Percolation Project

Anaya Lizarazo, Bryan Johan; Pinilla Correa, Santiago & Torres Otálora, Erick

11 de junio de 2025

1. Introducción

La percolación es un fenómeno físico y matemático que describe el comportamiento colectivo de elementos conectados en una red, lo cual puede servir para simular el movimiento de fluidos en medios porosos. En este proyecto se estudia el modelo de percolación porosa por sitios sobre una red cuadrada bidimensional de tamaño $L \times L$. Este modelo consiste en recorrer cada uno de los sitios de la matriz y ocuparlo con una probabilidad p. A partir de esta ocupación aleatoria, se identifican los conglomerados de sitios vecinos ocupados, conocidos como *clusters*.

Para el reconocimiento eficiente de estos clusters se implementó el algoritmo de Hoshen-Kopelman, que permite etiquetar de manera rápida y sin redundancias los distintos conglomerados conectados de la matriz. Con esta información es posible determinar si existe un cluster que conecta los bordes opuestos de la red, es decir, si el sistema *percola*.

En la figura 1 se ilustran dos representaciones de una red: una antes de identificar los clusters en donde los sitios ocupados se representan con 1 y otra después de aplicar el algoritmo de Hoshen-Kopelman, donde cada cluster ha sido etiquetado y representado con un color diferente. Re realiza la simulación con los parametros $L = 4$ y $p = 0{,}6$ (target simul). En el límite termodinámico ($L \to \infty$), se define una probabilidad crítica p_c.

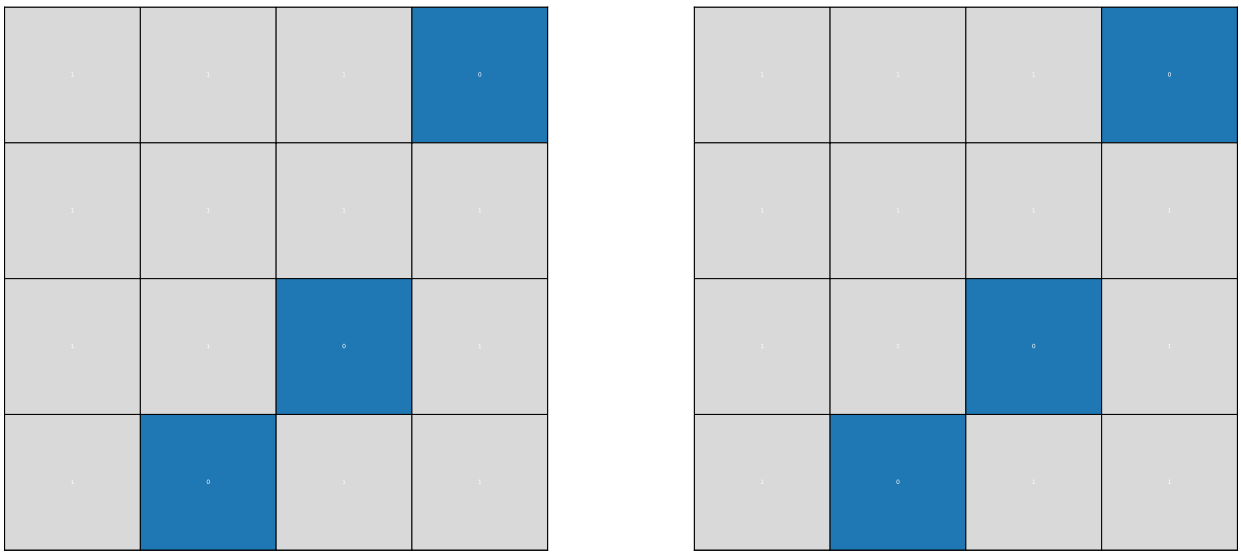

Figura 1: (Izquierda) Lattice de tamaño $L \times L = 16$ con ocupación aleatoria según probabilidad $p = 0{,}6$. (Derecha) Identificación de clusters mediante el algoritmo de Hoshen-Kopelman.

Por debajo de este umbral ($p < p_c$), no existe clúster percolante de tamaño macroscópico, mientras que para $p \geq p_c$ aparece con probabilidad uno un clúster de tamaño infinito que conecta la red.

El objetivo de este artículo es estudiar el comportamiento de la probabilidad de percolación y del tamaño del cluster percolante en función de los parámetros L y p, con el fin de comprobar las predicciones teóricas de la teoría de percolación. Además, se analizará la eficiencia del código implementado para distintos niveles

de optimización, empleando herramientas de profiling como `gprof` y `perf`, con el propósito de identificar posibles optimizaciones y mejorar el rendimiento computacional en la simulación de redes percolates.

