Final report

Section 1

MPI programming

Abstract

This section is about the implementation of analysis of MPI codes.

- Ring
- Matrix

1.1. Ring

A ring is a simple 1D topology where each processor or node has a left and a right neighbour.

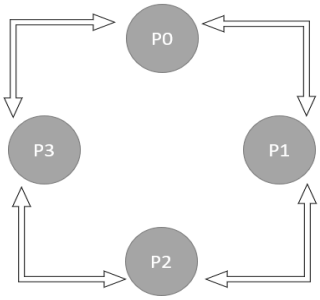


Figure 1: ring example with 4 processors

1.1.1 Objective

The ring code implements a stream of messages in both directions, the objective is to iterates the stream message between nodes until each processor receives again the initial message. In each step a processor sends two messages, one to the left and another to the right and receives two messages, one from the right and another one from the left. The message has the following:

- in the first step each processor sends its rank value to the left and its rank value to the right
- at level of every single iteration each processor subtracts it's rank to the received message if it comes from left, adding if it comes from right

The final aim is that every processor has the same value of the two received message. For example, in the case of 4 processor each processor receives from the left and right the values, respectively, +6 and -6.

1.1.2. Coding

The ring code use the open MPI library to perform parallel process, in the case of implementation were used two functions for sending and receiving messages send and receive:

- **Send**: performs a blocking send that terminates when the message is received by the recipient.
- **Recv**: performs a blocking receive that terminates when a message is received by the caller.

Ring.c: snippet

```
public main ( ){
   MPI_Init(&argc , &argv) ;

   left = (rank + size − 1) % size;
   right = (rank + 1) % size;
   msgleft = rank;
   msgrigth = rank − 2*rank;

   for ( i = 0; i < size ; i ++) {
      MPI_Send(&msgleft , 1 , MPI_INT , left , tag , MPI_COMM_WORLD) ;
      MPI_Recv(&msgleft, 1 , MPI_INT , right , MPI_ANY_TAG, MPI_COMM_WORLD, &status ) ;
      i f ( s i z e − 1 != i ) {
        ms g l e f t = ms g l e f t + r a n k ;
      }

      MPI_Send(&msgright , 1 , MPI_INT , right , ag , MPI_COMM_WORLD) ;
      MPI_Recv(&msgright , 1 , MPI_INT , left , MPI_ANY_TAG, MPI_COMM_WORLD, &status ) ;
      if ( size − 1 != i ) {
       msgright = msgright − rank ;
      }
   }
   MPI _Finalize ( );
   return 0;
}
```

1.1.3. Examination

The computational examination was done on Orfeo, which is a super-computer, with a number of processor and iterations (100) on the same number of processor.

The slowest processor time was taken for each of the 100 iterations. The time was obtained by the functions available of OpenMPI:

- **Wtime:** is a function that returns the wall time from an arbitrary time in the past. For the test a MPIWtime was put before the main work and after the main work and then a difference was computed.

- **Reduce:** is a function that applies a reduction calculation to all processors. In this case the maximum time.

- **Barrier:** is a function that synchronizes all processors. This function has been used to wait all the processor after the last iteration. In addition to the measured time, two performance indices were used for measuring the scalability of the implementation:

- **Speedup:** the speed up index is the ratio between the best sequential algorithm to solve the problem to the time required by parallel algorithm using p processor to solve the same problem.

$$S(P) = T(1)/T(P)$$

- **Efficiency:** the average contribution to speedup by each processor working in parallel.

$$E(P) = S(P)/P$$

N° Processor	1	2	4	8	16	**32**
Average Time	0,00128	0,00210	0,00235	0,00272	0,00396	0,00554
S(P) = T(1) / T(P)	1	0,60799	0,54331	0,46906	0,32276	0,23090
E(P) = S(P) / P	1	0,30399	0,13582	0,05863	0,02017	0,00721

Table 1: Computational average time of 100 iterations for different numbers of processor, speedup and efficiency indices

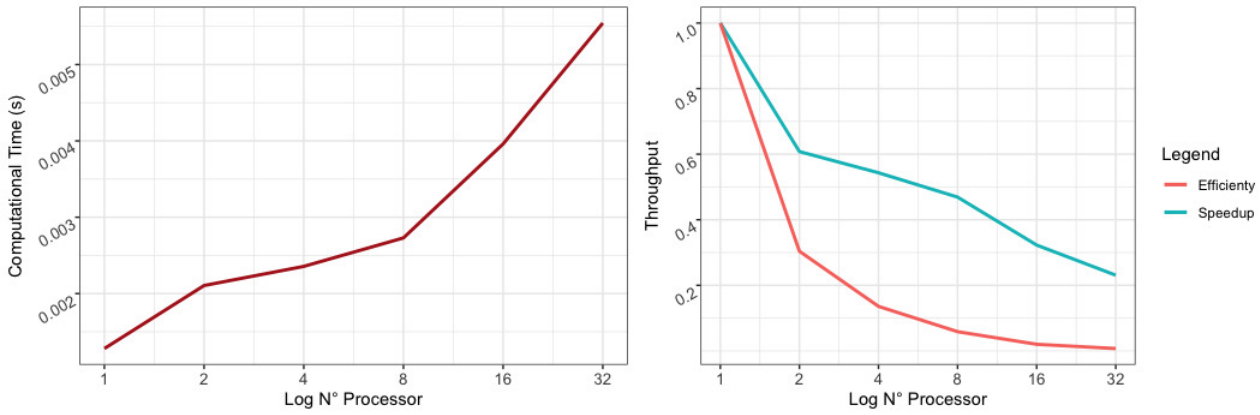


Figure 2: on the left Log N° Processor vs Computational Time (s), on the right the comparison between speedup and efficiency

In the left graph the result obtained show how the computational time grows linearly as the number of processors increases. In the right graph the result obtained show how the speedup and the efficiency decrease as the number of processors increases. It means that we are in a condition of weak scalability in which the time increases by increasing the number of processors.

1.2. Matrix

It's expected the implementation of a 3d matrix-matrix addition in parallel using collective operations to communicate among MPI processors.

1.2.1. Objective

The 3d matrix-matrix code implements and addition between two 3D matrices. The aim is to divide the matrices into small matrices, and then each processor computes the sum between two small matrices. At the end when all the processors computes the sum among all the small matrices, the small matrices are merged to create the final matrix.

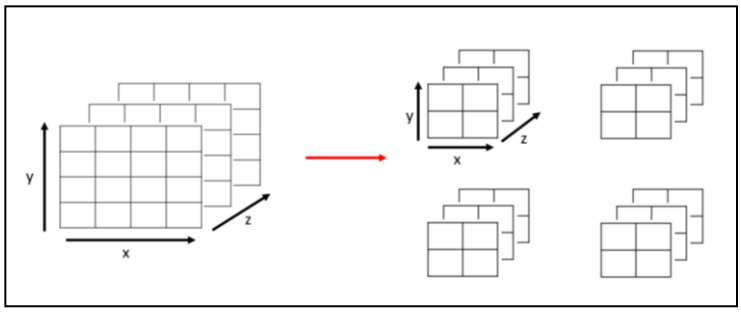


Figure 3: process of splitting a matrix [4 * 4 * 3] in 4 small matrices[2 * 2 * 3]

1.2.2. Coding

The matrix code uses the Open MPI library for performing parallel processing. To implement it, were used two different functions. One for dividing the matrix in small matrices and the other for merging the small matrices in the big matrix:

- Scatter: divides a big array into a number of smaller parts equal to the number of processes and sends each process a piece of the array in rank order.

