Implementación *out-of-core* para producto matriz-vector y transpuesta de matrices dispersas
Out-of-core implementation for sparse operations: matrix-vector product and transposition

Jorge Castellanos Germán Larrazábal
Centro Multidisciplinario de Visualización y Cómputo Científico (CEMVICC),
Facultad de Ciencias y Tecnología (FACYT), Tlf–Fax: +582418678243,
Universidad de Carabobo, Valencia, Venezuela.
{*jcasteld, glarraza*}*@uc.edu.ve*

Resumen

*Este artículo describe el desarrollo, implementación y evaluación de un soporte de memoria fuera de núcleo (*out-of-core*) para las operaciones básicas sobre matrices dispersas, es decir, producto matriz-vector y transpuesta de la matriz. La implementación está basada en un núcleo que optimiza el acceso a un* nodo *(fila/columna) desde y hacia el disco a través de una caché en memoria y un archivo en disco. El* nodo *se maneja en un formato comprimido con el objeto de minimizar el tiempo de acceso y el espacio de almacenamiento. Los resultados obtenidos con un procesador AMD Opteron 880™ muestran ahorros significativos en el uso de memoria con una baja penalización en tiempo de ejecución.*

Abstract

This paper describes the development, implementation and benchmarking for an out-of-core *support for the basic operations on sparse matrices, e. g. matrix-vector product and transposition. The core of this implementation is an* out-of-core *kernel that optimizes the accesses to a node (row/column) from and to disk through a cache and a temporal file. Each node is managed in a compressed format variation (CRS/CCS), minimizing in that way both, the access time and the storing space. The results obtained on a AMD Opteron 880™ processor showed significant memory savings at a low overhead cost in execution time.*

1. Introducción

La solución eficiente de sistemas lineales de ecuaciones y los cálculos con autovalores pueden estar limitados cuando el conjunto de matrices asociado no cabe en la memoria. Cuando los datos no caben en la memoria, entonces deben almacenarse en el disco duro. Los accesos de datos en disco duro son muy lentos (latencia + ancho de banda) en comparación con los accesos en memoria. Los algoritmos que se diseñan para obtener un buen comportamiento cuando sus estructuras son almacenadas en disco duro se denominan algoritmos *out-of-core*. Según [18] para tener un buen desempeño, el algoritmo *out-of-core* debe acceder los datos almacenados en disco en forma de grandes *bloque* contínuos, y una vez que un *bloque* se carga en memoria, este debe reutilizarse muchas veces. Se pueden encontrar ejemplos de aplicaciones *out-of-core* en: cómputo científico (modelado, simulación, etc.) [18] [13] [6], visualización científica [15] y manejo de bases de datos grandes [11].

El concepto *out-of-core* permite a los usuarios resolver grandes problemas eficientemente usando computadores económicos. El almacenamiento en disco duro es más económico que en memoria (DRAM) y su costo actual está en una relación de 1 a 80. Un algoritmo *out-of-core* ejecutándose sobre una máquina con memoria limitada da una mejor relación costo/beneficio que su equivalente algoritmo *in-core* corriendo sobre una máquina con suficiente memoria.

En concordancia con [2], los sistemas operativos actuales ofrecen un desempeño pobre cuando las estructuras de datos de una aplicación de cómputo numérico no cabe en la memoria principal porque el mecanismo provisto por el sistema de memoria virtual es ineficiente para las aplicaciones de cómputo y visualización científica cuando se emplean las políticas de paginado estándar. Como resultado, los programadores que desean resolver eficientemente problemas *out-of-core* se enfrentan típicamente con la tarea onerosa de reescribir una aplicación para usar explícitamente operacio-

nes de Entrada/Salida (es decir, lectura/escritura en disco).

En la práctica, los programadores que trabajan con aplicaciones *out-of-core*, escriben por lo general una versión separada del programa con llamadas explícitas de Entrada/Salida con el fin de obtener un desempeño razonable. La tarea de escribir la versión *out-of-core* de un programa es una actividad compleja – no se trata de insertar simplemente unas pocas sentencias de lectura y/o escritura a disco, sino que por lo general involucra restructuraciones del código, que en algunos casos puede tener un impacto negativo en la estabilidad numérica del algoritmo. Así la carga de escribir una segunda versión del algoritmo (y asegurar que trabaja correctamente) presenta una barrera importante en la solución de grandes problemas científicos [3].

