Report of the first assigment of HPC 2021-2022

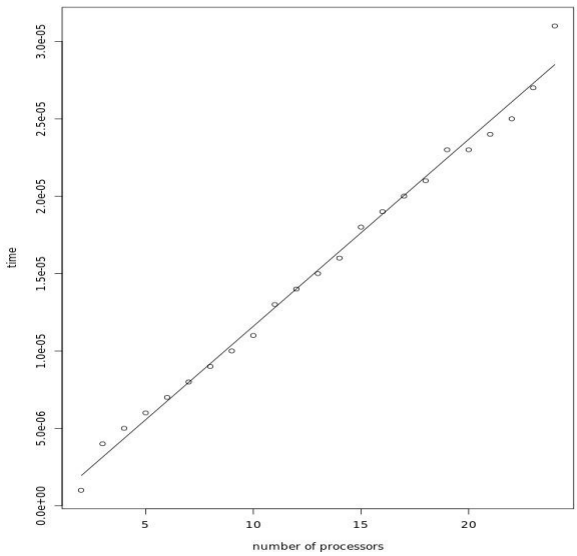
Guglielmo Padula

Section 1

Ring part

The ring executables executes the program 10000 times and then takes the mean execution time of process 0, then stores in a file (ring.txt) the printed lines time for every processor, as requested by the assigment.

The first part of section1.sh executes mpirun ./ring with -np from 2 to 24, stores the mean execution time of process 0 for each number of processors in a csv file (ring.csv) and then calls Rscript that creates a graph (ring.jpeg), which is provided below.



In the graph the points represent the data and the line the best fitting a line. I choose line since the number of messages exchanged grows with the the number of processors when considering from the point of view of a single, the models fits the data well.

Matrix part

The second part of section1.sh is divided in two algorithms: summatrix and summatrixfast. At first is executed mpirun -np 24 ./summatrix with all the input specified in cases.txt. After each run it saves the MPI time in a csv (matrix.csv). Then is executed mpirun -np 24 ./summatrixfast 2400 100 100, mpirun -np 24 ./summatrixfast 1200 200 100, mpirun -np 24 ./summatrixfast 800 300 100. The algorithm summatrix requires as input the sizes of the matrices and the distribution of the topology, it inizialises the matrix and divides the matrix in blocks in a way that matrix blocks have the same distribution of the topology using a custom MPI_Datatype and Scatterv; if the the submatrix can't have the same distribution of the topology, the dimension of the matrix is increased until is is possible to have the same distribution. For this reason matrix are implemented using std::vectors which are dinamically allocated. The algorithm summatrixfast wants as input, the sizes of the matrix, and simply 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 inizialization the algorithm increases the size until it is multiple. With this trick it is possible to use static allocation, because the program is written in C++, howewer it does not respect the assignment requirements. The summatrixfast algorithm is faster than the summatrix algorithm with any

topology, because there are only collective operations in both algorithms, so topology has no effect and also scatter matrix in blocks is costly because the submatrix elements are not contiguos in memory, and also access static matrices is faster then access dynamic matrices. For time measurement see summatrix.csv, which is provided below (the last three rows are related to summatrixfast, the others to summatrix, size1,size2,size3 are the sum of the matrix, ndim is the number of dimension of the topology, dim1,dim2,dim3 are the dimensions of the topology).

