Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Tesis de Licenciatura

Análisis e implementación secuencial y paralela de un algoritmo iterativo para resolución de grandes sistemas lineales

Director

Dr. Hugo Scolnik

Alumnos

Esteban Franqueiro
eaf@dc.uba.ar

Ximena Pisani
xpisani@dc.uba.ar

Abstract

En este trabajo se estudia e implementa un algoritmo iterativo para resolución de grandes sistemas lineales denominado *AcCim* ([1]), que es una alternativa acelerada a los métodos PPAM. Con este nuevo método se obtuvieron buenos resultados en la mayoría de los casos sin necesidad de precondicionamiento previo de los sistemas, incluso para aquellos considerados difíciles como los de Bramley y Sameh ([14]).
Para otros problemas cuyos resultados aún presentaban dificultades de convergencia, se estudió el uso de precondicionadores y de una nueva estrategia de descomposición en bloques que trabaja en conjunto con el algoritmo.

Organización

En la primera sección presentaremos una breve introducción teórica a los métodos iterativos de resolución de sistemas de ecuaciones lineales consistentes. Nos enfocaremos en los métodos de proyecciones y aquellas variantes relevantes al trabajo que realizamos.
En las siguientes tres secciones se presenta el trabajo desarrollado explicando, para cada versión, los detalles relevantes y las dificultades encontradas durante la implementación, el análisis de costo, las decisiones tomadas, los resultados obtenidos y los problemas numéricos que fueron marcando varios de los pasos a seguir. También se mostrarán comparaciones de eficiencia y precisión contra otro solver iterativo para sistemas lineales que es comúnmente usado hoy en día.
Finalmente, en los apéndices presentaremos el pseudo-código completo de los algoritmos, detalles de las ejecuciones de los resultados que presentamos, una breve descripción de los problemas con los que trabajamos y por último los formatos conocidos utilizados para almacenar las matrices ralas con el objetivo de disminuir el consumo de memoria.

Agradecimientos

A Hugo, por ser un excelente director de tesis, siempre dispuesto a llevar todo este trabajo adelante y a dedicarnos tiempo a pesar de no tenerlo.
A las chicas de La Plata, Nelly y Marité, por su constante ayuda, buena predisposición y paciencia. ¡Son geniales!
A la gente de SIDERCA, por darnos un entorno de aplicación para nuestro trabajo.

Ximena y Esteban

A Esteban, por ser también un gran compañero de tesis.
A mi mamá, que me apoyó y alentó siempre en esta carrera. No tengo palabras suficientes para agradecerle.
A mis hermanos Bruno, Federico y Julieta, porque es buenísimo tenerlos.
A mi abuela Malu, que siempre fue y es una importante compañía.
A mi papá y a mi abuela Tani, porque nunca dejaron de estar conmigo.
A la UBA y a los docentes de esta carrera, por la educación que brindan.
Y a todos los maestros, ayudantes y profesores que me enseñaron, motivaron e inspiraron durante mi formación. En especial a Graciela, Pedro, Joaquín y Hugo.

Ximena

A mis padres, porque gracias a ellos estoy donde estoy.
Al resto de mi familia por bancarme durante todos estos años.
A la UBA por la educación que me dio.
A todos los docentes del DC, porque también gracias a su esfuerzo llegué hasta acá.
A todos mis amigos de la facultad, que hicieron que esto sea más interesante.
A Juan Santos y Julio Jacobo, para que no me desaprueben el final que me queda :)
A Spawn, Gatoso y a la Rana, por quedarse conmigo mientras estudiaba o trabajaba a la noche.
A Ximena, no es necesario decir por qué...
Gracias... totales!!

Esteban

Indice

Introducción 7

Teoría básica 9

Métodos de proyecciones 9

Elección de la combinación lineal 9

Elección de la longitud del "paso" 10

Métodos de proyecciones por bloques 11

Forma general de los métodos de proyecciones por bloques 11

Elección de la combinación lineal 12

Métodos de proyecciones agregadas (PAM) 12

Métodos de proyecciones paralelas (PPAM) 13

Método para la elección de los pesos en PPAM 13

Sucesión óptima 14

Aceleración 15

Sucesión óptima 16

Convergencia global 16

Descomposición en bloques 17

Implementación de AcCim para matrices densas 19

Detalles de Implementación 19

Análisis de costo 19

Costo temporal 19

Costo espacial 20

Criterios de parada 20

Opciones de compilación 21

Paralelización 22

Librerías 24

Implementación de AcCim para matrices ralas 27

Detalles de implementación 27

Matrices ralas simétricas 28

Análisis de costo 28

Costo temporal 28

Costo espacial 29

Análisis de convergencia 29

Precondicionamiento 36

Balanceo 36

Precondicionamiento diagonal 37

Resultados obtenidos 37

Comparación con otros solvers 39

Descomposición en bloques ***40***

Algoritmo de descomposición 40

Detalles de implementación 40

Análisis de costo 41

Algoritmos de resolución 43

Detalles de implementación 43

Análisis de costo 43

Resultados 45

ALG1 vs. ALG2 45

Variaciones de μ 46

Sistemas mal condicionados 47

Conclusiones ***49***

Trabajo Futuro ***50***

Apéndices ***51***

Formatos de almacenamiento de matrices ralas 51

Indice de filas y columnas 51

Compressed Sparse Row (CSR) 51

Compressed Sparse Column (CSC) 52

Matrices simétricas 52

Formato de almacenamiento en disco 52

Descripción de los problemas 52

Sistemas H_n 53

Bramley & Sameh 53

SIDERCA 53

Algoritmos 55

Algoritmo AcCim 55

Algoritmo ALG1 56

Algoritmo ALG2 57

Algoritmo de descomposición en bloques 58

Reproducción de pruebas 59
Ejecutables disponibles en el CD 59
AcCim 59
AcCimMT 59
AcCimSP 59
Utilización de precondicionadores 60
Referencias 61

Introducción

Los métodos directos para resolver sistemas de ecuaciones lineales son muy utilizados debido a que por lo general obtienen muy buenos resultados en términos de precisión y velocidad. El problema es que a medida que el tamaño de los sistemas aumenta, también lo hace el espacio de memoria requerido para factorizar la matriz. Los métodos iterativos, por el contrario, al no factorizar el sistema y por lo tanto no utilizar tanta memoria, son la única alternativa para sistemas de grandes dimensiones. Estos también son una buena opción cuando se necesita conseguir una rápida aproximación a la solución de sistemas más pequeños, permitiendo obtenerla en menor cantidad de operaciones que las que puede requerir completar la resolución del sistema mediante métodos directos, en cuyos casos el resultado recién está disponible al finalizar el procedimiento.

Entre los métodos iterativos más usados para sistemas no simétricos se encuentran los basados en subespacios de Krylov como GMRES ([16]) y Bi-CGSTAB ([17]). Un subespacio de Krylov es aquel generado por los vectores b, Ab, A^2b, ..., $A^{n-1}b$, donde A es una matriz de dimensión $n \times n$. En la práctica Bi-CGSTAB es más útil porque sus requerimientos de memoria no crecen con el número de iteraciones, pero aunque estos algoritmos son muy rápidos y eficientes, no siempre convergen. Por ejemplo, cualquier combinación de algoritmo y precondicionador disponible en SPARSKIT2 ([18]) falla en al menos dos de los seis problemas de Bramley y Sameh ([3], en el apéndice de problemas hay una descripción de estos y otros casos de prueba utilizados en este trabajo). En el caso de GMRES, como la *i*-ésima iteración minimiza el residuo en el subespacio $K_i = \langle b, Ab, \ldots, A^{i-1}b \rangle$ y como $K_i \subset K_{i+1}$, el mismo decrece de manera monótona. De todas formas, a diferencia de los métodos que presentaremos, GMRES no tiene una demostración de convergencia global (pues acota el residuo, y no la distancia a la solución real).

En este trabajo describiremos tres algoritmos iterativos para resolución de grandes sistemas lineales. Estos algoritmos, inicialmente presentados en [1] y [2], se denominan *AcCim*, *ALG1* y *ALG2*.

Dado que para la mayoría de los sistemas el método *AcCim* muestra una convergencia acelerada, en general no requiere del uso de precondicionadores. Sin embargo, experiencias con el precondicionador de Canga-Becker y los sistemas de SIDERCA (ver apéndice de problemas) han demostrado que al aumentar la penalización de los mismos no varía el número de iteraciones requeridas para la convergencia.

En una primera etapa de este trabajo se implementa una versión para matrices densas. Sobre esta versión se definen detalles de implementación, se estudia el costo de

ejecución, se comprueba el comportamiento numérico sobre ciertos sistemas, se analiza la eficiencia lograda con distintos métodos de compilación y se consideran optimizaciones como la utilización de funciones BLAS de diferentes librerías. Finalizando esta etapa se implementa una versión que explota el paralelismo intrínseco de ciertas operaciones de estos métodos y permite comprobar las diferencias de performance.
Luego se implementa una versión para matrices ralas que permite resolver sistemas de dimensiones mayores, uno de los principales objetivos de este tipo de solvers. En especial en este punto se empieza a trabajar sobre sistemas extraídos de problemas reales y de mayor complejidad. Para éstos, se presenta un análisis detallado de convergencia dado que presentan dificultades geométricas y mal condicionamiento.
En vista de resultados obtenidos por *AcCim* con sistemas muy mal condicionados, y con el objetivo de extender su alcance, posteriormente se analiza el uso de precondicionadores sobre los sistemas y una técnica de partición en bloques que fue especialmente desarrollada para este algoritmo, pero que no había sido implementada para matrices ralas.

Todas las implementaciones presentadas fueron escritas en el lenguaje de programación `C/C++` y compiladas con el compilador de Intel. Las pruebas se ejecutaron en una computadora con dos procesadores Itanium de 1.5 GHz, 2 GB de RAM y el sistema operativo Linux con 20 GB de swap file.

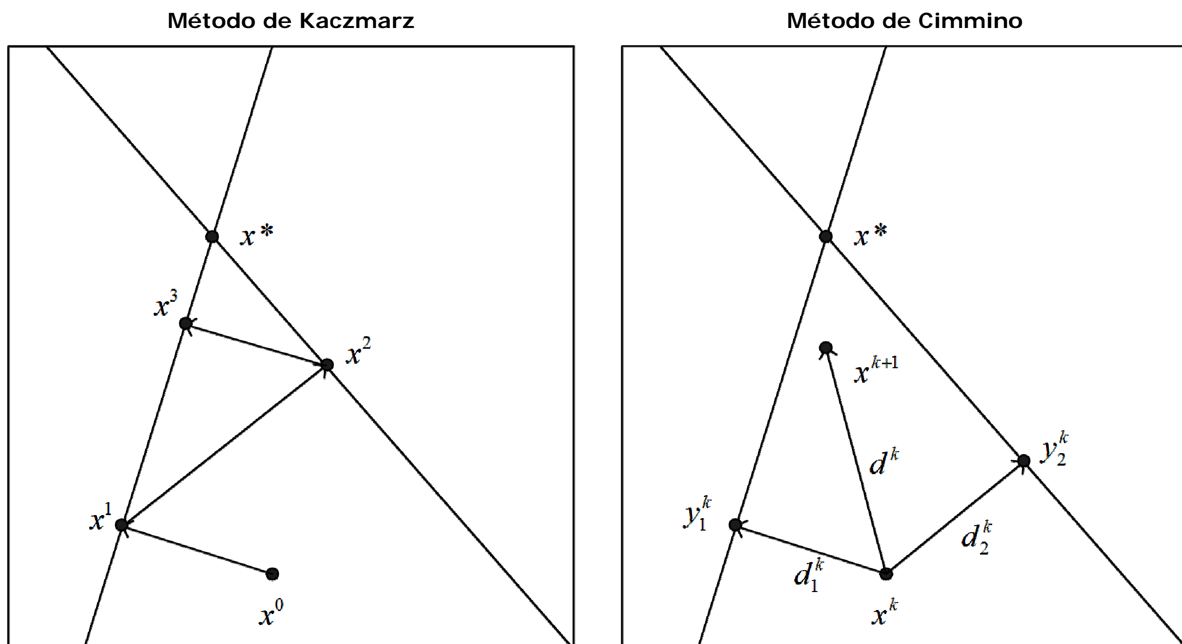
Teoría básica

Métodos de proyecciones

Los métodos de proyecciones para resolver sistemas de ecuaciones lineales consistentes generan una nueva iterada x^{k+1} proyectando el punto actual x^k sobre los hiperplanos que conforman el sistema, y luego combinan estas proyecciones para obtener una nueva dirección sobre la que avanzar.

Hay diferentes formas de hacer esto. El método de Kaczmarz ([9]) proyecta secuencialmente x^k sobre todos los hiperplanos hasta que converge. En el método de Cimmino ([8]), en cambio, desde el punto x^k se calculan las proyecciones a todos los hiperplanos y luego se las combina para generar una nueva dirección sobre la que estará el punto x^{k+1}. En sus formulaciones clásicas, la convergencia de estos métodos usualmente es muy lenta si los sistemas están mal condicionados.

Esta clase de métodos presenta dos problemas a resolver. Por un lado cómo deben combinarse las proyecciones para obtener una nueva dirección, y por otro cuánto avanzar sobre esa dirección.



x^* **representa la solución exacta,** y_i^k **es la proyección de** x^k **sobre el hiperplano** i **,** $d_i^k = y_i^k - x^k$ **y** d^k **es la combinación de los** d_i^k **.**

Elección de la combinación lineal

Un enfoque clásico es utilizar una combinación convexa de las direcciones, eligiendo los pesos según las distancias a los hiperplanos.

Elección de la longitud del "paso"

Un vez que se tiene una dirección sobre la cual avanzar, es necesario calcular la longitud del paso sobre la misma. Hay varias formas de hacer esto. A continuación presentaremos una formulación que se puede utilizar para sistemas con matriz simétrica definida positiva para minimizar a lo largo de cualquier dirección. En particular es utilizada por el método de gradientes conjugados ([10]).

Para resolver el sistema $Ax = b$, con $A \in \Re^{n \times n}$ simétrica y definida positiva, se buscan los ceros de la función $g(x) = Ax - b$. Para esto, se define la función $f(x) = 1/2\, x^t Ax - b^t x$, tal que sus mínimos coinciden con los ceros de $g(x)$ (debido a que $\nabla f(x) = Ax - b = g(x)$). Minimizando para λ la función $F(\lambda) = f(x^k + \lambda p)$ donde p es una dirección de descenso (por ejemplo la dirección obtenida mediante la combinación lineal de las proyecciones) y λ es la longitud del paso que se quiere calcular, se obtiene:

$$
\begin{aligned}
F(\lambda) &= \frac{1}{2}\left(x^k + \lambda p\right)^t A\left(x^k + \lambda p\right) - b^t\left(x^k + \lambda p\right) \\
&= \frac{1}{2}\left[\left(x^k + \lambda p\right)^t\left(Ax^k + \lambda Ap\right)\right] - b^t x^k - \lambda b^t p \\
&= \frac{1}{2}\left[x^{k^t} Ax^k + \lambda x^{k^t} Ap + \lambda p^t Ax^k + \lambda^2 p^t Ap\right] - b^t x^k - \lambda b^t p \\
&= \frac{1}{2}\left[x^{k^t} Ax^k + 2\lambda x^{k^t} Ap + \lambda^2 p^t Ap\right] - b^t x^k - \lambda b^t p
\end{aligned}
$$

Derivando ahora con respecto a λ, se tiene:

$$F'(\lambda) = x^{k^t} Ap + \lambda p^t Ap - b^t p$$

Finalmente, igualando a 0 y despejando:

$$
\begin{aligned}
&F'(\lambda) = 0 \Rightarrow \\
&\quad \lambda = \frac{b^t p - x^{k^t} Ap}{p^t Ap} = -\frac{x^{k^t} Ap - b^t p}{p^t Ap} = -\frac{\left(x^{k^t} A - b^t\right) p}{p^t Ap} = -\frac{\left(Ax^k - b\right)^t p}{p^t Ap} \\
&\quad\ = -\frac{r^{k^t} p}{p^t Ap}
\end{aligned}
$$

Donde $r^k = Ax^k - b$.

El problema de los métodos asociados a buscar los mínimos de una forma cuadrática en una dirección es que requieren que la matriz A sea simétrica y definida positiva, lo cual limita su aplicación. Sin embargo, siempre pueden utilizarse para el sistema $A^t Ax = A^t b$, aunque su número de condición es el cuadrado del original.

Métodos de proyecciones por bloques

Dado el sistema $Ax = b$, con $A \in \Re^{m \times n}$, los métodos de proyección por bloques consideran una partición del sistema A en q bloques de la siguiente forma:

$$A = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_q \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_q \end{bmatrix}, \ A_i \in \Re^{m_i \times n}, \ \sum_{i=1}^{q} m_i = m, \ 1 \le q \le m,$$

y la correspondiente partición de b.

Esto genera una partición del conjunto $\{1,2,\ldots,m\} = I_1 \cup I_2 \cup \ldots \cup I_q$, donde I_i contiene los índices de las filas de $Ax = b$ pertenecientes a $A_i x = b_i$. Se define $S_i = \{x \in \Re^n : A_i x = b_i\}$ (4) con $i = 1,\ldots,q$, el conjunto de soluciones del *i*-ésimo bloque.

La idea básica de estos métodos es que, dado un punto x^k, se calcula un nuevo punto y_i^k más cercano a S_i para cada $i = 1,\ldots,q$, y la nueva iterada se define, moviéndose a partir de x^k con una dirección que es una combinación de las direcciones $d_i^k = y_i^k - x^k$ generadas.

Estos métodos tienen dos características que los hacen atractivos:

- Las proyecciones sobre cada bloque pueden calcularse en forma paralela, aprovechando las capacidades de multiprocesamiento disponibles.
- Eligiendo los bloques de manera conveniente se puede conseguir un mejor condicionamiento en cada uno de ellos, y de esta forma mejorar la calidad numérica de las proyecciones. Los métodos tradicionales para la construcción de los bloques son menos costosos, pero las proyecciones que se obtienen pueden estar seriamente afectadas por el mal condicionamiento.

Forma general de los métodos de proyecciones por bloques

Dada una solución x^k, para cada $i = 1,\ldots,q$:

- En el caso de las proyecciones exactas, se calcula, para cada S_i, la proyección de x^k:

  $$y_i^k = P_i(x^k) = \arg\min_{y \in S_i} \left\{ \left\| y - x^k \right\| \right\}$$

- Como alternativa, para cada S_i se pueden calcular proyecciones aproximadas y_i^k bajo ciertas condiciones que aseguren convergencia, como por ejemplo:

  $$\left\| y_i^k - x* \right\| < \left\| x^k - x* \right\|$$

Para asegurar la convergencia a una solución $x*$ se define d^k como una combinación convexa de las proyecciones $d_i^k = y_i^k - x^k$, de forma tal que

$d^k = \sum_{i=1}^{q} w_i^k d_i^k$, con $w_i^k \geq 0$ y $\sum_{i=1}^{q} w_i^k = 1$.

Finalmente, la nueva iterada se define como $x^{k+1} = x^k + \lambda_k d^k$ donde λ_k es un parámetro de relajación tal que $0 < \lambda_k < 2$ en esta clase de métodos.

Como puede verse, dos problemas importantes que se presentan son el cálculo del "peso" w_i^k de la proyección sobre cada bloque en la dirección combinada resultante, y la longitud del "paso" λ_k sobre la nueva dirección.

