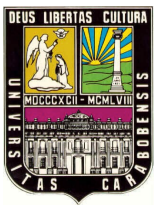


UNIVERSIDAD DE CARABOBO
ÁREA DE ESTUDIOS DE POSTGRADO
FACULTAD DE INGENIERÍA
DOCTORADO EN INGENIERÍA

Soporte de memoria out-of-core para biblioteca de solución de sistemas dispersos

Autor: Jorge Arturo Castellanos Díaz

Tutor: Dr. Germán Larrazábal

bárbula, abril de 2012

.

VEREDICTO

Nosotros, miembros del jurado designado por el Consejo de Postgrado de la Facultad de Ingeniería, de acuerdo a lo previsto en el artículo 146 del citado Reglamento, para estudiar la tesis doctoral titulada:

Soporte de memoria out-of-core para biblioteca de solución de sistemas dispersos

Presentada como requisito parcial para obtener el grado de
Doctor en Ingeniería por el aspirante

Jorge Arturo Castellanos Díaz
C.I.: V-9.212.276

Hemos verificado que las correcciones sugeridas durante la discusión de Tesis Doctoral fueron realizadas por el aspirante.

Bárbula, 20 de abril de 2.012.

Dr. Demetrio Rey
C.I.: V-7.127.552

Dr. Braulio De Abreu
C.I.: V-8.736.737

Dr. Francisco Hidrobo
C.I.: V-10.039.622

Resumen

En esta investigación se crea el soporte para memoria fuera de núcleo (*out-of-core*) para una biblioteca que resuelve sistemas numéricos dispersos (`UCSparseLib`) con el objeto de ampliar su ámbito de aplicación. El soporte *out-of-core* reduce las necesidades de memoria de la biblioteca para cada una de las funciones relacionadas con la manipulación de vectores y/o matrices dispersas y/o densas, con un costo reducido en tiempo de ejecución. El soporte *out-of-core* funciona de forma transparente para el programador, pues su activación/desactivación no influye en la forma de utilización del programador en las funciones que componen la biblioteca. El soporte *out-of-core* se basa en la teoría clásica de cachés y en la utilización del recurso de macros provisto por el lenguaje de programación "C". Las pruebas realizadas para las operaciones básicas (producto matriz-vector, producto matriz-matriz y transpuesta de la matriz), la factorización de matrices mediante métodos directos ($Cholesky$, LU y LDL^t) y la solución de sistemas lineales mediante el método multinivel algebraico mostraron ahorros significativos en el uso de la memoria con el núcleo *out-of-core* activado pagando como precio un bajo *overhead* en tiempo de ejecución. Este nuevo soporte *out-of-core* permitirá a la comunidad científica la solución de sistemas numéricos medianos en computadores modestos y la solución de sistemas numéricos grandes en estaciones de trabajo y supercomputadores disponibles.

Agradecimientos

A Dios, por darme la fortaleza y determinación para culminar esta etapa de mi vida.

A Germán, mi tutor: primero por darme este tema de tesis que permitió desarrollar mis capacidades y segundo por acompañarme en este largo camino hasta el final.

Al FONACIT, por el financiamiento otorgado para llevar a cabo mis estudios doctorales.

A mi hijo Jorge Luis, a mi familia y amigos que siempre me han apoyado.

Índice

Índice de tablas

Índice de figuras

1. Introducción

Este capítulo introduce en el tema objeto de ésta investigación, describe el propósito de la investigación y justifica su importancia.

1.1. El Problema

El software utilizado para la realización de simulaciones en ingeniería cobra cada día más importancia debido a la utilidad creciente que ha demostrado desde sus inicios (a mediados del siglo pasado) cuando se utilizó en el cálculo estructural. Actualmente el software de simulación se ha generalizado a otras áreas de la ingeniería como la mecánica de fluidos, la petroquímica, la meteorología, etc.

Los modelos matemáticos son el insumo principal para la realización de simulaciones, pues la operación de un modelo dinámico permite obtener una secuencia de resultados que podrían ocurrir en el mundo real. La simulación es entonces la representación en la que se pretende hacer algo con un objeto real, cuando realmente se está trabajando con una imitación (modelo de alguna clase).

El proceso de simulación (ver figura 1) se puede descomponer en un esquema de tres pasos: pre-procesador, procesador y post-procesador. En el primer paso se

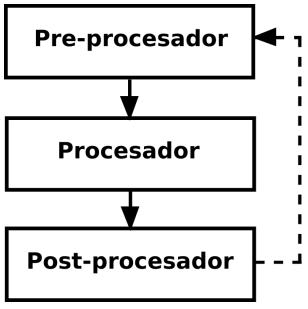


Figura 1: Esquema de un proceso de simulación.

preparan los datos para su posterior procesamiento; generalmente se organizan como

una malla ó como un conjunto de estructuras de datos del tipo matriz y/o vector. En el paso siguiente se realiza el procesamiento de los datos, ya sea a través de bucles en el tiempo ó en el dominio de las frecuencias; en estos bucles se conforma el sistema de ecuaciones a resolver y finalmente se procede a la solución del sistema de ecuaciones. En el último paso se efectúa el post-procesamiento de los datos obtenidos en el paso anterior de forma que puedan ser entendidos cabalmente por el usuario del sistema.

De los tres pasos del proceso de simulación se le presta mayor atención al segundo paso (procesamiento), pues es el que toma más tiempo para su realización y es aquel que consume una mayor cantidad de recursos computacionales del sistema. Dentro del procesamiento la etapa de resolución del sistema se ha estimado que toma del 70 % al 90 % del tiempo total del proceso de simulación, de allí su importancia.

Al software que realiza la solución del sistema de ecuaciones se le denomina núcleo computacional y está constituido por una biblioteca de solución de sistemas de ecuaciones lineales. Este núcleo resuelve el sistema lineal: $Ax = b$, donde, A es una matriz de dimensión $n \times n$, no singular, de gran tamaño, y en la mayoría de los casos dispersa. Como ejemplo tenemos que una malla de $50 \times 50 \times 50$ con 5 grados de libertad por nodo, genera una matriz con 625.000 filas con una densidad de 0,024 % si se discretiza un operador escalar elíptico mediante diferencias finitas.

Dependiendo del tamaño de los datos, del método numérico utilizado en la solución y de las características del sistema computacional, es posible resolver el sistema; y en el caso de ser posible obtener la solución usando la memoria física de la plataforma disponible, se puede obtener la solución en un mayor ó menor tiempo dependiendo de las características de la matriz de entrada y del método utilizado.

Los métodos utilizados son:

- Directos: $O(n^3)$
- Iterativos con precondicionador: $O(n^2)$
- Multinivel algebraico: $O(kn)$

Los métodos directos son más sencillos para implementar y usan menos memoria pero el tiempo de ejecución es el orden de n^3. Los métodos iterativos con precondicionador representan un buen compromiso entre uso de recursos y tiempo de ejecución. Los métodos multinivel algebraico son más complejos para implementar y usan más cantidad de memoria pero permiten obtener mejores tiempos de ejecución para sistemas grandes. Si se desea utilizar métodos iterativos con precondicionador y/o métodos multinivel para resolver sistemas de ecuaciones de gran tamaño es altamente recomendable la utilización de algoritmos con soporte *out-of-core*.

1.2. Objetivos de la investigación

1.2.1. Objetivo general

Crear el soporte de memoria *out-of-core* para la biblioteca numérica `UCSparseLib`.

1.2.2. Objetivos específicos

a Generar el subsistema de memoria caché para el soporte *out-of-core* inspirado en la teoría general de cachés, para manejar eficientemente las transferencias de datos entre la memoria y el disco duro.

b Crear la capa *out-of-core* para operaciones básicas (producto matriz-vector, transpuesta de la matriz) empleando recursos del lenguaje de programación "C" tales como las macros y las banderas de compilación condicional para permitir a la biblioteca `UCSparseLib`(Larrazábal, 2004) funcionar con el soporte activado o desactivado de forma transparente para el programador.

c Construir el soporte *out-of-core* para operaciones complejas (métodos directos, métodos iterativos y multinivel algebraico) aplicando el soporte *out-of-core* creado para operaciones básicas de la biblioteca `UCSparseLib`(Larrazábal, 2004).

d Evaluar el desempeño del soporte *out-of-core*, tanto para operaciones básicas como para operaciones complejas, efectuando pruebas sintéticas con el soporte desactivado y activado, respectivamente; éstas pruebas permitirán por comparación, cuantificar el ahorro de memoria y el costo en tiempo de ejecución cuando el soporte *out-of-core* está activado.

1.3. Justificación

La creación del soporte *out-of-core* para una biblioteca de solución de sistemas numéricos divulgada en la comunidad científica venezolana está en sintonía con el plan Nacional de Ciencia, Tecnología e Innovación (2005-2030) porque contribuye con nuestra soberanía y autonomía tecnológica ya que facilitará el desarrollo de nuevos proyectos en diversos ámbitos como la industria petrolera, petroquímica y metalmecánica.

El soporte *out-of-core* beneficiará directamente a los usuarios actuales de la biblioteca `UCSparseLib` al permitirles resolver problemas con conjuntos de datos más grandes, ya sea mediante el uso de computadoras económicas ó de supercomputadores.

El nuevo soporte *out-of-core* permitirá el planteamiento de futuros trabajos doctorales en la Facultad de Ingeniería de la Universidad de Carabobo ya sea mediante

incorporación de nuevas funcionalidades al soporte recién creado ó a través de la utilización de la biblioteca `UCSparseLib` mejorada con el nuevo soporte en la solución de problemas de aplicación en ingeniería.

2. El soporte *Out-of-core*

En este capítulo se presenta en detalle la capa *out-of-core* que se considera fundamental para la comprensión de los capítulos subsiguientes de esta tesis. Para entrar en materia se presentan generalidades sobre la biblioteca `UCSparseLib` sobre la cual se implementa esta capa, así como también se presentan elementos teóricos del almacenamiento de matrices dispersas, necesarios para la comprensión del funcionamiento de la capa *out-of-core*.

2.1. La biblioteca `UCSparseLib`

Es una biblioteca numérica que posee un conjunto de funcionalidades para la resolución de sistemas lineales dispersos (Larrazábal, 2004). Maneja y almacena las matrices usando formatos dispersos (CSR, CSC) lo cual le permite la solución de sistemas medianos con una alta eficiencia en el uso de recursos de memoria. Fue desarrollada para ser liberada en un futuro como software libre y su código fuente está disponible desde internet.

La biblioteca `UCSparseLib` está conformada por siete módulos, según se describe a continuación:

- Operaciones de E/S: lee y escribe matrices en un formato simple (CSR, CSC).
- Operaciones Matriz-vector: realiza operaciones básicas: vector-vector, matriz-vector, reordenamiento, etc.
- Métodos directos e iterativos: *Cholesky*, LDL^T, LU, Jacobi, Gauss-Seidel, Gradiente Conjugado, GMRES, etc.f
- Precondicionadores: factorizaciones incompletas.
- Multinivel algebraico: AMG (*Algebraic Multigrid*) con diversas etapas de inicialización: agregación, coloreado rojo-negro, fuertemente conexo, etc.
- Otras funciones: temporizadores, manejo de memoria y depuración.

2.2. *Out-of-core*

Según ya se ha mostrado, la solución numérica eficiente de sistemas lineales de ecuaciones y otras operaciones como los cálculos de autovalores (Z. Bai y van der Vorst, 2000) pueden estar limitados cuando el conjunto de matrices asociadas a su cómputo no caben en la memoria física RAM (*Random Access Memory*) del computador. El conjunto de datos que no cabe en la memoria física debe ser almacenado en un dispositivo de almacenamiento secundario como el disco duro.

El acceso a los datos almacenados en disco duro es lento comparado con el acceso a memoria física debido a factores como la latencia y ancho de banda. Para obtener un comportamiento aceptable, el algoritmo que accede a los datos desde el disco duro debe (Toledo, 1999):

- Almacenar en memoria física una fracción de los datos en forma de bloques consecutivos grandes.
- Usar la fracción de los datos almacenada en memoria, tantas veces como sea posible.

  Los algoritmos que han sido diseñados para obtener un comportamiento eficiente cuando la estructura de los datos está almacenada en disco se denominan *algoritmos out-of-core* (Toledo, 1999).

Las aplicaciones *out-of-core* pueden manejar conjuntos de datos muy grandes o de datos con uso infrecuente a través del almacenamiento temporal en memoria de una fracción de los datos del disco (Moor, 2002). Por supuesto que existe un conjunto de aplicaciones denominadas *out-of-core paralelo* cuyos conjuntos de datos exceden la capacidad de la memoria virtual del computador (Caron y Utard, 2004), ejemplos de ellas están en las siguientes áreas de aplicación:

- Cálculo científico: modelado, simulación, etc. (Toledo, 1999).
- Visualización científica (Silva, Chiang, El-Sana, y Lindstrom, 2002).
- Bases de Datos.

El concepto *out-of-core* tiene además otras connotaciones; le permite a los usuarios resolver eficientemente problemas grandes usando computadores económicos.

Actualmente[1] el almacenamiento de datos en disco duro es más económico que en memoria física en una relación aproximada de 1 a 64 veces[2]; por tanto a nuestra

[1] Noviembre de 2011.

[2] Precios de referencia tomados de *www.amazon.com* - memoria RAM 4GB $22,50 / disco duro 1TB $90,00.

comunidad científica limitada en recursos para la adquisición de equipos puede resultarle atractivo este ahorro importante, pues ya se ha demostrado que muchos de los algoritmos *out-of-core* tienen un desempeño similar a los algoritmos *in-core* (que se ejecutan sólo en la memoria física del computador). Se puede decir entonces que un algoritmo *out-of-core* ejecutándose en un computador con memoria limitada nos da una mejor relación costo/desempeño que su contraparte *in-core* ejecutándose en una máquina con suficiente memoria.

2.3. Soluciones al problema *out-of-core*

Hasta el momento se han implementado diversas soluciones cuando la aplicación requiere una gran cantidad de memoria física. Entre ellas tenemos:

- Incrementar la memoria física disponible. Esta solución es costosa y poco viable, sobre todo cuando el volumen de los datos es muy grande, del orden de los terabytes y está limitado por la capacidad de direccionamiento del procesador.
- Usar el mecanismo de memoria virtual de los sistemas operativos modernos. Es ineficiente para las aplicaciones de cómputo y visualización científicas si se emplean las políticas de paginado estándar (Demke B., Mowry, y Krieger, 2001).
- Reestructurar el algoritmo original con llamadas explícitas a E/S para obtener un mejor desempeño. Esta ha sido la alternativa más utilizada hasta el momento, aunque resulta laboriosa e ineficiente, pues sólo resuelve un problema particular específico con restricciones asociadas a la implementación y al hardware utilizado en ella (Suh y Prasanna, 2002).
- El soporte *out-of-core* es frecuentemente utilizado por los *solvers* que resuelven sistemas lineales dispersos mediante la aplicación de métodos directos debido a su aplicación en la solución de problemas grandes en tres dimensiones. Entre ellos están: HSL_MA78, MUMPS y PARDISO. En (Raju y Khaitan, 2009) se presenta una evaluación de estos *solvers*.
- El proyecto OOCORE (*Out-of-core solver*) desarrollado por el grupo SCALAPACK de la Universidad de Tennessee en Knoxville. Está limitado a aplicaciones muy específicas con matrices del tipo denso y está restringido a la ejecución en clusters con más de 10 procesadores (Samuel y D'Azevedo, 2004).
- Un compilador que maneja automáticamente las necesidades *out-of-core* de la aplicación. Este tema está actualmente en investigación y su aplicación está restringida al ambiente académico (Demke B., 2005).

2.4. El nuevo soporte *out-of-core*

En esta propuesta de investigación, a diferencia de los soluciones presentadas, se agrega el soporte *out-of-core* a una biblioteca numérica tipo disperso (`UCSparseLib`), aprovechando los códigos *in-core* que posee actualmente la biblioteca. Este nuevo soporte permite al usuario de la biblioteca despreocuparse por las limitaciones de la memoria física del computador ya que el soporte administra la memoria utilizada por la biblioteca cuando la capa *out-of-core* está en uso. Según se apreciará a lo largo de este trabajo, el soporte *out-of-core* incorpora todas las operaciones básicas con matrices dispersas (producto matriz-vector, producto matriz-matriz, transpuesta de matrices), incluye los métodos directos de factorización de matrices (*Cholesky*, LU y LDL^t), métodos iterativos (Jacobi, Gauss-Seidel, SOR) y métodos Multinivel Algebraico. La eficiencia de este nuevo soporte *out-of-core* se basa en: el uso de un formato CSR (CSC) modificado que permite la conversión en linea de las matrices en formato disperso *in-core* a un formato disperso *out-of-core* y en el aprovechamiento de la nueva tecnología de procesadores multi-core mediante algoritmos de prebúsqueda multihilos que solapan las operaciones de entrada/salida con cómputo.

El soporte *out-of-core* que se crea en esta investigación es diferente a los existentes y representa una **innovación** porque:

- No existían bibliotecas de solución de sistemas lineales basada en formato disperso con soporte *out-of-core*.
- No habían bibliotecas de solución de sistemas lineales con soporte *out-of-core* para resolver sistemas lineales medianos en computadores económicos.
- Existían soportes *out-of-core* solo para métodos directos usando formato disperso.
- Habían implementaciones *out-of-core* para bibliotecas de solución de sistemas lineales pero en formato denso para ser ejecutadas en clusters.

2.5. Estructura *out-of-core* para el almacenamiento de matrices dispersas

2.5.1. Almacenamiento de matrices dispersas

Según se establece en (Z. Bai y van der Vorst, 2000), los formatos de almacenamiento comprimido por fila y por columna son la forma más general de almacenar matrices dispersas, porque ellos no toman en cuenta el patrón de dispersión de la matriz y no almacenan elementos innecesarios.

$$A = \begin{bmatrix} 11 & 13 & & 16 & \\ & 22 & 24 & & \\ & 32 & 33 & & \\ 41 & & 44 & & 47 \\ & 53 & & 55 & \\ & 62 & & 66 & \\ & & 73 & 75 & 77 \end{bmatrix}$$

Figura 2: Matriz dispersa.

El formato comprimido por fila CSR (*Compressed Sparse Row*) almacena los elementos subsecuentes no nulos de las filas de la matriz en localidades contiguas de memoria. Asumiendo que se tiene una matriz dispersa no-simétrica A (ver figura 2), entonces se crean tres vectores para el almacenamiento de la matriz (ver figura 3): uno de ellos para los números en punto flotante (`val`) y los otros dos para enteros (`ia`, `id`). El vector `val` almacena los elementos no nulos de la matriz (A) en el mismo orden en el cual se obtienen, cuando la matriz se recorre por filas. El vector `id` almacena los índices correspondientes a las columnas de los elementos del vector `val`. Es decir, si $\texttt{val(k)} = a_{i,j}$, entonces $\texttt{id(k)} = j$. El vector `ia` almacena las posiciones del vector `val` que se corresponden con el inicio de cada fila; así, si $\texttt{val(k)} = a_{i,j}$, entonces $\texttt{ia(i)} \leq k < \texttt{ia(i + 1)}$. Por convención, $\texttt{ia(n)} = nnz$ para un esquema 0-índice, donde nnz es el número de elementos no nulos de la matriz A.

Los ahorros obtenidos en almacenamiento usando este formato son significativos. En lugar de almacenar n^2 números en punto flotante usando el formato denso, solamente se almacenan nz números en punto flotante mas $(n + 1 + nz)$ números enteros al usar el formato CSR. Como un ejemplo, al aplicar el formato CSR a la matriz no simétrica A, mostrada en la figura 2, se obtienen los vectores `val`, `ia` y `id` (mostrados en la figura 3). El formato de almacenamiento comprimido por columna CSC (*Compressed Sparse Column*) es idéntico al formato CSR, excepto que en vez de las filas, las columnas se almacenan subsecuentemente.

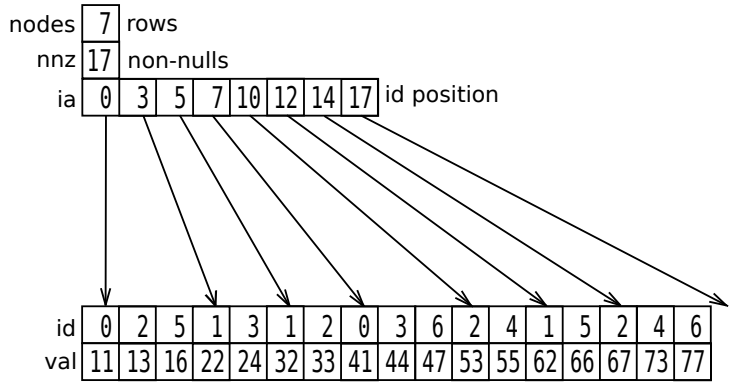


Figura 3: Formato de almacenamiento comprimido por fila (CSR).

2.5.2. Estructura de almacenamiento

La matriz es la estructura de datos central de capa *out-of-core* porque además de ser la principal consumidora de memoria, está presente en la mayoría de las funciones de la biblioteca `UCSparseLib`. La figura 4 presenta la estructura usada para el almacenamiento de una matriz.

```
struct DMatrix_t
{
  MatFormat      format; /* DENSE, CSR, CSC, DIAG */
  size_t          nnz;    /* AN/JA dimension */
  int             nn;     /* IA dimension */
  int             mm;     /* SDENSE_R, DENSE_R, SCSR or CSR -> cols */
                          /* SDENSE_C, DENSE_C, SCSC or CSC -> rows */
  SparseVec       *ia;    /* array of sparse vectors */
  double          *invD;  /* Inverse of the diagonal MATRIX_DIAG */
};
```

Figura 4: Estructura de almacenamiento de la matriz.

Como puede verse, la estructura tiene cuatro campos para el almacenamiento de datos relativos a la matriz: `format` es el formato de almacenamiento de la matriz (CSR, CSC, Denso o Diagonal), `nnz` es el número de elementos no nulos de la matriz, `nn` es la dimensión del vector `ia` (ver sección §2.5.1) y `mm` es el número de columnas (o filas) dependiendo del formato de almacenamiento de la matriz.

Cuando la capa *out-of-core* no está activada (*in-core*), cada una de las matrices se almacena en memoria en un arreglo de vectores dispersos (`*ia`), cuya dimensión es el número de filas (columnas) de la matriz de entrada. Cada uno de los elementos del arreglo `*ia`, usados por la estructura en la figura 4, es un vector disperso de tipo `SparseVec`. La estructura `SparseVec` contiene la información necesaria para el almacenamiento de un vector disperso, el cual puede representar una fila o columna dependiendo del formato de la matriz de entrada, CSR o CSC. Mediante el uso de la estructura `SparseVec` (ver figura 5), la información presente en los vectores de entrada `ia`, `id` y `val` (ver sección §2.5.1) se subdivide en tantos vectores dispersos como filas (columnas) tenga la matriz.

```
struct SparseVec
{
  int       nz;    /* Non-null elements */
  int       diag;  /* Index of the diagonal element */
  int      *id;    /* col/row values */
  double   *val;   /* row/col elements */
};
```

Figura 5: Estructura para almacenar una fila (columna).

Como se observa en la figura 5 cada uno de los vectores dispersos contiene la siguiente información: `nz`, número de elementos no nulos de la fila (columna); `diag`,

posición del elemento de la diagonal; `id`, vector de índices de la columna (fila); `val` vector con los elementos no nulos de la fila (columna).

2.5.3. Almacenamiento en disco de los vectores dispersos

Aun cuando se sabe que los formatos de almacenamiento CSR y CSC representan un ahorro significativo de memoria cuando se trabaja con matrices dispersas de gran tamaño, el espacio ocupado por el arreglo `ia` (ver figura 4) representa, en la mayoría de los casos, mas del 90 % del total del espacio requerido en memoria para almacenar toda la información relativa a la matriz. Por lo tanto, se propone como un primer paso en el diseño de la capa *out-of-core*, almacenar el arreglo `ia` en un archivo de disco de forma que solamente se mantiene en memoria un subconjunto de vectores dispersos de este arreglo, y de esta forma se reducen los requerimientos en el uso de memoria.

Dado que cada vector disperso incluye: el total de elementos no nulos (`nz`), la posición del elemento de la diagonal (`diag`), el vector de índices (`id`) y el vector de valores (`val`) como se muestra en la figura 5, la matriz dispersa puede verse como un conjunto de vectores dispersos que son almacenados secuencialmente en un archivo temporal en disco (ver figura 6). Cuando no hay elemento diagonal, se asigna el valor -1 al campo `diag`. Como la matriz dispersa se almacena en un archivo temporal en disco, la memoria principal solamente mantiene un subconjunto de los vectores dispersos contenidos en el archivo de disco. Para traer, uno o mas vectores dispersos consecutivos desde el archivo temporal en disco a la memoria, se mantiene en memoria el arreglo `pos`, que contiene las posiciones de inicio de cada uno de los vectores dispersos del archivo de disco (ver figura 6).

