U.N.L.P - FACULTAD DE INFORMÁTICA
TRABAJO DE GRADO

Desarrollo de una implementación óptima de un algoritmo acelerado de proyección en bloques

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

TES
99/2
DIF-02066
SALA

UNIVERSIDAD NACIONAL DE LA PLATA
FACULTAD DE INFORMATICA
Biblioteca
50 y 120 La Plata
catalogo.info.unlp.edu.ar
biblioteca@info.unlp.edu.ar

DIF-02066

Baragatti, Hernán

2b

Bianchi, José

Director: Lic. María Teresa Guardarucci

DONACION..................................... TES
$.. 99/2 y.1
Fecha 29-09-05
Inv. E..............Inv. B. 2066

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

" La maravillosa riqueza de la experiencia humana perdería parte de su alegría gratificante si no existieran limitaciones que superar.
La cima de la colina no sería ni la mitad de maravillosa si no hubiera oscuros valles que atravesar.
Vale la pena... esperar lo que realmente se desea, vivir intensamente, disfrutar de cada momento, tener ilusiones, apostar al amor. Vale la pena pensar que lo mejor está por venir... "

Leo Buscaglia

A mis Grandes Maestros, mis padres Jorge y Verónica.
A mis hermanas Paola, Julieta y Analía
A Lore
José Bianchi

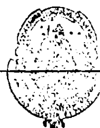
A mis padres Eduardo y Mónica.
A mis hermanos.
A Jorgelina
Hernán Baragatti

AGRADECIMIENTOS

Queremos agradecer muy especialmente a Marité, por ayudarnos a hacer realidad nuestro tantas veces soñado Trabajo de Grado.

A Nelson Maculan, por otorgarnos un usuario en el equipo Cray J90 PVP localizado en la Universidad Federal de Río de Janeiro, sin el cual no podríamos haber realizado los experimentos computacionales de ejecución y testeo de nuestro Trabajo.

A Paola Bianchi, por su colaboración en el diseño gráfico de la presentación.


BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Índice

Introducción

Introducción 1

Capítulo I
Modelos

I.1 El modelo discretizado para la reconstrucción de imágenes por Tomografía computarizada 3
I.2 La resolución de ecuaciones diferenciales por diferencias finitas 6

Capítulo II
Algoritmos

II.1 Clasificación de los algoritmos de proyección 9
II.2 Proceso de división en bloques bien condicionados usando un estimador del número de condición 13
Idea de proyección sobre un bloque 13
Número de condición de una matriz 14
Formación de los bloques 15
II.3 Algoritmo de proyección en bloques acelerado 19
Criterios de parada del algoritmo 23
Convergencia 24

Capítulo III
Desarrollo

III.1 Desarrollo 27
III.1.1 Especificación del problema 27
III.1.2 Diseño 29
III.1.2.1 Estructuras de datos principales. Descripción 31
i. Caso A matriz densa 32
ii. Caso A matriz esparsa 34
III.2 Implementación 37
Elección del lenguaje de programación 39
Características técnicas de Cray J90 PVP 41
III.2.1 Proceso de optimización 41
III.2.1.1 Vectorización 43
Procesamiento escalar vs. procesamiento vectorial 43
Tipos de vectorización 45
Inhibidores de vectorización 46

Resolviendo la dependencia de datos 51
Técnicas de vectorización 53
Cómo evitar dependencia de datos 54
III.2.1.2 Procesamiento paralelo 54
Costos y beneficios del procesamiento paralelo 56
Directivas de procesamiento paralelo 59
Distribución del trabajo 60
Aplicación de los conceptos de optimización a nuestro código 64
III.3 Verificación 69

Capítulo IV
Resultados

IV.1 Problemas de testeo 72
IV.1.1 Casos esparsos 72
IV.1.2 Caso denso 73
IV.2 Resultados numéricos 73
IV.2.1 Resultados numéricos para P1-P6 74
IV.2.2 Resultados numéricos para P7 80
IV.3 Análisis comparativo de estructuras 81
IV.4 Conclusiones 82

Anexo I

Diseño. Refinamiento procedural Top Down 84

Anexo II

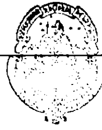
AII.1 Prueba numérica de la implementación del algoritmo de división en bloques 88
AII.2 Solución del algoritmo acelerado de proyección en bloques 93

Glosario

Glosario 94

Bibliografía

Bibliografía 98


BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

INTRODUCCIÓN

En diversas áreas de las ciencias exactas, naturales, sociales e ingeniería encontramos problemas de optimización matemática. Los desarrollos que este vasto campo provee son motivados, en gran medida, por problemas del mundo real, que a través de los años han sido tratados tanto por la matemática como por las ciencias de la computación. La matemática crea el cimiento para el análisis y diseño de algoritmos de optimización. Las ciencias de la computación proveen las herramientas para el diseño de estructuras de datos y para trasladar los algoritmos matemáticos en procedimientos que luego son implementados para que finalmente puedan ejecutarse en una computadora. La eficiencia en la implementación de algoritmos de optimización es muy importante cuando se quieren emplear para resolver problemas reales de gran escala.

El advenimiento de la computación paralela junto con las innovaciones tecnológicas producidas en este campo, han beneficiado la manera de atacar este tipo de problemas. Así, tenemos métodos de optimización paralelos, que introducen ideas y técnicas de la computación paralela en la teoría y en los algoritmos numéricos de optimización.

A menudo la implementación de un algoritmo cambia la perspectiva que uno tiene del mismo. Es con los experimentos computacionales de testeo cuando uno puede tener realmente confianza en la eficiencia y robustez de un algoritmo de optimización matemático de gran envergadura.

Una de las cosas que más nos motivó a emprender el desafío de este Trabajo de Grado es la posibilidad de que el mismo realice un aporte para poder resolver en forma más eficiente cada uno de los importantes problemas que se encuentran subyacentes en el modelo matemático a optimizar: reconstrucción de imágenes por proyecciones, aplicaciones médicas como el planeamiento de la terapia de radiación, programación no lineal para el planeamiento bajo incertidumbre, balanceo de matrices, optimización de redes, planeamiento financiero, etc.

El grupo de optimización que trabaja en el Departamento de Matemática de la Facultad de Ciencias Exactas de la Universidad Nacional de La Plata bajo la dirección del Dr. Hugo Scolnik, Prof. Titular del Departamento de Computación de la Universidad de Buenos Aires, y del cual la Lic. Guardarucci es integrante, ha desarrollado un algoritmo para resolución de sistemas de ecuaciones lineales no simétricos altamente paralelizable. Un convenio con COPPE, Universidad Federal de Río de Janeiro, nos permitió tener acceso al equipo CRAY J90 PVP (*Parallel Vector Processing*) para testear y comparar los resultados del proceso de optimización del algoritmo durante su implementación, y poder así extraer conclusiones acerca del mismo.

En el Capítulo I analizamos el modelo matemático de los problemas reales que sirvieron como motivación práctica para este trabajo: la reconstrucción de imágenes por proyecciones en tomografía computarizada, y la resolución de ecuaciones diferenciales por diferencias finitas, observando cómo es necesario resolver sistemas de ecuaciones lineales de grandes dimensiones y cuán importante es la optimización para la resolución de los mismos.

El Capítulo II está dedicado a mostrar una visión general de los distintos métodos de proyección para resolver sistemas de ecuaciones, y a explicar con profundidad el algoritmo elegido a implementar.

En el Capítulo III mostramos el desarrollo completo de la implementación del algoritmo: especificación, diseño, justificación de la elección y análisis de las estructuras de datos adoptadas, elección de la plataforma de ejecución y lenguaje de implementación, así como los pasos del proceso de optimización que nos llevaron a la implementación paralela del mismo.

Finalmente, en el Capítulo IV presentamos los resultados numéricos que reflejan el proceso de optimización y extraemos conclusiones.

Los autores*

* José Bianchi y Hernán Baragatti son Analistas de Computación de la UNLP. El presente es su Trabajo de Grado para la Licenciatura en Informática. Comentarios o sugerencias: boxbianchi@ciudad.com.ar

CAPÍTULO I
Modelos

Muchas aplicaciones reales en diferentes campos de investigación son problemas de inversión. Un objeto x está relacionado con algún dato t a través de la relación $t = O(x)$, donde la obtención de x requiere, esencialmente, la inversión del operador O. En la primera parte de este capítulo vamos a estudiar un caso particular de problemas de inversión: la reconstrucción de imágenes por tomografía computarizada. Luego veremos un ejemplo de cálculo de ecuaciones diferenciales por diferencias finitas, de aplicación en diferentes campos en ingeniería.

Un camino comúnmente empleado para atacar problemas de inversión es formular un modelo continuo y usar herramientas matemáticas para la inversión analítica del operador O. Si se tiene éxito, la fórmula final es luego "discretizada" para su implementación final.

La discretización (total o parcial) del modelo en fases tempranas del proceso de modelización a menudo llevan al análisis convexo (o álgebra lineal dimensional) y, consecuentemente, a la formulación de teorías de optimización. El gran tamaño acompañado con lo esparso que estos sistemas pueden resultar nos hace recurrir generalmente a técnicas iterativas. Una clasificación de estos algoritmos es presentada en el Capítulo II. De acuerdo a esa clasificación los algoritmos iterativos de proyección por bloques pueden procesar grupos de restricciones (bloques) secuencialmente o en paralelo.

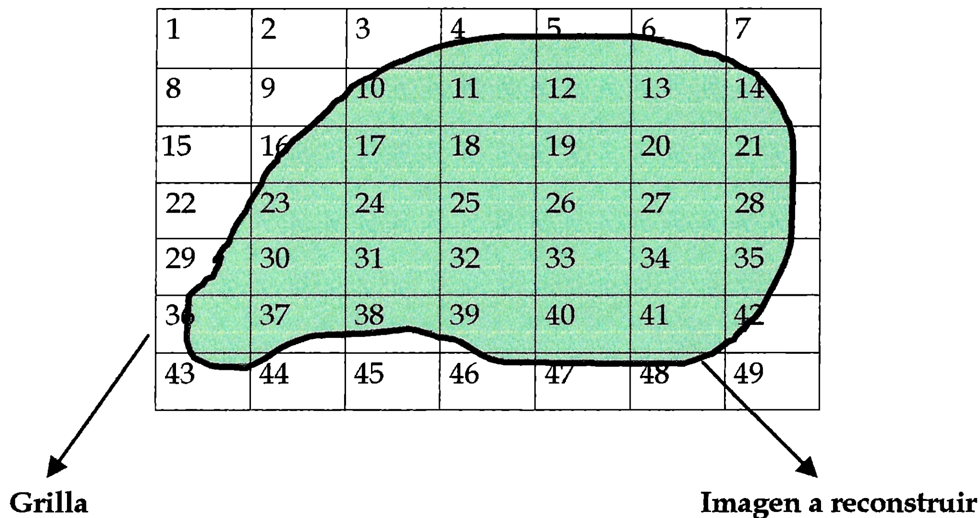
I.1 El modelo discretizado para la reconstrucción de imágenes por tomografía computarizada

Los problemas de reconstrucción de imágenes difieren significativamente dependiendo de su área de aplicación. Incluso dentro de un mismo campo -como la medicina o la industria- existen diferentes formas de reconstruir debido a los diferentes métodos físicos de recolección de datos. A pesar de esas diferencias existe una naturaleza matemática común en estos problemas: hay una distribución desconocida (en dos o tres dimensiones) de un parámetro físico. Este parámetro puede ser, por ejemplo, la atenuación lineal de los rayos X en el tejido humano, la atenuación de los coeficientes de materia en un reactor nuclear, o la densidad de electrones en la corona del sol. Un número finito de integrales de línea de este parámetro pueden ser estimadas de acuerdo a medidas físicas, y lo que buscamos finalmente es una estimación de la distribución original del parámetro.

En la aplicación médica de reconstrucción de imágenes por tomografía computarizada (TCT), se considera como objeto de estudio una sección transversal plana del cuerpo humano, y el objetivo es reconstruir la imagen dada por la atenuación de los rayos X en cualquier lugar de la sección.

El modelo de aproximación por series finitas se define así: se considera que una grilla cartesiana o cuadrícula de pixels cubre completamente la región de interés a reconstruir.

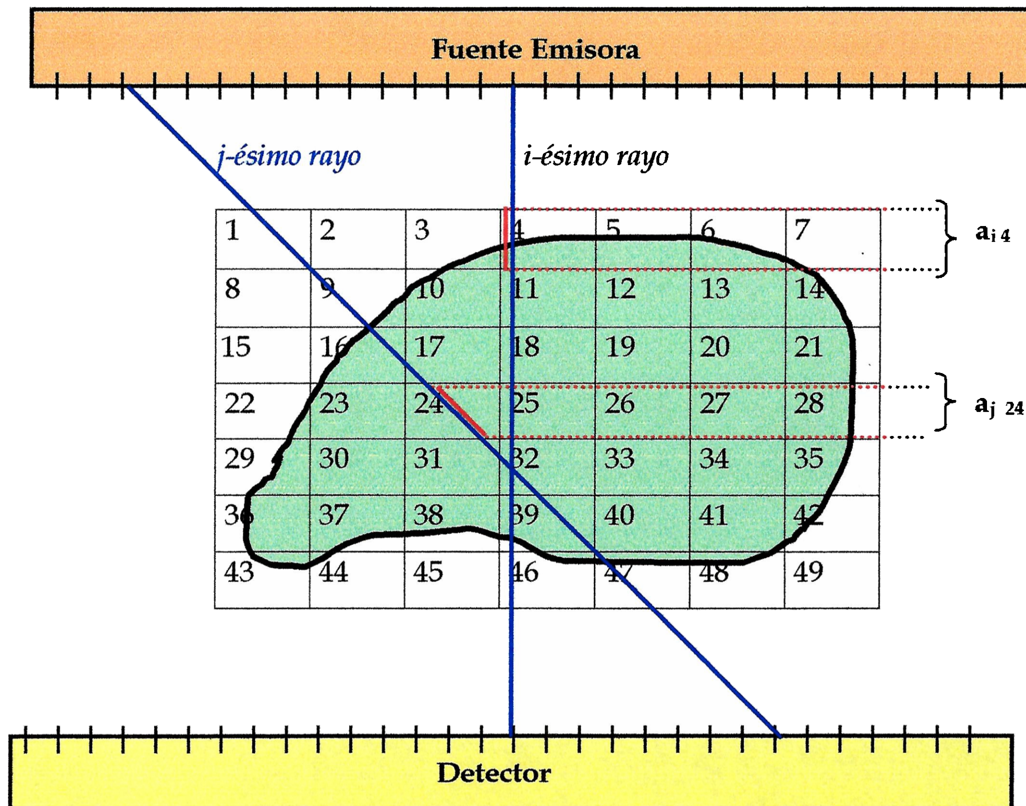
Los pixels son numerados desde 1 (arriba a la izquierda), hasta J (abajo a la derecha).



Asumimos que la función de atenuación de rayos X toma el valor constante uniforme x_j a través del j-ésimo pixel para j=1,2,......,J (es decir, el rayo se atenúa o degrada en forma constante a lo largo de cada pixel atravesado).

Las fuentes emisoras de rayos X y los detectores son puntos, y los rayos entre ellos líneas.

La **longitud de la intersección** del i-ésimo rayo con el j-ésimo pixel, llamado a_{ij} para todo i=1,2,......,I, j=1,2,......,J representa el peso de contribución del pixel j al total de atenuación a lo largo del rayo i.



La medida física de la atenuación total a lo largo del rayo i, llamado y_i, queda representada por la integral de línea de la función de atenuación desconocida a lo largo del camino del

rayo. En este modelo discretizado, cada integral de línea es una suma finita y el modelo queda descripto por el sistema de ecuaciones lineales:

$$\sum_{j=1}^{n} x_j . a_{ij} = y_i$$

i=1,2,......,I

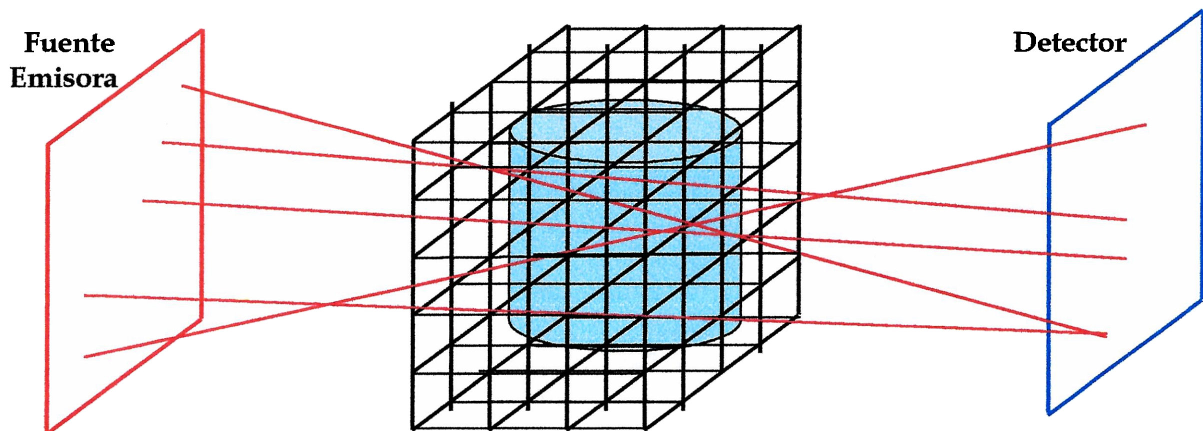
En notación matricial, $y = A^t.x$, donde

$y=(y_i) \in R^I$ es el **vector de datos medidos por el detector**

$x=(x_i) \in R^J$ es el **vector imagen**, y $\mathbf{A^t=(a_{ij})}$ de dimensiones *I x J* es la matriz de **longitudes de intersección del i-ésimo rayo con el j-ésimo pixel**. Si en su recorrido el rayo ***i*** "no toca" al pixel ***j*** entonces $a_{ij}=0$.

Observaciones importantes:

- Como se desprende del modelo, tanto el *vector **y*** como la *matriz **A*** son datos. El primero es el valor de radiación de cada rayo detectado luego de atravesar la sección a reconstruir, mientras que el segundo es la longitud de la intersección de cada rayo con cada pixel. La incógnita es el vector imagen **x**, que denota cuánto se atenúa cada rayo X al pasar por cada pixel. Definiendo *intensidades de color* diferentes para cada valor x_i tendremos una idea de la imagen que queríamos reconstruir inicialmente.
- Para tener **mayor precisión** de la imagen a reconstruir, la cuadrícula debe ser muy pequeña, y la cantidad de rayos que la atraviesan muy grande. Es decir, tanto J como I deben ser números naturales grandes, y, **en consecuencia, el sistema a resolver toma dimensiones "gigantescas"** (la matriz A^t tiene tantas filas como rayos atraviesan la región a reconstruir y tantas columnas como pixels definan la región de estudio).
- **Se trata de un "sistema esparso"**, ya que en su recorrido, cada rayo atraviesa "pocos pixels", y ***J*** es un número muy grande (veamos el gráfico de la página 4, en el que el rayo ***i*** atraviesa sólo 7 pixels de un total de 49).
- El **tiempo de ejecución de la implementación de un algoritmo** que resuelva sistemas de ecuaciones para reconstruir imágenes, como en este caso, es *sumamente importante*, ya que si un equipo médico está realizando una tomografía de urgencia a un paciente para determinar los pasos a seguir, el tiempo de respuesta (para conocer la imagen a estudiar) en estos casos es un *factor crítico.*
- Extensión del modelo: hemos descripto el modelo en dos dimensiones, pero claramente este modelo se puede extender a tres dimensiones (podemos pensar que para reconstruir una imagen en tres dimensiones, dividimos el espacio en una "cubícula" numerada; luego a_{ij} es la longitud de la intersección del rayo *i* con el cubo *j* e y_i es el valor del rayo *i* medido por el detector).



Esquema de extensión del modelo

I.2 La resolución de ecuaciones diferenciales por diferencias finitas

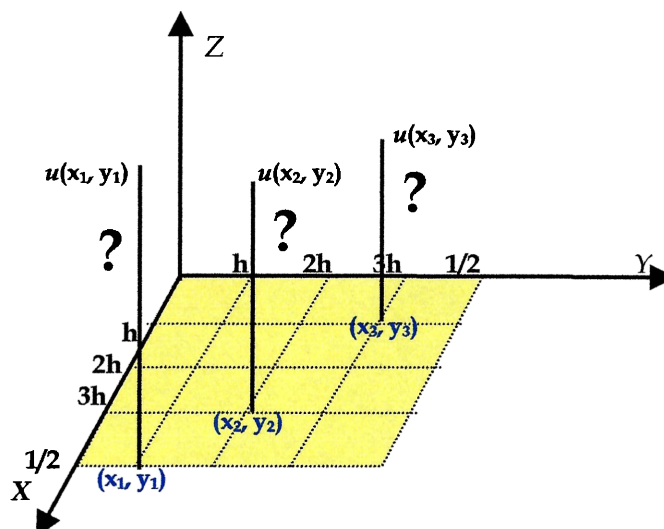
El objetivo en este caso es la reconstrucción de funciones. Para comprender el método, comenzaremos estudiando funciones de dos variables a través de un ejemplo, y luego el mismo método se extiende para la reconstrucción de funciones de más de dos variables.

Supongamos que queremos conocer la función $u(x,y)$, que sabemos está definida en el dominio [0,1/2]x[0,1/2].

Vamos a hallar en forma aproximada los valores que toma $u(x_i, y_j)$, donde (x_i, y_j) son las coordenadas de algún punto del dominio de u.

Para ello subdividimos el dominio en una cuadrícula tomando espacios de tamaño h en ambos ejes.

h=(1/8)



¿ Qué datos conocemos de ***u*** ?

Generalmente se conoce alguna propiedad física de sus derivadas parciales, como por ejemplo la siguiente, que es la fórmula de distribución de calor a lo largo de una placa:

$$\partial^2 u / \partial^2 x + \partial^2 u / \partial^2 y = 0 \qquad (1.1)$$

Esta información se suma a las condiciones de borde (cómo se comporta la función "en los bordes" de su dominio):

$u(x,0)=0$
$u(0,y)=0$
$u(x,1/2)=200x$
$u(1/2,y)=200y$

Para poder reconstruir la función realizamos el desarrollo aproximado de las derivadas parciales en torno a los puntos (x_i, y_j) del dominio:

$$\partial u(x_i, y_j)/\partial x \approx [u(x_i+h, y_j) - u(x_i, y_j)]/h \approx [u(x_i, y_j) - u(x_i - h, y_j)]/h$$,entonces

cuando h→0 **cuando h→0**

$$\partial^2 u(x_i, y_j)/\partial^2 x \approx [\partial u(x_i+h, y_j)/\partial x - \partial u(x_i, y_j)/\partial x]/h$$
$$\approx \mathbf{[u(x_i+h, y_j) - 2u(x_i, y_j) + u(x_i-h, y_j)]/h^2}$$

$$\partial u(x_i, y_j)/\partial y \approx [u(x_i, y_j+h) - u(x_i, y_j)]/h \approx [u(x_i, y_j) - u(x_i, y_j-h)]/h$$,entonces

cuando h→0 **cuando h→0**

$$\partial^2 u(x_i, y_j)/\partial^2 y \approx (\partial u(x_i, y_j+h)/\partial y - \partial u(x_i, y_j)/\partial y)/h$$
$$\approx \mathbf{[u(x_i, y_j+h) - 2u(x_i, y_j) + u(x_i, y_j-h)]/h^2}$$

Desconocemos los valores de

- $\mathbf{u(x_i, y_j)}$
- $\mathbf{u(x_i+h, y_j)}$
- $\mathbf{u(x_i-h, y_j)}$
- $\mathbf{u(x_i, y_j+h)}$
- $\mathbf{u(x_i, y_j-h)}$

Pero sabemos que deben cumplir (1.1), con lo cual

$$\mathbf{[u(x_i+h, y_j) - 2u(x_i, y_j) + u(x_i-h, y_j)]/h^2 + [u(x_i, y_j+h) - 2u(x_i, y_j) + u(x_i, y_j-h)]/h^2 = 0}$$
$$\mathbf{u(x_i+h, y_j) - 2u(x_i, y_j) + u(x_i-h, y_j) + u(x_i, y_j+h) - 2u(x_i, y_j) + u(x_i, y_j-h) = 0}$$

$$\mathbf{u(x_i+h, y_j) - 4u(x_i, y_j) + u(x_i-h, y_j) + u(x_i, y_j+h) + u(x_i, y_j-h) = 0}$$

Si llamamos w_{ij} a $u(x_i , y_j)$ **CADA PUNTO DE LA GRILLA DEBE cumplir:**

$$-4\, w_{i\,j} + w_{i+1\,j} + w_{i-1\,j} + w_{i\,j+1} + w_{i\,j-1} = 0$$

Agregando las condiciones de borde:

$w_{i\,0} = 0$
$w_{0\,i} = 0$
$w_{i\,1/2} = 200.i$
$w_{1/2\,j} = 200.j$

Resulta el sistema de ecuaciones

$$\begin{cases} -4\, w_{i\,j} + w_{i+1\,j} + w_{i-1\,j} + w_{i\,j+1} + w_{i\,j-1} = 0 \\ w_{i\,0} = 0 \\ w_{0\,i} = 0 \\ w_{i\,1/2} = 200.i \\ w_{1/2\,j} = 200.j \end{cases}$$

Observaciones importantes:

- Para poder reconstruir de mejor manera la función, la cuadrícula de la **grilla** en la que se dividió el dominio debe ser **"pequeña"**.
- El sistema a resolver tiene tantas ecuaciones e incógnitas como puntos de intersección tiene la grilla, con lo cual resulta un **sistema de grandes dimensiones**.
- Es un "**sistema esparso**", ya que cada punto (x_i , y_j) "se relaciona" sólo con sus puntos adyacentes $(x_i +h, y_j)$, $(x_i -h, y_j)$, $(x_i , y_j +h)$ y $u(x_i , y_j -h)$.
- El modelo se **extiende** siguiendo la misma filosofía empleada: por ejemplo, si u(x,y,z) desconocida está definida en $\Re^3$, entonces su dominio se divide en una "cubícula", y se aplican los pasos del ejemplo, empleando de las derivadas parciales apropiadas.
- Existen numerosos problemas de la vida real que resuelven sistemas de ecuaciones de este tipo, y para los cuales el tiempo de resolución es un factor crítico. Por ejemplo, se emplean en ingeniería hidráulica para definir el movimiento de las aguas en un dique. Como las condiciones a las que se encuentra sometido el entorno del dique son dinámicas (cambios climatológicos, accidentes naturales o humanos, etc,), la resolución de estos sistemas a tiempo determina cuando abrir o cerrar las compuertas de la represa frente a las mismas.

CAPÍTULO II
Algoritmos

II.1 Clasificación de los algoritmos de proyección

Existen diferentes maneras de atacar los problemas de inversión involucrados en diferentes campos de aplicación. En nuestro caso, el principal interés está en la propuesta que discretiza totalmente el modelo físico al principio del proceso de modelización matemática. Esta propuesta induce inicialmente a un sistema de ecuaciones o inecuaciones lineales o no lineales, es decir un sistema de la forma:

$$f_i(x) * 0, \quad i = 1,2,\dots,I, \tag{2.1}$$

donde $x \in R^J$,R^J es el espacio Euclídeo J-dimensional, $f_i : R^J \to R$ y * pueden ser signos de igualdad o desigualdad. Se puede resolver el *problema de factibilidad* (2.1) cuando el sistema es *factible*, es decir, $\Im = \{x \in R^J | f_i(x) * 0, i = 1,2,\dots,I\} \neq \varnothing$.

Aun cuando el problema no es factible en algunas ocasiones se desea hallar una solución en R^J que sea "la mejor" en algún sentido.

El modelo matemático puede ser visto como un *problema de optimización* con una función objetivo $f_0 : R^J \to R$ impuesta, y (2.1) sirviendo de restricciones.

$$\begin{cases} \text{optimizar} \quad f_0(x) \\ \text{sujeto a} \quad f_i(x) * 0, \quad i = 1,2,\dots,I \end{cases} \tag{2.2}$$

con o sin restricciones adicionales de la forma $x \in Q$,donde $Q \subseteq R^J$es un subconjunto dado, usualmente llamado caja de restricciones.

Algoritmos iterativos de propósito especial diseñados para resolver cualquiera de este tipo de problemas pueden emplear iteraciones las cuales usan en cada paso la información de una única fila del sistema de restricciones (2.1) o un grupo de filas.

Por ejemplo podríamos avanzar desde un iterado x^k a otro iterado x^{k+1} , pidiendo que x^{k+1} satisfaga o se acerque a una ecuación predeterminada.

Una *iteración de acción por filas* tiene la forma funcional

$$x^{k+1} = R_{i(k)}(x^k, \{f_i\}_{i \in I_{i(k)}}) \tag{2.3}$$

donde $i(k)$ es el *índice de control,* $1 \le i(k) \le I$,que especifica la fila sobre la cual se está actuando cuando el operador algorítmico $R_{i(k)}$ genera, a partir del iterado actual x^k y la información contenida en $f_{i(k)}$, el siguiente iterado x^{k+1}. $R_{i(k)}$ puede incluir parámetros adicionales los cuales varían de iteración en iteración tales como parámetros de relajación, pesos, tolerancias, etc.

El sistema (2.1) puede ser descompuesto en M grupos de restricciones ("bloques") eligiendo enteros $\{m_t\}_{t=0}^{M}$ tales que

$$0 = m_0 < m_t < \dots < m_{M-1} < m_M = I, \tag{2.4}$$

y definiendo para cada t, $1 \le t \le M$, el subconjunto

$$I_t = \{m_{t-1} + 1, m_{t-1} + 2, \dots, m_t\} \tag{2.5}$$

produciendo la partición

$$\{1,2,\ldots,I\} = I_1 \cup I_2 \cup \cdots \cup I_M. \tag{2.6}$$

Podríamos entonces avanzar desde un iterado x^k a otro iterado x^{k+1} pidiendo que satisfaga o se acerque a todas las ecuaciones que conforman un bloque predeterminado.
Una *iteración de bloque* tiene la forma funcional

$$x^{k+1} = B_{t(k)}(x^k, \{f_i\}_{i\in I_{t(k)}}) \tag{2.7}$$

donde $t(k)$ es el índice de control, $1 \leq t(k) \leq M$, especifica el bloque que es utilizado cuando el operador algorítmico $B_{t(k)}$ genera x^{k+1} a partir de x^k y de la información contenida en todas las filas de (2.1) cuyos índices pertenecen a $I_{t(k)}$. Otra vez, parámetros adicionales pueden ser incluidos en cada $B_{t(k)}$. El algoritmo iterativo de propósito especial puede aplicarse a cualquier problema de la forma (2.1) o (2.2) y puede ser clasificado según tenga alguna de las cuatro estructuras básicas siguientes.

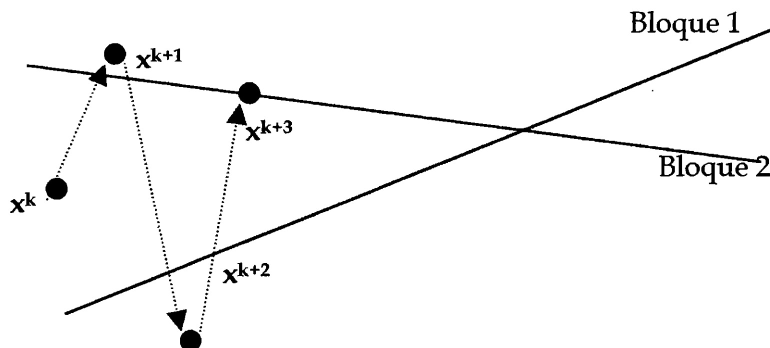
(i) *Algoritmo Secuencial.* Una secuencia de control $\{i(k)\}_{k=0}^{\infty}$ es definida y el algoritmo realiza, de una manera estrictamente secuencial, iteraciones de acción por filas de acuerdo a (2.3), desde un punto inicial apropiado hasta que se satisfaga una regla de parada.

