Report Assignment 1

GitHub page

https://github.com/thomasverardo/HPC_Assignment1

Verardo Thomas

30/12/2021

SECTION 1 (Ring)

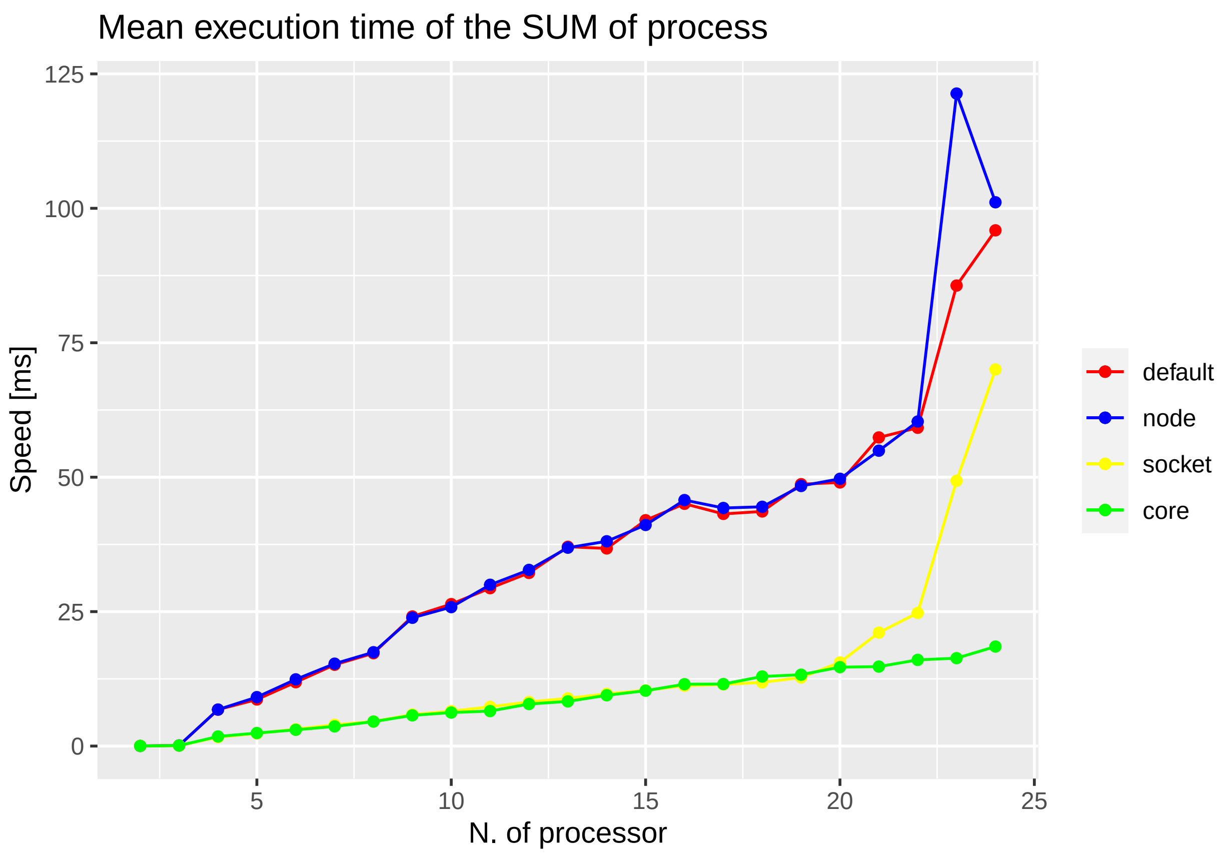
The first section was focused on the implementation on a ring throw processor code. The ring is implemented with a 1D virtual topology and using non-blocking communication.

