Pandemic Exemplar with OpenMP

CSinParallel Project

March 11, 2014

CONTENTS

This example contains a fully functional simulation of the type of modeling done by epidemiologists to answer the question: what happens when an infectious disease hits a population? Code for an original serial version is provided and described in detail. Next, descriptions of a parallel version using OpenMP is provided, where the code is modified from the original serial version.

Acknowledgment: Many thanks to Aaron Weeden of the Shodor Foundation for the original version of this material and code.

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:

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
- OpenMP programming
- CUDA programming
- Scaling parallel code

Each of these topics is mentioned to some degree in the material that follows. Some familiarity with each of these is assumed, except that the disease modeling is explained here.

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

1.1 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.

1.2 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.

[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.

1.3 Motivation for Parallelism

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 in Figure 2.

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

1.3.1 Amdahl's Law

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 an equation:

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

where

[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.'

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

1.3.2 Gustafson's Law

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.

1.3.3 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 available for download in archive files. 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.

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, please follow along the rest of this document.

CHAPTER

TWO

PROGRAM STRUCTURE

`Download Pandemic-Serial.tgz`

There are in total 8 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 CUDA functions in the *CUDA.cu* file is not included in the file table, even though they are shown in the diagram. The initial serial program does not need these functions or the file. However, we will be using them later in a CUDA version of this code.

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 need to include all the necessary code files. We first include file files that are needed with our without display.

```
#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
```

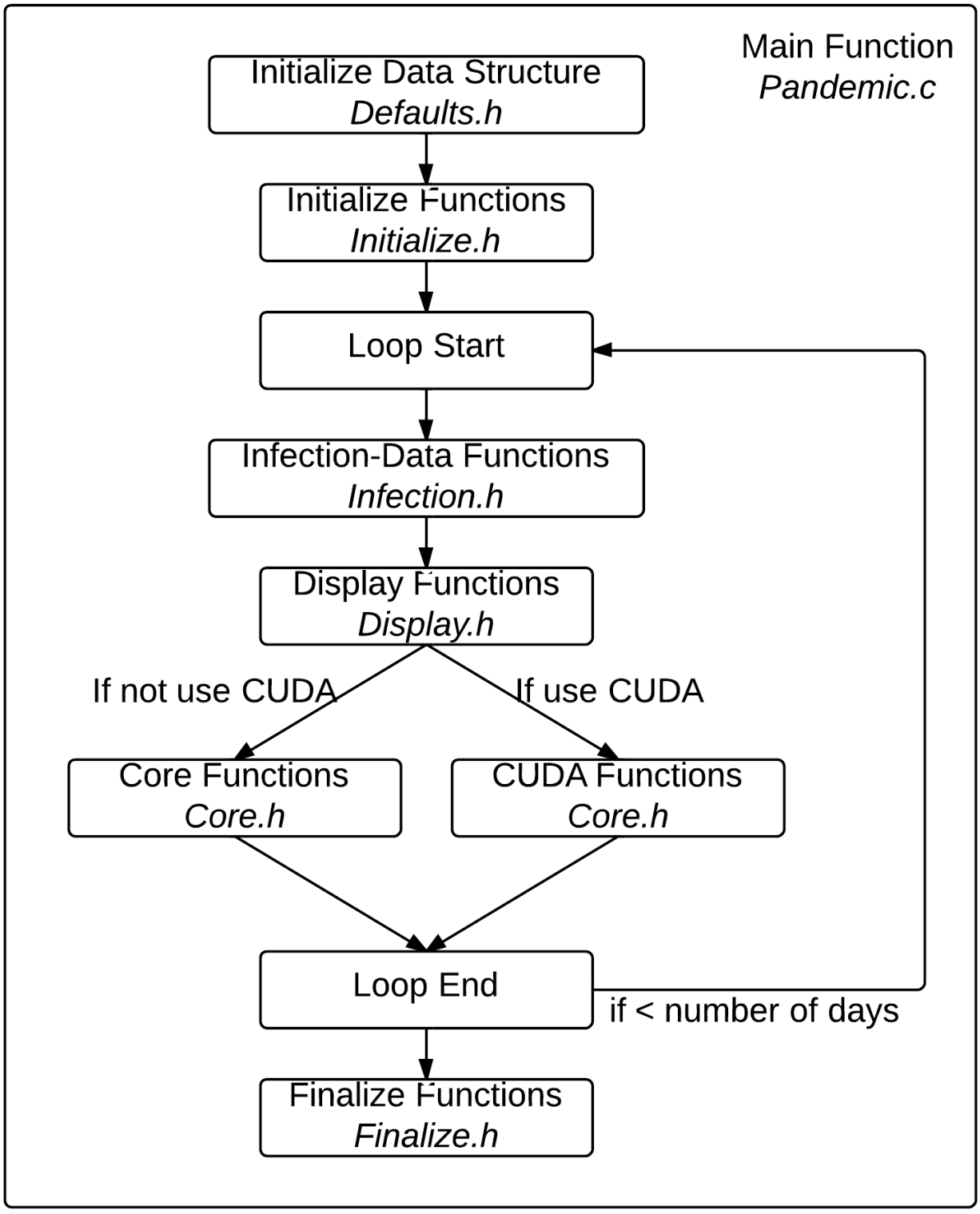


Figure 2.1: Overall Program Structurer

2.2.1 main()

