

Assignment 1

Niccolò Tosato

Repository with code and scripts: https://github.com/NiccoloTosato/HPC_assignment1

Section 1: MPI programming

Section 1.1: 1D ring

The ring has been implemented using non-blocking communication routines and 1D virtual topology with periodic boundaries. The non-blocking implementation leads to a linear growth of the execution time of the program, indeed the execution time of a single iteration can be modeled roughly as a *double PingPing*. The real performance will be of course worse than the ideal one, since the PingPing takes into account only 2 processes and does not consider crowded configuration with more than 2 processes. However, we have a lower bound ideal model to estimate the expected performance.

Software stack and measure setup.

The performances of the program are measured over 10000 iterations, using **UCX** and **InfiniBand**, across *core*, *socket* and *node*. Across two nodes the *mlx5_0* hardware interface is used. Times of theoretical models have been taken from Intel® MPI benchmark PingPing (Figure 1). Program can be compiled by Makefile with different options. Times have been measured on rank zero process, it is expected to be slower than the other ranks due topology reasons. Using *ldd* on executable it is possible to report specific library used.

```
[s271550@login ring]$ ldd ring.x
linux-vdso.so.1 => (0x00007ffce79ef000)
libm.so.6 => /lib64/libm.so.6 (0x00007f9e9dc88000)
libmpi.so.40 => /opt/area/shared/programs/x86_64/openmpi/4.0.3/gnu/9.3.0/lib/libmpi.so.40 (0x00007f9e9d964000)
```

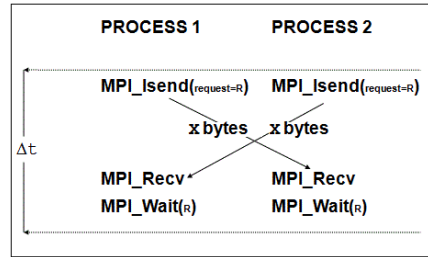


Figure 1: PingPing structure from Intel® MPI Benchmarks User Guide.

Map by node model

The network model across two nodes may take into account the latency of the network (dominated by the switch) and the number of processes involved. Each iteration will be lower bounded by the network.

$$Time = N_{procs} \cdot \lambda_{network}$$

The estimated latency between two node using the Intel MPI benchmark is a bit lower than the one declared by switch constructor. This lower latency holds only when few communications are going on. With multiple processes involved, a more realistic declared latency of 1.35 microseconds leads to a more accurate model. Thus the model will be experimentally outperformed when $N_{procs} = 2$. When the number of processes grow, it can be seen a slight slowdown of the actual implementation.

Across two nodes the latency can be estimated optimistically as $\lambda_{network} = 1.01\mu Sec$.

```
#-----
# Benchmarking PingPing
# #processes = 2
#-----
#bytes  #repetitions  t[usec]  Mbytes/sec
0        1000        1.00      0.00
1        1000        1.00      1.00
2        1000        1.01      1.98
4        1000        1.01      3.94
```

We expect the time to be $2 \cdot \lambda_{network}$ for each iterations, but experimentally this model doesn't hold. The experimental evidence suggests an iteration time of $\lambda_{network}$. My guess is that 2 consecutive messages are merged into one unique request routed to the switch. This behaviour can be confirmed using Intel MPI Benchmark **Exchange** again, that implements 1D non periodic ring (Figure 2).

```
#-----
# Benchmarking Exchange
# #processes = 36
#-----
#bytes  #repetitions  t_min[usec]  t_max[usec]  t_avg[usec]  Mbytes/sec
0        10000        1.46        1.46        1.46        0.00
1        10000        1.45        1.45        1.45        2.75
2        10000        1.45        1.45        1.45        5.50
4        10000        1.46        1.46        1.46        10.97
```

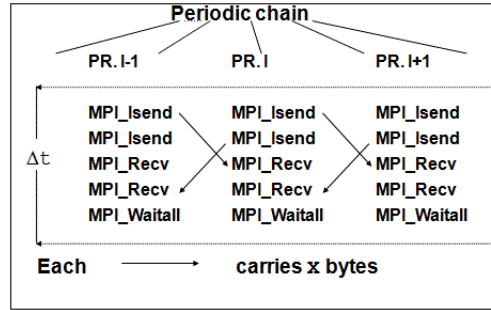


Figure 2: Exchange benchmark structure from Intel® MPI Benchmarks User Guide.

Map by socket model

With the socket round robin binding selected, we can model the ring behaviour as following, with the usual Intel MPI benchmark estimation.

$$Time = N_{procs} \cdot \lambda_{socket} \cdot 2$$

Again when the sockets become crowded the real performance is worse than the expected model.

Across two sockets the estimated time is $t = 0.49\mu Sec$

```
#-----
# Benchmarking PingPing
# #processes = 2
#-----
#bytes  #repetitions  t[usec]  Mbytes/sec
0        1000        0.49      0.00
1        1000        0.48      2.06
2        1000        0.49      4.12
4        1000        0.49      8.18
```

Map by core model

When mapping by core the processes, two factors have to be taken into account: how many processes are spawned and where. When N_{procs} is less or equal than the number of cores in a single socket, the expected execution time is bounded by the core communication. When the first socket is filled, the first process on the second socket will be the slowest among all processes. Indeed its neighbors will be placed in the other socket. One iteration is long as the slowest process communication time. Experimentally a spike when $N_{procs} = 13$ is clearly visible. Counterintuitively, the performance increases when the number of processes is greater than $N_{procs} = 13$ and other processes are spawned in the other socket.

Indeed, the slowest process has one neighbor on the same socket and one on the other socket, thus the overall iteration time for the slowest process is $\lambda_{core} + \lambda_{socket}$.

$$Time = \begin{cases} N_{procs} \cdot \lambda_{core} \cdot 2 & N_{procs} \leq N_{cpucore} \\ N_{procs} \cdot (\lambda_{core} + \lambda_{socket}) & N_{procs} > N_{cpucore} + 2 \\ N_{procs} \cdot \lambda_{socket} \cdot 2 & N_{procs} = N_{cpucore} + 1 \end{cases}$$

Across two sockets the estimated time is $t = 0.23\mu Sec$

```
#-----
# Benchmarking PingPing
# #processes = 2
#-----
```

#bytes	#repetitions	t[usec]	Mbytes/sec
0	1000	0.23	0.00
1	1000	0.23	4.27
2	1000	0.24	8.44
4	1000	0.23	17.18

