IMPLEMENTACIÓN DE UNA LIBRERÍA ORIENTADA A OBJETO PARA MATRICES ESPARCIDAS EN PARALELO

Por

Catalina María Rúa Alvarez

Tesis sometida en cumplimiento parcial de los requerimientos para el grado de

MAESTRÍA EN CIENCIAS

en

COMPUTACIÓN CIENTÍFICA

UNIVERSIDAD DE PUERTO RICO
RECINTO UNIVERSITARIO DE MAYAGÜEZ

Octubre, 2007

Aprobada por:

Dorothy Bollman, Ph.D Miembro, Comite Graduado	Fecha
Paul Castillo, Ph.D Miembro, Comite Graduado	Fecha
Esov Velázquez, Ph.D Presidente, Comite Graduado	Fecha
Dorial Castellanos, Ph.D. Representante de Estudios Graduados	Fecha
Julio Quintana, Ph.D Director del Departmento	Fecha

Resumen de Disertación Presentado a Escuela Graduada
de la Universidad de Puerto Rico como requisito parcial de los
Requerimientos para el grado de Maestría en Ciencias

IMPLEMENTACIÓN DE UNA LIBRERÍA ORIENTADA A OBJETO PARA MATRICES ESPARCIDAS EN PARALELO

Por

Catalina María Rúa Alvarez

Octubre 2007

Consejero: Ph.D Esov Velázquez
Departamento: Departamento de Ciencias Matemáticas

Al buscar la solución matemática en muchos de los problemas científicos, se ven involucradas matrices de dimensiones altas y cuyos elementos distintos de cero son pocos comparados con el orden de la matriz, estas matrices se conocen como *matrices esparcidas* y existen diferentes formatos para almacenarlas, entre ellos el Compressed Sparse Rows (CSR).

Para lograr la solución de algunos de los problemas que tienen matrices, se necesitarán solucionar sistemas lineales y a su vez se efectuarán un gran número de veces el producto de matrices con vectores. Esta cantidad de operaciones y las dimensiones de las matrices, hacen que se requiera del uso de implementaciones para computadoras en paralelo.

Al efectuar el producto de una matriz esparcida con un vector distribuidos en paralelo, debe haber comunicación entre procesadores del vector o parte del vector que cada uno de ellos almacenan y esta comunicación podría hacer que el tiempo de cómputo sea alto.

En esta tesis se desarrolló una implementación para matrices esparcidas en paralelo con diferentes estrategias para atacar el problema de comunicación en el producto de una matriz esparcida con un vector. La implementación se realizó con el paradigma orientado a objetos en C++ y para la programación en paralelo con las librerías de openMPI.

Abstract of Disertación Presented to the Graduate School
of the University of Puerto Rico in Partial Fulfillment of the
Requirements for the Degree of Master of Science

IMPLEMENTATION OF A PARALLEL OBJECT ORIENTED LIBRARY FOR SPARCE MATRICES

By

Catalina María Rúa Alvarez

October 2007

Chair: Ph.D Esov Velázquez
Major Department: Mathematical Sciences

The mathematical solution of many scientific problems involves matrices of high dimensions and whose nonzero elements are few compared to the order of the matrix. These matrices are known as sparse matrices and there exist several storage formats, among them the Compressed Sparse Rows (CSR) format.

In order to obtain tbe solution of some of the problems involving matrices, it is necessary to solve linear systems requiring the product of a matrix by a vector to be performed a large number of times. The number of operations and the dimensions of the matrices make necessary the use implementations for parallel computers.

When performing the product of a sparse matrix by a vector distributed in parallel there must be communication between the processes keeping each part of the vector, and this communication may cause the overall computing time to be high.

In this thesis we developed an implementation for parallel sparse matrices with several strategies for solving the communication problem for performing the product of a sparse matrix by a vector. The implementation uses the object-oriented paradigm in C++ and the Open MPI library.

Copyright © 2007

por

Catalina María Rúa Alvarez

A un gran amigo que desde niña me dio todo su apoyo y comprensión:

El gordo, Jairo.

AGRADECIMIENTOS

Gracias, para todos mis amigos y familia que me ayudaron a lograr este sueño en compañía de mi hijo, en especial a John y a mi madre que nunca dejaron de confiar en mi.

El gusto que dejo por mi carrera y las ganas de continuar investigando, se los debo al profesor Paul Castillo que durante mis dos años de estudio compartió conmigo gran parte de su conocimiento en métodos numéricos y matemáticas aplicadas.

Agradezco a mi asesor, quien me recordó que no hay cosas imposibles y me enseñó lo que hoy se de programación paralela.

Índice general

página

LISTA DE TABLAS

Tabla página

LISTA DE FIGURAS

Figura página

LISTA DE ABREVIATURAS

CSC	Compressed Sparse Columns.
COO	Coordinate Format.
CSR	Compressed Sparse Rows.
DIAG	Diagonal Format.
EDP	Ecuación Diferencial Parcial.
MPI	Message Passing Interface.
NNZ	No Number Zero.
PCSR	Parallel Compressed Sparse Rows.
PETSc	Portable Extensible Toolkit for Scientific Computation.
PVECTOR	Parallel Vector.
SPMD	Single Program Multiple Data Model.

LISTA DE SIMBOLOS

p	Número de procesos.
p_i	Proceso i.
rank	Rango de un proceso.
$\|\|\|_2$	Norma L2.

Capítulo 1
INTRODUCCIÓN

En la búsqueda de las soluciones a diversas aplicaciones de tipo científico, encontramos la necesidad de plantear modelos matemáticos que requieren ecuaciones diferenciales parciales (EDP). Al resolver las EDP, tratando que la solución que se obtenga con un modelo sea bien cercana a la exacta, es necesario el uso de sistemas de gran escala y métodos de alto orden.

Diferencias finitas y elementos finitos son algunos de los métodos numéricos que se usan para resolver ecuaciones diferenciales que generalmente resultan de problemas físicos. Cuando estos métodos son usados usualmente se generan matrices de dimensiones enormes, cuyos elementos distintos de cero son pocos comparados con el orden de la matriz. Estas matrices se llaman esparcidas.

Para aprovechar la estructura de estas matrices y almacenar únicamente los elementos diferentes de cero, se han creado diferentes formatos de almacenamiento de datos entre ellos el "Compressed Sparse Rows" (CSR).

El CSR es una técnica para manipular matrices esparcidas y consiste en almacenar por fila el número de entradas diferentes de cero, cada elemento no nulo y su correspondiente columna. Este formato reduce el número de operaciones y el uso de memoria.

Por las dimensiones de estas matrices, un computador personal puede tomar mucho tiempo para los cómputos por lo que se necesita distribuir la matriz y las tareas a varios procesadores para reducir el tiempo de espera de resultados.

Entre las operaciones que se utilizan en estos procesos se destaca, el producto de una matriz esparcida por un vector; ya que para que esta operación se complete, cada proceso necesitará información de la porción de vector que los otros procesos tienen almacenado y según la estrategia de comunicación utilizada entre los procesos, el tiempo de computo puede disminuir o aumentar.[**?**]

En este proyecto se realizó una implementación para matrices esparcidas distribuidas en procesos con bloques CSR, en la cual se crearon diferentes métodos para realizar la comunicación entre los procesos y lograr el producto de matriz esparcida por vector. Esto se realizó porque al querer encontrar las soluciones de las EDP, se necesita realizar muchos productos. La implementación fue realizada utilizando MPI-2.

Para verificar si los resultados obtenidos con la implementación son correctos y ver el comportamiento del programa, se realizaron pruebas con diferentes matrices esparcidas. Algunas de estas matrices, provenientes de problemas en temas de estadística, aeronáutica', teoría de grafos y soluciones por medio de elementos finitos. Se tomaron de la página "Matrix Market" (http://math.nist.gov/MatrixMarket/data), convirtiéndolas del formato de origen al compatible con el formato de la implementación realizada en la tesis. Otras pruebas se realizaron comparando la solución de la ecuación de transporte en 2D y la ecuación de calor en 2D, con la aproximación obtenida usando la forma matricial de diferencias finitas[**?**].

Para efectos de comparación se hicieron pruebas del producto de matrix por vector en paralelo con el programa PETSc "Portable Extensible Toolkit for Scientific Computation"[**?**]. PETSc es un conjunto de estructuras de datos y rutinas paralelas para la solución de modelos científicos a gran escala, en especial para solucionar EDP, que usa para la comunicación entre procesos el “estándar MPI".

Las pruebas se llevaron a cabo en el cluster del Departamento de Ciencias Matemáticas que cuenta con 8 nodos, cada nodo tiene un procesador intel Xeon 64 bits (4 procesos con 3.0Ghz y 4 procesos con 3.2Ghz) y 4Gb de memoria, y en el cluster del Departamento de Ingeniería de 65 nodos, cada nodo tiene dos procesadores Intel 386 de 120 MHz y 1Gb de memoria.

En el segundo capítulo se encuentra una descripción de matrices esparcidas y su implementación en serie y en paralelo. El tercer capítulo es una descripción de la implementación de una librería para vectores, que incluye funciones en paralelo, la cuál se hizo con el fin de usarse en las pruebas del producto de matriz esparcida con vectores desarrolladas en la tesis. En el cuarto capítulo se presentan las estrategias de comunicación entre procesos que se usaron en la implementación. En el quinto capítulo, se tiene una descripción de los diferentes productos implementados en paralelo con el fin de lograr los objetivos de la tesis. El sexto capítulo, contiene pruebas numéricas con el producto de ejemplares aleatorios de matrices esparcidas de Matrix Market con vectores y pruebas numéricas con la solución de la ecuación de transporte y de onda en 2D usando diferencias finitas; además de productos de matrices esparcidas y vectores en paralelo usando el programa PETSc. En el septimo capítulo observamos trabajos futuros y observaciones para desarrollar otras implementaciones sobre el tema de la tesis.

Capítulo 2
MATRICES ESPARCIDAS

En este capítulo, encontramos una descripción sobre matrices esparcidas y las clases serial y paralela implementadas para almacenar y operar con estas matrices.

Como ya se mencionó, *una matriz es esparcida* cuando las entradas no nulas son muy pocas comparadas con el orden de la matriz. Para tomar ventaja de esta propiedad y almacenar únicamente los elementos diferentes de cero de la matriz y sus respectivas posiciones, se han creado diferentes técnicas.

El objetivo principal de cada estructura de datos que se genera en cada técnica, es lograr que las matrices esparcidas puedan ser empleadas en forma eficiente para la solución de sistemas lineales almacenando únicamente los valores de las componentes no nulas de la matriz.

La idea de que los elementos nulos de la matriz no necesitan ser guardados, fué dada por ingenieros de varias disciplinas. Inicialmente, se desarrollaron solamente técnicas especiales para trabajar con matrices en bandas; es decir, matrices que tienen elementos diferentes de cero sólo en algunas de sus diagonales. En los 60's, ingenieros eléctricos lideres en redes eléctricas, fueron los primeros en explotar la esparcidad de las matrices para solucionar sistemas lineales de matrices esparcidas con estructura irregular.[**?**]

Muchas de estas técnicas han logrado ser económicas, en el sentido de almacenamiento y eficiencia computacional. Además, al usarse estas técnicas para matrices esparcidas en la solución de sistemas lineales, se ha logrado obtener soluciones que no se podían calcular al usar una estructura de datos para matrices donde la mayor parte de sus componentes son diferentes de cero, es decir matrices densas .

Otra ventaja de usar estructuras esparcidas es la disminución del orden computacional en algunas operaciones de matrices de $O(n \times m)$ a $O(nnz)$, donde n es el número de filas y m el número de columnas de la matriz y nnz (no number zero) es la cantidad de entradas distintas de cero de la matriz.

Las matrices esparcidas pueden ser estructuradas o no estructuradas:

1. *Una matriz esparcida es estructurada* cuando las posiciones de los elementos diferentes de cero tienen una forma o patrón regular, como por ejemplo las matrices por bandas. También pueden presentarse matrices estructuradas por bloques de tamaños iguales con una forma o patrón regular.
2. Una matriz esparcida para la cual las posiciones de los elementos son irregulares y no se pueden predecir las posiciones de los elementos no nulos de la matriz *es llamada irregularmente estructurada o no estructurada.*

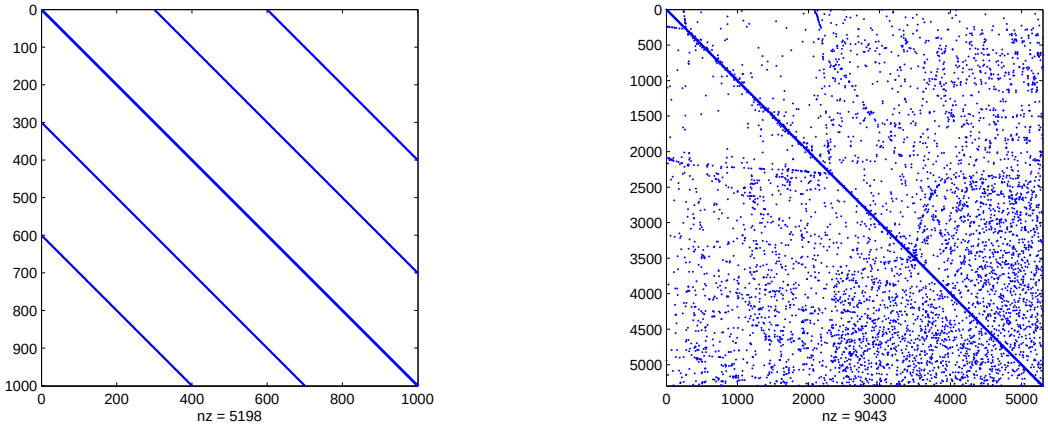

Figura 2–1: Matrices esparcidas, Estructurada y No Estructurada

En las gráficas del patrón de esparcidad de una matriz esparcida, únicamente se observan los elementos distintos de cero de la matriz sin diferenciar de forma alguna su valor numérico. En la figura 2–1 observamos la gráfica del patrón de esparcidad para una matriz esparcida estructurada y una matriz esparcida no estructurada.

Una de las operaciones esenciales de matrices esparcidas al resolver sistemas lineales o esquemas numéricos para solucionar ecuaciones diferenciales, es el producto de matriz por vector. Al realizar esta operación en computadores de alto rendimiento, la eficiencia y el rendimiento computacional se ven afectados por la estructura de la matriz; dependiendo de si la matriz es regularmente estructurada o no, por la técnica de almacenamiento que se use, entre otras posibles causas.

2.1. Técnicas para almacenamiento de matrices esparcidas

En esta sección se presentan algunas de las técnicas para almacenar matrices esparcidas teniendo en cuenta únicamente los valores de las componentes no nulas de la matriz son:

2.1.1. Formato de Coordenadas (COO - Coordinate format)

Es el esquema más sencillo para almacenamiento de datos en matrices esparcidas, porque almacena cada elemento diferente de cero y los indices de su correspondiente fila y columna. La estructura de datos está dada por tres arreglos:

1. Un arreglo de reales **AA**, que contiene todos los valores diferentes de cero de la matriz.
2. Un arreglo de enteros **IA**, que contiene respectivamente el valor del índice de la fila de cada elemento diferente de cero.
3. Un arreglo de enteros **JA**, que contiene el valor del índice de la columna de cada elemento diferente de cero.

Los arreglos **IA** y **JA**, indican la posición en la matriz de cada elemento almacenado. Los tres arreglos son de tamaño nnz. En este formato los elementos son almacenados en cualquier orden.

La numeración de los indices de las matrices será desde cero, para cualquiera de los formatos de almacenamiento.

Ejemplo. *Veamos para la matriz esparcida A, la estructura de datos con los diferentes formatos de almacenamiento para matrices esparcidas.*
Con el formato COO, tenemos que la matriz A se almacena de la siguiente forma.

$$A = \begin{pmatrix} 2 & 4 & 0 & 0 & 0 \\ 0 & -1 & -5 & 0 & 0 \\ 0 & 0 & 7 & 1 & 0 \\ 11 & 0 & 0 & 1 & 2 \\ 0 & 12 & 0 & 0 & 1 \end{pmatrix} \tag{2.1}$$

AA	2	1	1	4	-5	-1	7	2	1	11	12
IA	0	3	2	0	1	1	2	3	4	3	4
JA	0	3	3	1	2	1	2	4	4	0	1

2.1.2. Formato por Filas Comprimidas (CSR - Compressed Sparse Rows)

Una modificación del esquema de coordenadas consiste en almacenar los elementos no en forma arbitraría sino manteniendo un orden; ya sea almacenando todos los datos en orden ascendente de filas o de columnas.

Con este cambio se puede notar que si se mantiene la estructura de datos de COO, se van a repetir algunos valores de las filas o las columnas según el caso en que se haya decidido ordenar la matriz; para evitar esto se hace un cambio a esta estructura de datos.

En el caso en que se almacenan los elementos de la matriz manteniendo un orden por filas, reemplazamos el segundo arreglo **IA** por un nuevo arreglo con la posición o apuntadores correspondientes cuando haya un cambio de fila.

Este nuevo formato es conocido como **CSR** y es probablemente uno de los más populares y usados para almacenamiento de matrices esparcidas. La nueva estructura de datos es:

1. Un arreglo de reales **AA**, que contiene los valores reales a_{ij}, almacenados en orden por filas, desde la primera hasta la última fila n de la matriz. La longitud de este arreglo es nnz.
2. Un arreglo de enteros **JA**, que contiene los indices de columna de los elementos a_{ij} respectivamente de como fueron almacenados en el arreglo **AA**. La longitud de este arreglo es nnz.
3. Un arreglo de enteros **ptrIA**, que contiene las posiciones a las que debe recurrirse, para el comienzo de cada nueva fila de elementos en la matriz, para los arreglos **AA** y **JA**. La longitud de este arreglo es $n + 1$, recordando que n es el número de filas de la matriz.

Ejemplo. *Con el formato CSR, tenemos que la matriz A en 2.1, se almacena de la siguiente forma.*

AA	4	2	-1	-5	7	1	1	11	2	1	12
JA	1	0	1	2	2	3	3	0	4	4	1
ptrIA	0	2	4	6	9	11					

La técnica **CSR** es preferida a la de coordenadas, porque da más rendimiento computacional, aunque es menos flexible y es más compleja que la de coordenadas.

Podemos notar que al restar dos elementos consecutivos de mayor posición con el de la posición anterior en el arreglo **IA**, obtenemos la cantidad de elementos diferentes de cero para cada fila, y el último valor del arreglo **IA**, tiene la cantidad total de elementos diferentes de cero de la matriz. Por lo tanto, el arreglo **IA** nos da información sobre la posición donde inicia cada fila, la cantidad de elementos distintos de cero por fila y la cantidad de elementos diferentes de cero de la matriz.

Otro formato que resulta de la modificación del COO, es el Formato de Columnas Comprimidas (CSC - Compressed Sparse Column), el cual es similar al **CSR**, solo que ahora se almacenan los elementos por orden de columnas y se sigue en forma análoga al proceso descrito para el **CSR**.

En la implementación del **CSR** que se diseñó, se realizó un cambio a la estructura de datos que se describió. Se almacena ahora por cada fila, la cantidad de elementos diferentes de cero, los elementos y los indices de las columnas correspondientes. Este cambio, permite búsqueda y acceso rápido a los datos.

La estructura de datos está dada por:

1. Un arreglo **NNZ** de enteros, en el cuál cada posición almacena la cantidad de elementos diferentes de cero para cada fila.
2. Un arreglo de apuntadores **AA**, de longitud nnz, donde cada apuntador da la dirección a un arreglo de reales, que contiene los valores de los elementos diferentes de cero de la

correspondiente fila, este subarreglo tiene por longitud la cantidad de elementos no nulos de la fila correspondiente.

3. Un arreglo de apuntadores **JA**, de longitud *nnz*, donde cada apuntador da la dirección a un arreglo de enteros, que contiene los valores de los indices de columna por cada elemento diferente de cero de la fila correspondiente. La longitud del subarreglo, es la cantidad de elementos no nulos de la fila correspondiente.

Ejemplo. *Con el formato CSR modificado, tenemos que la matriz A en , se almacena de la siguiente forma.*

	NNZ		JA		AA
0 →	2	0 →	0 1	0 →	2 4
1 →	2	1 →	1 2	1 →	-1 -5
2 →	2	2 →	2 3	2 →	7 1
3 →	3	3 →	0 3 4	3 →	11 1 2
4 →	2	4 →	1 4	4 →	12 1

Notese que los elementos no necesariamente son almacenados por orden de las columnas, sino por orden de inserción de los datos. Para que la búsqueda de datos sea más eficiente, y por lo tanto las operaciones de matrices esparcidas, el programador debe almacenar los datos en orden de columnas por medio de la estrategia de ordenamiento que considere más conveniente.

En el desarrollo de la implementación para la tesis, se decidió realizar una étapa de ensamble de la matriz, en la cual los elementos son almacenados en apuntadores a mapas, pues la búsqueda en mapas es logarítmica. Esto dá la posibilidad que al finalizar la étapa de ensamble, se almacene en forma rápida y además en orden ascendente por columnas los elementos en la estructura de datos ya descrita.

2.1.3. Formato para diagonales (DIAG)

Este formato es usado principalmente para matrices con diagonal principal no nula y para las cuales los otros elementos diferentes de cero se encuentran únicamente en bandas superiores o inferiores (es decir en subdiagonales de la matriz). En este formato, se almacenan en un orden los elementos de cada diagonal.

La diagonal principal se enumera con 0 y a partir de ella se cuenta la distancia del resto de subdiagonales, enumerando hacia las subdiagonales a la derecha de la diagonal principal como positivas y las subdiagonales que se encuentran hacia abajo de la diagonal principal se enumeran con el signo negativo.

La estructura de datos está dada por:

1. Un arreglo de enteros **IOFF**, en el que se almacenan las posiciones de cada uno de las diagonales, iniciando con los indices negativos de las diagonales a la derecha de principal, luego el cero de la diagonal principal y por último los indices positivos de las subdiagonales hacia abajo de la principal.
2. Una submatriz **DIAG** (puede ser un arreglo de arreglos), que almacena los elementos de las subdiagonales. El tamaño de esta matriz es la cantidad de elementos de la diagonal principal y el número total de subdiagonales mas la diagonal principal.

Ejemplo. *Con el formato DIAG, tenemos que para la matriz A usada en los ejemplos anteriores, se almacena de la siguiente forma.*

Primero se tiene en cuenta la enumeración de la diagonal principal y las subdiagonales teniendo en cuenta sus respectivos signos.

$$A = \begin{array}{c} \begin{array}{cc} 0 & \Rightarrow + \end{array} \\ \Downarrow \\ - \end{array} \begin{pmatrix} \mathbf{2} & 4 & 0 & 0 & 0 \\ 0 & \mathbf{-1} & -5 & 0 & 0 \\ 0 & 0 & \mathbf{7} & 1 & 0 \\ 11 & 0 & 0 & \mathbf{1} & 2 \\ 0 & 12 & 0 & 0 & \mathbf{1} \end{pmatrix}$$

Luego colocamos los correspondientes valores en la estructura de datos.

IOFF	Diag		
	*	2	4
-3	*	-1	-5
0	*	7	1
1	11	1	2
	12	1	*

Este formato es muy usado para almacenar matrices por bandas. No es útil para almacenar matrices esparcidas no estructuradas. Un caso en el que una matriz esparcida quedaría almacenada como densa con el formato DIAG, es cuando se tiene una matriz con una diagonal en la cual el primer elemento por fila se encuentra en la última posición y así continua descendiendo hasta llegar al elemento de la fila final, que se encuentra en la primera posición.

2.2. Producto Matriz Esparcida en formato CSR por vector

Si expresamos una matriz esparcida como una matriz densa, el producto $y = Ax$ o $y = Ax+y$ entre una matriz densa $A \in \mathbb{R}^{n \times m}$ y un vector $x \in \mathbb{R}^m$, el orden del producto es $O(n \times m)$.

El algoritmo para calcular $y = Ax + y$ es

for $\mathbf{i} = \mathbf{0}$ to $n - 1$ do
  for $\mathbf{j} = \mathbf{0}$ to $m - 1$ do
    $y[i]$=$y[i] + AA[i][j]x[j]$ $\quad O(n \times m)$
  end
end

En cambio al efectuar el producto $y = Ax$ o $y = Ax+y$, entre una matriz esparcida $A \in \mathbb{R}^{n\times m}$ cuya cantidad de elementos diferentos de cero es nnz y un vector $x \in \mathbb{R}^m$, el orden del producto es $O(nnz)$. El algoritmo para calcular $y = Ax + y$ es

for $\mathbf{i} = \mathbf{0}$ to $n - 1$ do
  for $\mathbf{j} = \mathbf{0}$ to $NNZ[i] - 1$ do
    $y[i]$=$y[i] + AA[i][j]x[JA[i][j]]$ $\quad O(nnz)$
  end
end

Vemos que la cantidad de operaciones en el producto al expresar una matriz esparcida como densa, es mucho mayor que al usar alguna de las técnicas de almacenamiento, en este caso CSR.

En el producto con formato CSR, podemos notar que el ciclo "for" interno, depende de la cantidad de entradas diferentes de cero por fila. Por lo cual, si todos los elementos de una fila son cero, la cantidad de entradas nulas para esta fila es cero y no se ejecutará ninguna

iteración en este ciclo. Así que si de antemano conocemos cuales filas tienen entradas diferentes de cero, y almacenamos estas posiciones en un arreglo *nzrows*, de dimensión *dimnzr* que es la cantidad de filas que tienen por lo menos una entrada diferente de cero.

Se espera que al conocer las filas diferentes de cero, se reduzca el tiempo computacional. Este fué uno de los primeros cambios que se realizó en los productos, implementando un método *"get_ nonzero_ rows"*, que retorna el mapa de posiciones para las filas no nulas, un método para el producto de matriz por vector que tiene en cuenta estas filas, es *mat_ vec_ nonzero_ rows*. El algoritmo es:

for $\mathbf{i} = \mathbf{0}$ to $dimnzr - 1$ do
  for $\mathbf{j} = \mathbf{0}$ to $NNZ[nzrows[i]] - 1$ do
    $y[nzrows[i]]$=$y[nzrows[i]] + AA[nzrows[i][j]]x[JA[nzrows[i][j]]]$
  end
end

Algo más que podemos notar al trabajar con matrices esparcidas, es que también puede ocurrir que la matriz tenga varias columnas iguales a cero. Conocer cuales son estas columnas, no disminuye el número de cómputos, pero con esta información ya no se necesitaría el vector x completo para encontrar el resultado del producto con la matriz, pues los elementos del vector x que se encuentran en la misma posición de una columna igual a cero, no afectaran el resultado del producto de la matriz y el vector.

Ejemplo.

Para la matriz B y el vector x, se ilustra como realizar el producto Bx tiendo en cuenta el

mapa de columnas diferentes de cero de la matriz B.

$$B = \begin{pmatrix} -2 & 0 & 0 & -3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 2 \end{pmatrix}, \; x = \begin{pmatrix} 21 \\ 101 \\ -302 \\ 10 \end{pmatrix}$$

$$y = Bx = \begin{pmatrix} -2 \times 21 + 0 + 0 + (-3) \times 10 \\ 0 + 0 + 0 + 0 \\ 0 + 0 + 0 + 1 \times 10 \\ 3 \times 21 + 0 + 0 + 2 \times 10 \end{pmatrix} = \begin{pmatrix} -72 \\ 0 \\ 10 \\ 83 \end{pmatrix}$$

En este ejemplo vemos como los valores 101 y -302 no fueron usados para obtener el resultado del producto. Luego si el vector x fuera cambiado por $x' = \begin{pmatrix} 21 \\ 10 \end{pmatrix}$ y además se conociera el mapa de posiciones $map\_nzc = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$, se podría realizar el producto de la matriz por el vector y el resultado continua siendo el mismo, de la siguiente forma

$$y = B[\text{map\_nzc}]x' = \begin{pmatrix} b_{00}x'_0 + b_{02}x'_1 \\ b_{10}x'_0 + b_{12}x'_1 \\ b_{20}x'_0 + b_{22}x'_1 \\ b_{30}x'_0 + b_{32}x'_1 \end{pmatrix} = \begin{pmatrix} -2 \times 21 + (-3) \times 10 \\ 0 + 0 \\ 0 + 1 \times 10 \\ 3 \times 21 + 2 \times 10 \end{pmatrix} = \begin{pmatrix} -72 \\ 0 \\ 10 \\ 83 \end{pmatrix}$$

Del ejemplo anterior también se puede observar, que la segunda fila de la matriz está conformada completamente por ceros y por lo tanto para esta fila el resultado del producto es cero, sin necesidad de hacer cálculos como ya se mencionó.

Para encontrar las columnas iguales a cero, se implementó el método *"get_ nonzero_ cols"*, el cual retorna el mapa de posiciones *"mapnzcols"*, de las columnas diferentes de cero y su dimensión *"dimnzc"*. Con esta información se implementó otro producto, *"mat_ vec_ nonzero_ cols"*, el cual recibe el vector x reducido y el mapa de columnas diferentes de cero de la matriz.

También, se implementó un producto teniendo en cuenta ambas situaciones, las filas y las columnas diferentes de cero de una matriz esparcida, *"mat_ vec_ nonzero_ rows_ cols"*.

Es importante tener en cuenta que obtener el mapa de filas y el de columnas diferentes de cero debe ser obtenido antes de realizar el producto de la matriz por vector teniendo en cuenta las filas diferentes de cero, las columnas o ambas, según el caso.

Se espera que estos métodos sean útiles cuando se debe multiplicar una misma matriz esparcida por diferentes vectores un gran número de veces. Por ejemplo, en el caso de la solución de ecuaciones diferenciales por diferencias finitas.

