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**.

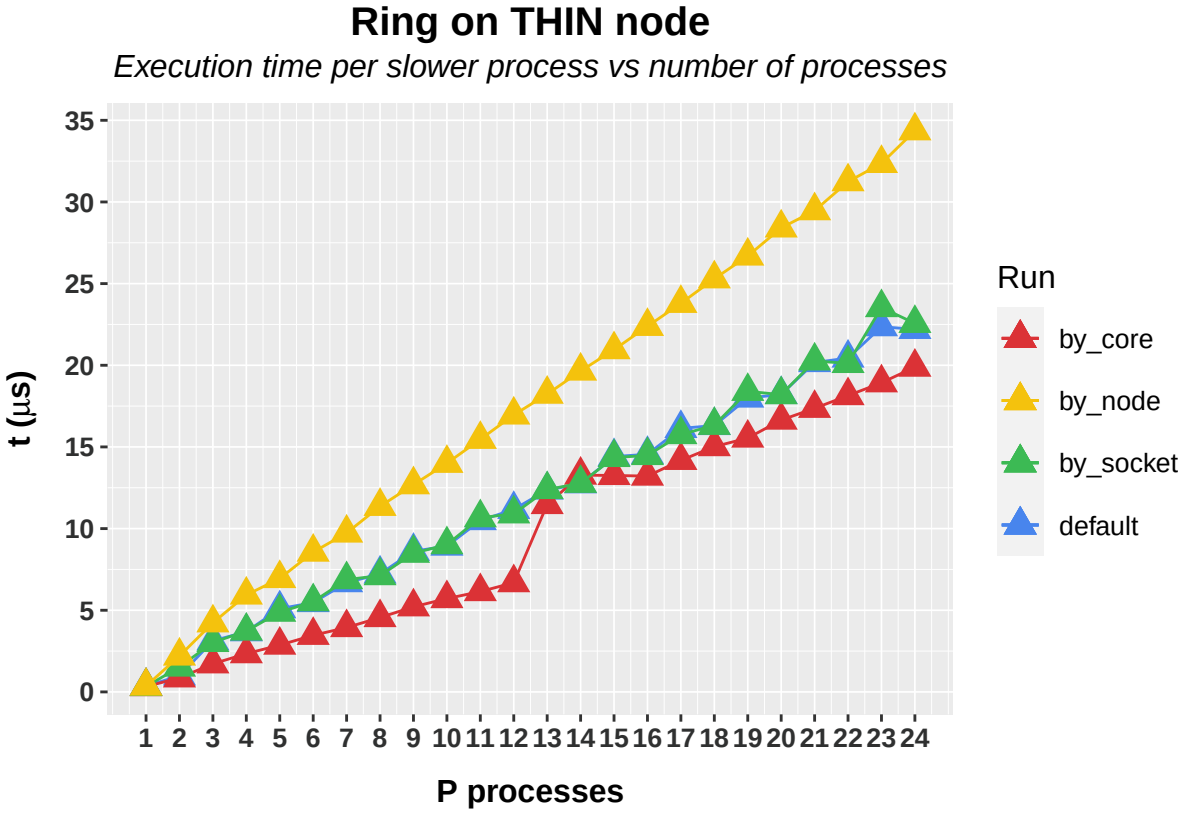
I have taken notes of the runtime (from the first send/receive to the last) when varying the number of processes P by means of the routine `MPI_Wtime()` which returns an elapsed time on the calling process in seconds.

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:

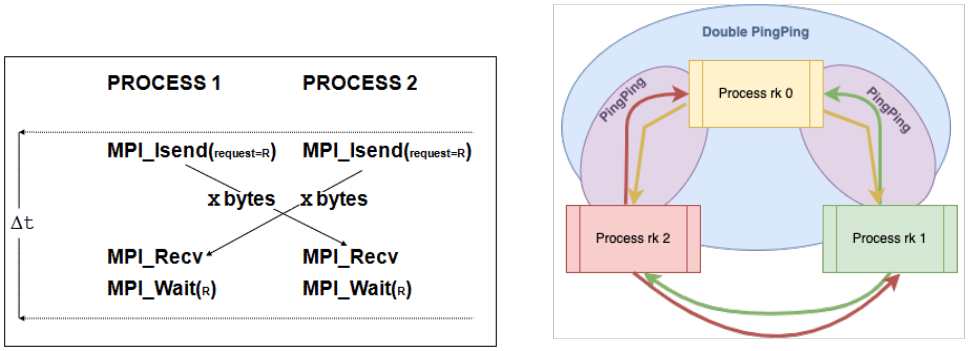


Figure 1: Fig.1 PingPing Pattern from Intel® MPI Benchmarks User Guide and Ring with P=3.

