Assignment 1 – Report

Submitted by: Fatma Sayed Mahmoud Sayed Moustafa

ORFEO username: fmoustafa

Section1: MPI programming

- Code 1: ring.c
  - Code and Readme file: on github
  - Set of variables used:
    - rank → rank of processor
    - size → number of processors
    - left → rank of the left node
    - right → rank of the right node
    - msgleft → message to be sent to the left
    - msgright → message to be sent to the right
    - tag → tag of the processor
    - lastLeftTag → tag of the left processor that sent the last message
    - lastRightTag → tag of the right processor that sent the last message
  - Formulas used:
    - Speedup is the ratio between time to execute the serial workload and time to execute the workload parallelly with P processors

$$S(P) = \frac{T(1)}{T(P)}$$

    - Efficiency is the average speedup by one processor in a parallel program

$$E(P) = \frac{S(P)}{P}$$

  - Test cases and iterations:
    - 100 iterations were performed for 2, 4, 8, 16, 32 processors
  - Results and conclusion:
    - As the number of processors increase, the time of computation increase. This shows that the ring code has a weak scalability, since the efficiency decrease with increasing the number of processors.
- Table1: shows the computational time, speedup and efficiency as the number of processors increase

Processor	Time		S(P) = T(1) / T(P)	E(P) = S(P) / P
1	0.00128008		1	1
2	0.00210541		0.6079955923	0.3039977962
4	0.00235606		0.5433138375	0.1358284594
8	0.00272901		0.469063873	0.05863298412
16	0.00396601		0.3227626758	0.02017266724
32	0.00554364		0.230909655	0.00721592672

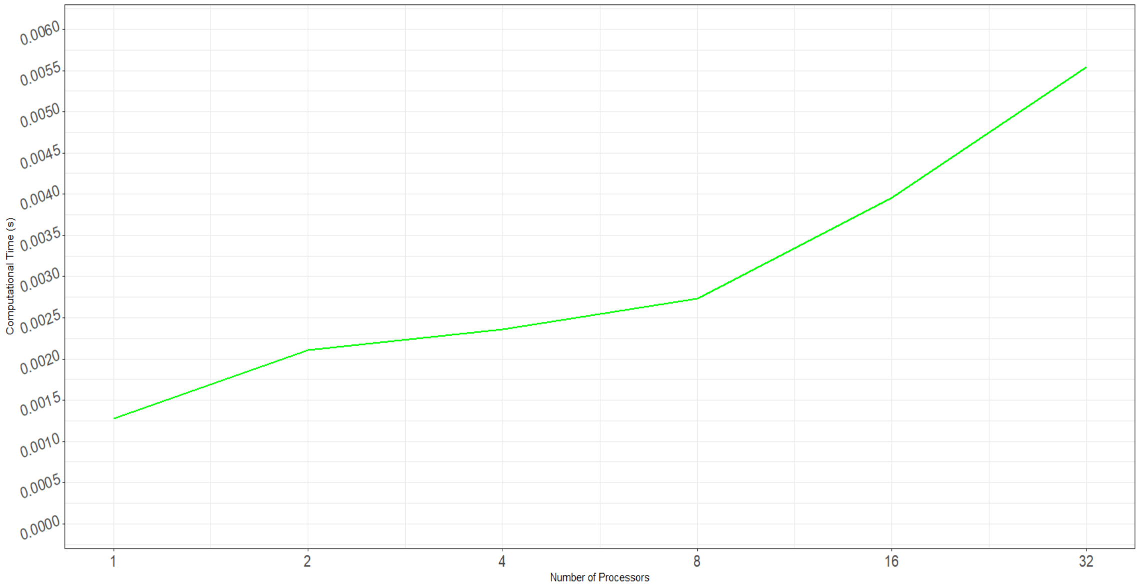


Figure 1: Computational time increases almost quadratically when number of processors doubles

