Pandemic with MPI

CSInParallel Project

March 29, 2016

CONTENTS

In this module you will read about how we can model the spread of infectious diseases computationally. We can make use of distributed computing with message passing to shorten the time needed to model very large populations, which can be computationally intensive on a single computer. Each section linked below explains both the computational model and the code, which uses the Message Passing Interface Library, or MPI, to build a distributed processing solution.

CHAPTER

ONE

INFECTIOUS DISEASE

By Aaron Weeden, Shodor Education Foundation, Inc. [1]

Heavily modified by Yu Zhao, Macalester College

Overview

Epidemiology is the study of infectious disease. Infectious diseases are said to be "contagious" among people if they are transmittable from one person to another. Epidemiologists can use models to assist them in predicting the behavior of infectious diseases. This module will develop a simple agent-based infectious disease model, develop a parallel algorithm based on the model, provide a coded implementation for the algorithm, and explore the scaling of the coded implementation on high performance cluster resources.

Pre-assessment Rubric

This rubric is to gauge students' initial knowledge and experience with the materials presented in this module. Students can be asked to rate their knowledge and experience on the following scale and in the following subject areas, bith before and after they complete this module.

Scale

1. no knowledge, no experience
2. very little knowledge, very little experience
3. some knowledge, some experience
4. a good amount of knowledge, a good amount of experience
5. high level of knowledge, high level of experience

Subject areas

- Disease modeling
- Parallel Algorithm Design
- Parallel Hardware
- MPI programming
- OpenMP programming
- Using a cluster
- Scaling parallel code

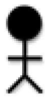
[1] For original documentation and code developed by Aaron Weeden, please go to original pandemic.

The goal of the reading and exercises in this module are to see gains in the above subject areas after completing it.

Model

The model makes certain assumptions about the spread of the disease. In particular, it assumes that the disease spreads from one person to another person with some "contagiousness factor", that is, some percent chance that the disease will be transmitted. The model further assumes that diseases can only be spread from a person who is carrying the disease, a so-called "infected" person, to a person who is capable of becoming infected, also known as a "susceptible" person. The disease is assumed to have a certain incubation period, or "duration" – a length of time during which the disease remains in the person. The disease is also assumed to be transmittable only within a certain distance, or "infection radius", from a person capable of transmitting the disease. The model further assumes that each person moves randomly at most 1 unit in a given direction each day. Finally, the model assumes that after the duration of the disease within a person, the person can become either "immune" to the disease, incapable of being further infected or of infecting other people but still able to move around, or "dead", incapable of being further infected, infecting other people, or moving.

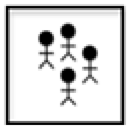
The description below explains the various entities in the model. Things in underlines are entities, things in **bold** are attributes of the entities, and things in *italics* refer to entities found elsewhere in the description.


(pl. people)

- Has a certain **X location** and a certain **Y location**, which tell where it is in the *environment*.
- Has a certain **state**, which can be either 'susceptible', 'infected', 'immune', or 'dead'. States are stored in the memories of *processes* and *threads*. They can also be represented by color (black for susceptible, red for infected, green for immune, no color for dead), or by a ASCII character (o for susceptible, X for infected, I for immune, no character for dead).

Disease

- Has a certain **duration**, which is the number of days in which a *person* remains infected.
- Has a certain **contagiousness factor**, which is the likelihood of it spreading from one *person* to another.
- Has a certain **deadliness factor**, which is the likelihood that a *person* will die from the disease. 100 minus this is the likelihood that a *person* will become immune to the disease.


Environment

- Has a certain **width** and **height**, which bound the area in which *people* are able to move.

Timer

- Counts the **number of days** that have elapsed in the simulation.

Thread (pl. threads)

- A computational entity that controls people and performs computations.
- Shares **memory** with other threads, a space into which threads can read and write data.

🟦 Process (pl. processes)

- A computational entity that controls people and performs computations.
- Has its own private **memory**, which is a space into which it can read and write data.
- Has a certain **rank**, which identifies it.
- Communicates with other processes by **passing messages**, in which it sends certain data.
- Can spawn threads to do work for it.
- Keeps count of how many susceptible, infected, immune, and *dead* people exist.

Introduction to Parallelism

In parallel processing, rather than having a single program execute tasks in a sequence, the program is split among multiple “execution flows” executing tasks in parallel, i.e. at the same time. The term “execution flow” refers to a discrete computational entity that performs processes autonomously. A common synonym is “execution context”; “flow” is chosen here because it evokes the stream of instructions that each entity processes.

Execution flows have more specific names depending on the flavor of parallelism being utilized. In “distributed memory” parallelism, in which execution flows keep their own private memories (separate from the memories of other execution flows), execution flows are known as “processes”. In order for one process to access the memory of another process, the data must be communicated, commonly by a technique known as “message passing”. The standard of message passing considered in this module is defined by the “Message Passing Interface (MPI)”, which defines a set of primitives for packaging up data and sending them between processes.

In another flavor of parallelism known as “shared memory”, in which execution flows share a memory space among them, the execution flows are known as “threads”. Threads are able to read and write to and from memory without having to send messages. [2] The standard for shared memory considered in this module is OpenMP, which uses a series of “pragma”s, or directives for specifying parallel regions of code to be executed by threads. [3]

A third flavor of parallelism is known as “hybrid”, in which both distributed and shared memory are utilized. In hybrid parallelism, the problem is broken into tasks that each process executes in parallel; the tasks are then broken further into subtasks that each of the threads execute in parallel. After the threads have executed their sub-tasks, the processes use the shared memory to gather the results from the threads, use message passing to gather the results from other processes, and then move on to the next tasks.

Parallel Hardware

In order to use parallelism, the underlying hardware needs to support it. The classic model of the computer, first established by John von Neumann in the 20:math:*{}^{th}* century, has a single CPU connected to memory. Such an architecture does not support parallelism because there is only one CPU to run a stream of instructions. In order for parallelism to occur, there must be multiple processing units running multiple streams of instructions. “Multi-core” technology allows for parallelism by splitting the CPU into multiple compute units called cores. Parallelism can also exist between multiple “compute nodes”, which are computers connected by a network. These computers may themselves have multi-core CPUs, which allows for hybrid parallelism: shared memory between the cores and message passing between the compute nodes.

Motivation for Parallelism

[2] It should be noted that shared memory is really just a form of fast message passing. Threads must communicate, just as processes must, but threads get to communicate at bus speeds (using the front-side bus that connects the CPU to memory), whereas processes must communicate at network speeds (Ethernet, infiniband, etc.), which are much slower.

[3] Threads can also have their own private memories, and OpenMP has pragmas to define whether variables are public or private.

We now know what parallelism is, but why should we use it? The three motivations we will discuss here are speedup, accuracy, and scaling. These are all compelling advantages for using parallelism, but some also exhibit certain limitations that we will also discuss.

"Speedup" is the idea that a program will run faster if it is parallelized as opposed to executed serially. The advantage of speedup is that it allows a problem to be modeled [4] faster. If multiple execution flows are able to work at the same time, the work will be finished in less time than it would take a single execution flow.

"Accuracy" is the idea of forming a better solution to a problem. If more processes are assigned to a task, they can spend more time doing error checks or other forms of diagnostics to ensure that the final result is a better approximation of the problem that is being modeled. In order to make a program more accurate, speedup may need to be sacrificed.

"Scaling" is perhaps the most promising of the three. Scaling says that more parallel processors can be used to model a bigger problem in the same amount of time it would take fewer parallel processors to model a smaller problem. A common analogy to this is that one person in one boat in one hour can catch a lot fewer fish than ten people in ten boats in one hour.

There are issues that limit the advantages of parallelism; we will address two in particular. The first, communication overhead, refers to the time that is lost waiting for communications to take place before and after calculations. During this time, valuable data is being communicated, but no progress is being made on executing the algorithm. The communication overhead of a program can quickly overwhelm the total time spent modeling the problem, sometimes even to the point of making the program less efficient than its serial counterpart. Communication overhead can thus mitigate the advantages of parallelism.

A second issue is described in an observation put forth by Gene Amdahl and is commonly referred to as "Amdahl's Law". Amdahl's Law says that the speedup of a parallel program will be limited by its serial regions, or the parts of the algorithm that cannot be executed in parallel. Amdahl's Law posits that as the number of processors devoted to the problem increases, the advantages of parallelism diminish as the serial regions become the only part of the code that take significant time to execute. In other words, a parallel program can only execute as fast as its serial regions. Amdahl's Law is represented as an equation as follows:

Speedup = $\frac{1}{1-P+\frac{P}{N}}$

where

- P = the proportion of the program that can be made parallel
- 1 – P = the proportion of the program that cannot be made parallel
- N = the number of processors

Amdahl's Law provides a strong and fundamental argument against utilizing parallel processing to achieve speedup. However, it does not provide a strong argument against using it to achieve accuracy or scaling. The latter of these is particularly promising, as it allows for bigger classes of problems to be modeled as more processors become available to the program. The advantages of parallelism for scaling are summarized by John Gustafson in Gustafson's Law, which says that bigger problems can be modeled in the same amount of time as smaller problems if the processor count is increased. Gustafson's Law is represented as follows:

Speedup(N) = $N\breve{}(1\breve{}P) * (N\breve{}1)$

where

- N = the number of processors
- 1–P = the proportion of the program that cannot be made parallel

[4] Note that we refer to "modeling" a problem, not "solving" a problem. This follows the computational science credo that algorithms running on computers are just one tool used to develop *approximate* solutions (models) to a problem. Finding an actual solution may involve the use of many other models and tools.'

Amdahl's Law reveals the limitations of what is known as "strong scaling", in which the number of processes remains constant as the problem size increases. Gustafson's Law reveals the promise of "weak scaling", in which the number of processes increases along with the problem size. These concepts will be explored further in Exercise 4.

Code

The code in this module is written in the C programming language, chosen for its ubiquity in scientific computing as well as its well-defined use of MPI and OpenMP.

The code is attached to this module in pandemic-MPI.zip (there will be a link to download it in the next section). After unpacking this using an archive utility, use of the code will require the use of a command line terminal. C is a compiled language, so it must be run through a compiler first to check for any syntax errors in the code. To compile the code in all its forms of parallelism, enter "make all" in the terminal. For other compilation options, see the Makefile. To run the program, enter "./pandemic.serial" to run the serial (non-parallel) version, "./pandemic.openmp" to run the OpenMP version, "mpirun –np <number of processes> pandemic.mpi" to run the MPI version, or "mpirun –np <number of processes> pandemic.hybrid" to run the hybrid OpenMP/MPI version. Each version of the code can be run with different options by appending arguments to the end of commands, as in "./pandemic.serial –n 100". These options are described below:

- -n <the number of people in the model>
- -i <the number of initially infected people>
- –w <the width of the environment>
- –h <the height of the environment>
- –t <the number of time days in the model>
- –T <the duration of the disease (in days)>
- –c <the contagiousness factor of the disease>
- –d <the infection radius of the disease>
- –D <the deadliness factor of the disease>
- –m <the number of actual microseconds in between days of the model> – this is used to slow or speed up the animation of the model

To help better understand the code, students can consult the data structures section below.

CHAPTER

TWO

PROGRAM STRUCTURE

`Download Pandemic-MPI.zip`

There are in total 7 files in this program.

File Name	Functions
Pandemic.c	Holds All the function calls
Defaults.h	Data structure and default values
Initialize.h	Initialize the runtime environment
Infection.h	Find and share all infected persons
Display.h	Display everyone's state and location
Core.h	Use serial or OpenMP for core operations
Finalize.h	Finalize the run time environment

2.1 Program Structure

The rest of the module will go through each of the code files. We can start with the *Pandemic.c* file.

2.2 Pandemic.c

At the very beginning of the file, We first include four files that are needed for all versions.

```
#include "Defaults.h"
#include "Initialize.h"
#include "Infection.h"
#include "Core.h"
#include "Finalize.h"
```

Then, if we are using display, we include the display code file.

```
#if defined(X_DISPLAY) || defined(TEXT_DISPLAY)
#include "Display.h"
#endif
```

2.2.1 main()

This function is the backbone of the whole program. It first initialize all the data structures need.

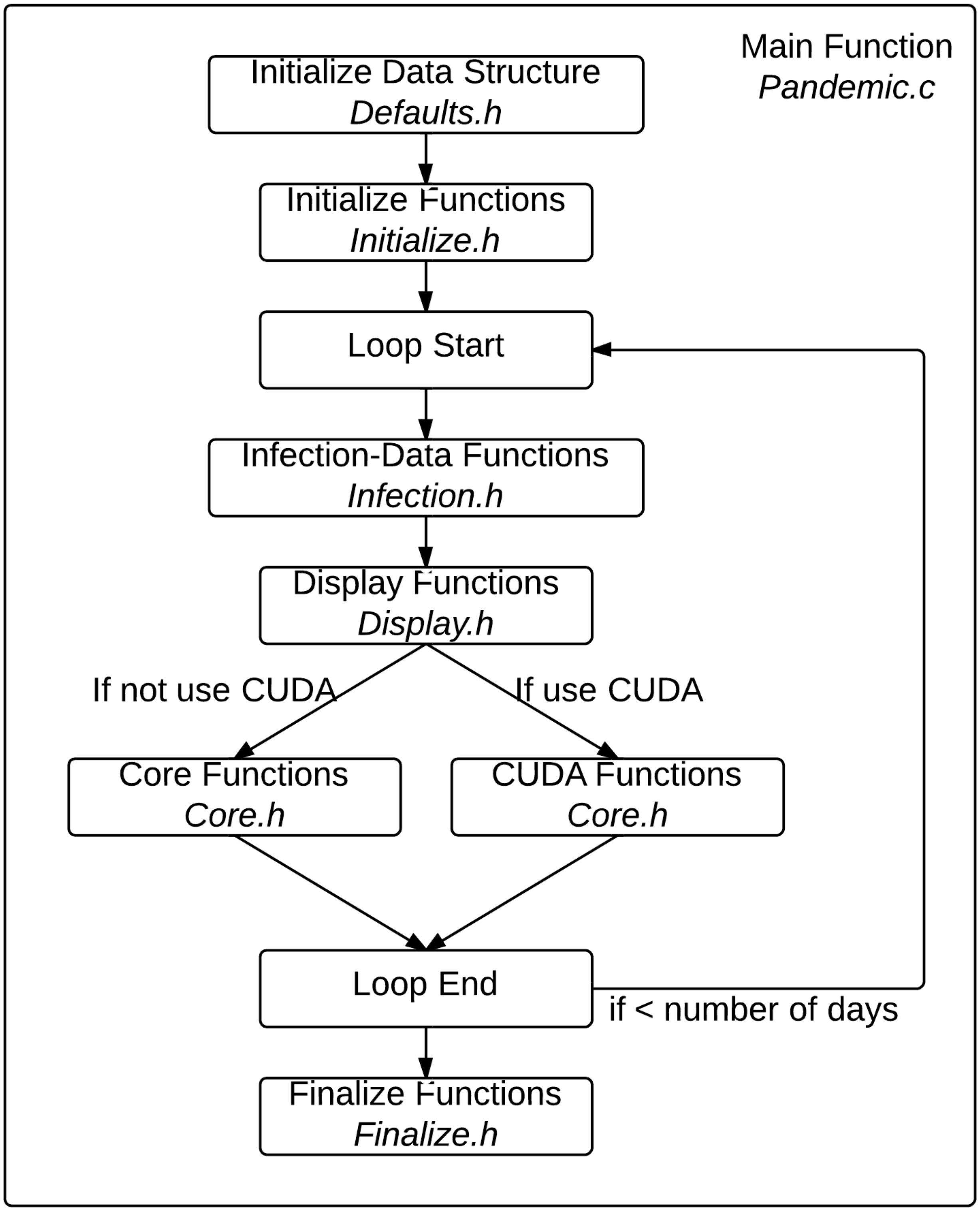


Figure 2.1: Overall Program Structurer

```
/**** In Defaults.h ****/
struct global_t global;
struct our_t our;
struct const_t constant;
struct stats_t stats;
struct display_t dpy;
/***********************/
```

Then it will initialize the runtime environment by calling **init()** function.

```
/******************* In Initialize.h *******************/
init(&global, &our, &constant, &stats, &dpy, &argc, &argv);
/*******************************************************/
```

Then we start the simulation. A for loop wraps around most of the functions, where the each iteration of the loop represents a day passing.

```
for(our.current_day = 0; our.current_day <= constant.total_number_of_days;
    our.current_day++)
{
}
```

Inside the for loop, we first find all data related to the infection.

```
   /****** In Infection.h ******/
    find_infected(&our);

    share_infected(&global, &our);
    /***************************/
```

Then, if display is enabled, we display the infection status. In other words, we display everyone's location and their states of infection.

```
    /*************** In Display.h ****************/
    #if defined(X_DISPLAY) || defined(TEXT_DISPLAY)

    share_display_info(&global, &our);

    do_display(&global, &our, &constant, &dpy);

    throttle(&constant);

    #endif
    /********************************************/
```

After display, we can call four core functions in *Core.h** code file.

```
    /****************** In Core.h *****************/
    move(&our, &constant);

    susceptible(&global, &our, &constant, &stats);

    infected(&our, &constant, &stats);

    update_days_infected(&our, &constant);
    /*********************************************/
```

This is the end of the loop.

