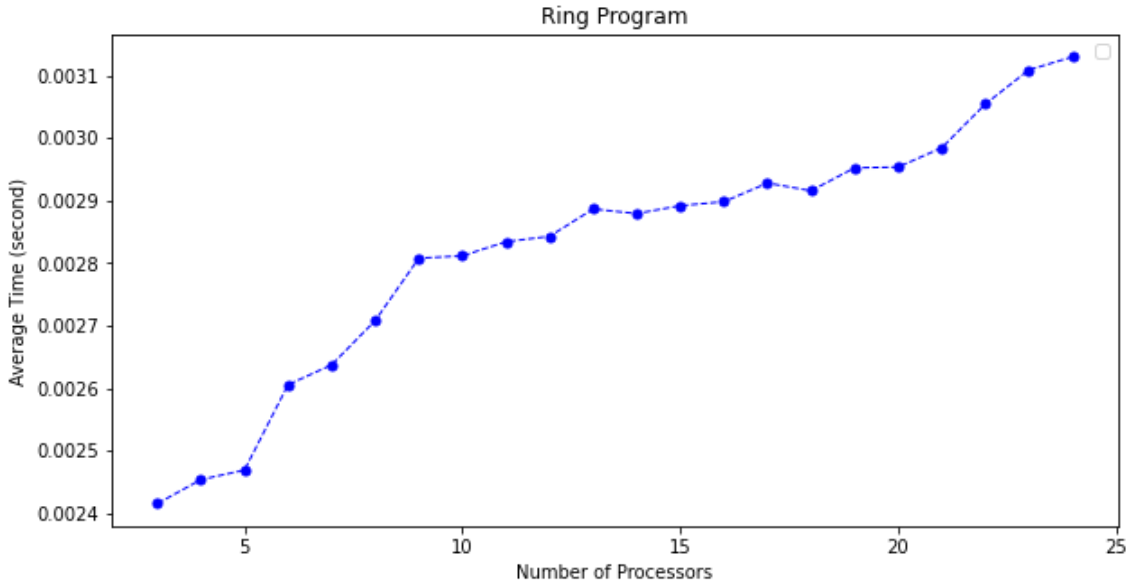
Introduction to High Performance Computing – Assignment 1
2021/22
Elham Babaei

Section 1

Ring

The ring MPI program implements a stream of messages to the left and right neighbors of each processor, in a 1D virtual topology. At the end, each processor will receive the original message created by itself.
The average walltime of program by running with 3 to 24 processors on a thin node is shown bellow. MPI_Wtime was used to get the time of parallel processors. It shows that the running time increases almost linearly by increasing the number of processors.



Matrix summation

The MPI program uses a 3D , 2D and 1D topology to calculated the summation of two matrices of size 24,000,000 elements. The collective operations MPI_scatter and MPI_gather are used to sum the matrices. This is an example output without printing the matrices because they are too large to be shown :

```
[MPI process, old 0, new 0] I am located at (0, 0, 0).
[MPI process, old 10, new 10] I am located at (1, 2, 0).
[MPI process, old 11, new 11] I am located at (1, 2, 1).
[MPI process, old 12, new 12] I am located at (2, 0, 0).
[MPI process, old 13, new 13] I am located at (2, 0, 1).
[MPI process, old 14, new 14] I am located at (2, 1, 0).
[MPI process, old 15, new 15] I am located at (2, 1, 1).
[MPI process, old 16, new 16] I am located at (2, 2, 0).
[MPI process, old 17, new 17] I am located at (2, 2, 1).
[MPI process, old 18, new 18] I am located at (3, 0, 0).
[MPI process, old 19, new 19] I am located at (3, 0, 1).
[MPI process, old 1, new 1] I am located at (0, 0, 1).
[MPI process, old 20, new 20] I am located at (3, 1, 0).
[MPI process, old 21, new 21] I am located at (3, 1, 1).
[MPI process, old 22, new 22] I am located at (3, 2, 0).
[MPI process, old 23, new 23] I am located at (3, 2, 1).
[MPI process, old 2, new 2] I am located at (0, 1, 0).
[MPI process, old 3, new 3] I am located at (0, 1, 1).
[MPI process, old 4, new 4] I am located at (0, 2, 0).
[MPI process, old 5, new 5] I am located at (0, 2, 1).
[MPI process, old 6, new 6] I am located at (1, 0, 0).
[MPI process, old 7, new 7] I am located at (1, 0, 1).
[MPI process, old 8, new 8] I am located at (1, 1, 0).
[MPI process, old 9, new 9] I am located at (1, 1, 1).
 # walltime on processor 0 : 1.12539119
 # walltime on processor 10 : 1.04224771
 # walltime on processor 1 : 1.03687985
 # walltime on processor 11 : 1.03234379
 # walltime on processor 12 : 1.06215239
 # walltime on processor 13 : 1.03267194
 # walltime on processor 14 : 1.04314746
 # walltime on processor 15 : 1.03325880
 # walltime on processor 16 : 1.12642224
 # walltime on processor 17 : 1.03381873
 # walltime on processor 18 : 1.04456451
 # walltime on processor 19 : 1.03461569
 # walltime on processor 20 : 1.06472813
 # walltime on processor 2 : 1.04642620
 # walltime on processor 21 : 1.03515196
 # walltime on processor 22 : 1.04555381
 # walltime on processor 23 : 1.03570765
 # walltime on processor 3 : 1.02915165
 # walltime on processor 4 : 1.06479072
 # walltime on processor 5 : 1.02962802
 # walltime on processor 6 : 1.04046188
 # walltime on processor 7 : 1.03031375
 # walltime on processor 8 : 1.09581226
 # walltime on processor 9 : 1.03092724
```

The following table shows the average time taken by each topology to sum the matrices using 24 cores. There is no significant difference between them.

