Implementing EpiSimdemics in Haskell

Collaborators: Andrew Schreiber ajs2409

December 20, 2021

1 Problem Statement

In the paper "EpiSimdemics: an Efficient Algorithm for Simulating the Spread of Infectious Disease over Large Realistic Social Networks" [1] the authors describe an algorithm for predicting the spread of a virus through a given population, given a series of interactions between the members of the population. Traditional disease models work by simulating each event (a group of people, some of whom are infectious, at a location for a given period of time), in serial, constantly updating each individuals health status after each interaction. This means to compute the model, you either implement a global clock and process each event in order, or build a dependency graph of all the events and parse this graph as you compute the outcomes. Both of these models will be hard to parallelize and expensive to compute.

The novel idea in EpiSimdemics relies on a fact that we have seen play out throughout the Covid pandemic: diseases have a minimum latency period, D_{min}. Latency in this case refers to a period of time between when an individual is infected and when they are infectious (can infect other people). What this means from an algorithm perspective though, is that if you are at time t in your simulation, no one that is not already infected can become infectious until time $t + D_{min}$. This means that all the events in $(t, t + D_{min})$ are independent of each other, and hence can be processed potentially out of order, and in parallel. EpiSimdemics does just that, it works by iterating in intervals of $\Delta t < D_{min}$, and for each iteration, process all the events in parallel. Once that is complete, update each individuals health status (which can also be done in parallel), and then start the next iteration.

The goal of this project is to implement (a slightly simplified) version of the algorithm described in the paper in Haskell, and to be able to show the performance benefit gained by enabling parallelism in the algorithm.

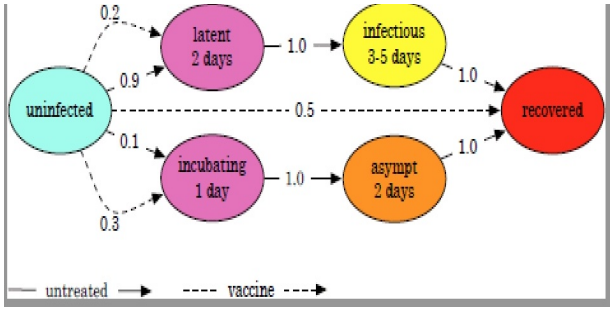
The final code for the project can be found in this github repo.

2 Explanation of Algorithm

There are three main steps in the algorithm: transitioning each individuals health state according to a predecided disease model, computing the outcome of each event in the specified time window, and updating each individuals state based on the outcome of each event. As mentioned above, the algorithm works by iterating over intervals of Δt and within each interval performing the steps described in the below sections.

2.a Disease Model Health State Transitioning

When predicting the spread of a disease, there needs to be a model of how the disease progresses through an individual. In a simple model, the basic states are uninfected, infected and latent, infectious, and recovered. See the below model for an example that has some probabilistic behavior:



As we have previously discussed, EpiSimdemics relies heavily on the fact that the infected and latent period, lasts for a non trivial amount of time, usually 1-2 days. The first step per time period is to update each person, to make sure they are in the appropriate health state for the next time period. I.E. if they have been infected and latent for the required number of days, they become infectious, and so on. The formal algorithm describes how someone can transition states part way through a time period, and in that case you split them up into two people, one for the time period where they are in the first state, and one for the second. To simplify the implementation for this project, I will not do this, and assume that people can only transition health states at the start of a new time period. I will use a simple model that roughly resembles Covid, using a 3 day latency period, and a 10 day infectious period.

2.b Event Outcome Computations

An event is a group of people in a particular location for a particular period of time. In order to compute the likelihood that a person, p_i, gets infected we first need define a few variables. Let τ be the length of the event, and N be the number of infectious people at the location. To resemble Covid, we will use τ in units of 15 minutes. Let r_j be the infectivity of the disease for person j, this is a percentage of how easy it is for a given person to spread the disease. To simplify the algorithm, we will assume infectivity is a constant, r. Let s_i be the susceptibility of person i, this is a percentage representing how susceptible person i is to getting infected. Let ρ be the transmissibility of the disease, which is how well the disease spreads from person to person. Some diseases spread in the air, like Covid, and are highly transmissible, while other diseases are less so. Putting all of this together we will use the following equation for determining the probability that person i gets infected with the disease, given they are in a room with N people for time τ:

$$p_i = 1 - \exp^{\tau N \ln(1 - r s_i \rho)}$$

What this equation looks like is for a fixed $y = \tau N$, if $y = 1$ (in a room with 1 person for 15 minutes), then as $r s_i \rho$ goes from 0 to 1, the percentage chance of getting infected increase linearly. For y between 0 and 1, the growth is exponential, meaning that you have a sub linear chance of getting the disease. As y increases past 1, the curve becomes logarithmic, steepening as y gets large, increasing your changes substantially of getting infected, even with low $r s_i \rho$.

This step of the algorithm will process each event, compute this equation for each person in the event, simulate the experiment to see if the person ended up getting infected, and store the outcome of that to a list of event of outcomes for each person.

2.c Post Events Per Person Processing

The post processing step is the most straightforward step. Given person *i* and a list of outcomes of the events for the person (which are 1 for newly infected, and 0 for not infected) just 'logical and' the results together and update the persons health state.

