# Foundations of high performance computing

Assignment 1

**Matteo Marturini**

**MAT. SM3500484**

**Data Science and Scientific Computing**

**University of Trieste**

December 2021

## Contents

# 1 MPI programming

In this section, 2 MPI problems are reviewed: the ring problem and the 3D matrix sum problem.

## 1.1 Ring problem

The program consists in building a 1D ring and making each process send a right and left message with a tag proportional to its rank in such a way that each receiving process will add or subtract its rank value depending whether the received message comes from the right or the left; then each process will resend the messages along the ring, and this procedure is iterated until the original message has come back to the original sender. At the end, each process should have 2 values, the sum and negative sum of the rank values of the processes composing the ring.

The problem is solved by creating a 1-dimensional topology through the use of the `MPI_Cart_create()` routine, and inside the for loop for sending and receiving messages, `MPI_Cart_shift()` is called in order to ensure that the tag of the original messages is proportional to the rank of the process whenever they will arrive back at the original sending process itself (for simplicity, the factor of proportionality is set to 1; solution in file `ring.c`).

By analyzing the algorithm, assuming that the time for sending and receiving messages, and the time to perform the sums are constant, since the for loop is repeated $P$ times (where $P$ is the number of processors), we should expect the complexity to belong to $\Theta(P)$, i.e. the execution time should increase linearly with the number of processors being used.

To study the complexity of the algorithm, the execution time is measured as the number of processes composing the ring varies; the results are reported in the following table.

| P | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Time [s] | 0.003 | 0.004 | 0.008 | 0.007 | 0.009 | 0.009 | 0.007 | 0.013 | 0.0013 | 0.025 | 0.018 | 0.026 |

Since the order of magnitude is too small to perform any significant analysis, the portion of code which perform the computation and the sending of the messages is repeated $10^5$ times. Furthermore, this process is repeated for 10 times, and the averages of the execution times are taken. The results are reported in figure 1.

It can be observed that both for the gpu and thin node there is a noticeable growth in execution time from $P = 12$ to $P = 14$; this is probably due to the fact that since the program had been run by mapping processes to cores, before 12, all processes are mapped inside 1 socket, but when we move from 12 to 14, the 2 additional processes are mapped to the other socket of the node, and hence the communication time to and from these processes may be slightly greater, increasing the total execution time.

Given the considerations about the complexity of the algorithm, 2 linear models are fit to the data via least square optimization, one model for processes up to 12, and one for processes from 12 to 24. The result is reported in figure 2. To test the goodness of fit, the $R^2$ coefficient is reported.

| | pre 12 | post 12 |
|---|---|---|
| $R^2$ | 0.994 | 0.962 |

In both cases, almost all the variability of data is explained by the model.

An alternative could be that of employing a quadratic model, considering the totality of the processors (figure 2). Even though the $R^2$ coefficient is still high ($R^2 = 0.982$), there is no clear interpretation for explaining this type of model.

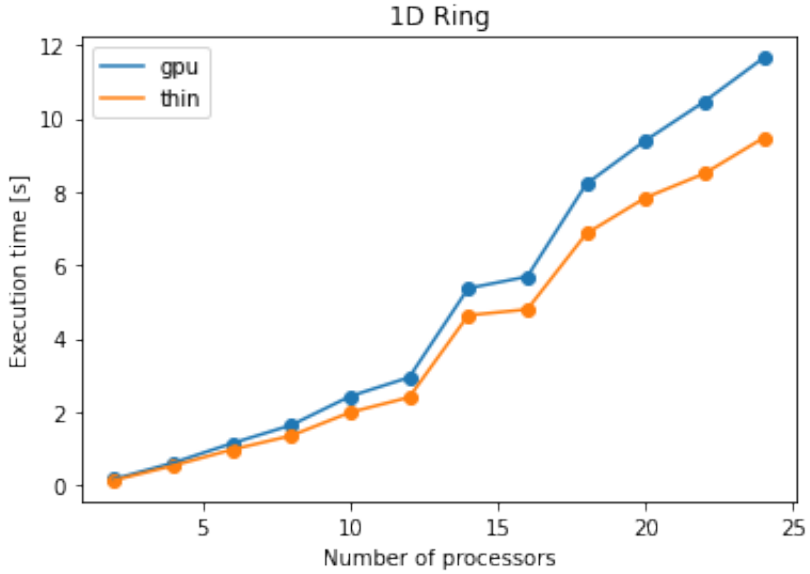


Figure 1: Execution time of the 1D ring problem as a function of the number of processes composing the topology, for a *thin* and *gpu* node.

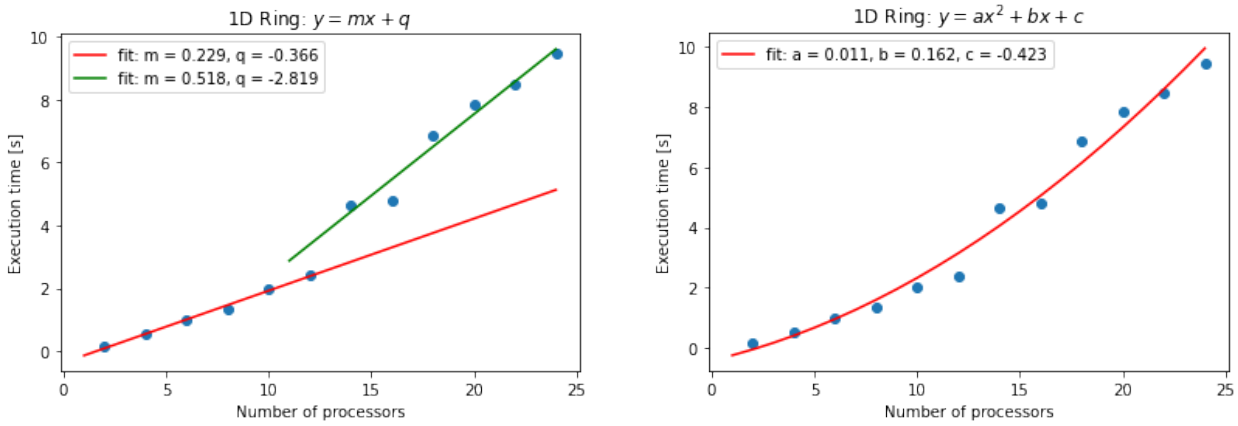


Figure 2: Quadratic and linear model for the execution time of the ring problem.

| (npx, npy) | 2400x100x100 | 1200x200x100 | 800x300x100 |
|---|---|---|---|
| (2,12) | 5.719 | 5.866 | 5.870 |
| (12,2) | 5.733 | 5.824 | 5.864 |
| (4,6) | 5.827 | 5.846 | 5.846 |
| (6,4) | 5.831 | 5.887 | 5.870 |
| (8,3) | 5.861 | 5.858 | 5.860 |
| (3,8) | 5.858 | 5.865 | 5.819 |

Table 1: 2D topology

| (npx, npy, npz) | 2400x100x100 | 1200x200x100 | 800x300x100 |
|---|---|---|---|
| (4,3,2) | 5.737 | 5.808 | 5.847 |
| (4,2,3) | 5.672 | 5.807 | 5.835 |
| (2,3,4) | 5.820 | 5.832 | 5.833 |
| (2,4,3) | 5.824 | 5.871 | 5.860 |
| (3,4,2) | 5.826 | 5.856 | 5.811 |
| (3,2,4) | 5.840 | 5.846 | 5.804 |
| (6,2,2) | 5.837 | 5.813 | 5.860 |
| (2,6,2) | 5.845 | 5.812 | 5.818 |
| (2,2,6) | 5.799 | 5.836 | 5.821 |

Table 2: 3D topology

### 1.2 Matrix problem

The problem consists in writing a program which performs the sum between 2 3D matrices in parallel ($C = A + B$), with different virtual topologies (through the use of the `MPI_Cart_create()` routine), and with only collective operations.

