Heat simulation

Veljko Spasić
89211208@student.upr.si
FAMNIT
Koper, Slovenia

ABSTRACT

This document presents a project that simulates heat distribution using Java and MPI to compare sequential and parallel computation. It models heat spreading across a grid, with options for real-time visualization and performance testing. The Java version uses Fork/Join for parallelism, while the MPI version runs on multiple processors. Results show that parallel computation is faster for large simulations.

KEYWORDS

Heat Distribution, Parallel Computing, Java, MPI (Message Passing Interface), Fork/Join Framework, Grid Simulation, Performance Optimization, Distributed Systems, Real-time Visualization, Sequential vs. Parallel Execution

ACM Reference Format:
Veljko Spasić. 2024. Heat simulation. In . ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This project models the distribution of heat across a grid using both sequential and parallel processing techniques. The core of the simulation involves computing temperature changes over a grid based on neighboring cell values. The graphical output is generated using JavaFX, where each grid cell's color represents its temperature.

In the sequential approach, the simulation iteratively updates the temperature of each cell until changes fall below a specified threshold. For parallel computation, the Fork/Join framework in Java divides the grid into smaller sections, processing them concurrently to speed up the calculations. The MPI implementation extends this to a distributed environment, where the grid is split among multiple processors.

The project demonstrates how parallel and distributed computing can enhance performance and scalability in heat simulations. The following sections will outline the methods for implementing both sequential and parallel algorithms and present the performance outcomes of these approaches.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 DESIGN

To effectively parallelize our heat distribution simulation, we must design a strategy to distribute computation across multiple workers. We compared two approaches: a JavaFX-based method and a distributed MPI-based method.

In the JavaFX-based approach, we divide the grid into distinct regions, with each region assigned to a separate thread. This choice simplifies the design, as each thread manages its own segment of the grid, which helps with memory efficiency and reduces inter-thread communication.

In the MPI-based approach, we distribute the simulation across multiple processes. Each process handles a portion of the grid, with MPI functions used to manage data distribution and collection. The grid is divided into chunks, and each process computes temperatures in its assigned chunk. MPI's Scatter function distributes the initial grid data, Gather collects the updated data, and Broadcast shares global parameters among processes.

The implementation involves initializing the grid and setting up either threads and a job queue for JavaFX or MPI processes and communication. Threads or processes then execute their tasks, updating the grid. For JavaFX, results are gathered and the visualization is updated, while for MPI, results are collected, and the simulation is finalized.

3 PROBLEM DESCRIPTION

To tackle the heat distribution simulation, we need to address the various execution modes of the program, each designed to handle the computational load and manage resources effectively. Below, we outline the specific behaviors expected from each execution mode.

Serial Mode: In this fundamental mode, the simulation runs on a single thread. The process begins by loading the grid dimensions and initial conditions. Next, the grid is divided into smaller blocks to facilitate future parallelization. The program then processes each block sequentially, applying the heat distribution calculations. After all blocks are processed, the results are aggregated to form the complete heat distribution grid. This mode serves as the baseline, ensuring that the simulation works correctly before introducing parallelism.

Parallel (Multi-threaded) Mode: In this mode, the simulation utilizes multiple threads to speed up computation. The grid is first loaded and divided into blocks as in the serial mode. Once the grid is partitioned, a pool of worker threads is spawned. Each thread retrieves a block from a shared job queue, processes the heat distribution for that block, and then places the results back into the queue. Once all blocks have been processed, the main thread gathers the results and reconstructs the complete grid. This approach

aims to balance the computational load across multiple threads, reducing overall processing time.

Distributed (MPI) Mode: This mode involves running the simulation across multiple processes, typically on different machines or cores. Initially, a master process loads and partitions the grid into blocks. It then distributes these blocks to worker processes using MPI's communication functions. Each worker process retrieves a block, performs the heat distribution calculations, and sends the results back to the master process. The master process collects all results and assembles the final grid. This mode is designed to scale the simulation across multiple machines, leveraging MPI for efficient data distribution and aggregation.