3 Dataset Generation

Although in the papers associated with EpiSimdemics they describe performance stats on particular datasets that they generated, they do not publish these datasets. This meant I had to generate the dataset for the simulation. What I did was pick a fixed amount of people (I used 100, 1000, 10000, and 100000) and a fixed number of locations (usually the number of people divided by 10, but more on this later), and then for each person per day, pick a random number of events (uniformly distributed from 0-5) and randomly distribute each event to a location. Each location per person then has a time they arrive and a time they leave, for this I randomly assigned a length of time, guaranteeing that no person is in two places at once. On top of this, I also assigned each event a random number between 0 and 1. What this means is that I can use this in the probability calculation, so that way the Haskell program is stateless and there are no discrepancies in performance stats due to odd probabilistic events. So in summary a row in the dataset looks like:

Day	Person	Location	StartTime	FinishTime	Chance
0	1	14	370	455	0.68

This means that on Day 0, Person 1 when to Location 14, from time 370 to 455, and for them to get infected in the event the disease model needs to predict a greater than 68 percent chance they got infected. Finally there is one extra dataset, People, that gives the initial state of each person, i.e. whether they are healthy or unhealthy. I chose to start with a small initially infected population of 2 people. On the main testing dataset I used, in 60 days, the disease had spread from those 2 people to 2230 people in total out of 10000.

4 Implementations

In this section I will talk through the different implementations of the algorithm, the performance of each version, and any pitfalls that each one might have had. All of the versions have the same main steps in common:

1. Read the data from the People (initial states) file and the Events file, and parse them into data structures.
2. For each day for 60 days process the events for that day according to the algorithm discussed above.
3. Summarize the results by seeing how many people got infected.

For the parallelism, I only focused on the second step, which is what the algorithm focuses on, i.e. how can we distribute the event result calculations across multiple cores. I will call step two the "core function" below. For each algorithm I will use the dataset with 10000 people and 100 locations as a reference to compare performance across them. Each algorithm was run on my local computer with is a 2013 Macbook pro with 4GB of memory and 2 CPUs, which are hyperthreaded into 4 logical CPUs. When running with parallelism I used the N4 argument.

For this dataset it takes about 25 seconds to read the data, so this will be worth keeping in mind for Amdahl's Law as we do performance calculations below.

4.a First Implementation

The first implementation of the core function was focused on simplicity and without explicit parallelism in mind. Each version in this implementation takes three arguments: an array of people types, a Map of Maps from Day to Location to an array of event types, and a day number. Each version produces an array of people in the updated state after the day has been processed.

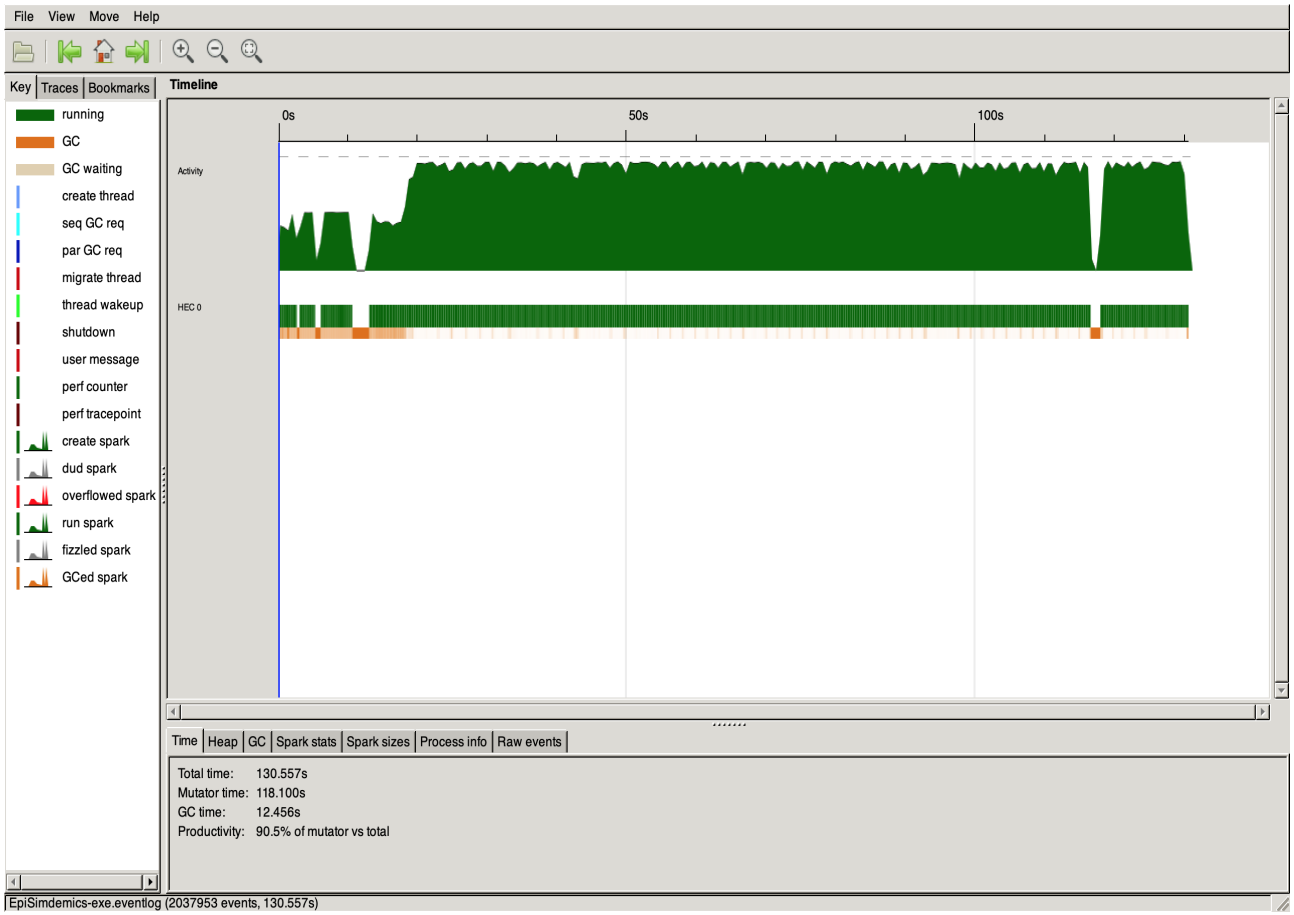
4.a.1 Basic Non Parallel Version

The basic steps of this implementation are as follows:

1. Transition each persons health state in the array using a function applied via a map function.
2. Create a map data structure of person id to their health state. This is partially done using a map function.
3. Generate the list of events per day by folding all the events for each location for each day.
4. For each event, generate the outcome of that event (whether the person got sick or not), by applying the probability function discussed above. This is also done using a map function.
5. Create a Map type to map each person to the list of event results for that person.
6. For each person, process the event results, by seeing if they got infected in any of the events. This is also done using a map function.

