Assignment 1 on MPI

Foundation of High Performance Computing

Baurice Nafack

Lecturers: **Prof. Stefano Cozzini and Prof. Luca Tornatore**

Date Last Edited: December 28, 2021

1 MPI programming

The whole point of MPI (Message Passing Interface) is to make processes communicate. Two types of communication exist: point-to-point and collective. Point-to-point communication is divided into two operations: Send and Receive. Its most basic forms are called blocking communications. The process sending a message will wait until the process receiving has finished receiving all the information before it can continue its work. If not, the sending process will be in a deadlock. Some applications may be severely limited by this. Non-blocking communication is an alternative. Instead of waiting for the send (or the receive) to complete, the process will continue working, checking intermittently whether the communication has been completed.

As part of this project, we will write two MPI programs that are based on blocking and non-communication in order to study the performance and scalability of a simple ring topology of one-dimensional processors with p processors, where each processor has a left and right neighbor.

1.1 Point-to-point communication : ring topology

Description of the problem:

the program should implement a stream of messages in both directions:

- As the first step, P sends a message (msgleft=rank) to its left neighbor (P-1) and received it from its right neighbor (P+1), and sends another message(msgright= -right) to P+1 and received from P-1.
- It them do enough iterations till all the processors receive back the initial message. At each iteration, each processor and its rank to receive messages originating from certain, processor P should have a tag proportional to its rank (i.e tag = P*10). Provide the code that should print on a file the following output : I am process irank and I have received np massages. My final message has tag and value msg-left, msg-right .

  where irank,np,itag and msg and are parameters that should be printed by the program. Make sure that your code produces the correct answer, independent of the number of processes you use. Take the time of the program using MPI_walltime routines and produce a plot of the runtime as a function of P. If the total time if very small, you might need to repeat several times the iterations in order to get reasonable measurements.

1.1.1 Sending message in blocking communication

A send operation, sends a buffer of data of a certain type to another process with the following properties :

- A reference to a buffer: the reference is a pointer to a buffer. This array will hold the data that you wish to send from the current process to another.
- A datatype : the datatype must correspond precisely to the data stored in the buffer.
- A number of elements: this is the number of elements in the buffer that we want to send to the destination.
- A tag: The tag is a simple integer that identifies the type of communication. This is a completely informal value that we put randomly.
- A destination id: The rank of the process we want to send the data or message to.
- A communicator : The communicator on which to send the data to.

1.1.2 Receiving message in blocking communication

Messages are received in exactly the same way as they are sent. Instead of a destination id, we will need a source id: the identifier of the process from which we are waiting for a message. On top of that, depending on if we are using blocking or non-blocking communications, we will need additional arguments.

1.1.3 Non-Blocking send and receive

The send and receive functions will return immediately, giving us more control of the flow of the program. The sending or receiving buffers cannot be modified after they are called, but the program is free to continue with other operations. MPI_Wait and MPI_Test must be used when the buffers need to be filled with data. There is a new parameter here, a request. This is used to track each separate transfer made by the program. It can be used to check the status of a transfer using the MPI_Test function or to wait until it is complete by calling MPI_Wait.

1.1.4 MPI Point-to-Point: Deadlock

Deadlock is a common problem in parallel processing. This occurs when two or more processes are competing for the same set of resources. A typical scenario in communication involves two processes wishing to exchange messages: each is trying to send a message to the other, but neither is ready to accept a message, figure [1]. We wrote our ring program by considering the possibility of deadlock. To protect our program from deadlock, we use different call ordering between tasks. This entailed having one task post its receive first and the other post its send first. We also used a non-blocking send in order to solve this problem.