Experimental results compared with theoretical model

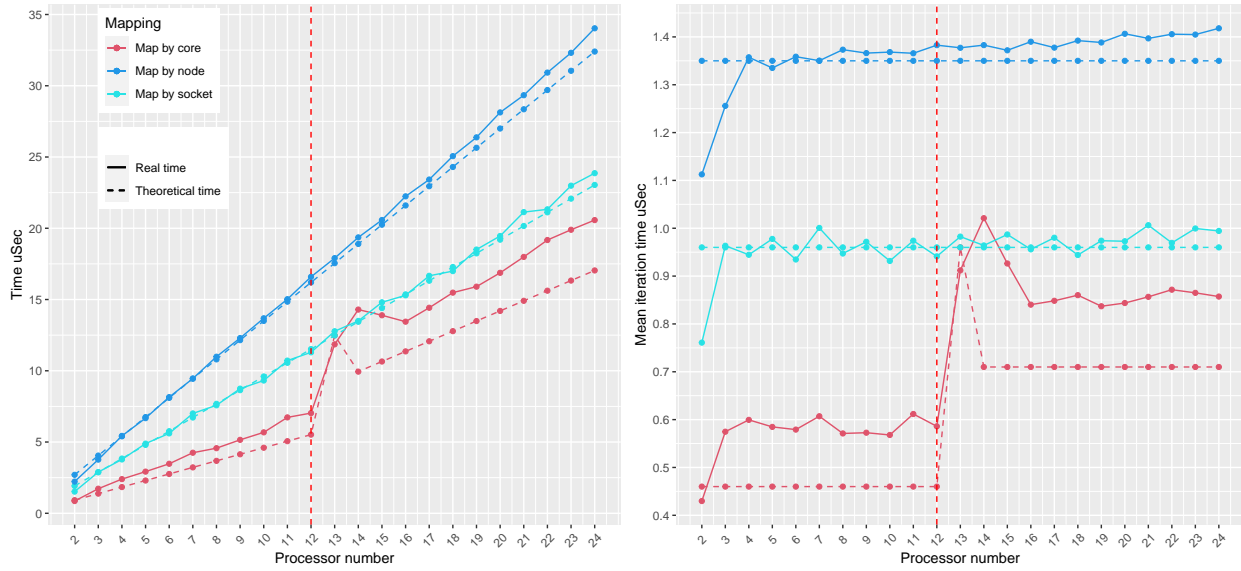


Figure 3: Ring on THIN node up to 24 cores

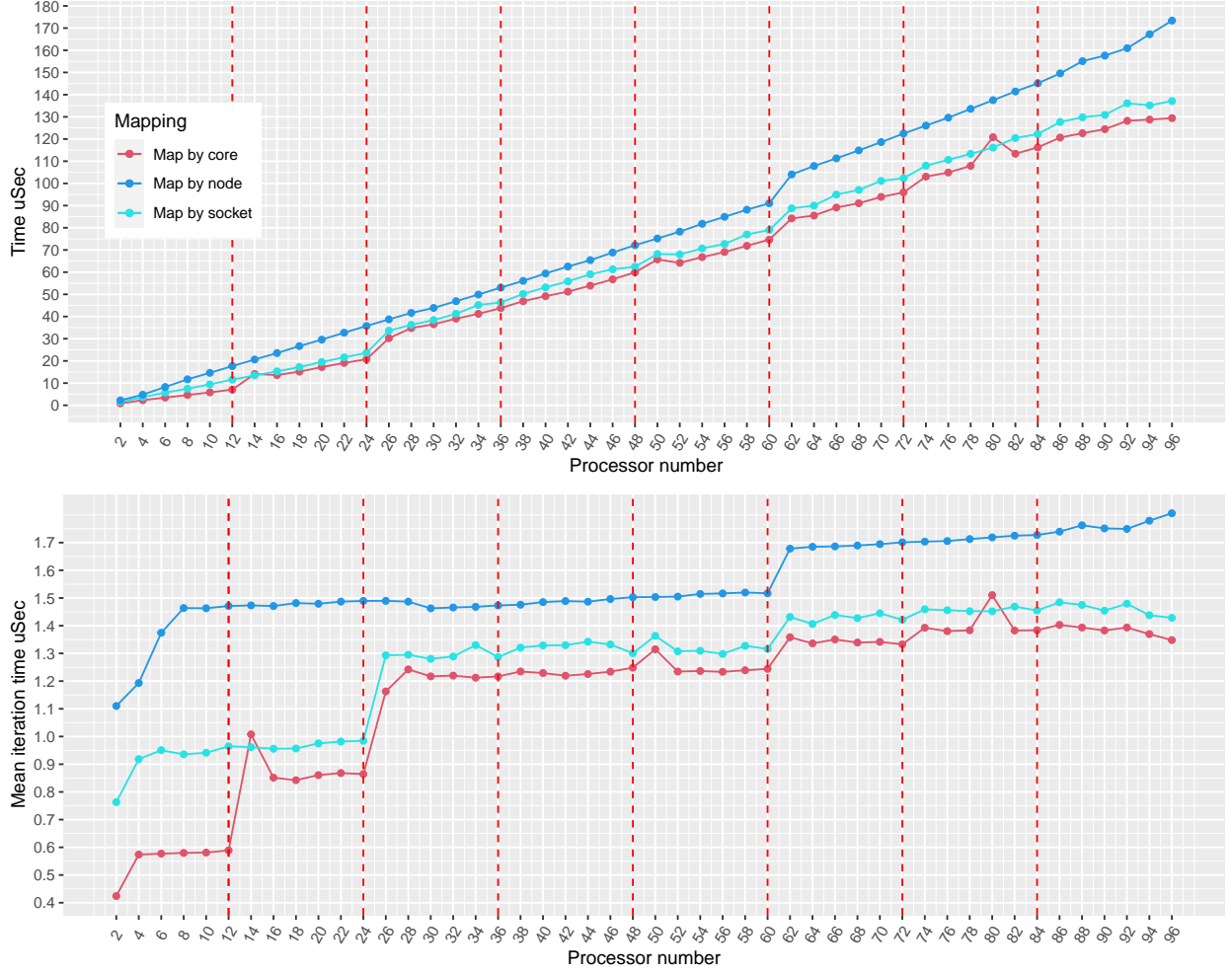


Figure 4: Ring on THIN node up to 96 cores and 4 nodes

By running the ring across 4 nodes, the pattern above can be visualized. After 24 processes, when the socket or the node is saturated the changes on the average iteration time are smaller. This is due to the fact that the communication time between nodes is the largest and it is more evident than the intra-socket or the intra-core ones. Moreover, with more than 24 processes we have at least one intra-node communication going on, the step confirms that.

Table 1: Matrix sum timings

N° procs	Scatter (S)	Gather (S)	Parallel (S)	Total (S)	N° procs	Scatter (S)	Gather (S)	Parallel (S)	Total (S)
1	16.82058	8.439177	2.635232	49.81	1	16.70198	8.388258	2.620498	49.44
4	13.46379	6.624008	0.657729	45.14	4	13.73519	6.798873	0.638340	46.35
8	14.08911	6.949652	0.335095	46.09	8	14.53400	7.174536	0.329786	47.14
12	14.10777	6.960865	0.223283	46.18	12	14.56708	7.080100	0.218447	46.70
16	15.14609	7.480717	0.161398	46.48	16	15.46470	7.604523	0.162430	46.82
20	14.59633	7.137715	0.131786	45.64	20	14.66246	7.109426	0.127266	45.37
24	15.10795	7.400281	0.109368	46.52	24	15.11786	7.318925	0.108059	46.20