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

```
/**** In Defaults.h ****/
struct global_t global;
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, &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(global.current_day = 0; global.current_day <= constant.total_number_of_days;
    global.current_day++)
{
}
```

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

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

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)

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

    throttle(&constant);

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

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

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

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

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

    update_days_infected(&global, &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(&global, &stats);

cleanup(&global, &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 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 "Model" section). 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 hold the window id.

screen

Screen, variable to hold 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

3.2 global_t struct

```
// All the data needed globally. Holds EVERYONE's location,
// states and other necessary counters.
struct global_t
{
        // current day
        int current_day;
        // people counters
        int number_of_people;
        int num_initially_infected;
        // states counters
        int num_infected;
        int num_susceptible;
        int num_immune;
        int num_dead;
        // locations
        int *x_locations;
        int *y_locations;
        // infected people's locations
        int *infected_x_locations;
        int *infected_y_locations;
        // state
        char *states;
        // infected time
```

```
        int *num_days_infected;
};
```

current_day

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

number_of_people

the total number of all people in the simulation. The value of this variable can be specified on the command line with the –n option.

num_initially_infected

the total number of people who are initially infected. 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 **number_of_people**.

num_infected

acount of the number of infected people. This value changes throughout the course of the simulation.

num_susceptible

acount of the number of susceptible people. This value changes throughout the course of the simulation.

our_num_immune

acount of the number of immune people. This value changes throughout the course of the simulation.

our_num_dead

acount of the number of dead people. This value changes throughout the course of the simulation.

x_locations

array, holds the x locations of all of the people; used if the environment needs to be displayed.

y_locations

array, holds the y locations of all of the people; used if the environment needs to be displayed.

infected_x_locations

array, used in **susceptible()** to keep track of the x locations of the infected people.

infected_y_locations

array, used in **susceptible()** to keep track of the y locations of the infected people.

states

array, holds the states of all of the people; used if the environment needs to be displayed.

num_days_infected

array, used to keep track of the number of days each person has been infected.

3.3 const_t struct

```
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

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

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

duration_of_disease

see the introduction chapter. The value of this variable can be specified on the command line with the –T option.

contagiousness_factor

see the introduction chapter. The value of this variable can be specified on the command line with the –c option.

deadliness_factor

see the introduction chapter. 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

```
// Data being used for SHOW_RESULTS
struct stats_t
{
    double num_infections;
    double num_infection_attempts;
    double num_deaths;
    double 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.

CHAPTER

FOUR

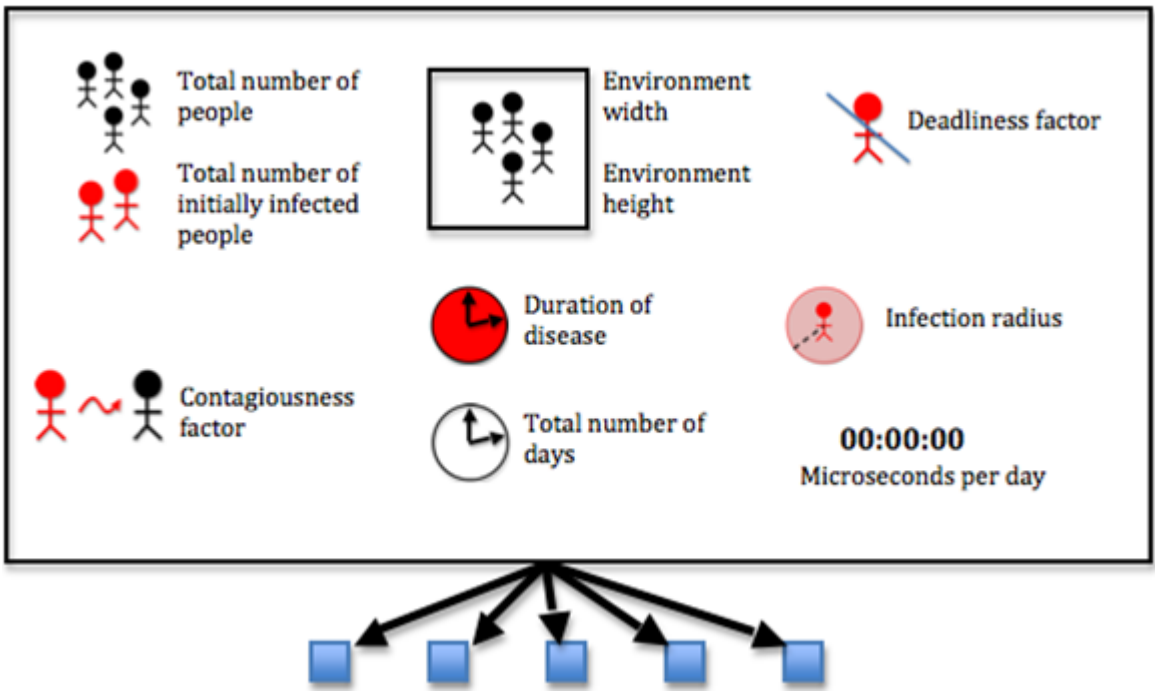
INITIALIZE FUNCTIONS

4.1 init

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

Then, **Init()** function will call the following 5 functions to finish the initialization process.

