Foundations of High Performance Computing
Assignment 1

Berti Michele

December 30, 2021

Exercise 1

1.1 Ring exercise

Implementation
The implementation is basically a for loop in which each node:

1. sends a non blocking message to left processor
2. receives the message from right
3. sends a non blocking message to right processor
4. receives the message from left
5. waits for completion of messages

A barrier is then set to take the measurement of the time of process receiving the last message as last. In order to obtain reasonable measurements of the execution time of the program, the messages do a large number of laps of the ring (100 000).

Performance model
Let's consider the time of a single communication:

$$T_{comm} = \lambda + \frac{size}{bandwidth}$$

due to the fact all the messages sent are just an int, the second term in the sum is negligible in first approximation and the time for a communication is just

$$T_{comm} \approx \lambda$$

In the for loop we have basically 2 messages passed from one processor to his neighbours and the number of iteration of the for cycle is proportional to the number of cores used we have that

$$T(P) = T_{init\ and\ fin} + 2 \times N\_LAPS \times P \times T_{comm} \approx T_{init\ and\ fin} + 2 \times N\_LAPS \times P \times \lambda$$

The execution time of the program is, in first approximation just the communication time

since the other operations are negligible compared to the communication one.
Given that the amount of work in the for loop is orders of magnitude bigger, a very simple model could be sufficient for this analysis.

$$T(P) \approx 2 \times N\_LAPS \times P \times \lambda$$

Actually non blocking routines have been used so, some latency of the messages might be hidden. Moreover other factor may varies a the average execution time of the for loop (e.g. the other operation evaluated, OS process, ...) What we can safely expect is that the time grows proportionally with the number of processes:

$$T(P) = k \times P \times \lambda$$

for some constant k.
By including λ in the constant we obtain:

$$T(P) = k \times P$$

One last thing to consider to have a better understanding of the process is the fact that the communication latency varies according to the the physical core in which the process are running: spatial locality brings lower latency.

For this reason we can measure 3 different proportionality :

1. k_1 : $P <= 12$ Processes run on the same socket
2. k_2 : $12 < P <= 24$ Processes run on the two cores within the same node
3. k_3 : $24 < P <= 48$ Processes run on different nodes

This phenomena has been investigated by adopting a core mapping policy. Below a plot of the execution time (normalized by the number of processes) is reported against the number of cores used.

The data, except for some noise, supports the model described above.
It is also possible to spot that anomalous values are obtained when only two cores are in a different "place" (new socket, new node).
Finally it is worth to mention that the theoretical model, that considers the just the time of the 2 latencies i.e. $k = 2\lambda$ is a decent, even if a little underestimating, model when processes are running in different sockets or nodes.
The best performance obtained using only 2 nodes is basically due to the fact the ring basically reduces to a ping ping with the obvious gain in execution time.

1.2 Matrix sum exercise

The exercise is ill posed in fact due to the fact that the problem of summing two matrices does not need any halo layer and no communication between processes is needed, the performance of the computation is independent from the topology used.
For communication instead the matrix is stored in a heap has a unidimensdional array and the fastest way to send it to the other processors is to do a 1D domain decomposition across the third dimension so that the buffered message is contiguous in memory.
For the same reason a 2D domain decomposition (across dimensions 2 and 3) would be more