Para una misma matriz únicamente es necesario calcular una sola vez el mapa de filas y de columnas diferentes de cero.

Tener en cuenta las filas y columnas diferentes de cero de una matriz esparcida, para tratar de simplificar las operaciones en el producto de una matriz esparcida por un vector, no son tomadas de un libro, sino que fueron concluidas en la práctica con matrices esparcidas.

2.2.1. CSR serial orientado a objeto

Al trabajar con matrices hay varias operaciones que se deben tener en cuenta para ser implementadas, tales como suma de matrices, multiplicación de una matriz por un escalar, la

transpuesta de una matriz, entre otras; también debe ser posible insertar, eliminar o cambiar elementos de la matriz.

Algunos de estos métodos implementados y un resumen de sus argumentos de salida fueron:

1. Lo primero que se debe realizar para trabajar con una matriz, es ingresar cada una de sus componentes. En nuestro caso, cada componente tendrá una misma tolerancia,para determinar si el valor de cada elemento es considerado como cero o no.
2. La fase para ingresar elementos de una matriz u objeto CSR, es la étapa de ensamble, *"start_ assembly"*. Luego de hacer el llamado de esta función podemos:
   - Cambiar la dimensión de la matriz con *"set_ dimension"*.
   - Sumar algún valor a una componente con *"add"*.
   - Eliminar componentes con *"erase"*.
   - Cambiar la tolerancia de una matriz que por defecto es 1×10^{-16}, con *"set_ tolerance"*.

   Cada componente de la matriz puede ser ingresada manualmente por el usuario con *"insert"*, ó todos los valores de las componentes de la matriz, pueden estar almacenados en un archivo de cualquier extensión, pero con el formato compatible para ser leído con la función *"read"*. El formato que debe tener el archivo, donde los datos deben ir separados únicamente por espacios o por "enter" es:
   - Dimensión de la matriz: Cantidad de filas y de columnas.
   - Por cada fila de la matriz: Cantidad de elementos diferentes de cero por fila, luego todos los elementos diferentes de cero, colocando primero la columna y luego el valor del elemento.
3. Cuando ya se tiene con certeza todas las componentes de la matriz que ya no serán cambiadas, pasamos a finalizar el ensamble de la matriz con *"end_ assembly"*.

4. Finalmente, tenemos el objeto de matriz CSR. Ahora podemos obtener información sobre la matriz o realizar operaciones con ella. La información que podemos obtener sobre una matriz es:
   - El valor del elemento a_{ij} con *"get_ aij"*.
   - La cantidad de elementos diferentes de cero por fila o de toda la matriz con *"get_ num_ nnz"*.
   - La cantidad de filas o de columnas de la matriz con *"get_ num_ rows"* o con *"get_ num_ cols"*, según el caso.
   - La estructura de datos para conocer las filas o columnas iguales a cero con *"get_ nonzero_ rows"* o con *"get_ nonzero_ cols"*.
   - La transpuesta de la matriz con *"get_ transpose"*.

   Las operaciones que podemos realizar son:
   - Multiplicar la matriz por un escalar.
   - Suma de matrices.
   - Diferentes métodos para multiplicar una matriz por un vector, en los cuales se asigna la respuesta a un vector($y = Ax$) o esta es sumada al vector de salida ($y = Ax + y$). La forma de diferenciar entre estos dos productos, es que el producto que suma el resultado al vector de salida, en la parte final del nombre que tiene la función en la implementación de estos productos lleve las letras *py*, por ejemplo *"mat_ vecpy"*, *"mat_ vec_ nonzero_ colspy"*, etc.

La implementación del CSR serial ha sido muy estable y dá los resultados esperados al efectuar alguna de sus operaciones. Esto se comprobó, al comparar con los resultados que se obtenían para los mismos datos y operaciones con MATLAB.

Para lograr estas pruebas se implementó un método que genera un archivo con un script compatible con MATLAB, el cual convierte un objeto de matriz CSR a una matriz "sparse" de MATLAB para el caso en que el resultado es una matriz, y para el caso de vectores también

se generó que el arreglo de datos sea compatible con el formato para vectores de MATLAB.

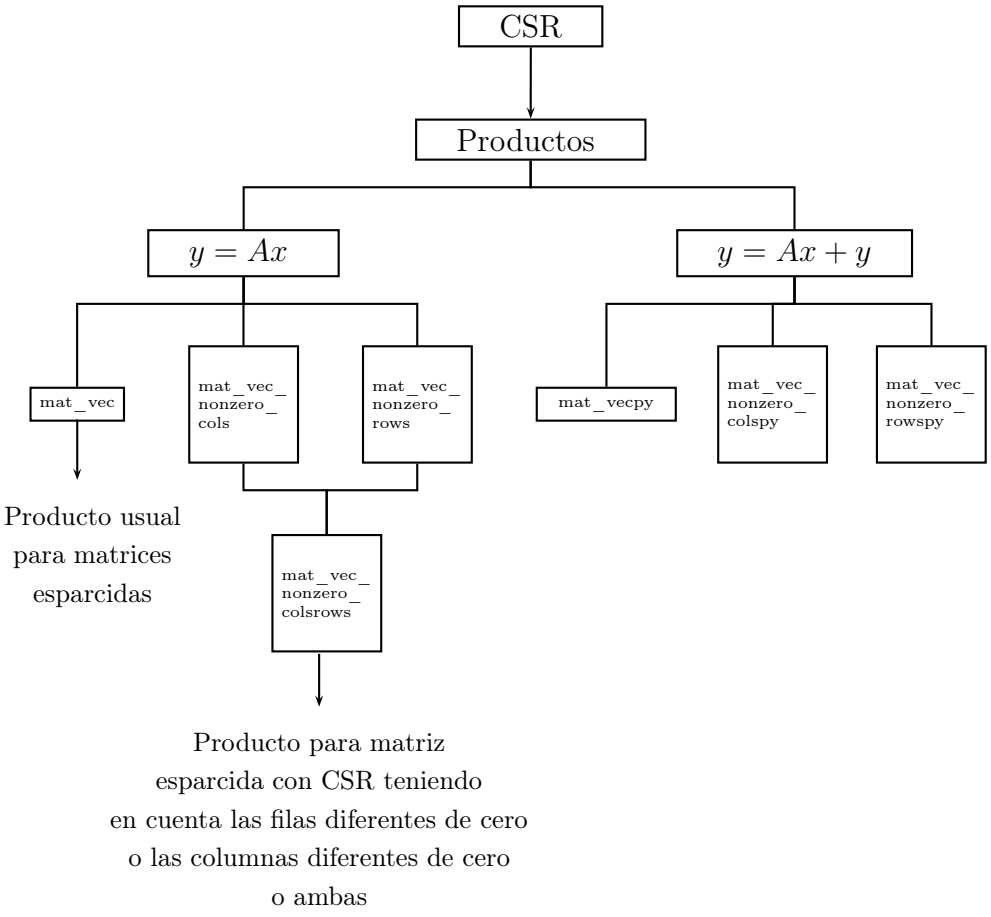

Figura 2–2: Diagrama de Productos implementados en la clase CSR

MATLAB incluye una gran cantidad de funciones para matrices esparcidas, entre ellas está *spy*, que nos permite graficar el patrón de esparcidad de las matrices. La comparación de resultados se hizo calculando la diferencia entre los dos resultados y usando las funciones normas de MATLAB para matrices y vectores, obteniendo siempre cero en el error. En estos se usaba la misma cantidad de cifras significativas, para así tener el mismo rango de comparación.

2.3. Matrices esparcidas en paralelo

La necesidad de operar con matrices esparcidas a gran escala nos lleva a usar computadores con procesadores buenos y alta capacidad de memoria, para lograr obtener resultados en un tiempo razonable.

Aunque las técnicas de almacenamiento en la mayoría de casos reduce la memoria necesaria para almacenar una matriz esparcida, para lograr obtener resultados bien aproximados a la solución esperada, las dimensiones de las matrices son tan grandes que la memoria de un computador se puede saturar bien sea por necesidad de almacenamiento o por los requerimientos de las operaciones a resolver.

Algunos de los problemas donde surgen matrices esparcidas manejan una gran cantidad de operaciones y de información, además también tienen la propiedad de poder dividir algunos cómputos de forma independiente de otros cómputos para lograr obtener soluciones necesarias. Esto genera la posibilidad de intentar solucionar estos problemas usando programación en paralelo y por tal motivo es útil que las estructuras y operaciones matemáticas requeridas para lograr la solución también se encuentren en paralelo.

Para p procesos la matriz esparcidad de orden $n \times m$ se divide manteniendo las posiciones de las filas, en p submatrices de orden $n_1 \times m$, $n_2 \times m, \ldots, n_p \times m$; donde $\sum_{i=1}^{p} n_i = n$, como se muestra en la figura 2–3. Cada submatriz es almacenada en un proceso y la partición de la matriz puede ser uniforme o puede ser dada por el usuario de acuerdo a su propio criterio de particionamiento.

A su vez cada submatriz se divide manteniendo las posiciones de las columnas, en p submatrices de orden $n_k \times m_1$, $n_k \times m_2, \ldots, n_k \times m_p$, como se ve en la figura 2–4; donde $\sum_{i=1}^{p} m_i = m$

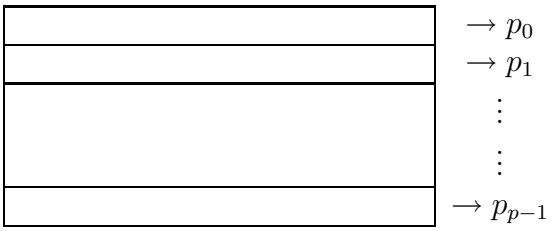

Figura 2–3: Distribución de matriz esparcida en p procesos por bloques de filas y k es el proceso donde está almacenada cada submatriz.

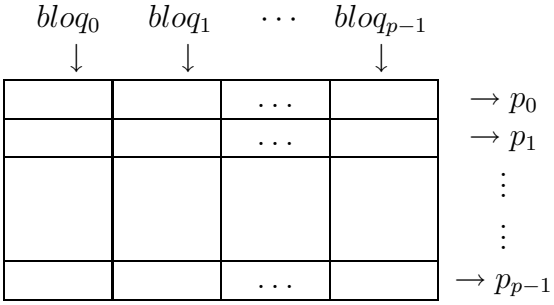

Figura 2–4: Matriz esparcida distribuida por proceso en p bloques

Cada uno de los valores que indican las dimensiones de la matriz y de la partición de filas y de columnas, son almacenados en todos los procesos. *"dim_ n"* y *"dim_ m"* son enteros con el valor del número de filas y de columnas de la matriz, *"local_ n"* y *"local_ m"* son dos arreglos de enteros que almacenan la dimensión de las filas y las columnas de cada bloque de matriz.

Todo proceso conoce la cantidad de filas de los otros procesos y la cantidad de columnas de cada bloque de matriz, esta cantidad de columnas es la misma en todos los procesos.

Cada proceso almacena p bloques de la matriz y los bloques que tienen valores diferentes de cero son matrices esparcidas que son almacenadas usando el formato CSR serial, los otros bloques quedan vacíos. La implementación para matrices esparcidas en paralelo es llamada PCSR.

Es importante notar que la forma en que se particionó la matriz nos permite por proceso, conocer aquellos otros procesos con los cuales es necesario hacer envío o recepción de elementos del vector x, para completar localmente cada producto del bloque de matriz con el vector.

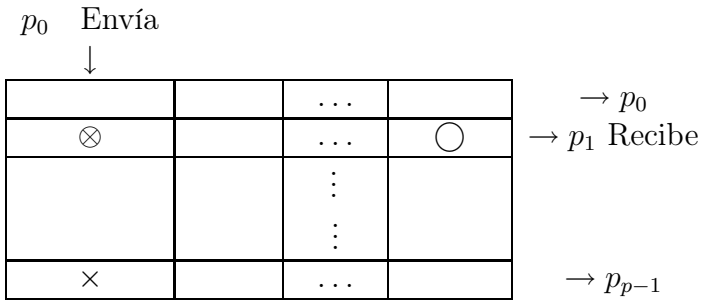

Figura 2–5: Distribución de tareas para matriz esparcida por bloques en paralelo

Por ejemplo, p_0 debe enviar datos a los otros procesos que no se encuentren vacíos señalados con $\times$, en la figura 2–5. Mientras que, p_1 debe recibir información de los procesos que almacenan señalados con $\bigcirc$ en el bloque 1, como se ve en la figura 2–5.

Al distribuir la matriz y hacer la partición de las columnas, cada bloque es almacenado a su vez con la estructura de datos CSR para una fila almacenando:

- La cantidad de bloques que no están vacíos *nbz* del proceso.
- Un arreglo *nzbloq* de tamaño *nbz*, con los indices de los bloques que no están vacíos en el proceso.

También, en la tesis se creó una estructura de datos independiente, para el bloque en la posición del rango del proceso proceso *my_rank*. Esto se hizo por que en el momento de hacer comunicación entre procesos, el bloque en la posición del rango del proceso no envía y no recibe información para completar la operación que se esté resolviendo, para éste bloque las operaciones se hacen localmente.

Para lograr el producto de la matriz esparcida por el vector en paralelo, el vector también es distribuido de acuerdo con la cantidad de columnas del bloque en la posición del rango del proceso.

Como lo que se quiere es realizar en forma eficiente la comunicación entre procesos para lograr el producto de la matriz esparcida y el vector que están distribuidos por los procesos, entonces se verán algunas de las cosas que se tuvieron en cuenta para lograr esta operación en la implementación.

Al igual que en el CSR serial se considera por cada bloque de matriz los arreglos de filas y de columnas diferentes de cero por cada proceso. Esto se logra al llamar cada bloque de matriz CSR no nulo a las funciones *"get_ nonzero_ rows"* y *"get_ nonzero_ cols"* del CSR serial. Cada uno de los arreglos de mapas es almacenado en una posición de los arreglos *"map_ nzero_ row"* y *"map_ nzero_ col"* de tamaño de la cantidad de bloques diferentes de cero del proceso. La respectiva dimensión de cada mapa de posiciones es almacenada en un arreglo para la cantidad de valores diferentes de cero por fila *"dim_ nzr"* y por columna *"dim_ nzc"*.

Todos los productos realizados en paralelo suman el resultado al vector de salida $y = Ax + y$, es decir que usan los productos del CSR serial que terminan con py, porque es la forma más eficiente de realizar el producto en paralelo.

Sin tener en cuenta aún la forma de comunicación de procesos, lo que se quiere es implementar al igual que en el CSR serial diferentes productos en los que se puede tener presente o no las filas y columnas no nulas de los bloques de matriz. Conocer esta información permite crear paquetes de vectores reducidos para ser enviados a los procesos que lo requieren.

Capítulo 3
VECTORES EN PARALELO

Este capítulo, incluye una descripción sobre la librería implementada para operar con vectores en forma serial o paralela

Para lograr el producto de matriz esparcida por vector en paralelo, se ha implementado una librería para vectores en paralelo PVector. Para formar un objeto de vector en paralelo sólo se necesita un arreglo "double" con los datos del vector y su correspondiente dimensión.

Dividir el vector entre procesos tiene varias opciones:

1. Crear un objeto PVector en cada proceso y agregar sus datos, ya sea al leerlos de un archivo que el proceso identifique ó insertando como elemento en forma manual.
2. Tener un arreglo double con los datos y usar la función *"part_vector"* que se implementó para un objeto PVector, la cual toma la parte de los datos que le corresponde a cada proceso según se indica en los parámetros ingresados a la función.

La clase PVector es compatible con las matrices esparcidas en paralelo PCSR; además para lograr que los datos de un vector en paralelo sean compatibles con el arreglo double que recibe las matrices esparcidas seriales CSR, la clase PVector es amiga "friend" de la clase PCSR y así se puede obtener el arreglo de datos con el método *"get_data"* que se pasará como argumento a los productos en CSR serial.

En la implementación realizada para vectores en la tesis la distribución de vectores en paralelo se hace en forma uniforme, esto es si $x \in \mathbb{R}^n$ y se distribuye en p procesos en forma consecutiva y el subvector de x que almacena cada proceso tiene dimensión n_i, con $i = \overline{0, p-1}$ y n_i es n/p en todos los procesos si p divide a n ó es $\lfloor n/p \rfloor + 1$ para los primeros r procesos y $\lfloor n/p \rfloor$ para los siguientes procesos, donde r es el residuo de n/p; por lo tanto se tiene que $n = \sum_{i=0}^{p-1} n_i$.

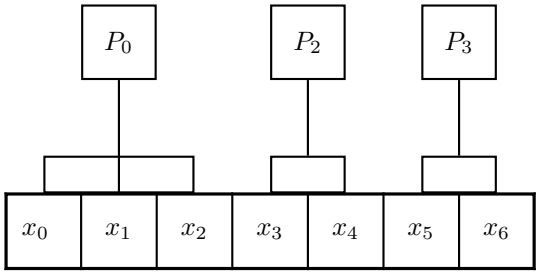

Figura 3–1: Ejemplo de distribución de un vector de 7 elementos en 3 procesos.

Al realizar en paralelo la suma de dos vectores $x \in \mathbb{R}^n$ y $y \in \mathbb{R}^n$, cada proceso tendrá una porción del vector x y del vector y. La distribución entre procesos de estos vectores se realiza en forma uniforme, entonces el subvector de x y y que almacena cada proceso tienen igual tamaño n_i, con $i = \overline{0, p-1}$. Por lo tanto, si x_i y y_i se encuentran en el proceso i, $x_i \in \mathbb{R}^{n_i}$ y $y_i \in \mathbb{R}^{n_i}$ y es posible obtener $x_i + y_i$ en el proceso i sin comunicación con otros procesos.

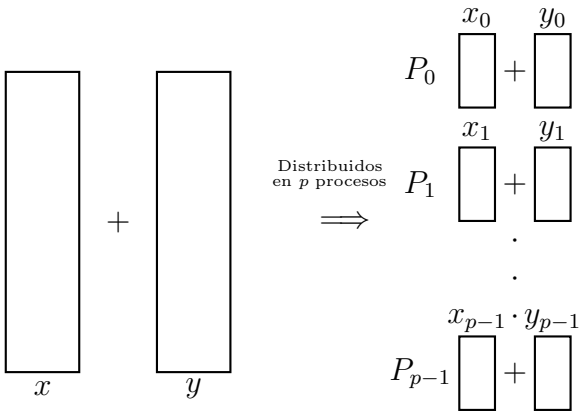

Figura 3–2: Suma de vectores en paralelo

Así mismo se pueden realizar otras operaciones como la multiplicación de un vector por un escalar sin necesidad de comunicación entre procesos.

Encontrar la norma de un vector es otra de las operaciones básicas de vectores. Ya sea la norma $L1$, $L2$ o $Infino$ se necesita comunicación entre procesos, para cada proceso tener el valor de la norma del vector. Por ejemplo, para obtener la norma $L2$ de un vector $x \in \mathbb{R}^n$ dada por $||x||_2 = \sqrt{\sum_{i=0}^{n-1} x_i^2}$, cada proceso debe calcular la norma $L2$ de la porción del vector que almacena y luego debe enviar el cuadrado de este valor a los demás procesos para que sea sumado con todos los valores que ellos están recibiendo de los procesos y con el que se cálculo localmente, como se puede observar en la figura 3–3; finalmente se realiza la raíz y cada proceso queda con el valor de la norma del vector. La comunicación es sólo de un número y cada proceso debe enviar este valor a todos los otros procesos.

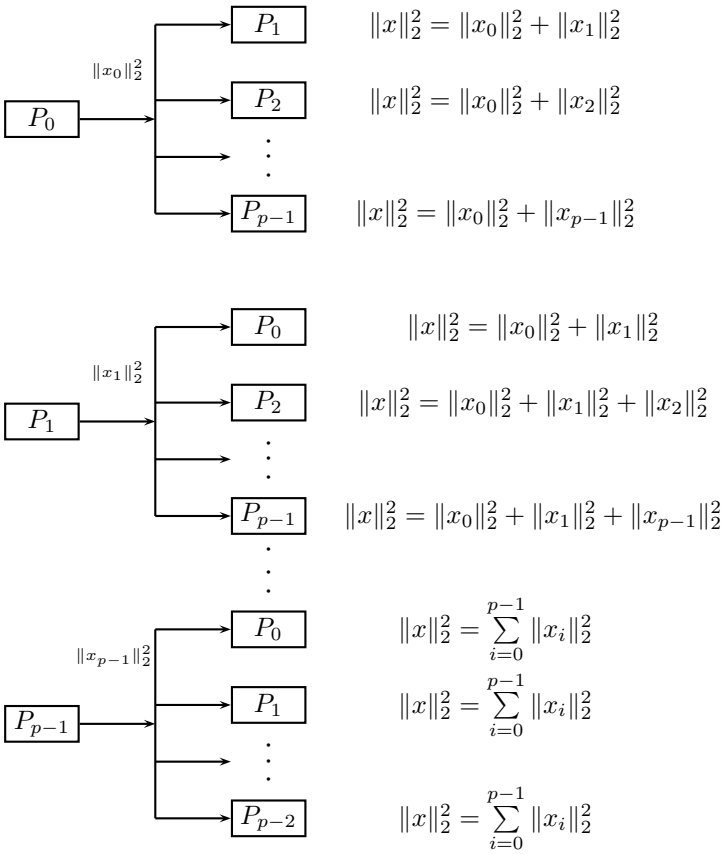

Figura 3–3: Comunicación entre procesos para el cálculo de $\|x\|_2$

Para realizar este tipo de comunicación, en la que un proceso debe operar con un valor que todos los demás procesos le van a enviar ó en la que todos los procesos se deben distribuir un valor con el que todos deben realizar una misma operación, MPI cuenta con dos funciones MPI_Reduce y MPI_Allreduce respectivamente. Estas funciones se encargan de recibir el valor de todos los procesos, almacenarlos en memoria y realizar con ellos la operación que se quiere, este valor es el que retorna al proceso o a los procesos que la esperan según el caso. Las operaciones que realiza esta función son el cálculo de máximos, mínimos, sumas, productos y operaciones lógicas.

Como se ha visto, las operaciones de vectores se pueden implementar en forma paralela sin necesidad de mucha comunicación. Por tal motivo la librería para vectores en paralelo PVECTOR, implementada para la tesis realiza las operaciones como suma de vectores, la multiplicación de un vector por un escalar, la norma $L1$, $L2$ ó INF de un vector y las funciones que se refieren al vector como inserción de componentes, obtener la dimensión o el valor de una componente del vector, se realizan en forma serial.

La pregunta es ¿porqué se dice que son vectores en paralelo?. Se realizó independiente de la clase PVECTOR, una librería de funciones para vectores en paralelo con el fin de realizar con ella las operaciones de vectores que necesitan comunicación entre procesos. Algunas de estas operaciones son el envío y recepción de arreglo de datos, distribuir un vector según una partición, encontrar la partición de un vector y el cálculo de producto punto o de normas.

La librería PVECTOR permite realizar operaciones con vectores serial o paralelamente, según como lo use el programador. Para usar las funciones paralelas, no se hace ningún tipo de declaración extra porque estas funciones están unidas a la clase PVECTOR al ser definidas como “friend"de la clase.

Existen Muchas librerías optimizadas para trabajar con vectores en paralelo. La finalidad de esta librería es usarla como parte de las pruebas, en el producto de matrices esparcidas con vectores y en la solución de la ecuación de transporte y de calor en 2D. Todos los vectores usados en las pruebas seriales y paralelas se definieron de tipo PVECTOR.

3.1. Uso de librería PVECTOR

La librería PVECTOR permite mucha flexibilidad al realizar cómputos que requieran de vectores al ser orientada a objetos y se acomoda fácilmente con la librería CSR y PCSR.

Se presenta un ejemplo de como se usa la librería PVECTOR, sumando dos vectores x y y en serial y en paralelo, donde el vector x es leído de un archivo y las componentes del vector y se generan en el archivo. Para el ejemplo en paralelo, se asume una cantidad p de procesadores.

Serial:

```
#include "Parallel_Vector.h"
using namespace std;
int main(int argc, char** argv){
    PVector x = PVector();
    PVector* y = new PVector();
    unsigned int dim;

    x.read("vectorx.txt");
    dim = x.get_dimension();
    y->set_dimension(dim);
    for(unsigned int j=0;j<dim;++j)   y->set_ai(j,j*j);
```

```
    y->paxpy(1.0,x);
    y->dump("sumaxy.txt");
    return 0;
}
```

Paralelo:

```
#include "Parallel_Vector.h"
using namespace std;
int main(int argc,char** argv){
MPI_Init(&argc, &argv);
    PVector x = PVector();
    PVector* y = new PVector();
    unsigned int dim, global_pos;
    int my_rank;
    string tmp;
    char text[20];

    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    sprintf(text,"vectorx-%d.txt",my_rank);
    tmp = text;

    x.read(tmp);
    dim = x.get_dimension();
    global_pos = x.return_global_position();
    y->set_dimension(dim);
```

```
    for(unsigned int j=0;j<dim;++j)
        y->set_ai(j,(global_pos+j)*(global_pos+j));

    y->paxpy(1.0,x);
    sprintf(text,"sumaxy-%d.txt",my_rank);
    tmp = text;
    y->dump(tmp);

MPI_Finalize();
}
```

Para conocer todas las funciones que incluye PVECTOR, mirar en el apéndice el manual de referencia.

Capítulo 4
ESTRATEGIAS DE COMUNICACIÓN

En este capítulo, se presentan las estrategias de comunicación que se usaron para lograr el producto de una matriz esparcida con un vector en paralelo.

Para lograr la solución computacional de muchos problemas científicos, se necesita de computadores que tengan la capacidad de llevar a cabo trillones de operaciones por segundo para no tener que esperar demasiado tiempo en obtener los resultados. También se requiere, que estos computadores posean una alta capacidad de memoria para almacenar información. ¿Qué tan costoso puede ser un computador con tales características, si lo hay?, ¿qué tan accesible para los grupos de investigación científica?.

En general, la imposibilidad de conseguir computadores con estas características, nos han llevado a usar otras estrategias para obtener resultados a problemas que requieren computadores potentes. Una de estas estrategias consiste en no asignar la tarea de cómputos a un sólo individuo muy potente, sino en dividirla en grupos a varios individuos que puedan resolver su labor en forma independiente y que luego se comuniquen para unificar las labores realizadas y alcanzar una solución global.

Con esta analogía, la estrategia para la solución de problemas de dimensiones enormes o que necesitan una gran cantidad de operaciones por segundo es clara: Usar varios procesadores y módulos de memoria a la vez.

En la tesis, se diseñó una implementación paralela que distribuye una matriz esparcida y un vector en p procesos, y permite realizar operaciones entre ellos o independientes.

El problema al efectuar el producto de una matriz esparcida, distribuida en p procesos y a su vez en cada proceso dividida en p bloques CSR, con un vector distribuido; es que para lograr completar el vector resultante, cada proceso necesita parte del vector o el vector completo que tienen almacenado los otros procesos y por tal motivo debe haber comunicación entre procesos al enviar y recibir datos. Si esta comunicación no se hace en forma correcta, la solución del producto de una matriz esparcida por un vector en paralelo podría tardar más en obtenerse que al realizarse en forma serial.

Para la implementación de esta tesis se usó programación en paralelo con las funciones de *openMPI* y el paradigma orientado a objetos de C++.

MPI "Message-Passing Interface", es una interfaz para la comunicación de mensajes en redes o en computadoras en paralelo. Este estándar fue realizado por el forum de MPI.

MPI no es un lenguaje de programación, sino una librería de funciones que puede ser llamada en un programa de C, C++ o Fortran. Esta librería esta fundamentada en un pequeño grupo de funciones que pueden ser usadas para lograr paralelismo por paso de mensajes "MP - Message-Passing". Una función de paso de mensajes, es una función que explícitamente transmite datos de un proceso a otro.

El paso de mensajes es un método poderoso y muy general para expresar paralelismo. Los programas con paso de mensajes pueden ser usados para crear programas en paralelo extremadamente eficientes, y además el paso de mensajes es un método de programación muy

usado habitualmente para muchos tipos de computadores paralelos.

La principal desventaja del paso de mensajes es que es muy complejo para el diseño y desarrollo de programas. De hecho, es llamado el "'lenguaje ensamblador para computación paralela" porque obliga al programador a hacer todo con mucho detalle.

Cada vez se desarrollan más programas que usan MPI, por tal motivo se han mejorado y diseñado algoritmos más sofisticados para ser usados en las librerías de MPI, llamándolos como funciones. Esto ha hecho que se creen versiones de MPI como *LAM/MPI* y *open MPI*, y en ellas se tienen mejoras con subversiones.

La programación en paralelo usada para la implementación, es para una arquitectura de red en paralelo con memoria distribuida. En un sistema de memoria distribuida, cada procesador tiene su propia memoria privada. En la figura 4–1 podemos observar un esquema general de un sistema de memoria distribuida.

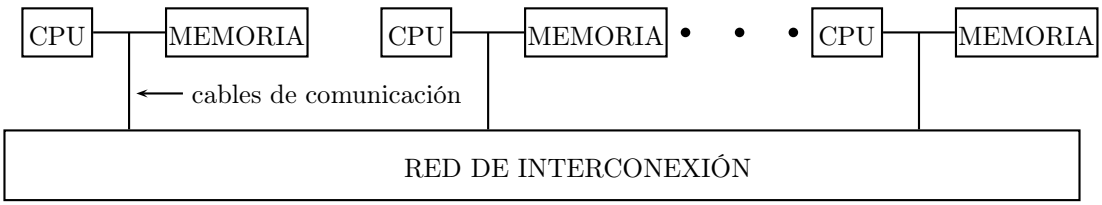

Figura 4–1: Sistema de Memoria Distribuida

Los sistemas de memoria distribuida pueden presentarse por medio de una red estática (mesh) o por una red dinámica (crossbar).

