Assignement 1 Report

Exercise 1

Ring

I have implemented an MPI program in C with $2P$ messages passing among P processes on a ring topology with periodic boundaries.

The program implements a stream of messages of $4B$ each, both clockwise and anticlockwise: each process in fact send/receive messages of **type int** (the *rank*) with a tag proportional to the rank ($tag = rank * 10$).

When running on P process, the program prints out in folder *./out* a file *'npP.txt'* with the following output ordered by rank (here $P = 5$):

```
I am process 0 and I have received 5 messages. My final messages have tag 0 and value msg-left -10, msg-right 10 I am process 1 and I have received 5 messages. My final messages have tag 10 and value msg-left -10, msg-right 10     I am process 2 and I have received 5 messages. My final messages have tag 20 and value msg-left -10, msg-right 10  I am process 3 and I have received 5 messages. My final messages have tag 30 and value msg-left -10, msg-right 10  I am process 4 and I have received 5 messages. My final messages have tag 40 and value msg-left -10, msg-right 10
```

I have used the following routines for message passing among processes:

- `MPI_Isend`: **non-blocking send** routine
- `MPI_Irecv`: **non-blocking receive** routine
- `MPI_Wait`: waits for a non-blocking operation to complete

Using latest version of OpenMPI available on ORFEO (openmpi-4.1.1+gnu-9.3.0), I have run the program up to $P = 24$ processes on a **thin node with InfiniBand network and native protocol**.

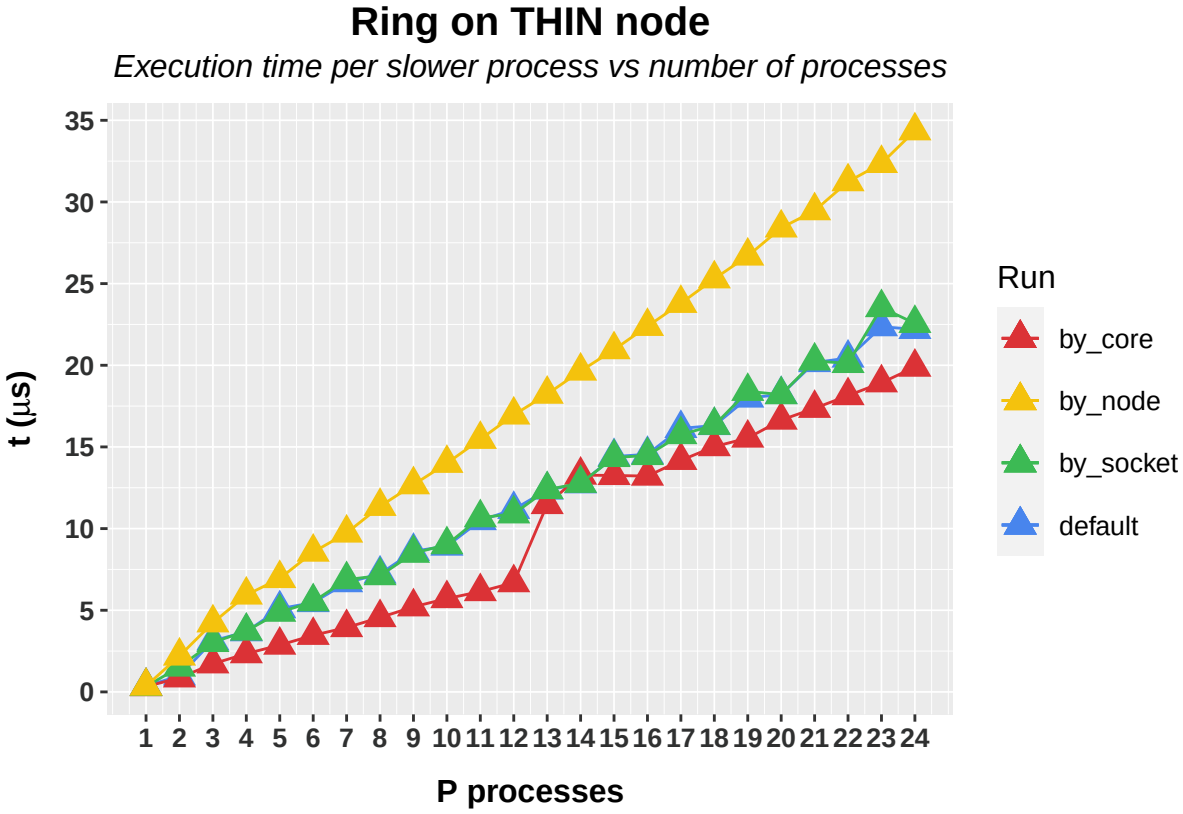
I have taken notes of the runtime (from the first send/receive to the last) when varying the number of processes P by means of the routine `MPI_Wtime()` which returns an elapsed time on the calling process in seconds.

At each run with P processes, for each process I have taken the mean of the **runtime in microseconds** out of **10000** iterations and print it out (along with some statistics) by ranking order on the file *'npP.csv'* in folder *./out*, as follows:

Table 1: Results with P = 5

rk	P	t_mean	t_xmsg	var	s2	N	dev	s
0	5	7.073931	1.414786	1425.467	1425.496	10000	37.75536	37.75574
1	5	7.072714	1.414543	1423.095	1423.123	10000	37.72393	37.72430
2	5	7.073540	1.414708	1428.383	1428.412	10000	37.79395	37.79433
3	5	7.072797	1.414559	1419.479	1419.507	10000	37.67598	37.67635
4	5	7.074060	1.414812	1430.666	1430.694	10000	37.82414	37.82452

Then, by taking for each run with different P only the **maximum value of t_mean** between all processes as measure of the performance, I was able to produce the following plot, running the program with different mappings:



Because of the routines that I have used in the program, **I expect my data to be in compliance with a P double PingPing model**.