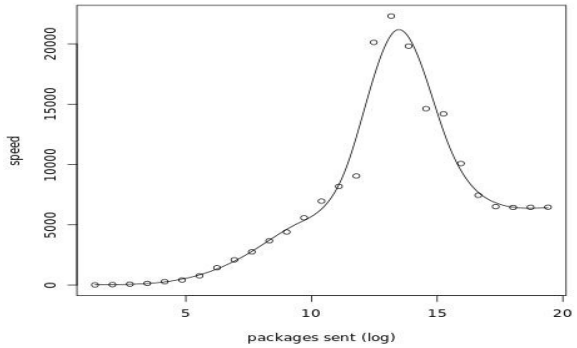
size1	**size2**	**size3**	**ndim**	**dim1**	**dim2**	**dim3**	**runtime**
2400	100	100	3	4	3	2	2.312
2400	100	100	3	4	2	3	2.35727
2400	100	100	3	3	2	4	1.90197
2400	100	100	3	3	4	2	1.89644
2400	100	100	3	2	3	4	2.39344
2400	100	100	3	2	4	3	2.40531
2400	100	100	3	6	2	2	1.85865
2400	100	100	3	2	6	2	2.3516
2400	100	100	3	2	2	6	2.38607
1200	200	100	3	4	3	2	2.29612
1200	200	100	3	4	2	3	2.34142
1200	200	100	3	3	2	4	1.87016
1200	200	100	3	3	4	2	1.82935
1200	200	100	3	2	3	4	2.35822
1200	200	100	3	2	4	3	2.35916
1200	200	100	3	6	2	2	1.83128
1200	200	100	3	2	6	2	2.32033
1200	200	100	3	2	2	6	2.40618
800	300	100	3	4	3	2	1.85288
800	300	100	3	4	2	3	2.32056
800	300	100	3	3	2	4	2.33307
800	300	100	3	3	4	2	2.26949
800	300	100	3	2	3	4	1.88213
800	300	100	3	2	4	3	2.3768
800	300	100	3	6	2	2	5.47409
800	300	100	3	2	6	2	1.8554
800	300	100	3	2	2	6	2.42482
2400	100	100	1	24	0	0	3.06764
1200	200	100	1	24	0	0	3.04557
800	300	100	1	24	0	0	3.05529
2400	100	100	2	3	8	0	2.51284
2400	100	100	2	8	3	0	2.4788
2400	100	100	2	6	4	0	2.37968
2400	100	100	2	4	6	0	2.37965
2400	100	100	2	12	2	0	2.46846
2400	100	100	2	2	12	0	2.62481
1200	200	100	2	3	8	0	2.46213
1200	200	100	2	8	3	0	2.36155
1200	200	100	2	6	4	0	2.37948
1200	200	100	2	4	6	0	2.38378
1200	200	100	2	12	2	0	2.36376
1200	200	100	2	2	12	0	2.62139
800	300	100	2	3	8	0	2.47932
800	300	100	2	8	3	0	2.37199
800	300	100	2	6	4	0	1.86139
800	300	100	2	4	6	0	2.40681

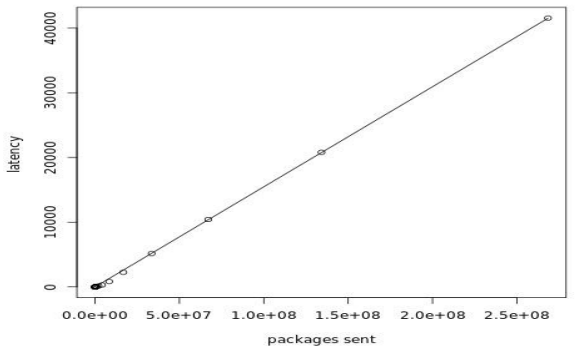
size1	size2	size3	ndim	dim1	dim2	dim3	runtime
800	300	100	2	12	2	0	1.85791
800	300	100	2	2	12	0	2.59505
2400	100	100	NA	NA	NA	NA	1.19679
1200	200	100	NA	NA	NA	NA	1.18617
800	300	100	NA	NA	NA	NA	1.1907

For the summatrix algorithm, the speed of the computation increases with the dimension of the topology, independent from the size of two matrices we want to compute the sum.