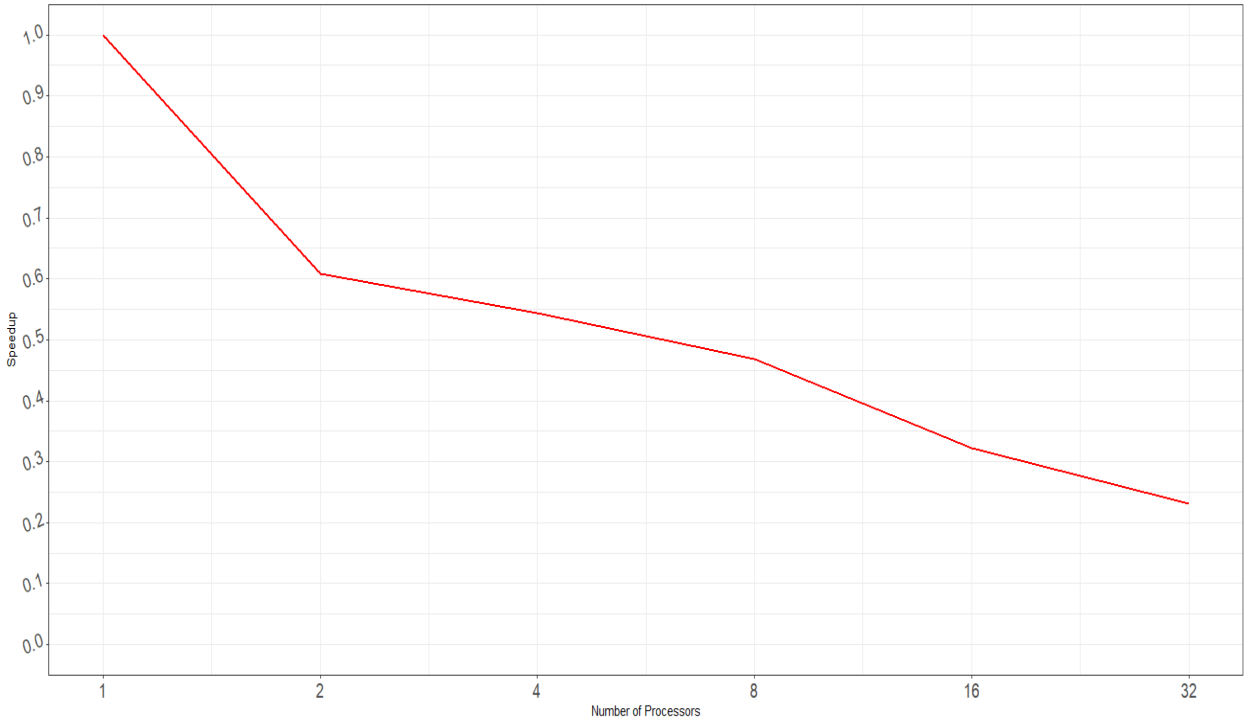


Figure 2: Speedup decrease as the number of processors increase

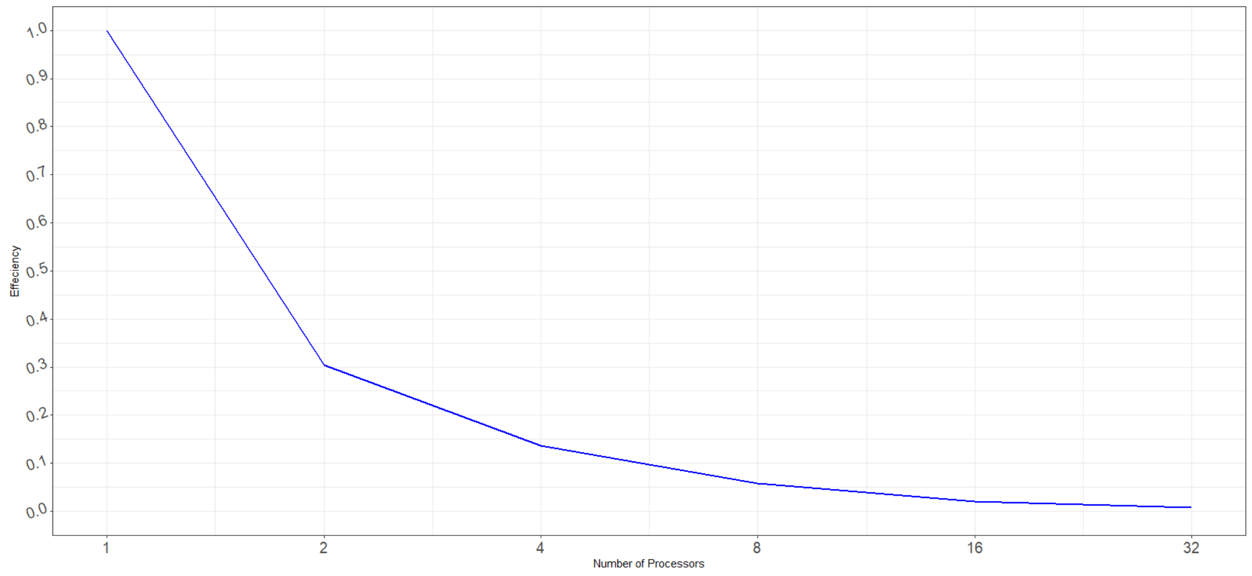


Figure 3: Efficiency decrease as the number of processors increase

- Code 2: sum3Dmatrix.cpp
    - Code and Readme file: on github
    - Set of variables used:
        - rank → rank of processor
        - size → number of processors
        - x → dimension x-axis
        - y → dimension y-axis
        - z → dimension z-axis
        - x_submatrix → dimension submatrix on x-axis
        - y_ submatrix → dimension submatrix on y-axis
        - z_ submatrix → dimension submatrix on z-axis
        - matrix_1, matrix_2, matrix_3, matrix_1_ submatrix, matrix_2_ submatrix, matrix_3_ submatrix allocated dynamically with corresponding dimension
        - elements_ submatrix → Multiplication of the submatrices with smaller dimensions. This value is needed in the split of the job in multiprocessors.
    - Test cases: Matrices dimensions with number of processors kept constant at 24
        - 2400 x 100 x 100
        - 1200 x 200 x 100
        - 800 x 300 x 100
    - Results and conclusion:

- Maximum, minimum and average time for different submatrix division for the following matrix size: 2400 x 100 x 100

	SUBMATRIX DIVISION	MAX TIME	MIN TIME	AVG TIME
1D	100 * 100 * 100	0.111555	0.110219	0.111346
2D	200 * 50 * 100	0.129135	0.129024	0.129085
	400 * 25 * 100	0.106145	0.106029	0.106082
3D	400 * 50 * 50	0.107988	0.106702	0.107354
	800 * 25 * 50	0.1086	0.108506	0.108555

- Maximum, minimum and average time for different submatrix division for the following matrix size: 1200 x 200 x 100

	SUBMATRIX DIVISION	MAX TIME	MIN TIME	AVG TIME
1D	50 * 100 * 100	0.070834	0.0707541	0.0707909
2D	100 * 100 * 100	0.108833	0.107592	0.107559
	200 * 50 * 100	0.109028	0.107788	0.108683
	200 * 200 * 25	0.108971	0.108835	0.108912
	100 * 200 * 50	0.109147	0.107856	0.108938
	400 * 25 * 100	0.109756	0.10964	0.109706
3D	200 * 100 * 50	0.108592	0.107287	0.108236
	400 * 50 * 50	0.120001	0.118698	0.119639

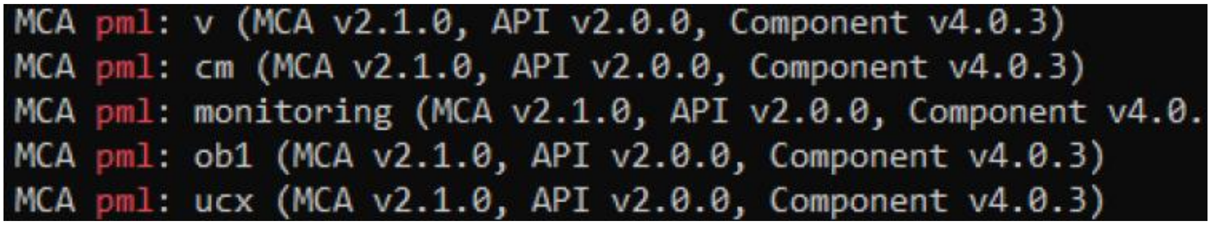
- Maximum, minimum and average time for different submatrix division for the following matrix size: 800 x 300 x 100

	SUBMATRIX DIVISION	MAX TIME	MIN TIME	AVG TIME
2D	100 * 100 * 100	0.108054	0.107968	0.108007
	200 * 50 * 100	0.108232	0.106991	0.10789
	400 * 25 * 100	0.105332	0.105194	0.105265
	800 * 25 * 50	0.108422	0.108318	0.108368
	800 * 50 * 25	0.108903	0.108833	0.108864
3D	200 * 100 * 50	0.106568	0.106472	0.106518
	400 * 100 * 25	0.108621	0.108503	0.108557

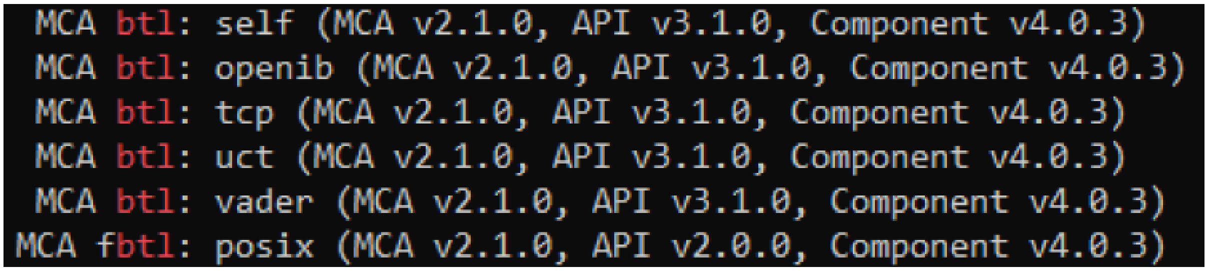
	400 * 50 * 50	0.108691	0.10857	0.108633

Section2: Measure MPI point to point performance

- Set of variables used:
  - N → map by node
  - S → map by socket
  - C → map by core
  - RB → report bindings
- Running Netwrok parameters:
  - MCA: modular component architecture
  - OB1: multidevice multi rail engine
  - PML: MPI point-to-point communication
  - BTL: byte transport layer
  - BML: BTL multiplexing layer
  - TCP: transmission control protocol
  - SELF: process-loopback communications
  - VADER: shared memory
  - UCX: unified communications X



```
MCA pml: v (MCA v2.1.0, API v2.0.0, Component v4.0.3)
MCA pml: cm (MCA v2.1.0, API v2.0.0, Component v4.0.3)
MCA pml: monitoring (MCA v2.1.0, API v2.0.0, Component v4.0.
MCA pml: ob1 (MCA v2.1.0, API v2.0.0, Component v4.0.3)
MCA pml: ucx (MCA v2.1.0, API v2.0.0, Component v4.0.3)
```



```
 MCA btl: self (MCA v2.1.0, API v3.1.0, Component v4.0.3)
 MCA btl: openib (MCA v2.1.0, API v3.1.0, Component v4.0.3)
 MCA btl: tcp (MCA v2.1.0, API v3.1.0, Component v4.0.3)
 MCA btl: uct (MCA v2.1.0, API v3.1.0, Component v4.0.3)
 MCA btl: vader (MCA v2.1.0, API v3.1.0, Component v4.0.3)
MCA fbtl: posix (MCA v2.1.0, API v2.0.0, Component v4.0.3)
```