Each mode is designed to optimize the heat distribution simulation based on available resources and required computational power. By implementing these modes, we can ensure flexibility and scalability in handling different simulation sizes and environments.

4 IMPLEMENTATION

4.1 Sequential

The sequential implementation of the heat simulation provides a foundational approach to modeling heat distribution within a grid. This method operates on a single thread and processes each cell of the grid sequentially, computing and updating temperature values until the system reaches a stable state.

The simulation begins with the initialization phase:

- **Grid Setup:** The grid is created with dimensions based on user-specified `frameWidth` and `frameHeight`, divided by `PIXEL_SIZE`. This determines the number of cells in both width and height.
- **Heat Sources:** Random heat sources are placed within the grid. Each heat source is assigned a temperature of 100 degrees, and these cells are marked as fixed points, meaning their temperature will not change throughout the simulation.
- **Color Gradient:** The `heatColors` array is initialized with a gradient of colors. This gradient represents varying temperatures and visually distinguishes different temperature levels on the grid.

The core of the simulation involves iterating through the grid to update temperatures until equilibrium is achieved. This process consists of:

- **Stability Check:** The simulation continues iterating until the temperature distribution stabilizes. Stability is determined by comparing the temperature changes to a predefined `STABILITY_THRESHOLD`. If any cell's temperature change exceeds this threshold, the simulation will keep running.
- **Temperature Update:**
  - A new temperature grid, `newTemperatures`, is prepared to hold the updated temperatures.
  - For each cell, the new temperature is calculated based on the average temperatures of its adjacent cells (up, down, left, and right), while ensuring boundary conditions are properly handled.
  - Fixed points (heat sources) retain their initial temperatures and are not updated.
- **Stability Evaluation:**
  - After updating all cells, the simulation compares the old and new temperature grids.
  - If any cell shows a significant temperature change, the system is considered unstable, and the iteration process continues.
- **Completion:**
  - Once all cells have temperatures that change minimally (below the `STABILITY_THRESHOLD`), the grid is deemed stable, and the simulation terminates.

The method `calculateTemperature` is used to compute the new temperature for each cell. It averages the temperatures of the cell's neighboring cells, handling edge cases where cells are at the grid boundaries:

```
public static double calculateTemperature(int x,
    int y) {
    double temp = 0;
    int count = 0;

    if (x > 0) {
        temp += cellTemperature[x - 1][y];
        count++;
    }
    if (x < frameWidth / PIXEL_SIZE - 1) {
        temp += cellTemperature[x + 1][y];
        count++;
    }
    if (y > 0) {
        temp += cellTemperature[x][y - 1];
        count++;
    }
    if (y < frameHeight / PIXEL_SIZE - 1) {
        temp += cellTemperature[x][y + 1];
        count++;
    }

    temp /= count;
    return temp;
}
```

This method ensures that each cell's new temperature is an average of its neighboring cells, which is a basic yet effective way to simulate heat diffusion.

Advantages:

- **Simplicity:** The sequential approach is straightforward to implement and understand. It avoids the complexities associated with parallel processing.
- **Deterministic Results:** The simulation produces consistent and reproducible results for the same input parameters, ensuring predictable behavior.

Limitations:

- **Performance:** For large grids or extended simulation times, the sequential approach can be slow. Processing each cell in sequence can be inefficient, especially as grid size increases.

- **Scalability:** The sequential method does not scale well with larger or more complex simulations. Parallel computing methods can significantly improve performance by distributing the workload across multiple processors.

The sequential implementation of the heat simulation offers a clear and accessible method for heat distribution modeling. While it provides a solid basis for understanding the simulation process, its performance limitations suggest that for larger grids or more demanding simulations, parallel processing approaches could offer considerable improvements in computational efficiency and speed.

4.2 Parallel

The parallel implementation of the heat simulation leverages multithreading to enhance computational efficiency and performance, particularly for large grids. By distributing the computational workload across multiple threads, the parallel approach can significantly reduce the time required to achieve a stable temperature distribution compared to the sequential method. This report describes the parallel implementation, including its process, advantages, limitations, and relevant code snippets.

