Estudio y evaluación de eficiencia de operaciones con matrices dispersas (sparse) en sistemas distribuidos

Memoria final del proyecto

Alumno : Pierre Raimbaud

Especialidad : Ingeniería del software

Director : María Luisa Gil Gómez

Fecha : 20/06/2014

Objetivo general

Estudiar y evaluar, en sistemas distribuidos, la eficiencia de operaciones con matrices dispersas (sparse), sabiendo la forma en la cual están almacenadas y los algoritmos usados, y proponer, en la medida de lo posible, mejoras o nuevas soluciones.

Objetivos del proyecto

- aprender cuales son los modos de almacenamientos de las matrices dispersas y como se accede a sus datos
- comparar el ahorro de espacio de memoria según los tipos de almacenamiento
- probar operaciones algebraicas sobre las matrices *sparse* con programas de ejemplo, en C, Java y C++
- descubrir puntos de conflictos en el entorno de operar con estas matrices
- estudiar la escalabilidad y el paralelismo en los algoritmos usados en estos cálculos
- proponer mejoras o soluciones a estos problemas detectados, en la medida de lo posible

Indice

I – Formatos de almacenamiento

0) **Introducción a los formatos** ……7

1) **Formato DNS**……10

2) **Formato COO**……10

3) **Formato CSR**……11

4) **Formato CSC**……12

5) **Formato MSR**……12

6) **Formato Poly**……13

7) **Formato PolySym**……14

8) **Formato PolySymFix**……15

9) **Formato ELL**……15

10) **Formato DIA**……16

11) **Formato DiaSym**……17

12) **Formato BDN**……17

13) **Formato BSR**……18

14) **Conclusión**……19

II – Operaciones algebraicas

0) **Introducción a las operaciones**……20

1) **Matriz por escalar**……20

2) **Lectura y escritura de valor**……20

3) **Matriz por vector**……23

III – Eficiencia de formato

0) Introducción a los diferentes criterios de eficiencia……………………………..25

1) En términos de memoria (espacio ocupado)………………………………….….25

2) En términos de cálculo (operaciones teóricas)……………………………..…...26

3) En términos de tiempo de cálculo (matriz por vector)…………………….…...27

IV – Matrices de ejemplo, paralelismo y pruebas

0) Introducción al paralelismo…………………………………………………………28

1) Presentación de los bancos de matrices sparse reales…………………………28

2) Metodología seguida……………………………………………………………….29

3) Resultados de cálculos matriz por vector……………………………………….29

4) Conclusiones………………………………………………………………………….31

V – Bibliografía 32

1) Matrices dispersas...32

2) Operaciones sobre las matrices dispersas..32

3) Lenguajes de programación y las librerías..32

4) Estructuras de datos y algoritmos...33

5) Cálculos con paralelismo...34

6) Tesis..34

Introducción

Como lo sabemos, cada día la informática tiene más importancia en el mundo actual. Pero siempre tenemos que acordarnos que la informática casi no es nada sin su aplicación en otras disciplinas, otros entornos. Existen muchos trabajos de ingenieros informáticos que se enfrentan a problemas multidisciplinarios, estando justamente en equipos multidisciplinarios.

De hecho, el lado virtual de la informática, le da éxito para disciplinas que necesitan simulaciones, antes de crear físicamente algo, por los riesgos posibles o también por la voluntad de buscar posibles optimizaciones. Algunas de esas disciplinas manipulan una gran cantidad de información, que además no tienen mucha regularidad (mal ordenada) o, aunque puedan seguir unos patrones, resultan muy dispersas. Por ejemplo, mecánica de las estructuras, estudio de la dinámica de los fluidos, tratamiento de las imágenes o de señales etc.

En esos casos se suele almacenar los datos en matrices. Pero se suelen llamar matrices sparse (dispersas) porque sus tamaños (fila por columna) son mucho más grande que el numeró de datos que contienen.

De esa constatación, se saca dos objetivos de mejora : almacenar los valores en formatos de matrices lo menos voluminoso posible (optimización al nivel de memoria) y de hecho, optimización al nivel de tiempo de cálculo, porque solamente se harán cálculos útiles (no se hará multiplicaciones por cero).

Entonces, primero se debe buscar formatos que permiten almacenar esas matrices sparse pero guardando la exactitud de los datos. Segundo se debe implementar las operaciones clásicas pensando de nuevo los algoritmos de cálculos : en efecto ya no se puede aplicar directamente un algoritmo de cálculo pensado con el formato “matemático”. Además no se debe olvidar que todas las matrices sparse no son iguales. Algunas tienen más elementos que otras, algunas son más dispersas, otras tienen características especificas (matrices cuadradas por ejemplo) o sus valores pueden seguir un patrón conocido.

En la primera parte de ese documento se explicará los distintos formatos que han sido estudiados. Primero, se dirá sobre qué matrices se puede aplicar, después cómo se almacenan los valores de la matriz y también si contiene algunas otras características que las mencionadas antes (tipo de matriz : por ejemplo, matriz cuadrada y con valores solamente por diagonales). Luego si necesario se mostrará un ejemplo sencillo (matriz de tamaño inferior a 5*5) y también explicaciones sobre la estructura de datos. Por fin se dará sus ventajas/inconvenientes.

En la segunda parte se explicarán las operaciones algebraicas que se han sido estudiadas sobre esas matrices y algunos algoritmos recurrentes, implementados en los formatos vistos anteriormente para que se sepa lo que implica utilizar formatos que, acuérdense, son distintos del formato matemático.

En la tercera parte se tratará de la eficiencia de los formatos. Primero en términos

de memoria, es decir, para un formato, comparándolo al espacio 'clásico' necesario. Luego, en términos de cálculo, es decir, mirando en cuanto a las operaciones, cuantas indirecciones se necesitan etc. Por fin, en términos de tiempo de cálculo, con las implementaciones que se han hecho de los formatos, probándolos con varias matrices.

La cuarta parte explicará las pruebas y las conclusiones que salieron de los estudios sobre algunas matrices sparse (reales o no) y utilizando unos sistemas distribuidos.

Por fin, se dará la bibliografía completa utilizada para ese proyecto.

I - Formatos de almacenamiento

0) Introducción a los formatos

Primero, recordar que en este proyecto el estudio está totalmente enfocado en matrices sparse. Entonces el carácter sparse de las matrices será el punto de entrada de la reflexión que daremos a continuación. No obstante, estas matrices, aunque compartan un carácter común, pueden ser muy distintas una a la otra. En la literatura se puede encontrar varias definiciones de una matriz sparse pero básicamente es la siguiente :

Definición :

Una matriz sparse (dispersa en castellano, *creuse* en francés) es una matriz que contiene como valores muchos ceros, y eso suficientemente para que se pueda aprovechar de ese carácter sparse al nivel de espacio de memoria y de tiempo cálculo.

Luego debemos distinguir dos grandes tipos de matrices sparse : las que siguen un patrón, una estructura regular, y las que tienen valores de manera dispersa (refiriéndose a las posiciones de los valores no nulos en la matriz), siguiendo ningún patrón conocido o explotable. Por lo tanto, una matriz, según si estructurada o no, puede ser almacenada en unos formatos o no. En otros términos, una matriz que no pertenece a ninguno grupo de estructura conocida, podrá solamente ser almacenada utilizando formatos genéricos, y por lo tanto con menos eficiencia que si fuese una matriz siguiendo un patrón conocido, un formato especifico aprovechando más del carácter sparse de una matriz que los genéricos/comunes.