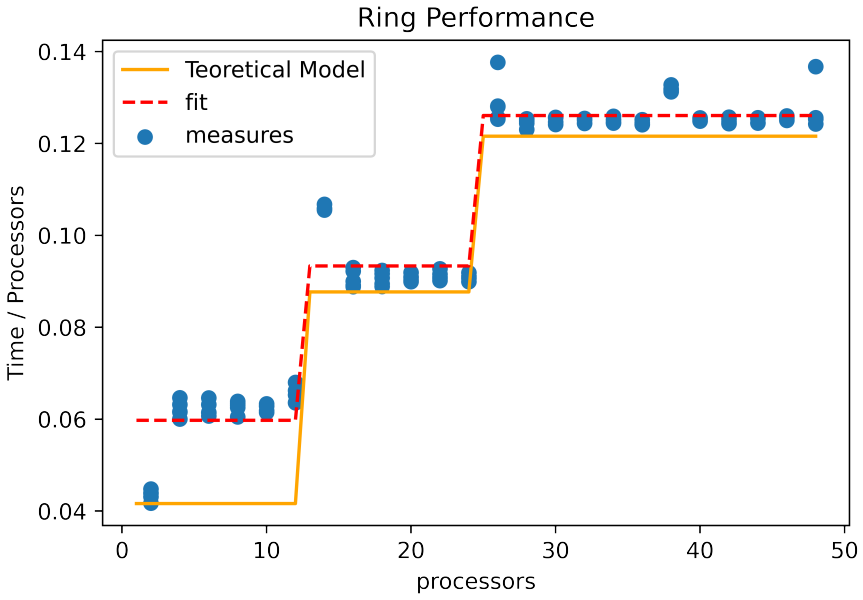

Figure 1: Time per processor number measured for the ring

effective than the 3d one, but less-performing than the 1D.
A code of a parallel sum of a matrix with a 1D domain decomposition along the 3rd axis is submitted.

Experimental Procedure
In order to have an int on how the code scales, different runs have been submitted to a thin node of the cluster, each execution was characterized by a growing number of core used.
To measure the execution time the usr/bin/time command has been used. In particular the elapsed time was taken.
The size of matrix used is 2400 x 1000x 100.

Results
In figure 2 the scalability curve of the code is presented. It can be seen that it doesn't scale well.
This fact is reasonable even if the communication part is the minimum possible, due to the fact the sizes of the problem are ridiculously small.
We have in fact that the serial part (data generation and communication) is predominant with respect to the computation part.

Exercise 2

The aim of this exercise is to obtain an estimation of the communication properties of the different network available on the cluster taking into account all possible combination of hardware and software layers.

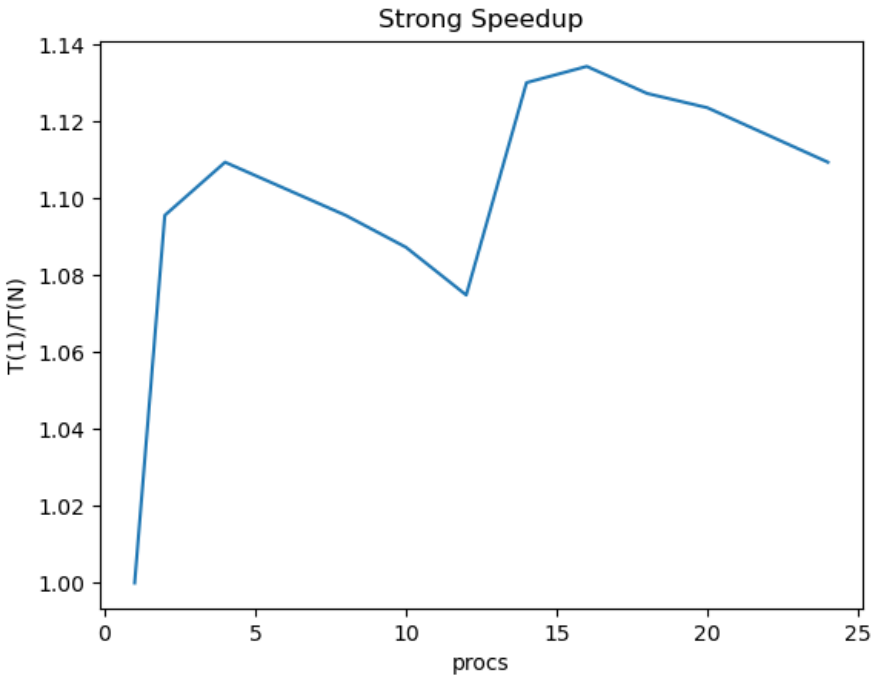

Figure 2: matrix strong scalability curve

Resources used
In this exercise the Intel®Ping Pong benchmark has been used.
Its implementation basically consists in sending a message of x bytes to a different process (ping) and then the process that received the messages sends it back to the original sender(pong). Then the time measured to perform this operation is divided by 2 in order to have an estimation of the time needed to send a single message of x Bytes.

