Adaptive Mesh Refinement for Solving the Advection-Diffusion Equation

A High-Performance Computing Approach

Kylie Hefner
CS 616: High Performance Computing
December, 2024

Purpose of the Project

Main Goal

To develop an efficient and accurate solver for the **advection-diffusion equation** using **adaptive mesh refinement (AMR)** and **parallelization** techniques.

Key Objectives

- **Numerical Accuracy:** Improve solution precision for dynamic systems.
- **Computational Efficiency:** Reduce unnecessary computations using AMR.
- **Scalability:** Leverage OpenMP and MPI to handle large-scale problems effectively.

Situation and Challenges

Situation

The advection-diffusion equation models physical phenomena such as:

- Heat transfer
- Pollution dispersion
- Population dynamics in ecosystems

01

Computational Cost

Uniform grids waste resources in areas without high resolution needs

02

Numerical Instabilities

Advection and diffusion can cause unphysical oscillations

03

Large-Scale Problems

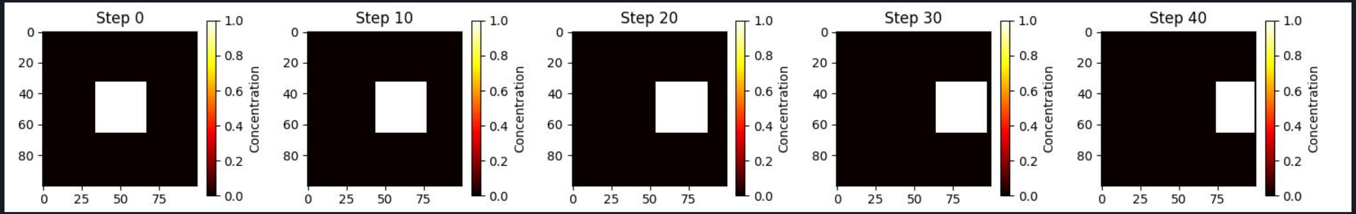
Real-world simulations demand scalability and efficiency

Problems

The Advection-Diffusion Equation

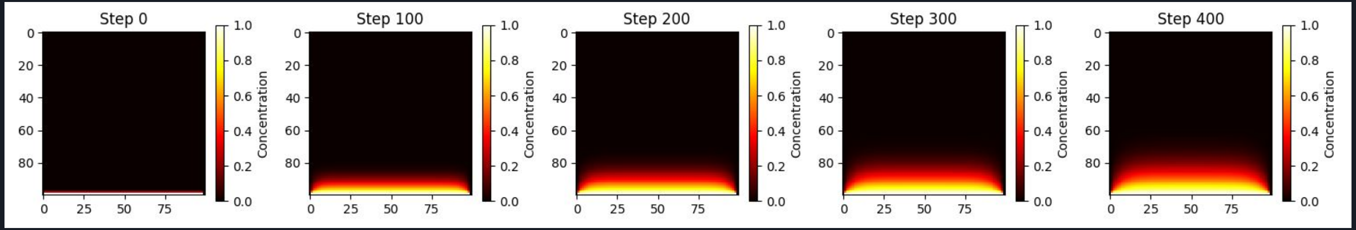
Advection

Describes the transport of a substance (e.g., heat, pollutants, or particles) due to a velocity field



Diffusion

Describes the spreading of a substance due to random motion, such as molecular collisions



The Advection-Diffusion Equation

Velocity Vector

$$\frac{\partial u}{\partial t} = D\nabla^2 u - \vec{v} \cdot \nabla u$$

Diffusion Coefficient

Scalar Field (e.g. concentration or temperature)

Methodology

Steps

Discretization

Apply upwind differencing for advection and central differencing for diffusion in C++

