Introduction to MPI programming, Nov. 5

Due Wednesday, Nov. 14.

The objective today is to run a simple MPI program on Agave and to modify it to parallelize your Hénon basin dimension program. *Please be sure to compile and test your programs on a compute node* by typing

```
interactive
```

before proceeding! You will need to load the following module:

```
module load intel-mpi/2018x
```

You need to run this command *exactly once* each time you log in; it makes available the Intel compilers and MPI libraries.

You can compile MPI codes with the commands mpiifort (Fortran), mpicc (C), and mpicxx (C++). The compiler options are the same as for non-MPI codes, but the mpi version handles the complex task of including the appropriate header and module files and linking your code with all the MPI libraries. You may include these commands in a makefile in the usual way, provided that you have loaded the intel-mpi module first.

The sample program in mpihenon.f90

Please download this file, which provides a skeleton for the MPI version of your Hénon basin dimension program. The program runs in the following steps:

1. Initialize MPI and get the rank of each process.
2. The root process (rank 0) reads the input file named on the command line (if none is specified, it looks for henon.txt).
3. The root process broadcasts the data to all processes.
4. A dummy version of the basin dimension subroutine is called, which simply echoes the input data to the standard output.
5. In this dummy version, each process fills only the first 3 entries of the array that counts the number of uncertain points as a function of ε. The entries are me, $2 \times$ me, and $3 \times$ me. For instance, all entries of uncert are zero on the root process (me $= 0$); they are $[1, 2, 3]$ on rank 1, and so on.
6. This dummy code demonstrates how to perform a collective computation, which in this case is to sum the number of uncertain points from each portion of the grid handled by each MPI process.
7. The root process prints the results, cleans up, and exits.

You may type the name of any MPI routine into your favorite search engine to find the details of its function and arguments.

- MPI_Bcast is a function that broadcasts the values in the send buffer on the root process (listed as the first argument) to all other processes. This architecture is common in MPI programs for two reasons:
  - You do not want many processes to access the same disk file at once. Disks are the slowest form of memory, and satisfying many i/o requests at once leads to high latency.
  - On some clusters, only selected MPI process many have access to the main filesystem.

  Of course, high latency also can arise if there is a lot of output and all processes must wait for the root process to write it. Large applications may take advantage of parallel filesystems that allow several MPI processes to read and write pieces of computational grids and other large data structures, but this important topic is beyond the scope of the course.
- MPI_Reduce is an example of a global reduction operation. In this case, we need to sum the number of uncertain points in each subgrid. MPI handles the details of getting the counts from each process, adding them together, and delivering the total to the root process.

Compiling the program

As mentioned above, after loading the appropriate software module, you can compile this code with the command

```
mpiifort mpihenon.f90 -o mpihenon
```

Running the program

To run the sample program, please download and copy over the file henon.txt. The file contains two entries: the array henonparams containing the map parameters a and b, and the entries of xextent (interpreted as xmin, xmax, ymin, ymax, followed by the number of grid points in the x and y directions, all as floating-point numbers).

Running your program. MPI programs may be run interactively on a single node. If you want to use multiple nodes, then you need to use the batch system. This program is simple enough that you should be able to run it interactively,

but please do not run your program on the login nodes!

On Intel MPI, you invoke your program with a command of the form

```
mpirun -n 2 ./mpihenon
```

where the -n option specifies the number of MPI processes to start (2 in this example). In this case, all your MPI processes will run on the *same* node.

The batch system. MPI programs that use multiple nodes or require substantial compute time must be run under the batch queuing system. The file batch.sh is an example of a *shell script* that sets up the computing environment for your program. It assumes that your program has already been compiled. A shell script contains a list of commands just as you would type them at the terminal. They are executed in order, one after the other.

At the start of the file are several lines starting with #SBATCH. Any line starting with # is a comment line to the shell and does not affect the execution of your program. However, these lines are significant to the batch scheduler, which uses them to place your program in the appropriate queue for execution. Each line has the following meaning:

```
#SBATCH -n 2
```

says that your program will run on only 2 cores. Edit this line accordingly to run your program on more cores.

```
#SBATCH -t 0-00:02:00
```

places a wall-clock time limit on your program. The format is days-hh:mm:ss. For this example program, the wall-clock time limit is 2 minutes. For your Hénon basin dimension program, you might increase it to 5 minutes. The batch scheduler queues programs according to their time requests; shorter programs run at higher priority than long programs.

