github.com/SilvioBaratto/High-Performance-Computing-benchmarks

High Performance Computing Benchmarks

Silvio Baratto

19 maggio 2022

Indice

1 Ring

1.1 Analysis

In Figure 1 we report the performance of `ring.c` for varying number of processors. We expect an approximately linear growth, since the addition of a new process introduces a new step in the ring (and therefore two more input and two output messages for each process).

The time is taken for two values of the parameter `--map-by`, namely `core` and `socket`. As expected the `--map-by core` case outperforms the other one until `P=12`. As soon as this threshold is passed two new communication channels are created between a process from `socket0` and a process from `socket1`, which is more costly than the communications we had in the region in the left part of the figure. This observation explains the steep increase in the execution time from 12 to 13 processes. However the evolution of the time recovers its linearity after the central region.

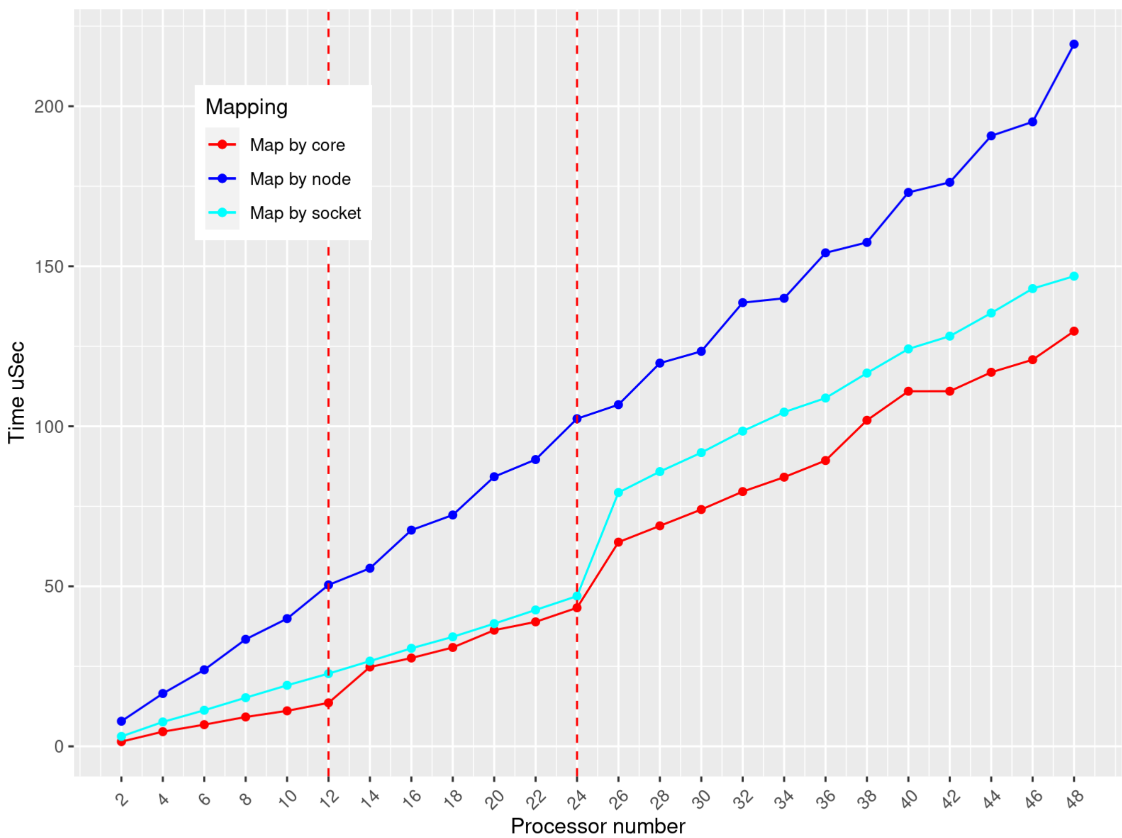


Figura 1: The script has been run multiple times (100) on a THIN node.

