Boston University

EC500: Final Project

Ariya Shajii, Huy Le, Winston Chen

May 1, 2016

1 Introduction

In this project, we solve in two dimensions the heat equation

$$\rho c_p \frac{\partial T}{\partial t} - \nabla \cdot (k \nabla T) = \dot{q}, \tag{1.1}$$

where

- k is a constant taken to be 1,
- ρ is the density of the medium (assumed to be constant),
- c_p is the specific heat of the medium (assumed to be constant),
- $\dot{q}$ is the heat flux as a function of spacial coordinates (x, y), and
- T is the temperature of the material as a function of spatial coordinates (x, y) and of time t.

In discrete form, the equation can be written as follows:

$$\rho c_p (T_{x,y,t+1} - T_{x,y,t}) - k(T_{x+1,y,t} + T_{x-1,y,t} + T_{x,y+1,t} + T_{x,y-1,t} - 4T_{x,y,t}) = \dot{q}. \tag{1.2}$$

The time step and spatial step have both been taken to be 1 in the finite difference approximations.

We now solve this problem using three different approaches:

- Red-black iteration, parallelized with OpenMP and MPI
- Conjugate gradient method
- Using a triangular lattice instead of a conventional square lattice

2 Parallelized Red-Black Iteration

When solving a system of linear equations in terms of the Gauss-Seidel and SOR methods, the order in which the variables are processed matter. A common ordering that can be used to solve the heat equation is red-black. The motivation for red-black is to order the nodes with flexibility for parallel implementation. Thinking of the matrix as a checkerboard, every red node only has black node as its neighbors and every black node only has red nodes as neighbors to its north, south, east, and west.
Each processor uses MPI to send its leftmost solution value to its left neighbor, and its rightmost solution value to its rightmost neighbor. Of course, each processor must then also receive the corresponding information that its neighbors send to it. (However, the first and last processor only have one neighbor, and use boundary condition information to determine the behavior of the solution at the node which is not next to another processor's node.)
The naive way of setting up the information exchange works, but can be inefficient, since each processor sends a message and then waits for confirmation of receipt, which can't happen until some processor has moved to the "receive" stage, which only happens because the first or last processor doesn't have to receive information on a given step.

2.1 Results

6 x86 nodes Intel(R) Xeon (R) CPU 5140 @ 2.33 GHz 4 None

3 Conjugate Gradient Method

In order to use conjugate gradient (CG), we must write our problem in the form $\mathbf{A}\vec{x} = \vec{b}$ for vectors $\vec{x}$, $\vec{b}$ and matrix $\mathbf{A}$. In this case, $\vec{x}$ will be our temperature T, $\vec{b}$ will be our heat flux $\dot{q}$ and $\mathbf{A}$ will encode the left-hand side of Equation 1.1. Note that we take $\frac{\partial T}{\partial t} = 0$ since we are only interested in the steady-state solution when using CG. We are therefore left with:

$$\mathbf{A}T_{x,y} = 4T_{x,y} - (T_{x+1,y} + T_{x-1,y} + T_{x,y+1} + T_{x,y-1}) = \vec{b} = \dot{q}. \tag{3.1}$$

Note that we make the time variable t implicit, since we no longer need it directly. The CG algorithm can now be applied as follows [1]:

[1] Adapted from `https://en.wikipedia.org/wiki/Conjugate_gradient_method#The_resulting_algorithm`

Figure 2.1: Result of applying red-black iteration.

Require: $\mathbf{A}$, $\vec{b}$, ϵ
Ensure: T
  $T \leftarrow \vec{0}$
  $\vec{r} \leftarrow \vec{b} - \mathbf{A}T$ {residual}
  $\vec{p} \leftarrow \vec{r}$
  loop
    $\alpha \leftarrow \frac{|\vec{r}|}{\vec{p} \cdot \mathbf{A}\vec{p}}$
    $T \leftarrow T + \alpha\vec{p}$
    $\vec{r}_{\text{new}} \leftarrow \vec{r} - \alpha\mathbf{A}\vec{p}$
    if $|\vec{r}_{\text{new}}| < \epsilon$ **then**
      return T
    $\beta \leftarrow \frac{|\vec{r}_{\text{new}}|}{|\vec{r}|}$
    $p \leftarrow \vec{r}_{\text{new}} + \beta\vec{p}$
    $\vec{r} \leftarrow \vec{r}_{\text{new}}$

Algorithm 1: Conjugate gradient algorithm

We take our initial guess T_0 to be zero everywhere. All that Algorithm 1 requires us to have is a way to perform dot products and a way of applying $\mathbf{A}$ to a vector. The former is fairly trivial and the latter is given by Equation 3.1. With these fundamental operations at hand, it is straightforward to implement Algorithm 1 in code.