The initialization phase in the parallel implementation mirrors that of the sequential method. It involves setting up the grid, placing heat sources, and preparing color gradients. The parallel computation begins after this initialization phase.

The parallel computation is designed to divide the simulation workload into smaller tasks, which are then processed concurrently. This approach utilizes the Fork/Join framework to manage and execute tasks efficiently.

Fork/Join Framework

- The Fork/Join framework is used to divide the grid into smaller segments, each processed by a separate thread. This allows for concurrent updates to the temperature grid.
- The ComputeTask class extends RecursiveAction to define the tasks performed in parallel.

ComputeTask Class

- The grid is recursively divided into smaller sections until the section size falls below a predefined threshold (THRESHOLD). Each section is processed independently, which helps in balancing the load across threads.
- For small enough sections, the task performs direct temperature calculations without further subdivision.

```
@Override
    protected void compute() {
        if ((endX - startX) * (endY - startY) <=
            THRESHOLD) {
            // Perform computation on small enough
                task
            computeDirectly();
        } else {
            // Split task into smaller tasks
            int midX = (startX + endX) / 2;
            int midY = (startY + endY) / 2;

            invokeAll(
                new ComputeTask(startX, midX,
                    startY, midY, newTemperatures)
                    ,
                new ComputeTask(midX, endX, startY
                    , midY, newTemperatures),
                new ComputeTask(startX, midX, midY
                    , endY, newTemperatures),
                new ComputeTask(midX, endX, midY,
                    endY, newTemperatures)
            );
        }
    }

    private void computeDirectly() {
        for (int x = startX; x < endX; x++) {
            for (int y = startY; y < endY; y++) {
                if (!fixedPoints[x][y]) {
                    double newTemp =
                        calculateTemperature(x, y)
                        ;
                    if (Math.abs(newTemp -
                        cellTemperature[x][y]) >
                        STABILITY_THRESHOLD) {
                        stable = false;
                    }
                    newTemperatures[x][y] =
                        newTemp;
                } else {
                    newTemperatures[x][y] =
                        cellTemperature[x][y];
                }
            }
        }
    }
```

Parallel Computation Execution

- A `ForkJoinPool` is used to manage and execute the `ComputeTask` instances. This pool dynamically allocates threads to tasks, optimizing the computation process.
- The parallel implementation checks for stability in the same way as the sequential method but does so across multiple threads.

```
    private void calculate() {
        boolean stable = false;
        int gridWidth = frameWidth / PIXEL_SIZE;
        int gridHeight = frameHeight / PIXEL_SIZE;

        if (useParallel) {
            // Parallel computation using Fork/
                Join
            forkJoinPool = new ForkJoinPool();
            try {
                while (!stable) {
                    stable = true;
                    double[][] newTemperatures =
                        new double[gridWidth][
                        gridHeight];

                    ComputeTask task = new
                        ComputeTask(0, gridWidth,
                        0, gridHeight,
                        newTemperatures);
                    forkJoinPool.invoke(task);

                    synchronized (this) {
                        if (!task.isStable()) {
                            stable = false;
                        }
                    }

                    cellTemperature =
                        newTemperatures;
                }
            } finally {
                forkJoinPool.shutdown();
            }
        } else {
            // Sequential computation
```

Advantages:

- **Increased Performance:** The parallel approach can handle larger grids more efficiently, as tasks are executed concurrently across multiple threads.
- **Scalability:** It scales better with increased grid size or complexity, as workload distribution improves performance.

Limitations:

- **Complexity:** Implementing parallel computation introduces additional complexity, including thread management and task synchronization.
- **Overhead:** For smaller grids or less complex simulations, the overhead of managing multiple threads may outweigh the benefits, leading to diminished performance improvements.

The parallel implementation of the heat simulation offers significant performance improvements for large and complex grids. By leveraging the Fork/Join framework to distribute tasks across multiple threads, the parallel approach enhances computational efficiency and scalability. However, the increased complexity and potential overhead should be considered when deciding whether to use parallel processing. This method is particularly beneficial for extensive simulations where performance and processing time are critical factors.