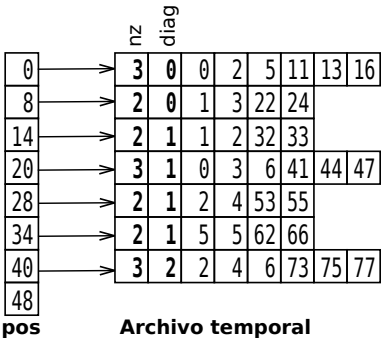


Figura 6: Formato de almacenamiento para el archivo temporal en disco.

Debido al hecho que la matriz se almacena en orden ascendente por fila (columna) en el archivo temporal de disco, se puede cargar fácilmente, desde el archivo a memoria RAM, un número consecutivo de filas (columnas), porque la posición de inicio y el tamaño del *bloque* a ser transferido puede determinarse del vector `pos`. En el

ejemplo, en la figura 6, y en el resto de esta tesis se asume que los índices y valores ocupan una unidad de almacenamiento. Esto se ha hecho para facilitar las explicaciones. En la implementación actual para la arquitectura `x86`™, los índices ocupan la mitad del espacio de los valores. Esto es porque los primeros son del tipo `integer` mientras que los últimos son `double`, en el lenguaje de programación ANSI C. Esta aproximación representa una variante del formato CSR para el almacenamiento de matrices dispersas que se propone, y se prueba con éxito, en este trabajo.

Si bien, es conocida la conveniencia de almacenar la matriz en bloques uniformes cuyo tamaño sea múltiplo del tamaño del bloque físico del dispositivo de E/S, esta tarea que es sencilla para matrices almacenadas en formato denso, resulta compleja cuando la matriz está almacenada en formato esparcido, pues requiere de una transformación fuera de linea, que ademas de incrementar el tiempo de carga de la matriz, aumenta la complejidad de la capa *out-of-core* al momento de acceder las filas (columnas) de la matriz en formato esparcido. Aprovechando las características de las funciones de entrada/salida de alto nivel `fread` y `fwrite` disponibles en ANSI C, las cuales leen/escriben desde/hacia *buffers* en memoria. En el caso de escritura, el *buffer* en memoria solo se escribe en disco cuando está lleno. El uso del archivo del archivo temporal resulta muy conveniente, como se prueba posteriormente en la parte experimental, siempre que se mantenga un acceso secuencial como es el caso de la mayor parte de las operaciones con matrices.

2.5.4. Descomposición de la matriz en bloques

Según (Toledo, 1999), para ser eficiente, un algoritmo *out-of-core* debe acceder los datos almacenados en disco como grandes *bloques* continuos. También es una buena práctica usar un *bloque* cargado en memoria tantas veces como sea posible. La capa *out-of-core* debe permitir el acceso a un conjunto consecutivo de filas/columnas (*nodos*) para aprovechar los principios de localidad espacial y temporal que soportan el concepto de memoria caché (Smith, 1982). A diferencia de las matrices densas, las matrices dispersas tienen un tamaño de fila/columna variable porque su formato de almacenamiento solamente mantiene los elementos no nulos y sus posiciones. Se debe almacenar en memoria principal al menos un *bloque* de *nodos* (filas/columnas) para aprovechar la teoría de memoria caché.

Para el acceso eficiente de los *nodos* almacenados en disco, el núcleo *out-of-core* crea una caché de correspondencia directa (Patterson y Hennessy, 2005) que aprovecha el direccionamiento de bits de los *bloques* que se corresponden con las direcciones de *nodos*. De esta forma, cuando el soporte *out-of-core* está habilitado, solamente un subconjunto de *nodos* consecutivos, llamado *bloque* de ahora en adelante, se transfiere desde el archivo temporal en disco hacia la memoria. Obviamente, este *bloque* contiene el *nodo* (fila/columna) referenciado.

Para dividir una matriz dada en *bloques* fácilmente direccionables, se completa el número total de *nodos* al próximo número entero 2^n (n: entero ≥ 0). En nuestro ejemplo, la matriz de la figura 7 tiene 7 *nodos*, así el próximo número entero 2^n es $8 = 2^3$. La figura 7 muestra la conveniencia de completar el total de *nodos* a $8 = 2^3$. Así, la matriz A puede dividirse fácilmente en *bloques* de $2^1 = 2$ *nodos* cada uno.

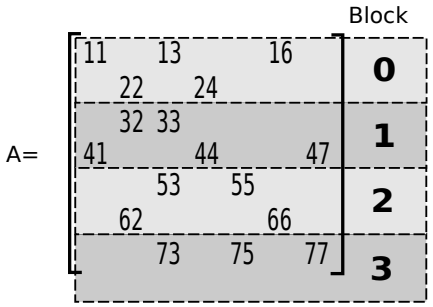


Figura 7: Division de una matriz dispersa en bloques.

Si en lugar de 7 *nodos*, la matriz tuviera 11 *nodos*, entonces el número total de *nodos* se habría completado a 2^4 *nodos*. Además, si la matriz de 11 *nodos* se divide en 8 *bloques* de $2^1 = 2$ *nodos* cada uno, solamente se almacenarán los primeros 6 *bloques* porque los dos últimos están vacíos. Para manejar los *bloques* vacíos, se almacena una referencia al último *bloque* que contiene *nodos*. Sin embargo, si uno (o mas de uno) de los primeros 6 *bloques* está vacío, entonces se almacena una estructura que indica que no existen *nodos* en este *bloque*.

2.5.5. Almacenamiento en memoria de los vectores dispersos

Si se activa la capa *out-of-core*, solamente se almacena en memoria un subconjunto de *nodos* consecutivos de la matriz. Esto se hace para acceder al disco tan pocas veces como sea posible y usar los datos cargados en memoria tantas veces como sea posible (Toledo, 1999). Para lograr este propósito, se crea en memoria, antes de cargar la matriz, una caché asociativa por conjuntos de 2^k-vías, con un tamaño de 2^m *bloques* de 2^n *nodos* cada uno (Patterson y Hennessy, 2005).

2.6. Diseño del núcleo *out-of-core*

El diseño del núcleo *out-of-core* propuesto en este trabajo soportará en el futuro todas las operaciones *out-of-core* de la biblioteca `UCSparseLib`, la cual resuelve sistemas lineales dispersos (Larrazábal, 2004). Cuando se inició el desarrollo del núcleo *out-of-core* se trabajó con el soporte de las operaciones básicas con matrices (Castellanos y Larrazábal, 2007, 2008), es decir, producto matriz-vector, producto matriz-matriz

y transpuesta de la matriz. Para facilitar la comprensión del desarrollo del núcleo en la siguientes secciones se tratarán las operaciones básicas de la capa *out-of-core* y más adelante se tratarán los refinamientos aplicados al desarrollo inicial que hicieron posible el soporte de operaciones más complejas como los métodos directos de factorización de matrices.

2.6.1. Carga de la matriz en la estructura *out-of-core*

Como ya se mencionó en la sección §2.5.3, la matriz que se leyó desde un archivo en disco se almacena en un archivo temporal cuyo formato es una variación del CSR/CSC (ver figura 6). Esta variación permite el acceso independiente de cada fila (columna) de la matriz. El núcleo *out-of-core* usa como unidad mínima indivisible un vector disperso donde se almacena una fila (columna) de la matriz. Este vector se denomina *nodo* a lo largo de este trabajo.

2.6.2. Macro para el acceso de la capa *out-of-core*

Para ocultar los detalles de implementación de la capa *out-of-core* al programador y para acceder transparentemente a cada uno de los *nodos* de la matriz, se han incorporado los algoritmos *out-of-core* dentro del código de dos macros presentes en la biblioteca `UCSparseLib`. Esta macros ya existían en la biblioteca porque ésta fue concebida desde su inicio para tener soporte *out-of-core*. De estas dos macros, una es para operar con matrices almacenadas por fila (CSR) (ver figura 8) y otra para matrices almacenadas por columna (CSC).

```
# define For_OOCMatrix_Row( MM, ii, row, mode )\
   ...                                          \
```

Figura 8: Definición de la macro `For_OOCMatrix_Row`.

Cuando el *nodo* es una fila (formato CSR) se usa la macro `For_OOCMatrix_Row` y cuando el *nodo* es columna (formato CSC) se usa la macro `For_OOCMatrix_Col`. De ahora en adelante solamente se usará el tipo fila, para simplificar las explicaciones, y por lo tanto sólo se mencionará la macro `For_OOCMatrix_Row`.

La macro `For_OOCMatrix_Row` tiene cuatro parámetros:

- `MM` es una estructura que contiene información general de la matriz, tal como: formato (CSR/CSC), número de *nodos*, apuntador al archivo temporal asociado a la matriz, etc (ver figura 4).
- `ii` es un entero que representa el *nodo* actual.

- `row` es una estructura de tipo `SparseVec` que contiene la información asociada al *nodo*; esta estructura tiene cuatro campos importantes: `nz`, total de valores no nulos del *nodo*; `diag`, posición de la diagonal; `id` y `val` son arreglos que contienen los índices del *nodo* y sus valores no nulos.

- `mode` es un campo que maneja el acceso al *nodo* actual. El puede tomar tres valores diferentes, es decir, `READ` (lectura), `WRITE` (escritura) o `READ_WRITE` (lectura/escritura).

Esta macro llama internamente todas las funciones necesarias para obtener/liberar un *nodo* desde la aplicación de alto nivel. Por supuesto, todas las rutinas necesarias para manejar tanto el archivo temporal en disco como el caché residente en memoria se invocan desde el ámbito de la macro. De esta forma los programadores pueden despreocuparse acerca de los detalles de implementación, que reducen su productividad y afectan la portabilidad de sus códigos.

2.7. Operaciones del núcleo *out-of-core*

La figura 9 muestra de una forma simplificada la operación del núcleo *out-of-core*. Se asume que la matriz A (ver figura 2) reside en un archivo con formato CSR (ver figura 3); para este ejemplo, el caché de la capa *out-of-core* se ha configurado de $2^0 = 1$ *vía* con un tamaño de *vía* de $2^0 = 1$ *bloque* de $2^1 = 2$ *nodos*. También hay un archivo temporal asociado a la matriz, como se discute en la sección §2.5.3. Para simplificar la explicación, los campos `nz` y `diag` (ver figura 6) fueron omitidos tanto en el caché como en el archivo temporal.

2.7.1. Carga de índices desde el archivo de entrada

Cuando se carga por primera vez la matriz A, ella pudiera estar almacenada en un formato por filas (CSR) o por columna (CSC). La carga del vector de índices (`ia`) se inicia después de leer la información del número total de *nodos* y del total de elementos no nulos `nnz` (ver figura 3). Los índices que se cargan desde el archivo de entrada se van escribiendo en caché hasta que éste se llena con los índices de los dos primeros *nodos*. Para liberar caché para los dos próximos *nodos* provenientes del archivo de entrada, se escribe el *bloque* previamente almacenado en caché a un archivo temporal en disco como se ve en la figura 9. Los índices leídos se almacenan junto con *ceros*, para reservar espacio para sus correspondientes valores no nulos, que se cargarán después de leer todos los índices.

El vector de valores (`val`) del *nodo* (ver figura 3) se cargará en la próxima etapa; que se inicia después que todos índices provenientes del vector `ia` han sido cargados.

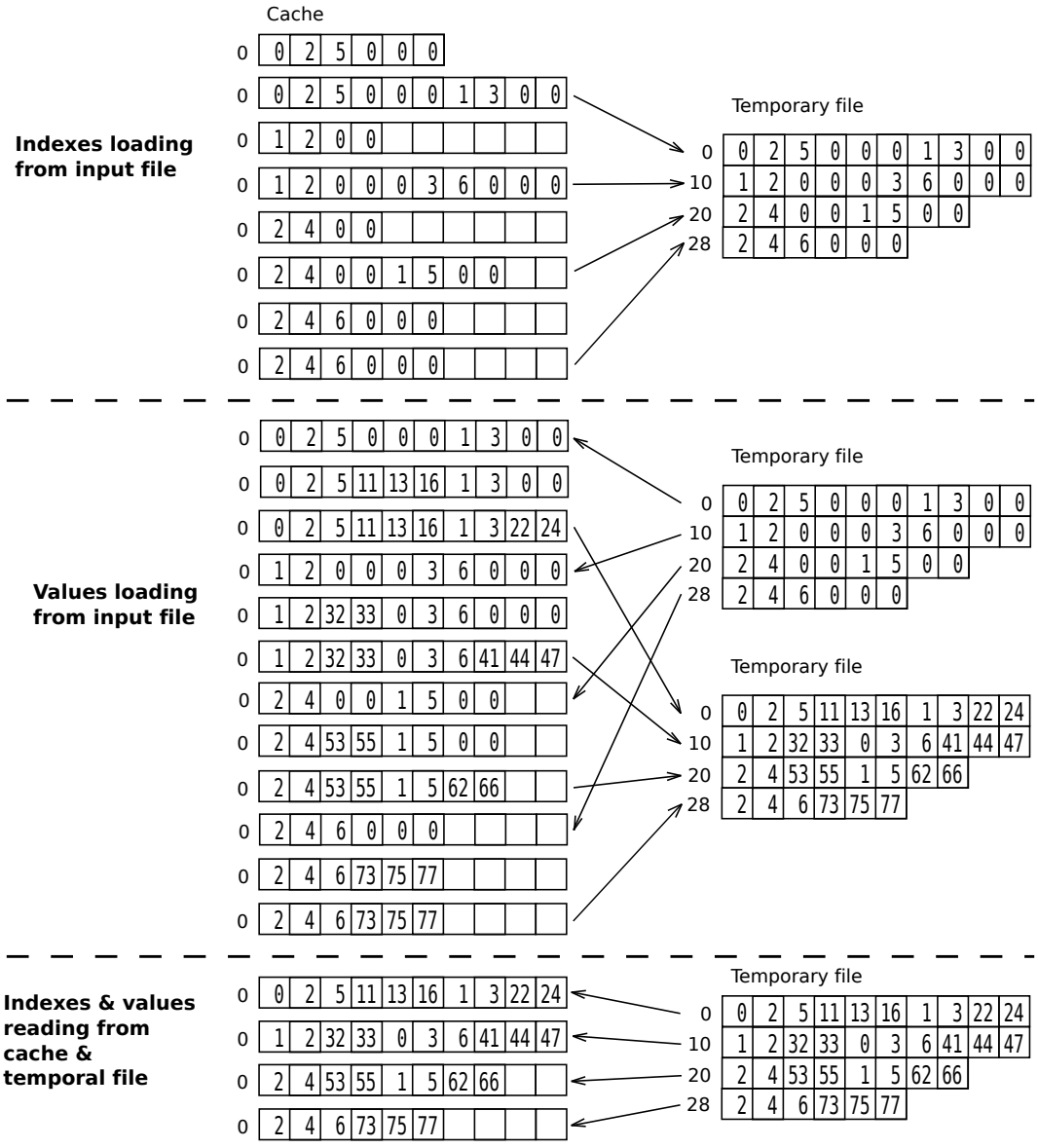


Figura 9: Operación del núcleo *out-of-core*.

La macro `For_OOCMatrix_Row` utiliza el modo `WRITE` para esta primera parte del proceso de carga de la matriz. Se usa este modo porque en esta etapa se debe preparar el tamaño de la caché para ajustarse al tamaño del bloque mas grande de filas dispersas. Además en esta etapa se crea el archivo temporal en disco que almacenará todos los *nodos* de la matriz.

2.7.2. Carga de valores desde el archivo de entrada

En la segunda parte se cargan los valores no nulos desde el vector `val` del archivo de entrada. En esta fase, se recargan desde el archivo temporal a la caché los índices que fueron cargados en la primera parte; de esta forma es posible combinar la información asociada de los índices de las columnas con sus valores no nulos correspondientes, almacenándolos en la estructura del caché y en el archivo temporal, como unidades independientes (vectores dispersos) para lograr un acceso mas eficiente a cada uno de los *nodos* cuando se trabaja con operaciones de matrices. En esta etapa se usa el modo `READ_WRITE` en la llamada a la macro `For_OOCMatrix_Row` durante el proceso de carga de la matriz. Como se mencionó anteriormente, en este modo, los *nodos* existentes en el archivo temporal de disco se cargan en caché para ser actualizados. Así, los *nodos* en caché (ver figura 9) se pueden actualizar con valores desde el vector de entrada `val` (ver figura 3). Este proceso se repite para cada *bloque* de la matriz. Cuando se inicia esta etapa, el caché no está vacío y sus contenidos se copian al archivo temporal en disco (`[2 4 6 0 0 0]`) porque la caché fue escrito en la etapa previa. Después de esto, se carga en caché el *nodo* proveniente del archivo temporal. El *bloque* que se carga en caché contiene los índices leídos en la etapa anterior para los primeros dos *nodos* de la matriz. Con estos índices en caché, los valores leídos desde el archivo de disco se mezclan con sus índices para cada uno de los dos *nodos* en el *bloque* recientemente cargado.

2.7.3. Lectura de índices y valores desde caché y el archivo temporal

La tercera parte de la figura 9 muestra como se leen secuencialmente los *nodos* desde el archivo temporal hacia caché. Esto aplica para la mayoría de las operaciones básicas sobre matrices para el acceso secuencial a los *nodos* de la matriz. Las operaciones básicas sobre matrices aprovechan el Principio de Localidad Espacial (Smith, 1982) del caché de correspondencia directa del núcleo *out-of-core*. Por ejemplo, cada vez que el algoritmo del producto matriz-vector trata de acceder a un *nodo* que no está en caché (ver figura 9), se carga un nuevo *bloque* desde el archivo temporal en disco al caché.

```
 1 for (ii= 0; ii< nn; ii++)
 2 {
 3   For_OOCMatrix_Row( M, ii, rc, READ )
 4   {
 5     raux = 0.0;
 6     for (kk= 0; kk< rc.nz; kk++)
 7      raux = raux + rc.val[kk] * xx[rc.id[kk]];
 8     yy[ii] = raux;
 9   }
10 }
```

Figura 10: Producto matrix-vector.

Por otra parte, si el *nodo* ya está en caché, entonces este se lee desde la memoria. Aunque el algoritmo del producto matriz-vector no aprovecha el Principio de Localidad Temporal del caché (Smith, 1982), el código de la figura 10 muestra buenos tiempos de ejecución. Esto es porque los *nodos* de la matriz de entrada se acceden secuencialmente y el núcleo *out-of-core* puede resolver automáticamente los *fallos* de caché cargando *bloques* de 2^n *nodos* consecutivos cada vez que ocurre un *fallo*. Así, el tiempo total de acceso a disco disminuye cuando la tasa de *aciertos* es alta (ver sección §3.2.5). En esta tercera fase se usa el modo `READ` en la macro `For_OOCMatrix_Row`, porque los *nodos* existentes en el archivo temporal de disco se cargan a caché en un modo de lectura solamente.

2.8. Mejorando la eficiencia de la capa *out-of-core*

La capa *out-of-core* en (Castellanos y Larrazábal, 2007, 2008) trabajó adecuadamente de acuerdo al propósito original de reducir los requerimientos de memoria física, permitiendo operar sobre matrices de tamaño medio en computadores económicos. Sin embargo, después de obtener esta meta, fue importante reducir el *overhead* causado por el acceso al archivo temporal en disco, sobre todo en el caso de la factorización de matrices al usar los métodos directos. Con este objetivo, se implementaron técnicas de prebúsqueda que anticipan la ocurrencia de los *fallos*. Esas técnicas también permiten solapar el acceso al archivo temporal con el cómputo, haciendo uso de la tecnología `multi-core`™ presente en casi todos los computadores modernos. Esta mejora hizo posible la reducción del *overhead*, porque al solapar los procesos de cómputo y de entrada/salida de disco, se reducen los tiempos de ejecución en las operaciones con matrices, los métodos directos de factorización de matrices y los métodos iterativos usados en el método multinivel algebraico.

2.8.1. Técnica de prebúsqueda

Cada vez que se invoca la macro `For_OOCMatrix_Row` usando los modos `READ` o `READ_WRITE` se determina un *slot*. Este *slot* es un *bloque* que contiene el *nodo* `ii`, referenciado por la macro. Si este *slot* ya está en caché, ocurre un *acierto*, de otra forma ocurre un *fallo*. A partir de este *slot*, al cual pertenece el *nodo* actual `ii`, se invoca el algoritmo de prebúsqueda, que devuelve el *slot* que debe buscarse anticipadamente. El algoritmo de prebúsqueda está basado en una tabla de predicción de referencias RPT (*Reference Prediction Table*) (Baer y Chen, 1991) que opera en conjunto con un diagrama de estado (ver figura 12), el cual determina el *slot* a ser cargado en caché. En el ejemplo de la figura 11, este algoritmo supone que cada uno de los *nodos* de la matriz `GG` está ordenado por columnas y que cada una tiene un elemento en la diagonal.

La tabla RPT contiene tantas entradas como niveles de anidamiento existan en

```
 1 for (ii= 0; ii< nn; ii++)
 2 {
 3   For_OOCMatrix_Row( GG, ii, rowI, READ_WRITE )
 4   {
 5     for (kk= 0; kk< rowI.diag; kk++)
 6     {
 7       jj = rowI.id[kk];
 8       For_OOCMatrix_Row( GG, jj, rowJ, READ )
 9       {
10         operations_1;
11       }
12       operations_2;
13     }
14   }
15 }
```

Figura 11: Anidamiento de macros.

las macros `For_OOCMatrix_Row`. Para el código de la figura 11, la RPT contiene dos entradas: la primera entrada corresponde con la macro externa y el índice de su canal es 0, y el segundo corresponde con la macro interna y el índice de su canal se establece en 1. Esto se debe al hecho que el *slot*, que se busca anticipadamente en un momento dado, puede ser diferente para cada una de las macros que hacen referencia a la misma matriz. En este caso los *slots* son los mismos, si los *nodos* a que hacen referencia las dos macros (`ii` y `jj`), pertenecen al mismo *slot*.

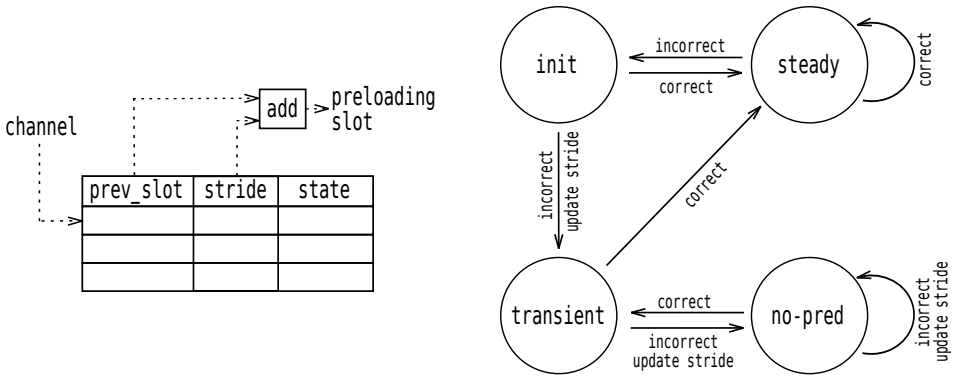


Figura 12: Tabla de predicción de referencias (RPT) y diagrama de estado.

Cada una de las entradas de la RPT contiene los siguientes campos: `prev_slot`, conteniendo el último *slot* referenciado; `stride`, la diferencia entre los dos últimos *slots* referenciados; y por último `state`, indicando cómo se realizarán las próximas prebúsquedas. El último campo está representado por una entrada que puede ser uno de los siguientes cuatro estados (ver figura 12):

- `init`: la variable de estado toma el valor `init` cuando ocurre la primera entrada en la tabla RPT o cuando se experimenta una predicción incorrecta en el estado `steady`.

- `transient`: se llega a este estado cuando el sistema no está seguro si la última predicción fue buena. En este caso se debe actualizar el valor del `stride`; el cual se calcula restando el `prev_slot` del `current_slot`.
- `steady`: este estado indica que la predicción permanecerá estable mientras que el `stride` no cambie.
- `no-pred`: este estado suspende la prebúsqueda mientras no se obtiene un `stride` correcto.

