reproducing the first and second splittings is rather straightforward

a_1	a_2	a_3	a_4	a_5	b_1	b_2	b_3	b_4	b_5	c_1	c_2	c_3	c_4	c_5	d_1	d_2	d_3	d_4	d_5	e_1	e_2	e_3	e_4	e_5

(4)

and

a_1	a_2	a_3	a_4	a_5	b_1	b_2	b_3	b_4	b_5	c_1	c_2	c_3	c_4	c_5	d_1	d_2	d_3	d_4	d_5	e_1	e_2	e_3	e_4	e_5

(5)

since data are contiguous in memory.
However, for the for the third splitting, we would either use a buffer or re-organize the elements in the vector as

a_1	a_2	a_3	b_1	b_2	b_3	c_1	c_2	c_3	a_4	a_5	b_4	b_5	c_4	c_5	d_1	d_2	d_3	e_1	e_2	e_3	d_4	d_5	e_4	e_5

(6)

However, we have not implemented such re-ordering as it is not really needed (there is no halo communication and we are summing matrices whose elements are random double), and in fact, in this case the code just distributes the 25 elements as

$$\boxed{a_1\,a_2\,a_3\,a_4\,a_5\,b_1\,b_2\,b_3\,b_4\,b_5\,c_1\,c_2\,c_3\,c_4\,c_5\,d_1\,d_2\,d_3\,d_4\,d_5\,e_1\,e_2\,e_3\,e_4\,e_5} \tag{7}$$

In conclusion, what we have really implemented is the following splitting of the load among the processors

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 & a_5 \\ b_1 & b_2 & b_3 & b_4 & b_5 \\ c_1 & c_2 & c_3 & c_4 & c_5 \\ d_1 & d_2 & d_3 & d_4 & d_5 \\ e_1 & e_2 & e_3 & e_4 & e_5 \end{pmatrix} \Longrightarrow \underbrace{\begin{pmatrix} a_1 & a_2 & a_3 & a_4 & a_5 \\ b_1 & b_2 & b_3 & b_4 & b_5 \\ c_1 & c_2 & c_3 & c_4 & c_5 \\ d_1 & d_2 & d_3 & d_4 & d_5 \\ e_1 & e_2 & e_3 & e_4 & e_5 \end{pmatrix}}_{a=0 \text{ (topology independent algorithm)}} \quad \text{or} \quad \underbrace{\left[\overbrace{\begin{pmatrix} a_1 & a_2 & a_3 & a_4 & a_5 \\ b_1 & b_2 & b_3 & b_4 & b_5 \\ c_1 & c_2 & c_3 & c_4 & c_5 \\ d_1 & d_2 & d_3 & d_4 & d_5 \\ e_1 & e_2 & e_3 & e_4 & e_5 \end{pmatrix}}^{\text{1D virtual topology}} \text{or} \overbrace{\begin{pmatrix} a_1 & a_2 & a_3 & a_4 & a_5 \\ b_1 & b_2 & b_3 & b_4 & b_5 \\ c_1 & c_2 & c_3 & c_4 & c_5 \\ d_1 & d_2 & d_3 & d_4 & d_5 \\ e_1 & e_2 & e_3 & e_4 & e_5 \end{pmatrix}}^{\text{2D virtual topology}} \right]}_{a \neq 0 \text{ (topology dependent algorithm)}} \tag{8}$$

and similarly for B.
In order to run this example, the user can enter the following

$\gg$ mpirun -np 4 ./sum3Dmatrix.x $\underbrace{5\ 5\ 1}$

$\gg$ mpirun -np 4 ./sum3Dmatrix.x $\underbrace{5\ 5\ 1}\ \underbrace{4\ 1\ 1}\ \underbrace{1}_{v}$

$\gg$ mpirun -np 4 ./sum3Dmatrix.x $\underbrace{5\ 5\ 1}_{size}\ \underbrace{2\ 2\ 1}_{topo}\ \underbrace{1}_{v}$

where

- 5 5 1 indicates matrices of size $5 \times 5 \times 1$
- 4 1 1 or 2 2 1 indicates that the 4 processors should be organized in a 1D or 2D virtual topology, respectively
- the final 1 enables the topology sensitivity $a \neq 0$

We have considered matrices of the following sizes

- $2400 \times 100 \times 100$;
- $1200 \times 200 \times 100$;
- $800 \times 300 \times 100$;

using a THIN node with 24 cores. We tested all possible configurations (to have enough statistics and minimize random fluctuations in data we performed 10 runs embedding the code in a loop of 50 cycles, taking the average time per one cycle T_i, and finally taking the average among the 10 runs $\bar{T}$):

$$\bar{T} = \frac{1}{10}\sum_{i=1}^{10} T_i \qquad S^2 = \frac{1}{9}\sum_{i=1}^{10}(T_i - \bar{T})^2 \qquad CI = \left[\bar{T} - \frac{cS}{\sqrt{10}}, \bar{T} + \frac{cS}{\sqrt{10}}\right] \tag{9}$$

with $c = 1.96$ for approximate 95% confidence intervals. The results are shown in Table I, II and III.

When $a = 0$ we do not expect any dependence on the virtual topology. On the other hand, in principle, there might be some dependence on the virtual topology when the topology sensitive option ($a \neq 0$) is turned on, as the distribution of the load is sometimes less balanced. However, the unbalancing is so small that there are no statistical evidence of such dependence from data. Namely, by repeating the process, we can see that the data are dominated by random statistics rather than a recurrent fixed pattern. In principle, the most balanced is the load, the most efficient is the code.
Again, we stress the fact that there is no halo communication, and the only communications are collective operations: specifically two $MPI\_Scatterv$ for scattering the two matrices A and B and one $MPI\_Gatherv$ to collect the results of the local matrices into the matrix C on rank 0 process.
As a final consideration, we measured (by means of the $MPI\_Wtime()$ routine) the time spent on $MPI\_Scatterv$ and $MPI\_Gatherv$, finding that around $\sim 87\%$ of the total time is spend on scattering the two matrices, $\sim 12\%$ of the total time in the gathering process while only $\sim 1\%$ in the rest part of the code.