**Trivial solution** The most straightforward solution is to initialize $A$ and $B$ on the root process, then to scatter portions of the 2 matrices to each process(`MPI_Scatter()`), performing the sum between each corresponding block, and gathering the results on root (`MPI_Gather()`); this approach works only if the total number of elements is divisible by the total number of processes, so that the `MPI_Scatter()` routine sends an equal number of elements to each process to perform the sum (the solutions can be found in the file `sum3Dmatrix.c`.

By setting the number of processes to 24, three different domains are analyzed for each of the 3 different topologies (1D, 2D and 3D), and the computational time is recorded (to perform a significant measure, the sum is performed several times). In tables 1 and 2 are reported the results of some possible combinations respectively for 2D and 3D topology (the result for the 1D topology is 5.724); the first column represents the number of processors along each dimension.

The tables do not show any significant difference among the combinations and the topologies, as expected. The choice of a particular topology does not affect the overall performance because by exploiting collective operations, adjacent portions of the array are sent to the various processes, and because there is no communication between them, we are not exploiting the particular shape of the domain of our data.

Fixing the size of the 2 matrices to (2400,100,100), the time as a function of the number of processes is reported in figure 3. After a drop in time from 1 to 2 processes, the time remains constant, meaning that the program does not scale, even though by increasing the number of processes used we should expect the time to decrease because the number of elements to add per process diminishes.

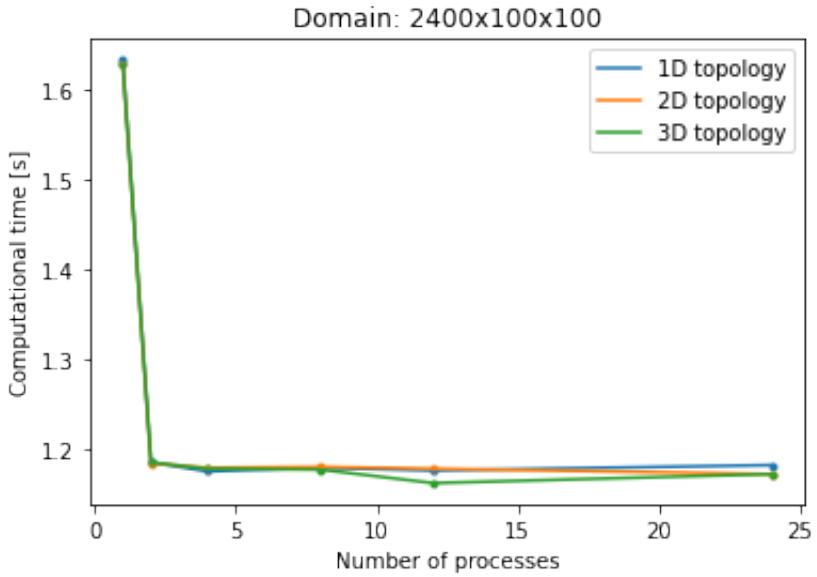


Figure 3: Computational time as a function of the number of processes for the three different topologies, fixing the size of the matrices to 2400x100x100, on a Thin node.

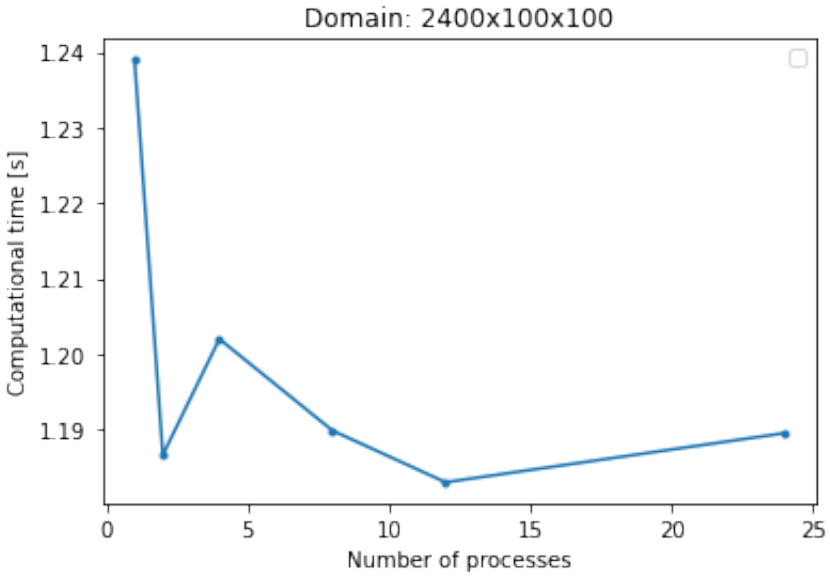


Figure 4: Computational time as a function of the number of processes for the 'not trivial' solution, fixing the size of the matrices to 2400x100x100 on a Thin node.

**Not trivial solution** A not trivial solution requires that to increase performance, the topology should resembles the shape of the domain of the data; however, as previously stated, since there is no communication between the processes, the gain in performance which a decrease in communication time would yield cannot be obtained in this problem. However, a program for the 3D topology which sends blocks of the matrices $A$ and $B$ to processes aligned to the physical topology is implemented in the file `sum3Dmatrix_with_MPI_Datatype.c`.

In figure 4 is reported the computational time vs number of processes; as expected, the result is analogous to the one obtained in figure 3.

## 2 ORFEO PingPong benchmark

In this section, the Intel MPI point-to-point PingPong benchmark (https://www.intel.com/content/www/us/en/develop/documentation/imb-user-guide/top/mpi-1-benchmarks/imb-p2p-benchmarks/pingpong.html) is used to model the performance of the different networks available at the ORFEO HPC cluster.
The PingPong benchmark consists of a message of $N$ bytes sent back and forth between two processes; the output reports the size of the message in number of bytes, the number of repetitions, the communication time in $\mu s$, and the effective bandwidth in $Mbytes/s$.

