In-class exercises for Nov. 19 (Final project)

These two exercises serve as a final project. You will submit all your materials on Blackboard *no later than 5 p.m. on Friday, Dec. 7.* Please see the submission instructions below. In lieu of an in-class final, we will meet in the regular classroom from 2:30–4:20 on Dec. 7. You will demonstrate that your program works (or, if it doesn't, then you can ask questions and will have until 5 p.m. on Dec. 7 to fix it).

Important: This schedule is subject to change if Agave goes down for scheduled or unscheduled maintenance. Please try to get your programs working as soon as you can. I will announce any alternative arrangements on Blackboard if the cluster will be down on Dec. 7.

Jacobi iteration

A typical MPI exercise is to write a program to implement a distributed-memory version of Jacobi iteration for Poisson's equation on the unit square,

$$\begin{aligned} \Delta u &= f \quad \text{on } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned} \tag{1}$$

where Ω is the interior of the unit square and $\partial\Omega$ is the boundary. Let

$$f(x,y) = e^{x+y}[(x^2+3x)(y^2-y)+(y^2+3y)(x^2-x)], \tag{2}$$

for which the "true" solution to Eq. (1) is

$$u(x,y) = e^{x+y}(x^2-x)(y^2-y). \tag{3}$$

Normally, of course, an analytical formula for u cannot be found in most cases, so a numerical method is used to approximate u. We will use the f in Eq. (2) as the forcing in Eq. (1) so that you can compare the final approximated solution with the true one in Eq. (3) and validate your MPI code.

The Jacobi iteration finds an approximate solution to the linear system of equations that arises from finite differencing. Suppose that the grid contains $M+1$ equally spaced points on each side, indexed from 0 to M, and let $\Delta = 1/M$ be the spacing between adjacent grid points in both directions. We define $u_{j,k}$ to be the numerical solution evaluated at the grid point (x_j, y_k), $1 \le j,k \le M-1$ and use the superscript n to count iterations. Starting from an initial guess of the solution (say $u^0_{j,k} = 0$), the Jacobi scheme is the update

$$u^{n+1}_{j,k} = \frac{1}{4}\left(u^n_{j+1,k} + u^n_{j-1,k} + u^n_{j,k+1} + u^n_{j,k-1} - (\Delta^2) f_{j,k}\right) \tag{4}$$

for $1 \le j,k \le M-1$. The boundary conditions are specified prior to the start of the iteration at $u_{0,k}$, $u_{M,k}$, $u_{j,0}$, and $u_{j,M}$, $0 \le j,k \le M$.

Important: Equation (1) is not time dependent! The objective of any numerical scheme is to find a steady-state solution. The superscript n in Eq. (4) does *not* refer to time; rather, it is the nth iteration (guess) of the Jacobi numerical scheme to approximate the "true" solution u. Please see the Nov. 7 lecture notes for additional information.

Let the array u(0:m,0:m) hold the current approximation of the steady-state solution. Upon each iteration, the Jacobi method updates only the interior points, u(1:m–1,1:m–1). However, the boundary conditions are required to determine a unique solution. Although Eq. (1) specifies zero boundary conditions, your program should be capable of handling any Dirichlet boundary condition (i.e., of the form $u = g$). Hence you will need to initialize and store the boundary points (which will be fixed for the duration of your program).

To check convergence, you need to compute

$$\delta = \max_{1 \le j,k \le M-1} \left| u_{j,k}^{n+q} - u_{j,k}^{n} \right| \tag{5}$$

from time to time. Consider the scheme to have converged if $\delta < \varepsilon$, where ε is a prespecified tolerance. This criterion says to halt if iteration $n+q$ has not changed from iteration n by more than ε at any point. The integer q determines how often you check for convergence. As discussed below, computing δ requires a collective communication, which is relatively expensive; hence there is a tradeoff between computation and communication in any given implementation.

Common problem requirements

Implement a distributed-memory version of the Jacobi solver for Eq. (1) on the unit square with a 2-d processor topology in two ways: one using MPI and the second with coarray Fortran. In both cases, take $M = 255$, giving a 256×256 grid with a 254×254 set of interior points. (Points indexed 0 and 255 are boundary points.) Use a 4×4 grid of processors, each of which is responsible for a 64×64 subgrid.

To initialize the iteration, let the root process (MPI rank 0 and coarray image 1) read two numbers from the standard input: the convergence criterion ε and the maximum number of iterations, in that order. Fix $u^0 = 0$ as the initial guess.

Every tenth iteration, check for convergence; that is, take $q = 10$ in Eq. (5). One each processor, compute the absolute value of the difference between the current $(n+q)$th iteration and the nth iteration for each interior point of the local grid. With MPI, you will do a collective computation, using MPI_Reduce or MPI_Allreduce, to choose the maximum of the differences among the processors and deliver the result to the root process.

To prevent an infinite loop, stop your program after, say, 5,000 iterations or when $\delta < 10^{-4}$, whichever occurs first. When the iteration stops, whether or not it has converged, compute

$$c = \max_{1 \le j,k \le M-1} \left| u_{j,k}^{n} - u(x_j, y_k) \right|,$$

i.e., the maximum pointwise difference between the numerical solution and the true solution (3). Do this evaluation in a similar manner to that for δ. Let the root process report

the final value of δ and c to the standard output. Altogether, your program will print out *only* these two numbers.