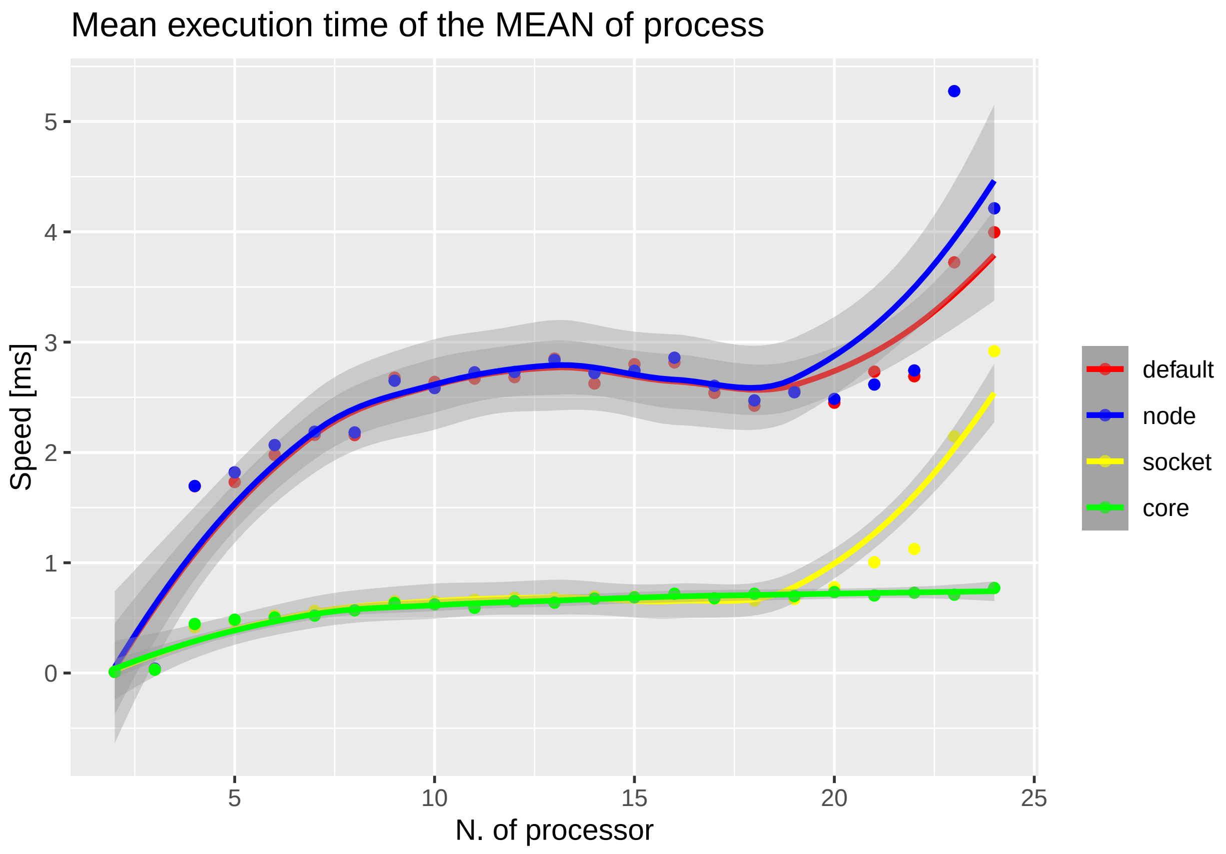
Once implemented with the OpenMPI library, I runned the test throw different processor of the THIN node of ORFEO. The maximum number of processor in one thin node is 24 processor. I have taken notes of the runtime of all sent and received messagges. To compare different number of total processor, I did the sum and the average of the time to complete the ring execution of one processor. Then, for each different number of processor, I repeated 100 times the execution of the ring and I computed the mean.

Plot results

I produce the plot in one thin node between different processor, to compare the difference mapping: defualt, node, socket or core. In the first plot it's possible to see that the default method is equal to the node method. Apart for some outliers, the time to compute increase linearly with increasing number of processes.

In the second plot, you can see the difference from using a difference number of processes, but instead to do the sum over the different time to complete a ring for each process, the processes are averaged. Then, I create a model to better representing the data. You can see that the line representing the mapping by core is constant over the number of processes. In the other three lines, if the number of processes is 23 or 24, the speed is bigger than other measurements.


Mean execution time of the SUM of process
Speed [ms]
N. of processor
125
100
75
50
25
0
5
10
15
20
25
default
node
socket
core

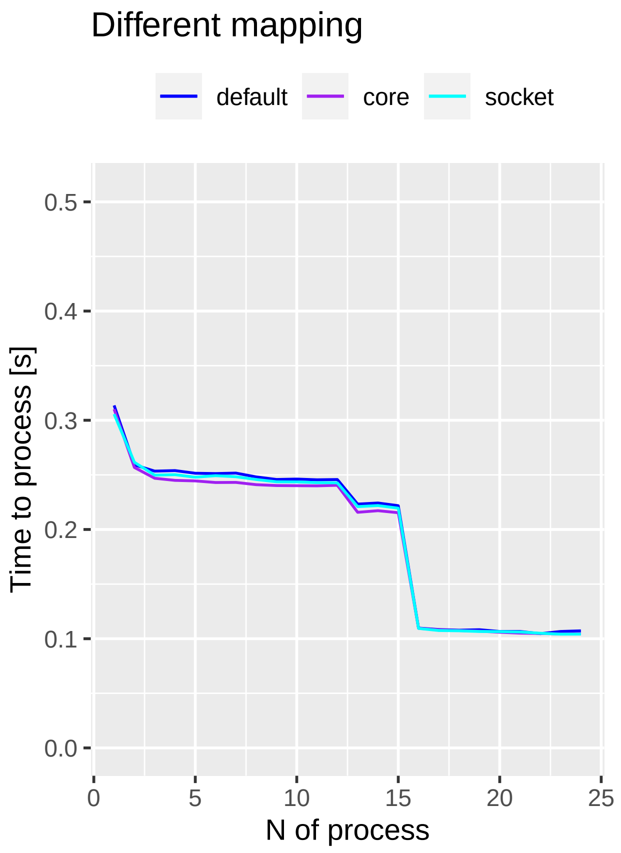

Mean execution time of the MEAN of process
Speed [ms]
N. of processor
5
4
3
2
1
0
5
10
15
20
25
default
node
socket
core