The analysis is performed on *GPU* and *Thin* nodes. For each type of node, it is modeled the communication between cores inside the same socket, across 2 different sockets, and across two different nodes; it is later analyzed the difference in performance between 2 different implementations of the MPI standard, Open MPI (https://www.open-mpi.org/ and Intel MPI (https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html)

The assumed model for the communication time is

$$T_{comm} = \lambda + \frac{m_{size}}{b_{network}} \tag{1}$$

where $\lambda$ is the latency, $m_{size}$ is the size of the message, and $b_{network}$ is the bandwidth.

### 2.1 THIN

#### 2.1.1 Open MPI

Open MPI uses Modular Component Architecture (MCA) parameters to provide users a way to customize the Open MPI environment to suit the specifics of the underlying hardware; for example, we can affect the byte transport layer for host-to-host communication by changing the `btl` parameter, or we can set the `pml` parameter to modify the more general point-to-point management layer.

**Inside socket** In figure 5 is reported the effective bandwidth in Mbytes/s as a function of the message size for different values of the `btl` parameter: `default` refers to a non specified value for the parameter (by default it will refer back to `UCX` communication library), `tcp` refers to the protocol used at byte transport layer, while `vader` refers to a mechanism for transferring data via shared memory. There is a clear difference among the 3 possible choices, `default` being the most performing, because the default behavior of Open MPI is that of using the most effective mechanism available.

The plot shows a peak at a message size between $10^5$ and $10^7$ bytes; for a plausible explanation of this behavior the caches and the architecture of the *thin* node must be taken into consideration. The single core of a node has a private l1-cache of size 32 kB, while an l2-cache of 1024 kB is shared among the cores of one socket. Given a small message size, the PingPong benchmark repeats the sending of the message several times in order to obtain significant measurements; as long as the message fits inside the l1-cache, after the first miss, the process will find the message to send inside the l1-cache, speeding up the program. When the message won't entirely fit inside the l1-cache, part of it will be stored inside the l2-cache. Analogous reasoning can be applied to the l2-cache: up to a message size of 1 MB, the message will be entirely stored inside the l2-cache, but as the size increases, the process will have to retrieve data from the l3-cache, slowing down the process, until we reach the region inside which the asymptotic bandwidth can be measured.

The right plot of figure 5 reports the fit for the communication time model in equation (1). The model is unable to explain the data obtained, as it predicts a negative latency, because it does not take into consideration the effects of the caches in speeding up the process; hence a more sophisticated model is required.

**Across sockets** Analogous results are obtained when pinning processes across 2 different sockets of the same *thin* node, as outlined in figure 6 .

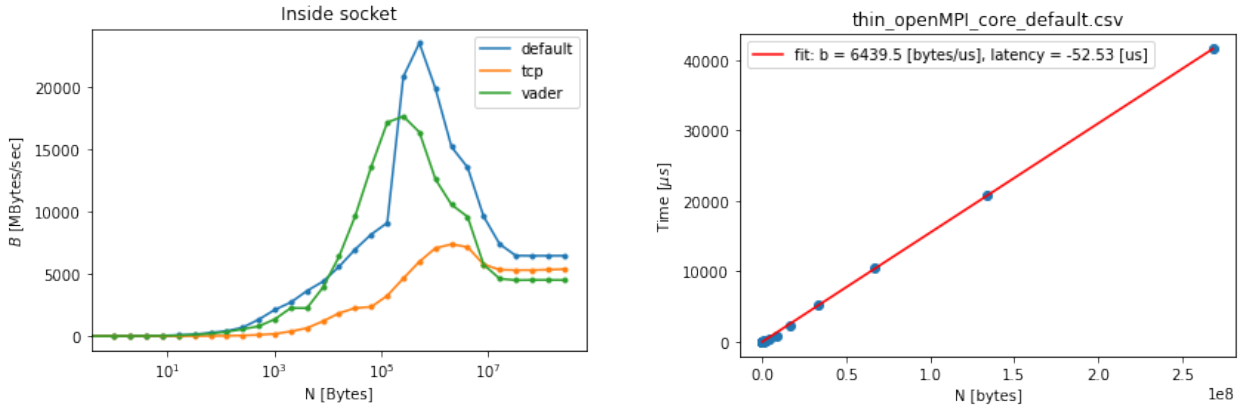


Figure 5: (left) Effective bandwidth as a function of message size when pinning processes inside the same socket of a thin node, for different values of the btl parameter. (right) Fit of the communication time model for the default case.

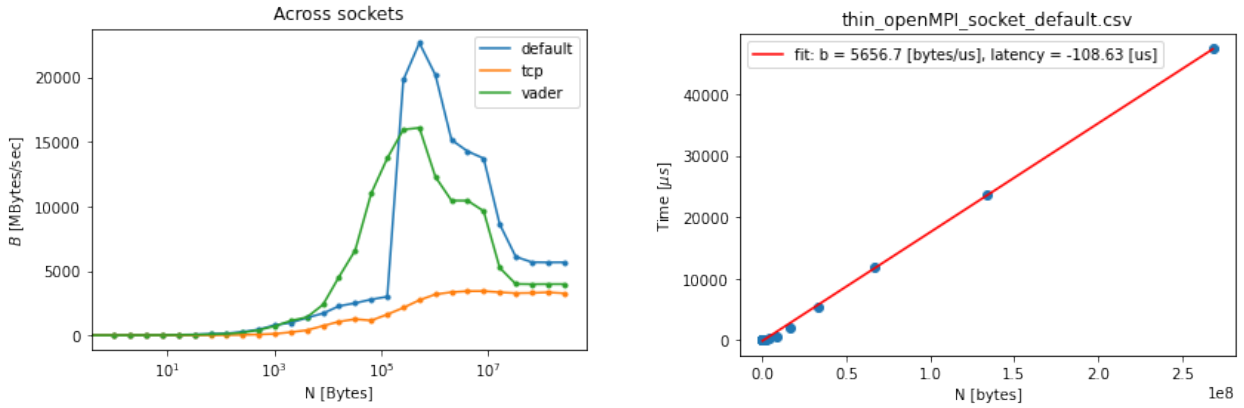


Figure 6: (left) Effective bandwidth as a function of message size when pinning processes on 2 different sockets of the same *thin* node. (right) Fit of the communication time model for the default case.

**Across nodes** In figure 7 are reported the results when pinning the MPI processes to different *thin* nodes. In this case, the simple communication model in equation 1 is able to appropriately describe the data obtained. In particular, it can be observed that without specifying any `btl` parameter, the messages are exchanged through the 100 Gbit HDR Infiniband which connects the computational nodes in the ORFEO cluster, as the latency ($\approx 4\ \mu s$) and the bandwidth ($\approx 12$ Gb/s) are comparable with the theoretical ones (1.35 $\mu$s and 12 Gb/s). In contrast, by specifying `btl = tcp`, the 25Gbit Ethernet network is used to exchange messages, leading to a much lower bandwidth of $\approx 2.7$ Gb/s, and a higher latency of 43 $\mu$s.