By means of **IMB-MPI1 PingPing benchmark** it's possible to measure startup Δt and throughput $\frac{X}{\Delta t}$ of single messages of size X that are obstructed by oncoming messages. To achieve this, two processes communicate with each other using *MPI_Isend/MPI_Recv/MPI_Wait* calls, just like in my program, as follows:

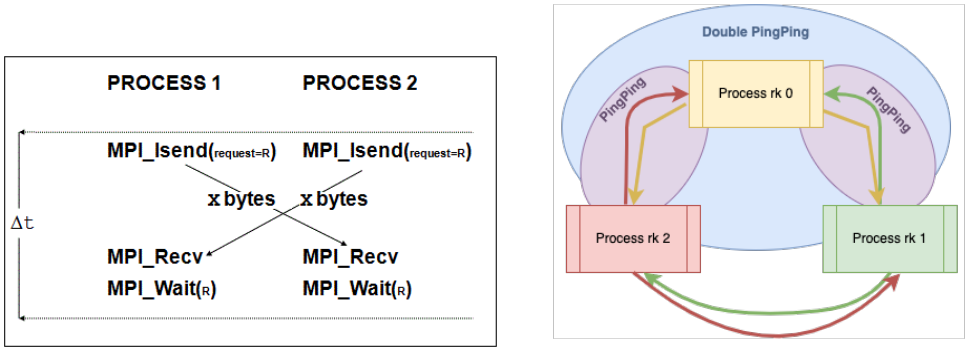


Figure 1: Fig.1 PingPing Pattern from Intel® MPI Benchmarks User Guide and Ring with P=3.

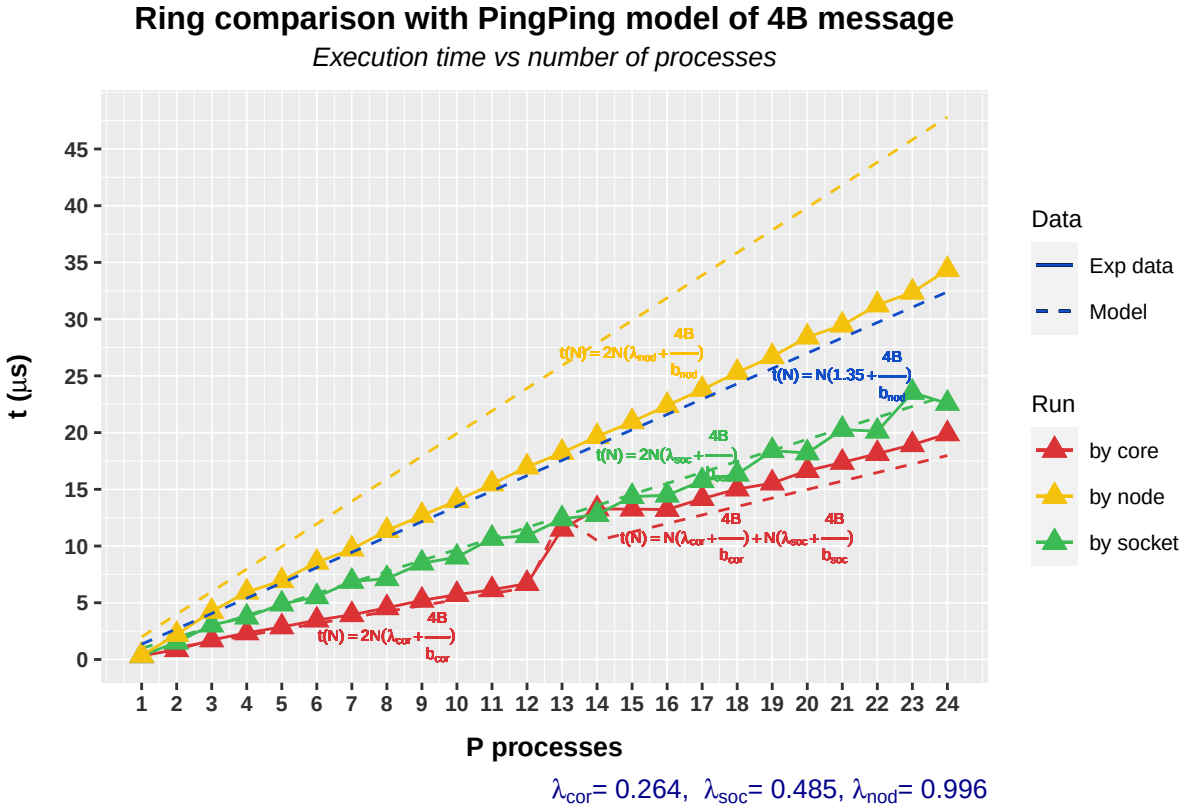
With IMB-MPI1 PingPing benchmark I was able to estimate the latency λ_{net} and bandwidth b_{net} on Infiniband network with different processes mappings: across nodes, sockets and cores. Since my t_{mean} is a measure of time in μs of a pair of opposite messages passing through all P process till returning back to the original one, $t_{xmsg} = \frac{t_{mean}}{P}$ is the variable accounting for half of the Δt time measured from the PingPing benchmark results.

Thus, by taking the inverse of the previous relation $t_{mean} = t_{xmsg}P$, our theoretical model will be $t_{theo} =$

$2\Delta t_{ping} P$, where $\Delta t_{ping} = \lambda_{ping} + \frac{4B}{b_{ping}}$ with λ_{ping} and b_{ping} estimated by least square method on the PingPing benchmark results.

