Una factorización Cholesky out-of-core

Jorge Arturo Castellanos Díaz
Centro Multidisciplinario de Visualización y Cómputo Científico (CEMVICC), Facultad de Ciencias y Tecnología (FACYT), Universidad de Carabobo, Valencia, Venezuela.
Email: jcasteld@uc.edu.ve

Germán A. Larrazábal S.
Centro Multidisciplinario de Visualización y Cómputo Científico (CEMVICC), Facultad de Ciencias y Tecnología (FACYT), Universidad de Carabobo, Valencia, Venezuela.
Email: glarraza@uc.edu.ve

***Resumen*—Con frecuencia, el núcleo computacional en el software general de simulación es el encargado de resolver el sistema lineal. Este núcleo puede ser denso o disperso dependiendo de la discretización numérica. Para la solución de sistemas lineales grandes es deseable el uso de una estructura dispersa. Los *solvers* directos dispersos como Cholesky, $\mathbf{LDL^T}$, o LU son perfectas cajas negras ya que ellos necesitan únicamente como entradas: la matriz (A) y el vector del lado derecho (b) del sistema lineal $Ax = b$. Pero presentan como principal desventaja que la cantidad de memoria necesaria para resolver el sistema se incrementa rápidamente con el tamaño del problema. En este trabajo, se propone la implementación de un software *out-of-core* para la factorización Cholesky que resuelve el problema de la memoria. La capa *out-of-core* se basa en el desarrollo de una memoria caché especializada que almacena solamente una parte de la matriz del problema (A) y de la matriz factorizada (L) que residen en un archivo de disco. La matriz factorizada (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 de cada fila/columna y el segundo proceso calcula los valores numéricos para cada una de las posiciones usando el Método Multifrontal. Se han obtenido ahorros significativos de memoria usando el soporte *out-of-core* propuesto. Los resultados preliminares muestran un buen desempeño usando el *solver* LU *out-of-core* para un conjunto de matrices que provienen de un operador escalar elíptico discretizado usando diferencias finitas.**

***Abstract*—Frequently, the computational core in general simulation software is the linear system solver. This solver may be dense or sparse depending on the numerical discretization. For solving large systems it is desirable the use of a sparse structure. The direct sparse solvers such as Cholesky, $\mathbf{LDL^T}$, or LU are perfect black boxes, i.e., they only need the matrix (A) and the right hand side vector (b) of the linear system $Ax = b$ as inputs. But their main disadvantage is that the memory required by them usually increases rapidly with problem size. In this work, we propose an out-of-core implementation for the Cholesky solver in order to overcome the memory problem. The out-of-core layer is based on a specialized cache memory development that stores only a part of the problem matrix (A) and the factorized matrix (L) whose data is stored in a file disk. The factorized matrix (L) is computed in a two step process, specifically: a symbolical and a numerical process. The first step computes position of the non- zero element of each row/col and the second process computes the numerical value for each position using the Multifrontal Method. We have obtained a significant save of memory with our proposal. The preliminary results show a good performance using our Cholesky out-of-core solver for a set of large matrices that arise from a scalar elliptic operator which is discretized using finite differences.**

***Keywords*-Out-of-core, Sparse cholesky factorization.**

I. INTRODUCCIÓN

El núcleo computacional que consume mayor tiempo de CPU en un paquete de simulación numérica en ingeniería es el *solver* lineal. Comúnmente, este *solver* esá basado en un clásico de solución, tal como un método directo o iterativo. Además, la matriz asociada al sistema lineal de ecuaciones, con frecuencia, es disperso y de gran tamaño. Los métodos iterativos tienen la desventaja que no son generales y requerien varios parámetros de usuario y además necesitan precondicionadores a fin de acelerar su convergencia. Sin embargo, estos métodos son populares ya que su principal operación es matriz por vector, y esta opreación es altamente paralelizable. Los métodos directos como Cholesky, $\mathrm{LDL^T}$, o LU son perfectas cajas negras ya que ellos necesitan únicamente como entradas la matriz (A) y el vector del lado derecho (b) del sistema lineal $Ax = b$. Pero presentan como principal desventaja que la cantidad de memoria necesaria para resolver el sistema se incrementa rápidamente con el tamaño del problema. Esta desventaja se ve claramente en un reciente estudio [5]. El problema de la factorización Cholesky *out-of-core* de matrices dispersas no es nuevo; ya en 1984 se planetea un *solver* multifrontal [11]. En [13] se revisa la eficiencia en la implementación de tres métodos *out-of-core* para la factorización Cholesky y se consideran alternativas para mejorar la eficiencia de los métodos revisados. En un trabajo reciente [12] se presenta un *solver* Cholesky *out-of-core* escrito en Fortran 95, cuyo funcionamiento se basa en un paquete de memoria virtual que provee las facilidades para la escritura y lectura directa de archivos en disco. A diferencia de [12] en este trabajo se implementa la capa *out-of-core* de la factorizacón Cholesky como una parte del soporte *out-of-core* de la biblioteca UCSparseLib [8] que cuenta con un conjunto de funciones para la resolución de sistemas lineales dispersos.

