

**Pontificia Universidad Javeriana**

Proyecto - Hdispersa y Pdispersa



Tomas Alejandro Silva Correal  
Gabriel Jaramillo Cuberos  
Juan Pabón Vargas  
Juan Felipe Rodriguez

Asignatura: Sistemas Operativos

Profesor: John Jairo Corredor Franco

26 de mayo de 2025

<b>1. Introducción.....</b>	<b>2</b>
<b>2. Planteamiento del problema.....</b>	<b>2</b>
2.1 Problema central:.....	2
2.2 Requisitos específicos:.....	2
2.3 Problemas técnicos:.....	3
<b>3. Propuesta de solución.....</b>	<b>3</b>
3.1 Arquitectura general:.....	3
3.2 Componentes clave:.....	4
3.2.1 Carga de la matriz.....	4
3.2.2 División del trabajo.....	4
3.2.4 Implementación con hilos.....	4
3.2.3 Implementación con procesos.....	4
3.2.5 Cálculo final.....	5
<b>4. Método de prueba.....</b>	<b>5</b>
<b>5. Diseño.....</b>	<b>5</b>
5.1 Flujo de ejecución.....	6
5.2 Decisiones de implementación.....	6
<b>6. Compilación.....</b>	<b>6</b>
<b>7. Ejecución y comandos.....</b>	<b>7</b>
7.1 Ejecución hdispersa.....	7
7.2 Ejecución pdispersa.....	7
7.3 Ejecución generador de matrices.....	7
7.4 Dar permisos de ejecución a lanzador.....	7
7.5 Ejecutar lanzador.....	7
7.6 Makefile.....	7
<b>8. Resultados.....</b>	<b>8</b>
8.1 Pruebas individuales.....	8
8.2 Pruebas de Rendimiento Automatizadas.....	9
<b>9. Análisis de resultados.....</b>	<b>19</b>
<b>10. Consideraciones adicionales.....</b>	<b>20</b>
<b>11. Conclusiones.....</b>	<b>20</b>
<b>12. Resumen.....</b>	<b>21</b>

# 1. Introducción

En álgebra lineal numérica, las matrices dispersas (matrices sparse) son estructuras binarias donde la mayoría de sus elementos son cero. Determinar si una matriz es dispersa es una operación computacionalmente intensiva para matrices grandes, lo que puede llevar al consumo de varios recursos y problemas de rendimiento. Este proyecto aborda el problema

implementando dos soluciones concurrentes: una utilizando procesos creados con `fork()` y otra utilizando hilos POSIX, permitiendo también comparar ambas soluciones.

El objetivo del proyecto es desarrollar un sistema eficiente que determine la dispersión de matrices grandes mediante paralelización, evaluando el porcentaje de ceros en relación a un porcentaje mínimo configurable a través de la consola. Para el proyecto se tuvo en cuenta los requisitos establecidos, como el ingreso de los datos y la comunicación entre procesos.

## 2. Planteamiento del problema

### 2.1 Problema central:

El objetivo principal de este proyecto es evaluar de manera eficiente si matrices de gran tamaño son dispersas. La dispersión de una matriz se refiere al porcentaje de elementos ceros que contiene, y determinar esto de manera precisa y optimizada requiere abordar varios aspectos técnicos, especialmente al trabajar con estructuras de datos de gran volumen.

### 2.2 Requisitos específicos:

Para alcanzar este objetivo, es necesario implementar dos versiones concurrentes del programa: una utilizando procesos y otra utilizando hilos. Estas versiones deberán ser capaces de leer matrices desde archivos en formato texto, lo que implica una carga inicial precisa de los datos. Una vez cargada la matriz, el trabajo deberá ser dividido entre  $n$  trabajadores (workers), quienes se encargarán de analizar distintas porciones de la matriz de forma paralela.

El sistema debe calcular el porcentaje exacto de ceros presentes en la matriz, ya que este valor será utilizado para determinar si la matriz es considerada dispersa. Para ello, el programa debe comparar el resultado con un porcentaje mínimo de dispersión que debe ser configurable por el usuario en la terminal de comandos. Además, será fundamental manejar adecuadamente la comunicación entre procesos para garantizar la correcta recolección de los resultados y una operación sincronizada. También se debe incluir una validación robusta de todos los parámetros de entrada para evitar errores en la ejecución.

## 2.3 Problemas técnicos:

Para el desarrollo del proyecto, se debe establecer una comunicación entre el proceso padre y los procesos hijos, lo cual es fundamental para el análisis distribuido de la matriz. También es importante asegurar una división equitativa del trabajo entre los trabajadores con el objetivo de evitar sobrecargas o tiempos de espera innecesarios.

Los resultados obtenidos por los trabajadores deben combinarse de forma adecuada para obtener un resultado final correcto. Para esto, es importante tener en cuenta los límites de la comunicación entre procesos. Por otra parte, en la versión del programa que trabaja con hilos, se debe tener cuidado con la sincronización para que el acceso a recursos compartidos sea seguro y libre de condiciones de carrera. Finalmente, se debe lograr una carga correcta de matrices, lo cual puede necesitar estructuras de datos para representar las matrices en la memoria.

## 3. Propuesta de solución

### 3.1 Arquitectura general:

El diseño general del sistema se centra en dos programas, cada uno con una estrategia de ejecución paralela: una basada en procesos y otra en hilos. Ambos programas tienen un mismo objetivo, y comparten varias funciones como la lectura de argumentos o los temporizadores, además de la división equitativa de la matriz para los trabajadores. Sin embargo, su ejecución interna es lo que cambia. La arquitectura modular permite comparar y evaluar ambas técnicas de concurrencia dentro de un mismo entorno de ejecución.

### 3.2 Componentes clave:

#### 3.2.1 Carga de la matriz

Uno de los componentes fundamentales del sistema es la carga inicial de la matriz. Esta se realiza a través de la lectura de archivos en formato texto, desde los cuales se extraen los datos numéricos para almacenarlos en una estructura de datos

bidimensional en memoria. Además, también se valida que las dimensiones de la matriz sean correctas.