Finally, after the loop, we can display the results and finalize the runtime environment.

```
/*************** In Finialize.h **************/
show_results(&our, &stats);

cleanup(&global, &our, &constant, &dpy);
/*********************************************/
```

CHAPTER

THREE

DATA STRUCTURES

Here is the list of variables and arrays used by the program. Note the naming scheme; variables whose names begin with "my" are private to the threads that use them. Variables whose names begin with "our" are private to the processes that use them, but public to the threads within that process. Variables are thus named from a thread's perspective; "my" variables are ones that I use, "our" variables are ones that I and the other threads in my process use.

3.1 global_t struct

```
// All the data needed globally. Holds EVERYONE's location,
// states and other necessary counters.
struct global_t
{
    // people counters
    int total_number_of_people;
    int total_num_initially_infected;
    int total_num_infected;
    // locations
    int *x_locations;
    int *y_locations;
    // infected people's locations
    int *their_infected_x_locations;
    int *their_infected_y_locations;
    // state
    char *states;
    // MPI related
    int total_number_of_processes;
};
```

total_number_of_people

the total number of all people in the simulation; the sum of people assigned to each process. The value of this variable can be specified on the command line with the –n option.

total_num_initially_infected

the total number of people who are initially infected; the sum of initially infected people assigned to each process. The value of this variable can be specified on the command line with the –i option. This is a subset of the total number of people, so the value of this variable must be smaller or equal to the value for **total_number_of_people**.

total_num_infected

the total number of infected people; the sum of the number of infected people assigned to each process. This value changes throughout the course of the simulation.

x_locations

array, holds the x locations of all of the people; only used if the environment needs to be displayed; otherwise, **our_x_locations** is used.

y_locations

array, holds the y locations of all of the people; only used if the environment needs to be displayed; otherwise, **our_y_locations** is used.

their_infected_x_locations

array, used in **susceptible()** function to keep track of the x locations of the infected people for which each process is responsible.

their_infected_y_locations

array, used in step **susceptible()** function to keep track of the y locations of the infected people for which each process is responsible.

states

array, holds the states of all of the people; only used if the environment needs to be displayed; otherwise, **our_states** is used.

total_number_of_processes

used to keep track of how many processes are being used. If MPI is disabled, the value of this variable will be 1. If it is enabled, the value is set in **init()** function.

3.2 our_t struct

```c
// All the data private to each node: Data being used by
// each process on a node in a cluster when using MPI.
// Each process holds data for location, states and
// other necessary counters for a subset of people.
struct our_t
{
    // current day
    int current_day;
    // MPI related
    int our_rank;
    // people counters
    int our_number_of_people;
    int our_num_initially_infected;
    // states counters
    int our_num_infected;
    int our_num_susceptible;
    int our_num_immune;
    int our_num_dead;
    // our people's locations
    int *our_x_locations;
    int *our_y_locations;
    // our infected people's locations
    int *our_infected_x_locations;
    int *our_infected_y_locations;
    // our people's states
    char *our_states;
    // our people's infected time
    int *our_num_days_infected;
};
```

our_current_day

a loop iterator representing the ID of the current day being simulated by the current process.

our_rank

used to keep track of the rank of the current process. If MPI is disabled, the value of this variable will be 0. If it is enabled, the value is set in **init()** function.

our_number_of_people

the number of people for which the current process is responsible. This will be a number less than or equal to the total number of people. The value is determined in **find_size()** function.

our_num_initially_infected

the count of initially infected people for which the current process is responsible.

our_num_infected

a count of the number of infected people for which the current process is responsible.

our_num_susceptible

a count of the number of susceptible people for which the current process is responsible.

our_num_immune

a count of the number of immune people for which the current process is responsible.

our_num_dead

a count of the number of dead people for which the current process is responsible.

our_x_locations

array, holds the x locations of all the people for which the current process is responsible.

our_y_locations

array, holds the y locations of all the people for which the current process is responsible.

our_infected_x_locations

array, holds the x locations of all the infected people for which the current process is responsible.

our_infected_y_locations

array, holds the y locations of all the infected people for which the current process is responsible.

our_states

array, holds the states of all the people for which the current process is responsible.

our_num_days_infected

array, used to keep track of the number of days each person has been infected for which the current process is responsible.

3.3 const_t struct

```
// Data being used as constant
struct const_t
{
    // environment
    int environment_width;
    int environment_height;
    // disease
    int infection_radius;
    int duration_of_disease;
    int contagiousness_factor;
    int deadliness_factor;
    // time
    int total_number_of_days;
    int microseconds_per_day;
};
```

environment_width

indicates how wide the environment is; used to draw the environment and to make sure people stay within the bounds of the environment.

environment_height

indicates how high the environment is; used to draw the environment and to make sure people stay within the bounds of the environment.

infection_radius

see the Introduction Chapter above. The value of this variable can be specified on the command line with the –d option.

duration_of_disease

see the Introduction Chapter above. The value of this variable can be specified on the command line with the –T option.

contagiousness_factor

see the Introduction Chapter above. The value of this variable can be specified on the command line with the –c option.

deadliness_factor

see the Introduction Chapter above. The value of this variable can be specified on the command line with the –D option.

total_number_of_days

the total number of days over which to run the simulation.

microseconds_per_day

used to tell how many microseconds to freeze in between frames of animation. The value of this variable can be specified on the command line with the –m option.

3.4 stats_t struct

```
// Stats data private to each node: Data being used by
// each process on a node in a cluster when using MPI.
// Each process holds stats data for a subset of people.
struct stats_t
```

```
{
    double our_num_infections;
    double our_num_infection_attempts;
    double our_num_deaths;
    double our_num_recovery_attempts;
};
```

our_num_infections

used to count the number of actual infections that take place (in which an infected person transmits the disease to a susceptible person). Only used if the showing of results is enabled (i.e., if the program is to print out final results from the simulation). Used to determine the actual contagiousness of the disease, which can be compared to the contagiousness factor by the user.

our_num_infection_attempts

used to count the number of times a susceptible person is within an infection radius of an infected person, even if the infection fails. Only used if the showing of results is enabled (i.e., if the program is to print out final results from the simulation). Used to determine the actual contagiousness of the disease, which can be compared to the contagiousness factor by the user.

our_num_deaths

used to count the number of times a person dies. Only used if the showing of results is enabled (i.e., if the program is to print out final results from the simulation). Used to determine the actual deadliness of the disease, which can be compared to the deadliness factor by the user.

our_num_recovery_attempts

used to count the number of times an infected person is able to become immune. Only used if the showing of results is enabled (i.e., if the program is to print out final results from the simulation). Used to determine the actual deadliness of the disease, which can be compared to the deadliness factor by the user.

3.5 display_t struct

```
// Data being used for the X display
struct display_t
{
    #ifdef TEXT_DISPLAY
    // Array of character arrays for text display
    char **environment;
    #endif

    #ifdef X_DISPLAY
    // Declare X-related variables
    Display         *display;
    Window          window;
    int             screen;
    Atom            delete_window;
    GC              gc;
    XColor          infected_color;
    XColor          immune_color;
    XColor          susceptible_color;
    XColor          dead_color;
    Colormap        colormap;
    char            *red;
    char            *green;
    char            *black;
    char             *white;
    #endif
};
```

environment

2D array, holds an ASCII representation of the environment (see “state” under “Person” in the Introduction Chapter). This variable is used only when we are using Text Display.

display

Display, display pointer for the connection to the X server

window

Window, variable to holds the window id.

screen

Screen, variable to holds default screen

delete_window

gc

infected_color

immune_color

susceptible_color

dead_color

red

array of char, holds value #FF0000, which is the hex code for color red.

green

array of char, holds value #00FF00, which is the hex code for color green.

black

array of char, holds value #000000, which is the hex code for color black.

white

array of char, holds value #FFFFFF, which is the hex code for color white.

colormap

CHAPTER

FOUR

INITIALIZE FUNCTIONS

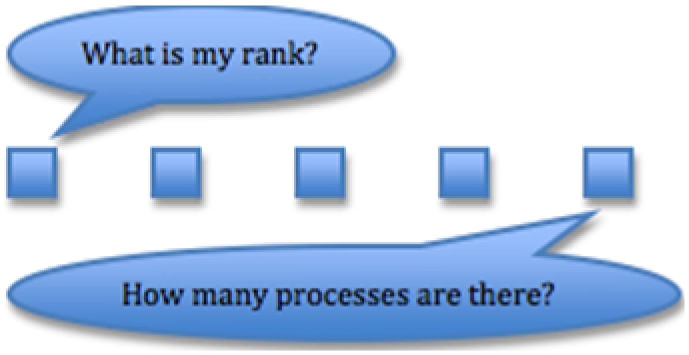
4.1 init()

This function will first initialize variables in the constant structure with default values. It will also initialize **total_number_of_people** variable, **total_num_initially_infected** variable and **total_num_infected** variable. After this, it will set all the counters inside stats structure to zero, as well as state counters inside global struct.

Before executing the algorithm, the code starts by initializing MPI using

```
// Each process initializes the distributed memory environment
MPI_Init(&argc, &argv);
```

We pass the addresses of the arguments to **main**, **argc** and **argv**, so that MPI can strip out anything from the command line related to MPI, such as **mpirun** or **–np**. **MPI_Init** must be called before any other MPI functions are executed, and we also want to call it before we parse the rest of the command line arguments in **parse_args()** function.



Here we see one process figuring out its rank. It does so by calling

```
#ifdef _MPI
MPI_Comm_rank(MPI_COMM_WORLD, &our->our_rank);
```

function. This function checks the MPI "world" (the "communicator" of all the MPI processes, MPI_COMM_WORLD). You pass the address of the variable for the process's rank to the function as the second argument using the ampersand (&).

If we only have 1 process total (i.e., if we are not using distributed memory), then the rank of the process will be 0, which we set in the code as **our_rank = 0**.

```
#else
our->our_rank = 0;
```

We also see another process figuring out how many processes there are. It does so by calling

```
#ifdef _MPI
MPI_Comm_size(MPI_COMM_WORLD, &global->total_number_of_processes);
```

function. Just as with **MPI_Comm_rank**, you pass the communicator of all the processes and the address of the variable for the number of processes.

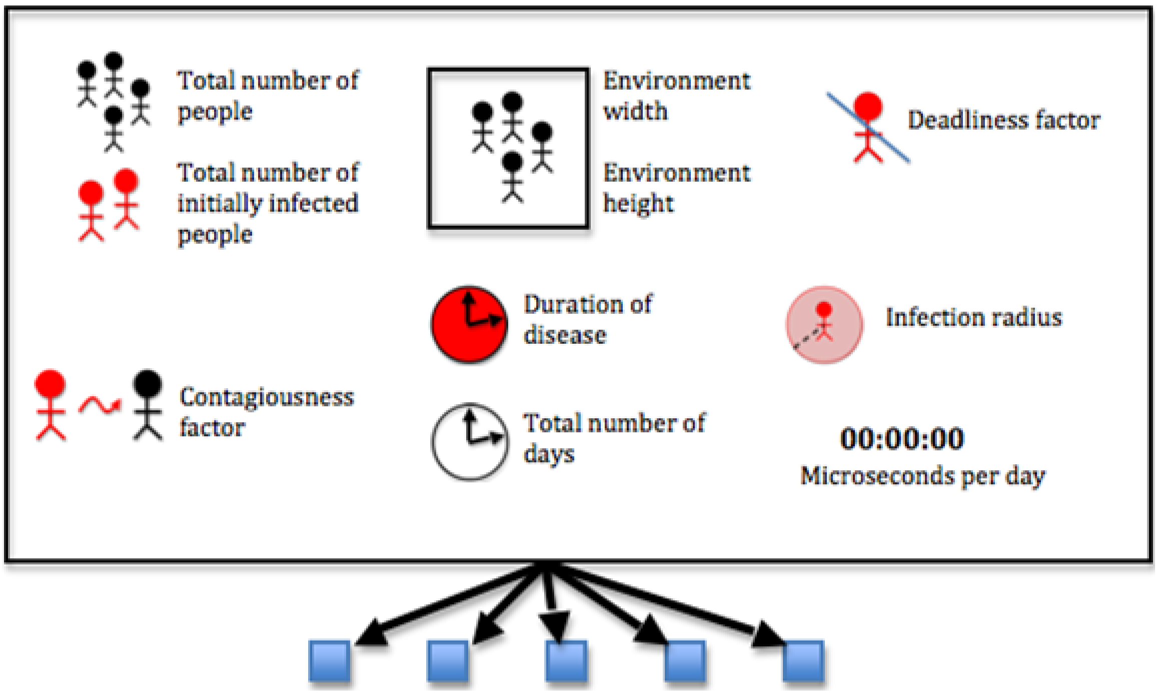
If we have only 1 process total, we set the number of processes by calling total_number_of_processes = 1.

```
#else
global->total_number_of_processes = 1;
```

After MPI initialization, **init**() function will call the following five functions.

```
init_check(global);
parse_args(global, constant, argc, argv);
allocate_array(global, our, constant, dpy);
init_array(our, constant);
// if use X_DISPLAY, do init_display()
#ifdef X_DISPLAY
    init_display(our, constant, dpy);
#endif
```

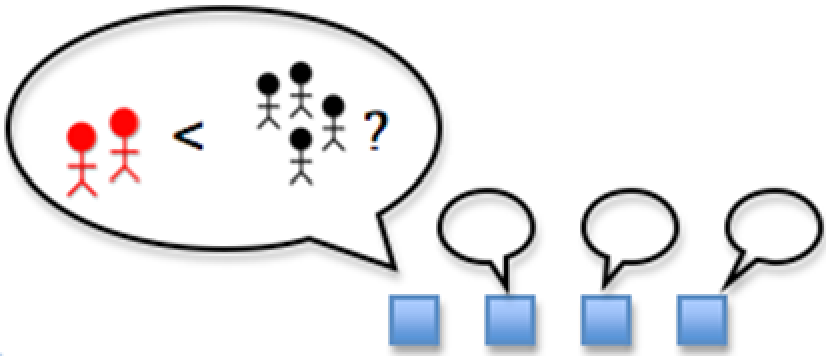
4.2 parse_args()



These parameters are specified via command line arguments when the program is run. Otherwise, default values are used. The code uses **getopt** function to do this. Type **man 3 getopt** in a shell if you are interested how it works.

4.3 init_check()

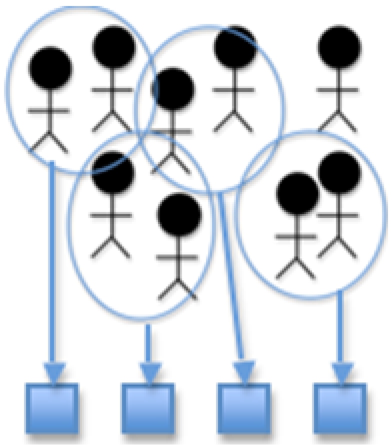
This function makes sure that for each process, the total number of initially infected people is less than the total number of people



The simulation can't run if there are more initially infected people than there are people. If there are, the code uses the fprintf function to print an error message to standard error, and it exits the program with exit code -1.

