Simulating RNA Virus's Propagation in Real World Environments

Rikky Roy Koganti (rkoganti)
Elena Feldman (efeldman)
15-418 Spring 2014

Summary
For our final project we simulated from scratch how 3 RNA viruses (HIV-1, Ebola, Influenza-a) propagate from person to person in six different cities around the world.

Background

- *Data structures*
  The most primitive type in our system is a Person. The Person object models a real world human and for our simulation, it has the following main attributes:
  - age: pretty straightforward..
  - real world (x,y) coordinates: where he/she is according to the **city** map, not according to the **world** map
  - days left: how much the person has until he/she (sadly) dies
  - is infected? : has the virus infected the person and if so, which ones
  - is immune? : is the person immune to a virus, if so which ones
  - is vaccinated? :is the person vaccinated against a virus, if so which ones

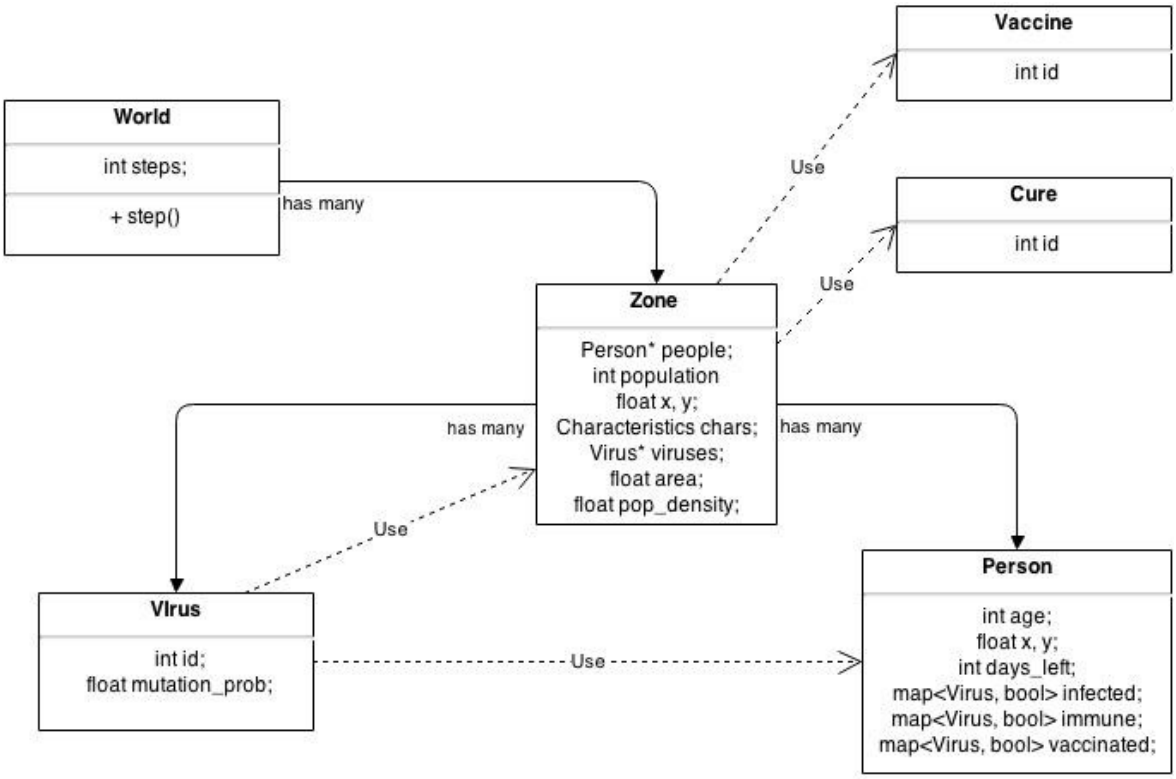
  Next is a Zone. A Zone object models a city, and has a lot of attributes, but to highlight the important ones:
  - all people : array of all Person objects of that zone
  - real world (x,y) coordinates: based on longitude and latitude of the real world map
  - [characteristics] : has a lot of defining constants that help out with the algorithm and probability calculation, examples of which include wealth, pollution, government stability, etc.
  - all viruses : array of all viruses that have been introduced to the area
  - area/ population/ population density : also used in calculations and determining each person's world coordinates when rendering

  We also have Virus object that models the functionality of a real virus. This is where the actual logic for infecting people happens. In order to imitate RNA, we had a unique id, and a mutation probability per replication. In more coding terms, we also had each Virus keep track of all the people it infected, and its strength to see the “radius of transmission” (will be described in more detail in the next section).