Para la red estática cada nodo consta de un par de elementos de memoria y procesador conectadas en forma fija. En la figura 4–2 los círculos representa los nodos. En la red dinámi-

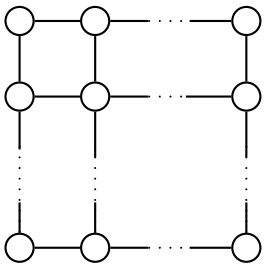

Figura 4–2: Memoria Distribuida por medio de red estática

ca algunos vertices corresponden a nodos y otros a interruptores que permiten intercambiar cada nodo con otro computador.

El cluster del departamento de matemáticas es un sistema de memoria distribuida dinámica, porque cada uno de los ocho computadores cuenta con su propio procesador y memoria, pero entre ellos la comunicación se realiza a través de un interruptor. En la figura 4–3 los círculos representan los computadores y los cuadros los interruptores. Las conexiones dinámicas son

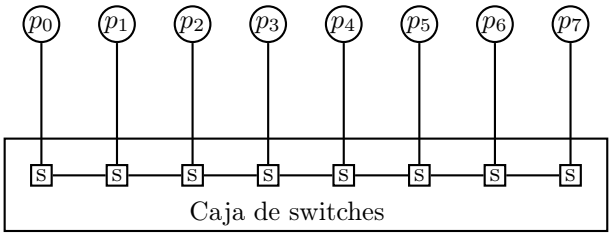

Figura 4–3: Conexión para el Cluster del Departamento de Ciencias Matemáticas

muy costosas y se ha diseñado diferentes formas de conexión para ciertas series de computadores.

Al programar en paralelo las funciones de MPI, son las funciones que comúnmente se usan para el paso de mensajes en sistema de memoria distribuida.

En la implementación que se realizó se asume que la cantidad de procesos es estática, es decir que el número de procesos involucradas en la ejecución de un programa es el mismo desde

que inicia hasta que finaliza de correr el programa. A cada proceso se le asigna un único número entero en el rango de $0, 1, ..., p-1$.

Los modelos de programas implementados en la tesis son llamados modelos de programas singulares con multiples datos (SPMD - Single Program Multiple Data Model). En este tipo de modelos, cada proceso corre el mismo programa ejecutable, pero los procesos ejecutan diferentes ramas o instrucciones en el programa determinadas por el rango del proceso.

Con la programación SPMD tenemos que aunque los programas corren el mismo ejecutable, según el rango de los procesos serán las operaciones que debe realizarse. Por lo tanto, no necesariamente los procesos tienen las mismas tareas por hacer. El programador es quien decide como se deben efectuar y repartir las tareas según la aplicación, para alcanzar la solución del problema deseado en el menor tiempo posible.

Al repartir las operaciones o tareas por proceso se debe tener mucho cuidado, porque se espera que todos los procesos tengan que efectuar casi la misma cantidad de operaciones por segundo, con el fin de evitar que unos procesos terminen sus tareas mucho más rápido que otros. La distribución de datos y tareas en lo posible debe hacerse en forma uniforme.

En el paso de mensaje básico, los procesos coordinan sus actividades de envió y de recepción de mensajes. La interface de paso de mensaje de MPI, tiene dos funciones básicas para estas actividades, “MPI_Send" y “MPI_Recev".

Cuando un proceso envía datos a otro proceso, la comunicación se puede hacer de dos formas, comunicación sincrónica o con un buffer de comunicación. Por ejemplo, si el proceso 0 envía

un mensaje al proceso 1, puede suceder que el proceso 1 no se encuentre listo todavía para recibir el dato enviado. Luego el envío del mensaje puede hacerse de tal forma que:

- con la comunicación sincrónica, el proceso 0 debe esperar y detener la ejecución de tareas hasta que el proceso 1 este disponible para recibir el dato. “MPI_Send" es una función sincrónica.
- con el buffer de comunicación, el contenido del mensaje es copiado en un bloque de memoria controlado (puede ser en el proceso 0 o en el proceso 1) y así el proceso 0 puede continuar sus tareas sin importar si el proceso 1 ya recibió su información.

El modelo de programa SPMD, permite declarar variables globales que todos los procesos almacenan con el mismo nombre y contenido, y variables locales que cada proceso define independiente de los otros procesos. Por ejemplo en la implementación del PCSR, el arreglo que almacena las dimensiones en la partición de los bloques de columnas es global, pero la variable que almacena la porción de vector en cada proceso es local y los otros procesos no tienen forma directa de conocer su contenido.

Otro caso que puede ocurrir en el envío y recepción de datos, es que el proceso que recibe se encuentre disponible primero que el proceso que envía el mensaje. Por tal motivo se hacen bloqueos y desbloqueos (blocking y nonblocking) en la comunicación. La función “MPI_Recv" es bloqueante, esto en el contexto del ejemplo anterior quiere decir que, cuando el proceso 1 llama “MPI_Recv" y el mensaje a recibir aún no está disponible, el proceso 1 estará ocupado para los demás procesos y no continuará sus tareas hasta que el mensaje del proceso que espera esté disponible.

Es importante tener en cuenta que las funciones “MPI_Send" y “MPI_Recv", únicamente envían o reciben mensajes de un sólo proceso a la vez.

Esta sincronización, bloqueo y limitación de comunicación a la misma vez con los procesos, es la que representa el problema principal de esta tesis, porque en la multiplicación de una matriz esparcida por un vector en paralelo con PCSR, los procesadores saben de quien tienen que recibir mensajes pero no saben a quien le deben de enviar mensajes. Esto ocurre porque hay bloques de matrices en los procesos que no tienen elementos y por lo tanto no necesitan conocer algún vector para efectuar el producto.

Con el fin de que el tiempo de comunicación y de ejecución total del programa en paralelo, para lograr el producto de una matriz esparcida almacenada con PCSR con un vector almacenado con PVECTOR, sea el menor posible se desarrollaron diferentes estrategias para la comunicación de los procesos:

- Broadcast: Transmite cierta información de cada proceso a todos los otros procesos.
- Window: Uso de memoria de acceso remoto para dejar o adquirir información.
- Schedule: Hace un horario u orden de tareas para enviar y recibir información.

Es posible otras formas de comunicación con funciones de MPI, pero con las que se observaron resultados de tiempos de cómputos fueron con las anteriores mencionadas. Queda como un trabajo futuro intentar atacar el problema con otras funciones y compararlas con las implementadas aquí.

4.1. Broadcast

Un patrón de comunicación que involucra todos los procesos en un comunicador *es una comunicación colectiva*. Así que, una comunicación colectiva debe involucrar más de dos procesos.

Si un proceso i tiene un mensaje que debe enviarse a todos los demás procesos en el comunicadoro si cada proceso de 0 a $p-1$, almacenan localmente un valor *"local_x"* y se necesita

que este valor sea conocido por todos los demás procesos del comunicador; para lograr la comunicación de los mensajes se podría hacer una comunicación de la siguiente forma:

- Para el primer caso, el proceso i puede enviar con "MPI_Send" el mensaje a cada uno de los otros procesos y ellos deberán llamar "MPI_Recv" del proceso i.
- Para el segundo caso, una opción es que cada proceso envíe el valor que almacena de la variable *"local_x"* a un proceso fijo que reunirá todos los valores y se encargará de distribuir luego el valor completo a los demás procesadores por medio de "MPI_Send" y "MPI_Recv".

Con esta forma de realizar la comunicación para ambos casos, se logra evitar el problema de la sincronización y el bloqueo de las funciones "MPI_Send" y "MPI_Recv" respectivamente, porque se conoce que proceso envía y todos los demás reciben de este. El problema en este caso, es que mientras el procesador encargado de recibir los datos está ocupado recibiendo el valor de *"local_x"* de cada proceso, los procesos que ya enviaron su información se encuentran detenidos esperando recibir el paquete completo de datos con el valor de *"local_x"* de todos los procesos que se le enviará; también a medida que se va enviando el dato van haber procesos que continúan en la espera mientras se termina la comunicación con los otros procesos y les toca su turno en el envío.

Con esta primera opción para la comunicación se logra la comunicación de los datos deseada, pero el costo computacional en el tiempo de comunicación será alto.

Hay otras técnicas que se han desarrollado para realizar este tipo de comunicaciones, una de ellas es realizar la comunicación por medio de un árbol estructurado.

Un árbol estructurado para la comunicación de datos, consiste en que todos los procesos envían su valor de *"local_x"* a un proceso encargado (podría ser el proceso 0) de agrupar los

valores y comenzar a distribuirlo, pero en este caso no a todos los procesos sino que a medida que entrega el valor a un proceso, ese proceso también se encarga de distribuir el valor a otro proceso y así continua la comunicación. En la figura 4–4, se observa un esquema de esta estructura de comunicación.

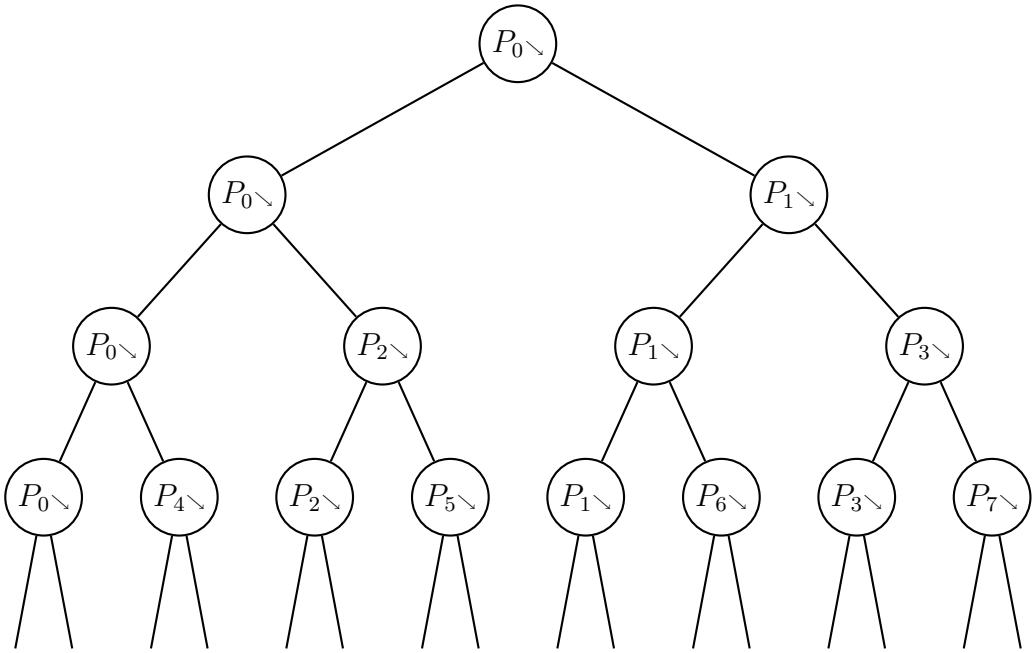

Figura 4–4: Árbol Estructurado para comunicación

Hay otras opciones que se pueden implementar con el fin de realizar comunicación colectiva. MPI incluye una función para estas comunicaciones, “MPI_Bcast".

Realizar un broadcast, es hacer una comunicación colectiva en la cual un proceso en particular envía el mismo dato a cada proceso en el comunicador. Como no se conoce en detalle la topología de cada sistema, MPI no asegura que el código para transmitir datos con broadcast es el más eficiente posible, pero al conocer con detalle el sistema puede ser optimizado. Aún así, esta opción de comunicación es más eficiente que las opciones que se han presentado.

Ahora para enviar un dato de un proceso hacia los otros procesos una excelente opción es usar la función “MPI_Bcast", siguiendo la sintaxis de la función es bien sencilla de usar. Así

con “MPI_Bcast" un proceso con rango *“root"* envía una copia de el mensaje deseado a cada proceso en el comunicador.

Para el ejemplo que se presentó anteriormente para la comunicación de los datos con broadcast, en el primer caso se tendría que el proceso i transmite el mensaje a todos los procesos directamente con el broadcast de MPI. Para el segundo caso, cada proceso transmite el mensaje con “MPI_Bcast" y así se logra la comunicación completa.

4.2. “Window”

La memoria de acceso remoto (RMA) extiende los mecanismos de comunicación de MPI permitiendo a un proceso especificar un parámetro para toda la comunicación, tanto para el envío como para la recepción de datos.

Esta forma de comunicación facilita la codificación de algunas aplicaciones que necesitan acceder a datos que cambian en forma dinámica para una distribución de datos fija. Así cada proceso decide a que dato necesita acceder o que dato debe actualizar. Sin embargo, un proceso puede conocer o no que datos de su propia memoria, pueden necesitar accederse o actualizarse por otros procesos remotos de los cuales se puede conocer o no su rango.

Las RMA permiten separadamente el envío y la recepción de datos, así la transferencia de parámetros se hace en un sólo sentido. Las funciones para la comunicación son:

1. *“MPI_PUT"*, se usa para escribir remotamente.
2. *“MPI_GET"*, se usa para leer remotamente.
3. *“MPI_ACUMULATE"*, Actualiza datos remotamente.

Estas funciones no son bloqueantes, así que una llamada inicia la transferencia, pero la transferencia puede continuar después de que es llamado el retorno.

Se puede especificar una operación colectiva para todos los procesos en un comunicador, creando una ventana - “window" en la memoria que es posible accederla por otros procesos remotos. Una ventana nos permite usar eficientemente las operaciones para RMA.

Una ventana es entonces una cierta cantidad de memoria que cada procesador crea, para realizar operaciones de acceso remoto que se especifiquen. En las ventanas se almacena un dato o un arreglo de datos, los cuales pueden ser accedidos o actualizados por ciertos procesos remotos.

Las funciones para escribir, leer o actualizar un dato de la ventana de un procesador, son las mismas funciones de RMA *“MPI_ PUT"*, *“MPI_ GET"* y *“MPI_ ACUMULATE"*.

Cada ventana que se crea tiene ciertos atributos, que son especificados por el programador, como la dirección inicial de la ventana, el tamaño de la información que almacena y el valor de desplazamiento en bytes para los valores usados, estos atributos no necesariamente son iguales en las ventanas de todos los procesos. MPI incluye la función *“MPI_ Win_ attr"* para retornar según el caso, uno de estos atributos.

Luego una ventana puede almacenar cualquier tipo de información, de cualquier tamaño ya sea para que los otros procesos lean lo que almacenan o para que ellos almacenen en ella cierta información.

No se tiene problema para acceder a la misma localización de una ventana, porque si una posición es actualizada al usarse una operación con *“put"* o *“acumulate"*, entonces esta posición no puede ser accedida por otra operación de RMA hasta que la operación de actualización se complete.

Realizar una comunicación de datos entre procesadores por medio del uso de ventanas, permite realizar comunicación entre procesos sin necesidad de conocer los procesos involucrados. Así al usarse la función de RMA *"MPI_GET"* o *"MPI_PUT"*, los procesos pueden no tener conocimiento de que proceso obtuvo información de la ventana o que proceso actualizó un nuevo valor. Esto es útil en el caso en que no se conoce cuales es el orden entre las operaciones de comunicación y entre quienes se debe efectuar. Si se necesita conocer que procesos se involucraron en la comunicación, el programador debe diseñar una estrategia para conocerlos.

Cuando un proceso no necesita adquirir o suministrar información a los demás procesos, como las ventanas deben crearse en forma colectiva, este proceso puede crear una ventana vacía.

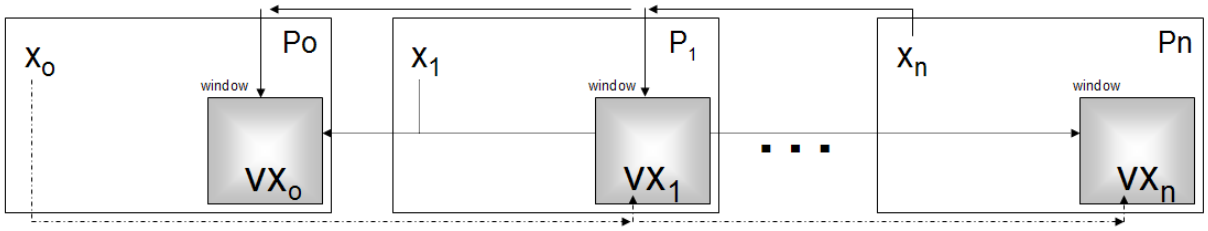

Figura 4–5: Ilustración del uso de window

Al usarse ventanas, puede ocurrir que varios procesos a la vez accedan a la información que la ventana almacena y que el proceso que tiene la ventana se encuentre realizando otra comunicación con otro proceso, sin detenerse las comunicaciones.

Una desventaja del uso de ventanas y las funciones RMA, es que no se conoce de que forma se hace la comunicación, sólo se usa como una caja negra que resuelve el problema del sincronización y bloqueo en la comunicación que tienen otras funciones para enviar o recibir datos. Además las ventanas fueron definidas en *"openMPI"* y sus versiones.

4.3. "Schedule"

Hay una gran cantidad de problemas en implementaciones en paralelo para los cuales todos los procesos del comunicador no necesitan comunicarse entre si o en los cuales cuando se realiza comunicación colectiva todos los procesos no comunican la misma cantidad de datos a los otros procesos. Esto hace que unos procesos terminen algunas de sus tareas de comunicación primero que otros, pero no pueden continuar con sus otras tareas hasta que los otros procesos con quienes aún no se ha comunicado, y debe hacerlo, se encuentren disponibles.

Si se organizan las tareas de comunicación en un orden tal que el tiempo que duren desocupados los procesos y el mayor tiempo de comunicación entre dos procesos sea el menor posible y además que durante toda la etapa de comunicación se encuentren ocupados el mayor número de procesos posible, lograríamos que la comunicación fuera eficiente.

Para hacer este orden de tareas se inicia con la idea de ocupar primero a los procesos que tienen las tareas de comunicación que toman más tiempo en comunicarse, ya sea para enviar o para recibir datos; teniendo presente que al hacerse envío o recepción de datos dos procesos se encuentran ocupados la misma cantidad de tiempo.

Al elegir los procesos que toman mayor cantidad de tiempo en comunicación, se deben tener en cuenta dos cosas el tamaño del mensaje a comunicarse y el tiempo que toma en comunicarse un mensaje cada par de procesos.

Como el tiempo de comunicación de un mensaje depende de los procesadores que se están comunicando y del tamaño del mensaje, es conveniente hacer una función que determine el tiempo de comunicación para un mensaje arbitrario y para cada par de procesadores. Esto se logra entre cada par de procesadores, enviando un mensaje con pocos "bits" y tomando

el tiempo que tarda la comunicación, luego duplicar la cantidad de “bits" del mensaje y enviarlo nuevamente tomando el tiempo, este proceso se hace una cantidad finita de veces y con los valores almacenados del tiempo de comunicación según el tamaño del mensaje, por interpolación se hace una función que estimará el tiempo de comunicación entre cada par de procesadores de un mensaje.

Se debe construir una función para estimar el tiempo de comunicación entre los procesadores de un comunicador cuando la red paralela es no uniforme, esto es que unos procesadores usan una tarjeta de red más hábil que otros, o tienen una topología diferente para la conexión de la red, entre otras opciones. En el caso que la red es bastante uniforme, se puede asumir que el tiempo de comunicación de un mensaje, tomará la misma cantidad de tiempo entre cualquier par de procesadores que lo comuniquen en la red y entonces el tiempo de comunicación sólo depende del tamaño del mensaje a comunicar.

A partir de estas ideas, se propone una estrategia para diseñar el orden de tareas en un comunicador, al conocer por proceso cuál es el tamaño de los mensajes a recibir o enviar y de quién o a quién se debe enviar. Esto se almacena en una matriz, llamada la matriz de envió y recepción de datos, en la que cada fila y columna representan un proceso (Ver figura 2–5). En los elementos de cada fila se almacena el tiempo de comunicación de los mensajes que se necesitan recibir, en la posición correspondiente del procesador que lo envía. Al observar la matriz por columnas, lo que se tiene es el tiempo de comunicación de los mensajes que cada proceso debe enviar a los procesos en la posición de estos elementos. Como los procesos no reciben, ni se envían mensajes a si mismos, entonces la diagonal de la matriz no tiene valores.

Como la estrategia consisten en que los procesos que toman más tiempo de comunicación de sus tareas inicien primero sus labores, teniendo en presente que cada tarea ocupa a un par

de procesos. Para conocer el tiempo de comunicación total de las tareas de cada proceso, se debe sumar el tiempo que toma recibir todos los mensajes y el tiempo que toma enviar todos los mensajes. En la matriz de envío y recepción de datos, es sencillo de ver porque será sumar cada elemento en la fila y la columna correspondiente a cada proceso. Como la diagonal es nula, se decidió aprovechar esta posición para almacenar el tiempo total de comunicación por proceso.

Si $t(c_{ij})$ es la función que determina el tiempo que tarda en comunicar el proceso p_i y el proceso p_j, un mensaje de tamaño c_{ij}, tenemos la matriz de envío y recepción de datos en la siguiente figura.

	p_0	p_1	$\cdots$	p_{p-1}
p_0	$\sum_{i=1}^{p-1} t(c_{0i}) + \sum_{i=1}^{p-1} t(c_{i0})$	$t(c_{01})$	$\cdots$	$t(c_{0p-1})$
p_1	$t(c_{10})$	$\sum_{i=0,i\neq 1}^{p-1} t(c_{1i}) + \sum_{i=0,i\neq 1}^{p-1} t(c_{i1})$	$\cdots$	$t(c_{1p-1})$
$\vdots$	$\vdots$		$\ddots$	$\vdots$
p_{p-1}	$t(c_{p-1,0})$	$t(c_{p-1,1})$	$\cdots$	$\sum_{i=0}^{p-2} t(c_{p-1,i}) + \sum_{i=0}^{p-2} t(c_{i,p-1})$

Figura 4–6: Matriz de envío y recepción de datos

Cuando se tiene el índice j del proceso con mayor tiempo de comunicación $\sum_{i=0,i\neq j}^{p-1} t(c_{j,i}) + \sum_{i=0,i\neq j}^{p-1} t(c_{i,j})$, lo siguiente que se hace es identificar el otro índice k de la tarea c_{jk} o c_{kj} que toma más tiempo de comunicación para enviar o recibir mensajes. Se logró identificar los procesos p_j y p_k que se encontrarán ocupados por un tiempo $t(c_{jk})$ si el proceso p_j recibe del proceso p_k ó $t(c_{kj})$ si es al contrario. Para ver luego que procesos continúan para realizar tareas, se deben poner en cero las filas y las columnas en los indices de los procesos p_j y p_k

que ya se encuentran ocupados y se realiza el mismo proceso para elegir el siguiente par de procesos.

Cuando todos los procesos posibles se encuentran ocupados, se debe tener en cuenta cual será el primer par de procesos que terminará sus tareas, para ponerlo nuevamente en disposición, agregando los valores en las filas y columnas de la matriz de envío y recepción de datos, de los tiempos de comunicación que estos procesos aún no han realizado y si es posible organizar una nueva tarea teniendo nuevamente el proceso con mayor tiempo de comunicación se hace, sino se espera hasta que otro par de proceso se desocupe y se inicia de nuevo el seguimiento de tareas. Cada tarea finalizada debe cambiar el tiempo de comunicación por cero.

El orden de tareas se almacena por proceso, veamos un ejemplo con 4 procesadores que nos ilustrará la idea del schedule.

Ejemplo. *Realizar un orden de tareas para la comunicación de 4 procesos, si se tiene la siguiente matriz de envío y recepción de datos:*

$$\begin{pmatrix} 76 & 15 & 20 & 0 \\ 10 & 57 & 0 & 21 \\ 26 & 11 & 68 & 5 \\ 5 & 0 & 6 & 37 \end{pmatrix}$$

Siguiendo el proceso que se describió anteriormente, el tiempo más alto en comunicación es 76*, que corresponde al proceso* p_0*; ahora la tarea que toma más tiempo al proceso cero es de* 26 *y es enviar al proceso* p_2*. Estos son los dos primeros procesos ocupados,* p_0 *debe enviar a* p_2 *y* p_2 *debe recibir de* p_0*.*

$$\begin{pmatrix} \underline{76} & \underline{15} & \underline{20} & \underline{0} \\ \underline{10} & 57 & \underline{0} & 21 \\ \mathbf{26} & \underline{11} & \underline{68} & \underline{5} \\ \underline{5} & 0 & \underline{6} & 37 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 21 & 0 & \mathbf{21} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 21 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Luego con los valores que nos quedaron, se tiene que el proceso p_1 debe recibir del proceso p_3. Con estas dos tareas asignadas, todos los procesos están ocupados hasta que los procesos p_3 y p_1 terminen su tarea que tarda un tiempo de comunicación de 21*, mientras que el tiempo de comunicación de la tarea asignada a los procesos p_0 y p_2 toma* 26*. Se debe esperar hasta que todos terminen para tener disponibles más tareas de comunicación.*

$$\begin{pmatrix} \underline{50} & \underline{15} & \mathbf{20} & \underline{0} \\ \underline{10} & 36 & \underline{0} & 0 \\ \underline{0} & \underline{11} & \underline{42} & \underline{5} \\ \underline{5} & 0 & \underline{6} & 16 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Nuevamente el proceso p_0 tiene el tiempo de comunicación más alto 50 *y la tarea que toma mas tiempo es* 20*. En este caso es proceso p_0 debe recibir del proceso p_2. Al ocuparse estos procesos como queda una matriz sin tareas los demás procesos deben esperar que finalicen sus tareas. Obtenemos una nueva matriz.*

$$\begin{pmatrix} \underline{30} & \mathbf{15} & 0 & 0 \\ \underline{10} & \underline{36} & 0 & 0 \\ \underline{0} & \underline{11} & 22 & 5 \\ \underline{5} & \underline{0} & 6 & 16 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 11 & 5 \\ 0 & 0 & \mathbf{6} & 11 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

En este caso primero tenemos que el proceso p_0 debe recibir del proceso $p1$ y en el segundo caso el proceso p_2 debe recibir del proceso p_3. Terminando sus tareas primero los procesos p_2 y p_3 y deben esperar los otros procesos.

Continuando con este proceso hasta terminar todas las tareas de comunicación, tenemos que el "schedule" o orden de tareas está dado por:

$p_0 \rightarrow$	$S(p_2)$	$R(p_2)$	$R(p_1)$	$S(p_3)$	$S(p_1)$
$p_1 \rightarrow$	$R(p_3)$	$S(p_0)$	$S(p_2)$	$R(p_0)$	
$p_2 \rightarrow$	$R(p_0)$	$S(p_0)$	$S(p_3)$	$R(p_1)$	$R(p_3)$
$p_3 \rightarrow$	$R(p_1)$	$R(p_2)$	$R(p_0)$	$S(p_2)$	

Donde $S(p_i)$ quiere decir que se debe enviar al proceso p_i y $R(p_i)$ indica que se debe recibir del proceso p_i.

Capítulo 5
PRODUCTOS IMPLEMENTADOS EN PARALELO

En este capítulo, se presentan los diferentes productos entre matriz esparcida y vector implementados con las estrategias de comunicación descritas en el cuarto capítulo.

Se han implementado para almacenar y operar con matrices esparcidas, las clases: CSR para usarse en programas seriales y PCSR para programas en paralelo.

En la implementación las matrices esparcidas en paralelo, se distribuyen en p bloques que son nulos ó son objetos CSR. Esto nos permite usar para cada bloque no vacío, un objeto de matriz esparcida CSR y así todas las operaciones que se implementaron para dicha clase . Por tal motivo la implementación serial, se realizó con demasiado cuidado y pasó por muchas pruebas antes de ser usada en la implementación en paralelo.

En el producto de una matriz esparcida por un vector en paralelo, como ya se mencionó debe haber comunicación entre los procesos. Esta comunicación se debe hacer en forma eficiente, porque de lo contrario el tiempo de cómputo aumentará con el número de procesos en vez de disminuir.

Cuando una matriz esparcida es distribuida en procesos y en cada proceso subdividida en bloques, puede ocurrir que algunos bloques queden nulos y para estos no se necesitará efectuar el producto con el vector; ya que una matriz para la cual todas sus componentes son

cero multiplicada por un vector, siempre produce el vector cero de resultado.

Cada proceso almacena una porción del vector con el cuál se esta efectuando el producto con la matriz esparcida. Esta porción del vector, es del tamaño del número de columnas de la matriz en el bloque del rango *rank* de cada proceso. La multiplicación de la matriz en el bloque *rank*, con el vector que almacena el proceso se puede llevar a cabo sin necesidad de comunicación.

Para lograr el producto de los otros bloques de matrices esparcidas, se necesitará que cada proceso cuyo rango es la posición de cada uno de estos bloques, envíen el vector que almacenan. Cada proceso debe enviar y recibir datos, pero los procesos solamente conocen de quien deben recibir información, pero no saben a quien le deben enviar información. Esto se debe a lo ya mencionado, sobre algunos bloques que quedan vacíos y no necesitan conocer el vector con quien multiplicar.