A continuación se explicará distintos formatos de la literatura de la informática de las matrices sparse. Cada uno tiene sus ventajas y inconvenientes, y se puede usar para un tipo de matriz u otro. Su utilización puede ser más o menos frecuente en *“el mundo del cálculo sparse”.*

Más abajo, les daré unas definiciones generales sobre las matrices y sus estructuras. Unas pueden parecer sencillas para uno que sabe de matemática, pero mejor ya explicarlo para que se planteen (y quedan sin duda) desde el inicio del documento esos conceptos que se utilizaran a continuación.

Matriz cuadrada :

Una matriz clásica se suele definir por su tamaño, diciendo su número de filas y él de columnas. Se dice por ejemplo : una matriz m*n, lo cual es una matriz que contiene m filas y n columnas. Su forma es *rectangular*. Cuando m = n, la matriz es una matriz cuadrada (n*n) : tiene n^2 valores y su forma es *cuadrada*. Unas ventajas de esas matrices es que tienen una diagonal de valores (lo cual sirve para algunos cálculos específicos) y que para una multiplicación de dos matrices cuadradas n*n, la matriz resultado será también cuadrada n*n. Acuérdense que la multiplicación de dos matrices sigue el formato : (m*n) * (n*p) = (m*p), en otros términos, el número de columnas de la matriz de la izquierda debe

ser igual al número de filas de la matriz de la derecha, sino la multiplicación no se permite - además el producto de dos matrices no es conmutativo.

1	2	3
4	5	6
7	8	9

Figura 1 : Matriz cuadrada 3 * 3

Matriz triangular :

Una matriz triangular es una matriz cuadrada que soló tiene valores no nulos por encimo o por abajo de la diagonal (diagonal incluida). En otros términos, mirando a la matriz representada de manera cuadrada, se ve una forma de triangulo (mirando a los valores no nulos) por la parte inferior o superior de la matriz (refiriéndose a la diagonal de la matriz). De hecho, se suele hablar de matriz triangular superior y matriz triangular inferior según la posición de su triangulo.

1	2	3
0	5	6
0	0	9

Figura 2 : Matriz triangular superior

Matriz diagonal :

Una matriz diagonal es una matriz cuadrada que soló tiene valores no nulos en su diagonal. Tiene muchas ventajas de cálculos : pocos cálculos (muchos valores nulos), algoritmos simplificados (por ejemplo, cálculo del determinante) etc. y pueden imaginarse que su almacenamiento será bastante simplificado por su carácter diagonal.

1	0	0
0	5	0
0	0	9

Figura 3 : Matriz diagonal

Matriz banda :

Una matriz banda es una matriz cuadrada que tiene solamente valores no nulos en su diagonal y en las “diagonales” en sus costados. Básicamente se define para esas matrices un número de “diagonales” superiores y inferiores. De hecho, si uno de los dos números es igual a cero, la matriz es triangular, y si los dos son iguales a ceros, estamos en el caso de una matriz diagonal.

1	2	0	0
0	6	7	0
9	10	11	0
0	14	15	16

Figura 4 : Matriz banda

Matriz simétrica :

Una matriz simétrica es una matriz cuadrada que tiene sus valores ubicados de manera que sea simétrica por el eje formado por su diagonal. En términos matemáticos, se dice que una matriz es simétrica cuando esa matriz es igual a su matriz transpuesta. En efecto, una matriz simétrica A, sigue la regla, usando los índices i y j para respectivamente las líneas y las columnas: A(i,j) = A(j,i), y como la matriz transpuesta de una matriz es la matriz que tiene los valores de A transfiriéndolos de manera que A(i,j) coja el valor de A(j,i) y vice versa...

1	2	0	13
2	6	7	0
0	7	11	0
13	0	0	16

Figura 5 : Matriz simétrica

1) Formato Denso (DNS)

Uso :

Se puede usar para cualquier matriz. Representación matemática de una matriz (para visualizar “bien” la matriz).

El primer formato que se puede usar para almacenar una matriz es el formato denso (DNS). Se puede considerar como el formato estándar, porque, básicamente, es la representación matemática de una matriz.

Con ese formato, para una matriz de m filas y n columnas se necesita una tabla de dimensión 2 ([m * n]). Se guardan todos los ceros, así que en ese caso no se aprovecha para nada del carácter sparse de una matriz. Lo utilizaremos sobre todo a modo de comparación con los demás, como linea base (porque los demás si que aprovecharan del carácter sparse de las matrices).

1	7	0	0
0	2	8	0
5	0	3	9
0	6	0	4

Figura 6 : Matriz formato DNS

Ventajas/Inconvenientes :

Su ventaja principal es que como sigue la representación matemática de las matrices, se puede leer (y entender) directamente (tanto al nivel de un ser humano como de un ordenador) y los algoritmos clásicos de cálculos se pueden utilizar directamente. Además se puede almacenar cualquiera matriz. Pero su inconveniente es que no permite ningún ahorro tanto al nivel de memoria como al nivel de cálculo. Las matrices muy grandes (billones de valores) casi no se pueden representar en ese formato tan el uso de memoria es importante.

2) Formato por coordenadas (COO)

Uso :

Se puede usar para cualquier matriz.

El segundo formato que se puede usar para almacenar una matriz es el formato por coordenadas (COO).

Al contrario del formato DNS, los ceros no se guardan y no se utiliza tablas de dimensión 2, sino 3 tablas de dimensión 1 : una de valores, una que da el indice de fila de un valor (PtrRow) y la ultima el indice de columna de la misma valor (PtrCol).

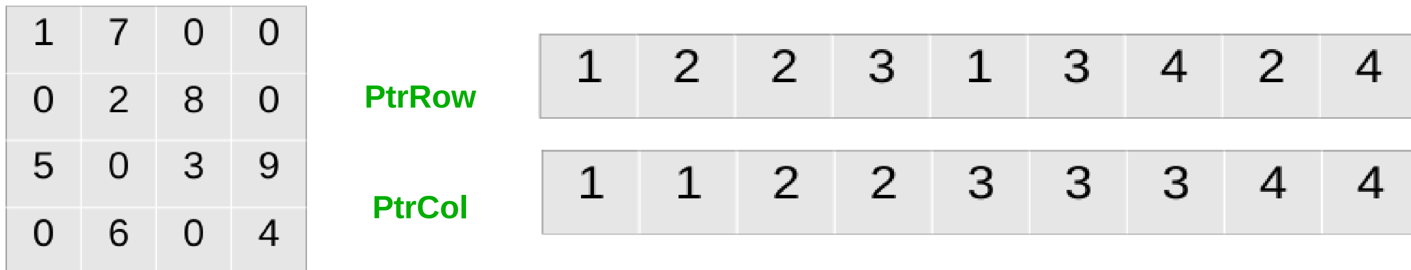

Figura 7 : Matriz formato COO

Ventajas/Inconvenientes :

Su ventaja principal es que se utiliza tablas de dimensión 2, las cuales siempre cuestan de usar. Pero el espacio en memoria todavía queda grande, ya que se utiliza tres tablas. Por lo menos esa vez si que se aprovecha el carácter sparse de la matriz ; de hecho para una matriz muy grande (billones de filas y de columnas) pero con muy pocos valores, ese formato será muy útil porque muy comprimido.

3) Formato CSR (Compressed Sparse Row)

Uso :

Se puede usar para cualquier matriz.

La idea de este formato es almacenar los valores por filas, sin los ceros. Se necesitan para eso tres tablas de dimensión 1 : una que contiene los valores no nulos (data), una que contiene el indice de columna para cada de esos elementos (indices) y una ultima tabla más pequeña (tamaño m+1) que permite saber a qué fila pertenece los valores (ptr) (los valores son almacenados leyendo de izquierda a derecha, fila por fila (bajando)).