Cada vez que se invoca el algoritmo de prebúsqueda, se actualiza la entrada de la RPT que se corresponde con el nivel de anidamiento de la macro que lo llama. El nivel de anidamiento mas frecuentemente usado es el primero (*channel*= 0). Esto es porque, haya o no anidamiento, el *channel*= 0 siempre está presente. En el diagrama de de la figura 12 ocurre una predicción correcta cuando `current_slot`=`prev_slot`+`stride` y ocurre una predicción incorrecta cuando `current_slot` $\neq$ `prev_slot`+`stride`. Dependiendo del valor de la variable `current_slot` y de la última entrada presente en la tabla RPT (`prev_slot`, `stride` y `state`), puede suceder uno de los siguientes casos:

- No hay entrada en la tabla RPT para el *channel* actual porque es la primera vez que el algoritmo de prebúsqueda se invoca para una matriz y un nivel de anidamiento específico. Luego los campos de la entrada de la tabla RPT correspondientes al nivel de anidamiento de la macro se actualizan como sigue: `prev_slot`=`current_slot`, `stride`= 0 y `state`=`init`. En este caso el algoritmo de prebúsqueda no devuelve un *slot* para buscar.
- El estado actual de la entrada RPT es `init` porque la tabla RPT se acaba de inicializar o porque ha ocurrido una predicción incorrecta en el estado `steady`. Si la predicción es incorrecta entonces se actualizan los siguientes campos de la entrada RPT: `stride`=`current_slot`−`prev_slot`, `prev_slot`=`current_slot`, `state`=`transient`. Si la predicción es correcta, entonces se actualizan los siguientes campos:`prev_slot`=`current_slot`, `state`=`steady` y se retorna un *slot* para buscar. Este *slot* se calcula como `prev_slot`+`stride`.
- El estado actual es `steady` como resultado de la detección de un patrón regular desde los estados `init` y `transient`. Si la predicción es incorrecta se tiene que `prev_slot`=`current_slot` y `state`=`init`. Si la predicción es correcta `prev_slot`=`current_slot` y se retorna un *slot* para buscar, calculado como `prev_slot`+`stride`.
- El estado actual es `transient` como resultado de la detección de patrones irregulares desde los estados `no-pred` e `init`. Si la predicción es incorrecta se actualizan los siguientes campos: `stride`= `current_slot`−`prev_slot`, `prev_slot`=`current_slot` and `state`=`no-pred`. Si la predicción es correcta,

entonces `prev_slot=current_slot`, `state=steady` y se retorna el *slot*, calculado como `prev_slot+stride`.

- El estado actual es `no-pred` como resultado de una falla en la detección de un patrón desde los estados `no-pred` y `transient`. Si la predicción es incorrecta, entonces `stride=current_slot−prev_slot`, `prev_slot=current_slot` y `state=no-pred`. Con predicción correcta entonces `prev_slot=current_slot` y `state=transient` y se retorna el *slot*, calculado como `prev_slot+stride`.

2.8.2. Nueva estructura del caché

La estructura original del caché, asociativa por conjuntos de 2^k vías de 2^m bloques de 2^n *nodos* cada uno, se modificó para acomodar el *slot* proveniente de la prebúsqueda, así como también, para evitar que este *slot* prebuscado sustituya al *slot* que está siendo utilizado (*cache thrashing*). Se incorporó en cada vía un *buffer* que puede alojar un *slot* (*bloque*) completo. Generalmente el *buffer* contiene el *slot* que proviene desde el algoritmo de prebúsqueda, pero que en algunos casos puede alojar el *slot* que se obtiene cuando ocurre un *fallo* de caché.

Como se ve en la figura 13, el registro `Cnode` representa la unidad básica de la estructura del caché. Si por ejemplo el caché es asociativo por conjuntos de 2 vías, con *bloques* de 2 *nodos*, entonces cada vía consistirá de cinco (5) `Cnodes`, uno que se comporta como un *buffer* y cuatro (4) que operan juntos como una caché de correspondencia directa para almacenar los *bloques* 0, 1, 2 y 3 de la matriz.

Cada registro `Cnode` puede almacenar en memoria un *slot* de la matriz, la cual está compuesta de 2^p *nodos* (filas o columnas). Cada `Cnode` almacena los datos almacenados en un *slot* de disco en la forma de un arreglo de datos del tipo `Cell` (ver figura 13). `Cell` es un tipo genérico que puede contener en un momento dado dos datos del tipo `int` o un dato del tipo `double`. Esto es necesario para leer/escribir eficientemente los *slots* desde el archivo temporal de disco. Cada *slot* está formado por una mezcla de índices (`int`) con valores (`double`). Ver el ejemplo del archivo temporal en la figura 6. El campo `size` mantiene el tamaño del vector `data`, el campo `slot` mantiene el número del *slot* en el archivo de disco, `lslot` corresponde al número del último *slot* que estuvo almacenado en el `Cnode` actual, `lru` es una variable entera que ayuda a determinar cual es el `Cnode` en caché que será reemplazado bajo la política LRU (*least recently used*). El campo `w` contiene un conjunto de banderas binarias que permiten determinar entre otras cosas: a cual vía del caché pertenece el `Cnode`, conocer si el `Cnode` es un `buffer` o es parte del caché de correspondencia directa. Si el `Cnode` no es un `buffer` entonces es un *bloque* de la caché de correspondencia directa, por lo tanto el campo `block` corresponde en este caso al número de *bloque*. La bandera `dirty` en el campo `w` permite conocer si el `Cnode` debe ser actualizado en disco antes de ser reemplazado; la bandera `writing` permite determinar si el vector `data`

```
typedef union
{
    int       id[2];
    double    val;
} Cell;
typedef struct
{
    unsigned    dirty : 1;
    unsigned    valid : 1;
    unsigned    fetch : 1;
    unsigned    noexc : 1;
    unsigned    nouse : 1;
    unsigned     lock : 1; /* 1: node lock */
    unsigned  writing : 1;
    unsigned      via : 3; /* 2^0, .. 2^3 */
    unsigned   buffer : 1; /* 1:yes / 0:no */
    unsigned    block : 5; /* 0, .. 32 */
    unsigned   unused :16;
} Cflfields;
typedef union
{
    char      todo[4];
    Cflfields t;
} Cflags;
struct Cnode
{
    Cell     *data;
    int       size;
    int       slot;
    int       lslot;
    unsigned  lru;
    Cflags    w; /* via, buffer, block, dirty, writing, lock */
};
```

Figura 13: Registro para almacenar un *slot* en caché.

del `Cnode` está siendo actualizado; la bandera `lock` sirve para evitar que el `Cnode` sea reemplazado por uno nuevo. Cuando un `Cnode` escrito (bandera `dirty=verdadero`) va a ser reemplazado, el campo `lslot` contiene el *slot* al cual pertenecen los datos en el vector `data` y el campo `slot` indica el *slot* que se debe obtener desde el archivo temporal en disco.

2.8.3. Lista de solicitudes pendientes (*Outstanding request list*)

Para obtener un buen desempeño de las técnicas de prebúsqueda aplicadas, el proceso de entrada/salida de disco se debe ejecutar mientras que el algoritmo de cálculo accede a los *nodos* de la matriz y realiza operaciones de punto flotante. Para lograr esto, se diseñó una lista de solicitudes pendientes (ORL: *Outstanding request list*) de acuerdo al esquema propuesto por (Kroft, 1981). Como se ve en la figura 14, la ORL es una lista enlazada simple de nodos del tipo `Cnode`. El `Cnode` se constituye en la unidad básica del caché (ver sección §2.8.2).

La ORL es una estructura re-entrante que se accede por dos hilos (*threads*). El

```
struct _ORLlist
{
    Cnode             *node;
    struct _ORLlist   *next;
} ORLlist;
```

Figura 14: Lista de solicitudes pendientes (ORL).

primer hilo corresponde al programa principal y el segundo es un hilo creado junto con la estructura de la matriz y ejecuta todas las operaciones de entrada/salida a través del programa `ActCache`. Cada vez que se invoca la macro `For_OOCMatrix_Row` usando los modos `READ` o `READ_WRITE`, se determina el *slot* al cual pertenece el *nodo* actual. Si se encuentra que el *nodo* actual `ii`, pertenece a un *slot*, diferente al *slot* previamente cargado en la última llamada a la macro `For_OOCMatrix_Row`, entonces se verifica si el nuevo *slot* ya está presente en caché. Al final, se invoca el algoritmo de prebúsqueda y éste retorna, o no, un nuevo número de *slot*, que debe obtenerse del archivo en disco.

Si el *slot* calculado no existe en caché, entonces se genera una nueva solicitud, por un `Cnode`, que se agrega a la lista ORL. La nueva solicitud por el `Cnode` es un apuntador a un `Cnode` que ya existe en el caché (ver figura 14). Este `Cnode` inicialmente hace referencia a un *slot* (en disco), necesario para cubrir la solicitud. Así, la ORL mantiene todo el tiempo, las solicitudes de los *slots* que están pendientes por obtenerse desde el archivo temporal en disco.

La ORL se accede concurrentemente por un segundo hilo que ejecuta el programa `ActCache`, el cual aparte de encontrar los *slots* solicitados por el programa principal, es responsable de actualizar los *slots*, modificados en caché, en el archivo temporal de disco. Solamente los *slots* modificados se actualizan en disco, de acuerdo a la política *write-back*(Patterson y Hennessy, 2005). Antes de actualizar un *slot* en disco, el programa `ActCache` verifica el contenido de la bandera `dirty` (ver sección §2.8.2). Para decidir cual `Cnode` debe ser reemplazado en caché, el programa `ActCache` sigue la política LRU (*least recently used*) usando el campo `lru` de `Cnode` (ver figura 13).

El hilo principal ejecuta el programa que realiza el cómputo. El segundo hilo, creado desde el hilo principal, se arranca cuando se crea la estructura *out-of-core* de la matriz, para manejar las solicitudes de la ORL a través del programa `ActCache`. Así en cualquier momento, para cada matriz, a lo sumo se están ejecutando dos hilos. Los dos hilos se sincronizan por medio de variables de condición de la biblioteca `pthread`. Cada vez que el programa principal genera una solicitud ORL, él envía una señal al segundo hilo. Entonces el segundo hilo accede a la lista de solicitudes y maneja la solicitud. Cuando el segundo hilo finaliza el manejo de la solicitud, éste envía una señal al hilo principal. Después revisa la lista ORL en busca de mas solicitudes pendientes. Si no hay mas solicitudes en la lista, el hilo secundario se suspende en una variable de condición y allí espera la señal del hilo principal para manejar futuras

solicitudes.

Aunque con este esquema se reducen los *fallos* de caché, estos todavía pueden ocurrir. Cuando un *slot* solicitado desde la macro no está en caché entonces ocurre un *fallo*. En este caso, se genera una solicitud que se agrega a la ORL. A diferencia de los *slots* prebuscados, para los cuales se puede solapar el cómputo con la entrada/salida, el programa principal se suspende en una variable de condición mientras que el *slot* requerido se obtiene desde el archivo temporal en disco por el hilo secundario.

3. Evaluación de la capa *out-of-core*

3.1. Introducción

En este capítulo se explica en detalle el proceso de implantación y evaluación de la capa *out-of-core*. Para ello este capítulo se descompone en tres grandes secciones: operaciones básicas con matrices dispersas, métodos directos de factorización de matrices y método multinivel algebraico. En cada sección se explican las modificaciones en el código original de la biblioteca `UCSparseLib` para soportar la capa *out-of-core* y se presentan tablas y gráficas que muestran los resultados experimentales con matrices de prueba que permiten apreciar, mediante comparaciones, el desempeño en uso de memoria y tiempo de ejecución del núcleo *out-of-core*.

3.2. Operaciones básicas con matrices dispersas

El soporte *out-of-core* se activa a través de una bandera de compilación llamada `OOC`. Esta bandera compila condicionalmente los códigos del núcleo *out-of-core* dentro de la macro `For_OOCMatrix_Row`. Este enfoque facilita la incorporación de nuevas funcionalidades en la capa *out-of-core* y la evaluación de la calidad de la implementación del soporte porque el uso de la macro es común para los códigos *in-core* y *out-of-core*. En la figura 10 se muestra el código para el producto matriz-vector; nótese que el acceso a cada una de las filas de la matriz se hace a través de la macro `For_OOCMatrix_Row`.

3.2.1. Carga de la matriz a la estructura *out-of-core*

Es es una operación que se realiza cada vez que se lee una matriz desde un archivo en disco y se carga en la estructura *out-of-core* conformada por: la estructura `DMatrix_t` (ver figura 4), el caché en memoria y el archivo temporal en disco (ver figura 9). Esta operación es común a todas las pruebas con operaciones básicas donde se lee la matriz de entrada desde un archivo en disco con formato CSR (ver figura 3) y se carga en la estructura *out-of-core*. Como ya se explicó detalladamente en la sección §2.6. esta operación se compone de: carga de índices desde el archivo de entrada y carga de valores desde el archivo de entrada.

3.2.2. Producto matriz-vector

El producto matriz-vector solamente toma ventaja del Principio de Localidad Espacial de la caché de correspondencia directa (Smith, 1982) del núcleo *out-of-core*. Cada vez que el algoritmo producto matriz-vector, accede a un *nodo* que no está en la caché, este se carga desde el disco a la caché dentro de un nuevo *bloque* de 2^n *nodos*. Si por el contrario, el *nodo* está en la caché, este se carga desde la memoria. El código de la figura 10 exhibe buenos tiempos de ejecución porque los *nodos* de la matriz de entrada se acceden en orden ascendente y el soporte *out-of-core* resuelve automáticamente los *fallos* de caché, cargando *bloques* consecutivos de *nodos*, así la caché puede operar con una tasa de *fallos* baja ($< 1\,\%$). Esta situación se analiza en el parágrafo Parámetros del caché de la sección §3.2.5.

3.2.3. Transpuesta de la matriz

Para la operación transpuesta de la matriz fue necesario aprovechar, además del Principio de Localidad Espacial, el Principio de Localidad Temporal(Smith, 1982), con el objeto de reducir el *overhead* ocasionado por el acceso a disco. Experimentalmente se determinó que configurando el caché con cuatro vías para la matriz de salida se podía reducir la tasa de *fallos* a menos del $2\,\%$. Para modificar el número de vías de caché, en tiempo de ejecución, se creó el procedimiento `TDMatrixSetVias`.

Según se aprecia en el código de la figura 15, en la primera línea se establecen los números de vías para la matriz de entrada M en 1 y para la matriz de salida W en 4. En este código podemos apreciar el uso de los tres modos de la macro `For_OOCMatrix_Row`. En la línea cuatro de la figura 15 se usa el modo `WRITE` para reservar el espacio donde se almacenará cada una de las filas (columnas) de la matriz transpuesta W. En la línea trece de la figura 15 se usa el modo `READ` para leer los elementos de cada una de las filas (columnas) de la matriz de entrada M. Finalmente, en la línea dieciocho de la figura 15 se usa el modo `READ_WRITE` para modificar el contenido de la matriz W a medida que se lee cada uno de los elementos de la matriz de entrada M. Según se aprecia en la linea once, la matriz M se recorre en orden inverso para obtener la matriz de salida W ordenada.

3.2.4. Producto matriz-matriz

En la figura 16 se presenta el código esquematizado del producto matriz-matriz. La operación producto matriz-matriz ($A \times B$) requiere del acceso a dos matrices, A y B, almacenadas en dos cachés independientes.

La matriz A se accede en forma ascendente aprovechando el Principio de Localidad Espacial(Smith, 1982), por lo tanto el diseño de la caché para esta matriz amerita

```
 1  TDMatrixSetVias( M, 1 ); TDMatrixSetVias( W, 4 );
 2  for (ii= 0; ii< mm; ii++)
 3  {
 4    For_OOCMatrix_Row( W, ii, rc1, WRITE )
 5    {
 6      /* el vector iwa contiene el núm de elementos de
 7         cada fila/col */
 8      rc1.nz  = iwa[ii];
 9    }
10  }
11  for (ii= nn-1; ii>= 0; ii--)
12  {
13    For_OOCMatrix_Row( M, ii, rc1, READ )
14    {
15      for (jj= 0; jj< rc1.nz;  jj++)
16      {
17        col  = rc1.id[jj];
18        For_OOCMatrix_Row( W, col, rc2, READ_WRITE )
19        {
20          kk = iwa[col] - 1;
21          iwa[col] = kk;
22          rc2.id[kk]  = ii;
23          rc2.val[kk] = rc1.val[jj];
24          if (ii == col)
25          {
26            rc2.diag = kk;
27            rc2.invd = 1.0 / rc2.val[kk];
28          }
29        }
30      }
31    }
32  }
```

Figura 15: Transpuesta de la matriz.

una sola vía. Según apreciamos en las líneas 17 de la figura 15 y la línea 12 de la figura 16 el acceso a la matriz W de la operación transpuesta es similar al acceso a la matriz B de la operación producto matriz-matriz por lo que experimentalmente se corroboró que el diseño del caché para la matriz B debía tener cuatro vías para tener una tasa de *fallos* inferior al 2 %. Según se aprecia en el código de la figura 16, en la primera línea se establece el número de vías para las matrices A y B en 1 y 4 respectivamente, mientras que la matriz $Z = A \times B$ se ajusta con un tamaño de 1 vía. Obsévese que las matrices A y B se acceden en modo `READ` mientras que la matriz del resultado (Z) se accede ascendentemente en modo `WRITE`.

3.2.5. Resultados con operaciones básicas.

Los códigos para evaluar el soporte *out-of-core* con operaciones básicas, se compilaron usando `gcc x86_64` versión 4.6 sin usar banderas de optimización[1] , sobre un

[1]Al habilitar las banderas de optimización cuando se usa la biblioteca `pthread` se genera un *overhead* sobre las operaciones de creación de hilos y en el uso de los elementos de sincronización (mutex y variables de condición) porque las banderas de optimización afectan el uso de las variables

```
 1  TDMatrixSetVias( A, 1 ); TDMatrixSetVias( B, 4 ); TDMatrixSetVias( W, 1 );
 2  iSet( nbColB, iwaIP, -1 ); /* inic. componentes del vector iwaIP con -1 */
 3  for (ii= 0; ii< nbRowA; ii++)
 4  {
 5    For_OOCMatrix_Row( W, ii, rowW, WRITE )
 6    {
 7      For_OOCMatrix_Row( A, ii, rowA, READ )
 8      {
 9        for (kk= 0, nu= 0; kk< rowA.nz; kk++)
10        {
11          val = rowA.val[kk];
12          jj  = rowA.id[kk];
13          For_OOCMatrix_Row( B, jj, rowB, READ )
14          {
15            for (ll= 0; ll< rowB.nz; ll++)
16            {
17              mm = rowB.id[ll];
18              if (iwaIP[mm] < 0)
19              {
20                rowW.id[nu]  = mm;
21                rowW.val[nu] = val*rowB.val[ll];
22                iwaIP[mm]    = nu;
23                nu++;
24              }
25              else
26                rowW.val[iwaIP[mm]] += val*rowB.val[ll];
27            } /* Unlook de B  */
28          } /* kk */
29        }
30      } /* A */
31      /* Código para determinar posición de la diagonal */
32    } /* W */
33  } /* ii */
```

Figura 16: Producto matriz-matriz.

computador portátil con procesador *dual-core*, Intel Core 2 Duo P8600™, con 4GB de RAM, ejecutando el sistema operativo `GNU/Linux` con Kernel `3.0`. Los archivos temporales se almacenaron en disco duro usando el sistema de archivos `ext3`. La memoria de intercambio de 8GB estuvo habilitada durante todas las pruebas (`swapon`).

Matrices de prueba. Las matrices de prueba (ver Tabla 1) se generaron a partir de la discretización por diferencias finitas de una ecuación $3D$ escalar de convección-difusión (Larrazábal, 2002).

El tamaño en filas de cada matriz de prueba se seleccionó de forma que coincidiera con una potencia de dos (2^n). Esto se hizo para analizar el comportamiento de la capa *out-of-core* para cada uno de los diferentes tamaños de bloque que a su vez son función de 2^n. El tamaño de la matriz mas pequeña (Identificador 1) se determinó por el menor tiempo apreciable para la operación producto matriz-vector ($0,001$ segundos). El tamaño de la matriz mas grande (Identificador 14) se determinó por la cantidad de memoria necesaria para efectuar el producto matriz-vector, de forma que excediera

globales necesarias para la sincronización y señalización de los hilos.

Tabla 1: Matrices de prueba para las operaciones básicas.

Identificador	**Filas** 2^n	**Total filas**	**No nulos**
1.	2^{13}	8.192	54.784
2.	2^{14}	16.384	110.592
3.	2^{15}	32.768	223.232
4.	2^{16}	65.536	448.512
5.	2^{17}	131.072	901.120
6.	2^{18}	262.144	1.810.432
7.	2^{19}	524.288	3.629.056
8.	2^{20}	1.048.576	7.274.496
9.	2^{21}	2.097.152	14.581.760
10.	2^{22}	4.194.304	29.196.288
11.	2^{23}	8.388.608	58.458.112
12.	2^{24}	16.777.216	117.047.296
13.	2^{25}	33.554.432	234.225.664
14.	2^{26}	67.108.864	468.713.472

el límite de la memoria RAM disponible (4GB).

Programas de prueba. Para evaluar el producto matriz-vector se creó un programa de prueba que carga desde disco la matriz de entrada A y la multiplica por un vector x, cuyas componentes se generan aleatoriamente. El programa produce un vector $c = Ax$ que se escribe en disco y sirve para verificar el resultado con y sin el núcleo *out-of-core* activado. Para evaluar tanto la operación transpuesta como el producto matriz-matriz, se creó un segundo programa de prueba que genera la matriz transpuesta $B = A^T$ a partir de la matriz de entrada A. Luego, este mismo programa calcula la matriz C, que se escribe en disco, como el producto de la matriz de entrada por la matriz B, que a su vez es la transpuesta de A, así $C = A \times A^T$. Al igual que en caso anterior, la matriz resultante sirve para verificar los resultados con el núcleo *out-of-core* activado y desactivado respectivamente.

Proceso de prueba. Como se discutió en el párrafo anterior, se implementaron dos programas de prueba que permiten evaluar el desempeño de las operaciones básicas. En las operaciones básicas, la capa *out-of-core* tiene la capacidad de solapar el cómputo con la comunicación, para atenuar el efecto producido por las latencias de memoria y disco. Por esta razón, las pruebas se realizaron usando un computador con procesador multi-core. Con el objeto de evaluar el desempeño de las técnicas de prebúsqueda agregadas al núcleo *out-of-core*, el proceso de prueba se realizó en tres fases:

a Compilación y ejecución de los programas de prueba con la capa *out-of-core* deshabilitada. En esta etapa la matriz de entrada A se carga en memoria física y las operaciones básicas se ejecutan totalmente sobre memoria.

b Compilación y ejecución de los programas de prueba con la capa *out-of-core* habilitada pero sin activar las técnicas de prebúsqueda. En esta etapa, no se efectúa prebúsqueda de *nodos* desde el archivo temporal mientras se realizan las operaciones de cómputo, así que la matriz de entrada A se carga en la estructura *out-of-core* y las operaciones básicas se ejecutan sobre los *nodos* cargados en el caché de la capa *out-of-core*. Durante la ejecución de las macros (`For_OOCMatrix_Row`) de la capa *out-of-core* se usa la ORL y no hay solapamiento entre cómputo y entrada/salida (ver sección §2.8.3). Aunque se usa la ORL, existe un *overhead* causado por la ausencia de prebúsqueda. Cuando sucede un *fallo*, el *nodo* que no está en caché (debido a la ausencia de prebúsqueda) debe ser leído desde el archivo temporal, ocasionando *overhead* por el tiempo adicional necesitado para obtenerlo desde el disco. El hilo principal que ejecuta el cómputo debe esperar hasta que el hilo secundario, encargado del proceso de entrada/salida, finalice la obtención del *nodo* desde el disco. Aun cuando esta etapa, con prebúsqueda desabilitada, se puede realizar usando un solo hilo, cuando se diseñaron los algoritmos de la capa *out-of-core*, todas las operaciones de entrada/salida se concentraron en el segundo hilo con el objeto de tener la entrada/salida exclusivamente en el segundo hilo. Esto permitió tener códigos multihilos robustos y probados para la prebúsqueda habilitada. Se considera que con este esquema se puede apreciar mejor el efecto de la prebúsqueda y el *overhead* que se genera al usar un segundo hilo en esta etapa es muy pequeño.

c Compilación y ejecución de los programas de prueba con la capa *out-of-core* y las técnicas de prebúsqueda habilitadas. En esta etapa la matriz de entrada A se carga en la estructura *out-of-core* y las operaciones se efectúan usando técnicas de prebúsqueda (ver sección §2.8.1), las cuales permiten la carga temprana de los *slots*, antes de que ocurran los *fallos*. En esta etapa tanto la ORL como las técnicas de prebúsqueda están activas.

En cada una de las fases se midieron las siguientes variables: tiempo que toma la operación (producto matriz-vector, transpuesta de la matriz, producto matriz-matriz), la cantidad de memoria necesaria para llevar a cabo la operación completa. En todos los casos, el tiempo se midió usando PAPI (*Performance Application Programming Interface*) versión 3.7.0 (Terpstra, 2009). Se usaron las funciones internas provistas por la biblioteca `UCSparseLib` para medir la cantidad de memoria utilizada.

Parámetros del caché. Antes de la prueba final, con la capa *out-of-core* activada, se realizaron pruebas de cada una de las operaciones con cada una de las matrices de entrada usando diferentes configuraciones de caché observando en cada caso el tiempo

de ejecución y la cantidad de memoria utilizada. De acuerdo a los resultados obtenidos en esta primera etapa de prueba, se seleccionó la organización de caché mas conveniente. Se determinó experimentalmente en cada caso el tamaño de *bloque* expresado como una potencia 2^n, es decir, si por ejemplo el tamaño de *bloque* determinado fue 5, esto significa que el número de *nodos* que contiene el *slot* es $2^5 = 32$. Se estableció por omisión para la capa *out-of-core* un caché de correspondencia directa (Patterson y Hennessy, 2005), porque es el más rápido en tasa de *aciertos*, aunque de acuerdo a lo comentado en las secciones §3.2.3 y §3.2.4, para las operaciones transpuesta de la matriz y producto matriz-matriz, se cambia dinámicamente la organización del caché para algunas matrices a asociativa por conjuntos de n-vías con el objeto reducir la tasa de *fallos* y obtener así mejoras en el desempeño.

