Foundation of High-Performance Computing (2021 - 22)

Assignment 01

Name: Debarshi Banerjee

Program: Master of High-Performance Computing

Affiliated Institutions: ICTP and SISSA

Contact details: banerjee@ictp.it

Contents -

1. Section 1: MPI programming
   1.1. Ring Communication
   1.2. Sum of 2 3-D matrices for various topologies
2. Section 2: Measure MPI point to point performance
3. Section 3: Compare performance observed against performance model for Jacobi solver

Section 1:

1.1 Ring Communication:

In the `ring` folder in Section 1, the main program is written in `C++` in the file `ring.cc`

1.1.1 Running the Program:

The `run.sh` file compiles and executes the code. It requests for `1 thin node` in the `dssc queue`, and 24 processors in that node.
We can submit the script as `qsub run.sh` from the login node on Orfeo.
This script loads `OpenMPI-4.1.1` and then compiles and runs `ring.cc` for the number of processors (`numproc`) from 2 to 24. It then stores the execution time (which is the output of the program) and prints it in `final.csv`, along with the corresponding `numproc`.

1.1.2 Details behind `ring.cc`:

The `ring.cc` file defines a structure where each processor sends 1 message in each direction in a simple ring topology. The original messages in the first iteration are sent with `tag = 10*rank_of_processor` and at every subsequent iteration we keep track of the tags associated with a particular received message. We make use of `MPI_ANY_TAG` and `status.MPI_TAG` to keep track of the tags of the messages that are received. Next, we modify the message by adding/subtracting the rank of the processor from that message (depending on which direction we received it from). Finally, we forward the modified message to the next processor in the chain, passing along the original tag that we had received for that message.
This process is repeated until a message with the original tag (`10*rank`) is received back at any given processor from either direction.
The entire ring structure is processed 1 million times, so as to allow us to get a good representation for an average time of execution. The time of execution of the program is calculated as the average time taken for the master processor (`rank=0`) to complete the ring. We also write the relevant text (as asked in the question) to a file `ring.dat` which is later renamed to `ring_numproc.dat` (where, `numproc=2,24`) by the `run.sh` script.

1.1.3 Explanation of other files and folders:

The `final.csv` file contains the processor counts (`numproc`) and corresponding execution time.
The `log_files` folder has the error (`run.sh.e*`), output files (`run.sh.o*`) generated by `PBSPRO`, `fit.log` - the graph fitting file generated by `gnuplot`, and the `ring.log` file which contains log output generated by the `run.sh` script.
The `output` folder has files with names of the format `ring_numproc.dat`, where `numproc=2,24`, with the output text corresponding to the question. An example -

```
I am processor 3 and I have received 10 messages.
My final messages have tags 30 from the left, and 30 from the right.
At least one of them should have the tag 30 to fulfil the condition of
receiving back the original message.
```

```
They have values +10 from the left, and -10 from the right.
The total time taken (not including the time taken for writing to file and
printing on-screen) is 5.121e-06 seconds.
```

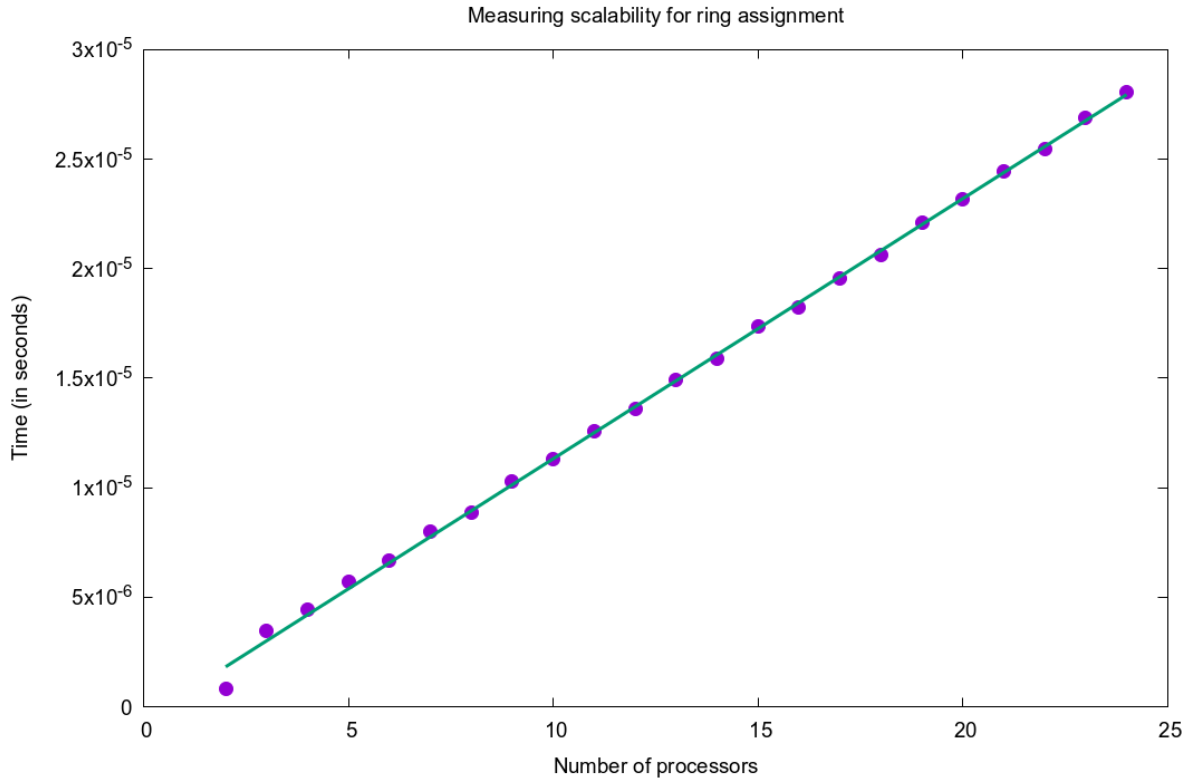
The `helper_scripts` folder has a version of `run.sh` suitable for my laptop (`ring-laptop.sh`), and another for interactive usage on Orfeo (`run-interactive.sh`). It also has a copy of the gnuplot script (`plot.gp`) that is used.

1.1.4 Scalability and Plots:

As asked in the question, I have also produced the scalability curves for this problem. Here, we measure the number of processors (X-axis) vs average time of execution (Y-axis). We use a simple linear model for the scalability curve.
The final image - `ring-scaling.png`, is generated by `plot.gp` - a `gnuplot` script, using the data in `final.csv`. The details of the curve fitting, also done by this same script, are stored in `fit.log` automatically.
The linear model for the fitting has the following parameters -

```
y = m.x + c, m = 1.18759e-06, c = -5.51635e-07
```

The linear model fits the data very well as can be seen in the graph below -



1.2 Sum of 2 3-D matrices for various topologies:

In the `sum3Dmatrix` folder in Section 1, the main program is written in `C++` in `sum3Dmatrix.cc`

1.2.1 Running the program:

The `run.sh` file compiles and executes the code. It requests for `1 thin node` in the `dssc queue`, and 24 processors in that node.
We can submit the script as `qsub run.sh` from the login node on Orfeo.
This script loads `OpenMPI-4.1.1` and then compiles and runs `sum3Dmatrix.cc` for all the topologies possibilities (3D, 2D, 1D) that are stored line-by-line in `matrix.in` - the input file.
The program is executed using all 24 processors on the node. The details of that topology and the corresponding execution time of the program are stored in `final.dat` and the log output is stored in `matrix.log`.

