Heat Equation Solver

(Parallel Programming, C++)

By:

Alexandr Cvevychalov (58738696)

Murad Gurbanli (34919073)

Mehlika Rana Akbay (51259883)

The heat equation is a partial differential equation that describes how heat is transferred through a substance. In this project, we used a 1D equation, which involves measuring heat conductivity in one-dimensional space.

Heat equation formula:

$$\frac{du}{dt} = \alpha \frac{d^2u}{dx^2}$$

First of all, we need to connect libraries for work:

#include <vector> - dynamic arrays

#include <cmath> - mathematical calculations (for example: *sin ()*)

#include <omp.h> - parallel calculations

#include <chrono> - time calculations

Constants

During solving the equation, we will use input data that will not change during the code execution. They are designated by constants.

Number π:

#define M_PI 3.14159265358979323846

Thermal conductivity coefficient was chosen as 0.01 because it is a simplified training value that gives a stable numerical scheme (no need for too small a time step):

const double alpha = 0.01;

The length of kernel (in meters):

const double L = 1.0;

The time of modelling (in seconds):

const double T = 0.1;

N – is a number of “dots in space”. These are segments where we keep and calculate the temperature. The more value is, the more accurate calculation will be (however it would take more time to compilate the project):

const int N = 1000000;

We want to know how the temperature changes over time. To do this, we need to divide time into small segments, called “time steps”:

const int steps = 20000;

Before solving the equation, itself, it is necessary to perform some more calculations using constants.

double dx = L / (N - 1); - Calculating the distance between adjacent points.

double dt = T / steps; - Time intervals for formula calculation.

Stability is a coefficient that determines the speed of heat propagation inside an object. This value is also necessary when solving the equation.

```
double stability = alpha * dt / (dx * dx);
```

void initialize() function

```
void initialize(std::vector<double>& u) {
   for (int i = 0; i < N; ++i) {
      double x = i * dx;
      u[i] = sin(M_PI * x);
   }
}
```

This function goes through all the points of the kernel (from *i = 0* to *i = N - 1*) and for each point calculates the coordinate XXX, and then sets the temperature using the formula $sin\,\pi x.$ Each value is entered into a vector for further calculations.

void heat_serial() function

The purpose of the *heat_serial* function is to model how the temperature in the rod changes over time, using the finite difference method.

```
void heat_serial(std::vector<double>& u) {
   std::vector<double> u_new(N);
   for (int t = 0; t < steps; ++t) {
      for (int i = 1; i < N - 1; ++i) {
         u_new[i] = u[i] + stability * (u[i + 1] - 2 * u[i] + u[i - 1]);
      }
      // Boundary conditions
      u_new[0] = 0.0;
      u_new[N - 1] = 0.0;
      u.swap(u_new);
   }
}
```

A new vector *u_new* is created, into which new temperature values at the next time step are written, without changing *u* immediately, so as not to spoil the data. Then we run two loops:

1. A time loop, which calculates all the temperature along the kernel.

2. A loop over the points of the kernel, within which the temperature at each point is read, based on its neighbors. After which the equation is calculated using the formula, observing the boundary conditions.

Finally, we swap the old and new values. Now *u* vector contains the new values, and *u_new* vector is ready for rewriting.

void heat_parallel() function

```
void heat_parallel(std::vector<double>& u) {
   std::vector<double> u_new(N);
   for (int t = 0; t < steps; ++t) {
#pragma omp parallel for
      for (int i = 1; i < N - 1; ++i) {
         u_new[i] = u[i] + stability * (u[i + 1] - 2 * u[i] + u[i - 1]);
      }
      u_new[0] = 0.0;
      u_new[N - 1] = 0.0;
      u.swap(u_new);
   }
}
```

This function works on a similar principle, but in it the cycle on the kernel's points is divided into several threads using OpenMP. Each thread processes its part of the rod points. All threads work simultaneously, thereby speeding up the calculations.

There is no need to parallelize the boundary conditions, because these are single values.

threads.cpp

First, we need to find out how many threads our computer allows you to use. This can be done using a separate code:

```
#include <omp.h>
#include <iostream>
int main() {
   return omp_get_num_procs();
}
```

The value that this code returns is the number of processor threads.

int main() function

To set the number of threads used when executing a parallel function, the following line is used. The more threads = the faster the compilation:

```
omp_set_num_threads(20);
```

Creating two vectors for calculations for further calculations:

```
std::vector<double> u_serial(N), u_parallel(N);
```

Fulfilling vector with data and make them equal for fair comparison:

```
initialize(u_serial);
u_parallel = u_serial;
```

Compilation time tracking of the first function:
`auto start1 = std::chrono::high_resolution_clock::now();` - starting timer

`heat_serial(u_serial);` - starting function

`auto end1 = std::chrono::high_resolution_clock::now();` - ending timer

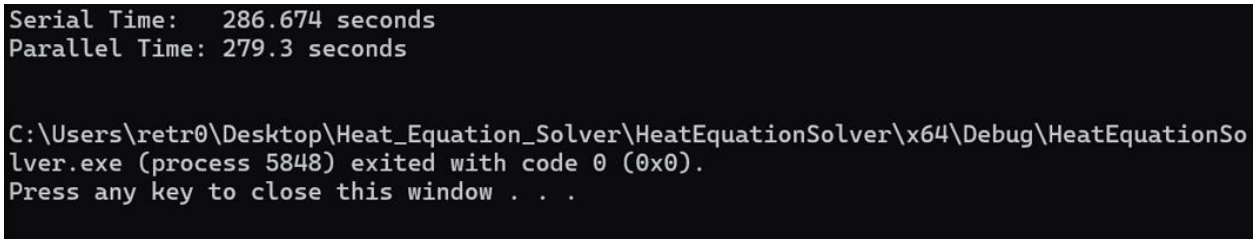
Compilation time tracking of the second function:
```
auto start2 = std::chrono::high_resolution_clock::now();
heat_parallel(u_parallel);
auto end2 = std::chrono::high_resolution_clock::now();
```

Calculations of the time it took for a function to run.
```
std::chrono::duration<double> time_serial = end1 - start1;
std::chrono::duration<double> time_parallel = end2 - start2;
```

Printing the time it took to run the code:
```
std::cout << "Serial Time:   " << time_serial.count() << " seconds\n";
std::cout << "Parallel Time: " << time_parallel.count() << " seconds\n";
```

Conclusion

As part of this project, we developed a parallel solver for the one-dimensional heat equation using the finite difference method and OpenMP technology. We began by implementing a sequential version of the algorithm that models the propagation of temperature along a rod over time. Then, we identified independent computations, which allowed us to effectively parallelize the main loop using the *#pragma omp parallel* for directive. The parallel implementation correctly handles boundary conditions and produces results that are nearly identical to the sequential version. We conducted runtime measurements of both versions and analyzed their accuracy. For larger problem sizes, the parallel version demonstrated a clear performance advantage. During the project, we gained practical experience in solving partial differential equations, applying numerical methods, and optimizing multithreaded computations. The project can serve as a foundation for more complex models, including two- and three-dimensional simulations. The skills and knowledge acquired are applicable in scientific and engineering tasks that require high-performance computing.



Example of Code Using, N=1000000, steps=20000