Quick note on boundary conditions: We incorporate our boundary conditions in $\vec{b}$ and apply $\mathbf{A}$ only to the interior points of T – that is, points that have exactly four neighbors.

3.1 Results

Figure 3.1 shows the result of applying the CG algorithm to a system with two opposite point sources on a 200-by-200 grid. The boundaries were fixed at zero and a residual tolerance of 10^{-6} was used. It is also instructive to view the result after each of the first few iterations of the main loop in Algorithm 1. Consider a simpler system with a single point source in the center on a 20-by-20 grid. The result after the first four iterations is shown in Figure 3.2. We can see from this figure how information propagates neighbor-by-neighbor from the source to the surrounding points. This makes sense since, by Equation 3.1, each non-zero point can only influence its closest neighbors on each iteration.

Finally, we compared timings of our conjugate gradient implementation to Gauss-Seidel iteration for several grid sizes, as shown in Figure 3.3. Clearly, conjugate gradient is a substantial algorithmic improvement over Gauss-Seidel.

4 Triangular Lattice

The triangular (or hexagonal) lattice algorithm is similar to the square lattice method used in successive over-relaxation (SOR). Figure 4.1 shows a graphical visualization of

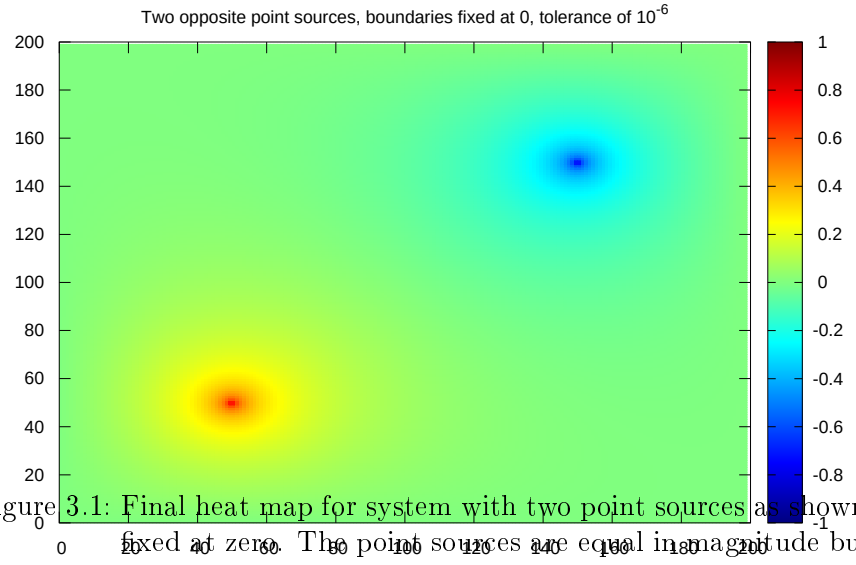


Figure 3.1: Final heat map for system with two point sources as shown, with boundaries fixed at zero. The point sources are equal in magnitude but opposite in sign.

a triangular lattice. While the iterative equation for a square lattice with relaxation parameter 1 is

$$T_{x,y} \leftarrow \frac{T_{x+1,y} + T_{x-1,y} + T_{x,y+1} + T_{x,y-1}}{4} + \dot{q}_{x,y}, \tag{4.1}$$

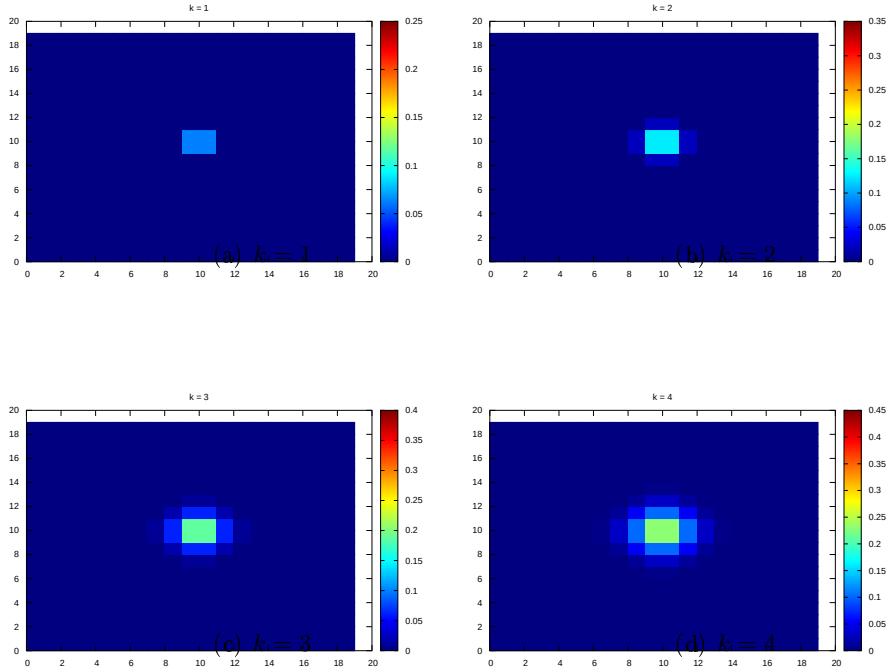
the equivalent equation for a triangular lattice is

$$T_{x,y} \leftarrow \frac{T_{x+1,y} + T_{x-1,y} + T_{x,y+1} + T_{x,y-1} + T_{x+1,y+1} + T_{x-1,y-1}}{6} + \dot{q}_{x,y}. \tag{4.2}$$

