Project 3: N-Body Simulation

Justin Cohler
6/6/2019

Background

The motivation for this project comes from an exemplary "embarrassingly parallel"[1] problem: an N-Body Problem. Given *n* distinct celestial bodies with distinct masses and positions (and optionally distinct non-zero velocities and accelerations), determine each bodies position at discrete time increments in the future, based on the relative gravitational pull of the other bodies at a given time *t*.

Assuming Newtonian physics, the mathematics behind this problem are quite simple, and can be represented by the following equation[2]:

$$\sum_{i \neq n}^{N} \frac{G m_n m_i}{{r_{ni}}^2} \hat{r}_{ni}.$$

Above, the total force on the *ith* object is the summation of the relative forces from each object on *i*. This is a simple enough calculation, but on large numbers of bodies, this problems computations increases polynomially, as each body must sum forces of every other body individually. This problem is easily parallelizable, as each body's updated velocity at time *t* is independent of all other bodies' updated velocities. Therefore, each body's calculation may be sent to a separate processor, in a functional decomposition. This architecture is described below.

The mathematical reasoning for this computation leveraged the work of Ben Eagan's blog post on n-body simulations[3], which performs a similar calculation in a single thread in Python. For more detail into the mathematical underpinnings of this simulation, I recommend this post.

Program Input

The input to the program consists of the following flags:

- bodies - the number of bodies to simulate
- steps - the number of steps to calculate new positions
- daysPerStep - the number of days between step calculations

[1] https://en.wikipedia.org/wiki/Embarrassingly_parallel
[2] http://physics.princeton.edu/~fpretori/Nbody/intro.htm
[3] http://www.cyber-omelette.com/2016/11/python-n-body-orbital-simulation.html#theprogram

- threads - the number of threads (by default will parallelize if not supplied)

On creation, the supplied number of bodies are created with a random mass between 1e20 and 1e30 kilograms, a random position in three dimensional space, a zero velocity, and a zero acceleration.

An example usage is shown below:

```
go run nbody.go -bodies=8 -steps=100000 -daysPerStep=5 -threads=4
```

The main methods for parallelization as well as the main runner are encapsulated in *nbody.go*, and the methods for computing updated accelerations, velocities, and positions are modularized in located in *physics/physics.go*.

The Worker Pool

I decided to use a worker pool for this project. The worker pool is created in the *simulate* function, inside of *nbody.go*. At the outset of simulation, a number of workers are created. This number is the minimum of either the number of threads provided at input, and the number of bodies provided at input.

Each worker is supplied a list of pointers to all the bodies generated, a channel to receive work, and a channel to alert when the worker has finished its task. On the receive-work channel come context objects containing a start and end position in the list of bodies to work on (see BSP Parallelization[4] for more context into this approach). Once the work has been done, a simple dummy integer is placed on the channel to alert that work is completed.

Simulating a Step

Once all workers have been spawned, a goroutine begins an infinite loop in the *simulate* function. Each time through the loop, the routine will split out the list of bodies into *n* blocks, and will send these blocks to the worker channel, which is buffered to accept *n* messages. This goroutine then waits to receive *n* messages back from the worker threads before continuing on. Once all worker threads have finished their work for a given step, this outer goroutine alerts a channel called *stepDone*, which tells the listener that all computation for a given step is completed. It does so in a *select* statement, which also looks for a termination of its own each time through the for-loop.

Lastly, the *simulate* function returns the *stepDone* channel, a generator[5] pattern.

[4] https://en.wikipedia.org/wiki/Bulk_synchronous_parallel
[5] http://www.golangpatterns.info/concurrency/generators

Calling *simulate* yields a generator function with a channel that will receive messages when a step of simulation is completed. Therefore the program loops through this generator *m* times, the number of steps input into the program, and then prints the final position of each body.

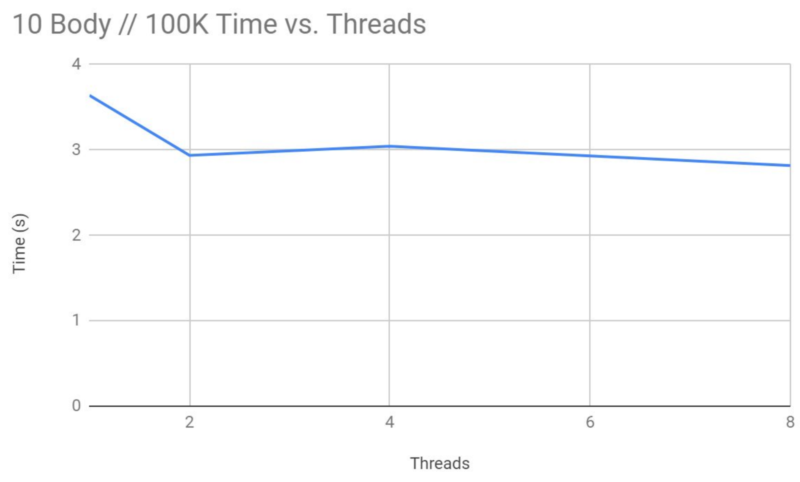
Results

All raw results can be found in the Appendix below.