### 2.1.2 Intel MPI

The Intel MPI library provides a set of environmental variables through which the user can modify the runtime environment to exploit the underlying hardware, and boost the performance of the application. In particular, the variables taken in consideration are:

- `I_MPI_PIN_PROCESSOR_LIST`
- `I_MPI_FABRICS`
- `I_MPI_OFI_PROVIDER`

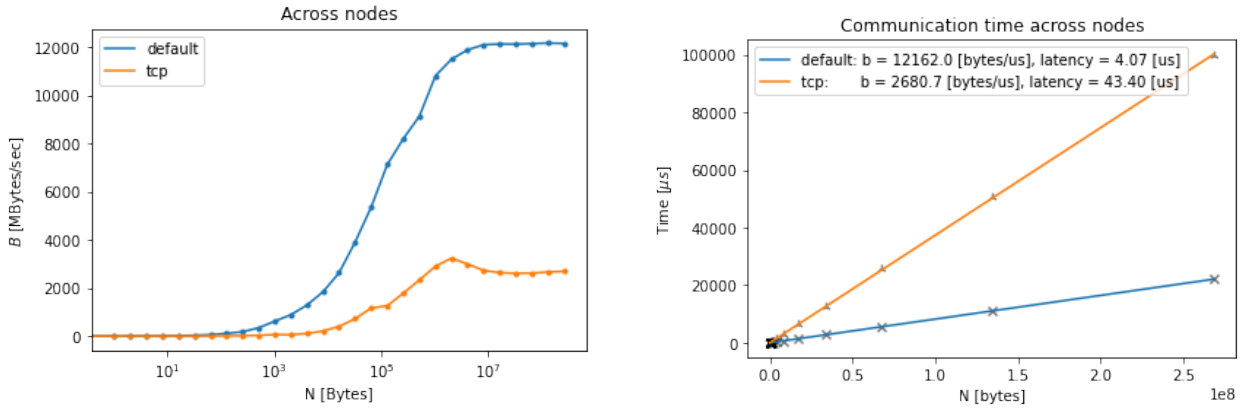


Figure 7: (left) Effective bandwidth as a function of message size pinning the processes across different thin nodes. (right) Fit of the communication time model.

- `I_MPI_DEBUG`

`I_MPI_PIN_PROCESSOR_LIST` allows to specify a list of processors (through the logical identifier) to which the MPI processes are pinned; for example, given a *thin* node on ORFEO, to pin processes inside the same socket, we can specify `I_MPI_PIN_PROCESSOR_LIST=0,2` for contiguous cores or `I_MPI_PIN_PROCESSOR_LIST=0,22` for far away cores. On the other hand, to pin processes to different sockets we can specify `I_MPI_PIN_PROCESSOR_LIST=0,1` (or 23 instead of 1) for close (far) cores.

`I_MPI_FABRICS` is used to select the particular fabrics to be used: `shm` for shared memory transport (for intra-node communication only) and `ofi` to employ OFI (OpenFabrics Interfaces) capable network fabrics.

`I_MPI_OFI_PROVIDER` is used to define the name of the OFI provider to load; if not specified, the OFI library chooses the provider automatically. Among the ones supported, those of interest are: `mlx`, a provider that runs over the UCX that is currently available for the InfiniBand hardware, `tcp`, a general purpose provider for the Intel MPI Library that can be used on any system that supports TCP sockets to implement the libfabric API (lets you run the application over regular Ethernet), and `verbs`. For more detailed information see https://www.intel.com/content/www/us/en/develop/documentation/mpi-developer-guide-linux/top/running-applications/fabrics-control/ofi-providers-support.html.

**Inside and across sockets** The results of the fit are analogous to those obtained in the corresponding section using Open MPI, hence they are not reported here. In figure 8 is reported the effective bandwidth as a function of the message size, pinning processes to both inside the same socket and across different ones. Analogously as to what obtained using Open MPI, a peak appears at a message size of $10^6$ bytes.

Regarding the choice of the environmental variables, the only difference can be seen by setting `I_MPI_FABRICS=ofi`, all other possible choices overlap one to the other on the default behavior, apart from small differences.

In figure 9 the difference when pinning processes to close or far away cores is reported, both for inside and across sockets; in neither case a significant difference is observed.

**Across nodes** In figure 10 is reported the effective bandwidth and the fit for the communication time model. In this case it can be observed that the default case falls back to the `mlx` option, and that there is a clear difference in performance among the various possibilities, being `tcp` the worst, consistent to the data obtained using Open MPI. The verbs provider enables applications using

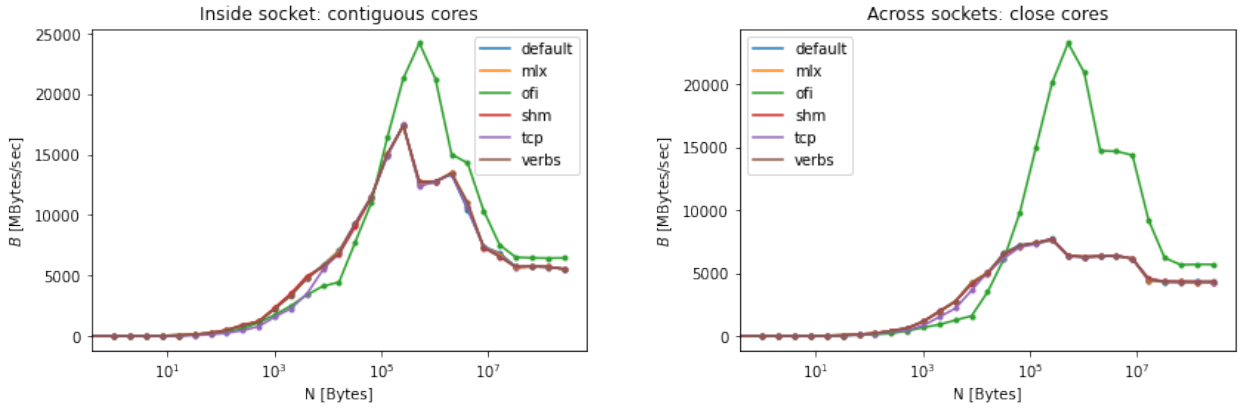


Figure 8: (left) Effective bandwidth as a function of message size when pinning processes to contiguous cores inside the same socket, for different values of the Intel MPI environmental variables. (right) Analogous plot but the processes are pinned to cores that are "close" but on different sockets.

