UNIVERSIDAD CENTRAL DE VENEZUELA
FACULTAD DE CIENCIAS
ESCUELA DE COMPUTACIÓN

FACTORIZACIÓN LU DE MATRICES DISPERSAS SOBRE UNA ARQUITECTURA MIMD MEDIANTE LA OBTENCIÓN DE CONJUNTOS COMPATIBLES

TRABAJO ESPECIAL DE GRADO
PRESENTADO ANTE LA ILUSTRE
UNIVERSIDAD CENTRAL DE VENEZUELA
POR LOS BACHILLERES
RHADAMÉS ELÍAS CARMONA SUJÚ
Y ADRIANA GABRIELA EGEA GUÉDEZ
PARA OPTAR AL TÍTULO DE
LICENCIADOS EN COMPUTACIÓN

TUTOR: PROF. BRÍGIDA C. MOLINA C.

CARACAS, FEBRERO DE 1995

Jurado

Prof. Brígida Molina

Prof. Rina Surós

Prof. Eurípides Montagne

Prof. Zenaida Castillo

Este libro, en tus manos de estudiante, es instrumento de trabajo para construir tu educación. Cuídalo para que sirva también a los compañeros que te siguen.

"Nunca se te da un deseo
sin que también se te de el poder de hacerlo realidad
... sin embargo, es posible que te cueste trabajo"

Richard Bach
(Ilusiones, pag. 28)

te, es instrumento de trabajo para construir tu educación. Cuídalo para que sirva también a los compañeros que te siguen.

DEDICATORIAS

A **Josefina**, mi madre,
por todo lo que diariamente aprendo de ella,
y por el amor al estudio que me ha inculcado.

Y a **Adriana**,
ya que juntos hicimos realidad este sueño,
y es el estímulo
para culminar de manera exitosa
todas las metas que me proponga.

Rhadamés

A los tres hombres que más he amado en mi vida:

A ti, **Enrique**, mi padre,
por despertar en mí desde niña ese amor al estudio,
por ese orgullo mal disimulado que siempre has sentido por todos mis logros,
te dedico éste ... POR EL PASADO

A ti, **Bernardo**, mi hermanito,
para que nunca, nunca, nunca dejes de soñar,
pues como ves, a veces, si tienes fe, puedes ver tus sueños hechos realidad,
te dedico éste ... POR EL FUTURO

A ti, **Rhadamés**, mi complemento,
por este maravilloso tiempo que estamos viviendo,
por nuestros esfuerzos diarios, y por sus frutos, que nos unen cada vez más,
te dedico éste ... POR EL PRESENTE

Adriana

AGRADECIMIENTOS

A **Dios**, nuestro amoroso Padre, por iluminar nuestro entendimiento, y guiar nuestros esfuerzos hasta la consecución de esta meta. Gracias a la Santísima Virgen **María**, por interceder por nosotros.

A nuestras familias, por entendernos y apoyarnos siempre (tanto en lo económico, como en lo moral), por estar allí en todas esas noches de desvelos interminables, y sobre todo, por soportar el tecleo nocturno en medio de susurros, solo interrumpido por el inevitable ruido de nuestras respectivas impresoras. Especialmente gracias a **Pili** y **Josefina** por todas sus oraciones... al despertar de madrugada.

A nuestra **U-U-UCV**, "la Casa que vence las sombras", por hacernos instrumentos para vencer las sombras que se ciernen sobre la Venezuela de los últimos tiempos, y sobre todo, por hacernos más que venezolanos: sensibles y críticos... algo que todo ucevista sabe comprender.

A todos los profesores que nos formaron, no sólo en lo académico en el aula, sino también en los pasillos, por ese estímulo que la mayoría de ellos nos ha brindado. Particularmente, gracias a **Eleonora, Otilio, Zenaida, Willy** y **Nora**, por su respeto y cariño, por estar tan pendiente de nosotros día a día

Al **LCPD**, por las valiosas oportunidades que nos ha brindado, no sólo la de realizar allí la tesis, sino, y sobre todo, por hacernos copartícipes del progreso (tal vez algo lento, pero real) de nuestra querida Escuela, que definitivamente no es la misma que nos vio entrar. Gracias a **Marta, Rina, Brígida, Eurípides,** y **Rodolfo** por hacernos ver que el científico de hoy no es el estereotipo del hombre "aparentemente sin sentimientos", solitario, sabelotodo, gris y de lentes; sino que es un ser humano completo, alegre, con familia y amistades, que no lo sabe todo, pero tiene hambre de seguir aprendiendo,... y bueno, por advertirnos que lo de los lentes viene con el tiempo.
Muy especialmente a **Claudia** por estar tan pendiente de nosotros, y por sus sugerencias para este trabajo. A **Mercedes** y sus errores involuntarios le debemos la experiencia que ahora tenemos resolviendo problemas técnicos de la PARSYTEC.

A la gente del **CESMa**, particularmente a **Javier** y **Jairo**, por facilitarnos los trámites necesarios para el uso de la PARSYTEC en la USB durante nuestras últimas pruebas.

A nuestra tutora y madre adoptiva, **Brígida**, gracias por mostrarnos el mundo del paralelismo de forma tan atractiva que hace dos años renunciamos a nuestras primeras inclinaciones (Rhadamés a Lenguajes y Programación; y Adriana a Arquitectura y Sistemas), para tomar como electivas materias de Matemáticas de la Computación. No lo lamentamos, tal vez jamás habríamos estado tan cerca de lo que cada uno quería hacer: diseñamos e implantamos algoritmos sobre una arquitectura especial para resolver un problema numérico. En serio que gracias.

A nuestros respectivos amigos de la niñez, quienes (cuando nos permitíamos un descanso) nos brindaron pruebas de que los destinos de quienes se aman no se separan con el tiempo ni la distancia: gracias **Richard "Momox"**, y gracias **Soraya** por animarnos tanto. Asimismo, gracias a la amistad y cariño de nuestros compañeros ucevistas (¡pronto colegas!) **Liliana, Ludwig, Arsene, Norita, Armando,** y muy especialmente a **Soralys** por nuestras comunicaciones electrónicas casi diarias justo en mitad de alguna modificación del programa.

A todos los alumnos que tuvimos siendo preparadores, quienes sin saberlo, hacían que nos sintiésemos más comprometidos con la carrera que con este paso concluimos. Gracias a **Cecilio** y a **Pedro,** por ayudarnos tan desinteresadamente en esos detalles de último minuto, pero indispensables, y aunque no nos acepten las gracias.

A quienes ahora se nos escapan de la memoria, que nos han ayudado de alguna forma y nos van a preguntar después por qué no los escribimos aquí:

Gracias, de todo corazón

Rhadamés y Adriana

Indices

INDICE DE TEMAS

INDICE DE FIGURAS

INDICE DE TABLAS

Introducción

Nunca pienso en el futuro.
Llega demasiado rápido

Albert Einstein

Existe un sinfín de problemas que surgen de manera natural en diferentes áreas de la ciencia y la tecnología, los cuales se resuelven mediante el planteamiento y solución de uno o varios sistemas de ecuaciones lineales de gran tamaño, que suelen ser dispersos. Ejemplos comunes de ellos son: simulación de yacimientos, modelación de plantas químicas, simulación de circuitos de alta escala de integración, análisis estructural, ecuaciones en derivadas parciales, ... etc.

Cuando se manejan sistemas tan grandes, en caso de que el vector independiente varíe, y la matriz permanezca intacta, resulta conveniente resolverlos vía factorización LU, ya que ésta se realiza una vez por matriz. Además de ello, si la matriz es dispersa, como en los casos citados, debe aprovecharse el hecho de que la mayoría de sus elementos son nulos para reducir tanto los cálculos como el almacenamiento empleado, y por ello, se han desarrollado diferentes métodos para resolver este tipo de sistemas.

Sin embargo, dichos métodos tienen, en general, una sobrecarga de procesamiento (por ejemplo, al generar y acceder estructuras de datos complejas, al hallar el ordenamiento adecuado,...etc. [GEO-81]), y puede decirse que los algoritmos secuenciales han llegado a su límite en cuanto a la reducción de su complejidad en tiempo. Incluso, también en la implantación de estos algoritmos en un computador secuencial existen límites

físicos dados por la arquitectura, cada vez más cerca de alcanzarse con los últimos avances de la tecnología. Es así como el procesamiento paralelo ha revolucionado al mundo científico, ya que promete grandes mejoras en tiempo con respecto a la forma tradicional de hacer las cosas. Un algoritmo secuencial, entonces, comienza a ser adaptado, o completamente desechado, en el diseño de diferentes algoritmos paralelos cuyos resultados sean los mismos, aunque los tiempos de ejecución puedan variar mucho dependiendo de la arquitectura en la que sean implantados. No obstante, es interesante destacar que son muchas las veces en que esta promesa contra el tiempo no es cumplida por los algoritmos paralelos, ya que los tiempos de comunicación entre procesadores son altamente significativos sobre el tiempo total de procesamiento, y no siempre los investigadores han desarrollado algoritmos que mejoren al respectivo secuencial.

Por lo antes expuesto, resulta importante e interesante el proponer algoritmos para resolver la factorización LU de matrices dispersas, que sean cada vez más eficientes en la utilización de los recursos computacionales disponibles, como bien puede serlo una red de procesadores con determinadas características.

En este trabajo se paralelizó de manera novedosa la factorización LU de matrices dispersas, mediante la obtención de un conjunto de pivotes que cumplan ciertas propiedades para que puedan procesarse simultáneamente, lográndose muy buenos resultados en tiempo de ejecución. En el Capítulo 1 ofrecemos un resumen de los principales conceptos básicos necesarios para abordar la factorización LU de matrices dispersas en paralelo, que se ha dividido en dos secciones: Álgebra Lineal y Computación Paralela, las cuales podrían evitar sin problemas los conocedores en descomposición LU dispersa y procesamiento paralelo respectivamente. En el Capítulo 2, nos referiremos a algunos trabajos realizados en torno al problema en cuestión, en cuyos resultados nos basamos para diseñar el algoritmo paralelo propuesto, el cual se presenta con detalle en el Capítulo 3. Finalmente, en el Capítulo 4, presentamos un estudio de los tiempos de ejecución obtenidos, y son comparados con los arrojados por un programa secuencial comercial

muy conocido en el área de matrices dispersas: el Y12M, mostrándose una aceleración creciente a medida que se incrementa el número de procesadores. Adicionalmente, incluimos un apéndice en el que se describen algunas rutinas provistas por el lenguaje de programación utilizado, y otros detalles de implantación.

Capítulo 1

Conceptos Básicos

" No somos estudiantes de materias,
sino estudiantes de problemas,
y los problemas pueden traspasar los límites
de cualquier materia de estudio ó disciplina..."

Karl Popper.

El propósito de este capítulo es mostrar el basamento teórico en que nos hemos apoyado para el desarrollo del presente trabajo especial de grado, y por tratar conceptos relacionados con dos diferentes áreas de la computación, lo hemos dividido en las siguientes secciones:

1) Álgebra Lineal:
   Presentamos los algoritmos clásicos de *Eliminación Gaussiana* y *Factorización LU* como métodos directos para la resolución de sistemas de ecuaciones lineales, una variante: la factorización LU con matrices dispersas, y algunos hechos importantes a considerar para la paralelización de la misma.

2) Computación Paralela:
   Inicialmente veremos el por qué del auge del paralelismo, particularmente en el campo del análisis numérico, para luego considerar una clasificación que divide a los computadores según cómo relacionan sus instrucciones con los datos que están siendo procesados. Presentaremos también una arquitectura paralela ejemplo del modelo MIMD: la *PARSYTEC MC-3 DE.*

1.- ÁLGEBRA LINEAL

Como nuestra meta es la factorización LU de matrices dispersas, usada para resolver sistemas de ecuaciones lineales, es necesario que recordemos las siguientes definiciones y métodos:

Definición 1.1

Un *Sistema de Ecuaciones Lineales* de orden n consiste de n ecuaciones con n variables, que puede ser representado en forma matricial como el problema

$$Ax = b \qquad \textbf{(1.1)}$$

donde $A_{n\times n}$ es la matriz de coeficientes, $b \in \Re^n$ es el vector independiente, y $x \in \Re^n$ es el vector incógnita o vector solución. Efectuar el cálculo necesario para hallar x (o un valor aproximado a x) constituye la *resolución numérica del sistema de ecuaciones lineales.*

a.) Eliminación Gaussiana

La eliminación Gaussiana es un método directo que permite la reducción de un sistema de ecuaciones lineales a un sistema triangular superior equivalente al original, el cual se resuelve posteriormente por sustitución hacia atrás.

Básicamente, el procedimiento es el siguiente: la primera ecuación se usa para eliminar la primera incógnita de las n-1 ecuaciones restantes; luego, la segunda ecuación modificada se usa para eliminar la segunda incógnita de las n-2 ecuaciones restantes. Este proceso se repite hasta alcanzar la penúltima fila, momento en que se ha obtenido un sistema triangular sencillo de resolver. Cabe destacar que el orden de las ecuaciones es arbitrario, y el método puede realizarse según cualquier ordenamiento.

