Proyecto Intermedio

M. Gallego,[*] J. Segura,[**] and A. Silva[***]
Introducción a la Computación Científica y de Alto Rendimiento [2015174]
Universidad Nacional de Colombia
Departamento de Física - Bogotá D.C
(Dated: 10 de junio de 2025)

Resumen: En este trabajo se

Palabras clave: Percolación.

I. INTRODUCCIÓN

El fenómeno de percolación es un modelo fundamental en física estadística y teoría de redes, con aplicaciones que abarcan desde la propagación de incendios forestales y epidemias hasta la conductividad de materiales porosos. En su forma más simple, el modelo de percolación representa un sistema en el cual cada sitio (o enlace) de una red es ocupado con cierta probabilidad p, y se estudia la formación de caminos conectados a través del sistema.

En este trabajo se considera el modelo de percolación por sitio en una malla bidimensional cuadrada de tamaño $L \times L$, donde cada celda se ocupa de forma independiente con probabilidad p. Dos celdas ocupadas se consideran conectadas si son vecinas en la dirección de von Neumann (norte, sur, este u oeste). A partir de estas conexiones se forman clusters o componentes conexas de celdas ocupadas.

Un aspecto central del modelo es la transición de fase que ocurre en el límite termodinámico ($L \to \infty$): existe un valor crítico de p, denotado p_c, por debajo del cual la probabilidad de que exista un cluster que atraviese el sistema es prácticamente nula, y por encima del cual la percolación ocurre casi con certeza. En este proyecto, se estudian tres aspectos fundamentales del comportamiento del sistema en función de p y del tamaño L:

1. La probabilidad de percolación, es decir la fracción de configuraciones en las que existe un cluster que conecta lados opuestos de la malla, ya sea en dirección vertical (de arriba a abajo) o horizontal (de izquierda a derecha).

2. El tamaño promedio del cluster más grande para en función de la probabilidad p para cada L.

3. El efecto que tiene el nivel de optimización del compilador `-O` en el tiempo de computo en función del tamaño del sistema L.

Dado el carácter probabilístico del modelo, las simulaciones se repiten ?? para cada valor de p, de L, y `-O` con el fin de obtener estimaciones precisas de las cantidades de interés. A partir de estas repeticiones se calculan promedios y desviaciones estándar, lo que permite evaluar la estabilidad y confiabilidad de los resultados.

Desde el punto de vista computacional, el código fue implementado en C++ siguiendo una estructura modular. Se utilizó un Makefile para automatizar tanto la ejecución del programa como el proceso de análisis del desempeño. Este análisis incluyó tareas como depuración, pruebas unitarias, medición de cobertura de código y perfilamiento, mediante *targets* específicos, los cuales se describen en detalle en la sección de metodología.

II. METODOLOGÍA

A. Código base

El programa se estructuró de manera modular en tres archivos principales:

- `functions.h`: contiene las declaraciones de funciones y las inclusiones de librerías necesarias para el resto del programa.

- `functions.cpp`: implementa las funciones que permiten analizar la malla, entre ellas las funciones principales `hay_cluster_percolante()` y `dfs()`, encargadas de identificar y etiquetar los clústeres, así como de detectar si alguno de ellos es percolante. Estas funciones se describen con mayor detalle más adelante.

- `main.cpp`: actúa como punto de entrada del programa. Recibe por línea de comandos los parámetros `L` (dimensión de la malla), `p` (probabilidad de ocupación) y `seed` (semilla aleatoria). Con estos valores, genera una malla binaria aleatoria de tamaño representada en un vector unidimensional de tipo `bool` y longitud `LxL`. También inicializa un vector `etiquetas` con ceros, del mismo tamaño, para registrar la pertenencia de cada sitio a un clúster, una variable `tamano_max` y un vector `percolantes` vacío que almacenará las etiquetas de los clústeres que percolan tras el llamado a la función `hay_cluster_percolante()`. Durante todo este proceso mide el tiempo de ejecución (tanto *CPU time* como *wall time*) e imprime finalmente

[*] mibarriosg@unal.edu.co
[**] jusegurag@unal.edu.co
[***] ansilvav@unal.edu.co

por consola si hay percolación, el tamaño del clúster percolante más grande y los tiempos registrados.

Ahora, procedemos a explicar la lógica de las dos funciones principales en las que recae el programa:

La función `hay_cluster_percolante()` recibe como argumento el vector de la `malla`, el tamaño `L`, la variable `tamano_max`, el vector `etiquetas` y el vector `percolantes`. Esta recorre los sitios ocupados de la primera fila y la primera columna de la malla de tamaño `LxL`. Si alguno de estos sitios aún no ha sido etiquetado (es decir, si su valor en el vector `etiquetas` es cero), se lanza una búsqueda en profundidad llamando a la función `dfs()`.

