Proyecto Intermedio - Simulación de Percolación en Redes 2D

J. Mayorga, J. Gutierrez & J. Vanegas
Introducción a la Computación Científica y de Alto Rendimiento
Universidad Nacional de Colombia
Bogotá D.C.

11 de junio de 2025

Resumen

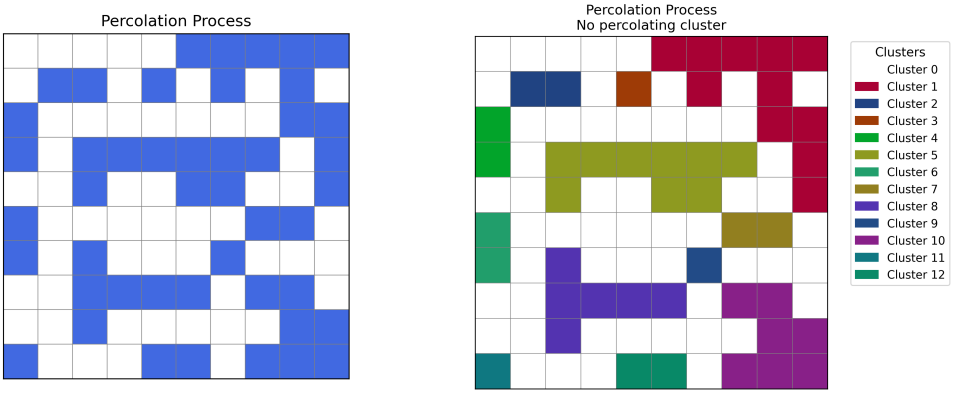
Este trabajo presenta una implementación computacional del modelo de percolación por sitio en redes bidimensionales, utilizando el algoritmo de Hoshen-Kopelman para la identificación de clústeres. Se estudian las propiedades fundamentales del sistema cerca del punto crítico $p_c \approx 0{,}5927$, incluyendo la probabilidad de percolación $P(p, L)$ y el tamaño normalizado del clúster percolante más grande $s(p, L)$. Mediante simulaciones exhaustivas para diferentes tamaños de red L desde 100 hasta 2000 y probabilidades de ocupación p entre 0 y 1, se caracteriza la transición de fase del sistema. Adicionalmente, se analiza el desempeño computacional del algoritmo bajo diferentes niveles de optimización -O1 y -O3, identificando los cuellos de botella principales mediante técnicas de profiling. Los resultados muestran el comportamiento característico de transición de fase, con $P(p, L)$ aproximándose a una función escalón cuando $L \to \infty$, y $s(p, L)$ mostrando máxima fluctuación cerca de p_c. El análisis de rendimiento revela que las operaciones con hash maps y la generación de números aleatorios consumen aproximadamente el 55 % y 23 % del tiempo de ejecución respectivamente.

1. Introducción

El fenómeno de percolación es un modelo fundamental en física estadística y teoría de redes, que describe la conectividad y propagación en medios desordenados. Sus aplicaciones abarcan desde la propagación de incendios forestales y epidemias hasta la conductividad de materiales porosos. En su forma más simple, el modelo representa un sistema donde cada sitio (o enlace) de una red se ocupa con probabilidad p, estudiándose la formación de caminos conectados. Un aspecto central es su transición de fase: al superar un valor crítico p_c, emerge un 'clúster percolante' que atraviesa el sistema, análogo a transiciones de fase en magnetización o condensación de gases [1,4].

En este trabajo se estudia el modelo de percolación por sitio en una malla cuadrada bidimensional de tamaño $L \times L$, donde cada celda se ocupa independientemente con probabilidad p. Dos celdas ocupadas son vecinas en la dirección de von Neumann (norte, sur, este u oeste), formando clústeres o componentes conexas. Como ilustran las Figuras 1

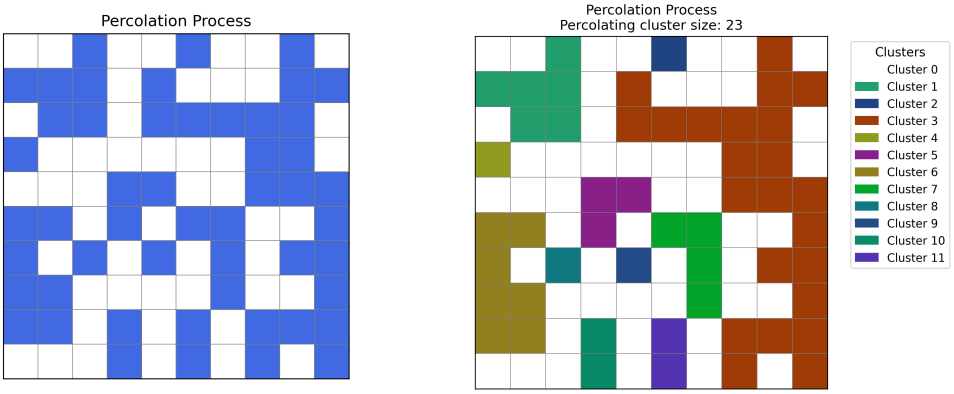
y 2, para $p < p_c$ predominan clústeres finitos (Figura 1), mientras que para $p > p_c$ surge un clúster percolante (Figura 2). El valor crítico para redes cuadradas bidimensionales es $p_c \approx 0{,}5927$ [4], aunque este límite teórico supone sistemas infinitos, y las simulaciones requieren analizar redes finitas.



(a) Red cuadrada.

(b) Clústeres individuales.

Figura 1: Proceso de percolación para una red cuadrada por debajo de la probabilidad crítica, $p = 0{,}5$ y $L = 10$.



(a) Red cuadrada.

(b) Clúster percolante.