2 Matrix

In memory, a 3D array can be represented as a unique linear array. The most effective way to sum it in parallel is using collective operations. The shape of the array doesn't make any difference on how the array is represented in memory. Also the virtual topology and its relative domain decomposition do not have any impact. Indeed, we can assume that the problem is not sensible to any topology, and thus is no need of communication between neighbours. The most efficient way to communicate the sum between matrices is to use `mpi_scatterv` and `mpi_gatherv` collective routines.

As we can see from the table 1 all the different topologies have the same behavior given the fact that we are using only collective operations. The times collected are pretty much the same except for some fluctuations in "Runtime" (caused by the matrix filling by random numbers and by the creation of virtual topology).

Distribution	Topology	Time	Nprocessor
2400x100x100	24x1x1	0.63740558	24
2400x100x100	12x2x1	0.63056059	24
2400x100x100	6x2x2	0.62238741	24
2400x100x100	4x3x2	0.68667042	24
1200x200x100	24x1x1	0.68244459	24
1200x200x100	6x2x2	0.71415972	24
1200x200x100	4x3x2	0.62713907	24
800x300x100	24x1x1	0.63395623	24
800x300x100	12x2x1	0.68399210	24
800x300x100	6x2x2	0.67890758	24
800x300x100	4x3x2	0.68265990	24

Tabella 1: Time is composed by the following operations: Scattering of matrix A and matrix B, Addition of chunk of matrices and Gathering of results into matrix C

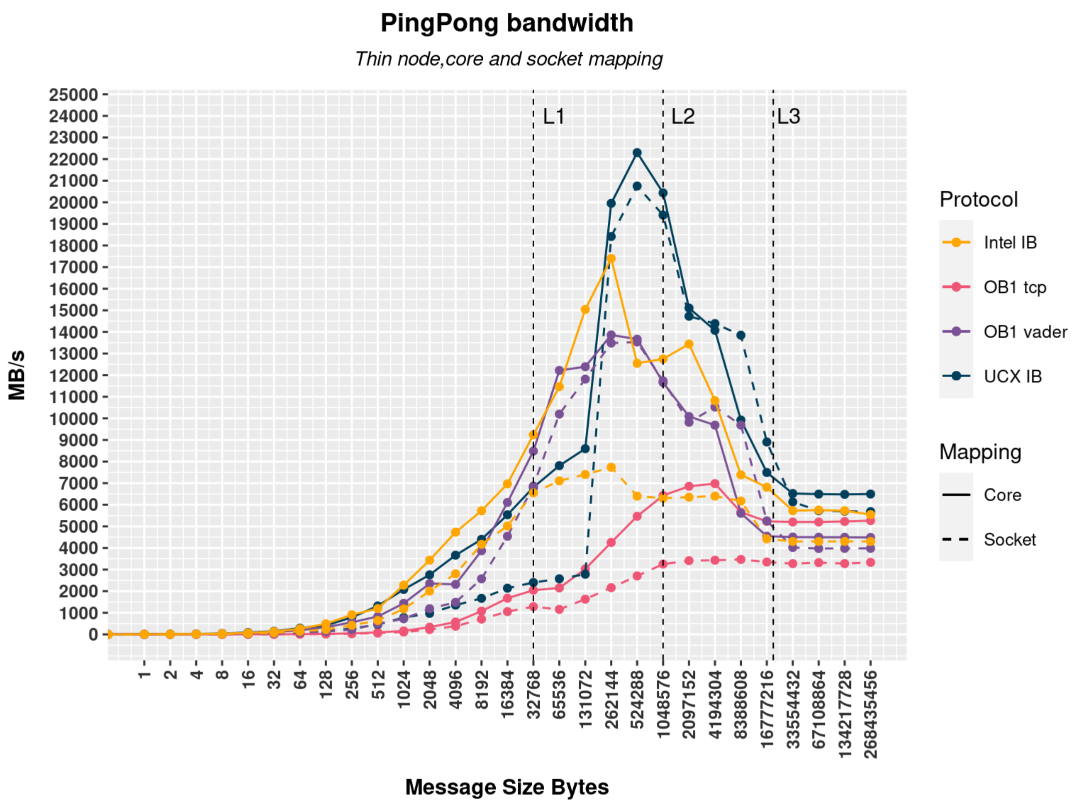
The complete table of results is found in the folder `section_1/build/matrix_times_24.csv`

