Assignment 1

Foundations of HPC 2021-2022

Francesco Ortu | francescortu@live.it |
github.com/Francesc0rtu/HPC-course-2021-2022

Section 1

Ring

In order to solve the exercise I provide two different implementation: one it uses only blocking operations while the other one uses non-blocking operations.

Implementation

In the blocking implementation I have divided the cores in two subsets based on the parity of ranks. The odd processors send to right and left, and then they receive from left and right. Instead, the even processors first receive from left and right and subsequently send to right and left. In this way the program avoid deadlock. Obviously, when the cardinality of the processors is an odd number I had to modify the behaviour of one processor, like the one with rank 0. In these cases, the processor 0 send to left and receive from right, and after send to right and receive from left. The execution in the two cases is shown in figure 1.

The implementation that use only non-blocking operations is simpler: each processor send and receive from e to its neighbours. Then, there is an *MPI_barrier* in order to prevent a processor from update its message before it has received it. This barrier implicitly makes the execution ”blocking”, however, as seen in the next section, this implementation is one time faster then the other one.

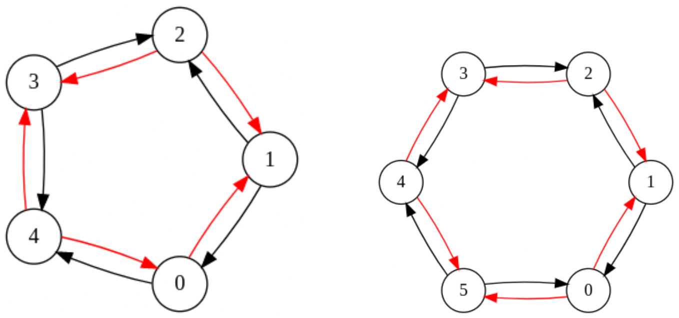


Figure 1: The blocking implementation with an odd and an even numbers of processors. In black there are the first operation executed while in red the second one.

Runtime analysis

Since each processor send and receive $2n$ messages, where n is the total number of processors, the runtime is expected to grow linearly on n. Meanwhile, the total number of messages exchanged between processors grows as $O(n^2)$.

The runtime is calculated as the time of the slowest core. In addition, to obtain significant data, I have taken the mean over 10000 repetition. The figure 2 show the time as a function of the number of processors used, i.e the number of vertices of the ring. As we expected, the time grows linearly and the non blocking implementation is faster then the blocking one. It can be notice that, when it has been added a socket, there is a step in the runtime; the biggest one is between 24 and 25 core, since the program starts using two nodes. The trivial reason is that the bandwidth and the latency decrease with more sockets and nodes used. However, after a first peak the time goes back down each time, and that is probably because after the first time that the program use a new socket or node, the channel is already open.

To estimate the runtime it is possible to consider the following model: each processors send $2n$ messages, so the time it takes will be

$$T = n\left(\lambda + \frac{\text{2*size of the msg}}{\text{bandwidth}}\right)$$

where I considered the bandwidth and the latency of the slowest network involved in the computation. In the figure 2, it is possible to see the trend of the model.

The model seems to be quite good for the non-blocking implementation. The blocking implementation should take around two times the other one, because processors take turns exchanging messages.