2. Resultados de Percolación

A continuación se presentan los resultados numéricos obtenidos para la probabilidad de percolación y el tamaño promedio del clúster percolante más grande (normalizado con el tamaño del sistema $L \times L$) en función de la probabilidad de llenado p y para distintos tamaños de red $L = 16, 32, 64, 128, 256, 512$. Dado que se trata de una probabilidad y de un tamaño promedio, se ejecutó el programa 10 veces para poder realizar la estadística sobre estos resultados.

En la Figura 2 se muestra la probabilidad de percolación $P(L, p)$ frente a p. Cada curva corresponde a un tamaño L distinto. Se observa que, para valores bajos de p, la probabilidad de que exista un clúster que conecte bordes opuestos es cero, mientras que al incrementarse p la probabilidad crece en el intervalo $p \in (0{,}5, 0{,}6)$ para despues ser uno. Además, a medida que L aumenta, la transición se vuelve más pronunciada y la curva se desplaza hacia el valor crítico $p_c \approx 0{,}6$. Este comportamiento es consistente con la teoría de percolación en sistemas finitos, pues la transición es abrupta con L grande, acercándose a una función escalón en el límite termodinámico.

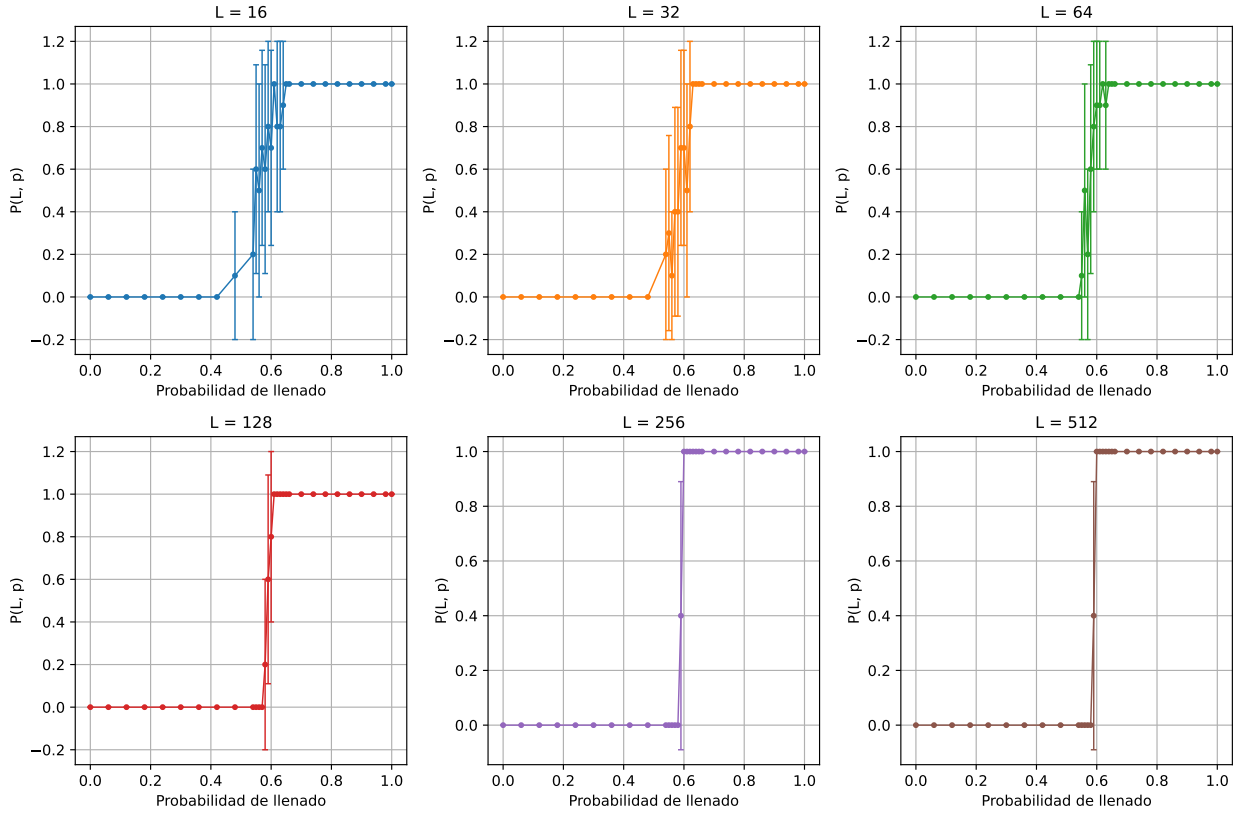

Figura 2: Probabilidad de percolación $P(L, p)$ frente a la probabilidad de llenado p, para redes cuadradas de tamaño $L = 16, 32, 64, 128, 256, 512$. Se observa una transición en la probabilidad de percolación que se hace más pronunciada al aumentar L.

