First assignment

Contents

Ring

The objective is to implement in C an MPI program using P processors on a ring. The program should implement a string of messages in both directions. More details are reported at https://github.com/Foundations-of-HPC/Foundations_of_HPC_2021/tree/main/Assignment1.

Implementation of the message passing in the ring

To communicate between processors I exploit the *MPI* paradigm. This provides different ways to send messages, each one with its features. This different ways can be either blocking or non blocking.

In my first implementation I use `MPI_Send()` which should be a **blocking** operation. However the MPI standard allows for some freedom in its implementation: often the use of a system buffer allows **small messages to be non blocking**. This means that the sender will just send the message and, once the sending is finished, it will go on with its own following computations. In any case, **deadlocks** may occur if the possible synchronicity of `MPI_Send()` is not taken into account. A typical communication pattern where this may become crucial is actually a ring.

In a ring, if the send is synchronous and all processes call it first to send a message then they all will **wait forever** until a matching receive gets posted. As we said, it may well be that the ring runs without problems if the messages are sufficiently short thanks to a possible buffer, but this may lead to sporadic deadlocks, which are hard to spot.

A simple solution to this deadlock problem is to interchange the `MPI_Send()` and `MPI_Recv()` calls on all odd-numbered processes, so that there is a matching receive for every send executed. I do this for both the streams of messages in the right and left.

```c
if(rank % 2 == 0) { // First send and then receive for both the left and right stream

  // Left stream
  MPI_Send(&msgleft, count, MPI_INT, left_prcs, left_tag, MPI_COMM_WORLD);
  MPI_Recv(...);

  // Right stream
  MPI_Send(&msgright, count, MPI_INT, right_prcs, right_tag, MPI_COMM_WORLD);
  MPI_Recv(&left_bfr, count, MPI_INT, left_prcs, MPI_ANY_TAG, MPI_COMM_WORLD, &left_sts);

} else { // First receive and then send for both the left and right stream

  // Left stream
  MPI_Recv(...);
  MPI_Send(&msgleft, count, MPI_INT, left_prcs, left_tag, MPI_COMM_WORLD);

  // Right stream
  MPI_Recv(&left_bfr, count, MPI_INT, left_prcs, MPI_ANY_TAG, MPI_COMM_WORLD, &left_sts);
  MPI_Send(&msgright, count, MPI_INT, right_prcs, right_tag, MPI_COMM_WORLD);

}
```

However, this solution **does not exploit the full bandwidth** of a non blocking network because only half the possible communication links can be active at any time, at least if `MPI_Send()` is really synchronous. A better alternative is the use a completely non blocking communication that I didn't implement.

Moreover with this implementation there is also a difference in the performance between the cases of an odd or even number of processors. This problem will be observed and explained during the analysis of the results.

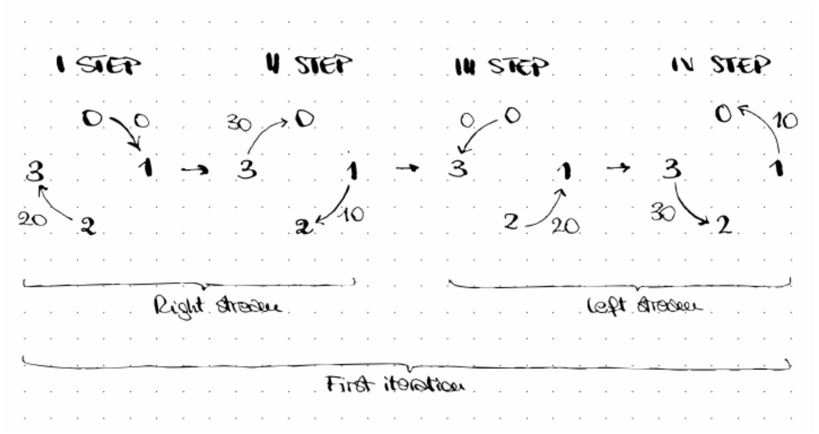


Figure 1: *Little drawing that explains the first iteration with my implementation.*

It is also possible to use `MPI_Ssend()` instead of `MPI_Send()`. This stands for **synchronous send** and it is a blocking operation in the sense that the sender has to wait that the receiver acknowledges the message before being able to go on with some other stuff. I implemented the program with both type of sending and I'll discuss the difference that I observed between the two implementations in a later section.

Messages and tags exchange and stopping criterion

The assignment requires that as first step every processor **sends** a message `msgleft = rank` to its **left** and receives from its right and it **sends** another message `msgright = -rank` to its **right** and receive from the left. Given the above code this only requires to initialize correctly the first values to send exploiting `MPI_Comm_rank(MPI_COMM_WORLD, &rank)` and to define who are the right and left neighbors of a certain processor.

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Getting the rank of current process
MPI_Comm_size(MPI_COMM_WORLD, &size); // Getting the number of processes

// Defining who are the right and left processors on the ring
int left_prcs = rank - 1, right_prcs = rank + 1;
if(rank == 0) left_prcs = size - 1;  // The first processor has the last on its left
if(rank == size - 1) right_prcs = 0; // The last processor has the first on its right

int msgleft = rank, msgright = -rank; // Creating the initial messages to send out
```

Another requirement is that both messages originating from a certain processor should have a **tag proportional to its rank**. This part is simple to implement: we just need to look at the tag received from the right (left), exploiting an object of type `MPI_Status`, and send the next message to the left (right) with that tag. This means that after each message exchange I have the following code.

