San Sebastián / Donostia

Simulación de propagación del SARS-CoV-2

Sistema de computo paralelo

Iñaki García
Asier Zubia

Universidad del País Vasco
Facultad de Ingeniería Informática
Ingeniería del Software

23/05/2021

Índice

1. Introducción

En este documento, se redacta la memoria del proyecto final de Sistemas de Computo Paralelo. Este proyecto, consta de dos partes, siendo esta la segunda. El objetivo de esta parte era realizar la propagación de un virus de manera paralela, con el uso de MPI. La implementación del proyecto se encuentra en un repositorio de GitHub : https://github.com/Asierzubia/COVID/tree/MPI.

En primer lugar, se redacta la implementación realizada en el proyecto. Para ello, se comienza de menos a más, explicando en primer lugar las estructuras y métodos más pequeños, y se terminan explicando los métodos más importantes y que más carga tienen en el proyecto.

Para finalizar, se comentan las dificultades que se han tenido en el proyecto y también como se tiene que ejecutar el programa. Se recomienda leer toda la memoria, ya que el programa requiere de ciertas condiciones para que funcione, ya que se han definido como requisitos del proyecto.

1.1 Diseño

El programa se encuentra dividido en diferentes ficheros ".c", para que sea un programa más comprensible.

- ***main.c***: fichero principal que tiene los métodos principales para realizar las acciones de las personas.
- ***person.c***: métodos para la creación de una persona.
- ***prob.c***: métodos que calculan las probabilidades necesarias durante la simulación, haciendo uso de la librería *gsl*.
- ***lists.c***: métodos de inicialización, liberación e imprimir de las listas y matrices que se utilizan en el programa.
- ***metrics.c***: posiciones y métricas de la simulación.
- ***conf.c***: configuración de la simulación mediante fichero o parametrización.

Muchos de los métodos no se han podio introducir dentro de un fichero ".c" ya que las estructuras de MPI y listas estáticas se tienen que declarar en el *main.c*. Todos estos ficheros disponen de una cabecera llamada "*definitions.h*". No se ha asignado un cabecera por cada fichero ya que sería necesario cambiar muchas de las variables, por lo que alargaría el tiempo de vida del proyecto.

2. Estructuras

El programa dispone de un gran número de listas, que muchas de ellas están relacionadas entre sí. Para que se entienda mejor esta relación, se ha creado el siguiente Diagrama que resume las estructuras del programa. Es un diagrama incompleto, por lo que no se muestran todas las relaciones, ya que permite su mejor compresión.

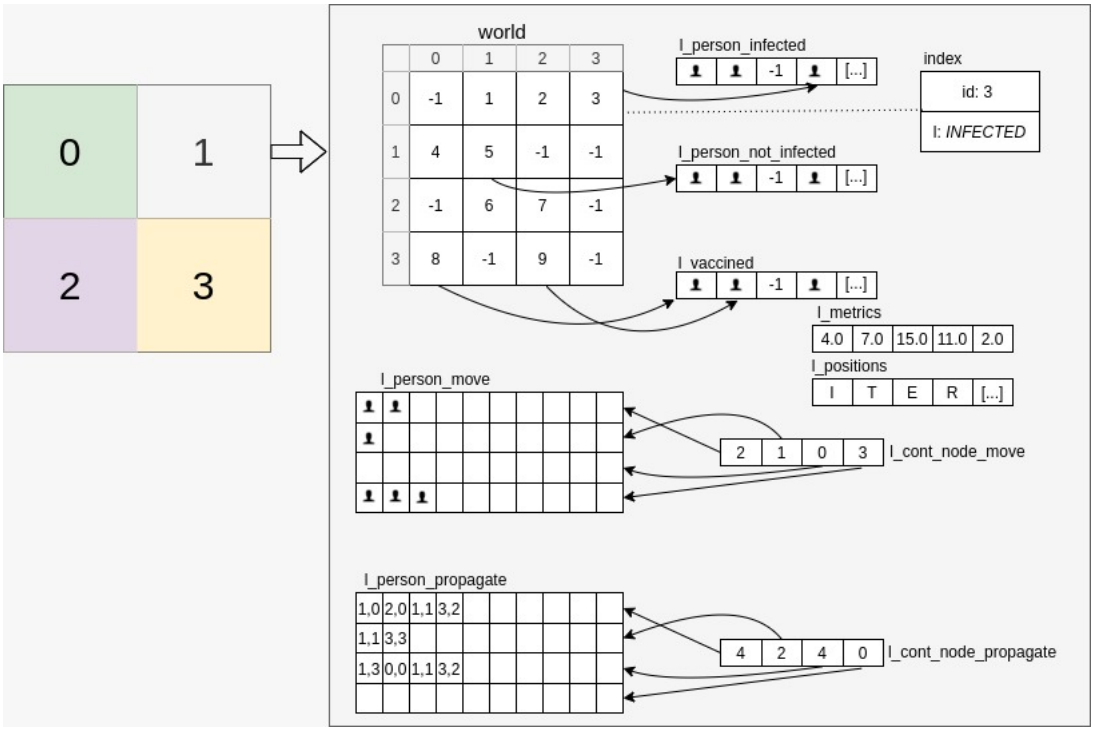

Figure 2.1: Diagrama de estructuras del programa

2.1 Persona

Una persona, que forma una población, se clasifica en 3 listas diferentes (se profundiza en el apartado 2.3):

- `person_t *l_person_infected`.
- `person_t *l_person_notinfected`.
- `person_t *l_vaccined`.