Por otro lado, en la Figura 3 se grafica el tamaño promedio del clúster percolante más grande normalizado en función de p. No hay casi datos para p por debajo del umbral crítico p_c dado que en estos casos es muy poco probable que se presente percolación (no hay clúster macroscópico). Para los valores de p tomados en cuenta, se observa un crecimiento pronunciado cerca de p_c y posteriormente un crecimiento lineal de $S(L, p)$ hasta llegar al tamaño del sistema con $p = 1$ como es esperado. Igualmente, el punto de inflexión en donde

comienza un crecimiento lineal se da aproximadamente para un tamaño de aproximadamente el 60 % del sistema para todos los valores de L. Sin embargo, conforme L crece la varianza en los datos disminuye y se obtiene una curva suave, acercandose al caso ideal del limite termodinámico.

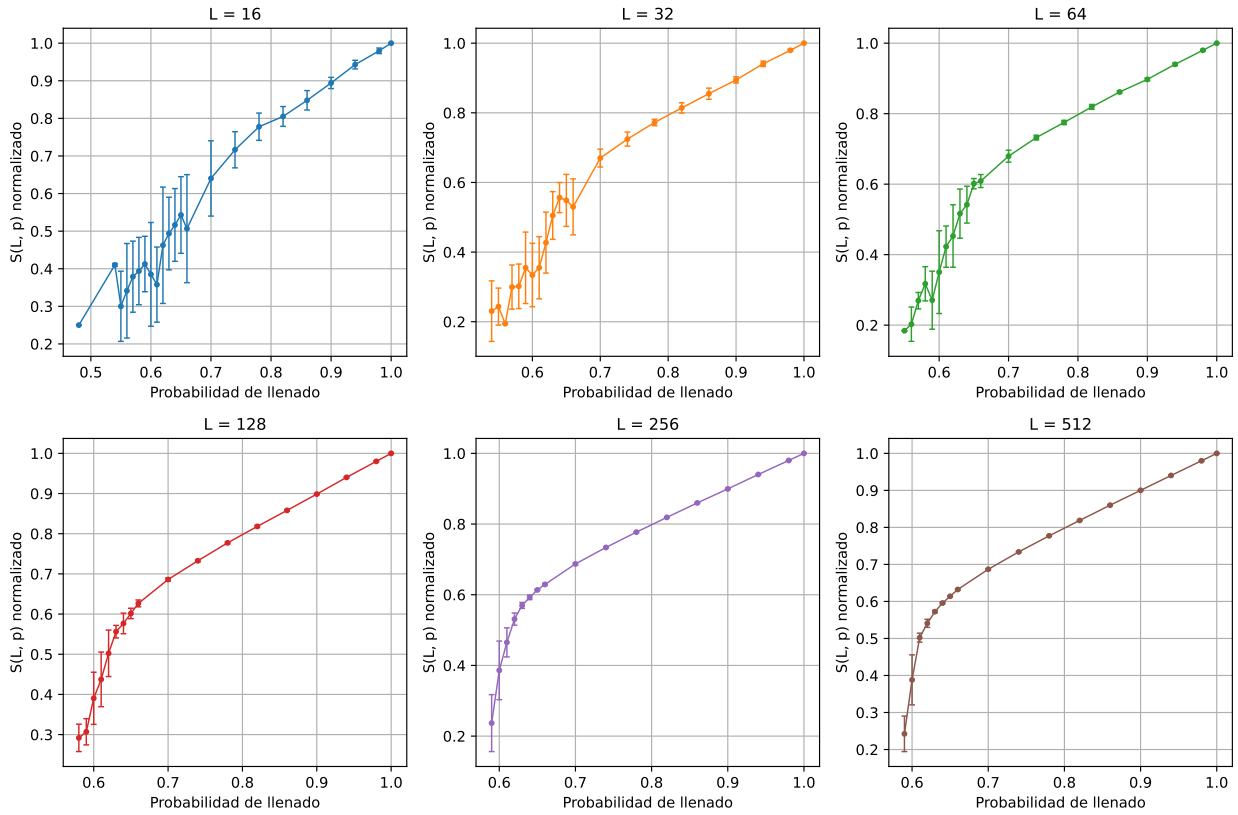

Figura 3: Tamaño promedio del clúster percolante más grande normalizado, $S(L,p) =$ (tamaño del clúster mayor)$/L^2$, frente a la probabilidad de llenado p, para redes cuadradas de tamaño $L = 16, 32, 64, 128, 256, 512$. Se aprecia el crecimiento pronunciado de S alrededor del umbral crítico.

