Assignment 1: High Performance Computing

Matteo Boi (s225241)

December 30, 2021

GitHub Repository: https://github.com/BoiMat/HPC.git

1 Ring

Aim of the exercise is to implement a program with N processes disposed on a ring. It consists in a 1D topology in which each process communicates only with its neighbours along one direction and boundary conditions allow communication between first and last one.

1.1 Methodology

The setup is done by the mean of virtual topology, an extra attribute that can be given to an intracommunicator that allows logical process arrangement in a topological pattern. Virtual topology can be exploited by the system in the assignment of processes to physical processors, and may help to improve the communication performance on the machine. The setup is shown in Figure 1.

- For the first iteration, each process P sends a message containing its $rank$ to its left neighbor $P-1$ and $-rank$ to its right neighbor $P+1$. It also receives two messages, one from left and one from right.
- From the second iteration onwards, each process adds its rank to the message received from left and subtracts it to the one coming from right.
- The stopping condition is given when each process receives back the original messages, i.e. when it receives the original tag.

With this setup, the number of iteration is exactly equal to the number of processes in a ring, that is the number of passes it takes the first message to go back to the original sender. Therefore, the expected behavior is to have a linear increase in the run-time of the program. To study if the hypothesis is correct, the walltime to run the ring one million times has been taken, thanks to the $MPI\_Wtime()$ call. Since different processes can take more or less time than others, a $MPI\_Barrier()$ call is invoked before the starting time and end time. Since each process invokes the $MPI\_Wtime()$ call, only the walltime registered by the root process has been taken into account.

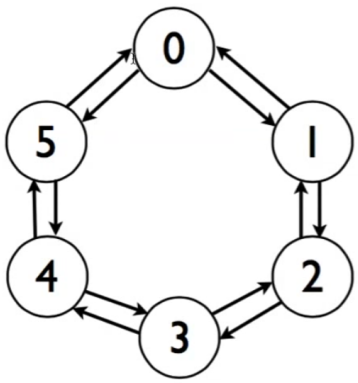

Figure 1: 6 processes arranged in a ring.

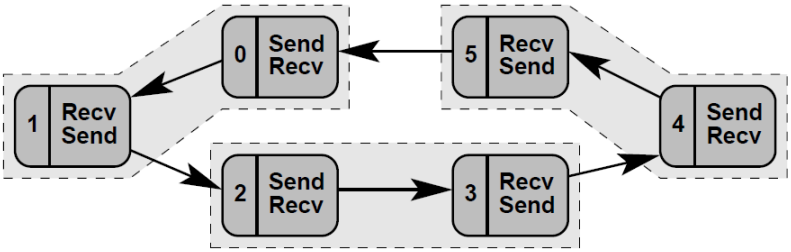

Figure 2: Simple solution for the deadlock problem in the ring.

1.2 Discussion

Given the fact that there are two $MPI\_Send()$ and two $MPI\_Recv()$ calls happening at the same time it is intuitive to think that there is a strong probability to fall in a deadlock. Considering just one direction, each process first sends the message to its left neighbor and then receives one from its right neighbor. If $MPI\_Send()$ is synchronous, all processes call it first and then wait forever until a matching receive gets posted.

1.2.1 Synchronous solution

A first simple solution is to interchange $MPI\_Send()$ with $MPI\_Recv()$ calls on all odd-numbered processes (or vice versa) so that every send has a matching receive, as show in Figure 2. Of course this is not the best solution, because while even processes are sending, odd ones are just waiting to receive and so only half of the possible communication links can be active at any time.

1.2.2 Non-blocking solutions

Better solutions can be found using non-blocking communication, here three are presented:

- **MPI_Send - MPI_Recv**:

  Actually the $MPI\_Send()$ call is not always blocking. It uses the standard communication mode, i.e. it is up to MPI to decide whether outgoing messages are buffered or not. The reason for this behaviour is that MPI tries to be as portable as possible, since most systems will run out of buffer space as message size increases but buffering can improve performance of some programs. That is why MPI chooses to buffer small-size messages. In this case the send call can complete before a matching receive is invoked. The last thing to point out is that standard mode send is non-local: a matching receive has to occur for the successful completion of the call. Thanks to this behaviour it is not necessary to resort to any "actual non-blocking call" to achieve the same result. It is sufficient to have Send calls and matching Receive calls in the correct order.

- **MPI_Isend - MPI_Recv**:

  Unlike the standard mode send, a buffered mode send is local, and so it does not need a matching receive to complete. However, a non-blocking operation immediately returns, without waiting for the underlying MPI routine to complete. An invocation of the MPI_Wait call is then needed, to wait until the routine has completed before the next iteration.

- **MPI_Isend - MPI_Irecv**:

  At last, a full non-blocking implementation is proposed. A non-blocking Receive call can improve performance, since it can reduce latency by being posted early. In fact the message-passing model implies that the communication is initiated by the sender. Overhead can be reduced if the receive call is already posted when the send call is started. However, a blocking receive can complete only when matched by a send. Therefore a non-blocking receive allows to reduce overhead without blocking the receiver. After the four calls, since all return before the underlying routines end, an invocation of $MPI\_Waitall$ call is invoked, to check whether all the calls have finished before going on with the next iteration.

1.3 Results

As explain in the Methodology section, the timings for 1 million runs of the programs have been. In Figure 3 the timings of the four models are plotted against the number of processes in the ring, from 1 to 48.

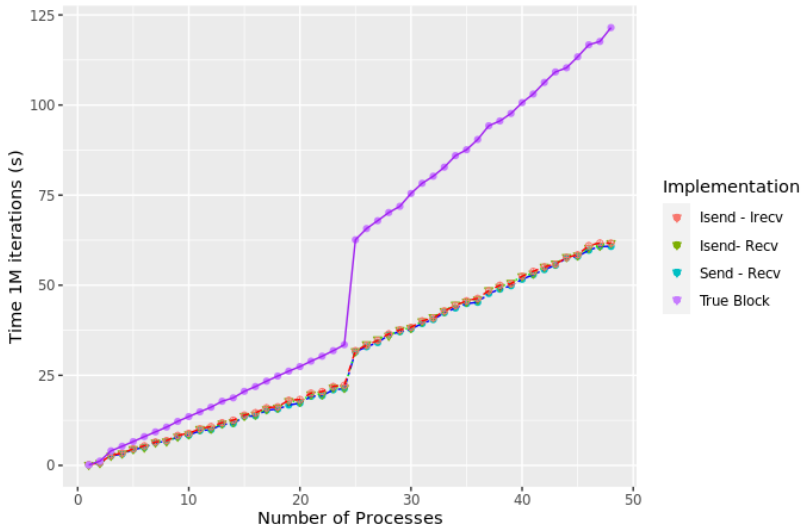

Figure 3: Measured timings plotted against the number of processes for the four implementations described.

	λ **true blocking**	λ **full non-blocking**
Intranode	1.417544	0.959813
Internode	2.54192	1.33861

Table 1: Latency estimated for the true blocking implementation and full non-blocking one.