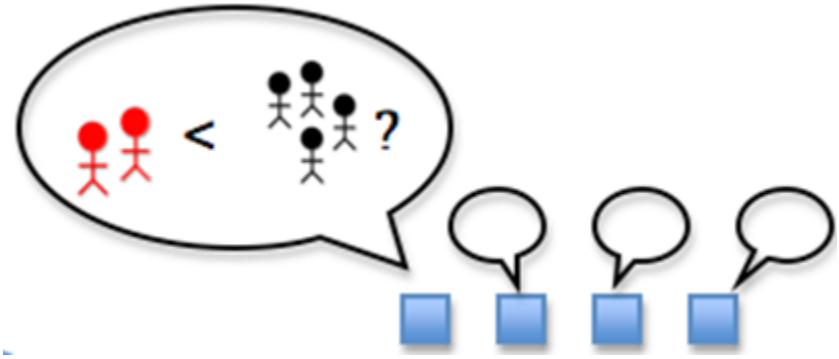
4.2 parse_args



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 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 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(number_of_people * sizeof(int));
```

Here we see that malloc has taken an argument, **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 **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:

```
dpy->environment = (char**)malloc(constant->environment_width *
    constant->environment_height * sizeof(char*));
int current_location_x;
for(current_location_x = 0;
    current_location_x <= constant->environment_width - 1;
        current_location_x++)
{
```

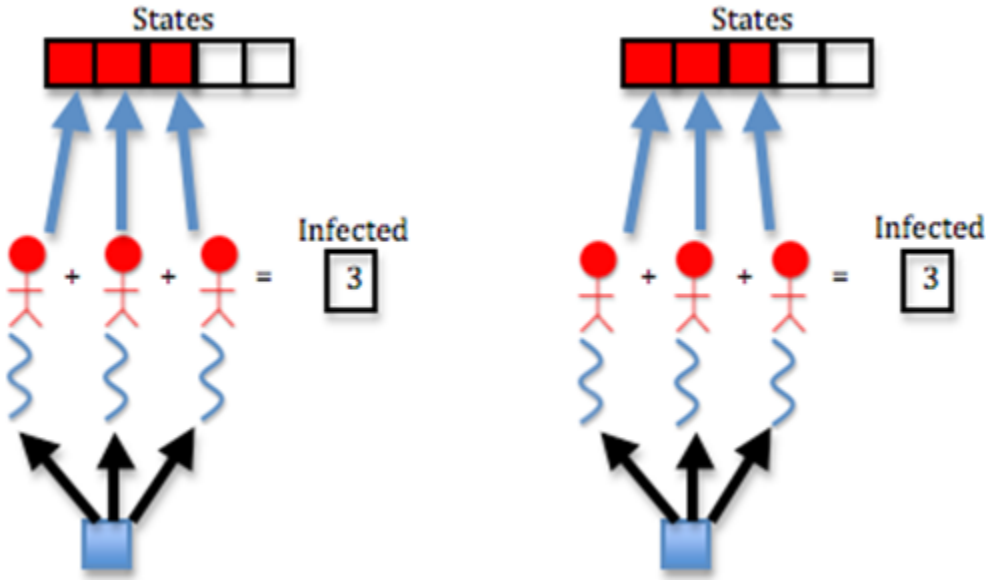
```
        dpy->environment[current_location_x] = (char*)malloc(
            constant->environment_height * sizeof(char));
    }
```

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).

4.5 init_array

First, the function set the states of the initially infected people and set the count of its infected people

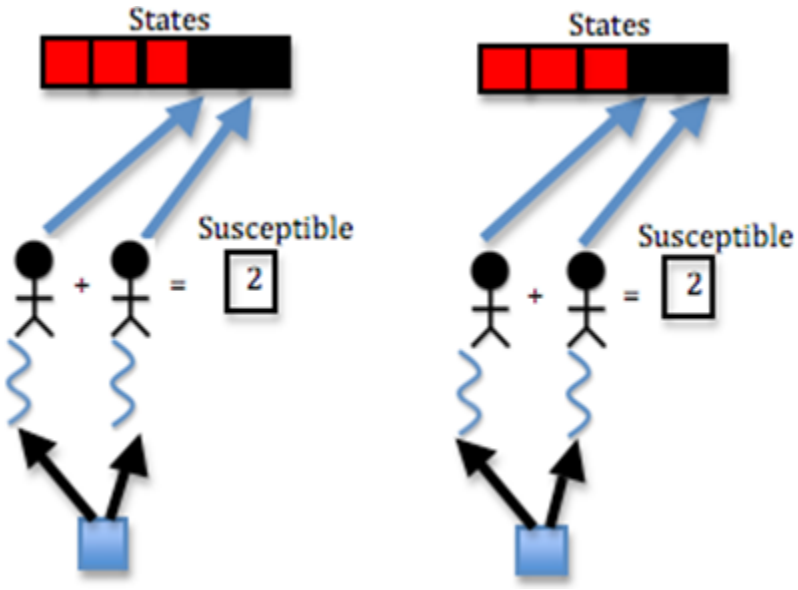


Threads set the states of infected people using the **states** array. They fill the first **num_initially_infected** cells of the array with the **INFECTED** constant; i.e. they fill in the **0** through **num_initially_infected – 1** positions of the array with **INFECTED** as below:

```
    // Each process spawns threads to set the states of the initially
    // infected people and set the count of its infected people
    for(current_person_id = 0; current_person_id
        <= num_initially_infected - 1; current_person_id++)
    {
        global->states[current_person_id] = INFECTED;
        global->num_infected++;
    }
```

Note we also add 1 to the **num_infected variable** using the plus-plus operator (++) at each iteration of the loop. This is how we count the number of infected people.

Next, the function set 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 **num_initially_infected** to **number_of_people - 1**) with the **SUSCEPTIBLE** constant, and we want to add **1** to the **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(current_person_id = num_initially_infected;
    current_person_id <= number_of_people - 1;
    current_person_id++)
{
    global->states[current_person_id] = SUSCEPTIBLE;
    global->num_susceptible++;
}
```

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

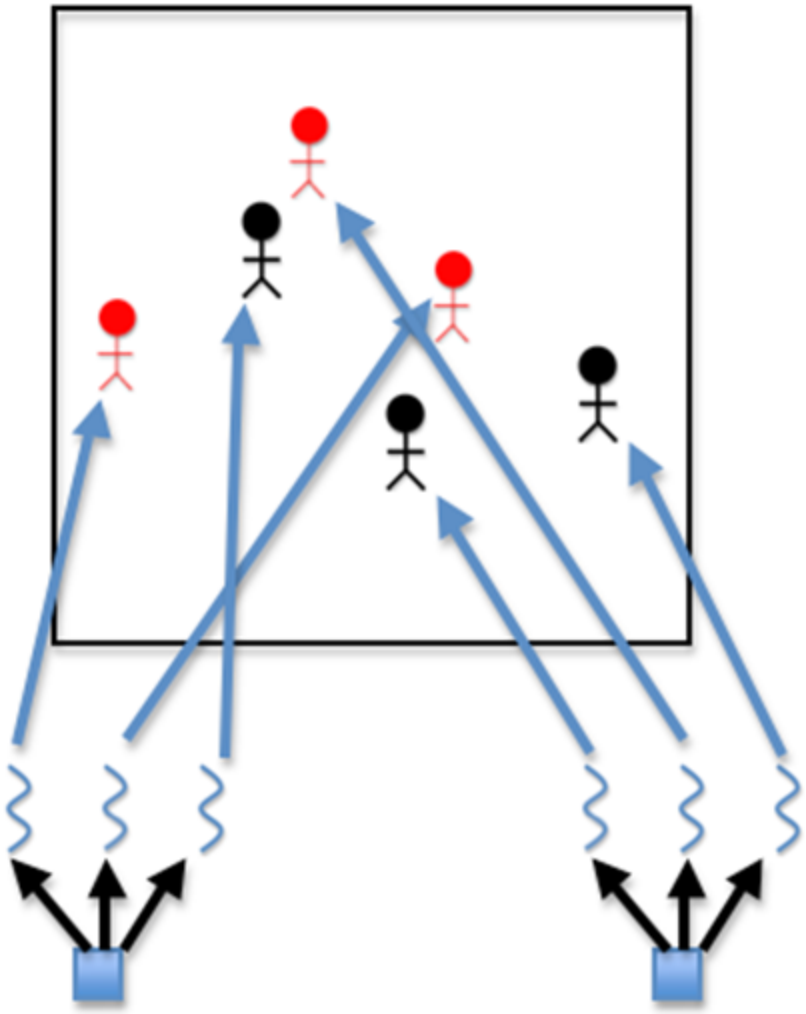
Then, the function sets random x and y locations for each people

Locations of people are stored in the **x_locations** and **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(current_person_id = 0;
    current_person_id <= number_of_people - 1;
    current_person_id++)
{
    global->x_locations[current_person_id] = random() % constant->environment_width;
    global->y_locations[current_person_id] = random() % constant->environment_height;
}
```

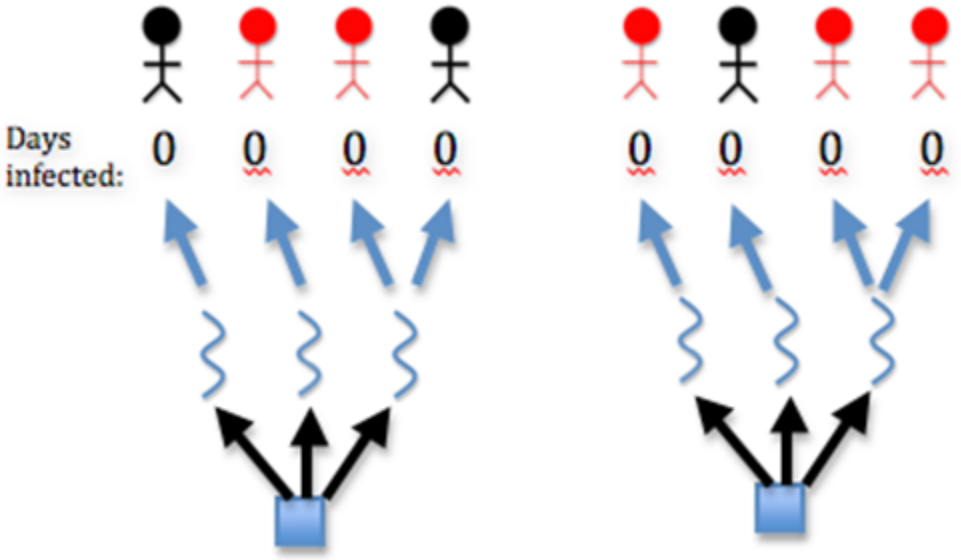
By calling random with the **modulus (%)** operator, we can restrict the size of the random number it generates. Since we cannot have x locations larger than the width of the environment, we call **random() % environment_width**; to make sure the **x location** of each person is less than **environment_width**. 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 **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 **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(current_person_id = 0;
    current_person_id <= number_of_people - 1;
    current_person_id++)
{
    global->num_days_infected[current_person_id] = 0;
}
```