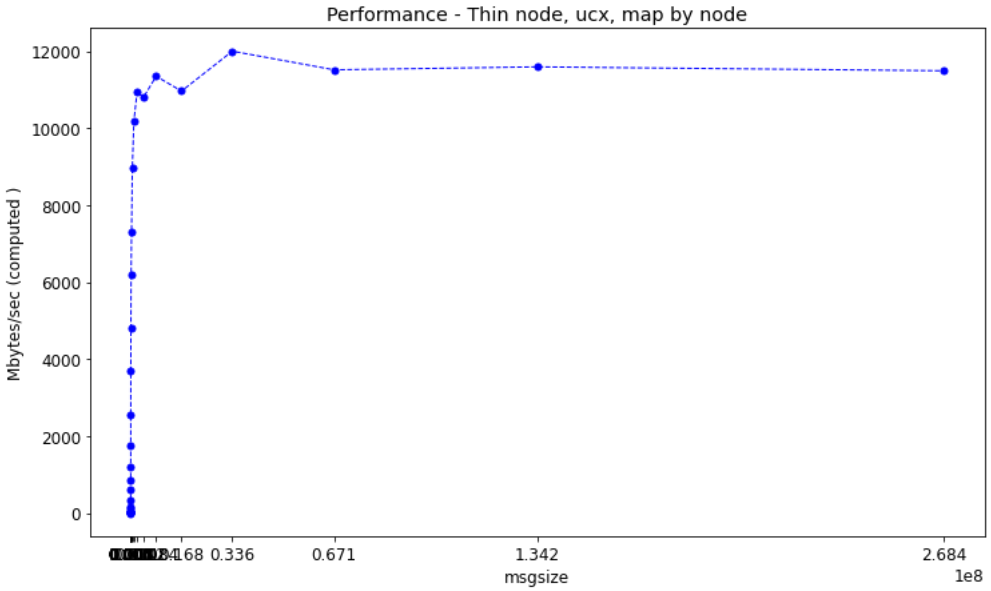
Time for matrix summation

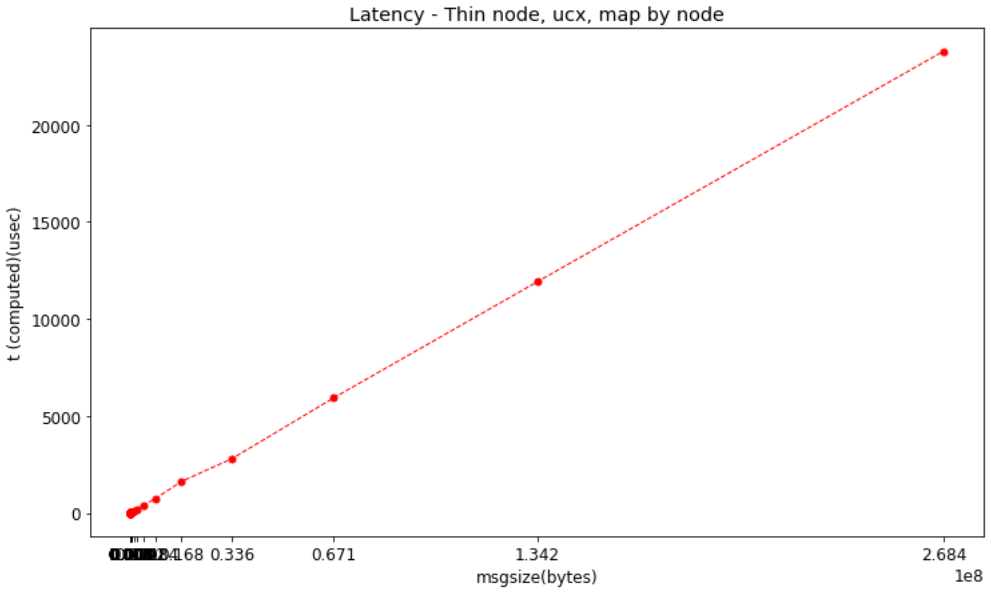
Processor	3D_topology	2D_topology	1D_topology
0	1.12316228	1.14025711	1.12134167
1	1.03486256	1.05298271	1.033897
2	1.0443308	1.06232767	1.04336923
3	1.02781756	1.0470062	1.02724903
4	1.06271431	1.08053301	1.06159431
5	1.02895656	1.04733367	1.02782139
6	1.03986602	1.05804719	1.03829657
7	1.0297575	1.04795348	1.02820754
8	1.09364157	1.11119236	1.09215196
9	1.02945411	1.04795596	1.01586432
10	1.04060209	1.05901745	1.03953664
11	1.030641	1.04904412	1.02954849
12	1.06117668	1.0791099	1.06015988
13	1.03118092	1.04947419	1.03026484
14	1.04206059	1.06003498	1.04100851
15	1.03212394	1.04981135	1.03110736
16	1.12417983	1.14131263	1.12233459
17	1.03179405	1.05038189	1.03070907
18	1.04263668	1.04914312	1.04169109
19	1.03266265	1.05128456	1.03189154
20	1.06291181	1.08118466	1.04892213
21	1.03328523	1.0518117	1.03241465
22	1.04383952	1.06202354	1.04273738
23	1.03408358	1.05222863	1.03283372
Average:	**1.048239243**	**1.065893837**	**1.046039705**

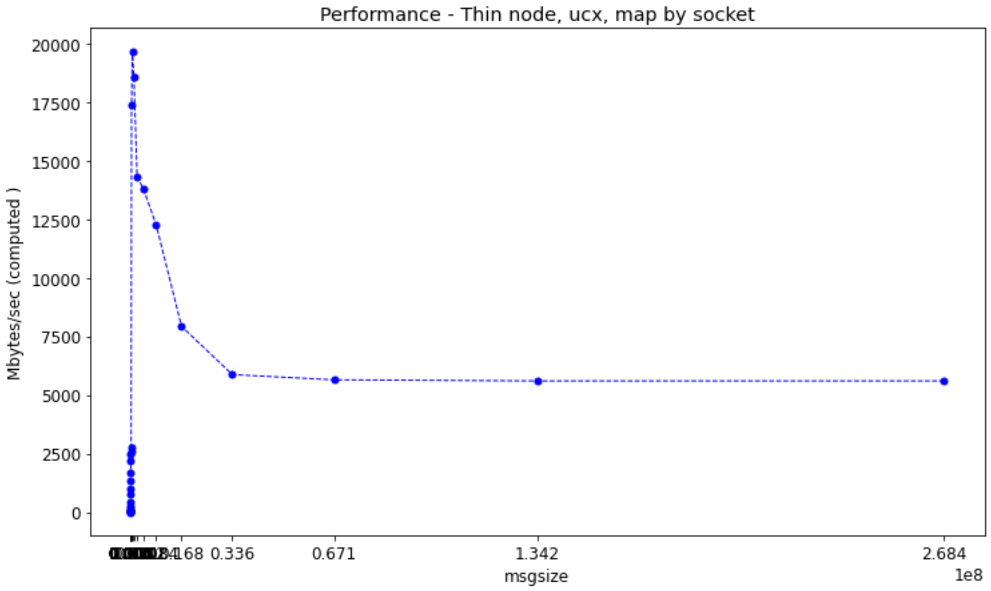
Section 2

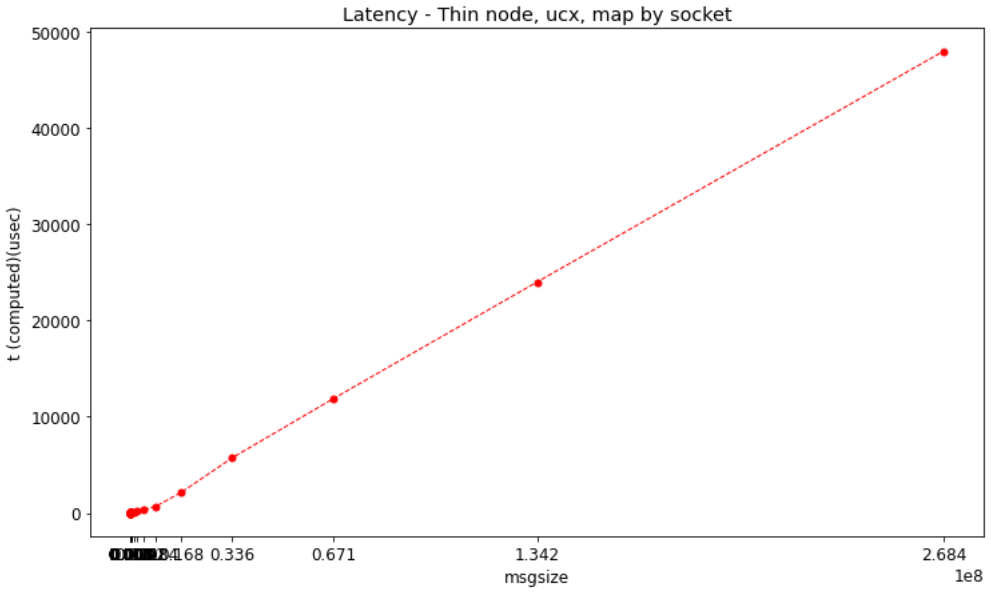
In this section we compile and run IMB-MPI1 which is an Intel MPI Benchmark with different networks and libraries and compare the results in terms of latency and bandwidth. We run each program 10 times to get the average values for latency and bandwidth that are shown in the following plots.
We also used the fitted linear regression model to estimate lambda and bandwidth. The results are shown in csv files in which the “computed” columns are the result of getting average between 10 runs and the “estimated” ones are calculated using the fitted least square model.

