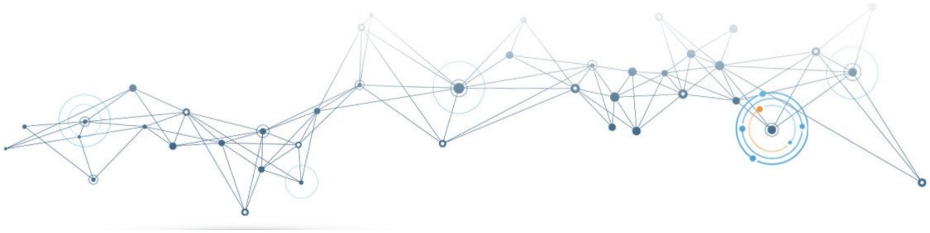
University of Trieste



Assignment I

Foundations Of High Performance Computing

December 28, 2021

https://github.com/BernardoManfriani/HPCAssignment.git

Bernardo Manfriani

Contents

1 MPI programming

1.1 MPI ring topology (section1/ring/ring.c)

In this exercise it is necessary to implement a ring topology, *ringCommunicator*, in which each processor sends to his left and right processor his message (-*rank* to right, *rank* to left). For this program I used a non-blocking solution since I think it could be a faster than the blocking approach. Each processor sends (**MPI_Isend()**), receives (**MPI_IReceive()**) and forwards his and the received *rank* (abducted from left, summed from right) and the *itag* to synchronise the communication. The whole procedure is done 100000 times and at the end the final time (*timeF*) is divided by 100000 to take the computational time normalized. Indeed just for one time, message passing procedure, has an high level of approssimation error. Moreover, the longest time is taken for each iteration, with a reduce procedure, for each processor from 2 to 48:

```
MPI_Reduce(&timeT, &timeRcv, 1, MPI_DOUBLE, MPI_MAX,0, ringCommunicator);
```

This served to further minimise the oscillation of the times which are represented in the figure 1.

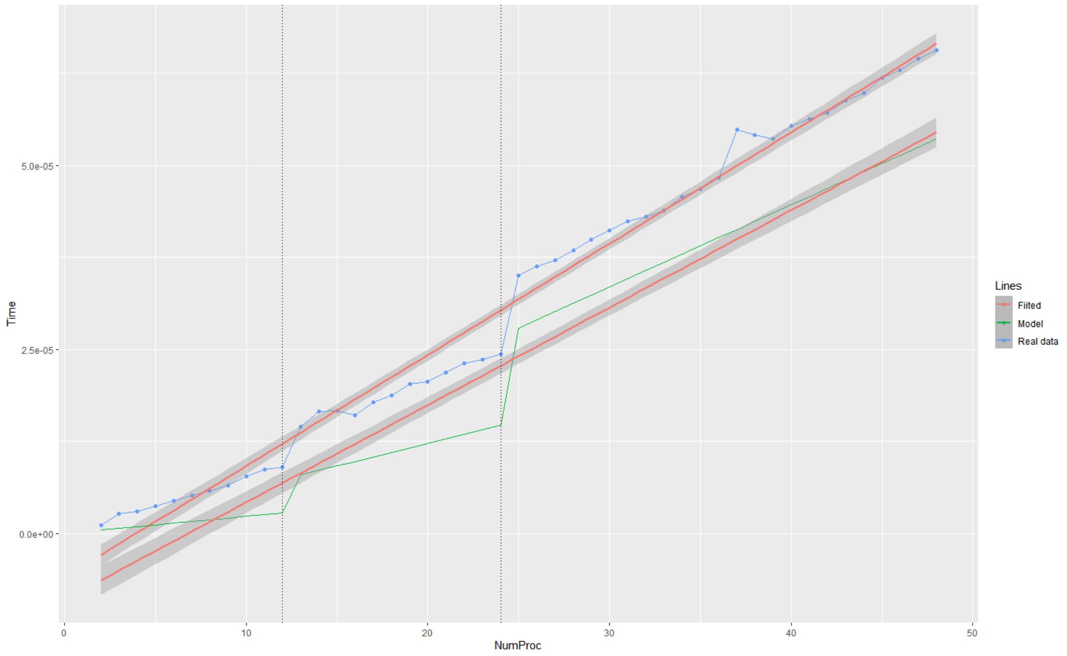


Figure 1: Comparison of times between model and real data with their fitted

The code in *ring.c* file is run through *ringScript.sh* script that iterate the procedure between 2 until 48 cores.