(ii) *Algoritmo Paralelo.* Iteraciones de acción por filas son realizadas concurrentemente sobre todas las filas y el siguiente iterado x^{k+1} es entonces generado a partir de todos los iterados intermedios $x^{k+1,i}$,

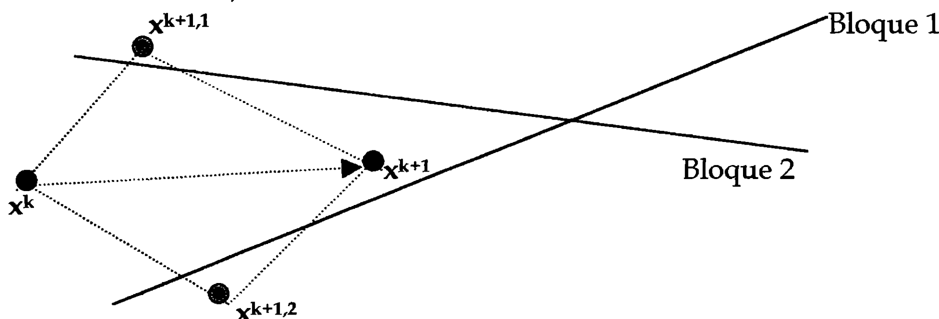
$$x^{k+1,i} = R_i(x^k, f_i), \quad i = 1,2,\ldots,I, \qquad \text{y luego}$$
$$x^{k+1} = S(\{x^{k+1,i}\}_{i=1}^{I}) \tag{2.8}$$

Aquí R_i y S son operadores algorítmicos, siendo los operadores R_i del tipo acción por filas.

(iii) *Algoritmo Secuencial Iterativo por Bloques.* El sistema (2.1) es descompuesto en bloques de acuerdo a (2.6) y se define la secuencia de control $\{t(k)\}_{k=0}^{\infty}$. El algoritmo realiza secuencialmente, de acuerdo a la secuencia de control, iteraciones por bloque de la forma (2.7).



(iv) *Algoritmo Paralelo Iterativo por Bloques.* Iteraciones por bloques son realizadas concurrentemente sobre todos los bloques y el iterado siguiente x^{k+1} se genera a partir de los iterados intermedios $x^{k+1,t}$,



$$x^{k+1,t} = B_t(x^k, \{f_i\}_{i\in I_t}), \quad t = 1,2,\ldots,I \qquad \text{y luego} \qquad (2.9)$$
$$x^{k+1} = S(\{x^{k+1,t}\}_{t=1}^{I})$$

Aquí S y B_t son operadores algorítmicos, siendo los operadores B_t del tipo iteración por bloques.

A continuación se incluye una tabla (Tabla 1) que resume la información referente a los distintos algoritmos, clasificados por tipo, que existen actualmente para los diferentes modelos matemáticos. Los algoritmos están identificados en esta tabla por sus nombres o por el apellido del autor. Notar que en la clasificación no se hace distinción entre algoritmos secuenciales o paralelos iterativos por bloque.

Tabla 1: Problemas y algoritmos

	Secuencial (acción por filas) S	Paralelo (simultáneo) P	Secuencial o paralelo iterativo por bloques B
1. $Ax = y$	"ART" (=Kaczmarz)	"SIRT" o "Cimmino"	"Bloques Kaczmarz"
2. $Ax \leq y$	"AMS"	Cimmino	Merzlyakov [54]
3. $c \leq Ax \leq b$	"ART3", "ARM"	?	Iusem & De Pierro
4. $\|\|x\|\|$, $Ax = y$	"ART" (=Kaczmarz)	"SIRT" o Cimmino	"Bloques Kaczmarz"
5. $\|\|x\|\|$, $Ax \leq y$	"Hildreth"	"Hildreth simultáneo"	?
6. $\|\|x\|\|$, $c \leq Ax \leq b$	"ART4"	?	?
7. ent x, $Ax = y$	"MART", "Bregman"	?	"Bloques Kaczmarz"
8. ent x, $Ax \leq y$	"MART", "Bregman"	?	?
9. ent x, $c \leq Ax \leq b$	"Bregman"	?	?
10. log x, $Ax = y$	"Bregman"	?	?
11. log x, $Ax \leq y$	"Bregman"	?	?
12. log x, $c \leq Ax \leq b$	?	?	?
13. CFP, $\leq$	"SOP" "CSP" "IP"		?
14. CFP, =			?

Los problemas matemáticos para los cuales se han diseñado los algoritmos de propósito especial que mencionamos en la tabla, están establecidos en la primer columna en forma abreviada y son los siguientes: 1, 2 y 3 son problemas de factibilidad lineal para igualdades, desigualdades e intervalos, respectivamente; 4, 5 y 6 son problemas de minimización de norma sujeto a restricciones como las mencionadas en 1, 2 y 3. La función objetivo ***ent x*** que aparece en 7, 8 y 9, mapea el cuadrante no negativo en R de acuerdo a :

$$\sum_{j=1}^{J} x_j . \log x_j$$

y, por definición, 0.log 0 = 0. La función objetivo *log x* que aparece en 10, 11 y 12 está definida en el octante positivo de R^J por $\log x = \sum_{j=1}^{J} \log x_j$ .Estas son dos entropías funcionales diferentes

las cuales necesitan ser maximizadas sobre las restricciones lineales escritas a su derecha. CFP viene de *Problema de Factibilidad Convexa* con las restricciones de igualdad o desigualdad $f_i : R^J \to R$ y donde $f_i(x) * 0, i = 1,2,\ldots,I$ son las funciones convexas.

Los signos de interrogación en la tabla indican que a la fecha no hay una versión conocida por nosotros. Debido a la variedad de formas en las cuales los operadores algorítmicos R_i, B_t y S pueden ser elegidos, es posible idear nuevos algoritmos para la columna B aún también para aquellas filas de la tabla para las cuales ya existen algoritmos iterativos por bloques de uno u otro tipo.

En las siguientes dos secciones de este Capítulo veremos en detalle el algoritmo que posteriormente implementaremos. El mismo es un "algoritmo acelerado de proyección en bloques" que resuelve sistemas de ecuaciones lineales no simétricos, y fue desarrollado por el grupo de optimización del Departamento de Matemática de la Facultad de Ciencias Exactas de la Universidad Nacional de La Plata bajo la dirección del Dr. Hugo Scolnik, Prof. Titular del Departamento de Computación de la Universidad de Buenos Aires. La sección II.2 muestra cómo dividir el sistema original en bloques "bien condicionados", y la sección II.3 el método para llegar a una solución.

II.2 Proceso de división en bloques bien condicionados usando un estimador del número de condición

Cuando al resolver (2.1) $f_i(x)$ es del tipo $a_i^t.x - b_i$ y * es el signo de igualdad, el problema a resolver se transforma en hallar una solución de $A^t.x=b$. Si la resolución se hace mediante un algoritmo iterativo usando iteraciones de bloque del tipo (2.7) se debe proceder de la siguiente manera:

1). Se divide la matriz A^t en bloques A_1^t , A_2^t , A_3^t ,......., A_q^t. En igual forma se dividen los términos independientes b.

$$A^t = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \ldots\ldots a_{1\,n-1} & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \ldots\ldots a_{2\,n-1} & a_{2n} \\ & & & \ldots\ldots & & \\ a_{m-1\,1} & a_{m-1\,2} & a_{m-1\,3} & a_{m-1\,4} & a_{m-1\,5} \ldots\ldots a_{m-1\,n-1} & a_{m-1\,n} \\ a_{m1} & a_{m2} & a_{m3} & a_{m4} & a_{m5} \ldots\ldots a_{m\,n-1} & a_{mn} \end{pmatrix} \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \\ x_{n-1} \\ x_n \end{pmatrix} \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \\ b_{m-1} \\ b_m \end{pmatrix}$$

$$A^t = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \ldots\ldots a_{1\,n-1} & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \ldots\ldots a_{2\,n-1} & a_{2n} \\ & & & \vdots & & \\ & & & \ldots\ldots & & \\ a_{m-1\,1} & a_{m-1\,2} & a_{m-1\,3} & a_{m-1\,4} & a_{m-1\,5} \ldots\ldots a_{m-1\,n-1} & a_{m-1\,n} \\ a_{m1} & a_{m2} & a_{m3} & a_{m4} & a_{m5} \ldots\ldots a_{m\,n-1} & a_{mn} \end{pmatrix} \begin{matrix} \} \text{ bloque 1 } A_1^t \\ \\ \\ \} \text{ bloque q } A_q^t \end{matrix} \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \\ x_{n-1} \\ x_n \end{pmatrix} \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ \\ b_{m-1} \\ b_m \end{pmatrix}$$

2). Idea de proyección sobre un bloque:

El operador B_i puede ser un operador que proyecte al iterado actual sobre el i-ésimo bloque, es decir, que lo transforme en un punto que sea lo más cercano a él y que satisfaga las ecuaciones del bloque.

Para poder proyectarnos a un bloque **i** desde un punto $\mathbf{x}^k$ necesitamos una dirección que permita movernos al bloque. Cuando las distancias son medidas en norma 2, la dirección, que llamaremos $\mathbf{d}_i$, es una combinación lineal de las filas del bloque, es decir, si la matriz $\mathbf{A}_i^t$ tiene **filas** $\mathbf{a}_{i1}, \mathbf{a}_{i2},\ldots, \mathbf{a}_{il}$ entonces

$$d_i = \mu_{i1}.\mathbf{a}_{i1} + \mu_{i2}.\mathbf{a}_{i2} + \mu_{i3}.\mathbf{a}_{i3} + \mu_{i4}.\mathbf{a}_{i4} + \ldots\ldots + \mu_{il}\,\mathbf{a}_{il}\text{, con}$$

l=cantidad de filas del bloque **i** . En notación matricial diremos que la dirección $\mathbf{d}_i$ es igual a la multiplicación de la matriz $\mathbf{A}_i$ por el vector μ_i. de componentes $(\mu_{i1}, \mu_{i2},\ldots\ldots, \mu_{il})$

$y_1^k = x^k + d_1 = x^k + A_1 . \mu_1^{(k)}$ (va al bloque 1)

d_1^k

Desde x^k,

d_q^k

$y_q^k = x^k + d_q = x^k + A_q . \mu_q^{(k)}$ (va al bloque q)

donde el vector $\mu_i^{(k)}$ es elegido de modo tal que $x^k + d_i$ satisfaga las ecuaciones del bloque **i**, es decir

$A_i^t . y_i^k = A_i^t . (x^k + d_i) = b_i$ lo que lleva a
$A_i^t . (x^k + A_i . \mu_i^{(k)}) = A_i^t . x^k + (A_i^t . A_i) . \mu_i^{(k)} = b_i$

Así $\mu_i^{(k)}$ debe hallarse resolviendo el sistema

$$(A_i^t . A_i) . \mu_i^{(k)} = b_i - A_i^t . x^k \qquad (2.10)$$

Si $rgo(A_i^t)$=l entonces $(A_i^t . A_i)$ es de rango l y tiene inversa. Ahora multiplicando por $(A_i^t . A_i)^{-1}$ a ambos lados de la desigualdad despejamos $\mu_i^{(k)}$; así:

$$\mu_i^{(k)} = (\mathbf{A_i^t . A_i})^{-1} . (b_i - A_i^t . x^k)$$

- Por ser $(A_i^t . A_i)$ una matriz simétrica y definida positiva se puede hallar su descomposición de Cholesky, es decir

$A_i^t . A_i = L_i . D_i . L_i^t$

donde, L_i es una matriz triangular inferior con los elementos de su diagonal iguales a 1 y $\mathbf{D_i}$ es una matriz diagonal. Así la resolución de (2.10) se reduce a la solución de dos sistemas triangulares.

- Número de condición de una matriz:

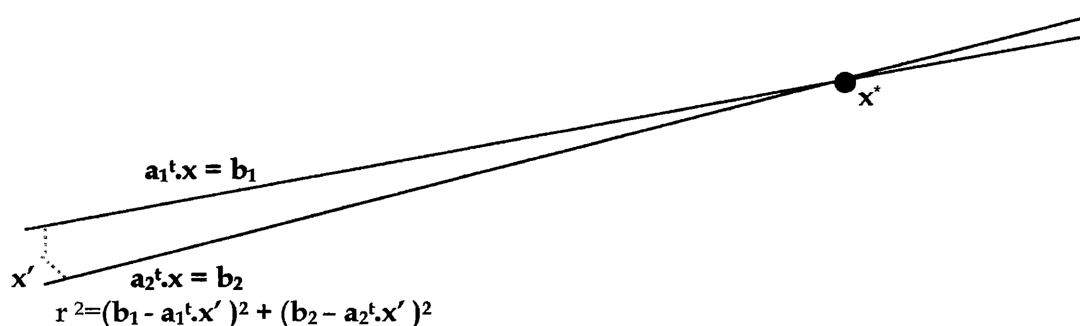
Es sabido que cuando no se trabaja con aritmética exacta, la solución **x'** obtenida es en realidad la solución de un sistema $(A + \Delta A)^t \mathbf{x'} = b + \Delta b$.
Así **x'** es una "aproximación" a la solución de $A^t . x = b$ buscada. Si definimos al residuo $r = A^t . \mathbf{x'} - b$ (lo que le falta a **x'** para satisfacer las ecuaciones) puede mostrarse que el error relativo será

$$||x^* - \mathbf{x'}|| / ||x^*|| \le ||A|| . ||A^{-1}|| . ||A^t . \mathbf{x'} - b|| / ||b||$$

Se denomina número de condición K(A) de la matriz no singular A relativo a una norma a

$$\mathbf{K(A) = ||A|| . ||A^{-1}||}$$

El número de condición es siempre un número mayor o igual a 1.
Notar que si K(A) es muy grande, una solución aproximada **x'** para la cual $||r|| = ||A^t . \mathbf{x'} - b||$ sea chica puede sin embargo estar muy lejos de la verdadera solución x^*, es decir tener $||x^* - \mathbf{x'}||$ grande.



Esto lleva a tratar de subdividir a la matriz A^t en bloques para los que las matrices ($\mathbf{A_i^t . A_i}$) tengan un número de condición K($\mathbf{A_i^t . A_i}$) aceptable.
Para ello vamos a construir los bloques en forma recursiva, estableciendo un control del número de condición a través de una estimación del mismo.

➢ Formación de los bloques:

Vamos a suponer que se han normalizado las filas del sistema original $A^t.x=b$, es decir, $||a_j||=1$. Dado un bloque en formación A_i^t, con j filas actualmente incluidas en él, y con un estimado α_i del número de condición de ($A_i^t .A_i$) menor que una cierta tolerancia k, se acepta la inclusión de una nueva fila a_{j+1} en A_i^t si el estimado del número de condición del bloque ampliado (es decir, al agregar a_{j+1} en A_i^t) no supera la tolerancia establecida.
Si llamamos $A_i' = [A_i\ a_{j+1}]$, el bloque ampliado tendrá las filas de $A_i'^t$.

Para que la matriz ($A_i'^t .A_i'$) tenga un "buen número de condición" hay que decidir qué filas incorporar para formar el bloque ampliado.

Un estimador del número de condición de una matriz M de la cual conocemos su descomposición de Cholesky $M=L^t.D.L$ es $\mathbf{K(M)\approx máx(D_{i\,i})/mín(D_{i\,i})}$

➢ Cuando se comienza a armar el bloque A_i,

$A_i^t = [\ a_{i1}^t\ ]$, es decir, tenemos una sola fila en el bloque.

Como el sistema está normalizado, $A_i^t .A_i = a_{i1}^t . a_{i1} = ||\ a_{i1}\ ||^2=1$

Como $L_i = [1]$ y $A_i^t .A_i = L_i. D_i . L_i^t$ entonces $[1]= [1]. D_i .[1]$ es decir $D_i =[1]$, y, en consecuencia, el estimado del número de condición de $A_i^t .A_i$ es 1.
Así, cuando comenzamos a formar un bloque, cualquier fila que decidamos agregarle hará que el bloque quede bien condicionado (la primer fila que forma un bloque siempre es aceptada).

➢ La idea del proceso de división es agregar una fila al bloque A_i sólo si el número de condición del bloque ampliado es menor que una tolerancia dada. Por simplicidad de notación supondremos que la fila $\mathbf{a_{ih}}= a_h$. Si

$$A_i^t = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \cdots\cdots \\ a_l \end{pmatrix} \quad \text{llamaremos } A'^t_i = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \cdots\cdots \\ a_l \\ a_{l+1} \end{pmatrix}$$

al bloque resultante de agregar la fila a_{l+1} al bloque A_i .

Sabemos que $A_i^t .A_i = L_i. D_i . L_i^t$ (descomposición de Cholesky de $A_i^t .A_i$)

Ahora $A'^t_i .A'_i = L'_i. D'_i . L'^t_i$

Veremos que se pueden definir L'_i y D'_i en forma recursiva, es decir, en base a las ya conocidas L_i y D_i:

$$A'^t_i . A'_i = \begin{bmatrix} A_i^t \\ a_{l+1}^t \end{bmatrix} . \begin{bmatrix} A_i & a_{l+1} \end{bmatrix} = \begin{bmatrix} A_i^t . A_i & A_i^t . a_{l+1} \\ a_{l+1}^t . A_i & 1 \end{bmatrix} \qquad (2.11)$$

$$1 = a_{l+1}^t . a_{l+1} = \| a_{l+1} \|^2$$

Queremos que L'_i y D'_i sean "ampliación" de L_i y D_i , y a su vez la descomposición de $A_i'^t.A'_i$:

i. Las matrices L'_i y D'_i son las respectivas "ampliaciones" de L_i y D_i si

$$L'_i = \begin{bmatrix} L_i & 0 \\ h_{l+1}^t & 1 \end{bmatrix} \qquad D'_i = \begin{bmatrix} D_i & 0 \\ 0^t & d_{l+1} \end{bmatrix}$$

donde **h_{l+1} es un vector de l componentes y $d_{l+1} \in \Re$.**

ii. Para que sea la descomposición de Cholesky de $A'^t_i . A'_i$

$A'^t_i . A'_i = L'_i . D'_i . L'^t_i = L'_i .(D'_i . L'^t_i)$ (por asociatividad del producto de matrices)

$$\begin{bmatrix} L_i & 0 \\ h_{l+1}^t & 1 \end{bmatrix} . \begin{bmatrix} D_i & 0 \\ 0 & d_{l+1} \end{bmatrix} . \begin{bmatrix} L_i^t & h_{l+1} \\ 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} L_i & 0 \\ h_{l+1}^t & 1 \end{bmatrix} \begin{bmatrix} D_i . L_i^t & D_i h_{l+1} \\ 0... & d_{l+1} \end{bmatrix} =$$

$$\begin{bmatrix} L_i . D_i . L_i^t & L_i^t . D_i . h_{l+1} \\ h_{l+1}^t . D_i . L_i^t & h_{l+1}^t . D_i . h_{l+1} + d_{l+1} \end{bmatrix} \qquad (2.12)$$

h_{l+1} y d_{l+1} son las incógnitas.

Queremos que (2.11) = (2.12), con lo cual:

a. $L_i^t . D_i . h_{l+1} = A_i^t . a_{l+1}$; despejando tenemos que $\boxed{h_{l+1} = D_i^{-1} . L_i^{-t} . A_i^t . a_{l+1}}$

b. $\boxed{d_{l+1} = 1 - h_{l+1}^t . D_i . h_{l+1}}$

Tenemos un nuevo $\mathbf{d_{l+1}}$,con lo cual podemos estudiar el número de condición del bloque ampliado.

Vimos que $D_{11} = 1$ y se puede mostrar que $d_{l+1} \leq 1$, es decir, máx $D_{i\,i} = 1$. Así, pues, la aceptación de la nueva fila depende del valor $1/\ d_{l+1}$, ya que

- Si $\mathbf{d_{l+1} \geq}$ **mín** $_{h=1,l}$ $\mathbf{D_{h\,h}}$, entonces el mínimo sigue siendo el mismo, y, en consecuencia, el estimador del número de condición de la matriz no varía, y ACEPTAMOS la nueva fila a_{l+1}.
- Si $\mathbf{d_{l+1} <}$ **mín** $_{h=1,l}$ $\mathbf{D_{h\,h}}$, entonces
  - si $1/d_{l+1} \leq$ tolerancia de aceptación, entonces ACEPTAMOS la nueva fila
  - si $1/d_{l+1} >$ tolerancia, entonces RECHAZAMOS la nueva fila en el bloque A_i ya que si la aceptamos, el nuevo bloque quedaría mal condicionado

Observación:

No debemos olvidar que para el cálculo de las proyecciones empleamos $\mathbf{(A_i^t . A_i)^{-1}}$. Todo el análisis previo partió de la definición de la fórmula de $\mu_i^{(k)}$, con lo cual sería deseable tener ésta inversa. Analicemos si con los datos definidos hasta ahora podemos darle forma:

➢ $\mathbf{(A_i^t . A_i)^{-1}} = (L_i . D_i . L_i^t)^{-1} = L_i^{-t} . D_i^{-1} . L_i^{-1}$

i. D_i^{-1} es fácil de calcular dada D_i, ya que la inversa de una matriz diagonal A es otra matriz diagonal B tal que $b_{i\,i} = 1/\ a_{i\,i}$

ii. $L'_i{}^{-1} . L'_i = I =$

$$\begin{bmatrix} L_i^{-1} & 0 \\ v_{l+1}^t & 1 \end{bmatrix} . \begin{bmatrix} L_i & 0 \\ h_{l+1}^t & 1 \end{bmatrix} = \begin{bmatrix} L_i^{-1} . L_i & 0 \\ v_{l+1}^t . L_i + h_{l+1}^t & 1 \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0^t & 1 \end{bmatrix}$$

Se sabe que la inversa de una matriz triangular que tiene unos en la diagonal es otra matriz triangular con unos en la diagonal.

$\mathbf{v_{l+1}^t}$ es la nueva incógnita.

Despejando obtenemos que $v_{l+1}^t . L_i + h_{l+1}^t = 0$, con lo cual

$$\boxed{v_{l+1}^t = -\, h_{l+1}^t . L_i^{-1}}$$

es decir, $\mathbf{v_{l+1}^t}$ se calcula recursivamente en base a $\mathbf{L_i^{-1}}$ y el valor calculado $\mathbf{h_{l+1}^t}$.

❖ A continuación describiremos el algoritmo que realiza la partición con la metodología propuesta. Denotaremos con $M=\{1,....,m\}$, $I_a=\{l:\ a_l$ fila de A^t asignada a algún bloque$\}$. Al inicializar la formación del bloque i, denominaremos $I_i= M - I_a$.

inicialmente i=1, $I_a = \emptyset$, setear mp=cant. máxima de filas por bloque, k=tolerancia de aceptación de fila

mientras $I_a \neq M$ **hacer**

inicializar $I_i = M - I_a$, j=1, est=1

elegir $l \in I_i$

hacer $A_i = \{a_l\}$, $D_j = [1]$, $L_j^{-1} = [1]$, $I_i = I_i - \{l\}$, $I_a = I_a \cup \{l\}$

mientras ($j < mp$ and $I_i \neq \emptyset$) **hacer**

elegir $l \in I_i$

calcular $\mathbf{h_{j+1}} = D_j^{-1}. L_j^{-1}. A_i^{-t}. a_l$

calcular $\mathbf{d_{j+1}} = 1 - h_{j+1}^t. D_j. h_{j+1}$

si est1 = $1/ d_{j+1} < k$ **entonces**

actualizar $A_i = [A_i\ a_l]$

$$L_{j+1} = \begin{bmatrix} L_j^{-1} & 0 \\ -h_{j+1}^t. L_j^{-1} & 1 \end{bmatrix} \quad D_{j+1} = \begin{bmatrix} D_j & 0 \\ 0 & d_{j+1} \end{bmatrix}$$

est=min(est,1/ d_{j+1})

$I_i = I_i - \{l\}$

$I_a = I_a \cup \{l\}$

j = j +1

sino

$I_i = I_i - \{l\}$

fin

fin

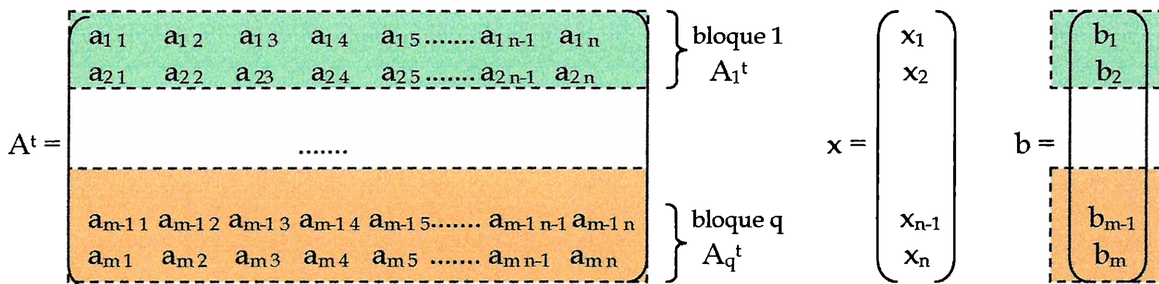
i=i+1

fin

- Observar que *"elegir $l \in I_i$"* no especifica *cómo* elegir la fila a ser analizada: este criterio se deja librado a la implementación. Así, se pueden ir tomando las filas consecutivamente de arriba hacia abajo, de abajo hacia arriba, o realizar la selección en forma intercalada. No se puede saber a priori qué filas van a formar cada bloque; lo que sí asegura el algoritmo es que los bloques que se forman están *bien condicionados,* siendo esta una propiedad *sumamente importante* para hallar una "buena solución" en cualquier algoritmo que resuelva sistemas de ecuaciones.

II.3 Algoritmo de proyección en bloques acelerado.

Una vez que se ha realizado el proceso de división en bloques bien condicionados, (reordenando las filas del sistema original para obtener bloques donde el número de condición de la matriz correspondiente a cada uno de ellos es aceptable) tenemos como resultado la siguiente descomposición del sistema $A^t.x=b$:



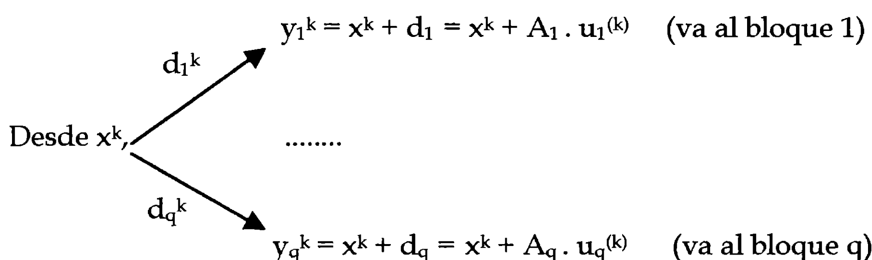
Asumiendo que el sistema $A^t.x=b$ tiene solución, la idea básica del algoritmo de búsqueda de una solución para el sistema es la siguiente:

En principio, dado un punto x^0 inicial, hallamos las direcciones d_i^0 de proyección a cada uno de los bloques A_i y las combinamos con un criterio de optimalidad para obtener una dirección d^0 que minimice la distancia del nuevo iterado $x^1 = x^0 + d^0$ al conjunto de soluciones del sistema.

A lo largo de todo el proceso iterativo las direcciones son combinadas con el mismo criterio de optimalidad. Esto sumado a la propiedad de que la *dirección que lleva al óptimo está en el hiperplano ortogonal a la dirección calculada en la iteración anterior,* es usado para **acelerar** el algoritmo. En conclusión, en cada iteración k del algoritmo, **k>0**, las direcciones d_i^k de proyección a los bloques son proyectadas sobre dicho hiperplano, ortogonal a la dirección $x^k - x^{k-1}$,para luego combinarlas óptimamente y así hallar la dirección d^k deseada.

Nuestro primer paso entonces es resolver como proyectar, en cada iteración k del proceso, un punto x^k a un bloque A_i. Como hemos mencionado anteriormente, para poder hacerlo necesitamos una dirección de proyección hacia el bloque. Esa dirección, que llamaremos d_i^k, es una combinación lineal de las filas del bloque, es decir, es igual a la multiplicación de la matriz A_i por un vector $u_i^{(k)}$ de componentes $(u_{i1}, u_{i2}, \ldots, u_{il})$ donde, como hemos visto al comienzo de la sección II.2, este vector es elegido de modo tal que $x^k + d_i^k$ satisfaga las ecuaciones del bloque A_i, es decir,

$$u_i^{(k)} = (A_i^t . A_i)^{-1} . (b_i - A_i^t . x^k) \qquad (2.13)$$



Es en el cómputo de (2.13) donde nos da sus frutos el esfuerzo realizado en el proceso de división en bloques, puesto que para cada bloque A_i tenemos la descomposición de Cholesky de $(A_i^t . A_i)^{-1} = (L_i . D_i . L_i^t)^{-1} = L_i^{-t} . D_i^{-1} . L_i^{-1}$ .Luego el cálculo (2.13) se reduce al producto de una matriz ya calculada y almacenada, por un vector, donde la matriz es producto de D_i^{-1} , matriz diagonal y L_i^{-t} y L_i^{-1} son matrices triangulares.

De esta manera podemos obtener $d_i^k = A_i . u_i^{(k)}$.

En principio, esta sería la dirección de proyección al bloque A_i deseada, pero este algoritmo "acelerado" tiene como característica importante que luego de una primera iteración, se puede utilizar la información de la iteración anterior para obtener una dirección "mejorada" para moverse desde el punto x^k . La idea es la siguiente:

A partir de la segunda iteración del algoritmo y una vez que hemos calculado la dirección d_i^k, sabemos que:

- la dirección d_i^k "apunta" hacia el bloque A_i
- se puede "mejorar la dirección"[1] d_i^k en la cual moverse, proyectándola en el hiperplano ortogonal a la dirección $x^k - x^{k-1}$ recién calculada. A esta nueva dirección mejorada la llamaremos Pd_i^k

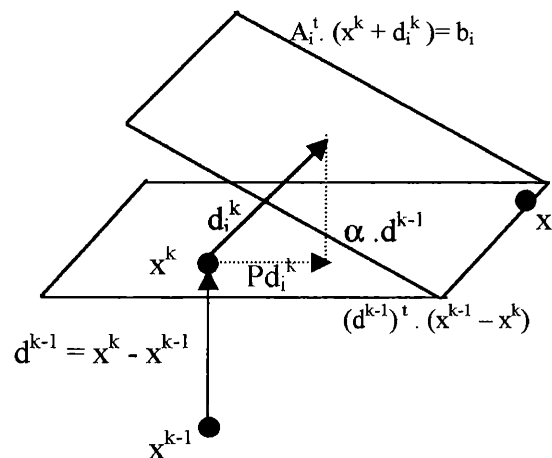
Escribiendo $d_i^k = \alpha . d^{k-1} + Pd_i^k$, $\alpha \in \Re$, tenemos que $\quad Pd_i^k = d_i^k - \alpha . d^{k-1} \quad$ (2.14)

Sabemos que $(d^{k-1})^t . (Pd_i^k) = 0$,porque son perpendiculares, entonces si multiplicamos ambos lados de (2.14) por $(d^{k-1})^t$ nos queda que $0 = (d^{k-1})^t . d_i^k - \alpha . (d^{k-1})^t . d^{k-1}$

Como $(d^{k-1})^t . d^{k-1} = \| d^{k-1} \|^2$ despejando llegamos a , $(d^{k-1})^t . d_i^k = \alpha . \| d^{k-1} \|^2$ con lo cual tenemos que $\alpha = (d^{k-1})^t . d_i^k / \| d^{k-1} \|^2$

Reemplazando en (2.14), obtenemos $\boxed{Pd_i^k = d_i^k - ((d^{k-1})^t . d_i^k / \| d^{k-1} \|^2) . d^{k-1}}$

Gráficamente:



➢ En azul se ve la proyección al bloque A_i (d_i^k). En rojo se puede apreciar cómo esta dirección es "mejorada" (Pd_i^k), **en el sentido de que está sobre el hiperplano apropiado.**

[1] Esta "mejor dirección" es la que **acelera** el algoritmo.