Hence, the following communication model has been used for plotting my data with the PingPing model:

- $t(P) = 2P(\lambda_{core} + \frac{4B}{b_{core}})$ for **core mapping when** $P \leq 12$.
- $t(P) = 2P(\lambda_{socket} + \frac{4B}{b_{socket}})$ for **core mapping when** $P = 13$, assuming that the process chosen as representative of my running is the one with the maximum value of t_{mean}, hence the one wich is in the other socket farther from the others 12 cores.
- $t(P) = P(\lambda_{core} + \frac{4B}{b_{core}}) + P(\lambda_{socket} + \frac{4B}{b_{socket}})$ for **core mapping when** $P > 13$, assuming again that the slower process is the one communicating with the previous core which is in the other socket, and the next core which is inside its socket.
- $t(P) = 2P(\lambda_{socket} + \frac{4B}{b_{socket}})$ for **socket mapping**.
- $t(P) = 2P(\lambda_{core} + \frac{4B}{b_{core}})$ for **node mapping**.



Oss. The extimated bandwiths of the network for the different mappings are:

- $b_{core} = 6372.991$ MB/s
- $b_{socket} = 5530.801$ MB/s
- $b_{node} = 11945.604$ MB/s

As expected, the measured bandwith for the 100 Gbit InfiniBand network (process message passing among nodes) reaches up to approximately 12000 MB/s ($= 12$ GB/s $= 12 * 8$ Gbit/s $= 96$ Gbit/s) which is 96% of the theoretical peak performance. Anyway this term in the model can easily be omitted since it's in the order of 10^{-10}.

The estimated latency between two nodes λ_{node} with the PingPing benchmark is somewhat less than the one declared by Mellanox switch constructor (1.35 μs) used in InfiniBand network: this fact seems to hold when few processes are communicating.

I was indeed expecting that the theoretical performance (evaluated by PingPing benchmark among just

2 process) would have been better than the real scenario when all cores/sockets in nodes are being used simultaneously, and in fact as can be seen from the previous plot, **my model seems to work good for core and socket mapping**, but quite bad for node mapping.

Data from node mapping seem to follow another model that is half the expected one: $t(P) = P(\lambda_{mlx5} + \frac{4B}{b_{node}})$, where $\lambda_{mlx5} = 1.35$ is the latency of the Mellanox switch indeed. This may suggest that the *switch is able to send simultaneously 2 messages of* $4B$ *in each direction of the network, halving the time for message passing* between processes placed in different nodes.

Exercise 2

Measure MPI point to point performance

The **Intel MPI IMB-MPI1 benchmark PingPong** has been used to estimate latency λ_{net} and bandwidth b_{net} of *all available combinations of topologies and networks on ORFEO computational nodes, using both IntelMPI and openmpi latest versions* libraries availables.

Let's start by looking at ORFEO computational nodes and resources:

```
[valinsogna@login 2021Assignement01]$ pbsnodes -ajS
                                                    mem       ncpus   nmics   ngpus
vnode           state           njobs   run   susp      f/t        f/t     f/t     f/t   jobs
--------------- --------------- ------ ----- ------ ---------- ------- ------- ------- -------
ct1pf-fnode001  job-busy             1     1      0    560gb/1tb    0/36     0/0     0/0 56793
ct1pf-fnode002  free                 1     1      0      1tb/1tb   12/36     0/0     0/0 55267
ct1pt-tnode001  job-busy             1     1      0   54gb/754gb    0/24     0/0     0/0 56845
ct1pt-tnode002  job-busy             1     1      0   54gb/754gb    0/24     0/0     0/0 56846
ct1pt-tnode004  job-busy             1     1      0   54gb/754gb    0/24     0/0     0/0 56847
ct1pt-tnode005  job-busy             1     1      0   54gb/754gb    0/24     0/0     0/0 56794
ct1pt-tnode006  free                 0     0      0  754gb/754gb   24/24     0/0     0/0 --
ct1pt-tnode007  job-busy             1     1      0   54gb/754gb    0/24     0/0     0/0 56795
ct1pt-tnode008  job-busy             1     1      0   54gb/754gb    0/24     0/0     0/0 56848
ct1pt-tnode009  job-busy             1     1      0  754gb/754gb    0/24     0/0     0/0 57073
ct1pt-tnode010  job-busy             1     1      0  754gb/754gb    0/24     0/0     0/0 57073
ct1pt-tnode003  offline              0     0      0  754gb/754gb   24/24     0/0     0/0 --
ct1pg-gnode001  free                 1     1      0  252gb/252gb   44/48     0/0     0/0 57057
ct1pg-gnode002  free                 1     1      0  252gb/252gb   24/48     0/0     0/0 56354
ct1pg-gnode003  free                 1     1      0  252gb/252gb   24/48     0/0     0/0 56862
ct1pg-gnode004  free                 0     0      0  252gb/252gb   48/48     0/0     0/0 --
```

On ORFEO there are:

- 2 fat nodes: with 2 CPUs (2 NUMA domains) of 18 cores each, with more than 1 TB of RAM.
- 4 gpu nodes: with hyper-threading enabled, with 2 CPUs (2 NUMA domains) of 12 physical cores each with more than 252 GB of RAM.
- 10 thin nodes: with 2 CPUs (2 NUMA domains) of 12 cores each, with more than 754 GB of RAM.
- login node: with 2 CPUs (2 NUMA domains) of 10 cores each.

The MPI point to point performance has been measured only on thin and gpu nodes.

Topology of tested nodes

Now we can look at the node topologies for the thin and gpu ones. This can be either done by typing on the selected node `module load likwid` and then `likwid-topology` or using `module load hwloc` and `lstopo`. The figure below represents the lstopo output on a thin node, which is more or less the same for the gpu node.