3. Análisis profiling

3.1. Niveles de optimización

Para evaluar el impacto de los distintos niveles de optimización del compilador en el rendimiento de nuestro programa de percolación, compilamos el código con las banderas `-O0`, `-O1`, `-O2`, `-O3` y `-Ofast`. A continuación, la Figura 4 muestra el tiempo de ejecución medido (en milisegundos) para diferentes tamaños de red L y la probabilidad de llenado $p = 0{,}6$ cercana a la probabilidad crítica.

En general, se observa que el tiempo de ejecución es muy corto incluso para un tamaño del sistema $L \approx 2000$ y una optimización `-O0`. Además, el tiempo de ejecución disminuye significativamente al aplicar optimizaciones, siendo `-O3` y `-Ofast` las más efectivas, reduciendo el tiempo de ejecución a menos de 1 segundo para $L = 2000$. La optimización `-Ofast` es la más rápida, pero puede introducir cambios en el comportamiento del programa al desactivar ciertas verificaciones de precisión. Por otro lado, la optimización *-O0* muestra un rendimiento significativamente peor, lo que indica que las optimizaciones del compilador son cruciales para mejorar la eficiencia del código, especialmente en algoritmos que requieren múltiples iteraciones y operaciones sobre estructuras de datos como matrices y vectores. Vemos también que, para tamaños pequeños, la diferencia entre los niveles de optimización es imperceptible, pero a medida que L aumenta, las optimizaciones tienen un impacto más significativo en el tiempo de ejecución.

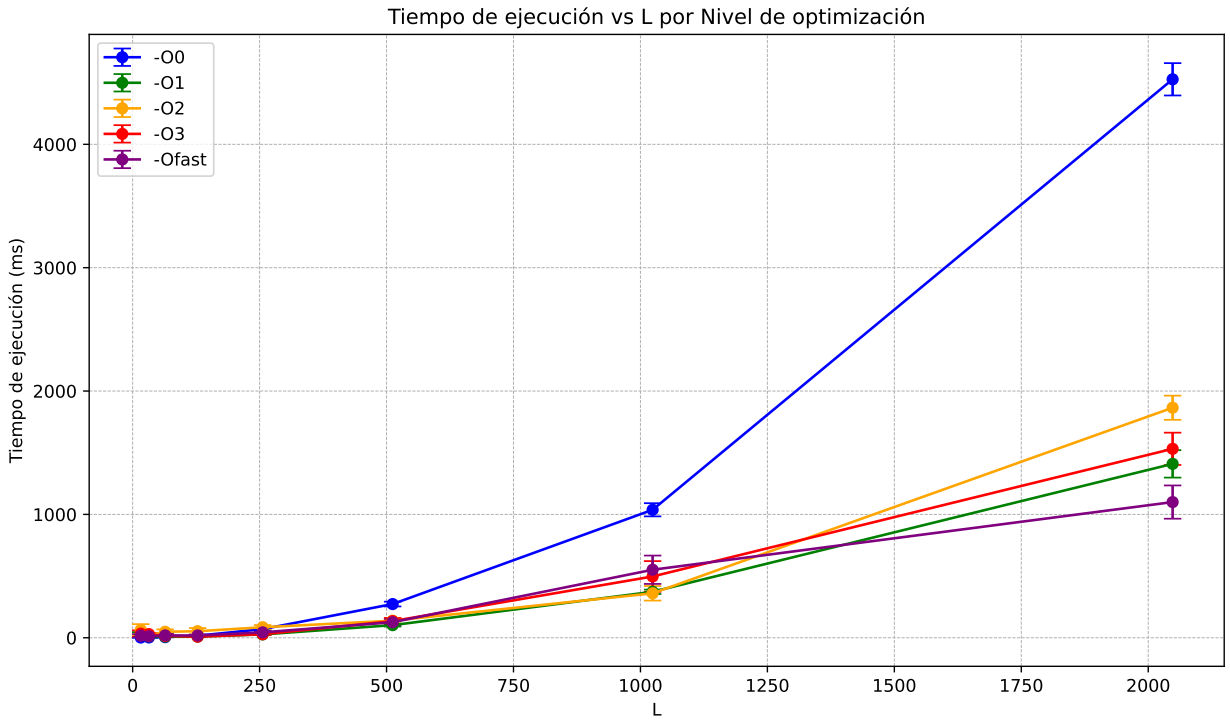

Figura 4: Tiempo de ejecución del programa de percolación en función del tamaño de la red L, compilado con distintos niveles de optimización: `-O0` (sin optimizar), `-O1`, `-O2`, `-O3` y `-Ofast`.