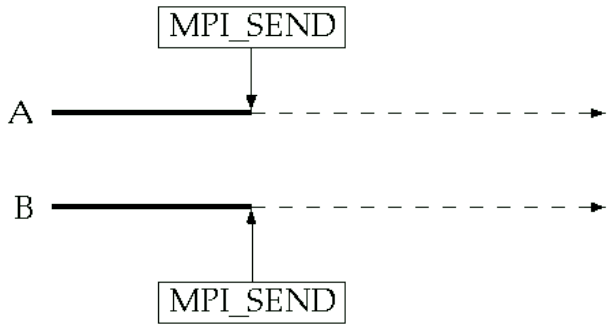

Figure 1: Deadlock illustration. We can see that they both call standard blocking sends at the same point in the program, but the receive that corresponds to each task's send occurs later in the other task's program.

1.1.5 Experimental result : Model network performance

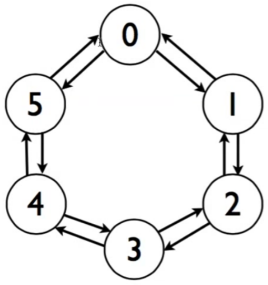

Figure 2: Ring topology Illustration

1.1.6 Methodology

To investigate model network performance, we used 1500 and 2500 iterations to measure run times as a function of processor count. An iteration ends when each process receives its initial message. We use MPIWtime() to measure each processor's run time for a single iteration, then record processor means time. We repeated the operation as many times as possible for P processors because, according to the law of large

numbers, the average of the results obtained from a large number of trials should be close to the expected value and will tend to become closer as more trials are performed.

1.1.7 result

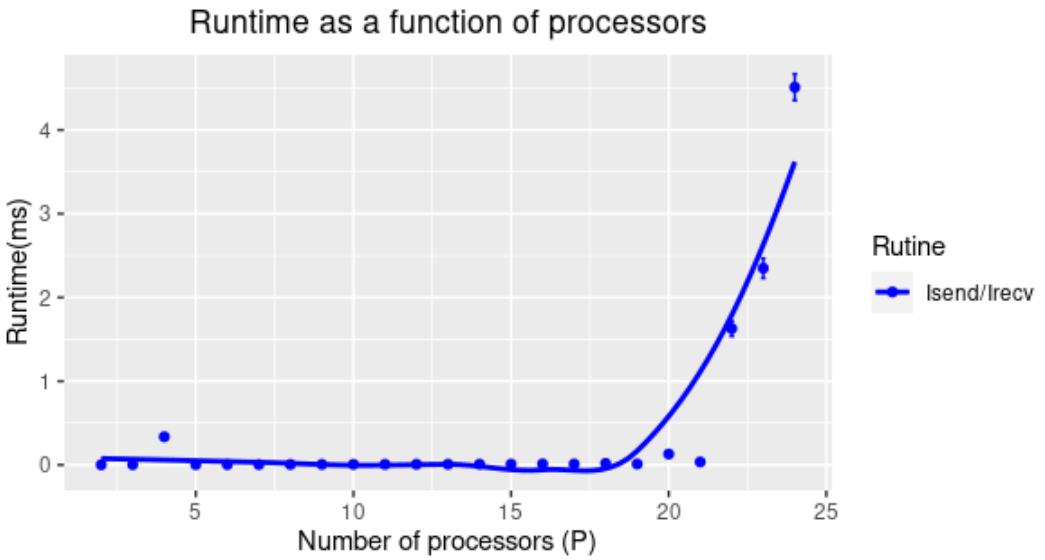

Figure 3: Mean execution times for non-blocking communication with 1500 iteration.