As it is visible from the plot, there is a huge gap between the non-blocking approaches and the true blocking one (even-odd approach). This behaviour is clear looking at the implementations: in the even-odd approach to ensure that the send is actually synchronous (full blocking) and avoid any deadlock while half of the processes send the message the other half is just waiting for their receive calls to be matched. In the non-blocking implementations instead the processes initialize the calls, without actually waiting for their completion and going on with the next instruction. For what concerns these implementations, they all obtained really similar results. That is due to the fact that the program is a really simple one and not much work has to be done between sends and receives. In addition, every implementation has the same kind of behaviour, with a sort of barrier that forces the faster processes to wait the slower ones before each new iteration.

Another important feature in the graph is the "jump" all implementations present when the number of processes involved in the ring goes from 24 to 25. This is due to the fact that from 1 to 24 processes, they can be bounded to cores situated in the same physical node and so the time for intranode communication is much lower than the internode one. Intead from 25 processes on at least one process is in a different core with respect to the others. Since each process must wait until all have ended the communication, processes on the same node (faster communication) must wait until the ones that have to communicate from different nodes are done.

To study in a better way this behaviour a theoretical model can be used. If we consider the simple full non-blocking network model:

$$T_{comm} = \lambda + \frac{size}{B_a} \tag{1}$$

where λ is the latency, $size$ is the message size and B_a is the asymptotic bandwidth. In this model the latency represents the communication time for one iteration, i.e. the time to send two messages and receive other two. Since the messages contain the rank that is just an integer (4 bytes), they are really small compared to the bandwidth, so the second term is negligible and we can just focus on the latency. As already mentioned, the number of iterations is equal to the number of processes in the ring. Eq 1 can be rewritten for the ring case:

$$T_{ring} = \lambda * N_{procs} \tag{2}$$

i.e. the time for completing a run of the program depends on the communication time for completing one iteration of the slowest process multiplied by the number of iterations. This model describes a linear relation between the time for completion and the number of processes with coefficient λ, that can be estimated via least square fit. In Figure 4 the fitted model is shown for the true blocking implementation and full non-blocking one for both intranode and internode communication. On the GitHub repository a csv file for each implementation is present with measured and estimated times.

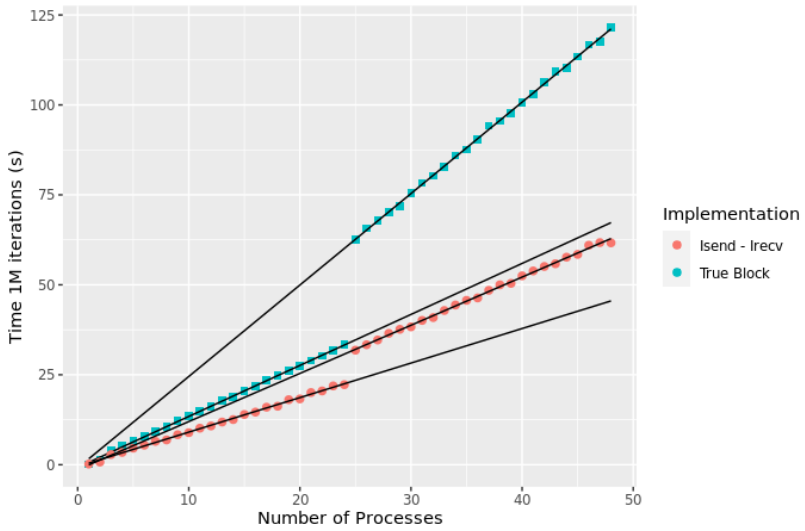

Figure 4: Least square fit for true blocking and full non-blocking implementation.

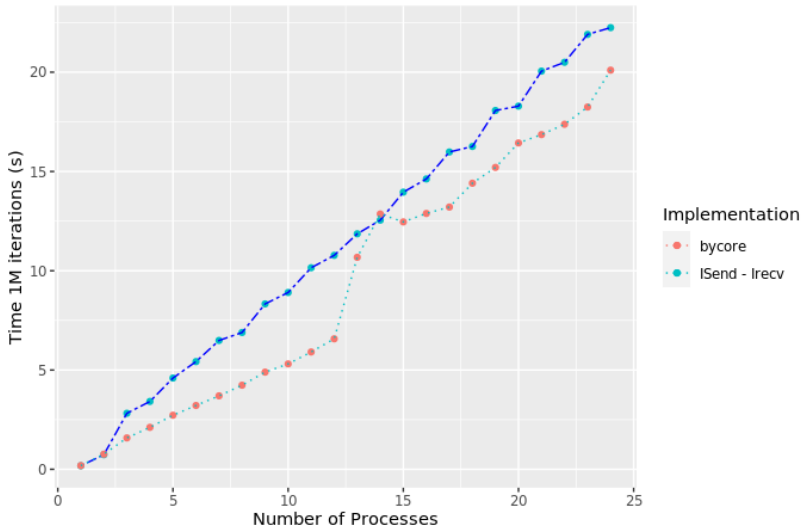

Figure 5: Timings from 1 to 24 processes of the full non-blocking implementation without any specification and with the map-by-core flag.

In table 1 the estimated latency is reported regarding the two implementations shown in the previous plot. It validates the previous beliefs: communication time in case of the non-blocking implementation is much faster than the true blocking one. This is however an ideal case, since we are considering that the communication time to send and receive the messages is the only overhead in the program. This is not true in the real case, there are other factors influencing the timings like the time to setup the communication between processes. One important thing to point out is that everything presented is done for one million runs of the program to make everything more readable, thus even the estimated latency has to be rescaled to obtain the results for one single run.

To end this section, one last behaviour needs to be explained. In fact, without any explicit specification by the user, MPI bounds processes to physical cores on the same node but alternating on different sockets. So the first process will be bounded to socket 0 - core 0 and the second to socket 1 - core 0. If we want to avoid this behaviour, the flag *–map-by core* must be specified when running the program. Thanks to it MPI will bound until 12 processes on the same socket, and only from 13 processes onward the behaviour will be again the "alternating sockets" one. This is visible in Figure 5. In the first part we see a clear drop in timings, given the smaller latency between processes on the same socket and another jump is present as the one previously explained for different nodes. As before, it is exactly between 12 and 13 processes. In fact while before the communication time was between cores on the same socket, from that point on the time for one iteration is dictated (since all have to wait the slowest one) by the communication between the cores on different sockets.

	1D topology	2D topology	3D topology	No topology
2400x100x100	0.4961092	0.4979744	0.4963410	0.4956637
1200x200x100	0.4973468	0.4974063	0.4971237	0.4962343
800x300x100	0.4973468	0.4974063	0.4971237	0.4956099

Table 2: Mean runtimes [s] for the matrix-matrix addition with and without the use of a virtual topology

2 3D Matrix

Aim of the exercise is to implement a matrix - matrix addition in parallel using virtual topology to distribute the processes in a 1D, 2D or 3D layout. Provided matrix dimensions are:

- $2400 * 100 * 100$
- $1200 * 200 * 100$
- $800 * 300 * 100$

2.1 Methodology & Discussion

When the program is run, 4 arguments are accepted via command line as main function arguments. A first argument defines the number of virtual dimensions in which the processes would be disposed via the $MPI\_Cart\_create()$ call in the new communicator. Then, since the program must be able to vary the dimensions of the matrices, three integer are accepted as arguments of the main function representing the three dimensions of the matrices. Matrix - matrix addition requires that the two matrices have the same dimensions to be added, so the three integers are enough.