El proyecto OOCORE [14] liderado por el grupo SCALAPACK de la Universidad de Tennessee en Knoxville desarrolló un solver *out-of-core* que está limitado a aplicaciones muy específicas con matrices del tipo denso. La ejecución de éste solver está restringida a clusters con más de 10 procesadores en una configuración de computo distribuído.

Se está en el proceso de desarrollar un soporte *out-of-core* completo para una biblioteca que resuelve sistemas lineales dispersos, la cual opera principalmente sobre matrices dispersas. Esta biblioteca se ha desarrollado por [8] en ANSI C, se conoce con el nombre de UCSparseLib y se ha compilado y ejecutado en arquitecturas x86 y sparc. En [4], [9] y [10] se ha usado la biblioteca UCSparseLib como herramienta de desarrollo. En tal sentido, se ha empezado el trabajo con las operaciones básicas. El producto matriz-vector es la operación más importante del núcleo *out-of-core* para los métodos de solución iterativos de sistemas lineales y su desempeño tiene un efecto importante en el tiempo de solución de un sistema lineal [5]. La transpuesta de la matriz es una primitiva clave en una amplia variedad de cómputos científicos. Por ejemplo, la transpuesta de la matriz es una una operación fundamental en la realización del producto de matrices y en el procesamiento de señales adaptativas [17].

De acuerdo con [19], la arquitectura de la jerarquía de memoria y las características del sistema de almacenamiento secundario influyen notablemente en el rendimiento de una aplicación *out-of-core*. Las mejoras que puedan introducir el uso de arreglos de discos (RAID) y la entrada/salida paralela serán consideradas en una etapa posterior de este trabajo, ya que el interés actual es la consolidación de una estructura de datos sencilla y homogénea que en conjunto con algoritmos eficientes permita minimizar el número de fallos en el acceso de las estructuras de datos almacenadas temporalmente en disco duro.

El presente artículo está organizado de la siguiente forma. En la sección §2, se presentan los formatos de almacenamiento para matrices dispersas, necesarios para comprender este trabajo. En la sección §3, 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 §5, se muestra en detalle la implementación del producto matriz-vector haciendo uso del del núcleo *out-of-core* implementado. Aún cuando no se explica en detalle la operación transpuesta de la matriz dispersa, se muestran resultados preliminares en la sección §6. En la seción §6, se presentan resultados que muestran la utilización de memoria y el tiempo de ejecución en las operaciones básicas 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 §7 se presentan las conclusiones del trabajo.

2. Almacenamiento de matrices dispersas

Como lo establece [1], 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 un 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 no-simétrica.

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 `val`, `ia`, `id` (mostrados en la figura 2).

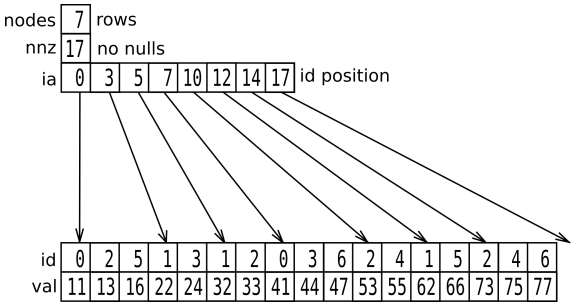

Figura 2. Formato de almacenamiento comprimido por fila (CRS).

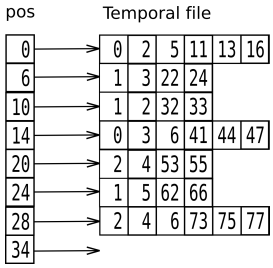

Figura 3. Formato de almacenamiento para el archivo temporal.

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 una ó mas filas (columnas) desde un archivo temporal en disco, se definió un formato de almacenamiento en disco según se ve en la figura 3. 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 una ó mas filas simultáneamente a la memoria, pués 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 que los valores, pues se considera que los primeros son del tipo entero mientras que los segundos son del tipo double.

3. El núcleo *out-of-core*

En la implementación del núcleo *out-of-core* para las operaciones básicas con matrices dispersas, se tomó como unidad básica de trabajo una fila completa (o una columna) de la matriz. En adelante esta unidad se denominará *nodo* para futuras referencias en este artículo. A diferencia de las matrices en formato denso, las matrices dispersas necesitan de una representación explícita de los elementos diferentes de cero y sus coordenadas. El tamaño de los *nodos* (filas/columnas), almacenados en un formato comprimido (CRS o CCS) es variable y depende del número de elementos diferentes de cero en cada uno de ellos.