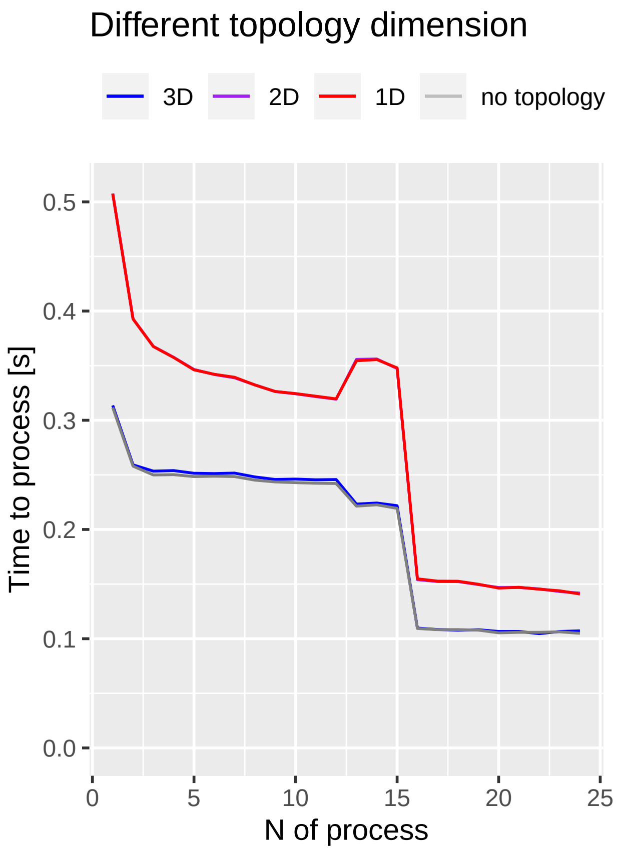
SECTION1 (Matrix)

The matrix is initialized with random number and, after building the topology, that is always (s, 1, 1) as s the numbers of processes, the matrix is first divided and then sent to all the processes. So, each process computed a part of the matrix, that is exactly the shape of the matrix ($x * y * z$) divided by the number of processes.

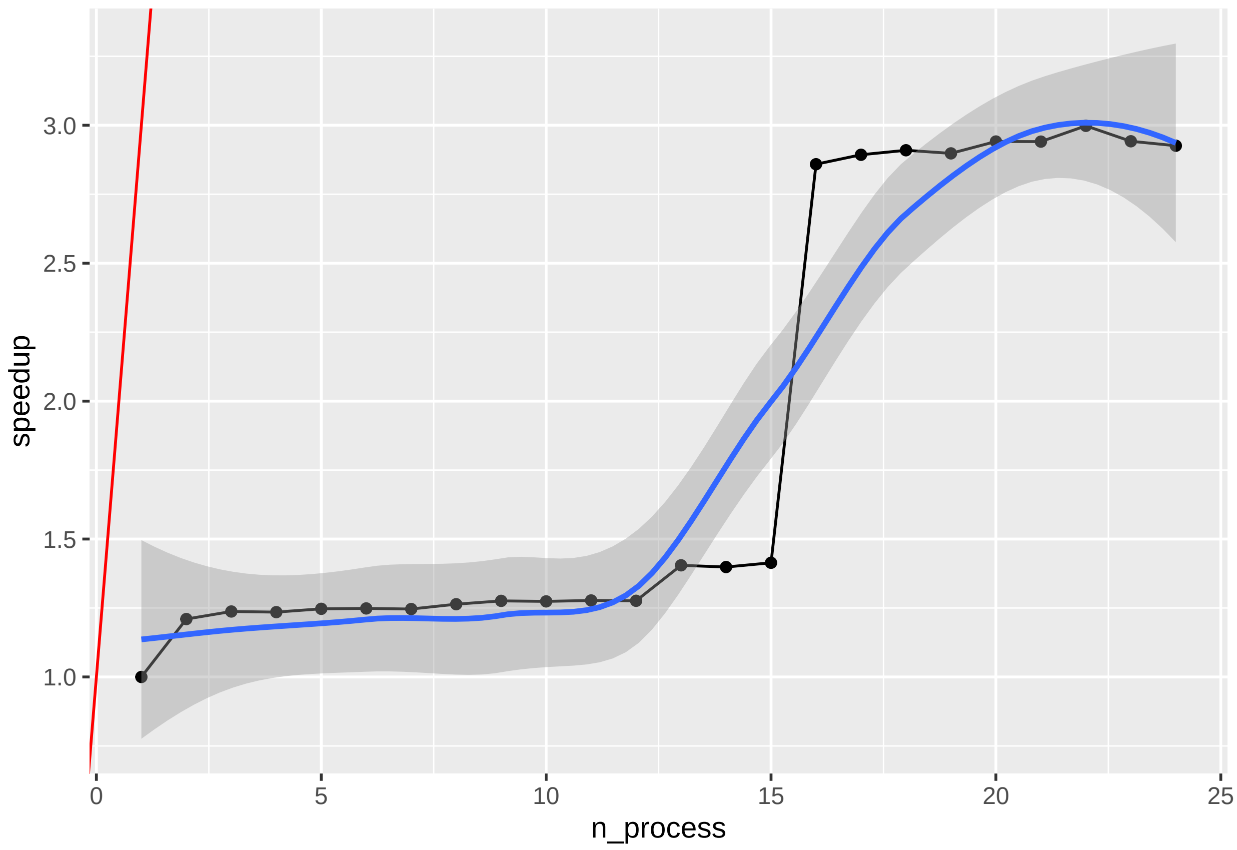
I produce different plots: the first one is not very useful and it represent the different shapes of the matrix in relation to the number of processes (the plot is in the LINK). In that graph, we can see that, if we use different shapes, that are 2400x100x100 (red), 1200x200x100 (orange) and 800x300x100 (yellow), the time to sum the matrix is always the same, even using multiple processes.