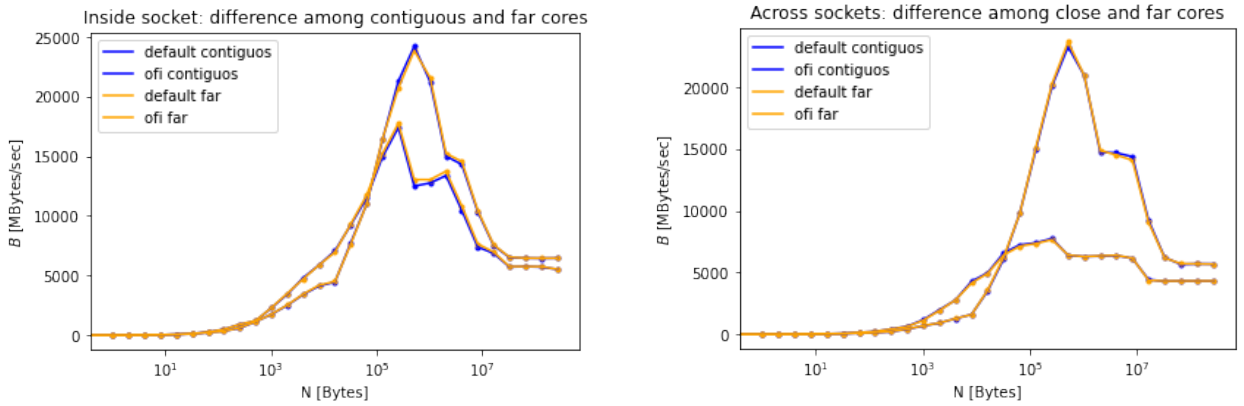


Figure 9: (left) Difference in the effective bandwidth when pinning processes to contiguous or far away cores inside the same socket. (right) Analogous plot but the processes are pinned to cores that are close and far away on 2 different sockets.

OFI to be run over any verbs hardware (InfiniBand, iWarp, and so on). It uses the Linux Verbs API for network transport and provides a translation of OFI calls to appropriate verbs API calls.

The results of the fit for the Intel MPI and Open MPI library for node-to-node communication are summarized in the following tables (Infiniband on the left, Ethernet on the right):

| Infiniband | Latency [$\mu s$] | Bandwidth [bytes/s] |
|---|---|---|
| Intel MPI | 5.74 | 12239.2 |
| Open MPI | 4.07 | 12162 |

| Ethernet | Latency [$\mu s$] | Bandwidth [bytes/s] |
|---|---|---|
| Intel MPI | 14.71 | 2509.8 |
| Open MPI | 43.40 | 2680.7 |

While the results using the Infiniband network are comparable, the latency and bandwidth using the Intel MPI library over the Ethernet network are estimated to be quite lower with respect to the ones estimated using the Open MPI library.

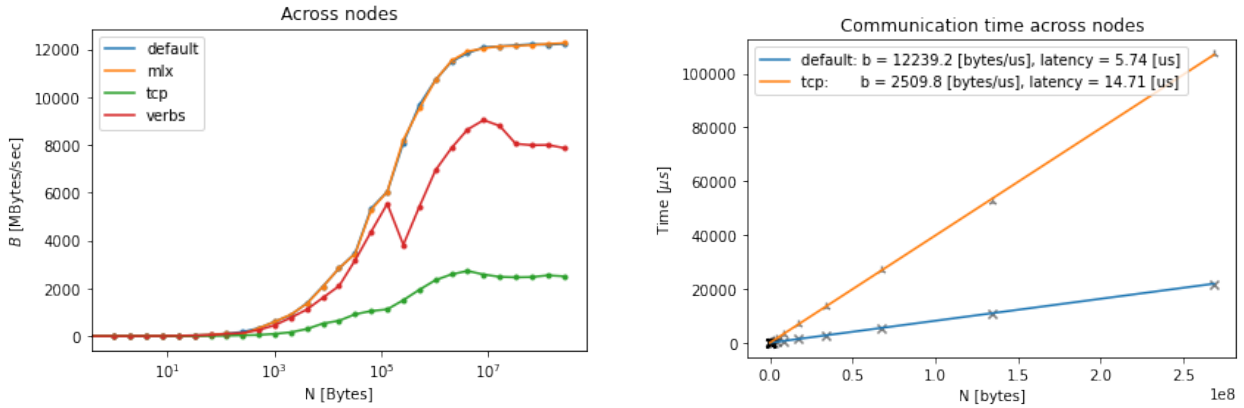


Figure 10: (left) Effective bandwidth when pinning the MPI processes to 2 different thin nodes, for different values of the Intel MPI env. variables. (right) Fit for the communication time model for the default and tcp case.

## 2.2 GPU

Analogous procedure has been followed to analyze the performance of GPU nodes.

### 2.2.1 Open MPI

In figure 11 it is reported the bandwidth as function of message size for the 3 types of communications analyzed (node-to-node, socket-to-socket and inside same socket) for GPU and Thin nodes. The only difference that can be seen between the 2 types of node, is that the peak for the thin node is slightly higher than the one for the GPU node; the peak however appears at the same message size, consistent to the explanation given in section 2.1.1.

### 2.2.2 Intel MPI

Analogous plot is reported in figure 12 using the Intel MPI implementation: the behavior is consistent to the one observed using the Open MPI implementation.

A summary of the results obtained by fitting the model (1) for the node-to-node communication is reported in table 3 (Infiniband), table 4 (Ethernet) and figure 13.

The results obtained for the communication through the Infiniband network using the Intel MPI or Open MPI implementation on thin and GPU nodes are almost identical, due to the fact that in both cases the UCX communication library is used. However, the results for the Ethernet communication are quite different: first of all it can be noticed that the latency on GPU nodes is estimated to be higher with respect to thin nodes, for both the MPI implementations; moreover, given thin nodes, the latency estimated with Intel MPI is lower than the one estimated with Open MPI, while it's the opposite if gpu nodes are considered. This high variability in the results may be due to the fact that the data are clumped at the beginning of the x-axis, so a small change in high values of the x-axis may produce still a valid fit but with quite different parameters.