The performance model
As stated before the time needed for a communication of x Bytes as follow:

$$T_{comm} = \lambda + \frac{x}{bandwidth}$$

As the result of the analysis will show, this is just a simple model because different factor may influence this value.

The measures
The measurements have been performed using using the 3 possible binding policies (mapping by core, by socket and by node).
Here is the list of different resources used for the benchmarking:

Pml:

- UCX
- Ob1

Ob1 Btl :

- Vader (within the same node)
- Tcp

PCI Networks:

- 25 Gbit Ethernet Network
- 100 Gbit Infiniband Network

The benchmark was also compiled with the Intel ®compiler and tested with with UCX and Infiniband network with the different binding policies.

Experimental Procedure
For each combination a series of 10 different consecutive measurements have been taken in order to take some statistics.
For the one executed on a thin node, all the node was reserved for the measurement, in order to reduce the possible bias introduced by other processes running and possibly communicating. This was not possible on the gpu nodes due to the workload present on the gpu queue.
With the statistics observed an ordinary least squares procedure has been used to have an estimation of latency and lambda for every combination of network and protocols.

Experimental Results
In the following table are reported the results obtained.

Core mapping

	latency (μs)	Bandwidth(MB/s)
ucx ib openmpi	0.2042	6445.871801
ucx ib intel	0.239	5427.428389
ob1 tcp openmpi	4.9322	5341.114703
ob1 vader openmpi	0.288	4497.881158

Socket mapping

	latency (μs)	Bandwidth(Mb/s)
ucx ib openmpi	0.412	5660.741341
ucx ib intel	0.4382	4217.057497
ob1 tcp openmpi	7.8758	3292.231738
ob1 vader openmpi	0.661	3945.640058

Node mapping

	latency (μs)	Bandwidth(MB/s)
ucx ib openmpi	1.0076	12169.471313
ucx ib intel	1.1018	12221.877606
ob1 tcp openmpi	15.598	2651.672488
ucx br0 openmpi	15.8256	2484.760864
ucx ib0 openmpi	9.6446	2879.878963

The difference in performance for the measurements within the same socket can be explained by the different performance of the protocols.
It can be spotted that ob1 with tcp has a huge latency with respect to the others, while the Remote Direct Memory Access of UCX protocol is brings to the fastest communication speed due to the lower number communication layer present with respect to TCP/IP protocol.

Considering that the 100Gbit/s infiniband newtwork corresponds to 12.5GB/s we have a actual ping pong performace of roughly 96% with infiniband across nodes, which is optimal considering that the encoding ratio $\frac{64}{66}$ is approximately 0.97.

To have a better representation of the numbers presented in the table the following barplot in figure 3 was produced.

