Interstellar
Project Report

Lucas Rondenat
Jack Ziesing
Jimmy Cheung
Tallack Graser

Introduction

We have implemented a UFO dogfight. When you open the program you see the initial home screen. Upon clicking anywhere on the screen you are taken to the simulation. UFOs are placed randomly on the screen and shoot at other UFOS while simultaneously trying to avoid the shots of other UFOs. The simulation runs until there is only one UFO left (ideally). We decided to use SFML (simple and fast multimedia library) to implement our graphics.

Design and division of responsibilities

Current project file structure is broken up into subdirectories:

```
├── build
├── docs
├── src
│   ├── main
│   ├── objects
│   └── view
├── styles
│   ├── font
│   └── images
└── test
    ├── Jack_Test
    ├── Jim_Test
    ├── Lucas_Test
    ├── Old
    └── Tallack_Test
```

build: Where makefiles, *.o files and .exe and built

docs: Where documentation can be found

src: Where objects for the game and views are put. It contains game objects (space objects.cpp) in the objects directory, view objects (background.cpp), and the main.cpp in the main directory

styles: Where images and fonts are put for game aesthetics

test: Where each one of our test cases are put

Sequence of Events

main.cpp => start.cpp => game.cpp

The main.cpp will create a start menu for the game. The start menu will continue to draw until a user clicks on the window. Once a user clicks, it will then create a game.cpp which will then draw objects passed on the user button actions. The window will be passed from the start.cpp to game.cpp once the game.cpp is initiated.

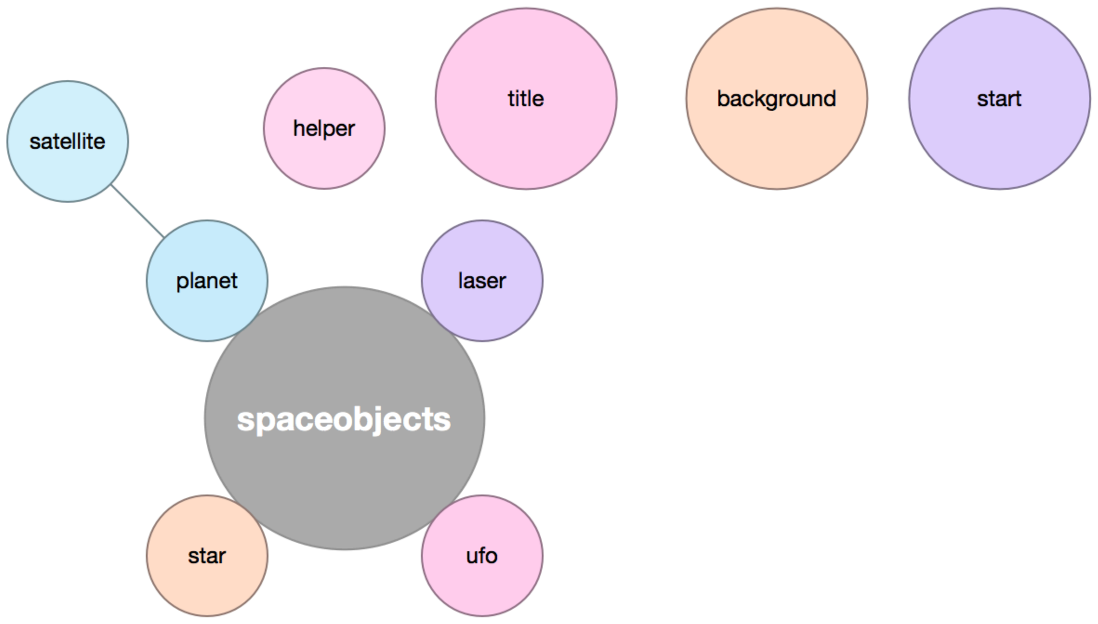
Data Model Design

```
src
├── main
├── objects
│   ├── helper.cpp
│   ├── helper.h
│   ├── planet.cpp
│   ├── planet.h
│   ├── satellite.h
│   ├── spaceobject.cpp
│   └── spaceobject.h
└── view
    ├── background.cpp
    ├── background.h
    ├── title.cpp
```

└── *title.h*

Helper.cpp/.h contains helper functions (random number generator)

planet.cpp/.h contains



*We never defined roles for team members, but rather everyone helped out with what ever needed to get done at a given time.