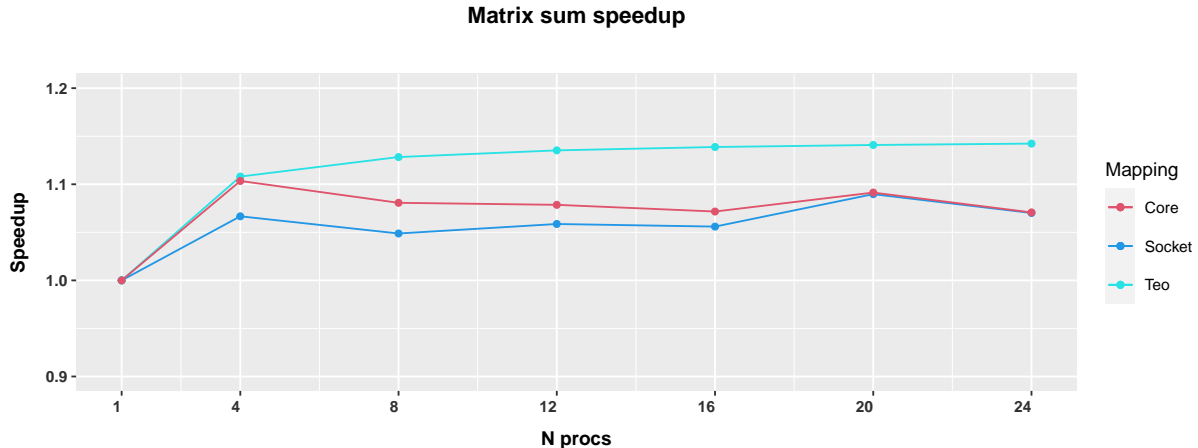
Section 1.2: Matrix sum

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.

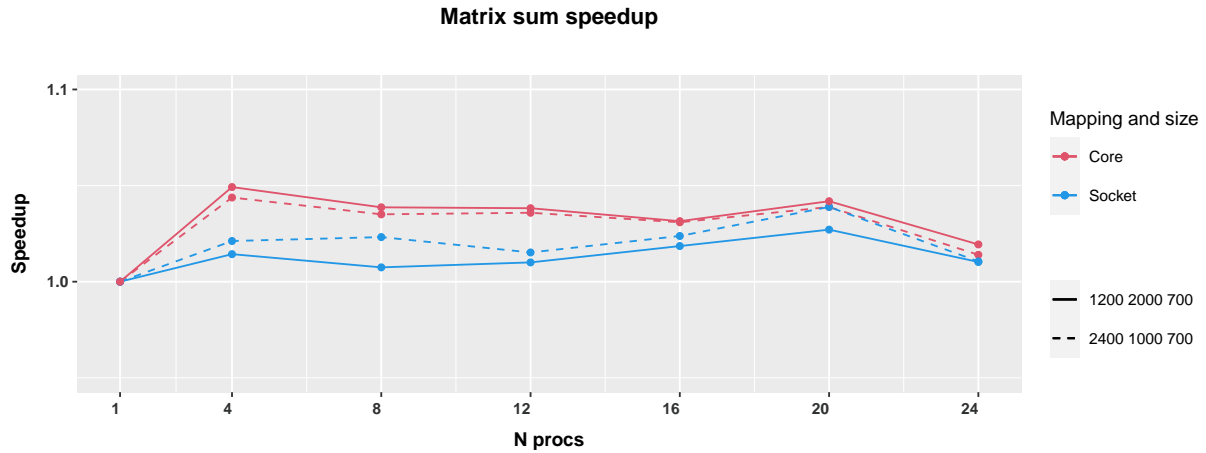
The shape of the array doesn't make any difference on how the array is represented in memory. Also the virtual topology and its relative domain decomposition do not have any impact. Indeed, we can assume that the problem is not sensible to any topology, and thus there is no need of communication between neighbors. The most efficient way to communicate and sum between matrices is to use MPI_Scatterv and MPI_Gatherv collective routines.

The expectations of this code are not so high in terms of scalability: the sum of two matrices containing n elements, requires $2n$ memory read accesses, n memory to write, a lot of communication and **only** n floating point operations (in other words: like Level 1 BLAS operation we can expect matrix sum at least to be memory bounded). Parallel portion of code is smaller with respect to the serial one. Using Ahmdal's law, a prediction of speed up bound can be made.

By testing the parallel code with two $2400 \times 1000 \times 700$ matrices of double values and by using only one processor, it can be seen that the parallel part takes only 2.63 seconds of the overall runtime of 49 seconds, representing about 5% of execution time (Elapsed is measured using `/usr/bin/time` and detailed Scatter and Gather with `MPI_Wtime()`). Total execution takes into account the matrix initialization and the error checking. The following graph represents the theoretical maximum speedup supported by Ahmdal's law assuming only 5% roughly of parallel code, while communication and serial code in the remaining part of the program is fixed with respect to the number of processors. The theoretical maximum speedup modelled is a good approximation and it catches the trend between [1,24] processes.



Moreover I've implemented a 3D matrix sum program that includes a 1D/2D/3D domain decomposition that matches virtual topology and that uses collective operations, as initially requested. This program leads to a very high overhead, since a collective operation (in general communication routine) needs contiguous memory region to communicate each subdomain to its worker. To achieve this an unrolling is performed for each subdomain. Once the worker has received the subdomain, it elaborates the array and sends back the summed array to the master processor. The master processor needs to reordinate the subdomains and to assemble the final matrix. Like in the first program, this code show an even lower performance. In this latter case we have some changes in performances with respect to matrices shape and domain decomposition, but this is due to matrix preparation overhead (buffering of subdomains before scatter it) and it is not due communication routine.



Section 2: MPI point to point performance

In order to measure MPI point to point performance, the Intel MPI benchmarks is used. By looking to Intel's documentation benchmark works as follow:

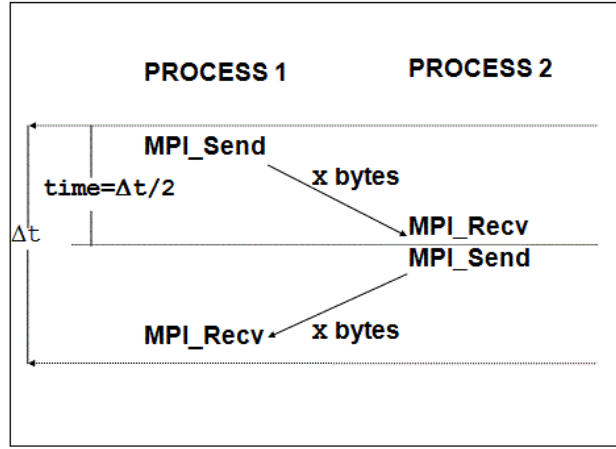


Figure 5: PingPong structure from Intel® MPI Benchmarks User Guide.

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 10 times and **openMPI 4.0.3** has been used. One entire node was reserved in order to reduce possible sources of noise in the measurements. The **pml** involved in the benchmarks are **OB1** and **UCX**. The **btl** used are **tcp** and **vader**. When the measurements across nodes were performed also different network with different protocols have been selected: 25 Gbit Ethernet and 100 Gbit InfiniBand through Mellanox network switch.

The following graphs show the behaviour inside the same node, i.e mapping the processes across two sockets or in the same socket. Mapping the processes in the same socket shows often a better performance, as expected. The behaviour before the asymptotic plateau is strange and shows very different performances among different implementations, the main cause of that is due to the cache. Before analyzing the performance we need to know more about the node topology, as explained in figure 6.