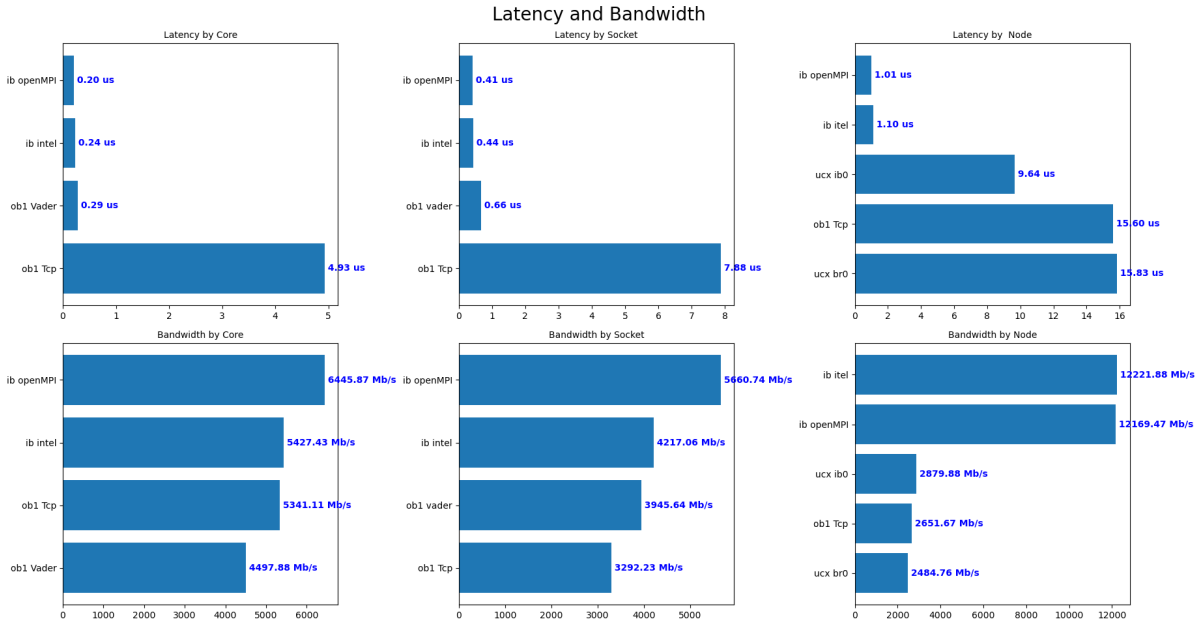

Figure 3: Summary of lantancies and bandwidth measured on thin node

We observe that the latency of the by core mapping is roughly the double of the one obtained with the by socket mapping, for every type of measure.

Moreover we observe a remarkable difference between the different Mpi implementations when the processes run within the same node.
A better insight about this phenomena can be obtained by analysing the plots in figure 4 that represents the actual band-width when the size of the message increases.
We see that after a certain size, the bandwidth for messages exchanged between processes running within the same node encounter some degradation, probably due to the fact the buffered message doesn't fit in the cache.
The degradation starts in fact when the size is between 0.1MB and 1MB and the L2 cache level has a size of 1MB.

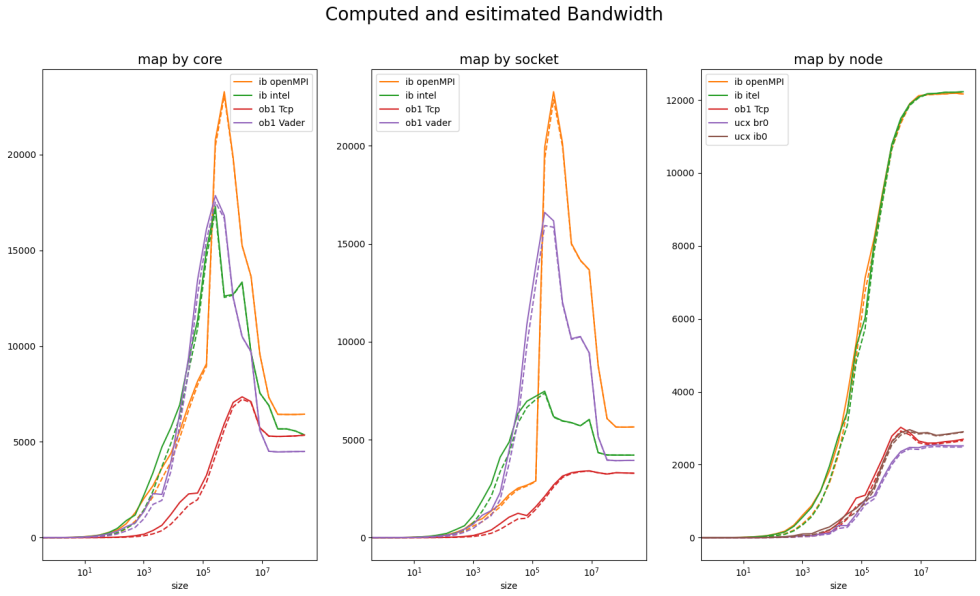

Figure 4: Actual bandwidth measured for different message sizes measured on a thin nodes

Gpu node

By executing the same benchmarks on a gpu node, we can elaborate similar conclusion.
The only thing it may be worth saying is the fact the overall performance obtained is slightly worse with respect to the one measured using a thin node.
There may be several explanations to this phenomena: for example it can be due to the fact that when taking the measurements, the node was used also by other processes, or hardware setting difference (e.g. the the processor on gpu node's clock rate is 10Hz smaller then the thin node or the fact that the software stack needed to take care of the hyper-threading may lead to a small overhead).
In figure 5 and 6 are reported the values obtained on the gpu node.