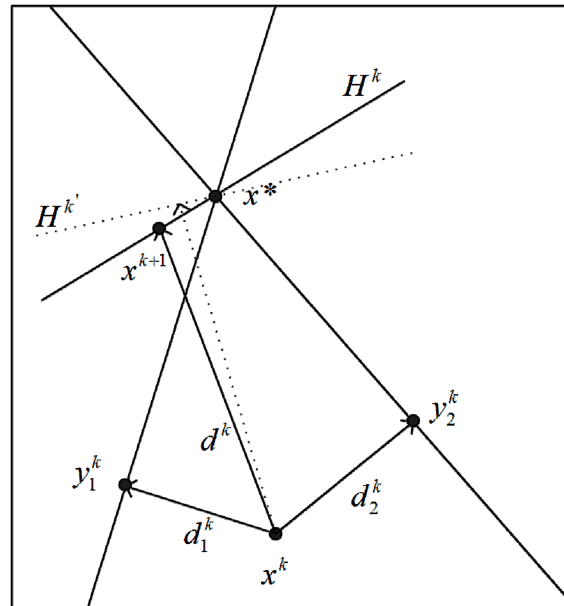
Elección de la combinación lineal

En el método clásico de Cimmino los pesos son constantes. En particular, elegir cualquier sucesión de pesos $\{w^k\}$ con $w_i^k > 0$ para todo i, k, lleva a un método Cimmino paralelo (ver [12]). Los mismos autores muestran que si $\varepsilon \leq \lambda_k \leq 2 - \varepsilon$, para todo $k \geq 0$ y para algún $\varepsilon > 0$ arbitrario, y que si $\sum_{k=0}^{\infty} w_i^k = +\infty$ para $i = 1,...,q$, entonces la sucesión $\{x^k\}$ converge al punto $x^* \in \bigcap_{i=1}^{q} S_i$.

Métodos de proyecciones agregadas (PAM)

Los métodos de proyecciones agregadas (*Projected Aggregation Methods*, PAM) para resolver sistemas de ecuaciones lineales generan una nueva iterada x^{k+1} moviéndose hasta un nuevo hiperplano H^k generado mediante una combinación lineal de los hiperplanos originales, que contiene a la dirección óptima $x^* - x^k$. El paso sobre la dirección generada se hace minimizando la distancia a la solución. En [1][2][4] y [5], se introdujeron esquemas de aceleración para resolver sistemas de ecuaciones e inecuaciones lineales respectivamente, dentro de un contexto PAM. En dichos trabajos, la idea básica es forzar a la siguiente iterada a pertenecer al nuevo hiperplano agregado o a la región convexa definida por el hiperplano separador, calculada en la iteración actual. El algoritmo *AcCim* presentado en [1] y expuesto más adelante, es la base para el cálculo de estas proyecciones.

Métodos de proyecciones agregadas



En la figura puede verse como una mejor elección del hiperplano agregado produce un mayor acercamiento a la solución. La elección de este hiperplano, que depende de los valores de los pesos w_i^k, es clave en el rendimiento del método ya que serán necesarias menos iteraciones para lograr la convergencia. En [2] se presenta un nuevo método para elegir en forma óptima los pesos en métodos PAM, que describiremos más adelante.

Métodos de proyecciones paralelas (PPAM)

Son variantes de los métodos PAM. En estos métodos las proyecciones se calculan en forma simultánea, aprovechando de esta forma las capacidades de multiprocesamiento de las computadoras (por ejemplo los nuevos procesadores multi-core). En el método de Cimmino puede paralelizarse por completo el cálculo de las proyecciones, debido a que no hay dependencias entre ellas. El método de Kaczmarz por el contrario, no es paralelizable debido a que es intrínsecamente secuencial. El método *AcCim* pertenece a la clase de los PPAM.

Método para la elección de los pesos en PPAM

La iterada x^{k+1} es la proyección de x^k sobre el hiperplano agregado

$H^k = \left\{x \in \Re^n : d^{k^t}\left(x - x^k\right) = d^{k^t}\left(x^* - x^k\right)\right\}$ (5),

donde $d^{k^t}\left(x^* - x^k\right) = \sum_{i=1}^{q} w_i^k \left\|d_i^k\right\|^2$ es una combinación de los hiperplanos

$H_i^k = \left\{x \in \Re^n : d_i^{k^t}\left(x - x^k\right) = d_i^{k^t}\left(x^* - x^k\right)\right\}$ para $i = 1,\ldots,q$.

En [2] se presenta un método para elegir los pesos $\{w_i^k\}_{i=1}^q$ de forma tal de obtener una combinación óptima de las direcciones $\{d_i^k\}_{i=1}^q$ que resulta de usar

$$w^k = \arg\min_{w\in\Re^q}\left\{\left\| x^k + \sum_{i=1}^q w_i d_i^k - x^*\right\|^2\right\} \quad (6).$$

De esto se obtiene el paso iterativo $x^{k+1} = x^k + \sum_{i=1}^q w_i^k d_i^k$ siendo $w^k \in \Re^q$ la solución al siguiente problema cuadrático

$$F(w) = \min_{w\in\Re^q}\left\{\left\| x^k - x^*\right\|^2\right\} + 2w^t D^{k^t}\left(x^k - x^*\right) + w^t D^{k^t} D^k w \quad (7)$$

donde $D^k = \left[d_1^k,\ldots,d_q^k\right]$.

Para obtener la solución, derivamos respecto a w:

$$F'(w) = 2D^{k^t}\left(x^k - x^*\right) + 2w^t D^{k^t} D^k = 0 \Rightarrow$$
$$w = \left(D^{k^t} D^k\right)^{-1} D^{k^t}\left(x^* - x^k\right)$$

Como en cada iteración se toman solamente direcciones linealmente independientes, resulta que $rank\left(D^k\right) = q_k \le q$. Por el Lema 2.1(ii) de [2] ($d_i^{k^t}\left(x^* - x^k\right) = \left\| d_i^k\right\|^2$) se obtiene la siguiente expresión explícita para w^k,

$w^k = \left(D^{k^t} D^k\right)^{-1}\left(\left\| d_1^k\right\|^2,\ldots,\left\| d_{q_k}^k\right\|^2\right)^t$. Combinando estos resultados con el método PPAM general, se obtiene el algoritmo *ALG1* (Algoritmo 2.3 de [2]), que reproducimos en el apéndice.

Sucesión óptima

La sucesión $\{x^k\}$ generada por el método *ALG1* satisface

$$\left\| x^{k+1} - x^*\right\|^2 = \left\| x^k - x^*\right\|^2 - \alpha_k{}^*$$

dónde

$$\alpha_k{}^* = \sum_{i=1}^q w_i \left\| d_i^k\right\|^2 = \left\| d^k\right\|^2 \quad (8).$$

Dado que la combinación lineal de las direcciones $\{d_i^k\}_{i=1}^q$ usadas en *ALG1* es óptima, se obtiene que $\alpha_k{}^* \ge \alpha_k$, siendo $\alpha_k = \dfrac{\left(\sum_{i=1}^1 w_i^k \left\| d_i^k\right\|^2\right)^2}{\left\| d^k\right\|^2}$ el valor definido para el método PPAM

por García-Palomares ([11]). De esta manera se demuestra que la sucesión $\{x^k\}$ generada por *ALG1* es óptima.

Aceleración

Como x^k se obtiene moviéndose hasta un hiperplano H^k, y considerando la definición de H^k ya mencionada (5), resulta que el incremento $x^* - x^k$ se encuentra dentro de este hiperplano.

Debido a esto, la idea es generar la siguiente iterada sobre dicho hiperplano elegido de forma óptima en base a los criterios ya descriptos en (6) y (7). Una opción es proyectar la dirección óptima d^k, calculada por *ALG1* sobre el hiperplano H^{k-1}. Esta dirección, denotada por $\tilde{d}^k$, es $\tilde{d}^k = \sum_{i=1}^{q_k} w_i^k P_{v^\perp}\left(d_i^k\right)$, con $v = x^k - x^{k-1}$ y $P_{M^\perp}$ el proyector ortogonal sobre el subespacio ortogonal a $range(M)$. De esta forma, la nueva iterada se define como $\tilde{x}^{k+1} = x^k + \tilde{\lambda}\tilde{d}^k$, donde $\tilde{\lambda} = \arg\min_{\lambda}\left\{\left\|x^k + \lambda\tilde{d}^k - x^*\right\|^2\right\}$ (9).

Los siguientes resultados (demostrados en [2]) se utilizan para explicar la convergencia del método acelerado que para $k \geq 1$ usa la dirección $\tilde{d}^k = P_{v^\perp}\left(d^k\right)$, donde $\tilde{d}^k$ es la combinación óptima obtenida por *ALG1*.

La diferencia $v = x^k - x^{k-1}$ satisface las condiciones:

(C1) $v^t\left(x^* - x^k\right) = 0$

(C2) $d_i^{k-1^t} v = \left\|d_i^{k-1}\right\|^2$, para $i = 1 \ldots q$

Lema

Si en x^k, $k \geq 1$, $x^k \neq x^*$, se considera la dirección óptima $d^k = D^k w^k$ definida por *ALG1*, y si v satisface las condiciones (C1) y (C2), entonces:

1. $\left\|P_{v^\perp}\left(d^k\right)\right\| > 0$
2. $\left\|P_{v^\perp}\left(d^k\right)\right\| < \left\|d^k\right\|$

Teorema

Si $\tilde{x}^{k+1} = x^k + \tilde{\lambda}\tilde{d}^k$, con $\tilde{\lambda}$ definida en (9), en la iteración $k \geq 1$, donde $\tilde{d}^k = P_{v^\perp}\left(d^k\right)$, $\tilde{d}^k$ definida según ALG1 y $v = x^k - x^{k-1}$ satisface (C1) y (C2), entonces $\left\|\tilde{x}^{k+1} - x^*\right\|^2 = \left\|x^k - x^*\right\|^2 - \tilde{\alpha}_k$, con $\tilde{\alpha}_k > \alpha_k{}^*$.

Para construir el nuevo método acelerado se considera este último resultado, donde la dirección usada pertenece al subespacio $\left[P_{v^\perp}\left(d_1^k\right)\ldots P_{v^\perp}\left(d_q^k\right)\right]$. Como la idea es minimizar la distancia entre $\tilde{x}^{k+1} = x^k + \tilde{d}^k$ y x^*, se elige $\tilde{d}^k$ como la mejor combinación de $\left\{P_{v^\perp}\left(d_i^k\right)\right\}_{i=1}^q$ partiendo de x^k.

Dado que x^k, $k \geq 1$, $x^k \neq x^*$, la siguiente iterada $x^{k+1} = x^k + \tilde{D}^k w^k$, donde $w^k \in \Re^q$ es la solución del problema cuadrático

$w^k = \arg\min_{w \in \Re^q}\left\{\left\|x^k + \tilde{D}^k w - x^*\right\|^2\right\}$ (11),

donde $\tilde{D}^k = \left[P_{v^\perp}\left(d_1^k\right)\ldots P_{v^\perp}\left(d_{q_k}^k\right)\right]$ y q_k es el número de direcciones linealmente independientes de $\left\{P_{v^\perp}\left(d_i^k\right)\right\}_{i=1}^q$. Combinando la nueva forma de calcular la dirección óptima con el método anterior, se obtiene finalmente el método acelerado *ALG2* que se reproduce en el apéndice.

Sucesión óptima

Cualquier sucesión $\left\{x^k\right\}$ generada por *ALG2* satisface

$\left\|x^{k+1} - x^*\right\|^2 = \left\|x^k - x^*\right\|^2 - \tilde{\alpha}_k$

con $\tilde{\alpha}_k > \alpha_k{}^*$, lo que demuestra la optimalidad de la misma.

Convergencia global

Lo anteriormente expuesto muestra que la sucesión $\left\{\left\|x^k - x^*\right\|^2\right\}$ es convergente. Resta, entonces, ver que la sucesión $\left\{x^k\right\}$ también lo es. Para estudiar la convergencia de esta sucesión, se utiliza la notación $P = \{1,\ldots,q\}$, $S = \bigcap_{i \in P} S_i$ (ver (4)), $d(x, C)$ la distancia euclídea entre $x \in \Re^n$ y el conjunto convexo cerrado $C \subseteq \Re^n$. También se define $\Phi(x) = \max_{i \in P}\{d(x, C_i)\}$.

Una sucesión $\left\{x^k\right\}$ se dice Féjer-monótona con respecto al conjunto C si para algún $x^* \in C$, $\left\|x^{k+1} - x^*\right\| \leq \left\|x^k - x^*\right\|$ para todo $k \geq 0$.

Teorema

Sean $C_i \in \Re^n$ conjuntos convexos cerrados para todo $i \in P$, y $C = \bigcap_{i \in P} C_i$ tal que $C \neq \phi$.

Si la sucesión $\{x^k\}$ satisface:

1. $\{x^k\}$ es Féjer-monótona con respecto a C
2. $\lim_{k \to \infty} \Phi(x^k) = 0$

Entonces $\{x^k\}$ converge a $x^* \in C$.

Lema

Cualquier sucesión $\{x^k\}$ generada por *ALG1* o *ALG2* satisface ambos casos del teorema anterior, considerando $C_i = S_i$, si $x^k \notin S$ para todo $k \geq 0$.

Dado que se asume que el sistema $Ax = b$ tiene solución, y considerando el lema y el teorema anterior, finalmente se llega a que cualquier sucesión $\{x^k\}$ generada por *ALG1* o *ALG2*, converge a una solución x^* de $Ax = b$.

Descomposición en bloques

Ahora presentaremos una descripción del método de descomposición en bloques originalmente introducido en [2]. Este método permite formar bloques con a lo sumo μ filas y tal que el número de condición estimado $\tilde{\kappa}\left(\left(A_i A_i^t\right)^{-1}\right) \leq \kappa$. Tanto μ como κ son parámetros del método.

Se asume que el sistema original ha sido normalizado por filas, y se denota como e_j al *j*-ésimo vector canónico y como d_{jj} al *j*-ésimo elemento de la matriz diagonal D.

Supongamos que se está generando un bloque de $A_i x = b_i$ que ya tiene *j* filas del sistema original, con $j < \mu$, y que se conoce la descomposición de Cholesky de $A_i A_i^t = L_j D_j L_j^t$.

El estimador β_i del número de condición $\kappa(A_i A_i^t)$ se define como $\beta_i = \frac{\tau}{\delta}$, donde $\tau = \max\{d_{hh}\}_{h=1}^{j}$ y $\delta = \min\{d_{hh}\}_{h=1}^{j}$. En [2] se demuestra que $\beta_i = \frac{1}{\delta}$ dado que $d_{11} = 1$ y que $d_{hh} \leq 1$ para todo $2 \leq h \leq j$.

Además se define el bloque A_i aumentado con a_l, la *l*-ésima fila de A, como $\tilde{A}_i = \begin{bmatrix} A_i \\ a_l \end{bmatrix}$.

Se agregará una nueva fila al *i*-ésimo bloque si y solo si dicha fila no pertenece a ningún otro bloque, y el estimador del número de condición del bloque aumentado no supera κ. De esta forma, para aceptar una nueva fila en el *i*-ésimo bloque, y usando que se conoce la descomposición de Cholesky de $A_i A_i^t$, se actualiza recursivamente la factorización de $\tilde{A}_i \tilde{A}_i^t$ y se re-calcula el número de condición del bloque aumentado $\tilde{A}_i$. Si se cumple que $\tilde{\kappa}\left(\tilde{A}_i\right) \leq \kappa$, entonces se agrega a_l a A_i.

El algoritmo completo, que como resultado secundario calcula los factores de las matrices $\left(D^{k^t} D^k\right)^{-1}$ usadas para el cálculo de las direcciones optimizadas de los métodos PAM, se describe en el apéndice.

Implementación de AcCim para matrices densas

Detalles de Implementación

El trabajo se inició con la implementación secuencial del algoritmo *AcCim* para matrices densas.

El pseudo-código del algoritmo implementado en esta etapa es el que está documentado en el apéndice de algoritmos como *AcCim*. En esta versión las matrices fueron implementadas mediante un array de punteros a vector. El proceso de normalización se lleva a cabo físicamente sobre los elementos de la matriz durante la fase de inicialización y luego se opera sin transformarla nuevamente.

Esta primera implementación permitió terminar de definir detalles del algoritmo como el criterio de parada y detalles de cálculos específicos, medir la eficiencia lograda mediante diferentes opciones de compilación, analizar en forma temprana su comportamiento numérico con distintos sistemas de prueba y tener una base para investigar optimizaciones y alternativas de paralelización.

A continuación presentamos el análisis de costo de esta versión y los distintos temas analizados a partir de esta.

Análisis de costo

Costo temporal

Siguiendo el pseudo-código de *AcCim* presentado en el apéndice de algoritmos, se puede calcular que su costo tiene un orden de:

$$[2M + n] + [3(M + n) + 5n + c1] + [ITER(2(M + n) + 9n + c2)].$$

Donde M es el costo de recorrer la matriz, n su dimensión, $ITER$ es el número de iteraciones que son ejecutadas antes de lograr la convergencia y $c1$ y $c2$ constantes.

El primer término corresponde al costo requerido para la normalización del sistema, el segundo al de la fase de inicialización y el tercero al del proceso iterativo.

Para simplificar esta expresión, podemos hacer el siguiente cálculo:

$$[2M + n] + [3(M + n) + 5n + c1] + [ITER(2(M + n) + 9n + c2)] =$$
$$5M + 9n + c1 + ITER \times (2M + 11n + c2)$$

Este costo tiene un orden de $O(M + ITER \times (M + n))$. Y dado que el costo de recorrer una matriz densa es de n^2, el costo total de ejecución de la versión de *AcCim* para matrices densas será de $O(n^2 + ITER \times (n^2 + n)) = O(n^2(ITER + 1) + nITER)$.

Si bien la cantidad de iteraciones necesarias depende del condicionamiento y las características geométricas del sistema, la cantidad de iteraciones teóricas necesarias para lograr una buena aproximación no debería superar la dimensión de la matriz, y en muchos casos es considerablemente menor.

Costo espacial

En cuanto al costo espacial, este está determinado principalmente por la necesidad de almacenar una estructura de n^2 elementos de punto flotante. Este es un costo bastante elevado que restringió la posibilidad de correr pruebas sobre sistemas cuyas matrices tuvieran una dimensión mayor a aproximadamente 15500×15500, trabajando con números de punto flotante de doble precisión en una computadora con 2 GB de RAM.

Criterios de parada

Como criterio de parada al proceso iterativo, se decidió usar $\sqrt{r_k} < \varepsilon \times \max(1, \sqrt{r_0})$.

Es decir, se finaliza la ejecución cuando el residuo es menor que cierto valor o bien ha experimentado una mejora considerable respecto al residuo inicial.

Por otro lado, dado que en distintos casos el valor definido como épsilon para el primer criterio de parada puede resultar demasiado exigente para un sistema y requerimiento de tiempo específico, se decidió dar la opción de forzar la terminación a un máximo permitido de iteraciones que también es definido como parámetro del programa.

Finalmente, se decidió agregar otro criterio que contempla el caso en el que por un número determinado de iteraciones el residuo no disminuya de forma significativa. Este último se introdujo en respuesta al análisis de convergencia que se presentará más adelante.