<u>Development and implementation</u>

Our plan was to write this simulation in serial, run it with Vtune, improve and parallelize our code, then run it with Vtune again. This would allow us to see exactly what could be improved upon, and would give us evidence that parallelism can actually improve the performance of a program!

Before jumping into our program we had to familiarize ourselves with the language and library we were going to use (we actually had to decide on a library). Through some research we chose to go with SFML. We created a few test programs that did simple things (loaded some shapes and moved them arbitrarily). Then we had to familiarize ourselves with sprites and textures and how we were suppose to load them, and how we would be manipulating during the simulation. Once we felt like we understood the library, the built in methods that were, and were not, available to us we were able to begin working on our program.

First Try: We began implementation by looking at our specifications and mapping out what exactly we would need in order to meet these specs. We decided to create a simulator object, which contained UFO objects, cannon objects, and laser objects. About two weeks into development we had a simulator that loaded UFO sprites and moved them across the screen, and a Cannon that would rotate. The graphics were crude but it worked for our purpose. At this point the team decided to scrap what we had and basically start over. We defined completely new objects, decided to get rid of the simulator object, and implemented everything in a clearer and more concise way. At this point everyone had to re-familiarize themselves with how the program was going to be implemented.

Second try: We quickly got back to the same point we were at in our first try. As we were working on the simulation we decided that it would be more interesting it we changed the game a bit. Initially we wanted UFOs to fly across the screen and to have cannons try to shoot them down, but had decided it might be more interest from a parallelization stand point if there were no cannons at all, and instead every UFO could shoot at any other UFO. This then proposed a new challenge for us: how were we going to move the UFOs around in such a way that it showed intelligent thought (that they were trying to avoid getting shot). Through some trial and error we decided that the following algorithm would dictate the movements and actions of our UFOs:

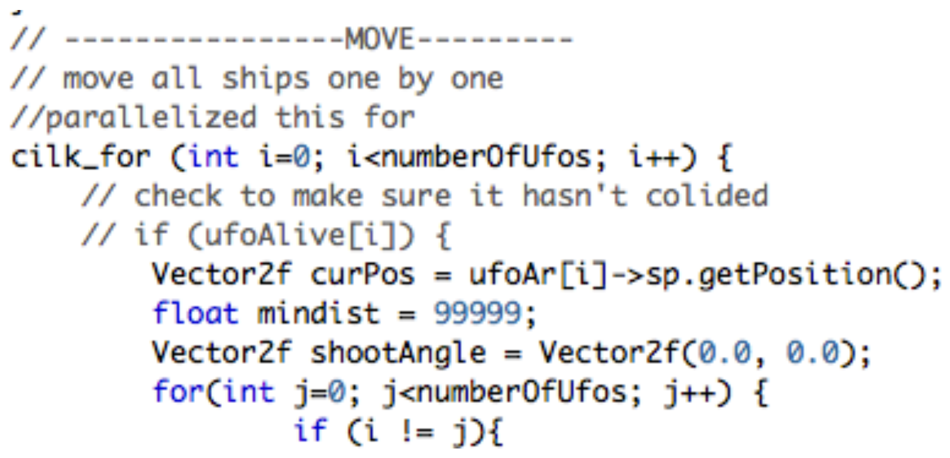
1) UFO shoots at the nearest object (by computing the angle)
2) UFO then moves away from the nearest object (perpendicular)
3) Check for collisions (UFOs that have been shot)

Once we got our code running in serial we began to go back and rework our code to run in parallel.