Figura 2: Proceso de percolación para una red cuadrada, $p = 0{,}58$ y $L = 10$, donde emerge un clúster percolante.

En este sentido, los objetivos específicos de este proyecto son calcular la probabilidad de percolación $P(p, L)$, es decir, la fracción de configuraciones con un clúster que conecta lados opuestos (vertical u horizontalmente), analizar el tamaño promedio del clúster más grande $s(p, L)$ en función de p y L y evaluar el impacto del nivel de optimización e implementación de simulaciones computacionales en el tiempo de cómputo para distintos L.

Así, dado el carácter probabilístico del modelo, las simulaciones se repiten múltiples veces para cada combinación de p, L y nivel de optimización, permitiendo estimar promedios y desviaciones estándar. Computacionalmente, el código se implementó en C++ con estructura modular, usando un `Makefile` para automatizar ejecución, análisis de desempeño, pruebas unitarias y perfilamiento, como se detalla en la metodología.

2. Metodología

Para realizar las simulaciones se trabajo con el programa contenido en el repositorio `Percolation_2025_I_IntroSciCompHPC` [3] en su rama `master` cuya estructura es

```
.Percolation_2025_I_IntroSciCompHPC
|__ .github
|   |__ workflows
|__ .vscode
|__ figures
|__ include
|__ profiling
|__ src
|   |__ resultados
|__ Makefile
```

tal que en `figures` se encuentran tanto las figuras utilizadas en este trabajo como los códigos necesarios para realizarlas, en `include` se encuentran las implementaciones de funciones y librerías necesarias para llevar a cabo las simulaciones, teniéndose `catch2`, `hoshen_kopelman.h`, `matrix.h`, `probvalues.h` y `union_find.h`, en `profiling` se encuentra el flat profile de la simulación del sistema para $L = 128$ y $p = 0.59271$ `profiling-report.txt`.

Por otro lado, en `src` se encuentran los programas:

- `hoshen_kopelman.cpp`: Contiene la función `hoshen_kopelman` que permitirá identificar los clústeres en una red bidimensional y la función `print_clusters` que permitirá obtener como salida la red bidimensional con los clústeres identificados en, por ejemplo, un archivo.txt.
- `matrix.cpp` : Contiene la función `generatematrix` que generara aleatoriamente mediante la semilla `seed` una red bidimensional dada la dimensión L y la probabilidad p, estos argumentos se pueden recibir por linea de comandos pero en caso de no pasar `seed`, se tomara de forma predeterminada como un número aleatorio mediante `std::random_device`, tambien contiene a `printmatrix` que permitirá obtener la salida de la red bidimensional como una matriz.
- `probvalues.cpp`: Contiene la función `generate_p_values` que permitirá generar N valores para p en el intervalo [0, 1] garantizando la existencia de minimo 10 en el intervalo [0.55, 0.65].
- `script.sh`: Este script en `bash` llevara a cabo la simulación para obtener los datos necesarios para encontrar $P(p, L)$ y $s(p, L)$.
- `printvalues.cpp`: Permite obtener la salida de `generate_p_values(N)` en, por ejemplo, un archivo .txt.
- `report.tex`: Archivo .tex que contiene el reporte en LaTeX.
- `report.bib`: Bibliografia para report.tex

- `resultados/resumen.csv`: Archivo .csv con un resumen de los datos obtenidos por medio de `script.sh`.
- `test.cpp`: Función main adaptada para realizar distintos tests con `Catch2`.
- `union_find.cpp`: Contiene las funciones `UnionFind`, `find` y `unite` que se utilizaran en la función `hoshen_kopelman`.
- `main.cpp` : Función main, recibe como entrada la dimensión de la matriz L, la probabilidad p y de forma opcional la semilla `seed` para el llenado aleatorio por medio de `generatematrix`, realiza la clasificación de los clusters por medio de `hoshen_kopelman` e imprime la matriz sin clasificar, la matriz con los clústeres clasificados, si hay percolación y en dado caso el tamaño del clúster percolante.
- `time_main.cpp` Función main para calcular los tiempos de Wall y CPU para la función `main` por medio de la libreria `chrono` y `ctime`.

Finalmente, mediante el archivo `Makefile` se automatizan todos los procesos descritos anteriormente.

2.1. Algoritmo HK

El algoritmo de Hoshen-Kopelman (HK), descrito originalmente por Joseph Hoshen y Raoul Kopelman en 1976 , es un método computacionalmente eficiente para identificar y etiquetar clústeres o componentes conectados en una cuadrícula o red. Su diseño está optimizado para minimizar el número de comparaciones necesarias, lo que lo hace particularmente adecuado para el análisis de fenómenos de percolación [2].

1) **Inicialización del proceso**
   La función `hoshen_kopelman` recibe como entrada una matriz que representa una cuadrícula, donde los valores, usualmente ceros y unos, indican si un sitio está vacío u ocupado, respectivamente. También recibe el tamaño lateral de la cuadrícula, L. Se crea un vector `labels` del mismo tamaño que la matriz, inicializado con ceros, para almacenar las etiquetas de los clústeres; un 0 en `labels` significa que la celda está vacía o aún no ha sido etiquetada. Se inicializa una estructura `UnionFind` (`uf`) con un tamaño N+1 (siendo N = L*L), la cual es esencial para manejar la equivalencia entre las etiquetas de los clústeres. Finalmente, `next_label` se establece en 1, marcando la primera etiqueta disponible para asignar a un nuevo clúster.

   i) **Estructura `UnionFind`:** La estructura `UnionFind` es fundamental para la eficiencia del algoritmo, permitiendo la gestión de conjuntos disjuntos. Se compone de un vector `parent`, donde `parent[i]` almacena el padre del elemento i; inicialmente, cada elemento es su propio padre. El método `find(x)` localiza la raíz del conjunto al que pertenece x, optimizando las búsquedas futuras mediante la compresión de caminos, lo que hace que los nodos apunten directamente a la raíz. El método `unite(x,y)` fusiona los conjuntos que contienen a x e y. Para ello, encuentra las raíces de x e y, y si estas son diferentes, establece la raíz de uno como padre de la raíz del otro, logrando así la unión de los dos conjuntos.