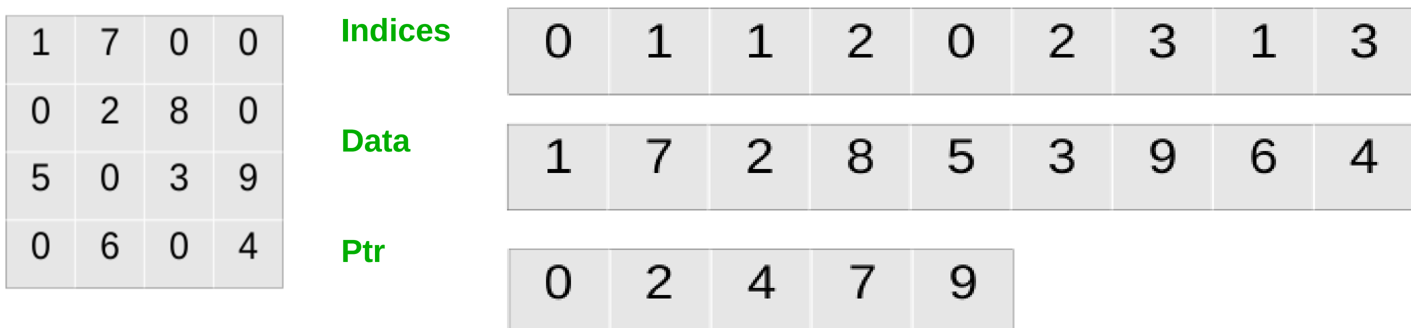

Figura 8 : Matriz formato CSR

Ventajas/Inconvenientes :

La ventaja principal de ese formato es que si lo comparamos al formato anterior, el COO, el uso de memoria es menor. En efecto, con el COO, se utilizaba tres tablas de tamaño igual al número de elementos no nulos de la matriz, mientras que aquí hay dos tablas de ese tamaño pero el tercero queda más pequeño, solamente del número de filas más uno. Cuidado, todo es cierto para una matriz que tiene menos valores que su número de filas, sino se utilizará más espacio de memoria con ese formato. De hecho, de manera general, el grado de sparsidad de una matriz nos indica su mayor o menor ocupación de espacio.

4) Formato CSC (Compressed Sparse Column)

Uso :

Se puede usar para cualquier matriz.

La idea de este formato es la misma que con el anterior CSR, pero por columnas, i.e almacenar los valores (no nulos) por columnas. Se necesitan para eso tres tablas de dimensión 1: una que contiene los valores no nulos, una que contiene el indice de fila para cada de estos elementos (valores) y una ultima tabla más pequeña (tamaño m+1) que permite saber a cual columna pertenece los valores (los valores se almacenan leyendo de arriba a abajo, columna por columna (yendo hacia la derecha).

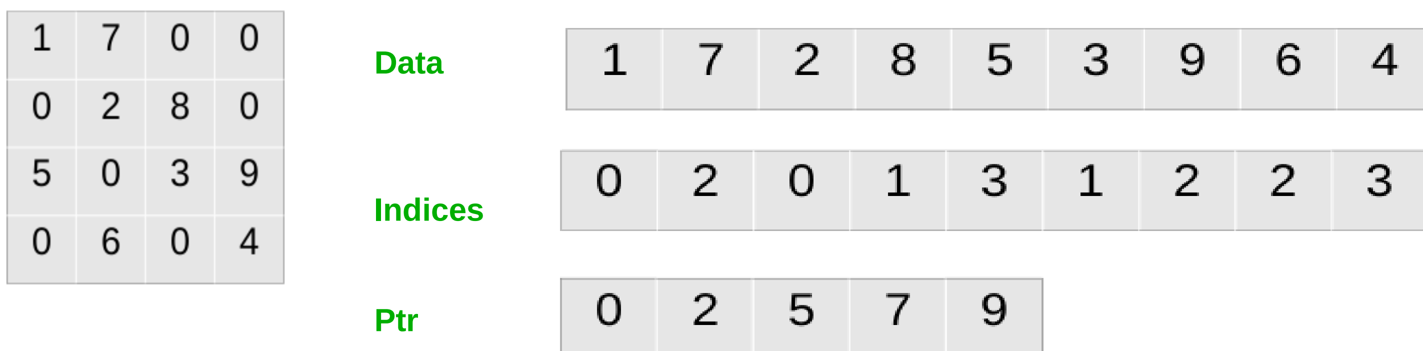

Figura 9 : Matriz formato CSC

Ventajas/Inconvenientes :

La ventaja de ese formato es la misma que con el formato CSR : el uso de memoria puede ser menor que con el COO, cuando la matriz tenga menos valores que su número de columnas.

5) Formato MSR (Modified Compressed Sparse Row)

Uso :

Se puede usar para las matrices triangulares.

Ese formato es una variante del formato CSR para matrices triangulares. Primero se almacena la diagonal en una tabla de dimensión 1 (Data). En esa misma tabla se guarda en seguida los valores por fila (sin los ceros).

En otra tabla (PtrIndices), del mismo tamaño (número de valores +1) se guarda : primero los indices (punteros) de fila y, en seguida, los indices de columna para cada valor que no esté en la diagonal.

Implementando el formato CSC después del CSR, me di cuenta que eran casi totalmente similares, entonces decidí no implementar el formato MSC, lo cual es lo mismo que el formato MSR pero por columna.

1	9		21	10	12
	2	19			
		3	15	14	22
			4	20	
				5	17
					6

PtrIndices	8	12	13	16	17	18	18	2	4	5	6	3	4	5	6	5	6
Data	1	2	3	4	5	6		9	21	10	12	19	15	14	22	20	17

Figura 10 : Matriz formato MSR

Ventajas/Inconvenientes :

La ventaja de ese formato es el uso de memoria que es mucho menor que con los formatos anteriores (un tercio menos que con el COO). Se utiliza solamente 2 tablas de tamaño igual al (número de valores + 1).

6) Formato Poly (Polígono)

Uso :

Se puede utilizar para cualquiera matriz.

Ese formato almacena los valores por fila, basándose en la idea que la matriz almacenada tiene una forma de polígono más o menos cerca de la diagonal. Se utiliza tres tablas de dimensión 1.

Se almacena los valores en una primera tabla (Data). En otra tabla de tamaño igual

al número de fila, se almacena los inicios de fila, i.e el desajuste en comparación a 0 (inicio de fila) (ifa). Y en la ultima tabla, de tamaño igual al número de fila + 1, los indices/punteros de fila (para saber cuales elementos pertenecen a cual fila) (ia).

Figura 11 : Matriz formato Poly

Ventajas/Inconvenientes :

La ventaja de ese formato es el uso de memoria que puede ser mucho menor que con algunos formatos anteriores, cuando la matriz tenga muchos valores “por la derecha”, ya que la única tabla “grande” va a ser la de los valores mientras que las dos demás solamente van a ocupar el espacio igual al número de fila (+1). Pero su desventaja principal es que se almacenan todavía algunos ceros inútiles.

7) Formato PolySym (Polígono Simétrico)

Uso :

Se puede utilizar para matrices simétricas.

Ese formato se utiliza para matrices simétricas : se almacena los valores por fila, con la misma manera que el formato Poly, pero esa vez solamente la parte superior de la matriz (todavía con los ceros “intermedios”, i.e entre dos valores no nulos) : la parte inferior se podrá deducir por simetría.