To implement a program as general as possible, the task of building the two matrices is assigned to the process with rank 0. Of course given the nature of the problem a better solution to improve performances would be to make each process construct part of the matrices, in this way one can avoid the communication cost of sending part of the matrices to each process. However, this would be a problem if, for example, the matrices are fed to the program from somewhere else, as it most likely would be in a real case scenario. Matrices are built as 1-dimensional arrays of size given by the multiplication of the three dimensions, that is also the way they are stored in memory. They are initialize using double precision random numbers.

When process 0 is done initializing the matrices, via the $MPI\_Scatter()$ call it sends chunks of the matrices to each process. $MPI\_Scatter()$ has the limitation that the total number of elements in the matrix has to be divisible by the number of processes. A more general solution is to use $MPI\_Scatterv()$ instead, in which the data dispatched from the root process can vary in the number of elements. When received, each process sums element by element the two chunks and then all elements from each process are gathered to the root process via the $MPI\_Gather()$ call. The elements are ordered by the rank of the process from which they were received. The runtime is again measured via the $MPI\_Wtime()$ call.

In the matrix-matrix addition each process has to perform, the elements involved are only the ones present in the chunks received by the root. This, plus the fact that the matrices are implemented as 1D arrays, makes the use of a Cartesian topology not useful in this case. Also, $MPI\_Scatter()$ requires that the elements that need to be sent are contiguous in memory, therefore the 1D implementation is the most suited. To actually exploit the virtual topology one solution would be to use MPI derived datatypes to create slices or sub-cubes of the original matrices, contiguous in memory, to send with $MPI\_Scatter()$. The use of single message calls (like $MPI\_Send()$) instead of collective operations is another solution to scatter different chunks to the different processes.

2.2 Results

To prove what we discussed in the previous section, in Table 2 the mean runtimes are presented in the case of 1D, 2D, 3D Cartesian topologies and without the use of a virtual topology. The arrangement of processes in the grid is left to the $MPI\_Dims\_Create()$ call. This routine, when the number of dimensions is specified, decomposes the processes over the Cartesian grid, attempting to balance the distribution by minimising the difference in the number of processes assigned to each dimension. The means are calculated on 100 runs of the program for each setup. In the GitHub repository csv files with timings for each run, computed means and standard deviations are present. As shown, the runtime is almost exactly the same in all cases, meaning that the virtual topology is not exploited at all in this case. Actually, the No topology case presents a decrease in the runtime of few milliseconds, that is probably the time needed in the other cases to setup the virtual topology.

Another performed experiment is to study the runtime of the program in parallel compared with the one of a completely serial program, in which a single process generates and then sums the two matrices. This can answer the

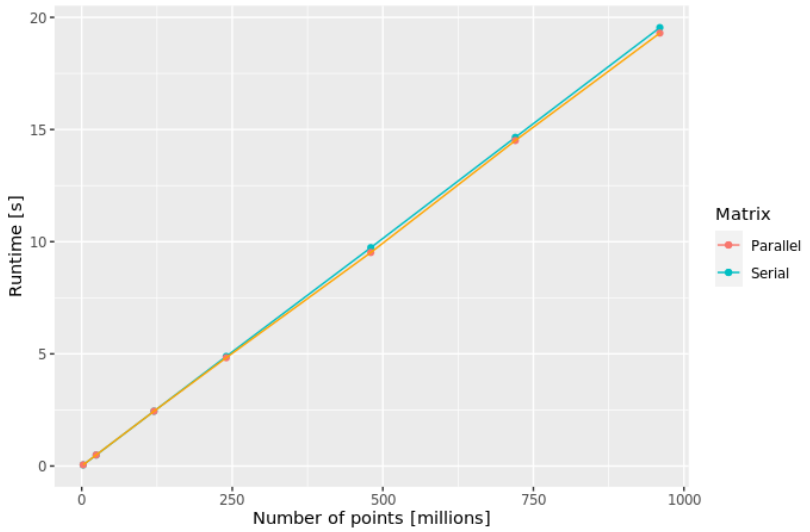

Figure 6: Mean runtime for the serial and parallel matrix - matrix addition implementation for different sizes of the matrices

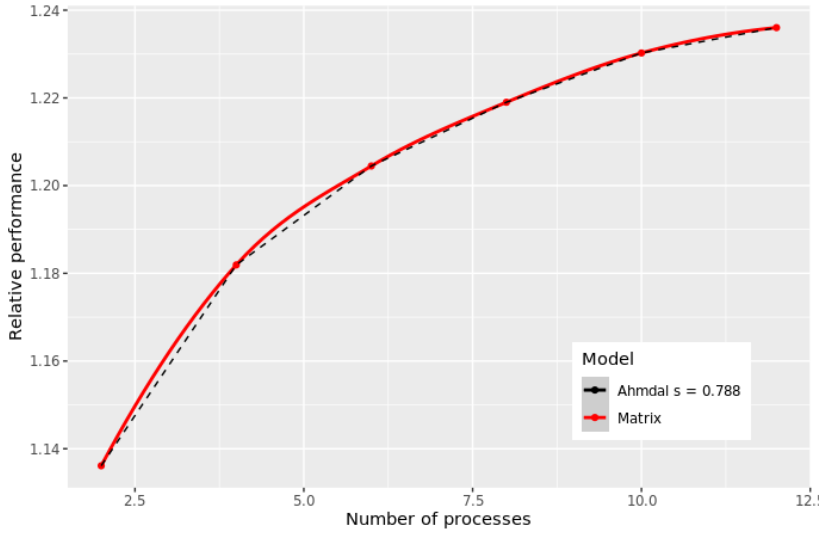

Figure 7: Performance of the matrix code versus the number of processors (strong scaling).

question: is it better to have less communication time but less computation power, or more communication time but more processes involved in the computation. To study the behaviour different sizes of the matrices has been adopted, from 2.4 million to 960 million elements. Then the timings for 100 runs have been taken both for the serial case and the parallel case (24 processes with no virtual topology). The mean runtimes in both parallel and serial cases are plotted in Figure 6 against the number of elements in the matrices. The timings are really similar, first of all because most of the program is actually serial with no parallelization, and second it probably means that the gain of computational power in the parallel case is suppressed by the communication cost of sending the sub-matrices to different processes. As the number of elements increases, we see a slight increase in performance for the parallel case, but still really small. This is also due to the fact that the computations involved are simple element-by-element additions that don't justify the parallel approach. Better performances for the parallel case with respect to the serial one may be achieved for more complex computations.