Las personas, disponen de las siguientes características:

```
typedef struct person
{
    int age;
    int state;
    float prob_infection;
    coord_t coord;
    int speed[2]; // 0 direction 1 speed
    int incubation_period;
    int recovery;
    int id;
    int id_global;
} person_t;
```

Entre todos estos atributos, se destacan los siguientes:

- `int id`: identificador de la persona, el cual es el mismo que el índice de la lista en la que se encuentra. Es decir, si se crea la primera persona infectada (`l_person_infected`), esta tendrá el identificador "0" y se introducirá en la primera posición de la lista. Para controlar esto, cada lista dispone de un contador que se incrementa/decrementa cuando la lista dispone de cambios.
  - `int id_ContI`: contador de la lista de infectados.
  - `int id_ContNotI`: contador de la lista de no infectados.
  - `int id_vaccined`: contador de personas vacunadas o inmunizadas
- `int id_global`: para poder realizar un seguimiento más exhaustivo, se dispone de un identificador global, para poder distinguir las personas entre los nodos. Es decir, una persona tiene un identificador "local" para el nodo en el que se encuentra (`int id`), que se relaciona con el escenario y con la lista en el que se encuentra, y por otra parte tiene un identificador que lo distingue entre todas las personas de la población.
- `float prob_infection`: cuando una persona quiere infectar a otra, la persona atacada puede resistir el contagio del virus. Para ello, dispone de una probabilidad que se calcula cuando es atacada con el método `void change_infection_prob(person_t *person)`.
- `coord_t coord`: estructura que dispone de las coordenadas "x" e "y". Estas coordenadas, son las coordenadas de la matriz del escenario (Véase el apartado 2.2). No se ha utilizado una lista para poder enviar esta estructura al resto de nodos, en lugar de enviar una lista de dos elementos. .
- `int speed`: lista de dos elementos que contiene : [0] dirección y [1] velocidad. Las direcciones en el programa, siguen la lógica de ir "en contra de las agujas del reloj". Las direcciones se comprenden entre los valores 1 y 8, como se pueden ver en la Figura 3.2. Por otro lado, la velocidad solo puede ser entre 0 y 2, por lo que solo se podrá desplazar 2 casillas como máximo (Véase más información sobre el movimiento en el apartado 3.2).

2.2 Escenario

El escenario se ha creado haciendo uso de una matriz 2D llamada `index_t **world`. Esta matriz, dispone de estructuras con el nombre de **índices** (`index_t index`). Esta estructura se compone de :

```
typedef struct index
{
    int id; // index
    enum list l;
} index_t;
```

Su función es relacionar a las **Personas** con el **Escenario**. Para ello, tiene un identificador (`int id`), que es el identificador de la persona. Como se ha comentado en el apartado anterior, este identificador de persona, a su vez es el identificador de la lista en la cual se encuentra, pero no se tiene que confundir con el identificador global (`int id_global`). Para conocer la lista en el que se encuentra, tiene otra variable de tipo enumerado, que contiene los 3 posibles valores de las listas:

- *INFECTED*,
- *NOT_INFECTED*,
- *VACCINED*

2.2.1 ¿Qué beneficios aporta esta estructura?

- **Propagar:** a la hora de propagar o infectar, no es necesario recorrer todas las personas implicadas. En el caso utilizar este mecanismo, se tendría que recorrer la lista de personas no infectadas (`person_t *l_person_notinfected`), por lo que se podría recorrer hasta un máximo de la población global (peor caso posible). Sin embargo, si no tuviese personas la lista, no se realizaría la propagación, por lo que sería muy eficiente.

  No obstante, se ha implementado de manera que se busque el radio en la matriz de la población, y el acceso a las personas es de acceso directo (con la ayuda del identificador). Por lo que en el peor caso, con radio 2, se recorren 12 posiciones (diagonales también se comprueban).

- **Mover:** para mover a las personas, se recorren las 3 listas y se miran las coordenadas de la persona (`coord_t coord`). Estas coordenadas, como están en concordancia con el escenario, solo es necesario revisar las 12 posiciones posibles (por restricción, solo se puede mover 2 posiciones como máximo). En el caso de no tener el índice en el escenario, sería necesario por cada persona recorrer las 3 listas buscando las posiciones implicadas (también se podrían tener las listas ordenadas, pero dificultaría la asignación del identificador y no sería eficiente).

2.2.2 Cuadrantes

Cada nodo, tendrá que disponer de un cuadrante fijo. Estos cuadrantes (denominados `quadrant_x` y `quadrant_y`) se calculan en un inicio de la siguiente manera:

```
quadrant_x = SIZE_WORLD / (int)floor(sqrt(world_size));
quadrant_y = SIZE_WORLD / (int)ceil(sqrt(world_size));
```

El cuadrante en el **eje x** se calcula dividiendo el número de nodos entre el valor inferior de la raíz cuadrada del tamaño del mundo (**siempre son cuadrados**). Por su parte, el cuadrante de **eje y** se divide entre el valor superior. Para obtener estos valores, se hace uso de las funciones `ceil (x)` y `floor (x)`, que se han obtenido gracias a la ayuda del siguiente enlace. Estos cuadrantes, se envían a todos los nodos con la ayuda de (`MPI_Bcast`).

Por restricción, el cuadrante tiene que ser como mínimo de 5x5, para que los movimientos de mover y propagar siempre se puedan realizar, aunque el programa funciona si esta restricción no se cumple.