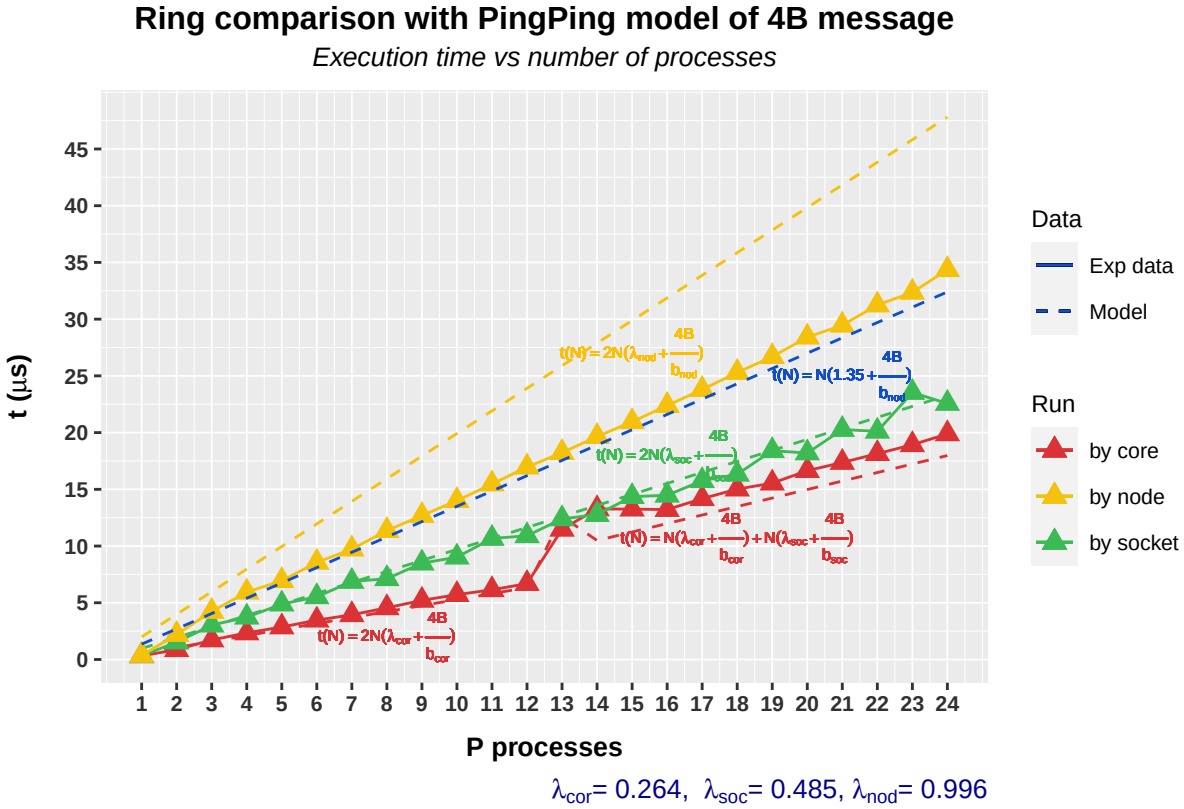
With IMB-MPI1 PingPing benchmark I was able to estimate the latency λ_{net} and bandwidth b_{net} on Infiniband network with different processes mappings: across nodes, sockets and cores. Since my t_{mean} is a measure of time in μs of a pair of opposite messages passing through all P process till returning 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:

- 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
```

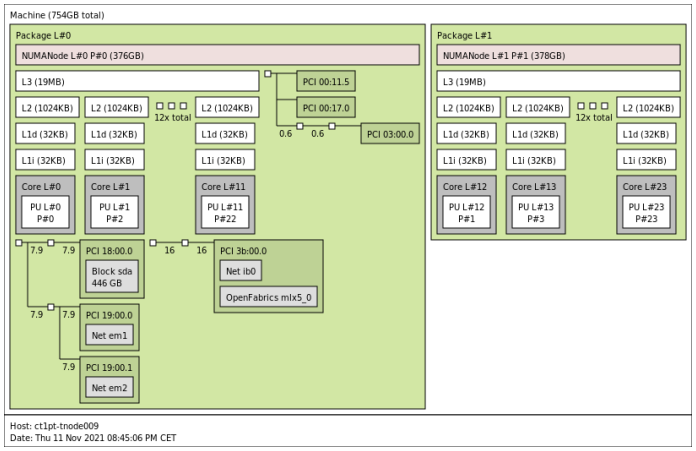


Figure 2: Fig.2 Topology on thin node

```
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  (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>
        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.**

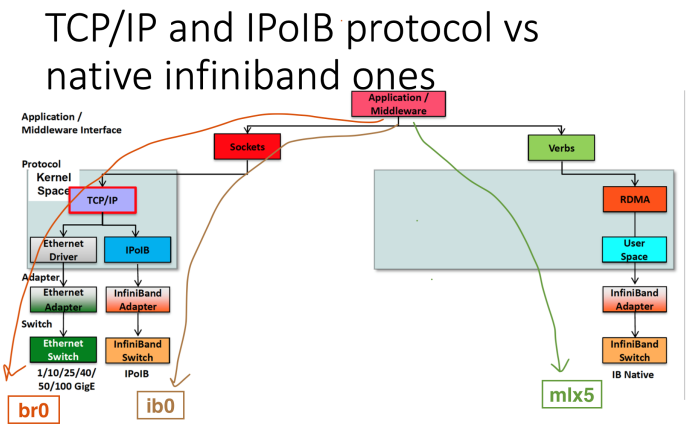


Figure 3: TCP/IP, IPoIB and native IB protocols

IMB-MPI1 Benchmark PingPong

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

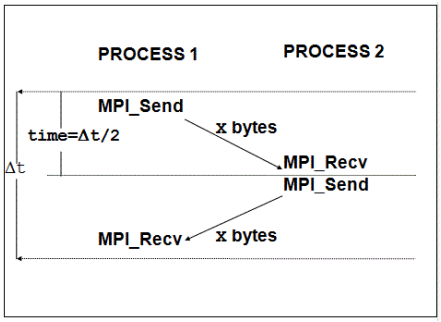


Figure 4: PingPong Pattern from Intel® MPI Benchmarks User Guide

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.

