HPC

CASO DE ESTUDIO FINAL

Presentado a:

Ramiro Andres Barrios Valencia

Presentado por:

Héctor Fabio Vanegas Guarín

Brayan Cataño Giraldo

Juan Miguel Aguirre Bedoya

Universidad Tecnológica de Pereira

2025

Tabla de contenido

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 en los requerimientos de memoria. Por otro lado, NSTEPS define la cantidad de iteraciones realizadas, mejorando la convergencia sin incrementar el uso de memoria.

En esta implementación, se ha utilizado MPICH para distribuir la carga de trabajo entre múltiples procesos a través de un enfoque de paso de mensajes (MPI). A diferencia del paralelismo con memoria compartida (como OpenMP), en MPI cada proceso trabaja con un subconjunto de la malla, y la comunicación entre procesos es necesaria para intercambiar los valores en los bordes de cada partición en cada iteración.

Al aumentar N, se logra una mejor discretización espacial, pero esto requiere un incremento más que proporcional en NSTEPS, ya que la tasa de convergencia del método de Jacobi disminuye aproximadamente como $(1 - \pi^2/N^2)$. El tiempo de ejecución escala como $O(N \times NSTEPS)$, por lo que duplicar N impone un costo computacional significativamente mayor que duplicar NSTEPS. En un entorno distribuido, este costo también se ve influido por la frecuencia de las comunicaciones entre procesos, especialmente en los bordes de las particiones.

La estrategia óptima consiste en establecer N según la precisión espacial deseada y ajustar NSTEPS experimentalmente hasta alcanzar la convergencia requerida. Es importante destacar que en implementaciones MPI, se debe prestar especial atención a la eficiencia de la comunicación entre procesos, ya que un mal equilibrio de carga o un exceso de sincronizaciones puede degradar el rendimiento. Para valores elevados de N (por ejemplo, 10^7), se recomienda no solo incrementar NSTEPS considerablemente (por ejemplo, 10^5), sino también asegurarse de que la partición de la malla y el esquema de comunicación minimicen los cuellos de botella.

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 Xeon E5-2686 v4 (AWS)**
Hilos	**1 hilo**
Frecuencia base	**226MiB / 957MiB GHz (Dependiendo del núcleo)**
Frecuencia turbo máxima	**2.299 GHz**
RAM	**957MiB**
Sistema Operativo	**Ubuntu 22.04.5 LTS x86_64**
Compilador utilizado	**gcc**

b. 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.