Ejemplo. *Se presenta una matriz B y un vector x, que serán distribuidos como se observa en* 4 *procesos. Luego se efectuará el producto Bx y se podrá observar que procesos se deben comunicar para obtener el resultado.*

$$B = \left(\begin{array}{cc|cc|cc|c} b_{00} & & & b_{03} & b_{04} & b_{05} & \\ & b_{11} & b_{12} & & & b_{15} & \\ \hline & & b_{22} & & & & \\ & & & b_{33} & & b_{35} & \\ \hline & b_{41} & & & b_{44} & & b_{46} \\ b_{50} & & b_{52} & & & b_{55} & \\ \hline & & & & b_{64} & & b_{66} \end{array}\right) \text{ y } x = \left(\begin{array}{c|l} x_0 & \Rightarrow p_0 \\ x_1 & \\ \hline x_2 & \Rightarrow p_1 \\ x_3 & \\ \hline x_4 & \Rightarrow p_2 \\ x_5 & \\ \hline x_6 & \Rightarrow p_3 \end{array}\right)$$

El producto asumiendo cada bloque no nulo como una nueva matriz esparcida B_{ij}*, con* $i, j = \overline{0,4}$ *es*

$$Bx = \left(\begin{array}{cccc} B_{00} & B_{01} & B_{02} & \\ \hline & B_{11} & B_{12} & \\ \hline B_{20} & B_{21} & B_{22} & B_{23} \\ \hline & & B_{32} & B_{33} \end{array}\right) \left(\begin{array}{c} X_0 \\ \hline X_1 \\ \hline X_2 \\ \hline X_3 \end{array}\right) \begin{array}{l} \Rightarrow p_0 \\ \Rightarrow p_1 \\ \Rightarrow p_2 \\ \Rightarrow p_3 \end{array}$$

$$= \left(\begin{array}{c} B_{00}X_0 + B_{01}X_1 + B_{02}X_2 \\ \hline B_{11}X_1 + B_{12}x_2 \\ \hline B_{20}X_0 + B_{21}X_1 + B_{22}X_2 + B_{23}X_3 \\ \hline B_{32}X_2 + B_{33}X_3 \end{array}\right) \begin{array}{l} \Rightarrow p_0 \\ \Rightarrow p_1 \\ \Rightarrow p_2 \\ \Rightarrow p_3 \end{array}$$

En este ejemplo observamos que para obtener el vector resultante en cada proceso, se debe realizar las siguientes comunicaciones:

1. *El proceso* 0 *necesita recibir el vector que almacena los procesos* 1 *y* 2*, pero no necesita recibir el vector del proceso* 3*. Además el proceso* 0 *debe enviar su vector únicamente al proceso* 2*.*
2. *El proceso* 1 *necesita recibir únicamente el vector que almacenan el proceso* 2 *y no necesita los vectores que tienen los procesos* 0 *y* 3*. También el proceso* 1 *debe enviar su vector a los procesos* 0 *y* 2*.*
3. *El proceso* 2 *debe recibir el vector que almacena todos los otros procesadores y también debe enviar su vector a todos los otros procesadores.*
4. *El proceso* 3 *sólo necesita el vector que almacena el proceso* 2 *y sólo debe enviar su vector al proceso* 2*.*

El ejemplo anterior, comprueba lo que se ha discutido acerca de que todos los procesos no necesitan recibir o enviar información a todos los otros procesos, además también nos muestra que la división por bloques en cada proceso de la matriz da información sobre cuales son los procesos que se deben comunicar.

Cada proceso conoce de que proceso debe recibir información, por que estos son los bloques no nulos que almacenan, pero no conocen a quien deben enviar. Si se observan por columnas en todos los procesos la posición de los bloques no nulos, se puede conocer el rango de los procesos a los cuales cada proceso debe enviar información.

Ejemplo. *Por medio de la distribución en bloques de la matriz B, se verificará comparando con las conclusiones en el ejemplo anterior que los bloques no nulos por filas dan la información de los procesos de los cuales se debe recibir información, y por columnas nos dice a cuales procesos se les debe enviar datos.*

	p_0	p_1	p_2	p_3
	Envía ⇓	Envía ⇓	Envía ⇓	Envía ⇓
p_0 Recibe ⇒	B_{00}	B_{01}	B_{02}	
p_1 Recibe ⇒		B_{11}	B_{12}	
p_2 Recibe ⇒	B_{20}	B_{21}	B_{22}	B_{23}
p_3 Recibe ⇒			B_{32}	B_{33}

Teniendo en cuenta que cada proceso no se envía, ni recibe de si mismo, tenemos a partir de la distribución por bloques de la matriz en cada proceso que:

1. *p_0 debe hacer envió a p_2, y p_0 debe recibir de p_1 y p_2.*
2. *p_1 debe hacer envió a p_0 y a p_2, y p_1 debe recibir de p_2.*
3. *p_2 debe hacer envió a p_0 a p_1 y a p_3, y p_2 debe recibir de p_0, p_1 y p_3.*
4. *p_3 debe hacer envió a p_2, y p_3 debe recibir de p_2.*

Para hacer la comunicación entre procesos, se usaron las estrategias que se explicaron en el capitulo 4 y para el producto los métodos explicados en el capitulo 2 para la clase CSR.

Los argumentos en todas las funciones implementadas para efectuar el producto de una matriz esparcida con un vector son: dos vectores en formato PVECTOR cada uno, el primero contiene los elementos con que se multiplicará la matriz y al segundo se le suma el resultado del producto del primer vector con la matriz. Esto hace que la comunicación de vectores, se realice en cada una de las funciones implementadas para efectuar el producto.

5.1. Productos usando Broadcast

En los primeros productos la comunicación entre procesos del vector, se realizó usando "Broadcast". De esta forma la comunicación es colectiva, así que en estos productos no se tiene en cuenta cuales son los vectores que cada proceso necesitaba recibir.

Al realizar la transmisión de cada uno de los vectores, cada proceso manteniendo el orden de los subvectores genera un vector global, que es el vector que inicialmente se distribuyó entre procesos. Teniendo en cuenta que ahora con el vector global en cada proceso, se puede efectuar sin ningún problema el producto de cada bloque no nulo de matriz, se usarán los productos de la clase CSR en los que se suma el resultado al vector de salida. Véase la siguiente figura.

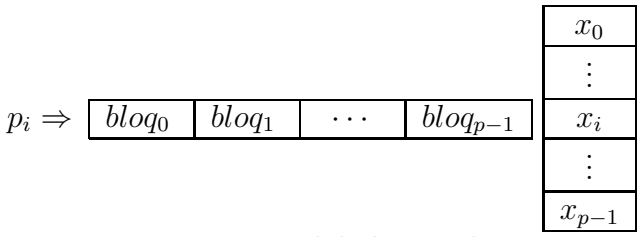

Figura 5–1: Vector global en cada proceso

5.1.1. "mat_vec"

Al generarse con broadcast un vector global, cada bloque no nulo se multiplica con la porción del vector que le corresponde usando en cada uno de los bloques de objetos CSR para el producto, la función *"mac_vecpy"*.

Aunque el envío del vector es completo, como la estructura de datos con la que se almacenaron los bloques es similar a la estructura de datos CSR, la multiplicación únicamente se hace con los bloques no vacíos. Por lo tanto puede haber parte del vector global que no es usada.

Este producto en la clase PCSR es llamado *"mat_vec"*.

5.1.2. "mat_vec_nonzero_cols"

Este producto usa el vector global que cada proceso generó por el broadcast, y de este vector se genera un nuevo subvector teniendo en cuenta las columnas diferentes de cero de cada matriz esparcida almacenada en los bloques del proceso.

Para obtener el mapa de columnas diferentes de cero, cada objeto CSR de los bloques usan la función *"get_nonzero_cols"* de la clase CSR. Como esto se hace por cada bloque no nulo, se creó una estructura de datos que consta de un arreglo de arreglos enteros cuyo tamaño es la cantidad de bloques no nulos y almacena en cada posición el mapa de arreglos que se obtuvo con la función *"get_nonzero_cols"* de CSR.

Obtener el mapa de columnas diferentes de cero de cada submatriz, se debe realizar antes de efectuar el producto que tiene en cuenta las columnas diferentes de cero. Por tal motivo, esta paso se implementó como un método a parte del producto y al igual que en el caso serial recibió el nombre *"get_nonzero_cols"*.

Es importante recordar que al usarse la función *"get_nonzero_cols"* para un objeto CSR, se retorna el mapa de posiciones y su dimensión. Por lo tanto, para cada bloque no nulo se almacena la cantidad de columnas diferentes de cero en un arreglo de enteros.

Esta información se almacena en las variables *"map_nzero_col"* y *"dim_nzc"*, respectivamente; estas variables son privadas, porque para una misma matriz estos datos no cambian y además nos dan información sobre el mapa de posiciones y la cantidad de elementos que los procesos deben recibir o enviar según el caso.

Finalmente, con la información adquirida por el método *"get_nonzero_cols"*, se crea a partir del vector global un subvector con los elementos en las posiciones de las columnas diferentes de cero y cada bloque de matriz es multiplicado con su correspondiente vector, usando la función *"mat_vec_nonzero_colspy"* de CSR.

Al igual que en el producto anterior, aunque se hizo la transmisión completa del vector global, sólo se usa las porciones correspondientes a bloques no nulos, tanto para obtener el mapa de columnas diferentes de cero como al efectuar el producto.

Este producto en la clase PCSR es llamado *"mat_vec_nonzero_cols"*.

5.1.3. "mat_vec_nonzero_rows"

Como en los dos anteriores productos, este se efectúa al completar cada proceso el vector global por medio de broadcast. Este producto tiene una estructura que no es muy diferente del anterior, teniendo en cuenta que en este caso se busca el mapa de filas diferentes de cero de cada bloque de matriz.

Conocer el mapa de filas o de columnas es independiente de si se a realizado la comunicación del vector global, luego la comunicación del vector se hace dentro de la función del respectivo producto.

Para obtener el mapa de filas, cada bloque no nulo usa la función *"get_nonzero_rows"* de la clase CSR y es almacenado en una estructura de datos como la usada para almacenar las columnas diferentes de cero de cada bloque. La variable que almacena el mapa de filas es *"map_nzero_row"* y la cantidad de filas diferentes de cero por bloques se almacena en *"dim_nzr"*.

La función para la clase PCSR que genera la estructura de datos, para las filas diferentes de cero por cada bloque se llama también *"get_nonzero_rows"* y debe ser usada antes de llamar el producto.

En la función que calcula el producto, se genera un subvector usando el mapa de filas diferentes de cero y cada bloque no nulo usa su correspondiente porción para efectuar el producto usando la función *"mat_vec_nonzero_rows"* de la clase CSR.

Este producto en la clase PCSR es llamado *"mat_vec_nonzero_rows"*.

Al usarse la función broadcast y con ella el envío innecesario de vectores, se logró realizar la comunicación de los procesadores para el producto pero con esta estrategia al aumentar la cantidad de procesadores también aumenta el tiempo de computo; esto se observará en el siguiente capitulo en las pruebas realizadas.

5.2. Productos usando window

Las ventanas son funciones de acceso remoto, que nos permite separar una porción de memoria en uno o en los procesadores deseados para almacenar datos que otros procesos colocan en la ventana ó que el propio proceso almacena para que los demás procesos lo adquieran sin interrumpir por comunicación, las actividades que ejecuta cada proceso.

Usando esta estrategia de comunicación se implementaron diferentes productos, con el fin de comunicar únicamente los vectores ó porciones de los vectores que cada proceso necesita.

En algunos de estos productos se debe usar de antemano funciones mencionadas en la sección anterior, *"mat_vec_nonzero_rows"* y *"mat_vec_nonzero_cols"*.

5.2.1. "mat_vec_win"

Como al tener la matriz esparcida subdividida en bloques en cada proceso, se ha notado que para lograr el producto con el vector, no se necesita la comunicación del vector que almacenan todos los procesos; cada proceso únicamente necesita recibir el vector que almacenan los procesos con rango igual al índice de los bloques no nulos.

En este producto, cada proceso adquiere los vectores que necesita de los demás procesos, al usarse como ventana en cada proceso el vector que almacenan; finalmente cada bloque no nulo efectúa el producto con su correspondiente porción de vector al usar la función *"mat_vecpy"* de la clase CSR.

Este producto únicamente puede ser usado con la librería de openMPI, que es quien incluye la función window y sus opciones. Esto es un limítate, porque aún hay muchos cluster que usan las librerías de LAM-MPI.

Una gran ventaja del uso de la ventana en el vector que almacena cada proceso, es que en forma sencilla con la función *"MPI_Get"*, los procesos que necesitan este vector lo toman y no afecta el trabajo que cada proceso se encuentre realizando en el momento estén tomando el vector.

La función de windows usada es no bloqueante en comunicación, esto permite que varios procesadores a la vez adquieran de una misma ventana el vector que almacena, sin que haya alguna espera de ningún tipo; pero la función de ventana es controlada por una opción que sólo permite finalizar las operaciones de ventanas a todos los procesos, cuando cada uno de los procesos involucrados de alguna forma con las ventanas creadas indiquen que se finalizó la comunicación.

Esta última observación es valida para todos los productos que se implementaron con el uso de las funciones de window y nos asegura que todos los procesos que entraron a una ventana, adquirieron el mismo vector.

La ventaja de realizar este producto es que al hacer la comunicación del vector con ventanas, cada proceso únicamente adquiere los vectores que necesita, que es diferente al producto *"mat_vec"* con el uso de broadcast en el que se forma el vector global con todos los vectores de los procesos.

Este producto en la clase PCSR es llamado *"mat_vec_win"*.

5.2.2. "mat_vec_nonzero_cols_win"

La comunicación entre procesos de los vectores en este producto, se hace de la misma forma que la del anterior producto. Así cada vector es una ventana, a la que los procesos que lo necesitan ingresan y hacen una copia de él.

Cuando cada proceso tiene los vectores que necesita, teniendo en cuenta el mapa de columnas diferentes de cero obtenido con la función *"get_nonzero_cols"* de la clase PCSR, forma subvectores que el correspondiente bloque usará para efectuar el producto con la función

"mat_vec_nonzero_colspy" de la clase CSR. En este producto, cada proceso adquiere el vector completo que necesita y luego de eso forma un nuevo subvector para usarlo en el producto.

Este producto en la clase PCSR es llamado *"mat_vec_nonzero_cols_win"*.

5.2.3. "mat_vec_elementnz"

La idea de comunicación del vector en este producto, está dada en que al usarse el producto *"mat_vec_nonzero_colspy"*, siempre se debe crear primero un subvector con los elementos en las posiciones de las columnas diferentes de cero de la matriz.

En los productos que se han discutido hasta ahora en las dos anteriores secciones que tienen en cuenta este hecho, se realiza la comunicación de los vectores y luego se forma el subvector. Lo que se quiere ahora es de alguna forma comunicar únicamente los elementos del vector que se tendrán en cuenta al usarse el producto *"mat_vec_nonzero_colspy"*.

Nuevamente cada proceso coloca como ventana al vector, y usando el mapa de posiciones de columnas no nulas, cada proceso irá recogiendo en los otros procesos elemento por elemento de los vectores que necesite y los irá almacenando en un nuevo vector, para usarse en el producto *"mat_vec_nonzero_colspy"* de la clase CSR.

En este caso se logra comunicar únicamente los valores que cada proceso necesita de los vectores almacenados en los otros procesos. El costo en este caso, es que se deben recoger elemento por elemento los valores necesitados en las ventanas y como para finalizar la comunicación todos los procesos deben haber finalizado el uso de las ventanas, el tiempo dependerá fuertemente de la mayor cantidad de elementos que un proceso deba recoger, es decir de la

mayor cantidad de columnas diferentes de cero entre todos los bloques de matrices esparcidas.

La forma de la comunicación que realiza las funciones con window, no es conocida y sólo se están usando como cajas negras que nos resuelven el problema de sincronización entre procesos, pero no se puede decir nada de la eficiencia.

Un caso extremo para este producto, es cuando por lo menos en un bloque de matriz esparcida, la cantidad de columnas diferentes de cero de la matriz sea la misma cantidad de columnas de la matriz. En estos casos no tiene sentido adquirir elemento por elemento sino el vector completo.

Este producto en la clase PCSR es llamado *"mat_vec_elementnz"*.

5.2.4. "mat_vec_winall"

El problema de que cada proceso reciba únicamente un subvector con los elementos en las posiciones de las columnas diferentes de cero, es que los procesos no saben a quien deben enviar. Así que hasta ahora lo que hemos logrado con las ventanas es que cada proceso que necesita recibir datos, entre por ellos a las ventanas de los procesos que necesita, ya sea por el vector completo o elemento por elemento de los valores que necesite.

Si de alguna forma cada proceso llega a conocer con cuales otros procesos debe hacer comunicación y cuales son las posiciones de los elementos que debe enviar, se logrará que la comunicación sea únicamente de los elementos deseados. Además para una misma matriz esta información no cambiará, aunque el vector si cambie; por tal motivo se creó para cada proceso una estructura de datos que almacena por proceso todos los correspondientes datos

adquiridos como valores privados de la clase PCSR y así sólo una vez se adquiere esta información.

Para realizar este producto primero se debe llamar por el momento a una función llamada *"get_nonzero_cols_all"*, la cual se encargará de distribuir a todos los procesos, cuales son los procesos a los que deben enviar parte del vector y las posiciones de estos elementos según el mapa de columnas diferentes de cero de cada bloque. Esta comunicación se hace usando las funciones windows de MPI.

En la función *"get_nonzero_cols_all"*, cada proceso crea una ventana para los arreglos privados *"dim_nzb_col"* y *"map_nzb_col"*. Luego en la posición de cada proceso que necesita que se le envíe parte del vector que esta en la ventana, se coloca con la función de windows *"MPI_Put"* la cantidad de elementos que necesita recibir y el correspondiente arreglo del mapa de posiciones.

Con la anterior información, cada proceso conoce los procesos a quienes debe enviar información y las posiciones del vector que debe enviar. Al efectuarse este producto, lo primero que se en los procesos es crear con la función para PVECTOR *"part_vector"* los subvectores que debe enviar, de acuerdo al mapa de posiciones que se almacenó en el arreglo *"map_nzb_col"*. Ahora cada proceso crea una ventana, en la cual los procesos de quien necesita recibir colocaran los vectores que formaron para el proceso y cuando se complete toda la comunicación entre ventanas, se realiza por cada bloque no nulo en cada proceso, el producto *"mat_vec_nonzero_colspy"*.

Con este producto se logró por primera vez que los procesos envíen cierta porción del vector que almacenan a los otros procesos que lo requieren. Pero el costo de lograr esto, es un excesivo uso de las funciones window.

Este producto en la clase PCSR es llamado *"mat_vec_win_all"*.

5.3. Productos usando schedule

La idea de usar schedule, es realizar un orden entre procesos para el envío y recepción de datos, teniendo en cuenta la cantidad de información y el tiempo que cada proceso toma en comunicar las tareas de todos los procesos.

En nuestro caso lo que se quiere es que cada proceso comunique del vector que almacena, un nuevo vector con los elementos en las posiciones del mapa de columnas no nulas para cada proceso que lo necesiten. Luego el schedule se aplica según la cantidad total de elementos que cada proceso debe comunicar y recibir, teniendo en cuenta que para enviar se forman subvectores de tamaño el número de columnas diferentes de cero de los procesos a quienes se les envían.

Formamos la matriz de envío y recepción de datos, con la cantidad de elementos que cada bloque de matriz debe recibir. En la matriz de envío y recepción de datos, como ya se ha mencionado, las columnas nos da por proceso la información de procesos a los que se debe hacer envío de datos y la cantidad, y las filas son los procesos de quien se debe recibir subvectores y la dimensión de los subvectores.

Para ser más general los valores en la matriz de envío y recepción de datos, deberían ser en vez de la cantidad de datos que se van a recibir o enviar, el tiempo que toman los procesos en

la comunicación de esa cantidad de datos. En nuestro caso, en los cluster de matemáticas y de ingeniería, se está usando por computador un sólo proceso y la comunicación es uniforme, así que al no tomarse el tiempo sino la cantidad de elementos no hay mucha diferencia. En redes paralelas, donde los procesos tengan mucha diferencia ya sea en memoria, velocidad del procesador o espacio en disco, se debe realizar una función para el tiempo de comunicación.

En PCSR se creó una estructura de datos privada en la que cada proceso calculará la matriz de orden de tareas con la clase schedule y usará la fila en la coordenada del rango del proceso.

Para usar el schedule, primero cada proceso debe comunicar a los otros procesos un vector con la cantidad de elementos que cada bloque de matriz debe recibir según la cantidad de columnas diferentes de cero que tiene; luego cada proceso forma una matriz de envío y recepción de datos en la cual para cada posición de su diagonal se tiene la suma de la fila y la columna correspondiente. Finalmente se crea un objeto SCHEDULE y se aplica la función *"find_schedule"*.

En la clase PCSR la función que completa la estructura de datos para el orden de tareas en cada proceso, es llamada *"assembly_schedule"*. Esta función para una misma matriz solo se calcula una vez y además para los productos que usan schedule, siempre debe ser llamada primero para conocer el orden en que será comunicado los vectores entre procesos.

Para comunicar a cada proceso, un vector con sólo las componentes deseadas por las columnas diferentes de cero de cada bloque de matriz, se tiene en cuenta que la estructura de datos para el mapa de columnas diferentes de cero de todos los procesos que se deben comunicar debe ser completada, ya sea con la función de la clase PCSR *"get_nonzero_cols_all"*, mencionada en la sección anterior con el uso de windows, o con una nueva función con la misma

finalidad de la anterior pero la comunicación la hace con el schedule encontrado para cada proceso. Esta nueva función es *"get_nonzero_cols_sch"*.

Con el fin de comparar si el orden de tareas según la cantidad de columnas diferentes es mejor o no que teniendo en cuenta recibir el vector completo, la clase PCSR tienen una opción para hacer el orden de tareas con la matriz de envío y recepción de datos con la dimensión de las columnas de cada bloque de matriz no nulo. Esta función es llamada *"assembly_schedule_dim"*.

5.3.1. "mat_vec_schedule"

En los productos que usan schedule la comunicación del vector se hace de acuerdo a como lo dice el orden de tareas para enviar o recibir vectores. Cada subvector es organizado en cada proceso para enviarlos a los procesos que lo requieren según la estructura de datos que almacena el mapa de columnas diferentes de cero; luego se sigue el orden de tareas que dice el schedule de enviar el vector correspondiente o recibirlo al proceso que indica el orden de tareas, usando las funciones *"MPI_Send"* y *"MPI_Recv"* de MPI.

Luego que se completa todo el envío y recepción de datos, se efectúa el producto de la clase CSR *"mat_vec_nonzero_cols_py"*.

Este producto en la clase PCSR es llamado *"mat_vec_schedule"*.

5.3.2. "mat_vec_schedule_recv_calc"

En el anterior producto, luego de que cada proceso finalizaba las tareas de envío y recepción de datos calculaban los productos de cada bloque con sus correspondientes vectores. En este

producto por el contrario, inmediatamente se recibe un vector se realiza el producto usando la función *"mat_vec_nonzero_cols_py"* de la clase CSR, entre el bloque de matriz en la posición del proceso que envía y el vector recibido.

Este producto en la clase PCSR es llamado *"mat_vec_schedule_recv_calc"*.

5.3.3. "mat_vec_schedule_all"

Para los dos productos anteriores los vectores que son comunicados, son porciones del vector que almacena cada proceso. En este caso se hace el envío completo del vector en cada proceso, a los otros procesos que lo necesitan manteniendo el orden dado por el schedule.

Como cada bloque de matriz recibe el vector completo que necesitaba para el producto, en este caso se decidió usar la función *"mat_vec"* de la clase CSR.

Este producto en la clase PCSR es llamado *"mat_vec_schedule_all"*.

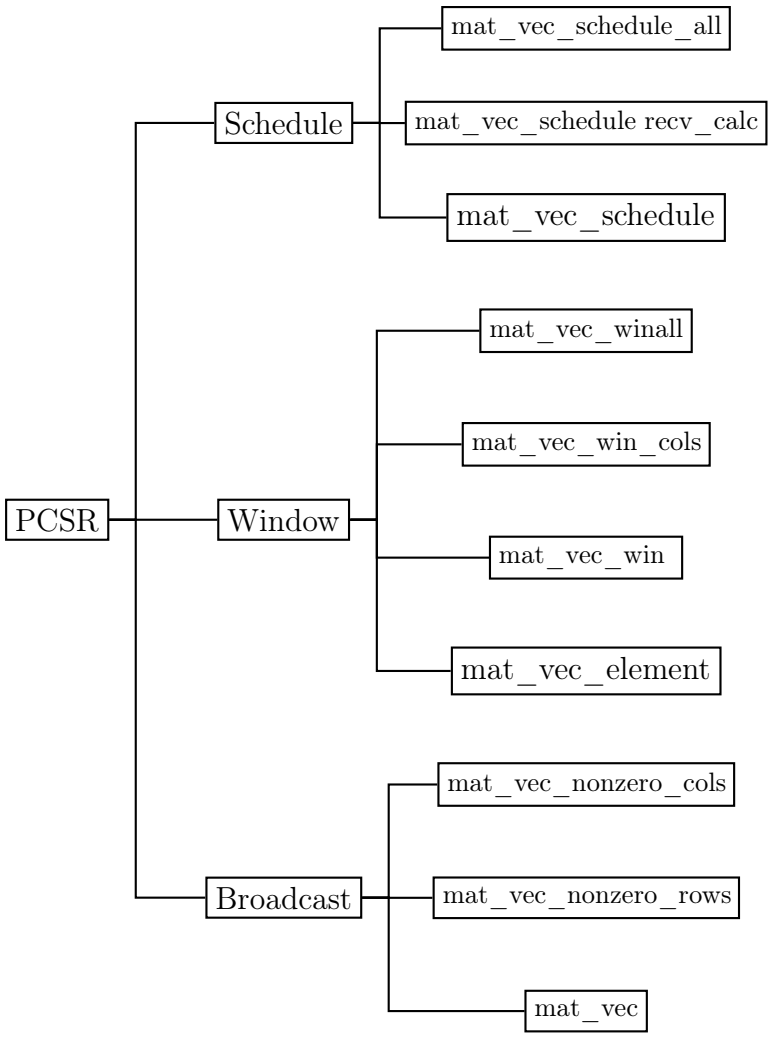

Figura 5–2: Diagrama de productos implementados para PCSR

Capítulo 6
PRUEBAS

En este capítulo, observamos diferentes pruebas con los productos implementados con el fin de ver la eficiencia en el tiempo de ejecución. Estas pruebas se realizaron en el cluster del Departamento de Matemáticas (8 nodos, 8 CPUs de 4G de memoria cada una y con procesadores Intel Xeon 64 bits $4 \times 3{,}0GHz$ *y* $4 \times 3{,}2GHz$*) y en un cluster del Departamento de Ingeniría (65 nodos, 130 CPUs de 1G de memoria cada una y con procesadores Intel-386 de 120 MHz)*

6.1. Solución de la ecuación de transporte de onda en 2D

La ecuación de transporte de onda, es una ecuación diferencial hiperbólica y está dada por la ecuación

$$u_t + au_x + bu_y = 0 \tag{6.1}$$

donde a y b son constantes que representan la velocidad de propagación de la onda en el eje x y en el eje y respectivamente, y x y y representan las variables del espacio. Los subíndices denotan diferenciación, es decir, $u_t = \frac{\partial u}{\partial t}$.

A partir de una condición inicial para el tiempo en la ecuación 6.1, $u(0, x, y) = u_0(x, y)$, se desea determinar los valores de $u(t, x, y)$ para todo valor de t. La solución de este problema de valor inicial es

$$u(t, x, y) = u_0(x - at, y - bt), \tag{6.2}$$

que se obtiene por el método de características y se verifica inmediatamente al evaluarla en la ecuación 6.1.

Se va a usar una aproximación de la ecuación de onda por medio de un esquema de diferencias finitas, para verificar por medio del cálculo de error y del orden de convergencia del esquema de aproximación usado, que las operaciones que efectúa las librerías implementadas en esta tesis son correctas.

Para aproximar la solución de la ecuación (1), con condición inicial $u_0(x, y)$ en la región $[x_0, x_f] \times [y_0, y_f]$; se usará diferencias finitas adelantadas en tiempo y en espacio ("Forward in time - forward in space", FTFS). La discretización es:

$$\frac{u_{l,m}^{n+1} - u_{l,m}^{n}}{\Delta t} + a\frac{u_{l+1,m}^{n} - u_{l,m}^{n}}{\Delta x} + b\frac{u_{l,m+1}^{n} - u_{l,m}^{n}}{\Delta y} = 0,$$

es decir,

$$u_{l,m}^{n+1} = u_{l,m}^{n} - a\frac{\Delta t}{\Delta x}(u_{l+1,m}^{n} - u_{l,m}^{n}) - b\frac{\Delta t}{\Delta y}(u_{l,m+1}^{n} - u_{l,m}^{n}).$$

Manteniendo la notación usada en el libro [**?**], sean $\lambda_x = \frac{\Delta t}{\Delta x}$ y $\lambda_y = \frac{\Delta t}{\Delta y}$, entonces el esquema es

$$u_{l,m}^{n+1} = (1 + a\lambda_x + b\lambda_y)u_{l,m}^{n} - a\lambda_x u_{l+1,m}^{n} - b\lambda_y u_{l,m+1}^{n} \tag{6.3}$$

Este esquema es consistente y el orden de convergencia es $(1, 1)$. Pero para que se tenga la convergencia en la solución se debe asegurar la estabilidad. Para el caso en que $\lambda_x = \lambda_y = \lambda$, se debe cumplir que $-1 \leq (a + b)\lambda \leq 0$ para que se tenga estabilidad en la solución con el esquema en 6.3.

Con la finalidad de usar el esquema 6.3, como parte de las pruebas de la implementación realizada en esta tesis, expresamos el esquema de diferencias finitas en forma matricial. Se

obtiene que $u^{n+1} = Au^n + b$, donde u^n es el vector solución de la ecuación $u(x, y)$ en el tiempo $t = n\Delta t$, A es una matriz cuyos valores dependen del esquema y b es un vector que depende de las condiciones de frontera.

