HPC

RETO II - OpenMP y Benchmarks

Algoritmo de Jacobi

Presentado a:

Ramiro Andres Barrios Valencia

Presentado por:

Héctor Fabio Vanegas Guarín

Juan Miguel Aguirre Bedoya

Brayan Cataño Giraldo

Universidad Tecnológica de Pereira

2025

1. RESUMEN

Por medio del presente documento se muestra el proceso realizado para implementar el algoritmo de Jacobi para resolver la ecuación de Poisson 1D en el lenguaje de programación C. El algoritmo fue ejecutado de dos formas: paralela con hilos utilizando dos librerías diferentes, esto con el fin de analizar su desempeño, variando los parámetros N (puntos de la malla) y NSTEPS (número de iteraciones). Se realizaron pruebas ejecutando el algoritmo con paralelismo con dos números de hilos (4 y 12) utilizando la librería pthread y otra versión con openmp. Se escogieron estas dos de manera subjetiva basándonos en los gráficos comparativos de la entrega anterior y tomando los que tuvieran mejor rendimiento, que en este caso fueron las pruebas hechas con 4 y 12 hilos.

Es así, que se realizaron las pruebas en el computador que se han estado haciendo y se compararon con pruebas hechas en otro con características diferentes para analizar su funcionamiento en cada uno y ver si la diferencia de arquitectura entre ambos puede generar una mejora significativa o no, además, se hicieron pruebas de benchmarks para analizar de una manera más cercana el rendimiento de ambos equipos.

2. INTRODUCCIÓN

En el método de Jacobi para resolver la ecuación de Poisson 1D, el parámetro N determina el número de puntos de la malla, influyendo directamente en la precisión espacial y los requerimientos de memoria. Por otro lado, NSTEPS define la cantidad de iteraciones realizadas, mejorando la convergencia sin incrementar la memoria necesaria.

Al aumentar N, se logra una mejor discretización espacial, pero esto requiere un incremento más que proporcional en NSTEPS debido a que la tasa de convergencia disminuye aproximadamente como $(1-\pi^2/N^2)$ para ecuaciones tipo Poisson. El tiempo de ejecución escala como $O(N \times NSTEPS)$, lo que significa que duplicar N tiene un impacto computacional significativamente mayor que duplicar NSTEPS.

La estrategia óptima consiste en establecer N según la precisión espacial requerida y posteriormente ajustar NSTEPS experimentalmente hasta alcanzar la convergencia deseada. Es importante considerar que aumentar NSTEPS solo reduce el error iterativo pero no puede superar las limitaciones inherentes a la discretización determinada por N. Para valores elevados de N (por ejemplo, 10^7), se recomienda incrementar NSTEPS por un factor significativo (como 10^5), monitoreando continuamente la convergencia para realizar ajustes según sea necesario.

3. MARCO CONCEPTUAL

a. Algoritmo de Jacobi

El algoritmo de Jacobi es un método iterativo para resolver sistemas de ecuaciones lineales, especialmente aplicable a problemas como la ecuación de Poisson. A diferencia de la sección sobre multiplicación de matrices que aparece en el texto original, el método de Jacobi se enfoca en encontrar soluciones aproximadas a través de iteraciones sucesivas.

En el contexto de la ecuación de Poisson 1D, el algoritmo de Jacobi:

- Discretiza el dominio continuo en N puntos equidistantes.
- Establece una aproximación inicial para la solución en cada punto.
- Actualiza el valor en cada punto basándose en los valores vecinos de la iteración anterior.
- Repite el proceso de actualización durante NSTEPS iteraciones.
- Converge gradualmente hacia la solución exacta del sistema.

La eficiencia y precisión del método dependen crucialmente de la interrelación entre los parámetros N y NSTEPS, como se explicó en la introducción. El balance adecuado entre estos parámetros es fundamental para lograr resultados precisos con un costo computacional razonable.

b. High Performance Computing

El High Performance Computing (HPC) o Computación de Alto Rendimiento comprende el uso de supercomputadoras y técnicas de computación paralela para resolver problemas complejos que requieren gran capacidad de procesamiento. En el contexto del método de Jacobi para la ecuación de Poisson, el HPC resulta fundamental cuando se trabaja con dominios extensos (valores grandes de N) o cuando se necesitan muchas iteraciones (NSTEPS elevado).

Las arquitecturas HPC modernas incorporan múltiples niveles de paralelismo, desde procesadores multinúcleo hasta clusters de computadoras interconectadas. Estas infraestructuras permiten distribuir la carga computacional, reduciendo significativamente los tiempos de ejecución para problemas de gran escala. El aprovechamiento eficiente de recursos HPC requiere algoritmos específicamente diseñados para explotar el paralelismo inherente de los problemas, como es el caso del método de Jacobi, cuyas operaciones pueden realizarse de forma concurrente.

c. Complejidad Computacional

La complejidad computacional proporciona un marco teórico para analizar la eficiencia de los algoritmos en términos de recursos requeridos (tiempo y espacio) en función del tamaño de entrada. Para el método de Jacobi aplicado a la ecuación de Poisson 1D, la complejidad temporal es $O(N \times NSTEPS)$, donde cada iteración requiere $O(N)$ operaciones y se realizan NSTEPS iteraciones.

Esta complejidad subraya la importancia de seleccionar adecuadamente los parámetros N y NSTEPS, ya que ambos influyen directamente en el tiempo de ejecución. La complejidad espacial es $O(N)$, determinada por el almacenamiento necesario para los valores en cada punto de la malla.