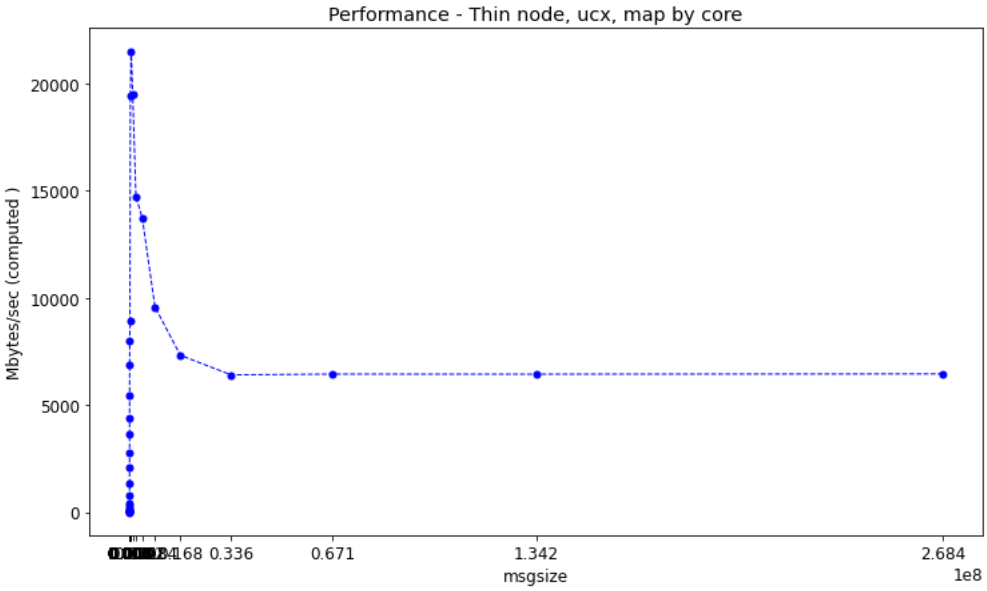
$$T_{comm} = \lambda + \text{message-size}/b_{network}$$

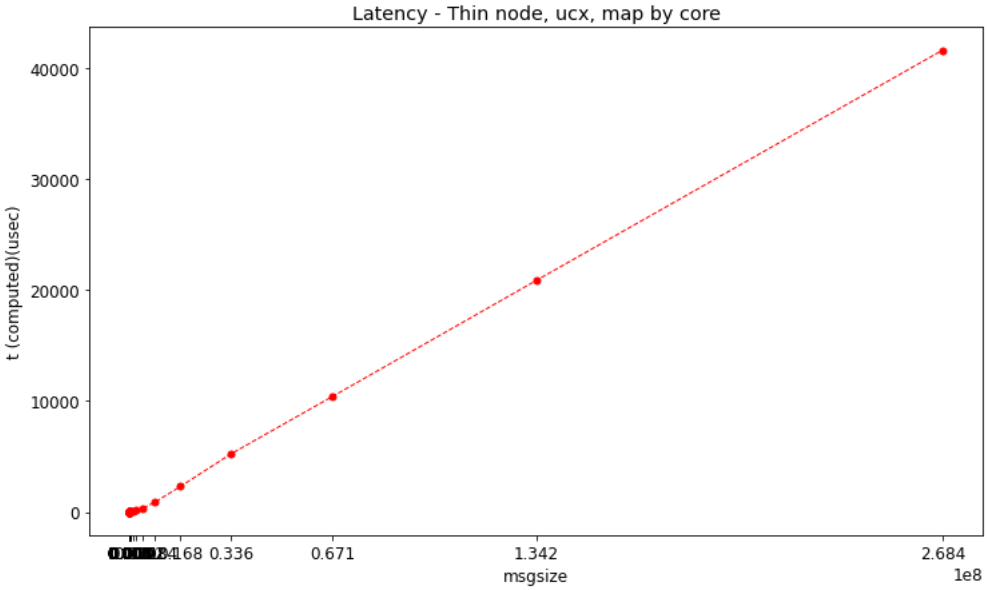

Performance - Thin node, ucx, map by node
Mbytes/sec (computed)
12000
10000
8000
6000
4000
2000
0
0.168
0.336
0.671
1.342
2.684
1e8
msgsize

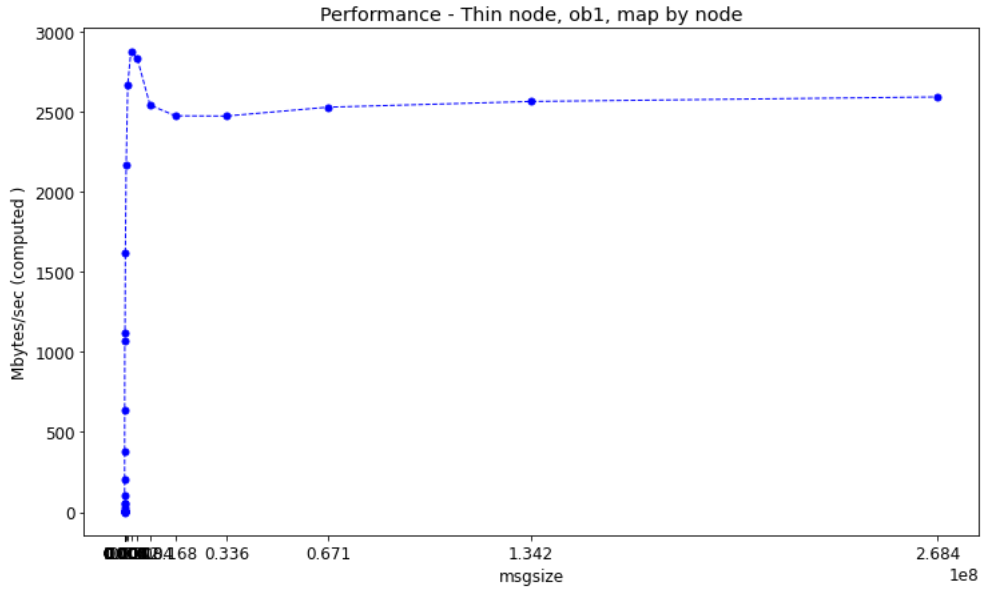

Latency - Thin node, ucx, map by node
t (computed)(usec)
20000
15000
10000
5000
0
0.168
0.336
0.671
1.342
2.684
1e8
msgsize(bytes)

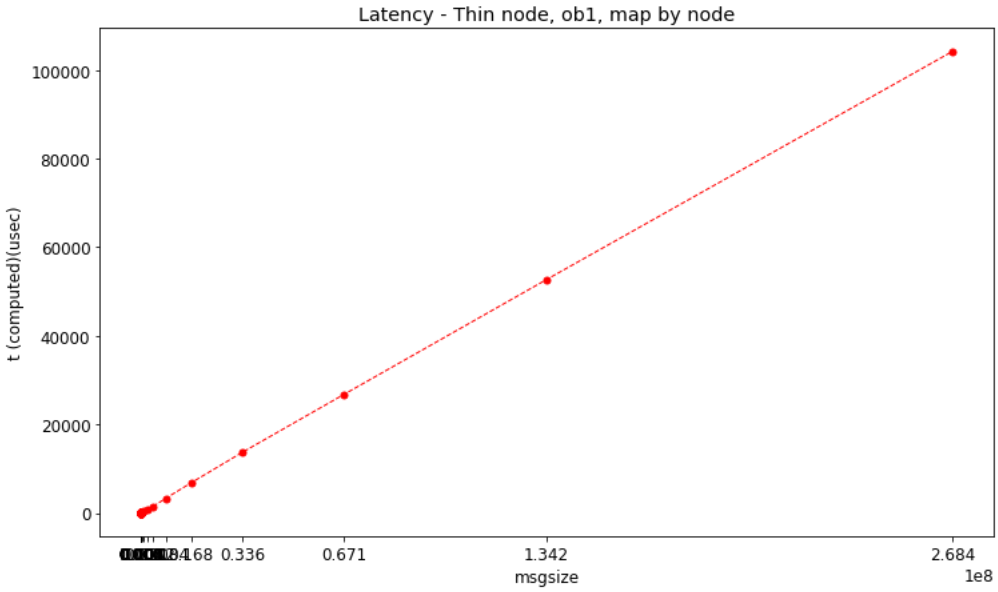

Performance - Thin node, ucx, map by socket
Mbytes/sec (computed)
20000
17500
15000
12500
10000
7500
5000
2500
0
0.168
0.336
0.671
1.342
2.684
msgsize
1e8