Important: Place a reasonable upper time limit in case your MPI program deadlocks. A deadlocked MPI program consumes CPU cycles in an infinite loop until it times out. (You are still charged for all that wasted time!) A time limit of a few minutes should suffice for this exercise.

The line

```
#SBATCH -o mpihenon%j.out
```

specifies that data written to your program's standard output stream is collected into a file with a name like mpihenon12345.out. (The %j is replaced by a 4- or 5-digit job number.) Similarly,

```
#SBATCH -e mpihenon%j.err
```

collects data written to the standard error stream into a file with a name like henon12345.err.

```
#SBATCH –mail-type=FAIL
```

sends you an email if your program fails. For a long-running program, you can modify this line to read

```
#SBATCH –mail-type=END,FAIL
```

which sends you an email when your program completes as well as if it fails.

```
#SBATCH –mail-user=asurite@asu.edu
```

tells the system where to send your email. You don't have to use your asu.edu account.

To run your program, you need to load the appropriate MPI module. The module command prints useful hints about the Intel MPI environment, but we don't need that information for our batch job. The addendum 2> /dev/null sends the output to the system's bit bucket.

Finally, you need to run your program under the control of MPI. In the case of Intel MPI, you launch your program with mpirun. (Some other MPI implementations use mpiexec.) The -n argument is required and tells the launcher how many copies of the program you want to run, i.e., the number of MPI processes to spawn.

After you have successfully compiled your program, you are ready to run it. Create batch.sh as described above and queue it for execution by typing

```
sbatch batch.sh
```

Your program should run within a minute or less. Please inspect the contents of the output and error files.

Practice exercise 1. Try running the example program interactively on 2, 4, and 8 processors.

Practice exercise 2. Try running the example program interactively with the name of a nonexistent input file, for example

```
mpirun -n 2 ./mpihenon no.such.file
```

to see how the error handling works.

The Hénon basin dimension program

Use the main program in mpihenon.f90 as a template for your program. Delete the basin_dim subroutine and modify your existing code as follows:

- Instead of parallelizing over the number of epsilons, divide the grid up into pieces based on the number of MPI processes. You may use any reasonable procedure, and you may assume that the number of MPI processes is even.
- You do *not* need to do the linear regression with linfit and you do *not* need to include linfit or any LAPACK calls in your program. Instead, simply print, for each epsilon, the total number of uncertain points as integer values. Only the root process should do any printing.
- You may write your program in C or C++ instead of Fortran.

- You decide how to break up your code among source files. Just be sure to include *all* source files when you submit your program for grading. (Do not include the standard headers for C/C++ programs, but if you write your own custom header, then please include it with the rest of the source.)
- Your program should run correctly and give essentially identical results on 2, 4, 6, 8, and 16 processors. Depending on the details of your partitioning algorithm, it is possible that differences in numerical roundoff as you define the initial grid points lead to slightly different answers as a function of the number of MPI processes. This is not an error, but the differences should be small (less than 10^{-3} relative difference) for the larger values of ε.
- Include a makefile.
- Do *not* use OpenMP.
- Scaling behavior is *not* a grading criterion, because your MPI program probably will run on only one node.

Submission instructions

Create a tar file with

```
tar -c -f mpihenon.tar Makefile filelist
```

where filelist contains all your source code files for this project. Upload your tar file to Blackboard.

Grading rubric

Plagiarism warning: Substantial copying of source code without attribution will result in a grade of zero on this assignment. Please see the syllabus for the rules. Oherwise, this assignment is worth 20 points, as follows.

Compiles without errors or warnings	8 points
Uses MPI and produces substantially similar results on up to 16 MPI processes	8 points
Includes a working Makefile	2 points
Includes adequate documentation in the source code	2 points
Total	20 points