Sea el problema (1.1) expresado en forma matricial como se aprecia en (1.2)

$$Ax = b \equiv \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{pmatrix} \qquad \textbf{(1.2)}$$

El procedimiento específico de operaciones aritméticas usadas en este método es el siguiente: Supongamos que $a_{0,0} \neq 0$; entonces se puede eliminar x_0 de las n-1 ecuaciones restantes tomando de la i-ésima ecuación los *multiplicadores*:

$$m_{i,0} = \frac{a_{i,0}}{a_{0,0}} \qquad 1 \leq i < n \qquad \textbf{(1.3)}$$

de la primera columna de A. Entonces, para ilustrar mejor el proceso, incorporamos a la notación un supraíndice en el cual se indique el último paso de eliminación donde se actualizó una posición dada, el sistema (1.2) se transforma en:

$$\begin{pmatrix} a_{0,0}^{(0)} & a_{0,1}^{(0)} & \cdots & a_{0,n-1}^{(0)} \\ a_{1,0}^{(1)} & a_{1,1}^{(1)} & \cdots & a_{1,n-1}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0}^{(1)} & a_{n-1,1}^{(1)} & \cdots & a_{n-1,n-1}^{(1)} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b_0^{(0)} \\ b_1^{(1)} \\ \vdots \\ b_{n-1}^{(1)} \end{pmatrix} \qquad \textbf{(1.4)}$$

donde los nuevos coeficientes están dados por

$$\begin{aligned} a_{i,j}^{(1)} &= a_{i,j}^{(0)} - m_{i,0} \cdot a_{0,j}^{(0)} \\ b_i^{(1)} &= b_i^{(0)} - m_{i,0} \cdot b_0^{(0)} \end{aligned} \qquad 1 \leq i, j < n \qquad \textbf{(1.5)}$$

Aplicando sucesivamente esta operación sobre la submatriz de orden menor (n-1 × n-1), finalmente se obtiene una matriz triangular superior, y un vector b modificado, convertiendo el problema $Ax=b$ en $Ux=\hat{b}$,que se resuelve mediante el *método de ascenso* o *sustitución hacia atrás*, en el que en forma general, la solución viene dada por:

$$x_k = \frac{b_k^{(k)} - \sum_{j=k+1}^{n-1} a_{k,j}^{(j)} \cdot x_j}{a_{k,k}^{(k)}} \qquad k = n-1,\ldots,0 \qquad \textbf{(1.6)}$$

b.) Factorización LU

En el método de la *Factorización LU* se construye una matriz L en la que de forma conveniente se conservan los multiplicadores hallados durante la Eliminación Gaussiana, y es después de realizado que se toma en cuenta al vector independiente para la resolución del sistema, el cual se resuelve en las siguientes cinco etapas:

1. Descomponer A para obtener L, U, π y ρ , que satisfagan:

$$A_{\pi_i,\rho_j} = (LU)_{i,j} \qquad \forall i,j \,/\, 0 \le i,j < n \qquad \textbf{(1.7)}$$

   donde L es una matriz triangular inferior de orden $n \times n$, con unos en la diagonal, U es una matriz triangular superior de orden $n \times n$, y π y ρ son vectores de permutación de orden n, los cuales surgen en esta ecuación debido a que las filas y columnas pueden haber sido permutadas durante la factorización LU por algún interés particular, como preservar *estabilidad numérica* y además asegurar que también L y U sean dispersas.

2. Permutar b de acuerdo a $c_i = b_{\pi_i}$, $0 \le i < n$, para obtener un vector c.
3. Resolver el sistema triangular $Ly = c$ para obtener un vector y. Este sistema se resuelve mediante *sustitución hacia adelante*.
4. Resolver el sistema triangular $Uz = y$ para obtener un vector z. Este sistema se resuelve por sustitución hacia atrás.

5. Permutar z de acuerdo a $x_{\rho_j} = z_j$, $0 \leq j < n$, para obtener el vector solución x.

Existen seis formas de llevar a cabo la factorización LU, dependiendo de como se recorra la matriz en cada paso, y del orden en que son hallados los multiplicadores respecto a las actualizaciones. De todas ellas detallaremos el algoritmo para la forma *kji*, que es orientada por columnas, tanto el cálculo de los multiplicadores, como las actualizaciones. Haremos abstracción, por simplicidad, del posible proceso de intercambio de filas y/o columnas, así como del paso de factorización que actualizó un elemento dado, volviendo a la notación de supraíndice cuando sea necesario.

Para $k=0$ *hasta* $n-2$ *hacer*
  Para $s=k+1$ *hasta* $n-1$ *hacer*
    $l_{s,k} = a_{s,k}/a_{k,k}$
  Fpara
  Para $j=k+1$ *hasta* $n-1$ *hacer*
    Para $i=k+1$ *hasta* $n-1$ *hacer*
      $a_{i,j} = a_{i,j} - l_{i,k} a_{k,j}$
    Fpara
  Fpara
Fpara

En este algoritmo, se fija el número del paso k, con el pivote asociado $a_{k,k}$; se calculan los multiplicadores $l_{i,k}$, con i=k+1, k+2, ... , n-1, y se actualiza cada columna de la matriz no factorizada.

Como puede observarse, la factorización LU exige que los pivotes $a_{k,k}$ sean no nulos, y en caso de encontrar uno, es necesario alterar el orden de las ecuaciones de la submatriz remanente para encontrar un nuevo pivote que no sea cero, realizando permutaciones por fila y/o columna pues el método puede realizarse según cualquier ordenamiento y obtener resultados idénticos teóricamente. Sin embargo, al implantar la factorización LU en un computador suele ser conveniente intercambiar ecuaciones, aún cuando no se encuentren pivotes ceros, ya que los cálculos se efectúan con una precisión finita, y este hecho afecta drásticamente los resultados, requiriendo

entonces que sea reexaminada la selección de pivotes. Para mantener la *estabilidad numérica* durante la descomposición LU y así obtener una solución confiable, es necesario incorporar un mecanismo que evite que elementos de la matriz con valores muy cercanos a cero sean tomados como pivotes, como por ejemplo, el *pivoteo parcial*.

b.1) Matrices Dispersas

A continuación definiremos un tipo especial de matrices para las cuales hay que tomar consideraciones adicionales durante su factorización, en la que no basta controlar la estabilidad numérica.

Definición 1.2

Una *matriz dispersa* es aquella en la que la "mayoría" de sus elementos son cero. Sin embargo, no existe un criterio absoluto que permita establecer la cantidad de elementos nulos que debe tener una matriz para que sea considerada dispersa, debido a que es un término relativo al algoritmo a implantar, ya que depende si éste explota o no la existencia de los ceros de la matriz.

Ahora bien, un algoritmo explota la existencia de los ceros de la matriz si posee estructuras de datos adecuadas, en las que no se almacenen los elementos nulos, y evita que se realicen cálculos sobre estos elementos. En el caso que nos ocupa, la *Factorización LU*, es necesario además evitar que la matriz factorizada se vuelva densa para hacer buen uso de los recursos computacionales. Para explicar mejor esto, debemos observar la diferencia existente entre los siguientes dos eventos que comúnmente ocurren durante la factorización:

1. Actualización: En una actualización, una posición a_{ij} *existente* cambia su valor numérico, y en el caso particular en que su nuevo valor sea cero, ocurre una *cancelación*, que se trata modificando la estructura de la matriz, eliminando la entrada en cuestión de dicha estructura.

2. Rellenos: Un relleno, (*"fill-in"*), surge cuando una posición *ai,j* que *no existe* en la estructura toma un valor numérico distinto de cero. Esto implica que debe asignarse el espacio físico para dicha posición, y hacer las modificaciones necesarias en las estructuras que permitan acceder a ella.

La ocurrencia de muchos rellenos pueden hacer que de una matriz A dispersa, se obtengan factores L y U densos ambos, siendo esto a todas luces indeseable, como se explicó anteriormente, siendo necesario idear un mecanismo de *control de rellenos*.

El relleno a generarse en la descomposición LU puede ser controlado mediante la escogencia apropiada de los elementos pivotes, siendo utilizada una estrategia heurística de búsqueda de pivotes llamada *estrategia de Markowitz* [MAR-57]. La escogencia de un elemento pivote $a_{i,j}$ conlleva a la creación de a lo más $r_{i,j}$ rellenos, con

$$r_{i,j} = (\eta_{FIL_i} - 1)\cdot(\eta_{COL_j} - 1) \qquad \textbf{(1.8)}$$

donde η_{FIL_i} es el número de elementos no cero de la fila *i*, y η_{COL_j} es el número de elementos no cero de la columna *j* (de la matriz reducida). Este límite superior es llamado *número de Markowitz* de $a_{i,j}$, y la estrategia consiste en tomar como pivote a aquel elemento que posea el menor número de Markowitz, número al cual llamaremos de ahora en adelante *mincount*.

Tal mecanismo consiste en aceptar sólo aquellos candidatos a pivotes $a_{i,j} \neq 0$ tales que satisfagan:

$$|a_{i,j}| \geq u \cdot \max_k \{|a_{k,j}|\} \qquad 0 \leq u \leq 1 \qquad \textbf{(1.9)}$$

donde *u* es llamado *parámetro umbral*, suministrado por el usuario [DUF-86]. Al tomar u=0 no se fijan restricciones, y se toma cualquier candidato, mientras que para el caso de u=1 se tiene una estrategia muy exigente,

porque en lo general se obtendrá sólo un candidato. Es entonces necesario utilizar un valor intermedio, como por ejemplo u=0.25, para el que se obtienen buenos resultados.

A continuación, presentamos una variante del algoritmo de factorización LU, que incorpora los mecanismos adecuados para trabajar con matrices dispersas: el control de rellenos y de estabilidad numérica:

```
Para k=0 hasta n-2 hacer
    tomar a_{w,k} de mínimo Nº de Markowitz entre los a_{i,k}
    que satisfacen la condición de estabilidad
    Intercambiar filas w y k.
    Para s=k+1 hasta n-1 hacer
      si a_{s,k} ≠ 0 entonces
         l_{s,k} = a_{s,k}/a_{k,k}
      fsi
    Fpara
    Para j=k+1 hasta n-1 hacer
      Para i=k+1 hasta n-1 hacer
        si l_{i,k} ≠ 0 y a_{k,j} ≠ 0 entonces
          si a_{i,j} ≠ 0 entonces
             a_{i,j} = a_{i,j} - l_{i,k}.a_{k,j}        (Actualizaciones)
          sino
             a_{i,j} = -l_{i,k}.a_{k,j}                 (Rellenos)
          fsi
        fsi
      Fpara
    Fpara
Fpara
```

Una de las formas de paralelizar este algoritmo es repartir las columnas de A entre los procesadores, debido a que la actualización está orientada por columnas. En cada paso k, el procesador con la columna k deberá enviar los elementos no nulos de la columna k, desde la posición de $a_{k,k}$ hasta $a_{n-1,k}$, a todos los procesadores. Cada procesador hallará los multiplicadores $l_{s,k}$, $s=k+1,...,n-1$, y actualizará sus columnas asignadas no factorizadas.

En el siguiente algoritmo se hacen actualizaciones de rango uno de la forma $a_{i,j} = a_{i,j} - l_{i,k}.a_{k,j}$.

```
Para k=0 hasta n-2 hacer
    si ColumnaAsignada(k) entonces
      BroadCast de la columna k (a_{k,k}..a_{n-1,k})
    sino
      Recibir la columna k
    fsi
    tomar a_{w,k} de mínimo Nº de Markowitz entre los a_{i,k}
    que satisfacen la condición de estabilidad.
    Intercambiar filas w y k.
    Para s=k+1 hasta n-1 hacer
      si a_{s,k} ≠ 0 entonces
          l_{s,k} = a_{s,k}/a_{k,k}
      fsi
    Fpara
    Para j=k+1 hasta n-1 hacer
      si ColumnaAsignada(j) entonces
          Para i=k+1 hasta n-1 hacer
              si l_{i,k} ≠ 0 y a_{k,j} ≠ 0 entonces
                  si a_{i,j} ≠ 0 entonces
                      a_{i,j} = a_{i,j} - l_{i,k}.a_{k,j}
                  sino
                      a_{i,j} = -l_{i,k}.a_{k,j}
                  fsi
              fsi
          Fpara
      fsi
    Fpara
Fpara
```

2.- COMPUTACIÓN PARALELA

Existe un sinfín de aplicaciones que demandan computadores veloces para ejecutar muchos cálculos rápidamente o procesar un gran conjunto de datos. Ejemplos de ellas son la exploración petrolera y solución de grandes sistemas de ecuaciones diferenciales que surgen de simulaciones numéricas en diversas disciplinas. Ningún computador actual produce la velocidad de procesamiento requerida por estas aplicaciones, y el paralelismo parece ser una vía para obtener mayor velocidad en el cálculo, pues con la ejecución de varias operaciones simultáneamente, el tiempo de cálculo puede ser reducido de manera significativa.

Ahora bien, cualquier computador, ya sea secuencial o paralelo, ejecuta instrucciones sobre datos. Un flujo de instrucciones (el programa) le indica al computador qué hacer en cada paso, y éstas instrucciones afectan a un flujo de datos (las entradas del programa). De acuerdo a la forma en que el conjunto de datos es afectado por el conjunto de instrucciones, pueden ser definidos diferentes modelos de computadores, y es así como se definen los siguientes cuatro modelos, dependiendo de si existen uno o múltiples flujos de instrucciones o datos, ejecutados u operados por un procesador:

a.) Simple flujo de Instrucciones - Simple flujo de Datos (SISD)
b.) Múltiple flujo de Instrucciones - Simple flujo de Datos (MISD)
c.) Simple flujo de Instrucciones - Múltiple flujos de Datos (SIMD)
d.) Múltiple flujos de Instrucciones - Múltiple flujos de Datos (MIMD)

Examinaremos a continuación en detalle únicamente nuestro modelo de interés en el contexto de este trabajo, los computadores MIMD, cuya arquitectura es mostrada en la Figura 1.1. Esta clase de computadores es la más general en el paradigma de la computación paralela que los clasifica de acuerdo a la secuencia de datos y/o instrucciones. Poseen p procesadores (p>1), cada uno operando bajo el control de una secuencia de instrucciones emitida por su propia unidad de control; así los procesadores están

potencialmente todos ejecutando diferentes programas sobre datos diferentes. Esto significa que los procesadores operan de manera asíncrona, y vale la pena mencionar que los algoritmos asíncronos son difíciles de diseñar, evaluar e implantar. Un algoritmo asíncrono es una colección de procesos que se ejecutan simultáneamente sobre un número disponible de procesadores.

