

Parallelize Particle Simulation
Concurrent Programming
ID1217 vt12

Dara Reyahi

Carl Regårdh

Abstract

This report describes the Parallelize Particle Simulation programming project, our goals for an implementation and the results from our implementation.

Given four different programs (one sequential, one using Pthreads, one using OpenMP and one using MPI) all running the simulation in time $O(n^2)$, we successfully improved the implementations to run in time $O(n)/p$ when using p processors.

1 Introduction

As processors continue to get more and more cores the concept of parallelization has become increasingly important. The key idea in parallelization is, as one might have guessed, to split the computations over several processes, or threads, such that each new process only works on a subset of the complete task. When all the computations are made the results are combined to form the final answer.

This gives rise to a whole host of problems. What happens if one process changes the value of a variable that another process uses later on? Or what happens when two processes change the value of a variable at the same time, what value does that variable end up with? In order to solve these types of problems different mechanisms (among others: *mutual exclusion*, *barriers* and *locks*) have been invented and are frequently used. These techniques work great to keep parallel programs dependable but they cause what is known as *parallel slowdown*, the delay caused by processes waiting for one another.

It is often very hard to decrease the amount of parallel slowdown in a program since the different synchronization techniques are there for a reason, without them it would be impossible to guarantee that the program will run in a dependable fashion. In order to get the best performance one must also look at other factors, such as what algorithm was chosen and how it was implemented.

Our task in this project then was to improve the performance of a particle simulation program not only through parallelization but also by developing a quicker algorithm.

1.1 Problem description

Given four particle simulation programs running in $O(n^2)$ time complexity, where n is the amount of particles, our task was to develop the following four programs:

- A sequential program that runs in $O(n)$.
- A parallel program using the native pthreads library that runs close to $O(n)/p$, where p is the number of processes.
- A parallel program using the OpenMP library that runs close to $O(n)/p$.
- A parallel program using the MPI library that runs close to $O(n)/p$.

The usage of the different libraries involved was known to us from earlier and it was clear to us that the make-it-or-break-it issue would be improving the algorithm.

2 Solution

Our first step for implementing our solution was to look at the already given implementation and see if we could find the bottleneck of the program. Starting with the serial implementation, we quickly found the critical section of the algorithm where (in each timestep) each particle is checked for collision:

```
n ← numberOfParticles
for i = 0 → n do
  particles[i].ax ← 0
  particles[i].ay ← 0
  for j = 0 → n do
    apply_force(particles[i], particles[j])
  end for
end for
```

Clearly, the given algorithm was running in time $O(n^2)$ because in order to update the simulation, it checked every particles movement with every other particle in the simulation. But was this really needed? The only way to find out was for us to take a look at the *apply_force*(*particle a*, *particle b*) function, were we found something interesting:

```
cutoff ← 0.01
dx ← b.x − a.x
dy ← b.y − a.y
r2 ← dx * dx + dy * dy
if r2 > cutoff * cutoff then
  return
end if
...
```

The function would simply return without doing anything at all if the two particles were not within 0.01 distance units away from each other. Since the width and height of the total area was about 0.7 when using the default value of 1000 particles, checking every particle with every other was incredibly inefficient when only about 0 – 2 particles would be in range each timestep.

The obvious way to improve this algorithm and take make it run in $O(n)$ was to reduce the *apply_force*(*particle a*, *particle b*) function calls so that each particle was only checked with particles in its close vicinity.

We had different ideas of how to do this, but we finally settled for a solution that would prove work exceptionally: by dividing the simulation area into a grid, where each grid-square had width and height of $\geq cutoff$. Thus, for each particle we only had to check the particles in the neighboring grids since they were the only ones who could possibly be close enough for a collision to happen.

fortsattning... hur vi gjorde griddet, find nearest etc.