En conclusión, en la primer iteración del cálculo de la solución obtenemos la dirección d_i^0 para cada bloque A_i , formando una matriz de direcciones D' con tantas filas (d_i^0) como bloques tenga el sistema; en las iteraciones posteriores, vamos a obtener "direcciones mejoradas" Pd_i^k, $k>0$, que, de la misma manera, conformarán la matriz D' de direcciones.

Una vez conseguidas las direcciones d_i^k "intermedias" debemos "combinarlas" para obtener una dirección $\mathbf{d^k}$ en la cual movernos. Pero no buscamos una combinación cualquiera sino la que forme la "mejor dirección", en el sentido que al moverse desde el punto x^k en esta dirección, **se obtenga un nuevo punto x^{k+1} lo más cercano posible al punto x^***, solución del sistema.

Entonces el objetivo que debemos definir queda expresado por:

$$\textbf{minimizar } \| x^k + d^k - x^* \|^2$$

con el cual encontraremos la "mejor combinación", tal cual lo refleja el siguiente procedimiento:
Suponiendo que hay j direcciones d_i^k en la matriz D', combinar esas direcciones significa hallar un vector w tal que:

$$\text{minimice } \| x^k + d^k - x^* \|^2 \text{ ; siendo } d^k = (w_1 . d_1^k + w_2 . d_2^k + \ldots + w_j . d_j^k)^t$$

Realizamos el siguiente desarrollo:

$$\begin{aligned} \| x^k + d^k - x^* \|^2 &= (x^k + d^k - x^*)^t . (x^k + d^k - x^*) = \\ & (x^k - x^* + d^k)^t . (x^k - x^* + d^k) = \\ & (x^k - x^*)^t . (x^k - x^*) + 2 (x^k - x^*)^t . d^k + (d^k)^t . d^k = \\ & \| x^k - x^* \|^2 + 2 (d^k)^t . (x^k - x^*) + (d^k)^t . d^k \end{aligned}$$

considerando que:

(i) $$\begin{aligned} (d^k)^t . (x^k - x^*) &= (w_1 . d_1^k + w_2 . d_2^k + \ldots + w_j . d_j^k)^t . (x^k - x^*) = \\ & w_1 . (d_1^k)^t . (x^k - x^*) + \ldots + w_j . (d_j^k)^t . (x^k - x^*) \end{aligned}$$

y usando la propiedad: $\quad d_i^k . (x^k - x^*) = - \| d_i^k \|^2 \quad (k=0) \quad$ (2.15)
llegamos a que

$$(d^k)^t . (x^k - x^*) = w_1 . (- \| d_1^k \|^2) + w_2 . (- \| d_2^k \|^2) + \ldots + w_j . (- \| d_j^k \|^2)$$

(ii) $$(d^k)^t . d^k = (w_1 . d_1^k + w_2 . d_2^k + \ldots + w_j . d_j^k)^t . (w_1 . d_1^k + w_2 . d_2^k + \ldots + w_j . d_j^k)$$

Luego como $\| x^k - x^* \|^2$ es una constante, minimizar $\| x^k + d^k - x^* \|^2$ consiste en calcular:

$$\text{mín} \left(-2 w_1 . \| d_1^k \|^2 + \ldots + -2 w_j . \| d_j^k \|^2 + (w_1 . d_1^k + \ldots + w_j . d_j^k)^t . (w_1 . d_1^k + w_j . d_j^k) \right).$$

Como las d_j^k son conocidas y así también las $\| d_j^k \|^2$,el problema se reduce a minimizar una función $f(w_1, w_2, \ldots, w_j)$, lo que equivale a hallar gradiente de $f = 0$.

Reescribiendo en f así,

$$w_1 . d_1^k + w_2 . d_2^k + \ldots + w_j . d_j^k = [d_1^k \ldots d_j^k] . \begin{pmatrix} w_1 \\ \vdots \\ w_j \end{pmatrix} = D' w,$$ siendo w el vector de

componentes $(w, w_2, \ldots, w_j)$,

resulta que

$$f(w) = -2\, w_1 . \| d_1^k \|^2 + \ldots + -2\, w_j . \| d_j^k \|^2 + (D' w)^t (D' w) =$$
$$-2\, w_1 . \| d_1^k \|^2 + \ldots + -2\, w_j . \| d_j^k \|^2 + w^t (D')^t D' w$$

de donde

$$\nabla f(w) = -2\,(\| d_1^k \|^2, \ldots, \| d_j^k \|^2) + 2(D'^t)D'w$$

Luego, $$\nabla f = 0 \;\; implica \;\; (D'^t . D') \cdot \begin{pmatrix} w_1 \\ \vdots \\ w_2 \end{pmatrix} = \begin{pmatrix} \| d_1^k \|^2 \\ \vdots \\ \| d_j^k \|^2 \end{pmatrix} \qquad (2.16)$$

- **Observación importante:**

Con el objetivo de resolver este sistema de ecuaciones en forma eficiente (2.16), vamos a considerar en D′ sólo aquellas direcciones que lleven a que ***(D′ᵗ. D′)*** -matriz simétrica- tenga un estimado de número de condición aceptable. Sabemos que D′ tiene tantas direcciones como bloques en los que quedó dividido el sistema original. Aplicamos entonces el mismo criterio del proceso de división en bloques, para obtener otra matriz D* bien condicionada: la primer dirección de D′ "es aceptada" en D*, y luego una nueva dirección es aceptada sólo si el número de condición de la matriz D* ampliada con la dirección como última fila, es menor que una tolerancia establecida.

Este proceso de **depuración de la matriz D′**, desde luego, se realiza antes de obtener la "mejor combinación" de direcciones. En este sentido, *el desarrollo realizado anteriormente es igualmente válido*: cambiamos D′ por D* y d_i^k serán, en consecuencia, las direcciones resultantes en D* en vez de ser las direcciones de D′. Consideramos hacer esta observación en este punto del desarrollo, para aclarar el por qué de la importancia de la "depuración" de la matriz D′.

La fórmula (2.16) se traduce, considerando esta observación importante, en:

$$(D^{*t} . D^*) \cdot \begin{pmatrix} w_1 \\ \vdots \\ w_2 \end{pmatrix} = \begin{pmatrix} \| d_1^k \|^2 \\ \vdots \\ \| d_j^k \|^2 \end{pmatrix} \qquad (2.17)$$

Culminando con nuestro desarrollo, multiplicamos a ambos lados de (2.17) por $(D^{*t} . D^*)^{-1}$, y tenemos que el vector w que minimiza $f(w)$ se puede calcular de la siguiente manera:

$$\begin{pmatrix} w_1 \\ \vdots \\ w_2 \end{pmatrix} = \left(D^{*t}.D^*\right)^{-1} \cdot \begin{pmatrix} \| d_1^k \|^2 \\ \vdots \\ \| d_j^k \|^2 \end{pmatrix} \tag{2.18}$$

En este punto debemos hacer dos consideraciones:

(i) Para la primer iteración del algoritmo, los términos independientes (vector de $\| d_j^k \|^2$) son los expresados en (2.18), pero en las iteraciones siguientes (cuando se usan las direcciones "mejoradas" $\mathbf{Pd_i^k}$) la propiedad (2.15) sumado a la definición de Pd_i^k llevan a:

$$\frac{(Pd_i^k)}{\| Pd_i^k \|}(x^* - x^k) = \frac{d_i^{k^t}}{\| Pd_i^k \|}(x^* - x^k) = \frac{\| d_i^k \|^2}{\| Pd_i^k \|} \tag{2.19}$$

así los términos independientes serán $\begin{pmatrix} \frac{\| d_1^k \|^2}{\| Pd_1^k \|} \\ \vdots \\ \frac{\| d_j^k \|^2}{\| Pd_j^k \|} \end{pmatrix}$

(ii) Nuevamente el esfuerzo puesto en ir armando la descomposición de Cholesky de $(D^{*t} . D^*)^{-1}$ a medida que se va realizando la depuración de las direcciones de D', encuentra justificativo en la robustez del cálculo para la resolución de (2.18) (disponemos de L^{-1} y D^{-1} de D^*, con lo cual podemos calcular $(D^{*t}.D^*)^{-1}$).

Una vez calculado el vector w, resta armar la dirección optimal d^k:

$$\mathbf{d^k = D^*.w}$$

Por último, nos movemos desde el punto x^k en la dirección d^k para obtener el iterado x^{k+1}, así:

$$\mathbf{x^{k+1} = x^k + d^k}$$

y empleamos alguno de los siguientes criterios de parada para analizar si el punto x^{k+1} es una solución aceptable; en caso contrario, se inicia una nueva iteración del algoritmo, tomando a $\mathbf{x^{k+1}}$ como nuevo punto de partida.

Criterios de parada del algoritmo

1. Cuando el residuo al cuadrado es menor que una tolerancia de aceptación de solución preestablecida, es decir, cuando $dist^2(A^t.x_k, b) <$ tolerancia.
2. Cuando el residuo al cuadrado es menor que la dimensión del problema por una tolerancia de aceptación de solución preestabelcida, es decir, cuando $dist^2(A^t.x_k, b) < n *$ tolerancia; con n = dimensión del problema.
3. Cuando máx $\| b_i - a_i^t .x_k \|$ es menor que la tolerancia de aceptación de la solución.

Los resultados que publicaremos en el Capítulo IV están basados en los criterios 1 y 2. Debe tenerse en cuenta que para n muy grande (por ejemplo, del orden de 10^9), el criterio 1 puede tardar muchísimas más iteraciones que los criterios 2 o 3 en llegar a una solución aceptable. Veamos qué sucede:

Si $n=10^9$, cuando $(bi - ai^t.x_k) \approx 10^{-10}$ para todo i :

Con el criterio1, el residuo=$10^9.(10^{-10})=10^{-1}$;es decir, necesito que ***cada*** $(bi - ai^t.x_k)$ sea del orden de 10^{-18} (casi 0) para parar.

Con el criterio 2, el residuo es del orden de $10^9.(10^{-10})=10^{-1}$
$10^{-1}<10^9.\ 10^{-9}=10^0=1$, con lo cual el algoritmo para.

El criterio 3 tiene la ventaja que para problemas de dimensión muy grande, puedo interrumpir el proceso de testeo de optimalidad cuando encuentro el primer índice para el cual $(bi - ai^t.x_k)$ > tolerancia, con el consecuente ahorro de tiempo de ejecución.

- **Convergencia:**

Teorema. La sucesión $\{ x^k \}$ generada por el algoritmo converge a una solución x^* de $A^t.x = b$.
Demostración. La demostración se obtiene como consecuencia de la hipótesis que el sistema $A^t.x = b$ tiene solución, y de la Proposición 3.2 del trabajo de U.M. García Palomares: "*Parallel Proyected Aggregation Methods for Solving The Convex Feasibility Problem*", que demuestra la convergencia de los métodos PAM en un esquema más general, junto con el trabajo "*A New method for Solving Large Sparse Systems of Linear Equations using Row Projections*" de H.Scolnik, N.Echebest, M.T.Guardarucci y M.C.Vacchino □

A continuación se resume el algoritmo que realiza el proceso de resolución del sistema con la metodología descripta, a partir de la partición obtenida luego del proceso de división en bloques.

inicio
k=0 , Tomar x^k inicial
Establecer tolerancia tol de parada y toldir, tolerancia de aceptación de dirección
mientras $dist^2(A^t . x^k, b)$ > tol **hacer**
  para cada bloque A_i **hacer**
    calcular vector $\mathbf{u}_i^k = (L^{-t}. D^{-1}. L^{-1}) . (b_i - A_i^t.x^k)$
    calcular $\mathbf{d}_i^k = A_i . u_i^k$
    $d'_i{}^k = \mathbf{d}_i^k$
    si k > 0 **entonces**
      {Calculamos la dirección mejorada}
      Recalcular $d_i^k = d_i^k - ((d^{k-1})^t . d_i^k)/ \| d^{k-1} \|^2) . d^{k-1}$
    fin
    Actualizar matriz $D' = [D'\ d_i^k]$
  fin

{normalización y depuración de la matriz de direcciones D'}
para cada dirección d_j^k de D' **hacer**
$d_j^k = d_j^k / ||d_j^k||$
si j =1 **entonces**
hacer $D_j = [1]$, $L_j^{-1}=[1]$
sino
calcular $l_{j+1} = D_j^{-1} . L_j^{-1} . D^{-t} . d_j^k$
calcular $z_{j+1} = 1 - l_{j+1}^t . D_j . l_{j+1}$
si est1 = $1/ z_{j+1}$ < toldir **entonces**

$$L_{j+1}=\begin{bmatrix} L_j^{-1} & 0 \\ -l_{j+1}^t . L_j^{-1} & 1 \end{bmatrix} \quad D_{j+1}=\begin{bmatrix} D_j & 0 \\ 0 & z_{j+1} \end{bmatrix}$$

est1=min(est1,$1/ z_{j+1}$)
fin
fin
fin

calcular vector $w^k = (L^{-t} . D^{-1} . L^{-1}) . \begin{pmatrix} \frac{\| d_1'^k \|^2}{\| d_1^k \|} \\ \vdots \\ \frac{\| d_j'^k \|^2}{\| d_j^k \|} \end{pmatrix}$

calcular $d^k = \begin{pmatrix} w_1^k & \cdots & w_j^k \end{pmatrix} . \begin{pmatrix} d_1^k \\ \vdots \\ d_j^k \end{pmatrix}$

$x^{k+1} = x^k + d^k$

k=k+1

fin
final

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

CAPÍTULO III
Desarrollo

La resolución de un problema [2] requiere de varios pasos, que van desde la comprensión del problema (**qué** es lo que queremos resolver), al diseño de una solución conceptual que luego será implementada, es decir, trasladada a un programa de computación en un lenguaje específico.

Una solución consiste básicamente de dos componentes: algoritmos y una forma de almacenar los datos. Un algoritmo es una especificación de un método para resolver un problema. Las acciones que el algoritmo ejecuta operan sobre conjuntos de datos. Cuando construimos una solución, debemos organizar las colecciones de datos de manera que se pueda operar sobre ellos fácilmente en la forma que el algoritmo lo requiera.

Tal vez esta descripción de una solución deje la falsa impresión de que toda la inteligencia en la resolución de un problema está puesta en desarrollar el algoritmo, y cómo se almacenan los datos cumple un papel secundario. Esto está muy alejado de la realidad, ya que se necesita mucho más que almacenar los datos: se necesita también **operar** sobre ellos. Por ejemplo, el tener que resolver sistemas de ecuaciones esparsos de grandes dimensiones nos llevó a trabajar con estructuras de datos que reflejen esta situación, para así poder operar sobre ellas en forma eficiente.

El poder de computación creció un millón de veces en el período 1955-1990, y se espera igual crecimiento para la próxima década ¿Cómo puede sustentarse un crecimiento tan rápido? La respuesta se encuentra en las arquitecturas de computación paralela: múltiples procesadores semiautónomos coordinados para la resolución de una tarea. Los últimos años de la década del 80 han sido testigos del incremento vertiginoso del paralelismo, donde el número de procesadores disponibles para la realización de una tarea pueden ser miles.

El primer diseño de computadoras paralelas fue motivado por las limitaciones de la arquitectura serie von Neumann. Desde entonces, el rápido avance tecnológico producido está transformando significativamente las ciencias exactas, sociales, y todas las ramas de la ingeniería. A medida que las máquinas paralelas fueron apareciendo, los científicos de diferentes disciplinas se dieron cuenta que el paralelismo era una forma natural de ver sus aplicaciones.

El paralelismo aumenta el tamaño y complejidad de modelos que pueden ser representados y resueltos por una computadora. Pero este crecimiento no sólo está sostenido por la arquitectura. Igualmente importante es el desarrollo de algoritmos apropiados, sustentados en un diseño que permita la descomposición del problema adecuadamente, para poder aprovechar efectivamente el uso de máquinas con arquitecturas paralelas.

En este capítulo veremos el ciclo de desarrollo completo de nuestro trabajo: "una implementación óptima de un algoritmo acelerado de proyección en bloques", y sus fases de optimización que nos condujeron a una implementación que explota las características paralelas del algoritmo.

[2] en este texto entenderemos por resolución de un problema al **proceso** completo de tomar la descripción inicial del problema y desarrollar un programa que lo resuelva.

III.1. DESARROLLO

III.1.1 Especificación del problema

El objetivo que perseguimos es llegar a una implementación óptima del algoritmo planteado y explicado en II.3.

Como vimos, dicho algoritmo resuelve sistemas de ecuaciones lineales, es decir, de la forma $\mathbf{A^t.x=b}$, donde

$\mathbf{A^t}$ es la matriz de los coeficientes del sistema, $\mathbf{A^t} \in \Re^{mxn}$

$\mathbf{x}$ es el vector de las incógnitas, $x \in \Re^n$,y

$\mathbf{b} \in \Re^m$ es el vector de los términos independientes

Características de la matriz A:

- Tal como lo demuestran los ejemplos que sirven de motivación práctica de este trabajo, nos interesan los casos en que **A** es una matriz de grandes dimensiones (miles de ecuaciones y miles de incógnitas).
- **A** puede ser una matriz ***densa o esparsa***, es decir, una matriz que tiene un alto porcentaje de elementos en 0.

Si bien queda en claro que una implementación para A densa contempla el caso esparso, consideramos conveniente definir estructuras de datos adecuadas para estos últimos, y tener así un mejor aprovechamiento de los recursos en la plataforma donde el programa sea ejecutado. En este caso, ahorramos espacio de almacenamiento y tiempo de ejecución; hay que tener en cuenta que en muchas plataformas puede resultar imposible alocar la matriz A; además, una representación esparsa ahorra tiempo de ejecución en diferentes cálculos por cuanto sólo operamos con los valores distintos de cero de la matriz, evitando así efectuar un recorrido exhaustivo a través de toda la estructura (por ejemplo, al multiplicar una matriz esparsa por un vector).

No debemos perder de vista que no sólo perseguimos una implementación del algoritmo, sino una implementación ***óptima*** del mismo. Debemos definir entonces qué será una implementación óptima para nosotros.

Diremos que una implementación del *algoritmo acelerado de proyección en bloques* alcanzó un nivel óptimo si:

- el tiempo de respuesta de la misma no puede a priori ser mejorado con ninguna otra implementación del algoritmo realizable sobre las plataformas de ejecución disponibles y
- la calidad de la solución obtenida en ese tiempo es aceptable,

a) Queremos el mejor tiempo de respuesta posible, justamente porque en muchos problemas de la vida real que tienen que resolver sistemas de ecuaciones como modelo matemático, el tiempo es un recurso crítico; por ejemplo, en la aplicación médica de reconstrucción de imágenes por tomografía computarizada analizada en el Capítulo I.

b) La calidad de la solución, es decir, parar lo más cerca posible de $\mathbf{x}^*$,debe ser garantizada por el algoritmo: en nuestro caso, esto se sustenta con el hecho de dividir el sistema en

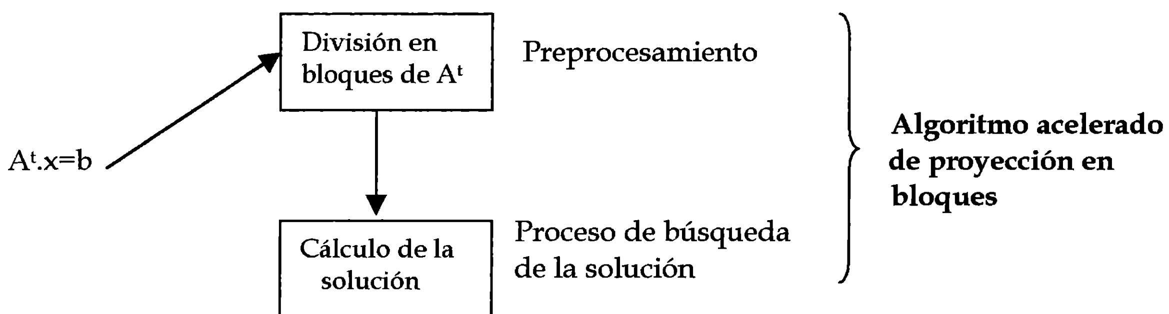
bloques bien condicionados usando un estimador del número de condición y con la combinación de las direcciones hacia los bloques con el objetivo de minimizar la distancia del nuevo iterado a $\mathbf{x}^*$. El algoritmo empleado intenta asegurar que las direcciones de proyección a los bloques estarán bien calculadas pues los bloques han sido formados con el objetivo de acotar el número de condición de las matrices involucradas en las proyecciones, aunque sabemos que cuando la matriz A está mal condicionada puede ocurrir que $||A^t x' - b||$ sea chico con $||x^* - x'||$ grande.

Debe poder verse además como a lo largo del proceso de optimización de la implementación del algoritmo los tiempos de respuesta mejoran.

Dadas las características del algoritmo, el proceso de optimización nos llevará naturalmente a una implementación en paralelo del mismo. Deberemos concluir además que una implementación paralela es más óptima que una implementación secuencial del algoritmo, de acuerdo a los parámetros de optimalidad definidos anteriormente.

Características del algoritmo que conducen a su implementación en paralelo:

Podemos decir que el algoritmo tiene dos fases bien diferenciadas: una primera que podemos llamar preprocesamiento, en donde el sistema original es dividido en bloques, y una segunda fase en donde se procede a buscar la solución.



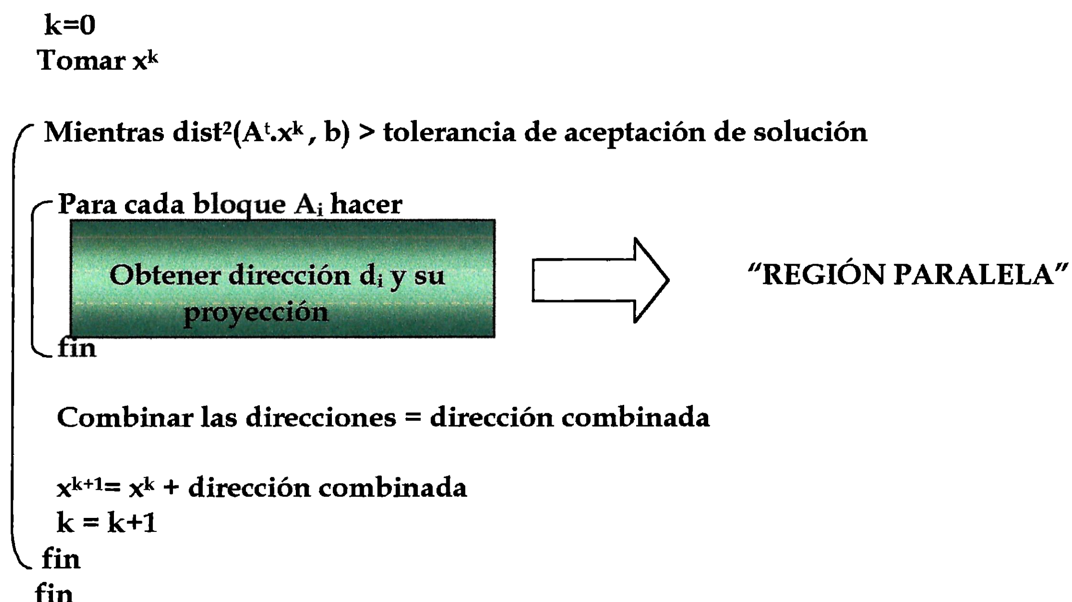
La primera fase no puede ser descompuesta en subprocesos independientes que definan una región paralela. Esta "falta de independencia" queda reflejada en el sentido que dos o más filas no pueden ser analizadas concurrentemente para su aceptación en un bloque determinado ya que luego de aceptar UNA fila en un bloque, se altera la composición del mismo, haciendo que todo análisis concurrente carezca de sentido: por ejemplo, supongamos que podemos analizar concurrentemente las filas f1 y f2 de A^t para su inclusión en el bloque A1. Entonces calcularemos en paralelo $\mathbf{h}_{j+1}$ y $\mathbf{d}_{j+1}$ para ambas filas. El ***absurdo*** de este cálculo concurrente queda reflejado en que si ambas filas pudiesen ser *aceptadas en el bloque,* la incorporación determinística o no determinística de cualquiera de ellas en A1 lleva a que se calcule la nueva descomposición de Cholesky del nuevo bloque "ampliado". Esta modificación en la composición del bloque A1 hace que NO SE PUEDA CONCLUIR QUE LA INCORPORACIÓN DE LA FILA RESTANTE AL BLOQUE HAGA QUE EL MISMO RESULTE BIEN CONDICIONADO. Y aún cuando la fila restante pudiese ser incorporada en el bloque A1 ampliado, esta inclusión involucraría tener que VOLVER A RECALCULAR $\mathbf{h}_{j+1}$ y $\mathbf{d}_{j+1}$ porque los valores calculados en paralelo no son válidos ya que el bloque A1 cambió.

La primera fase es el cimiento de la segunda, ya que en ella calculamos las matrices triangular inferior y diagonal de la descomposición de Cholesky de $(A_i^t . A_i)$, y sus inversas que luego serán usadas en el cálculo de la solución.

Podemos decir que la segunda fase sí tiene características que la hacen paralelizable, ya que, si bien cada iteración del proceso de búsqueda de la solución no es independiente de las anteriores (por cuanto cada iterado se calcula en base al punto anterior más el resultado de la "combinación" de las direcciones), los cálculos de las direcciones de proyección para cada bloque son independientes entre sí, exhibiendo así altas posibilidades de paralelismo si disponemos de una plataforma de corrida multiprocesador. Teniendo en cuenta lo denso en cantidad de los cálculos de las proyecciones, lograr distribuir esta importante porción del código entre múltiples procesadores debería ser un factor significativo de optimización del algoritmo. Que el cálculo de las direcciones hacia cada bloque sea independiente significa que "no se producen interferencias" entre las operaciones que se efectúan para calcularlas.

Esquemáticamente, a grandes rasgos tenemos:

Proceso de búsqueda de la solución de $A^t.x = b$



Output del procesamiento

El output principal del sistema es la solución x^k hallada por el algoritmo. Además, es importante también incorporar como parte del output al *residuo*, que nos permite tener una idea de cuán cerca hemos finalizado de satisfacer el sistema, es decir, la distancia de $(A^t. x^k)$ al vector ***b***.

Hemos agregado como output en nuestra implementación los tiempos de pre y procesamiento, así como el número de iteraciones del algoritmo hasta satisfacer el criterio de parada.

III.1.2 Diseño

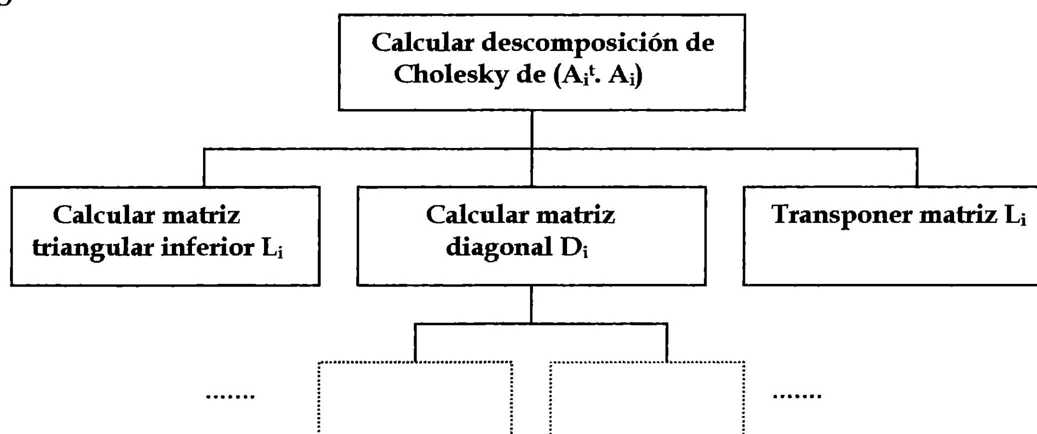
Existen muchas herramientas que ayudan a diseñar la solución a un problema. Cada metodología de programación introduce diferentes variantes o notaciones. En el Anexo I de este texto presentamos el refinamiento modular top-down del algoritmo, que fue la base de la implementación. Como se puede observar, es absolutamente independiente del lenguaje

de programación escogido para la implementación final. Obtuvimos así un diseño claro y transparente del algoritmo, que es uno de los requisitos básicos en el desarrollo de soft, ya que lo hacen entendible por terceros ajenos al desarrollo, y fácil de modificar y mantener. La modularidad a través del diseño top-down en sistemas complejos es la clave para obtener un diseño exitoso.

Hablamos de diseño top-down y modularidad, pero, ¿qué significan estos términos? La filosofía de un diseño top-down dice que se debería tratar un problema en sucesivos ***niveles de detalle***, dividiendo el programa en módulos independientes -procedimientos, funciones, o bloques de código con algún significado particular-.

Para ilustrar el concepto de solución modular, consideremos el siguiente ejemplo. Supongamos que queremos calcular la descomposición de Cholesky de (A_i^t. A_i). La figura III-1 usa una notación tipo árbol para ilustrar la relación jerárquica entre los módulos que resuelven este problema. En los niveles superiores, cada módulo no es más que una frase que dice *qué* es lo que va a resolver, evitando detalles. Cada módulo es luego refinado dividiéndolo en módulos adicionales más pequeños. El resultado es una jerarquía de módulos; cada módulo es refinado por sus sucesores, que resuelven problemas más pequeños y contienen más detalle que sus predecesores acerca de *cómo* resolver el problema. El proceso de refinamiento continúa hasta que los módulos al pie de la jerarquía son tan simples como para ser trasladados en procedimientos, funciones o bloques de código en un lenguaje de programación particular elegido por el diseñador. Un programa que se obtiene de esta manera se dice que es modular.

Figura III-1



En la figura III-1 se muestra cómo se dividió la solución en tres módulos:

Calcular matriz triangular inferior L_i
Calcular matriz diagonal D_i
Transponer matriz L_i

Así, se puede desarrollar cada módulo en forma independiente de los demás. De hecho, se busca diseñar módulos que sean tan independientes como sea posible. La *interfaz* de un módulo define el mecanismo de comunicación del mismo con los otros módulos. Especifica un contrato de comunicación entre módulos. Por ejemplo, la interfaz del módulo "Calcular matriz triangular inferior L_i" especifica que retorna una matriz triangular inferior, dada otra matriz que sea simétrica de entrada. "Calcular matriz diagonal D_i" tiene una interfaz compuesta por dos parámetros, en este orden: una matriz diagonal (de salida), y una matriz simétrica (de entrada). "Transponer matriz" tiene también dos parámetros, una matriz

transpuesta, de salida, y otra matriz, de entrada, que es la matriz a transponer. Veamos cómo queda la solución, una vez definidas las interfaces de los módulos:

Calcular matriz triangular inferior L_i **(**L_i**,(**A_i^t**.** A_i**))**
Calcular matriz diagonal D_i **(**D_i**,(**A_i^t**.** A_i**))**
Transponer matriz (L_i^t**,** L_i**)**

Llamando a los módulos en el orden visible, encontraremos la descomposición de Cholesky de (A_i^t. A_i). Nótese que en este punto no sabemos *cómo* cada módulo realiza su tarea, sino *qué* tarea realiza.