Parallelism in our code

Map:

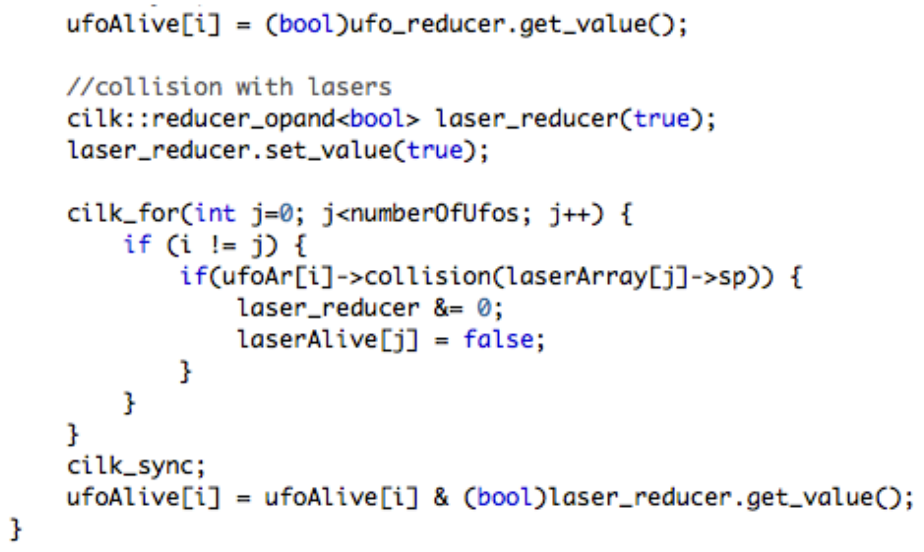
We use map to gain better performance on loops without dependencies. When playing around with the map pattern it seems that sometimes the map in-fact hurts the program. On the first iteration of parallelizing our program we ran into a few errors. We found more dependencies existed in our code than we thought. This caused us to sit back and rethink our parallelization.



```
// ----------------MOVE---------
// move all ships one by one
//parallelized this for
cilk_for (int i=0; i<numberOfUfos; i++) {
    // check to make sure it hasn't colided
    // if (ufoAlive[i]) {
        Vector2f curPos = ufoAr[i]->sp.getPosition();
        float mindist = 99999;
        Vector2f shootAngle = Vector2f(0.0, 0.0);
        for(int j=0; j<numberOfUfos; j++) {
                if (i != j){
```

Map/reduce:

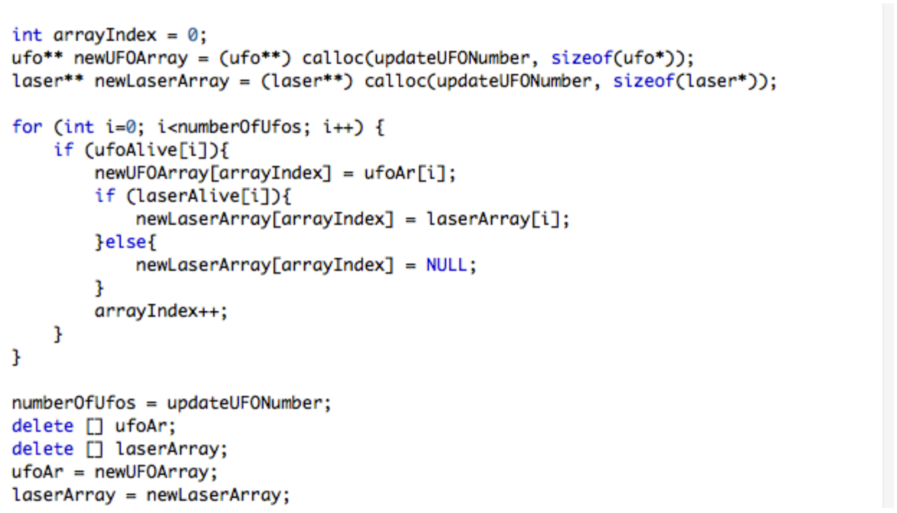
Our map reduce pattern is used when we change Boolean values in our array's that keep track of what objects are still alive.



```
        ufoAlive[i] = (bool)ufo_reducer.get_value();

        //collision with lasers
        cilk::reducer_opand<bool> laser_reducer(true);
        laser_reducer.set_value(true);

        cilk_for(int j=0; j<numberOfUfos; j++) {
            if (i != j) {
                if(ufoAr[i]->collision(laserArray[j]->sp)) {
                    laser_reducer &= 0;
                    laserAlive[j] = false;
                }
            }
        }
        cilk_sync;
        ufoAlive[i] = ufoAlive[i] & (bool)laser_reducer.get_value();
    }
```