2.3 Población

Una vez definida el número de personas que conforman la población, se tiene que calcular cuantas personas pertenecen a cada nodo. Para ello. se realiza una operación sencilla:

```
population = round(POPULATION_SIZE / world_size);
```

Se dividen el número de nodos de la simulación, entre el tamaño del mundo. No se ha implementado para asignar las personas sobrantes a los nodos, por lo que no se realiza ninguna operación con esas personas. Cuando se ha realizado el calculo, se envía esta variable a todos los nodos con `MPI_Bcast`. Con esta variable, cada nodo inicializa cada lista de personas a este valor multiplicado por el número de nodos, teniendo en cuenta el peor caso. El peor caso se da cuando todas las personas posibles han estado en el mismo nodo con el mismo estado. Las listas son las siguientes:

- ***l_person_infected:*** lista de personas que están infectadas (estados 1 y 2).
- ***l_person_notinfected:*** lista de personas que no están infectadas, pero tampoco vacunadas (estado 0).
- ***l_vaccined:*** lista de personas vacunadas e inmunizadas (estado 3 y 4). Se ha decidido añadir en esta lista las personas recuperadas, ya que se tratan como inmunes por lo que no se pueden volver a infectar. De esta manera, no aparecen cuando se recorre la lista de personas no infectadas.

En el caso de que una persona **cambie de estado**, cambiaría su identificador a "-1" en la lista, para que a la hora de recorrer la lista no tener en cuenta esa posición. Esto, podría hacer que las listas se llenen, por lo que se ha creado un método (`void realocate_lists()`). El objetivo de este método es desplazar todas las personas a la izquierda de la lista, cambiando sus identificadores locales en la lista y en el escenario. También, disminuye los contadores de cada lista.

2.3.1 Edad Media

Para obtener diferentes simulaciones, las edades de las personas de la población se calculan en base a una edad media que se considera como uno de los parámetros de entrada.

Se ha hecho uso de la distribución Beta para sacar dichas edades. Dicha distribución depende de dos parámetros *Beta* y *Alpha* que caracterizan la distribución y hay que calcular a partir de los parámetros de entrada. El otro parámetro que se necesita calcular para definir completamente *Alpha* y *Beta* es la varianza. Como tampoco se dispone de la varianza, se toma el valor de la varianza del ejemplo que se dispone en *eGela*. La fórmula para hallar la **varianza normalizada** es:

$$V(VarianzaNormalizada) = (\frac{DesviaciónEstándar}{EdadMáxima})^2$$

El otro parámetros que se necesita es la **edad media normalizada**:

$$E(EdadMediaNormalizada) = \frac{EdadMedia}{EdadMáxima}$$

Con estos dos parámetros, varianza normalizada y edad media normalizada se calculan los valores de *Alpha* y *Beta* como:

$$Alpha = \frac{(1-E)E^2 - EV}{V}$$

$$Beta = \frac{[(1-E)E^2 - EV](1-E)}{EV}$$

Estas expresiones se han sido deducido por los autores a raiz de la página de *Wikipedia* sobre distribuciones Beta facilitada por el profesor. Como se ha comentado anteriormente, *Alpha* y *Beta* caracterizan completamente la distribución *Beta*.

3. Acciones principales

3.1 Vacunación

Al principio del programa, se calculan cuantas vacunas le pertecen a cada nodo. Para ello, se utiliza la siguiente formula:

$$VacunasPorNodo = \frac{\frac{Iteraciones}{Poblacion * PorcentajeInmunizada}}{NumeroNodos}$$

El porcentaje a inmunizar, es una constante que indica el porcentaje de la población que se quiere vacunar, y se le ha asignado un valor de **70%** (mismo valor que el enunciado).

Para vacunar a una persona, primero se comprueba que esa persona pertenezca al grupo de vacunación. Para no complicar el ejercicio, los grupos de vacunación cambian al transcurrir el **15%** de las iteraciones totales. En el caso de que la persona a vacunar sea de ese grupo, se vacuna, realizando las operaciones de eliminar y añadir de las listas. El método devuelve **0** en el caso de que no se haya vacunado, y *1* en el caso contrario. Esto es necesario ya que las operaciones de vacunación, se realizan sobre las personas sanas. En el caso de que esa persona cambie de sano a vacunado (estado 0 $\longrightarrow$ 4), no se moverá en esa iteración, para que se mueva en la siguiente.

3.2 Mover

Como se ha comentado anteriormente, cuando se mueven las personas, se puede mirar hasta un máximo de 12 posiciones posibles, que se determinan con la combinación de la posición actual de `coord` y la velocidad. Estos valores cambian cada vez que se mueve una persona, con el método `change_move_prob(person_t *person)`. Cuando se obtienen los nuevos valores de la nueva posición, se pueden dar dos casos:

- **Mismo cuadrante:** las coordenadas pertenecen al mismo cuadrante, por lo que se comprueba si esa posición es diferente a "-1". Para saber si es del mismo cuadrante, las posiciones "x" e "y" tienen que ser inferiores al cuadrante de su eje, y mayores o iguales a "0".
- **Otro cuadrante:** en el caso de que la nueva posición sea de otro nodo adyacente, en primer lugar se calcula que nodo se trata. Para ello, se utiliza el método: `int search_node(int world_rank, int x, int y)`, que busca el nodo destino (Vease apartado 3.2.1). También, hace uso de otro método que comprueba que no se salga del escenario llamado `is_inside_world(int from, int direction, int to_node)`. En el caso de que se envíen visitantes (Véase apartado 4.3), se guarda la persona a enviar en la matriz `person_move_t **l_person_moved`. Esta matriz se inicializa con el número de nodos de la simulación, y por cada nodo, guarda las personas que se quieren enviar. Para saber en qué posición se tiene que guardar la persona, y saber cuantas personas se tienen que enviar a cada nodo, se dispone de una lista auxiliar `int *l_cont_node_move`, con los contadores de cada nodo.

