COMP 429/529- Parallel Programming: Assignment 2

Due: 6.00 pm on Sunday, April 21, 2019

Notes:
You may discuss the problems with your peers but the submitted work must be your own work. Please submit your report and source code through blackboard. This assignment is worth 20% of your total grade. **The project can be done individually or as a team of 2 (including graduates).** Only one of the group members needs to submit the project on blackboard.

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/debugging parallel codes under message passing can be more time consuming than under threads, give yourself 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. The performance results will be collected on the KUACC cluster. For instructions about how to compile and run MPI jobs on KUACC, please read the Assignment and Environment sections carefully.

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 purposes. 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 as the Aliev-Panfilov model, that maintains 2 state variables, and solves one PDE. This simple model can account for complex behavior such as 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 $kk, a, b, \epsilon, M1$ and $M2$ 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

We are providing 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 performance results will be taken **with the plotting is disabled.** The simulator has various options as follows.

```
./cardiacsim

With the arguments
-t <float> Duration of simulation (time units)
-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 are contiguous 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 legal as 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 ON** on KUACC cluster. Please strictly follow the instructions below.

    - You will be utilizing 32 processor cores for the pure MPI and hybrid OpenMP+MPI performance studies. To make performance results more consistent, we have created a group of E5-2965 v4 processors on which you will be running your jobs. For instructions on how to use this group, refer to the environment section.

- 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 on a single node 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 32 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) Repeat the strong scaling study on two KUACC nodes with number of tasks per node equal to 16. Compare the single node performance against dual node performance. Explain your findings.

(c) 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. Conduct this part of the study only on 16 cores with different process geometries. Do not use OpenMP in this performance study.

(d) Repeat single and dual node performance study with OpenMP parallelism turned ON by using 2, 4, 8, and 16 OpenMP threads per process. 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.

(e) Conduct a weak scaling study. Observe the running time as you successively double the number of cores, starting at P=1 core with $N = 256$ and $i = 10000$ and ending at 32 cores on two nodes, growing the workload (N) in proportion to P. However, you will need to decrease number of iterations, since the running time grows as N^2.

Suggestions: We recommend you to start with 1D geometry, test its correctness, add OpenMP, test its correctness and then add support for 2D geometry, then optimize the code.

Environment

We will be using KUACC cluster for performance studies. However, if you want to develop and test the application on your local machine:

- 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. One MPI implementation is available here: http://www.open-mpi.org/. There are other options available. It doesn't matter which implementation you use.

- In order to compile with MPI on your local machine, you need to export paths on your shell or on the development environment (e.g. eclipse). On Unix-based systems, you should add the following lines to your .bashrc file:

```

# If you wish to use intel compiler
source <Path to Intel parallel studio bin folder>/compilervars.sh intel64

# MPI paths
export PATH=<Path to MPI bin folder>:$PATH
export LD_LIBRARY_PATH=<Path to MPI lib folder>:$LD_LIBRARY_PATH
export PATH=<Path to MPI include folder>:$PATH
```

- Accessing KUACC outside of campus requires VPN. You can install VPN through this link: https://my.ku.edu.tr/faydali-linkler/
- In this section, we will briefly describe how to run programs on KUACC cluster. For more details, please refer to http://login.kuacc.ku.edu.tr/
- As previously described in assignment 1, you can log into KUACC cluster using **ssh** with your KU username and password as follows:

```
    bash$ ssh $<$username$>$@login.kuacc.ku.edu.tr
    bash$ ssh dunat@login.kuacc.ku.edu.tr //example
```

- On KUACC, machine you logged into is called login node or front-end node. Please make sure you only use front-end node for compiling and submitting jobs, **not to execute jobs**.
- For faster and stable execution on jobs on KUACC, it is mandatory to run jobs from the scratch folder which is located at /scratch/users/username/. Place your executable, related files and jobscript under scratch folder and submit your jobs as follows. From the login node, change directory to scratch folder as follows:

```
    bash$ cd /scratch/users/username/
```

  then submit the job as follows:

```
    bash$ sbatch <scriptname>.sh
```

- To compile your assignment on the cluster, you can use GCC compiler with OpenMPI (or MPICH) or Intel compiler with Intel MPI. For this purpose, you first need to load respective modules on the cluster as follows:

```
    bash$ sbatch  module avail //shows all available modules in KUACC
    bash$ module list //list currently loaded modules.
    bash$ module load gcc/7.2.1/gcc //loads GNU compiler
    bash$ module load openmpi/3.0.0 // load OpenMPI module
    bash$ module load mpich/3.2.1   // load MPICH module
    bash$ module load intel/ipsxe2019-u1ce //loads Intel compiler
```

Mention the tools you used for the assignment in the report.

- Submit your jobscript with –exclusive option to make sure other jobs do not interfere with your job. You need to add the following line in your jobscript:

```
    #sbatch --exclusive
```

- To request our selected group of processors to run your jobs, use the following line in the jobscript:

```
  #sbatch --constraint=e52695v4,36cpu
```

- Makefile 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.

- For visual verification of your MPI implementation, you can use gnu plotting capability in conjunction with X11 forwarding to view the simulation plots while it runs on the cluster. For this purpose, you should log into KUACC cluster with X11 forwarding as follows:

```
    bash$ ssh -X $<$username$>$@login.kuacc.ku.edu.tr
```

  Once logged into your KAUCC account, request resources for interactive session with X11 forwarding as follows:

```
    bash$ srun -x11 -A users -p short -n1 -qos=users --pty $SHELL
```

  where -n option is used to set number of tasks to be launched. For instances, if you wish to use interactive session for launching up to 4 tasks, use -n4. You might have to wait for the resources to become available before your interactive session begins. Once logged into your interactive session, you can directly run your interactive jobs on the allocated node. An example command line for executing cardiac simulation with 4 MPI processes is listed below:

```
    mpirun -np 4  ./cardiacsim -n 1024 -x 1 -y 4 -t 100
```

  For numerical correctness verification, please refer to the Testing Correctness section of this document.

  DO NOT use interactive session to collect performance data. This is only for testing.

- For mixing OpenMP and MPI, you will request the same number of total processors and then vary OpenMP threads and MPI processes to span over those processors. For instance, when you request 32 processes, you can use them to spawn 32 processes, one OpenMP thread, 16 MPI processes and 2 OpenMP threads and so on.

  Here is an example job script to be used on KUACC:

```bash
#!/bin/bash
#
# You should only work under the /scratch/users/<username> directory.
#
# Example job submission script
# -= Resources =-
#
#SBATCH --job-name=cardiac-sim
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=16
##SBATCH --partition=short
#SBATCH --exclusive
#SBATCH --constraint=e52695v4,36cpu
#SBATCH --time=30:00
#SBATCH --output=cardiacsim-%j.out
#SBATCH --mail-type=ALL
# #SBATCH --mail-user=nahmad16@ku.edu.tr

## Load openmpi version 3.0.0
echo "Loading openmpi module ..."
module load openmpi/3.0.0

## Load GCC-7.2.1
echo "Loading GCC module ..."
module load gcc/7.3.0

echo ""
echo "======================================================================"
env
echo "======================================================================"
echo ""

# Set stack size to unlimited
echo "Setting stack size to unlimited..."
ulimit -s unlimited
ulimit -l unlimited
ulimit -a
echo

echo "Serial version ..."
./cardiacsim -n 400 -t 100

echo "2 MPI + 8 OpenMP"
export OMP_NUM_THREADS=8
mpirun -np 2 ./cardiacsim -o 8
```

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 over all 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 blackboard.
- 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: correctness, performance, and explanations of observed performance in your report.

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

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

GOOD LUCK.
Reference: https://www.simula.no/publications/stability-two-time-integrators-aliev-panfilov-system.