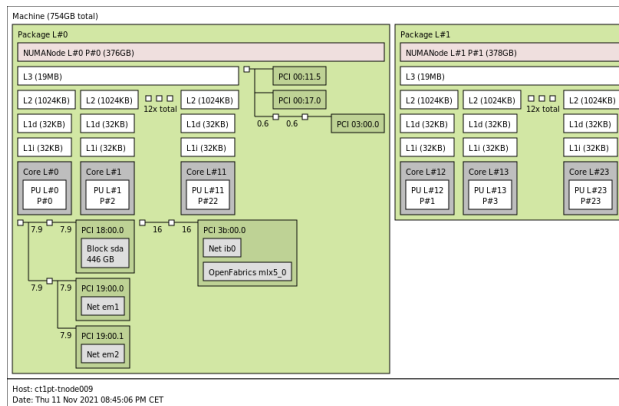
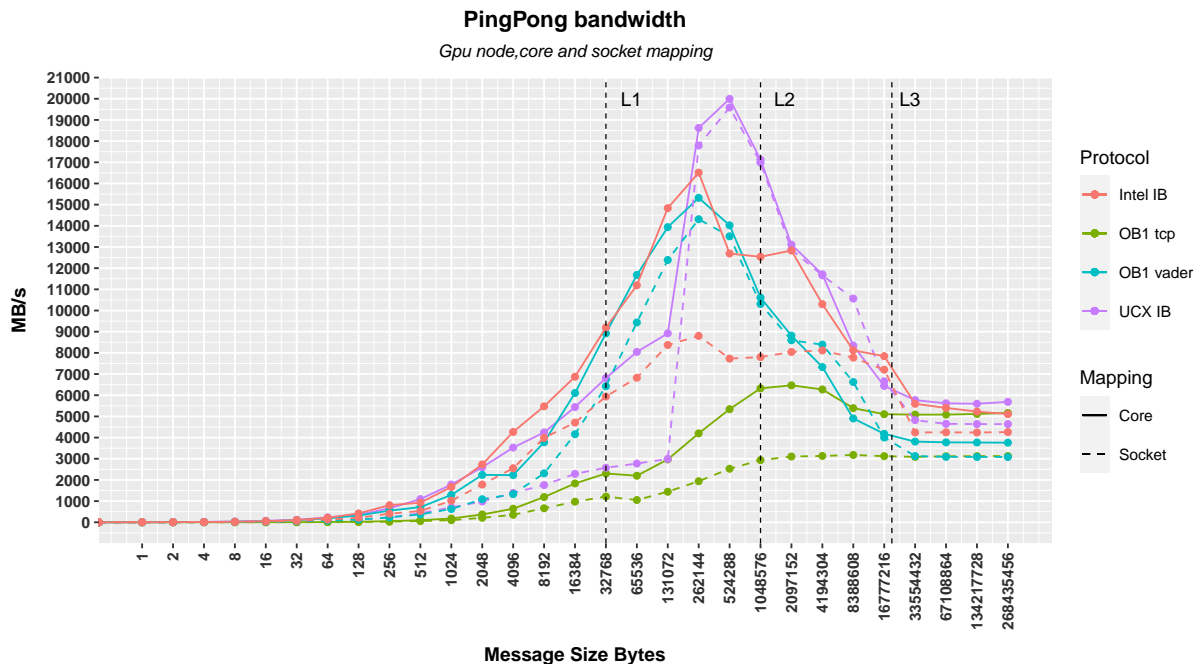
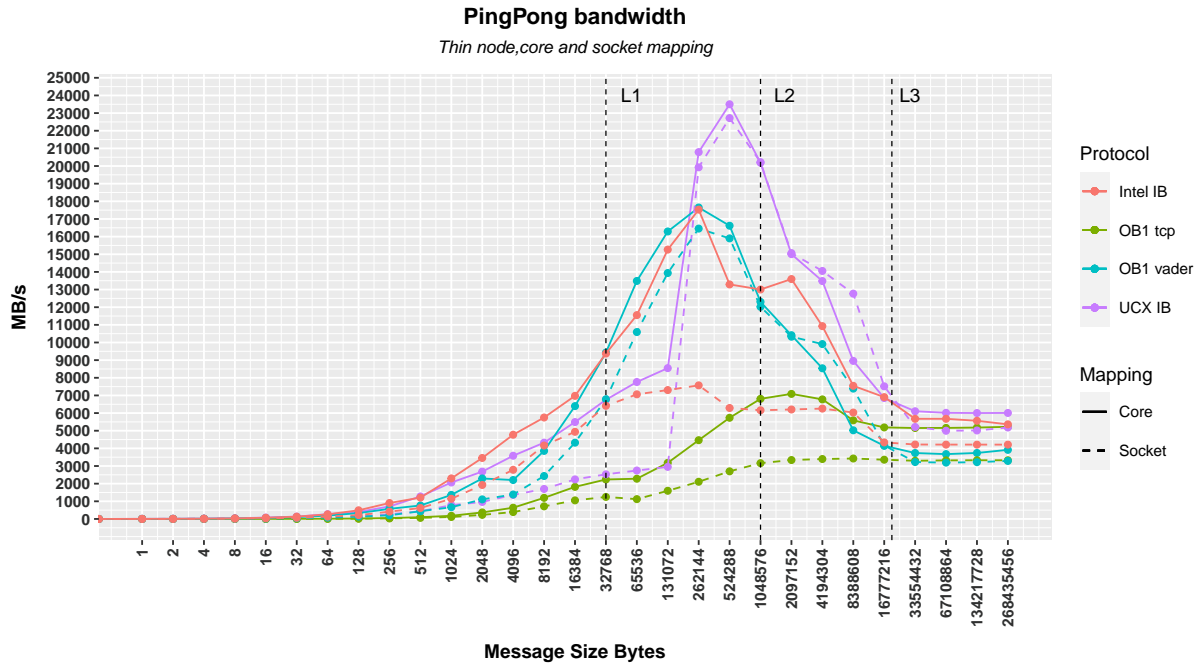
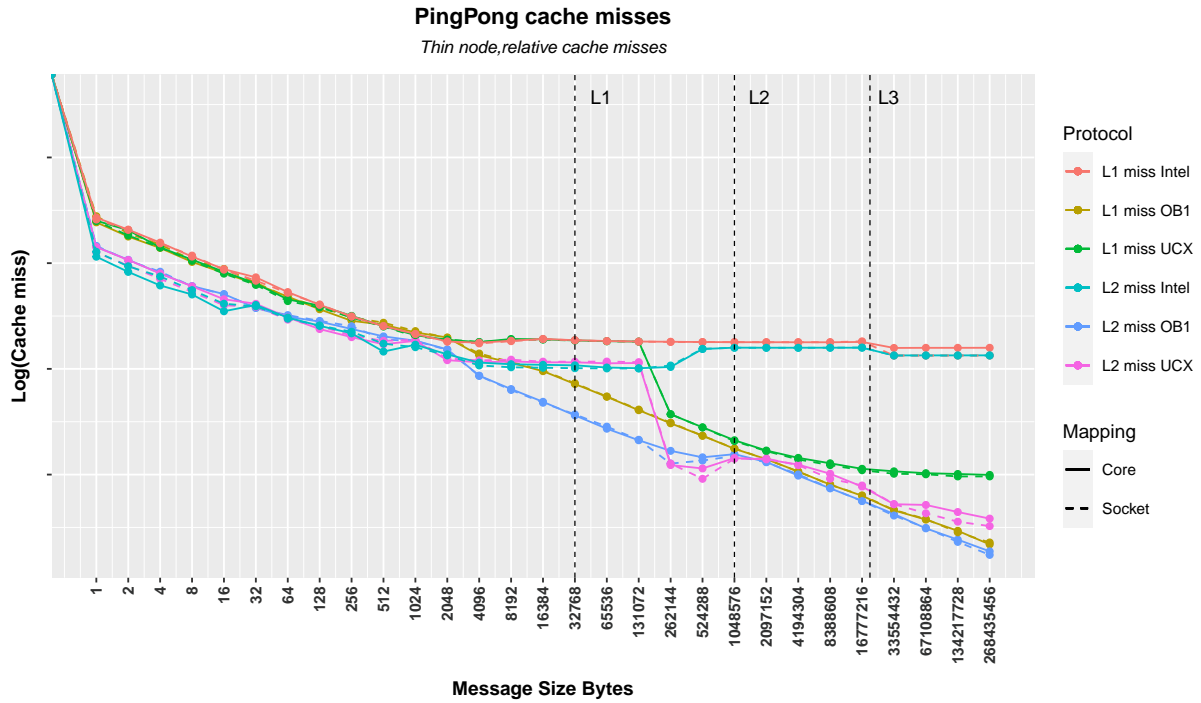
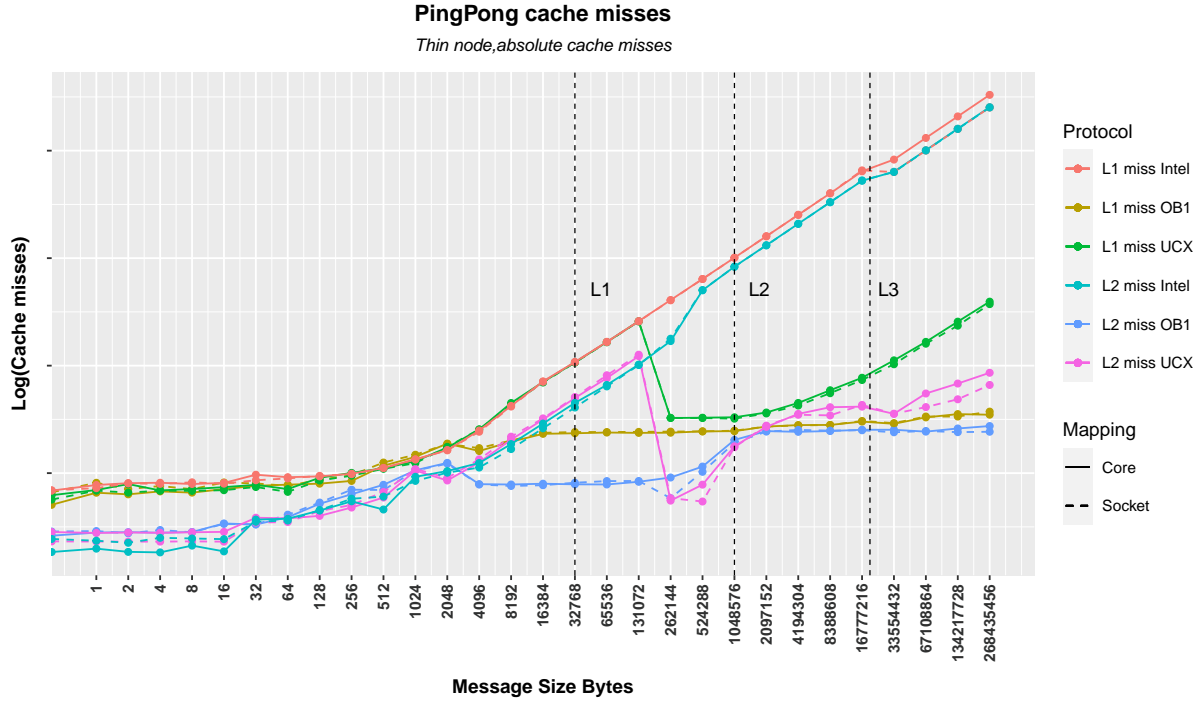