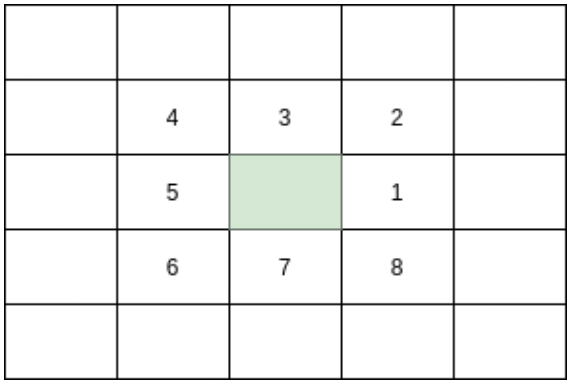

Figure 3.1: Direcciones posibles en el programa

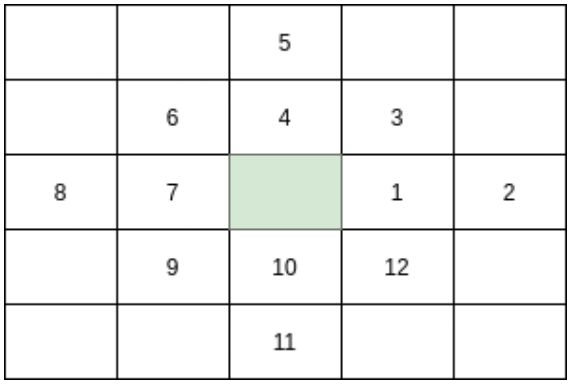

Figure 3.2: Movimientos posibles de las personas

3.2.1 `int search_node(int world_rank, int x, int y)`

Para buscar el nodo destino, se tienen que tener en cuenta todos los casos. En este caso, también se han tenido en cuenta las diagonales, por lo que hay 8 casos posibles. Se determina cada caso, teniendo en cuenta las coordenadas destino y sus valores en qué o cuales ejes se excede. Una vez averiguado la dirección, se calcula el nodo destino dependiendo de la dirección.

No se puede utilizar la dirección que tiene el vector de velocidad de la persona, ya que algunos movimientos no concuerdan con esa dirección (p. ej.: si se mueve en diagonal hacia arriba, pero solo ha subido al nodo de arriba).

3.2.2 `int is_inside_world(int from, int direction, int to_node)`

Para averiguar, si el nodo destino es un nodo fuera del escenario, se tienen en cuenta los siguientes casos:

- **Abajo:** en el caso de que se supere el margen de abajo, se comprueba si es un nodo inferior al número de nodos por fila.
- **Derecha:** en el caso de que se supere el margen de la derecha, se comprueba que no sea el último nodo posible por fila. En el caso de que sea de una fila superior a la primera, se "convierte" ese nodo al nodo correspondiente a la primera fila, realizando su módulo.

- **Izquierda:** en el caso de que se supere el margen de la izquierda, se comprueba que no sea el primer nodo por fila. En el caso de que sea de una fila superior a la primera, se "convierte" ese nodo al nodo correspondiente a la primera fila, realizando su módulo.
- **Arriba:** en el caso de que se supere el margen de arriba, se comprueba que el nodo destino no sea superior al número de nodos utilizado en la simulación.

También, es necesario convertir las coordenadas que se quiere ir con las coordenadas del nodo destino. Para ello, es necesario tener en cuenta por qué limite se sobrepasa y sumar/restar la diferencia. Esta operación la realiza el método `calculate_coord(x,y)`.

Cuando se conoce el nodo destino, se añade la persona con las nuevas coordenadas con una nueva estructura a `person_move_t **l_person_moved;`, y esta matriz tiene una lista conjunta llamada `l_cont_node_move`. El objetivo de estas dos listas es controlar por cada nodo, que persona se tiene que enviar. En la lista `int *l_cont_node_move`, se almacenan las personas que se quieran enviar a otro nodo. Por otra parte, en `l_cont_node_move` es una lista que se inicializa con el número de nodos de la simulación, y se guarda el contador de personas por nodo, para poder acceder a la última posición de cada nodo de la matriz. Después, el método `send_visitors()` encarga de enviar la lista de personas a cada nodo correspondiente.

3.3 Propagación

Gracias al diseño e implementación realizado, se pueden utilizar métodos ya definidos para otras funcionalidades para realizar la propagación más sencilla. La propagación, sigue un esquema similar al movimiento de las personas, con la diferencia de que propagate tiene que comprobar todas las posiciones disponibles dentro de su radio. Por consiguiente, se iteran todas las posiciones posibles en el escenario (`world`). Por cada posición, al igual que el movimiento de las personas, se comprueban las dos posibilidades:

- **Mismo cuadrante:** en el caso de que sea del mismo cuadrante, se comprueba que la posición del escenario no tenga un identificador "-1", y que su estado sea *NOT-INFECTED*. En el caso de que cumpla con las condiciones, se decide si la persona muere (característica especial del programa), o si se infecta. En cada caso, se realizarán las modificaciones en las listas pertinentes.
- **Otro cuadrante:** se realizan las mismas operaciones que se realizan en el movimiento de las personas. En este caso, si se envían visitantes (Véase apartado 4.3), se guarda la coordenada a enviar en la matriz `coord_t **l_person_propagate`. No es necesario guardar la persona, ya que no dispone de información extra a la hora de propagar. La probabilidad de que se infecte la persona destino depende de su propia probabilidad de infección. Esta matriz se inicializa con el número de nodos de la iteración, y por cada nodo, guarda las personas que se quieren envíar. Para saber en qué posición se tiene que guardar la persona, y saber cuantas personas se tienen que enviar a cada nodo, se dispone de una lista auxiliar `int *l_cont_node_propagate`, con los contadores de cada nodo.