This is the relatively straight forward way to implement the algorithm described in the paper in a non parallel way.

For performance this algorithm averaged 130 seconds to process the dataset. This 130 seconds will serve as our initial baseline for improvements. You can see the Threadscope output here:



The Threadscope shows relatively consistent performance with a few short garbage collects.

Above we noted that it takes about 25 seconds to read the data, and hence this part is non parallel. This is about 19 percent of the total time. Hence using Amdahl's Law we see:

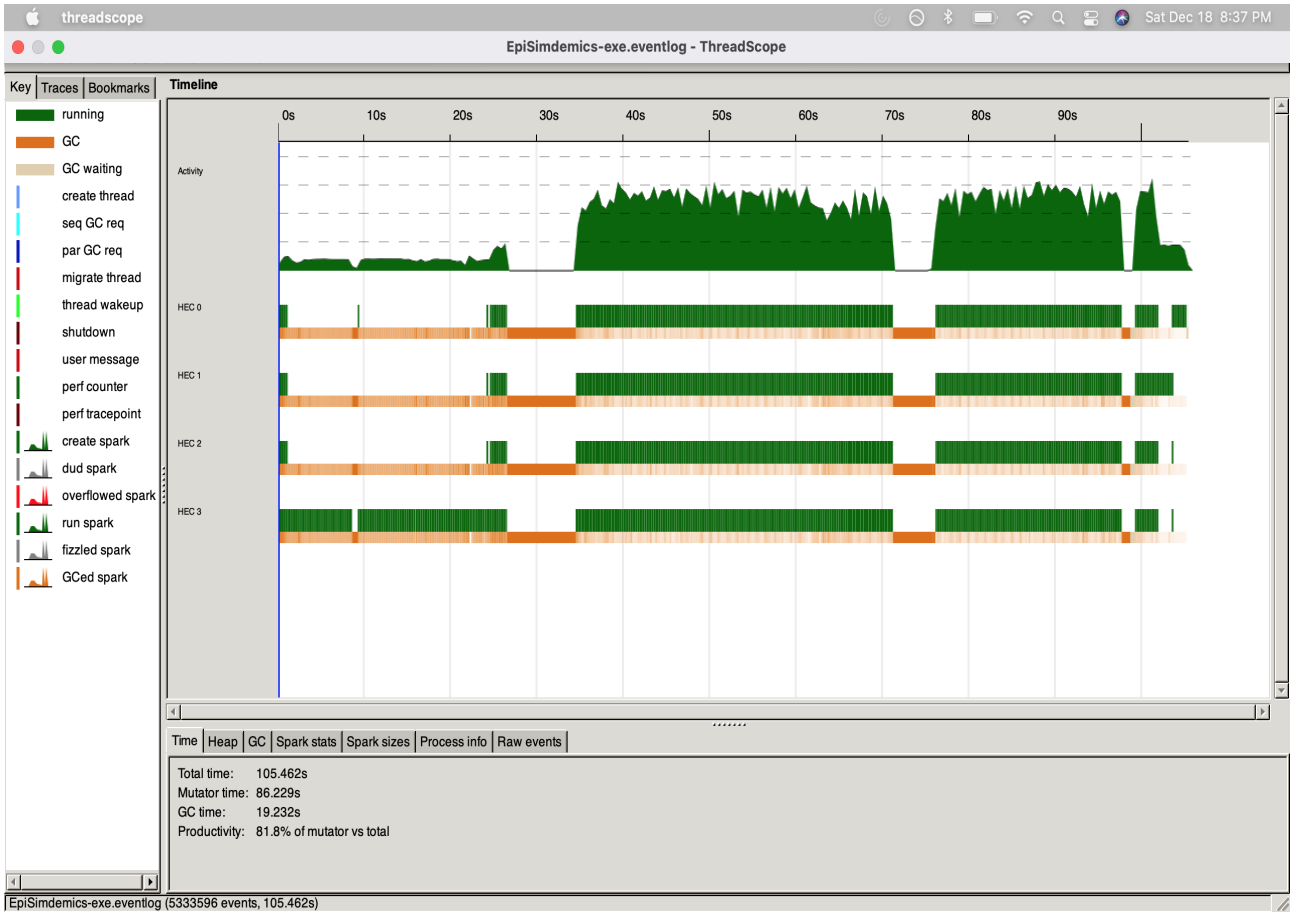
$$S = \frac{1}{(1-P) + \frac{P}{N}} = \frac{1}{.19 + \frac{.81}{4}} = 2.54$$

In summary, given 4 CPUs, the best we could hope for is 2.54 times performance improvement would be 51 seconds.

4.a.2 Basic Implementation With Parlist

This next implementation was just to naively add "`using` parList rseq" after the 4 map statements listed in the above breakdown.

This took it down to 105 Seconds, a 20 percent reduction. This is good, but not as good as I would have hoped or Amdahl's Law would have predicted.