### 3.2.2 División del trabajo

Una vez cargada la matriz, se procede a dividir el trabajo entre un número determinado de trabajadores. Por ejemplo, si se tienen 100 filas y 3 workers, el sistema podría distribuirlas de la siguiente manera: el Worker 1 se encargará de las filas 0 a 33, el Worker 2 de las filas 34 a 66, y el Worker 3 de las filas 67 a 99. Esta división busca balancear la carga de forma equitativa para aprovechar al máximo los recursos concurrentes. Cuando hay filas sobrantes, cada fila que sobra se le asigna a un trabajador diferente para distribuir las sobras de manera equitativa.

### 3.2.4 Implementación con hilos

En la versión basada en hilos, se utilizan las funciones `pthread_create()` y `pthread_join()` para crear y coordinar los hilos. Los datos necesarios para la ejecución de cada hilo se guardan en una estructura llamada `DatosHilos`, la cual indica a cada hilo con que datos va a trabajar y se pasa como argumento al hilo al momento de su creación. Una vez finalizado su trabajo, cada hilo devuelve el resultado correspondiente mediante `pthread_join()`.

### 3.2.3 Implementación con procesos

La implementación con procesos (usando `fork()`) crea copias independientes del programa, donde cada proceso hijo analiza una porción de la matriz. Como los procesos no comparten memoria, deben usar archivos temporales para comunicar resultados cuando el valor exceda 254. En cambio, la versión con hilos (usando `pthread_create()`) divide el trabajo entre múltiples hilos dentro del mismo proceso, que comparten la memoria. Esto facilita el acceso y acumulación de datos sin archivos intermedios, lo que la hace tener un mejor rendimiento. La principal diferencia es que los procesos no comparten memoria y son más pesados, mientras que los hilos comparten memoria y son más ligeros.

### 3.2.5 Cálculo final

Después de la ejecución de los procesos o hilos y la recolección de los resultados parciales por cada trabajador, el programa principal realiza el cálculo final.

## 4. Método de prueba

Primero, se ingresaran diferentes matrices de diferentes tamaños con los comandos establecidos:

- `./hdispersa -f NumFilas -c NumColumnas -a NombreArchivo -n NumProcesos -p Porcentaje`
- `./pdispersa -f NumFilas -c NumColumnas -a NombreArchivo -n NumProcesos -p Porcentaje`

**Para los datos de entrada, se probarán diferentes combinaciones en diferente orden, con el objetivo de evaluar si los programas leen los datos correctamente.**

Para las pruebas de rendimiento se va a hacer una comparación simultánea de ambos programas. Para ello, se utilizará un programa lanzador llamado `lanza.pl`, el cual se va a encargar de ejecutar ambos programas al mismo tiempo. Cada programa será ejecutado 16 veces en un sistema de cómputo Ubuntu de 16 hilos, y cada vez que se ejecute, se aumentará en 1 la cantidad de trabajadores usados. El programa evaluará el rendimiento en 5 diferentes matrices, cada una con ceros y unos distribuidos aleatoriamente:

- Matriz 1: Matriz de 1000x1000 con un total de 720000 ceros (72% de la matriz) y un porcentaje establecido de 70%.
- Matriz 2: Matriz de 2000x2000 con un total de 2880000 ceros (72% de la matriz) y un porcentaje establecido de 70%.
- Matriz 3: Matriz de 5000x5000 con un total de 18000000 de ceros (72% de la matriz) y un porcentaje establecido de 70%.
- Matriz 4: Matriz de 8000x8000 con un total de 46080000 ceros (72% de la matriz) y un porcentaje establecido de 70%.
- Matriz 5: Matriz de 10000x10000 con un total de 72000000 de ceros (72% de la matriz) y un porcentaje establecido de 70%.

El programa lanzador guardará los tiempos registrados en microsegundos de cada ejecución en programas `.dat`.

## 5. Diseño

### 5.1 Flujo de ejecución

El flujo de ejecución del programa está diseñado para ser eficiente. En primer lugar, se realiza la lectura de los argumentos de línea de comandos, lo que permite configurar aspectos como el modo de ejecución (procesos o hilos), la ruta del archivo de la matriz, el número de trabajadores, el porcentaje de dispersión, entre otros. Después de esto, se evalúan cada uno de los argumentos de entrada con el objetivo de evitar cualquier error de usuario.

Después de la validación de las entradas, el programa carga los datos de la matriz para posteriormente ser evaluada por los trabajadores. Cada programa divide la matriz en subsecciones iguales, y asigna a cada trabajador una sección para evaluar la cantidad de elementos diferentes a cero. Luego, el programa hace el cálculo correspondiente para determinar la cantidad de ceros que hay en toda la matriz, basándose en la cantidad de elementos diferentes a cero obtenidos por los trabajadores.

Finalmente, se muestran los resultados obtenidos en la consola y se libera la memoria utilizada por la matriz y los trabajadores.

## 5.2 Decisiones de implementación

Para la división de las matrices, se optó por dividir las en filas, ya que esto hace más fácil el recorrido de la matriz y el conteo individual de cada elemento.

Por otra parte, se prioriza la seguridad y robustez del programa a través de la validación de todos los parámetros de entrada. Esto incluye asegurar que los valores numéricos estén dentro de rangos permitidos, que los archivos existan y sean accesibles, y que el número de workers sea coherente con el tamaño de la matriz.