Section 2

The section2.sh scripts erases all previous calculations, does again the tests and generates all the csvs and the graphs (using Rscript) present in the section2 except for cache.csv and cache.jpeg, which are created by cache.sh. Before running section2.sh run preparation.sh, which recompiles the benchmark using gnu and intel libraries. Tests have been done with openmpi 4.0.3 and gnu 9.3.0, and intel 20.4. There are two graphs (one for latency and one for bandwith) and a csv file for every combination tested. Then graph of the runtime is fitted with the basic performance model, which fits well the data. For estimating the basi performance model least square is not ideal model because if often produces negative latency, so I choose to fix the latency to the runtime corresponding to a send of one byte, and then estimate the bandwith using least squares. Examples with ucx are provided below.



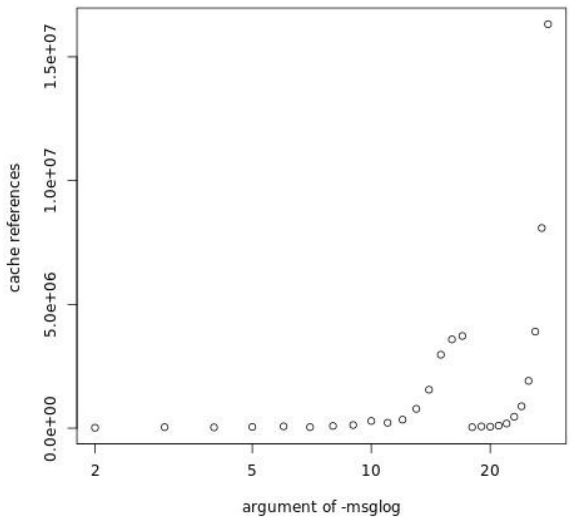


Summary of latency and bandwith is provided in section2.csv, and also below.

mpi implementation	library-compiler	map	network	pml(opempi) / fabric(intel)	btl(openmpi)/ofi(intel)	bandwidth	latency
openmpi	gnu-gcc	node	br0	ob1	tcp	2540.28	15.78
openmpi	gnu-gcc	node	ib0	ob1	tcp	2303.9	16.31
openmpi	gnu-gcc	core	br0	ob1	tcp	5332.87	5.43
openmpi	gnu-gcc	core	ib0	ob1	tcp	5295.17	5.36
openmpi	gnu-gcc	core	ib0	ob1	vader	4506.42	0.26
openmpi	gnu-gcc	socket	br0	ob1	tcp	3244.51	7.97

mpi implementation	library-compiler	map	network	pml(opempi) / fabric(intel)	btl(openmpi)/ofi(intel)	bandwidth	latency
openmpi	gnu-gcc	socket	ib0	ob1	tcp	3294.04	8.06
openmpi	gnu-gcc	socket	ib0	ob1	vader	3954.2	0.55
openmpi	gnu-gcc	node	ib0	ucx	none	3040.85	0.99
openmpi	gnu-gcc	core	ib0	ucx	none	1615.97	0.19
openmpi	gnu-gcc	socket	ib0	ucx	none	1415.22	0.40
intelmpi	intel	contiguos	none	shm	none	4208.28	0.43
intelmpi	intel	contiguos	none	ofi	shm	3930.78	1.51
intelmpi	intel	contiguos	br0	ofi	sockets	3193.57	9.89
intelmpi	intel	contiguos	br0	ofi	tcp	3262.93	7.84
intelmpi	intel	contiguos	ib0	ofi	mlx	5618.48	0.44
intelmpi	intel	socket	none	shm	none	5499.57	0.22
intelmpi	intel	socket	none	ofi	shm	4472.02	2.27
intelmpi	intel	socket	br0	ofi	sockets	5333.14	6.49
intelmpi	intel	socket	br0	ofi	tcp	5356.3	4.68
intelmpi	intel	socket	ib0	ofi	mlx	6201.84	0.24

As for the latency, the protocol with the latest latency is ucx, followed by shm/vader. As for tcp, it is slightly faster with ethernet than with infiniband. Taking the other things fixed, latency is lower with mapping by core, greater with mapping with socket and reaches is maximum with mapping by node, which is reasonable since the distance is greater. As for the bandwith, keep other things fixed it is lowest with mapping by node, greater with mapping by socket and greatest with mapping by node. Fixed the implementation and mapping the maximum bandwidth is achieved by ucx, at second we have shm/vader and then tcp. Keeping other things constant, the intel implementation has a slightly lower latenctìy then the openmpi implementation, while there is no clear winner for bandwith. Before converging, speed to a 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.