- Gather: receives data stored in small arrays from all the processes and concatenates it in the receive array in rank order.

Sum3Dmatrix: snippet

```
public main ( ) {
   MPI _Init (&argc, &argv) ;

   int bigMatrixElements = x * y * z ;
   int smallMatrixElements= smallX * smallY* smallZ ;
   double * bigMatrix1 = new double [bigMatrixElements] ;
   double * smallMatrix1 = new double [ smallMatrixElements] ;

   MPI _Scatter (bigMatrix1, smallMatrixElements, MPI_DOUBLE, smallMatrix1 , ...) ;
   MPI _Scatter( bigMatrix1 , smallMatrixElements , MPI_DOUBLE, smallMatrix1 , ...) ;

   for ( int i = 0 ; i < smallX ; i ++) {
      for ( int j = 0 ; j < smallY ; j ++) {
         for ( int k = 0 ; k < smallZ ; k++) {
            *( smallMatrix3 + i * smallY * smallZ + j * smallZ + k ) =
            *( smallMatrix1 + i * smallY * smallZ + j * smallZ + k ) +
            *( smallMatrix2 + i * smallY * smallZ + j * smallZ + k )
         }
      }
   }

   MPI_Gather (smallMatrix3 , smallMatrixElements , MPI_DOUBLE, bigMatrix3 , . . . ) ;

   MPI _Finalize( ) ;
   return 0;
}
```

1.2.3. Examination

The computational examination was done on Orfeo with a stuck number of processors on a different distributions (1D, 2D, 3D) on a single thin node for 3 different case (2400 · 100 · 100, 1200 · 200 · 100, 800 · 300 · 100). For each case there are different splitting solution, the results obtained are shown below:

[2400 · 100 · 100]	Split	Average	Minimum	Maximum
1D	[100 · 100 · 100]	0.111346	0.110219	0.111555
2D	[200 · 50 · 100]	0.129085	0.129024	0.129135
	[400 · 25 · 100]	0.106082	0.106029	0.106145
3D	[400 · 50 · 50]	0.107354	0.106702	0.107988
	[800 · 25 · 50]	0.108555	0.108506	0.1086

Table 2: results for the 2400 · 100 · 100 case

1200 · 200 · 100	Split	Average	Minimum	Maximum
1D	[50 · 100 · 100]	0.0707909	0.0707541	0.070834
2D	[100 · 100 · 100]	0.107559	0.107592	0.108833
	[200 · 50 · 100]	0.108683	0.107788	0.109028
	[200 · 200 · 25]	0.108912	0.108835	0.108971
	[100 · 200 · 50]	0.108938	0.107856	0.109147
	[400 · 25 · 100]	0.109706	0.10964	0.109756
3D	[200 · 100 · 50]	0.108236	0.107287	0.108592
	[400 · 50 · 50]	0.119639	0.118698	0.120001

Table 3: results for the 1200・200・100 case

800 · 300 · 100	Split	Average	Minimum	Maximum
2D	[100 · 100 · 100]	0.108007	0.107968	0.108054
	[200 · 50 · 100]	0.10789	0.106991	0.108232
	[400 · 25 · 100]	0.105265	0.105194	0.105332
	[800 · 25 · 50]	0.108368	0.108318	0.108422
	[800 · 50 · 25]	0.108864	0.108833	0.108903
3D	[200 · 100 · 50]	0.106518	0.106472	0.106568
	[400 · 100 · 25]	0.108557	0.108503	0.108621
	[400 · 50 · 50]	0.108633	0.10857	0.108691

Table 4: results for the 800 ・ 300 ・ 100 case

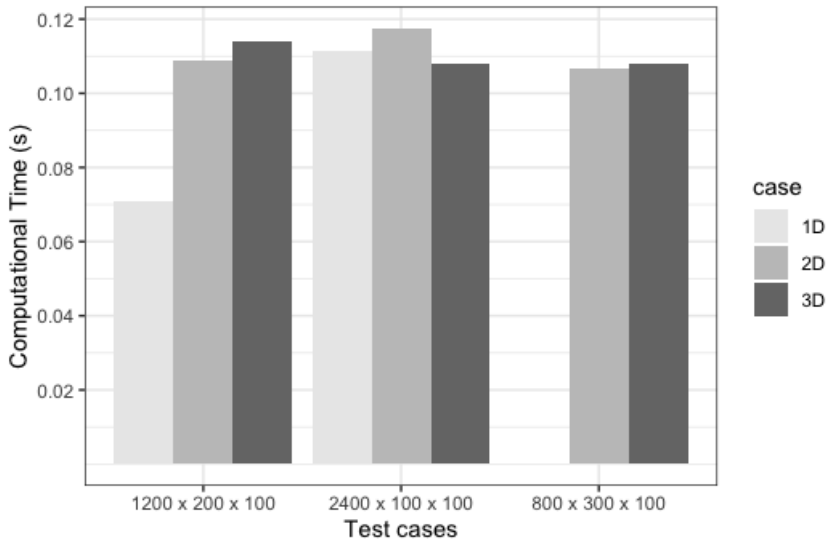


Figure 4: computational time on the different case on different dimension (1D, 2D, 3D)
In the graph above the result obtained show how the computational time, using 24 cores, is similar in the case of 2D and 3D decomposition of the matrix for each case.

In the 1D decomposition, in the first case the computational time is approximately 2/3 of the other cases. In the second case is similar to the other cases and in the last case there is not a possible decomposition.

Section 2

MPI Point to Point Performance

Abstract

The following section will explain the estimation of latency and bandwidth from the topologies and networks on Orfeo, the following was done by: OpenMPI and IntelMPI.

2.1. Objective

Applying the use of OpenMPI and IntelMPI compilers the expected is to obtain latency and bandwidth from the topologies and networks on Orfeo, afterwards compare the estimated latency and bandwidth values against the fitting model least-square.