4.4 find_size()

For each process, this function determines the number of people for which it is responsible

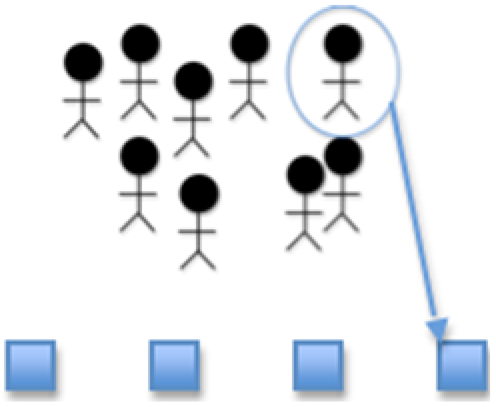


Each process will try to take an even split of the number of people. It does so by dividing the number of people by the total number of processes and throwing away any remainder. Because the variables involved are integers in C, the throwing away of the remainder is handled automatically in the division

```
our->our_number_of_people = total_number_of_people / total_number_of_processes;
```

The last process is responsible for the remainder

```
// The last process is responsible for the remainder
if(our_rank == total_number_of_processes - 1)
{
    our->our_number_of_people += total_number_of_people % total_number_of_processes;
}
```



Every person has to be accounted for, so any remainder of the division is assigned to the last process. We can obtain the remainder by using the modulo operator (%), and we add it to the existing value using the plus-equals operator (+=):

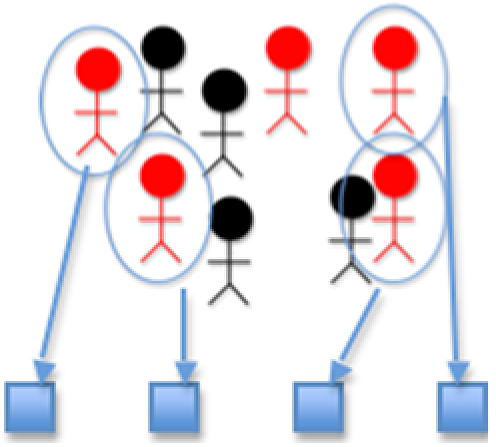
```
        our->our_number_of_people += total_number_of_people % total_number_of_processes;
```

We only want the last process to do this, so we surround the code with

```
    if(our_rank == total_number_of_processes - 1)
```

since the last process has rank **N–1**, where N is the total number of processes.

Each process determines the number of initially infected people for which it is responsible



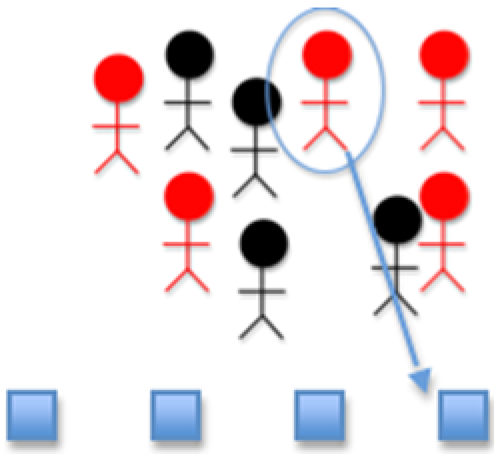
```
    our->our_num_initially_infected = total_num_initially_infected
    / total_number_of_processes;
```

This is the same method used before, but it considers only the infected people.

The last process is responsible for the remainder

```
        our->our_num_initially_infected += total_num_initially_infected
        % total_number_of_processes;
```

This is the same method used before, but it considers only the infected people.



4.5 allocate_array

At this point we are ready to allocate our arrays, which must be performed before we can start filling the arrays. Allocating an array means reserving enough space in memory for it; if we don't reserve the space the program will assume that it is a zero-length array. The allocation must happen in the "heap" memory, meaning we must allocate it dynamically (i.e. as the program is running). To allocate memory on the heap, we use the **malloc** function, which takes the amount of space that is requested and returns a pointer to the newly allocated memory, which we can then use as an array. Let's see an example with the x_locations array:

```
global->x_locations = (int*)malloc(total_number_of_people * sizeof(int));
```

Here we see that malloc has taken an argument, **total_number_of_people * sizeof(int)**. This is how we specify that we want to fill the array with a certain number of integers, namely the amount stored in the **total_number_of_people** variable. We also need to specify how big these integers are, for which we use the **sizeof(int)** function. We then take the return from **malloc** and tell the program to "cast" it (i.e. use it) as a pointer to integers, for which we use **(int*)**. This is then assigned to x_locations, and we can now use **x_locations** as an array.

For the 2D array **environment**, we must allocate not only the array itself but also each of the arrays that it contains (since a 2D array is an array whose elements are arrays). The array has horizontal strips of length **environment_width** and vertical strips of length **environment_height**. We perform the allocation by allocating enough space for the entire array first using

```
dpy->environment = (char**)malloc(constant->environment_width *
    constant->environment_height * sizeof(char*));
```

That is, we are allocating enough **char***'s for **environment_width** times **environment_height**, casting this as a **char**** and assigning it to **environment**. Then we allocate each array within **environment**, like so:

```
for(our_current_location_x = 0;
    our_current_location_x <= constant->environment_width - 1;
    our_current_location_x++)
{
    dpy->environment[our_current_location_x] = (char*)malloc(
        constant->environment_height * sizeof(char));
}
```

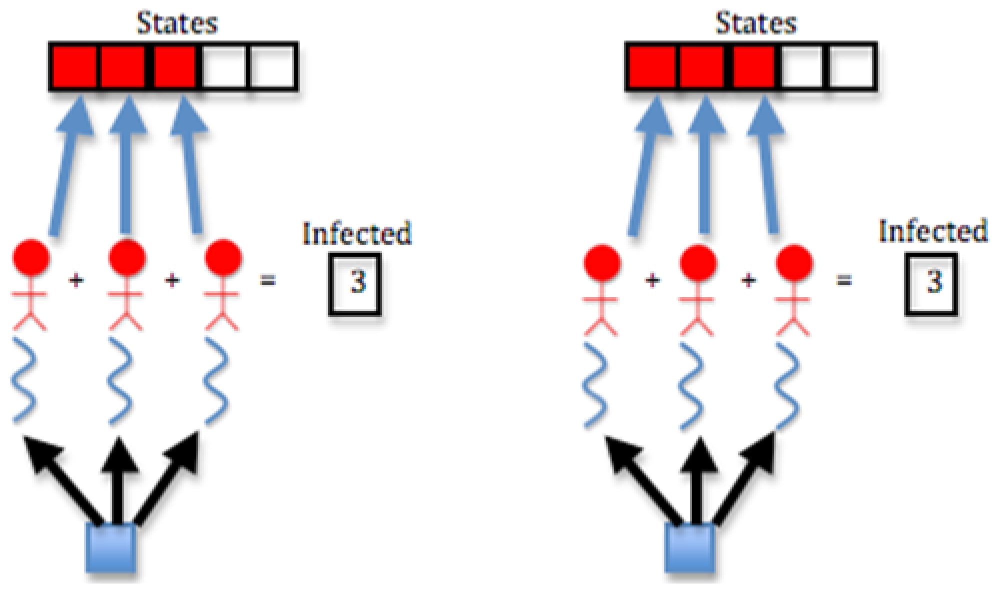
The number of arrays we need is stored in **environment_width**, so we loop from **0** to **environment_width – 1** and allocate enough space in each element of environment for **environment_height** chars, each one of which has size **sizeof(char)**.

This can be a convoluted process but is necessary for allocating arrays dynamically, which allows us to specify options on the command line (so we don't have to edit the source code and re-compile each time we want to run a simulation with different parameters).

4.6 init_array()

This function can be divided into four parts.

The function sets the states of the initially infected people and set the count of its infected people



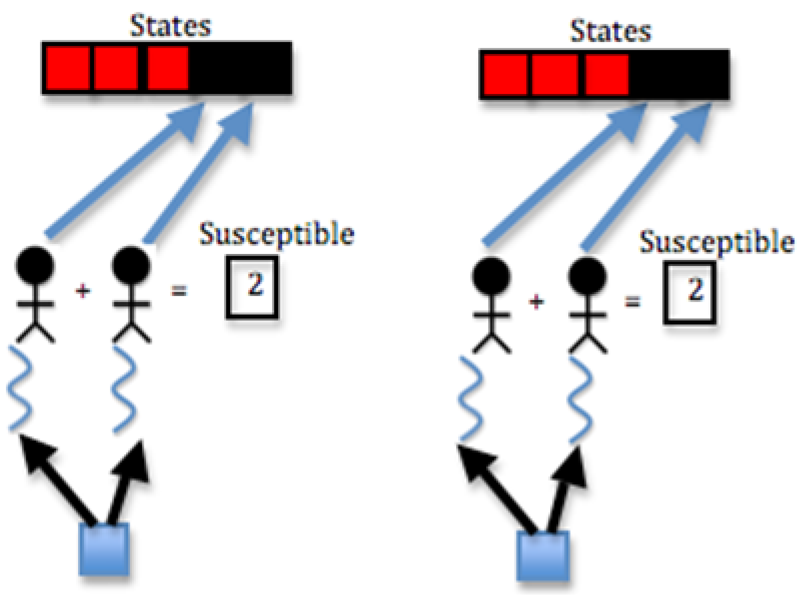
The function also sets the states of infected people using the **our_states** array. They fill the first **our_num_initially_infected** cells of the array with the **INFECTED** constant; i.e. they fill in the **0** through **our_num_initially_infected – 1** positions of the array with **INFECTED** as below:

```
// Each process spawns threads to set the states of the initially
// infected people and set the count of its infected people
for(my_current_person_id = 0; my_current_person_id
    <= our_num_initially_infected - 1; my_current_person_id++)
{
    our->our_states[my_current_person_id] = INFECTED;
    our->our_num_infected++;
}
```

The function sets the states of the rest of its people and set the count of its susceptible people

This is similar to last step, but we want to fill the rest of the array (from **our_num_initially_infected** to **our_number_of_people - 1**) with the **SUSCEPTIBLE** constant, and we want to add **1** to the **our_num_susceptible** variable at each iteration of the loop:

```
// Each process spawns threads to set the states of the rest of
// its people and set the count of its susceptible people
for(my_current_person_id = our_num_initially_infected;
    my_current_person_id <= our_number_of_people - 1;
    my_current_person_id++)
{
    our->our_states[my_current_person_id] = SUSCEPTIBLE;
```



```
        our->our_num_susceptible++;
    }
```

The **our_states** array is now full; the first **our_num_initially_infected** cells have the **INFECTED** constant, and the rest have the **SUSCEPTIBLE** constant.

The third step is that the function sets random x and y locations for each of its people

Locations of people are stored in the **our_x_locations** and **our_y_locations** arrays. To fill these arrays with random values, we use a for loop and the random function:

```
    // Each process spawns threads to set random x and y locations for
    // each of its people
    for(my_current_person_id = 0;
        my_current_person_id <= our_number_of_people - 1;
        my_current_person_id++)
    {
        our->our_x_locations[my_current_person_id] = random() % constant->environment_width;
        our->our_y_locations[my_current_person_id] = random() % constant->environment_height;
    }
```

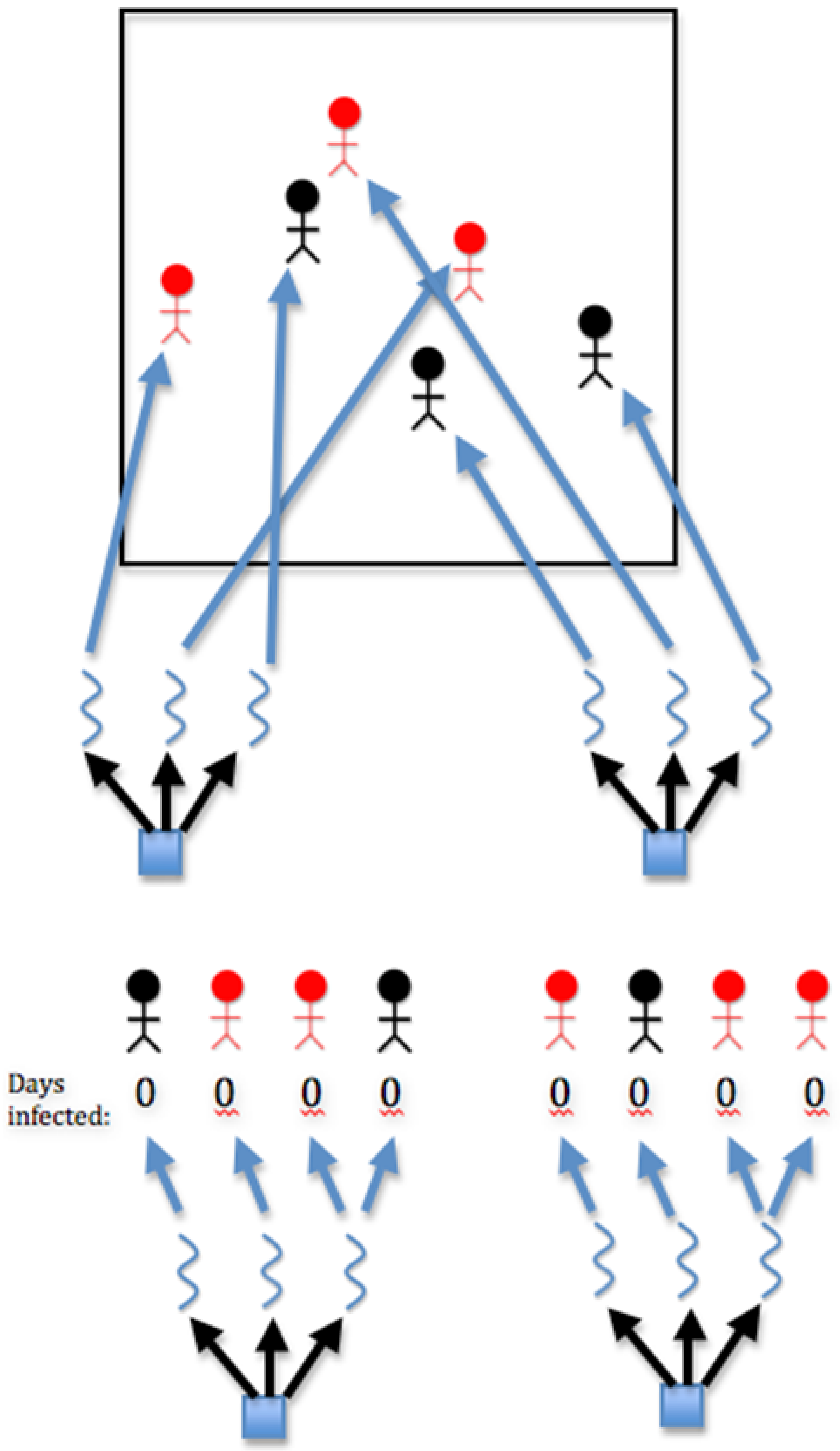
By calling random with the **modulus (%)** operator, we can restrict the size of the random number it generates. Since we cannot have x locations larger than the width of the environment, we call **random() % environment_width**; to make sure the **x location** of each person is less than **environment_width**. Similarly for the **y location** and **environment_height**.

We are filling the x and y location arrays for all of the people for which the process is responsible, so we loop from **0** to **our_number_of_people – 1**.