- Test cases and iterations:
  - CSV files with all the runs using GCC and INTEL modules: on GitHub
  - MPI Benchmark comparison between GCC and INTEL

		Latency	Bandwidth	
GCC - No mapping		0.2	13431	

GCC - No mapping with ucx		0.21	13674	
INTEL		0.42	6445	
Run on 2 contiguous processors				
INTEL PROCESSORLIST 0,1		0.42	6377	
Run on the same socket				
INTEL PROCESSORLIST 0,2		0.23	10926	

- Different mappings for MPI Benchmark comparison between GCC and INTEL

	SOCKET		**NODE**		**CORE**	
	Latency	Bandwidth	Latency	Bandwidth	Latency	Bandwidth
basic	0.39	14125	0.21	13270	0.2	13776
pml = ob1	0.53	10284	0.26	9685	0.25	9724
pml = ob1, btl = self	7.97	3477	5.39	7037	5.43	7000
pml = ucx, btl = self	0.41	14058	0.2	13656	0.2	13771
pml = ucx, btl = vader	0.4	14277	0.2	13646	0.2	13083
INTEL	0.43	6341	0.42	6439	0.43	6336

- Results and conclusion:
  - GENERAL OVERVIEW:
    - For the OpenMPI the latency and bandwidth are 0.2 microsecond and 13431 respectively. Using a UCX protocol does not have an effect on these values.
    - It is noticed that Intel has almost double the latency and half the bandwidth, even we run it on 2 contiguous processors.
    - However, when run on the same socket the latency and bandwidth has values almost similar to that with the OpenMPI.
    - UCX for infiniband, OB1 PML + BTL for all others
  - OPENMPI:
    - Mapping by core or by node has similar results in terms of latency and bandwidth. When mapping by socket we get almost similar bandwidth but almost double the latency.
    - Using the PML as UCX maintains the values of latency and bandwidth, regardless of the BTL being self or vader.

    - With the PML being OB1 the latency increases by around 25% and the bandwidth decreases by almost 25%.
    - In the case of having PML as OB1 and BTL as self, the latency is the highest and the bandwidth is the lowest among all the other cases.
        - Latency shows an increase of 20 times in the map by socket case and even more when mapping is done by node or core.
        - The bandwidth decreases to quarter its value when mapping by socket and to half when mapping by core or node.
        - TCP protocol has high latency because it has high travelling distance.

- INTEL:
    - Whatever the type of mapping that is being done, the values of latency and bandwidth remains almost the same for all these mappings.
    - Also, they are similar to the values obtained when we run on 2 contiguous processors.
    - Latency does not increase. We have 2 processes one on core 0 and the other process is on the other core with same socket and same node.