2.1.1. The Model

The model used to fit the data was a polynomial regression with different degree. By using R, the model was created, and it predicts the latency and the bandwidth on different runs. The full representation of the commands and modules are following:

Mapping - OpenMPI	PML	BTL	File Name
No Mapping	-	-	openmpi-nomapping.csv
Socket - Node - Core	-	-	openmpi-{S-N-C}.csv
Socket - Node - Core	OB1	-	openmpi-{S-N-C}-ob1.csv
Socket - Node - Core	OB1	SELF	openmpi-{S-N-C}-ob1-self.csv
Socket - Node - Core	UCX	SELF	openmpi-{S-N-C}-ucx-self.csv
Socket - Node - Core	UCX	VADER	openmpi-{S-N-C}-ucx-vader.csv

Table 1: runs with OpenMPI module

Mapping	- IntelMPI File Name
no mapping	-
socket	intelmpi-S.csv
node	intelmpi-N.csv
core	intelmpi-C.csv
[contiguous processors]	intelmpi-contiguousprocessors.csv
[same socket]	intelmpi-samesocket.csv

Table 2: runs with IntelMPI module

2.1.2 Evaluation

After a couple of iterations, the analysis obtained made by OpenMPI and IntelMPI modules with different mapping and different protocol.

Command	Socket		Node		Core	
	latency	bandwidth	latency	bandwidth	latency	bandwidth
-	0.39	14125	0.21	13270	0.2	13776
ob1	0.53	10284	0.26	9685	0.25	9724
ob1, self	7.97	3477	5.39	7037	5.43	7000
ucx, self	0.41	14058	0.2	13656	0.2	13771
ucx, vader	0.4	14277	0.2	13646	0.2	13083
intel	0.43	6341	0.42	6439	0.43	6336

Table 3: latency and bandwidth by different mapping

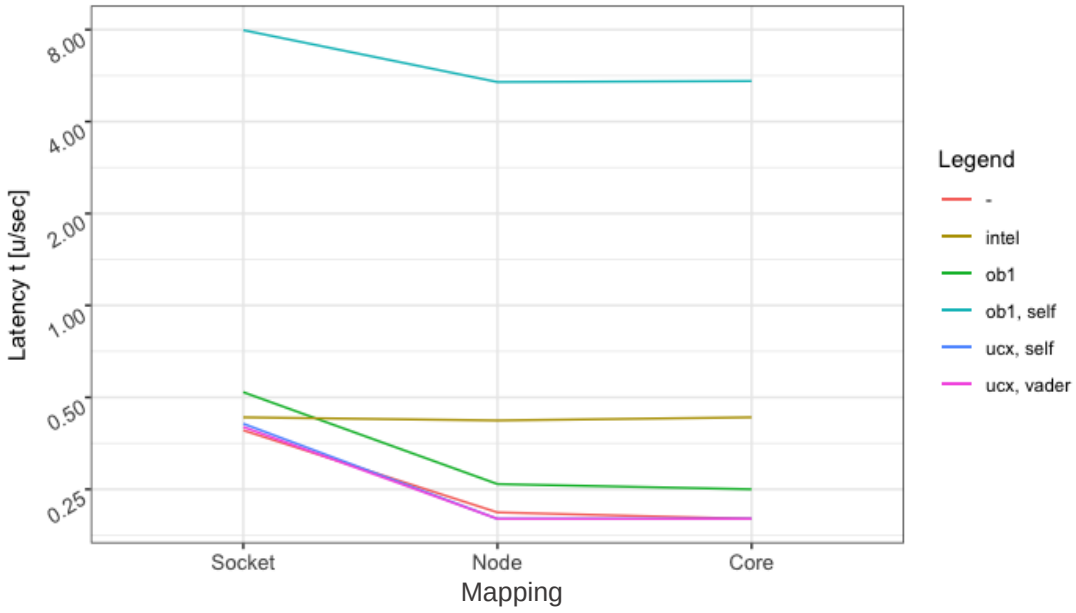


Figure 1: latency by different type of mpl and btl on different mapping

To examinate the latency it observed a similarity between the mapping, from the graph it's certain when mapping by socket the value of the latency is close to the double compared to mapping by node and core.

When PML is equals to ob1 the latency increases by 20% and when PML is equals to ob1 and BTL is equals to self the latency grows a lot. TCP protocol has a high latency.

In the case of PML equals to UCX the latency is similar on different mapping approaches regardless of the BTL being self or vader.

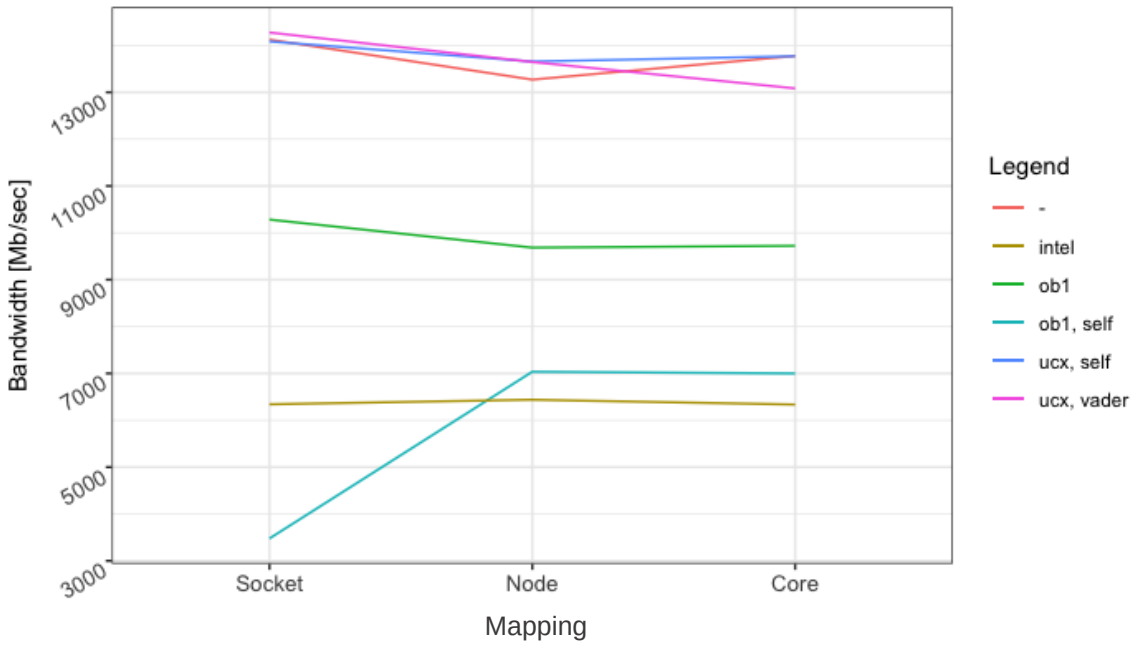
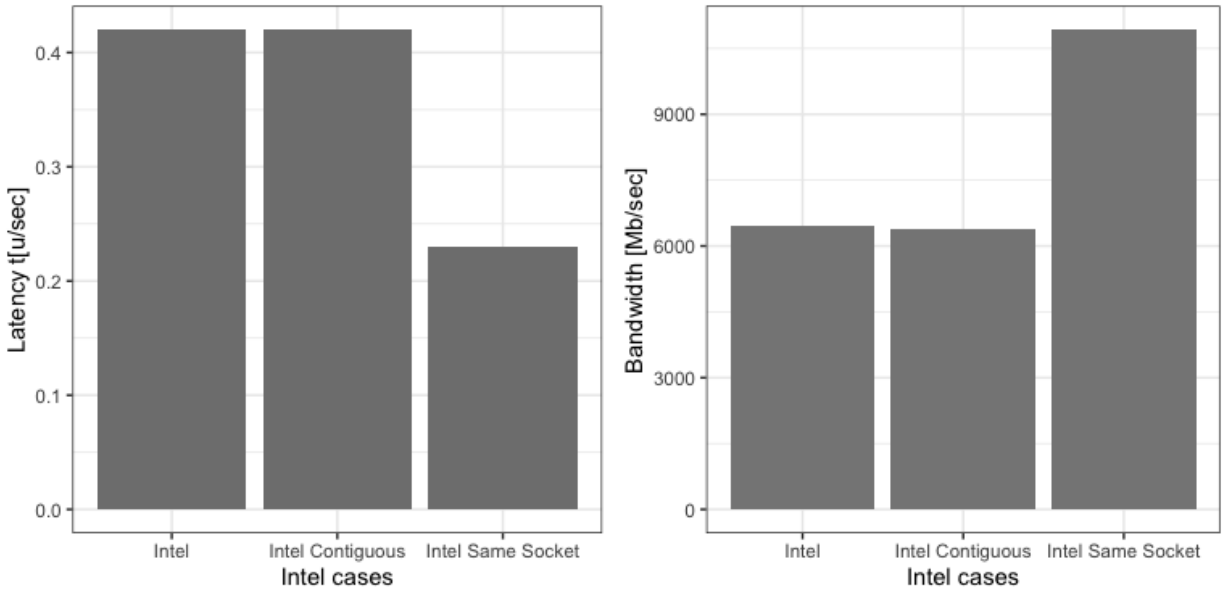