Se almacena los valores en una tabla (data) (tamaño número de valores) y en otra tabla se almacena los indices/punteros de fila (ptr) (tamaño número de fila). Se ahorra la otra tabla de “desajuste en comparación a 0 (inicio de fila)”, ya que sabemos que lo que se almacena para cada fila empieza al nivel de la diagonal.

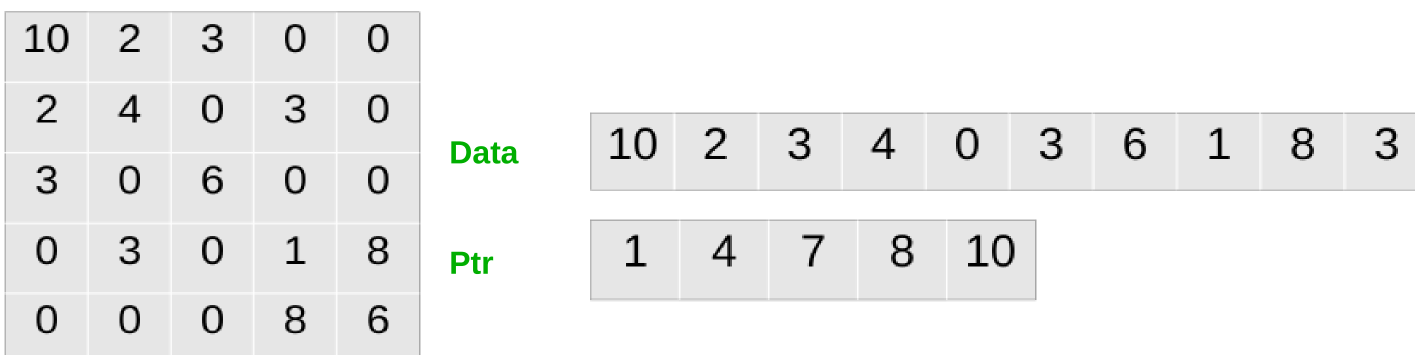

Figura 12 : Matriz formato PolySym

Ventajas/Inconvenientes :

La ventaja de ese formato es el uso de memoria que es menor que con el formato Poly porque se usa una tabla menos. Su desventaja principal es que todavía se almacenan algunos ceros inútiles.

8) Formato PolySymFix (Polígono Simétrico Fijo)

Uso :

Se puede utilizar para matrices simétricas y con anchura de '"sub-diagonales" fija (no variable como con el formato Poly).

Ese formato almacena los valores como el formato PolySym. Pero ese formato se utiliza cuando la anchura de las "bandas del polígono" sea fija : por ejemplo, la diagonal y dos '"sub-diagonales" (por la simetría, por los dos lados). El espacio usado es muy reducido ya que se utiliza solamente una tabla.

Se almacena los valores en la tabla (tamaño número de valores), y además, a parte, la anchura del polígono (un entero).

Ventajas/Inconvenientes :

La ventaja principal de ese formato es el ahorro de espacio importante. Pero solamente se puede utilizar cuando la matriz tenga una estructura bastante especifica (desventaja).

9) Formato ELL (ELLPACK)

Uso :

Se puede utilizar para cualquiera matriz, pero es útil solamente si, dando k = la anchura máxima de una linea quitando los ceros y m = m el número de fila por una matriz $m * n$, k << m (que no haya ni una fila con muchos valores, sino el almacenamiento no sería eficiente, porque quedarían muchos espacios libres inútiles (las * en el ejemplo más abajo)).

Ese formato almacena los valores por fila, quitando los ceros, utilizando dos tablas de dimensión 2.

Antes de poder utilizar ese formato, se tiene que saber/calcular la anchura máxima de linea quitando los ceros (el máximo entre todas las lineas de la matriz). Ese número k debe ser inferior a m (número de fila por una matriz $m * n$) para que sea útil utilizar ese formato, i.e la matriz no debe tener una fila que contiene m valores.

Sabiendo eso, se almacena los valores usando las dos tablas de tamaño $m * k$. La primera contiene los valores y la segunda el indice de columna para cada valor.

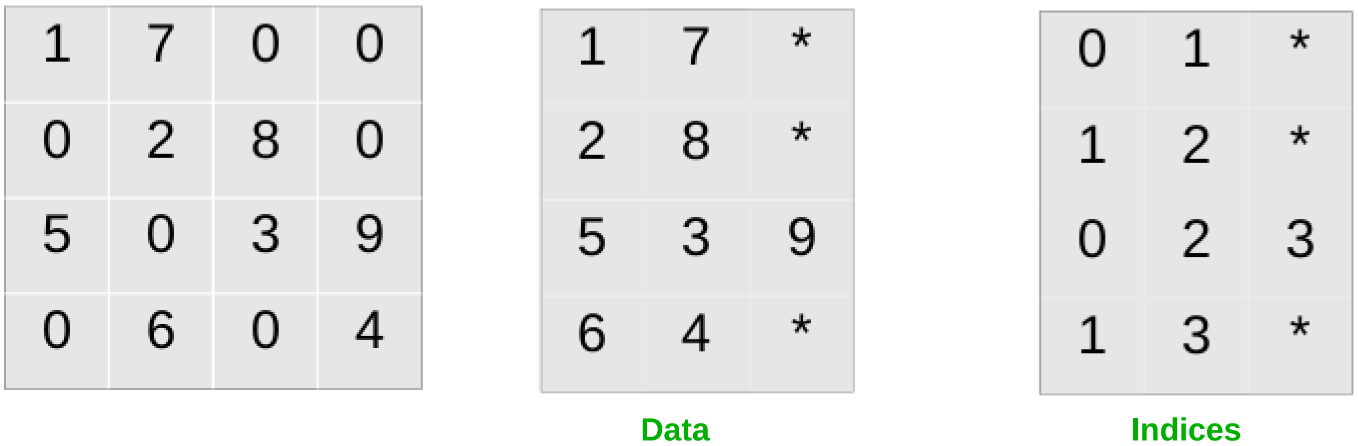

Figura 13 : Matriz formato ELL (ELLPACK)

Ventajas/Inconvenientes :

La ventaja principal de ese formato es la flexibilidad del formato, en otros términos la posibilidad de poder casi siempre almacenar una matriz con ese formato, y eso con un ahorro de espacio que aun puede ser bastante interesante. Su inconveniente principal es que se utiliza tablas de dimensión 2.

10) Formato DIA (Diagonal)

Uso :

Se puede utilizar para matrices que tienen valores no nulos solamente por la diagonal o las “sub-diagonales”.

Ese formato almacena los valores en una tabla de dimensión 2 (*data* en el ejemplo), de tamaño (número de diagonales * número de fila) de la manera siguiente : cada diagonal/sub-diagonal se almacena en una columna. Y se utiliza la otra tabla (*offsets* en el ejemplo) de tamaño (número de diagonales), para saber la posición, en comparación con la diagonal, de la diagonal/subdiagonal almacenada.