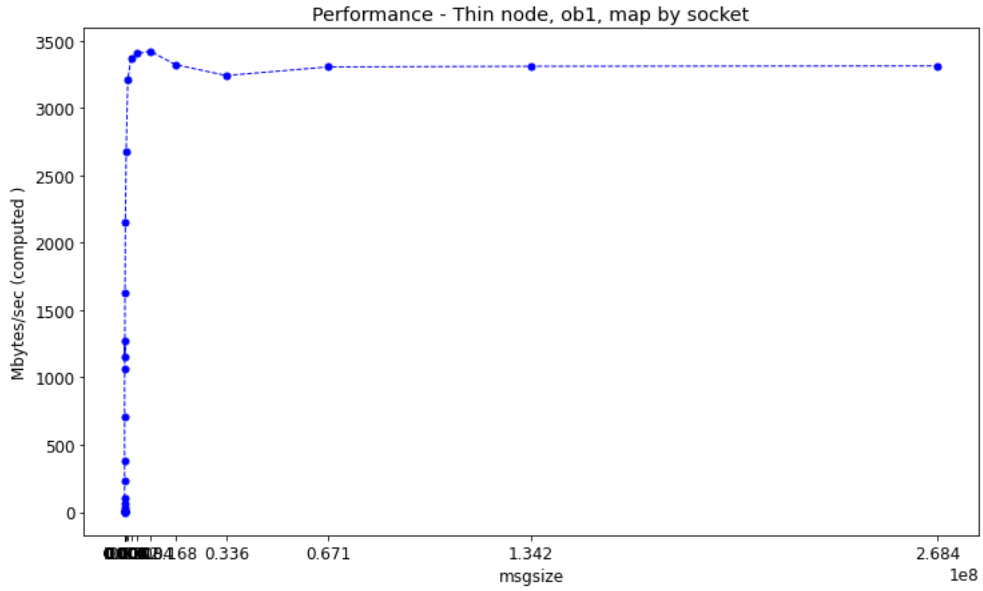

Latency - Thin node, ucx, map by socket
t (computed)(usec)
50000
40000
30000
20000
10000
0
0.168
0.336
0.671
1.342
2.684
msgsize
1e8

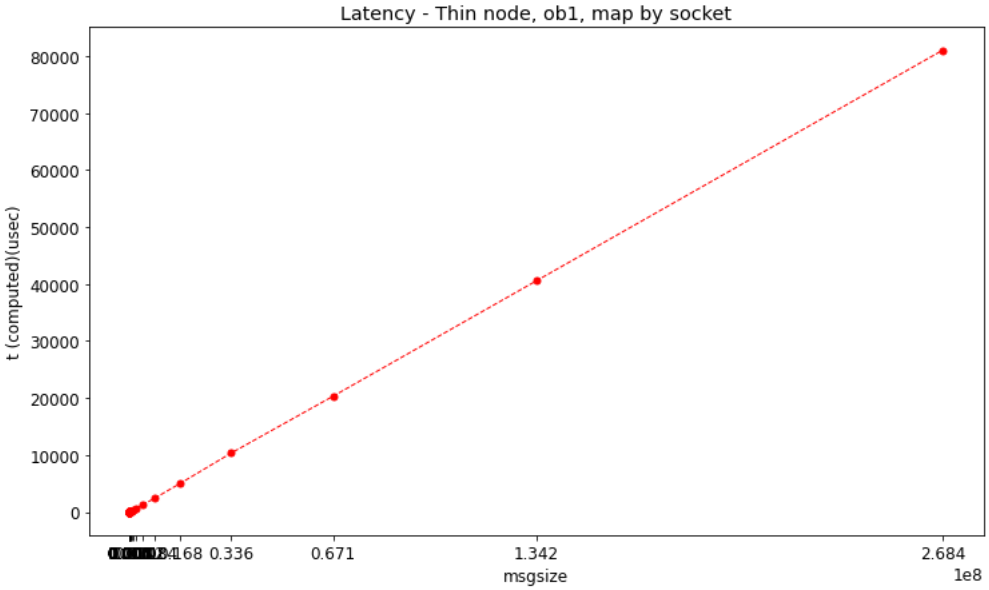

Performance - Thin node, ucx, map by core
Mbytes/sec (computed)
20000
15000
10000
5000
0
0.168
0.336
0.671
1.342
2.684
msgsize
1e8

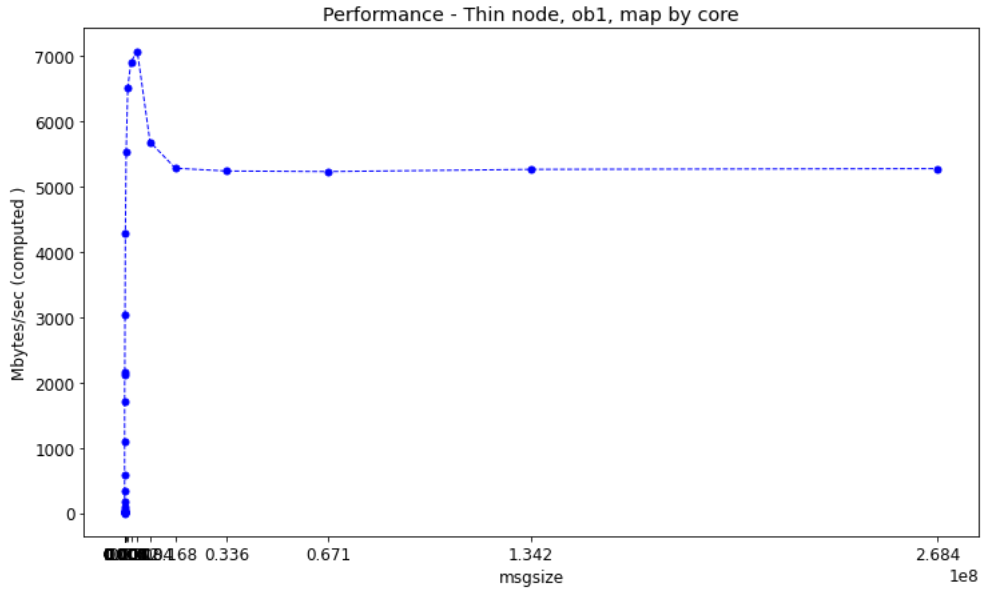

Latency - Thin node, ucx, map by core
t (computed)(usec)
40000
30000
20000
10000
0
0.168
0.336
0.671
1.342
2.684
msgsize
1e8

