eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Facultad de Informática

Grado de Ingeniería Informática

▪ Trabajo Fin de Grado ▪

Ingeniería de Computadores

Producto de Matrices Sparse en un entorno MPI

Ander González Alonso

Director

Clemente Rodríguez

Junio 2019

informatika fakultatea
facultad de informática

Agradecimientos

Tras un largo periodo de estudio y trabajo, quisiera agradecer a mi director de proyecto Clemente Rodríguez por darme la oportunidad de realizar este Trabajo de Fin de Grado, además de toda la ayuda y el apoyo ofrecido durante estos meses. Gracias también por facilitarme las herramientas necesarias y por los conocimientos adquiridos en las asignaturas de Procesadores de Alto Rendimiento y de Sistemas de Cómputo Paralelo.

Por otra parte, me gustaría agradecer a mi novia Cristina, por estar ahí siempre, apoyándome y ayudándome, no solo durante el curso del proyecto, sino durante estos años de carrera.

Otro apoyo muy grande que he tenido ha sido mi familia, apoyándome en todo momento, sin los cuales no hubiese conseguido llegar hasta aquí.

Por último, querría agradecer a mis amigos gracias a los cuales he sido capaz de evadirme del trabajo y poder desconectar en los momentos de tensión y estrés.

Resumen

Proyecto de Fin de Grado de Ingeniería Informática en la especialidad de Ingeniería de Computadores. El objetivo principal de este proyecto es: Partiendo del artículo *Parallel Sparse Matrix-Matrix Multiplication: A Scalable Solution 1-D Algorithm* [1], proponer un algoritmo eficiente para la multiplicación de matrices *sparse* de grandes dimensiones, del orden de 10^6*10^6 y una probabilidad de que un elemento sea no cero de 75E-06. Nos hemos autoimpuesto la restricción de que un nodo del clúster realice como máximo el producto de matrices del orden de 10^5 x 10^5 por motivos de la memoria necesaria. Esto implica que, si queremos resolver el caso de matrices 10^6 x 10^6, se deba complementar con una solución 2D, producto de submatrices. La memoria ocupada en este caso por las estructuras necesarias en la solución propuesta es del orden de *500MB*. El algoritmo usa la descomposición de matrices en bloques basada en el número de procesadores. Las matrices con las que se trabajan no tienen estructura, repartiéndose los elementos no nulos de forma aleatoria. Se ha mejorado el algoritmo inicial propuesto y se ha estudiado la posibilidad y las consecuencias de aplicar otros algoritmos, como el algoritmo de Cannon y se han propuesto mejoras y soluciones a los problemas detectados.

Para lograr el objetivo se ha estudiado la escalabilidad y el paralelismo de los algoritmos propuestos en dos clústeres de la facultad, uno de *1Gb* con 64 nodos de 4 *cores* cada nodo y otro de *10Gb* con 16 nodos de 4 *cores* cada uno.

Como resumen de los resultados obtenidos se puede destacar lo siguiente. Para los clústeres con los que hemos trabajado la solución óptima es de 4 procesos en un nodo, cada proceso en un *core* del procesador del nodo. Para el clúster de *1Gb* la opción *-map by node* es la mejor. Para el clúster de *10Gb* se puede realizar el reparto de los procesos a procesadores por nodo o por *core*, ya que no hay diferencias significativas. Los tiempos totales del producto matricial de 10^6 x 10^6 son comparables a los del artículo de referencia. La comparación solo puede ser grosera debido a las diferencias entre los recursos computacionales que ellos y nosotros usamos, pero los tiempos avalan nuestra solución.

Pensamos que la solución propuesta puede adaptarse fácilmente a clústeres que se diferencien en el ancho de banda de la red de la comunicación, procesadores con distinto número de *cores* y diferente jerarquía de memoria.

Queda como posible estudio el análisis teórico de la necesidad de memoria asociada a una matriz en función de su dimensión y la probabilidad de que un elemento de la matriz sea no cero.

Por último, es importante analizar las entradas/salidas a disco para almacenar matrices de semejante tamaño. El hecho de paralelizar las entradas/salidas parece más que razonable.

Índice de Contenidos

Índice de Figuras

Índice de Tablas

1

Introducción

Desde los inicios del procesamiento paralelo se han estudiado, diseñado y experimentado distintas formas de paralelizar la multiplicación de matrices.

La multiplicación de matrices es una de las operaciones más importantes para muchas aplicaciones, ya que involucran un elevado cálculo de datos con complejidad creciente de acuerdo a las dimensiones de las mismas. La multiplicación de matrices es ampliamente utilizada en diferentes campos como la computación científica, la teoría de grafos, la teoría de redes, la combinatoria, los métodos numéricos etc.