The main differences between a gpu and a thin node are the hyper-threading enabled on the previous, the different RAM size and the different CPUs models:

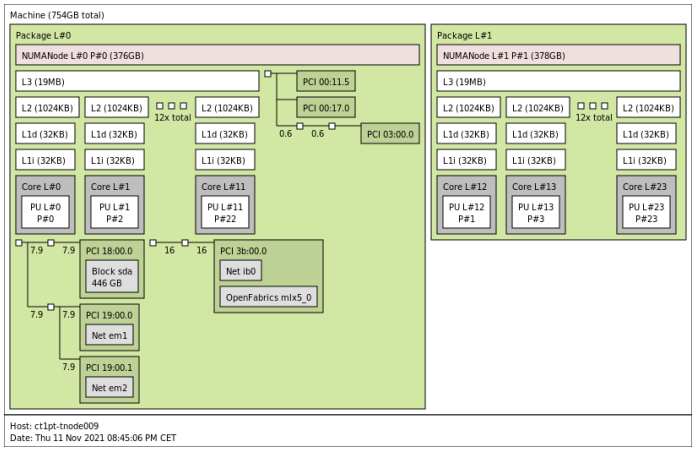


Figure 2: Fig.2 Topology on thin node

- Intel(R) Xeon(R) Gold 6226 CPU @ **2.70GHz** for gpu node
- Intel(R) Xeon(R) Gold 6126 CPU @ **2.60GHz** for thin node

As can be seen by typing `lscpu`:

```
[valinsogna@ct1pt-tnode008 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    1
Core(s) per socket:    12
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz
Stepping:              4
CPU MHz:               3299.999
CPU max MHz:           3700.0000
CPU min MHz:           1000.0000
BogoMIPS:              5200.00
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              19712K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23
```

```
[valinsogna@ct1pg-gnode001 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                48
On-line CPU(s) list:   0-47
Thread(s) per core:    2
```

```
Core(s) per socket:    12
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Gold 6226 CPU @ 2.70GHz
Stepping:              7
CPU MHz:               3499.999
CPU max MHz:           3700.0000
CPU min MHz:           1200.0000
BogoMIPS:              5400.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              19712K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47
```

Networks and protocols used

- High Speed Network **100 Gbit InfiniBand**: with $peak\ perf_{theo}$ of 100 Gbit/s = 12.5 GB/s = **12500 MB/s**; eq. (1)
- In band management network **25 Gbit Ethernet**: with $peak\ perf_{theo}$ of 25 Gbit/s = 3.125 GB/s = **3125 MB/s**; eq. (2)

The main differences among these two are the usage of Sockets Interface and TCP/IP protocol for the Ethernet network, and the usage of much more rapid OpenFabrics Verbs (no Kernel stack) with the native IB protocol for InfiniBand network. Moreover, IB protocol has RDMA (Remote Direct Memory Access) that makes InfiniBand faster: it is an operation which access the memory directly without involving the CPU. We thus, expect InfiniBand network with native IB protocol to be much faster (low λ_{net}, high b_{net}) then the Ethernet network. Moreover, I am going to test also InfiniBand network performance when applying plain IP protocol: **IPoIB (IP-over-InfiniBand) is the protocol that defines how to send IP packets over IB**, passing through the Kernel space.

As can be seen from Fig.2, the network cards are placed in each node only inside one of the two NUMA domains, and there are several PCI (Peripherical Component Interconnect) devices that can be seen by typing `ifconfig`:

```
[valinsogna@ct1pt-tnode007 ~]$ ifconfig
bond0: flags=5187<UP,BROADCAST,RUNNING,MASTER,MULTICAST>  mtu 1500
        inet6 fe80::3680:dff:fe4e:5568  prefixlen 64  scopeid 0x20<link>
        ether 34:80:0d:4e:55:68  txqueuelen 1000  (Ethernet)

br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.128.2.127  netmask 255.255.255.0  broadcast 10.128.2.255
        inet6 fe80::3680:dff:fe4e:5568  prefixlen 64  scopeid 0x20<link>
        ether 34:80:0d:4e:55:68  txqueuelen 1000  (Ethernet)

em1, em2 two physical cards that we have on the node
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
        inet 10.128.6.127  netmask 255.255.255.0  broadcast 10.128.6.255
        inet6 fe80::ba59:9f03:d4:27d6  prefixlen 64  scopeid 0x20<link>
Infiniband hardware address can be incorrect! Please read BUGS section in ifconfig(8).
        infiniband 00:00:09:07:FE:80:00:00:00:00:00:00:00:00:00:00:00:00:00:00  txqueuelen 256  (Infinil

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>
        loop  txqueuelen 1000  (Local Loopback)
```

As from above, **em1, em2 are two physical distinguished Ethernet cards that refers to one interface br0**, whilst **ib0 is InfiniBand**. More details are shown below with `lstopo`:

```
[valinsogna@ct1pt-tnode007 ~]$ lstopo
Machine (754GB total)
  Package L#0
    NUMANode L#0 (P#0 376GB)
    L3 L#0 (19MB)
      L2 L#0 (1024KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0 + PU L#0 (P#0)
      L2 L#1 (1024KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1 + PU L#1 (P#2)
      ...
    HostBridge
      PCI 00:11.5 (SATA)
      PCI 00:17.0 (SATA)
      PCIBridge
        PCIBridge
          PCI 03:00.0 (VGA)
    HostBridge
      PCIBridge
        PCI 18:00.0 (RAID)
          Block(Disk) "sda"
      PCIBridge
        PCI 19:00.0 (Ethernet)
          Net "em1"
        PCI 19:00.1 (Ethernet)
          Net "em2"
    HostBridge
      PCIBridge
        PCI 3b:00.0 (InfiniBand)
          Net "ib0"
          OpenFabrics "mlx5_0"
```