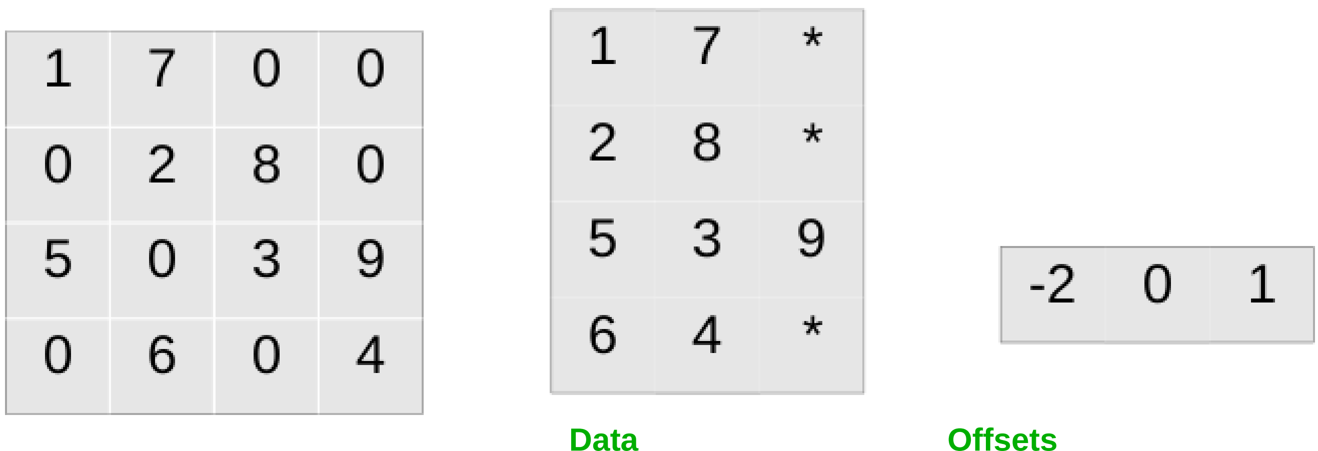

Figura 14 : Matriz formato DIA

Ventajas/Inconvenientes :

La ventaja principal de ese formato es el ahorro de espacio que es muy importante ya que casi solamente se almacenan valores útiles (si las diagonales no tienen muchos ceros, mientras la matriz está muy grande y entonces con muchos ceros afuera de las diagonales). Además la otra tabla es de tamaño tan pequeño que casi no ocupa espacio en memoria. Su desventaja es que la primera tabla es una tabla de dimensión 2 y no de dimensión 1.

11) Formato DiaSym (Diagonal y Simetría)

Uso :

Se puede utilizar para matrices que siguen el patrón de DIA pero que además son matrices simétricas.

Ese formato almacena los valores fila por fila, en una tabla de tamaño (número de valores), y usa una otra tabla de tamaño (m = número de fila), que contiene los indices de fila, la cual sirve para saber de cual fila es el valor.

Ventajas/Inconvenientes :

La ventaja principal de ese formato es que se quita la desventaja principal del formato DIA ya que la primera tabla es una tabla de dimensión 1 y no 2 esa vez.

12) Formato BDN (Linpack Banded)

Uso :

Se puede utilizar para matrices que tienen valores no nulos solamente por la diagonal o por las “sub-diagonales”. (i.e que siguen el mismo patrón que para el formato DIA).

Ese formato se utiliza para matrices que siguen el patrón de DIA también, pero la manera de almacenar los valores no es la misma. Además se supone que no hay ceros en las diagonales/sub diagonales, salvo “intermedios” (entonces, no los extremos de la diagonal/sub diagonal).

Hay solamente una tabla (m * número de diagonales) y ya no hay una otra de indices, porque como no hay estos ceros que expliqué más arriba, se sabe cual subdiagonal/diagonal está en cada columna de la tabla del formato BDN, según el número de ceros (o estrella, o casilla vacila) que empiezan o acaban esa columna.

1	9		21	10	12
	2	19			
		3	15	14	22
			4	20	
				5	17
					6

	1	21	16
9	2	19	15
23	3	0	27
20	4	22	12
10	5	25	
11	6		

Figura 15 : Matriz formato BDN

Ventajas/Inconvenientes :

La ventaja principal de ese formato es que se utiliza solamente una tabla y nada más, toda la información la tenemos en un lugar. Pero otra vez la desventaja es que es una tabla de dimensión 2.

13) Formato BSR (Block Sparse Row)

Uso :

Se puede utilizar para matrices por bloques (todos de tamaño iguales).

Para acabar, explicaré ese formato que me pareció interesante porque es muy distinto de los anteriores. Eso es normal ya que se puede utilizar solamente para matrices que siguen un patrón muy especifico : matrices por bloques (valores no nulos formando bloques todos iguales y lo demás, ceros).

1	10			19	14
9	2			20	11
		3	15		
		23	4		
				5	22
				21	6

Figura 16 : Matriz por bloques

Se construye una lista de mini-tablas, las cuales contienen cada una un bloque de valores. Esos bloques son cogidos fila por fila Además se tiene que guardar dos otros tipos de informaciones en dos tablas de tamaño igual al número de bloques : la primera tabla (JA) indica el inicio, de indice de columna, del bloque en la matriz (high-left corner), y la segunda los indices/punteros de fila (IA).

1	10	19	14	3	15	5	22
9	2	20	11	25	4	21	6

Data

1	3	4	5

IA

1	5	3	5

JA

Figura 17 : Matriz formato BSR

Ventajas/Inconvenientes :

La ventaja principal de ese formato es el ahorro de memoria muy importante para esas matrices muy especiales, ya que a parte de los bloques con los valores (pero siempre se deben almacenar los valores entonces...), solamente se necesita dos pequeñas tablas, las cuales son de tamaño muy pequeño aunque sea grande la matriz porque se basan solamente en el número de bloques de la matriz.

14) Conclusiones

Esos formatos les implementé todos en Java para probarlos : permitir la creación de unas matrices sencillas, leyendo los datos a partir de ficheros .txt, implementar una función para devolver la matriz en formato DNS a partir de la matriz almacenada en el formato de su “Class” - eso también permitiendo de verificar que la creación de la matriz se ha hecho bien. A continuación, implementé algunas operaciones algebraicas, tema que trataré justamente en la segunda parte.

Pero, al momento, en la cuarta parte, de implementar código paralelo, utilizando el lenguaje C++ y los pragma de OpenMP, tuve que elegir cuales formatos implementar en C++. Elegí los formatos ELL y CSR. Por cuales motivos ?

Primero esos formatos, aunque no permitan tanto ahorro de espacio como por ejemplo el formato DIA o BSR, tienen la ventaja de ser genéricos y aceptar el almacenamiento de cualquiera matriz sparse. Entonces eso permite de poder la posibilidad de hacer pruebas sobre más ejemplos de matrices reales – porque no se necesita matrices con datos ubicadas de manera particular (diagonal etc.).

Segundo, la información de una matriz en los bases de datos en Internet suele ser un fichero .txt conteniendo los datos por coordenadas (formato COO). Esos formatos tienen la ventaja que no es complicado llegar a obtener la matriz almacenada en sus formatos yendo del formato COO o DNS.

Tercero y para acabar, estudiar y desarrollar código nuevo tratando de esos formatos es útil porque son formatos ya bastante utilizados y entonces ya existe varias librerías quienes implementan funcionalidades sobre o usando esos formatos de matriz.

II – Operaciones algebraicas

0) Introducción a las operaciones

A la hora de haber implementado para las matrices sparse los formatos explicados más arriba, se ha de explicar qué operaciones traté. Explicaré primero el producto matriz por escalar, después la lectura y escritura (modificación) de valor en una matriz y al final, pero la operación la más importante, hablaré del producto matriz por vector.

Esas tres operaciones las implementé en Java para todos los formatos, entonces hablaré de los algoritmos/maneras de calcular que implementé para todos los formatos, pero igual se ha de acordar que después desarrollé en C++ solamente los formatos CSR y ELL. Entonces, sus maneras de calcular prevaldrán en ese documento y deben ser consideradas por el lector como lo más importante de esa parte.

