The project involves a significant number of computations, which require a lot of processing time. The main problem we have struggled with is minimizing the amount of time it takes for everything to complete. A decent amount of time was reduced through standard optimization means, however the most important step is implementing multithreading or multiprocessing. The initial manual process given to us by the client describes a step which is effectively manual multiprocessing, the client opened multiple command terminals, and continuously gave them small sets of queries.

The two main options we looked at were multiprocessing and multithreading. Multiprocessing involves running multiple processes or programs at the same time. Multithreading splits a single program into multiple smaller processes, threads, and running those simultaneously. Initially, we thought that multiprocessing would be the optimal solution as it is most similar to what the client was doing in his process, however, in practice it was not effective. Because our program is a GUI based one, when the program was split up for multiprocessing it created a new interface popup for each process. This of course was impractical from a usability perspective, as well not working correctly with the actual functionality. This left threading as the best option.

We implemented multithreading using the powerful Pathos library. Pathos builds upon the standard python multithreading capabilities, increasing their effectiveness in many ways, and fixing many problems that exist. The implementation itself is simple:

```
p = pools.ThreadPool()
p.amap(lambda point: proc(point), test)
p.close()
p.join()
```

The first line creates a thread pool. This is a pool of worker threads that can be called upon to do tasks. The next line is calling the worker threads to do a task, in this case it is calling the processing of gridpoints to generate isochrons. This functions similarly to a loop, however instead of going through the data (in this case "test") one by one, it gives each worker a piece of data to process, when that worker completes the task, it is given more data. This allows. These threads can work in parallel, increasing the efficiency of program significantly. The next two lines complete the thread process and destroy the worker pool. This can be replicated for any part of the program which processes multiple points of data.

The above example of threading is not fully functional in the program. We ran into an issue with part of the processing where it writes to an external file to temporarily store data. The process could not progress beyond that point, our assumption is that it locked up as multiple threads attempted to access and write to the same temporary document. This problem can be solved using a lock, however we have not had the time to properly research and implement this.

Initially, we assumed that the overlap calculation would be the most important part of the application to multithreaded as it was the longest and most intensive part of the client's initial process, and the one that was manually multi-processed. However, once the database was optimized and we added the calculation to the app, it turned out to already be a fast process. Even so, it may be important to add threading to this, as larger datasets or slower machines may take longer than our testing. This implementation should be simple, as it only requires replicating the above described code and changing the values in the amap call, as well as changing the loop that runs the query to a function with the data as a parameter.