  We had a Cure and Vaccine objects, that modeled their counterparts as well. They are basic objects which just carry a unique id in order to calculate whether it will cure against/help prevent some virus.

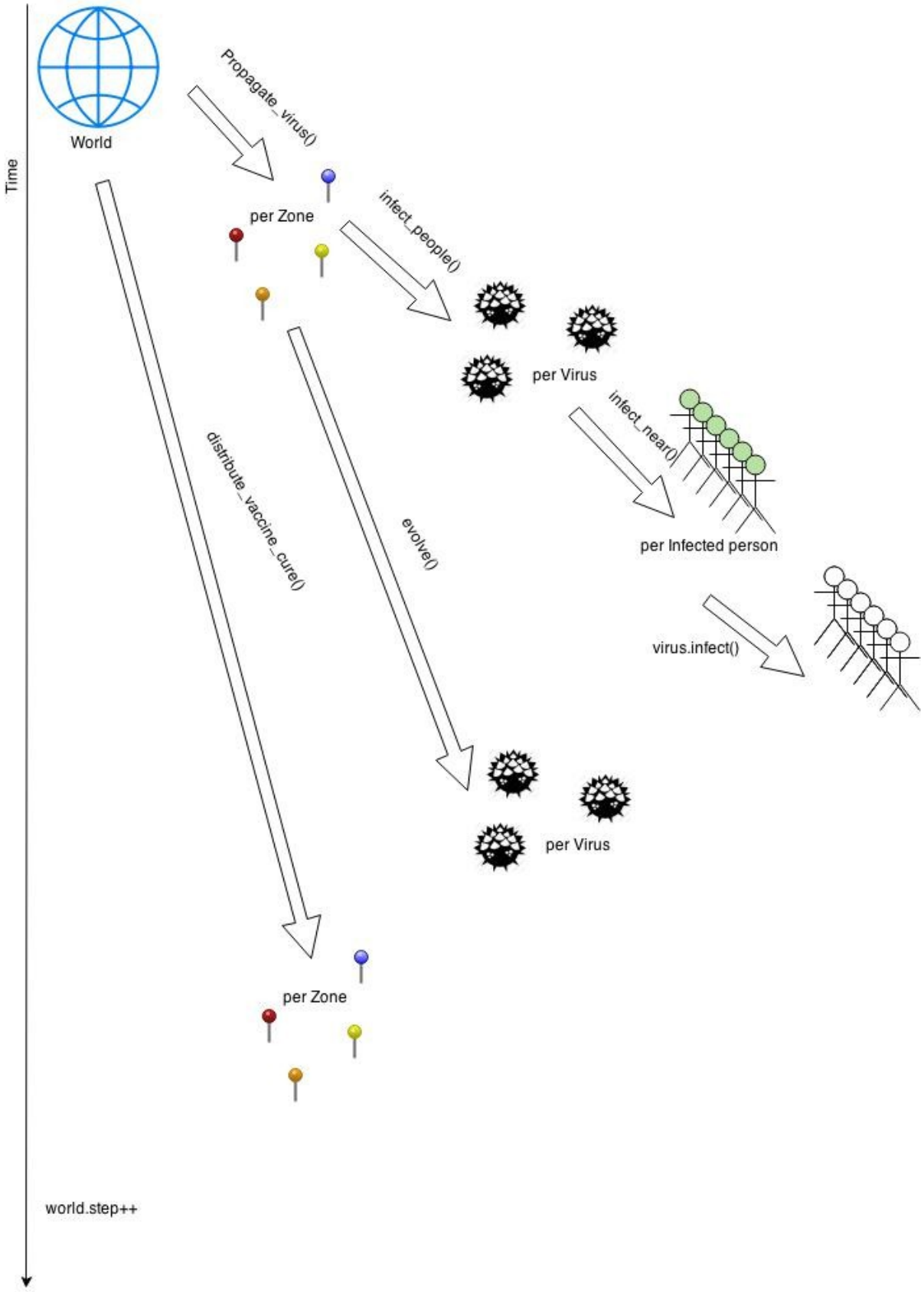
  Finally, we have a World. The World contains the zones which are its only main attributes in addition to keeping track to how many steps have passed (days).