Figure 6: lstopo output on THIN node.

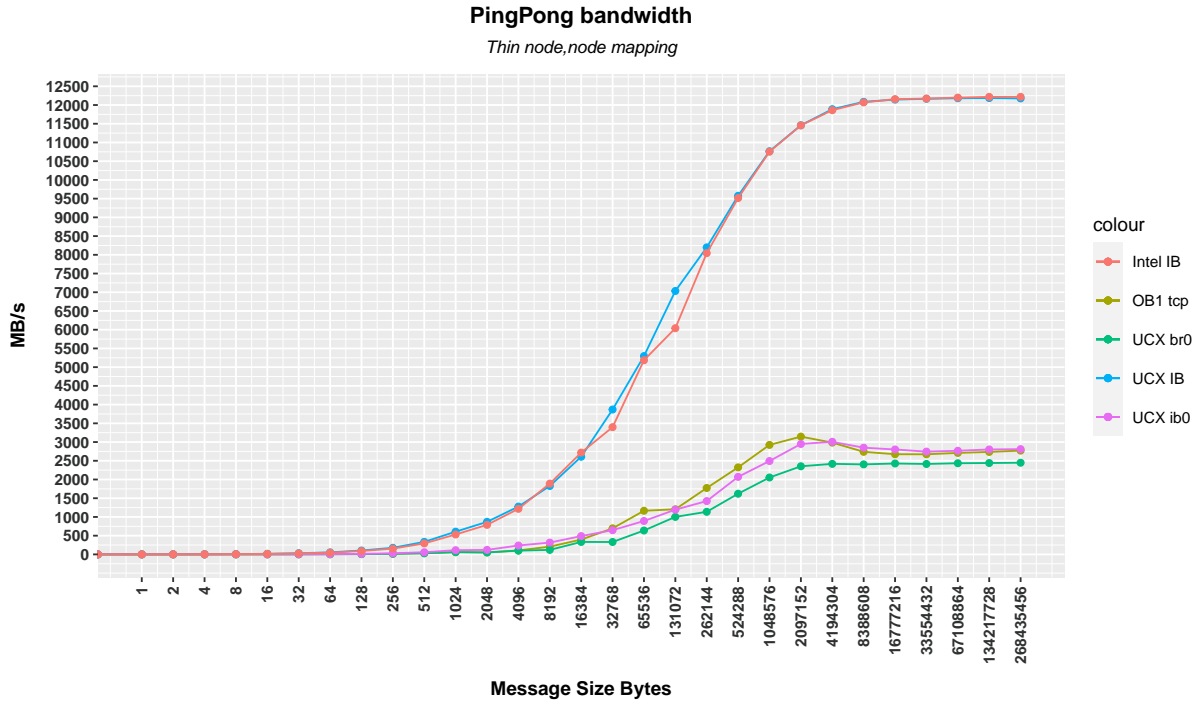
Cache sizes are reported on the following graphs with vertical black dashed lines. As it can be noticed in the following two graphs, the bandwidth behaviour appears strange before 16 MB included. After that point, the behaviour is stable because the message size is larger than all the caches. The larger cache is **L3** with 19 MB. The effect of **L2** becomes clear after 1 MB, after it all implementations start losing bandwidth.