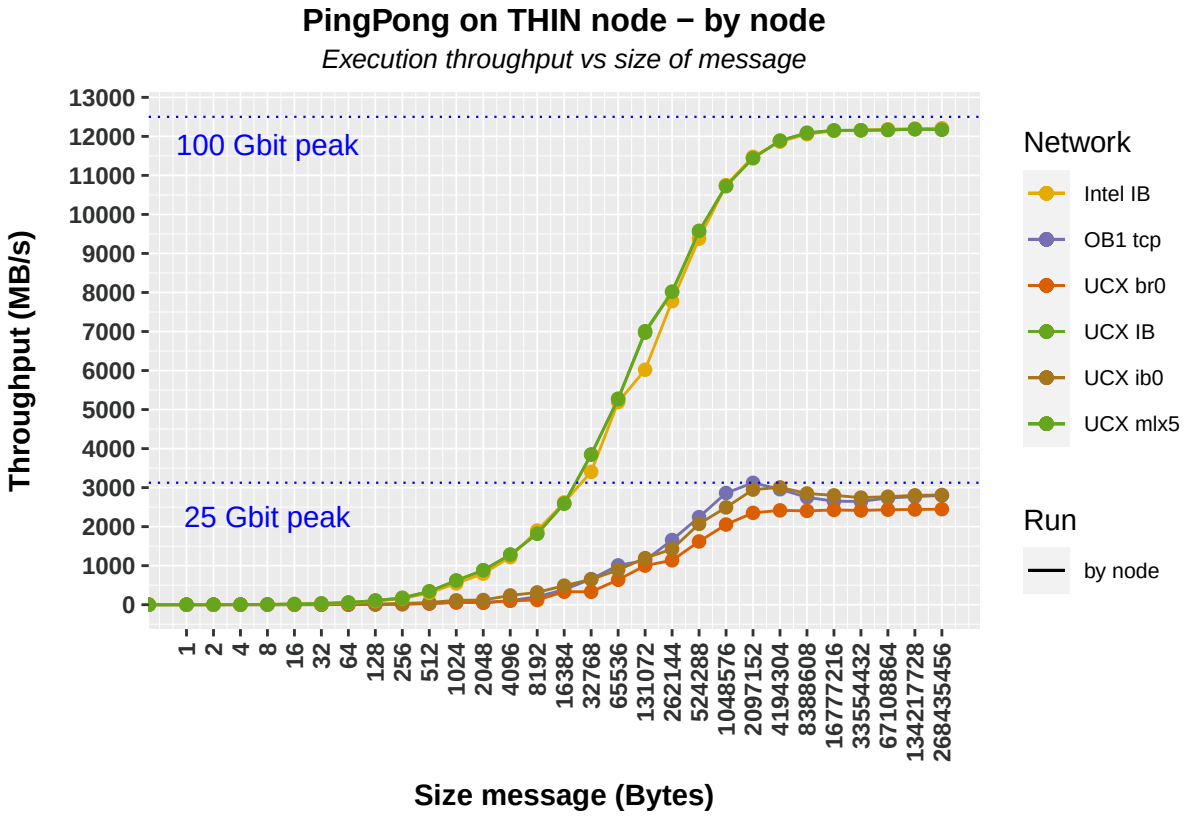
With **OpenMPI** implementation it is possible to specify the MPI frameworks and plugins (like pml and btl) that can be used while running. Across nodes, cores and sockets the following as been tested:

- *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.

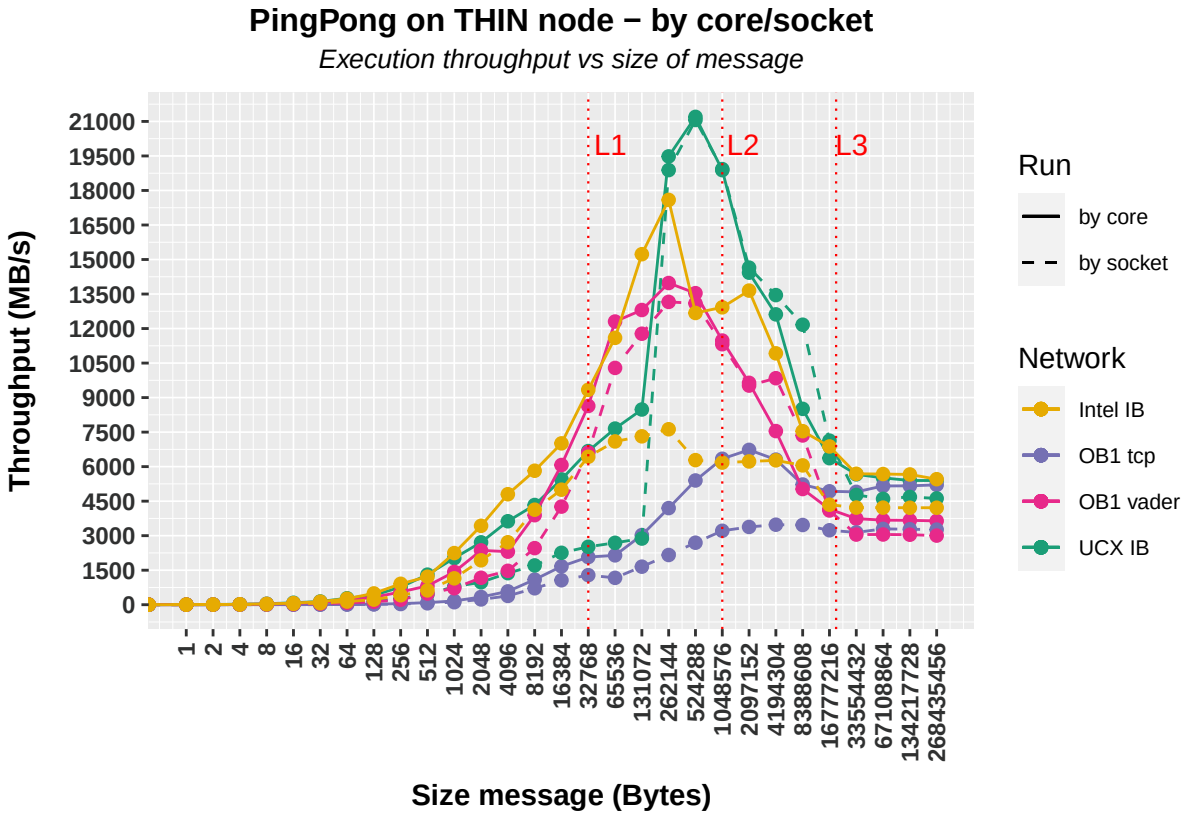


From this first plot it is evident that **with Intel MPI, UCX OpenMPI InfiniBand** (the default command) **and UCX native IB with mlx5_0:1** device (which is the same as the previous), **the asymptotic bandwith is very high compared to the others protocols/devices and similar to the one expected from 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.