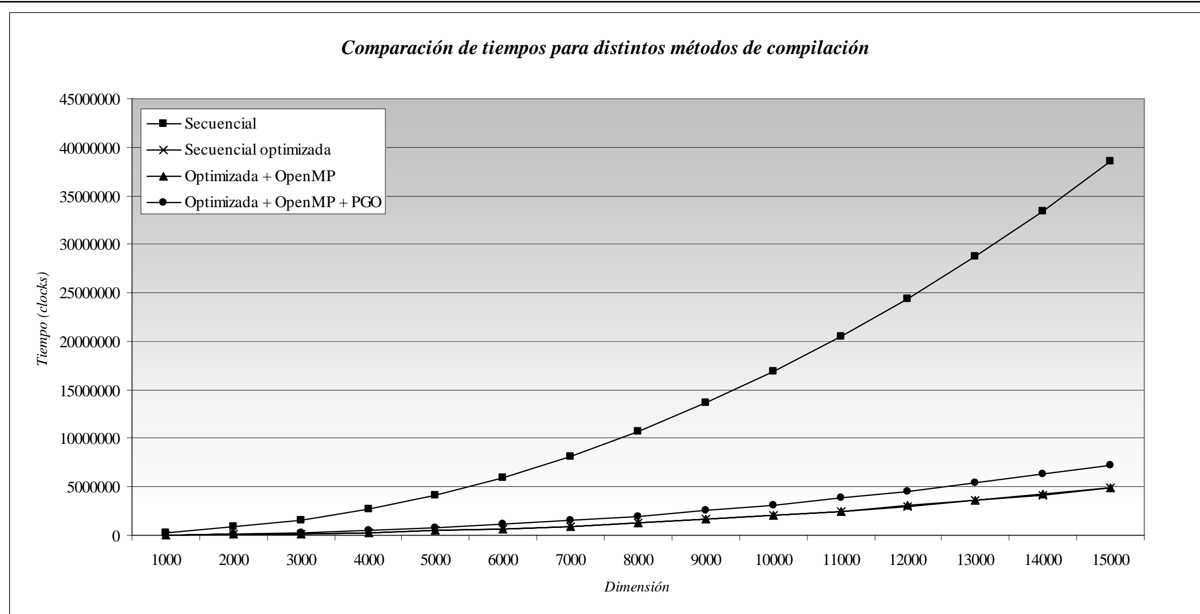
Los valores de ε de cada criterio de finalización, el máximo número de iteraciones permitidas, y la tolerancia en la cantidad de iteraciones que producen un estancamiento en la evolución del residuo son valores que pueden ser definidos paramétricamente al momento de la ejecución del solver. Estos valores, si bien pueden modificar el resultado final obtenido debido al número de iteración en el que producen el resultado, no modifican en ningún caso la convergencia de los sistemas.

Opciones de compilación

Se evaluaron distintas técnicas de compilación a fin de seleccionar aquella que resulte más eficiente para este programa en particular. Las opciones consideradas fueron las siguientes:

- Compilación sin optimizaciones. Produce un programa totalmente secuencial y sin optimizaciones por parte del compilador. Fue considerada solamente a fin de establecer una base contra la cual puedan compararse las otras opciones.
- Compilación optimizada. Produce un programa secuencial con optimizaciones por parte del compilador, incluyendo software pipelining, que es una técnica de optimización de ciclos disponible en la arquitectura Itanium.
- Compilación optimizada utilizando directivas OpenMP. El código del programa cuenta con directivas para el compilador de forma que este agregue a sus optimizaciones normales el uso de threads en aquellas secciones donde el programador indica la existencia de paralelismo. Para este caso también se evaluó la eficiencia obtenida variando el número de threads que se crean a fin de ejecutar las secciones paralelas, pero esto no produjo diferencias significativas.
- Compilación optimizada utilizando directivas OpenMP y la opción "profile guided optimization" (PGO). Al caso anterior se le suma una segunda etapa de compilación. En la primera etapa el programa es compilado y luego ejecutado una cantidad de veces a fin de generar archivos que definen un perfil de ejecución que es utilizado en la segunda etapa de compilación.

Cada uno de estos programas fue ejecutado sobre sistemas H_n (descriptos en el apéndice de problemas) de dimensión creciente a fin de obtener una estadística sobre la eficiencia de cada técnica. El siguiente gráfico ilustra estos resultados.



Naturalmente la mejora obtenida por la ejecución de la versión secuencial con optimizaciones con respecto a la versión secuencial sin optimizaciones fue significativa. Sin embargo la misma no fue superada por la versión optimizada con OpenMP. Esto lo atribuimos a que OpenMP agrega un cierto overhead en la ejecución pero fue efectivamente explotado en pocos fragmentos de código, por lo que esta pérdida de eficiencia no llegó a compensarse. En la siguiente sección mostraremos que al utilizar programación paralela explícita sí se obtuvieron mejoras significativas en los tiempos de ejecución.
Por último, no solo no se produjo ninguna mejora al agregarle PGO, sino que los tiempos fueron mayores.

En vista de estos resultados, en adelante decidimos utilizar compilación secuencial con optimizaciones del compilador, que es una técnica simple que no requiere ningún esfuerzo de programación y resultó estar entre las dos opciones más eficientes.

Paralelización

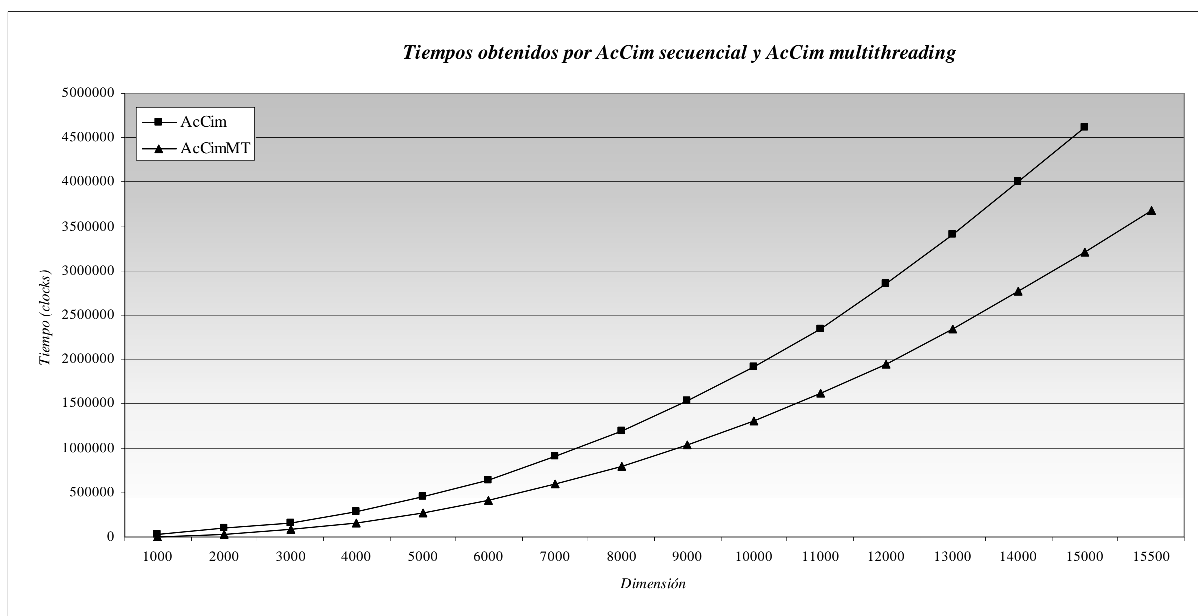
En un programa la paralelización puede darse a distintos niveles:

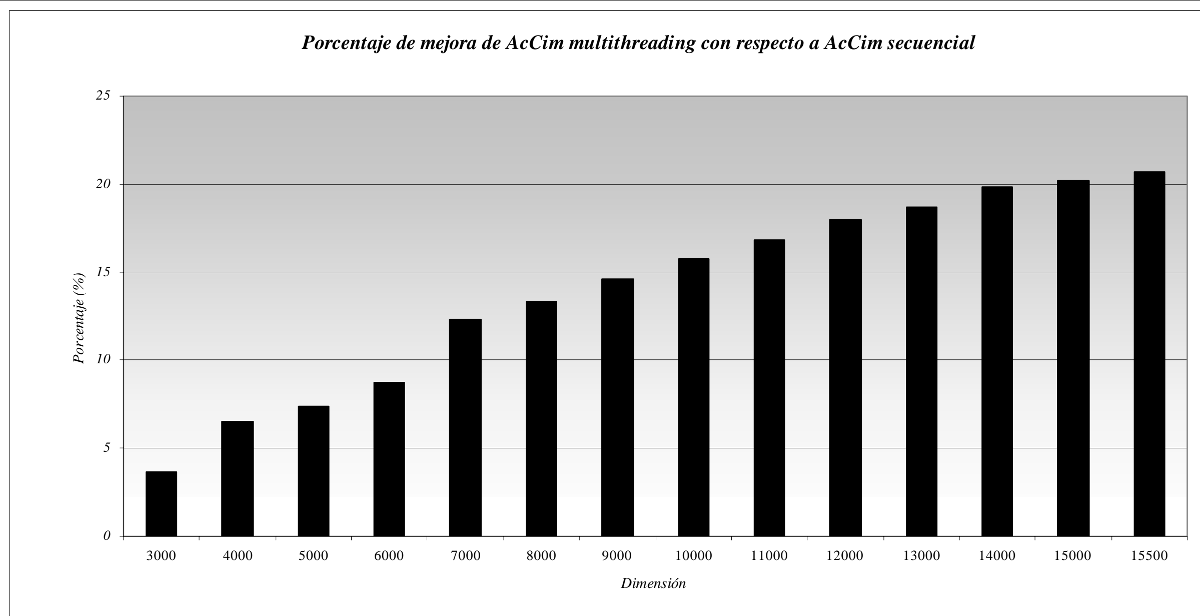
- Software pipelining. Esta paralelización está a cargo del compilador y si bien la ejecución de esta versión sigue siendo prácticamente secuencial, las instrucciones son reordenadas para obtener paralelismo en los ciclos. Esto fue conseguido usando las opciones de compilación optimizada.
- Utilización implícita de threads. Esta técnica consiste en incluir directivas para el compilador en determinadas líneas del programa en donde la existencia de

paralelismo podría explotarse. La misma ya fue evaluada en la sección anterior mediante la utilización de OpenMP.

- Utilización explícita de threads. Finalmente el paralelismo se puede alcanzar con la creación y ejecución explícita de threads en el programa. Para lograr esto se utilizó la librería pthreads ([13]) en una nueva versión que fue llamada AcCimMT. En esta versión una cantidad de threads son creados al efecto de calcular las proyecciones. De esta forma, éstas se calculan en forma totalmente paralela sobre distintas secciones de la matriz.
  Dado que trabajamos con una máquina con dos procesadores single-core y que el programa es CPU-bound, solo dos threads son creados para ser ejecutados simultáneamente. Utilizar un número mayor no produjo mejoras en los resultados.

El programa obtenido también fue ejecutado con sistemas estables de dimensión creciente a fin de poder establecer una comparación con las versiones anteriores. En el siguiente gráfico se puede observar esta comparación.



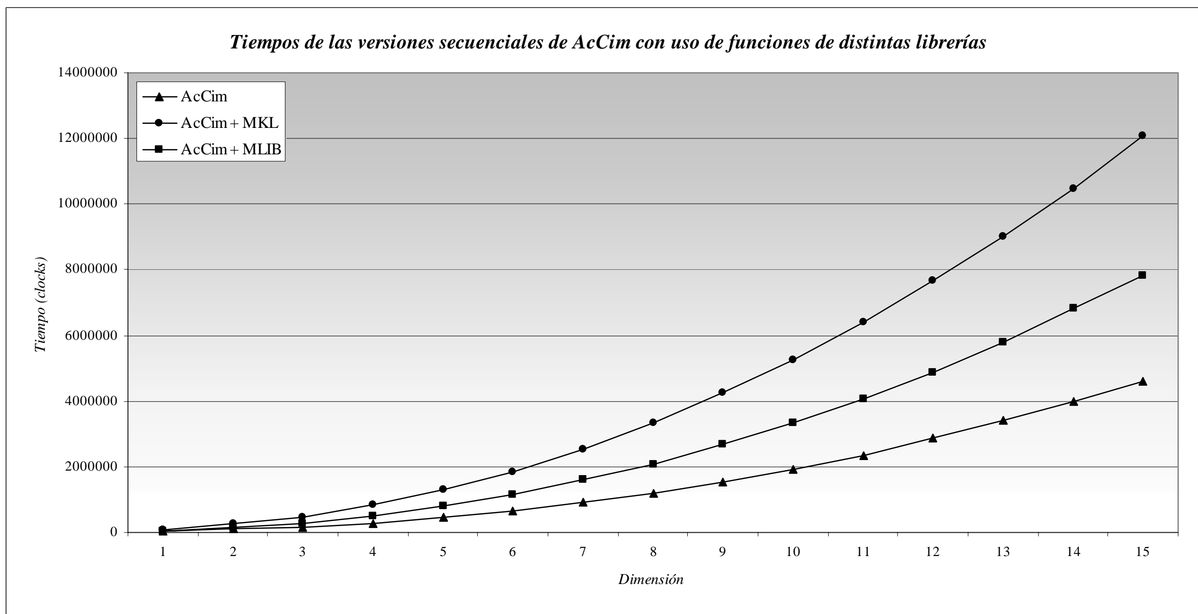


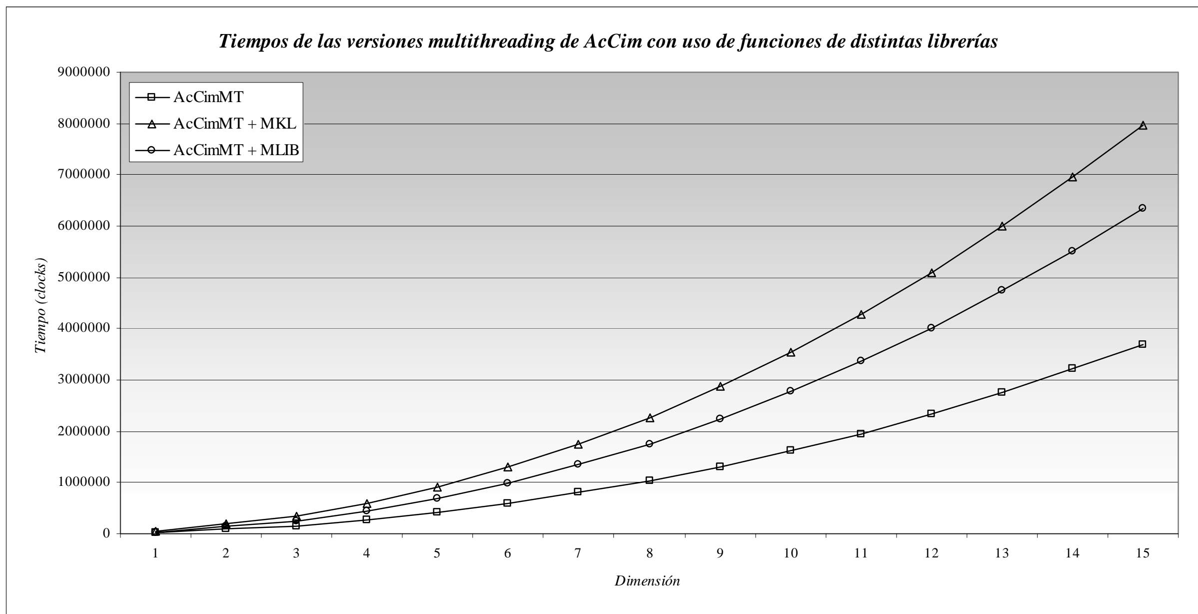
De ambos gráficos se puede concluir que la ganancia obtenida con la versión paralela fue significativa y que la misma se incrementa a medida que aumenta la dimensión de los sistemas.

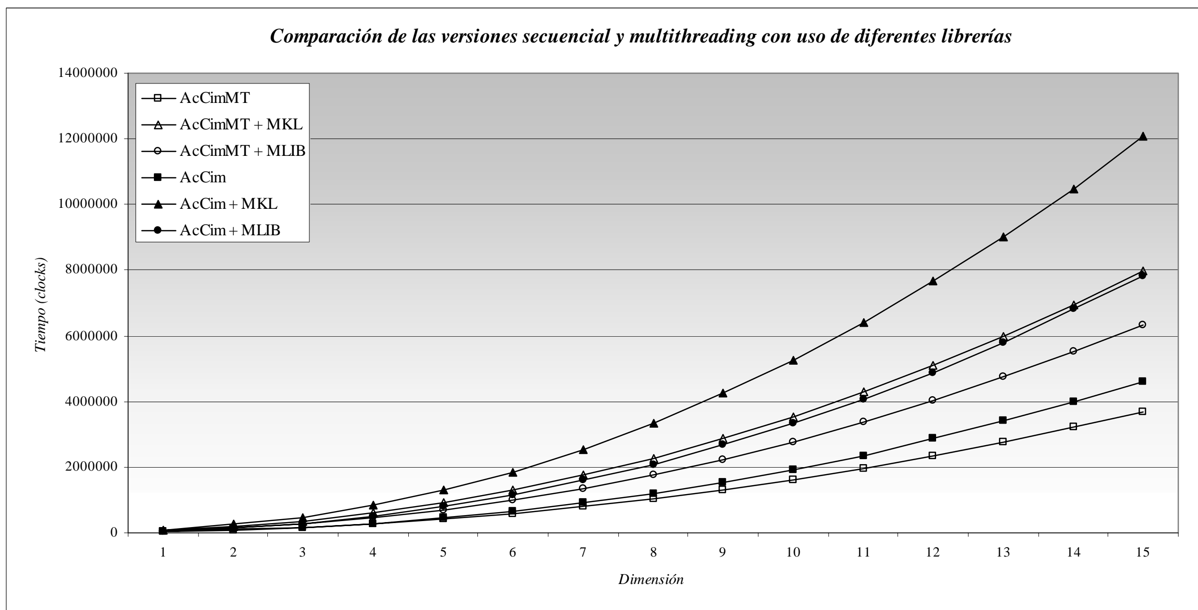
Librerías

En un principio usamos implementaciones propias para las funciones de cálculo de normas y producto matriz-vector que define el algoritmo.

A continuación mostramos los resultados obtenidos por versiones que implementamos a partir de la versión secuencial y paralela ya presentada, pero haciendo uso de funciones BLAS pertenecientes a las librerías comerciales MKL de Intel ([22]) y MLIB de HP ([23]).


Tiempos de las versiones secuenciales de AcCim con uso de funciones de distintas librerías
AcCim
AcCim + MKL
AcCim + MLIB
Tiempo (clocks)
Dimensión
0
2000000
4000000
6000000
8000000
10000000
12000000
14000000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


Tiempos de las versiones multithreading de AcCim con uso de funciones de distintas librerías
AcCimMT
AcCimMT + MKL
AcCimMT + MLIB
Tiempo (clocks)
Dimensión
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000
9000000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


Comparación de las versiones secuencial y multithreading con uso de diferentes librerías
AcCimMT
AcCimMT + MKL
AcCimMT + MLIB
AcCim
AcCim + MKL
AcCim + MLIB
Tiempo (clocks)
Dimensión
0
2000000
4000000
6000000
8000000
10000000
12000000
14000000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Como se puede ver en los gráficos, no encontramos mejoras de tiempo ni de precisión haciendo uso de estas librerías. Debido a esto continuamos trabajando con implementaciones propias de estas funciones.

Implementación de AcCim para matrices ralas

El objetivo de *AcCim* y del trabajo presentado es disponer de un algoritmo que permita llegar a resolver sistemas de dimensiones mucho mayores, por lo que resulta imprescindible contar con una implementación de *AcCim* para matrices ralas. Por este motivo, en esta etapa se modificó completamente la estructura interna con la que trabaja el algoritmo y por ende toda la implementación.