La comprensión de estas complejidades resulta esencial para predecir el comportamiento del algoritmo al escalar el problema y para desarrollar implementaciones eficientes que minimicen los recursos computacionales requeridos.

d. Programación paralela

La programación paralela comprende técnicas y paradigmas para desarrollar algoritmos que ejecuten múltiples instrucciones simultáneamente. Para el método de Jacobi, la programación paralela ofrece oportunidades significativas de optimización, ya que las actualizaciones de los valores en cada punto de la malla pueden realizarse de manera independiente dentro de cada iteración.

Existen diversos modelos de programación paralela aplicables al método de Jacobi:

Paralelismo de datos: División del dominio espacial entre múltiples unidades de procesamiento.

Paralelismo de tareas: Asignación de diferentes componentes algorítmicos a distintas unidades de procesamiento.

Modelos híbridos: Combinación de múltiples niveles de paralelismo para maximizar el rendimiento.

e. Hilos y Speedup

Tanto en implementaciones con pthreads como con OpenMP, los hilos ejecutan trozos de la malla de Jacobi en paralelo. Para cuantificar la mejora que aporta OpenMP frente al esquema basado en pthreads, definimos un speedup relativo como

$$speedup_{OMPvsPT} = \frac{T_{pthreads}}{T_{OpenMP}}$$

donde

- $T_{pthreads}$ es el tiempo medido usando la versión con pthreads como línea base.
- T_{OpenMP} es el tiempo medido con la versión OpenMP.

En un mundo ideal, un speedup mayor que 1 indica que OpenMP supera a pthreads; un valor cercano a 1 señala rendimientos parejos. Al igual que en cualquier paralelización, esta mejora real dependerá de:

Overhead de sincronización y gestión de hilos: Diferencias en cómo cada librería inicia, coordina y destruye los hilos.

Estrategias de scheduling: OpenMP y pthreads pueden emplear distintos esquemas (p. ej. estático vs. dinámico) al repartir iteraciones.

Contención de recursos: Competencia por memoria compartida y ancho de banda, que afecta de forma distinta a cada implementación.

4. Marco Contextual

a. Características de las máquinas

Para la realización de este laboratorio, se hizo uso de dos dispositivos de cómputo, los cuales constan de las siguientes características:

Feature	Details
Procesador	**AMD Ryzen 5 4600h**
Núcleos e Hilos	**6 núcleos 12 hilos**
Frecuencia base	**3.00 GHz**
Frecuencia turbo máxima	**4.00 GHz**
RAM	**24GB DDR4 @ 3200 MHz**
Sistema Operativo	**Ubuntu 24.04.2 LTS**
Compilador Utilizado	**gcc**

Feature	Details
Procesador	**INTEL Core i9-13900KF**
Núcleos e Hilos	**24 núcleos 32 hilos**
Frecuencia base	**2.20 - 3.00 GHz (Dependiendo del núcleo)**
Frecuencia turbo máxima	**5.80 GHz**
RAM	**128GB**
Sistema Operativo	**Ubuntu 24.04.2 LTS**
Compilador utilizado	**gcc**

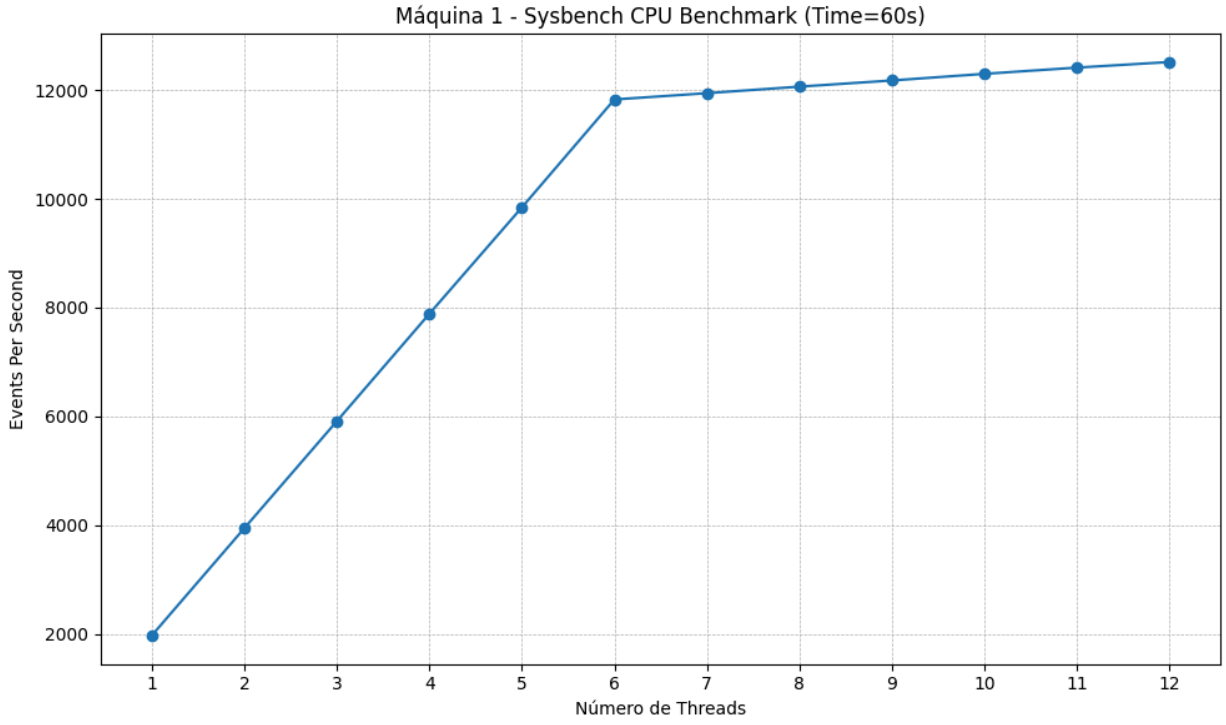
b. Benchmarks

Se realizaron benchmarks preliminares para comparar las dos máquinas en sysbench, todas con una configuració de —time=60:

- **Número de threads vs Eventos por segundo.**

  Para la máquina 1:
  - Desde 1 hasta 12 threads.