Matrix size	Topology	a	Distribution of matrix load/elements per processors	Average time (95% CI) [s]
$2400 \times 100 \times 100$	$(24, 1, 1)$	0	P. 0-23: 1000000	[0.62277, 0.630782]
$2400 \times 100 \times 100$	$(12, 2, 1)$	0	P. 0-23: 1000000	[0.623443, 0.628308]
$2400 \times 100 \times 100$	$(8, 3, 1)$	0	P. 0-23: 1000000	[0.623607, 0.639931]
$2400 \times 100 \times 100$	$(6, 4, 1)$	0	P. 0-23: 1000000	[0.624049, 0.630209]
$2400 \times 100 \times 100$	$(6, 2, 2)$	0	P. 0-23: 1000000	[0.623954, 0.642249]
$2400 \times 100 \times 100$	$(4, 3, 2)$	0	P. 0-23: 1000000	[0.623597, 0.629541]
$2400 \times 100 \times 100$	$(24, 1, 1)$	1	P. 0-23: 1000000	[0.624222, 0.628871]
$2400 \times 100 \times 100$	$(12, 2, 1)$	1	P. 0-23: 1000000	[0.623427, 0.634731]
$2400 \times 100 \times 100$	$(8, 3, 1)$	1	P. 0,3,6,9,12,15,18,21: 1020000 P. 1,2,4,5,7,8,10,11,13,14,16,17,19,20,22,23: 990000	[0.62446, 0.630308]
$2400 \times 100 \times 100$	$(6, 4, 1)$	1	P. 0-23: 1000000	[0.623361, 0.62908]
$2400 \times 100 \times 100$	$(6, 2, 2)$	1	P. 0-23: 1000000	[0.623361, 0.62908]
$2400 \times 100 \times 100$	$(4, 3, 2)$	1	P. 0,1,6,7,12,13,18,19: 1020000 P. 2,3,4,5,8,9,10,11,14,15,16,17,20,21,22,23: 990000	[0.624406, 0.630358]

TABLE I: Average time for different configurations.

Matrix size	Topology	a	Distribution of matrix load/elements per processors	Average time (95% CI) [s]
$1200 \times 200 \times 100$	$(24, 1, 1)$	0	P. 0-23: 1000000	[0.623598, 0.628811]
$1200 \times 200 \times 100$	$(12, 2, 1)$	0	P. 0-23: 1000000	[0.622559, 0.628498]
$1200 \times 200 \times 100$	$(8, 3, 1)$	0	P. 0-23: 1000000	[0.623823, 0.628943]
$1200 \times 200 \times 100$	$(6, 4, 1)$	0	P. 0-23: 1000000	[0.62274, 0.628668]
$1200 \times 200 \times 100$	$(6, 2, 2)$	0	P. 0-23: 1000000	[0.624154, 0.630386]
$1200 \times 200 \times 100$	$(4, 3, 2)$	0	P. 0-23: 1000000	[0.622725, 0.628027]
$1200 \times 200 \times 100$	$(24, 1, 1)$	1	P. 0-23: 1000000	[0.623546, 0.628845]
$1200 \times 200 \times 100$	$(12, 2, 1)$	1	P. 0-23: 1000000	[0.621256, 0.642412]
$1200 \times 200 \times 100$	$(8, 3, 1)$	1	P. 0,1,3,4,6,7,9,10,12,13,15,16,18,19,21,22: 1005000 P. 2,5,8,11,14,17,20,23: 990000	[0.619586, 0.640735]
$1200 \times 200 \times 100$	$(6, 4, 1)$	1	P. 0-23: 1000000	[0.62336, 0.628699]
$1200 \times 200 \times 100$	$(6, 2, 2)$	1	P. 0-23: 1000000	[0.623165, 0.629591]
$1200 \times 200 \times 100$	$(4, 3, 2)$	1	P. 0,1,2,3,6,7,8,9,12,13,14,15,18,19,20,21: 1005000 P. 4,5,10,11,16,17,22,23: 990000	[0.622751, 0.633312]

TABLE II: Average time for different configurations.

II. SECTION 2: MEASURE MPI POINT TO POINT PERFORMANCE

In this Section, we use the Intel MPI benchmark to estimate latency and bandwidth of all available combinations of topologies and networks on ORFEO computational nodes, using both Intel MPI and Open MPI libraries.
Results can be found in the directory "**section2**/" where the reader can find ".csv" files with

- the command line used to produce the data
- the list of nodes involved
- the latency and the bandwith computed by
  - fitting the data by a least-square fitting model ("*_fit.csv" files)
  - extrapolating the latency and the bandwith obtained by looking at (averaging over) the first [1-4] bytes (latency dominated) data and the last [33554432 - 268435456] bytes (bandwidth dominated) data ("*_extra.csv" files).

Moreover, we provide ".png" files depicting graphs/plots for the time and the throughput with both the least-square fit and the "extrapolated" curve. Here we comment the tests we have performed, the theoretical expectations, and

Matrix size	Topology	a	Distribution of matrix load/elements per processors	Average time (95% CI) [s]
$800 \times 300 \times 100$	$(24, 1, 1)$	0	P. 0-23: 1000000	[0.623896, 0.629036]
$800 \times 300 \times 100$	$(12, 2, 1)$	0	P. 0-23: 1000000	[0.623639, 0.629578]
$800 \times 300 \times 100$	$(8, 3, 1)$	0	P. 0-23: 1000000	[0.620987, 0.642372]
$800 \times 300 \times 100$	$(6, 4, 1)$	0	P. 0-23: 1000000	[0.624624, 0.647822]
$800 \times 300 \times 100$	$(6, 2, 2)$	0	P. 0-23: 1000000	[0.623852, 0.629266]
$800 \times 300 \times 100$	$(4, 3, 2)$	0	P. 0-23: 1000000	[0.623753, 0.629793]
$800 \times 300 \times 100$	$(24, 1, 1)$	1	P. 0,1,2,3,4,5,6,7: 1005000 P. 8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23: 990000	[0.622801, 0.628243]
$800 \times 300 \times 100$	$(12, 2, 1)$	1	P. 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15: 1005000 P. 16,17,18,19,20,21,22,23: 990000	[0.623037, 0.629581]
$800 \times 300 \times 100$	$(8, 3, 1)$	1	P. 0-23: 1000000	[0.6224, 0.628294]
$800 \times 300 \times 100$	$(6, 4, 1)$	1	P. 0,1,2,3,4,5,6,7: 1005000 P. 8,9,10,11,12,13,14,15,16,17,18,19,20,21,22: 997500	[0.62232, 0.639477]
$800 \times 300 \times 100$	$(6, 2, 2)$	1	P. 0,1,2,3,4,5,6,7: 1005000 P. 8,9,10,11,12,13,14,15,16,17,18,19,20,21,22: 997500	[0.625482, 0.632679]
$800 \times 300 \times 100$	$(4, 3, 2)$	1	P. 0-23: 1000000	[0.623618, 0.628644]

TABLE III: Average time for different configurations.

the data analysis procedure.