```
// Updating the tags with the ones received
left_tag = right_sts.MPI_TAG;
right_tag = left_sts.MPI_TAG;
```

Moreover, messages should also be then exchanged between processors for as many times as it takes to **receive back the initial messages**. Then the stopping criterion is that the tag of the message received is equal to the tag of the current process, that is `left_tag == my_tag` (equivalent to `right_tag == my_tag` if things are implemented correctly).

The final requirement is that at every iteration each processor should **add (subtract) its rank** to the received message if it comes from left (right) and at the end should print on a file a certain output. Since the implementation of this part is straightforward I don't discuss it here in details, to leave more space to the analysis of the results.

Network performance model

Before I go on with the measures obtained, let me discuss briefly about what I expect to obtain by measuring the **time** required to the processors **to complete the communication**, that is to say to exchange messages until the first message sent by each processor do a full circle and come back to the processor that has sent it.

In my implementation, every pair of processors, divided in odd and even, sends and receives a pair of messages that constitute the left and right streams in the ring. For this reason, at the end of the complete communication I expect that the **number of messages** sent by a single processor to be $2P$ where P is the number of processors. Moreover, I also expect that a processor receives the same number of messages, i. e. $2P$, at the end. So now the question is: how much time is required to send or receive a message?

In class we've seen that a simple model for the network performance which describes the total transfer time of a message is the following:

$$T_c = \lambda + \frac{s}{b_n}$$

where:

- T_c is the total transfer **time** of a message;
- λ is the **latency** of the network, i. e. the time to setup the communication channel
- s is the **size** of the message to send out;
- b_n is the asymptotic network **bandwidth** measured in Mb/s.

In our case, as I observed while doing the benchmarking on ORFEO (I'll report in more details later), the latency to send 4 bytes between 2 processors could be 0.2 μs if we pin the processes on the same socket, 0.4 μs if we pin the processes on the different sockets in the same node or about 1 μs if we are on different nodes.

Anyhow, the asymptotic network bandwidth on the other hand should be in the order of the GB/s while the message that we are sending consists of 4 bytes (i. e. 32 bits). Therefore the second term should be negligible. So our model is reduced to $T_c = \lambda$, that is to say the total transfer time of a message should be simply equal to the latency.

At this point I think I have to identify 2 distinct cases:

- **non blocking** operations (`MPI_Send()`): when a processor sends a message it doesn't have to wait for any acknowledgment from the receiver, so the **time to send is negligible**. On the other hand when a processor receives a message it has to wait *at least* the total transfer time of a message, i. e. the latency in this case. Moreover, in my implementation, after a processor has sent a message it then waits to receive another message. Has I just explained, it has to wait at least T_c to receive a message from another processor. But that other processor that should be sending is receiving from someone else and so it has to wait T_c. This implies that **each processor**, after the first batch of messages has been sent out from the even processors, **has to wait** $2T_c$ **to receive** a message (I hope that the figure is clearer than my explanation). For this reasons, for the fact that each processor have to exchange

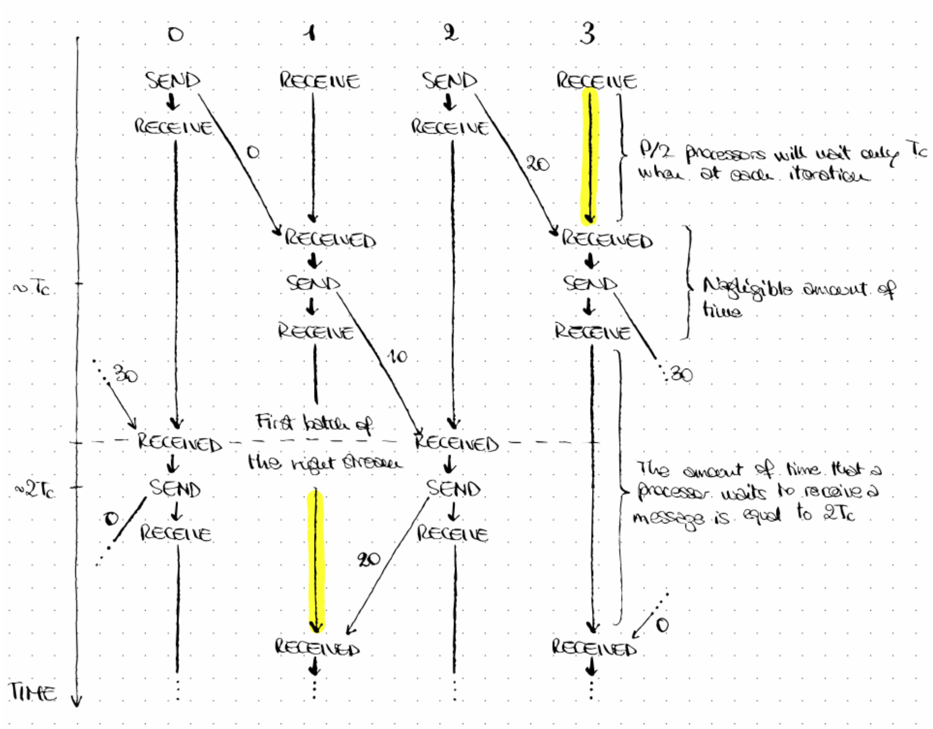

0
1
2
3
SEND
RECEIVE
RECEIVED
0
20
10
30
P/2 processors will wait only T_c when at each iteration
Negligible amount of time
First batch of the right stream
The amount of time that a processor waits to receive a message is equal to $2T_c$
$\sim T_c$
$\sim 2T_c$
TIME

Figure 2: *Little drawing that explains the times in the non blocking case.*

$N = nP$ messages, where n constant expresses the number of streams in the ring, and the processors work in parallel I expect that:

$$T_t = \sum_{i=1}^{N} 2T_c = 2NT_c = 2n\lambda P$$

is the **total time** necessary to complete the exchange of all the messages;

- **blocking** operations (`MPI_Ssend()`): in this case when a processor sends a message it **has to wait for the acknowledgment** that the message has been received by the receiver before going on with other operations. This means that the message travels once from the sender to the receiver in a time T_c and, moreover, the acknowledgment has to travel back from the receiver to the sender again in a time equal to T_c. Therefore at each send operation a processor will wait $2T_c$ before being able to go on with the receiving part. Moreover (I suppose but I do not know that), in this situation, after the message has arrived, the channel of a processor who has sent is occupied by the arriving of the acknowledgment and, until the acknowledgment arrives, no other processor can send a message trough that channel. With this I mean that also any other processor who wants to send to that processor has to wait the the acknowledgment is arrived before it can send a message to it. For this reasons, I think I should have:

$$T_t = \sum_{i=1}^{N} 4T_c = 4NT_c = 4n\lambda P$$

as the **total time** necessary to complete the exchange of messages.

But this is not the end of it. The model above reported works only in the case that all the processors are **on the same socket** and so they should have the same latency for pair communication. What if the processors are on different socket or even different nodes?

Well, this complicate things. However I can try to explain also this situation in some way. Let's suppose to have $P = F + S$ processors of which F are on the **first socket** and S are on **second socket** in the same node. The processors within one of the two sockets, indifferently which one, will communicate between them with a latency $\lambda_F = \lambda_S = \lambda = 0.2$ μs but between the two sockets the latency will be $\lambda_D = 0.4$ μs.

Let's now suppose to do one full circle of the ring with 1 message. Then, up to reordering, the first $S-1$ communications will be characterized by a latency λ (and so to receive a processor waits 2λ), the following one by a latency λ_D, the $A-1$ following ones will have λ and, at the end, the final one that closes the circle will have λ_D. Notice that to receive a message from another socket in our implementation a processor has to wait $\lambda_D + \lambda$. Indeed this is intuitive by looking at the previous drawing. Translating this in a formula and supposing to exchange $N = nP$ messages in a complete communication, with n constant, I should have:

$$T_t = n(F-1)(2\lambda) + n(\lambda_D + \lambda) + n(S-1)(2\lambda) + n(\lambda_D + \lambda = 2n\lambda P + 2n(\lambda_D - \lambda)$$

in the case on **non blocking** routines. So, basically, we add a constant but the slope should not change. A similar reasoning could be done if processors are on different nodes and the communication is blocking I think. It would be slightly more complicated though. I'm not doing it here because It would require more time that the one I have.

To conclude, as we'll observe during the analysis of the results, this models are pretty good in some cases, but not so good in many others.

Measures

Now to the boring part: measurements. So, to take measures of the time I repeated a lot of times (10^6) the complete exchange of messages between different numbers of processors. At each iteration, for each processor, I **added** the time that the particular processor took to complete the communication to the previous ones. That is to say, **at the end** of the program using P processor I have P of the following measures:

$$\sum_{i=0}^{m} T_i^k$$

where m is the total **number of repetitions** of the complete communication and T_i^k is the time that the processor k, with $k \in 1, ..., P$, took to complete the i-th complete exchange of messages. This is basically equivalent to take the mean for the time taken by each processor but I'll analyze the results in the relative section.

Now, I did the above for the **8 cases** reported in the following table.

Label	Processors	Nodes	streams	Sending mode
1	24	1	1	Non blocking
2	24	1	1	Blocking
3	24	1	2	Non blocking
4	24	1	2	Blocking
5	48	2	1	Non blocking
6	48	2	1	Blocking
7	48	2	2	Non blocking
8	48	2	2	Blocking

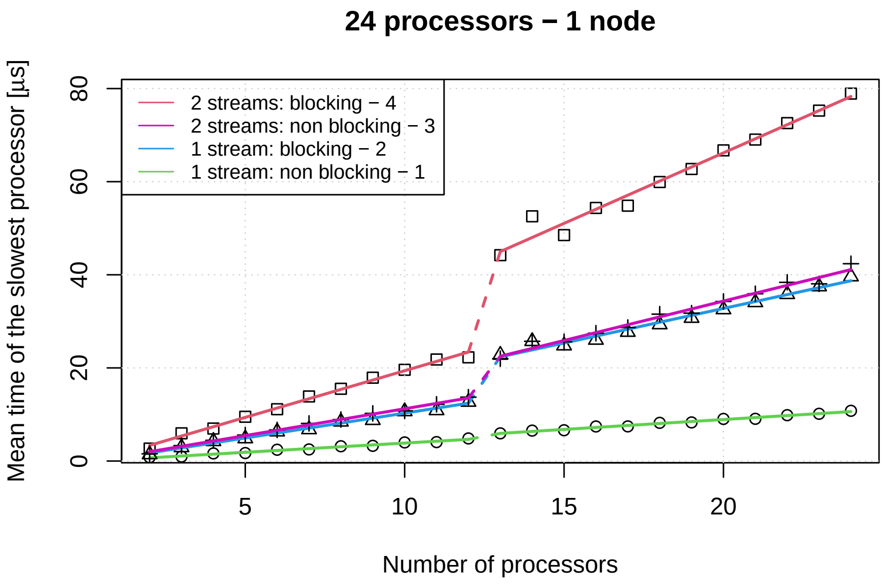
And, **important note**, i pinned at each run the processes **by core**. In this way I was making sure that the runs with a number of processors smaller than 13 would have been on the same socket and the runs with a number of processors smaller than 25 on the same node.

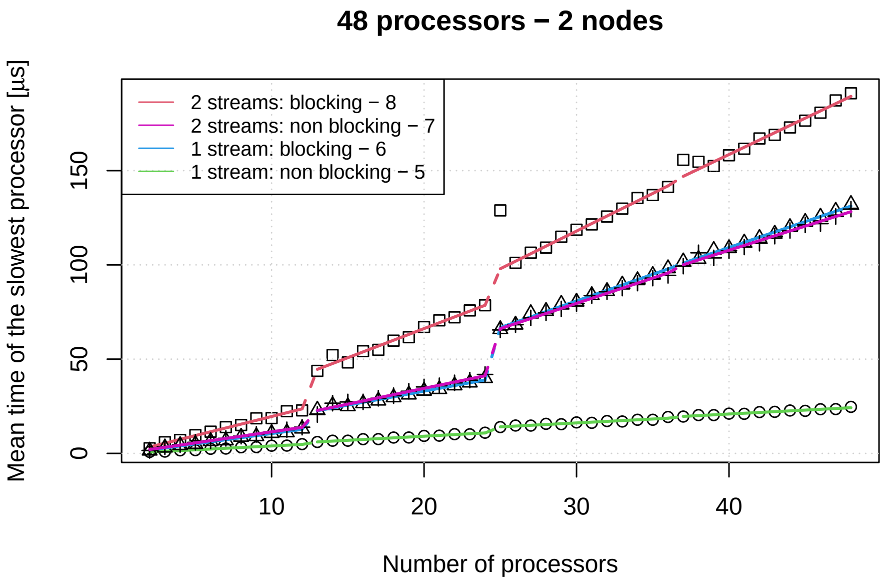
Another important note: the time reported in the following plots for each number of processors P is given by

$$\max_{k\in\{1,\ldots,P\}} \sum_{i=0}^{m} \frac{T_i^k}{m}$$

that is to say time required by the **slowest processor on average** to finish the communication.

Well, let's see what I got. By the way, I already fitted the data with the linear model reported above (details about the fit will be briefly discussed in the section of the data analysis).





Notice the various jumps when we start using processors in different sockets and node as is intuitive.

One last comment, in the above fits I **omitted** the measures taken by the processors 25 and 37 from the fit in the case of 2 streams with blocking mode, since they are clearly **out liners**. Why they are out liners it is not completely clear to me.

Analysis of the results

Let's now estimate a couple of values for the latency and see if they make sense with what I was expecting.

As I explained, I suppose a linear regression model to explain the network communication. Without going in too much detail since this isn't the aim of the assignment, this means that I can fit the data using the least squares criterion. In other words I minimize the sum of squared residuals which appear in the linear relation. Therefore that I can estimate the various parameters. I'd like to discuss here all the details but I don't have time and I think that the report is already too long. For this reasons I just report the results in the following tables for the first 4 cases.

24 processors - 1 Node

Label	Processors	Intercept [μs]	Slope [μs]	Estimated latency in socket [μs]	Estimated latency between sockets [μs]
1	<13	-0.11	0.40	0.20	/
1	>12	0.41	0.43	0.21	0.42

Label	Processors	Intercept [μs]	Slope [μs]	Estimated latency in socket [μs]	Estimated latency between sockets [μs]
3	<13	-0.31	1.15	0.27	/
3	>12	0.49	1.69	0.42	0.55

Label	Processors	Intercept [μs]	Slope [μs]	Estimated latency in socket [μs]
2	<13	-0.39	1.07	0.27
2	>12	3.08	1.48	0.37

Label	Processors	Intercept [μs]	Slope [μs]	Estimated latency in socket [μs]
4	<13	-0.65	2.00	0.27
4	>12	5.61	3.03	0.38

I do not report the results in the other 4 cases for brevity and also because I didn't discussed a model for the communication between nodes (and also for blocking communication on different sockets).

The estimated latency in a sockets or between two sockets is calculated from the models that followed from my reasoning. We know from the benchmarking that they should be about 0.2 μs and 0.4 μs respectively. Clearly, from the estimate values, only the case with 1 stream of messages and non blocking communication seems to give expected results while the others not really. Moreover we can observe that the slope in the non blocking communication case with 2 streams changes differently from my expectations and, in general, it changes for every case except the first one.

Moreover, the model for blocking communication doesn't seem to give good results. Indeed it would be better to have a factor 5 and not 4, but why? I don't know at the moment.

This means that there is something more to take into account when we consider 2 streams of messages and blocking communication in my particular implementation of the ring. In any case, it seem that, at least in the simplest case, my observations could be correct. This validates also the model discussed in class in the simplest case since the other one come from it.

Matrix sum

The objective is to implement a simple 3D matrix-matrix addition in parallel using a 1D,2D and 3D distribution of data using virtual topology and collective operations. More details are reported at https://github.com/Foundations-of-HPC/Foundations_of_HPC_2021/tree/main/Assignment1.

Implementation of the sum

So, first of all, the request is to use collective operations to communicate among MPI processes. Moreover the program should accept as input the sizes of the matrices and should then allocate and initialize them using double precision random numbers.

Let's start tackle this from the bottom by explaining how I generated random numbers. To generate random double I just exploited the `rand()` function present in the `stdlib.h` library of C in the proper way. I defined a function `randfrom(double min, double max)` to generate random doubles from `min` to `max` (by default in my implementation from -100 to 100). Moreover I used `srand(time(NULL))` present in the library `time.h` to seed them with the time at which the program runs.

Now, accepting as input the sizes and the allocating and initializing the matrices with what we received is a little bit more complicated. In my implementation the processor 0 was taking care of reading the values from the input and then send them to all the other processes with collective operations. Then all the processes will allocate a copy of the matrices, called `a`, `b` and `sum` in my implementation, but only the processor 0 will initialize `a` and `b`, that is the matrices to sum, them with the function that I defined.