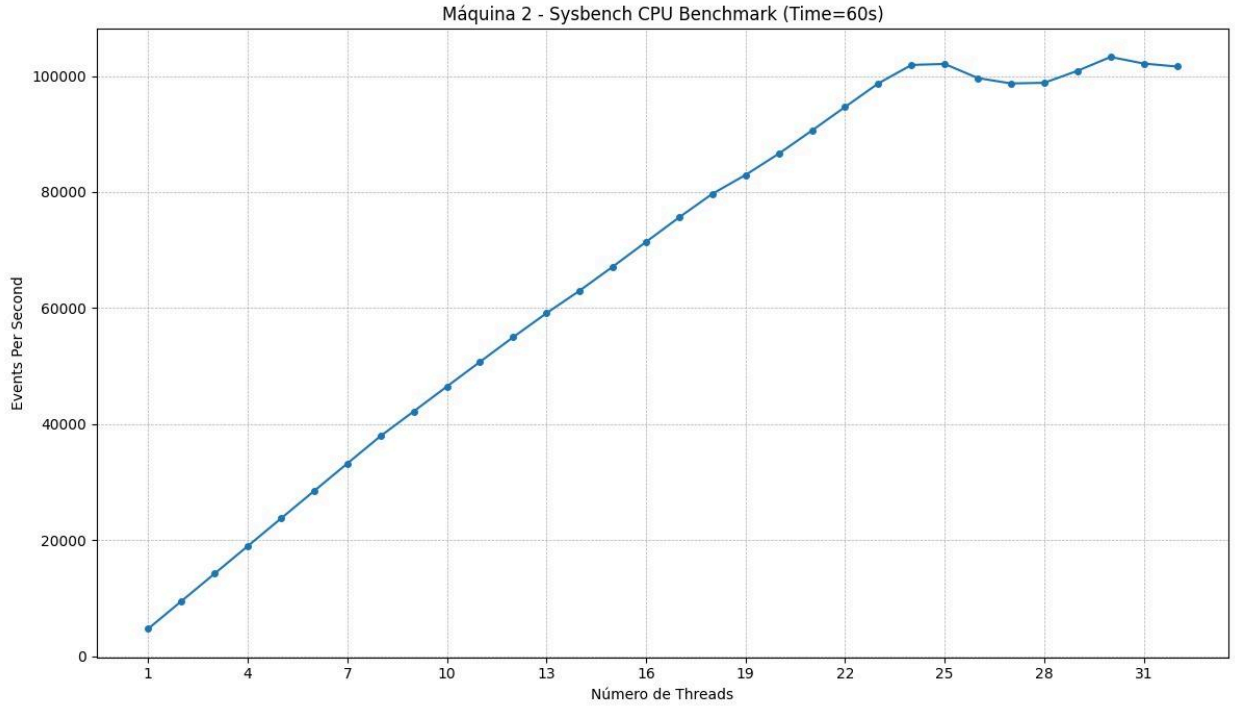
  Llegando a 12000 eventos por segundo utilizando 7 hilos, comenzando en 2000 eventos con solo 1 hilo. A partir de 6 hilos, no hay mucha diferencia en los eventos, esto puede ser porque a pesar de haber 12 hilos solo hay 6 núcleos físicos.



Para la máquina 2:

- Desde 1 hasta 32 threads.

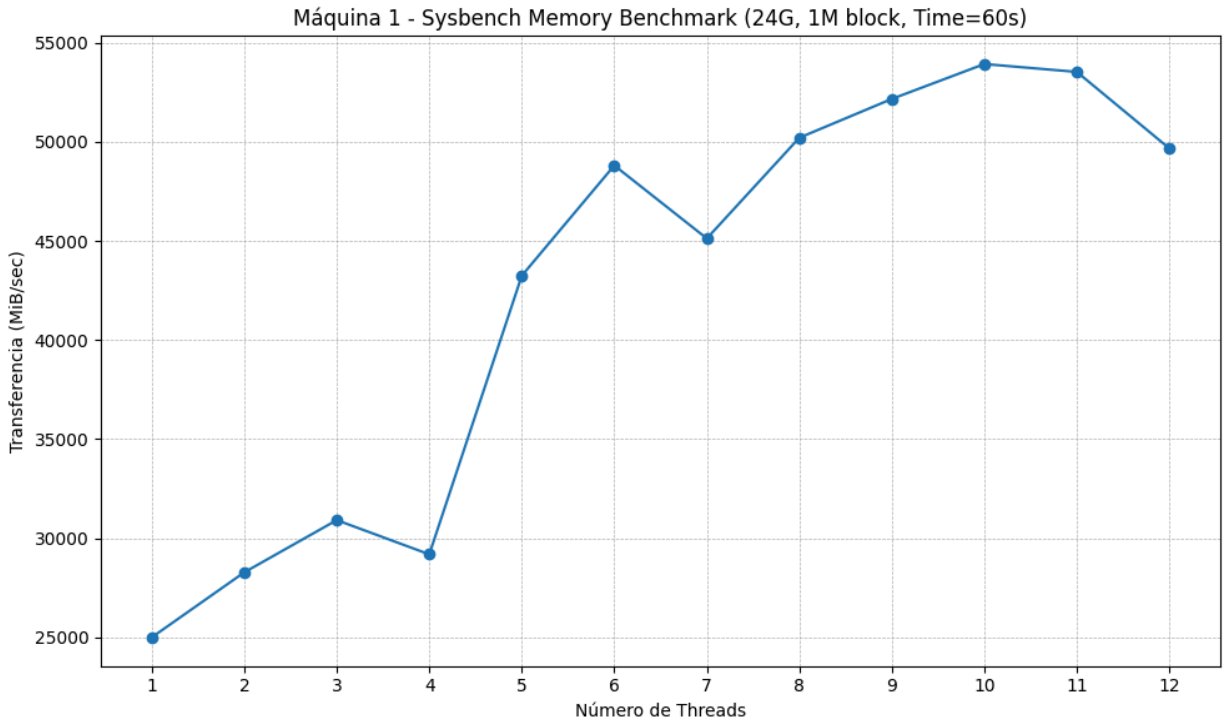
Llegando a 100000 eventos por segundo utilizando 24 hilos, y desde 25 hilos en adelante los eventos por segundo se reducen o se mantienen iguales, esto es probablemente porque solo hay 24 núcleos físicos a pesar de haber 32 threads, con 6 hilos se llegó a casi 30000 eventos por segundo y con 1 solo hilo aproximadamente 5000.