3.3.1 Probabilidad de muerte

En base a los datos proporcionados en la documentación del proyecto sobre los datos de mortalidad de una población, en primer lugar se han normalizado los datos para que la suma de los datos nos de la unidad.

$$C * \sum_{i=1}^{n} P = 1$$

Donde C es la constante de normalización. En este caso haciendo las operaciones se obtiene que:

$$C = 1/\sum_{i=1}^{n} P = 3,460$$

Una vez obtenido el valor de C, renormalizamos todas los datos. Se observa efectivamente que sumando todos ellos se obtiene la unidad. Los nuevos valores en tanto por mil son:

- ≥ 0 y < 10 años: 6,9204
- ≥ 10 y < 20 años: 6,9204
- ≥ 20 y < 30 años: 6,92040
- ≥ 30 y < 40 años: 6,92042
- ≥ 40 y < 50 años: 13,8406
- ≥ 50 y < 60 años: 44,9827
- ≥ 60 y < 70 años: 124,5674
- ≥ 70 y < 80 años: 276,8166
- ≥ 80 años: 512,1107

Finalmente con la función `int random_number(int min_num, int max_num)` entre 1 y 1000, si el número que devuelve es igual o inferior a los valores normalizados de la probabilidad, se considera que el individuo ha fallecido. En caso contrario, se supondrá que sobrevive. Por consiguiente, una persona solo puede morir en el momento que es infectada, lo que supone una nueva restricción del programa.

4. Métricas y Posiciones

4.1 Posiciones

Para imprimir las **posiciones** de las personas, se ha optado por la vía eficiente y fácil. Para no sobrecargar la red, cada nodo guarda las posiciones calculadas en cada *BATCH* dentro de una lista de *char*: `char *l_positions` De esta manera, cuando termine la simulación del programa, cada nodo envía la listas al nodo 0, con la ayuda de `MPI_Gather`. De esta manera, el nodo 0 unificará todos los textos enviados por cada nodo, y lo imprimirá en un fichero llamado : *POSITIONS.pos*. A continuación, se presenta un ejemplo del fichero en un simulación con 4 procesadores, 20 personas, 30 iteraciones, *BATCH* de 5 y un escenario de 8x8.

```
P0  ITERACCION: 4  {INFECTED -  0[0,0] 2[0,2] 3[0,3]} {NOT-INFECTED -  1[3,1] 4[1,0]} {VACCINED - }
P0  ITERACCION: 9  {INFECTED -  0[0,0] 2[0,2] 3[0,3]} {NOT-INFECTED -  1[3,0]} {VACCINED -  4[1,0]}
P0  ITERACCION: 14  {INFECTED - } {NOT-INFECTED - } {VACCINED -  4[1,0] 0[0,0] 1[3,0] 2[0,1]}
P0  ITERACCION: 19  {INFECTED - } {NOT-INFECTED - } {VACCINED -  4[1,0] 0[2,0] 1[1,3] 2[2,1]}
P0  ITERACCION: 24  {INFECTED - } {NOT-INFECTED - } {VACCINED -  4[1,2] 0[2,0] 1[0,3]}
P0  ITERACCION: 29  {INFECTED - } {NOT-INFECTED - } {VACCINED -  0[2,0] 1[0,1]}
P1  ITERACCION: 4  {INFECTED -  5[2,0] 6[2,3] 7[3,3] 8[1,2] 9[3,0]} {NOT-INFECTED - } {VACCINED - }
P1  ITERACCION: 9  {INFECTED -  5[0,1] 6[1,2] 8[2,0] 9[1,1]} {NOT-INFECTED - } {VACCINED - }
P1  ITERACCION: 14  {INFECTED -  5[0,1] 6[0,3] 8[0,0] 9[1,1]} {NOT-INFECTED - } {VACCINED - }
P1  ITERACCION: 19  {INFECTED -  5[1,0] 6[3,3] 8[0,0] 9[0,3]} {NOT-INFECTED - } {VACCINED - }
P1  ITERACCION: 24  {INFECTED -  5[1,0] 6[3,2]} {NOT-INFECTED - } {VACCINED -  8[0,2]}
P1  ITERACCION: 29  {INFECTED -  5[0,1]} {NOT-INFECTED - } {VACCINED -  6[1,2]}
P2  ITERACCION: 4  {INFECTED -  11[2,0] 14[1,0]} {NOT-INFECTED -  12[1,2]} {VACCINED -  10[0,0] 13[0,3]}
P2  ITERACCION: 9  {INFECTED -  11[2,0] 14[0,1]} {NOT-INFECTED - } {VACCINED -  10[2,3] 13[3,0] 12[0,2]}
P2  ITERACCION: 14  {INFECTED -  3[0,3]} {NOT-INFECTED - } {VACCINED -  10[1,2] 13[3,0] 12[0,3] 11[0,0]}
P2  ITERACCION: 19  {INFECTED -  3[1,2]} {NOT-INFECTED - } {VACCINED -  10[0,1] 13[3,0] 12[0,2] 11[0,0]}
P2  ITERACCION: 24  {INFECTED - } {NOT-INFECTED - } {VACCINED -  10[0,1] 13[2,0] 12[1,3] 11[0,0] 3[0,2] 2[2,3]}
P2  ITERACCION: 29  {INFECTED - } {NOT-INFECTED - } {VACCINED -  10[0,1] 12[3,3] 11[0,0] 3[0,2] 2[1,2] 4[2,3]}
P3  ITERACCION: 4  {INFECTED -  15[0,0] 16[2,1] 18[0,2] 19[1,0]} {NOT-INFECTED -  17[1,3]} {VACCINED - }
P3  ITERACCION: 9  {INFECTED -  15[0,0] 16[2,1] 18[0,2] 19[1,0] 7[1,3]} {NOT-INFECTED - } {VACCINED -  17[3,3]}
P3  ITERACCION: 14  {INFECTED -  15[0,0] 16[3,2] 18[0,3] 19[1,2] 7[3,3] 14[2,1]} {NOT-INFECTED - } {VACCINED -  17[3,3]}
P3  ITERACCION: 19  {INFECTED -  15[2,0] 16[2,1] 18[0,3] 19[2,2] 7[1,3] 14[2,1]} {NOT-INFECTED - } {VACCINED -  17[3,1]}
P3  ITERACCION: 24  {INFECTED -  18[0,3] 19[2,2] 7[1,2] 14[2,1] 9[0,3]} {NOT-INFECTED - } {VACCINED -  17[1,3] [...]
P3  ITERACCION: 29  {INFECTED -  18[0,3] 7[2,2] 14[2,1] 9[2,3]} {NOT-INFECTED - } {VACCINED -  17[1,3] 14[3,1] [...]
```