Here, there are two plots. The first is the plot with always the same matrix (1200x200x100), with the same topology (3D) but with different mapping and we can see the result doesn't change. In fact, the time to process is always similar. We note that for the first 15 processes executed in parallel, the time is quiet constant, but after 16 processes in parallel, the time to compute decrease drastically and remains constant until the end.





After this, I computed the time to do the matrix-matrix sum with the same mapping and the same shape (1200x200x100) but with different topology, expressed in the second plot. The 2D and 1D topologies are the same, even because the matrix is a 3D array and the two the topologies are built with the same Nx, that is the number of processes ($[size]$ for 1D and $[size, 1]$ for 2D, as size the number of processes). We can see also that built a topology is useless because it has the same performance of the matrix without any topology.



Furthermore, I calculate the speedup for the matrix with the shape of 1200x200x100 with the default configuration and with the 3D topology. To calculated the speedup, I used the formula:

$$S_p = \frac{T_1}{T_p}$$

Where T_1 is the best time with one single process and T_p is the time to calculated the matrix with p processes. Then I calculated the efficiency, that is *speedup/nřprocesses* and it's a value, that varies between zero and one, describe a measure of how well our parallel algorithm uses processors. As seen from the red line that represent the best theoretical parallel efficiency, the efficiency is not really good.

SECTION 2

In this section, the main purpose was to estimate the latency and the bandwidth of all available combinations of topologies and networks on ORFEO computational nodes. For finding the latency and the bandwidth with the OpenMPI library, I writed a bash script that first submitted a job for running the code in the CPU or in the GPU, and than runned it for 10 times for each topologies. In this way, I can take the mean of the results of each topologies, so to have better data to study.

OpenMPI

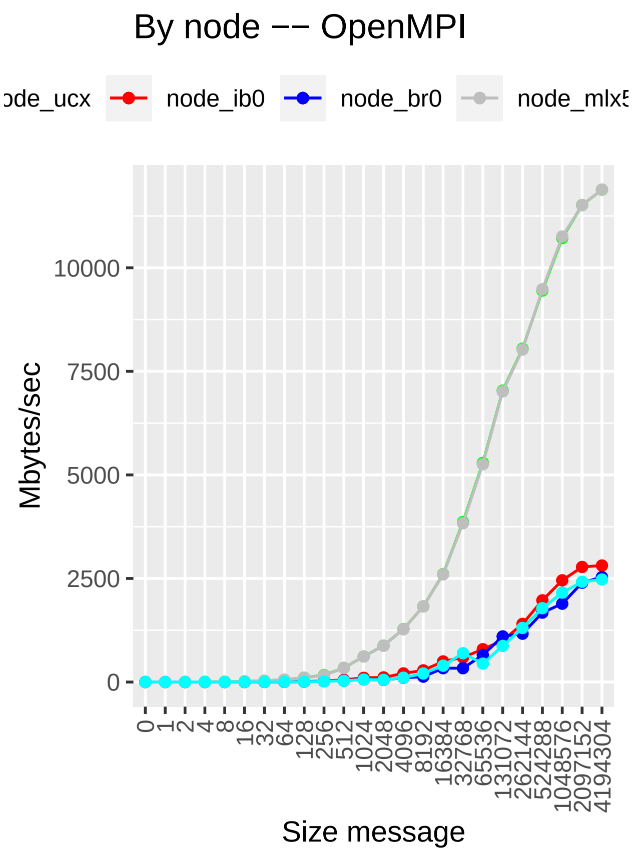
First of all, I did a benchmark with OpenMPI of some topologies that I found. The topologies tested in PingPong for across 2 nodes that I found were these:

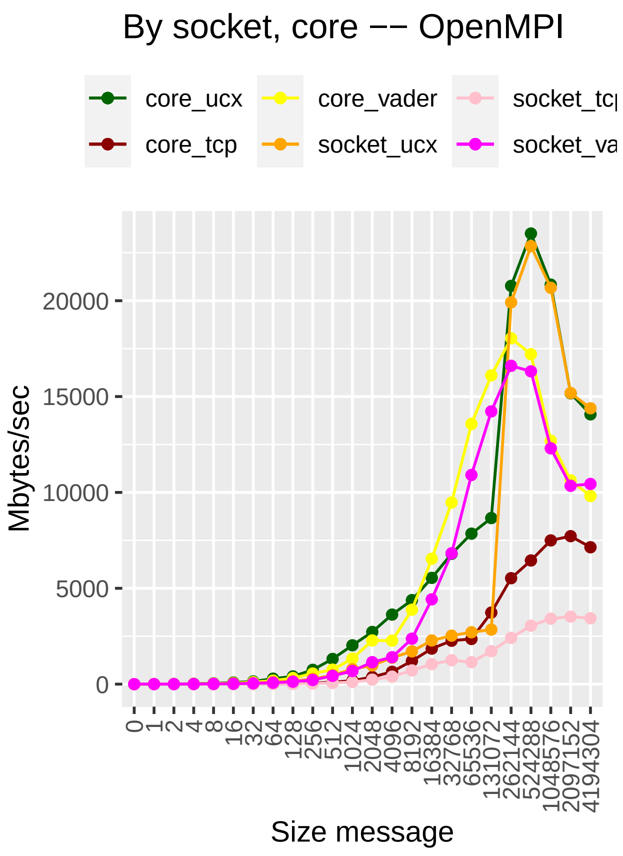
- *node_ucx*: Use the UCX communication library over InfiniBand using the default configuration
- *node_ib0*: Use the InfiniBand adabter over the TCP/IP protocol
- *node_br0*: Use UCX with Ethernet (and so also the protocol TCP/IP)

- *node_mlx5_0_1*: Use UCX but with the configuration 0 of the interface number (InfiniBand, that is the default one)
- *node_tcp*: Use ob1 communication library, that is a multi-device and multi-rail engine

In the first graph we can see the difference between the communication protocol and their interfaces. This plots are made with Thin nodes using the OpenMPI library. In particolar, *node_ucx* and *node_mlx5_0*, also with different package size, have equal bandwidth; this because *node_mlx5_0* describes the default configuration, that is *node_ucx*. In the graph, with message size very high, you can see the difference between the default configuration and the new configuration with *ib0*, *br0* and the last one that uses *ob1* as point-to-point messaging layer and *TCP* as byte transfer layer.
Then I did the same benchmark across 2 socket and across 2 nodes. The results are in the second graph. In both, I first used the default configuration and then I tried to use a different PML. In fact, I used *ob1* as PML and first I used *TCP* as BTL and then I used *Vader* (that used shared memory). In this graph you can see that send messages across 2 core is faster than sending messages across 2 socket. This because cores are within the same socket and are physically closer.

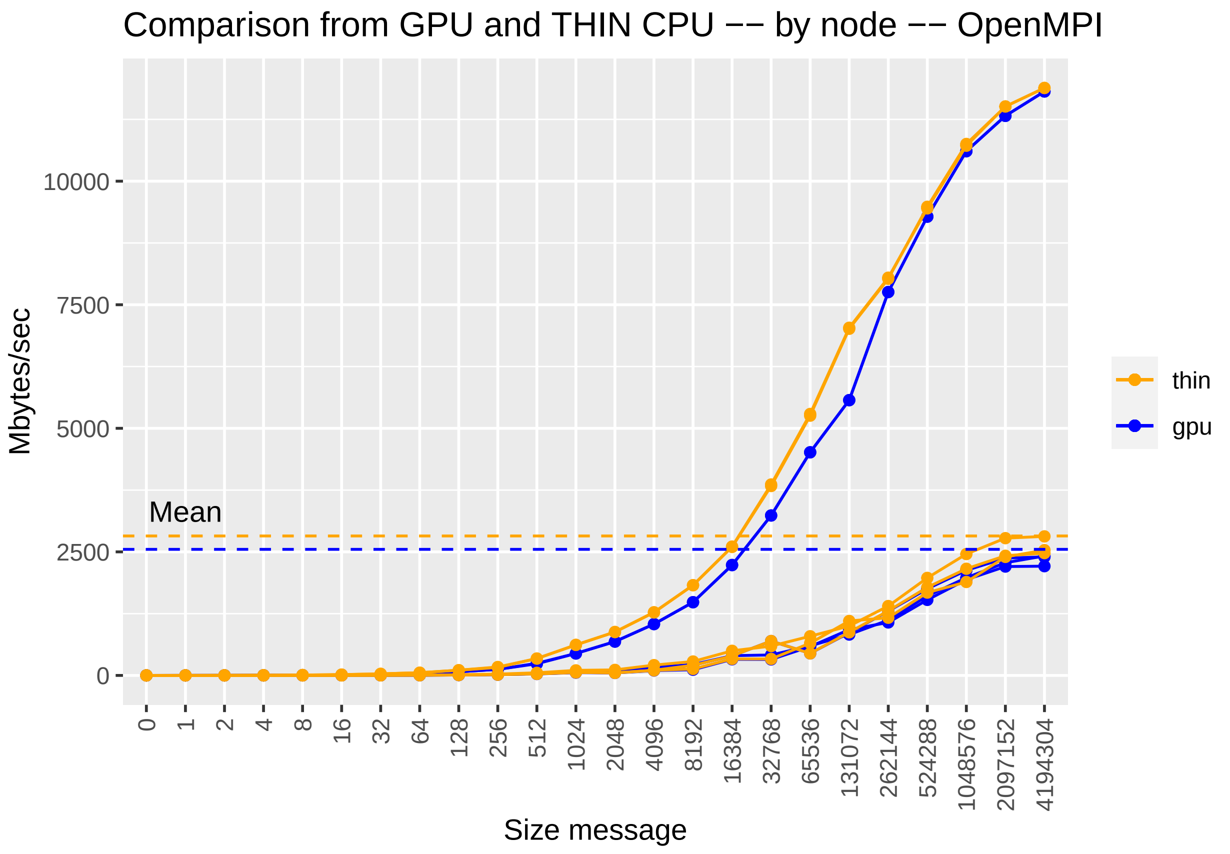
The main difference between the chart of the nodes and the chart of the cores and sockets is that in the first one, the line continue to rise without never discending and so the bandwith is always getting bigger as the size memory. In my opinion, in the core and socket chart, there is this descending curve because the L2 cache fills up if the message size is grater than 1024 KB (1048576 B) and the message goes in the L3 cache, that can save up to 14080 KB.