Detalles de implementación

En el apéndice de formatos de almacenamiento se encuentran explicadas estructuras utilizadas para este tipo de sistemas. En resumen ellas son:

- Compressed Sparse Column (CSC)
- Compressed Sparse Rows (CSR)
- Simétrica, que es equivalente en ambos formatos ya que almacena la sección triangular superior en CSR simétrica y triangular inferior en CSC simétrica

El formato sobre el que se decidió trabajar fue CSC debido a que en un principio no había razones claras para inclinarse especialmente por alguno de los dos y a que es el formato usado en el almacenamiento de los sistemas con los que se estaba trabajando.

Debido al cambio en la estructura de representación de los datos, fue necesario re-implementar las operaciones que trabajan sobre estas estructuras para que sean resueltas en forma eficiente. En particular, se modificó el patrón de accesos a las mismas. Como un ejemplo de esto, en la versión densa, al multiplicar una matriz por un vector se accedía a cada fila y se la multiplicaba por el vector. En esta versión, el costo de acceder a una fila es muy alto, ya que requiere recorrer toda la matriz.
Es por eso que a medida que se recorre esta matriz se opera simultáneamente sobre todas las filas. Como un ejemplo de la dinámica con la que se opera en este tipo de operaciones, a continuación se muestra un pseudo-código del producto matriz-vector implementado.

```
res = zeros(n);
for (j = 0; j < n; ++j) {
      for (k = colstr[j]; k < colstr[j + 1]; ++k) {
            res[rowind[k]] = res[rowind[k]] + values[k] * v[j];
      }
}
```

Una implementación como esta, que resuelve las distintas operaciones sobre la estructura de la matriz en forma global, dificulta la posibilidad de implementar fácilmente una versión paralela para matrices ralas ya que deja de ser trivial obtener una división de los datos sobre los que debería trabajar cada thread. Sin embargo permitió obtener una ganancia mayor que la obtenida al ejecutar una versión paralela sin eliminar la opción de

poder hacerlo mediante el uso de nuevas estructuras u otras soluciones de implementación.

Matrices ralas simétricas

Esta versión también soporta la resolución de sistemas con matrices ralas simétricas.
Un inconveniente que se presentó con este tipo de matrices fue que, al momento de la normalización del sistema, cada valor no nulo de la matriz es dividido por la norma de la fila a la que pertenece, pero cada valor almacenado físicamente en las mismas, a excepción de los valores correspondientes a la diagonal, corresponde a dos posiciones de la matriz pertenecientes a distintas filas.
Para resolver este problema se implementaron dos soluciones:

- Creación de un vector que contiene la norma de cada fila de la matriz. De esta forma cada vez que un elemento $M(i,j)$ es accedido luego de la normalización del sistema, se devuelve el valor correspondiente a (i,j) que se encuentra almacenado en la matriz original, dividido por la norma correspondiente almacenada en este vector.
- Extensión de la matriz rala simétrica a rala no-simétrica. La matriz es extendida y luego normalizada como en el caso no-simétrico.

Esta última opción resultó ser menos eficiente que la primera, por lo que por defecto el solver utiliza la opción de la creación del vector de normas.

Análisis de costo

Costo temporal

El costo de ejecución de *AcCim* para matrices densas, que estaba en el orden de $O(n^2 + ITER \times (n^2 + n))$, puede ser reducido al trabajar con matrices ralas si se utilizan estructuras de datos adecuadas. Esto se debe a la cantidad de elementos nulos sobre los que no es necesario operar.

Como ya mencionamos, el costo de ejecución del algoritmo *AcCim* está en el orden de $M + ITER \times (M + n)$, donde M es el costo de recorrer la matriz, n su dimensión e $ITER$ el número de iteraciones que son ejecutadas antes de lograr la convergencia.

Teniendo en cuenta las estructuras utilizadas para el almacenamiento de matrices ralas presentado en el apéndice de formatos de almacenamiento, podemos concluir fácilmente que el costo de recorrerla es de $2nz + n$ operaciones.

Finalmente, reemplazando M en la fórmula anterior por el costo requerido para recorrer una matriz rala, obtendremos que el costo de ejecución de la versión de *AcCim* para matrices ralas será de $O((2nz+n)+ITER\times((2nz+n)+n))$.

Esto último es válido porque la implementación presentada conserva la propiedad de que las operaciones que trabajan con la matriz rala lo hacen recorriendo la matriz la misma cantidad de veces que era necesario hacerlo en la implementación de *AcCim* para matrices densas.

Costo espacial

En cuanto al costo espacial, este cambia al cambiar la forma en que la matriz es almacenada. Mientras que n^2 valores eran requeridos para almacenar una matriz densa, solo $2nz+n$ valores son requeridos para almacenar una matriz rala siguiendo las estructuras definidas en el apéndice de formatos de almacenamiento para matrices ralas. De estos, solo nz son elementos de punto flotante, y el resto son valores enteros sin signo.

En el caso de que la matriz rala sea simétrica, el espacio puede ser reducido aún más. Esto es porque aquellos valores fuera de la diagonal, y sus posiciones, son almacenados una sola vez. Por este motivo, en este caso solo se requieren aproximadamente $nz+n$ valores para almacenar la matriz y $nz+2n$ valores al normalizarla, teniendo en cuenta que luego de esta operación se requiere conservar un vector de normas de sus filas.

Análisis de convergencia

A continuación se muestran los resultados obtenidos por la versión de *AcCim* para matrices ralas en los conjuntos de sistemas de SIDERCA y B&S (para una descripción de los mismos, ver el apéndice de problemas).

Resultados finales obtenidos luego de n iteraciones

Nombre	$\|A_N x^k - b\|$	$\|Ax^k - b\|$	$\|x^k - x^*\|$	$\frac{\|x^k - x^*\|}{\|x^*\|}$	$\|x^k - x^*\|_i$	Tiempo (segs)
base17p1	$3,77658\times10^{-18}$	$2,68059\times10^{-17}$	$6,82414\times10^{-15}$	$1,80420\times10^{-15}$	$6,22081\times10^{-16}$	375,1430
base17p3	$3,27036\times10^{-3}$	1,93914	1,85278	$4,85264\times10^{-1}$	$6,04775\times10^{-2}$	374,9230
base17p5	$5,10705\times10^{-5}$	2,97330	2,10597	$5,51510\times10^{-1}$	$7,12612\times10^{-2}$	374,6840
base17p7	$5,96279\times10^{-7}$	3,53290	2,18175	$5,71355\times10^{-1}$	$7,69522\times10^{-2}$	375,0060
B&S *P1*	$6,31149\times10^{-20}$	$1,83297^{-18}$	$1,29241\times10^{-16}$	$1,69893\times10^{-16}$	$1,00752\times10^{-17}$	43,1050
B&S *P2*	$2,00259\times10^{-17}$	$1,21243\times10^{-15}$	$4,51968\times10^{-14}$	$2,44098\times10^{-16}$	$2,26989\times10^{-15}$	43,1177
B&S *P3*	$6,56436\times10^{-6}$	$5,24566\times10^{-5}$	$3,57123\times10^{-3}$	$7,07824\times10^{-5}$	$4,07074\times10^{-4}$	43,5364
B&S *P4*	$1,70142\times10^{-17}$	$3,40304\times10^{-14}$	$3,70538\times10^{-14}$	$7,34413\times10^{-16}$	$1,65391\times10^{-15}$	43,0133
B&S *P5*	$3,58432\times10^{-18}$	$1,40871\times10^{-16}$	$7,86878\times10^{-15}$	$1,55960\times10^{-16}$	$4,34738\times10^{-16}$	43,4349
B&S *P6*	$4,75432\times10^{-18}$	$1,18176\times10^{-16}$	$6,83203\times10^{-15}$	$1,35412\times10^{-16}$	$3,67111^{-16}$	43,0933

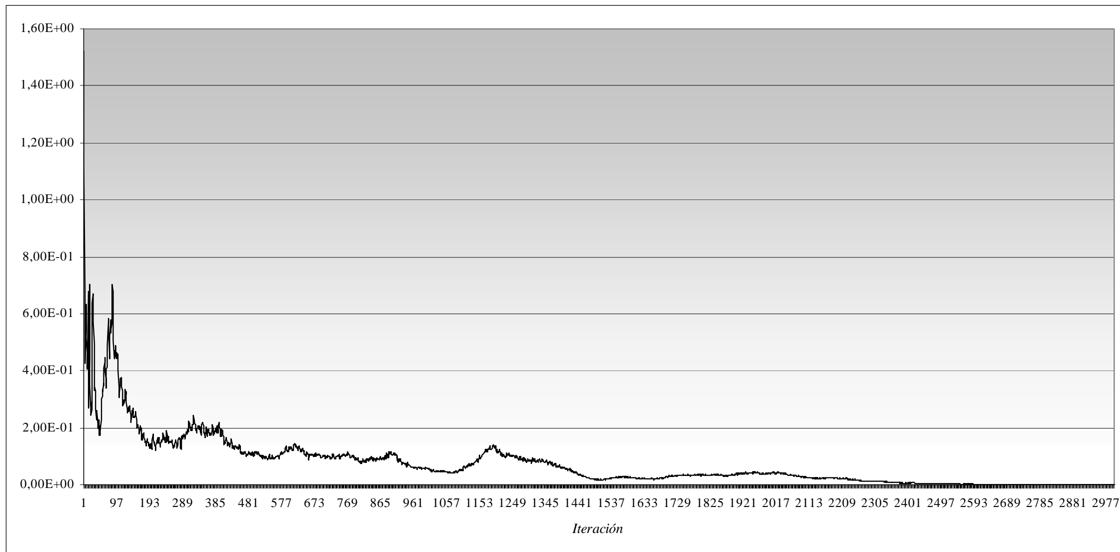
Resultados finales obtenidos luego de 3000 iteraciones

Nombre	$\|A_N x^k - b\|$	$\|Ax^k - b\|$	$\|x^k - x^*\|$	$\frac{\|x^k - x^*\|}{\|x^*\|}$	$\|x^k - x^*\|_i$	Tiempo (segs)
base17p1	$1,93280\times10^{-4}$	$1,37569\times10^{-3}$	$1,53890\times10^{-3}$	$4,06862\times10^{-4}$	$7,06782\times10^{-5}$	84,3547
base17p3	$8,45180\times10^{-3}$	4,94027	2,07338	$5,43043\times10^{-1}$	$7,47042\times10^{-2}$	84,3664
base17p5	$1,83393\times10^{-4}$	10,6980	2,27241	$5,95097\times10^{-1}$	$7,69837\times10^{-2}$	84,3459
base17p7	$3,25121\times10^{-6}$	19,0514	2,27233	$5,95077\times10^{-1}$	$7,70015\times10^{-2}$	84,2903
B&S *P1*	$6,31149\times10^{-20}$	$1,83297\times10^{-18}$	$1,29241\times10^{-16}$	$1,69893\times10^{-16}$	$1,00752\times10^{-17}$	9,3589
B&S *P2*	$2,00259\times10^{-17}$	$1,21243\times10^{-15}$	$4,51968\times10^{-14}$	$2,44098\times10^{-16}$	$2,26989\times10^{-15}$	9,3764
B&S *P3*	$1,94449\times10^{-5}$	$1,72224\times10^{-4}$	$4,86345\times10^{-3}$	$9,63944\times10^{-5}$	$6,43222\times10^{-4}$	9,3696
B&S *P4*	$1,33085\times10^{-12}$	$2,03107\times10^{-11}$	$2,03107\times10^{-11}$	$4,02561\times10^{-13}$	$1,01665\times10^{-12}$	9,3813
B&S *P5*	$3,74888\times10^{-18}$	$1,46507\times10^{-16}$	$7,86877\times10^{-15}$	$1,55960\times10^{-16}$	$4,34711\times10^{-16}$	9,3716
B&S *P6*	$4,75432\times10^{-18}$	$1,18176\times10^{-16}$	$6,83203\times10^{-15}$	$1,35412\times10^{-16}$	$3,67111\times10^{-16}$	9,3677

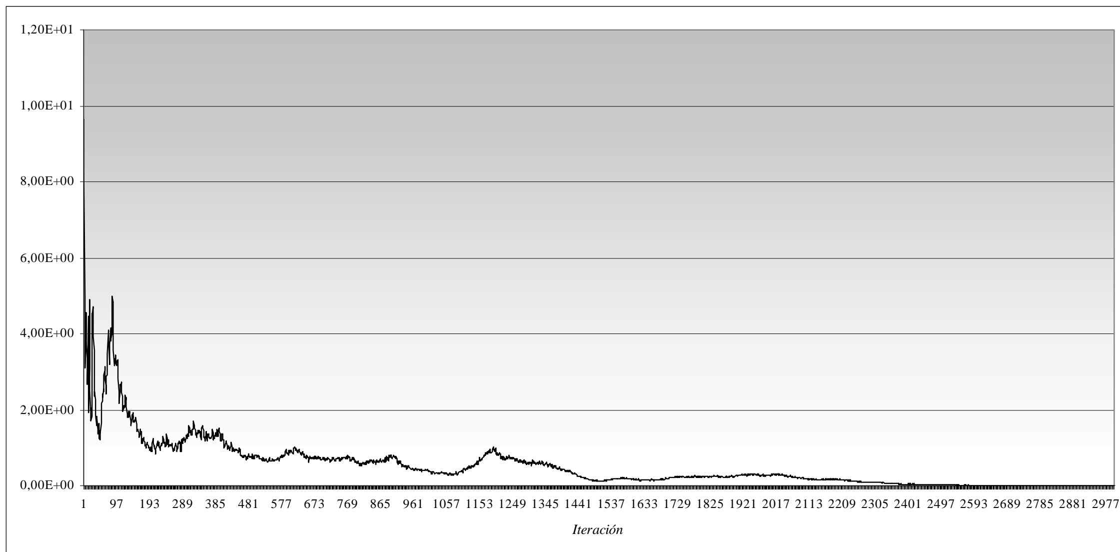
Se puede observar que para el conjunto de sistemas de B&S las soluciones alcanzadas son satisfactorias en todos los casos. Sin embargo esto no sucede para el conjunto de sistemas de SIDERCA. Para analizar estos casos se presentan, a continuación, las siguientes evoluciones obtenidas durante las primeras 3000 iteraciones de ejecución del algoritmo *AcCim*:

- la evolución del residuo $\|A_N x^k - b\|$ para la matriz A normalizada
- la evolución del residuo $\|Ax^k - b\|$ para la matriz A original
- la evolución de la distancia a la solución exacta $\|x^* - x^k\|$, calculada mediante un método directo

