Programación Multinúcleo y extensiones SIMD

Asier Cabo Lodeiro, Martin Castro Vázquez

Arquitectura de Computadores
Grupo 47
{asier.cabo,martin.castro.vazquez}@rai.usc.es

Este informe describe la implementación y optimización global del método iterativo de Jacobi para resolver sistemas de ecuaciones lineales. Se desarrollaron cuatro versiones: una base secuencial, optimización de caché (fusión y desenrollado de bucles, blocking), vectorización SIMD con AVX256 y paralelismo OpenMP. Se evaluó el rendimiento en el Finisterrae III del CESGA en matrices de tamaño n=250, 2500 y 5000, tomando la mediana de 15 ejecuciones y comparando speedups frente a compilaciones -O0 y -O3. Los resultados muestran aceleraciones máximas de ≈ 14.7× con optimización de caché, ≈ 18.6× con SIMD y ≈ 58.4× con OpenMP. Se concluye que optimización y paralelismo explotan eficazmente la arquitectura multinúcleo y SIMD.

Método de Jacobi, optimización de caché, speedup de rendimiento.

I. Introducción

El método de Jacobi es un algoritmo iterativo clásico para resolver sistemas de ecuaciones lineales de la forma Ax=b, ampliamente utilizado en ingeniería y ciencias aplicadas. Aunque su sencillez lo hace atractivo, su eficiencia depende críticamente del uso óptimo de la memoria caché y de las capacidades de paralelismo modernas. Este trabajo se basa en los fundamentos teóricos de optimización de arquitecturas de memoria descritos por Hennessy y Patterson [1], en la guía de intrinsics AVX de Intel [2] para explotar unidades SIMD y técnicas de paralelismo en memoria compartida.

El objetivo principal es implementar cuatro versiones del algoritmo de Jacobi en lenguaje C:

- Secuencial "base", siguiendo el pseudocódigo original.
- Secuencial optimizado con técnicas de fusión y desenrollado de bucles, blocking y reordenación de accesos para mejorar la localidad de caché.
- Vectorizada con AVX256, empleando intrinsics para paralelismo a nivel de datos.
- Paralela con OpenMP, explotando múltiples hilos y distintas estrategias de reparto de carga.

Para cada versión se midió el tiempo de cómputo en matrices de tamaño n=250, 2500 y 5000, tomando la mediana de quince ejecuciones para garantizar robustez estadística. Se evaluaron los speedups relativos a compilaciones con −O0 y −O3, así como las ganancias absolutas entre implementaciones.

El resto del documento se organiza así:
Sección II describe la metodología de implementación y los detalles experimentales.
Sección III presenta los resultados obtenidos y un análisis comparativo de rendimiento.
Sección IV expone las conclusiones y propuestas de trabajo futuro.

II. Implementación y experimentación

Para abordar la evaluación comparativa de las cuatro versiones del método de Jacobi implementadas (v1, v2, v3, v4), esta sección se estructura en tres apartados: descripción del entorno de ejecución, detalle de la implementación y metodología de benchmarking.

A. Entorno de ejecución

Los experimentos se llevaron a cabo en el supercomputador FinisTerrae III del CESGA, cuyos principales parámetros se resumen en la Tabla I.

TABLA I
VALORACIÓN DE ESPECIFICACIONES TÉCNICAS [3]

Procesador	Intel Xeon Ice Lake 8352Y, 2.2 GHz
Núcleos	22656
Nodos	354
Memoria	118TB
SO	SUSE Linux Enterprise
Compilador	GCC `10.1.0`
Caché L1	49152 bytes
Caché L2	1310720 bytes
Línea Caché	64 bytes

Como podemos observar, si hacemos el sencillo cálculo del número de núcleos por nodo, sale que cada nodo tiene 64 cores. Por ello, para evitar que haya interferencias de planificaciones con otros trabajos, por lo que pedimos siempre 64 cores para que nos asignen un nodo completo.