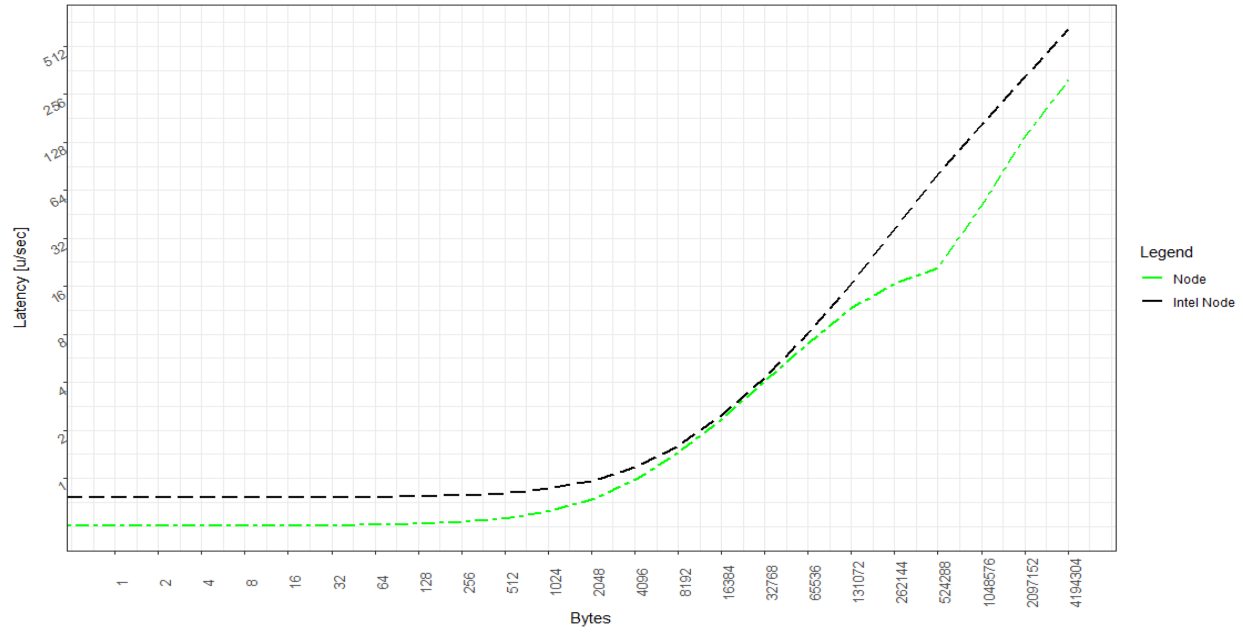


Figure 4: fitted model of latency computed for mapping by node comparing GCC and INTEL modules

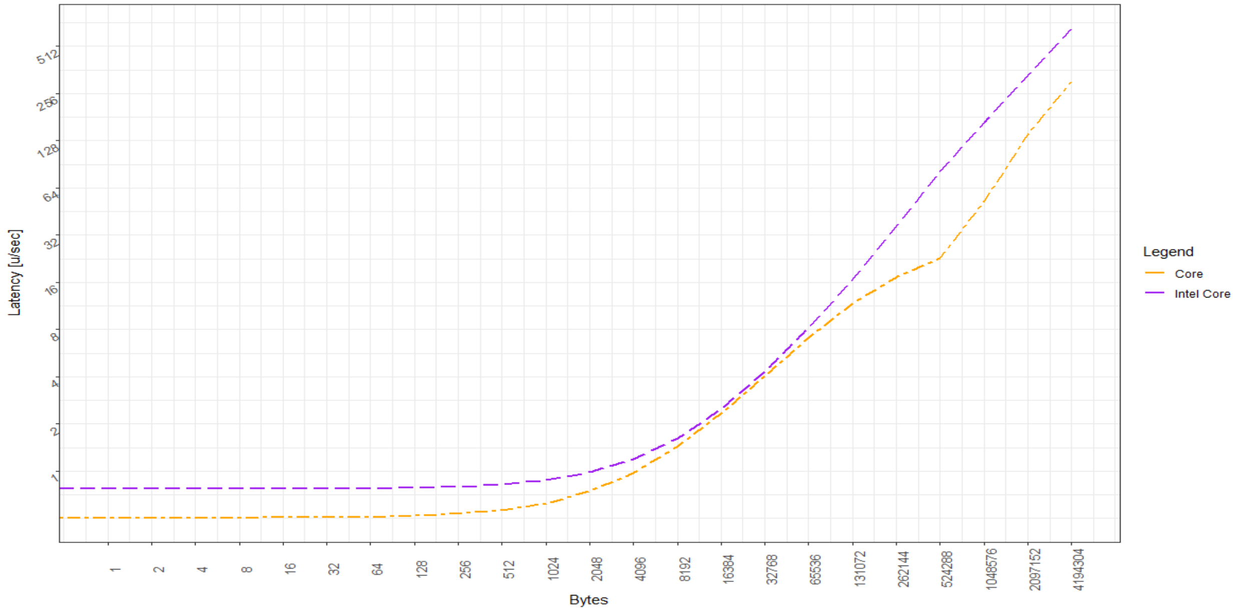


Figure 5: fitted model of latency computed for mapping by core comparing GCC and INTEL modules