Intead, in the nodes chart, I think that is different from the sockets and cores chart because Infiniband provides native support for RDAM (Remote Direct Memory Access). In fact, integral to RDMA is the concept of zero-copy networking, which makes it possible to read data directly from the main memory of one computer and write that data directly to the main memory of another computer. RDMA data transfers bypass the kernel networking stack in both computers, so it doesn't have to pack and unpack the message, improving network performance. As a result, the conversation between the two systems will complete much quicker than comparable non-RDMA networked systems.

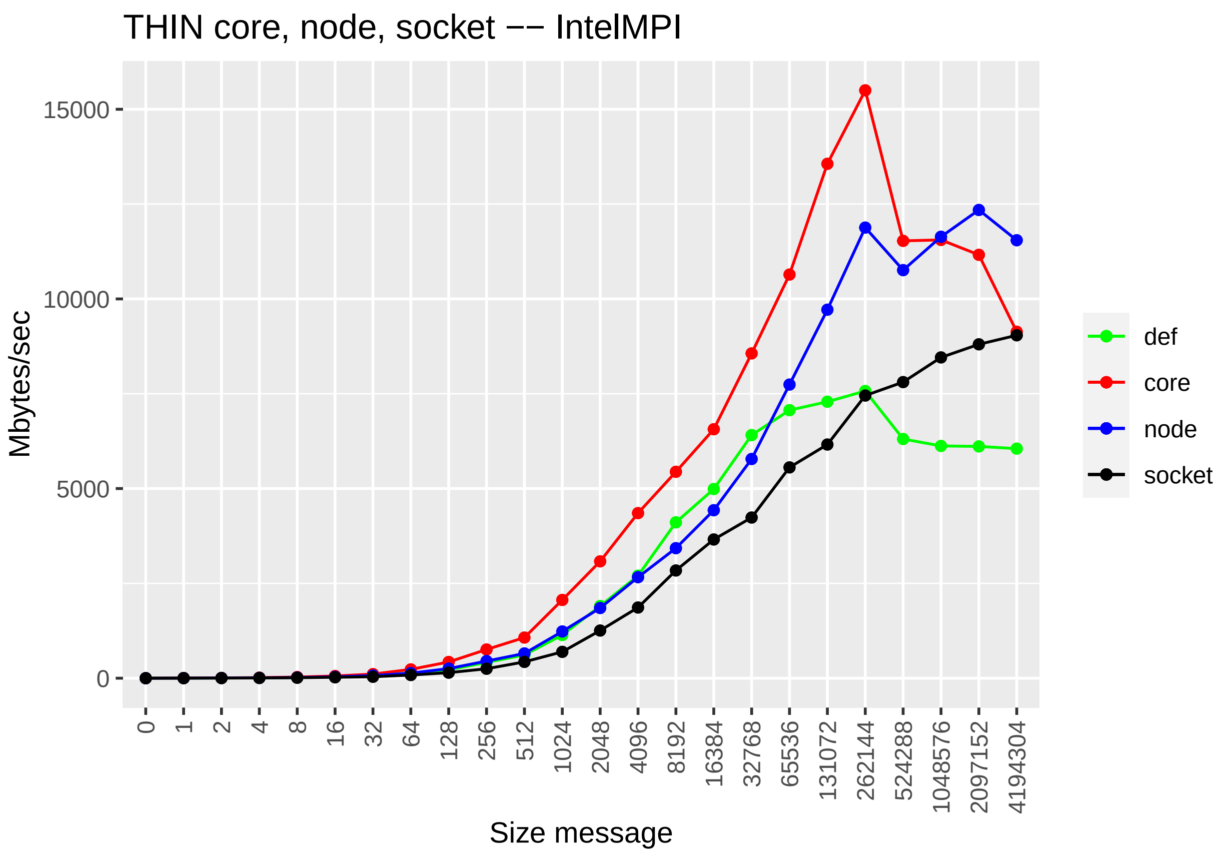
Then, I did the charts also in the GPU nodes (see the github repository).



Here there is the comparison between the benchmark in the Thin CPU nodes or in the GPU nodes. You can see from this chart and from the mean, that the Thin node is faster then the GPU node.