Cuando se usa diseño top-down y modularidad para resolver un problema, cada algoritmo comienza siendo una "caja negra". A medida que avanza el proceso de solución, se definen nuevas cajas negras para los submódulos; cada una de ellas dice qué hace, pero no cómo. Luego, son implementadas como subprogramas.

La *abstracción procedural* separa el propósito de un subprograma de su implementación. Una vez que un subprograma es escrito en un lenguaje, se puede usar sin necesidad de conocer los detalles del mismo o cómo está implementado. Asumiendo que el subprograma está debidamente documentado, se podrá usar simplemente conociendo su cabecera y descripción inicial, la cual incluye la descripción de sus parámetros (interface).

La modularidad y la abstracción procedural se complementan una con la otra. La primera abarca la tarea de dividir la solución en módulos; la segunda abarca la especificación clara de cada módulo *antes* de su implementación. Por ejemplo, qué precondiciones asume el módulo y, en lenguaje natural, qué acciones realizará: en el ejemplo, requerimos que las matrices de entrada de los módulos "Calcular matriz triangular inferior L_i" y "Calcular matriz diagonal D_i" sean simétricas. Estas especificaciones clarificarán el diseño de la solución porque permiten que nos centremos en la funcionalidad de alto nivel de la solución sin distraernos en los detalles de la implementación. Además, estos principios permiten que se modifique una parte de la solución sin afectar otras partes de la misma. En esta etapa no sólo especificamos el propósito sino también el flujo de datos entre los módulos. Para cada módulo respondemos a preguntas del tipo: ¿qué datos están disponibles antes de la ejecución del módulo?, ¿qué información asume el módulo?, ¿cómo se modifican los datos luego de su ejecución?

III.1.2.1 Estructuras de datos principales. Descripción.

Dijimos que la implementación del algoritmo consta de dos partes fundamentales:, una primera o preprocesamiento, en la cual la matriz del sistema original $A^t .x = b$ es dividida en bloques bien condicionados, determinando el sistema $A'^t.x=b$ (donde A'^t es una matriz cuyas filas son permutación de las filas de A^t ya que sobre ella hemos reordenado sus filas de manera de tener los bloques ordenados en forma consecutiva), y una segunda parte, que en base a la división anterior calcula la solución del nuevo "sistema ordenado". Sin lugar a dudas, el preprocesamiento define el cimiento sobre el que se sustenta el algoritmo de cálculo de la solución.

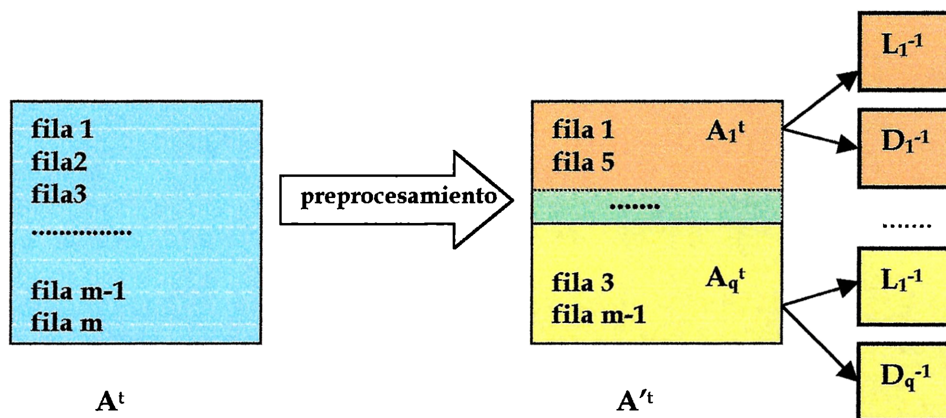
Notar que empleamos una nueva matriz A'^t para almacenar la matriz A^t ordenada en bloques consecutivos resultante del preprocesamiento. Esto ***no*** representa un gasto de espacio de almacenamiento extra, ya que la matriz A^t, que resulta obsoleta luego del preprocesamiento, la *reusamos* para almacenar las direcciones de proyección hacia cada bloque en cada iteración del cálculo de la solución.

Para ubicar cada bloque en A'^t definimos un arreglo ***vbloq*** que nos dice la cantidad acumulada total de filas por bloque. Así, sabemos que el bloque 1 comienza en la fila 1 de

A'^t, y el bloque i, i>1, comienza en la fila ***vbloq(i-1) +1***. La cantidad de filas del bloque 1 es ***vbloq(1)***, y la cantidad de filas del bloque i, i>1 es ***vbloq(i)-vbloq(i-1)***. Obsérvese que con un solo vector tenemos tanto la cantidad de filas por bloque como la ubicación física de comienzo de cada uno de ellos en A'^t.

Hasta ahora, en el Capítulo II hablamos de la potencia del preprocesamiento para definir bloques bien condicionados y su importancia; sin embargo, ahora nos va a interesar desde otro punto de vista fundamental: como fuente de datos para el cálculo de la solución. Esto es así ya que al dividir el sistema en bloques bien condicionados y obtener la descomposición de Cholesky de (A_i^t. A_i), tenemos de manera sencilla la inversa de la descomposición (L_i^{-1} y D_i^{-1}), que es empleada como dato estático invariable en el cálculo de la solución. Nótese que al tener **L_i^{-1}, tenemos también L_i^{-t}, ya que $(L_i^{-1})_{kj} = (L_i^{-t})_{jk}$** (para referenciar a un elemento de la matriz transpuesta de una dada, se invierten los índices de fila y columna). Debemos entonces asociar a cada bloque la información acerca de L_i^{-1} y D_i^{-1}.

Buscaremos estructuras de datos que reflejen esta situación, tratando de hacerlo de una forma eficiente.



- L_i^{-1} es una matriz triangular inferior, tal que $(L_i^{-1})_{jj} = 1$
- D_i^{-1} es una matriz diagonal

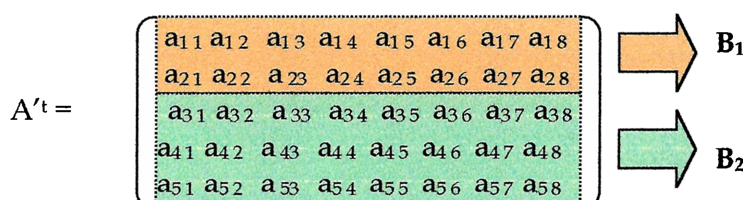
Existen dos casos bien diferenciados:

i. **A^t** es una matriz densa
ii. **A^t** es una matriz esparsa

<u>Observación</u>: el hecho de que A^t sea una matriz esparsa no garantiza que L^{-1} también lo sea.

i. **<u>Caso A matriz densa</u>**

Vamos a visualizar las estructuras con un ejemplo. Supongamos A'^t, matriz densa de dimensión 5x8, resultante del proceso de división en bloques de la matriz A^t:

A'^t =


Como vemos, A^t fue dividida en dos bloques; el primero contiene 2 filas y el segundo 3. Entonces $(A_1^t.A_1)$ será una matriz de 2x2 y $(A_2^t.A_2)$ será una matriz de 3x3.
Sean L_1^{-1} , D_1^{-1} y L_2^{-1} ,D_2^{-1} las matrices de la descomposición de Cholesky de los bloques 1 y 2 respectivamente, arrojadas por el preprocesamiento, con la siguiente forma.

$$L_1^{-1} = \begin{bmatrix} 1 & 0 \\ n1 & 1 \end{bmatrix} \qquad D_1^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & d1 \end{bmatrix}$$

$$L_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ n2 & 1 & 0 \\ n3 & n4 & 1 \end{bmatrix} \qquad D_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & d2 & 0 \\ 0 & 0 & d3 \end{bmatrix}$$

L_i^{-1} y D_i^{-1} son matrices cuadradas, y la dimensión de las mismas está dada por la cantidad de filas que contiene cada bloque.

- **$(L_i^{-1})_{1\,1}$ y $(D_i^{-1})_{1\,1}$ son 1 (ya que la primera fila de cada bloque siempre se acepta y tanto L_i^{-1} como D_i^{-1} en ese punto son [1] pues el sistema está normalizado)**
- **$(L_i)_{j\,j}$ =1**

Manteniendo dos estructuras separadas para almacenar cada matriz, desperdiciamos muchísimo espacio, ya que L_i^{-1} y D_i^{-1} son siempre triangular inferior y diagonal respectivamente, con lo cual sabemos a priori qué valores seguro son 0. Además, también sabemos a priori que los valores de la diagonal de L_i^{-1} son unos.
Sabiendo esto podríamos *unificar* el almacenamiento de las matrices de la descomposición de cada bloque en una sola estructura así:

$$U_1 = \begin{bmatrix} 1 & 0 \\ n1 & d1 \end{bmatrix}$$

$$U_2 = \begin{bmatrix} 1 & 0 & 0 \\ n2 & d2 & 0 \\ n3 & n4 & d3 \end{bmatrix}$$

U_i se forma almacenando en su diagonal los valores de D_i^{-1} y en su parte triangular inferior los elementos de L_i^{-1} .
Es más, la primer fila de cada U_i "no es necesario guardarla", ya que es el vector (1,0,......,0) conocido a priori.
Si bien la unificación de las estructuras ahorra considerablemente espacio de almacenamiento, se puede mejorar, ya que cada U_i sigue siendo una matriz triangular inferior ($u_{i\,j}$ = 0, si j>i), con lo cual, si es de dimensión nxn , estamos desperdiciando (n*n)/2 - n/2 espacios que a priori sabemos que son 0. ¡Imaginemos cuánto espacio desperdiciado en sistemas gigantes!
Para mejorar esto, nosotros hemos almacenado las matrices L_i^{-1} y D_i^{-1} "unificadas en un vector"que a partir de ahora llamaremos ***mv***, el cual se forma con las filas consecutivas de U_i **(excepto la primera que no se guarda)**, sin los elementos 0 que se encuentran a la derecha de la diagonal.

Para el ejemplo, ***mv*** es:

mv = [n1 d1 n2 d2 n3 n4 d3]

- Los datos están almacenados consecutivamente por bloques, y dentro de estos por filas

Junto a ***mv*** definimos otro vector, ***inimv***, tal que ***inimv[i]*** nos dice la posición donde comienza físicamente en ***mv*** el almacenamiento de L_i^{-1} y D_i^{-1} para cada bloque i. Si el bloque i tuviese una sola fila, entonces ***inimv[i]=-1*** (como la primer fila de L_i^{-1} y D_i^{-1} no es almacenada en esta estructura, -1 actúa como flag indicando que el bloque i tiene una sola fila, y, en consecuencia, L_i^{-1} y D_i^{-1} valen [1]).

Para el ejemplo que estamos desarrollando,

inimv= [1 3]

Esto significa que el bloque 1 comienza físicamente en la posición 1 de ***mv*** y el bloque 2 en la posición 3.

<u>Acceso a los datos</u>

- Puede ocurrir que un bloque k conste de una única fila, en cuyo caso L_k^{-1} y D_k^{-1} son [1]. Identificamos esta situación con ***inimv[k] = -1***

  Asumiendo que **inimv[k] ≠ -1,**

- Si queremos acceder al elemento (i,j) de la matriz triangular asociada a la descomposición de Cholesky del bloque k, entonces

  - $(L_k^{-1})_{ij} = 0$,si j>i
  - $(L_k^{-1})_{ij} = 1$,si i=j
  - $(L_k^{-1})_{ij} = mv[inimv[k] + (i^2 - i)/2 + j - 2]$,si j<i

    - con ***inimv[k]*** ubicamos el comienzo del bloque en ***mv***
    - **$(i^2 - i)/2$** es la distancia que nos movemos para encontrar la fila *i*:

$$\sum_{t=1}^{i-1} t = (i-1)i/2 = (i^2 - i)/2$$

- De la misma manera, si queremos acceder al elemento (i,j) de la matriz diagonal asociada a la descomposición de Cholesky del bloque k, entonces

  - $(D_k^{-1})_{ij} = 1$,si i=j=1
  - $(D_k^{-1})_{ij} = 0$,si i≠j
  - $(D_k^{-1})_{ij} = mv[inimv[k] + (i^2 - i)/2 + i - 2]$,si i=j, e i≠1

ii. **<u>Caso A matriz esparsa</u>**

Las matrices esparsas o ralas merecen una mención muy importante en este trabajo ya que los problemas de motivación vistos en el Capítulo I son de este tipo.

Una representación especial para esta clase de matrices se hace necesario, por un lado, para poder ahorrar espacio de almacenamiento, ya que si $A^t \in \Re^{mxn}$, no se alocan los mxn elementos de la misma, y por otro lado, para ganar en performance, ya que, como veremos por ejemplo en la multiplicación de matrices esparsas por vectores, ciertas operaciones se realizan sólo iterando sobre los valores almacenados, logrando así mucha más rapidez de cálculo y ejecución.

Para poder implementar la representación de una matriz esparsa, necesitaremos de algún esquema de almacenamiento indexado, que almacene sólo los elementos distintos de 0 de la matriz, junto con suficiente información auxiliar para determinar la ubicación lógica de cada elemento en la matriz. No hay un esquema standard a seguir para desarrollar este tipo de implementaciones. Por ejemplo, " theYale Sparse Matrix Package" e "ITPACK" describen varios métodos. El que nosotros hemos elegido es el empleado por "PCGPACK", que es similar a uno de los métodos del "Yale Sparse Matrix Package". La ventaja de este esquema, cuya traducción es *"modo de almacenamiento esparso indexado por filas"* es que **requiere almacenamiento para a lo sumo el doble de elementos distintos de 0 de la matriz (otros métodos pueden requerir de 3 a 5 veces este valor).**

Para representar una matriz $A \in \Re^{nxn}$,este método trabaja con dos vectores, llamados ***sa*** e ***ija***. El primero almacena elementos de la matriz, y el segundo números naturales. Las reglas de almacenamiento son las siguientes:

- Los primeros n lugares de ***sa*** almacenan los elementos de la diagonal de A, en orden. Notar que los elementos iguales a 0 de la diagonal *también* son almacenados.
- Cada uno de los primeros n lugares de ***ija*** almacena la posición de ***sa*** que contiene el primer elemento distinto de 0 correspondiente a cada fila, que no sea el elemento de la diagonal. Si no hay elementos distintos de 0 fuera de la diagonal para una fila, entonces se almacena la posición en ***sa*** al elemento más recientemente almacenado para la fila previa *más 1*.
- ***ija[1] = n***+2 (este valor puede ser leído para determinar n).
- ***ija[n+1]*** es *uno más* que la posición en ***sa*** para el último elemento distinto de 0 de la última fila (puede ser leído para determinar la cantidad de elementos distintos de 0 de la matriz, o el tamaño lógico de los vectores ***sa*** e ***ija***).
- Las posiciones ≥ n+2 en ***sa*** contienen los elementos fuera de la diagonal distintos de 0, ordenados por fila, y dentro de cada fila por columna.
- Las posiciones ≥ n+2 en ***ija*** contienen el *número de columna* del correspondiente elemento de ***sa***.

Aunque estas reglas pueden parecer un tanto rebuscadas a simple vista, producen un esquema de almacenamiento muy elegante y compacto. Por ejemplo, consideremos la matriz

$$\begin{pmatrix} 3.1 & 0 & 1 & 0 & 0 \\ 0 & 4.7 & 0 & 0 & 0 \\ 0 & 7 & 5.3 & 9 & 0 \\ 0 & 0 & 0 & 0 & 2.6 \\ 0 & 0 & 0 & 6.2 & 5 \end{pmatrix}$$

Siguiendo el método de almacenamiento descripto obtenemos la siguiente representación:

K	1	2	3	4	5	6	7	8	9	10	11
Ija(k)	7	8	8	10	11	12	3	2	4	5	4
Sa(k)	3.1	4.7	5.3	0	5	X	1	7	9	2.6	6.2

x es un valor cualquiera.

El elemento diagonal para la fila i es ***sa(i)***, y los elementos distintos de 0 para esa misma fila *fuera de la diagonal* se encuentran ***en sa(k)***, con ***k*** desde ***ija(i)*** hasta ***ija(i+1)-1***, si el límite superior es mayor o igual que el inferior.

- Observación1: esta forma de almacenamiento se puede extender para matrices no cuadradas, "completando con ceros" las filas o columnas necesarias hasta lograr una matriz de dimensión cuadrada, y así poder emplear la representación anterior; luego se agrega un "control" que permite no hacer referencia a los elementos agregados, *respetando así las dimensiones de la matriz original*. Notar que este "agregado" sólo se refleja en la diagonal, ya que los elementos iguales a cero fuera de ella NO se almacenan.
- Observación2: para **"localizar un bloque"** en esta estructura de almacenamiento esparsa, empleamos el vector ***vbloq*** visto al comienzo de III.1.2.1, que almacena la *cantidad acumulada de filas por bloque*. Así, el bloque 1 comienza en la fila 1 de la matriz esparsa, y en *sa(1) tenemos el elemento (1,1)* del bloque 1. El bloque i, i>1, comienza en la fila ***vbloq(i-1)+1***, y en *sa*(***vbloq(i-1)+1***), tenemos el elemento (***vbloq(i-1)+1, vbloq(i-1)+1***) del bloque i.

En base a este método generamos una implementación que dada una matriz esparsa A almacenada en forma densa, la "transforma" en la representación *esparsa indexada por filas* para su posterior manipulación.

El siguiente es el pseudocódigo del módulo que realiza la transformación (desarrollado para el caso A^t matriz cuadrada):

```
SUBROUTINE transforma (a,n,sa,ija)
**convierte la matriz cuadrada a(n,n) a la versión esparsa indexada por filas en sa e ija

**primero almacenamos los elementos de la diagonal**
do j=1,n
  sa(j)=a(j,j)
enddo

ija(1)=n+2
k=n+1

**loop sobre las filas**
do i=1,n
  **loop sobre las columnas**
  do j=1,n
     if (a(i,j)=0 and i≠j) then
      **almacenamos los elementos distintos de 0 fuera de la diagonal**
        k=k+1
        sa(k)=a(i,j)
        ija(k)=j
     end if
   enddo
**ya se completó la fila, entonces almacenamos el índice a la siguiente**
 ija(i+1)=k+1
enddo
end
```

Uno de los usos más importantes donde se puede ver la potencia de este modo de almacenamiento es cuando multiplicamos una matriz esparsa por un vector a derecha. De hecho, este modo de almacenamiento fue optimizado para este propósito. La siguiente es la rutina que lo realiza. Observemos la sencillez de la misma:

```
SUBROUTINE mult(sa,ija,x,b,n)
**multiplica la matriz esparsa almacenada en sa e ija por el vector x(1,n), dando por
**resultado el vector b(1,n)
do i=1,n
  b(i)=sa(i)*x(i)
  do k=ija(i) , ija(i+1)-1
    b(i)=b(i)+sa(k)*x(ija(k))
   enddo
  enddo
end
```

Notar que esta subrutina sólo multiplica los elementos distintos de cero de la matriz almacenada en forma esparsa, con el consiguiente ***ahorro de tiempo de cálculo***.

De esta manera, la rutina es empleada en cada iteración del algoritmo que calcula la solución del sistema para chequear si el iterado en el que estamos parados satisface el criterio de parada, es decir, cuando hacemos $A^t.x_k$.

Además, la hemos empleado con una pequeña adaptación en el preprocesamiento, cuando calculamos cada h_{j+1} al realizar el análisis de aceptación de cada fila en un bloque. En estos casos uno de los productos es $A_i^t.\ a_{j+1}$. La adaptación de la rutina se debió a que en este caso A_i^t es un ***bloque esparso***, es decir, una *porción* de la matriz cuadrada almacenada en forma esparsa, y además el bloque no es necesariamente cuadrado. Si llamamos ***inicio*** al índice de inicio del bloque A_i en ***sa***, y ***cant*** es la cantidad de filas del bloque A_i entonces la siguiente es la adaptación implementada a tal fin:

```
SUBROUTINE mult_bloque(sa,ija,x,b,n,inicio,cant)
 **multiplica el bloque esparso almacenada en sa que comienza en inicio y tiene cant
 **filas por el vector x(1,n), dando por resultado el vector b(1,n)**
 do i=idif,(idif+cant)
  b(i-idif+1)=sa(i)*x(i)
  do k=ija(i) , ija(i+1)-1
    b(i-idif+1)=b(i-idif+1)+sa(k)*x(ija(k))
   enddo
  enddo
 end
```

- Esta adaptación la empleamos de igual manera en el cálculo de la solución, al hacer $A_i^t.\ x^k$ cuando obtenemos la dirección hacia cada bloque.

III.2. IMPLEMENTACIÓN

Una de las primeras tareas previas a la codificación, fue la selección de la plataforma de corrida de nuestro programa.

Debíamos elegir una plataforma multiprocesador sobre la cual probar que efectivamente el algoritmo podía implementarse para ser ejecutado en forma paralela, y que los tiempos de

respuesta de dicha implementación disminuyen considerablemente si los comparamos con una implementación secuencial del mismo algoritmo (corrida en una plataforma serie escalar).

Definiciones:
Procesamiento secuencial: existe un único thread de control; el contador de programa comienza con la primer instrucción atómica del proceso y se mueve a través del mismo a medida que las acciones atómicas son ejecutadas, *una detrás de otra*.
Procesamiento paralelo: (se da en plataformas multiprocesador) el problema complejo se divide en un número de tareas más pequeñas entre varios procesadores. Las tareas se ejecutan concurrentemente. La distribución de los elementos de software entre los procesadores puede dejarse al sistema operativo o al *programador/diseñador*.

Un algoritmo paralelo es aquel que puede ser implementado en una máquina con arquitectura paralela, de manera tal que su ejecución ahorra tiempo de ejecución, si lo comparamos con un programa idéntico corrido bajo una plataforma serie o secuencial.

Esquema de arquitecturas multiprocesador:

i. Clasificación de procesadores por instrucción y datos.
La siguiente es una clasificación de arquitecturas propuesta por Flynn en 1972, y que es empleada en la actualidad para distinguir entre distintas alternativas de arquitecturas paralelas. Flynn propone las siguientes cuatro clases, basadas en la interacción entre las instrucciones y los datos de los procesadores:

- SISD (Single Instruction Single Data): sólo una instrucción es decodificada por ciclo de instrucción. La memoria afectada a la instrucción sólo es usada por ella. La arquitectura tradicional von Neumann es un monoprocesador escalar que cae en esta categoría.
- SIMD (Single Instruction Multiple Data): procesadores idénticos con memoria local ejecutan la misma instrucción bajo control de un host o supervisor. Los array processors son un ejemplo característico. El host (master) se encarga de la coordinación y utiliza los resultados parciales.
- MISD (Multiple Instruction Single Data): procesadores idénticos o no, con memoria local ejecutan diferentes instrucciones sobre un mismo conjunto de datos, bajo el control de un host o supervisor.
- MIMD (Multiple Instruction Multiple Data): procesadores idénticos o no, con memoria local ejecutan diferentes instrucciones sobre diferentes datos. Puede haber un host master o no. Data flow, systolic arrays, son los ejemplos más utilizados de estas arquitecturas.

ii. Existen varios *ejemplos de arquitecturas multiprocesador*: **vector processor** (procesador vectorial), **array processor, data flow** y **transputer** son las principales. A grandes rasgos,

- La arquitectura **vector processor** es muy empleada para realizar operaciones sobre vectores de datos. Un vector processor es una máquina SIMD, dado que la misma instrucción se ejecuta simultáneamente en un conjunto de procesadores con datos diferentes. Existe un control centralizado, normalmente manejado por un host SISD.
- La arquitectura **array processor** consiste de una grilla de procesadores SISD conectados configurando una arquitectura SIMD. La misma instrucción se ejecuta simultáneamente en un conjunto de procesadores con datos diferentes. Existe un control centralizado,

normalmente manejado por un host SISD. Cada procesador tiene una memoria local y capacidad de E/S local y puede comunicarse con los procesadores adyacentes.

- La arquitectura **data flow** se trata de un conjunto de n procesadores con arquitectura MIMD con bajo acoplamiento. Las instrucciones no se ejecutan en secuencia, ni bajo la coordinación de ninguna unidad de control. Simplemente cuando están disponibles los datos que se necesitan y algún procesador con capacidad de ejecución, las acciones son activadas y ejecutadas.
- Un **transputer** es un microprocesador RISC (Reduced Instruction Set Computer: microcomputadora de gran velocidad de procesamiento) de alta performance, con memoria en el chip y capacidad de comunicación externa. La arquitectura paralela se logra conectando varios transputers sobre un bus común bajo supervisión, o estructurando una grilla de transputers vinculados por comunicaciones de alcance localizado y coordinados por un master o coordinador.

En consecuencia, hay diferentes arquitecturas que logran el efecto de la computación paralela. Por ejemplo, la máquina **Cray PVP** tiene pocos procesadores muy poderosos que comparten una memoria muy grande. Otras computadoras, como por ejemplo **Sequent** o **BBN Butterfly**, tienen muchos procesadores menos poderosos que los de Cray, que comparten memoria. Otros procesadores paralelos muy utilizados, como los de **Intel** y **Ncube hipercubos**, tienen miles de procesadores. En estos casos, no se comparte una memoria central, sino la misma se encuentra distribuida, y cada procesador tiene su propia porción. En computadoras con memoria distribuida, los procesadores cooperan y se comunican por pasaje de mensajes.
Los ejemplos mencionados pertenecen a la clase MIMD. Esta clase incluye sistemas con un arreglo de procesadores asociados, cada uno ejecutando su propia secuencia de instrucciones independientes.
Para toda la variedad de arquitecturas, el total del poder de computación depende de algo más que la velocidad y el número de CPUs. Cada arquitectura tiene sus fortalezas y debilidades que afectan la performance global del sistema; por ejemplo, en máquinas con memoria distribuida, el costo de comunicación entre los procesadores puede ser elevado. En consecuencia, para obtener una buena performance, *las características del código* (así como la cantidad y nivel de paralelismo y el tipo de operaciones ejecutadas más frecuentemente) deben ser lo más adecuadas a la arquitectura de la máquina.

Dadas las características del algoritmo (algoritmo matemático con gran cantidad de operaciones sobre vectores y matrices), consideramos que la arquitectura vector processor es la más adecuada. Un convenio existente entre el Departamento de Matemática de la UNLP y la COPPE , Universidad Federal de Río de Janeiro nos posibilitó tener acceso a la supercomputadora CRAY J90 PVP (Parallel Vector Processing), cuya arquitectura se corresponde a la buscada, sobre la que finalmente ejecutamos los procesos y extrajimos los resultados que publicaremos en el Capítulo IV.

Elección del lenguaje de programación:

Necesitábamos un lenguaje de programación que, por un lado, tenga una gran precisión y robustez numérica, y cuyo ejecutable, por otro lado, pueda correr en la supercomputadora elegida.
El equipo **Cray J90 PVP** tiene compiladores muy potentes (en el sentido que optimizan el código fuente para explotar al máximo las características vectoriales de los procesadores) para los lenguajes Fortran 77, Fortran 90 y C++.

Performance y programabilidad son los dos factores más importantes a tener en cuenta en el momento de seleccionar un lenguaje de programación adecuado para la implementación de aplicaciones científicas.

Tradicionalmente Fortran 77 ha sido el lenguaje más empleado en implementaciones de tipo numérico científicas. Sin embargo la informática ha avanzado considerablemente desde la aparición de Fortran 77, surgiendo paradigmas de programación más modernos como el orientado a objetos que se sustenta en los conceptos de clasificación (encapsulamiento de "comportamiento" y estructura), herencia y polimorfismo. En consecuencia, Fortran 77 debería (y decimos debería porque todavía no ha desaparecido y sigue siendo empleado) dar paso a lenguajes que soporten estas técnicas, como C++ o Fortran 90.

Los métodos orientados a objetos pueden ser empleados para simplificar considerablemente el proceso de desarrollo de software. Es posible aprovechar la aplicación de estos conceptos en C++ o Fortran 90. Sin embargo, C++ es un lenguaje completamente orientado a objetos mientras que Fortran 90 implementa algunos de los conceptos del paradigma. Para el desarrollo de soft fuera del ámbito numérico científico, C++ es claramente mejor que Fortran 90. Sin embargo, la performance es un aspecto crítico. *Los compiladores para Fortran 90 se benefician con las décadas de experiencia en la optimización de Fortran77* (Fortran 77 es un "subconjunto" de Fortran 90); **consecuentemente, el código Fortran 90 será más rápido que su correspondiente en C++** (aunque la brecha ha disminuido en los últimos años). Cuando se consideran todos los aspectos generales involucrados en el ciclo de desarrollo de soft, C++ es , en muchos casos, la mejor opción. **Si la preocupación, en cambio, está centrada en obtener una buena performance, Fortran 90 (o Fortran 77) son la mejor opción cuando hablamos de aplicaciones científicas numéricamente sensibles.**

Varias publicaciones rescatan la robustez numérica de Fortran por sobre C++. Como el objetivo de nuestro trabajo es realizar una implementación óptima del algoritmo, y en muchos casos las matrices son "numéricamente sensibles" optamos por Fortran 90 como lenguaje de implementación.

La siguiente es una tabla comparativa de los lenguajes más empleados en computación científica: Fortran (en sus versiones 77 y 90) y C (en sus versiones C y C++) , extraída del ensayo *"Fortran90 versus C++ for Object Oriented Scientific Programming"* (www.math.liu.se/~frber), realizado por Fredrik Berntsson, investigador del Departamento de Matemática de la Universidad de Linköpings (Suecia).

	F77	**C**	**C++**	**F90**
Robustez numérica	2	4	3	1
Posibilidades de def. de paralelismo	3	3	3	1
Abstracción de datos	4	3	2	1
Conceptos de POO	4	4	1	2
Prog. fcional.	4	3	2	1
Promedio	**3,4**	**3,2**	**2,2**	**1,2**

1 es la mejor y 4 la peor calificación.