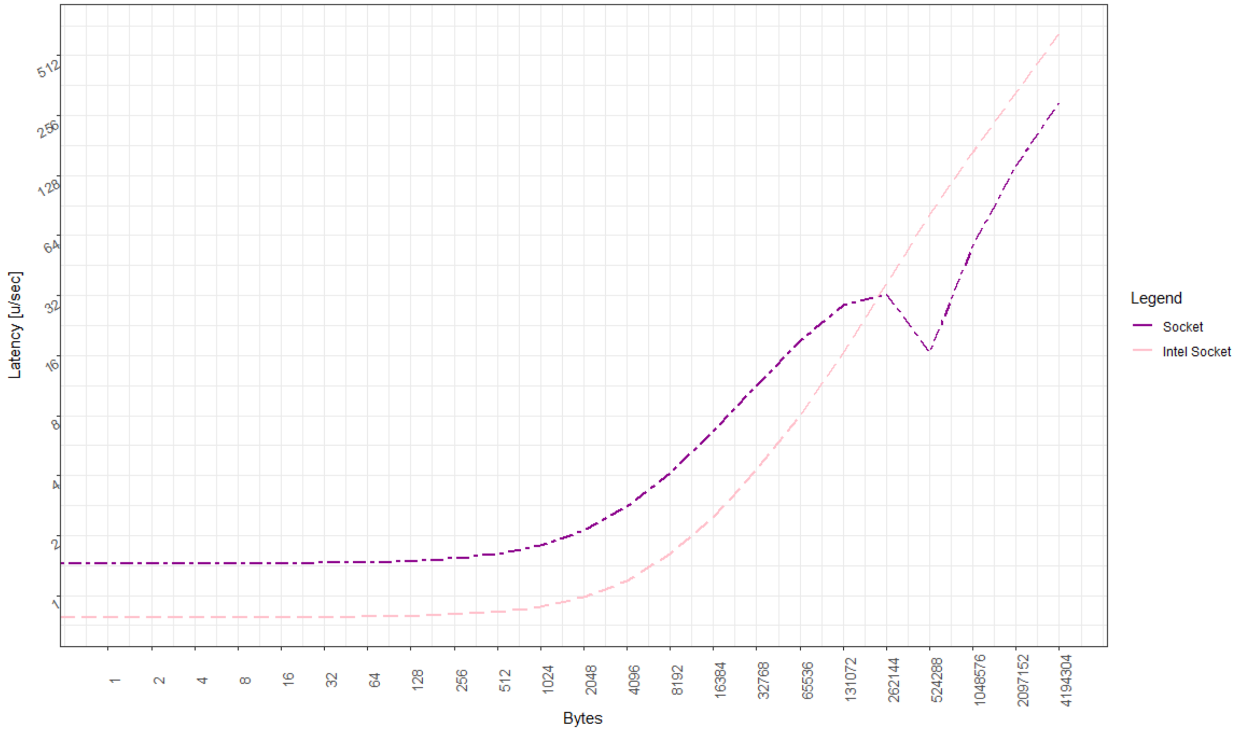


Figure 6: fitted model of latency computed for mapping by socket comparing GCC and INTEL modules

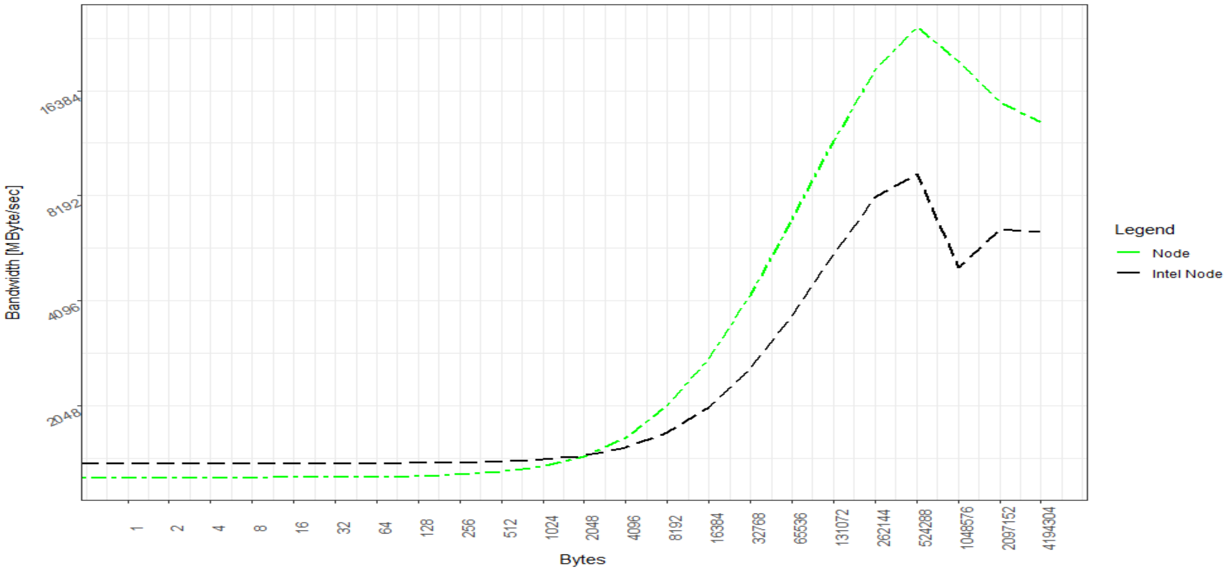


Figure 7: fitted model of Bandwidth computed for mapping by node comparing GCC and INTEL modules