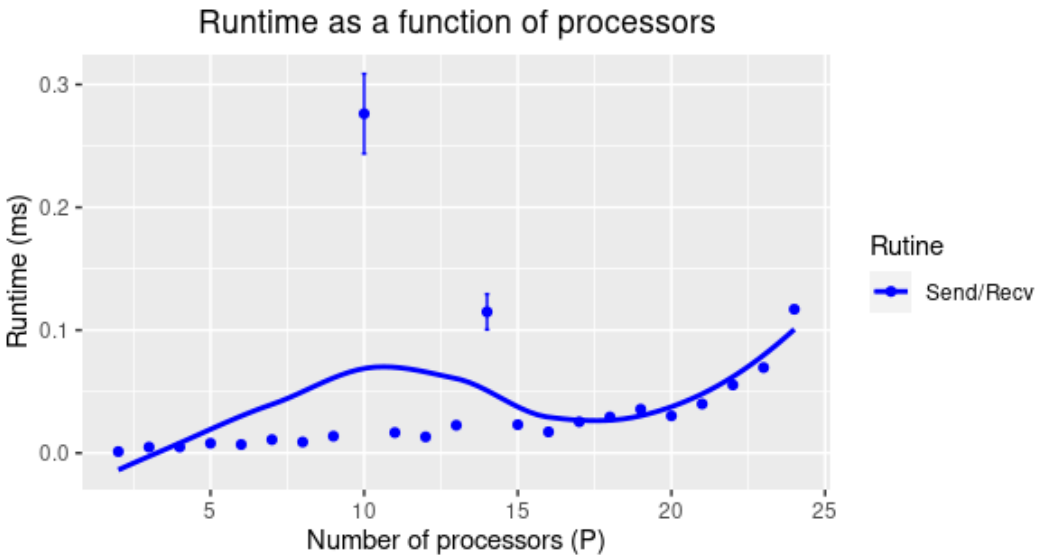

Figure 4: Mean execution times for blocking communication with 1500 iteration.

1.1.8 result Interpretation

In non-blocking communication, the running times started to increase quadratically when more than 22 of processors were used. As the number of processors increases, the running times for blocking communication also increase.

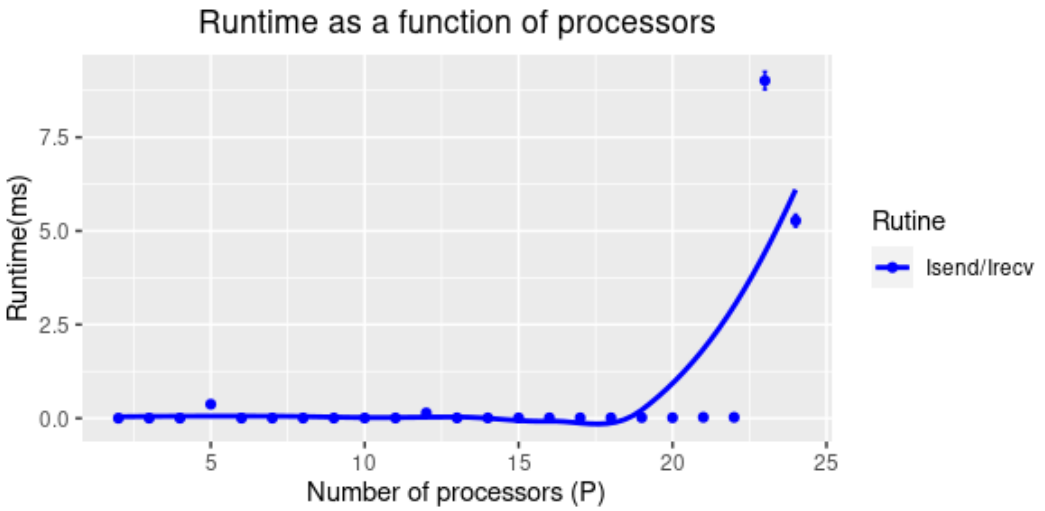

Figure 5: Mean execution times for non-blocking communication with 2500 iteration.