2) Etiquetado preliminar (Primera pasada)
El algoritmo recorre cada celda de la matriz. Para cada celda (i, j) que se encuentra ocupada,es decir, `matrix[id] == 1)`, se examinan sus vecinos adyacentes: "norte"(up) y "oeste"(left). Si ambos vecinos están vacíos o no ocupados, se le asigna una nueva etiqueta a la celda actual `(labels[id] = next_label++)`. Si solo el vecino Norte está ocupado, la celda actual hereda la etiqueta de su vecino Norte `(labels[id] = up)`. De manera similar, si solo el vecino Oeste está ocupado, la celda actual recibe la etiqueta de su vecino Oeste `(labels[id] = left)`. Cuando ambos vecinos (Norte y Oeste) están ocupados, la celda actual se etiqueta con la menor de las dos etiquetas de sus vecinos `(labels[id] = std::min(up, left))`. Además, se utiliza la estructura UnionFind para unir las dos etiquetas `(uf.unite(up, left))`, indicando que pertenecen al mismo clúster. Esta unión es crítica porque permite reconocer que dos etiquetas inicialmente distintas forman parte de un mismo clúster.

3) Compactación de etiquetas (Segunda Pasada)
Después de la primera pasada, es posible que algunas celdas tengan etiquetas diferentes pero pertenezcan al mismo clúster debido a las uniones realizadas por UnionFind. Esta etapa se encarga de ”compactar” esas etiquetas, asegurando que todos los miembros de un mismo clúster compartan una única etiqueta unificada. Para esto, se emplean dos `(unordered_map): root_to_compact`, que relaciona la ”raíz"de un clúster obtenida mediante `UnionFind::find()` con una etiqueta compactada y consecutiva, y `cluster_sizes`, que almacena el tamaño de cada clúster utilizando estas etiquetas compactadas. El algoritmo itera sobre todas las celdas en labels. Para cada celda que ya ha sido etiquetada, es decir, no es cero, se busca la raíz de su etiqueta utilizando `uf.find(labels[i])`. Si esta raíz aún no ha sido mapeada a una etiqueta compactada, se le asigna una nueva `(compact_label++)`. Posteriormente, la etiqueta de la celda actual se actualiza a su etiqueta compactada `(labels[i] = label)`, y el tamaño del clúster correspondiente en `cluster_sizes[label]` se incrementa.

4) Detección de percolación
La percolación es un fenómeno que ocurre cuando un clúster logra conectar lados opuestos de la matriz. Para identificar esto, se crean cuatro conjuntos desordenados `(unordered_set)` que almacenan las etiquetas de los clústeres que tocan cada uno de los cuatro bordes de la matriz: top (fila superior), bottom (fila inferior), left (columna izquierda), y right (columna derecha). Se recorren las celdas de estas filas y columnas específicas, insertando las etiquetas de las celdas ocupadas en sus respectivos conjuntos.

Para la percolación vertical, se verifica si alguna etiqueta está presente tanto en el conjunto top como en el conjunto bottom. Si se encuentra una etiqueta común, indica la existencia de un clúster que conecta la parte superior e inferior de la matriz. De manera similar, para la percolación horizontal, se comprueba si alguna etiqueta existe tanto en el conjunto left como en el conjunto right. Si es así, significa que un clúster conecta el lado izquierdo con el derecho de la matriz. Un conjunto `percolating_labels` almacena las etiquetas de los clústeres que percolan, ya sea vertical u horizontalmente. La variable booleana `percolates` indica si se ha detectado al menos un clúster percolante. Finalmente, se calcula `max_cluster_size` entre todos los clústeres que percolan.

5) Conclusión del algoritmo
La función concluye devolviendo una estructura `ClusterInfo`. Esta estructura encapsula los resultados clave del análisis, que incluyen `cluster_sizes`, un mapa que asocia las etiquetas compactadas con sus respectivos tamaños de clúster, `percolates`, un valor booleano que indica si se produjo percolación, `max_cluster_size` el tamaño del clúster percolante más grande, y `labels` el vector final de etiquetas compactadas asignadas a cada celda de la matriz.

2.2. Wall time y CPU time

Con el fin de obtener los Wall time y CPU time para el tiempo de computo de nuestra función `main.cpp` se utilizo la función `time_main.cpp` esta realiza el mismo procedimiento que `main.cpp`, sin embargo, no imprime la salida de la matriz después del llenado y después de la identificación de los clústeres, tal que su unica salida es de la forma L, Wall time y CPU time y mediante el comando de `GNU Parallel`

```
  parallel './time_main0{1}.x {3} {2} >> time-{1}-{3}.txt'
::: 1 3 ::: $(cat probabilidades10.txt) ::: {100..2000..100}'
```

se obtienen distintos archivos .txt de la forma `time-opt-`L`.txt` que permitirán graficar los tiempos de computo para los niveles de optimización `-O1` y `-O3`. Notese que no hay valor de entrada por la linea de comandos para `seed`, por ende, la salida es aleatoria.

