M3C 2018 Final Project

Due date/time: 14/12/2018, 11:59pm GMT

Getting Started: If needed, update your local copy of the course repo (*sync* and *pull*). There will be a *project* directory with files needed for this assignment.

Corrections and clarifications (added 6/12/18)

Part 1: (4/12/18)

1) A few comments have been mistakenly carried over from the HW2 versions of the provided codes:

- FGD runs for 1000, not 400 epochs
- p1serial uses FGD with nnmodel, not SGD with snmodel
- You do not need to make any changes to p1serial.f90 or p1main.f90

2) dtrain is initialized to 5000 in the main codes provided, but it is generally better to set it so that $dtrain + dtest \leq d$

Part 3: (6/12/18) The equation for the white update for the AOS method had a typo which has now been corrected. It should be $\Delta^2 S_{i,j})]$ as in the other update equations. The incorrect version had $\Delta^2)S_{i,j}]$

Using software version control (2 pts)

You should develop the code in a **private** *git* repository named, *m3cproject*. You will submit this project by providing me with read-access to your bitbucket repo. You will also have to provide a commit id setting the version of the project that should be marked. You should periodically add/commit updates to your project. The repository should contain three sub-directories, part1, part2, and part3, containing relevant files as described below. The final version of the repository should also contain a readme file with a list of new subroutines/functions/modules added to the template codes accompanied with short 1-sentence descriptions of the new routines.

Part 1: Parallel neural network (15 pts)

You will develop a parallel neural network simulation code based on *nnmodel* from HW2. *Full* gradient descent (*fgd*) will be used rather than *sgd* to train the model. Recall that the components of ∇C are calculated with sums over *d* training images, and the *sgd* approach truncated this sum so that one randomly chosen image was used instead of all *d* images. We now remove this truncation and use the full sum to compute the gradient of the cost function which is then used to update the vector of fitting parameters:

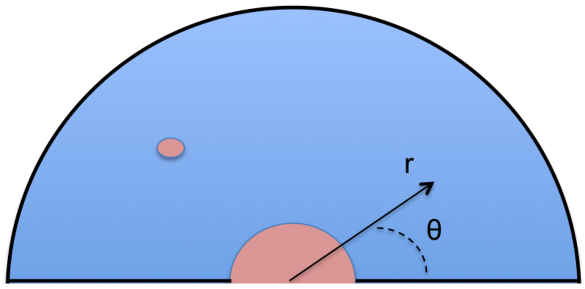
$$\mathbf{f}^{[q+1]} = \mathbf{f}^{[q]} - \alpha \nabla C^{[q]}$$

Complete Fortran routines for using *fgd* with *nnmodel* have been provided in *p1serial.f90*. The main program, *p1main.f90*, 1) reads in training and testing images, 2) reads in model parameters from *data.in* (which must be created), 3) trains the model with *fgd*, and 4) computes the test error. You should complete *p1mpi.f90* and *p1main_mpi.f90* so that the computation of ∇C is parallelized with MPI. The initial MPI setup and distribution of *dtrain* images to *numprocs* processes (each process works with *dlocal* training images) has been provided in the main program.

1. Complete *p1main_mpi.f90* so that a copy of the initial guess for the fitting parameters, *fvec0*, is available on each process prior to the call to *fgd*.

2. Implement a parallelized computation of ∇C in *fgd_mpi* and *nnmodel_mpi*. After receiving output from *nnmodel_mpi*, code should be added to *fgd_mpi* to update *fvec* appropriately each iteration. The *mpi* and serial versions of the codes should produce the same output when using the same model parameters (*e.g. dtrain* and *m*).

Part 2: Bacterial contamination



Consider the following model for the spread of bacteria on a liquid surface:

$$\frac{\partial C}{\partial t} = \nabla^2 C - gC + S$$

where $C(r, \theta, t)$ is the concentration of bacteria at location, (r, θ), and time, t, $S(r, \theta)$ is a source of contamination, and g is the death rate ($g \geq 0$). We consider the following configuration:

$$1 \leq r \leq \pi + 1$$
$$0 \leq \theta \leq \pi$$

and assume that antibiotics have been applied to surfaces of the container such that $C(r = \pi + 1) = C(\theta = 0) = C(\theta = \pi) = 0$. The surface, $r = 1$, however, is an additional source of contamination with $C(r = 1) = sin^2(k\theta)$ where k is a model parameter.

We will only consider the time-independent problem which, in polar coordinates, is:

$$\frac{\partial^2 C}{\partial r^2} + \frac{1}{r}\frac{\partial C}{\partial r} + \frac{1}{r^2}\frac{\partial^2 C}{\partial \theta^2} - gC = -S$$

We will use a Gaussian model for the source, $S = exp\left[-20\left((r - r_0)^2 + (\theta - \theta_0)^2\right)\right]$, where the parameters, r_0 and θ_0, set the source location.

In this part, you will complete Fortran and Python routines to 1) compute numerical solutions to this model using the over-step method described here: Iterative methods for contamination model (edited 6/12/18), 2) analyze the performance of your codes, and 3) analyze the contamination dynamics.

Note that you have been provided with both *Fortran* and *Python* codes for computing solutions using Jacobi iteration.

---

1) (10 pts) Complete the function *simulate* in *p2.py* and compute solutions to this model with the *OSI* method described in the accompanying notes. The function should return the final concentration distribution and an array (or list) containing the maximum change in concentration for each iteration ($max(|C_{i,j}^{k+1} - C_{i,j}^{k}|)$). Iterations should terminate when this maximum change falls below *tol* which is an input parameter.