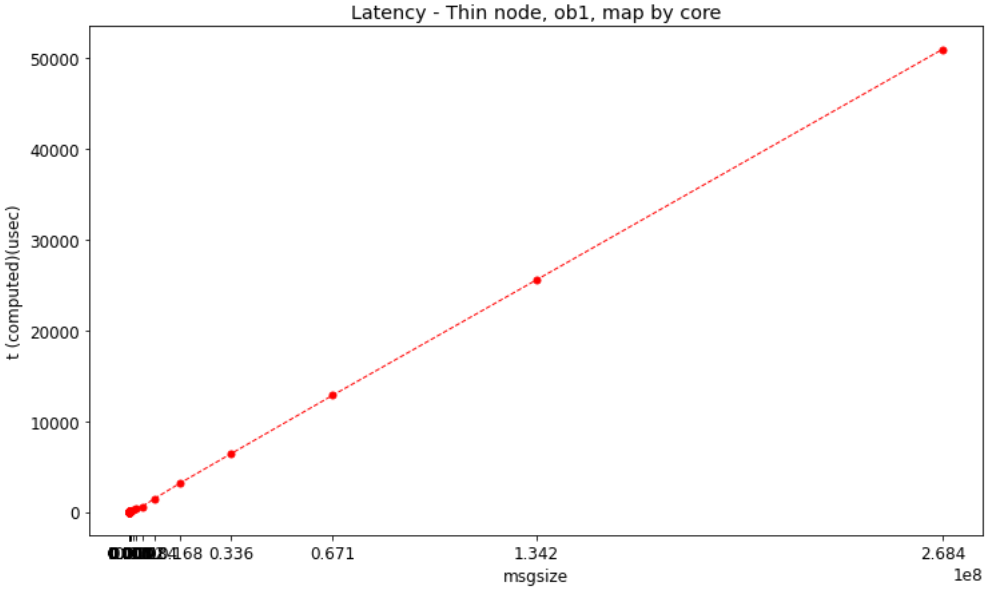

Performance - Thin node, ob1, map by node
Mbytes/sec (computed)
3000
2500
2000
1500
1000
500
0
0.168
0.336
0.671
1.342
2.684
msgsize
1e8

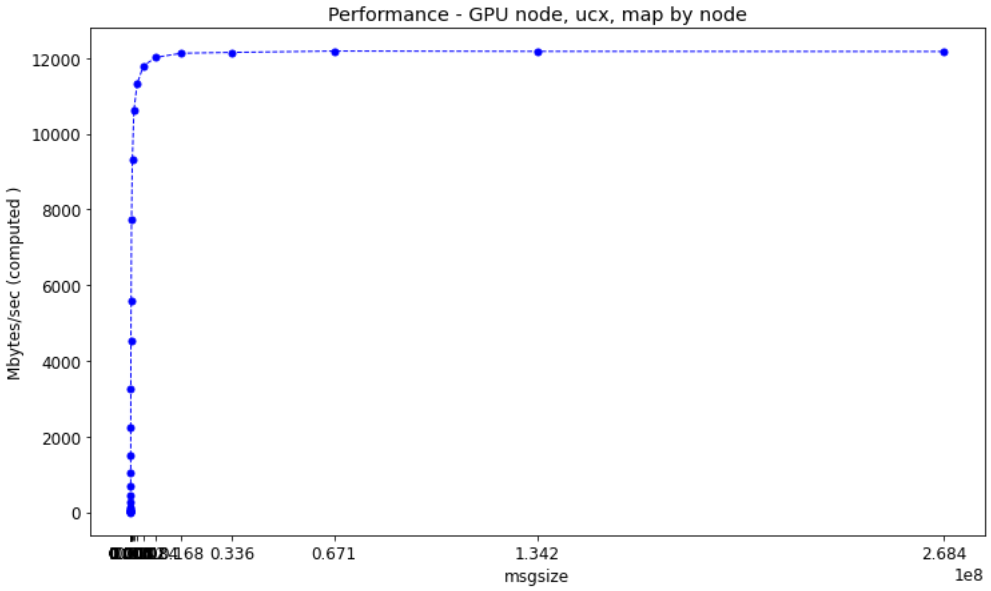

Latency - Thin node, ob1, map by node
t (computed)(usec)
100000
80000
60000
40000
20000
0
0.168
0.336
0.671
1.342
2.684
msgsize
1e8

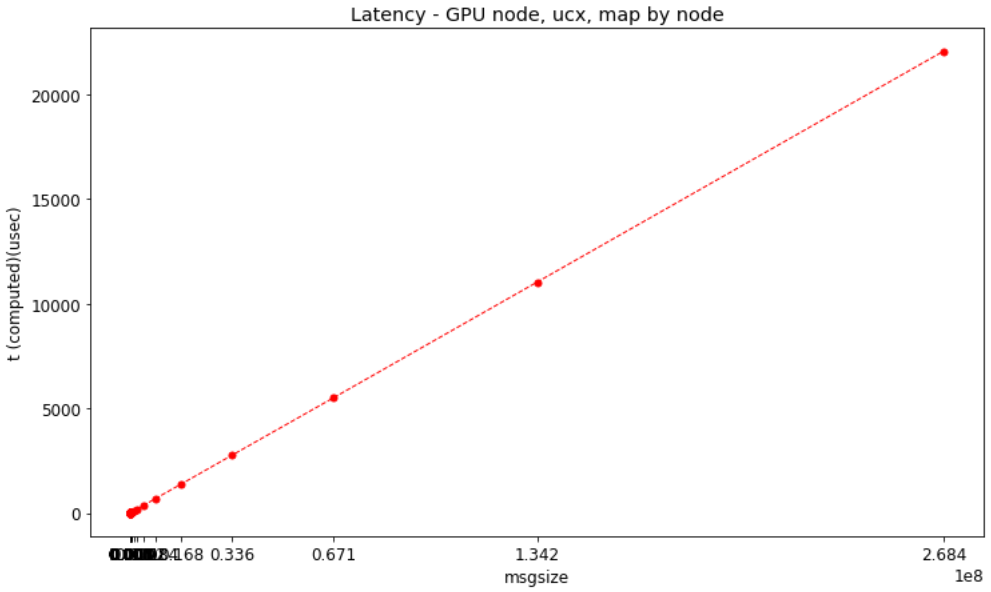

Performance - Thin node, ob1, map by socket
Mbytes/sec (computed)
3500
3000
2500
2000
1500
1000
500
0
0.336
0.671
1.342
2.684
msgsize
1e8

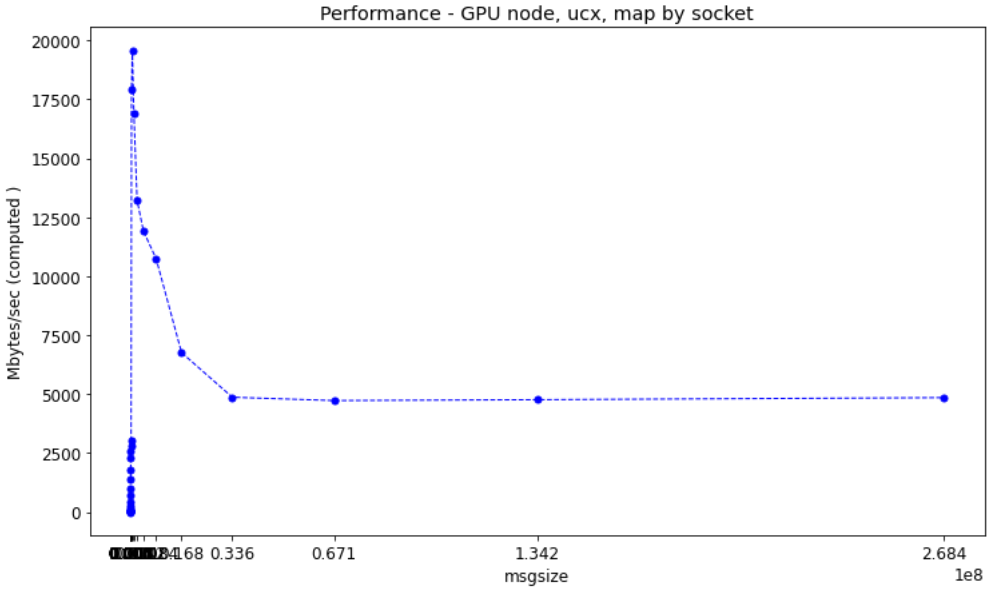

Latency - Thin node, ob1, map by socket
t (computed)(usec)
80000
70000
60000
50000
40000
30000
20000
10000
0
0.336
0.671
1.342
2.684
msgsize
1e8