3.2. Informe profiling

Se realizó un informe detallado de *profiling* utilizando `gprof`. Sin embargo, debido al uso de librerías sobre las cuales no se tiene control, resultaba difícil obtener información precisa sobre el comportamiento interno del programa. Por esta razón, se filtró el reporte para mostrar únicamente las funciones definidas explícitamente en el código del proyecto. Este informe filtrado se incluye en el Anexo A.1. Como se puede observar, las funciones creadas corresponden a menos del 10 % del tiempo total de ejecución. Por lo tanto, se concluye que el algoritmo implementado es sumamente eficiente y que la mayor parte del tiempo se consume en llamadas a funciones de la biblioteca estándar de `C++`.

Adicionalmente, se generó un reporte plano con `perf`, el cual proporciona información complementaria y una mayor resolución temporal en comparación con `gprof`. A partir de este reporte, se generó un *flamegraph* que permite visualizar jerárquicamente las llamadas a funciones y el tiempo acumulado en cada una. En la Figura 5 se presenta dicho *flamegraph*, donde el largo de cada banda corresponde al tiempo de CPU consumido por esa función y sus subllamadas. Esta herramienta resulta particularmente útil para identificar visualmente los puntos más costosos en términos de tiempo de ejecución. El reporte plano usado como base se incluye en el Anexo A.2.

Como se observa en la Figura 5, las funciones que concentran el mayor uso de CPU son `find_clusters` y `print`, seguidas por `HoshenKopelman` y `fill_lattice`. La función `find_clusters` es responsable de identificar los clústeres en la red, mientras que `print` se encarga de mostrar los resultados. Por otro lado, `HoshenKopelman` es el algoritmo que etiqueta los clústeres y `fill_lattice` llena la red con ocupación aleatoria. Estas funciones son críticas para el rendimiento del programa, ya que representan las operaciones más costosas en términos de tiempo de ejecución. No obstante, y como ya mencionó anteriormente, al analizar los reportes planos de `perf` y `gprof`, se evidencia que gran parte del tiempo dentro de estas funciones se consume en llamadas a funciones de la biblioteca estándar, como operaciones de inserción y recorrido de estructuras `std::map` o el uso de `std::ostream`, lo cual sugiere que las estructuras y métodos utilizados internamente también tienen un impacto significativo en el rendimiento general del programa, lo que limita

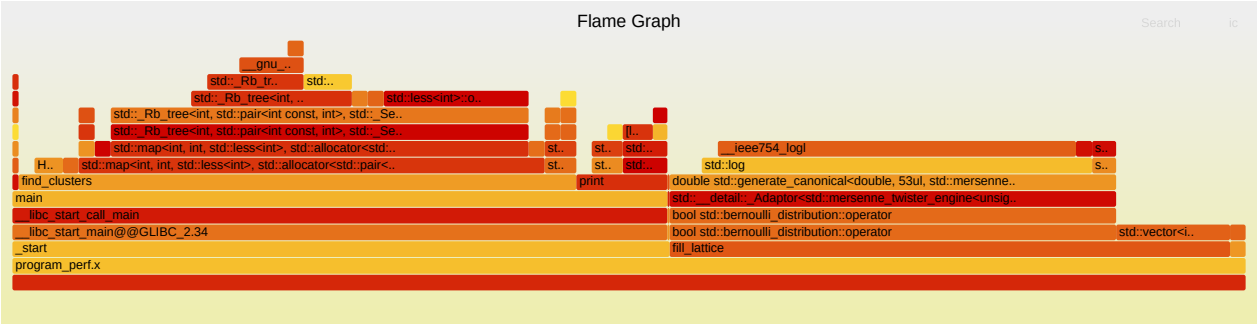

Figura 5: Flamegraph del perfil de ejecución generado con `perf`.

las oportunidades de optimización a nivel de código fuente.