With openMPI implementation and UCX, it's possible to directly select the devices (using `UCX_NET_DEVICES` specification in the run command), that lead to a specific protocol as consequence. **The devices tested with openMPI across nodes are:**

- **ib0: IPoIB protocol.**
- **br0: TCP communication, Ethernet.**
- **mlx5_0:1: native IB protocol.**

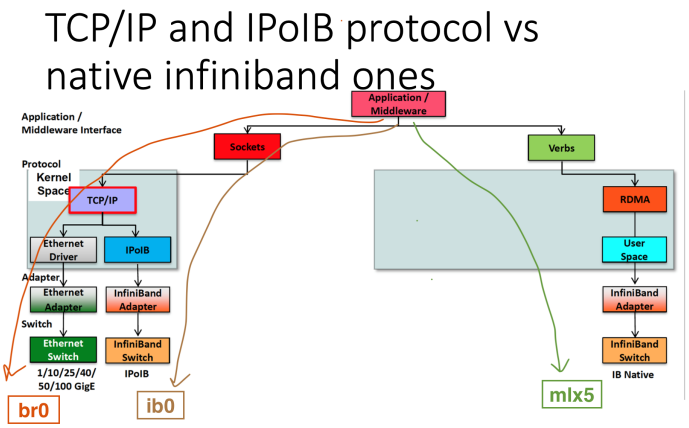


Figure 3: TCP/IP, IPoIB and native IB protocols

IMB-MPI1 Benchmark PingPong

Intel MPI benchmark IMB-MPI1 PingPong measure message passing between two processes and works as follow:

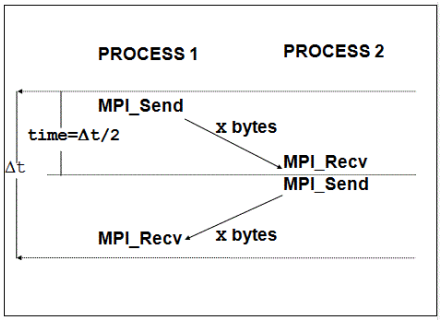


Figure 4: PingPong Pattern from Intel® MPI Benchmarks User Guide

It reports the time $\Delta t/2$ (in μs), throughput $\frac{2X}{\Delta t}$ (in MB/s), number of repetitions and the message size X (in B) in the standard output:

```
#---------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#---------------------------------------------------
       #bytes #repetitions      t[usec]   Mbytes/sec
            0,         1000,        0.20,         0.00
            1,         1000,        0.21,         4.90
            2,         1000,        0.20,        10.07
            4,         1000,        0.22,        20.26
            8,         1000,        0.24,        40.59
```

Measurements

The bandwidth and the latency estimation for gpu and thin nodes is done across cores, sockets and nodes, with different protocols, PCI devices and libraries.

The nodes involved are:

- ct1pt-tnode007, thin node for results across cores, sockets and nodes.
- ct1pt-tnode008, thin node for results across nodes.
- ct1pg-gnode001, gpu node for results across cores, sockets and nodes.
- ct1pg-gnode003, gpu node for results across nodes.

Each specific run is repeated 10 times with `-msglog 28` and each time **for the thin ones the entire node was reserved** in order to reduce noise in measurements, **instead for the gpu nodes it wasn't possible due to some queue traffic**. Thus, from the measurements, it was taken the mean time $\Delta t/2$ and the mean throughput $2X/\Delta t$ between the 10 results for each message size X, along with some statistics (ex. maximum error on time and on throughput $e = \frac{t_{max} - t_{min}}{2}$).

The PingPong benchmark is compiled on either the thin or gpu node using two different libraries:

- **OpenMPI**: version **openmpi-4.0.3** for the running across nodes with `UCX_NET_DEVICES` specification and latest **openmpi-4.1.1** for the others.
- **Intel MPI**: version `intel` available on ORFEO.

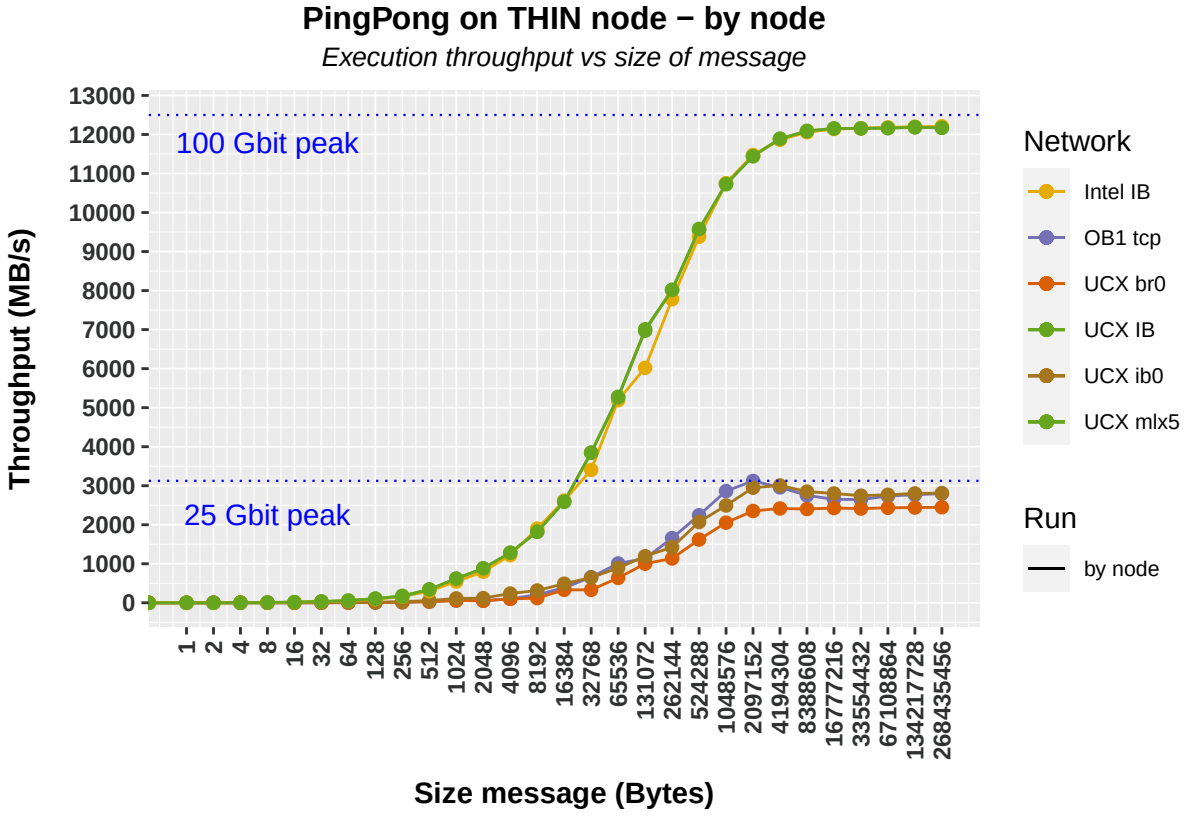
With **OpenMPI** implementation it is possible to specify the MPI frameworks and plugins (like pml and btl) that can be used while running. Across nodes, cores and sockets the following as been tested:

- *PML* : **ob1** and **ucx**.
- *BTL* : **tcp**, **self** and **vader**.

The following graphs show the results both for thin and gpu nodes from plotting the benchmark throupout $2X/\Delta t$ against the message size X with different mappings of the processes: across two different nodes for the first plot and across two sockets or in the same socket for the latter.

Along with the results of 'across nodes' mapping, there has been drawn the lines for the 100 Gbit InfiniBand and 25 Gbit Ethernet, in order to compare the theoretical peak performances with the tested ones.

Moreover, in the plot 'by socket/core' mapping, cache size lines have been drawn too in order to discuss about a general behavior.

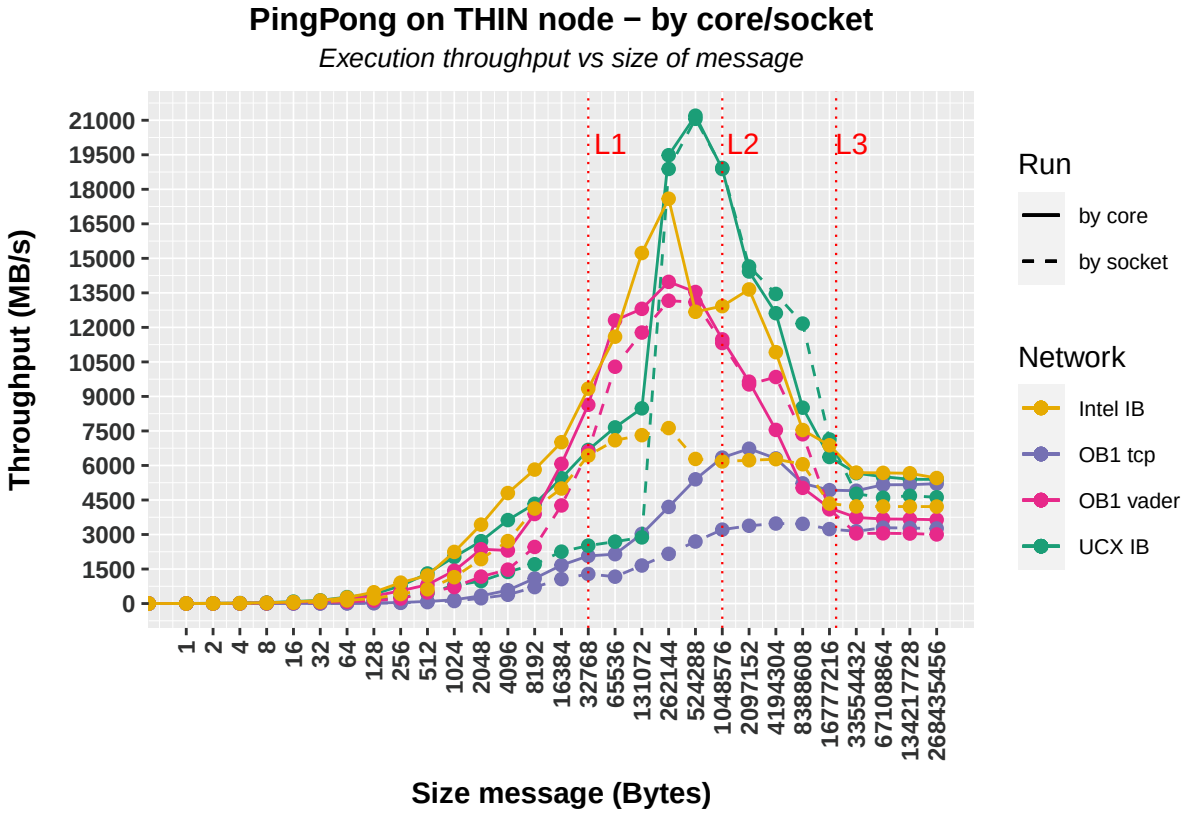


From this first plot it is evident that **with Intel MPI, UCX OpenMPI InfiniBand** (the default command) **and UCX native IB with mlx5_0:1** device (which is the same as the previous), **the asymptotic bandwith is very high compared to the others protocols/devices and similar to the one expected from the theoretical peak**: more or less my data **perform like** 97% **of the theoretical limit** (see eq (2)).