  All of these objects can be summed up in the diagram below.



- *Operations on data structures*
  Going from the most general to the most specific, we'll start with the World.
  A world has a function step() which represents one day. In it, it operates on all its zones.
  Each zone then in turn operates on all of the viruses in its propagate_virus() function call.
  For each virus, it goes through all the people it has infected and calculates the probability of infection, taking into the consideration all the key attributes of people, zones and vaccines.
  It now goes through the people surrounding the infected person and calculates the probability of infection based on all the attributes of the person(s) and the cities.
  Once the infection stage has passed, the virus gets a chance to evolve by using past infected people as samples and machine learning to figure out the best way to change.
  Back in the world, still in the same day, we see if a cure or a vaccine has been developed, and if it has (depending on the zone's characteristics) distribute it to the people in the zone.

  These operations can be described more generally in the diagram below.


Time
World
Propagate_virus()
per Zone
infect_people()
per Virus
infect_near()
per Infected person
virus.infect()
distribute_vaccine_cure()
evolve()
per Virus
per Zone
world.step++

- *Inputs and Outputs*
  The inputs into our system would be cities and people. The virus does some work, and the system outputs the finalized people's states (such as infected and where he/she is now).

- *Benefits from Parallelization*
  The most computationally expensive parts is a zone's propagate_virus(), or in other words, where operations happen on each person in every city.
  The second most computationally expensive part (but not really on par with propagate_virus()) is the update_time_left(). This is also per person operation, however, while the propagate_virus() is $O(N^2)$, update_time_left() is O(N).

- *Workload Breakdown*
  The major dependencies in our program include the ones of nearby infection. In example, for our simulation, an healthy person's state (healthy or sick) depends on any surrounding infected people.
  However, there is still a ton of parallelism that can be exploited such as per zone (zones are completely independent of each other, except for one small "cure trading operation" that happens once every 200 or so steps, which is done serially). There is a per all people parallelism in determining the neighbors as well as per all sick people data parallelism.
  Our simulation can definitely benefit from SIMD because in actuality, our simulation is *heavily* computation dependent. What we mean is that in each step, each comparison between one sick and one healthy person involves various probability/math functions that don't vary from situation to a situation.
  We exploit locality by distributing work on the basis of spatial locality, which is in turn based on the person's and his/her surrounding people's x,y coordinates. The details of how this is done is described further in the Parallelization of the System section.

(Optional) Details of Serial Implementation of Simulation

We started off the project by first implementing a simulation completely in serial. We were more concerned with having a functional, working simulation. As mentioned previously, we input 6 different zones and 3 different viruses as arguments to the simulation. The world class then sets up the initial world, with the 6 different zones, the thousands of Person data structures in each of those zones, and we then start running the step() function. We basically have the simulation continue running by continuously calling the step() function. All the functionality of the simulation is called from this function.

In the step() function, the very first thing we do is call an introduce_virus() function. This will start infecting 100 random people in each zone with a virus. We call this function one every 500 steps, and only call it 3 times at most since we have 3 viruses in the system. Each virus is introduced in the virtual world with different attributes, attributes that we will go into later. We next check that the current step is greater than 0. If it is, we can now start propagating all viruses affecting each zone, by calling the propagate_virus() function. This function is run by each zone. Each virus in each zone keeps a track of the people it has infected. Each virus also has a strength attribute that the propagate_virus() function uses to go through each of the infected people and then finds all the nearest people to the former (within +- strength units of their position). Once each infected person has been mapped to a list of the nearest people to them, the virus then calls its' infect function on each of the nearest neighbors.

The infect function is not a 100% sure thing of infection. It has a certain probability of infecting the neighbor, and this probability is influenced by various factors. Firstly, a zone with higher population density would have higher transmission rates of viruses so these zones have higher probabilities of infection. There is also a certain element of randomness involved, by using (rand() % 100) to obtain a random value between 1-100. Finally, the probability also depends on the virus that is trying to infect a person. If the virus is Influenza, then the chances of infection also take in account the target's age as elderly people and children are more susceptible to it. We would also look at the zone's pollution levels; higher pollution levels provide more opportunity for influenza to spread. If the virus is HIV, we take into account the age differential between the target and the infected person. Since HIV is spread widely through sexual relations, a larger age difference would mean the likelihood of there being a sexual relation falls accordingly. HIV can also be spread through poor healthcare practices so zones with a lower healthcare attribute have higher infection rates for HIV. The third and final virus, Ebola, is different. It does not discriminate amongst its victims and has a very high infection rate, so we just use a high constant value and the level of pollution in the zone here.