De estos datos, también es extremadamente relevante el tamaño de línea caché, ya que será necesario conocerlo especialmente en la v3, con la implementación de paralelismo a nivel de datos.

B. *Implementación de las versiones*

Para la implementación se toma como base el pseudocódigo del método Jacobi proporcionado (ver Figura 1). A partir de este, se procede a la programación de 4 variantes en C estándar, nombradas v1.c, v2.c, v3.c, v4.c, respectivamente.

La primera versión, consiste en una implementación directa del algoritmo, sin ningún tipo de optimización.

FIGURA I
PSEUDOCÓDIGO BASE DEL MÉTODO JACOBI

```
Entradas:
a: Matriz de coeficientes del sistema (float[n x n])
b: Vector de términos independientes (float[n])
x: Vector solución (float[n])
tol: Tolerancia para la convergencia (float)
max_iter: Número máximo de iteraciones (int)

Variables auxiliares:
x_new: Vector nueva solución (float[n])

Cómputo:
Para iter (int) desde 0 hasta iter:
    norm2 = 0: norma del vector al cuadrado (float)

    Para i (int) desde 0 hasta n:
        sigma = 0.0 (float)
        Para j desde 0 hasta n:
            Si i ≠ j:
                sigma += a[i][j] * x[j]
        x_new[i] = (b[i] - sigma) / a[i][i]
        norm2 += (x_new[i] - x[i]) ^ 2

    x = x_new
    Si sqrt(norm2) < tol:
        Terminar
Salida:
Imprimir valor de norm2
```

Para la implementación del método de Jacobi se tomó como referencia el pseudocódigo clásico del algoritmo, representado en la Figura 1. Este se empleó como punto de partida para el desarrollo de la versión base (v1), sobre la cual se aplicaron posteriormente distintas optimizaciones.

El pseudocódigo implementa el algoritmo de Jacobi [4] para resolver sistemas lineales *Ax=b* mediante un procedimiento iterativo. En cada iteración, se calcula una nueva estimación del vector solución *x*, actualizando cada componente *xi* a partir de la ecuación del sistema correspondiente.

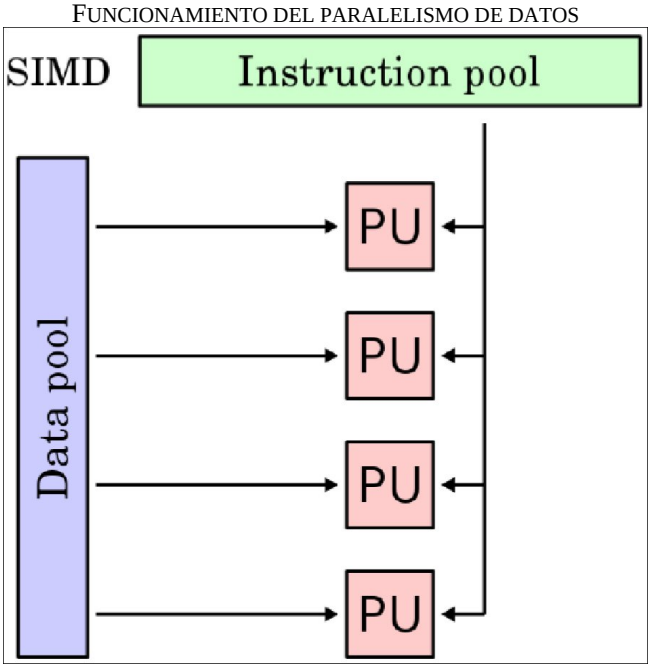
Esto implica que el nuevo valor de *xi* depende exclusivamente de los valores anteriores del resto de componentes *xj* (con *j* distinto de *i*).