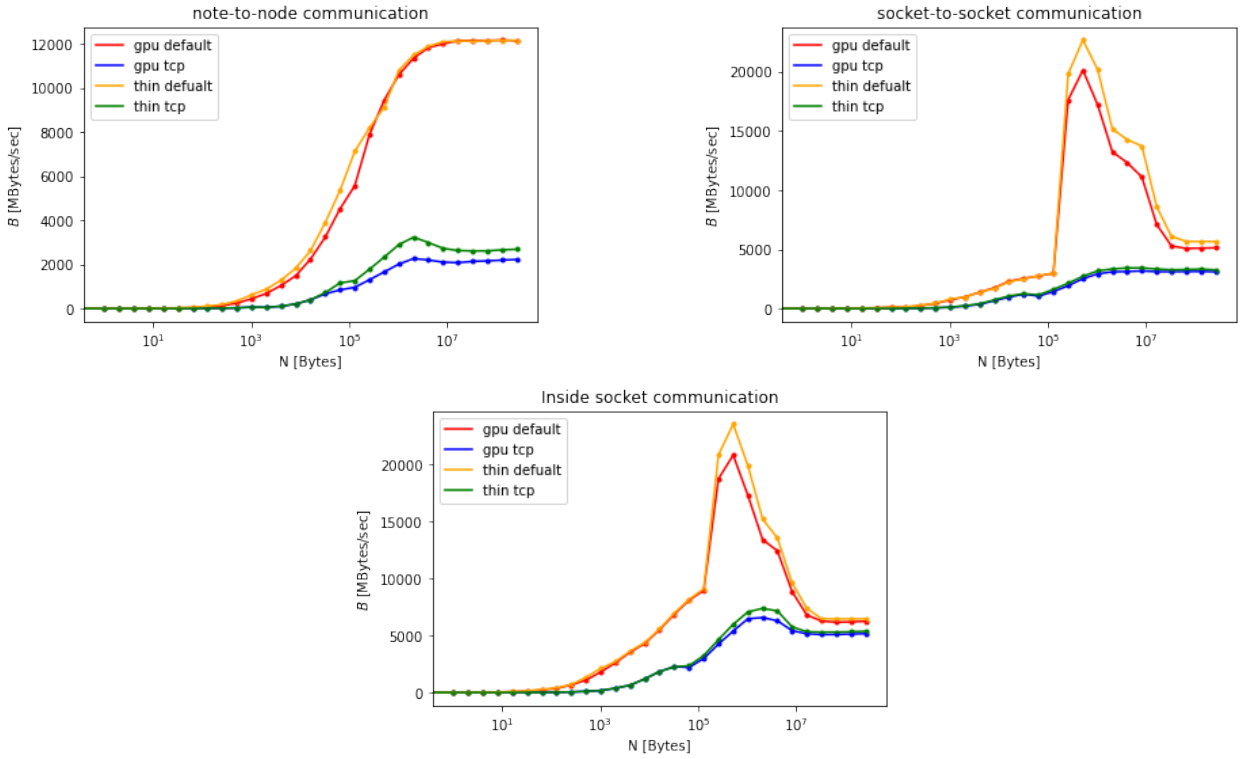


Figure 11: Effective bandwidth as a function of message size for the 3 types of communications and for the different types of node, using the Open MPI implementation.

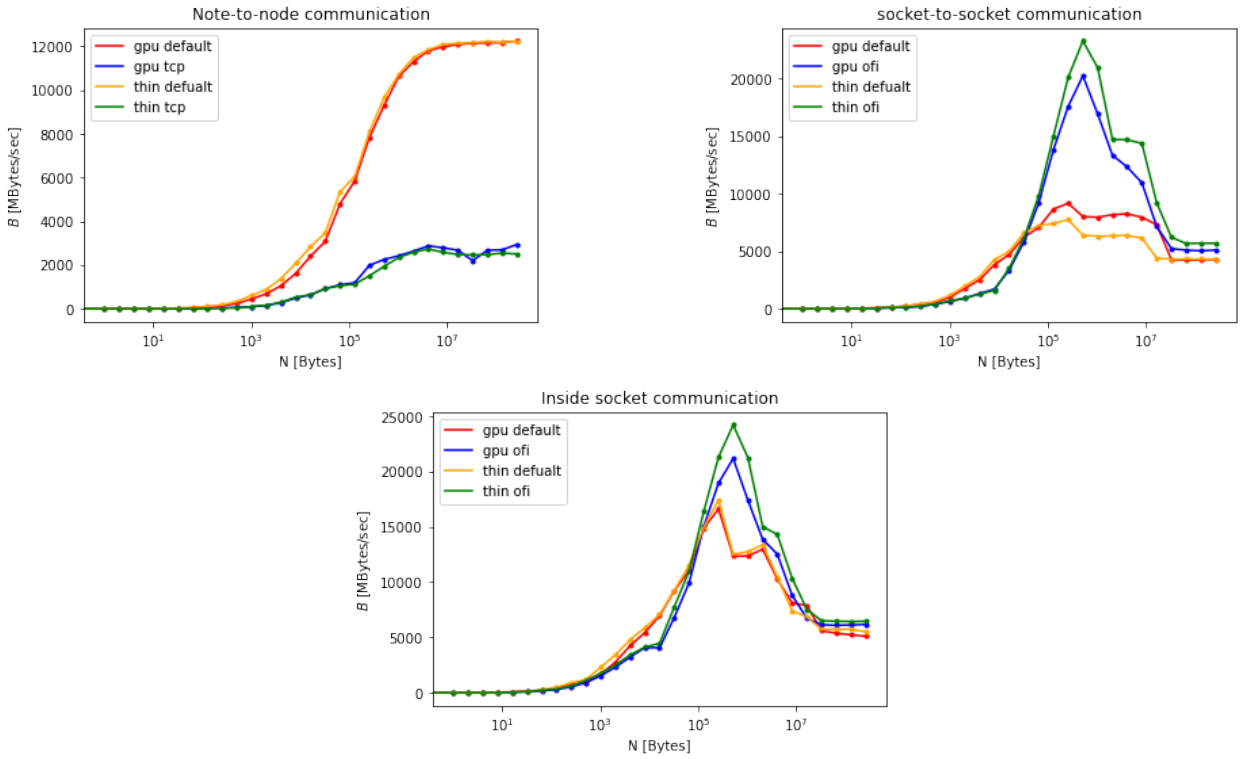


Figure 12: Effective bandwidth as a function of message size for the 3 types of communications and for the different types of node, using the Intel MPI implementation.

| | Latency [$\mu s$] | Bandwidth [bytes/s] |
|---|---|---|
| Intel MPI (Thin) | 5.74 | 12239.2 |
| Intel MPI (GPU) | 8.19 | 12237 |
| Open MPI (Thin) | 4.07 | 12162 |
| Open MPI (GPU) | 4.62 | 12161.4 |

Table 3: Infiniband

| | Latency [$\mu s$] | Bandwidth [bytes/s] |
|---|---|---|
| Intel MPI (Thin) | 14.71 | 2509.8 |
| Intel MPI (GPU) | 248.33 | 2872.9 |
| Open MPI (Thin) | 43.40 | 2680.7 |
| Open MPI (GPU) | 96.76 | 2216 |

