Report of the first assigment of HPC 2021-2022

Guglielmo Padula

December 24, 2021

All the scripts and files mentioned in this report are available on my Github (https://github.com/guglielmopadula/HPCAssignment01) . All main scripts erase previously created files and recreate them after the calculations. All python scripts have been tested in a conda enviroment with all necessary modules installed.

1 Section1

For section1 the main script is called section1.sh All the programs have one version written in python and one in C++. All observations and statistics that will follow are related to the C++ version. At the end of the section, there will be a comparison between Python and C++.

First, I depict here a very short description of section1.sh

- Executes ./ring from -np 2 to 24, stores the mean execution time of the master process for each number of processors in a csv file and then calls Rscript that creates a graph.
- Executes mpirun -np 24 ./summatrix with all the input specified in cases.txt. After each run it saves the MPI time in a csv.
- Executes mpirun -np 24 ./summatrixfast 2400 100 100, mpirun -np 24 ./summatrixfast 1200 200 100, mpirun -np 24 ./summatrixfast 800 300 100 and stores all the results in another csv.

The graphs and the CSV mentioned will be shown in the subsections.

1.1 Ring part

The ring executable executes the main program (the one requested by the assignment) 10000 times and then takes the mean execution time of the master process. It then stores in a file (ring.txt) the printed lines time for every processor, as requested by the assignment. Mapping is by the socket.

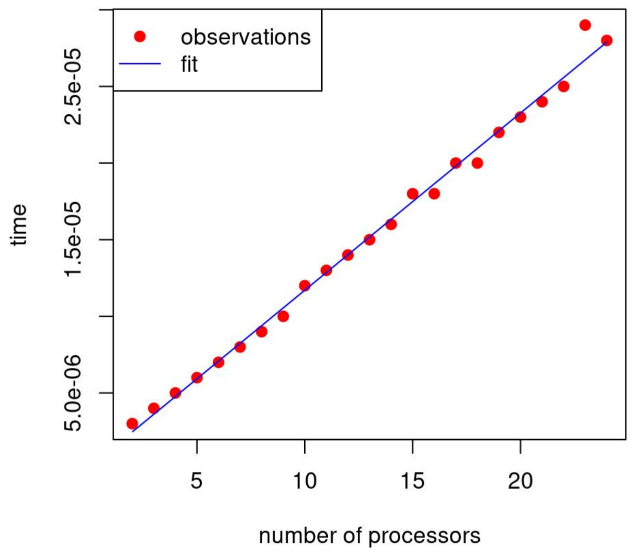

Figure 1: Graph of the rank 0 communication time in the ring program with different number of processors (generated by section1.sh)

The points represent the data and the line the best fitting associated with a linear function in the graph. I choose linear fitting since the number of messages exchanged grows linearly with the number of processors (from the point of view of a single processor). The linear model fits the data well.

1.2 Matrix part

For this part, I have coded two algorithms, *i.e.* summatrix and summatrixfast.

1.2.1 summatrix algorithm

The algorithm summatrix takes the sizes of the matrices and the distribution of the topology as input. Note that matrices are implemented as a custom class.

I give here a short description of the program:

- It initializes the two matrices to sum them randomly.
- Divides the matrix into blocks in a way that matrix blocks have the same spatial distribution of the topology (in 1D and 2D case the matrix blocks are created in the most possible cache-friendly way); if the submatrix cannot have the same distribution of the topology, the dimension of the matrix is increased until it is possible to have the same distribution. For this reason, matrices are implemented using std vectors which are dynamically allocated.
- Using a custom MPI Datatype and Scatterv, the master process sends the matrices to the other processes.
- Sum of the submatrices is computed.
- Master process executes a Gatherv.
- If the matrices size have been increased, they are restored to their original size.