Durante cada paso del algoritmo:

1. Se calcula, para cada fila *i*, la suma ponderada de los elementos de *x* distintos de *xi*,
2. Con este resultado, se obtiene el nuevo valor de *xi*.
3. Se acumula el cuadrado de la diferencia entre el nuevo valor y el anterior para calcular la norma del cambio.
4. Al final de cada iteración, se evalúa si el vector solución ha convergido, es decir, si la raíz cuadrada de la norma calculada es menor que una tolerancia prefijada, en nuestro caso 10e-8. Si es así, el proceso finaliza.
5. En caso contrario, se copia el nuevo vector *xnew* sobre *x* y se repite el proceso hasta alcanzar la convergencia o el número máximo de iteraciones permitido.

Este procedimiento permite aproximar la solución del sistema lineal sin necesidad de invertir la matriz *A*, y es especialmente adecuado para matrices dispersas y diagonales dominantes.

FIGURA II
FUNCIONAMIENTO DEL PARALELISMO DE DATOS



La segunda versión aplica optimizaciones secuenciales centradas en la mejora del uso de caché. Se evaluaron cuatro estrategias: reducción de instrucciones, fusión y desenrollado de lazos, y operaciones por bloques. Se realizarán pruebas individuales y combinadas para seleccionar la versión óptima.
Para la tercera versión, se utiliza el procesamiento vectorial SIMD. Este es un modelo de paralelismo que permite operar varios datos de forma simultanea con una única instrucción, como se muestra en la Figura II. Es decir, en cada iteración *c[i]=a[i]+b[i]* en vez de operar únicamente con el elemento *i*, se opera con los elementos desde *i* hasta *i+x*, siendo *x* el tamaño del paralelismo, ahorrando *x* iteraciones del bucle. Para estas operaciones, es indispensable el alineamiento correcto de memoria.

Para la última versión (v4) se emplea paralelismo de memoria compartida mediante OpenMP, aprovechando la directiva *#pragma omp parallel for* para distribuir las iteraciones del bucle principal entre múltiples hilos y sincronizarlos en cada paso. Se realizarán pruebas con diferentes números de hilos, en el rango (1,64), dado que, como se menciona en el apartado II.A, con apoyo de la Tabla I, los nodos del CESGA están compuestos de 64 cores, por lo que a partir de dicho número, los hilos se empezarán a pelear por la CPU, produciendo una disminución de rendimiento.

Tras los experimentos, se seleccionará el número de hilos óptimo.

C. *Metodología de benchmarking*

Para cada una de las versiones se ejecutaron pruebas con 3 tamaños diferentes de matriz, n={250, 2500, 5000}, fijando una tolerancia de 10^{-8}, y un máximo de iteraciones de 20000. Cada combinación de versión y tamaño, fue repetida 15 veces, en las cuales se registraron el número de ciclos de CPU de la sección de cómputo puro, es decir, excluyendo la generación y la impresión de los datos. Con las medidas obtenidas, se calculó una mediana que permite una mayor fiabilidad de los datos.

Para realizar este proceso, se usaron scripts de Bash que permitieron automatizar la solicitud de recursos al CESGA, (64 cores y 64GB de memoria) y la posterior ejecución de cada una de las versiones, iterando sobre el número de elementos de la matriz, el número de hilos usados (de ser el caso) y cada una de las 15 iteraciones.

En la siguiente versión se presentan tablas y gráficas con los resultados y su interpretación detallada.

III. Resultados y análisis

En esta sección se presentan los resultados de los experimentos realizados para evaluar el rendimiento de las distintas versiones del método de Jacobi. Todas las ejecuciones se realizaron en el supercomputador FinisTerrae III del CESGA, tomando 15 mediciones del número de ciclos y calculando su mediana, siguiendo la metodología descrita en la Sección II.