1.2.2 Details behind sum3Dmatrix.cc:

We build a cartesian grid corresponding to a certain topology using periodic boundary conditions (`period = 1 OR TRUE`) for all dimensions, and allowing MPI to reorder the processes if it deems it to be necessary (`reorder = 1 OR TRUE`). The topologies are read from the command-line argument used to run the program in the following format -

```
./sum3Dmatrix.x size_1 size_2 size_3 number_of_dimensions dimension_1
dimension_2 dimension_3
```

The appropriate input data is stored in this format in `matrix.in` which serves as an input file by `run.sh` which executes the whole program. If the total number of elements in the matrix (`size_tot = size_1 . size_2 . size_3`) are not a multiple of the number of available processors (`numproc`), the program increases the size of the matrix till it fits.

We allocate 3 matrices as an 1-D array of total size (`size_tot`). Depending on how we access the elements, this can be regarded as a flattened 3-D matrix -

```
matrix[k + (j * size_3) + (i * size_2 * size_3)], where i, j, k are counters
corresponding to size_1, size_2, size_3 respectively.
```

The first 2 of these matrices are initialized using random numbers by the master process (`rank = 0`). For initialization, we use `rand()`, but there is a commented out section which can also be uncommented to use Mersenne Twisters -

```
std::random_device rd;
std::mt19937_64 gen(rd());
std::uniform_real_distribution<> dis(INT_MIN, INT_MAX);
```

The latter is the recommended way to generate random numbers in C++, but is significantly slower, hence, we use `rand()`.
We divide the matrices into blocks of `size_div = size_tot / numproc`.
Then each block is sent using `MPI_Scatter()` to each processor. Here, the 2 matrices are summed up in the 3rd matrix.
Then, we use `MPI_Gather()` to receive them all back at the master processor (`rank = 0`).
Finally, we calculate the total walltime of this process and the entire process is repeated 50 times

(using `ITER` - a macro defined at the beginning of the code), and we report the average walltime at the end. Since the running script inputs every topology, one by one, this way, we get 50 iterations and the average walltime is quite representative.

1.2.3 Explanation of other files and folders:

The input file - `matrix.in`, the log file - `matrix.log`, the main output file - `final.dat` are all stored in the base folder. The formatted result files - `results.csv` and `results.png` (the latter of which is attached in this document to show the full results) are also present here. These are not generated automatically from `final.dat`, rather they were created using some `vim` and `imagemagick` tricks.
The `log_files` folder has the error (`run.sh.e*`), output files (`run.sh.o*`) generated by `PBSPRO`, and matrix.log - the log file generated by the execution script in case of errors.
The `other_version` folder has the same data as the base folder, but a slightly different program is used for it. Explanation is given below.

1.2.4 Other Version:

In this version, the overall same logic of the main program is used. However, instead of allocating the matrix as a flattened 1-D array using `new` (in C++, similar to `malloc` in C), we declare the matrix using the standard `[ ]` operator notation. For example, instead of `double matrix = new double[n]`, we do `double matrix[n1][n2][n3]`.
The rest of the matrix decomposition is done similarly to the main program, where we `MPI_Scatter` this by `n / numproc` elements at a time then `MPI_Gather` it similarly again.
The overall results show that this process is slightly slower across the board than the main version we have implemented. This can easily be explained as here, the elements are not contiguous in memory, unlike in the main version.

1.2.5 Discussion of Results:

There are a total of 117 possible combinations - 1D, 2D, 3D - of 2400x100x100, 1200x200x100, 800x300x100. The full data can be seen in `final.dat` (uncleaned data), `results.csv` (formatted), `results.png`. The latter is attached below (next page).
For 2400x100x100, we find that the best performance is achieved by 4x3x2, 6x2x2, 24x1. Even the flat 1D topology is very close to the best performance. Clearly whatever benefit there exists in a higher dimensional topology, at least for matrices of this size, the benefit of being contiguous in memory counteracts this, and so, the 1D performance is very good. The worst performance is seen in 2x6x2,1x24x1, 2x1x12.
For 1200x200x100, the best performance is seen in 2x6x2, 2x2x6, 1x8x3. The worst performance is seen for 1x24, 3x8, 8x3x1. Generally, for this the best and worst performance is a bit surprising than the other two distributions. The 1D performance is also not as high. The 2D performance is amongst the worst. Relative to the other two, this also has overall the worst performance. Since the data is averaged over 50 runs, it is possible that there were some anomalous results which took significantly longer than expected and affected the performance.
For 800x300x100, the best performance is seen in 24x1x1, 3x8x1, 12x1x2. This also (like in 2400x100x100) shows very good 1D and reasonable 2D performance. It shows the same benefits of contiguous memory in 1D clearly.
Generally, we see good results for 1D and topology which match the data distribution very well. Practically, however, the process is so fast (~1 second), differences are small, and matrix sizes are small, so even with 50 times averaging it is reasonable to expect significant noise in our data.

