Greenwich Conduction Task

Contents

Introduction

Purpose

Documentation explaining the heat conduction differential equation, the finite difference method used to solve the equation (implemented in C++ code), and optimizations applied to improve the code's performance.

The Heat Equation

The heat conduction, or *heat diffusion equation*, describes how heat spreads in a material over time. In two dimensions, the partial differential heat equation is,

$$\frac{\partial T}{\partial t} = k\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right), \tag{1}$$

where k is the *thermal diffusivity constant* and $T(x, y)$ is the spatial the temperature distribution.

The *steady-state heat equation* is given by:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{2}$$

where the heat entering any point is equal to the heat leaving it, resulting in a stable (or steady) temperature distribution, i.e., when $\partial T/\partial t = 0$.

Numerical solution using the finite difference method

The solution to the heat equation can be approximated with a *finite difference method.* Finite difference methods discretize the spatial domain, replacing continuous derivatives with difference quotients.

For a point (i, j) on a 2D grid, the finite difference approximation of the heat equation can be written as:

$$T_{i,j} = \frac{1}{4}\left(T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1}\right) \tag{3}$$

This formula calculates the temperature $T_{i,j}$ as the average of its four neighbouring points. (See Appendix for its derivation).

By iterating this formula across all points in the grid, we can update each point's temperature until reaching a steady-state solution defined as some minimum change tolerance.

Boundary Conditions

The stated problem uses fixed temperature values at the boundaries (Dirichlet boundary conditions):

- $T(0, y, t) = T_{\text{left}}(y, t) = 40°\text{C}$
- $T(L_x, y, t) = T_{\text{right}}(y, t) = 90°\text{C}$
- $T(x, 0, t) = T_{\text{bottom}}(x, t) = 30°\text{C}$
- $T(x, L_y, t) = T_{\text{top}}(x, t) = 50°\text{C}$

These values are set initially and remain constant throughout the simulation.

Implementation

Parameters and setup

The code takes the following arguments:

- `m`: the number of grid points in the x-direction.
- `n`: the number of grid points in the y-direction.
- `tol`: the tolerance level for the maximum temperature difference, which serves as the stopping criterion for reaching steady state.

The grid `t` is initialized with dimensions $(m + 2) \times (n + 2)$ and filled with an initial temperature of 30°C. Boundary conditions are then applied based on the specified temperatures.

Main Iterative Loop

The main code block implements the following logic:

1. *Temperature Update*: Inside a while loop, each internal grid point $T_{i,j}$ is updated according to the finite difference formula. The updated temperature values are stored in a separate grid `tnew` to prevent interference with the current iteration's calculations.
2. *Convergence Check*: The difference between the old and new temperatures (`diff`) is calculated for each point. The maximum of these differences (`difmax`) is tracked. Once `difmax` is smaller than `tol`, the loop exits, indicating that the solution has converged to a steady state.
3. *Boundary Re-assignment*: After updating temperatures, `tnew` values are assigned back to `t` for the next iteration.

Output

The code outputs:

- The total number of iterations taken to reach convergence.
- The final temperature distribution across the grid, including boundary points.

Example usage

To run the code with a grid of 1000x1000 and a tolerance of 0.01, use:

```
build/gauss 1000 1000 0.01
```

Gauss-Seidel and Jacobi methods

The two methods, *Gauss-Seidel* and *Jacobi*, are iterative techniques used to solve partial differential equations like the heat equation by approximating values on a grid. Both methods calculate a steady-state temperature distribution across the grid but update calculations differently. Below, we discuss the key differences between the Gauss-Seidel and Jacobi methods in the provided codes.

Gauss-Seidel Method

The Gauss-Seidel method updates each grid point's temperature in place, meaning it immediately uses the most recent values within the same iteration. This can lead to faster convergence because each calculation benefits from the latest available information.

Key Characteristics

- *In-Place Update*: When calculating the new temperature for $T_{i,j}$, the Gauss-Seidel method immediately updates the temperature in the original grid `t`.
- *Fast Convergence*: Since the latest values are used immediately, convergence generally occurs faster than with the Jacobi method.
- *Implementation*:
  - The updated values are written directly back into `t`.
  - The maximum difference `difmax` is calculated within the same loop where `t[i][j]` is updated.

