/*--
 * Tanner Bernth
 * Robert Walters
 * Project 1 Report
 --/

/*--
 * **Documentation**
 --/
Our program includes three special features:

- Ability to handle bodies of various masses
- Graphical interface
- Percentage of time spent in barrier

Because of these features, our command line arguments vary slightly from the 4 recommended in the spec. Our program is run using:

- `java <# workers> <# bodies> <# time steps> <GUI (0 or 1)> <timings (0 or 1)>`

For example:

- java 8 100 1000 1 0

/*--
 * **Introduction**
 --/
These programs involve solving the n-body problem, which simulates the evolution of the galaxy. Each body in the galaxy has a mass and an initial position and velocity. Gravity causes acceleration and movement between these bodies. The n-body system simulates the evolution by calculating the forces on every body at each time step. To make the programs more interesting, elastic collisions were included, meaning the total energy remains the same after the two objects collide.

To simulate the n-body problem, we created both a sequential and parallel version in order to see the differences in performance for each. Additionally, we created a graphical representation of the programs in order to see the results of the calculations and the collisions.

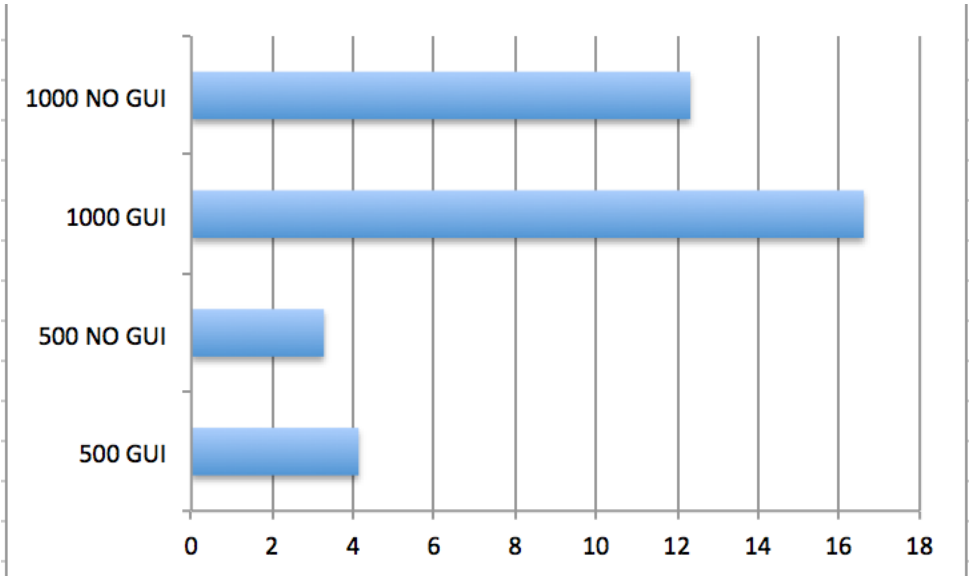
/*--
 * **Programs**
 --/
For our n-body problem solutions, both the sequential and parallel versions use the same GUI, planet, and point classes, while the parallel version has additional classes, which make it parallel. For the initial conditions, we just used random positions, velocities, forces, and masses. Using random initial conditions allows for many varying types of interactions between the bodies. The biggest enhancement to the solution is the GUI, using the java.swing class, that makes it very simple to visualize what is happening in the code. Since the GUI is in use during the calculations, it slows the execution time

somewhat, but the option to hide the GUI is available as well. The input for the programs includes the number of threads (which does nothing for the sequential version), the number of bodies, the size of the bodies (although we simply made the sizes of all bodies randomly), the number of time steps, and either 0 or 1 for whether or not to use the GUI.

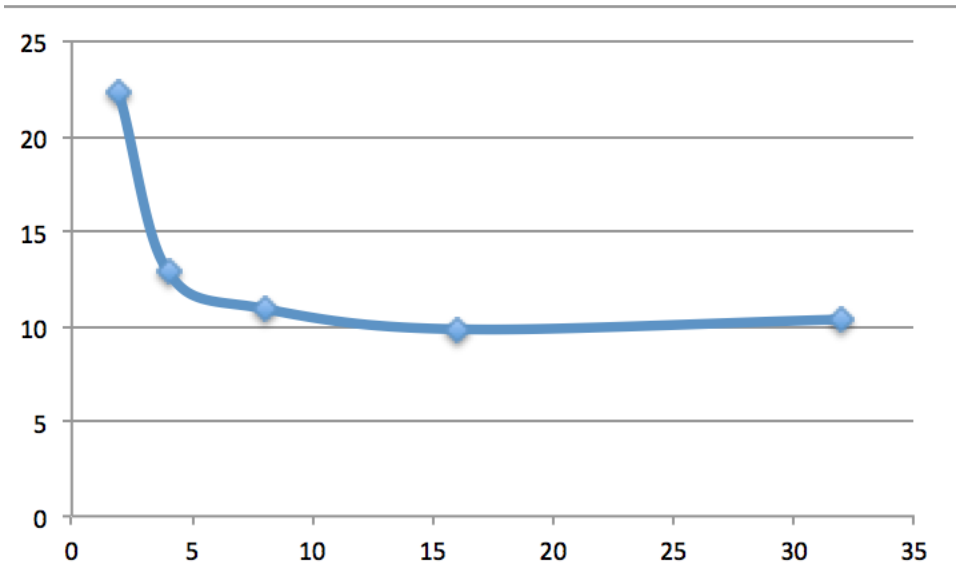
The programs time the duration of the calculations as soon as all of the bodies and threads have been initialized. After the calculations are finished, the computation time and the number of collisions is output and the final positions and velocities of all of the bodies is input into a file named results.txt.