- **Número de threads vs Transferencia (MiB/sec).**

  Para la máquina 1:
  - Desde 1 hasta 12 threads, memoria total = 24G y tamaño de bloque = 1M.

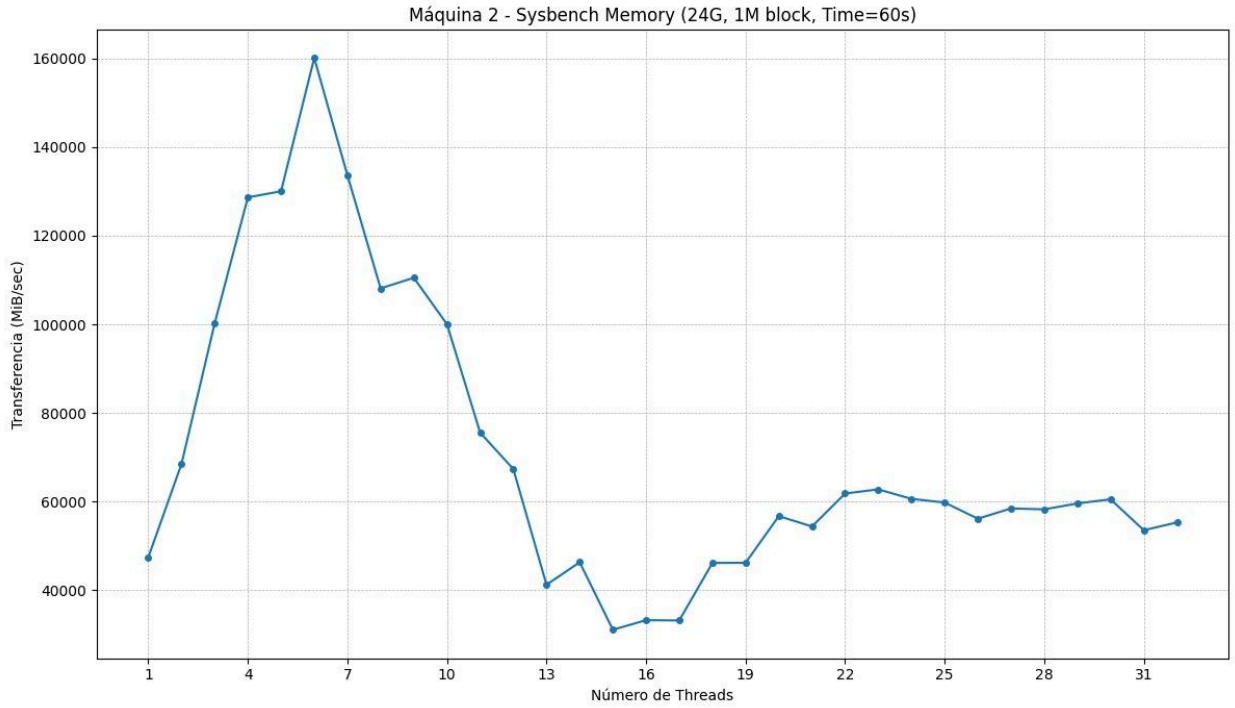
  En esta máquina podemos ver que es rentable aumentar los hilos para mejorar la transferencia hasta los 10 hilos, luego, comienza a bajar la velocidad. Comenzando en 25000 MiB/sec con un hilo, pasando por casi 55000 MiB/sec con 10 hilos en su punto máximo y terminando en 50000 MiB/sec en 12 hilos.