Comments on ucx do not depend on btl and eth because ucx ignores btl parameters, and default eth is ib0, as shown by the following two tables (the first for latency, the second for speed). All calculations are done with ucx pml and default mapping. The tables are generated by the file ucx.sh, using calls to R.

n	vader.br0	vader.ib0	tcp.br0	tcp.ib0
0	0.21	0.2	0.2	0.21

n	vader.br0	vader.ib0	tcp.br0	tcp.ib0
1	0.21	0.21	0.2	0.21
2	0.19	0.2	0.19	0.21
4	0.19	0.2	0.2	0.19
8	0.19	0.19	0.18	0.19
16	0.19	0.19	0.18	0.19
32	0.23	0.22	0.24	0.23
64	0.23	0.22	0.24	0.23
128	0.31	0.32	0.32	0.31
256	0.34	0.33	0.34	0.33
512	0.37	0.38	0.4	0.39
1024	0.51	0.5	0.5	0.51
2048	0.75	0.75	0.75	0.75
4096	1.13	1.12	1.12	1.12
8192	1.9	2.07	1.87	1.88
16384	3.21	3.3	2.96	2.97
32768	5.1	5.34	4.78	4.78
65536	8.61	8.77	8.17	8.15
131072	15.46	16.02	14.72	14.73
262144	14.35	15.13	13.73	12.6
524288	27.07	27.97	24.94	22.3
1048576	57.21	57.76	56.57	51.13
2097152	146.06	153.77	145.11	141.03
4194304	314.38	310.24	336.15	310.67
8388608	891.78	867.52	899.45	891.31
16777216	2335.23	2301.87	2357.69	2337.91
33554432	5330.24	5291.21	5337.25	5300.62
67108864	10625.59	10593.59	10636.97	10659.83
134217728	21228.53	21166.67	21245.14	21238.62
268435456	42559.07	42344.33	42398.91	41976.71

n	vader.br0	vader.ib0	tcp.br0	tcp.ib0
0	0	0	0	0
1	4.82	4.74	5.08	4.7
2	10.47	9.84	10.41	9.59
4	20.81	19.81	19.7	20.74
8	41.59	42.05	44.07	41.26
16	82.94	83.9	87.58	82.47
32	140.28	142.81	133.46	138.11
64	279.8	284.83	268.02	278.64
128	414.54	405.68	402.36	411.88
256	746.91	765.17	743.45	779.3
512	1401.07	1358.19	1293.88	1318.87
1024	2026.2	2058.72	2053.06	2012.98
2048	2742.63	2722.26	2718	2731.52
4096	3636.13	3643.44	3643.98	3650.54
8192	4321.4	3951.45	4375.79	4368.68
16384	5100.72	4961.37	5525.85	5519.14
32768	6427.02	6138.6	6857.86	6848.45
65536	7614.93	7470.44	8021.18	8040.26
131072	8479.99	8183.34	8903.71	8898.64
262144	18267.79	17330.17	19099.62	20797.13
524288	19367.34	18744.14	21025.8	23510.67
1048576	18327.35	18153.56	18535.98	20506.41
2097152	14357.75	13638.03	14451.89	14870.61

n	vader.br0	vader.ib0	tcp.br0	tcp.ib0
4194304	13341.39	13519.51	12477.6	13500.96
8388608	9406.58	9669.66	9326.39	9411.56
16777216	7184.38	7288.51	7115.97	7176.15
33554432	6295.1	6341.54	6286.84	6330.29
67108864	6315.78	6334.86	6309.02	6295.49
134217728	6322.52	6340.99	6317.57	6319.51
268435456	6307.36	6339.35	6331.19	6394.87

Section3

The bash script section3.sh is designed to perform calculations using cpus and section3gpu.sh using gpus. Both scripts erases previous calculations and need the section2 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), both are provided below.