El algoritmo más sencillo para la multiplicación de matrices realiza $O(n^3)$ operaciones. *Strassen* fue el primero en demostrar que este algoritmo no es óptimo, dando un algoritmo de complejidad $O(n^{2.81})$ para el problema. Muchas mejoras le siguieron, hasta la fecha que el algoritmo para la multiplicación de matrices más rápido que existe, con una complejidad de $O(n^{2.38})$ fue obtenido por *Coppersmith* y *Winograd*. Incluso después de estas mejoras, estos algoritmos han demostrado limitaciones en su rendimiento [2].

Por otro lado, en este proyecto se va a trabajar con matrices *sparse* en lugar de densas. Las matrices *sparse* son matrices donde una gran parte de sus valores son ceros, por lo cual, estos valores pueden ser omitidos en su almacenamiento.

El objetivo principal de este Trabajo de Fin de Grado es proponer un algoritmo eficiente para la multiplicación de matrices *sparse* de grandes dimensiones del orden de 10^6 x 10^6, que usa la descomposición de matrices en bloques basada en el número de procesadores. Las matrices con las que se van a trabajar no tienen estructura, repartiéndose los valores no nulos de forma aleatoria en la matriz.

A continuación, se describen las secciones en los que se estructura la memoria:

- En la sección Introducción, se da una breve introducción del tema principal del proyecto y se describen los objetivos planteados inicialmente, la motivación alcanzada para su realización y las herramientas utilizadas a lo largo del curso del proyecto.
- En la sección Planificación Inicial, se describen los ciclos por los que pasará el proyecto, la planificación del proyecto, el sistema de respaldo de la información y la estructura de descomposición del trabajo. Se presentan los riesgos potenciales a los que está expuesto

el proyecto y el plan de prevención y contingencia para cada uno de ellos. Por último, se concluye con el análisis de la factibilidad del proyecto.

- En la sección **Diseño**, se describe la justificación de las matrices *sparse*, unos conceptos básicos acerca del tema principal del proyecto, las formas de almacenamiento de las matrices y las consideraciones iniciales que se han tenido en cuenta.
- En la sección **Desarrollo**, se describe el algoritmo principal junto con sus versiones y las funciones que componen el programa.
- En la sección **Análisis de resultados**, se analizan los resultados obtenidos de las ejecuciones del programa mediante gráficas, tablas y trazas.
- En la sección **Conclusiones**, se recogen las conclusiones a nivel de objetivos del proyecto y dificultades la realización de este y se proponen unas futuras mejoras.

1.1 Objetivos

Los objetivos principales de este proyecto son los siguientes:

1. Proponer un algoritmo eficiente para la multiplicación de matrices *sparse* de grandes dimensiones, del orden de 10^6 x 10^6 y una probabilidad de que un elemento sea no cero de 75E-06.
2. Estudiar la escalabilidad y el paralelismo en los algoritmos utilizados. Dichos algoritmos serán ejecutados en un clúster de 64 nodos con 4 *cores* cada uno, con enlaces de *1Gb* y en un clúster de 16 nodos con 4 *cores* cada uno, con enlaces de *10Gb*.
3. Proponer mejoras o soluciones a los problemas detectados.

1.2 Motivación

El objetivo principal del proyecto es proponer un algoritmo eficiente para la multiplicación de matrices *sparse* de grandes dimensiones: mejorar el algoritmo que los autores del artículo Parallel Sparse Matrix-Matrix Multiplication: A Scalable Solution with 1-D Algorithm [1] han propuesto.

Este artículo presenta una implementación novedosa de la multiplicación paralela de matrices *sparses* basándose en una descomposición 1D. Trabajan con matrices de dimensiones del orden de 10^6*10^6 con una probabilidad de que un elemento sea no cero de 75E-06. Las matrices son generadas aleatoriamente. Utilizan *hash lists* distribuidas para almacenar las matrices, ya que reducen la latencia al acceder y modificar un elemento de la matriz producto y al concatenar los resultados parciales calculados. Usan el método del recorrido de bucles *i, k, j* en el que cada procesador lleva a cabo una comunicación punto a punto con el resto para fusionar los resultados. Gracias a estas implementaciones, logran conseguir un *speedup* mucho mejor que el de otros algoritmos 1D existentes hasta la fecha. La escalabilidad tiende a ser mejor para matrices *sparses* de grandes dimensiones. En parte, esta escalabilidad se logra debido a que, a medida que aumenta el número de procesadores, disminuye la sobrecarga de comunicación MPI.