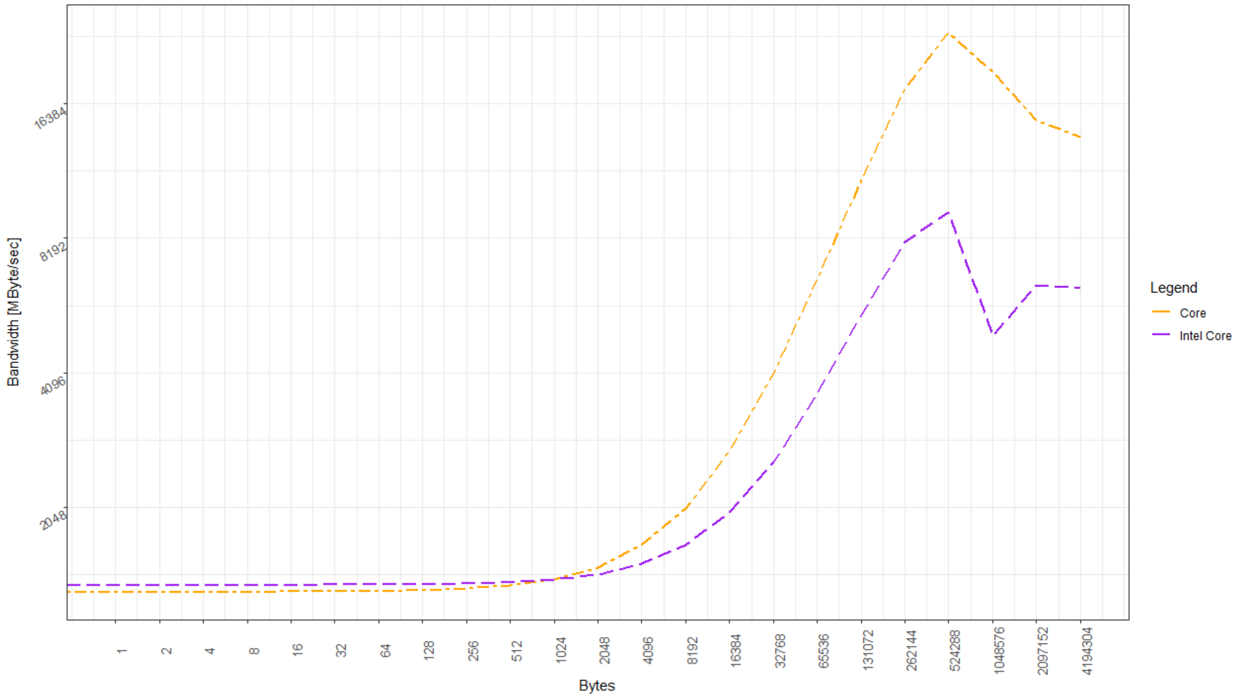


Figure 8: Fitted model of Bandwidth computed for mapping by core comparing GCC and INTEL modules

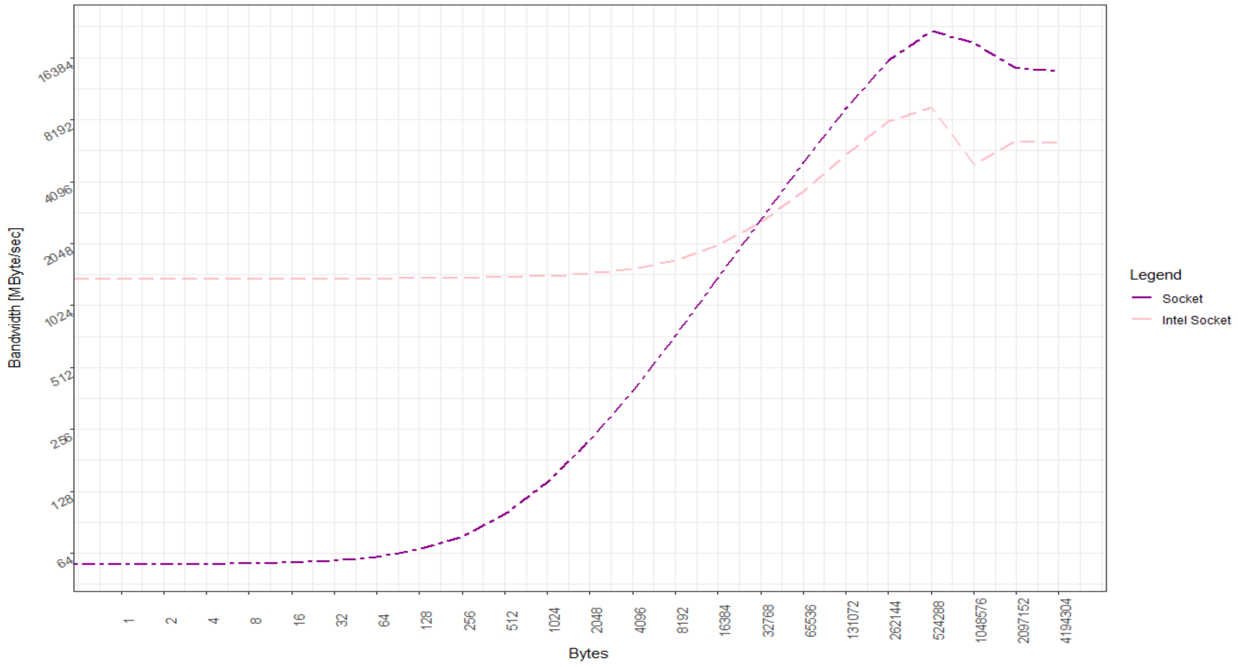


Figure 9: fitted model of Bandwidth computed for mapping by socket comparing GCC and INTEL modules

Section3: Compare performance observed against performance model for Jacobi solver