Finally, the function initializes the number of days infected of each of its people to 0

The number of days each person is infected is stored in the **our_num_days_infected** array, so we loop over all of the people and fill this array with 0, since the simulation starts at day 0, at which point no days have yet elapsed:

```
    // Each process spawns threads to initialize the number of days
    // infected of each of its people to 0
    for(my_current_person_id = 0;
        my_current_person_id <= our_number_of_people - 1;
```


Days
infected:
0
0
0
0
0
0
0
0

```
        my_current_person_id++)
    {
        our->our_num_days_infected[my_current_person_id] = 0;
    }
```

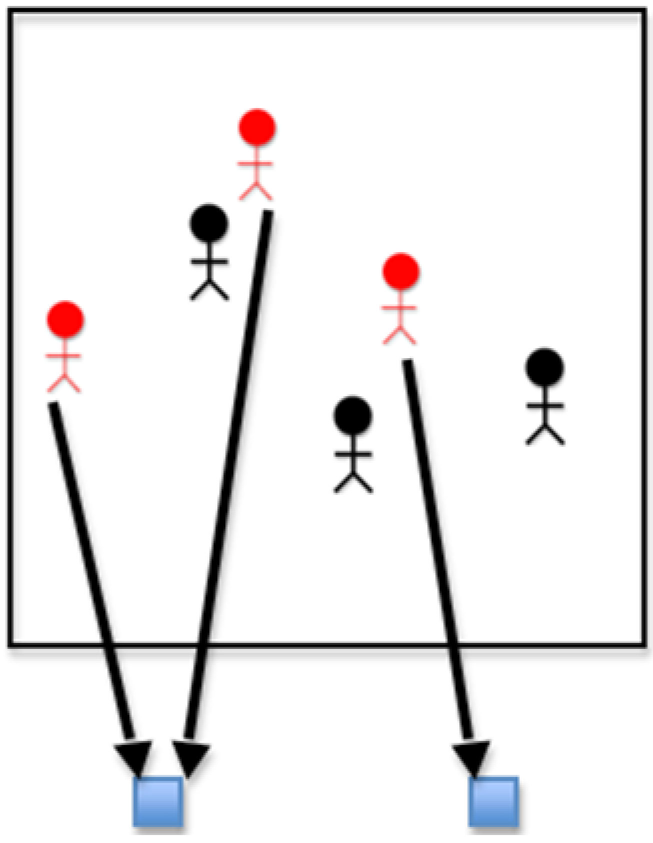
CHAPTER

FIVE

INFECTION FUNCTIONS

5.1 find_infected

For each process, this function determines its infected x locations and infected y locations

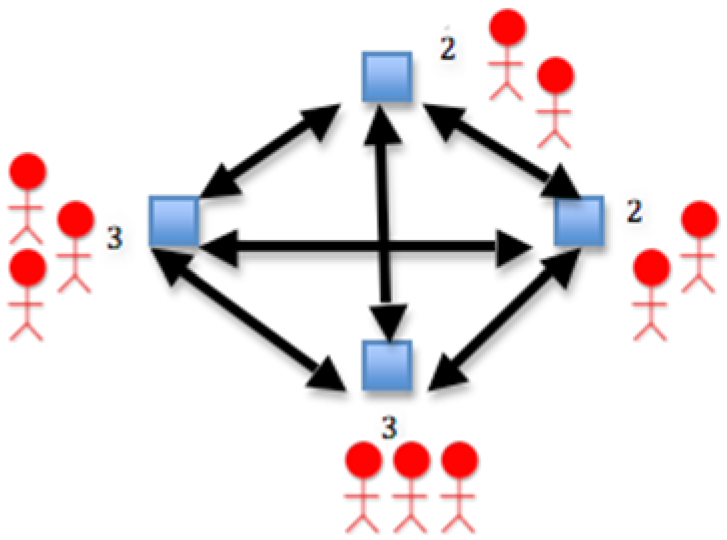


We have already set the states of the infected people and the positions of all the people, but we need to specifically set the positions of the infected people and store them in the **our_infected_x_locations** and **our_infected_y_locations** arrays. We do this by marching through the **our_states** array and checking whether the state at each cell is **INFECTED**. If it is, we add the locations of the current infected person from the **our_x_locations** and **our_y_locations** arrays to the **our_infected_x_locations** and **our_infected_y_locations** arrays. We determine the ID of the current infected person using the **our_current_infected_person** variable:

```
for(our_person1 = 0; our_person1 <= our->our_number_of_people - 1; our_person1++)
{
    if(our->our_states[our_person1] == INFECTED)
    {
        our->our_infected_x_locations[our_current_infected_person] =
        our->our_x_locations[our_person1];
        our->our_infected_y_locations[our_current_infected_person] =
        our->our_y_locations[our_person1];
        our_current_infected_person++;
    }
}
```

5.2 share_infected

First, for each process, this function sends its count of infected people to all the other processes and receives their counts



```
// Each process sends its count of infected people to all the
// other processes and receives their counts
MPI_Allgather(&our->our_num_infected, 1, MPI_INT, recvcounts, 1,
    MPI_INT, MPI_COMM_WORLD);
```

This step is handled by the MPI command **MPI_Allgather** whose arguments are as follows:

- **&our_num_infected** – the address of the sending buffer (the thing being sent).
- **1** – the count of things being sent.
- **MPI_INT** – the datatype of things being sent.
- **recvcounts** – the receive buffer (an array of things being received).
- **1** – the count of things being received.
- **MPI_INT** – the datatype of things being received.
- **MPI_COMM_WORLD** – the communicator of processes that send and receive data.

Once the data has been sent and received, we count the total number of infected people by adding up the values in the **recvcounts** array and storing the result in the **total_num_infected** variable:

```
global->total_num_infected = 0;
int current_rank;
for(current_rank = 0; current_rank <= total_number_of_processes - 1;
    current_rank++)
{
    global->total_num_infected += recvcounts[current_rank];
}
```

Next, for each process, the function sends the x locations of its infected people to all the other processes and receives the x locations of their infected people

For this send and receive, we need to use **MPI_Allgatherv** instead of **MPI_Allgather**. This is because each process has a varying number of infected people, so it needs to be able to send a variable number of x locations. To do this, we first need to set up the displacements in the receive buffer; that is, we need to indicate how many elements each process will send and at what points in the receive array they will appear. We can do this with a **displs** array, which will contain a list of the displacements in the receive buffer:

```
// Each process sends the x locations of its infected people to
// all the other processes and receives the x locations of their
// infected people
MPI_Allgatherv(our->our_infected_x_locations, our->our_num_infected, MPI_INT,
    global->their_infected_x_locations, recvcounts, displs,
    MPI_INT, MPI_COMM_WORLD);
```

We are now ready to call the **MPI_Allgatherv**. Here are its arguments:

- **our_infected_x_locations** – the send buffer (array of things to send).
- **our_num_infected** – the count of elements in the send buffer.
- **MPI_INT** – the datatype of the elements in the send buffer.
- **their_infected_x_locations** – the receive buffer (array of things to receive).
- **recvcounts** – an array of counts of elements in the receive buffer
- **displs** – the list of displacements in the receive buffer, as determined above.
- **MPI_INT** – the data type of the elements in the receive buffer.
- **MPI_COMM_WORLD** – the communicator of processes that send and receive data.

Once the command is complete, each process will have the full array of the x locations of the infected people from each process, stored in the **their_infected_x_locations** array.

Finally, each process sends the y locations of its infected people to all the other processes and receives the y locations of their infected people

```
// Each process sends the y locations of its infected people
// to all the other processes and receives the y locations of their
// infected people
MPI_Allgatherv(our->our_infected_y_locations, our->our_num_infected, MPI_INT,
    global->their_infected_y_locations, recvcounts, displs,
    MPI_INT, MPI_COMM_WORLD);
```

The y locations are sent and received just as the x locations are sent and received. In fact, the function calls have exactly 2 letters difference; the x's in the **Allgatherv** from last step. are replaced by y's in the **Allgatherv** in this step.

Note that the code will only execute previous two steps if MPI is enabled. If it is not enabled, the code simply copies the **our_infected_x_locations** and **our_infected_y_locations** arrays into the **their_infected_x_locations** and

their_infected_y_locations arrays and the **our_num_infected** variable into the **total_num_infected** variable.

CHAPTER

SIX

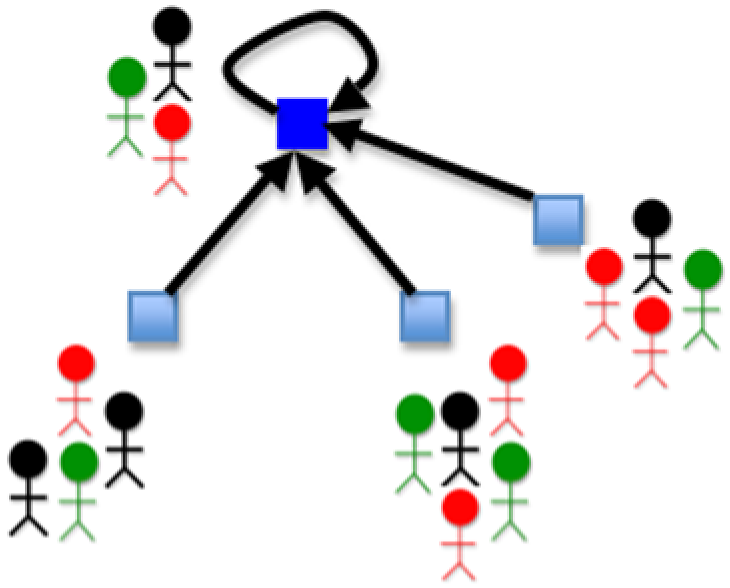
DISPLAY FUNCTIONS

6.1 init_display

Rank 0 initializes the graphics display. The code uses X to handle graphics display.

6.2 share_location

If display is enabled, Rank 0 gathers the states, x locations, and y locations of the people for which each process is responsible



We set up the displs here just as we did in function **share_infected()**.

```
// Distributed Memory Information
int *recvcounts;
int *displs;
recvcounts = (int*)malloc(total_number_of_processes * sizeof(int));
displs = (int*)malloc(total_number_of_processes * sizeof(int));

// Set up the receive counts and displacements in the
// receive buffer (see the man page for MPI_Gatherv)
```

```
    int current_displ = 0;
    int current_rank;
    for(current_rank = 0; current_rank <= total_number_of_processes - 1;
       current_rank++)
    {
        displs[current_rank] = current_displ;
        recvcounts[current_rank] = total_number_of_people / total_number_of_processes;
        if(current_rank == global->total_number_of_processes - 1)
        {
            recvcounts[current_rank] += total_number_of_people
            % total_number_of_processes;
        }
        current_displ += recvcounts[current_rank];
    }
```

Three calls to Gatherv take place for each process to send each of their **our_states**, **our_x_locations**, and **our_y_locations arrays**. Rank 0 copies these into its **states**, **x_locations**, and **y_locations** arrays, respectively.

```
    MPI_Gatherv(our->our_states, our->our_number_of_people, MPI_CHAR,
        global->states, recvcounts, displs, MPI_CHAR, 0, MPI_COMM_WORLD);
    MPI_Gatherv(our->our_x_locations, our->our_number_of_people, MPI_INT,
        global->x_locations, recvcounts, displs, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Gatherv(our->our_y_locations, our->our_number_of_people, MPI_INT,
        global->y_locations, recvcounts, displs, MPI_INT, 0, MPI_COMM_WORLD);
```

Note that if MPI is not enabled, Rank 0 just does a direct copy of the arrays without using Gatherv.

```
    int my_current_person_id;
    for(my_current_person_id = 0; my_current_person_id
       <= global->total_number_of_people - 1; my_current_person_id++)
    {
       global->states[my_current_person_id]
       = our->our_states[my_current_person_id];
       global->x_locations[my_current_person_id]
       = our->our_x_locations[my_current_person_id];
       global->y_locations[my_current_person_id]
       = our->our_y_locations[my_current_person_id];
    }
    #endif
}
```

6.3 do_display

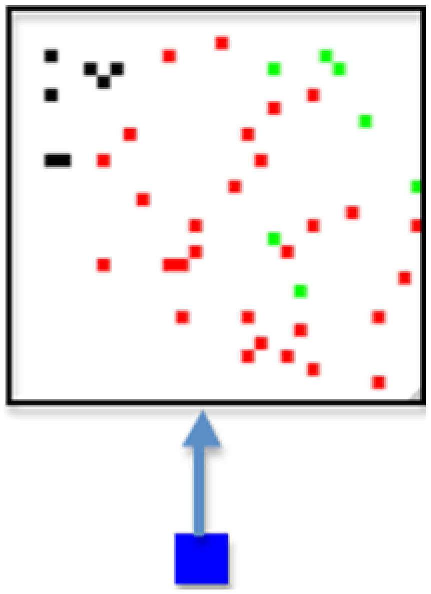
If display is enabled, Rank 0 displays a graphic of the current day

6.4 close_display

If X display is enabled, then Rank 0 destroys the X Window and closes the display

6.5 throttle

In order for better display, we wait between frames of animation.



CHAPTER

SEVEN

CORE FUNCTIONS

7.1 move()

For of the each process's people, this function moves them around randomly.

For everyone handled by this process,

```
for(my_current_person_id = 0; my_current_person_id
    <= our->our_number_of_people - 1; my_current_person_id++)
```

If the person is not dead, then

```
    if(our_states[my_current_person_id] != DEAD)
```

First, The thread randomly picks whether the person moves left or right or does not move in the x dimension.

The code uses (random() % 3) - 1; to achieve this. (random() % 3) returns either 0, 1, or 2. Subtracting 1 from this produces -1, 0, or 1. This means the person can move to the right, stay in place (0), or move to the left (-1).

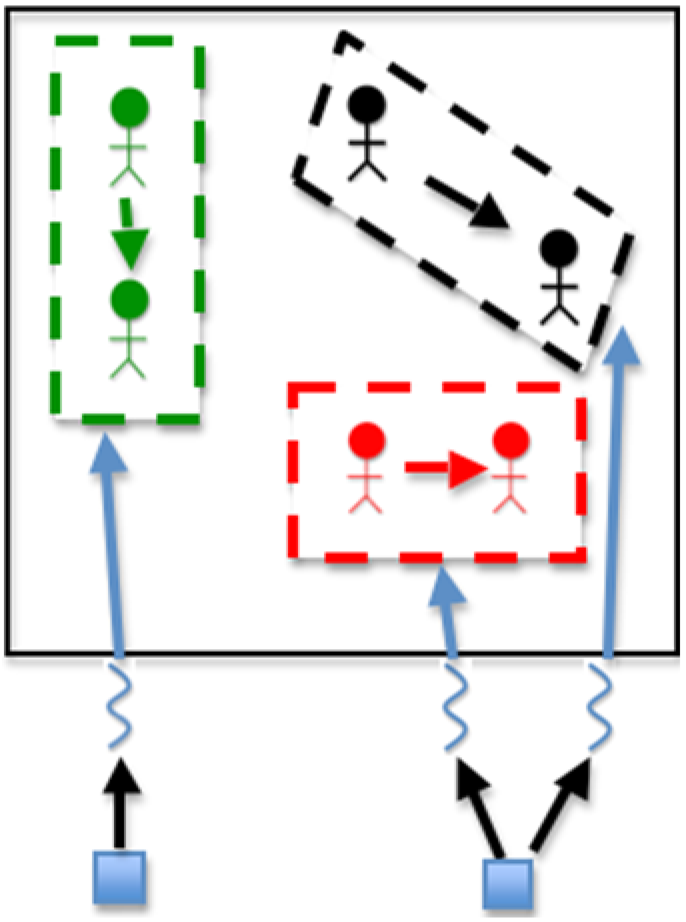
```
        // The thread randomly picks whether the person moves left
        // or right or does not move in the x dimension
        my_x_move_direction = (random() % 3) - 1;
```

The thread then randomly picks whether the person moves up or down or does not move in the y dimension. This is similar to movement in x dimension.

```
        // The thread randomly picks whether the person moves up
        // or down or does not move in the y dimension
        my_y_move_direction = (random() % 3) - 1;
```

Next, we need to make sure that the person will remain in the bounds of the environment after moving. We check this by making sure the person's x location is greater than or equal to 0 and less than the width of the environment and that the person's y location is greater than or equal to 0 and less than the height of the environment. In the code, it looks like this:

```
        if( (our_x_locations[my_current_person_id]
                + my_x_move_direction >= 0) &&
            (our_x_locations[my_current_person_id]
                + my_x_move_direction < environment_width) &&
            (our_y_locations[my_current_person_id]
                + my_y_move_direction >= 0) &&
            (our_y_locations[my_current_person_id]
                + my_y_move_direction < environment_height) )
```



Finally, The thread moves the person

The thread is able to achieve this by simply changing values in the **our_x_locations** and **our_y_locations** arrays.

```
// The thread moves the person
our_x_locations[my_current_person_id] += my_x_move_direction;
our_y_locations[my_current_person_id] += my_y_move_direction;
```

7.2 susceptible()

For of the each process's people, this function handles those that are ssusceptible by deciding whether or not they should be marked infected.