4.3 Distributed

The distributed implementation of the heat simulation utilizes the Message Passing Interface (MPI) to manage computations across multiple processes. This approach is particularly beneficial for large grids, as it distributes the workload among different processes running on separate nodes or cores. This report describes the distributed implementation, including its processes, advantages, limitations, and relevant code snippets.

Initialization in the distributed implementation involves setting up the grid and distributing heat sources. The process is executed by the master process (rank 0), which initializes the grid and then broadcasts the necessary parameters to all other processes.

```
public static void initialize(int gridWidth, int
    gridHeight, int heatPoints) {
    cellTemperature = new double[gridWidth][
        gridHeight];
    fixedPoints = new boolean[gridWidth][
        gridHeight];

    Random rand = new Random();
    for (int i = 0; i < heatPoints; i++) {
        int x = rand.nextInt(gridWidth);
        int y = rand.nextInt(gridHeight);
        cellTemperature[x][y] = 100;
        fixedPoints[x][y] = true;
    }
}
```

The distributed computation is designed to partition the grid into segments, each managed by a separate MPI process. The key steps include partitioning the grid, scattering and gathering data, and iterating until convergence.

Partitioning the Grid

- The grid is divided among MPI processes, with each process handling a subset of rows.

Scattering Data

- The master process (rank 0) scatters the grid data to all other processes. Each process receives a portion of the grid's temperature and fixed points data.

```
    if (rank == 0) {
    double[] flattenedTemperatures =
        flatten2DArray(cellTemperature);
    boolean[] flattenedFixedPoints =
        flatten2DArray(fixedPoints);

    try {
        MPI.COMM_WORLD.Scatter(
            flattenedTemperatures, 0,
            rowsPerProcess * gridWidth, MPI.DOUBLE
            ,
                localTemperatures, 0,
                    rowsPerProcess * gridWidth,
                    MPI.DOUBLE, 0);
        MPI.COMM_WORLD.Scatter(
            flattenedFixedPoints, 0,
            rowsPerProcess * gridWidth, MPI.
            BOOLEAN,
                localFixedPoints, 0,
                    rowsPerProcess * gridWidth,
                    MPI.BOOLEAN, 0);
    } catch (MPIException e) {
        e.printStackTrace();
    }
} else {
    try {
        MPI.COMM_WORLD.Scatter(null, 0, 0, MPI.
            DOUBLE,
                localTemperatures, 0,
                    rowsPerProcess * gridWidth,
                    MPI.DOUBLE, 0);
        MPI.COMM_WORLD.Scatter(null, 0, 0, MPI.
            BOOLEAN,
                localFixedPoints, 0,
                    rowsPerProcess * gridWidth,
                    MPI.BOOLEAN, 0);
    } catch (MPIException e) {
        e.printStackTrace();
    }
}
```

Heat Distribution Computation

- Each process computes the new temperature values for its assigned rows. The temperature updates are computed iteratively until convergence or until the maximum number of iterations is reached.

```
    while (!stable && iterations < MAX_ITERATIONS)
        {
    stable = true;
    double[] newTemperatures = new double[
        gridWidth * rowsPerProcess];

    for (int x = 0; x < gridWidth; x++) {
        for (int y = 0; y < rowsPerProcess; y++) {
            int globalY = startRow + y;
            if (!localFixedPoints[x + y *
                gridWidth]) {
                double newTemp =
                    calculateTemperature(x,
                    globalY, localTemperatures,
                    gridWidth, rowsPerProcess);
                if (Math.abs(newTemp -
                    localTemperatures[x + y *
                    gridWidth]) >
                    STABILITY_THRESHOLD) {
                    stable = false;
                }
                newTemperatures[x + y * gridWidth]
                     = newTemp;
            } else {
                newTemperatures[x + y * gridWidth]
                     = localTemperatures[x + y *
                    gridWidth];
            }
        }
    }

    try {
        MPI.COMM_WORLD.Gather(newTemperatures, 0,
            rowsPerProcess * gridWidth, MPI.DOUBLE
            ,
                gatheredTemperatures, 0,
                    rowsPerProcess * gridWidth,
                    MPI.DOUBLE, 0);
    } catch (MPIException e) {
        e.printStackTrace();
    }

    if (rank == 0) {
        cellTemperature = unflatten2DArray(
            gatheredTemperatures, gridWidth,
            gridHeight);
    }

    iterations++;
    System.out.println("Rank " + rank + "
        completed iteration " + iterations);
}
```