Adaptive Mesh Refinement

Refine grids dynamically in regions with high solution gradients

Parallelization

Use OpenMP and MPI to optimize performance and scalability

Visualization

Export results and generate heatmaps, contour plots, and animations in Python

Discretized Equation

$$u_{i,j}^{n+1} = u_{i,j}^{n} - \Delta t \cdot (v_x \cdot \frac{\partial u}{\partial x} + v_y \cdot \frac{\partial u}{\partial y}) + \Delta t \cdot D \cdot \frac{\partial^2 u}{\partial x^2}$$

To approximate partial derivatives, I used upwind and central differencing

Numerical Methods

Upwind Differencing for Advection

Uses flow direction when calculating spatial derivatives

Prevents instability common with central differencing

$$\frac{\partial u}{\partial x} \approx \frac{u_i - u_{i-1}}{\Delta x}, \quad \text{if } v > 0$$

$$\frac{\partial u}{\partial x} \approx \frac{u_{i+1} - u_i}{\Delta x}, \quad \text{if } v < 0$$

Central Differencing for Diffusion

Approximates second derivatives symmetrically

Ensures numerical accuracy for diffusion process

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}$$

Time Integration

Explicit time-stepping with adaptive $\triangle$t

$\triangle$t is dynamically chosen to satisfy stability criteria

Stability Criteria

Advection Stability (Courant-Friedrichs-Lewy condition)