## 6. Compilación

### **Hdispersa:**

```
gcc -o hdispersa hdispersa.c UtilsHdispersa/hilos.c UtilsGeneral/matriz.c  
UtilsGeneral/temporizador.c UtilsGeneral/utilidades.c -lpthread -lm
```

### **Pdispersa:**

```
gcc -o pdispersa pdispersa.c UtilsPdispersa/procesos.c UtilsGeneral/matriz.c  
UtilsGeneral/temporizador.c UtilsGeneral/utilidades.c -lpthread -lm
```

### **GeneradorMatrices:**

```
g++ -o generador generadormatrices.cpp
```

## 7. Ejecución y comandos

Para compilar los programas hdispersa y pdispersa, se utilizaron los siguientes comandos de compilación con GCC en un entorno Linux:

## 7.1 Ejecución hdispersa

`./hdispersa -f NumFilas -c NumColumnas -a NombreArchivo -n NumProcesos -p Porcentaje`

## 7.2 Ejecución pdispersa

`./pdispersa -f NumFilas -c NumColumnas -a NombreArchivo -n NumProcesos -p Porcentaje`

## 7.3 Ejecución generador de matrices

`./generador [opcion (1 o 2) (opcional)]`

## 7.4 Dar permisos de ejecución a lanzador

`chmod -x lanza.pl`

## 7.5 Ejecutar lanzador

`./lanza.pl`

## 7.6 Makefile

Para simplificar el proceso, se incluyó un Makefile con los siguientes comandos:

- `make` (Compila los 3 programas)
- `make clean` (Elimina los ejecutables creados y todas las matrices junto con los registros de los tiempos)

# 8. Resultados

## 8.1 Pruebas individuales

Para verificar el correcto funcionamiento de hdispersa y pdispersa, se ejecutaron dos casos de prueba con matrices de diferente tamaño y distribución de ceros.

- **Caso 1: Matriz 5x10**



$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

**Tabla 1.** Matriz #1 de tamaño 5x10

- Cantidad de elementos: 50 (5 Filas x 10 Columnas)
- Número de ceros = 36 (72% de la matriz)
- Porcentaje a evaluar: 70%
- Cantidad de Trabajadores: 4
- Resultados esperados: Matriz dispersa

Hdispersa:

```
@GabrielJaramilloCuberos →/workspaces/Proyecto-Sistemas-Operativos (main) $ ./hdispersa -f 5 -c 10 -a matriz.txt -n 4 -p 70
178.000000
La matriz en el archivo matriz.txt tiene un total de 36 ceros (72%), por lo tanto, se considera dispersa.
```

**Figura 1.** Resultado de ejecución de hdispersa con la Matriz #1

Pdispersa:

```
s-Operativos (main) $ ./pdispersa -f 5 -c 10 -a matriz.txt -n 4 -p 70
Se va trabajar con 4 procesos.
2253.000000
La matriz en el archivo matriz.txt tiene un total de 36 ceros (72%), por lo tanto, se considera dispersa.
```

**Figura 2.** Resultado de ejecución de pdispersa con la Matriz #1

Como se ve en la **Figura 1** y **Figura 2**, ambos programas hdispersa y pdispersa funcionan correctamente para la Matriz #1 mostrada en la **Tabla 1**.

## - Caso 2: Matriz 10x10

1	1	0	0	1	0	0	1	0	0
0	0	1	1	1	1	0	0	0	1
1	1	0	0	0	0	0	0	1	1
1	1	0	1	1	1	1	0	1	1
0	0	1	1	1	0	0	0	0	0
1	1	1	1	0	1	0	1	1	1
0	1	1	1	1	1	1	0	0	1
1	1	0	0	1	0	0	0	0	0
1	1	0	1	0	0	1	1	1	0
0	0	1	0	1	0	1	0	1	1

**Tabla 1. Matriz #1 de tamaño 5x10**

- Cantidad de elementos: 100 (10 Filas x 10 Columnas)
- Número de ceros = 47 (47% de la matriz)
- Porcentaje a evaluar: 60%
- Cantidad de trabajadores: 2
- Resultados esperados: Matriz no dispersa

Hdispersa:

```
@GabrielJaramilloCuberos →/workspaces/Proyecto-Sistemas-Operativos (main) $ ./hdispersa -f 10 -c 10 -a matriz.txt -n 2 -p 60
Se va a trabajar con 2 trabajadores.
140.000000
La matriz en el archivo matriz.txt tiene un total de 47 ceros (47%), por lo tanto, no se considera dispersa.
```

**Figura 3.** Resultado de ejecución de hdispersa con la Matriz #2

Pdispersa:

```
@GabrielJaramilloCuberos →/workspaces/Proyecto-Sistemas-Operativos (main) $ ./pdispersa -f 10 -c 10 -a matriz.txt -n 2 -p 60
Se va a trabajar con 2 procesos.
333.000000
La matriz en el archivo matriz.txt tiene un total de 47 ceros (47%), por lo tanto, no se considera dispersa.
```

**Figura 4.** Resultado de ejecución de pdispersa con la Matriz #2

Como se ve en la **Figura 3** y **Figura 4**, ambos programas hdispersa y pdispersa funcionan correctamente para la Matriz #2 mostrada en la **Tabla 2**.

## 8.2 Pruebas de Rendimiento Automatizadas

Para poder hacer una evaluación del rendimiento en matrices más grandes (Desde 1000x1000 hasta 10000x10000), se utilizó un sistema automatizado para las pruebas compuesto por:

### A. Compilación con Makefile

El Makefile permitió compilar los programas de manera eficiente como se muestra en la **Figura 5**:

```
@GabrielJaramilloCuberos →/workspaces/Proyecto-Sistemas-Operativos (main) $ make
gcc -c UtilsGeneral/matriz.c -o UtilsGeneral/matriz.o
gcc -c UtilsGeneral/temporizador.c -o UtilsGeneral/temporizador.o
gcc -c UtilsGeneral/utilidades.c -o UtilsGeneral/utilidades.o
gcc -c UtilsHdispersa/hilos.c -o UtilsHdispersa/hilos.o
gcc -o hdispersa hdispersa.c UtilsGeneral/matriz.o UtilsGeneral/temporizador.o UtilsGeneral/utilidades.o UtilsHdispersa/hilos.o -Wall -lm -lpthread
gcc -c UtilsPdispersa/procesos.c -o UtilsPdispersa/procesos.o
gcc -o pdispersa pdispersa.c UtilsGeneral/matriz.o UtilsGeneral/temporizador.o UtilsGeneral/utilidades.o UtilsPdispersa/procesos.o -Wall -lm
g++ -o generador generadormatrices.cpp
```