El punto negativo que tiene esta implementación, es que cada nodo envía toda su información concatenada, por lo que en el fichero se imprimen en orden, por lo que no se puede separar la información por iteraciones. Esto tampoco supone un gran problema, ya que se especifica siempre la iteración y el nodo, por lo que la trazabilidad de la información se puede realizar de la misma manera, y se ha utilizado de manera activa en la fase de pruebas. también, se han utilizado comandos que proporciona Linux como *grep*, para buscar alguna

persona en específico en el fichero. Esto es posible, ya que la persona siempre mantiene su identificador global.

4.1.1 *Buffer*s

Se han tenido muchos problemas con los tamaños de los *buffer*, ya que no se querían definir tamaños muy grandes, para no desperdiciar memoria. Para ello, se han definido dos tipos de tamaño *buffer*:

- `int buffer_node`: tamaño de *buffer* que utiliza cada nodo.
- `int buffer_total`: tamaño de *buffer* que se utiliza para el fichero final, *METRICAS.metricas*.

Se definen siguiendo las siguientes fórmulas:

$$bufferNode = tamañoEscenario * tamañoEscenario * \frac{Iteraciones}{BATCH} * 10$$

$$bufferTotal = bufferNode * numeroNodos$$

Tras realizar muchas simulaciones del programa, se ha comprobado que cuando los tamaños de los *buffer* son muy elevados, el programa aborta y muestra un error de *Segmentation Fault*. Por consiguiente, se ha obtenido un valor estable de *buffer* de 7.000.000, por lo que en caso de que el valor del tamaño obtenido se exceda de ese valor, es sustituido por este.

A raíz de este problema, al finalizar el proyecto se ha obtenido como conclusión que no es la mejor implementación posible, pero es la que se ha realizado. Esto se debe a que los *buffer* se pueden llenar, y además, ralentiza en demasía las últimas iteraciones (solo apreciable con simulaciones muy ambiciosas). La mejor opción sería utilizar las funciones de MPI para escribir en ficheros de manera paralela, pero tras dedicar una gran cantidad de tiempo a esta tarea y no conseguir resultados satisfactorios, se decidió no utilizar esa implementación.

4.2 Métricas

Para recolectar las medias para las métricas, se ha definido una mecanismo diferente. Esto se debe a que para realizar el calculo de las medias, es necesario disponer de los valores del resto de los nodos, sino no se puede realizar la operación matemática. Para ello, se sigue enviando toda la información al nodo 0 al finalizar la simulación, pero en su lugar cada nodo envía 1 lista con 5 posiciones:

- `l_metrics[0]`: media de personas sanas. Son aquellas nuevas personas que se han recuperado, o que se han vacunado.
- `l_metrics[1]`: media de personas contagiadas. Son aquellas nuevas personas que han sido contagiadas por otra persona.
- `l_metrics[2]`: media de personas recuperadas. Son aquellas nuevas personas que se han recuperado de una infección.
- `l_metrics[3]`: media de personas fallecidas. Son aquellas nuevas personas que han fallecido.
- `l_metrics[4]`: media de R0. Son todas las personas que pueden ser contagiadas por otras.

Esta lista se inicializa de manera estática, ya que solo se tienen que enviar 5 métricas. Las métricas, se calculan con la ayuda de contadores, que aumentan cuando uno de los casos mencionados ocurre. Cuando finaliza un *BATCH*, se llama al método `void save_metrics(int world_rank, int iteration)`, que guarda los valores de los contadores en su respectiva posición, sumando el valor guardado del anterior *BATCH*. De esta manera, cuando se termina la simulación, en la lista se tienen guardados los valores de **contadores nuevos por cada *BATCH***, y se divide entre el número de *BATCH* que ha tenido la simulación. De esta manera, se obtiene el **valor medio del nodo**. Por ello, a no ser que se utilicen simulaciones ambiciosas, estos valores serán bajos.

Estos valores medios se envían al nodo 0, para que sume todos los valores medios de cada métrica. Para realizar esta tarea más sencilla, se concatenan las listas de posiciones fijas (5), una detrás de otra en el nodo 0 con la ayuda de `MPI_Gather`. Esta lista tiene como nombre `l_metrics`, y se inicializa multiplicando el número de nodos de la simulación por el valor 5 (número de métricas). De esta manera, se obtienen los valores de cada métrica saltando de 5 en 5 posiciones. Una vez sumado los valores medios de cada métrica, se imprime en el fichero *METRICAS.metricas*.