The effect of **L1** is not clearly visible from the bandwidth due to the latency. To infer with more accuracy about the cache effect we can perform some profiling test using hardware counters. I have modified Intel MPI Benchmark PingPong injecting some code in order to inspect L1 data cache misses and L2 cache misses through PAPI. I tracked all cache misses on MPI_Send and MPI_Recv routine of rank 0, after some cycles of warmup. The code is available here https://github.com/NiccoloTosato/HPC_assignment1/tree/main/mpi-benchmarks_mod. The cache misses are reported on absolute value and normalized with respect to the message size. Before 128 KB UCX shows poor performance with respect to other protocols, while after that size a huge speed up is present. This fact will be explained later. Intel InfiniBand also shows poor performance with respect to UCX implementation, this behaviour is explained with cache too.





The behaviour of Intel implementation can be seen in a clearer way. It has the largest cache misses number among all other configurations. The bandwidth spike of **UCX** after 128 KB, can be explained by **L1** and **L2** drop in terms of cache misses. **OB1** implementation seems to be the most cache friendly *PML*. Disclaimer: those graphs must be used in a qualitative way and not in a quantitative way.



Inter-node benchmark point out the network performance and topology. Two physical networks are available, 25 Gbit Ethernet and 100 Gbit InfiniBand. The PCI-E devices are visible from lstopo. We can physically see that em1, em2 are bonded together, creating the interface bond0. To infer this configurations we can use ifconfig and ip link command.

```
[s271550@ctipt-tnode007 etc]$ ifconfig
bond0: flags=5187<UP,BROADCAST,RUNNING,MASTER,MULTICAST> mtu 1500
        ether 34:80:0d:4e:55:68 txqueuelen 1000 (Ethernet)

br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
        ether 34:80:0d:4e:55:68 txqueuelen 1000 (Ethernet)

em1: flags=6211<UP,BROADCAST,RUNNING,SLAVE,MULTICAST> mtu 1500
        ether 34:80:0d:4e:55:68 txqueuelen 1000 (Ethernet)

em2: flags=6211<UP,BROADCAST,RUNNING,SLAVE,MULTICAST> mtu 1500
        ether 34:80:0d:4e:55:68 txqueuelen 1000 (Ethernet)

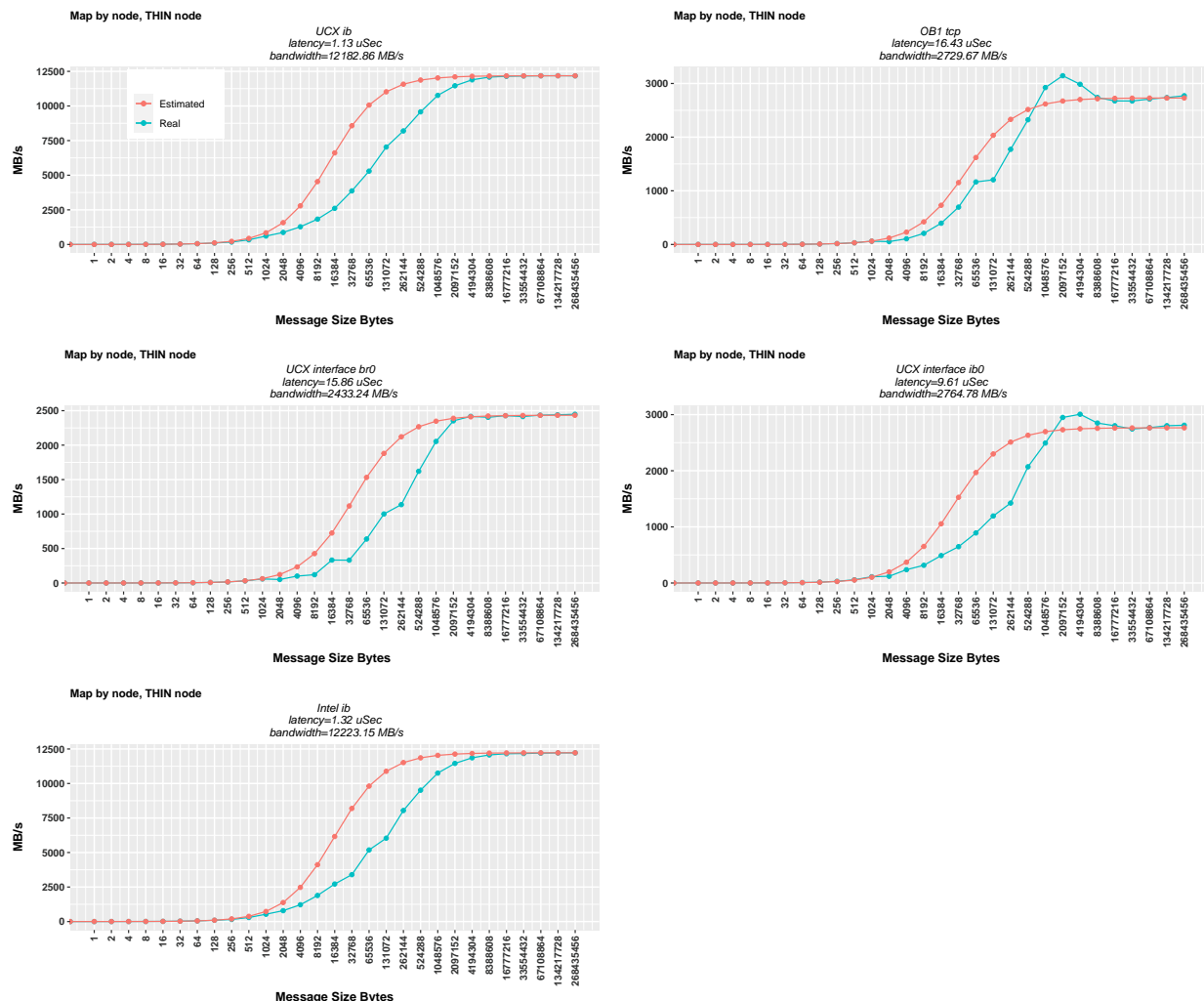
ib0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 2044
Infiniband hardware address can be incorrect! Please read BUGS section in ifconfig(8).
        infiniband 00:00:09:07:FE:80:00: ..... txqueuelen 256 (InfiniBand)

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
        inet 127.0.0.1 netmask 255.0.0.0
        inet6 ::1 prefixlen 128 scopeid 0x10<host>

[s271550@ctipt-tnode007 etc]$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
2: em1: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_UP> mtu 1500 qdisc mq master bond0 state UP mode DEFAULT group default qlen 1000
3: em2: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_UP> mtu 1500 qdisc mq master bond0 state UP mode DEFAULT group default qlen 1000
4: ib0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 2044 qdisc mq state UP mode DEFAULT group default qlen 256
    link/infiniband ....
5: bond0: <BROADCAST,MULTICAST,MASTER,UP,LOWER_UP> mtu 1500 qdisc noqueue master br0 state UP mode DEFAULT group default qlen 1000
6: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default qlen 1000
```

With openMPI implementation and UCX, we can directly select the devices that lead to a specific protocol. The devices tested are ib0, br0 and mlx5_0:1. Ib0 is the IPoIB protocol, br0 leads a TCP communication and mlx5_0:1 is a pure native InfiniBand device. The theoretical maximum performances are 12.5 GB/s or 12800 MB/s for InfiniBand and 3.125 GB/s or 3200 MB/s for the Ethernet network.