**Figura 5.** Resultado de ejecución del makefile

### B. Ejecución del script [lanza.pl](#)

El script lanza.pl hecho en el lenguaje de programación Perl permitió automatizar las pruebas de rendimiento como se muestra en la **Figura 6**.

```
@GabrielJaramilloCuberos →/workspaces/Proyecto-Sistemas-Operativos (main) $ ./lanza.pl
Generando matrices con ./generador 2...
Matriz generada correctamente: matriz1000.txt
Tamaño: 1000x1000
Ceros: 770000, Unos: 230000

Matriz generada correctamente: matriz2000.txt
Tamaño: 2000x2000
Ceros: 3080000, Unos: 920000

Matriz generada correctamente: matriz5000.txt
Tamaño: 5000x5000
Ceros: 19250000, Unos: 5750000





















Matriz generada correctamente: matriz8000.txt
Tamaño: 8000x8000
Ceros: 49280000, Unos: 14720000

Matriz generada correctamente: matriz10000.txt
Tamaño: 10000x10000
Ceros: 77000000, Unos: 23000000
```

**Figura 6.** Resultado de ejecución del script [lanza.pl](#)

### C. Resultados Obtenidos

El script lanza.pl automatizó las pruebas de rendimiento, generando archivos de datos para cada combinación de programa y tamaño de matriz. Estos archivos que se ven en la **Figura 7** almacenan los tiempos de ejecución (en microsegundos) de hdispersa y pdispersa en 16 iteraciones, incrementando el número de trabajadores en cada ejecución.

 hdispersa.-1000.dat		22/05/2025 11:29 a. m.	Archivo DAT	3 KB
 hdispersa.-2000.dat		22/05/2025 11:29 a. m.	Archivo DAT	3 KB
 hdispersa.-5000.dat		22/05/2025 11:29 a. m.	Archivo DAT	3 KB
 hdispersa.-8000.dat		22/05/2025 11:29 a. m.	Archivo DAT	3 KB
 hdispersa.-10000.dat		22/05/2025 11:29 a. m.	Archivo DAT	3 KB
 pdispersa.-1000.dat		22/05/2025 11:29 a. m.	Archivo DAT	7 KB
 pdispersa.-2000.dat		22/05/2025 11:29 a. m.	Archivo DAT	7 KB
 pdispersa.-5000.dat		22/05/2025 11:29 a. m.	Archivo DAT	7 KB
 pdispersa.-8000.dat		22/05/2025 11:29 a. m.	Archivo DAT	7 KB
 pdispersa.-10000.dat		22/05/2025 11:29 a. m.	Archivo DAT	7 KB

**Figura 7.** Archivos generados vistos desde el navegador de archivos

A continuación, se analizan las tablas y gráficas generadas para cada tamaño de matriz, destacando patrones clave y diferencias entre ambas implementaciones:

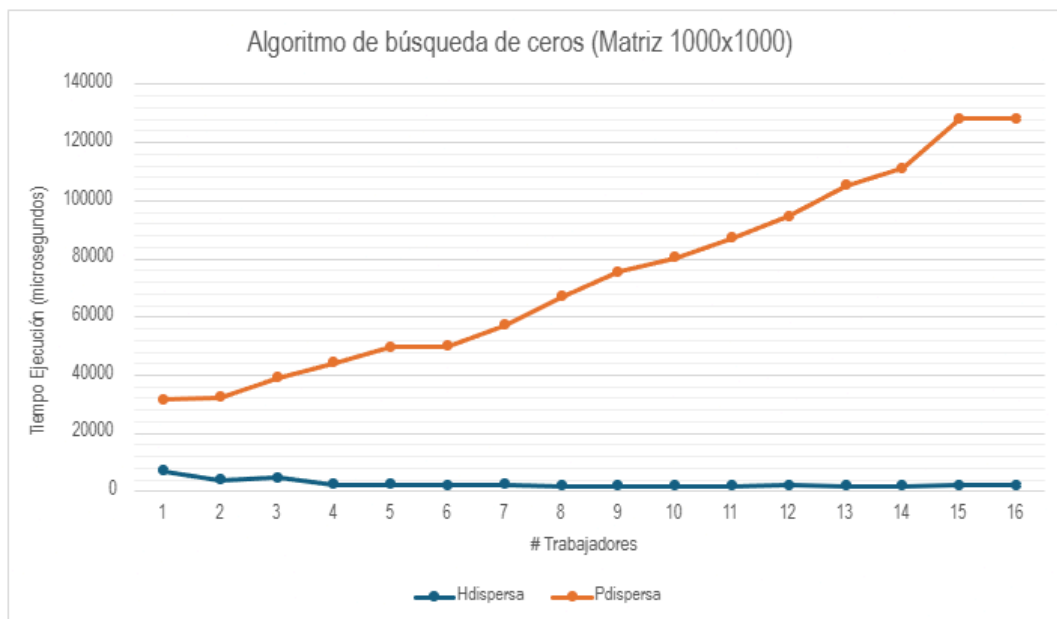
**- Caso Matriz 1000x1000:**

Algoritmo de búsqueda de ceros (microsegundos) (Matriz 1000x1000)		
Repetición	Tiempo (Hdispersa)	Tiempo (Pdispersa)
1	7083	31717
2	3861	32445
3	4736	39205
4	2306	44181
5	2361	49681
6	2050	49953
7	2501	57196
8	1826	67050
9	1835	75415
10	1782	80365
11	1792	87109
12	2094	94535

