Project #3: Heat Diffusion Simulation

MPI Simulation

To implement the MPI simulation, we have developed three core components: **main.cpp** (the entry point), **Heat.hpp** (the class interface), and **Heat.cpp** (the implementation). Let's examine each component in detail.

main.cpp

The main function serves as the MPI program coordinator. Here we initialize the MPI environment and launch the heat diffusion solver with specific physical parameters.

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Comm_size(MPI_COMM_WORLD, &p);

// Create an instance of the Heat class with parametrs alpha, dt, dx, L, threshold, maxSteps
Heat heat(0.01, 0.001, 0.01, 0.08, 10e-6, 300, p, id);
heat.solve();

MPI_Finalize();
```

Heat.hpp

With the Heat class, we achieve complete encapsulation of the parallel heat diffusion solver. This class manages domain decomposition, inter-processor communication, and numerical computation in a coordinated fashion. To accomplish this, we use the following class design.

```
Heat(double alpha, double dt, double dx, double L, double threshold, int maxSteps, int p, int id)
    : alpha(alpha), dt(dt), dx(dx), L(L), threshold(threshold), maxSteps(maxSteps),
      n(static_cast<int>(L / dx)),
      p(p),
      id(id),
      px(floor(sqrt(p))),
      py(p / px),
      nx(floor(n / px)),
      ny(floor(n / py)),
      rx(n % px),
      ry(n % py),
      grid((nx + 2) * (ny + 2), -1.0),
      newGrid((nx + 2) * (ny + 2), -1.0)
      {}

void initializeGhostValues();
void initialCondition(double cornerTemp, int x, int y);
void printGrid();
int get(int i, int j);
void writeCSV(int timeStep, int x, int y);
void writeVTK(const std::string &filename);
void writeVTKParallel(int timeStep, int x, int y);
void applyBoundaryConditions(double cornerTemp, int x, int y);
void solve();
```

The constructor calculates an optimal 2D processor grid layout using px = floor(sqrt(p)) to minimize communication overhead by creating a nearly square processor arrangement.

The initializer list computes all derived parameters and allocates memory for temperature grids including ghost cells for boundary communication.

Variable Explanations:

- alpha, dt, dx, L: Physical and discretization parameters
- n: Total grid points per side (calculated as L/dx)

- px, py: Processor grid dimensions (px × py = p processors)
- nx, ny: Local grid size per processor
- rx, ry: Remainder points for load balancing when grid doesn't divide evenly
- grid, newGrid: Double-buffered temperature arrays including ghost cells

<u>Heat.cpp</u>

The Heat.cpp implementation represents the most critical component of our parallel simulation system. This file contains all the algorithms, communication protocols, and numerical methods that enable the parallel computation of the heat diffusion equation.

To efficiently access our 2D temperature data stored in 1D vectors. This function converts 2D coordinates (i,j) to linear array indices using row-major ordering. The (nx + 2) term accounts for ghost cells, ensuring that our logical 2D grid.

For proper boundary communication between processors, we initialize ghost cells around each local subdomain. The ghost cells create a buffer zone that will be filled with data from neighboring processors. val = 0: Initial ghost cell value (will be overwritten by neighbor communication). 0 and nx+1, ny+1 are the boundary indices, representing the ghost cell layers.

For establishing realistic physical conditions, we implement a initialization system that sets up the thermal boundary conditions across the distributed domain, while maintaining zero temperature elsewhere.

One of the most sophisticated aspects of our implementation is the parallel CSV writing system. This method enables all processors to write simultaneously to a single file without conflicts. Processor 0 handles file cleanup while others wait at the barrier. Each processor calculates its unique write positions mathematically:

- offsetH: Space reserved for header
- id * (nx * ny * line_size): Block space for this processor

MPI_File_write_at ensures no conflicts between processors.

To enable scientific visualization and validation of our simulation results, we implement sophisticated VTK (Visualization Toolkit) output capabilities. VTK is the industry standard format for scientific visualization, compatible with tools like ParaView and VisIt.

For debugging and small-scale simulations, we provide a basic VTK writer that outputs the local domain of each processor. For large-scale production runs, we use an advanced parallel VTK writer that gathers data from all processors into a single, unified visualization file.

The parallel VTK writer: Processor 0 writes a global VTK header with full domain dimensions, Each processor packs its interior points (no ghost cells) into a local array. MPI_Gatherv collects variable-sized data from all processors. Finally, processor 0 rebuilds the full domain in correct spatial order and writes it to the file.

The solve() method orchestrates the entire parallel simulation process:

The first part is topology mapping, each processor determines its 2D position and identifies its neighbors using arithmetic operations on processor ranks, -1 values indicate domain boundaries (no neighbor). We have included a resizer if the grid is not divisible by the number of processors.

MPI_Type_vector creates a non-contiguous datatype for column transfers. We have implemented deadlock avoidance by carefully ordering Send/Recv operations, which prevents circular dependencies.

Rows are contiguous in memory (use MPI_DOUBLE directly), columns are strided (use derived datatype)

Then, we apply the numerical algorithm lies in the finite difference computation that updates temperature values according to the heat equation.