A. *Comparativa entre versiones*

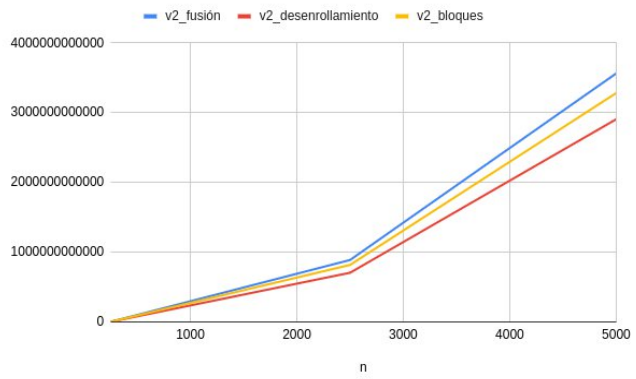
En primer lugar se realizará una comparativa entre diferentes modificaciones dentro de cada versión, para seleccionar la modificación óptima. Se compararán, las diferentes configuraciones de v2 y el número de hilos usados en v4.

En la Figura III se muestra una comparativa entre distintas configuraciones evaluadas en el proceso de optimización de la versión 2. Se observa una mejora progresiva en el rendimiento a medida que se incorporan técnicas como la reducción de instrucciones, la fusión de lazos y su desenrollado. La mayor eficiencia se alcanza con la combinación de estas tres optimizaciones. Sin embargo, la inclusión de operaciones por bloques (blocking) provocó una pérdida de rendimiento, especialmente en tamaños de problema intermedios y pequeños.

Por este motivo, se descartó el uso de blocking al comprobar que introducía penalizaciones en lugar de mejoras. En contraste, la combinación de las otras tres técnicas mostró beneficios consistentes respecto a la versión base, particularmente para tamaños de matriz grandes. En consecuencia, esta configuración fue adoptada como la versión secuencial final optimizada utilizada en los experimentos posteriores.

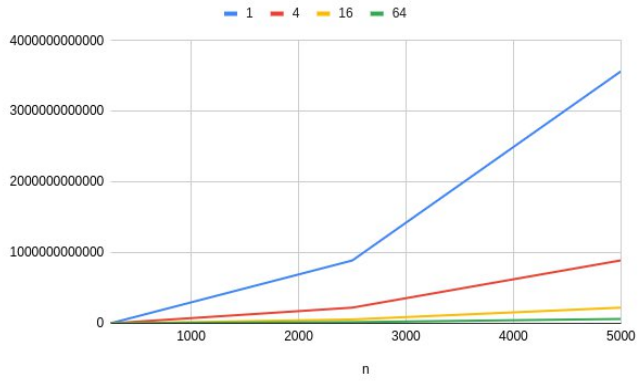
En la Figura IV se muestra el número de ciclos de CPU registrados al variar el número de hilos de ejecución de la versión 4. Se observa una mejora de rendimiento al incrementar el grado de paralelismo: para n=5000, el tiempo de cómputo desciende de 3.57×10^{12} a 6.11×10^{10} ciclos al pasar de 1 a 64 hilos, lo que equivale a un speedup efectivo superior a 58 veces más rápido frente a la ejecución secuencial.

Figura III
Comparativa de optimizaciones en v2



Dado el significativo beneficio obtenido con 64 hilos y la eficiencia cercana a la ideal en tamaños de problema elevados, el resto de los experimentos se llevarán a cabo utilizando esta configuración de hilos.

Figura IV
Comparativa numero de hilos en v4



Además de evaluar el número de hilos, exploramos varias políticas de reparto de iteraciones y seleccionamos *schedule (static)*, que distribuye de antemano segmentos contiguos iguales entre los hilos, pues resultó ser la opción de menor sobrecarga y mejor rendimiento en nuestro bucle regular. Para la acumulación de la norma, comparamos dos estrategias:

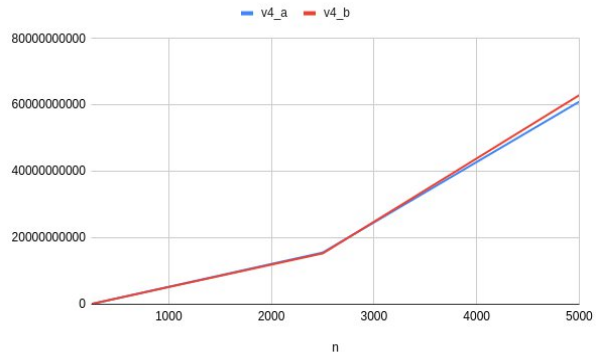
- Cláusula de reducción: cada hilo mantiene una copia privada de la norma durante la ejecución de la región paralela y cuando esta termina se fusionan

todas las copias en una única suma, concentrando la sincronización en un único punto.

- Operaciones atómicas: cada hilo actualiza directamente la variable compartida norm2 de forma atómica sumándole su norma local..

Aunque la reducción suele minimizar la contención al agrupar la sincronización en una fase final, en nuestro caso el uso de *atomic* entregó un rendimiento ligeramente superior, ya que la cantidad de actualizaciones por hilo es relativamente pequeña y el coste de inicializar y combinar copias de reducción supera el de unas pocas operaciones atómicas. Por tanto, adoptamos la versión con *atomic* para la acumulación de la norma en todas las ejecuciones.

FIGURA VII
COMPARATIVA OPTIMIZACIONES EN V4



B. *Tiempos de ejecución*

Tabla II muestra los tiempos de ejecución medianos (en ciclos) de cada versión para los diferentes tamaños de problema.

TABLA II
TIEMPOS DE EJECUCIÓN MEDIANOS (CICLOS)

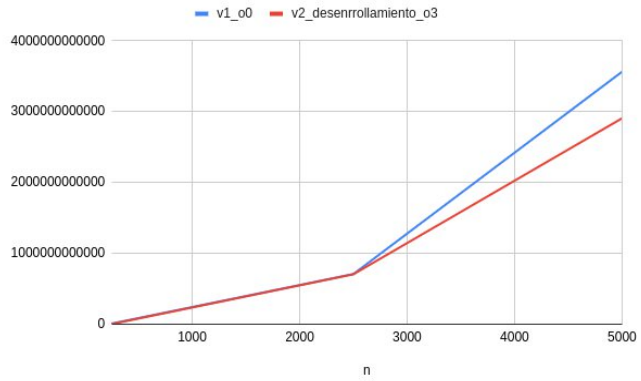
Versión	N=250	N=2500	N=5000
V1 (o0)	9.34e9	7.08e11	3.57e12
V1 (o3)	9.35e9	7.09e11	3.58e12
V2 (opt)	6.36e8	7.04e11	2.91e12
V3 (AVX256)	5.02e8	5.54e11	2.25e12
V4 (64 hilos) A	9.41e7	1.56e10	6.11e10

C. *Evaluación de las optimizaciones secuenciales*

En primer lugar, puede verse en la Figura V una representación del speedup de la versión secuencial optimizada con respecto a la versión base. Se observa una reducción en el número de ciclos empleados, especialmente a medida que crece el tamaño de la matriz. Esta mejora se atribuye directamente a las optimizaciones aplicadas, que reducen la sobrecarga computacional y mejoran la localidad de referencia en memoria. En particular, la fusión de bucles disminuye el número de accesos redundantes, el desenrollado reduce el coste de control de bucle, y la reordenación de accesos mejora el aprovechamiento de las líneas de caché.

Además, los beneficios se amplifican en tamaños grandes (n = 5000), donde la presión sobre la jerarquía de memoria es mayor. Por tanto, se confirma que las optimizaciones aplicadas resultan especialmente eficaces en escenarios donde el coste de acceso a memoria es un factor limitante del rendimiento.