1) Matriz por escalar

Esa operación consiste en multiplicar una matriz por un entero o un real ; esa vez si que esa operación de multiplicación está conmutativa (aunque una búsqueda profunda en temas de matemática muestre que para números que pertenecen al anillo de los cuaterniones, no está conmutativa).

De manera concreta lo que se hace durante esa operación es multiplicar cada valor de la matriz por ese escalar. Entonces, sabiendo que una matriz almacenada, con cualquier tipo de formato, tiene siempre sus valores no nulos en una tabla del formato, que sea de dimensión 1 o 2, utilizando la simetría o no, solamente se ha de multiplicar todos los valores con el escalar y el cálculo ya está hecho.

2) Lectura de valor / Escritura (modificación) de valor

Esas dos operaciones consisten en buscar una coordenada en una matriz y leer o modificar el valor almacenado. Utilizando el formato matemático DNS, el acceso al valor es muy simple, y también para el formato COO, pero para formatos especiales, no es tan sencillo, y se ha de calcular (en vez de solamente acceder directamente) donde está almacenado el valor de la coordenada querida. A continuación explicaré los diferentes tipos de acceso/búsqueda para lograr leer o cambiar una coordenada de una matriz con los formatos que expliqué en la primera parte de ese documento.

<u>a) Accesos directos o casi-directos</u>

Formato : DNS - Acceso directo

Con ese formato, ya que se tiene una tabla de dimensión 2 con todos los valores, ubicadas en la tabla tal cual como en la versión matemática, el acceso al valor es directo :

Lectura :

```
return this.data[l][c];
```

Escritura :

```
this.data[l][c] = v ;
```

Formato : COO - Acceso por busqueda de la coordenada, fila y columna directas

Con ese formato, ya que se tiene una tabla con valores y dos tablas (de dimensión 1 todas), solamente se ha de verificar si la coordenada tiene un valor. Después, se tiene que acceder a ese valor y, por fin, leerlo/modificarlo.

b) Búsqueda por fila/columna directa y indirecta por la tabla de indices (columna/fila)

*Formatos : CSC, **CSR**, **ELL**, MSR, Poly, PolySym*

En los formatos CSR, CSC (y en el ELL, un poco distinto, pero dará lo mismo), se almacena dos tablas, además de los valores: una que contiene los indices de fila o de columna para cada valor (por ejemplo, *indices*), mientras una otra que contiene “indices de delimitación de fila o columna (el opuesto a la opción escogida antes)” (por ejemplo, *ptr* (para punteros)) sobre la tabla de valores (por ejemplo, *data*).

En otros términos, primero se debe buscar en la última tabla donde se ubican los valores que pertenecen a la fila o columna de la coordenada a la cual queremos acceder. Y después se ha de recorrer la tabla de indices de fila/columna para saber si existe bien un valor no nulo para la coordenada que queremos. Si existe bien, se puede leer/modificar el valor en la tabla de valores, al rango determinado por las búsquedas que acabamos de hacer.

Lectura de la coordenada (f,c) (fila, columna), formato CSR

```
int i = ptr[l];
boolean found = false;
while (!found && i<ptr[l+1]){
      found = (indRow[i] == c);
      i++;
}

if (!found) throw new noExistenceException();
return data[i-1];
}
```

En el formato MSR, se almacena dos tablas de dimensión 1 :

- *data* con los valores de la diagonal y después las demás
- *ptr* con primero los indices (punteros) de fila y, en seguida, los indices de columna para cada valor que no esté en la diagonal.

Entonces, cuando la coordenada sea una de la diagonal, el acceso se hace directamente en la tabla de data y sino de ha de utilizar el mismo algoritmo que acabamos de explicar más arriba con las filas y columnas, solamente es que la información esa vez esta en una misma tabla con los dos tipos de informaciones y no dos tablas.

Lectura de la coordenada (f,c) (fila, columna), formato CSR

```
int rep =0;

//diagonal value
if(l==c){
      rep = data[l-1];
}

//other cases
else{

      //inf
      if (l>c) throw new noExistenceException();
      //supp
      else{
            int k1= ptr[l]-1;
            int k2= ptr[l+1]-1;
            boolean sortie= false;
            for (int j = k1; j< k2 && !sortie ;j++){
                  if (ptr[j]-1 == c){
                        rep = data[j];
                        sortie = true;
                  }
            }
            if (!sortie) throw new noExistenceException();
      }
}
return rep;
```

En los formatos Poly y PolySym, seguimos la misma idea para acceder a la coordenada querida, solamente que no se utiliza tablas exactamente iguales a les que describí más arriba.

Lectura de la coordenada (f,c) (fila, columna), formato Poly

```
int debut = ind[l];
int larg = ind[l+1] - debut;
int iniCo = iniRow[l];
int diff = c - iniCo;
if (c<iniCo || c>= iniCo+larg) throw new noExistenceException();
return data[debut+diff];
```

Para el formato PolySym, por supuesto, se ha de pensar que también existen los valores para los coordenadas simétricas, entonces a la hora de acceder a la matriz y buscar una coordenada, se ha de no olvidar de añadir una parte al algoritmo de lectura/escritura.

Búsquedas utilizando una tabla de “punteros” = “indices de delimitación de fila o columna”

Además, para algunos formatos (*DiaSym, BSR*) los algoritmos siguen utilizando esas tablas de “indices de delimitación de fila o columna” refiriéndose a la otra tabla que es de los valores, pero lo que sigue en el algoritmo no cabe en lo expliqué más arriba, porque es más especifico del formato de almacenamiento utilizado.

Búsquedas muy especificas al formato :

Por fin, para algunos formatos (*PolySymFix, BDN, DIA*) los algoritmos para encontrar el valor de una coordenada en la matriz almacenada son muy específicos al formato y no siguen unas reglas o unas maneras de buscar la coordenada que valgan la pena explicar porque no son para nada genéricas.

3) Matriz por vector

Esa operación consiste en multiplicar una matriz por un vector de enteros o de reales ; cuando multiplicamos una matriz por un vector, es necesario que el número de filas del vector coincida con el número de columnas de la matriz.

Formato CSR

```
void vectMult (vector<double> & vec, CSR & a, vector<double> & res){
        for (int i = 0; i< a.ptr.size() -1 ;i++){
                int k1 = a.ptr[i];
                int k2 = a.ptr[i+1];
                double sum =0;
                for (int j = k1; j < k2; j++){
                        sum = sum + a.data[j]* vec[a.indRow[j]];
                }
                res[i] = sum;
        }
}
```

Nota : El producto se hace accediendo directamente a los datos, usando la tabla de punteros de filas.

Formato ELL

```
void vectMult (vector<double> & vec, ELL & a, vector<double> & res){
   int imax = a.data.size();
   int jmax = a.data[0].size();

        for (int i = 0; i< imax;i++){
                double sum =0;
                for (int j = 0; j < jmax; j++){
                        double val = a.data[i][j];
                        if  (val != 0){
                                sum = sum + val * vec[a.indices[i][j]];
                        }
           }
                res[i] = sum;
        }
}
```

Nota : El producto se hace accediendo directamente a los datos. Cuidamos no hacer producto inútiles con los algunos ceros que son almacenados con el formato ELL.

III – Eficiencia de formato

0) Introducción a los diferentes criterios de eficiencia

En informática uno siempre debe explicar a qué se refiere cuando habla de eficiencia de un algoritmo. Es por ejemplo totalmente distinto, en teoría de la computación, de hablar del carácter “costoso o no costoso” de una solución a un problema refiriéndose al coste al nivel de la memoria que usa la solución implementada o bien al coste al nivel del tiempo necesitado para acabar de calcular la respuesta querida.