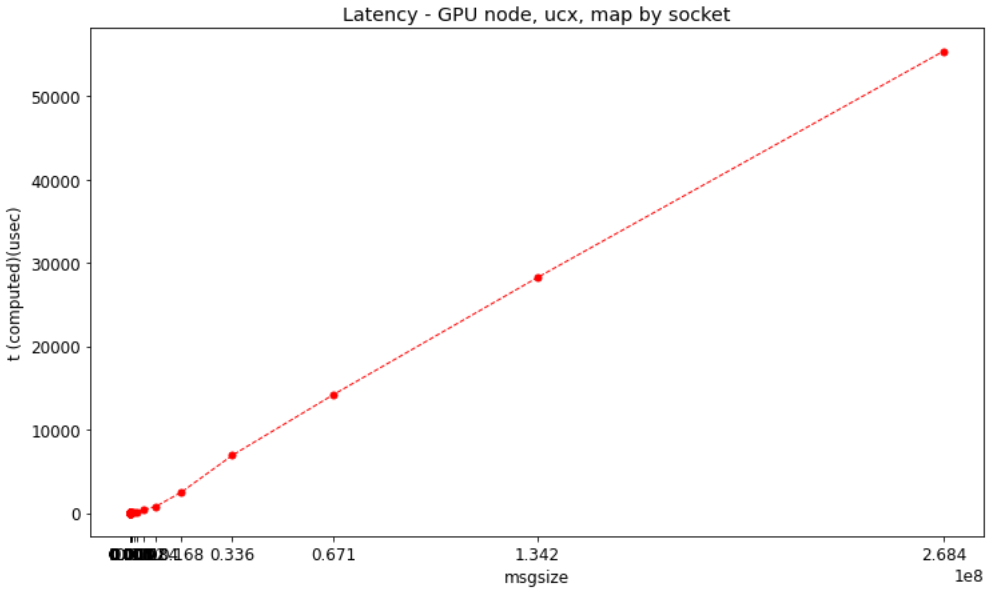

Performance - Thin node, ob1, map by core
Mbytes/sec (computed)
7000
6000
5000
4000
3000
2000
1000
0
0.336
0.671
1.342
2.684
msgsize
1e8

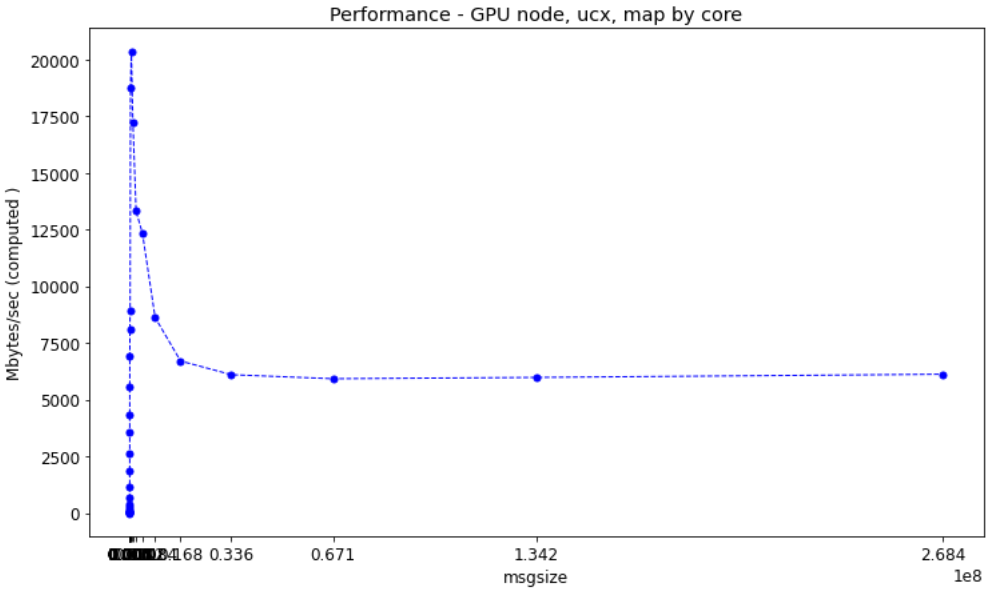

Latency - Thin node, ob1, map by core
t (computed)(usec)
50000
40000
30000
20000
10000
0
0.336
0.671
1.342
2.684
msgsize
1e8

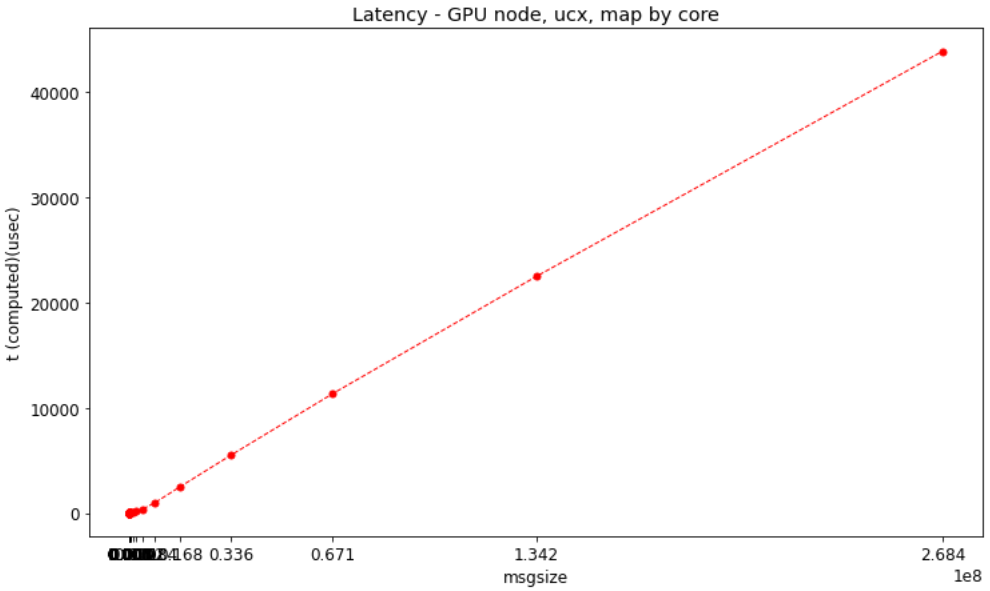

Performance - GPU node, ucx, map by node
Mbytes/sec (computed)
12000
10000
8000
6000
4000
2000
0
0.336
0.671
1.342
2.684
msgsize
1e8