4. Optimizaciones realizadas

El análisis del reporte de profiling mostró que las funciones definidas en nuestro código consumen un porcentaje muy bajo del tiempo total, por lo que no se identificaron optimizaciones significativas a través de gprof. Sin embargo, al examinar las gráficas de tiempos de ejecución con distintos niveles de compilación y revisar detenidamente el código, detectamos algunas mejoras puntuales: en lugar de inicializar el vector de etiquetas con un tamaño fijo de $L^2/2$ y llenarlo, resultó más eficiente usar $labels.push_back()$ dado que el número real de etiquetas es generalmente mucho menor que $L^2/2$. Asimismo, la obtención de $unique_ids$ se cambió de operar sobre la matriz completa a utilizar directamente el vector `labels`, más pequeño y suficiente para nuestro propósito. Finalmente, en la función de `Find` se implementó la rutina de compresión de camino, permitiendo que en llamadas porteriores se encontrara el cluster padre un poco más rapido, esto es especialmente util dado que es la función que más veces se llama en el codigo cómo lo demuestra el reporte de`gprof` . El código mencionado se observa en el Listing 1.

Listing 1: Codigo de optimizaciones

```
// ------------ Funcion HoshenKopelman: ------------
//   Inserta nueva etiqueta en el vector de labels
labels.push_back(next_label);

// ------------ Funcion find_clusters: ------------
//   Crea un unique_ids con vec labels en lugar de lattice
std::set<int> unique_ids(labels.begin(), labels.end());

// ------------ Funcion Find: ------------
//   Compresion de camino en union-find
while (parent[ii] != ii) {
    kk = parent[ii];
    parent[ii] = jj;
    ii = kk;
}
```

5. Conclusiones

El proyecto de percolación ha permitido explorar y comprender el fenómeno de percolación en redes bidimensionales, implementando el algoritmo de Hoshen-Kopelman para identificar clústeres y determinar la probabilidad de percolación. Los resultados obtenidos muestran una transición clara en la probabilidad de percolación a medida que se varía la probabilidad de llenado p, corroborando las predicciones teóricas del modelo. Además, se ha realizado un análisis exhaustivo del rendimiento del código mediante herramientas de profiling como `gprof` y `perf`, lo que ha permitido identificar las funciones más críticas en términos de tiempo de ejecución. Aunque el código original ya era eficiente, se implementaron algunas optimizaciones menores que

mejoraron ligeramente el rendimiento, especialmente en la gestión de estructuras de datos. El uso de distintos niveles de optimización del compilador ha demostrado ser crucial para mejorar el rendimiento del programa, especialmente en tamaños de red grandes. Las optimizaciones más agresivas (`-O3` y `-Ofast`) lograron reducir significativamente el tiempo de ejecución, aunque con un pequeño riesgo de alterar el comportamiento del programa. Finalmente, se observó que gran parte del tiempo de CPU se concentra en operaciones internas de funciones estándar de C++, como manejo de mapas y flujo de salida, más que en la lógica explícita del algoritmo implementado.

A. Anexos

A.1. Flat profile de `gprof` filtrado

A continuación se presenta el perfil plano (flat profile) generado por `gprof`, filtrado para mostrar únicamente las funciones definidas en el código del proyecto (se han eliminado entradas de librerías del sistema).

Listing 2: Flat profile filtrado de gprof

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
  4.17      0.38     0.02        1    20.00   365.24  find_clusters(std::vector<int, std::
      allocator<int> >&)
  0.00      0.48     0.00        1     0.00     0.07  detec_perc(std::vector<int, std::
      allocator<int> > const&)
  0.00      0.48     0.00        1     0.00    22.98  HoshenKopelman(std::vector<int, std::
      allocator<int> >&)
  0.00      0.48     0.00        2     0.00     4.98  print(std::vector<int, std::allocator<
      int> > const&, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator
      <char> > const&)
  0.00      0.48     0.00   214776     0.00     0.00  Union(std::vector<int, std::allocator<
      int> >&, int, int)
  0.00      0.48     0.00  1029117     0.00     0.00  Find(std::vector<int, std::allocator<
      int> >&, int)

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.


Copyright (C) 2012-2024 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.