De hecho, en esa parte, trataré tres aspectos a propósito de los formatos explicados en la primera parte del documento. Primero explicaré las diferencias de eficiencia en términos de memoria, i.e el espacio ocupado en memoria cuando se almacena una matriz con un formato preciso, luego en términos de cálculos, i.e el número de operaciones básicas que se hace a la hora de hacer un cálculo con una matriz almacenada en ese formato.

1) En términos de memoria (espacio ocupado)

Aquí, me voy a referir, como dicho en la introducción, al espacio que utiliza en memoria los formatos para almacenar una matriz.

Datos sobre las matrices que se van a utilizar para las comparaciones

Se hablará a continuación de :

- Matriz de tamaño m * n (m filas y n columnas)
- Número de valores no nulas : k

Trataremos el caso de matrices de enteros ya que eso permite unir el uso de memoria necesitado para almacenar un valor o un número de fila/columna.

Para algunos casos específicos :

- y = algunos ceros
- z = número de diagonales
- nbb = número de bloques

Cálculos

- Con el estándar (DNS), se almacenan m * n int.
- Con el por coordenadas (COO), se almacenan 3 * k int.

A continuación, como se usará también otras variables que m, n y k, por ejemplo, y o z, como lo expliqué más arriba, se hará simplificaciones. Se entiende por

"simplificaciones", que esas otras posibles constantes se considerarán como nulas porque son potencialmente mucho mas pequeñas que las otras variables, y entonces no relevantes para el cálculo.

Estos porcentajes de ahorro anunciados son "sencillos", en el sentido que :

1) se han hecho simplificaciones
2) se aplican cada uno en un caso preciso :
    1. DNS, m,n >> k → matriz muy grande, con muy pocos valores (matriz muy sparse)
    2. COO, n, m << k → matriz no tan grande, con bastante valores (matriz no tan sparse)

	Número de int en memoria	Número simplificado	Ahorro vs DNS (con m,n >> k) en %	Ahorro vs COO (con n, m << k) en %
CSC	k+ k +n	2k+n	1-(1/m)	0,33
CSR	k+ k+m	2k+m	1-(1/n)	0,33
Dia	k+y + (m-z)	k+m	1-(1/n)	0,66
DiaSym	k+y +m	k+m	1-(1/n)	0,66
ELL	2 * (k+y)	2k	~ 1	0,33
MSR	2 * (k+1)	2k	~ 1	0,33
BDN	k+y	k	~ 1	0,66
Poly	m+m+1+k+y	k+2m	1 – (2/n)	0,66
PolySym	m+k+y	k+m	1-(1/n)	0,66
PolySymFix	k+y+1	k	~ 1	0,66
BSR	k+ nbb*2	k	~ 1	0,66

Tabla 1 : Ahorro de espacio

2) En términos de cálculos (operaciones teóricas)

Se puede también evaluar directamente en los algoritmos usados el número de cálculos de la operación más importante del cálculo (la que se repite más y que cuesta más).

Las matrices son de tamaño n*n.

Formato	Número de reales	Número de enteros	Número de accesos indirectos para matriz*vector (simple indirección)	Número de accesos indirectos para matriz*vector (doble indirección)
CSC	k	k + n	3k	k

CSR	k	k + n	2k	2k
Dia	z*n	z*n	3k	k
ELL	z*n	z*n		
MSR	k + 1	k + 1	3k	k
BDN	n*(z+1)			
BSR	k	2k	3k	k

Tabla 2 : Producto matriz * vector y indirecciones

3) En términos de tiempo de cálculo (pruebas)

En Java

Classe : System
Method : `nanoTime()`

Se puede usar poniéndolo justo antes de la ejecución del cálculo matriz por vector y justo después, y al final calcular la diferencia entre los dos. Así se sabe cuanto tiempo ha sido necesario para el cálculo.

Para tener datos fiables, se puede hacer esos cálculos en una bucle que hace el mismo cálculo muchas veces, almacenar los resultados en un ArrayList/tabla y calcular el valor medio.

En C/C++

De manera similar, se puede utilizar métodos de la *STL (standard template library) : QueryPerformanceCounter*...

También se puede usar otras métodos, del OpenMP :omp_get_wtime( ), por ejemplo.

Comanda linux

La comanda */usr/bin/time* permite medir el tiempo total de ejecución de un programa. Lo interesante de esa ultima manera es que también se recibe una información sobre el uso del CPU : así se sabe si realmente varios threads se utilizaron en la ejecución del código o no.

IV – Matrices de ejemplo, paralelismo y pruebas

0) Introducción al paralelismo

Los ordenadores paralelos son maquinas que tienen una arquitectura constituida de varios procesadores, los cuales permiten el tratamiento de una misma tarea de manera concurrente. Existen varios tipos de ordenadores o procesadores paralelos, con distintos tipos de inter-conexiones entre los procesadores entre ellos y entre los procesadores y la memoria.

Existe una clasificación bastante famosa, la de Flynn, que clasifica los ordenadores paralelos (y secuenciales) según sus tipos de organización del flujo de datos y del flujos de instrucciones. Los tipos más clásicos son las maquinas :

- SIMD (*Single Instruction stream-Multiple Data stream*), en las cuales hay un único flujo de instrucciones que trata varios datos
- MIMD (*Multiple Instruction stream-Multiple Data stream*), en las cuales hay varios flujos de instrucciones y varios de datos

La meta del paralelismo es acabar cálculos en tiempo menores que si fuese con un sistema serial.

1) Presentación de los bancos de matrices sparse reales

- http://math.nist.gov/MatrixMarket/

Matrix Market es un servicio proporcionado por la “Mathematical and Computational Sciences Division” del “Information Technology Laboratory of the National Institute of Standards and Technology”. El NIST es una agencia del “U.S. Department of Commerce”.

- http://www.cise.ufl.edu/research/sparse/matrices/

Es la colección de matrices sparse de la universidad de Florida. Esas matrices también vienen de aplicaciones reales. La colección es muy usada por la comunidad de álgebra lineal numérica para el desarrollo y la evaluación de resultado de los algoritmos para matrices sparse. También se utiliza para experiencias de pruebas de robustez y de repeticiones de acciones, por su carácter real (no son matrices sparse generadas de maneras artificiales).

2) Metodología seguida

Primero, tuve que escribir el código para los formatos CSR y ELL en C++. Probé que los formatos estaban bien implementados con unos ejemplos muy sencillos (de tamaño pequeño, matrices 5*5).

Después implementé funciones que permiten creer matrices de manera artificiales, y permitiendo al usuario elegir el tamaño de la matriz : con eso pude hacer pruebas con matrices muy grandes, tipo 300 000 * 300 000 y con 3 000 000 de valores.

Luego, descargué ejemplos reales en los dos sitios web, en formato .txt, con los valores de la matriz por coordenadas. Entonces, implementé, en Java para empezar, funciones para leer esos ficheros .txt y crear otros ficheros .txt con las informaciones en formato CSR y ELL. Pero me enfrenté a un problema de escalabilidad. En efecto, Java utilizar el Java heap para su gestión de memoria, y al creer los ficheros .txt para matrices de más de 10 000 * 10 000 aproximadamente, el *Java heap* colapsa y no permite la creación de los ficheros.

A continuación, para intentar resolver ese problema de escalabilidad, implementé en C++ las funciones de lectura y creación de ficheros de matrices reales en formato CSR y ELL. Eso me permite probar matrices un poco más grande, tipo 20 000 * 20 000, pero esa vez, sin la "protección" del Java heap, es la maquina (ordenador) que colapsa totalmente.