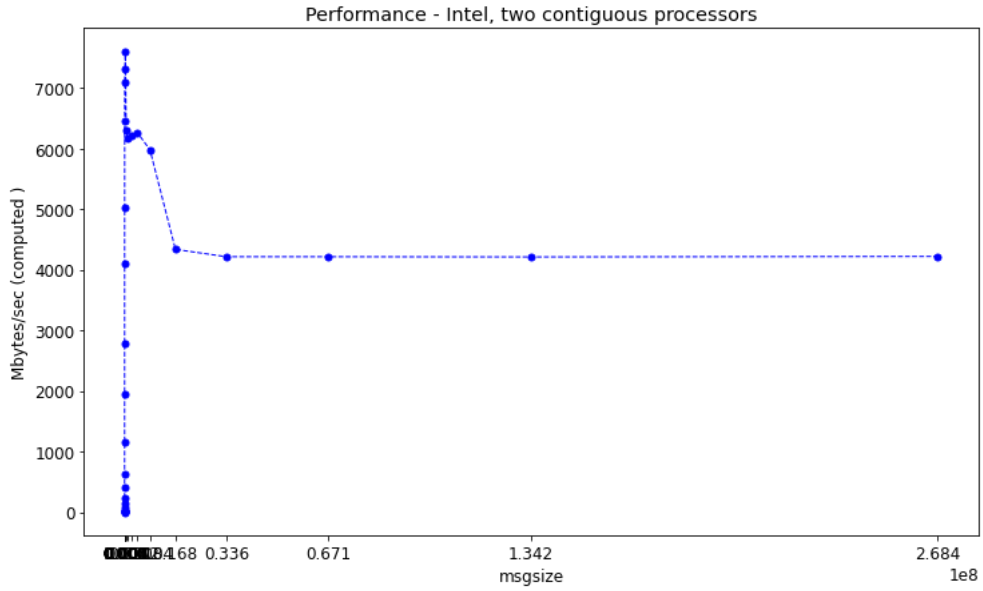

Latency - GPU node, ucx, map by node
t (computed)(usec)
20000
15000
10000
5000
0
0.336
0.671
1.342
2.684
msgsize
1e8


Performance - GPU node, ucx, map by socket
Mbytes/sec (computed)
20000
17500
15000
12500
10000
7500
5000
2500
0
0.168
0.336
0.671
1.342
2.684
msgsize
1e8

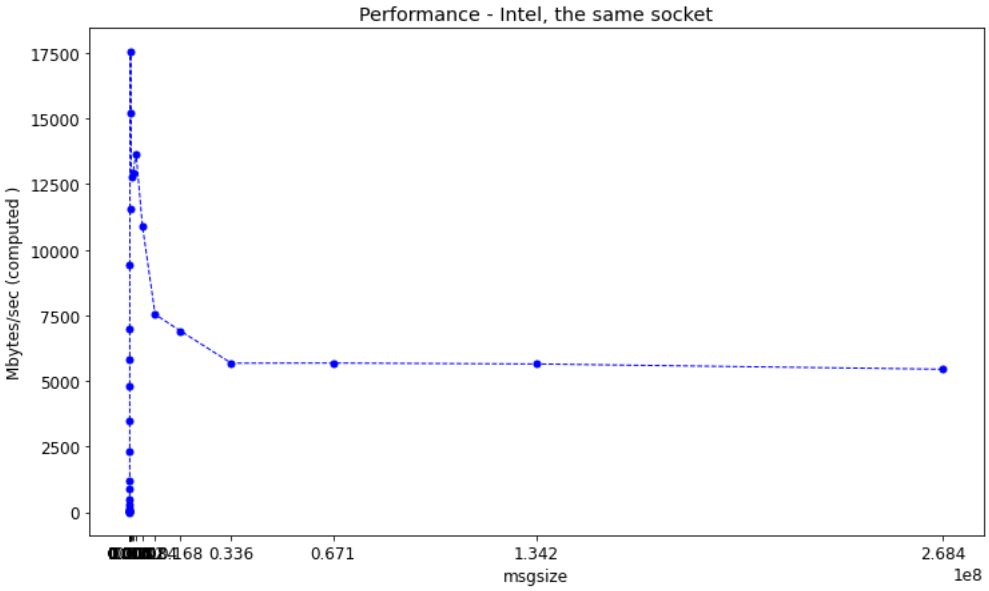

Latency - GPU node, ucx, map by socket
t (computed)(usec)
50000
40000
30000
20000
10000
0
0.168
0.336
0.671
1.342
2.684
msgsize
1e8

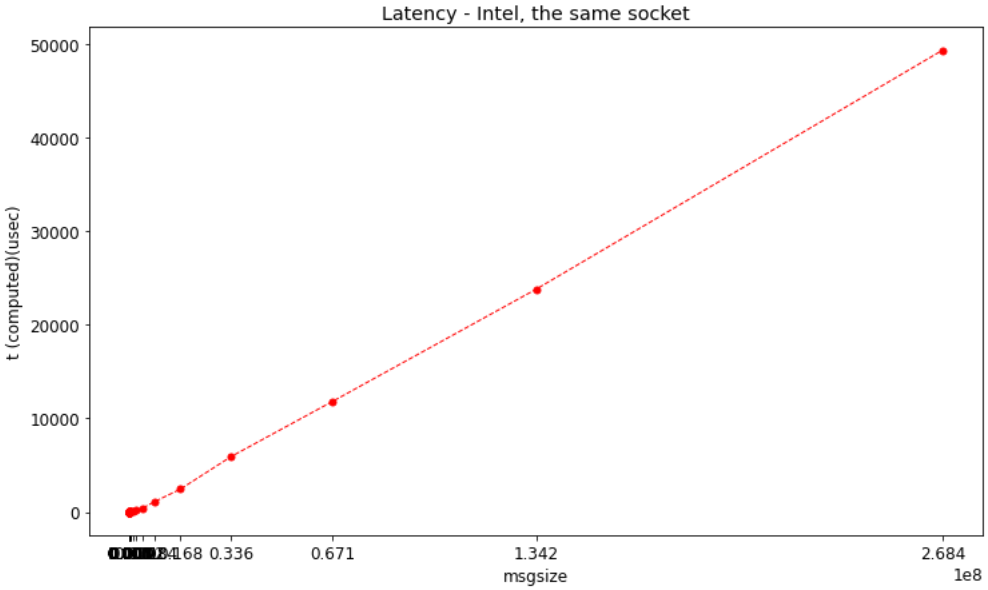

Performance - GPU node, ucx, map by core
Mbytes/sec (computed)
20000
17500
15000
12500
10000
7500
5000
2500
0
0.168
0.336
0.671
1.342
2.684
msgsize
1e8


Latency - GPU node, ucx, map by core
t (computed)(usec)
40000
30000
20000
10000
0
0.168
0.336
0.671
1.342
2.684
msgsize
1e8


Performance - Intel, two contiguous processors
Mbytes/sec (computed)
7000
6000
5000
4000
3000
2000
1000
0
0.336
0.671
1.342
2.684
msgsize
1e8


Latency - Intel, two contiguous processors
t (computed)(usec)
60000
50000
40000
30000
20000
10000
0
0.336
0.671
1.342
2.684
msgsize
1e8


Performance - Intel, the same socket
Mbytes/sec (computed)
17500
15000
12500
10000
7500
5000
2500
0
0.336
0.671
1.342
2.684
msgsize
1e8


Latency - Intel, the same socket
t (computed)(usec)
50000
40000
30000
20000
10000
0
0.336
0.671
1.342
2.684
msgsize
1e8

In all the plots, the (computed) latency increases by enlarging message size that makes sense.

By looking at the plots, we can see there is a significant drop in the performance on a Thin node by using UCX when mapping by core and by socket; while there is a slight reduction when mapping by node. However; in general, the performance is smaller when mapping by node in this case.
On the other hand, it is shown that performance is improved a lot using OB1 framework instead of UCX on the Thin node. Moreover, mapping by socket has the least performance declining in this case. But mapping by core has a higher performance in general compared to the other two mapping methods.

By using GPU node and UCX, the performance plot strictly decreases for larger message sizes when mapping by socket and by core. While it slightly goes down when mapping by node. The behavior is more or less similar to Thin node with UCX for all the three cases but GPU node works a bit better than Thin node when mapping by node and the performance is more stable at higher message sizes.