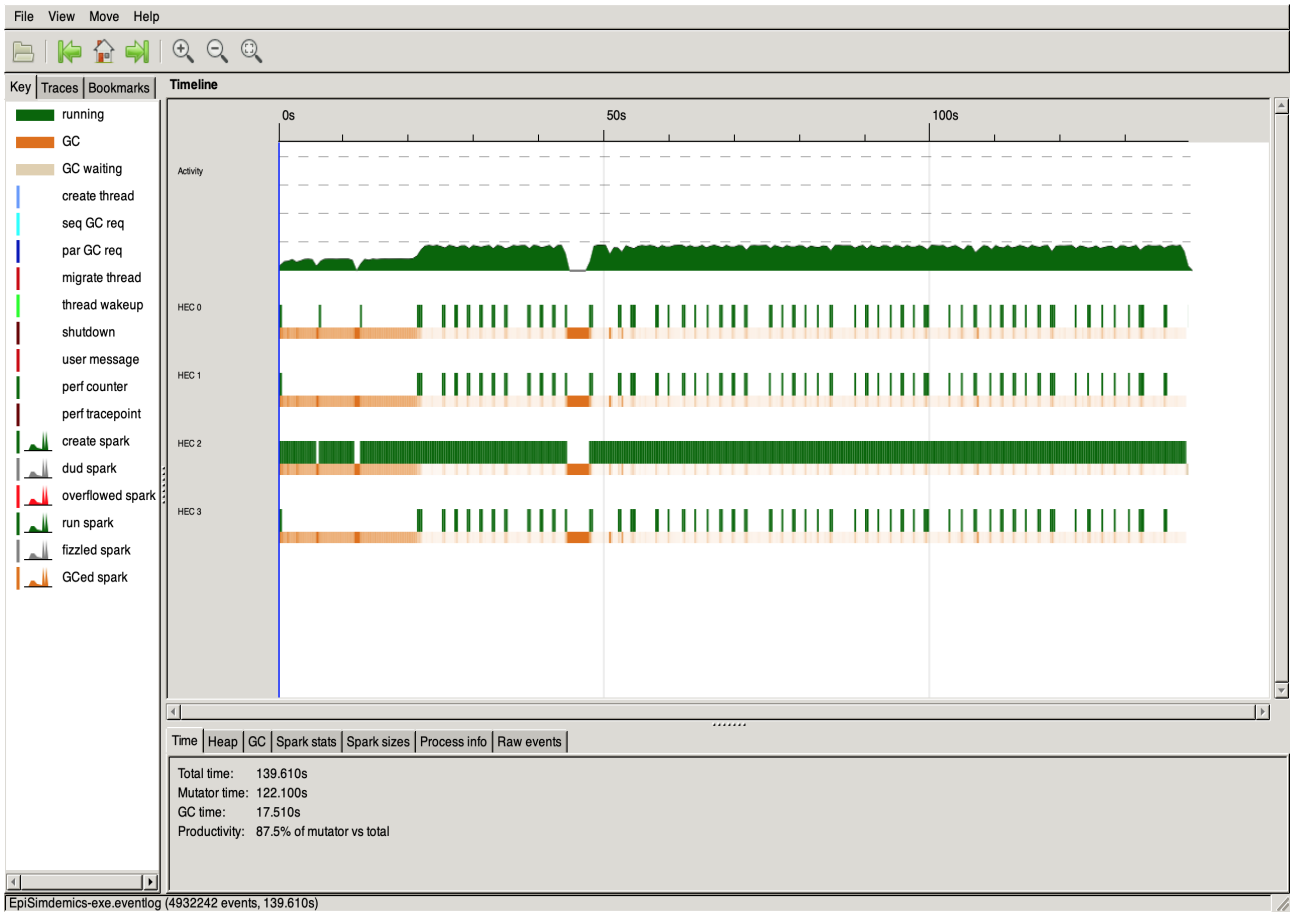


Taking a look at the Threadscope we see we have few larger garbage collects, and although each core is kept busy they are not fully busy, suggesting we are probably spending a bunch of time creating sparks and not enough time doing valuable work.

4.a.3 Basic Implementation With Chunks and Parlist

Based on this, the idea here was to break the work up into chunks so that we could spend less time creating sparks and more time doing valuable work. For this I broke the work into chunk sizes of 40, there is a discussion on why 40 below.

In the end this new implementation took 139 Seconds, more time then the non parallel implementation! How could things get worse? Looking at the Threadscope:



We see that we are barely utilizing the other cores and most of the time is just spent on the main core. After reviewing the code it became apparent that it was spending most of its time going back and forth between chunked and non chunked datasets, and not actually doing valuable work. This then gave rise to the idea that algorithm could be refactored to more gracefully handle the chunking.

4.b Refactored Implementation

The goal of the refactor was two fold: first, to reduce the amount of non parallel calculations that need to be done in each step, and second, to make it easier to have a chunked implementation of the algorithm. Some of the main inefficiencies with the first algorithm were:

1. Creating a map from Person id to health state each iteration
2. Folding events per location into events per day each iteration
3. Adding all event results (positive and negative) into a map
4. Processing event results for individuals who are already infected and it doesn't matter for

The refactored implementation eliminated all of these inefficiencies, by reordering some of the operations, and also by keeping two maps, one the original map of maps for all events and all locations, and one that just had the events for the given day, this way we do not need the fold every iteration. One example is previously we created the map from person id to health state, because the event simulation required to know each persons health state (already

infected people cant get sick). Instead I changed it to not require that, and then later in the final step only update a persons health status if they are not already sick. This removed one map statement from the algorithm.