Si usamos un formato comprimido (CRS o CCS) para el amacenamiento de la matriz dispersa necesitamos entonces dos vectores para almacenar cada *nodo*; un vector para almacenar las posiciones (`id`) dentro del *nodo* y otro para almacenar los valores (`val`) diferentes de cero (ver sección 2). 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 a la memoria un subconjunto de *nodos* consecutivos de la matriz que en adelante denominaremos *bloque*. Para permitir el almacenamiento de un *bloque* (o mas de uno), en memoria principal, el núcleo *out-of-core* crea una caché de correspondencia directa en la memoria principal. Ver en [12] el diseño de esta caché. La organización de esta caché saca provecho de los bits de direccionamiento de *bloque* en relación a la dirección del *nodo* (ver figura 4).

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.

Hasta este punto, se ha definido para las matrices dispersas un almacenamiento consistente tomando como unidad básica el *nodo*. Adicionalmente, se ha mostrado una forma consistente de dividir la matriz en *bloques* de igual tamaño. En la próxima sección se mostrará gráficamente cómo se almacenan los *bloques* en una estructura dinámica de caché .

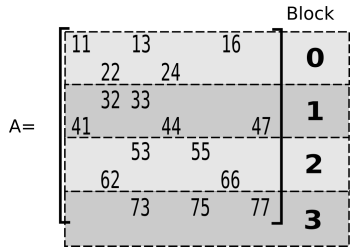

Figura 4. División en bloques de una matriz dispersa.

4. Operación del núcleo *out-of-core*

Para ocultar los detalles de implementación de la capa *out-of-core* al programador se han definido dos macros: una para operar sobre los *nodos* de la matriz cuando esta se viene almacenada en el formato CRS y otra para operar sobre *nodos* de la matriz cuando esta se lee en el formato 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 contine 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` vectores del tipo entero y del tipo double que contienen los índices del *nodo* y los valores no-cero 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 también un archivo temporal asociado a la matriz, como se comentó en la sección 3.

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 2). 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-cero 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* cuando se trabaje 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 al caso del producto matriz-vector cuando se leen secuencialmente los *nodos* de la matriz para multiplicar cada fila de la matriz por el vector (ver sección 5). Para la transpuesta de la matriz, se debe leer un conjunto de filas de la matriz antes de transponerlas.

