UNIVERSIDAD EAFIT
Abierta al mundo
Acreditada Institucionalmente por el Ministerio de Educación Nacional

ISSN 1692-0694

MÉTODOS DIRECTOS PARA LA SOLUCIÓN DE SISTEMAS DE ECUACIONES LINEALES SIMÉTRICOS, INDEFINIDOS, DISPERSOS Y DE GRAN DIMENSIÓN

JUAN DAVID JARAMILLO JARAMILLO
INGENIERÍA DE SISTEMAS
INGENIERÍA MATEMÁTICA
UNIVERSIDAD EAFIT

ANTONIO M. VIDAL MACIÁ
GRUPO DE INVESTIGACIÓN EN COMPUTACIÓN PARALELA
UNIVERSIDAD POLITÉCNICA DE VALENCIA

FRANCISCO JOSÉ CORREA ZABALA
GRUPO DE INVESTIGACIÓN EN LÓGICA Y COMPUTACIÓN
GRUPO DE INVESTIGACIÓN EN INGENIERÍA DE SOFTWARE
UNIVERSIDAD EAFIT

Medellín, Febrero de 2006 DOCUMENTO 40-022006

Comentarios: Favor dirigirlos a **fcorrea@eafit.edu.co; jjaram14@eafit.edu.co; avidal@dsic.upv.es**

Tabla de contenido

LISTA DE FIGURAS

Pág.

“El secreto del éxito no es no tener problemas.
El secreto es encontrar soluciones”.

(Anónimo)

Autores

JUAN DAVID JARAMILLO JARAMILLO
Estudiante Ingeniería de Sistemas de la Universidad EAFIT.
Estudiante Ingeniería Matemática de la Universidad EAFIT.
Dirección electrónica: jjaram14@eafit.edu.co

ANTONIO M. VIDAL MACIÁ
Grupo de Investigación en Computación Paralela
Universidad Politécnica de Valencia
Ph. D. en Informática
Dirección electrónica: avidal@dsic.upv.es

FRANCISCO JOSÉ CORREA ZABALA
Grupo de Investigación en Lógica y Computación
Grupo de Investigación en Ingeniería de Software
Universidad EAFIT
Ph.D. en Informática
Dirección electrónica: fcorrea@eafit.edu.co

Resumen

Este trabajo se centra en la optimización de los métodos numéricos para la solución de sistemas de ecuaciones lineales con matrices asociadas simétricas, indefinidas, dispersas y de gran dimensión. Los métodos se definen en entornos paralelos, haciendo uso de librerías de optimización y buscando el correcto manejo de las matrices dispersas para lograr la máxima optimización de los métodos definidos para la solución de sistemas de ecuaciones lineales con matrices asociadas que posean las características mencionadas.

Abstract

This work centers in the optimization of the methods for solving linear equation systems with sparse, symmetric, indefinite, large associated matrices. The methods are defined in parallel computation environments, making use of optimization libraries and looking for a correct use of sparse matrices to reach maximum optimization of the methods defined for solving linear equations systems with associated matrices that posses the characteristics mentioned above.

1 INTRODUCCIÓN

La solución de sistemas de ecuaciones lineales es un tema de gran interés en el mundo de la computación actual. Son muchos los problemas que pueden resolverse utilizando estas técnicas, y en áreas tan variadas como se quiera. Por eso es de gran importancia el estudio de algoritmos que optimicen el uso de recursos computacionales como la memoria física y el tiempo de ejecución, a la vez que se encuentren soluciones correctas de manera óptima.

Al estudiar las matrices generadas por los sistemas de ecuaciones se pueden observar ciertas características y propiedades, que pueden permitirnos en algunas ocasiones un tratamiento especial del problema mejorando los algoritmos tradicionales al explotar la estructura y las propiedades de la matriz.

En nuestro trabajo particular hemos pretendido encontrar métodos para la solución de sistemas de ecuaciones con matrices asociadas simétricas, indefinidas, de estructura dispersa y de gran dimensión. Los métodos se definen con base en métodos ya existentes, y se busca la solución tanto en procesamiento secuencial como paralelo.

Este tipo de sistemas se encuentran en diversas áreas como optimizaciones lineales y no lineales, análisis de elementos finitos, problemas inversos de valores propios y muchas más. En general se debe usar algún tipo de pivoteamiento para resolver correctamente estos sistemas y el reto es mantener una estabilidad computacional y la estructura dispersa en las matrices [1].

1.1 DEFINICIÓN CONTEXTUAL

Los sistemas de ecuaciones lineales sirven para resolver múltiples problemas de todos los tipos imaginables en procesos naturales, en producción, en redes eléctricas, y en innumerables campos más. Los problemas varían en sus características y pueden ser resueltos con la ayuda de un computador de manera más eficiente si se trabaja explotando las características del problema mismo.

El interés por la optimización de los métodos para la solución de sistemas de ecuaciones lineales ha generado diversos enfoques en cuanto a los métodos existentes para mejorar los algoritmos actuales. Aparecen entonces herramientas tan importantes como la computación paralela, librerías optimizadas con funciones básicas, librerías más avanzadas, tratamiento de la estructura de la matriz, etc. Todos estos enfoques no son mutuamente excluyentes, y por el contrario se intentará en este proyecto obtener el máximo provecho de cada uno de ellos, obteniendo un buen rendimiento global.

Las aplicaciones son muchas, incluso en un enfoque tan específico como el de las matrices simétricas, dispersas e indefinidas. Entre ellas están los problemas de redes eléctricas, problemas de elementos finitos, y en general muchas aplicaciones de ingeniería, como por ejemplo el resultado de la discretización de ecuaciones diferenciales parciales en diferentes tipos de simulaciones. Cualquier optimización que se logre en este campo, será un gran aporte, debido a la gran cantidad de datos que

suelen manejarse en estos casos, y a la necesidad de altos niveles de eficiencia que se necesitan.

1.2 MOTIVACIÓN

El principal interés de esta investigación es el estudio y la comprensión de los métodos de optimización existentes para métodos numéricos determinísticos. En específico se trabajará con miras a la computación paralela, el uso de librerías de optimización de álgebra lineal, el uso del lenguaje de programación FORTRAN, y el uso adecuado de estructuras para el almacenamiento y computación de matrices dispersas.

El problema de optimizar un código para la solución de sistemas de ecuaciones lineales simétricos, dispersos e indefinidos es un problema abierto, ya que no existe hasta ahora una aproximación que sea completamente optimizada. Esto le añade un valor especial de motivación al trabajo, y hace que cualquier logro obtenido sea de gran importancia en el medio.

El proceso de investigación es coordinado por el profesor Francisco José Correa de la Universidad EAFIT de Medellín, Colombia y el profesor Antonio Vidal de la Universidad Politécnica de Valencia, España. Ellos, desde su amplia experiencia en investigación y su formación docente, ayudarán para que el trabajo apunte siempre hacia el logro de las metas planteadas y el aprendizaje en los campos citados anteriormente.

1.3 PLAN DE TRABAJO

Inicialmente trabajamos en las principales formas de almacenamiento de las matrices dispersas, implementando algoritmos de conversión de formato denso a 5 diferentes formatos y viceversa, así como la ejecución del producto matriz-vector para cada uno de los formatos. La programación de los mismos se realiza en el lenguaje C.

Posteriormente estudiamos el entorno MPI para la programación paralela, realizando algunos ejemplos de implementación en lenguaje C. Al mismo tiempo se retomaron algunas bases importantes del álgebra lineal para definir algoritmos básicos como productos matriz-vector y matriz-matriz en tipos de matrices densas, simétricas, triangulares. En el proceso se refrescan las nociones y propiedades de matrices inversas, matrices singulares, normas, determinantes, ortogonalidad, condición, entre otros. Igualmente los algoritmos de eliminación Gaussiana, sustituciones progresiva y regresiva, factorización LU, pivoteo, entre otros.

El trabajo realizado anteriormente permite el diseño de los algoritmos básicos de descomposición LU, LDL^T y GG^T (Cholesky) en formato denso, implementándolos en sus formas matriciales y escalares en los lenguajes Matlab y Fortran.

Finalmente nos centramos en el problema de mayor interés para la investigación, que es el de la descomposición LDL^T para matrices dispersas, donde aún hay mucho por hacer. Este tipo de descomposición nos plantea varios desafíos: conservar las propiedades de las matrices, obtener una buena estabilidad numérica, lograr optimizar los métodos existentes y el manejo del almacenamiento en un formato comprimido, teniendo en cuenta que la matriz resultante podrá tener más elementos que la anterior. Utilizamos una reordenación de la matriz y una descomposición simbólica antes de pasar a la descomposición final. Para mayor optimización se usan también bibliotecas de funciones como BLAS, LAPACK, entre otras.

1.4 ORGANIZACIÓN

Con base en el plan de trabajo trazado y las características del problema en nuestra investigación, la organización de este reporte técnico será de la siguiente manera:

Sección 1. Introducción. Se presentan las nociones básicas y objetivos de la investigación, y planteamos el alcance y motivación para su realización.

Sección 2. Conceptos preliminares. Se plantean diferentes conceptos necesarios en otras secciones

para dar las bases suficientes para el buen entendimiento del documento.

Sección 3. Matrices Dispersas. Se profundiza sobre el manejo de las matrices dispersas en la computación numérica. Se trabajan diferentes formatos de almacenamiento y se dan ejemplos de su utilización.

Sección 4. Métodos de resolución de sistemas lineales. Se trabajan los métodos numéricos para la solución de sistemas lineales, tanto directos como iterativos, haciendo comparaciones entre ellos, y mostrando algunos ejemplos de los códigos.

Sección 5. Computación paralela. Se da un completo repaso sobre la computación paralela, desde las diferentes topologías de red, la conformación de clusters y los entornos de programación paralela MPI y PVM.

Sección 6. Bibliotecas numéricas optimizadas. Se muestran otras formas de optimización mediante el uso de técnicas manuales de optimización de códigos y las bibliotecas numéricas optimizadas.

Sección 7. Caso de estudio: descomposición LDL^T. Se trabaja sobre el problema específico de la solución de sistemas lineales simétricos, dispersos e indefinidos. Uno de los algoritmos bases para este tipo de sistemas es la descomposición LDL^T.

Sección 8. Conclusiones. Se dan las conclusiones que deja este largo proceso de investigación, dejando interrogantes abiertos y posibilidades de un enfoque para trabajos futuros.

2 CONCEPTOS PRELIMINARES

Durante el desarrollo de este reporte se hará referencia a diferentes conceptos básicos en los temas aquí tratados. Para su mayor comprensión, y debido a la importancia que poseen, se hace en este capítulo un breve repaso sobre las definiciones y propiedades de dichos conceptos, buscando con ello que el documento sea autocontenido.

Sistema de ecuaciones lineales

Un sistema de ecuaciones lineales es un conjunto de *m* ecuaciones con *n* incógnitas, cuya solución es un conjunto de valores para las incógnitas con el que se satisfacen todas las ecuaciones. En nuestro caso se asumirá que siempre hay la misma cantidad de ecuaciones que de incógnitas, para los cuales hay una única solución, cuando ésta existe.

En el planteamiento matemático de muchos problemas realistas los sistemas de ecuaciones algebraicas, y de una manera especial los lineales, aparecen de manera natural. También se presentan con frecuencia cuando se hace una discretización de ecuaciones diferenciales ordinarias y en derivadas parciales [13]. Para hallar la solución al sistema con ayuda de un computador se utilizan diferentes métodos, en los que se profundizará en este reporte. Para el manejo del sistema se hace uso de una matriz que representa los coeficientes de las incógnitas en cada ecuación, un vector de términos independientes y un vector solución, como lo muestra el siguiente ejemplo.

Supongamos el siguiente sistema de 4 ecuaciones con 4 incógnitas:

$$\begin{aligned} x_1 + 2x_2 + x_3 - x_4 &= 4 \\ 2x_1 - x_2 + 5x_3 - 2x_4 &= -3 \\ x_1 + 4x_2 - x_3 + 5x_4 &= 26 \\ x_1 + x_2 + x_3 + x_4 &= 10 \end{aligned}$$

Podemos representarlo como $\boldsymbol{Ax = b}$, así:

$$\begin{bmatrix} 1 & 2 & 1 & -1 \\ 2 & -1 & 5 & -2 \\ 1 & 4 & -1 & 5 \\ 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ -3 \\ 26 \\ 10 \end{bmatrix}$$

El objetivo es encontrar los valores del vector *x* mediante alguno de los métodos numéricos que se explicarán más adelante.

Esta matriz de coeficientes *A* puede tener diferentes propiedades que pueden ser explotadas en la resolución del sistema. Para su mayor comprensión se explican algunas propiedades con detalle.

Matriz triangular

Una matriz es triangular superior si cumple que $\forall_{i,j}\, i < j \rightarrow a_{ij} = 0$. Es decir, cada elemento de *A* ubicado por debajo de la diagonal principal tiene valor nulo. De igual manera se puede decir que matriz es triangular superior cuando cumple que

$\forall_{i,j}\ i > j \rightarrow a_{ij} = 0$. Los siguientes son ejemplos de matrices triangulares inferior (*L*) y superior (*U*).

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 6 & 3 & 0 & 0 \\ 3 & 3 & 1 & 0 \\ 2 & 7 & -4 & 8 \end{bmatrix} \qquad U = \begin{bmatrix} 1 & 2 & 5 & 1 \\ 0 & 3 & -2 & 3 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 8 \end{bmatrix}$$

Matriz simétrica

Una matriz es simétrica si se cumple que $\forall_{i,j}\ a_{ij} = a_{ji}$. Es decir, cada elemento de *A* ubicado en la fila *i*, columna *j*, es igual al elemento ubicado en la fila *j*, columna *i*. Este tipo de matrices nos puede ahorrar espacio de memoria en su almacenamiento, ya que con almacenar los elementos localizados por debajo de la diagonal principal de la misma (triángulo inferior), se conocen los valores de los elementos ubicados por encima de la diagonal superior (triángulo superior), y viceversa. Del mismo modo podemos ahorrar operaciones al aplicar algoritmos que exploten esta estructura.

Matriz definida positiva

Una matriz $A \in \Re^{n \times n}$ es definida positiva si es simétrica y $x^T Ax > 0$ para todo $x \in \Re^n$ diferente de cero. Los sistemas con matrices asociadas definidas positivas constituyen una de las clases más importantes de problemas $Ax = b$ [4]. Las matrices definidas positivas tienen una diagonal de peso, es decir con valores relativamente grandes.

Además, las matrices definidas positivas cumplen las siguientes propiedades:

- Una matriz *A* definida positiva es no singular.
- Si una matriz *A* es definida positiva entonces todas sus submatrices principales son definidas positivas. En particular, todas las entradas de la diagonal son positivas.
- Si una matriz *A* es definida positiva entonces existe una factorización $A = LDM^T$, y la matriz diagonal *D* tiene valores positivos.
- Si una matriz *A* es simétrica y definida positiva entonces existe una matriz triangular inferior única *G* con entradas positivas en la diagonal, tal que $A = GG^T$ (factorización de Cholesky).
- Una matriz simétrica *A* es definida positiva si y sólo si sus primeras submatrices principales tienen determinante positivo [32].

Matriz indefinida

Una matriz *A* es considerada indefinida cuando su forma cuadrática x^TAx toma valores tanto positivos como negativos. Aunque la matriz *A* puede tener una factorización LDL^T, las entradas en los factores pueden tener una magnitud arbitraria [4].

Matriz dispersa

Una matriz *A* se considera dispersa cuando un alto porcentaje de sus elementos son ceros. En nuestro caso consideraremos las matrices dispersas en cuanto pueda optimizarse la computación de la misma mediante el uso de estructuras especiales para el almacenamiento. Las matrices dispersas nos ahorran operaciones en las que su resultado se conoce de antemano, como productos y sumas con ceros. Las características de las matrices dispersas se estudian en detalle en la sección siguiente.

Sistemas de gran dimensión

Los sistemas de ecuaciones se consideran de gran dimensión cuando se trabaja con miles de incógnitas y los costos de computación son elevados. En estos casos se debe prestar mucha atención a minimizar los errores de redondeo y truncamiento que puedan generar los algoritmos. Al mismo tiempo se desea minimizar los tiempos de ejecución y el almacenamiento en memoria, haciendo algoritmos eficientes y con soluciones adecuadas en tiempos aceptables. En problemas de ingeniería y aplicaciones reales aparecen frecuentemente sistemas de este tipo.

La resolución de sistemas de ecuaciones algebraicas que son lineales de gran dimensión subyace en la solución numérica de problemas de la mecánica del continuo y muchos otros problemas de ingeniería,

llegando a constituir en muchos casos el principal factor de costo computacional [14].

Métodos para la solución de sistemas de ecuaciones lineales

Los métodos clásicos de resolución se suelen clasificar en directos o iterativos. Los primeros proporcionan una solución en un número fijo de operaciones y son más robustos. Sus principales inconvenientes son el costo tanto en tiempo de procesamiento como en almacenamiento en memoria y la cantidad de error propagado, que es mayor a medida que el número de ecuaciones aumenta, hasta el punto de calcular soluciones incorrectas para el sistema. La cantidad de operaciones requeridas para una matriz densa es de orden n^3, siendo n la cantidad de incógnitas. Los procedimientos iterativos son entonces preferidos para resolver sistemas de gran dimensión. Sin embargo en dichos sistemas de ecuaciones el condicionamiento de la matriz empeora y la corrección de los métodos de solución resulta afectada.

Factorización *LU*

La factorización *LU* es un método utilizado para factorizar la matriz A de $n \cdot n$ en el producto de la matriz triangular inferior L y la matriz triangular superior U; es decir, $A = LU$, donde la diagonal principal de L o U consta de unos o son iguales entre sí. Este algoritmo se explica en la sección 4.3 con los métodos directos.

Pivoteo

En cada paso de los métodos directos, un elemento de la diagonal es utilizado como pivote para calcular la factorización de las filas restantes, dividiéndolas por este valor. Elementos muy pequeños en la diagonal pueden entonces ocasionar elementos muy grandes en el resultado, ocasionando el crecimiento de los errores por redondeo. Es decir, en la computación de los métodos para la solución de sistemas de ecuaciones se pueden presentar problemas por acumulación de errores cuando los elementos de los factores triangulares computados son muy grandes. El problema viene ocasionado por el uso de pivotes muy pequeños. Una posible solución a esto es permutar la matriz, intercambiando filas o columnas para lograr que los elementos de la diagonal sean lo mayor posible. Esto se puede lograr al multiplicar A por una matriz identidad con sus filas reordenadas, a la que se da el nombre de matriz de permutación. Estas matrices se pueden representar mediante un vector que almacene los índices de las filas en el orden que se desean permutar.

Por ejemplo, supongamos una matriz A y su descomposición *LU* [4]

$$A = \begin{bmatrix} .0001 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 10000 & 1 \end{bmatrix} \begin{bmatrix} 0001 & 1 \\ 0 & -9999 \end{bmatrix} = LU$$

Ahora tomemos la matriz de permutación

$$P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Al aplicarla a la matriz inicial

$$PA = \begin{bmatrix} 1 & 1 \\ .0001 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ .0001 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & .9999 \end{bmatrix} = LU$$

Los factores triangulares quedan con valores aceptablemente pequeños.

Si P es una matriz de permutación, entonces PA es una versión de A con filas permutadas y AP es una versión de A con columnas permutadas. Una permutación simétrica de A tiene la forma PAP^T, e intercambia tanto filas como columnas.

3
MATRICES DISPERSAS

No siempre las matrices asociadas a un sistema de ecuaciones lineales son matrices completamente llenas de valores. En muchas ocasiones una gran parte de sus elementos, o incluso la mayoría, son ceros, y por lo tanto estos elementos no toman parte en los cálculos del algoritmo ya que el resultado de operar un número con un cero se conoce de antemano. Un buen tratamiento de estas matrices nos ahorra espacio en memoria y tiempo de ejecución en los métodos. Siempre que sea posible utilizar métodos alternativos de almacenamiento y de solución para explotar la estructura de una matriz, podemos considerar a ésta como dispersa.

Los primeros en hacer uso de dichas técnicas fueron los ingenieros eléctricos en los 60s, sin embargo no recibieron mucha atención hasta ahora. El uso de matrices dispersas es especialmente útil en métodos iterativos, donde la principal operación suele ser un producto matriz-vector, que se ve afectado de gran manera por los diferentes esquemas de almacenamiento y computación.

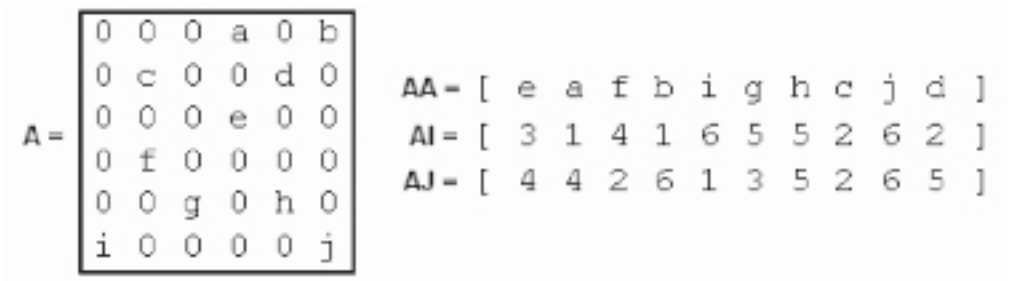
3.1 MÉTODOS DE ALMACENAMIENTO

Se consideran dos tipos de matrices dispersas: estructuradas y no estructuradas. La primera clase contiene las matrices cuyas entradas no nulas forman un patrón regular; ejemplos de ellas son las matrices banda, o con una estructura por bloques. La matriz no estructurada es la que no presenta ningún patrón claro en sus entradas no nulas [2].

Una matriz dispersa puede ser almacenada guardando solamente sus elementos no nulos, para lo cual se utiliza una estructura diferente a la tradicional para matrices densas. Estos almacenamientos hacen necesario modificar los algoritmos que recorren normalmente matrices densas.

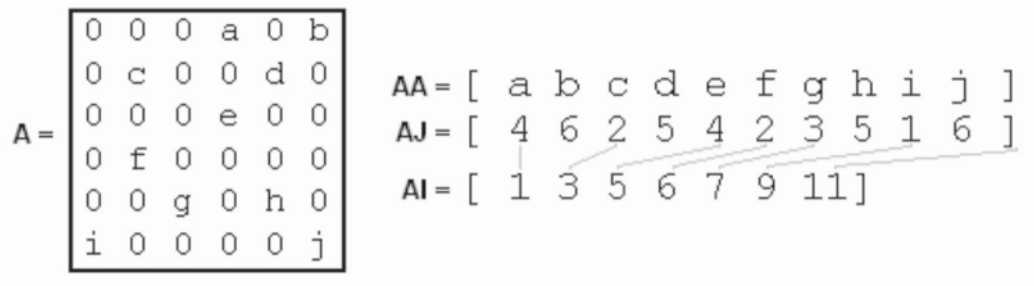
El esquema de almacenamiento más simple para matrices dispersas y base para los demás esquemas es conocido como el *formato coordinado* (COO). Esta estructura de datos, como se muestra en la figura 1, consiste en tres arreglos: el primero (*AA*) contiene todos los valores reales (o complejos) de los elementos no nulos de la matriz; el segundo (*AI*) es un arreglo de enteros y contiene los índices de fila de los elementos correspondientes; el último (*AJ*) es igualmente un arreglo de enteros y contiene todos los índices de columna. El espacio de almacenamiento que ocupa esta estructura es de *3nz*, donde *nz* representa el número de elementos no nulos de la matriz.

FIGURA 1
Almacenamiento disperso por coordenadas (COO)



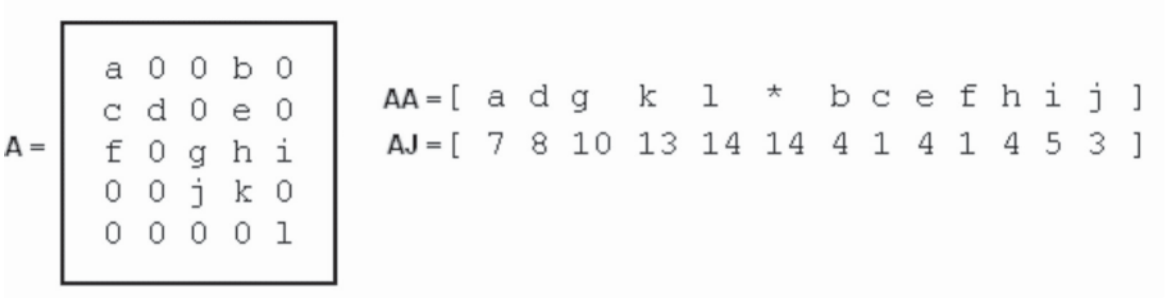
Para mejorar este formato se pueden utilizar dos variantes sencillas, una de compresión por filas *CSR* (Compressed Sparse Row) y otra de compresión por columnas *CSC* (Compressed Sparse Column). Ambas poseen también tres arreglos, que en el caso de CSR (figura 2) son los siguientes: Un arreglo que contiene los valores *Aij* diferentes de cero almacenados en orden por filas, desde la fila 1 hasta n. Este vector se denomina *AA* y tiene tamaño *nz*. El segundo es un arreglo de enteros que contiene los índices de columna de los elementos *Aij* almacenados en el arreglo *AA*, denominado *JA* y del mismo tamaño que el anterior. El último es un arreglo de enteros que contiene apuntadores a la posición en los arreglos *AA* y *JA* donde esté ubicado el primer elemento de cada fila. El contenido de este tercer arreglo llamado *IA,* en el elemento *i,* es la posición en los arreglos *AA* y *JA* donde comienza la fila *i*, y su tamaño es *n+1,* con *n* el número de filas de la matriz. Este formato es probablemente el más popular para el almacenamiento general de matrices dispersas. El formato de almacenamiento por CSC puede deducirse fácilmente a partir del CSR, haciendo una compresión por columnas en lugar de filas.

FIGURA 2
Almacenamiento disperso por fila comprimida (CSR)



Otra variación común explota el hecho que los elementos de la diagonal son usualmente diferentes de cero y accedidos con mayor frecuencia que los demás elementos. El formato modificado de la compresión por filas (*MSR*) contiene sólo dos arreglos. El primero llamado *AA* contiene las entradas no nulas de la matriz comenzando por la diagonal principal seguida de un marcador (*) y finalmente las demás entradas no nulas en el orden del almacenamiento CSR. El segundo arreglo, llamado *AJ*, contiene en los primeros *n* elementos las posiciones de los elementos en el primer arreglo que corresponden a primeras entradas de cada fila. En la posición *n+1* aparece el numero *nz+1* que indica que termina la diagonal y que continúan los otros elementos. De la posición *n+2* en adelante están los índices de columna de los elementos en *AA*.

FIGURA 3
Almacenamiento disperso por fila comprimida modificada (MSR).



Otros formatos de almacenamiento pueden ser usados para explotar propiedades específicas de cada problema como la simetría o estructuras de banda o por bloques.

Para observar el efecto de la utilización de estos tipos de almacenamiento de las matrices dispersas se efectuaron pruebas con varias matrices aleatorias con diferentes densidades (nnz / n^2). Se utilizaron matrices de 300x300, en formatos denso, coordenado (COO), comprimido por fila (CSR) y comprimido por columna (CSC), utilizando el lenguaje Matlab. Este lenguaje nos ofrece facilidades para el tratamiento de este tipo de matrices, ya que trae implementado el formato disperso COO. Para convertir una matriz de este formato a CSR o CSC se pueden usar los algoritmos 1 y 2 respectivamente. Los algoritmos utilizan como entrada una matriz en formato coordenado de Matlab (sparse) y retornan los vectores correspondientes a los formatos comprimidos.

ALGORITMO 1. Conversión COO a CSC en Matlab

```
1.  function [I,J,S] = coo2csc (AS)
2.  [I2,J2,S2] = find(AS)
3.  S = S2
4.  I = I2
5.  cont = 2
6.  J(1,1) = 1
7.  i = 2
8.  while i <= length(J2)
9.      while i < length(J2) & J2(i) == J2(i-1)
10.         i = i+1
11.     end
12.     J(cont,1) = i
13.     cont = cont+1
14.     i = i+1
15. end
16. J(cont-1,1) = i
```

ALGORITMO 2. Conversión COO a CSR en Matlab

```
1.  function [I,J,S] = coo2csr (AS)
2.   [I2,J2,S2] = find(AS)
3.  nfilas = size(AS,1)
4.  nz = nnz(AS)
5.  Itemp = zeros(1,nfilas+1)
6.  for i2 = 1:nz
7.      Itemp(I2(i2)) = Itemp(I2(i2))+1
8.  end
9.  I(1)=1
10. for i = 1:nfilas
11.     I(i+1) = I(i ) + Itemp(i)
12. end
13. cont = zeros(1,nz)
14. for i = 1:nz
15      S(I(I2(i))+cont(I2(i))) = S2(i)
16.     J(I(I2(i))+cont(I2(i))) = J2(i)
17.     cont(I2(i)) = cont(I2(i))+1
18. end
```

Teniendo las matrices convertidas a los diferentes formatos de almacenamiento podemos realizar algunos cálculos básicos que muestren la eficiencia que pueden alcanzar las operaciones realizadas sobre matrices dispersas al utilizar diversos formatos para su almacenamiento, que influyen en los tiempos de ejecución al tiempo que reducen el uso de la memoria física del computador.

ALGORITMO 3. Producto matriz-vector en formato denso en Matlab

```
1. function [b] = mv_den (A,x)
2. b = zeros(length(x),1)
3. [nfil,ncol] = size(A)
4. for i = 1:nfil
5.     for j = 1:ncol
6.         b(i) = b(i) + (x(j) * A(i,j))
7.     end
8. end
```

ALGORITMO 4. Producto matriz-vector en formato coordenado en Matlab

```
1. function [b] = mv_coo (I,J,S,x)
2. b = zeros(length(x),1);
3. nz = length(S);
4. for k = 1:nz
5.     b(I(k)) = b(I(k)) + (x(J(k)) * S(k))
6. end
```

ALGORITMO 5. Producto matriz-vector en formato comprimido CSR en Matlab

```
1. function [b] = mv_csr (I,J,S,x)
2. b = zeros(length(x),1)
3. nz = length(S)
4. nfil = length(x)
5. for i = 1:nfil
6.     for j = I(i):I(i+1)-1
7.         b(i) = b(i) + (x(J(j)) * S(j))
8.     end
9. end
```

ALGORITMO 6. Producto matriz-vector en formato comprimido CSC en Matlab

```
1. function [b] = mv_csc (I,J,S,x)
2. b = zeros(length(x),1)
3. nz = length(S)
4. ncol = length(x);
5. for j = 1:ncol
6.     for i = J(j):J(j+1)-1
7.         b(I(i)) = b(I(i)) + (x(j) * S(i))
8.     end
9. end
```

Para comprobar el rendimiento de dichos formatos se han ejecutado algoritmos del producto matriz-vector para cada tipo de almacenamiento, tomando tiempos de ejecución y el número de operaciones realizadas (FLOPS) para comparar el comportamiento de cada uno de ellos. Los procedimientos ejecutados se muestran en los algoritmos 3 a 6, diseñados para probar la eficiencia de cada uno de los formatos de almacenamiento. Las pruebas se hicieron para matrices de 300x300, con un nivel de dispersión o densidad variable.

En la tabla 1 se muestra el número de elementos no nulos en cada caso (*nnz*) y los resultados de los FLOPS y el tiempo de ejecución para cada uno de los almacenamientos. Las pruebas se realizaron para el producto matriz-vector debido a que éste es un núcleo computacional básico en los métodos iterativos y en muchas aplicaciones que pueden aprovecharse de estos tipos de almacenamiento. Se utilizó el lenguaje Matlab debido a su facilidad de implementación y el óptimo manejo que se tiene de las matrices, además de traer una implementación del tipo de dato *matriz dispersa* en formato coordenado. Los mismos métodos de almacenamiento se utilizaron también en el lenguaje C y los algoritmos se encuentran en el apéndice A.