A. Open MPI, Intel MPI and ORFEO network

1. Open MPI

Consider first Open MPI's functionality (MPI_API) based on the Modular Component Architecture (MCA), i.e. a series of frameworks, components, and modules that are assembled at run-time to create an MPI implementation. In general several MCA are available, however, in this Section, we focus on MCA btl (MPI point-to-point Byte Transfer Layer, used for MPI point-to-point messages on some types of networks) and MCA pml (Point-to-point management layer: used for fragmenting, reassembly, top-layer protocols, etc.). On OFERO we have the following MCA pml

- *MCA pml: v (MCA v2.1.0, API v2.0.0, Component v4.1.1)*
- *MCA pml: cm (MCA v2.1.0, API v2.0.0, Component v4.1.1)*
- *MCA pml: monitoring (MCA v2.1.0, API v2.0.0, Component v4.1.1)*
- MCA pml: ob1 (MCA v2.1.0, API v2.0.0, Component v4.1.1)
  Multi-device, multi-rail engine using BTL components (byte transfer layer)
- MCA pml: ucx (MCA v2.1.0, API v2.0.0, Component v4.1.1)
  Multi-device multi-rail transport library using the UCX communication library
  (Unified Communications X)

and we considered the MCA pml: ob1 and MCA pml: ucx. Moreover, on OFERO we have the following MCA btl

- MCA btl: self (MCA v2.1.0, API v3.1.0, Component v4.1.1)
  Process-loopback communications, i.e., when an MPI process sends to itself
- MCA btl: openib (MCA v2.1.0, API v3.1.0, Component v4.1.1)
  The openib BTL uses the OpenFabrics Alliance's (OFA) verbs API stack to support InfiniBand, RoCE, and iWARP devices.
- MCA btl: tcp (MCA v2.1.0, API v3.1.0, Component v4.1.1)
  This will direct Open MPI to use TCP-based communications over IP interfaces / networks.
- MCA btl: vader (MCA v2.1.0, API v3.1.0, Component v4.1.1)
  Low-latency, high-bandwidth mechanism for transferring data between two processes via shared memory.
  This BTL can only be used between processes executing on the same node.

2. Intel MPI

Similarly to Open MPI, also Intel MPI Library enables to select a communication fabric at runtime without having to recompile the application. The Intel MPI Library supports the MPI low level transport Open Fabrics Interfaces (OFI) framework. In particular, the following fabrics are available:

- shm: Shared memory transport (used for intra-node communication only).
- ofi: OpenFabrics Interfaces (OFI)-capable network fabrics, such as Intel True Scale Fabric, Intel Omni-Path Architecture, InfiniBand, and Ethernet (through OFI API).

and there are several OFI providers, for example

- MLX
  The MLX provider runs over the UCX that is currently available for the Mellanox InfiniBand hardware.
- verbs
  The verbs provider enables applications using OFI to be run over any verbs hardware (InfiniBand, iWarp, and so on). It uses the Linux Verbs API for network transport and provides a translation of OFI calls to appropriate verbs API calls. It uses librdmacm for communication management and libibverbs for other control and data transfer operations.
- TCP
  The TCP provider is a general purpose provider for the Intel MPI Library that can be used on any system that supports TCP sockets to implement the libfabric API. The provider lets you run the Intel MPI Library application over regular Ethernet, in a cloud environment that has no specific fast interconnect or using IPoIB.

3. ORFEO hardware specifications

We conclude this introductory part with some considerations on ORFEO network:

- On ORFEO the High Speed network consists of 100 Gbit HDR Infiniband. Thus, for communication between different nodes, the maximum bandwith is 12.5 GB/s using Infiniband. There are also a 25 Gbit Ethernet typically for in band management network and 1 Gbit Ethernet for out of band management network.
  According to interface configuration (ifconfig), "ib0" identifies Infiniband while "br0" can be used for Ethernet.
- The TCP/IP and IPoIB protocols are typically more complicated (more sofware layers) and thus slower (larger software latency) compared to native Infiniband protocols[1].
- For communications inside one single node, shared memory can play an important role.

On THIN nodes we have 24 processors divided into two sockets, 48 processes for GPU because of hyper-threading

- Cache Level 1: 32 kB
  ```
  THIN Groups:
  (0) (2) (4) (6) (8) (10) (12) (14) (16) (18) (20) (22) (1) (3) (5) (7) (9) (11) (13) (15) (17) (19) (21) (23)
  GPU Groups:
  (0 24)(2 26)(4 28)(6 30)(8 32)(10 34)(12 36)(14 38)(16 40)(18 42)(20 44)(22 46)(1 25)(3 27)(5 29)(7 31)(9 33)(11 35)(13 37)(15 39)(17 41)(19 43)(21 45)(23 47)
  ```
- Cache Level 2: 1 MB
  ```
  THIN Groups:
  (0) (2) (4) (6) (8) (10) (12) (14) (16) (18) (20) (22) (1) (3) (5) (7) (9) (11) (13) (15) (17) (19) (21) (23)
  GPU Groups:
  (0 24)(2 26)(4 28)(6 30)(8 32)(10 34)(12 36)(14 38)(16 40)(18 42)(20 44)(22 46)(1 25)(3 27)(5 29)(7 31)(9 33)(11 35)(13 37)(15 39)(17 41)(19 43)(21 45)(23 47)
  ```
- Cache Level 3: 19 MB (9.5 MB per socket)
  ```
  THIN Groups:
  ( 0 2 4 6 8 10 12 14 16 18 20 22 ) ( 1 3 5 7 9 11 13 15 17 19 21 23 )
  GPU Groups:
  ( 0 24 2 26 4 28 6 30 8 32 10 34 12 36 14 38 16 40 18 42 20 44 22 46 ) ( 1 25 3 27 5 29 7 31 9 33 11 35 13 37 15 39 17 41 19 43 21 45 23 47 )
  ```

In the following, we will use the IMB-MP1 PingPong point-to-point communications benchmark to measure the latency and the throughput for different configurations in terms of "topology" (inter-node, intra-node inter-socket, intra-node intra-socket communications), libraries (Open MPI, Intel MPI) as well as different OFI/MCA.

---

[1] For instance there is no DNS management tool because the network is private and not on a wide area network

B. Theoretical background

In class we have discussed a very simple model for single point-to-point connection (see also [1]: 4.5.1 Basic performance characteristics of networks):
Assuming that the total transfer time for a message of size N [bytes] is composed of latency and streaming parts,

$$T = T_\ell + \frac{N}{B} \tag{10}$$

and B being the maximum (asymptotic) network bandwidth in MB/sec, the effective bandwidth is

$$B_{eff} = \frac{N}{T_\ell + \frac{N}{B}} = \frac{B}{\frac{T_\ell B}{N} + 1} \tag{11}$$

Note that:

- In the most general case T_ℓ and B depend on the message length N.
- We have the following limits

$$\lim_{N\to\infty} B_{eff} = B \qquad \lim_{N\to 0} B_{eff} = 0 \qquad \lim_{N\to 0} T = T_\ell \tag{12}$$

- Latency and bandwidth of message transfers between two cores on the same socket depend on whether the message fits into the shared cache.