- Set of variables used:
  - N→ number of processes
  - N_x → number of processors in along x
  - N_y → number of processors in along y
  - N_z → number of processors in along z
  - k → largest number of coordinate directions in which number of processess is greater than one
  - c(L,N) → maximum bidirectional data volume transferred over a node's network link
  - B → full-duplex bandwidth
  - T_c(L,N)→ communication time
  - T_s(L,N)→ raw compute time for all cell updates in a Jacobi sweep
  - P_1(L) → single-processor performance for a domain size L^3
  - P(L,N) → prediction of
  - L → spatial dimension
  - T_l → latency of network
- Formulas used:
  - $c(L,\vec{N}) = L^2 \cdot k \cdot 2 \cdot 8$
  - $T_c(L,\vec{N}) = \frac{c(L,\vec{N})}{B} + kT_l$

        - $P(L,\vec{N}) = \frac{L^3 N}{T_C(L,\vec{N})+T_S(L)}$
    - Running parameters:
        - Spat_dim: problem size = L , in this case it was 1200
        - proc_dim: possible presets for number of processes, in this case it is 0
        - pbc_check = periodicity, in this case is t
    - Test cases and iterations:
        - Runs on a thin node for 4,8,12 processors
        - Runs on 2 thin nodes for 12, 24, 48 processors
        - Runs on GPU node
    - Results and conclusion:
      In the following tables, some of the output of the runs performed are shown.
        - Runs on a thin node for 4,8,12 processors
            - Map by 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	114.10122	1
4	2	2	1	4	92160	7.661737771	303.075668	1.505910665
8	2	2	2	6	138240	11.49260666	518.976297	1.758865998
12	3	2	2	6	138240	11.49260666	778.464446	1.758865998

            - Map by 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	114.10122	1
4	2	2	1	4	92160	7.48989547	305.376651	1.494563781
8	2	2	2	6	138240	11.23484321	524.047439	1.741845672
12	3	2	2	6	138240	11.23484321	786.071158	1.741845672

            - Map by 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	114.10122	1
4	2	2	1	4	92160	8.08460177	297.558448	1.533832713
8	2	2	2	6	138240	12.12690265	506.905586	1.80074907
12	3	2	2	6	138240	12.12690265	760.35838	1.80074907

        - Runs on 2 thin nodes for 12, 24, 48 processors
            - Map by 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	114.10122	1
12	3	2	2	6	138240	11.49260666	778.464446	1.758865998
24	4	3	2	6	138240	11.49260666	1556.92889	1.758865998
48	4	4	3	6	138240	11.49260666	3113.85778	1.758865998

- Map by 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	114.10122	1
12	3	2	2	6	138240	11.23484321	786.071158	1.741845672
24	4	3	2	6	138240	11.23484321	1572.14232	1.741845672
48	4	4	3	6	138240	11.23484321	3144.28463	1.741845672

- Map by 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	114.10122	1
12	3	2	2	6	138240	12.12690265	760.35838	1.80074907
24	4	3	2	6	138240	12.12690265	1520.71676	1.80074907
48	4	4	3	6	138240	12.12690265	3041.43352	1.80074907

- ▪ Runs on GPU node

- Map by 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	114.10122	1
4	2	2	1	4	92160	7.661737771	303.075668	1.505910665
8	2	2	2	6	138240	11.49260666	518.976297	1.758865998
12	3	2	2	6	138240	11.49260666	778.464446	1.758865998
24	4	3	2	6	138240	11.49260666	1556.92889	1.758865998
48	4	4	3	6	138240	11.49260666	3113.85778	1.758865998

- Map by 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	114.10122	1
4	2	2	1	4	92160	7.48989547	305.376651	1.494563781
8	2	2	2	6	138240	11.23484321	524.047439	1.741845672
12	3	2	2	6	138240	11.23484321	786.071158	1.741845672
24	4	3	2	6	138240	11.23484321	1572.14232	1.741845672
48	4	4	3	6	138240	11.23484321	3144.28463	1.741845672

- Map by 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	114.10122	1
4	2	2	1	4	92160	8.08460177	297.558448	1.533832713
8	2	2	2	6	138240	12.12690265	506.905586	1.80074907
12	3	2	2	6	138240	12.12690265	760.35838	1.80074907
24	4	3	2	6	138240	12.12690265	1520.71676	1.80074907
48	4	4	3	6	138240	12.12690265	3041.43352	1.80074907

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

- Runs on a thin node for 4,8,12 processors

by node		by core		by socket	
N	P	N	P	N	P
1	1	1	1	1	1
4	1.505910665	4	1.494563781	4	1.533832713
8	1.758865998	8	1.741845672	8	1.80074907
12	1.758865998	12	1.741845672	12	1.80074907

- Runs on 2 thin nodes for 12, 24, 48 processors

by node		by core		by socket	
N	P	N	P	N	P
1	1	1	1	1	1
12	1.758865998	12	1.741845672	12	1.80074907
24	1.758865998	24	1.741845672	24	1.80074907
48	1.758865998	48	1.741845672	48	1.80074907

- Runs on GPU node

by node		by core		by socket	
N	P	N	P	N	P
1	1	1	1	1	1
4	1.505910665	4	1.494563781	4	1.533832713
8	1.758865998	8	1.741845672	8	1.80074907
12	1.758865998	12	1.741845672	12	1.80074907
24	1.758865998	24	1.741845672	24	1.80074907
48	1.758865998	48	1.741845672	48	1.80074907