The execution time ranges between about $1.1 \cdot 10^{-6}$s (2 cores) and $65.66 \cdot 10^{-6}$s (48 cores). These times differ by an order of magnitude but its growth is linear on number of processors as we expect.

Model derives from this formula

$$time = nproc\left(\frac{2 \cdot size \cdot 10^{-6}}{bandwidth} + latency\right)$$

$Nproc$ is the number of processors, $size$ is the dimension of an integer in MPI namely 2 (is multiplied for 2 because each processor sends two messages) , $bandwidth$ is an array of bandwidths for each number of processor (from 2 to 12 is mapped by core: 0.2329; from

13 to 24 is mapped by core: 0.6113; from 25 to 25 is mapped by node: 1.1157). As we can see in the figure 1 on $NumProcm = 12$ and $NumProc = 24$ there are steps that derive from a different mapping. Actually on 12 there is a little step (mapping from core to socket) on 24 there a bigger one because passes from 2 sockets to 2 nodes. However there is another little step, that is not supported by the model, on 36 processors. This comes from the activation of connection between two sockets in the second node.

1.2 3D Matrix-Matrix addition (section1/matrix/sum3Dmatrix.c)

MY solution for sum3Dmatrix is without using virtual topology. This choice derives from the fact that there are no communication or *halo* between processors except for the communication with master (rank 0). Actually the domanin decomposition does not increase the performance so the difference in terms of time is not significant. My solution allocate a 3D matrix of dimension r1 r2 r3 in rank 0, where sizes are:

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

In the other processors i allocate a matrix of dimension

```
r1=(atoi(argv[1])*atoi(argv[2])*atoi(argv[3]))/size, r2 = 0 , r3 = 0;
```

where argv[1], argv[2], argv[3] are the input sizes of the above list, so actually an array. Indeed **MPI_Scatter()** and **MPI_Gather()**, by dafault, take an array and operate on arrays and not on matrices.

Time are computed by code considering just communication operations (so collective operations) and not the sum of matrices operations. Of correspondence the model time is described by the underlying formula:

$$time = n \cdot \left(\frac{size}{bandwidth} + latency \right),$$

where $size = 192$ (24MB as size of matrix $\cdot$ 8 MB as size of double), n is the number of collective operation that could be 3 (2$\cdot$*MPI_Scatter()* and *MPI_Gather()*), bandwidth and latency are taken from the ping pong benchmark, mapped by socket with infiniband communication and using Intel library. *MatrixScript.sh* file run the code with different set of sizes:

```
    mpirun -np 24 -mca btl ^openib ./matrix 2400 100 100 > matrix_time_2400
    mpirun -np 24 -mca btl ^openib ./matrix 1200 200 100 > matrix_time_1200
    mpirun -np 24 -mca btl ^openib ./matrix 800 300 100 > matrix_time_800
```

For each set of sizes times are :

Size	Sperimental time	Model time
2400 100 100	0.16794164	0.08000786
1200 200 100	0.15592271	0.08000786
800 300 100	0.15756187	0.08000786

2 MPI point to point performance (section2)

In the section 2 it is necessary to estiamate latency and bandwidth of all available combinations of topologies and networks on ORFEO computational nodes with the Ping-Pong benchmark. I have tested an amount of 32 combinations with different implementation, mapping and networking:

- 16 from THIN nodes and 16 from GPU nodes of wich:
  - 8 OpenMPI and 8 Intel each mapped:
    - 3 mapped by core
    - 3 mapped by socket
    - 2 mapped bu node

All this combination generates 32 .csv files stored in *section2/csv/thin* and *section2/csv/gpu*. File are generated from the scripts (stored on github repository/section2) **benchmarkThin.sh -> csv/thin** and **benchmarkGPU.sh -> csv/gpu**. File names represent mapping, implementation and networking (*e.g. csv/gpu/bycoketUcxGPU.csv* is mapped by socket with ucx networking, with openMPI and on GPU nodes).

2.1 Analysis and performances of ping pong benchmark

In the folder *section2/img* there are 24 plots that describe all combination in terms of bandiwidth (i.e. speed in MB/s) and the time of communication in μsec (when message size is 0 it is the latency). Plots shown empirical data, communication model and fitted data (reported also on the csv files). The communication model is:

$$time = \frac{size}{bandwidth} + latency,$$

where latency and bandwidth are taken from the experimental data.

The first thing that it can been noticed is that are not many differences between using THIN node and GPU node, indeed the benchmark measures networks performance so the differences between thees two nodes are not relevant.