size1	**size2**	**size3**	**ndim**	**dim1**	**dim2**	**dim3**	**runtime**
2400	100	100	3	4	3	2	1.90
2400	100	100	3	4	2	3	1.92
2400	100	100	3	3	2	4	1.69
2400	100	100	3	3	4	2	1.64
2400	100	100	3	2	3	4	1.95
2400	100	100	3	2	4	3	1.93
2400	100	100	3	6	2	2	1.64
2400	100	100	3	2	6	2	1.91
2400	100	100	3	2	2	6	1.97
1200	200	100	3	4	3	2	1.87
1200	200	100	3	4	2	3	1.93
1200	200	100	3	3	2	4	1.70
1200	200	100	3	3	4	2	1.64
1200	200	100	3	2	3	4	1.93
1200	200	100	3	2	4	3	1.93
1200	200	100	3	6	2	2	1.64
1200	200	100	3	2	6	2	1.91
1200	200	100	3	2	2	6	1.99
800	300	100	3	4	3	2	1.64
800	300	100	3	4	2	3	1.92
800	300	100	3	3	2	4	1.92
800	300	100	3	3	4	2	1.86
800	300	100	3	2	3	4	1.69
800	300	100	3	2	4	3	1.92
800	300	100	3	6	2	2	4.26
800	300	100	3	2	6	2	1.64
800	300	100	3	2	2	6	1.97
2400	100	100	1	24	0	0	1.60
1200	200	100	1	24	0	0	1.60
800	300	100	1	24	0	0	52.90
2400	100	100	2	3	8	0	2.09
2400	100	100	2	8	3	0	2.01
2400	100	100	2	6	4	0	1.96
2400	100	100	2	4	6	0	1.99
2400	100	100	2	12	2	0	2.02
2400	100	100	2	2	12	0	2.19
1200	200	100	2	3	8	0	2.06
1200	200	100	2	8	3	0	1.93
1200	200	100	2	6	4	0	1.96
1200	200	100	2	4	6	0	2.00
1200	200	100	2	12	2	0	1.91
1200	200	100	2	2	12	0	2.18
800	300	100	2	3	8	0	2.05
800	300	100	2	8	3	0	1.96
800	300	100	2	6	4	0	1.69
800	300	100	2	4	6	0	1.97
800	300	100	2	12	2	0	1.65
800	300	100	2	2	12	0	2.17

Table 1: Table of runtime of the summatrix algorithm for various combinations of matrix dimension (represented by size) and topology dimension (represented by dim) (generated by section1.sh)

Generally, the algorithm is faster when the dimensions of the topology divide the corresponding size of the matrix because the padding is costly. Best speed is achieved with ndim=1 because, in this case, every submatrix has elements contiguous in memory.

1.2.2 summatrixfast algorithm

The algorithm summatrixfast takes the sizes of the matrix and scatters the matrix using a Scatter without using any topology. If the total size of the matrix is not multiple of the number of processors, before the matrix initialization, the algorithm increases the size until it is multiple. This trick makes it possible to use static allocation because the program is written in C++. However, it does not respect the assignment requirements.

size1	**size2**	**size3**	**runtime**
2400	100	100	1.06
1200	200	100	1.05
800	300	100	1.05

Table 2: Table of runtime of the summatrixfast algorithm for various combinations of matrix dimension

The summatrixfast algorithm is faster than the summatrix algorithm with any topology. This behavior happens for several reasons:

- There are only collective operations in both algorithms, so topology has no effect.
- Scatter matrix in 2D or 3D blocks is costly because the submatrix elements are not contiguous in memory.
- Accessing static matrices is faster than accessing dynamic matrices.

1.3 Python versus C++

The python versions of the summatrix and summatrixfast are similar to the C++, with the main difference that matrices are implemented using numpy array. The ring program in python is practically identical to the C++ one.

	cpp	**python**
ring (24)	0.000028	0.000286
summatrixfast (2400 100 100)	0.97	0.78
summatrix (2400 100 100 1 24 1 1)	1.78	0.95

Table 3: Comparison of the cpp and python version of the algorithms (walltime) (generated why pythonvscpp.sh)

As we can see from the table, only algorithms, like ring, C++, are better than python in communication. When operations on matrix are involved, python is better because numpy (which I used for matrix operations) is compiled in C using the MKL, which is faster than the standard C++ code.

2 Section 2

Tests have been done with openmpi 4.1,1, gnu 9.3.0, and intel 20.4. The main script that generates files is section2.sh There are two graphs (one for latency and one for bandwidth), and a csv file for every combination tested. Then the runtime graph is fitted with the basic performance model, which fits the data well. For the fitting procedure at the start, I have chosen the least square model, but it produced negative latency, so I decided to fix the latency to the runtime corresponding to a send of one byte and then estimate the bandwidth using least squares. Examples with ucx are provided below.