Size 1	Size 2	Size 3	ndims	Dim 1	Dim 2	Dim 3	time (sec)	Size 1	Size 2	Size 3	ndims	Dim 1	Dim 2	Dim 3	time (sec)
2400	100	100	3	4	3	2	0.969181	1200	200	100	3	1	24	1	0.97015
2400	100	100	3	4	2	3	0.970497	1200	200	100	3	1	1	24	0.972007
2400	100	100	3	3	4	2	0.971159	800	300	100	3	4	3	2	0.969921
2400	100	100	3	3	2	4	0.969366	800	300	100	3	4	2	3	0.970466
2400	100	100	3	2	4	3	0.971673	800	300	100	3	3	4	2	0.972984
2400	100	100	3	2	3	4	0.971329	800	300	100	3	3	2	4	0.97146
2400	100	100	3	6	2	2	0.969201	800	300	100	3	2	4	3	0.970707
2400	100	100	3	2	6	2	1.26325	800	300	100	3	2	3	4	0.96998
2400	100	100	3	2	2	6	0.97197	800	300	100	3	6	2	2	0.972449
2400	100	100	3	6	4	1	0.972785	800	300	100	3	2	6	2	0.969515
2400	100	100	3	6	1	4	0.971021	800	300	100	3	2	2	6	0.970386
2400	100	100	3	4	6	1	0.972294	800	300	100	3	6	4	1	0.970055
2400	100	100	3	4	1	6	0.969767	800	300	100	3	6	1	4	0.972255
2400	100	100	3	1	6	4	0.970017	800	300	100	3	4	6	1	0.971162
2400	100	100	3	1	4	6	0.972067	800	300	100	3	4	1	6	0.97183
2400	100	100	3	8	3	1	0.97018	800	300	100	3	1	6	4	0.971316
2400	100	100	3	8	1	3	0.970472	800	300	100	3	1	4	6	0.970782
2400	100	100	3	3	8	1	0.970825	800	300	100	3	8	3	1	0.97041
2400	100	100	3	3	1	8	0.970386	800	300	100	3	8	1	3	0.970522
2400	100	100	3	1	8	3	0.971822	800	300	100	3	3	8	1	0.969432
2400	100	100	3	1	3	8	0.970121	800	300	100	3	3	1	8	0.971875
2400	100	100	3	12	2	1	0.970233	800	300	100	3	1	8	3	0.969602
2400	100	100	3	12	1	2	0.969581	800	300	100	3	1	3	8	0.972338
2400	100	100	3	2	12	1	0.970694	800	300	100	3	12	2	1	0.969786
2400	100	100	3	2	1	12	0.973747	800	300	100	3	12	1	2	0.969472
2400	100	100	3	1	12	2	0.971296	800	300	100	3	2	12	1	0.970661
2400	100	100	3	1	2	12	0.971596	800	300	100	3	2	1	12	0.970868
2400	100	100	3	24	1	1	0.970785	800	300	100	3	1	12	2	1.00332
2400	100	100	3	1	24	1	0.99714	800	300	100	3	1	2	12	0.970793
2400	100	100	3	1	1	24	0.971151	800	300	100	3	24	1	1	0.96832
1200	200	100	3	4	3	2	0.973484	800	300	100	3	1	24	1	0.969894
1200	200	100	3	4	2	3	0.970409	800	300	100	3	1	1	24	0.970061
1200	200	100	3	3	4	2	0.972187	2400	100	100	2	6	4	0	0.970808
1200	200	100	3	3	2	4	0.972015	2400	100	100	2	4	6	0	0.969324
1200	200	100	3	2	4	3	0.971465	2400	100	100	2	8	3	0	0.971032
1200	200	100	3	2	3	4	0.970249	2400	100	100	2	3	8	0	0.97031
1200	200	100	3	6	2	2	0.969817	2400	100	100	2	12	2	0	0.970795
1200	200	100	3	2	6	2	0.969222	2400	100	100	2	2	12	0	0.970382
1200	200	100	3	2	2	6	0.96942	2400	100	100	2	24	1	0	0.969218
1200	200	100	3	6	4	1	0.97082	2400	100	100	2	1	24	0	0.96992
1200	200	100	3	6	1	4	0.973288	1200	200	100	2	6	4	0	0.972073
1200	200	100	3	4	6	1	0.971359	1200	200	100	2	4	6	0	0.970673
1200	200	100	3	4	1	6	0.969713	1200	200	100	2	8	3	0	0.971841
1200	200	100	3	1	6	4	0.971887	1200	200	100	2	3	8	0	0.988769
1200	200	100	3	1	4	6	0.97064	1200	200	100	2	12	2	0	0.970696
1200	200	100	3	8	3	1	0.97385	1200	200	100	2	2	12	0	0.97147
1200	200	100	3	8	1	3	0.970378	1200	200	100	2	24	1	0	0.970148
1200	200	100	3	3	8	1	0.973189	1200	200	100	2	1	24	0	1.26694
1200	200	100	3	3	1	8	0.973171	800	300	100	2	6	4	0	0.973838
1200	200	100	3	1	8	3	0.969685	800	300	100	2	4	6	0	0.969772
1200	200	100	3	1	3	8	0.970405	800	300	100	2	8	3	0	0.970237
1200	200	100	3	12	2	1	0.973555	800	300	100	2	3	8	0	0.971431
1200	200	100	3	12	1	2	0.971399	800	300	100	2	12	2	0	1.02391
1200	200	100	3	2	12	1	0.970694	800	300	100	2	2	12	0	0.971183
1200	200	100	3	2	1	12	0.971773	800	300	100	2	24	1	0	0.971685
1200	200	100	3	1	12	2	0.970124	800	300	100	2	1	24	0	0.971865
1200	200	100	3	1	2	12	0.969352	2400	100	100	1	24	0	0	0.969444
1200	200	100	3	24	1	1	0.970254	1200	200	100	1	24	0	0	0.970527
								800	300	100	1	24	0	0	0.970913

Section 2: Measure MPI point to point performance

2.1 Explanation of Files and Directory Structure:

In the base folder there are 3 subfolders, `scripts`, `gpu` and `thin`, which contain the data of the `GPU` nodes and `THIN` nodes on Orfeo respectively. Inside `scripts`, there is a file `data_analysis.py` (and its `ipynb` version for interactive usage) which I have written to help plot the relevant graphs and calculate the relevant data. There is also `linear-data_analysis.py` which generates the latency curves without using logarithmic scale along the X-axis (message size). There are also 2 `plot.gp` scripts which need to be run using `gnuplot` with a file containing the appropriate data. One of them is used for generating comparisons across `core`, `socket`, and `node`. The other is used for generating comparisons across `THIN` and `GPU` nodes.
Certain aspects of the directory structure are symmetric across both `thin` and `gpu`.
We have `IMB-MPI1_gcc` and `IMB-MPI1_intel` - the 2 executables compiled using `gcc` and `icc` (Intel) respectively. We use the `gcc` version with `OpenMPI` and the Intel version with `IntelMPI`. The base folder also has `run_thin.sh` or `run_gpu.sh` to run the relevant jobs and collect the data.
There's a `backup` folder which contains results from older runs for posterity and comparison.
There's a `log` folder which contains the error and outputs of the script which submits the job.
The data folder has the full data of all 10 runs for each one of `IntelMPI` and `OpenMPI`.
Then there's an `output_thin_sample` OR `output_gpu_sample` folder which has a randomly chosen sample of the multiple runs for us to analyze. Inside this we have subfolders for `OpenMPI` and `IntelMPI` separately - one for `msglog = $default`, the other for `msglog = 28`. There's also a `comparison` folder which has the data that is used for generating the comparison between `nodes`, `sockets`, and `cores` for a couple of different scenarios (`ob1-tcp-ib0`, `ucx`).
There's a folder called `bandwidth-latency-data` which contains files with a summary of the bandwidth and latency data for all `OpenMPI` combinations and all `IntelMPI` combinations.
We also have 2 varieties of each file, one for `msglog = $default`, the other for `msglog = 28`.
Finally, there's a folder called `images_thin` or `images_gpu`. We have 2 subfolders - one for `msglog = $default`, the other for `msglog = 28`. Inside each we have further subfolders for `IntelMPI` and `OpenMPI` and they contain the full set of images for each corresponding data set that we have obtained. A `tree` command output is attached here for clarity:

```
.
├── gpu
│   ├── backup
│   │   └── output_gpu
│   ├── bandwidth-latency-data
│   ├── data
│   │   ├── output_gpu_msglog_28
│   │   └── output_gpu_msglog_default
│   ├── images_gpu
│   │   ├── msglog_28
│   │   │   ├── images_intelmpi_gpu
│   │   │   └── images_openmpi_gpu
│   │   └── msglog_default
│   │       ├── images_intelmpi_gpu
│   │       └── images_openmpi_gpu
│   ├── log
│   └── output_gpu_sample
│       ├── comparison
│       │   ├── ob1_tcp_ib0
│       │   │   └── gpu_thin
│       │   └── ucx
│       │       └── gpu_thin
│       ├── intelmpi-msglog_28
│       │   ├── csv
│       │   └── images
│       ├── intelmpi-msglog_default
│       │   └── csv
│       ├── openmpi-msglog_28
│       │   └── csv
│       └── openmpi-msglog_default
│           └── csv
├── scripts
└── thin
    ├── backup
    ├── bandwidth-latency-data
    ├── data
    │   ├── output_thin_msglog_28
    │   └── output_thin_msglog_default
    ├── images_thin
    │   ├── msglog_28
    │   │   ├── images_intelmpi_thin
    │   │   └── images_openmpi_thin
    │   └── msglog_default
    │       ├── images_intelmpi_thin
    │       └── images_openmpi_thin
    ├── log
    └── output_thin_sample
        ├── comparison
        │   ├── ob1_tcp_ib0
        │   │   └── gpu_thin
        │   └── ucx
        │       └── gpu_thin
        ├── intelmpi-msglog_28
        │   └── csv
        ├── intelmpi-msglog_default
        │   └── csv
        ├── LINEAR-openmpi-msglog_default
        │   └── csv
        ├── openmpi-msglog_28
        │   └── csv
        └── openmpi-msglog_default
            └── csv
```

2.2 Running the benchmark and generating data and plots:

As asked in the question, we run and generate data for both `OpenMPI` and `IntelMPI` for a variety of network options available on Orfeo. The executables - `IMB-MPI1_gcc`, `IMB-MPI1_intel` - have already been compiled with `OpenMPI-4.1.1+GNU-9.3.0` and `Intel-20.4` respectively.

