Cardiac Electrophysiology Simulation

.

In this assignment you'll implement the Aliev-Panfilov heart electrophysiology simulator discussed in class (Lecture 12) using MPI and OpenMP. Along with this project description, studying Lecture 12 might be very helpful for the assignment. Since implementing/debugginparallel codes under message passing can be more time consuming than under threads, giveyourself sufficient time to complete the assignment. In addition, this assignment requires you to conduct various performance studies. Please also allow yourself enough time for performance measurements.

Background

Simulations play an important role in science, medicine, and engineering. For example, a cardiac electrophysiology simulator can be used for clinical diagnostic and therapeutic pur- poses. Cell simulators entail solving a coupled set of a equations: a system of Ordinary Differential Equations (ODEs) together with Partial Differential Equations (PDEs). We will not be focusing on the underlying numerical issues, but if you are interested in learning more about them, please refer to references at the end of this manuscript.

Our simulator models the propagation of electrical signals in the heart, and it incorporates a cell model describing the kinetics of a the membrane of a single cell. The PDE couples multiple cells into a system. There can be different cell models, with varying degrees of complexity. We will use a model known asthe Aliev-Panfilov model, that maintains 2 state variables, and solves one PDE. This simple model can account for complex behavior suchas how spiral waves break up and form elaborate patterns. Spiral waves can lead to life threatening situations such as ventricular fibrillation (a medical condition when the heart muscle twitches randomly rather than contracting in a coordinated fashion).

Our simulator models electrical signals in an idealized system in which voltages vary at discrete points in time, called timesteps on discrete positions of a mesh of points. In our case we'll use a uniformly spaced mesh (Irregular meshes are also possible, but are more difficult to parallelize.) At each time step, the simulator updates the voltage according to nearest neighboring positions in space and time. This is done first in space, and next in time. Nearest neighbors in space are defined on the north, south, east and west.

In general, the finer the mesh (and hence the more numerous the points) the more accurate the solution, but for an increased computational cost. In a similar fashion, the smaller the timestep, the more accurate the solution, but at a higher computational cost. To simplify our performance studies, we will run our simulations for a given number of iterations rather than a given amount of simulated time because actual simulation takes too long (several days) on large grids.

Simulator

The simulator keeps track of two state variables that characterize the electrophysiology we are simulating. Both are represented as 2D arrays. The first variable, called the excitation, is stored in the *E*[ ][ ] array. The second variable, called the recovery variable, is stored in the *R*[ ][ ] array. Lastly, we store *Eprev*, the voltage at the previous timestep. We need this to advance the voltage over time

Since we are using the method of finite differences to solve the problem, we discretize the variables E and R by considering the values only at a regularly spaced set of discrete points. Here is the formula for solving the PDE, where the *E* and *Eprev* refer to the voltage at current and previous timestep, respectively, and the constant α is defined in the simulator:

```
E[i,j] =  Eprev[i,j] +  alpha* ( Eprev[i+1,j] + Eprev[i-1,j] +
                                 Eprev[i,j+1] + Eprev[i,j-1] - 4 * Eprev[i,j])
```

Here is the formula for solving the ODE, where references to *E* and *R* correspond to whole arrays. Expressions involving whole arrays are pointwise operations (the value on i,j depends only on the value of i,j), and the constants *kk, a, b,* ϵ*, M* 1 and *M* 2 are defined in the simulator and *dt* is the time step size.

```
E = E -  dt*( kk * E * (E - a)             * (E - 1) + E * R);

R = R + dt*(epsilon + M1*R / (             E + M2)) * (-R - kk * E * (E-b-1));
```

Serial Code

Serial code is provided to you with a working serial simulator that uses the Aliev-Panfilov cell model. The simulator includes a plotting capability (using gnuplot) which you can use to debug your code, and also to observe the simulation dynamics. The plot frequency can be adjusted from command line. Your timings results will be taken **when the plotting is disabled.**
The simulator has various options as follows.

```
./cardiacsim

With the arguments
-t <float> Duration of simulation
-n <int> Number of mesh points in the x and y dimensions
-p <int> Plot the solution as the simulator runs, at regular intervals
-x <int> x-axis of the the processor geometry (Used only for your MPI implementation)
-y <int> y-axis of the the processor geometry (Used only for your MPI implementation)
-k Disable MPI communication
-o <int> Number of OpenMP threads per process
Example command line
        ./cardiacsim -n 400 -t 1000  -p 100
```

The example will simulate on a 400 x 400 box, and run to 1000 units of simulated time, plotting the evolving solution every 100 units of simulated time.

You'll notice that the solution arrays are allocated 2 larger than the domain size (n+ 2). We pad the boundaries with a cell on each side, in order to properly handle ghost cell updates. See Lecture 12 for explanation about ghost cells.