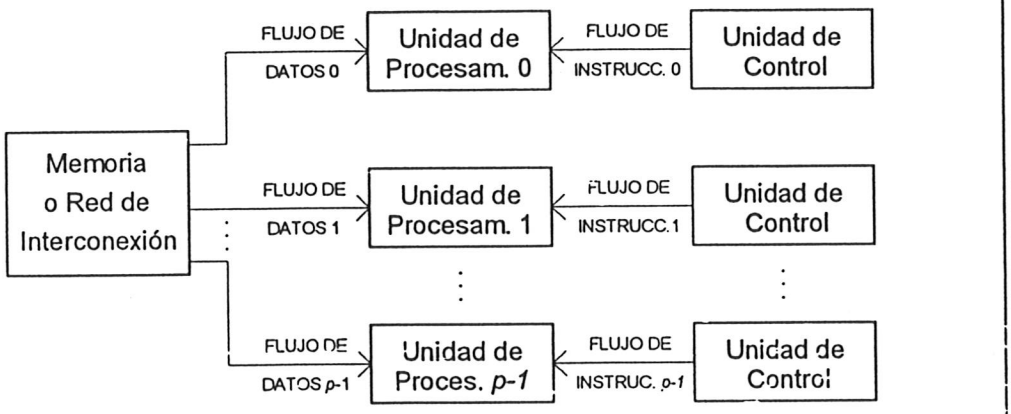


Figura 1.1: COMPUTADORES MIMD

Los procesadores se comunican entre sí ya sea a través de una memoria común, o por medio de una red de interconexión, y este hecho subdivide esta clase de computadores en otras dos, a saber:

a.) Arquitecturas MIMD a Memoria Compartida
b.) Arquitecturas MIMD a Memoria Distribuida

a.) Arquitecturas MIMD a Memoria Compartida

Esta clase de computadores es también conocida como modelo de máquina paralela de acceso aleatorio (PRAM). Aquí p procesadores ($p > 1$) comparten una memoria común y Cuando se comunican, lo hacen a través de

la misma. Si se desea transmitir un dato desde el procesador P_i al procesador P_j, esto lo realiza en dos pasos: en el primero, el procesador P_i escribe el dato en una dirección de memoria conocida por el procesador P_j, y en el segundo, el procesador P_j lee esa localización.

El modelo básico permite a todos los procesadores acceder simultáneamente la memoria compartida, si la posición de memoria que ellos están tratando de escribir es diferente. Sin embargo, el tipo de memoria puede dividir este modelo en cuatro subclases dependiendo de si dos o más procesadores pueden acceder a la misma posición de memoria simultáneamente [AKL-89], a saber:

1. *Lectura-Exclusiva, Escritura-Exclusiva*:
2. *Lectura-Concurrente, Escritura-Exclusiva*
3. *Lectura-Exclusiva, Escritura-Concurrente*
4. *Lectura-Concurrente, Escritura-Concurrente*

El permitir múltiples lecturas simultáneas sobre la misma posición de memoria no debe ocasionar ningún problema. Conceptualmente, si cada procesador requiere leer desde una misma posición de memoria copia el contenido de esa posición y lo almacena en su memoria local. Sin embargo, si varios procesadores requieren escribir simultáneamente diferentes datos sobre la misma posición de memoria, debe existir una manera determinística de especificar el contenido de esa posición de memoria una vez realizadas las escrituras. En estos casos (subclases 3 y 4), los conflictos de escritura se resuelven por hardware, mientras que para las subclases 1 y 2, es el sistema operativo que los resuelve, dando como resultado que diferentes ejecuciones de un mismo programa posiblemente arrojen resultados distintos, dada la aleatoriedad descrita.

b.) Arquitecturas MIMD a Memoria Distribuida

La otra forma de comunicación entre los procesadores es a través de una red de interconexión. En este modelo la memoria es dividida entre el

conjunto de procesadores, para su acceso local; además, cada procesador es conectado con sus vecinos a través de una línea bidireccional de comunicación, que le permite enviar o recibir datos en cualquier instante de tiempo, habiéndose desarrollado una amplia variedad de topologías que permiten abarcar una gran cantidad de problemas de manera eficiente, tales como: el *arreglo lineal,* el *anillo*, la *malla*, el *toroide*, el *árbol*, el *fat-tree*, y el *hipercubo* [RUK-94], que tienen una baja cantidad de enlaces entre procesadores, de manera tal que cuando sea necesario comunicar un mensaje entre dos procesadores que no tienen conexión directa, debe encaminarse o enrutarse dicho mensaje por procesadores intermedios entre éstos dos.

Un algoritmo muy utilizado para la comunicación de procesadores en este tipo de arquitecturas es el conocido como *"fan-in"*. En un *fan-in* realizado entre p procesadores, se tienen $Log_2(p)$ etapas de comunicación. En la Figura 1.2 podemos observar un ejemplo con p=8 (y por consiguiente 3 etapas), en el que se aprecia los procesadores que intervienen en cada etapa, y según el sentido de las flechas, se indica cuáles procesadores envían y cuáles reciben en un momento dado.

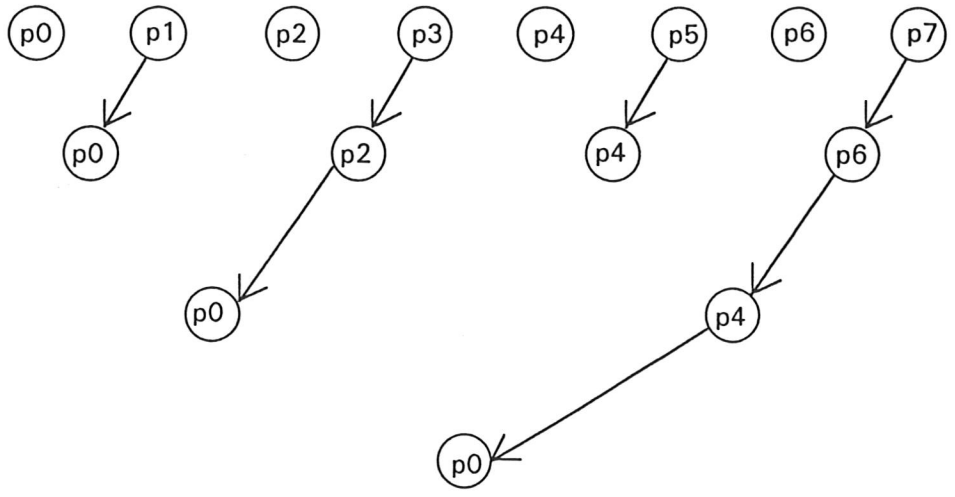


Figura 1.2: COMUNICACIONES EN UN FAN-IN DE 8 PROCESADORES

Por ejemplo, en una máquina hipercúbica, los mensajes que se transmiten en cada etapa del *fan-in* no necesitan pasar por procesadores intermedios para llegar a su destino. Por ejemplo, si comparamos las figuras 1.2 y 1.3, la comunicación que se realiza en el *fan-in* es punto a punto.

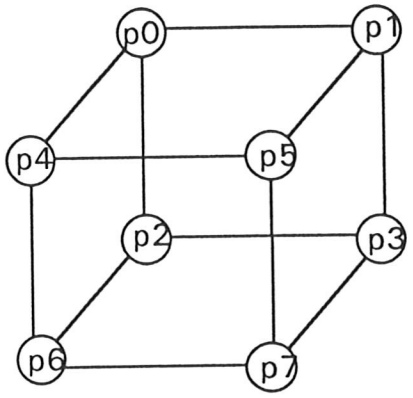


Figura 1.3: HIPERCUBO DE 8 PROCESADORES

Otro representante importante de este tipo de arquitectura son las máquinas basadas en *transputers.* La palabra *Transputer*, formada de la unión de los términos TRANsistor y comPUTER, señala el interés de sus creadores de proponer un componente que, al igual que los transistores en los circuitos eléctricos, sirva de base para los sistemas masivamente paralelos [RUK-94]. Un *transputer* es un microcomputador que contiene un procesador, memoria local y capacidad de comunicaciones a través de enlaces que permiten conectarlo con otros transputers u otros dispositivos, todo en un solo chip, por lo que nos permitiremos el abuso de considerar sinónimos las palabras "transputers" y "procesadores", usándolos indistintamente de ahora en adelante.

Cada transputer cuenta con cuatro enlaces de comunicación que le permiten formar parte de redes de procesadores en diversidad de topologías.

Sin embargo, se dipone de *crossbar* o *switches* programables que permiten conectar lógicamente cualquier canal de un transputer con diferentes canales en diferentes instantes de tiempo, y además, permiten que la comunicación entre canales pueda ser realizada entre transputers ó entre un transputer y otro dispositivo. Generalmente, es el transputer 0 el considerado "raíz", por estar conectado al computador anfitrión ó *Host*, pero, como se dijo, cualquier transputer puede además estarlo, y ese hecho varía de una arquitectura a otra.

Al conectar varios procesadores entre sí, de acuerdo a alguna topología, se crean máquinas del procesamiento paralelo a un costo relativamente bajo, lo cual hace que cobren cada vez mayor importancia como arquitecturas dentro del modelo de computación MIMD a memoria distribuida, siendo de nuestro especial interés en este contexto debido a que la arquitectura paralela sobre la cual desarrollamos este trabajo, está basada en transputers.

b.1) PARSYTEC MultiCluster-3 DE

La *PARSYTEC MultiCluster-3 DE* (*MC-3 DE*), sirve como una estación de desarrollo local para algoritmos masivamente paralelos, ya que permite desarrollar, probar y optimizar aplicaciones antes de cargarlas en una máquina de "producción" [PAR-91], es decir, una máquina masivamente paralela de costo mucho mayor.

Un miembro de esta familia, la máquina *PARSYTEC MC-3/14 DE,* está constituida por un mueble que soporta hasta 14 módulos de transputers, que pueden acomodarse en configuraciones mixtas de hasta 28 procesadores. Los tipos de procesadores que soporta son: T225, T805 y T9000, PowerPC-601 y PowerPC-604, con memorias cuyos tamaños varían desde 64 KB SRAM hasta 32 MB DRAM.

Una *PARSYTEC MC-3 DE* puede conectarse a un computador anfitrión (*"Host"*) a través de uno a cuatro transputers, cada uno de los cuales

debe pertenecer a una partición distinta de la red de procesadores, lo que permite que la máquina sea efectivamente multiusuario, pudiendo coexistir tantos usuarios como particiones se hayan definido, excluyendo la partición p_n, la cual contiene en total n de procesadores. Cada uno de los usuarios en un momento dado accede a la red desde el computador anfitrión a través de enlaces exclusivos. Sin embargo, cada usuario se restringe al uso de la cantidad de procesadores de la partición asignada, y en el caso de que éste desee utilizarlos todos, debe solicitar (ya sea explícitamente o no) la partición p_n, y hasta que concluya la ejecución, se bloquea el acceso a otros usuarios.

Capítulo 2

Antecedentes

"...indudablemente, no podemos conocer bien el mundo por medio de la teoría. La práctica es absolutamente necesaria, pero seguramente, para un joven que piense emprender un viaje hacia ese país lleno de laberintos, vueltas y recovecos, es de gran utilidad tener, por lo menos, un mapa general del mismo, hecho por algún viajero experimentado."

Lord Chesterfield.

En este capítulo haremos una revisión de algunos trabajos realizados anteriormente sobre el problema que nos ocupa, con el fin de considerar resultados importantes a los que se llegaron, y así tomar algunas ideas exitosas para el algoritmo propuesto, el cual se presentará en el capítulo 3. Primero haremos un repaso general de estrategias utilizadas en la factorización LU de matrices dispersas sobre arquitecturas secuenciales (Sección 1: *Arquitecturas Secuenciales*), para luego pasar a estudiar cómo han resuelto algunos investigadores este mismo problema sobre diferentes arquitecturas paralelas del tipo MIMD, siendo tratadas por separado aquellas que poseen memoria compartida (Sección 2: *Arquitecturas a Memoria Compartida*) y las de memoria distribuida, basadas en sistemas de pase de mensajes (Sección 3: *Arquitecturas a Memoria Compartida*).

1.- ARQUITECTURAS SECUENCIALES

La descomposición LU dispersa en forma secuencial normalmente consiste de muchas etapas, cada una de las cuales involucra una búsqueda de

un elemento pivote sobre la matriz reducida, seguida de permutaciones por fila y por columna, y una actualización de rango 1 en dicha matriz.

Una de las variantes de la estrategia básica de Markowitz descrita en el capítulo anterior es la de *Zlatev* [ZLA-80], quien restringió la búsqueda de elementos pivotes a las tres filas más dispersas, de manera de evitar largas búsquedas, y sus experimentos demostraron que no necesariamente el relleno total generado se reduce por el hecho de buscar en más filas. Además, es importante preservar la estabilidad numérica durante la descomposición LU con el fin de obtener una solución confiable. Existen programas secuenciales de propósito general para la descomposición LU dispersa, tales como MA28 [DUF-79] y Y12M [ZLA-80], que controlan el relleno, manteniendo la estabilidad.