Gather:

As our simulation continues lasers and UFOs are either created or deleted. In our early implementation of the simulator we used a Boolean array to keep track of the objects alive without deleting an object from an array when a collision occurs. This created buggy code that ended up looping though unnecessary elements finding objects that should be deleted and accounting for them. We were able to gain performance and practice a data reorganization by simply reorganizing the array into a new smaller array that is arranged slightly better.



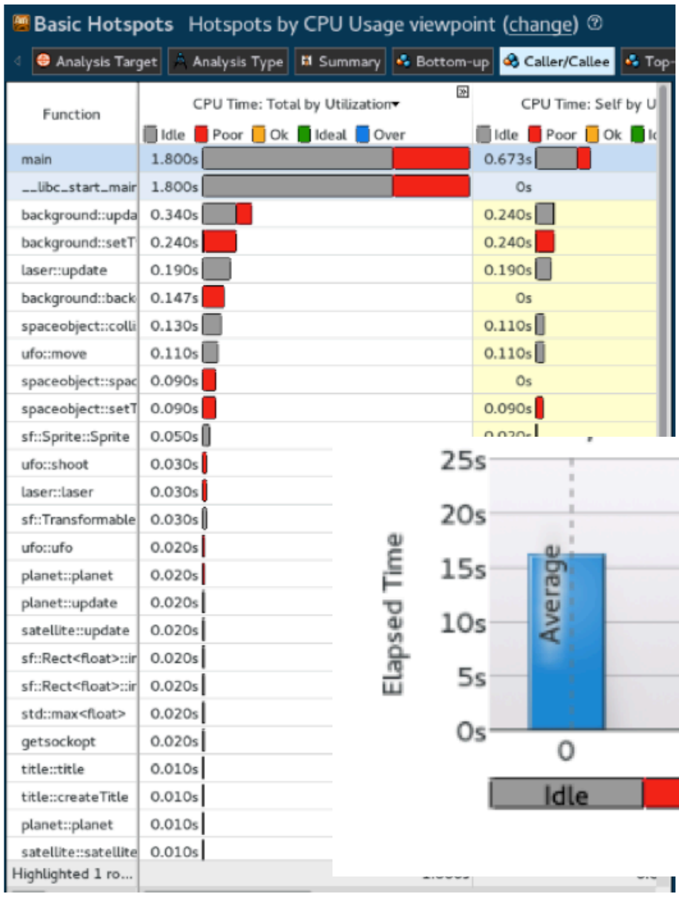
```
int arrayIndex = 0;
ufo** newUFOArray = (ufo**) calloc(updateUFONumber, sizeof(ufo*));
laser** newLaserArray = (laser**) calloc(updateUFONumber, sizeof(laser*));

for (int i=0; i<numberOfUfos; i++) {
    if (ufoAlive[i]){
        newUFOArray[arrayIndex] = ufoAr[i];
        if (laserAlive[i]){
            newLaserArray[arrayIndex] = laserArray[i];
        }else{
            newLaserArray[arrayIndex] = NULL;
        }
        arrayIndex++;
    }
}

numberOfUfos = updateUFONumber;
delete [] ufoAr;
delete [] laserArray;
ufoAr = newUFOArray;
laserArray = newLaserArray;
```