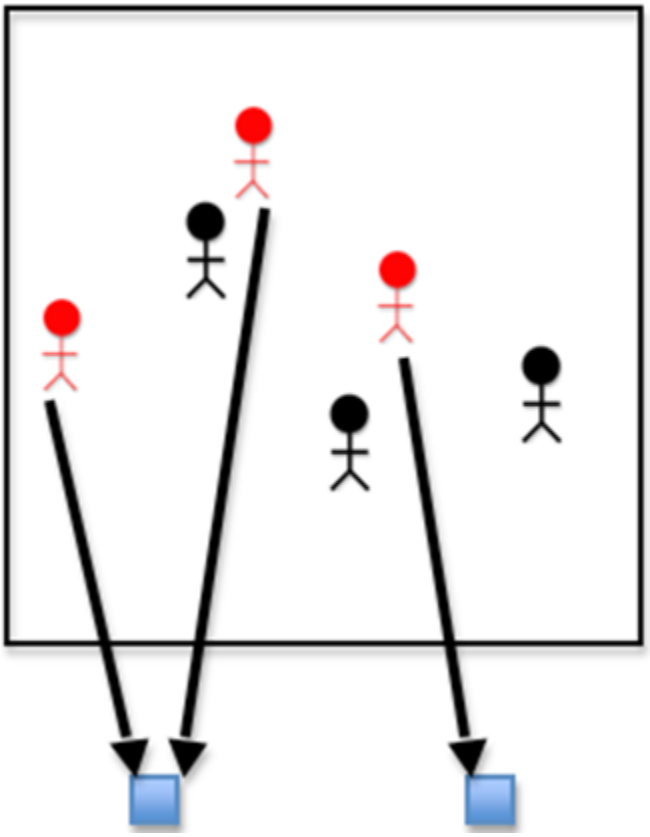
CHAPTER

FIVE

INFECTION FUNCTIONS

5.1 find_infected

This function determines the **x_location** and **y_location** of all the infected people.



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 **infected_x_locations** and **infected_y_locations** arrays. We do this by marching through the **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 **x_locations** and **y_locations** arrays to the **infected_x_locations** and **infected_y_locations** arrays. We determine the ID of the current infected person using the **current_infected_person** variable:

```
int current_infected_person = 0;
for(current_person_id = 0; current_person_id <= global->number_of_people - 1;
    current_person_id++)
{
    if(global->states[current_person_id] == INFECTED)
    {
        global->infected_x_locations[current_infected_person] =
        global->x_locations[current_person_id];
        global->infected_y_locations[current_infected_person] =
        global->y_locations[current_person_id];
        current_infected_person++;
    }
}
```

CHAPTER

SIX

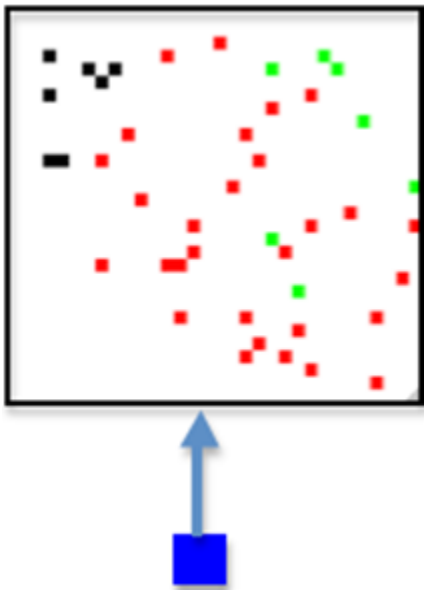
DISPLAY FUNCTIONS

6.1 init_display

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

6.2 do_display

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



6.3 close_display

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

6.4 throttle

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

CHAPTER

SEVEN

CORE FUNCTIONS

7.1 move

If the person is not dead, then

```
if(states[current_person_id] != DEAD)
```

First, the function 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).

then the function 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 left
// or right or does not move in the x dimension
x_move_direction = (random() % 3) - 1;

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

Next, If the person will remain in the bounds of the environment after moving, then