For everyone handled by this process,

```
for(my_current_person_id = 0; my_current_person_id
      <= our->our_number_of_people - 1; my_current_person_id++)
```

If the person is susceptible,

```
if(our_states[my_current_person_id] == SUSCEPTIBLE)
```

For each of the infected people (received earlier from all processes) or until the number of infected people nearby is 1, the thread does the following

```
for(my_person2 = 0; my_person2 <= total_num_infected - 1
    && my_num_infected_nearby < 1; my_person2++)
```

If this person is within the infection radius,

```
if((our_x_locations[my_current_person_id]
    > their_infected_x_locations[my_person2] - infection_radius) &&
   (our_x_locations[my_current_person_id]
    < their_infected_x_locations[my_person2] + infection_radius) &&
   (our_y_locations[my_current_person_id]
    > their_infected_y_locations[my_person2] - infection_radius) &&
   (our_y_locations[my_current_person_id]
    < their_infected_y_locations[my_person2] + infection_radius))
```

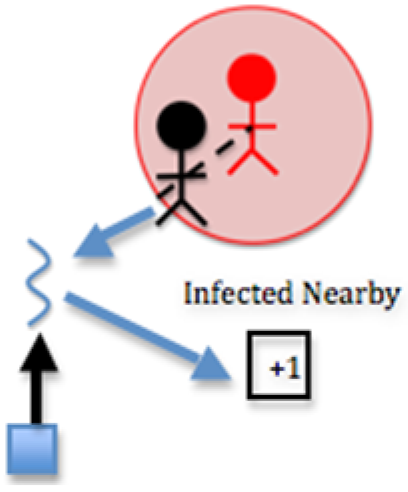
then, the function increments the number of infected people nearby

```
my_num_infected_nearby++;
```

This is where a large chunk of the algorithm's computation occurs. Each susceptible person must be computed with each infected person to determine how many infected people are nearby each person. Two nested loops means many computations. In this step, the computation is fairly simple, however. The thread simply increments the **my_num_infected_nearby** variable.

Note in the code that if the number of infected nearby is greater than or equal to 1 and we have **SHOW_RESULTS** enabled, we increment the **our_num_infection_attempts** variable. This helps us keep track of the number of attempted infections, which will help us calculate the actual contagiousness of the disease at the end of the simulation.

```
// The thread updates stats counter
#ifdef SHOW_RESULTS
if(my_num_infected_nearby >= 1)
    stats->our_num_infection_attempts++;
#endif
```



If there is at least one infected person nearby, and a random number less than 100 is less than or equal to the contagiousness factor, then

```
if(my_num_infected_nearby >= 1 && (random() % 100)
    <= contagiousness_factor)
```

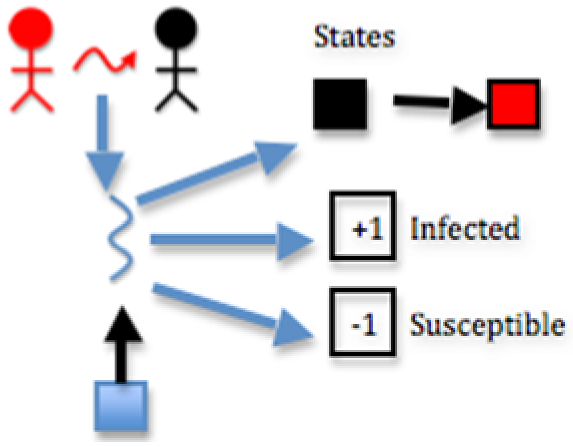
Recall that the contagiousness factor is the likelihood that the disease will be spread. We measure this as a number less than 100. For example, if there is a 30% chance of contagiousness, we use 30 as the value of the contagiousness factor. To figure out if the disease is spread for any given interaction of people, we find a random number less than 100 and check if it is less than or equal to the contagiousness factor, because this will be equivalent to calculating the odds of actually spreading the disease (e.g. there is a 30% chance of spreading the disease and also a 30% chance that a random number less than 100 will be less than or equal to 30).

The thread changes this person state to infected

```
// The thread changes person1's state to infected
our_states[my_current_person_id] = INFECTED;
```

The thread updates the counters

```
// The thread updates the counters
our->our_num_infected++;
our->our_num_susceptible--;
```



Note in the code that if the infection succeeds and we have **SHOW_RESULTS** enabled, we increment the

our_num_infections variable. This helps us keep track of the actual number of infections, which will help us calculate the actual contagiousness of the disease at the end of the simulation.

```
                // The thread updates stats counter
                #ifdef SHOW_RESULTS
                stats->our_num_infections++;
                #endif
```

7.3 infected()

For of the each process's people, this function to handles those that are infected by deciding whether they should be marked immune or dead.

For everyone handled by this process,

```
    for(my_current_person_id = 0; my_current_person_id
        <= our->our_number_of_people - 1; my_current_person_id++)
```

If the person is infected and has been for the full duration of the disease, then

```
        if(our_states[my_current_person_id] == INFECTED
            && our_num_days_infected[my_current_person_id] == duration_of_disease)
```

Note in the code that if we have **SHOW_RESULTS** enabled, we increment the **our_num_recovery_attempts** variable. This helps us keep track of the number of attempted recoveries, which will help us calculate the actual deadliness of the disease at the end of the simulation.

```
            // The thread updates stats counter
            #ifdef SHOW_RESULTS
                stats->our_num_recovery_attempts++;
            #endif
```

If a random number less than 100 is less than the deadliness factor,

```
            if((random() % 100) < deadliness_factor)
```

then, the thread changes the person's state to dead

```
                our_states[my_current_person_id] = DEAD;
```

and then the thread updates the counters

```
                // The thread updates the counters
                our->our_num_dead++;
                our->our_num_infected--;
```

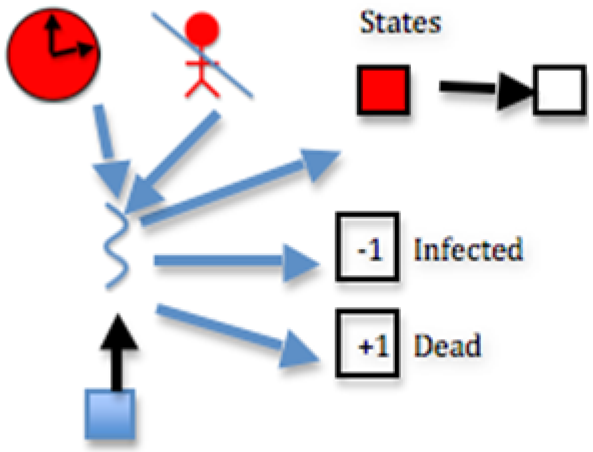
This step is effectively the same as function susceptible, considering deadliness instead of contagiousness. The difference here is the following step:

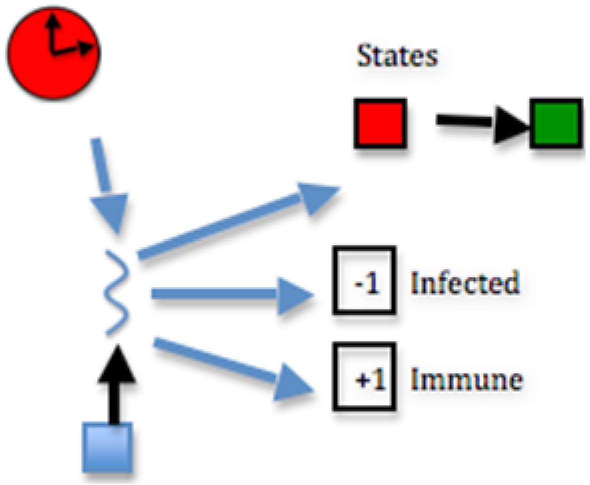
if a random number less than 100 is less than the deadliness factor, the thread changes the person's state to immune

```
                // The thread changes the person's state to immune
                our_states[my_current_person_id] = IMMUNE;
```

The thread updates the counters

```
                // The thread updates the counters
                our->our_num_immune++;
                our->our_num_infected--;
```


States
-1 Infected
+1 Dead


States
-1 Infected
+1 Immune

If deadliness fails, then immunity succeeds.

Note in the code that if the person dies and we have **SHOW_RESULTS** enabled, we increment the **our_num_deaths** variable. This helps us keep track of the actual number of deaths, which will help us calculate the actual deadliness of the disease at the end of the simulation.

```
// The thread updates stats counter
#ifdef SHOW_RESULTS
    stats->our_num_deaths++;
#endif
```

7.4 update_days_infected()

For of the each process's people, this function to handles those that are infected by increasing the number of days infected.

For everyone handled by this process,

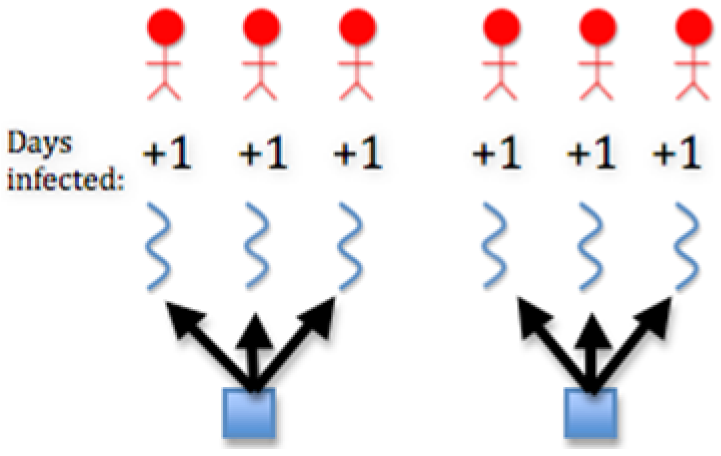
```
for(my_current_person_id = 0; my_current_person_id
    <= our->our_number_of_people - 1; my_current_person_id++)
```

If the person is infected,

```
if(our_states[my_current_person_id] == INFECTED)
```

then, the function increment the number of days the person has been infected

```
our_num_days_infected[my_current_person_id]++;
```



CHAPTER

EIGHT

FINISH FUNCTIONS

8.1 show_results()

At the end of the code, if we are choosing to show results, we print out the final counts of susceptible, infected, immune, and dead people. We also print the actual contagiousness and actual deadliness of the disease. To perform these two calculations, we use the following code (using the contagiousness as the example):

```
#ifdef SHOW_RESULTS
printf("Rank %d final counts: %d susceptible, %d infected, %d immune, %d dead \nRank %d actual co
    our->our_rank, our->our_num_susceptible, our->our_num_infected,
    our->our_num_immune, our->our_num_dead, our->our_rank,
    100.0 * (stats->our_num_infections / (stats->our_num_infection_attempts
        == 0 ? 1 : stats->our_num_infection_attempts)),our->our_rank,
    100.0 * (stats->our_num_deaths / (stats->our_num_recovery_attempts
        == 0 ? 1 : stats->our_num_recovery_attempts)));
#endif
```

In this code, the ternary operators (? and :) are used to return one value if something is true and another value if it isn't. The thing we are checking for truth is **our_num_infection_attempts == 0**. If this is true, i.e. if we didn't attempt any infection attempts at all, then we say there was actually 1 infection attempt (this is to avoid a divide by zero error). Otherwise, we return the actual number of infection attempts. This value becomes the dividend for **our_num_infections**; in other words, we divide the number of actual infections by the number of total infections. This will give us a value less than 1, so we multiply it by 100 to obtain the actual contagiousness factor of the disease. A similar procedure is performed to calculate the actual deadliness factor.

8.2 cleanup

If X display is enabled, then Rank 0 destroys the X Window and closes the display

```
// if use X_DISPLAY, do close_display()
#ifdef X_DISPLAY
close_display(our, dpy);
#endif
```

Since we allocated our arrays dynamically, we need to release them back to the heap using the **free** function. We do this in the reverse order than we used **malloc**, so **environment** will come first and **x_locations** will come last.

```
// free text display environment
#ifdef TEXT_DISPLAY
int our_current_location_x;
for(our_current_location_x = constant->environment_width - 1;
        our_current_location_x >= 0; our_current_location_x--)
    {
        free(dpy->environment[our_current_location_x]);
    }
    free(dpy->environment);
    #endif

    // free arrays allocated in our struct
    free(our->our_num_days_infected);
    free(our->our_states);
    free(our->our_infected_y_locations);
    free(our->our_infected_x_locations);
    free(our->our_y_locations);
    free(our->our_x_locations);

    // free arrays allocated in global struct
    free(global->states);
    free(global->their_infected_x_locations);
    free(global->their_infected_y_locations);
    free(global->y_locations);
    free(global->x_locations);
```

Just as we initialized the MPI environment with **MPI_Init**, we must finalize it with **MPI_Finalize()**. No MPI functions are allowed to occur after **MPI_Finalize**.

```
    #ifdef _MPI
        // MPI execution is finished; no MPI calls are allowed after this
        MPI_Finalize();
    #endif
```

CHAPTER
NINE

BUILD AND RUN THE PARALLEL VERSION

When you create the executable for this program, you will need to set some flags that are particular for your machine, particularly if you want to run it with the graphical display, which uses the X11 library. This should work on linux machines and Mac OS X machines that have X11 installed.

Lines 13-15 in the Makefile, shown below and included with the code, are where you set paths to the X11 library and include directories. On some linux machines you may not need to set either of these, which is why they are commented out.

In this case, lines 13 and 15 are commented because rather than seeing the display, we want to start looking at how the parallel code runs (real code wouldn't use the display for simulation modeling). When rigging the code to test for performance, you really want to eliminate most of the output, so we have just left line 15 uncommented to see the final statistics after the whole simulation is completed.

```
1   # DESCRIPTION:  Makefile for serial codes in C
2   # AUTHOR:       Yu Zhao, Macalester College
3   # DATE:         Original for Area Under A Curve module, September, 2011.
4   #                                 Modified for Infectious Disease module, November, 2011. by Aaron Wee
5   #                                 Modified for Infectious Disease module, July, 2013, by Yu Zhao
6
7   # Code prefix
8   PROGRAM_PREFIX=Pandemic
9
10  # Compilers and flags
11  MPICC=mpicc
12
13  #CFLAGS+=-DTEXT_DISPLAY # Uncomment to show text display
14  #CFLAGS+=-DX_DISPLAY -L/usr/X11R6/lib -lX11 # Uncomment to show X display
15  CFLAGS+=-DSHOW_RESULTS # Uncomment to make the program print its results
16
17  # Source files
18  SRCS=$(PROGRAM_PREFIX).c
19
20  # Make targets
21  all: $(PROGRAM_PREFIX)-mpi
22
23  clean:
24          rm -f $(PROGRAM_PREFIX)-mpi
25
26  run:
27          mpirun -machinefile machines -np 6 ./$(PROGRAM_PREFIX)-mpi
28
```

```
# Make rules
$(PROGRAM_PREFIX)-mpi: $(SRCS)
        $(MPICC) -o $(PROGRAM_PREFIX)-mpi $(SRCS) $(CFLAGS)
```

9.1 Build

```
make
```

9.2 Run

```
mpirun -machinefile machines -np 6 ./Pandemic-mpi
```

Your instructor will provide a machines file for your cluster. You can eliminate the use of the -machinefile machines option if you are running multiple processes on the same machine.

The default values start with a simulation of approximately 50 people.

To see what elements of the computation you can change, try this:

```
./Pandemic-mpi -?
```

It should give you something like this:

```
/Pandemic-mpi -?

Usage: ./Pandemic-mpi [-n total_number_of_people][-i total_num_initially_infected][-w environment_wic
```

Note that these are defined and set in the *parse_args()* function in Initialize.h.