Figure 2: bandwidth of different type of mpl and btl on different mapping policies

From the graph it is show that when mapping by socket the value of the bandwidth is very low, approx 3500 Mb/sec. In the other cases the bandwidth is almost constant. It is similar when we are in the case of PML equals to ucs regardless of the BTL being self or vader and also in the case in which the PML and BTL is not specified. When we use IntelMPI the bandwidth is constant but smaller

	Intel	**Intel on 2 contiguous processors**	**Intel on the same socket**
Latency	0.42	0.42	0.23
Bandwidth	6445	6337	10926

Table 4: latency and bandwidth with IntelMPI module



From the graph of the latency and bandwidth analysis of the Intel runs is show that when mapping on two processors in the same socket the latency decrease by half of the normal and contiguous case. Consequently the bandwidth in the case of two processors in the same socket is doubled compared to the other cases.

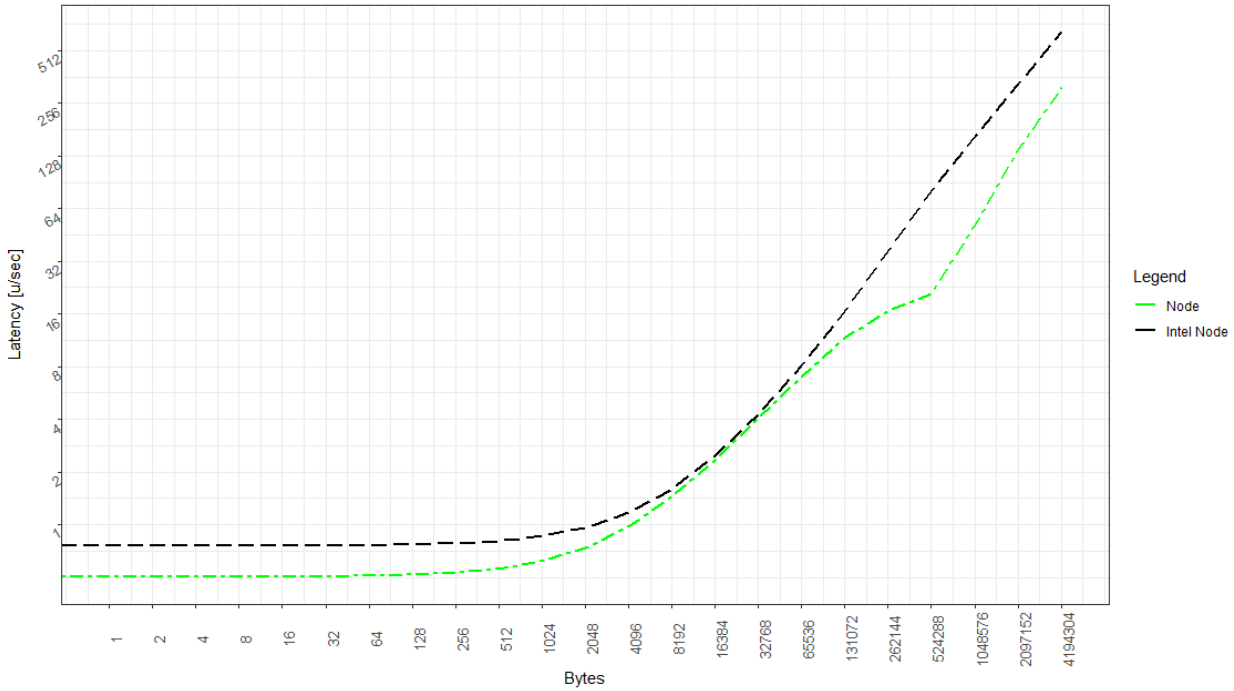


Figure 4: fitted model of latency computed for mapping by node comparing GCC and INTEL modules