```
if(rank == root) { // If we are on processor 0

 scanf("%d", &dim_x); // Elements on the first direction
 scanf("%d", &dim_y); // Elements on the second direction
 scanf("%d", &dim_z); // Elements on the third direction

}

unsigned dim_mat = dim_x * dim_y * dim_z; // Computing the dimension of the matrix

// Broadcast the dimensions
MPI_Bcast(&dim_x, 1, MPI_INT, root, MPI_COMM_WORLD);
MPI_Bcast(&dim_y, 1, MPI_INT, root, MPI_COMM_WORLD);
MPI_Bcast(&dim_z, 1, MPI_INT, root, MPI_COMM_WORLD);

// Everyone allocates the matrices using doubles
double *a = (double *)calloc(dim_mat, sizeof(double));
double *b = (double *)calloc(dim_mat, sizeof(double));
double *sum = (double *)calloc(dim_mat, sizeof(double));

if(rank == root) { // Processor 0 initializes the matrices with random numbers

  for(unsigned i = 0; i < dim; i ++) {

    a[i] = randfrom(-100, 100);
    b[i] = randfrom(-100, 100);

  }

}
```

After we did this we just have to allocate in each processor 3 arrays, called `scattered_a`, `scattered_b` and `scattered_sum` in my implementation, which will contain the entries of the matrices that each processor has to work with. Each one of these array will contain `elements` entries. `elements` is an integer computed by dividing the number of total entries of the matrices by the number of processors. Notice that in this case we are required to use always 24 processors and the dimension of the matrices that we want to sum is always a multiple if 24. So we don't have to take care of any remainder for this particular exercise but this is not true in general. So, since in a computer we represent any multi-dimensional matrix with a 1 dimensional array, the processor i will take care of the i-th chunk of the 1 dimensional arrays that represent the matrices.

After all this preparation phase we can finally use collective operation to scatter the matrices, sum them in the matrix *sum* and them gather the results in the processor 0. To do this we can simply use `MPI_Scatter()` and `MPI_Gather` in the following way.

```
MPI_Scatter(a, elements, MPI_DOUBLE, scattered_a, elements, MPI_DOUBLE, root,
            new_communicator);
MPI_Scatter(b, elements, MPI_DOUBLE, scattered_b, elements, MPI_DOUBLE, root,
            new_communicator);

for(unsigned i = 0; i < elements; i++) {

 scattered_sum[i] = scattered_a[i] + scattered_b[i];

}

MPI_Gather(scattered_sum, elements, MPI_DOUBLE, sum, elements, MPI_DOUBLE, root,
           new_communicator);
```

And that should be it for the implementation of the sum.

A final note regarding this last snippet of code: I measures the times at the begin and at the end of it in my implementation. So at the end each processor will compute the time that it took to execute only this particular part of the whole program. I also use a `MPI_Barrier()` before starting to measure the time.

Virtual topology

To implement a virtual topology given a number of dimensions `dim` i did the following.

```
// Create a new communicator
MPI_Comm new_communicator;