The experimental asymptotic bandwidth measured are:



UCX with InfiniBand and Intel shows the best performances among all the configurations both in terms of latency and bandwidth. UCX overperforms a bit Intel latency. No big cache effects are visible, thanks to RDMA that exclude all caches, CPU and kernel action. This explain also the highest bandwidth with respect to core and socket mapping. Real InfiniBand performance is about 95% of theoretical bandwidth, this is a very nice result assuming the encoding 64b/66b.

UCX with br0 and OB1 with tcp show comparable latency, but OB1 gains a better bandwidth. The real maximum performance is about 85% of theoretical bandwidth, this is a good result taking into account the heavier tcp protocol with respect to InfiniBand (ACK, handshake, encoding...). Moreover, transport overhead and some inefficiency introduced by the cache and CPU (no RDMA available) are present.

IPoIB has a good latency (no Ethernet switch involved) but the bandwidth is not so high as expected. 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 CPU frequency and node configuration. Cache sizes are the same of thin nodes, and this cache effects are similar. The summary of fitting results on THIN and GPU node is presented in the next page.

Table 2: Fitting results

THIN node	Latency uSec	Bandwidth MB/s	GPU node	Latency uSec	Bandwidth MB/s
Core mapping			Core mapping		
UCX IB	0.2440430	5987.179	UCX IB	0.2745335	5652.041
Intel IB	0.2526662	5385.856	Intel IB	0.3247967	5117.350
OB1 vader	0.3658680	3866.456	OB1 vader	0.3850294	3750.822
OB1 tcp	5.0266853	5207.742	OB1 tcp	5.0516070	5150.110
Socket mapping			Socket mapping		
Intel IB	0.5198369	4204.387	Intel IB	0.5685345	4237.763
UCX IB	0.6033689	5123.148	UCX IB	0.6513036	4615.458
OB1 vader	0.8531296	3258.498	OB1 vader	0.8521363	3070.333
OB1 tcp	7.9899043	3325.931	OB1 tcp	8.9058952	3127.998
UCX IB	1.1326157	12182.859			
Node mapping					
Intel ib	1.3189110	12223.147			
UCX ib0	9.6072277	2764.778			
UCX br0	15.8550522	2433.241			
OB1 tcp	16.4297090	2729.666			

Section 3 : Jacobi solver

Performance model

In order to predict Jacobi model performance the following model has been used:

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

L represent the subdomain size (assuming it has cube shape), in our case it is constant. N represent the number of processes. The problem's size is $L^3 N$. This amount of work increases linearly as a function of N , producing a weak scalability. The performance is evaluated using $MLUP/s$, mega lattice update per sec.

T_s is the time swept, is constant and it can be estimated using the serial output. For Thin node $T_s = 3.05$ and for Gpu node $T_s = 4.3$.

T_c represents the communication time in seconds. This quantity can be modeled by estimating the latency λ , bandwidth B and the sizes of messages C .

$$T_c = \frac{C}{B} + 4k\lambda [seconds]$$

Regarding C , we assume grid point as double:

$$C = L^2 k \cdot 2 \cdot 2 \cdot 8 [byte]$$

k represents the number of directions in which the halo exchange occurs, it must be multiplied by 2 taking into account the bidirectional halo exchange and again it must be multiplied by two taking into account the positive and the negative directions.

Domain decomposition

Several domain decompositions are possible, some of them can be discarded a priori assuming poor performance due to buffering. For example between (12, 1, 1) (1, 12, 1) (1, 1, 12), the first one is the best and more

memory friendly decomposition. Moreover, it has the lowest buffering needs. The latter exapme can be proved experimentally: the first configuration behave better than the other in terms of MLUP/s.

Experimentals results

Thin node, core mapping.

$\lambda = 0.24[usec]$ and $B = 5987[MB/s]$.

N° procs	Nx	Ny	Nz	k	C MB	Tc s	Theo perf MLUP/s	Real perf MLUP/s	Elapsed time s	NP(1)/P(N)
1	1	1	1	0	0.00000	0.0000000	112.4590	112.2617	54.26	1.000000
4	4	1	1	1	14.95361	0.0024986	449.4678	448.0708	54.44	1.002178
	2	2	1	2	29.90723	0.0049973	449.1002	448.9819	54.37	1.000145
8	8	1	1	1	14.95361	0.0024986	898.9357	894.6112	54.59	1.003893
	4	2	1	2	29.90723	0.0049973	898.2005	896.9956	54.61	1.001224
	2	2	2	3	44.86084	0.0074959	897.4664	894.3671	54.61	1.004167
12	12	1	1	1	14.95361	0.0024986	1348.4035	1329.5301	55.27	1.013246
	4	3	1	2	29.90723	0.0049973	1347.3007	1315.6552	55.70	1.023931
	3	2	2	3	44.86084	0.0074959	1346.1997	1324.6981	55.42	1.016942
	6	2	1	2	29.90723	0.0049973	1347.3007	1323.8481	55.46	1.017595

Thin node, socket mapping.

$\lambda = 0.6[usec]$ and $B = 5123[MB/s]$.

N° procs	Nx	Ny	Nz	k	C MB	Tc s	Theo perf MLUP/s	Real perf MLUP/s	Elapsed time s	NP(1)/P(N)
1	1	1	1	0	0.00000	0.0000000	112.4590	112.3160	54.24	1.000000
4	4	1	1	1	14.95361	0.0029213	449.4056	448.5776	54.35	1.001530
	2	2	1	2	29.90723	0.0058426	448.9760	448.3072	54.36	1.002134
8	8	1	1	1	14.95361	0.0029213	898.8112	895.9560	54.44	1.002871
	4	2	1	2	29.90723	0.0058426	897.9520	895.1857	54.49	1.003734
	2	2	2	3	44.86084	0.0087640	897.0944	895.8977	54.47	1.002936
12	12	1	1	1	14.95361	0.0029213	1348.2169	1332.8755	54.83	1.011191
	4	3	1	2	29.90723	0.0058426	1346.9280	1336.4717	54.84	1.008471
	3	2	2	3	44.86084	0.0087640	1345.6416	1342.5132	54.63	1.003932
	6	2	1	2	29.90723	0.0058426	1346.9280	1342.4993	54.62	1.003943

Thin node, node mapping.

$\lambda = 1.35[usec]$ and $B = 12182[MB/s]$.