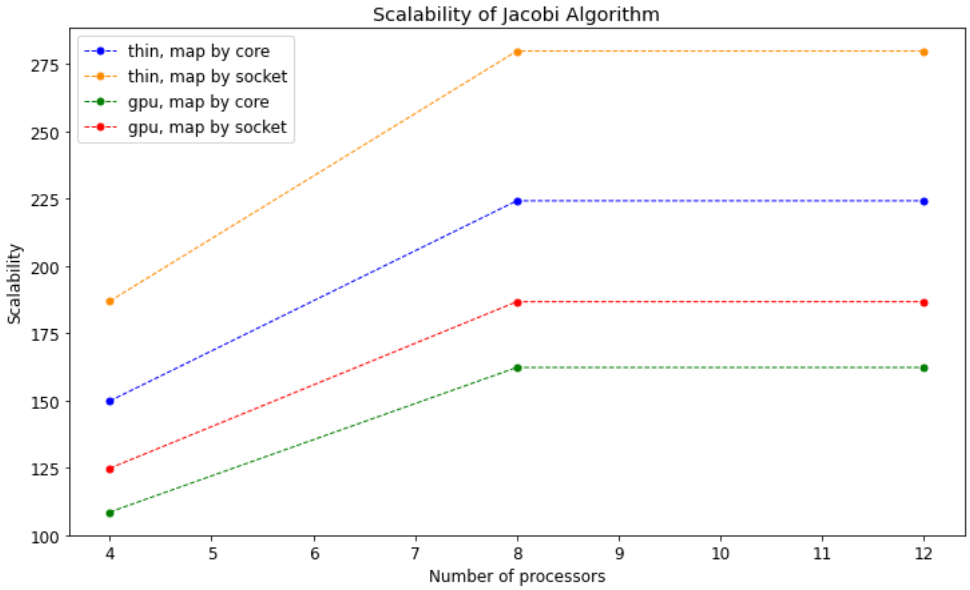
IntelMPI library gives a smaller performance comparing to openMPI when using the same socket. If we don't enforce program to use two continuous processors, the performance is much better.

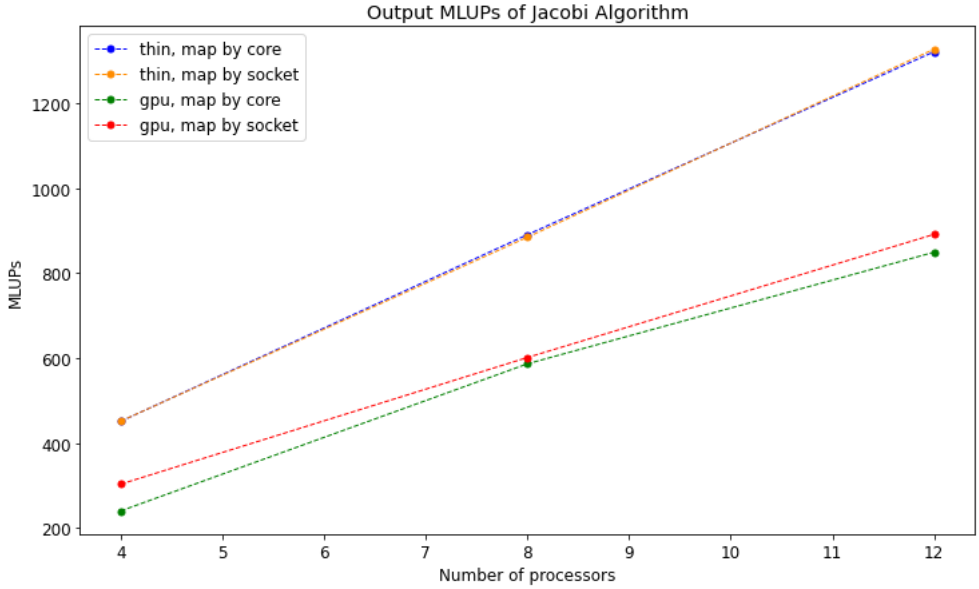
Section 3

The following table shows all the runs of Jacobi program using Thin and GPU nodes and computed scalability.

Run	Sequential time	N	Nx	Ny	Nz	K	C(L,N)	Latency (sec)	Bandwidth (Mb/sec)	Tc(L,N) (sec)	P(L,N)	P(1)*N/P(L,N)	MLUPs
thin serial	15.06	1	1	1	1	0	0.00	0.00000117	20562.29	0.00	114737124.10	1.00	112.72
gpu serial	21.80	1	1	1	1	0	0.00	0.00000115	16458.58	0.00	79269440.79	1.00	78.32
thin4_map_by_core	15.06	4	2	2	1	2	46080000.00	0.00000117	20562.29	2241.00	3063753.67	149.80	451.08
thin8_map_by_core	15.06	8	2	2	2	3	69120000.00	0.00000117	20562.29	3361.49	4094115.11	224.20	890.79
thin12_map_by_core	15.06	12	3	2	2	3	69120000.00	0.00000117	20562.29	3361.49	6141172.67	224.20	1321.84
thin4_map_by_socket	15.06	4	2	2	1	2	46080000.00	0.00000429	16458.58	2799.76	2455577.89	186.90	451.36
thin8_map_by_socket	15.06	8	2	2	2	3	69120000.00	0.00000429	16458.58	4199.63	3279953.59	279.85	884.80
thin12_map_by_socket	15.06	12	3	2	2	3	69120000.00	0.00000429	16458.58	4199.63	4919930.38	279.85	1327.91
gpu4_map_by_core	21.80	4	2	2	1	2	46080000.00	0.00000115	19664.1	2343.36	2922429.11	108.50	239.94
gpu8_map_by_core	21.80	8	2	2	2	3	69120000.00	0.00000115	19664.1	3515.04	3908580.29	162.25	586.52
gpu12_map_by_core	21.80	12	3	2	2	3	69120000.00	0.00000115	19664.1	3515.04	5862870.44	162.25	849.43
gpu4_map_by_socket	21.80	4	2	2	1	2	46080000.00	0.00000404	17077.06	2698.36	2541030.90	124.78	303.02
gpu8_map_by_socket	21.80	8	2	2	2	3	69120000.00	0.00000404	17077.06	4047.54	3397115.93	186.67	601.38
gpu12_map_by_socket	21.80	12	3	2	2	3	69120000.00	0.00000404	17077.06	4047.54	5095673.89	186.67	891.59
twothin12_map_by_node	15.06	12	3	2	2	3	69120000.00	0.00000246	8614.58	8023.61	2579532.15	533.76	1336.30
twothin24_map_by_node	15.06	24	4	3	2	3	69120000.00	0.00000246	8614.58	8023.61	5159064.30	533.76	2637.17
twothin48_map_by_node	15.06	48	4	4	3	3	69120000.00	0.00000246	8614.58	8023.61	10318128.60	533.76	4144.50
onegpu12_map_by_socket	21.80	12	3	2	2	3	69120000.00	0.00000404	17077.06	4047.54	5095673.89	186.67	896.38
onegpu24_map_by_socket	21.80	24	4	3	2	3	69120000.00	0.00000404	17077.06	4047.54	10191347.78	186.67	1684.00
onegpu48_map_by_socket	21.80	48	4	4	3	3	69120000.00	0.00000404	17077.06	4047.54	20382695.55	186.67	2510.70

These plots show the scalability of Jacobi program using Thin and GPU nodes and 4, 12, and 24 processors. We can see that in all the cases the program doesn't scale for larger number of processors, while the output MLUPs of the program increases linearly.





These plots show the scalability of Jacobi program using two Thin and one GPU nodes and 8, 24, and 48 processors. In this case, not only the program doesn't scale but there is a significant gap between the scalability using two Thin nodes and one GPU node with hyperthreading enabled. Also, the output MLUPs of Jacobi program itself doesn't scale in this case; however, the two Thin nodes still work better than a GPU node.