// Create a cartesian virtual topology using the new communicator
MPI_Cart_create(MPI_COMM_WORLD, dim, dims, periods, reorder, &new_communicator);
```

The parameter `dims` which appears in the code is an array with `dim` entries that constitute the number of processors in each direction. Also these values are given as input by the user at the beginning of the run.

Then I just used `new_communicator` in the collective operations.

Measures

We are required to model the network performance and try to identify which the best distribution given the topology of the node you are using for the following sizes:

- 2400 x 100 x 100 ;
- 1200 x 200 x 100 ;
- 800 x 300 x 100;

Moreover we have to discuss performance for the three domains in term of 1D, 2D or 3D distribution keeping the number of processor constant at 24. Therefore below I provide a table with all possible distribution on 1D, 2D and 3D partition where I report the time that the slowest processor took to complete the scattering and gathering.

Virtual topology dimension	Partition considered	Matrix dimension	Time of the slowest [s]
1	24	2400 x 100 x 100	0.108882
1	24	1200 x 200 x 100	0.105766
1	24	800 x 300 x 100	0.106709
2	24 x 1	2400 x 100 x 100	0.106352
2	24 x 1	1200 x 200 x 100	0.107124
2	24 x 1	800 x 300 x 100	0.106613
2	12 x 2	2400 x 100 x 100	0.106829
2	12 x 2	1200 x 200 x 100	0.106709
2	12 x 2	800 x 300 x 100	0.106933
2	8 x 3	2400 x 100 x 100	0.108002
2	8 x 3	1200 x 200 x 100	0.108461
2	8 x 3	800 x 300 x 100	0.107451
2	6 x 4	2400 x 100 x 100	0.105051
2	6 x 4	1200 x 200 x 100	0.107451
2	6 x 4	800 x 300 x 100	0.109398
3	24 x 1 x 1	2400 x 100 x 100	0.106271
3	24 x 1 x 1	1200 x 200 x 100	0.105744
3	24 x 1 x 1	800 x 300 x 100	0.105707
3	12 x 2 x 1	2400 x 100 x 100	0.106650
3	12 x 2 x 1	1200 x 200 x 100	0.107856
3	12 x 2 x 1	800 x 300 x 100	0.112985
3	8 x 3 x 1	2400 x 100 x 100	0.108221
3	8 x 3 x 1	1200 x 200 x 100	0.108621
3	8 x 3 x 1	800 x 300 x 100	0.106750
3	6 x 4 x 1	2400 x 100 x 100	0.107564
3	6 x 4 x 1	1200 x 200 x 100	0.106120
3	6 x 4 x 1	800 x 300 x 100	0.103427
3	6 x 2 x 2	2400 x 100 x 100	0.108241
3	6 x 2 x 2	1200 x 200 x 100	0.104122
3	6 x 2 x 2	800 x 300 x 100	0.105098
3	3 x 4 x 2	2400 x 100 x 100	0.106015
3	3 x 4 x 2	1200 x 200 x 100	0.108835
3	3 x 4 x 2	800 x 300 x 100	0.107558

Differences in the measure

Since we use collective operations it should not make a difference which dimension of virtual topology we are implementing and how many processors we put in each direction. Moreover, since we are representing the matrices as a 1 dimensional array of N = 2400 x 100 x 100 = 1200 x 200 x 100 = 800 x 300 x 100 = 240'000'000 elements and, in order to make the sum, we give to the first processor entries [0, …, $N/P-1$], to the second the elements [N/P, …, $2N/P$] and so on, it shouldn't even make a difference if we're using

2400 x 100 x 100, 1200 x 200 x 100 or 800 x 300 x 100 matrices. This because processors don't have to communicate between each other in order to make the sum: after the scattering is done each processor will do its thing and then wait for the gathering of the results by the root.

This is basically what we can see from the data: we obtain the same results for each possible configuration if we consider the differences between the different measures randomly caused, which is a fair assumption seen that they are pretty small.

A possibly interesting remark

So, in the section about the measures I didn't put the measure of the time of each processor but only the ones of the slowest one (that, by the way, it is always the 16th). If I would had put them the would have look something like this.

Proc	1D, 24, 2400 x 100 x 100 [s]	2D, 6 x 4, 1200 x 200 x 100 [s]	3D, 4 x 3 x 2, 800 x 200 x 100 [s]
5	0.040738	0.042466	0.044197
3	0.043157	0.044088	0.041102
7	0.043665	0.044233	0.044712
11	0.043707	0.045077	0.044778
9	0.044083	0.044325	0.044213
13	0.044852	0.045379	0.046144
21	0.045058	0.046162	0.046419
23	0.045590	0.046201	0.046375
15	0.045738	0.044803	0.045663
19	0.045850	0.045128	0.045532
17	0.045999	0.046235	0.045401
1	0.046748	0.046570	0.047169
6	0.050684	0.051562	0.050848
10	0.050837	0.052288	0.050880
18	0.052343	0.052728	0.051536
22	0.052483	0.053468	0.052309
14	0.052612	0.051777	0.051611
2	0.053646	0.053512	0.053033
20	0.063476	0.064800	0.063373
12	0.063734	0.063147	0.062767
4	0.064632	0.064274	0.063979
8	0.086866	0.085402	0.085696
0	0.108759	0.106599	0.107468
16	0.108882	0.107451	0.107558

Also the remaining measures are very very similar. So there's clearly a pattern and something is going on. What is it?

When I first saw the results I immediately thought at a possible trick that we've seen in class for the broadcast operation in order to reduce the time spent: the binary tree. So, to see if also `MPI_scatter()` and `MPI_Gather` are performed in similar way one would have to measure independently the times required to perform the various operations. And so I did just in one case (1D with 2400 x 100 x 100), just for the sake of curiosity. The results are reported below (notice that the total time required by the slowest processor to complete the whole communication is quite different than the ones of the previous measures, I don't really know why but this is not important for what this section aims to explain).

Processor	One scatter [s]	Two scatters [s]	Scatters and sum [s]	Only gather [s]	Everything [s]
3	0.012646	0.029528	0.035923	0.010836	0.046807
5	0.013774	0.030254	0.036467	0.010757	0.047468
7	0.005948	0.030907	0.037348	0.010746	0.048014
9	0.026816	0.032117	0.038347	0.010775	0.048283
11	0.015362	0.032497	0.038875	0.010876	0.049439
13	0.016086	0.033281	0.039493	0.010745	0.050215
17	0.017567	0.034755	0.040953	0.010756	0.050803
15	0.016749	0.034041	0.040177	0.010738	0.050922
19	0.018153	0.035344	0.041477	0.010707	0.052166
21	0.018465	0.036032	0.042230	0.010800	0.052769
23	0.006321	0.036759	0.042876	0.010732	0.053145
1	0.012355	0.028975	0.035569	0.010939	0.054223
6	0.013777	0.030192	0.036574	0.020625	0.058078
10	0.015329	0.031674	0.038153	0.020799	0.059361
14	0.016433	0.033202	0.039416	0.020661	0.060859
18	0.017474	0.034518	0.040785	0.020650	0.061965
22	0.018805	0.035876	0.042117	0.020700	0.062983
2	0.012210	0.028724	0.035220	0.020711	0.063621
12	0.015676	0.032491	0.038747	0.040022	0.080035
4	0.012804	0.029504	0.035813	0.039820	0.081934
20	0.018109	0.035139	0.041398	0.040047	0.082013
8	0.014137	0.030372	0.036739	0.072662	0.113231
0	0.026767	0.036783	0.042878	0.102537	0.142818
16	0.016447	0.033201	0.039635	0.103627	0.143882