2.3. Script en bash `script.sh`

El script en bash automatiza la ejecución de simulaciones de percolación para distintos tamaños de red y probabilidades, utilizando un programa ejecutable llamado `main.x`. Al iniciar, elimina cualquier carpeta de resultados previa y crea una nueva estructura de directorios para almacenar datos crudos, archivos auxiliares y gráficos. Luego, define una lista de tamaños de red L que serán evaluados y genera todas las combinaciones posibles con los valores de probabilidad listados en el archivo `probabilidades50.txt`, es decir, mediante 50 probabilidades distintas obtenidas mediante `probvalues.cpp`. Estas combinaciones se almacenan en un archivo temporal para su procesamiento posterior.

El núcleo del script es una función llamada `simulate`, que toma como entrada un valor de L y una probabilidad p. Para cada combinación, ejecuta 10 bloques de 10 repeticiones del programa principal, recolectando el tamaño del mayor clúster percolante y si ocurrió o no percolación en cada ejecución. Con esta información, calcula el valor medio de la probabilidad de percolación $P(p, L)$ y el tamaño promedio del clúster percolante $s(p, L)$ normalizado respecto al área de la red, junto con sus respectivas desviaciones estándar. Los resultados se guardan en archivos individuales para cada combinación y también se imprimen en formato `.csv` como una línea resumen disponible para el usuario en caso de desear ver la salida de datos especifica. Cabe mencionar que, como se pudo inferir, la entrada de la semilla `seed` por la linea de comandos es nula, se considera que dado que se realizan por cada valor de probabilidad p 100 iteraciones y se itera 50 veces por tamaño, la magnitud del tamaño de los datos utilizados es lo suficientemente grande como para permitir un muestreo estadístico solido y confiable sin importar la generación de estos, es decir, los resultados son absolutamente reproducibles a pesar de la aleatoriedad de la semilla.

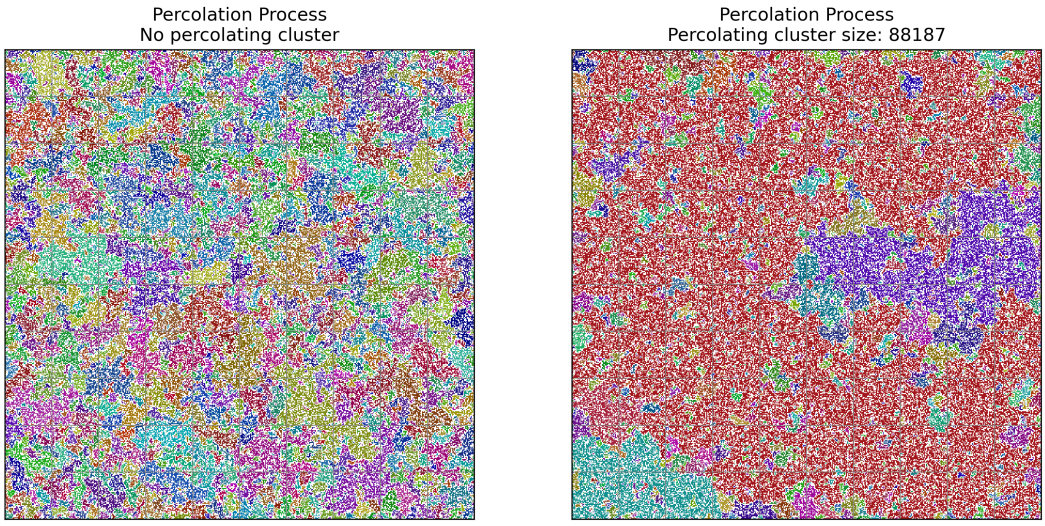
Finalmente, el script utiliza `GNU parallel` para ejecutar múltiples simulaciones de forma concurrente, acelerando considerablemente el proceso. Una vez completadas todas las combinaciones, se genera un archivo `resumen.csv` que contiene los resultados ordenados por tamaño de red y probabilidad. Finalmente, se limpian los archivos temporales utilizados durante la ejecución y se ordenan los archivos de resultados finales para facilitar su análisis posterior

2.4. Código para gráficar

Para gráficar las matrices llenadas como las matrices con los clústeres identificados se utilizaron los codigos `visualization.cpp` y `clustervisualization.cpp` que permiten la salida en archivos.txt de las matrices obtenidas y controlar la dimensión L, la probabilidad p y la semilla `seed` con el fin de obtener gráficas reproducibles. De esta manera, mediante los scripts de `python visualize.py` y `clustervisualize.py` reciben las matrices, por ejemplo, de la forma:

```
./figures/visualization.x 10 0.5 0.6 > figures/data1.txt
./figures/visualization.x 10 0.58 15 > figures/data2.txt
python3 ./figures/visualize.py
./figures/clustervisualization.x 10 0.5 0.6 > figures/data_clusters1.txt
./figures/clustervisualization.x 10 0.58 15 > figures/data_clusters2.txt
./figures/clustervisualization.x 500 0.55 0.6 > figures/data_clusters3.
   txt
./figures/clustervisualization.x 500 0.595 0.6 > figures/data_clusters4.
   txt
python3 ./figures/clustervisualize.py
```

De esta manera, en el ejemplo presentado, la salida para `data1.txt` y `data_clusters1.txt` junto con la salida para `data2.txt` y `data_clusters2.txt` permiten obtener las figuras 1 y 2. Por otro lado, la salida de `data_clusters3.txt` y `data_clusters4.txt` nos permiten obtener las figuras 3a y 3b.



(a) $p = 0.5$. (b) $p = 0.595$.

Figura 3: Proceso de percolación para una red cuadrada de dimensiones $L = 500$.