Data Reorganization:

In addition to our gather we reorganization our array of ufo objects based on distance when we setup our program. This will slightly help a few computations when we check distances of objects for movements and laser firing. It is hard to say if this data reorganization is even helpful because we end up loosing the organization as the simulation plays out. Also we still look at the distances of each element so the data reorganization doesn't really save us here.

<u>Experiments and performance</u>

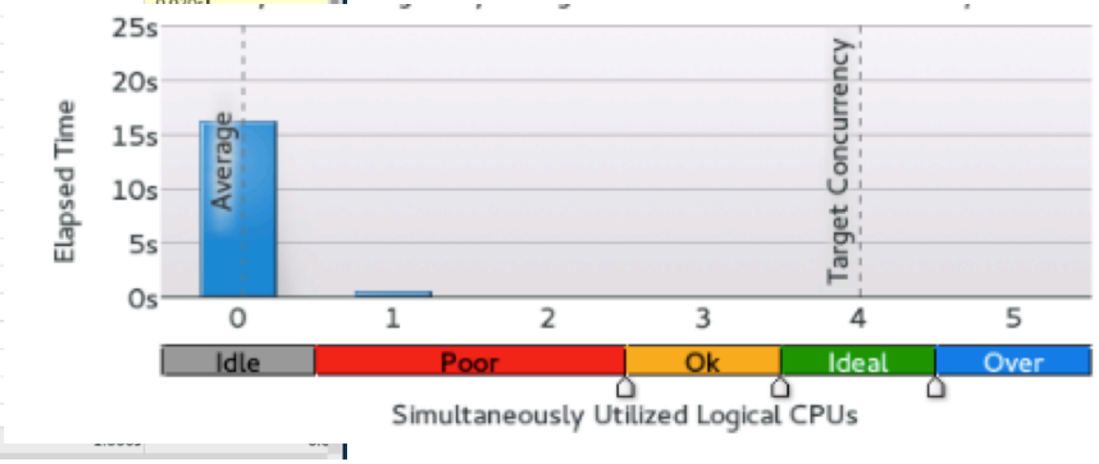
As mentioned above, we first wrote our program in serial then "upgraded" it to parallel after using Vtune. When we looked at our serial code we really didn't see anything too alarming. We concluded that this was probably a good thing, but it didn't give us much direction of where to improve our code for efficiency.



Basic Hotspots Hotspots by CPU Usage viewpoint (change)