In the table the rows are ordered by increasing order of the time that the whole communication takes. In two words what I can observe is that the part that takes more time and basically dictate the final order on which the processors will end the communication is the `MPI_Gather()` Moreover, by looking at it I can guess that it is implemented using a binary tree like the one reported in the picture below. Probably also the `MPI_Scatter()` is implemented in the same way but it takes less time on average on each processors and the average time is smaller. I think that this is dictated by the fact that during the `MPI_Gather()` the processor that has to receive everything has to wait for all the others to send.

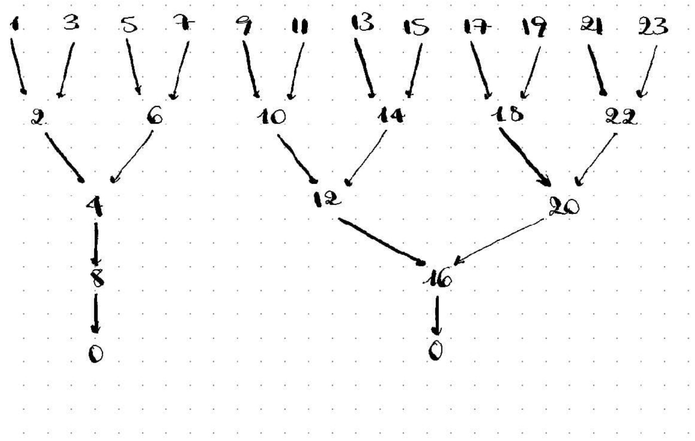


Figure 3: *Little drawing that explains the implementation of the `MPI_Gather()`.*

So the `MPI_Gather()` makes pairs of processors send to a third one. In this way the total time of the communication is decreased and the bandwidth better exploited.

Benchmarking

The objective is to use the MPI Intel benchmark to estimate latency and bandwidth of all available combinations of topology and networks on ORFEO computational nodes. More details are reported at https://github.com/Foundations-of-HPC/Foundations_of_HPC_2021/tree/main/Assignment1.

The MPI Intel benchmark

To exploit the MPI Intel benchmark using Open MPI libraries I followed precisely the instructions given at https://github.com/Foundations-of-HPC/Foundations_of_HPC_2021/tree/main/MPI loading the module `openmpi-4.1.1+gnu-9.3.0`. To exploit the Intel MPI libraries I followed similar steps but I compiled loading the module `intel`. In this case the commands to pin the processors on the same socket, on different sockets or on different nodes using `mpirun` are different. The ones that I used are reported in the relative *.csv* files.

Communication model and a note about the topology

As reported in the part of the report dedicated to the ring, in class we've seen that a simple model for the network performance which describes the total transfer time T_c of a message is the following:

$$T_c = \lambda + \frac{s}{b_n}$$

where λ is the **latency** of the network, i. e. the time to setup the communication channel, s is the **size** of the message to send out and b_n is the asymptotic network **bandwidth** measured in Mb/s.

So I will try to fit this model using a square least method, as requested, on the data that I obtained from the *PingPong* benchmark.

Before going on with some observations about the results a brief comment about the nodes used in the communication: `tnode009` and `tnode010`. These are Intel CPUs with 2 sockets, 12 cores per socket and hyperthreading disabled. Using the `likwid` module and `likwid topology` we can also get to know that the socket 0 hosts the processors 0, 2, 4, . . . , 20, 22 and the socket 1 the processors 1, 3, 5, . . . , 21, 23. Moreover the cache topology is the following:

- the level 1 cache has 32kB (2^{15}B) and is not shared between processors, so each processor has its own;
- the level 2 cache has 1MB (2^{20}B) and is also not shared;
- the level 3 cache has 19MB (~2^{24}B) and is shared between processors on the same sockets, so each socket has a level 3 cache shared among its processors.

It is also possible to observe that the RAM is shared between processors on the same socket, therefore there are 2 NUMA domains.

Given this structure, it is fair to expect 3 different point-to-point communication characteristics deepening whether the message transfer occurs in a L3 group (same socket), between cores on different sockets on the same node or between different nodes.

Measures

The data that I obtained and used in the fits are all present in the *section 2* folder. A note: the values reported there are the means of 10 measures, that is to say of 10 runs of the benchmark. Indeed I did 10 runs of the benchmark for each one of the following cases. Where - is present it means that I didn't specify anything at `mpirun` regarding the network protocols to use so it should had used the default one.

Label	Topology	Point-to-point Messaging Layer	Bit Trasport Layer	Libraries
1	Different nodes	ucx	openib	Open MPI
2	Different sockets, same node	ucx	openib	Open MPI
3	Same socket	ucx	openib	Open MPI
4	Different nodes	ob1	tcp,self	Open MPI
5	Different sockets, same node	ob1	tcp,self	Open MPI
6	Same socket	ob1	tcp,self	Open MPI
7	Different nodes	-	-	Intel MPI
8	Different sockets, same node	-	-	Intel MPI
9	Same socket	-	-	Intel MPI

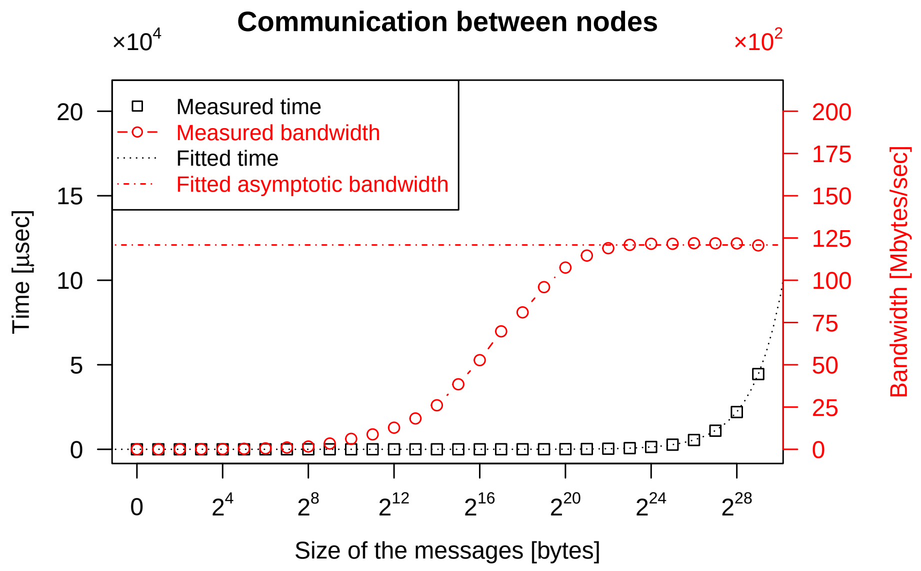
Analysis of the results

With the data from the measures I fitted the linear model using a least square method. By doing so I obtained two fitted values: one for the parameter λ (latency) and one for the parameter b_a (asymptotic bandwidth), which both appear in the model discussed in class. I then plugged these 2 values in the expression above to obtain some values for the time. I will call the values calculated *fitted values*.

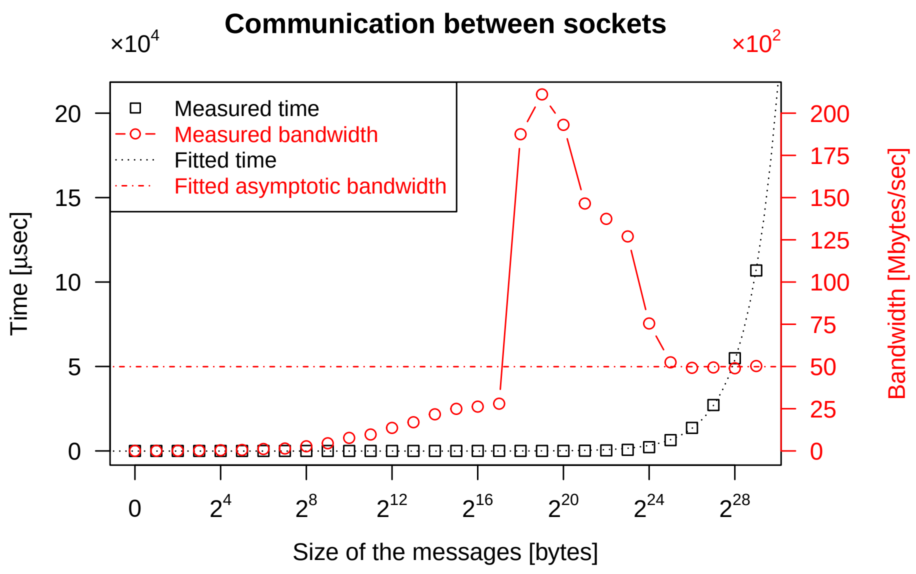
Given what I just explained, in the following plots I show the measured time against the fitted time and the measured bandwidth against the fitted asymptotic bandwidth. Notice that in the graph I putted a logarithmic scale on the x-axis since the benchmark was performed sending messages gradually increasing by a factor of 2. Moreover I plotted every graph with the same scale on the y-axis in order to easily compare them between each other.

Influence of the topology

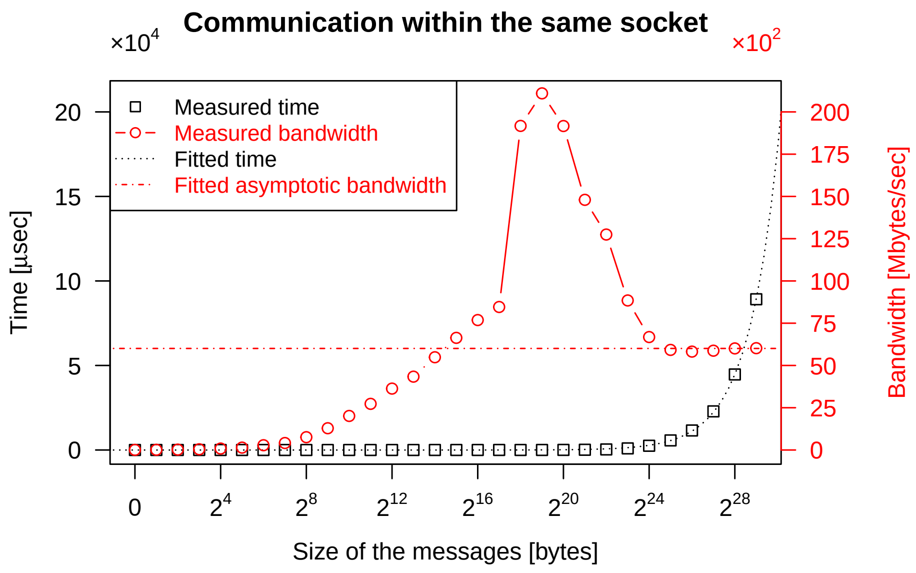
To begin, I'll put the plots of the cases 1, 2 and 3.



Using Open MPI libraries and ucx with openib



Using Open MPI libraries and ucx with openib



Using Open MPI libraries and ucx with openib

So, the above graph compare the results from the communication of 2 processors on different cores, different sockets or the same socket. All of the results are obtain using InfiniBand, that is the high speed network of ORFEO, which should have a bandwidth of 100Gb/s (12.5GB/s).

Now, a little bit of numbers.

Label	Latency for small size messages [μs]	Fitted latency [μs]	Fitted asymptotic bandwidth [MB/s]
1	0.98	-4.06	12091.15
2	0.40	-97.93	4985.99
3	0.20	-22.22	6010.01

First of all the values on the first column (that is to say the latency for small size messages) seem to be correct if compared to what we've seen in class.

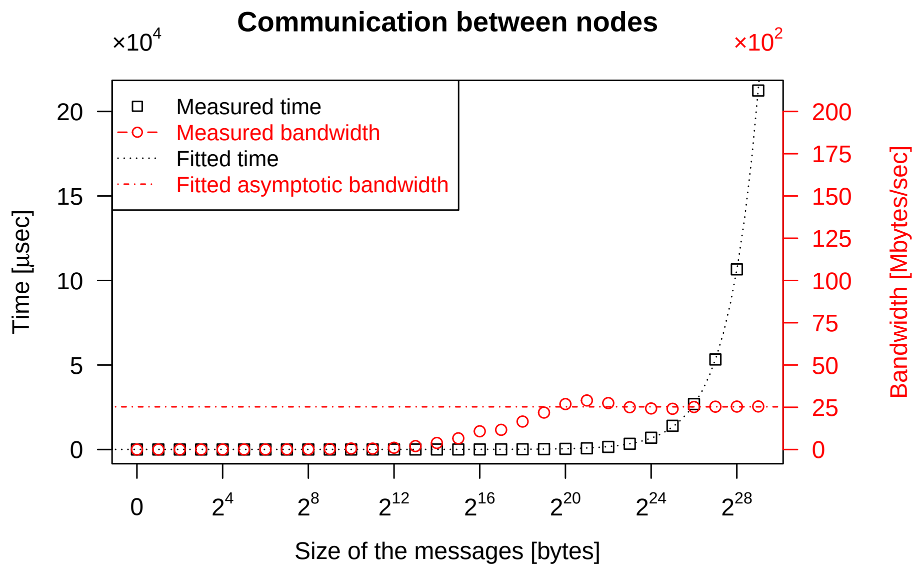
Secondly, the fitted values for the latency are all negative which doesn't make any sense. I think that this is actually caused by the simplicity of the model: a linear model doesn't take into consideration that a negative latency does not make sense and that the latency is constant for the first sizes.