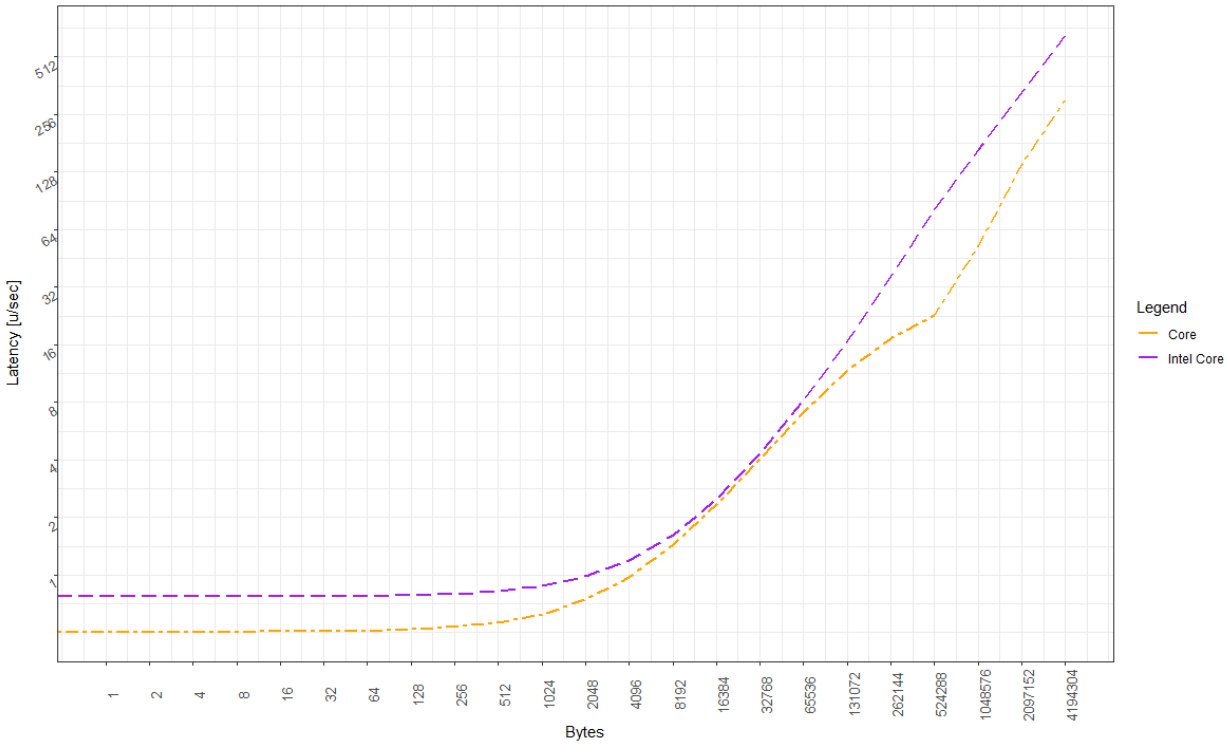


Figure 5: fitted model of latency computed for mapping by core comparing GCC and INTEL modules

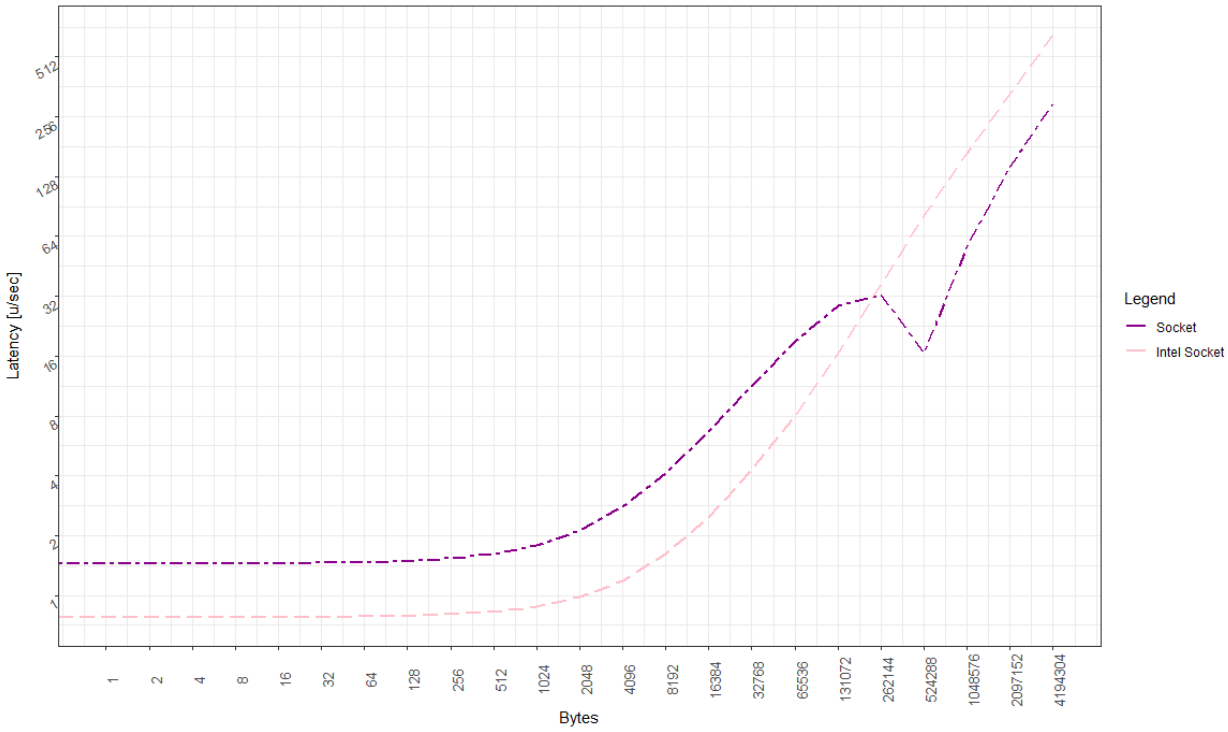


Figure 6: fitted model of latency computed for mapping by socket comparing GCC and INTEL modules

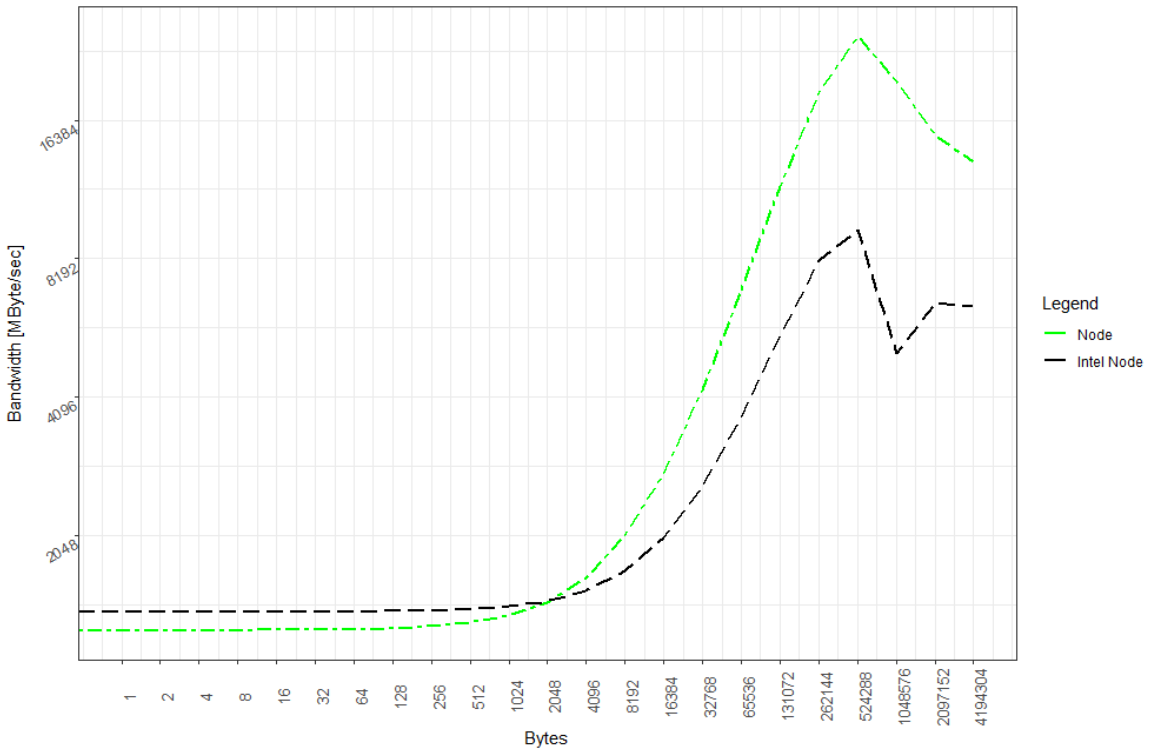


Figure 7: fitted model of Bandwidth computed for mapping by node comparing GCC and INTEL modules