Now you can experiment running different problem sizes with different numbers of threads, like this:

```
mpirun -machinefile machines -np 6 ./Pandemic-mpi -n 70000 -m 0
mpirun -machinefile machines -np 8 ./Pandemic-mpi -n 70000 -m 0
```

9.3 To think about

There are preferable ways to instrument your code to time it, using the MPI function *MPI_Wtime()*. Investigate how to use it and update this code to print running times of various sections of the code. What loop takes the most time?

Can you calculate the speedup you get by using varying numbers of processes for a fairly large problem size?

CHAPTER

TEN

INCLUDING OPENMP

`Download Pandemic-MPI-OMP.zip`

It is really easy to include OpenMP features into existing code we have. All we need to do is to identify all the functions that could use OpenMP. There are in total 5 functions that could use OpenMP to increase performance. The first function is the **init_array()** function in *Initialize.h* file. The next four functions are all the core functions inside *Core.h* file.

10.1 In Initialize.h

10.1.1 init_array()

This function can be divided into four parts: the first part sets the states of the initially infected people and sets the count of infected people. The second part sets states of the rest of the people and sets the of susceptible people. The third part sets random x and y locations for each people. The last part initilize the number of days infected of each people to 0.

Normally, to include OpenMP, all we need is to put **#pragma omp parallel** in front of each of the for loops. However, our case is a little tricky. The problem is that we are reducing the counter **our_num_infected**. Different from most parallel structure, reduction in OpenMP is pretty easy to implement. We just need to add a reduction literal,

```
        reduction(+:our_num_infected_local)
```

The problem lies on that the counter we are reducing is inside a structure, namely, the our structure. OpenMP does not support reduction to structures. Therefore, we solve this problem by first create local instance such as **our_num_infected_local** that equals to counter **our_num_infected** in our struct.

```
    int our_num_infected_local = our->our_num_infected;
```

we can then, reduce to local instance,

```
        our_num_infected_local++;
```

Finally, we put local instance back to struct.

```
    our->our_num_infected = our_num_infected_local;
```

We then use the same reduction method for the second part of the function. The third and Fourth part of the function does not reduce any counters, which means we don't need worry about reduction at all.

10.2 In Core.h

There are four core functions inside *Core.h* file, and all of them can be parallelized using OpenMP.

10.2.1 move()

This function is easy to parallelize because it does not perform any reduction. However, we need to specify the variables that is private to each OpenMP threads. **current_person_id** is iterator that is clearly private. **x_move_direction** and **y_move_direction** are different for every thread, which means they are private as well.

```
#ifdef _OPENMP
#pragma omp parallel for private(current_person_id, x_move_direction, \
    y_move_direction)
#endif
```

10.2.2 susceptible()

This function is relatively hard to parallelize because it has four counters to reduce. Luckily, we already developed our way of reducing counters in **init_array()** function, which means we can use same method in here.

Creating local instances

```
// OMP does not support reduction to struct, create local instance
// and then put local instance back to struct
int num_infection_attempts_local = stats->num_infection_attempts;
int num_infections_local = stats->num_infections;
int num_infected_local = global->num_infected;
int num_susceptible_local = global->num_susceptible;
```

OpenMP initialization

```
#ifdef _OPENMP
#pragma omp parallel for private(current_person_id, num_infected_nearby, \
    my_person) reduction(+:num_infection_attempts_local) \
    reduction(+:num_infected_local) reduction(+:num_susceptible_local) \
    reduction(+:num_infections_local)
#endif
```

Put local instances back to global struct

```
// update struct data with local instances
stats->num_infection_attempts = num_infection_attempts_local;
stats->num_infections = num_infections_local;
global->num_infected = num_infected_local;
global->num_susceptible = num_susceptible_local;
```

10.2.3 infected()

Similar to **susceptible()** function, we have five counters to reduce in this function.

Creating local instances

```
// OMP does not support reduction to struct, create local instance
// and then put local instance back to struct
int num_recovery_attempts_local = stats->num_recovery_attempts;
int num_deaths_local = stats->num_deaths;
int num_dead_local = global->num_dead;
int num_infected_local = global->num_infected;
int num_immune_local = global->num_immune;
```

OpenMP initialization

```
#ifdef _OPENMP
#pragma omp parallel for private(current_person_id) \
    reduction(+:num_recovery_attempts_local) reduction(+:num_dead_local) \
    reduction(+:num_infected_local) reduction(+:num_deaths_local) \
    reduction(+:num_immune_local)
#endif
```

Put local instances back to global struct

```
// update struct data with local instances
stats->num_recovery_attempts = num_recovery_attempts_local;
stats->num_deaths = num_deaths_local;
global->num_dead = num_dead_local;
global->num_infected = num_infected_local;
global->num_immune = num_immune_local;
```

10.2.4 update_days_infected()

We don't have any reduction in this function, which means that the parallelization is relatively easy.

```
#ifdef _OPENMP
    #pragma omp parallel for private(current_person_id)
#endif
```

CHAPTER

ELEVEN

INCLUDING CUDA

Download Pandemic-MPI-CUDA.zip

In this chapter, we will include CUDA functions into the pandemic program we developed. Since CUDA only takes over the program when we execute the core functions, most of the program remain unchanged. However, following changes are needed for CUDA set up and initialization.

11.1 In Defaults.h

We need to include one extra structure in the Defualts.h file. This structure will include all the pointers used for GPU device memory and other necessary data, such as CUDA block size and CUDA grid size.

11.1.1 cuda_t struct

```
// All the data needed for CUDA operation: CUDA needs memory
// pointers and other information on CPU side. As more than
// one function (mainly used by CUDA.cu) need to use these
// data, we decided to use a struct to hold all these data.
struct cuda_t
{
    // correspond with their_infected_locations in global struct
    int *their_infected_x_locations_dev;
    int *their_infected_y_locations_dev;
    // correspond with our_infected_locations in our struct
    int *our_x_locations_dev;
    int *our_y_locations_dev;
    // correspond with our_states and our_num_days_infected in our struct
    int *our_num_days_infected_dev;
    char *our_states_dev;

    // some counter variables require atomic operations
    // correspond with states counters in our struct
    int *our_num_susceptible_dev;
    int *our_num_immune_dev;
    int *our_num_dead_dev;
    int *our_num_infected_dev;

    // correspond with variables in stats struct
    int *our_num_infections_dev;
    int *our_num_infection_attempts_dev;
    int *our_num_deaths_dev;
    int *our_num_recovery_attempts_dev;

    // the following four variables serve as the intermediate
    // variables. we initialized variables in stats struct as
    // doubles, but cuda atomic operations works better for
    // int. So we cast doubles to int and then cast them back
    int our_num_infections_int;
    int our_num_infection_attempts_int;
    int our_num_deaths_int;
    int our_num_recovery_attempts_int;

    // size used by cudaMalloc
    int our_size;
    int their_size;
    int our_states_size;

    // size used by cuda kernel calls
    int numThread;
    int numBlock;
};
```

their_infected_x_locations_dev

pointer, pointed to the memory location on device of array **their_infected_x_locations_dev**, a copy of **their_infected_x_locations** on host memory.

their_infected_y_locations_dev

pointer, pointed to the memory location on device of array **their_infected_y_locations_dev**, a copy of **their_infected_y_locations** on host memory.

our_x_locations_dev

pointer, pointed to the memory location on device of array **our_x_locations_dev**, a copy of **our_x_locations** on host memory.

our_y_locations_dev

pointer, pointed to the memory location on device of array **our_y_locations_dev**, a copy of **our_y_locations** on host memory.

our_num_days_infected_dev

pointer, pointed to the memory location on device of array **our_num_days_infected_dev**, a copy of **our_num_days_infected** on host memory.

our_states_dev

pointer, pointed to the memory location on device of array **our_states_dev**, a copy of **our_states** on host memory.

our_num_susceptible_dev

pointer, pointed to the memory location on device of counter **our_num_susceptible_dev**, a copy of **our_num_susceptible** on host memory.

our_num_immune_dev

pointer, pointed to the memory location on device of counter **our_num_immune_dev**, a copy of **our_num_immune** on host memory.

our_num_dead_dev

pointer, pointed to the memory location on device of counter **our_num_dead_dev**, a copy of **our_num_dead** on host memory.

our_num_infected_dev

pointer, pointed to the memory location on device of counter **our_num_infected_dev**, a copy of **our_num_infeced** on host memory.

our_num_infections_dev

pointer, pointed to the memory location on device of counter **our_num_infections_dev**, a copy of **our_num_infections** on host memory.

our_num_infection_attempts_dev

pointer, pointed to the memory location on device of counter **our_num_infection_attempts_dev**, a copy of **our_num_infection_attempts** on host memory.

our_num_deaths_dev

pointer, pointed to the memory location on device of counter **our_num_deaths_dev**, a copy of **our_num_deaths** on host memory.

our_num_recovery_attempts_dev

pointer, pointed to the memory location on device of counter **our_num_recovery_attempts_dev**, a copy of **our_num_recovery_attempts** on host memory.

our_num_infections_int

int, holds temporary instance of **our_num_infections** when we cast it into a int.

our_num_infection_attempts_int

int, holds temporary instance of **our_num_infection_attempts** when we cast it into a int.

our_num_deaths_int

int, holds temporary instance of **our_num_deaths** when we cast it into a int.

our_num_recovery_attempts_int

int, holds temporary instance of **our_num_recovery_attempts** when we cast it into a int.

our_size

int, holds the size of any **integer** arrays inside **our_t** struct.

their_size

int, holds the size of any **integer** arrays inside **global_t** struct.

our_states_size

int, holds the size of any **char** arrays inside **our_t** struct.

numThread

int, holds the number of threads per block, or block size.

numBlock

int, holds the number of blocks per grid, or grid size.

11.2 In Initialize.h

Since we are using CUDA, we need to initialize the CUDA runtime environment. To do this, we add another function in the **init**() function called **cuda_init**(). Don't forget to include the *cuda* structure in the function parameters.

```
int init (struct global_t *global, struct const_t *constant, struct stats_t *stats,
    struct display_t *dpy, struct cuda_t *cuda, int *c, char ***v)
    cuda_init(global, cuda);
```

Further, as we want to keep all the CUDA functions in one file, we put **cuda_init()** inside CUDA.cu file. Therefore, we need to include this file before we can use any functions inside it.

```
#include "CUDA.cu"      // for cuda_init()
```

11.2.1 cuda_init()

This function will setup the CUDA runtime environment.

Since we are allocating lots of arrays on the CUDA device memory, we first need to find out the size of each array. In total we need six arrays, of which **their_infected_x_locations_dev** and **their_infected_y_locations_dev** should be as long as the **total_number_of_people**, and the rest four arrays should have length as **our_number_of_people**. Note that of the four arrays above, **our_states_dev** is different from the rest because it holds char instead of int, which means we have to assign different size to it. The following line sets sizes we want.

```
    // initialize size needed for cudamalloc operations
    cuda->our_size = sizeof(int) * our->our_number_of_people;
    cuda->their_size = sizeof(int) * global->total_number_of_people;
    cuda->our_states_size = sizeof(char) * our->our_number_of_people;
```

After setting up the sizes, we can allocate arrays on the device. Note that all the pointers are already initialized in the cuda structure.

```
    // arrays in global and our struct
    cudaMalloc((void**)&cuda->their_infected_x_locations_dev, cuda->their_size);
    cudaMalloc((void**)&cuda->their_infected_y_locations_dev, cuda->their_size);
    cudaMalloc((void**)&cuda->our_x_locations_dev, cuda->our_size);
    cudaMalloc((void**)&cuda->our_y_locations_dev, cuda->our_size);
    cudaMalloc((void**)&cuda->our_states_dev, cuda->our_states_size);
    cudaMalloc((void**)&cuda->our_num_days_infected_dev, cuda->our_size);
```

Besides arrays, we also need in allocate spaces for the eight counters in our structure and stats structure.

```
    // states counters in our struct
    cudaMalloc((void**)&cuda->our_num_susceptible_dev, sizeof(int));
    cudaMalloc((void**)&cuda->our_num_immune_dev, sizeof(int));
    cudaMalloc((void**)&cuda->our_num_dead_dev, sizeof(int));
    cudaMalloc((void**)&cuda->our_num_infected_dev, sizeof(int));
    #ifdef SHOW_RESULTS
    // stats variables in stats struct
    cudaMalloc((void**)&cuda->our_num_infections_dev, sizeof(int));
    cudaMalloc((void**)&cuda->our_num_infection_attempts_dev, sizeof(int));
    cudaMalloc((void**)&cuda->our_num_deaths_dev, sizeof(int));
    cudaMalloc((void**)&cuda->our_num_recovery_attempts_dev, sizeof(int));
    #endif
```

After allocating structure, we need to set up the random number generator. Since all the device code are executed on GPU device instead of on CPU, functions like **random()** will not work. Therefore, we need to use NVIDIA cuRAND library to generate all the random numbers. According to the documentation of cuRAND library, the normal sequence of operations to generate random number for CUDA device can be divided into seven steps. **cuda_init()** function will cover three steps, **cuda_run()** function will cover three steps, and **cuda_finish()** function will cover the last step.

1. Create a new generator of the desired type with curandCreateGenerator().

```
// create cuda random number generator
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
```

2. Set the generator options; for example, use curandSetPseudoRandomGeneratorSeed() to set the seed.

```
// get time
time(&current_time);
// generate seed for the rand number generator
curandSetPseudoRandomGeneratorSeed(gen, (unsigned long)current_time);
```

3. Allocate memory on the device with cudaMalloc().

```
// array to hold random number
cudaMalloc((void**)&rand_nums, 2 * our->our_number_of_people * sizeof(float));
```

After generating random numbers, we need to set up block size and grid size for CUDA operations. Since the primary data type of our program is array, we can initialize only 1-D array for CUDA device functions.

Since the primary test machine for this module is LittleFe, which features NVIDIA ION Graphics (ION2), we set the block size to be 256 threads per block as the maximum active threads per multiprocessor on ION Graphics (Compute Capability 1.3) is 512. However, if you have GPU cards that are more advanced (Compute Capability 2.0+), you can set the block size to 512, 1024 or even 2048.

```
cuda->numThread = (our->our_number_of_people < 256 ? our->our_number_of_people : 256);
```

Further, if we have less than 256 people in our simulation, we initialize exactly number of people many of threads.

As for grid size, we decide grid size according to our simulation size. For example, if you have 1000 people in your simulation, program will initialize 4 blocks.

```
cuda->numBlock = (our->our_number_of_people+cuda->numThread-1)/cuda->numThread;
```

11.3 Replace file Core.h with file CUDA.cu

11.3.1 CUDA Global Variable

At any time, use of global variables outside of **main()** function is discouraged in C programming, mainly because it is really difficult to handle the scope of the program. However, as we are building a CUDA and MPI hybrid, **all** CUDA code need to be compiled with **nvcc** compiler, which means we need to separate CUDA code from other code. Normally, we should declare these variables inside cuda_t structure we initialized in **main()** function, but the problem is that MPI compiler **mpicc** or C compiler **gcc** or **icc** does not recognize the type curandGenerator_t, which forces us to declare global variable inside this file, which will eventually compiled by **nvcc**.

gen

curandGenerator_t, which is effectively a random generator on CUDA device. A generator in CURAND encapsulates all the internal state necessary to produce a sequence of pseudorandom or quasirandom numbers.

current_time

time_t, variable we use to hold the current time. We will use this as seed.

rand_nums

array, this is a pointer pointed to an array of random float numbers.

11.3.2 CUDA Device Functions

Inside Core.h file, we have four core functions for our pandemic simulation. **move()**, **susceptible()**, **infected()** and **update_days_infected**. Inside CUDA.cu file, we implemented those four functions with CUDA architecture.

11.3.3 cuda_move()

This is a CUDA implementation of the **move()** function in core functions chapter.

