Heat Simulation

Miha Štih
8922134
UP Famnit
Programiranje 3 - Vzporedno Programiranje

June 2025

Abstract

This report presents the implementation and analysis of a heat simulation using sequential, parallel, and distributed approaches, with the distributed approach using MPJ-Express[2]. This report compares the different approaches and their efficiency by running each simulation multiple times and comparing the results by the time it took to simulate them. The instructions for implementing the problem are from [1].

1 Introduction

The heat simulation problem is a classical problem in computational physics, where the goal is to simulate the distribution of heat in a given domain over time.

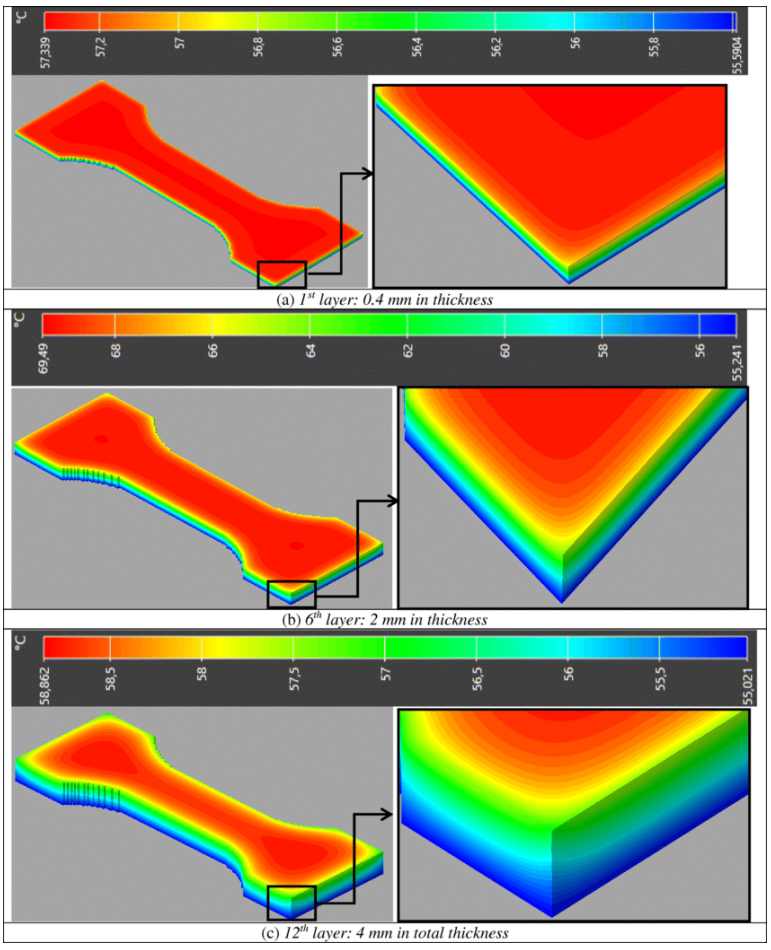

Figure 1: Example of a 3D polymer heat distribution. From [3]

In this report, we explore three different approaches to solving the heat simulation problem: sequential, parallel, and distributed. For the simplicity of the project, we use a 2D grid, the size of which can be defined manualy. The sequential approach involves solving the iteration of the heat equation by iteration on a single core, which is the simplest implementation of the three but computationally expensive for large problems.

To address computational challenges, parallel and distributed computing techniques are used. The parallel implementation leverages the power of multicore processors to divide the computational workload across multiple cores within a single machine. This method reduces computation time by performing temperature updates concurrently for different chunks of the original grid.

The distributed implementation extends this concept by using multiple computers connected over a network. Each machine, or node, is responsible for a portion of calculation, and communication between nodes is needed to ensure that everything is running properly. This approach is particularly useful for very large simulations that would be inefficient to run on a single machine.

Comparing the performance of these three implementations will give us insight into the differences between computational speed and complexity between implementations.

2 Temperature Distribution

This equation represents the approximation used to update the temperature at a specific point on the grid $G_{i,j}$. The temperature at each point on the grid is calculated as the average of the temperatures of its four neighboring points on the grid: the points on the grid directly above, below, to the left and to the right. This averaging process models the heat diffusion, where heat naturally spreads from hotter to cooler regions, eventually reaching a stable state.

$$G_{i,j} = \frac{G_{i,j+1} + G_{i+1,j} + G_{i,j-1} + G_{i-1,j}}{4} \tag{1}$$

3 Implementation

As already mentioned, 2D grid is used to simplify the implementation of this problem. However, I have decided to use a more complicated equation that includes all eight surrounding grid points, as this approach more accurately reflects the real heat distribution on a thin plate of a material. The equation used is the following:

$$G_{i,j} = \frac{G_{i-1,j-1} + G_{i,j-1} + G_{i+1,j-1} + G_{i-1,j} + G_{i+1,j} + G_{i-1,j+1} + G_{i,j+1} + G_{i+1,j+1}}{8} \tag{2}$$