4.3 Enviar visitantes

Se denomina un "visitante" a todo acceso a un nodo diferente al actual. Un visitante puede ser una Persona + Coordenadas cuando se realiza los movimientos, o las coordenadas cuando se realiza la propagación. Para ello, se dispone del método `void send_visitors(int flag)`. Este método, diferencia los visitantes de propagación y movimiento con la ayuda de un *flag*:

- **0:** personas y coordenadas de movimiento.
- **1:** coordenadas de propagación.

Este método se encarga de enviar en un orden concreto estructuras *MPI*, que se han creado previamente. El orden en el que se envían los datos es fijo para evitar el bloqueo de las comunicaciones entre los diferentes nodos. Cada nodo tiene 8 direcciones posibles a las que mover o propagar una persona. Primeramente se envía la cantidad de datos que va a enviar en el siguiente envío, y después se envía una lista de datos con las estructuras *MPI* creadas. Si no hay datos a enviar entonces enviará un contador con el valor 0 indicando que no hay datos que enviar. El orden que siguen para el el envío y recibo es el siguiente:

- **Derecha:** El nodo envía los datos al nodo de su derecha. Recibirán todos aquellos nodos que tengan un nodo a su izquierda, el resto no.
- **Arriba Derecha:** El nodo envía los datos al nodo que esté arriba a la derecha. Recibirán todos aquellos nodos que tengan un nodo abajo a su izquierda, el resto no.
- **Arriba:** El nodo envía los datos al nodo que tenga arriba. Recibirán todos aquellos nodos que tengan un nodo abajo, el resto no.
- **Arriba Izquierda:** El nodo envía los datos al nodo que esté arriba a la izquierda. Recibirán todos aquellos nodos que tengan un nodo abajo a su derecha, el resto no.
- **Izquierda:** El nodo envía los datos al nodo de su izquierda. Recibirán todos aquellos nodos que tengan un nodo a su derecha, el resto no.
- **Abajo Izquierda:** El nodo envía los datos al nodo que esté abajo a la izquierda. Recibirán todos aquellos nodos que tengan un nodo arriba a su derecha, el resto no.

- **Abajo:** El nodo envía los datos al nodo que tenga abajo. Recibirán todos aquellos nodos que tengan un nodo arriba, el resto no.
- **Abajo Derecha:** El nodo envía los datos al nodo que esté abajo a la derecha. Recibirán todos aquellos nodos que tengan un nodo arriba a su izquierda, el resto no.

Cabe destacar la utilización de la función *MPI_Barriers* para sincronizar los envíos y recibos de los visitantes. Esto provoca la ralentización general del programa. Existe una alternativa que se ha contemplado, pero no se ha llevado a cabo. La alternativa sería utilizar la función *MPI_Isend*. De ésta manera, el programa deja en un *buffer* los datos a enviar y sigue a la siguiente línea de código. Sin embargo, al de un tiempo habría que comprobar si dicho envío se ha realizado correctamente para saber si habría que volverlo a enviar o no.

5. Métodos principales

5.1 Iteraciones

Una vez inicializadas las estructuras pertinentes, se pueden realizar las iteraciones de la simulación.

- En primer lugar, se comprueba el grupo de vacunación, y se actualiza en caso de que sea necesario. También, se guardan en variables auxiliares los contadores de las listas.
- La primera lista que se recorre es la de los infectados, ya que se tiene que verificar que se puede cambiar de estado. A su vez, también se realizan las operaciones de propagación, para no tener que iterar sobre esas personas dentro de la iteración. En el caso de que se cambie de estado, no se mueve. Si no se cambia el estado, se calculan los nuevos valores de movimiento y se mueve a la persona infectada.
- Después, se itera sobre las personas vacunadas. Es necesario iterar antes sobre los vacunados que los infectados, ya que los no-infectados pueden pasar a este grupo, y no interesa mover a esas personas en esta iteración.
- Para finalizar con la lista de personas, quedan las personas no infectadas. En primer lugar, es necesario comprobar si se pueden vacunar. En el caso de que se vacunen, ya se moverán en la siguiente iteración. Si no se vacunan, se calcula los nuevos valores de movimiento y se mueve a la persona no infectada.
- Después, se envían los visitantes (accesos a otros nodos) a otros nodos. Para ello, se dispone de un método llamado `send_visitors()`.
- Después, se mueven a las personas que han llegado y se responde al nodo en el caso de que se haya podido mover, para que el nodo origen sepa si debe eliminar o no a la persona. También, se infectan las posiciones recibidas, sin la necesidad enviar respuesta al nodo origen.
- Una vez tratado los visitante, se liberan o vacían las listas a utilizar en la siguiente iteración.
- Para finalizar, se comprueba el *BATCH* actual. En el caso de que sea el estipulado, se realizan las siguientes acciones:
  - Guardar los valores de las métricas en su correspondiente lista.
  - Guardar los valores de las posiciones en su correspondiente lista.
  - Modificar las listas de las personas para que eliminar a las personas con identificador "-1" (`realocate_lists()`).