Para la implementación de un algoritmo paralelo que realice la factorización LU dispersa, debe entonces tomarse en cuenta un buen criterio que asegure la estabilidad numérica, que deberá conjugarse con otro que asegure un alto paralelismo (evitando que la submatriz remanente en cada paso se vuelva densa), para alcanzar una combinación aceptable en términos de confiabilidad y rapidez del algoritmo.

El paralelismo potencial que posee la factorización LU dispersa es doble, ya que además del paralelismo inherente a las operaciones matriciales, utilizado en la paralelización de la factorización LU densa (pueden ser actualizados diferentes filas o columnas a la vez), se tiene potencialmente un paralelismo adicional por ser dispersa la matriz, y puede darse la ejecución paralela de ciertos cálculos que tendrían que realizarse secuencialmente en el caso denso. En el caso disperso, varias actualizaciones de rango 1 sobre la matriz pueden combinarse en una actualización de rango múltiple, evitando la sincronización y espera de procesadores después de cada actualización de rango 1.

Calahan [CAL-73] fue el primero en explotar el hecho de que dos elementos pivotes $a_{i,j}$ y $a_{k,l}$ pueden procesarse simultáneamente si

$$a_{i,l} = 0 \quad \wedge \quad a_{k,j} = 0 \tag{2.1}$$

Tales pivotes son llamados *compatibles* o *independientes*. Es de observar que si dos pivotes $a_{i,j}$ y $a_{k,l}$ son compatibles, y $a_{k,l}$ y $a_{r,s}$ también lo son, entonces $a_{i,j}$, $a_{k,l}$ y $a_{r,s}$ no necesariamente son compatibles. Decimos entonces que la relación de compatibilidad es *reflexiva* y *simétrica* pero no transitiva, por lo cual, para que un conjunto de pivotes sea compatible, es necesario que lo sean dos a dos, y por ello el problema de hallar conjuntos compatibles de gran tamaño es generalmente considerado como de alto costo computacional.

2.- ARQUITECTURAS A MEMORIA COMPARTIDA

Son muchos los investigadores que han aprovechado el paralelismo potencial de la factorización LU dispersa sobre este tipo de arquitecturas paralelas, tales como *Smart y White* [SMA-88], *Alaghband* [ALA-89], *Davis y Yew* [DAV-90], *Gallivan, Sameh y Zlatev* [GAL-91], entre otros. En estos algoritmos, se modifica la estrategia básica de Markowitz para obtener un conjunto S de elementos pivotes que puedan ser procesados simultáneamente, tales que satisfagan la relación de compatibilidad descubierta por *Calahan*. Para hacer este conjunto lo más grande posible, se aceptan cierta cantidad de elementos pivotes con un número de Markowitz superior a *mincount*.

Smart y White [SMA-88] investigaron la complejidad en tiempo paralelo de la descomposición LU dispersa para un número ilimitado de procesadores, usando la profundidad de los grafos de tareas como una medida de complejidad. Ellos presentan un algoritmo en el cual el conjunto pivote *S* contiene elementos de la diagonal compatibles con un número de Markowitz entre *mincount* y *mincount+a*, donde *a* es un parámetro de entrada. El conjunto *S* es construido partiendo del conjunto vacío y añadiendo sucesivamente nuevos pivotes compatibles con *S* en orden creciente de

número de Markowitz. Para una matriz tridiagonal de 1000x1000, el algoritmo con a=2 tiene asociado un grafo de tareas de profundidad 27, valor muy cercano al óptimo teórico de 23; y en este caso, la estrategia básica de Markowitz tiene asociado un grafo de tareas de profundidad 1998. Sin embargo, para muchas de las matrices que examinaron, provenientes de circuitos eléctricos, la mejora en profundidad respecto de la estrategia de Markowitz fue de tan sólo el 50%.

Alaghband y Jordan [ALA-89] presentan un algoritmo de eliminación Gaussiana con control de rellenos en un multiprocesador a memoria compartida, que genera conjuntos de candidatos a pivotes, y entonces escoge el conjunto pivote S de máximo tamaño. Ellos comparan en su trabajo dos estrategias diferentes para la escogencia de un conjunto de pivotes compatibles entre todos los maximales compatibles obtenidos mediante el método descrito. La primera estrategia, llamada *suma de Markowitz* selecciona aquel conjunto en el cual la suma de los números de Markowitz de todos sus elementos es mínima. Esta suma representa el número de actualizaciones y/o rellenos en la matriz A que ocurren cuando este conjunto de pivotes es utilizado para reducir la matriz, y por tanto es una cota superior del número de rellenos que puedan ocurrir. La segunda estrategia empleada es la llamada *producto de Markowitz ("ORed Markowitz")* donde se usa una matriz booleana B asociada a la matriz sparse bajo consideración con el fin de contar los elementos no ceros en la unión de filas de B, para los pivotes en el conjunto, y multiplicar este número por el número de elementos no ceros en la unión de columnas de B de estos pivotes. Este producto es la cantidad de elementos de A distintos que serán actualizados en un paso paralelo usando este conjunto de pivotes, y es entonces el relleno máximo posible. El criterio selecciona el conjunto maximal compatible que minimice el producto de Markowitz entre todos los de tamaño más grande.

De su estudio sobre la escogencia de pivotes paralelos, tenemos que destacar dos conclusiones importantes:

1. El uso del número de Markowitz para formar un conjunto compatible ayuda a controlar el relleno; pero la forma en que los números de Markowitz son combinados para formar un criterio de selección para el conjunto no es de especial importancia.
2. Si se reduce ligeramente el paralelismo en cada paso, descartando aquellos pivotes con más alto Markowitz, el relleno total puede reducirse significativamente, y con ello, el costo total de la factorización de la matriz.

También *Davis* y *Yew* [DAV-90] presentan un algoritmo paralelo a memoria compartida y un programa que tiene la completa funcionalidad de programas secuenciales tales como MA28 y Y12M. En este algoritmo, el conjunto pivote *S* contiene elementos compatibles con un número de Markowitz entre *mincount y a×mincount,* donde *a* es un parámetro de entrada (usaron *a*=4 en sus experimentos). Todos los procesadores buscan candidatos a pivote aceptables, y tratan de incluirlos en el conjunto actual *S*. Si un candidato es compatible con todos los elementos de *S*, entonces es añadido a *S*. Los conflictos que puedan existir entre procesadores que intenten añadir simultáneamente un pivote son prevenidos por *secciones críticas* en el programa. Aquel procesador que llegue primero a la sección crítica gana el acceso a él. Esto implica que el sistema operativo es un factor influyente en la escogencia del pivote, haciendo que el programa sea no determinístico. Los experimentos realizados en el computador de memoria compartida *Alliant FX/8* mostraron que D2 es en promedio 3.9 veces más rápido en ocho procesadores que en un procesador, y que la versión secuencial de D2 es 4.3 veces más rápida que el programa secuencial MA28.

Finalmente, *Gallivan, Sameh* y *Zlatev* [GAL-91] presentan tres versiones paralelas a memoria compartida del programa secuencial Y12M:

- Y12M1: el cual está basado en actualizaciones de rango1;
- Y12M2: el cual está basado en actualizaciones de rango m; y
- Y12M3: el cual explota el paralelismo de grano grueso convirtiendo a la matriz A en una triangular superior en bloques.

Aquí, la mejora del paralelismo del programa Y12M2 respecto del Y12M1 es significativa. En el algoritmo Y12M2, se relaja el criterio de compatibilidad (2.1) exigiendo solamente que $a_{k,j} = 0$ para que $a_{i,j}$ preceda al elemento $a_{k,l}$ en el ordenamiento del conjunto S de m pivotes "compatibles". Esta relajación del requerimiento de compatibilidad permite la creación de conjuntos pivotes mucho más grandes, a expensas de una actualización de la matriz mucho más complicada. Los resultados experimentales en el computador paralelo a memoria compartida *Alliant FX/80* con 8 procesadores mostraron que el programa Y12M1 es más rápido que el Y12M2 para matrices que son relativamente densas o se vuelven densas durante el proceso de factorización, mientras que el Y12M2 es más rápido para matrices que son muy dispersas y permanecen siéndolo. La aceleración de Y12M1 está entre 2.4 y 5.4, mientras que la de Y12M2 está entre 2.3 y 5.0, para un conjunto de 27 matrices de prueba de la colección de matrices dispersas Harwell-Boeing [DUF-92].

3.- ARQUITECTURAS A MEMORIA DISTRIBUIDA

Muchos investigadores han trabajado también sobre esta clase de computadores, ya sea diseñando nuevos algoritmos, o adaptando algunos diseñados inicialmente para máquinas paralelas a memoria compartida. Sin embargo la experiencia de *Echeverría* [ECH-92] en la adaptación de las ideas de *Alaghband y Jordan* a un sistema de memoria distribuida, ha demostrado que no siempre se producen buenos resultados en tiempo de ejecución, y de hecho, su implantación paralela tuvo tiempos mayores al programa secuencial.

Un algoritmo concebido para una arquitectura a memoria distribuida, considerando que en este caso la comunicación es un factor de gran peso (a diferencia del caso de memoria compartida), debe, en principio, tener mejor comportamiento que un algoritmo adaptado. Un ejemplo de esto, lo constituyó el trabajo de Van Der Stappen, Bisseling y Van De Vorst [VAN-93], que

realizaron la descomposición LU de una matriz dispersa general sobre una red de transputers con una topología de malla cuadrada de q^2 procesadores.

En el algoritmo, la matriz se distribuye entre los procesadores de acuerdo a la distribución de malla, o *grid*, de tal forma que al procesador *P(s,t)* (el cual se encuentra en la fila *s* y columna *t* de la malla) se le asignan los elementos $a_{i,j}$ tal que (i mod q) = s y (j mod q) = t, con $0 \leq s,t < q$, y los vectores de permutación π y ρ distribuidos de tal forma que π_i sea conocido por los procesadores *P(i mod q, *)*, y ρ_j por *P(*, j mod q)*. De igual forma se tienen distribuidos los vectores π y ρ de orden n, los cuales almacenan el número de elementos no nulos de cada fila y columna respectivamente.

Para encontrar un conjunto de pivotes compatibles *S*, descomponen este subproblema a su vez en tres partes:

a) Buscar candidatos a pivotes

b) Hallar las incompatibilidades entre los candidatos encontrados; y

c) Construir el conjunto compatible a partir de las incompatibilidades encontradas entre los candidatos .

a) Buscar candidatos a pivotes:

La búsqueda de candidatos se separa por columnas de procesadores. Cada columna de procesadores *P(*,t)* debe obtener *ncol* candidatos a pivote, donde *ncol* es un parámetro de entrada.

Ahora bien, el algoritmo busca en las *ncol* columnas más dispersas los candidatos a pivote, basándose en la heurística empleada por Zlatev [ZLA-80]. Así, se define *SearchCols(t)* como el conjunto de columnas asociadas a los procesadores *P(*,t)* en donde se van a buscar los candidatos. Por cada columna de *SearchCols(t)*, se elige el pivote de menor número de Markowitz entre los que preserven la estabilidad de la matriz reducida, según el criterio (1.9), para construir L y U dispersas. Como cada columna *j* de la matriz está distribuida entre los procesadores *P(*, j mod p)*, esta búsqueda requiere mucha comunicación.

El conjunto de candidatos obtenidos en los procesadores *P(*,t)* se ubican en el conjunto *ColCandidates(t)*. Los procesadores *P(0,*)* unen todos los candidatos en el conjunto *Candidates*, mediante un pipeline formado por dichos procesadores. Luego de la unión, *P(0,q-1)* hace un broadcast de *Candidates*, ordenado por número de Markowitz, para que se conozca en cada procesador.

b) Compatibilidad de los candidatos

El procesador *P(s,t)* construye un conjunto de "incompatibilidades" entre pares de candidatos. Dicho conjunto se define como:

$$InComp(s,t) = \{(i,j,i',j') \;/\; (i \underline{mod}\ q=s) \text{ y } (j' \underline{mod}\ q=t) \text{ y } (a_{i,j'}\neq 0)\}$$

donde los índices (i, j) e (i', j') son conocidos por todos los procesadores, debido al broadcast de *Candidates;* y el valor de $a_{i,j'}$ es conocido por *P(s,t)* por la distribución de los datos. Basta que dicho valor no sea cero para lo que los candidatos $a_{i,j}$ y $a_{i',j'}$ no sean compatibles, ya que se viola una de las dos condiciones de compatibilidad descritas en (2.1).

Para la construcción de *InComp(s,t)* el procesador *P(s,t)* no necesita comunicación alguna; sin embargo, para la construcción del conjunto compatible definitivo, se requiere utilizar todos los conjuntos *InComp(s,t)* hallados.

c) Construcción del conjunto de pivotes:

Para la construcción del conjunto definitivo (Conjunto Compatible *S*), se usa un pipeline entre los procesadores *P(0,t)*, $0 \leq t < q$. Inicialmente, se reparten los candidatos equitativamente, de tal forma que cada candidato de *P(0,t)* tiene menor o igual número de Markowitz que los de *P(0,t+1)*. Además, a *P(0,t)* se le asignan las incompatibilidades de sus elementos entre sí, y la de éstos con los candidatos de los procesadores predecesores. Es de notar que hay una fuerte comunicación, ya que las incompatibilidades están distribuidas

entre todos los procesadores. El proceso completo puede verse en detalle en [VAN-93].

En cuanto a las pruebas realizadas con este algoritmo, se tomaron como parámetros de entrada *ncol*=1 y *u*=0.1, dando buenos resultados en tiempo total de ejecución, usando como matrices de prueba varias de las existentes en la colección de matrices dispersas de Harwell-Boeing.