A final remark. As we can see from the first plot, the one which reports the communication between different nodes, the asymptotic bandwidth is a little below 12500 MB/s as we expect. But in the other 2 cases this is not true: the asymptotic bandwidth seems to be around 5000 GB/s. Moreover, while in the first plot the bandwidth behavior seems to be pretty continuous until it reaches the asymptotic bandwidth, in the other 2 cases there's a hump (i. e. a rapid increase between 2^{16} and 2^{17} bytes and then a rapid decrease after 2^{20} bytes) and the asymptotic bandwidth is reached after 2^{24} bytes. A possible explanation is given by the book *Introduction to HPC for scientists and engineers*. There is reported that the explanation lies in how the IMB PingPong code is implemented. Indeed, to get accurate timings measurements even for small messages, the Ping-Pong message transfer is repeated a certain number of times. Therefore the transfer of $sendb_0$ from process 0 to $recvb_1$ of process 1 can be implemented as a *single-copy* operation on the receiver side. If the number of bytes of the message is sufficiently small then the data from $sendb_0$ is located in the cache of process 1 an there is no need to replace or modify these cache entries unless $sendb_0$ get modified. However the send buffers aren't changed in the loop. Thus, after the first iteration the send buffers are located in the caches of the receiving processes and in-cache copy operations occur in the subsequent ones. There are 2 reasons for the performance drop at larger message sizes: first the L3 cache (19MB) is to small to hold both or at least one of the local receive buffer and the remote send buffer. Second, the IMB is performed so that the number of repetitions is decreased with increasing message size until only 1 iteration (the initial copy operation) is done for large messages. Still this doesn't explain why the drop is so big, that I actually do not know.

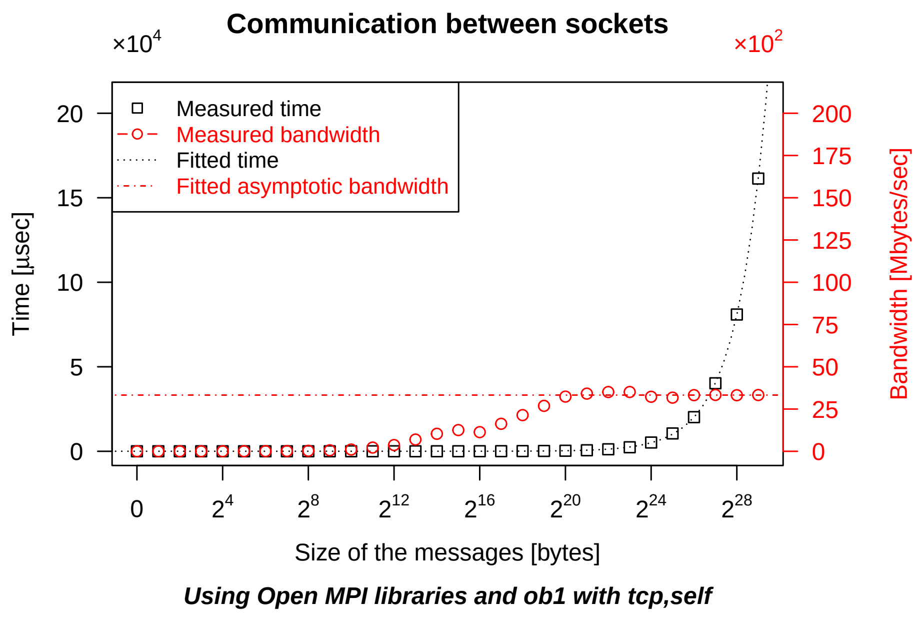
From the bandwidth and latency characteristics shown above, the most important conclusion that I draw from this is that process-core affinity can play a major role for application performance.

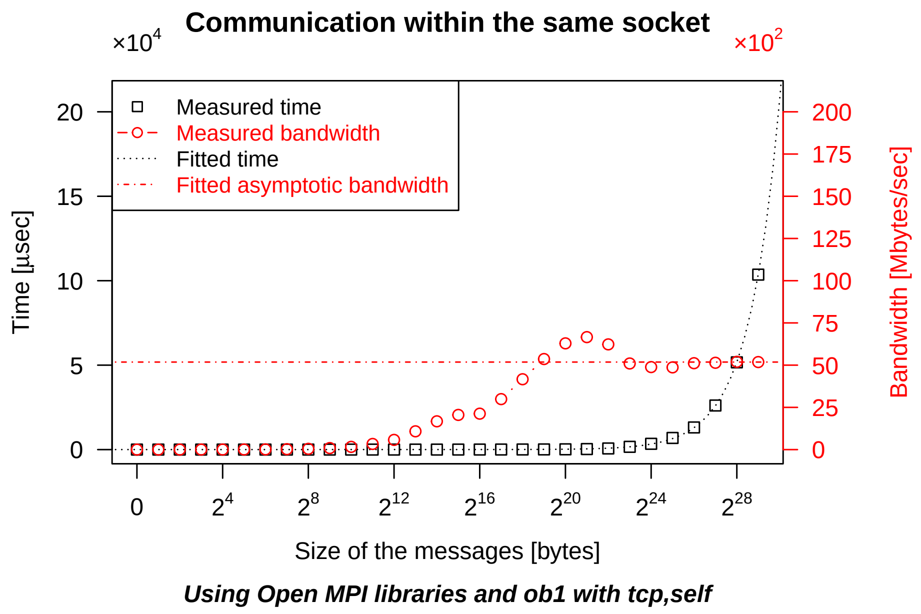
Influence of the network protocol

Here I put the tree plots obtained of the cases 4, 5 and 6.



Using Open MPI libraries and ob1 with tcp,self





The above plots compare the results from the communication of 2 processors on different cores, different sockets or the same socket. All of the results are obtain using TCP which is a different BTL protocol. In this case we're using a 25Gb/s (3.1GB/s) card.

As previously, before commenting I put some numbers.

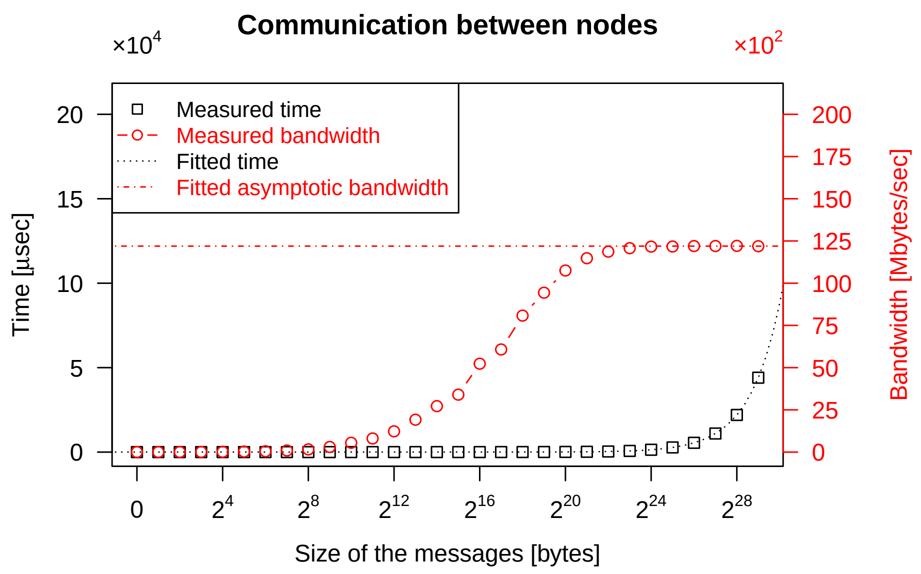
Label	Latency for small size messages [μs]	Fitted latency [μs]	Fitted asymptotic bandwidth [MB/s]
4	16.02	3.4	2592.03
5	8.11	38.88	3283.33
6	5.52	18.49	5032.98

The same comments as before can be done for the first 2 columns. Regarding the asymptotic bandwidth we can see that between nodes it is pretty close to the one theoretically obtainable. In the case of processors on the same node we are even able to obtain an higher value since we exploit the shared L3 cache (I think). Moreover this value is pretty close to the previous one.

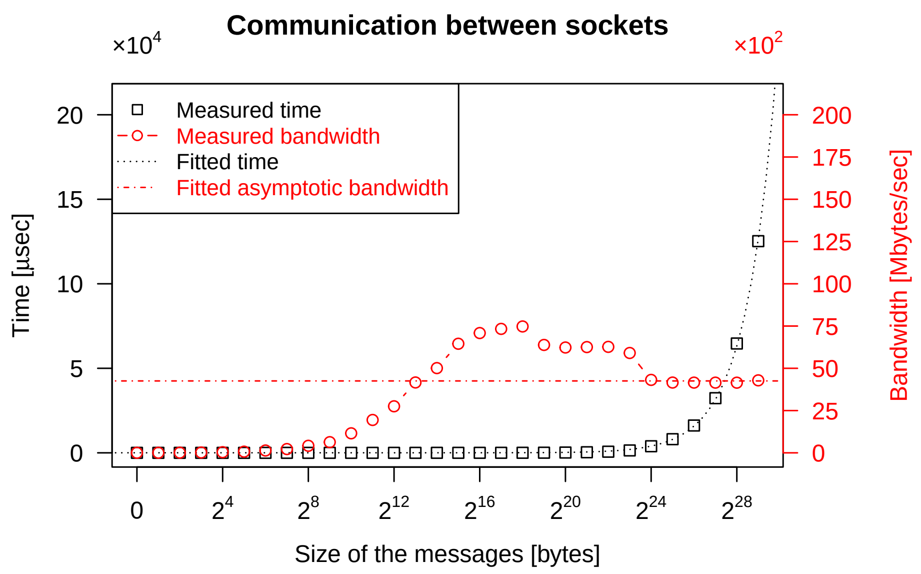
In any case, in all 3 cases we obtain an higher latency and lower bandwidth with respect of before. This is caused by the overhead of TCP, that is to say is only caused by the software

Differences using different libraries

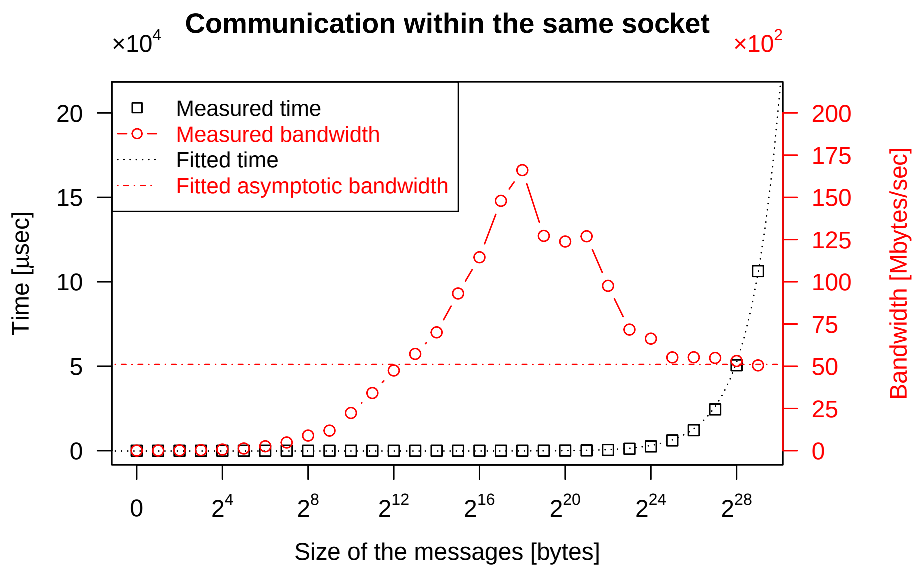
Here put the tree plots obtained in the remaining case and comment them.



Using Intel MPI libraries with default options



Using Intel MPI libraries with default options



Using Intel MPI libraries with default options

So, some numbers.

Label	Latency for small size messages [μs]	Fitted latency [μs]	Fitted asymptotic bandwidth [MB/s]
7	1.15	3.36	12196.3
8	0.43	23.17	4252.59
9	0.23	-194.6	5118.39

We can see that results for pretty small and pretty big messages as one can expect are quite similar while for medium sized messages there is a quite different behavior.

Difference between gpu and thin nodes

I can't do an extend analysis because I don't want to spend anymore time and the report is already pretty long so I report only the following measures for comparison (more measures in the relative folder).

Label	Latency for small messages on thin [μs]	Latency for small messages on gpu [μs]	Asymptotic bandwidth on thin [MB/s]	Asymptotic bandwidth for on gpu [MB/s]
1	0.98	1.36	12091.15	12066.73
2	0.40	0.42	4985.99	5671.91
3	0.20	0.22	6010.01	6382.62
4	16.02	15.46	2592.03	2157.72
5	8.11	8.73	3283.33	3120.7
6	5.52	5.02	5032.98	5167.68
7	1.15	1.34	12196.3	12187.82
8	0.43	0.48	4252.59	4432.4
9	0.23	0.29	5118.39	5063.43

As we can see performances are pretty similar.

Jacobi

In this fourth and last exercise we are required to compare performance observed against performance model for Jacobi solver. More details are reported at https://github.com/Foundations-of-HPC/Foundations_of_HPC_2021/tree/main/Assignment1.

The Jacobi solver

As we've discussed in class, the Jacobi solver is a prototype for many stencil based iterative methods in numerical analysis and simulation. In its basic form it aims to solve the diffusion equation fo a scalar function $\phi(\mathbf{r}, t)$.

In the implementation that we're working with, the Jacobi solver is a 7-point stencil method, that is to say to update the value in a certain point we need the values of 6 nearby points, 2 in each direction (we're working with 3D matrices). So, if we want to work in parallel and distribute the data between the processors, we have to take particular care of the sub-domains boundaries. This is done using halo layers in the implementation that we're using. Therefore virtual topology and point-to-point communication between processors are implemented.

For these reasons, the Jacobi solver is a pretty complex task that involves both computation and communication. The code that we're using provides different measures at the end of the execution: the time taken by the fastest and the slowest processors complete the whole algorithm, taken by the fastest and the slowest processors complete only the computation, the residuals and the MLPUs (Million Lattice Updates per second). In particular, the following pseudo code taken directly from the implementation that we're using shows where the measures of the time are taken precisely.

