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 N^2. 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 time, during each 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 N^2. 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. 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.

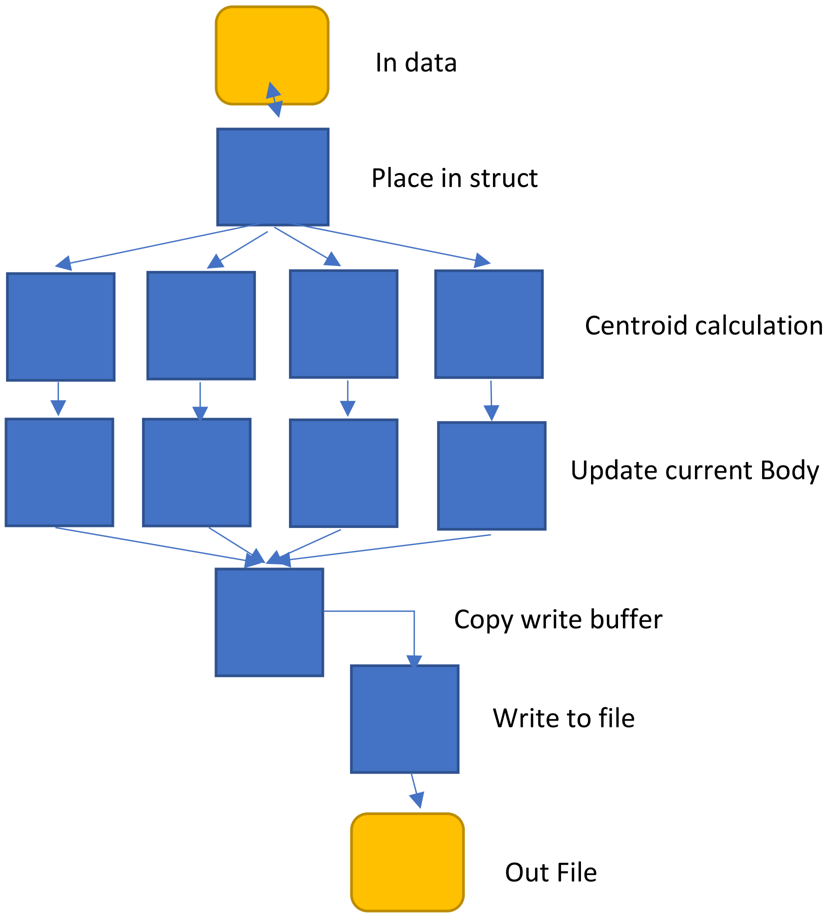
2. Design:

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 parts of 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 N^2. 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.



3. **Performance Analysis:**

To analyze the performance of my algorithm, I used the unistd time function which grabs the clock time of the CPU, and used it to time many different iterations of my algorithm. I timed across different galaxy sizes and different numbers of timesteps, on several different versions of my algorithm. My control was a serial algorithm, with no parallel constructs. Algorithm 2 was the same algorithm but contained the OpenMP reducer in the centroid calculation. Algorithm 3, had both the reducer and the parallel for constructs, but not fork-join. Algorithm 4 had all the constructs, a reducer, a for, and a fork join. This was my initial setup, but I found that another Algorithm was needed to be tested, Algorithm 5 contained the reducer and the fork join, but no parallel for.

The initial results of my data are below, all times are in milliseconds

	Serial	2 (reduce)	3 (reduce, par_for)	4(fork-join, reduce, par_for)
3 bodies 100 timesteps	150	175	169	61
600 bodies 100 timesteps	386	381	383	117
6000 bodies 10 timesteps	2380	2533	2558	2332
6000 bodies 100 timesteps	25596	24999	25291	22299
12000 bodies 100 timesteps	9765	9604	9862	8965
100000 bodies 1 timestep	68684	61481	64316	61689

The first observation we can make from this data is a confirmation of where the parallel work is being done. As the number of bodies in the algorithm increases, so does the parallel work we can do. We can see this in that with a small number of bodies, the serial and parallel versions of the algorithm have very little difference. We also see very little difference between the parallel version with just the reducer and the one with the parallel. As the body count gets higher, we start to see a difference. Somewhere between 6,000 and 12,000 bodies, the parallel algorithm overtakes the serial algorithm in performance. We can also see this trend continue as at 100,000 bodies the parallel algorithm is a significant number of seconds faster.

A separate but important observation is that the parallel for appears to make our algorithm slower. Instead of overtaking the serial algorithm somewhere between 600 and 12,000 bodies, the parallel for algorithm overtakes the serial algorithm somewhere between

12,00 and 100,000 bodies, and even at 100,000 bodies, it is 4% slower than the parallel algorithm with just the reducer. These results are a little baffling, but the most likely solution has to do with spawning threads. Depending on how the parallel for works, it could be doing a lot of work spawning and destroying threads in between each timestep, this is a very performance heavy activity and could significantly affect the time of the algorithm.

If we start to look at the fork join implementation in our analysis, we can see that the fork clearly had the biggest impact on performance. For all the results except 100,000 bodies, the implementation with the fork-join has the fastest time. This makes clear that the write to file was the most time-intensive operation in the problem, and so forking that operation into its own thread while the other program kept running allowed for writes to happen concurrently which drastically increased speedup.

These results led me to question if the parallel for was necessary in the fork-join implementation, or if it was perhaps slowing it down, so for another test I compared the reduce only implementation, to the fork-join and reduce implementations

	Serial	2 (reduce)	5(fork-join, reduce)
3 bodies 100 timesteps	154	175	60
600 bodies 100 timesteps	393	436	79
6000 bodies 10 timesteps	2668	2492	2331
6000 bodies 100 timesteps	25430	25018	22945
12000 bodies 100 timesteps	10416	9573	9007
100000 bodies 1 timestep	64938	63498	61305

The results of this test solidified my suspicions. The fork got even faster with the removal of the parallel for, faster than the reduce on every test. It turns out that most of the time being spent in the serial algorithm was simply waiting for the file writing process to complete. Letting that task run concurrently was a huge boost to performance. On the large numbers of bodies, we get improvements of two seconds. On a simulation that takes much longer, with millions of bodies over 1000s of timesteps, this could be an improvement of hours.

4. Conclusion:

In the end, my simulations were small. 100,000 bodies does not come close to the size of a single galaxy, and I only simulated it over 1 timestep. I now have a greater understanding of the optimizations in terms of memory, parallelization, and hardware that must be done to do a more realistic and long term gravitational simulation. This problem, while mathematically simple, scales very quickly to be hugely computationally complex. I used simple parallel constructs to improved performance, my maximum speedup was only 1.05, which is not enough for a larger scale simulation. Clearly the fork-join, my maximum speedup technique, was useful, but was still limited by the organization of my memory and the centroid calculation, which I was not doing fast enough. It was interesting that the parallel for caused a decrease in performance, and perhaps with more time and experience, I could adjust the memory allocation and thread building so that the main part of the algorithm ran much faster. As often seems to be the case with parallel programming, a speedup was achieved, but it wasn't enough.