Capítulo 3

Algoritmo Paralelo

"Hay pocas cosas que se pueden mejorar mucho,
pero si mejoramos poco muchas cosas,
las cosas pueden mejorar mucho"

Rhadamés y Adriana

En este capítulo se presenta un algoritmo paralelo novedoso que realiza la factorización LU de matrices dispersas, mediante la obtención de conjuntos compatibles, los que permiten actualizaciones múltiples de la matriz, logrando que en un solo paso paralelo se realizan varios pasos de factorización. Dichos conjuntos están constituídos por pivotes que preservan la estabilidad numérica de la matriz reducida y son generadores de poco relleno (para obtener matrices L y U también dispersas). Los detalles se muestran a continuación.

1.) Distribución de la carga:

La matriz A se distribuye entre los p procesadores en forma intercalada por columnas; de tal forma que la columna j es asignada al procesador $j \underline{mod}\ p$, $\forall j = 0..n-1$. Así, se denota el conjunto de índices de las columnas asignadas al procesador t como *ColAsignadas(t)*, de tal manera que:

$$ColAsignadas(t) = \{ j \mid j \underline{mod}\ p = t, \forall\ j=0..n-1 \}$$

Mediante esta distribución de la matriz A, cada procesador tiene asignadas n/p columnas de A si n/p es exacta; de lo contrario, los procesadores $P_0, P_1, \ldots, P_{n \bmod p-1}$ tendrán asignada una columna más.

Los vectores π y ρcorrespondientes a la permutación por filas y columnas respectivamente, son conocidos en todos los procesadores. El número de elementos no cero por filas (ηFil) será conocido también por todos los procesadores, mientras que por columnas (ηCol) será distribuido de igual forma que las columnas de *A*. Así, ηCol_j es conocido sólo por el procesador $P_{j \bmod p}$.

A continuación se muestra el algoritmo paralelo a alto nivel, para luego detallar los subproblemas que lo ameriten.

k = 0
π = Identidad
ρ = Identidad
ηFil_i = #NoCeros(Fila$_i$); con i=0..n-1
ηCol_j = #NoCeros(Columna$_j$); con j=0..n-1
mientras (k<n-1) hacer
Obtener conjunto compatible CC
m = |CC|
Actualizar π_i y ρ_j, k≤i,j<n
Calcular los $l_{i,j}$, con k<i<n, y k≤j<k+m
Actualización de la matriz remanente en la factorización
Actualizar ηFil_i y ηCol_j, k+m≤i,j<n
k = k+m
fMientras

2.) Obtener conjunto compatible CC:

Obtener el conjunto compatible más grande en cada paso, disminuye el número de pasos paralelos requeridos para la factorización de la matriz. Sin embargo, se sabe que el orden de complejidad para obtener el conjunto compatible más grande es NP-Completo [ALA-89], si se toman como candidatos los elementos de la diagonal, asumiendo que son no nulos y que preservan la estabilidad de la matriz; e incluso, hallar el conjunto compatible más grande entre los elementos no nulos de la matriz, que preserven la estabilidad de la misma tiene mayor complejidad.

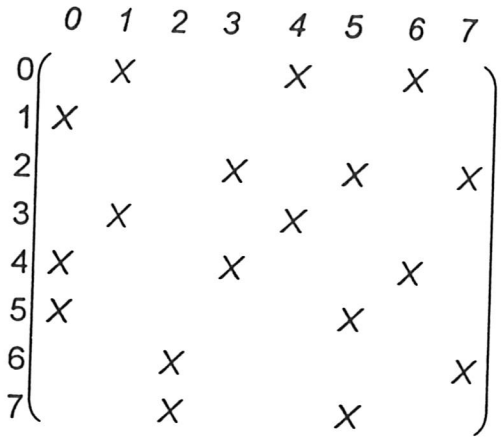
Encontrar el mejor conjunto compatible reduce el número de actualizaciones de rango m, pero por otra parte, el tiempo total para factorizar la matriz puede ser muy grande por el costo que amerita dicha búsqueda. Es de

considerar que el conjunto compatible más grande, puede generar muchos rellenos, que involucran cálculo en futuros pasos de factorización, por lo que puede influir en el tiempo total de ejecución, y convertir a la matriz en densa. Es por todo esto que en la práctica no se busca el conjunto compatible más grande, sino aquél que genere poco relleno y sus elementos preserven la estabilidad de la matriz, y que la complejidad en tiempo para encontrarlo sea muy inferior a NP-completo.

La heurística propuesta para hallar conjuntos compatibles se puede descomponer en seis pasos, a realizarse por cada procesador P_t:

1) Seleccionar un conjunto de columnas entre las no factorizadas, llamado *ColumnasCand(t)*.
2) Por cada columna j $\in$ *ColumnasCand(t)*, obtenemos un pivote que preserve la estabilidad de la matriz, con mínimo número de Markowitz. El conjunto de estos candidatos será llamado *CandPiv(t)*.
3) Entre los candidatos de *CandPiv(t)* se halla un conjunto compatible llamado *Comp(t)*.
4) Cooperar con los otros procesadores para obtener un conjunto *Conj* de tal forma que *Conj = Comp(0) U Comp(1) U ... U Comp(p-1)*, y que luego sea conocido por todos los procesadores.
5) Se eliminan los elementos de *Conj* con mayor número de Markowitz que no son compatibles (según la información de la matriz presente en el procesador), obteniendo un conjunto *Prod(t)* en donde sus candidatos no necesariamente son compatibles.
6) Cooperar con los otros procesadores para hallar la intersección de los conjutos *Prod(t)*, $0 \leq t < p$. El conjunto intersección lo denominaremos CC (conjunto compatible), y será conocido por todos los procesadores para llevar a cabo la fase de factorización.

A continuación se detalla cada paso, y usaremos para ayudar a su entendimiento, una matriz dispersa A_{8x8} y una red de cuatro procesadores (p=4). Distribuyendo las columnas entre los procesadores tenemos:



ColAsignadas(0) = {0,4}
ColAsignadas(1) = {1,5}
ColAsignadas(2) = {2,6}
ColAsignadas(3) = {3,7}

Paso 1:

Sea *NoFact(t)* el número de columnas no factorizadas en el procesador *Pt*. Buscar en las *NoFact(t)* columnas no factorizadas un candidato a pivote, resulta ser un trabajo muy costoso a medida que *n/p* se hace más grande. Es por ésto que se selecciona un subconjunto del total de columnas no factorizadas, tal que contenga las columnas más dispersas asociadas al procesador P_t, y sea de cardinalidad *ncol* en donde *ncol* es un parámetro de entrada al algoritmo, que debe ser mayor o igual que 1. Llamaremos *ColumnasCand(t)* a este conjunto, que por contener las columnas más dispersas es más probable encontrar en ellas un conjunto compatible más grande que si tomamos *ncol* columnas arbitrarias. Además, suponemos que el tomar como columnas candidatas a todas las que faltan por factorizar en P_t, no aumentará proporcionalmante la cardinalidad del conjunto compatible hallado, según el trabajo de *Zlatev* [ZLA-80] en su algoritmo secuencial. También *Van Der Stappen, Bisseling* y *Van De Vorst* [VAN-93] obtuvieron conjuntos compatibles de tamaño razonable, partiendo de muy pocas columnas como candidatas.

Para la matriz considerada, tomemos *ncol* = 2 y obtenemos en cada procesador las siguientes columnas candidatas:

ColumnasCand(0) = {0,4}
ColumnasCand(1) = {1,5}
ColumnasCand(2) = {2,6}
ColumnasCand(3) = {3,7}

Paso 2:

El criterio de estabilidad escogido es el (1.9). Dentro de todos los elementos que cumplan este criterio en una columna, se tomará el de mínimo número de Markowitz, para reducir el riesgo de que la matriz reducida se vuelva densa en pocos pasos de factorización. Debido a que las columnas de *ColumnasCand(t)* son las más dispersas, la búsqueda resultará menos costosa que buscar en columnas arbitrarias entre las no factorizadas. Cada candidato será visto como una tripleta (i,j,w_{ij}), en donde i,j representan los índices por fila y por columna del candidato, y w_{ij} su número de Markowitz. Después se ordena ascendentemente el conjunto de candidatos a pivotes *CandPiv(t)* por dos campos: número de Markowitz e índice por columna, es decir, los candidatos con igual número de Markowitz quedan ordenados ascendentemente según índices por columna. Posteriormente, se eliminan todos aquellos $(i'j',w_{i'j'})$ con $w_{i'j'}$ mayor que $a \times mincount(t)$, en donde *mincount(t)* es el menor número de Markowitz entre los candidatos del procesador P_t, y a es un parámetro de entrada.

CandPiv(t) = $\varnothing$
Para todo j en ColCandidatas(t) hacer
    Obtener (i,j,w_{ij}) *asociado a* $a_{i,j}$ *tal que*
    $a_{i,j} \in \{a_{i,r} \ / \ |a_{i,r}| \geq u \times Max\{|a_{i,j}| \ / \ k \leq i < n\}$ *con menor número de Markowitz*
    $CandPiv(t) = CandPiv(t) \cup (i,j,w_{ij})$
fPara
Ordenar CandPiv(t) por número de Markowitz
$mincount(t) = Min\{w_{ij} / (i,j,w_{ij}) \in CandPiv(t)\}$
Eliminar los (i,j,w_{ij}) *de CandPiv(t) tal que* $w_{ij} > a \times mincount(t)$

Si en una columna, dos o más elementos satisfacen el criterio de estabilidad con igual número de Markowitz, se toma aquél de mayor valor en módulo, pero si además tienen igual módulo, se toma el de menor índice por fila.

En la matriz de ejemplo, se asume que todos los elementos cumplen el criterio de estabilidad, por lo que el conjunto *CandPiv(t)* queda en principio como:

CandPiv(0) = {(1,0,0), (3,4,1)} mincount(0) = 0
CandPiv(1) = {(3,1,1), (5,5,2)} mincount(1) = 1
CandPiv(2) = {(7,2,1), (4,6,2)} mincount(2) = 1
CandPiv(3) = {(6,7,1), (2,3,2)} mincount(3) = 1

Nótese que en la columna 2, dos elementos tienen igual número de Markowitz, pero se toma $a_{7,2}$, asumiendo que su valor es mayor en módulo que $a_{6,2}$. Caso similar ocurre en las columnas 5 y 6.

Suponiendo *a*=2, los conjuntos *CandPiv(t)* quedan como sigue:

CandPiv(0) = {(1,0,0)}
CandPiv(1) = {(3,1,1), (5,5,2)}
CandPiv(2) = {(7,2,1), (4,6,2)}
CandPiv(3) = {(6,7,1), (2,3,2)}

Paso 3:

Entre los elementos $(i,j,w_{ij}) \in CandPiv(t)$, se obtiene un conjunto compatible, dándole prioridad a los de menor número de Markowitz. Para realizar ésto se empieza con: *Comp(t)* = ∅; luego en el mismo orden de los elementos en *CandPiv(t)* se añaden elementos que sean compatibles con *Comp(t)*. Por la manera en que se construye *Comp(t)*, se asegura que está ordenado por número de Markowitz. Siguiendo el ejemplo, se obtiene en cada procesador:

Comp(0) = {(1,0,0)}
Comp(1) = {(3,1,1), (5,5,2)}
Comp(2) = {(7,2,1), (4,6,2)}
Comp(3) = {(6,7,1)}

Nótese que P_3 eliminó el elemento (2,3,2) de *CandPiv(3)* por ser incompatible con (6,7,1) y por ser sucesor del mismo en dicho conjunto (tiene un número de Markowitz mayor).

```
Comp(t) = ∅
Para cada (i,j,wij) ∈ CandPiv(t) hacer
        si (i,j,wij) es compatible con Comp(t) entonces
                Comp(t) = Comp(t) ∪ (i,j,wij)
        fsi
fPara
```

Paso 4:

Mediante un algoritmo de fan-in serán unidos todos los conjuntos *Comp(t)*, con $0 \leq t < p$, obteniendo *Conj* también ordenado por número de Markowitz. Cabe destacar que en cada etapa del *fan-in*, un procesador activo deberá realizar la unión entre dos conjuntos ordenados por número de Markowitz e índice por columna, y para la obtención de *Conj* igualmente ordenado, se utiliza el conocido algoritmo de *mezcla ordenada de dos conjuntos*.

```
{ Algoritmo de fan-in y BroadCast}
j = Identificador del procesador;
i = 1;
CC2 = Comp(t)
Mientras (i < p) Hacer
        si (j mod (2*i) = 0) entonces
                Recibir CC1 de Pj+i
                CC2 = Unión ordenada de CC1 y CC2
        sino
                Enviar CC2 a Pj-i
                Salir del mientras
        fsi
        i = i*2;
fMientras
si (j=0) entonces
        Conj = CC2
        BroadCast de Conj
fsi
```

El procesador raíz del fan-in (P_0) hará finalmente un *broadcast* del conjunto unión *CC2* (el cual será llamado ahora *Conj*) para que sea conocido en todo

procesador de la red. Si la cardinalidad de *Conj* es 1, se obviarán los pasos 5 y 6.