IntelMPI

After doing all charts of the PingPong benchmark with OpenMPI for the topologies that I mentioned before across two nodes, two sockets and two core in the Thin CPU nodes and in GPU nodes of ORFEO, I taken new data with the IntelMPI library. Therefore, I ran the PingPong code first with the defaul configuration, and than across two nodes, across 2 sockets and in the end across two core.



Also with the IntelMPI library, it's easy to see that the line that represent the PingPong across 2 core is the only line that at one moment no longer grows, without considering the default line.

Model fitting

Finally, I compered the resulting estimated latency and bandwidth parameters against the one provided by a least-square fitting model. I used the model provied in class that described the total transfer time of a message:

$$t_{comm} = \lambda + \frac{(Size\ of\ message)}{b_{network}}$$

To fit the model, first I divided my data in two parts and I calculated the linear model for the first half and another linear model for the second half. Then I take the slope from the first model, that is similar to the latency and then I take the angular coefficient from the second model, that is similar to the $1/bandwidth$. After that, I used the formula above to estimate my latency and that I divided the size message with this latency estimation.

SECTION 3

In the section 3, we have to compile and run the Jacobi program and test the performance in the theoretical and practical way. To test the first one and so to predict the Jacobi model performance, it's used this model:

$$P(L, N) = \frac{L^3 * N}{T_s + T_c}[MLUP/s]$$

With L as total time on a single process, N as the number of process, Ts as a time that is constant and it's estimated, Tc as the communication time per second, that is:

$$Tc(L, N) = \frac{c(L, N)}{B} + 4kT_l[seconds]$$

B is the bandwidth, T_l is the latency (λ) and c(L,N) is the message size:

$$c(L,N) = L^2 * k * 2 * 2 * 8[byte]$$

$L^2 * k$ must be multiplied for 8 because the message is a double, for 2 because we are in a bidirectional case and another time for 2 for the positive and negative direction.

Model prediction

There are several domain decomposition that can be done, but, for the purposes of the exercise and the performance, it's for little use to calculate all the decomposition performance, therefore only those that are studied best have been calculated.

All models are calculated with L = 700.

Table 1: Model thin, mapping by node

N_process	Nx	Ny	Nz	k	C	Tc	Model_perf	Real_perf	Elapsed_time	Perf_ratio
1	1	1	1	2	0.000	0.000	114.716	112.194	54.26	1.00000
12	12	1	1	2	7.477	0.001	1376.300	1334.310	54.80	1.00021
12	4	3	1	4	14.954	0.001	1376.011	1334.824	54.75	1.00042
12	3	2	2	6	22.430	0.002	1375.722	1342.339	54.57	1.00063
12	6	2	1	4	14.954	0.001	1376.011	1341.984	54.58	1.00042
24	24	1	1	2	7.477	0.001	2752.599	2680.570	54.80	1.00021
24	12	2	1	4	14.954	0.001	2752.021	2683.972	54.75	1.00042
24	8	3	1	4	14.954	0.001	2752.021	2678.966	54.78	1.00042
24	6	4	1	4	14.954	0.001	2752.021	2676.793	54.79	1.00042
24	6	2	2	6	22.430	0.002	2751.444	2681.906	54.82	1.00063
24	4	3	2	6	22.430	0.002	2751.444	2674.086	54.85	1.00063
48	48	1	1	2	7.477	0.001	5505.198	5260.938	56.39	1.00021
48	24	2	1	4	14.954	0.001	5504.043	5247.754	56.51	1.00042
48	12	4	1	4	14.954	0.001	5504.043	5249.959	56.58	1.00042
48	12	2	2	6	22.430	0.002	5502.887	5215.275	56.73	1.00063
48	8	6	1	4	14.954	0.001	5504.043	5222.120	56.67	1.00042
48	6	4	2	6	22.430	0.002	5502.887	5244.173	56.52	1.00063

Table 2: Model thin, mapping by socket

N_process	Nx	Ny	Nz	k	C	Tc	Model_perf	Real_perf	Elapsed_time	Perf_ratio
1	1	1	1	2	0.000	0.000	114.716	112.316	54.24	1.000000
4	4	1	1	2	7.477	0.001	458.655	448.578	54.35	1.000452
4	2	2	1	4	14.954	0.003	458.448	448.307	54.36	1.000905
8	8	1	1	2	7.477	0.001	917.311	888.956	54.74	1.000452
8	4	2	1	4	14.954	0.003	916.896	895.186	54.49	1.000905
8	2	2	2	6	22.430	0.004	916.482	895.898	54.47	1.001357
12	12	1	1	2	7.477	0.001	1375.966	1332.876	54.83	1.000452
12	4	3	1	4	14.954	0.003	1375.344	1336.472	54.84	1.000905
12	3	2	2	6	22.430	0.004	1374.723	1342.513	54.63	1.001357
12	6	2	1	4	14.954	0.003	1375.344	1342.499	54.62	1.000905