                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 1.72% of 0.58 seconds
```

```
index % time    self  children    called     name
                                                 <spontaneous>
           the gprof listing if it were to be printed.
                0.00    0.01       2/2           print(std::vector<int, std::allocator<int>
                    > const&, std::__cxx11::basic_string<char, std::char_traits<char>, std::
                    allocator<char> > const&) [37]
                0.01    0.00 2000000/2006447     print(std::vector<int, std::allocator<int>
                    > const&, std::__cxx11::basic_string<char, std::char_traits<char>, std::
                    allocator<char> > const&) [37]
[37]     2.1    0.00    0.01       2         print(std::vector<int, std::allocator<int> >
    const&, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >
    const&) [37]
                0.00    0.00       2/74          print(std::vector<int, std::allocator<int>
                    > const&, std::__cxx11::basic_string<char, std::char_traits<char>, std::
                    allocator<char> > const&) [37]
                The index number is printed next to every function name so
                printed after it.  If the function is a member of a
                cycle, the cycle number is printed between the
                number is printed after it.  If the parent is a
                member of a cycle, the cycle number is printed between
 '<spontaneous>' is printed in the 'name' field, and all the other
                number is printed after it.  If the child is a
                member of a cycle, the cycle number is printed
  [37] print(std::vector<int, std::allocator<int> > const&, std::__cxx11::basic_string<char,
       std::char_traits<char>, std::allocator<char> > const&) [208] std::
      _Rb_tree_key_compare<std::less<int> >::_Rb_tree_key_compare() [150] std::_Rb_tree<int,
       int, std::_Identity<int>, std::less<int>, std::allocator<int> >::_M_put_node(std::
      _Rb_tree_node<int>*)
                0.00    0.01  599565/1029117     Find(std::vector<int, std::allocator<int>
                    >&, int) [19]
                0.02    0.00 4504334/8516570     Find(std::vector<int, std::allocator<int>
                    >&, int) [19]
[19]     4.4    0.00    0.02 1029117         Find(std::vector<int, std::allocator<int> >&,
    int) [19]
                0.00    0.01  429552/1029117     Find(std::vector<int, std::allocator<int>
                    >&, int) [19]
  [19] Find(std::vector<int, std::allocator<int> >&, int) [96] std::_Rb_tree_iterator<int>::
      _Rb_tree_iterator(std::_Rb_tree_node_base*) [175] std::_Rb_tree<int, int, std::
      _Identity<int>, std::less<int>, std::allocator<int> >::_Alloc_node::_Alloc_node(std::
      _Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >&)
                0.00    0.00  214776/8516570     Union(std::vector<int, std::allocator<int>
                    >&, int, int) [38]
                0.00    0.01  214776/214776      Union(std::vector<int, std::allocator<int>
                    >&, int, int) [38]
                0.00    0.01  429552/1029117     Union(std::vector<int, std::allocator<int>
                    >&, int, int) [38]
[38]     2.0    0.00    0.01  214776         Union(std::vector<int, std::allocator<int> >&,
    int, int) [38]
                0.00    0.00  214776/214776      Union(std::vector<int, std::allocator<int>
                    >&, int, int) [38]
                0.00    0.00  214776/214776      Union(std::vector<int, std::allocator<int>
                    >&, int, int) [38]
  [38] Union(std::vector<int, std::allocator<int> >&, int, int) [158] std::_Rb_tree_iterator
      <int>::operator--() [149] std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>,
       std::allocator<int> >::_M_get_node()
                0.00    0.02       1/1           HoshenKopelman(std::vector<int, std::
                    allocator<int> >&) [18]
                0.01    0.00 2797458/8516570     HoshenKopelman(std::vector<int, std::
                    allocator<int> >&) [18]
[18]     4.8    0.00    0.02       1         HoshenKopelman(std::vector<int, std::allocator<
    int> >&) [18]
                0.00    0.01  214776/214776      HoshenKopelman(std::vector<int, std::
                    allocator<int> >&) [18]
                0.00    0.00   96537/96537       HoshenKopelman(std::vector<int, std::
                    allocator<int> >&) [18]
                0.00    0.00       1/74          HoshenKopelman(std::vector<int, std::
                    allocator<int> >&) [18]
```