Para la máquina 2:

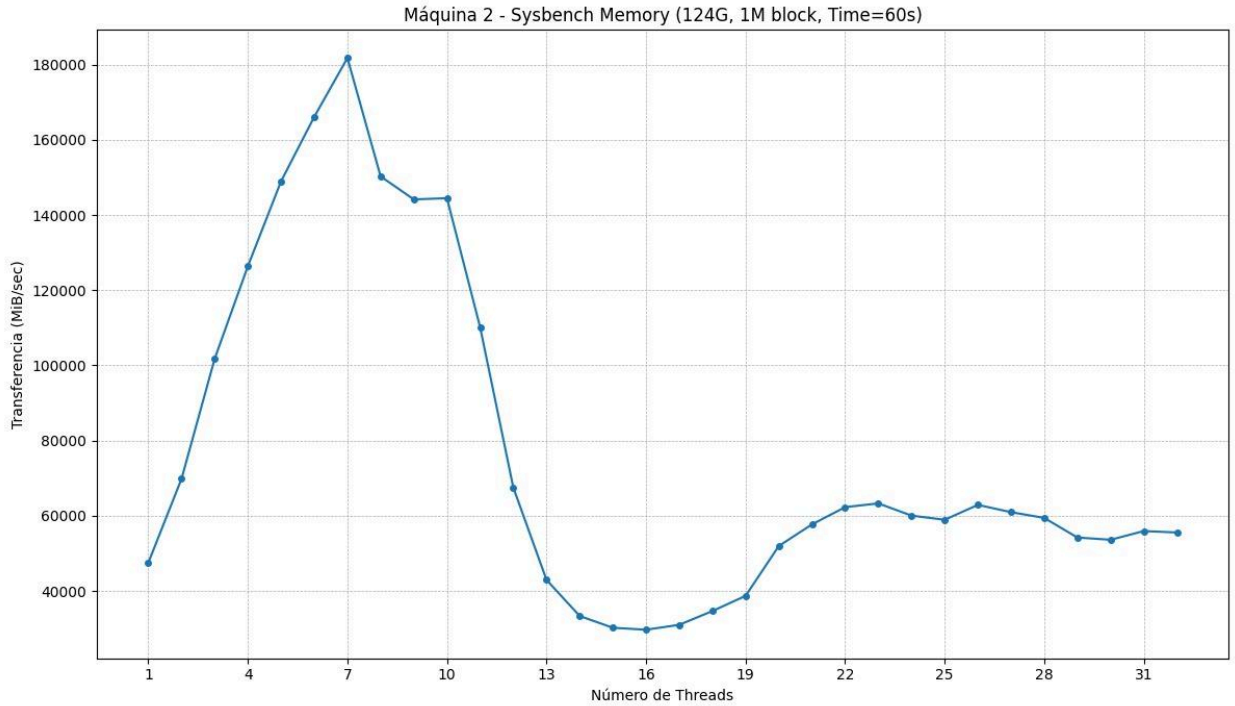
- Desde 1 hasta 32 threads, memoria total = 24G y tamaño de bloque = 1M.

En esta máquina vemos que el pico máximo en esta prueba fue utilizando menos de 7 hilos, alcanzando 160000 MiB/sec, luego desde 15 hilos hasta 17 hilos vemos una taza de transferencia muy baja (menos de 40000 MiB/sec).



- Desde 1 hasta 32 threads, memoria total = 124G y tamaño de bloque = 1M.

También se hizo el mismo cálculo pero con 124G de memoria total. Muy parecido con la anterior prueba pero con una curva mucho más suave y una transferencia máxima de 180000 MiB/sec utilizando 7 hilos.



c. Desarrollo de tests de jacobi

En el desarrollo de esta actividad, se tomó la implementación anterior de hilos, que dio mejor rendimiento utilizando la librería de pthread y se realizó una nueva implementación de esta pero ahora utilizando la librería de openmp, con el fin de comparar su rendimiento y analizar las mejoras de velocidad (speedup) obtenidas mediante las diversas técnicas de optimización ya implementadas en el anterior entregable.

Todas las pruebas se realizaron con valores de N = {10000, 50000, 100000, 500000, 1000000} y NSTEPS = {100, 500, 1000, 2000, 5000}, además, de enfocarnos en 4 y 12 hilos.

Inicialmente, es pertinente recordar el análisis hecho para la implementación de hilos en la versión secuencial y pasar a versiones paralelas de hilos, donde en aquel momento se identificó que el algoritmo de Jacobi presenta oportunidades naturales para la ejecución paralela. Específicamente, detectamos dos iteraciones principales en la función base que podían ser paralelizadas: la actualización de los valores temporales (utmp) y la actualización de los valores del arreglo resultante (u). Como estos cálculos se realizan de manera independiente para cada punto de la malla, desarrollamos una solución con hilos para distribuir esta carga de trabajo. Para estructurar adecuadamente esta implementación, definimos una estructura de datos específica para los hilos, que contenía toda la información necesaria para que cada hilo pudiera realizar su trabajo: el rango de índices a procesar (inicio y fin), punteros a los arreglos u, utmp y f, valores constantes como h2 (el cuadrado del espaciado de la malla), un identificador del hilo y una barrera compartida para sincronización entre hilos.

Además, implementamos una función específica para ser ejecutada por cada hilo, que incluía un bucle para los barridos (sweeps), la lógica para actualizar utmp basado en u, sincronización con otros hilos usando barreras, la lógica para actualizar u basado en utmp, y otra sincronización antes del siguiente barrido. Para dividir eficientemente el trabajo entre los hilos, calculamos cuántos puntos debía procesar cada hilo mediante una fórmula: con n-1 puntos interiores y p hilos, cada hilo procesaría aproximadamente (n-1)/p puntos. Para casos donde (n-1) no era exactamente divisible por p, implementamos un mecanismo para asignar un punto adicional a algunos hilos y así equilibrar la carga. Por ejemplo, para el valor máximo de n que podíamos procesar con 20GB de RAM en nuestra máquina de pruebas, el cálculo era (833333332-1)/12 = 69444444.25, lo que requería asignar un punto adicional a algunos hilos para asegurar que todos los puntos fueran procesados.