Para el ejemplo que se está usando, el conjunto *Conj* obtenido en este paso es: *Conj = {(1,0,0), (3,1,1), (7,2,1), (6,7,1), (5,5,2), (4,6,2) }*

Paso 5:

La idea es construir un conjunto llamado *Prod(t)* a partir de *Conj*, de tal manera que para cada $(i,j,w_{ij}) \in Prod(t) \cap Comp(t)$ y cada $(i',j',wi'j') \in Prod(t) \cap (Conj - Comp(t))$ se satisfaga que $a_{i',j} = 0$. Para lograrlo, se parte de $Prod(t) = \varnothing$; y en cada etapa del algoritmo se incluye en *Prod(t)* un pivote $(i,j,w_{ij}) \in Conj$, si no es rechazado por el siguiente test:

1) $(i,,j,w_{ij}) \in Comp(t)$: En este caso, de existir $(i',j',wi'j') \in Prod(t) \cap (Conj - Comp(t))$ tal que $a_{i',j} \neq 0$, se descarta (i,j,w_{ij}), ya que por tratar de incluirse luego de $(i',j'w_{i'j'})$ tiene un número de Markowitz mayor o igual a $w_{i'j}$.
2) $(i,,j,w_{ij}) \in (Conj - Comp(t))$: Si existe $(i',j',wi'j') \in Prod(t) \cap Comp(t)$ tal que $a_{ij} \neq 0$, se descarta (i,j,w_{ij}), por no ser compatible con otro pivote de menor o igual número de Markowitz ya aceptado. De esta manera, tienen prioridad los pivotes de menor número de Markowitz, lo cual es importante para disminuir el relleno.

```
Prod(t) = ∅
Para todo (i,j,w_ij) ∈Conj Hacer
    si (i,j,w_ij) ∈ Compt(t) entonces
        si no existe (i',j',w_i'j') ∈ Prod(t) ∩ (Conj - Comp(t)) tal que
        a_i',j ≠ 0 entonces
                Prod(t) = Prod(t) ∪ (i,j,w_ij)
        fsi
    sino
        si no existe (i',j',w_i'j') ∈ Prod(t) ∩ Comp(t) tal que
        a_i,j' ≠ 0 entonces
                Prod(t) = Prod(t) ∪ (i,j,w_ij)
        fsi
    fsi
fPara
```

Nótese que para el ejemplo, el procesador P_0 elimina *(6,6,1)* ya que $a_{6,4} \neq 0$ indicando que $a_{6,4}$ en incompatible con $a_{2,4}$ que tiene menor número de Markowitz.

Prod(0) = {(1,0,0), (3,1,1), (7,2,1), (6,7,1) }
Prod(1) = {(1,0,0), (3,1,1), (7,2,1), (6,7,1), (4,6,2) }
Prod(2) = {(1,0,0), (3,1,1), (7,2,1), (5,5,2), (4,6,2) }
Prod(3) = {(1,0,0), (3,1,1), (7,2,1), (6,7,1), (5,5,2), (4,6,2) }

Paso 6:

Realizando un *fan-in* entre los procesadores, se hallará la intersección de los conjuntos *Prod(t)* $0 \leq t < p$. Dicha intersección nos da como resultado un conjunto compatible en el procesador final del *fan-in* (P_0), que deberá realizar un *broadcast* de dicho conjunto para que todos los procesadores lo conozcan.

```
j = Identificador del procesador;
i = 1;
CC2 = Prod(t)
Mientras (i < p) Hacer
        si (j mod (2*i) = 0) entonces
                Recibir CC1 de Pj+i
                CC2 = Interseción ordenada de CC1 y CC2
        sino
                Enviar CC2 a Pj-i
                Salir del mientras
        fsi
        i = i*2
fMientras
si (j=0) entonces
        CC = CC2
        Broadcast de CC
fsi
```

Para demstrar que el conjunto intersección es compatible se tomarán dos elementos cualesquiera del conjunto intersección, (i,j,w_{ij}) y $(i',j',w_{i'j'})$, presentándose dos casos:

1) ($j \underline{mod}\ p = j' \underline{mod}\ p$): En este caso ambos elementos son compatibles por construcción, ya que provienen del mismo procesador $P_{j \underline{mod} p}$, y éste determinó en el paso 3 que efectivamente son compatibles.

2) ($j \underline{mod}\ p \neq j' \underline{mod}\ p$): Son compatibles, ya que:
$(i,j,w_{ij}) \in CC \wedge (i',j',w_{i'j'}) \in CC \Rightarrow$
$(i,j,w_{ij}) \in Prod(j \underline{mod}\ p) \wedge (i',j',w_{i'j'}) \in Prod(j \underline{mod}\ p) \Rightarrow$
$a_{i'j} = 0 \wedge a_{ij'} = 0 \Rightarrow$ Son compatibles por [2.3]

Para el ejemplo, el conjunto compatible resultante viene dado por:
$CC = \{(1,0,0), (3,1,1), (7,2,1)\}$ que no necesariamente es el más grande, pero preserva la estabilidad en la matriz reducida, y mantiene L y U dispersas.

Cabe destacar que cuando la matriz reducida se haga densa es poco probable que se construyan conjuntos compatibles de cardinalidad razonable, por lo que se usará otra modalidad en el algoritmo que consiste en realizar pivoteo parcial en la columna asociada al paso de factorización *k*. Sin embargo, como no existe un criterio absoluto para considerar que una matriz es dispersa o densa según su número de elementos, se considerará una matriz reducida densa cuando al menos la mitad de sus elementos sean no nulos; así, se cambiará de modalidad en el algoritmo de conjuntos compatibles a partir del momento en que se cumpla:

$$\frac{m^2}{d} \leq Nz',\ d \in [1,2]$$

donde *m* es orden de la matriz no factorizada, y *Nz'* el número de elementos no nulos de la matriz no factorizada. De esta forma se evita el sobrecargo de procesamiento asociado al algoritmo de conjuntos compatibles en pasos posteriores, en los que es poco factible encontrar conjuntos compatibles de una cardinalidad que justifiquen el uso de este algoritmo.

3.) Calcular los $l_{i,j}$

En esta parte del algoritmo, cada procesador hallará los multiplicadores asociados a los pivotes del conjunto compatible que tiene asociado.

```
Para w = 0 Hasta LongCC-1 Hacer
        j = CC[w].j
        Si (j mod p = Id_Proc) entonces
                Obtener li,j con i>j
        fSi
fPara
```

4.) Actualización de la matriz remanente en la factorización

La actualización será de rango m. La manera de llevarse a cabo, se basa en que cada procesador deberá actualizar sus columnas no factorizadas con las m columnas del conjunto compatible, y para éllo, el procesador que tenga asignadas una o más columnas del conjunto compatible deberá darlas a conocer a los otros procesadores.

El algoritmo para actualizar la matriz reducida se puede expresar como sigue:

```
{ Precondición: k = Paso de factorización, y CC = Conjunto compatible}
m = |CC|;
Para w=k hasta k+m-1 hacer
        si ColumnaAsignada(w) entonces
                Enviar l_{i,w} con w< i <n
        sino
                Recibir l_{i,w} con w< i <n
        fsi
        Para cada columna j : (k+m ≤j<n) y (j mod p=t) hacer
                Actualización de a_{i,w} con los l_{i,w}, k+m ≤i < n.
        FPara
FPara
```

5.) Actualizar Vectores de Permutación

La actualización de los vectores de permutación debe ser tal que:

1) π_{k+i} y ρ_{k+j} contengan los índices originales por fila y columna respectivamente de los elementos del conjunto compatible, con $0 \leq i,j < m$.

2) Los índices de las filas y columnas ya factorizadas se mantengan en las posiciones π_i y ρ_j, con $0 \leq i,j < k$.

3) Obligar a que los índices de las filas y las columnas que faltan por factorizar ocupen las posiciones π_{k+m+i} y ρ_{k+m+j} respectivamante, con $0 \leq i, j < n - k - m$.

6.) Actualizar ηFil y ηCol

Para actualizar ηCol basta con que cada procesador determine cuántos elementos no cero hay por cada columna no factorizada, entre sus asignadas.

Se retomará la notación $a_{i,j}^{(k)}$ utilizada en el capítulo 1 para indicar el paso de factorización, además de la posición del elemento en cuestión. Supongamos que la columna *j* pertenece a las no factorizadas en el paso *k+m*. Nos interesa hallar el número de elementos no nulos entre los elementos $a_{k+m,j}$, ⋯ , $a_{n-1,j}$ ya que es la porción no factorizada de la columna j-ésima. Llamemos a esa porción la columna *j* en *A(k+m)*.

Columna j

$$\begin{pmatrix} a_{0,j}^{(0)} \\ \vdots \\ a_{k,j}^{(k)} \\ \vdots \\ a_{k+m-1,j}^{(k)} \\ \text{-----} \\ a_{k+m,j}^{(k)} \\ \vdots \\ a_{n-1,j}^{(k+m)} \end{pmatrix}$$

Conociendo que ηCol_i^k almacenaba el número de elementos no cero de la columna j en A^k, ηCol_j^{k+m} puede obtenerse como:

$$\eta Col_j^{k+m} = \eta Col_i^k + RellenosC_j^k - NoCerosC_j(k, k+m)$$

donde $NoCerosC_j(k, k+m)$ el es número de elementos no cero de la columna j en las posiciones $a_{i,j}^{(k)}$, k≤i<k+m, y $RellenosC_j^k$ el número de rellenos que ocurren

al factorizar m columnas de $A^{(k)}$. Toda la información requerida para esta actualización es local al procesador que tiene asignado dicha columna.

ηFil tiene asociada una fórmula de actualización similar a la de la actualización de ηCol:

$$\eta Fil_i^{k+m} = \eta Fil_i^k + RellenosF_i^k - NoCerosF_i(k,\ k+m)$$

Sin embargo, llevar a cabo esta actualización es más elaborado, ya que la información de cada fila de la matriz está distribuída entre los p procesadores en general.

Sea $Rell\_F_{(i,t)}^k$ el número de rellenos asociados a la fila i, y columna j tal que j *mod* p = t, en $A^{(k)}$. Sea además, $NoCerF_i(k,k+m,t)$ el número de elementos no ceros de $a_{i,j}^{(k)}$ con k≤i<k+m y j *mod* p=t. La idea es que cada procesador t consiga $Reli\_F_{(i,t)}^k$ y $NoCerF_i(k,k+m,t)$ para cada fila i que falta por factorizar, para que con un *fan-in* se calcule:

a) $RellenosF_i^k = \sum_{t=0}^{p-1} Rell\_F_{(i,t)}^k,\ k+m \le i < n$

b) $NoCerosF_i(k,\ k+m) = \sum_{t=0}^{p-1} No\_CerF_i(k,k+m,t)$, k+m≤i<n

De esta manera, el procesador final del *fan-in* podrá hallar $Nu\_Fil_i^{k+m}$ para luego darlo a conocer por los procesadores con un *broadcast*.

Capítulo 4

Implantación y Análisis de Resultados

"Aprender es recordar lo que ya sabes,
actuar es demostrar que lo sabes"

Richard Bach

En este capítulo presentaremos las máquinas paralelas empleadas para la implantación y prueba del algoritmo recién descrito; así como los tiempos de ejecución del programa paralelo con 1, 2, 4, 8 y 16 procesadores, y los comparamos con el tiempo del programa comercial Y12M implantado en el mismo lenguaje de programación (C), ejecutándose en un procesador del mismo tipo, con las mismas matrices de prueba, es decir, bajo las mismas condiciones. Es importante destacar que Y12M es uno de los mejores programas secuenciales conocidos, y ha sido ampliamente utilizado por diversos investigadores en esta área.

1.- MÁQUINAS DE DESARROLLO Y PRUEBA

El algoritmo que describimos en el capítulo anterior fue implementado en una arquitectura MIMD basada en transputers, la *PARSYTEC MC-3/14 DE*, existente en el Laboratorio de Computación Paralela y Distribuida (*LCPD*) de la Universidad Central de Venezuela (UCV), la cual dispone de 8 procesadores del tipo T805, y constituyó nuestra máquina de desarrollo y prueba. Adicionalmente, se realizaron pruebas del programa hasta 16 procesadores en una máquina de características similares, ubicada en el

Centro de Estadística y Software Matemático (*CESMa*) de la Universidad Simón Bolívar (USB).

En la Figura 4.1 puede apreciarse un esquema de la *PARSYTEC MC-3/14 DE* existente en la UCV, y su conexión al *"host"*. Esta posee actualmente 4 módulos de transputers del tipo *MTM-2-12*, cada uno de los cuales tiene a su vez dos transputers del tipo T805, con 4 MB de memoria RAM cada uno, arrojando un total de 8 procesadores disponibles. Tiene como computador anfitrión a una estación de trabajo *Sun sparc station/2*, la cual tiene acceso a la red de procesadores a través de dos transputers distintos, pudiendo trabajar con ella incluso dos usuarios simultáneamente, cada uno en una partición de cuatro procesadores.