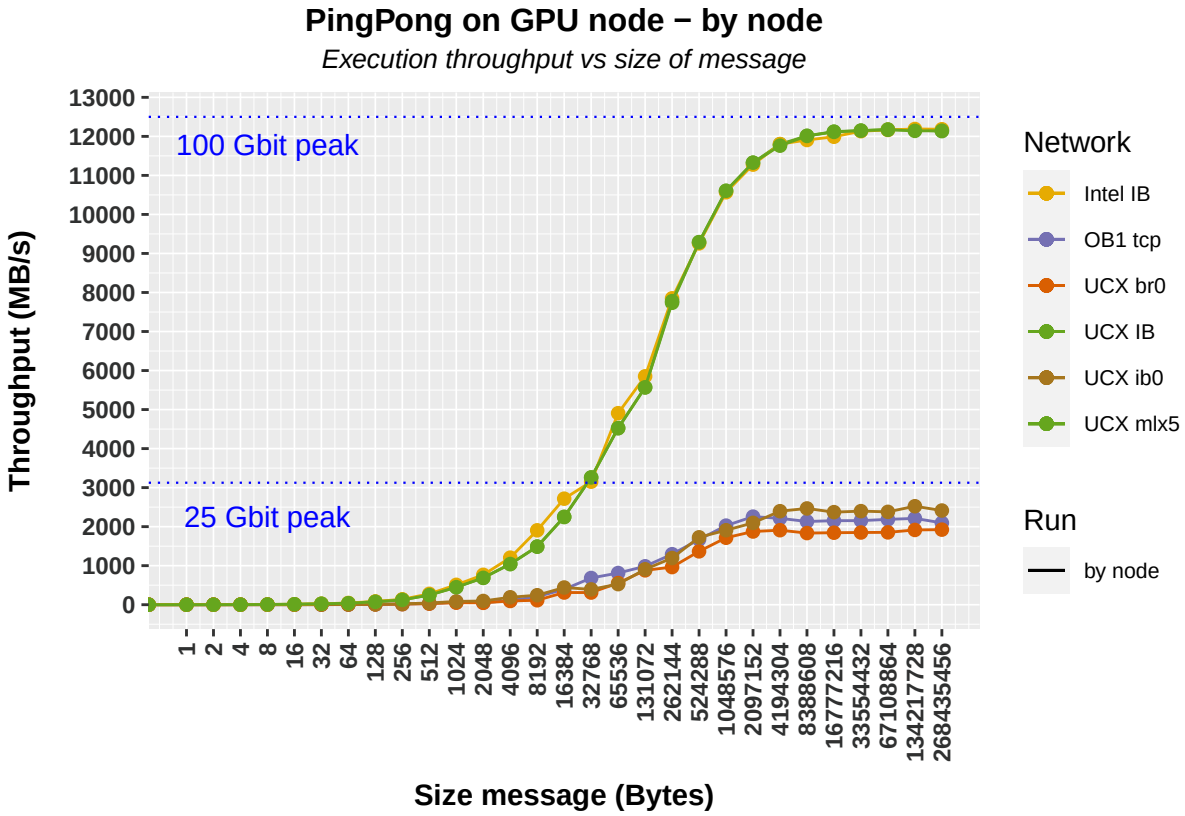
From first glance it is clear that these performance are smaller in bandwith, but better in latency (see table in 'Report section') when compared to across nodes configuration. Moreover, **mapping the processes in the same socket show often a better performance than mapping them in separate sockets**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

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.

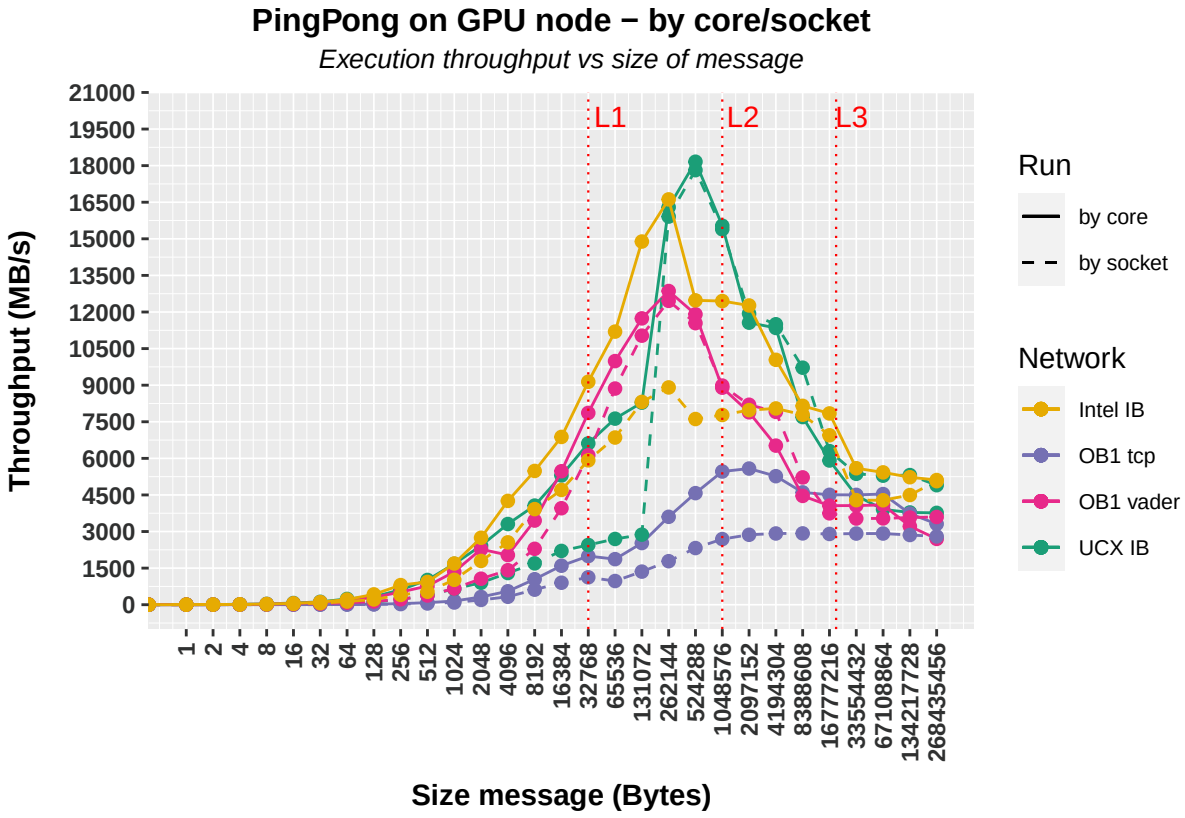
Now let's have a look at the GPU results in terms of throughput.