TABLA 1
Comparación de costes por flops y tiempo de ejecución para el producto matriz-vector con diferentes tipos de almacenamiento en matlab

dim: 300x300		Flops				Tiempo			
densidad	nnz	densa	coo	csr	csc	densa	coo	csr	csc
10%	8570	180001	17141	26309	26309	0,33	0,05	0,06	0,11
15%	12896	180001	25793	39287	39287	0,38	0,06	0,11	0,16
20%	17232	180001	34465	52295	52293	0,32	0,11	0,17	0,16
25%	21510	180001	43021	65129	65129	0,39	0,05	0,22	0,22
30%	25722	180001	51445	77765	77765	0,33	0,11	0,27	0,27
35%	30016	180001	60033	90647	90647	0,33	0,16	0,28	0,33
40%	34488	180001	68977	104063	104063	0,33	0,17	0,32	0,39
45%	38488	180001	76977	116063	116063	0,32	0,17	0,38	0,39
50%	42840	180001	85681	129119	129119	0,39	0,16	0,44	0,44
55%	47092	180001	94185	141875	141875	0,38	0,22	0,44	0,5
60%	51360	180001	102721	154679	154679	0,33	0,22	0,5	0,54
65%	55806	180001	111613	168017	168015	0,33	0,28	0,54	0,55
70%	59998	180001	119997	180593	180593	0,33	0,27	0,6	0,61
75%	64392	180001	128785	193775	193775	0,33	0,33	0,6	0,66

FIGURA 4
Flops del producto matriz-vector

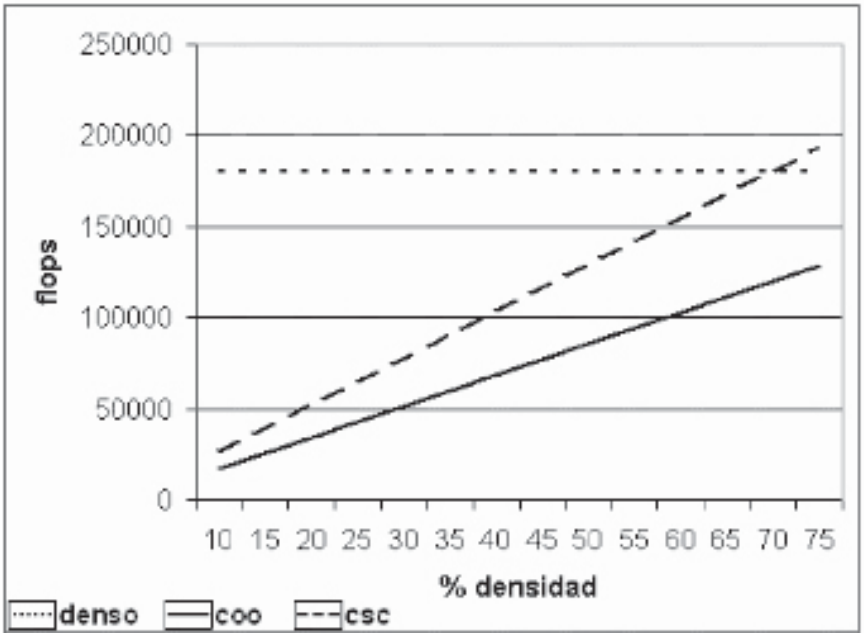
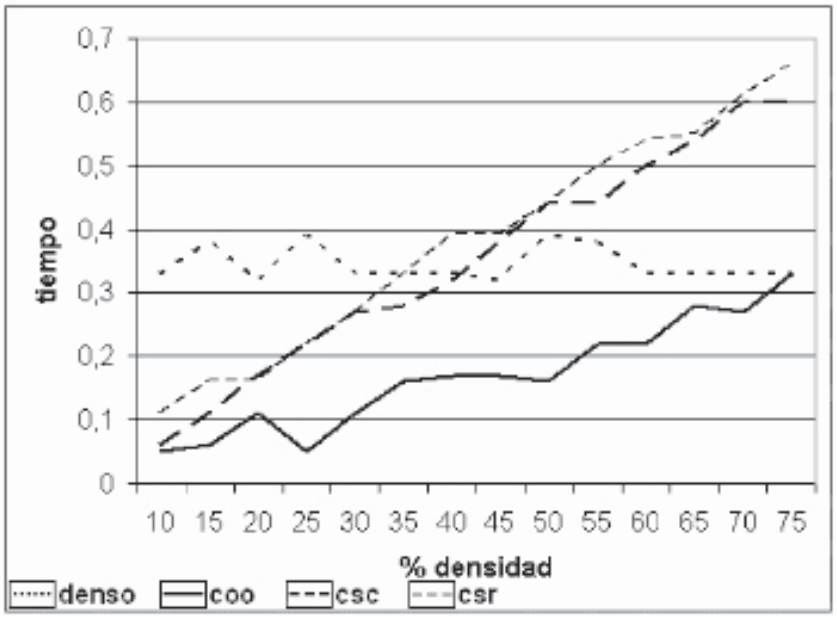


FIGURA 5
Tiempos de ejecución del producto matriz-vector



A partir de los resultados podemos observar en las figuras 4 y 5 que en el formato denso se mantiene una misma velocidad de procesamiento, que es superada por los formatos definidos para matrices dispersas, especialmente cuando la densidad es muy baja. En los formatos de coordenadas y denso solo es necesario hacer 2 operaciones por cada elemento de la matriz (elementos no nulos en el caso del formato COO). En los formatos de CSC y CSR es necesario efectuar 3 operaciones por cada elemento no nulo de la matriz, además del costo de convertir la matriz a estos formatos en las pruebas realizadas.

Igualmente los resultados pueden ser alterados por el manejo óptimo que tiene el lenguaje Matlab con este tipo de matrices en el formato coordenado, pero podría ser diferente si se implementaran en un lenguaje común, ya que el formato COO no tendría las ventajas de la optimización que hace Matlab, o por lo menos ser menor la diferencia entre los diferentes formatos de almacenamiento de las matrices dispersas. Este menor tiempo de ejecución conlleva igualmente a un mayor requerimiento de memoria para el almacenamiento de los vectores del formato coordenado. También influye el hecho de que los formatos comprimidos requieran mayor cantidad de direccionamientos para cada valor, y ciclos anidados, que pueden hacer diferencias en el acceso a la memoria.

Así pues podemos concluir que escoger un formato adecuado para almacenar las matrices con que se trabaje es importante para optimizar al máximo los algoritmos, teniendo en la cuenta que el uso de formatos poco comunes, no aceptados por las distintas librerías que existen, puede traer problemas, o incluso el uso de aquellos formatos comunes de almacenamiento puede agregar un tiempo de ejecución asociado a la conversión de formato de las matrices tantas veces como sea necesario, influyendo en el tiempo global de ejecución de los algoritmos.

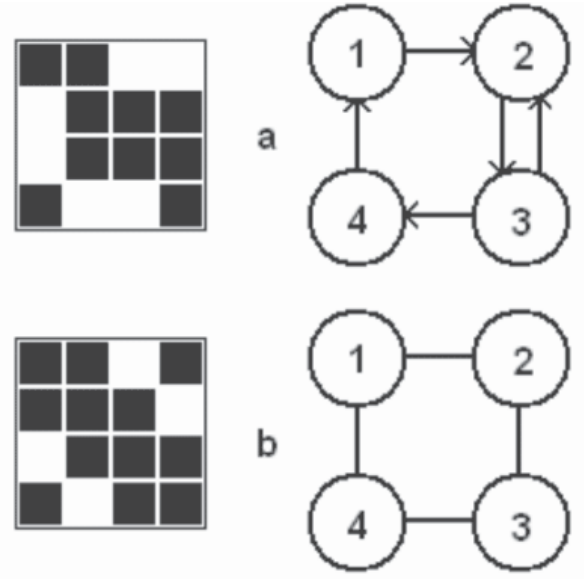
3.2 REPRESENTACIÓN POR GRAFOS

La teoría de grafos constituye una herramienta bastante útil para la representación de matrices dispersas [2], ayudando en la búsqueda de precondicionadores y algoritmos paralelos.

Un grafo se representa formalmente mediante dos conjuntos, un conjunto de vértices $V = \{v_1, v_2 \ldots, v_n\}$ y un conjunto de aristas *E* formado por pares (v_i, v_j) donde v_i y v_j son elementos de *V*. Así, $E \subseteq VxV$.

El grafo *G = (E,V)* puede representarse mediante un conjunto de puntos en el plano que representan el conjunto *E* y unidos por líneas que representan la relación *V*. Cuando esta relación es simétrica el grafo es *no dirigido (figura 6b)*, y en otro caso se debe dar dirección a las relaciones mediante flechas y se dice que el grafo es *dirigido (figura 6a)*.

FIGURA 6
Grafos de matrices dispersas 4x4



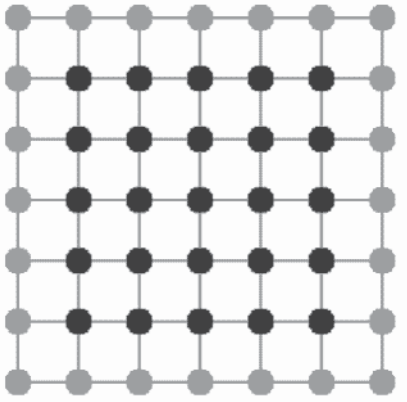
En el caso de las matrices dispersas, el conjunto *V* representa las *n* incógnitas del sistema y *E* es una relación que indica que existe una arista del nodo *i* al nodo *j* cuando $a_{ij} \neq 0$. Así pues, la arista representa la relación "*la ecuación i incluye la incógnita j*". Cuando la matriz es simétrica el grafo es no dirigido. Una de sus aplicaciones puede verse en la implementación paralela de la eliminación Gaussiana, buscando las variables que son independientes en una cierta fase de la eliminación. Estas son variables que no dependen la una de la otra de acuerdo a la relación establecida en *E*. Así, las filas de dichas incógnitas pueden usarse como pivotes simultáneamente [2]. En matrices diagonales cada incógnita es independiente, mientras que en una matriz densa todas dependen de todas; las matrices dispersas están entre ambos extremos.

En el caso de las ecuaciones diferenciales parciales que involucren una sola incógnita física por cada punto de la malla, el grafo de adyacencia resultado de la discretización de las ecuaciones suele ser la malla misma. Este tipo de sistemas se suelen encontrar en problemas de ingeniería, y constituyen una aplicación importante de los métodos para la resolución de sistemas de ecuaciones lineales dispersos.

Gran número de fenómenos físicos son modelados por ecuaciones que relacionan derivadas parciales de magnitudes físicas (fuerzas, momentos, velocidades, temperaturas, etc.). Una de las más comunes es la ecuación de Poisson, que aparece en muchos problemas físicos. Una de las técnicas más desarrolladas para la resolución de este tipo de ecuaciones es el método de diferencias finitas, un método que discretiza un espacio continuo de soluciones obteniendo como resultado un sistema lineal de ecuaciones disperso.

Para una superficie cuadrada donde aplique la ecuación de Poisson se puede discretizar el problema mediante una malla de *(n+2)*(n+2)* puntos, separados a una distancia *h = 1 / (n+1)* entre sí. Como se aprecia en la figura 7, cada punto de la malla es afectado por 4 puntos a su alrededor. Se deben conocer los valores de la función en los puntos frontera y con esta información, discretizar las ecuaciones diferenciales mediante el método de diferencias finitas.

FIGURA 7
Malla con puntos de la discretización de la ecuación diferencial



De la ecuación de Poisson de segundo grado

$$-\frac{\partial^2 v(x,y)}{\partial x^2}-\frac{\partial^2 v(x,y)}{dy^2}=f(x,y)$$

Podemos hacer la aproximación a las derivadas segundas

$$\frac{\partial^2 v}{\partial x^2}(x_i,y_i)\cong\frac{v(x_i-h,y_j)-2v(x_i,y_j)+v(x_i+h,y_j)}{h^2}=\frac{v_{i-1,j}+2v_{ij}-v_{i+1,j}}{h^2}$$

$$\frac{\partial^2 v}{\partial y^2}(x_i,y_i)\cong\frac{v(x_i,y_j-h)-2v(x_i,y_j)+v(x_i,y_j+h)}{h^2}=\frac{v_{i,j-1}+2v_{ij}-v_{i,j+1}}{h^2}$$

$$, u(x_1, y_1) = u_{ij}$$

Así, obtenemos el sistema de n ecuaciones

$$4v_{ij}-v_{i-1,j}-v_{i+1,j}-v_{i,j-1}-v_{i,j+1}=h^2 f_{ij},\quad i,j=1,2,...,n.$$
$$v_{o,j}=v_{n+1,j}=v_{i,0}=v_{i,n+1}=0\quad (frontera)$$

Este es un sistema de ecuaciones disperso, ya que tiene n variables pero cada una de ellas se relaciona a lo sumo con otras 4 variables diferentes y por tanto podemos tratarlo con métodos especiales de almacenamiento para la matriz asociada al sistema de ecuaciones.

Con este método se puede resolver, por ejemplo, el movimiento de una membrana que vibra sobre una dimensión (figura 9), dada unas condiciones iniciales y los valores de la función en la frontera, que en este caso son cero. Las aplicaciones en este campo son numerosas, lo que le da mayor importancia a la optimización de los métodos para resolver los sistemas que resultan. En este caso particular de la ecuación para una superficie, la matriz resultante está formada por solo 5 diagonales. La estructura de esta matriz se muestra en la figura 8.

FIGURA 8
Matriz de discretización de ecuación de Poisson con una malla de 7x7

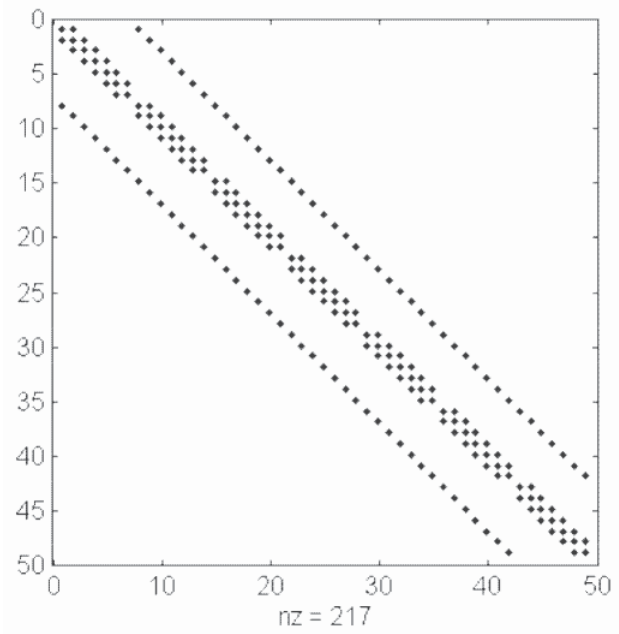
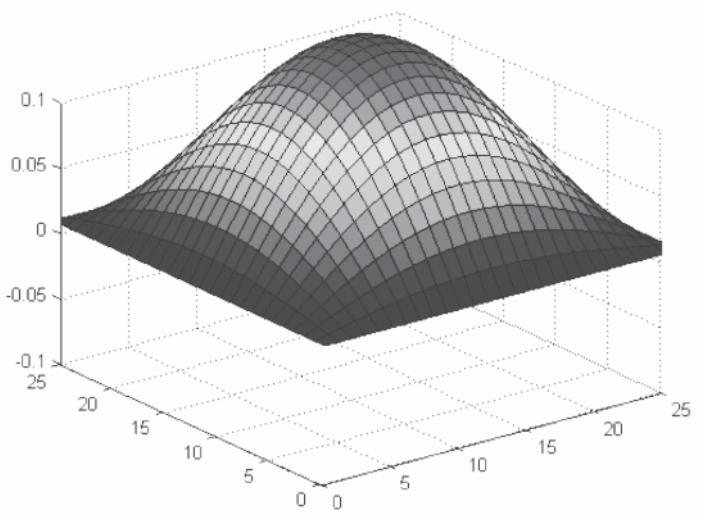


FIGURA 9
Simulación de la vibración de una membrana



3.2.1 Métodos de reordenación

Uno de los pasos importantes en la solución de sistemas dispersos es el reordenamiento de la matriz, mediante una permutación de filas, columnas o ambas. Una permutación puede verse como el producto de la matriz con una permutación de la matriz identidad. Si esta matriz de permutación pre-multiplica la matriz original, se intercambian las filas. Si se post-multiplica la matriz por la matriz de permutación se obtiene una permutación de columnas. Si se hacen ambas multiplicaciones se logra intercambiar tanto las filas como las columnas, obteniendo así una *permutación simétrica,* que es la más usada en los métodos para matrices dispersas. Hay que tener en la cuenta los efectos de dichas permutaciones sobre el sistema de ecuaciones, ya que un intercambio de columnas está cambiando realmente el orden de las incógnitas del sistema. En la teoría de grafos una permutación simétrica de la matriz es equivalente a cambiar las etiquetas de los nodos sin modificar las aristas o la forma del grafo. Estas permutaciones simétricas pueden alterar en gran manera el desempeño de los métodos sin modificar la estructura del grafo.

En las figuras 10 y 11 podemos apreciar un caso extremo del efecto de la reordenación en una matriz dispersa de 9x9 y la matriz que resulta al efectuar una descomposición LU de cada una. Claramente la reordenación de la matriz hace que los elementos que estaban afectando a los demás en el primer paso del algoritmo ahora no tengan un efecto de *fill-in* (llenado de la matriz) y se conserve la estructura dispersa de la matriz. Esta reducción del fill-in es bastante importante en los métodos directos, por lo que es conveniente realizar la reordenación de la matriz previa a la aplicación del método.

FIGURA 10
Matriz dispersa de 9x9 (b), su grafo de adyacencia (a) y resultado de una factorización LU de la matriz (c)

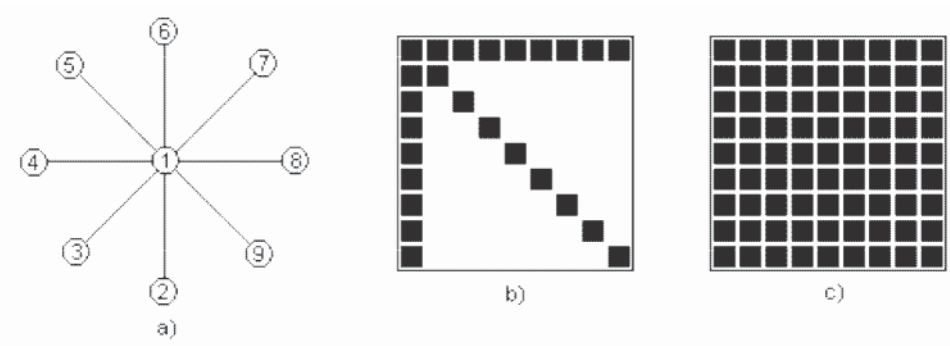
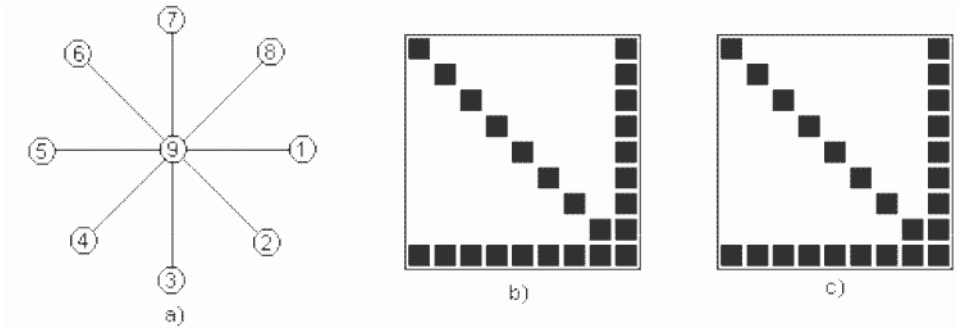


FIGURA 11
Reordenación de la matriz dispersa (b), grafo de adyacencia (a) y el resultado de una factorización LU de la matriz reordenada(c)



Diferentes ordenaciones de la matriz pueden ser más convenientes para métodos directos o para métodos iterativos. Dos de los métodos de reordenación más utilizados son el algoritmo de *grado mínimo* y el de *Cuthill-McKee.*

Algoritmo de Grado Mínimo. Este algoritmo se usa para matrices con estructura simétrica y dispersa. Consiste en hacer una renumeración de los nodos en orden creciente de sus grados respectivos (conexiones de dicho nodo en el grafo asociado) [5]. Los nodos de mayor grado originarán en teoría, un mayor fill-in. La idea es renumerarlos al final para evitar el incremento del fill-in durante el proceso.

ALGORITMO 7. Reordenación de grado mínimo

1. Construir el grafo asociado a la matriz *A, g(x) = (V, E)*
2. **while** $V \neq 0$
3. Elegir un nodo *v* de grado mínimo en *g(x)* y reordenar como siguiente.
4. $V_v = V - \{v\}$
5. $E_v = \{\{a,b\} \in E \mid a,b \in V_v\} \cup \{\{a,b\} a \neq b \mid a,b \in Adj_g(v)\}$
6. % siendo, $Adj_g(v)$ el conjunto de nodos conectados a *v* en el grafo *g(x)* %
7. $V = V_v$
8. $E = Ev$
9. $g(x) = (V, E)$
10. **end while**

Algoritmo de Cuthill-McKee inverso. El algoritmo de Cuthill-Mckee proporciona un método sencillo para reordenar una matriz dispersa con objeto de reducir el efecto fill-in transformando la matriz en una matriz banda. La ordenación resultante al invertir el algoritmo de Cuthill-Mckee resulta frecuentemente mejor que el original [5], en términos de reducción de perfil, manteniendo la anchura de banda.

ALGORITMO 8. Reordenación de Cuthil-McKee inverso

1. Construir el grafo asociado a la matriz *A, g(x) = (V, E)*
2. Determinar un nodo inicial (pseudo-periférico) y renumerarlo como x_1.
3. Renumerar los nodos conectados a x_i en orden ascendente de grado.
4. Efectuar el ordenamiento inverso.

En las figuras 12 a 14 se aprecia el efecto de estas reordenaciones al aplicar una descomposición LU sobre ellas. La elección del algoritmo de reordenación depende tanto del método que se utilizará posteriormente como de la estructura de la matriz original, por lo que un buen entendimiento del problema y de los métodos es siempre importante. En el ejemplo de las figuras 12 a 14 se aprecia que una reordenación de grado mínimo nos reduce casi a la mitad el llenado de la matriz resultante de la descomposición LU. La principal utilidad de reordenar los elementos de una matriz es mejorar las propiedades estructurales de la misma para la aplicación de algún método.

FIGURA 12
Matriz dispersa de 60x60 y su descomposición LU

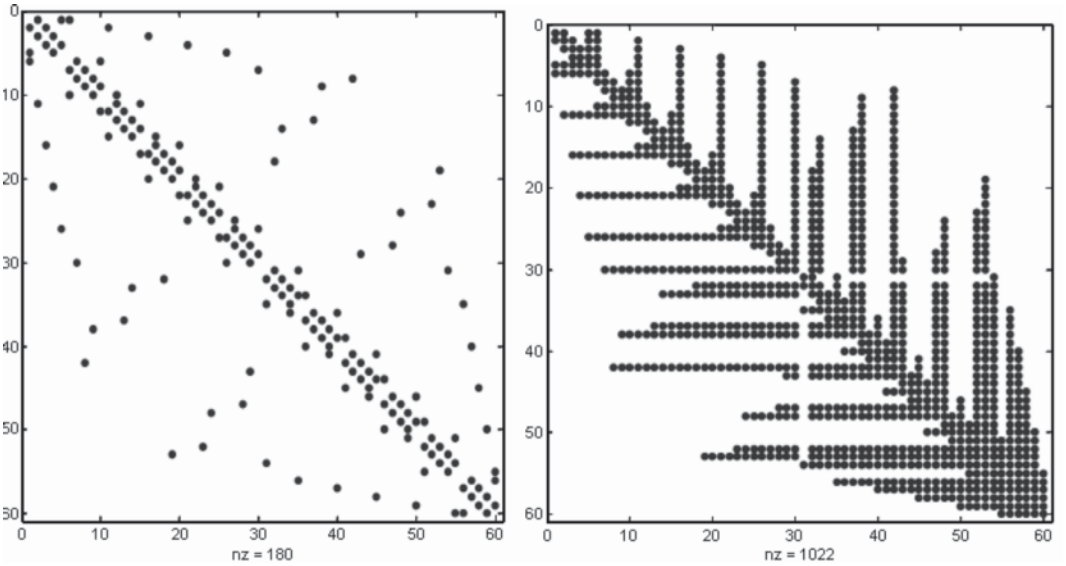


FIGURA 13
Reordenación Cuthill-McKee de la matriz y su descomposición LU

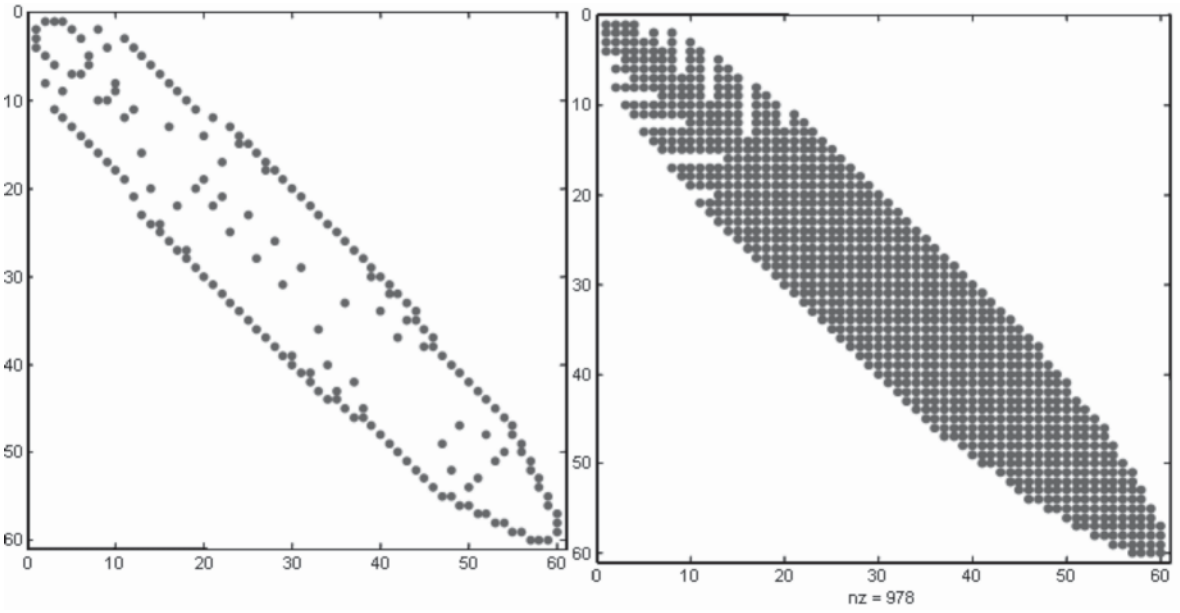
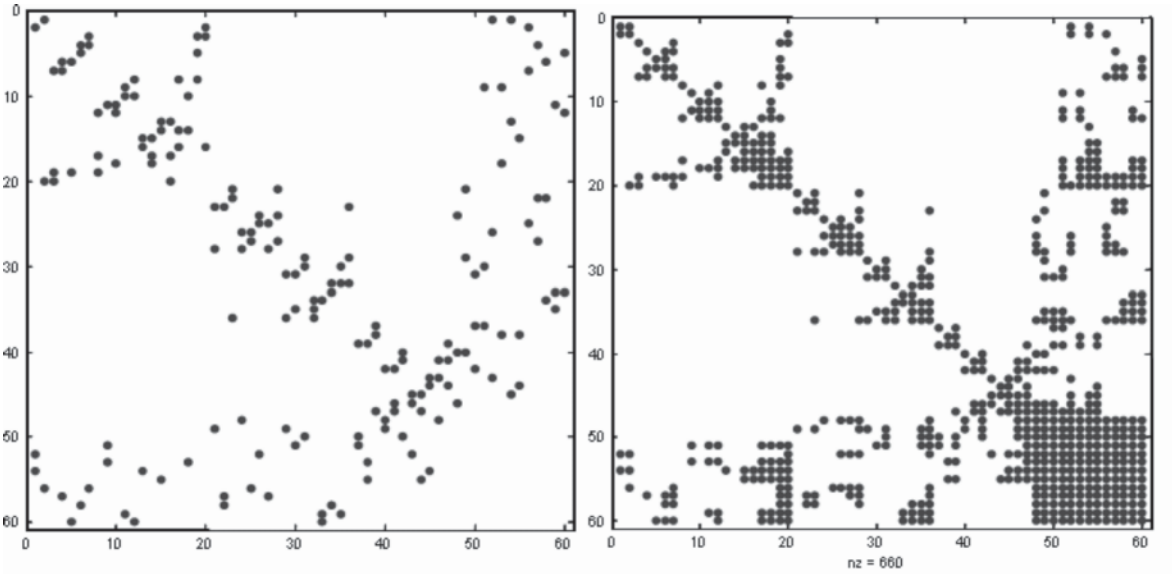


FIGURA 14
Reordenación de grado mínimo de la matriz y su descomposición LU



En función de un método u otro interesará un tipo u otro de reordenación. Para métodos directos interesa una reordenación donde se garantice que el grado de llenado sea mínimo (algoritmo de grado mínimo), mientras que en los métodos iterativos interesa que la reordenación minimice el ancho de banda de la matriz dispersa.

Los métodos directos realizan descomposiciones LU de la matriz intentando minimizar el llenado de la matriz. Las estructuras utilizadas para almacenar los datos hacen que sea más complicada la inserción de los nuevos valores que aparecen con el llenado. En principio se utilizaron listas enlazadas que permiten una inserción más fácil, pero se ha optado por vectores simples, reservando espacio para los elementos que puedan aparecer al hacer la descomposición.

4

MÉTODOS DE RESOLUCIÓN DE SISTEMAS LINEALES

Los métodos para resolver los sistemas lineales se dividen en dos grandes grupos: directos e iterativos. Los métodos directos encuentran la solución en un número finito de pasos y cada paso en las operaciones intermedias depende de los pasos previos. Los métodos iterativos generan una sucesión cuyos valores se espera converjan a la solución del sistema. El error característico en los métodos directos e iterativos es el de redondeo.

A medida que se aumenta la dimensión de los sistemas de ecuaciones, la principal desventaja de los directos frente a los iterativos es que los errores de redondeo se acumulan durante el proceso, además de requerir mayor espacio de memoria debido al efecto de llenado (fill-in). Por otro lado, en procesos donde se tienen que resolver diversos sistemas del mismo tipo, los métodos iterativos pueden aprovechar, como aproximación inicial, la solución obtenida en el paso de tiempo anterior. Todo ello hace que actualmente, en general, se opte por la resolución mediante un método iterativo.

Las técnicas de reordenación, aplicadas principalmente en la resolución de sistemas por métodos directos y que están basadas en la teoría de grafos, proporcionan matrices con ancho de banda o perfil menor, lo que reduce, en la resolución por métodos directos, el coste de almacenamiento en la factorización, pero pueden ser utilizadas por los métodos iterativos para mejorar el efecto del precondicionamiento, que acelera la convergencia del método.

4.1 MÉTODOS ITERATIVOS BÁSICOS

Los métodos iterativos parten de una solución inicial y van modificando sus valores hasta alcanzar una convergencia deseada cuando ésta sea posible. Se dividen en estacionarios y no estacionarios. Los métodos estacionarios son métodos clásicos como Jacobi, Gauss-Seidel o SOR y tienen una convergencia más lenta, aunque su programación es mucho más simple y son más fáciles de entender. Los métodos no estacionarios tienen una mayor convergencia.

La convergencia de los métodos estacionarios depende de algunas propiedades en la matriz inicial como la simetría o que sea definida positiva o diagonal dominante. En general estos métodos estacionarios tienen la forma $Mx^{(k+1)} = Nx^{(k)} + b$ para la solución del sistema lineal $Ax=b$, donde $A=M-N$ y $x^{(k)}$ es el vector aproximación a la solución del sistema en la etapa k. Despejando $x^{(k+1)}$ de la ecuación anterior obtenemos una recurrencia de la forma $x^{(k+1)} = Bx^{(k)} + c$, donde $B = M^{-1}N$ es la matriz de iteración y $c = M^{-1}b$.

La velocidad de convergencia a la solución depende de los valores propios de la matriz de iteración B, y

en concreto, del radio espectral de dicha matriz. El radio espectral ρ de una matriz se define como el valor propio máximo, en valor absoluto, de la misma: $\rho(B) = max\{|\lambda| : \lambda \in \lambda(B)\}$. Si $\rho < 1$, el método converge, y cuanto más próximo esté a 0 mayor será la velocidad de convergencia.

ALGORITMO 9. Versión matricial de los métodos iterativos estacionarios básicos

1. Realizar descomposición $A=L+D+U$
2. Calcular M según el método y $N=M-A$
3. Calcular matriz de iteración $B=M^{-1}N$ y el vector $c= M^{-1}b$
4. $k=0$
5. $x^{(k+1)}=Bx^{(k)} + c$
6. $tol=calc\_tol(x^{(k+1)})$
7. **while** $tol >$ & $k \leq maxiter$
8. $x^{(k+1)}=Bx^{(k)} + c$
9. $tol=calc\_tol(x^{(k+1)})$
10. $k=k+1$
11. **fin while**
12. **if** $k < maxiter$ **then** $x'=x^{(k+1)}$ **else** imprima *'no ha convergido'*

En el paso *2* del algoritmo 9 la matriz M se calcula de la siguiente manera para cada uno de los métodos:

Jacobi:	D.
Gauss-Seidel:	$D-L$.
Sor:	$D - wL$

Y la recurrencia escalar del algoritmo se puede expresar de la forma:

Jacobi $$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_j^{(k)}) \quad , \quad i=1,...,n$$

Gauss-Seidel $$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}) \quad , \quad i=1,...,n$$

SOR $$x_i^{(k+1)} = w \cdot \left[\frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)})\right] + (1-w)x_j^{(k)} \quad , \quad i=1,...,n$$

El algoritmo 10 muestra una implementación del algoritmo SOR en versión escalar utilizando un almacenamiento por columna comprimida (CSR), donde:

A, b: matriz de coeficientes y vector de términos independientes del sistema.