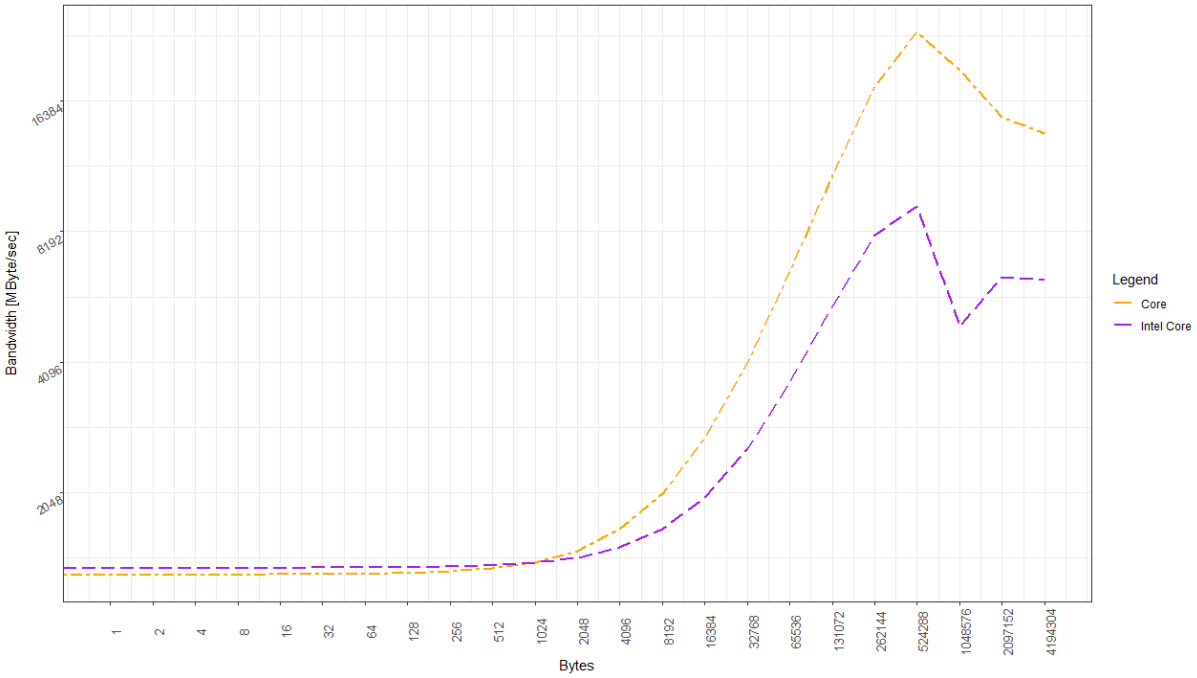


Figure 8: Fitted model of Bandwidth computed for mapping by core comparing GCC and INTEL modules

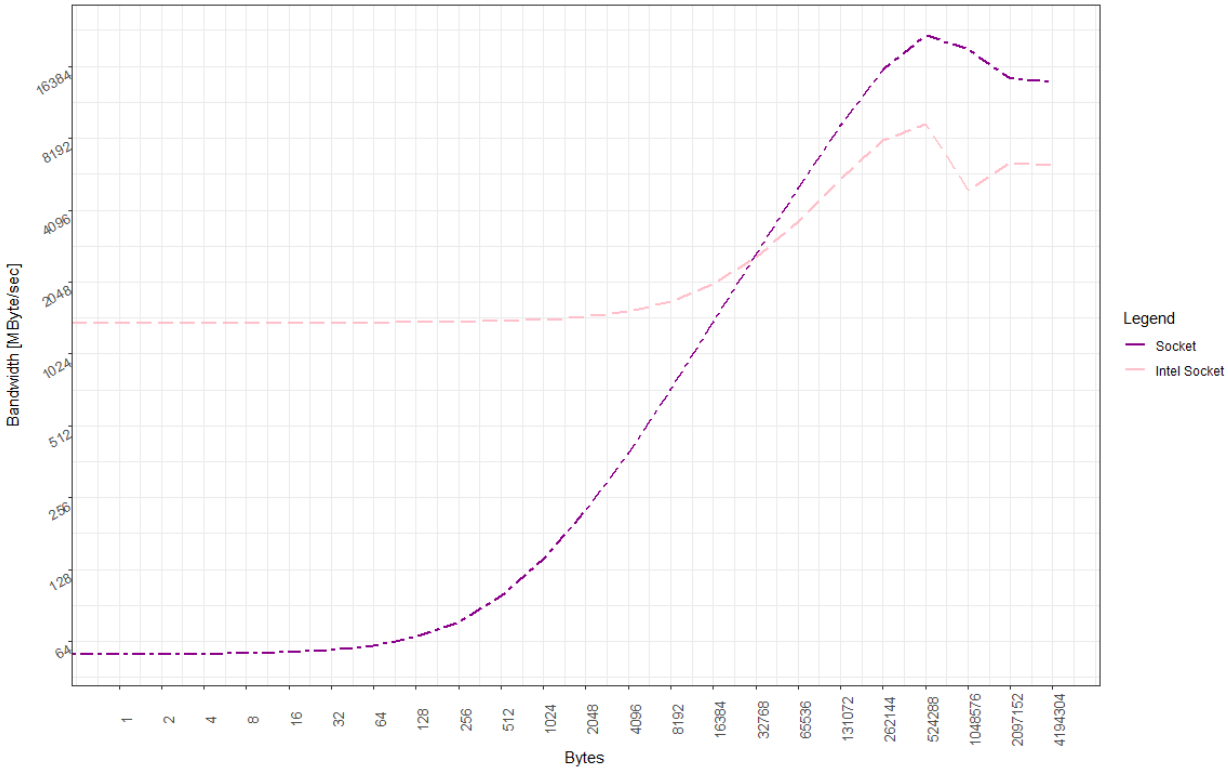


Figure 9: fitted model of Bandwidth computed for mapping by socket comparing GCC and INTEL modules

Section 3

MPI Jacobid

Abstract

The following is done in order to obtain the performance applying the Jacobid solver, taking the data from section 2 which are Latency and Bandwidth.

3.1. Objective

Applying the use of Jacobid model, the objective is to obtain the performance from latency and bandwidth on Orfeo, to obtain the scalability from the different mapping on different processors. We need to do the following computations to obtain the performance. The performance comes from a given dimension, knowing the amount of data volume on the network and the communication time.