5.2 Método principal

Haciendo uso de todas las estructuras y funciones presentadas, estas tienen que ser inicializadas en el método principal. Este método, realiza las siguientes operaciones:

- En primer lugar, inicializa las variables necesarias para *MPI*.
- Si es el nodo 0, se comprueban los diferentes métodos de configuración del programa, y se inicializan lass varibles globales de configuración de la simulación.
- Se inicializan las variables globales.
- Se inicializa la librería de *gsl*, para el calculo de las probabilidades.
- En el caso de que sea el nodo 0, se calculan los cuadrantes, las personas a vacunar y la población por nodo. Una vez calculados, se envía cada variable con un `MPI_Bcast`.
- Inicializan las listas de manera dinámica.
- Para crear los *DataTypes*, en primer lugar se inicializan y se relaciona con una estructura ficticia, para que se pueda reconocer.
- Cada nodo, crea su población, con los parámetros calculados previamente (población por nodo, cuadrantes...).
- Iteraciones (Véase apartado 5.1), con un bucle hasta el número de iteraciones estipulado.
- Cada nodo envía sus métricas y posiciones al nodo 0.
- El nodo 0, imprime las métricas y las posiciones en su correspondiente fichero.
- Se liberan todas las estructuras que faltan por liberar.
- Se finaliza la simulación

5.3 Parametrización

En cuanto a la parametrización, los valores posibles a modificar para así obtener una mayor flexibilidad en el funcionamiento del programa son los siguientes:

- ***-s*** **o** ***–world_size***: Lado de la matriz cuadrada. Valor por defecto 8.
- ***-p*** **o** ***–percent***: Porcentaje de la población a vacunar por iteración. Valor por defecto 0.05.
- ***-g*** **o** ***–population_size***: Tamaño de población. Valor por defecto 60.
- ***-b*** **o** ***–batch***: Número de iteraciones por *Batch*. Valor por defecto 2.
- ***-i*** **o** ***–iter***: Número de iteraciones. Valor por defecto 10000.
- ***-d*** **o** ***–seed***: Número que actúa como semilla. Valor por defecto 3.
- ***-m*** **o** ***–ageMean***: Edad media de la población. Valor por defecto 28.

Todos los valores para dichos parámetros deben ser números enteros exceptuando el parámetro *percent*, el cual debe ser un número decimal.

Así mismo, también se dispone del parámetro *-h o –help* el cual mostrará por pantalla la misma información anteriormente descrita sobre los parámetros y su descripción. Además mostrará varios ejemplos sobre la ejecución de los programas con la parametrización. Ejemplo:

- **Por línea de comando** : ./run.sh "covid.out -s8 -p0.2 -g60 -b2 -i10 -d2 -m20" 4
- **Mediante fichero** : ./run.sh "covid.out parametros.txt" 4

En caso de indicar los parámetros mediante un fichero, la estructura a seguir es la siguiente:

```
-l
-s 16
-p 0.05
-g 60
-b 2
-i 1
-d 3
-m 30
```

El primer valor *-l* es un valor aleatorio que debe indicarse. Durante el desarrollo de la lectura de parámetros mediante fichero, no se ha conseguido poder leer el primer parámetro *-s* sin tener un valor aleatorio en la primera línea del fichero.

Por defecto la parametrización debe indicarse mediante un fichero. En caso de que se quiera utilizar por línea de comandos, se deberá des-comentar las líneas descritas en el fichero *configuration.c*.

5.4 Tiempos

Ejecución	N 1	N 2	N 3
Población	62	252	1000
Iteraciones	50000	5000	10000
Mundo	8 x 8	16 x 16	16 x 16
Procesadores	4	4	4
Tiempo(segundos)	60.2354	20.5753	40.3647

Table 5.1: Comparativa de tiempos de diferentes simulaciones

6. Problemas en el desarrollo y Gestión del tiempo del proyecto

En este apartado, se documentan los problemas que se han tenido durante el desarrollo de la paralelización del proyecto. Se ha añadido este apartado para contar la experiencia personal del grupo y utilizarlo como lecciones aprendidas para el futuro.

- **Identificador negativo:** se han tenido dificultades para sincronizar los valores de las listas con el escenario, durante la simulación. Durante el desarrollo del proyecto, se han tenido problemas relacionados con este aspecto, con diferentes fuentes de fallo.
- **Diagonales:** las diagonales han dificultado el desarrollo del proyecto. A pesar de que funcionan correctamente estos accesos, ha demorado en gran medida el desarrollo del proyecto, ya que al añadir más direcciones posibles, aumentaban el número de posibilidades de errores de implementación.
- **Pruebas de MPI:** para realizar todas las funcionalidades de MPI, se han implementado una a una, para poder comprobar su correcto funcionamiento. Esto ha generado, que las pruebas demoren el desarrollo del proyecto, pero era algo necesario de realizar ya que sino se incrementaban las probabilidades de error.
- **Diseño:** se ha dedicado mucho tiempo a pensar el diseño del proyecto, para poder realizar un proyecto acorde al objetivo de la asignatura: la paralelización y eficiencia del mismo. Sin embargo, se ha gestionado mal el tiempo de la implementación, ya que MPI ha generado más problemas de los esperados. No obstante, creemos que el diseño del proyecto es el correcto, por lo que es algo positivo, pero en futuros proyectos tendremos en cuenta en primer lugar el conocimiento de la complejidad de la herramienta.
- ***Testing:*** la fase más larga de todo el proyecto ha sido el *testing* o pruebas. Una vez finalizada la implementación, era necesario comprobar su correcto funcionamiento. A pesar de que cada vez que se implementaba un nuevo módulo se hacían pruebas del mismo, era necesario probar todo el programa en su conjunto. Para ello, se ha hecho uso de los mensajes que se imprimían por pantalla, ficheros que genera el programa, herramientas de búsqueda por terminal (por ejemplo, el comando `grep`) y herramientas externas como Excel.