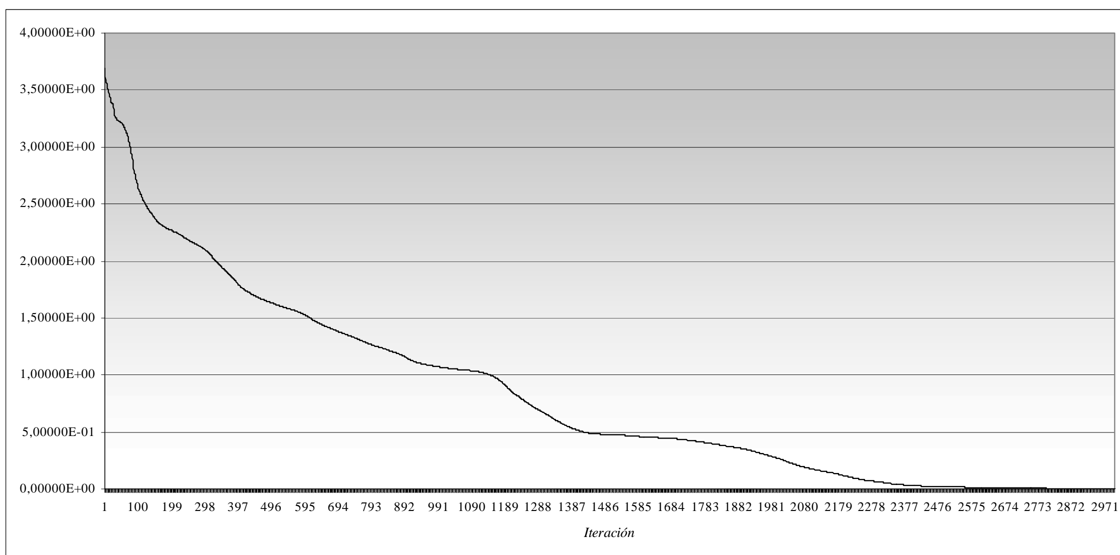
$$\left\| A_N x^k - b \right\|$$



$$\left\| A x^k - b \right\|$$

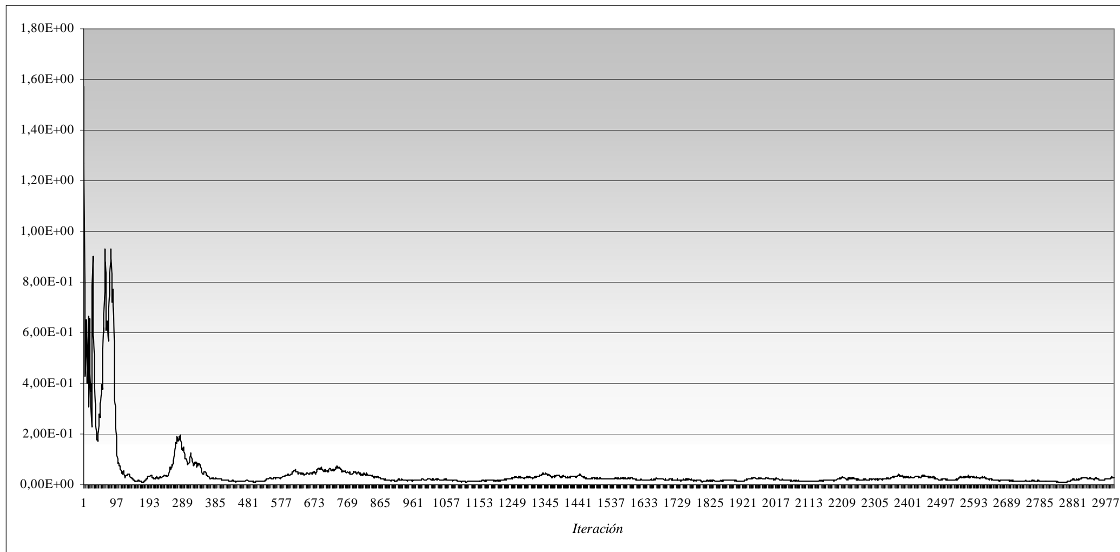


$$\left\| x^* - x^k \right\|$$

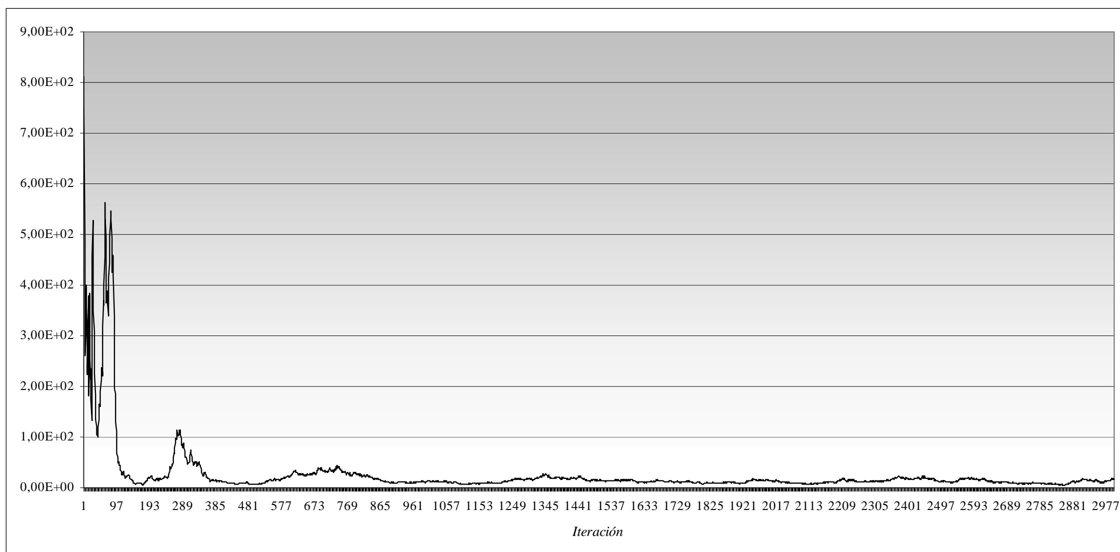


base17p1

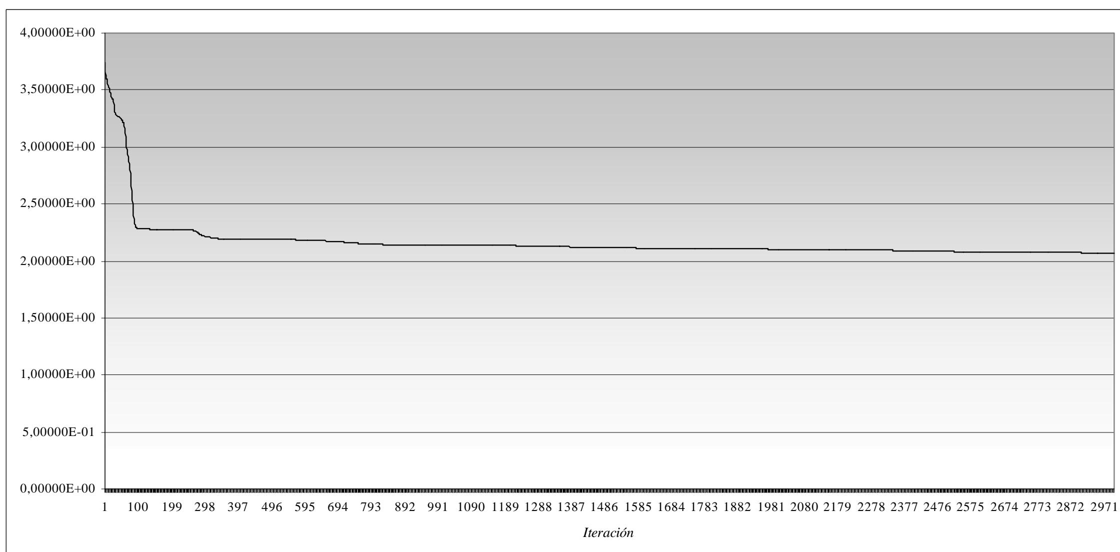
$$\left\| A_N x^k - b \right\|$$



$$\left\| A x^k - b \right\|$$

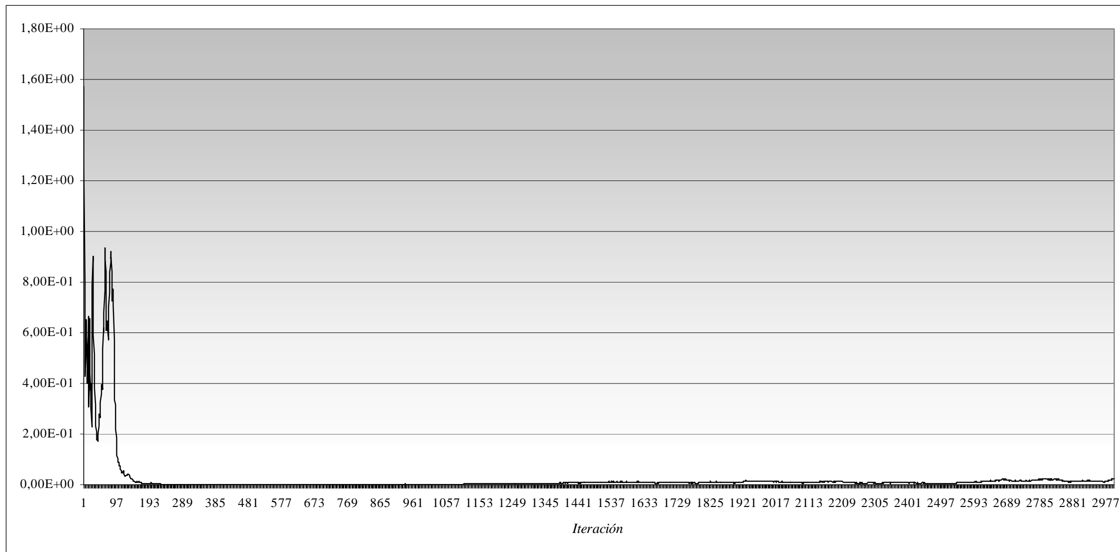


$$\left\| x^* - x^k \right\|$$

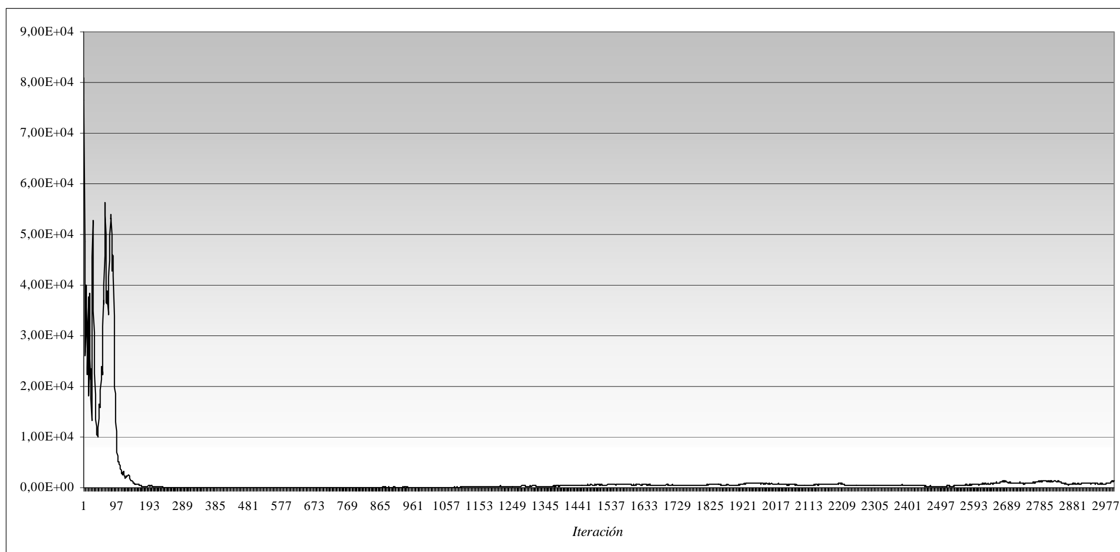


base17p3

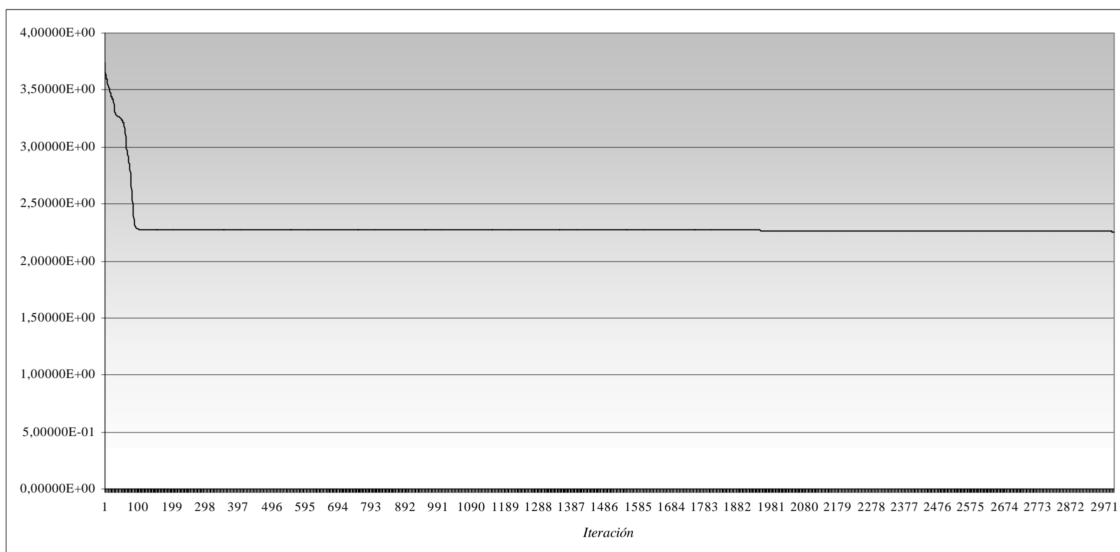
$$\left\| A_N x^k - b \right\|$$



$$\left\| Ax^k - b \right\|$$

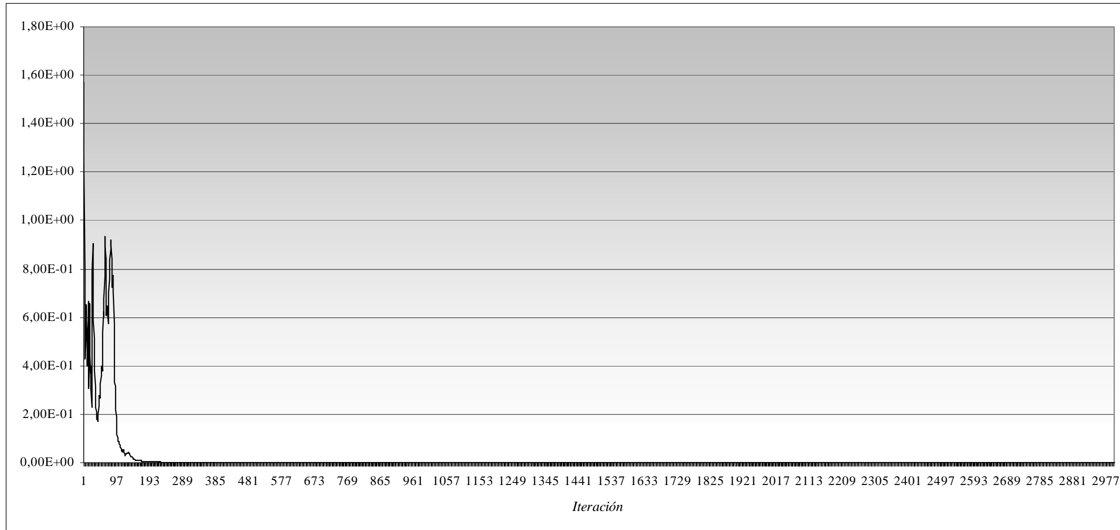


$$\left\| x^* - x^k \right\|$$

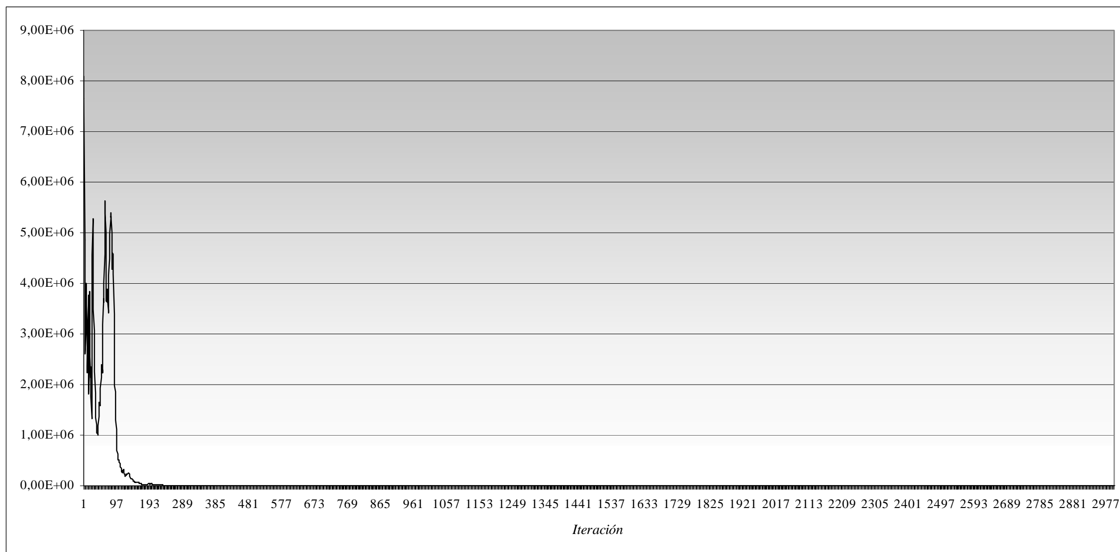


base17p5

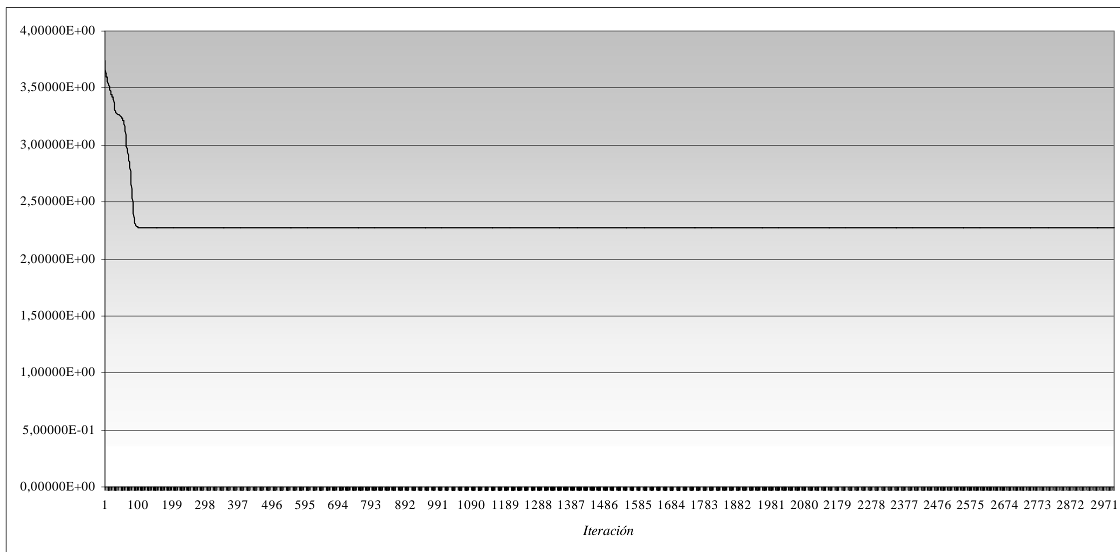
$$\left\| A_N x^k - b \right\|$$



$$\left\| A x^k - b \right\|$$



$$\left\| x^* - x^k \right\|$$



base17p7

Debido a los resultados obtenidos, se realizó en esta etapa un análisis sobre aquellos sistemas que presentaron problemas de convergencia. El mismo consiste en evaluar los ángulos entre todas las filas de la matriz con el objetivo de conocer la estructura geométrica del espacio que generan.
El resultado mostró que los sistemas problemáticos contenían filas casi paralelas, con lo cual resultan numéricamente singulares. Los resultados obtenidos se resumen en el siguiente cuadro, utilizando un $\varepsilon = 10^{-3}$.

Nombre	**Filas en las que** $1-\lvert\cos(\alpha)\rvert < \varepsilon$	$1-\lvert\cos(\alpha)\rvert$
base17p1	0	-
base17p3	17	10^{-6}
base17p5	17	10^{-10}
base17p7	17	10^{-14}

Los resultados anteriores permiten observar características de convergencia y alcance del algoritmo *AcCim* para resolver distintos tipos de problemas.

Como se mencionó en el apéndice de problemas, el conjunto de tests proporcionado por SIDERCA presenta una sucesión de sistemas cuyo número de condición es cada vez mayor a la vez que su geometría menos favorable.
Observando las características de estos sistemas y las evoluciones obtenidas, se puede ver que cuando el sistema se encuentra bien condicionado y su geometría es favorable, como es el caso de base17p1, el algoritmo presenta un comportamiento ideal en el que se pueden observar las siguientes características:

- La convergencia es rápida. Si bien continúa mejorando hacia el final de la evolución, ya a las 3000 iteraciones se cuenta con una buena solución.
- La evolución del residuo con respecto a la matriz original está dentro del mismo orden que la evolución del residuo con respecto a la matriz normalizada. Esto indicaría que con este tipo de sistemas se puede tener cierta confianza de que la solución obtenida con el sistema normalizado para cierta precisión será válida como solución del sistema original con una precisión de orden similar.
- La distancia de los sucesivos x^k a la solución exacta decrece rápidamente acompañando el comportamiento que presenta la evolución del residuo $Ax^k - b$.

Estas características van desapareciendo a medida que el número de condición del sistema se incrementa y la geometría se hace menos favorable.

También se observa que el residuo decrece rápidamente al principio para estancarse luego de cierta cantidad de iteraciones. Si bien la convergencia continúa, el acercamiento obtenido con cada iteración es cada vez menor. En vista de este resultado se decidió agregar a los criterios de parada uno que finalice el proceso iterativo cuando la ganancia

obtenida luego de una cantidad paramétrica de iteraciones resulte insuficiente como para justificar que el mismo continúe.

Precondicionamiento

Tratando de extender el alcance de este algoritmo para los sistemas en los que *AcCim* no obtuvo tan buenos resultados, y dado que se concluyó que esta dificultad coincidía con el mal condicionamiento de los mismos, se pasó a analizar la opción de usar precondicionadores para estos casos.

A continuación mostraremos los precondicionadores evaluados.

Balanceo

Este precondicionador ([21]) intenta obtener un balanceo de las magnitudes del sistema mediante el siguiente procedimiento:

1. Calcular ρ_0, el mayor radio entre dos elementos pertenecientes a una misma columna de la matriz A del sistema,
   $\rho_0 = \max_j \max_{r,s} \{|a_{rj}/a_{sj}|\}$, para $a_{sj} \neq 0$.
2. Escalar el sistema por filas. Para esto se divide cada fila i de la matriz A y su rhs correspondiente por
   $\left(\min_j\{|a_{ij}|\} \times \max_j\{|a_{ij}|\}\right)^{1/2}$, para $a_{ij} \neq 0$.
3. Escalar el sistema por columnas dividiendo cada columna i de A por
   $\left(\min_i\{|a_{ij}|\} \times \min_i\{|a_{ij}|\}\right)^{1/2}$, para $a_{ij} \neq 0$.
4. Calcular ρ, el mayor radio entre dos elementos pertenecientes a una misma columna de la matriz A modificada,
   $\rho = \max_j \max_{r,s} \{|a'_{rj}/a'_{sj}|\}$, para $a'_{sj} \neq 0$.
5. En caso que de cumplirse la condición
   $$\frac{|\rho - \rho_0|}{|\rho_0|} \geq 10^{-1}$$
   volver al punto 2, y sino terminar.

Vemos que con este procedimiento el sistema es modificado en cada iteración en los pasos 2 y 3.