UCX with br0 and OB1 with tcp show a real maximum performance of **about 86%-89% respectively of theoretical bandwidth**: it is a good result if we take into account that tcp protocol is heavier with respect to the native IB one (encoding, no RDMA available). Their latencies are comparable (see table in 'Results' section), but OB1 has a highier bandwidth.

IPoIB (aka UCX ib0) happen to have a good latency (see table in 'Results' section) but the bandwidth is not so high and in fact it is comparable to the one of the 25 Gbit Ethernet network.

Now let's look at the performance with the across core and socket configuration.



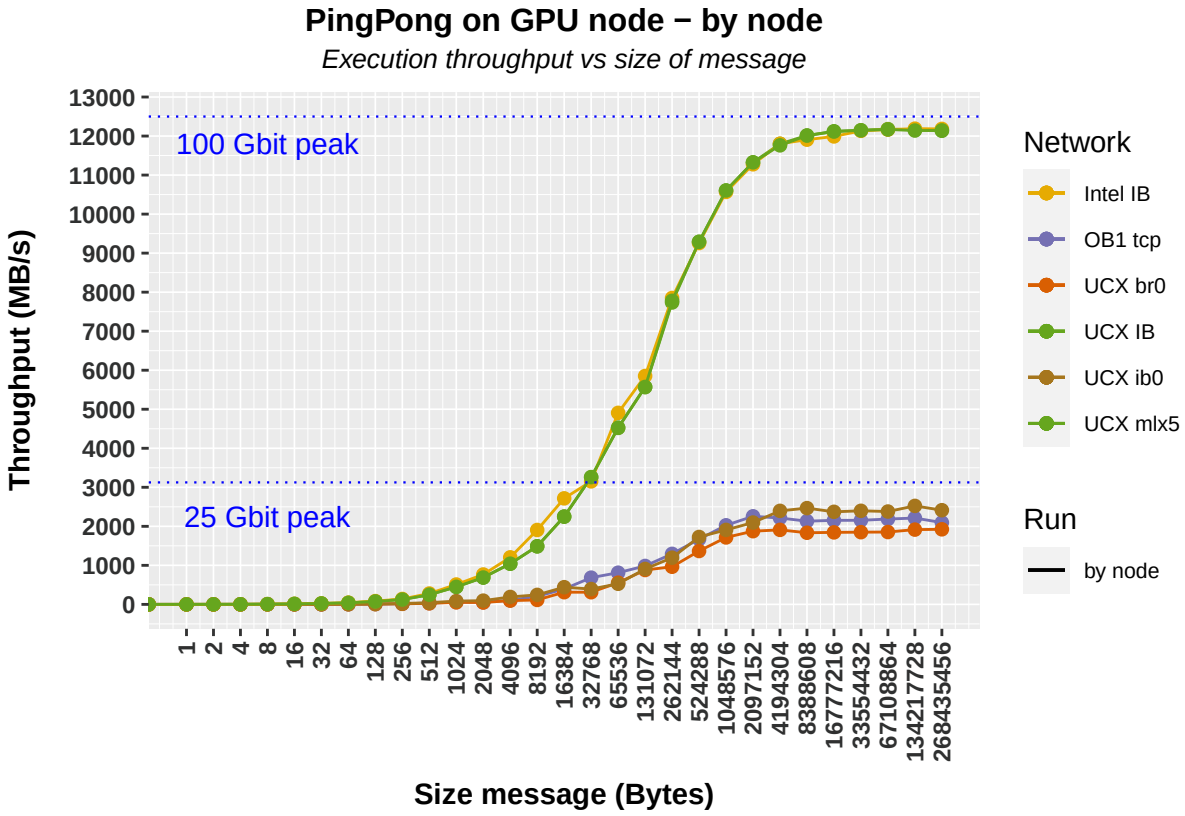
From first glance it is clear that these performance are smaller in bandwith, but better in latency (see table in 'Report section') when compared to across nodes configuration. Moreover, **mapping the processes in the same socket show often a better performance than mapping them in separate sockets** .

The drop in bandwidth before the asymptotic plateau of throughput is common in both the 2 configurations (by sockets, by nodes) and it shows very different performances among different implementations. **This pattern happens between 32KB and 16 MB included: after that size, measurements become stable**.

By plotting the cache lines it is clear that this behavior **must be related to the cache**: in fact, when the size of the PingPong message becomes too large to fit in **L1**, there's an L1 miss and data are retrieved from cache **L2**. This happens too for L2 as the size grows so there is a visible drop in the throughput. L1 effects is not clearly visible from the bandwidth (probably due the latency), instead for L2 misses after 1 MB, all implementations start loosing bandwidth. In the plateau area, instead, the message size is larger than all caches and a stable bandwith is reached.

Now let's compare protocols. Firstly, **UCX IB shows poor performance before 131 MB** respect to the other protocols and after that size there is an huge increase. **Also Intel InfiniBand, this time shows poor performance respect UCX implementation**, and it might be related to the fact that it undergoes large cache misses. **OB1** implementation seems to be better by usage of *vader* btl rather than *tcp*, as expected since *vader* is suitable for shared memory transfer.