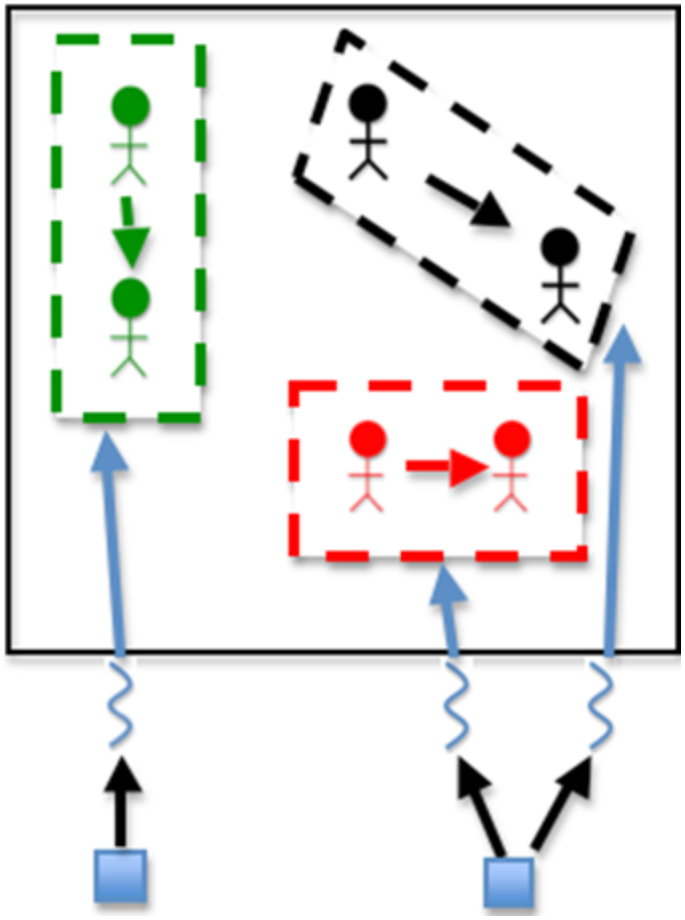
We check this by making sure the person's x location is greater than or equal to 0 and less than the width of the environment and that the person's y location is greater than or equal to 0 and less than the height of the environment. In the code, it looks like this:

```
if((x_locations[current_person_id] + x_move_direction >= 0)
    && (x_locations[current_person_id]
        + x_move_direction < environment_width)
    && (y_locations[current_person_id]
        + y_move_direction >= 0)
    && (y_locations[current_person_id]
        + y_move_direction < environment_height))
```

Finally, The function moves the person

```
    x_locations[current_person_id]
    += x_move_direction;
    y_locations[current_person_id]
    += y_move_direction;
```

The function is able to achieve this by simply changing values in the **x_locations** and **y_locations** arrays.



7.2 susceptible

For each people, the function to do the following

```
for(current_person_id = 0; current_person_id
    <= global->number_of_people - 1; current_person_id++)
```

If the person is susceptible,

```
if(states[current_person_id] == SUSCEPTIBLE)
```

For each of the infected people or until the number of infected people nearby is 1, the function does the following

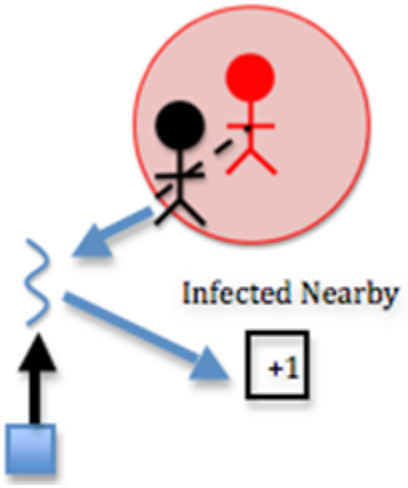
```
for(my_person = 0; my_person <= global->num_infected - 1
    && num_infected_nearby < 1; my_person++)
```

If the person is within the infection radius, then

```
if((x_locations[current_person_id]
    > infected_x_locations[my_person] - infection_radius)
    && (x_locations[current_person_id]
        < infected_x_locations[my_person] + infection_radius)
    && (y_locations[current_person_id]
        > infected_y_locations[my_person] - infection_radius)
    && (y_locations[current_person_id]
        < infected_y_locations[my_person] + infection_radius))
```

Finally, the function 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 function 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.

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 && (random() % 100)
    <= contagiousness_factor)
```

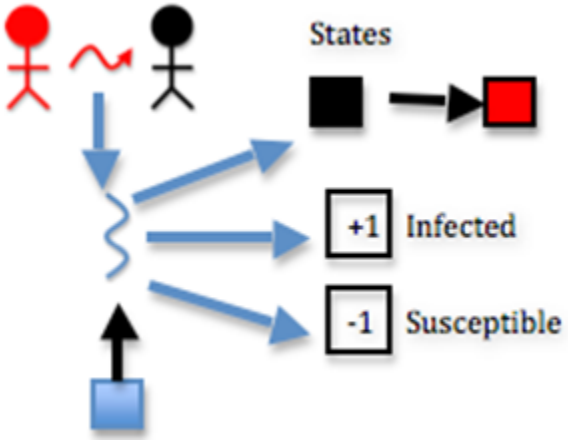
Recall that the contagiousness factor is the likelihood that the disease will be spread. We measure this as a number less than 100. For example, if there is a 30% chance of contagiousness, we use 30 as the value of the contagiousness factor. To figure out if the disease is spread for any given interaction 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 function changes the state to infected

```
states[current_person_id] = INFECTED;
```

The function updates the counters

```
// The thread updates the counters
global->num_infected++;
global->num_susceptible--;
```



These steps are as simple as updating the **states** array by **states[my_current_person_id] = INFECTED**, incrementing the **num_infected** variable, and decrementing the **num_susceptible** variable.

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

7.3 infected

For each people, the function to do the following

```
for(current_person_id = 0; current_person_id
    <= global->number_of_people - 1; current_person_id++)
```

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

```
if(states[current_person_id] == INFECTED
    && num_days_infected[current_person_id]
    == duration_of_disease)
```

Note in the code that if we have **SHOW_RESULTS** enabled, we increment the **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.