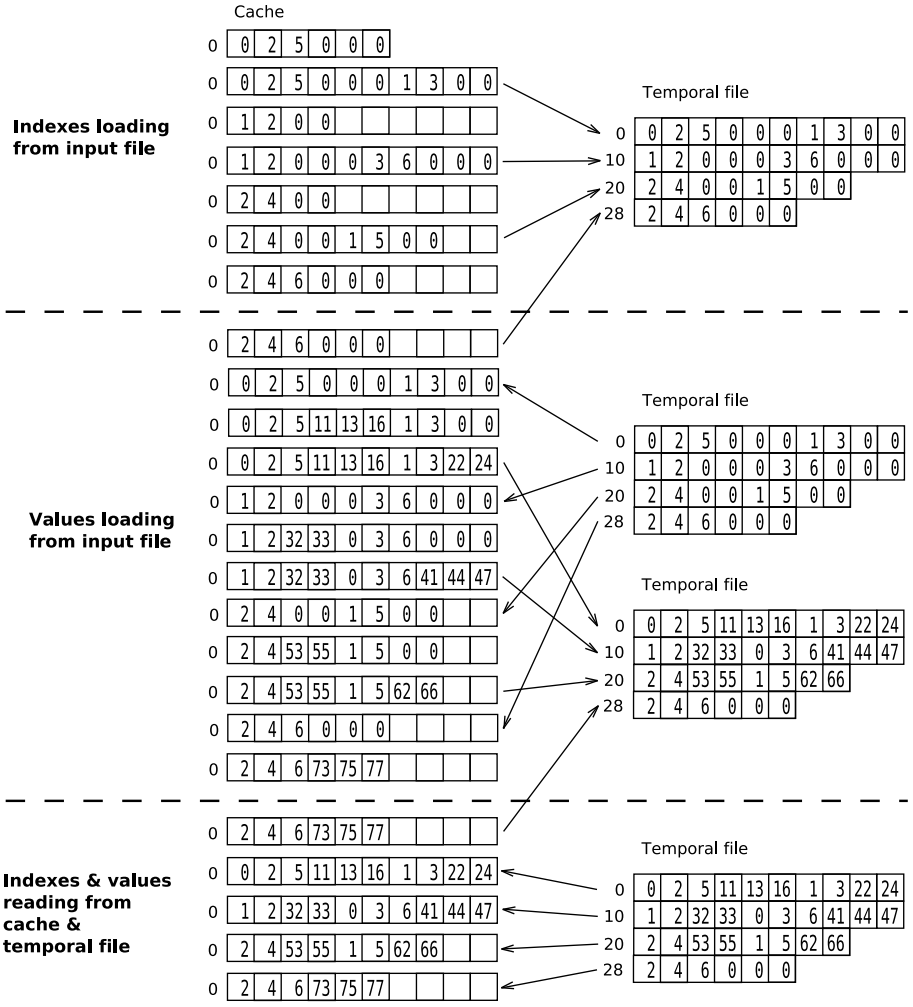

Figura 6. Operación del núcleo *out-of-core*.

5. Producto matriz-vector

El soporte *out-of-core* para el producto matriz-vector se habilita a través de una bandera de compilación llamada `OOC`. Esta bandera activa el núcleo *out-of-core* dentro de los códigos 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 las implementaciones *in-core* y *out-of-core*.