Assignment

You'll make modifications to the provided code and parallelize it with MPI and OpenMP.

1. Parallelize the simulator using MPI. You need to support both one and two dimensional processor geometries. Under 1D geometry conditions (only -y is set), ghost cells arecontiguous in memory. But in 2D geometry, you need to pack and unpack discontiguous memory locations to create messages. The command line flags -x -y have been set up to specify the geometry. Your code must correctly

handle the case when the processor geometry doesn't evenly divide the size of the mesh. You only need to support square meshes.

2. Add OpenMP parallelism within an MPI process to support hybrid parallelism with MPI+OpenMP.
3. Optimize your code to improve performance as much as you can. Any code transformations are legalas long as the program produces the same simulation results as the initial serial implementation. Document these optimizations in your report.
4. Conduct a performance study **without the plotter is** on your machine.
   - Use Gflops rates when you report performance NOT the execution time !!!
   - Present your results as a plot and interpret them in your report.

   (a) Conduct a strong scaling study for N = 1024 and T = 100. Observe the running time as you successively double the number of processes, starting at P=1 core and ending at 16 cores, while keeping N fixed. Determine optimal processor geometry. Compare single processor performance of your parallel MPI code against the performance of the original provided code. Do not use OpenMP in this performance study.

   (b) Using the indirect method by disabling communication, measure communication overhead. Shrink N by a factor of 1.4 ($\sqrt{2}$) so that the workload shrinks by a factor of 2. Measure the communication overhead for this reduced problem size and repeat, until communication overhead is 25% or greater. Do not use OpenMP in this performance study.

   (c) Turn on OpenMP parallelism and conduct the same strong scaling study by using 2, 4, 8OpenMP threads per process (starting at P=8 and ending at P=2. In a figure, compare the performance numbers with the strong scaling with MPI only. Which combination of MPI+OpenMP is the fastest? Use 1D geometry for the MPI data decomposition for this study.

Suggestions: I would start with 1D geometry, test correctness, add OpenMP, test correctness and then support for 2D geometry, then optimize the code.

Environment

- You will need the gnuplot plotting program to be installed on your computer. You can download the software from http://www.gnuplot.info
- You will need an MPI library to be installed on your computer. Download the MPIlibrary from http://www.open-mpi.org/ .
- In order to compile with MPI on your local machine, you need to export these pathson your shell or on the development environment (e.g. eclipse).
- Make file that we have provided includes an "arch.gnu" file that defines appropriate command line flags for the compiler. See the arch.gnu file to turn on the MPI flag to enable MPI compilers.
- Running an MPI program requires you to use a wrapper script (mpirun) by indicating the number of processes to be used:

```

mpirun -n num_procs ./cardiacsim -n 400 -t 200
```

- For mixing OpenMP and MPI, you still need to set the smp to 16, and then specify the number of processors with with mpirun. The simulation should take the number of threads per processor as an argument.

Testing Correctness

Test your code by comparing against the provided implementation. You can catch obvious errors using the graphical output, but a more precise test is to compare two summary quantities reported by the simulator: the L2 norm and the L max, the maximum value (magnitude) of the excitation variable. The former value is obtained by summing the squares of all solution values on the mesh, dividing by the number of points in the mesh (called normalization), and taking the square root of the result. For the latter measure, we take the ”absolute max,” that is the maximum of the absolute value of the solution, taken overall points.While there may be slight variations in the last digits of the reported quantities, variations in the first few digits indicate an error. Verify that results are independent of the number of cores and the processor geometry, by examining the summary quantities.

Submission

- Document your work in a well-written report which discusses your findings. Offer insight into your results and plots.
- Your report should present a clear evaluation of the design of your code, including bottlenecks of the implementation, and describe any special coding or tuned parameters.
- Submit both the report and source code electronically through google classroom.
- Please create a parent folder named after your username(s). Your parent folder should include a report in pdf and a subdirectory for the source. Include all the necessary files to compile your code. Be sure to delete all object and executable files before creating a zip file.

Grading

Your grade will depend on 3 factors: performance, correctness and the depth and your explanations of observed performance in your report.

Implementation (70 points): MPI 1D partitioning (30 pts), and MPI 2D partitioning (30pts), OpenMP parallelism (10 points)

Report (30 points): implementation description and any tuning/optimization you performed(6 pts), strong scaling study and your observations (8 pts), communication vs computation studies and your observations (8 pts), OpenMP+MPI results and your observations (8 points).

GOOD LUCK!

References

https://www.simula.no/publications/stability-two-time-integrators-aliev-panfilov-system.Prof. Scott Baden
http://cseweb.ucsd.edu/~baden/