I,J,S: Vectores del formato CSR de la matriz A.

x_o*:* aproximación inicial

maxiter: número máximo de iteraciones (generalmente 1000)

ε: tolerancia deseada expresada como error relativo (8 cifras)

x: solución del sistema

ω*:* factor de relajación

ALGORITMO 10. Versión escalar con formato CSR de algoritmo SOR

1. $nz = length(S)$
2. $nfil = length(x_o)$
3. $x = zeros(nfil)$
4. $k = 0$
5. $err = \varepsilon + 1$
6. while $err > \varepsilon$ & $k \leq maxiter$
7. **for** i = 1 : nfil
8. $x(i) = 0$
9. **for** $j = I(i) : I(i+1) - 1$
10. **if** $i < J(j)$
11. $x(i) = x(i) + S(j) * x_o(J(j))$
12. **else if** $i > J(j)$
13. $x(i) = x(i) + S(j) * x(J(j))$

```
14.                 else
15.                     diag=S(j)
16.                 end if
17.             end for
18.             x(i) = (b(i) - x(i)) / diag
19.             x(i) = w*x(i) + (1-w)*x_o(i)
20.         end for
21.         err = norm(x-x_o) / norm(x)
22.         x_o = x
23.         k = k+1
24.     end while
25.     if k < maxiter
26.         imprimir "Ha convergido en k iteraciones"
27.     else
28.         imprimir "no ha convergido"
29.     end if
```

De los métodos básicos explicados anteriormente el que mejor desempeño suele tener es el SOR, siempre que se haga una elección acertada de ω. En las pruebas de la velocidad de convergencia de los 3 métodos en función del número de iteraciones, tomando para el SOR diferentes valores de ω entre 0 y 2 para encontrar un óptimo, se aprecia este comportamiento. Las pruebas se hicieron para las matrices resultantes de la discretización de ecuaciones diferenciales parciales en 1D, 2D y 3D, así como en matrices aleatorias dispersas y simétricas. Para un valor de ω =1 el método de SOR debe hacer las mismas iteraciones que Gauss. Éste a su vez realiza aproximadamente la mitad de las iteraciones que Jacobi en cada caso.

Los resultados de las pruebas se muestran en las figuras 15 a 20 para los algoritmos de SOR (__), Gauss (-.-.-) y Jacobi (---). rc representa el inverso del número de condición de la matriz. Como se aprecia, algoritmo de SOR tiene un bajo rendimiento para valores ω < 1, lo que nos limita la elección de este valor al intervalo (1,2). Este factor de relajación ω indica el peso que se le da a la solución actual (k) y el peso de la solución anterior (k-1) para pasar a la siguiente iteración.

FIGURA 15
Iteraciones para matriz Poisson 1D de 100x100

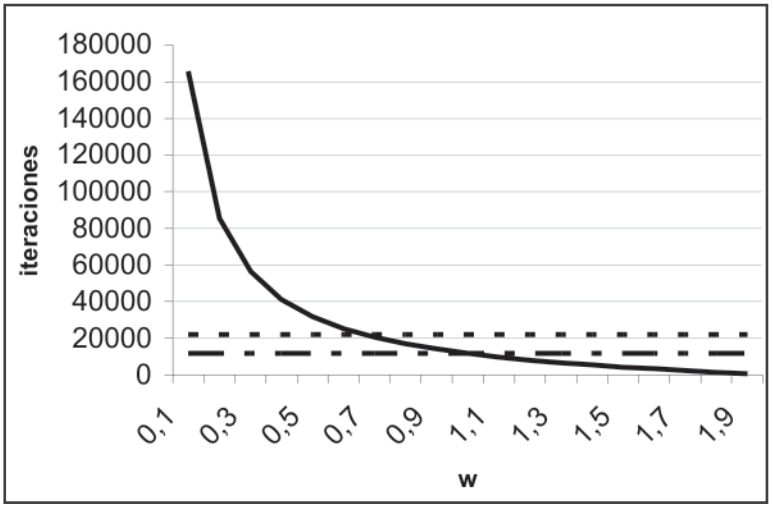


FIGURA 16
Iteraciones para matriz Poisson 2D de 400x400

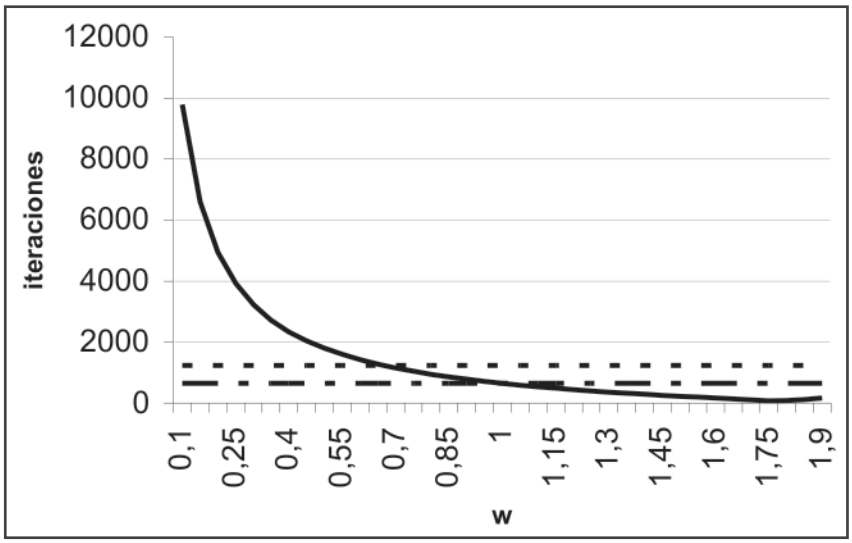


FIGURA 17
Iteraciones para matriz Poisson 3D de 729x729

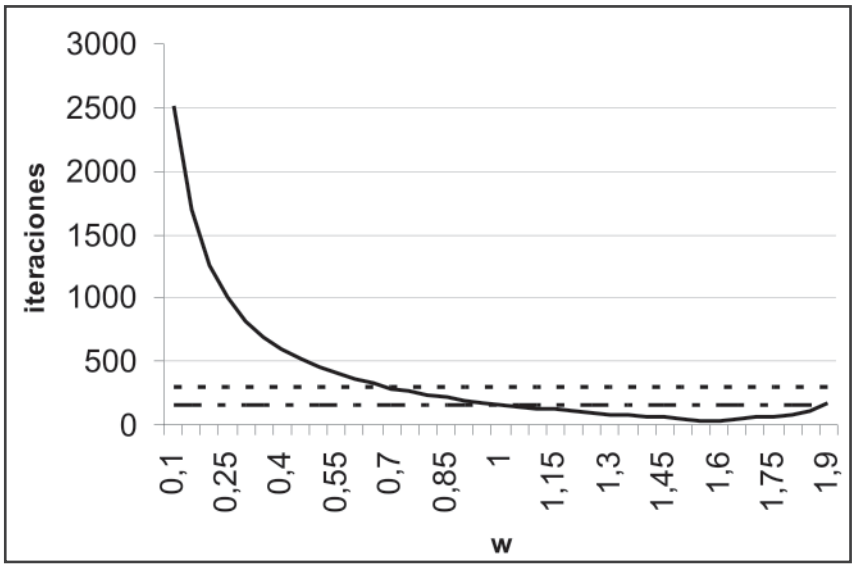


FIGURA 18
Iteraciones para matriz definida positiva de 200x200. rc=0.01

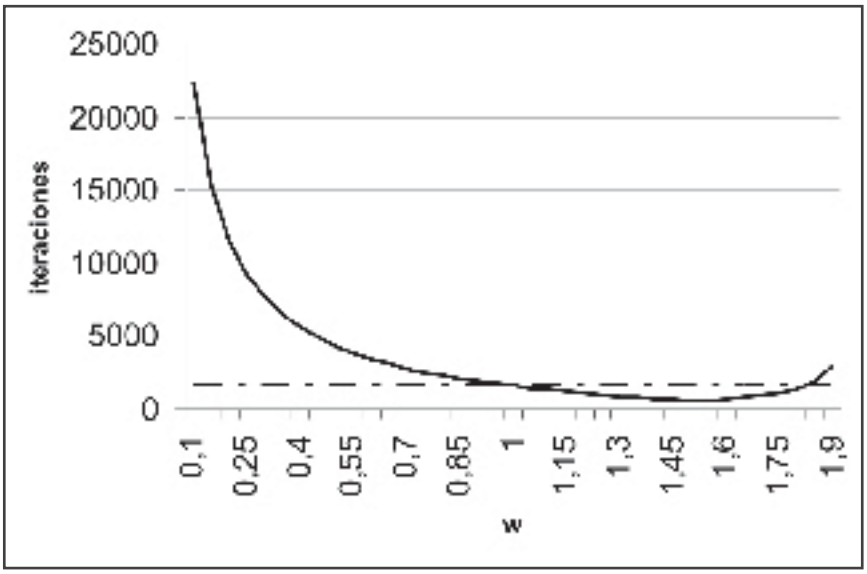


FIGURA 19
Iteraciones para matriz definida positiva de 200x200. rc=0.001

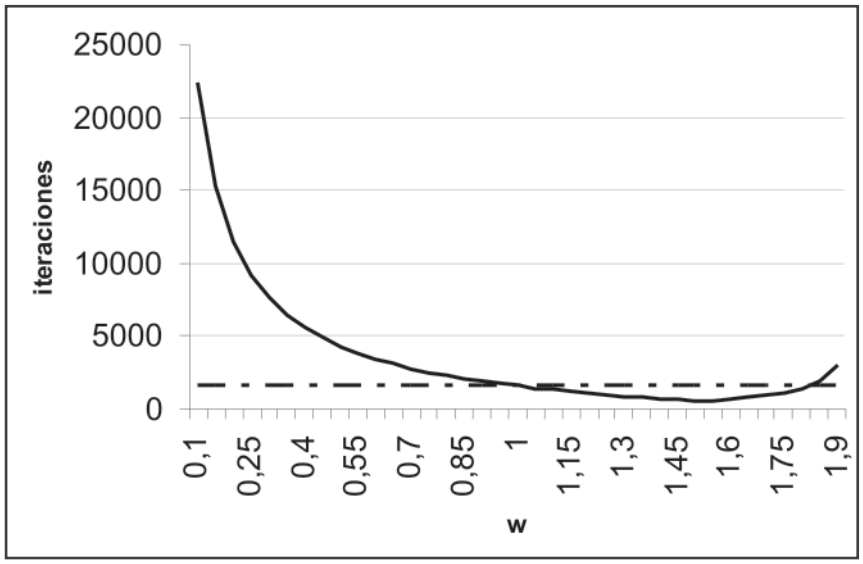
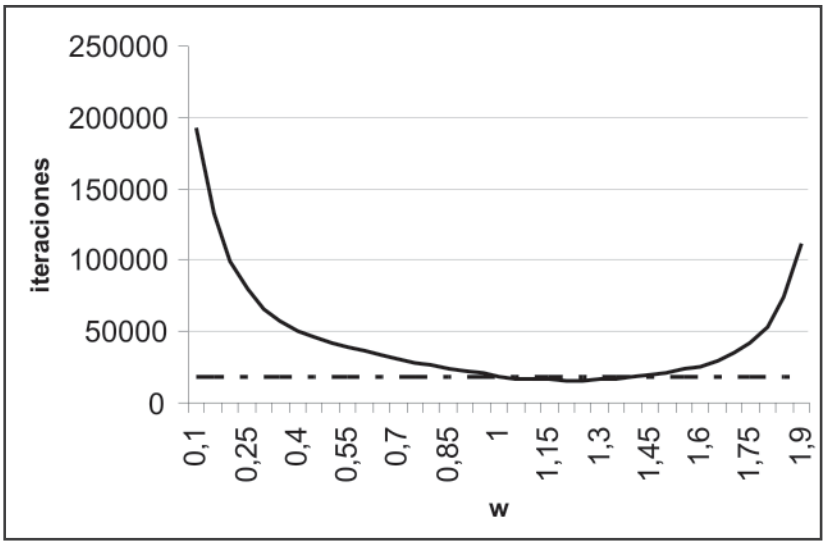


FIGURA 20
Iteraciones para matriz definida positiva de 100x100. rc=0.00001



El algoritmo SOR siempre llega a mejorar el rendimiento del algoritmo Gauss-Seidel cuando el ω utilizado es el óptimo. En caso de que se desconozca este valor o no se tengan los métodos para hallarlo, el uso del método de Gauss-Seidel es más apropiado.

4.2 MÉTODOS ITERATIVOS DE PROYECCIÓN

La mayoría de aplicaciones reales de métodos iterativos para la solución de sistemas de ecuaciones utilizan de alguna manera un método de proyección. Un proceso de proyección representa una forma canónica de encontrar una aproximación a la solución de un sistema lineal de un subespacio.

Consideremos el sistema $Ax=b$, donde A es una matriz de $n_x n$. Los métodos de proyección buscan una solución aproximada de un subespacio en R^n. Si K es el subespacio de búsqueda y m la dimensión, se deben imponer m restricciones para encontrar la aproximación requerida. Dichas restricciones suponen ortogonalidad en m. El vector residuo $b - Ax$ debe ser ortogonal a m vectores independientes. El subespacio de restricciones L distingue 2 tipos de métodos, los ortogonales con L igual a K y los oblicuos con L diferente de K.

Los métodos de proyección pueden ser usados para hallar los valores propios de una matriz, como en el caso de los métodos de Gram-Schmidt, Arnoldi y Lanczos. Mediante estos es posible generar igualmente subespacios de Krylov.

Un método basado en subespacios de Krylov es un método que permite obtener aproximaciones a la solución de un sistema lineal $Ax=b$ en un determinado subespacio. La mayoría de los métodos iterativos utilizados para resolver sistemas lineales de gran dimensión son de este tipo. En todos ellos el principal núcleo computacional es el producto matriz vector.

Estos métodos utilizados para la resolución de grandes sistemas lineales se obtienen para adaptarse, en principio, a dos requerimientos básicos: Por un lado, minimizar una cierta norma del vector residuo sobre un subespacio de Krylov generado por la matriz del sistema y que se traduce en una convergencia suave sin grandes fluctuaciones, y por otro, ofrecer un bajo coste computacional por iteración y no exigir una capacidad excesiva de almacenamiento. Sin embargo, esto no es siempre posible. Sólo en sistemas de ecuaciones lineales cuya matriz de coeficientes es simétrica y definida positiva, el

algoritmo del Gradiente Conjugado propuesto por Hestenes y Stiefel en 1952 y desarrollado en la práctica a partir de 1970, alcanza teóricamente, salvo errores de redondeo, la solución exacta en un número de iteraciones igual a la dimensión del sistema y verifica los requisitos esenciales de minimalidad y optimalidad anteriores. Cuando la matriz del sistema no cumple las condiciones de ser simétrica y definida positiva, el método del Gradiente Conjugado no es aplicable y, en general, no existen métodos que cumplan los dos requisitos anteriores, simultáneamente, sin añadir inconvenientes y desventajas [6]. Por todo ello, los métodos utilizados para estos sistemas, se construyen para adaptarse a uno de ellos, bien al de minimización, o bien al de ofrecer un bajo coste computacional y de almacenamiento. Los podemos agrupar en tres grandes familias: métodos de ortogonalización, métodos de biortogonalización y los métodos basados en la ecuación normal.

4.2.1 Método del Gradiente Conjugado (CG)

El método del gradiente conjugado está basado en una técnica de proyección ortogonal sobre el subespacio de Krylov $K_k(A,r^0)$ donde r^0 es el residuo inicial, y fundamentado en el algoritmo de Lanczos.

Una aproximación x^{j+1} se expresa como $x^{j+1} = x^j + \alpha^j d^j$. Así, los vectores residuales satisfacen $r^{j+1} = r^j - \alpha^j A d^j$ y son ortogonales.

Éste es un gran método para matrices simétricas y definidas positivas. Ya que aprovecha bien la estructura de la matriz y tiene buena estabilidad numérica. Este método iterativo a partir de una solución inicial va calculando sucesivas iteraciones que se van acercando, cuando converge, a la solución exacta del sistema lineal. El iterante *k+1* será la solución, si la diferencia entre él y el iterante *k* es menor que un cierto número prefijado (ε). El método del gradiente conjugado se enfoca como un método de minimización del funcional *J,* convexo y con un único mínimo:

$$J(u) = \frac{1}{2}(Au,u) - (b,u), \forall u \in R^N,$$

siendo *(x, y)* el producto interior de *x* por *y.*

Así para *k = 1* hasta *N-1,* se calcula el mínimo del funcional *J* sobre la variedad lineal $x^0 + \langle d^0,\ldots, d^N\rangle$. Esto es, sea $x^0 \in R^N$, construimos $\{d^0, d^1,\ldots, d^N\}$ base ortogonal de R^N respecto al producto escalar $(A\cdot, \cdot)$. El algoritmo se puede definir pues de la siguiente manera [12]:

Inicialización:
Sea la solución inicial $x^0 \in R^N$ cualquiera.

$$r^0 = b - Ax^0$$

$$d^0 = r^0$$

Iteraciones:
Para k = 0,..., N-1 se hace la minimización correspondiente:

$$\alpha^k = \frac{(r^k, r^k)}{(Ad^k, d^k)}$$

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

$$r^{k+1} = r^k - \alpha^k A d^k$$

$$\beta^{k+1} = \frac{(r^{k+1}, r^{k+1})}{(r^k, r^k)}$$

$$d^{k+1} = r^{k+1} + \beta^{k+1} d^k$$

4.2.2 Método del Residuo Mínimo (GMRES)

El GMRES es un método de proyección sobre un subespacio de Krylov K_k de dimensión *k,* basado en minimizar la norma del residuo. Así, el desarrollo del algoritmo consiste en encontrar un vector *x* de $x_0 + K_k$ tal que, $x = x_0 + V_k y$.

Imponiendo la condición de mínimo para

$$J(Y) = ||b - Ax||.$$

Como

$$b - Ax = b - A(x_0 + V_k y) = r_0 - A_k y$$

y teniendo en cuenta que

$$AV_k = V_k H_k + \omega_k e_k^T = V_{k+1} \overline{H}_k$$

y que

$$\upsilon_1 = r_0 / ||r_0||$$

llamando

$$\beta = ||r_0||$$

entonces,

$$b - Ax = \beta \upsilon_1 - V_{k+1} \overline{H}_k y$$

Pero, $\upsilon_1 = V_{k+1} e_1$, con $e_1 \in R^{k+1}$, por tanto,

$$b - Ax = V_{k+1} (\beta e_1 - \overline{H}_k y)$$

y la condición de mínimo quedará,

$$J(y) = \left\| V_{k+1} (\beta e_q - \overline{H}_k y \right\|$$

Como las columnas de la matriz V_{k+1} son ortonormales por construcción, podemos simplificar la expresión anterior,

$$J(y) = \left\| \beta e_1 - \overline{H}_k y \right\|$$

El algoritmo 11 busca el único vector de $x_0 + K_k$ que minimiza la funcional *J(y)*.

ALGORITMO 11. GMRES

1. Aproximación inicial $x_0 . r_0 = b - Ax_0$
2. $\overline{H}_k = \{H\}_{1 \le i \le k+1, 1 \le j \le k}$. Poner $\overline{H}_k = 0$
3. **for** $j = 1 : k$
4. $\omega_j = A\nu_j$
5. **for** $i = 1 : j$
6. $\{H\}_{ij} = \langle w_j \nu_j \rangle$
7. $w_j = w_j - \{H\}_{ij} \nu_i$
8. **end for**
9. $\{H\}_{j+1,j} = ||w_j||$
10. **if** $\{H\}_{j+1,j} = 0$ **hacer** $k = j$; *parar;*
11. $\nu_{j+1} = w_j / \{H\}_{j+1,j}$
12. **end for**
13. Hallar y_k que minimiza $||\beta e_1 - \overline{H}_k y||$
14. $x_k = x_0 + V_k y_k$, con $V_k ? = [\upsilon_1, \upsilon_2, ..., \upsilon_k]$
15. $r_k = b - Ax_k$

Este algoritmo es inadecuedo para valores grandes de *k,* por lo que se utiliza la siguiente técnica de truncamiento.

ALGORITMO 12. Restarted GMRES

1. Aproximación inicial
   $x_0 r_0 = b - Ax_0$, $\beta = ||r_0||$, $\upsilon_1 = r_0 / \beta$
2. Generar la base de Arnoldi y la matriz $\overline{H}_k$ usando el algoritmo de Arnoldi
3. Iniciar con υ_1
4. Hallar y_k que minimiza $||\beta e_1 - \overline{H}_k y||$ y $x_k = x_0 + V_k y_k$
5. Si se satisface entonces parar. En caso contrario poner $x_0 = x_k$ y volver a 1.

4.3 MÉTODOS DIRECTOS

Los métodos directos son aquellos donde el número de operaciones que se realizan para obtener la solución se conoce de antemano, y permiten realizar la factorización directa en lugar de aproximaciones cada vez mejores, como es el caso de los métodos iterativos. En los métodos directos no tiene importancia el criterio de convergencia como sí lo tiene en los iterativos.

A pesar de su alto costo computacional, los métodos directos son muy útiles para resolver sistemas lineales dispersos ya que son generales y robustos. Hasta ahora son muy pocos los avances que se han logrado en la implementación de métodos directos en paralelo para matrices dispersas, debido al alto costo de comunicaciones que pueden necesitar.

Estos métodos se basan en la eliminación Gaussiana (matrices generales), la factorización de Cholesky (matrices simétricas definidas positivas), entre otros. Sus implementaciones en arquitecturas paralelas

pueden ocasionar bastantes problemas. Los algoritmos 13 y 14 muestran una versión matricial densa de la eliminación Gaussiana (Doolittle) y la factorización de Cholesky.

ALGORITMO 13. Eliminación Gaussiana versión matricial densa

1. **for** $k=1 : n-1$
2. $rows = <k+1 : n>$
3. $A_{rows,k} = A_{rows,k} / A_{k,k}$
4. $A_{rows,rows} = A_{rows,rows} - A_{rows,k} * A_{k,rows}$
5. **end for**
6. $L = I$
7. $U = A$
8. **for** $k=1 : n$
9. $L_{k,1:k-1} = A_{k,1:k-1}$
10. $U_{k,1:k-1} = 0$
11. **end for**
12. y = *sustitución Progresiva (L,b)*
13. x = *sustitución Regresiva (U,y)*

ALGORITMO 14. Factorización de Cholesky versión matricial densa

1. $G = I$
2. **for** $j=1 : n$
3. $v_{j:n} = A_{j:n,j}$
4. **for** $k=1 : j-1$
5. $v_{j:n} = v_{j:n} - G_{j,k} * G_{j:n,k}$
6. **end for**
7. $G_{j:n,j} = v_{j:n} / \sqrt{v_j}$
8. **end for**
9. y = *sustitución Progresiva (G,b)*
10. x = *sustitución Regresiva (G',y)*

El proceso para obtener la solución directa a un sistema general disperso de ecuaciones lineales de la forma $Ax = b$ consiste en cuatro fases: ordenación, factorización simbólica, factorización numérica y solución del sistema triangular.

4.3.1 Ordenación

El ordenamiento de una matriz dispersa es un paso importante en la solución del sistema porque determina la eficiencia de los pasos siguientes. El objetivo en esta fase es generar una permutación de la matriz original tal que la matriz permutada conlleve a una solución más rápida y estable. La estabilidad se logra haciendo los elementos de la diagonal o los pivotes sean de valor grande en comparación de los demás elementos, la velocidad de computación del algoritmo en paralelo es más difícil de mejorar.

Durante la factorización, cuando una fila de la matriz A se resta a otra fila, algunos de los ceros en la segunda pueden tomar valores diferentes a cero. Cuando la *k-ésima* fila es el pivote, un cero en la posición A_{ij} se hace diferente de cero para todo $i, j > k$ tal que $A_{ik} \neq 0$ y $A_{kj} \neq 0$. En estos casos se dice que el elemento A_{ij} es resultado de un *fill-in* (llenado de la matriz).

Además de prever estabilidad numérica y reducir el fill-in, otro objetivo del ordenamiento en implementaciones paralelas de métodos directos es aumentar el número de tareas independientes, esto es usar diferentes filas como pivotes al mismo tiempo. Esto solo es posible en eliminación Gaussiana si la matriz es de estructura dispersa. Lo principal en estos métodos para computación paralela es reducir comunicación entre procesos, aumentar el grado de paralelismo y balancear estos costos con los costos computacionales del fill-in.

El problema de ordenamiento de la matriz para reducir el fill-in tiene una complejidad exponencial, por lo que se utilizan métodos heurísticos para lograr buenas aproximaciones. Algunos de estos algoritmos se explicaron en la sección 3.2.1.

Para la ordenación es posible también utilizar algoritmos y programas ya optimizados para este tipo de problemas. Uno de estos paquetes de programas es el METIS [30], un conjunto de algoritmos que trabaja sobre la partición de grafos. Los ordenamientos producidos por METIS son mejores

que los producidos por otros algoritmos usados comúnmente, incluyendo el algoritmo de grado mínimo. Para muchos problemas de computación científica y programación lineal METIS es capaz de reducir las necesidades de almacenamiento y computación en la factorización de matrices dispersas por un orden de magnitud. Además, a diferencia del algoritmo de grado mínimo, los árboles de eliminación producidos por METIS son válidos para factorización directa en paralelo. Finalmente METIS es bastante rápido y puede reordenar matrices de más de 200,000 filas en unos pocos segundos.

4.3.2 Factorización Simbólica

La fase de factorización simbólica determina la estructura de las matrices triangulares que resultarán de factorizar las matrices de coeficientes ordenadas. Se crea la estructura de las matrices resultantes y se reserva la memoria necesaria para estas.

La factorización simbólica es más complicada si se requiere pivoteo. En estos casos se suele combinar la factorización simbólica con la numérica. Si no se requiere pivoteo, determinar la estructura dispersa de los factores es trivial. En este caso la factorización simbólica determina el fill-in causado por la eliminación de cada fila de la matriz en orden. Como se dijo anteriormente, cuando la *k-ésima* fila es el pivote, un cero en la posición A_{ij} se hace diferente de cero para todo $i, j >$ k tal que $A_{ik} \neq 0$ y $A_{kj} \neq 0$.

Para la factorización simbólica sin pivoteo existen algoritmos seriales con un costo proporcional al número de elementos no nulos de la matriz. Las implementaciones en paralelo suelen ser poco eficientes, por lo que dichos algoritmos son los más usados.

4.3.3 Factorización Numérica

En la factorización numérica es donde se hacen las operaciones aritméticas sobre la matriz de coeficientes *A* para producir la matriz triangular inferior *L* y la matriz triangular superior U. Los algoritmos que utilizamos son la eliminación Gaussiana (para matrices generales) y la factorización de Cholesky (para matrices simétricas y definidas positivas).

En la eliminación Gaussiana para matrices densas se escogen los pivotes secuencialmente, ya que éstos modifican todas las filas de la parte no factorizada de la matriz. Una fila puede ser escogida como pivote solo después de haber sido modificada por el pivote anterior. Sin embargo, en el caso de las matrices dispersas dos filas pueden ser completamente independientes. En una matriz dispersa *A,* la fila *k* modifica directamente la fila i si $i > k$ y $A_{ik} \neq 0$. La fila *k* modifica indirectamente la fila i si una fila j modificada (directa o indirectamente) por *k* modifica la fila *i.* Las filas *i* y *k* son independientes si la fila *k* no modifica *i* directa ni indirectamente.

Las filas independientes de una matriz dispersa pueden ser utilizadas como pivotes en cualquier orden, o incluso simultáneamente en una implementación paralela. Para saber cuáles filas pueden ser utilizadas como pivotes al mismo tiempo se recurre a menudo a la utilización de árboles de eliminación. El tiempo de ejecución dependerá entonces de la altura del árbol. Este grado de paralelismo juega un papel muy importante en la optimización de los métodos directos. Por lo general la ordenación de grado mínimo conlleva a menor fill-in, pero la ordenación por disección anidada genera un árbol mejor balanceado y aumenta el paralelismo.

4.3.4 Solución del sistema triangular

La solución de un sistema triangular requiere poca computación a comparación de la fase de factorización. Por lo general esta fase permite solo un cierto paralelismo dependiendo de las caracerísticas de la matriz asociada, pero aún así puede generar una ganancia en la eficiencia del algoritmo, y todo esto suma para la ejecución total del proceso.

La solución del sistema triangular en paralelo es trivial. Primero se resuelven las ecuaciones con una sola variable, luego se reemplazan estos valores en

las demás ecuaciones y se repite esto hasta que se resuelvan todas las incógnitas. A continuación se ilustra la resolución de un sistema triangular.

Se tiene el sistema triangular $Lx=b$.

$$\begin{bmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

Que corresponde al sistema de ecuaciones

$$\begin{aligned} a_{11}x_1 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= b_4 \end{aligned}$$

De donde se obtiene fácilmente el valor $x_1 = b_1 / a_{11}$. Luego se puede reemplazar este valor en la segunda ecuación para encontrar x_2, y así sucesivamente hasta calcular el valor de todas las incógnitas.

Así, para cada x_i se tiene: $x_i = \dfrac{b_i - \sum_{k=1}^{i-1} a_{ik} x_k}{a_{ii}}$.

5
COMPUTACIÓN PARALELA

Desde su creación, los computadores secuenciales han ido mejorando su velocidad de cómputo de manera constante, pero ahora se empieza a disminuir esta tendencia. Para continuar superando los tiempos pasados se acude entonces al uso de grupos de procesadores para computar datos al mismo tiempo, mejorando la capacidad de procesamiento, sin elevar los costos tanto como otros tipos de desarrollos en este campo. Para una revisión más detallada del tema ver [3, 8, 9, 10, 11].

El concepto de computación paralela puede definirse como una división de una tarea entre varios trabajadores que juntos pueden llevarla a cabo en un tiempo menor al empleado por un único trabajador. Así, una tarea específica puede ser completada en menor tiempo si se divide en subtareas que puedan llevarse a cabo simultáneamente por diferentes trabajadores, estableciendo entre ellos una comunicación. Las características más importantes a tener en la cuenta en este tema son: particionamiento de las tareas, comunicación entre tareas o subtareas, identificación clara de los grados de paralelismo a los que se puede llegar según sea el problema, distribución de las actividades y mecanismos de control.

Para un uso eficiente de la computación paralela se deben tener en cuenta [3] los siguientes aspectos:

- ***Diseño de ordenadores paralelos:*** pensando en la escalabilidad, el rendimiento y el soporte de altas velocidades en la comunicación y compartimiento de los datos.
- ***Diseño de algoritmos eficientes:*** es de poca utilidad un computador en paralelo si no se cuenta con algoritmos paralelos que exploten al máximo su arquitectura.
- ***Métodos para evaluar los algoritmos paralelos:*** debemos tener una forma de analizar el tiempo de respuesta a problemas y la eficiencia en diferentes escalas.
- ***Lenguajes para ordenadores paralelos:*** lenguajes lo suficientemente flexibles que permitan una fácil programación e implementaciones eficientes.
- ***Herramientas de programación paralela:*** alto desarrollo en herramientas para facilitar la comprensión en el momento del desarrollo y la simulación.
- ***Programas paralelos portables:*** portabilidad, disminución o eliminación de cambios requeridos entre diferentes arquitecturas.
- ***Programación automática de ordenadores paralelos:*** generación de compiladores que paralelicen un código que no halla sido escrito en paralelo explícitamente.

5.1 MODELOS DE ORDENADORES PARALELOS

Tradicionalmente los ordenadores secuenciales se basan en la arquitectura de John Von Neumann, donde dicha estructura se conforma por una

unidad central de control (CPU) y una memoria. Los ordenadores que adoptan dicha formación se conocen como *single instruction stream, single data stream (SISD).* La velocidad de dichos computadores tiene dos factores limitantes: la tasa de ejecución de instrucciones y la velocidad a la cual la información se intercambia entre la memoria y la CPU. La segunda se puede mejorar mediante el aumento del número de canales a través de los cuales los datos son accedidos simultáneamente, que se logra mediante el uso de diferentes bancos de memoria que pueden accederse al mismo tiempo. Otra forma de mejorarla es utilizar memorias caché, las cuales son memorias pequeñas, pero de alta velocidad, que sirven como buffer a la memoria principal. Para mejorar la tasa de ejecución de instrucciones se utiliza la ejecución en pipeline, donde se pueden ejecutar instrucciones simultáneamente en diferentes unidades, sobrelapando la búsqueda de una instrucción con la ejecución de la anterior. Sin embargo estas mejoras también tienen sus límites, y el hardware se hace más costoso para mejoras muy pequeñas, por lo que el uso de clusters de computadores para trabajar en paralelo se hace cada vez más popular.

