Grant Mitchell

CP341 Interstellar Simulation

Simulating Interstellar Bodies using OpenMP Parallelism

*Simulating Gravity on a large scale is a problem with an enormous amount of calculations. Every gravitational body in the system is affected by every other body, making the problem scale at least on the order of N2. In this project the OpenMP parallel framework was used to speed up the calculation of this algorithm, using a map pattern, a reduce pattern, and a fork-join pattern. In the final implementation, it was found that writing to a file was the slowest operation in our algorithm, so forking the write to a new thread was the biggest factor in speedup.*

1. **Introduction**

To simulate a gravitational system. Over every timestep of some arbitrary length in the system. The acceleration, change in velocity, and change in position must be calculated for each object. Change in velocity and change in position come directly from acceleration in one operation, so calculating the accelerations is really the critical path for the algorithm. Each acceleration is calculated based on the position and velocity of every single other object in the system. This means that as the number of bodies increases, the complexity of calculating accelerations scales on the order of N2. The other operation that is important to my implementation of the problem, is record keeping. At each timestamp I wanted to record the fields of every single object, so that these data files could be used to visualize a record of what happened in the simulation. Using the text data files created, I could draw dots on a canvas for interstellar objects, and create a useful visualization of the simulation.

These two problems, calculating acceleration and writing to a file, are what make up the bulk of my processing time for this algorithm. This setup becomes an interesting and relevant problem for parallelization, because each of these problems appears to want a different form of parallelism. The calculation of acceleration can be easily applied to a map parallel pattern. Each object takes in the same initial state of the galaxy, does some math on the current body, and produces a new set of output values for that body. This process lends itself nicely to a map pattern. Each bodies calculation can run on a separate thread, and then all the sets of values produced can be written to a single file. As long as we make sure each thread starts on a different current body, we can parallelize the acceleration calculation.

The parallel pattern that lends itself to the file write is the fork-join pattern. Anytime we want to write to file, instead of waiting for the acceleration calculation of the next step to finish, we can just send the file write function off into its own thread, for it to complete whenever it can. File writing is often an inefficient operation, so giving it its own thread could significantly improve performance of acceleration calculations. In the following sections, I describe how I have used OpenMP parallelism to implement these forms of parallelism, and induced some speedup of the algorithm.

1. **Design:**

My implementation of this problem focuses around the manipulation of text files, and is divided into three parts. The first phase is the generation of the text file with the initial state of the system. The second phase, in which the most work is done, is the transformation and copy of the initial text file for each timestep in the simulation. The third phase is unnecessary but aesthetically pleasing. In the third phase the text files that contain each timestep are taking and used to generate an image for each timestep. This allows the end user to get a good idea of the physical results of the simulation in 2D space.

The first and third pieces of this problem I accomplished by writing simple, serial python scripts. Doing graphics in parallel in c seemed like more than I wanted to tackle. The bulk of the work, and all the parallel processes. Happened in the second portion. The generation of the text file with the state of the system for each timestep. Below is the basic logical code structure.

for t in timestep {

#pragma omp parallel for

for b in body\_list {

centroid = calculate\_Centroid(body\_list);

out\_buf.append(update\_current\_body(b,centroid));

}

Write\_buff = copy(out\_buff);

#pragma omp parallel task

{

writetoFile(Write\_buf);

}

body\_list = out\_buf;

out\_buf = NULL;

}

This pseudocode demonstrates the logical flow of the solver, and show two of the three parallel constructs used in my solution. The OpenMP parallel for construct is applied to this solution. Each body operates on its own thread, so the centroid calculation and body updating can occur concurrently. This is a significant amount of processing when body count goes up, because centroid calculation scales on the order of N2. The other parallel construct implemented here is a fork. Writing to a file is a CPU intensive operation, and so I send the write off into its own thread where it can accomplish the write without slowing down the main computation of the program. This is done with the OpenMP task pragma. It is important to note that in this fork we must give the forked thread its own write buffer, so that we do not instate a race condition between the write thread and the processing thread. The third parallel construct is not represented in this source code, but is important nonetheless. Inside the centroid calculation function, I implemented a OpenMP reducer, because most of that function is summing three different variables across a large array. The following task graph shows the work flow of the program for one timestamp.

Centroid calculation

Out File

Write to file

Copy write buffer

Update current Body

Place in struct

In data

1. **Performance Analysis:**