En este trabajo, se propone la implementación de un software *out-of-core* para la factorización Cholesky que resuelve el problema de la memoria. La capa *out-of-core* se basa en el

desarrollo de una memoria caché especializada que almacena solamente una parte de la matriz del problema (A) y de la matriz factorizada (L) que residen en un archivo de disco. La matriz factorizada (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 de cada fila/columna y el segundo proceso calcula los valores numéricos para cada una de las posiciones usando el Método Multifrontal. Además, este trabajo representa un avance de [1], [2], mediante el cual se agrega una capa *out-of-core* a una bilioteca de solución de sistemas lineales. Como parte de este proyecto, el núcleo *out-of-core* debe estar en capacidad de resolver eficientemente el problema de la factorización de matrices dispersas. En este artículo se tratan la adiciones necesarias al núcleo *out-of-core* ya existente que hicieron posible resolver este problema.

El presente artículo está organizado de la siguiente forma. En la sección §II, se presentan los formatos de almacenamiento para matrices dispersas, necesarios para comprender este trabajo. En la sección §III, se explica el diseño del núcleo *out-of-core*, también se muestra la interacción con el núcleo a través de una macro en alto nivel. En la sección §IV, se muestra los detalles de implementación de la factorización Cholesky haciendo uso del del núcleo *out-of-core* implementado. En la seción §V, se presentan resultados que muestran la utilización de memoria y el tiempo de ejecución de la factorización Cholesky cuando el núcleo *out-of-core* está activado y se comparan con versiones *in-core* equivalentes sin soporte *out-of-core*. Finalmente en la sección §VI se presentan las conclusiones del trabajo.

II. Almacenamiento de matrices dispersas

Como lo establece [4], los formatos de almacenamiento comprimido por fila y por columna son los más generales para el almacenamiento de matrices dispersas, pues no toman en cuenta el patrón de esparcidad 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 1. Matriz dispersa.

El almacenamiento comprimido por fila (CRS) guarda los elementos subsecuentes diferentes de cero de las filas de la matriz en localidades contiguas de memoria. Asumiendo que tenemos una matriz dispersa no-simétrica A (ver figura 1), creamos entonces tres vectores para almacenar la matriz (ver figura 2): uno para números en punto flotante (`val`) y los otros dos para enteros (`ia`, `id`). El vector `val` almacena los valores de los elementos diferentes de cero de la matriz A en el orden que ellos se obtienen cuando se recorre la matriz por filas. El vector `id` almacena los índices de las columnas de los elementos en el vector `val`. Esto es, si $\texttt{val(k)} = a_{i,j}$, entonces $\texttt{id(k)} = j$. El vector `ia` almacena las posiciones en el vector `val` que inician una fila; asi, si $\texttt{val(k)} = a_{i,j}$, entonces $\texttt{ia(i)} \leq k < \texttt{ia(i+1)}$. Por convención, se define $\texttt{ia(n+1)} = nnz$, donde nnz es el número de elementos diferentes de cero en la matriz A. Los ahorros en almacenamiento por este formato son significativos. En lugar de almacenar n^2 elementos, solo utilizamos $2nnz + n + 1$ localidades. Como un ejemplo, consideremos la matriz no-simétrica A mostrada en la figura 1. Aplicando el formato CRS se tienen los vectores $\texttt{val}, \texttt{ia}, \texttt{id}$ (mostrados en la figura 2).

nodes	7	rows
nnz	17	no nulls
ia	0 3 5 7 10 12 14 17	id position

id	0	2	5	1	3	1	2	0	3	6	2	4	1	5	2	4	6
val	11	13	16	22	24	32	33	41	44	47	53	55	62	66	73	75	77

Figura 2. Formato de almacenamiento comprimido por fila (CRS).