Para garantizar que la sincronización se estaba logrando correctamente, verificamos condiciones específicas: todos los hilos debían completar la actualización de utmp antes de que cualquiera comenzará a actualizar y todos los hilos debían completar la actualización de u antes de comenzar el siguiente barrido. Las modificaciones principales en la función de Jacobi para implementar esta versión con hilos incluyeron la inicialización de la barrera compartida, creación de las estructuras de datos para cada hilo, inicio de los hilos, espera hasta que todos los hilos terminaran, y liberación de los recursos de la barrera.

Como extensión de la versión con hilos, implementamos una variante que optimiza el uso de memoria compartida entre ellos. Esta implementación se enfocó en minimizar la contención por acceso a memoria y maximizar la localidad de datos para mejorar el rendimiento en sistemas con múltiples procesadores o núcleos. Además, desarrollamos una versión que utilizaba múltiples procesos mediante la biblioteca MPI (Message Passing Interface), especialmente relevante para entornos de computación distribuida donde los recursos de memoria están físicamente separados.

Una parte importante de nuestras pruebas consistió en experimentar con diferentes opciones de compilación para evaluar su impacto en el rendimiento. Probamos varias combinaciones de flags de optimización del compilador (-O1, -O2, -O3), así como opciones específicas para arquitecturas vectoriales y multinúcleo. Los resultados obtenidos con cada implementación y configuración fueron registrados sistemáticamente, permitiéndonos calcular el speedup para cada caso y analizar cómo escala el rendimiento con el número de hilos/procesos y con diferentes tamaños de problema. Todo el código fuente resultante de nuestras implementaciones, junto con las instrucciones para su correcta ejecución y reproducción de los experimentos, fue debidamente documentado y archivado para referencia futura.

d. Pruebas

Se automatizó el proceso para obtener los resultados por medio de este código bash:

benchmark.sh

```
#!/bin/bash

# Crear archivo para resultados
echo "N,NSTEPS,TIEMPO(s)" > resultados_benchmark.csv

# Array de valores N a probar
N_VALUES=(10000 50000 100000 500000 1000000)

# Array de valores NSTEPS a probar
NSTEPS_VALUES=(100 500 1000 2000 5000)

# Compilar el programa
gcc -DUSE_CLOCK -O3 "1. original-jacobi1d.c" timing.c -o jacobi1d

# Ejecutar benchmarks
for N in "${N_VALUES[@]}"; do
    for STEPS in "${NSTEPS_VALUES[@]}"; do
        echo "Ejecutando con N=$N, NSTEPS=$STEPS"
        TIEMPO=$(./jacobi1d $N $STEPS | grep "Elapsed time" | awk '{print $3}')
        echo "$N,$STEPS,$TIEMPO" >> resultados_benchmark.csv
        # Pequeña pausa para que el sistema se recupere
        sleep 1
    done
done

echo "Benchmark completado. Resultados en resultados_benchmark.csv"
```

Esta fue la base para crear otros archivos bash que sirvieron para poner a funcionar las diferentes pruebas, es decir, se les hizo algunas mejoras en otras versiones del archivo bash para cumplir diferentes objetivos.

i. **Máquina 1:**

1. **Tabla 1: Resultados de la ejecución con 4 hilos con la librería pthreads.**

NSTEPS/N	10000	50000	100000	500000	1000000
100	0.002802	0.005455	0.009252	0.217807	0.502592
500	0.008564	0.017384	0.026479	0.868598	2.251880
1000	0.014593	0.037064	0.076506	1.709170	4.792590
2000	0.025906	0.073970	0.110513	3.372600	9.491670
5000	0.075448	0.158755	0.248312	8.860620	23.090600
Promedio	0.025462	0.058525	0.094212	3.005759	8.025866

2. **Tabla 2: Resultados de la ejecución con 12 hilos con la librería pthreads.**

NSTEPS/N	10000	50000	100000	500000	1000000
100	0.002091	0.005737	0.010174	0.193273	0.506876
500	0.010232	0.018743	0.037772	0.822792	2.355320
1000	0.016109	0.034601	0.056945	1.486550	4.925630
2000	0.027970	0.067134	0.107003	2.981880	9.775470
5000	0.071351	0.164277	0.274882	8.416160	22.982900
Promedio	0.025550	0.058098	0.097355	2.780131	8.108239

3. **Tabla 3: Resultados de la ejecución con 4 hilos con la librería openmp.**

NSTEPS/N	10000	50000	100000	500000	1000000
100	0.000743	0.001377	0.003001	0.095588	0.245161
500	0.002755	0.007248	0.010995	0.512993	1.162549
1000	0.005126	0.013986	0.018457	0.850979	2.318154
2000	0.011923	0.021349	0.035368	1.508615	4.585857
5000	0.018369	0.044756	0.083068	3.931784	11.100433
Promedio	0.007783	0.0177430	0.030177	1.379991	3.882430
Speedup	3.271489	3.298483	3.121980	2.178100	2.067227