$$\Delta t < \frac{\Delta x}{max(|v_x|, |v_y|)}$$

Diffusion Stability

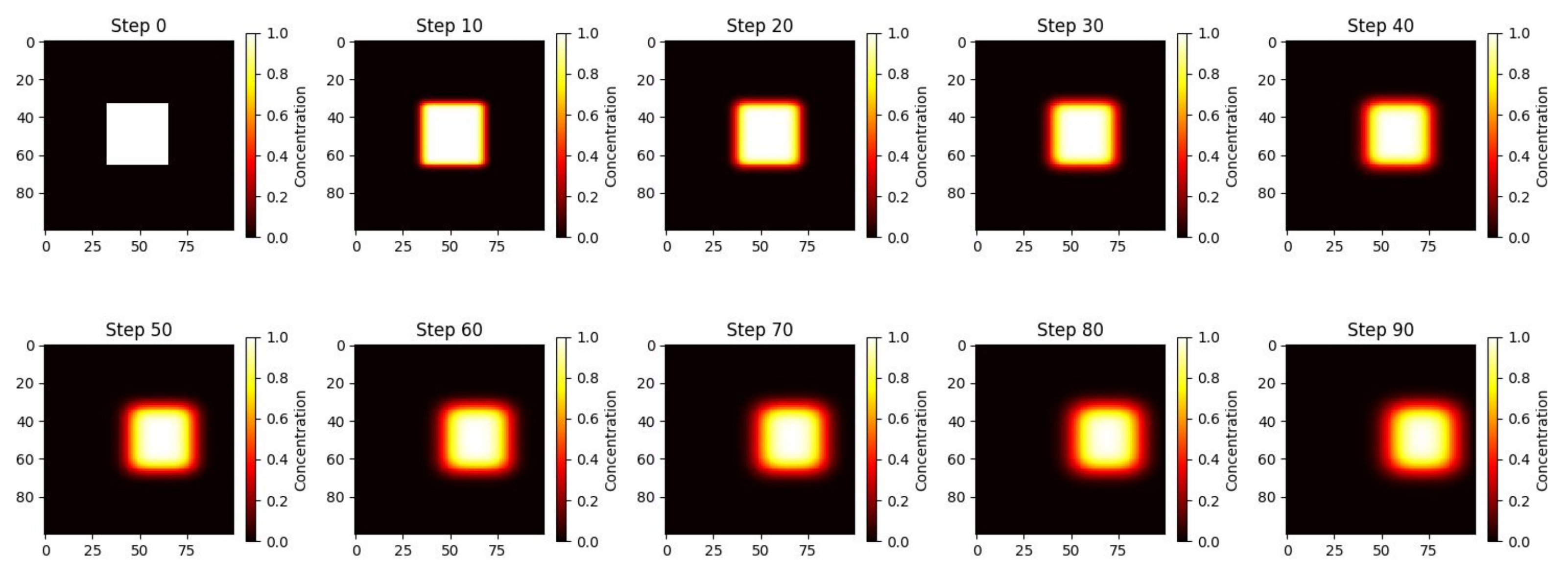
$$\Delta t < \frac{\Delta x^2}{4D}$$

Combined Stability

$$\Delta t < min(\frac{\Delta x}{max(|v_x|, |v_y|)}, \frac{\Delta x^2}{4D})$$

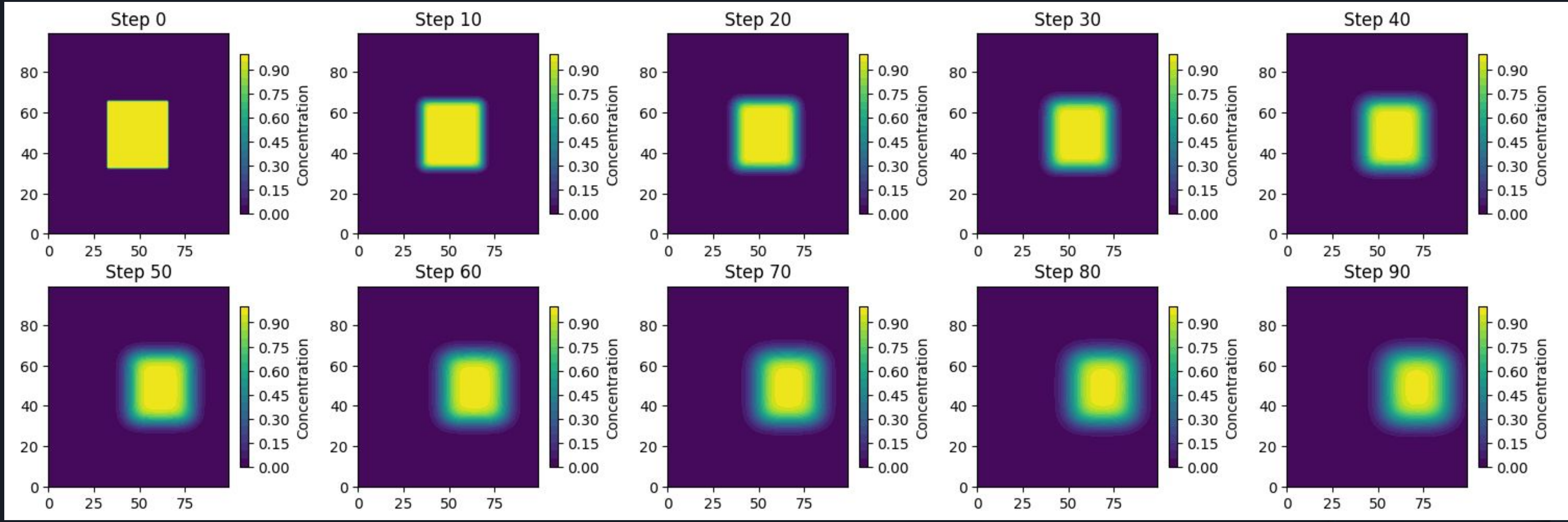
Results

100x100 uniform grid, with heatmap showing the evolution over time.