En el paso 2 se realiza una división por filas que implica pre-multiplicar ambos lados del sistema por una matriz diagonal cuyos valores están dados por las magnitudes por las que se divide cada fila. Esto produce un sistema $A'x = rhs'$ equivalente al original.
En el paso 3 se realiza una división por columnas equivalente a post-multiplicar la matriz del sistema por una matriz diagonal cuyos valores están dados por las magnitudes por las que se divide cada columna. En este paso no se conserva la equivalencia del sistema, por lo que habrá que hacer cálculos posteriores a la resolución.
Se observa entonces que, luego de k iteraciones, se habrá aplicado la siguiente sucesión de operaciones sobre el sistema original:

$$(Df_k \times \ldots \times Df_1 \times Df_0 \times A \times Dc_0 \times Dc_1 \times \ldots \times Dc_k) \times x = Df_k \times \ldots \times Df_1 \times Df_0 \times rhs$$

$$(Df \times A \times Dc) \times x = Df \times rhs$$

Luego de resolver este nuevo sistema se deberá recuperar un sistema equivalente al original. Dado que la siguiente expresión es válida,

$$Dc_0 \times Dc_1 \times \ldots \times Dc_k \times Dc_k^{-1} \times \ldots \times Dc_1^{-1} \times Dc_0^{-1} = Dc_0 \times Dc_1 \times \ldots \times Dc_k \times (Dc_0 \times Dc_1 \times \ldots \times Dc_k)^{-1}$$

Para esto se deberán haber guardado la sucesión de matrices diagonales $Dc_0 \times Dc_1 \times \ldots \times Dc_k$ durante el proceso de balanceo. Esto permitirá poder pre-multiplicar la solución obtenida por la inversa de este producto.
Dado que la matriz Dc es diagonal y el producto de matrices diagonales también es diagonal, mantener esta sucesión solo requirió mantener una matriz diagonal, la cual es almacenada en un vector de la dimensión del sistema.

Precondicionamiento diagonal

Este método consiste en pre-multiplicar ambos lados del sistema por la inversa de la matriz D definida como:

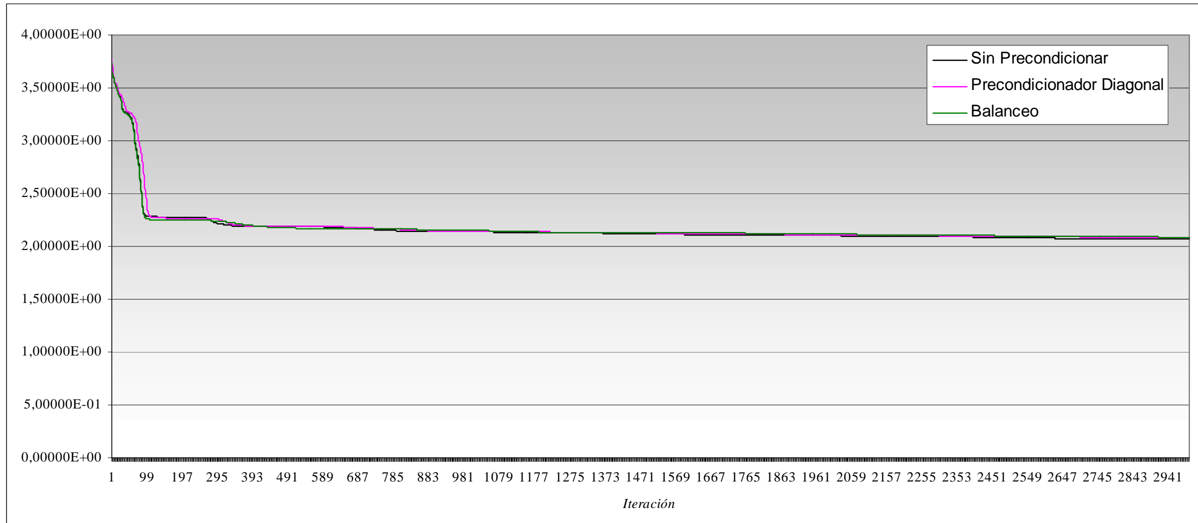
$$d_{ii} = \begin{cases} a_{ii} & a_{ii} \neq 0 \\ 1 & en\ otro\ caso \end{cases}$$

Es decir, dividir cada fila de A y elemento correspondiente de rhs por la inversa del elemento de la diagonal de dicha fila en caso de que este no sea nulo.
Este método transforma el sistema original en uno equivalente, con lo que la solución obtenida sobre el nuevo sistema resultará válida para el primero.

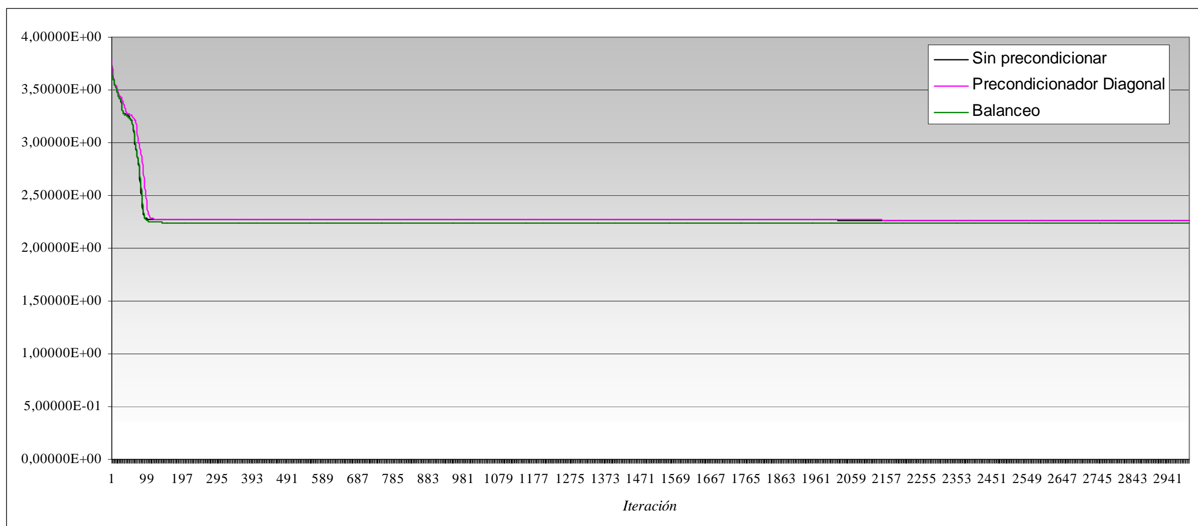
Resultados obtenidos

Ninguno de los dos métodos resultó útil para mejorar la convergencia sobre los sistemas que se trataban de resolver en esta etapa. En los casos evaluados la misma no presentó diferencias significativas haciendo uso de precondicionadores. Esto puede verse en los siguientes gráficos.

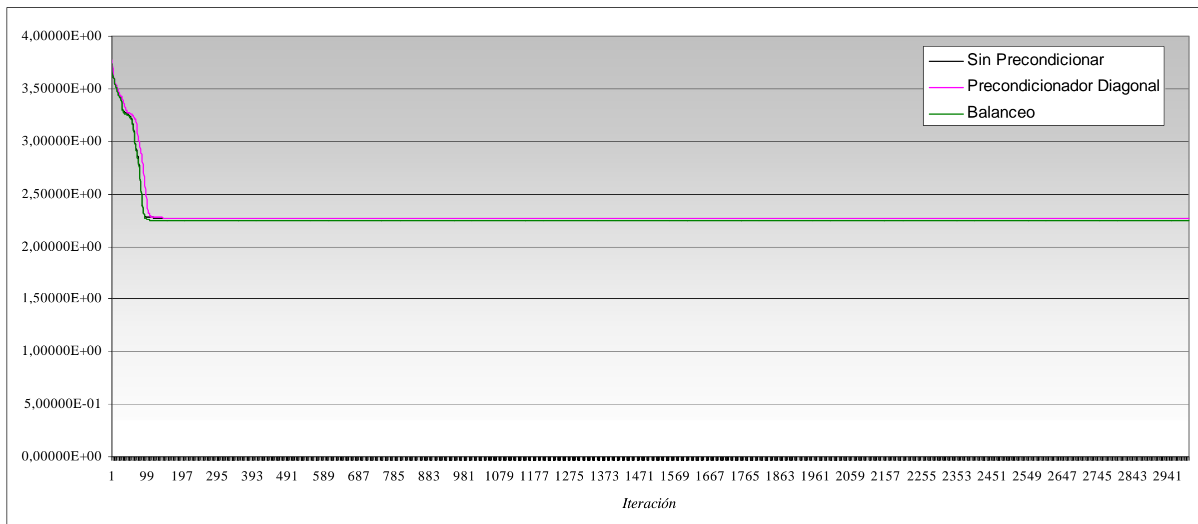
base17p3 - Evolución de $\left\| x^* - x^k \right\|$ con el uso de distintos precondicionadores



base17p5 - Evolución de $\left\| x^* - x^k \right\|$ con el uso de distintos precondicionadores



base17p7 - Evolución de $\left\| x^* - x^k \right\|$ con el uso de distintos precondicionadores



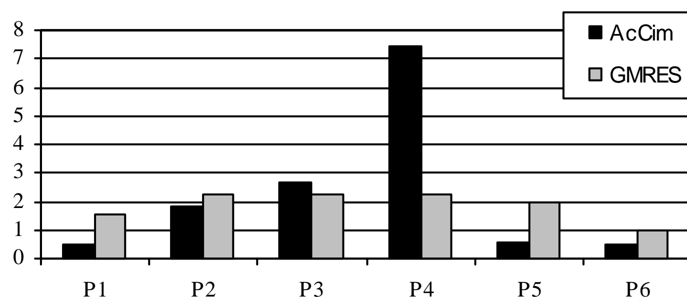
Más allá de estos resultados, se notó que no es conveniente realizar operaciones sobre los sistemas para la obtención de otros equivalentes cuando estos son tan inestables como los sistemas que estamos tratando de resolver, en especial si el método requiere un tratamiento posterior a la obtención de la solución.
Por este motivo, en la sección siguiente evaluaremos otra opción, diferente al uso de este tipo de precondicionadores.

Comparación con otros solvers

Para finalizar, en esta sección compararemos la performance en tiempo y precisión obtenida por *AcCim* con la obtenida por el solver iterativo para matrices ralas GMRES (Generalized Minimal RESidual, [24]).
Para esta comparación fueron usados los problemas de B&S ya presentados sin hacer uso de precondicionadores.
A continuación se muestran los resultados obtenidos por ambos algoritmos.

Tiempo

Solver	B&S *P1*	B&S *P2*	B&S *P3*	B&S *P4*	B&S *P5*	B&S *P6*
AcCim	0,521184	1,84366	2,67326	7,45664	0,564128	0,466528
GMRES	1,578	2,25	2,25	2,25	1,969	1,016



Precisión obtenida

Solver	B&S *P1*	B&S *P2*	B&S *P3*	B&S *P4*	B&S *P5*	B&S *P6*
AcCim	$9{,}682\times10^{-8}$	$1{,}955\times10^{-5}$	$1{,}540\times10^{-4}$	$6{,}119\times10^{-6}$	$2{,}520\times10^{-6}$	$5{,}733\times10^{-6}$
GMRES	$2{,}167\times10^{-6}$	$5{,}064\times10^{-2}$	$4{,}745\times10^{-2}$	$1{,}088\times10^{+2}$	$3{,}339\times10^{-4}$	$2{,}467\times10^{-4}$

La precisión obtenida por *AcCim* fue mejor en todos los casos, y en los dos problemas en que *AcCim* no es superado en tiempo este logra una solución de mayor precisión.
Por otro lado, es importante destacar que GMRES no garantiza convergencia a la solución exacta en todo tipo de matrices mientras que *AcCim* si lo hace.

Descomposición en bloques

Dado que la aplicación de precondicionadores sobre los sistemas más difíciles de SIDERCA no produjo una mejora en las soluciones obtenidas, la siguiente alternativa a evaluar fueron los métodos de descomposición en bloques y posterior resolución del sistema que fueron presentados en la introducción teórica.

Algoritmo de descomposición

Como ya se dijo en la introducción teórica, este proceso consiste en particionar la matriz a fin de obtener una división en bloques bien condicionados que ayuden a obtener mejores resultados al hacer las proyecciones.
Vale aclarar que la división en bloques que resulta de este proceso puede ser reutilizada para resolver diferentes sistemas con la misma matriz.
A continuación explicamos algunos detalles relevantes sobre la implementación de este método, cuyo pseudo-código mostramos en el apéndice de algoritmos.

Detalles de implementación

Para representar la partición de una matriz, se utilizó una estructura como la que ilustra el siguiente gráfico.

Estructura de partición de una matriz de 5×5 en dos bloques

`blockStr`	1	3	6		
`blockInd`	1	4	2	3	5

1
2
3
4
5

El vector `blockStr`, cuya dimensión coincide con la cantidad de bloques total (que en adelante notaremos como q) más uno, contiene en `blockStr[i]` la posición en la que se comienzan a enumerar las filas del *i*-ésimo bloque en el segundo vector `blockInd`. Por lo tanto este último, de dimensión n, contiene los números de todas la filas de la matriz en el orden en que se encuentran repartidos en cada bloque. El número de la última fila del *i*-ésimo bloque se encontrará en la posición `blockStr[i+1]-1`, es decir, un lugar antes de que comiencen a numerarse las filas del siguiente bloque.

Los bloques obtenidos son armados teniendo en cuenta el máximo de filas por bloque (μ) y la tolerancia para el número de condición de cada uno (κ), ambos valores definidos como parámetros.

Durante la ejecución de este algoritmo son calculadas tanto la partición en bloques de la matriz del sistema sobre esta estructura, como la factorización LDL^t de $A_iA_i^t$ para cada bloque. Como ya se dijo, lo que se guarda realmente de estas factorizaciones son las matrices L^{-1} y D^{-1}, generadas por el algoritmo y requeridas por los algoritmos de resolución.

Análisis de costo

Costo temporal

Si bien el costo depende de las características del sistema que está siendo particionado, dado que al final del proceso de descomposición los bloques generalmente llegan a tener el máximo de filas permitido μ, y a los fines de estimar el costo promedio del proceso, podemos plantear lo siguiente.

Dada una matriz de $n \times n$ y una partición de q bloques de a lo sumo μ filas cada uno, el costo de obtener la partición es el costo de armar los bloques sumado al costo de haber intentado agregar una cantidad de filas que fueron rechazadas.

El costo de este proceso está determinado, principalmente, por los siguientes pasos en el algoritmo:

$$l_{j+1} = D_j^{-1} L_j^{-1} A_i a_l$$

$$\delta_{j+1} = 1 - l_{j+1}^t D_j l_{j+1},$$

donde j indica la cantidad de filas ya agregadas al bloque y se incrementará desde 1 hasta μ durante la formación del bloque, i el número de bloque actual y l el número de la fila que se está intentando agregar. Teniendo en cuenta que la dimensión de las matrices D_j^{-1} y L_j^{-1} es de $j \times j$, la dimensión de A_i es de $j \times n$, y la de a_l es de n, se puede ver que la cantidad de operaciones necesarias para resolver estas cuentas es de $2j^2 + n \times j$ para el primer paso y $2n+1$ para el segundo. Luego, si la fila efectivamente es agregada al bloque, el costo aumenta en j^2 operaciones.

Esto suma un total de $n \times j + 2j^2 + 2n + j^2 + 1$ operaciones para la asignación de una fila a un bloque de j filas, lo que está en el orden de $O(n \times j)$.

Este costo está determinado por la eficiencia con la que se pueda resolver el producto del bloque corriente A_i por el vector a_l. Esto debe ser tenido en cuenta especialmente en el caso de que la matriz A sea rala, ya que en ese caso esta operación es más complicada

de realizar que el resto de los productos y puede implicar un número mayor de operaciones.

En una implementación en la que el producto $A_i a_l$ se pueda hacer en orden $n \times j$, podemos ver que luego de agregar μ filas a un bloque, el costo total habrá sido del orden de:

$$0 + 1n + 2n + 3n + ... + \mu\, n = n\frac{\mu(\mu+1)}{2}.$$

Por lo que el costo de particionamiento estará dado por:

$$q \times n \times \frac{\mu(\mu+1)}{2} + CFR.$$

Lo que se encuentra en el orden de $O(qn\mu^2 + CFR)$.

Si bien el costo sumado por las filas que son rechazadas, es decir por CFR, puede ser muy grande y aumentar el orden en un peor caso, a los fines de buscar un costo promedio es válido considerar que este es menor que el de la formación de los bloques. Por este motivo, y para simplificar análisis futuros, de ahora en más diremos que el costo del algoritmo de descomposición está en el orden de $O(qn\mu^2)$.

Costo espacial

Durante la ejecución de este procedimiento, se le suma al espacio requerido para almacenar la matriz rala, el requerido para almacenar la estructura de descomposición en bloques y las matrices L^{-1} y D^{-1} correspondientes a la factorización de $(A_i A_i^t)^{-1}$ para cada bloque. Si bien este requerimiento depende de la cantidad de bloques y de filas por bloque que resulten de la descomposición, se puede decir que hacia el final del procedimiento el espacio requerido será de $[n + 2nz] + [q + 1 + n] + [q(\mu^2 + \mu)]$.

El primer término corresponde al espacio de almacenamiento de una matriz rala de dimensión n y nz valores no nulos que ya fue calculado anteriormente.

El segundo término corresponde al espacio de la estructura de descomposición que se explicó en una sección anterior y que es generada completamente hacia el final de este procedimiento.

El tercer término corresponde al espacio de las matrices L^{-1} y D^{-1} para cada bloque definido. Este es así porque, si bien cada una de estas matrices tiene una dimensión de μ^2, la matriz D^{-1} es una matriz diagonal que puede ser almacenada como un vector de dimensión μ.

Este costo se puede reescribir como $2n + q(\mu^2 + \mu + 1) + 2nz + 1$.

Algoritmos de resolución

Utilizando la estructura de descomposición en bloques y la representación de matrices rala como base, se implementaron los algoritmos de resolución presentados en el apéndice de algoritmos como *ALG1* y *ALG2*.
A continuación explicamos algunos detalles relevantes sobre la implementación de los métodos de resolución que hacen uso del sistema particionado y hacemos un breve análisis de su costo.

Detalles de implementación

Una vez encontrada la partición del sistema, comienza el proceso iterativo. Observando cualquiera de los dos algoritmos de resolución presentados, podemos ver que ambos requieren del cálculo de inversas. Esto podría ser un problema si requiriera resolver efectivamente un problema completo de inversión en cada iteración. Afortunadamente, este no es el caso.
Esta situación se presenta en dos pasos. El primero es el paso 7.1.1:

$$d_i^k = A_i^t \left(A_i A_i^t\right)^{-1}\left(b_i - A_i x^k\right).$$

Sin embargo, dado que el algoritmo de partición ha guardado las matrices L^{-1} y D^{-1} correspondientes a la factorización LDL^t de $\left(A_i A_i^t\right)^{-1}$, la inversa se consigue multiplicando $L^{-1^t} D^{-1} L^{-1}$.

El segundo paso en el que se requiere de la inversión de matriz es el 7.3:

$$w^k = \left(D^{k^t} D^k\right)^{-1}\left(\left\|d_1^k\right\|^2, \ldots, \left\|d_{q_k}^k\right\|^2\right)^t$$

Si bien no se dispone de la factorización LDL^t de la matriz de direcciones $\left(D^{k^t} D^k\right)^{-1}$, se la puede conseguir haciendo uso del proceso de partición en bloques. Con este fin el mismo es llamado para la formación de un bloque de, potencialmente, toda la matriz. Esto permite acceder a la factorización buscada y al mismo tiempo descartar filas que empeoren el número de condición de esta matriz.