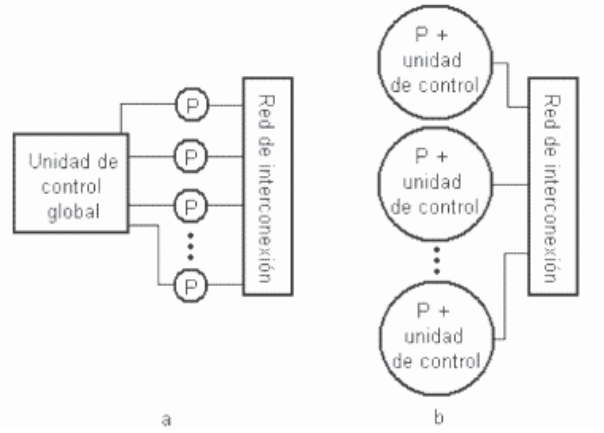
5.1.1 Taxonomía de computadores paralelos

Los computadores paralelos pueden ser construidos de muchas maneras, dependiendo de los mecanismos de control, la organización de espacios de direcciones, la red de interconexión y la granularidad de procesadores.

Mecanismos de control: Las unidades de procesamiento en computadores paralelos pueden operar bajo el control centralizado de una unidad de control o bien trabajar de forma independiente. Así, se pueden dividir en dos grupos de arquitecturas. La primera (figura 21-a) son los computadores *single instruction stream, multiple data stream* (SIMD), donde una unidad de control envía instrucciones a cada unidad de procesamiento. Aquí, una misma instrucción es ejecutada sincrónicamente por todas las unidades de procesamiento. La segunda (figura 21-b) son los computadores *multiple instruction stream, multiple data stream* (MIMD), donde cada procesador es capaz de ejecutar un programa diferente.

Los computadores SIMD requieren menos hardware y menos memoria, y son útiles para programas en donde se requiere aplicar un mismo conjunto de instrucciones a una gran cantidad de datos. En un computador MIMD cada procesador tiene su propia unidad de control, pero pueden ser construidos mediante arreglos de computadores de uso general. Así, los computadores MIMD pueden llegar a ser más rápidos y baratos que los SIMD, aunque necesitan mejores algoritmos de sincronización.

FIGURA 21
Arquitecturas SIMD (a) y MIMD (b)

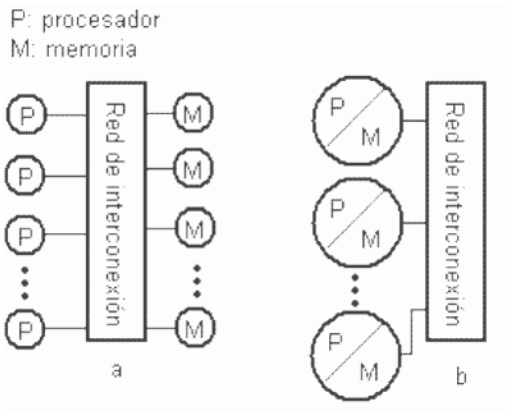


Redes de interconexión: Las redes de interconexión pueden ser estáticas o dinámicas. En las estáticas o directas la comunicación se da punto a punto con vínculos de comunicación o enlaces (links) entre los procesadores como redes de estrella, anillo, árbol o hipercubos. Las redes dinámicas o indirectas son construidas usando switches y enlaces de comunicación para establecer diferentes rutas entre los procesadores, como las redes basadas en bus, multiestado, o tipo crossbar.

Espacio de direcciones: La programación en paralelo requiere de interacción entre los procesadores, y ésta se puede lograr mediante arquitecturas de *paso de mensajes* o de *espacio*

de direcciones compartido. La arquitectura de paso de mensajes tiene que ver con sistemas donde los procesadores están conectados usando una red de interconexión y cada procesador tiene una memoria local (privada), y los procesos de intercambio de datos se realizan por medio de paso de mensajes. A este conjunto de elementos se le suele denominar como multicomputadores. Los computadores con arquitectura de espacio de memoria compartida tienen un hardware que soporta accesos de lectura y escritura de todos los procesadores a los mismos espacios de esta memoria, aunque en ocasiones se implementa con memorias locales que actúan como caché. El intercambio de información se da al modificar los registros que pueden ser leídos por todos los procesadores. Estos equipos se denominan multiprocesadores.

FIGURA 22
Arquitecturas de memoria compartida (a) y paso de mensajes (b)



Granularidad: Los computadores paralelos pueden estar compuestos por pocos procesadores muy potentes, o muchos procesadores de menor capacidad. Los procesos de comunicación y sincronización en la computación paralela crea sobrecargas que disminuyen las prestaciones. La granularidad se define como el tiempo que se utiliza en operaciones de comunicación, sobre el tiempo utilizado en la computación de los datos. Así podemos hablar de grano fino, tarea repartida entre muchos procesadores con un menor tamaño de cada proceso y un mayor número de operaciones de comunicación; o podemos hablar de grano grueso, tarea repartida entre pocos procesadores con un mayor tamaño de cada proceso y un menor número de operaciones de comunicación.

5.1.2 Modelo de un computador paralelo

En un computador paralelo PRAM (parallel random access machine), con memoria compartida y múltiples procesadores, donde hay un único reloj pero los procesadores pueden ejecutar direcciones diferentes al mismo tiempo, hay que manejar de alguna manera las lecturas y escrituras en memoria.

PRAM de lectura exclusiva, escritura exclusiva (EREW): El acceso a una posición de memoria es exclusivo. Es el modelo más débil, permitiendo el mínimo de concurrencia en acceso a memoria.

PRAM de lectura concurrente, escritura exclusiva (CREW): Múltiples lecturas a una posición de memoria son permitidas. Accesos múltiples de escritura son serializados.

PRAM de lectura exclusiva, escritura concurrente (ERCW): Múltiples escrituras a una posición de memoria son permitidas. Accesos múltiples de lectura son serializados.

PRAM de lectura concurrente, escritura concurrente (CRCW): Permite múltiples accesos de lectura y escritura a una posición de memoria.

Permitir lecturas concurrentes no crea discrepancias semánticas en el programa. Sin embargo, accesos de escritura concurrentes requieren arbitración. Algunos protocolos de arbitración son:

- ***Común:*** Se permite la escritura concurrente si todos los valores que se intentan escribir son iguales.

- ***Arbitrario:*** Un procesador arbitrario puede escribir. El resto fallan.

- ***Prioridad:*** Los procesadores tienen un orden de prioridad preestablecido y el de mayor prioridad logra la escritura. El resto fallan.

- ***Suma:*** Se escribe el total de la suma de todos los intentos de escritura. (modificable por cualquier operación).

5.1.3 Redes de interconexión dinámicas

Consideremos la implementación de un PRAM EREW como un computador de memoria compartida con *p* procesadores y *m* palabras de memoria. Los procesadores se conectan a la memoria mediante switches, que determinan la posición accesada por cada procesador. Para reducir la complejidad del sistema de switches se divide la memoria en bancos.

Redes de conmutación de barra transversal: Una manera simple de conectar *p* procesadores y *b* bancos de memoria es mediante un interruptor de barra transversal, que emplea una malla de interruptores o switches. La conexión de un procesador a un banco de memoria no bloquea la conexión de otro. El número de interruptores necesarios es *b* por cada procesador, por lo cual se hace bastante costoso.

Redes basadas en buses: Los procesadores se conectan a una memoria global por medio de un camino común llamado bus. Cuando un procesador accede a la memoria hace un requerimiento del bus, luego los datos son devueltos por el mismo bus. Este sistema es simple pero solo permite un número máximo de datos entre la memoria y los procesadores. Si se aumenta el número de procesadores, cada uno gastará más tiempo esperando por el acceso a memoria. Una manera de reducir este cuello de botella es utilizando memorias locales caché en cada procesador.

Redes de interconexión multietapas: Las redes de interconexión multietapas proveen un equilibrio entre costo y rendimiento que no tienen las anteriores. Una conexión común es la red Omega. Esta red contiene *log(p)* etapas (*p* el número de procesadores y bancos de memoria). Cada etapa es un patrón de interconexiones donde existen vínculos entre una entrada *i* y una salida *j* si:

$$j = 2i, \qquad para\ 0 < i < p/2 - 1$$
$$j = 2i + 1 - p,\ para\ p/2 < i < p - 1$$

En cada etapa hay *p/2* interruptores, y cada uno puede actuar como directo o inverso. Las entradas (procesadores) se comunican con las salidas (bancos de memoria) de manera sencilla: en cada etapa el interruptor actúa como directo si los bits en la posición de la etapa en la entrada y la salida son iguales, y como inverso si son diferentes. En algunos casos se puede presentar un bloqueo de comunicación cuando se requiere usar un mismo camino entre 2 etapas.

5.1.4 Redes de interconexión estáticas

Red completamente conectada: Cada procesador tiene comunicación directa con todos los demás. Es un sistema ideal, ya que cada procesador puede enviar mensajes a otro en un solo paso.

Red estrella: Un procesador actúa como el procesador central. Los demás tienen vínculos de conexión con éste. El procesador central es un cuello de botella.

Arreglo lineal y anillo: Es una manera simple de conectar los procesadores, donde cada uno se comunica con otros 2 (excepto los extremos). Cuando se hace una conexión entre los extremos se refiere como anillo. Los procesadores se comunican enviando mensajes hacia la derecha o izquierda y pasando hasta su destinatario.

Malla: Una malla bidimensional es una extensión del arreglo lineal. Cada procesador se comunica directamente con otros 4. Cuando se comunican los procesadores de los extremos se refiere a él como torus. También se puede extender a una malla tridimensional.

Red de árbol: En una red de árbol existe una única ruta entre dos procesadores. En un árbol estático cada nodo es un procesador, mientras que en los dinámicos los nodos intermedios actúan como interruptores y las hojas son procesadores. Para enviar un mensaje el procesador emisor envía el mensaje hacia arriba en el árbol hasta encontrar el procesador o interruptor del sub-árbol más pequeño que contiene al emisor y al receptor. Luego el mensaje es enviado hacia abajo hasta el receptor. Puede haber cuellos de botella en la parte más alta de los mismos, por lo que suelen utilizarse más conexiones a medida que se acerca a la raíz.

Hipercubo: Es una malla multidimensional con 2 procesadores en cada dimensión. Un hipercubo d-dimensional contiene $p=2d$ procesadores. Se puede construir un hipercubo de manera recursiva, así:

- Un hipercubo 0-dimensional es un procesador.
- Un hipercubo 1-dimensional es la conexión de 2 procesadores o hipercubos 0-dimensionales.
- Un hipercubo (d+1)-dimensional es construido conectando los procesadores correspondientes de dos hipercubos d-dimensionales.

Dos procesadores están conectados directamente si sus representaciones binarias difieren en un solo bit. Cada procesador de un hipercubo d-dimensional está conectado directamente con *d* procesadores. El camino con el menor número de conexiones entre dos procesadores está dado por la distancia de Hamming, que consiste en contar el número de bits que difieren en la representación binaria de ambos procesadores. Esta distancia en un hipercubo d-dimensional será de máximo *d.*

5.1.5 Mecanismos de enrutamiento

Algoritmos eficientes de enrutamiento de mensajes a sus destinos son críticos para el desempeño de un computador paralelo. Un mecanismo de enrutamiento determina el camino que un mensaje sigue por la red desde el emisor hasta el destino. Las entradas son los procesadores fuente y destino, e información del sistema. Las salidas son uno o más caminos posibles. Los algoritmos pueden ser de orden mínimo o no. Los mínimos escogen siempre el camino más corto y pueden provocar congestión. Los no mínimos escogen caminos que pueden ser más largos pero evitan la congestión.

El tiempo que tarda la comunicación entre procesadores es un factor importante en la computación paralela. El tiempo para comunicar un mensaje entre dos procesadores es la latencia, que incluye la preparación del mensaje para el envío y lo que tarda en llegar a su destino. Los principales parámetros para determinar la latencia son el *tiempo de arranque*, que es el tiempo requerido para manejar el mensaje en el procesador fuente incluyendo el algoritmo de enrutamiento; el *tiempo por salto*, que es el tiempo que tarda un mensaje para pasar de un procesador al siguiente; y el *tiempo de transferencia por palabra*, que es el tiempo que toma una palabra para atravesar el vínculo.

5.2 RENDIMIENTO Y ESCALABILIDAD DE SISTEMAS PARALELOS

Un factor importante en un sistema paralelo, es la medición de los niveles de uso de los recursos y la posibilidad que se tiene de aumentar el trabajo, buscando mejorar el desempeño, por lo que es de gran importancia comprender y analizar una serie de parámetros generales que se dan en el momento de la evaluación de los resultados, tiempos y eficiencias. Los algoritmos secuenciales se evalúan según el tiempo de ejecución en función del tamaño de la salida, los algoritmos paralelos no solo dependen de este factor sino también de su arquitectura y del número de procesadores. Algunos de los parámetros que nos ayudan en el análisis sobre determinadas implementaciones, son los siguientes:

Tiempo de Ejecución: a nivel secuencial, es el tiempo transcurrido entre el inicio y el final de la ejecución (T_s). En un ambiente paralelo es denominado el run-time y consiste en el tiempo que transcurre desde

el momento en que el ordenador paralelo comienza, hasta el momento en que el último procesador finaliza la ejecución (T_p).

Aceleración (SpeedUp): es una medida que captura el beneficio relativo de resolver un problema en paralelo, se denomina como S_p y se calcula como T_s/T_p.

Eficiencia*:* en la teoría la eficiencia de tener p-procesadores es *p*, pero en la práctica es difícil porque no se puede obtener el 100% de la productividad para computar un algoritmo. La eficiencia es una medida de la fracción de tiempo para la cual un procesador se mantiene en uso, definido a su vez como una proporción entre la velocidad y el número de procesadores. Se denota con E_p y se calcula como S_p/p. Normalmente la representamos como:

$$\%E_p = \frac{100 * T_s}{T_p * p}$$

Costo: es el producto entre el tiempo de ejecución en paralelo y el número de procesadores usados. El costo también refleja la suma del tiempo que cada procesador gasta resolviendo un problema. La ley de Amdahl dice que la aceleración de un programa paralelo se encuentra limitado por la fracción de tiempo que se consume en realizar operaciones que no se puedan ejecutar en paralelo, por lo cual la aceleración también se encuentra limitada por el tiempo necesario para realizar la comunicación correspondiente a los datos. $T_p = T_s/p + T_c$

Características para la evaluación de problemas paralelos:

Descenso de la escalabilidad: se da cuando se usan menos procesadores que el máximo número posible de ellos al ejecutar un algoritmo paralelo. Un sistema se determina como escalable, si al incrementar el número de procesadores y el tamaño del problema la eficiencia se mantiene igual.

Tamaño del problema: consiste en el número básico de pasos computacionales que se logra en el mejor algoritmo secuencial para resolver un problema sobre un simple procesador. En general se da como el orden del problema a nivel secuencial denotado como *W* y es equivalente al T_s.

Función de sobrecarga: dicha función en un sistema paralelo se da como parte del costo (tiempo de producción del procesador) que no es incurrido por el algoritmo serial más rápido sobre un ordenador secuencial. (Diferencia entre costo y "run-time" serial). Se calcula como $T_0(W, p) = p*T_p - W$.

Grado de concurrencia: máximo número de tareas que pueden ser ejecutadas simultáneamente en un algoritmo paralelo. Se denota como *C(W)* siendo *W* el tamaño del problema, e indica que no más de esta cantidad de procesadores puede ser empleada efectivamente.

Fuentes de sobrecarga paralela: se ve caracterizado por algunos aspectos como la comunicación interprocesador, las cargas desbalanceadas, la extra computación y la distribución de recursos: disco, memoria, entre otros.

Criterios de sobrecarga para tener en cuenta:

1. **Comunicación entre procesadores:** cada procesador gasta un tiempo T_c efectuando las comunicaciones, por lo que la comunicación entre procesadores contribuye un tiempo $T_c * p$ a la función de sobrecarga.

2. **Desbalanceo de cargas:** problemas para conocer el tamaño de las subtareas asignadas a varios procesadores, generando un desequilibrio en las cargas que obliga a que algunos procesadores pasen a estados de inactividad, siendo este tiempo un contribuyente más a la función de sobrecarga. Es entonces $(p-1)*W_s$ el contribuyente a dicha sobrecarga, donde W_s representa los componentes secuenciales del algoritmo.

3. **Extra computación:** el algoritmo secuencial más rápido conocido puede ser difícil de paralelizar,

por lo que se dan algoritmos menos buenos. (alto grado de concurrencia). Entonces $w_0 - w$ es la contribución a la sobrecarga de trabajo extra que se realiza, donde w es el tiempo de ejecución del algoritmo más rápido y w_0 el menos bueno.

Otro aspecto fundamental en las métricas de un ambiente paralelo es la función de i*soeficiencia*, que reúne los anteriores criterios donde el tiempo de ejecución puede ser expresado en función del tamaño del problema, de la función de sobrecarga y del número de procesadores, donde resulta:

$$T_p = \frac{W + T_0(W,p)}{p}$$

Y teniendo en cuenta que:

$$S_p = \frac{W}{T_p} = \frac{W * p}{W + T_0(W,p)}$$

Tenemos la expresión de isoeficiencia:

$$E_p = \frac{S_p}{p} = \frac{W}{W + T_0(W,p)} = \frac{1}{1 + T_0(W,p)/W}$$

De donde podemos obtener,

$$W = \frac{E_p}{1 - E_p} * T_0(W,p) = K * T_0(W,p)$$

5.3 ENTORNOS PARALELOS

Para la computación paralela se debe contar con las herramientas necesarias que nos permitan optimizar realmente los algoritmos y nos ayuden en la búsqueda de soluciones a problemas de gran magnitud. Una forma relativamente barata de hacerlo es mediante la construcción de un cluster de computadoras, utilizando un sistema operativo Linux con entornos MPI o PVM, todo ésto de libre distribución.

La llegada de la computación clusterizada siguiendo el estilo Beowulf, [8] extiende la utilidad de Linux al mundo de la computación paralela de alto rendimiento. Hoy en día, estos valiosos clusters basados en PCs están apareciendo en laboratorios de investigación, departamentos de I+D de empresas, universidades e incluso pequeñas facultades. Si un problema informático no puede solucionarse en un entorno de memoria distribuida flojamente acoplada, un cluster Beowulf puede ser la respuesta, a un precio que los fabricantes tradicionales de computadoras paralelas no pueden llegar.

Construcción de un Beowulf: [9] Cualquiera puede construir un computador paralelo adecuado para la enseñanza de la programación paralela y la ejecución de programas paralelos, muchas veces usando PCs sobrantes o ya existentes. Las PCs en un laboratorio de informática pueden adaptarse para un uso dual gracias a sistemas de encendido dual que les permiten ser encendidas en Linux o Windows, dependiendo de las necesidades del momento. Alternativamente, los equipos no utilizados pueden ser recogidos y fusionados.

Nunca dos clusters Beowulf son iguales. De hecho, sus configuraciones hardware y software son tan flexibles y adaptables que presentan un amplio abanico de posibilidades. Aunque cada cluster es diferente y las configuraciones están dictadas por las necesidades de la aplicación, es posible especificar algunos requerimientos mínimos.

Un nodo tendría que contener, como mínimo, una CPU 486 y una placa madre Intel. Los procesadores Intel 386 funcionarían, pero su rendimiento raramente valdría la pena. Los requerimientos de memoria dependen de los requerimientos de datos de la aplicación y del paralelismo, pero un nodo tendría que contener como mínimo 16MB de memoria. La mayoría de aplicaciones necesitaran 32MB o más por nodo. Usando un espacio de disco centralizado, los nodos se pueden inicializar desde un disquete, un pequeño disco duro o un sistema de archivos en red. Entonces los nodos pueden acceder a su partición raíz desde un servidor de archivos a través de la red, normalmente usando el Network File System (NFS). Esta configuración funciona mejor en un entorno con mucho ancho de banda en las conexiones y con un gran rendimiento en el servidor de archivos. Para un

mejor rendimiento bajo otras condiciones, cada nodo tendría que tener suficiente espacio en el disco local para el sistema operativo, la memoria virtual y los datos. Cada nodo tendría que tener como mínimo 200 MB de espacio de disco para los componentes del sistema operativo y la memoria virtual, pero 400 MB o más permite tener espacio libre que puede ser usado para las aplicaciones en tiempo de ejecución. Como mínimo cada nodo tiene que incluir una tarjeta de red Ethernet o Fast Ethernet. Alternativamente, interconexiones de más alto rendimiento, incluyendo la Gigabit Ethernet y la Myrinet, podrían ser usadas en conjunción con CPUs más rápidas. Finalmente, cualquier tarjeta de video, una lectora de disquetes, la caja y la batería, completan un nodo funcional. Los teclados y los monitores solo se necesitan para la carga y configuración inicial del sistema operativo, a no ser que las máquinas individuales sean usadas interactivamente además de servir como nodos en el sistema paralelo.

Si es posible, los nodos tendrían que estar aislados en una red privada de área local con su propio hub o switch Ethernet. Esto evitará que el tráfico de la red normal interfiera en la comunicación entre los nodos y viceversa. Para incrementar aún más el ancho de banda entre los nodos, se pueden instalar tarjetas de red adicionales en los nodos.

Hay disponibles para Linux tanto compiladores comerciales como gratuitos. En general desarrollar código paralelo requerirá un paso explícito de mensajes entre los procesadores utilizando la PVM (Parallel Virtual Machine - Máquina virtual paralela), MPI (Message Passing Interface - Interfaz de paso de mensajes), u otras librerías de comunicaciones. Ambas, la PVM y la MPI están disponibles gratuitamente y permiten al programador definir fácilmente los nodos usados para ejecutar el código paralelo y pasar datos entre los nodos durante la ejecución usando simples invocaciones a la librería.

Uno de los errores más peligrosos es convertir un problema computacional en un problema de comunicaciones. Esto puede suceder cuando el problema está demasiado dividido, con lo que el tiempo que se necesita para comunicar los datos entre los nodos y sincronizarlos sobrepasa el tiempo de computación real de la CPU. En este caso, usar menos nodos puede resultar en un mejor tiempo de ejecución y una utilización más eficiente de los recursos. Este equilibro entre la carga de computación local en un nodo y las comunicaciones para coordinar esfuerzos entre los nodos tiene que ser optimizado para cada aplicación paralela.

Finalmente, la heterogeneidad de los clusters juega un papel importante en el desarrollo de los algoritmos paralelos. Un factor de dos o más entre las velocidades de las CPUs de los nodos es muy significativo cuando se ejecutan aplicaciones paralelas. Si el trabajo se distribuye uniformemente entre todas las CPUs de un cluster heterogéneo, las CPUs más rápidas tendrán que esperar a las más lentas para completar su parte dentro de la tarea global. Los algoritmos diseñados correctamente pueden superar esta heterogeneidad sobre dividiendo la tarea de forma que se puedan asignar nuevas subtareas a los nodos que completen las anteriores subtareas asignadas.

5.3.1 PVM – Parallel Virtual Machine

PVM es un entorno de programación [10] que permite que un conjunto de máquinas heterogéneas, conectadas en red, puedan ser utilizadas como una única máquina paralela desde el punto de vista de un usuario.

Velocidad computacional: Una máquina PVM puede estar compuesta por computadores con potencias computacionales muy diferentes. Esto ha de ser tenido en cuenta tanto por el sistema PVM como por el programador.

Carga de la máquina: Los computadores que forman parte de una máquina PVM utilizan UNIX. Esto supone que pueden estar ejecutando varios procesos a la vez. Esto influye sobre el rendimiento de las aplicaciones PVM.

Red: La gran mayoría de máquinas UNIX suelen estar conectadas a través de la red Ethernet. El rendimiento de la máquina PVM se verá afectado en gran manera en función del tipo y de la carga de la red. El tipo puede ser Ethernet (10 Mbit/s), Fast Ethernet (100Mbit/s), FDDI (100 Mbit/s), HiPPI (800 Mbit/s ó 1.6Gbit/s), ATM, entre otras. La carga depende del tráfico y número de máquinas conectadas a la red.

PVM utiliza el modelo de programación de los computadores MPP y de la computación distribuida mediante paso de mensajes, pero también puede utilizarse para computadores con Memoria Compartida, simulando la gestión de mensajes a través de memoria compartida. Sin embargo los programas deben ser diseñados según el modelo de computación distribuida.

PVM maneja de forma transparente al usuario el encaminamiento de mensajes, la conversión de datos entre computadores y la gestión de tareas (tasks). El usuario escribe su aplicación como un conjunto de tareas que cooperan entre sí, y éstas acceden a los recursos PVM a través de una librería de rutinas, que permiten iniciar tareas (pvm_spawn), finalizar tareas (pvm_kill, pvm_exit, ...), comunicación entre tareas, sincronizar tareas, agrupar tareas.

El entorno PVM está compuesto de tres partes:

- El **daemon** (demonio) conocido como pvmd o a veces pvmd3, que reside en todos los computadores que forman parte de la máquina virtual. Su misión principal es crear la máquina virtual, de tal forma que una aplicación de un determinado usuario pueda acceder a todos los computadores (hosts). Cuando un usuario desea ejecutar una aplicación PVM, en primer lugar crea la máquina PVM arrancando el entorno PVM, con lo que arrancan todos los daemon´s de todos los computadores. Una vez han arrancado todos los daemons en todos los hosts y se ha efectuado la configuración de la maquina virtual, el usuario puede arrancar la aplicación PVM desde cualquiera de los hosts.

- Una **librería de funciones** que contiene las primitivas del sistema PVM para paso de mensajes, arranque (spawn) de procesos, coordinación de tareas y manejo de la máquina virtual. Acepta llamadas de los lenguajes Fortran, C y C++. Está compuesta por 3 ficheros que contienen la librería en sí: libpvm.a (para C , C++ y Fortran), libfpvm.a (para Fortran), libgpvm3.a (para grupos). Y varios ficheros que contienen definiciones, constantes del sistema y algunas variables: pvm3.h, fpvm3.h (Fortran), pvmsdpro.h, pvmtev.h.

- Y las **tareas de usuario**, donde un conjunto de ellas constituye la aplicación PVM. Cada tarea PVM es identificada mediante un identificador de tarea único TID (Task Identifier) dentro de la máquina virtual y proporcionado por ésta. El destino de un mensaje, se referencia mediante el identificador de la tarea a la cual se le quiere enviar dicho mensaje. PVM contiene varias rutinas para obtener el TID de una determinada tarea. También es posible construir grupos de tareas.

Ejemplo de programación en PVM:
Producto matriz-vector

Se ha implementado un algoritmo que realiza el producto de una matriz y un vector dados, suponiendo que cada procesador guarda un bloque de filas de la matriz y un bloque igual del vector.

La implementación consta de dos procesos: un maestro y un esclavo. El proceso maestro genera la matriz y el vector y las envía a los esclavos. La matriz la envía por bloques, y el vector completo. Luego espera a que los esclavos retornen sus resultados y los almacena en un nuevo vector. Finalmente realiza el producto en secuencial para comparar resultados y calcular el tiempo del algoritmo secuencial. Los esclavos reciben un bloque de la matriz y el vector completo, pero como en métodos iterativos se utiliza este producto repetidamente y los resultados quedan repartidos por bloques en cada procesador, se ha hecho una multidifusión de cada bloque del vector, para que cada procesador cuente con el vector completo. Antes de esta multidifusión se crea una

barrera para sincronizar los esclavos, luego se toma el tiempo, se realiza la multidifusión, se realiza el cálculo del producto por parte de cada esclavo, y se toma nuevamente el tiempo, antes de retornar los resultados obtenidos.

Las pruebas se han realizado con un tamaño de la matriz variando entre 50 y 2000, y para 1, 2, 4, 6 y 8 procesadores, tanto en computadores de una sala con libre acceso a Internet, como en el cluster de uso exclusivo. Se analizaron también las diferentes topologías de bus y en anillo. Para los cálculos teóricos se utilizaron los datos obtenidos en la práctica anteriormente:

β **= 0.000193**

τ **= 1,0905 $_x$ 10^{-5}**

El parámetro β representa el tiempo para establecer la conexión entre dos procesadores y τ representa el tiempo de transferencia de un byte por la red.

Se tomaron los tiempos experimentales, se calcularon los tiempos teóricos, se calculó la eficiencia, y se analizó el efecto de todas las posibles variantes de la ejecución.