La función `dfs()` toma como entrada la posición inicial del sitio (`id`), el tamaño de la malla (`L`), una etiqueta entera única (`etiqueta`) para identificar el clúster actual, el vector binario de la malla (`malla`), un vector de etiquetas para marcar los sitios ya visitados (`etiquetas`), y cuatro referencias booleanas (`toca_arriba`, `toca_abajo`, `toca_izquierda`, `toca_derecha`) que se actualizan si el clúster llega a los bordes respectivos. La función `dfs()` explora todos los sitios conectados al sitio inicial (`id`) que están ocupados y aún no etiquetados, asignándoles la etiqueta correspondiente y devolviendo el tamaño total del clúster encontrado.

De vuelta en `hay_cluster_percolante()`, si el clúster encontrado cumple la condición de tocar simultáneamente los bordes superior e inferior (`toca_arriba && toca_abajo`) o izquierdo y derecho (`toca_izquierda && toca_derecha`), se considera un clúster percolante. En tal caso, se actualiza la variable `tamano_max` si el nuevo clúster es mayor a los previos, y se agrega su etiqueta al vector `percolantes`.

Opcionalmente, `main.cpp` puede llamar a la función `imprimir_clusters()`, que exporta la malla etiquetada a un archivo `.txt`. En este archivo, los sitios no ocupados se marcan con `0`, los ocupados que no forman parte de clústeres percolantes con `1`, y los clústeres percolantes con etiquetas enteras crecientes a partir de `2`. Esta salida se utiliza para visualizar gráficamente los clústeres percolantes como se muestra en la Figura **??**.

Con esta estructura base del código, se procedió a realizar simulaciones sistemáticas. Para cada combinación de los parámetros L, p y nivel de optimización, el programa se ejecutó 10 veces utilizando diferentes semillas aleatorias. Se consideraron cinco tamaños de malla: $L = 32, 64, 128, 256, 512$; treinta valores de p distribuidos entre 0 y 1, con mayor densidad (10 valores) en la región crítica entre 0.55 y 0.65; y cinco niveles de optimización del compilador: `-O0`, `-O1`, `-O2`, `-O3` y `-Ofast`.

B. Automatización con Makefile

Para automatizar las distintas tareas del proyecto, se implementó un archivo `Makefile` con múltiples *targets*, cada uno diseñado para realizar una tarea específica relacionada con la compilación, ejecución, análisis y validación del código. A continuación se describe la funcionalidad de cada uno de estos comandos y su uso desde la línea de comandos:

- `make analisis`: compila y ejecuta el código para todas las combinaciones de L, p y niveles de optimización especificadas anteriormente, utilizando 50 semillas distintas. Los datos generados se procesan mediante un script en Python, que produce las gráficas en formato PDF presentadas en la sección de resultados.

- `make simul L=... p=... seed=...`: compila y ejecuta una simulación individual con los valores de L, p y semilla aleatoria especificados. Posteriormente, genera una visualización del clúster percolante.

- `make test`: ejecuta pruebas unitarias utilizando la librería Catch2 sobre cuatro casos base: una malla completamente vacía ($p = 0$), una malla completamente ocupada ($p = 1$), una malla con una línea horizontal de unos (para verificar percolación horizontal) y una con una línea vertical (para verificar percolación vertical). Estas pruebas comprueban la detección correcta de clústeres percolantes y el cálculo de sus tamaños.

- `make debug`: compila el programa con banderas de depuración y lo ejecuta bajo GDB para facilitar el análisis detallado de errores.

- `make valgrind`: compila el programa con banderas de depuración y ejecuta Valgrind para detectar pérdidas de memoria o accesos inválidos en un caso de prueba.

- `make profile`: genera un informe de *flat profiling* usando la herramienta `perf`, tanto para la versión optimizada como para una versión previa no optimizada del código, para el caso de probabilidad crítica ($L = 128$, $p = 0.59271$, `seed` $= 10$). Esta comparación se analiza en detalle en la sección de dificultades y optimizaciones.

- `make coverage`: ejecuta los casos de prueba y genera un reporte en formato HTML que muestra la cobertura del código utilizando `gcovr`.

- `make clean`: elimina archivos binarios, intermedios y temporales, incluyendo ejecutables, archivos `.gcda`, `.gcno`, `.txt`, `.pdf` y carpetas como `resultados` o `coverage`.

- `make report`: crea este informe a partir de un archivo latex y las figuras en fomato PDF en el repositorio.