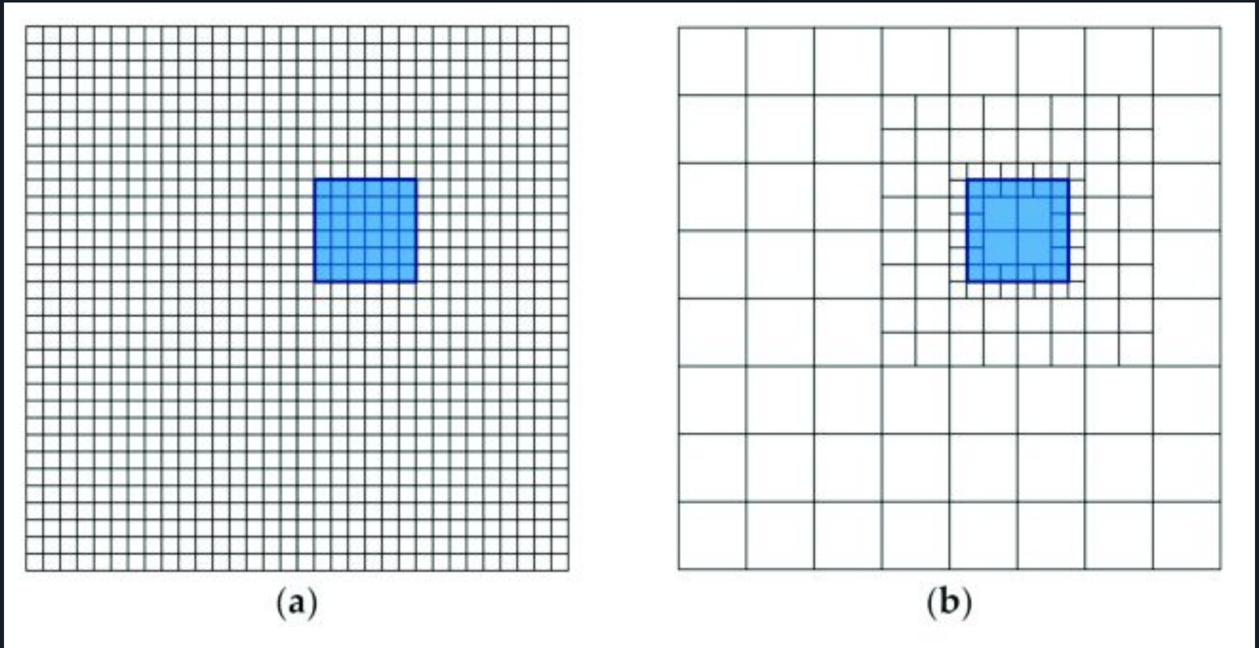


Results

100x100 uniform grid, with contour plot showing the gradient over time.



What if We Saved Computational Power with an Adaptive Grid?



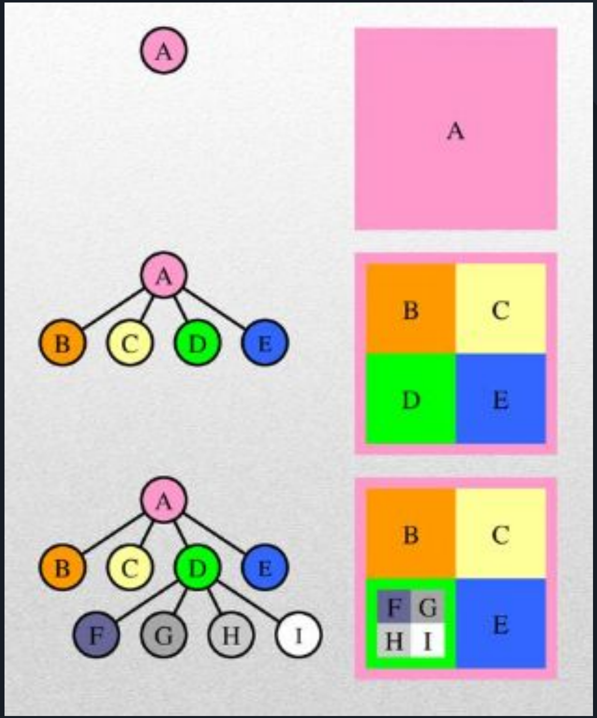
Quadtree Grid for Adaptive Refinement

What is a Quadtree?

- A hierarchical structure that dynamically divides the domain into smaller cells
- Each parent cell can split into four child cells for higher resolution

Why use a Quadtree?