One last result we present is the study of the strong scaling of the matrix code. In Figure 7 the performance is plotted against the number of processes from 1 to 12. The choice of stopping at 12 processes is given by the fact that if we add another process the communication passes from intrasocket to intersocket, causing another jump in the timings. The serial percentage of the program is then estimated from the Ahmdal's law:

$$S = \frac{1}{s + \frac{1-s}{N}} \tag{3}$$

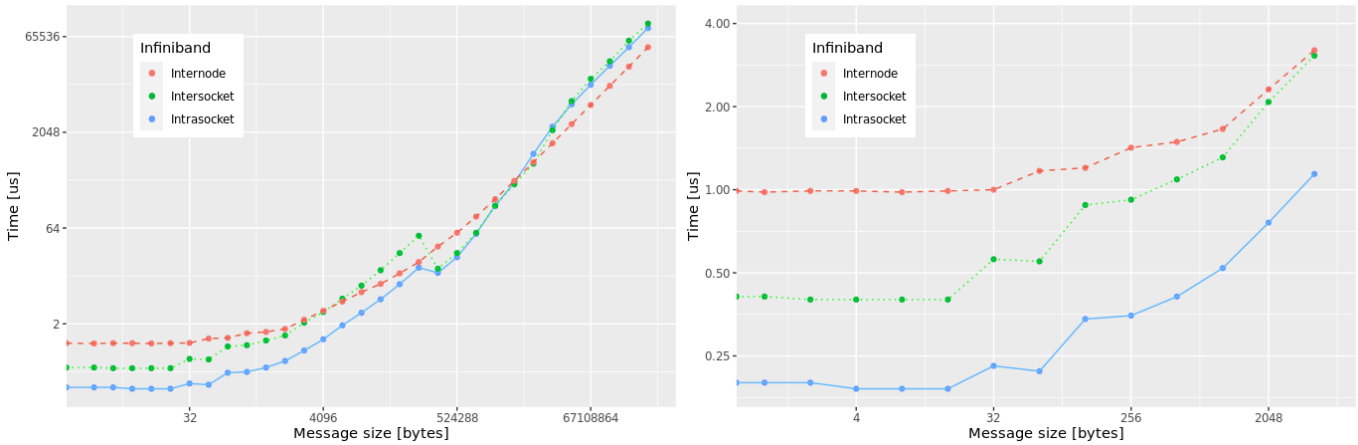

Figure 8: Measured time against message size for the Ping Pong benchmark for intrasocket, intersocket and intranode communication on Infiniband (right). Zoom on the latency zone of the plot (left).

	latency $[\mu s]$
Intrasocket	0.19
Intersocket	0.40
Internode	0.99

Table 3: Comparison between intrasocket, intersocket and internode latency on Infiniband network.

with the scalability S given by a function of the serial part of the program s and the number of processes N. The computed s ($= 0.78$) gives an idea of which percentage of the code is actually serial, meaning that almost 80% of the program is not parallel and therefore it cannot scale well. This is a reasonable result since the whole part of matrix building is serial, and the parallel summation of the elements is just a small part of the overall program.

3 MPI Point to Point Performance

Aim of the exercise is to use the Ping Pong benchmark provided by the Intel MPI benchmarks to estimate latency and bandwidth of all available combinations of topologies and networks on ORFEO computational nodes. Results are obtained using both IntelMPI and Openmpi libraries.

3.1 Methodology

The Ping Pong code sends a message of size N [bytes] once back and forth between two processes running on different processors. The code reports latency (in microseconds), bandwidth (in million bytes per second), and message rate (in messages per second).

To test network performances on ORFEO with different MPI point to point component frameworks, two different PMLs (Point to point messaging layers) have been used: **Ucx** library for Infiniband (the default option) and **Ob1** with the further specification of the BTL (Byte transfer layer), between TCP and Vader, with process-loopback communications ("*self*").

3.2 Results

3.2.1 Time model

The analysis that follows will consider only measured Ping Pong data on the Infiniband network with internode communication using the Openmpi library. However, the same technique can be used for any type of network and protocols.

In Figure 8a the measured time is plotted against the message size for the different cases in which message transfer occurs between processes on the same socket (intrasocket), different sockets on the same node (intersocket) and processes on different nodes (internode). Both axis have been scaled logarithmically for a more readable result. For small message sizes, latency dominates the transfer time and we observe a plateau in the plot (Figure 8b). Considering the simple network model:

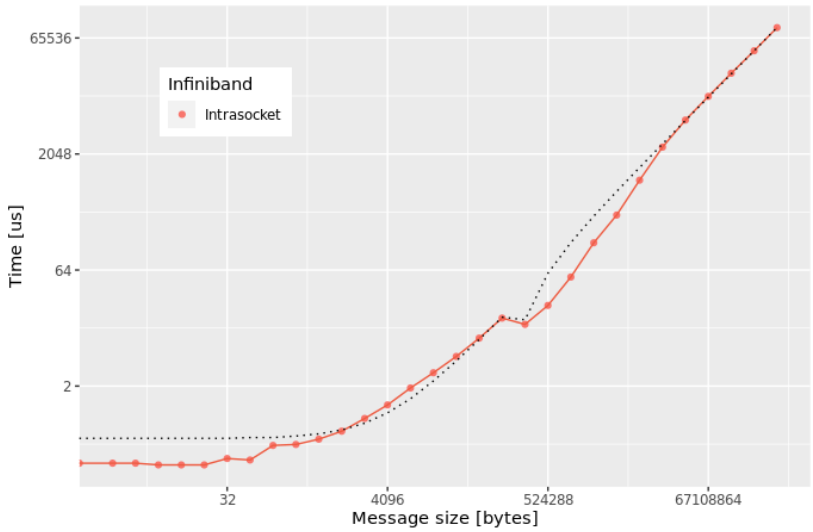

Figure 9: Two-parts fitted time model for intrasocket communication on Infiniband.

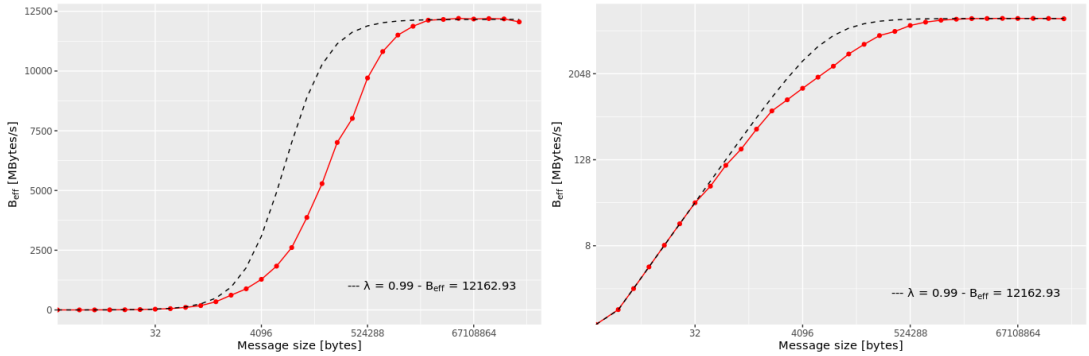

Figure 10: Fits of the Ping Pong benchmark for effective bandwidth to data measured on InfiniBand network.

$$T_{comm} = \lambda + \frac{size}{B_a} \tag{4}$$

with T_{comm} the total transfer time of a message, λ the latency, $size$ the message size and B_a the asymptotic bandwidth. For small message sizes the second term is negligible and the communication time is equal to the latency, that can be estimated from the measured times. In Table 3 a comparison of the results is reported. As expected, latency increases if the cores are on different sockets and even more for cores on different nodes.