One of the interesting detail is the difference between the OpenMPI and IntelMPI implementation, in particular about the difference between the infiniband connection intersocket in a THIN node. As we can see in the figure 2 using infiniband the Ping-Pong benchmark with OpenMPI has the peak performance around $18\,000\,\mathrm{MB/s}$ while with IntelMPI has the peak around $7500\,\mathrm{MB/s}$. We can observe this behaviour also in intra-socket communication. One of the possible explanation of this phenomena is that with IntelMPI there are a lot of cache misses compared to OpneMPI implementation. The possible explanation of this behaviour is that with IntelMPI there are more cache missings than with OpenMPI.

Another interesting thing is that in intra-node communication, at around 1MB as size of message, there is a drop of the bandwidth. Maybe, after the message size becomes larger than L2 cahce, $1\,\mathrm{MB}$, there are a lot of cache misses. Furthermore, quoting the book *Introduction to high performance computing for scientist and engineers G. Hager, G. Wellein* "the IMB 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", so when the number of repetitions decrease the performance could affect by this decreasing.

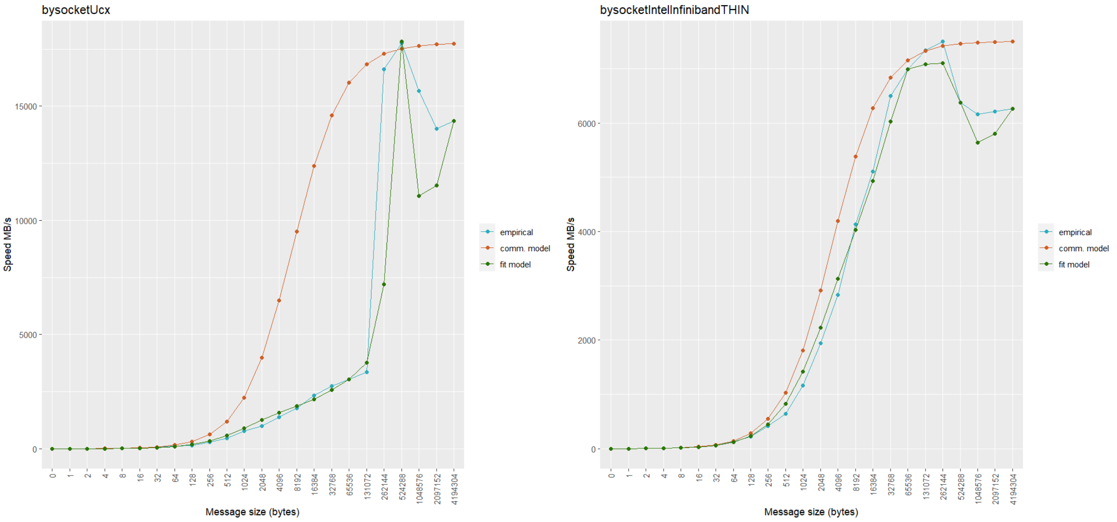


Figure 2: Comparison of speed between OpenMPI and IntelMPI inter-socket

3 Performance observed vs model for Jacobi solver

To evaluate performances of Jacobi solver I run two bash scripts: *JacobGPU.sh* and *JacobiTHIN.sh* that put results in *Results_Jacobi.csv* and *Results_JacobiGPU.csv*. These scripts use $1200, 0, tx3$ as input and so 1200 is the size of one side. Tables 3 and 4 summarize data collected on THIN and GPU nodes running 3D Jacobi solver and their respective performance models *P(L,N)* with total size of the problem 1200^3. The model is the one discussed in class:

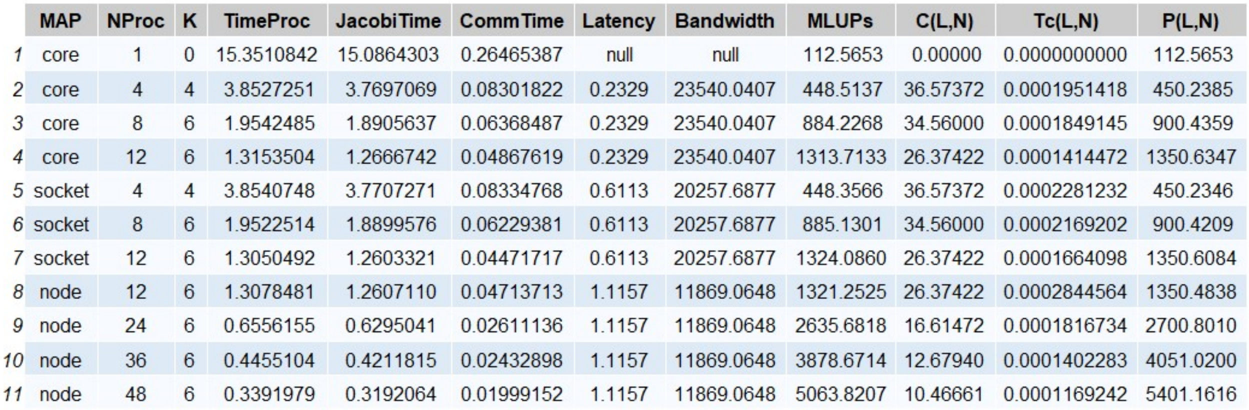
$$P(L, N) = \left(\frac{L^3 \cdot N}{T_s + T_c(L, N)}\right),$$

