Particles

Kristina Đukić
UP FAMNIT
89211057@student.upr.si

ABSTRACT

In this paper, we propose a framework for simulating the behavior of charged particles in a two-dimensional plane. The key idea is to see how the particles behave due to changes in their environment (i.e. how it affects their forces and movements). This is accomplished by using different modes: sequential (run on only one processor), parallel (run on multiple threads) and distributive (message passing between multiple single-threaded processes). The results obtained from the performance measurements demonstrate the efficiency and effectiveness of our proposed framework. Finally, we also describe the design of a simple GUI that is used to give visual representation for better understanding of the effects of charged particles.

1 INTRODUCTION

Particles is a simulator that demonstrates the behavior of charged particles, either positively or negatively charged, which attract or repel each other, respectively. The movement of the particles is affected by force between them, which is calculated by the following formula:

$$F = |C_i \cdot C_j|/d^2 \tag{1}$$

where c_i and c_j are the charges of the respective particles i and j and d is the euclidean distance between them.

After the particles have been assigned random position and velocity, as well as positive or negative charge, the distance is calculated. That is accomplished by looking at the position of two particles as vectors and aligning them to a standard position, and then calculating the distance between the vectors, which corresponds to the distance in the given formula. Subsequently, the calculated force influences the particles' movement by affecting their speed. The speed is added to the particles' positions on the x and y axes, resulting in the movement of the particles as a vector addition operation.

Particles move in two-dimensional plane that is bounded by rectangle, which also repels particles. Each particle is assigned either orangered color, if charged negatively, or blueviolet color, if charged positively. This color assignment facilitates the visualization of interactions between differently charged particles.

Algorithm 1 Calculate and Apply Forces

```
1: function CalculateAndApplyForces(particles)
2:     CONSTANT $k \leftarrow 10$
3:     CONSTANT $minDistance \leftarrow 20$
4:     for each $i$ from 0 to $n-1$ do
5:         for each $j$ from $i+1$ to $n$ do
6:             $dx \leftarrow particles[j].x - particles[i].x$
7:             $dy \leftarrow particles[j].y - particles[i].y$
8:             $distance \leftarrow$ Max(SQRT($dx^2 + dy^2$), $minDistance$)
9:             $force \leftarrow -k * particles[i].charge * particles[j].charge/distance^2$
10:            $forceX \leftarrow force * dx/distance$
11:            $forceY \leftarrow force * dy/distance$
12:            particles[i].speedX += forceX
13:            particles[i].speedY += forceY
14:            particles[j].speedX −= forceX
15:            particles[j].speedY −= forceY
16:        end for
17:    end for
18: end function
```

As we can see from the pseudo code, different particles can be processed independently, thus the algorithm can be further parallelized using multiple threads or processes. The main goal is to find out for which parameters we have better performance when using a sequential, parallel or distributed approach.

2 DESIGN

Parallelization offers significant performance gains when dealing with heavy computations, such as increasing the number of particles or cycles, where a each cycle represents update of positions of all the particles. Our chosen approach for parallelizing the algorithm involves assigning one partition of particles to each worker, where due to the structure of our algorithm (nested for loop) every particle from each of worker's partition would still be able to interact with every particle, without any problems

While this approach has proven effective in many cases, we have encountered drawbacks when the number of particles is small compared to the number of workers. In such situations, the overhead associated with managing multiple workers outweighs the benefits of parallel computation, leading to efficiency issues. Consequently, the performance becomes comparable to sequential execution, undermining the intended gains.

Another potential concern shared by both the Message Passing Interface (MPI) and parallel implementations is the uneven distribution of particles across threads or processes. If the number of particles cannot be evenly distributed, the last thread or process will be assigned the remaining particles. However, in our specific case, this does not pose significant problems. We work with a fixed, relatively small number of threads or processes, which is important especially in the context of MPI, which operates within a multicore configuration rather than across multiple devices. Consequently, the number of remaining particles is always minimal and does not significantly impact the overall performance.

3 PROBLEM DESCRIPTION

Every simulation approach has different challenges to meet and each of them is unique in their own way. The fundamental algorithm of the simulation always stays the same, but the environment where it is executed varies significantly. Let's have a look at these three solutions:

Sequential part

The single-threaded simulation is the most straightforward one. Firstly, in the main class, we initialize an array of particles with random properties such as position, speed, charge, and radius. Each particle is then processed in a sequential manner. We calculate the forces acting on each particle due to all other particles and update their velocities and positions accordingly. In this approach, all calculations are performed in a single thread, one after the other. After the computation is completed for all cycles, we note down the total runtime for analysis and comparison.