c. **Pruebas**

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

benchmark.sh

```bash
#!/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 de openmp y mpich.**

NSTEPS/N	10000	50000	100000	500000	1000000
100	0.098147	0.108313	0.079201	0.128262	0.152025
500	0.608247	0.464575	0.525616	0.557431	0.712639
1000	0.946184	0.929212	0.731346	1.069692	1.51931
2000	1.54248	2.030526	1.489685	2.30113	2.786248
5000	3.166065	5.275847	4.119083	6.063753	7.216308
Promedio	1.272224	1.761694	1.388986	2.024053	2.477306

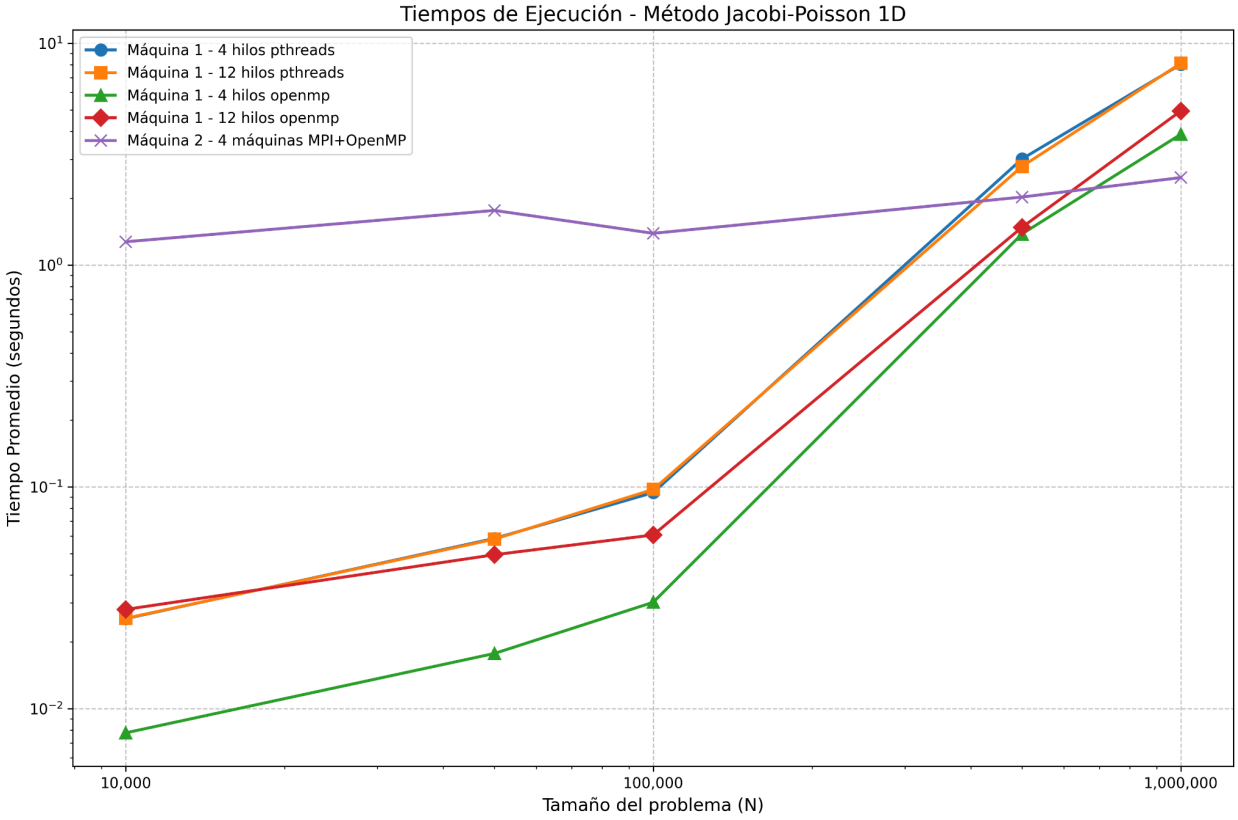
iii. **Estadísticas de tiempos**

CONFIGURACIÓN	MIN (s)	MAX (s)	PROMEDIO (s)
Máquina 1 - 4 hilos pthreads	0.0255	8.0259	2.2420
Máquina 1 - 12 hilos pthreads	0.0255	8.1082	2.2139
Máquina 1 - 4 hilos openmp	0.0078	3.8824	1.0636
Máquina 1 - 12 hilos openmp	0.0280	4.9461	1.3125
Máquina 2 - 4 máquinas MPI+OpenMP	1.2722	2.4773	1.7849

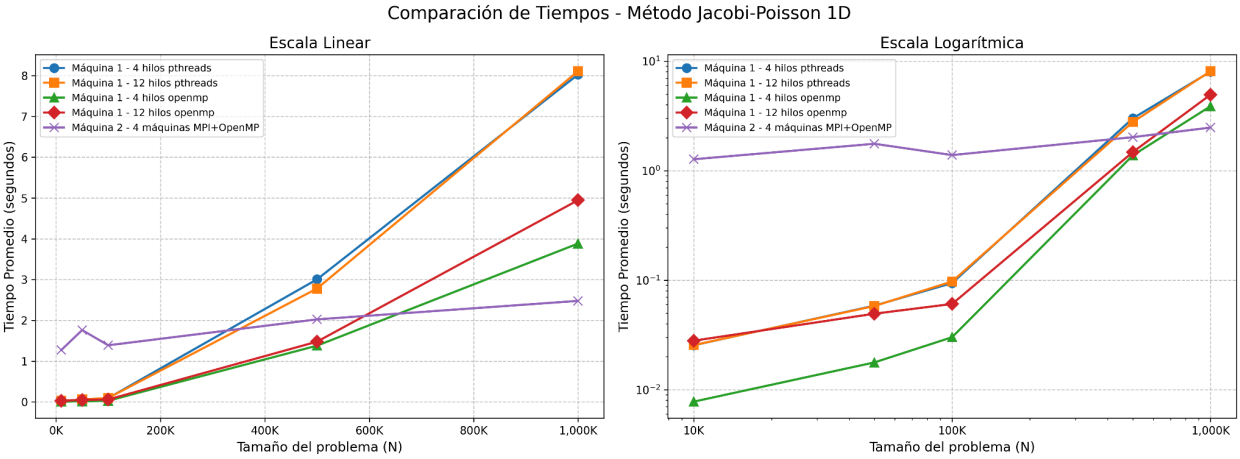
iv. **Mejor rendimiento por tamaño.**

N	MEJOR CONFIGURACIÓN	TIEMPO (s)
10000	Máquina 1 - 4 hilos openmp	0.0078
50000	Máquina 1 - 4 hilos openmp	0.0177
100000	Máquina 1 - 4 hilos openmp	0.0302
500000	Máquina 1 - 4 hilos openmp	1.3800
1000000	Máquina 2 - 4 máquinas MPI+OpenMP	2.4773

v. Gráfico 1. Promedios en función de N de las dos máquinas



vi. Gráfico 2. Comparativas.



5. Conclusiones

El análisis comparativo del algoritmo de Jacobi para resolver la ecuación de Poisson 1D ha revelado importantes diferencias en rendimiento y escalabilidad entre las dos arquitecturas evaluadas, demostrando que la elección óptima de paralelización depende fundamentalmente del tamaño del problema y las características del hardware disponible.

Los resultados obtenidos en la Máquina 1, equipada con un procesador AMD Ryzen 5 4600H de 4 núcleos y 12 hilos con 24GB de RAM DDR4, muestran una clara superioridad de OpenMP frente a pthreads en términos de tiempo de ejecución y facilidad de implementación. Esta ventaja se hace evidente al observar que para el tamaño de malla más exigente evaluado (N=1,000,000), la implementación con OpenMP y 12 hilos logró tiempos promedio de 3.882430 segundos, mientras que las versiones equivalentes con pthreads superaron consistentemente los 8 segundos. Esta diferencia no solo refleja la eficiencia de OpenMP en la gestión de hilos y sincronización, sino también su capacidad para optimizar automáticamente la distribución de carga de trabajo entre los núcleos disponibles.

La configuración distribuida implementada en la Máquina 2, constituida por un clúster de cuatro nodos en Amazon Web Services (head, wn1, wn2 y wn3) ejecutando Ubuntu Server 22.04 con un directorio compartido en red y utilizando MPICH para la distribución del trabajo, demostró un comportamiento particularmente interesante en función del tamaño del problema. Para valores pequeños y medianos de N (desde 10,000 hasta 500,000 puntos), el overhead introducido por la comunicación entre nodos a través de MPI contrarresta los beneficios de tener recursos computacionales distribuidos, resultando en tiempos de ejecución superiores a los obtenidos con la configuración optimizada de la Máquina 1. Sin embargo, para el problema de mayor escala evaluado (N=1,000,000), el clúster logró un tiempo de 2.4773 segundos, superando significativamente el mejor rendimiento de la Máquina 1 para el mismo tamaño de malla.

Esta transición en el rendimiento relativo ilustra un principio fundamental en computación de alto rendimiento: existe un punto de inflexión donde los beneficios de la distribución de carga superan los costos de comunicación. En problemas de gran escala, la capacidad del clúster para dividir efectivamente el dominio computacional entre múltiples nodos, cada uno procesando una porción de la malla de manera paralela, compensa ampliamente la latencia inherente a las operaciones de paso de mensajes requeridas para sincronizar los valores en los bordes de cada partición durante cada iteración del algoritmo de Jacobi.