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.* (ricorda aggiungere dati sul gpu ex1)

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

It's a prototype for as stencil-based iterative method used in numerical analysis to solve partial differential equations.

In its most straightforward form, it can be used for solving the diffusion equation for a scalar function $\Phi(r,t)$:

$$\frac{\delta\Phi}{\delta t} = \Delta\Phi$$

on a rectangular lattice subject to Dirichlet boundary conditions. The differential operators are discretized using finite differences:

$$\frac{\delta\Phi(x_i,y_i)}{\delta t} = \frac{\Phi(x_i+1,y_i)+\Phi(x_i-1,y_i)-2\Phi(x_i,y_i)}{\delta x^2} + \frac{\Phi(x_i+1,y_i+1)+\Phi(x_i,y_i-1)-2\Phi(x_i,y_i)}{\delta y^2}$$

In each time step, a correction $\delta\Phi$ to $\Delta\Phi$ at coordinate (x_i,y_i) is calculated using the "old" values from the four next neighbor points. Of course, the updated Φ values must be written to a second array. After all points have been updated (a "sweep"), the algorithm is repeated till some basic convergence bounded by a threshold *eps*.

The following is a 2D Implementation of the Jacobi algorithm on an N × N lattice, with a convergence criterion added, taken by G.Hager and G.Wellein "Introduction to high performance computing for scientists and engineers":

```
double precision, dimension(0:N+1,0:N+1,0:1) :: phi
double precision :: maxdelta,eps
  integer :: t0,t1
  eps = 1.d-14 ! convergence threshold
  t0=0;t1=1
  maxdelta = 2.d0*eps
  do while(maxdelta.gt.eps)
  maxdelta = 0.d0
do k = 1,N
  do i = 1,N
    phi(i,k,t1) = ( phi(i+1,k,t0) + phi(i-1,k,t0)
                + phi(i,k+1,t0) + phi(i,k-1,t0) ) * 0.25
    maxdelta = max(maxdelta,abs(phi(i,k,t1)-phi(i,k,t0)))
enddo
  enddo
! swap arrays
    i = t0 ; t0=t1 ; t1=i
enddo
```

In a parallel computation, the core of the code implementation is similar but:

- convergence criterion needs to be computed globally (collective reduce operations)
- boundary layers must retrieve info form adjacent subdomains (halo layers exchange) in order to update their values.

For this exercise, we used as **Jacobi solver 3D program given by the professor Cozzini written in FORTRAN**.

Performance model

In order to predict 3D Jacobi model performance P the following model has been used:

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

where:

- L is the subdomain size, assuming it's a cube..
- $N = N_x N_y N_z$ is the number of the processes involved.
- $L^3 N$ is the problem size. The amount of work, therefore, increase linearly as a function of N; hence a weak scalability will be considered.
- $MLUP/s$ is the unit for measuring this performance (Mega Lactice Updates Per sec).
- T_s is the time for an entire sweep, thus it can estimated using the program as serial.
- T_c represent the communication time for total halo exchanges.

The latter is modeled by simply estimating the latency λ, bandwidth b of the network and messages size c.

In a 3D problem, the number of points used to estimate a point of the subdomain is not 4 (5 stencil case) but 6 instead (7 stencil case). When processes communicates whit each other they simultaneously sends each of the 6 point-to-point communication at the same time, so we need to consider full duplex data transfer for bandwidth.

Moreover we assume that the time for copying the halo to/from an intermediate buffer and the communication of a process with itself come at no costs. T_c then can be model as:

$$T_c(L, N) = \frac{c(L)}{b} + k\lambda[s] \qquad (**)$$

where:

- $c(L)$ is the amount of data volume transferred over a node's network link.
- b is the bidirectional bandwith of the network link
- λ is the latency
- k is the largest number of coordinate directions in which $N_i > 1$.

Then c is, :

$$c(L) = L^2 k \cdot 2 \cdot 8[B] \qquad (***)$$

where:

- k represents the number of directions in which the halo exchange occurs.
- L^2 is the halo surface to transmit.
- 2 is for taking into account the bidirectional halo exchange.
- 8 is for double conversion, assuming that the grid points in the subdomain are double floating points.

Domain decomposition

Since we use a **3D Jacobi FORTRAN program (column-major order)**, there are domains decomposition that are more suitable than others when dealing with contiguous location in memory for halo exchange.

In order to find the best decomposition possible, several tests has been done in order to find the most performing permutation of (N_x, N_y, N_z).