2.2.1 How to run and analyse the data?

To run the benchmark, we use `run_thin.sh` (for `THIN` nodes) or `run_gpu.sh` (for `GPU` nodes). We can submit these using `qsub run_thin.sh`, for example. The data files of 10 runs of each of `IntelMPI` and `OpenMPI` will be generated in a folder called `output_thin` or `output_gpu`. We go inside any of the relevant folders. Then we copy `data_analysis.py` into this folder. Next we do the following -

```
mkdir csv
cp *.csv csv/
python3 data_analysis.py
```

The `*.csv` files in the base folder (not to be confused with the ones we have copied to the `csv/` folder we just created), will now be modified with complete calculations like the question asks for. A sample output (`openmpi_core_ob1_tcp_br0.csv - msglog = 28, THIN`) is shown -

```
# Command used - mpirun -np 2 --report-bindings --map-by core --mca pml ob1
--mca btl self,tcp --mca btl_tcp_if_include br0 ./IMB-MPI1_gcc PingPong
-msglog 28
# Nodes used - ct1pt-tnode009.hpc, ct1pt-tnode010.hpc
# Calculated Latency = 5.522 usec; Calculated Bandwidth = 5190.298 MBytes/sec
#bytes,#repetitions,t[usec],Mbytes/sec,t[usec] computed,Mbytes/sec (computed)
0,1000,5.48,0.0,18.307,0.0
1,1000,5.55,0.18,18.307,0.055
2,1000,5.53,0.36,18.308,0.109
4,1000,5.53,0.72,18.308,0.218
8,1000,5.52,1.45,18.309,0.437
16,1000,5.54,2.89,18.31,0.874
32,1000,5.54,5.77,18.313,1.747
64,1000,5.55,11.53,18.319,3.494
128,1000,5.59,22.88,18.332,6.982
256,1000,5.62,45.54,18.356,13.946
512,1000,5.7,89.76,18.406,27.817
1024,1000,6.36,161.0,18.504,55.338
2048,1000,6.15,333.05,18.702,109.509
4096,1000,7.17,571.48,19.096,214.492
8192,1000,7.62,1074.95,19.885,411.959
16384,1000,9.68,1692.51,21.464,763.331
32768,1000,16.11,2033.97,24.62,1330.924
65536,640,30.24,2166.84,30.934,2118.589
131072,320,40.48,3238.28,43.56,3008.969
262144,160,56.11,4672.06,68.814,3809.474
524288,80,89.62,5850.18,119.32,4393.956
1048576,40,156.01,6721.27,220.333,4759.044
```

```
2097152,20,293.06,7156.15,422.36,4965.324
4194304,10,660.36,6351.55,826.412,5075.319
8388608,5,1609.0,5213.55,1634.517,5132.164
16777216,2,3437.28,4880.96,3250.726,5161.067
33554432,1,6908.11,4857.25,6483.145,5175.641
67108864,1,13041.05,5145.97,12947.983,5182.959
134217728,1,25917.08,5178.74,25877.658,5186.626
268435456,1,51633.98,5198.81,51737.01,5188.461
```

At the end of this process, the directory will also be full of 2 sets of images (`*.jpg`) for each file - one for the latency curve, the other for the bandwidth. They both have the X-axis plotted in log-scale for better representation of data, and the least-squares fitting is also shown in this image. To produce purely linear latency plots (as shown in 2.4.3), please use the file linear-data_analysis.py in `scripts` in the base folder. Instead of the original `data_analysis.py` file, copy this to any of the output folders, follow the steps described here, and you will get linear latency curves if needed.

2.2.2 A look behind the scenes:

The `OpenMPI` section uses roughly the following command with the following parameter choices -

```
mpirun -np 2 --report-bindings --map-by $1 --mca pml $2 --mca btl self,$3
--mca btl_$3_if_include $4 ./IMB-MPI1_gcc PingPong -msglog 28

$1 -> core socket node
$2 -> ob1 ucx
$3 -> tcp vader
$4 -> br0 ib0
```

Care should be taken to ensure that `'node'` and `'vader'` aren't selected together as when using 2 different nodes, we obviously can't use the shared memory approach.
For `IntelMPI`, the following 2 commands were used, one for `socket`/`contiguous`, the other for `node` -

```
Default:
mpiexec -np 2 -env I_MPI_DEBUG 5 -genv I_MPI_FABRICS=$1 -genv
I_MPI_OFI_PROVIDER=$2 -genv I_MPI_PIN_PROCESSOR_LIST 0,$3 ./IMB-MPI1_intel
PingPong -msglog 28

$1 -> shm ofi shm:ofi
$2 -> tcp mlx shm sockets
$3 -> 1 (contiguous) 2 (socket)

For node:
mpiexec -np 2 -ppn=1 -env I_MPI_DEBUG 5 -genv I_MPI_FABRICS=$1 -genv
I_MPI_OFI_PROVIDER=$2 ./IMB-MPI1_intel PingPong -msglog 28

$1 -> ofi shm:ofi
$2 -> tcp mlx sockets
```

Like in `OpenMPI`, here also, `shm` (shared memory approach) can't be used when we use `'node'`.

The `data_analysis.py` script needs the following modules to function properly - `numpy pandas matplotlib scipy scikit-learn`. It reads in the values from the `*.csv` files using `pandas`. Then I do 2 different linear regression fits just to verify that data is matching. One is done using `curve_fit` from `scipy.optimize` and the other is done using `LinearRegression` from `sklearn.linear_model`. The data and fit parameters from both are extremely similar. We calculate the relevant slope (1/bandwidth), intercept (latency), and t (from corresponding message sizes). Since we often get negative latency which does not make any physical sense, I decided to take absolute value of this. When reporting the final latency however, for the sake of accuracy, I use the mean of the time taken to open a communication channel when the message size is 0 - 16 bytes. Since the messages are extremely small in this range, the entire time is basically the latency. Finally all this data is written to the csv files in the requested format and the corresponding images are generated.

2.3 Data:

Here I present data for `msglog = 28` for `OpenMPI` and for `msglog = $default` for `IntelMPI`. The counterparts for both are available in `bandwidth-latency-data` as stated earlier.