Por otro lado, el script `figures.py` permite obtener las gráficas para $P(p, L)$ y $s(p, L)$ gracias a la salida obtenida mediante el script en `bash` y el script `time_figure.py` permite obtener la gráfica para el tiempo de computo en función del tamaño del sistema para dos niveles distintos de optimización mediante los datos obtenidos por los archivos `time-opt-`L`.txt`

2.5. Automatización por medio de Makefile

Para automatizar los procesos descritos anteriormente se crearon distintos targets para el archivo `Makefile` destinados a varias tareas especificas. Los mas relevantes para el programa y que no son subdependencias de otro target son:

- `make simul`: Compila y ejecuta una simulación rápida con parámetros predeterminados (L=4, p=0.6, `seed`=10). Útil para verificar que el código funciona correctamente.
- `make test`: Ejecuta pruebas unitarias utilizando la librería `Catch2`. Puede filtrar pruebas específicas usando `FILTER=...`.
- `make report`: Compila el documento LaTeX del reporte, procesa la bibliografía si existe, y genera un PDF final. Verifica que todas las figuras necesarias estén presentes antes de compilar.
- `make profile`: Realiza un análisis de rendimiento completo, incluyendo:
  - Flat profile con `gprof` (guardado en `profiling/analysis.txt`)
  - Flame graph con `perf` (guardado como `profiling/flamegraph.svg`)

  Usa parámetros de prueba (L=8, p=0.5, `seed`=10).
- `make debug`: Compila el programa con flags de depuración y lo ejecuta bajo GDB para facilitar la identificación de errores.
- `make valgrind`: Realiza un análisis de memoria con Valgrind para detectar leaks y accesos inválidos, usando parámetros de prueba (L=6, p=0.6, `seed`=10).
- `make clean`: Elimina todos los archivos temporales, ejecutables, datos de cobertura y resultados de profiling. No afecta las figuras ni el reporte final.
- `make profiling-report.txt`: Genera un flat profile detallado usando `perf` para el caso crítico (L=128, p=0.59271, `seed`=10).
- `make run-simulation`: Ejecuta la simulación completa generando todos los archivos de datos necesarios para las gráficas, pero sin generar las figuras.
- `make figures`: Genera todas las figuras del análisis, incluyendo:
  - Gráficas de probabilidad de percolación y tamaño de clústers
  - Visualizaciones de mallas de ejemplo
  - Análisis de tiempos de ejecución

  Requiere que los datos de simulación ya estén generados.

- `make clean-figures`: Elimina todas las figuras generadas, permitiendo forzar su regeneración.
- `make check-figures`: Verifica si todas las figuras necesarias existen, y si no, las genera automáticamente.
- `make help`: Muestra los distintos targets y su uso.

3. Resultados

3.1. Figura para el tiempo de computo en función del tamaño del sistema

Después de realizar el proceso descrito en la sección 2.2 y ejecutar el script `time_figure.py` se obtuvo la gráfica presentada en la figura 4.

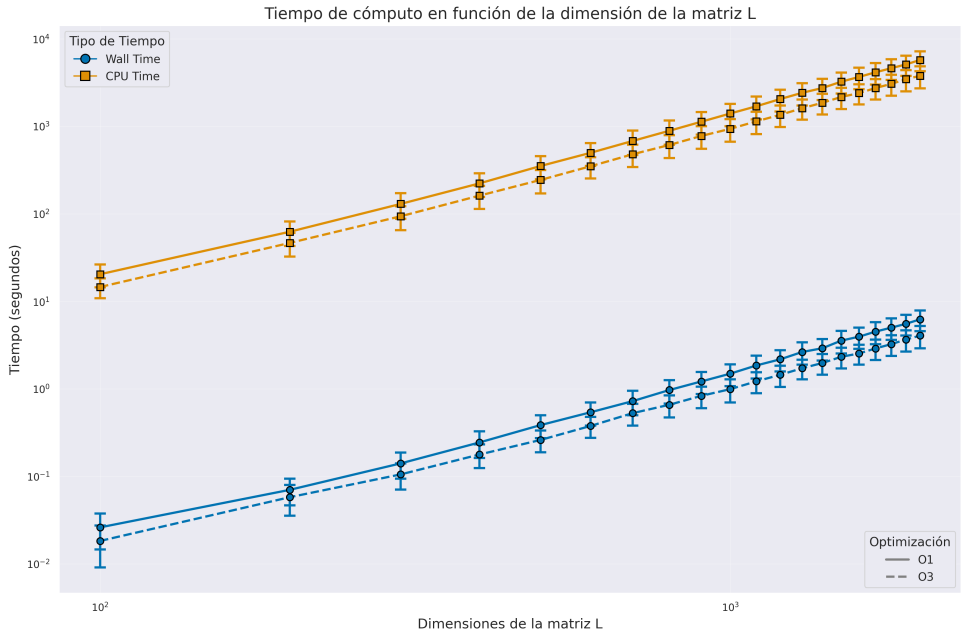


Figura 4: Tiempos de computo, Wall time y CPU time en función de la dimensión L de la red bidimensional para los niveles de optimización `-O1` y `-O3`.

Como se observa, el tiempo de CPU es relativamente mayor al tiempo de Wall debido al uso de `GNU Parallel`, ademas, también se encuentra una mínima diferencia para ambos niveles de optimización, siendo `-O3` mas rápido. Por estos motivos, para el ejecutable de la función `main.cpp` se decidió utilizar la bandera de optimización `-O3` para su compilación y su futura ejecución en las distintas tareas del programa, como, por ejemplo, su aplicación en el script en bash.

3.2. Optimización por medio del flat profile

Después de obtener el flat profile mediante `perf` con el target `make profiling-report.txt` para el caso crítico, se obtuvo el reporte

`profiling-report.txt`.