Table 3: Model thin, mapping by core

N_process	Nx	Ny	Nz	k	C	Tc	Model_perf	Real_perf	Elapsed_time	Perf_ratio
1	1	1	1	2	0.000	0.000	114.716	112.262	54.26	1.00000
4	4	1	1	2	7.477	0.001	458.683	448.071	54.44	1.00039
4	2	2	1	4	14.954	0.002	458.503	448.982	54.37	1.00079
8	8	1	1	2	7.477	0.001	917.366	894.611	54.59	1.00039
8	4	2	1	4	14.954	0.002	917.006	896.996	54.61	1.00079
8	2	2	2	6	22.430	0.004	916.646	894.367	54.61	1.00118
12	12	1	1	2	7.477	0.001	1376.048	1329.530	55.27	1.00039
12	4	3	1	4	14.954	0.002	1375.509	1315.655	55.70	1.00079
12	3	2	2	6	22.430	0.004	1374.970	1324.698	55.42	1.00118
12	6	2	1	4	14.954	0.002	1375.509	1323.848	55.46	1.00079

Data are calculated with the following parameters:

- By node: $\lambda = 1.02$ [usec], B = 11946 [MB/s]
- By socket: $\lambda = 0.49$ [usec], B = 5530 [MB/s]
- By core: $\lambda = 0.22$ [usec], B = 6372 [MB/s]

The model performed seems accurate, but you can notice that the estimated performance is better with mapping by node and socket and worse with core. In the expected theoretical model this data should be different, that us the mapping by core better than node.

Table 4: Model gpu, mapping by socket

N_process	Nx	Ny	Nz	k	C	Tc	Model_perf	Real_perf	Elapsed_time	Perf_ratio
1	1	1	1	2	0.000	0.000	79.453	77.285	75.47	1.000000
12	12	1	1	2	7.477	0.001	953.138	900.188	83.73	1.000317
12	4	3	1	4	14.954	0.003	952.836	900.329	83.15	1.000634
12	3	2	2	6	22.430	0.004	952.535	893.078	83.52	1.000950
12	6	2	1	4	14.954	0.003	952.836	899.948	83.18	1.000634
24	24	1	1	2	7.477	0.001	1906.276	1698.968	88.97	1.000317
24	12	2	1	4	14.954	0.003	1905.672	1699.918	89.11	1.000634
24	8	3	1	4	14.954	0.003	1905.672	1699.285	88.88	1.000634
24	6	4	1	4	14.954	0.003	1905.672	1698.867	88.94	1.000634
24	6	2	2	6	22.430	0.004	1905.069	1699.699	88.86	1.000950
24	4	3	2	6	22.430	0.004	1905.069	1699.090	88.90	1.000950
48	48	1	1	2	7.477	0.001	3812.552	2528.730	115.50	1.000317
48	24	2	1	4	14.954	0.003	3811.345	2542.104	114.50	1.000634
48	12	4	1	4	14.954	0.003	3811.345	2531.668	115.00	1.000634
48	12	2	2	6	22.430	0.004	3810.138	2546.495	114.30	1.000950
48	8	6	1	4	14.954	0.003	3811.345	2531.779	114.60	1.000634
48	6	4	2	6	22.430	0.004	3810.138	2542.649	114.50	1.000950

Table 5: Model gpu, mapping by core

N_process	Nx	Ny	Nz	k	C	Tc	Model_perf	Real_perf	Elapsed_time	Perf_ratio
1	1	1	1	2	0.000	0.000	79.453	77.285	75.47	1.000000
12	12	1	1	2	7.477	0.001	953.178	850.527	88.24	1.000275
12	4	3	1	4	14.954	0.002	952.916	849.571	88.14	1.000550
12	3	2	2	6	22.430	0.004	952.654	849.634	88.03	1.000825

N_process	Nx	Ny	Nz	k	C	Tc	Model_perf	Real_perf	Elapsed_time	Perf_ratio
12	6	2	1	4	14.954	0.002	952.916	850.352	88.10	1.000550
24	24	1	1	2	7.477	0.001	1906.355	1690.635	89.12	1.000275
24	12	2	1	4	14.954	0.002	1905.831	1700.242	88.86	1.000550
24	8	3	1	4	14.954	0.002	1905.831	1699.032	88.91	1.000550
24	6	4	1	4	14.954	0.002	1905.831	1700.316	88.96	1.000550
24	6	2	2	6	22.430	0.004	1905.308	1698.990	88.89	1.000825
24	4	3	2	6	22.430	0.004	1905.308	1699.548	88.89	1.000825
48	48	1	1	2	7.477	0.001	3812.711	2538.199	114.90	1.000275
48	24	2	1	4	14.954	0.002	3811.663	2534.727	115.10	1.000550
48	12	4	1	4	14.954	0.002	3811.663	2523.002	115.40	1.000550
48	12	2	2	6	22.430	0.004	3810.615	2550.307	114.40	1.000825
48	8	6	1	4	14.954	0.002	3811.663	2549.906	114.50	1.000550
48	6	4	2	6	22.430	0.004	3810.615	2527.282	115.30	1.000825

Unlike the CPU, in the GPU performance we can see that the estimated model performance of the mapping by core is sligthy better than the mapping by socket.