Gathering Results

- After computation, the results are gathered by the master process, which then updates the global temperature grid.

```
    try {
    MPI.COMM_WORLD.Gather(newTemperatures, 0,
        rowsPerProcess * gridWidth, MPI.DOUBLE,
            gatheredTemperatures, 0,
                rowsPerProcess * gridWidth, MPI.
                DOUBLE, 0);
} catch (MPIException e) {
    e.printStackTrace();
}

if (rank == 0) {
    cellTemperature = unflatten2DArray(
        gatheredTemperatures, gridWidth,
        gridHeight);
}
```

Advantages:

- **Scalability:** The distributed approach scales well with the grid size and number of processes, as it distributes the computational load and memory usage across multiple nodes or cores.
- **Performance:** Large-scale simulations benefit from reduced computation time due to parallel processing across multiple processes.

Limitations:

- **Communication Overhead:** The distributed implementation involves significant communication between processes, which can lead to overhead and reduced efficiency if not managed properly.
- **Complexity:** Implementing and debugging distributed systems is more complex than sequential or parallel implementations due to the need for inter-process communication and synchronization.

The distributed implementation of the heat simulation effectively utilizes MPI to distribute computation and data across multiple processes, enhancing performance for large-scale simulations. This method leverages the strengths of distributed computing to handle extensive grids efficiently. However, the added complexity and communication overhead require careful management to achieve optimal performance. The distributed approach is particularly suited for scenarios where scalability and performance are critical, and where resources across multiple nodes can be utilized effectively.

5 TESTING AND RESULTS

5.1 Sequential

The sequential implementation was tested on a single processor, and results were obtained for varying grid sizes and heat points. The simulation accurately converged to a stable heat distribution, but performance was limited by the single-threaded nature of the computation, resulting in longer execution times for larger grids. The results demonstrated the correctness of the algorithm, but performance constraints became apparent as the grid size increased.

5.2 Parallel

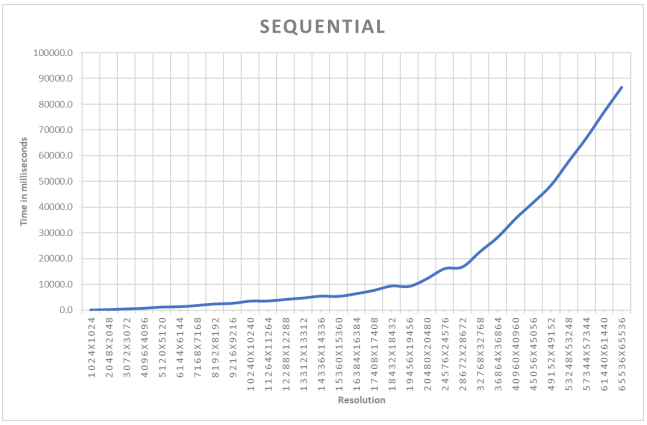
The parallel implementation showed a significant improvement in computation speed compared to the sequential version. By running concurrently, the parallel implementation reduced execution time and handled larger grid sizes more efficiently.