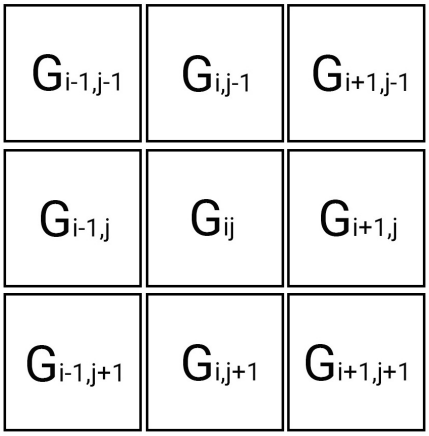

Figure 2: Visual representation of points in the grid

3.1 Sequential Implementation

In the sequential implementation, the simulation is executed on a single core without any parallel processing. The algorithm iteratively updates the temperature at each point in the grid one by one by running the following code for each point:

Algorithm 1 Pseudocode for calculating the temperature

```
Function calculateNewTemperature(x, y)

Check that x and y are inside the grid size
Set the totalTemp to the temperature of position given

for x-1 to x+1 do
    for y-1 to y+1 do
        if x and y are both 0 then
            continue                                  ▷ Skip the center cell
        end if
        if x and y are inside grid boundaries then
            Add temperature to totalTemp
            Increment totalCells by 1
        end if
    end for
end for
Calculate average temperature by totalTemp / totalCells
Return average temperature
End Function
```

3.2 Parallel Implementation

There are two different versions of parallel implementation. The first one focuses on the use of ForkJoinPool, while the second one uses IntStream. The version with IntStream seems to be 10-15% faster than ForkJoinPool. Both parallel implementations utilize multiple cores to simultaneously compute temperature updates for different sections of the grid. This approach is expected to reduce computation time significantly, and data shows that is true. Here are snippets of both versions of the code.

```
public void simulateParallel() {
    RecursiveTask<int[][]> task = new RecursiveTask<int[][]>() {

        @Override
        protected int[][] compute() {
            int[][] newGrid = new int[width][height];

            // splits work into chunks
            int chunkSize = Math.max(a:1, width / pool.getParallelism()); // adapts to the number of threads
            List<RecursiveTask<Void>> tasks = new ArrayList<>();

            for (int i = 0; i < width; i += chunkSize) { // for each chunk
                final int start = i;
                final int end = Math.min(i + chunkSize, width);

                tasks.add(new RecursiveTask<Void>() { // creates a thread
                    @Override
                    protected Void compute() {
                        for (int x = start; x < end; x++) // calculates new temperature for each cell
                            for (int y = 0; y < height; y++) // in the chunk
                                newGrid[x][y] = calculateNewTemperature(x, y); // and updates the new grid
                        return null;
                    }
                });
            }

            // Starts all threads
            for (RecursiveTask<Void> task : tasks)
                task.fork();
            // Waits until all threads end
            for (RecursiveTask<Void> task : tasks)
                task.join();

            return newGrid;
        }
    };

    try {
        int[][] newGrid = pool.invoke(task); // calls compute()
        updateGrid(newGrid);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Figure 3: Implementation with ForkJoinPool

```
    public void simulateParallel() {  2 usages   Miha
        int[][] newGrid = new int[width][height];

        // Parallelize over the X-dimension (columns)
        IntStream.range(0, width).parallel().forEach( int x -> {
            for (int y = 0; y < height; y++) {
                newGrid[x][y] = calculateNewTemperature(x, y);
            }
        });

        updateGrid(newGrid);