Análogo al formato CRS, hay un formato comprimido por columna (CCS). El formato CCS es idéntico al formato CRS, excepto que las columnas de A se almacenan (subsecuentemente) en vez de las filas. En otras palabras, el formato CCS es el formato CRS para A^T.

Con el objeto de facilitar el acceso independiente de un conjunto de filas (columnas) desde un archivo en disco, se definió dentro de la capa *out-of-core* un formato de almacenamiento temporal en disco (ver figura 3).

pos	Temporal file
0	0 2 5 11 13 16
6	1 3 22 24
10	1 2 32 33
14	0 3 6 41 44 47
20	2 4 53 55
24	1 5 62 66
28	2 4 6 73 75 77
34	

Figura 3. Formato de almacenamiento para el archivo temporal.

Este formato representa una variación del formato CRS (CCS), mediante el cual se han segmentado los vectores `id` y `val` en fraccciones que se corresponden con cada una de las filas (columnas) de la matriz (ver figura 1). Las filas (columnas) representadas por segmentos de los vectores

id y val se almacenan entonces concecutivamente en el archivo temporal. El acceso de las filas (columnas) en el archivo se logra a través del vector pos que mantiene las posiciones de inicio de cada una de las filas (columnas) dentro del archivo. Como la matriz se almacena en orden ascendente por fila (columna) dentro del archivo temporal, es posible transferir fácilmente un conjunto de filas (columnas) simultáneamente a la memoria, pues a través del vector pos se puede determinar la posición de inicio y el tamaño del bloque a transferir. En el ejemplo de la figura 3 y en el resto del artículo se supone que los índices y los valores ocupan una unidad de almacenamiento; esto se ha hecho para facilitar las explicaciones. En la implementación real los índices ocupan la mitad de espacio de los valores, pues se considera que los primeros son del tipo entero mientras que los segundos son del tipo double.

III. La capa *out-of-core*

Según se indicó en la sección I, la implementación *out-of-core* de la factorización Cholesky obedece al desarrollo de una capa de software que soportará todas las operaciones *out-of-core* de la biblioteca de solución de sistemas dispersos UCSparseLib [8]. Cuando se inició el soporte *out-of-core* para la factorización Cholesky ya se contaba con el núcleo *out-of-core* para las operaciones básicas con matrices dispersas [1], [2]; este software soportaba eficientemente las operaciones: transpuesta de matrices y los productos matriz-vector y matriz-matriz. Para la comprensión de este trabajo se explicará a continuación brevemente la operación de esta capa.

Según ya se indicó en la sección II, la matriz que se lee del archivo de disco se almacena en un archivo temporal que obedece a una variación del formato de almacenamiento CRS/CCS que facilita el acceso a cada fila (columna) de la matriz según sea el formato utilizado CRS o CCS. El ńucleo *out-of-core* usa como unidad mínima e indivisible de almacenamiento una fila (columna), la cual se denominará *nodo* para futuras referencias en este artículo.

De acuerdo con [15] para tener un buen desempeño, un algoritmo *out-of-core* debe acceder los datos almacenados en disco en forma de grandes *bloques* contínuos, y una vez que un *bloque* se carga en memoria, este debe reutilizarse muchas veces. Esto supone que la capa *out-of-core* debe permitir el acceso de conjuntos de *nodos* concecutivos para aprovechar los principios de localidad temporal y espacial que soportan la teoría de la memoria caché explicada en detalle en [14].

A diferencia de las matrices en formato denso cuyas filas (columnas) tienen un tamaño fijo, las matrices dispersas presentan un tamaño variable por fila (columna) porque solo almacenan los elementos no-nulos y su posición. Si se quiere aprovechar la teoría de la memoria caché, se debe almacenar en memoria principal un (o mas de un) bloque de *nodos*. Para el acceso eficiente a los *nodos* almacenados en disco, el núcleo *out-of-core* crea en memoria una caché de correspondencia directa [10] que saca provecho de los bits de direccionamiento de *bloque* en relación a la dirección del *nodo*.

Así, cuando se activa el soporte *out-of-core*, la matriz dispersa no se carga completamente a la memoria y en lugar de ello solo se transfiere desde el archivo temporal en disco a la memoria un subconjunto de *nodos* consecutivos de la matriz que en adelante denominaremos *bloque*.