Using the model described by equation 4 we can perform a least square fit to obtained estimates of the timings. The results we obtain with a simple linear model, even tho they are able to correctly fit the observed data for large message sizes, are not able to correctly reproduce the latency for small message sizes. The reason for this is that an MPI message transfer is more complex than what the simple model can represent. In fact, most MPI implementations switch between different protocols depending on the message size and other factors, like the use of an eager or rendezvous protocol.

To obtain a better fit one solution could be to split the fitting data in two parts and have two different fits, in this way the results obtained are better than before, but still fail at fitting the latency. In Figure 9 the two-parts linear model is plotted in the case of intrasocket communication. On the GitHub repository the "Estimated Time" column in the csv files represents the time estimated with this model for each case.

3.2.2 Latency - Bandwidth model

As before, we consider only measured Ping Pong data on the Infiniband network with internode communication using the Openmpi library. However, this is enough to describe the gross features well in term of latency and bandwidth and it is applicable to any type of network and protocols. Anyhow a comparison will be presented later.

In Figure 10 the measured bandwidth is plotted against message size. In Figure 10a the x-axis is scaled logarithmically while in Figure 10b both axis are scaled to have more readable results. For very low message sizes a very small

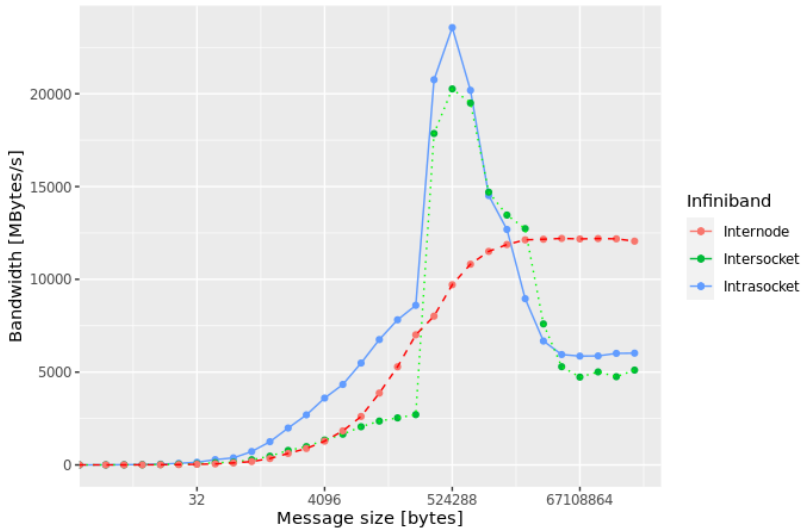

Figure 11: Comparison of intrasocket, intranode and internode effective bandwidth against the message size using Infiniband network on ORFEO Thin nodes.

bandwidth is observed, due to the fact that latency dominates the transfer time. At the end, for large message sizes, latency plays no role and bandwidth saturates to the maximum asymptotic value. Latency can then be estimated by the observed time values for low message sizes ($\lambda = 0.99\ \mu s$) while the asymptotic bandwidth from the effective one at large message sizes ($B_a = 12162.93\ Mbytes/s$). This result is really good since it almost reach the theorical limit of the Infiniband network (12500 $Gbytes/s$).

The parametrized model representing the effective bandwidth (dotted black line) is given by the equation:

$$B_{eff} = \frac{N}{\lambda + \frac{N}{B}} \tag{5}$$

with N the size of the message, λ the estimated latency and B the estimated asymptotic bandwidth. While the model yields good estimates for latency and bandwidth, it produces bad results for intermediate message sizes. An explanation may be that messages have to be split into smaller chunks if they exceed a certain limit, or the fact that message-passing layers and network protocol layers may switch between different buffering algorithms at certain message sizes.

3.2.3 Intrasocket - Intersocket - Internode comparison

We can now compare the result obtained whether message transfer occurs inside an L3 group (intranode intrasocket), between cores on different sockets (intranode intersocket), or between different nodes (internode). This comparison is shown in Figure 11. The same analysis done for the internode case in terms of latency and asymptotic bandwidth can be done for the other two, and both can be easily estimated looking at timings for low message sizes and bandwidth for large message sizes respectively.

As a reminder, these results are drawn using ORFEO Thin nodes. Each node comprises two sockets with twelve cores each with 19 MB L3 cache shared among cores in the same socket. Nodes are connected one another via an HDR Infiniband switch. Given the topology three behaviours are present, depending on whether the communication is internode, intranode or intrasocket. As expected, the intranode behaviour is really different from the internode one for small and intermediate-length messages. The "hump" present in the intrasocket case is explained by the fact that cores on the same socket really benefit from the shared L3 cache achieving a peak bandwidth of over 23 $Gbytes/s$. However this behaviour is present also in the intersocket case, where there is no shared cache and all data must be exchanged via main memory. To explain it, one has to keep in mind that the message transfer is repeated a number of times to get more accurate timing measurements even for small messages. So what happens is that the transfer of the message from sender to receiver can be done as a single-copy operation, i.e. a simple block copy that is sufficient to copy the message from send to receive buffer. If the message size is small enough it is copied in the cache of the receiving process. Since the message is not modified in the iteration loop, after the first iteration the send buffer is in the cache of the receiving process that can exploit in-cache copy operations instead of transferring the data in the subsequent iterations. The last thing to explain is the final drop in performance present in the intrasocket and intersocket case. This is due to the fact that from a certain point on, the L3 cache (19 MB) is too small to hold both or at least one of the local receive buffer and the remote send buffer. In addition, as shown in the csv files, the

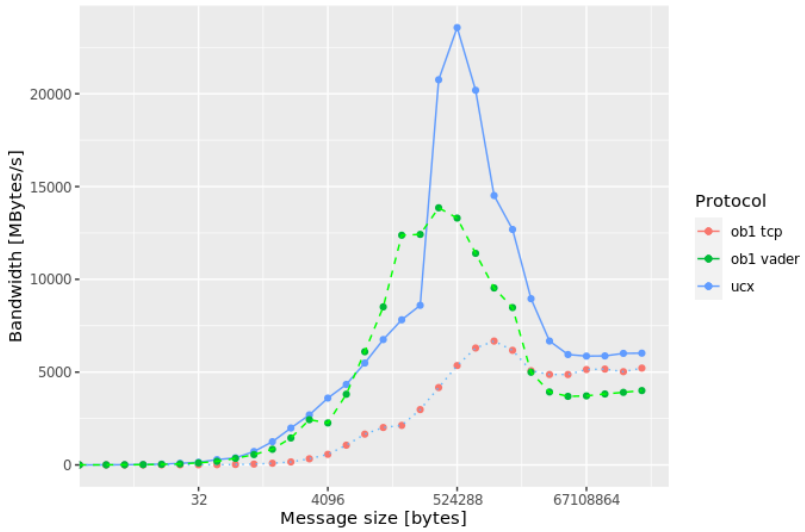

Figure 12: Comparison of effective bandwidth against the message size using Ucx, Ob1 TCP and Ob1 Vader protocols on ORFEO Thin node for cores on the same socket.