Uno de los casos más frecuentes de acceso al caché sucede cuando se accede secuencialmente a las filas (columnas) de una matriz; un ejemplo de ello es el producto matrix-vector (ver figura 10). En la tabla 2 se muestra el comportamiento del caché para matriz de prueba 7. En ta tabla se aprecia que a medida que se incrementa el número de nodos/bloque va aumentando la tasa de aciertos y por tanto desciende el tiempo que toma la operación producto matriz-vector debido a que con una mayor tasa de aciertos hay un menor número de accesos a disco duro (lecturas). Experimentalmente se determinó que para una tasa de aciertos mayor a 99,90 se obtiene un comportamiento aceptable del caché. Cuando el tamaño del bloque es muy grande ocurren dos efectos adversos: hay un incremento en la cantidad de memoria necesaria para mantener el bloque en memoria principal y hay un aumento en el costo de entrada/salida porque los bloques transferidos son mas grandes, los cuales demandan *buffers* de entrada/salida residentes en memoria RAM de mayor tamaño. Esta situación se refleja en los resultados presentados en la tabla 2 cuando el tamaño de bloque es mayor de 2^{11}.

En referencia al tamaño del caché, en número de *bloques*, se observó que el tiempo de ejecución para todas las operaciones básicas no decrece significativamente incrementando el número de bloques en caché; entonces el caché se configuró con un tamaño de un único *bloque*. El tamaño en *nodos* (vectores dispersos) del *bloque* fue variable y se seleccionó para cada matriz de entrada con el objeto de obtener el mejor tiempo de ejecución. Cada caso de prueba se repitió tres veces y en cada caso se tomó el menor tiempo para decidir el mejor tamaño de *bloque* y así obtener los resultados de las tablas 4, 5 y 6.

Tiempo de carga de la matriz. Con el objeto de cuantificar el tiempo que toma el proceso de carga del archivo de la matriz en disco a la estructura *out-of-core* se evaluaron las matrices de prueba solamente para la operación producto matriz-vector porque este proceso es igual para las otras operaciones básicas. Para determinar el *overhead* que introduce la capa *out-of-core* en este proceso se midieron los tiempos de carga desde el archivo en disco a la estructura *in-core* para cada una de las matrices

Tabla 2: Desempeño del caché para el producto matriz-vector sin prebúsqueda.

Nodos/Bloque	**Tiempo**	**Aciertos**	**Lecturas**
2^0	3,596	0,00	262.144
2^1	1,479	50,00	131.072
2^2	0,919	75,00	65.536
2^3	0,483	87,50	32.768
2^4	0,267	93,75	16.384
2^5	0,133	96,88	8.192
2^6	0,107	98,44	4.096
2^7	0,072	99,22	2.048
2^8	0,055	99,61	1.024
2^9	0,047	99,80	512
2^{10}	0,043	99,90	256
2^{11}	0,042	99,95	128
2^{12}	0,043	99,98	64
2^{13}	0,047	99,99	32
2^{14}	0,049	99,99	16

Tiempo: expresado en segundos; **Aciertos**: en porcentaje.
Matriz de prueba: **Id.** 6. **Total de accesos**: 262.144

de prueba (ver tabla 3). Según se aprecia en la tabla la matriz 13 tiene una tiempo de carga muy alto con la capa *out-of-core* desabilitada porque la matriz de entrada es muy grande y rebasa la memoria física disponible. Para la matriz 14 no se muestra el tiempo de carga *in-core* porque no fue posible realizar la operación con la memoria física disponible.

En la figura 17 se presenta la relación de tiempos $\frac{in-core}{out-of-core}$ para las matrices 1 hasta 12. Las matrices 1 hasta 11 muestran una relación de tiempos sobre 0,75 lo cual significa que el *overhead* que introduce la capa no supera el 25 % del tiempo *in-core*. También se puede concluir que la prebúsqueda no introduce mejoras en el proceso de carga; esto era de esperarse porque en el proceso de carga no se realizan cálculos que pueden solaparse con las operaciones de entrada/salida. La relación de tiempos de la matriz 12 está por debajo de 0,5 debido posiblemente a que el tamaño del bloque seleccionado es pequeño en comparación con el tamaño total de la matriz y no se está aprovechando en forma apropiada el acceso al archivo temporal en disco.

Tamaños de bloque utilizados. Para el producto matriz-vector (ver tabla 4), el tamaño mínimo de bloque seleccionado fue 2^5 *nodos* para la matriz mas pequeña y el máximo fue 2^{12} *nodos* para la matriz 12. Para la operación transpuesta (ver tabla 5) el tamaño mínimo de bloque seleccionado fue 2^8 *nodos* para la matriz 2 y el máximo

Tabla 3: Tiempo para la carga de la matriz

Id.	**in-core**	**out-of-core**			
		sin prebúsqueda		**con prebúsqueda**	
	Tiempo	**Bloque**	**Tiempo**	**Bloque**	**Tiempo**
1.	0,029	7	0,037	5	0,038
2.	0,060	9	0,069	7	0,070
3.	0,119	9	0,140	7	0,140
4.	0,239	9	0,279	9	0,272
5.	0,481	11	0,551	10	0,547
6.	0,975	11	1,118	10	1,098
7.	1,955	11	2,244	10	2,211
8.	3,931	11	4,770	11	4,645
9.	7,951	11	9,225	10	9,139
10.	16,050	11	20,192	11	20,296
11.	35,239	11	41,112	11	40,551
12.	70,661	12	153,854	12	143,522
13.	3.635,529	11	498,121	11	436,082
14.	**	10	895,786	10	891,144

Tiempo: está en segundos; **Blq**: tamaño de bloque en 2^n filas.

******: La operación no se pudo realizar por limitaciones en la memoria física.

fue 2^{14} *nodos* para las matrices 13 y 14. Para el producto matriz-matriz (ver tabla 6) el tamaño mínimo de bloque seleccionado fue 2^8 *nodos* para la matriz 2 y el máximo fue 2^{14} *nodos* para las matrices 11, 13 y 14. Esto muestra una tendencia de incremento en el tamaño de bloque según se incrementa el tamaño de la matriz de entrada para las operaciones básicas.

Presentación de los resultados. Los resultados mostrados en las tablas 4, 5 y 6 tienen la misma estructura; la tabla 4 presenta los requerimientos de memoria, los resultados del tiempo de ejecución y los tamaños de bloque determinados experimentalmente para la operación matriz-vector. En cada tabla, la primera columna presenta un identificador numérico (**Id**) el cual se corresponde con una matriz de prueba seleccionada desde la tabla 1. Las dos siguientes columnas muestran el uso de memoria y el tiempo de ejecución con la capa *out-of-core* deshabilitada. Las últimas seis columnas muestran los resultados con la capa *out-of-core* habilitada. Las columnas cuatro, cinco y seis presentan los resultados con la prebúsqueda deshabilitada y las últimas tres columnas (siete, ocho y nueve) se corresponden con los resultados con la prebúsqueda habilitada. Los resultados con la capa *out-of-core* habilitada se presentan en cada caso (sin prebúsqueda, con prebúsqueda) en tres columnas: la primera es la potencia

Tabla 4: Uso de memoria y tiempo de ejecución para el producto matriz-vector

Id.	**in-core**		**out-of-core**					
			sin prebúsqueda			**con prebúsqueda**		
	Mem.	**Tiempo**	**Blq.**	**Mem.**	**Tiempo**	**Blq.**	**Mem.**	**Tiempo**
1.	1,06	0,001	7	0,15	0,002	5	0,13	0,002
2.	2,14	0,001	9	0,35	0,003	7	0,28	0,002
3.	4,30	0,003	9	0,60	0,006	7	0,53	0,004
4.	8,63	0,005	9	1,10	0,011	9	1,10	0,007
5.	17,31	0,011	11	2,41	0,021	10	2,20	0,016
6.	34,72	0,022	11	4,41	0,042	10	4,21	0,033
7.	69,53	0,045	11	8,41	0,083	10	8,21	0,061
8.	139,25	0,087	11	16,41	0,164	11	16,41	0,117
9.	278,88	0,173	11	32,41	0,325	10	32,22	0,232
10.	558,13	0,347	11	64,42	0,658	11	64,42	0,456
11.	1.117,00	0,831	11	128,44	1,315	11	128,44	0,904
12.	2.235,50	1,504	12	256,84	2,657	12	256,84	1,924
13.	4.472,50	1.676,734	11	512,53	31,050	11	512,53	29,130
14.	**	**	10	1024,70	91,165	10	1024,70	83,132

Mem.: Memoria expresada en Megabytes; **Tiempo** está en segundos; **Blq**: tamaño de bloque en 2^n filas.

******: La operación no se pudo realizar por limitaciones en la memoria física.

entera de 2 que indica el número de filas que tiene cada bloque, la segunda muestra el uso de memoria en Megabytes y la tercera el tiempo de ejecución en segundos.

En las figuras 18, 20 y 22 se presenta gráficamente el comportamiento en tiempo de ejecución de la capa *out-of-core* para el producto matriz-vector, operación transpuesta y producto matriz-matriz. En cada una de las figuras se muestra la relación entre el tiempo *in-core* (con la capa deshabilitada) y el tiempo *out-of-core* para las matrices de prueba con la prebúsqueda habilitada y deshabilitada. La relación $\frac{tiempo\ \ in-core}{tiempo\ \ out-of-core}$ es para la mayoría de los casos un número menor que uno porque *tiempo in-core*<*tiempo out-of-core* debido al *overhead* ocasionado por el acceso al archivo temporal en disco cuando la capa *out-of-core* está habilitada.

Finalmente, las figuras 19, 21 y 23 muestran gráficamente la cantidad de memoria utilizada para cada operación con cada una de las matrices de prueba. En cada caso se graficó la relación entre la memoria necesaria para cada operación con la capa *out-of-core* desactivada y la memoria requerida con la capa activada. Cuando la capa está activa se graficaron los casos con la prebúsqueda habilitada y deshabilitada. La relación $\frac{memoria\ \ in-core}{memoria\ \ out-of-core}$ es siempre un número mayor que uno, porque cuando la capa *out-of-core* está activa solamente se almacena en memoria principal un

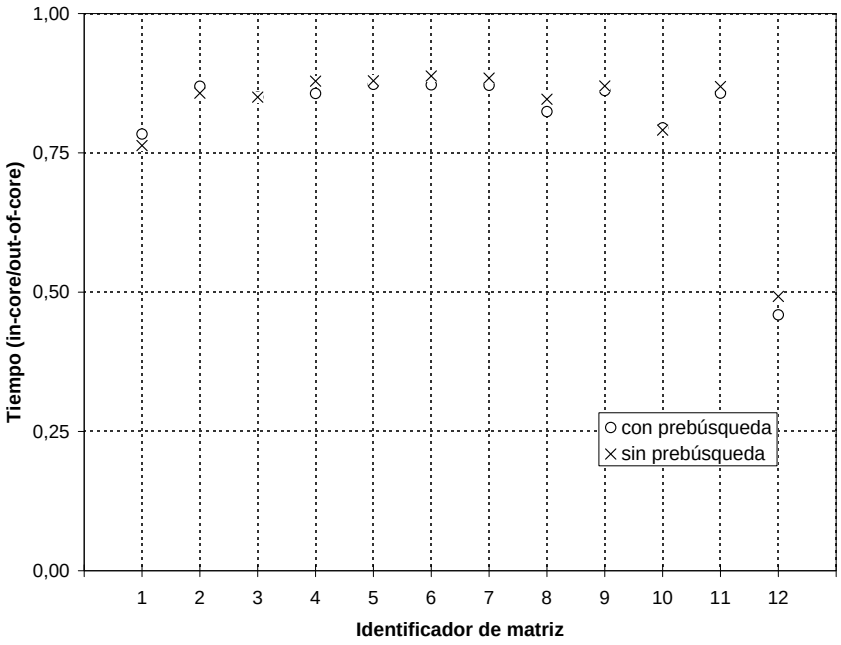


Figura 17: Relación de tiempo $\frac{in-core}{out-of-core}$ para la carga de la matriz desde un archivo.

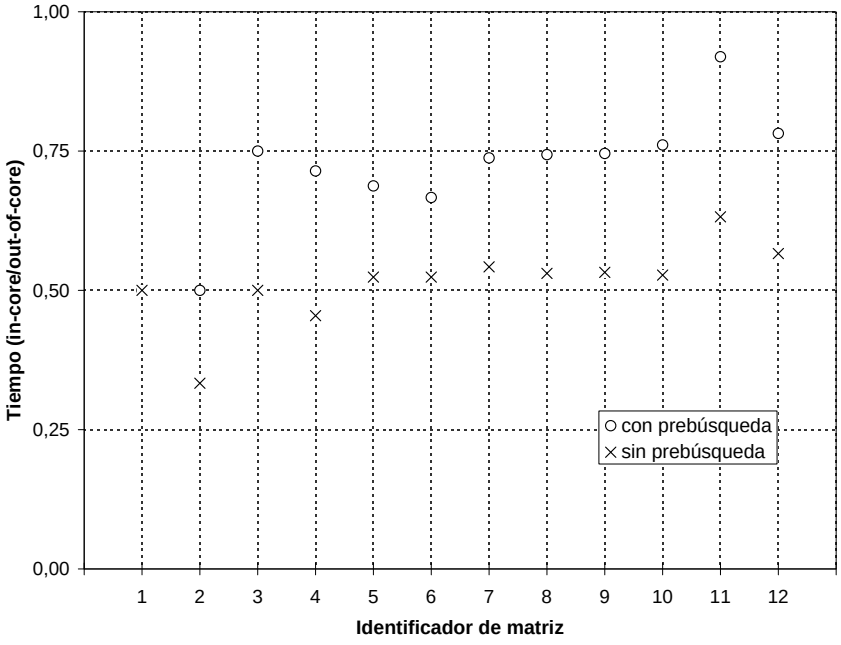


Figura 18: Relación de tiempo $\frac{in-core}{out-of-core}$ para el producto matriz-vector.

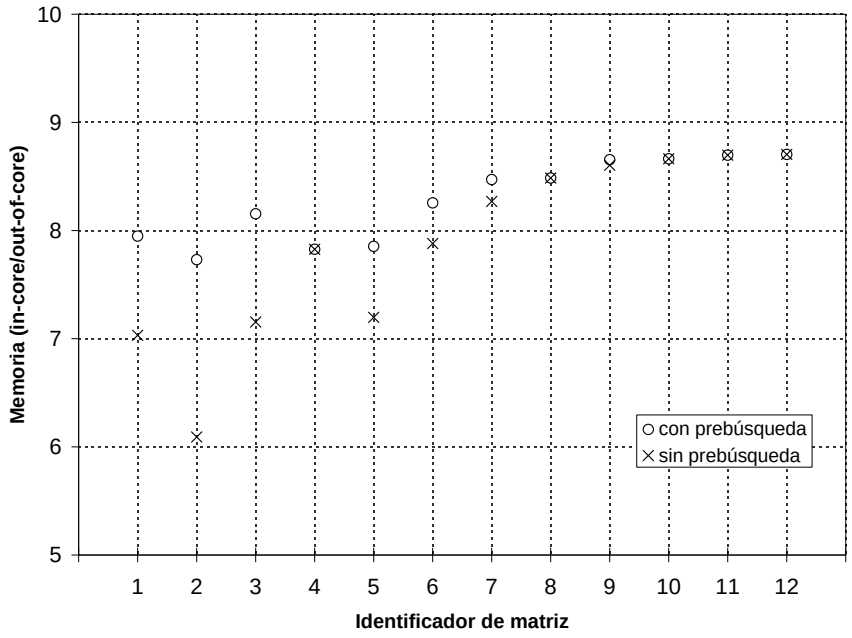


Figura 19: Relación de memoria $\frac{in-core}{out-of-core}$ para el producto matriz-vector.

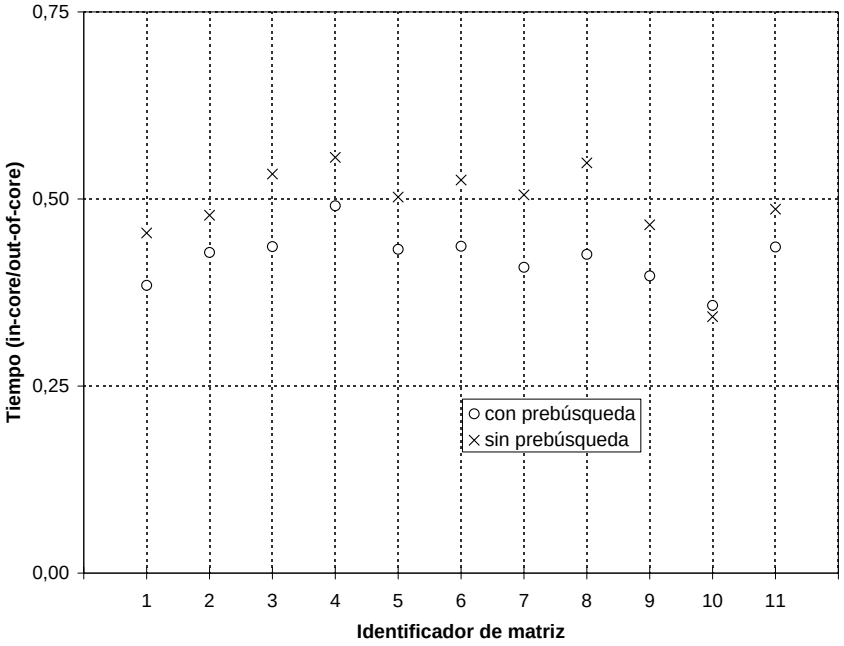


Figura 20: Relación de tiempo $\frac{in-core}{out-of-core}$ para la operación transpuesta.

Tabla 5: Uso de memoria y tiempo de ejecución para operación transpuesta

Id.	**in-core**		**out-of-core**					
			sin prebúsqueda			**con prebúsqueda**		
	Mem.	**Tiempo**	**Blq.**	**Mem.**	**Tiempo**	**Blq.**	**Mem.**	**Tiempo**
1.	1,91	0,005	9	0,63	0,011	9	0,63	0,013
2.	3,84	0,011	8	0,67	0,023	8	0,67	0,028
3.	7,73	0,024	9	1,01	0,045	9	1,01	0,055
4.	15,52	0,050	10	2,01	0,090	10	2,01	0,110
5.	31,13	0,098	10	3,69	0,195	10	3,69	0,238
6.	62,44	0,218	10	5,02	0,415	9	4,51	0,499
7.	125,06	0,458	9	8,52	0,906	10	8,23	0,893
8.	250,50	0,921	12	22,79	1,680	9	16,88	2,218
9.	501,75	1,842	10	33,05	3,957	11	34,04	4,710
10.	1.024,25	3,650	10	65,08	10,650	9	63,63	10,545
11.	2.010,00	7,517	12	132,09	15,455	10	129,14	17,246
12.	4.023,00	2.986,510	13	264,15	74,493	13	264,15	78,003
13.	**	**	14	528,27	314,534	14	538,28	337,328
14.	**	**	14	1.051,29	816,472	14	1051,29	729,181

Mem.: Memoria expresada en Megabytes; **Tiempo** está en segundos; **Blq**: tamaño de bloque en 2^n.

******: La operación no se pudo realizar por limitaciones en la memoria física.

subconjunto de filas de la matriz de entrada (A).

Análisis de resultados. Para el análisis de resultados se usarán las figuras 18, 19, 20, 21, 22 y 23. La figura 18 muestra que el comportamiento del tiempo de ejecución del producto matriz-vector sin usar técnicas de prebúsqueda es bastante uniforme y tiende a un valor de $0,50$, lo cual significa que el tiempo de ejecución con la capa *out-of-core* habilitada tiende a ser el doble del tiempo de ejecución *in-core* para la mayor parte de las matrices de entrada. Esto se debe principalmente al *overhead* causado por el acceso del archivo temporal en disco que contiene la totalidad de los vectores dispersos de la matriz de entrada A. Cuando observamos en esta misma figura los resultados con las técnicas de prebúsqueda habilitadas vemos una mejora notable en todos los casos (con la excepción de la matriz 1), debido al hecho que el algoritmo del producto matriz-vector (ver figura 10) aprovecha el solapamiento entre entrada/salida con el cómputo. Como se aprecia en la figura 18, la relación de tiempos se incrementa a 0,75 con la prebúsqueda habilitada, lo cual señala que el tiempo de ejecución con la capa *out-of-core* habilitada tiende a ser un 50 % mas que el tiempo *in-core*, lo cual señala que el *overhead* causado por el acceso del archivo temporal

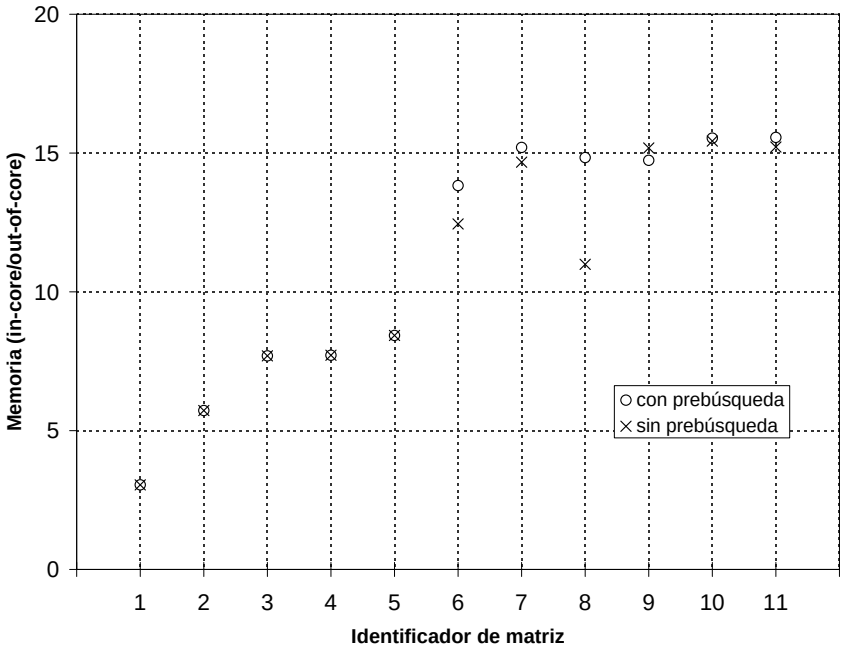


Figura 21: Relación de memoria $\frac{in-core}{out-of-core}$ para la operación transpuesta.

en disco se ha reducido a la mitad. En la figura 10 no se muestran los resultados de las matrices 13 y 14 de la tabla 4. En el caso de la matriz 13 esto se debe a que la relación de tiempos $\frac{in\text{-}core}{out\text{-}of\text{-}core}$ es muy grande (56,57) y al incluir este resultado en la gráfica no se podrían apreciar los resultados para las matrices 1 al 12; el incremento en esta relación para la matriz 13 sucede porque la memoria virtual resulta muy ineficiente en el manejo de los accesos a la memoria de intercambio (en disco duro) en comparación con el desempeño de la capa *out-of-core*. En caso de la matriz 14 no fue posible calcular la relación $\frac{in\text{-}core}{out\text{-}of\text{-}core}$ porque no se pudieron obtener resultados al realizar la operación *in-core*. La figura 19 muestra el comportamiento del producto matrix-vector en referencia al uso de memoria sin y con técnicas de prebúsqueda habilitadas. Se aprecia que en el grupo de las matrices mas pequeñas la ausencia de prebúsqueda incrementa el uso de memoria con la capa *out-of-core* habilitada. Esta situación se debe al hecho que para disminuir el efecto incremento del tiempo ejecución ocasionado por los *fallos* de caché para las matrices mas pequeñas sin aplicar técnicas de prebúsqueda se seleccionaron tamaños de bloque mas grandes con el consiguiente aumento en el uso de memoria de la capa *out-of-core*. Recordemos que el producto matriz-vector solamente aprovecha el Principio de Localidad Espacial(Smith, 1982), por lo cual para disminuir el número de *fallos* se debe incrementar el tamaño del *bloque*.