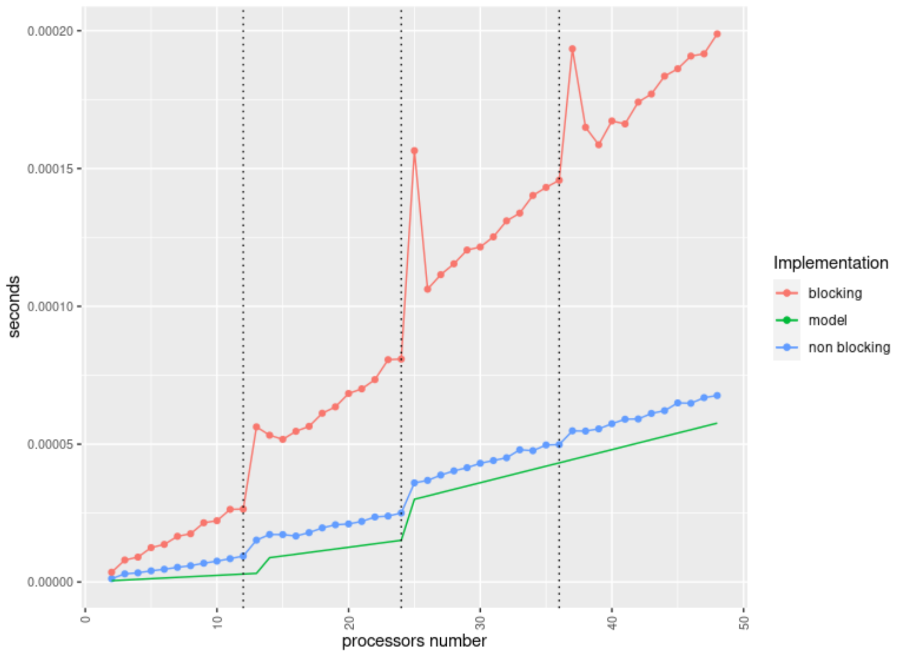
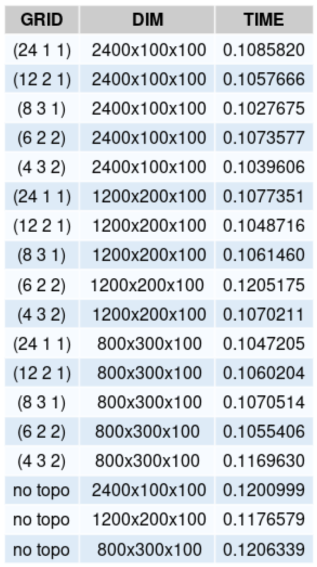


Figure 2: Walltime of the ring as a function of the number of processors.

Matrix-Matrix sum

I implemented the matrix-matrix sum program both with topology and not. As the figure 3 shown, the topology is useless to gain more performance in this exercise. In fact, since there are not halo or communications between processors, the domain decomposition can not result in a speed up.



GRID	DIM	TIME
(24 1 1)	2400x100x100	0.1085820
(12 2 1)	2400x100x100	0.1057666
(8 3 1)	2400x100x100	0.1027675
(6 2 2)	2400x100x100	0.1073577
(4 3 2)	2400x100x100	0.1039606
(24 1 1)	1200x200x100	0.1077351
(12 2 1)	1200x200x100	0.1048716
(8 3 1)	1200x200x100	0.1061460
(6 2 2)	1200x200x100	0.1205175
(4 3 2)	1200x200x100	0.1070211
(24 1 1)	800x300x100	0.1047205
(12 2 1)	800x300x100	0.1060204
(8 3 1)	800x300x100	0.1070514
(6 2 2)	800x300x100	0.1055406
(4 3 2)	800x300x100	0.1169630
no topo	2400x100x100	0.1200999
no topo	1200x200x100	0.1176579
no topo	800x300x100	0.1206339

Figure 3: Execution of the 3Dmatrix sum with different topologies and sizes.

In both implementation, the *master* allocates three matrix with random numbers. Then, it send to the other their sub-matrices to sum, where the sub-matrices are simply allocated as an array of dimension $\frac{size}{n}$. Even if I used the topology, I did not split the data in the right way, because I always cut the domain in slices. That because the topology is not useful at all in this program and I only implemented the topology to have a feedback about that. Anyway, a possible way to divide the data is by use of *MPI Scatterv* and/or *MPI Type*.

Since in provided code there are three collective operation (2 *MPI_Scatter*, 1 *MPI_Gather*), a simple communication model could be

$$T_{comm} = 3 * \left(lambda + \frac{size}{band}\right)$$

where $size$ is the dimension of the matrices. However, the real behaviour of the program is may different and the results obtained with this model are far from the experimental times.