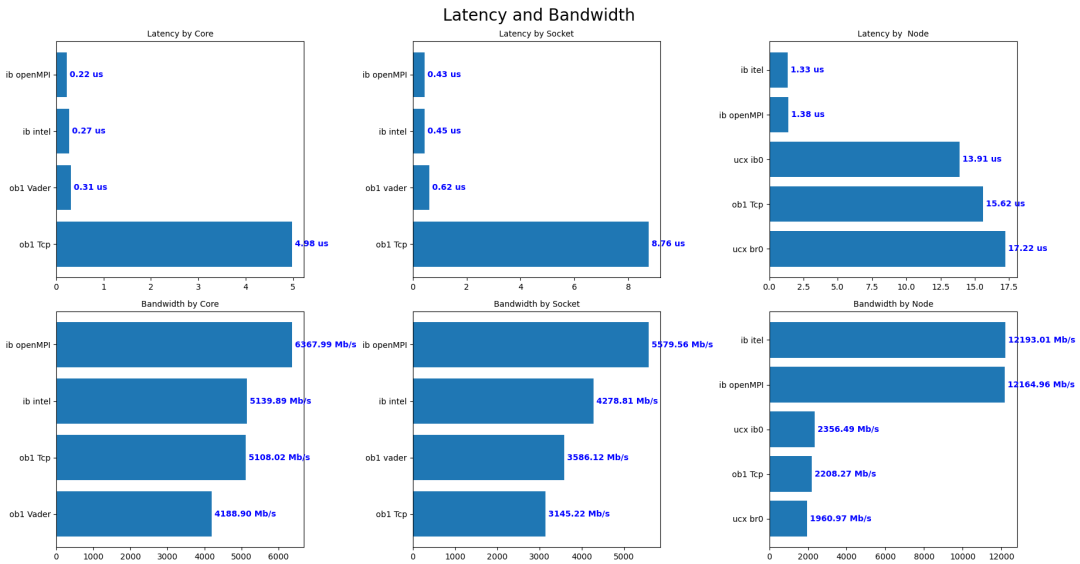

Figure 5: Summary of lantancies and bandwidth measured on gpu nodes

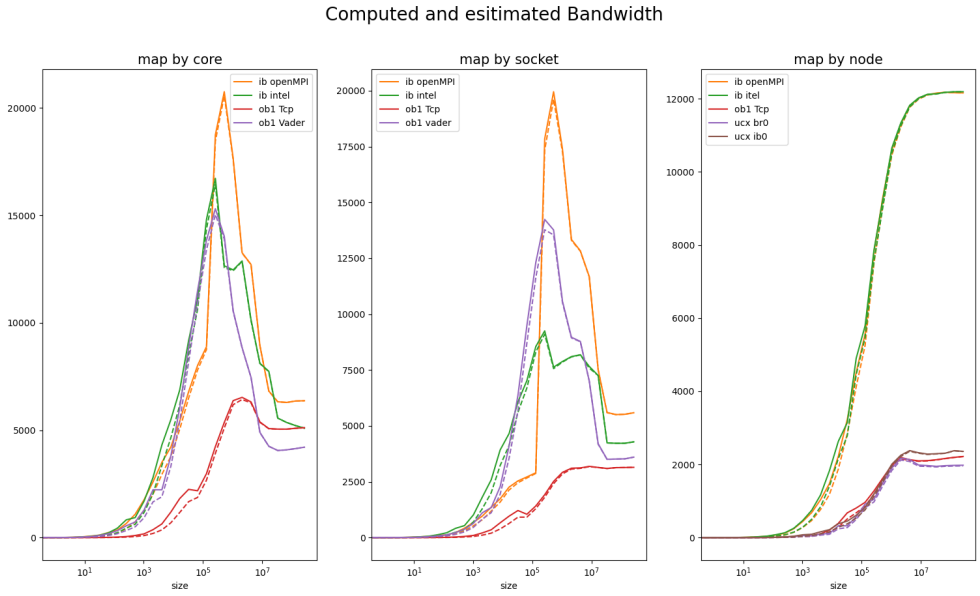

Figure 6: Actual bandwidth measured for different message sizes measured on gpu nodes