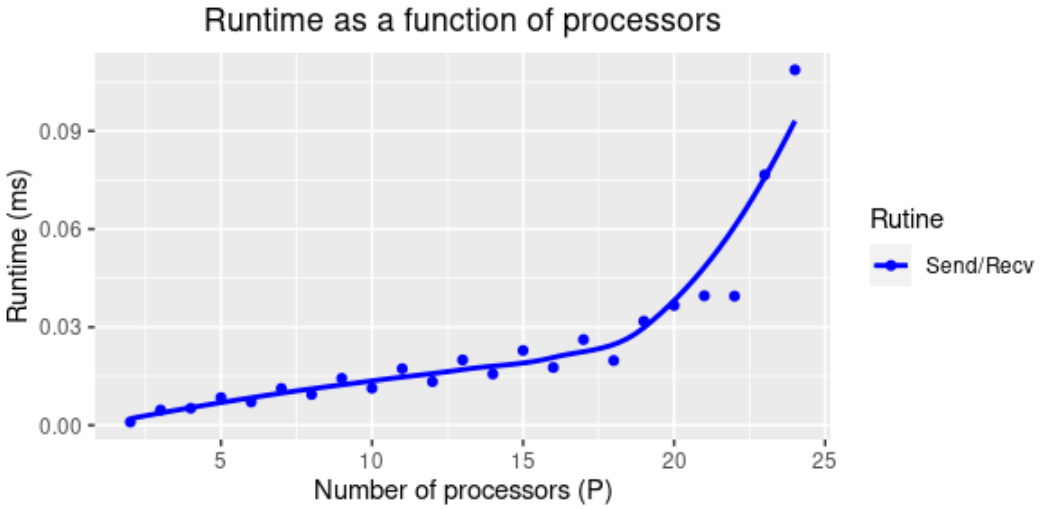

Figure 6: Mean execution times for non-blocking communication with 2500 iteration.

1.1.9 Perspective

In this study, processors could be bound by or on the same nodes, by socket, or on the same socket, and then the results could be compared. Additionally, we can try to understand the significance of the outlier observed on the graph.

1.2 Collective Communication : 3d matrix-matrix addition in parallel

Collective communication is defined as communication that involves a group of processes. Some of the functions of this type provided by MPI are the following:

- Barrier synchronization across all group members.
- Broadcast from one member to all members of a group.
- Gather data from all group members to one member.

- Scatter data from one member to all members of a group.
- Scatter/Gather data from all members to all members of a group.
- A combined reduction and scatter operation

In this section, we will use some of these functions for $3D$ matrix addition.

Description of the problem

implement a simple 3d matrix-matrix addition in parallel using a 1D,2D and 3D distribution of data using virtual topology and study its scalability in term of communication within a single THIN node. Use here just collective operations to communicate among MPI processes. Program should accept as input the sizes of the matrixes and should then allocate the matrix and initialize them using double precision random numbers.

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:

- $2400 \times 100 \times 100$;
- $1200 \times 200 \times 100$;
- $800 \times 300 \times 100$;

1.2.1 Methodology

We used a Single Program Multi Data (SPMD) to add two $3D$ matrices . For each process, we used the same seed to initialize two matrices. As a result, each process has a duplicate matrices. We generated data with n rows and $m \times m$ columns because a $3D$ matrix is a set of $2D$ matrix. For the parallel computation, we used the following step for $1D$ toplogy:

- We divide the number of columns by the size of the collective communication to determine the number of iterations (iter) that each process must complete. Process P with rank k began adding from column iter$\times k$ and ended at iter$\times k$+iter.
- The gather routine was used to collect a single row addition performed by each process. If any operations were left unfinished, they were completed by the master process. We use the Barrier routine to force each process to wait until the master process has completed those operations.
- We repeated this operations m times.
- using MPI_Wtimes() routine, we record the run time.

1.2.2 Result

The different run times for different matrix dimension are display on table.

3D matrix	Times for 1D topology (s)
$2400 \times 100 \times 100$	1.41
$1200 \times 200 \times 100$	1.40
$800 \times 300 \times 100$	1.42

2 Measure MPI point to point performance: benchmark

Benchmark is a standard workloads used to compare performance across machines. the elementary patterns of the benchmarks are message lengths, sample repetition counts, timer, synchronization, number of processes and communicator management, and display of results. For a clear structuring of the set of benchmarks, IMB introduces classes of benchmarks:

- Single Transfer: The benchmarks in this class are to focus on a single message transferred between two processes (i.e PingPong and PingPing).
- Parallel Transfer benchmarks : Benchmarks focusing on global mode, say, patterns. The activity at a certain process is in concurrency with other processes, the benchmark measures message passing efficiency under global load.
- Collective benchmarks: This class contains all benchmarks that are collective in proper MPI convention. Not only is the message passing power of the system relevant here, but also the quality of the implementation.