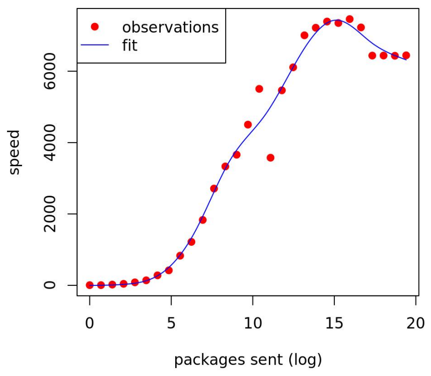

Figure 2: Graph of the speed of the intel ping pong benchmark on ucx with mapping by core (generated by section2.sh)

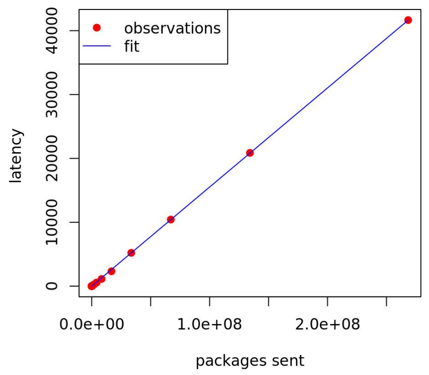

Figure 3: Graph of the speed of the intel ping pong benchmark on ucx with mapping by core (generated by section2.sh)

Summary of latency and bandwith is provided in section2.csv, and also below.

mpi implementation	**library-compiler**	**map**	**network**	**pml(opempi)/fabric(intel)**	**btl(openmpi)/ofi(intel)**	**bandwidth**	**latency**
openmpi	gnu-gcc	node	br0	ob1	tcp	2566.71	16.28
openmpi	gnu-gcc	node	br0	ucx	tcp	12202.7	0.99
openmpi	gnu-gcc	node	ib0	ob1	tcp	2526.63	16.24
openmpi	gnu-gcc	node	ib0	ucx	tcp	12204.1	1
openmpi	gnu-gcc	core	br0	ob1	tcp	5285.03	6.58
openmpi	gnu-gcc	core	br0	ucx	tcp	6490.32	0.2
openmpi	gnu-gcc	core	br0	ucx	vader	6492.73	0.19
openmpi	gnu-gcc	core	ib0	ob1	tcp	5280.9	6.18
openmpi	gnu-gcc	core	ib0	ob1	vader	4536.74	0.24
openmpi	gnu-gcc	core	ib0	ucx	tcp	6498.36	0.19
openmpi	gnu-gcc	core	ib0	ucx	vader	6491.32	0.24
openmpi	gnu-gcc	socket	br0	ob1	tcp	3327.23	8.09
openmpi	gnu-gcc	socket	br0	ucx	tcp	5690.43	0.46
openmpi	gnu-gcc	socket	br0	ucx	vader	5709.95	0.42
openmpi	gnu-gcc	socket	ib0	ob1	tcp	3353.99	9.32
openmpi	gnu-gcc	socket	ib0	ob1	vader	3998.46	0.56
openmpi	gnu-gcc	socket	ib0	ucx	tcp	5682.88	0.42
openmpi	gnu-gcc	socket	ib0	ucx	vader	5742.63	0.41
intelmpi	intel	contiguos	none	shm	none	2542.8	1.04
intelmpi	intel	contiguos	none	ofi	shm	2592.02	1.76
intelmpi	intel	contiguos	br0	ofi	sockets	1613.18	9.38
intelmpi	intel	contiguos	br0	ofi	tcp	1689.85	8.98
intelmpi	intel	contiguos	ib0	ofi	mlx	3938.98	0.45
intelmpi	intel	socket	none	shm	none	2866.96	0.87
intelmpi	intel	socket	none	ofi	shm	2932.01	1.46
intelmpi	intel	socket	br0	ofi	sockets	2658.49	7.08
intelmpi	intel	socket	br0	ofi	tcp	2671.8	5.46
intelmpi	intel	socket	ib0	ofi	mlx	4231.71	0.25