Code Snippet (Gauss-Seidel)

Here's a snippet illustrating the Gauss-Seidel approach:

```
for (int i = 1; i <= m; i++) {
  for (int j = 1; j <= n; j++) {
      tnew[i][j] = (t[i-1][j] + t[i+1][j] + t[i][j-1] +
      ↪  t[i][j+1]) / 4.0;
      diff = std::fabs(tnew[i][j] - t[i][j]);
      if (diff > difmax) {
          difmax = diff;
      }
      t[i][j] = tnew[i][j]; // immediate in-place update
  }
}
```

Jacobi Method

The Jacobi method calculates all updated values for the grid in a separate matrix (`tnew`) and only transfers these values to the main grid (`t`) once the entire grid has been processed. This can be more computationally expensive since it doesn't benefit from the latest updates in the current iteration, leading to a slower convergence.

Key Characteristics

- *Separate Grid Update*: Each point's updated temperature is stored in `tnew`, and `t` is only updated at the end of each iteration.
- *Slower Convergence*: Because it uses only the previous iteration's values, convergence is slower than in Gauss-Seidel.
- *Implementation*:
  - The update for `tnew[i][j]` is done in one loop.
  - A second loop copies values from `tnew` to `t` and calculates `difmax`.

Code Snippet (Jacobi)

In the Jacobi code, updates and copying are separated:

```
for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
        tnew[i][j] = (t[i-1][j] + t[i+1][j] + t[i][j-1] +
        ↪  t[i][j+1]) / 4.0;
    }
}

for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
        diff = std::fabs(tnew[i][j] - t[i][j]);
        if (diff > difmax) {
            difmax = diff;
        }
        t[i][j] = tnew[i][j]; // copy tnew to t at the end of
        ↪  each iteration
    }
}
```

Summary of differences

The *Gauss-Seidel method* generally converges faster than *Jacobi* due to its in-place update mechanism, which leverages the latest information during calculations. However, the Jacobi method is easier to parallelize since each update only depends on values from the previous iteration, making it more suitable for certain parallel computing environments. Both methods ultimately reach the same steady-state solution for the heat conduction problem, but Gauss-Seidel is typically more computationally efficient in sequential implementations.

Results

Hardware

All following calculations are run on a Dell Core i5 with 4 cores.

Benchmark

We run the C++ code on a grid of 1000x1000 with a tolerance of 0.01. Figure 1 plots the result.

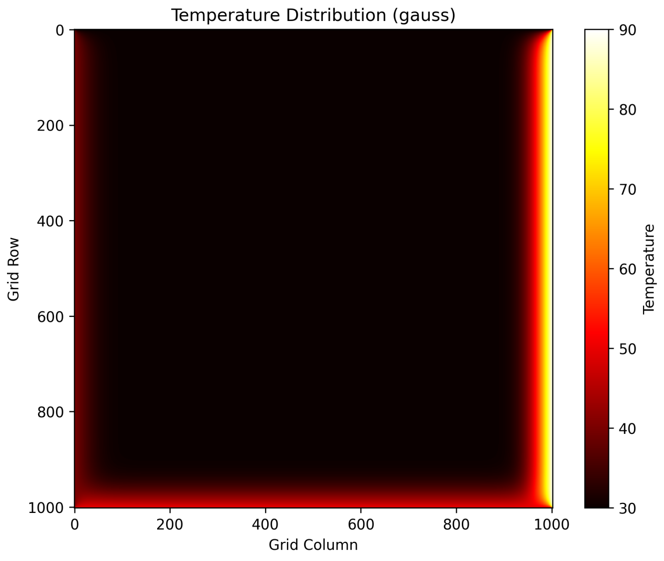

Figure 1: Steady state temperature distribution on a 1000x1000 grid with tolerance 0.01 using the Gauss-Seidel method.

Comparison of run times shows that the execution of the Gauss-Seidel method takes 72 seconds compared to the Jacobi method that takes 79 seconds in agreement with the summary in the previous section.

Optimization

We now apply parallelization to speed up calculations for the two methods. We focus on OpenMP (Open Multi-Processing), an application programming interface (API) that supports multi-platform shared-memory parallel programming in C, C++, and Fortran. OpenMP enables developers to write code that can run in parallel across multiple CPU cores, effectively allowing tasks to be divided and processed simultaneously.