A es una matriz de orden $m_x.m_y \times m_x.m_y$, donde m_x y m_y son la cantidad de subintervalos en que fueron particionados los intervalos $[x_0, x_f]$ y $[y_0, y_f]$. Los elementos de la diagonal principal de A son el valor constante $1 + (a + b)\lambda$, la banda exactamente sobre la diagonal principal de A toma los valores $-b\lambda$ ó 0 en los casos que indica la condición de frontera, y se tiene una última banda superior a una distancia my de la diagonal principal, cuyos valores son la constante $-a\lambda$. Todas las demás posiciones de la matriz A son ceros y por tanto A es una matriz esparcida. Se debe notar, que la matriz A es la misma en cualquier iteración del esquema.

Los valores de los elementos del vector b, cambian con cada paso de tiempo o iteración del esquema. Esto sucede, porque este vector controla los valores $u^n_{l,m}$ que dependen de la condición de frontera. Para esta prueba, se tomó como condición de frontera la solución exacta para estos elementos.

Para que la aproximación de la solución de la ecuación 6.1 sea muy cercana a la solución exacta 6.2, el mallado del dominio debe ser muy fino. Como las dimensiones de la matriz A dependen de la cantidad de subintervalos en que se divida el eje x y el eje y, entonces entre más fino sea el mallado mayor será la cantidad de elementos de la matriz A. Esto hace que la matriz A sea a gran escala y se necesite distribuirla en paralelo para disminuir el tiempo de ejecución en la discretización.

Al ubicarlos valores de $u^n_{l,m}$ en una matriz U e identificar la condición de frontera que se necesita para el esquema FTFS en 6.3, se decidió hacer la partición de los datos para buscar

la solución del problema dividiendo esta matriz en forma uniforme; por ejemplo en la figura 6–1, se tiene la partición de una matriz u con el mallado del dominio en 10 subintervalos para el eje x y para el eje y, y distribuida en 3 procesos.

$$
\begin{array}{c}
\hspace{6cm} CF \\
\hspace{6cm} \Downarrow \\
\text{U=}\left(\begin{array}{ccccc}
u_{00} & u_{01} & \cdots & u_{09} \ \downarrow & u_{0,10} \\
u_{10} & u_{11} & \cdots & u_{19} \ \downarrow & u_{1,10} \\
u_{20} & u_{21} & \cdots & u_{29} \ \downarrow & u_{2,10} \\
u_{30} & u_{31} & \cdots & u_{39} \ \downarrow & u_{3,10} \\
\hline
u_{40} & u_{41} & \cdots & u_{49} \ \downarrow & u_{4,10} \\
u_{50} & u_{51} & \cdots & u_{59} \ \downarrow & u_{5,10} \\
u_{60} & u_{61} & \cdots & u_{69} \ \downarrow & u_{6,10} \\
u_{70} & u_{71} & \cdots & u_{79} \ \downarrow & u_{7,10} \\
\hline
u_{80} & u_{81} & \cdots & u_{89} \ \downarrow & u_{8,10} \\
u_{90} & u_{91} & \cdots & u_{99} \ \downarrow & u_{9,10} \\
\rightarrow & \rightarrow & \rightarrow & \rightarrow & \\
u_{10,0} & u_{10,1} & \cdots & u_{10,9} & u_{10,10}
\end{array}\right)
\begin{array}{l}
\Rightarrow P_0 \\ \\ \Rightarrow P_1 \\ \\ \Rightarrow P_2
\end{array} \\
\Uparrow \\
\text{Condición de Frontera}
\end{array}
$$

Figura 6–1: Distribución en 3 procesos de la matriz de nodos de soluciones U

A cada proceso le corresponde un bloque de la matriz U y a partir de estos valores se debe formar en cada proceso la porción de la matriz A y del vector correspondiente b. En la figura6–2, se observa como sería las porciones de matriz y vector para cada uno de los tres procesos de la figura 6–1.

Como para determinar la solución de la ecuación 6.1 con el esquema de diferencias finitas 6.3 en forma matricial, se debe realizar el producto de una matriz esparcida A fija con un vector u^n que cambia en cada iteración, se usaron las librerías PCSR y PVECTOR que se implementaron en este proyecto de tesis, para buscar la solución de la ecuación de transporte en 2D paralelo.

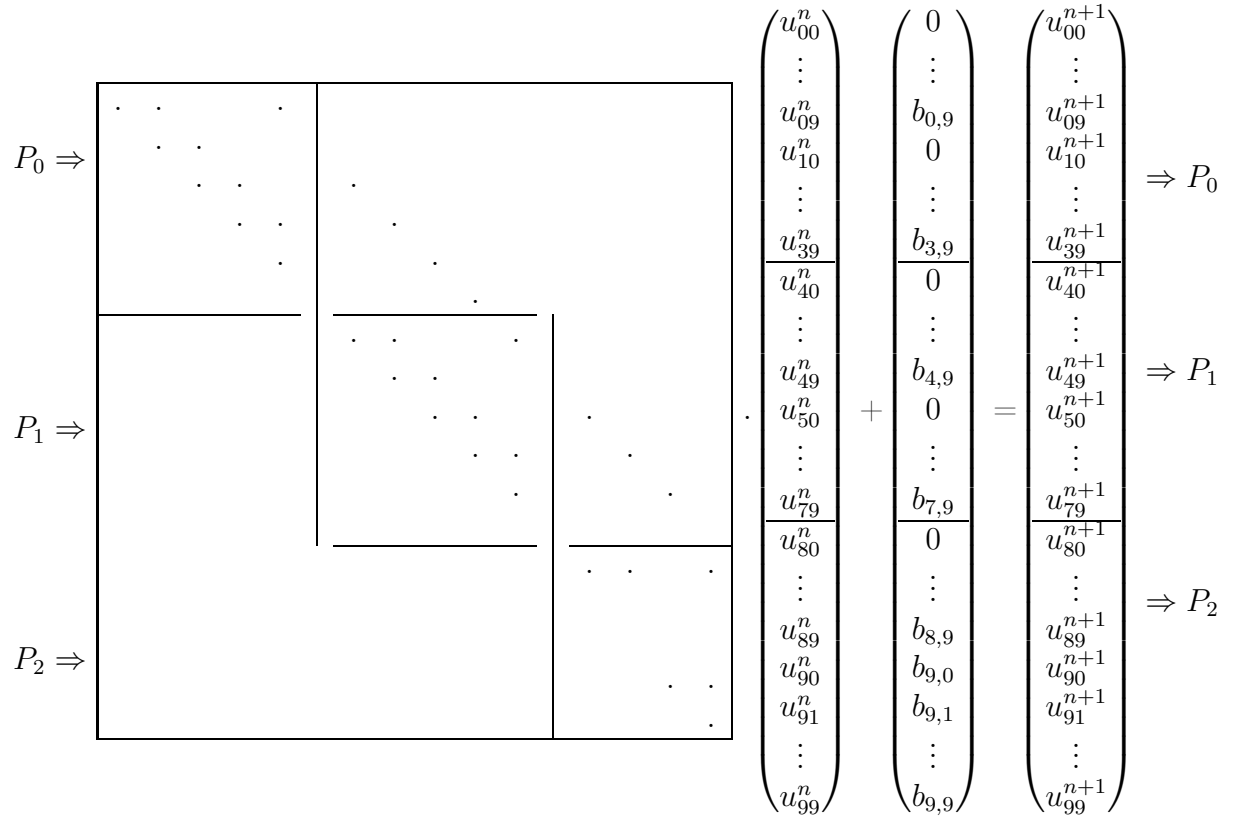

Figura 6–2: Estructura por proceso de la matriz A y el vector b del esquema matricial

El ensamble de los datos por cada proceso se hace en forma local y la comunicación que se necesita, sucede por el producto de la matriz esparcida A con el vector u^n en cada iteración y al final para verificar el error al calcularse la norma del vector resultante menos el esperado.

Con el fin de verificar que cada proceso estaba generando el bloque de matriz que le correspondía, se implemento en la clase CSR un método que a partir de los archivos con la porción de matriz que cada proceso ensambló, se unen los bloques de matrices y se forma un objeto de matriz CSR; la cual al compararla con la matriz A que se generó en el esquema serial, restándolas y calculando una norma de matrices, se comprobó su igualdad.

En las pruebas tenemos que la aproximación de la solución se calculó varias veces, usando en cada prueba los diferentes productos implementados en PCSR, con el fin de comparar el

tiempo esperado para obtener la solución de la ecuación de transporte en 2D con las diferentes estrategias seguidas en la implementación de los productos.

En la implementación de las clases serial y paralela para solucionar la ecuación 6.1 con la forma matricial del esquema 6.3, se tiene en los métodos la siguiente estructura:

1. Constructor con los parámetros de entrada para la ecuación a aproximar. Δx, x_0, x_f, Δy, y_0, y_f, λ, a, b, t y $u_0(x, y)$.
2. *"set_ interval"*, que cambia los valores que se almacenaron y los relacionados con Δx y Δy. Este método se hizo con el fin de hacer refinamientos del mallado.
3. *"begin_ assembly"*, únicamente ensambla con los valores correspondientes la matriz como un objeto CSR o PCSR según el caso. Este método no llama ninguna función para preparar las estructura de datos que se necesitan en algunos de los productos de la clase PCSR.
4. *"sol_ FTFS2D"*, realiza la iteración $u^{i+1} = Au^i + b$ para i desde 0 hasta el valor de n que cumple $t = n\Delta t$. En las pruebas, este método se modificó al cambiar únicamente la función que realiza el producto de la matriz esparcida A con el vector u^i, con el fin de observar cuanto tiempo toma en lograrse la solución del mismo problema al usarse los diferentes productos.

   El tiempo que se mide, es el "CPU - time", que tiene en cuenta el tiempo que toman todas las funciones y operaciones realizadas desde la primera linea luego del llamado de la función *"sol_ FTFS2D"*, hasta finalizar la función. Como la matriz A no adelantó estructura de datos para los productos, entonces todas las funciones para completar estructura de datos que necesitan algunos de los productos, se realizarán en esta parte y son contadas en el tiempo que se mide.
5. *"error"*, es un método que calcula el error entre la solución exacta 6.2 y la aproximación de la solución, con el uso de la norma $\|\cdot\|_h$ [**?**].

Para la prueba se aproximó la solución de la ecuación

$$u_t - 0{,}2u_x - 0{,}3u_y = 0, \tag{6.4}$$

con condición inicial $u_0(x, y) = \sin(xy)$ en la región $[0, 1] \times [0, 1]$. Como esta ecuación cumple la condición de estabilidad, al aproximarla con el esquema en 6.3, se tendrá convergencia de orden 1.

Tanto en las pruebas realizadas en el cluster de matemáticas, como en el cluster de ingeniería; el error y el orden de convergencia para la implementación serial y paralela, fueron los mismos y se tienen en la tabla 6–1:

h	$2{,}5e-02$	$1{,}25e-02$	$6{,}25e-03$	$3{,}12e-03$
Error	$3{,}612e-04$	$1{,}838e-04$	$9{,}277e-05$	$4{,}661e-05$
Orden		$9{,}745e-01$	$9{,}866e-01$	$9{,}930e-01$

Tabla 6–1: Error y orden de convergencia para 6.4

Se observa en la tabla 6–1 que el error disminuye al refinar la malla y se comprueba que el orden de convergencia del esquema es 1.

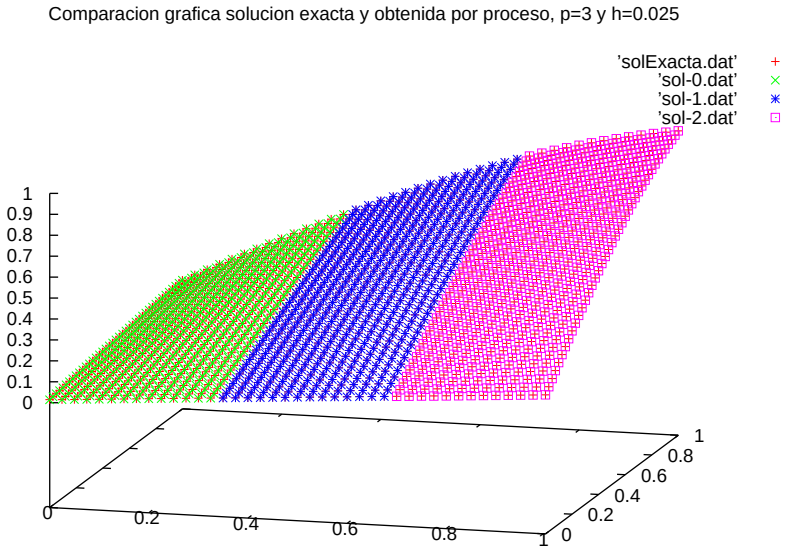

Figura 6–3: Solución exacta y por proceso de la ecuación 6.4

En la figura 6–3, tenemos la solución obtenida por cada proceso y la gráfica de la solución exacta para la ecuación 6.4. Este gráfica nos muestra la continuidad en las soluciones obtenidas en cada proceso, aunque cada solución se realizó independientemente.

6.1.1. Pruebas cluster de matemáticas

Los tiempos de ejecución del método *"sol_FTFS"* más representativos con diferente número de procesos, de cantidad de incógnitas, y de los productos implementados en PCSR, para el cluster de matemáticas se tienen en la tabla 6–2.

El nombre de los productos se redujo a iniciales *"mat_vec=mv"*, *"mat_vec_win=mvw"*, *"mat_vec_nonzero_cols_win=mvncw"*, *"mat_vec_winall=mvwa"*, *"mat_vec_schedule=mvs"*, *"mat_vec_schedule_recv_calc=mvsrc"* y *"mat_vec_schedule_all=mvsa"*.

h	Incógnitas	Tiempo serial(sg)	P	Tiempo(sg) con los productos						
				mv	mvw	mvncw	mvwa	mvs	mvsrc	mvsa
3.12 e-03	1.024 e05	5.54	2	10.54	7.48	8.7	5.16	3.22	3.23	6.11
			4	16.94	5.06	5.92	6.71	1.64	1.59	4.51
			8	24.16	4.65	5.4	12.21	0.8	0.83	2.48
1.56 e-03	4.096 e05	44.45	2	79.69	49.75	59.75	32.22	24.84	24.82	46.29
			4	172.66	28.56	35.81	25.14	12.49	12.83	34.11
			8	191.62	18.53	25.58	31.41	6.37	6.38	17.58

Tabla 6–2: Tiempo de ejecución en el cluster de Matemáticas para *"sol_FTFS"*

En las figuras 6–4, 6–5 y 6–6, se tiene el tiempo de ejecución en la solución de la ecuación de transporte en 2D con la función *"sol_FTFS"*, según la cantidad de procesos, el número de

incógnitas y los productos usados.

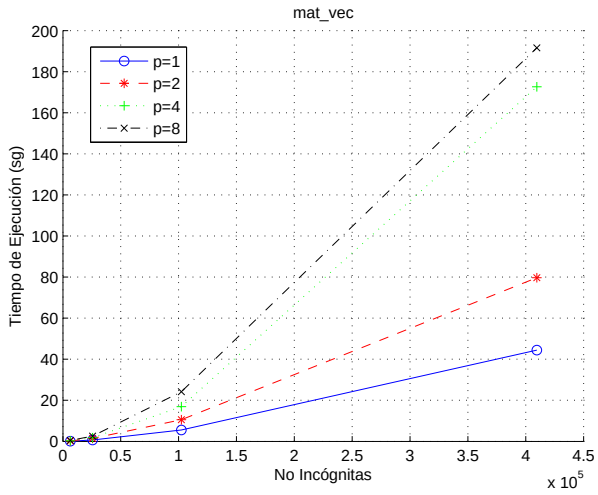

Figura 6–4: Tiempo de cómputo por proceso con "mat_vec"

Podemos observar en la figura 6–4, que al usarse el producto *"mat_vec"* para el cual la estrategia de comunicación es el *"broadcast"*, el tiempo de ejecución aumenta con el número de procesadores en vez de disminuir. Esta misma experiencia, se tiene al usarse los productos *"mat_vec_nonzero_cols"* y *"mat_vec_nonzero_rows"*.

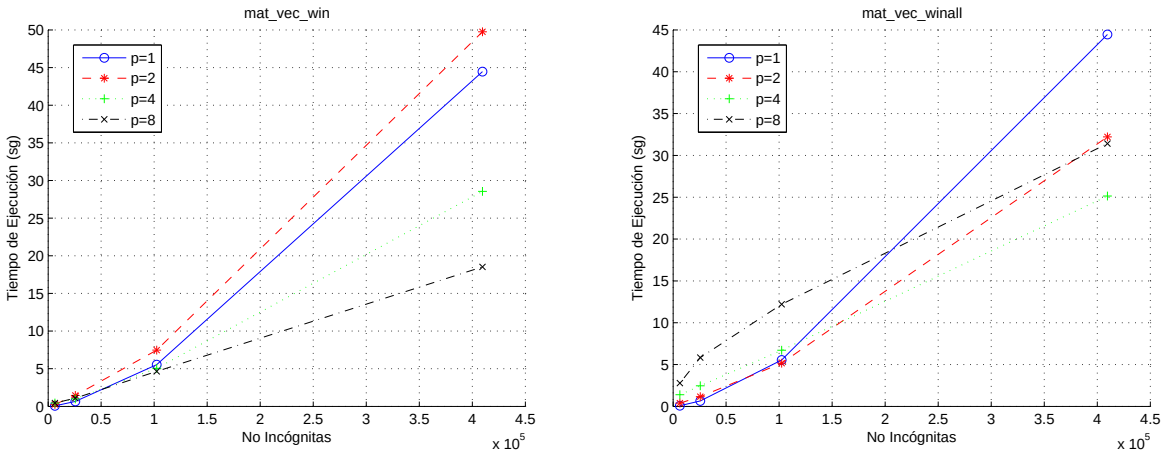

Figura 6–5: Tiempo de cómputo por proceso con mat_vec_win y mat_vec_winall

En la figura 6–5, se tiene el tiempo de ejecución con dos productos que usan para la comunicación entre procesos *"windows"*. Con *"mat_vec_win"*, se logró que al aumentar la cantidad de procesos en más de dos procesos, el tiempo de ejecución disminuya proporcionalmente con

el número de procesos, y de la tabla 6–2 se observa que este producto mejora el tiempo de ejecución para matrices de dimensiones enormes; pero con *"mat_vec_win_all"*, no es muy estable la reducción del tiempo con el número de procesos, se tiene dependencia de la dimensión de la matriz (es decir, el número de incógnitas) y de la forma en que las *"ventanas"* realicen la comunicación remota.

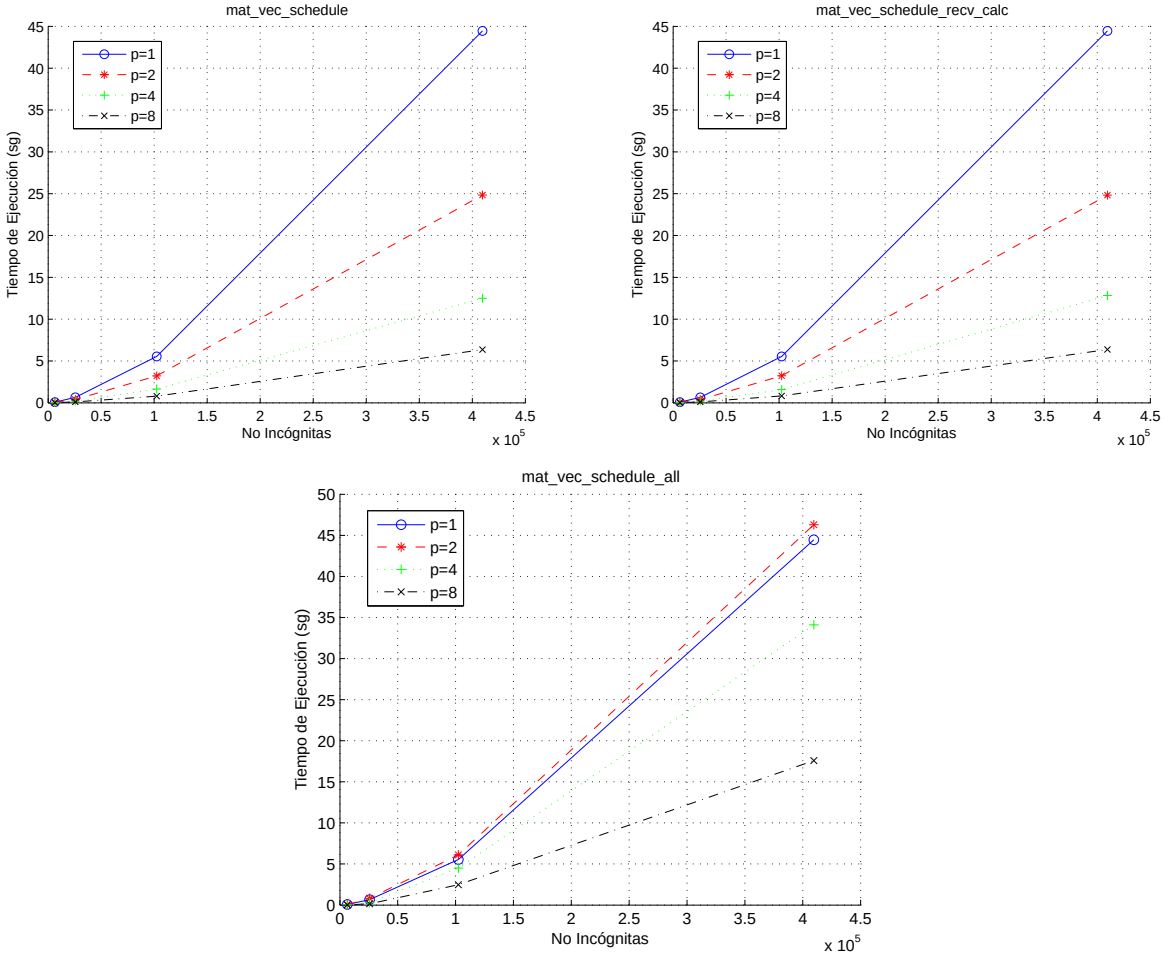

Figura 6–6: Tiempo de cómputo por proceso con los productos que usan *"schedule"*

El tiempo de ejecución obtenido con los productos que usan para la comunicación *"schedule"*, son los menores de todos los otros productos. Se observa claramente en la figura 6–6, que el tiempo de ejecución disminuye con el número de procesos cuando se envía las porciones de vector según el mapa de columnas no nulas de cada bloque de matriz, al enviarse el vector completo disminuye a partir de más de dos procesos. *"mat_vec_schedule"* y

"mat_vec_schedule_recv_calc", tardan casi la misma cantidad de tiempo de cómputo y es muy inferior al que toma *"mat_vec_schedule_all"*.

Se compara cada uno de los productos implementados para una cantidad de incógnitas, con el "speedup"de cada uno de ellos. El $speedup = \frac{tiempo\_serial}{tiempo\_paralelo}$ y como se espera que al duplicar el número de procesos el tiempo de ejecución se reduzca a la mitad, lo ideal es que el speedup sea lo más cercano posible a la función identidad. Si el speedup para un proceso es menor que 1, esto indica que para ese número proceso la función usada toma mayor cantidad de tiempo que la función serial; pero si es mayor que 1 quiere decir que tarda menos tiempo que la función serial. En las figuras 6–7 y 6–8 se tiene el speedup para cada uno de los productos para = 1,024e05 y = 4,096e05 incógnitas.

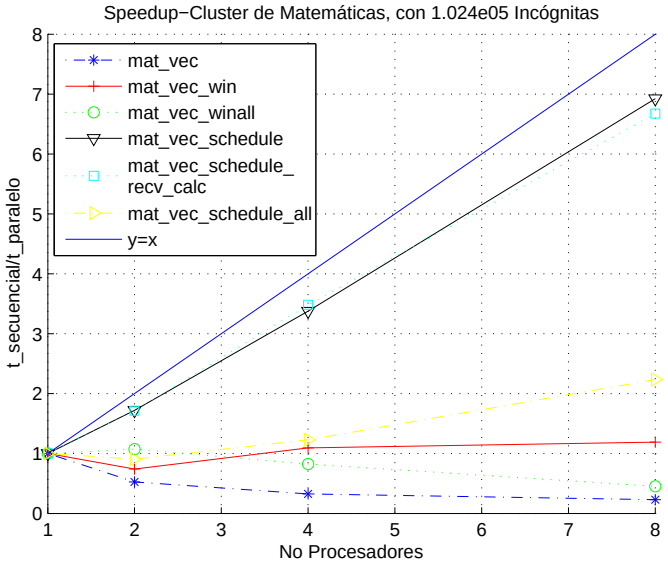

Figura 6–7: Speedup con No Incógnitas= 1,024e05

Por las figuras 6–7 y 6–8 del speedup, se puede ver que algunos productos mejoran el tiempo de ejecución con la cantidad de elementos de la matriz y de procesadores. También se observa, lo ya mencionado sobre el producto *"mat_vec"* con respecto al aumento del tiempo de ejecución con el número de procesos.

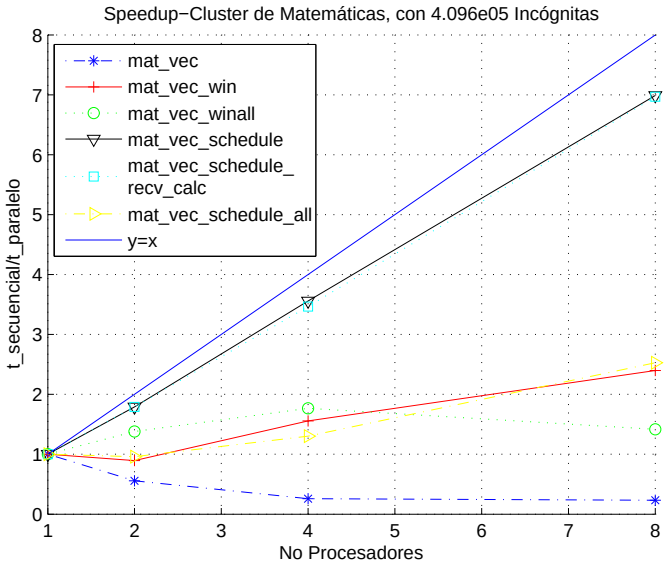

Figura 6–8: Speedup con No Incógnitas = 4,096e05

El porcentaje en que disminuyó o aumentó el tiempo de cómputo con una cantidad p de procesadores t_p, respecto al tiempo serial t_s está dado por la relación $\frac{t_s - t_p}{t_s} 100\,\%$. Si el valor obtenido es positivo, indica que t_p es menor en un $\frac{t_s - t_p}{t_s} * 100\,\%$ que t_s; pero si es negativo, entonces t_p es mayor. En la tabla 6–3, tenemos para algunos productos este porcentaje.

Incógnitas	P	$\frac{t_s - t_p}{t_s} * 100\,\%$				
		mvw	mvwa	mvs	mvsrc	mvsa
1.024 e05	2	-35.0	6.9	41.9	41.7	-10.3
	4	8.7	-21.1	70.4	71.3	18.6
	8	16.1	-120	85.6	85.0	55.2
4.096 e05	2	-11.9	27.5	44.1	44.2	-4.1
	4	35.7	43.4	71.9	71.1	23.3
	8	58.3	29.33	85.7	85.6	60.4

Tabla 6–3: Porcentaje de aumento o disminución del tiempo de ejecución con p procesos vs el serial

6.1.2. Pruebas cluster de Ingeniería

Los tiempos de ejecución del método *"sol_FTFS"* más representativos con diferente número de procesos, de cantidad de incógnitas, y de los productos implementados en PCSR, para el cluster de Ingeniería se tienen en la tabla 6–4.

h	Incógnitas	Tiempo serial(sg)	P	Tiempo(sg) con los productos						
				mv	mvw	mvncw	mvwa	mvs	mvsrc	mvsa
3.12 e-03	1.024 e05	26.37	2		56.8	62.89	29.91	14.15	14.82	44.37
			4	171.0	50.51	54.71	52.31	7.41	7.41	38.63
			8		37.54	41.33	97.74	3.85	3.85	21.34
			16		30.36	33.54	344.2	2.69	2.58	4.04
			32	243.69	30.13	33.33	765.9	2.76	2.78	2.94
1.56 e-03	4.096 e05	230.76	2		389.4	431.1	185.1	134.39	134.93	364.2
			4	229.82	253.7	288.4	174.8	67.41	68.55	300.68
			8		161.0	187.8	230.3	29.61	29.86	148.7
			16		125.76	153.9	707.9	15.5	15.42	77.51
			32	1465.6	101.9	125.7	1541.7	9.38	9.56	43.37

Tabla 6–4: Tiempo de ejecución en el cluster de Ingeniería para *"sol_FTFS"*

En esta tabla nuevamente se observa que los menores tiempos de ejecución, se presentan con los productos de schedule *"mat_vec_schedule"* y *"mat_vec_schedule_recv_cal"*; los tiempos de estos dos productos nuevamente se comportan muy semejantes. El producto *"mat_vec_schedule_all"*, que también usa schedule para la comunicación pero envía los vectores completos, tarda mayor tiempo de ejecución que los otros dos, pero se observa que al aumentar el número de procesos disminuye el tiempo de ejecución, que aún es mejor que el

tiempo de ejecución que toman los otros productos con estrategias de comunicación diferentes.

De los valores en la tabla, también se tiene que el tiempo de ejecución con el producto *"mat_vec"*, aumenta en gran cantidad con el número de procesos y se observa para 4 y 32 procesadores. El producto *"mat_vec_nonzero_cols_win"*, nuevamente toma mas tiempo de ejecución que *"mat_vec_win"*.