Exercise 3

Performance Model

The performance of the Jacobi solver can be modeled as follows:

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

where:

- L is the number of Lattice points that form the length in a direction of a single cubic subdomain;
- N $= N_1 N_2 N_3$ is the number of processors that forms the topology;
- T_s is the compute time for all cell updates in a Jacobi sweep.
- T_c is the communication time.

The communication time, as in the previous exercises has a latency component kT_l and a a component that depends on the dimension of the message size (C(**N**, L)) and the Bandwidth.

$$T_c = \frac{C(\mathbf{N}, L)}{Bandwidth} + kT_l$$

where k is the number of direction in which the halo layers are exchanged which corresponds to the number of topology dimensions with size greater than 1 and $C(\mathbf{N}, L) = 32L^2k$.
To have and estimation of the performance, this model has been computed using the correspondent value of latency and bandwidth obtained as a results of the previous exercise.

Experimental procedure

A weak scalability study has been done using the Jacobi solver used during classes.

The initial lattice size are $L = L_1 = L_2 = L_3 = 600$.
Then for different numbers of N = 4, 8, 12 for the socket mapping vs core mapping comparison and N = 12, 24, 48 for the thin vs gpu comparison.
For each N all the possible domain subdivision execution time has been measured. Since this is a weak scalability test, each time, the input lattice dimensions have been enlarged proportionally to the number core in the corresponding size of the topology.
E.g. if the topology has sizes (N_1, N_2, N_3), then the lattice dimension will be $(N_1 \cdot L, N_2 \cdot L, N_3 \cdot L)$ so that the workload on a single core remains constant.
To measure the execution time of a Jacobi run the elapsed time of the command usr/bin/time has been taken.

Experimental results
Here are presented two table similar to the one in the slides, that produces the theoretical model with the parameter of this experiment and also the computed values obtained by the analysis of the measurements.

Mapping by core

N	n1	n2	n3	k	time	P	Measured P	P(1) * N / P(N)
12	4	3	1	2	34.73	72.059593	71.175463	1.000113
12	12	1	1	1	34.79	72.063671	71.052711	1.000057
12	6	2	1	2	34.61	72.059593	71.422243	1.000113
12	3	2	2	3	34.61	72.055515	71.422243	1.00017
8	2	2	2	3	34.51	48.03701	47.752803	1.00017
8	8	1	1	1	34.57	48.042447	47.669922	1.000057
8	4	2	1	2	34.59	48.039728	47.64236	1.000113
4	4	1	1	1	34.53	24.021224	23.862572	1.000057
4	2	2	1	2	34.45	24.019864	23.917986	1.000113
1	1	1	1	0	34.3	6.005646	6.005646	1

Mapping by socket

N	n1	n2	n3	k	time	P	Measured P	P(1) * N / P(N)
12	12	1	1	1	36.9	72.064169	66.989806	1.00005
12	6	2	1	2	36.66	72.060588	67.428364	1.000099
12	4	3	1	2	36.44	72.060588	67.835451	1.000099
12	3	2	2	3	36.29	72.057007	68.11584	1.000149
8	4	2	1	2	35.25	48.040392	46.750332	1.000099
8	2	2	2	3	35.24	48.038005	46.763599	1.000149
8	8	1	1	1	35.3	48.042779	46.684114	1.00005
4	4	1	1	1	34.42	24.02139	23.938832	1.00005
4	2	2	1	2	34.43	24.020196	23.931879	1.000099
1	1	1	1	0	34.34	6.005646	5.99865	1