Parallelizing the Jacobi Method

Since the Jacobi method uses a separate grid (`tnew`), updates for each grid point only depend on values from the previous iteration. OpenMP can parallelize the nested loops used to update grid points. Here's an example:

```
// Update temperature for next iteration in parallel
#pragma omp parallel for
for (int i = 1; i <= m; i++)
{
    for (int j = 1; j <= n; j++)
    {
        tnew[i][j] = (t[i - 1][j] + t[i + 1][j] + t[i][j - 1]
        ↪  + t[i][j + 1]) / 4.0;
    }
}
```

Parallelizing the Gauss-Seidel method

The Gauss-Seidel method requires each point to use the most recent values, making it harder to parallelize than the Jacobi method. The outer loop time loop can be parallelized, which updates the temperature and calculates the maximum difference. This allows each iteration to be executed concurrently across multiple threads.

```
// Update temperature and calculate maximum difference
#pragma omp parallel for reduction(max : difmax) private(diff)
for (int i = 1; i <= m; i++)
{
    for (int j = 1; j <= n; j++)
    {
        tnew[i][j] = (t[i - 1][j] + t[i + 1][j] + t[i][j - 1]
        ↪  + t[i][j + 1]) / 4.0;

        // Calculate difference and update difmax in a
        ↪  thread-safe manner
        diff = std::fabs(tnew[i][j] - t[i][j]);
        if (diff > difmax)
        {
            difmax = diff;
        }

        // Copy new value to old temperature array
        t[i][j] = tnew[i][j];
    }
}
```

Implementation

When using libraries like OpenMP, it's important to compile the code with the `-fopenmp` flag:

```
g++ -fopenmp parallel/gauss.cpp -o build/gauss
```

Comparison of runtimes

Applying parallelization roughly halves the run times for both algorithms. The Gauss-Seidel method benefits from parallelization with a runtime reduction from 72.7995 seconds to 42.1727 seconds. The Jacobi method benefits from parallelization with a runtime reduction from 79.0255 seconds to 39.1713 seconds.

Appendix

The heat equation in two dimensions is given by:

$$\frac{\partial T}{\partial t} = k\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) \tag{4}$$

where T is the temperature, t is time, k is the thermal diffusivity, and x and y are spatial dimensions.

To solve this equation numerically, we discretize the spatial domain into a grid with points (i, j), where: i corresponds to the index in the x-direction, and j corresponds to the index in the y-direction. We denote the temperature at point (i, j) as $T_{i,j}$.

Next, replace the continuous derivatives with finite differences. The second spatial derivatives can be approximated using central difference formulas:

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2}, \tag{5}$$

and,

$$\frac{\partial^2 T}{\partial y^2} \approx \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2}. \tag{6}$$

where Δx and Δy are the grid spacings in the x and y directions, respectively.

Substituting the finite difference approximations into the heat equation gives us:

$$\frac{\partial T_{i,j}}{\partial t} = k\left(\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2}\right). \tag{7}$$

Assuming uniform grid spacing ($\Delta x = \Delta y = \Delta$) yields:

$$\frac{\partial T_{i,j}}{\partial t} = k\left(\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta^2}\right) \tag{8}$$

Next, we use an explicit time-stepping method (like Euler's method) to discretize time. We denote the temperature at the next time step as $T_{i,j}^{n+1}$, and at the current time step as $T_{i,j}^{n}$ so that:

$$T_{i,j}^{n+1} = T_{i,j}^{n} + \Delta t \cdot \frac{\partial T_{i,j}}{\partial t} \tag{9}$$

Substituting this into equation 8 gives:

$$T_{i,j}^{n+1} = T_{i,j}^{n} + k\Delta t\left(\frac{T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n}}{\Delta^2} + \frac{T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}}{\Delta^2}\right) \tag{10}$$

For large n, the solution approaches a steady-state where $T_{i,j}^{n+1} = T_{i,j}^{n}$. Again, substituting this into equation 10 yields:

$$T_{i,j} = \frac{1}{4}\left(T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1}\right). \qquad (11)$$