$$T_c(L, N) = \left(\frac{C(L, N)}{Band} + k \cdot lat\right),$$

$$C(L, N) = 16kL^2,$$

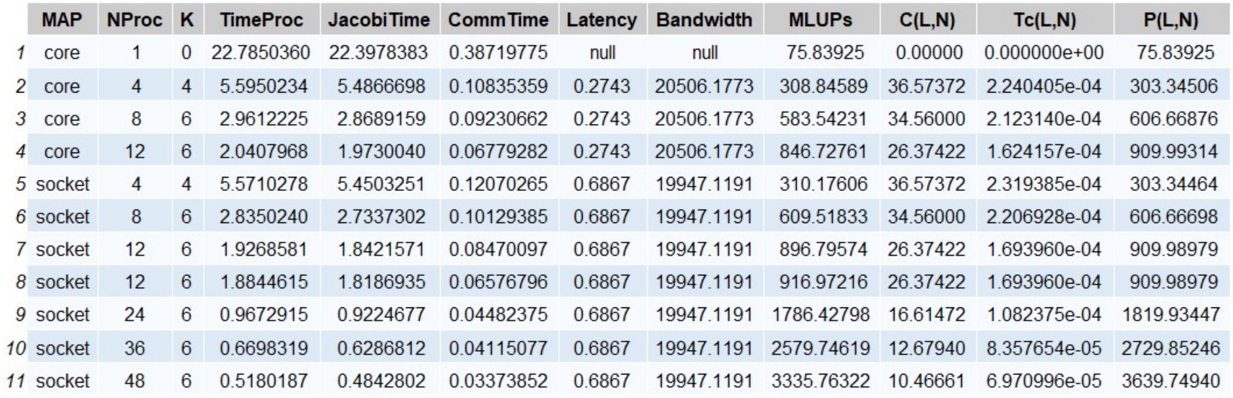
where N is the number of processors, L is the subdomain size ($\frac{1200}{\sqrt[3]{N}}$) and T_s is sequential time:

$$T_s = \frac{TimeProc[1]}{Nproc},$$



	MAP	NProc	K	TimeProc	JacobiTime	CommTime	Latency	Bandwidth	MLUPs	C(L,N)	Tc(L,N)	P(L,N)
1	core	1	0	15.3510842	15.0864303	0.26465387	null	null	112.5653	0.00000	0.0000000000	112.5653
2	core	4	4	3.8527251	3.7697069	0.08301822	0.2329	23540.0407	448.5137	36.57372	0.0001951418	450.2385
3	core	8	6	1.9542485	1.8905637	0.06368487	0.2329	23540.0407	884.2268	34.56000	0.0001849145	900.4359
4	core	12	6	1.3153504	1.2666742	0.04867619	0.2329	23540.0407	1313.7133	26.37422	0.0001414472	1350.6347
5	socket	4	4	3.8540748	3.7707271	0.08334768	0.6113	20257.6877	448.3566	36.57372	0.0002281232	450.2346
6	socket	8	6	1.9522514	1.8899576	0.06229381	0.6113	20257.6877	885.1301	34.56000	0.0002169202	900.4209
7	socket	12	6	1.3050492	1.2603321	0.04471717	0.6113	20257.6877	1324.0860	26.37422	0.0001664098	1350.6084
8	node	12	6	1.3078481	1.2607110	0.04713713	1.1157	11869.0648	1321.2525	26.37422	0.0002844564	1350.4838
9	node	24	6	0.6556155	0.6295041	0.02611136	1.1157	11869.0648	2635.6818	16.61472	0.0001816734	2700.8010
10	node	36	6	0.4455104	0.4211815	0.02432898	1.1157	11869.0648	3878.6714	12.67940	0.0001402283	4051.0200
11	node	48	6	0.3391979	0.3192064	0.01999152	1.1157	11869.0648	5063.8207	10.46661	0.0001169242	5401.1616