4.1 Results

Table 4.1: Table of runtimes for parallelized Jacobi iteration on a triangular lattice for system with a single point source in the center and a tolerance of 10^{-6}. All times are given in seconds

		Threads					
Grid width	**Baseline (serial)**	**1**	**2**	**4**	**6**	**8**	**12**
50	0.215	0.237	0.199	0.154	0.154	0.151	0.157
100	3.359	3.711	2.989	2.300	2.170	2.087	2.088
200	51.066	56.330	45.299	34.489	32.494	30.970	29.947
300	248.500	274.343	223.119	173.063	162.174	153.766	145.72



(a) $k = 1$ (b) $k = 2$ (c) $k = 3$ (d) $k = 4$

Figure 3.2: First four steps of conjugate gradient algorithm for system with a single point source in the center.

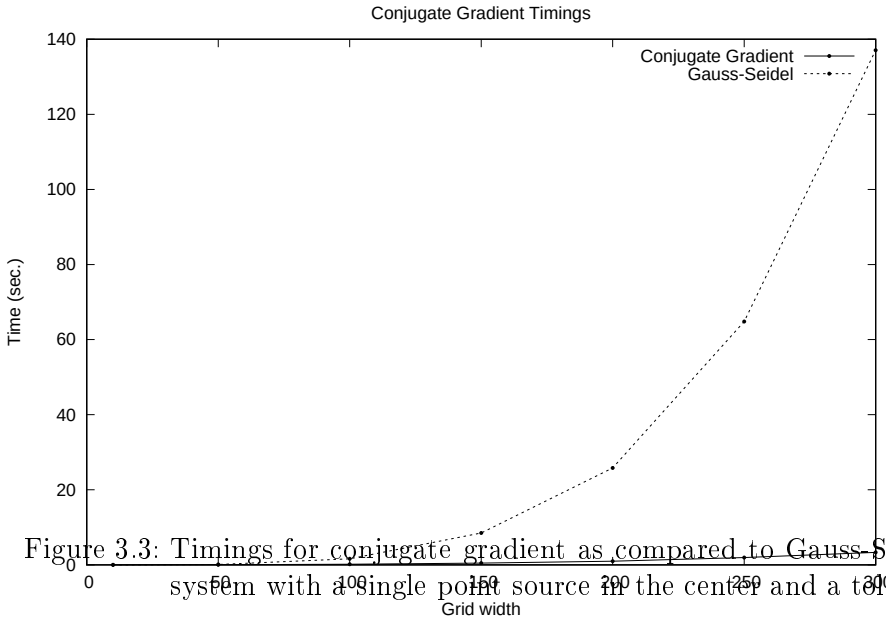


Figure 3.3: Timings for conjugate gradient as compared to Gauss-Seidel iteration for a system with a single point source in the center and a tolerance of 10^{-6}.

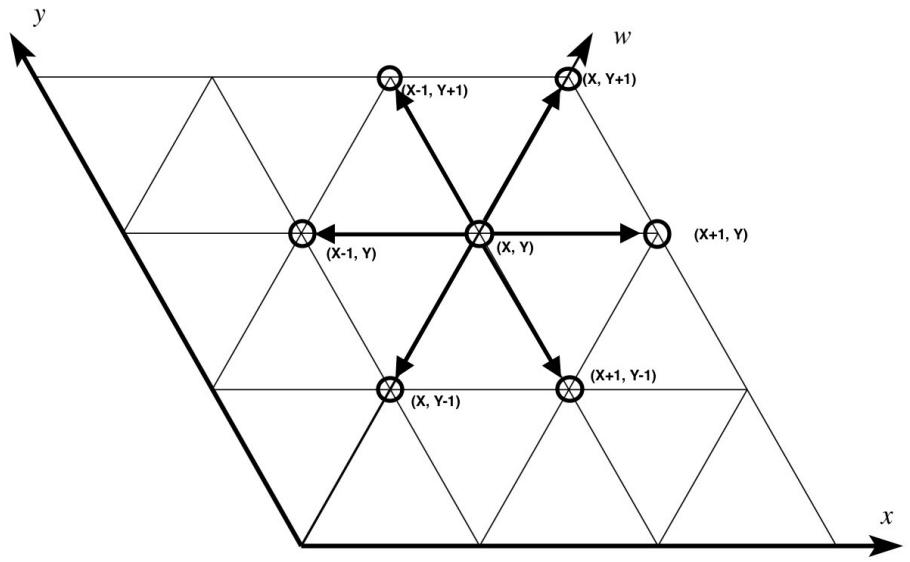


Figure 4.1: Visualization of a triangular lattice.