Thin across 2 nodes

N	n1	n2	n3	k	time	P	Measured P	P(1) * N / P(N)
48	24	2	1	2	36.04	288.242352	274.353366	1.000099
48	8	6	1	2	35.88	288.242352	275.576792	1.000099
48	6	4	2	3	35.89	288.228029	275.500009	1.000149
48	16	3	1	2	35.98	288.242352	274.810876	1.000099
48	12	2	2	3	35.92	288.228029	275.269914	1.000149
48	8	3	2	3	35.94	288.228029	275.116731	1.000149
48	48	1	1	1	36.03	288.256675	274.429512	1.00005
24	8	3	1	2	35.72	144.121176	138.405589	1.000099
24	4	3	2	3	35.79	144.114015	138.134888	1.000149
24	6	2	2	3	35.8	144.114015	138.096303	1.000149
24	24	1	1	1	35.74	144.128338	138.328138	1.00005
24	6	4	1	2	35.67	144.121176	138.599598	1.000099
24	12	2	1	2	35.66	144.121176	138.638465	1.000099
12	3	2	2	3	35.18	72.057007	70.265032	1.000149
12	4	3	1	2	35.2	72.060588	70.225109	1.000099
12	12	1	1	1	35.26	72.064169	70.105611	1.00005
12	6	2	1	2	35.24	72.060588	70.145398	1.000099
1	1	1	1	0	34.34	6.005646	5.99865	1

GPU node

N	n1	n2	n3	k	time	P	Measured P	P(1) * N / P(N)
48	8	6	1	2	75.89	195.357228	130.289831	1.000067
48	48	1	1	1	75.78	195.363808	130.478956	1.000034
48	12	2	2	3	76.18	195.350649	129.793848	1.000101
48	6	4	2	3	76.23	195.350649	129.708715	1.000101
48	24	2	1	2	76.08	195.357228	129.964449	1.000067
48	8	3	2	3	75.69	195.350649	130.634104	1.000101
48	16	3	1	2	75.74	195.357228	130.547865	1.000067
24	8	3	1	2	56.13	97.678614	88.078526	1.000067
24	24	1	1	1	56.12	97.681904	88.094221	1.000034
24	6	4	1	2	56.32	97.678614	87.781386	1.000067
24	12	2	1	2	56.16	97.678614	88.031475	1.000067
24	4	3	2	3	56.17	97.675325	88.015803	1.000101
24	6	2	2	3	56.16	97.675325	88.031475	1.000101
12	4	3	1	2	55.44	48.839307	44.587371	1.000067
12	12	1	1	1	55.51	48.840952	44.531144	1.000034
12	6	2	1	2	55.5	48.839307	44.539168	1.000067
12	3	2	2	3	55.63	48.837662	44.435086	1.000101
1	1	1	1	0	50.61	4.070216	4.070216	1

The performance values are plotted in figures 7 and 8.