Para dividir apropiadamente la matriz de trabajo en *bloques* fácilmente direccionables, el número total de *nodos* de la matriz se completa al próximo número entero 2^n (n: entero ≥ 0). En nuestro ejemplo, para la matriz de la figura 4 que tiene 7 *nodos*, el próximo número 2^n es 8. La figura 4 muestra la conveniencia de completar el total de *nodos* a $8 = 2^3$. De esta forma, la matriz A puede ser fácilmente dividida en 4 *bloques* de $2^1 = 2$ *nodos* cada uno.

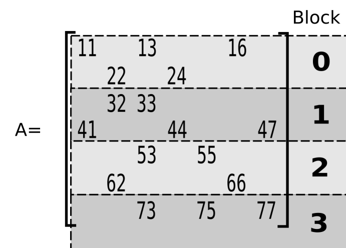

Figura 4. División en bloques de una matriz dispersa.

Para ocultar los detalles de implementación de la capa *out-of-core* al programador y acceder en forma transparente a cada uno de los *nodos* de la matriz, se han definido dos macros: una para operar sobre matrices almacenadas por fila (CRS) y la otra para operar sobre matrices almacenadas por columna (CCS). Cuando el *nodo* es una fila (formato CRS) se usa la macro For_OOCMatrix_Row, y cuando el *nodo* es una columna (formato CCS) se usa la macro For_OOCMatrix_Col. Para simplificar la explicación, de aqui en adelante solo se hará referencia a *nodos* del tipo fila y por lo tanto se usará solamente la macro For_OOCMatrix_Row (ver figura 5).

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

Figura 5. Definición de la macro For_OOCMatrix_Row.

Como se ve en la figura 5, la macro For_OOCMatrix_Row tiene cuatro parámetros:

- MM es una estructura que contiene información general de la matriz, por ejemplo: formato (CRS/CCS), número de *nodos*, apuntador al archivo temporal asociado a la matriz, etc.
- ii es un entero que representa el *nodo* actual.

- `row` es una estructura que contiene la información asociada al *nodo*; esta estructura tiene tres campos importantes: `nz`, de tipo entero que contiene el número de elemento diferentes de cero del *nodo*, `id` y `val` son vectores del tipo entero y del tipo *double* que contienen los índices del *nodo* y los valores no-nulos del *nodo* respectivamente.
- `mode` es un campo que maneja el acceso al *nodo* actual; 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 la rutinas necesarias para el manejo de la caché y del archivo temporal en disco se llaman dentro del ámbito de la macro. De esta forma el programador elimina su preocupación por los detalles de implementación de bajo nivel que reducen su productividad y la transportabilidad de sus códigos.

La figura 6 muestra gráficamente como opera el nucleo *out-of-core*. Se supone que la matriz A (ver figura 1) reside en un archivo en formato CRS (ver figura 2); para este ejemplo, la caché en memoria se configuró con un tamaño de 1 *bloque* de 2^1 *nodos*. Se tiene tambien un archivo temporal asociado a la matriz, como se comentó en la sección II.

Cuando la matriz A se carga por primera vez desde un archivo en disco, los índices se leen desde el archivo de acuerdo a la especificación del formato CRS (ver la sección II). Como se ve en la primera parte de la figura 6, los índices cargados desde el archivo de entrada se escriben en la caché. Cuando la caché está totalmente llena con los índices de los dos primeros *nodos* (filas), es decir, cuando cuando se van a escribir en la caché los índices del tercer *nodo*, provenientes del archivo en disco, se escribe en el archivo temporal en disco el contenido total del *bloque* almacenado en caché. Los índices se almacenan en conjunto con ceros (en formato double) con el objeto de reservar espacio para los elementos no-cero asociados a tales índices; los valores del *nodo* se cargarán en la siguiente fase, despues que se hayan cargado todos los índices.

La segunda parte consiste de la carga de los valores no-nulos desde el archivo de entrada. En esta fase, los índices que se cargaron en la primera parte son recargados a la caché desde el archivo temporal; de esta forma es posible combinar la información de los índices con los valores en la estructura de la caché y en el archivo temporal en forma de unidades independientes (*nodos*), para tener un acceso más eficiente a cada uno de los *nodos* al trabajar con operaciones sobre matrices.

La tercera parte de la figura 6 muestra como los *nodos* se leen secuencialmente desde el archivo temporal hacia la caché. Esto aplica a la mayoría de los casos de operaciones básicas con matrices cuando se accede secuencialmente a los *nodos* de la matriz.

