HPC

CASO DE ESTUDIO II

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 tres formas: secuencial, paralela con hilos, y paralela con memoria compartida, 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 de forma secuencial y luego implementando paralelismo con diferentes números de hilos (2, 4, 8, 12) utilizando la librería pthread. Contrario a lo esperado, las implementaciones paralelas no mostraron mejoras significativas sobre la versión secuencial, debido a que el problema está limitado por acceso a memoria más que por capacidad de cómputo. Con base en estas pruebas se han identificado alternativas para mejorar el rendimiento como vectorización, técnicas de blocking y algoritmos alternativos.

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

Los hilos representan unidades básicas de ejecución dentro de un proceso, permitiendo la concurrencia en arquitecturas multinúcleo. En implementaciones paralelas del método de Jacobi, los hilos pueden utilizarse para distribuir el procesamiento de diferentes segmentos de la malla entre los núcleos disponibles.

El speedup mide la mejora en tiempo de ejecución al utilizar múltiples unidades de procesamiento comparado con la ejecución secuencial. Idealmente, el speedup sería lineal (un factor N de mejora al utilizar N procesadores), pero en la práctica está limitado por:

Overhead de comunicación: Tiempo dedicado a sincronizar datos entre hilos.

Secciones secuenciales: Partes del algoritmo que no pueden paralelizarse (Ley de Amdahl).

Contención de recursos: Competencia por recursos compartidos como memoria o ancho de banda.

Para el método de Jacobi, el potencial de speedup es significativo debido a su alta paralelizabilidad, especialmente para valores grandes de N donde el costo de comunicación se amortiza frente al procesamiento realizado por cada hilo.

4. Marco Contextual

a. Características de la máquina

Para la realización de este laboratorio, se hizo uso de un dispositivo de cómputo, el cual consta 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 22.04.5 LTS**

b. Desarrollo

En el desarrollo de esta actividad, comenzamos implementando el algoritmo de Jacobi para resolver la ecuación de Poisson 1D, explorando diferentes enfoques desde una versión secuencial hasta implementaciones paralelas con el fin de comparar su rendimiento y analizar las mejoras de velocidad (speedup) obtenidas mediante diversas técnicas de optimización.

Inicialmente, nos enfocamos en la versión secuencial del algoritmo para establecer una base de referencia para comparaciones posteriores. Implementamos el método de Jacobi siguiendo su formulación clásica, donde en cada iteración se actualiza cada punto de la malla utilizando los valores de los puntos vecinos de la iteración anterior.

Durante esta fase de pruebas, descubrimos que a pesar de las limitaciones teóricas respecto al parámetro N, si se utiliza un valor relativamente bajo de NSTEPS, el consumo de memoria RAM no aumenta drásticamente, lo que nos permitió explorar rangos más amplios de valores para ambos parámetros. Las pruebas realizadas fueron con N = {10000, 50000, 100000, 500000, 1000000} y con NSTEPS = {100, 500, 1000, 2000,

5000}. Estas combinaciones nos permitieron establecer una relación clara entre el tamaño del problema, el número de iteraciones y el tiempo de ejecución.