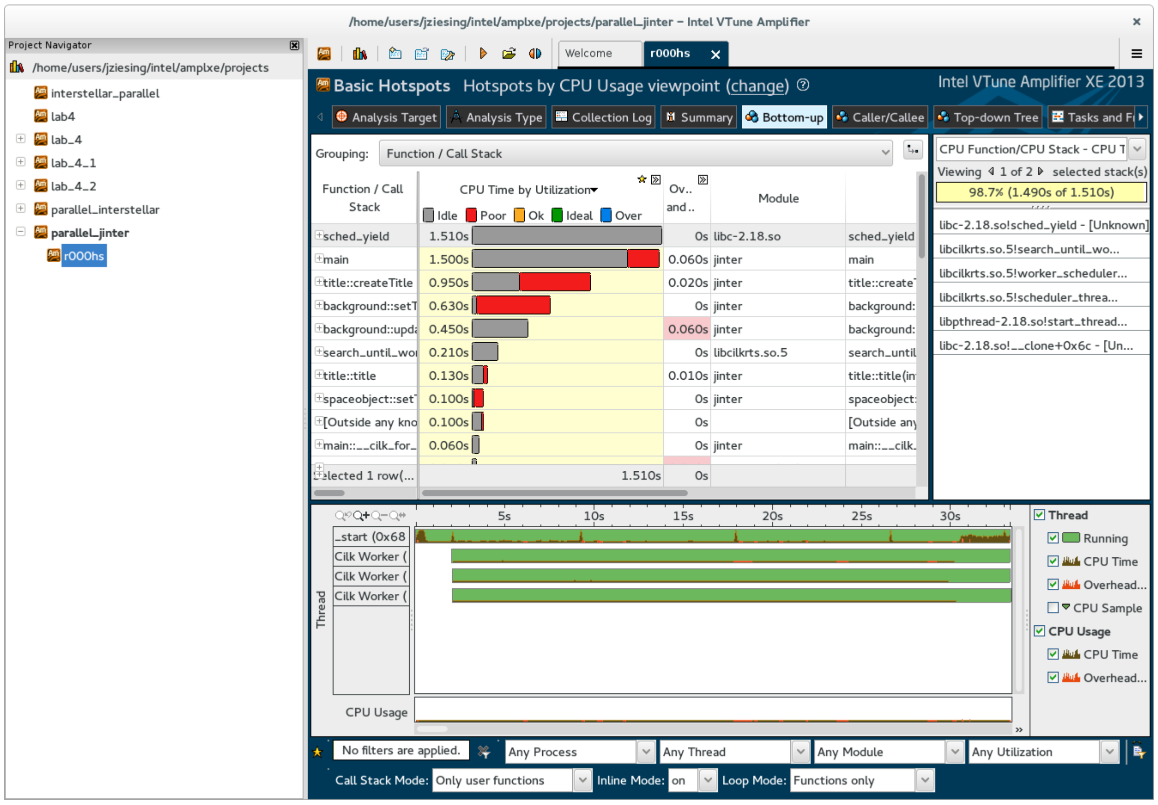
Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-

Function	CPU Time: Total by Utilization	CPU Time: Self by U
	▨ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over	▨ Idle 🟥 Poor 🟧 Ok 🟩 Id
main	1.800s	0.673s
__libc_start_mair	1.800s	0s
background::upda	0.340s	0.240s
background::setT	0.240s	0.240s
laser::update	0.190s	0.190s
background::back	0.147s	0s
spaceobject::colli	0.130s	0.110s
ufo::move	0.110s	0.110s
spaceobject::spac	0.090s	0s
spaceobject::setT	0.090s	0.090s
sf::Sprite::Sprite	0.050s	
ufo::shoot	0.030s	
laser::laser	0.030s	
sf::Transformable	0.030s	
ufo::ufo	0.020s	
planet::planet	0.020s	
planet::update	0.020s	
satellite::update	0.020s	
sf::Rect<float>::ir	0.020s	
sf::Rect<float>::ir	0.020s	
std::max<float>	0.020s	
getsockopt	0.020s	
title::title	0.010s	
title::createTitle	0.010s	
planet::planet	0.010s	
satellite::satellite	0.010s	
Highlighted 1 ro...		



After thinking about how we could improve our code form here, we thought we should try to find the number of UFOs that it took to make our serial code noticeable slower, or crash completely. Through experimenting we found than any more than 200 UFOs our program started to visually seem stressed (UFO rendering slow/jittery). Our plan was to then improve the number of UFOs we can use in our simulation through parallelization. However, due to the way the image drawing/rendering works with SFML/openGL we realized we would not be able to improve the number of acceptable UFOs for our simulation through parallelization.

After reworking our program and implementing stuff in parallel we ran our program again with Vtune. By looking at the following graph, you can see that we improved our program by limiting the red (the poor performance areas).



