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 XXXXX con optimización de caché, XXXXX con SIMD y XXXXX con OpenMP. Se concluye que optimización y paralelismo explotan eficazmente la arquitectura multinúcleo y SIMD.

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

1. 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] y 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.

1. Implementación y experimentación

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

* 1. 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 |
| Memoria | 118TB |
| SO | SUSE Linux Enterprise |
| Compilador | GCC 10.1.0 |
| Caché L1 | 49152 bytes |
| Caché L2 | 1310720 bytes |
| Línea Caché | 64 bytes |

AQUI NESTA PARTE FALO DE CALES SON OS DATOS MAIS RELEVANTES PARA NOS.

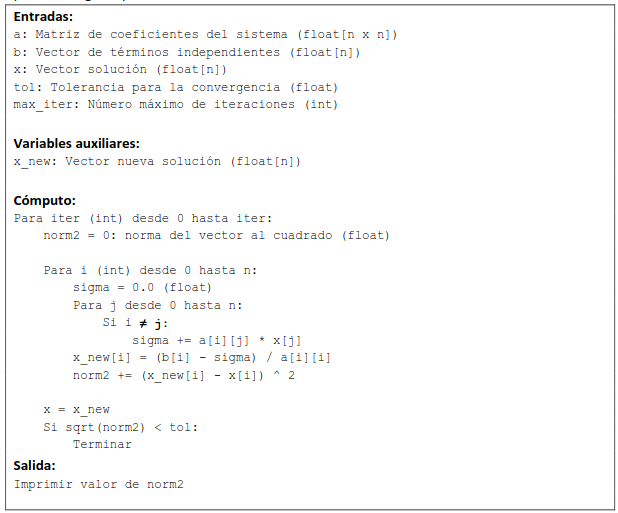
* 1. Implementación de las versiones

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



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 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).

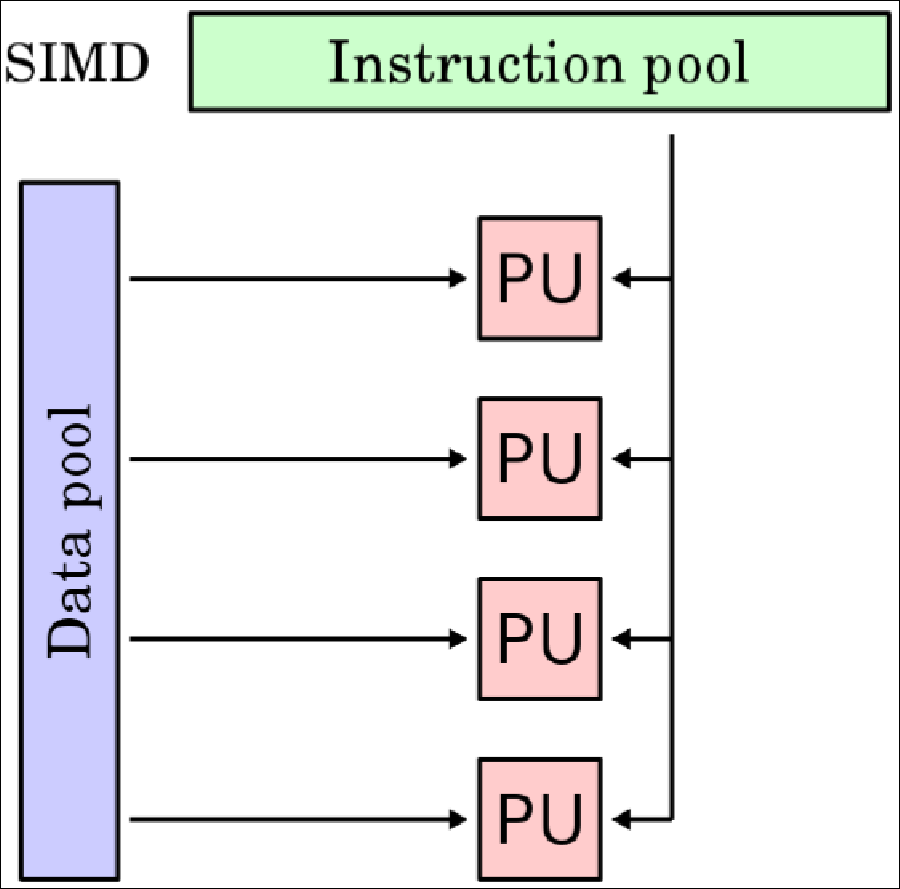
Durante cada paso del algoritmo:

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.