3.2 the Computation

The computations are done by the mapping of

- Thin node: for 4,8,12 processors
- 2 thin nodes: for 12, 24,48 processors
- GPU node: for 4,8,12,24,48 processors

After a couple of iterations, the calculations perform are the following:

On Thin node:

Per node:

N	Nx	Ny	Nz	k	c(L,N)	Tc(L,N)	P(L,N)	P(1)*N/P(L,N)
1	1	1	1	0	0	0	114.1012204	1
4	2	2	1	4	92160	7.661737771	303.0756684	1.505910665
8	2	2	2	6	138240	11.49260666	518.9762972	1.758865998
12	3	2	2	6	138240	11.49260666	778.4644459	1.758865998

Per core:

N	Nx	Ny	Nz	k	c(L,N)	Tc(L,N)	P(L,N)	P(1)*N/P(L,N)
1	1	1	1	0	0	0	114.1012204	1
4	2	2	1	4	92160	7.48989547	305.3766506	1.494563781
8	2	2	2	6	138240	11.23484321	524.0474388	1.741845672
12	3	2	2	6	138240	11.23484321	786.0711583	1.741845672

Per socket:

N	Nx	Ny	Nz	k	c(L,N)	Tc(L,N)	P(L,N)	P(1)*N/P(L,N)
1	1	1	1	0	0	0	114.1012204	1
4	2	2	1	4	92160	8.08460177	297.558448	1.533832713
8	2	2	2	6	138240	12.12690265	506.9055864	1.80074907
12	3	2	2	6	138240	12.12690265	760.3583796	1.80074907

Per no mapping:

N	Nx	Ny	Nz	k	c(L,N)	Tc(L,N)	P(L,N)	P(1)*N/P(L,N)
1	1	1	1	0	0	0	114.1012204	1
4	2	2	1	4	92160	7.661737771	303.0756684	1.505910665
8	2	2	2	6	138240	11.49260666	518.9762972	1.758865998
12	3	2	2	6	138240	11.49260666	778.4644459	1.758865998

On 2 Thin nodes:

Per node:

N	Nx	Ny	Nz	k	c(L,N)	Tc(L,N)	P(L,N)	P(1)*N/P(L,N)
1	1	1	1	0	0	0	114.1012204	1
12	3	2	2	6	138240	11.49260666	778.4644459	1.758865998
24	4	3	2	6	138240	11.49260666	1556.928892	1.758865998
48	4	4	3	6	138240	11.49260666	3113.857783	1.758865998

Per core:

N	Nx	Ny	Nz	k	c(L,N)	Tc(L,N)	P(L,N)	P(1)*N/P(L,N)
1	1	1	1	0	0	0	114.1012204	1
12	3	2	2	6	138240	11.23484321	786.0711583	1.741845672
24	4	3	2	6	138240	11.23484321	1572.142317	1.741845672
48	4	4	3	6	138240	11.23484321	3144.284633	1.741845672

Per socket:

N	Nx	Ny	Nz	k	c(L,N)	Tc(L,N)	P(L,N)	P(1)*N/P(L,N)
1	1	1	1	0	0	0	114.1012204	1
12	3	2	2	6	138240	12.12690265	760.3583796	1.80074907
24	4	3	2	6	138240	12.12690265	1520.716759	1.80074907
48	4	4	3	6	138240	12.12690265	3041.433518	1.80074907

Per no mapping:

N	Nx	Ny	Nz	k	c(L,N)	Tc(L,N)	P(L,N)	P(1)*N/P(L,N)
1	1	1	1	0	0	0	114.1012204	1
12	3	2	2	6	138240	11.49260666	778.4644459	1.758865998
24	4	3	2	6	138240	11.49260666	1556.928892	1.758865998
48	4	4	3	6	138240	11.49260666	3113.857783	1.758865998

On GPU node:

Per node:

N	Nx	Ny	Nz	k	c(L,N)	Tc(L,N)	P(L,N)	P(1)*N/P(L,N)
1	1	1	1	0	0	0	114.1012204	1
4	2	2	1	4	92160	7.661737771	303.0756684	1.505910665
8	2	2	2	6	138240	11.49260666	518.9762972	1.758865998
12	3	2	2	6	138240	11.49260666	778.4644459	1.758865998
24	4	3	2	6	138240	11.49260666	1556.928892	1.758865998
48	4	4	3	6	138240	11.49260666	3113.857783	1.758865998

Per core:

N	Nx	Ny	Nz	k	c(L,N)	Tc(L,N)	P(L,N)	P(1)*N/P(L,N)
1	1	1	1	0	0	0	114.1012204	1
4	2	2	1	4	92160	7.48989547	305.3766506	1.494563781
8	2	2	2	6	138240	11.23484321	524.0474388	1.741845672
12	3	2	2	6	138240	11.23484321	786.0711583	1.741845672
24	4	3	2	6	138240	11.23484321	1572.142317	1.741845672
48	4	4	3	6	138240	11.23484321	3144.284633	1.741845672

Per socket:

N	Nx	Ny	Nz	k	c(L,N)	Tc(L,N)	P(L,N)	P(1)*N/P(L,N)
1	1	1	1	0	0	0	114.1012204	1
4	2	2	1	4	92160	8.08460177	297.558448	1.533832713
8	2	2	2	6	138240	12.12690265	506.9055864	1.80074907
12	3	2	2	6	138240	12.12690265	760.3583796	1.80074907
24	4	3	2	6	138240	12.12690265	1520.716759	1.80074907
48	4	4	3	6	138240	12.12690265	3041.433518	1.80074907

Per no mapping:

N	Nx	Ny	Nz	k	c(L,N)	Tc(L,N)	P(L,N)	P(1)*N/P(L,N)
1	1	1	1	0	0	0	114.1012204	1
4	2	2	1	4	92160	7.661737771	303.0756684	1.505910665
8	2	2	2	6	138240	11.49260666	518.9762972	1.758865998
12	3	2	2	6	138240	11.49260666	778.4644459	1.758865998
24	4	3	2	6	138240	11.49260666	1556.928892	1.758865998
48	4	4	3	6	138240	11.49260666	3113.857783	1.758865998

3.3 Evaluation

In the following tables, the slowdown factor compared to perfect scaling for the different number of processors and different mappings are shown

On Thin node							
Per node		Per core		Per socket		Per no map	
N	P	N	P	N	P	N	P
1	1	1	1	1	1	1	1
4	1.505910665	4	1.494563781	4	1.533832713	4	1.505910665
8	1.758865998	8	1.741845672	8	1.80074907	8	1.758865998
12	1.758865998	12	1.741845672	12	1.80074907	12	1.758865998

On 2 Thin nodes							
Per node		Per core		Per socket		Per no map	
N	P	N	P	N	P	N	P
1	1	1	1	1	1	1	1
12	1.758865998	12	1.741845672	12	1.80074907	12	1.758865998
24	1.758865998	24	1.741845672	24	1.80074907	24	1.758865998
48	1.758865998	48	1.741845672	48	1.80074907	48	1.758865998

