SIENAcollege
Computer Science

Computer Science 335
Parallel Processing and High Performance Computing
Siena College
Fall 2021

Programming Project 1: Introduction to Jacobi Iteration

Due: 4:00 PM, Friday, September 17, 2021

In this programming project, you will be introduced to a computation we will be using as a case study from time to time this semester. To start, you will write a Java program that solves Laplace's equation on a two-dimensional, uniform, square grid, using Jacobi iteration. Don't worry if none of those terms make any sense – this document tells you what little you need to know about the math and physics.

You may work alone or with one or two partners on this programming project.

Learning goals:

1. To gain familiarity with a computation using Jacobi iteration.

Getting Set Up

You will receive an email with the link to follow to set up your GitHub repository `jacobiintro-proj-yourg` for this Programming Project. One member of the group should follow the link to set up the repository on GitHub, then that person should email the instructor with the other group members' GitHub usernames so they can be granted access. This will allow all members of the group to clone the repository and commit and push changes to the origin on GitHub. At least one group member should make a clone of the repository to begin work.

All GitHub repositories must be created with all group members having write access and all group member names specified in the `README.md` file by 4:00 PM, Monday, September 13, 2021. This applies to those who choose to work alone as well!

Some Background

Laplace's equation is an elliptic partial differential equation that governs physical phenomena such as heat. In two dimensions, it can be written

$$\Phi_{xx} + \Phi_{yy} = 0.$$

Given a spatial region and values for points on the boundaries of the region, the goal is to approximate the steady-state solution for points in the interior. We do this by covering the region with an evenly-spaced grid of points. A grid of 8×8 would look like this:

```
* * * * * * * * * *
* . . . . . . . . *
* . . . . . . . . *
* . . . . . . . . *
* . . . . . . . . *
* . . . . . . . . *
* . . . . . . . . *
* . . . . . . . . *
* . . . . . . . . *
* * * * * * * * * *
```

The 8×8 grid represented by the dots is surrounded by a layer of boundary points, represented by the $\star$'s. Each interior point is initialized to some value. Boundary points remain constant throughout the simulation. The steady-state values of interior points are calculated by repeated iterations. On each iteration, the new value of a point is set to a combination of the old values of neighboring points. The computation terminates either after a given number of iterations or when every new value is within some acceptable difference ϵ of every old value.

There are several iterative methods for solving Laplace's equation. Your program is to use Jacobi iteration, which is the simplest and, as we will see later, easily parallelizable. However, it is certainly not the most efficient in terms of convergence rate.

In Jacobi iteration, the new value for each grid point in the interior is set to the average of the old values of the four points left, right, above, and below it. This process is repeated until the program terminates. Note that some of the values used for the average will be boundary points. Values of boundary points never change.

Details, Tips, and Requirements

- To avoid special cases at the boundaries, you should allocate your grid for an $n \times n$ simulation to be $(n+2) \times (n+2)$, so you can use row and column 0 and row and column $n+1$ to store the boundary values.
- You will need two copies of the grid, one to store the "current" solution values and one to store the "next" solution values. Do not copy the "next" solution to the "current" solution at each step. Instead, always do two iterations. The first uses one grid as "current" and the other as "next," then the second swaps their roles. This is an example of a simple *loop unrolling*.
- Each grid cell computation looks something like this:

```
nextgrid[i][j] = (grid[i-1][j] + grid[i+1][j] +
                  grid[i][j-1] + grid[i][j+1]) * 0.25;
```

  Computing `* 0.25` is usually faster than `/ 4` since multiplication is a faster operation for a computer than division.

- After each pair of iterations, the corresponding values in each cell of the two grids are compared, and the maximum such value is computed. If this value is less than ϵ, the computation terminates. Otherwise, it continues.
- A maximum number of iteration pairs is also specified, after which the computation stops even if the error tolerance has not been reached.
- Your program should take three required command-line arguments: the size of the grid, the maximum number of iterations and the error tolerance ϵ.
- Initialization of the boundary and interior will determine the exact problem being solved. For now, simply initialize your interior to all 0's and the boundary to have two sides set to 1, two sides set to 0, as follows:

```
1 1 1 1 1 1 1 1 1 1
1 . . . . . . . . 0
1 . . . . . . . . 0
1 . . . . . . . . 0
1 . . . . . . . . 0
1 . . . . . . . . 0
1 . . . . . . . . 0
1 . . . . . . . . 0
1 . . . . . . . . 0
1 0 0 0 0 0 0 0 0 0
```

  This keeps the initialization simple, but still allows for some work during the solution phase. We will introduce ways to solve more interesting variations of this problem later.
- At the end of the computation, print out the total number of iteration pairs needed, the last "maximum difference" value you computed, and the time taken to achieve the solution.
- During development and debugging, you will want to be able to print the intermediate states and the final solution. However, for large runs this is unreasonable. Of course, we would like to be able to see some results. So your program should take an optional fourth command-line parameter.
  - If no fourth command-line parameter is specified, your program should not produce any solution information.
  - If the fourth command-line parameter is specified as `"-"`, your program should print a grid of all final solution values to the terminal at the end of the simulation.
  - If the fourth command-line parameter is specified as any other string, it should be used as the name of a file to create, containing a grid of the final solution values.

Examples

For example, the reference solution produces the following outputs for the given inputs.

```
java Jacobi 10 10 .001 -

Completed 10 iteration pairs, last maxDiff 0.0173376446, 0 ms
1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000
1.000 0.941 0.887 0.841 0.807 0.782 0.762 0.742 0.710 0.641 0.471 0.000
1.000 0.887 0.783 0.697 0.632 0.584 0.549 0.517 0.473 0.394 0.248 0.000
1.000 0.841 0.697 0.577 0.487 0.422 0.377 0.340 0.297 0.235 0.136 0.000
1.000 0.807 0.632 0.487 0.377 0.301 0.250 0.212 0.178 0.134 0.074 0.000
1.000 0.782 0.584 0.422 0.301 0.217 0.164 0.128 0.101 0.073 0.039 0.000
1.000 0.762 0.549 0.377 0.250 0.164 0.110 0.077 0.056 0.038 0.020 0.000
1.000 0.742 0.517 0.340 0.212 0.128 0.077 0.048 0.031 0.019 0.010 0.000
1.000 0.710 0.473 0.297 0.178 0.101 0.056 0.031 0.017 0.010 0.005 0.000
1.000 0.641 0.394 0.235 0.134 0.073 0.038 0.019 0.010 0.005 0.002 0.000
1.000 0.471 0.248 0.136 0.074 0.039 0.020 0.010 0.005 0.002 0.001 0.000
1.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000

java Jacobi 100 100 .001

Completed 100 iteration pairs, last maxDiff 0.0018007141, 41 ms

java Jacobi 100 1000 .001

Completed 180 iteration pairs, last maxDiff 0.0009998158, 30 ms

java Jacobi 1000 10000 .000001

Completed 10000 iteration pairs, last maxDiff 0.0000180076, 69429 ms
```

Submission

Commit and push!

Grading

The program will be graded as a programming assignment.

This assignment will be graded out of 40 points.

Feature	Value	Score
Command-line parameters	3	
Grid allocation	5	
Jacobi iterations	5	
Stop based on iteration count	3	
Stop based on error tolerance	4	
Print simulation stats	4	
Output solution to terminal	2	
Output solution to file	3	
Documentation	6	
Style	5	
Total	40	