

Parallelization: Infectious Disease

By Aaron Weeden, Shodor Education Foundation, Inc.

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 are 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
- MPI programming
- OpenMP programming
- Using a cluster
- Scaling parallel code

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.



Person (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.¹ 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.²

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 20th 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

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

¹ 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.

² Threads can also have their own private memories, and OpenMP has pragmas to define whether variables are public or private.

to be modeled³ 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.

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

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

³ 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.

Figure 2: 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 in Figure 3.

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

where

N = the number of processors

1 - P = the proportion of the program that cannot be made parallel

Figure 3: 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. 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.zip. 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.

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.

`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.

`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 steps IV and V of the algorithm.

`our_person1` - a loop iterator, used in step XIV . A of the algorithm to iterate over all of the people for which the current process is responsible.

`our_current_infected_person` - an iterator, used in step XIV . A of the algorithm to iterate over all the infected people for which the current process is responsible.

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

`our_num_infected` - the count of infected people for which the current process is responsible.

`our_current_location_x` - a loop iterator, used in step XIV . F of the algorithm to draw the environment.

`our_current_location_y` - a loop iterator, used in step XIV . F of the algorithm to draw the environment.

`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.

`my_current_person_id` - a loop iterator, used each time threads are spawned to handle people.

`my_num_infected_nearby` - used in step XIV . H of the algorithm to determine whether any infected people are nearby the current susceptible person.

`my_person2` - a loop iterator, used in step XIV . H . 1 . a to iterate over the infected people.

`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 Model section above. The value of this variable can be specified on the command line with the `-d` option.

`duration_of_disease` - see the Model section above. The value of this variable can be specified on the command line with the `-T` option.

`contagiousness_factor` - see the Model section above. The value of this variable can be specified on the command line with the `-c` option.

`deadliness_factor` - see the Model section above. The value of this variable can be specified on the command line with the `-D` option.

`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.

`total_number_of_days` - the total number of days over which to run the simulation.

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

`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.

`my_x_move_direction` - used in step XIV.G of the algorithm for moving people.

`my_y_move_direction` - used in step XIV.G of the algorithm for moving people.

`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 step I of the algorithm.

`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 step I of the algorithm.

`current_rank` – a loop iterator, used in step XIV.B of the algorithm to iterate over all of the processes.

`current_displ` – used in step XIV.B of the algorithm to set up the `displs` array.

`c` – used in the `getopt` function in step II of the algorithm to parse the arguments from the command line.

*`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.

*`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.

*`their_infected_x_locations` – array, used in step XIV.H of the algorithm 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 XIV.H of the algorithm to keep track of the y locations of the infected people for which each 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.

*`recvcunts` – array, used for any MPI calls that need it.

*`displs` – array, used for any MPI calls that need it.

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

`*our_states` – array, holds the states of all the people for which the current process is responsible.
`**environment` – 2D array, holds an ASCII representation of the environment (see “state” under “Person” in the “Model” section).

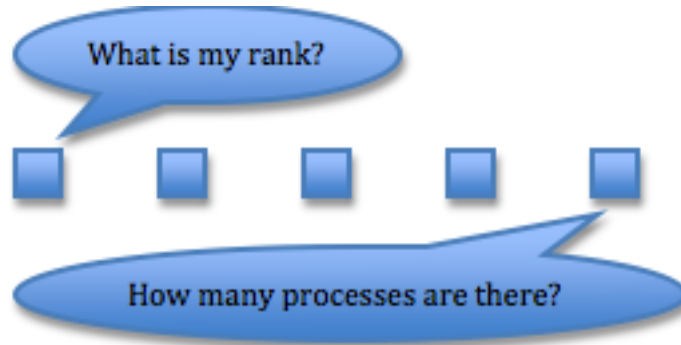
Students can use this Data Structures section and the attached code as reference as they read the Algorithm section.

Algorithm

Before executing the algorithm, the code starts by initializing MPI using the `MPI_Init(&argc, &argv);` function. 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 step I of the algorithm.

Once `MPI_Init` has been called, we begin the algorithm.

- I. Each process determines its rank and the total number of processes



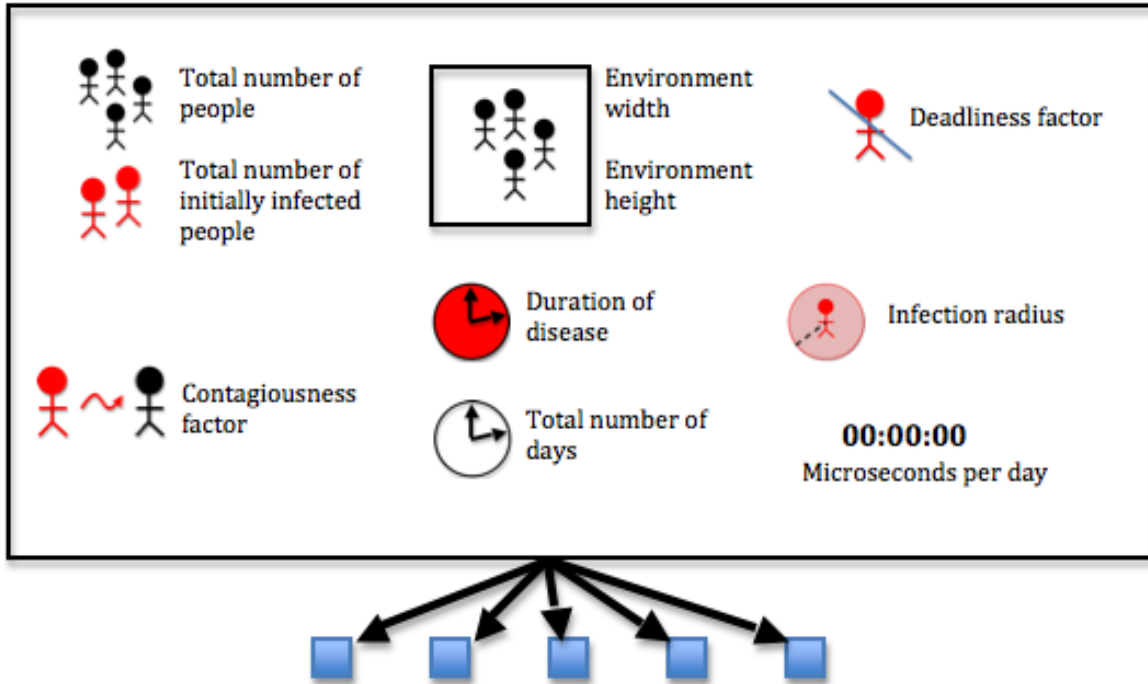
Here we see one process figuring out its rank. It does so by calling the `MPI_Comm_rank(MPI_COMM_WORLD, &our_rank);` function. This function checks the MPI “world” (the “communicator” of all the MPI processes, `MPI_COMM_WORLD`). You pass the address of the variable for the process’s rank to the function 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;`

We also see another process figuring out how many processes there are. It does so by calling the `MPI_Comm_size(MPI_COMM_WORLD, &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.

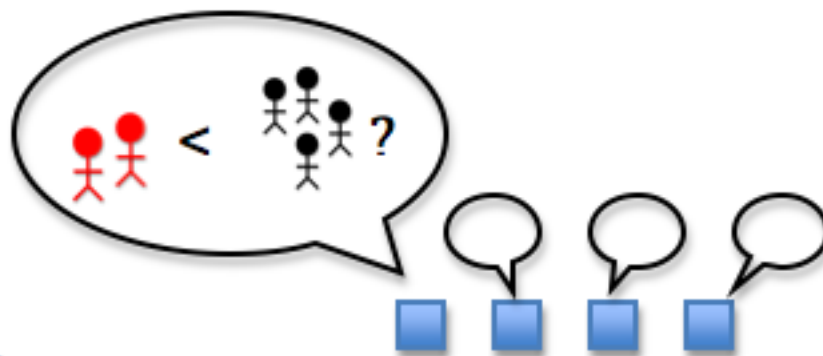
If we have only 1 process total, we set the number of processes by calling `total_number_of_processes = 1;`

II. Each process is given the parameters of the simulation



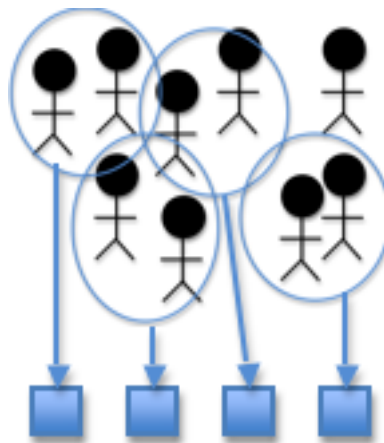
These parameters are specified via command line arguments when the program is run. Otherwise, default values are used. The code uses `getopt` function to do this. Type `man 3 getopt` in a shell if you are interested how it works.