<b>13</b>	1954	105241
<b>14</b>	1894	111194
<b>15</b>	2143	128271
<b>16</b>	2039	128116

**Tabla 3.** Resultados para Matriz 1000x1000

A continuación en la **Figura 8** se realiza la comparativa de tiempos entre hdispersa y pdispersa para el caso de una matriz 1000x1000:



**Figura 8.** Gráfica hdispersa vs pdispersa para matriz 1000x1000

Como se observa en la Tabla 3 y en la Figura 8, hdispersa mostró tiempos de entre 1782  $\mu$ s con diez trabajadores y 7083  $\mu$ s con un solo trabajador, una reducción progresiva en los tiempos a medida que aumentan los trabajadores. Mientras que pdispersa registró tiempos significativamente mayores comparados a hdispersa (Entre 31717  $\mu$ s y 128271  $\mu$ s), lo que demuestra una mala escalabilidad en este caso.

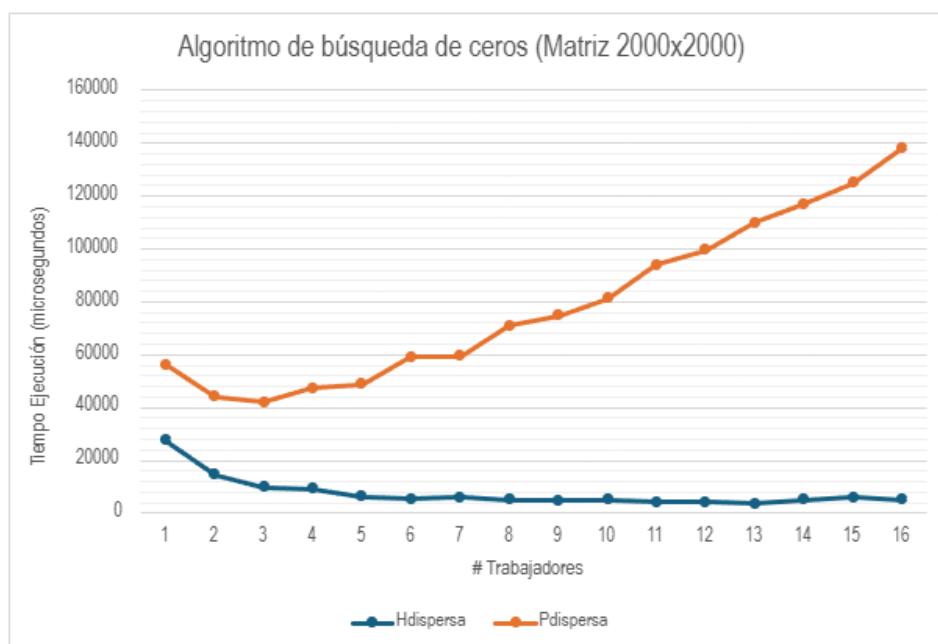
- **Caso Matriz 2000x2000:**

Algoritmo de búsqueda de ceros (microsegundos) (Matriz 2000x2000)		
Repetición	Tiempo (Hdispersa)	Tiempo (Pdispersa)
<b>1</b>	27572	56302
<b>2</b>	14509	44005

<b>3</b>	9925	42052
<b>4</b>	9381	47284
<b>5</b>	6256	48828
<b>6</b>	5506	59019
<b>7</b>	6087	59457
<b>8</b>	5079	70865
<b>9</b>	4885	74719
<b>10</b>	5044	81414
<b>11</b>	4180	94092
<b>12</b>	4085	99537
<b>13</b>	3737	109953
<b>14</b>	5024	116996
<b>15</b>	5867	125011
<b>16</b>	5230	138199

**Tabla 4.** Resultados para Matriz 2000x2000

A continuación en la **Figura 9** se realiza la comparativa de tiempos entre hdispersa y pdispersa para el caso de una matriz 2000x2000:



**Figura 9.** Gráfica hdispersa vs pdispersa para matriz 1000x1000

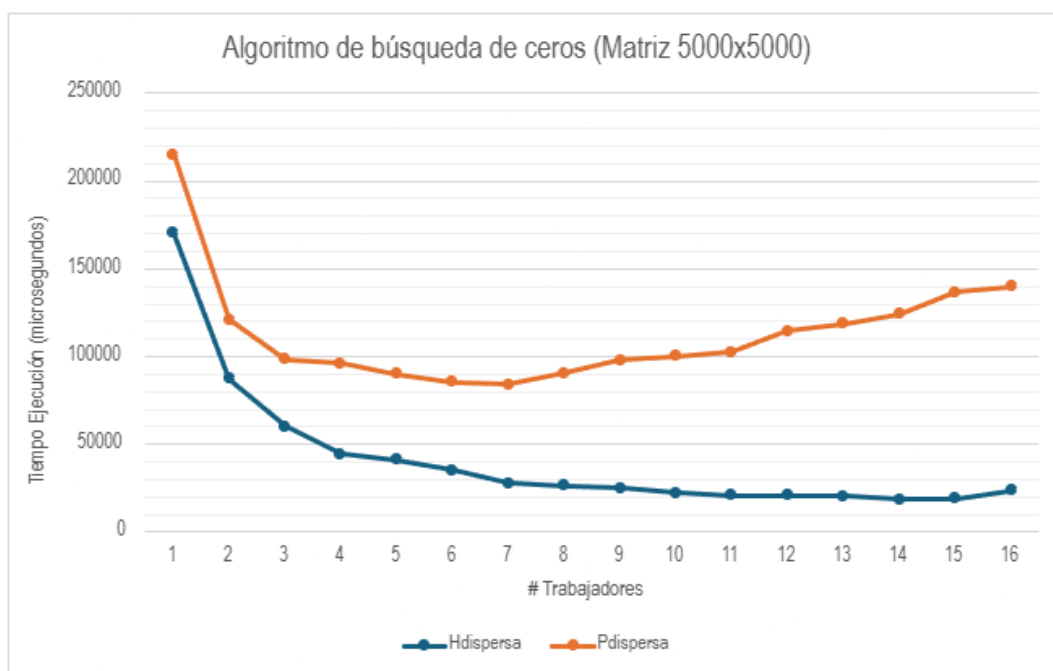
Como se observa en la **Tabla 4** y **Figura 9**, hdispersa mostró una reducción de tiempo a la hora de aumentar los trabajadores, empezando con un tiempo de  $27572 \mu s$  con un solo trabajador y disminuyendo a  $3737 \mu s$  con 13 trabajadores. Mientras que pdispersa, al igual que en el caso de matriz  $1000 \times 1000$ , registró tiempos mayores comparados a hdispersa (*Entre  $42052 \mu s$  y  $138199 \mu s$* ), con menor mejora al aumentar los trabajadores.