    }
}
```

Figure 4: Implementation with IntStream

3.3 Distributed Implementation

In the distributed implementation, the domain is divided between multiple nodes in a network, each responsible for a portion of the computation. Communication between nodes is required to share boundary information. I have used MPI (Message Passing Interface) inside MPJExpress[2]. The code should work similarly to the parallel one, splitting work to different threads(workers), but this time, each worker gets the whole grid sent by Broadcast method included in MPI. Then each of the workers is assigned to compute part of the grid. Recalculated data are collected and the process is repeated until the grid has cooled down.

4 Results

The tests were done on a 2021 Macbook Pro with an M1 Pro processor, 16GB or RAM, using MacOS 15.5 Sequoia and a Java JDK 23 and might vary when using a different computer configuration.

4.1 Testing by Increasing the Number of Heat Sources

In this test, the grid size was limited to 1024x1024, and the number of heat sources was set to 10. The algorithm was run multiple times, with the number of heat sources increased by 10 in each iteration until the number of heat sources reached 1024. Each configuration was run at least three times, and the average run-time was calculated for each mode (Sequential, Parallel, Distributed).

Sources	**Sequential (seconds)**			**Parallel (seconds)**			**Distributed (seconds)**		
	Run 1	**Run 2**	**Run 3**	**Run 1**	**Run 2**	**Run 3**	**Run 1**	**Run 2**	**Run 3**
10	4.47s	4.56s	4.42s	0.765s	0.791s	0.783s	7.556s	7.542s	7.832s
20	4.71s	4.59s	4.58s	0.854s	0.874s	0.858s	8.611s	8.471s	8.469s
30	4.59s	4.57s	4.70s	0.883s	0.880s	0.858s	8.428s	8.570s	8.615s
...	...	...	...	...	...	...	...	...	...
100	5.12s	5.00s	5.19s	0.991s	0.999s	0.966s	10.005s	9.860s	10.038s
...	...	...	...	...	...	...	...	...	...
250	5.25s	5.20s	5.26s	1.012s	1.057s	1.034s	10.760s	10.453s	10.525s
...	...	...	...	...	...	...	...	...	...
500	5.54s	5.14s	5.34s	1.065s	1.132s	1.041s	10.624s	10.876s	10.746s
...	...	...	...	...	...	...	...	...	...
1024	5.87s	6.08s	6.04s	1.094s	1.083s	1.072s	11.101s	10.966s	10.961s

Table 1: Runtime for Different Number of Heat Sources (Grid Size: 1024x1024) in Sequential, Parallel, and Distributed Modes

Heat Sources	**Sequential (s)**	**Parallel (s)**	**Distributed (s)**
10	4.48s	0.78s	7.64s
20	4.63s	0.86s	8.52s
30	4.62s	0.87s	8.54s
...	...	...	...
100	5.10s	0.99s	9.97s
...	...	...	...
250	5.24s	1.03s	10.58s
...	...	...	...
500	5.34s	1.08s	10.75s
...	...	...	...
1024	6.00s	1.08s	11.01s

Table 2: Average Runtimes with Varying Number of Heat Sources (Grid Size: 1024x1024)

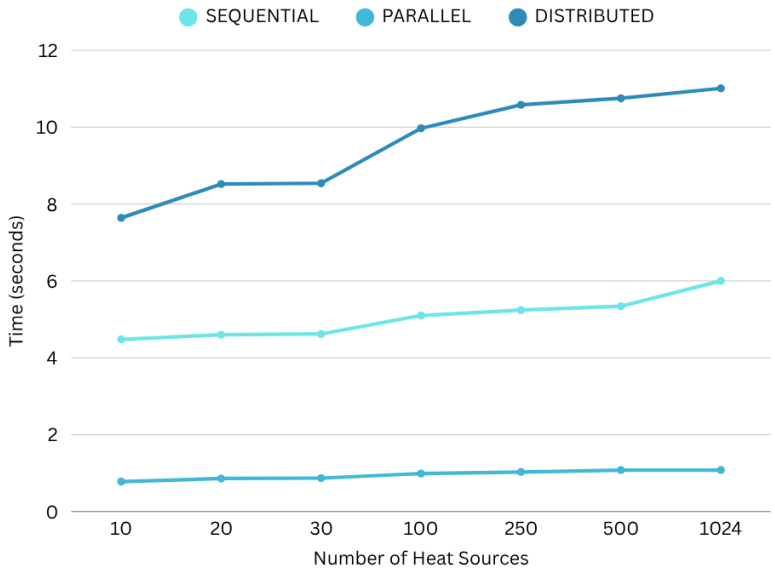

Figure 5: Comparison of Average Runtimes with Varying Number of Heat Sources

4.2 Testing by Limiting the Number of Heat Sources

In this test, the number of heat sources was fixed at 1024 and the size of the grid was started at 1024x1024. The grid size was increased by 1024 in both dimensions in each iteration to 10240x10240. Each configuration was run at least three times, and the average run-time was calculated for each mode (Sequential, Parallel, Distributed).

Size	**Sequential (seconds)**			**Parallel (seconds)**			**Distributed (seconds)**		