Para la eficiencia se utilizó la siguiente fórmula:

$$E_p = \frac{1}{1+\frac{pT_p - T_s}{T_s}} = \frac{1}{1+\frac{p(P\beta+\tau n)}{2n^2F}}$$

$$= \frac{1}{1+\frac{p^2\beta+p\tau n}{2n^2F}} = \frac{T_s}{pT_p}.$$

Los tiempos teóricos se calcularon así:

$$T_{teorico} = \frac{2\cdot n^2\cdot F}{p} + p\cdot\beta + n\cdot\tau.$$

ALGORITMO 15. Producto matriz-vector en PVM – Programa Maestro

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include “pvm3.h”
4.  #include “cab2.h”
5.  #include “ctimer.c”
6.  #define SLAVENAME “prodmvS”
7.
8.  main(int argc[], char *argv[])
9.  {
10.     int mytid, tid, cc;
11.     int tids[32],i,j, who, msgtype, nhost, narch,
        numt;
12.     int n,m,p,tb,tb_ult;
13.     struct pvmhostinfo *hostp;
14.     double *A, *b, *b2, *x;
15.     double elapsed, ucpu, scpu, time, t, tmax;
16.     p = (int)atoi(argv[1]);
17.     m = n = (int)atoi(argv[2]);
18.
19.    A = (double *) malloc(n*m * sizeof(double));
20.    x = (double *) malloc(n * sizeof(double));
21.    b = (double *) malloc(n * sizeof(double));
22.    b2 = (double *) malloc(n * sizeof(double));
23.
24.  for (i=0;i<n;i++){
25.      x[i]=1;
26.      for (j=0;j<m;j++)
27.           A(i,j,m)=i;
28.  }
29.
30.  /* enroll in pvm */
31.  mytid = pvm_mytid();
32.
33.  /* start up slave tasks */
34.     numt=pvm_spawn(SLAVENAME,
        (char**)0, 0, “”, p, tids);
35.  if (numt < p){
36.        printf(“\n Trouble spawning slaves.
          Aborting. Error codes:\n”);
37.         for( i=numt ; i<p ; i++ ) {
38.         printf(“TID %d %d\n”,i,tids[i]);
39.         }
40.         for( i=0 ; i<numt ; i++ ){
41.         pvm_kill( tids[i] );
42.         }
43.         pvm_exit();
44.         exit(1);
45.      }
```

```
46.       tmax = 1000000;
47.   /* Enviar parámetros iniciales */
48.       distri_param (p, tids, n, &tb, &tb_ult);
49.   /* Enviar matriz A */
50.       distri_matriz_bloq_f (p, n, tb, tb_ult, A, n,
          tids);
51.   /* Enviar vector x */
52.       pvm_initsend(PvmDataRaw);
53.       pvm_pkdouble(x, n, 1);
54.       pvm_mcast(tids, numt, 10);
55.   /* Recibir vector resultado b y tiempos */
56.    for (i=0;i<numt-1;i++) {
57.            pvm_recv(tids[i],15);
58.            pvm_upkdouble(&b[i*tb], tb, 1);
59.            pvm_upkdouble(&t, 1, 1);
60.        if (t < tmax)
61.            tmax = t;
62.       }
63.       pvm_recv(tids[numt-1],15);
64.       pvm_upkdouble(&b[(numt-1)*tb], tb_ult, 1);
65.       pvm_upkdouble(&t, 1, 1);
66.       if (t < tmax)
67.            tmax = t;
68.       ctimer(&elapsed, &ucpu, &scpu);
69.       prod(n,m,A,x,b2);
70.       ctimer(&elapsed, &ucpu, &scpu);
71.       /* Program Finished exit PVM before
          stopping */
72.       pvm_exit();
73.   }
```

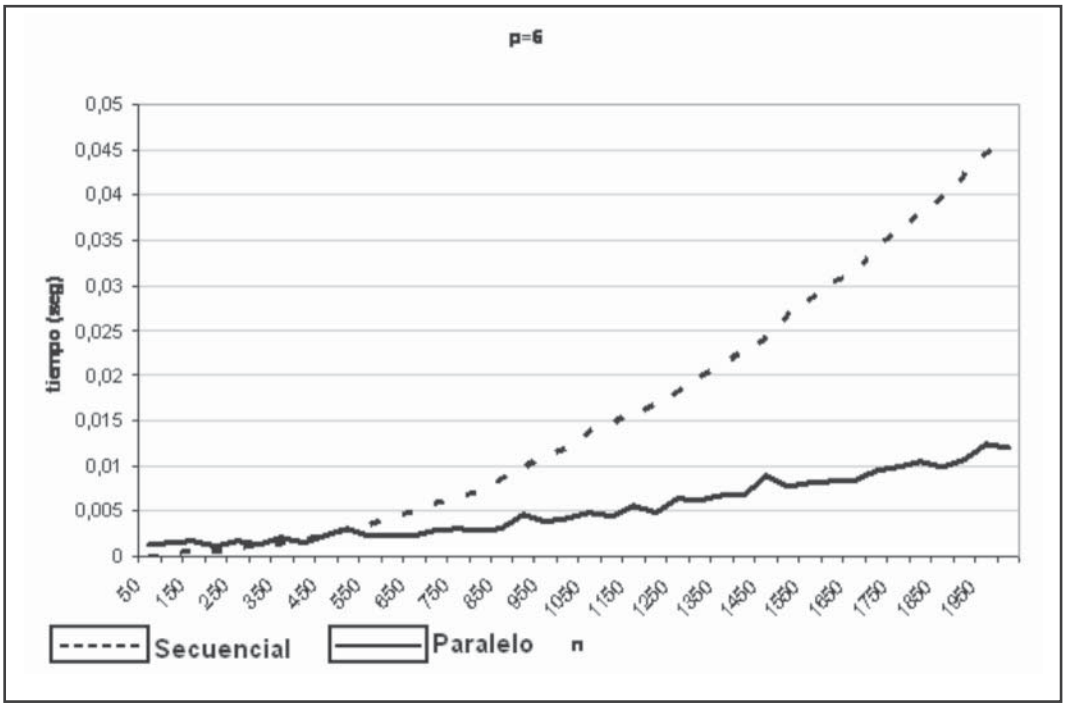
ALGORITMO 16. Producto matriz-vector en PVM – Programa Esclavo

```
1.    #include <stdio.h>
2.    #include <stdlib.h>
3.    #include “pvm3.h”
4.    #include “cab2.h”
5.    #include “ctimer.c”
6.    #define GROUPNAME “PROD”
7.
8.    main(char *argv[])
9.    {
10.       int mytid, me, myparent, cc;
11.       int tids[32],i,j, who, msgtype, nhost, narch,
          numt;
12.       int n,m,p,tb,tb_ult;
13.       struct pvmhostinfo *hostp;
14.       double *A, *b, *x;
15.       double elapsed, ucpu, scpu, time;
16.   /* Parámetros iniciales */
17.       mytid = pvm_mytid();
18.       myparent = pvm_parent();
19.       pvm_joingroup(GROUPNAME);
20.       recibe_param (mytid, &me, &p, tids, &n,
          &tb, &tb_ult);
21.   /* Reservar espacio para matrices */
22.       if (me==p-1)
23.            A = (double *) malloc(tb_ult*n *
              sizeof(double));
24.       else
25.            A = (double *) malloc(tb*n *
              sizeof(double));
26.       x = (double *) malloc(n * sizeof(double));
27.       b = (double *) malloc(n * sizeof(double));
28.       recibe_matriz_bloq_f (p, me, n, tb, tb_ult,
          A);
29.       pvm_recv(myparent,10);
30.       pvm_upkdouble(x, n, 1);
31.       pvm_barrier(GROUPNAME, p);
32.       ctimer(&elapsed, &ucpu, &scpu);
33.       distr_vector (m, n, p,tb,tb_ult,tids,me,A,m,
          x,b); /* Algoritmo 17 */
34.       if (me==p-1)
35.            tb=tb_ult;
36.       prod(n,tb,A,x,b);
37.       ctimer(&elapsed, &ucpu, &scpu);
38.       pvm_initsend(PvmDataRaw);
39.       pvm_pkdouble(b, tb, 1);
40.       pvm_pkdouble(&elapsed, 1, 1);
41.       pvm_send(myparent, 15);
42.    /* Program finished. Exit PVM before stopping */
43.       pvm_lvgroup(GROUPNAME);
44.       pvm_exit();
45.   }
```

ALGORITMO 17. Producto matriz-vector en PVM – Distribución del vector

```
1.  void distr_vector(int m, n, p, tb, tb_ult, *tids, me,
    maxcol, double *A, *x, *b){
2.  int who, i, j;
3.  /* Distribución de subvector x */
4.   pvm_initsend(PvmDataRaw);
5.   pvm_pkint(&me,1,1);
6.       if (me==p-1)
7.        pvm_pkdouble(&x[me*tb],tb_ult,1);
8.       else
9.            pvm_pkdouble(&x[me*tb],tb,1);
10.      pvm_mcast (tids, p, 5);
11.  /* Recibir vector x */
12.      for (i=0;i<p-1;i++){
13.          pvm_recv(-1,5);
14.          pvm_upkint(&who,1,1);
15.          if(who == p-1)
16.              pvm_upkdouble(&x[who*tb],  tb_ult,
                 1);
17.          else
18.              pvm_upkdouble(&x[who*tb], tb, 1);
19.      }
20.  }
```

FIGURA 23
Tiempo de ejecución en secuencial y paralelo con 6 procesadores



5.3.2 MPI – Message Passing Interface

MPI es un estándar que ha sido implementado por numerosos desarrolladores sobre diferentes plataformas. MPI funciona sobre Linux, MS Windows, IRIX, AIX, HP-UX y muchos otros sistemas operativos, utilizando de forma optimizada los recursos de cada máquina.

Existen implementaciones específicas para sistemas concretos, desarrolladas por los fabricantes y centros de investigación, que permiten aprovechar al máximo las prestaciones de una máquina determinada. La compatibilidad entre versiones es un hecho, lo que ha permitido una amplia portabilidad del software desarrollado sobre MPI.

La versión más utilizada es mpich, desarrollada por el Laboratorio Nacional de Argonne y la Universidad de Chicago. Esta versión, de libre distribución, ha sido portada a numerosos sistemas operativos y máquinas, incluyendo clusters Linux y Windows. La segunda versión más utilizada y que proporciona muy buenos resultados bajo cluster es LAM (Local Area Multicomputer), desarrollada por el Centro de Supercomputación de Ohio.

El objetivo básico de MPI [11] es desarrollar un estándar de amplia utilización para escribir programas de paso de mensajes. Esta interfaz intenta establecer un práctico, portable, eficiente y flexible estándar para el paso de mensajes. En el diseño del MPI se utilizó en gran medida el "MPI Forum" de forma que se tomaba en cuenta la opinión de los programadores. Ha sido influenciado en gran medida por el trabajo en el IBM T. J. Watson Research Center, Intel's NX/2, Express, nCUBE's Vertex, p4, y PARMACS. Otras contribuciones importantes provienen de Zipcode, Chimp, PVM, Chamaleon y PICL. La principal ventaja de establecer un estándar en el paso de mensajes es la portabilidad y la facilidad de uso. En un sistema de memoria compartida en el cual las rutinas de más alto nivel y las abstracciones están construidas sobre una capa de bajo nivel de paso de mensajes los beneficios de la estandarización son particularmente aparentes. Además permite que se ofrezca soporte hardware y se aumente la escalabilidad.

MPI ofrece varias características al usuario:

Operaciones colectivas: una operación colectiva es una operación ejecutada por todos los procesos que intervienen en un cálculo o comunicación. En MPI existen dos tipos de operaciones colectivas: operaciones de movimiento de datos y operaciones de cálculo colectivo. Las primeras se utilizan para intercambiar y reordenar datos entre un conjunto de procesos. Un ejemplo típico de operación colectiva es la difusión (broadcast) de un mensaje entre varios procesos. Las segundas permiten realizar cálculos colectivos como mínimo, máximo, suma, OR lógico, etc., así como operaciones definidas por el usuario.

Topologías virtuales: ofrecen un mecanismo de alto nivel para manejar grupos de procesos sin tratar con ellos directamente. MPI soporta grafos y redes de procesos.

Modos de comunicación: MPI soporta operaciones bloqueantes, no bloqueantes o asíncronas y síncronas. Una operación de envío bloqueante bloquea al proceso que la ejecuta sólo hasta que el buffer pueda ser reutilizado de nuevo. Una operación no bloqueante permite solapar el cálculo con las comunicaciones. En MPI es posible esperar por la finalización de varias operaciones no bloqueantes. Un envío síncrono bloquea la operación hasta que la correspondiente recepción tenga lugar. En este sentido, una operación bloqueante no tiene por qué ser síncrona.

Soporte para redes heterogéneas: los programas MPI están pensados para poder ejecutarse sobre redes de máquinas heterogéneas con formatos y tamaños de los tipos de datos elementales totalmente diferentes.

Tipos de datos: MPI ofrece un amplio conjunto de tipos de datos predefinidos (caracteres, enteros, números en coma flotante, etc.) y ofrece la posibilidad de definir tipos de datos derivados. Un tipo de datos

derivado es un objeto que se construye a partir de tipos de datos ya existentes. En general, un tipo de datos derivado se especifica como una secuencia de tipos de datos ya existentes y desplazamientos (en bytes) de cada uno de estos tipos de datos. Estos desplazamientos son relativos al buffer que describe el tipo de datos derivado.

Modos de programación: con MPI se pueden desarrollar aplicaciones paralelas que sigan el modelo SPMD o MPMP. Además el interfaz MPI tiene una semántica multithread (MT-safe), que le hace adecuado para ser utilizado en programas MPI y entornos multithread (multi-hilos).

Ejemplo de programación en MPI: Producto matriz-matriz

Se implementó un algoritmo para realizar el producto de dos matrices con diferentes dimensiones. La entrada del programa son las dimensiones *n, q, m* de dos matrices A_{nxq} y B_{qxm}, y la salida es el producto de ellas, una matriz C_{nxm}. En la programación se ha utilizado el lenguaje C con el entorno MPI bajo Linux.

El proceso principal genera las matrices con las dimensiones que el usuario ingresa, luego reparte la matriz *A* por filas a todos los procesadores y la matriz *B* completa. Cada procesador hace el cálculo correspondiente y le devuelve el valor hallado al proceso principal que los organiza, y comprueba con un algoritmo secuencial que el resultado sea correcto.

Algoritmo 18. Producto matriz-matriz en c y MPI

```
1.  #include <stdlib.h>
2.  #include “opciones.h”
3.  #include <math.h>
4.  #include <mpi.h>
5.  #include “ctimer.c”
6.  #define DEFAULT_SIZE 1000
7.  #define EPS 1e-12
8.  #define COORD(i,j,n) ((i)*(n)+(j))
9.  void vPrintVecDist(int my_rank, *pnProcRow,
    *pnGlb2Lcl, nDim, double *pdV) {
10.     int i;
11.     for (i=0;i<nDim;i++) {
12.         if (my_rank == pnProcRow[i]) {
13.             printf(“%f\n”, pdV[pnGlb2Lcl[i]]);
14.         }
15.     }
16. }
17. void vPrintVec(double *pdV, int nDim) {
18.     int i;
19.     for (i=0;i<nDim;i++) {
20.         printf(“%f\n”, pdV[i]);
21.     }
22. }
23. void vPrintMatDist
24.         (int my_rank, *pnProcRow, *pnGlb2Lcl,
        nDimF, nDimC, double *pdM) {
25.     int i,j;
26.     for (i=0;i<nDimF;i++) {
27.         if (my_rank == pnProcRow[i]) {
28.             for (j=0;j<nDimC-1;j++) {
29.                 printf(“%10.6f “,pdM[COORD(pnG
                    lb2Lcl[i],j,nDimC)]);
30.             }
31.             printf(“%10.6f\n”,pdM[COORD(pnGl
                b2Lcl[i],nDimC-1,nDimC)]);
32.         }
33.     }
34. }
35. void vPrintMat(double *pdM, int nDimF, int
    nDimC) {
36.     int i,j;
37.     for (i=0;i<nDimF;i++) {
38.         for (j=0;j<nDimC-1;j++) {
39.             printf(“%10.6f “,pdM[COORD(i,j,nD
                imC)]);
40.         }
41.         printf(“%10.6f\n”,pdM[COORD
            (i,nDimC-1,nDimC)]);
42.     }
43. }
44. int nGenerate(double *pdA, int nDim, int mDim)
    {
```

```
45.     int i,j;
46.     for (i=0;i<nDim;i++)
47.         for (j=0;j<mDim;j++)
48.             pdA[COORD(i,j,mDim)] = (double)
                rand()/((double)RAND_MAX+1.0);
49.     return 0;
50. }
51. int nCheck(double *pdX, double *pdSol, int
    nDim) {
52.     int i;
53.     for (i=0;i<nDim;i++) {
54.         if (fabs(pdX[i]-pdSol[i])>EPS) {
55.             return -1;
56.         }
57.     }
58.     return 0;
59. }
60. int Secuencial(double *pdA, *pdB, *pdSol, int
    nDim, qDim, mDim) {
61.     int i,j,k;
62.     for (i=0;i<nDim;i++) {
63.         for (j=0;j<mDim;j++) {
64.                 pdSol[COORD(i,j,mDim)]=0;
65.             for (k=0;k<qDim;k++) {
66.                 pdSol[COORD(i,j,mDim)]+=
                    pdA[COORD(i,k,qDim)] *
67. pdB[COORD(k,j,mDim)];
68.             }
69.         }
70.     }
71.     return 0;
72. }
73. int Solve(int my_rank, p, double *pdA, *pdB,
    *pdX, int nDim, qDim, mDim, nLclDim,
    *pnProcRow, *pnGlb2Lcl) {
74.     int i, j, k;
75.     int pk; /* procesador al que pertenece la
        k-ésima fila */
76.  int lcli, lclk; /* Índice local para cada fila global
     */
77.  for (i=0;i<nLclDim;i++) {
78.     for (j=0;j<mDim;j++) {
79.     pdX[COORD(i,j,mDim)] = 0;
80.         for (k=0;k<qDim;k++) {
81.             pdX[COORD(i,j,mDim)] +=
                pdA[COORD(i,k,qDim)] *
82. pdB[COORD(k,j,mDim)];
83.         }
84.     }
85.   }
86.   return 0;
87. }
88.
89. int main(int argc, char *argv[]) {
90.   int       my_rank, p;
      /* Índice y número de procesos      */
91.   int       source, dest;
      /* Índices para origen y destino    */
92.   int       tag = 0;
      /* Etiqueata para los mensajes      */
93.   double    *pdA, *pdB, *pdX, *pdSol;
      /* Matrices                         */
94.   int       nDim, qDim, mDim;
      /* Tamaños de las matrices */
95.   int       nLclDim;          /*
      Tamaño del vector local */
96.   MPI_Status  status;
      /* Estado de la Operación */
97.   int       rc;
      /* Retorno genérico de la función   */
98.   int       i,j;
      /* Contador de propósito general    */
99.   int       *pnDispX;
      /* Vector de Desplazamientos (x)    */
100.  int       *pnSizeX;
      /* Vector de Tamaños Locales (x)    */
101.  int       *pnDispA;
      /* Vector de Desplazamientos (A)    */
102.  int       *pnSizeA;
      /* Vector de Tamaños Locales (A)    */
103.  int       *pnRows;
      /* Vector de Tamaños Locales (A)    */
104.  int       nPos;
      /* Posición base para los tamaños   */
105.  int       *pnProcRow;
      /* Procesador para cada fila        */
```

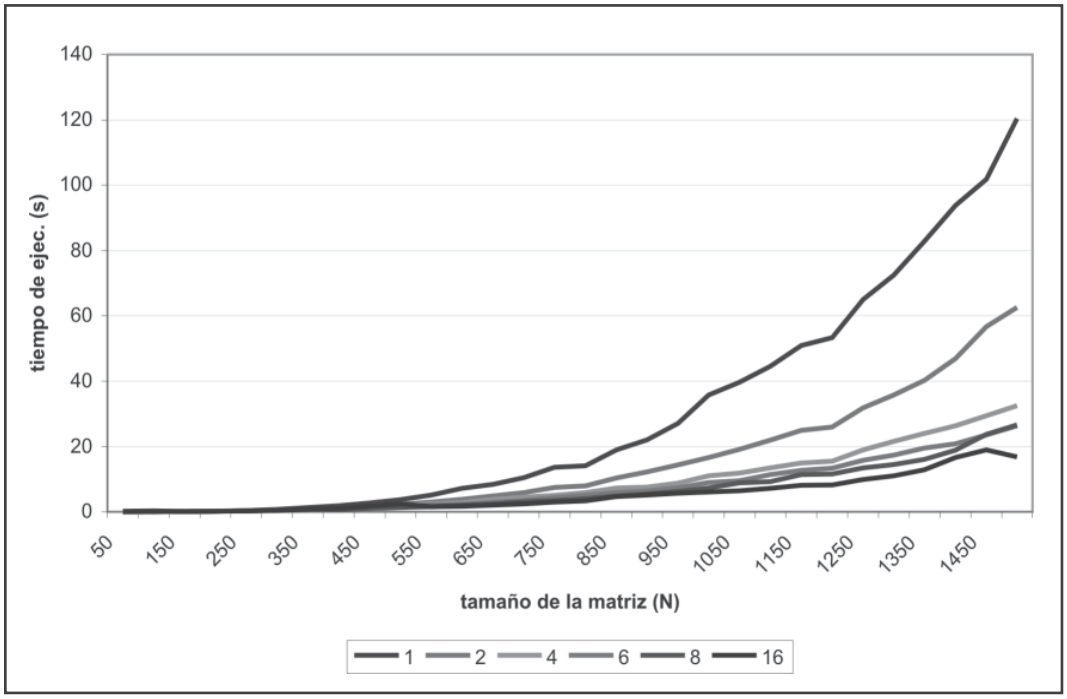
```
106.    int             *pnGlb2Lcl;
        /* Indice local para cada fila          */
107.    double          elapsed, ucpu, scpu, time,
                        t1, tmax, tpar, tsec;
108.
109.  /* Puesta en marcha de MPI */
110.  MPI_Init(&argc, &argv);
111.  MPI_Comm_rank(MPI_COMM_WORLD,
      &my_rank);
112.  MPI_Comm_size(MPI_COMM_WORLD, &p);
113.   /* Lee tamanyos de las matrices */
114.   rc = LeeEntero(argc, argv, "-n", &nDim);
115.   if (rc) {
116.      nDim = DEFAULT_SIZE;
117.   }
118.   rc = LeeEntero(argc, argv, "-q", &qDim);
119.   if (rc) {
120.      qDim = nDim;
121.   }
122.   rc = LeeEntero(argc, argv, "-m", &mDim);
123.   if (rc) {
124.      mDim = nDim;
125.   }
126.   /* Cálculo de los factores de distribución de los
      datos */
127.   nLclDim = nDim/p;
128.   pnDispA = (int *)malloc(sizeof(int)*p);
129.   pnSizeA = (int *)malloc(sizeof(int)*p);
130.   pnDispX = (int *)malloc(sizeof(int)*p);
131.   pnSizeX = (int *)malloc(sizeof(int)*p);
132.   pnRows = (int *)malloc(sizeof(int)*p);
133.   pnProcRow = (int *)malloc(sizeof(int)*nDim);
134.   pnGlb2Lcl = (int *)malloc(sizeof(int)*nDim);
135.   nPos = 0;
136.   for (i=0;i<p;i++) {
137.      pnDispA[i] = nPos*qDim;
138.            pnDispX[i] = nPos*mDim;
139.      if (i < nDim%p) {
140.         pnRows[i] = nLclDim+1;
141.      } else {
142.         pnRows[i] = nLclDim;
143.      }
144.   for (j=nPos;j<nPos+pnRows[i];j++) {
145.      pnProcRow[j] = i;
146.        pnGlb2Lcl[j] = j-nPos;
147.      }
148.      nPos += pnRows[i];
149.      pnSizeA[i] = pnRows[i]*qDim;
150.            pnSizeX[i] = pnRows[i]*mDim;
151.   }
152.   nLclDim = pnRows[my_rank];
153.    /* Reserva de Memoria y rellenado de los
      datos */
154.   if (my_rank == 0) {
155.      pdA = (double *)malloc(nDim*qDim*
      sizeof(double));
156.      pdB = (double *)malloc(qDim*mDim*
      sizeof (double));
157.      pdX = (double *)malloc(nDim*mDim*
      sizeof(double));
158.      pdSol = (double *)malloc(nDim*mDim*
      sizeof(double));
159.      nGenerate(pdA, nDim, qDim);
160.      nGenerate(pdB, qDim, mDim);
161.      printf("\nA:\n");
162.      vPrintMat(pdA, nDim, qDim);
163.      printf("\nB:\n");
164.      vPrintMat(pdB, qDim, mDim);
165.   } else {
166.      pdA = (double *)malloc(nLclDim*qDim*
      sizeof(double));
167.      pdX = (double *)malloc(nLclDim*mDim*
      sizeof(double));
168.      pdB = (double *)malloc(qDim*mDim*
      sizeof(double));
169.   }
170.
171.   /* Iniciar cronometro */
172.       ctimer(&elapsed, &ucpu, &scpu);
173.   /* Enviar la matriz A a los procesadores por
      filas */
174.      rc = MPI_Scatterv(pdA, pnSizeA,
      pnDispA, MPI_DOUBLE, pdA,
175.            nLclDim*nDim, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
176.   /* Enviar matriz B a los procesadores entera */
177.  rc = MPI_Bcast(pdB, qDim*mDim,
      MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
178.  /* Llamar la funcion que hace el producto */
179.      rc = Solve(my_rank, p, pdA, pdB, pdX,
          nDim, qDim, mDim, nLclDim,
180.                          pnProcRow, pnGlb2Lcl);
181.  /* Recibir los resultados de cada procesador */
182.      rc = MPI_Gatherv(&pdX[0], pnSizeX
          [my_rank], MPI_DOUBLE, pdX,
183.              pnSizeX, pnDispX, MPI DOUBLE,
                0, MPI_COMM_WORLD);
184.  /* Detener cronómetro */
185.  ctimer(&elapsed, &ucpu, &scpu);
186.  tpar = elapsed;
187.  if (my_rank == 0) {
188.     printf("\nSol:\n");
189.     vPrintMat(pdX, nDim, mDim);
190.     ctimer(&elapsed, &ucpu, &scpu);
191.     t1 = elapsed;
192.     rc = Secuencial(pdA, pdB, pdSol, nDim,
          qDim, mDim);
193.      printf("\nSolucion secuencial:\n");
194.      vPrintMat(pdSol, nDim, mDim);
195.      ctimer(&elapsed, &ucpu, &scpu);
196.      tsec = elapsed - t1;
197.      printf(«\n%d\t%d\t%f», p,nDim, tpar);
198.      }
199.       if (my_rank == 0) {
200.           if (nCheck(pdX, pdSol, nDim*mDim))
201.           printf("Solucion incorrecta\n");
202.       else
203.           printf("Solucion correcta\n");
204.       }
205.           free(pdA);
206.           free(pdB);
207.           free(pdX);
208.        if (my_rank == 0)
209.           free(pdSol);
210.       free(pnDispA);
211.        free(pnDispX);
212.       free(pnSizeA);
213.       free(pnSizeX);
214.       free(pnGlb2Lcl);
215.       free(pnProcRow);
216.       MPI_Finalize();
217.  }
```

Al ejecutar el programa con diferentes números de procesadores y diferentes tamaños de la matriz, podemos observar la mejora de la eficiencia al utilizar paralelismo. Cuando se utilizaron 16 procesadores para ejecutar el algoritmo se obtuvieron mejoras bastante buenas con respecto al código secuencial que se ejecuta en un simple procesador. También es importante que el problema sea de gran dimensión para que los resultados obtenidos con el paralelismo sean realmente importantes, como se aprecia en la figura 24. El speed-up en cada caso fue el siguiente (n=1500):

p	t_{sec}/t_p
2	1,92290065
4	3,69968288
6	4,56451076
8	4,51412541
16	7,12877382

FIGURA 24
Tiempos de ejecución con diferente número de procesadores



6
BIBLIOTECAS NUMÉRICAS OPTIMIZADAS

Hasta ahora se han mostrado algunas formas de optimizar los programas como la computación paralela y el uso de diferentes formatos de almacenamiento en el caso de las matrices dispersas. En este capítulo se estudiarán otras maneras de optimizar un código, utilizando librerías optimizadas de álgebra lineal y mediante métodos de optimización manuales. Nos centraremos en el uso de bibliotecas de funciones numéricas optimizadas para el desarrollo de programas secuenciales y paralelos de altas prestaciones. Existen diversas bibliotecas de funciones que ofrecen rutinas que resuelven de forma eficiente diversos problemas numéricos tanto básicos (sumas y productos de matrices y vectores, normas, etc.) como de mayor nivel (resolución de sistemas de ecuaciones, mínimos cuadrados, valores propios, etc.).

Se analizarán las bibliotecas secuenciales BLAS y LAPACK, que constituyen dos kernels básicos computacionales ampliamente utilizados y con buen soporte. Sus versiones originales y actualizadas son de libre distribución y código abierto [31], además de versiones comerciales optimizadas para diversas arquitecturas. También existen versiones en paralelo de estas bibliotecas (PBLAS y ScaLAPACK), además de la librería de comunicaciones BLACS, adaptada a problemas de álgebra lineal. Se completa el tema con una descripción de técnicas manuales de optimización de códigos.

6.1 OPTIMIZACIÓN MANUAL DE CÓDIGOS SECUENCIALES

Los códigos secuenciales de los programas pueden optimizarse manualmente utilizando diferentes técnicas que buscan, en general, reducir el número de operaciones realizadas y el número de accesos a memoria para mejorar el rendimiento global.

6.1.1 Optimización de bucles

La mayor parte de las acciones orientadas a optimizar el coste computacional de los algoritmos numéricos se centran en mejorar la localidad de referencia y las operaciones innecesarias en bucles.

Reordenación de bucles. El acceso a los datos en los bucles debe realizarse de forma que se maximice la localidad de referencia. Las matrices en C se almacenan por filas, y por tanto conviene, en la medida de lo posible, ordenar los bucles para que sucesivas iteraciones reutilicen las mismas posiciones de datos. Cambiando únicamente el orden de los bucles, se puede realizar la misma operación con costes diferentes. El siguiente es un ejemplo de reordenación de bucles.

```
1. for (j=0; j<M; j++)                    for (i=0; i<N; i++)
2.     for (i=0; i<N; i++)        ⇒          for (j=0; j<M; j++)
3.         c[i,j] = a[i,j] * b[i,j];             c[i,j] = a[i,j] * b[i,j];
```

Desenrollado de bucles. La sobrecarga causada por el manejo de los índices de los bucles no sólo influye en el número de operaciones, sino que además afecta a la localidad de referencia. En determinados casos es recomendable expresar de forma explícita varias iteraciones dentro del bucle. Al hacer el desenrollado del bucle se obtiene una mejora en cuanto al número de accesos a memoria y el uso de los registros del computador. Esta técnica se conoce como *loop unrolling*.

```
1. for (i=0; i<N; i++)                    for (i=0; i<N; i++)
2.     for (j=0; j<M; j++)                    for (j=0; j<M; j+=4){
3.         c[j] += b[i] * a[i,j];      ⇒          c[j] += b[i] * a[i,j];
4.                                                c[j+1] += b[i] * a[i,j+1];
5.                                                c[j+2] += b[i] * a[i,j+2];
6.                                                c[j+3] += b[i] * a[i,j+3];
7.                                            }
```

Fusión de bucles. Cuando no hay interferencias con la localidad de referencia (por ejemplo, no hay escrituras y lecturas del mismo componente en la misma iteración), es conveniente fusionar los bucles para reducir la sobrecarga de los contadores de bucles. Así se reduce el impacto de los accesos a los contadores, se aumenta el grano de las operaciones y se mejora el acceso a la caché.

```
1. for (i=0; i<N; i++)                    for (i=0; i<N; i++)
2.     for (j=0; j<M; j++)                    for (j=0; j<M; j+=4) {
3.         c1[j] += b1[i] * a[i,j];    ⇒          c1[j] += b1[i] * a[i,j];
4.                                                c2[j] += b2[i] * a[i,j];
5. for (i=0; i<N; i++)                        }
6.     for (j=0; j<M; j++)
7.       c2[j] += b2[i] * a[i,j];
```

Orientación a bloques. Cuando los datos que se manejan no caben en la memoria caché disponible, es conveniente orientar el problema de forma que la operación básica sea un bloque. De esta forma es posible que se aumente el impacto de la caché. Este método es recomendable en operaciones de orden cúbico o superior.

```
1. for (i=0; i<N; i++)
2.     for (j=0; j<M; j++)
3.       for (k=0; k<L; k++)
4.           c[i,j] += a[i,k]*b[k,j];
5.
6.
7.
8.
9.
10.
11.
12.
13.
```

```
for (i0=0; i0<M; i0+=NB_M){
    i1 = i0 + NB_M;
    for (j0=0; j0<N; j0+=NB_N){
        j1 = j0 + NB_N;
    for (k0=0; k0<L; k0+=NB_L){
        k1 = k0 + NB_L;
        for (i=i0;i<i1;i++)
            for (j=j0;j<j1;j++)
                for (k=k0;k<k1;k++)
                        c[i,j]+=a[i,k]*b[k,j];
        }
    }
}
```

Optimización escalar. El acceso a una componente de un vector requiere resolver una referencia a memoria. Este acceso puede penalizar notablemente los tiempos de cómputo si se realiza de forma repetida e innecesaria dentro de un bucle. Una solición a este problema es la ubicación del valor en un registro, como se muestra a continuación

```
1.  for (i=0; i<N; i++)
2.      for (j=0; j<M; j++)
3.          c[j] += b[i] * a[i,j];
4.
5.          }
```

```
for (i=0; i<N; i++) {
    reg = b[i];
    for (j=0; j<M; j+=4)
        c[j] += reg * a[i,j];
```

Pruebas

Para analizar el efecto de los diferentes métodos de optimización de códigos secuenciales, como el reordenamiento y desenrollado de bucles y la ordenación orientada a bloques se hicieron pruebas utilizando un algoritmo sencillo de orden cúbico para calcular el producto de dos matrices. Además de las diferentes optimizaciones mencionadas anteriormente se analiza el efecto de compilar el código con diferentes opciones que nos da el compilador como se observa en la tabla 2. Los resultados (en segundos) obtenidos con los diferentes métodos son los siguientes:

TABLA 2
Comparación de tiempos de ejecución del producto matricial con diferentes técnicas de optimización de bucles

	i-j-k	i-k-j	j-i-k	j-k-i	k-i-j	k-j-i	k-i-j LU	i-j-k OB	i-k-j OB	i-k-j LU+OB
-o	4,14	1,17	4,61	13,59	1,36	14,43	1,21	4,17	1,34	1,15
-o1	4,13	1,21	4,72	15,41	1,35	13,73	1,27	4,15	1,41	1,19
-o2	4,14	1,22	4,63	13,46	1,43	14,01	1,26	4,23	1,39	1,17
-o3	4,17	1,21	4,66	15,31	1,43	15,32	1,24	4,17	1,44	1,19

LU: Loop unrolling de 4 iteraciones.
OB: Orientado a bloques de 128.

Como se aprecia claramente en la tabla, el ordenamiento de bucles juega un papel importante en el desempeño del programa, ejecuta más de 10 veces más rápido el algoritmo en algunas ocasiones con el simple hecho de cambiar el recorrido por filas o por columnas. Esto se debe al manejo que le da el procesador a la memoria, y el funcionamiento de las cachés.

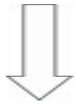
Al combinar los diferentes métodos de optimización se consiguió el mejor tiempo, pero éste no es significativamente superior al de la simple reordenación del bucle. También observamos que no siempre el compilador optimiza más el código al utilizar los métodos de compilación -o1, -o2, -o3.

Entre mayor sea la cantidad de datos que se deben procesar, mayor será el impacto que puede tener la utilización de estas sencillas técnicas de optimización, aunque requieren de un buen conocimiento de la arquitectura sobre la que se trabaja y del lenguaje usado, pues lo que se busca es explotar aspectos propios de cada uno de ellos.

6.1.2 Inlining

La sobrecarga producida por las llamadas a función puede llegar a ser mayor que el costo de la propia función. Cuando se realiza una llamada a función se reserva espacio en la pila para las variables locales, se realiza el apilamiento de la dirección de retorno y se realiza la asignación de los resultados. Cuando las funciones sean suficientemente sencillas, es conveniente sustituir su código directamente en el código de la función que realiza la llamada. La sustitución puede hacerse de forma explícita, mediante el uso de macros o con la directiva *inline*. El proceso de inlining consiste en la sustitución de una llamada a función por el código propiamente dicho. Reduce la legibilidad del código, pero evita el sobrecosto de paso de parámetros, llamada a función, reserva de pila y retorno de resultados.