Parallel part

In this part, we employ a parallel computation strategy. We split the list of particles into equal-sized chunks, and assign each chunk to a separate worker thread. Each thread then computes the forces and updates the velocities and positions for its own set of particles. This process is repeated for all cycles. The calculations for different particles can thus be performed simultaneously on different threads, which can significantly reduce the total runtime on a multi-core processor. After all the threads finish their computations for all cycles, we again note down the total runtime.

Distributed part

For the distributed simulation, we use the Message Passing Interface (MPI) to distribute computations among multiple processes. In the beginning, the master process initializes an array of particles and broadcasts it to all worker processes. Each process then computes the forces and updates the velocities and positions for a unique set of particles. This process is repeated for all cycles. After each cycle, the master process gathers updated particle information from all worker processes and then broadcasts it back to them for the next cycle. The computations for different particles are thus distributed among multiple processes. After the computations for all cycles are completed, we again note down the total runtime on the master process.

4 GRAPHICAL USER INTERFACE

The drawing of the graphics is done independently of the computational threads. The user has option to toggle the graphics off, for the sake of the runntime and clear insight into computation, which is the most important point of this project. For the testing we have omitted graphics since it is impacting the run time and we cannot get concise computation results. Another thing to note is that Message Passing Interface (MPI) does not work with JavaFx library, so the distributive mode does not have graphical representation. The default size of the window is set to 800x600px ,but can be adjusted manually.

The animation timer is used to create the animation, which is responsible for the update of the state of objects, in our case particles. The animation timer also limits the drawing of frames to 60 frames per second (FPS), which is the requirement of our project. It also overrides the handle method in which the positions of particles are updated, boundaries are checked, and the particles are drawn on the canvas. After updating the particle positions and drawing them, the method for calculating the forces between particles based on their positions and charges is called. This process repeats in each frame of the animation until the specified number of cycles is reached.

So the drawing and the calculations are happening simultaneously for each frame and update of particle position. According to the graphics, we can easily see if the forces and movements of particles are being calculated correctly.

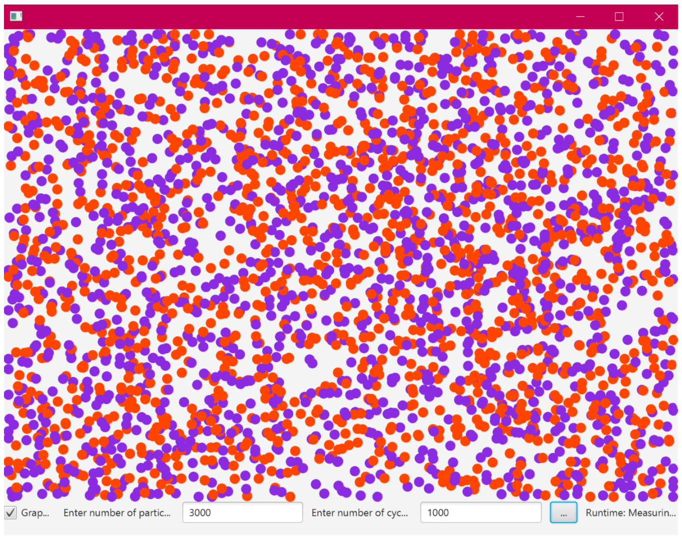

Figure 1: GUI Example

5 PROCESS AND THREAD

As mentioned before, the logic behind parallel and distributive mode is the same, but they behave differently, so it is interesting to see more in depth how these two work.

5.1 Process

The work is distributed between processes using main process where all of the particles are initiated and added to an array that is later broadcasted to all of the other processes using Bcast function from MPJ Express library. This way all of the other processes get their copy of the initial array. After that chunks of particle array are being divided per processes and starting and ending index of each chunk is calculated. We use Algorithm 1 to calculate forces on each process and then we save them to array again by using Gather function, again from MPJ express library. We then broadcast the updated array to all processes and we do this for every cycle. We can see that this is costly operation, since we have to broadcast and gather for each cycle, so as the number of cycles increase the performance gets really costly. We can see from the following algorithm the main idea for distributing particles between processes:

Algorithm 2 Particle Distributive Simulation Algorithm

```
1:  Get $processNum$ and $totalNumOfProc$
2:  numOfParticles $\leftarrow$ total number of particles
3:  particlesArrL $\leftarrow \emptyset$
4:  **if** $processNum = 0$ **then**
5:      Generate particles with speed, position and charge
6:      particlesArrL $\leftarrow$ generated particles
7:  **end if**
8:  Broadcast numOfParticles to all processes
9:  particlesPerProcess $\leftarrow$ numOfParticles/totalNumOfProc
10: startIndex $\leftarrow$ processNum $\times$ particlesPerProcess
11: endIndex $\leftarrow$ startIndex + particlesPerProcess
12: **if** processNum = totalNumOfProc − 1 **then**
13:     endIndex $\leftarrow$ numOfParticles
14: **end if**
15: Broadcast particlesArrL to all processes (startIndex $\longrightarrow$ endIndex)
```

5.2 Thread

For threads we use ExecutorService with fixed thread pool size equal to number of threads. The executor service is responsible for managing and executing the threads. We are dividing the array of particles same way as in distributive mode - by having starting and ending index and we give chunk of the array to each thread. Using for loop we assign chunk of array to each task, that threads will execute, of custom type that implements callable interface that implements Algorithm 1.

Each task is submitted to ExecutorService for execution. The main idea of parallel execution can be seen in the algorithm below:

Algorithm 3 Particle Parallel Simulation Algorithm

```
1: particles ← initializeParticles()
2: executor ← initializeExecutor()
3: numParticles ← particles.size()
4: particlesPerThread ← numParticles / numOfThreads
5: futures ← createEmptyList()
6: for i ← 0 to numOfThreads − 1 do
7:     startingIndex ← i × particlesPerThread
8:     endingIndex ← startIndex + particlesPerThread
9:     task ← createParticleTask
10:    future ← submitTaskToExecutor(executor, task)
11:    futures.add(subitted.task)
12: end for
13: waitForTasksToComplete(futures)
14:
15: shutdownExecutor(executor)
```

6 TECHNICAL REMARKS

The program was implemented in Java with helper libraries. The program can be run using Java 8 due to incompatibilities of MPJ Express and JavaFx in later versions of Java.

7 RESULTS

The testing was done on AMD Ryzen 5 PRO 2500U processor with 4 cores and 8 logical processors. First the number of particles was fixed at 3000 and cycles were increased by 500 up to 10000. Each mode was run three times with every parameter and average time was taken as the resulting one. The second testing was done by setting the number of cycles to 10000 and particles were increased, again by 500 up to 3000, since this has proven to be taking a lot more time to run, we stopped at 3000 particles.

Based on the graphs depicting the testing results with fixed particle size, it is evident that the execution time increases exponentially across all algorithms. The parallel implementation consistently outperforms the sequential and distributive approaches. However, in the distributive testing, although initially comparable to the