- Reduces computational cost by refining only where needed
- Ensures accuracy in regions with high gradients



Error Estimation Based on Gradients

Objective: Identify regions requiring refinement based on gradients

Error Estimator:

$$\text{Error} = \left|\frac{\partial u}{\partial x}\right| + \left|\frac{\partial u}{\partial y}\right|$$

Gradient Calculation (Central Differencing):

$$\frac{\partial u}{\partial x} \approx \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x}, \quad \frac{\partial u}{\partial y} \approx \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta y}$$

AMR Implementation

1. Calculate Errors:
   a. Use gradients to compute the error for each cell.

2. Refine or Coarsen:
   a. When the error exceeds the threshold, refine cells by dividing a cell into four child cells
   b. When the error is below the lower threshold coarsen cells by merging child cells back into parent cell

3. Update Solution:
   a. Apply numerical methods to the refined grid

Uniform Grid Vs. Adaptive Grid

Uniform Runtime:
1.95264 s

Adaptive Runtime:
5.85792 s

Initially, my adaptive grid solver was significantly slower than my uniform grid solver due to excessive overhead. (200x200 starting grid)

Implemented Fixes:

- Apply refinement and coarsening less often (10 time steps instead of every step)
- Increase the refine and coarsen thresholds to reduce computational cost

Uniform Grid Vs. Adaptive Grid

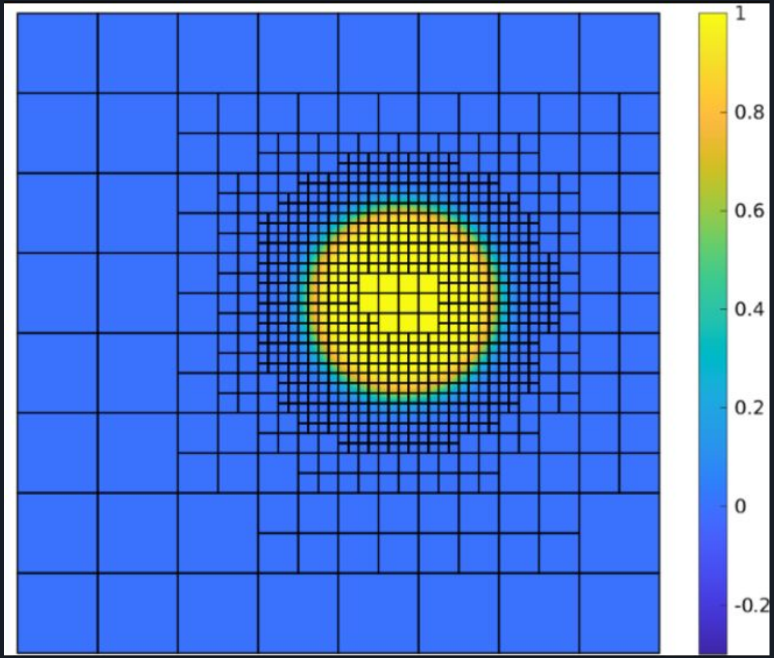
Uniform Runtime:
1.95264 s

Adaptive Runtime:
2.23960 s

After adjusting the refinement and coarsening, the adaptive grid now performs similar to the uniform grid given the same starting size.

Important Note: The runtime with AMR is not always faster when using the same initial grid size as a uniform grid. Instead, the goal of AMR is to focus computational effort on areas of interest (regions with high gradients or errors) while maintaining lower resolution elsewhere.

Uniform Grid Vs. Adaptive Grid



Example Use Case:

- A uniform grid of 64x64 might need to be fully refined to capture small-scale features, leading to high runtime and memory usage.
- With AMR, you can start with a 8×8 grid and refine locally, achieving similar accuracy with fewer cells.