La figura 20 muestra que el comportamiento del tiempo de ejecución de la operación transpuesta no es tan uniforme como en el producto matriz-vector. Contrario

Tabla 6: Uso de memoria y tiempo de ejecución para el producto matriz-matriz

Id.	**in-core**		**out-of-core**					
			sin prebúsqueda			**con prebúsqueda**		
	Mem.	**Tiempo**	**Blq.**	**Mem.**	**Tiempo**	**Blq.**	**Mem.**	**Tiempo**
1.	4,46	0,027	8	0,53	0,030	9	0,93	0,033
2.	13,29	0,112	8	0,97	0,115	8	0,97	0,115
3.	18,25	0,114	10	2,11	0,120	9	1,31	0,138
4.	36,70	0,228	10	2,62	0,246	10	2,62	0,242
5.	109,64	0,935	10	4,90	0,963	10	4,90	0,998
6.	148,48	0,923	11	7,26	1,013	9	4,83	1,170
7.	297,79	1,863	12	14,53	2,027	10	9,14	1,026
8.	890,54	7,645	12	27,64	8,504	12	27,64	9,097
9.	1.197,89	7,503	13	45,07	8,347	11	35,30	8,653
10.	2.399,15	15,120	12	70,58	18,143	9	65,01	18,712
11.	4.805,03	201,714	9	129,20	50,854	14	154,18	58,888
12.	**	**	13	269,16	173,156	13	269,16	142,521
13.	**	**	13	525,21	398,203	14	525,21	324,798
14.	**	**	14	1.070,81	1.315,464	14	1.070,81	1056,930

Mem.: Memoria expresada en Megabytes; **Tiempo** está en segundos; **Blq**: tamaño de bloque en 2^n.

******: La operación no se pudo realizar por limitaciones en la memoria física.

al caso anterior, se observa que las técnicas de prebúsqueda no producen mejoras en el desempeño del algoritmo de la operación transpuesta (ver figura 15). Sin la prebúsqueda habilitada, la relación de tiempo $\frac{in\text{-}core}{out\text{-}of\text{-}core}$ tiende a un valor de 0,5, la cual es muy similar al obtenido para el producto matriz-vector sin usar prebúsqueda. En este caso, cuando se habilita la prebúsqueda, el tiempo de ejecución se desmejora; en el caché de la capa *out-of-core* ocurre un problema conocido como *cache thrashing* el cual consiste en que los *slots* prebuscados reemplazan en caché los *slots* que se van a utilizar, incrementando el número de *fallos* y por consiguiente incrementando el número de accesos a disco, en vez de disminuirlos. Esto se debe a que los accesos a la matriz W (ver figura 15) no siguen un patrón que se pueda detectar eficientemente desde el algoritmo de prebúsqueda. Como en el caso de la figura 18 no se muestran los resultados para las matrices 12, 13 y 14. Para las matrices 13 y 14 no fue posible realizar la operación *in-core* por limitaciones en la memoria física y en en la matriz 12, la relación de tiempo $\frac{in\text{-}core}{out\text{-}of\text{-}core}$ es muy alta (40,09) para ser representada correctamente en conjunto con los resultados de las otras matrices. Como en el caso del producto matriz-vector la memoria virtual presenta un desempeño inferior al de la capa *out-of-core* cuando debe acceder los datos desde el disco duro. La figura 21 muestra el comportamiento de la operación transpuesta en referencia al uso de me-

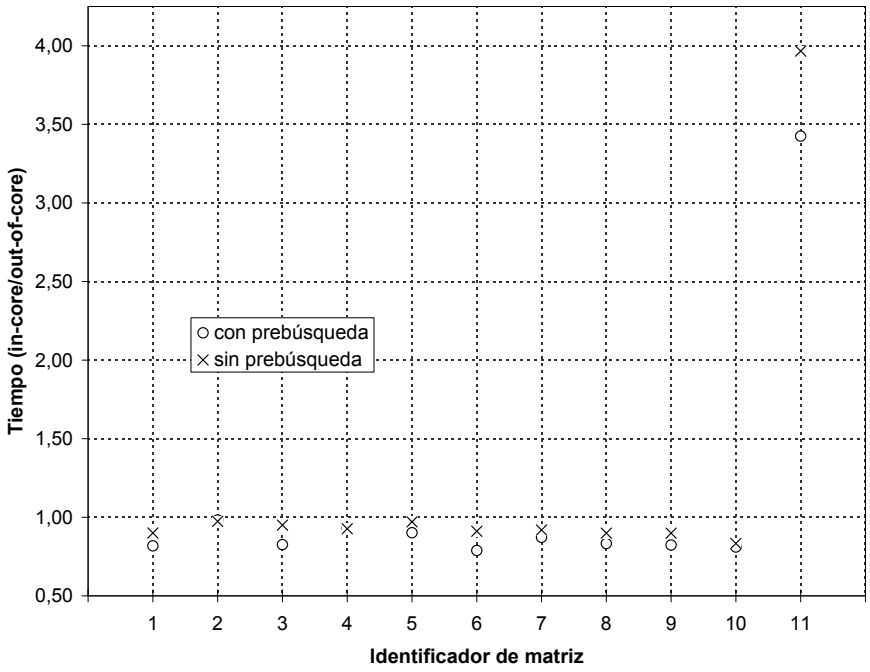


Figura 22: Relación de tiempo $\frac{in-core}{out-of-core}$ para el producto matriz-matriz.

moria, sin y con técnicas de prebúsqueda, habilitadas. Se aprecia que en la mayoría de las matrices, con excepción de las matrices 6 y 8, el uso de memoria no se afecta por hecho de aplicar las técnicas de prebúsqueda. Esto se debe a que las técnicas de prebúsqueda no mejoran el desempeño del algoritmo de la transpuesta por efecto del ya comentado problema del *cache thrashing*.

La figura 22 muestra que el comportamiento del tiempo de ejecución del producto matriz-matriz no está muy afectado por el uso de técnicas de prebúsqueda. Además el comportamiento es bastante uniforme, con excepción de la matriz 11 y tiende a un valor de $0,75$, lo cual significa que el tiempo de ejecución, con la capa *out-of-core* habilitada, tiende a ser muy similar al obtenido para el producto matriz-vector con la prebúsqueda habilitada. Con la matriz 11 sucede una situación especial donde el tiempo, con la capa *out-of-core* habilitada, es mucho mejor que el tiempo de solución *in-core*. Observando la tabla 6 vemos que para la matriz 11, la solución *in-core* requiere mas de 4GB, lo cual excede a la memoria física disponible por el sistema. En este caso, el sistema operativo hace uso de la memoria de intercambio, la cual no ofrece la misma eficiencia de la capa *out-of-core* para el acceso de las estructuras almacenadas en la memoria de intercambio en disco duro. Por su parte podemos apreciar en la misma tabla que cuando la capa *out-of-core* está habilitada solamente se necesitan $129,20$ y $154,18$MB (sin y con prebúsqueda) para la realizar el producto de matrices con la capa habilitada. No se presentan en la figura los resultados para las matrices 12, 13

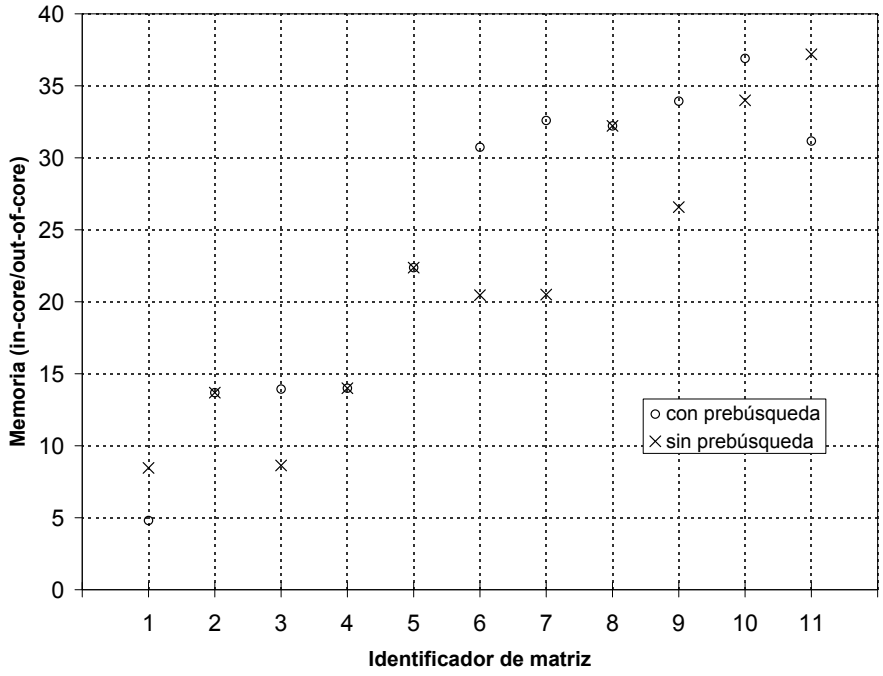


Figura 23: Relación de memoria $\frac{in-core}{out-of-core}$ para el producto matriz-matriz.

y 14 porque no fue posible obtener resultados sin la capa *out-of-core* habilitada. En estos casos de prueba se nota la conveniencia del uso de la capa *out-of-core*, cuando el *hardware* que se dispone es limitado. La figura 23 muestra el comportamiento del producto matriz-matriz con respecto al uso de memoria, sin y con técnicas de prebúsqueda habilitadas. Se aprecia en esta gráfica que no existe un patrón claro de comportamiento que permita concluir, que al aplicar las técnicas de prebúsqueda, haya una mejora apreciable. Entre las once matrices de prueba se distinguen cinco matrices (3, 6, 7, 9, 10) donde se obtienen mejoras en el uso de memoria al aplicar prebúsqueda; hay cuatro matrices (2, 4, 5 8) donde la prebúsqueda no muestra cambios en las necesidades de memoria y finalmente hay un grupo de dos matrices (1, 11) que casualmente son la mas pequeña y la mas grande, donde al activar la prebúsqueda desmejora el ahorro de memoria.

3.3. Métodos directos de factorización de matrices

Frecuentemente el núcleo computacional en el software de simulación es el *solver* del sistema lineal. Este solver puede ser denso o disperso dependiendo de la discretización numérica. Si la matriz del sistema relacionada con el sistema lineal es dispersa, es deseable usar una estructura dispersa. Los *solvers* lineales dispersos directos tales

como *Cholesky*, LDL^T, or LU son perfectas cajas negras, es decir, necesitan como entradas solamente la matriz A y el vector del lado derecho b del sistema lineal $Ax = b$. No obstante, una de las principales desventajas es que la memoria que ellos necesitan para resolver el sistema se incrementa rápidamente con el tamaño del problema. De allí la importancia de extender la capa *out-of-core* para los métodos de factorización de forma que se resuelve el problema del uso de memoria. Esta desventaja se puede ver claramente en un estudio reciente (N. I. M. Gould y Hu, 2007).

```
 1  iSet( nbrows, first, NIL );
 2  iSet( nbrows, seen,   -1 );
 3  for (ii= 0; ii< nbrows; ii++)
 4  {
 5    maxCol = -1; minCol = nbrows;
 6    For_OOCMatrix_Row( AA, ii, rowIA, READ )
 7    {
 8      for (jj= 0; jj< rowIA.diag; jj++)
 9      { /* Lazo en la vecindad (sin el elemento diagonal) */
10        col = rowIA.id[jj];
11        if (seen[col] != ii)
12        {
13          seen[col] = ii;
14          minCol = MIN( minCol, col );
15          maxCol = MAX( maxCol, col );
16          col = first[col];
17          /* Calc. conj alcanzable desde L */
18        }
19      }
20      For_OOCMatrix_Row( GG, ii, rowIG, WRITE )
21      {
22        /* operaciones varias */
23      }
24    }
25  }
```

Figura 24: Factorización simbólica *Cholesky*.

3.3.1. Factorización *Cholesky*

El problema de la factorización *Cholesky out-of-core* de matrices dispersas no es nuevo; ya en 1984 se presentó un *solver Cholesky* Multifrontal *out-of-core* (Reid, 1984). En (Rothberg y Schreiber, 1999) se comparan tres métodos de factorización *Cholesky* basados en su eficiencia y se proponen alternativas para su mejora. En un artículo reciente (Reid y Scott, 2009) se presenta un *solver Cholesky out-of-core*. En un artículo ya publicado de este autor (Castellanos y Larrazábal, 2011) y en esta tesis, a diferencia de (Reid y Scott, 2009), la factorización *Cholesky out-of-core* se implementa como parte del soporte *out-of-core* para la biblioteca `UCSparseLib` (Larrazábal, 2004). Según ya se ha comentado en secciones anteriores, la capa *out-of-core* se basa en el desarrollo de un caché especializado (en memoria principal) que almacena solamente una parte de la matriz del problema A y del factor L. Las matrices A y L se almacenan en archivos temporales en disco. El factor L se calcula en un

proceso de dos pasos, específicamente: un proceso simbólico y uno numérico. El primer paso calcula la posición de los elementos no nulos para cada fila/columna y el segundo paso calcula el valor numérico de cada posición usando el Método Multifrontal.

Factorización de la matriz. La factorización *out-of-core Cholesky* de una matriz dispersa se ejecuta en proceso de cuatro pasos:

a Lectura de la matriz del sistema desde un archivo en disco y almacenamiento en un archivo temporal usando el formato modificado CSR/CSC de acuerdo a la sección 2.5.3.

b Reordenamiento de la matriz de entrada usando METIS (Karypis y Kumar, 1999) para reducir el número de elementos no nulos en el factor L y por lo tanto reducir el tiempo del proceso de factorización. Este paso de reordenamiento no usa la estructura *out-of-core*. El software METIS lee la matriz de entrada desde el archivo y retorna dos vectores de permutación para reordenar la matriz de entrada.

c Factorización simbólica de la matriz, para determinar la posición de los elementos no nulos del factor L. Durante este paso se crea una estructura *out-of-core* para manejar la matriz factor L. Esta matriz es mas densa que la matriz de entrada (ver figura 24).

d Factorización numérica de la matriz, para calcular el valor de cada uno de los elementos no nulos del factor L. La factorización numérica se basa en el algoritmo *Cholesky* por filas `ikj`, donde las filas del factor L se calculan sucesivamente; cada uno de los elementos de la fila se calcula usando las filas previamente calculadas así como los elementos calculados en la misma fila (ver figura 25).

El tiempo de procesamiento de la factorización está determinado por el cuarto paso del proceso, es decir, la factorización numérica de la matriz, la cual toma mas del 50 % del tiempo total para la factorización*in-core* (N. I. M. Gould y Hu, 2007). El primer y tercer paso del proceso de factorización tienen la ventaja que dependen del acceso secuencial de la matriz de entrada. Esto significa que el núcleo *out-of-core*, ya probado para las operaciones básicas con matrices(Castellanos y Larrazábal, 2007, 2008), muestra un buen comportamiento. Por otra parte, los pasos tres y cuatro del proceso de factorización presentan dos problemas adicionales que forzaron el rediseño del núcleo *out-of-core* original.

El primer problema es el anidamiento de las macros `For_OOCMatrix_Row` necesarias para la factorización numérica. En la figura 24 se aprecia un extracto del código de la factorización simbólica. En las líneas 6 y 20 se aprecian las llamadas a las macros `For_OOCMatrix_Row`. Aunque las dos macros están anidadas, en este caso no hay

```
 1  TDMatrixSetVias( GG, 2 );
 2  for (ii= 0; ((ii< nbrows) && !singular); ii++)
 3  {
 4    For_OOCMatrix_Row( GG, ii, rowI, READ_WRITE )
 5    { /* Para todos los elem. antes de la diag en la fila ii */
 6      for (kk= 0; kk< rowI.diag; kk++)
 7      {
 8        jj = rowI.id[kk];
 9        For_OOCMatrix_Row( GG, jj, rowJ, READ )
10        {
11          nnz = rowJ.nz;
12          rowJ.nz = rowJ.diag;
13          TDSparseVecDot( &rowJ, &rowI, &tt );
14          rowJ.nz = nnz;
15          rowI.val[kk] = (rowI.val[kk] - tt) * rowJ.invd;
16        }
17      }
18      /* El elemento diag. en la fila ii */
19      nnz = rowI.nz;
20      rowI.nz = rowI.diag;
21      TDSparseVecDot( &rowI, &rowI, &tt );
22      rowI.nz = nnz;
23      tt = rowI.val[rowI.diag] - tt;
24      tt = sqrt(tt);
25      rowI.val[rowI.diag] = tt;
26      rowI.invd   = 1.0 / tt;
27      range = range + 1;
28    }
29  }
```

Figura 25: Factorización numérica *Cholesky*.

anidamiento de matrices porque las dos macros hacen referencia a matrices diferentes; en este caso `AA` y `GG`. El anidamiento de macros lo podemos ver en la factorización numérica, según se evidencia en las líneas 4 y 9 de la figura 25. Si tomamos como referencia la figura 25, para implementar eficientemente el anidamiento, el caché debe mantener el *slot* que contiene el *nodo* `ii` (macro externa), mientras que el *slot* que contiene el *nodo* `jj` (macro interna) también se mantiene en caché. Desde el punto de vista del diseño de la capa *out-of-core*, esto significó mejorar la organización del caché para soportar múltiples canales (*channels* - niveles de anidamiento). También esto tuvo que ver con la incorporación de un mecanismo que permite bloquear en caché el *slot* que contiene el *nodo* `ii`, mientras se acceden los *slots* que contienen cada *nodo* `jj`. En la sección §2.8.2 se explica la inclusión de un *buffer* en la estructura del caché que ayuda en la solución de este problema. En el campo `w` de la estructura `Cnode` se agregó una bandera llamada `lock` para evitar que el *slot* que contiene al *nodo* `ii` sea removido del caché mientras que se llevan a cabo las iteraciones en el lazo interno. De esta forma, el *slot* que contiene al *nodo* `ii` se mantiene en caché mientras la bandera `lock` es `cierta`.

El segundo problema corresponde al incremento de las tasas de *fallos* durante la factorización numérica de la matriz, el cual desmejora el tiempo de ejecución por el alto costo del acceso a disco. El incremento en la tasa de *fallos* se debe, en gran parte, al anidamiento de macros requerido para ejecutar la factorización numérica de

la matriz (ver figura 25). Para resolver este problema se incorporó un algoritmo de prebúsqueda (ver sección §2.8.1). Este algoritmo determina el *slot* que debe traerse al caché antes de que suceda el *fallo*. Para resolver los requerimientos concurrentes de acceso al caché por los algoritmos que manejan los *fallos* y las prebúsquedas, se diseñó e implementó una lista de solicitudes pendientes (ORL) cuyo funcionamiento se explicó en la sección §2.8.3.

Para mejorar aún mas el desempeño de la factorización numérica se determinó experimentalmente que usando un caché asociativo por conjuntos de 2 vías para la matriz `GG` (ver figura 25), se reducía significativamente el número de *fallos* de caché y por tanto se mejoraba el tiempo de ejecución. Por eso en la línea 1 de la figura 25 se establece en 2 el número de vías de la matriz `GG` previa a la realización de la factorización numérica.

3.3.2. Factorización *LU*

El problema de la factorización *LU out-of-core* de matrices dispersas es muy similar al ya tratado para la factorización *Cholesky*. El proceso de factorización se realiza igualmente en cuatro pasos de forma similar al ya explicado para la factorización *Cholesky* (ver sección 3.3.1). La principal diferencia desde el punto de vista algorítmico está en que, tanto la fase de factorización simbólica como la fase de factorización numérica, producen a la salida dos factores (L y U) en lugar de un único factor como era el caso de la factorización *Cholesky*. En la figura 26 se aprecia la factorización simbólica. Podemos ver, al igual que en la factorización *Cholesky*, que no hay anidamiento de macros en esta sección del código porque las macros externa e interna hacen referencia diferentes matrices.

En la figura 27 se aprecia la factorización numérica. Podemos ver al igual que en la factorización *Cholesky* que hay anidamiento de macros en esta sección del código porque las macros externa e internas hacen referencia a la misma matriz según se ven las líneas 5 y 14 donde las macros externa e interna invocan a la matriz `UU`. Al igual que en la factorización *Cholesky*, en la factorización *LU* el tiempo de procesamiento de la factorización está determinado por el cuarto paso del proceso, es decir, la factorización numérica de la matriz, la cual toma mas del 50 % del tiempo total para la factorización *in-core*(N. I. M. Gould y Hu, 2007). Al igual que en la factorización *Cholesky*, hay un incremento de las tasas de *fallos* durante la factorización numérica de la matriz la cual desmejora el tiempo de ejecución por el alto costo del acceso a disco. El incremento en la tasa de *fallos* se debe al anidamiento de macros requerido para ejecutar la factorización numérica de la matriz (ver figura 27). Para resolver este problema se usa el algoritmo de prebúsqueda (ver sección §2.8.1) ya implementado y probado para la factorización *Cholesky*. Este algoritmo determina el *slot* que debe traerse al caché antes de que suceda el *fallo*. Para mejorar aún mas el desempeño de la factorización *LU* se determinó experimentalmente que usando un caché asociativo

```
 1  TDMatrixSetVias( UU, 2 );
 2  for (ii= 0; ii< nbrows; ii++)
 3  {
 4    minColL = nbrows; maxColL = -1;
 5    minColU = nbrows; maxColU = -1;
 6    For_OOCMatrix_Row( AA, ii, rowIA, READ )
 7    {
 8      for (jj= 0; jj< rowIA.diag; jj++)
 9      { /* Lazo en la vecindad (sin el elem diagonal */
10        col = rowIA.id[jj];
11        if (seen[col] != ii)
12        {
13          /* Calcula conj alcanzable desde L */
14        }
15      }
16      for (jj= rowIA.diag+1; jj< rowIA.nz; jj++)
17      {
18        col = rowIA.id[jj]; seen[col] = ii;
19        minColU = MIN( minColU, col );
20        maxColU = MAX( maxColU, col );
21      }
22      /* Calcula conj alcanzable desde U */
23      kk = firstU[ii];
24      while (kk != NIL)
25      {
26        For_OOCMatrix_Row( *UU, kk, rowIU, READ )
27        {
28          /* operaciones varias */
29        }
30        kk = nextU[kk];
31      }
32      /* Lazo en no nulos de L. Se debe agregar diagonal */
33      For_OOCMatrix_Row( *LL, ii, rowIL, WRITE )
34      {
35        /* operaciones varias */
36        /* Agrega elem diagonal */
37      }
38      /* Lazo en los no nulos de U. Se debe agregar diagonal */
39      For_OOCMatrix_Row( *UU, ii, rowIU, WRITE )
40      {
41        /* Agrega elem diagonal */
42        /* Ajuste final */
43      }
44    }
45  }
```

Figura 26: Factorización simbólica LU.

por conjuntos de 2 vías para la matriz `UU` (ver figura 26), se reducía significativamente el número de *fallos* de caché y por tanto se mejoraba el tiempo de ejecución tanto para la factorización simbólica como para la factorización numérica. Por eso en la línea 1 de la figura 26 se establece en 2 el número de vías de la matriz `UU`, previa a la realización de las factorizaciones simbólica y numérica.

```
 1  for (ii= 0; ((ii< nbrows) && !singular); ii++)
 2  {
 3    For_OOCMatrix_Row( LL, ii, rowIL, READ_WRITE )
 4    {
 5      For_OOCMatrix_Row( UU, ii, rowIU, READ_WRITE )
 6      {
 7        for (kk= 0; kk< rowIL.diag; kk++)
 8          wa[rowIL.id[kk]] = rowIL.val[kk];
 9        for (kk= rowIU.diag; kk< rowIU.nz; kk++)
10          wa[rowIU.id[kk]] = rowIU.val[kk];
11        for (kk= 0; kk< rowIL.diag; kk++)
12        {
13          jj = rowIL.id[kk];
14          For_OOCMatrix_Row( UU, jj, rowJU, READ )
15          {
16            pivot = wa[jj] * rowJU.invd;
17            wa[jj] = pivot;
18            for (ll= rowJU.diag+1; ll< rowJU.nz; ll++)
19            {
20              col = rowJU.id[ll];
21              wa[col] = wa[col] - pivot * rowJU.val[ll];
22            }
23          }
24        }
25        rowIL.invd = 1.0;
26        pivot = wa[ii];
27        if (pivot != 0.0)
28        {
29          rowIU.invd = 1.0 / pivot;
30          for (kk= 0; kk< rowIL.diag; kk++)
31            rowIL.val[kk] = wa[rowIL.id[kk]];
32          for (kk= rowIU.diag; kk< rowIU.nz; kk++)
33            rowIU.val[kk] = wa[rowIU.id[kk]];
34          range = range + 1;
35        }
36        else
37        {
38          WARNING( "Matriz singular. Se requiere pivoteo" );
39          singular = CIERTO;
40          /* operaciones varias */
41        }
42      }
43    }
44  }
```

Figura 27: Factorización numérica LU.