The interesting thing here is that when a virus successfully infects another person, it has a chance to mutate. Each virus has a different probability of mutation, and mutation basically changes the viruses' id attribute. We'll talk more about the importance of virus ids later. For now, after the propagate virus has finishing propagating viruses in all zones for this step, we then move on to the update_time_left() function. Each Person struct has a time_left attribute, which is basically how many steps left that the person has left to live. If the person was completely healthy, that attribute would decrease by 1 every step. However, each virus that the person is infected with (a person can be infected with more than 1) reduces that attribute by a greater amount. Thus, this function goes through every live person in each zone, and updates the value of that attribute depending on whether they are healthy or infected.

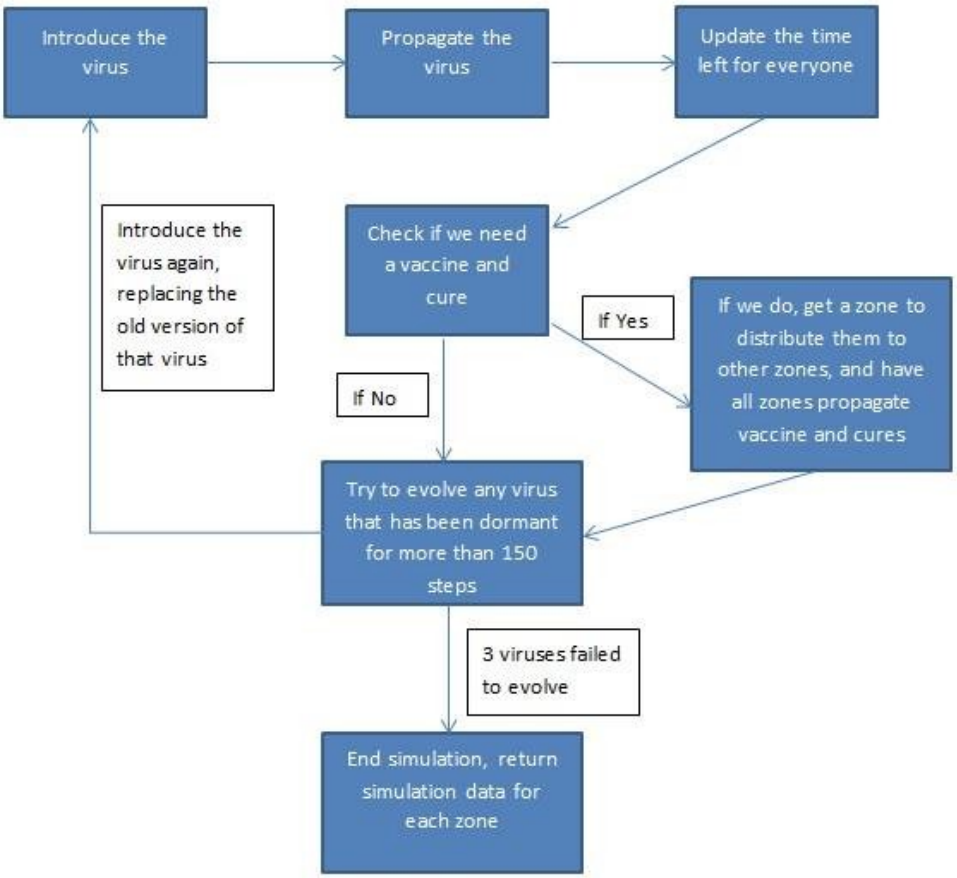
The next phase in the step function is to check whether it is time to introduce vaccines and cures for a virus into the world. We call a check function that basically checks if a certain amount of time has passed since the virus was running active in the zones, or if more than 10% of the people in any one

zone have died. If either condition is true, we call the distribute_vaccine_and_cure() function. This function first check which zones have the virus for which we need a vaccine and cure. It then assigns each of these zones a probability that has a randomness element, and also takes into account the wealth of the zone, the amount of infected people in the zone, and the quality of the government of that zone. Then, the zone with the highest probability is assigned the role of the distributor. Each of the zones has a friendship level with every other zones. The distributor makes an unlimited number of vaccines but a limited amount of cures. Therefore, the distributor first uses as many cures as its people need, and then spreads the remainder proportionally to the other zones based on their friendship levels.