Avanzando hacia implementaciones concurrentes, identificamos 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"
```

i. **Tabla 1: Resultados de la ejecución secuencial (archivo original-jacobi1d.c):**

NSTEPS/N	10000	50000	100000	500000	1000000
100	0.001005	0.005101	0.010328	0.152036	0.279111
500	0.004734	0.021922	0.046324	0.706279	1.372700
1000	0.009185	0.043294	0.090367	1.359130	2.076210
2000	0.018433	0.086442	0.175192	2.328000	3.751720
5000	0.048251	0.210616	0.421619	4.419940	7.624980
Promedio	0.016321	0.073475	0.148766	1.793077	3.020944

ii. **Tabla 2: Resultados de la ejecución con 2 hilos.**
A partir de aquí aparece la fila de speedup, en la cual está representada la cantidad de veces que se mejora la ejecución al correr el código con varios hilos.

NSTEPS/N	10000	50000	100000	500000	1000000
100	0.002745	0.005580	0.008939	0.204985	0.495860
500	0.006670	0.021886	0.034342	0.834577	2.340170
1000	0.015773	0.036177	0.069823	2.119660	4.568860
2000	0.032718	0.068154	0.113362	4.389290	9.464190
5000	0.073574	0.153281	0.306964	10.364000	22.353300
Promedio	0.034687	0.057015	0.106686	3.582502	7.844476
Speedup	0.470522	1.288695	1.394428	0.500509	0.385104

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

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
Speedup	0.640994	1.255446	1.579055	0.596547	0.376401

iv. **Tabla 4: Resultados de la ejecución con 8 hilos.**

NSTEPS/N	10000	50000	100000	500000	1000000
100	0.002252	0.004628	0.008349	0.209652	0.547142
500	0.006599	0.017271	0.032598	0.936077	2.452160
1000	0.016002	0.034530	0.062371	2.249050	4.988740
2000	0.031455	0.064682	0.148556	4.087440	10.127800
5000	0.067361	0.159901	0.336135	11.266500	23.364600
Promedio	0.024733	0.281012	0.117601	3.749743	8.296088
Speedup	0.659887	0.261465	1.265006	0.478186	0.364140

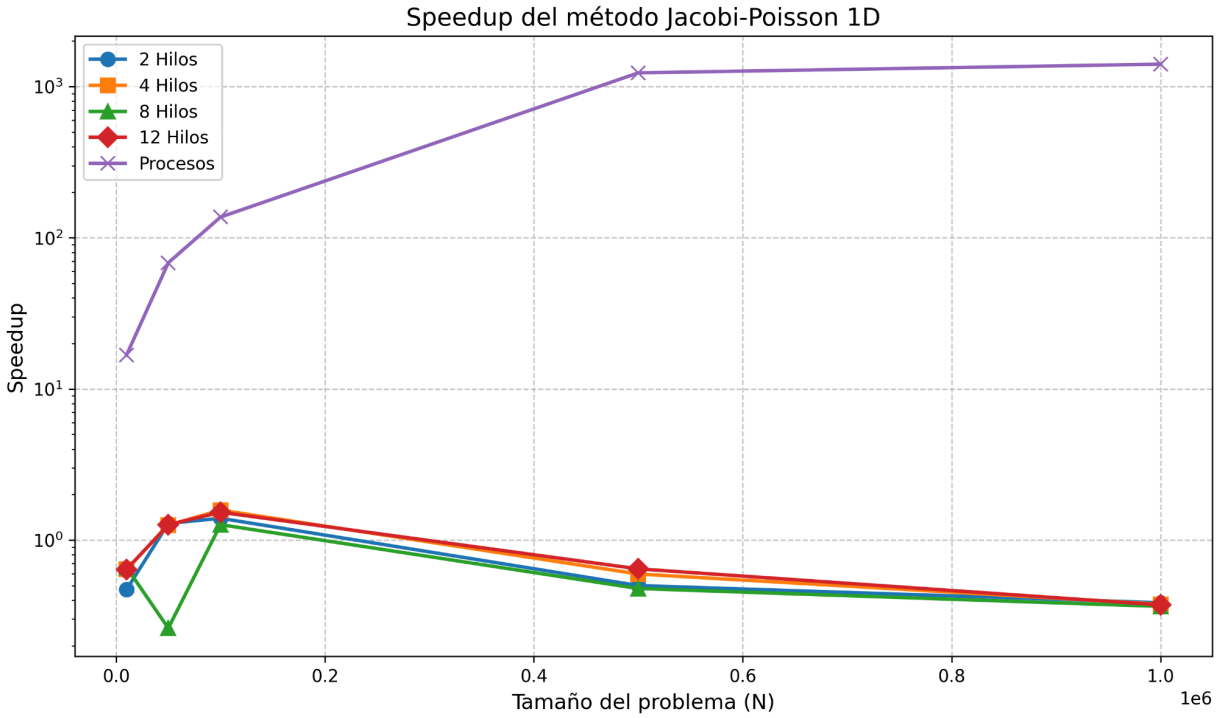
v. **Tabla 5: Resultados de la ejecución con 12 hilos.**

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
Speedup	0.638786	1.264673	1.528077	0.644961	0.372577

vi. **Tabla 7: Resultados de la ejecución por procesos.**

NSTEPS/N	10000	50000	100000	500000	1000000
100	0.000971	0.001075	0.001209	0.002010	0.001968
500	0.000987	0.001397	0.001241	0.001200	0.001310
1000	0.000924	0.000967	0.001037	0.001319	0.002640
2000	0.001020	0.001001	0.000965	0.001223	0.002001
5000	0.000972	0.000937	0.000977	0.001513	0.002806
Promedio	0.000974	0.001075	0.001085	0.001453	0.002145
Speedup	16.756673	68.348837	137.111520	1234.051617	1408.365501

vii. Gráfico 1. Valor de Speed Up en función de N



5. Conclusiones

Tras analizar la implementación con hilos y memoria compartida, se concluye que esta no mejora el rendimiento en comparación con la versión secuencial. Luego de un análisis, con ayuda de inteligencias artificiales, se llega a la conclusión que esto puede tener varios motivos. En primer lugar, el método de Jacobi 1D realiza pocas operaciones aritméticas por punto de la malla, por lo que la mayor parte del tiempo se consume en acceder a memoria y no en realizar cálculos, lo que limita el beneficio de la