Table 4: Estimates of latency and bandwith for various tests of Ping Pong (generated by section2.sh)

2.1 Comments on openmpi

Note that ucx ignores btl parameters; in fact, results with ucx and mapping fixed are the same. As for the latency, fixed the other things, the pml with the least latency is ucx, followed by ob1. Fixed the other things, the btl with the least latency is tcp, followed by vader/shm. As for tcp, it is slightly faster with ethernet than with infiniband, which should not happen in theory. Taking the other things fixed, latency is lower with mapping by core, greater with mapping with socket and reaches its maximum with mapping by node, which is reasonable since the distance is more significant. As for the bandwith, with ob1 is lower with mapping by node, greater with mapping by socket and reaches its maximum with mapping by core. Using ucx best bandwith is achieved with mapping by node because the Infiniband network can send data directly without writing on cache (RDMA). Fixed the mapping, the maximum bandwidth is achieved by ucx and vader.

2.2 Comments on intelmpi

Socket mapping seems to be better for bandwidth and latency. The mlx fabrics (which runs on infiniband) bring the best performance.

2.3 intelmpi vs openmpi

Between the intel implementation and the openmpi one, there is no clear winner for bandwidth or latency because they use a different way of mapping processors and networks (openmpi uses btl and pml, and intelmpi uses fabrics).

2.4 Comment on the use of cache

Before converging, speed reaches a local maximum and then decreases because of a growth of cache references when the message is too big and then goes back to memory. A graph of the cache references is provided below for the ucx case, for which the behaviour is more evident.

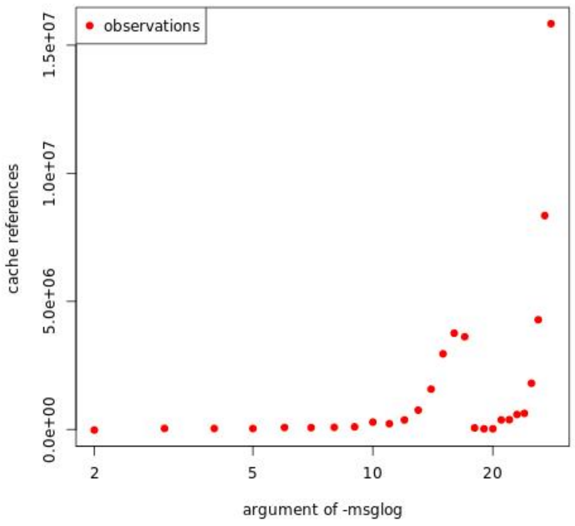

Figure 4: graph of cache references with growing message size (generated by cache.sh)

To decrease this phenomenon, all calculations were done with the parameter -off-cache -1. Below there is a comparison of the speed of ucx with the off-cache flag enabled and disabled (both with mapping by core and on infiniband).

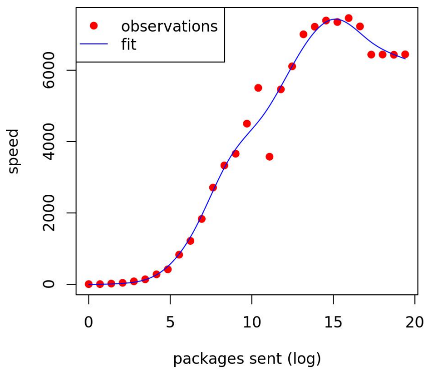

Figure 5: Graph of speed of ucx pingpong with off cache set (generated by section2.sh)

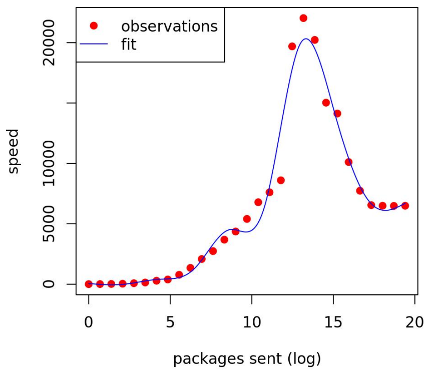

Figure 6: Graph of speed of ucx pingpong with off cache not set (generated by section2.sh)

3 Section3

The bash script section3.sh is designed to perform calculations using CPUs and section3gpu.sh using GPUs. Both scripts erase previous calculations and need the section 2 to be fully calculated to work. Calculations in a cpu node are provided in section3.csv and calculations in a gpu node are provided in section3gpu.csv (both files are created using R). All calculations are done with message passing on ucx.