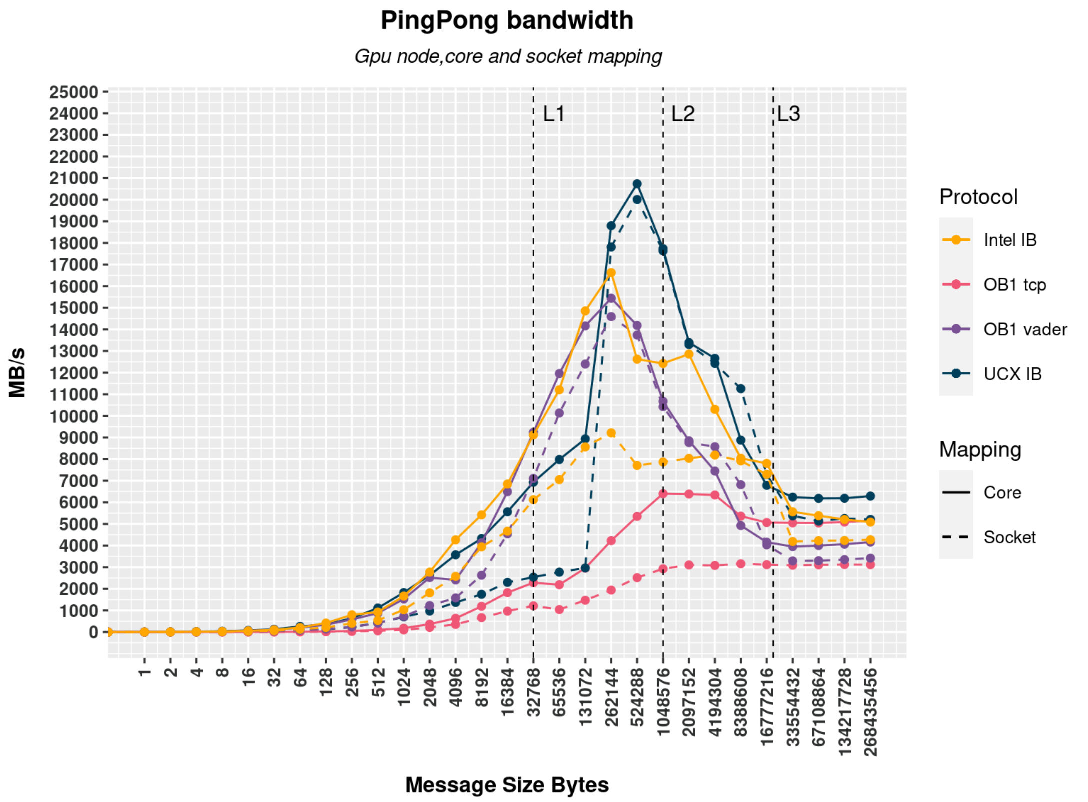
3 MPI PingPong performance

In this section are presented the result obtained with the `PingPong` benchmark on ORFEO. The benchmark has been ran on multiple kind of nodes and networks, and against different implementations of MPI.

The bandwidth and the latency estimates are computed across core, socket and different nodes, combined with different protocols and hardware devices. Each of the different setup of the program has been runned 5 times and `openMPI-4.1.1` has been used. The `pml` involved in the benchmarks are `OB1` and `UCX`. The `btl` used are `tcp` and `vader`. When the measurements across nodes were perfomed also different networks with different protocols have been selected: 25 Gbit Ethernet and 100 Gbit InfiniBand.