	Run 1	**Run 2**	**Run 3**	**Run 1**	**Run 2**	**Run 3**	**Run 1**	**Run 2**	**Run 3**
1024x1024	5.91s	5.90s	6.09s	1.09s	1.08s	1.11s	11.20s	11.06s	10.96s
2048x2048	21.68s	21.70s	20.98s	4.01s	4.16s	4.12s	28.72s	27.76s	29.34s
3072x3072	48.46s	46.87s	48.40s	9.02s	9.28s	8.90s	63.96s	64.09s	63.88s
4096x4096	85.10s	82.16s	85.06s	16.73s	16.89s	17.05s	95.03s	95.42s	95.28s
5120x5120	127.19s	131.79s	126.62s	23.98s	23.51s	23.46s	117.98s	117.23s	117.34s
6144x6144	187.11s	187.34s	187.51s	33.94s	34.47s	33.96s	151.47s	150.98s	151.01s
7168x7168	255.98s	256.12s	256.03s	44.55s	43.59s	45.67s	211.10s	212.13s	211.48s
8192x8192	320.91s	322.10s	320.67s	61.97s	61.21s	60.86s	247.71s	247.32s	247.88s
9216x9216	415.93s	416.47s	417.03s	67.82s	66.20s	66.39s	263.70s	264.13s	263.99s
10240x10240	516.95s	514.98s	516.34s	89.88s	89.97s	89.73s	383.89s	384.57s	384.81s

Table 3: Average Runtime for Different Grid Sizes (Number of Heat Sources: 1024) in Sequential, Parallel, and Distributed Modes

Grid Size	Sequential (s)	Parallel (s)	Distributed (s)
1024x1024	5.97s	1.09s	11.07s
2048x2048	21.45s	4.10s	28.61s
3072x3072	47.91s	9.07s	63.98s
4096x4096	84.11s	16.89s	95.24s
5120x5120	128.53s	23.65s	117.98s
6144x6144	187.32s	34.12s	151.15s
7168x7168	256.04s	44.60s	211.57s
8192x8192	321.23s	61.28s	247.64s
9216x9216	416.48s	66.80s	263.94s
10240x10240	516.09s	89.86s	384.42s

Table 4: Average Runtimes with Varying Grid Size (Heat Sources: 1024)

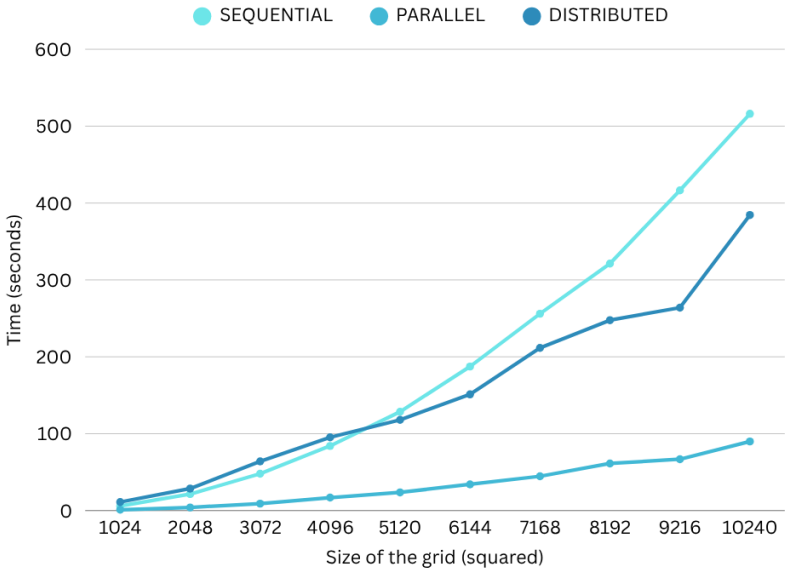

Figure 6: Comparison of Average Runtimes with Varying Grid Size

5 Analysis

The results show clear performance differences between the sequential, parallel, and distributed versions of the simulation. The sequential version was always the slowest, especially as the number of heat sources or the grid size increased, due to its single-threaded nature.

Both parallel versions were much faster—up to 5–6 times—thanks to multithreading. The IntStream version was slightly better than ForkJoinPool, likely due to lower overhead and better optimization in Java's stream API.

The distributed version had the highest runtimes on smaller problems, mainly because of the communication cost between nodes. However, for very large grids, it became more competitive. This suggests distributed computing only pays off when each node has a large enough workload.

In summary, parallel processing on one machine offers the best balance of speed and simplicity for medium-sized simulations.

6 Conclusion

The analysis shows that the parallel implementation provide significant performance improvements over the sequential approach, while the Distributed implementation only provides a slight improvement over the Sequential implementation when grid sizes are larger.

References

[1] Domen Vake Aleksandar Tošič Niki Hrovatin. *List of problems for Programming 3*. Accessed: 10 Dec 2024. 2024. URL: https://e.famnit.upr.si/pluginfile.php/731447/mod_resource/content/1/Programiranje_3_Projekti-3.pdf.

[2] MPJExpress. *About*. 2006/2008. URL: http://www.mpjexpress.org.

[3] ResearchGate. *Modelling of the temperature and residual stress fields during 3D printing of polymer composites - Scientific Figure*. Accessed: 15 Aug 2024. 2024. URL: https://www.researchgate.net/figure/A-few-frames-for-3D-printing-of-polymer-composites-using-layer-deposit-FDM-process_fig2_333719097.