4. **Tabla 4: Resultados de la ejecución con 12 hilos con la librería openmp.**

NSTEPS/N	10000	50000	100000	500000	1000000
100	0.029204	0.002211	0.035642	0.106626	0.274190
500	0.018690	0.057858	0.014524	0.440765	1.540795
1000	0.011741	0.018201	0.047320	0.890940	3.088458
2000	0.038392	0.029133	0.040977	1.903148	5.950990
5000	0.042086	0.139763	0.165033	4.049050	14.091674
Promedio	0.028022	0.049433	0.060699	1.478105	4.946107
Speedup	0.911783	1.175287	1.603897	1.880875	1.639317

ii. **Máquina 2:**

1. **Tabla 5: Resultados de la ejecución con 4 hilos con la librería pthreads.**

NSTEPS/N	10000	50000	100000	500000	1000000
100	0.001232	0.002363	0.004288	0.042722	0.084446
500	0.003697	0.010582	0.018074	0.208474	0.411121
1000	0.007197	0.019543	0.036215	0.406036	0.833877
2000	0.013462	0.039968	0.071185	0.827816	1.670070
5000	0.032545	0.098215	0.181097	2.067550	4.053560
Promedio	0.011626	0.034134	0.062171	0.710519	1.410614

2. **Tabla 6: Resultados de la ejecución con 12 hilos con la librería pthreads.**

NSTEPS/N	10000	50000	100000	500000	1000000
100	0.003540	0.005188	0.006138	0.025329	0.085728
500	0.013158	0.017733	0.025594	0.113802	0.422704
1000	0.023372	0.032413	0.048110	0.222263	0.821272
2000	0.047189	0.063581	0.101995	0.442292	1.665580
5000	0.114176	0.163235	0.230246	1.094820	4.109910
Promedio	0.040287	0.056430	0.082416	0.379701	1.421038

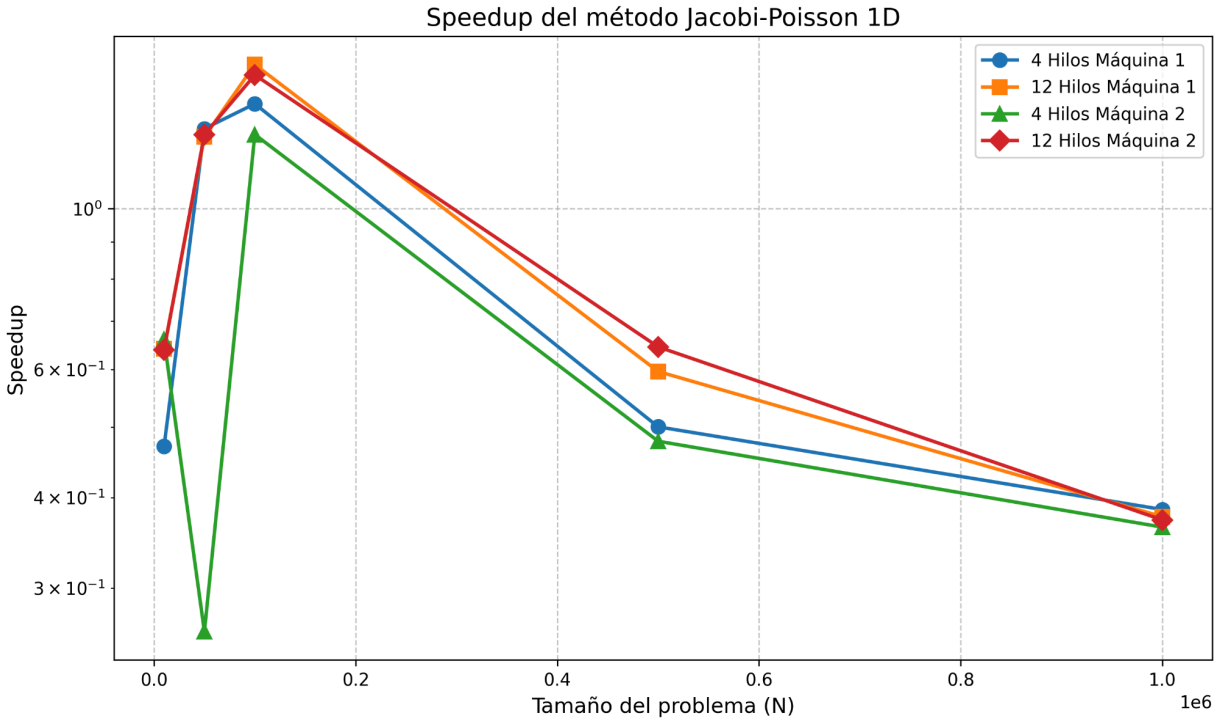
3. **Tabla 7: Resultados de la ejecución con 4 hilos con la librería openmp.**

NSTEPS/N	10000	50000	100000	500000	1000000
100	0.002208	0.002462	0.003422	0.011986	0.033500
500	0.007747	0.012087	0.011669	0.053146	0.114892
1000	0.014095	0.018141	0.028092	0.108637	0.234185
2000	0.025464	0.029427	0.041820	0.219245	0.435918
5000	0.067630	0.082449	0.104501	0.533135	1.119893
Promedio	0.023428	0.028913	0.037900	0.185229	0.387677
Speedup	0.496243	1.180576	1.640395	3.835895	3.638632

4. **Tabla 8: Resultados de la ejecución con 12 hilos con la librería openmp.**

NSTEPS/N	10000	50000	100000	500000	1000000
100	0.003182	0.003683	0.004300	0.010029	0.012198
500	0.011820	0.013839	0.016628	0.037995	0.091445
1000	0.021147	0.025280	0.030547	0.073152	0.136586
2000	0.042549	0.051791	0.060908	0.127986	0.280915
5000	0.103862	0.125188	0.146058	0.349823	0.747921
Promedio	0.036512	0.043956	0.051688	0.119797	0.253813
Speedup	1.103390	1.283783	1.594490	3.169536	5.598759

iii. Gráfico 1. Valor de Speed Up en función de N de las dos máquinas



5. Conclusiones

En base al análisis exhaustivo de los resultados obtenidos en ambas plataformas computacionales y las diversas configuraciones de paralelismo implementadas, podemos establecer conclusiones significativas sobre el rendimiento del algoritmo de Jacobi para la ecuación de Poisson 1D.