FIGURA V
COMPARATIVA ENTRE VERSIONES 1 Y 2



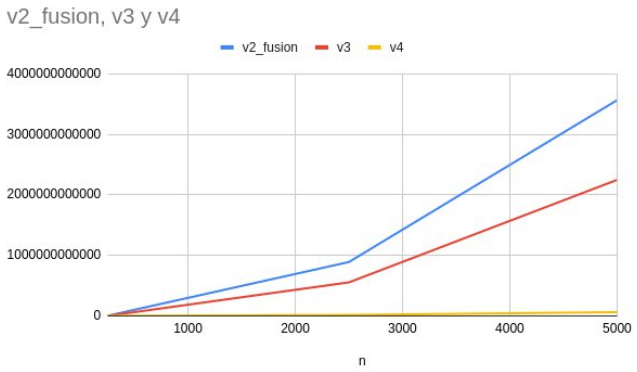
D. *Comparativa entre vectorización y paralelismo OpenMP*

A continuación, en la Figura VI se presenta una comparativa del speedup obtenido por las versiones tercera (vectorizada con AVX256) y cuarta (paralela con OpenMP) respecto a la versión secuencial optimizada (v2). Cabe destacar que ambas variantes parten de la segunda versión, que ya incorpora optimizaciones como la reducción de instrucciones redundantes y la fusión de bucles, lo que sirve como base común de mejora.

Como muestra la gráfica, ambas implementaciones logran reducir de forma notable el número de ciclos necesarios para completar la computación, especialmente en tamaños de problema grandes. La versión vectorizada (v3) presenta mejoras progresivas debido al aprovechamiento del paralelismo a nivel de datos, aunque su impacto depende del correcto alineamiento de memoria y del número de elementos procesables en paralelo. Por su parte, la versión paralela con OpenMP (v4) ofrece los mayores beneficios en entornos multinúcleo, alcanzando speedups superiores al distribuir eficientemente la carga de trabajo entre varios hilos.

Estas diferencias reflejan la naturaleza complementaria de ambos enfoques: mientras AVX acelera el procesamiento dentro de cada hilo, OpenMP mejora la escalabilidad general del algoritmo al emplear múltiples hilos en paralelo.

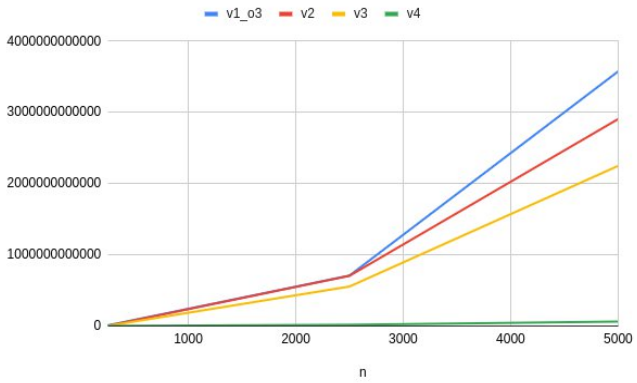
FIGURA VI
COMPARATIVA VERSIONES 2, 3 Y 4



E. Comparación frente a la compilación optimizada -O3

Por último compararemos las versiones desarrolladas en los apartados ii), iii) y iv) respecto de la versión inicial compilada con -O3.

FIGURA VIII
COMPARATIVA DE TODAS LAS VERSIONES



En la Figura VIII se muestran, para cada tamaño de problema y tomando la mediana de 15 ejecuciones, los tiempos de ejecución absolutos (ciclos de CPU) de las versiones v2, v3 y v4, todas compiladas con -O3, comparados con la versión inicial v1 (-O3).

De la gráfica anterior se extraen tres conclusiones clave:

- v2 (-O3) (optimización de caché) aporta una reducción moderada de ciclos respecto a v1 (-O3) (≈ 15–20 %) para n=250 y n=5000, pero el beneficio se diluye ligeramente en tamaños intermedios donde la versión base ya está muy optimizada por el compilador.
- v3 (-O3) (SIMD AVX256) obtiene mejoras adicionales, gracias a la vectorización explícita con AVX256. El beneficio es más visible en bucles con gran cantidad de datos por cada iteración.
- v4 (-O3) (OpenMP) es la versión más rápida en términos absolutos, especialmente para n grandes. Al distribuir el trabajo entre múltiples hilos, reduce los ciclos en dos órdenes de magnitud frente a v1 (-O3) y llega a ser más de 50 veces más rápida que la versión base para n=5000.

