COMP 429/529- Parallel Programming: Assignment 3

Due: January 12, 2024

Notes: You may discuss the problems with your peers but the submitted work must be your own work. Late assignment will be accepted with a penalty of -10 points per day up to three days. Please submit your source code through blackboard. This assignment is worth 20% of your total grade. **You have the option to do this project either individually or with a partner. However, team sizes are limited to two only.**

Corresponding TA: Ilyas Turimbetov (iturimbetov18@ku.edu.tr)

TA office hours: Tuesdays 17:30-19:00, Thursdays 13:00-14:30 or by appointment if preferred, Location Eng230

Cardiac Electrophysiology Simulation

In this assignment you'll implement the Aliev-Panfilov heart electrophysiology simulator using MPI and OpenMP. Along with this project description, studying Lecture 4.3 and Lecture on Stencils 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 (I am giving you more than enough time). 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 the cluster, 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 timing 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 relevant MPI lectures for explanation about ghost cells.

Assignment

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

Part I: MPI Parallelization with 1D Geometry

Parallelize the simulator using MPI by supporting one dimensional processor geometry in this part. Under 1D geometry conditions (only -y is set), thus ghost cells are contiguous in memory. Your code must correctly handle the case when the process geometry doesn't evenly divide the size of the mesh. You only need to support square meshes.

Part II: MPI + OpenMP Parallelism

Add OpenMP parallelism within an MPI process to support hybrid parallelism with MPI + OpenMP.

Part III: MPI Parallelization with 2D Geometry

In 2D geometry, you add support for two dimensional process geometry. In this part, your implementation needs to pack and unpack discontiguous memory locations to create messages. The command line flags -x -y have been set up to specify the process geometry.

Suggestions: I would start with 1D geometry, test correctness, add OpenMP, test correctness and then support for 2D geometry, test correctness, then optimise the code. However, her yiğidin bir yoğurt yiyişi vardır. Please translate this proverb to your peers.

Part IV: Performance Study

Conduct a performance study **without the plotter is on**. Please strictly follow the instructions below.

1. Study 1: 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. Compare single process performance of your parallel MPI code against the performance of the original provided code. Determine optimal processor geometry. Do not use OpenMP in this performance study.
2. Study 2: Conduct the same strong scaling study above this time with 2D geometry. Determine optimal processor geometry.
3. Study 3: Using the indirect method by disabling communication, measure communication overhead. Shrink n by a factor of 2 so that the workload shrinks by a factor of 4. Measure the communication overhead for this reduced problem size and repeat, until $n = 64$. Conduct this part of the study only on 16 cores with different process geometries (x,y) = (1,16) | (2,8) | (4,4) | (8,2) | (16,1). Do not use OpenMP in this performance study.
4. Study 4: Turn on OpenMP parallelism and conduct the same strong scaling study by using 2, 4, 8 OpenMP 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.

Use Gflops rates when you report performance NOT the execution time !!!

Environment

- We will be conducting experiments within a single compute node. In KUACC, there are compute nodes with 4 to 64 cores, although the majority has more than 16 cores in the shorter queue. To ensure consistent performance data, please run all your jobs on a compute node with 16 or more cores. We request that you include the following command in your job script to exclude all compute nodes with fewer than 16 cores.

```

# For running 16 MPI processes within a node
srun --qos=shorter --partition=shorter -N 1  #command continues in the next line
   --ntasks-per-node=16 --cores-per-socket=16 -t 00:05:00 --pty bash

# For running 8 OpenMP threads/MPI process with 2 MPI processes within a node
# (total 16 threads, 2 MPI processes)
srun --qos=shorter --partition=shorter -N 1 #command continues in the next line
   --cpus-per-task=8 --ntasks-per-node=2  --cores-per-socket=16 -t 00:05:00 --pty bash

# For running 2 OpenMP threads/MPI process with 8 MPI processes within a node
# (total 16 threads, 8 MPI processes)
srun --qos=shorter --partition=shorter -N 1 #command continues in the next line
   --cpus-per-task=2 --ntasks-per-node=8  --cores-per-socket=16 -t 00:05:00 --pty bash
```