OpenMP demuestra consistentemente un rendimiento superior frente a pthreads, especialmente cuando se trabaja con problemas de dimensiones medianas a grandes. Este comportamiento se evidencia por un speedup relativo que supera la unidad en la mayoría de los escenarios con mallas de 50,000 puntos o más. La diferencia es particularmente notable en la Máquina 2 (equipada con procesador Intel i9), donde se alcanzaron factores de aceleración de hasta 5.6 veces para mallas de 1,000,000 puntos.

La mejora de rendimiento exhibe una correlación directa con el tamaño del problema (N). Para valores reducidos de N (10,000), el costo operacional inherente a la paralelización apenas se compensa con la ganancia en velocidad de procesamiento. Sin embargo, a medida que la carga computacional se incrementa ($\geq$500,000 puntos), los costos de sincronización se diluyen proporcionalmente, permitiendo que OpenMP alcance su máxima eficiencia y ventaja comparativa.

Nuestros experimentos confirman la existencia de un límite natural de escalabilidad determinado por la arquitectura física del hardware. En ambos equipos de prueba, al exceder el número de núcleos físicos disponibles (6 en la Máquina 1 y 24 en la Máquina 2), el rendimiento se estabiliza o incluso decrece ligeramente. Este fenómeno revela la presencia de contención por recursos compartidos y sobrecarga adicional generada por la gestión de hilos redundantes.

La determinación del número óptimo de hilos emerge como un factor crítico que debe alinearse precisamente con las características del hardware subyacente. Los resultados de benchmarking demuestran que utilizar tantos hilos como núcleos físicos disponibles maximiza tanto la transferencia de memoria como los eventos procesados por segundo. En contraste, sobredimensionar el número de hilos no solo no aporta beneficios adicionales sino que puede deteriorar el throughput general del sistema.

Los análisis comparativos de las bibliotecas de paralelismo sugieren una recomendación clara para implementaciones futuras: adoptar OpenMP configurado con un número de hilos equivalente al número de núcleos físicos, complementado con opciones de compilación optimizadas (-O3 y ajustes específicos de arquitectura). Esta configuración permite aprovechar más eficientemente las capacidades de vectorización y localidad de datos que ofrecen los procesadores modernos.

Adicionalmente, observamos que la diferencia arquitectónica entre ambas máquinas influye significativamente en los patrones de rendimiento. La Máquina 2, con su mayor número de núcleos y arquitectura más reciente, no solo proporciona un rendimiento absoluto superior, sino que también muestra una mejor escalabilidad con OpenMP en comparación con pthreads. Esto sugiere que las optimizaciones internas de OpenMP están mejor adaptadas para aprovechar las características avanzadas de procesadores de última generación.

Estos hallazgos refuerzan la idoneidad de utilizar OpenMP como solución preferente para paralelizar el método de Jacobi en entornos multicore contemporáneos, siempre que se dimensionen adecuadamente los recursos de paralelismo según la arquitectura disponible y se implementen las optimizaciones de compilación apropiadas para maximizar tanto el speedup como la eficiencia en el uso de memoria.

6. **Bibliografía**

Algoritmo de Jacobi

- Burden, R. L., & Faires, J. D. (2011). *Numerical Analysis* (9th ed.). Brooks/Cole, Cengage Learning, pp. 516-521.
- Quarteroni, A., Sacco, R., & Saleri, F. (2007). *Numerical Mathematics* (2nd ed.). Springer-Verlag, pp. 182-187.

High Performance Computing

- Hager, G., & Wellein, G. (2010). *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Taylor & Francis Group.
- Sterling, T., Anderson, M., & Brodowicz, M. (2018). *High Performance Computing: Modern Systems and Practices*. Morgan Kaufmann.
- Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., & White, A. (2003). *Sourcebook of Parallel Computing*. Morgan Kaufmann Publishers.

Complejidad Computacional

- Sipser, M. (2012). *Introduction to the Theory of Computation* (3rd ed.). Cengage Learning.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press, pp. 43-52.
- Arora, S., & Barak, B. (2009). *Computational Complexity: A Modern Approach*. Cambridge University Press.

Programación Paralela

- Pacheco, P. (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann.
- McCool, M. D., Robison, A. D., & Reinders, J. (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann.

- Mattson, T., Sanders, B., & Massingill, B. (2004). *Patterns for Parallel Programming*. Addison-Wesley Professional.

Hilos y Speedup

- Herlihy, M., & Shavit, N. (2012). *The Art of Multiprocessor Programming* (Revised ed.). Morgan Kaufmann.
- Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). *Introduction to Parallel Computing* (2nd ed.). Addison-Wesley, pp. 120-138.
- Amdahl, G. M. (1967). "Validity of the single processor approach to achieving large scale computing capabilities." *AFIPS Spring Joint Computer Conference*, pp. 483-485.
- Gustafson, J. L. (1988). "Reevaluating Amdahl's Law." *Communications of the ACM*, 31(5), 532-533.

Ecuación de Poisson y Métodos Numéricos

- Strikwerda, J. C. (2004). *Finite Difference Schemes and Partial Differential Equations* (2nd ed.). Society for Industrial and Applied Mathematics.
- Trefethen, L. N. (2000). *Spectral Methods in MATLAB*. Society for Industrial and Applied Mathematics.
- LeVeque, R. J. (2007). *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*. Society for Industrial and Applied Mathematics.