First, each thread randomly picks whether the person moves left or right or does not move in the x dimension.

```
        // The thread randomly picks whether the person moves left
        // or right or does not move in the x dimension
        int x_move_direction = (int)(rand_nums[id]*3) - 1;
```

The code uses (int)(rand_nums[id]*3) - 1; to achieve this. rand_num is a array of random numbers generated before. All the random numbers in this array are floats between 0 and 1. Then, rand_nums[id]*3 will turn all the floats to numbers between 0 and 3. After this, we can cast all the floats to int, which eventually will make all the numbers as either 0, 1 or 2. Finally, we subtract 1 from this to produce -1, 0, or 1. This means the person can move to the right(1), stay in place (0), or move to the left (-1).

The thread randomly picks whether the person moves up or down or does not move in the y dimension. This is similar to movement in x dimension.

```
        // The thread randomly picks whether the person moves up
        // or down or does not move in the y dimension
        int y_move_direction = (int)(rand_nums[id+SIZE]*3) - 1;
```

Next, we need to make sure the person remain in the bounds of the environment after moving. We check this by making sure the person's x location is greater than or equal to 0 and less than the width of the environment and that the person's y location is greater than or equal to 0 and less than the height of the environment. In the code, it looks like this:

```
        if( (x_locations_dev[id] + x_move_direction >= 0) &&
            (x_locations_dev[id] + x_move_direction < environment_width) &&
            (y_locations_dev[id] + y_move_direction >= 0) &&
            (y_locations_dev[id] + y_move_direction < environment_height) )
```

Finally, The thread moves the person

```
            // The thread moves the person
            x_locations_dev[id] = x_locations_dev[id] + x_move_direction;
            y_locations_dev[id] = y_locations_dev[id] + y_move_direction;
```

11.3.4 cuda_susceptible()

This is a CUDA implementation of the **susceptible()** function in core functions chapter.

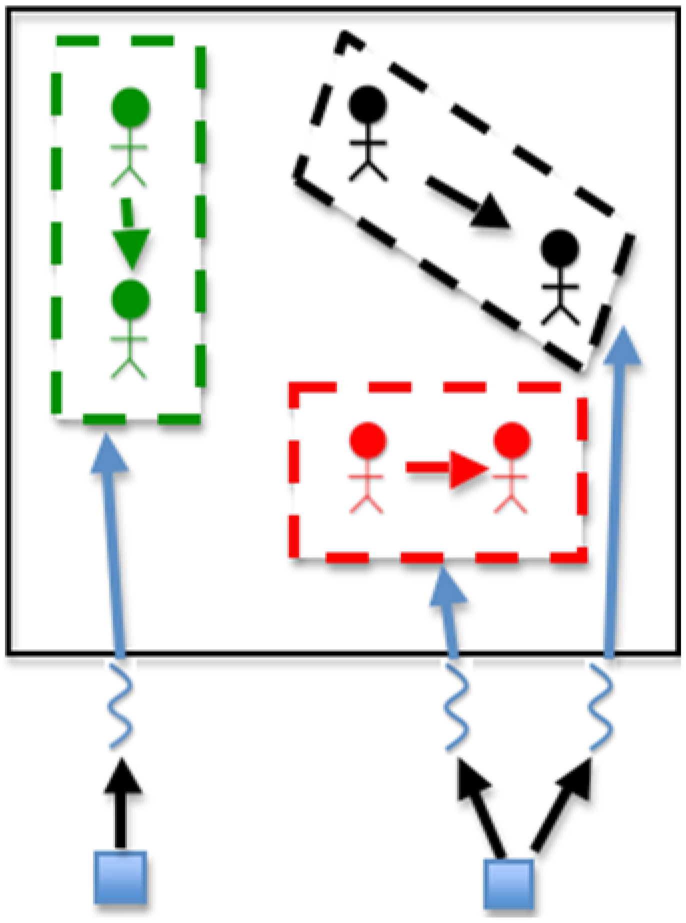
If the person is susceptible,

```
    if(states_dev[id] == SUSCEPTIBLE)
```

For each of the infected people (received earlier from all processes) or until the number of infected people nearby is 1,

```
        for(i=0; i<=global_num_infected-1 && num_infected_nearby<1; i++)
```

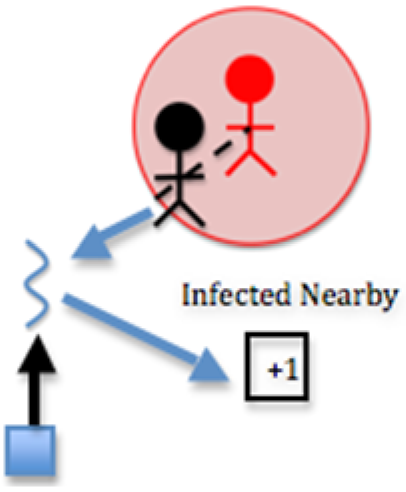
If this person is within the infection radius,



```
            if( (x_locations_dev[id] > infected_x_locations_dev[i] - infection_radius) &&
                (x_locations_dev[id] < infected_x_locations_dev[i] + infection_radius) &&
                (y_locations_dev[id] > infected_y_locations_dev[i] - infection_radius) &&
                (y_locations_dev[id] < infected_y_locations_dev[i] + infection_radius) )
```

then, the thread increments the number of infected people nearby

```
                num_infected_nearby++;
```



This is where a large chunk of the algorithm's computation occurs. Each susceptible person must be computed with each infected person to determine how many infected people are nearby each person. Two nested loops means many computations. In this step, the computation is fairly simple, however. The thread simply increments the **num_infected_nearby** variable.

Note in the code that if the number of infected nearby is greater than or equal to 1 and we have **SHOW_RESULTS** enabled, we increment the **num_infection_attempts** variable. This helps us keep track of the number of attempted infections, which will help us calculate the actual contagiousness of the disease at the end of the simulation.

Similar to **cuda_move()**, we also need random numbers in this function. The difference is that we need integers between -1 and 1 in **cuda_move()** function but we need integers between 0 and 100 in this function. We obtain this random number using

```
        // generate a random number between 0 and 100
        int rand_num = (int)(rand_nums[id]*100);
```

where rand_nums is still an array of random floats between 0 and 1 and we can multiply it with 100 and cast it into a int.

If there is at least one infected person nearby, and a random number less than 100 is less than or equal to the contagiousness factor, then

```
        if(num_infected_nearby >= 1 && rand_num <= contagiousness_factor)
```

The thread changes person's state to infected

```
            states_dev[id] = INFECTED;
```

So far the code is similar to the **susceptible()** function executed on the CPU end. However, things get trickier from here. Since every threads need to update counters like **num_infected** or **num_susceptible** if someone is infected, we have racing conditions. In order to handle racing conditions and to maximize performance at the same time, we use both cuda shared memory and cuda atomic operations to update counters.

We use shared memory as temporary arrays to holds counters changed by each thread, then we would reduce this array to a single number. Finally, we use CUDA atomic operations to add the number back to actual counter.

CUDA shared memory is cache assigned to each multiprocessor. In case some of you are not familiar with the concept of multiprocessor, you can think of multiprocessor is the physical phase of blocks in CUDA coding. A typical NVIDIA GPU card with Fermi architecture (perfectly fine if you don't know what this is) supports maximum 1024 active threads per multiprocessor. This means that you can run 1024 threads concurrently on each multiprocessor. The reason we usually chose 128, 256 or 512 threads per block is that we want each multiprocessor can host exactly 8, 4 or 2 blocks on it.

However, even if we use 128 threads per block when we launch the device functions, we don't necessarily get 8 blocks per multiprocessor. Why? Because each multiprocessor has limited shared memory and registers available. GPU with Fermi architecture usually have 48KB of shared memory per multiprocessor, which means that if each block uses 8KB of shared memory, you can only initialize 6 blocks on each multiprocessor. For us, this is less of a concern because we only allocate four or five (later you will see why is four or five) arrays per block. Even we are using 1024 threads per block, we need maximum 5 * 1024 * sizeof(int) = 20KB, which is less than half of the shared memory available.

We first need to find out how many counters need atomic operations, in this function, there are four of them: **num_infected_dev**, **num_susceptible_dev**, **num_infection_attempts_dev** and **num_infections_dev**. This is important because we need to allocate enough memory when we invoke the device function calls. Since we have four counters need atomic operations, we need to allocate four arrays, each having the length of the numbers of threads per block. The following line declares the shared memory:

```
/* CUDA shared memory allocation */
extern __shared__ int array[];
```

This line suggests that we allocated an array of the data type int. However, it does not specify how long the array should be. Then, inside cuda_susceptible function, the following lines set up the four arrays we use for reduction.

```
    // set up shared memory
    int *num_infected = (int*)array;
    int *num_susceptible = (int*)&num_infected[numThread];
    #ifdef SHOW_RESULTS
    int *num_infection_attempts = (int*)&num_susceptible[numThread];
    int *num_infections = (int*)&num_infection_attempts[numThread];
    #endif
```

we set the pointer of the first array as the pointer of the shared memory array. Then, we set the pointer of the second array as the pointer exactly **numThread** away from the pointer of the first array. We are essentially dividing the initial shared memory array into four equal sized arrays.

After shared memory setup, we need to reset the shared memory. So each thread set its corresponding shared memory elements to zero at the very beginning of the function.

```
    // reset the shared memory
    num_infected[blockId] = 0;
    num_susceptible[blockId] = 0;
    #ifdef SHOW_RESULTS
    num_infection_attempts[blockId] = 0;
    num_infections[blockId] = 0;
    #endif
```

Again this is very important. Shared memory will not clear itself after usage, and failing to clear shared memory before usage usually meaning you are starting from what ever values the shared memory is left with from last CUDA operations.

When we are updating counters, instead of adding one to or subtracting one from the actual counter located on GPU device, in this case the **num_infected_dev** or **num_susceptible_dev**counter, we add one to or subtract one from the thread's corresponding array elements.

```
            #ifdef SHOW_RESULTS
            num_infection_attempts[blockId]++;
            #endif
            num_infected[blockId]++;
            num_susceptible[blockId]--;
            #ifdef SHOW_RESULTS
            num_infections[blockId]++;
            #endif
```

Finally, we need to add up the values in each array to obtain the final result. We do this using CUDA binary tree reduction. This is the official way to perform reduction operations in CUDA. The basic idea is that you create a half point on the array, use the first half thread to add the values of second half thread. This means that the array shrinks to one half of its original size. Then you can do another reduction, which will shrinks the array to one fourth of its original size. When the operation is done, the correct sum is stored at the first element of the array.The following is the implementation:

```
        i = numThread/2;
        while (i != 0) {
            if (blockId < i){
                num_infected[blockId] += num_infected[blockId + i];
                num_susceptible[blockId] += num_susceptible[blockId + i];
                #ifdef SHOW_RESULTS
                num_infection_attempts[blockId] += num_infection_attempts[blockId + i];
                num_infections[blockId] += num_infections[blockId + i];
                #endif
            }
            __syncthreads();
            i /= 2;
        }
```

As you probably already see, one limitation of this operation is that the array size has to be the power of 2, which essentially meaning that the block size should be power of 2 as well. If we are dealing with problem size as large as tens of thousands even millions, this won't hurt us because we are always initializing 128, 256, 512 or even 1024 threads per block. However, if we are dealing with problem size as small as 50, things gets a little bit tricker.

Therefore, we put a if statement that checks whether the size of the block is power of 2 before we do any reduction operations. Such as:

```
    if(((numThread!=0) && !(numThread & (numThread-1)))){
```

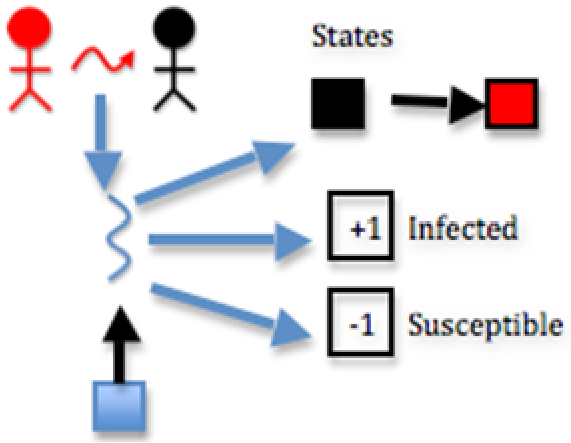
if we indeed do not have some power of 2 many of threads in a block, we can use the **first thread** of the block to add all other entries in the array to the first element.

```
        if(blockId == 0) {
            for(i=1; i<numThread; i++){
                num_infected[0] += num_infected[i];
                num_susceptible[0] += num_susceptible[i];
                #ifdef SHOW_RESULTS
                num_infection_attempts[0] += num_infection_attempts[i];
                num_infections[0] += num_infections[i];
                #endif
            }
        }
```

The good news is that when we run into this problem, normally means that we are dealing with a very small problem size, which should not affect the performance significantly. Notice that we could use the first thread to add up the values even if we have 128 or 256 threads per block, but the reduction takes 127 or 255 steps. However, the binary tree reduction takes 7 or 8 steps to do the same. This will make our program run much faster.

Finally, the **first thread** update the acutal counter with the first value of the array. However, we still face racing condition because more than one block could be updating the actual counter at the same time. CUDA designs functions like **atomicAdd** to handle situations like this, it can slow down your program significantly if you use **atomicAdd** too much, but since we are doing this once per block per counter, we do not suffer too much from performance loss.

```
if(blockId == 0) {
    atomicAdd(num_infected_dev, num_infected[0]);
    atomicAdd(num_susceptible_dev, num_susceptible[0]);
    #ifdef SHOW_RESULTS
    atomicAdd(num_infection_attempts_dev, num_infection_attempts[0]);
    atomicAdd(num_infections_dev, num_infections[0]);
    #endif
```



Note in the code that if the infection succeeds and we have **SHOW_RESULTS** enabled, we increment the **num_infections_dev** variable. This helps us keep track of the actual number of infections, which will help us calculate the actual contagiousness of the disease at the end of the simulation.

11.3.5 cuda_infected()

This is a CUDA implementation of the **infected**() function in core functions chapter.

If the person is infected and has been for the full duration of the disease, then

```
if(states_dev[id] == INFECTED && num_days_infected_dev[id] == duration_of_disease)
```

Note in the code that if we have **SHOW_RESULTS** enabled, we increment the **num_recovery_attempts_dev** variable. This helps us keep track of the number of attempted recoveries, which will help us calculate the actual deadliness of the disease at the end of the simulation.

```
    #ifdef SHOW_RESULTS
    num_recovery_attempts[blockId]++;
    #endif
```

After this, if a random number less than 100 is less than the deadliness factor, then

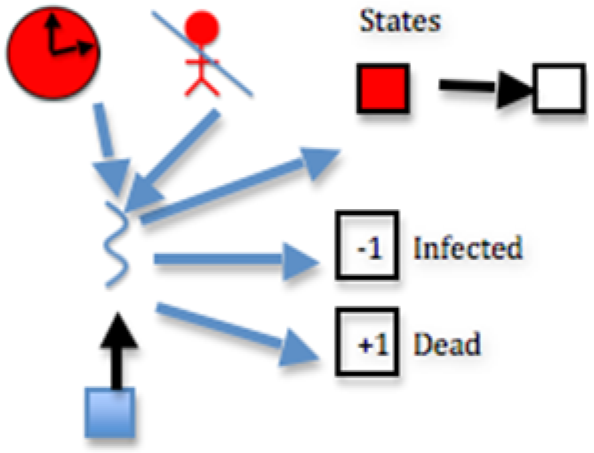
```
    // generate a random number between 0 and 100
    int rand_num = (int)(rand_nums[id]*100);
```

The thread changes the person's state to dead

```
        // The thread changes the person's state to dead
        states_dev[id] = DEAD;
```

and then the thread updates the counters

```
// The thread updates the counters
num_dead[blockId]++;
num_infected[blockId]--;
#ifdef SHOW_RESULTS
num_deaths[blockId]++;
#endif
```



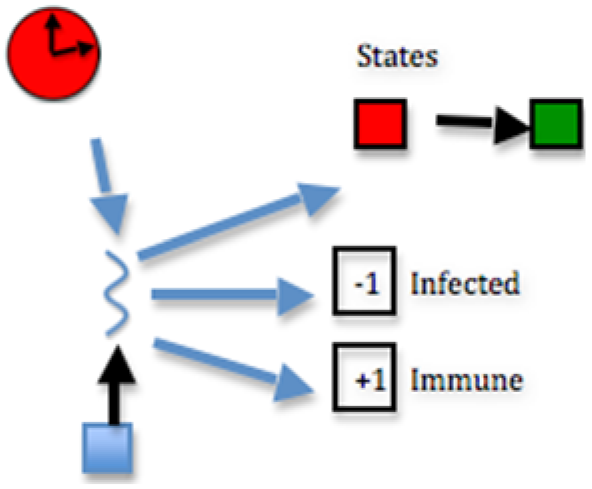
This step is effectively the same as function susceptible, considering deadliness instead of contagiousness. The difference here is the following step:

if a random number less than 100 is less than the deadliness factor, the thread changes the person's state to immune

```
// The thread changes the person's state to immune
states_dev[id] = IMMUNE;
```

and then thread updates the counters

```
// The thread updates the counters
num_immune[blockId]++;
num_infected[blockId]--;
```



If deadliness fails, then immunity succeeds.