- **Caso Matriz  $5000 \times 5000$ :**

<b>Algoritmo de búsqueda de ceros (microsegundos) (Matriz <math>5000 \times 5000</math>)</b>		
<b>Repetición</b>	<b>Tiempo (Hdispersa)</b>	<b>Tiempo (Pdispersa)</b>
<b>1</b>	170380	215187
<b>2</b>	87389	121042
<b>3</b>	60193	98544
<b>4</b>	44349	95911
<b>5</b>	41330	89916
<b>6</b>	35281	85484
<b>7</b>	27929	84321
<b>8</b>	26224	90424
<b>9</b>	24867	97934
<b>10</b>	22400	100447
<b>11</b>	20911	102767
<b>12</b>	20989	114326
<b>13</b>	20229	118680
<b>14</b>	18796	124186
<b>15</b>	19225	136556
<b>16</b>	23489	139876

**Tabla 5.** Resultados para Matriz  $5000 \times 5000$

A continuación en la **Figura 10** se realiza la comparativa de tiempos entre hdispersa y pdispersa para el caso de una matriz 5000x5000:



**Figura 10.** Gráfica hdispersa vs pdispersa para matriz 5000x5000

Como se observa en la **Tabla 5** y **Figura 10**, hdispersa mostró una reducción de tiempo a la hora de aumentar los trabajadores, empezando con un tiempo de  $170380 \mu s$  con un solo trabajador y disminuyendo a  $18796 \mu s$  con 14 trabajadores. Mientras que pdispersa, registró alta variabilidad en los tiempos, mostrando una reducción de tiempo entre tener un solo trabajador  $215187 \mu s$  hasta 7 trabajadores  $84321 \mu s$ , para luego mostrar un aumento de tiempo de los 8 trabajadores en adelante llegando hasta un tiempo de  $139876 \mu s$ .

**- Caso Matriz 8000x8000:**

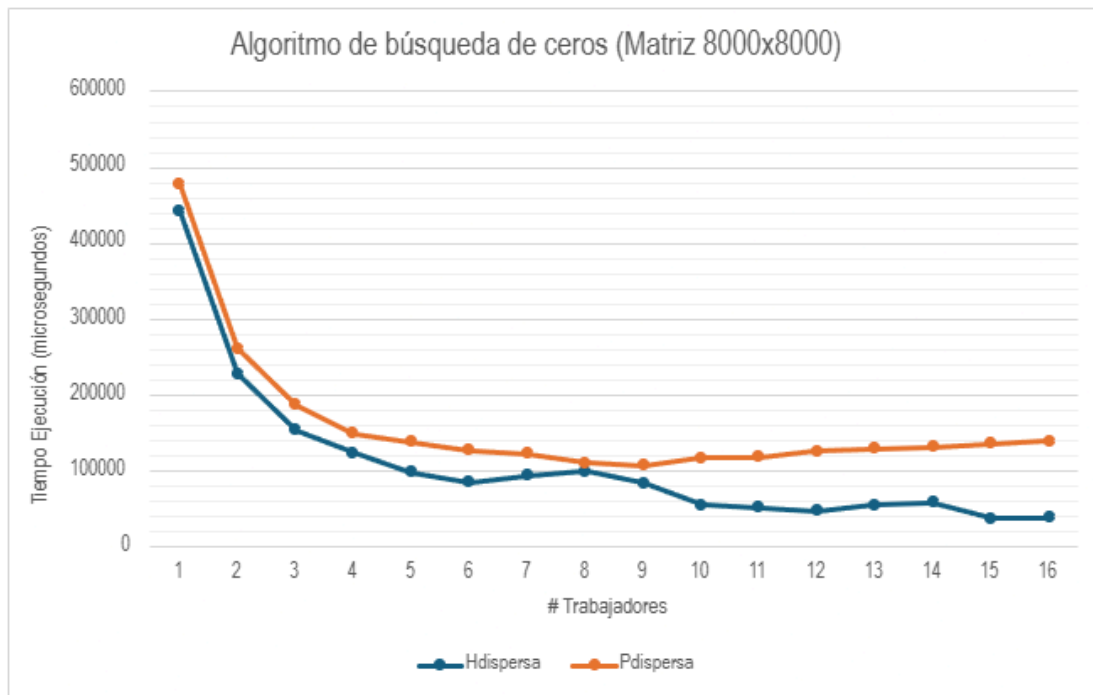
Algoritmo de búsqueda de ceros  
(microsegundos) (Matriz 8000x8000)



<b>Repetición</b>	<b>Tiempo (Hdispersa)</b>	<b>Tiempo (Pdispersa)</b>
<b>1</b>	442107	477216
<b>2</b>	228348	261763
<b>3</b>	154479	186984
<b>4</b>	124893	149916
<b>5</b>	99162	138226
<b>6</b>	85751	127522
<b>7</b>	94331	123692
<b>8</b>	99841	110727
<b>9</b>	84004	107709
<b>10</b>	55604	117521
<b>11</b>	52205	118964
<b>12</b>	48193	126840
<b>13</b>	55247	130433
<b>14</b>	59118	132479
<b>15</b>	38384	136100
<b>16</b>	38655	140227