```
1.  double hilbert(int i, int j) {
2.    double res;
3.    res = 1/( (double)i + (double)j - 1);
4.    return res;
5.  }
```



```
1.  #define hilbert(i,j) (1.0/((double)(i)+
    (double)(j)-1))
```

6.2 BIBLIOTECAS SECUENCIALES DE OPTIMIZACIÓN

Una de las aproximaciones más efectivas para mejorar la velocidad, precisión y estabilidad de los

programas es mediante el uso de librerías estándar para la resolución de los problemas numéricos. El uso de este tipo de librerías permite acelerar el proceso de desarrollo garantizando altos niveles de calidad y robustez. Las librerías numéricas secuenciales más utilizadas en la actualidad son BLAS y LAPACK. El código fuente básico de ambas librerías se encuentra libremente disponible, aunque existen versiones optimizadas para diferentes plataformas (Intel Pentium, Intel Itanium II, G5).

6.2.1 BLAS – Basic Linear Algebra Subroutines

El paquete BLAS contiene una serie de funciones básicas para la resolución de problemas simples de álgebra lineal, como la suma o el producto de vectores y matrices elaboradas en el lenguaje Fortran, y existen interfaces que permiten su uso en C. BLAS dispone de versiones especiales para matrices estructuradas (triangulares, banda, etc.) y se encuentra organizado por niveles. El nivel 1 corresponde a las funciones que realizan operaciones sobre vectores (como la suma de vectores). El nivel 2 corresponde a las funciones que realizan operaciones sobre matrices y vectores (como el producto de matriz por vector) y el nivel 3 corresponde a las funciones que realizan operaciones sobre matrices.

Cuanto mayor sea el nivel de las operaciones, mejores resultados de optimización se obtienen. Todas las funciones BLAS mantienen una nomenclatura estándar. Asumiendo un nombre de función con siete letras agrupadas en tres bloques (TMMOOOO, tipo de datos, tipo de matriz, código de operación), las opciones son las siguientes:

T	s	para datos reales en simple precisión
	d	para datos reales en doble precisión
	z	para datos complejos en simple precisión
	c	para datos complejos en doble precisión
MM	ge	Matrices generales (no triangulares, banda, hessenberg, etc.)
	tr	Matrices triangulares (superiores o inferiores)
	tp	Matrices triangulares empaquetadas (sin almacenar ceros)
	<nodef>	Operaciones sobre vectores o escalares
OOO		Nombre de la operación.

Las siguientes son algunos ejemplos de funciones BLAS:

dgemm	Producto matricial, matrices generales rectangulares, doble precisión.
strmv	Producto matriz por vector, matriz triangular, simple precisión.
daxpy	Suma de dos vectores.
dtrsv	Resolución de un sistema triangular de ecuaciones lineales.

Para comprobar las mejoras que se pueden obtener con el uso de estas bibliotecas en la eficiencia de los programas, hemos realizado el producto matricial en C mediante tres diferentes aproximaciones. Primero, utilizando un algoritmo normal con los correspondientes bucles; segundo, mediante el uso de la función de producto matriz-vector de blas (dgemv) para calcular cada columna; y por último con una sola llamada a la función del producto matricial de blas (dgemm).

ALGORITMO 19. Producto matriz-matriz en c usando BLAS

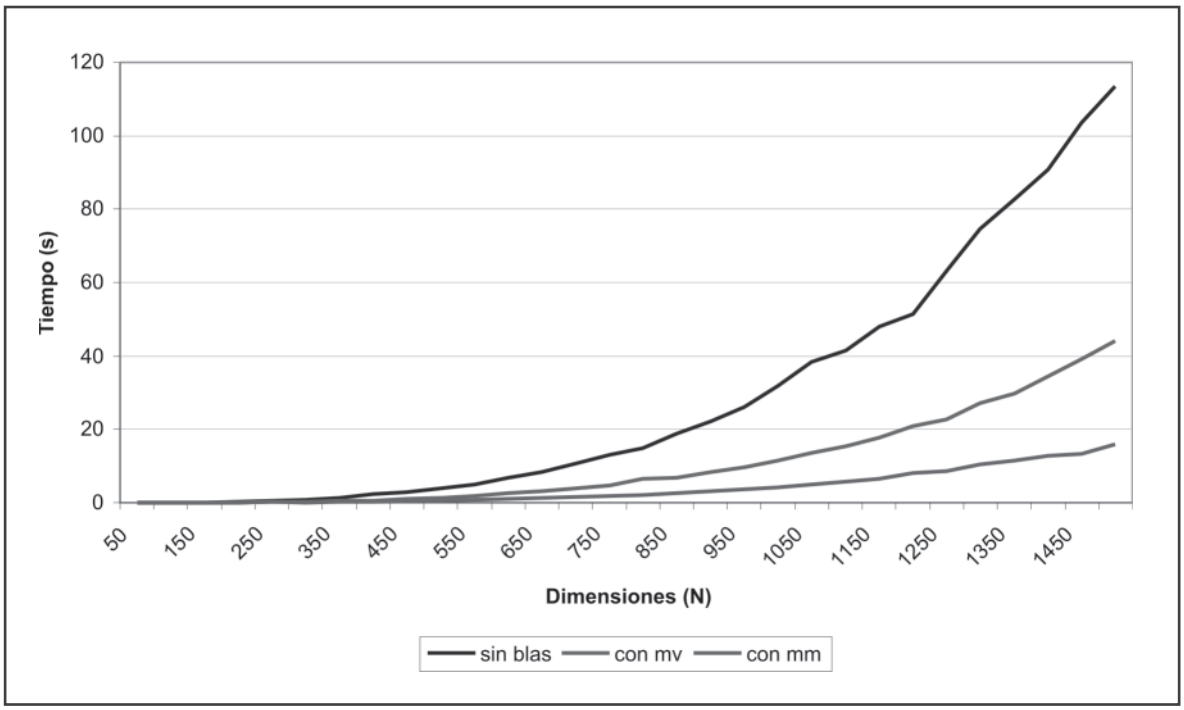
```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <cblas.h>
4.  #include <math.h>
5.  #include “opciones.h”
6.  #include “ctimer.c”
7.  #define DEFAULT_SIZE 100
8.  #define COORD(i,j,n) ((i)*(n)+(j))
9.
10. int main (int argc, char *argv[]){
11. double *A, *X, *B;
12. int i,j,k,n,q,m,rc;
13. double ALPHA = 1.0, BETA = 0.0;
14. char T = ‘N’;
15. double elapsed, ucpu, scpu, time, t1, t2, t3;
16.
17. A = (double*)malloc(n*q*sizeof(double));
18. X = (double*)malloc(m*n*sizeof(double));
19. B = (double*)malloc(q*m*sizeof(double));
20.
21. % Sin usar librerías %
22. ctimer(&elapsed, &ucpu, &scpu);
23. for (i=0;i<n;i++) {
24.    for (j=0;j<m;j++) {
25.       B[COORD(i,j,m)]=0;
26.       for (k=0;k<q;k++) {
27.          B[COORD(i,j,m)] += A[COORD(i,k,q)] *
          X[COORD(k,j,m)];
28.       }
29.    }
30. }
31. ctimer(&elapsed, &ucpu, &scpu);
32. t1 = elapsed;
33.
34. % Usando M*v %
35. ctimer(&elapsed, &ucpu, &scpu);
36. time = elapsed;
37. for (j=0;j<m;j++) {
38.    cblas_dgemv(CblasRowMajor,
       CblasNoTrans, m, n, ALPHA, A, m, &X[j], m,
       BETA, &B[j], m);
39. }
40. ctimer(&elapsed, &ucpu, &scpu);
41. t2 = elapsed-time;
42.
43. % Usando M*M %
44. ctimer(&elapsed, &ucpu, &scpu);
45. time = elapsed;
46. cblas_dgemm(CblasRowMajor,CblasNoTrans,
    CblasNoTrans, n, m, q, ALPHA, A, q, X, m,
    BETA, B, m);
47. ctimer(&elapsed, &ucpu, &scpu);
48. t3 = elapsed-time;
49.
50. printf(«\n%d\t%10.6f\t%10.6f
    t%10.6f»,n,t1,t2,t3);
51. }
```

Como se observa en la figura 25, la eficiencia que alcanza esta biblioteca es bastante alta. Los promedios de operaciones por segundo que realizó cada uno de los métodos son los siguientes:

- sin librerías: 32,50 MFLOPS
- con mv: 100,22 MFLOPS
- con mm: 228,81 MFLOPS

Como se puede observar, la diferencia es de varios millones de operaciones por segundo, lo que nos puede dar una gran ventaja en tiempo de ejecución a nuestros programas. Así como hemos logrado optimizar el producto matricial en gran proporción, especialmente al utilizar matrices grandes, podríamos optimizar mediante librerías gran parte de las operaciones que en nuestros programas debamos realizar sobre matrices y vectores.

FIGURA 25
Tiempos del producto matricial en C utilizando BLAS



6.2.2 LAPACK – Linear Algebra Package

El paquete LAPACK contiene una serie de funciones para la resolución de problemas fundamentales del álgebra lineal, como la resolución de sistemas de ecuaciones, la factorización en valores propios, numerosas descomposiciones matriciales, etc. LAPACK dispone de versiones especiales para matrices estructuradas (triangulares, banda, etc.) y se encuentra en continuo desarrollo desde 1992. LAPACK utiliza BLAS para la implementación de las funciones.

Las rutinas de LAPACK se estructuran en tres niveles. En primer lugar, se definen las rutinas *driver*, que resuelven problemas completos (resolución de un sistema de ecuaciones, resolución de un problema de mínimos cuadrados, cálculo de los valores propios o singulares de una matriz). Las funciones *driver* son las que se recomienda que utilice el usuario. En segundo lugar se definen las funciones computacionales, que resuelven tareas computacionales completas (factorización LU, descomposción QR) que si bien pueden utilizarse directamente, están generalmente orientadas a resolver una parte de un problema completo que resuelve una rutina *driver*. Finalmente se encuentran las rutinas auxiliares, utilizadas por las rutinas *driver* y computacionales para la resolución de los problemas.

La nomenclatura de las funciones LAPACK sigue los mismos patrones que en el caso de las funciones BLAS. Como ejemplo se citan las siguientes funciones:

dgesv_ Función que resuelve un sistema de ecuaciones Ax=b
dgeqrf_ Función que calcula la descomposición QR de A (A = QR)
dgetrf_ Función que calcula la factorización LU de una matriz

La biblioteca LAPACK tiene numerosas funciones de álgebra lineal, y se basa en las operaciones de BLAS para ejecutarlas eficientemente. Para analizar la optimización que se puede alcanzar al utilizar esta biblioteca se realizaron pruebas comparando algoritmos convencionales con las funciones optimizadas de LAPCAK. El análisis se ha realizado tomando los tiempos de ejecución de la descomposición LU de una matriz. Se hicieron nuevamente 3 pruebas: Sin librerías; con funciones separadas para la descomposición y la sustitución; y con la función directa de resolución. Las dos últimas mostraron una eficiencia bastante cercana, y muy superior a nuestro método manual, confirmando una vez más que el uso de estas bibliotecas optimizadas mejora bastante las prestaciones de nuestros programas.

Algoritmo 20. Solución de sistemas por descomposición LU

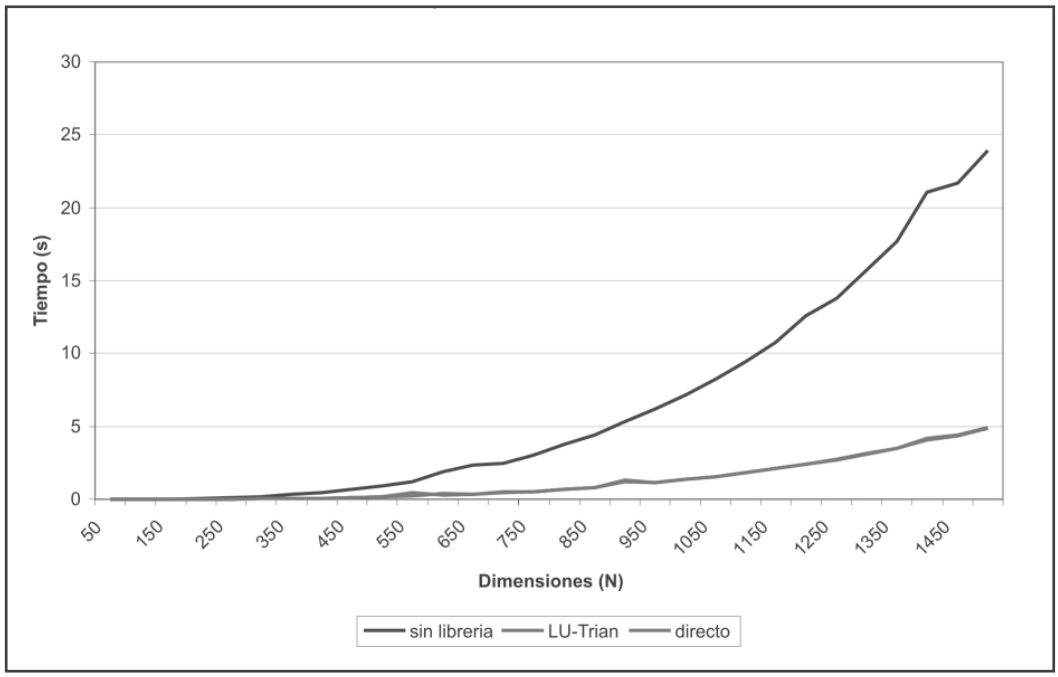
```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <cblas.h>
4.  #include <math.h>
5.  #include “opciones.h”
6.  #include “ctimer.c”
7.  #define DEFAULT_SIZE 100
8.  #define COORD(i,j,n) ((i)*(n)+(j))
9.
10. int descLU(double *A, double *B, double *X, int
      n){
11.    int i,j,k;
12.    for(k=0;k<n-1;k++){
13.      for(i=k+1;i<n;i++){
14.        A[COORD(i,k,n)] /= A[COORD(k,k,n)];
15.          for(j=k+1;j<n;j++){
16.            A[COORD(i,j,n)] -= A[COORD(i,k,n)] *
               A[COORD(k,j,n)];
17.          }
18.          B[i] -= (A[COORD(i,k,n)] * B[k]);
19.    }
20.    }
21.  for (i=n-1;i>0;i--) {
22.      for (j=i+1;j<n;j++) {
23.          B[i] -= (A[COORD(i,j,n)] * X[j]);
24.      }
25.      X[i] = B[i] / A[COORD(i,i,n)];
26.   }
27.   return 0;
28. }
29.
30. int main(int argc, char *argv[]){
31.   double *A,*A2,*A3, *X, *B,*B2,*B3;
32.   int i,j,k,n,rc,*piv,error,nrhs;
33.   double ALPHA = 1.0, BETA = 0.0;
34.   char T = ‘N’; // T:transpose, N: no
35.   double elapsed, ucpu, scpu, time, t1, t2, t3;
36.
37.   nrhs = 1;
38.   A = (double*)malloc(n*n*sizeof(double));
39.   A2 = (double*)malloc(n*n*sizeof(double));
40.   A3 = (double*)malloc(n*n*sizeof(double));
41.   X = (double*)malloc(n*sizeof(double));
42.   B = (double*)malloc(n*sizeof(double));
43.   B2 = (double*)malloc(n*sizeof(double));
44.   B3 = (double*)malloc(n*sizeof(double));
45.   piv = (int*)malloc(n*sizeof(int));
46.   nGenerate(A,B,n);
47.   cblas_dcopy(n*n, A, 1, A2, 1);
48.   cblas_dcopy(n*n, A, 1, A3, 1);
49.   cblas_dcopy(n, B, 1, B2, 1);
50.   cblas_dcopy(n, B, 1, B3, 1);
51.
52.   % Sin usar librerías %
53.   ctimer(&elapsed, &ucpu, &scpu);
54.   descLU(A,B,X,n);
55.   ctimer(&elapsed, &ucpu, &scpu);
56.   t1 = elapsed;
57.
58.   % Usando descomposición y sustitución %
59.   ctimer(&elapsed, &ucpu, &scpu);
60.   time = elapsed;
61.   dgetrf_(&n, &n, A2, &n, piv, &error);
62.   dgetrs_(&T, &n, &nrhs, A2, &n, piv, B2, &n,
      &error);
63.   ctimer(&elapsed, &ucpu, &scpu);
64.   t2 = elapsed-time;
65.
```

```
66.   % usando resolucion directa %
67.   ctimer(&elapsed, &ucpu, &scpu);
68.   time = elapsed;
69.   dgesv_(&n, &nrhs, A3, &n, piv, B3, &n,
      &error);
70.   ctimer(&elapsed, &ucpu, &scpu);
71.   t3 = elapsed-time;
72.
73.   printf(«\n%d\t%10.6f\t%10.6f\
      t%10.6f»,n,t1,t2,t3);
74. }
```

Los promedios de operaciones por segundo que realizó cada uno de los métodos, considerando un total de n^3 operaciones, son los siguientes:

- sin librerías: 129,49 MFLOPS
- con mv: 741,70 MFLOPS
- con mm: 724,46 MFLOPS

FIGURA 26
Tiempos de ejecución de descomposición LU



Nuevamente la superioridad sobre nuestro algoritmo es bastante notable. La diferencia entre los otros dos métodos sin embargo es muy poca, lo que nos deja ver que cada parte de este algoritmo está bien optimizada, y que una llamada de más a una función no hace mucha diferencia en el comportamiento del programa, ya que la función que realiza el proceso completo utiliza prácticamente el mismo tiempo que al llamar las funciones por separado.

El único inconveniente que puedan presentar estas bibliotecas es que están basadas en fortran, y por tanto se debe tener en cuenta el tipo de almacenamiento que este lenguaje maneja al utilizarlas sobre un lenguaje diferente como C, y también recordar que los parámetros deben pasarse siempre como punteros a memoria. Una vez acostumbrados a esto, permite programar de manera más rápida, legible y eficiente, centrándonos en otros problemas a la

hora de hacer nuestros programas. Por otro lado, no siempre encontramos funciones que se adapten exactamente a nuestras necesidades, pero vale la pena hacer ensayos con aproximaciones utilizando estas bibliotecas.

6.3 BIBLIOTECAS DE OPTIMIZACIÓN PARALELAS

Las bibliotecas de optimización secuenciales tienen sus respectivas versiones paralelas para la optimización de programas ejecutados en paralelo. El conjunto de bibliotecas que constituye ScaLAPACK (versión paralela de LAPACK) se estructura en tres partes fundamentales:

- Versiones paralelas de las funciones de LAPACK. Contenidas en el directorio *SRC* del paquete, son las versiones equivalentes paralelas de casi todas las funciones disponibles en LAPACK. Estas funciones utilizan como núcleo computacional básico la biblioteca PBLAS.

- Versiones paralelas de las funciones de BLAS. Contenidas en la biblioteca PBLAS distribuida con el paquete, contiene las versiones paralelas de las funciones básicas que utiliza ScaLAPACK. Estas funciones están implementadas utilizando las funciones de BLAS y utilizando funciones de comunicaciones adaptadas a los problemas de álgebra lineal.

- Versiones especializadas de funciones de comunicaciones. Contenidas en la biblioteca BLACS, constituyen las funciones necesarias para la distribución de datos y resultados entre los diferentes procesadores.

Las bibliotecas PBLAS y ScaLAPACK son versiones muy similares a las versiones originales secuenciales tanto en su nomenclatura como en la forma de utilizarlas, por lo que se explicará únicamente la biblioteca BLACS.

6.3.1 BLACS – Basic Linear Algebra Communication Subprograms

La biblioteca de funciones BLACS constituye un nivel de abstracción superior a las bibliotecas de comunicaciones estándar (MPI, PVM, etc), proporcionando herramientas que son específicas para los problemas de computación numérica (Grids de procesos bidimensionales, operaciones de comunicación orientadas a bloques de filas o columnas, etc.). Los conceptos fundamentales de la biblioteca BLACS son el grid de procesadores y la distribución de las matrices. Los procesadores se estructuran en un grid bidimensional de N X Q procesadores, sobre los que se definen varios contextos de comunicación independientes que permiten especificar los subgrupos de comunicación naturales en álgebra lineal (fila, columna o todos). BLACS es una biblioteca de código abierto que puede descargarse e instalarse libremente y su instalación es sencilla.

De la misma forma que BLAS proporciona una interfaz compatible con Fortran y otra con C, además BLACS incluye la versión CBLACS con una interfaz en C. La diferencia es que éste viene incluido con el código original y se elige en tiempo de compilación. Las funciones que se muestran en esta sección se corresponden con la interfaz en C. Todas las funciones en la interfaz en C comienzan por el carácter *C* en mayúsculas y el nombre de la función se define en minúsculas. La nomenclatura de las funciones de comunicaciones, por ejemplo, utiliza la siguiente estructura: CvXXYY2d

Tipo de datos (v)	i (Integer), s, d, c, z
Forma de la matriz (XX)	ge, tr
Comunicación (YY)	sd, rv (Send, Receive) bs, br (Broadcast)

Las funciones disponibles se pueden clasificar en los siguientes grupos:

Funciones de Creación y Destrucción del Grid

El número de procesos disponibles en la ejecución de un programa BLACS debe estructurarse en un grid de procesos. Esta estructuración en un grid implica que los procesos se referencian mediante coordenadas cartesianas (fila – columna), de forma similar a las topologías GRID. Las funciones más utilizadas son:

*Cblacs_get(int ictxt, int what, int *val).* Devuelve información interna de BLACS (context handle, rango de IDs de mensajes, nivel de depuración,...). Un uso típico esictxt a 0 y what a 0. Devuelve el contexto por defecto.

*Cblacs_gridinit(int *ictxt, char *major, int nrow, int ncol).* Crea un contexto para la aplicación en marcha. Debe invocarse tras obtener el contexto por defecto de BLACS, que se suministra como entrada en el primer parámetro (el contexto por defecto puede obtenerse mediante la función Cblacs_get). El segundo parámetro define el orden de distribución de los procesos en el grid y los dos parámetros siguientes las dimensiones del mismo.

Cblacs_gridexit(int ICTXT). Libera un contexto BLACS.

Cblacs_exit(int Val). Esta rutina se debe llamar cuando un proceso ha terminado de usar BLACS. Un valor de 1 en Val permite seguir utilizando el MPI.

*Cblacs_gridinfo(int ICTXT, int *nP_Row, int *nP_Col, int *my_P_Row, int *my_P_Col).* Devuelve información acerca de un proceso en un determinado contexto (ICTXT), devolviendo en nP_Row, nP_Col el número de procesos por fila y columna, además de las coordenadas my_P_Row, my_P_Col, (posición fila / columna) del proceso que ha realizado la llamada.

Funciones de Comunicaciones

Las funciones de comunicaciones permiten envíos y recepciones punto a punto y multipunto. Las funciones de comunicación punto a punto son:

*Cvgesd2d(int ICONTX, int M, int N, XXX *A, int LDA, int RDEST, int CDEST).* Envío de un bloque de MxN elementos del tipo especificado por la función (segundo carácter). Los valores de los parámetros son los siguientes:

- ICONTX: Contexto de Comunicación (Cblacs_ gridinit)
- M, N, A, LDA: Dimensiones de la matriz, puntero a la matriz con los datos y dimensión de almacenamiento.
- RDEST: Índice de fila del procesador destino.
- CDE ST: Índice de columna del procesador destino.

*Cvgerv2d(int ICONTX, int M, int N, XXX *A, int LDA, int RSRC, int CSRC).* Recepción de un bloque de MxN elementos desde el procesador en la fila RSR y en la columna CSRC (-1 implica a todos los procesos). Los parámetros tienen el mismo significado que en la función anterior.

*Cvgebs2d(int ICONTX, char SCOPE, char *TOP, int M, int N, XXX *A, int LDA*). Difusión del bloque apuntado por A a todos los procesos de una fila, columna o del grid. Scope Indica el Ámbito de la Difusión: “COLUMNS”, Implica a todos los procesadores en la columna del procesador que realiza el envío, “ROWS”, Equivalente a la anterior pero por filas o “ALL”, Implica a todos los procesadores del grid. Top Indica la Topología: “ “ Para la topología por defecto o “INCREASING_RING”, “DECREASING_RING”, etc para otras topologías.

*Cvgebr2d(int ICONTX, char SCOPE, char *TOP, int M, int N, XXX *A, int LDA, int RSRC, int CSRC).* Recepción de una difusión iniciada con la operación anterior.

7

CASO DE ESTUDIO: SISTEMAS SIMÉTRICOS, DISPERSOS E INDEFINIDOS

Como hemos visto hasta ahora, la explotación de la estructura de una matriz en la resolución de un sistema de ecuaciones lineales es un campo que despierta mucho interés en el mundo de la computación numérica, y donde queda aún muchísimo por hacer. En el álgebra lineal, los algoritmos para resolver sistemas generales pueden optimizarse al enfocarlos a problemas que involucren propiedades como la simetría, la dispersión y el hecho de ser definidas positivas. Así, múltiples variaciones a los algoritmos básicos como la descomposición LU pueden generar soluciones más eficientes para matrices estructuradas.

Las matrices que hemos trabajado en este proyecto son simétricas, dispersas e indefinidas, por lo cual se comenzó por analizar los algoritmos comunes para la factorización de matrices simétricas.

7.1 FACTORIZACIÓN LDL^T

Se desea desarrollar un método que nos permita explotar la estructura simétrica para un sistema de ecuaciones lineales $Ax = b$. Se hace entonces una variación a la factorización LU donde A se factoriza al producto de tres matrices LDM^T. Donde D es una matriz diagonal y L y M son matrices triangulares inferiores. Al obtener esta factorización podwemos encontrar la solución a $Ax = b$ en $O(n^2)$ flops [4] resolviendo $Ly = b$ (sustitución progresiva), $Dz = y$, y $M^Tx = z$ (sustitución regresiva). En el caso simétrico, cuando $A = A^T$, se da también la igualdad entre L y M, y el trabajo asociado a la factorización es la mitad del requerido en la eliminación Gaussiana.

Algoritmo 21. Factorización LDM^T

```
1.  for j = 1 : n
2.      v(1:j) = A(1:j,j)
3.      for k = 1 : j – 1
4.          v(k+1:j) = v(k+1:j) – v(k) A(k+1:j,k)
5.      end for
6.      for I = 1 : j-1
7.          A(i,j) = v(i) / A(i,i)
8.      end for
9.      A(j,j) = v(j)
10.     for k = 1 : j-1
11.         A(j+1:n) = A(j+1:n,j) – v(k) A(j+1:n,k)
12.     end for
13.     A(j+1:n,j) = A(j+1:n,j) / v(j)
14. end
```

Este algoritmo requiere la misma cantidad de trabajo que la factorización LU, cerca de $2n^3/3$ flops. Sin embargo para obtener una mejor solución, haría falta aplicarle pivoteamiento a este algoritmo.

El siguiente es un ejemplo de la factorización LDM^T:

$$A=\begin{bmatrix}10 & 10 & 20\\ 20 & 25 & 40\\ 30 & 50 & 61\end{bmatrix}=\begin{bmatrix}1 & 0 & 0\\ 2 & 1 & 0\\ 3 & 4 & 1\end{bmatrix}\begin{bmatrix}10 & 0 & 0\\ 0 & 5 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 1 & 2\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

Cuando *A* es simétrica, entonces *L* = *M*, y podemos ahorrar buena parte de la computación, haciendo la mitad del trabajo que el algoritmo anterior, cerca de $n^3/3$ flops.

ALGORITMO 22. Factorización LDL^T

1. *d* = diagonal(*A*)
2. υ = zeros$_n$
3. *L* = I_n
4. **for** *j = 1:n*
5. **for** *i = 1:j-1*
6. $\upsilon(i)$ = *L(j,i) * d(i)*
7. **end**
8. $\upsilon(j)$ = *A(j,j) - L(j,1:j-1) * v(1:j-1)*
9. *d(j) = v(j)*
10. *L(j+1:n,j) = (A(j+1:n,j) - L(j+1:n,1:j-1) * v(1:j-1)) / v(j)*
11. **end**
12. *y = sustitucionProgresiva(L,b)*
13. *y = y ./ d*
14. *x = sustitucionRegresiva(L',y)*

Un ejemplo de la factorización LDL^T es el siguiente:

$$A=\begin{bmatrix}10 & 20 & 30\\ 20 & 45 & 80\\ 30 & 80 & 171\end{bmatrix}=\begin{bmatrix}1 & 0 & 0\\ 2 & 1 & 0\\ 3 & 4 & 1\end{bmatrix}\begin{bmatrix}10 & 0 & 0\\ 0 & 5 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 2 & 3\\ 0 & 1 & 4\\ 0 & 0 & 1\end{bmatrix}$$

7.2 SISTEMAS DEFINIDOS POSITIVOS

Una matriz $A \in \Re^{nxn}$ es definida positiva si $x^T Ax>0$ para todo diferente de cero. En tal caso, existe una factorización $A = GG^T$, que tiene además gran estabilidad numérica. La mtriz *G* es triangular inferior y su diagonal tiene valores positivos. Esta factorización se conoce comúnmente como *factorización Cholesky.*

ALGORITMO 23. Factorización de Cholesky - GG^T

1. *G* = I_n
2. **for** j = 1 : n
3. $\upsilon(j:n)$ = *A(j:n,j)*
4. **for** k = 1 : j-1
5. $\upsilon(j:n)$ = $\upsilon(j:n)$ - *G(j,k) * G(j:n,k)*
6. **end for**
7. *G(j:n,j)* = $\upsilon(j:n) / \sqrt{\upsilon(j)}$
8. **end for**
9. *y = sustitución Progresiva (G, b)*
10. *x = sustitución Regresiva (G', y)*

La factorización Cholesky requiere $n^3/3$ flops, y tiene una gran estabilidad numérica, pero solo funciona con matrices simétricas definidas positivas.

7.3 SISTEMAS TRIDIAGONALES

La solución de sistemas tridiagonales se hace importante en algunos de los métodos que se explicarán más adelante, por lo que presentamos aquí un corto algoritmo para resolver dichos sistemas, luego de aplicar una descomposición LDL^T, con un orden lineal.

ALGORITMO 24. Factorización de Cholesky - GG^T

1. **for** *k = 2:n*
2. *t = e(k-1)*
3. *e(k-1) = t / d(k-1)*
4. *d(k) = d(k) – t.e(k-1)*
5. **end for**
6. **for** *k=2:n*
7. *b(k) = b(k) – e(k-1) b(k-1)*
8. **end for**
9. *b(n) = b(n) / d(n)*
10. **for** *k = n – 1 : -1 : 1*
11. *b(k) = b(k) / d(k) – e(k) b(k+1)*
12. **end for**

Con el algoritmo 24 se sobrescribe el vector *b* con la solución al sistema $Ax = b$, donde *A* es simétrica, tridiagonal, definida positiva, con diagonal principal *d* y diagonal secundaria *e*.

7.4 SISTEMAS SIMÉTRICOS INDEFINIDOS

Una matriz cuya forma cuadrática x^TAx toma valores positivos y negativos se denomina indefinida. Aunque una matriz indefinida *A* puede tener una factorización LDL^T, las entradas en estos factores pueden tener una magnitud arbitraria. Es cierto que las estrategias de pivoteo comunes podrían usarse para evitar este inconveniente, pero también es cierto que dichas estrategias destruirían la estructura simétrica de la matriz, y con ello la oportunidad de explotar su estructura en la solución del sistema. Así pues debemos utilizar un pivoteo simétrico, de la forma $A \leftarrow PAP^T$, pero lamentablemente así no siempre se obtiene estabilidad en la factorización LDL^T.

El reto es entonces utilizar elementos fuera de la diagonal para hacer el pivoteamiento. Con este objetivo surgen 2 formas de conseguirlo. La primera forma es mediante el método de Aasen que computa una factorización de la forma $PAP^T = LTL^T$, donde L es triangular inferior y T es tridiagonal. La segunda forma es utilizando el método de Bunch y Parlett que computa una factorización $PAP^T = LDL^T$, donde D es una suma directa de bloques pivote de tamaño 1x1 y 2x2 (matriz diagonal por bloques). Ambas factorizaciones involucran $n^3 / 3$ flops y una vez computadas pueden utilizarse para resolver el sistema $Ax = b$ en $O(n^2)$.

7.4.1 Método de Aasen

El método de Aasen está basado en el algoritmo de Parlett y Reid (1970) para computar la descomposición $PAP^T = LTL^T$. El proceso requiere $2n^3 / 3$ flops, por lo que no es tan bueno como deseamos, pero es un punto de partida para los métodos siguientes. El método premultiplica y postmultiplica la matriz *A* por matrices de permutaciones y de multiplicadores para generar así la factorización $PAP^T = LTL^T$.

El algoritmo 25 realiza dicha factorización en el lenguaje MATLAB.

ALGORITMO 25. Método de Parlett y Reid. $PAP^T = LTL^T$

