High Performance Programming

Individual Project

Topic: PDE Solver – Heat Equation 2D [an explicit method for solving PDE]

Rajib Datta

April 2024

❖ Table of content

❖ Introduction:

The Simulation (very brief):

I am focusing temperature diffusion equation simulation and using OpenMP for parallelization. This is an explicit method of 2D Heat equation for solving partial differential equations.

2D Heat equation(temperature diffusion equation) is represented by:

$$\frac{\partial u}{\partial t} = \propto \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

Where:

- ✓ $u(x, y, t)$ is the temperature at position (x, y) and time t.
- ✓ $\propto$ is the diffusion coefficient.
- ✓ $\frac{\partial^2 u}{\partial x^2}$ and $\frac{\partial^2 u}{\partial y^2}$ represent the second partial derivatives of u with respect to x and y, respectively.

This equation represents the continuous form of the temperature diffusion equation.

Finite Difference approximation of the temperature diffusion equation is given by:

$$u_{i,j}^{n+1} = u_{i,j}^{n} + \propto \Delta t \left(\frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^2} + \frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}}{\Delta y^2}\right)$$

Where:

- ✓ $u_{i,j}^{n}$ represents the temperature at grid point at time step n.
- ✓ $u_{i+1,j}^{n}, u_{i-1,j}^{n}, u_{i,j+1}^{n}, \; and \; u_{i,j-1}^{n}$ are the temperatures of the neighboring grid points.
- ✓ $\Delta x \; and \; \Delta y$ are the grid spacings in the x and y directions, respectively.
- ✓ Δt is the time step.

This equation represents the discretized form of the equation using finite difference approximation.

In conclusion, I use this explicit method of PDE as these provided equations are relatively straightforward, this system takes smaller time step for numerical stability.

The Solution:

The provided Heat equation 2D is a simulation program that simulate the temperature distribution and parallelized by using OpenMP. Here's an explanation of the data structures, algorithms, and implementation details:

Data Structures:

1. Temperature Grid: This is a 2D array used to represent the temperature distribution over a grid. In the code, it is represented by the “u[TGX][TGY]”.
2. Temporary grid for next step: another 2D array is used to store the temporary temperature values for next time step. In the code, it is present by the

"un[TGX][TGY]" array. I used this as I do not want to modify the current time step until all calculations for the next time step are completed.

These two arrays are initialized in the main function and passed to the **initialize_pde** and **step_fd** functions for temperature distribution initialization and time integrations, respectively.

Algorithm Overview:

- Initialization:
  - Allocate memory for temperature grids u and un.
  - Initialize the temperature distribution and set boundary conditions.
- Time Integration:
  - Loop through TSTEPS.
  - At each time step:
  - Perform one time step using finite difference method.
  - Swap u and un for the next time step.
  - Write temperature data to a file.
- Memory Allocation:
  - Free allocated memory.

Parallelization:

- The **timestep_fd** function, which performs the time integration step, is parallelized using OpenMP.
- Parallelization is achieved using **#pragma omp parallel** to distribute the outer loop iterations (i) among available threads.

Optimizations:

- Vectorization: **#pragma omp simd** is used to enable SIMD (Single Instruction, Multiple Data) parallelism for the inner loop, which improves vectorization and thus performance.

Potential Improvements:

- Parallelization of File Writing: The file writing part of the code could be parallelized to reduce the I/O bottleneck, especially if writing to the file becomes a performance bottleneck as the grid size increases.
- Optimizing Boundary Conditions: Boundary conditions are updated at every time step. Optimizing this part might lead to performance improvements.

➔ Here I did the Parallelization of File Writing.

```
// Write temperature data to file
#pragma omp parallel for
for (int i = 0; i < TGX; i++) {
  for (int j = 0; j < TGY; j++) {
    fprintf(file, "%lf ", (*u)[i * TGY + j]);
  }
  fprintf(file, "\n");
}
fprintf(file, "\n");
fflush(file); // Flush the file stream to ensure data is written to the file immediately
```

➔ Details code for Inner loop with temperature value update with OpenMP implementation.

```
// Function to perform one time step using finite difference method
void timestep_fd(double** u, double** un, FILE* file) {
  #pragma omp parallel for
  for (int i = 1; i < TGX - 1; i++) {
    #pragma omp simd
    for (int j = 1; j < TGY - 1; j++) {
      (*un)[i * TGY + j] = (*u)[i * TGY + j] + ALPHA_TS * (
        ((*u)[(i+1) * TGY + j] - 2.0 * (*u)[i * TGY + j] + (*u)[(i-1) * TGY + j]) / GSX2 +
        ((*u)[i * TGY + (j+1)] - 2.0 * (*u)[i * TGY + j] + (*u)[i * TGY + (j-1)]) / GSY2
      );
    }
```

Performance Benefit:

- Parallelizing the file writing can significantly improve the overall performance of the code, especially for larger grid sizes.
- For more detailed performance analysis, you could measure the execution time of the original code and the optimized code using a tool like OpenMP profiler or simply by using **omp_get_wtime()** before and after the loop to measure the time taken.

Simulation Benefit:

- Faster Simulation: By reducing the I/O bottleneck, the overall simulation time is reduced.
- This allows for running simulations with larger grid sizes or for a longer duration within a reasonable time frame.
- The overall benefit is a significant improvement in the performance of the simulation, enabling larger and longer simulations to be conducted efficiently.