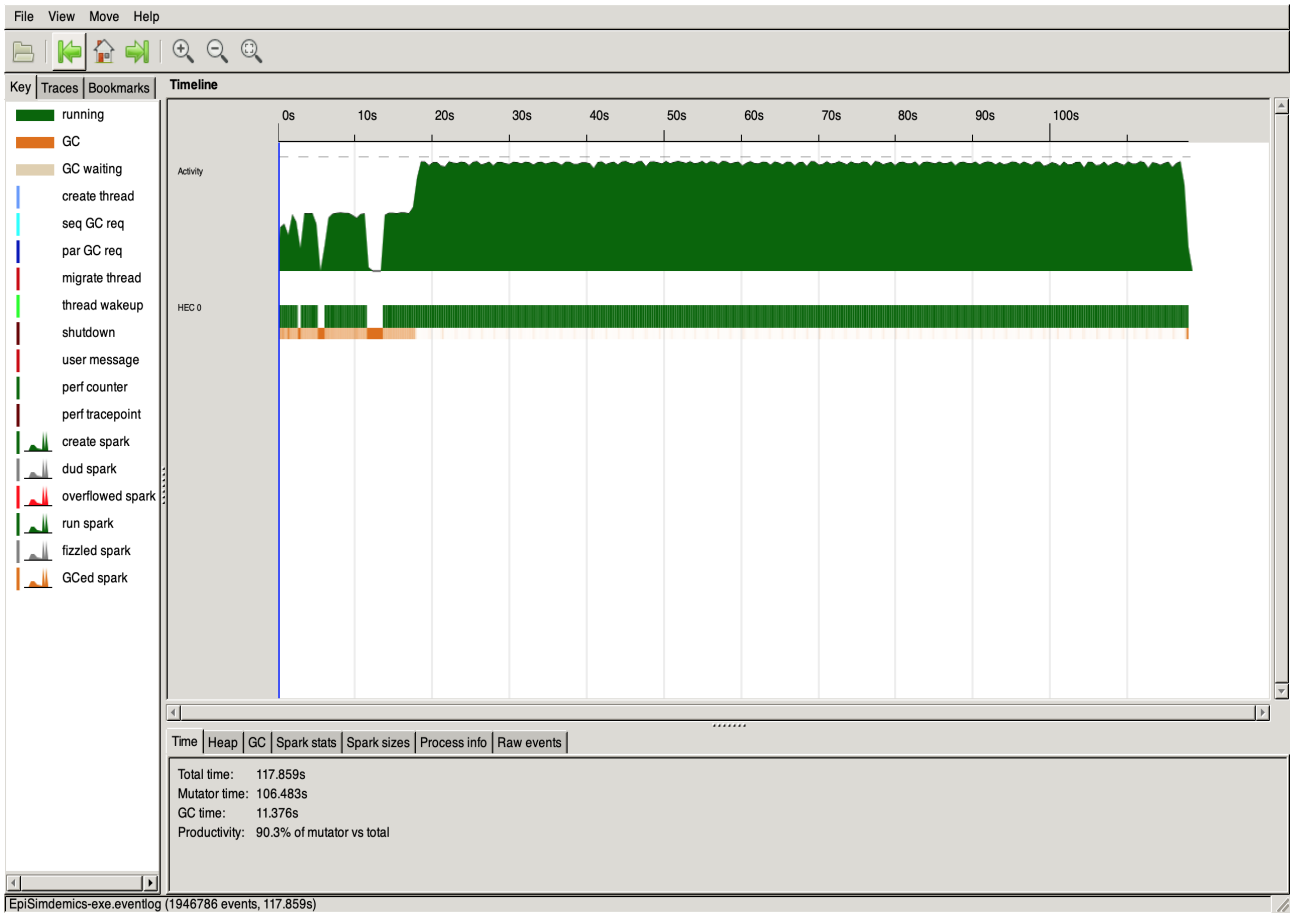
4.b.1 Refactored Version Non Parallel

The refactor did eliminate some inefficiencies so it makes sense to set a new baseline for performance. Running the refactored version without parallelism resulted in a run time of 117 seconds, a 10 percent reduction in the original runtime. Keeping the 25 second file loading time, we see using Amdahl's Law:

$$S = \frac{1}{(1-P) + \frac{P}{N}} = \frac{1}{.214 + \frac{.786}{4}} = 2.43$$

So we see in a theoretically perfect parallel implementation we would see a 2.43 times improvement given 4 cores, resulting in a run time of 48.1 seconds.

For reference we can see the Threadscope output here:

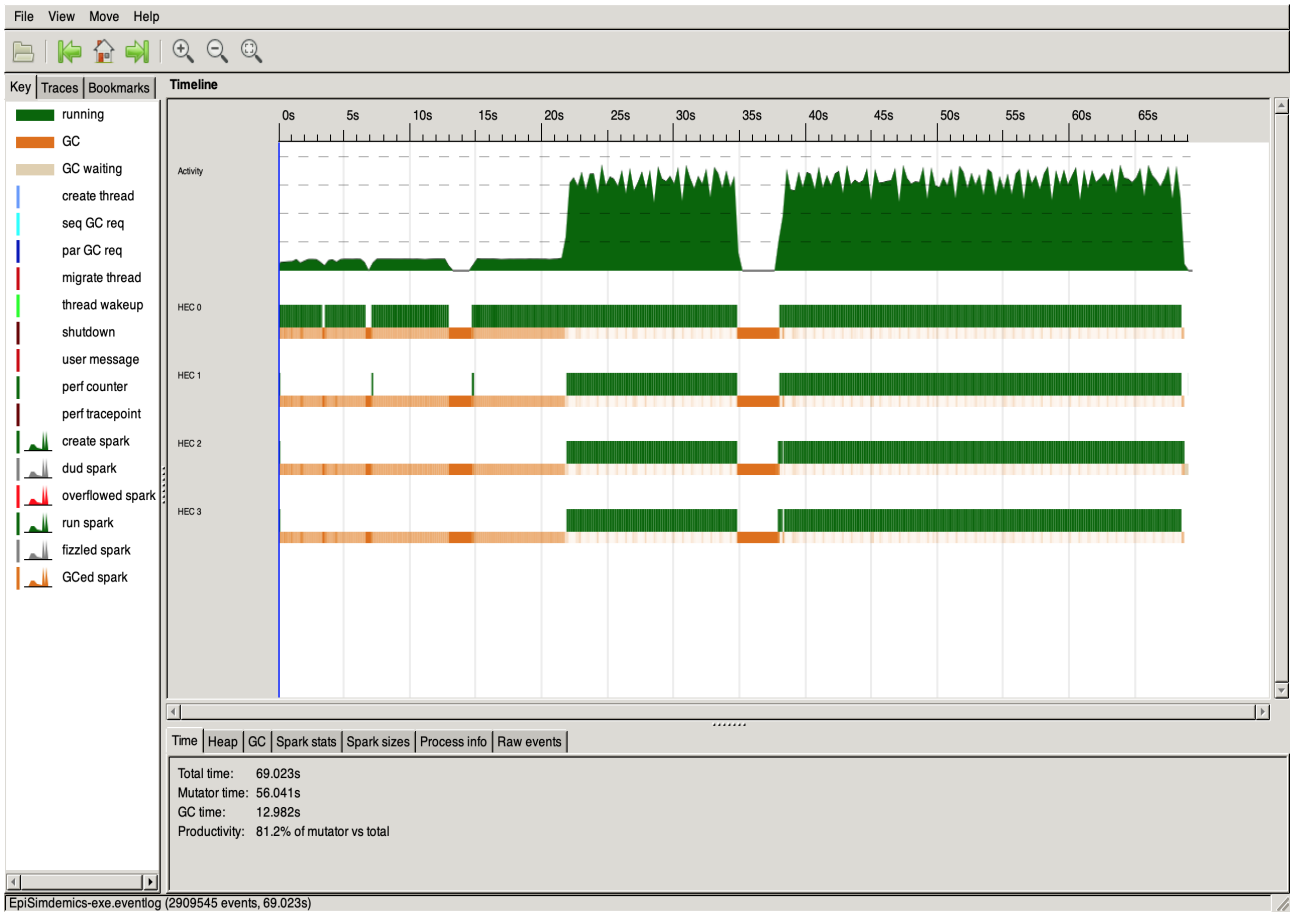


4.b.2 Refactored Version with Chunks and Parlist

Finally, we get to the refactored version where the work is broken up into chunks of 40 (40 people per group, 40 events per group), and we use parlist to distribute the chunks across multiple cores. One of the key changes here is that the core function in this implementation takes a list of lists of people instead of a list of people. This way the core function does not need to chunk and dechunk the list of people at each step, reducing the total amount of work

done. Also because we processing chunks of events instead of individual events, it was much easier for the work to filter out non positive event results to happen in the distributing setting, instead of in the non-distributed aggregation phase.

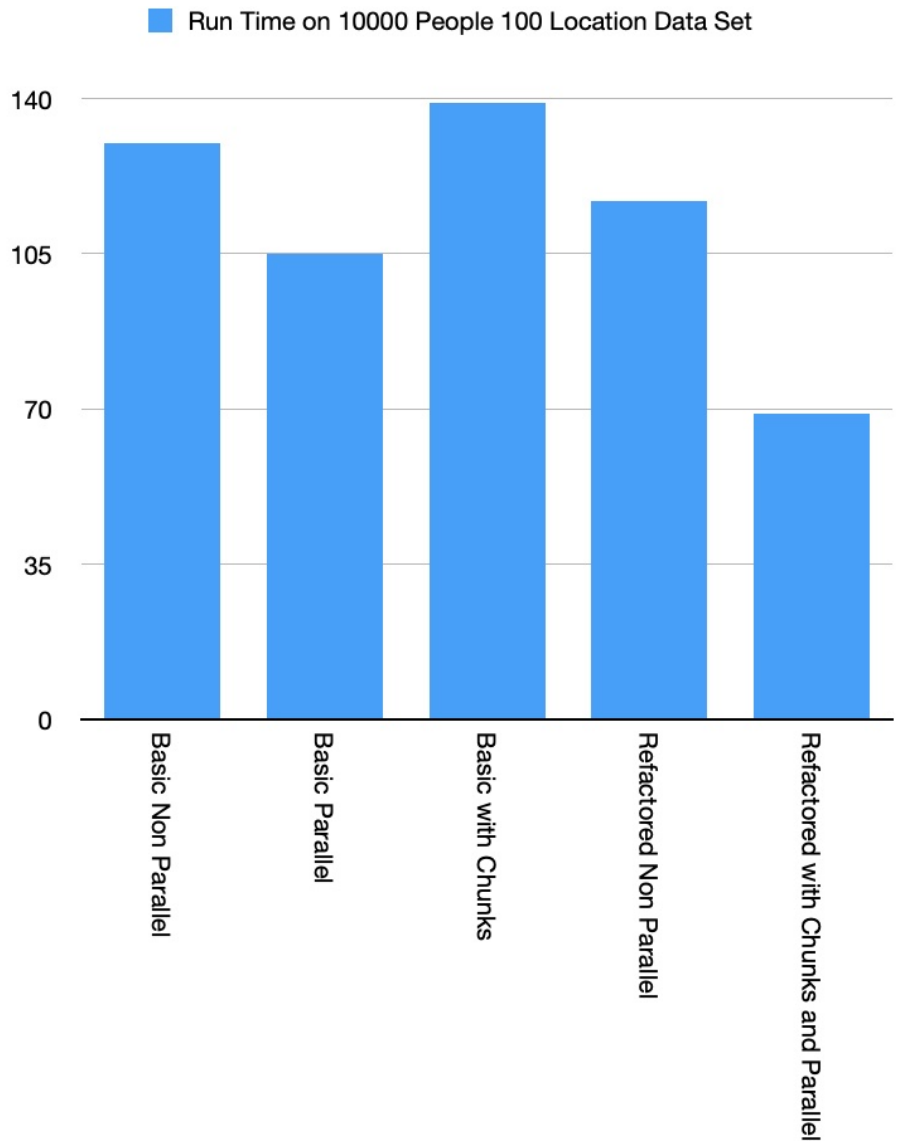
All of these changes resulted in a net runtime of 69 seconds, a net improvement 1.7 times. Unfortunately this does not get particularly close the 2.43 times improvement we would hope to see in a perfect implementation, but it is still a significant improvement. As mentioned above however, my machine is only hyperthreaded to 4 cores which does not actually provide the same performance as having 4 separate CPUs. So potentially a lot of the performance gap is coming from that discrepancy.