```
1.  ind = [1:n];
2.  I = eye(n);
3.  P = I;
4.  M = I;
5.  L = I;
6.  for i = 1 : n-2
7.     % se busca el mayor elemento de la columna %
8.     [mayor,index] = max(A(i,i+1:n));
9.     index = index + i;
10.    % se crea el vector de permutación %
11.    p = ind;
12.    p(i+1) = index;
13.    p(index) = i+1;
14.    % se crea la matriz de permutación %
15.    per = I(:,p);
16.    A = per * A * per';
17.       P = per * P;
18.    L(i+1:n,i) = L(p(i+1:n),i);
19.    % se crea el vector de multiplicadores alfa %
20.    m = zeros(n,1);
21.    for j = i+2 : n
22.       m(j) = A(i,j) / mayor;
23.    end for
24.    % se crea la matriz de multiplicadores %
25.    mult = I - m * I(:,i+1)';
26.    A = mult * A * mult';
27.    M = mult * M;
28.    L = L - tril(mult,-1);
29. end for
30. T = A;
31. z=L\(P*b); % Lz = Pb %
32. ω=T\z; % Tω = z %
33. y=L'\w; % L'y = w %
34. x=P*y; % x = Py %
```

Dicho método aplicado a la matriz

$$A=\begin{bmatrix}0 & 1 & 2 & 3\\ 1 & 2 & 2 & 2\\ 2 & 2 & 3 & 3\\ 3 & 2 & 3 & 4\end{bmatrix}$$

En cada paso tendríamos

$$P_1 = [e_1, e_4, e_3, e_2]$$

$$M_1 = I_4 - (0,0,2/3,1/3)^T e_2^T$$

$$P_2 = [e_1, e_2, e_4, e_3]$$

$$M_2 = I_4 - (0,0,0,1/2)^T e_3^T$$

Y $PAP^T = LTL^T$, con $P = [e_1, e_3, e_4, e_5]$,

$$L=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 1/3 & 1 & 0\\ 0 & 2/3 & 1/2 & 1\end{bmatrix} \quad y$$

$$T=\begin{bmatrix}0 & 3 & 0 & 0\\ 3 & 4 & 2/3 & 0\\ 0 & 2/3 & 10/9 & 0\\ 0 & 0 & 0 & 1/2\end{bmatrix}$$

Para mejorar este algoritmo y acercarlo a los $n^3/3$ flops, Aasen utilizó el hecho de que la matriz $H = TL^T$ es una matriz Hessenberg superior. Así, podemos ver el sistema $PAP^T = LTL^T = LH$, y utilizar esta matriz en la resolución. Así, para calcular las columnas de la matriz *H* en cada paso, utilizamos el algoritmo 26. El cálculo de esta matriz se convierte en el núcleo principal del método de Aasen.

ALGORITMO 26. Método para calcular columna *j* de *H*

```
1.  β = [0, β]
2.  if j == 1
3.    h(1) = A(1,1)
4.  else if j == 2
5.    h(1) = β (1)
6.    h(2) = A(2,2)
7.  else
8.    l(1) = 0
9.    l(2) = 0
10.   l(3:j) = L(j,2:j-1)
11.   l(j+1) = 1
12.   h(j) = A(j,j)
13.   for k = 1:j-1
14.     h(k) = β (k) * l(k) + α(k) * l(k+1) +
        β (k+1) * l(k+2)
15.     h(j) = h(j) - l(k+1) * h(k)
16.   end
17. end
```

Como podemos presumir, cuando los elementos de las columnas de L son muy grandes, estamos en problemas. Para resolver esto necesitamos permutar el elemento mayor de *v(j+1:n)* a la posición superior. Por supuesto, esta permutación debe ser aplicable a la parte no resucida de *A* y la parte ya computada de *L*. El algoritmo 27 muestra el método de Aasen con pivoteamiento, donde los elementos de la diagonal de *T* se guardan en el vector α y la subdiagonal en el vector β. Este método tiene la misma estabilidad que la eliminación Gaussiana con pivoteo parcial.

ALGORITMO 27. Método de Aasen LTL'

1. $\alpha = A_{11}$
2. $\upsilon = \text{zeros}_n$
3. $L = I_n$
4. $\beta(1) = 0$
5. **for** $j = 1:n$
6. % computar h(1:j) %
7. $h(1:j) = computarH(A,L,a,B,j)$ % algoritmo 16%
8. % computar α (j) %
9. **if** $j == 1$ **or** $j == 2$
10. $\alpha(j) = h(j)$
11. **else**
12. $\alpha(j) = h(j) - \beta(j-1) * L(j,j-1)$
13. **end**
14. **if** $j \leq n-1$
15. $\upsilon(j+1:n) = A(j+1:n,j) - L(j+1:n,1:j) * h(1:j)$
16. % Encontrar q con $abs(v(q)) = max(\upsilon(j+1:n))$ y $j+1 \leq q \leq n$ %
17. [*m*,*q*] = max(abs(*v*(*j*+1:*n*)))
18. % Intercambiar filas y columnas %
19. swap(*v*(*j*+1),*v*(*q*))
20. swap(*L*(*j*+1,2:*j*),*L*(*q*,2:*j*))
21. swap(*A*(*j*+1,*j*+1:*n*),*A*(*q*,*j*+1:*n*))
22. swap(*A*(*j*+1:*n*,*j*+1),*A*(*j*+1:*n*,*q*))
23. $\beta(j) = \upsilon(j+1)$
24. **end**
25. **if** $j \leq n-2$
26. $L(j+2:n,j+1) = \upsilon(j+2:n)$
27. **if** $v(j+1) \neq 0$
28. $L(j+2:n,j+1) = L(j+2:n,j+1) / \varpi(j+1)$
29. **end**
30. **end**
31. **end**
32. $T = (diagonal(\alpha,0) + diagonal(\beta(2:n),1) + diagonal(\beta(2:n),-1))$
33. $Lz = Pb$ % P matriz de permutaciones %
34. $T\omega = z$
35. $L^T y = \omega$
36. $x = Py$

Al aplicar este método a diferentes tipos de matrices logramos comprobar su estabilidad y la eficiencia en la computación, pero llegamos también a concluir que el método no es eficiente para resolver sistemas dispersos, ya que una matriz *A* dispersa la factoriza en una matriz tridiagonal *T* y una matriz densa triangular inferior *L*. Este efecto lo podemos apreciar en las figuras 27 y 28, donde una matriz con una densidad inicial de 11% se factoriza en una triangular inferior con una densidad de 66%, haciendo inútil el manejo especial de la estructura dispersa de la matriz inicial. Aún con diferentes métodos de reordenamiento se generan demasiados fill-ins y se hace imposible seguir manejando la salida como una matriz dispersa. Una implementación del método de Aasen en el lenguaje Fortan se encuentra en el apéndice B.

FIGURA 27
Fill-in de método Aasen para una matriz en diferentes ordenaciones

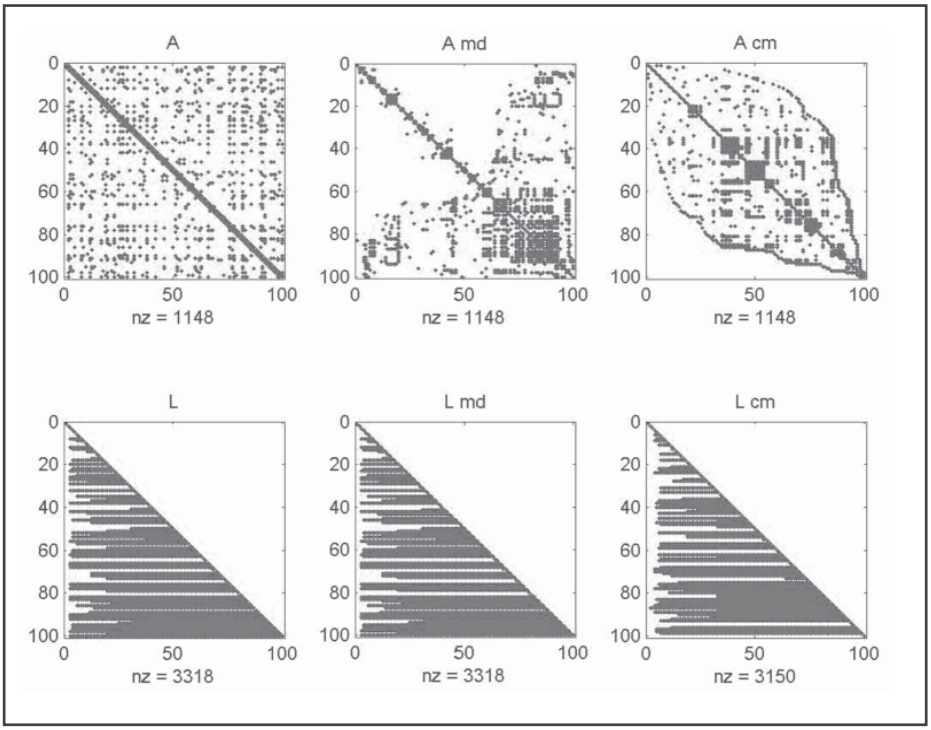
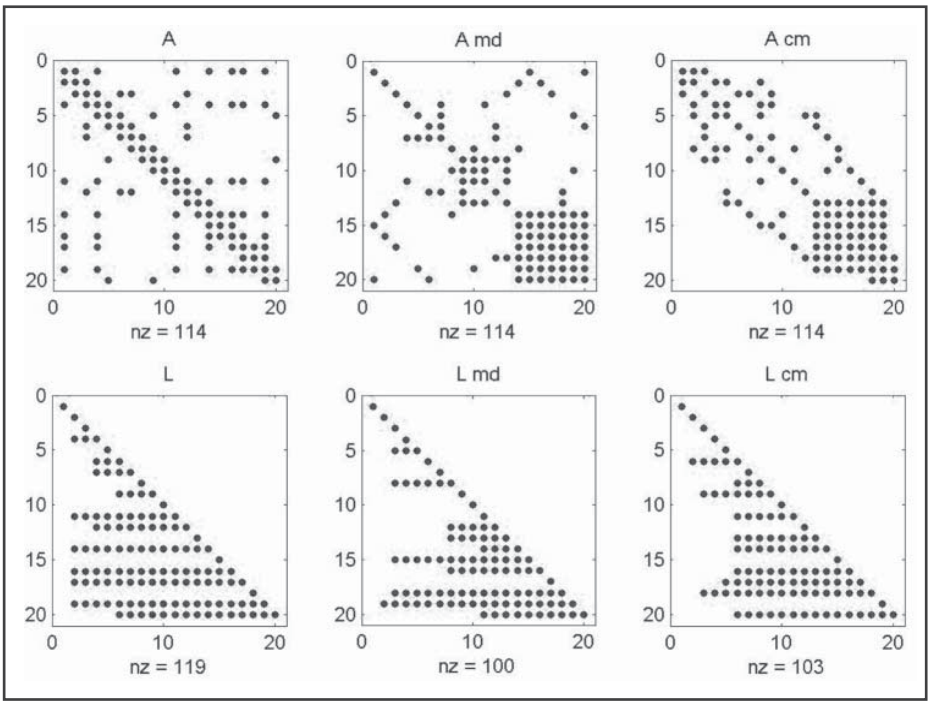


FIGURA 28
Fill-in de método Aasen para una matriz en diferentes ordenaciones



7.4.2 Factorización LDL^T por bloques

El algoritmo de Bunch-Kaufman para la factorización LDL^T por bloques realiza la descomposición de una matriz *A* en una matriz triangular inferior *L* y una matriz diagonal por bloques *D*, donde cada bloque tiene un tamaño 1x1 o 2x2. Un bloque de 2x2 indica que un pivote de 2x2 fue requerido para una eliminación estable de las columnas correspondientes; el elemento subdiagonal correspon-diente en *L* será cero. La librería de optimización LAPACK [31] implementa este algoritmo para la factorización de matrices simétricas e indefinidas en la función *dsytrf*.

En la eliminación de la columna *j* se pueden dar 4 casos [24]:

D1 $|A_{ii}| \geq \alpha |A_{ij}|$, donde $j < I < N$ y $|A_{ij}| = \max_{k=j+1}^{N-1} |A_{kj}|$. Aquí un pivote de 1x1 de A_{ij} será estable, no se necesita intercambio simétrico.

D2 Las condiciones D1 y D4 no se dan. Aquí $A_{j,j}$ es usado como pivote de 1x1 y no se requiere intercambio simétrico.

D3 Un pivote de 1x1 de $A_{i,i}$ será estable. Se hace intercambio simétrico con las filas y columnas *i* y *j*.

D4 Un pivote de 2x2 usando columnas *j* e *i* será estable. Se hace un intercambio simétrico con las filas y columnas *i* y *j*+1. Ambas columnas son eliminadas en este paso.

α es una constante para el algoritmo. El valor $\alpha = \frac{1+\sqrt{17}}{8}$ maximiza la estabilidad de este algo-algoritmo [4]. Para sistemas definidos positivos solo se necesita el caso D1. El caso D3 es requerido también cuando el sistema es semi-definido. El caso D4 se necesita para sistemas indefinidos. El caso D2 existe para prevenir los casos donde D4 pueda ser inestable. Este algoritmo, comparado con la descomposición LTL^T requeriría mayor tiempo de comunicación al ser implementado en su versión paralela. Para un análisis más completo de este algoritmo en su versión densa ver [24].

8
CONCLUSIONES

Para la solución de sistemas de ecuaciones lineales cuya matriz asociada sea dispersa hemos encontrado que los métodos iterativos funcionan de manera adecuada, logrando buenos niveles de convergencia y con grandes posibilidades de implementar los algoritmos de forma paralela.

En el caso de matrices dispersas en el que la factorización no requiera pivoteo y se preserve en gran medida el grado de dispersión observamos que Cholesky y LDL^T funcionan correctamente y se obtienen resultados adecuados en el proceso.

A pesar de que el algoritmo de Gauss-Seidel no se puede paralelizar en el caso de matrices densas, hemos encontrado que si la matriz es dispersa con muchos ceros por debajo de la diagonal principal y con valores diferentes de cero cercanos a la diagonal principal, es posible paralelizar dicho método, teniendo en la cuenta la comunicación entre diferentes procesadores (computadores) de los resultados parciales. Adicionalmente creemos que generar métodos híbridos en los que se aplique un preprocesamiento del sistema, por ejemplo una factorización o reodenamiento para el caso de matrices dispersas, y luego se aplique un algoritmo iterativo en forma paralela puede ser útil para lograr eficiencia en la solución del sistema.

En el caso de los métodos directos hemos encontrado que su aplicación, para matrices asociadas dispersas, no es apropiada porque la dispersión de la matriz se puede perder en los procesos intermedios cuando se intenta hallar la inversa de la matriz de entrada. Para evitar esto existen algunas soluciones que nos permiten conservar la estructura de la matriz y ejecutar el método sin encontrar explícitamente la inversa de la matriz.

Con la experimentación realizada con el método de Aasen encontramos que éste funciona adecuadamente para el caso de matrices densas, pero al aplicarlo a matrices dispersas esta propiedad se pierde, haciendo que el método sea más lento en el sentido que los métodos de almacenamiento para matrices dispersas no son adecuados ni eficientes para el caso de matrices densas.

La utilización de las librerías de optimización para operaciones de álgebra lineal (Blas, Lapack, etc) para la utilización con matrices dispersas se ha comprobado que funcionan de manera eficiente. Sin embargo, éstas librerías nos restringen la estructura de la matriz y no están desarrolladas para utilizar todos los tipos de almacenamiento mencionados en este reporte.

La optimización de los métodos numéricos existentes para la solución de diversos problemas, y más concretamente para la solución de sistemas de ecuaciones lineales es un tema de bastante interés en el mundo de la computación debido a la gran cantidad de problemas que pueden resolverse a partir de estos métodos, y los beneficios que cualquier optimización alcanzada puede generar.

Actualmente existen diferentes estrategias para mejorar la eficiencia de los métodos numéricos para la resolución de sistemas de ecuaciones lineales, entre las cuales se encuentran la explotación de la estructura de las matrices, la utilización de librerías numéricas y la computación paralela. Dichas estrategias no son mutuamente excluyentes y por el contrario se pueden combinar para obtener una máxima eficiencia de los algoritmos, aunque no siempre todas son aplicables a los problemas que se deben resolver.

El desarrollo de métodos específicos para la solución de sistemas de ecuaciones lineales con matrices asociadas dispersas, simétricas e indefinidas es un campo de investigación importante, en el cual se dispone de pocos resultados, según la bibliografía consultada. Las mejoras que se puedan lograr son de gran importancia por las múltiples aplicaciones que tienen y la eficiencia que se puede lograr al resolver dichos sistemas.

Los procesos de investigación requieren de una buena coordinación entre los diferentes participantes, una buena formación teórica y espíritu investigativo, al igual que paciencia y perseverancia. No siempre los resultados obtenidos son los esperados, pero se debe buscar alternativas que puedan mejorar éstos resultados.

Como trabajo futuro planteamos la necesidad de enfatizar en los métodos de ordenamiento de la matriz dispersa de tal manera que al aplicar el método seleccionado se conserve en gran medida el grado de dispersión de la matriz. Adicionalmente es necesario indagar la eficiencia al utilizar otros métodos para la solución de sistemas de ecuación lineales trales como factorización Croult, factorización QR y otros métodos que no se han señalado en este reporte.

9 APÉNDICES

A. MATRICES DISPERSAS EN C

Códigos en el lenguaje C para convertir una matriz a diferentes formatos de almacenamiento disperso, y ejecutar productos matriz-vector.

ALGORITMO A1. Programa principal – md.c

```
1.  /************************************************
2.  Programa para convertir una matriz dispersa
    a diferentes formatos de
3.  almacenamiento y ejecutar productos matriz-
    vector con los mismos.
4.  Por: Juan David Jaramillo J.
5.  ************************************************/
6.  #include “coo.h”
7.  #include “csr.h”
8.  #include “csc.h”
9.  #include “msr.h”
10. #include “msc.h”
11. #include “matrix.h”
12. #include <math.h>
13. #include <stdlib.h>
14.
15. int main(int argc, char *argv[]){
16. //Creación de las variables que se utilizan
17.     int i, j, nz, size=50;
18.     double **A;
19.     double *V, *R;
20.     Coord sparseCoor;
21.     CSC sparseCsc;
22.     CSR sparseCsr;
23.     MSC sparseMsc;
24.     MSR sparseMsr;
25.
26. //Locación de memoria para los vectores
27.     A = (double **)malloc(size*sizeof
        (double*));
28.     for(i=0;i<size;i++){
29.         A[i] = (double *)malloc(size*sizeof
            (double));
30.     }
31.     V = (double *)malloc(size*sizeof(double));
32.     R = (double *)malloc(size*sizeof(double));
33.     sparseCoor.AA = (double *)malloc(size*siz
        e*sizeof(double));
34.     sparseCoor.JC = (int *)malloc(size*size*siz
        eof(int));
35.     sparseCoor.JR = (int *)malloc(size*size*siz
        eof(int));
36.     sparseCsr.AA = (double *)malloc(size*size
        *sizeof(double));
37.     sparseCsr.JA = (int *)malloc(size*size*size
        of(int));
38.     sparseCsr.IA = (int *)malloc(size*size*size
        of(int));
```

```
39.     sparseCsc.AA = (double *)malloc(size*siz
        e*sizeof(double));
40.     sparseCsc.JA = (int *)malloc(size*size*size
        of(int));
41.     sparseCsc.IA = (int *)malloc(size*size*size
        of(int));
42.     sparseMsr.AA = (double *)malloc(size*size
        *sizeof(double));
43.     sparseMsr.JA = (int *)malloc(size*size*size
        of(int));
44.     sparseMsc.AA = (double *)malloc(size*siz
        e*sizeof(double));
45.     sparseMsc.IA = (int *)malloc(size*size*size
        of(int));
46.
47.  //inicializacion del vector V
48.     for(i=0;i<size;i++){
49.        V[i]=i+1;
50.     }
51.
52.  double dens=0.1;
53.  A = fcn_genMatrix(size,dens,1,10);
54.
55.  /*********************************************/
56.  //Imprimir la matriz original
57.     printf("Matriz original...\n");
58.     fcn_printMat(A,size);
59.
60.  //Imprimir vector original
61.     printf("Vector original...\n");
62.     fcn_printVec(V,size);
63.
64.  //multiplicar matriz-vector
65.     R = fcn_mulMatVec(A,V,size);
66.     printf("Producto Matriz-vector...\n");
67.     fcn_printVec(R,size);
68.
69.  /*********************************************/
70.  //Convertir la matriz a formato CSR
71.     nz = fcn_mat2csr(A, size, &sparseCsr);
72.
73.  //Imprimir los componentes del CSR
74.     printf("Formato CSR...\n");
75.     fcn_printCsr(sparseCsr,nz);
76.
77.  //Multiplicar la matriz en CSR por un vector
78.     R = fcn_mulCsrVec(sparseCsr,V,size,nz);
79.     printf("Producto matriz CSR-vector...\n");
80.     fcn_printVec(R,size);
81.
82.  //Convertir la matriz de CSR a formato
     normal
83.     A = fcn_csr2mat(size, nz, sparseCsr);
84.     printf("Convertir de CSR a matriz...\n");
85.     fcn_printMat(A,size);
86.
87.  /*********************************************/
88.  //Convertir la matriz a formato CSC
89.     nz = fcn_mat2csc(A, size, &sparseCsc);
90.
91.  //Imprimir los componentes del CSC
92.     printf("Formato CSC...\n");
93.     fcn_printCsc(sparseCsc,nz);
94.
95.  //Multiplicar la matriz en CSC por un vector
96.    R = fcn_mulCscVec(sparseCsc,V,size,nz);
97.    printf("Producto matriz CSC-vector...\n");
98.    fcn_printVec(R,size);
99.
100.   //Convertir la matriz de CSC a formato
      normal
101.   A = fcn_csc2mat(size, nz, sparseCsc);
102.   printf("Convertir de CSC a matriz...\n");
103.   fcn_printMat(A,size);
104.
105.   /*********************************************/
106.   //Convertir la matriz a formato MSR
107.   nz = fcn_mat2msr(A, size, &sparseMsr);
108.
109.   //Imprimir los componentes del MSR
```

```
110.    printf("Formato MSR...\n");
111.    fcn_printMsr(sparseMsr,nz);
112.
113.    //Multiplicar la matriz en MSR por un
        vector
114.    R = fcn_mulMsrVec(sparseMsr,V,size,nz);
115.    printf("Producto matriz MSR-vector...\n");
116.    fcn_printVec(R,size);
117.
118.    //Convertir la matriz de MSR a formato
        normal
119.    A = fcn_msr2mat(size, nz, sparseMsr);
120.    printf("Convertir de MSR a matriz...\n");
121.    fcn_printMat(A,size);
122.
123.    /*********************************************/
124. //Convertir la matriz a formato MSC
125.    nz = fcn_mat2msc(A, size, &sparseMsc);
126.
127. //Imprimir los componentes del MSC
128. printf("Formato MSC...\n");
129. fcn_printMsc(sparseMsc,nz);
130.
131. //Multiplicar la matriz en MSC por un vector
132. R = fcn_mulMscVec(sparseMsc,V,size,nz);
133. printf("Producto matriz MSC-vector...\n");
134. fcn_printVec(R,size);
135.
136. //Convertir la matriz de MSC a formato
        normal
137. A = fcn_msc2mat(size, nz, sparseMsc);
138. printf("Convertir de MSC a matriz...\n");
139. fcn_printMat(A,size);
140.
141. /*********************************************/
142. //Convertir a coordenadas e imprimir
        vectores
143. nz = fcn_mat2coor(A, size, &sparseCoor);
144.
145. //Imprimir los componentes del Coor
146. printf("Formato Coordenadas...\n");
147. fcn_printCoo(sparseCoor,nz);
148.
149. //Multiplicar la matriz en Coor por un vector
150. R = fcn_mulCooVec(sparseCoor,V,size,nz);
151. printf("Producto matriz Coord-vector...\n");
152. fcn_printVec(R,size);
153.
154. //Convertir la matriz de Coor a formato
        normal
155. A = fcn_coor2mat(size, nz, sparseCoor);
156. printf("Convertir de Coordenadas a
        matriz...\n");
157. fcn_printMat(A,size);
158.
159. return 0;
160 }
```

ALGORITMO A2. Encabezado con funciones para formato COO – coo.h

```
1.  /*************************************************
2.  Fichero: coo.h -> Sparse Matrices stored in
    Coordinates format
3.  Funciones para trabajar matrices dispersas
    almacenadas en formato
4.  de coordenadas.
5.  Por: Juan David Jaramillo J.
6.  *************************************************/
7.  #ifndef COO_H
8.  #define COO_H
9.
10. typedef struct coord Coord;
11.
12. struct coord{
13.  double* AA;
14.  int* JR;
15.  int* JC;
16. };
17.
```

```
18.  //Convertir una matriz a Coordenadas
19.  int fcn_mat2coor(double* A[], int size, Coord
     *sparse){
20.     int i,j;
21.     int cont=0;
22.     for(i=0; i < size; i++){
23.        for(j=0; j < size; j++){
24.        if(A[i][j]!=0){
25.           sparse->AA[cont]=A[i][j];
26.           sparse->JR[cont]=i;
27.           sparse->JC[cont]=j;
28.           cont++;
29.           }
30.       }
31.          }
32.          return cont;
33.  }
34.
35.  //Convertir de Coordenadas a Matriz
36.  double ** fcn_coor2mat(int size, int nz,
     Coord sparse){
37.        int i,j;
38.        double **A;
39.        A = (double **)malloc(size*sizeof(
           double*));
40.    for(i=0;i<size;i++){
41.        A[i] = (double *)malloc(size*sizeof
           (double));
42.      }
43.     for(i=0; i < nz ; i++){
44.  A[sparse.JR[i]][sparse.JC[i]]=sparse.AA[i];
45.     }
46.     return A;
47.  }
48.
49.  //Producto Matriz-Vector
50.  double * fcn_mulCooVec(Coord
     sparse, double V[], int size, int nz){
51.     int i,j;
52.     double *R;
53.     R = (double *)malloc(size*sizeof(double));
54.      for(i=0; i < size; i++){
55.  R[i]=0;
56.      }
57.      for(i=0; i < nz; i++){
58.  R[sparse.JR[i]] += (V[sparse.JC[i]] *
     sparse.AA[i]);
59.      }
60.      return R;
61.  }
62.
63.  //Imprimir componentes de matriz Coord
64.  void fcn_printCoo(Coord sparse, int nz){
65.        int i;
66.        for(i=0;i<nz;i++){
67.        printf("AA: %lf\tJR: %d\tJC: %d\n",
           sparse.AA[i], sparse.JR[i], sparse.JC[i]);
68.        }
69.  }
70.
71.  #endif
```

ALGORITMO A3. Encabezado con funciones para formato CSC – csc.h

```
1.  /************************************************
2.  Fichero: csc.h -> Sparse Matrices stored in
    Compressed Sparse Column format.
3.  Funciones para trabajar matrices dispersas
    almacenadas en formato
4.  de columna comprimida (CSC).
5.  Por: Juan David Jaramillo J.
6.  ************************************************/
7.  #ifndef CSC_H
8.  #define CSC_H
9.
10. typedef struct csc CSC;
11.
12. struct csc{
13.       double* AA;
14.       int* IA;
```

```
15.       int* JA;
16.   };
17.
18.   //Convertir una matriz a CSC
19.   int fcn_mat2csc(double* A[], int size, CSC
      *sparse){
20.       int i,j;
21.        int cont=0;
22.       for(j=0; j < size ; j++){
23.          sparse->JA[j]=cont;
24.          for(i=0; i<size; i++){
25.          if(A[i][j]!=0){
26.              sparse->AA[cont]=A[i][j];
27.              sparse->IA[cont]=i;
28.              cont++;
29.          }
30.       }
31.    }
32.    sparse->JA[size]=cont;
33.    return cont;
34.   }
35.
36.   //Convertir de CSC a Matriz
37.   double ** fcn_csc2mat(int size, int nz, CSC
      sparse){
38.      int i,j;
39.      double **A;
40.      A = (double **)malloc(size*sizeof(double*));
41.      for(i=0;i<size;i++){
42.          A[i] = (double *)malloc(size*sizeof
          (double));
43.      }
44.      j=0;
45.      for(i=0; i < nz ; i++){
46.          while(sparse.JA[j+1]==i){
47.              j++;
48.       }
49.       A[sparse.IA[i]][j]=sparse.AA[i];
50.          }
51.          return A;
52.   }
53.
54.   //Producto Matriz-Vector
55.   double * fcn_mulCscVec(CSC sparse, double
       V[], int size, int nz){
56.       int i,j;
57.       double *R;
58.       R = (double *)malloc(size*sizeof(double));
59.       for(i=0; i < size; i++){
60.     R[i]=0;
61.        }
62.        i=0;
63.        for(j=0; j < nz; j++){
64.       while(sparse.JA[i+1]==j)
65.           i++;
66.       R[sparse.IA[j]] += (V[i] * sparse.AA[j]);
67.     }
68.     return R;
69.   }
70.
71.   //Imprimir componentes de matriz CSC
72.   void fcn_printCsc(CSC sparse, int nz){
73.      int i;
74.      for(i=0;i<nz;i++){
75.          printf("AA: %lf \tIA: %d \tJA: %d
          n",sparse.AA[i],sparse.IA[i],sparse.JA[i]);
76.      }
77.   }
78.
79.   #endif
```

ALGORITMO A4. Encabezado con funciones para formato CSR – csr.h

```
1.  /************************************************
2.  Fichero: csr.h -> Sparse Matrices stored in
    Compressed Sparse Row format.
3.  Funciones para trabajar matrices dispersas
    almacenadas en formato
4.  de fila comprimida (CSR).
5.  Por: Juan David Jaramillo J.
6.  ************************************************/
7.  #ifndef CSR_H
8.  #define CSR_H
9.
10. typedef struct csr CSR;
11.
12. struct csr{
13.    double* AA;
14.    int* JA;
15.    int* IA;
16. };
17.
18. //Convertir una matriz a CSR
19. int fcn_mat2csr(double* A[], int size, CSR
    *sparse){
20.    int i,j;
21.    int cont=0;
22.    for(i=0; i < size ; i++){
23.         sparse->IA[i]=cont;
24.         for(j=0; j<size; j++){
25.         if(A[i][j]!=0){
26.              sparse->AA[cont]=A[i][j];
27.              sparse->JA[cont]=j;
28.              cont++;
29.         }
30. }
31.     }
32.     sparse->IA[size]=cont;
33.     return cont;
34. }
35.
36. //Convertir de CSR a Matriz
37. double ** fcn_csr2mat(int size, int nz, CSR
    sparse){
38.    int i,j;
39.    double **A;
40.    A = (double **)malloc(size*sizeof(double*));
41.    for(i=0;i<size;i++){
42.         A[i] = (double *)malloc(size*sizeof
         (double));
43.    }
44.    i=0;
45.    for(j=0; j < nz ; j++){
46.         while(sparse.IA[i+1]==j){
47.         i++;
48.    }
49.    A[i][sparse.JA[j]]=sparse.AA[j];
50.         }
51.         return A;
52. }
53.
54. //Producto Matriz-Vector
55. double * fcn_mulCsrVec(CSR sparse, double
    V[], int size, int nz){
56.    int i,j;
57.    double *R;
58.    R = (double *)malloc(size*sizeof(double));
59.    for(i=0; i < size; i++){
60. R[i]=0;
61.         }
62.         j=0;
63.         for(i=0; i < nz; i++){
64.      while(sparse.IA[j+1]==i)
65.              j++;
66.      R[j] += (V[sparse.JA[i]] * sparse.AA[i]);
67.         }
68.         return R;
```

```
69.  }
70.
71.  //Imprimir componentes de matriz CSR
72.  void fcn_printCsr(CSR sparse, int nz){
73.      int i;
74.      for(i=0;i<nz;i++){
75.          printf("AA: %lf \tJA: %d \tIA: %d\n",
             sparse.AA[i], sparse.JA[i], sparse.IA[i]);
76.      }
77.  }
78.
79.  #endif
```

ALGORITMO A5. Encabezado con funciones para formato MSC – msc.h