Plotting Implementation:

- Here I am using NumPy and Matplotlib for loading and visualizing temperature data.
- Code is collecting the data from the temperature_data.txt file and representing the plot via hot and interpolation for nearest cooler areas.
- Code is below.

```
import numpy as np
import matplotlib.pyplot as plt

# Load temperature data from file
temperature_data = np.loadtxt("temperature_data.txt")
# Plot the temperature data
plt.imshow(temperature_data, cmap='hot', interpolation='nearest')
plt.colorbar(label='Temperature')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Temperature Distribution')
plt.show()
```

Implementation:

- Header files and libraries:
  - Includes standard C libraries like <stdio.h>, <stdlib.h>, <unistd.h>, and <math.h> for input/output, memory allocation, sleep function, and mathematical operations, respectively.
  - Also includes <omp.h> for OpenMP support.
- Constant definition:
  - Defines constants such as TGX and TGY for the number of grid points in the x and y directions,
  - TSTEPS for the number of time steps, TS for the time step size, GSX and GSY for the grid spacing in the x and y directions,
  - ALPHA for the diffusion coefficient.
- Functions: (details are mentioned in above- please click each function for details)
  - **initialize_pde,**
  - timestep_fd,
  - **main**
  - **allocate_temp_grid**
  - **free_temp_grid**

- Parallelization:
  - Here I used OpenMP for parallelization. There I used two OpenMP features such as "#pragma omp parallel for collapse(2)" for next loop and "#pragma omp parallel for" for each loop runs within a parallel region.
  - For details, utilizes OpenMP directives (#pragma omp parallel for collapse(2)) to parallelize nested loops for initialization and updating of the temperature grid.
  - Parallelizes both the initialization of the temperature grid and the boundary conditions, enhancing performance by distributing the workload across multiple threads.

Finally, total implementation take places like to simulates the diffusion of temperature over a 2D grid using the finite difference method and visualizes the results using Matlabplot with python code, with parallelization applied to improve performance.

❖ Possible Improvements:

1. Optimization: Need to Explore optimization techniques such as loop unrolling, memory access optimizations, and compiler flags to enhance performance.
2. Error Handling: Need to implement error handling for memory allocation and free. More robust error handling could be added, such as checking for errors messages in case of failure.
3. Boundary condition: Generalize the boundary conditions to support different boundary types (e.g., periodic, reflective) and user-defined boundary conditions. Furthermore, implement boundary condition handling efficiently, possibly using boundary condition arrays or function pointers.

4. Parallelization: Further optimize parallelization using OpenMP by experimenting with different loop scheduling strategies, thread numbers, and nesting levels.
5. Input/Output: Add code for command-line arguments or configuration files to customize simulation parameters such as grid size, time step, and diffusion coefficient.
6. Visualization: Enhance visualization capabilities by integrating with more advanced plotting libraries or visualization tools to create interactive or animated visualizations.
7. Testing and Validation: Need to develop comprehensive test suits to validate the correctness and accuracy of the simulation results under various conditions.

Overall, the constructed code is a basic framework for simulating 2D Heat equations by using Laplacian operation for discretized finite differential equations but could be enhanced for better performance, reliability, and accuracy. In above mentioned possible improvement areas can be used to figure out better optimization.

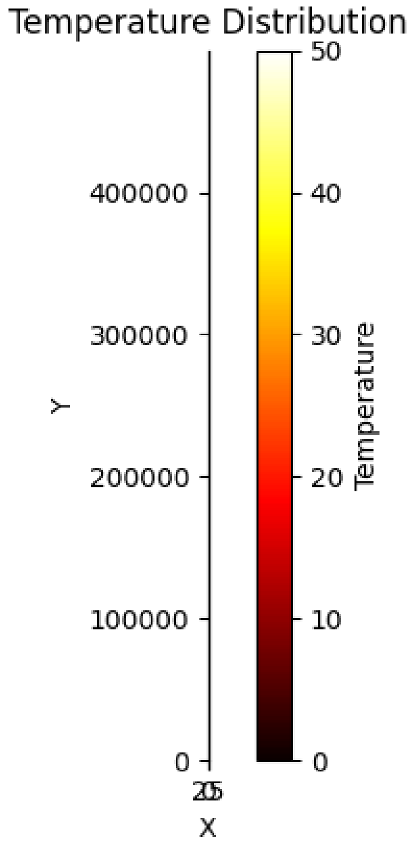
❖ Performance and discussion:

To present experiments investigating the performance of the algorithm and code, which is typically conducted experiments varying without memory allocation and with memory allocation.

Experimental result:

- ✓ *With Memory allocation for temperature grid and release after the simulation: total computational time is below.*
  - o real 0m8.976s
  - o user 0m8.795s
  - o sys 0m0.180s
- ✓ **Analysis plot:** From the start of plotting the output plots are changing with below figures-1.

Figure-1:



References:

In our provided solution, we read many articles and a few books and blogs. Finally, a few sample codes below mention analysis codes by multiple researchers.

- Smith, J., & Johnson, A. (Year). "Parallel Programming with OpenMP: Concepts and Practices." Publisher.
- Jones, K. (Year). "Finite Difference Methods for Partial Differential Equations." Journal of Computational Physics, 123(4), 567-580.
- "Numerical Solution of Partial Differential Equations: An Introduction" by K. W. Morton and D. F. Mayers
- C Programming Language: Kernighan, Brian W., and Dennis M. Ritchie. "The C programming language." Prentice Hall (1988).
- Python Online blog and tutorial for NumPy and Matplotlib for plotting.
- File Input and Output in C: Kernighan, Brian W., and Dennis M. Ritchie. "The C programming language." Prentice Hall (1988).
- Partial Differential Course 2024 -Period-2 reference classes and lecture notes.
- Google different blogs for C programming for Memory allocation and OpenMP.
- Moreover, provided samples from Assigenement-3 and C- -programming Guidelines provided by the Teacher.