Una vez elegida la plataforma y el lenguaje, el siguiente paso fue lograr una implementación secuencial del algoritmo. Cray J90 PVP permite setear variables de entorno de corrida que posibilitan especificar cuantos procesadores estarán dedicados a nuestro proceso. *Seteando dicha variable en 1 e* ***inhabilitando la vectorización*** *en la compilación de*

nuestro código simulamos la ejecución en una plataforma secuencial monoprocesador, que nos permitió tener una ***marca inicial de nuestro proceso de optimización.***

Características técnicas de Cray J90 PVP:

- **8 CPUs: 10ns clock cycle y palabras de 64 bits**
- **200 MFLOPS por CPU**
- **Memoria: 2 Gbytes RAM (compartida)**
- **Disco: 72 Gbytes**
- **Sistema operativo: UNICOS (compatible con Unix System V)**
- **Compiladores: Fortran 77, Fortran 90 y C++**

III.2.1 Proceso de optimización

La optimización del código de una aplicación es una de las tareas que cobra más importancia cuando se quiere mejorar el tiempo de respuesta de la misma o cuando el manejo de los recursos es un tema crítico.
Existen una serie de tareas relacionadas con la optimización de un código:

- Analizar su performance para determinar y aplicar un método de optimización
- Determinar cuándo el código está completamente optimizado

Hay dos razones fundamentales para optimizar un código, y esas razones están directamente relacionadas con el criterio de optimización:

1. Obtener una respuesta más rápida
2. Minimizar el consumo de recursos

Obviamente, estos dos criterios no son mutuamente excluyentes; por ejemplo, reduciendo el consumo de CPU, se reduce el tiempo de respuesta del sistema. Sin embargo, existen casos en los que al mejorar el tiempo de respuesta se incrementa el uso de recursos como CPU o memoria. Es la persona encargada de optimizar quién decide qué criterio es más importante.
Se debe establecer un criterio de optimización para:

- Establecer una medida inicial o benchmark, usada para medir nuestro progreso durante la optimización (en nuestro caso tiempo de respuesta)

- Elegir técnicas específicas de optimización que mejoren la performance para el criterio elegido

Se debe poner énfasis en el chequeo de la correctitud de las respuestas luego de cada modificación efectuada en el código, y cada vez que se mejora el benchmark. Muchas técnicas de optimización reordenan operaciones provocando resultados erróneos en algoritmos que son numéricamente sensitivos.

Una vez establecida una marca inicial para nuestro código, podemos comenzar con el proceso de optimización. En particular, las tácticas de optimización utilizadas en nuestro trabajo caen dentro de las siguientes categorías:

- Inserción de directivas especiales en la compilación
  El ambiente provee un mecanismo muy sencillo para comunicar nuestras necesidades al compilador: por ejemplo, se puede compilar un código habilitando a full optimización escalar y vectorial, o incluso habilitar tasking automático (multitasking)

- Modificación del código fuente, por ejemplo para reflejar vectorización o la inserción de directivas de programación paralela

Por ello, antes de optimizar nuestro programa fue necesario estudiar el comportamiento de **CRAY J90 PVP**, ya que algunas de las tácticas de optimización utilizadas dependen de la manipulación de las opciones de compilación.

Dando directivas al compilador

Se pueden agregar directivas de compilación en el código fuente durante el proceso de optimización: los compiladores Cray J90 PVP tiene una sintaxis específica para expresarlas, las cuales se traducen como comentarios para otros compiladores, ***NO afectando así la portabilidad de nuestro código***. Específicamente, se le puede pedir que realice las siguientes tareas:

- Ignorar dependencias vectoriales en el loop siguiente a la directiva. Esto elimina la dependencia de los datos en los registros vectoriales, el cual es un inhibidor potencial para la vectorización.
- Forzar la generación de código escalar para el loop siguiente a la directiva
- Especificar tipo de precisión aritmética
- Especificación de distribución de trabajo entre los distintos procesadores

Por otro lado, debemos tener en cuenta que estamos trabajando en un entorno multiusuario, donde nuestro proceso compite por los recursos del sistema como tiempo de CPU, canales de I/O, acceso a memoria, espacio de memoria, etc.

Los métodos de optimización en los que nos basamos para mejorar la performance de nuestro programa sugieren **primero mejorar la performance del sistema corriendo en una única CPU antes de analizar la optimización en paralelo**. Una vez finalizada la optimización para el caso de una única CPU, se continúa el proceso para analizar si el código puede ser paralelizado.

Dadas las características de la plataforma de ejecución, y las estructuras de datos y tipos de operaciones del algoritmo (procesamiento de los datos a través de matrices y vectores con gran cantidad de operaciones entre ellos que se realizan dentro de loops), sin lugar a dudas la clave de la optimización está en la ***vectorización*** del código. A continuación estudiaremos en detalle el concepto de vectorización, y explicaremos las técnicas empleadas.

III.2.1.1 Vectorización

La arquitectura vectorial de Cray (es decir, la presencia de registros vectoriales y unidades funcionales vectoriales) introdujo la noción de vectorización en la computación científica. Diseñar o reestructurar programas numéricos para explotar la arquitectura vectorial es un factor de suma importancia en el proceso de optimización de un código ejecutable en esta plataforma.

Definición

- **Es el procesamiento de grupos de números utilizando registros y unidades funcionales vectoriales del hardware**

Procesamiento escalar vs. procesamiento vectorial

Un *escalar* es un número simple u otro ítem almacenado en una palabra de hardware. Éste puede ser un número en punto flotante, un entero o un valor lógico. Los caracteres son también tratados como escalares.
El procesamiento escalar ejecuta operaciones (add, load, store,......) sobre escalares.

Un *vector* es una secuencia indexada de ítems. El procesamiento vectorial ejecuta operaciones (add, load, store,......) sobre vectores usando los vectores del hardware.
Como la vectorización es el concepto más importante en CRAY PVP para poder **optimizar el uso de la CPU**, necesitamos conocer qué expresiones o constructores se vectorizarán en nuestro código. Fundamentalmente, alguna clase de loop explícito o implícito debe existir para que la vectorización pueda ocurrir.
El procesamiento vectorial usa registros vectoriales para operar sobre vectores. El procesamiento escalar usa registros escalares para operar sobre ítems escalares. Sin embargo, la diferencia más importante entre procesamiento escalar y vectorial es que *el procesamiento vectorial cambia el orden de ejecución de las operaciones en el loop.*
Con el procesamiento vectorial, cada línea de código procesa todos los elementos en el registro vectorial (va de 64 a 128 elementos, según el número de serie de CRAY PVP) **antes de seguir con la próxima línea de código**. En consecuencia, el orden en el procesamiento vectorial difiere del orden en el escalar.
Por ejemplo, observemos el siguiente loop:

```
DO I= 1, 3

   L(I) = J(I) + K(I)
   N(I) = L(I) + M(I)

ENDDO
```

Siendo:
J= [2 -4 7]
K=[5 3 8]
M=[4 6 0] , y suponiendo un registro vectorial de dimensión 3

Modo Escalar

Evento	Fortran	Resultado
1	L(1) = J(1) + K(1)	7=2+5
2	N(1) = L(1)+M(1)	11=7+4
3	L(2) = J(2)+K(2)	-1=-4+3
4	N(2) = L(2)+M(2)	5=-1+6
5	L(3) = J(3)+K(3)	15=7+8
6	N(3) = L(3)+M(3)	15=15+0

Modo Vectorial

Evento	Fortran	Resultado
1	L(1) = J(1) + K(1)	7=2+5
1	L(2) = J(2)+K(2)	-1=-4+3
1	L(3) = J(3)+K(3)	15=7+8
2	N(1) = L(1)+M(1)	11=7+4
2	N(2) = L(2)+M(2)	5=-1+6
2	N(3) = L(3)+M(3)	15=15+0

Con procesamiento escalar, el programa calcula L(1), luego N(1), L(2), luego N(2), L(3) y finalmente N(3). Este es el orden de eventos que uno espera que ocurra cuando escribe un loop de este tipo.

Con procesamiento vectorial, el programa calcula todos los valores L(i) antes de calcular los valores N(i). Este orden de eventos está dado por el vector de hardware.

Consecuencias: no siempre el valor final de los elementos del arreglo es el mismo; esto es porque el procesamiento vectorial cambia el orden de las operaciones.

Cualquier situación en la que el procesamiento vectorial de por resultado valores diferentes a los del procesamiento escalar debido a cambios en el orden de las operaciones, es llamado dependencia de datos, dependencia vectorial, o simplemente dependencia.
Los compiladores Cray PVP reconocen dependencias en el código y no generarán una vectorización incorrecta.

Anatomía de un loop vectorial

A continuación describiremos los distintos elementos que componen un loop vectorial. Con distintos colores representamos cada uno de ellos.

```
J = 100
DO I = 2, 100
   J = J - 1
   SCA = A(I) * C(K)
   A(I) = Q(I,K) / K
   B(J,K) = B(J,K) + SCA * J
ENDDO
```

1. **Invariante del loop**

   Es un escalar de sólo lectura (su valor no cambia dentro del loop).

2. **Variable de inducción del loop (LIV)**

   Escalar que es incrementado o decrementado por una expresión invariante en cada pasada del loop. La variable índice del **DO loop o FOR loop** es siempre una variable de inducción.

3. **Vector (candidato)**

   Una referencia a un arreglo cuya dirección apuntada NO es invariante. Si el loop es vectorizable, todos los vectores son procesados con registros vectoriales y unidades funcionales vectoriales.

4. **Expresión vectorizable**

   Una expresión aritmética o lógica que calcula el resultado vectorial. Toda la asignación es considerada como una expresión. Cualquier puntero, variable o referencia a un arreglo en una expresión vectorizable puede ser clasificada como: invariante de loop, variable de inducción de loop, vector o vector temporario.

5. **Vector temporario (candidato)**

   Una variable simple que primero aparece en el lado izquierdo de una asignación de una expresión vectorizable y luego es usada dentro del loop. Si el loop es vectorizable, el vector temporario almacena más de 64 iteraciones de valores dentro del registro vectorial en cada iteración del loop.

6. **Loop vectorizable**

   Es un loop que contiene solamente sentencias armadas con expresiones vectorizables.

Tipos de vectorización

Básicamente existen dos tipos de loops:

➢ Loops completamente vectorizados, en los cuales todas las operaciones son ejecutadas con los vectores del hardware.
➢ Loops parcialmente vectorizados, que contienen un mix de operaciones vectoriales y no vectoriales. Los siguientes son los tipos de loops parcialmente vectorizados:

a. ***Loop reductor,*** que reduce el contenido de un vector a un escalar.
b. ***Loop de búsqueda,*** que busca un valor escalar simple o índice en un vector.

- *Loops completamente vectorizados*

El compilador Fortran genera instrucciones de máquina que ejecutarán un código construido lo más eficientemente posible. En muchos casos, las operaciones dentro de loops pueden ser implementadas directamente con instrucciones vectoriales. Esto es llamado vectorización completa. *Un loop es completamente vectorizable si las siguientes afirmaciones son TODAS verdaderas*:

1. **Al menos un vector está definido por una sentencia de asignación.**
2. **Todas las variables simples definidas en el loop son variables de inducción del loop o vectores temporarios.**
3. **No hay sentencias condicionales que podrían causar la salida del loop.**

a). *Loops de reducción*

Un loop de reducción "condensa" los elementos de un vector a un resultado escalar que, de alguna manera, representa el vector. El ejemplo más común de un loop de reducción es el de la suma de los elementos del vector. Los loops de reducción contienen tanto operaciones escalares como vectoriales.
Ejemplo:

```
SUM=0
DO I=1,200
 SUM=SUM+A(I)
ENDDO
```

En este caso, el compilador setea un vector que suma diferentes grupos de elementos dentro del vector, para producir un registro vectorial de *sumas parciales*. Esas sumas parciales son luego sumadas (con un mix de sumas escalares y vectoriales), para producir la suma total.

Muchos cálculos pueden ser vectorizados como un loop de reducción, si:

-El cálculo es una operación binaria aplicada a una expresión vectorizable
-La operación binaria es conmutativa y asociativa
-Los resultados parciales no requieren ningún orden particular

b). Loop de búsqueda

Es un loop que contiene sentencias condicionales que transfieren el control del programa fuera del loop, antes que éste complete totalmente la cantidad total de iteraciones (sentencia WHILE en Fortran 90).
Estos casos son muy difíciles de optimizar en compilación; es más, en el nivel de optimización de compilación default (*standard*), estos loops no serán vectorizados.

Inhibidores de vectorización

Si un loop contiene alguno de los siguientes inhibidores, no podrá ser optimizado en el "nivel default":

- Cualquier llamado a un subprograma que no es expandido inline por el compilador, *incluyendo Fortran CALL*
- Cualquier sentencia de I/O
- Cualquier condicional Fortran que resulte obsoleto en Fortran 90, como IF's aritméticos, GOTO's computados, y GOTO's asignados
- Cualquiera de las directivas explícitas al compilador que inhiban vectorización
- Saltos no estructurados de programa, como un salto a un loop desde afuera del loop
- **Dependencia de datos** (cuando se producen distintos resultados en modo escalar y vectorial)

Dependencia de datos

Una de las condiciones más comunes que inhiben vectorización es la dependencia de datos. El orden de carga y almacenamiento de las operaciones en modo vectorial hace que se produzcan diferentes resultados que en modo escalar.

Tipos de dependencias:
Existen tres tipos de dependencias:

1. **Data overwritten (almacenamiento antes que carga)**
2. **Result overwritten (dependencia entre dos almacenamientos)**
3. **Result not ready (carga antes que el almacenamiento)**

Ejemplos:
Sean

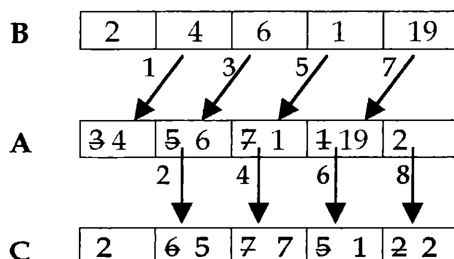
A	3	5	7	1	2
B	2	4	6	1	19
C	2	6	7	5	2

Y supongamos vectores de hardware de tamaño mayor a 5.

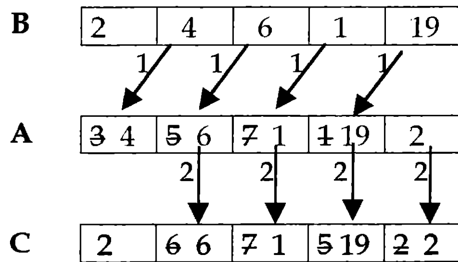
Data overwritten

```
DO I=2,5
 A(I-1) = B(I)
 C(I) = A(I)
ENDDO
```

En MODO ESCALAR:



En <u>MODO VECTORIAL</u>:

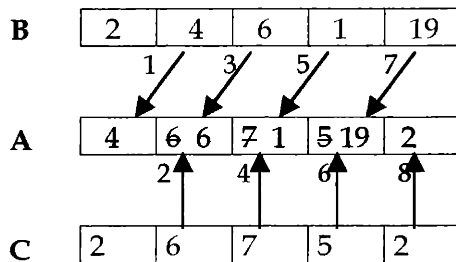


En modo escalar, los "valores anteriores" *A(2)* a *A(5)* son copiados a C antes que el correspondiente nuevo valor sea copiado de B a A.
En modo vectorial, los cuatro primeros elementos son copiados en cadena de B a A. Esto sobreescribe los valores que estaban almacenados anteriormente en A antes que sean copiados de A a C.

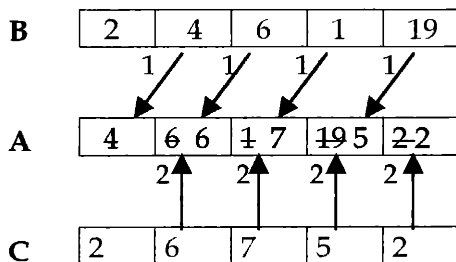
Result overwritten

```
DO I=2,5
  A(I-1) = B(I)
  A(I) = C(I)
ENDDO
```

En <u>MODO ESCALAR</u>:



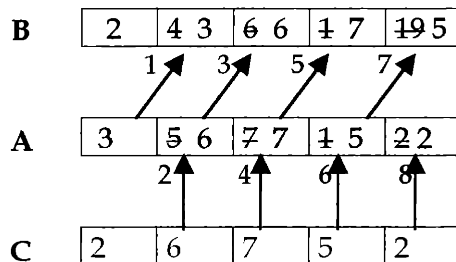
En <u>MODO VECTORIAL</u>:



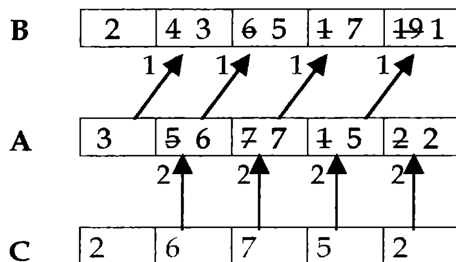
Result not ready

```
DO I=2,5
 B(I) = A(I-1)
 A(I) = C(I)
ENDDO
```

En MODO ESCALAR:



En MODO VECTORIAL:



Encontrando dependencias de datos

Como el compilador usa un sofisticado método para analizar las dependencias en cada loop vectorizable, la manera más fácil y completa de encontrarlas en nuestro código es, justamente, dejar que el compilador las encuentre por nosotros. Sin embargo, a continuación veremos tests de dependencias simples que uno puede usar cuando necesite chequear si existen dependencias en su código.

Cuando las siguientes condiciones son verdaderas, el loop tiene dependencia

1. **El loop contiene al menos dos referencias al mismo arreglo**
2. **Al menos una de las dos referencias está en el lado izquierdo de una asigación**
3. **Los dos conjuntos de valores de índices se overlapean en memoria entre las dos referencias**
4. **Una de las dos referencias, la que debe ser almacenada o cargada primero, tiene un índice que ocurre primero en la secuencia de iteración**

El siguiente ejemplo ilustra el encadenamiento de condiciones hasta llegar a la dependencia de datos. Veremos una a una las condiciones, analizando en cada caso cuándo es aplicable y cuándo no. **Todas** las condiciones deben ser verdaderas para que se produzca la dependencia.

1. Dos referencias a los mismos arreglos

```
   DO 10 I=1,N, 2
10  A(I) = B(I)
```

No es verdadera, porque A y B son vectores distintos.

```
   DO 10 I=1,N, 2
10  B(I) = A(I) + A(I)
```

La condición 1 es verdadera, ya que en la expresión hay dos referencias al mismo arreglo

2. Al menos una de las dos referencias está en el lado izquierdo de una asiganción

```
   DO 10 I=1,N, 2
10  B(I) = A(I) + A(I)
```

No es verdadera porque ambas referencias al vector A están del lado derecho de la asignación

```
   DO 10 I=1,N, 2
10  A(I) = A(I+1)
```

La condición 2 es verdadera, ya que las referencias al vector A se encuentran a la derecha e izquierda de la asignación

3. Los valores de los índices hacen overlap en memoria

```
   DO 10 I=1,N, 2
10  A(I) = A(I+1)
```

Como el loop va de 2 en 2, y el valor a almacenar es *I+1* no se produce overlap.

```
   DO 10 I=1,N
10  A(I) = A(I+1)
```

La condición 3 es verdadera (el loop va de 1 en 1)

4. La referencia "más temprana" tiene un índice anterior con respecto a LIV

```
   DO 10 I=1,N
10  A(I) = A(I+1)
```

la referencia más "temprana"es I, *igual* a LIV

```
   DO 10 I=1,N
10  A(I) = A(I-1)
```

La condición 4 es verdadera: (I-1) es menor que I

Ejemplo:

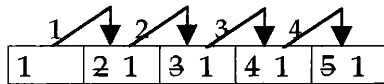
Si N=5 y

A	1	2	3	4	5

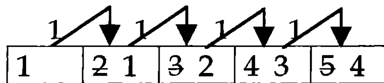
BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Entonces el siguiente esquema muestra la diferencia entre la ejecución secuencial y la vectorial en la expresión recuadrada anteriormente, que contiene dependencia de datos:

-Secuencial o escalar:



-Vectorial:



Para chequear dependencias en un loop con arreglos multidimensionales, se puede aplicar el mismo test que para arreglos unidimensionales.

Resolviendo la dependencia de datos

Los *tests de chequeo de dependencias* nos dan algunas pistas para resolver el problema de dependencia. **En muchos casos, simplemente reescribiendo las sentencias de alguna otra manera logramos solucionarlo.**
Si se puede cambiar el orden de las dependencias sin que cambien los resultados, entonces el loop debería poder ser vectorizado. Las siguientes técnicas son empleadas para cambiar el orden de las referencias:

- Pre-loading
- Switching lines

En loops anidados que contienen arreglos multidimensionales, la siguiente técnica puede resolver cualquier dependencia en el loop interno:

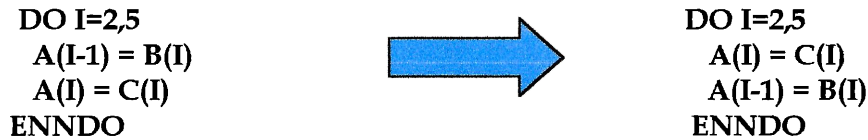
- Reestructuración del anidamiento de los loops.

Estas técnicas son utilizadas naturalmente por el compilador; sin embargo, algunos loops son largos y complejos, y entonces es bueno poder "ayudar" al compilador en la tarea de vectorización.

En el caso de dependencias "data overwritten", la referencia de carga puede ser movida al tope del loop (en un vector temporario). A esta técnica se la llama *pre-loading*:

```
DO I=2,5                       DO I=2,5
 A(I-1) = B(I)        ==>        TEMP= A(I)
 C(I) = A(I)                     A(I-1) = B(I)
ENDDO                            C(I) = TEMP
                               ENDDO
```

Si hay sólo una dependencia en el loop y la dependencia no es una recurrencia, se puede emplear la técnica de "switcheo de líneas": se mueve la línea con la primer referencia (y las que le siguen) después de la línea con la segunda referencia:



Dependencias que se pueden vectorizar

Aunque las dependencias no permiten vectorización, muchas pueden ser vectorizadas parcialmente.

Vectorizando loops que contienen expresiones condicionales

Ahora veremos como el compilador vectoriza loops que contienen expresiones condicionales.

1- Vector mezcla condicional

Para vectorizar loops que contienen estructuras condicionales con múltiples bloques, el compilador Fortran 90 utiliza un vector mezcla condicional (CVM) o simplemente vector mezcla. El siguiente algoritmo muestra cómo el CVM trabaja en una estructura de dos bloques (verdadero-falso), como este:

```
DO I=1,n

  if (a(I).eq.1) then
   X(I) = sqrt(c(I))
  else
   X(I) = sqrt(d(I))
  end if

ENDDO
```

1. Evalúa la condición para formar un **vector de registro de máscara (VM)**. Si el bit es igual a 1, indica que la condición es verdadera; si es 0, la condición es falsa
2. Calcula todos los casos posibles verdaderos en un registro vectorial
3. Calcula todos los casos falsos en otro vector registro
4. Usa VM y los vectores registro verdadero y falso para ejecutar la instrucción vector merge, que combina los valores en los registros verdadero y falso de acuerdo a los valores de VM.

Si la estructura condicional es más complicada (bloques *if else*, *if's anidados*, etc.), el compilador también puede vectorizar. En estos casos complicados, el compilador genera código para múltiples vectores máscara.

Como el vector merge requiere calcular todos los posibles casos verdaderos y todos los falsos, un bloque verdadero-falso como el del ejemplo requiere ejecutar el doble del trabajo necesario.

2- *Compress-index, gather/scatter (CIGS)*

Cuando un loop contiene un único bloque condicional que ejecuta un gran número de operaciones elementales, el compilador genera operaciones CIGS en vez de VM. El siguiente es el algoritmo que sigue el compilador:

1- Setea VM basado en los valores de las expresiones lógicas (igual que en VM)
2- Ejecuta la instrucción compress-index, que almacena en un registro vectorial todas las posiciones verdaderas de VM. Este es el vector índice
3- Junta sólo los valores indicados por el vector índice (casos verdaderos)
4- Calcula los casos verdaderos
5- Almacena (indireccionando con el vector índice) los valores calculados.

Técnicas de vectorización

Un loop es, generalmente, una de las partes del código que con frecuencia debemos tener en cuenta al momento de optimizar, ya que ejecuta grandes cantidades de operaciones repetitivamente. *Solamente los datos que son procesados dentro de un loop pueden usar los registros vectoriales y así aprovechar al máximo la velocidad que Cray ofrece.*

¿Cómo vectorizar un loop escalar ?

Si el código pasa la mayor cantidad de tiempo en loops escalares, se debe tratar de vectorizarlos. A continuación veremos técnicas para evitar tres de los inhibidores de vectorización más comunes:

- Evitar llamados a subprogramas
- Evitar I/O
- Evitar dependencia

¿Cómo evitar un llamado a un subprograma o I/O dentro de un loop?

En un programa muy modular, algunos procedimientos o funciones pueden ejecutar algunos pocos cómputos antes de retornar a la rutina llamadora. Si alguno de estos procedimientos es llamado dentro de un loop, esto está inhibiendo la vectorización, cuando tal vez lo más conveniente es ejecutar esas sentencias directamente dentro del loop. El overhead de llamados a subrutinas es también un tema a tener en cuenta: puede tomar más de 75 períodos de reloj llamar a una subrutina Fortran que no tenga argumentos. *Con un solo argumento, el tiempo se duplica!*

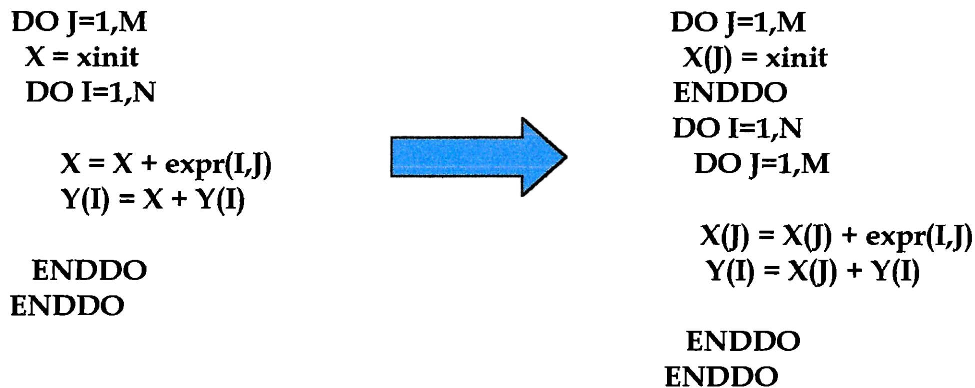
Para subprogramas muy pequeños en loops vectorizables, es mejor ejecutar las sentencias de la subrutina directamente dentro del loop. Las dos siguientes técnicas de programación pueden mejorar la performance en estas condiciones, sin sacrificar la modularidad de nuestro código fuente:

- Expandir el inline del subprograma, o inlining, dentro del loop (se realiza con la instrucción Fortran SUBROUTINE EXPANDED INLINE)
- Poner el loop dentro del subprograma llamado (y luego tratar de vectorizarlo)
- Segmentar el loop en dos o más loops, de los cuales uno sólo contiene llamados a subrutinas o I/O

¿Cómo evitar dependencias de datos?

Para prevenir la dependencia que puede inhibir la vectorización, se pueden emplear las siguientes tres técnicas:

- Convertir un escalar recurrente en un vector



Tener en cuenta que al vectorizar el escalar, cambia el orden de anidamiento de los loops.

- Evitar alias de punteros ambiguos
- Segmentar el loop

Tener en cuenta también para mejorar la performance de un loop:

- Fusionar loops
- Unwind de un loop (en caso de loops anidados, escribir por "extensión" alguno de los loop)

III.2.1.2 Procesamiento paralelo

Hasta ahora, hemos visto como optimizar nuestro código para el mejor aprovechamiento de los recursos, focalizando nuestros esfuerzos en un ambiente con una única CPU. Ahora veremos conceptos de programación paralela, que pueden hacer aun mucho más óptimo nuestro código.

El *procesamiento paralelo* es la técnica de descomposición de una tarea computacional en una serie de subtareas que luego son ejecutadas simultáneamente (con procesamiento secuencial, las tareas son ejecutadas una detrás de la otra). **El objetivo primario que persigue el ejecutar un código simultáneamente sobre múltiples CPUs es hacer más eficiente el uso de los recursos del supercomputador así como disminuir el tiempo de respuesta del sistema.**

Hay diferentes formas en las que el hardware y el sistema operativo posibilitan que una tarea sea ejecutada en paralelo:

❖ **Multiprogramación**

Si un sistema **Cray J90 PVP** tiene sólo un procesador principal, este procesador "switchea" entre los distintos procesos, haciendo parecer que muchas procesos estan ejecutándose al mismo tiempo, cuando en realidad existe un solo proceso activo en la

CPU en cada instante de tiempo. Esto es importante, ya que, por ejemplo, el procesador puede trabajar sobre un proceso mientras otro está esperando que se complete su I/O. Casi todos los sistemas operativos son capaces de brindar esta clase de multiprogramación.

❖ **Multiprocesamiento**

Con más de un procesador principal disponible, los procesadores trabajan concurrentemente sobre diferentes programas

❖ **Multitasking**

Multitasking es un término genérico empleado para describir un escenario en el que más de un procesador trabaja para completar una única tarea. Es la ejecución paralela de dos o más partes de un programa sobre diferentes CPUs; esas partes comparten un área de memoria. Usaremos intercambiablemente los términos multitasking y procesamiento paralelo.

Niveles de paralelismo en el código

Los códigos pueden contener los siguientes niveles de paralelismo:

- De bajo nivel

El nivel más bajo de paralelismo se da entre las instrucciones dentro de la CPU y el pipelining (vector de procesamiento). El compilador para Fortran que provee el ambiente Cray PVP automáticamente detecta y explota esta forma de paralelismo.

- Intermedio

Es aquel que se encuentra entre los elementos de un programa dentro de un módulo. La forma más importante de paralelismo intermedio es el DO......LOOP, porque un gran porcentaje del tiempo de ejecución de un módulo generalmente transcurre dentro de los mismos.
Los DO......LOOP de Fortran operan sobre conjuntos de arreglos, y pueden ser divididos en secciones más pequeñas en las cuales cada procesador aplica el mismo conjunto de operaciones. La separación puede dar a cada CPU arreglos continuos de elementos, puede causar que diferentes iteraciones del loop sean ejecutadas en CPUs diferentes, o dos o más loops independientes pueden ser distribuidos entre las distintas CPUs.

- Alto

El más alto nivel de paralelismo se da entre porciones de programa de diferentes módulos. *Este nivel de paralelismo es muy difícil de lograr en forma automática, ya que es muy difícil que un compilador pueda determinar cuando porciones de subprogramas separados pueden correr concurrentemente de manera segura. En el caso de Cray, es el usuario quien debe determinar cómo se dividen las tareas y hacer explícita la sincronización entre ellas.* Este es el tipo de paralelismo detectado al identificar que el cálculo de las direcciones de proyección del algoritmo pueden ser ejecutadas concurrentemente.

Costos y beneficios del procesamiento paralelo

¿Por qué emplear muchas CPUs para ejecutar un proceso? ¿Cuáles son las ventajas y desventajas?

Beneficios:
El procesamiento paralelo reduce el tiempo de CPU ociosa. Si el trabajo es dividido entre las distintas CPUs, la cantidad de trabajo ejecutado por unidad de tiempo se incrementa.
Desde el punto de vista del sistema operativo, un programa es un consumidor de recursos como ciclos de CPU, palabras de memoria, y canales de I/O. Acelerando la ejecución del código, estos recursos se pueden liberar más rápidamente para que puedan ser consumidos por otros procesos.
Desde el punto de vista del usuario, el principal beneficio de la computación paralela es la mejora en el tiempo de respuesta del sistema, así como la posibilidad de procesar problemas de mayor envergadura.

Costos:
Los costos tienen que ver con el empleo de múltiples CPUs para la resolución de la tarea. El multitasking introduce algo de overhead en la ejecución del programa. Tal vez una de las cosas más importantes es el tiempo que le toma al analista de sistemas analizar el código e identificar el paralelismo.
Los programas que se ejecutan en paralelo requieren más memoria, tienen mayor código, requieren más variables temporales, y espacio adicional en la pila de ejecución.