```
                0.00    0.00       1/9           HoshenKopelman(std::vector<int, std::
                    allocator<int> >&) [18]
                0.00    0.00       1/2           HoshenKopelman(std::vector<int, std::
                    allocator<int> >&) [18]
  [18] HoshenKopelman(std::vector<int, std::allocator<int> >&) [70] std::_Rb_tree_iterator<
      std::pair<int const, int> >::_Rb_tree_iterator(std::_Rb_tree_node_base*) [25] std::
      _Rb_tree_iterator<int> std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>,
      std::allocator<int> >::_M_insert_<int&, std::_Rb_tree<int, int, std::_Identity<int>,
      std::less<int>, std::allocator<int> >::_Alloc_node>(std::_Rb_tree_node_base*, std::
      _Rb_tree_node_base*, int&, std::_Rb_tree<int, int, std::_Identity<int>, std::less<int
      >, std::allocator<int> >::_Alloc_node&)
                0.02    0.35       1/1           find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
[2]     76.1    0.02    0.35       1         find_clusters(std::vector<int, std::allocator<
    int> >&) [2]
                0.00    0.29 2025682/2025684     find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
                0.00    0.02       1/1           find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
                0.00    0.01  599565/1029117     find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
                0.00    0.01       1/1           find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
                0.01    0.00 2000000/2096539     find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
                0.00    0.00       1/3           find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
                0.00    0.00 2000002/2096543     find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
                0.00    0.00 2000000/2096539     find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
                0.00    0.00   25683/25916       find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
                0.00    0.00   25682/25911       find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
                0.00    0.00   25682/25910       find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
                0.00    0.00       3/21          find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
                0.00    0.00       3/21          find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
                0.00    0.00       1/6           find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
                0.00    0.00       1/4           find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
                0.00    0.00       1/4           find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
                0.00    0.00       2/3           find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
                0.00    0.00       1/3           find_clusters(std::vector<int, std::
                    allocator<int> >&) [2]
   [2] find_clusters(std::vector<int, std::allocator<int> >&) [207] std::__new_allocator<int
       >::~__new_allocator() [60] std::_Rb_tree_iterator<int> std::_Rb_tree<int, int, std::
       _Identity<int>, std::less<int>, std::allocator<int> >::_M_insert_<int const&, std::
       _Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::
       _Alloc_node>(std::_Rb_tree_node_base*, std::_Rb_tree_node_base*, int const&, std::
       _Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::
       _Alloc_node&)
                0.00    0.00       1/1           detec_perc(std::vector<int, std::allocator<
                    int> > const&) [48]
                0.00    0.00    6447/2006447     detec_perc(std::vector<int, std::allocator<
                    int> > const&) [48]
[48]     0.0    0.00    0.00       1         detec_perc(std::vector<int, std::allocator<int>
     > const&) [48]
                0.00    0.00    2449/2449        detec_perc(std::vector<int, std::allocator<
                    int> > const&) [48]
                0.00    0.00     227/227         detec_perc(std::vector<int, std::allocator<
                    int> > const&) [48]
```

```
                0.00    0.00     229/25916       detec_perc(std::vector<int, std::allocator<
                    int> > const&) [48]
                0.00    0.00     227/25911       detec_perc(std::vector<int, std::allocator<
                    int> > const&) [48]
                0.00    0.00     227/25910       detec_perc(std::vector<int, std::allocator<
                    int> > const&) [48]
                0.00    0.00       1/74          detec_perc(std::vector<int, std::allocator<
                    int> > const&) [48]
                0.00    0.00       1/9           detec_perc(std::vector<int, std::allocator<
                    int> > const&) [48]
                0.00    0.00       5/6           detec_perc(std::vector<int, std::allocator<
                    int> > const&) [48]
                0.00    0.00       5/5           detec_perc(std::vector<int, std::allocator<
                    int> > const&) [48]
                0.00    0.00       3/4           detec_perc(std::vector<int, std::allocator<
                    int> > const&) [48]
                0.00    0.00       3/4           detec_perc(std::vector<int, std::allocator<
                    int> > const&) [48]
                0.00    0.00       1/1           detec_perc(std::vector<int, std::allocator<
                    int> > const&) [48]
                0.00    0.00       1/1           detec_perc(std::vector<int, std::allocator<
                    int> > const&) [48]
  [48] detec_perc(std::vector<int, std::allocator<int> > const&) [194] std::__new_allocator<
      int>::deallocate(int*, unsigned long) [50] std::_Rb_tree<int, std::pair<int const, int
      >, std::_Select1st<std::pair<int const, int> >, std::less<int>, std::allocator<std::
      pair<int const, int> > >::~_Rb_tree()

# --- Fin del reporte filtrado ---
```

A.2. Flat profile de `perf` filtrado

A continuación se presenta el perfil plano (flat profile) generado por `perf`, filtrado para mostrar únicamente las funciones definidas en el código del proyecto (se han eliminado entradas de librerías del sistema).

Listing 3: Flat profile filtrado de perf

```
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 2K of event 'cycles:Pu'
# Event count (approx.): 2666324979
#
# Overhead  Command          Shared Object        Symbol
# ........  ...............  ...................
    ..........................................................................................

#
     1.76%  program_perf.x  program_perf.x       [.] find_clusters(std::vector<int, std::
         allocator<int> >&)
     1.55%  program_perf.x  program_perf.x       [.] HoshenKopelman(std::vector<int, std::
         allocator<int> >&)
     1.25%  program_perf.x  program_perf.x       [.] Find(std::vector<int, std::allocator<
         int> >&, int)
     0.52%  program_perf.x  program_perf.x       [.] print(std::vector<int, std::allocator<
         int> > const&, std::__cxx11::basic_string<char, std::char_traits<char>, std::
         allocator<char> > const&)
     0.37%  program_perf.x  program_perf.x       [.] Union(std::vector<int, std::allocator<
         int> >&, int, int)

# --- Fin del reporte filtrado (perf) ---
```