2.3.1 OpenMPI (`msglog = 28`):

Mapping	PML	BTL	Network	THIN		GPU	
				Latency (usec)	Bandwidth (Mbytes/sec)	Latency (usec)	Bandwidth (Mbytes/sec)
core	ob1	tcp	br0	5.522	5190.298	4.992	5155.561
core	ob1	tcp	ib0	5.502	5194.299	4.976	5138.107
core	ob1	vader	br0	0.276	4116.389	0.278	4008.606
core	ob1	vader	ib0	0.286	4120.754	0.274	3994.715
core	ucx	vader	br0	0.196	6240.607	0.210	6137.379
core	ucx	vader	ib0	0.198	6238.581	0.212	6146.922
core	ucx	tcp	br0	0.198	6245.204	0.212	6167.449
core	ucx	tcp	ib0	0.200	6236.583	0.214	6150.128
socket	ucx	vader	ib0	0.404	5342.649	0.412	4981.419
socket	ucx	vader	br0	0.406	5072.060	0.418	4961.802
socket	ucx	tcp	br0	0.400	5338.064	0.418	4968.399
socket	ucx	tcp	ib0	0.400	5288.351	0.424	4966.062
socket	ob1	vader	ib0	0.638	3417.72	0.554	3158.508
socket	ob1	tcp	ib0	7.830	3152.645	8.616	3112.658

Mapping	PML	BTL	Network	THIN		GPU	
socket	ob1	vader	br0	0.600	3403.753	0.562	3152.816
socket	ob1	tcp	br0	7.964	3291.030	8.796	3112.127
node	ucx	tcp	ib0	0.986	12180.663	1.362	12177.116
node	ob1	tcp	ib0	12.172	2447.668	13.910	2162.637
node	ucx	tcp	br0	0.994	12174.980	1.358	12172.084
node	ob1	tcp	br0	15.920	2339.065	15.444	2409.718

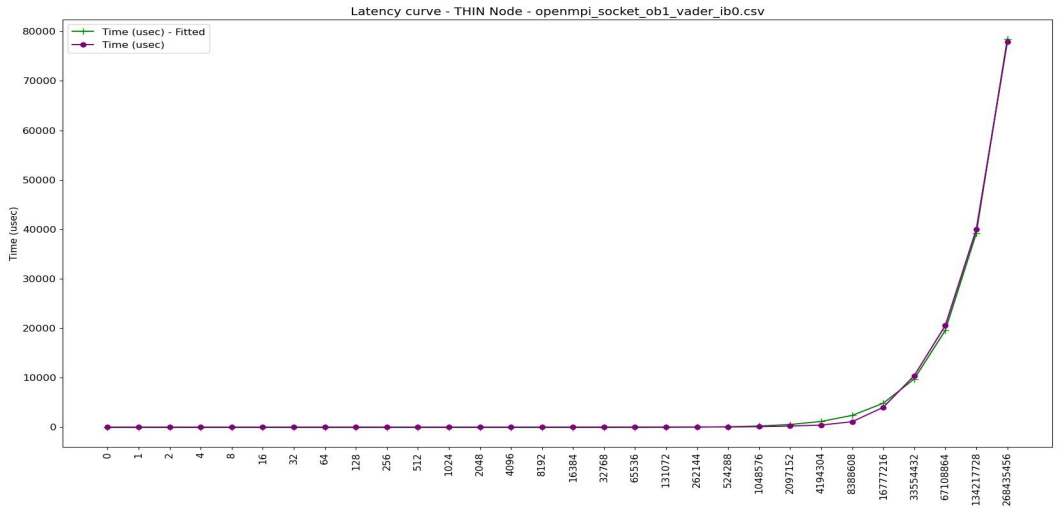
2.3.2 IntelMPI (`msglog = $default`):

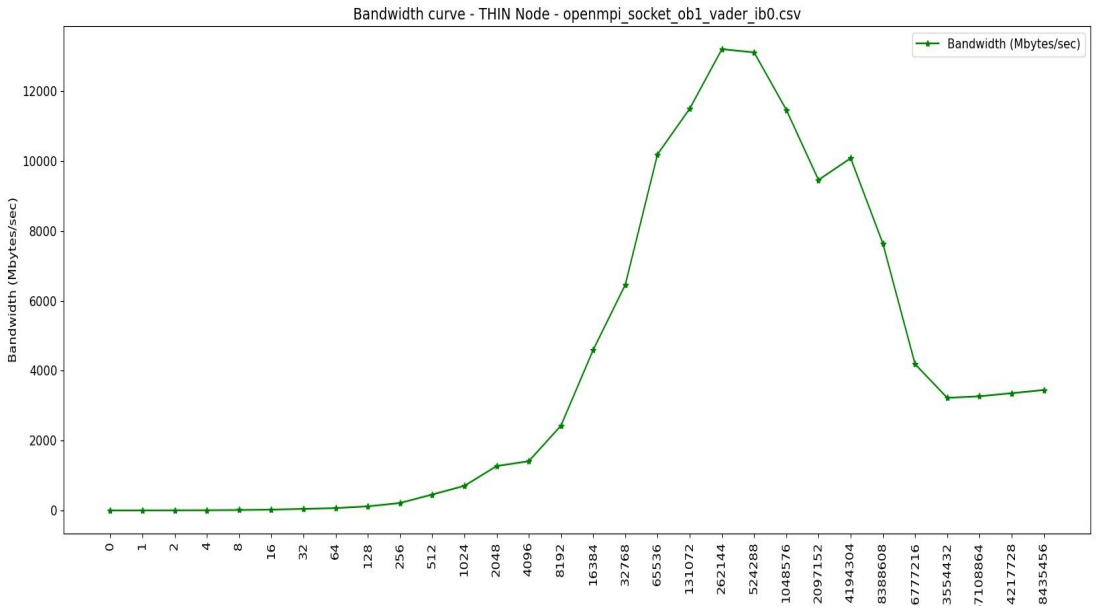
Mapping	Fabrics	OFI	THIN		GPU	
			Latency (usec)	Bandwidth (Mbytes/sec)	Latency (usec)	Bandwidth (Mbytes/sec)
socket	shm	tcp	0.232	11053.512	0.256	10973.759
socket	shm	sockets	0.220	11286.206	0.252	10808.918
socket	shm	shm	0.238	11423.763	0.254	10976.739
socket	shm	mlx	0.222	11355.377	0.258	11019.893
socket	shm:ofi	tcp	0.494	11316.039	0.498	10794.462
socket	shm:ofi	sockets	0.272	11087.357	0.326	10848.204
socket	shm:ofi	shm	0.228	11373.951	0.280	10585.258
socket	shm:ofi	mlx	0.230	11347.445	0.288	10687.020
socket	ofi	sockets	6.414	5477.513	6.474	5779.84
socket	ofi	shm	3.004	10340.651	1.216	8315.604
socket	ofi	mlx	0.240	14942.566	0.294	13124.325
socket	ofi	tcp	4.734	7731.142	5.062	6729.736
contiguous	shm	tcp	0.428	6214.825	0.440	8221.715
contiguous	shm	sockets	0.428	6263.127	0.440	8186.119
contiguous	shm	shm	0.430	6263.249	0.444	8214.473
contiguous	shm	mlx	0.428	6255.335	0.444	8229.531
contiguous	shm:ofi	tcp	0.672	6242.712	0.650	8206.173
contiguous	shm:ofi	sockets	0.460	6255.682	0.500	8206.137