```
call MPI_BARRIER(...)
starttime = MPI_WTIME() ! Measure starts

...
! Computation
...

JacobiTime = MPI_WTIME() - starttime ! Computation time

do dir = 0, 1

  ...

  do counter = 1, 3

    call MPI_CART_SHIFT(...) ! Neighbour coordinates

    ...
    ! Communication
    ...

  enddo

enddo

call MPI_BARRIER(...)
runtime = MPI_WTIME() - starttime ! Run time
```

Moreover, all of these measures are taken 10 different times for 10 different executions. That is, the above code is repeated 10 times.

I'll try to exploit these information to see if the theoretical results from the next section are more or less correct.

Performance analysis

As we've seen in class, the overall performance depends on the number of grid points which is distributed over a number $N = N_x N_y N_z$ processors. Indeed we have that the global performance is give by:

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

where:

- $T_s(L)$ is the sweep part, i. e. the sequential time;
- $T_c(L, \mathbf{N})$ is the communication time;
- L^3 is the sub domain size.

A note: with my measure I'm trying to discuss weak scaling so I keep the sub domain size the same regardless of the number of processors. This implies that for each number of processors that I consider I have to change the size of the whole domain accordingly. For example, if for 1 processor I consider a 1200 x 1200 x 1200 matrix then for 12 processors (distributed in the 3 direction as 4 x 3 x 2) I have to consider a 4800 x 3600 x 2400 matrix. Therefore the time T_s required to compute all the cell updates in a Jacobi sweep should remain constant in the different cases.

Regarding the communication time we've again seen in class that, with some proper assumptions, it should be given by:

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

where:

- $c(L, \mathbf{N})$ is the amount of data volume transferred over a node network link;
- B is the bidirectional bandwidth of the network link;
- T_l is the latency of the network;
- k is the largest number (over all the domains) of coordinate directions in which the number of processors is greater than one.

In particular, $c(L, \mathbf{N})$ is given by:

$$c(L, \mathbf{N}) = 2kL^2 8$$

where the 8 is present because we're sending *doubles* (8 bytes).

Before ending the section a remark: we know from the benchmark that, for any kind of topology that we consider on ORFEO, the latency is in the order of μs (10^{-9} s), the bandwidth intranode for large messages is about 5 GB/s (10^9 B/s) an the the bandwidth between nodes for large messages is about 12 GB/s. Since we are sending messages large as one of the surfaces of the sub domains, that is to say $L^2 = 1200^2$ B (in the order of 10^6 or 2^{20} bytes) then the second term in the sum that gives T_c in negligible.

I will compare this theoretical results with the measure that I obtained.

Measures

So, as said previously, the code used to implement the Jacobi solved provided us with some different measures. For my aims I will only use the time taken but the slowest processor to complete the whole execution and the time taken by the slowest processor to complete only the computation. Our code runs the all thing 10 times so all the values reported are the mean computed over the different values obtained from those runs. I report in the table below the values obtained in this way.

Mapped by	N	N_x	N_y	N_z	Run [s]	Computation [s]	Communication [s]	MLPUs [MB^3/s]
-	1	1	1	1	15.32	15.06	0.26	112.77
core	4	2	2	1	15.36	15.08	0.28	449.92
core	8	2	2	2	15.41	15.12	0.29	896.96
core	12	3	2	2	15.54	15.24	0.30	1334.57
socket	4	2	2	1	15.39	15.12	0.27	449.12
socket	8	2	2	2	15.39	15.11	0.28	898.30
socket	12	3	2	2	15.57	15.27	0.30	1331.88
node	12	3	2	2	15.46	15.18	0.28	1340.93
node	24	4	3	2	15.43	15.12	0.30	2688.60
node	48	4	4	3	15.87	15.56	0.30	5227.87

Analysis of the results

First of all an observation: in the case of only 1 processor there isn't any time spent in communicating therefore the time that we observe does not make sense. I think that it is caused by the fact that the code goes in any case trough the *do* loops and the *if* statements wasting some time. If this is true then we can consider that the same amount of time is required by the other processors to do the same things and so the effective time to communicate is what remains.

The values measured are reported in the following table together with the theoretical one computed using the equations above.

Mapped by	N	Communication [s] (measured)	Communication [s] (computed)	Speedup	P(L, N) [MB^3/s]
-	1	0	-	1.00	114.73
core	4	0.018	0.009	1.00	458.25
core	8	0.028	0.014	0.99	914.12
core	12	0.039	0.014	0.99	1360.92
socket	4	0.012	0.009	1.00	457.25
socket	8	0.02	0.014	1.00	915.04
socket	12	0.038	0.014	0.98	1357.99
node	12	0.023	0.006	0.99	1366.04
node	24	0.039	0.006	0.99	2741.97
node	48	0.042	0.006	0.97	5329.83

Influence of the topology

From the above tables we can see that the topology seems to have some effects not much on the communication time but on the computation time. Regarding the performance we get that basically T_c is negligible in respect to T_s ($T_c << T_s$) which we measure and remains constant troughout all the cases. This means that the different value of $P(L, N)$ in different cases is only dictacted by N at the denominator as it is noticable. Values of MLPUs and P(L, N) are similar for sure but in the calculation we are actually pluggin the measured value of T_s (the only one that counts): so we are a little bit cheating.

Difference between gpu and thin nodes

In the table below are reported the values obtained using GPU nodes.

Mapped by	N	N_x	N_y	N_z	Run [s]	Computation [s]	Communication [s]	MLPUs [MB^3/s]
-	1	1	1	1	22.08	21.73	0.35	78.25
core	4	2	2	1	22.26	21.89	0.37	310.47
core	8	2	2	2	23.30	22.79	0.51	593.30
socket	4	2	2	1	22.26	21.90	0.36	310.46
socket	8	2	2	2	22.43	22.06	0.37	616.20
node	12	3	2	2	22.31	21.94	0.37	929.27

We can notice that using GPU nodes performance are worse.

A note about strong scaling

I also tried to test the strong scaling scenario with the Jacobi code. In this case we give the same amount of work to an increasing number of processors so we expect that the time required to do the job decreases as the number of processors grows.

In particular we have the *Amdhal's law*:

$$T(P) = s + \frac{p}{P}$$

where:

- p is the time taken by the perfectly parallelizable fraction of the algorithm;
- $s = 1 - p$ is the time taken by the serial fraction that is in no way parallelizable;
- $T(P)$ is the time required by P processors to complete the task.

Therefore the speedup theoreticcaly obtainable using P processors is:

$$s(P) = \frac{1}{s + \frac{p}{P}}$$

this means that for $P \to \infty$ the speedup reach an asymptot at $1/s$. So, even if the parallel part speeds up perfectly, we are limited by the sequential portion of the code.

Hovever, the formulas reported before are obtained in an ideal case where we are neglecting the time required by the processors to communicate between each other. A more correct version of the equation for the time is:

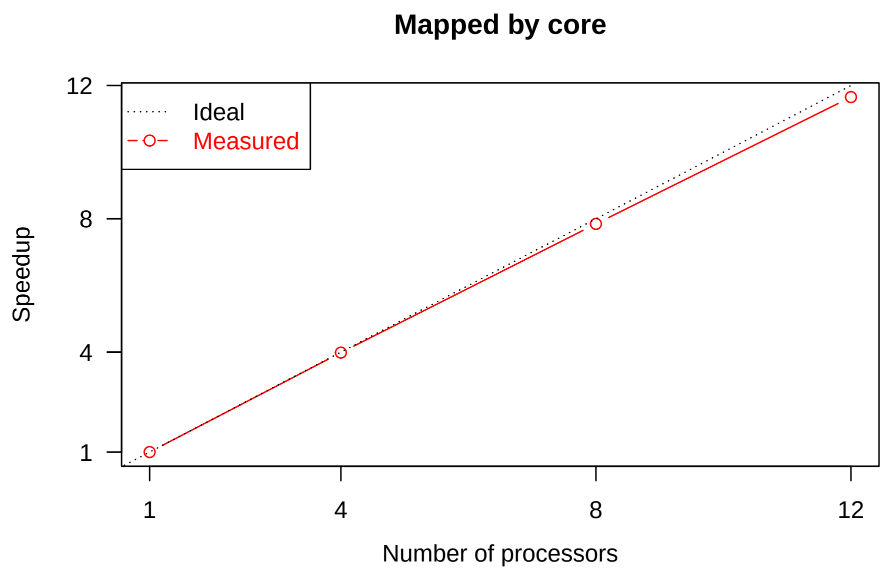
$$T(P) = s + \frac{p}{P} + c(P)$$

where $c(P)$ is the communication time when using P processor that may depend on:

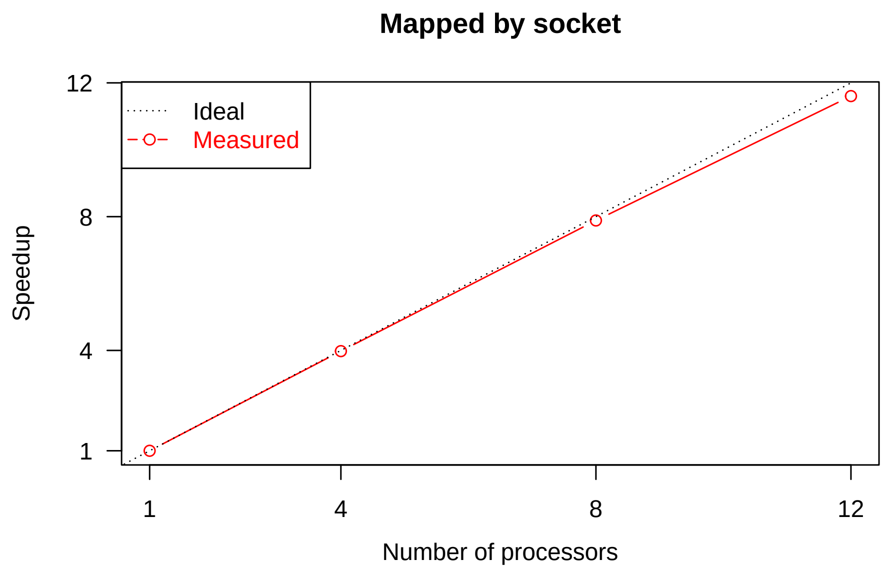
- network topology;
- communication pattern;
- message size.

In our case we consider s to be only the overhead for the communication, so the time taken to find the neighbors in the virtual topology and the other things reported in the code previously provided.

In the following plot is reported the speedup computed from the measured values obtained by feeding the Jacobi code always a 1200 x 1200 x 1200 matrix while increasing the number of processors (1, 4, 8, 12) and mapping them by core. It is possible to see that increasing the number of the processors worsen the behavior with respect to the ideal one as expected.



Now, to see if this the topology as some influence in this case below are reported the results obtained in the same case as the one just reported with the only difference of mapping the processors by socket.



At first glance there seems to be no difference, but let's look closely as the actual number obtained.

Processors	Run time by core [s]	Run time by socket[s]
4	3.82	3.83
8	1.94	1.93
12	1.31	1.31

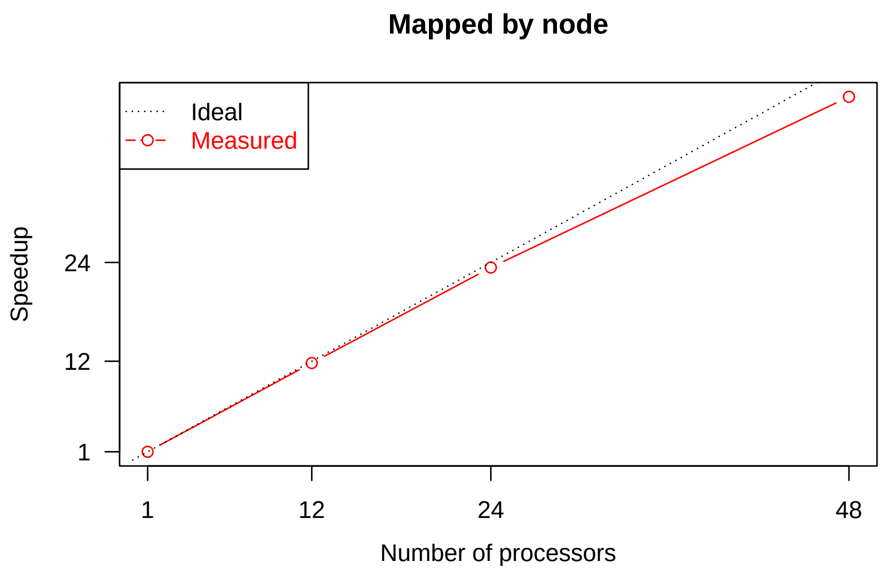
So, as I expect, in the case of 4 and 12 processors the speedup is worse when mapping on different sockets instead of mapping on the same socket but this is not true for 8 processor.

We can also compare these results with the ones obtained by mapping by node. However in this case the number of processors used was 12, 24 and 48 so we can compare only the case with 12 processors.

Mapped by	Run time with 12 processors [s]
Core	1.31
Socket	1.31
Node	1.29

Differently from what we expected the time decreases in the last one. A possible motivation for this behavior is that probably the number of processors (12) is still too small to have an non negligible contribution of the communication time in the total time required to complete the execution and so the differences that we're seeing in the different cases are just random noise and fluctuations.

To conclude this brief section I report below the plot with the measured obtained by using processors mapped by node. It is possible to observe the increased degradation of the speedup if we compare with the previous cases in accord with what is expected to obtain for $P \to \infty$.


Mapped by node
Ideal
Measured
Speedup
Number of processors