	UCX	**Ob1 - TCP**	**Ob1 - Vader**
Intrasocket	0.19	5.61	0.28
Intersocket	0.40	8.15	0.59
Internode	0.99	16.70	\

Table 4: Comparison between intrasocket, intersocket and internode latency in the UCX, Ob1-TCP and Ob1-Vader frameworks.

benchmark is implemented so that the number of iterations decreases with the message size, until the message is sent just one time, that is the initial copy through the network and the in-cache copy cannot be exploited.

3.2.4 Protocols comparison

As already mentioned before, in addition to testing network performances for different mappings of the binned processes, different frameworks for point to point message passing have been used. The UCX, Ob1-TCP and Ob1-Vader protocols has been used for each mapping (intrasocket, intersocket, internode), and results in the form of csv files are present on the GitHub repository. In Figure 12 we show the comparison between UCX, Ob1-TCP and Ob1-Vader for processes on the same socket using the OpenMPI library. As expected, UCX is the fastest one, while the Ob1-TCP case the slowest with a peak bandwidth of around 6.6 $Gbytes/s$, given the limits of the Ethernet network and the large amount of software layers that add a consistent latency in communication. An increase in performance is present for the Ob1-Vader framework, able of exploiting the shared memory between the processes, with a peak bandwidth of almost 14 $Gbytes/s$.

In Table 4 latency for the three frameworks is compared, confirming what has been previously said, Vader is almost as performant as UCX but still has a bigger latency. Also, given the fact that Vader exploits shared memory between processes, it cannot be used for internode communication, that is why latency is missing in the table. Plots showing the comparison in the case of intersocket and internode communication are present on the GitHub repository.

3.2.5 Intel MPI

Next is the comparison in the performances obtained with the OpenMPI and Intel MPI library. The latter has been developed specifically for Intel processors and should achieve best latency, bandwidth and scalability through automatic tuning and optimization. As before, Intel MPI let the user specify which message passing frameworks to use, so the benchmarks has been run with the UCX, TCP and SHM BTLs (Shared memory transport, corresponding to Vader for OpenMPI). Results and plots are present in the GitHub repository for all the cases. In Figure 13a we show the effective bandwidth against the message size for intrasocket, intersocket and internode communication on Infiniband. The behaviour is similar to the OpenMPI results, with the exception of the intersocket case in which the single-copy behaviour is not exploited and since in reality it is not something usually possible, is a more realistic measure of a real application. On Figure 13b we see another interesting result, the effective bandwidth for different protocols in the

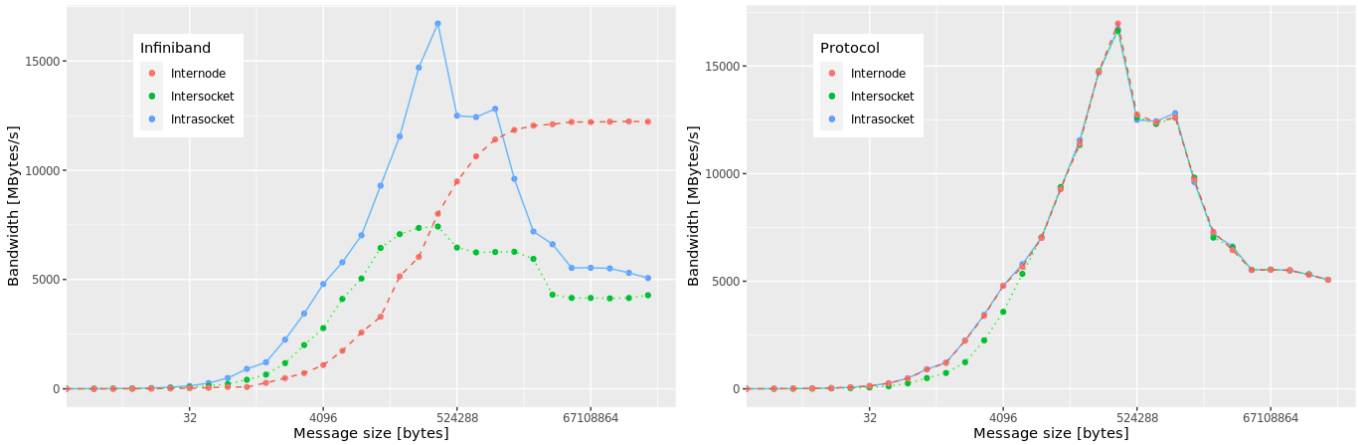

Figure 13: Bandwidth over message size for intrasocket, intersocket and internode communication using UCX (left) and for intrasocket communication using UCX, TCP and SHM frameworks (right).

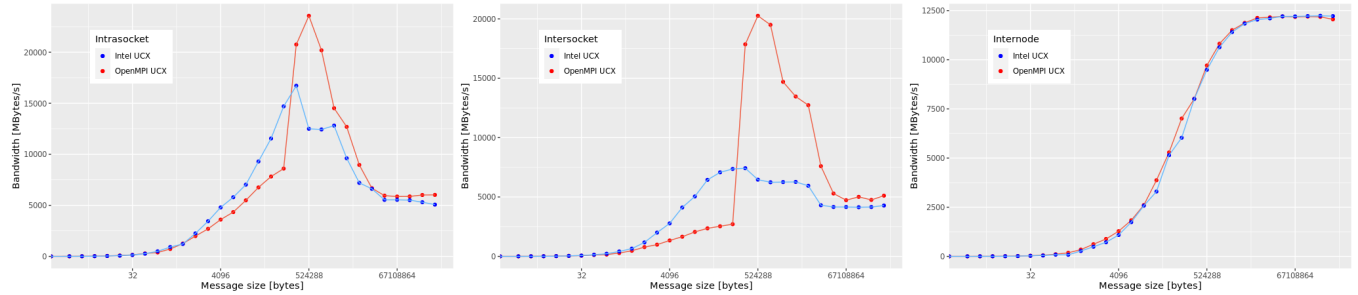

Figure 14: Comparison between OpenMPI and Intel MPI for intrasocket (left), intersocket (center) and internode (right) communication using the UCX protocol.

intrasocket case is around the same in all three cases. An explanation could be that Intel MPI optimizes the protocols in the same way.

Last, in Figure 14 we show the comparison between the OpenMPI and Intel MPI results for intrasocket communication using UCX for Infiniband. The main difference noticed is that even if Intel MPI achieves worse results for medium-size messages, its main focus is probably to get the best optimization for real life applications, since it achieves better results for low-medium size messages, where the grand majority of real programs work. Plus, it doesn't exploit the single-copy behaviour (hump) in the intersocket case, scenario impossible to happen for real life application.