Note in the code that if the person dies and we have **SHOW_RESULTS** enabled, we increment the **num_deaths_dev** variable. This helps us keep track of the actual number of deaths, which will help us calculate the actual deadliness of the disease at the end of the simulation.

Note that the reduction process is the same as the **susceptible_cuda()** function, which involves shared memory reduction and CUDA atomic operations. The only difference is that we have five counters to reduce instead of four. This will be reflected when we assign shared memory space for each block.

11.3.6 cuda_update_days_infected()

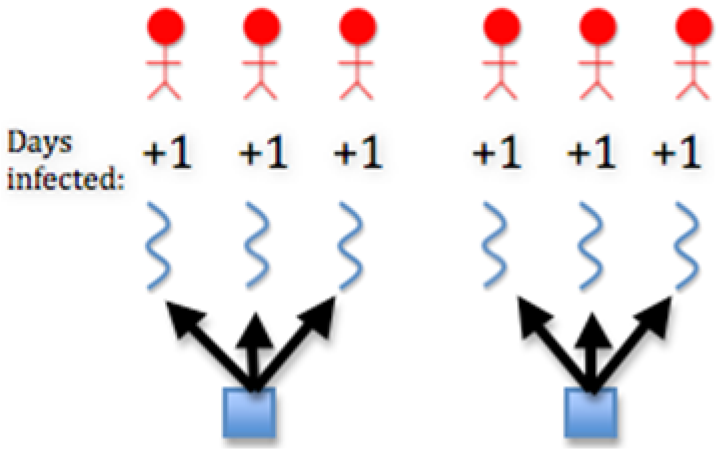
This is the CUDA implementation of the **update_days_infected()** function in core functions chapter.

If the person is infected, then

```
    if(states_dev[id] == INFECTED)
```

Increment the number of days the person has been infected

```
        // Increment the number of days the person has been infected
        num_days_infected_dev[id]++;
```



11.4 Change function calls in Pandemic.c File

Since we are not using core functions in Core.h file and we are using device functions on CUDA device, we need to change function calls in **main()** function.

Before changing function calls, we first need to include *Cuda.cu* file before we can use any of the functions in it.

```
#include "Infection.h"
#include "CUDA.cu"
#include "Finalize.h"
```

Then we need to create a *cuda* structure.

```
    struct cuda_t cuda;
```

Finally, we replace the four core function calls with a single function call. Why only one function call? Because calling a CUDA function is more complicated than calling a normal function, and we want to keep all the CUDA code together in the same file. Therefore, we created a **cuda_run()** function.

11.4.1 cuda_run()

This function will execute the CUDA device functions.

We first use **cudaMemcpy()** to copy data on host memory to GPU device memory. Since all of the code only performs one day's simulation, we need to put **cuda_run()** function inside a loop. One could call all the **cudaMemcpy()** functions in each iteration, or we could divide them into two categories, those that requires constantly communicating with CPU and those who do not.

After careful examination of the code, it is not hard to find out that some functions, especially MPI functions, on host end need **infected_x_locations** and **infected_y_locations** to share infected information to all other nodes. They also need these arrays to do display. Therefore, in every iteration, we need to copy these two arrays to GPU device and copy then back to host after execution. However, other arrays or counters can reside on card from start to finish without re-copy from host to GPU device. Therefore, we implement **cudaMemcpy()** functions in the following fashion,

```
// copy infected locations to device in EVERY ITERATION
cudaMemcpy(cuda->their_infected_x_locations_dev, global->their_infected_x_locations, cuda->their_
cudaMemcpy(cuda->their_infected_y_locations_dev, global->their_infected_y_locations, cuda->their_

// copy other information to device only in FIRST ITERATION
// we don't need to copy these information every iteration
// becuase they can be reused in each iteration without any
// process at the host end.
if(our->current_day == 0){
    // copy arrays in our struct
    cudaMemcpy(cuda->our_x_locations_dev, our->our_x_locations, cuda->our_size, cudaMemcpyHostToD
    cudaMemcpy(cuda->our_y_locations_dev, our->our_y_locations, cuda->our_size, cudaMemcpyHostToD
    cudaMemcpy(cuda->our_states_dev, our->our_states, cuda->our_states_size, cudaMemcpyHostToDevi
    cudaMemcpy(cuda->our_num_days_infected_dev, our->our_num_days_infected, cuda->our_size, cudaM
    // copy states counters in our struct
    cudaMemcpy(cuda->our_num_susceptible_dev, &our->our_num_susceptible, sizeof(int), cudaMemcpyH
    cudaMemcpy(cuda->our_num_immune_dev, &our->our_num_immune, sizeof(int), cudaMemcpyHostToDevic
    cudaMemcpy(cuda->our_num_dead_dev, &our->our_num_dead, sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(cuda->our_num_infected_dev, &our->our_num_infected, sizeof(int), cudaMemcpyHostToD

    #ifdef SHOW_RESULTS
    // variables in stats data are initialized as doubles, yet CUDA
    // atomic operations prefer integer than doubles. Therefore, we
    // cast doubles to integer before the cudaMemcpy operations.
    cuda->our_num_infections_int = (int)stats->our_num_infections;
    cuda->our_num_infection_attempts_int = (int)stats->our_num_infection_attempts;
    cuda->our_num_deaths_int = (int)stats->our_num_deaths;
    cuda->our_num_recovery_attempts_int = (int)stats->our_num_recovery_attempts;
    // copy stats variables in stats struct
    cudaMemcpy(cuda->our_num_infections_dev, &cuda->our_num_infections_int, sizeof(int), cudaMemc
    cudaMemcpy(cuda->our_num_infection_attempts_dev, &cuda->our_num_infection_attempts_int, sizeo
    cudaMemcpy(cuda->our_num_deaths_dev, &cuda->our_num_deaths_int, sizeof(int), cudaMemcpyHostTo
    cudaMemcpy(cuda->our_num_recovery_attempts_dev, &cuda->our_num_recovery_attempts_int, sizeof
    #endif
}
```

where **if(our->current_day == 0)** makes sure that most of the data only gets copied in the first iteration, instead of in every iteration.

Another thing you probably noticed is that we cast stats counters to int before sending them to the GPU device memory.

```
    cuda->our_num_infections_int = (int)stats->our_num_infections;
    cuda->our_num_infection_attempts_int = (int)stats->our_num_infection_attempts;
    cuda->our_num_deaths_int = (int)stats->our_num_deaths;
```

```
        cuda->our_num_recovery_attempts_int = (int)stats->our_num_recovery_attempts;
```

This is because that variables in *stats* structure are initialized as doubles, but CUDA atomic operations prefer integer. Since we always perform integer operations on stats counters (either add one or subtract one), we can first cast them into int, and we can later cast them back to double after device function's execution.

After the copying the data, we need to generate the random numbers. Recall that we performed the first three steps of the seven steps CUDA random number generation process, the next step, or the fourth step is:

4. Generate random numbers with curandGenerate() or another generation function.

```
curandGenerateUniform(gen, rand_nums, 2 * our->our_number_of_people);
```

Notice that we are generating twice many of total people number of random numbers. This is because the next device function call is **cuda_move()**, which moves every person in both x direction and y direction.

Then, we can call device functions from host:

```
// execute device code on updating people's movement
int environment_width = constant->environment_width;
int environment_height = constant->environment_height;
cuda_move<<<cuda->numBlock, cuda->numThread>>>(cuda->our_states_dev,
    cuda->our_x_locations_dev, cuda->our_y_locations_dev, DEAD,
    environment_width, environment_height, rand_nums, our->our_number_of_people);
// Sync Threads
cudaThreadSynchronize();

// generate our_number_of_people many of randome numbers.
curandGenerateUniform(gen, rand_nums, our->our_number_of_people);

// execute device code on susceptible people
int infection_radius = constant->infection_radius;
int contagiousness_factor = constant->contagiousness_factor;
int total_num_infected = global->total_num_infected;
cuda_susceptible<<<cuda->numBlock, cuda->numThread, 4*cuda->numThread*sizeof(int)>>>(
    cuda->our_states_dev, cuda->our_x_locations_dev, cuda->our_y_locations_dev,
    cuda->their_infected_x_locations_dev, cuda->their_infected_y_locations_dev,
    cuda->our_num_infected_dev, cuda->our_num_susceptible_dev,
    cuda->our_num_infection_attempts_dev, cuda->our_num_infections_dev,
    rand_nums, total_num_infected, infection_radius,
    contagiousness_factor, SUSCEPTIBLE, INFECTED);
// Sync Threads
cudaThreadSynchronize();

// generate our_number_of_people many of randome numbers.
curandGenerateUniform(gen, rand_nums, our->our_number_of_people);

// execute device code on infected people
int duration_of_disease = constant->duration_of_disease;
int deadliness_factor = constant->deadliness_factor;
cuda_infected<<<cuda->numBlock, cuda->numThread, 5*cuda->numThread*sizeof(int)>>>(
    cuda->our_states_dev, cuda->our_num_days_infected_dev,
    cuda->our_num_recovery_attempts_dev, cuda->our_num_deaths_dev,
    cuda->our_num_infected_dev, cuda->our_num_immune_dev,
    cuda->our_num_dead_dev, duration_of_disease, deadliness_factor,
    IMMUNE, DEAD, INFECTED, rand_nums);
// Sync Threads
cudaThreadSynchronize();

// execute device code to update infected days
```

```
cuda_update_days_infected<<<cuda->numBlock, cuda->numThread>>>(
    cuda->our_states_dev, cuda->our_num_days_infected_dev, INFECTED);
// Sync Threads
cudaThreadSynchronize();
```

Most of the device function calls are straight forward, however, two things needed to be pointed out. First is that we perform the 5th step and 6th step of CUDA random number generation process in between, which are

5. Use the results.

6. If desired, generate more random numbers with more calls to curandGenerate().

Another thing is that when calling **cuda_susceptible()** and **cuda_infected()** functions, we passed a third argument other than **numThread** and **numBlock** to device function.

```
cuda_susceptible<<<cuda->numBlock, cuda->numThread, 4*cuda->numThread*sizeof(int)>>>(
cuda_infected<<<cuda->numBlock, cuda->numThread, 5*cuda->numThread*sizeof(int)>>>(
```

The third parameter is the size of the shared memory, which depends on how many counters we need to reduce in each function.

Finally, we need to copy GPU device data back to host. However, just like when we copy data from host to GPU device, we need to differentiate data that needs to be copied in every iteration and those that needs to be copied only once. In this case, we need to copy arrays **x_locations**, **y_locations** and **states** back to host memory. This is because MPI functions will need them to perform **Allgather()** and **Allgatherv()** operations. We also copied counter **num_infected** back because we need it in other functions as well.

As for other arrays or counters, we can copy them back in the last iteration. Notice that we never copy **num_infected_days** array back to host memory, this is because non of the host functions need this array.

```
// copy our locations, our states and our_num_infected back to host
// in EVERY ITERATION
cudaMemcpy(our->our_x_locations, cuda->our_x_locations_dev, cuda->our_size, cudaMemcpyDeviceToHos
cudaMemcpy(our->our_y_locations, cuda->our_y_locations_dev, cuda->our_size, cudaMemcpyDeviceToHos
cudaMemcpy(our->our_states, cuda->our_states_dev, cuda->our_states_size, cudaMemcpyDeviceToHost);
cudaMemcpy(&our->our_num_infected, cuda->our_num_infected_dev, sizeof(int), cudaMemcpyDeviceToHos

// copy other information back to host only in LAST ITERATION
// we only copy the counters back for results calculation.
// we don't need to copy our_num_days_infected back.
if(our->current_day == constant->total_number_of_days){
    // copy states counters in our struct
    cudaMemcpy(&our->our_num_susceptible, cuda->our_num_susceptible_dev, sizeof(int), cudaMemcpyD
    cudaMemcpy(&our->our_num_immune, cuda->our_num_immune_dev, sizeof(int), cudaMemcpyDeviceToHos
    cudaMemcpy(&our->our_num_dead, cuda->our_num_dead_dev, sizeof(int), cudaMemcpyDeviceToHost);

    #ifdef SHOW_RESULTS
    // copy stats variables in stats struct
    cudaMemcpy(&cuda->our_num_infections_int, cuda->our_num_infections_dev, sizeof(int), cudaMemc
    cudaMemcpy(&cuda->our_num_infection_attempts_int, cuda->our_num_infection_attempts_dev, sizec
    cudaMemcpy(&cuda->our_num_deaths_int, cuda->our_num_deaths_dev, sizeof(int), cudaMemcpyDevice
    cudaMemcpy(&cuda->our_num_recovery_attempts_int, cuda->our_num_recovery_attempts_dev, sizeof
    // cast interger back to double after the cudaMemcpy operations.
    stats->our_num_infections = (double)cuda->our_num_infections_int;
    stats->our_num_infection_attempts = (double)cuda->our_num_infection_attempts_int;
    stats->our_num_deaths = (double)cuda->our_num_deaths_int;
    stats->our_num_recovery_attempts = (double)cuda->our_num_recovery_attempts_int;
    #endif
}
```

11.5 In Finalize.h

After the CUDA operations, we need to perform clean up operations, such as free memory allocated on device and destroy random number generator. All these operations are packed in the **cuda_finish()** function in the *CUDA.cu* file. However, we still need to call this function from somewhere. We decided to call this function inside **cleanup()** function in Finalize.h file.

Just like modifying **Initialize.h**, we first need to include CUDA.cu file,

```
#include "CUDA.cu"      // for cuda_finish()
```

Then we can call the **cuda_finish()** function

```
    cuda_finish(cuda);
```

11.5.1 cuda_finish()

This function will finish the CUDA environment.

After allocating all the arrays and counters on GPU device memory, we need to free them.

```
    // free the memory on the GPU
    // arrays in global and our struct
    cudaFree(cuda->their_infected_x_locations_dev);
    cudaFree(cuda->their_infected_y_locations_dev);
    cudaFree(cuda->our_x_locations_dev);
    cudaFree(cuda->our_y_locations_dev);
    cudaFree(cuda->our_states_dev);
    cudaFree(cuda->our_num_days_infected_dev);
    // states counters in our struct
    cudaFree(cuda->our_num_susceptible_dev);
    cudaFree(cuda->our_num_immune_dev);
    cudaFree(cuda->our_num_dead_dev);
    cudaFree(cuda->our_num_infected_dev);

    #ifdef SHOW_RESULTS
    // stats variables in stats struct
    cudaFree(cuda->our_num_infections_dev);
    cudaFree(cuda->our_num_infection_attempts_dev);
    cudaFree(cuda->our_num_deaths_dev);
    cudaFree(cuda->our_num_recovery_attempts_dev);
    #endif
```

Further, the last step of CUDA random number generation process is:

7. Clean up with curandDestroyGenerator().

```
    // array to hold random number
    cudaFree(rand_nums);
    // destroy cuda random number generator
    curandDestroyGenerator(gen);
```

Hitting the next links takes you from one chapter to another and previous takes you back one chapter.