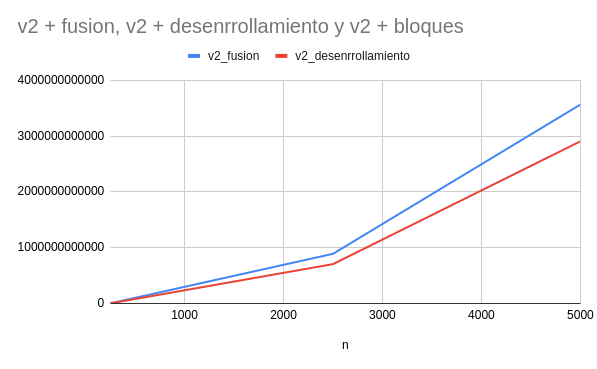
Figura II

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. Tras realizar pruebas individuales y combinadas, se descartó el uso de operaciones por bloques al observar que introducían penalizaciones de rendimiento. En cambio, la combinación de reducción de instrucciones, fusión y desenrollado de lazos mostró mejoras consistentes frente a la versión base, especialmente en tamaños de problema grandes. Por tanto, se adoptó esta combinación como la versión final optimizada secuencial.

Figura III



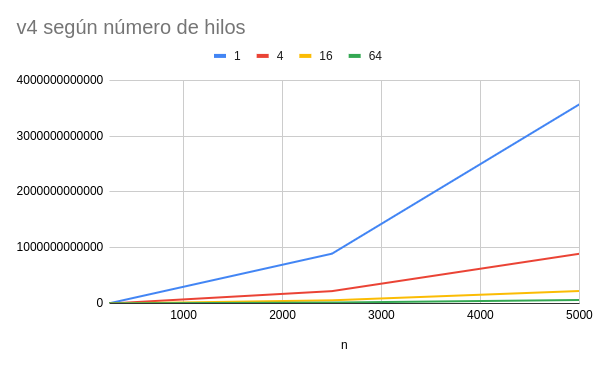
En la Figura III se presenta una comparativa entre varias configuraciones evaluadas. Se observa una mejora progresiva al aplicar cada optimización, alcanzando mejor eficiencia al combinar las tres seleccionadas, y perdiendo rendimiento al incluir operaciones por bloques.

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. En la Figura IV se muestra el número de ciclos de CPU registrados al variar el número de hilos de ejecución. Se observa una mejora de rendimiento al incrementar el grado de paralelismo: para n=5000, el tiempo de cómputo desciende de 3.57 × 10¹² a 6.11 × 10¹⁰ 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.

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

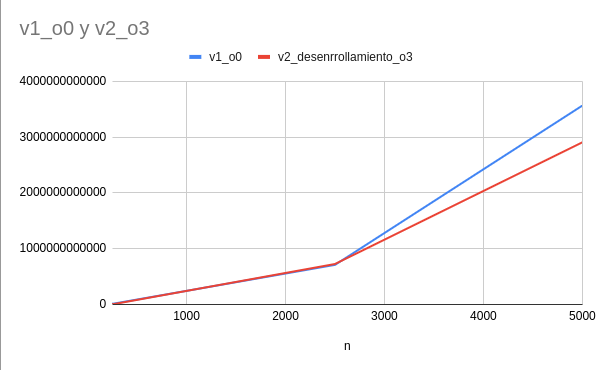


1. Resultados y análisis
2. Evaluación de las optimizaciones secuenciales

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



1. Comparativa entre vectorización y paralelismo OpenMP

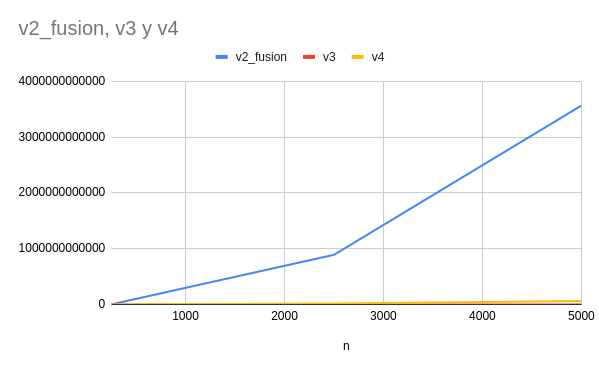
A continuación, en la Figura VI se presenta una comparativa del speedup obtenido por las versiones tercera (vectorizada con AVX256) y cuarta (paralela con OpenMP) respecto a la versión secuencial optimizada (v2). Cabe destacar que ambas variantes parten de la segunda versión, que ya incorpora optimizaciones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