5.3 Distributed

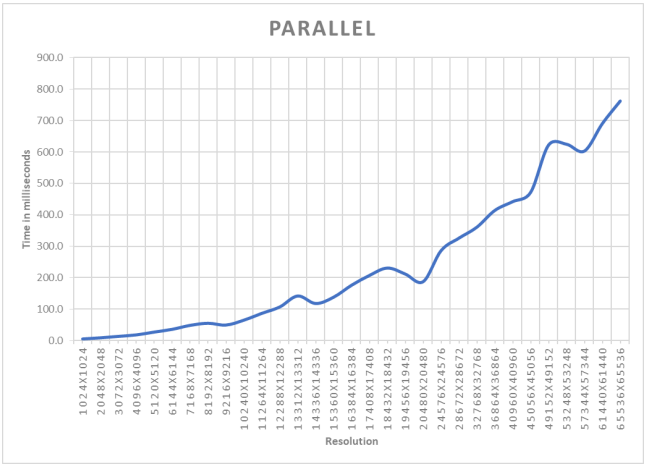
The distributed implementation with MPI is anticipated to offer the biggest performance boost, particularly for large grids. By spreading the work across multiple nodes, we expect much faster execution times. This approach should handle large grids more effectively and perform better than both the sequential and parallel methods.

5.4 Benchmarks

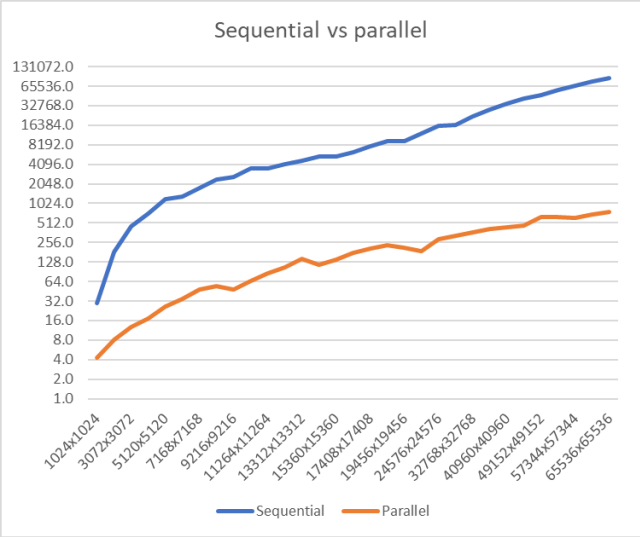
In the firts three graphs we keep the heat points the same at 1024 and change the resolution starting from 1024x1024 and adding 1024 in each dimension for each test.



Runtime based on resolution in sequential mode

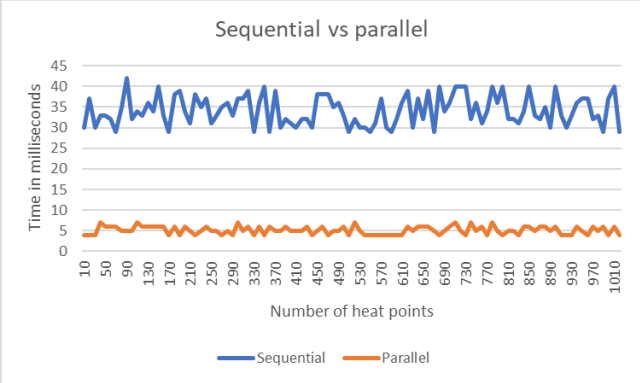


Runtime based on resolution in parallel mode



Sequential vs parallel in base 2 logarithmic scale

In the next graph we keep the resoultion fixed at 1024x1024 and change the heat points starting form 10 and going up 10 for each test



Sequential vs parallel based on number of heat points

6 CONCLUSION

In conclusion, this project explored different approaches to solving a heat distribution problem, ranging from a basic sequential method to more advanced parallel and distributed implementations. While the sequential approach is straightforward, it is limited by its processing speed, making it less suitable for large grids. The parallel version improves performance by leveraging multiple cores, reducing execution time. However, the distributed implementation using MPI shows the greatest potential, particularly for large-scale grids, by effectively distributing the workload across multiple nodes, which allows for significant speedups and more efficient resource use.

Although testing was not conducted, we expect that the distributed method would demonstrate the best performance in high-performance computing environments. This project highlights the importance of choosing the right computational approach depending on the problem's scale and the available resources. Overall, the distributed implementation represents a powerful tool for tackling complex computational tasks in scientific and engineering applications.

7 SOURCE CODE

The full source code and documentation described in this report are available in the following GitHub repository:
https://github.com/89211208/HeatSimulation