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. Download the MPI library from http://www.open-mpi.org/ . There are other options available. It does not 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:

```

#load intel compiler if it is not the default 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

# MPI Paths on KUACC
module load openmpi/4.0.1
export PATH=/kuacc/apps/openmpi/4.0.1/bin/:$PATH
export LD_LIBRARY_PATH=/kuacc/apps/openmpi/4.0.1/lib/:$LD_LIBRARY_PATH
export PATH=/kuacc/apps/openmpi/4.0.1/include:$PATH
```

- 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.
- Note that on KUACC, you won't be able to use the gnu plot with an interactive interface because there is no X-forwarding from compute-nodes. You can test your program with comparing the L2 norms. See the 'Testing Correctness' section for details.

- Running an MPI program requires you to use a wrapper script (mpirun) by indicating the number of processes to be used:

```

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

- For mixing OpenMP and MPI, you still need to set the ntask-per-node to 16, and then specify the number of processes with mpirun. The simulation should take the number of threads per processor as an argument. We are providing a sample job script that binds MPI processes to the cores and binds OpenMP threads in a compact fashion.

Sample Job Script

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

```
#!/bin/bash
#
# You should only work under the /scratch/users/<username> directory.
#
# Example job submission script
#
# -= Resources =-
#
#SBATCH --job-name=cardiacsim
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=16
#SBATCH --cores-per-socket=16
#SBATCH --partition=short
#SBATCH --time=00:30:00
#SBATCH --output=cardiacsim.out

module load openmpi/4.0.1
export PATH=/kuacc/apps/openmpi/4.0.1/bin/:$PATH
export LD_LIBRARY_PATH=/kuacc/apps/openmpi/4.0.1/lib/:$LD_LIBRARY_PATH
export PATH=/kuacc/apps/openmpi/4.0.1/include:$PATH

################################################################################
#################### !!! DO NOT EDIT ABOVE THIS LINE !!! #######################
################################################################################


echo "Running Job...!"
echo "==============================================================================="
echo "Running compiled binary..."

#serial version
lscpu
echo "Serial version..."
./cardiacsim -n 1024 -t 100

```

```
#parallel version
echo "Parallel version with 1 process"
mpirun -np 1 ./cardiacsim -n 1024 -t 100 -y 1

echo "Parallel version with 2 processes"
mpirun -np 2 ./cardiacsim -n 1024 -t 100 -y 2 -x 1

echo "Parallel version with 4 processes"
mpirun -np 4 ./cardiacsim -n 1024 -t 100 -y 4 -x 1

...


#Different configuration of MPI+OpenMP
#[1 + 16]  [2 + 8]   [4 + 4]   [8 + 2]  [ 1 + 16]

export KMP_AFFINITY=verbose,compact

echo "MPI1 + OMP16"
export OMP_NUM_THREADS=16
mpirun -np 1 ./cardiacsim [program arguments]

echo "MPI2 + OMP8"
export OMP_NUM_THREADS=8
export SRUN_CPUS_PER_TASK=8
mpirun -np 2 -cpus-per-proc 8 ./cardiacsim [program arguments]

...
```

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 process 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: performance, correctness and the depth and your explanations of observed performance in your report.

- 40 pts. MPI 1D partitioning implementation and its related performance study (study 1 and 3),
- 10 pts. OpenMP parallelism and its related performance study (study 4)
- 30 pts. MPI 2D partitioning implementation and its related performance study (study 3),
- 20 pts. Implementation and Performance Study Report: (i) brief implementation description and any tuning/optimisation you performed (5 pts) (ii) 1D and 2D strong scaling study plots and your observations (5 pts), (iii) communication vs computation study plots and your observations (5 pts), (iv) OpenMP + MPI result plots and your observations (5 points).

GOOD LUCK.

References

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