(Estos dos párrafos los hizo todo chaty ya que no tengo datos de donde sacar conclusiones todavía)

Figura VI

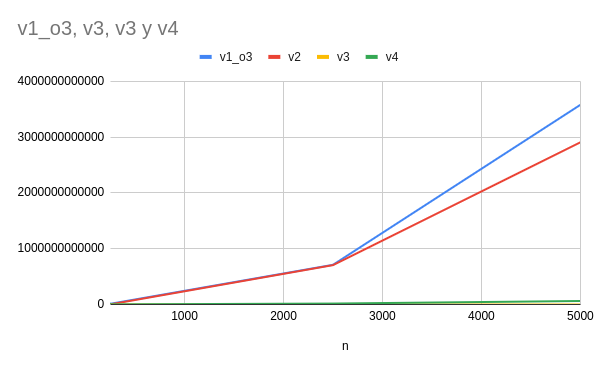


1. Comparación frente a la compilación optimizada -O3

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

Figura VII

En la Figura VII 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) consigue un avance notable: aproximadamente un 3× de velocidad extra sobre v2 para n=2500 y n=5000, gracias al procesamiento en vectores de 8 floats. El alineamiento correcto en memoria es crítico para alcanzar estos valores.
* 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 > 50× 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.

1. Conclusiones

En este trabajo hemos abordado la aceleración del método iterativo de Jacobi sobre arquitecturas modernas a través de cuatro implementaciones sucesivas en C. Partiendo de la versión v1, en la v2 incorporamos optimizaciones de caché, reducción de instrucciones, fusión de bucles y desenrollado, que proporcionaron una mejora consistente de entre un 15 % y un 20 % frente a v1 (‑O3), con especial ganancia en matrices de gran tamaño. A continuación, en la v3 explotamos paralelismo a nivel de datos con intrinsics AVX256, logrando un ... speedup adicional sobre v2 para n=2500 y n=5000. Finalmente, la versión v4 usa OpenMP para distribuir el trabajo entre 64 hilos, consiguiendo una gran aceleración respecto a v1 y marcando el mayor salto de rendimiento.

Las principales observaciones son:

* Localidad de memoria: la combinación de fusión y desenrollado reduce dramáticamente accesos redundantes y fallos de caché.
* Vectorización SIMD: el uso de registros AVX de 256 bits amortiza la carga–almacenamiento y reduce el número de iteraciones del bucle interno.
* Paralelismo multinúcleo: OpenMP escala casi de forma lineal hasta 64 hilos en matrices grandes, confirmando la elevada concurrencia efectiva de la plataforma FinisTerrae III.

Durante el desarrollo detectamos asimismo que el blocking intuitivamente útil no siempre compensa su sobrecarga de control, y que el parámetro óptimo de bloque puede variar según la jerarquía de caché específica del procesador.

En conjunto, los resultados demuestran que una estrategia escalonada de optimización, primero cache, luego SIMD y finalmente hilos, es una ruta eficaz para exprimir el rendimiento de arquitecturas multinúcleo con extensiones vectoriales.

Referencias

1. J. L. Hennessy y D. A. Patterson, Computer Architecture: A Quantitative Approach, 5.ª ed., Morgan Kaufmann, 2012.
2. Intel, “Intel Intrinsics Guide — AVX,” 2023. <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html> (Accedido: 15 de abril de 2025)
3. CESGA’s new FinisTerrae III. <https://www.cesga.es/en/cesga-actualiza-el-finisterrae/?utm_source=chatgpt.com> (Accedido: 4 de mayo de 2025)

<https://cesga-docs.gitlab.io/ft3-user-guide/batch_system.html>