N° procs	Nx	Ny	Nz	k	C MB	Tc s	Theo perf MLUP/s	Real perf MLUP/s	Elapsed time s	NP(1)/P(N)
1	1	1	1	0	0.00000	0.0000000	112.459	112.1941	54.26	1.000000
12	12	1	1	1	14.95361	0.0012329	1348.963	1334.3096	54.80	1.009008
	4	3	1	2	29.90723	0.0024658	1348.418	1334.8244	54.75	1.008619
	3	2	2	3	44.86084	0.0036988	1347.874	1342.3393	54.57	1.002973
	6	2	1	2	29.90723	0.0024658	1348.418	1341.9843	54.58	1.003238
24	24	1	1	1	14.95361	0.0012329	2697.926	2680.5702	54.80	1.004510
	12	2	1	2	29.90723	0.0024658	2696.836	2683.9718	54.75	1.003237
	8	3	1	2	29.90723	0.0024658	2696.836	2678.9662	54.78	1.005111
	6	4	1	2	29.90723	0.0024658	2696.836	2676.7929	54.79	1.005927
	6	2	2	3	44.86084	0.0036988	2695.747	2681.9062	54.82	1.004009
	4	3	2	3	44.86084	0.0036988	2695.747	2674.0858	54.85	1.006946
48	48	1	1	1	14.95361	0.0012329	5395.852	5260.9379	56.39	1.023642
	24	2	1	2	29.90723	0.0024658	5393.672	5247.7545	56.51	1.026214
	12	4	1	2	29.90723	0.0024658	5393.672	5249.9587	56.58	1.025783
	12	2	2	3	44.86084	0.0036988	5391.494	5215.2751	56.73	1.032605
	8	6	1	2	29.90723	0.0024658	5393.672	5222.1200	56.67	1.031251
	6	4	2	3	44.86084	0.0036988	5391.494	5244.1732	56.52	1.026915

Even if this model seems accurate, some comments are required: mapping by socket leads to a slightly better performance with respect to core mapping and this is not what we expected considering the theoretical model. If, on one hand socket map lead to poor communication performance, on other hand it exploits better the memory allocation. When one socket is filled with 12 processes the entire grid is stored on socket competence LRDIMM, but spreading processes across 2 sockets lead to a better exploit of bandwidth using different memory controller and then all memory channels available. At each iteration, this solver elaborates huge amount of memory: given $L = 1200$ and 12 workers at least 300 GB of ram are elaborated (half LRDIMMs are saturated, then all single socket channels are used). Also with map-by node option we obtain better results when several processes are involved. The following results report different domain decompositions and binding with $L = 1200$, $N = 12$ on thin nodes. Is reported the maximum value among all runs.

Mapping	Nx	Ny	Nz	MLUP/s
core	12	1	1	1335
	1	1	12	1332
socket	12	1	1	1347
	1	1	12	1345

Performance model doesn't take into account the buffering requirement, given a domain decomposition, needed to exchange the halo point. It is not possible to infer theoretically using same domain decomposition but on different direction. Experimentally differences on performance are negligible but still present.

Gpu node, core mapping.

N° procs	Nx	Ny	Nz	k	C MB	Tc s	Theo perf MLUP/s	Real perf MLUP/s	Elapsed time s	NP(1)/P(N)
1	1	1	1	0	0.00000	0.0000000	79.76744	77.4580	76.52	1.000000
12	12	1	1	1	14.95361	0.0026468	956.62047	850.5267	88.24	1.093200
	4	3	1	2	29.90723	0.0052936	956.03236	849.5713	88.14	1.094429
	3	2	2	3	44.86084	0.0079404	955.44497	849.6338	88.03	1.094348
	6	2	1	2	29.90723	0.0052936	956.03236	850.3519	88.10	1.093424
24	24	1	1	1	14.95361	0.0026468	1913.24094	1690.6353	89.12	1.099936
	12	2	1	2	29.90723	0.0052936	1912.06472	1700.2421	88.86	1.093721
	8	3	1	2	29.90723	0.0052936	1912.06472	1699.0319	88.91	1.094500
	6	4	1	2	29.90723	0.0052936	1912.06472	1700.3159	88.96	1.093674
	6	2	2	3	44.86084	0.0079404	1910.88995	1698.9900	88.89	1.094527
	4	3	2	3	44.86084	0.0079404	1910.88995	1699.5484	88.89	1.094168
48	48	1	1	1	14.95361	0.0026468	3826.48188	2538.1989	114.90	1.465284
	24	2	1	2	29.90723	0.0052936	3824.12944	2534.7265	115.10	1.467291
	12	4	1	2	29.90723	0.0052936	3824.12944	2523.0021	115.40	1.474110
	12	2	2	3	44.86084	0.0079404	3821.77989	2550.3071	114.40	1.458327
	8	6	1	2	29.90723	0.0052936	3824.12944	2549.9056	114.50	1.458557
	6	4	2	3	44.86084	0.0079404	3821.77989	2527.2825	115.30	1.471613

Gpu node, socket mapping.

N° procs	Nx	Ny	Nz	k	C MB	Tc s	Theo perf MLUP/s	Real perf MLUP/s	Elapsed time s	NP(1)/P(N)
1	1	1	1	0	0.00000	0.0000000	79.76744	77.4580	76.52	1.000000
12	12	1	1	1	14.95361	0.0032428	956.48797	900.1881	83.73	1.032890
	4	3	1	2	29.90723	0.0064856	955.76773	900.3287	83.15	1.032729
	3	2	2	3	44.86084	0.0097285	955.04857	893.0782	83.52	1.041113
	6	2	1	2	29.90723	0.0064856	955.76773	899.9480	83.18	1.033166
24	24	1	1	1	14.95361	0.0032428	1912.97595	1703.9676	88.97	1.091330
	12	2	1	2	29.90723	0.0064856	1911.53546	1701.9177	89.11	1.092644
	8	3	1	2	29.90723	0.0064856	1911.53546	1700.2848	88.88	1.093694
	6	4	1	2	29.90723	0.0064856	1911.53546	1698.8672	88.94	1.094606
	6	2	2	3	44.86084	0.0097285	1910.09714	1699.6994	88.86	1.094070
	4	3	2	3	44.86084	0.0097285	1910.09714	1699.0902	88.90	1.094463
48	48	1	1	1	14.95361	0.0032428	3825.95189	2528.7299	115.50	1.470771
	24	2	1	2	29.90723	0.0064856	3823.07092	2542.1043	114.50	1.463033
	12	4	1	2	29.90723	0.0064856	3823.07092	2531.6678	115.00	1.469064
	12	2	2	3	44.86084	0.0097285	3820.19428	2546.4954	114.30	1.460510
	8	6	1	2	29.90723	0.0064856	3823.07092	2531.7794	114.60	1.468999
	6	4	2	3	44.86084	0.0097285	3820.19428	2542.6490	114.50	1.462719

On gpu node differences on socket and core mapping are even bigger and enhanced, again we can guess the memory is the main cause. Even if has higher frequency LDIMMs modules, Gpu node setup doesn't exploit all possible memory channel. When worker number grow over physical core number, the real performance goes dramatically down respect the model due hyperthreading. Also the execution time show poor performance

with 48 workers. All materials, scripts, bash code, tools and machinery developed to achieve assignment goals are available on thi github repository https://github.com/NiccoloTosato/HPC_assignment1