```
double maxChange = 0.0;
for (int i = 1; i < nx + 2; ++i)
{
    for (int j = 1; j < ny + 2; ++j)
    {
        newGrid[get(i,j)] = grid[get(i,j)] + alpha_dt_dx2 * (grid[get(i - 1,j)] + grid[get(i + 1,j)] + grid[get(i,j - 1)] + grid[get(i,j + 1)] - 4 * grid[get(i,j)]);

        maxChange = max(maxChange, abs(newGrid[get(i,j)] - grid[get(i,j)]));
    }
```

To check the convergence, we use MPI_Allreduce with MAX operation finds the largest change across all processors, all processors participate in the converge decision.

Spark Data Analysis

For the Spark analysis component, we have developed a Java application that processes the temperature data generated by the MPI simulation.

The first part, create the Spark Session we configure Spark with advanced optimization features. We read all timestep files into a unified dataset, using regex to extract timestep numbers from filenames, converting them to proper timestamps.

Then we use the data ingestion to process raw CSV files from the MPI simulation into a structured DataFrame suitable for temporal analysis.

Query 1: Consecutive Temperature Difference Analysis

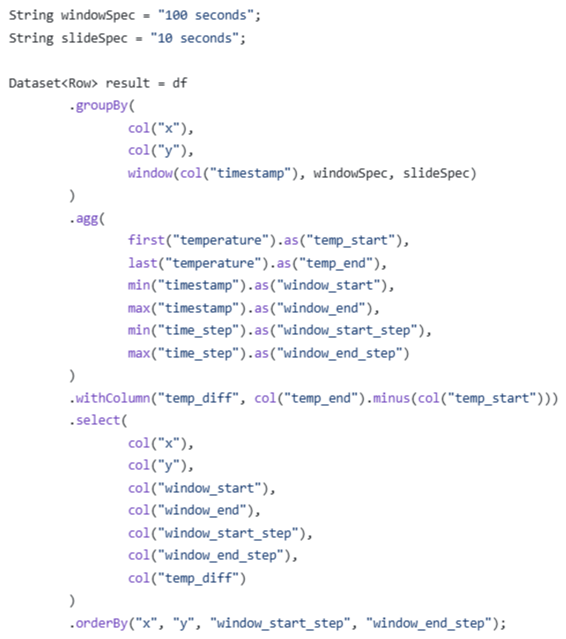
This query creates separate time-ordered windows for each spatial location, then uses the lag function to access the previous timestep's temperature.

```
WindowSpec windowSpec = Window.partitionBy("x", "y").orderBy("time_step");

Dataset<Row> result = df
                .withColumn("prev_temperature", lag("temperature", 1).over(windowSpec))
                .withColumn("temp_diff", col("temperature").minus(col("prev_temperature")))
                .filter(col("prev_temperature").isNotNull())
                .groupBy("x", "y")
                .agg(
                                min("temp_diff").as("min_temp_diff"),
                                max("temp_diff").as("max_temp_diff"),
                                avg("temp_diff").as("avg_temp_diff"))
                .orderBy("x", "y");

result.show(10);

writeToCSV(result, "../results/", "query1_results.csv");
```

wirteToCSV : This method exports a Spark DataFrame to a CSV file with headers, automatic partitioning, and overwrite enabled.

Query 2: Sliding Window Temperature Analysis

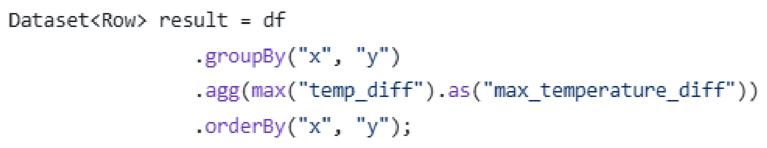
We implement overlapping time windows to capture temperature changes over longer periods.



```
String windowSpec = "100 seconds";
String slideSpec = "10 seconds";

Dataset<Row> result = df
        .groupBy(
                col("x"),
                col("y"),
                window(col("timestamp"), windowSpec, slideSpec)
        )
        .agg(
                first("temperature").as("temp_start"),
                last("temperature").as("temp_end"),
                min("timestamp").as("window_start"),
                max("timestamp").as("window_end"),
                min("time_step").as("window_start_step"),
                max("time_step").as("window_end_step")
        )
        .withColumn("temp_diff", col("temp_end").minus(col("temp_start")))
        .select(
                col("x"),
                col("y"),
                col("window_start"),
                col("window_end"),
                col("window_start_step"),
                col("window_end_step"),
                col("temp_diff")
        )
        .orderBy("x", "y", "window_start_step", "window_end_step");
```

Query 3: Maximum Temperature Variance Detection

To identify spatial locations with the highest thermal activity, we find the maximum temperature difference across all time windows.



```
Dataset<Row> result = df
                .groupBy("x", "y")
                .agg(max("temp_diff").as("max_temperature_diff"))
                .orderBy("x", "y");
```