3.3.3. Factorización LDL^T

Desde el punto de vista de implantación de la capa *out-of-core* podemos ver que el código de la factorización LDL^T presenta mucha similitud con el caso ya tratado para la factorización *Cholesky* en el sentido que la función retorna un factor L, pero adicionalmente devuelve un vector denso D, que contiene los elementos de la diagonal. Para efecto del funcionamiento de la capa *out-of-core*, el código de la factorización simbólica mostrado en la figura 28 es muy similar al código de la figura 24 puesto que se tienen dos macros `For_OOCMatrix_Row`, la externa lee desde la matriz de entrada `AA` y la interna escribe en la matriz `LL`.

```
 1  for (ii= 0; ii< nbrows; ii++)
 2  {
 3    maxCol = -1; minCol = nbrows;
 4    For_OOCMatrix_Row( AA, ii, rowIA, READ )
 5    {
 6      for (jj= 0; jj< rowIA.diag; jj++)
 7      { /* Lazo en la vecindad sin el elem diagonal */
 8        col = rowIA.id[jj];
 9        if (seen[col] != ii)
10        {
11          seen[col] = ii;
12          minCol = MIN( minCol, col );
13          maxCol = MAX( maxCol, col );
14          col = first[col];
15          /* Calcula conjunto alcanzable desde L */
16        }
17      }
18      For_OOCMatrix_Row( LL, ii, rowIL, WRITE )
19      {
20        rowIL.invd = 1.0;
21        /* Lazo sobre los elem no nulos de G sin la diagonal
22           Los elem se alamcenan en orden ascend por columna */
23        for (nxt = 0, jj= minCol; jj<= maxCol; jj++)
24        {
25          if (seen[jj] == ii)
26          {
27            rowIL.id[nxt] = jj;
28            nxt = nxt + 1;
29            if (first[jj] == NIL)
30              first[jj] = ii;
31          }
32        }
33        /* Agrega elem diagonal */
34        /* Reajusta espacio para los valores */
35        /* Inicia los valores desde la matriz A */
36      }
37    }
38  }
```

Figura 28: Factorización simbólica LDL^T.

En referencia al código para la factorización numérica LDL^T podemos apreciar en la figura 29 que su estructura es muy similar a la ya tratada factorización *Cholesky* numérica (ver figura 25). Según vemos en las líneas 4 y 9 de la figura 29 las macros que acceden a la matriz `LL` son idénticas en su anidamiento a las macros que acceden a la matrix `GG` en la figura 25. Los modos de acceso tanto para la macro externa como la macro interna que acceden a la matriz `LL` son también iguales. La única diferencia de este código es el acceso a la matriz diagonal `DD`, la cual como ya se comentó, se maneja con un vector denso y no afecta el desempeño de la capa *out-of-core*. La optimización experimental que se aplicó a la matriz `GG` de la factorización *Cholesky* para reducir el número de *fallos* también se usó en este caso para la correspondiente matrix `LL`. Por eso vemos que en la línea 1 de la figura 29 se establece en 2 el número de vias de la matriz LL.

```
 1  TDMatrixSetVias( LL, 2 );
 2  for (ii= 0; ((ii< nbrows) && !singular); ii++)
 3  {
 4    For_OOCMatrix_Row( LL, ii, rowI, READ_WRITE )
 5    {
 6      for (kk= 0; kk< rowI.diag; kk++)
 7      { /* para los elem antes de la diag en la fila ii */
 8        jj = rowI.id[kk];
 9        For_OOCMatrix_Row( LL, jj, rowJ, ACCESS_READ )
10        {
11          nnz = rowJ.nz;
12          rowJ.nz = rowJ.diag;
13          TDSparseVecWDot( &rowJ, &rowI, diag, &tt );
14          rowJ.nz = nnz;
15        }
16        rowI.val[kk] = (rowI.val[kk] - tt) * invdiag[jj];
17      }
18      /* The diag. element in the row i-th */
19      nnz = rowI.nz; rowI.nz = rowI.diag;
20      TDSparseVecWDot( &rowI, &rowI, diag, &tt );
21      rowI.nz = nnz;
22      tt = diag[ii] - tt;
23      diag[ii] = tt;
24      if (tt != 0.0)
25      {
26        invdiag[ii] = 1.0 / tt;
27        range = range + 1;
28      }
29      else
30      {
31        WARNING( "Matriz Singular. Se requiere Pivoteo" );
32        singular = CIERTO;
33        rowI.val[rowI.diag] = 0.0;
34      }
35    }
36    For_OOCMatrix_Row( DD, ii, rowI, WRITE )
37    { /* Actualizar matriz D */
38      rowI.val[0] = diag[ii];
39    }
40  }
```

Figura 29: Factorización numérica LDL^T.

3.3.4. Resultados con métodos directos

Para evaluar el soporte *out-of-core* para los métodos directos de factorización de matrices, los códigos se compilaron usando el compilador `gcc` versión 4.3.4 (`x86_64`) con las siguientes banderas de optimización: `-O2`, `-funroll-loops`, `-fprefetch-loop-arrays`, y éstos fueron ejecutados sobre un computador tipo portátil con el sistema operativo `GNU/Linux`, kernel `2.6.30`. El computador de prueba tenía las siguientes características: procesador Intel Core 2 Duo de 2.4GHz P8600™ dual-core con 4GB de RAM. Durante todas las pruebas la memoria de intercambio de 8GB estuvo habilitada (`swapon`). Los archivos temporales en disco se almacenaron en una partición usando el sistema de archivos `ext3`.

Matrices de prueba. Para conocer el desempeño de la capa *out-of-core* se seleccionó un conjunto de matrices (ver tabla 7). Cada una de esas matrices se tomó de ejemplos prácticos y todas ellas están disponibles en la Colección de Matrices dispersas de la Universidad de La Florida (Davis y Hu, 2010). Este conjunto de matrices se seleccionó desde (Rozin y Toledo, 2005) y (Reid y Scott, 2009), tomando en consideración aspectos como: tipo de matriz: simétrica, positiva definida que sirva como entrada para todos los tres métodos de factorización, disponibilidad de las matrices de prueba, patrón de localidad de referencias, densidad de matrices de entrada y de salida, campo de aplicación y no exceder la cantidad de memoria física del computador de prueba (4 GB), en la mayor parte de las matrices de prueba. Este último aspecto fue importante para obtener la solución *in-core* del sistema lineal $Ax = b$. Para probar si la solución del algoritmo de factorización fue correcta, se usaron los factores para resolver el sistema lineal. En el caso de la factorización *Cholesky* si el sistema a resolver es: $Ax = b$, entonces $A = LL^T$, $Ly = b$, $L^T x = y$. Para la factorización LU si el sistema a resolver es: $Ax = b$, entonces $Ax = LUx = b$, primero resolvemos $Ly = b$ para y, y luego $Ux = y$ para x. Para la factorización LDL^T, si el

Tabla 7: Matrices de prueba para métodos directos.

Identificador	n	$nz(A)$	*Den.* %	**Aplicación/Descripción**
1. `vanbody`	47,1	2,3	0,11	Problema estructural
2. `oilpan`	73,8	3,6	0,07	Problema estructural
3. `thread`	29,7	4,5	0,51	Conector/contactor trenzado
4. `bmw7st_1`	141,3	7,3	0,04	Problema estructural
5. `x104`	108,4	10,2	0,09	Problema estructural
6. `m_t1`	97,6	9,8	0,10	Junta tubular
7. `crankseg_1`	52,8	10,6	0,38	Análisis lineal estático
8. `shipsec8`	114,9	6,5	0,05	Sección de barco
9. `cfd2`	123,4	3,1	0,02	Matriz de presiones CFD
10. `shipsec1`	140,9	7,8	0,04	Sección de barco
11. `nd6k`	18,0	6,9	2,13	Problema de malla 3D
12. `crankseg_2`	63,8	14,1	0,35	Análisis lineal estático
13. `pwtk`	217,9	11,4	0,02	Tunel de viento presurizado
14. `thermal2`	1228,0	8,6	0,00	Térmico no-estructurado FEM
15. `shipsec5`	179,9	10,1	0,03	Sección de barco
16. `msdoor`	415,9	20,2	0,01	Problema estructural
17. `ship_003`	121,7	8,1	0,05	Ship structure-production
18. `bmwcra_1`	148,8	10,6	0,05	Modelo de cigüeñal automotriz
19. `af_shell3`	504,9	17,6	0,01	Matrix de form. de hoja metálica
20. `g3_circuit`	1585,5	7,7	0,00	Simulación de circuito
21. `nd12k`	36,0	14,2	1,10	Problema de malla 3D
22. `ldoor`	952,2	46,5	0,01	Puerta grande
23. `inline_1`	503,7	36,8	0,01	Patinador en línea

sistema a resolver es: $Ax = b$, entonces $Ax = LDL^T x = b$, $Lz = b$, $Dy = z$, $L^T x = y$. En todos los casos, el vector del lado derecho b se obtuvo artificialmente asumiendo que la solución del sistema lineal es el vector $(1, 1, \ldots, 1)^t$.

La tabla 7 presenta el conjunto de matrices de prueba. La primera columna tiene un número que será usado en las tablas y figuras subsecuentes y un identificador con el nombre de la matriz, n denota el orden de la matriz en miles, *nz(A)* denota el número de elementos no nulos en millones de la matriz de entrada, la columna *Den %* muestra la densidad de la matriz expresada como un porcentaje del total de elementos no nulos entre el total de elementos de la matriz ($\frac{nz(A)*100}{n^2}$).

Proceso de prueba. Como se discutió en la sección §3.3.1, el sistema lineal $Ax = b$ se resuelve para cada una de las matrices de prueba. La capa *out-of-core* para todas las factorizaciones (*Cholesky*, LU, LDL^T), solapa el cómputo con la entrada/salida, para compensar las latencias de memoria/disco. Para evaluar el comportamiento de las técnicas de prebúsqueda implementadas para el proceso de factorización, el proceso de prueba se llevó a cabo en tres fases:

a Solución *in-core* del sistema lineal $Ax = b$ con la capa *out-of-core* desabilitada. En esta etapa la matriz de entrada A se carga en memoria física y se factoriza usando en proceso de cuatro pasos descrito en la sección §3.3.1.

b Solución del sistema lineal $Ax = b$, con la capa *out-of-core* habilitada y sin usar técnicas de prebúsqueda. En esta etapa, no hay prebúsqueda de *nodos* desde el archivo temporal al caché mientras que se realiza el cómputo, así que la matriz A se carga en la estructura *out-of-core* y se factoriza usando el proceso de cuatro pasos descrito en la sección §3.3.1. Note que durante la carga y factorización de la matriz se usa la ORL y no hay solapamiento entre cómputo y entrada/salida (ver sección §2.8.3). Aunque se usa la ORL, hay un *overhead* ocasionado por la ausencia de prebúsqueda. Cuando sucede un *fallo*, el *nodo* que no está en caché (por la ausencia de prebúsqueda) se debe obtener desde el archivo temporal, causando un *overhead* por el tiempo adicional que se necesita para leerlo. El hilo principal que ejecuta el cómputo debe esperar hasta que el segundo hilo obtiene el *nodo*. Aun cuando esta etapa, con prebúsqueda desabilitada, se puede realizar usando un solo hilo, cuando se diseñaron los algoritmos de la capa *out-of-core*, todas las operaciones de entrada/salida se concentraron en el segundo hilo con el objeto de tener la entrada/salida exclusivamente en el segundo hilo. Esto permitió tener códigos multihilos robustos y probados para la prebúsqueda habilitada. Se considera que con este esquema se puede apreciar mejor el efecto de la prebúsqueda y el *overhead* que se genera al usar un segundo hilo en esta etapa es muy pequeño.

c Se resuelve sistema lineal $Ax = b$, con el núcleo *out-of-core* y la prebúsqueda habilitados. En esta etapa se carga la matriz A en la estructura *out-of-core* y

se factoriza usando el proceso de cuatro pasos mencionado en la sección §3.3.1. En esta fase los procesos de carga y factorización se ejecutan usando técnicas de prebúsqueda (ver sección §2.8.1), las cuales cargan en forma temprana en caché los *nodos* antes que sucedan los *fallos*. En esta etapa, tanto la ORL como las técnicas de prebúsqueda están activas, tomando ventaja de los procesadores multi-core.

En cada una de las fases se midieron las siguientes variables: el tiempo que toma la suma de las factorizaciones simbólica y numérica, la cantidad de memoria necesaria para el proceso de factorización completo (ver sección §3.3.1). En todos los casos el tiempo se midió usando PAPI (*Performance Application Programming Interface*) version 3.7.0 (Terpstra, 2009). Se usaron las funciones internas que provee la biblioteca `UCSparseLib` para medir la cantidad de memoria utilizada.

Antes de la prueba final, se resolvió cada una de las matrices de entrada usando diferentes configuraciones de caché; durante las pruebas se observó el tiempo total de factorización. De acuerdo a los resultados, se seleccionó la configuración de caché mas conveniente. Experimentalmente se determinó que el caché debía ser al menos asociativo por conjuntos de 2 vías durante el proceso de factorización, para obtener un desempeño eficiente en el anidamiento de las macros según se discutió en la sección §2.8.1. Por defecto se estableció un caché de correspondencia directa, porque es el mas rápido en tazas de *aciertos*, cambiándolo dinámicamente a asociativo por conjuntos de 2 vías solamente durante el proceso de factorización.

En referencia al tamaño del *bloque* de caché se observó que el tiempo de ejecución no decrece significativamente incrementando el número de *bloques* en caché; por lo tanto el caché se configuró con un tamaño de un simple *bloque*. El tamaño, en *nodos*, del *bloque* (vectores dispersos) fue variable y se seleccionó por cada matriz de entrada para obtener el mejor tiempo de ejecución. El tamaño mínimo seleccionado para el *bloque* fue 2^1 *nodos* y el máximo 2^6 *nodos*.

Experimentalmente para la factorización *Cholesky*, se encontró que de las 23 matrices (ver tabla 7), solamente los números 7, 10 y 11 tienen un tamaño de *bloque* de 2^2 *nodos*; las matrices 2 y 14 tienen un tamaño de *bloque* de 2^4 *nodos*; la matriz 20 tiene un tamaño de *bloque* de 2^5 *nodos* y el resto de las 17 matrices tienen un tamaño de *bloque* de 2^3 *nodos*. Para la factorización LU, se encontró que de las 23 matrices (ver tabla 7), solamente los números 11 y 21 tienen un tamaño de *bloque* de 2^2 *nodos*; las matrices 3, 5, 7, 8, 10, 12, 15, 17 y 18 tienen un tamaño de *bloque* de 2^3 *nodos*; las matrices 2, 5, 6, 9, 13, 19, 20, 22 y 23 tienen un tamaño de *bloque* de 2^4 *nodos*; las matrices 4, 14 y 16 tienen un tamaño de *bloque* de 2^5 *nodos* y la matriz 1 tiene un tamaño de *bloque* de 2^6 *nodos*. Cada prueba se repitió tres veces y en cada caso se tomó el menor tiempo para decidir el tamaño de *bloque* apropiado y también para obtener los resultados en la tabla 8.

Tabla 8: Uso de memoria y tiempo de ejecución para factorización *Cholesky*.

Identificador	**in-core**		**out-of-core**			
			sin prebúsqueda		**con prebúsqueda**	
	Mem.	**Tiempo**	**Mem.**	**Tiempo**	**Mem.**	**Tiempo**
1. vanbody	129,38	7,960	27,56	24,818	27,20	20,518
2. oilpan	198,15	16,051	42,06	50,171	41,81	39,309
3. thread	385,85	142,720	52,03	404,539	51,47	259,432
4. bmw7st_1	476,34	56,788	85,44	175,954	85,27	133,533
5. x104	551,69	73,880	117,48	220,953	117,34	160,859
6. m_t1	616,81	102,062	112,60	309,808	112,57	213,580
7. crankseg_1	633,37	145,013	122,55	425,200	122,12	284,673
8. shipsec8	578,84	176,863	77,12	530,829	77,25	356,285
9. cfd2	533,44	163,150	36,34	485,839	36,52	339,546
10. shipsec1	644,33	179,537	90,76	538,247	90,77	364,758
11. nd6k	612,60	380,129	79,78	1063,054	79,16	673,987
12. crankseg_2	824,65	200,108	163,17	582,973	162,70	389,281
13. pwtk	851,76	128,324	135,09	399,309	135,24	285,695
14. thermal2	959,28	88,189	107,69	270,589	107,86	230,959
15. shipsec5	856,76	272,295	117,38	803,836	117,80	564,972
16. msdoor	1127,52	105,751	235,20	341,260	235,22	267,684
17. ship_003	888,07	367,461	94,00	1085,748	93,94	735,072
18. bmwcra_1	1099,32	354,557	123,14	1058,603	123,26	725,625
19. af_shell3	1538,32	284,005	205,28	919,106	206,11	659,977
20. g3_circuit	1513,19	315,526	104,03	983,680	100,53	773,094
21. nd12k	1632,81	1727,944	163,75	4916,600	163,31	3326,169
22. ldoor	2858,51	475,749	539,92	1514,646	541,50	1117,924
23. inline_1	2891,23	778,859	425,85	2412,318	426,21	1734,658

Mem.: Memoria expresada en Megabytes. **Tiempo** está en segundos.

Los resultados mostrados en las tablas 8, 9 y 10 tienen la misma estructura. La tabla 8 presenta los requerimientos de memoria y el tiempo de ejecución para la factorización *Cholesky*, la tabla 9 muestra los mismos resultados para la factorización LU y la tabla 10 muestra los resultados para la factorización LDL^T. En cada tabla, la primera columna presenta un identificador numérico seguido por el nombre de la matriz. Las siguientes columnas muestran los requerimientos de memoria y el tiempo de ejecución para los ajustes para los distintos experimentos. Esto es, con la capa *out-of-core* desabilitada, con la capa *out-of-core* habilitada pero sin prebúsqueda y finalmente con la capa *out-of-core* y la prebúsqueda habilitadas.

En las figuras 30, 32 y 34 se muestra el comportamiento del tiempo de ejecución cuando se factorizan la matrices de prueba usando los métodos de factorización *Cholesky*, LU y LDL^T. En cada una de las figuras se grafica la relación entre los tiempos

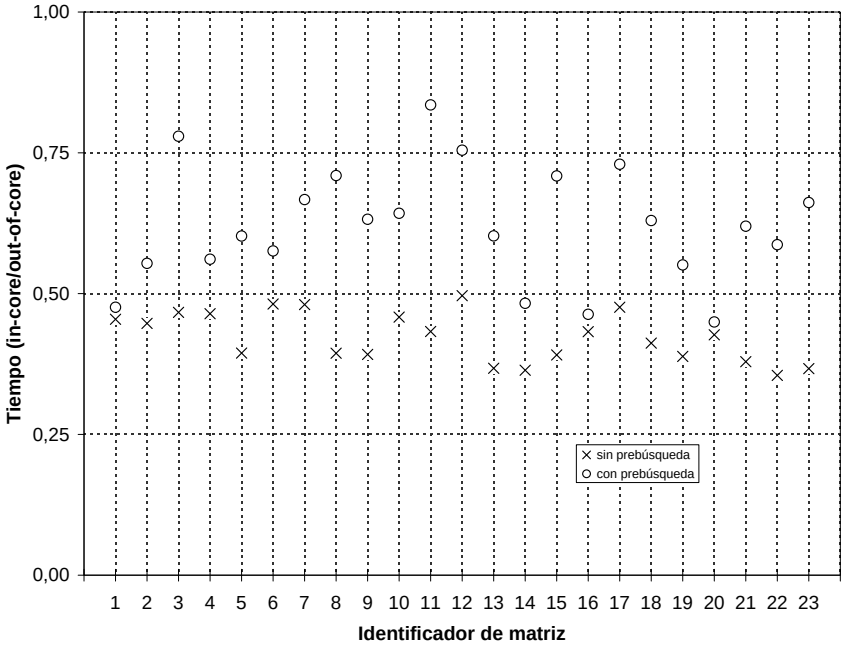


Figura 30: Relación de tiempo $\frac{in-core}{out-of-core}$ para la factorización *Cholesky*.

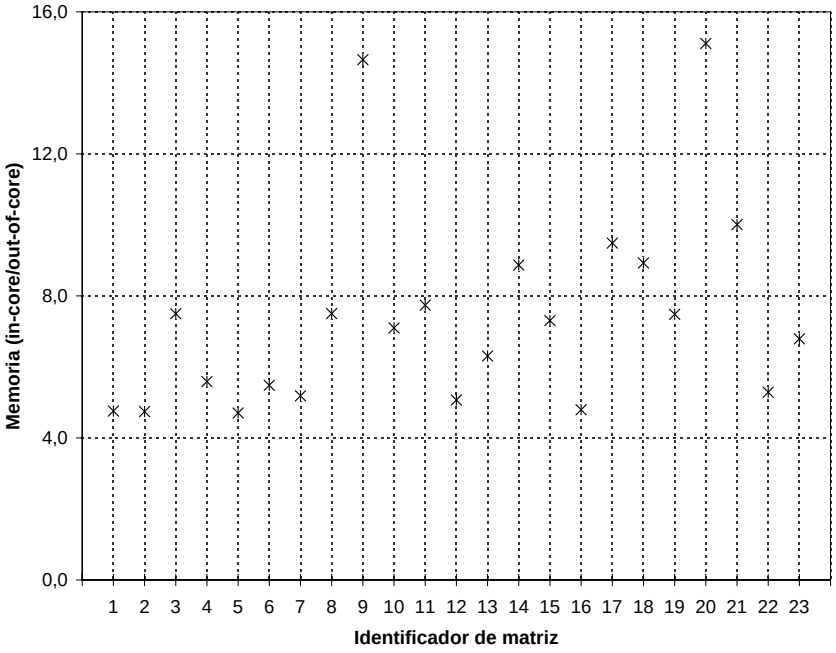


Figura 31: Relación de memoria $\frac{in-core}{out-of-core}$ para la factorización *Cholesky*.

Tabla 9: Uso de memoria y tiempo de ejecución para factorización LU.

Identificador	**in-core**		**out-of-core**			
			sin prebúsqueda		**con prebúsqueda**	
	Mem.	**Tiempo**	**Mem.**	**Tiempo**	**Mem.**	**Tiempo**
1. `vanbody`	200,86	4,162	27,25	15,529	27,20	12,005
2. `oilpan`	306,99	8,044	41,82	28,633	41,79	22,837
3. `thread`	666,66	83,825	51,45	251,125	51,45	149,119
4. `bmw7st_1`	771,31	27,691	85,22	100,737	85,22	75,038
5. `x104`	860,52	33,022	117,32	112,822	117,32	81,755
6. `m_t1`	1.001,15	49,304	112,52	166,442	112,48	114,653
7. `crankseg_1`	1.018,84	76,297	122,25	230,404	122,22	153,613
8. `shipsec8`	994,51	87,562	77,11	271,840	77,15	177,714
9. `cfd2`	984,64	82,575	36,35	269,271	36,35	158.980
10. `shipsec1`	1.096,64	89,048	90,60	274,711	90,65	179,126
11. `nd6k`	1.065,69	256,517	79,18	657,120	79,15	412,613
12. `crankseg_2`	1.319,46	102,820	162,95	316,573	162,84	201,817
13. `pwtk`	1.416,65	58,273	134,93	203,169	134,95	140,499
14. `thermal2`	959,28	77,577	107,72	169,479	107,86	128,483
15. `shipsec5`	1.465,01	135,413	117,23	413,023	117,30	270,991
16. `msdoor`	1.752,64	48,995	234,96	183,986	234,96	142,973
17. `ship_003`	1.579,71	190,215	93,57	553,314	93,61	351,568
18. `bmwcra_1`	1.941,04	168,957	123,07	511,825	123,14	329,856
19. `af_shell3`	1.538,32	132,672	205,29	456,710	205,14	317,500
20. `g3_circuit`	2.701,02	211,424	100,15	561,139	100,53	396,794
21. `nd12k`	2.936,83	1.106,180	163,14	3.011,547	163,14	1.853,594
22. `ldoor`	4.562,47	560,240	540,15	842,194	540,15	612,159
23. `inline_1`	4.892,71	1.717,853	425,70	1.255,594	425,70	848,404

Mem.: Memoria expresada en Megabytes; **Tiempo** está en segundos;

$\frac{in\text{-}core}{out\text{-}of\text{-}core}$, con y sin la prebúsqueda activada. La relación de tiempos $\frac{in-core}{out-of-core}$ siempre es un número menor que uno dado que *tiempo in-core*<*tiempo out-of-core* porque hay un *overhead* ocasionado por el acceso al archivo temporal en disco cuando la capa *out-of-core* está habilitada.