Section 2

I ran the PingPong benchmark on THIN and GPU nodes, both with OpenMPI and IntelMPI. The first thing I noticed is that, as I expected, there are not big differences between GPU and THIN: the PingPong measure networks performance, so the differences between CPUs are not such relevant.

In the folder */fast/dssc/francescortu/2021Assignement01/section2* there are plenty of csv files and plots that describe what I found. In addition to the experimental data, as requested, the plots shown the simple communication model discussed during lessons

$$T = \lambda + \frac{size}{band}$$

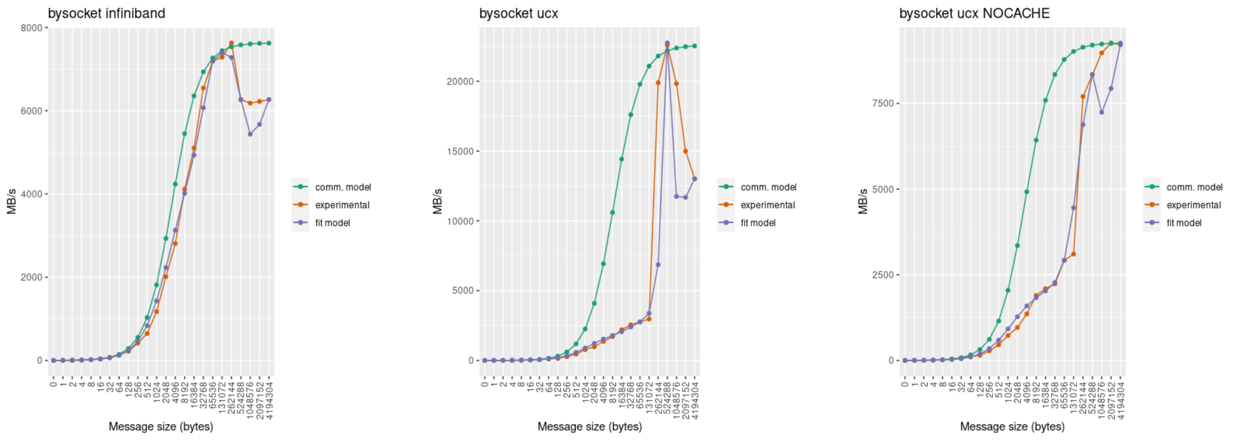
computed taken the latency and the bandwidth from the experimental data. Furthermore, in the plots there is the fitted model, which is reported also on the csv files.

The most interesting thing is the difference between the data obtained with OpenMPI and IntelMPI while measuring the connection inter-socket. As it shown in figure 4 (a) (b) using infiniband, the PingPong ran with IntelMPI has the peak at 8000 MB/s, while the other one performed with OpenMPI reaches 20000 MB/s. It is possible to observed similar difference also in intra-socket communication and with all the other network tried. The possible explanation of this behaviour is that with IntelMPI there are more cache missing than with OpenMPI.

Another thing is that all intra-node communications shown a big step in the bandwidth when the message size is around 1MB. That could be caused by the use of the L2 cache, and it is visible for instance in figure 4 (b). If it is specified to not to use the cache I obtain lower bandwidth but with a less steep jump, as it possible to see in figure 4 (c).

Moreover, again in intra-node communication we can observe that, after a certain point, there is drop of the bandwidth: probably, after the message size becomes larger than the L2 cache, there are a lot of cache missing. In addition, as it explained in the book *Introduction to High Performance Computing for Scientists and Engineers, G.H., G.W.*, the design of the benchmarks could affect the performance when the number of repetitions start decreasing.

In conclusion, the intra-node communications are harder to understand because of the cache missing and other behaviour releted to shared memory. In addition, OpenMPI behave slightly better than IntelMPI.