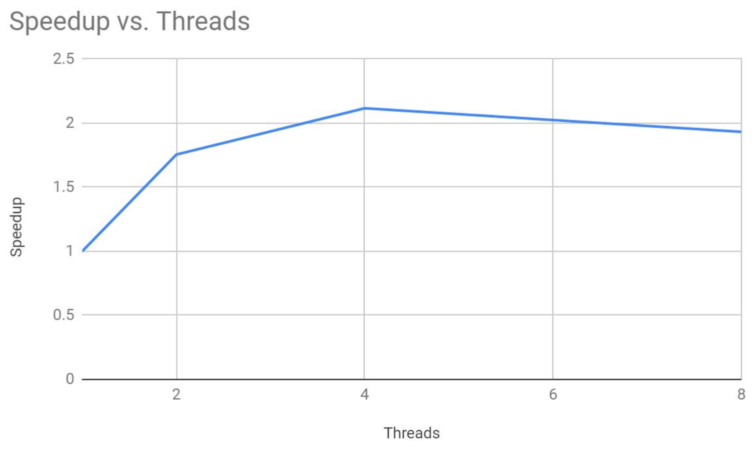
Results came from my laptop, with the following specs:

- Intel Core i5 Processor
    - 2 cores, 4 hardware threads
- 8GB DDR3 RAM

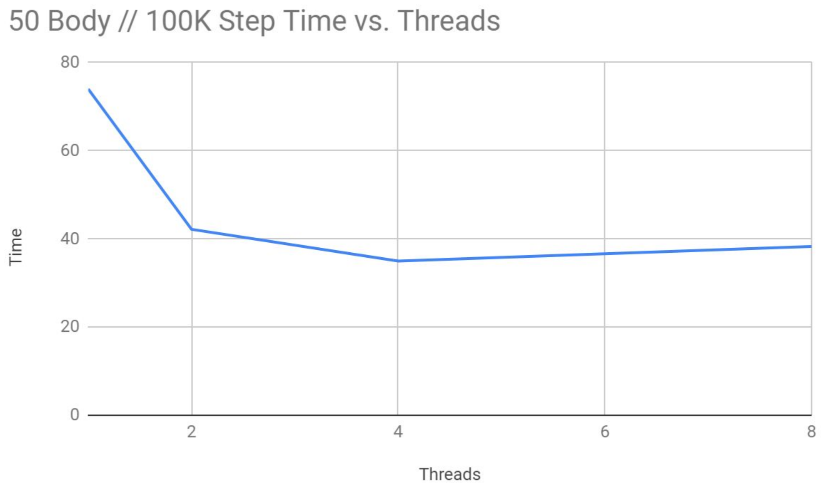
The biggest factor during runtime was in regard to the number of bodies, whereas the number of steps appeared to be linear in time, as was expected for this model, where all computation needs to be completed at each step before any worker can move forward with future steps. What is interesting in our first result below, which shows 10 bodies and 100,000 steps, is how little our addition of more threads helps. This may very well improve more as we increase the step count, but this appeared to be the pattern for both 10,000 and 100,000 step counts, which shows that at low body numbers, the benefit to adding threads is fairly menial.



As bodies increase however, the benefits grow greatly. As we see below, up to 4 threads (which is the number of hardware threads available on the testing machine), we see steady increases in performance in terms of our speedup metric, dropping off slightly between the two and four thread statistic, but then rapidly flatlining after four threads. This is to be expected, as the runtimes ability to leverage threads will not greatly exceed the threads on the machine.



The reason for the potential slowdown in performance gains likely stems from the fact that not all of this logic could be parallelized, and that when blocks of computation were split up in a BSP parallelization process, there may be cases where some cores were slower than others and caused the others to wait. Additional rebalancing logic could be implemented in this project as a future step to see how much more this program could conform to the ideals of Amdahl's law, which states that out to four threads in our case, we should see parallelization speedup ratios of nearly 4 for an embarrassingly parallel simulation like this.



Conclusion

More work is necessary to ensure blocks of computation at each step of execution in this simulation do not act as weak links the chain, requiring threads to wait for the slowest to complete. Splitting into smaller chunks, rather than a BSP model may help get the speedup ratio of this program closer to the ideal Amdahl metric (out to the number of threads available on the machine). More testing is required to determine whether this is possible, and how much performance gains may be realized.

Appendix: Raw Results

Threads	**Bodies**	**Steps**	**Time**
1	10	10000	0.485
1	10	100000	3.639
1	50	10000	7.426
1	50	100000	74
2	10	10000	0.383
2	10	100000	2.936
2	50	10000	4.297
2	50	100000	42.173
4	10	10000	0.378
4	10	100000	3.044
4	50	10000	3.552
4	50	100000	34.958
8	10	10000	0.361
8	10	100000	2.817
8	50	10000	3.882
8	50	100000	38.29