2) (10 pts) Complete the subroutine *simulate* in *p2.f90* and again compute solutions to this model with the *OSI* method. The routine should again return the final concentration matrix, but note now that several model and numerical parameters are module variables rather than input to the subroutine. Additionally the maximum change in C for each iteration should be stored in the module variable, *deltac*.

3) (15 pts) Analyze the performance of your Fortran and Python routines. Your analysis should consist of one or more figures and accompanying discussion. Place the code used to generate your figures in the function, *performance*, in *p2.py*, and add your discussion to the docstring of this function. You should save these figures and include them in your project repo.

4) (20 pts) It is possible to fight against the contamination by applying antibacterial agents locally along the contaminated boundary. This modifies the boundary condition at *r=1* to, $C(r = 1) = exp(-10(\theta - \theta^*)^2)sin^2(k\theta)$. What is the best choice of θ^*? It is sufficient to consider the case, $g = 1, r_0 = 1 + \pi/2$, *k*=2, and $S_0 = 2$, but you are encouraged to consider other parameter values as well. Place your answer, an explanation of how you arrived at it, and accompanying code in *analyze* in *p2.py*. You should also save and submit figures generated by your code.

Part 3: MPI simulation of bacterial contamination

Here, you will develop a distributed-memory approach for the model introduced in part 2, and you will utilize the alternating over-step method (AOS) described here: Iterative methods for contamination model (edited 6/12/18). A radial domain-decomposition approach is used where process *p* is assigned a portion of the domain $r_{p1} \leq r \leq r_{p2}$. This decomposition is computed for you as described below.

The file, *p3.f90* contains a main program which initializes MPI, reads in input parameters from a text file (which you should create), calls the subroutine *simulate_mpi*, and writes the output from *simulate_mpi* to a file. You will have to complete *simulate_mpi*.

1. (25 pts) The domain decomposition takes place at the begining of *simulate_mpi*. The *n* radial points are distributed to *numprocs* processes using *MPE_DECOMP1D*. Points on the radial boundaries should be assigned to the first and last processes (there are *n+2* points total in each direction). Then, each process should solve the model from part 2 for r_i from *istart* to *iend* (and $0 \leq \theta \leq \pi$) using the AOS method. Since finite-difference approximations require information from *i+1* and *i-1*, each process must typically exchange information with one or more neighbors during each iteration.

Complete *simulate_mpi* so the model system of equations is distributed across the *numprocs* processes and iteratively solved in parallel using the AOS method.

Note: After iterations in *simulate_mpi* have terminated, each processor's solution, *clocal*, is gathered into the variable *c* on *myid=0* using *MPI_gather* and *MPI_gatherv*. The main program will then write this variable out to the file, *fmpi.dat*. This file can be loaded in python with *np.loadtxt*.

2. (3 pts) It is possible to decompose the square grid into $numprocs = m^2$ squares (where *m* is an integer greater than 1) rather than *numprocs* 'rectangles'. Clearly and concisely describe one key advantage and one disadvantage of using 'smaller-squares' approach for this problem. Place your answer in a comment at the top of *p3.f90*

Note: It is only after the k-loop has been completed that the full solution C should be assembled. During the loop, each process should only work with the portion of C needed to update *Clocal*.

Further comments

1. Your fortran code should work to double precision accuracy.

2. Figures generated by your python functions should be saved and included in your repository following the naming convention pXY.png for the Yth figure created for question X.

3. Both serial and parallel codes should be efficient where possible. Marking will focus on computationally intensive parts of routines. It is unlikely that code for data input/output, initialization of variables, post-processing/analysis, and similar tasks will be examined closely (for speed). You have the option of adding comments explaining/defending your approach.

4. Add module variables and sub-programs as needed, but clearly add comments on substantive additions. Do not remove/modify existing code or module variables unless you are explicitly instructed to do so (or consult with the instructor).

5. Parallelization should be implemented for a general (but reasonable) number of threads/processes. You can assume that $d \gg numprocs > 1$ for part 1 and $n \gg numprocs > 1$ for part 3.

6. Below the *name==main* section of *p2.py*, add code that calls *performance* and *analyze* and generates the figures you are submitting.

7. If you are using MLC machines and *mpif90* and/or *mpiexec* are not working, run the following in the unix terminal (the first for *mpif90*, the second for *mpiexec*):

```
$ export PATH=$PATH:/usr/lib64/openmpi/bin/

$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib64/openmpi/lib/
```

8. You may use the following Python modules as needed: *numpy*, *scipy*, *matplotlib*, *time*, *timeit*. Please ask if you would like to use any others.

9. Be sensible when choosing problem sizes. However, at this point in the term, it is fine for calculations to take several minutes. You do not need to run for very large problem sizes, though you should consider large problems when developing and parallelizing your codes.

Submitting the assignment

To submit the assignment for assessment, go to the course blackboard page, and click on the "Assignments" link on the left-hand side of the page, and then click on Project. Click on "Write Submission" and add the statement: "This is all my own unaided work unless stated otherwise." Provide the url for your bitbucket project repo as well as the commit id for the version which you would like to have assessed. Remember to provide read access to user, imperialm3c (go to Settings –> User and group access). Finally, click "Submit".