III. RESULTADOS

A. Probabilidad de percolación

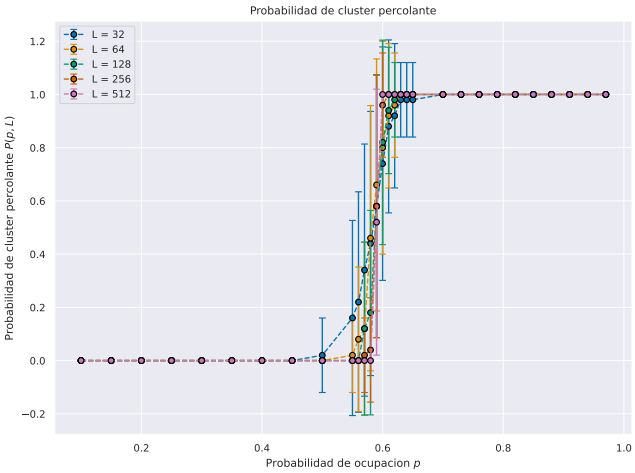

Figura 1: Caption

B. Tamaño promedio del cluster más grande

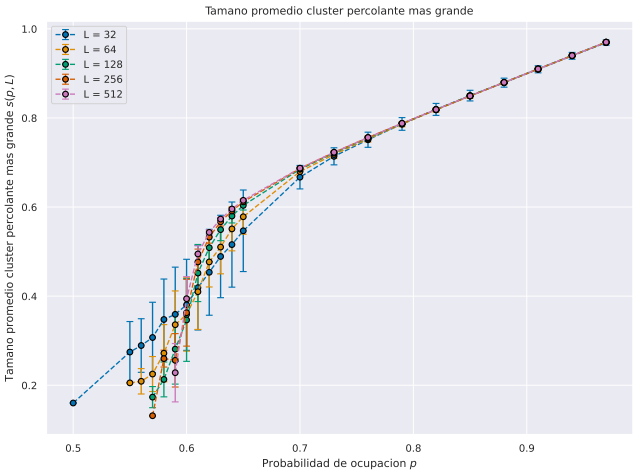

Figura 2: Caption

C. Tiempo de computo para cada optimización

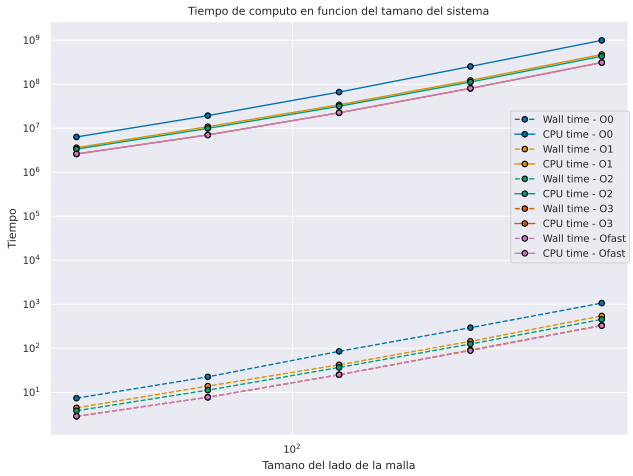

Figura 3: Caption

D. Dificultades y optimizaciones

Una de las principales dificultades encontradas durante el desarrollo del proyecto fue la implementación inicial de la función `dfs()`, la cual realizaba llamadas recursivas para explorar los vecinos de un sitio ocupado. Esta aproximación recursiva generaba un desbordamiento de pila (*stack overflow*) en mallas grandes, como aquellas con $L = 512$, impidiendo que se completara la detección de clústeres percolantes y el cálculo de su tamaño.

La solución consistió en reemplazar la recursividad por una implementación iterativa utilizando un objeto `stack` de la biblioteca estándar `<stack>`. Esta estructura permitió gestionar explícitamente el recorrido por vecinos: se elimina el sitio actual de la pila y se agregan únicamente los vecinos ocupados y no etiquetados. Cuando la pila se vacía, se retorna el tamaño del clúster. Esta versión iterativa evita el desbordamiento y mejora el control del flujo de ejecución. La lógica completa está documentada mediante comentarios en `functions.cpp`.

Una vez resuelto este problema, se identificó una segunda optimización importante. La versión original de `hay_cluster_percolante()` evaluaba la conectividad para toda la malla ($L \times L$), lo cual era innecesariamente costoso para el objetivo de detectar si existe al menos un clúster percolante. Se observó que basta con explorar únicamente los sitios ocupados en la primera fila y la primera columna, ya que cualquier clúster percolante debe estar conectado a uno de estos bordes. Esta optimización redujo significativamente el tiempo de ejecución.

La mejora lograda puede observarse en los reportes de *profiling* obtenidos con `perf` mostrados en la Figura **??**, comparando dos versiones del código: una optimizada que solo recorre la primera fila y columna (`functions.cpp`), y otra sin optimización que recorre toda la malla (`functions_viejo.cpp`). El caso analizado

corresponde a los parámetros $L = 128$, $p = 0.59271$ y `seed` $= 10$.

IV. CONCLUSIONES