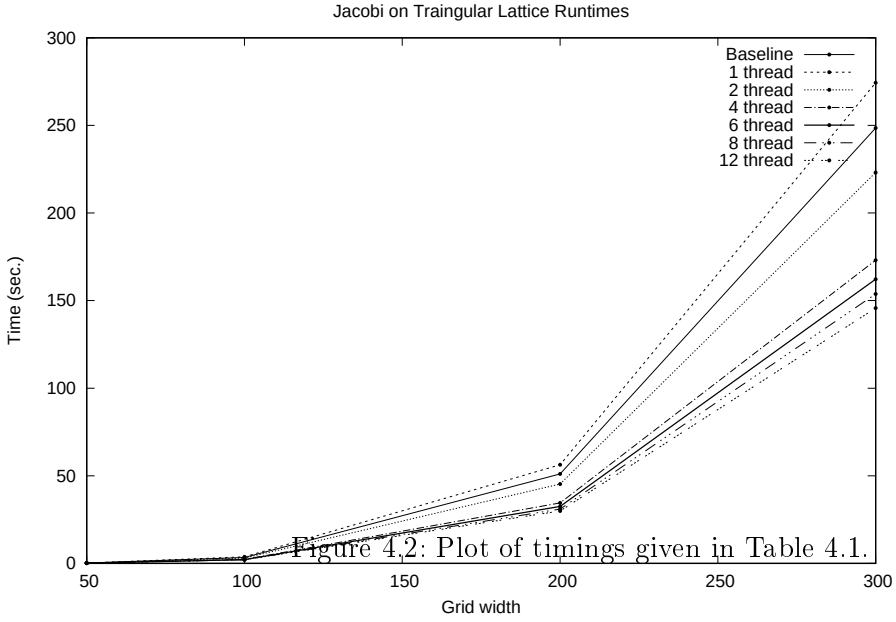


Figure 4.2: Plot of timings given in Table 4.1.

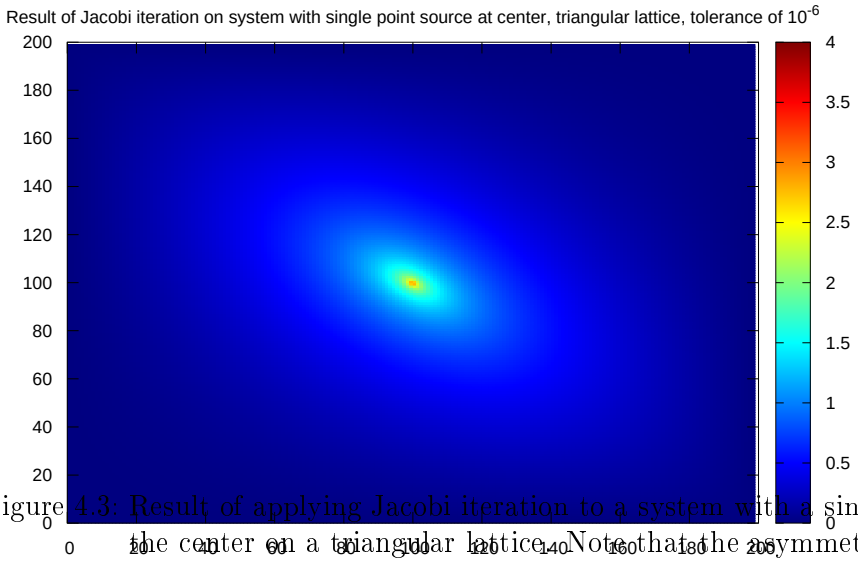


Figure 4.3: Result of applying Jacobi iteration to a system with a single point source at the center on a triangular lattice. Note that the asymmetry is an artifact of how the result is being plotted, since a shear must be applied to use standard plotting methods.

Different methods can be used to implement a triangular lattice algorithm, such as Jacobi, Gauss-Seidel and Red-Black. Jacobi's method is used here to implement multithreading with OpenMP. The timings of our implementation for various numbers of threads is given in Table 4.1 and plotted in Figure 4.2.

5 Miscellaneous

5.1 Code

The code for our project can be found at `https://github.com/winstonchen/ec500`.

5.2 Contributions

- Ariya Shajii – Conjugate gradient
- Huy Le – Parallelized red-black iteration
- Winston Chen – Triangular lattice