Now that every zone has cures and vaccines, each zone then calls the propagate_vaccine() and propagate_cure() functions in order to spread the vaccine and cure to its people. The propagate_cure() function is ran first, and goes through as many infected people as it has cures, and calls the cure function on each of them. The cure function basically modifies the person's attributes to read uninfected. It also vaccinates the person against all viruses with the same id (the same mutation) in the future. The propagate_vaccine() function simply goes through every alive uninfected person in the zone, and vaccinates them against a large class of mutations of a single virus. It does this in the following manner. If it is a strong vaccine with strength 8, then that person is immune to all viruses of that class who have that virus->id is not equivalent to 0 modulo 8. This means that the person is immune to 7 of the 8 different kinds of mutations for the virus, thanks to the strong vaccine. The strength of the vaccine varies depending on the virus. Ebola and HIV have weaker vaccines while the flu has a stronger one.

The last phase for the step function involves the evolution of a virus. After everyone in the zone has been vaccinated against a virus, it enters a period of dormancy. The virus can still propagate, but it has no harmful effects on a persons' time left. After an interval of dormancy, the virus then tries to evolve. So at this phase every call to the step function, we call the evolve_virus function. Whether a virus evolves or not depends on randomness, the level of healthcare and pollution in the zone (higher healthcare means the virus has greater potential for evolution as all viruses build up their offences against a strong defense, and higher pollution means the virus propagates more easily and has more chances to mutate and evolve). It also depends on the amount of infected people previously. This is because a virus depends on the amount of mutations that have arisen as one of these mutations eventually goes on to become a new evolved virus. If the probability allows the virus to evolve, we then make the virus immune to all past vaccines and cures by changing its id and activate the virus again. It propagates and causes harmful effects to people one again. However, if the virus fails to evolve even once, it does not get a second chance and remains dormant forever. So the simulation would theoretically end when all 3 viruses become dormant.

A more detailed overview of the propagate virus simulation:



Parallelization of System

We wrote the entire code in C++ and since we were aiming to test the code on the Gates computers with NVIDIA GTX 680, we decided to use CUDA in order to parallelize our code. Also, the graphics portion of our code was written using openGL and GLSL as we could only run those on select machines in Gates so this played a part in our decision too.

We will refer to figure 1 above to describe which parts will be run in parallel. Introducing a virus into every zone will be executed in parallel as this involves infecting 600 people in total each time we call it. Propagating a virus in a zone is also done in parallel since this would potential involve going through almost the entire population of each zone in a single step. Updating the time left and propagating a vaccine also go through everyone in each zone so this part has huge potential for parallelism as well. Finally, the propagation of the cure to every infected person can be done in parallel as well; when the cure finally comes into play, a lot of steps will have occurred so the amount of infected people will likely be greater than the amount of uninfected people in each zone. Since there are only 6 zones, we will go through each of the zones sequentially and parallelize the functions run on each zone.

The first thing we notice is that all of these operations involve Person objects in each zone. This

provided a useful idea for mapping people to threads in thread blocks for execution. We think of each zone has a grid, and for every person in that grid has their own x and y coordinates. Every time I have a global kernel function, to parallelize each of the functions mentioned above, I have a for loop going through Person objects and device functions inside the loop iterating on the objects. I loop through an entire blocksize worth of objects at once.