En este, se realizaron 76 muestras del evento `cycles:Pu` que corresponde a ciclos de CPU, 64927805 ciclos totales sin tener muestras perdidas obteniendose datos completos. Así, para una sola simulación el tiempo de programa fue del 82.36 % para la función `main` repartiendose de la forma

- 54.77 % para la función `hoshen_kopelman`, donde las funciones mas costosas son `std::unordered_map::operator[]` con un 27.69 % debido al acceso e inserción en `hash map` y `std::unordered_map::find` con un 13.16 % debido a la búsqueda en `hash map`.
- 22.86 % para la función `generatematrix` donde 21.36 % se destino a `std::uniform_real_distribution::operator()`.
- 3.06 % y 1.66 % para las funciones `printmatrix` y `print_clusters` respectivamente.

En consecuencia, se observa que los principales cuellos de botella se encuentran en las operaciones de `unordered_map`, relacionados con el algoritmo HK y en la generación de números aleatorios para el llenado de las matrices. De esta manera, se propuso optimizar el las funciones `hoshen_kopelman` y `generatematrix` para obtener una simulación mas eficiente. Los cambios implementados se encuentran en la rama `optimized_main`.

Primero, se implemento para `generatematrix` la función

```
FastRandom::FastRandom(int seed) : dis(0.0, 1.0) {
    if (seed == -1) {
        std::random_device rd;
        gen.seed(rd());
    } else {
        gen.seed(seed);
    }
}

// Implementacion de fill_vector
void FastRandom::fill_vector(std::vector<double>& vec) {
    for (auto& val : vec) {
        val = dis(gen);
    }
}

// Implementacion de next()
inline double FastRandom::next() {
    return dis(gen);
}
```

tal que la función `FastRandom` esta incluida en `matrix.h`,

```
#ifndef MATRIX_H
#define MATRIX_H

#include <vector>
#include <random>
```

```cpp
#include <iostream>

class FastRandom {
private:
    std::mt19937 gen;
    std::uniform_real_distribution<double> dis;

public:
    FastRandom(int seed = -1);
    void fill_vector(std::vector<double>& vec);
    inline double next();
};

std::vector<int> generatematrix(int L, double p, int seed = -1);
void printmatrix(const std::vector<int>& matrix, int L);

#endif
```

y permitía acelerar la generación aleatoria de las matrices. De la misma manera, para `hoshen_kopelman` se realizo la optimización

```cpp
ClusterInfo hoshen_kopelman(const std::vector<int>& matrix, int L) {
    int N = L * L;
    std::vector<int> labels(N, 0);  // 0: vacio
    UnionFind uf(N+1);
    int next_label = 1;

    // Paso 1: etiquetado preliminar con vecinos N y O
    for (int i = 0; i < L; ++i) {
        for (int j = 0; j < L; ++j) {
            int id = i * L + j;
            if (matrix[id] == 0) continue;

            int up = (i > 0 && matrix[(i - 1) * L + j] == 1) ? labels[(i
                - 1) * L + j] : 0;
            int left = (j > 0 && matrix[i * L + (j - 1)] == 1) ? labels[
               i * L + (j - 1)] : 0;

            if (up == 0 && left == 0) {
                labels[id] = next_label++;
            } else if (up != 0 && left == 0) {
                labels[id] = up;
            } else if (up == 0 && left != 0) {
                labels[id] = left;
            } else {
                labels[id] = std::min(up, left);
                uf.unite(up, left);
            }
        }
    }

    // OPTIMIZACION 1: Reservar capacidad para hash maps
    std::unordered_map<int, int> root_to_compact;
    std::unordered_map<int, int> cluster_sizes;

    // Estimar numero de clusters (tIpicamente mucho menor que N)
    int estimated_clusters = std::min(next_label, N/4);
```

```
    root_to_compact.reserve(estimated_clusters);
    cluster_sizes.reserve(estimated_clusters);

    int compact_label = 1;

    // OPTIMIZACION 2: Usar emplace en lugar de operator[] y find
    for (int i = 0; i < N; ++i) {
        if (labels[i] == 0) continue;
        int root = uf.find(labels[i]);

        // Usar emplace para evitar doble lookup
        auto [it, inserted] = root_to_compact.emplace(root,
            compact_label);
        if (inserted) {
            compact_label++;
        }

        int label = it->second;
        labels[i] = label;

        // Incrementar cluster size directamente
        cluster_sizes[label]++;
    }

    // OPTIMIZACION 3: Reservar capacidad para sets y usar vectores para
        bordes
    std::vector<int> top_labels, bottom_labels, left_labels,
       right_labels;
    top_labels.reserve(L);
    bottom_labels.reserve(L);
    left_labels.reserve(L);
    right_labels.reserve(L);

    // Recopilar labels de bordes
    for (int i = 0; i < L; ++i) {
        if (labels[i] > 0) top_labels.push_back(labels[i]);
        if (labels[(L - 1) * L + i] > 0) bottom_labels.push_back(labels
            [(L - 1) * L + i]);
        if (labels[i * L] > 0) left_labels.push_back(labels[i * L]);
        if (labels[i * L + (L - 1)] > 0) right_labels.push_back(labels[i
            * L + (L - 1)]);
    }

    // OPTIMIZACION 4: Usar sets solo cuando sea necesario
    std::unordered_set<int> percolating_labels;

    // Ordenar para hacer interseccion mas eficiente
    std::sort(top_labels.begin(), top_labels.end());
    std::sort(bottom_labels.begin(), bottom_labels.end());
    std::sort(left_labels.begin(), left_labels.end());
    std::sort(right_labels.begin(), right_labels.end());

    .
    .
    .

    return {
        cluster_sizes,
```

```
88          percolates,
89          max_cluster_size,
90          labels
91      };
92
93  }
```

de la que se esperaba que el manejo de memoria por medio de `reserve` y `emplace` mejorara la eficiencia del algoritmo HK y redujera los tiempos de ejecución de nuestro programa.

