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:

$n \leftarrow numberOfParticles$
for $i = 0 \rightarrow n$ **do**
  $particles[i].ax \leftarrow 0$
  $particles[i].ay \leftarrow 0$
  for $j = 0 \rightarrow n$ **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:

$cutoff \leftarrow 0.01$
$dx \leftarrow b.x - a.x$
$dy \leftarrow b.y - a.y$
$r2 \leftarrow dx * dx + dy * dy$
if $r2 > cutoff * cutoff$ **then**
  $return$
end if
...

The function would simply return without doing anything at all if the two particles where not within 0.01 distance units away from each other. Since the width and hight 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 widht and hight of $\geq cutoff$. Thus, for each particle we only had to check the particles in the neighboring eight grids since they were the only ones who could possibly be close enough for a collision to happen.

After creating the grid, we implemented a *findNearest* function that returned all the particles in the vicinity of the input particle. In each timestep the grid had to be filled with the new positions of all particles, but this is done in $O(n)$ by simply checking each particle and now each particle only needed to be checked for collision by the ones close to it:

$n \leftarrow numberOfParticles$
for $i = 0 \rightarrow n$ **do**
  $particles[i].ax \leftarrow 0$
  $particles[i].ay \leftarrow 0$
  $nearParticles[] = findNearest(particles[i])$
  for $j = 0 \rightarrow nearParticles.size$ **do**
    $apply\_force(particles[i], nearParticles[j])$
  end for
end for

Since the particles are evenly distributed over the area, *nearParticles.size* will rarely be greater than one (only in the case of a particle colliding with multiply others in the same timestep) and thus we had reduced the algorithm to $O(n)$.

2.1 Pthreads

When adapting this solution to be able to run parallel with *pthreads* a slight change was needed: we split the particles up in a way so that each thread only checked collisions for a range of particles depending on the thread's id. Thus dividing the workload evenly.

$first \leftarrow min(thread\_id * particles\_per\_thread, n)$
$last \leftarrow min((thread\_id + 1) * particles\_per\_thread, n)$
$n \leftarrow numberOfParticles$
for $i = first \rightarrow last$ **do**
  $particles[i].ax \leftarrow 0$
  $particles[i].ay \leftarrow 0$
  $nearParticles[] = findNearest(particles[i])$
  for $j = 0 \rightarrow nearParticles.size$ **do**

```
    apply_force(particles[i], nearParticles[j])
  end for
end for
```

Where $particles\_per\_thread = (n + numThreads - 1)/numThreads$. By having a barrier at the end of this section of code we also guaranteed that no thread would finish and start moving particles in the next timestep while others where still working.

2.2 OpenMP

To adapt the solution to be run in parallel in OpenMP we had the same approach of dividng up the particles checked between the threads. However, in OpenMP this was done much easier by simply using *omp for*:

```
n ← numberOfParticles
#pragma omp for
for i = 0 → n do
  particles[i].ax ← 0
  particles[i].ay ← 0
  nearParticles[] = findNearest(particles[i])
  for j = 0 → nearParticles.size do
    apply_force(particles[i], nearParticles[j])
  end for
end for
```

Since the grid was shared among the threads, we also had to make sure that it was only filled by one of the threads using *#pragma omp single* to avoid segmentation faults when adding particles.

2.3 MPI

Since we are unable to have global data accessable among the processes when using MPI, we had to collect all global data locally at the beginning of every timestep using the *MPI_Allgatherv* function. Each process had a local array containing the particles it was responsible for checking, this was (just as when using pthreads) calculated using the processe's *rank* and the number of processes. Thus, the collision checking loop only checked the particles contained in its local array:

```
nlocal   //size of local array
local[]  //particles recieved from MPI_Allgatherv
```

for $i = 0 \rightarrow nlocal$ **do**
  $local[i].ax \leftarrow 0$
  $local[i].ay \leftarrow 0$
  $nearParticles[] = findNearest(local[i])$
  for $j = 0 \rightarrow nearParticles.size$ **do**
    $apply\_force(local[i], nearParticles[j])$
  end for
end for

---

3 Results

3.1 Serial

3.2 Pthreads

3.3 OpenMP

3.4 MPI

4 Discussion