SIENAcollege
Computer Science

Computer Science 400
Parallel Processing
Siena College
Fall 2017

# Programming Project 1: Jacobi Iteration

**Due: 11:59 PM, Friday, September 22, 2017**

Your task this week is to write a C program using 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 a partner on this project.

---

## Getting Set Up

You will receive an email with the link to follow to set up your GitHub repository `jacobi-project-yourgit` 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.

---

## 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 $8 \times 8$ would look like this:

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

The $8 \times 8$ grid represented by the dots is surrounded by a layer of boundary points, represented by *'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 $\epsilon$ 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 easily parallelizable, though 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.

---

## 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 $\epsilon$, the computation terminates. Otherwise, it continues.
- A maximum number of iterations is also specified, after which the computation stops even if the error tolerance has not been reached.
- Your program should take two command-line arguments: the maximum number of iterations and the error tolerance $\epsilon$.
- Initialization of the boundary and interior will determine the exact problem being solved. I suggest initializing 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.

- At the end of the computation, print out the total number of iterations needed, 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. Use `#ifdef` / `#endif` blocks to allow that code to be included or not by including or not `-D` flags in your `Makefile`.
- For a small amount of extra credit, you can extend your program to include extra features, such as support for more interesting initial conditions and boundary conditions.

---

## Submitting

Your submission requires that all required deliverables are committed and pushed to the master for your repository on GitHub.

---

## Grading

The program will be graded as a programming assignment.

This assignment is worth 50 points, which are distributed as follows:

| Feature | Value | Score |
|---|---|---|
| Correct `Makefile` | 1 | |
| Command-line parameters | 2 | |
| Grid allocation | 3 | |
| Jacobi iteration | 20 | |
| Stops based on iteration limit | 2 | |
| Stops based on error tolerance | 2 | |
| Print states/solution protected by `#ifdefs` | 3 | |
| Print simulation stats | 4 | |
| Documentation | 8 | |
| Code Style | 5 | |
| Total | 50 | |