Mapping the processes in the same socket show often a better performance, as expected. The behaviour before asymptotic plateau is strange and shows very different performance among different implementations, the main cause of that is due to the cache. As it can be noticed in the following two graphs, the bandwidth behaviour appears strange before 16MB included. After that point, the behaviour is stable because the message size is larger than all the caches. The larger cache is `L3` with 19MB. The effect of `L2` becomes clear after 1MB, after it all implementations start loosing bandwidth. The effect of `L1` is not clearly visible from the bandwidth due to the latency. Intel InfiniBand shows poor perfomance with respect to UCX implementation, this behaviour can be explained with cache too.





3.1 Differences observed between IntelMPI and OpenMPI

It is clear that the main difference is that IntelMPI looks much more stable than OpenMPI, especially in the left region. Since the region interested by this phenomenon is the one in which the latency is dominant with respect to the bandwidth, we may assume that the two implementations communicate using different protocols with the network interface, thus encountering different costs in establishing the communication and sending the message.

3.2 Differences observed between Infiniband and Gigabit

UCX with InfiniBand and Intel shows the best performance among all the configurations both in terms of latency and bandwidth. This is most likely due to the fact that Gigabit networks perform some more controls on the communication (e.g it verifies that the message arrives to the receiver), therefore message needs to pass through more layers with respect to InfiniBand networks. Also, InfiniBand employs several ”tricks” to make communication faster, which likely contributes to the reduced latency. For instance when using Gigabit the user of the networks has to pre-process the message using its own computational resources. This does not happen with InfiniBand, since the pre-processing happens on a software module of the network.

3.3 Thin node latency and bandwidth

Thin node	Protocol	Latency μSec	Bandwidth MB/s
Core mapping	UCX ib	0.22	6450.36
	Intel ib	0.24	5702.59
	OB1 vader	0.31	4472.75
	OB1 tcp	5.5	5123.82
Socket mapping	UCX ib	0.54	5679.78
	Intel ib	0.48	4289.15
	OB1 vader	0.72	3954.68
	OB1 tcp	8.19	3284.49
Node mapping	UCX ib	1.07	12176.39
	Intel ib	1.34	12235.85
	OB1 vader	1.06	12187.99
	OB1 tcp	15.76	2402.39

3.4 Gpu node latency and bandwidth

Gpu node	Protocol	Latency μSec	Bandwidth MB/s
Core mapping	UCX ib	0.26	6157.24
	Intel ib	0.32	5216.97
	OB1 vader	0.30	4030.76
	OB1 tcp	5.03	5072.48
Socket mapping	UCX ib	0.60	5201.56
	Intel ib	0.53	4209.24
	OB1 vader	0.71	3318.39
	OB1 tcp	8.73	3121.66
Node mapping	UCX ib	1.58	12185.46
	Intel ib	1.48	12200.16
	OB1 vader	1.53	12178.97
	OB1 tcp	15.62	2038.15

3.5 Other observations

GPU nodes behave like THIN nodes, and thus no difference can be seen, GPU Node is only a bit slower than THIN node. This can be due to different GPU frequency and node configuration. Cache size are the same of THIN nodes, and this cache effect are similar.

4 Jacobi solver

4.1 Performance

To predict the performance of the Jacobi model we use the following formula:

$$P(L,N) = \frac{L^3 N}{T_s + T_c}[MLUP/s] \tag{1}$$

L is the size of cubic sub-domains, T_s is the time for the lattice updates of a domain with size L and T_c is the communication time. The quantity T_s can be modeled estimating the latency λ, bandwidth B and messages size C.

$$T_c = \frac{C(L,N)}{B} + 4k\lambda[s] \tag{2}$$

$C(L,N)$ is the maximum bidirectional bandwidth data volume transferred over a node's network link, k is the largest number of coordinate directions in which the number of processes is greater than one and λ is the latency. The formula for C is:

$$C = L^2 \times k \times 2 \times 8[byte] \tag{3}$$

4.2 Results

The following tables compare the results obtained by running Jacobi with the performance model. In thin nodes, the performance $P(N,L)$ predicted from theoretical model reflect the real performance obtained from computation well. In Gpu nodes, instead there is an overestimation from the model, this could happen because hyper-threading enabled is on this node.