(a) IntelMPI (b) OpenMPI (c) OpenMPI without using cache

Figure 4: Difference behaviour between OpenMPI and IntelMPI while measuring the bandwidth.

Section 3

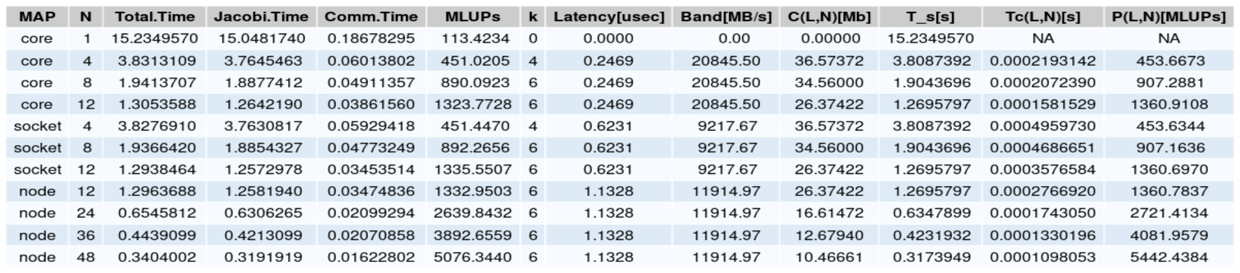
The tables 5, 6 summarize the data collected running the Jacobi solver on THIN and GPU nodes and their respective performance model keeping the size of the problem fixed to 1200^3. In particular I used the model discussed in class:

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

$$T_c(L,N) = \frac{C(L,N)}{B} + k\lambda$$

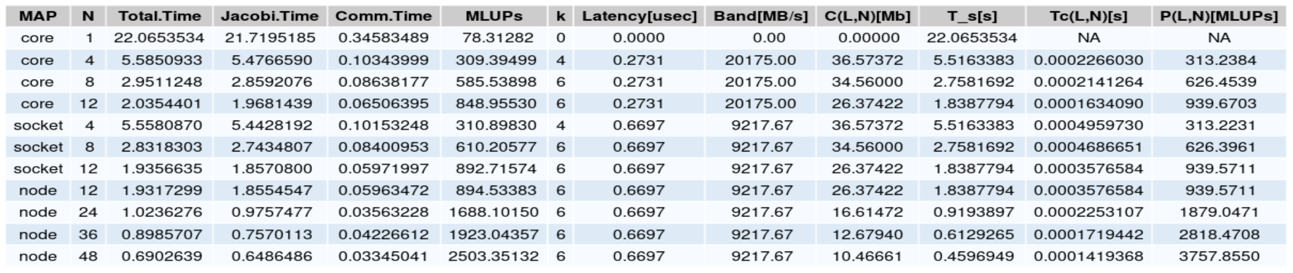
$$C(L,N) = 16kL^2$$

where N is the number of the processors, L is the size of each subdomain and T_s is the sequential time. T_s is calculated as the ratio between the Total Time with N equal to 1