Firstly, let's discuss the propagate virus function. A lot of the computation takes place here because this function is called every step and is an $O(N^2)$ algorithm due to way it finds nearest people (a naïve manner) I split up the zone into smaller grid blocks. Each thread is then mapped to a distinct person in the zone using the thread id. We then check the grid that person is in in the zone. If the thread block the thread is in fits inside that grid, this thread does work later in getting all the nearest people to the current infected person. This partition helps to speed up the later parts of the propagate virus function, when I get all the nearest people. At this point, I have to decide whether or not to go through the list of infected people in parallel and the list of nearest neighbors for each person in serial or the other way round. In the end, what I decided to do was to base my decision based on the amount of infected people I have. If the amount of infected people is less than 20% of the population, I go through the list of infected people in serial and the list of nearest neighbors in parallel. If more than 20% are infected, I go through the infected people in parallel, infecting their neighbors in serial. I choose 20% as the benchmark instead of a higher number due to a few reasons. Firstly, threads are likely to be doing work for people near to each other in the grid due to way I partitioned them. This means that after one thread has finished infecting the neighbors, another thread might be trying to infect some of these neighbors too but does not need to do any work as they are already infected. Thus, with a higher number of infected people, the neighbors are more likely to overlap with each other. 20% seemed to be a good benchmark, after some testing. Lastly, I wrote my code such that I did not need to make the infect function atomic since we are accessing shared memory. If two threads are trying to infect the same person, that is fine in my simulation. Either both threads succeed, both fail, or just 1 fails. In the end, the person is either infected or not. This is fine as in the real world, a person can be in danger of getting infected from two neighbors too. The logic of the code overall does not break.

The other function that was important to optimize was the update_time_left function. This was because this thread was an O(n) algorithm that went through everyone in the world. However, the computation for this is lesser than the propagate virus. We similarly partition threads again here as well. We then update the time left for everyone in parallel. The propagate vaccine and cure functions provide a bit of speedup when parallelized but in the end, they are called a small fraction of the time update_time_left and propagate_virus are called so the contribution to speedup from the former functions is relatively lower. As a result, we also did not focus too deeply on trying to optimize those parallel functions. However, the major issue that we tried to optimize for all of them was when it came to atomicity. The good part of our system is that many of the parallel functions can be done without worrying about race conditions. We just have to sync threads between the different global kernels but we don't need to worry about atomicity inside of the device functions. Originally, we implemented the propagate_cure function such that the people with higher wealth attributes got the cures first. However, there were a limited number of cures, and if we had decided to keep going with this implementation for the parallel version, the race conditions might have caused people with lower wealth attributes to be cured first and a thread in the middle of curing another person might find itself with no cure left to access (it will go out of bounds error). Thus, I changed this implementation to simply curing as many people as possible without worrying about who we were curing.

Another optimization we looked at was to minimize data transfers with low bandwidth. We needed data transfers as the main function for the graphics updated the Person objects and required

updates of the data from the kernels. One way we did this was by identifying the timeline of functions. Instead of copying data from host to device and back to host with every global kernel, we instead only updated the data when the data had been modified by previous functions. By going through the timeline by which functions were executed, we were able to find out when and when not to call the cudaMemcpy functions and still obtain correctness. Another choice we made was to have some of the host memory that was had to be constantly updated be global memory instead. Data transfers between global memory and device memory also have lower bandwidth so this helped. When doing this, we declared a struct storing pointers to attributes and passed those pointers to device memory. However, when we declared this struct as __device__, we received an error saying that CUDA can't declare structs that have members with non-empty constructors or destructors as __device__. Thus, we made the design change to remove all non-empty constructors and instead utilized an init_class function for each class that returned an instance of that class.

When actually trying to make the code parallel, we couldn't reuse much of the code from the serial implementation as device code can't call host code and vice versa. Thus, we could only use the pseudocode of the serial implementation and instead had to write numerous device functions in order to replicate the functionality of the serial implementation. Still, writing the serial implementation gave us an extremely good idea of our overall system so when it came to writing the numerous device functions, it was extremely straightforward.