THIN node, core mapping

$\lambda = 0.22$, Bandwidth = 6450.36

N	k	C[Mb]	Tc/s	MLUP/s est	MLUP/s real	MLUP/s diff	NP(1)P(N)
1	2	46.08	0.00714423	112.741	112.738	-0.00312003	1
4	4	92.16	0.0142885	450.755	451.917	-1.16281	0.997864
8	6	138.24	0.0214327	901.089	890.34	-10.749	1.01299
12	6	138.24	0.0214327	1351.63	1325.43	-26.2079	1.0207

THIN node, socket mapping

$\lambda = 0.54$, Bandwidth = 5679.78

N	k	C[Mb]	Tc/s	MLUP/s est	MLUP/s real	MLUP/s diff	NP(1)P(N)
1	2	46.08	0.00811407	112.741	112.738	-0.00401334	1
4	4	92.16	0.0162281	450.755	447.417	-3.28006	1.0079
8	6	138.24	0.0243422	901.089	883.429	-17.4894	1.02091
12	6	138.24	0.0243422	1351.63	1336.19	-15.1886	2.09478

THIN node, node mapping

$\lambda = 1.07$, Bandwidth = 12176.39

N	k	C[Mb]	Tc/s	MLUP/s est	MLUP/s real	MLUP/s diff	NP(1)P(N)
1	2	46.08	0.00378651	112.766	112.738	-0.0278236	1
4	4	92.16	0.0113595	1352.52	1326.58	-25.942	1.01981
8	6	138.24	0.0113595	2705.04	2627.13	-77.913	1.02991
12	6	138.24	0.0113595	5410.09	5131.27	-278.814	1.0546

GPU node, core mapping

$\lambda = 0.26$, Bandwidth = 6157.24

N	k	C[Mb]	Tc/s	MLUP/s est	MLUP/s real	MLUP/s diff	NP(1)P(N)
1	2	46.08	0.00748439	78.1282	78.1421	-0.0138761	1
4	4	92.16	0.0149688	312.407	309.684	-2.72303	1.00931
8	6	138.24	0.0224532	624.603	586.338	-38.2651	1.06617
12	6	138.24	0.0224532	936.905	850.567	-86.3376	1.10245

GPU node, socket mapping

$\lambda = 0.60$, Bandwidth = 5201.56

N	k	C[Mb]	Tc/s	MLUP/s est	MLUP/s real	MLUP/s diff	NP(1)P(N)
1	2	46.08	0.00886008	78.1234	78.1421	-0.0187353	1
4	4	92.16	0.0177202	312.368	311.43	-0.938776	1.00366
8	6	138.24	0.0265802	624.487	611.262	-13.2245	1.0227
12	6	138.24	0.0265802	936.73	899.664	-37.066	1.04228

GPU node, node mapping

$\lambda = 1.58$, Bandwidth = 12185.46

N	k	C[Mb]	Tc/s	MLUP/s est	MLUP/s real	MLUP/s diff	NP(1)P(N)
1	2	46.08	0.00378472	78.1413	78.1421	-0.000805104	1
4	4	92.16	0.0113541	937.375	897.053	-40.3223	1.04532
8	6	138.24	0.0113541	1874.75	1686.53	-188.224	1.112
12	6	138.24	0.0113541	3749.5	2488.66	-1260.83	1.50716

The biggest difference between model and data is the communication time. In fact, the time T_c predicted from the model is hundred time smaller then the observed one. However, the simple model used is good enough to predict the performance of the program.