With $N = 6$ take $(6, 1, 1)$ $(1, 6, 1)$ $(1, 1, 6)$ decompositions when $L = 1200$ on a thin node: experimentally, the first configuration behave better than the other in terms of MLUP/s (oss. use large values of L to see this effect). Example:

Mapping	Nx	Ny	Nx	MLUP/s
	6	1	1	668
core	1	6	1	665
	1	1	6	665
	6	1	1	674
socket	1	6	1	671
	1	1	6	670

Instead when comparing always with $N = 6$, $(6, 1, 1)$ and $(3, 2, 1)$ the first has more buffering cost involved but the latter has more halo exchanges, thus both the two configurations will be taken into account.

Results

As a first step I have tried to **compile and run the code on single processor of a thin and gpu node to estimate the serial time on one single core** T_s.

I have used **OpenMPI implementation on ORFEO both on thin and gpu nodes** using native IB protocol.

The **size of the subdomain** $L = 700$ **was fixed and used both for gpu and thin** measurements since, due to RAM size on the gpu node (see `pbsnodes -ajS` in Exercise 2), it corrisponds to the maximum amount of memory availble for comparison between the two different nodes.

The Jacobi program was runned on:

- 4/8/12 processes within the same thin node pinning the MPI processes within the same socket.
- 4/8/12 processes within the same thin node pinning the MPI processes across two sockets.
- 12/24/48 processes using two thin nodes.
- 12/24/48 processes using one gpu node mapping by core (hyperthreading is enabled).
- 12/24/48 processes using one gpu node mapping by socket (hyperthreading is enabled).

The *jacobi3D* program prints out the performance in $MLUP/s$ **as last column for each run: I have taken the mean of the 10 results**. Along with the performance measurements, I took the elapsed time by means of `/usr/bin/time -f "user : %U system: %S elapsed: %e CPU: %P CMD: %C \n"` run command.

The following is an example of running with $L = 700$ and $N = (4, 1, 1)$:

```
# Threadlevel provided:        3 out of        0        1        2        3
  spat_dim , proc_dim, PBC ?
           1 -Dim Input          2800           4 T
           2 -Dim Input           700           1 T
           3 -Dim Input           700           1 T
#   0 Process grid   4   1   1
           0           3           2
           2           3           2
           3           3           2
           1           3           2
#   0  Allocated 2 arrays with a total of  5535.17      Mbytes
# StartResidual    0.00000000000
  4  Maxtime , Mintime + JacobiMi , JacobiMa  3.06243716300  3.06242734200  2.98796024800 2.99414415400
  Residual 12273333.3238  MLUPs 448.009192344
  4  Maxtime , Mintime + JacobiMi , JacobiMa  3.06078858000  3.06078676400  2.98838307400 2.99486265400
  Residual 2041666.66667  MLUPs 448.250496282
  4  Maxtime , Mintime + JacobiMi , JacobiMa  3.06090752400  3.06090532700  2.98793178100 2.99505510000
  Residual 434799.382550  MLUPs 448.233077688
  4  Maxtime , Mintime + JacobiMi , JacobiMa  3.06095547000  3.06095308700  2.98808474500 2.99490125800
  Residual 129179.526716  MLUPs 448.226056683
  4  Maxtime , Mintime + JacobiMi , JacobiMa  3.06516349800  3.06516215100  2.99001243700 2.99797766000
  Residual 52905.8427676  MLUPs 447.610706866
  4  Maxtime , Mintime + JacobiMi , JacobiMa  3.06256750200  3.06256510400  2.98815335100 2.99561585600
  Residual 27258.4295825  MLUPs 447.990125639
  4  Maxtime , Mintime + JacobiMi , JacobiMa  3.06197104500  3.06196926100  2.98880531000 2.99486067300
  Residual 16225.1238619  MLUPs 448.077391927
  4  Maxtime , Mintime + JacobiMi , JacobiMa  3.06272575000  3.06272477100  2.98845891200 2.99545644500
  Residual 10594.4563848  MLUPs 447.966978434
  4  Maxtime , Mintime + JacobiMi , JacobiMa  3.06178146800  3.06177916200  2.98808213100 2.99487845400
  Residual 7372.72712122  MLUPs 448.105135634
  4  Maxtime , Mintime + JacobiMi , JacobiMa  3.06086679800  3.06086465500  2.98821124300 2.99498761300
  Residual 5375.90215905  MLUPs 448.239041600
```

The following tables show the results of performance from relations $(*)$, $(**)$ and $(***)$ for both thin and gpu node with different mapping of the processes.

For evaluating T_c, the bidirectional bandwith b and latency λ considered are taken by Intel® IMB-MPI1 PingPing Benchmark.

THIN

BY CORE

L	Lat [us]	B [MB/s]	Ts [s]
700	0,222	6372,991	2,99

N	Nx	Ny	Nz	2k	c [MB]	Tc [s]	P [MLUPs]	Pexp [MLUPs]	T elapsed [s]	P(1)*N / P(N)	ΔP/Pth [%]
1	1	1	1	2	0,0000000	0,0000000	114,7157191	112,2617203	54,26	1,0000000	2,14
4	4	1	1	2	7,4768066	0,0011736	458,6828323	448,0708203	54,44	1,0021784	2,31
4	2	2	1	4	14,9536133	0,0023473	458,5029296	448,9819070	54,37	1,0001447	2,08
8	8	1	1	2	7,4768066	0,0011736	917,3656647	894,6112024	54,59	1,0038928	2,48
8	4	2	1	4	14,9536133	0,0023473	917,0058593	896,9955594	54,61	1,0012243	2,18
8	2	2	2	6	22,4304199	0,0035209	916,6463360	894,3670611	54,61	1,0041669	2,43
12	12	1	1	2	7,4768066	0,0011736	1376,0484970	1329,530125	55,27	1,0132457	3,38
12	4	3	1	4	14,9536133	0,0023473	1375,5087889	1315,655181	55,70	1,0239314	4,35
12	3	2	2	6	22,4304199	0,0035209	1374,9695040	1324,69806	55,42	1,0169417	3,66
12	6	2	1	4	14,9536133	0,0023473	1375,5087889	1323,84809	55,46	1,0175946	3,76