N	**map**	**L**	**latency**	**band**	**timesing**	**k**	**c**	**tc**	**P(L_N) (estimated)**	**P(L_N) (real)**	**P(L_1)*n/P(L_N) (estimated)**	**P(L_1)*n/P(L_N) (real)**
4	socket	600	0.47	5673.19	1.8392	1	5.4932	0.001	447.7722	445.741	1.0005	1.0051
8	socket	600	0.47	5673.19	1.8392	1	5.4932	0.001	895.5443	889.023	1.0005	1.0079
12	socket	600	0.47	5673.19	1.8392	1	5.4932	0.001	1343.3165	1335.57	1.0005	1.0063
4	core	600	0.19	6153.98	1.8392	1	5.4932	9e-04	447.7906	446.305	1.0005	1.0038
8	core	600	0.19	6153.98	1.8392	1	5.4932	9e-04	895.5813	888.34	1.0005	1.0086
12	core	600	0.19	6153.98	1.8392	1	5.4932	9e-04	1343.3719	1308.06	1.0005	1.0275
12	node	600	0.98	12187.1	1.8392	1	5.4932	5e-04	1343.694	1334.59	1.0002	1.0071
24	node	600	0.98	12187.1	1.8392	1	5.4932	5e-04	2687.388	2644.2	1.0002	1.0166
48	node	600	0.98	12187.1	1.8392	1	5.4932	5e-04	5374.7759	5236.58	1.0002	1.0266

Table 5: Jacobi performance on a CPU node (the variables have the same meaning of the ones of the Jacobi Evaluation Model that we studied in class) (generated by section3.sh)

N	map	L	latency	band	timesing	k	c	tc	P(L_N) (estimated)	P(L_N) (real)	P(L_1)*n/P(L_N) (estimated)	P(L_1)*n/P(L_N) (real)
4	socket	600	0.19	6153.98	2.6657	1	5.4932	9e-04	309.0041	306.614	1.0003	1.0081
8	socket	600	0.19	6153.98	2.6657	1	5.4932	9e-04	618.0082	609.466	1.0003	1.0144
12	socket	600	0.19	6153.98	2.6657	1	5.4932	9e-04	927.0123	895.967	1.0003	1.035
24	socket	600	0.19	6153.98	2.6657	1	5.4932	9e-04	1854.0246	1695.04	1.0003	1.0942
48	socket	600	0.19	6153.98	2.6657	1	5.4932	9e-04	3708.0493	2518.44	1.0003	1.4729
4	core	600	0.19	6153.98	2.6657	1	5.4932	9e-04	309.0041	305.119	1.0003	1.0131
8	core	600	0.19	6153.98	2.6657	1	5.4932	9e-04	618.0082	583.743	1.0003	1.0591
12	core	600	0.19	6153.98	2.6657	1	5.4932	9e-04	927.0123	844.65	1.0003	1.0979

Table 6: Jacobi performance on a GPU node (the variables have the same meaning of the ones of the Jacobi Evaluation Model that we studied in class) (generated by section3gpu.sh)

For all the number of processor N, the grid is divided into N parts on z and not divided in x and y, because, in this way, communication operations are done with submatrices contiguous in memory, since Fortran stores matrices by column, and in this way (as section 1 shows), operations are faster. Actual performance is lower than the one estimated using the scalability model discussed in class, which is reasonable. Estimated performance is better with mapping by core, worse with mapping by socket, the worst with mapping by node because of latency increases (see section 2), but this accounts only marginally. Actual performance exhibits opposite behaviour, with only marginal differences. Efficiency decreases when the number of processors grows.

Performance with GPU nodes is worse than with CPU nodes, even when the mapping in the CPUs node is by nodes because hyperthreading causes high degradation, especially when all virtual threads are used (case N=48) and efficiency drops.

Methodological note: serial time is calculated from the performance (using Jacobi singular time directly results in over super scalability everywhere, which of course can't be).