Sin embargo, a pesar de las optimizaciones realizadas los valores reportados en el flat profile con `perf` y obtenido para estas funciones, al momento de revisar el reporte se encontro que los porcentajes obtenidos no solo se daban en las mismas funciones que se encontraron en el código no optimizado sino que ademas, los porcentajes eran de la misma magnitud teniendo en cuenta las fluctuaciones propias debido a la naturaleza del procesador donde se llevaron a cabo las simulaciones. Este reporte se puede revisar en el repositorio para la rama `optimized_main` y se puede obtener de la misma forma que en la rama `master` para el target `make profiling-report.txt`.

Estos resultados se interpretaron tal que el algoritmo HK es una excelente implementación para la tarea que realiza y cualquier implementación subyacente no afectara el desarrollo de esta, por estos motivos, se descarto cualquier intento de optimización posterior y se decidió utilizar los programas ya realizados.

3.3. Probabilidad de percolación $P(p, L)$

Los resultados obtenidos para la probabilidad de percolación $P(p, L)$ por medio del script en `bash` son presentados en la figura 5.

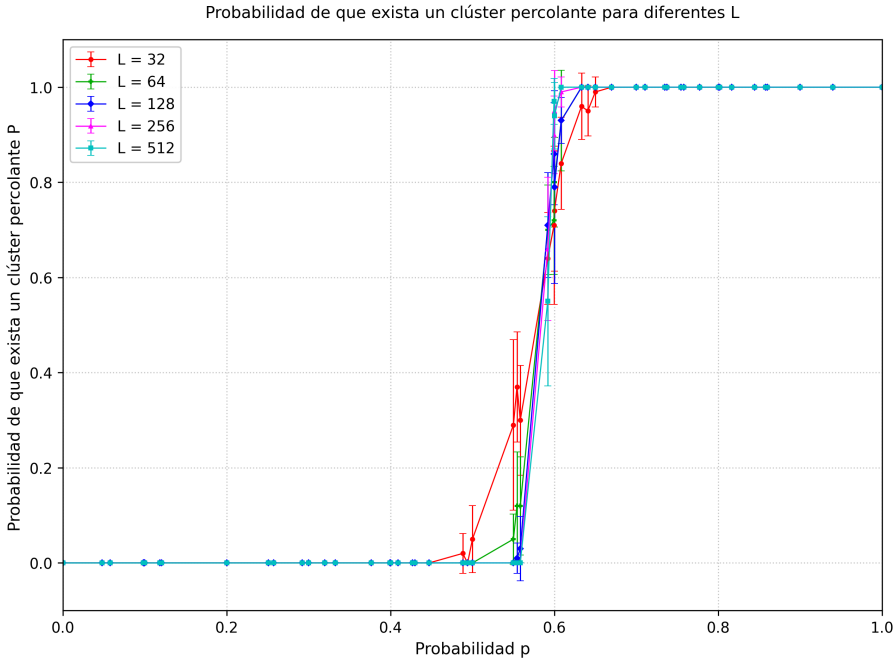


Figura 5: Probabilidad de percolación en función de la probabilidad de llenado para distintos L.

Como se puede observar, $P(p, L)$ se aproxima a una función de Heaviside $\Theta(p - p_c)$ para $L \to \infty$, tal que $p_c \approx 5.9$, es decir, el limite termodinámico. También, se observa que las incertidumbres son mayores en la vecindad de p_C, lo cual es esperable porque cerca del punto crítico las fluctuaciones son más intensas.

En ese sentido, es posible observar la transición de fase del sistema entre $p \in (0.55, 0.62)$ al pasar $P(p < p_C, L) = 0$ a $P(p > p_C, L) = 1$. Sin embargo, el comportamiento en general para $P(p, L)$ varia según la dimensión del sistema L cerca de p_c, a este fenómeno se le conoce como escalamiento crítico.

3.4. Tamaño promedio del clúster percolante mas grande $s(p, L)$

Los resultados obtenidos para el tamaño medio de clúster de percolación $s(p, L)$ por medio del script en `bash` son presentados en la figura 6.

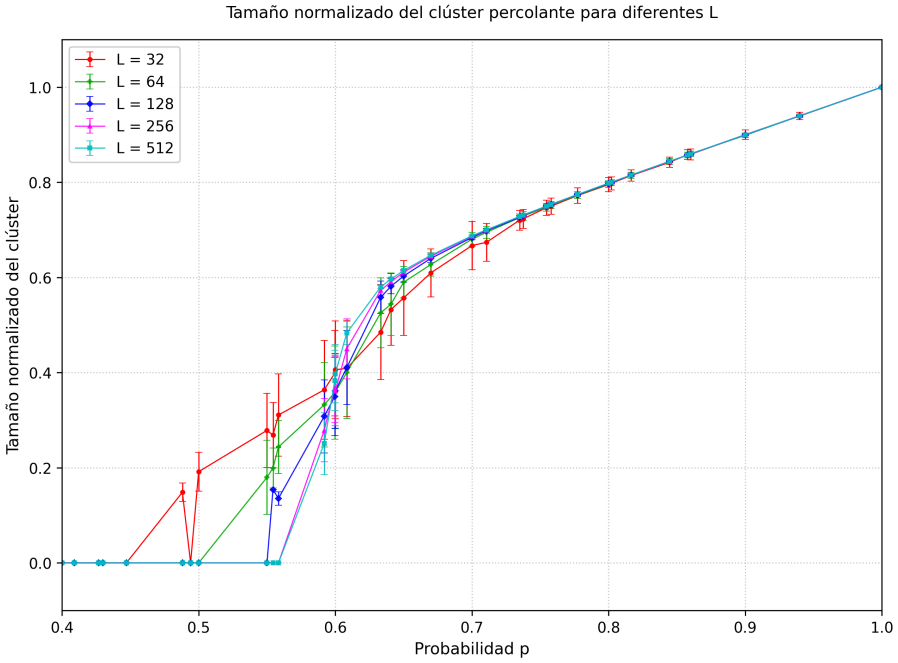


Figura 6: Tamaño medio de clúster de percolación en función de la probabilidad de llenado para distintos L.