```
1.  /************************************************
2.  Fichero: msc.h -> Sparse Matrices stored in
    Modified Sparse Column format.
3.  Funciones para trabajar matrices dispersas
    almacenadas en formato
4.  de columna comprimida modificado (MSC).
5.  Por: Juan David Jaramillo J.
6.  ************************************************/
7.  #ifndef MSC_H
8.  #define MSC_H
9.
10. typedef struct msc MSC;
11.
12. struct msc{
13.     double* AA;
14.     int* IA;
15. };
16.
17. //Convertir una matriz a MSC
18. int fcn_mat2msc(double* A[], int size, MSC
    *sparse){
19.     int i,j;
20.     for(i=0; i < size ; i++){
21.         sparse->AA[i]=A[i][i];
22.     }
23.     sparse->AA[size]=-1;
24.     int cont=size+1;
25.     for(j=0; j < size ; j++){
26.         sparse->IA[j]=cont;
27.         for(i=0; i<size; i++){
28.         if((A[i][j]!=0)&&(i!=j)){
29.             sparse->AA[cont]=A[i][j];
30.             sparse->IA[cont]=i;
31.             cont++;
32.         }
33. }
34.     }
35.     sparse->IA[size]=cont;
36.     return cont;
37. }
38.
39. //Convertir de MSC a Matriz
40. double ** fcn_msc2mat(int size, int nz, MSC
    sparse){
41.   int i,j;
42.   double **A;
43.   A = (double **)malloc(size*sizeof(double*));
44.   for(i=0;i<size;i++){
45.       A[i] = (double *)malloc(size*sizeof
          (double));
46.    }
47.    for(i=0; i < size ; i++){
48.       A[i][i]=sparse.AA[i];
49.       }
50.       i=0;
51.       for(j=size+1; j < nz; j++){
52.   while(sparse.IA[i+1]==j)
53.           i++;
54.   A[sparse.IA[j]][i] = sparse.AA[j];
55.   }
56.   return A;
57. }
58.
```

```
59.  //Producto Matriz-Vector
60.  double * fcn_mulMscVec(MSC sparse, double
     V[], int size, int nz){
61.       int i,j;
62.       double *R;
63.       R = (double *)malloc(size*sizeof(double));
64.       for(i=0; i < size; i++){
65.            R[i] = V[i] * sparse.AA[i];
66.       }
67.       i=0;
68.       for(j=size+1; j < nz; j++){
69.     while(sparse.IA[i+1]==j)
70.            i++;
71.       R[sparse.IA[j]] += (V[i] * sparse.AA[j]);
72.         }
73.         return R;
74.  }
75.
76.  //Imprimir componentes de matriz MSC
77.  void fcn_printMsc(MSC sparse, int nz){
78.       int i;
79.       for(i=0;i<nz;i++){
80.         printf("AA:  %lf  \tIA:  %d\n",sparse.
          AA[i],sparse.IA[i]);
81.       }
82.  }
83.  #endif
```

ALGORITMO A6. Encabezado con funciones para formato MSR – msr.h

```
1.   /************************************************
2.   Fichero: msr.h -> Sparse Matrices stored in
     Modified Sparse Row format.
3.   Funciones para trabajar matrices dispersas
     almacenadas en formato
4.   de fila comprimida modificado (MSR).
5.   Por: Juan David Jaramillo J.
6.   ************************************************/
7.   #ifndef MSR_H
8.   #define MSR_H
9.
10.  typedef struct msr MSR;
11.
12.  struct msr{
13.       double* AA;
14.       int* JA;
15.  };
16.
17.  //Convertir una matriz a MSR
18.  int fcn_mat2msr(double* A[], int size, MSR
     *sparse){
19.       int i,j;
20.       for(i=0; i < size ; i++){
21.         sparse->AA[i]=A[i][i];
22.       }
23.       sparse->AA[size]=-1;
24.       int cont=size+1;
25.       for(i=0; i < size ; i++){
26.         sparse->JA[i]=cont;
27.         for(j=0; j<size; j++){
28.         if((A[i][j]!=0)&&(i!=j)){
29.             sparse->AA[cont]=A[i][j];
30.             sparse->JA[cont]=j;
31.             cont++;
32.         }
33.       }
34.    }
35.    sparse->JA[size]=cont;
36.    return cont;
37.  }
38.
39.  //Convertir de MSR a Matriz
40.  double ** fcn_msr2mat(int size, int nz, MSR
sparse){
41.    int i,j;
42.    double **A;
```

```
43.    A = (double **)malloc(size*sizeof(double*));
44.    for(i=0;i<size;i++){
45.    A[i] = (double *)malloc(size*sizeof(double));
46.    }
47.    for(j=0; j < size ; j++){
48.       A[j][j]=sparse.AA[j];
49.    }
50.    j=0;
51.    for(i=size+1; i < nz; i++){
52.        while(sparse.JA[j+1]==i)
53.          j++;
54.          A[j][sparse.JA[i]] = sparse.AA[i];
55.       }
56.       return A;
57.    }
58.
59.    //Producto Matriz-Vector
60.    double * fcn_mulMsrVec(MSR sparse, double
       V[], int size, int nz){
61.       int i,j;
62.       double *R;
63.       R = (double *)malloc(size*sizeof(double));
64.       for(i=0; i < size; i++){
65.          R[i]= V[i] * sparse.AA[i];
66.       }
67.       j=0;
68.       for(i=size+1; i < nz; i++){
69.    while(sparse.JA[j+1]==i)
70.          j++;
71.          R[j] += (V[sparse.JA[i]] * sparse.
             AA[i]);
72.       }
73.       return R;
74.    }
75.
76.    //Imprimir componentes de matriz MSR
77.    void fcn_printMsr(MSR sparse, int nz){
78.       int i;
79.       for(i=0;i<nz;i++){
80.          printf("AA: %lf \tJA: %d\n",sparse.
             AA[i],sparse.JA[i]);
81.       }
82.    }
83.
84.    #endif
```

ALGORITMO A7. Encabezado con funciones para matrices densas – matrix.h

```
1.    /*************************************************
2.    Fichero: matrix.h -> Functions for handling
      Matrices
3.    Funciones para operar matrices.
4.    Por: Juan David Jaramillo J.
5.    *************************************************/
6.    #include <time.h>
7.    #ifndef MATRIX_H
8.    #define MATRIX_H
9.
10.   //Generacion de una matriz dispersa
      aleatoria
11.   //den: density. a,b: limits for the values
12.   double ** fcn_genMatrix(int size, double den,
      int a, int b){
13.      int i,j,x,y,n,r;
14.      double **A;
15.      srand (time(NULL));
16.      n=(int)(size*size*den);
17.      A = (double **)malloc(size*sizeof(double*))
         ;
18.      for(i=0;i<size;i++){
19.      A[i] = (double *)malloc(size*sizeof(double))
         ;
20.      }
21.      if(den < 1 && n >= size){
22.         for(i=0;i<size;i++){
23.            r = rand() % (b-a+1) + a;
```

```
            A[i][i] = (double)r;
        }
        j=size;
        while(j<n){
            x= rand()%size;
            y= rand()%size;
            if(!A[x][y]){
                r = rand() % (b-a+1) + a;
                A[x][y] = (double)r;
                j++;
            }
        }
    }
    return A;
}

//Producto Matriz-Vector
double * fcn_mulMatVec(double* M[], double
V[], int size){
    int i,j;
    double *R;
    R = (double *)malloc(size*sizeof(double));
    for(i=0; i < size; i++){
        R[i]=0;
        for(j=0; j < size; j++){
            R[i] += (V[j] * M[i][j]);
        }
    }
    return R;
}
double ** fcn_genMatrix(int size, double den,
int a, int b){
    int i,j,x,y,n,r;
    double **A;
    srand (time(NULL));
    n=(int)(size*size*den);
    A = (double **)malloc(size*sizeof
    (double*));
    for(i=0;i<size;i++){
        A[i] = (double *)malloc(size*sizeof
        (double));
    }
    if(den < 1 && n >= size){
        for(i=0;i<size;i++){
            r = rand() % (b-a+1) + a;
            A[i][i] = (double)r;
        }
        j=size;
        while(j<n){
            x= rand()%size;
            y= rand()%size;
            if(!A[x][y]){
                r = rand() % (b-a+1) + a;
                A[x][y] = (double)r;
                j++;
            }
        }
    }
    return A;
}

//Producto Matriz-Vector
double * fcn_mulMatVec(double* M[], double
V[], int size){
    int i,j;
    double *R;
    R = (double *)malloc(size*sizeof(double));
    for(i=0; i < size; i++){
        R[i]=0;
        for(j=0; j < size; j++){
            R[i] += (V[j] * M[i][j]);
        }
    }
    return R;
}

//Imprimir una matriz double
void fcn_printMat(double * A[], int size){
```

```
97.         int i,j;
98.         for(i=0;i<size;i++){
99.             for(j=0;j<size;j++){
100.                printf("%lf \t", A[i][j]);
101.            }
102.            printf("\n");
103.    }
104.    }
105.
106.    //Imprimir un vector double
107.    void fcn_printVec(double V[], int size){
108.    int i;
109.    for(i=0;i<size;i++){
110.            printf("%lf \t", V[i]);
111.    }
112.    printf("\n");
113.    }
114.    #endif
```

B. MÉTODO DE AASEN CON PIVOTEO EN FORTRAN

Códigos en el lenguaje Fortran para resolver el sistema $Ax = b$ con la factorización de Aasen $PAP^T = LTL^T$.

ALGORITMO B1. Programa principal – descLTL.f90

```
1.      program descLTL
2.      !
3.      ! function [L,T,x,P] = descLTL(A,b)
4.      ! A,b : matriz de coeficientes y vector de
        términos independientes
5.      ! L,T,P: descomposicion PAP'=LTL'
6.      ! x: solucion del sistema
7.      !
8.      implicit none
9.      integer:: i,j,n,iseed(1),info
10.     real(kind=8)::suma
11.     real(kind=8), allocatable:: A(:,:), L(:,:), alfa(:,:),
        betaL(:,:), betaU(:,:), b(:), x(:), y(:), w(:)
12.     integer, allocatable::P(:,:),piv(:)
13.
14.     n=600
15.
16.     allocate(A(n,n))
17.     allocate(L(n,n))
18.     allocate(alfa(n,1))
19.     allocate(betaL(n-1,1))
20.     allocate(betaU(n-1,1))
21.     allocate(P(n,n))
22.     allocate(b(n))
23.     allocate(x(n))
24.     allocate(y(n))
25.     allocate(w(n))
26.     allocate(piv(n))
27.
28.     iseed(1)=5
29.     call random_seed(put=iseed)
```

```
30.  call random_number(A)
31.  A = matmul(A,transpose(A))
32.  x = 1.0
33.  b = matmul(A,x)
34.
35.  call aasen(A,L,alfa,betaL,P,n)
36.  betaU = betaL
37.  b = matmul(P,b)
38.  call sustProg(L,w,b,n)
39.  call dgtsv(n,1,betaL,alfa,betaU,w,n,info)
40.  call sustReg(transpose(L),y,w,n)
41.  x = matmul(P,y)
42.  suma=sum(x)
43.  write(*,*)suma
44.  100 format(D12.5)
45.  deallocate(A,L,alfa,betaL,betaU,P,x,b,y,w,piv)
46.  stop
47.  end program descLTL
```

ALGORITMO B2. Método de Aasen – aasen.f90

```
1.   subroutine aasen(A,L,alfa,B,P,n)
2.
3.   implicit none
4.   integer:: i,j,q,P(n,n)
5.   integer,intent(in)::n
6.   real(kind=8)::v(n,1),T(n,n),h(n,1)
7.   real(kind=8),intent(in)::A(n,n)
8.   real(kind=8),intent(out)::L(n,n),alfa(n,1),
     B(n-1,1)
9.   real(kind=8),allocatable::S1(:,:),S2(:,:),Tmp(:,:)
10.
11.  !
12.  ! inicializar variables
13.  !
14.
15.  v = 0.0
16.  alfa(1,1) = A(1,1)
17.  B = 0.0
18.  P = 0.0
19.  L = 0.0
20.  do i = 1 , n
21.     L(i,i) = 1.0
22.     P(i,i) = 1.0
23.  end do
24.
25.  !
26.  ! bucle principal
27.  !
28.
29.  do j = 1 , n
30.     call compH(A,L,alfa,B,j,n,h);
31.     if (j == 1 .or. j == 2) then
32.        alfa(j,1) = h (j,1);
33.     else
34.        alfa(j,1) = h(j,1) - B(j-1,1) * L(j,j-1)
35.     end if
36.     if (j <= n-1) then
37.        allocate(S1(n-j,j))
38.        allocate(S2(j,1))
39.        allocate(Tmp(n-j,1))
40.        S1(:,:) = L(j+1:n,1:j)
41.        S2(:,1) = h(1:j,1)
42.        Tmp(:,:) = matmul(S1,S2)
43.        v(j+1:n,1) = A(j+1:n,j) - Tmp(:,1)
44.        q = maxloc(abs(v(j+1:n,1)),1)
45.        q = q + j
46.        call inter(A,v,P,L,q,j,n)
47.        B(j,1) = v(j+1,1)
48.        deallocate(S1,S2,Tmp)
49.     end if
50.     if (j <= n-2) then
51.        L(j+2:n,j+1) = v(j+2:n,1) / v(j+1,1)
52.     end if
53.  end do
54.
55.  T = 0.0
56.  do i = 1 , n-1
57.     T(i,i) = alfa(i,1)
58.     T(i+1,i) = B(i,1)
```

```
59.       T(i,i+1) = B(i,1)
60.    end do
61.    T(n,n)=alfa(n,1)
62.
63.    return
64.    end subroutine aasen
```

ALGORITMO B3. Computar columna j de H – compH.f90

```
1.     subroutine compH(A,L,alfa,B,j,n,h)
2.
3.     implicit none
4.     integer:: i,q,k
5.     integer,intent(in)::n,j
6.     real(kind=8)::lcl(n+1),beta(n,1)
7.     real(kind=8),intent(in)::A(n,n),L(n,n),alfa(n,1),
       B(n-1,1)
8.     real(kind=8),intent(out)::h(n,1)
9.
10.    beta(1,1) = 0
11.    beta(2:n,1) = B(:,1)
12.    if (j == 1) then
13.       h(1,1) = A(1,1)
14.    else if (j == 2) then
15.       h(1,1) = beta(2,1)
16.       h(2,1) = A(2,2)
17.    else
18.       lcl(1) = 0
19.       lcl(2) = 0
20.       lcl(3:j) = L(j,2:j-1)
21.       lcl(j+1) = 1
22.       h(j,1) = A(j,j)
23.       do k = 1 , j - 1
24.          h(k,1) = (beta(k,1) * lcl(k)) + (alfa(k,1) *
             lcl(k+1)) + (beta(k+1,1) * lcl(k+2))
25.          h(j,1) = h(j,1) - lcl(k+1) * h(k,1)
26.       end do
27.    end if
28.
29.    return
30.    end subroutine compH
```

ALGORITMO B4. Intercambiar filas en las matrices – inter.f90

```
1.     subroutine inter(A,v,P,L,q,j,n)
2.     ! esta funcion se encarga de intercambiar las
filas y columnas
3.     ! correspondientes. hace la permutacion
simetrica.
4.
5.     implicit none
6.     integer,intent(in)::n,j,q
7.     integer,intent(inout)::P(n,n)
8.     integer::i,Pt(n,n),per(n)
9.     real(kind=8)::t
10.    real(kind=8),intent(inout)::A(n,n),L(n,n),v(n,1)
11.    real(kind=8),allocatable::S1(:,:),S2(:,:),Tmp(:,:)
12.
13.    !
14.    ! v(j+1)<->v(q)
15.    !
16.       t = v(j+1,1)
17.       v(j+1,1) = v(q,1)
18.       v(q,1) = t
19.
20.       per = [1:n]
21.       t = per(j+1)
22.       per(j+1) = per(q)
23.       per(q) = t
24.       Pt = 0.0
25.       do i = 1 , n
26.          Pt(i,i) = 1.0
27.       end do
28.       Pt(:,:) = Pt(per,:)
29.       P(:,:) = matmul(Pt,P)
30.
31.    ! L(j+1,2:j)<->L(q,2:j)
32.       call dswap(j-1,L(j+1,2),n,L(q,2),n);
33.
34.    ! A(j+1,j+1:n)<->A(q,j+1:n)
```

```
35.       call dswap(n-j,A(j+1,j+1),n,A(q,j+1),n)
36.
37.   ! A(j+1:n,j+1)<->A(j+1:n,q)
38.       call dswap(n-j,A(j+1,j+1),1,A(j+1,q),1)
39.
40.   return
41.   end subroutine inter
```

ALGORITMO B5. Sustitución progresiva - sustProg.f90

```
1.    subroutine sustProg(L,x,b,n)
2.
3.    implicit none
4.    integer:: i
5.    integer,intent(in)::n
6.    real(kind=8),intent(in)::L(n,n),b(n,1)
7.    real(kind=8),intent(out)::x(n,1)
8.    real(kind=8)::T(1,1)
9.    real(kind=8),allocatable::S1(:,:),S2(:,:)
10.
11.   x=b
12.   x(1,1)=x(1,1)/L(1,1)
13.   do i=2,n
14.       allocate(S1(1,i-1))
15.       allocate(S2(i-1,1))
16.       S1(1,:)=L(i,1:i-1)
17.       S2(:,1)=x(1:i-1,1)
18.       T=matmul(S1,S2)
19.       x(i,1)=(x(i,1)-T(1,1))/L(i,i)
20.       deallocate(S1,S2)
21.   end do
22.
23.   return
24.   end subroutine sustProg
```

ALGORITMO B6. Sustitución regresiva – sustReg. f90

```
1.    subroutine sustReg(U,x,b,n)
2.
3.    implicit none
4.    integer:: i
5.    integer,intent(in)::n
6.    real(kind=8),intent(in)::U(n,n),b(n,1)
7.    real(kind=8),intent(out)::x(n,1)
8.    real(kind=8)::T(1,1)
9.    real(kind=8),allocatable::S1(:,:),S2(:,:)
10.
11.   x=b
12.   x(n,1)=x(n,1)/U(n,n)
13.   do i=n-1,1,-1
14.       allocate(S1(1,n-i))
15.       allocate(S2(n-i,1))
16.       S1(1,:)=U(i,i+1:n)
17.       S2(:,1)=x(i+1:n,1)
18.       T=matmul(S1,S2)
19.       x(i,1)=(x(i,1)-T(1,1))/U(i,i)
20.       deallocate(S1,S2)
21.   end do
22.
23.   return
24.   end subroutine sustReg
```

REFERENCIAS

[1] O. Schenk and K. Gärtner. *On fast factorization pivoting methods for sparse symmetric indefinite systems*. 2004.

[2] Y. Saad. *Iterative methods for sparse linear systems*. 2000.

[3] V. Kumar. *Introduction to parallel computing*. The Benjamin/Cummings Publishing Company, California. 1994.

[4] G. Golub and C. Van Loan. *Matrix computations*. The Jones Hopkins University press, Baltimore. 1996

[5] G. Montero et al. *Resolución de sistemas de ecuaciones tipo sparse: la estrategia RPK.* 2003.

[6] E. Flórez. *Construcción de inversas aproximadas tipo sparse basada en la proyección ortogonal de Frobenius para el precondicionamiento de sistemas de ecuaciones no simétricos.* Las Palmas de Gran Canaria. 2003.

[7] J. Brandts. *Projection methods for oversized linear algebra problems.* Utrecht. NAW 5/1 nr. 3, September 2000.

[8] D. Becker et al. *Beowulf: A Parallel Workstation for Scientific Computation.* Proceedings, International Conference on Parallel Processing. 1995.

[9] A. Rodríguez. *Paralelismo en Linux.* http://eureka.ya.com/arielrodriguez/tutoriales/linuxpar.html

[10] A. Geist. *PVM: Parallel Virtual Machine*. The MIT Press, Massachusetts. 1994.

[11] A. Vidal and J. Pérez. *Introducción a la programación en MPI*. Universidad Politécnica de Valencia.

[12] CESGA. *Introducción a Fortran 90*. Sección 5: Métodos numéricos. http://www.cesga.es/telecursos/F90/sec5/cap1/Tema5_Cap1_5.html

[13] V. Pérez and H. Herrero. *Métodos matemáticos, apuntes de la asignatura.* Universidad de Castilla-La Mancha. 2002.

[14] V. Sonzogni et al. *Resolución de grandes sistemas de ecuaciones en un cluster de computadoras.* Argentina. 2004.

[15] N. Scenna et al. *Modelado, simulación y optimización de procesos químicos.* Universidad Tecnológica Nacional. Argentina. 1999.

[16] M. Heath. *Parallel numerical algorithms - Cholesky factorization.* University of Illinois. 2004.

[17] M. Heath. *Parallel numerical algorithms - Band and tridiagonal systems*. University of Illinois. 2004.

[18] J. Bunch and R. Marcia. *A pivoting strategy for symmetric tridiagonal matrices*. John Wiley and sons, ltd. EEUU. 2005.

[19] S. Wright. *Modified Cholesky factorizations in interior-point algorithms for linear programming*. US department of energy. EEUU.

[20] F. Villatoro et al. *Métodos directos para ecuaciones lineales*. España. 2002.

[21] P. Matstoms. *Sparse QR Factorization in MATLAB*. Linköping University. Suecia. 1994.

[22] W. Lin. *Finding optimal ordering of sparse matrices for column-oriented parallel Cholesky factorization*. I-Shou University. Taiwan. The Journal of Supercomputing, 24, 259–277, 2003.

[23] A. George and M. Saunders. *Solution of sparse linear equations using cholesky factors of augmented systems*. Canadá. 2000.

[24] P. Strazdins. *A Dense Complex Symmetric Indefinite Solver for the Fujitsu AP3000*. Australian National University. Australia Mathematical Society. 1999.

[25] N. Higham. *Stability of block LDLT factorization of a symmetric tridiagonal matrix*. University of Manchester. Inglaterra. 1998.

[26] I. Duff and S. Pralet. *Strategies for scaling and pivoting for sparse symmetric indefinite problems*. CCLRC Rutherford Appleton Laboratory. Inglaterra. 2004.

[27] B. Lang. *Parallel Direct Methods for Dense Linear Systems*. Institute for Scientific Computing. Aachen University of Technology. Alemania. 2001.

[28] K. Shen et al. *Efficient 2D sparse LU factorization on parallel machines*. University of California. EEUU. 2000.

[29] D. Irony and S. Toledo. *The snap-back pivoting method for symmetric banded indefinite matrices*. Israel Science Foundation. 2002.

[30] G. Karypis et al. *METIS: Family of multilevel partinioning algorithms*. http://www-users.cs.umn.edu/~karypis/metis/metis/index.html

[31] The Netlib Repository: Colección de software matemático de libre distribución. http://www.netlib.org/

[32] R. Burden and J Faires. Análisis numérico. Thomson Learning. México. 2002. Secciones 6 y 7.

Títulos publicados en esta colección

Copia disponible en: www.eafit.edu.co/investigacion/cuadernosdeinv.htm

Cuaderno 1 - Marzo 2002

SECTOR BANCARIO Y COYUNTURA ECONÓMICA EL CASO COLOMBIANO 1990 - 2000 Alberto Jaramillo, Adriana Ángel Jiménez, Andrea Restrepo Ramírez, Ana Serrano Domínguez y Juan Sebastián Maya Arango

Cuaderno 2 - Julio 2002

CUERPOS Y CONTROLES, FORMAS DE REGULACIÓN CIVIL. DISCURSOS Y PRÁCTICAS EN MEDELLÍN 1948 – 1952
Cruz Elena Espinal Pérez

Cuaderno 3 - Agosto 2002

UNA INTRODUCCIÓN AL USO DE LAPACK
Carlos E. Mejía, Tomás Restrepo y Christian Trefftz

Cuaderno 4 - Septiembre 2002

LAS MARCAS PROPIAS DESDE LA PERSPECTIVA DEL FABRICANTE
Belisario Cabrejos Doig

Cuaderno 5 - Septiembre 2002

INFERENCIA VISUAL PARA LOS SISTEMAS DEDUCTIVOS LBPCO, LBPC Y LBPO
Manuel Sierra Aristizábal

Cuaderno 6 - Noviembre 2002

LO COLECTIVO EN LA CONSTITUCIÓN DE 1991
Ana Victoria Vásquez Cárdenas,
Mario Alberto Montoya Brand

Cuaderno 7 - Febrero 2003

ANÁLISIS DE VARIANZA DE LOS BENEFICIOS DE LAS EMPRESAS MANUFACTURERAS EN COLOMBIA, 1995 – 2000
Alberto Jaramillo (Coordinador),
Juan Sebastián Maya Arango, Hermilson Velásquez Ceballos, Javier Santiago Ortiz, Lina Marcela Cardona Sosa

Cuaderno 8 - Marzo 2003

LOS DILEMAS DEL RECTOR: EL CASO DE LA UNIVERSIDAD EAFIT
Álvaro Pineda Botero

Cuaderno 9 - Abril 2003

INFORME DE COYUNTURA: ABRIL DE 2003
Grupo de Análisis de Coyuntura Económica

Cuaderno 10 - Mayo 2003

GRUPOS DE INVESTIGACIÓN
Escuela de Administración
Dirección de Investigación y Docencia

Cuaderno 11 - Junio 2003

GRUPOS DE INVESTIGACIÓN ESCUELA DE CIENCIAS Y HUMANIDADES, ESCUELA DE DERECHO, CENTRO DE IDIOMAS Y DEPARTAMENTO DE DESARROLLO ESTUDIANTIL
Dirección de Investigación y Docencia

Cuaderno 12 - Junio 2003

GRUPOS DE INVESTIGACIÓN - ESCUELA DE INGENIERÍA
Dirección de Investigación y Docencia

Cuaderno 13 - Julio 2003
PROGRAMA JÓVENES INVESTIGADORES – COLCIENCIAS: EL ÁREA DE LIBRE COMERCIO DE LAS AMÉRICAS Y LAS NEGOCIACIONES DE SERVICIOS
Grupo de Estudios en Economía y Empresa

Cuaderno 14 - Noviembre 2003
BIBLIOGRAFÍA DE LA NOVELA COLOMBIANA
Álvaro Pineda Botero, Sandra Isabel Pérez, María del Carmen Rosero y María Graciela Calle

Cuaderno 15 - Febrero 2004
PUBLICACIONES Y PONENCIA 2003
Dirección de Investigación y Docencia

Cuaderno 16 - Marzo 2004
LA APLICACIÓN DEL DERECHO EN LOS SISTEMAS JURÍDICOS CONSTITUCIONALIZADOS
Gloria Patricia Lopera Mesa

Cuaderno 17 - Mayo 2004
PRODUCTOS Y SERVICIOS FINANCIEROS A GRAN ESCALA PARA LA MICROEMPRESA: HACIA UN MODELO VIABLE
Nicolás Ossa Betancur

Cuaderno 18 - Mayo 2004
ARTÍCULOS RESULTADO DE LOS PROYECTOS DE GRADO REALIZADOS POR LOS ESTUDIANTES DE INGENIERÍA DE PRODUCCIÓN QUE SE GRADUARON EN EL 2003
Departamento de Ingeniería de Producción

Cuaderno 19 - Junio 2004
ARTÍCULOS DE LOS PROYECTOS DE GRADO REALIZADOS POR LOS ESTUDIANTES DE INGENIERÍA MECÁNICA QUE SE GRADUARON EN EL AÑO 2003
Departamento de Ingeniería Mecánica

Cuaderno 20 - Junio 2004
ARTÍCULOS RESULTADO DE LOS PROYECTOS DE GRADO REALIZADOS POR LOS ESTUDIANTES DE INGENIERÍA DE PROCESOS QUE SE GRADUARON EN EL 2003
Departamento de Ingeniería de Procesos

Cuaderno 21 - Agosto 2004
ASPECTOS GEOMORFOLÓGICOS DE LA AVENIDA TORRENCIAL DEL 31 DE ENERO DE 1994 EN LA CUENCA DEL RÍO FRAILE Y SUS FENÓMENOS ASOCIADOS
Juan Luis González, Omar Alberto Chavez, Michel Hermelín

Cuaderno 22 - Agosto 2004
DIFERENCIAS Y SIMILITUDES EN LAS TEORÍAS DEL CRECIMIENTO ECONÓMICO
Marleny Cardona Acevedo, Francisco Zuluaga Díaz, Carlos Andrés Cano Gamboa, Carolina Gómez Alvis

Cuaderno 23 - Agosto 2004
GUIDELINES FOR ORAL ASSESSMENT
Grupo de investigación Centro de Idiomas

Cuaderno 24 - Octubre 2004
REFLEXIONES SOBRE LA INVESTIGACIÓN DESDE EAFIT
Dirección de investigación y Docencia

Cuaderno 25 - Septiembre 2004
LAS MARCAS PROPIAS DESDE LA PERSPECTIVA DEL CONSUMIDOR FINAL
Belisario Cabrejos Doig

Cuaderno 26 - Febrero 2005
PUBLICACIONES Y PONENCIAS -2004-
Dirección de investigación y Docencia

Cuaderno 27 - Marzo 2005
EL MERCADEO EN LA INDUSTRIA DE LA CONFECCIÓN - 15 AÑOS DESPUÉS -
Belisario Cabrejos Doig

Cuaderno 28 - Abril 2005
LA SOCIOLOGÍA FRENTE A LOS ESPEJOS DEL TIEMPO: MODERNIDAD, POSTMODERNIDAD Y GLOBALIZACIÓN
Miguel Ángel Beltrán, Marleny Cardona Acevedo

Cuaderno 29 - Abril 2005
"OXIDACIÓN FOTOCATALÍTICA DE CIANURO"
Grupo de Investigación Procesos Ambientales y Biotecnológicos -GIPAB-

Cuaderno 30 - Mayo 2005
EVALUACIÓN A ESCALA DE PLANTA PILOTO DEL PROCESO INDUSTRIAL PARA LA OBTENCIÓN DE ACEITE ESENCIAL DE CARDAMOMO, BAJO LA FILOSOFÍA "CERO EMISIONES"
Grupo de Investigación Procesos Ambientales y Biotecnológicos -GIPAB-

Cuaderno 31 - Junio 2005
LA DEMANDA POR FORMACIÓN PERMANENTE Y CONSULTORÍA UNIVERSITARIA
Enrique Barriga Manrique

Cuaderno 32 - Junio 2005
ARTÍCULOS DE LOS PROYECTOS DE GRADO REALIZADOS POR LOS ESTUDIANTES DE INGENIERÍA MECÁNICA QUE SE GRADUARON EN EL AÑO 2004
Escuela de Ingeniería
Departamento de Ingeniería Mecánica

Cuaderno 33 - Julio 2005
PULVERIZACIÓN DE COLORANTES NATURALES POR SECADO POR AUTOMIZACIÓN
Grupo de Investigación Desarrollo y Diseño de Procesos -DDP-
Departamento de Ingeniería de Procesos
Cuaderno 34 - Julio 2005

"FOTODEGRADACIÓN DE SOLUCIONES DE CLOROFENOL-CROMO Y TOLUENO-BENCENO UTILIZANDO COMO CATALIZADOR MEZCLA DE DIÓXIDO DE TITANIO (TiO_2), BENTONITA Y CENIZA VOLANTE"
Grupo de Investigación Procesos Ambientales y Biotecnológicos -GIPAB-
Edison Gil Pavas

Cuaderno 35 - Septiembre 2005
HACIA UN MODELO DE FORMACIÓN CONTINUADA DE DOCENTES DE EDUCACIÓN SUPERIOR EN EL USO PEDAGÓGICO DE LAS TECNOLOGÍAS DE INFORMACIÓN Y COMUNICACIÓN
Claudia María Zea R., María del Rosario Atuesta V., Gustavo Adolfo Villegas L., Patricia Toro P., Beatriz Nicholls E., Natalia Foronda V.

Cuaderno 36 - Septiembre 2005
ELABORACIÓN DE UN INSTRUMENTO PARA EL ESTUDIO DE LOS PROCESOS DE CAMBIO ASOCIADOS CON LA IMPLANTACIÓN DEL TPM EN COLOMBIA
Grupos de Investigación:
Grupo de Estudios de la Gerencia en Colombia
Grupo de Estudios en Mantenimiento Industrial (GEMI)

Cuaderno 37 - Septiembre 2005
PRODUCTOS Y SERVICIOS FINANCIEROS A GRAN ESCALA PARA LA MICROEMPRESA COLOMBIANA
Nicolás Ossa Betancur
Grupo de Investigación en Finanzas y Banca
Área Microfinanzas

Cuaderno 38 - Noviembre 2005
PROCESO "ACOPLADO" FÍSICO-QUÍMICO Y BIOTECNOLÓGICO PARA EL TRATAMIENTO DE AGUAS RESIDUALES CONTAMINADAS CON CIANURO
Grupo de Investigación Procesos Ambientales y Biotecnológicos -GIPAB-

Cuaderno 39 - Enero 2006
LECTURE NOTES ON NUMERICAL ANALYSIS
Manuel Julio García R.

Cuaderno 40 - Febrero 2006
MÉTODOS DIRECTOS PARA LA SOLUCIÓN DE SISTEMAS DE ECUACIONES LINEALES SIMÉTRICOS, INDEFINIDOS, DISPERSOS Y DE GRAN DIMENSIÓN
Juan David Jaramillo, Antonio M. Vidal Maciá, Francisco José Correa Zabala