Las operaciones básicas con matrices toman ventaja del principio de localidad espacial [14] de la caché de correspondencia directa del núcleo *out-of-core*. Por ejemplo, cada vez que el algoritmo producto matriz-vector, trata de acceder a un *nodo* que no está en caché (ver figura 6), se carga desde el archivo temporal un nuevo *bloque* de 2^n *nodos* a la caché. Caso contrario, si el *nodo* está en la caché, este se lee desde la memoria. Aunque el algoritmo del producto matriz-vector no hace uso del principio de localidad temporal de la caché (enunciado en [14]), el código de la figura 7 muestra buenos tiempos de ejecución. Esto es porque los *nodos* de la matriz de entrada se acceden secuencialmente y el soporte *out-of-core* puede resolver automáticamente los fallos de caché, cargando un *bloque* de 2^n *nodos* concecutivos. De esta forma el efecto de los altos tiempos accesos al disco se minimizan cuando la tasa de aciertos es alta ($> 99\%$).

```
for (ii= 0; ii< nn; ii++)
{
  For_OOCMatrix_Row( M, ii, rc, READ )
  {
    raux = 0.0;
    for (kk= 0; kk< rc.nz; kk++)
     raux = raux + rc.val[kk] * xx[rc.id[kk]];
    yy[ii] = raux;
  }
}
```

Figura 7. Producto matriz-vector.

IV. Factorización de la matriz

La factorización Cholesky *out-of-core* de la matriz dispersa se realiza en un proceso de cuatro pasos:

1) Lectura de la matriz desde un archivo de disco y almacenamiento en un archivo temporal usando el esquema descrito en la sección III.
2) Ordenaniento de la matriz usando METIS [6] con el objeto de reducir el número de elementos no-nulos de la matriz factorizada y por tanto reducir el tiempo del proceso de factorización.
3) Factorización simbólica de la matriz, para determinar la posición de los elementos no nulos de la matriz factorizada.
4) Factorización numérica de la matriz, para calcular el valor de cada uno de los elementos no-nulos de la matriz factorizada.

El tiempo de procesamiento que toma la factorización viene determinado por el cuarto paso, o sea, la factorización numérica de la matriz, el cual toma aproximadamente el 50% del tiempo total para el proceso *in-core*, es decir, sin usar la capa *out-of-core*. Los tres primeros pasos del proceso de factorizar tienen la ventaja que dependen del acceso secuencial de la matriz de entrada, lo cual significa que el nucleo *out-of-core* probado con las operaciones básicas de matrices presenta un comportamiento aceptable.

El cuarto paso del proceso de factorización presenta dos problemas adicionales que forzaron un rediseño del núcleo