Mapping	Fabrics	OFI	THIN		GPU	
contiguous	shm:ofi	shm	0.432	6255.608	0.468	8235.705
contiguous	shm:ofi	mlx	0.430	6255.717	0.478	8218.565
contiguous	ofi	sockets	9.404	2894.566	9.624	3133.300
contiguous	ofi	shm	1.726	10464.499	1.434	8581.824
contiguous	ofi	mlx	0.444	14716.878	0.490	12295.864
contiguous	ofi	tcp	7.972	3443.55	8.944	3234.755
node	shm:ofi	tcp	8.500	3590.079	8.376	2737.183
node	shm:ofi	sockets	9.650	2300.140	10.288	2201.037
node	shm:ofi	mlx	1.124	11848.885	1.272	11826.065
node	ofi	tcp	8.820	3187.781	8.488	2853.570
node	ofi	sockets	11.906	2968.601	9.874	2725.431
node	ofi	mlx	1.192	11827.264	1.280	11792.357

2.4 Sample Graphs:

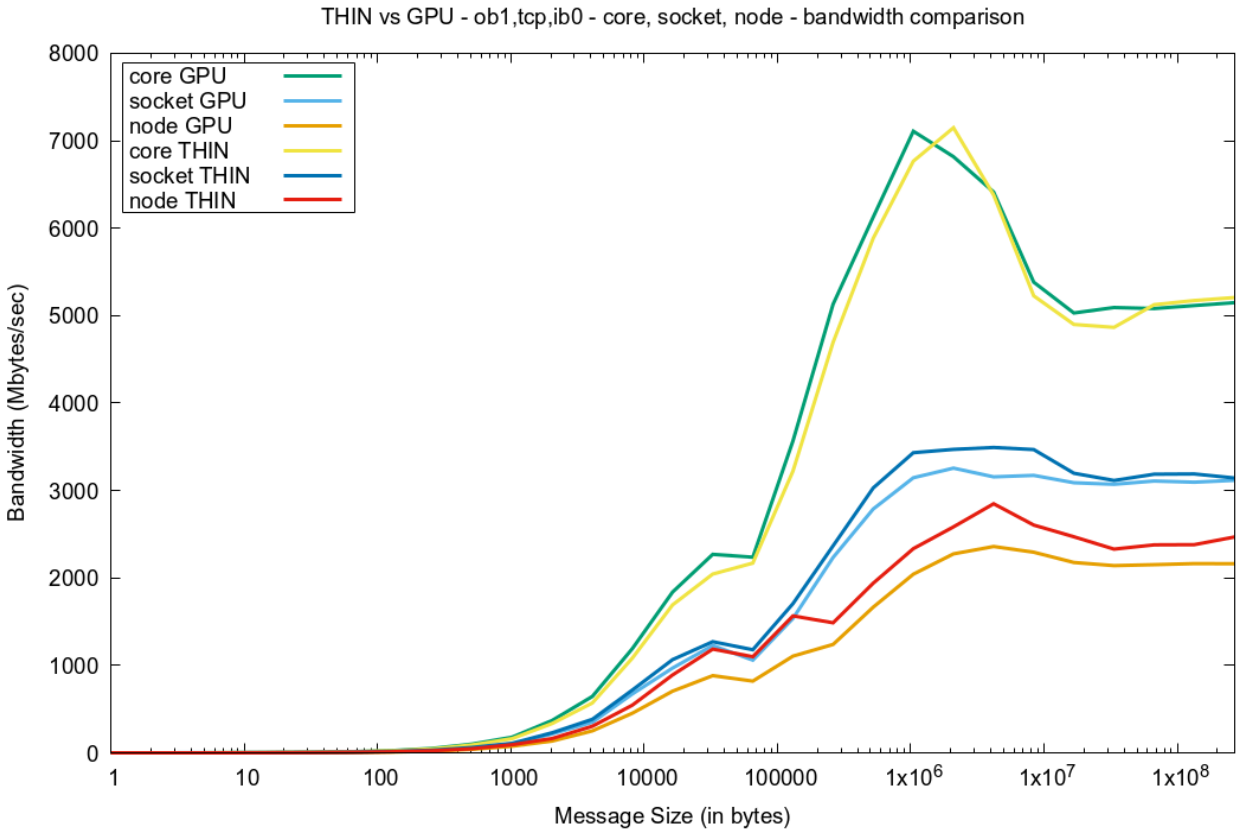
2.4.1 An example of a latency curve and a bandwidth curve:





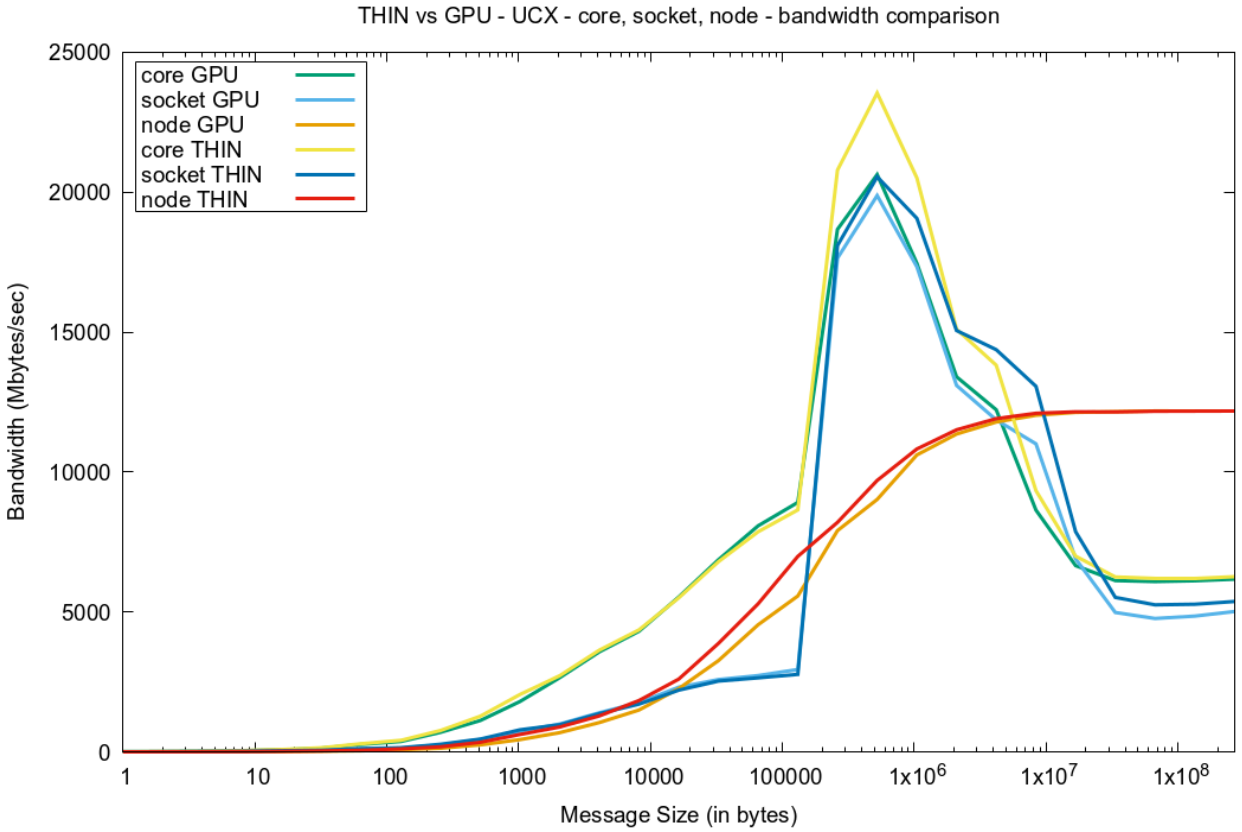
These 2 are examples of a bandwidth and latency curve. The folders as described in Section 2.1 have the full complement of these.
These 2 are for `OpenMPI` - on the `THIN` nodes - with the parameters `--map-by socket --mca pml ob1 --mca btl self,vader ..... -msglog 28`.