Finalmente, las figuras 31, 33 y 35 muestran gráficamente la cantidad de memoria requerida para las distintas factorizaciones con cada una de las matrices de prueba. Para cada caso se grafica el cociente de las cantidades de memoria $\frac{in-core}{out-of-core}$ con la capa desactivada y activada respectivamente. La relación $\frac{Memoria\ in-core}{Memoria\ out-of-core}$ siempre es un número mayor que uno porque en caché se almacena solamente un subconjunto de filas/columnas de la matrices necesarias para resolver la factorización cuando la capa *out-of-core* está habilitada mientras que en el otro caso se almacenan en memoria

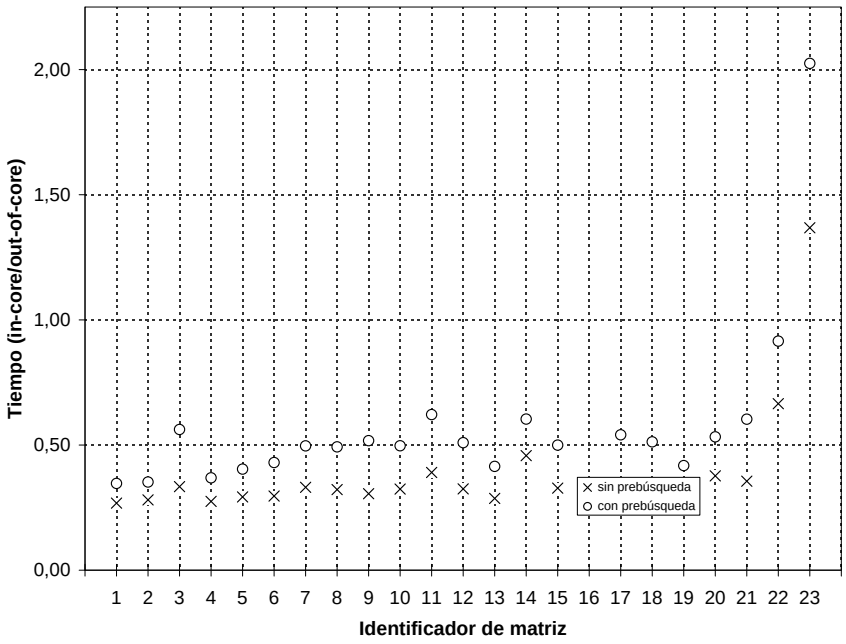


Figura 32: Relación de tiempo $\frac{in-core}{out-of-core}$ para la factorización LU.

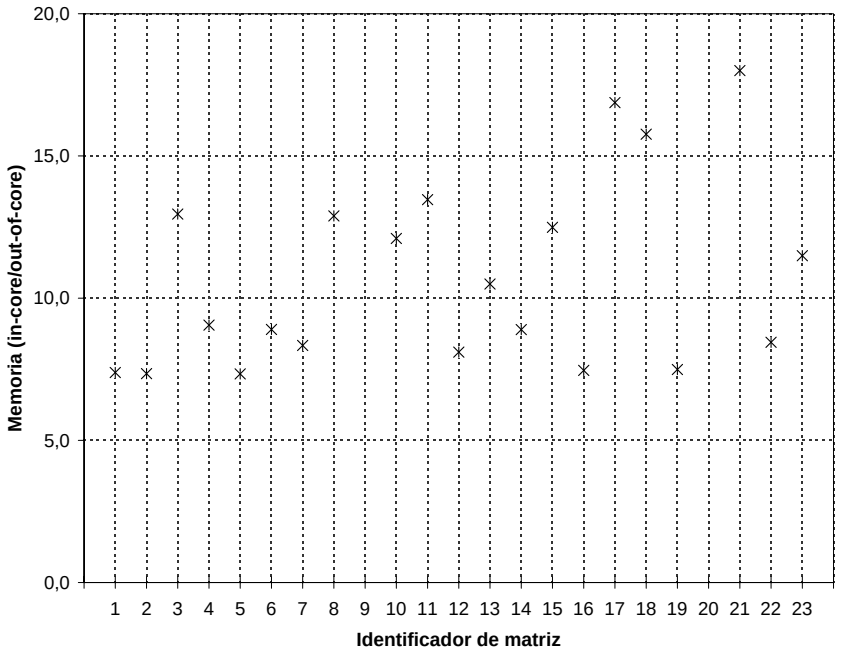


Figura 33: Relación de memoria $\frac{in-core}{out-of-core}$ para la factorización LU.

física la totalidad de las matrices.

Análisis de resultados factorización *Cholesky*. Para el análisis de resultados se usarán las figuras 30 y 31. En la figura 30 se puede ver la relación de tiempos $\frac{in-core}{out-of-core}$ para la factorización *Cholesky*. Cuando se observa en la figura el comportamiento del tiempo de ejecución con la prebúsqueda desactivada, la relación de tiempos tiende a un valor de 0,42. Cuando se activa la prebúsqueda, se nota que hay un grupo de matrices con una relación excelente de alrededor de 0,75 la cual significa que el *overhead* se reduce en una proporción muy importante debido al algoritmo de prebúsqueda. Este valor muestra que el *overhead* es menor que una tercera parte del tiempo de ejecución *in-core* para este grupo de matrices. Las matrices 3, 11 y 12 pertenecen a este grupo. Su característica común es que todas ellas tienen una alta densidad que les permite explotar mejor el solapamiento entre cálculo y entrada/salida. Hay también un segundo grupo de matrices (1, 14, 16 y 20) cuyo relación de tiempos es menor de 0,5. Este grupo tiene como característica común su baja densidad y su patrón de dispersión. En este caso, el algoritmo de prebúsqueda no resulta ser tan efectivo como en el caso anterior y se refleja en su alto *overhead* en tiempo de ejecución, comparado con el caso anterior.

La figura 31 muestra el comportamiento de la memoria para la capa *out-of-core*. Se presenta un gráfico sencillo para el uso de memoria porque no hay variación en los requerimientos de memoria con la activación del algoritmo de prebúsqueda, porque para la factorización *Cholesky* el algoritmo de prebúsqueda no necesita modificar la organización del caché apreciablemente.

Análisis de resultados factorización *LU*. Para el análisis de resultados se usarán las figuras 32 y 33. En la figura 32 se puede ver la relación de tiempos $\frac{in-core}{out-of-core}$ para la factorización LU. Cuando se observa en la figura el comportamiento del tiempo de ejecución con la prebúsqueda desactivada, la relación de tiempos tiende a un valor de 0,32. Cuando se activa la prebúsqueda, se nota que hay un grupo de matrices con una relación excelente de alrededor de 0,5 la cual significa que el *overhead* se reduce en una proporción importante debido al algoritmo de prebúsqueda. Llama la atención en este grupo las matrices 22 y 23 cuya relación está por encima de 0,9. En estas matrices la solución *in-core* necesita mas de 4GB de RAM, según de ve en la tabla 9. En la matriz 23, el *overhead* de la memoria virtual supera el *overhead* de la capa *out-of-core*, de allí que el tiempo de solución con la capa habilitada se reduce a menos de la mitad. En la matriz 22, el *overhead* de la memoria virtual no supera el *overhead* de la capa *out-of-core* porque a diferencia de la matriz 23 la matriz 22 es mas esparcida y la capa *out-of-core* no puede aprovechar el principio de localidad espacial del caché como en la matriz 23. De todas formas se nota que el tiempo, con la capa *out-of-core* habilitada, está muy cercano al tiempo de solución *in-core* (0,9) a diferencia del resto de matrices, cuyo tiempo de solución *out-of-core* está alrededor de

la mitad (0.5) del tiempo *in-core*, debido a la contribución del *overhead* de la memoria virtual.

La figura 33 muestra el comportamiento de la memoria para la capa *out-of-core*. Se presenta un gráfico sencillo para el uso de memoria porque no hay variación en los requerimientos de memoria con la activación del algoritmo de prebúsqueda, porque para la factorización LU el algoritmo de prebúsqueda no necesita modificar la organización del caché apreciablemente.

Análisis de resultados factorización LDL^T. Para el análisis de resultados se usarán las figuras 34 y 35. En la figura 34 se puede ver la relación de tiempos $\frac{in-core}{out-of-core}$ para la factorización LDL^T. Cuando se observa en la figura el comportamiento del tiempo de ejecución con la prebúsqueda desactivada, la relación de tiempos tiende a un valor de 0,32 el cual está por debajo de la factorización *Cholesky* y de la factorización LU. Cuando se activa la prebúsqueda, se nota que en todas las matrices hay una mejora en la relación de tiempos lo cual significa que la prebúsqueda atenúa efectivamente las latencias de memoria y disco duro. Llama la atención, al igual que en la factorización LU, que hay dos matrices (22 y 23) cuya relación de tiempos, con y sin prebúsqueda activada, está por encima de 1,4. En estas matrices, la solución *in-core* requiere mas de 4GB de RAM según de se ve en la tabla 10.

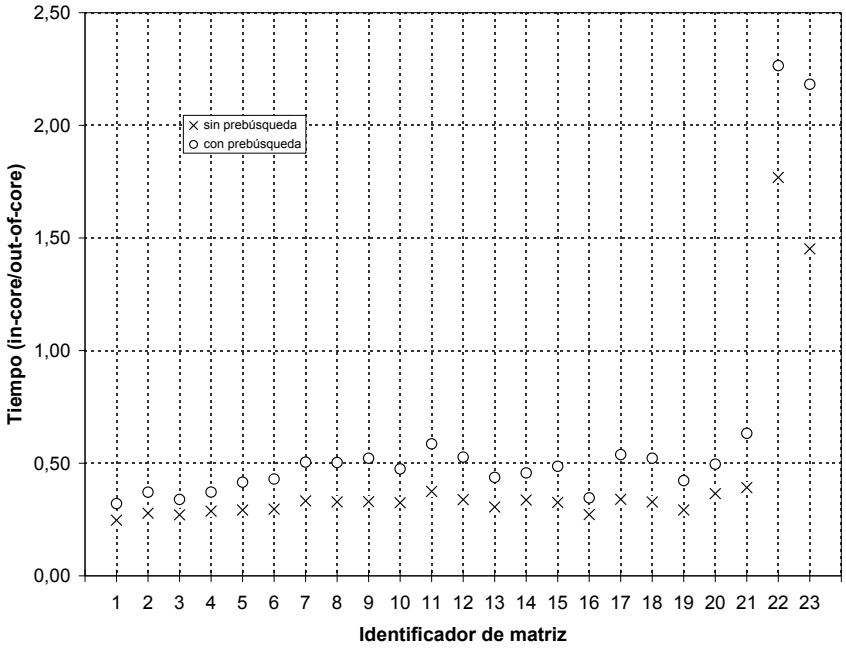


Figura 34: Relación de tiempo $\frac{in-core}{out-of-core}$ para la factorización LDL^T.

En estas dos matrices, el *overhead* de la memoria virtual supera el *overhead* de la

Tabla 10: Uso de memoria y tiempo de ejecución para factorización LDL^T.

Identificador	**in-core**		**out-of-core**			
			sin prebúsqueda		**con prebúsqueda**	
	Mem.	**Tiempo**	**Mem.**	**Tiempo**	**Mem.**	**Tiempo**
1. vanbody	200,86	3,905	27,25	15,815	27,20	12,171
2. oilpan	306,99	7,817	41,79	28,099	41,79	21,034
3. thread	1.752,64	49,456	234,96	182,595	235,04	145,806
4. bmw7st_1	771,31	27,039	85,22	94,122	85,22	72,717
5. x104	860,52	33,270	117,36	113,877	117,32	80,008
6. m_t1	1.001,15	49,304	112,52	166,442	112,48	114,653
7. crankseg_1	1.018,84	75,843	122,25	227,844	122,22	150,222
8. shipsec8	994,51	90,332	77,11	275,139	77,15	179,506
9. cfd2	984,64	82,564	36,35	250,694	36,35	158,102
10. shipsec1	1.096,64	88,608	90,60	273,137	90,65	186,585
11. nd6k	1.065,69	248,888	79,18	663,762	79,15	424,721
12. crankseg_2	1.319,46	104,520	162,95	308,918	162,84	198,228
13. pwtk	1.416,65	57,847	134,93	189,204	134,95	132,302
14. thermal2	1.607,25	56,301	107,86	167,261	107,86	123,199
15. shipsec5	1.465,01	132,750	117,23	407,270	117,30	272,941
16. msdoor	1.752,64	49,594	234,96	182,315	234,96	143,246
17. ship_003	1.579,71	190,608	93,57	560,979	93,61	354,119
18. bmwcra_1	1.941,04	171,014	123,07	521,174	123,07	327,429
19. af_shell3	2.626,14	137,545	205,40	470,482	205,40	325,252
20. g3_circuit	2.701,02	192,177	100,15	526,197	100,53	387,514
21. nd12k	2.936,83	1.121,911	163,14	2.863,531	163,14	1.773,229
22. ldoor	4.562,47	1.481,607	540,15	837,703	540,15	654,089
23. inline_1	4.892,71	1.818,321	425,70	1.252,459	425,70	833,236

Mem.: Memoria expresada en Megabytes; **Tiempo** está en segundos;

capa *out-of-core*; de allí que el tiempo de solución con la capa habilitada se reduce a menos de la mitad. En estos casos se ve la ventaja del uso de la capa *out-of-core*, cuyo desempeño es mucho mejor al de la memoria virtual.

La figura 35 muestra el comportamiento de la memoria para la capa *out-of-core* con la factorización LDL^T. Se presenta un gráfico sencillo para el uso de memoria porque no hay variación en los requerimientos de memoria con la activación del algoritmo de prebúsqueda, porque para la factorización LDL^T el algoritmo de prebúsqueda no necesita modificar la organización del caché apreciablemente.

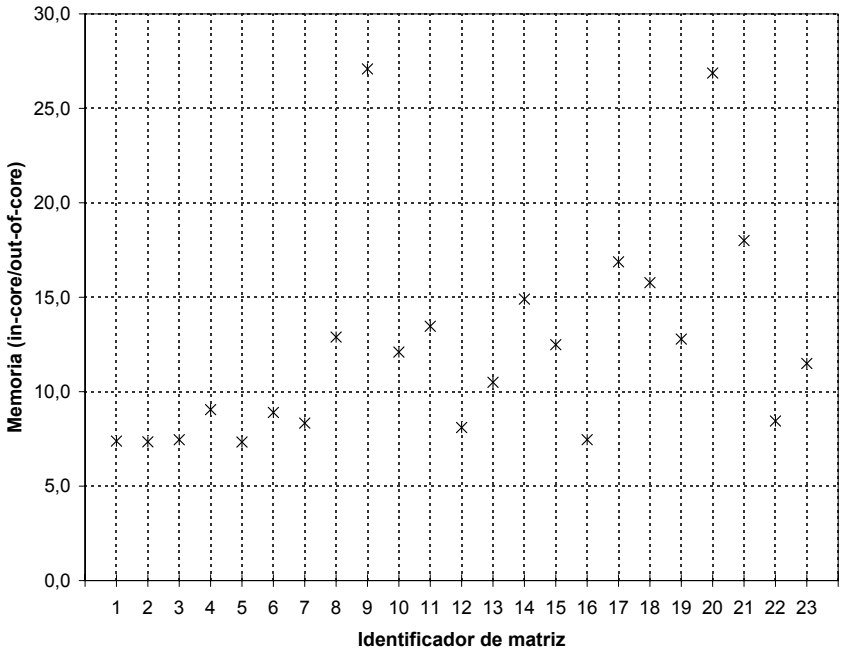


Figura 35: Relación de memoria $\frac{in-core}{out-of-core}$ para la factorización LDL^T.

3.4. Métodos multinivel algebraico

Típicamente cuando un sistema es suficientemente grande se necesita un método de solución cuyo tiempo para obtener el resultado sea proporcional al tamaño del problema. Esta característica se ha alcanzado con los métodos multinivel (MG) (Brandt, 1977). Otros métodos, tales como los métodos iterativos clásicos, presentan una complejidad cuadrática. Pero, a diferencia de otros métodos, los métodos multinivel consumen mas memoria porque su estrategia de solución implica mantener en memoria un conjunto de matrices, a diferencia de los otros métodos. Así, para los métodos MG los requerimientos de memoria se incrementan rápidamente con el tamaño de la matriz de entrada A (Castellanos y Larrazábal, 2012).

En (X. Shi y Zhou, 2009) se presenta un *solver* multinivel algebraico para mallas serializadas (*streaming*) el cual exhibe excelentes ahorros en memoria; sin embargo, el proceso de solución requiere de un paso de pre-procesamiento en el cual las mallas de entrada se serializan por medio de un secuenciamiento espectral. Este paso de pre-procesamiento incrementa notablemente el tiempo total de solución del sistema lineal. En (Feng y Li, 2008) se presenta un *solver* multinivel implementado sobre GPU (*Graphic Processing Unit*); allí se obtiene una aceleración excelente del tiempo de ejecución pero el balance entre cómputo y acceso de memoria es un aspecto de cuidado que afecta el rendimiento del solver. En este último artículo solamente se presentan soluciones *in-core*. En este trabajo, a diferencia de los anteriores, se implementa el

Solver Multinivel Algebraico como una parte del soporte *out-of-core* completo para la biblioteca `UCSparseLib` library (Larrazábal, 2004).

El método multinivel consiste de los siguientes elementos:

a Una secuencia de grillas con una matriz asociada a cada grilla.

b Operadores de transferencia entre gillas (Interpolador y restrictor).

c Un método iterativo clásico (Gauss-Seidel, Jacobi, SSOR, etc.), el cual se denomina relajador.

Para explicar el método multinivel, supongamos que existen dos grillas llamadas M^h (grilla fina) y M^H (grilla gruesa). A_h y A_H son matrices que se obtienen desde una discretización en M^h y M^H. Las dimensiones de A_h y A_H son $n \times n$ y $N \times N$, respectivamente. El problema es entonces resolver un sistema lineal en la grilla fina como sigue:

$$A_h u^h = f^h \tag{1}$$

donde u^h, $f^h \in \mathbb{R}^n$, se denominarán $V^h = \mathbb{R}^n$, y similarmente $V^H = \mathbb{R}^N$. Se deben construir dos operadores de transferencia entre V^h y V^H. Ellos son el operador de interpolación $I : V^H \to V^h$ y el operador de restricción $R : V^h \to V^H$. En los métodos multinivel algebraico (AMG) A_H se define como la ecuación 2. Análogamente, $V^H = \mathbb{R}^N$.

$$A_H = R A_h I \tag{2}$$

Las matrices A_h and A_H deben ser al menos débilmente diagonal dominante.

3.4.1. El algoritmo multinivel algebraico

El método multinivel se basa en el algoritmo multinivel mostrado en la figura 3.4.1. Como el problema usualmente tiene mas de dos grillas, el paso c del algoritmo del algoritmo se aplica recursivamente hasta que el problema se reduce a una grilla suficientemente gruesa. Este algoritmo se conoce como el algoritmo *V-ciclo*. Existen diferentes algoritmos multinivel y su nombre depende de la forma en la cual se visitan los niveles.

La figura 37(a) muestra el esquema para el *V-ciclo* aunque hay otros esquemas usados como el *W-ciclo* (figura 37(b)) y el *F-ciclo* (figura 37(c)). Los métodos de relajación, tales como Jacobi o Gauss-Seidel, reducen efectivamente sólo ciertas componentes del error, en particular aquellas componentes asociadas con los autovalores del relajador que son cercanas a cero. Ellas se denominan componentes del error de alta frecuencia.

a Relajar v veces $A_h u^h = f^h$ en V^h, con la solución inicial u_0^h.

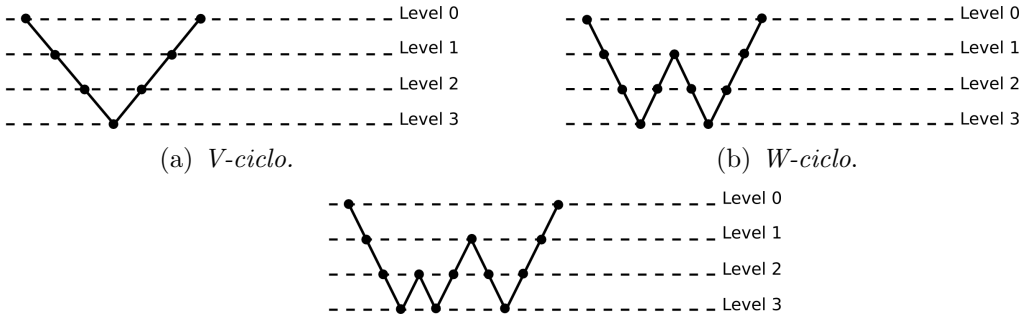
b Calcular $r^H = R(f^h - A_h u_v^h)$.

c Resolver $A_H e^H = r^H$ in V^H.

d Correjir la aproximación en la grid fina: $u_v^h = u_v^h + Ie^H$.

e Relajar μ veces $A_h u^h = f^h$ con solución inicial u_v^h

Figura 36: Algoritmo multinivel *V-ciclo*



(a) *V-ciclo.* (b) *W-ciclo.*

(c) *F-ciclo.*

Figura 37: Tipos de *ciclo* para multinivel algebraico.

3.4.2. El método multinivel algebraico

El método multinivel se compone de dos fases. En la primera fase o fase de inicialización, se construye una jerarquía multilevel de matrices. También se definen los operadores de transferencia. De acuerdo a la fase de inicialización, los AMG se dividen en dos grupos: métodos basados en técnicas de interpolación y métodos basados en técnicas de agregación. La diferencia entre esos métodos es la forma en la cual ellos construyen la partición del grafo de conectividad de la matriz de entrada y los operadores de transferencia. En nuestro caso se utilizó una técnica de agregación conocida como coloreado rojo/negro (Kickinger, 1997). La segunda fase o fase de solución consiste en la aplicación del algoritmo multinivel (ver figura 3.4.1) hasta que se alcanza un criterio de parada definido. En este trabajo se presentan los resultados para el *F-ciclo* aunque se probaron exitosamente el *V-ciclo* y el *W-ciclo*. El relajador usado para el *F-ciclo* fue el Gauss-Seidel.

Consideraciones de diseño para el soporte *out-ofcore*. Como se comentó en la sección §3.4, la implementación del *Solver* Multinivel Algebraico es parte del desarrollo de una capa de software que soportará todas las operaciones *out-of-core* para la biblioteca `UCSparseLib` (Larrazábal, 2004). Cuando se inició el *solver* multinivel

out-of-core, ya se había completado con el desarrollo del núcleo *out-of-core* para operaciones básicas con matrices dispersas (Castellanos y Larrazábal, 2007, 2008); tanto la operación transpuesta, como los productos matriz-vector y matriz-matriz tenían un soporte *out-of-core* eficiente para ese momento. También el núcleo *out-of-core* se había optimizado para soportar la factorización *Cholesky* (Castellanos y Larrazábal, 2011). Además una versión *in-core* del *solver* AMG se había desarrollado e incorporado en la biblioteca `UCSparseLib` library (Castellanos y Larrazábal, 2010).

La fase de inicialización del *solver* Multinivel está compuesta principalmente de productos matriz-matriz, operaciones transpuesta de matrices y el uso de un método directo en el nivel mas grueso (Castellanos y Larrazábal, 2010), por lo que el núcleo desarrollado funcionó bien sin mayores modificaciones. Pero apareció un requerimiento adicional para la capa *out-of-core*, porque durante la etapa de inicialización del AMG se genera una jerarquía de matrices. Cada una de las matrices generadas tiene diferente tamaño y necesita ser manipulada a través del caché que mantiene en memoria solamente un conjunto de las filas (columnas) de cada matriz. El caché para cada matriz tiene parámetros de configuración de acuerdo al número total de filas (columnas). Entonces fue necesario desarrollar una estrategia para seleccionar automáticamente la configuración del caché como una función del tamaño de la matriz. Como el método de direccionamiento usado por la capa *out-of-core* es función de 2^n, el valor 2^n que sigue al tamaño de la matriz en filas (columnas) determinó los parámetros del caché empezando en 2^{10} y finalizando en 2^{23}, esto es: $2^{10}, 2^{11}, .., 2^{22}, 2^{23}$. Así, se generaron artificialmente un conjunto de matrices, discretizando un operador escalar elíptico $3D$ por medio de un método de diferencias finitas de segundo orden 7-*stencil*. Después de probar cada una de las matrices para las operaciones producto matriz-matriz y transpuesta, con la capa *out-of-core* activada, se seleccionaron los mejores parámetros de caché que permitieron obtener un buen tiempo de ejecución usando una cantidad de memoria razonable. Todos los parámetros obtenidos experimentalmente se organizaron de forma que se pueden seleccionar desde una tabla, bajo control de programa, los valores de configuración del caché como una función del tamaño de la matriz.

La fase de solución del AMG se compone principalmente de productos matriz-vector. Para esta fase, todos los inconvenientes fueron aparentemente resueltos porque la capa *out-of-core* ya estaba preparada para calcular productos matriz-vector. Además, en este punto ya la capa era capaz de seleccionar los parámetros apropiados del caché según el tamaño de las matrices involucradas. Se observó que el *overhead*, en porcentaje de la capa *out-of-core* se incrementaba con el tamaño de la matriz de entrada. Para eliminar este *overhead* se modificó la capa *out-of-core* para tener un mejor solapamiento entre cómputo y operaciones de entrada/salida. Las modificaciones consistieron en usar un *mutex* para sincronizar las operaciones de entrada/salida de disco usando las funciones no re-entrantes `fread_unlocked()/fwrite_unlocked()`, en lugar de usar las funciones re-entrantes `fread()/fwrite()` que generan mayor detención usando un *mutex* del sistema que es común a todas las funciones de entra-

da/salida de disco.