CPU

N	map	L	latency	band	timesing	k	c	tc	P(LN) (estimated)	P(LN) real	P(L1)*n/P(LN) (estimated)	P(L1)*N/P(LN) (real)
4	socket	1200	0.4	1412.29	14.5376895646529	2	43.9453125	0.831116351811597	428.907548890201	450.650205157	1.05716976883537	1.00616418038472
8	socket	1200	0.4	1412.29	14.5376895646529	3	65.9179687 5	1.2466745277174	835.231224574488	892.081289892	1.08575465325306	1.01656227845983
12	socket	1200	0.4	1412.29	14.5376895646529	3	65.9179687 5	1.2466745277174	1252.84683686173	1332.04420952	1.08575465325306	1.02120055266497
4	core	1200	0.2	1617.46	14.5376895646529	2	43.9453125	0.427169334944914	440.485066997669	451.789859298	1.02938357797814	1.00362609956882
8	core	1200	0.2	1617.46	14.5376895646529	3	65.9179687 5	0.640754002417371	868.573493174354	888.188890046	1.04407536696672	1.02101726196669
12	core	1200	0.2	1617.46	14.5376895646529	3	65.9179687 5	0.640754002417371	1302.86023976153	1324.34338287	1.04407536696672	1.02713865643223
12	node	1200	1	3039.7	14.5376895646529	3	65.9179687 5	3.02168568238642	1126.20126552249	1322.38407887	1.20785185080 12	1.02866051147439
24	node	1200	1	3039.7	14.5376895646529	3	65.9179687 5	3.02168568238642	2252.40253104499	2626.48237001	1.20785185080 12	1.03582213112728
48	node	1200	1	3039.7	14.5376895646529	3	65.9179687 5	3.021685682386	4504.80506208997	4459.60636051	1.20785185080 1	1.22009358940 8

N	map	L	latency	band	timesing	k	c	tc	P(LN) (estimated)	P(LN) real	P(L1)*n/P(LN) (estimated)	P(L1)*N/P(LN) (real)
								42			2	58

GPU

N	map	L	latency	band	timesing	k	c	tc	P(LN) (estimated)	P(LN) real	P(L1)*n/P(LN) (estimated)	P(L1)*N/P(LN) (real)
4	socket	1200	0.4	1412.29	21.050288328 1634	2	43.9453125	0.831116351811597	301.251083804165	309.904167737	1.039482421278 75	1.010458195164 87
8	socket	1200	0.4	1412.29	21.050288328 1634	3	65.917 96875	1.246674527717 4	591.273073163095	609.140745081	1.059223631918 12	1.028153866029 63
12	socket	1200	0.4	1412.29	21.050288328 1634	3	65.917 96875	1.246674527717 4	886.909609744642	895.533496369	1.059223631918 12	1.049023427739 78
24	socket	1200	0.4	1412.29	21.050288328 1634	3	65.917 96875	1.246674527717 4	1773.81921948928	1688.48041021	1.059223631918 12	1.112758682109 86
48	socket	1200	0.4	1412.29	21.050288328 1634	3	65.917 96875	1.246674527717 4	3547.63843897857	2498.30525482	1.059223631918 12	1.504116626588 11

Actual performance is better than the one estimated using the scalability model discussed in class. Performance is better with mapping by core, lower with mapping by socket, and lowest with mapping by node, because of latency increases (see section 2). With the mapping fixed performance worsens when the nodes increase, because communication time increases.

Performance with GPU nodes is worst then the one with CPU nodes, even when the mapping in the CPUs node is by nodes, because hyperthreading causes high degradation.

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).