Cray provee tres maneras de lograr multitasking para Fortran:

- Macrotasking
- Microtasking
- Autotasking

Macrotasking

Se insertan llamados a rutinas en el código fuente Fortran que permiten que 2,3,4,.....,n procesadores trabajen en una rutina o en n rutinas diferentes concurrentemente. El macrotasking es una opción viable de multitasking. Sin embargo, funciona bien sólo en programas con gran granularidad paralela (grandes porciones de código que pueden ser explícitamente particionados en tareas que luego pueden ser ejecutadas en múltiples procesadores simultáneamente); además particionar el programa frecuentemente requiere gran tiempo de trabajo (sobre todo de programación). *El código paralelizable de esta forma no es portable porque se requieren llamadas a librerías específicas de Cray.*

Microtasking

El microtasking fue creado para permitirle a los usuarios tomar ventaja del multitasking dentro de una subrutina (a nivel loop) en un entorno batch. No requiere gran granularidad como el caso anterior; el microtasking hace uso eficiente de los procesadores que están disponibles por cortos períodos de tiempo. Si no hay procesadores extra disponibles, el código se ejecutará correctamente en una única CPU. *Para utilizar microtasking, se debe analizar el programa e insertar directivas en el código fuente Fortran indicando las secciones de código que pueden ser ejecutadas sobre múltiples CPUs.* Un traductor preprocesa el programa, y basado en las directivas, crea un programa que puede utilizar todos los procesadores disponibles en el

momento de la ejecución. Otra ventaja del microtasking (sobre el macrotasking) es que el código es portable.

Autotasking

Es similar al microtasking en el sentido que se puede explotar el paralelismo a nivel loop o bloque del código fuente; sin embargo, existen dos diferencias fundamentales:

1. Autotasking tiene menor overhead que microtasking
2. El análisis requerido para detectar regiones paralelas y la inserción de directivas de Autotasking *pueden* ser ejecutadas por el compilador *sin necesidad de la intervención del programador*

El compilador genera código para ejecutar esas regiones en todas las CPUs que estén disponibles en el tiempo de ejecución. Las principales ventajas del autotasking son:

- Alta granularidad
- Bajo overhead
- Portabilidad del código
- Puede efectuarse automáticamente

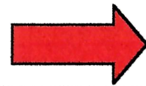
Las implementaciones de micro y autotasking difieren significativamente. Bajo microtasking, cuando una rutina ingresa, todas las CPUs deben ejecutar desde el comienzo hasta el final de la rutina. Bajo autotasking, una CPU master no activa a una CPU esclava hasta que alcanza la región paralela, lo cual provee una reducción del overhead.

Cuando un programa comienza su ejecución, una tarea master creará pilas y contextos para las tareas esclavas, y luego la tarea master comenzará la ejecución de las tareas esclavas. La tarea master realiza la ejecución de todo lo que se encuentra fuera de las regiones paralelas.

Cuando la tarea master *encuentra una directiva que indica ejecución paralela,* activa las CPUs suspendidas. Si las esclavas están disponibles, son administradas por el scheduler y ejecutan el código paralelo para así contribuir a completar la tarea. No hay garantías que hayan esclavos disponibles, con lo cual puede haber casos en los que la tarea master deba efectuar todo el trabajo sola. Cuando el esclavo encuentra el fin de la región paralela, es suspendido. Al final de un trabajo paralelo, la tarea master aguardará que todos los esclavos finalicen la ejecución del código paralelo, para así continuar ejecutando el código restante que se encuentra fuera de dicha región.

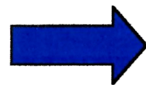
Terminología: anatomía de una rutina autotask

```
Subroutine vecpar (a,q,b,sum,n,m)
Common / c / c
rsum = 0
Write (10,11) 'ingreso a vecpar'
```



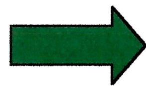
SERIE

```
 do i=1,n
  do j=1,m
    a(i,j) = c * b(i) + tan(q(j))
    rsum(i) = rsum(i) + a(i,j)
  enddo
```



PARTICIONADA

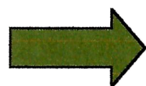
```
 sum = sum + rsum(i)
```



CRÍTICA

```
enddo

tmp= a(i-1,j-1)
```



REDUNDANTE

```
do i=1,n
  b(i) = 4 * tmp * sqrt (b(i)**2)
enddo
```

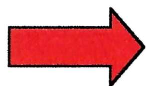
PARTICIONADO

REGIÓN PARALELA

```
write (10,12) 'salgo de rutina vecpar'

return

end
```



SERIE

Región paralela
Es la región de código ejecutada por múltiples procesadores. El código dentro de una región paralela puede ser clasificado en particionado o redundante. Además, pueden existir ciertas porciones de código críticas.

Código particionado
Código dentro de una región paralela en el que múltiples procesadores comparten la tarea que debe ser ejecutada. Cada procesador colabora ejecutando una parte del trabajo. Un conjunto de iteraciones de un loop o una serie secuencial de bloques de código pueden ejecutarse en paralelo si se asegura que no hay computaciones de una iteración o bloque que dependan del valor computado en cualquier otra iteración o bloque.

Código redundante
Código en una región paralela en el cual los procesadores duplican la tarea necesaria, que está disponible en todos los procesadores.

Región crítica
Región de un código paralelo que *debe* ser ejecutada por una única CPU a la vez.

Región serie
Es la región de código que es ejecutada por un solo procesador.

Directivas de procesamiento paralelo

Habilitación del autotasking: el compilador Fortran realiza tanto optimización escalar como vectorización por default. Sin embargo, no realiza autotasking a menos que se le especifique desde una línea de comando especial.
El procesamiento paralelo también puede lograrse manualmente insertando directivas de compilación, ya que hay construcciones paralelas dentro de un programa que el autotasking automático no es capaz de detectar.
Las directivas de procesamiento paralelo en Fortran toman la forma !MIC$, y es seguida por la directiva o cualquier modificación de la misma.

Constructores de regiones paralelas y sus correspondientes directivas

¿Cómo declarar una región paralela?

Las regiones paralelas se declaran en un código encerrándolas entre un conjunto de directivas; se debe declarar explícitamente el comienzo y el fin de la misma. Cuando es declarada, todos los datos referenciados dentro de la región deben tener un "alcance" (scope); esto se logra agregando modificadores a la directiva que define el comienzo de la región.
Dentro de la región, el "código particionado" debe ser marcado con directivas particulares, ya que sino será ejecutado como código redundante. El código particionado puede ser clasificado en las siguientes categorías: loops con iteraciones independientes o bloques secuenciales independientes. También debe marcarse dentro de la región, aquellas secciones de código crítico.

Existen un número mandatorio y optativo de modificadores a la directiva de creación de una región paralela (parallel). Estos modificadores son los siguientes:

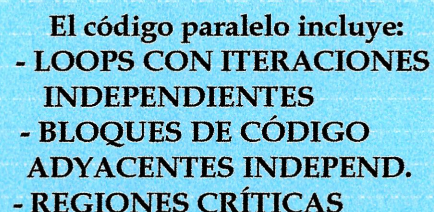
- Data scoping
- Threshold tests
- Work distribution

Declaración de una región paralela:

```
!MIC$      PARALLEL [If (exp)]
!MIC$*     SHARED (var,......)
!MIC$*     PRIVATE (var,......)

    trabajo paralelo

!MIC$      END PARALLEL (var,......)
```



Data scoping

Los modificadores ***shared y private*** son empleados para declarar el alcance de los datos compartidos y privados respectivamente.
Data scoping es el proceso de clasificar los datos accedidos dentro de una región paralela. Es uno de los aspectos más importantes de la conversión de un código secuencial a uno en paralelo.

Todas las variables y objetos dentro de una región paralela deben pertenecer a una de las siguientes categorías:

- SHARED

Un único lugar de memoria es empleado para almacenar el dato, y todos los procesadores lo comparten. Las variables deben declararse como shared si tienen alguna de las siguientes propiedades:

- De sólo lectura
- Lectura y luego escritura
- Indexada por una variable de inducción de loop de un loop particionado

También es recomendable poner arreglos privados de gran dimensión dentro del espacio shared o en un bloque common.

- PRIVATE

Cada procesador ejecutando tiene su propia copia del dato almacenado en su propia pila (cada CPU aloca una copia separada del dato privado). Deben ser inicializadas dentro de la región. Las variables dentro de una región deben ser declaradas privadas si tienen algunas de las siguientes propiedades:

- De sólo escritura
- Escritura y luego lectura
- Variables de inducción de loop

- THRESHOLD TESTS

El modificador ***if(exp)*** es un test opcional. La expresión ***exp*** será usada en tiempo de ejecución para determinar si la región es ejecutada en múltiples o en una única CPU.
Si la expresión es verdadera, la región es ejecutada en múltiples CPUs. Generalmente, este test se emplea para evitar el overhead de llamar a CPUs adicionales cuando la cantidad de trabajo de computación es poca.

Distribución del trabajo

Los loops pueden ser particionados si no hay dependencias. La razón para esta restricción es que cada procesador tomará una iteración (o bloque de iteraciones) y no hay una forma para predecir qué iteración (o bloque de iteraciones) terminará primero (el proceso de selección es no determinístico).
La directiva para particionar un loop dentro de una región en Fortran es simplemente encerrarlo entre las directivas DO PARALLEL y END DO. La directiva puede ser seguida por instrucciones que digan cómo el trabajo debería ser dividido:

- Single

Pasa iteraciones individuales a los procesadores a medida que se encuentran disponibles para realizar el trabajo.

- Numchunks (n)

  Divide las iteraciones en n partes iguales y las pasa a medida que los procesadores se encuentran disponibles.

- Chunksize (n)

  Pasa a cada procesador disponible n iteraciones.

- Guided

  Usa un algoritmo de sheduling propio para pasar grandes porciones del loop al comienzo, y porciones más chicas al final del loop.

Ejemplo

DO I=1,100
trabajo paralelo
ENDDO

Single

CPU 1 :	I=1	I=4	I=8	I=11	....	I=99
CPU 2 :	I=2	I=5	I=9	I=12		I=100
CPU 3 :	I=3	I=6	I=10	I=13		
CPU 4 :		I=7		I=14		

Numchunks (4)

CPU 1 : I=1,25
CPU 2 : I=26,50
CPU 3 : I=51,75
CPU 4 : I=76,100

Chunksize (10)

CPU 1 : I=1,10	I=41,50	I=91,100
CPU 2 : I=11,20	I=51,60	
CPU 3 : I=21,30	....	
CPU 4 : I=31,40		

Guided

CPU 1 : I=1,25	I=68,75	I=100
CPU 2 : I=26,43	....	
CPU 3 : I=44,57		
CPU 4 : I=58,67		

Loops paralelos stand-alone

Cuando la región paralela es sólo un loop y no hay nada fuera de él, hay una forma más rápida de declarar la región paralela, y es a través de la sentencia **DO ALL**. Los modificadores para esta sentencia son los mismos que para data scoping, threshold test y trabajo particionado. Las palabras claves son las mismas que para la directiva **PARALLEL**. El único modificador adicional es **SAVELAST**, que hace que el proceso master ejecute la última iteración del loop.
La sentencia **DO ALL** establece la región paralela y la partición del trabajo.

Se pueden especificar loops stand-alone en Fortran así:

```
!MIC$  DO ALL [if (exp)]
!MIC$1 SHARED (var,......)
!MIC$2 PRIVATE (var,......)
!MIC$3 [single,chunksize(n),numchunks(n)]

 do i=1,n
   ......
 enddo
```

<u>*Particionando bloques secuenciales independientes de código*</u>

Cuando bloques secuenciales de código no tienen dependencia entre ellos, pueden ser ejecutados en paralelo; esto es, ningún valor definido por un bloque es usado por otro. Para lograrlo, se marca el comienzo de cada bloque con una directiva CASE y se termina el conjunto de los bloques a ser ejecutados en paralelo con ENDCASE:

```
!MIC$  PARALLEL if(exp) shared(......)
!MIC$2 PRIVATE (......)
  ......
!MIC$  CASE
  ......
!MIC$  CASE
  ......
!MIC$  ENDCASE
  ......
!MIC$  ENDPARALLEL
```

El trabajo de la **directiva CASE** no debe ser confundido con el uso de la **sentencia case**, ya que la **directiva CASE** dentro de una *región paralela* hace que TODOS los bloques case se ejecuten cuando haya un procesador disponible, mientras que en el caso de la **sentencia case**, <u>sólo uno</u> de los bloques case es ejecutado (aquel bloque para el que su guarda es verdadera).

<u>*Regiones críticas*</u>

Pueden haber ciertas zonas de un código que corre en paralelo que *deban* ser ejecutadas por un solo procesador a la vez. En estos casos, la región crítica de código es encerrada entre las directivas **GUARD** y **ENDGUARD**, que hacen que sólo un procesador esté ejecutando dentro de la misma a la vez.

Hay dos tipos de **GUARD**: numerados y no numerados. El **GUARD** no numerado no utiliza el parámetro opcional **exp** en la directiva. El **GUARD** numerado puede ser usado para permitir que más de un procesador esté ejecutando en la región crítica a la vez (los procesadores están ejecutando en regiones críticas para las que la expresión **exp** es distinta).

```
!MIC$  PARALLEL if(exp) shared(......)
!MIC$2 PRIVATE (......)
  ......
!MIC$  GUARD[exp]
  región crítica
!MIC$  ENDGUARD
  ......
!MIC$  ENDPARALLEL
```

Ejemplo de uso de las directivas de paralelismo

El siguiente es un código sencillo:

```
REAL TOTAL
REAL, DIMENSION(N):: ROWSUM
REAL, DIMENSION(N,N):: MATRIX
INTEGER I,J

DO I=1,N
 DO J=1,N
  ROWSUM(I) = ROWSUM(I) + MATRIX(I,J)
 ENNDO
 TOTAL = TOTAL+ROWSUM(I)
ENDDO
```

Se procesa la matriz **MATRIX**, sumando los elementos de cada una de sus filas y almacenándolos en el arreglo **ROWSUM**; luego se calcula la suma total de las mismas, y se almacena en la variable **TOTAL**.
Para optimizar el código, es deseable primero vectorizar, y, luego, de ser posible, paralelizar. Sabiendo que el compilador Fortran es capaz de vectorizar los loops internos, uno generalmente mirará los loops externos para buscar posibilidades de paralelización.
Si miramos este código, resulta claro que teóricamente la suma individual de cada fila puede ser calculada en paralelo (no hay dependencias).
El constructor (o tipo de región paralela) del código es un loop stand-alone. Una de las primeras tareas que involucra el paralelismo es analizar el alcance de los datos. La región paralela contiene una región crítica, que es la variable que sumariza las filas: **TOTAL**.

Código paralelizado:

```
 REAL TOTAL
  REAL, DIMENSION(N):: ROWSUM
  REAL, DIMENSION(N,N):: MATRIX
  INTEGER I,J
!MIC$ DOALL PRIVATE(I,J) SHARED(N,ROWSUM,MATRIX,TOTAL)
  DO I=1,N
   DO J=1,N
    ROWSUM(I) = ROWSUM(I) + MATRIX(I,J)
```

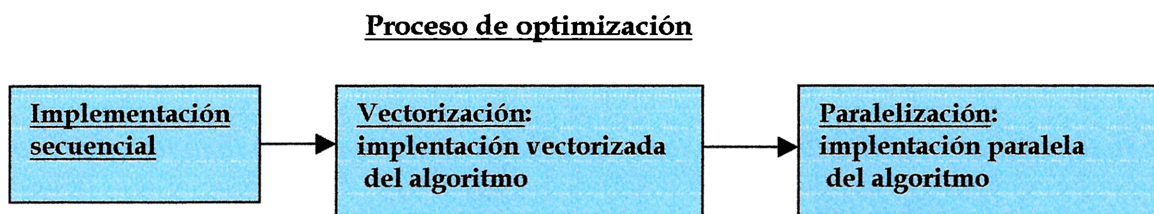
```
  ENDDO
!MIC$ GUARD
  TOTAL = TOTAL+ROWSUM(I)
!MIC$ END GUARD
 ENDDO
```

En este ejemplo, la directiva de autotasking **!MIC$ DOALL** es usada para identificar el paralelismo del loop externo. Esto es, cada iteración del loop externo es independiente de las restantes y puede ser procesada en paralelo en cualquier orden.
Todas las variables tienen alcance **private** o **shared**, y son definidas junto a la directiva **DOALL**.
El ejemplo no tiene un threshold test, aunque podría tener uno si el problema es de una longitud variable: si el tamaño de la matriz puede ser tan pequeño como 5X5 o tan grande como 10000X10000, es deseable usar un threshold test, asegurándonos así que múltiples procesadores serán activados sólo si el tamaño lo merece.
Este ejemplo es interesante en otro aspecto, ya que contiene una región crítica: la variable shared **TOTAL**, guarda la suma de todos los elementos de la matriz. Sin embargo, cada procesador computa la suma de una fila individual. A medida que cada procesador computa la suma de reducción para cada fila, **debe lockear el acceso a la variable compartida TOTAL**, modificarla, y volverla a deslockear: esta región de código es encerrada entonces con la directiva GUARD.

Aplicación de los conceptos de optimización a nuestro código

El proceso de optimización de nuestro código, que siguió como base los preceptos enunciados arriba, abarcó las siguientes etapas:

Proceso de optimización



- **Implementación secuencial**

Al implementar una versión secuencial del algoritmo tuvimos en cuenta el empleo de estructuras de datos eficientes que reflejen el problema a resolver, tratando de ahorrar al máximo espacio de almacenamiento y cantidad de accesos a las estructuras para localizar lógica o físicamente los datos. Las mismas fueron explicadas en detalle en la sección 1.2.1 del presente Capítulo.
Por otro lado, el algoritmo realiza una gran cantidad de operaciones sobre matrices y vectores (fundamentalmente multiplicaciones), con lo cual fue necesario buscar alguna forma de optimizar la implementación de las mismas. En las multiplicaciones esparsas, como vimos, la estructura de representación elegida es la mejor para realizar multiplicaciones matriz por vector a derecha. El resto de las operaciones que involucró la multiplicación de matrices densas completas por vectores (por ejemplo, al chequear si el iterado actual es solución: $\mathbf{A'^{t}.x^{k}}$ fueron implementadas "por columna", *ya que esta es la forma en la que Fortran almacena las matrices.* Sabemos que cualquier computadora almacenará la matriz B como un arreglo bidimensional. Sin embargo, la memoria de la computadora es numerada

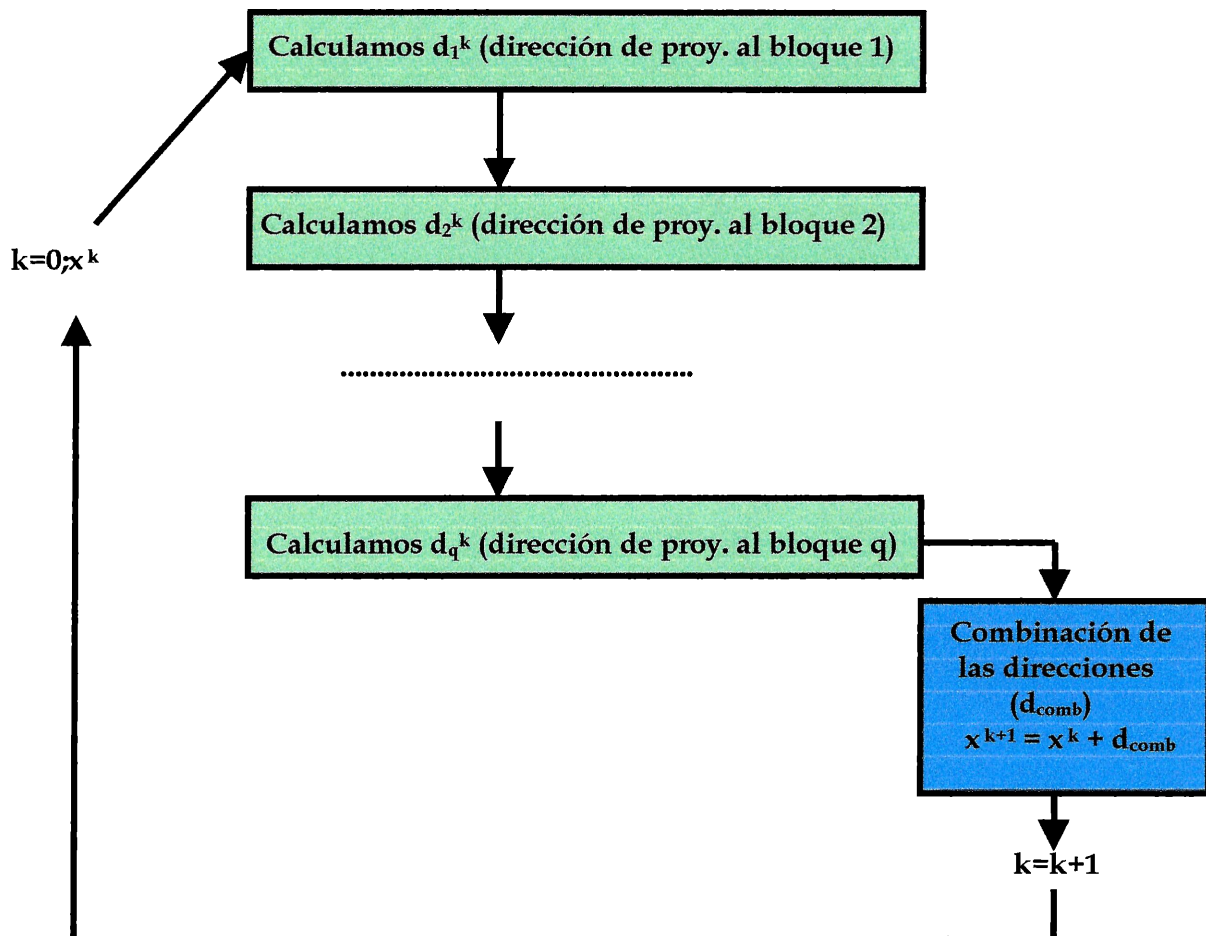
secuencialmente por su dirección, siendo esta unidimensional. Por lo tanto el arreglo bidimensional **A** será, a nivel hardware, almacenado secuencialmente por columnas.

1 **4**	**7** **7**	**13** **90**	**19** x	**25** x
2 **67**	**8** **89**	**14** **90**	**20** x	**26** x
3 **-7**	**9** **100**	**15** **0**	**21** x	**27** x
4 **765**	**10** **0**	**16** **0**	**22** x	**28** x
5 **4**	**11** **90**	**17** **1**	**23** x	**29** x
6 x	**12** x	**18** x	**24** x	**30** x

En este ejemplo, la matriz de dimensiones lógicas 5x3 es dimensionada físicamente de 6x5 (el dimensionamiento físico mayor que el lógico ocurre cuando, por ejemplo, dimensionamos a la matriz en base al peor caso posible). Los valores x son "basura". En azul figura el ordenamiento de la matriz en memoria. Observar que la matriz lógica no ocupa posiciones consecutivas en memoria, y cómo se alocan los valores por columna.

Si bien no se necesita conocer esta información porque la manipulación de los elementos se realiza lógicamente en el lenguaje mediante referenciamientos **A(i,j)**, cuando se implementan programas como el nuestro, en la que la performance de las operaciones es sumamente importante, el hecho de contar con esta información al momento de la implementación es importante, ya que se pueden lograr implementaciones con mejor performance simplemente recorriendo las estructuras en la misma forma en la que son almacenadas (por columna).

- <u>Esquema de búsqueda de una solución secuencial</u>



➢ **Vectorización del código**

La mayoría de los loops trabajan con estructuras vectoriales (por ejemplo para realizar los numerosos cálculos aritméticos sobre las matrices o vectores del problema), con lo cual el análisis del código para su vectorización fue un paso muy importante.
Consistió en un estudio exhaustivo del código en el que se analizaron todos los loops, viendo en cada caso si existían *inhibidores de vectorización* y, en aquellos que correspondiera, adaptamos el código para poder lograr vectorización. Si bien muchos de los loops ya estaban vectorizados en la implementación secuencial, otros los tuvimos que reescribir. <u>Por ejemplo</u>:

- Una normalización de las matrices del sistema implementada inicialmente fue la siguiente:

```
**   Dividimos cada fila de la matriz A y su correspondiente vector de términos
**   independientes b por la norma correspondiente

     do i=1,m
       rnorma=0.d0

       do j=1,n
         rnorma=rnorma+vA(i,j)*vA(i,j)
       enddo

       rnorma=dsqrt(rnorma)

       vb(i)=vb(i)/rnorma

       do j=1,n
         vA(i,j)=vA(i,j)/rnorma
       enddo

     enddo
```

Este loop no está vectorizado

Este cálculo no está vectorizado. Utilizamos la técnica de *conversión de un escalar recurrente en un vector* introduciendo el vector auxiliar ***vnorm*** para lograr la vectorización del mismo.

- La normalización se emplea tanto al comienzo del preprocesamiento para normalizar el sistema a dividir en bloques como en el cálculo de la solución, al normalizar las direcciones hacia cada bloque previo al "filtrado" de las mismas

La siguiente es una versión vectorizada del proceso de normalización:

```
c  Dividimos cada fila de vA y vb por la norma correspondiente
c  loops completamente vectorizados

      do i=1,m
        vnorm(i)=0.d0
      enddo

      do j=1,n
        do i=1,m
         vnorm(i)=vnorm(i)+vA(i,j)*vA(i,j)
        enddo
      enddo

      do i=1,m
        do j=1,n
         vA(i,j)=vA(i,j)/dsqrt(vnorm(i))
        enddo
        vb(i)=vb(i)/dsqrt(vnorm(i))
      enddo
```

LOOPS VECTORIZADOS

Las operaciones matriz-vector o vector-vector están vectorizadas.

El siguiente es un ejemplo, que ilustra la multiplicación de h_{l+1} *por* L_i^{-1}, en el cálculo de v_{h+1}, con las estructuras de datos ya vistas.

```
c*Calculo vh+1*
c*vh+1=-hj+1^t.Lj^-1 (queda en el vector vh)
c*cantfilas es la cantidad de filas del bloque
c*hj+1 se encuentra en el vector vrhj

 do i=1,cantfilas
   vh(i)=-(vrhj(i))
  do  j=1,(i-1)
    vh(j)=vh(j)+((-vrhj(i)))*(mv(inimv(ibloqactual)+(i*i-i)/2+j-2))
   enddo
 enddo
```

LOOP VECTORIZADO

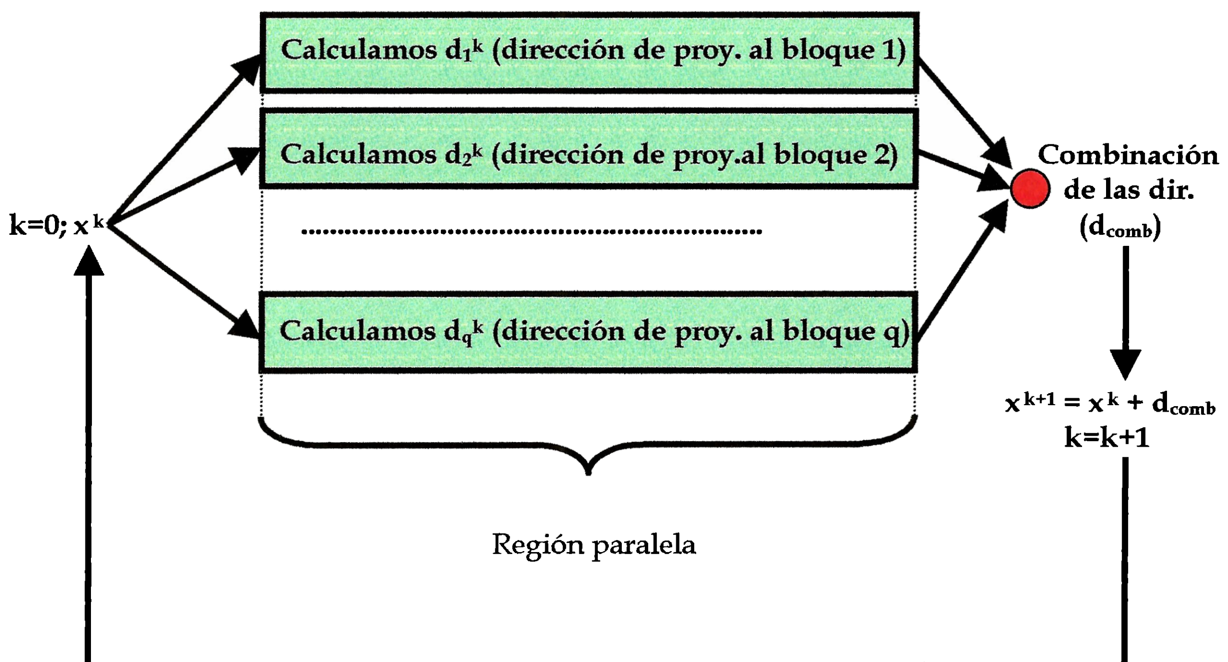
- Ver también como este producto vector-matriz no se recorre exhaustivamente (el loop interno va desde 1 hasta i-1), ya que L_i^{-1} es una matriz triangular.

NOTAR QUE AL LOGRAR UN CÓDIGO VECTORIZADO Y EJECUTARLO EN UN PROCESADOR VECTORIAL (como lo es CRAY J90 PVP) YA ESTAMOS LOGRANDO PARALELISMO (el que se produce a nivel registros vectoriales de hardware, donde se ejecutan en paralelo los loops vectorizados).

➢ Paralelización del código

Como planteamos en la sección 1.1 de este Capítulo, la región paralela en nuestro algoritmo se encuentra dentro del cálculo de la solución, y es aquella en la que se calcula la dirección de proyección hacia cada uno de los bloques. Esquemáticamente:

- **<u>Esquema de búsqueda de una solución paralelizada</u>**



Como

$$\mathbf{d_i^k = (L_i^{-t}.D_i^{-1}.L_i^{-1}).(b_i - A_i^{-t}.x^k)}$$

cada cálculo es absolutamente independiente de los demás (ya que en cada uno intervienen las matrices de la descomposición de Cholesky *propias de cada bloque*, y $\mathbf{x^k}$ es un vector estático invariable para este cálculo), con lo cual no se producen interferencias y cada $\mathbf{d_i^k}$ y $\mathbf{Pd_i^k}$ pueden ser calculadas en paralelo.

El punto de sincronización final de la región es justamente el fin del loop del cálculo de las direcciones de proyección a los bloques.

Empleamos la técnica de ***autotasking***, agregando en el código fuente directivas especiales de distribución del *loop del cálculo de las direcciones de proyección* entre varios procesadores para poder así lograr una ejecución concurrente de esta porción del código.

Para ello empleamos la directiva **!MIC$ DO ALL**. El siguiente es el encabezado de su uso extraído de nuestro programa fuente:

```
!MIC$ DO ALL PRIVATE(icantbloq,imax,n1,iterado,idif,i,vr1,j,
!MIC$*idif,vr,iformula,k,vaordij,rnormaDi,rcalculo,rnormaD,vdmezcla2)
!MIC$*SHARED(ivbloq,rsaord,ijaord,vX,vBord,ivinl,vl,ivind,vd,vA,iteracion,
!MIC$*vnormaDi,vdmezcla)

    do iterado=1,icantbloq
  ....................
```

Sin lugar a dudas el paso más importante luego de la detección de la región paralela y comprobación de su implementación posible (por no haber interferencias o dependencias), fue realizar un análisis exhaustivo del alcance de las variables dentro de la región paralela, para determinar su acceso privado (**PRIVATE**) o compartido (**SHARED**).