For this study, we will use only the single transfer benchmarks with PingPong figure [7]. The latency tests are carried out in a ping-pong fashion. The sender sends a message with a certain data size to the receiver and waits for a reply from the receiver. The receiver receives the message from the sender and sends back a reply with the same data size. Many iterations of this ping-pong test are carried out and average one-way latency numbers are obtained. The benchmark reports latency (in microseconds), bandwidth (in million bytes per second).

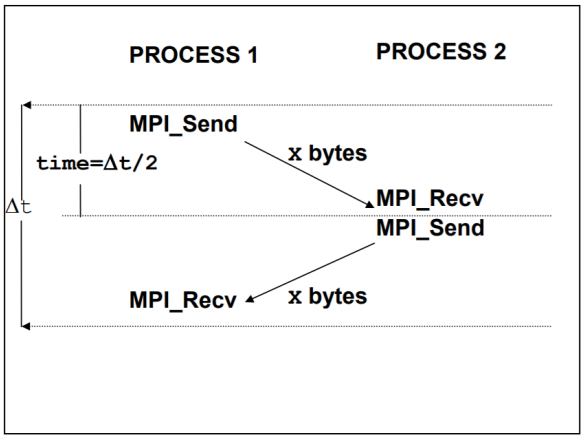

Figure 7: PingPong pattern

2.1 Description of the problem

Use the Intel MPI benchmark to estimate latency and bandwidth of all available combinations of topologies and networks on ORFEO computational nodes. Use both IntelMPI and openmpi latest version libraries available on ORFEO. Report numbers/graph and also fit data obtained against the simple communication model we discussed. Compare estimated latency and bandwidth parameters against the one provided by a least-square fitting model. Provide a csv file for each measure with the following format and provide a graph (pdf and/or jpeg or any image format you like)

2.2 Methodology

To get the benckmark, we used the following step :

- we get the intel benchmark using : git pull https://github.com/intel/mpi-benchmarks
- We enter in the follow directory: cd mpi-benchmarks/src_c
- we upload the lastest version of openmpi suing : module load openmpi-4.1.1+gnu-9.3.0
- We compile the code using: make
- We run the executable across two nodes using: mpirun -np 2 –map-by node –report-bindings ./IMB-MPI1 PingPong, then extrat the repport as csv file.
- to run run the executable across two socket we used : mpirun -np 2 –report-bindings –map-by socket ./IMB-MPI1 PingPong
- to run the executable using a different protocol: tcp and not infiniband, we used :

  mpirun -np 2 –mca pml ob1 –mca btl tcp,self –mca btl_tcp_if_include br0 ./IMB-MPI1 PingPong

To run the intel benchmark, we use the following step

- We used make clean to clean the compile programs.
- We used module purge to delete all module load.
- We used module load intel
- We used make to compile the program.