Actually, intra-node point-to-point communications are very well explained in [1] (10.5 Understanding intra-node point-to-point communication). Here, we try to briefly summarize the logic.
In general, we can expect three different levels of point-to-point communication characteristics, depending on whether message transfer occurs

1. intra-node intra-socket: communications between cores on the same socket can benefit from the shared cache (leading to a peak bandwidth of over 20 GB/sec).
2. intra-node inter-socket: communications between cores on different sockets
3. inter-node: communications between different nodes

Surprisingly, the characteristics for inter-socket communication are very similar to the intra-node intra-socket ones, although there is no shared cache and the large bandwidth "hump" should not be present because all data must be exchanged via main memory.
Without entering in the details (that can be found e.g. in [1, 2]), a possible explanation may come from the specific implementation of the IMB-MPI1 PingPong benchmark, where to get accurate timing measurements even for small messages, the Ping-Pong message transfer is repeated a number of times.
Remarkably, in the first iteration (first ping and first pong) buffers are copied from the local memories of the two processors to the relative caches. Then, after the first iteration (second ping, second pond and so on), the send buffers are located in the caches of the receiving processes and in-cache copy operations occur in the subsequent iterations instead of data transfer through memory. Therefore, even for inter-socket communication, in-cache copy operations play a significant role.
On the other hand, there are two reasons for the performance drop at larger message sizes:

1. The L3 cache is too small to hold both or at least one of the local receive buffer and the remote send buffer.
2. The IMB benchmark is performed so that the number of repetitions is decreased with increasing message size until only one iteration – which is the initial copy operation through the network – is done for large messages.

C. Data analysis

We have performed ten runs for each configuration, calculated mean values, and reported the results in the ".csv" files together with the plots in the ".png" files. For each configuration, there are in fact two ".csv" and the ".png" files contain plots with data points together with two fitting curves. The reason is that there are two possible ways to get the latency and the asymptotic bandwidth from data.
Consider the time and throughput measured by means of IMB-MPI1 PingPong, we can calculate T_ℓ and B:

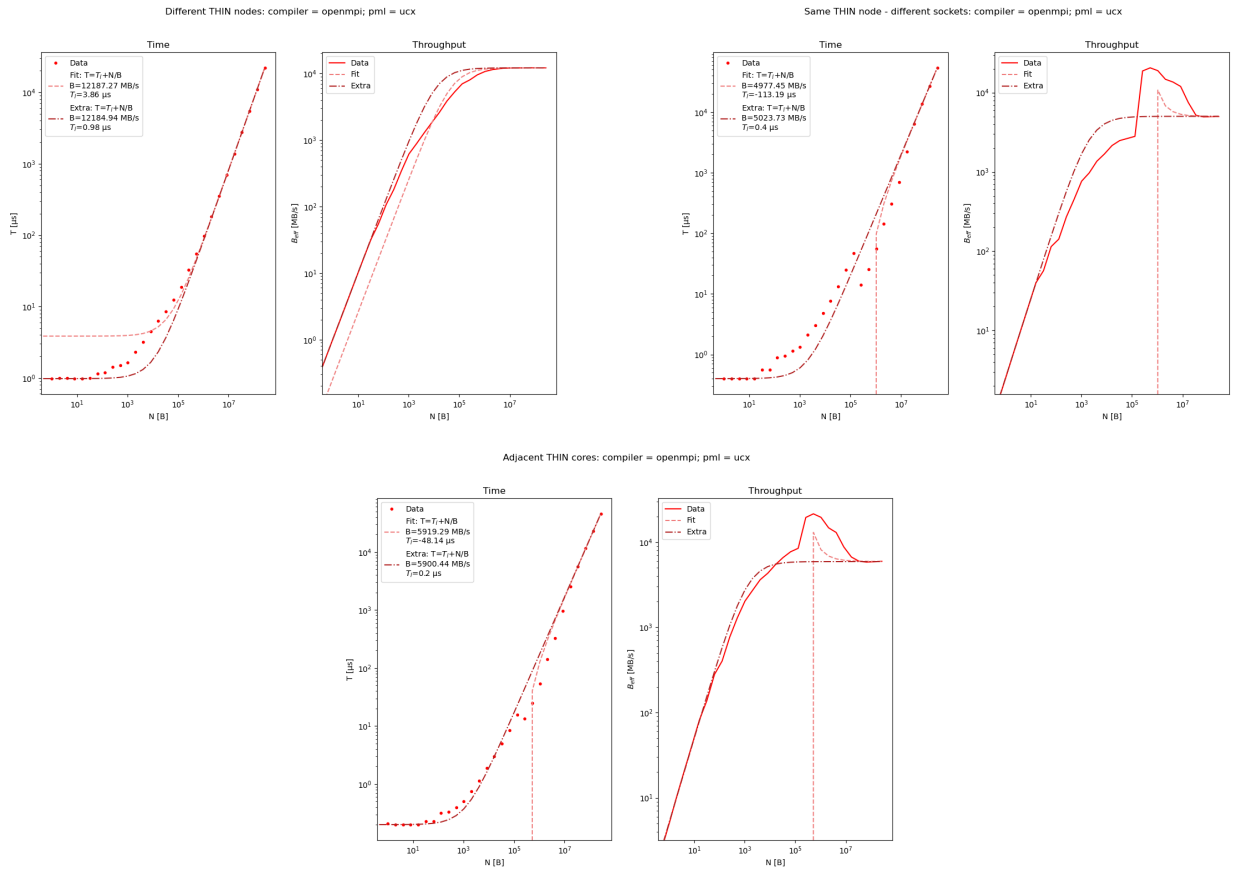

FIG. 2: Time and throughput $B_{eff}(N)$ with least-square fit and "extrapolated" fit

1. using a least-square fitting model.
2. by taking the limits

$$\lim_{N\to\infty} B_{eff} = B \qquad\qquad \lim_{N\to 0} T = T_\ell \tag{13}$$

So we have the files "*_fit.csv" with time and throughput computed using the latency and asymptotic bandwidth obtained from a least-square fitting model, and we have "*_extra.csv" with time and throughput computed using the latency and asymptotic bandwidth obtained by averaging the first data points and the last data points. More specifically, we averaged the latency using time associated with messages $N \in [1, 4]$ bytes (latency dominated) data and for the asymptotic bandwidth we averaged over the last $N \in [33554432, 268435456]$ bytes (bandwidth dominated) data. There are pros and cons for the two methods:

- There is a very large separation of scales and large (asymptotic bandwidth dominated) messages are more relevant for the least-square fitting model compared to small (latency dominated) messages.
  - As a consequence the least-square fitting model, even though by definition is the method which minimizes the error with respect to the linear model, sometimes (especially in the case of in-cache copy operations) results in a nonphysical negative fitted latency. As an example, in Figure 2, one can see data points together with least-square fit and "extrapolated" fit for THIN inter-node, inter-socket, intra-socket Open MPI communications using pml:ucx. For inter-node communications the latency computed extrapolating from the first data transferred $T_\ell^{extra} = 0.98\ \mu s$ kind agrees with the least-square fit $T_\ell^{l\Box fit} = 3.96\ \mu s$ (obtained again using the numpy.polynomial.polynomial.polyfit module). On the other hand, for inter-socket the two methods give very different latencies: $T_\ell^{extra} = 0.4\ \mu s$ and $T_\ell^{l\Box fit} = -113.19\ \mu s$, similarly for intra-socket we have $T_\ell^{extra} = 0.2\ \mu s$ and $T_\ell^{l\Box fit} = -48.14\ \mu s$. Remarkably, bandwidth are always in reasonably agreement.