THIN

BY SOCKET

L	Lat [us]	B [MB/s]	Ts [s]
700	0,485	5530,801	2,99

N	Nx	Ny	Nz	2k	c [MB]	Tc [s]	Ptheo [MLUPs]	Pexp [MLUPs]	T elapsed [s]	P(1)*N / P(N)	ΔP/Pth [%]
1	1	1	1	2	0,0000000	0,0000000	114,7157191	112,3160269	54,24	1,0000000	2,09
4	4	1	1	2	7,4768066	0,0013528	458,6553587	448,5776102	54,35	1,0015304	2,20
4	2	2	1	4	14,9536133	0,0027056	458,4480287	448,3071757	54,36	1,0021345	2,21
8	8	1	1	2	7,4768066	0,0013528	917,3107173	888,9560134	54,74	1,0107679	3,09
8	4	2	1	4	14,9536133	0,0027056	916,8960574	895,1857155	54,49	1,0037339	2,37
8	2	2	2	6	22,4304199	0,0040585	916,4817721	895,8976873	54,47	1,0029362	2,25
12	12	1	1	2	7,4768066	0,0013528	1375,9660760	1332,875523	54,83	1,0111914	3,13
12	4	3	1	4	14,9536133	0,0027056	1375,3440861	1336,471697	54,84	1,0084705	2,83
12	3	2	2	6	22,4304199	0,0040585	1374,7226582	1342,513155	54,63	1,0039323	2,34
12	6	2	1	4	14,9536133	0,0027056	1375,3440861	1342,499263	54,62	1,0039427	2,39

THIN

BY NODE

L	Lat [us]	B [MB/s]	Ts [s]
700	1,020	11945,604	2,99

N	Nx	Ny	Nz	2k	c [MB]	Tc [s]	Ptheo [MLUPs]	Pexp [MLUPs]	T elapsed [s]	P(1)*N / P(N)	ΔP/Pth [%]
1	1	1	1	2	0,0000000	0,0000000	114,7157191	112,1941234	54,26	1,0000000	2,20
12	12	1	1	2	7,4768066	0,0006279	1376,2995854	1334,309648	54,80	1,0090083	3,05
12	4	3	1	4	14,9536133	0,0012559	1376,0106634	1334,824422	54,75	1,0086192	2,99
12	3	2	2	6	22,4304199	0,0018838	1375,7218626	1342,339290	54,57	1,0029726	2,43
12	6	2	1	4	14,9536133	0,0012559	1376,0106634	1341,984259	54,58	1,0032379	2,47
24	24	1	1	2	7,4768066	0,0006279	2752,5991708	2680,570156	54,80	1,0045098	2,62
24	12	2	1	4	14,9536133	0,0012559	2752,0213268	2683,971787	54,75	1,0032367	2,47
24	8	3	1	4	14,9536133	0,0012559	2752,0213268	2678,966198	54,78	1,0051112	2,65
24	6	4	1	4	14,9536133	0,0012559	2752,0213268	2676,792858	54,79	1,0059273	2,73
24	6	2	2	6	22,4304199	0,0018838	2751,4437253	2681,906202	54,82	1,0040094	2,53
24	4	3	2	6	22,4304199	0,0018838	2751,4437253	2674,085758	54,85	1,0069456	2,81
48	48	1	1	2	7,4768066	0,0006279	5505,1983416	5260,937918	56,39	1,0236422	4,44
48	24	2	1	4	14,9536133	0,0012559	5504,0426535	5247,754482	56,51	1,0262138	4,66
48	12	4	1	4	14,9536133	0,0012559	5504,0426535	5249,958724	56,58	1,0257829	4,62
48	12	2	2	6	22,4304199	0,0018838	5502,8874506	5215,275131	56,73	1,0326048	5,23
48	8	6	1	4	14,9536133	0,0012559	5504,0426535	5222,119960	56,67	1,0312513	5,12
48	6	4	2	6	22,4304199	0,0018838	5502,8874506	5244,173203	56,52	1,0269146	4,70

The model seems quite accurate for the thin node: our data seems to differ of about 3% with respect to the model, expect for the node mapping with 12 process in which it arise up to 5%.

Mapping by socket lead a sightly better performance respect core mapping and this is not what expected by theoretical model. **Socket mapping leads to poor communication performance but it exploits better the memory allocation**. When one socket is filled with 12 processes the entire grid is stored on socket competence, but spreading processes across 2 socket lead to better exploit of bandwidth using different memory controller and then all memory channel available. This solver at each iteration elaborates an huge amount of memory, given $L = 700$ and 12 worker at least 250 GB of ram are elaborated and **more than one third of the memory is saturated**.

Also with map-by node option experimentally we obtain better results when several processes are involved.