La implementación del paralelismo se hago copiado el código en C++ del fichero serial y añadiendo los pragma de OpenMP.

3) Resultados de cálculos matriz por vector

Con las matrices generadas de manera artificiales, el uso del paralelismo permite nota directamente una eficiencia mayor. A continuación podremos ver dos ejemplos, uno con una maquina con dos cores y uno con una maquina con cuatro cores.

(ver figura 18)

Con las matrices reales, no logré que el tiempo ganado usando el paralelismo sea relevante. A pesar de los esfuerzos para obtener algo con el código, ese trabajo queda para el futuro de este proyecto.

(ver figura 19)

```
pierre@pierre-ubuntu-HP-Pavilion-dv7-Notebook-PC:~/Bureau$ /usr/bin/time sh -c 'echo "30000 300000" | ./CSR.o'
1       3       2       4
1       7       2       8       5       3       9       6       4

0       1       1       2       0       2       3       1       3

0       2       4       7       9
22      22      47      34
Enter a number (data)
Enter a number (size)
vector lleno
etapa 1
etapa 2
etapa 3
matriz llena
holastart 169.424
holaend 173.653
Temps 4.22901 sec . time
4.22user 0.00system 0:04.24elapsed 99%CPU (0avgtext+0avgdata 35232maxresident)k
0inputs+0outputs (0major+2644minor)pagefaults 0swaps
pierre@pierre-ubuntu-HP-Pavilion-dv7-Notebook-PC:~/Bureau$ /usr/bin/time sh -c 'echo "30000 300000" | ./CSRpar.o'
1       3       2       4
1       7       2       8       5       3       9       6       4

0       1       1       2       0       2       3       1       3

0       2       4       7       9
Enter a number (data)
Enter a number (size)
vector lleno
etapa 1
etapa 2
etapa 3
matriz llena
start 177.819
end 180.43
Temps 2.61166 sec . time
5.21user 0.00system 0:02.63elapsed 198%CPU (0avgtext+0avgdata 35232maxresident)k
0inputs+0outputs (0major+2652minor)pagefaults 0swaps
```

Figura 18 : Tiempo & uso de CPU serial y con paralelismo (2cores)

```
1181315/1181315> /usr/bin/time ./CSR.o
vector lleno
etapa 1
etapa 2
etapa 3
matriz llena
holastart 14150.7
^[[Aholaend 14152
Temps 1.30263 sec . time
4.84user 0.00system 0:01.31elapsed 368%CPU (0avgtext+0avgdata 39760maxresident)k
0inputs+0outputs (0major+3731minor)pagefaults 0swaps
```

Figura 19 : Tiempo & uso de CPU con paralelismo (4cores) – Maquina de laboratorio

4) Conclusiones

C++ permite manejar matrices más grande que Java, por culpa de su Java heap, pero ese permite preservar el ordenador contra el colapso dado a demasiado memoria requerida por la creación de ficheros.

Vemos que ese problema de escalabilidad aparece no durante los cálculos sino durante la recuperación de datos reales para hacer las pruebas : se ha hecho pruebas con matrices generadas de manera artificial mucho más grande (100^2 veces más grandes).

El paralelismo implementado funciona para matrices artificiales pero queda para el futuro lograr que funcione para matrices reales.

V - Bibliografía completa

1) Matrices dispersas

Los conocimientos sobre las matrices dispersas son :

1) Los que vienen de los estudios matemáticos. Se supone que esos no van cambiar mucho hoy en día.
2) Los que vienen de los estudios sobre el almacenamiento. Se supone que siempre se puede inventar nuevas cosas o mejorar lo que ya existe.

http://fr.wikipedia.org/wiki/Matrice_creuse, Wikipedia, febr. 2014

http://gilles.dubois10.free.fr/algebre_lineaire/creuses.html, Gilles Dubois, sept. 2011

deptmedia.cnam.fr/new/spip.php?pdoc5318 (pdf), CNAM (clase a distancia)

2) Operaciones sobre las matrices dispersas

Aquí estamos en el mismo caso que en la ultima parte.

https://www.hds.utc.fr/~boufflet/PAGES-NF16/tp4051101.html, M.Boufflet, nov. 2001

3) Lenguajes de programación y las librerías

En la red se encuentra muchas informaciones sobre distintos lenguajes. Se tiene que tener en cuenta que a veces son clases, a veces, conocimientos de una persona que lo comparte o también “solamente” (en el sentido que puede ser falso, o enfocado en algo demasiado preciso) un foro.

C

http://pastix.gforge.inria.fr/files/README-txt.html, INRIA, marzo 2012

C++

http://forum.hardware.fr/hfr/Programmation/C-2/affection-conception-reduite-sujet_127387_1.htm, forum Hardware, enero 2010

http://eigen.tuxfamily.org/index.php?title=Main_Page, Wiki, feb 2014

http://sourceforge.net/projects/hasem/, Sourceforge, marzo 2014

http://www.zebulon.fr/questions-reponses/determinant-d-une-enorme-matrice-creuse-249.html, forum Zebulon, 2005

Java

http://www.sanfoundry.com/java-program-implement-sparse-matrix/, Manish Bhojasia, marzo 2014

http://answers.yahoo.com/question/index?qid=20100221123816AAUOX3k, forum Yahoo, 2010

http://www.java-forums.org/new-java/36213-sparse-matrix-java.html, Java forum, 2010

http://introcs.cs.princeton.edu/java/44st/SparseMatrix.java.html, Robert Sedgewick and Kevin Wayne, feb 2011

Fortran

http://www.netlib.org/blas/, National Science Foundatio, junio 2011

Python

https://karczmarczuk.users.greyc.fr/TEACH/Paral/HPfor.html, “karczmarczuk”, nov 2012

Scilab

http://fr.wikibooks.org/wiki/D%C3%A9couvrir_Scilab/Matrices_creuses, wikipedia, marzo 2014

4) Estructuras de datos y algoritmos

Se encuentra muchas informaciones sobre esta tema, ya que es mucho más amplio que solamente el tema del proyecto. Entonces, en esta parte, se ha de vigilar que las informaciones recogidas son bien sobre el tema querido.

http://forge.scilab.org/index.php/p/docscisparse/downloads/337/, Michael Baudin, nov. 2011

Algorithmes et structures de données avec Ada, C++ et Java, Abdelali Guerid, Pierre Breguet, Henri Röthlisberger, 2002

Analyse numérique et optimisation, Grégoire Allaire, 2005

MATLAB R2009, SIMULINK Et STATEFLOW Pour Ingenieurs, Chercheurs Et Etudiants, Nadia Martaj y Mohand Mokhtari, 2011

https://www.irisa.fr/sage/jocelyne/cours/INSA/chapcreux2014.pdf (pdf) J. Erhel, enero 2014

Algorithmes matriciels pour les graphes , Guesmi Hela, Hasni Hamadi & Mahjoub Zaher (pdf), 2012

5) Cálculos con paralelismo

Calcul de structures et parallélisme : un bilan et quelques développements récents, David Dureisseix y Laurent Champaney, enero 2000

Calcul scientifique parallèle - Cours, exemples avec openMP et MPI , exercices corrigés
Frédéric Magoulès y François-Xavier Roux, 2013

http://mumps.enseeiht.fr/*, project PARASOL, 2014*

6) Tesis

Algorithmes parallèles efficaces pour le calcul formel : algèbre linéaire creuse et extensions algébriques, Jean-Guillaume Dumas, diciembre 2000 (pdf)

Étude de la distribution, sur système à grande échelle, de calcul numérique traitant des matrices creuses compressées , Olfa Hamdi-Larbi , marzo 2010 (pdf)