En la figura 7 se muestra el código para el producto matriz-vector; según se puede notar, el acceso a cada una de las filas de la matriz se hace a través de la macro `For_OOCMatrix_Row`. Está claro que la utilización del soporte *out-of-core* es transparente al programador a través de la macro `For_OOCMatrix_Row` porque ella oculta convenientemente las complejidades *out-of-core* al programador y le permite ejecutar los mismos códigos con la capa *out-of-core* desactivada.

```
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.

El producto matriz-vector solamente toma ventaja del principio de localidad espacial [16] de la caché de correspondencia directa del núcleo *out-of-core*. Cada vez que el algoritmo producto matriz-vector, trata de acceder a un *nodo* que no está en la 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 carga desde la memoria. Aunque el algoritmo del producto matríz-vector no hace uso del principio de localidad temporal de la caché (enunciado en [16]), el código de la figura 7 muestra buenos tiempos de ejecución. Esto es porque los *nodos* de la matriz de entrada se acceden en orden ascendente y el soporte *out-of-core* puede resolver automáticamente los fallos de caché, cargando un *bloque* de *nodos* concecutivos. De esta forma el efecto de los altos tiempos accesos al disco se minimizan cuando la tasa de aciertos es alta ($> 99\,\%$).

6. Resultados

Para evaluar el soporte *out-of-core*, todos los códigos se compilaron usando `gcc x86_64` versión 3,4,6 con la bandera de optimización `-O3`, ejecutándose sobre un procesador AMD Opteron 880™, sistema operativo `GNU/Linux`, kernel `2.6.9-42.0.10.ELsmp`, los datos se almacenan temporalmente en un disco SATA-2™, la memoria de intercambio está desabilitada (`swapoff`). El tiempo mostrado en las tablas comprende el tiempo de usuario y el de sistema.

Los resultados se presentan en dos tablas, la primera muestra los resultados para la operación producto matriz-vector y la segunda muestra los resultados corespondientes con la transpuesta de una matriz dispersa. Cada tabla tiene tres filas de datos que se corresponden con tres tamaños de matriz (1,0E+06; 2,5E+06; 5,0E+06). Las matrices de prueba se generaron a partir de la discretización por diferencias finitas de una ecuación $3D$ escalar de convección-difusión [7].

Cada tabla está dividida en cuatro secciones: **Matriz**, **In-core**, **Out-of-core** y **Resultados**. La primera sección se refiere a los detalles de la matriz dispersa, es decir, número de *nodos* ($Nodos$: filas en formato CRS) y total de elementos no-cero (Nnz). La segunda sección contiene los resultados obtenidos sin el soporte *out-of-core* activado, es decir, $memoria$ usada en bytes y tiempo medido en segundos. La tercera sección presenta los resultados en términos de ahorro de memoria ($ahorro$) y penalización 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.

Analizando los resultados en las dos tablas se puede apreciar que hubo ahorros significativos en el uso de memoria (70 %), con una penalización relativa en tiempo de ejecución menor del 40 % para la operación matriz-vector y de aproximadamente 100 % para la operación de transpuesta.

7. Conclusiones y trabajos futuros

A continuación se presentan las conclusiones más relevantes:

El núcleo *out-of-core* presentado en este trabajo soporta eficientemente las operaciones producto matriz-vector y transpuesta de matrices dispersas.

Para obtener un buen comportamiento de un soporte *out-of-core* consideramos que es importante seleccionar un formato apropiado para el almacenamiento temporal y una organización conveniente de la matriz dispersa. Estas dos características permiten explotar apropiadamente las ventajas de la caché.

La definición de una macro es un factor clave que facilita el desarrollo de la capa *out-of-core* y el uso del núcleo *out-of-core* por parte del programador de aplicaciones.

Como posibles trabajos futuros tenemos:

Incorporar mejoras en el algoritmo de la operación transpuesta 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: Choleski, LDL$^{\mathrm{T}}$, y LU.

Implementación de la capa *out-of-core* para otros métodos de solución de sistemas lineales dispersos como los métodos iterativos y multinivel algebráico.

Agradecimientos.

Este trabajo forma parte de los proyectos CDCH-UC N° 2004-002 y CDCH-UC N° 2004-011. Se agradece especialmente a José Luis Ramírez por su contribución en la evaluación del soporte *out-of-core*.

Referencias

[1] 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.

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

[3] A. Demke B., T. C. Mowry, and O. Krieger. Compiler-based I/O prefetching for out-of-core applications. *ACM Transactions on Computer Systems*, 19(2):111–170, January 2001.

[4] F. Hernández, J. Castillo, and G. Larrazábal. Large sparse linear systems arising from mimetic discretization. *Journal of Computers and Mathematics with Applications*, 53(1):1–11, 2007.

Tabla 1. Uso de memoria y tiempo de ejecución para el producto matriz-vector.

Matriz		In-core		Out-of-core		Resultados %	
Nodos	*Nnz*	*Memoria*	*tiempo*	*memoria*	*tiempo*	*ahorro*	*ovhd*
1.000.000	6.940.000	139.280.096	0,107	44.076.476	0,127	31,65	18,69
2.500.000	17.380.000	348.560.096	0,258	110.123.380	0,349	31,59	35,27
5.000.000	34.780.000	697.360.096	0,540	220.168.436	0,677	31,57	25,37

Tabla 2. Uso de memoria y tiempo de ejecución para la transpuesta de la matriz.

Matriz		In-core		Out-of-core		Resultados %	
Nodos	*Nnz*	*Memoria*	*tiempo*	*memoria*	*tiempo*	*ahorro*	*ovhd*
1.000.000	6.940.000	250.560.192	1,072	73.751.624	2,137	29,43	99,35
2.500.000	17.380.000	627.120.192	3,183	175.765.912	6,544	28,03	105,59
5.000.000	34.780.000	1.254.720.192	5,435	363.029.848	11,309	28,93	108,08

[5] H. Kotakemori, H. Hasegawa, T. Kajiyama, A. Nukada, R. Suda, and A. Nishida. Performance evaluation of parallel sparse matrix-vector products on sgi altix3700. *Proceedings of the First International Workshop on OpenMP (IWOMP2005), Lecture Notes in Computer Science, Springer, in press*, 2005.

[6] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, C.-C. Lam, J. Ramanujam, P. Sadayappan, and V. Choppella. Efficient synthesis of out-of-core algorithms using a nonlinear optimization solver. *J. Parallel Distrib. Comput.*, 66(5):659–673, 2006.

[7] 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.

[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 and J. Martínez. Wavelet-based spai preconditioner using local dropping. *Journal of Applied Numerical Mathematics*, 47:200–214, 2003.

[10] G. Larrazábal, C. Torres, and J. Castillo. An efficient and robust algorithm for 2d stratified fluid flow calculations. *Journal of Mathematics and Computer in Simulation*, 73:493–502, 2006.

[11] E. Masciari, G. Raimondo, C. Pizzuti, and D. Talia. Using an out-of-core technique for clustering large data sets. In *DEXA '01: Proceedings of the 12th International Workshop on Database and Expert Systems Applications*, page 133, Washington, DC, USA, 2001. IEEE Computer Society.

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

[13] E. Rabani and S. Toledo. Out-of-core svd and qr decompositions. *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing*.

[14] B. Samuel and E. D'Azevedo. Benchmarking oocore, an out-of-core matrix solver. *Oak Ridge National Laboratory*, 2004.

[15] C. Silva, Y. Chiang, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics, 2002.

[16] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

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

[18] 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.

[19] J. S. Vitter. External memory algorithms and data structures. *External memory algorithms, American Mathematical Society, ISBN: 0-8218-1184-3*, pages 1–38, 1999.