En conjunto, esta comparación demuestra que, además de las optimizaciones de caché y la vectorización, el paralelismo a nivel de hilos es imprescindible para explotar todo el potencial de arquitecturas multinúcleo como la del FinisTerrae III.

IV. CONCLUSIONES

En este trabajo se abordó la implementación y optimización del método iterativo de Jacobi para la resolución de sistemas de ecuaciones lineales. Partiendo de un código base secuencial, se desarrollaron tres versiones adicionales que incorporan mejoras progresivas: una optimización del acceso a memoria, una versión vectorizada mediante extensiones SIMD AVX256 y una versión final paralelizada mediante OpenMP.

La evaluación se realizó en el supercomputador FinisTerrae III, comparando el tiempo de ejecución de las distintas versiones para tamaños de problema crecientes (n = 250, 2500, 5000). Se estudiaron los efectos de cada optimización aplicada, observándose diferencias significativas en el rendimiento. A partir de esta experimentación, se destacan las siguientes conclusiones:

La mejora en la gestión del acceso a memoria (versión v2) proporciona una aceleración notable frente a la versión secuencial, especialmente visible en tamaños de matriz grandes, gracias al aprovechamiento de la localidad espacial y temporal de los datos.

• La vectorización con AVX256 (versión v3) permite una aceleración adicional al explotar el paralelismo a nivel de datos. No obstante, su correcta implementación exige una alineación adecuada de memoria y una mayor atención a los detalles técnicos.

• La paralelización mediante OpenMP (versión v4) obtuvo los mejores resultados en matrices de gran tamaño, evidenciando la escalabilidad de esta técnica. En problemas más pequeños, sin embargo, la sobrecarga asociada a la gestión de hilos limita su efectividad.

• La combinación de técnicas (optimizaciones de memoria, vectorización y paralelización) resulta fundamental para obtener el máximo provecho del hardware moderno, permitiendo una ejecución más rápida y eficiente del algoritmo.

Durante el desarrollo se encontraron asimismo retos técnicos como la correcta alineación de memoria para el uso de intrínsecos SIMD, la necesidad de evitar condiciones de carrera en OpenMP, y la adaptación del código al sistema de ejecución automatizada mediante Makefile y scripts.

Entre las posibles líneas de trabajo que se podrían desarrollar a partir de esta práctica se incluyen:

- Implementar versiones del algoritmo con otras bibliotecas de paralelismo (como MPI [5] para entornos distribuidos).
- Evaluar el rendimiento con matrices dispersas o de estructura especial.
- Explorar técnicas de paralelismo híbrido combinando OpenMP y SIMD simultáneamente.
- Incorporar medidas de precisión numérica para comparar estabilidad entre versiones.

REFERENCIAS

[1] J. L. Hennessy y D. A. Patterson, Computer Architecture: A Quantitative Approach, 5.ª ed., Morgan Kaufmann, 2012.

[2] Intel® Intrinsics Guide [Internet]. Intel. [citado 16 de abril de 2025]. Disponible en: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

[3] Tiagonce RG. CESGA's new FinisTerrae III [Internet]. Cesga - Centro de Supercomputación de Galicia. 2023 [citado 10 de abril de 2025]. Disponible en: https://www.cesga.es/en/cesga-actualiza-el-finisterrae/

[4] Wikipedia contributors. Jacobi method [Internet]. Wikipedia. 2025 [citado 12 de abril de 2025]. Disponible en: https://en.wikipedia.org/wiki/Jacobi_method

[5] Web de programación paralela [Internet]. [citado 16 de abril de 2025]. Disponible en: https://lsi2.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php