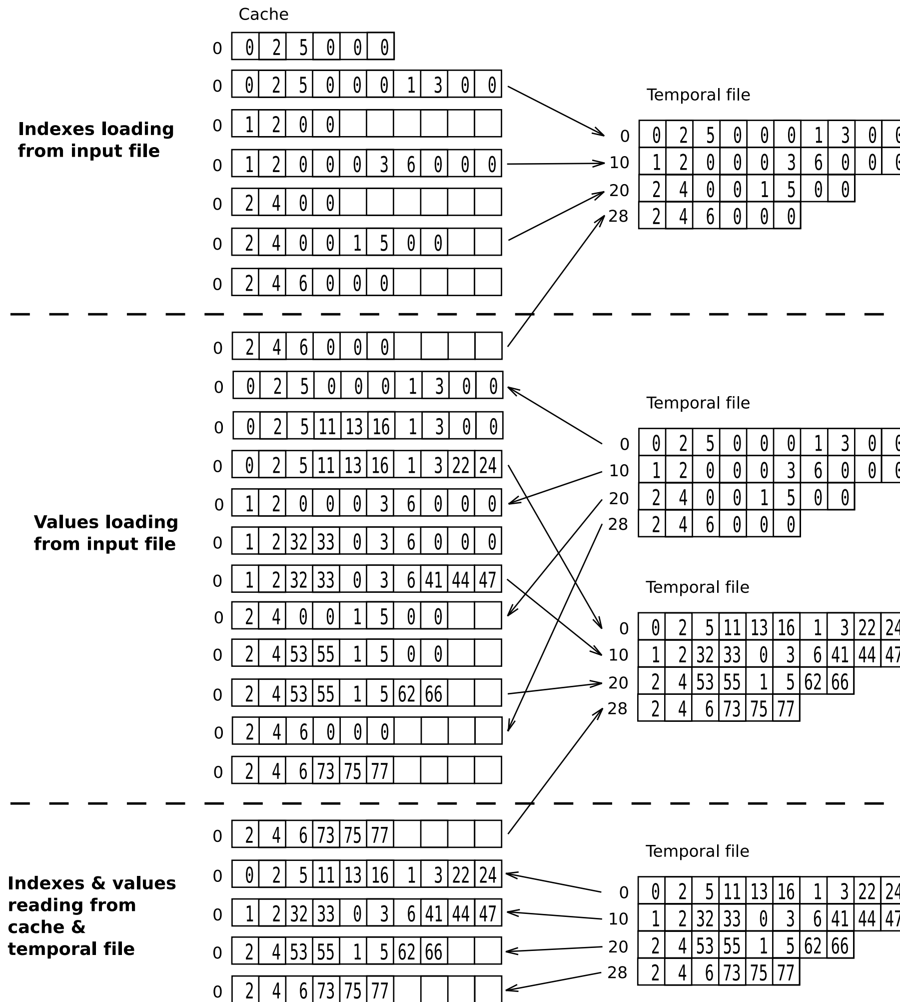

Figura 6. Operación del núcleo *out-of-core*.

out-of-core existente. El primer problema lo constituyó el anidamiento de macros `For_OOCMatrix_Row` necesario por el algoritmo de factorización numérica segun se ve en la figura 8. Para realizar eficientemente este anidamiento se debe mantener en memoria el bloque que contiene al *nodo* `rowI` mientras que se carga en memoria el bloque que contiene al *nodo* `rowJ`. Desde el punto de vista del diseño del nucleo *out-of-core* esto significó cambiar la organización de la caché para soportar múltiples vías para mantener el *nodo* `rowI` en una vía mientras que se cargan los distintos *nodos* `rowJ` en la otra vía. El segundo problema lo representó el incremento en la tasa de fallos durante el proceso de factorización de la matriz que aumentaba el tiempo de ejecución por el alto costo del acceso a disco. Para resolver este problema se incorporó una Tabla de Predicción de Referencias (RPT), siguiendo el esquema planteado por [3], la cual determina el bloque a ser prebuscado en la caché. Para resolver el acceso concurrente de peticiones a la caché tanto para fallos como para prebúsquedas, se diseñó e implementó una Lista de Solicitudes Salientes (ORL) como una lista enlazada, partiendo del esquema planteado por [7]. Cada entrada de la lista enlazada contiene la siguiente información: un apuntador al *slot* (bloque dentro de la caché) que se desea actualizar; el tamaño del *slot* en memoria; un entero que indica el número de bloque en el archivo de disco (ver figura 4), este entero indica el nuevo bloque que se copiará al *slot* desde el disco; un entero que indica el bloque actual en el *slot* de la caché; un entero que indica el numero de bloque del *slot* de la caché, un entero que indica en cual vía de la caché está el *slot*; también existe para cada entrada un conjunto de banderas que señalan el estado actual de la entrada, ellas son: *valid*, indica si el *slot* actual se está actualizando o contiene datos válidos, *dirty* indica si los datos del *slot* actual deben copiarse a disco antes de ser actualizados, *nouse* indica si el *slot* actual contiene datos recien prebuscados que aun no han sido utilizados, *lock* indica si el *slot* actual no puede ser reemplazado porque esta siendo utilizado.

Tabla I
USO DE MEMORIA Y TIEMPO DE EJECUCIÓN PARA LA FACTORIZACIÓN CHOLESKY.

Matriz			**In-core**			**Out-of-core**			**Resultados %**	
nodos	*NnzA*	*NnzL*	*memoria*	*tiempo*	*error*	*memoria*	*tiempo*	*error*	*memoria*	*ovhd*
8.000	53.600	719.350	11.083.180	0,868	6,39e−13	1.810.032	1,800	6,39e−13	16,33	107,37
27.000	183.600	4.166.303	58.356.968	10,433	2,52e−12	2.521.168	20,113	2,52e−12	4,32	92,78
64.000	438.400	13.683.905	184.069.636	57,696	6,58e−12	5.839.084	115,215	6,58e−12	3,17	99,69
125.000	860.000	36.699.814	479.441.548	223,306	1,35e−11	11.447.284	552,383	1,35e−11	2,39	147,37

Tabla II
COMPORTAMIENTO DE LA PREBÚSQUEDA AL FACTORIZAR LAS MATRICES.