On GPU node							
Per node		Per core		Per socket		Per no map	
N	P	N	P	N	P	N	P
1	1	1	1	1	1	1	1
4	1.505910665	4	1.494563781	4	1.533832713	4	1.505910665
8	1.758865998	8	1.741845672	8	1.80074907	8	1.758865998
12	1.758865998	12	1.741845672	12	1.80074907	12	1.758865998
24	1.758865998	24	1.741845672	24	1.80074907	24	1.758865998
48	1.758865998	48	1.741845672	48	1.80074907	48	1.758865998

The scalability is observed by the following graphs

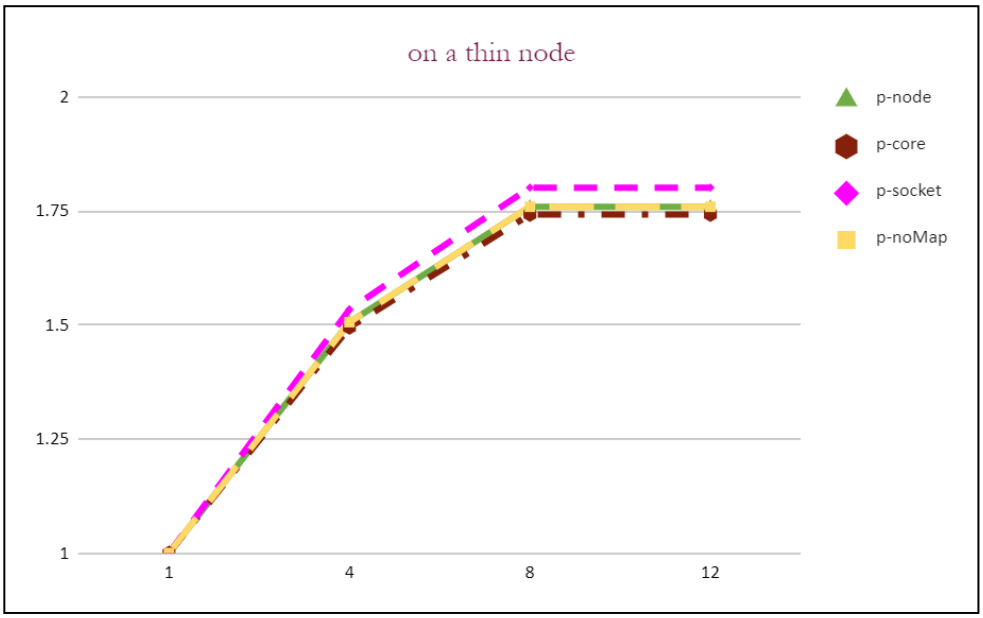


Figure 1: on a Thin node performance scalability is present increasing twice

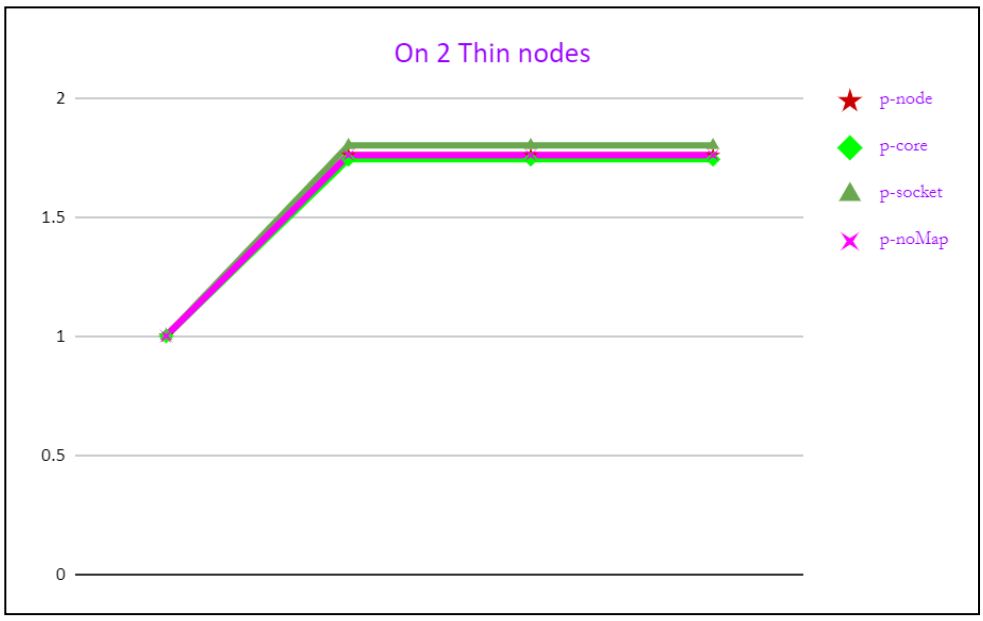


Figure 2: on 2 Thin nodes stable performance and not increasing with no scalability

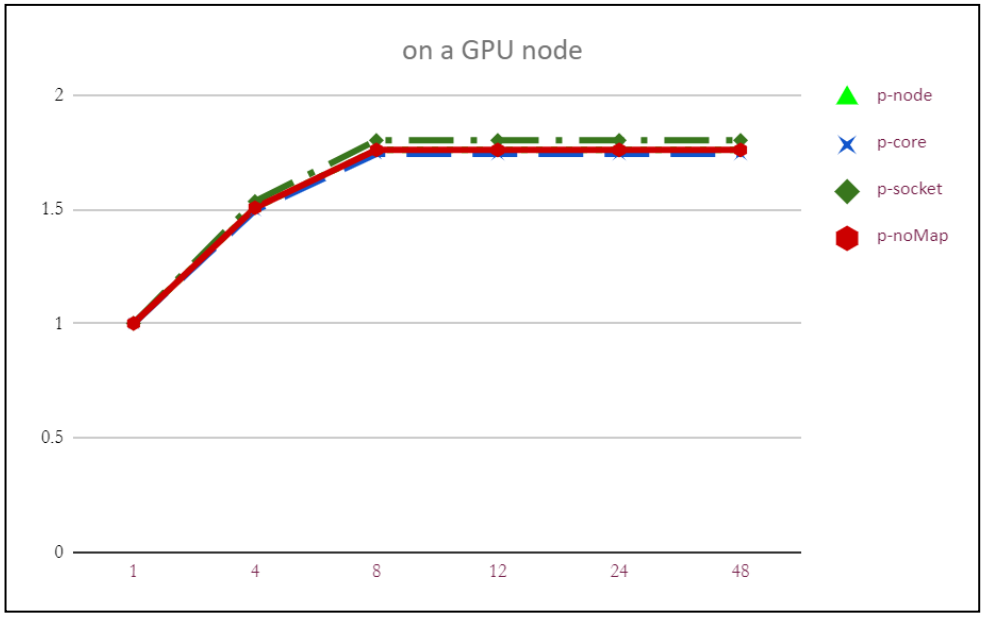


Figure 3: on a GPU node performance scalability is present increasing twice