GPU

BY CORE

L	Lat [us]	B [MB/s]	Ts [s]
700	0,279	6300,328	4,317

N	Nx	Ny	Nz	2k	c [MB]	Tc [s]	Ptheo [MLUPs]	Pexp [MLUPs]	T elapsed [s]	P(1)*N / P(N)	ΔP/Pth [%]
1	1	1	1	2	0	0,0000000	79,4533241	77,4580241	76,52	1,0000000	2,51
12	12	1	1	2	7,4768066	0,0011873	953,1777393	850,5266604	88,24	1,0928479	10,77
12	4	3	1	4	14,9536133	0,0023746	952,9157340	849,5713458	88,14	1,0940768	10,85
12	3	2	2	6	22,4304199	0,0035619	952,6538726	849,6338328	88,03	1,0939963	10,81
12	6	2	1	4	14,9536133	0,0023746	952,9157340	850,3519011	88,10	1,0930725	10,76
24	24	1	1	2	7,4768066	0,0011873	1906,3554787	1690,635257	89,12	1,0995823	11,32
24	12	2	1	4	14,9536133	0,0023746	1905,8314680	1700,242117	88,86	1,0933693	10,79
24	8	3	1	4	14,9536133	0,0023746	1905,8314680	1699,031920	88,91	1,0941481	10,85
24	6	4	1	4	14,9536133	0,0023746	1905,8314680	1700,315941	88,96	1,0933219	10,78
24	6	2	2	6	22,4304199	0,0035619	1905,3077453	1698,989954	88,89	1,0941751	10,83
24	4	3	2	6	22,4304199	0,0035619	1905,3077453	1699,548395	88,89	1,0938156	10,80
48	48	1	1	2	7,4768066	0,0011873	3812,7109574	2538,198883	114,9	1,4648124	33,43
48	24	2	1	4	14,9536133	0,0023746	3811,6629359	2534,726547	115,1	1,4668190	33,50
48	12	4	1	4	14,9536133	0,0023746	3811,6629359	2523,002058	115,4	1,4736354	33,81
48	12	2	2	6	22,4304199	0,0035619	3810,6154905	2550,307105	114,4	1,4578578	33,07
48	8	6	1	4	14,9536133	0,0023746	3811,6629359	2549,905602	114,5	1,4580874	33,10
48	6	4	2	6	22,4304199	0,0035619	3810,6154905	2527,282456	115,3	1,4711395	33,68

GPU

BY SOCKET

L	Lat [us]	B [MB/s]	Ts [s]
700	0,54	5471,172	4,317

N	Nx	Ny	Nz	2k	c [MB]	Tc [s]	Ptheo [MLUPs]	Pexp [MLUPs]	T elapsed [s]	P(1)*N / P(N)	ΔP/Pth [%]
1	1	1	1	2	0	0,0000000	79,4533241	77,4580241	76,52	1,0000000	2,51
12	12	1	1	2	7,476806641	0,0013677	953,1379266	900,1880882	83,73	1,0325579	5,56
12	4	3	1	4	14,95361328	0,0027353	952,8361556	900,3287287	83,15	1,0323966	5,51
12	3	2	2	6	22,43041992	0,0041030	952,5345757	893,0782092	83,52	1,0407782	6,24
12	6	2	1	4	14,95361328	0,0027353	952,8361556	899,9480417	83,18	1,0328333	5,55
24	24	1	1	2	7,476806641	0,0013677	1906,2758532	1698,967619	88,97	1,0941895	10,88
24	12	2	1	4	14,95361328	0,0027353	1905,6723113	1699,917744	89,11	1,0935780	10,80
24	8	3	1	4	14,95361328	0,0027353	1905,6723113	1699,284762	88,88	1,0939853	10,83
24	6	4	1	4	14,95361328	0,0027353	1905,6723113	1698,867199	88,94	1,0942542	10,85
24	6	2	2	6	22,43041992	0,0041030	1905,0691514	1699,6994	88,86	1,0937184	10,78
24	4	3	2	6	22,43041992	0,0041030	1905,0691514	1699,090177	88,90	1,0941106	10,81
48	48	1	1	2	7,476806641	0,0013677	3812,5517065	2528,729925	115,50	1,4702974	33,67
48	24	2	1	4	14,95361328	0,0027353	3811,3446225	2542,104349	114,50	1,4625620	33,30
48	12	4	1	4	14,95361328	0,0027353	3811,3446225	2531,667816	115,00	1,4685912	33,58
48	12	2	2	6	22,43041992	0,0041030	3810,1383027	2546,49542	114,30	1,4600400	33,17
48	8	6	1	4	14,95361328	0,0027353	3811,3446225	2531,779431	114,60	1,4685265	33,57
48	6	4	2	6	22,43041992	0,0041030	3810,1383027	2542,648993	114,50	1,4622487	33,27

For the gpu node, the model doesn't seem quite accurate: data in the socket mapping differs of about 5%, 10%, 33% (using 12, 24, 48 processes respectively) with respect to the model, whilst in the core mapping they do about 10%, 10%, 33%.

On gpu node differences on socket and core mapping are even bigger: **again we can guess the memory as main cause**. **Gpu node setup even if has higher CPU frequency, doesn't exploit all possible memory channel**. When worker number grow over physical core number the real performance, due hyper-threading, goes dramatically down respect the model. Also the execution time show poor performance with 48 workers.

Exercise 1

3D matrix sum

Since a 3D array in memory can be represented with a unique linear array, the most effective way to sum it in parallel is using collective operation.

The shape of the array doesn't make any difference, also the virtual topology and its relative domain decomposition. We can assume that the problem is not sensible to any topology, since that there is no need of communication between neighbors. The most efficient way is to use `MPI_Scatterv` and `MPI_Gatherv` collective routine.

The expectations for this code are not so high in terms of scalability, matrix sum assuming n elements each, require $2n$ memory read accesses, n memory write, a lot of communication and **only** n floating point operations. Parallel portion of code then is small respect serial one. **Using Ahmdal's law we can make a prediction of speed up bound**.

Table 2: 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

Testing parallel code with two $2400 \times 1000 \times 700$ matrices filled with double and using only one processor we can see that the parallel part take only 2.63 seconds over a total 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 take into account of matrix initialization and error checking.

This graph represent then the theoretical maximum speedup supported by Ahmdal's law assuming roughly only 5% of parallel code and communication code and serial code in remaining part and fixed respect processors numbers. This experimentally is a good approximation and catch the trend between [1,24] processes.