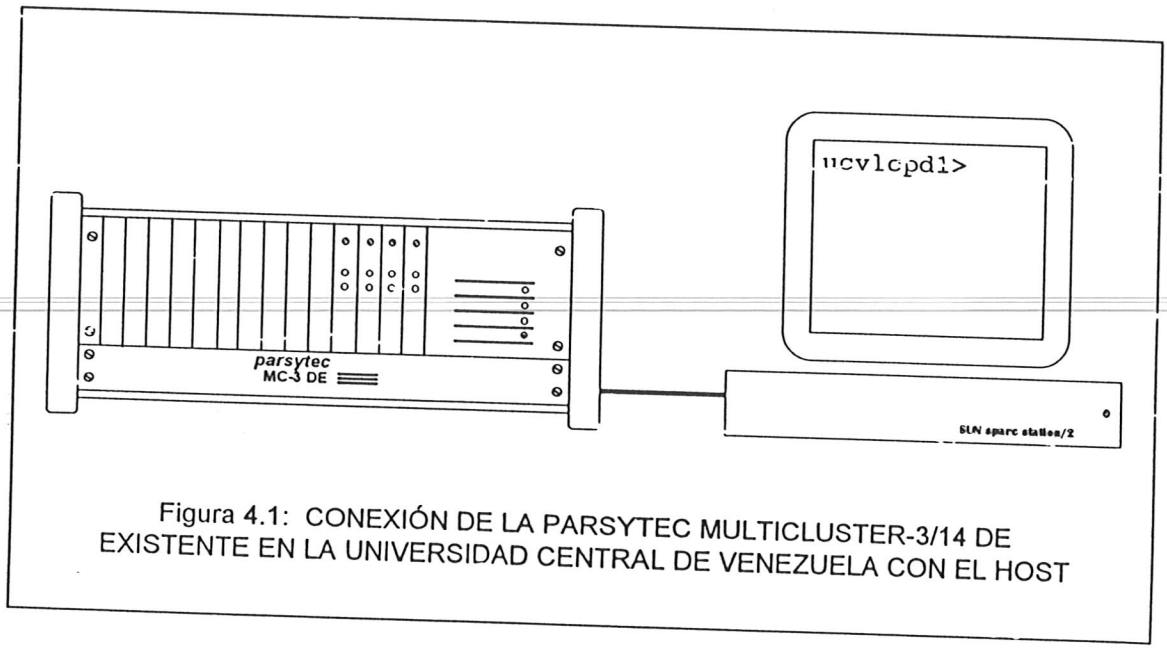


Figura 4.1: CONEXIÓN DE LA PARSYTEC MULTICLUSTER-3/14 DE EXISTENTE EN LA UNIVERSIDAD CENTRAL DE VENEZUELA CON EL HOST

En la USB, en cambio, se cuenta con 7 módulos de transputers del tipo *MTM-2-12*, y un módulo *MTM-2-13*, el cual sólo difiere del anterior en que sus transputers poseen 8 MB de memoria, teniendo un total de 16 procesadores.

El lenguaje de programación empleado fue C, y para mayores detalles, tales como justificación del lenguaje y algunas rutinas de interés, puede consultarse el apéndice.

2.- MATRICES DE PRUEBA

Las matrices de prueba fueron seleccionadas de la colección de matrices dispersas del paquete de Harwell-Boeing, cuya información asociada puede obtenerse en [DUF-92]. En este trabajo se presentan una serie de matrices agrupadas por familias, según las fuentes de origen y/o características de las mismas.

En la Tabla 4.1 veremos el tamaño y el número de elementos no nulos (después de eliminar las entradas almacenadas con valor cero) de cada una de las matrices seleccionadas, así como una breve descripción de las mismas, como la disciplina en que surgieron, o en qué forma se añadieron a la colección.

Tabla 4.1: MATRICES DISPERSAS DE PRUEBA

Nombre de la Matriz	orden (N)	No ceros (Nz)	Descripción
ARC130	130	1115	Matriz Jacobiana asociada a un sist. de ecuac. diferenc. ord.
GRE 343	343	1310	Resultado de modelación en sistemas de computación
HOR 131	434	4338	Sistemas de ecuaciones en flujo en redes
NNC 666	666	4032	Modelación de un reactor nuclear (National Nuclear Corp.)
ORSIRR 2	886	5970	Simulación de yacimientos
PDE 9511	961	4681	Matriz no simétrica suministrada por Tom Hansenffel (1984)
JPWH 991	991	6027	Simulación de circuitos eléctricos (genera muchos rellenos)
SHERMAN1	1000	3750	Matriz simétrica proveniente de ecuaciones diferenciales
GRE 1107	1107	5664	Resultado de modelación en sistemas de computación
PORES 2	1224	9613	Matriz no simétrica con patrón simétrico
ORSREG 1	2205	14133	Simulación de yacimientos
SAYLR 4	3564	22316	Matriz heterogénea proveniente de simulac. de yacimientos

Para cada una de las matrices, se hicieron varias ejecuciones del programa paralelo para la medición de tiempos con 1, 2, 4, 8 y 16 procesadores, variando los parámetros *d* (en 1, 1.5, 2), *a* (entre 2 y 5) y *ncol* (en 4,7,10,15,20,25,30,35,40), y encontramos que para cada matriz, los tiempos variaban significativamente según el número de procesadores y la combinación de los tres parámetros.

3.- VALORES DE LOS PARÁMETROS DE ENTRADA

Para encontrar una combinación adecuada para los tres parámetros, hicimos un análisis de lo que ocurría al fijar dos parámetros con un valor y variar el otro, del cual obtuvimos las siguientes conclusiones sobre los valores de cada uno de estos parámetros:

d:

- Al variar el parámetro *d*, observamos que para el valor 1, en general se cambia de modalidad cuando la matriz reducida es muy pequeña, ya que se está exigiendo que todos los elementos de dicha matriz sean no nulos. Esto hace que el tiempo total de factorización aumente, porque ha habido un sobrecargo de procesamiento al hallar conjuntos compatibles muy pequeños en una matriz que ya puede considerarse densa, aunque tenga algunos elementos cero.

- Para *d*=2, se trata a la matriz remanente como densa apenas cuando poco más de la mitad de sus elementos son no nulos, dando lugar a que se pase a pivoteo parcial cuando aún podían obtenerse conjuntos compatibles de tamaño razonable, por lo que aumenta el tiempo total debido a que se realiza una mayor cantidad de pasos de factorización, y a un incremento en el relleno total, ya que durante el pivoteo parcial no hay control de rellenos.

- Estos resultados nos llevaron a la conclusión de que es necesario un valor intermedio entre 1 y 2, como lo es *d*=1,5, valor para el cual se obtienen mejores tiempos.

a:

- Al variar el parámetro *a*, observamos que para valores muy pequeños (tal como 2), se está siendo muy estricto en el control de rellenos, debido a que después de haber buscado pivotes en las columnas candidatas en cada procesador, se rechaza gran parte de los mismos (incluso hasta más de la mitad de los candidatos), lo cual hace que se sacrifique mucho tiempo en la búsqueda, aún cuando la matriz reducida permanezca dispersa durante más pasos y por ello se pueda ahorrar tiempo de cálculo.

- Para valores muy grandes, en cambio, al aceptar la mayoría de los candidatos a pivotes, se está siendo muy permisivo en cuanto al control de rellenos, pudiendo tener como resultado una matriz reducida ya densa al cabo de pocos pasos de factorización.

- Según nuestras pruebas para hallar el valor intermedio apropiado, éste puede ser 3 o 4. Sin embargo, usaremos 4 debido a que ese fue el valor que usaron Van De Vorst, Bisseling y Var Der Stappen, en su publicación.

ncol:

- Para valores pequeños de *ncol* como 4 y 7, se hace una búsqueda en menos columnas, obteniéndose necesariamente conjuntos compatibles muy pequeños, dando como resultado una factorización en muchos pasos, en la que el tiempo de comunicación pesa notablemente sobre el tiempo total

- Para valores grandes de *ncol*, como 30, 35, y 40, se hace una búsqueda de candidatos muy pesada en cada paso, siendo muy probable que se descarten los pivotes pertenecientes a las columnas más densas, y que entonces se haya "perdido" un valioso tiempo de búsqueda en estos candidatos, lo cual degrada el tiempo total de factorización, por lo que se obtuvo mejores resultados con *ncol*=15.

4.- RESULTADOS EXPERIMENTALES

A continuación presentaremos en la Tabla 4.2 la cantidad de rellenos que ocurren durante la factorización, tanto para el programa secuencial Y12M, como para el programa paralelo con 1, 2, 4, 8 y 16 procesadores, dados los parámetros de entrada ya escogidos (d=1,5; a=4 y $ncol$=15).

Tabla 4.2: RELLENOS OCURRIDOS DESPUÉS DE LA FACTORIZACIÓN

Matriz	Y12M	p = 1	p = 2	p = 4	p = 8	p = 16
ARC130	1137	79	50	37	37	37
GRE 343	6780	6263	5923	5837	5828	5523
HOR 131	21565	29928	32861	25801	32235	27130
NNC 666	21348	9108	9592	9926	9677	10635
ORSIRR 2	40782	48609	46688	52363	54648	56851
PDE 9511	22077	19544	20832	21071	22105	22095
JPWH 991	54508	54964	59605	55520	54025	54547
SHERMAN1	17713	19052	19007	21071	19860	22043
GRE 1107	47570	44920	48444	47014	45713	50000
PORES 2	62145	30384	54983	49317	40314	37193
ORSREG 1	48662	198198	225968	264151	268851	291472
SAYLR 4	223779	299894	426161	431182	377857	361596

Podemos observar que en cuanto a la generación de rellenos, el programa paralelo es mejor que Y12M para ciertas matrices, pero para otras no. Por ejemplo, para matrices pequeñas, como ARC130 y GRE 343, así como para algunas matrices grandes, como PORES 2 el programa paralelo genera menos rellenos, pero en cambio para otras matrices grandes como SAYLR 4 y ORSREG 1, es Y12M quien genera menos cantidad de rellenos. De hecho, para la matriz SAYLR 4 no se culminó la ejecución con p=1 para transputers pertenecientes a módulos MTM-2-12, ya que la cantidad de rellenos que genera su factorización, requiere más de los 4 MB que éstos poseen, haciéndose necesario realizar la ejecución en un transputer de un módulo MTM-2-13, en los que se cuenta con el doble de la memoria.

Para la mayoría de las matrices, la cantidad de rellenos generados por ambos programas es comparable. Esto era de esperarse, ya que además de que ambos poseen un criterio de control de rellenos, a ambos le colocamos el valor de 0,25 al parámetro de estabilidad u, y por ello se están discriminando de igual forma los pivotes inestables.

Uno de los objetivos de este trabajo era mejorar los tiempos de factorización a medida que se incrementa el número de procesadores, manteniendo un control de rellenos aceptable y, tal como se aprecia en las tablas 4.2 y 4.3, éste objetivo fue logrado.

Tabla 4.3: TIEMPOS DE FACTORIZACIÓN (en seg.)

Matriz	Y12M	p = 1	p = 2	p = 4	p = 8	p = 16
ARC130	0,50	0,62	0,55	0,49	0,55	0,69
GRE 343	5,05	5,22	3,07	2,33	2,18	2,41
HOR 131	28,46	35,10	23,16	10,67	10,10	7,16
NNC 666	15,86	17,83	10,32	7,18	5,45	5,58
ORSIRR 2	42,63	68,28	36,01	26,75	18,69	15,26
PDE 9511	15,95	31,30	19,44	13,71	10,88	10,53
JPWH 991	110,51	91,04	54,05	28,82	18,88	14,48
SHERMAN1	15,58	33,51	20,03	13,57	8,71	8,32
GRE 1107	70,20	78,69	47,16	30,47	20,22	17,27
PORES 2	67,42	60,59	65,15	37,06	23,32	19,75
ORSREG 1	300,23	471,79	290,71	233,97	170,65	127,18
SAYLR 4	941,92	830,24	760,58	446,65	205,68	147,82

Si comparamos la Tabla 4.2 con la 4.3, podemos observar que para aquellas matrices con gran cantidad de rellenos, también se tienen mayores tiempos de factorización, debido a que se tienen que realizar más cáiculos. Veremos que, a medida que se incrementa el número de procesadores, hay una disminución mayor del tiempo, que para aquellas matrices que tienen menos rellenos, tal como las matrices JPWH 991 y SAYLR 4, respecto de

GRE 343, por ejemplo. Esto es debido a que el tiempo de cálculo pesa mucho más sobre el tiempo de comunicación que para las matrices con menos rellenos.

De igual forma, podemos observar que las matrices "pequeñas", como ARC 130 y GRE 343, no tienen un buen rendimiento a medida que que se incrementa el número de procesadores, lo cual tiene sentido, debido a que presentan poco cálculo, y el tiempo de comunicación pesa mucho más en el tiempo total de factorización, por lo que no se justifica la paralelización de la factorización para estos casos. Ahora bien, la disminución del tiempo no es proporcional al número de procesadores, puesto que a pesar de que idealmente se divide a la mitad el tiempo de cálculo duplicando el número de procesadores, el costo de comunicación se multiplica por un factor dado por la arquitectura. Para explicar mejor la observación anterior, podemos observar la Tabla 4.4, en la que se muestra el número de pasos en que se realiza la factorización para cada una de ellas

Tabla 4.4: NÚMERO DE PASOS EN QUE SE LLEVA A CABO LA FACTORIZACIÓN

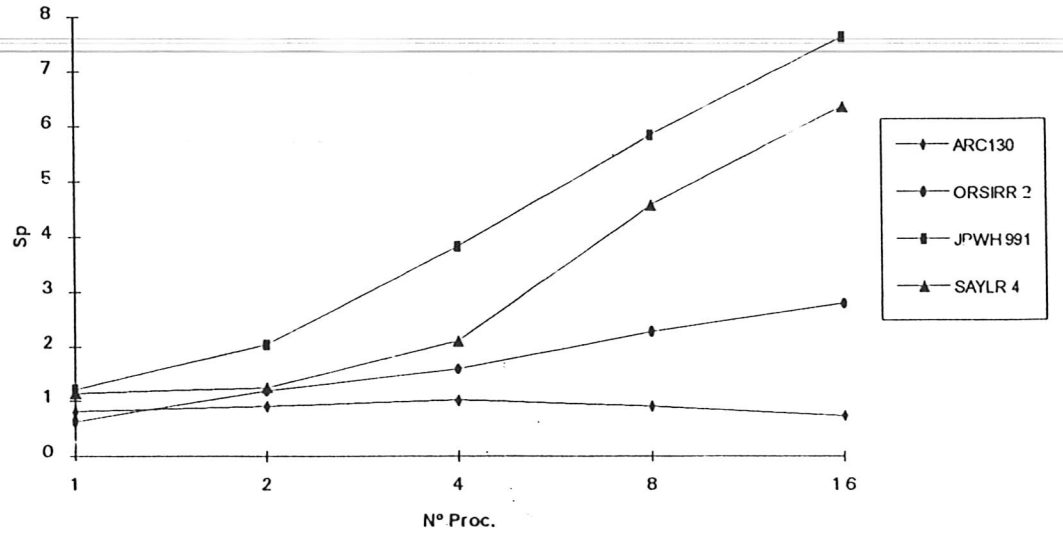
Matriz	Y12M	p = 1	p = 2	p = 4	p = 8	p = 16
ARC130	129	25	22	20	19	18
GRE 343	342	90	78	74	73	71
HOR 131	433	191	191	164	189	170
NNC 666	665	199	140	120	104	100
ORSIRR 2	885	279	230	223	231	236
PDE 9511	960	179	142	129	132	131
JPWH 991	990	288	268	238	233	232
SHERMAN1	999	187	153	150	138	148
GRE 1107	1106	343	283	226	203	218
PORES 2	1223	286	298	252	206	190
ORSREG 1	2204	613	490	533	497	539
SAYLR 4	3563	837	796	654	554	491

Como podemos observar, el número de pasos en que se lleva a cabo la factorización es, en general, poco variable para cada matriz (considerando sólo el algoritmo paralelo), por lo que es lógico preguntarse por qué el tiempo no disminuye en la misma proporción que el número de procesadores; y la respuesta es el costo de comunicación, como ya habíamos comentado. Otro hecho a observar en la Tabla 4.4 es que el número de pasos con el programa paraleloes mucho menor que el número de pasos de Y12M, lo cual es prueba de que el algoritmo que permite obtener los conjuntos compatibles es bueno.

