Computer Science 335
Parallel Processing and HPC
Siena College
Fall 2024

Programming Project 1: Introduction to Jacobi Iteration

Due: 4:00 PM, Friday, September 13, 2024

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.
2. To prepare a version of a Jacobi iteration simulator for parallelization in future projects.

Getting Set Up

In Canvas, you will find a link to follow to set up your GitHub repository, which will be named `jacobiintro-proj-yourgitname`, for this programming project. Only one member of the group should follow the link to set up the repository on GitHub, then others should request a link to be granted write access.

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 9, 2024. 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 (we will think of them as temperatures) for points on the boundaries of the region, the goal is to approximate the solution for points in the interior, which we can think about as the steady-state temperatures that will result after enough time has passed. 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, often 0. Boundary points are given values that do not change 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.

A Jacobi Simulation in Java

A Java program that simulates this process is provided in your repository. This version includes a simple graphical user interface (GUI) and basic visualization capabilities to help you understand the problem and this implementation.

Some items to note about this program:

- To avoid special cases at the boundaries, it allocates the grid for an $n \times n$ simulation to be $(n + 2) \times (n + 2)$, so it can use row and column 0 and row and column $n + 1$ to store the boundary values.
- It has two copies of the grid, one to store the "current" solution values and one to store the "next" solution values. It does not copy the "next" solution to the "current" solution at each step. Instead, it always does 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.
- Initialization of the boundary and interior determine the exact problem being solved. This version simply initializes the 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 . . . . . . . . 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.

Take some time to make sure you understand this program well before proceeding on the required modifications.

Required Modifications

Your task is to write a version of this simulator that will be suitable for later parallelization and for use in timing studies. Your program should be named `Jacobi.java`. Your version should not create a GUI, nor should it have a graphical display of the grid.

You may use the provided program as a starting point, as a guide, or not at all. Cite your usage appropriately. You may use AI assistance, but all steps must be documented and should be git commits with an appropriate citation in each commit message.

- Instead of setting simulation parameters using a GUI, your program should take three required command-line arguments: the size of the grid, the maximum number of iterations and the error tolerance ϵ.

- 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, reported in milliseconds. The time should only include the iterations toward the solution, not initialization or any final output.
- 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.
- Improve the documentation throughout to demonstrate your understanding of code you used as a starter or model, and to describe your modificatios.

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 final.dat

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

And the final solution grid is written to the file `final.dat`.

A Brief Timing Study

Once your program is finished, use it for an informal timing study. Once we start parallelizing programs we will be conducting many more timing studies. Please run your program on `noreaster.teresco.org` for consistency when you are gathering your timings. We will be varying grid sizes and keeping the number of iterations constant across trials. Therefore, you will want a very small error tolerance (ϵ) so that the simulations always end because of the iteration limit, not because of the error tolerance.

Find a combination of grid size and number of iterations that runs for between 0.5 and 1.5 seconds. We will use this as our baseline. Run this on noreaster 5 times and take the fastest run. Keep doubling the grid size (if your first was $n \times n$, the next should be $2n \times 2n$, then $4n \times 4n$, etc.) and gathering timings until the elapsed time exceeds 2 minutes. Again, for each problem size, do 5 runs and take the fastest.

Present your results in tabular form in your repository's `README.md` file. After the table, describe your results in a few sentences and explain how they (hopefully) make sense.

Submission

Commit and push!

Grading

This assignment will be graded out of 40 points.

Feature	Value	Score
GUI and graphics removed	3	
3 basic command-line parameters	5	
Timer accuracy	2	
Print simulation stats	4	
Output solution to terminal	4	
Output solution to file	4	
Documentation	5	
Style	3	
Timing Study	10	
Total	40	