Suppose that process k is responsible for updating rows r_1 to r_2 and columns c_1 and c_2 of the domain. Then image k might allocate an array dimensioned $(r_1-1:r_2+1, c_1-1:c_2+1)$, with rows numbered from r_1-1 to r_2+1: and columns numbered from c_1-1 to c_2+1. Rows r_1-1 and r_2+1 and columsn c_1-1 and c_2+1 are ghost points: they are exchanged with the neighboring processor and used to update the interior grid points $u(r_1:r_2, c_1:c_2)$. Implement a protocol by which each process exchanges ghost points with its neighbors.

Exercise 1

Implement the Jacobi iteration using MPI on a 4×4 processor arrangement. Use MPI_Sendrecv to implement simultaneous data exchanges with neighboring processes. You will need to devise a protocol by which each process exchanges data with its neighbors to the left and right and to its north and south, as appropriate. You may call MPI_Reduce or MPI_Allreduce at your option to test convergence and decide whether to exit. (See the Nov. 7 lecture notes for details.)

Put all the source code for your program in a file called jacobimpi.f90. (You may also write this program in C or C++, in which case call your source code file jacobimpi.c.) Your program must compile and run with Intel Fortran (or C/C++) and include MATLAB-style documentation explaining what it computes and what the required inputs are. As usual, be sure to load

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

and compile your program with mpiifort (or mpiicc or mpiicpc). When you submit your job to sbatch, be sure to allow 5 minutes of wall-clock time and specify 16 processors both to sbatch and to mpirun.

Exercise 2

Implement the same computation and processor arrangement using coarray Fortran. Compile and link your program with –coarray=shared. Include the command

```
export FOR_COARRAY_NUM_IMAGES=16
```

prior to the mpirun command to specify 16 images. A sample batch script might look like this:

```
#SBATCH -n 1
#SBATCH -t 0-00.06        # 6 minutes wall-clock time
#SBATCH -o jacobi%j.out
#SBATCH -o jacobi%j.err
...
```

```
module load intel-mpi/5.0 2> /dev/null
export FOR_COARRAY_NUM_IMAGES=16
mpirun -n 1 $HOME/jacobicoarray < $HOME/jacobi.at
```

Please see Gil Speyer's Introduction to Agave presentation for additional options, such as email upon completion or failure.

For debugging, you may find it helpful to compile your programs with the options –traceback and –check bounds, which provide a stack trace and array subscript checks whenever possible. Place your source code into a file called jacobicoarray.f90.

Collaboration and attribution policy

An Internet search will turn up many examples of MPI-based implementations of the Jacobi iteration method in both C and Fortran for various elliptic PDEs. You are welcome to look through these codes to understand how the iteration works and how the communication should proceed. However, if you model your programs on one of these MPI examples, then *you must acknowledge the source of the original program by including the full URL in your program's documentation.* Failure to do so may constitute plagiarism, which is subject to serious sanctions (including, but not limited to, a grade of zero on this assignment).

You are welcome to talk with me about your program, but you may *not* consult with any other person. By submitting your program, you assert that your code is your own work, unless you have borrowed aspects from another implementation or example, for which you have included a full URL in your program's documentation.

Suggestions for getting started

Start by writing a 1-processor code on a 256×256 grid that implements the Jacobi iteration and checks the result with the "true" answer. After you have a working numerical scheme, work on an implementation that includes a halo of ghost points as necessary that surrounds each 64×64 subgrid with MPI exchanges. Use whatever scheme you wish to map process numbers to subgrids. First debug a program that has adjacent processors exchange with neighbors to the left and right, then neighbors above and below, perhaps by exchanging a single value. Then add your Jacobi iteration on top of that.

Please do not run your programs on the login nodes on Agave. Be sure to type

```
interactive
```

before you try to compile and debug! You may run MPI programs interactively on the compute nodes if they don't use a lot of time; otherwise, use the batch system to submit your jobs. It suffices to run your program on 16 cores on one compute node.

Submission instructions

Make a tar archive of each of the following files:

1. jacobimpi.f90
2. jacobcoarray.f90
3. Makefile

Call your archive final_yourname.tar, where yourname is your last name. Upload the tar file to the Blackboard site. You will have up to *two* attempts, but I will grade only the second one. Your program is due by 5 p.m. on Friday, December 7. **No extensions!**

Caveat: The due dates for the project may be changed if Agave is taken down for maintenance after classes are over. I need to have a couple days to go through all the programs and test them. I will provide you with as much advance warning as possible if the due date needs to be changed.

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, each program is worth 20 points, as follows.

Exercise 1 (MPI):

Compiles without errors or warnings	6 points
Uses MPI and produces an answer close to the "true" one	6 points
Uses a 4×4 process arrangement	4 points
Includes a working Makefile	2 points
Includes adequate documentation in the source code	2 points
Total	20 points

Exercise 2 (Coarrays):

Compiles without errors or warnings	6 points
Uses Coarray Fortran and produces an answer close to the "true" one	6 points
Uses a 4×4 process arrangement	4 points
Includes a working Makefile	2 points
Includes adequate documentation in the source code	2 points
Total	20 points