Análisis de costo

Costo temporal

Según se puede analizar observando el código presentado, el costo del algoritmo de resolución *ALG1*, por iteración, es de $q\left(\mu^3 + n\mu^2 + \mu^2 + nz\right) + nq^2 + 2q_k$ siendo q la cantidad de bloques en los que se particionó el sistema, μ la cantidad máxima de filas por bloque, nz la cantidad de valores no nulos de la matriz original de dimensión n y q_k

la cantidad de direcciones linealmente independientes de D^k que se calculan durante la descomposición de $\left(D^{k^t} D^k\right)^{-1}$.

El primer término corresponde al cálculo de las direcciones que formarán parte de la matriz D^k, el segundo a la factorización de la misma utilizando el método de descomposición por bloques teniendo en cuenta el costo ya calculado para este algoritmo y sabiendo que se forma un solo bloque de hasta q filas, y el último al producto final para la obtención de los pesos y posterior cálculo de la dirección final.

Dependiendo del tamaño del sistema y de las restricciones espaciales que tengamos, estos costos pueden reducirse. Por ejemplo, dada una partición, el producto $A_i^t\left(A_i A_i^t\right)^{-1}$ es el mismo para todas las iteraciones del algoritmo de resolución y no necesita ser recalculado si se dispone del espacio físico para conservarlo entre dos iteraciones. En este caso el costo podría reducirse a $q(\mu^2 + nz) + nq^2 + 2q_k$, evitando así repetir las $q\left(\mu^3 + n\mu^2\right)$ operaciones correspondientes a este producto en cada iteración.

En cuanto al costo del algoritmo de resolución *ALG2*, este agrega el cálculo de las proyecciones de las q direcciones calculadas, lo que suma $4nq$ operaciones por iteración dando un total de $q\mu^2 + nq^2 + 2q_k + 4nq$ en caso de poder hacerse la optimización explicada en el párrafo anterior.

Más allá de dar la idea general del costo de estos algoritmos, lo que es necesario destacar es que, como se puede deducir de la fórmulas, el costo de partición crece rápidamente si el valor de μ crece. Por lo que para bajar el costo del mismo convendría que este valor no sea elevado. Sin embargo, si este valor es muy pequeño, se generarán más bloques (es decir, aumentará q), y esto hará aumentar el costo del algoritmo de resolución.

Costo espacial

En cuanto al costo espacial, se mantiene en estos algoritmos el requerimiento de memoria de la descomposición en bloques, ya que tanto la matriz como las estructuras y factorizaciones que fueron generadas por este son usadas al resolver el sistema. A esto se le suma, además, el de los vectores y matrices que forman parte del algoritmo de resolución, resultando en el siguiente costo:

$$[n + 2nz] + [q + 1 + n] + [q(\mu^2 + \mu)] + [2n + q + n \times q + (q^2 + q)].$$

En donde los tres primeros términos corresponden a la matriz y estructuras de descomposición según lo que ya fue explicado y el cuarto término contempla el espacio para los vectores b y x^k de dimensión n, el vector w de dimensión q y la matriz D^k

de dimensión $n \times q$ que figuran en el algoritmo. Además, dado que en cada iteración se hace la descomposición en un bloque de hasta q filas de esta matriz densa D^k, se le suma un espacio de $q^2 + q$ correspondiente a las matrices L^{-1} y D^{-1} de la factorización de $\left(D^{k^t} D^k\right)^{-1}$. Notar que la estructura de descomposición de D^k no es almacenada dado que la misma es una matriz densa que no necesita ser conservada, por lo que el único bloque obtenido se guarda en el mismo espacio de la matriz.

La expresión final para este costo, será entonces de:

$nq + 4n + 2nz + q^2 + q(\mu^2 + \mu + 3) + 1$.

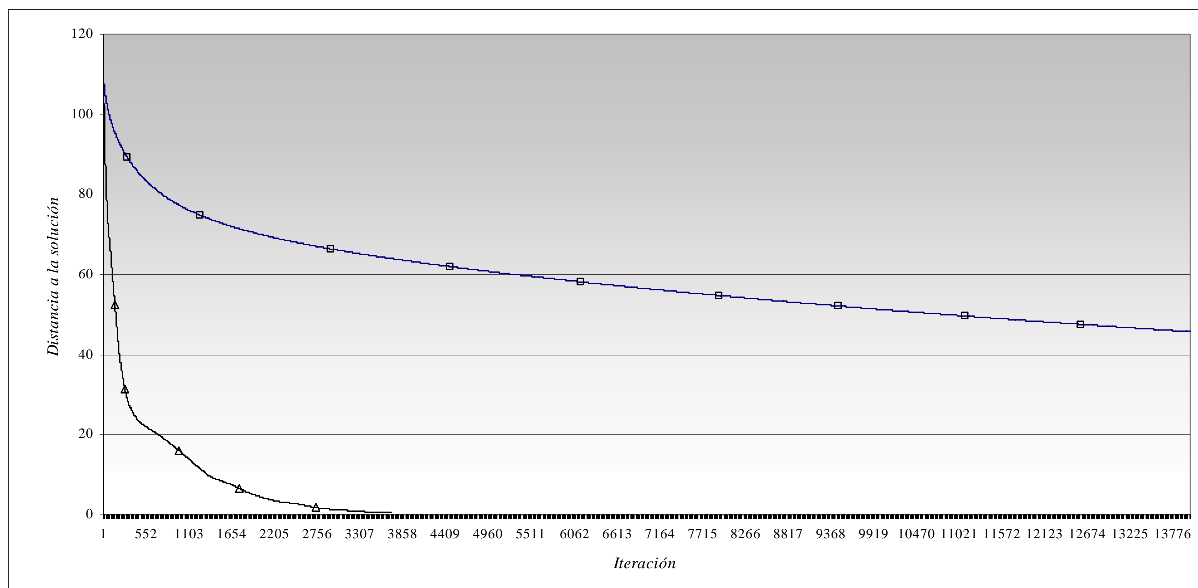
En esta implementación, las matrices ralas simétricas son extendidas a matrices ralas no simétricas, por lo que en este caso no es válida la reducción de espacio para este tipo de matrices que era válida en la versión de *AcCim* para matrices ralas.

Resultados

ALG1 vs. ALG2

En primer lugar nos interesa evaluar la aceleración en la convergencia que se consigue utilizando *ALG2* con respecto a *ALG1* según lo que se explicó en la introducción teórica. En el siguiente gráfico podemos ver la evolución de la distancia a la solución a través de n iteraciones de ambos algoritmos, siendo n la dimensión de la matriz, para el problema base17p1 de SIDERCA.

Evolución de la distancia a la solución obtenida por *ALG1* y *ALG2*



En este gráfico la línea cuadriculada corresponde a *ALG1* y la línea triangulada corresponde a *ALG2*.

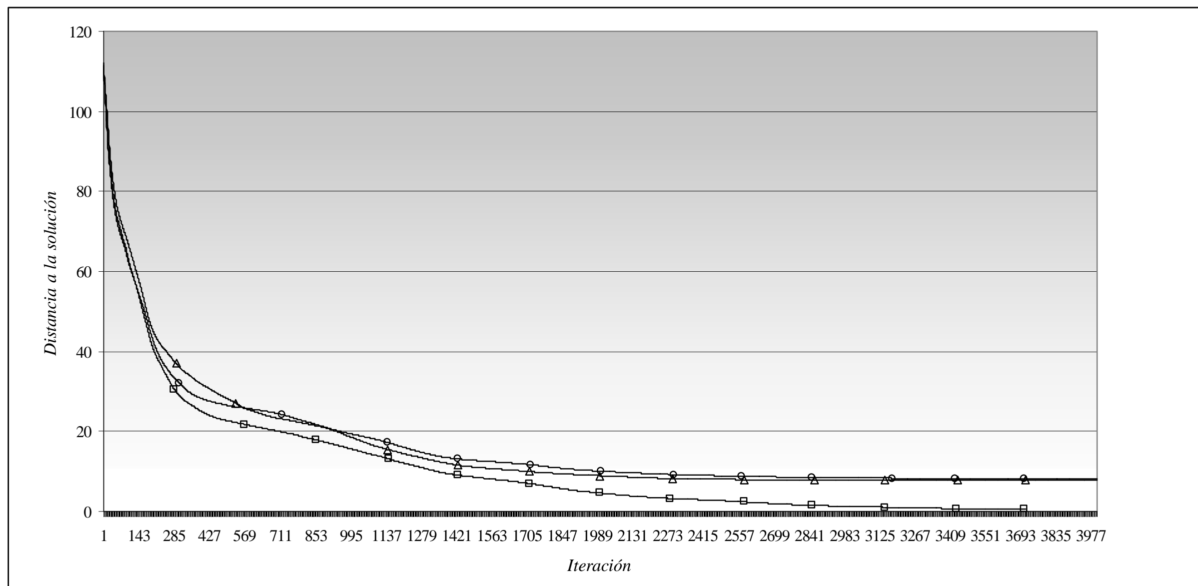
Como se esperaba, *ALG2* converge en forma más acelerada, consiguiendo antes de la iteración número 500 el acercamiento a la solución que *ALG1* consigue recién después de n iteraciones (donde n es la dimensión de la matriz). Este comportamiento se observó en todos los casos evaluados.

Variaciones de μ

Otro aspecto que es importante analizar es cómo afecta a la convergencia el tamaño de los bloques en los que se descompone la matriz. Para esto corrimos el algoritmo *ALG2* sobre el sistema base17p1 evaluando tanto la convergencia obtenida como el tiempo consumido utilizando distintos valores de μ.

En el siguiente gráfico se pueden ver las evoluciones a la solución exacta obtenidas en estas ejecuciones.

Evolución de la distancia a la solución obtenida por *ALG2* para distintos valores de μ



En este gráfico la línea triangulada representa la evolución con μ = 50, la línea cuadriculada la evolución con μ = 100 y la línea con círculos la evolución con μ =200. En estos casos, los tiempos cada 100 iteraciones fueron los siguientes.

Tiempo requerido para ejecutar 100 iteraciones del algoritmo *ALG2* sobre base17p1 para distintos valores de μ

	μ = 50	μ = 100	μ = 200
Tiempo (segs)	313	119	82

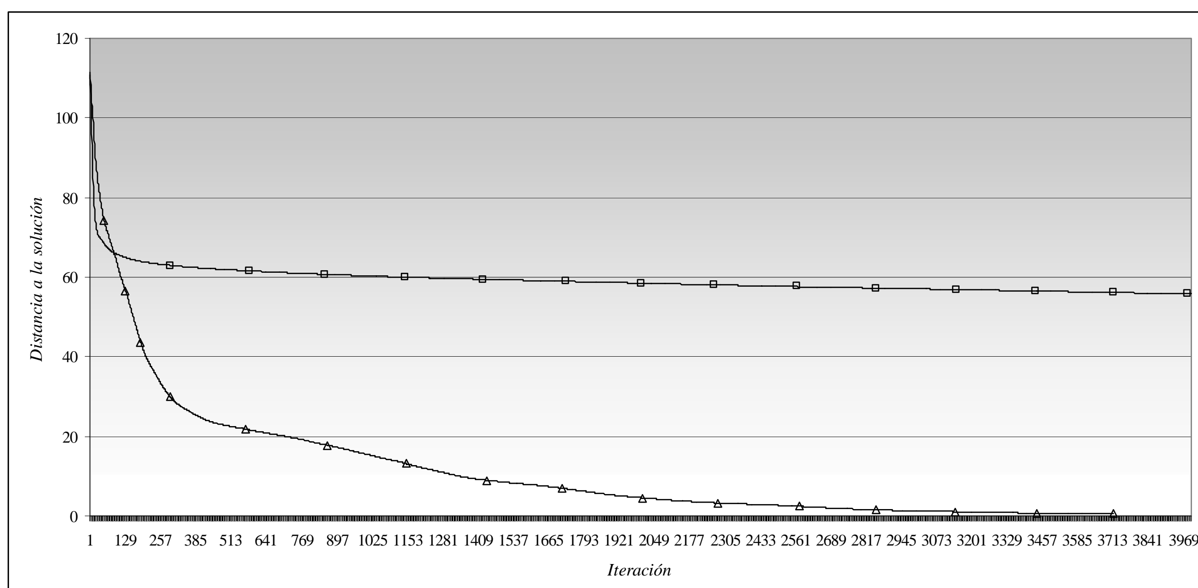
Por un lado se puede ver que el tiempo por iteración decrece al crecer el tamaño de los bloques. Esto se debe a que el proceso de descomposición en bloques de la matriz de

direcciones, que según vimos se lleva a cabo en cada iteración y es la operación que más tiempo consume, es más rápida cuanto menor es la cantidad de bloques en la que se particionó el sistema (esto es válido si se puede ejecutar la optimización explicada en el análisis de costo temporal del método).
Por otro lado vemos que la mejor convergencia se obtuvo particionando este sistema, de dimensión 13345, en bloques de hasta 100 filas. En los otros dos casos, vemos que la convergencia se estanca antes de lograr un buen acercamiento a la solución.
De esto podemos observar que la elección de un valor adecuado para μ es fundamental en el éxito de la ejecución. Si bien este valor variará entre distintos sistemas, por una cuestión de eficiencia es recomendable que sea un valor tal que el algoritmo de descomposición no produzca una gran cantidad de bloques.

Sistemas mal condicionados

Una característica de estos algoritmos que es importante evaluar es el comportamiento obtenido sobre sistemas mal condicionados. Idealmente, se necesita que el mal condicionamiento no afecte la convergencia a la solución.
El siguiente gráfico muestra una comparación entre el comportamiento obtenido por *ALG2* sobre los problemas base17p1, sistema con número de condición aproximado de 10, y base17p3, sistema con número de condición aproximado de 1000. Las evoluciones corresponden a ejecuciones con $\mu = 100$, ya que este fue, en ambos casos, el valor de μ con el que mejor convergencia se obtuvo.

Evolución de la distancia a la solución durante la ejecución del algoritmo ALG2 para los sistemas base17p1 y base17p3



En este gráfico la línea triangulada corresponde al sistema base17p1 y la línea cuadriculada corresponde al sistema base17p3.
Se puede ver que con el sistema base17p1 el algoritmo *ALG2* converge rápidamente mientras que con el sistema base17p3 converge rápidamente sólo en un principio para estancarse a las pocas iteraciones a una gran distancia a la solución. Este mismo comportamiento se observó con el resto de los problemas de SIDERCA, por lo que en estos casos se puede ver que el mal condicionamiento si afectó la convergencia.

Conclusiones

Si bien los experimentos realizados en [1] sobre matrices densas ya demostraban que las técnicas de aceleración aplicadas en *AcCim* son efectivas para resolver una amplia gama de problemas, aquí presentamos una nueva versión de este solver sobre el que pudimos investigar una serie de optimizaciones que mejoraron su performance. De estas optimizaciones, la que ofreció mayor ventaja fue la paralelización mediante la utilización de una librería de threads. Esta versión, que explota explícitamente el paralelismo intrínseco del algoritmo, produjo mejoras de más de un 20% en el tiempo de ejecución con respecto a la versión secuencial.

Además pudimos comprobar, mediante una nueva implementación, que estos resultados pueden extenderse a matrices ralas de dimensiones mayores en una implementación de costo computacional significativamente menor que resulta competitiva en tiempo con otros solvers iterativos para matrices ralas usados ampliamente hoy en día, superándolos en su comportamiento numérico y sin el uso de precondicionadores.

Esta implementación también nos permitió detectar un grupo de problemas que presentan dificultades de convergencia, a través de los cuales pudimos evaluar distintos métodos para ampliar el alcance del algoritmo.

Si bien no pudimos obtener métodos generales para lograr este objetivo, esto nos llevó a implementar un método acelerado de resolución con descomposición en bloques que no había sido implementado para matrices ralas.

Trabajo Futuro

Los algoritmos implementados en el transcurso de este trabajo sirven para resolver sistemas lineales consistentes. En [19] y [20] se presentan nuevos métodos que sirven para buscar la solución de mínimos cuadrados a problemas inconsistentes. Los algoritmos presentados utilizan un esquema de proyecciones oblicuas incompletas sobre un conjunto convexo para minimizar una norma del residuo. Dado que estas proyecciones oblicuas tienen un menor costo computacional que las proyecciones exactas, se espera que una implementación de estos métodos sea más rápida que las aquí presentadas.

En la parte matemática queda estudiar el desarrollo de nuevos precondicionadores generales, que además de mejorar el condicionamiento de la matriz tengan en cuenta su estructura rala.

Por otra parte, si bien hicimos implementaciones paralelas del método *AcCim*, esta técnica no se aplicó a la versión para matrices ralas, por lo que ese es un campo que aún queda por explorar.

Finalmente, como ya se mencionó, todavía es posible aplicar algunas optimizaciones a las implementaciones aquí presentadas. Entre estas optimizaciones se incluyen:

- El estudio de los patrones de acceso a memoria, de forma de optimizar el uso de caches
- Utilización más extensiva de profilers
- Implementación de operaciones críticas en lenguajes de más bajo nivel

Apéndices

Formatos de almacenamiento de matrices ralas

Las matrices ralas son aquellas en las que la mayoría de sus coeficientes son ceros.
Evitando almacenar las entradas con valor nulo, es posible reducir considerablemente los requerimientos de almacenamiento de las mismas, con la consecuente posibilidad de trabajar con sistemas mucho más grandes.
Existen varios formatos de almacenamiento para esta clase de matrices. A continuación presentaremos algunos de ellos mediante un ejemplo.

Sea $A \in \Re^{n \times m}$,

$$A = \begin{pmatrix} 11 & 0 & 13 & 14 & 0 & 0 \\ 0 & 22 & 23 & 0 & 25 & 0 \\ 31 & 32 & 33 & 0 & 35 & 0 \\ 41 & 0 & 0 & 44 & 0 & 0 \\ 0 & 52 & 53 & 54 & 55 & 0 \\ 0 & 0 & 0 & 0 & 0 & 66 \end{pmatrix},$$

con $n = m = 6$.

Notar que los ejemplos usan la convención `FORTRAN`, en la que los índices de los arrays comienzan en 1.

Indice de filas y columnas

El método más sencillo consiste en almacenar, para cada elemento no nulo, la fila y la columna a la que pertenece. De esta forma, si la matriz tiene nz entradas no nulas, el requerimiento de memoria será del orden de $O(3nz)$, dependiendo del tamaño del tipo de datos usado para los índices y para los elementos en sí.
Nuestra matriz de ejemplo sería almacenada de la siguiente forma:

`row`	1	1	1	2	2	2	3	3	3	3	4	4	5	5	5	5	6
`col`	1	3	4	2	3	5	1	2	3	5	1	4	2	3	4	5	6
`value`	11	13	14	22	23	25	31	32	33	35	41	44	52	53	54	55	66

Compressed Sparse Row (CSR)

Observando la estructura obtenida con el método anterior, podemos ver que en el array de filas los elementos se repiten, dado que la matriz está almacenada por filas.
Considerando esto, se puede comprimir dicho array haciendo que el *i*-ésimo elemento de `row` indique dónde comienza la *i*-ésima fila. De esta forma queda:

rowstr	1	4	7	11	13	17	18										
colind	1	3	4	2	3	5	1	2	3	5	1	4	2	3	4	5	6
value	11	13	14	22	23	25	31	32	33	35	41	44	52	53	54	55	66

Notar que `rowstr[i+1]-rowstr[i]` es igual a la cantidad de elementos no nulos de la *i*-ésima fila y que `rowstr[n+1]` (donde n es la cantidad de filas de la matriz) debe fijarse de forma de cumplir este invariante.
Con esta modificación se pasa de un orden $O(3nz)$ a $O((n+1)+2nz)$.