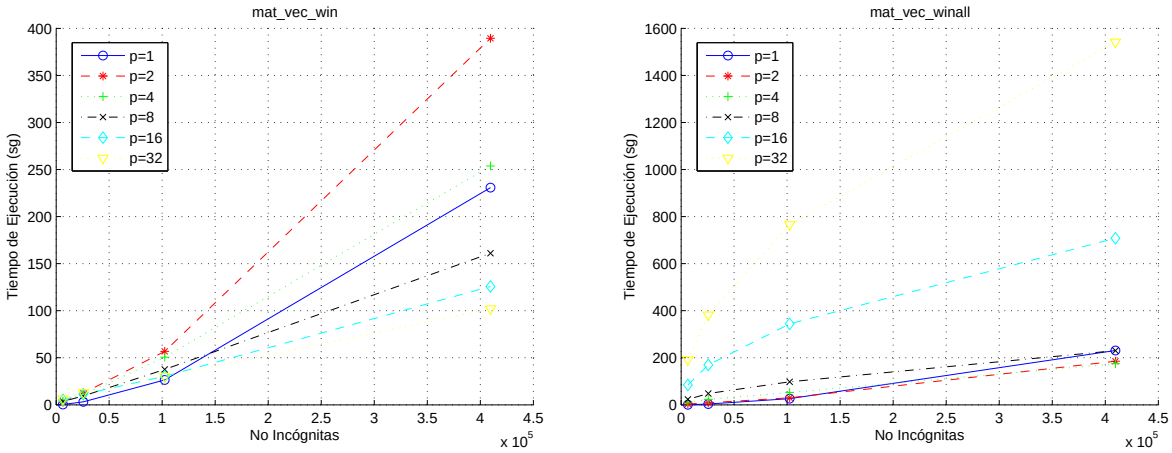

Figura 6–9: Tiempo de cómputo por proceso con "mat_vec_win" y "mat_vec_winall"

Los productos *"mat_vec_win"* y *"mat_vec_winall"* que se observan en la figura 6–9, tienen dos comportamientos diferentes para el tiempo de ejecución. *"mat_vec_win"*, únicamente disminuye el tiempo de ejecución con un número de procesos mayor a 4, si la cantidad de elementos de la matriz es grande, de lo contrario toma tiempos mayores al serial. *"mat_vec_winall"*, aumenta en gran medida el tiempo de ejecución con la cantidad de procesadores, y si se tienen menos de 8 procesadores y una gran cantidad de elementos, el tiempo de ejecución en paralelo es un poco menor al serial.

En la figura 6–10, aunque los valores graficados son de los tiempos tomados para *"mat_vec_schedule"*, también son representativos para *"mat_vec_schedule_recv_calc"* por tomar tiempos muy similares. Si el cantidad de elementos de la matriz es grande, entonces el tiempo de ejecución disminuye con el aumento del número de procesos. También se observa que entre

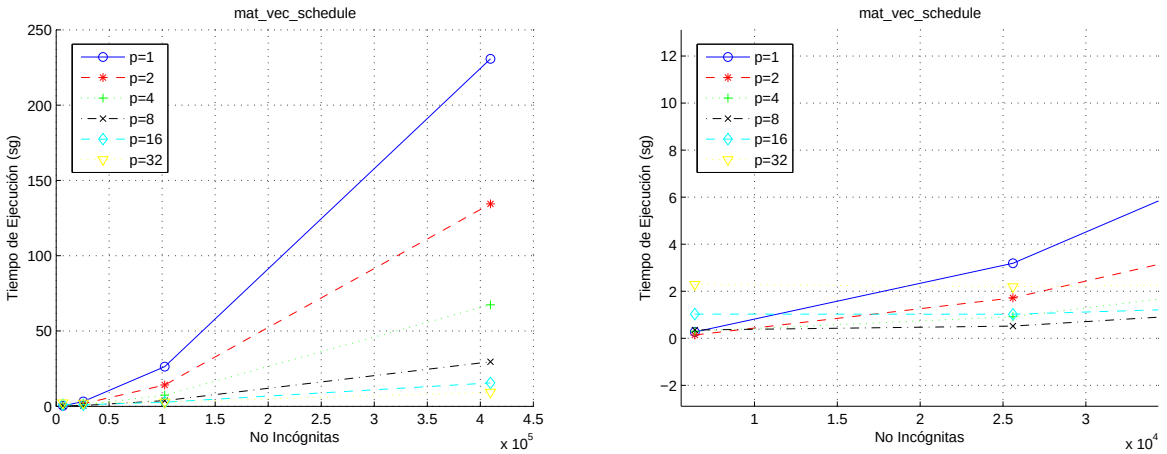

Figura 6–10: Tiempo de cómputo por proceso con “mat_vec_schedule”

mayor es la dimensión de la matriz y al usarse más procesadores, la diferencia entre el tiempo de ejecución serial y paralela es muy grande.

El “speedup", nuevamente se observó para las dos matrices de dimensiones mayores, que son las que mayor tiempo serial toman. Los valores de speedup que más se acercan a la recta $y = x$ en la figura 6–11, son los que tienen los productos con schedule, lo cual se esperaba al verse los datos de la tabla 6–4.

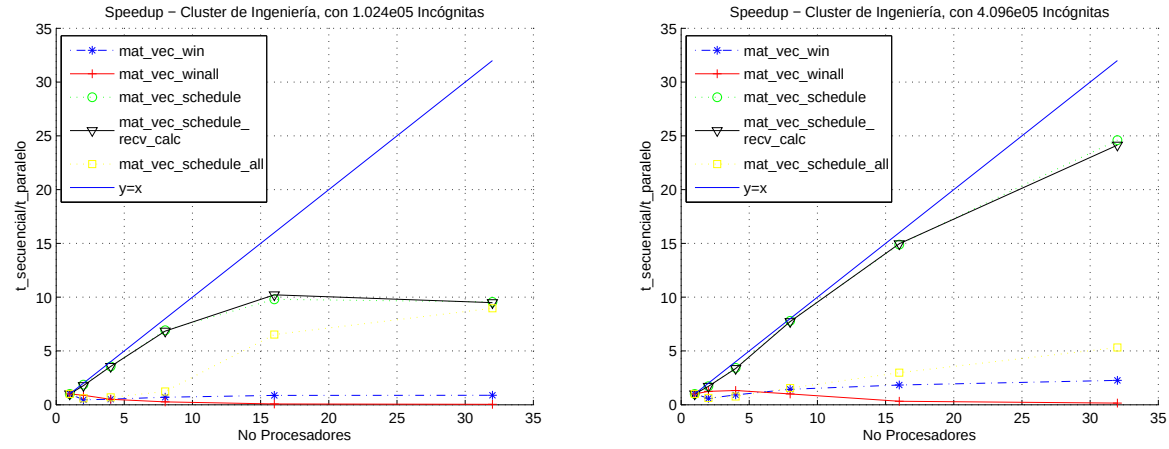

Figura 6–11: Speedup para 1,024e05 y 4,096e05 incógnitas

El speedup de *“mat_vec_schedule"* y *“mat_vec_schedule_recv_calc"* para 1,024e05 incógnitas, indica que para más de 16 procesadores el tiempo de ejecución no será menor que el

obtenido con 16 procesos. Para 4,096e05 incógnitas, aún con 32 procesos el tiempo de ejecución es menor que el obtenido con una mayor cantidad de procesadores, pero ya se aleja un poco de la recta $y = x$.

El porcentaje en que disminuye o aumenta el tiempo de ejecución con respecto al número de procesos y comparado con el tiempo de ejecución serial, se tiene en la tabla 6–5. En ella observamos como para la mayoría de productos, al aumentar el número de procesos va aumentando el porcentaje en que mejora el tiempo de ejecución paralela con respecto al tiempo serial.

Incógnitas	P	$\frac{t_s - t_p}{t_s} * 100\,\%$				
		mvw	mvwa	mvs	mvsrc	mvsa
1.024 e05	2	-115	-13.4	46.3	43.8	-68.3
	4	-90.1	-98.4	71.9	71.9	-46.5
	8	-42.4	-270	85.6	85.4	19.1
	16	-15.1	-1205	89.8	90.2	84.7
	32	-14.3	-2804	89.5	89.5	88.9
4.096 e05	2	-68.7	19.7	41.8	41.6	-57.8
	4	-9.95	24.2	70.8	70.3	-30.3
	8	30.21	0.18	87.2	87.1	35.6
	16	45.50	-206	93.3	93.3	66.4
	32	55.83	-568	95.9	95.9	81.2

Tabla 6–5: Porcentaje de aumento o disminución del tiempo de ejecución con p procesos vs el serial

Para el caso en que se usan los productos *"mat_vec_schedule"* y *"mat_vec_schedule_recv_calc"*, se logró una mejora en el tiempo de hasta un 90 % con 1,024e05 incógnitas y de un 95 % con

4,096e05 incógnitas. Se observa también lo mencionado con el speedup, sobre que el tiempo de ejecución no mejoraría para más de 16 procesadores con 1,024e05 incógnitas, ya que el porcentaje obtenido es menor en 32 que en 16.

Al comparar para la misma cantidad de procesos las tablas 6–4 y 6–5, observamos que aunque los valores de tiempo son realizados en diferentes cluster y tienen diferentes valores, para los productos *"mat_vec_schedule"* y *"mat_vec_schedule_recv_calc"*, el porcentaje obtenido en la mejora del tiempo de ejecución es casi el mismo.

6.2. Solución de la ecuación de calor 2D

La ecuación de calor, es una ecuación diferencial parabólica y está dada por la ecuación

$$u_t = au_{xx} + bu_{yy} \tag{6.5}$$

donde a y b son números positivos, y x y y representan las variables del espacio. La función $u(t, x, y)$ da la temperatura en un tiempo t y en una localización (x, y). Se desea determinar la solución $u(t, x, y)$ para un t positivo, dada la condición inicial $u(0, x, y) = u_0(x, y)$ para una función u_0.

La solución analítica para la ecuación 6.5 con condición inicial $u_0(x, y)$, en $[x_0, x_f] \times [y_0, y_f]$ está dada por

$$u(x, y, t) = \sum_{n,m=1}^{\infty} B_{n,m} \sin \frac{n\pi x}{lx} \sin \frac{m\pi y}{ly} e^{-((\frac{n\pi}{lx})^2 + (\frac{m\pi}{ly})^2)t}, \tag{6.6}$$

donde $lx = x_f - x_0$, $ly = y_f - y_0$ y $B_{n,m} = \dfrac{4}{lxly} \int_0^{lx} \int_0^{ly} u_0(x, y) \sin \dfrac{n\pi x}{lx} \sin \dfrac{m\pi y}{ly} dxdy$.

Aproximamos la solución por medio de diferencias finitas, adelantadas en tiempo y centradas en espacio ("forward in time - central space", FTCS). La discretización de la ecuación 6.5 es:

$$\frac{u_{l,m}^{n+1} - u_{l,m}^{n}}{\Delta t} = a\frac{u_{l+1,m}^{n} - 2u_{l,m}^{n} + u_{l-1,m}^{n}}{(\Delta x)^2} + b\frac{u_{l,m+1}^{n} - 2u_{l,m}^{n} + u_{l,m-1}^{n}}{(\Delta y)^2}$$

es decir,

$$u_{l,m}^{n+1} = u_{l,m}^{n} + a\frac{\Delta t}{(\Delta x)^2}(u_{l+1,m}^{n} - 2u_{l,m}^{n} + u_{l-1,m}^{n}) + b\frac{\Delta t}{(\Delta y)^2}(u_{l,m+1}^{n} - 2u_{l,m}^{n} + u_{l,m-1}^{n})$$

Manteniendo la notación del libro [**?**], sean $\mu_x = \dfrac{\Delta t}{(\Delta x)^2}$ y $\mu_y = \dfrac{\Delta t}{(\Delta y)^2}$, entonces el esquema es

$$u_{l,m}^{n+1} = a\mu_x u_{l-1,m} + b\mu_y u_{l,m-1} + (1 - 2(a\mu_x + b\mu_y))\, u_{l,m}^{n} + b\mu_y u_{l,m+1}^{n} + a\mu_x u_{l+1,m}^{n} \qquad (6.7)$$

El cual es consistente y para ser convergente se debe cumplir la condición de estabilidad, la cual si $\mu_x = \mu_y = \mu$ está dada por $(a+b)\mu \leq \dfrac{1}{2}$. El orden de convergencia de el esquema es 2

Como $\Delta t = \mu * (\Delta x)^2$ y para lograr que la aproximación tenga un error pequeño $\Delta x \to 0$, luego Δt también tiende a cero. Por tal motivo en este caso para lograr la solución del problema se deben realizar un gran número de iteraciones.

Para realizar las pruebas nuevamente se expresa el esquema en la forma matricial $u^{n+1} = Au^n + b$, donde u^n es la solución de $u(n\Delta t, x, y)$, A es una matriz esparcida de cinco bandas y b es un vector que depende de la condición de frontera.

Para la distribución en paralelo se hizo también, al repartir en forma uniforme la matriz U cuyos elementos son u_{lm}, teniendo en cuenta las condiciones de frontera que debe cumplir el esquema 6.7. La distribución para tres procesadores, se observa en la figura 6–12.

$$
\mathrm{U}=\left(\begin{array}{ccccc}
u_{00} & u_{01} & \dots & u_{09} & u_{0,10} \\
 & \rightarrow & \rightarrow & \rightarrow & \\
u_{10} & u_{11} & \dots & u_{19} & \downarrow\ u_{1,10} \\
\vdots & & \dots & \vdots & \vdots \\
u_{30} & \downarrow\ u_{31} & \dots & u_{39} & \downarrow\ u_{3,10} \\
\hline
u_{40} & \downarrow\ u_{41} & \dots & u_{49} & \downarrow\ u_{4,10} \\
\vdots & & \dots & \vdots & \vdots \\
u_{70} & \downarrow\ u_{71} & \dots & u_{79} & \downarrow\ u_{7,10} \\
\hline
u_{80} & \downarrow\ u_{81} & \dots & u_{89} & \downarrow\ u_{8,10} \\
u_{90} & \downarrow\ u_{91} & \dots & u_{99} & \downarrow\ u_{9,10} \\
 & \rightarrow & \rightarrow & \rightarrow & \\
u_{10,0} & u_{10,1} & \dots & u_{10,9} & u_{10,10}
\end{array}\right)
\begin{array}{l}
\\
\\
\Rightarrow P_0 \\
\\
\\
\Rightarrow P_1 \\
\\
\\
\Rightarrow P_2 \\
\\
\end{array}
$$

Figura 6–12: Distribución para la solucionar la ecuación de calor con 3 procesos

Con el bloque de U que le corresponden a cada proceso, se forma la porción de la matriz A y del vector b correspondientes. Para el caso de tres procesos, se tiene en la figura 6–13.

$$
\begin{array}{l}
P_0 \Rightarrow \\
\\
P_1 \Rightarrow \\
\\
P_2 \Rightarrow
\end{array}
\left(\begin{array}{ccc|ccc|ccc}
\cdot & \cdot & & & \cdot & & & & \\
\cdot & \cdot & \cdot & & & \cdot & & & \\
 & \cdot & \cdot & \cdot & & & \cdot & & \\
 & & \cdot & \cdot & & & & \cdot & \\
\hline
 & \cdot & & & \cdot & \cdot & & \cdot & \\
 & & \cdot & & \cdot & \cdot & \cdot & & \cdot \\
 & & & \cdot & & \cdot & \cdot & & \\
\hline
 & & & \cdot & & & & \cdot & \cdot \\
 & & & & \cdot & & & \cdot & \cdot \\
 & & & & & \cdot & & & \cdot
\end{array}\right)
\cdot
\left(\begin{array}{c}
u_{00}^n \\ \vdots \\ u_{39}^n \\ \hline u_{40}^n \\ \vdots \\ u_{79}^n \\ \hline u_{80}^n \\ \vdots \\ u_{99}^n
\end{array}\right)
+
\left(\begin{array}{c}
b_0 \\ \vdots \\ b_{39} \\ \hline b_{40} \\ \vdots \\ b_{79} \\ \hline b_{80} \\ \vdots \\ b_{99}
\end{array}\right)
$$

Figura 6–13: Estructura por proceso de la matriz A y el vector b del esquema matricial

La clase implementada para solucionar la ecuación 6.5, tiene la misma estructura que la realizada para solucionar la ecuación de transporte de onda. Constructor, *"set_interval"*, *"begin_assembly"*, *"sol_FTCS2D"* y *"error"*.

Inicialmente las pruebas se realizaron para solucionar la ecuación

$$u_t = u_{xx} + u_{yy}, \tag{6.8}$$

para $t = 0{,}1$ en la región $[0,1] \times [0,1]$, con $\mu = 0{,}25$ y $u_0(x,y) = sin(xy)$. Con las cuales se logró verificar el orden de convergencia, pero como al refinar la malla las iteraciones para llegar a t se duplicaban, tomaban demasiado tiempo hasta para matrices de dimensiones no muy grandes y la solución serial y algunas en paralelo, no se lograban porque se saturaba la memoria. Los resultados de error y orden de convergencia se tienen en la tabla 6–6 y en la figura 6–14 se tiene la solución por cada proceso con $p = 3$.

h	$1{,}25e-02$	$6{,}25e-03$	$3{,}12e-03$	$1{,}56e-03$
Error	$3{,}52e-05$	$8{,}81e-06$	$2{,}20e-06$	$5{,}51e-07$
Orden		2,00017	2,00004	2,00001

Tabla 6–6: Error y orden de convergencia para 6.8

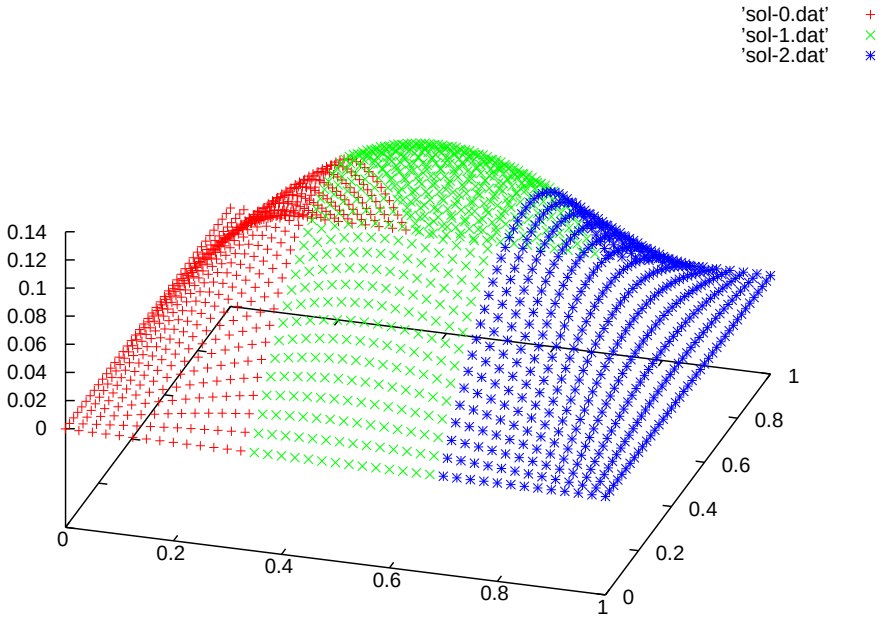

Figura 6–14: Solución por proceso para la ecuación 6.8

Algunos resultados de tiempos de ejecución en el cluster de Matemáticas se tienen en la tabla 6–7, los espacios en blanco es porque no se logró obtener el tiempo de ejecución ya que era demasiado alto. Se observa que a medida que crece la dimensión de la matriz, el producto *"mat_vec_win"* disminuye el tiempo de ejecución con el número de procesos, aunque no logra tomar en este caso un tiempo menor al serial. Al usarse el producto *"mat_vec_schedule"*,

se logra obtener un tiempo de ejecución menor al serial y además este tiempo disminuye al aumentar la cantidad de procesos; para la solución con *"mat_vec_schedule"*, se tiene el cálculo del speedup en la figura 6–15 para 1,02e05 y 4,10e05 incógnitas.

h	Incógnitas	Iteración	Tiempo serial(sg)	P	Tiempo(sg)		
					mv	mvw	mvs
3.12e-03	1.02e05	4.10e04	270.44	2	436.8	432.2	149.0
				4	953.2	441.61	99.37
				8		351.5	62.01
1.56e-03	4.10e05	1.64e05	4293.8	2			2297.3
				4			1415.0
				8		3727.8	790.6

Tabla 6–7: Tiempo de ejecución en el cluster de Matemáticas para *"sol_FTCS"*

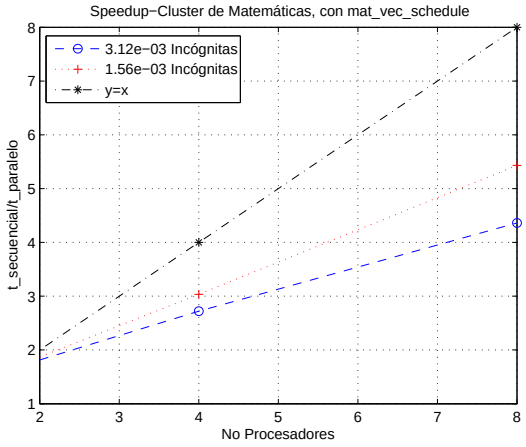

Figura 6–15: Speedup para *"sol_FTCS"* con *"mat_vec_schedule"*

Los resultados del tiempo de ejecución en la tabla 6–7, también nos muestran la dependencia de la comunicación de las ventanas a la cantidad de procesadores que se comunican entre si, ya que como la matriz A que se tiene es de 5 bandas y tiene valores en la matriz triangular inferior y en la superior; esto no sucedía para el caso de la solución de la ecuación de transporte de onda con FTFS, porque la matriz en ese caso tenía 3 bandas y todos los valores se encontraban en la matriz triangular superior.

Para lograr hacer comparaciones, se solucionará la ecuación 6.8, con la misma región y μ, pero ahora se fijaran el número de iteraciones en N, por lo que la solución será para un valor de $t = N\Delta = N\mu(\Delta x)^2$. Se usó $N = 5120$.

6.2.1. Prueba cluster de matemáticas

Fijando las iteraciones en $N = 5120$ y usando la solución de la ecuación de calor, tenemos en la tabla 6–9 el tiempo de ejecución al usar algunos de los productos. Nuevamente podemos notar, que los menores tiempos de ejecución suceden cuando se usa el producto *"mat_vec_schedule"*, seguido por *"mat_vec_schedule_recv_calc"*; para estos dos productos en este caso se tiene una pequeña diferencia en el tiempo de ejecución, siendo menor el dado por *"mat_vec_schedule"*.

Incógnitas	Tiempo serial(sg)	P	Tiempo(sg)			
			mvw	mvs	mvsrc	mvsra
2.56e04	8.48	2	20.99	5.25	5.65	15.48
		4	19.12	4.12	4.19	10.98
		8	16.14	3.3	3.93	9.02
1.02e05	33.89	2	56.55	18.81	20.6	55.49
		4	54.39	12.46	13.7	51.37
		8	42.82	7.85	9.08	31.97
4.10e05	134.71	2	184.9	72.42	79.99	214.9
		4	177.8	44.64	48.89	190.0
		8	118.5	24.82	28.59	114.1

Tabla 6–8: Tiempo de ejecución en el cluster de Matemáticas para *"sol_FTCS"* y 5120 iteraciones

El “speedup" dado por el tiempo de ejecución con *“mat_vec_schedule"*, para matrices de diferentes dimensiones se observa en la figura 6–16. Donde se concluye que entre mayor es la dimensión de la matriz, el tiempo de ejecución disminuye en mayor proporción para un número grande de procesadores.

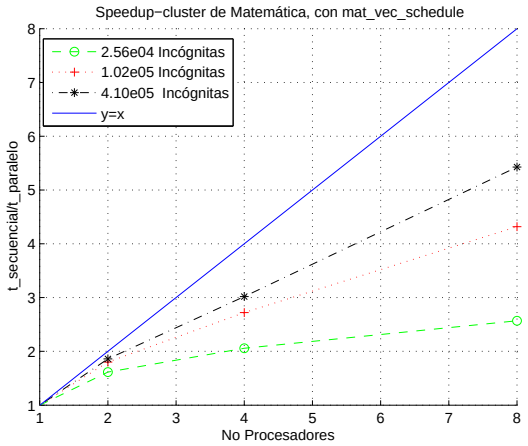

Figura 6–16: Speedup para *sol_FTCS* con *“mat_vec_schedule"* y 5020 iteraciones

6.2.2. Prueba cluster de Ingeniería

Los valores del tiempo de ejecución tomados en el cluster de Ingeniería, para la solución de la ecuación de calor con el número de iteraciones fijo en $N = 5120$, se tienen en la tabla 6–9. Los valores de los tiempos, obtenidos al usar el producto *“mat_vec_schedule"* y *“mat_vec_schedule_recv_calc"* son los menores.

La estrategia que usa ventanas, definitivamente se descarta porque sólo parece disminuir el tiempo de ejecución, para matrices de dimensiones muy grandes y usando muchos procesadores. El uso del producto *“mat_vec_winall"*, ya se había descartado porque para lograrse este se hace repetidamente el uso de ventanas y se observó que al aumentar los procesos aumenta también el tiempo de ejecución.

Incógnitas	Tiempo serial(sg)	P	Tiempo(sg)			
			mvw	mvs	mvsrc	mvsa
1.02e05	100.04	2	450.99	70.73	76.76	445.3
		4	447.54	47.96	49.95	444.4
		8	337.62	31.37	43.64	292
		16	236.35	31	32.13	129.9
		32	212.5	31.08	31.23	74.54
4.10e05	495.67	2	1515.0	284.75	322.56	1715.9
		4	1492.9	170.8	185.91	1635.4
		8	870.7	108.1	122.1	1015.7
		16	619.2	69.12	78.32	536.73
		32	441.66	51.24	56.03	292.9

Tabla 6–9: Tiempo de ejecución en el cluster de Matemáticas para *"sol_FTCS"* y 5120 iteraciones

Al usarse el producto *"mat_vec_schedule"*, se a observado en todas las pruebas que se obtienen los menores tiempos de ejecución. En este caso se observa que si la matriz no es de dimensión grande, se deben usar varios procesadores para que el tiempo de ejecución en paralelo sea menor al serial. Pero se tiene que el uso de este producto, presenta mayor estabilidad para matrices de altas dimensiones; no solo se observa en los valores de la tabla, sino también en el speedup, ver figura 6–17. La gráfica de speedup, nos muestra como a partir de cierto valor, no tiene sentido aumentar el número de procesos, por que se presenta la misma eficiencia que con un número menor de procesadores; por ejemplo para el caso en que la matriz es de $1{,}02e05$ filas y columnas, para más de 8 procesadores el tiempo de ejecución se mantiene estable y no disminuye.

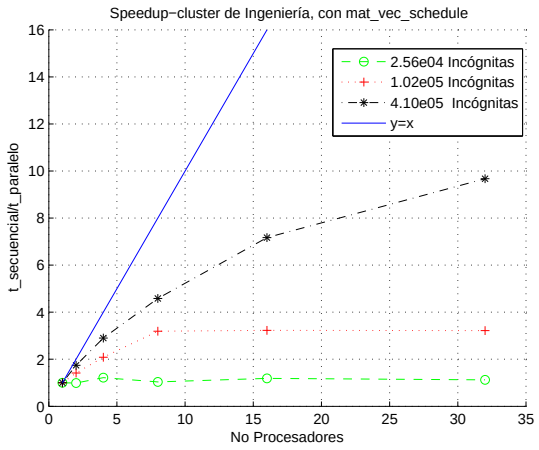

Figura 6–17: Speedup para *"sol_FTCS"* con *"mat_vec_schedule"* y 5020 iteraciones

6.3. Matrices de Matriz Market

Se seleccionaron de la página de matriz market, cuatro matrices esparcidas *bcspwr01, conf6, fidapm11 y memplus*. cierta información de estas matrices, tomada de la página de Matrix Market es la siguiente:

1. *bcspwr01 ("Power network patterns Standard IEEE test power system- New England):* Es una matriz cuadrada de 39×39, cuyos elementos de la diagonal son diferentes de cero. Sobre la diagonal tiene 46 elementos no nulos, al igual que debajo de la diagonal; en total tiene 131 elementos diferentes de cero de los $1,521$ elementos de la matriz.
2. *conf6 ("Generated by MMSTATS"):* Es una matriz cuadrada de $49,152 \times 49,152$, cuyos elementos de la diagonal son diferentes de cero. Sobre la diagonal tiene $958,464$ elementos no nulos, al igual que en la parte estrictamente inferior de la diagonal; en total la matriz tiene $1,966,080$ elementos diferentes de cero, de los $2,415,919,104$ elementos de la matriz.
3. *fidapm11 ("Generated by the FIDAP Package"):* Es una matriz cuadrada de $22,294 \times 22,294$, cuyos elementos de la diagonal son diferentes de cero. Sobre la diagonal de la matriz se tienen $300,630$ elementos no nulos, al igual que en la parte estrictamente inferior de la diagonal de la matriz; en total la matriz tiene $617,874$ elementos diferentes de cero, de los $497,022,436$ elementos de la matriz.
4. *memplus ("Computer component design memory circuit"):*Es una matriz cuadrada de $17,758 \times 17,758$, cuyos elementos de la diagonal son diferentes de cero. Sobre la diagonal de la matriz

se tienen $39,855$ elementos no nulos, al igual que en la parte estrictamente inferior de la diagonal de la matriz; en total la matriz tiene $99,147$ elementos diferentes de cero, de los $315,346,564$ elementos de la matriz.