Grouping: Function / Call Stack

Function / Call Stack	CPU Time by Utilization (🟫 Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)	Ov.. and..	Module	
sched_yield	1.510s	0s	libc-2.18.so	sched_yield
main	1.500s	0.060s	jinter	main
title::createTitle	0.950s	0.020s	jinter	title::create
background::setT	0.630s	0s	jinter	background:
background::upd	0.450s	0.060s	jinter	background:
search_until_wo	0.210s	0s	libcilkrts.so.5	search_until
title::title	0.130s	0.010s	jinter	title::title(in
spaceobject::set	0.100s	0s	jinter	spaceobject:
[Outside any kno	0.100s	0s		[Outside any
main::__cilk_for_	0.060s	0s	jinter	main::__cilk_
elected 1 row(...	1.510s	0s		

CPU Function/CPU Stack - CPU T

Viewing 1 of 2 selected stack(s)

98.7% (1.490s of 1.510s)

- libc-2.18.so!sched_yield - [Unknown]
- libcilkrts.so.5!search_until_wo...
- libcilkrts.so.5!worker_scheduler...
- libcilkrts.so.5!scheduler_threa...
- libpthread-2.18.so!start_thread...
- libc-2.18.so!__clone+0x6c - [Un...

No filters are applied. Any Process | Any Thread | Any Module | Any Utilization

Call Stack Mode: Only user functions Inline Mode: on Loop Mode: Functions only

<u>Lessons Learned</u>

Parallelism is hard.

For how easy it is to understand most of the parallel patterns, we found it very difficult to actually implement them. One thing we naively found was that we couldn't parallelize loops that create new objects on the heap.

Projects are easier if you make your deadlines.

As mentioned earlier, we rarely hit the deadlines we made for ourselves, which made it very difficult to work effectively as a team.

Caffeine helps your allergies.

Did you know caffeine helps relieve allergy symptoms? Neither did we until today.

Do research before programming

SFML and OpenGl have dependencies with different operating systems- Macs require the window to be in the main thread, you cant draw and move the object at the same time. Games are serial by nature defined by key events to actions to drawing.

Communication is key

We found that the biggest struggles we encountered while working on this project could have been alleviated, for the most part, by good and effective communication. More often than not, when we ran into an issue, it was either due to someone else's change in the code, or was an issue that someone had already ran into and had found a work around.

<u>**Conclusions**</u>

Parallelism isn't always faster

After we first implemented our program in serial we realized that parallelizing our code wouldn't improve our performance that much (again, this is due to the nature of SFML/openGL and how all the drawing must be done in one thread).

Our program didn't really need to be parallelized

-Transferring from serial to parallel is hard, and wasn't exactly necessary for our program.

SFML/OpenGL was not the best library to use

In the broad picture we may have picked a weak topic to parallelize. Parallel programming is interesting because our initial object oriented design prevented us from lots of parallelism we though we could implement. It seems that a more functional/script might be easier to parallelize. For example when we check for collisions we run into dependency issues that prevent our parallelization. Also sometimes when we do computations on all our lasers and ufos we do parallelization that probably doesn't improve our performance.

One way we could make our program better is add a stencil pattern to detect objects. Currently we have one data structure for all our ufos and another for all our lasers and then we iterate through all of them to check nearby objects. This seems like something that could easily be stenciled if we had a better program structure. I think we made parallelism harder with our architecture which really prevented us from experimenting with a stencil.

Two thread-implementation with MPI

In reflection of finishing the program we decided that if we were to do it again we would attack the program with a two threaded MPI approach where all the drawing was being done on one thread. We would also implement a physics package that allowed for more complicated movement.

Serial AI vs parallel AI

When we first implemented our AI in serial, every UFO finds the closest object and moves perpendicularly away from it. However, when we parallelized this algorithm we got a small change in how the algorithm works. This is due to the fact that all movements are being computed concurrently. This means that each UFO is not guaranteed to accurately identify the closest object. (as one of the objects could have been moved, but a different UFO might still be looking at its old location). Although this gives indeterminist behavior in our AI it doesn't really negatively effect our program, or the efficiency of the AI.