2.4.2 Comparison between THIN and GPU nodes for different mappings:

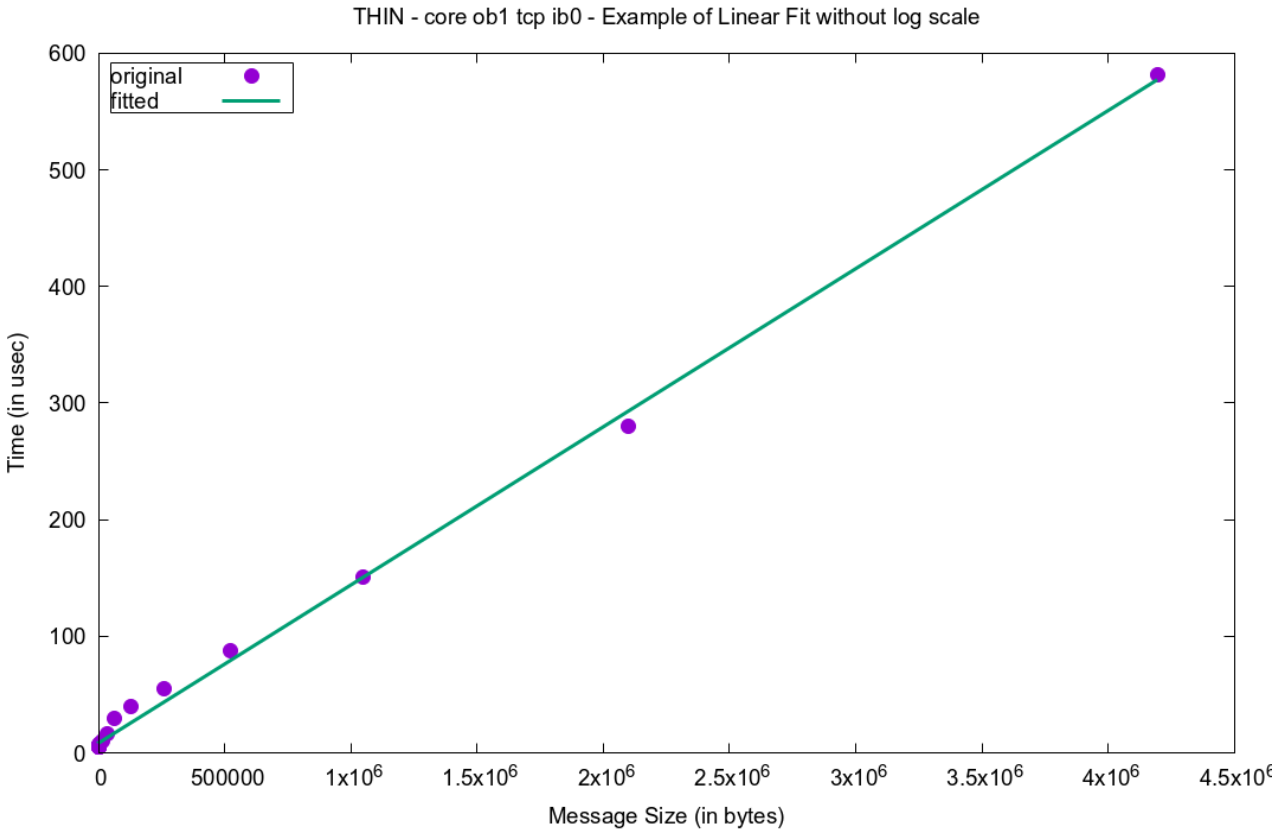


These images are a comparison between `THIN` and `GPU` node performance for `--mca pml ob1 --mca btl self,tcp --mca btl_tcp_if_include ib0` and also for `--mca pml ucx`. Similar

graphs can be generated by merging some `csv` files and using the `plot-comparison-thin-gpu.gp` file in the `scripts` folder. Of course, some little things need to be renamed in that file so that the appropriate titles, legends etc. are used.



2.4.3 Example of an explicit linear fit for the latency curve:



This curve is for the parameters `--map-by core --mca pml ob1 --mca btl self,tcp --mca btl_tcp_if_include ib0` using `OpenMPI` on a `THIN` node.
Since the log-scale curves that have been generated in the main folders don't actually show the linear nature of the plot (in class we were shown a log scale curve, hence that is what I reproduced), this is an example of that very same plot with an equispaced X-axis. The linear behaviour and the success of our fitting is very evident in this plot.
To produce more purely linear plots, please use the file linear-data_analysis.py in scripts in the base folder. Instead of the original data_analysis.py file, copy this to any of the output folders, follow the steps described in 2.2.1 and you will get linear latency curves if needed.

2.5 Discussion of results:

In all these cases the data is fitted against the simple communication model discussed in class -

$$t_{comm} = \lambda + size/b$$

where, λ is the latency, t_{comm} is the time needed for sending $size$ bytes across a network of bandwidth b.

2.5.1 UCX in `OpenMPI`:

`UCX` is highly optimized and it uses the `Infiniband` network by default. As can often be seen in the results table as well, it basically ignores `btl` parameters. It achieves the lowest latency and the highest bandwidth consistently across the board. Therefore, it is highly recommended to use this.

2.5.2 Other notes on `OpenMPI`:

The best choice for `pml` is clearly `ucx`, followed by `ob1`. The best choice for `btl` (when using `ob1`) for least latency and highest bandwidth is `vader` (shared memory), when all other things are equal. The performance of `tcp` is noticeably worse than `vader`. When using `tcp`, `infiniband` (`ib0`) clearly performs better than the ethernet (`br0`), which is in line with our expectations.
When mapping by `node`, the advantage of infiniband over ethernet is the most obvious.

2.5.3 Notes on `IntelMPI`:

When we set `I_MPI_FABRICS` to `shm` (shared memory), the `I_MPI_OFI_PROVIDER` parameter is basically ignored and this can be verified by our results as the values are often exactly the same, if not similar.
When `I_MPI_FABRICS` is set to `shm:ofi`, it uses `shm` in the same node, and `ofi` for communications across different nodes. This can also be used with > 1 node, unlike the pure `shm` option. Here, `I_MPI_OFI_PROVIDER` can be set to `tcp`, `sockets`, `shm`, `mlx`. The best performance (least latency) is seen for `mlx` and `shm` (when set to `shm`, it is actually exactly the same as setting `I_MPI_FABRICS` to `shm`). Then `sockets` performs moderately, and `tcp` has the worst performance in terms of latency. Despite this, the bandwidth changes are minimal for both `contiguous` and `socket` mappings and can be judged to be broadly in the same range. This suggests much more consistent performance for `IntelMPI` vs `OpenMPI` particularly when it comes to using `tcp` (intra-node), where `IntelMPI` shows significantly better performance.
When we set `I_MPI_FABRICS` to `ofi`, then `mlx` performs by far the best (least latency and highest bandwidth), followed by `shm` at a distant second, then `tcp`, and `sockets` at the end.
For mapping by `node`, regardless of whether we set `I_MPI_FABRICS` to `ofi` or `shm:ofi`, `mlx` performs by far the best again, and `tcp` is a very distant second, followed by `sockets` in last place.