  - Conversely, extrapolating the latency using the small messages data and the asymptotic bandwidth using large messages (the tails of the data points) results in a more physical evaluation of the latency but a less accurate overall fit for the linear model.

- In fact, the linear model doesn't appear to be a good model to describe the transition between latency dominated and asymptotic bandwidth dominated regimes, especially for the shared memory intra-node communications.

- A possibility is to consider different models, for instance piecewise linear models (fitting the first half of the data with a linear model and then the second half with a different linear model) or even more complicated models (in the spirit, for instance, of [3], even though their analysis is for the multi-mode PingPong benchmark).

In Figure 3, 4, 5 we identify typical behaviors for inter-node, intra-node inter-socket and intra-node intra-socket communications, respectively.
The main features for intra-node communications are

- inter-node openib, ucx, mlx have similar behavior on THIN and GPU nodes reaching the expected Infiniband asymptotic bandwidth of around $B \sim 12500$ MB/s.

- inter-node tcp protocols (both Open MPI and Intel MPI, "ib0" and "br0") are slower with an asymptotic bandwidth in the range $B \sim 2000-3000$ MB/s with highest values on THIN nodes rather than GPU nodes. Moreover, we observe some discontinuities with Open MPI at $N \sim 10^5$ B.

- Intel MPI verbs, both on THIN and GPU nodes, is in the middle with $B \sim 8000$ MB/s and featuring the characteristic and evident discontinuities at $N \sim 10^5$ B.

The main features for intra-node inter-socket communications are

- ucx, mlx have similar behavior with the large bandwidth "hump" (around $B_{eff} \sim 20000$ MB/s) peaking on $N \sim 10^6$ bytes (i.e. L2 cache size) reaching an asymptotic bandwidth $B \sim 5000$ MB/s for $N \gtrsim 10^7$ bytes (i.e. messages larger than L3 cache). Note that, for both ucx and mlx, the peaks in throughput are on THIN nodes.

- Among the other shared memory possibilities we identify three situations which differ in the maximum throughput and the asymptotic throughput. In general, we recognize the typical in-cache copy features (throughput peak corresponding to $N \sim 10^6$ bytes; performance drop and asymptotic bandwidth regime corresponding to messages of size $N \gtrsim 10^7$ bytes). More specifically, Ompen MPI vader shows a significantly higher maximum bandwidth peak B_{eff}, and a slightly lower asimptotic bandwidth B with respect to Intel MPI shm OFIs.

- verbs, openib and tcp have similar behavior with respect to inter-node communications, with tcp asymptotic bandwidth $B \sim 3500$ MB/s slightly higher than the inter-node one.

For intra-node intra-socket, we find an overall similarity compared to inter-socket communications, the main differences being:

- Intel MPI OFI shm have similar (on GPU nodes) or even higher (on THIN nodes) maximum bandwidth peak B_{eff} and a slightly higher asimptotic bandwidth B compared to Open MPI vader.

- The intra-socket tcp protocols feature a higher asymptotic bandwidth $B \sim 5000$ MB/s with respect to inter-socket ones $B \sim 3000$ MB/s and a peak of $B_{eff} \sim 7000$ MB/s associated to message of size $N \sim 10^6$ bytes.

III. SECTION 3: COMPARE PERFORMANCE OBSERVED AGAINST PERFORMANCE MODEL FOR JACOBI SOLVER

In the last Section of the Assignment 1, we have to

1. Compile and run the code on on single processor of a THIN and GPU node to estimate the serial time on one single core.

2. Run it on 4/8/12 processes within the same node pinning the MPI processes within the same socket and across two sockets

   - Identify for the two cases the right latency and bandwith to use in the performance model.

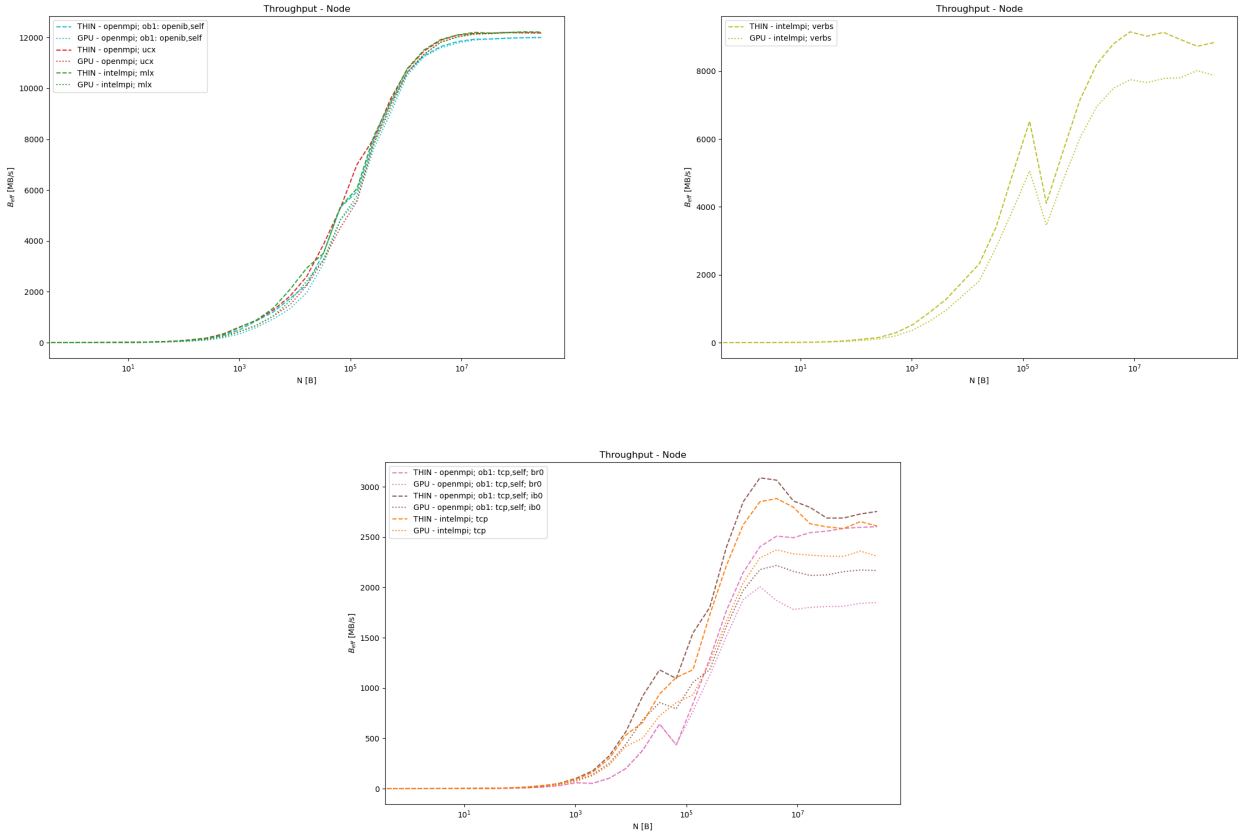

FIG. 3: Throughput $B_{eff}(N)$ for inter-node communications

- Report and check if scalability fits the expected behaviour of the model discussed in class.