```
stats->num_recovery_attempts++;
```

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

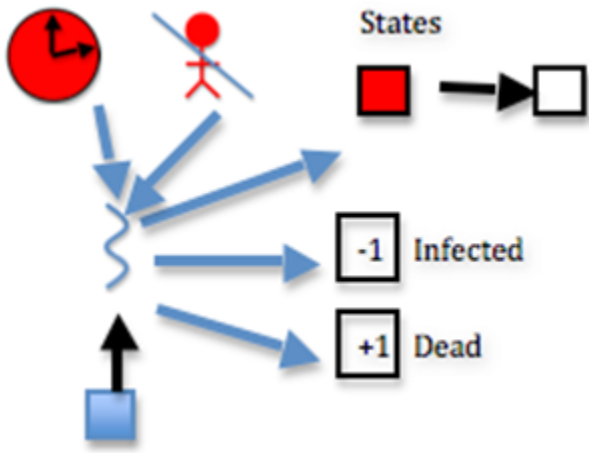
```
if((random() % 100) < deadliness_factor)
```

The function changes the person's state to dead

```
states[current_person_id] = DEAD;
```

The function updates the counters

```
// The thread updates the counters
global->num_dead++;
global->num_infected--;
```



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

Otherwise,

The function changes the person's state to immune

```
states[current_person_id] = IMMUNE;
```

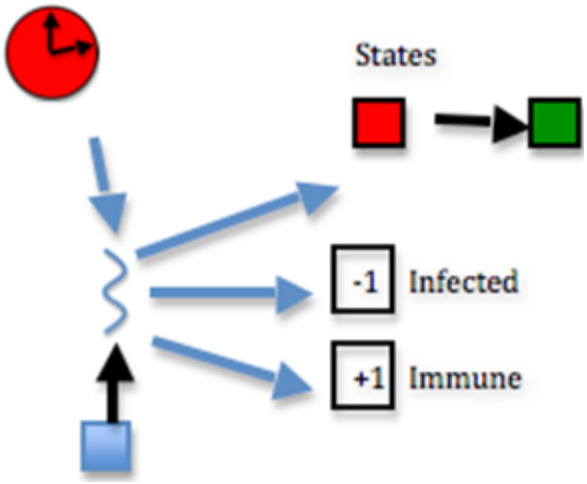
The function updates the counters

```
// The thread updates the counters
global->num_immune++;
global->num_infected--;
```

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** 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->num_deaths++;
#endif
```



7.4 update_days_infected

For each people, the function to do the following

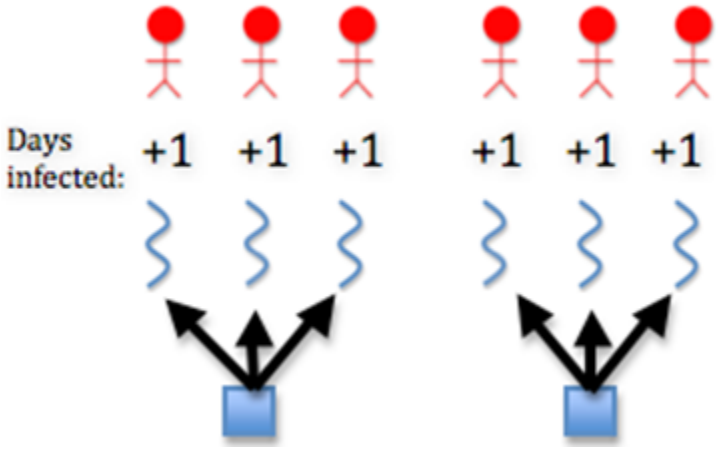
```
for(current_person_id = 0; current_person_id
    <= global->number_of_people - 1; current_person_id++)
```

If the person is infected, then

```
if(states[current_person_id] == INFECTED)
```

Increment the number of days the person has been infected

```
num_days_infected[current_person_id]++;
```



This is achieved by incrementing each member of the **num_days_infected** array, which can be done as follows: **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):

```
100.0 * (stats->num_infections / (stats->num_infection_attempts
        == 0 ? 1 : stats->num_infection_attempts)),
100.0 * (stats->num_deaths / (stats->num_recovery_attempts
        == 0 ? 1 : stats->num_recovery_attempts)));
```

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 **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 **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

```
close_display(dpy);
```

Since we allocated 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 current_location_x;
    for(current_location_x = constant->environment_width - 1;
            current_location_x >= 0; current_location_x--)
    {
            free(dpy->environment[current_location_x]);
    }
    free(dpy->environment);
    #endif

    // free arrays allocated in global struct
```

```
free(global->x_locations);
free(global->y_locations);
free(global->infected_y_locations);
free(global->infected_x_locations);
free(global->states);
free(global->num_days_infected);
```

CHAPTER

NINE

BUILD AND RUN

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 12-14 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.

If you want to use a text display instead of the graphical X11 display, uncomment line 20 and comment line 22. When rigging the code to test for performance, you might want to eliminate the display of each iteration completely, in which case you can comment line 20 and 22.