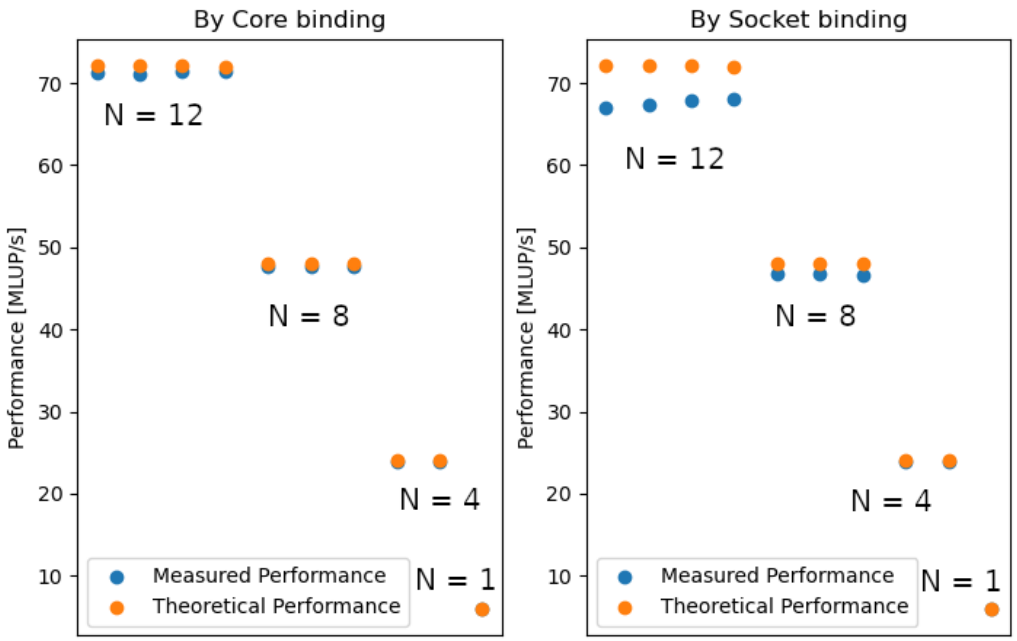

Figure 7: Performance comparison of Jacoby solver between within a thin node using different bindings

In the first figure (7) it can be seen that the performance model works well, especially for the core binding policy. For the socket binding there is some degradation in terms of performance. It could be caused by needs of many communication between cores of different sockets causing a degradation in the communication performance.

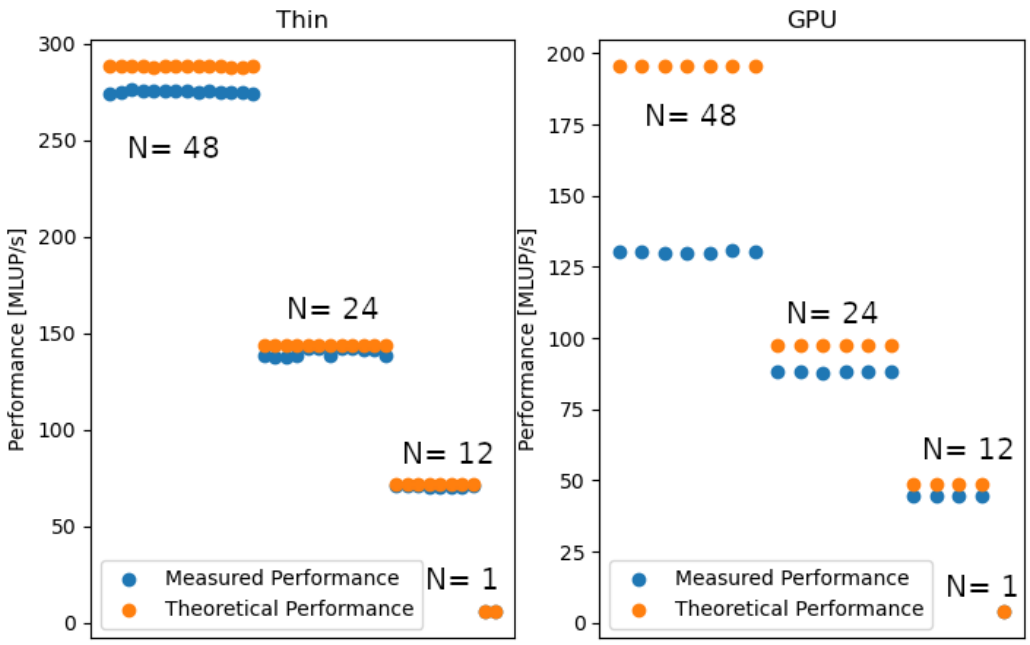

Figure 8: Performance comparison of Jacoby solver between thin and GPU node

In figure 8 the comparison between two different types of nodes is presented. While in on the thin node the model, behaves still quite good (91% of the theoretical one) for 48 cores, on the GPU the performance drops down dramatically, we see in fact a very small improvement passing from 24 to 48 cores.
This was however quite predictable, because even if the the virtual core seen by the user are 48 thanks to the hyperthreading, the number of physical core in the node is just 24.
Due to the fact that the Jacoby code is quite computational expensive and the core do not spend more than 50% of clock cycle in idle, the gain due to the presence of the double of cores is dumped.
Moreover there's a software layer that has to take care of the work distribution between the processes running on the same core and this brings an overhead.
This factors together brings to a real performance of 65% of the theoretical one.