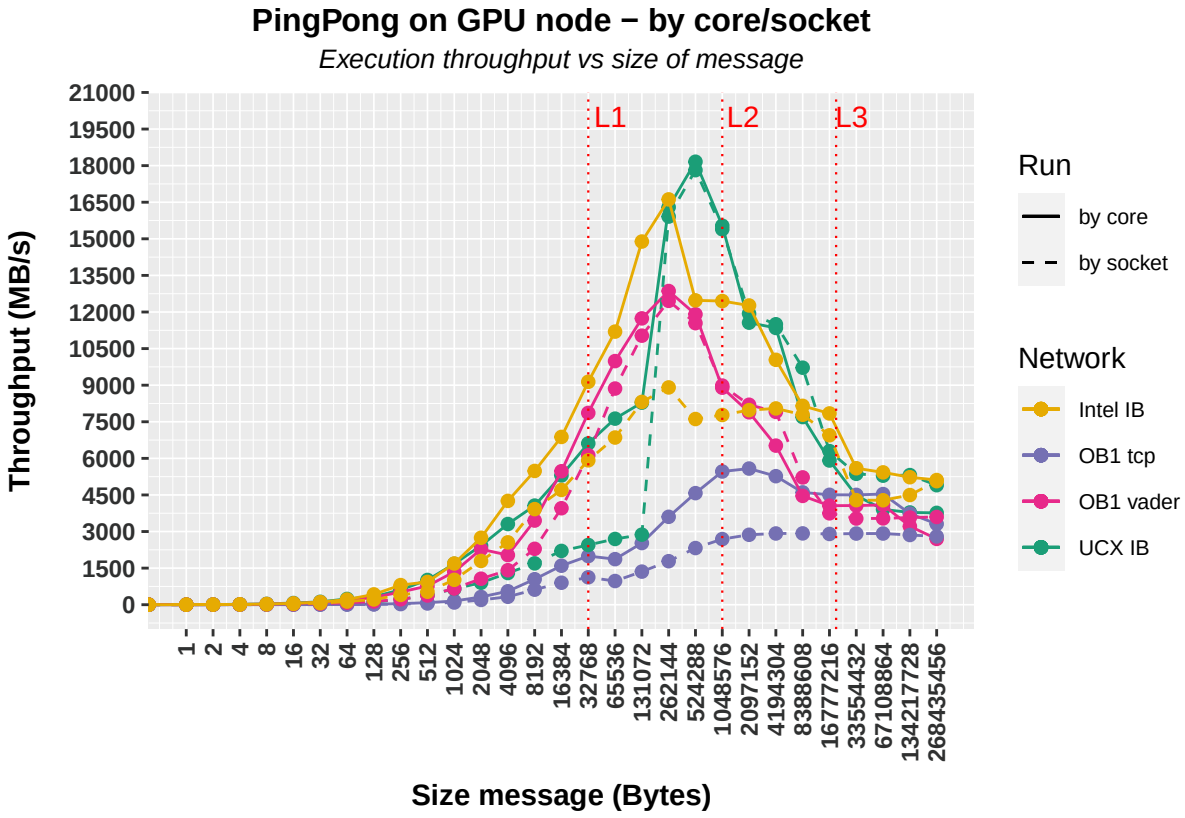
Now let's have a look at the GPU results in terms of throughput.



As can be inferred from the previous plot, Gpu nodes behave like thin node, no major difference pops out, but **GPU node is a bit slower (about 15%)** than thin node, *as already discovered in the exercise 1.*

This can be caused either by the fact that not being able to reverse a whole node (if not 2 in the across node configuration) or, more likely, by the different CPU frequencies and node configurations: remember that even if gpu CPU has a theoretical higher maximum frequency (2.70GHz) with respect to the one of the thin node (2.60GHz), these frequencies are much smaller when running simultaneously on several cores.

Cache size are the same as thin node, then cache effects are similar, as it is shown here:



Results

Here are shown the estimation via least square methods of the latency λ_{net} and bandwidth b_{net} of the networks used. To see the fit model estimation look in the folder *section2/mydata/csv-files/*.

THIN NODE		
	Latency (us)	Bandwith (MB/s)
By core		
UCX IB	0,196710	5384,612
Intel IB	0,234158	5485,757
OB1 vader	0,276249	3639,492
OB1 tcp	5,472147	5196,069
By socket		
UCX IB	0,414452	4612,174
Intel IB	0,432647	4208,231
OB1 vader	0,602071	3000,796
OB1 tcp	7,877553	3265,919
By node		
UCX IB	0,981388	12180,267
Intel IB	1,124170	12210,497
UCX ib0	10,277036	2764,394
OB1 tcp	16,036402	2784,559
UCX br0	15,943306	2433,027

GPU NODE		
	Latency (us)	Bandwith (MB/s)
By core		
UCX IB	0,239114	3760,817
Intel IB	0,289496	5129,400
OB1 vader	0,330979	2801,128
OB1 tcp	5,944020	3400,770
By socket		
UCX IB	0,479683	4949,990
Intel IB	0,474816	4890,206
OB1 vader	0,623670	3578,910
OB1 tcp	9,515453	2808,541
By node		
UCX IB	1,357744	12145,917
Intel IB	1,287712	12188,889
UCX ib0	12,945012	2432,176
OB1 tcp	15,480444	2118,792
UCX br0	17,777864	1920,804

Figure 5: Latency and bandwith fit estimation

Exercise 3

Jacobi solver - Performance model

In order to predict Jacobi model performance the following model is used:

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

L represent sub domain size (assuming it as cube), for us it is a constant. N represent the processes number. The L^3N quantity thus is considered the problem size, this amount of work is increased linearly in function of N, this is a weak scalability. About measure unit: performance is evaluated using $MLUP/s$, mega lactic update per sec.

T_s is time swept, is constant and can be estimated using the serial output.

T_c represent the communication time in seconds. This quantity can be modeled simple estimating the latency λ, bandwidth B and messages size C.

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

About C,assuming grid point as double:

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

About k: this quantity represent 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 by two taking into account the positive and negative direction.

Domain decomposition

Several domain decompositions are possible, a priory some of them can be discarded assuming poor performance due buffering, as example between $(12,1,1)$ $(1,12,1)$ $(1,1,12)$, the first one is the better and more memory friendly decomposition, it has the lowest buffering needs. This can be proved experimentally, the first configuration behave better than the other in terms of MLUP/s.