```
1  # DESCRIPTION:  Makefile for serial code 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 Weedon
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 CC=gcc
12 #XLIB_LOC=/opt/X11/lib    #Mac OS X XQuartz installed here
13 #XLIB_LOC=/usr/X11R6/lib   #some unix systems may have this
14 #XLIB_INC=/opt/X11/include
15
16 ifdef ICC
17         CC=icc
18 endif
19
20 #CFLAGS+=-DTEXT_DISPLAY # Uncomment to show text display
21
22 CFLAGS+=-DX_DISPLAY -I $(XLIB_INC) -L$(XLIB_LOC) -lX11 # Uncomment to show X display
23
24 CFLAGS+=-DSHOW_RESULTS # Uncomment to make the program print its results
25
26 # Source files
27 SRCS=$(PROGRAM_PREFIX).c
28
29 # Make targets
30 all: $(PROGRAM_PREFIX)-serial
31
32 clean:
```

```
33          rm -f $(PROGRAM_PREFIX)-serial
34
35  run:
36          ./$(PROGRAM_PREFIX)-serial
37
38  # Make rules
39  $(PROGRAM_PREFIX)-serial: $(SRCS)
40          $(CC) -o $(PROGRAM_PREFIX)-serial $(SRCS) $(CFLAGS)
41
42  $(SRCS): Core.h Defaults.h Display.h Finalize.h Infection.h Initialize.h
```

9.1 Build

```
make
```

9.2 Run

```
./Pandemic-serial
```

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

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

```
./Pandemic-serial -?
```

It should give you something like this:

```
./Pandemic-serial: illegal option -- ?

Usage: ./Pandemic-serial [-n number_of_people][-i num_initially_infected
[-w environment_width][-h environment_height][-t total_number_of_days]
[-T duration_of_disease][-c contagiousness_factor][-d infection_radius]
[-D deadliness_factor][-m microseconds_per_day]
```

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

If you comment out lines 20 and 22 of the Makefile shown above, you can try getting some preliminary basic sense of how fast the program runs with various sizes of the problem (in this case the number of people). First, comment lines 20 and 22 in the Makefile and do the following:

```
make clean
make
```

Then execute various problem sizes, taking no time between iterations, as follows:

```
time ./Pandemic-serial -n 20000 -m 0
```

Experiment by changing the value of n higher and lower. You should see the program take more time as you increase the number of people. Experiment with some of the other parameters also. Do some investigation of what the unix time command does. This is a very rough way to see how fast your program runs. There are preferable ways to instrument the code itself that you could investigate.

CHAPTER

TEN

INCLUDING OPENMP

`Download Pandemic-OMP.tgz`

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 two counters in the first two parts of the function. Different from most parallel structure, reduction in OpenMP is pretty easy to implement. We just need to add a reduction literal,

```
#pragma omp parallel for private(current_person_id) \
    reduction(+:num_infected_local)
```

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

```
int num_infected_local = global->num_infected;
```

we can then, reduce to local instance,

```
    num_infected_local++;
```

Finally, we put local instance back to struct.

```
global->num_infected = 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

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 12-14 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 20 and 22 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 24 uncommented to see the final statistics after the whole simulation is completed.

```
1  # DESCRIPTION:  Makefile for OpenMP code 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 Weedon
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 CC=gcc
12 XLIB_LOC=/opt/X11/lib    #Mac OS X XQuartz installed here
13 #XLIB_LOC=/usr/X11R6/lib   #some unix systems may have this
14 XLIB_INC=/opt/X11/include    #Mac OS X XQuartz installed here
15
16
17 # OpenMP
18 OPENMP_FLAGS=-fopenmp
19
20 #CFLAGS+=-DTEXT_DISPLAY # Uncomment to show text display
21
22 #CFLAGS+=-DX_DISPLAY -I $(XLIB_INC) -L$(XLIB_LOC) -lX11 # Uncomment to show X display
23
24 CFLAGS+=-DSHOW_RESULTS # Uncomment to make the program print its results
25
26 # Source files
27 SRCS=$(PROGRAM_PREFIX).c
28
```

```
# Make targets
all: $(PROGRAM_PREFIX)-openmp

clean:
        rm -f $(PROGRAM_PREFIX)-openmp

run:
        ./$(PROGRAM_PREFIX).c-openmp

# Make rules
$(PROGRAM_PREFIX)-openmp: $(SRCS)
        $(CC) -o $(PROGRAM_PREFIX)-openmp $(SRCS) $(OPENMP_FLAGS) $(CFLAGS)

$(SRCS): Core.h Defaults.h Display.h Finalize.h Infection.h Initialize.h
```

11.1 Build

```
make
```

11.2 Run

```
./Pandemic-openmp
```

The default values start with a simulation of approximately 50 people. The OpenMP code will also use a default number of threads, which it determines from your machine's hardware configuration.

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

```
./Pandemic-openmp -?
```

It should give you something like this:

```
/Pandemic-openmp -?

./Pandemic-openmp: illegal option -- ?
Usage: ./Pandemic-openmp [-n number_of_people][-i num_initially_infected][-w environment_width]
[-h environment_height][-t total_number_of_days][-T duration_of_disease]
[-c contagiousness_factor][-d infection_radius][-D deadliness_factor]
[-m microseconds_per_day] [-p number of threads]
```

Note that these are defined and set in the *parse_args()* function in Initialize.h. There is a new option, -p, for setting the number of threads.

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

```
time ./Pandemic-openmp -n 70000 -m 0 -p 4
time ./Pandemic-openmp -n 70000 -m 0 -p 8
```

11.3 To think about

There are preferable ways to instrument your code to time it, using the OpenMP function *opm_get_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 threads for a fairly large problem size?