Table 4: Ethernet

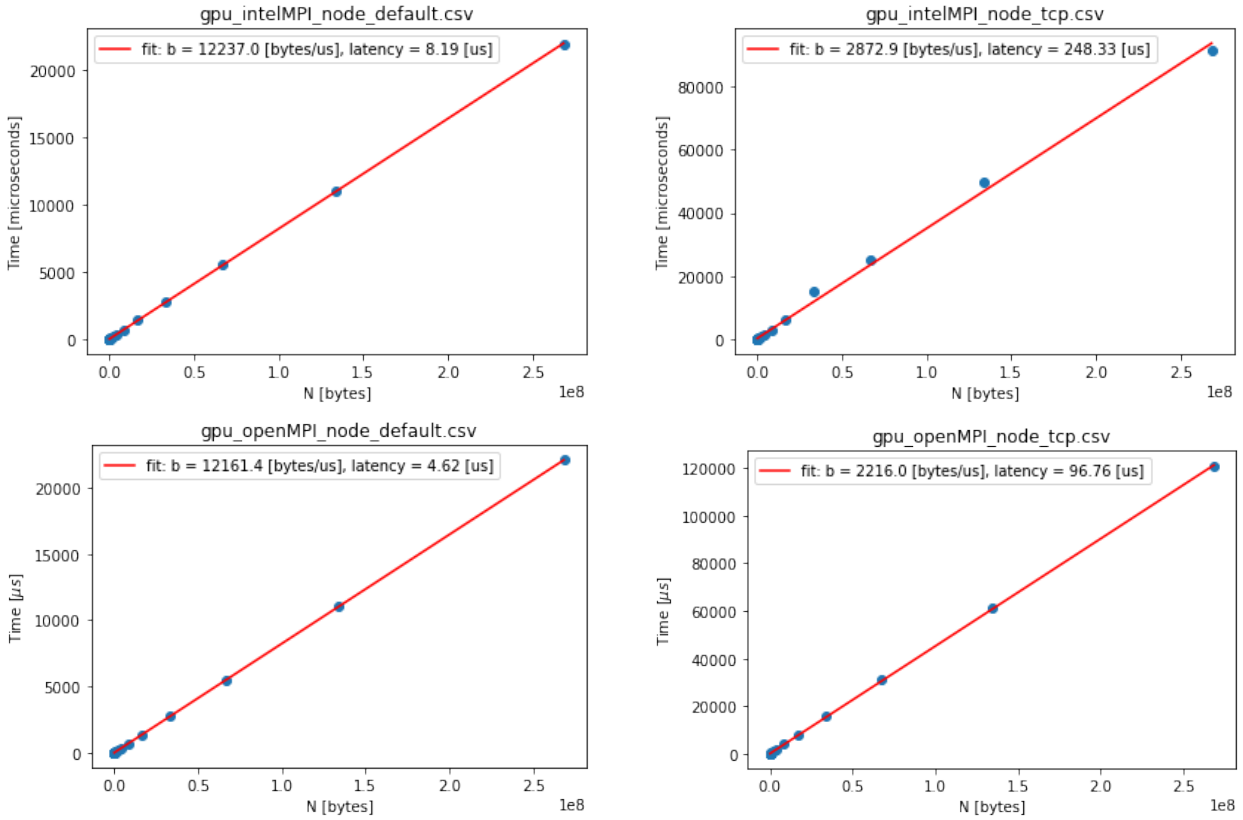


Figure 13: Fit of the communication time model for the node-to-node communication between GPU nodes through the Infiniband and Ethernet network with Intel and Open MPI.

| | Latency [$\mu s$] | Bandwidth [bytes/s] |
|---|---|---|
| Intra-socket | 0.2 | 6447.9 |
| Socket-to-socket | 0.43 | 5771.1 |
| Node-to-node | 4.07 | 12162 |

Table 5: Thin

## 3 Jacobi solver

The goal of this section is to analyze the performance of the Jacobi solver on ORFEO.

The Jacobi solver is a stencil iterative method used to solve the diffusion equation

$$\frac{\partial \phi(\vec{r},t)}{\partial t} = \nabla[D(\phi,\vec{r})\nabla[\phi(\vec{r},t)]] \tag{2}$$

where $\phi(\vec{r},t)$ is the density of the diffusing material at location $\vec{r}$ and time $t$, and $D(\phi,\vec{r})$ is the collective diffusion coefficient for density $\phi$ at location $\vec{r}$. In the domain decomposition framework of MPI programming, the 3-dimensional grid is split into $N$ sub-regions, where $N$ represents the total number of processes, and each of this sub-domain is assigned to a different process. For updating the values of the grid at the borders, each process must know the border values assigned to the neighboring processes, hence time for communicating these values must be taken into consideration.

The performance model assumed is

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

where $N = N_x N_y N_z$ is the total number of processes, L is the size of cubic sub-domains, $T_s$ is the time for the lattice updates of a domain with size L (hence it's constant), and $T_c$ is the communication time; the latter can be modeled in the following way

$$T_c(L,\vec{N}) = \frac{c(L,\vec{N})}{B} + kT_l \tag{4}$$

where $c(L,\vec{N})$ is the maximum bidirectional bandwidth data volume transferred over a node's network link, $B$ is the full-duplex bandwidth, $k$ is the largest number of coordinate directions in which the number of processes is greater than one, and $T_l$ is the latency. We are implicitly assuming a weak-scaling scenario because the program is designed to allocate a cubic subdomain of size $L$ for every process, creating a whole domain of size $L^3N$, which increases proportionally as the number of processes increases.

Because the communication model in equation (1) failed in providing a fitted latency and bandwidth for intra-socket and socket-to-socket communication, the values to insert in the theoretical model are recovered simply by averaging the first 4 elements in the time (for the latency) and the last 4 elements in the effective bandwidth (for the asymptotic bandwidth); instead the fitted values for node-to-node communication are used. The results are summarized in table 5 for *Thin* node and table 6 for GPU node.

The program is run for 10 iterations (convergence is not reached), and for each iteration the program outputs the number of million lattice updates per second (MLUPs), and 4 different times: Maxtime, Mintime, JacobiMin, JacobiMax. Maxtime and Mintime correspond to the maximum and minimum computational times among the processes, while JacobiMin and JacobiMax are respectively the minimum and maximum time considering just the time spent in the lattice updates. For the following analyses, the averages across the 10 iterations of Maxtime and JacobiMax are considered.

| | Latency [$\mu s$] | Bandwidth [bytes/s] |
|---|---|---|
| Intra-socket | 0.23 | 6209.8 |
| Socket-to-socket | 0.43 | 5149.1 |
| Node-to-node | 4.62 | 12161.4 |