Section 3: Compare performance observed against performance model for Jacobi solver:

3.1 Explanation of Files and Directory Structure:

The base folder contains the `Jacobi_MPI_vectormode.F` file, `input.1200` (used as input for the program), 2 scripts - `run_thin.sh` and `run_gpu.sh`, which runs the relevant benchmarks on `THIN` and `GPU` nodes respectively. There are 2 executables - `jacobi3D_thin.x`, `jacobi3D_gpu.x` - compiled by these scripts - one for `THIN`, and the other for the `GPU` nodes.
The main data is stored in `data_gpu` and `data_thin` folders. These each have `run_data` and `task_data` subfolders. The `task_data` folder stores the `task.xy.dat` files for each combination of a run of the benchmark. The `run_data` folder stores the proper output of the Jacobi program.
There's a `log` folder which stores the error and output scripts generated by `PBS` job scheduler.
And we have `final-thin.csv` and `final-gpu.csv` files which have the relevant data we need.

3.2 Running the benchmark and generating data:

The `run_thin.sh` and `run_gpu.sh` are for `THIN` and `GPU` nodes respectively. The command used in these scripts to compile their respective executables are -

```
mpif77 -ffixed-line-length-none Jacobi_MPI_vectormode.F -o jacobi3D.x
```

Then finally we run these using the command -

```
mpirun --map-by $1 --mca btl ^openib -np $2 ./jacobi3D_thin.x <input.1200
2>/dev/null 1>jacobi-$1-$2-thin.dat

$1 -> core socket node
$2 -> number of CPUs - 4 8 12 24 36 48
```

We use an `awk` one-liner to collect the data from these files and generate average values of `Maxtime`, `MLUPs`, and the total size of the arrays and store it in either `final-thin.csv` or `final-gpu.csv`.

3.3 Data:

For all of these we use `OpenMPI`. We can get the relevant bandwidth and latency parameters from Section 2. Some relevant parameters (L, serial time and P(L,1) for `THIN` and `GPU` nodes) are -

$$L = 1200$$
$$T_s(thin) = 15.327sec$$
$$T_s(gpu) = 22.086sec$$
$$P(L, 1, thin) = 112.737$$
$$P(L, 1, gpu) = 78.237$$

3.3.1 Network Data from Section 2:

The data is taken from `{thin,gpu}/output_{thin,gpu}_sample/openmpi-msglog_28` in section 2. For `core` and `socket`, the `ucx` parameters are used since that gives highest performance and is also used for the Jacobi solver. For `node`, `ucx` with `tcp` and `ib0` parameters are used (although these are chosen by default by `ucx` anyway).

System	Mapping	Latency ($\lambda - \mu sec$)	Bandwidth (Mbytes/sec)
THIN	core	0.198	6240.607
	socket	0.404	5342.649
	node	0.986	12180.664
GPU	core	0.213	6167.449
	socket	0.412	4981.419
	node	1.362	12177.116

3.3.2 THIN - core:

N	2*k	c	T_c	$P(L,N)$ theoretical	$P(L,N)$ estimated	$\frac{N.P(L,1)}{P(L,N)}$ estimated
4	4	46.08	0.00738	450.763	448.718	1.004
8	6	69.12	0.01107	901.290	797.603	1.130
12	6	69.12	0.01107	1351.936	1195.871	1.131

3.3.3 THIN - socket:

N	2*k	c	T_c	$P(L,N)$ theoretical	$P(L,N)$ estimated	$\frac{N.P(L,1)}{P(L,N)}$ estimated
4	4	46.08	0.00862	450.733	449.186	1.003
8	6	69.12	0.01293	901.173	804.844	1.120
12	6	69.12	0.01293	1351.760	1198.316	1.128

3.3.4 THIN - node:

N	2*k	c	T_c	$P(L,N)$ theoretical	$P(L,N)$ estimated	$\frac{N.P(L,1)}{P(L,N)}$ estimated
12	6	69.12	0.00567	1352.465	1198.756	1.128
24	6	69.12	0.00567	2704.401	2394.708	1.129
48	6	69.12	0.00567	5408.803	4583.697	1.180

3.3.5 GPU - core:

N	2*k	c	T_c	$P(L,N)$ theoretical	$P(L,N)$ estimated	$\frac{N.P(L,1)}{P(L,N)}$ estimated
4	4	46.08	0.00862	312.836	308.927	1.013
8	6	69.12	0.01293	625.605	533.895	1.172
12	6	69.12	0.01293	938.407	769.121	1.220

3.3.6 GPU - socket:

N	2*k	c	T_c	$P(L,N)$ theoretical	$P(L,N)$ estimated	$\frac{N.P(L,1)}{P(L,N)}$ estimated
4	4	46.08	0.00925	312.827	310.954	1.006
8	6	69.12	0.01387	625.524	555.085	1.127
12	6	69.12	0.01387	938.286	814.378	1.152

3.3.7 GPU - node:

N	2*k	c	T_c	$P(L,N)$ theoretical	$P(L,N)$ estimated	$\frac{N.P(L,1)}{P(L,N)}$ estimated
12	6	69.12	0.00567	938.662	834.903	1.124
24	6	69.12	0.00567	1877.325	1616.895	1.161
48	6	69.12	0.00567	3754.651	2462.398	1.525

3.4 Equations used behind the scenes:

The performance model follows the following equations -

$$P(L,N) = \frac{L^3 N}{T_S(L, N=1) + T_C(L,N)}$$

$$T_C(L,N) = \frac{c(L,N)}{B} + k.T_L$$

$$c(L,N) = L^2.k.2.8(10^{-6} MBytes)$$

where,

$c(L,N)$ = amount of data volume transferred over a node's network link

B = bidirectional bandwidth of the network link

T_L = latency of the network

k = largest number (over all domains) of coordinate directions in which the number of processes is greater than one.

$T_C(L,N)$ = communication time

$T_S(L)$ = sequential/serial time taken for $N = 1$

N = number of processors used

L = 1200 (for this particular instance)

3.5 Discussion of results:

For mapping by `core` and `socket`, the `THIN` nodes perform somewhat better than the `GPU` nodes across the board. This is despite us using less processors so that the effect of hyperthreading is kept to a minimum.

When mapping by `node`, for `N = 48`, we get much worse performance in the `GPU` node than in the `THIN` nodes. One could easily explain this as the result of hyperthreading causing significant performance degradation. This is in line with expectations.

Overall the serial time and communication overload, both of which are significantly worse for the `GPU` nodes, decreases the scalability of the Jacobi solver on the `GPU` nodes compared to the `THIN` nodes.

The performance is of course worse than the theoretical scalability model discussed in class.

The scalability is around the same when mapping by `core` and mapping by `socket`, and significantly worse when mapping by `node`.