Results

- *Our tests*

  ~Timing~

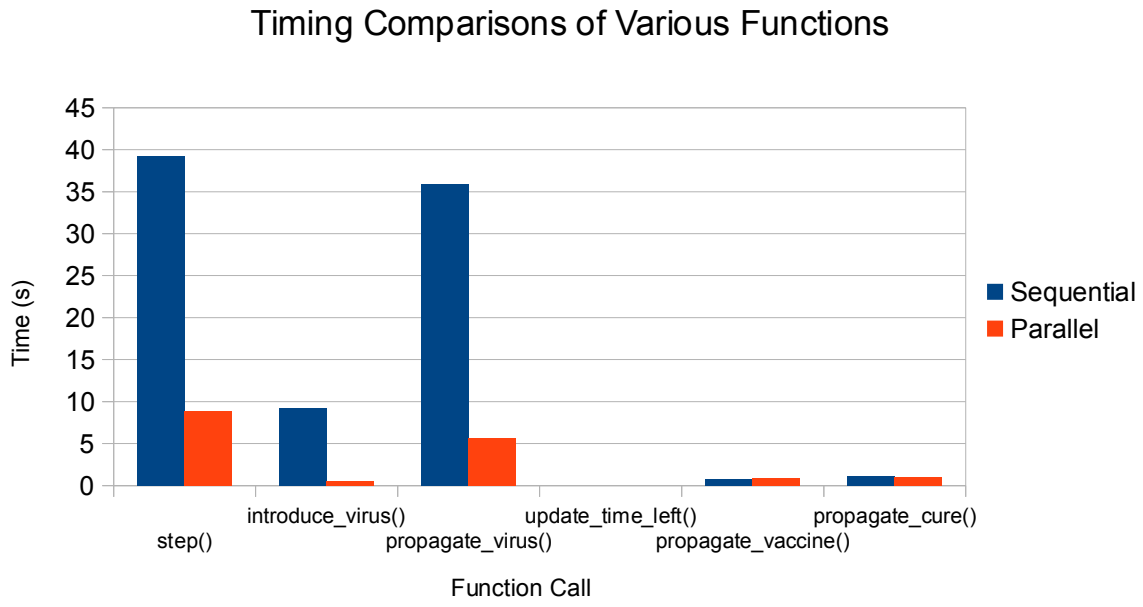
  Important note: Before going into the detail about our results, it is important to note that the best way to measure our outcomes is by simply timing (using 418's CycleTimer) 5 different function calls (described below).
  In addition, the timing of the rendering and OpenGL is not taken into account when calculating timing.

  - step() function: we took the time at the beginning and at the end of the whole world.step function which includes everything that was mentioned previously in the background-> operations on data structures part.
  - propagate_virus() function: we took the time inside the world.step function, right at the beginning of all zone's propagate_virus() and at the end of it.

  ~Inputs/Setup~

  We ran our tests on the 6 sample zones with the attributes that were chosen (hardcoded on initialization) by ourselves based on research about areas. i.e. between 0 and 1, how well the zone's healthcare technology is, etc.
  Unfortunately, altogether, the 6 zones had too large of populations for our algorithm to process in a quickly enough time, so we scaled it down to 1 tenth of the actual population sizes.
  Finally, in the beginning 200 steps, we only had one virus (the flu) introduced.
  This is an implementation for the GPU using multi-core/SIMD parallelism.