Parallelization with OpenMP

Parallelizing Solution Updates:

- Each advection-diffusion grid cell is updated independently, so we can use openmp threads to parallelize the nested loops

#pragma omp parallel for collapse(2)

- Parallelizes the two nested loops over i and j
- collapse(2) combines the two loops into a single iteration space for efficient scheduling

```
// Function to update the grid values based on advection-diffusion equation
void update_solution(Matrix &grid, const Matrix &velocity_x, const Matrix &velocity_y, c
    int N = grid.size(); // number of grid points
    Matrix new_grid = grid; // create copy of current grid

    // Parallelize the grid update
    #pragma omp parallel for collapse(2)
    for (int i = 1; i < (N - 1); ++i) {
        for (int j = 1; j < (N - 1); ++j) { // iterate over the interior grid points

            // compute advection term using upwind differencing
            double advection_x;
            if (velocity_x[i][j] > 0) {
                advection_x = velocity_x[i][j] * (grid[i][j] - grid[i][j-1]) / dx;
            } else {
                advection_x = velocity_x[i][j] * (grid[i][j+1] - grid[i][j]) / dx;
```

Parallelization with OpenMP

Parallelizing AMR Error Calculation:

- Since each grid cell or quadtree leaf is processed independently during AMR, you can use OpenMP to parallelize the error calculation and refinement/coarsening

#pragma omp parallel for collapse(2)

- Speeding up traversal in the nested loops

#pragma omp critical

- For thread-safe updates to the quadtree

```
void apply_amr(Node* root, const Matrix &grid, double dx, double dy, double refir
    #pragma omp parallel for collapse(2)
    for (int i = 1; i < grid.size() - 1; ++i) {     // loop through interior cell
        for (int j = 1; j < grid[0].size() - 1; ++j) {
            // calculate the error at the current grid cell
            double error = calculate_error(grid, i, j, dx, dy);

            #pragma omp critical // lock to ensure thread-safety
            {
                // refine the cell if the error exceeds the refine threshold
                if (error > refine_threshold) {
                    refine_grid(root);
                }
                // coarsen the cell if the error is below the coarsen threshold
                else if (error < coarsen_threshold) {
                    coarsen_grid(root);
                }
            }
        }
    }
```

Parallelization with OpenMP

Parallelizing Max Velocity Calculation:
#pragma omp reduction(max : max_velocity)

- The reduction clause ensures that OpenMP combines the local maxima from all threads into max_velocity after the loop

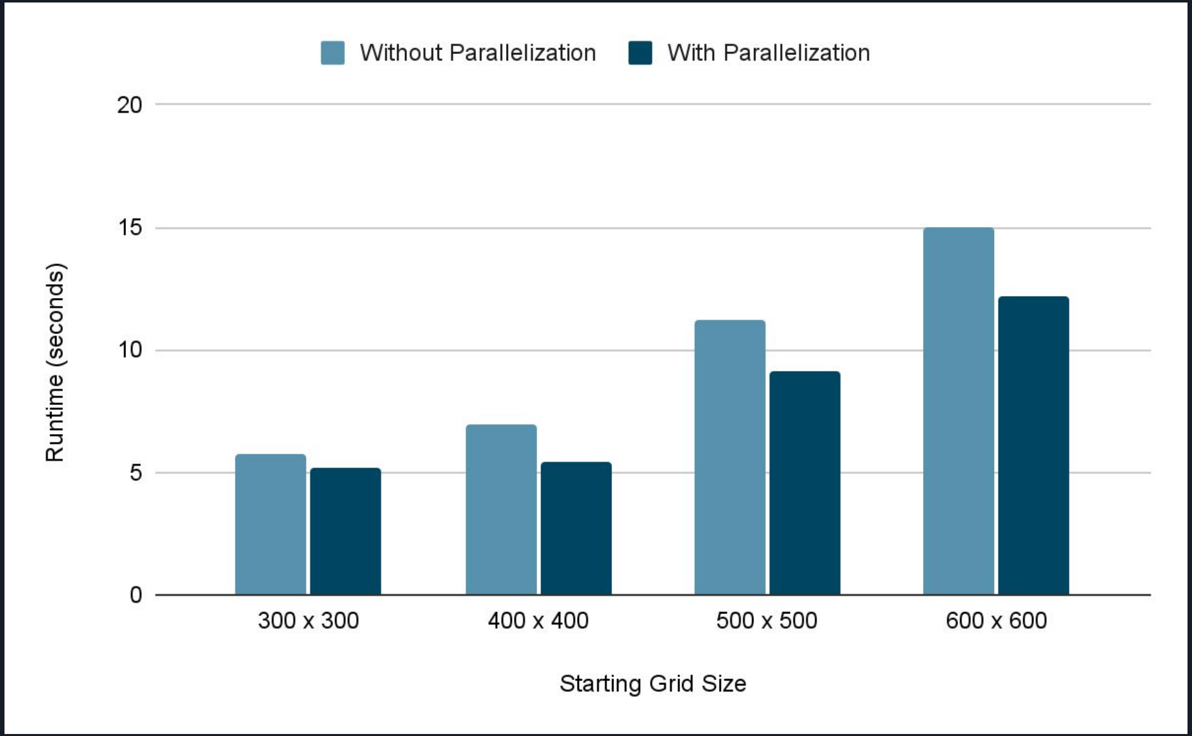
```
// calculate stable time step
double max_velocity = 0.0;
#pragma omp parallel for reduction(max:max_velocity)
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
        max_velocity = max(max_velocity, max(fabs(velocity_x[i][j]), fabs(velocity_y[i][j])));
    }
}
```