Aceleración

La aceleración *Sp* ("*speed-up*") de un algoritmo paralelo viene dada como el cociente del tiempo de ejecución del mejor algoritmo secuencial conocido entre el tiempo de ejecución del algoritmo con *p* procesadores. En la Gráfica 4.1, podemos observar la aceleración del programa paralelo para cuatro de las matrices seleccionadas.

Gráfica 4.1: ACELERACIÓN



verse, la aceleración máxima se
ara la matriz JPWH 991, y es S16 = 7,631;
promedio con 16 procesadores es 3,305.

En general, puede observarse que la aceleración es crec
ero de procesadores, pero para matrices pequeñas, como A
celeración es baja (menor que 1 en casi todos los casos), ratifi
conclusión a la que ya habíamos llegado de que para dichas matr
referible utilizar un programa secuencial, puesto que no se mejora
tiempos de factorización.

5.- CONCLUSIONES

En este trabajo se diseño e implementó un algoritmo paralelo de descomposición LU de matrices dispersas utilizando un método novedoso para la construcción de conjuntos compatibles, el cual permitió que se efectúen varios pasos de factorización en un solo paso paralelo, concluyendo el proceso total con una ganancia significativa de tiempo respecto de uno de los mejores algoritmos secuenciales. Es relevante destacar las principales características del algoritmo, que permitieron dicha mejora en tiempo:

1.) El uso de permutaciones lógicas evita realizar costosos intercambios de elementos entre procesadores.
2.) El cambio de modalidad del algoritmo a pivoteo parcial puede disminuir el tiempo total de factorización, ya que es preferible evitar el sobrecargo de procesamiento que ocurre cuando se obtienen conjutos compatibles muy pequeños en una matriz reducida posiblemente densa.
3.) La búsqueda de pivotes no se restringe a los elementos de la diagonal, obteniéndose conjuntos compatibles más grandes, sin que esto tenga un costo computacional demasiado alto.
4.) La paralelización de la factorización de matrices dispersas tiene sentido si el orden de dichas matrices en alto, y más aún en matrices que tienden a generar mucho relleno durante el proceso de factorización, ya

tiempo de cálculo pesa más sobre el tiempo total de factorización que el tiempo de comunicación, y por consiguiente, se mejoren los tiempos de procesamiento a medida de que se incremente el número de procesadores.

5.) El algoritmo que permite la obtención de conjuntos compatibles diseñado genera conjuntos de alta cardinalidad, con pivotes estables numéricamente, a un bajo costo computacional, lo que constituyó nuestra ayuda principal para mejorar los tiempos respecto del reconocido algoritmo secuencial Y12M.

6.) Por último, es recomendable pruebar la implementación del mismo algoritmo en una arquitectura hipercúbica, o en su defecto, en una red de transputers con una topología virtual de hipercubo, si la arquitectura y el sistema operativo lo permiten. Es posible que los tiempos mejoren aún más de esta forma, ya que el costo de comunicación disminuiría al utilizarse los enlaces directos entre los procesadores que el algoritmo requiere, y no utilizar el sistema operativo para enrutarlo a través de procesadores intermedios.

Apéndice:

Algunos Detalles de Implantación

"El mundo está hecho de detalles"

Proverbio Antiguo

En este apéndice discutiremos algunos detalles importantes de la implementación del algoritmo que describimos en el capítulo 3, tal como lo es el lenguaje utilizado y algunos procedimientos del mismo.

El algoritmo fue implementado en una arquitectura MIMD basada en transputers, la *PARSYTEC MC-3 DE*, en la cual puede realizarse comunicación síncrona y asíncrona, y como ya habíamos mencionado, elegimos hacerlo de forma asíncrona. En la comunicacion síncrona, se permiten mensajes de cualquier longitud, pero el proceso que envía el mensaje tiene que esperar que el proceso destino lo reciba en su totalidad para continuar con la ejecución de otras instrucciones; mientras que en la asíncrona, los mensajes tienen una longitud límite de 1024 bytes (específicamente para la *PARSYTEC MC-3 DE*), pero el proceso que origina el mensaje no debe esperar por el proceso destino, sino que continúa inmediatamente su ejecución. Es claro que ambos tipos de comunicación tienen ventajas y desventajas. Sin embargo, usamos rutinas asíncronas ya que en la práctica hemos obtenidos mejores tiempos con ellas, aún cuando se tenga que dividir mensajes grandes en pequeños mensajes de a lo sumo 1024 bytes.

Ahora bien, existen dos formas distintas de implementar la comunicación asíncrona, según se definan o no *enlaces virtuales*. En caso de que se definan, los procesadores que van a comunicarse en algún momento dado, establecen la comunicación de forma aún más rápida (a nivel de ejecución) que en el caso de que no sean definidos y tengan que enrutarse de forma aleatoria. Sin embargo, la programación de la comunicación aleatoria es de mayor simplicidad, y hemos elegido hacerla de dicha forma, debido a que no se ha podido realizar aún una implantación exitosa de la comunicación asíncrona basada en enlaces virtuales.

Lenguaje de programación empleado

El lenguaje que usamos para la programación es C. Los motivos que originaron esta decisión fueron los siguientes:

1. Es uno de los dos lenguajes con que contamos actualmente en la UCV para programar bajo el sistema operativo de la *PARSYTEC MC-3 DE*, *PARIX* (extensiones *PAR*alelas de UN*IX*).
2. El código C es portable, lo que nos permite probar rutinas no paralelas en ambientes integrados como Turbo C o C++, para microcomputadores compatibles con IBM, por ejemplo.
3. La experiencia que tenemos de trabajar con el lenguaje, gracias a la cual se disminuyó el tiempo de desarrollo del programa.
4. Cuenta con rutinas de manejo de archivos, memoria, comunicación (en *PARIX*) y otras, que fueron de gran utilidad a la hora de escribir el programa.
5. Es un lenguaje de nivel medio que suele producir programas tan eficaces como los codificados en lenguaje ensamblador, manteniendo las ventajas de un lenguaje de alto nivel.
6. Permite manejar cómodo y eficientemente apuntadores y estructuras dinámicas que es en lo que se basan nuestras estructuras, a diferencia de la otra opción que teníamos, el lenguaje FORTRAN.

Procedimientos de C utilizados en nuestra implantación:

Los procedimientos que más usamos en nuestro programa se dividen en tres clases:

1) Rutina de E/S: Los procedimientos de E/S usados en nuestro programa son los que permiten abrir o leer un archivo con formato (*fopen* y *fread* respectivamente).

2) Rutinas para manejo de memoria: El uso de rutinas para manejo de memoria permite hacer un uso eficiente de la misma. Estas rutinas son:

- ***memcpy***: Permite copiar un bloque de memoria de n bytes desde una dirección origen hasta una dirección destino. Este procedimiento es muy usado cuando necesitamos incluir un relleno en una columna cuyos elementos deben estar ordenados ascendentemente por índices de fila. Para ello, una vez ubicada la posición donde debe insertarse el relleno, se desplazan en una posición los elementos con índice mayor, para poder colocar el relleno y mantener la condición.

- ***memset***: Copia un valor (byte) repetidas veces a partir de una dirección de memoria. La usamos cuando necesitamos llenar las posiciones de un vector con un valor especificado; por ejemplo al inicializar un arreglo con ceros.

- ***malloc***: Asigna memoria a un apuntador. Su único parámetro es la cantidad de bytes requeridos, y retorna la dirección de memoria donde comienza la cadena que asigna. Lo usamos al inicializar todas las estructuras de datos, ya que no manejamos estructuras estáticas.

- ***free***: Libera la memoria pedida con malloc a partir de un apuntador. Usamos dicha instrucción cuando deseamos liberar espacio de memoria que anteriormente haya sido pedida con *malloc*.

- ***realloc***: Permite relocalizar la cadena de bytes referenciada por un apuntador. Lo usamos cuando queremos aumentar el tamaño de una columna al surgir un relleno que haga exceder su tamaño actual.

3) Rutinas para la comunicación: Las rutinas de comunicación asincrónica utilizadas fueron:

- ***PutMessage***: Permite al procesador que la invoca enviar un mensaje de cierto tipo con un procesador dado, o con todos si se desea. Otros parámetros de la rutina se utilizan para el control por parte del sistema operativo.
- ***GetMessage***: Es la contrapartida del anterior, y permite al procesador que la invoca recibir un mensaje de algún tipo desde cierto procesador, o de cualquiera, si así se le especifica.

Bibliografía

- [ALA-89] G. ALAGHBAND y H. JORDAN
  Sparse Gaussian elimination with controlled fill-in on a shared memory multiprocessor
  8/IEEE Transactions on Computers vol. 381 (1989), pag 210-221

- [ALA-89a]G. ALAGHBAND
  Parallel pivoting comined with parailel reduction and fill-in control
  Parallel Computing 11 (1989), pag 210-221

- [CAL-73] D. A. CALAHAN
  Parallel solution of sparse simultaneous linear equations
  Proceedings of 11th Annual Allerton on Circuits and System Theory, 1973, pp.729-735

- [DAV-90] T. A. DAVIS y P. C. YEW
  A nondeterministic parallel algorithm for sparse unsymmetric LU factorization
  SIAM Journal on Matrix Analysis Appl., 11 (1990), pp 383-402

- [DUF-79] I. S. DUFF y J. K. REID
  Some design features of a sparse matrix code
  ACM Transactions on Mathematical Software, 5 (1979), pp18-35

- [DUF-86] I. S. DUFF, A. M. ERISMAN y J. K. REID
  Direct Methods for Sparse Matrices
  Cap. 9
  Oxford University Press,
  Oxford, U.K., 1981.

- [ECH-92] J. A. ECHEVERRIA
  Factorización LU por Control de Rellenos de una Matriz Dispersa sobre un Sistema Multiprocesador de Memoria Local
  Tesis de Grado en Computación, UCV
  Caracas, 1992

- [GAL-91] K. GALLIVAN, A. SAMEH, y Z. ZLATEV
  Parallel direct method codes for general sparse matrices
  Tech. Rep. 1143, Center of Supercomputing Research and Development, Univ. of Illinois, Urbana, IL, 1991.

- [GEO-81] A. GEORGE y J. LIU.
  Computer Solution of large sparse positive definite systems
  Prentice Hall, N.Y., 1981.

- [HWA-88 HWANG, Kai y Fay. A. Briggs
  Arquitectura de Computadoras y Procesamiento Paralelo
  Mc Graw Hill, 1988

- [MAR-57] H. M. MARKOWITZ
  The elimination form of the inverse and its application to linear programming
  Management Sci., 3 (1957), pp. 255-269

- [PAR-92] PARSYTEC COMPUTER GmbH
  Manuales de PARIX Documentation
  Release 1.1
  Septiembre de 1992

- [RUK-94] M. RUKOZ y R.SURÓS
  Procesamiento Paralelo
  Séptima Escuela Venezolana de Matemáticas,
  Asociación Matemática Venezolana,
  Centro de Estudios Avanzados - IVIC
  Caracas, 1994

- [SCH-89] U. SCHENDEL
  Sparse Matrices: Numerical Aspects with Applications for Scientists and Engineers
  Ellis Horwood Limited.
  England, 1989.

- [SMA-88] D. SMART y J. WHITE
  Reducing the parallel solution time of sparse circuit matrices using reordered Gaussian elimination and relaxation
  Proc. IEEE international Symposium of Circuits and Systems
  1988, pp. 627-630

- [TER-87] TE RIELE, DEKKER, VAN DER VORST
  Algorithms and Applications on Vector and Parallel Computers
  Serie Special Topics in Supercomputing
  Número 3
  Elsevier Science Publishers B.V.,1987

- [THO-92] THOMPSON LEIGHTON, F
  Introduction to Parallel Algorithms and Architectures
  Arrays - Trees - Hypercubes
  Morgan Kauffman Publishers, 1992.

- [ZLA-80] Z. ZLATEV
  On some pivotal strategies in Gaussian elimination by sparse technique.
  SIAM Journal on Numerical Analysis, 17 (1980), pp. 18-30