3.3 Thin vs GPU node comparison

At last, we show the comparison between the results obtained running the Ping Pong benchmark on a Thin node and a GPU node on ORFEO. To quickly recap, a Thin node comprises an Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz, with 2 sockets of 12 sockets each and 19 MB L3 cache. Instead a GPU node has an Intel(R) Xeon(R) Gold 6226 CPU @ 2.70GHz, again with 2 sockets, 12 cores each but with hyperthreading enable, so 24 physical and 24 virtual

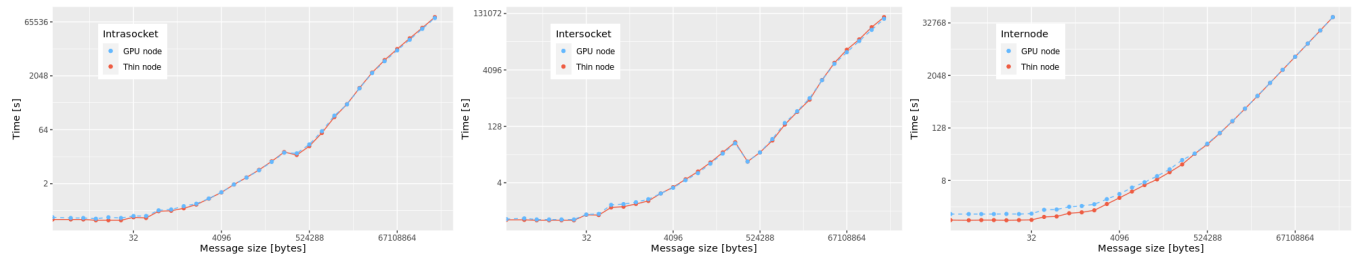

Figure 15

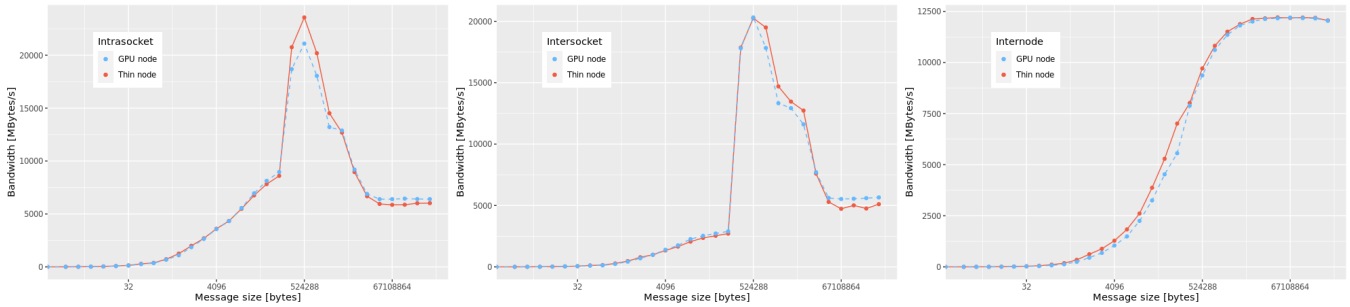

Figure 16: Comparison between time performances (first row) and bandwidth (second row) for Thin and GPU nodes for intrasocket (left), intersocket (center) and internode (right) communication using the UCX protocol on Infiniband.

cores and 19 MB L3 cache. As shown in Figure 15 and 16, results obtained in for timings and bandwidth are plotted against the number of processes. We can notice that the results are really similar in the two cases, probably due to the fact that both nodes rely and can exploit the fast Infiniband network. What we can notice is an improvement in the latency in the case of the GPU node.

4 Jacobi solver

Aim of the exercise is to compile and run the code on different numbers of cores for both Thin and GPU nodes on ORFEO. Then, to compare measured performance with the theorical performance model. The program exploits the Jacobi method to determine the solutions of a system of linear equations.

4.1 Methodology

The code has been run on one single process to have a measurement of the serial execution time. Then it has been run using $4, 8$ and 12 MPI processes pinned within the same socket and across two sockets. Then with $12, 24$ and 48 processes on two different nodes. In the intersocket case, processes are bounded to the different sockets alternating (process 0 to socket 0 - core 0, process 1 to socket 1 - core 0). In the internode case, even processes are bounded to the sockets of the first node and odd processes to the other node, as in the intersocket case for each node.

The Jacobi solver is a good example of domain decomposition in which virtual topology can really be exploited. In fact, in order to yield the correct results for each subdomain halo layers need to be used to store copies of the boundary information from neighboring domains. Used topologies for the different numbers of processes are listed below:

- 4 processes: (2 x 2 x 1)
- 8 processes: (2 x 2 x 2)
- 12 processes: (3 x 2 x 2)
- 24 processes: (4 x 3 x 2)
- 48 processes: (4 x 4 x 3)

Since the problem grid is cubic, the optimal configuration for communication overhead is with all the dimension extensions as close together as possible.

4.2 Results

We can now study the results in term of weak and strong scaling. All shown results are done using Thin nodes on ORFEO.

4.2.1 Weak scaling

Since subdomains synchronize before each new iteration, the overall execution time of the program is dictated by the slowest process that is the one with the largest number of halo layers, i.e. the one that has to exchange more data with the other processes. Another important thing to notice is that while the Ping Pong benchmark exploits just

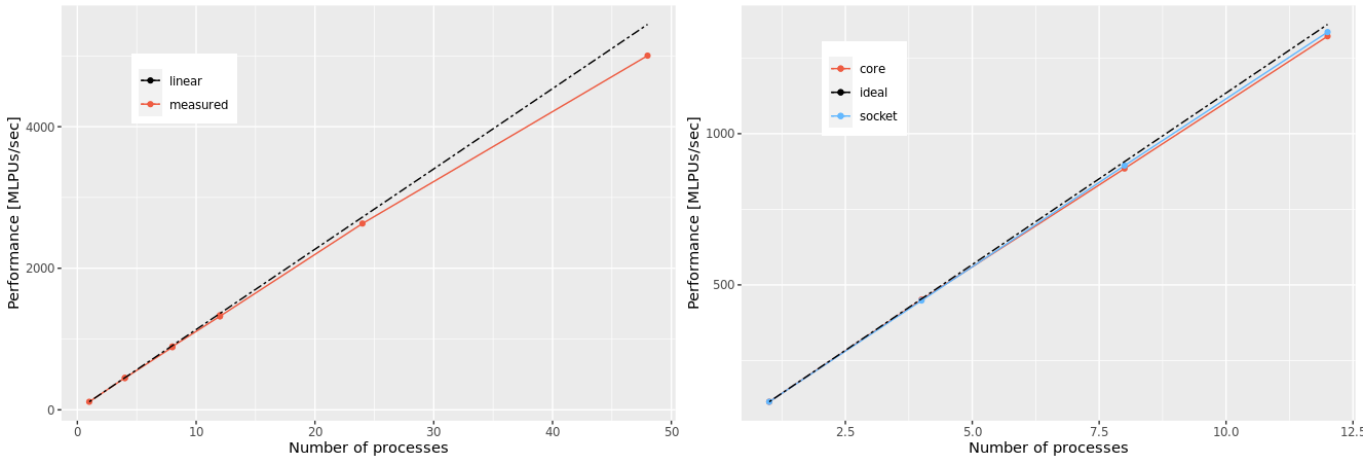

Figure 17: Weak scaling of the MPI Jacobi solver with problem size 1200^3 on InfiniBand (right). Zoom for a comparison of performances in the intrasocket and intersocket cases (left).

half-duplex data transfer over a single link (since one message is sent at a time), here sending and receiving halo layers can overlap for the different processes in each direction and so we talk about full-duplex data transfer.