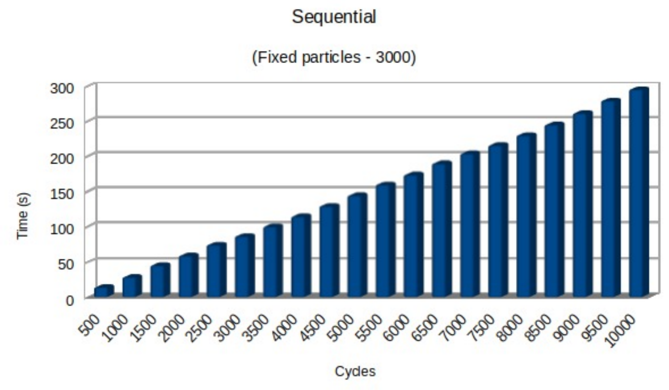

Figure 2: Results for sequential testing with fixed particles

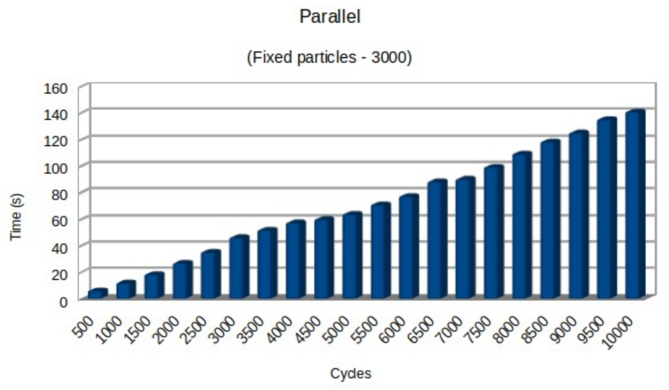

Figure 3: Results for parallel testing with fixed particles

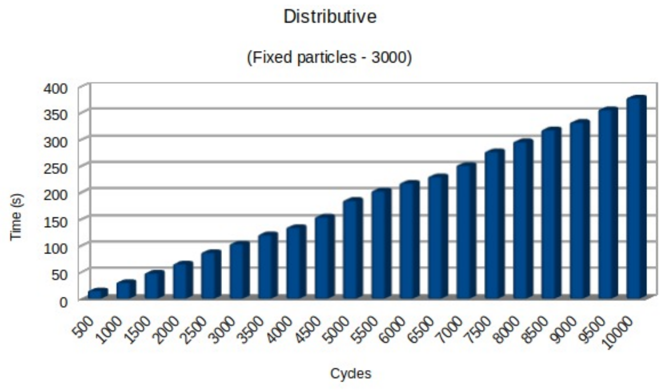

Figure 4: Results for distribtuive testing with fixed particles

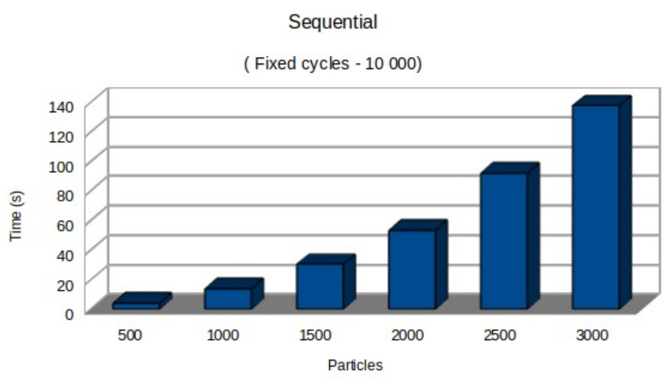

Figure 5: Results for sequential testing with fixed cycles

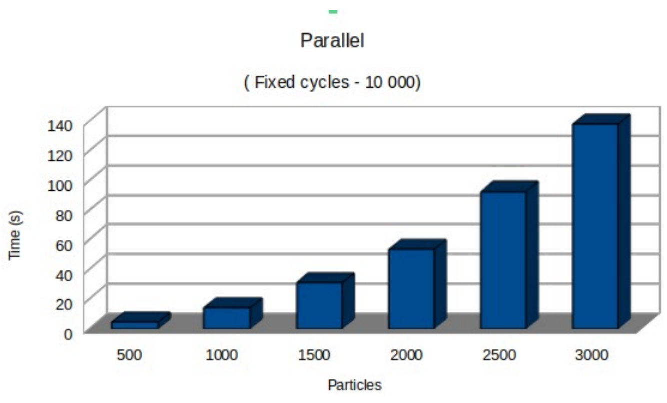

Figure 6: Results for parallel testing with fixed cycles

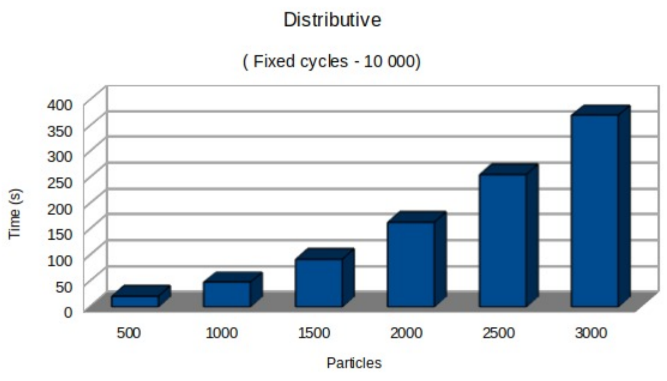

Figure 7: Results for distributive testing with fixed cycles

sequential approach, the distributive algorithm exhibits slower performance as the number of cycles increases, as described in Section 5.1.

On the other hand, in the testing with fixed cycle size, notable differences emerge with each 500 particle increment. The execution time is considerably higher compared to the testing with fixed particle size. Despite this, the parallel implementation remains the most efficient. Notably, a big performance gap is evident from the beginning between the sequential and distributive testing, owing to the large number of particles involved."

8 CONCLUSION

From everything written above, we can draw the following conclusion: the given problem can be addressed using three different approaches - sequential, parallel, and distributive implementations. Extensive testing has consistently demonstrated that the parallel implementation offers the highest efficiency. But, the distributive implementation, which shares a similar underlying logic to the parallel approach, shows a significantly longer runtime due to the expensive functions involved in distributing particles among processes. As the number of cycles increases, this becomes more and more visible.

We can also conclude that the program has better performance with large number of particles and smaller number of cycles, then the other way around.