3. Run it on 12 24 48 processor using two thin nodes
   - Identify for the two cases the right latency and bandwith to use in the performance model.
   - Report and check if scalability fits the expected behaviour of the model discussed in class.
4. Repeat the previous experiment on a GPU node where hyper-threading is enabled.

The results are shown in Table IV, V, VI, VII, VIII, IX, X, XI for $L = 1200$ with the latency and bandwidth obtained from Section 2, using the IMP-MPI1 Pingpong program for the cases [THIN/GPU node(s) : pml ucx –map-by core/socket/ppr:$\frac{N}{2}$:node].
In particular, we are interested in the performance (P measured in [MLUPs/sec]) computed by means of the theoretical model (discussed in class, see also [1] - Section 9.3.2 Performance properties) and the one measured by the Jacobi code.
From the theoretical point of view, the performance on $N = N_x \times N_y \times N_z$ processes for a particular overall problem size of L^3N grid points (using cubic subdomains of size L^3) is

$$P(L, \vec{N}) = \frac{L^3 N}{T_s(L) + T_c(L, \vec{N})} \tag{14}$$

with

$$T_c(L, \vec{N}) = \frac{c(L, \vec{N})}{B} + kT_\ell \tag{15}$$

where B is the bandwidth, k the largest number (over all subdomains) of coordinate directions in which the number of processes is greater than one and

