1. General Remarks

This assignment is an introduction to MPI programming. You will be using the ece mpi cluster composed of 8 nodes

`en-openmpiXX.ece.cornell.edu`

where `XX = 00, 02, 03, 04, 05, 06, 07`. The node `01` has a different configuration than the other nodes so at the moment please do not use it.

`en-openmpi00.ece.cornell.edu` is a two socket, 16 cores per socket server. All other nodes are single socket 16 core servers. The nodes are connected by a 100MB router.

The operating system is `Linux Ubuntu SMP`.

You should be able to login to any of the nodes using your Cornell netid. When you login your directory will be visible from any of the nodes so there is no need to copy files among the nodes.

For mpi directives consult

`https://computing.llnl.gov/tutorials/mpi/`

or any other convenient source.

2. Compile and execute

Your MPI code must include the header file `"mpi.h"`.

The code must be compiled with `mpicc -o filename filename.c` (plus any flags that are required by your code.

In order to run a multiprocess code you need to specify in the file `my_hostfile` the nodes that will be used, for example `my_hostfile` might look like

```
en-openmpi00.ece.cornell.edu
en-openmpi02.ece.cornell.edu
en-openmpi03.ece.cornell.edu
```

Unless necessary, please do not specify more nodes than needed, (counting the number of cores in the nodes). For example, if you need less then 17 processes a single node will sufice. If you need 32 processes then either specify `en-openmpi00.ece.cornell.edu` or any two of other nodes. Note however that running 32 processes on two different nodes will be much slower than running them on `en-openmpi00.ece.cornell.edu`. This is because our router is relatively slow.

The following command will run your executable `filename`

```
mpirun -mca plm_rsh_no_tree_spawn 1 --hostfile my_hostfile -np N
./filename
```

where `N` is the number of processes you want to create.

3. Assignments

There are two codes to be written

I Benchmarking latency and bandwidth.

II Jacobi method for computing the SVD

Part **I**.

There are three levels of communication that should be benchmarked.

- Within a single socket (any node, less than 17 processes).
- Between sockets on the same node `en-openmpi00.ece.cornell.edu` only.
- Between different nodes.

The idea is to send progressively longer messages from one process to another, then send the message back and time this so called "ping-pong" exchange.

If you send a single character then this will give you some sense of latency. Clearly the "ping-pong' needs to be repeated several times and the total time needs to be averaged. You may also record the slowest and the fastest run.

Next send messages of length 2^k with, for example, $k = 4, 8, 16, 20$. Measure the time and avarage by the length of the message. This will give time per unit of info sent.

Present your result in a table or on a graph and describe your observations.

Part **II**.

After completing Part I you got some sense of the communication overhead. With that knowledge in mind we want to develop an MPI code for computing the SVD of an $m \times n$ matrix A by Jacobi iteration, see Lecture Notes for Lecture 12.

For a given A we need to compute U, V, Σ where

$$A \cdot V = U \cdot \Sigma,\ U^T U = I,\ V^T V = V,\ \Sigma = \mathrm{diag}(\sigma_1, ..., \sigma_n),\ \sigma_1 \geq \cdots \geq \sigma_n \geq 0.$$

Jacobi iteration proceeds in sweeps and generates a sequence of approximations $A^{(0)} = A$, $A^{(1)} = A^{(0)} J^{(0)}$,..., $A^{(i+1)} = A^{(i)} J^{(i)}$,.., where $J^{(i)}$ denotes the ith sweep. The exact U, V, Σ are the limits of the (infinite number of) iterations.

We need to terminate the iteration after applying a finite number of sweeps. A possible stopping criterion checks whether the off-diagonal "mass" of the current $(A^{(i)})^T A^{(i)}$,

$$\text{off}((A^{(i)})^T A^{(i)}) = \sum_{p \neq q} \left((A^{(i)}_{:,p})^T A^{(i)}_{:,q} \right)^2$$

is smaller than a user selected threshold. If there are too many iterations to satisfy the stopping criterion one can impose a hard limit which is an upper bound on the number of allowed sweeps.

Start with a small matrix and write a sequential code which uses Brent-Luk ordering. When you are satisfied that the code is correct, move to a parallel code. First run it on a single socket, and next run on different nodes.

After you are convinced that your parallel code is correct, measure the execution time for a number of combinations (matrix size, number of processes). Describe your findings.

Please use at least the following combinations of (matrix size, number of processes)

matrix size	# processes
1024×256	1
1024×256	2
1024×256	16
1024×960	32
1024×960	48
as large as you can fit	all available

If you use more combinations it will be easier for you to observe trends.

4. Format

- Your codes must be written in standard C language with MPI directives, and compiled with `mpicc`.
- Your code must be well documented.
- The first line in the code must show how the code should be compiled.
- For a fixed number of processes N you need to run your code for progressively larger matrices.
- For a fixed size of the matrix (make it large) you need to run your code for progressively larger number of processes and graph timing results.

For each case report speed up over a sequential algorithm ($N = 1$). If there is a slow down, identify the parts of the code that contribute to the slow down.

1. Your code(s) must be saved in separate file(s) named `your_net_id_hw3_part_p.c` where `p=I, II`.

2. Your codes must be described in a file `your_net_id_hw3_part_n.pdf`. Please include your name and net ID on all pages of your write up.

3. All files need to be packed with the `tar` or `gzip` facilities. The packed file must have the name `your_net_id_hw_3.suffix` where `suffix` is either `tar` or `zip`.

4. If you relay on resources outside lecture notes but publically available, you need to cite sources in your write-up.

5. Timing your codes.

For timing use `MPI_Time()` directive.