Nodos	*Accesos*	*Aciertos*	*Aciertos*(%)	*Fallos*	Prebúsquedas	*lecturas*	*escrituras*
8.000	8.000	7.997	99,96	3	247	250	250
27.000	27.000	26.998	99,99	2	1.686	1.688	1.688
64.000	64.000	63.973	99,96	27	7.973	8.000	8.000
125.000	125.000	124.632	99.71	368	15.257	15.625	15.625

```
for (ii= 0; ii< nn; ii++)
{
  For_OOCMatrix_Row( GG, ii, rowI, READ_WRITE )
  {
    for (kk= 0; kk< rowI.diag; kk++)
    {
      jj = rowI.id[kk];
      For_OOCMatrix_Row( GG, jj, rowJ, READ )
      {
        operaciones_1;
      }
      operaciones_2;
    }
  }
}
```

Figura 8. Anidamiento de macros.

V. RESULTADOS

Para evaluar el soporte *out-of-core*, todos los códigos se compilaron usando `gcc x86_64` versión 4.3 con las banderas de optimización `-O2 -funroll-loops -fprefetch-loop-arrays`, ejecutándose sobre un procesador Intel Core™2 Duo $P8600$, sistema operativo `GNU/Linux`, kernel `2.6.28-11-SMP`, los datos se almacenaron temporalmente en un disco SATA-3™ $WD5000BEVT$, el cual tiene una latencia promedio de 5.5*ms*, una velociad del *buffer* al *host* de 3*GB/s* y una velocidad máxima de *buffer* a disco de 97*MB/s*, la memoria de intercambio está habilitada (`swapon`). El tiempo mostrado en las tablas es la suma del tiempo de usuario mas el de sistema que corresponden con el proceso de factorización simbólica mas el de factorización numérica y esta expresado en segundos.

La tabla I muestra los resultados para la factorización Cholesky. La tabla tiene cuatro filas de datos que se corresponden con cuatro tamaños de matriz (8.000; 27.000; 64.000; 125.000). Las matrices de prueba se generaron a partir de la discretización por diferencias finitas de una ecuación $3D$ escalar de convección-difusión [9] para tamaños de malla: $2.000 \times 2.000 \times 2.000$, $3.000 \times 3.000 \times 3.000$, $4.000 \times 4.000 \times 4.000$ y $5.000 \times 5.000 \times 5.000$. El lado derecho (vector b) del sistema lineal generado fue obtenido artificialmente suponiendo que la solución del sistema lineal es el vector $(1, 1, \ldots, 1)^t$.

La tabla está dividida en cuatro secciones: **Matriz**, **In-core**, **Out-of-core** y **Resultados**. La primera sección se refiere a los detalles de las matrices dispersas, es decir, el número de *nodos* ($Nodos$: filas en formato CRS) y total de elementos no-nulos de las matrices de entrada ($NnzA$) y salida ($NnzL$). La segunda sección contiene los resultados obtenidos sin el soporte *out-of-core* activado, es decir, *memoria* usada en *bytes*, tiempo medido en segundos y error relativo resultante en la resolución del sistema lineal. La tercera sección presenta los resultados cuando el soporte *out-of-core* está activado, es decir, *memoria* usada en *bytes*, tiempo medido en segundos y error relativo cometido en la solucion del sistema lineal. La cuarta sección compara los resultados sin y con el soporte *out-of-core* activado en términos de porcentaje de uso de memoria (*memoria*) y porcentaje de *overhead* en tiempo de ejecución ($ovhd$). Los ahorros de memoria muestran la fracción de memoria física que usa la implementación con la capa *out-of-core* en relación con el total de memoria física que necesita la implementación *in-core*. La penalización en tiempo se calculó como un incremento porcentual en el tiempo de ejecución que muestra la implementación cuando el soporte *out-of-core* está activado.

La tabla II muestra el comportamiento del algoritmo de prebúsqueda durante el proceso de factorización numérica. La primera columna (*Nodos*) permite conocer a cual matriz se está haciendo referencia, la segunda (*Accesos*) se refiere al número de accesos a memoria del algoritmo, la tercera y cuarta hacen referencia al comportamiento de aciertos en forma absoluta y porcentual. La cuarta y quinta columna se refieren al numero de fallos y prebúsquedas. Las columnas siete y ocho se refieren a los accesos a disco. El total de accesos de lectura a disco es igual al número de fallos mas

el de prebúsquedas.