III. Each process 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.

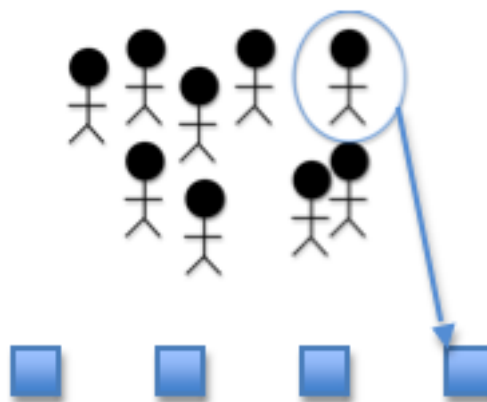
IV. Each process determines the number of people for which it is responsible



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_number_of_people = total_number_of_people /  
total_number_of_processes;
```

V. The last process is responsible for the remainder

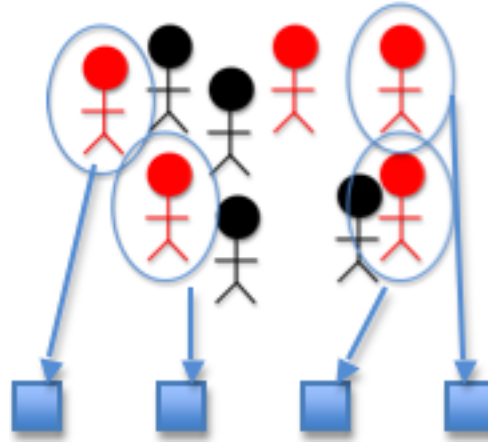


Every person has to be accounted for, so any remainder of the division is assigned to the last process. We can obtain the remainder by using the modulo operator (`%`), and we add it to the existing value using the plus-equals operator (`+=`):

```
our_number_of_people += total_number_of_people %
```

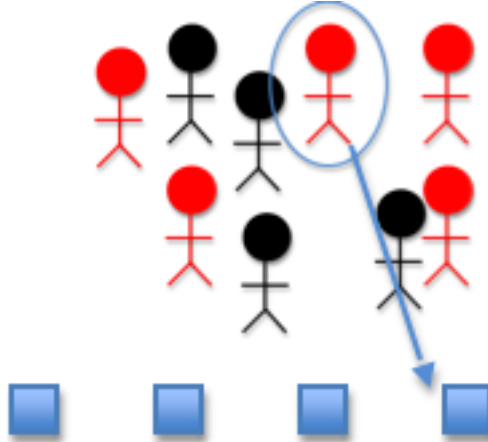
`total_number_of_processes;`. We only want the last process to do this, so we surround the code with `if(our_rank == total_number_of_processes - 1)`, since the last process has rank $N - 1$, where N is the total number of processes.

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



This is the same method used in step IV, but it considers only the infected people.

VII. The last process is responsible for the remainder



This is the same method used in step V, but it considers only the infected people.

After this step 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:

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

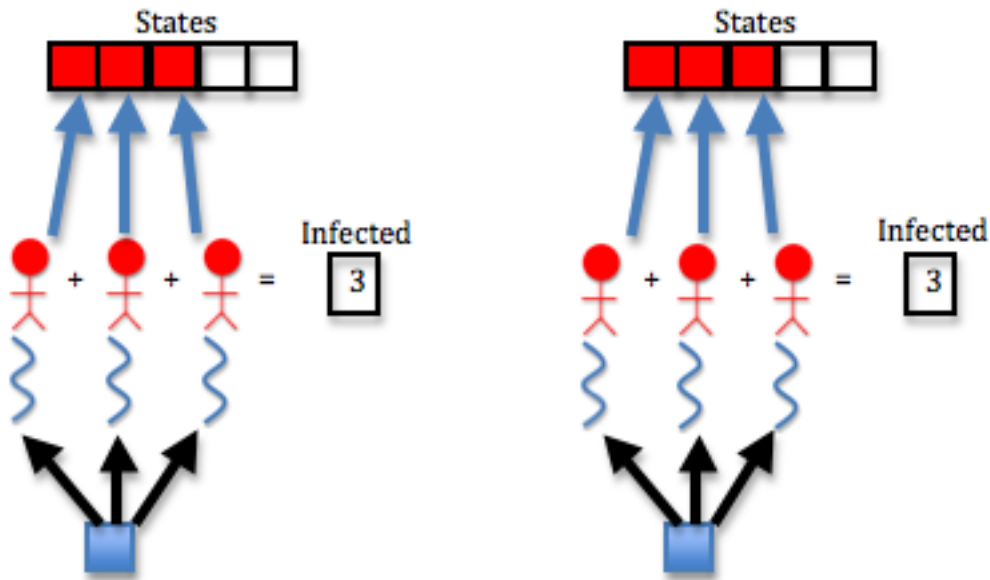
The number of arrays we need is stored in `environment_width`, so we loop from 0 to `environment_width - 1` and allocate enough space in each element of `environment` for `environment_height` chars, each one of which has size `sizeof(char)`.

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).

VIII. Each process seeds the random number generator based on the current time

“Seeding” a random number generator means providing it with an initial value from which other numbers are generated algorithmically. This means the generator will be “pseudorandom”, because it is not actually producing numbers in a random fashion, but the numbers themselves are acceptably random, provided the seed is different each time the program is run. Thus, we pass it the current time, which by definition is always changing. We do this in the code using the `srandom` function, which calls the `time` function: `srandom(time(NULL));`

IX. Each process spawns threads to set the states of the initially infected people and set the count of its infected people



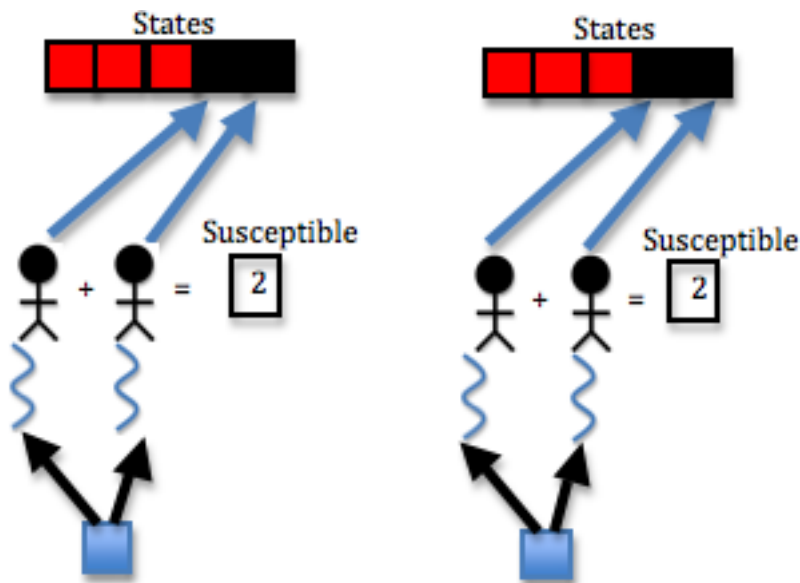
The code spawns threads using the `#pragma omp parallel for` line, which parallelizes the `for` loop below it; threads execute the iterations of the loop in parallel.

Threads set 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:

```
for(my_current_person_id = 0; my_current_person_id
    <= our_num_initially_infected - 1;
    my_current_person_id++)
{
    our_states[my_current_person_id] = INFECTED;
    our_num_infected++;
}
```


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

X. Each process spawns threads to set the states of the rest of its people and set the count of its susceptible people

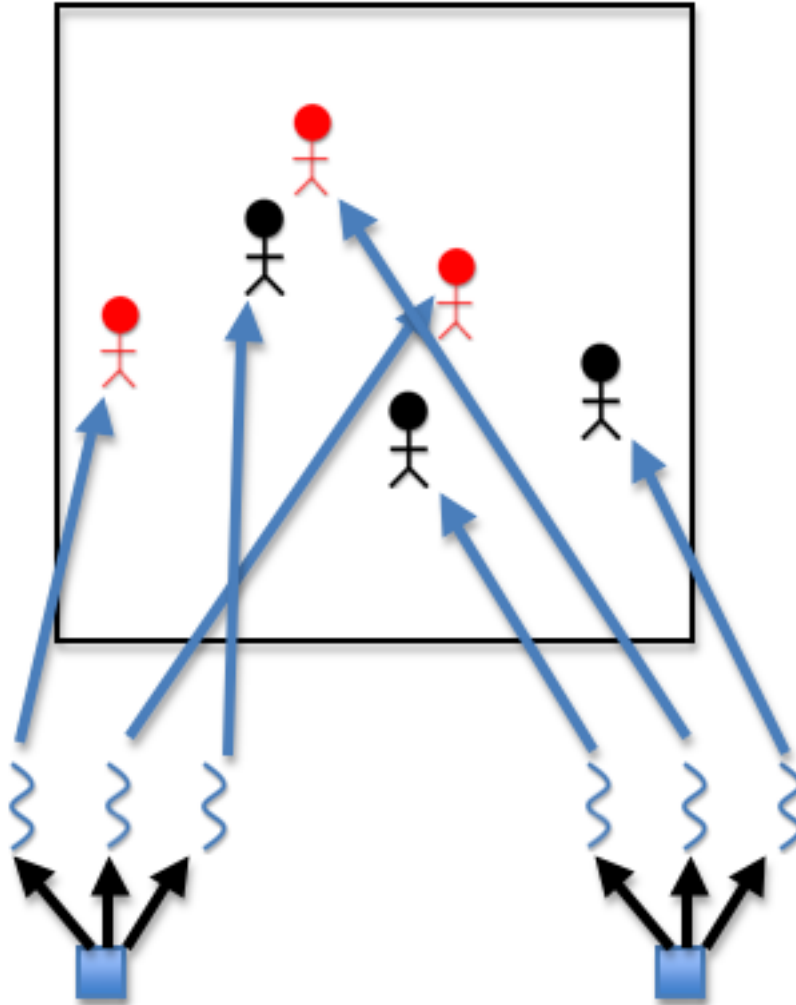


This is similar to step IX, but we want to fill the rest of the array (from `our_num_initially_infected` to `our_number_of_people - 1`) with the `SUSCEPTIBLE` constant, and we want to add 1 to the `our_num_susceptible` variable at each iteration of the loop:

```
#pragma omp parallel for \
    private(my_current_person_id) \
    reduction(+:our_num_susceptible)
for(my_current_person_id = our_num_initially_infected;
    my_current_person_id
        <= our_number_of_people + 1;
    my_current_person_id++)
{
    our_states[my_current_person_id] = SUSCEPTIBLE;
    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.

XI. Each process spawns threads to set 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:

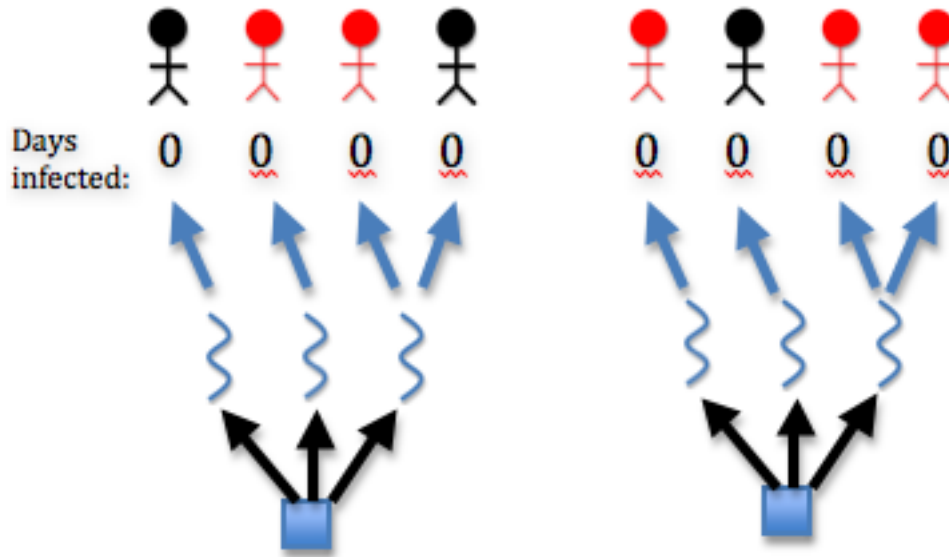
```
#pragma omp parallel for private(my_current_person_id)
for(my_current_person_id = 0;
    my_current_person_id <= our_number_of_people - 1;
    my_current_person_id++)
{
    our_x_locations[my_current_person_id] = random()
        % environment_width;
    our_y_locations[my_current_person_id] = random()
        % 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`.

XII. Each process spawns threads to initialize 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:

```
#pragma omp parallel for private(my_current_person_id)
for(my_current_person_id = 0;
    my_current_person_id <= our_number_of_people - 1;
    my_current_person_id++)
{
    our_num_days_infected[my_current_person_id] = 0;
}
```

XIII. Rank 0 initializes the graphics display

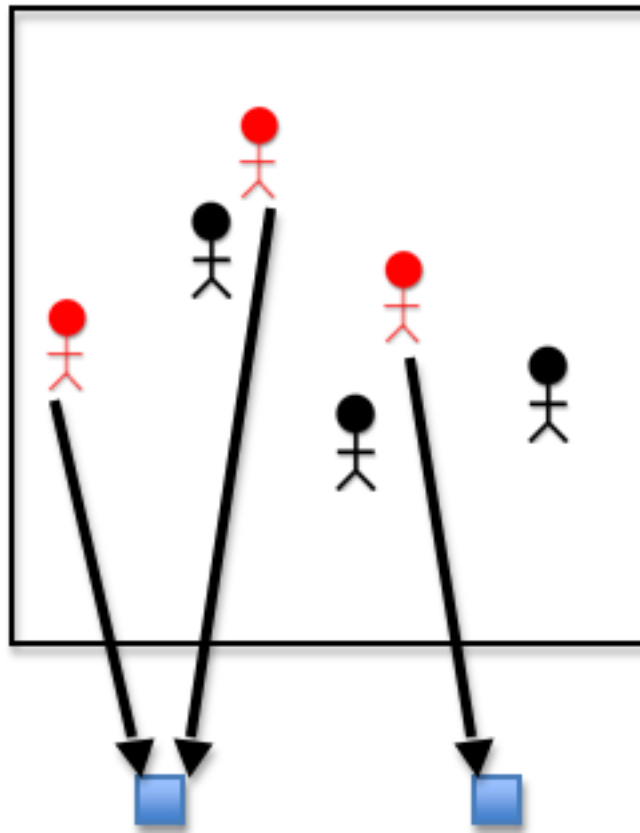
The code uses X to handle graphics display. See the comments under ALG XIII in the code if you are interested in how X works.

XIV. Each process starts a loop to run the simulation for the specified number of days

The code uses a for loop from 0 to `total_number_of_days - 1`, which covers each of the days for which the simulation will run:

```
for(our_current_day = 0;  
    our_current_day <= total_number_of_days - 1;  
    our_current_day++)
```

A. Each process 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:

```
our_current_infected_person = 0;
```

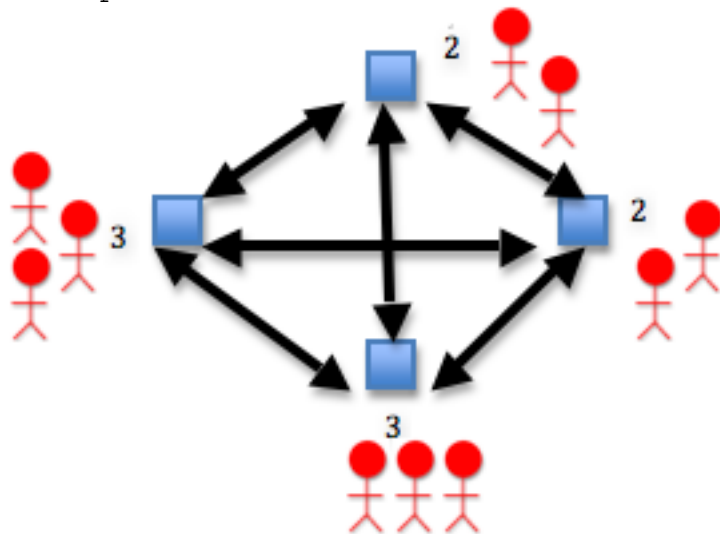
```

for(our_person1 = 0;
    our_person1 <= our_number_of_people - 1;
    our_person1++)
{
    if(our_states[our_person1] == INFECTED)
    {
our_infected_x_locations[our_current_infected_person] =
        our_x_locations[our_person1];

our_infected_y_locations[our_current_infected_person] =
        our_y_locations[our_person1];
        our_current_infected_person++;
    }
}

```

B. Each process sends its count of infected people to all the other processes and receives their counts



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:

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

C. Each process 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` (note the `v`). 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:

```
current_displ = 0;
for(current_rank = 0;
    current_rank <= total_number_of_processes - 1;
    current_rank++)
{
    displs[current_rank] = current_displ;
    current_displ += recvcounts[current_rank];
}
```

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 (note that we obtained this in step XIV.B.).
- `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.

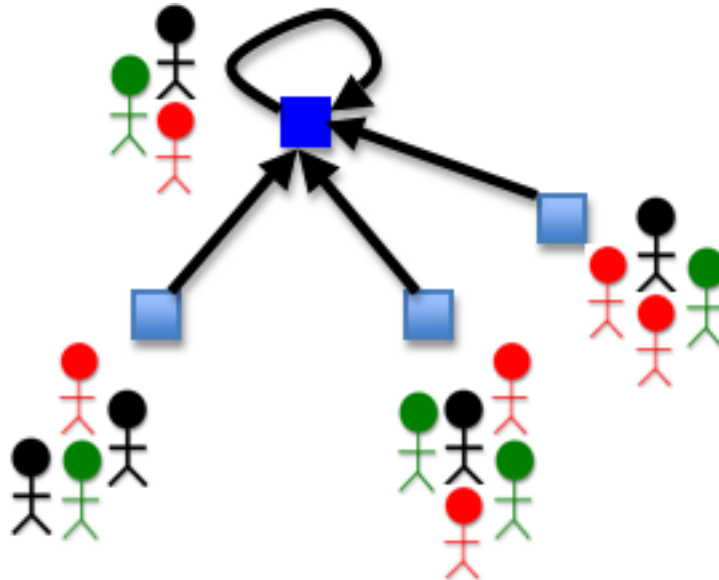
Once the command is complete, each process will have the full array of the x locations of the infected people from each process, stored in the `their_infected_x_locations` array.

D. Each process sends the y locations of its infected people to all the other processes and receives the y locations of their infected people

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 step XIV.C. are replaced by y's in the `Allgatherv` in this step.

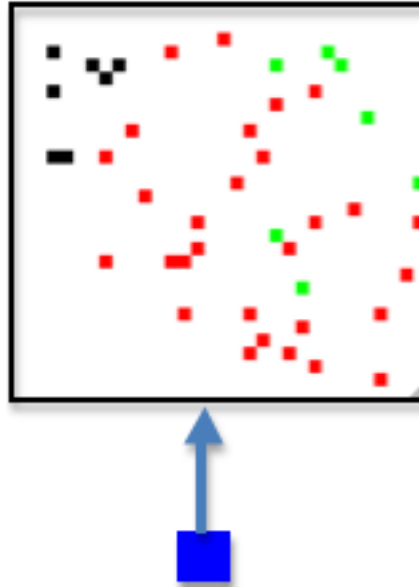
Note that the code will only execute steps XIV.C. and XIV.D. if MPI is enabled. If it is not enabled, the code simply copies the `our_infected_x_locations` and `our_infected_y_locations` arrays into the `their_infected_x_locations` and `their_infected_y_locations` arrays and the `our_num_infected` variable into the `total_num_infected` variable.

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

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



- G.** For each of the process's people, each process spawns threads to do the following
- 1.** If the person is not dead, then
 - a.** 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 (1), stay in place (0), or move to the left (-1).

- b.** The thread randomly picks whether the person moves up or down or does not move in the y dimension

This is similar to XIV.G.1.a.

- c.** 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((our_x_locations[my_current_person_id]
    + my_x_move_direction >= 0)
```

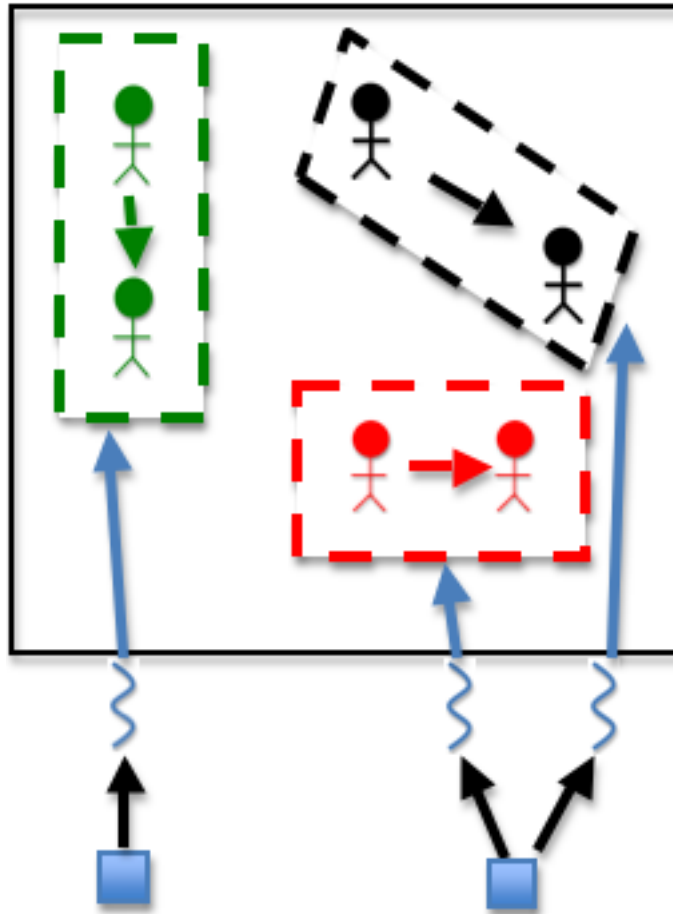


```

    && (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))

```

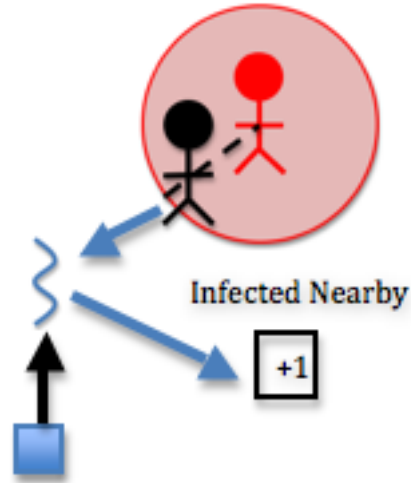
i. The thread moves the person



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

- H. For each of the process's people, each process spawns threads to do the following
 - 1. If the person is susceptible, then
 - a. 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
 - i. If person 1 is within the infection radius, then

1. The thread increments the number of infected people 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. Step XIV.H.1.a appears complicated in the code, but it is actually fairly similar to step XIV.G.1.c.