El patrón de esparcidad para cada una de las matrices tomadas de Matriz Market se tiene en la figura 6–21, en el mismo orden que fueron mencionadas de izquierda a derecha. Las figuras también son de la página de Matriz Market.

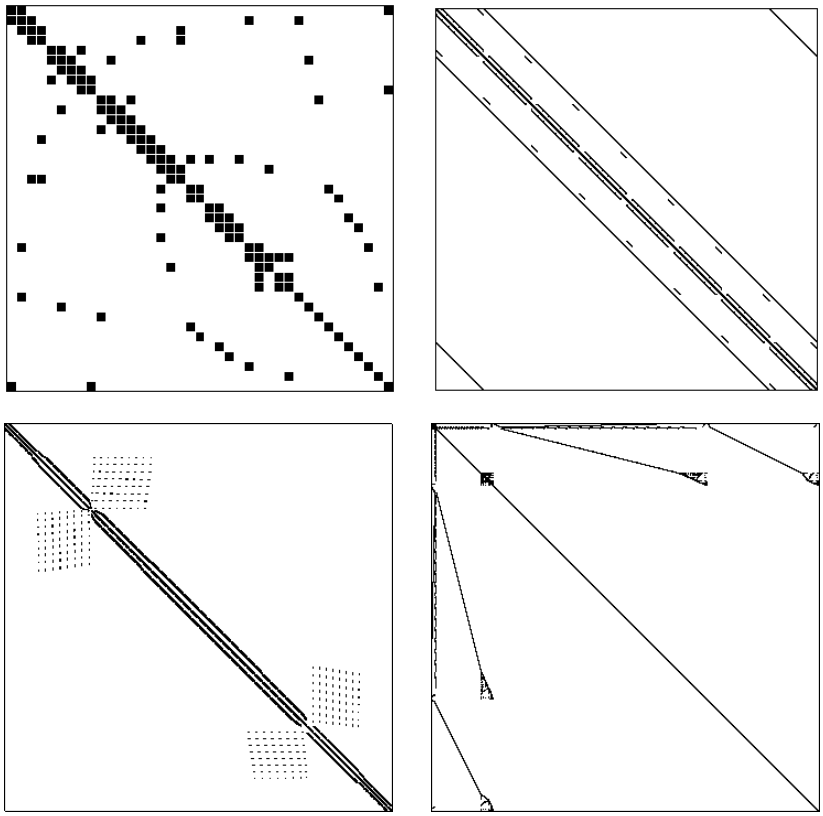

Figura 6–18: Patrón de esparcidad, ejemplares de Matriz Market

Las pruebas realizadas con estas matrices, consistió en generar y almacenar en un archivo un vector de elementos aleatorios. Se realizó el producto de cada una de estas matrices distribuidas en una cantidad p de procesos, con este vector. El tiempo tomado es para 10000 veces el producto y en el ensamble de la matriz, no se realizó ningún llamado a las funciones que crean estructura de datos para ser usada en algunos de los productos.

El tiempo de ejecución tomado, es el CPU-time. Se inicia el conteo del tiempo a partir del ciclo for para el producto del objeto PCSR de matriz esparcida con el objeto PVECTOR del vector, el conteo del tiempo termina al finalizar el ciclo for de 10000 iteraciones.

Nombre	Tiempo $serial_{(sg)}$	p	Tiempo (sg) para 10000 iteraciones						
			mv	mvnc	mvnr	mvw	mvncw	mvwa	mvs
bcspwr	1.89	8	20.63	24.18	20.94	27.34	28.95	155.8	13.1
		4	12.38	14.26	12.5	25.49	25.69	72.03	9.57
		2	7.76	9.42	8.39	19.8	20.84	28.18	5.83
conf6	186.04	8	137.46	150.92	141.17	69.59	76.35	213.6	39.92
		4	157.9	168.4	156.87	140.1	148.7	146.4	65.14
		2	149.11	166.63	155.1	159.2	171.2	146.94	114.92
fidap	61.75	8	63.7	70.95	65.4	41.85	45.73	164.2	16.5
		4	53.22	59.98	54.83	50.39	54.79	92.31	22.64
		2	59.43	66.3	61.39	72.59	77.35	64.18	37.91
memplus	64.98	8	51.02	64.65	52.63	41.11	44.82	168.94	16.2
		4	36.27	45.49	37.6	43.12	47.14	90.97	19.06
		2	28.04	34.7	29.26	37.54	41.29	43.22	19.3

Tabla 6–10: Tiempo de ejecución en el cluster de Matemáticas para el producto de matriz por vector

Los tiempos de ejecución de los productos que usan “Broadcast"para la comunicación, siempre aumentan con el número de procesadores. en la figura 6–19, se observa para la matriz memplus este hecho.

Los productos que usan ventanas tienen dos variantes, en ciertos casos el tiempo de ejecución por proceso puede llegar a ser un poco menor al tiempo serial. En los productos

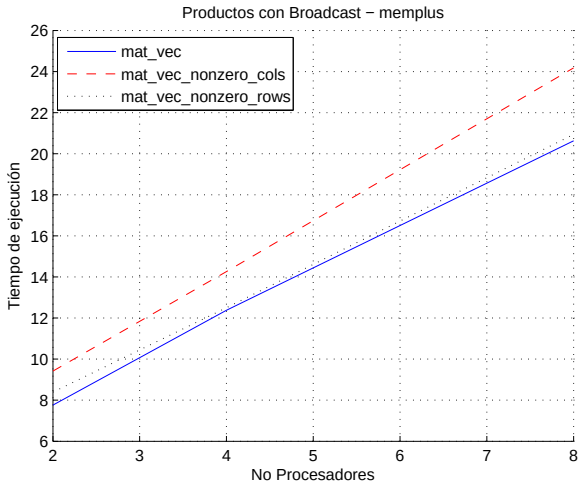

Figura 6–19: Tiempo de ejecución por proceso, para productos con broadcast

"mat_vec_win" y *"mat_vec_nonzero_cols_win"* al aumentar el número de procesos el tiempo de ejecución disminuye, pero en el producto *"mat_vec_winall"* el tiempo de ejecución aumenta con el número de procesos. Ver figura 6–20.

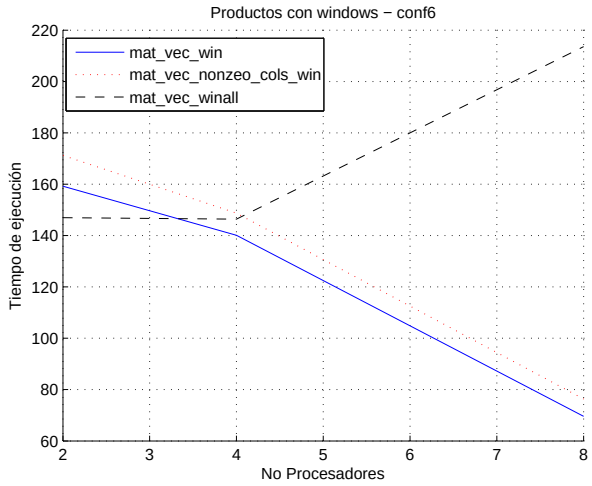

Figura 6–20: Tiempo de ejecución por proceso, para productos con windows

El producto *"mat_vec_schedule"*, tiene los menores tiempos de ejecución. Para la matriz bcspwr, es el único caso en el que el tiempo de ejecución aumenta con el número de procesadores y además es mayor al tiempo serial; esto sucede porque la dimensión de esta matriz es muy poca para ser paralelizada. En la figura 6–21, se tiene el speedup para este producto con todas las matrices.

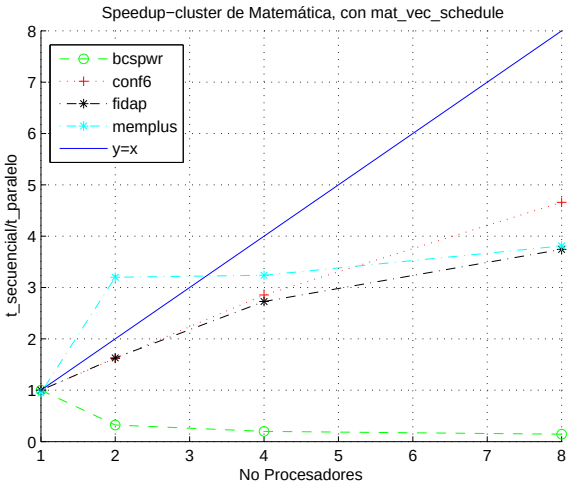

Figura 6–21: Speedup *"mat_vec_schedule"*, para cada una de las matrices de Matrix Market

Como fue mencionado, no todos los productos necesitan de una preparación anticipada de estructura de datos por parte de la matriz. Los productos:

- *"mat_vec_nonzero_cols"* y *"mat_vec_win_cols"*, solamente necesitan que de antemano se conozca el mapa de columnas diferentes de cero de cada bloque de matriz con *"get_nonzero_cols"*.
- *"mat_vec_winall"*, necesita que de antemano se tenga el mapa de columnas diferentes de cero de cada bloque de matriz con *"get_nonzero_cols"* y además que cada proceso conozca el mapa de columnas diferentes de cero de los procesos a los que debe enviar parte del vector con *"get_nonzero_cols_all"*.
- *mat_vec_schedule*, *mat_vec_schedule_recv_calc* y *mat_vec_schedule_all*, necesitan conocer el mapa de columnas diferentes de cero de cada bloque de matriz con *"get_nonzero_cols"*, también cada proceso debe tener el mapa de columnas diferentes de cero de los procesos a los que debe enviar parte del vector con *"get_nonzero_cols_all"* y además realizar el "scheduleçon *"find_schedule"*.

Todos los tiempos de ejecución presentados en las pruebas, incluyen el tiempo de ejecución tomado por estas funciones. Para las matrices de Matrix Market, el tiempo de cada una de estas funciones según el número de procesos se tiene en la siguiente tabla.

Nombre	p	Tiempo (sg)		
		get_nonzero_cols	get_nonzero_cols_all	find_schedulle
bcspwr	8	0.31	0.55	0.23
	4	0.28	0.51	0.17
	2	0.23	0.49	0.16
conf6	8	0.65	0.71	0.24
	4	1.04	1.26	0.57
	2	1.65	2.03	0.70
fidap	8	0.19	0.29	0.08
	4	0.48	0.63	0.19
	2	0.17	0.35	0.11
memplus	8	0.85	0.76	0.43
	4	0.96	0.71	0.54
	2	0.86	0.76	0.39

Tabla 6–11: Tiempo de ejecución tomado en la preparación de la matriz

6.4. PETSc

"Portable, Extensible Toolkit for Scientific Computation"PETSc, es un conjunto de estructuras de datos y rutinas paralelas para la solución de modelos científicos a gran escala, en especial para solucionar EDP. Para el paso de mensajes entre procesos, PETSc utiliza el estándar MPI. Algunas de las herramientas que incluye son: Vectores en paralelo (distribución y recolección), matrices en paralelo (formato con matrices esparcidas, con un fácil y eficiente ensamblaje), precondicionadores en paralelo, métodos en subespacios de Krylov, método de Newton en paralelo para solución de ecuaciones no lineales. Esta incluye excelente documentación y permite trabajar tanto en UNIX como en Windows.

Las pruebas con PETSc, se realizaron solamente en el cluster de matemáticas, ya que el cluster de ingeniería no cuenta con este software.

Este programa cuenta con excelente documentación e incluye una gran cantidad de funciones para el trabajo con vectores, matrices, precondicionadores, entre otros; también incluye sus propias funciones para la programación en paralelo. Luego se debe convertir al formato de petsc, todos los datos que se quieren ingresar o usar con este programa y para ello se debe conocer la definición de variables, el formato que debe tener cada objeto y como se hacen las operaciones.

PETSc prácticamente es un nuevo lenguaje compatible con C y que acepta algunas de las estructuras y definiciones de C, pero también incluye una forma de definición de los diferentes tipos de datos. Es un programa que se encuentra optimizado y es muy reconocido.

Las pruebas realizadas con PETSc, se hicieron para las matrices de Matix Market, con la misma cantidad de iteraciones para el producto y el mismo vector aleatorio. Los valores del tiempo de ejecución con PETSc, se tienen en la tabla 6–13

p	Tiempo (sg) con PETSc			
	bcspwr	conf6	fidap	memplus
8	1.8	20.91	8.87	4.59
4	0.85	28.62	15.01	4.84
2	0.84	54.73	29.73	4.84

Tabla 6–12: Tiempo de ejecución en el cluster de Matemáticas para el producto de matriz por vector 10000 veces con petsc

Al comparar los tiempos de ejecución obtenidos para esta prueba con PETsc, con los tiempos de ejecución obtenidos para la misma prueba con el producto con schedule en la tabla

6–10, observamos que los tiempos con PETSc son menores que los dados con schedule. En la siguiente tabla tenemos la cantidad de veces que supera el tiempo de ejecución del producto de schedule al de PETSc, para cada una de las matrices de Matrix Market.

p	*"mat_vec_schedule"* Vs *PETSc*			
	bcspwr	conf6	fidap	memplus
8	7.2778	1.9091	1.8602	3.5294
4	11.2588	2.2760	1.5083	3.9380
2	6.9405	2.0998	1.2751	3.9876

Tabla 6–13: Comparación del tiempo de ejecución dado con PETSc y con *"mat_vec_schedule"*

En algunos casos el número de veces que *"mat_vec_schedule"* supera a PETSc, es casi el doble y en otros muy superior, dependiendo de la estructura de la matriz y de su dimensión.

Aún así en la figura 6–22, vemos que el comportamiento en el aumento o disminución del tiempo de ejecución se comporta similarmente tanto con el producto de PETSc como con el producto con *"mat_vec_schedule"*.

No conozco la forma en que PETSc realiza la comunicación y aunque este supera el tiempo de cómputo del mejor de los productos implementados en esta tesis, no es decepcionante porque para usar las estructuras de PETSc se debe señir a los formatos que ellos tienen y se deben de conocer gran parte de sus funciones y además no es sencillo hacer modificaciones en la programación de sus rutinas o entender sus implementaciones. Además puede que en el momento de finalizar el ensamble de la matriz, PETSc la prepare según la estrategia de comunicación que usen y así este tiempo no se está incluyendo en el tiempo de ejecución medido.

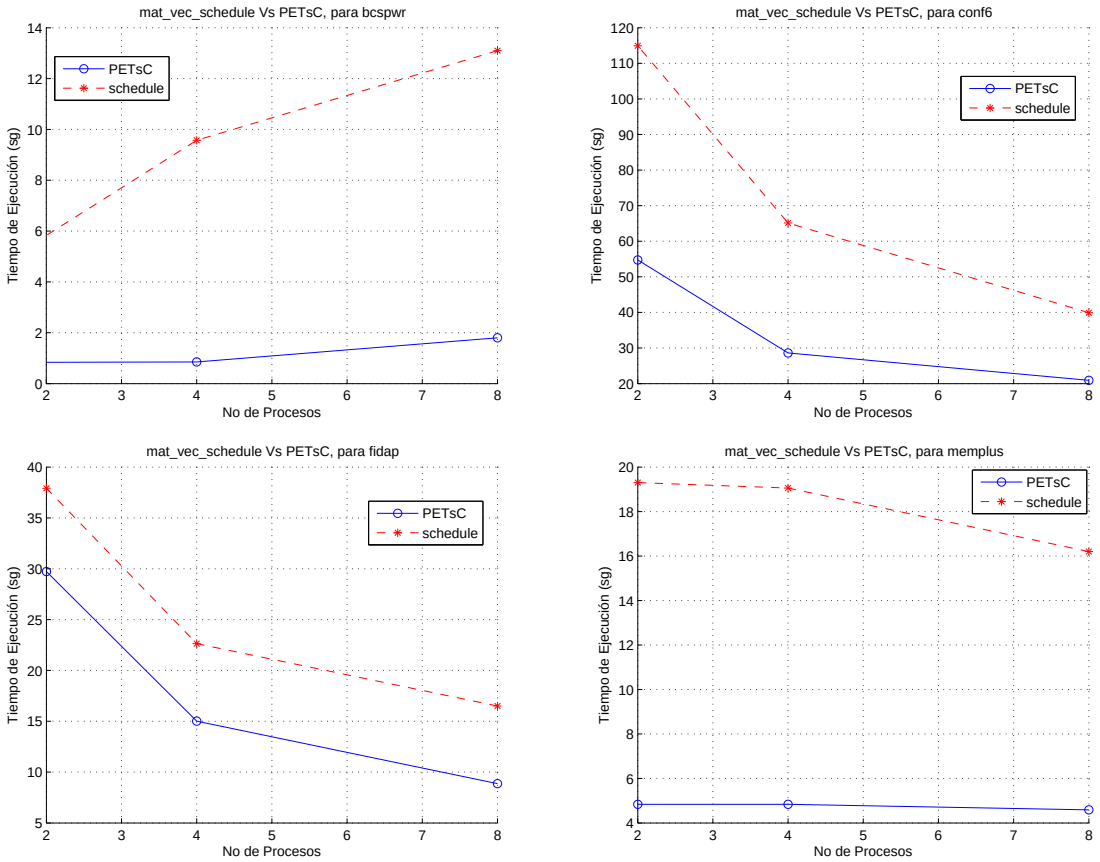

Figura 6–22: Comparación del tiempo de ejecución con PETSc y *"mat_ vec_ schedule"*

Capítulo 7
TRABAJOS FUTUROS

Usar la programación orientada a objeto y las librerías de MPI para la programación en paralela, permitieron hacer una librería accesible para usuarios que tienen experiencia en el uso de objetos en C++, sin necesidad que tengan conocimiento en las funciones de MPI.

A medida en que se fue desarrollando esta tesis, surgían otros interrogantes sobre ciertas cosas que se pueden tener en cuenta, para ayudar que la comunicación sea bien uniforme, entre estas están:

1. Desarrollar una estrategia, que distribuya en forma eficiente la matriz esparcida entre procesos, teniendo en cuenta la cantidad de entradas diferentes de cero que tendrá la matriz por proceso y sus dimensiones.
2. Hacer un template, en el cual se puedan usar diferentes formatos para almacenar matrices esparcidas y según el caso usarlos, en vez de únicamente usarse el CSR. Por ejemplo para matrices en banda, será conveniente usar el formato DIAG.
3. Controlar cuando es conveniente adquirir el vector completo o parte del vector de los procesos que se necesite.

También se podrían intentar otras estrategias de comunicación, como usar uno o dos procesadores como esclavos, almacenando en ellos únicamente el vector que será multiplicado con la matriz, en vez de distribuir una porción del vector por todos los procesos; la matriz puede ser distribuida en el resto de procesadores del comunicador, así los dos procesos esclavos

serán los encargados de enviar la información y la comunicación será uno a uno. Comparar los resultados que se obtengan para el tiempo, con los obtenidos con el schedule.

Como la implementación es en paralelo, se debe intentar realizar el Schedulle en paralelo; y encontrar una función usando interpolación y extrapolación para el tiempo de comunicación para cualquier cantidad de procesos en un comunicador y usarla para formar la matriz de envío y recepción de datos en el Schedulle.

APENDICES

Apéndice A

Manual de Referencia

A.1. Referencia de la Clase CSR

Clase **CSR**(p. 103).

`#include <csr.h>`

Métodos públicos

- **CSR** ()

  Constructor de matriz vacía.

- **CSR** (unsigned int m, unsigned int n)

  Constructor de matriz de orden `m` *$$* `n`.

- **CSR** (const **CSR** &)

  Copy constructor.

- ∼**CSR** ()

Destructor.

- void **start_assembly** ()

  Inicio de ensamble de la matriz.

- void **insert** (unsigned int row, unsigned int col, double value)

  Inserción de elementos en la matriz.

- void **add** (unsigned int row, unsigned int col, double value)

  Suma el elemento `value`*, al elemento en la posición indicada.*

- void **erase** (unsigned int row, unsigned int col)

  Elimina de la matriz el elemento en la posición indicada.

- void **end_assembly** ()

  Finaliza la Fase de Ensamblado de la Matriz.

- void **read** (const string &filename)

  Lee el archivo donde se encuentra la matriz.

- void **read** (istream &input)

  Lee datos de la matriz de un archivo abierto.

- void **write** (ostream &output)

  Escribe datos de la matriz en un archivo abierto.

- void **dump** (const string &filename) const

Genera el archivo `filename` *con la matriz.*

- void **dump_script** (const string &filename) const

  Genera el archivo `filename` *con la matriz en formato para ser leído en MATLAB.*

- void **create_parallel** (const string &filename, unsigned int p, unsigned int *part_row=0, unsigned int *part_col=0) const

  Genera `p` *archivos con la matriz distribuida en ellos (Se usa para convertir de serial a paralelo).*

- void **create_CSR** (const string &filename)

  Genera una matriz CSR, a partir de una matriz que se encontraba distribuida en archivos (Se usa para convertir de paralelo a serial).

- void **set_tolerance** (double tol_)

  Cambia la tolerancia por defecto de 1.0e-16 por `tol_`.

- double **get_tolerance** () const

  Retorna la tolerancia de la Matriz.

- void **set_dimension** (unsigned int m, unsigned int n)

  Cambia las dimensiones de la matriz y la deja vacía.

- double **operator()** (unsigned int i, unsigned int j) const

  Retorna elemento A(i,j) sobre cargando ().

- double **get_aij** (unsigned int i, unsigned int j) const

Retorna el elemento A(i,j) de haberlo, sino devuelve cero.

- unsigned int **get_num_rows** () const

  Retorna la cantidad de filas de la matriz.

- unsigned int **get_num_cols** () const

  Retorna la cantidad de columnas de la matriz.

- unsigned int **get_num_nnz** () const

  Retorna el numero de elementos diferentes de cero de la matriz.

- unsigned int **get_num_nnz** (unsigned int i) const

  Retorna el numero de elementos diferentes de cero de la fila i de la matriz.

- unsigned int * **get_nonzero_cols** (unsigned int &n, unsigned int *col_map=0) const

  Obtiene las columnas diferentes de cero de la Matriz.

- unsigned int * **get_nonzero_rows** (unsigned int &n, unsigned int *row_map=0) const

  Obtiene las filas diferentes de cero de la Matriz.

- void **mat_vec** (const double *x, double *y) const

  Multiplica la matriz por el vector x y asigna el resultado al vector y.

- void **mat_vecpy** (const double *x, double *y) const

  Multiplica la matriz por el vector x y suma el resultado al vector y.

- void **mat_vec_nonzero_cols** (const double *x, double *y, unsigned int n, unsigned int *cols, unsigned int *col_map) const

  Multiplicación de matriz por vector teniendo en cuenta los elementos en posición de las columnas nulas de la matriz.

- void **mat_vec_nonzero_colspy** (const double *x, double *y, unsigned int n, unsigned int *cols, unsigned int *col_map) const

  Multiplicación de matriz por vector teniendo en cuenta los elementos nulos de las columnas de la matriz y suma el resultado a y.

- void **mat_vec_nonzero_rows** (const double *x, double *y, unsigned int n, unsigned int *rows, unsigned int *row_map) const

  Multiplicación de matriz por vector teniendo en cuenta las filas nulas de la matriz.

- void **mat_vec_nonzero_rowspy** (const double *x, double *y, unsigned int n, unsigned int *rows) const

  Multiplicación de matriz por vector teniendo en cuenta las filas nulas de la matriz y se suma el resultado a y.

- void **mat_vec_nonzero_row_col** (const double *x, double *y, unsigned int n1, unsigned int *rows, unsigned int n2, unsigned int *cols, unsigned int *col_map) const

  Producto teniendo en cuenta filas y columnas nulas de la Matriz y los respectivos vectores de filas y columnas no nulas con los mapas, asigna el resultado a y.

- void **mat_vec_nonzero_row_colpy** (const double *x, double *y, unsigned int n1, unsigned int *rows, unsigned int n2, unsigned int *cols, unsigned int *col_map) const

Producto teniendo en cuenta filas y columnas nulas de la Matriz y los respectivos vectores de filas y columnas no nulas con los mapas, suma el resultado a y.

- **CSR** $*$ **get_transpose** () const

  Retorna una matriz **CSR**(p. 103)*, con la transpuesta de la matriz.*

- void **scalar_mat** (double alpha)

  Realiza el producto del escalar alpha con A y el resultado lo almacena en A. A=aA.

- void **plus_mat** (const **CSR** &B)

  Le asigna a la Matriz, la suma de ella con la matriz B. se mantiene la tolerancia de la Matriz que llama la función. A=A+B.

- void **supr_mat** ()

  Suprime una matriz, dejándola vacía.

A.1.1. Descripción detallada

Clase **CSR**(p. 103).

A.1.2. Documentación de las funciones miembro

A.1.2.1. void CSR::add (unsigned int *row*, unsigned int *col*, double *value*)

Adiciona el elemento `value`, al elemento en la posición indicada.

Suma el elemento `value`, al elemento que se encuentre en la posición `row`, `col` en caso de haberlo, sino lo inserta

A.1.2.2. void CSR::create_CSR (const string & *filename*)

Genera una matriz que se encontraba distribuida en archivos.

Lee el archivo filename = nombre-∗.extension, con el formato de matrices esparcidas en paralelo, iniciando la lectura con nombre-0.extension, y completando la matriz de todos los archivos

A.1.2.3. void CSR::create_parallel (const string & *filename*, unsigned int *p*, unsigned int ∗ *part_row* = 0, unsigned int ∗ *part_col* = 0) const

Genera `p` archivos con la matriz distribuida en ellos.

Se generan `p` archivos `filename` = nombre-∗.extension, donde se divide la matriz en bloques según la partición de filas (`part_row`) y columnas (`part_col`) ingresadas, si se no se da una partición por defecto se realizara una partición uniforme. Este archivo se hace siguiendo el formato esperado por **read(const string &filename)**(p. 112) de matrices esparcidas en paralelo

A.1.2.4. void CSR::dump (const string & *filename*) const

Genera el archivo `filename` con la matriz.

En el archivo filename se almacena la matriz con el formato descrito en **read(const string &filename)**(p. 112)

A.1.2.5. void CSR::dump_script (const string & *filename*) const

Genera el archivo `filename` con la matriz en formato para ser leído en MATLAB.

En el archivo filename = nombre.m, se almacena la matriz A, usando el formato sparse de MATLAB

A.1.2.6. void CSR::end_assembly ()

Finaliza la Fase de Ensamblado de la Matriz.

Inicia `assembly_flag` = end_assembly. Los elementos que se almacenaron por fila en el vector de mapas `vec_index`, los extrae y los almacena en arreglos según el test de tolerancia y en la fila donde se inserte se aumenta el numero de elementos diferentes de cero.

No se permite inserción, adición, ni borrado de elementos.

A.1.2.7. void CSR::erase (unsigned int *row*, unsigned int *col*)

Elimina el elemento en la posición indicada de la matriz.

Se borra el elemento en la fila `row`, columna `col` de la matriz en caso de haberlo.

A.1.2.8. unsigned int* CSR::get_nonzero_cols (unsigned int & *n*, unsigned int * *col_map* = 0) const

Obtiene las columnas diferentes de cero de la Matriz.

Retorna un arreglo con las columnas diferentes de cero de la matriz, además de su tamaño. Si se ingresa `col_map`, que debe ser del tamaño del numero de columnas de la matriz, en el se retorna un mapa donde si la columna es igual a cero se tiene UINT_MAX, sino la posición de dicha columna en el vector de salida para columnas no nulas

A.1.2.9. unsigned int* CSR::get_nonzero_rows (unsigned int & *n*, unsigned int * *row_map* = 0) const

Obtiene las filas diferentes de cero de la Matriz.

Retorna un arreglo con las filas diferentes de cero y un valor por referencia `n` con su tamaño. Además el mapa de filas de tal forma que si la fila es nula se tiene UINT_MAX y sino la posición de la fila en el vector que se retorna para las filas no nulas

A.1.2.10. void CSR::insert (unsigned int *row*, unsigned int *col*, double *value*)

Inserción de elementos en la matriz.

Coloca en la posición `row`, `col` el elemento `value`, en caso de haber otro elemento lo sustituye. La búsqueda es logarítmica por que se esta usando mapas, donde la llave es la columna

A.1.2.11. void CSR::mat_vec_nonzero_cols (const double * *x*, double * *y*, unsigned int *n*, unsigned int * *cols*, unsigned int * *col_map*) const

Multiplicación de matriz por vector teniendo en cuenta los elementos en posición de las columnas nulas de la matriz.

Se ingresa el vector x de dimension `n`, reducido de un vector con quien se desea realizar la multiplicación con la matriz, únicamente con los elementos que se encuentran en las columnas `cols` de tamaño `n` que retorna **get_nonzero_cols()**(p. 110) y además del mapa de posiciones. Este producto asigna la solución a y

A.1.2.12. void CSR::mat_vec_nonzero_rows (const double * *x*, double * *y*, unsigned int *n*, unsigned int * *rows*, unsigned int * *row_map*) const

Multiplicación de matriz por vector teniendo en cuenta las filas nulas de la matriz.

El vector a multiplicar se ingresa completo, es decir de tamaño `num_cols_`; además se ingresa el vector `rows` de filas no nulas, obtenido con **get_num_rows()**(p. 106) y el mapa de posiciones, Para asignar las soluciones a las filas correspondientes de `y`

A.1.2.13. void CSR::mat_vec_nonzero_rowspy (const double * *x*, double * *y*, unsigned int *n*, unsigned int * *rows*) const

Multiplicación de matriz por vector teniendo en cuenta las filas nulas de la matriz y se suma el resultado a y.

Aqui no se necesita el mapa de filas que asigna **get_num_rows()**(p. 106), solamente el vector `row` de filas nulas.

