Assignments in Shared Memory Parallel Computation
IPDC, Spring 2010

You can work in pair with another student in both assignments.

ASSIGNMENT 1: LAPLACE'S EQUATION

Laplace's equation is a partial differential equation that governs physical phenomena such as heat. For two dimensions it can be written as

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 .$$

When the equation models heat conduction, $u(x, y)$ denotes the temperature at a given point in the plane. Laplace's equation is widely used in scientific applications, for example weather forecasting, ocean current simulation and climate modeling.

Your task is to solve this equation using a solution method known as *Red-Black Gauss-Seidel*. 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. The general idea is to cover the region with an evenly spaced grid of points and iteratively approximate the steady-state values at the grid points until the error is below a given threshold.

A grid of 8×8 would look like this:

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

The 8×8 grid presented by dots is surrounded by 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. The computation terminates when every new value is within some acceptable difference *eps* of every old value.

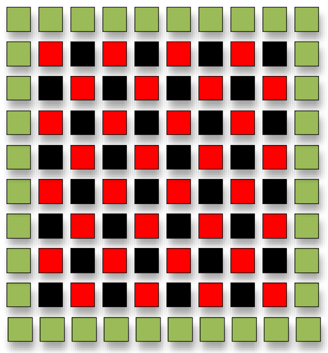
There are many iterative algorithms for calculating the steady-state values. One of these is Jacobi Iteration, where at iteration k, the new value of an interior point u_{ij} is set to the average of the old values of the four points left, above, right, and below it:

$$u_{ij}^{k} = \frac{1}{4}\left[u_{i(j-1)}^{k-1} + u_{(i-1)j}^{k-1} + u_{i(j+1)}^{k-1} + u_{(i+1)j}^{k-1}\right]$$

All values at the same iteration may be updated simultaneously. However, this method converges very slowly, especially for large problem sizes, and is not used to solve practical problems. A clear improvement in efficiency is obtained with *Gauss-Seidel*, where the update procedure at step k may be presented as:

$$u_{ij}^{k} = \frac{1}{4}\left[u_{i(j-1)}^{k} + u_{(i-1)j}^{k} + u_{i(j+1)}^{k-1} + u_{(i+1)j}^{k-1}\right]$$

As seen, each value is of u_{ij} at iteration k is obtained from two newly computed values for iteration k and two old values of iteration k. This minor change to Jacobi Iteration results in much faster convergence, but the values of the same iteration can no longer be updated simultaneously. Fortunately, there are other strategies that converge faster than Jacobi and are more parallelizable than Gauss-Seidel, for example, *Red-Black Gauss-Seidel*. To implement Red-Black Gauss-Seidel, the grid can be arranged in a checker-board ordering of all grid points as illustrated below:



The algorithm first updates all red values using known black values. Then it updates all black values using the new red values:

$$u_{ij}^{k} = \frac{1}{4}\left[u_{i(j-1)}^{k-1} + u_{(i-1)j}^{k-1} + u_{i(j+1)}^{k-1} + u_{(i+1)j}^{k-1}\right] \text{ when } i+j \text{ is even, followed by}$$
$$u_{ij}^{k} = \frac{1}{4}\left[u_{i(j-1)}^{k} + u_{(i-1)j}^{k} + u_{i(j+1)}^{k} + u_{(i+1)j}^{k}\right] \text{ when } i+j \text{ is odd}$$

It is easy to see that all red values can be computed in parallel, and all black values can be computed in parallel.

Problem 1

Write a serial C program that solves Laplace's equation using Red-Black Gauss Seidel on a $n \times n$ grid. Your program should take two command-line arguments: the number of grid points in each dimension, n, and the error tolerance, *eps*.

The interior of the grid should be initialized to all 0's and two borders to 1's, as illustrated for an 8×8 grid below.

```
1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
```

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.

At the end of the computation, print out the total number of iterations needed, and the time taken to achieve the solution (i.e., the time taken for Red-Black Gauss-Seidel Iteration).

Problem 2

Choose values for the size of the grid and *eps* so that your program will run for a few minutes.