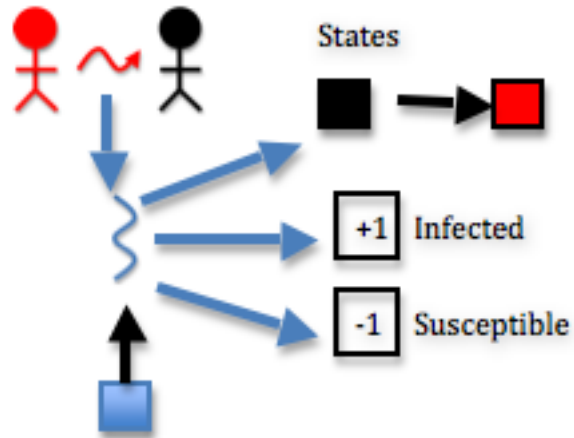
Note in the code that if the number of infected nearby is greater than or equal to 1 and we have `SHOW_RESULTS` enabled, we increment the `our_num_infection_attempts` variable. This helps us keep track of the number of attempted infections, which will help us calculate the actual contagiousness of the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

- b. 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

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).

- i. The thread changes `person1`'s state to infected

ii. The thread updates the counters



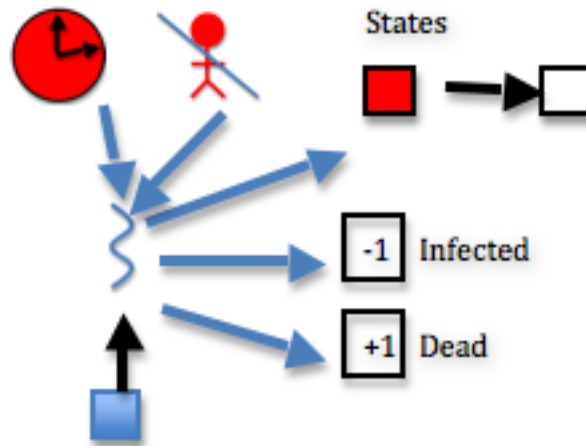
These steps are as simple as updating the `our_states` array by `our_states[my_current_person_id] = INFECTED;`, incrementing the `our_num_infected` variable, and decrementing the `our_num_susceptible` variable.

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.

- I.** For each of the process's people, each process spawns threads to do the following
 - 1.** If the person is infected and has been for the full duration of the disease, then

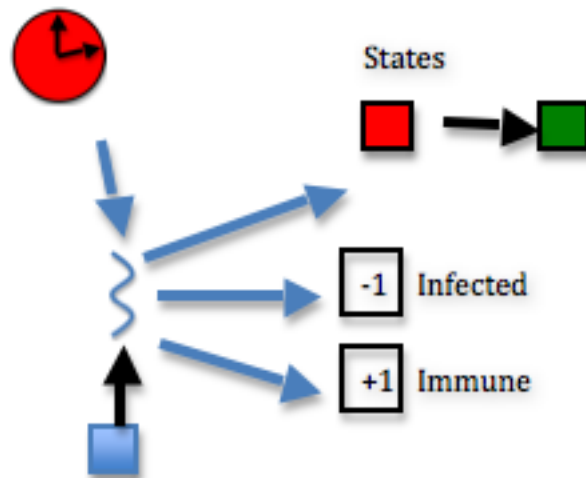
Note in the code that if we have `SHOW_RESULTS` enabled, we increment the `our_num_recovery_attempts` variable. This helps us keep track of the number of attempted recoveries, which will help us calculate the actual deadline of the disease at the end of the simulation.

- a.** If a random number less than 100 is less than the deadliness factor, then
 - i.** The thread changes the person's state to dead
 - ii.** The thread updates the counters



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

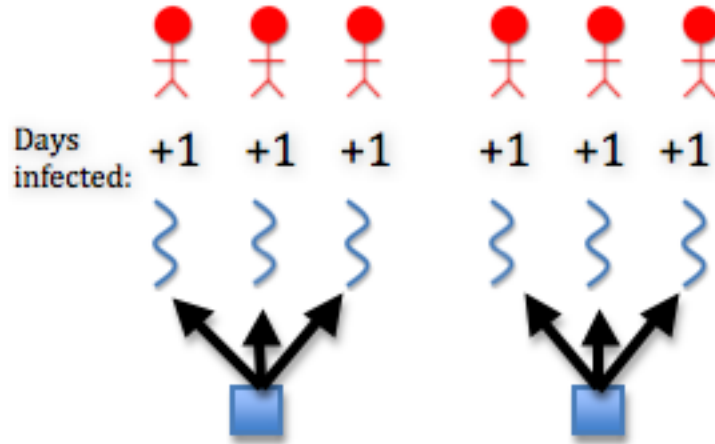
- b. Otherwise,
 - i. The thread changes the person's state to immune
 - ii. The thread updates the counters



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.

- J. For each of the process's people, each process spawns threads to do the following
 - 1. If the person is infected, then
 - a. Increment the number of days the person has been infected



This is achieved by incrementing each member of the `our_num_days_infected` array, which can be done as follows:

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

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

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 deadline of the disease. To perform these two calculations, we use the following code (using the contagiousness as the example):

```
100.0 * (our_num_infections /
        (our_num_infection_attempts == 0 ? 1
         : our_num_infection_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 `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 deadline factor.

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.

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`.

Exercise 1 – Exploring the Model

This exercise requires running the program and setting various options. See the previous section for information on setting options. You can run any version of the program in this exercise.

1. What do you observe with a small environment (e.g. -w 20 -h 20) and a large number of people (e.g. -n 100)?
2. What do you observe with a large environment (e.g. -w 100 -h 100) and the same large number of people (e.g. -n 100)?
3. What do you observe with a very long but narrow environment with a large number of people (e.g. -n 100 -w 100 -h 10)?
4. How does the time it takes to infect the entire population vary as you adjust the contagiousness factor (e.g. -c 20, -c 50, -c 80)?
5. How does the number of infected vary as you adjust the duration of the disease (e.g. -T 5, -T 20, -T 100)?
6. What happens with a very low contagiousness factor but very high duration of disease (e.g. -c 1 -T 1000)?
7. Explore changing other options. What options did you change? What did you observe?

Exercise 2 – Scaling the Model on A Non-cluster

Computers are powerful but are only so powerful by themselves. Scaling the code on high performance computers means that additional computers can be assigned to the problem, thereby making it possible to simulate a bigger model or solve a bigger problem that a single computer would have difficulty simulating or solving. In order for code to take advantage of high performance computers and clusters, it can employ parallelism to make use of the multiple cores that a computer has or the multiple compute nodes that a cluster rack has. The code attached to this module is written with parallel capabilities, and Exercise 4 explores scaling it on a cluster. First let's explore scaling on a serial computer.

Notes:

- The line in the Makefile that begins “-DSHOW_RESULTS” must be un-commented for this next exercise. If it not, un-comment it and re-compile the program.
 - For faster results, try commenting out the line in the Makefile that begins “-DX_DISPLAY” and re-compiling the program.
1. Run the program with 10 people (-n 10). What is the actual contagiousness and actual deadliness?

2. Run the program with 100 people (-n 100). What is the actual contagiousness and actual deadliness?
3. Run the program with 1000 people (-n 1000). What is the actual contagiousness and actual deadliness?
4. What trend are you seeing? What do you think accounts for this trend?
5. What are you noticing about how long the program takes to run?

Exercise 3 – Introduction to Parallel Programming on a Cluster

This exercise will take you through writing a small piece of C code, outfitting it with MPI and OpenMP directives, and running it on a cluster.

You will need to obtain an account from the Cluster Computing Group at Earlham. Send email to ccg@cs.earlham.edu to request an account.

You will make use of the **vi** text editor, which is provided by default on most Linux-based operating systems, such as the one used by Earlham’s cluster.

In this exercise, any line with a dollar sign (\$) in front of it is a command to be entered in a shell (a command line utility used by the operating system to interact with the user).

Part I: Write, compile, and run a serial program

1. Log into al-salam:

```
$ ssh <yourusername>@cluster.earlham.edu
$ ssh as0
```

2. Create a small “Hello, World” program in C:
 - a. Open a new file called `hello.c` in `vi`:

```
$ vi hello.c
```
 - b. Enter `vi`’s “insert mode” by pressing the **i** key.
 - c. Write a small C code that will print “Hello, World!” on the screen:

```
#include <stdio.h>

int main(int argc, char** argv)
{
    printf("Hello, World!\n");

    return 0;
}
```

- d. Press **Escape (esc)** to exit vi's insert mode.
 - e. Save the file and exit vi by entering **<Shift>-Z-Z**.
3. Compile the code with GNU's compiler. This will produce an executable file called `hello`:

```
$ gcc -o hello hello.c
$
```

If any errors are listed, make sure there are no typos in `hello.c` (go back through step 2).

4. Create a script to run the program on the cluster.
 - a. Open a new file called `hello.qsub` in vi:

```
$ vi hello.qsub
```
 - b. Enter insert mode (as you did in step 2b) and write a small Portable Batch System (PBS) script:

```
#PBS -q ec
#PBS -o hello.out
#PBS -e hello.err
```

```
cd $PBS_O_WORKDIR
```

```
./hello
```

Each line of this script tells the scheduler to do something:

`#PBS -q ec` says to use the "ec" queue.

`#PBS -o hello.out` says to save the output of standard out to a file called `hello.out` rather than to print it on the terminal.

`#PBS -e hello.err` says to save the output of standard error to a file called `hello.err` rather than to print it on the terminal.

`cd $PBS_O_WORKDIR` tells the scheduler to change directories to the directory from which the job is submitted.

`./hello` says to run the `hello` executable.

- c. Save the file and exit vi (as you did in steps 2d and 2e).
5. Submit a job to the scheduler:

```
$ qsub hello.qsub
19098.as0.al-salam.loc
$
```


6. This will submit a job and output its job ID, 19098 in this example. Your job will now be waiting in the queue, running, or finished. You can monitor it at any time by entering `qstat 19098` (or whatever your Job ID is) in the shell.

You may see something like the following:

```
qstat 19098
qstat: Unknown Job Id 19098.as0.al-salam.loc
```

This means the job is complete.

If the job were instead still running, you would see something like the following table:

Job id	Name	User	Time Use	S	Queue
19098.as0	STDIN	amweeden06	0	R	ec

In this output, the S column is the status column. The letter under this column tells you the status of the job; Q means it is waiting in the queue and R means it is running.

7. Once the job is complete, show the contents of `hello.out` with the `cat` command:

```
$ cat hello.out
Hello, World!
$
```

8. `hello.err` should be empty if there were no errors in running the program. Show the contents of `hello.err` with the `cat` command:

```
$ cat hello.err
$
```

If this command returns just a prompt (`$`), then the file is empty and there were no errors. Otherwise, the errors will be listed.

Part II. Outfit the program with MPI

9. We will now make a parallel version of the code using MPI. First we will tell the program to include the MPI library. We also tell the program that we are using MPI by putting `MPI_Init` at the top of `main` and `MPI_Finalize` at the bottom.

- a. Open `hello.c` and add the lines to the code as below:

```
#include <mpi.h>
#include <stdio.h>
```

```
int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);

    printf("Hello, World!\n");

    MPI_Finalize();

    return 0;
}
```

b. Save and quit the file.

10. Compile the code with GNU's MPI compiler:

```
$ mpicc -o hello hello.c
$
```

If any errors are listed, make sure there are no typos in `hello.c` (go back through step 9).

11. Edit the PBS script to use the MPI run command:

a. Open the `hello.qsub` file in `vi`:

```
$ vi hello.qsub
```

b. Enter insert mode and change the last line to use `mpirun` as below:

```
#PBS -q ec
#PBS -o hello.out
#PBS -e hello.err

cd $PBS_O_WORKDIR

mpirun -np 2 ./hello
```

Here `-np 2` tells MPI to use 2 processes. Both processes will run the `hello` executable.

c. Save and quit the file.

12. Submit a job to the scheduler:

```
$ qsub hello.qsub
19099.as0.al-salam.loc
$
```

13. Monitor the job with `qstat`. Once it finishes, view the contents of standard out and standard error:

```
$ cat hello.out
```

```
Hello, World!  
Hello, World!  
$ cat hello.err  
$
```

What do you notice about hello.out this time?

14. Let's have the processes print some useful information. We will have them print their rank, the total number of processes, and the name of the processor on which they are running.

a. Open hello.c in vi:

```
$ vi hello.c
```

b. Add the following lines to hello.c:

```
#include <mpi.h>  
#include <stdio.h>  
  
int main(int argc, char** argv)  
{  
    int rank = 0;  
    int size = 0;  
    int len = 0;  
    char name[MPI_MAX_PROCESSOR_NAME];  
  
    MPI_Init(&argc, &argv);  
  
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
    MPI_Comm_size(MPI_COMM_WORLD, &size);  
    MPI_Get_processor_name(name, &len);  
  
    printf("Hello, World from rank %d of %d on %s\n",  
          rank, size, name);  
  
    MPI_Finalize();  
  
    return 0;  
}
```

rank will be the rank of the process, size the total number of processes, and name the name of the processor on which the process is running.

c. Save and quit the file.

15. Compile the code with GNU's MPI compiler:

```
$ mpicc -o hello hello.c  
$
```

16. We don't need to change the PBS script because we will be using the same `mpirun -np 2 ./hello` command to execute the program. We will expect to see something different in `hello.out`, however. Let's submit the job and see what we get:

```
$ qsub hello.qsub
19100.as0.al-salam.loc
$
```

17. Monitor the job with `qstat`, and once it is finished check the contents of `hello.out` and `hello.err`:

```
$ cat hello.out
Hello, World from rank 0 of 2 on as1.al-salam.loc
Hello, World from rank 1 of 2 on as1.al-salam.loc
$ cat hello.err
$
```

On which processor did Rank 0 run for you? How about Rank 1?

18. Let's try running across multiple nodes instead of just one node (`as1.al-salam.loc` in the example above). Edit the `hello.qsub` file to include the following lines:

```
#PBS -q ec
#PBS -o hello.out
#PBS -e hello.err
#PBS -l nodes=2:ppn=1
```

```
cd $PBS_O_WORKDIR
```

```
mpirun -np 2 ./hello
```

The line that we added, `#PBS -l nodes=2:ppn=1`, says to run the job on 2 nodes with 1 process per node.

19. Submit a job:

```
$ qsub hello.qsub
19104.as0.al-salam.loc
$
```

20. Monitor the job with `qstat` until it finishes, then output the contents of `hello.out` and `hello.err`:

```
$ cat hello.out
Hello, World from rank 0 of 2 on as2.al-salam.loc
Hello, World from rank 1 of 2 on as1.al-salam.loc
```

```
$ cat hello.err
$
```

Now on which processor did Rank 0 run for you? Rank 1?

Part III. Outfit the program with OpenMP

1. We first need to tell the program to include the OpenMP library. Open `hello.c` in `vi` and add a line to the top:

```
#include <omp.h>
#include <mpi.h>
#include <stdio.h>
```

2. Compile the code with OpenMP support through the GNU compiler by using the `-fopenmp` option:

```
$ mpicc -fopenmp -o hello hello.c
$
```

3. OpenMP does not require any special run command or arguments. We may wish to tell the program how many OpenMP threads over which to parallelize, however. Open `hello.qsub` and add a line before the `mpirun` command:

```
cd $PBS_O_WORKDIR

export OMP_NUM_THREADS=2

mpirun -np 2 ./hello
```

This line tells the program to spawn 2 OpenMP threads per process when it executes an OpenMP parallel section.

4. Submit a job with `qsub`, monitor it with `qstat` until it finishes, and then view the contents of `hello.out` and `hello.err`:

```
$ qsub hello.qsub
19105.as0.al-salam.loc
$ qstat 19105
qstat: Unknown Job Id 19105.as0.al-salam.loc
$ cat hello.out
Hello, World from rank 0 of 2 on as2.al-salam.loc
Hello, World from rank 1 of 2 on as1.al-salam.loc
$ cat hello.err
$
```

What do you notice about the output? You might expect to see 4 “Hello, World”s because the program is supposed to spawn 2 OpenMP threads per process. However, OpenMP will not spawn any threads unless it is explicitly told to do so by marking a parallel section with an OpenMP **pragma**, hence we still only get 2 “Hello, World”s.

5. Let’s mark the `printf` as part of a parallel section so each thread will print the rank of the process, the total number of processes, the thread number, the total number of threads, and the processor on which it is running. Open `hello.c` in `vi` and make the following change to the `printf`:

```
#pragma omp parallel
{
    printf("Hello, World from rank %d of %d, thread %d
of %d on %s\n", rank, size, omp_get_thread_num(),
omp_get_num_threads(), name);
}
```

Note that we have now surrounded the `printf` by `#pragma omp parallel` followed by curly braces. This indicates that the `printf` is part of a parallel section that will be executed by multiple OpenMP threads.

Note also that we added the functions `omp_get_thread_num()` and `omp_get_num_threads()`. These will return the thread number of the thread and the total number of threads, respectively.

6. Compile the code and submit a job. Monitor it with `qstat` until it finishes, then view the contents of `hello.out` and `hello.err`:

```
$ mpicc -fopenmp -o hello hello.c
$ qsub hello.qsub
19106.as0.al-salam.loc
$ qstat 19106
qstat: Unknown Job Id 19106.as0.al-salam.loc
$ cat hello.out
Hello, World from rank 0 of 2, thread 1 of 2 on
as2.al-salam.loc
Hello, World from rank 0 of 2, thread 0 of 2 on
as2.al-salam.loc
Hello, World from rank 1 of 2, thread 0 of 2 on
as1.al-salam.loc
Hello, World from rank 1 of 2, thread 1 of 2 on
as1.al-salam.loc
$ cat hello.err
$
```

What do you notice about the output now?

This completes the exercise.

Exercise 4 – Scaling the Model on a Cluster

Part I – strong scaling

This exercise will take you through filling the table below, which will indicate how many seconds it takes to execute each program for the given numbers of nodes and cores per node used.

The program is being scaled through strong scaling, so the number of people stays constant at 10,000.

Strong Scaling, 10,000 People

# of nodes used	Total # of cores	Total # of People	Serial	OpenMP	MPI	Hybrid
1	4	10,000				
2	8	10,000				
3	12	10,000				
4	16	10,000				
5	20	10,000				
6	24	10,000				
7	28	10,000				
8	32	10,000				
9	36	10,000				
10	40	10,000				
11	44	10,000				
12	48	10,000				

1. Copy the `pandemic` directory to your account on `al-salam` using Secure Shell

Copy (`scp`):

```
$ scp -r pandemic yourusername@cluster.earlham.edu:
```

2. Log into `al-salam` using a secure shell (`ssh`):

```
$ ssh yourusername@cluster.earlham.edu
```

```
$ ssh as0
```

3. Change directories into the `pandemic` code directory:

```
$ cd pandemic
$
```

4. Edit the Makefile in **vi** and make sure this line is commented out:

```
#CFLAGS+==DX_DISPLAY -L/usr/X11R6/lib -lX11 -lm # Uncomment
to show X display
```

5. Compile all versions of the code:

```
$ make all
make clean
make[1]: Entering directory
`/nfs/cluster/home/amweeden06/pandemic'
rm -f pandemic.{serial,openmp,mpi,hybrid} *.o
make[1]: Leaving directory
`/nfs/cluster/home/amweeden06/pandemic'
make pandemic.{serial,openmp,mpi,hybrid}
make[1]: Entering directory
`/nfs/cluster/home/amweeden06/pandemic'
gcc -DSHOW_RESULTS pandemic.c -o pandemic.serial
gcc -DSHOW_RESULTS -fopenmp -DOPENMP pandemic.c -o
pandemic.openmp
mpicc -DSHOW_RESULTS -DMPI pandemic.c -o pandemic.mpi
mpicc -DSHOW_RESULTS -DMPI -fopenmp -DOPENMP
pandemic.c -o pandemic.hybrid
make[1]: Leaving directory
`/nfs/cluster/home/amweeden06/pandemic'
$
```

6. Open a new file called `pandemic.serial.qsub` and add the following lines:

```
#PBS -q ec
#PBS -o pandemic.serial.out
#PBS -e pandemic.serial.err
```

```
cd $PBS_O_WORKDIR
```

```
time ./pandemic.serial -n 10000
```

The `time` command will show us how much time it takes the program to run.

7. Submit a job to the scheduler using `pandemic.serial.qsub`:

```
$ qsub pandemic.serial.qsub
19545.as0.al-salam.loc
$
```


8. Monitor the job with `qstat` until the job ID is unknown (i.e. until the job finishes):

```
$ qstat 19545
qstat: Unknown Job Id 19545.as0.al-salam.loc
$
```

9. Check the contents of `pandemic.serial.err` to see how long it took to run the program:

```
$ cat pandemic.serial.err
```

```
real 0m0.765s
user 0m0.763s
sys  0m0.002s
$
```

This shows three times. The `real` time is the time spent executing the program from start to finish, including times during which the program is interrupted by other programs using the operating system or is blocking waiting for I/O. `User` time is the time spent by the program in user space, and `sys` time is the time spent by the program in the kernel. We are interested in how much time it took the program to finish once it was started, so we will consider `real` time in this exercise.

10. Enter your result for `real` time in the table under the “Serial” column in the row with 1 node used.

11. Open a new file called `pandemic.openmp.qsub` and enter the following text:

```
#PBS -q ec
#PBS -o pandemic.openmp.out
#PBS -e pandemic.openmp.err

export OMP_NUM_THREADS=8

cd $PBS_O_WORKDIR

time ./pandemic.openmp -n 10000
```

12. Submit a job to the scheduler, wait for it to finish, and show the results:

```
$ qsub pandemic.openmp.qsub
19550.as0.al-salam.loc
$ qstat 19550
qstat: Unknown Job Id 19550.as0.al-salam.loc
$ cat pandemic.openmp.err
```

```
real 0m1.106s
user 0m1.790s
sys 0m6.241s
$
```

13. Enter your result for `real` time in the table under the “OpenMP” column in the row with 1 node used.

14. Open a new file called `pandemic.mpi.qsub` and enter the following text:

```
#PBS -q ec
#PBS -o pandemic.mpi.out
#PBS -e pandemic.mpi.err
#PBS -l nodes=1:ppn=4

cd $PBS_O_WORKDIR

time mpirun -np 4 -machinefile $PBS_NODEFILE \
./pandemic.mpi -n 10000
```

15. Submit a job to the scheduler with `pandemic.mpi.qsub`, output the result (`pandemic.mpi.err`), and add the `real` time to the table under the “MPI” column for 1 node used.

16. Change the following bold sections of `pandemic.mpi.qsub`:

```
#PBS -q ec
#PBS -o pandemic.mpi.out
#PBS -e pandemic.mpi.err
#PBS -l nodes=2:ppn=4

cd $PBS_O_WORKDIR

time mpirun -np 8 -machinefile $PBS_NODEFILE \
./pandemic.mpi -n 10000
```

17. Repeat step 15 but enter the result in the row with 2 nodes used.

18. Repeat steps 16 and 17, but change `nodes=2` to `nodes=3` and `-np 8` to `-np 12`. Enter the result in the row with 3 nodes used.

19. Continue to fill out the table under the MPI column. Whenever you change `nodes=X` and `-np Y`, make sure that $Y = X * 4$. For example, when you run 8 nodes, make sure you specify 32 MPI processes with `-np`.

20. Open a new file called `pandemic.hybrid.qsub` and enter the following text:

```

#PBS -q ec
#PBS -o pandemic.hybrid.out
#PBS -e pandemic.hybrid.err
#PBS -l nodes=1:ppn=4

export OMP_NUM_THREADS=2

cd $PBS_O_WORKDIR

time mpirun -np 4 -machinefile $PBS_NODEFILE \
./pandemic.hybrid -n 10000

```

21. Scale the hybrid jobs just as you did the MPI jobs in steps 21 – 28.

When you finish this step, the table should be complete (except for the serial and OpenMP columns for more than 1 node – pure serial and pure OpenMP programs have no meaningful result if they are run using distributed memory, so we leave these cells blank).

Part II – weak scaling

In weak scaling, the problem size varies as the number of cores increases. Thus, instead of using a constant 10,000 people, we increase the number of people as we increase the number of cores. You can choose the factor by which we increase the number of people, but we will use 1,000 people per core as the example in this case.

Weak Scaling, 1,000 People per Core

# of nodes used	Total # of cores	Total # of people	Serial	OpenMP	MPI	Hybrid
1	4	4,000				
2	8	8,000				
3	12	12,000				
4	16	16,000				
5	20	20,000				
6	24	24,000				
7	28	28,000				
8	32	32,000				
9	36	36,000				
10	40	40,000				
11	44	44,000				
12	48	48,000				

1. Open `pandemic.serial.qsub` and make sure it looks like this to start:

```
#PBS -q ec
#PBS -o pandemic.serial.out
#PBS -e pandemic.serial.err
```

```
cd $PBS_O_WORKDIR
```

```
time ./pandemic.serial -n 4000
```

2. Fill in the first rows of the table under the Serial column as you did in Part I.

3. Fill out the table for OpenMP, MPI, and Hybrid, but make sure that for every job you submit the value for `-n` is equal to the value of `-np` times 1,000. The PBS files should look like the following to start:

a. **pandemic.openmp.qsub:**

```
#PBS -q ec
#PBS -o pandemic.openmp.out
#PBS -e pandemic.openmp.err
```

```
export OMP_NUM_THREADS=8
```

```
cd $PBS_O_WORKDIR
```

```
time ./pandemic.openmp -n 4000
```

b. **pandemic.mpi.qsub:**

```
#PBS -q ec
#PBS -o pandemic.mpi.out
#PBS -e pandemic.mpi.err
#PBS -l nodes=1:ppn=4
```

```
cd $PBS_O_WORKDIR
```

```
time mpirun -np 4 -machinefile $PBS_NODEFILE \
./pandemic.mpi -n 4000
```

c. **pandemic.hybrid.qsub:**

```
#PBS -q ec
#PBS -o pandemic.hybrid.out
#PBS -e pandemic.hybrid.err
#PBS -l nodes=1:ppn=4
```

```

export OMP_NUM_THREADS=2

cd $PBS_O_WORKDIR

time mpirun -np 4 -machinefile $PBS_NODEFILE \
./pandemic.hybrid -n 4000

```

Scaling Discussion

Exercise 4 is likely to produce tables that look something like the following:

Strong Scaling, 10,000 People

# of nodes used	Total # of cores	Total # of People	Serial	OpenMP	MPI	Hybrid
1	4	10,000	0m0.929s	0m1.120s	0m1.333s	0m1.418s
2	8	10,000			0m1.544s	0m1.560s
3	12	10,000			0m1.631s	0m1.607s
4	16	10,000			0m1.607s	0m1.618s
5	20	10,000			0m1.711s	0m1.711s
6	24	10,000			0m1.718s	0m1.741s
7	28	10,000			0m1.698s	0m1.784s
8	32	10,000			0m1.764s	0m1.737s
9	36	10,000			0m1.837s	0m1.853s
10	40	10,000			0m1.851s	0m1.862s
11	44	10,000			0m1.909s	0m1.876s
12	48	10,000			0m1.868s	0m1.899s

This table shows Amdahl's Law in action. Notice that the program is not speeding up beyond 4 cores; in fact, it is slowing down. This shows that in terms of speedup there is no advantage to scaling the problem of 10,000 people beyond 4 cores. The slow-down may be incurred by the communication overhead that occurs as we add more nodes.

Weak Scaling, 1,000 People per Core

# of nodes used	Total # of cores	Total # of people	Serial	OpenMP	MPI	Hybrid
1	4	4,000	0m0.170s	0m0.484s	0m1.204s	0m1.241s
2	8	8,000			0m1.447s	0m1.522s
3	12	12,000			0m1.665s	0m1.685s
4	16	16,000			0m1.752s	0m1.732s

5	20	20,000			0m1.887s	0m1.817s
6	24	24,000			0m2.055s	0m1.971s
7	28	28,000			0m2.212s	0m2.182s
8	32	32,000			0m2.248s	0m2.058s
9	36	36,000			0m2.397s	0m2.204s
10	40	40,000			0m2.482s	0m2.403s
11	44	44,000			0m2.412s	0m2.568s
12	48	48,000			0m2.816s	0m2.610s

At first glance, it may appear that Amdahl's Law is in effect here as well; there is no speedup beyond 4 cores; in fact, there is a more pronounced slow-down. The difference is that in this scaling we are changing the total number of people as we change the core count. Thus, even though the problem takes longer to complete as we add more cores, it is not the same problem. We are seeing here the effect of Gustafson's Law. Even though we cannot speedup a single problem, we are able to model a larger problem in roughly the same amount of time. What the table above tells us is that if we are willing to spend about 2.6 seconds to model the problem, we can model a problem that has 48,000 people on this cluster. This is an improvement over the 4,000 person problem that we can model with 4 cores.

This concludes the exercises; below are student project ideas and a post-assessment rubric to gauge student learning of the module.

Student Project Ideas

1. Explore adding a limited immunity to the model – after a certain length of time, people who are immune become susceptible again. What changes need to be made to the algorithm to support this? If you are comfortable with the C language, what changes need to be made to the code? Try making these changes – what issues arose in the process of modifying the algorithm and code?
2. The scaling exercise explored wall times on the level of a few seconds, but scientific code can often takes minutes, hours, or even days to run. Try scaling the code with much larger problem sizes. What sort of effects of communication overhead and Amdahl's/Gustafson's Laws do you observe when the code takes longer to run? Do you observe any strange behavior at certain core counts?

Post-assessment Rubric

This rubric is to gauge students' knowledge and experience after using the materials and completing the exercises 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
Serial (non-parallel) Algorithm Design
Parallel Algorithm Design
Parallel Hardware
MPI programming
OpenMP programming
Using a cluster
Scaling parallel code

In addition, students are asked the following questions:

1. What did you find to be the most useful aspect of this module?
2. What did you find to be the least useful aspect?
3. What did you learn about that you would be excited to learn more about?
4. What did you find particularly unuseful or irrelevant?

References

Shiflet, Angela B. and George W. Shiflet. "Getting the "Edge" on the Next Flu Pandemic: We Should'a "Node" Better". Undergraduate Petascale Education Program module.
<http://shodor.org/petascale/materials/UPModules/socialNetworks/>.
Wofford College, Spartanburg, South Carolina.

Wilensky, U. (1998). NetLogo Virus model.
<http://ccl.northwestern.edu/netlogo/models/Virus>. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.