$$c(L, \vec{N}) = L^2 \cdot k \cdot 2 \cdot 8 \tag{16}$$

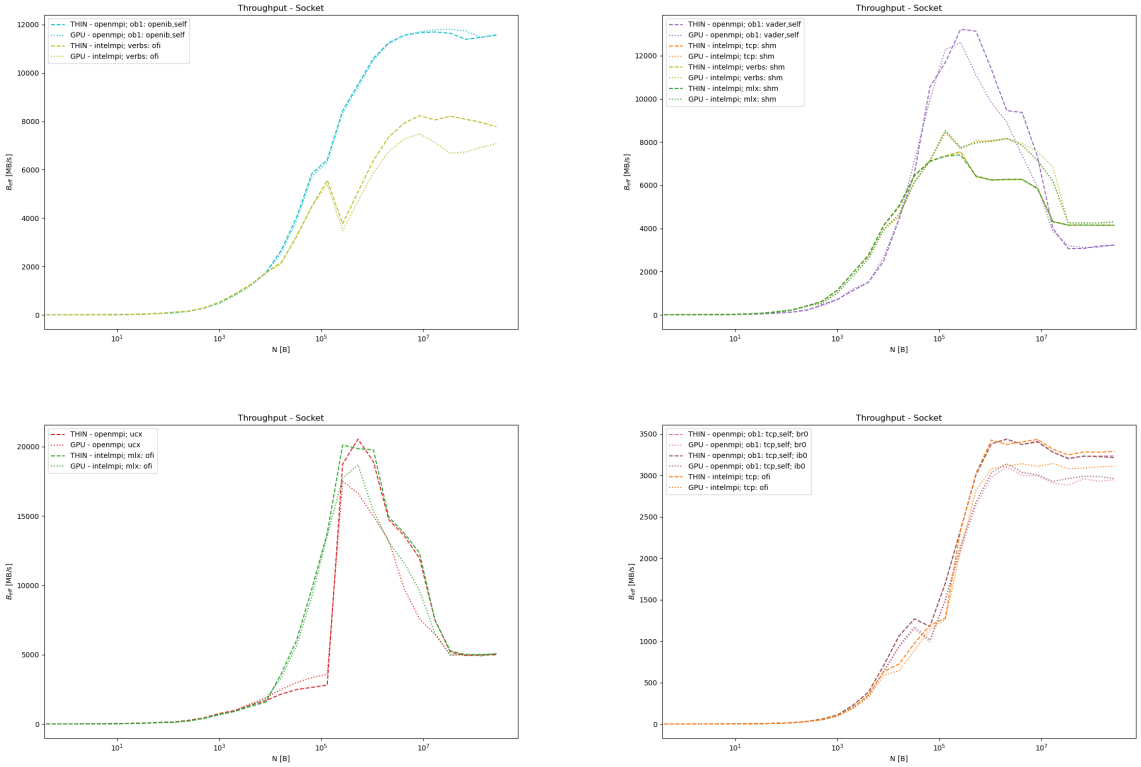

FIG. 4: Throughput $B_{eff}(N)$ for intra-node inter-socket communications

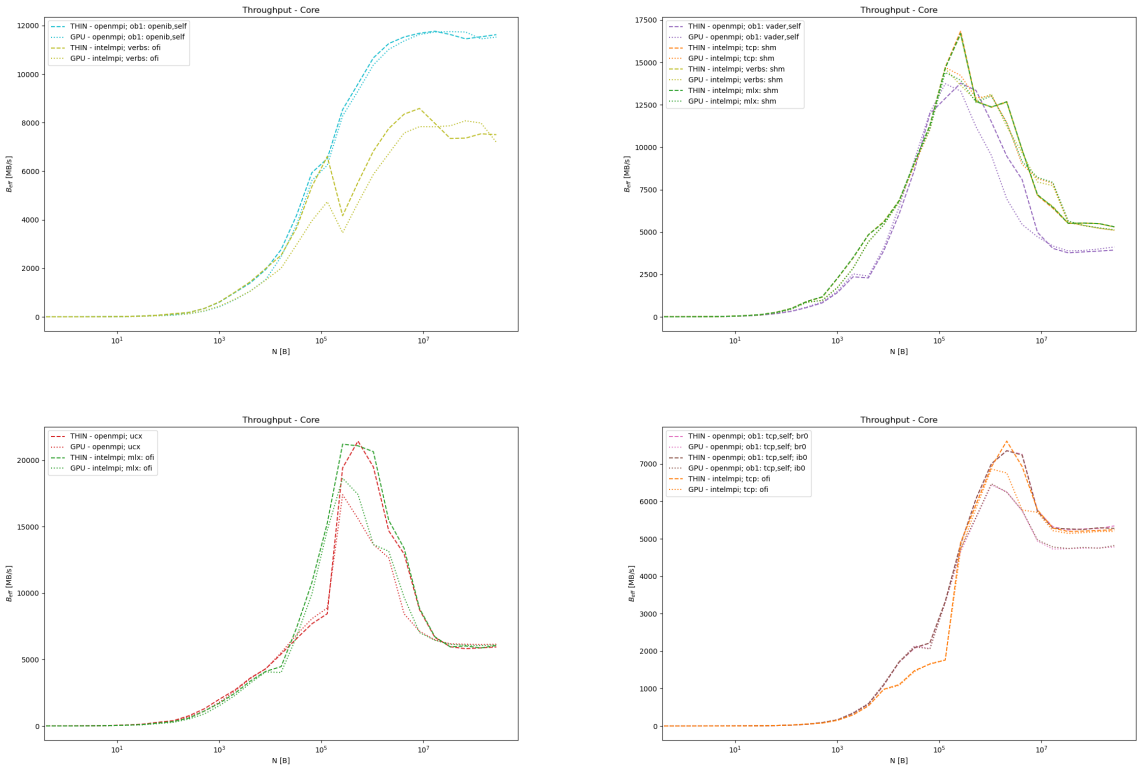

FIG. 5: Throughput $B_{eff}(N)$ for intra-node intra-socket communications

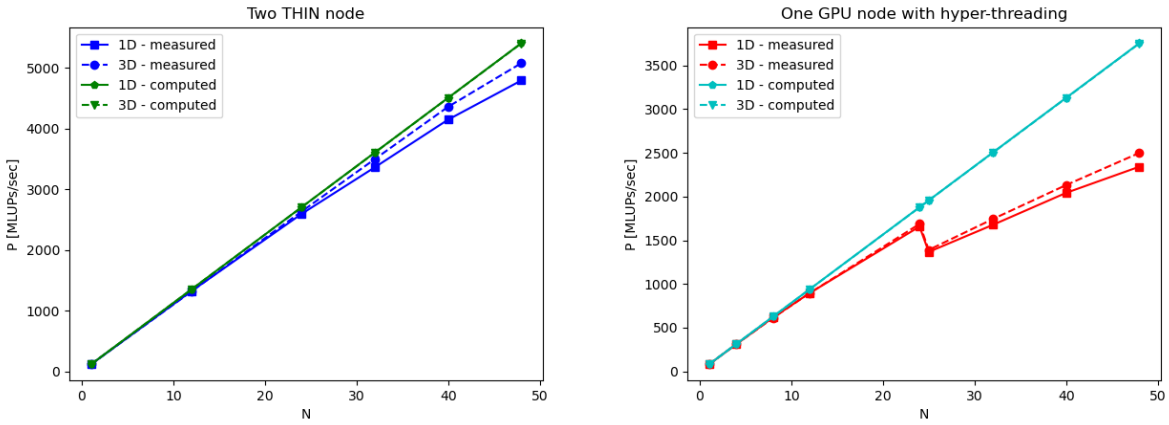

FIG. 6: Comparison between P as a function of N on two THIN nodes and on one GPU with hyper-threading

However, it is important to note that on ORFEO $T_s^{THIN} \simeq T_s^{GPU} \simeq 10^1$ s, while typically $k \simeq 10^1$, $T_\ell \simeq 10^{-6}s$, $B \simeq 10^{9-10}$ B/s and $c \simeq 10^{7-8}$ B so

$$T_c \simeq \frac{10^{7-8}B}{10^{9-10}B/s} + 10^1 10^{-6}s \simeq 10^{-2}s \ll 10^1 s \simeq T_s \tag{17}$$

In other words, we would expect from dimensional arguments and rough order of magnitude analysis

$$P(L,\vec{N}) \simeq \frac{NL^3}{T_s(L)(1+10^{-3})} \qquad \Longrightarrow \qquad \frac{NP_1(L)}{P(L,\vec{N})} \simeq \frac{N(1\times L^3)}{T_s}\frac{T_s(1+10^{-3})}{N\times L^3} \simeq 1.001 \tag{18}$$

i.e. we can expect reasonably linear scaling behavior as the serial time is much larger than Infiniband or shared memory communication (low latency / high bandwidth). Note that even the TCP communication is fast compared to the typical time-scales of the problem. Moreover, changing the size L affects both the communication time and the serial time, so it is not easy to find the L which maximize the communication time over the serial time.
As far as THIN nodes are concerned, by comparing the theoretical predicted MLUPs/sec with the real MLUPs/sec measured by the Jacobi solver, we see that the model is able to describe the features of weak scaling well.
More interesting is the GPU case, where we observe larger deviations. In particular, on one single GPU node we find a linear behavior up to 24 processes. As soon as hyper-threading is enabled we observe a discontinuous drop in the measured MLUPs/sec. After this drop in performance (possibly indicating a conflict between the two threads), we observe again a linear scaling behavior from 25 to 48 processes. See, for example, Figure 6.

TABLE IV: Single processor of a THIN node to estimate the serial time on one single core. $T_s^{THIN} = 15.3276$ s

N	(N_x, N_y, N_z)	k	$c(L,\vec{N})$ [MB]	$P^{com}(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1^{com}(L)}{P^{com}(L,\vec{N})}$	$P^{Jac}(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1^{Jac}(L)}{P^{Jac}(L,\vec{N})}$
1	(1,1,1)	0	0	112.74	1	112.74	1

TABLE V: Single processor of a GPU node to estimate the serial time on one single core. $T_s^{GPU} = 22.0729$ s

N	(N_x, N_y, N_z)	k	$c(L,\vec{N})$ [MB]	$P^{com}(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1^{com}(L)}{P^{com}(L,\vec{N})}$	$P^{Jac}(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1^{Jac}(L)}{P^{Jac}(L,\vec{N})}$
1	(1,1,1)	0	0	78.29	1	78.29	1

TABLE VI: Same THIN node pinning the MPI processes within the same socket
$T_s^{THIN} = 15.3276$ s, $T_\ell = 0.21$ μs, $B = 6155$ MB/s

N	(N_x, N_y, N_z)	k	$c(L,\vec{N})$ [MB]	$P^{com}(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1^{com}(L)}{P^{com}(L,\vec{N})}$	$P^{Jac}(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1^{Jac}(L)}{P^{Jac}(L,\vec{N})}$
4	(4, 1, 1)	2	46.08	450.722	1.001	447.622	1.007
4	(2, 2, 1)	4	92.16	450.492	1.001	447.390	1.008
8	(8, 1, 1)	2	46.08	901.443	1.001	887.270	1.016
8	(2, 2, 2)	6	138.24	900.526	1.002	879.848	1.025
12	(12, 1, 1)	2	46.08	1352.165	1.001	1311.020	1.032
12	(3, 2, 2)	6	138.24	1350.789	1.002	1310.580	1.032