2.3 Experimental Result

2.3.1 IntelMPI Benchmarck

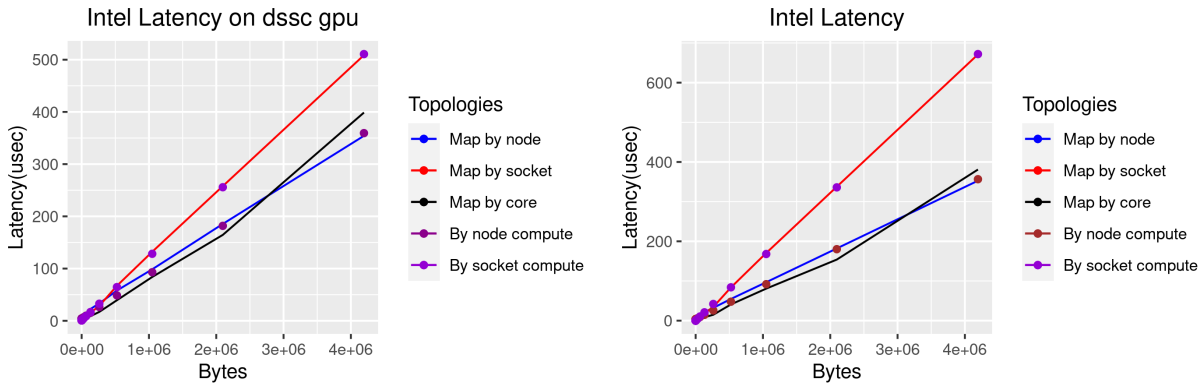

Figure 8: Intel latency on dssc gpu and dssc node on ORFEO

2.3.2 OpenMPI

Interpretation

The plot indicates that dssc and dssc gpu behave the same. Node-level mapping has a bandwidth that goes up and remains relatively constant at some point.