Analizando los resultados en la primera tabla se puede apreciar que hubo ahorros significativos en el uso de memoria que varían entre un 83% $(16,33)$ para una matriz de entrada de 8.000 *nodos* y 97% $(2,39)$ para una matriz de 125.000 nodos; el ahorro en el uso de memoria se incrementa en porcentaje cuando se incrementa el orden de la matriz de entrada, lo cual resulta muy ventajoso porque permite factorizar matrices grandes usando máquinas con poca memoria. En relación al *overhead* ocasionado por la capa *out-of-core*, se tiene que está en un promedio del 100% con matrices de entrada de 8.000, 27.000 y 64.000, pero desmejora hasta un 147% para la matriz de entrada de 125.000 *nodos*. En referencia a los resultados presentados en la segunda tabla tenemos que el algoritmo de prebúsqueda tiene muy buen comportamiento ayudando a mantener la tasa de aciertos en un valor mayor al 99%.

VI. Conclusiones y trabajos futuros

El núcleo *out-of-core* presentado en este trabajo soporta eficientemente la factorización Cholesky de matrices dispersas porque muestra ahorros importantes de memoria con overheads menores del 148%. Para obtener un buen comportamiento de un soporte *out-of-core* consideramos que es importante tener un algoritmo de prebúsqueda de alta eficiencia, como el desarrollado en este trabajo, que pueda reducir el efecto adverso de la tasa de fallos de la caché. El algoritmo de prebúsqueda debe poder operar en paralelo con el cómputo para aprovechar la tecnología *multicore* presente en la mayoría de computadores personales actuales. Como trabajo futuros se propone incorporar mejoras en la paralelización del algoritmo de prebúsqueda para reducir la penalización en tiempo de ejecución, estudiar el efecto del tamaño del *bloque* del cache y tamaño de la caché (en número de *bloques*) con el objeto de definir heurísticas para la selección automática de estos parámetros, extender la utilización de la capa *out-of-core* a otras funciones de la biblioteca *UCSparseLib*, entre ellas, los métodos directos: $\mathrm{LDL}^{\mathrm{T}}$, y LU e implementar la capa *out-of-core* para otros métodos de solución de sistemas lineales dispersos como los métodos iterativos y multinivel algebráico de la biblioteca *UCSparseLib*.

Agradecimientos

El presente trabajo ha sido posible gracias al financiamiento del CDCH–UC para los proyectos CDCH-UC No. 2004-002 y CDCH-UC No. 2004-011.

Referencias

[1] J. Castellanos and G. Larrazábal, *Implementación out-of-core para producto matriz-vector y transpuesta de matrices dispersas*, Conferencia Latinoamericana de Computación de Alto Rendimiento, Santa Marta, Colombia. Pags. 250–256. ISBN: 978–958–708–299–9, 2007.

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

[3] T.F. Chen and J.L. Baer, *A performance study of software and hardware data prefetching schemes*, In International Symposium on Computer architecture, Proceedings of the 21st annual international symposium on Computer Architecture, Chicago Ill, USA, Pages 223-232, 1994.

[4] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe and H. van der Vorst, *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, SIAM, Philadelphia, 2000.

[5] N. I. M. Gould, J. A. Scott and Y. Hu, *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, 2007.

[6] G. Karypis and V. Kumar, *A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs*, SIAM Journal on Scientific Computing, Vol. 20, No. 1, pp. 359392, 1999.

[7] D. Kroft, *Lockup-free instruction fetchprefetch cache organization*, In International Symposium on Computer architecture, Proceedings of the 8th annual symposium on Computer Architecture, Minneapolis, Minnesota USA, Pages 81-87, 1981.

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

[9] G. Larrazábal, *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, 2002.

[10] D.A. Patterson and J.L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, Third Edition, 2005.

[11] J. Reid, *TREESOLV, a Fortran package for solving large sets of linear finite element equations*, Report CSS 155. AERE Harwell, Harwell, U.K., 1984.

[12] J. Reid and J. Scott, *An out-of-core sparse Cholesky solver*, ACM Transactions on Mathematical Software (TOMS), Volume 36, Issue 2, Article No. 9, 2009.

[13] E. Rothberg and R. Schreiber, *Efficient Methods for Out-of-Core Sparse Cholesky Factorization*, SIAM Journal on Scientific Computing, Vol 21, Issue 1, pages: 129 - 144, 1999.

[14] A.J. Smith, *Cache Memories*, ACM Computing Surveys, Vol 14, Issue 3, pages: 473-530, 1982.

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