MAP	N	Total.Time	Jacobi.Time	Comm.Time	MLUPs	k	Latency[usec]	Band[MB/s]	C(L,N)[Mb]	T_s[s]	Tc(L,N)[s]	P(L,N)[MLUPs]
core	1	15.2349570	15.0481740	0.18678295	113.4234	0	0.0000	0.00	0.00000	15.2349570	NA	NA
core	4	3.8313109	3.7645463	0.06013802	451.0205	4	0.2469	20845.50	36.57372	3.8087392	0.0002193142	453.6673
core	8	1.9413707	1.8877412	0.04911357	890.0923	6	0.2469	20845.50	34.56000	1.9043696	0.0002072390	907.2881
core	12	1.3053588	1.2642190	0.03861560	1323.7728	6	0.2469	20845.50	26.37422	1.2695797	0.0001581529	1360.9108
socket	4	3.8276910	3.7630817	0.05929418	451.4470	4	0.6231	9217.67	36.57372	3.8087392	0.0004959730	453.6344
socket	8	1.9366420	1.8854327	0.04773249	892.2656	6	0.6231	9217.67	34.56000	1.9043696	0.0004686651	907.1636
socket	12	1.2938464	1.2572978	0.03453514	1335.5507	6	0.6231	9217.67	26.37422	1.2695797	0.0003576584	1360.6970
node	12	1.2963688	1.2581940	0.03474836	1332.9503	6	1.1328	11914.97	26.37422	1.2695797	0.0002766920	1360.7837
node	24	0.6545812	0.6306265	0.02099294	2639.8432	6	1.1328	11914.97	16.61472	0.6347899	0.0001743050	2721.4134
node	36	0.4439099	0.4213099	0.02070858	3892.6559	6	1.1328	11914.97	12.67940	0.4231932	0.0001330196	4081.9579
node	48	0.3404002	0.3191919	0.01622802	5076.3440	6	1.1328	11914.97	10.46661	0.3173949	0.0001098053	5442.4384

Figure 5: Performance table of 3D Jacobi solve for thin nodes



MAP	N	Total.Time	Jacobi.Time	Comm.Time	MLUPs	k	Latency[usec]	Band[MB/s]	C(L,N)[Mb]	T_s[s]	Tc(L,N)[s]	P(L,N)[MLUPs]
core	1	22.0653534	21.7195185	0.34583489	78.31282	0	0.0000	0.00	0.00000	22.0653534	NA	NA
core	4	5.5850933	5.4766590	0.10343999	309.39499	4	0.2731	20175.00	36.57372	5.5163383	0.0002266030	313.2384
core	8	2.9511248	2.8592076	0.08638177	585.53898	6	0.2731	20175.00	34.56000	2.7581692	0.0002141264	626.4539
core	12	2.0354401	1.9681439	0.06506395	848.95530	6	0.2731	20175.00	26.37422	1.8387794	0.0001634090	939.6703
socket	4	5.5580870	5.4428192	0.10153248	310.89830	4	0.6697	9217.67	36.57372	5.5163383	0.0004959730	313.2231
socket	8	2.8318303	2.7434807	0.08400953	610.20577	6	0.6697	9217.67	34.56000	2.7581692	0.0004686651	626.3961
socket	12	1.9356635	1.8570800	0.05971997	892.71574	6	0.6697	9217.67	26.37422	1.8387794	0.0003576584	939.5711
node	12	1.9317299	1.8554547	0.05963472	894.53383	6	0.6697	9217.67	26.37422	1.8387794	0.0003576584	939.5711
node	24	1.0236276	0.9757477	0.03563228	1688.10150	6	0.6697	9217.67	16.61472	0.9193897	0.0002253107	1879.0471
node	36	0.8985707	0.7570113	0.04226612	1923.04357	6	0.6697	9217.67	12.67940	0.6129265	0.0001719442	2818.4708
node	48	0.6902639	0.6486486	0.03345041	2503.35132	6	0.6697	9217.67	10.46661	0.4596949	0.0001419368	3757.8550

Figure 6: Performance table of 3D Jacobi solver for GPU node

and N. I used the bandwidth B and the latency λ computed from the fitted model of the *Section 2*.

In THIN nodes, the performance $P(L,N)$ predicted form the theoretical model reflect quite well the real performance obtained from the computation. Instead, in the GPU node, there is an overestimation from the model. Probably this happens because of the *hyper-trading* enabled on this node: in the Jacobi program each core do the same kind of operations, while the *hyper-trading* is designed to work well when cores are doing different tasks.

The biggest difference between model and data is the communication time. In fact, the time $T_c(L,N)$ predicted from the model is hundred time smaller then the observed one. This could happen because the time of the model is computed using the peak performance of the bandwidth, while as we can see in the section 2, the real speed of the network depend from the size of the messages. However, the simple model used is enough good to predict the performance of the program.