Como se puede observar, $s(p, L)$ se observa una tendencia en su comportamiento para $L \to \infty$ de converger a una misma solución, la cual es distinta de cero cerca de la probabilidad critica $p_c \approx 5.9$.

4. Conclusiones

El estudio realizado permitió caracterizar satisfactoriamente el fenómeno de percolación en redes bidimensionales mediante simulaciones computacionales. Los principales hallazgos y conclusiones son:

- Se confirmó el comportamiento crítico del sistema alrededor de $p_c \approx 0{,}5927$, donde la probabilidad de percolación $P(p, L)$ muestra una transición abrupta al aumentar el

tamaño de la red L, aproximándose a una función escalón en el límite termodinámico, $L \to \infty$.

- El algoritmo de Hoshen-Kopelman demostró ser eficiente para el análisis de sistemas de tamaño moderado, $L \leq 2000$, aunque el análisis de profiling reveló que aproximadamente el 78 % del tiempo de ejecución se concentra en solo dos funciones: la identificación de clústeres (55 %) y la generación de la matriz aleatoria (23 %).
- Las optimizaciones implementadas, -O1 y -O3, mostraron mejoras modestas en el tiempo de ejecución, sugiriendo que el algoritmo ya está altamente optimizado en su forma básica. Los intentos de optimización manual adicional no produjeron mejoras significativas.
- Las simulaciones para diferentes tamaños de red permitieron observar el fenómeno de escalamiento crítico, donde las propiedades del sistema cerca de p_c dependen fuertemente del tamaño L, mientras que lejos del punto crítico el comportamiento es más universal.
- La implementación modular y el uso de herramientas modernas (`GNU Parallel`, `perf`, `Catch2`) demostraron ser efectivos para manejar la naturaleza intensiva en cómputo de las simulaciones, permitiendo la generación de resultados estadísticamente robustos.

5. Apéndice

5.1. Uso de herramientas externas: Todoist

Para la realización de este proyecto se utilizo Todoist para coordinar y gestionar el trabajo entre los miembros del grupo. Se adjunta evidencia gráfica en la figura 7.

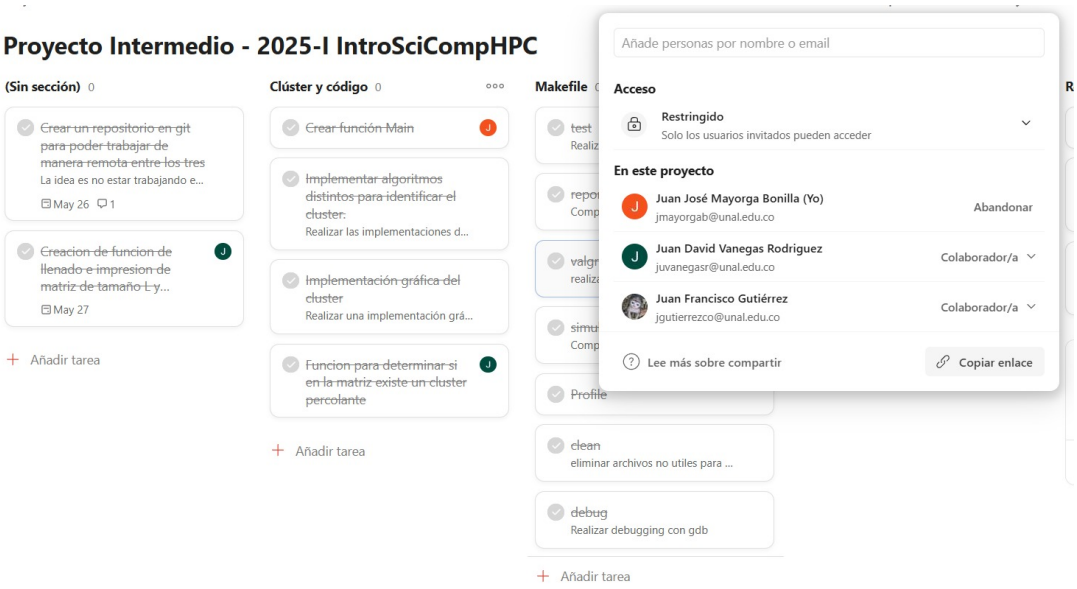


Figura 7: Gestión de tareas en Todoist.

Referencias

[1] A. Cantabrana Barrio. *Estudio de la percolación en redes complejas mediante simulaciones de Monte Carlo*. Tfg, Universidad del País Vasco/Euskal Herriko Unibertsitatea, 2018.

[2] Hoshen-Kopelman Algorithm Overview. Hoshen-kopelman algorithm overview. https://ontosight.ai/glossary/term/hoshen-kopelman-algorithm-overview--67a179026c3593987a57a236, n.d. Recuperado de https://ontosight.ai/glossary/term/hoshen-kopelman-algorithm-overview--67a179026c3593987a57a236.

[3] J. Mayorga, J. Gutierrez, and J. Vanegas. Percolation_2025_i_introscicomphpc. https://github.com/JJMayorgaB/Percolation_2025_I_IntroSciCompHPC, 2025.

[4] PER-Central. Percolation. https://www.per-central.org/wiki/File%3A2155, n.d. Recuperado de https://www.per-central.org/wiki/File%3A2155.