**Tabla 6.** Resultados para Matriz 8000x8000

A continuación en la **Figura 11** se realiza la comparativa de tiempos entre hdispersa y pdispersa para el caso de una matriz 8000x8000:



**Figura 11.** Gráfica hdispersa vs pdispersa para matriz 8000x8000

Como se observa en la **Tabla 6** y **Figura 11**, hdispersa mantuvo mayormente una curva descendente del tiempo a la hora de aumentar los trabajadores, empezando con un tiempo de  $442107 \mu s$  con un solo trabajador y disminuyendo a  $38384 \mu s$  con 15 trabajadores. Mientras que pdispersa tuvo una reducción de tiempo entre un solo trabajador  $477216 \mu s$  y luego se estabilizó entre  $110727-140227 \mu s$  alrededor de los 6 trabajadores en adelante, mostrando un mejor comportamiento que en los casos anteriores.

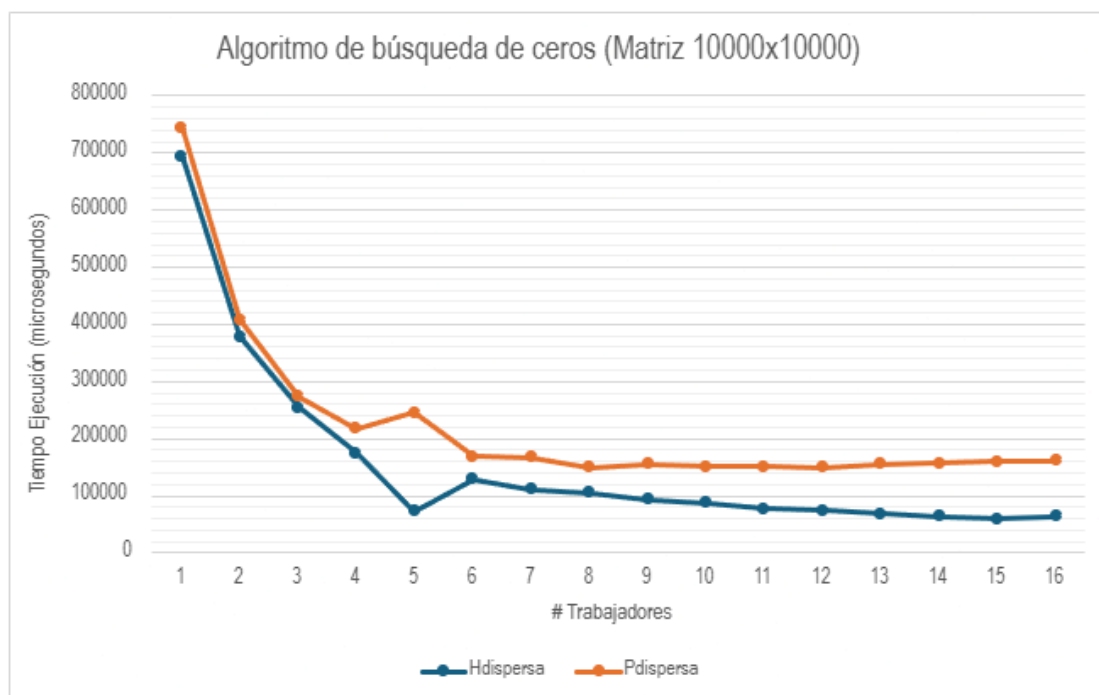
**- Caso Matriz 10000x10000:**

Algoritmo de búsqueda de ceros (microsegundos) (Matriz 10000x10000)		
Repetición	Tiempo (Hdispersa)	Tiempo (Pdispersa)
1	694498	744174
2	378325	407481
3	253846	274206
4	174850	216770
5	73089	245862
6	129109	169133
7	110511	167157

<b>8</b>	105613	149444
<b>9</b>	92779	154931
<b>10</b>	87795	151152
<b>11</b>	77048	151204
<b>12</b>	73608	150003
<b>13</b>	68051	155777
<b>14</b>	63332	157469
<b>15</b>	60045	160482
<b>16</b>	63943	160852

**Tabla 7.** Resultados para Matriz 10000x10000

A continuación en la **Figura 12** se realiza la comparativa de tiempos entre hdispersa y pdispersa para el caso de una matriz 8000x8000:



**Figura 12.** Gráfica hdispersa vs pdispersa para matriz 10000x10000

Como se observa en la **Tabla 7** y **Figura 12**, tanto hdispersa como pdispersa mostraron una reducción del tiempo a la hora de aumentar los trabajadores, con la particularidad de que a partir de los 6 trabajadores hdispersa tuvo una mejora de rendimiento empezando con 129109  $\mu$ s con 6 trabajadores y terminando con 63943  $\mu$ s con 16 trabajadores, mientras que pdispersa tuvo tiempos similares en estas iteraciones (149444–160852  $\mu$ s), lo que muestra que no hubo una mejora significativa.

Es importante tener en cuenta que para todos los resultados de todos los tamaños de matrices probados, ambos programas contaron exitosamente la cantidad de elementos diferentes a cero en todas las matrices analizadas, lo que demuestra una implementación correcta en el algoritmo de búsqueda de ceros y un buen control de las condiciones de carrera para ambos programas.

## 9. Análisis de resultados

El presente análisis tiene como objetivo interpretar los resultados obtenidos y mostrados en la sección anterior, estableciendo relaciones entre el comportamiento observado, las características técnicas del equipo utilizado y los fundamentos teóricos de sistemas concurrentes. Buscamos explicar las diferencias de desempeño entre las implementaciones con hilos (hdispersa) y procesos (pdispersa), identificando los factores que influyeron en su escalabilidad y eficiencia.