Table 6: GPU

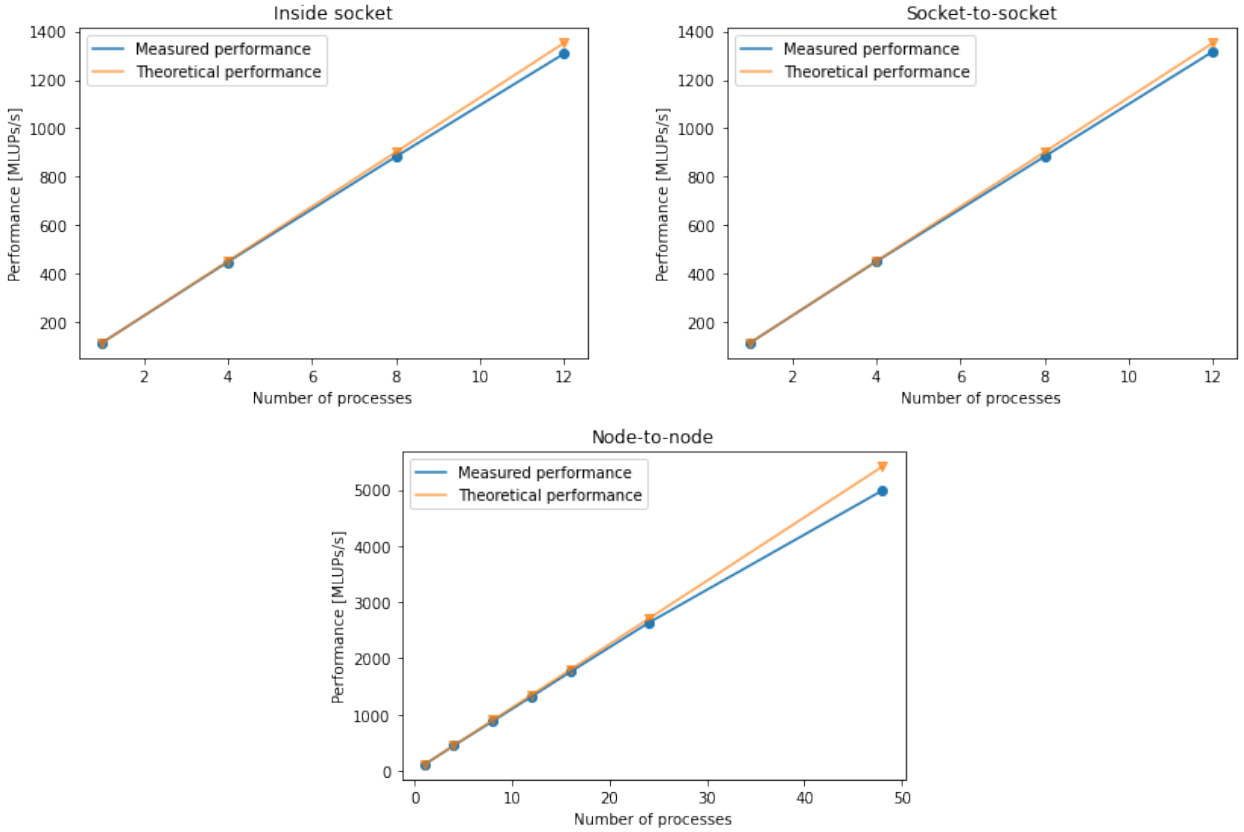


Figure 14: Measured performance vs theoretical performance of the Jacobi solver on Thin node for 3 different types of communication.

### 3.1 THIN

In figure 14 is reported the performance in MLUPs/s for 3 different types of communication: intra-socket, socket-to-socket and node-to-node; the library used is Open MPI with the default behavior.

In all three cases the graphs show that the measured performance is almost identical to the theoretical one; because of the small latency and high bandwidth, $T_c$ (4) plays a less relevant role than $T_s$, and because the latter remains constant, the expected behavior of having a performance proportional to the number of processes used is confirmed by the data.

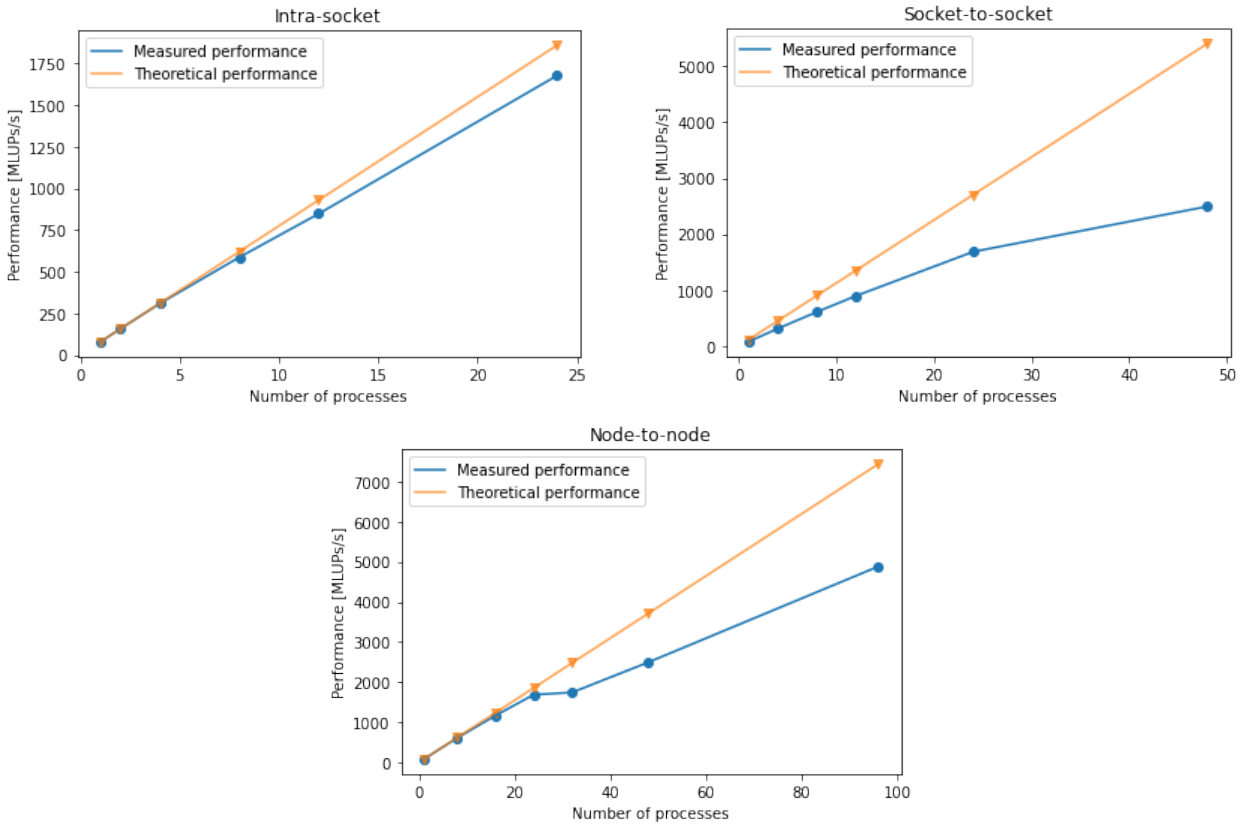


Figure 15: Measured performance vs theoretical performance of the Jacobi solver on GPU node for 3 different types of communication.

### 3.2 GPU

In figure 15 are reported the theoretical and measured performance in MLUPs/s for 3 different types of communication on a GPU node with hyper-threading enabled.

For the communication between cores inside the same socket the measured performance is similar to theoretical one, analogously as to what obtained on Thin nodes. However, in the case of socket-to-socket communication the performance starts to significantly decrease as the number of processes increases. Since this behavior was not observed on *thin* nodes, the explanation can be found in the use of hyper-threading: in particular, in the case of 48 processes, where 2 MPI processes are assigned to a single physical core, the overhead associated for a physical core to deal with more than one process is degrading the performance. Analogous behavior is obtained for node-to-node communication.