5 Performance Summary

5.a Algorithm Overall Performance

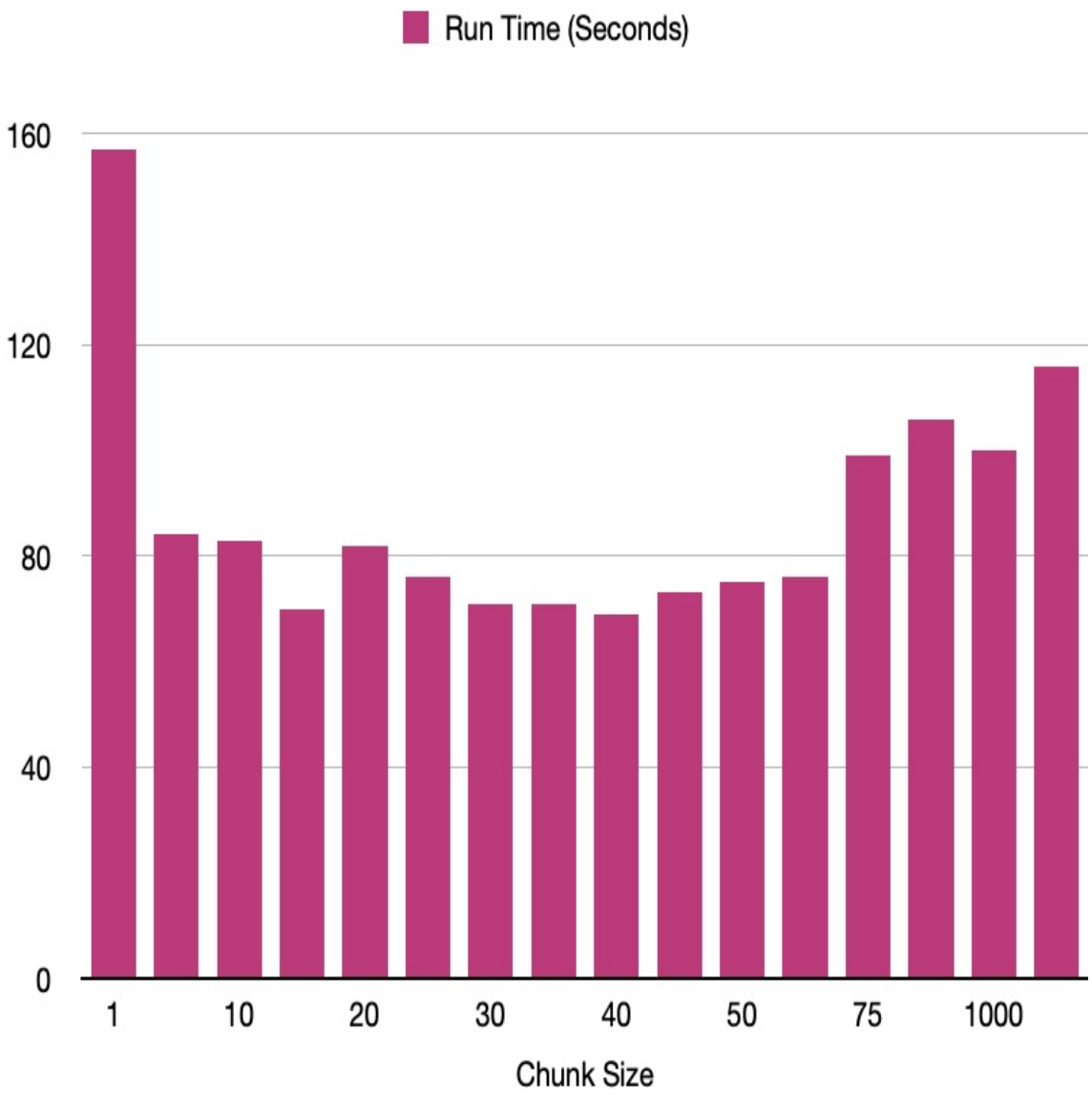
Here we can see a summary of the performance of each algorithm described above.



We can see that from the initial non parallel algorithm to the final parallel algorithm, the improvement was almost 2 times on my machine, a significant improvement.

5.b Chunk Size Performance

One important question when deciding to split the data up into chunks is to decide what should the chunk size be. To determine this I picked a few random chunk sizes and then tried to optimize doing some kind of binary search/gradient descent to find the best performance. You can see a summary of chunk sizes for the 10000 people, 100 location dataset here:

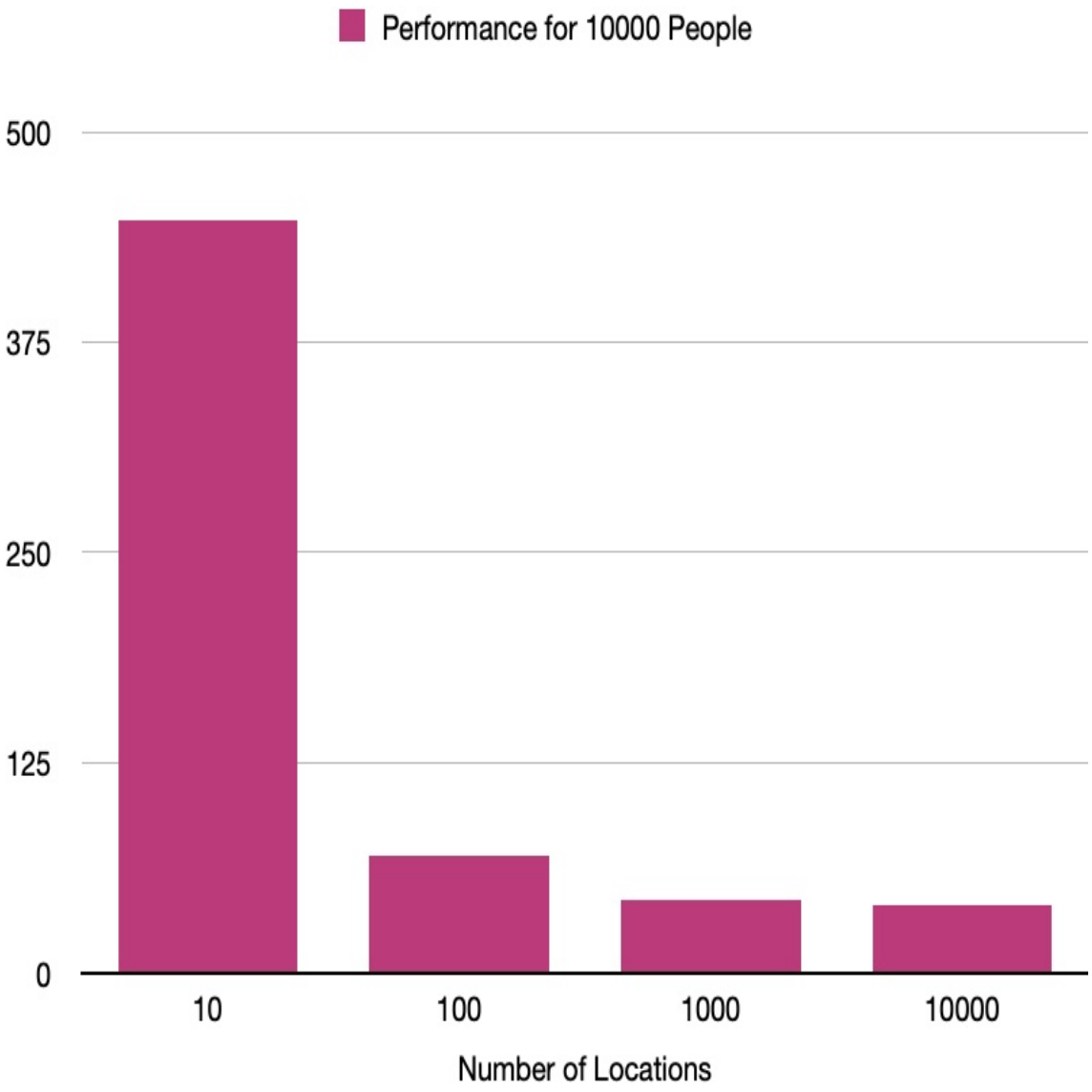


We can see that at really large or really small chuck sizes, performance is quite bad, in fact at chunk sizes of 1 and 10000 performance is worse than the non parallel version. However somewhere around a chunk size of 40 we hit peak performance. One interesting feature I am not able to explain is that although performance starts to get worse as you get less than 40, at a chunk size of 15, performance was almost identical to 40 (event though at 10 and 20 it was worse.

Potentially one improvement here would be to have one chunk size for events and one for people as they are different sets and could move independent of one another.

5.c Performance by Number of Locations

One interesting question is how does the number of locations impact the performance. Well looking at the algorithm we can see that per event we filter the events at that location for that day down to the intersecting ones. This means that the lower number of locations, the more work there is to do per event, thus we would expect to see a smaller number of locations meaning a slower algorithm. This is exactly what we do see:



At 100 Locations we saw we achieve performance of about 69 seconds, at 10 locations though performance is 447 seconds, much much worse. However at 1000 and 10000 locations performance steadies out to 43 and 40 seconds respectively.

This does point to a potential optimization in our implementation, where instead of filtering all events for a day to see which overlap, we could store the events in sorted order and do a binary search to find the first event that overlaps, and then search from there. This could potentially improve the runtime on 100 location dataset down to 40 seconds, like we see with the 10000 location dataset.

5.d Dataset Growth Performance

Finally another interesting section is to analyse how the algorithm performs as the size of the dataset increases. When I increased the number of people from 10,000 to 100,000 it started to wreak havoc on my computer, using too much memory, and causing it to run unnecessarily slow. We can see performance stats here:

Performance by Number of Locations

People	Locations	Performance
100	10	0.410
1000	100	3.7
10000	1000	43
100000	10000	1980

So whereas up to 10000 we were seeing linear growth, adding 10 times more people took 10 times longer, it took almost 50 times longer to do handle the last 10 times jump. As stated above, I believe this is a matter of the resources available on my computer. One potentially improvement to reduce the amount of resources necessary would be to only load the events for one day at a time from disk into memory, instead of loading all the events for all days into memory.

6 Continuations

In the performance sections above I listed several performance improvements that could potentially be made that would help reduce the runtime of the implementations. A summary of those that would be worth trying out:

1. Using different chunk sizes for people and for events as they are different sizes (x people, and on average n*x events a day, where n is the average number of events per person per day).
2. Implementing a O(LogN) binary serach based algorithm for finding overlapping events instead of doing an O(N) algorithm.
3. Loading the events for one day into memory at a time to reduce overall memory use
4. Run this on a dedicated machine with more cores instead of my laptop. My laptop has a bunch of other processes running on it, so results can be inconsistent based on whatever else the laptop is running. Also my laptop only has two CPUs which is not ideal for testing things out across multiple cores.

7 Summary

In summary over the course of the project I tried multiple different approaches to improve performance including better structuring of the algorithms, better storing of the data, better chunking of the data, and most importantly parallelism. After the several rounds of iterations I was able to reduce the initial performance of 130 seconds, down to a final performance of 69 seconds, an 88 percent performance improvement.

References

[1] Barrett, Christopher L. and Bisset, Keith R. and Eubank, Stephen G. and Xizhou Feng and Marathe, Madhav V. *EpiSimdemics: an Efficient Algorithm for Simulating the Spread of Infectious Disease over Large Realistic Social Networks*. SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing

[2] Talwar, Kunal, and Udi Wieder. *Overcoming the Scalability Challenges of Epidemic Simulations on Blue Waters*. 2014 IEEE 28th International Parallel and Distributed Processing Symposium, 2014