3.4.3. Resultados con métodos multinivel

Para evaluar el soporte *out-of-core* del *solver* Multinivel, se compilaron todos los códigos usando el compilador `gcc` versión 4.6.1 (`x86_64`) sin banderas de optimización. Los programas de prueba se ejecutaron en un computador portátil de 2.4GHz, con procesador Intel Core 2 Duo P8600™ dual-core con 4GB de RAM con el sistema operativo `GNU/Linux`, kernel `3.0`. Los datos se almacenaron temporalmente en una partición de disco usando el sistema de archivos `ext3`. La memoria de intercambio de 8GB estuvo habilitada durante todas las pruebas (`swapon`).

Matrices de prueba. Para evaluar el *Solver* Multinivel *Out-of-core*, se generó un conjunto de 8 matrices (ver tabla 11). El orden de esas matrices va desde un mínimo de 343.000 ($70 \times 70 \times 70$) a un máximo de 2.744.000 ($140 \times 140 \times 140$) con el objeto de incluir un rango de al menos un orden de magnitud. Las matrices de entrada del *solver* se generaron discretizando un operador escalar elíptico $3D$ por medio de un método de diferencias finitas de segundo orden 7-*stencil*. Las matrices generadas caracterizan una gran variedad de problemas industriales y se definen como sigue:

$$L(u) = \Delta u + Cc\nabla \cdot u \tag{3}$$

Donde $Cc = 0$ (coeficiente de convección), con condiciones de frontera de Dirichlet y un cubo unitario como dominio computacional; esas matrices fueron probadas en (Larrazábal, 2002). Esto asegura que las matrices generadas son del tipo simétrica positiva definida. Las principales características de estas matrices se pueden observar en la tabla 11.

Para probar si la solución del sistema lineal fue correcta, se usó cada matriz de prueba como entrada A para resolver el sistema lineal $Ax = b$. El vector del lado derecho b se obtuvo artificialmente asumiendo que la solución al sistema lineal es el vector $(1, 1, \ldots, 1)^t$.

Proceso de prueba. Como se discutió en la sección §3.4.2, el *Solver* Multinivel Algebraico se optimizó para solapar eficientemente cómputo con entrada/salida para atenuar las latencias de memoria/disco. Para evaluar si las técnicas de prebúsqueda, ya implementadas para el proceso de factorización *Cholesky* (Castellanos y Larrazábal, 2010) producen mejoras en el tiempo de ejecución, se condujo el proceso de prueba en tres etapas:

Tabla 11: Matrices de prueba para el *Solver* Multinivel.

Identificador	$n_x \times n_y \times n_z$	**Tamaño**	**no nulos**
1.	$70 \times 70 \times 70$	343.000	2.371.600
2.	$80 \times 80 \times 80$	512.000	3.545.600
3.	$90 \times 90 \times 90$	729.000	5.054.400
4.	$100 \times 100 \times 100$	1.000.000	6.940.000
5.	$110 \times 110 \times 110$	1.331.000	9.244.400
6.	$120 \times 120 \times 120$	1.728.000	12.009.600
7.	$130 \times 130 \times 130$	2.197.000	15.277.600
8.	$140 \times 140 \times 140$	2.744.000	19.090.400

$n_x \times n_y \times n_z$: dimensión de la grilla de entrada; **Tamaño** en filas.

a Solución *in-core* del sistema lineal $Ax = b$ con la capa *out-of-core* desabilitada. En esta etapa todas las matrices se cargan en memoria física y el sistema lineal se resuelve usando las dos fases descritas en la sección §3.4.2.

b Solución del sistema lineal $Ax = b$, con la capa *out-of-core* habilitada pero sin usar técnicas de prebúsqueda. En esta etapa, no hay prebúsqueda de *nodos* (filas/columnas) desde los archivos temporales a caché, mientras se efectúa el cómputo, así las matrices se cargan en la estructura *out-of-core* y el sistema lineal se resuelve usando las dos fases descritas en la sección §3.4.2.

c El sistema lineal $Ax = b$ se resuelve con la capa *out-of-core* y el algoritmo de prebúsqueda activados. En esta etapa las matrices se cargan en la estructura *out-of-core* y el sistema lineal se resuelve usando las dos fases descritas en la sección §3.4.2. En esta fase tanto la carga como los procesos de transposición de la matriz y producto de matrices se ejecutan usando técnicas de prebúsqueda (Castellanos y Larrazábal, 2010), las cuales cargan tempranamente los *slots* (filas/columnas) en caché, antes de que sucedan los *fallos*, sacando provecho de los procesadores multi-core.

En cada una de las fases se midieron las siguientes variables: el tiempo que suman los pasos de inicialización y solución, la cantidad de memoria requerida el proceso completo de solución del sistema lineal (ver sección §3.4.2) y el error relativo $\|e\|$ en la solución de cada uno de los sistemas lineales determinado por la matriz de entrada. En todos los casos se midió el tiempo usando PAPI (*Performance Application Programming Interface*) versión 3.7.0 (Terpstra, 2009). Se usaron las funciones internas suministradas por la biblioteca `UCSparseLib` para medir la cantidad de memoria utilizada por el proceso completo. Cada prueba se repitió tres veces y en cada prueba se tomó el mejor resultado para obtener los valores de la tabla 12. En esta tabla la primera columna representa un identificador numérico *Id* que identifica la matriz de

Tabla 12: Uso de memoria y tiempo de ejecución (*F-ciclo*).

Id.	**in-core**		**out-of-core**				**error**
			sin prebúsqueda		**con prebúsqueda**		$\|e\|$
	Mem.	**Tiempo**	**Mem.**	**Tiempo**	**Mem.**	**Tiempo**	
1.	205,19	13,03	35,22	23,01	26,90	16,51	5,22e-07
2.	306,82	20,61	38,33	34,32	32,66	26,79	3,00e-06
3.	437,45	22,88	43,06	43,85	43,25	29,86	2,5-e-05
4.	600,71	29,20	58,65	53,15	58,83	38,56	1,37e-05
5.	800,25	42,41	78,01	72,81	78,00	55,56	3,32e-05
6.	1039,71	49,52	100,83	87,88	100,82	67,72	1,45e-05
7.	1322,72	69,57	127,79	119,87	128,54	105,99	1,65e-04
8.	1652,92	81,44	159,24	138,28	159,98	110,06	5,37e-05

Mem. memoria expresada en Megabytes; **Tiempo** está en segundos.

prueba de acuerdo a la tabla 11. Las siguientes columnas muestran el uso de memoria y el tiempo total de solución del sistema lineal para los diferentes ajustes del experimento. Esto es, con el núcleo *out-of-core* deshabilitado, con el núcleo *out-of-core* habilitado pero sin prebúsqueda y finalmente con ambos, el nucleo *out-of-core* y la prebúsqueda habilitados.

En la figura 38 se muestra el comportamiento del tiempo de ejecución cuando el sistema lineal se resuelve usando el *solver* Multinivel Algebraico *out-of-core*. En la figura se presentan las gráficas de la relación de tiempos $\frac{in-core}{out-of-core}$ para las matrices de prueba con la prebúsqueda habilitada y desabilitada. La relación de tiempos $\frac{in-core}{out-of-core}$ siempre es un número menor que uno, dado que *tiempo in-core*<*tiempo out-of-core*, porque hay un *overhead* ocasionado por el acceso al archivo temporal en disco cuando la capa *out-of-core* está habilitada. Finalmente, la figura 39 muestra gráficamente la cantidad de memoria que usada por el AMG para cada una de las matrices de prueba. Se graficó la relación entre la memoria usada con la capa *out-of-core* activada y desactivada. La relación $\frac{Memoria\ in-core}{Memoria\ out-of-core}$ siempre es un número mayor que uno porque en caché se almacena solamente un subconjunto de filas/columnas de las matrices necesarias para resolver el sistema lineal cuando la capa *out-of-core* está habilitada mientras que en el otro caso se almacenan en memoria fisica la totalidad de las matrices.

Análisis de resultados. Para el análisis de resultados se usarán las figuras 38 y 39. La figura 38 muestra que el comportamiento del tiempo de ejecución para la solución del sistema lineal sin usar técnicas de prebúsqueda es bastante uniforme y tiende a un valor sobre $0,50$, lo cual significa que el tiempo de ejecución con la capa *out-of-core* habilitada tiende a ser menos del doble que el tiempo de ejecución *in-core*

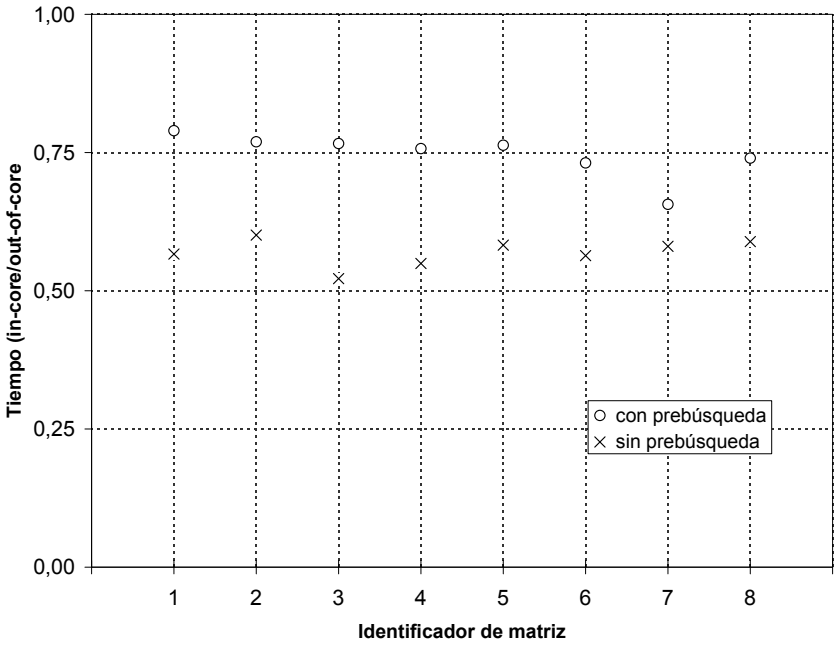


Figura 38: Relación de tiempo $\frac{in-core}{out-of-core}$ para AMG *F-ciclo*.

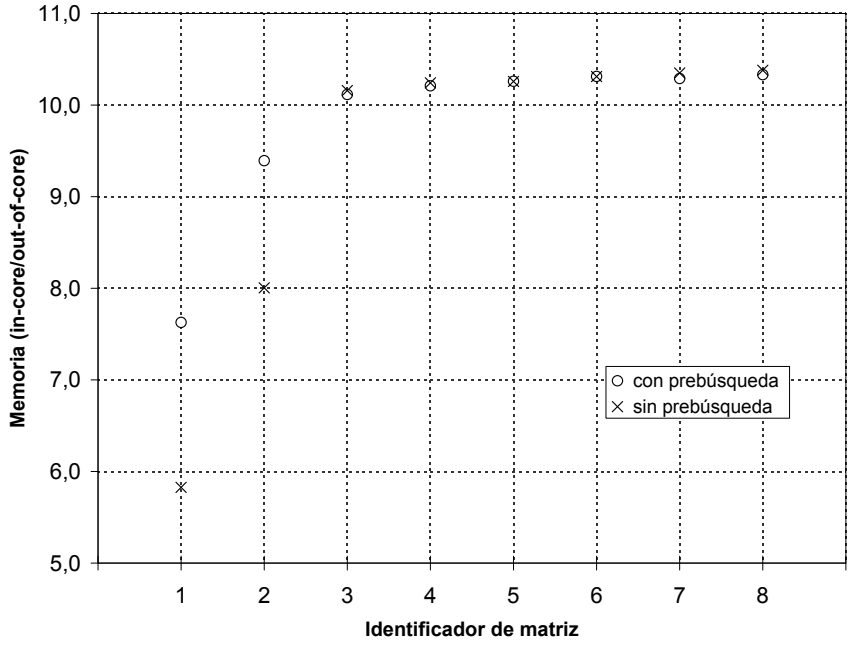


Figura 39: Relación de memoria $\frac{in-core}{out-of-core}$ para AMG *F-ciclo*.

para todas las matrices de entrada. Esta diferencia de tiempos se debe principalmente al *overhead* ocasionado por el acceso a los archivos temporales en disco. Esos archivos mantienen almacenados todos los vectores dispersos que componen todas las matrices involucradas en la solución del sistema lineal a través del Método Multinivel Algebraico.

Al observar los resultados de la figura 38, ellos muestran que hay una mejora significativa en tiempo de ejecución para todas las matrices de prueba cuando se habilita el algoritmo de prebúsqueda. Todas las matrices de entrada con la excepción de la número 7 muestran una relación de tiempo de ejecución excelente alrededor de $0,75$, lo cual significa que el *overhead* de la capa *out-of-core* está alrededor del 33 %. Se considera que la matriz 7 no saca provecho del algoritmo de prebúsqueda porque su número de filas 2.197.000 está ligeramente por encima de 2^{21} =2.097.152; como ya se comentó en la sección §3.4.2 el nucleo *out-of-core* se entonó para seleccionar los parámetros del caché de acuerdo al número de filas (columnas) según el valor 2^n que supera el total de filas (columnas). En este caso, el valor seleccionado de 22 (2^{22} =4.194.304) no obtiene el mejor desempeño del caché, porque en realidad el valor 21 (2^{21} =2.097.152) representa mejor los parámetros de caché para esta matriz.

La figura 39 muestra el uso de memoria de la capa *out-of-core*. Como se ve en esta figura, el algoritmo de prebúsqueda solamente afecta los resultados para las matrices 1 y 2, para las cuales permite un mayor ahorro de memoria. Con la excepción de las primeras dos matrices de prueba, la capa *out-of-core* permite resolver el sistema lineal usando menos de un 10 % de la memoria necesaria para la solución del sistema lineal *in-core*.

4. Conclusiones y Trabajo futuro

Se ha visto en esta tesis la concepción original de una capa *out-of-core*, basada en la teoría de memoria caché, la cual se desarrolló en tres etapas: en la primera se trabajaron los aspectos de más bajo nivel de la capa como: concebir el tipo `TDMatrix` *out-of-core* para manejar todos los aspectos relacionados con la matriz; tomar como unidad básica indivisible el vector disperso que representa una fila (columna) de la matriz (*nodo*); dividir la matriz en bloques consecutivos de vectores dispersos de tamaño 2^n (*slot*); usar un formato CSR/CSC modificado para almacenar los vectores en un archivo temporal y poderlos transferir eficientemente desde/hacia el caché. Para evaluar esta primera etapa se escogieron las operaciones básicas de matrices y con ellas en esta primera versión se puso a prueba exitosamente la estructura *out-of-core*. En la segunda etapa se trabajó el problema de los métodos directos para la factorización de matrices. Esto conllevó a la optimización de la estructura del caché y también a la incorporación de técnicas de prebúsqueda para atenuar el alto costo que representaba el manejo de los *fallos* de caché. En esta segunda etapa debieron incorporarse técnicas de programación multihilos (*multithreading*) para posibilitar el aprovechamiento de la prebúsqueda. Se aprovecharon en este momento los procesadores multi-core que permitieron solapar efectivamente el cómputo con la entrada/salida de memoria/disco mediante la utilización de una lista de solicitudes pendientes que mediante una cola re-entrante posibilitó la paralelización efectiva de cómputo con entrada/salida. En la tercera etapa se asumió un reto aun mayor con la incorporación de la capa *out-of-core* al *solver* multinivel algebraico, lo cual supuso, además de la incorporación de los métodos iterativos, la automatización de la capa *out-of-core* para que esta seleccionara automáticamente los valores de configuración de su caché para cada una de las matrices de la jerarquía de niveles que maneja el *solver* multinivel.

4.1. Conclusiones

Terminada la tercera etapa se puede concluir antes de todo que se han alcanzado los objetivos de esta tesis, pues se tiene una capa *out-of-core* operativa que se desarrolló desde cero y permite, con un bajo *overhead*, la realización de las diferentes operaciones de la biblioteca `UCSparseLib` con grandes ahorros de memoria, posibilitando la solución de sistemas complejos en computadores con hardware básico, según se pudo corroborar en el capítulo 3.

Fue importante en este desarrollo el diseño inicial de la capa de software basado en lineamientos generales que permitieron en varias etapas optimizar su funcionamiento con poco retrabajo. En este caso las decisiones iniciales de asumir como unidad básica el vector disperso que representa una fila/columna representó un aporte inicial muy importante.

La incorporación de hilos (*threads*) combinada con la incorporación de técnicas de prebúsqueda, en la segunda etapa de esta tesis, fue determinante para obtener mejoras en el desempeño que atenuaron efectivamente las latencias ocasionadas por el acceso a memoria/disco.

En la tercera etapa del desarrollo se pudo corroborar la estabilidad del nucleo *out-of-core*, operando con y sin prebúsqueda. Esto permitió la automatización del proceso de configuración de parámetros del caché de la capa *out-of-core*.

Se puede concluir que la capa *out-of-core* presenta su mayor eficiencia cuando la memoria requerida por una aplicación excede a la memoria física disponible por el sistema; cuando la memoria requerida puede ser suministrada por la memoria física mas la de intercambio, la capa out-of-core muestra ser muy eficiente, pero cuando la memoria necesaria supera la memoria física mas la memoria virtual, la única forma de ejecutar la aplicación es con la capa *out-of-core* habilitada.

4.2. Trabajo futuro

La realización de esta tesis nos abre nuevos espacios de investigación donde las técnicas desarrolladas y aplicadas en este trabajo tienen aplicación. No obstante el software desarrollado hasta este momento puede perfeccionarse y extender su aplicación a otros métodos de resolución de sistemas lineales dentro de la biblioteca `UCSparseLib`.

Entre las áreas de investigación donde se pueden aplicar estas técnicas *out-of-core* está el desarrollo de visualizadores para volúmenes de gran tamaño que requieren de la aplicación de nuevas técnicas *out-of-core* para mejorar su rendimiento. Actualmente estas técnicas tienen aplicación y son de interés en la visualización de imágenes sísmicas para la exploración petrolera.

Mejorar la eficiencia de las transferencias de entrada/salida implementando una capa de bajo nivel que maneje bloques de datos de tamaño fijo para favorecer las operaciones de entrada/salida entre disco duro y memoria principal.

Otra área de reciente aplicación, tiene que ver con la utilización de las unidades de hardware gráfico (GPU) para realizar cómputo. En esta área, ante la limitación de memoria para la resolución de sistemas grandes, se pueden aplicar estas técnicas *out-of-core* para la solución eficiente sistemas lineales de mayor tamaño.

Referencias

Baer, J., y Chen, T. (1991). An effective on-chip preloading scheme to reduce data access penalty. En *Proceedings of the 1991 acm/ieee conference on supercomputing, albuquerque, new mexico* (p. 176-186). New York, NY, USA: Supercomputing '91. ACM.

Brandt, A. (1977). A. multi-level adaptive solutions to boundary-value problems. *Math. Comput*, 31(138): 333-390.

Caron, E., y Utard, G. (2004). On the performance of parallel factorization of out-of-core matrices. *Parallel Computing archive*, *30*, 357-375.

Castellanos, J., y Larrazábal, G. (2007). Implementación out-of-core para el producto matriz-vector y transpuesta de matrices dispersas. En *Conferencia latinoamericana de computación de alto rendimiento* (p. 250-256). Santa Marta, Colombia: ISBN: 978–958–708–299–9.

Castellanos, J., y Larrazábal, G. (2008). Soporte out-of-core para operaciones básicas con matrices dispersas. En *Desarrollo y avances en metodos numéricos para ingeniería y ciencias aplicadas* (pp. 978–980). Caracas, Venezuela: Sociedad Venezolana de Métodos Numéricos en Ingeniería, ISBN: 978–980–7161–00–8.

Castellanos, J., y Larrazábal, G. (2010). An efficient implementation of an algebraic multigrid solver. *Faraute de Ciencias y Tecnología. Facultad de Ciencias y Tecnología. Universidad de Carabobo*, *5*(1-2010).

Castellanos, J., y Larrazábal, G. (2011). A cholesky out-of-core factorization. *Mathematical and Computer Modelling*. (disponible en linea http://dx.doi.org/10.1016/j.mcm.2011.05.057.)

Castellanos, J., y Larrazábal, G. (2012). Algebraic multigrid solver: an out-of-core approach. En *Avances en simulación computacional y modelado numérico* (pp. 37–42). Caracas, Venezuela: Sociedad Venezolana de Métodos Numéricos en Ingeniería, ISBN: 978–980–7161–03–9.

Davis, T. A., y Hu, Y. F. (2010). The university of florida sparse matrix collection. (Submitted to ACM Transactions on Mathematical Software. http://www.cise.ufl.edu/ davis/techreports/matrices.pdf (updated March 2010))

Demke B., A. (2005). Explicit compiler-based memory management for out-of-core-applications. *Carnegie Mellon University Ph.D Dissertation CMU-CS-05-140*.

Demke B., A., Mowry, T. C., y Krieger, O. (2001). Compiler-based I/O prefetching

for out-of-core applications. *ACM Transactions on Computer Systems*, *19*(2), 111-170.

Feng, Z., y Li, P. (2008). Multigrid on gpu: Tackling power grid analysis on parallel simt platforms. *2008 IEEE/ACM International Conference on Computer-Aided Design (iccad)*, 647-654.

Karypis, G., y Kumar, V. (1999). A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, *20*, 359-392.

Kickinger, F. (1997). Algebraic multigrid for discrete elliptic second-order problems. En *Technical report. institute for mathematics.* Johannes Kepler University Linz, Austria.

Kroft, D. (1981). Lockup-free instruction fetchprefetch cache organization. En *Proceedings of the 8th annual symposium on computer architecture* (pp. 81–87). Los Alamitos, CA, USA: IEEE Computer Society Press. Descargado de `http://dl.acm.org/citation.cfm?id=800052.801868`

Larrazábal, G. (2002). Técnicas algebráicas de precondicionamiento para la resolución de sistemas lineales. *Departamento de Arquitectura de Computadores (DAC), Universidad Politécnica de Cataluña, Barcelona, Spain. Tesis Doctoral ISBN: 84–688–1572–1*.

Larrazábal, G. (2004). UCSparseLib: Una biblioteca numérica para resolver sistemas lineales dispersos. *Simulación Numérica y Modelado Computacional, SVMNI, TC19–TC25, ISBN:980-6745-00-0*.

Moor, K. (2002). A I/O performance enhancements of Out-of-Core applications. *Notre Dame University, Department of Computer Science and Engineering*. (http://www.cs.indiana.edu/∼ksiek/PAPER/paper.pdf, [Consulta: 2007, marzo 8])

N. I. M. Gould, J. A. S., y Hu, Y. (2007). A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Trans. Math. Softw. 33,2, Article 10*.

Patterson, D. A., y Hennessy, J. L. (2005). *Computer organization and design: The hardware/software interface*. Third Edition: Morgan Kaufmann.

Raju, M. P., y Khaitan, S. (2009). High performance computing using out-of- core sparse direct solvers. *International Journal of Mathematical Physical and Engineering Sciences*(57).

Reid, J. (1984). Treesolv, a fortran package for solving large sets of linear finite-element equations. harwell report css 155. *PDE software: modules, interfaces and systems*, 1-17.

Reid, J., y Scott, J. (2009, april). An out-of-core sparse cholesky solver. *ACM Trans. Math. Softw.*, *21*, 9:1–9:33.

Rothberg, E., y Schreiber, R. (1999). Efficient methods for out-of-core sparse cholesky factorization. *SIAM Journal on Scientific Computing*, *21*, 129 - 144.

Rozin, E., y Toledo, S. (2005). Locality of reference in sparse cholesky methods. *Electronic Transactions on Numerical Analysis*, 21:81-106.

Samuel, B., y D'Azevedo, E. (2004). Benchmarking oocore, an out-of-core matrix solver. *Oak Ridge National Laboratory*. Descargado de `http://www.ornl.gov/~webworks/cppr/y2001/misc/123564.pdf`

Silva, C., Chiang, Y., El-Sana, J., y Lindstrom, P. (2002). *Out-of-core algorithms for scientific visualization and computer graphics.* Descargado de `citeseer.ist.psu.edu/article/silva02outcore.html`

Smith, A. J. (1982, September). Cache memories. *ACM Computing Surveys*, *14*(3), 473-530.

Suh, J., y Prasanna, V. K. (2002, April). An efficient algorithm for out-of-core matrix transposition. *IEEE Transactions on Computers*, *51*(4).

Terpstra, J. H. Y. H. D. J., D. (2009). Collecting performance data with papi-c. *Tools for High Performance Computing*, 157-173.

Toledo, S. (1999). A survey of out-of-core algorithms in numerical linear algebra. *External Memory Algorithms and Visualization, J. Abello and J. S. Vitter, Eds., DIMACS Series in Discrete Mathematics and Theoretical Computer Science.*.

X. Shi, H. B., y Zhou, K. (2009). Out-of-core multigrid solver for streaming meshes. *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia*, *28*.

Z. Bai, J. D. A. R., J. Demmel, y van der Vorst, H. (2000). *Templates for the solution of algebraic eigenvalue problems: A practical guide*. Philadelphia: SIAM.