Figure 3: 3D Jacobi solver with THIN nodes



	MAP	NProc	K	TimeProc	JacobiTime	CommTime	Latency	Bandwidth	MLUPs	C(L,N)	Tc(L,N)	P(L,N)
1	core	1	0	22.7850360	22.3978383	0.38719775	null	null	75.83925	0.00000	0.000000e+00	75.83925
2	core	4	4	5.5950234	5.4866698	0.10835359	0.2743	20506.1773	308.84589	36.57372	2.240405e-04	303.34506
3	core	8	6	2.9612225	2.8689159	0.09230662	0.2743	20506.1773	583.54231	34.56000	2.123140e-04	606.66876
4	core	12	6	2.0407968	1.9730040	0.06779282	0.2743	20506.1773	846.72761	26.37422	1.624157e-04	909.99314
5	socket	4	4	5.5710278	5.4503251	0.12070265	0.6867	19947.1191	310.17606	36.57372	2.319385e-04	303.34464
6	socket	8	6	2.8350240	2.7337302	0.10129385	0.6867	19947.1191	609.51833	34.56000	2.206928e-04	606.66698
7	socket	12	6	1.9268581	1.8421571	0.08470097	0.6867	19947.1191	896.79574	26.37422	1.693960e-04	909.98979
8	socket	12	6	1.8844615	1.8186935	0.06576796	0.6867	19947.1191	916.97216	26.37422	1.693960e-04	909.98979
9	socket	24	6	0.9672915	0.9224677	0.04482375	0.6867	19947.1191	1786.42798	16.61472	1.082375e-04	1819.93447
10	socket	36	6	0.6698319	0.6286812	0.04115077	0.6867	19947.1191	2579.74619	12.67940	8.357654e-05	2729.85246
11	socket	48	6	0.5180187	0.4842802	0.03373852	0.6867	19947.1191	3335.76322	10.46661	6.970996e-05	3639.74940

Figure 4: 3D Jacobi solver with GPU nodes

As we can see in table 3 in THIN nodes P(L,N) is predicted by the model quite good. In the GPU nodes is also quite good but is a bit higher than CPU and so there is an overestimation (figure 5). Probably this happens due to the hyper-threading. Indeed each core do the same thing and hyper-threading performs better when cores do different tasks. One of the biggest difference between model and data is T_c. Indeed, this time predicted from the model is hundred time smaller then the observed one. However, this model is enough good to predict the performances, i.e. the thing of interest to us.

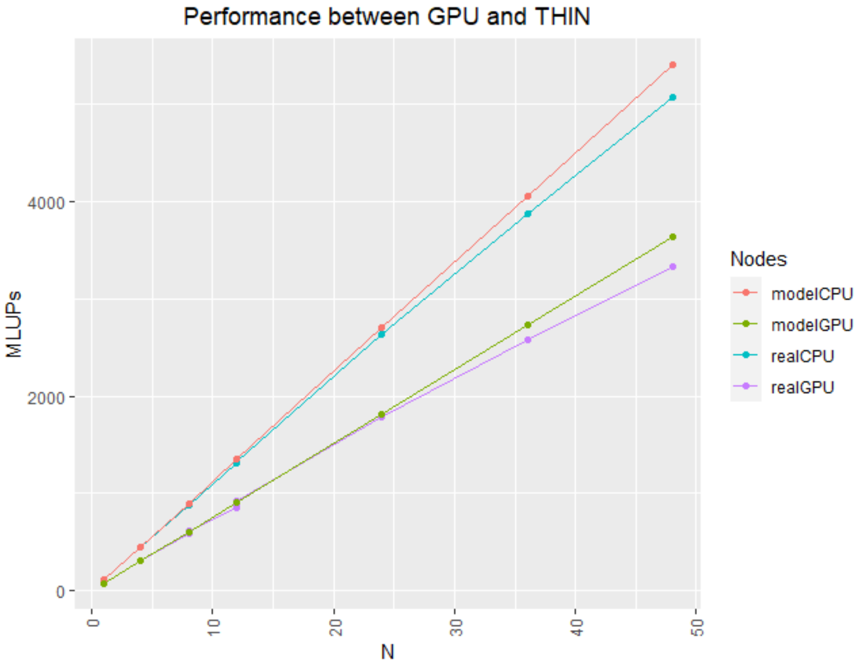


Figure 5: Scalability of Jacobi solver with THIN and GPU nodes