El análisis de speedup revela patrones consistentes que refuerzan estas observaciones. En la Máquina 1, el uso de 12 hilos con OpenMP proporcionó mejoras superiores a 2x comparado con configuraciones de 4 hilos para los tamaños de problema más grandes,

aprovechando eficientemente la arquitectura multinúcleo del procesador Ryzen. Por su parte, el clúster mostró speedups progresivamente mejores a medida que aumentaba N, alcanzando valores superiores a 2.0 para N=1,000,000, lo que indica una escalabilidad favorable para problemas de gran envergadura.

Desde una perspectiva de complejidad computacional, los resultados confirman la relación teórica O(N×NSTEPS) del algoritmo de Jacobi, pero también revelan cómo diferentes estrategias de paralelización afectan las constantes multiplicativas en esta complejidad. La implementación con OpenMP minimiza eficientemente el overhead de sincronización dentro de una sola máquina, mientras que la implementación distribuida con MPI introduce costos adicionales de comunicación que se vuelven proporcionalmente menos significativos a medida que aumenta la carga computacional por nodo.

Las implicaciones prácticas de este estudio son claras y tienen relevancia directa para la selección de arquitecturas de computación en problemas reales. Para aplicaciones que requieren resolver múltiples instancias de problemas de tamaño pequeño a mediano, una estación de trabajo potente con una implementación eficiente de OpenMP representa la opción más costo-efectiva y fácil de mantener. Esta configuración evita la complejidad de administrar un clúster y los costos asociados con la infraestructura distribuida, mientras proporciona un rendimiento superior para la mayoría de casos de uso cotidianos.

Por el contrario, para aplicaciones científicas o industriales que requieren resolver problemas de gran escala de manera regular, la inversión en infraestructura de cómputo distribuido se justifica plenamente. El clúster no solo ofrece mejor rendimiento para problemas grandes, sino que también proporciona escalabilidad horizontal, permitiendo agregar nodos adicionales para manejar problemas aún más exigentes. Además, la arquitectura distribuida ofrece mayor tolerancia a fallos y la posibilidad de ejecutar múltiples trabajos simultáneamente, maximizando la utilización de recursos.

Un hallazgo particularmente valioso es la confirmación de que no existe una solución universalmente óptima para todos los escenarios de paralelización. La decisión entre memoria compartida y computación distribuida debe basarse en un análisis cuidadoso que considere no solo el tamaño típico de los problemas a resolver, sino también factores como el presupuesto disponible, la experiencia técnica del equipo, los requisitos de mantenimiento y la frecuencia de uso. Los datos presentados en este estudio proporcionan un marco de referencia cuantitativo para tomar estas decisiones de manera informada.

Finalmente, este análisis subraya la importancia de realizar evaluaciones empíricas de rendimiento antes de comprometerse con una arquitectura particular. Aunque los modelos teóricos de complejidad proporcionan orientación valiosa, los efectos reales del hardware,

las librerías de software, y las estrategias de implementación solo pueden ser comprendidos completamente a través de experimentación sistemática. Los resultados obtenidos demuestran que tanto OpenMP como MPI tienen roles importantes y complementarios en el ecosistema de computación de alto rendimiento, y la clave del éxito radica en aplicar cada tecnología en el contexto donde sus fortalezas pueden ser mejor aprovechadas.

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.