In Figure 17 the weak scaling of the measured performance of the Jacobi solver is plotted and it is compared with the ideal case. We can notice that given how fast the Infiniband network is, scalability is almost perfect. Performance gets worse whenever a new direction gets cut but the effect is almost invisible given the really low latency and high bandwidth.

The theoretical performance model is given by the following equation:

$$P(L, \vec{N}) = \frac{L^3 N}{T_s(L) + T_c(L, \vec{N})} \tag{6}$$

with L the subdomain size (constant regardless the number of processes), N the number of processes, T_s the raw compute time for all cell updates in a sweep (also constant). The performance $P(L, \vec{N})$ is measured in Million Lattice Update Per Second (MLUPs/sec). T_c is the communication time and depends on the number and size of domain cuts through the formula:

$$T_c(L, \vec{N}) = \frac{c(L, \vec{N})}{B} + k\lambda \tag{7}$$

where B is the full-duplex bandwidth, λ the latency and k the largest number of coordinate directions in which the number of processes is greater than one. $c(L, \vec{N})$ is the maximum bidirectional data volume transferred over a node's network link and is given by:

$$c(L, \vec{N}) = L^2 * k * 2 * 8 \tag{8}$$

Thanks to equation 7 we are able to produce an estimate of the full-duplex bandwidth exploited. Results can be found in each csv file under the "Est_B" column in the GitHub repository. Latencies used in the equations are the ones reported in the first column of Table 4. In Table 5 we report the results obtained with the model for $L = 1200$. $P_1(L)$ is the measured performance of the execution on the single processor. The ratio in the last column is the slow-down factor compared to the ideal case. If we compare the estimated performance $P(L, \vec{N})$ we can conclude that the model is able to correctly reproduce the observed values.

4.2.2 Strong scaling

In Figure 18 we plot the performance against the number of processors varying the size of the input grid between 120, 480, 960 and 1200 elements per dimension. We can see that the penalty with respect to the ideal case increases when the problem size gets smaller, due to the communication cost of the halo layers, that also increases with the number of processes. The gap between the cases is instead mainly due to the different subdomain sizes.

	N	$N_x N_y N_z$	$c(L,\vec{N})$ **[MB]**	$P(L,\vec{N})$ **[MLUPs/sec]**	$\frac{NP_1(N)}{P(L,\vec{N})}$
serial	1	(1,1,1)	0.00	113.45	1.0000
core	4	(2,2,1)	92.16	451.86	1.0056
	8	(2,2,2)	184.32	904.03	1.0258
	12	(3,2,2)	276.48	1357.54	1.0296
socket	4	(2,2,1)	92.16	451.47	1.0123
	8	(2,2,2)	184.32	904.66	1.0145
	12	(3,2,2)	276.48	1358.11	1.0190
node	12	(3,2,2)	92.16	1357.17	1.0299
	24	(4,3,2)	552.96	2718.02	1.0345
	48	(4,4,3)	1105.92	5436.16	1.0883

Table 5: Model prediction for the Jacobi solver using Infiniband.

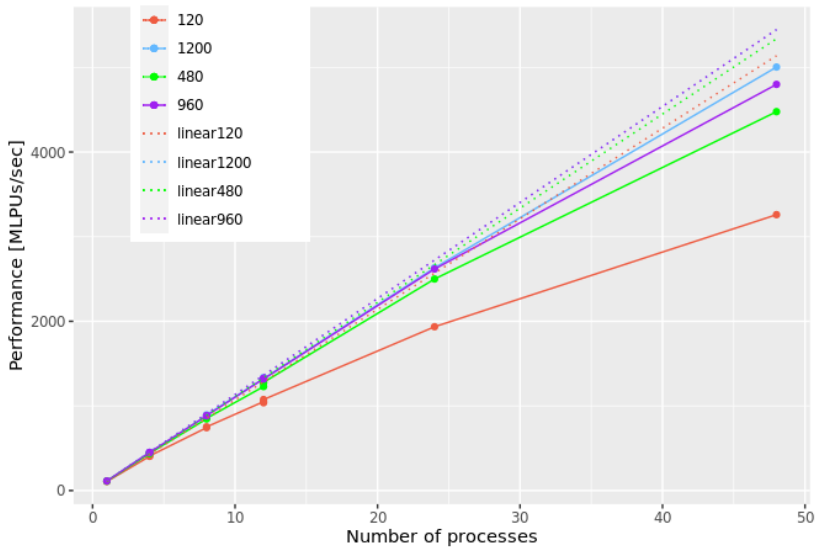

Figure 18: Strong scaling of the MPI Jacobi solver with problem size 120^3, 480^3, 960^3 and 1200^3 on InfiniBand.

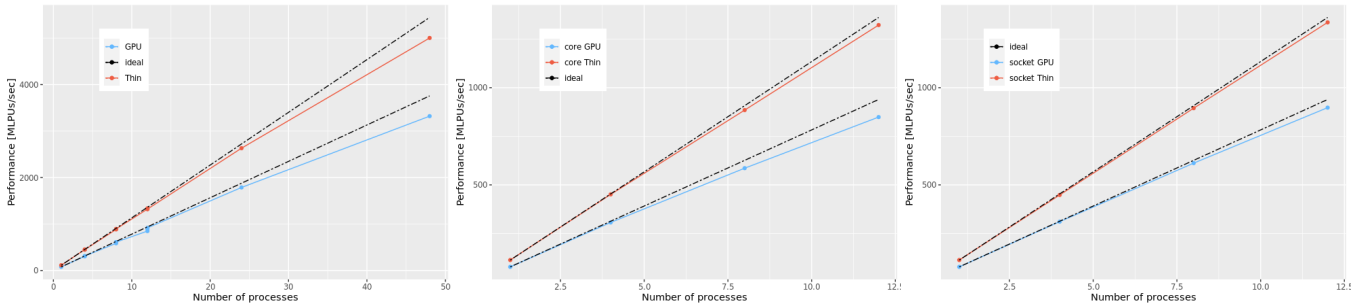

Figure 19: Comparison of the scalability of the Jacobi solver on Thin and GPU nodes using Infiniband.

4.3 Thin vs GPU node comparison

To conclude, we show the comparison between the results for the Jacobi solver obtained on Thin and GPU nodes on ORFEO. All results are obtained for a problem grid of 1200^3 elements. In Figure 19, the performance of the model on the two types of nodes is plotted against the number of processes. In Figure 19b and 19c a zoom of the plot is done to show the comparison in the intrasocket and intersocket case. As the figure shows, the Thin nodes behaves much better in term of scalability the GPU ones. The reasons for this behaviour is probably due to the hyperthreading enable in the GPU nodes.