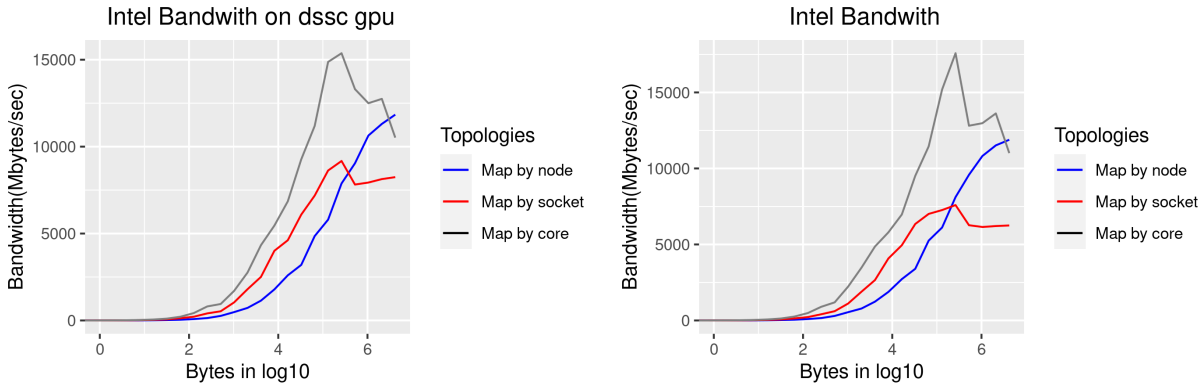

Figure 9: Intel bandwidth on dssc gpu and dssc node on ORFEO

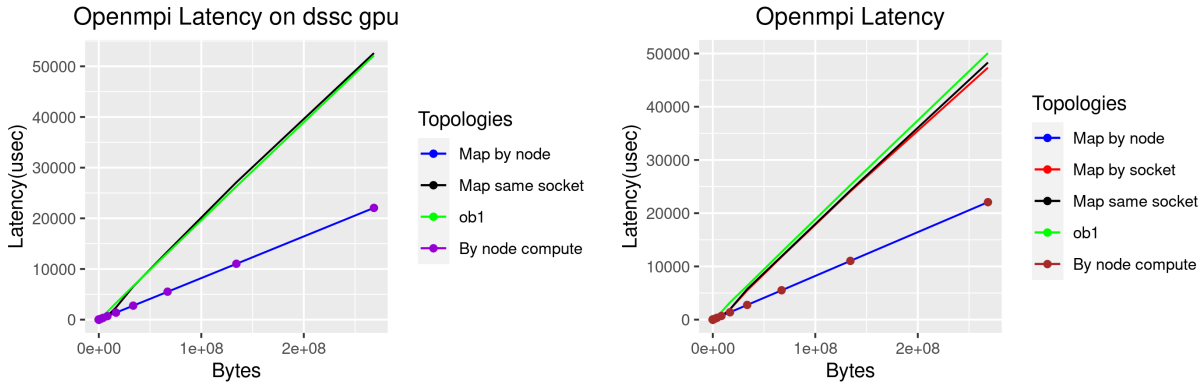

Figure 10: Openmpi latency on dssc gpu and dssc node on ORFEO

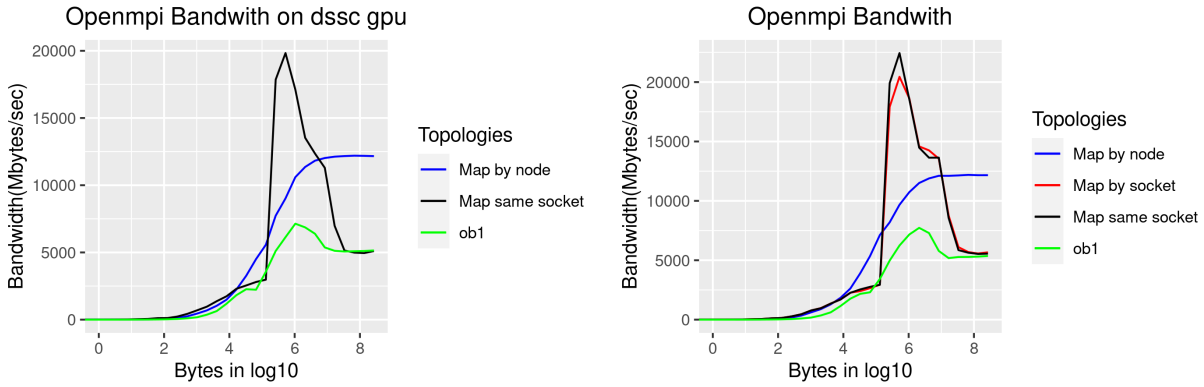

Figure 11: Openmpi bandwidth on dssc gpu and dssc node on ORFEO

2.4 Fitting model

To find the fitting bandwith and latency, we use the relation below:

$$T_{\text{Comm}} = \lambda + \text{message-size}/b_{\text{network}}, \tag{1}$$

where λ is the fitting lattency. We use R to fit a linear model using lm routine. The interpretation of some maps does not make sense since the time can not be negative. Thus, we consider only the mapping

with a positive intercept to determine the fitting latency and bandwidth.See table 2.

Table 1: Benchmark

Mapping	Estimated Latency (μs)	Fitting Lattency (μs)	Fitting bandwith (Mbytes/s)
by node openmp on dssc	0.99	3.79	11885
By node openmp on dssc gpu	1.36	4.614	12173
By node intel dssc	1.12	3.7	11885
By node intel dssc gpu	1.35	4.23	11811
By socket intel dssc	0.48	0.026	62421.97
By socket intel dssc gpu	0.48	0.81	10936

From the table 2, we observe that mapping the process by socket using Intel MPI have a high bandwidth network with low estimate latency.

3 Compare Performance Observed Against Performance Model for Jacobi Solver

Description

The application we are using here is a Jacobi application already discussed in class.
Steps to do:

- Compile and run the code on on single processor of a THIN and GPU node to estimate the serial time on one single core.

- Run it on 4/8/12 processes within the same node pinning the MPI processes within the same socket and across two sockets
  - Identify for the two cases the right latency and bandwith to use in the performance model.
  - Report and check check if scalability fits the expected behaviour of the model discussed in class.
- Run it on $12, 24$ and 48 processor using two thin nodes
  - Identify for this case the right latency and bandwith to use in the performance model.
  - Report and check check if scalability fits the expected behaviour of the model discussed in class.
- Repeat the previous experiment on a GPU node where hyperthreading is enabled.
  - Identify for this case the right latency and bandwith to use in the performance model.
  - Report and check check if scalability fits the expected behaviour of the model discussed in class.

3.1 Result

- Serial time on one single core
  - THIN: $29.07s$
  - GPU: $28.38s$
- Latency and Bandwith using $4, 8$ and 12 processes.

Table 2: Latency and Bandwith

Mapping by Node and Socket	Latency on dssc (s)	Lattency on gpu (s)	Bandwith on dssc (Mb/s)	Bandwith on gpu (Mb/s)
4 processors	3.83	5.58	450.71	309.2
8 processors	1.93	2.88	892.7030	598.5
12 processors	1.29	1.947	1332.50	887.20
two nodes 12 processors		1.88		917.5
two nodes 24 processors		0.96		62421.97