Problem 3

Extend your program such that it prints n and the $(n+2)\times(n+2)$ steady-state grid values to a file. Visualize the steady-state solution by giving this file as input to the Java program PlotSolution. Class PlotSolution and its helper class StdDraw can be found in

www.ruc.dk/~keld/teaching/IPDC_f10/Assignments/

Problem 4

Parallelize your serial program using OpenMP. You may break down the computation any way you like. Justify your choice.

Problem 5

Run your OpenMP program on alvin.ruc.dk using 1, 2, 4, 8, and 12 threads, *eps* = 0.001, and at least the following problem sizes: $n = 200, 400, 600$. For each problem plot the timing results as: Time vs. number of threads, speedup vs. problem size. On each graph plot the theoretical linear speedup for reference.

Problem 6

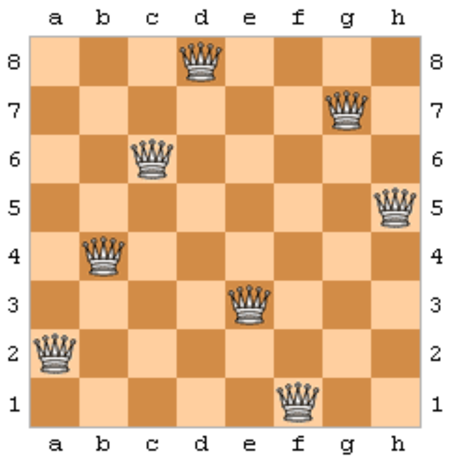
Parallelize your serial program using Java threads. You may break down the computation any way you like. Justify your choice.

Problem 7

Run your Java threads program on alvin.ruc.dk using 1, 2, 4, 8, and 12 threads and at least the following problem sizes: $n = 200, 400, 600$. Report the timings and speedup as in Problem 5.

ASSIGNMENT 2: *N*-QUEENS PUZZLE

The *n*-queens puzzle is the problem of placing *n* chess queens on an *n*×*n* chessboard such that no queen is under attack by another using the standard chess queen's moves, that is, there is at most one queen at each row, column and diagonal. Below is shown one solution to the 8-queens puzzle:



The 8-queens puzzle has 92 distinct solutions. There is currently no known formula for the exact number of solutions. The largest *n*-queens puzzle solved today is a 26-queens puzzle, which has 22,317,699,616,364,044 solutions.

Your task is to write a parallel program for counting the number of solutions to the *n*-queens problem. Use the following recursive function as template for your algorithm:

```
void nqueens(int row) {
    if (row == n)
        solutions++;
    else {
        int col;
        for (col = 0; col < n; col++) {
            if (!underAttack(row, col)) {
                setQueen(row, col);
                nqueens(row + 1);
                removeQueen(row, col);
            }
        }
    }
}
```

Problem 1

Write a serial C program that prints the number of solutions to the *n*-queens problem.

Problem 2

Parallelize your serial program using OpenMP. You may break down the computation any way you like. Justify your choice.

Problem 3

Run your program on alvin.ruc.dk using 1, 2, 4, 8, and 12 threads and at least the following problem sizes: $n = 8, 9, 10$. For each problem plot the timing results as: Time vs. number of threads, speedup vs. problem size. On each graph plot the theoretical linear speedup for reference. Report how large a puzzle your program is able to solve within one minute.

Problem 4

Parallelize your serial program using Java threads. You may break down the computation any way you like. Justify your choice.

Problem 5

Report the performance of the Java program as in Problem 3.

DEADLINE AND MATERIALS TO BE HANDED IN

Friday, April 23, 2010.

For each assignment write a short report that includes the following aspects:

a) Introduction.
b) Reasoning of your parallelization of the program. For example, how did you figure out the parallel regions, why did you use (if any) special OpenMP primitives?
c) Specify anything you have found interesting and didn't think of before.
d) Present and explain your results.
e) Your conclusions.
f) Appendices. Your program code.

Three printed copies of the report and the source code should be handed in by **Friday, April 23, 17.00**. The source code for the programs should be e-mailed to keld@ruc.dk by **23.59** on the same day.