III.3 Verificación

Existen métodos formales para probar propiedades de un algoritmo que se pueden emplear para demostrar que el mismo es correcto. Aunque no vamos a presentar en este trabajo una demostración formal detallada de la correctitud del programa implementado, mencionaremos algunos aspectos del proceso de verificación.

Una *afirmación* es una declaración acerca de una condición particular en un punto determinado del programa. Las pre y postcondiciones son simplemente *afirmaciones* acerca de condiciones que se deben verificar al comienzo y fin de los módulos. Un *invariante* es una condición que es siempre verdadera en un punto del algoritmo. El *invariante de un loop* es una condición que es verdadera antes y después de cada ejecución del loop en el algoritmo.

Probar que un programa es correcto es **similar a probar cualquier teorema**: para probar que un módulo es correcto, se debería comenzar con las precondiciones (axiomas) y demostrar que los pasos de la implementación del algoritmo conducen a las postcondiciones. Para hacerlo, se debería considerar cada paso en el algoritmo y probar que una afirmación antes de la ejecución del mismo conduce a una afirmación particular posterior a la ejecución del mismo. Hay maneras formales de hacer estas demostraciones para los constructores *if, loops y asignaciones del programa.*

Una vez probada que una sentencia individual es correcta, se necesita juntarlas para probar que las secuencias de sentencias, luego los módulos, y finalmente el programa son correctos. Llegar a una *demostración completa formal* del algoritmo excede el objetivo de este trabajo. Si

bien no desarrollamos una prueba formal, como vimos en el proceso de paralelización del código, hemos detectado en base a la fórmula matemática de proyección al bloque que *no se producen interferencias* dentro de la región paralela.

En base a las ***propiedades matemáticas*** de la división en bloques, *implementamos las siguientes pruebas*:

Prueba del preprocesamiento: verificación de la división en bloques

Implementamos esta prueba una vez finalizado el preprocesamiento, con el objetivo de verificar que la descomposición de Cholesky de los bloques en los que quedó dividido el sistema original A.x=b eran correctos. La prueba se sustenta en la siguiente propiedad:

Para toda matriz M inversible, $M.M^{-1} = I$, con lo cual

$(A_i^t. A_i).(A_i^t. A_i)^{-1} = I$

Sabemos que $(A_i^t. A_i)$ es simétrica, y su descomposición de Cholesky es $L_i.D_i.L_i^t$
También vimos en el Capítulo II que es "facil" obtener L_i^{-1} y D_i^{-1} en base a L_i y D_i; es más, estas matrices las calculamos y almacenamos en el preprocesamiento para emplearlas luego en el cálculo de la solución, con lo cual son conocidas. Entonces es cierto que

$(A_i^t. A_i).(L_i.D_i.L_i^t)^{-1} = I$;distribuyendo la inversa tenemos que
$(A_i^t. A_i).(L_i^{-t}. D_i^{-1}. L_i^{-1}) = I$

A continuación presentamos el pseudocódigo que realiza la verificación de la división en bloques del sistema original:

Para todo bloque i en que quedó dividido A hacer

Calcular $(A_i^t. A_i) = X_1$
Calcular $(L_i^{-t}. D_i^{-1}. L_i^{-1}) = X_2$
Calcular $X_1 . X_2 = X_3$
If $X_3 /\approx I$ entonces 'error en el proceso de división'; terminar prueba

fin

En el *Anexo II, sección Prueba de división en bloques* del presente texto, mostramos el output de las pruebas numéricas de la división en bloques para una matriz de Hilbert
($H_{ij} = 1/i+j-1$) de 10x10, con bloques de tamaño máximo igual a 4. Si bien este ejemplo tiene un carácter meramente ilustrativo, es interesante porque este tipo de matrices son "muy sensibles", con lo cual los bloques deben estar bien divididos (es decir, bien condicionados), si queremos llegar a una buena resolución del sistema.

Una vez implementado el algoritmo que halla la solución, procedimos a efectuar la siguiente prueba:

Prueba general del algoritmo:

Creamos sistemas de ecuaciones con matrices cuadradas de rango completo (es decir, la matriz es inversible) en los que definimos b_i así: $b_i = \sum_{j=1}^{n} a_{ij}$ con el fin de testear los resultados obtenidos.

Bajo estas condiciones, estamos seguros que el sistema tiene *única solución* $x^*=(1,1,......,1)$.

Partiendo desde distintos puntos iniciales procedimos a efectuar corridas del algoritmo para diferentes sistemas que cumplen la condición del párrafo anterior, verificando en cada caso la solución obtenida cuando se satisfacía el criterio de parada (*debía* ser aproximadamente (1,1,.......,1)), e imprimimos el residuo correspondiente, comprobando que el error obtenido era pequeño.

En el *Anexo II, sección Soluciones del algoritmo acelerado de proyección en bloques* del presente texto, mostramos un ejemplo de esta prueba para una matriz de Hilbert de 10x10, con tamaño máximo de filas por bloque igual a 4, armada de tal manera que $\sum a_{i\,j}=b_i$, y partiendo del punto inicial (0,0,......,0).

En estos problemas en los que x^* se conocía corroboramos *a lo largo del proceso iterativo* que las relaciones (2.15) y (2.19) se verificaban en nuestra implementación. Asimismo la relación $\|x^{k+1} - x^*\|^2 = \|x^k - x^*\|^2 - \|d^k\|$ también fue verificada en estos problemas.

CAPÍTULO IV
Resultados

El proceso de optimización de la implementación del algoritmo acelerado de proyección en bloques, que luego de su análisis y corroboración de propiedades paralelas nos condujo a una implementación en donde aprovechar el *cálculo de las proyecciones en paralelo,* tuvo como objetivo la disminución del tiempo de respuesta del algoritmo. *El objetivo del proceso de optimización para nosotros fue lograr demostrar que los tiempos de respuesta a lo largo del proceso disminuyen.* El querer que los tiempos de respuesta sean los más bajos posibles, se fundamenta en la rapidez de resolución que requieren los problemas para los que resolver sistemas de ecuaciones es el modelo matemático. Como vimos en el Capítulo I, en aplicaciones médicas, por ejemplo, el tiempo de respuesta es un factor crítico.

No debemos perder de vista que el proceso de optimización del algoritmo no disminuye cantidad de iteraciones para llegar a la solución, SÍ el tiempo en calcularlas. Podemos establecer la siguiente analogía: supongamos que queremos viajar de Bariloche a Buenos Aires. Tenemos distintos medios de transporte para realizar el viaje. Aunque la distancia a recorrer es la misma, un viaje en avión resultará mucho más rápido que un viaje efectuado en auto. A estos efectos, los medios de transporte representan para nosotros los distintos programas implementados en el proceso de optimización, y la cantidad de kilómetros a recorrer el número de iteraciones hasta llegar a una solución aceptable. Si tuviésemos que trasladar un enfermo grave de Bariloche a Buenos Aires, ¿qué medio de transporte elegiríamos? Sin lugar a dudas, aunque sabemos que el viajar en auto garantiza que lleguemos a destino, un viaje en avión parece ser la mejor opción. Las aplicaciones de "misión crítica" pueden verse entonces como los enfermos a los que hay que trasladar a destino lo más rápidamente posible.

En este Capítulo veremos la demostración numérica que efectivamente corrobora cómo la aplicación de técnicas de optimización del código disminuyen los tiempos de respuesta. Veremos los resultados comparativos entre las distintas implementaciones del algoritmo en el proceso de optimización.

IV.1 Problemas de testeo

IV.1.1 Casos esparsos

Los problemas esparsos usados para testear el algoritmo son los propuestos por **R. Bramley** y **Sameh** en *"Row Proyection Methods for Large Nonsymmetric Linear Systems"*, **SIAM Journal Sci. Stat. Compt, vol 13,1992**: los problemas P1-P6 fueron obtenidos de la discretización, usando diferencias centrales, de las ecuaciones diferenciales parciales elípticas de la forma $au_{xx} + bu_{yy} + cu_{zz} + du_{x} + eu_{y} + fu_{z} + gu = F$, donde a-g son funciones de (x,y,z) y el dominio es el cubo unitario [0,1]x[0,1]x[0,1].

Se asumieron las condiciones de borde de Dirichlet. Las soluciones de estas ecuaciones diferenciales parciales son conocidas, pudiendo así chequear de modo experimental los errores en el cómputo de las soluciones.

Cuando discretizamos n_1 puntos en cada dirección, el sistema no simétrico resultante es de orden n_1^3 , y por lo tanto el tamaño del sistema crece muy rápidamente cuando la grilla es refinada.

Estos problemas tienen aplicaciones en *mecánica estructural y de fluidos.*
Los coeficientes de las ecuaciones diferenciales fueron elegidos de tal manera que la matriz A tenga o no propiedades tales como estar bien condicionada, ser diagonal dominante, etc.

P1 : $\Delta u + 1000\, u_{xx} = F$
Con solución $u(x,y,z) = xyz(1-x)(1-y)(1-z)$

P2 : $\Delta u + 1000\, e^{xyz}\,(u_x + u_y - u_z) = F$
Con solución $u(x,y,z) = x + y + z$

P3 : $\Delta u + 1000\, xu_x - yu_y + zu_z + 100(x+y+z)(u/xyz) = F$
Con solución $u(x,y,z) = e^{xyz}\,\text{sen}(\pi x).\,\text{sen}(\pi y).\,\text{sen}(\pi z)$

P4 : $\Delta u - 10^5.x^2\,(u_x + u_y + u_z) = F$
Con solución $u(x,y,z) = e^{xyz}\,\text{sen}(\pi x).\,\text{sen}(\pi y).\,\text{sen}(\pi z)$

P5 : $\Delta u - 1000(1+x^2)u_x + 100(u_y + u_z) = F$
Con solución $u(x,y,z) = e^{xyz}\,\text{sen}(\pi x).\,\text{sen}(\pi y).\,\text{sen}(\pi z)$

P6 : $\Delta u - 1000((1-2x)u_x + (1-2y)u_y + (1-2z)\,u_z) = F$
Con solución $u(x,y,z) = e^{xyz}\,\text{sen}(\pi x).\,\text{sen}(\pi y).\,\text{sen}(\pi z)$

IV.1.2 Caso denso

P7 : $A^t.x = b$, A matriz de Hilbert, donde $a_{ij}=1/(i + j - 1)$, con **b** elegido de tal manera que el sistema tenga única solución $x^*=(1,1,1,1,1,1,......,1)$.
El número de condición de las matrices de Hilbert es del orden de $\mathbf{e^{3.5*n}}$. Este caso de prueba es relevante por la sensibilidad numérica de la matriz (como se ve, se trata de una matriz muy mal condicionada).

IV.2 Resultados numéricos

Las referencias de abajo representan las diferentes implementaciones (reflejando de arriba hacia abajo el proceso de optimización) del ***algoritmo acelerado de proyección en bloques***.
Plataforma de ejecución: Cray J90 PVP (procesador vectorial), de características técnicas descriptas en el Capítulo III.

<u>Referencias:</u>
AE1: implementación esparsa secuencial
AE2: implementación esparsa vectorizada
AE3: implementación esparsa paralela

<u>Referencias:</u>
AD1: implementación densa secuencial
AD2: implementación densa vectorizada
AD3: implementación densa paralela

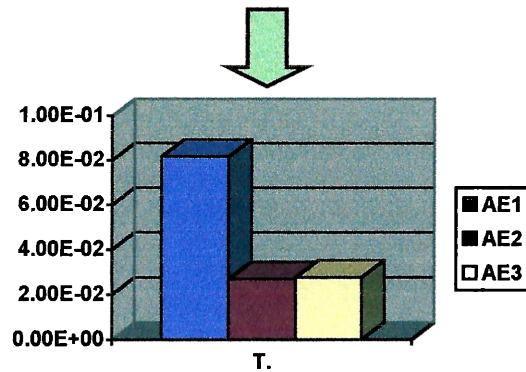
Los tiempos de respuesta fueron medidos en segundos.

IV.2.1 Resultados numéricos para P1-P6

cantidad de filas = 6^3 = **216**
cantidad de columnas = 6^3 = **216**
cant. máxima de filas por bloque = 6^2 = **36**
tolerancia de aceptación de filas= 10^4
tolerancia de aceptación de dirección= 10^4
Criterio de parada: $rkn^2 < 10^9$ en los problemas impares, y $rkn^2 < 216^2.10^9$ en los problemas pares

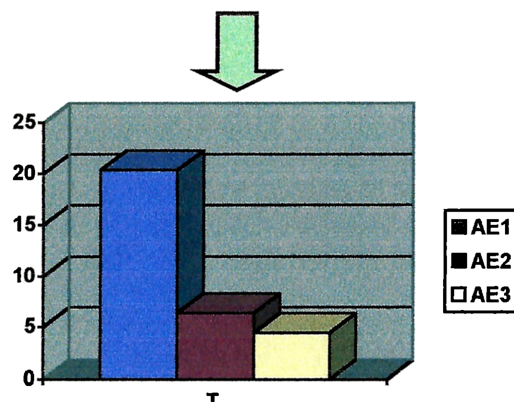
P1

	Tiempo de Preprocesamiento	**Tiempo de procesamiento**	Residuo	Cant. de iteraciones
AE1	0.2258741999999998	**8.162150999999795E-2**	3.650730561349256E-6	2
AE2	7.113542999999999E-2	**2.696339999999963E-2**	3.650730561339688E-6	2
AE3	0.1106527700000015	**2.756076999999912E-2**	3.650730561019849E-6	2



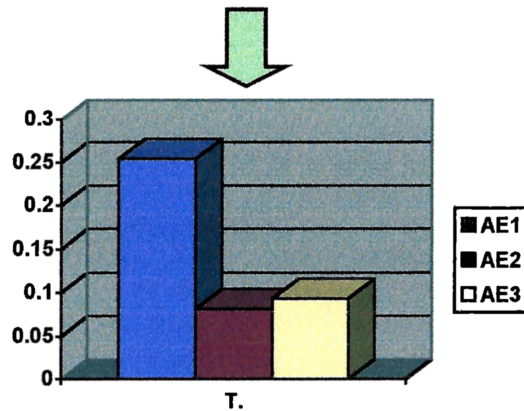
P2

	Tiempo de Preprocesamiento	**Tiempo de procesamiento**	Residuo	Cant. de iteraciones
AE1	0.224991450000001	**20.33661013000005**	6.623369093123554E-3	469
AE2	6.988934999999862E-2	**6.359561150000019**	6.623369071169088E-3	469
AE3	9.490633000000237E-2	**4.471715190000026**	6.623369189795808E-3	469



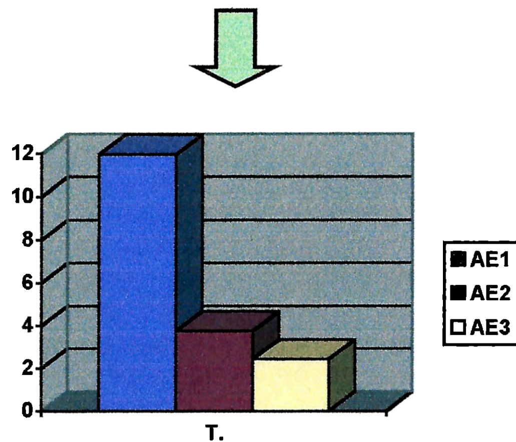
P3

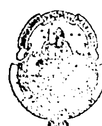
	Tiempo de Preprocesamiento	**Tiempo de procesamiento**	Residuo	Cant. de iteraciones
AE1	0.227577359999998	**0.2540810100000002**	2.96429471946688E-5	6
AE2	7.142625000000002E-2	**8.133323999999993E-2**	2.964294719459345E-5	6
AE3	9.484297000000019E-2	**9.287547000000095E-2**	2.964294719576569E-5	6



P4

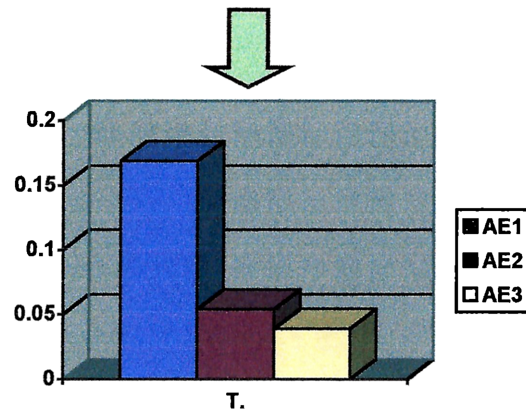
	Tiempo de Preprocesamiento	**Tiempo de procesamiento**	Residuo	Cant. de iteraciones
AE1	0.2276488800000038	**11.94694347000006**	6.295306014538537E-3	275
AE2	7.017674999999989E-2	**3.750207619999998**	6.299833570528462E-3	275
AE3	9.073495000000164E-2	**2.442411610000008**	6.808090566059771E-3	275




BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

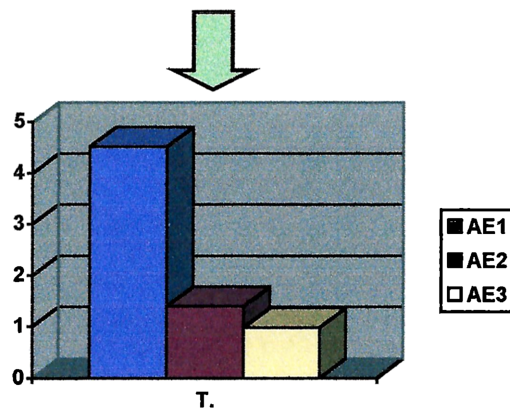
P5

	Tiempo de Preprocesamiento	**Tiempo de procesamiento**	Residuo	Cant. de iteraciones
AE1	0.2250904300000016	**0.1690693799999963**	1.103077708348131E-5	4
AE2	6.930963000000112E-2	**5.397991999999973E-2**	1.103077708507275E-5	4
AE3	9.441181999999948E-2	**3.909988000000019E-2**	1.103077708963773E-5	4



P6

	Tiempo de Preprocesamiento	**Tiempo de procesamiento**	Residuo	Cant. de iteraciones
AE1	0.2283503399999987	**4.517454560000033**	6.610398622458935E-3	104
AE2	6.915305999999965E-2	**1.397793240000006**	6.610398692955932E-3	104
AE3	9.294310000000116E-2	**0.986619740000009**	6.610399656003546E-3	104



En TODOS los ejemplos presentados anteriormente (de dimensión "pequeña", con partición en seis bloques de 36 filas en cada caso) se puede apreciar la mejora en los tiempos de respuesta del cálculo de la solución (procesamiento) para el programa vectorizado ejecutado en plataforma vectorial por sobre la versión secuencial ejecutada inhibiendo vectorización (simulamos así una plataforma escalar). No se aprecia una mejora de la versión paralela por sobre la vectorial *en todos los casos* debido al overhead que produce la primera en sistemas de ecuaciones pequeños en los que hay pocos bloques y se llega a una solución satisfactoria en

pocas iteraciones. Es decir, el tiempo de distribución de los cálculos de las proyecciones hacia los bloques entre varios procesadores en los problemas P1 y P3, y su posterior sincronización acompañado con las pocas iteraciones que tarda el algoritmo en encontrar una solución aceptable, es mayor que los tiempos de corrida de la versión vectorial ejecutada en un solo procesador. Esta situación es perfectamente razonable.

En cuanto al preprocesamiento, también se verifica que en TODOS los casos el tiempo de ejecución de la versión secuencial es mayor que la versión vectorizada, tal cual es esperado. Las diferencias en los tiempos de preprocesamiento entre la versión vectorizada y la paralela (teóricamente deberían ser iguales, ya que en el preprocesamiento no definimos regiones paralelas en forma explícita como en el procesamiento para el cálculo de las direcciones, ya que este es inherentemente secuencial), se produce porque el compilador de **CRAY J90 PVP** para Fortran 90, al observar que tiene múltiples procesadores disponibles, genera un código en el que, además de reconocer y aplicar el paralelismo definido por el usuario, realiza su propio análisis de dependencias, tratando de distribuir el código tanto como sea posible. Esta "paralelización implícita", que no podemos controlar (si la inhibimos no se detectaría la región paralela definida por nosotros), en casos de sistemas de ecuaciones pequeños, produce overhead, haciendo que los tiempos de preprocesamiento de la versión vectorizada sean mejores. Es de esperar que en sistemas de dimensión mucho mayor, la versión de preprocesamiento con paralelismo implícito vaya mejorando hasta ser mejor que la vectorizada.

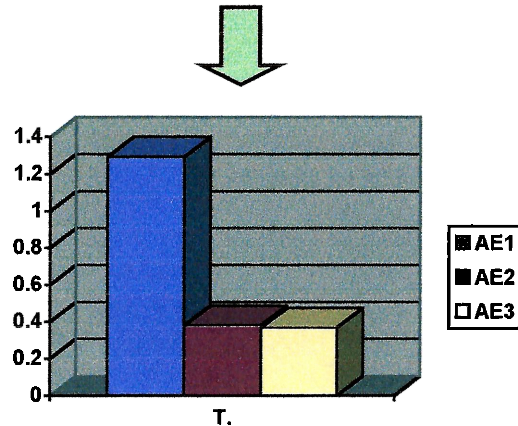
Limitaciones de tipo técnico ajenas a nosotros o al programa (el usuario asignado tiene privilegios acotados) no permitieron que pudiésemos chequear el algoritmo para casos de matrices de dimensiones mayores a los que se verán a continuación.

cantidad de filas = 9^3 = **729**
cantidad de columnas = 9^3 = **729**
cant. máxima de filas por bloque = 9^2 = **81**
tolerancia de aceptación de filas= 10^4
tolerancia de aceptación de dirección= 10^4
Criterio de parada: $rkn^2 < 10^9$

El objeto de las siguientes pruebas es verificar como la brecha del tiempo de ejecución del procesamiento entre las versiones vectorizada y paralela disminuye al aumentar el tamaño del sistema. Nos interesan particularmente aquellos casos de prueba anteriores en los que el tiempo de ejecución de la versión paralela es mayor que la vectorizada (P1 y P3). Verificamos así cómo el overhead que produce la ejecución paralela NO influye en los tiempos de respuesta; por el contrario, la versión paralela es mejor que la secuencial.

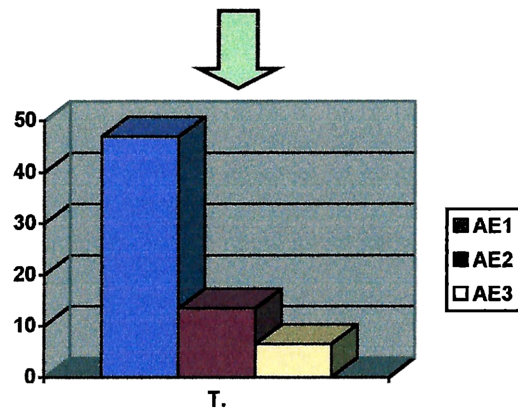
P1

	Tiempo de Preprocesamiento	**Tiempo de procesamiento**	Residuo	Cant. de iteraciones
AE1	3.233202900000038	**1.292343519999974**	5.32308994594062E-4	5
AE2	0.7421915500000011	**0.3803800599999931**	5.323089945939163E-4	5
AE3	0.918095470000003	**0.3677223699999956**	3.062004710381593E-5	5



P3

	Tiempo de Preprocesamiento	**Tiempo de procesamiento**	Residuo	Cant. de iteraciones
AE1	3.276762700000006	**46.90989494000019**	3.1379635019 10847E-5	177
AE2	0.7420988300000033	**13.45798095999999**	3.13796350025475E-5	177
AE3	0.9129413800000066	**6.492956970000023**	3.137963501177384E-5	177



- El siguiente es el resultado de un reporte estadístico (llamado hpm) que provee Cray. *Lo ejecutamos sobre este ejemplo para ilustrar las diferencias que se producen a nivel uso de CPU global, y cantidad de millones de operaciones resueltas por segundo en la ejecución de las diferentes implementaciones:*

Para AE1:

CPU seconds : 59.14	**CP executing : 5914203737**
Floating adds/sec : 1.34M	**F.P. adds : 79520451**
Floating multiplies/sec : 0.89M	**F.P. multiplies : 52715345**
Floating reciprocal/sec : 0.08M	**F.P. reciprocals : 4685735**
I/O mem. references/sec : 4.14M	**I/O references : 245011197**
CPU mem. references/sec : 7.49M	**CPU references : 442956945**
Floating ops/CPU second : 2.32M	

Para AE2:

CPU seconds : 18.40	**CP executing: 1840198750**
Floating adds/sec : 2.12M	**F.P. adds : 38921471**
Floating multiplies/sec : 1.88M	**F.P. multiplies : 34556780**
Floating reciprocal/sec : 0.00M	**F.P. reciprocals : 39140**
I/O mem. references/sec : 1.56M	**I/O references : 28760994**
CPU mem. references/sec : 4.95M	**CPU references : 91095050**
Floating ops/CPU second : 4.00M	

Para AE3:

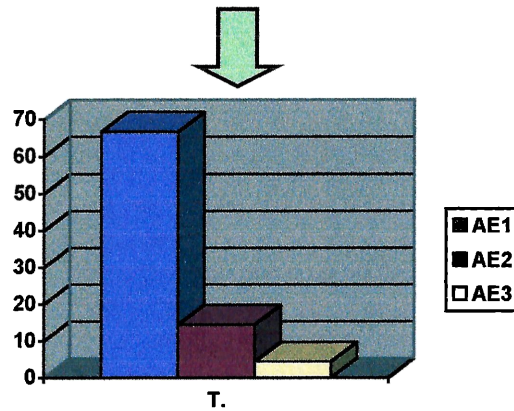
CPU seconds : 10.28	**CP executing: 1027902325**
Floating adds/sec : 5.71M	**F.P. adds : 58742221**
Floating multiplies/sec : 3.42M	**F.P. multiplies : 35164472**
Floating reciprocal/sec : 0.00M	**F.P. reciprocals : 39141**
I/O mem. references/sec : 0.00M	**I/O references : 2755**
CPU mem. references/sec :14.74M	**CPU references :151511644**
Floating ops/CPU second : 9.14M	

El siguiente caso de prueba es significativo: aquí podemos apreciar la potencia de ejecución del procesamiento paralelo por sobre la versión vectorizada: consideramos las mismas dimensiones que el caso anterior para P1, pero tomamos 5 filas como tamaño máximo por bloque (la matriz del caso P1 está bien condicionada, con lo cual tenemos *146 bloques*). Se puede observar claramente cómo los tiempos de la versión paralela mejoran respecto de la versión vectorizada.

cantidad de filas = 9^3 = **729**
cantidad de columnas = 9^3 = **729**
cant. máxima de filas por bloque = **5**
tolerancia de aceptación de filas= 10^4
tolerancia de aceptación de dirección= 10^4
Criterio de parada: $rkn^2 < 729^2.10^9$

P1

	Tiempo de Preprocesamiento	**Tiempo de procesamiento**	Residuo	Cant. de iteraciones
AE1	0.3223722899999757	**66.68015796000054**	1.201450578339496E-2	4
AE2	9.47550599999829E-2	**14.48440661999996**	1.201450578339496E-2	4
AE3	0.1004467199999937	**4.328407090000013**	1.201450578340135E-2	4



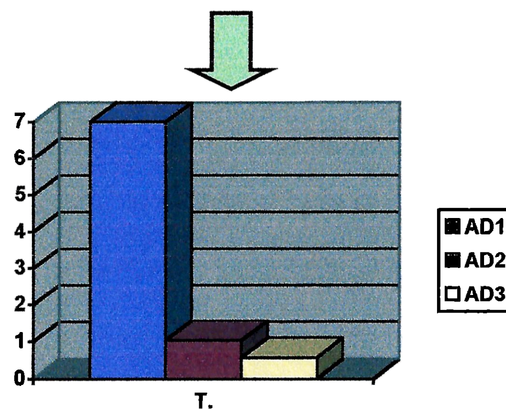
Observación: la diferencia de tiempos de procesamiento respecto de P1 de 216x216 (en este caso el sistema original queda dividido en 9 bloques) se debe a que luego del cálculo de las proyecciones se resuelve, en cada iteración del algoritmo, *un sistema de 9x9 en el primer caso y de 146x146 en el segundo.*

IV.2.2 Resultados numéricos para P7

cantidad de filas = **500**
cantidad de columnas = **500**
cant. máxima de filas por bloque = **10**
tolerancia de aceptación de filas= 10^4
tolerancia de aceptación de dirección= 10^4
Criterio de parada: $rkn^2 < 10^9$

P7

	Tiempo de Preprocesamiento	**Tiempo de procesamiento**	Residuo	Cant. de iteraciones
AD1	95.14748184000064	**6.989725269999781**	6.605323651539835E-6	2
AD2	16.42269054000008	**1.029441150000025**	6.750477516419961E-6	2
AD3	17.15239832000009	**0.5677620699999579**	6.974796153188871E-6	2



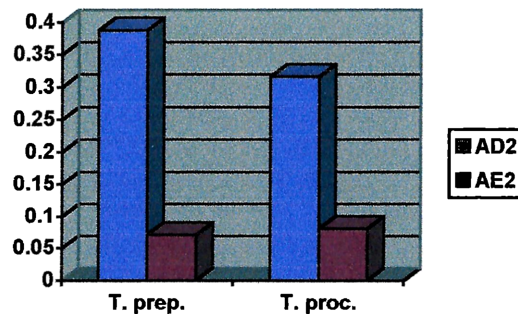
En este caso denso con A matriz muy mal condicionada (Hilbert), se puede ver la mejora en los tiempos de la versión vectorizada por sobre la secuencial, y de la paralela por sobre la vectorizada. Obsérvese además la cantidad de tiempo de preprocesamiento (armado de bloques bien condicionados) en sistemas que están mal condicionados.

IV.3 Análisis comparativo de estructuras

El objetivo de esta prueba es corroborar la influencia de la elección de una estructura de datos adecuada para resolver el problema. Para ello ejecutamos el ejemplo P3 de Sameh con la implementación densa, y comparamos tiempos de respuesta contra una implementación empleando estructuras de datos para representar matrices esparsas.

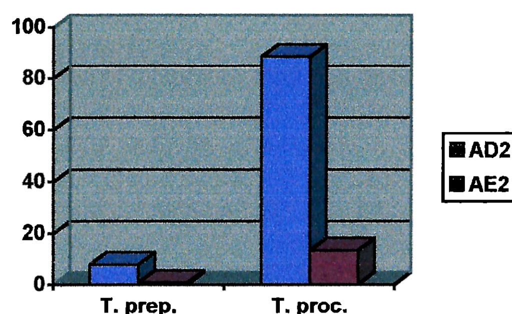
P3 (dimensión:216x216 con bloques de tamaño 36)

	Tiempo de Preprocesamiento	Tiempo de procesamiento	Residuo	Cant. de iteraciones
AD2	**0.3875558700000017**	**0.316119920000002**	2.96429471945091E-5	6
AE2	**7.142625000000002E-2**	**8.133323999999993E-2**	2.964294719459345E-5	6



P3 (dimensión:729x729 con bloques de tamaño 81)

	Tiempo de Preprocesamiento	Tiempo de procesamiento	Residuo	Cant. de iteraciones
AD2	**7.843262910000021**	**88.49830455000028**	3.137963498797821E-5	117
AE2	**0.7420988300000033**	**13.45798095999999**	3.13796350025475E-5	117



IV.4 Conclusiones

Podemos establecer conclusiones sobre tres ejes fundamentales:

- **La importancia del uso de estructuras de datos adecuadas**

  Los experimentos computacionales de testeo muestran una mejora considerable en los tiempos de ejecución tanto en la división en bloques como en el cálculo de la solución del algoritmo, *cuando los sistemas son ralos* y empleamos estructuras de datos especiales para la representación de matrices esparsas (en comparación con los tiempos de ejecución de una implementación del algoritmo para "matrices densas"), corroborando que una elección de estructuras de datos adecuada beneficia la performance global del algoritmo en estos casos.
  En tal sentido, el *"modo de almacenamiento esparso indexado por filas"* adoptado es el más adecuado para el algoritmo de proyección en bloques, ya que realiza en forma muy eficiente las operaciones matriz por vector a derecha, y además requiere espacio de almacenamiento para a lo sumo el doble de elementos distintos de 0 de la matriz, mientras que otros métodos pueden requerir de 3 a 5 veces este valor.
  Este análisis es independiente de la plataforma de ejecución del algoritmo.
  Fue sumamente importante corroborar esta mejora, ya que sistemas esparsos como los vistos en el Capítulo I fueron la fuente de motivación práctica de este trabajo.

