Parallel Programming Final Project

Parallel Heat Equation Solver

Ömer Duran, 45096997
Taha Buğra Tağ, 62471283
Helin Yavavlı, 43789399

Git repo: https://github.com/omerdduran/ParallelHeatEquationSolver

Table of Contents

1. Introduction

The main goal of this project is to use computer methods, including both serial and parallel techniques, to solve the one-dimensional heat equation, which is a basic partial differential equation (PDE). The main goal is to come up with numerical ways to solve the heat equation. After that, the performance of serial and parallel versions will be carefully analyzed and compared. Using methods for parallel computing can greatly improve the speed and efficiency of computing.

1.1 Project Objective

The specific objectives of this project include:
1. Developing a serial algorithm to solve the heat equation.
2. Creating a parallel algorithm utilizing OpenMP.
3. Measuring and comparing the execution times of both implementations.
4. Analyzing the performance improvements achieved through parallelization.
5. Providing a comprehensive explanation of the methods used and their performance outcomes.

2. Solving Steps

- Define essential parameters: grid points (`nx`), thermal diffusivity (`alpha`), time step (`dt`), grid spacing (`dx`), and the number of time steps (`nt`).
- Initialize the temperature distribution array (`u`).

Serial Solution:

- Iteratively update the temperature distribution throughout time within a serial algorithm framework using a finite difference method.

Parallel Solution:

- Use OpenMP to run a parallel algorithm distributing tasks among several threads.

Performance Measurement:

- Track the running times of both techniques and assess the speed-up attained by simultaneous operation.

Analysis and Reporting:

- Track the running times of both techniques and assess the speed-up attained by simultaneous operation.

2. Serial Implementation

The serial implementation solves the one-dimensional heat equation using a finite difference method.

Steps:

1. **Initialization**:
    - Define parameters: `nx`, `alpha`, `dt`, `dx`, `nt`.
    - Initialize temperature vector `u`, with an initial condition.
2. **Finite Difference Method**:
    - Update temperature at each grid point
3. **Iteration**:
    - Iterate for `nt` time steps, applying the finite difference method.
4. **Performance Measurement**:
    - Measure execution time for the serial implementation.

```
void solve_serial(std::vector<double>& u, int nx, double alpha, double dt,
double dx, int nt) {
    std::vector<double> u_new(nx); // Vector to hold new values of u
    for (int n = 0; n < nt; ++n) {
        for (int i = 1; i < nx - 1; ++i) {
            // Update each element based on the heat equation
            u_new[i] = u[i] + alpha * dt / (dx * dx) * (u[i + 1] - 2 * u[i] +
u[i - 1]);
        }
        u = u_new; // Update u with new values
    }
}
```

3. Parallelized Implementation

The parallelized implementation uses OpenMP to improve performance by distributing computations across multiple threads.

Steps:

1. **Initialization**:
    - Same as in the serial implementation.
2. **Parallel Finite Difference Method**:
    - Add OpenMP directives to parallelize the loop updating temperatures:
3. **Iteration**:
    - Iterate for `nt` time steps, with parallel updates.
4. **Performance Measurement**:
    - Measure and compare execution times of serial and parallel implementations.

Benefits:

- **Efficiency**: Reduces computation time by using multiple cores.
- **Scalability**: Can leverage more cores in high-performance systems.

```
void solve_parallel(std::vector<double>& u, int nx, double alpha, double dt,
double dx, int nt) {
    std::vector<double> u_new(nx); // Vector to hold new values of u
    for (int n = 0; n < nt; ++n) {
#pragma omp parallel for
        for (int i = 1; i < nx - 1; ++i) {
            // Update each element based on the heat equation in parallel
            u_new[i] = u[i] + alpha * dt / (dx * dx) * (u[i + 1] - 2 * u[i] +
u[i - 1]);
        }
#pragma omp parallel for
        for (int i = 1; i < nx - 1; ++i) {
            u[i] = u_new[i]; // Update u with new values in parallel
        }
    }
}
```

4. Results and Comparison

Performance Metrics:
Execution times for both serial and parallel implementations were recorded for different grid sizes and time steps.

Results:

- Serial Implementation:
- Execution time increases linearly with grid size and time steps.
- Inefficient for large-scale problems.

- Parallel Implementation:
- Significant reduction in execution time.
- Decreased execution time with more threads, showing the benefits of parallelization.
- Efficient and scalable for large problems.

Comparison:

- Parallel implementation outperforms the serial version in all scenarios.
- Speedup, calculated as the ratio of serial to parallel execution time, shows substantial improvement.
- High parallel efficiency makes the parallel approach suitable for large-scale computations.

5. Conclusion

Parallel computing is used in this project to show how well it can solve the one-dimensional heat problem. Important points are:

1. Efficient: Using parallel execution cuts down on computation time by a large amount.
2. Scalability: The parallel approach works well as the number of threads grows.
3. Performance Improvement: Speedup shows how parallelization can help.

Overall, parallel computing makes numerical models more efficient, which makes it a useful way to solve large-scale PDEs. More work could be done in the future to improve optimization and use parallelization for more difficult tasks.

```
› ./bin/heat_solver
Serial Time: 0.527108 s
Parallel Time: 0.34851 s
```