A.1.2.14. void CSR::read (istream & *input*)

Lee datos de la matriz de un archivo abierto.

Lee para todas las filas `num_rows_` el numero de entradas diferentes de cero, las columna y los valores por cada una, en un archivo que se encuentra abierto por la función que hace el llamado. Se usa en el formato de lectura para matrices esparcidas en paralelo **PCSR**(p. 114).

A.1.2.15. void CSR::read (const string & *filename*)

Lee el archivo donde se encuentra la matriz.

Según el formato esperado, se lee el archivo `filename` y se construye la matriz.

1. Dimension de filas `num_rows_`

2. Dimension de Columnas `num_cols_`

3. Numero de entrada diferente de cero por cada fila seguida por 4. columna y valor diferente de cero

A.1.2.16. void CSR::start_assembly ()

Inicio de ensamble de la matriz.

Inicia `assembly_flag` = start_assembly y esto permite el uso de funciones que modifican la matriz, insert, add, erase, supr_bloq, set_tolerance.

No es posible el uso de esta función luego de end_assembly.

A.1.2.17. void CSR::write (ostream & *output*)

Escribe datos de la matriz en un archivo abierto.

En un archivo abierto, por la función que realiza el llamado de esta instrucción, almacena por fila el numero de entradas no nulas y sus respectivas columnas y valores. Se Usa para escribir matrices en matrices esparcidas en paralelo **PCSR**(p. 114)

La documentación para esta clase fué generada a partir del siguiente archivo:

- csr.h

A.2. Referencia de la Clase PCSR

Clase **PCSR**(p. 114).

`#include <PCSR.h>`

Métodos públicos

- **PCSR** (unsigned int dim_row, unsigned int dim_col, MPI_Comm COMM_=MPI_COMM_WORLD)

  Constructor.

- **PCSR** (unsigned int *part_row, unsigned int *part_col, MPI_Comm COMM_=MPI_COMM_WORLD)

  Constructor.

- **~PCSR** ()

  Destructor.

- void **read** (const string &filename)

  Cada proceso lee el archivo donde se encuentra la matriz.

- void **write** (const string &filename) const

  Cada proceso escribe un archivo con la matriz.

- void **start_assembly** ()

  Indica que se va a comenzar a ensamblar una matriz.

- void **insert** (unsigned int Num_bloq, unsigned int row, unsigned int col, double value)

Inserta `value` *en la fila* `row` *, columna* `col` *del bloque* `Num_bloq`.

- void **insert** (unsigned int row, unsigned int col, double value)

  Inserta `value` *en la fila* `row`*, columna* `col` *de la matriz, según el bloque, proceso y posición que debe corresponderle.*

- void **add** (unsigned int Num_bloq, unsigned int row, unsigned int col, double value)

  Suma `value` *al elemento que se encuentre en la fila* `row` *, columna* `col` *del bloque* `Num_bloq` *en caso de haberlo, sino se inserta.*

- void **add** (unsigned int row, unsigned int col, double value)

  Suma `value` *al elemento que se encuentre en la fila* `row`*, columna* `col` *de la matriz, según el bloque, proceso y posición que debe corresponderle. En caso de no haber alguno lo inserta.*

- void **erase** (unsigned int Num_bloq, unsigned int row, unsigned int col)

  Elimina en caso de haber, al elemento que se encuentre en la fila `row` *, columna* `col` *del bloque* `Num_bloq`.

- void **supr_bloq** (unsigned int Num_bloq)

  Elimina todos los elementos del bloque `Num_bloq` *, dejándolo vacío.*

- void **set_tolerance** (double tol_)

  Cambia la tolerancia por defecto de 1.0e-16 de matriz **CSR**(p. 103) *por* `tol_`*, la cual se tendrá en cuenta para decidir si un elemento es considerado diferente de cero o no, en la inserción final.*

- void **end_assembly** ()

  Finaliza la fase de ensamble de la matriz.

- double **get_aij** (unsigned int Num_bloq, unsigned int row, unsigned int col) const

  Retorna el elemento en la posición `row`, `col` *del bloque Num_bloq en caso de haber* `para` *el procesador que se solicita, sino hay elemento retorna cero.*

- double **get_aij** (unsigned int row, unsigned int col) const

  Retorna el elemento en la posición `row`, `col` *según el bloque y proceso que corresponde a dicha posición; sino hay almacenado algún elemento retorna cero.*

- unsigned int **get_num_rows** () const

  Retorna la cantidad de filas de `my_rank`.

- unsigned int **get_num_cols** (unsigned int bloq) const

  Retorna la cantidad de columnas del bloque `bloq`, *de* `my_rank`.

- unsigned int **get_num_nnz** () const

  Retorna el numero de elementos diferentes de cero de la porción de matriz de `my_rank`.

- unsigned int **get_num_nnz** (unsigned int bloq) const

  Retorna el numero de elementos diferentes de cero del bloque `bloq`, *de* `my_rank`.

- void **get_nonzero_cols** ()

  Por cada bloque no nulo, genera cierta información para las columnas diferentes de cero.

- void **get_nonzero_rows** ()

  Por cada bloque no nulo, genera cierta información para las filas diferentes de cero.

- void **get_nonzero_cols_all** ()

  Por medio de ventanas se comunica la información de la cantidad de datos que el procesador `my_rank` *debe enviar a los procesos que le correspondan. Luego también por medio de ventanas se envían los arreglas de mapas correspondientes, para lograr hacer paquetes de datos.*

- void **get_nonzero_cols_sch** ()

  Genera el arreglo con el orden de tareas de todos los procesos usando inicialmente "windo" para la comunicación de la cantidad de columnas diferentes de cero a enviar y finalmente "schedule" para completar la estructura de datos con los respectivos mapas de posiciones de los elementos a comunicar.

- void **mat_vec** (const **PVector** &x, **PVector** &y) const

  Producto de la matriz con el **PVector**(p. 125) `x` *distribuido en cada proceso, formando vector global con el vector de cada proceso.*

- void **mat_vec_win** (**PVector** &x, **PVector** &y)

  Producto de la matriz con el **PVector**(p. 125) `x` *distribuido en cada proceso, tomando por medio de ventana el vector que se necesita para lograr el producto.*

- void **mat_vec_winall** (**PVector** &x, **PVector** &y)

  Producto de la matriz con el **PVector**(p. 125) `x` *distribuido en cada proceso, enviando por medio de ventanas a los procesadores que se necesitan para el producto: la cantidad de elementos y la posición donde se encuentran. Para así recibir de cada uno de dichos procesos, un paquete con los valores de cada vector que se necesitaban.*

- void **mat_vec_nonzero_cols** (const **PVector** &x, **PVector** &y)

  Producto de la matriz con el **PVector**(p. 125) `x` *distribuido en cada proceso, en el cual se*

tiene en cuenta las columnas diferentes de cero por cada bloque, en un vector global formado con el vector de cada proceso.

- void **mat_vec_nonzero_cols_win** (**PVector** &x, **PVector** &y)

  Producto de la matriz con el **PVector**(p. 125) `x` *distribuido en cada proceso, en el cual se tiene en cuenta las columnas diferentes de cero por cada bloque, de los vectores que se necesitan tomados con el uso de ventanas.*

- void **mat_vec_nonzero_rows** (const **PVector** &x, **PVector** &y)

  Producto de la matriz con el **PVector**(p. 125) `x` *distribuido en cada proceso, teniendo en cuenta las filas diferentes de cero por bloque y formando un vector global con el vector de cada proceso.*

- void **mat_vec_elementnz** (**PVector** &x, **PVector** &y)

  Producto de la matriz con el **PVector**(p. 125) `x` *distribuido en cada proceso, usando ventanas para adquirir únicamente los elementos que se necesitan para hacer el producto por estar en columnas diferentes de cero de la matriz.*

- void **assembly_schedule** ()

  Realiza el orden de tareas a partir de la cantidad de columnas diferentes de cero de los bloques a quien debe enviar o recibir datos. Cada proceso tiene los mismos datos de la matriz de envió y recepción de datos y el schedule se hace localmente por proceso.

- void **assembly_schedule_dim** ()

  Realiza el orden de tareas a partir del numero de columnas de cada bloque con quien se debe hacer envió o recepción de datos.

- void **mat_vec_schedule** (**PVector** &x, **PVector** &y)

Siguiendo el schedule, realiza la comunicación de los paquetes de datos que le corresponde enviar o recibir para realizar el producto teniendo en cuenta las columnas diferentes de cero.

- void **mat_vec_schedule_recv_calc** (**PVector** &x, **PVector** &y)

  Siguiendo el schedule, envia y recibe datos teniendo en cuenta las columnas diferentes de cero y cada vez que un proceso recibe información inmediatamente calcula el producto correspondiente.

- void **mat_vec_schedule_all** (**PVector** &x, **PVector** &y)

  Siguiendo el schedule, realiza la comunicación de los paquetes de datos que le corresponde enviar o recibir para realizar el producto enviando el vector completo que almacena.

- void **CG** (**PVector** &b, **PVector** &x, unsigned int max_iter, double tol, **PVector** &x0, unsigned int &num_iter, bool flag, **PVector** &residuals)

  Soluciona por medio del Gradiente Conjugado sin precondicionador, el sistema lineal Ax=b. Se debe ingresar el numero máximo de iteraciones, la tolerancia, un vector inicial y se obtiene como salida b, el numero de iteraciones y un vector con los residuos.

A.2.1. Descripción detallada

Clase **PCSR**(p. 114).

A.2.2. Documentación del constructor y destructor

A.2.2.1. PCSR::PCSR (unsigned int *dim_row*, unsigned int *dim_col*, MPI_Comm *COMM_* = `MPI_COMM_WORLD`)

Constructor.

Genera una matriz vacía esparcida de orden `dim_row` X `dim_col` con `p` bloques esparcidos de orden `local_n`[`my_rank`] X `local_m`[`my_rank`], distribuidos uniformemente en cada proceso.

Inicializa las variables `my_rank`, `p`, `tol`, `flag` = before_assembly y arreglos en cero

A.2.2.2. PCSR::PCSR (unsigned int * *part_row*, unsigned int * *part_col*, MPI_Comm *COMM_* = MPI_COMM_WORLD)

Constructor.

genera una matriz vacía esparcida, donde cada distribución por proceso de filas y columnas es ingresada en los arreglos `part_row` y `part_col`

Inicializa las variables `my_rank`, `p`, `tol`, `flag` = before_assembly y arreglos en cero

A.2.3. Documentación de las funciones miembro

A.2.3.1. void PCSR::end_assembly ()

Finaliza la Fase de ensamble de la matriz.

Cambia `assembly_flag` = end_assembly, Indicando así que ya no están permitido el uso de funciones que modifican a la matriz. Con los cambios realizados en la fase de ensamble, se genera cada bloque de matriz y se inician ciertas variables, según los cambios realizados. También tiene encuentra la tolerancia para aceptar o no la inserción de elementos en los bloques. Almacena el valor de bloques diferentes de cero `nbz` y almacena estos bloques en `nzbloq`

A.2.3.2. void PCSR::get_nonzero_cols ()

Por cada bloque no nulo, genera cierta información para las columnas diferentes de cero.

Por bloque diferente de cero se miran que columnas son diferentes de cero usando la función **get_nonzero_cols()**(p. 120) de **CSR**(p. 103) y por cada uno de estos bloques se genera un

arreglo con: las columnas no nulas (`vec_nzero_col`[.]), con un mapa con la ubicacion de cada una de estas columnas en la matriz (`map_nzpos_col`[.]) y un mapa donde por cada columna se tiene `UINT_MAX` si la columna es igual a cero o la posición de la columna en el vector que se retorna para columnas no nulas(`map_nzero_col`[.]).

A.2.3.3. void PCSR::get_nonzero_rows ()

Por cada bloque no nulo, genera cierta información para las filas diferentes de cero.

Inicializa por cada bloque no nulo, un vector con las filas diferentes de cero (`vec_nzero_row`[.]) y un mapa donde se tiene `UINT_MAX` si la fila es cero y sino la correspondiente posición en el vector (`map_nzero_row`[.])

A.2.3.4. void PCSR::mat_vec (const PVector & *x*, PVector & *y*) const

Producto de la matriz con el **PVector**(p. 125) `x` distribuido en cada proceso, formando vector global con el vector de cada proceso.

Para todos los productos, `x` debe ser del tamaño del numero de columnas local y el producto suma el resultado al **PVector**(p. 125) `y`.

En este producto todos los procesos envían por medio de un broadcast su correspondiente vector, para así cada proceso crear un vector global y realizar el producto local.

A.2.3.5. void PCSR::mat_vec_nonzero_cols (const PVector & *x*, PVector & *y*)

Producto de la matriz con el **PVector**(p. 125) `x` distribuido en cada proceso, en el cual se tiene en cuenta las columnas diferentes de cero por cada bloque, en un vector global formado con el vector de cada proceso.

Inicia **get_nonzero_cols()**(p. 120) en caso de no haberse ejecutado anteriormente y con los datos que se asignan, se genera un vector temporal de tamaño `dim_nzc`, cuyas entradas

son los elementos del vector global que se genero con Broadcast, en las posiciones `map_nzpos_col` y luego se sigue el formato del producto que tiene en cuenta el `map_nzero_col` del **CSR**(p. 103) secuencial

A.2.3.6. void PCSR::mat_vec_nonzero_cols_win (PVector & *x*, PVector & *y*)

Producto de la matriz con el **PVector**(p. 125) `x` distribuido en cada proceso, en el cual se tiene en cuenta las columnas diferentes de cero por cada bloque, de los vectores que se necesitan tomados con el uso de ventanas.

Los datos del **PVector**(p. 125) `x` son colocados en una ventana para que cada proceso que los necesite los tome en un vector local, luego usando `map_nzpos_col` se forma un **PVector**(p. 125) temporal con las entradas que se indicaron y finalmente se sigue el formato del producto que tiene en cuenta el `map_nzero_col` del **CSR**(p. 103) secuencial

A.2.3.7. void PCSR::mat_vec_nonzero_rows (const PVector & *x*, PVector & *y*)

Producto de la matriz con el **PVector**(p. 125) `x` distribuido en cada proceso, teniendo en cuenta las filas diferentes de cero por bloque y formando un vector global con el vector de cada proceso.

Inicia **get_nonzero_rows()**(p. 121) en caso de no haberse ejecutado anteriormente y con los datos que se asignan, ahora el producto solo se hace para las filas diferentes de cero por bloque almacenadas en `vec_nzero_row`[.], según el caso

A.2.3.8. void PCSR::mat_vec_win (PVector & *x*, PVector & *y*)

Producto de la matriz con el **PVector**(p. 125) `x` distribuido en cada proceso, tomando por medio de ventana el vector que se necesita para lograr el producto.

Todos los datos del **PVector**(p. 125) `x`, se colocan en una ventana a la cual acceden únicamente acceden los procesadores que necesitan de este vector.

A.2.3.9. void PCSR::read (const string & *filename*)

Cada proceso lee el archivo donde se encuentra la matriz.

Formato del Archivo, cuyo nombre debe ser de la forma `filename` = nombre-my_-rank.extension:

1. Dimension de la matriz: `dim_n`, `dim_m`

2. Numero de Procesos

3. Arreglos de partición de filas `local_n` y partición de columnas `local_m`

4. Numero de bloques diferentes de cero

5. Bloques diferentes de cero

6. Datos por fila: numero de columnas diferentes de cero, columna y valor para cada uno de estos datos. Los valores de los elementos son aceptados como se encuentran en el archivo y no se cambian con la tolerancia

A.2.3.10. void PCSR::start_assembly ()

Indica que se va a comenzar a ensamblar una matriz.

Inicia `assembly_flag` = start_assembly y esto permite el uso de funciones que modifican la matriz, insert, add, erase, supr_bloq, set_tolerance.

No es posible el uso de esta función luego de end_assembly.

A.2.3.11. void PCSR::write (const string & *filename*) const

Cada proceso escribe un archivo con la matriz.

En el archivo `filename`, el cual debe ser de la forma nombre-∗.extension; se almacena un archivo con el formato descrito en la función read

La documentación para esta clase fué generada a partir del siguiente archivo:

- PCSR.h

A.3. Referencia de la Clase PVector

Clase **PVector**(p. 125).

`#include <Parallel_Vector.h>`

Métodos públicos

- **PVector** ()

  Constructor de **PVector**(p. 125) *vació.*

- **PVector** (unsigned int n)

  constructor de **PVector**(p. 125) *de tamaño* `n`*, con entradas nulas*

- **PVector** (const **PVector** &)

  Copy constructor.

- ∼**PVector** ()

  Destructor.

- void **read** (const string &filename)

  Genera un **PVector**(p. 125) *a partir del archivo* `filename`.

- void **dump** (const string &filename) const

  Escribe en el archivo `filename`*, un* **PVector**(p. 125).

- void **one** (unsigned int n)

  Cambia todas las entradas de **PVector**(p. 125) *a uno.*

- void **set_zero** ()

  Cambia todas las entradas de **PVector**(p. 125) *a cero.*

- double **get_ai** (unsigned int i) const

  Retorna el valor en la posición `i`.

- unsigned int **get_dim** () const

  Retorna la dimension del **PVector**(p. 125).

- void **set_ai** (unsigned int i, double value)

  Cambia el valor en la posición `i`. *por* `value`.

- void **set_dimension** (unsigned int n)

  Cambia la dimension del vector y coloca en cero los valores de las componentes.

- void **erase** ()

  Borra un **PVector**(p. 125)*, dejándolo vació.*

- void **add** (unsigned int i, double value)

  Suma `value`*, al elemento en la posición* `i` *de un* **PVector**(p. 125).

- void **pvax** (double a)

  Multiplica todas las componentes de un **PVector**(p. 125) *por* `a`.

- double **pdot** (const **PVector** &x) const

Retorna el producto punto de el **PVector**(p. 125) *que hace el llamado de la instrucción, con el* **PVector**(p. 125) `x`.

- void **paxpy** (const double a, const **PVector** &x)

  Multiplica todas las componentes del **PVector**(p. 125) `x` *por* `a` *y se suma con el* **PVector**(p. 125) *que hizo la llamada, asignándole el resultado. y=ax+y.*

- void **pvcopy** (const **PVector** &x)

  Copia en el **PVector**(p. 125) `x`*, al* **PVector**(p. 125) *que hace la llamada de la rutina.*

- **PVector** & **operator**= (**PVector** &x)

  Sobre carga el símbolo de =.

- double **norm** (NormType Norm=L2) const

  Calcula la norma L1, L2 o INF a un **PVector**(p. 125)*, por defecto es L2.*

- void **part_vector** (**PVector** &x, unsigned int n, unsigned int *map_index) const

  Retorna en el **PVector**(p. 125) `x`*, las componentes de un* **PVector**(p. 125)*, que se indica en el arreglo* `map_index` *de tamaño* `n`.

- void **part_vector_rank** (**PVector** &x, unsigned int a, unsigned int b) const

  Retorna en el **PVector**(p. 125) `x`*, las componentes de un* **PVector**(p. 125)*, desde la posición* `a` *hasta la posición* `b`.

Amigas

- class **PCSR**

- void **PVector_send** (const **PVector** &x, unsigned int p, int tag=0)

  Función en Paralelo que envia un **PVector**(p. 125) `x`*, sin conocer su tamaño.*

- void **PVector_ssend** (const **PVector** &x, unsigned int size, unsigned int p, int tag=0)

  Función en Paralelo que envia un **PVector**(p. 125) `x`*, conociendo su tamaño* `size`*.*

- void **PVector_recv** (**PVector** &x, unsigned int p, int tag=0)

  Función en Paralelo que recibe un **PVector**(p. 125) `x`*, sin conocer su tamaño.*

- void **PVector_srecv** (**PVector** &x, unsigned int size, unsigned int p, int tag=0)

  Función en Paralelo que recibe un **PVector**(p. 125) `x`*, conociendo su tamaño* `size`*.*

- double **PVector_norm** (const **PVector** &x, NormType norm=L2)

  Función en Paralelo, que retorna por cada proceso La norma L1, L2 o INF de los **PVector**(p. 125) `x` *y* `y`*, globalmente. Por defecto es la norma L2.*

- double **PVector_dot** (const **PVector** &x, const **PVector** &y)

  Función en Paralelo, que retorna por cada proceso el producto punto de los **PVector**(p. 125) `x` *y* `y`*, globalmente.*

- unsigned int ∗ **PVector_find_partition** (const **PVector** &x)

  Función en Paralelo, que encuentra la partición de un **PVector**(p. 125) *en todos los procesos.*

- unsigned int **return_global_position** ()

  Función en Paralelo, que retorna la posición que tiene el vector globalmente.

A.3.1. Descripción detallada

Clase **PVector**(p. 125).

A.3.2. Documentación de las funciones miembro

A.3.2.1. void PVector::read (const string & *filename*)

Genera un **PVector**(p. 125) a partir del archivo `filename`.

El formato del archivo es:

1. Dimension del vector

2. Valores para cada posición del **PVector**(p. 125)

La documentación para esta clase fué generada a partir del siguiente archivo:

- Parallel_Vector.h

A.4. Referencia de la Clase Schedule

Clase **Schedule**(p. 130).

`#include <Schedule.h>`

Métodos públicos

- **Schedule** ()

  Constructor vació de schedule.

- **Schedule** (unsigned int proc, double **R)

  Constructor con matriz de recepción y envió de datos R, cuya diagonal debe tener el total de datos q debe enviar y recibir cada proceso.

- ~**Schedule** ()

  Destructor.

- void **find_schedule** ()

  Encuentra el orden de recepción y envió de datos. Si el valor S[i] el positivo el proceso recibe de s[i]-1, si es negativo el proceso envía a -S[i]-1.

- void **print_send_recv** ()

  Imprime en pantalla el orden de tareas.

- int ** **return_schedule** (unsigned int *dim_schedule)

  Retorna el arreglo de tareas para todos los procesos en cada fila respectivamente por proceso.

- double **time_total** ()

Retorna el tiempo total de comunicación, del orden de tareas encontrado.

- void **read** (const string &filename)

  Lee matriz de recepción y envió de datos.

- void **dump** (const string &filename) const

  Imprime en un archivo el orden de tareas encontrado.

A.4.1. Descripción detallada

Clase **Schedule**(p. 130).

A.4.2. Documentación de las funciones miembro

A.4.2.1. void Schedule::dump (const string & *filename*) const

Imprime en un archivo el orden de tareas encontrado.

Imprime numero de procesos, dimension de tareas pro proceso y tareas

A.4.2.2. void Schedule::read (const string & *filename*)

Lee matriz de recepción y envió de datos.

Primero se lee el numero de procesos y luego la matriz

La documentación para esta clase fué generada a partir del siguiente archivo:

- Schedule.h

Bibliografía

[1] Bisseling R. y Meese W. *Communication Balancing in Parallel Sparse Matrix-Vector Multiplication*, ETNA, Volume 21, pp. 47-65, 2005.

[2] Deitel, H. *C++ How to Program*, Prentice Hall, 1994.

[3] Golub, G. *Matrix Computations*, third edition,1983.

[4] Kulkarni, D., Sosonkina, M. *Using Dynamic Network Information to improve the Runtime Perfonmance of a Distributed Sparse Linear System Solution*, NSF/ACI-0000443 y NSF/INT-0003274, y Minnesota Supercomputing Institute. http://wwwusers.cs.umn.edu/∼saad/projects/ACI/reports.html

[5] Karypis, G.,Kumar, V., et al. *An Introduction to Parallel Computing: Design and Analysis of Algorithms*, Segunda edición, Benjamin-Cummings Publishing Co. USA, 1994.

[6] Leung, J. *Handbook of Scheduling: Algortihms, Models, and Performance Analysis*, Chapman and Hall CRC, New York, 2004

[7] Oliker, L., Li, X., et all. *Effects of ordering strategies and programming paradigms on sparse matrix computations*, SIAM Rev. **44** (2002), no. 3, 373–393.

[8] Pacheco, P. *Parallel programming with MPI*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[9] Smith, B. *An Interface for Efficient Vector Scatters and Gathers on Parallel Machines, Mathematical, Information, and Computational Sciences Division Subprogramn of the Office of Computational and Technology Research*, U.S. Deperment of Energy.

[10] Strikwerda, J. *Finite Differences and Partial Differential Equations*, SIAM second edition, 2004.

[11] Saad, Y. *Iterative Methods for Large Sparse Linear Systems*, SIAM, second edition, 2000.

[12] *Extensions to the Message-Passing Interface, forum-MPI.* http://www.mpi-forum.org/docs

[13] *Matrix Market.* http://math.nist.gov/MatrixMarket

[14] *Portable, Extensible Toolkit for Scientific Computation.* http://www-unix.mcs.anl.gov/petsc/petsc-as

IMPLEMENTACIÓN DE UNA LIBRERÍA ORIENTADA A OBJETO PARA MATRICES ESPARCIDAS EN PARALELO

Catalina María Rúa Alvarez
(787) XXX-XXXX
Departamento de Departamento de Ciencias Matemáticas
Consejero: Ph.D Esov Velázquez
Grado: Maestría en Ciencias
Fecha de Graduacion: Octubre 2007

En la búsqueda soluciones a diversas aplicaciones, científicas encontramos la necesidad de plantear modelos matemáticos, que requieren de ecuaciones diferenciales parciales (EDP). Al resolver las EDP, tratando que la solución que se obtenga con un modelo sea bien cercana a la exacta, es necesario el uso de sistemas a gran escala y además métodos de alto orden.

Diferencias finitas y elementos finitos son algunos de los métodos numéricos que se usan para resolver ecuaciones diferenciales que generalmente resultan de problemas físicos. Cuando estos métodos son usados habitualmente se trabaja con matrices de dimensiones altas y cuyos elementos distintos de cero son pocos comparados con el orden de la matriz, estas matrices se conocen como *matrices esparcidas*.

Hay otra gran variedad de problemas que involucran matrices esparcidas en temas de estadística, en aeronáutica y en teoría de grafos, entre otros. Una forma de hacer pruebas con técnicas para manejar matrices esparcidas resultantes de problemas en estos temas, es usar la matrices ejemplares que se encuentran en la página de *"Matriz Market"*.

Para las matrices esparcidas se puede aprovechar la propiedad de tener muchos elementos nulos al momento de almacenarlas, esto se haría almacenando únicamente los elementos diferentes de cero. Otra ventaja de este tipo de matrices es que al conocer las posiciones de los elementos nulos, en operaciones como la multiplicación de una matriz esparcida por un vector se conocería de antemano que algunas elementos del vector resultante son ceros y esto ayuda a reducir el número de operaciones.

Una técnica muy sencilla para manipular las matrices esparcidas, economizar el uso de memoria almacenando únicamente los valores de los elementos diferentes de cero de la matriz y reducir el número de operaciones, es el formato Compressed Sparse Rows (CSR). Este formato consiste en almacenar en tres vectores del tamaño de número de filas cada uno la siguiente información de la matriz: la cantidad de elementos diferentes de cero por fila, los índices de las columnas donde se encuentran los elementos diferentes de cero por fila y por último los valores de cada uno de los elementos diferentes de cero por fila en el mismo orden en el que fueron almacenados los índices de columnas.

Al buscar la solución matemática de muchos de los problemas que tienen matrices esparcidas se necesitará la solución de sistemas lineales y a su vez del producto de matrices esparcidas con vectores un gran número de veces. Esto se ve en los esquemas matriciales de diferencias finitas, en la búsqueda de soluciones por elementos finitos al resolver el sistema lineal ya sea con alguna versión del gradiente conjugado ó con otros métodos numéricos para aproximar la solución de sistemas lineales, y en muchas otras aplicaciones.

Esto hace que al tener matrices a gran escala, un computador común sature su memoria ya sea por la cantidad de elementos almacenados o por el número de operaciones que se debe resolver por segundo. Como no se quiere esperar demasiado tiempo para obtener la solución de problemas con matrices esparcidas, se requiere del uso de implementaciones para computadoras en paralelo.

Al trabajar con varios procesadores en paralelo, cada uno tendrá una porción de la matriz esparcida y del vector con los que se está trabajando. Las matrices en cada proceso se subdividieron en bloques y cada bloque que quede no nulo contendrá una matriz esparcida almacenada con el formato CSR. No necesariamente todos los bloques lograron almacenar una matriz, por lo cual los bloques son también esparcidos y por proceso se almacenaron con una estructura de datos similar a la del CSR.

las matrices esparcidas en paralelo distribuidas y almacenadas como se mencionó, diremos que tienen el formato Parallel CSR (PCSR).

Para obtener el vector solución al realizar el producto de una matriz esparcida con un vector distribuidos en paralelo, debe haber comunicación entre los procesadores del vector o parte del vector que cada uno de los procesos almacena. Esta comunicación es una de las principales desventajas de la programación en paralelo, porque al no hacerse en forma correcta puede suceder que el tiempo de cómputo sea alto.

En esta tesis se desarrolló una implementación para matrices esparcidas en paralelo con diferentes estrategias para atacar el problema de comunicación en el producto de una matriz esparcida con un vector.

Con el fin de verificar que las operaciones implementadas eran correctas y también ver el comportamiento del tiempo en los cómputos con los métodos implementados al cambiar la cantidad de procesadores, se realizaron pruebas con esquemas de diferencias finitas en la solución de la ecuación de transporte de onda y en la ecuación de calor con cálculo de error y orden de convergencia.

La implementación se realizó con el paradigma orientado a objetos en C++ y con las librerías de openMPI. Logrando con esta tesis una librería para trabajar con matrices esparcidas y vectores, en forma serial y paralela.

Las clases implementadas fueron: para matrices esparcidas CSR (serial) y PCSR (paralelo), para vectores PVECTOR (serial y paralelo).