Compressed Sparse Column (CSC)

Dado que en `FORTRAN` las matrices se almacenan por columnas en lugar de por filas (como es la costumbre en `C/C++`), existe un formato análogo a CSR, donde en lugar de indicar el comienzo de cada fila, se indica el de cada columna.
De esta forma, el ejemplo quedaría:

colstr	1	4	7	11	14	17	18										
rowind	1	3	4	2	3	5	1	2	3	5	1	4	5	2	3	5	6
value	11	31	41	22	32	52	13	23	33	53	14	44	54	25	35	55	66

Matrices simétricas

En el caso de las matrices simétricas, el formato CSR almacena solo el triángulo superior, y el formato CSC el triángulo inferior. En ambos casos también se guarda la diagonal de la matriz.

Formato de almacenamiento en disco

Además de la estructura utilizada para los datos, al guardar los elementos de cada vector en disco estos se pueden almacenar en formato little-endian o big-endian (en memoria siempre se utiliza little-endian).

Nuestra implementación contiene código `C++` para manejar los siguientes formatos: little-endian, big-endian, CSC simétrico y no simétrico, CSR simétrico y no simétrico, y también para convertir de uno a otro.

Descripción de los problemas

En esta sección presentaremos los problemas usados para probar los diferentes algoritmos e implementaciones.

Sistemas H_n

Estos son problemas densos muy simples, de dimensión $n \times n$, generados en forma aleatoria pero asegurando que sean convergentes. Para esto se los hace diagonales dominantes, o sea que $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$ para todo *i*.

Bramley & Sameh

Estos seis problemas ([14], llamados comúnmente B&S *P1* a *P6*) surgen de la discretización, usando diferencias centrales, de ecuaciones diferenciales parciales elípticas de la forma $au_{xx} + bu_{yy} + cu_{zz} + du_x + eu_y + fu_z + gu = F$, donde $a \ldots g$, F son funciones de (x, y, z) y el dominio es el cubo unitario $[0,1] \times [0,1] \times [0,1]$. Sobre los sistemas fueron aplicadas las condiciones de Dirichlet para tener una solución conocida contra la que calcular errores.

Cuando la discretización se hace utilizando n_1 puntos en cada dirección, se obtiene un sistema ralo, no simétrico, de orden $n = n_1^3$. En los sistemas de prueba se utilizó $n_1 = 24$, por lo que $n = m = 13824$, y los sistemas resultantes tienen 93312 elementos no nulos.

Los problemas están definidos de la siguiente manera:

- *P1*: $\Delta u + 10^3 u_x = F$ con solución $u(x, y, z) = xyz(1-x)(1-y)(1-z)$ y $\kappa(A) \approx 4$
- *P2*: $\Delta u + 10^3 e^{xyz}(u_x + u_y - u_z) = F$ con solución $u(x, y, z) = x + y + z$ y $\kappa(A) \approx 57$
- *P3*: $\Delta u + 10^2 xu_x - yu_y + zu_z + 10^2 (x + y + z)(u/xyz) = F$ con solución $u(x, y, z) = e^{xyz} \sin(\pi x)\sin(\pi y)\sin(\pi z)$ y $\kappa(A) \approx 40000$
- *P4*: $\Delta u - 10^5 x^2 (u_x + u_y + u_z) = F$ con la misma solución que *P3* y $\kappa(A) \approx 4786$
- *P5*: $\Delta u - 10^3 (1 + x^2) u_x + 10^2 (u_y + u_z) = F$ con la misma solución que *P3* y $\kappa(A) \approx 36$
- *P6*: $\Delta u - 10^3 ((1 - 2x)u_x + (1 - 2y)u_y + (1 - 2z)u_z) = F$ con la misma solución que *P3* y $\kappa(A) \approx 77$

SIDERCA

Se basan en modelos matemáticos de la fabricación de tubos de acero sin costura por medio del proceso de Mannesmann. Para generar estos modelos se utilizan métodos de elementos finitos como se explica a continuación.

En cada etapa del proceso de fabricación se obtiene una malla de elementos finitos. Luego se obtienen las ecuaciones de métodos finitos mediante el método de Galerkin y en cada punto de la malla se resuelve una sucesión de sistemas lineales para obtener la

minimización. Finalmente, el estado de cada punto de la malla permite modelar el proceso.
Tenemos 4 de estos sistemas, que corresponden a 4 variantes del mismo problema, en las cuales se ha modificado un factor de penalización con el fin de imponer las condiciones de incompresibilidad del material. Esta penalización provoca una variación en el condicionamiento del sistema. Los problemas fueron llamados base17pk donde k es un factor de penalización, de forma que $\kappa(A) \approx 10^k$.

En el siguiente cuadro resumimos las características de los sistemas de prueba que utilizamos.

Nombre	n	nz	$\tilde{\kappa}(A)$	Simétrico	Almacenamiento	Endianess
base17p1	13345	474966	10^1	sí	CSC	big
base17p3	13345	474966	10^3	sí	CSC	big
base17p5	13345	474966	10^5	sí	CSC	big
base17p7	13345	474966	10^7	sí	CSC	big
B&S *P1*	13824	93312	4	no	CSC	little
B&S *P2*	13824	93312	57	no	CSC	little
B&S *P3*	13824	93312	40000	no	CSC	little
B&S *P4*	13824	93312	4786	no	CSC	little
B&S *P5*	13824	93312	36	no	CSC	little
B&S *P6*	13824	93312	77	no	CSC	little

Siendo n la dimensión de la matriz ($A \in \Re^{n \times n}$), nz la cantidad de entradas no nulas de la matriz y $\tilde{\kappa}(A)$ el número de condición aproximado del sistema.

Algoritmos

Algoritmo AcCim

1. Normalizar el sistema
2. $k = 0$
3. $r^0 = b - Ax^0$
4. $r_0 = 0$
5. $d^0 = 0$
6. *for i = 1...m* (en paralelo en la version AcCimMT)
   - 6.1. $d^0 = d^0 + r_i^0 A_i$
   - 6.2. $r_0 = r_0 + \left(r_i^0\right)^2$
7. $\lambda = r_0 \Big/ \left\| d^0 \right\|_2^2$
8. $x^1 = x^0 + \lambda d^0$
9. $rv = b - Ax^1$
10. $r^1 = \left\| rv \right\|_2^2$
11. *while* $\left(\sqrt{r^{k+1}} > \varepsilon \times \max\left(1, \sqrt{r_0}\right)\right)$ *&* $\left(k < \max iter - 1\right)$
    - 11.1. $k = k + 1$
    - 11.2. $r = 0$
    - 11.3. $\lambda^k = r$
    - 11.4. $d^k = 0$
    - 11.5. *for i = 1...m* (en paralelo en la version AcCimMT)
      - 11.5.1. $d^k = d^k + rv_i A_i$
      - 11.5.2. $r = r + \left(r_i^{k-1}\right)^2$
    - 11.6. $d^k = d^k - \dfrac{\left(d^{k-1^t} \times d^k\right)}{\left\| d^{k-1} \right\|_2^2} d^{k-1}$
    - 11.7. $\lambda^{k+1} = \lambda^k \Big/ \left\| d^k \right\|_2^2$
    - 11.8. $x^{k+1} = x^k + \lambda^{k+1} d^k$
    - 11.9. $rv = b - Ax^{k+1}$
    - 11.10. $r^{k+1} = \left\| rv \right\|_2^2$
12. *return* x^k

Algoritmo ALG1

El siguiente es el pseudo-código del método acelerado que incluye descomposición en bloques y la combinación óptima de las proyecciones.

1. Descomponer la matriz en bloques por filas
2. q = cantidad de bloques
3. $k = 0$
4. $r^0 = b - Ax^0$
5. $r = r_0 = \left\| r^0 \right\|$
6. $Q_{-1} = I$
7. *while* $(r > \varepsilon \times \max(1, r_0))$ *&* $(k < \max iter)$
    - 7.1. *for i = 1...q*
        - 7.1.1. $d_i^k = A_i^{\,t}\left(A_i A_i^{\,t}\right)^{-1}\left(b_i - A_i x^k\right)$
    - 7.2. $D^k = \left[d_1^k, \ldots, d_{q_k}^k\right]$
    - 7.3. $w^k = \left(D^{k\,t} D^k\right)^{-1}\left(\left\|d_1^k\right\|^2, \ldots, \left\|d_{q_k}^k\right\|^2\right)^t$
    - 7.4. $x^{k+1} = x^k + D^k w^k$
    - 7.5. $r^{k+1} = b - Ax^{k+1}$
    - 7.6. $r = \left\| r^{k+1} \right\|$
    - 7.7. $k = k + 1$
8. *return* x^k

Algoritmo ALG2

El siguiente pseudo-código corresponde al método acelerado que, además de las técnicas introducidas en *ALG1*, agrega la proyección sobre el hiperplano óptimo.

1. Descomponer la matriz en bloques por filas
2. q = cantidad de bloques
3. $k = 0$
4. $r^0 = b - Ax^0$
5. $r = r_0 = \left\| r^0 \right\|$
6. $Q_{-1} = I$
7. *while* $\left(r > \varepsilon \times \max(1, r_0)\right)$ *&* $\left(k < \max iter\right)$
    - 7.1. *for i = 1...q*
        - 7.1.1. $d_i^k = A_i^t \left(A_i A_i^t\right)^{-1} \left(b_i - A_i x^k\right)$
        - 7.1.2. $\tilde{d}_i^k = Q_{k-1}\left(d_i^k\right)$
    - 7.2. $\tilde{D}^k = \left[\tilde{d}_1^k, \ldots, \tilde{d}_{q_k}^k\right]$
    - 7.3. $w^k = \left(\tilde{D}^{k\,t} \tilde{D}^k\right)^{-1} \left(\left\|d_1^k\right\|^2, \ldots, \left\|d_{q_k}^k\right\|^2\right)^t$
    - 7.4. $\tilde{d}^k = \tilde{D}^k w^k$
    - 7.5. $x^{k+1} = x^k + \tilde{d}^k$
    - 7.6. $v = \tilde{d}^k$
    - 7.7. $r^{k+1} = b - Ax^{k+1}$
    - 7.8. $r = \left\| r^{k+1} \right\|$
    - 7.9. $k = k + 1$
8. *return* x^k

Donde

$$Q_{k-1}(x) = P_{v^\perp}(x) = x - \alpha v$$

$$\alpha = \frac{v^t x}{\|v\|^2}$$

Algoritmo de descomposición en bloques

1. $i = 1$
2. $I_S = \phi$
3. *while* $\mathrm{I}_S \neq \mathrm{I}$
   - 3.1. $I_C = I - I_S$, $j = 1$, $I_i = \phi$
   - 3.2. *elegir* $l \in I_C$
   - 3.3. $A_i = [a_l]$, $D_1 = [1]$, $L_i^{-1} = [1]$
   - 3.4. $I_C = I_C - \{l\}$, $I_S = I_S \cup \{l\}$, $I_i = I_i \cup \{l\}$
   - 3.5. *while* $(j < \mu)$ & $(I_C \neq \phi)$
     - 3.5.1. *elegir* $l \in I_C$
     - 3.5.2. $l_{j+1} = D_j^{-1} L_j^{-1} A_i a_l$
     - 3.5.3. $\delta_{j+1} = 1 - l_{j+1}^t D_j l_{j+1}$
     - 3.5.4. *if* $1/\delta_{j+1} < \kappa$
       - 3.5.4.1. $A_i = \begin{bmatrix} A_i \\ a_l \end{bmatrix}$
       - 3.5.4.2. $L_{j+1}^{-1} = \begin{bmatrix} L_j^{-1} & 0 \\ -l_{j+1}^t L_j^{-1} & 1 \end{bmatrix}$
       - 3.5.4.3. $D_{j+1}^{-1} = \begin{bmatrix} D_j^{-1} & 0 \\ 0 & 1/\delta_{j+1} \end{bmatrix}$
       - 3.5.4.4. $I_S = I_S \cup \{l\}$, $I_i = I_i \cup \{l\}$
       - 3.5.4.5. $j = j + 1$
     - 3.5.5. $I_C = I_C - \{l\}$
   - 3.6. $i = i + 1$
4. *return* L_j^{-1}, D_j^{-1} $\forall$ $1 \leq j \leq q$

Dónde:

$I = \{1, \ldots, m\}$

$I_S = \{l :$ la fila a_l de A está asignada a algún bloque $\}$

q es la cantidad de bloques formados

Reproducción de pruebas

Ejecutables disponibles en el CD

En el directorio `/src/` del CD se encuentra un archivo `Makefile` que permite generar, en Linux, los diferentes binarios. En dicho directorio también se pueden encontrar los archivos de proyecto para Visual C++ 6.0. En `/data/` están los sistemas de prueba usados a lo largo de este trabajo. Y finalmente, en el directorio `/bin/` se encuentran los ejecutables en sus versiones para Windows.

Nombre ejecutable/ target makefile	Observaciones
`accim-d-plain`	*AcCim* para sistemas densos
`accim-d-mkl`	*AcCim* para sistemas densos, usando MKL
`accimmt-d-plain`	*AcCim* para sistemas densos, con threads explícitos
`accimmt-d-mkl`	*AcCim* para sistemas densos, con threads explícitos, usando MKL
`accimsp-d-plain`	*AcCim* para sistemas ralos
`accimsp-d-mkl`	*AcCim* para sistemas ralos, usando MKL
`alg1-d-plain`	*ALG1* para sistemas ralos
`alg2-d-plain`	*ALG2* para sistemas ralos
`checkgeometry`	Estudia la geometría del sistema
`condestimator`	Estima el número de condición del sistema usando MKL
`svd`	Calcula los valores singulares de la matriz usando MKL

Todos los binarios pueden ser ejecutados sin parámetros para obtener ayuda sobre los mismos.
En los siguientes ejemplos, para probar con un sistema de B&S, se debe agregar el parámetro `–l` para indicar que el mismo está almacenado en formato little-endian.

AcCim

```
accim-d-plain –d10000 –s0 -v
```

AcCimMT

```
accimmt-d-plain –d10000 –s0 -v
```

AcCimSP

```
accimsp-d-plain base17p1.matrix -s -m -xn0 -e0 -ee0 -a9999999999 -i3000 -v5
accimsp-d-plain base17p1.matrix -s -m -xn0 -e0 -ee0 -a9999999999 -i13824
      -v5
accimsp-d-plain sameh1-csc.matrix -u –l -m -xn0 -e0 -ee0
      -a9999999999 –i13824 -v5
```

Utilización de precondicionadores

Balanceo

```
accimsp-d-plain base17p1-balanced.matrix -u –m
      -bbase17p3-balanced.matrix.col -i3000 -e0.0000000001 -ee0.0000000001
      -a3000 -v5
```

Diagonal

```
accimsp-d-plain base17p1.matrix -s -m -r -i3000
      -e0.0000000001 -ee0.0000000001 -a3000 -v5
```

Referencias

[1] H. D. Scolnik, N. Echebest, M. T. Guardarucci, M. C. Vacchino; New Optimized and Accelerated PAM Methods for solving large Non-symmetric Linear Systems: Theory and Practice, in: *Inherently Parallel Algorithms in Feasibility and Optimization and their Applications*, Studies in Computational Mathematics **8**, The Netherlands, pp. 457-470, 2001.

[2] H. D. Scolnik, N. Echebest, M. T. Guardarucci, M. C. Vacchino; A class of optimized row projection methods for solving large non-symmetric linear systems, *Applied Numerical Mathematics* **41** (Issue 4), pp. 499-513, 2002.

[3] H. D. Scolnik, N. Echebest, M. T. Guardarucci, M. C. Vacchino; A class of optimized row projection methods for solving large non-symmetric linear systems, Technical Report, Notas de Matemática 74, Departamento de Matemática, Universidad Nacional de La Plata, Buenos Aires, Argentina, 2000.

[4] H. D. Scolnik, N. Echebest, M. T. Guardarucci, M. C. Vacchino; Acceleration scheme for Parallel Projected Aggregation Methods for solving large linear systems, *Annals of Operations Research* 117 (Issue 1-4), pp. 95-115, 2002.

[5] N. Echebest, M. T. Guardarucci, H. D. Scolnik, M. C. Vacchino; *An acceleration scheme for solving convex feasibility problems using incomplete projection algorithms*, Numerical Algorithms 35, pp. 331-350, 2004.

[6] N. Echebest, M.T. Guardarucci, H. D. Scolnik, M. C. Vacchino; An Accelerated Iterative Method with Diagonally Scaled Oblique Projections for Solving Convex Feasibility Problems, Annals of Operations Research 138, Number 1, pp. 235-257, September 2005.

[7] H. D. Scolnik, N. Echebest, M. T. Guardarucci, M. C. Vacchino; Incomplete Oblique Projections Algorithms for Solving Large Inconsistent Linear Systems, Mathematical Programming, Ser. B, 111, pp. 273-300, 2008.

[8] G. Cimmino; Calcolo approssimato per le soluzioni dei sistemi di equazioni lineari; Ricerca Sci. II.9, pp. 326-333, 1938.

[9] S. Kaczmarz; Angenäherte Auflösung von Systemen linearer Gleichungen; Bull. Internet. Acad. Polonaise Sci. Lett. 35, pp. 335-357, 1937.

[10] M. R. Hestenes, E. Stiefel; Methods of Conjugate Gradients for Solving Linear Systems; *Journal of Research of the National Bureau of Standards* **49** (6), December 1952.

[11] U. M. García-Palomares; Parallel projected aggregation methods for solving the convex feasibility problem; SIAM J. Optim. **3**, pp. 882-900, 1993.

[12] R. Aharoni, Y. Censor; Block-iterative projection methods for parallel computation of solutions to convex feasibility problems; Linear Algebra Appl. 120, pp. 167-175, 1989.

[13] POSIX Threads
http://en.wikipedia.org/wiki/POSIX_Threads
POSIX Threads Programming
https://computing.llnl.gov/tutorials/pthreads/
Open Source POSIX Threads for Win32
http://sources.redhat.com/pthreads-win32/index.html

[14] R. Bramley, A. Sameh; Row projection methods for large nonsymmetric linear systems; SIAM J. Sci. Statist. Comput. 13, pp. 168-193, 1992.

[15] M. E. Canga, E. B. Becker; An iterative technique for the finite element analysis of near-incompressible materials; Comput. Meth. Appl. Mech. Eng. 170, pp. 79-101, 1999.

[16] Y. Saad, M. H. Schultz; Conjugate gradient-like algorithms for solving nonsymmetric linear systems, Math. Comp. 44, pp. 417-424, 1985.

[17] H. van der Vorst; A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems, SIAM J. Sci, Statisct. Comput 13, pp. 631-644, 1992.

[18] Y. Saad; SPARSKIT: A basic tool for sparse matrix computations, Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA, 1990.

[19] H. D. Scolnik, N. Echebest, M. T. Guardarucci, M. C. Vacchino; An Incomplete Projections Algorithm for Solving Large Inconsistent Linear Systems, Septiembre 2003.

[20] H. D. Scolnik, N. Echebest, M. T. Guardarucci, M. C. Vacchino; Incomplete Oblique Projections Algorithms for Solving Large Inconsistent Linear Systems, Enero 2004.

[21] P. E. Hill, W. Murray, M. H. Wright; Practical Optimization, Academic Press, pp. 353, 1981.

[22] Intel Math Kernel Library
http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm

[23] HP's Mathematical Software Library (MLIB)
http://h21007.www2.hp.com/portal/site/dspp/PAGE.template/page.document?ciid=c008a8ea6ce02110a8ea6ce02110275d6e10RCRD&jumpid=reg_R1002_USEN

[24] MGMRES, http://people.scs.fsu.edu/~burkardt/cpp_src/mgmres/mgmres.html