- **La importancia de una plataforma de corrida vectorial (vs. procesamiento escalar) para la ejecución de este tipo de algoritmos**

  El algoritmo acelerado de proyección en bloques, es un algoritmo matemático que exige, por un lado, ser implementado en un lenguaje con gran robustez numérica, ya que la precisión es un factor fundamental, y, por otro lado, efectúa una gran cantidad de operaciones sobre matrices y vectores dentro de loops. En consecuencia, *la plataforma de ejecución "vector processor", como lo es CRAY J90 PVP, es la más adecuada* para este tipo de algoritmos. Como se puede apreciar en los experimentos de testeo, en TODOS los casos se observa una mejora sustancial en los tiempos de ejecución de una implementación vectorizada del algoritmo corriendo en una plataforma *vector processor,* que la misma implementación ejecutada en una plataforma escalar.
  La *vectorización del código* a ser ejecutado en una plataforma vectorial, es decir, el análisis del mismo y su estructuración para un mejor aprovechamiento de la plataforma, es una tarea sumamente importante. Se debe efectuar un análisis exhaustivo del código, detectando los posibles inhibidores de vectorización para solucionarlos.
  Es importante destacar que al lograr un código vectorizado y ejecutarlo en una plataforma *vector processor* se logra un nivel de paralelismo: el que se produce a nivel registros vectoriales de hardware, donde se ejecutan en paralelo los loops vectorizados.

- **La importancia de una implementación aprovechando las características paralelas del algoritmo.**

  El algoritmo acelerado de proyección en bloques tiene una importante región que puede ser ejecutada en paralelo: dicha región es alcanzada en cada iteración del algoritmo, cuando se calcula la *dirección de proyección de cada bloque del sistema.* Como dispusimos de una plataforma de corrida multiprocesador, agregamos al código directivas para el procesamiento paralelo, corroborando que una implementación paralela del algoritmo

mejora los tiempos de ejecución en comparación con la implementación vectorizada. Esto lo pudimos apreciar tanto en la versión densa ejecutada sobre matrices de Hilbert, como en los ejemplos Sameh ejecutados con distintas dimensiones. Allí se apreció cómo la versión paralela mejora al tener más dimensión y bloques (el overhead de distribución del trabajo paralelo *se solapa* completamente haciendo que esta versión mejore respecto de la vectorizada). Estos resultados nos hacen pensar en una proyección de mejora aún mayor para sistemas de ecuaciones más grandes.
La "versión paralela" es también ejecutada explotando las características vectoriales de la plataforma. Es en realidad una implementación "vectorizada con inserción de directivas para el procesamiento paralelo". La llamamos simplemente "versión paralela", por una cuestión de simplicidad, pero queda claro que su ejecución no excluye el aprovechamiento de las características vectoriales de la plataforma.

Por lo expuesto anteriormente creemos que la implementación obtenida respetando estos tres ejes puede considerarse una implementación *óptima* del algoritmo acelerado de proyección en bloques.
Las técnicas de vectorización y las estructuras de datos aquí utilizadas son recomendables para todo algoritmo matemático que involucre muchos cálculos con matrices y vectores, y las técnicas de paralelización para todo algoritmo al que se le detecte una región paralela.

ANEXO I

Diseño.
Refinamiento procedural Top Down

A continuación presentamos el refinamiento Top - Down del algoritmo ideado para resolver sistemas lineales no simétricos del tipo $A^t.x = b$ (donde A es la Matriz de los coeficientes, x es el Vector de las incógnitas, b es el Vector de los términos independientes) basándonos en la estrategia de partición y resolución por métodos de proyección en bloques (PAM).

De acuerdo a lo explicado en el Capítulo III, esta metodología de diseño consiste en ir refinando los módulos que conforman la solución a nuestro problema; se parte de un nivel inicial, el cual describe las principales tareas a realizar por nuestro algoritmo, y luego, sucesivamente en cada nivel, se va desarrollando más detalladamente cada módulo.

Las referencias para interpretar el refinamiento son las siguientes: Las tareas que se encuentran en *cursiva* son aquellas que serán refinadas en el nivel anterior; las tareas que están en **negrita** indican que a continuación sigue su descripción y las tareas en formato normal indican que ya no serán detalladas.

NIVEL 1

1. *Obtener datos del sistema correspondiente al problema a resolver*
2. *Preprocesamiento de los datos del sistema*
3. *Búsqueda de una solución aceptable*
4. *Retornar los datos de la resolución*

NIVEL 2

1. **Obtener datos del sistema correspondiente al problema a resolver**
   - 1.1. Obtener datos de la matriz A
   - *1.2. Almacenar eficientemente la matriz A*
   - 1.3. Obtener datos del vector b
2. **Preprocesamiento de los datos del sistema**
   - 2.1. Normalizar la matriz A y el vector b
   - *2.2. Dividir óptimamente la matriz A en bloques A_i*
3. **Búsqueda de una solución aceptable**
   - 3.1. Tomar como punto inicial un vector $x = x_0$
   - 3.2. MIENTRAS {*la calidad de la solución no sea aceptable*} HACER
     - 3.2.1. PARA {cada bloque A_i} HACER
       - *3.2.1.1. Obtener dirección d_i de proyección del punto x al hiperplano A_i*

       FIN
     - 3.2.2. Normalizar la matriz D donde se almacenan las direcciones
     - *3.2.3. Obtener dirección óptima d de proyección del punto x*
     - *3.2.4. Proyectar el punto x en la dirección d*

     FIN
4. **Retornar los datos de la resolución**
   - 4.1. Mostrar solución
   - 4.2. Mostrar residuo
   - 4.3. Mostrar tiempos de respuesta
   - 4.4. Mostrar cantidad de iteraciones empleadas en llegar a la solución

NIVEL 3

1. **Obtener datos del sistema correspondiente al problema a resolver**
   1.1. Obtener datos de la matriz A
   1.2. Almacenar eficientemente la matriz A
      1.2.1. SI {A es una matriz esparsa} ENTONCES
         1.2.1.1. Almacenar la matriz A con la representación elegida
         FIN
   1.3. Obtener datos del vector b
2. **Preprocesamiento de los datos del sistema**
   2.1. Normalizar la matriz A y el vector b
   2.2. Dividir óptimamente la matriz A en bloques A_i
      2.2.1. MIENTRAS {Existan filas de A no asignadas a ningún bloque} HACER
         2.2.1.1. Armar bloque A_i
         FIN
3. **Búsqueda de una solución aceptable**
   3.1. Tomar como punto inicial un vector $x = x_0$
   3.2. MIENTRAS {El punto x no cumpla el criterio de aceptación de solución elegido} HACER
      3.2.1. PARA {cada bloque A_i} HACER
         3.2.1.1. Obtener dirección d_i de proyección del punto x al hiperplano A_i
            3.2.1.1.1. Hallar vector u_i calculando $(A_i \, . \, A_i^t)^{-1} . (b_i - A_i^t . x)$ (Usando Desc. de Cholesky)
            3.2.1.1.2. Hallar dirección d_i calculando $u_i \, . \, A_i^t$
            3.2.1.1.3. Obtener dirección mejorada d_i'
            3.2.1.1.4. Almacenar la dirección en la matriz D (Usando Desc. de Cholesky)
         FIN
      3.2.2. Normalizar la matriz D donde se almacenan las direcciones
      3.2.3. **Obtener dirección óptima d de proyección del punto x**
         3.2.3.1. Depurar direcciones de la matriz D
         3.2.3.2. Mezclar direcciones restantes para obtener dirección optimal d
      3.2.4. Proyectar el punto x en la dirección d
         3.2.4.1. Tomar como punto actual el punto x más la dirección d
      FIN MIENTRAS
4. **Retornar los datos de la resolución**
   4.1. Mostrar solución
   4.2. Mostrar residuo
   4.3. Mostrar tiempos de respuesta
   4.4. Mostrar cantidad de iteraciones empleadas en llegar a la solución

NIVEL 4

1. **Obtener datos del sistema correspondiente al problema a resolver**
   1.1. Obtener datos de la matriz A
   1.2. Almacenar eficientemente la matriz A
      1.2.1. SI {A es una matriz esparsa} ENTONCES
         1.2.1.1. Almacenar la matriz A con una representación eficiente
         FIN
   1.3. Obtener datos del vector b

2. **Preprocesamiento de los datos del sistema**
   2.1. Normalizar la matriz A y el vector b
   2.2. Dividir óptimamente la matriz A en bloques A_i
      2.2.1. MIENTRAS {Existan filas de A no asignadas a ningún bloque} HACER
         2.2.1.1. Armar bloque A_i
            2.2.1.1.1. MIENTRAS {Existan filas de A no asignadas a ningún bloque} Y {el bloque no tenga más filas que límite permitido} HACER
               2.2.1.1.1.1. Tomar 1er fila no asignada
               2.2.1.1.1.2. SI {El bloque no tiene ninguna fila asignada} ENTONCES
                  2.2.1.1.1.2.1. Incluir fila en el bloque y almacenar su Desc. de Cholesky
                  SINO
                  2.2.1.1.1.2.2 Estudiar número de condición de la matriz correspondiente al bloque con la nueva fila incluida
                  2.2.1.1.1.2.3 SI {El número de condición es aceptable} ENTONCES
                     2.2.1.1.1.2.3.1 Incluirla en el bloque y almacenar su Desc. de Cholesky
                  FIN
               FIN
      FIN

3. **Búsqueda de una solución aceptable**
   3.1. Tomar como punto inicial un vector $x = x_0$
   3.2. MIENTRAS {El punto x no cumpla el criterio de aceptación de solución elegido} HACER
      3.2.1. PARA {cada bloque A_i} HACER
         3.2.1.1. Obtener dirección d_i de proyección del punto x al hiperplano A_i
            3.2.1.1.1. Hallar vector u_i calculando $(A_i . A_i^t)^{-1} .(b_i - A_i^t .x)$ (Usando Desc. de Cholesky)
            3.2.1.1.2. Hallar dirección d_i calculando $u_i . A_i^t$
            3.2.1.1.3. Obtener dirección mejorada d_i'
               3.2.1.1.3.1. SI {no es la 1er iteración del algoritmo} ENTONCES
                  3.2.1.1.3.1.1. Tomar **d**, dirección proyectada en la iteración anterior
                  3.2.1.1.3.1.2. Hallar dirección d_i' calculando $d_i - ((\mathbf{d}^t . d_i)/ |\mathbf{d}|^2) . \mathbf{d}$
               FIN
            3.2.1.1.4. Almacenar la dirección en la matriz D (Usando Desc. de Cholesky)
         FIN
      3.2.2. Normalizar la matriz donde se almacenan las direcciones A_i (matriz D)

3.2.3. Obtener dirección óptima d de proyección del punto x

3.2.3.1. Depurar direcciones de la matriz D

3.2.3.1.1. PARA {cada dirección d_i'de D} HACER

3.2.3.1.1.1. SI {es la 1er dirección} ENTONCES

3.2.3.1.1.1.1. Incluirla en la matriz de las direcciones depuradas D´ y almacenar la Desc. de Cholesky

SINO

3.2.3.1.1.1.2. Estudiar el nro. de condición de la matriz D´ con d_i'incluida

3.2.3.1.1.1.3. SI {el nro. de condición es aceptable} ENTONCES

3.2.3.1.1.1.3.1. Incluir dirección en D´ y almacenar la Desc. de Cholesky

FIN

FIN

FIN

3.2.3.2. Mezclar direcciones restantes para obtener dirección optimal d

3.2.3.2.1. Hallar vector w calculando $(D'^t . D')^{-1} . [\,|d_1|^2/|d_1'|, \ldots, |d_h|^2/|d_h'|\,]$

3.2.3.2.2. Hallar dirección optimal d calculando $w_1 . d_1' + w_2 . d_2' + \ldots + w_h . d_h'$

3.2.4. Proyectar el punto x en la dirección d

3.2.4.1. Tomar como punto actual el punto x más la dirección d

FIN MIENTRAS

4. Retornar los datos de la resolución

4.1. Mostrar solución
4.2. Mostrar residuo
4.3. Mostrar tiempos de respuesta
4.4. Mostrar cantidad de iteraciones empleadas en llegar a la solución

Anexo II

AII.1 Prueba numérica de la implementación del algoritmo de división en bloques

Ejemplo del proceso de prueba del algoritmo de división en bloques de una matriz de Hilbert de 10x10.
Tamaño máximo de filas por bloque: 4.
Tolerancia de aceptación de filas: 10^4.
Plataforma de corrida: Procesador Pentium 233 Mhz., 64M memoria RAM.

I.i Elementos de la matriz de Hilbert original normalizada

vA[1,1]	0.803279517221
vA[1,2]	0.401639758610
vA[1,3]	0.267759839074
vA[1,4]	0.200819879305
vA[1,5]	0.160655903444
vA[1,6]	0.133879919537
vA[1,7]	0.114754216746
vA[1,8]	0.100409939653
vA[1,9]	8.925327969120E-02
vA[1,10]	8.032795172208E-02
vA[2,1]	0.669330132430
vA[2,2]	0.446220088286
vA[2,3]	0.334665066215
vA[2,4]	0.267732052972
vA[2,5]	0.223110044143
vA[2,6]	0.191237180694
vA[2, 7]	0.167332533107
vA[2,8]	0.148740029429
vA[2,9]	0.133866026486
vA[2,10]	0.121696387714
vA[3,1]	0.593935895579
vA[3,2]	0.445451921685
vA[3,3]	0.356361537348
vA[3,4]	0.296967947790
vA[3,5]	0.254543955248
vA[3,6]	0.222725960842
vA[3,7]	0.197978631860
vA[3, 8]	0.178180768674
vA[3,9]	0.161982516976
vA[3,10]	0.148483973895
vA[4,1]	0.545827216095
vA[4,2]	0.436661772876
vA[4,3]	0.363884810730
vA[4,4]	0.311901266340
vA[4,5]	0.272913608048

vA[4,6]	0.242589873820
vA[4,7]	0.218330886438
vA[4,8]	0.198482624035
vA[4,9]	0.181942405365
vA[4,10]	0.167946835722
vA[5,1]	0.512341191256
vA[5,2]	0.426950992714
vA[5,3]	0.365957993754
vA[5,4]	0.320213244535
vA[5,5]	0.284633995142
vA[5,6]	0.256170595628
vA[5,7]	0.232882359662
vA[5,8]	0.213475496357
vA[5,9]	0.197054304329
vA[5,10]	0.182978996877
vA[6,1]	0.487610556882
vA[6,2]	0.417951905899
vA[6,3]	0.365707917662
vA[6,4]	0.325073704588
vA[6,5]	0.292566334129
vA[6,6]	0.265969394663
vA[6,7]	0.243805278441
vA[6,8]	0.225051026253
vA[6,9]	0.208975952950
vA[6,10]	0.195044222753
vA[7,1]	0.468553735569
vA[7,2]	0.409984518622
vA[7,3]	0.364430683220
vA[7,4]	0.327987614898
vA[7,5]	0.298170558998
vA[7,6]	0.273323012415
vA[7,7]	0.252298165306
vA[7,8]	0.234276867784
vA[7,9]	0.218658409932
vA[7,10]	0.204992259311
vA[8,1]	0.453393756659
vA[8,2]	0.403016672586
vA[8,3]	0.362715005327
vA[8,4]	0.329740913934
vA[8,5]	0.302262504440
vA[8,6]	0.279011542560
vA[8,7]	0.259082146662
vA[8,8]	0.241810003552
vA[8,9]	0.226696878330
vA[8,10]	0.213361767840
vA[9,1]	0.441031336546
vA[9,2]	0.396928202892
vA[9,3]	0.360843820811
vA[9,4]	0.330773502410
vA[9,5]	0.305329386840
vA[9,6]	0.283520144923

vA[9,7]	0.264618801928
vA[9,8]	0.248080126807
vA[9,9]	0.233487178172
vA[9,10]	0.220515668273
vA[10,1]	0.430748422844
vA[10,2]	0.391589475313
vA[10,3]	0.358957019037
vA[10,4]	0.331344940649
vA[10,5]	0.307677444889
vA[10,6]	0.287165615229
vA[10,7]	0.269217764277
vA[10,8]	0.253381425202
vA[10,9]	0.239304679358
vA[10,10]	0.226709696234

I.ii Elementos de la matriz de Hilbert original normalizada ordenada, luego del preprocesamiento

El proceso de división genera 4 bloques:

El bloque 1 tiene 4 filas
El bloque 2 tiene 3 filas
El bloque 3 tiene 2 filas
El bloque 4 tiene 1 fila

Los cuatro colores indican cada uno de los distintos bloques. Se puede ver claramente cómo el algoritmo efectúa la partición del sistema original.

vAord[1,1]	0.803279517221
vAord[1,2]	0.401639758610
vAord[1,3]	0.267759839074
vAord[1,4]	0.200819879305
vAord[1,5]	0.160655903444
vAord[1,6]	0.133879919537
vAord[1,7]	0.114754216746
vAord[1,8]	0.100409939653
vAord[1,9]	8.925327969120E-02
vAord[1,10]	8.032795172208E-02
vAord[2,1]	0.669330132430
vAord[2,2]	0.446220088286
vAord[2,3]	0.334665066215
vAord[2,4]	0.267732052972
vAord[2,5]	0.223110044143
vAord[2,6]	0.191237180694
vAord[2,7]	0.167332533107
vAord[2,8]	0.148740029429
vAord[2,9]	0.133866026486
vAord[2,10]	0.121696387714
vAord[3,1]	0.593935895579
vAord[3,2]	0.445451921685
vAord[3,3]	0.356361537348

vAord[3,4]	0.296967947790
vAord[3,5]	0.254543955248
vAord[3,6]	0.222725960842
vAord[3,7]	0.197978631860
vAord[3,8]	0.178180768674
vAord[3,9]	0.161982516976
vAord[3,10]	0.148483973895
vAord[4,1]	0.468553735569
vAord[4,2]	0.409984518622
vAord[4,3]	0.364430683220
vAord[4,4]	0.327987614898
vAord[4,5]	0.298170558998
vAord[4,6]	0.273323012415
vAord[4,7]	0.252298165306
vAord[4,8]	0.234276867784
vAord[4,9]	0.218658409932
vAord[4,10]	0.204992259311
vAord[5,1]	0.545827216095
vAord[5,2]	0.436661772876
vAord[5,3]	0.363884810730
vAord[5,4]	0.311901266340
vAord[5,5]	0.272913608048
vAord[5,6]	0.242589873820
vAord[5,7]	0.218330886438
vAord[5,8]	0.198482624035
vAord[5,9]	0.181942405365
vAord[5,10]	0.167946835722
vAord[6,1]	0.512341191256
vAord[6,2]	0.426950992714
vAord[6,3]	0.365957993754
vAord[6,4]	0.320213244535
vAord[6,5]	0.284633995142
vAord[6,6]	0.256170595628
vAord[6,7]	0.232882359662
vAord[6,8]	0.213475496357
vAord[6, 9]	0.197054304329
vAord[6, 10]	0.182978996877
vAord[7,1]	0.453393756659
vAord[7,2]	0.403016672586
vAord[7,3]	0.362715005327
vAord[7,4]	0.329740913934
vAord[7,5]	0.302262504440
vAord[7,6]	0.279011542560
vAord[7,7]	0.259082146662
vAord[7,8]	0.241810003552
vAord[7,9]	0.226696878330
vAord[7,10]	0.213361767840
vAord[8,1]	0.487610556882
vAord[8,2]	0.417951905899
vAord[8,3]	0.365707917662
vAord[8,4]	0.325073704588

vAord[8,5]	0.292566334129
vAord[8,6]	0.265969394663
vAord[8,7]	0.243805278441
vAord[8,8]	0.225051026253
vAord[8,9]	0.208975952950
vAord[8,10]	0.195044222753
vAord[9,1]	0.441031336546
vAord[9,2]	0.396928202892
vAord[9,3]	0.360843820811
vAord[9,4]	0.330773502410
vAord[9,5]	0.305329386840
vAord[9,6]	0.283520144923
vAord[9,7]	0.264618801928
vAord[9,8]	0.248080126807
vAord[9,9]	0.233487178172
vAord[9,10]	0.220515668273
vAord[10,1]	0.430748422844
vAord[10,2]	0.391589475313
vAord[10,3]	0.358957019037
vAord[10,4]	0.331344940649
vAord[10,5]	0.307677444889
vAord[10,6]	0.287165615229
vAord[10,7]	0.269217764277
vAord[10,8]	0.253381425202
vAord[10,9]	0.239304679358
vAord[10,10]	0.226709696234

I.iii Verificación de la división

El siguiente es el output del proceso de prueba de los bloques. Efectivamente se puede ver como $(A_i^t . A_i).(L_i . D_i . L_i^t)^{-1}$ se aproxima muchísimo a la I para cada bloque.

Identidad para Bloque: 1

I[1,1]=	**1.00000000000**
I[1,2]=	**5.480060849550E-12**
I[1,3]=	**-9.036327242029E-12**
I[1,4]=	**6.892264536873E-13**
I[2,1]=	**-9.179323967601E-13**
I[2,2]=	**1.00000000001**
I[2,3]=	**-2.620126338115E-11**
I[2,4]=	**1.685318551381E-12**
I[3,1]=	**-4.976907774790E-12**
I[3,2]=	**2.688516076432E-11**
I[3,3]=	**0.999999999977**
I[3,4]=	**4.979128220839E-12**
I[4,1]=	**1.983746500400E-12**
I[4,2]=	**-1.897149104479E-12**
I[4,3]=	**-8.808953566586E-12**
I[4,4]=	**0.999999999998**

Identidad para Bloque: 2

I[1,1]= 1.00000000001
I[1,2]= -1.040767472205E-11
I[1,3]= 4.505729123139E-12
I[2,1]= 2.891020756124E-12
I[2,2]= 0.999999999995
I[2,3]= 2.676081578556E-12
I[3,1]= 1.882938249764E-12
I[3,2]= -4.554578936222E-12
I[3,3]= 1.00000000000

Identidad para Bloque: 3
I[1,1]= 1.00000000000
I[1,2]= 2.819966482548E-14
I[2,1]= 0.000000000000E+00
I[2,2]= 1.00000000000

Identidad para Bloque: 4
I[1,1]= 1.00000000000

AII.2 Solución del algoritmo acelerado de proyección en bloques

Ejemplo de la solución del algoritmo acelerado de proyección en bloques para una matriz de Hilbert de 10x10, armada de tal manera que $\sum a_{i\,j}=b_i$, y partiendo del punto inicial (0,0,......,0).
Tamaño máximo de filas por bloque: 4.
Tolerancia de aceptación de filas: 10^4.
Plataforma de corrida: Procesador Pentium 233 Mhz., 64M memoria RAM.

SOLUCIÓN:

vX[1] = 0.997890975127
vX[2] = 1.00299833230
vX[3] = 1.00210268608
vX[4] = 0.997652560873
vX[5] = 0.997210592160
vX[6] = 0.999232588735
vX[7] = 1.00141573922
vX[8] = 1.00229774190
vX[9] = 1.00120195596
vX[10]= 0.997952166147

RESIDUO: 7.216260641855E-04

GLOSARIO

alcance de una variable

Región de un programa en la que una variable es definida y puede ser referenciada.

algoritmo de resolución de sistemas de ecuaciones secuencial

Se define una secuencia de control y el algoritmo realiza, de una manera estrictamente secuencial, iteraciones de acción por filas, desde un punto inicial apropiado hasta que se cumpla una condición de parada.

algoritmo de resolución de sistemas de ecuaciones paralelo

Iteraciones de acción por filas son realizadas concurrentemente sobre todas las filas y el iterado x^{k+1} es entonces generado a partir de todos los iterados intermedios $x^{k+1,i}$

algoritmo de resolución de sistemas de ecuaciones iterativo por bloques secuencial

El sistema original es descompuesto en bloques y la secuencia de control realiza, secuencialmente, iteraciones de acción por bloque.

algoritmo de resolución de sistemas de ecuaciones iterativo por bloques paralelo

Iteraciones por bloques son realizadas concurrentemente sobre todos los bloques y el iterado siguiente x^{k+1} se genera a partir de los iterados intermedios $x^{k+1,t}$

arreglo

(1) Estructura de datos cuyos elementos son accedidos en forma indexada.

(2) En Fortran 90, un objeto con atributo DIMENSION. Es un conjunto de datos homogéneos (es decir, pertenecientes al mismo tipo de datos), los cuales se hallan indexados. Las matrices son arreglos de dos dimensiones.

autotasking

Es una forma de procesamiento palalelo de Cray Research, que *automáticamente,* y en base a directivas de procesamiento paralelo insertadas por el programador (optativas), realiza una división óptima de las tareas entre varios procesadores, haciendo más eficiente el uso de los recursos de hardware. El agregado de directivas de procesamiento paralelo se hace necesario en programas complejos, en los cuales el autotasking no es capaz de detectar regiones paralelas automáticamente.

benchmark

Medida inicial del proceso de optimización de un código.

dependencia de datos

Es un inhibidor de vectorización, que se produce dentro de un loop, cuando el resultado de una ejecución vectorizada difiere de su correspondiente ejecución escalar.

descomposición de Cholesky de una matriz

Si A es una matriz simétrica y definida positiva, su descomposición de Cholesky es:

$A = L . D . L^t$

donde, L es una matriz triangular inferior con los elementos de su diagonal iguales a 1 y D es una matriz diagonal.

escalar

En Fortran 90, un objeto simple de un tipo de datos predefinido o derivado (no vectorizado)

estimador del número de condición de una matriz

Un estimador del número de condición de una matriz M de la cual conocemos su descomposición de Cholesky $M=L^t.D.L$ es $K'(M)=máx(D_{i\,i})/mín(D_{i\,i})$

interface

Conjunto de especificaciones que definen las características de un procedimiento o función: qué tarea realiza (en lenguaje natural), así como la forma en la que debe ser invocado.

loop de búsqueda

Loop del que se puede salir mediante una sentencia IF.

loop de reducción

Loop que contiene al menos una sentencia que reduce un arreglo a un valor escalar ejecutando operaciones acumulativas sobre sus elementos, las cuales incluyen el resultado de la iteración actual y las previas.

matriz esparsa

Matriz con un alto porcentaje de valores nulos o cero.

multitasking

(1) La ejecución paralela de dos o más partes de un programa en múltiples CPUs.

(2) Es un método en sistemas multiusuario que incorpora múltiples CPUs interconectadas; esas CPUs ejecutan sus programas simultáneamente (en paralelo) y comparten recursos como memoria, dispositivos de almacenamiento, o impresoras. Este término puede ser empleado como sinónimo de *procesamiento paralelo*.

número de condición de una matriz

Se denomina número de condición K(A) de la matriz no singular A relativo a una norma a $K(A)=||A||.||A^{-1}||$

optimización

Proceso iterativo en el que se analiza un código y se lo modifica a fin de lograr un criterio de performance preestablecido (por ejemplo, reducción del tiempo de respuesta, optimización de uso de CPU, de I/O, etc.)

overhead

Es el "trabajo extra" que debe ser realizado por programas que son ejecutados en múltiples CPUs. Por ejemplo, la tarea de dividir la región paralela entre múltiples procesadores, con la consiguiente alocación de las variables privadas de la misma, o el tiempo extra de espera en puntos de sincronización.

procedimiento

Subprograma (conjunto de datos y/o estructuras de control), que tiene un nombre y una función conceptual determinada, es invocado desde un programa, (se transfiere el control de ejecución al mismo), para ser ejecutado y retomar luego con la ejecución de la instrucción siguiente a su llamado.

procesamiento paralelo

Tipo de procesamiento en el que múltiples procesadores trabajan en la ejecución de una aplicación.

región crítica

Porción de código que debe ser ejecutada en un solo procesador a la vez. Es un mecanismo de sincronización que fuerza al acceso secuencial serial de una porción de código.

región paralela

Región de código que puede ser ejecutada por múltiples procesadores.

threshold test

Se emplea una expresión booleana para, en tiempo de ejecución, determinar si la región paralela es ejecutada en múltiples o en una única CPU.

variable

Objeto cuyo valor puede ser definido y redefinido en ejecución.

vectorización

Es el procesamiento de grupos de números empleando los registros y unidades vectoriales de hardware. El procesamiento vectorial permite que una operación simple sea ejecutada concurrentemente sobre un conjunto (o vector) de operandos.

BIBLIOGRAFÍA

- H.Scolnik, N.Echebest, M.T.Guardarucci, M.C.Vacchino ,**A New method for Solving Large Sparse Systems of Linear Equations using Row Projections,** Proceeding CESA'98, Tunez.

- Y. Censor y S. Zenios; **Paralell Optimization - Theory, Algorithms and Applications. Numerical Mathematics and scientific computation**; Oxford University Press, 1997.

- Bjorck, **Numerical Methods for Least Squares Problems**; SIAM Journal, Fiadelfia, 1996.

- R. Bramley y Sameh; **Row Proyection Methods for Large Nonsymmetric Linear Systems,** SIAM Journal Sci. Stat. Compt, vol 13,1992.

- M. Garcia Palomares; **Parallel Proyected Aggregation Methods for Solving The Convex Feasibility Problem,** SIAM Journal Optimization 3-4, Nov. 1993.

- **Optimizing Application Code on Cray PVP Systems,** Volumen I y II. Software Education Services, Cray Research, 1998.

- William H. Press, Saul A. Teukolsky, William T. Vetterling y Brian P. Flannery, **Numerical recipes in Fortran. The art of scientific computing,** Oxford University Press, Segunda Edición, 1998.

- Paul Helman y Robert Veroff, **Problem Solving and Data Structures. Walls and Mirrors,** The Benjamin/Cummings Publishing Company Inc., Segunda Edición, 1991.

- Viktor K. Decyk, Charles D. Norton y Boleslaw K. Szymanski, **Expressing Object-Oriented Concepts in Fortran 90,** ACM Fortran Forum, vol. 16, num. 1, Abril 1997.

- Viktor K. Decyk, Charles D. Norton y Boleslaw K. Szymanski, **Introduction to Object-Oriented Concepts using Fortran 90,** ACM Fortran Forum, vol. 16, num. 1, Abril 1997.

- Gregory R. Andrews, **Concurrent Programming - Principles and Practice,** The Benjamin/Cummings Publishing Company Inc., 1991

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

DONACION .... TES
$ .... 9912 ej. 1
Fecha 29-9-05
Inv. E .... Inv. B 2066

TES
99/2
DIF-02066
SALA

UNIVERSIDAD NACIONAL DE LA PLATA
FACULTAD DE INFORMATICA
Biblioteca
50 y 120 La Plata
catalogo.info.unlp.edu.ar
biblioteca@info.unlp.edu.ar

DIF-02066