  ~Graphical Analysis~



~Analysis/Limits of Speedup~

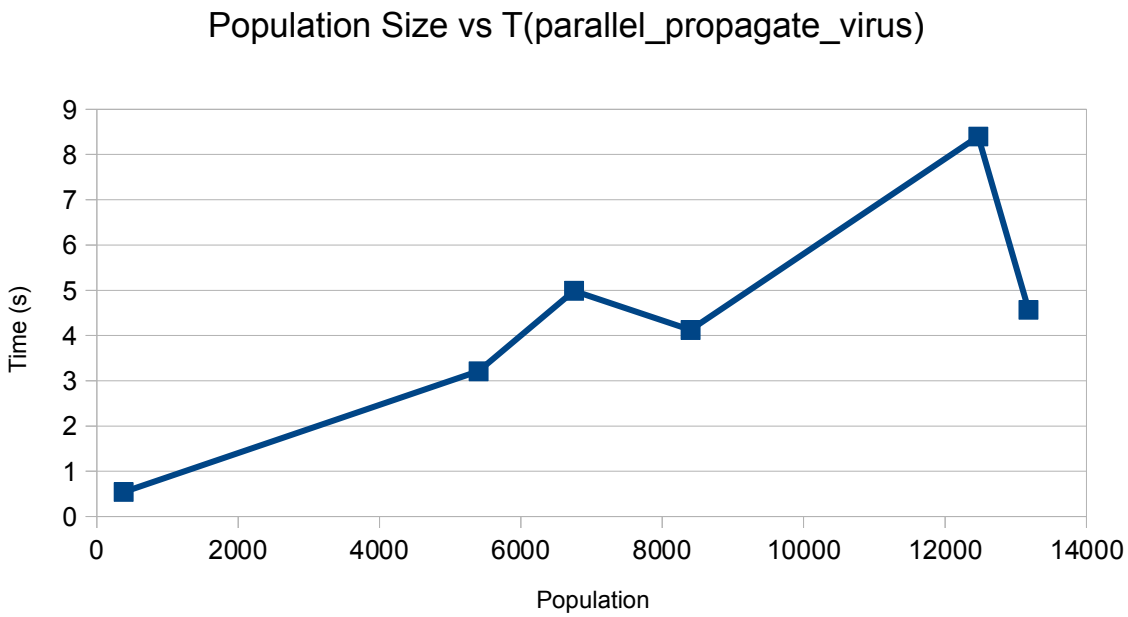
Although we feel like we had good parallel optimizations, our speedup was not as great as we expected. One of the reasons for this is likely that to be because our propagate virus algorithm is not as efficient as we would like it to be. It uses a naïve algorithm to find all the nearest people for each infected person to propagate the virus too and this makes it an $O(N^2)$ algorithm. However, the majority of our computation is spent inside the propagate virus function. A smaller part is spent inside the update_time_left function and the rest of the code where we update attributes and we have the people moving inside each zone. Also, the time spent inside the propagate_virus function between serial and parallel implementations has about the same speedup as the overall time spent. This goes to show that the propagate_virus function contributes to a large part of the speedup. Any inefficiency here ends up slowing down the entire program. A future optimization would be to use a faster O(nlog n) algorithm instead.

One of the reasons that we speculate could be another reason is the lack of parallelism when it comes to the zones. Right now, our parallel implementation runs in parallel on each person in a zone, but not over all people in the world. It iterates serially through the 6 zones. If we modify our algorithm such that we make it in fact run in parallel over everyone in the world, it is highly likely we will get more speedup.

~Varying Problem Size~

The only way we can vary the problem size and see its effects on our current implementation is to increase the population size. Due to time and resource limitations (~1 week for parallelization) we were not able to scale up to real population sizes. However, we still saw how dependent it was on the population. For that, we broke down our timing per zone and measured each zone's propagate_virus():

Population	propagate_virus() [parllel]
5400	3.211 s
8400	4.123 s
13180	4.566 s
12470	8.398 s
380	0.542 s
6750	4.986 s



Even with larger population sizes, the propagate virus function had similar scale up as when the population size increased, the time taken for the function increased nearly linearly. However, Mumbai takes a longer time than Tokyo even though the population is smaller, because in our simulation time depends on the amount infected people in the zone as well. Mumbai is slower because it has much higher population density and high pollution, causing our simulation to have much more infected people. This is a good result to see because it proves some validity in our probability functions and showed that time does not increase quasi-linearly with population size, as it depends on attributes.

In fact, to prove this point, we looked at the number of infected people for Tokyo and Mumbai:

Tokyo gave 4261 infected people, while Mumbai gave 9768 at the end of our simulation. This is more than twice, as supported by our timings (timings were twice as slow for Mumbai).

Even so, as we go from smaller to larger sizes, the execution behavior does not change. Still, we did not test it with the real populations sizes in the millions since it takes a much longer time to run and uses a lot of resources. Since the algorithm is O(n2), when we go into the millions, the increase in time taken won't be linear anymore, but more quadratic.

What we took away from the project/Future plans

One of the biggest drawbacks for our simulation was the lack of speedup. However, we know that the musician is to be blamed, and not the instrument, as our finding nearest neighbors algorithm could have been more complexed and improved. In the future, that is the first part we will work on. Next, we would want to parallelize per zone, as that is completely independent and is trivial, and because we are going to continue working on this as a side research project, where we hope to expand this to much more than 6 zones.

In the good news, we saw results that our probability functions and logic worked very well together, appropriately to the attributes we gave. We were able to reach about 18x speedup on perfectly O(n) independent functions (not propagate_virus()). In that, we reached only about 7-8x speedup.

References

– The Visual Science Company : for their detailed information about RNA viruses
http://visualscience.ru/en/company/
- 15-418 Website : for their very interesting notes on parallelism
- StackOverflow : for everything <3

The work has been balanced between the two partners.