Es importante tener en cuenta que las pruebas se ejecutaron en un equipo con un procesador AMD Ryzen 9 5900HS (8 núcleos físicos, 16 hilos lógicos), lo que permitió evaluar el rendimiento en un escenario real de paralelismo y a su vez muestra por qué el límite de 16 workers en las pruebas.

Los resultados obtenidos en las pruebas de rendimiento de la **Sección 9** demuestran diferencias significativas entre las implementaciones con hilos (hdispersa) y procesos (pdispersa), las cuales pueden explicarse mediante los fundamentos teóricos de sistemas concurrentes y las características del hardware utilizado.

Para matrices pequeñas y medianas ( $1000 \times 1000$  a  $5000 \times 5000$ ), se observó que hdispersa logró una mejora constante en el rendimiento al incrementar el número de trabajadores, reduciendo los tiempos de ejecución de manera casi lineal. Este comportamiento se debe principalmente a la eficiencia de los hilos POSIX, que al operar en un espacio de memoria compartido se minimiza el overhead de comunicación y sincronización. Por el contrario, pdispersa mostró un aumento en los tiempos de ejecución al utilizar más trabajadores, consecuencia directa del alto costo asociado a la creación de procesos mediante `fork()` y la necesidad de comunicación mediante archivos temporales, lo que introduce una latencia significativa.

En el caso de matrices más grandes ( $8000 \times 8000$  a  $10000 \times 10000$ ), aunque hdispersa mantuvo su ventaja en rendimiento, la diferencia con pdispersa disminuyó. Esto ocurre porque el tiempo de cómputo pasa a dominar sobre el overhead de comunicación, mitigando parcialmente las limitaciones de pdispersa. Sin embargo, incluso en estos escenarios, la implementación con hilos demostró una mejor escalabilidad gracias a su capacidad para

aprovechar óptimamente los recursos del sistema, particularmente los 16 hilos lógicos del procesador AMD Ryzen 9 5900HS utilizado en las pruebas.

Ambos programas tuvieron sus propias limitaciones. En el caso de hdispersa, el principal factor que afectó el rendimiento fue el overhead de creación y gestión de hilos. Por otro lado, pdispersa enfrentó desafíos más significativos debido al alto costo de creación de procesos y especialmente a la necesidad de comunicación mediante archivos temporales

El análisis también muestra que el rendimiento no es proporcional por completo al número de trabajadores, sino que está limitado por factores como la arquitectura del hardware y la naturaleza de las operaciones.

## 10. Consideraciones adicionales

- Para la ejecución de ambos programas, se debe tener en cuenta el procesador del sistema de cómputo, ya que su misma arquitectura afecta directamente el rendimiento de los programas.
- Para una mayor rigurosidad en las pruebas de rendimiento de ambos programas, es necesario repetir dicha prueba en varios sistemas de cómputo, cada uno diferente al otro, con el objetivo de poder concluir cuál de los dos programas es más eficiente. En este caso, se usó solo un sistema de cómputo ya que el objetivo del proyecto no es evaluar el rendimiento de los programas sino implementarlos de manera correcta. Las pruebas de rendimiento se hicieron por dos razones:
  - Confirmar que ambos programas funcionan de manera correcta, independientemente de cuántas repeticiones se hagan o que matrices se evalúen y cuántos trabajadores se usen.
  - Observar y analizar cómo se comporta la ejecución de programas a través de hilos y procesos, y comparar las diferencias que existen entre ambos.

## 11. Conclusiones

Una vez compilado y ejecutado el programa, se obtuvo las siguientes conclusiones:

1. **Funcionamiento correcto:** Ambas implementaciones, determinaron correctamente si una matriz es dispersa en todos los casos de prueba, independientemente del tamaño de la matriz (Desde 5x5 en los casos personalizados o hasta 10000x10000 en los de pruebas de rendimiento) o del número de trabajadores (1 a 16). Esto valida la solidez del diseño y la implementación de ambos enfoques.

2. **Rendimiento superior de hilos:** La versión con hilos (hdispersa), tuvo un mejor rendimiento que la de procesos (pdispersa), especialmente en matrices pequeñas (1000x1000 a 5000x5000). Esto se debe a que los hilos comparten memoria y no necesitan mecanismos de comunicación externos los cuales generan overhead.
3. **Escalabilidad:** En sistemas con múltiples núcleos lógicos, como el utilizado para las pruebas, el rendimiento mejora al incrementar los trabajadores. Sin embargo, el beneficio se estabiliza cerca del número de núcleos lógicos disponibles.
4. **Importancia del paralelismo:** El proyecto demuestra cómo técnicas de programación concurrente pueden reducir considerablemente los tiempos de cómputo para tareas intensivas como el análisis de grandes volúmenes de datos.
5. **Usos:** Las matrices dispersas pueden usarse en otros problemas de álgebra numérica o análisis de datos en matrices, optimizando su procesamiento mediante paralelización.

## 12. Resumen

El proyecto desarrollado aborda el problema de determinar si una matriz de gran tamaño es o no dispersa, es decir, si en su totalidad cuenta con una mayoría de valores en cero. Para ello, se implementaron dos soluciones concurrentes:

- **Hdispersa:** Usa hilos POSIX (pthread\_create), que aprovechan la memoria compartida para un rendimiento más eficiente.
- **Pdispersa:** Usa procesos con fork(), que requieren comunicación por archivos ya que no comparten memoria.

Ambas versiones dividen el trabajo entre múltiples trabajadores (hilos o procesos), cada uno procesando un subconjunto de la matriz. El programa permite configurar desde la línea de comandos el número de trabajadores, el archivo de la matriz y el porcentaje mínimo de dispersión para considerarla dispersa.

Se realizaron pruebas funcionales con matrices pequeñas y pruebas de rendimiento con matrices grandes, utilizando un script automatizado en un equipo con 16 hilos. Los resultados se almacenan en archivos .dat y se graficaron para comparar el rendimiento entre ambas implementaciones.