TABLE VII: Same GPU node pinning the MPI processes within the same socket
$T_s^{GPU} = 22.0729$ s, $T_\ell = 0.21$ μs, $B = 6155$ MB/s

N	(N_x, N_y, N_z)	k	$c(L,\vec{N})$ [MB]	$P^{com}(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1^{com}(L)}{P^{com}(L,\vec{N})}$	$P^{Jac}(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1^{Jac}(L)}{P^{Jac}(L,\vec{N})}$
4	(4, 1, 1)	2	46.08	313.038	1.000	308.679	1.014
4	(2, 2, 1)	4	92.16	312.932	1.001	309.602	1.011
8	(8, 1, 1)	2	46.08	626.076	1.000	591.725	1.058
8	(2, 2, 2)	6	138.24	625.652	1.001	585.606	1.069
12	(12, 1, 1)	2	46.08	939.114	1.000	845.099	1.112
12	(3, 2, 2)	6	138.24	938.478	1.001	846.490	1.110

TABLE VIII: Same THIN node pinning the MPI processes across two sockets
$T_s^{THIN} = 15.3276$ s, $T_\ell = 0.40$ μs, $B = 5024$ MB/s

N	(N_x, N_y, N_z)	k	$c(L,\vec{N})$ [MB]	$P^{com}(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1^{com}(L)}{P^{com}(L,\vec{N})}$	$P^{Jac}(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1^{Jac}(L)}{P^{Jac}(L,\vec{N})}$
4	(4, 1, 1)	2	46.08	450.682	1.001	448.171	1.006
4	(2, 2, 1)	4	92.16	450.412	1.001	448.859	1.005
8	(8, 1, 1)	2	46.08	901.363	1.001	888.767	1.015
8	(2, 2, 2)	6	138.24	900.286	1.002	883.721	1.021
12	(12, 1, 1)	2	46.08	1352.045	1.001	1324.270	1.022
12	(3, 2, 2)	6	138.24	1350.429	1.002	1324.380	1.022

TABLE IX: Same GPU node pinning the MPI processes across two sockets
$T_s^{GPU} = 22.0729$ s, $T_\ell = 0.42$ μs, $B = 5007$ MB/s

N	(N_x, N_y, N_z)	k	$c(L,\vec{N})$ [MB]	$P^{com}(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1^{com}(L)}{P^{com}(L,\vec{N})}$	$P^{Jac}(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1^{Jac}(L)}{P^{Jac}(L,\vec{N})}$
4	(4, 1, 1)	2	46.08	313.014	1.000	309.776	1.011
4	(2, 2, 1)	4	92.16	312.883	1.001	310.721	1.008
8	(8, 1, 1)	2	46.08	626.027	1.000	615.174	1.018
8	(2, 2, 2)	6	138.24	625.506	1.001	611.147	1.025
12	(12, 1, 1)	2	46.08	939.041	1.000	895.448	1.049
12	(3, 2, 2)	6	138.24	938.259	1.001	895.091	1.050

TABLE X: Two THIN nodes
$T_s^{THIN} = 15.3276$ s, $T_\ell = 0.98$ μs, $B = 12185$ MB/s

N	(N_x, N_y, N_z)	k	$c(L,\vec{N})$ [MB]	$P^{com}(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1^{com}(L)}{P^{com}(L,\vec{N})}$	$P^{Jac}(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1^{Jac}(L)}{P^{Jac}(L,\vec{N})}$
12	(12, 1, 1)	2	46.08	1352.520	1.000	1324.530	1.021
12	(3, 2, 2)	6	138.24	1351.853	1.001	1318.910	1.026
24	(24, 1, 1)	2	46.08	2705.040	1.000	2594.560	1.043
24	(4, 3, 2)	6	138.24	2703.705	1.001	2633.940	1.027
32	(32, 1, 1)	2	46.08	3606.719	1.000	3363.330	1.073
32	(4, 4, 2)	6	138.24	3604.940	1.001	3498.550	1.031
40	(40, 1, 1)	2	46.08	4508.399	1.000	4148.880	1.087
40	(5, 4, 2)	6	138.24	4506.175	1.001	4361.020	1.034
48	(48, 1, 1)	2	46.08	5410.079	1.000	4795.320	1.128
48	(4, 4, 3)	6	138.24	5407.410	1.001	5080.640	1.065

TABLE XI: Same GPU node where hyper-threading is enabled mapping by socket.
$T_s^{GPU} = 22.0729$ s, $T_\ell = 0.42$ μs, $B = 5007$ MB/s

N	(N_x, N_y, N_z)	k	$c(L,\vec{N})$ [MB]	$P^{com}(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1^{com}(L)}{P^{com}(L,\vec{N})}$	$P^{Jac}(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1^{Jac}(L)}{P^{Jac}(L,\vec{N})}$
12	(12, 1, 1)	2	46.08	939.041	1.000	895.448	1.049
12	(3, 2, 2)	6	138.24	938.259	1.001	895.091	1.050
24	(24, 1, 1)	2	46.08	1878.082	1.000	1656.790	1.134
24	(4, 3, 2)	6	138.24	1876.518	1.001	1686.510	1.114
25	(25, 1, 1)	2	46.08	1956.335	1.000	1366.520	1.432
25	(5, 5, 1)	4	92.16	1955.520	1.001	1389.400	1.409
32	(32, 1, 1)	2	46.08	2504.109	1.000	1675.980	1.495
32	(4, 4, 2)	6	138.24	2502.023	1.001	1742.530	1.438
40	(40, 1, 1)	2	46.08	3130.137	1.000	2042.430	1.533
40	(5, 4, 2)	6	138.24	3127.529	1.001	2131.680	1.469
48	(48, 1, 1)	2	46.08	3756.164	1.000	2341.970	1.605
48	(4, 4, 3)	6	138.24	3753.035	1.001	2499.760	1.503

IV. ACKNOWLEDGMENT

MC would like to thank DSSC and MHPC fellow students for helpful conversations.

---

[1] Hager, G. and Wellein, G. (2010). *Introduction to High Performance Computing for Scientists and Engineers*. Chapman & Hall/CRC Computational Science. CRC Press. ISBN: 9781439811931

[2] Hager, Georg & Stengel, Holger & Zeiser, Thomas & Wellein, Gerhard. (2008). *RZBENCH: Performance Evaluation of Current HPC Architectures Using Low-Level and Application Benchmarks*. 10.1007/978-3-540-69182-2_39.

[3] Hager, G. & Wellein, G. (2007). *A performance model for the IMB multi-mode PingPong benchmark.*