paralelización. Además, la sincronización entre hilos introduce una sobrecarga adicional, ya que cada hilo debe esperar a los demás antes de continuar, lo que puede anular las ventajas de la ejecución paralela. También se observa que el uso de memoria compartida mediante shm_open y mmap genera una complejidad innecesaria, dado que la memoria heap ya es compartida por defecto entre los hilos de un mismo proceso. Otro factor que afecta negativamente el rendimiento es el falso compartir (false sharing), que ocurre cuando varios hilos actualizan datos en la misma línea de caché, causando invalidaciones constantes y reduciendo la eficiencia. Además, el patrón de acceso a memoria en Jacobi 1D no es óptimo, ya que los hilos necesitan acceder a los bordes de los segmentos de otros hilos, lo que dificulta la eficiencia en el uso de caché. En conclusión, la paralelización en una dimensión no aporta mejoras sustanciales a menos que N sea extremadamente grande. Para obtener una verdadera ganancia de rendimiento, es preferible aplicar paralelización en problemas 2D o 3D, donde la carga computacional es mayor en relación con los accesos a memoria.

En contraste, la versión basada en procesos supera a la de hilos en Jacobi-Poisson 1D debido a varias ventajas específicas. En primer lugar, el aislamiento de memoria que ofrecen los procesos reduce problemas de coherencia de caché, ya que cada proceso tiene su propio espacio de direcciones y solo las regiones explícitamente compartidas requieren sincronización. Además, al minimizar la cantidad de memoria compartida, se reduce el falso compartir, lo que evita invalidaciones constantes de líneas de caché y mejora el rendimiento. Otro beneficio clave es la mejor localidad de caché, ya que cada proceso mantiene sus propias cachés para variables locales y pilas, optimizando el acceso a memoria para operaciones que no requieren datos compartidos. También hay menor contención de recursos, ya que el uso limitado de memoria compartida disminuye la competencia entre procesos por caché y memoria. Desde el punto de vista de la implementación, el uso eficiente de mmap con MAP_SHARED | MAP_ANONYMOUS permite que solo se creen las regiones necesarias para compartir, evitando sobrecargas innecesarias. Además, la barrera de sincronización basada en semáforos puede ser más eficiente que pthread_barrier en este tipo de patrones de trabajo. Por último, el planificador del kernel puede asignar prioridades más adecuadas a procesos independientes en comparación con hilos dentro de un mismo proceso, mejorando el uso de la CPU y el comportamiento de la Translation Lookaside Buffer (TLB). En

conclusión, para el método de Jacobi 1D, donde la relación entre accesos a memoria y operaciones computacionales es alta, el uso de procesos proporciona ventajas de aislamiento y eficiencia que superan las desventajas de la creación inicial, especialmente en escenarios con sincronización frecuente y patrones de acceso a memoria que generan contención entre hilos.

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.