Scalability Results

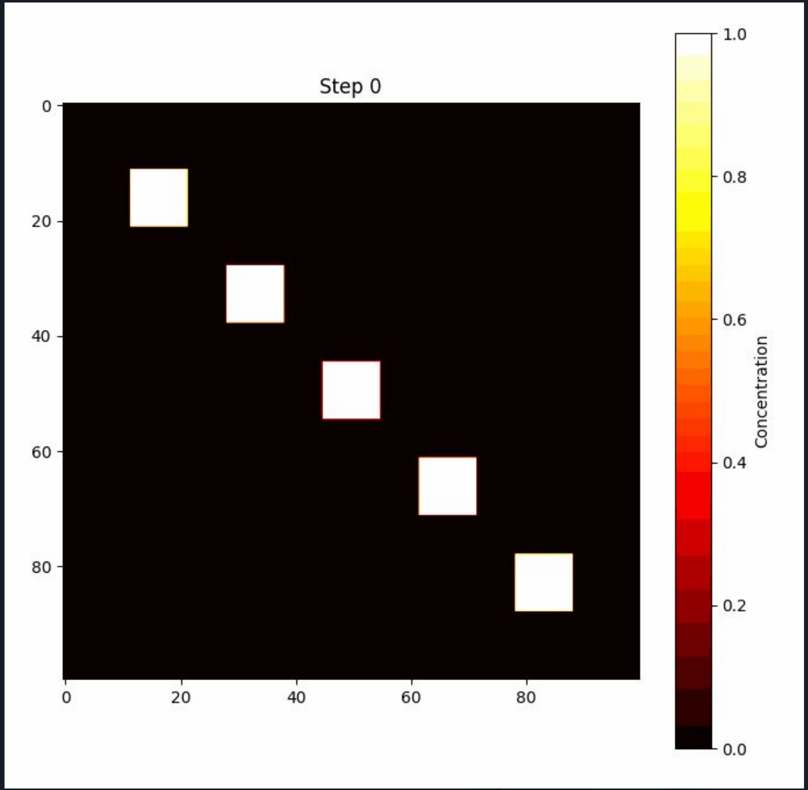
Starting Grid Size	Without Parallelization	With Parallelization (OpenMP)
300 x 300	5.75946 s	5.21159 s
400 x 400	6.92825 s	5.40180 s
500 x 500	11.2233 s	10.0942 s
600 x 600	15.0424 s	13.2068 s

Note: I used four threads

Scalability Results



Final Results



Questions?