/*----------
 * **Verification**
 ----------/
The programs were much easier to verify they were correct with the use of the GUI. The GUI allowed us to visualize the changes that were happening as a result of the code. When two bodies collide, they are supposed to be launched in a different direction based on the forces and velocities of the two bodies, which is visible from our trial runs. Two bodies colliding together also maintains that they are supposed to elastic collisions because a faster body may slow down after colliding, but the other body will speed up accordingly. Using the GUI, we see that the collisions are resolved correctly because the bodies only have changes in their velocities when they directly touch another body. They can come very close to another body and have no effect as a result of a collision.

/*----------
 * **Timing Experiments**
 ----------/



This shows the speed test with the average time on the sequential program using a GUI and without a GUI. The top two use 1000 bodies with 1000 time steps and the bottom ones use 500 bodies with 1000 time steps. These results show that the sequential program's algorithms are n^2, since the program run time doubled when the number of bodies doubled.



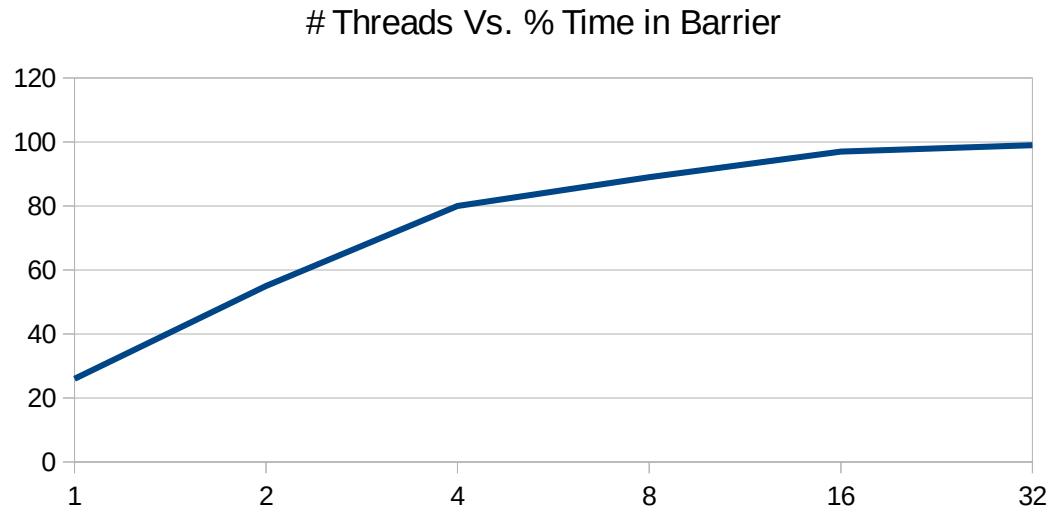
This shows the average time of the execution of the parallel version of the n-body program run on the oxford server. This shows the difference in times from 2, 4, 8, 16, and 32 threads. This graph shows that increasing the threads significantly increases the execution time of the program but quickly plateaus near 10 seconds. 16 threads is most likely the most optimal number of threads to be using with this number of bodies and time steps.

/*--
 * **Other Experiments**
 --/

We ran timings to examine how much time is spent at the barrier in relation to the number of threads. These timings were run on Oxford.

	1 Threads	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads
Timing 1	25.68%	57.87%	79.75%	89.40%	97.32%	99.41%
Timing 2	27.38%	54.27%	81.47%	88.61%	97.28%	99.56%
Timing 3	24.85%	53.75%	78.98%	89.87%	97.55%	99.20%
Average	26%	55%	80%	89%	97%	99%

When combined with the timing info, it seems that in many cases, the power of using many threads is mitigated by them constantly waiting at the barrier!



/*--
 * **Conclusion**
 --/

Our takeaways from this project are several:

- For a hugely complex problem such as this, parallel processing isn’t enough to make the calculations trivial!
- For a problem thats easy to parallelize such as this, there are no guarantees that the parallelization goes smoothly, as evidenced by the huge barrier wait times.
- Running a GUI on top of the calculations made a relatively small difference to the computation time. The math is what takes a long time and the GUI is cheap in comparison.