A diferencia de su enfoque 1D, en este proyecto se utiliza un enfoque 2D basado en la descomposición de submatrices. Esto se debe, en parte a que se trabaja con grandes cantidades de memoria que un procesador podría no poder manejar. Nos hemos autoimpuesto la restricción de que un nodo del clúster realice como máximo el producto de matrices del orden de 10^5 x 10^5 por motivos de la memoria necesaria. Esto implica que, si queremos resolver el caso de matrices 10^6 x 10^6, se deba complementar con una solución 2D, producto de submatrices. Por otro lado, utilizan la clase vector de C para beneficiarse de la asignación dinámica de memoria contigua y *hash lists* para almacenar las matrices. La función hash no se ha usado, aunque podría ser una mejora en caso de necesidad. El uso de la clase vector de C se aprovecha del *Garbage Collection*. El uso del *Garbage Collection* genera varios problemas como: rendimiento impredecible, mala escalabilidad, uso intensivo de recursos y pérdidas de memoria. Para mejorar esto, en este proyecto se ha generado una función que predice el tamaño que van a tener las estructuras para evitar sobreestimar a la hora de reservarlas. Finalmente se mejora la comunicación entre procesadores estableciendo una comunicación basada en un árbol binario. Gracias a estas implementaciones, se ha logrado un *speedup* comparable con el reseñado en el artículo.

1.3 Herramientas utilizadas

A continuación, se enumeran las herramientas más importantes que han sido necesarias para el desarrollo de este proyecto. Se pueden agrupar en dos grupos: Las herramientas utilizadas para interactuar con el *clúster* y las herramientas utilizadas para la generación de matrices, gráficas etc.

Dentro del primer grupo podemos encontrar las siguientes herramientas: *Cisco AnyConnect Secure Mobility Client, FileZilla, Putty, OpenMPI, C++* y *el Compilador Intel®*.

Dentro del segundo grupo podemos encontrar: *MATLAB* y *RStudio*.

1.3.1 Cisco AnyConnect Secure Mobility Client

Cisco AnyConnect Secure Mobility Client es un cliente *VPN* basado en web de *Cisco*.

La conexión *VPN* permite a los usuarios conectarse a la red de la UPV/EHU desde sitios remotos utilizando internet como vínculo de acceso. Una vez facilitadas las credenciales de acceso, los usuarios tienen un nivel de acceso muy similar al que tienen en la red local de la universidad.

1.3.2 FileZilla

FileZilla es un programa multiplataforma para la transferencia de archivos que soporta los protocolos *FTP, FTPS y SFTP*. Se ha utilizado para transferir los archivos desde el ordenador local al servidor y viceversa.

1.3.3 Putty

Putty es un cliente *SSH* y *Telnet* con el que podemos conectarnos a servidores remotos iniciando una sesión en ellos que nos permita ejecutar comandos. Ha sido necesario para conectarse al servidor de la UPV/EHU remotamente después de conectarse mediante *VPN*.

1.3.4 OpenMPI

MPI (*Message Passing Interface*) es una especificación para la programación de paso de mensajes, que proporciona una librería de funciones para *C, C++* o *Fortran* que son empleadas en los programas para comunicar datos entre procesos.

A continuación, se resumen las funciones de MPI utilizadas en el proyecto:

Para enviar datos de un procesador a otro se han utilizado las funciones MPI_Send y MPI_Isend y para recibir datos de un procesador se han utilizado las funciones MPI_Recv y MPI_Irecv. La principal diferencia entre las funciones con I (*Isend*) frente a las funciones sin ella (*send*), radica en que las primeras no son bloqueantes. Esto significa que devuelven el control a la ejecución del programa lo antes posible, es decir, hasta que el hardware y el sistema operativo sean capaces de realizar el resto del envío por sí solos. Para asegurarse de que todos los procesos han terminado todas sus operaciones de envió o de recibo pendientes, se ha utilizado la función MPI_Waitall.

Para enviar un mensaje desde el proceso raíz a todos los procesos del grupo (en este caso, los del comunicador especificado), incluyéndose a sí mismo, se ha utilizado la función MPI_Bcast.

Se ha utilizado la función MPI_Scatter para dividir un mensaje en partes iguales y enviarlos individualmente al resto de procesos y a sí mismo.

Para medir el tiempo de ejecución se ha utilizado la función MPI_Wtime, obteniendo el tiempo inicial y el final, de forma que la diferencia entre ambos es el tiempo empleado.

Para poder utilizar todas las funciones relacionadas con MPI es necesario disponer de la cabecera mpi.h.

1.3.5 C

Para poder realizar algunas de las múltiples operaciones que aparecen a lo largo del programa, es necesario incluir la cabecera de C, **math.h**. Gracias a esta cabecera se han podido resolver las siguientes operaciones: La función **pow (**base**,** exponente**)** devuelve, sobre el propio identificador de la función, el resultado que se obtiene de elevar la base al exponente. La función $\boldsymbol{log_2()}$ devuelve el logaritmo en base dos del argumento.

Se ha llevado a cabo un uso intensivo de *mallocs* para asignar la memoria necesaria para las estructuras y *frees* para liberar el bloque de memoria especificado al sistema. Además, se han utilizado punteros para acceder a las direcciones de memoria que contienen los datos de interés.

1.3.6 Compilador Intel®

Intel® C++ Compiler es un conjunto de compiladores para los lenguajes *C* y *C++,* desarrollado por Intel. Con el compilador *Intel® C++* y gracias a la compatibilidad de utilizar procesadores *Intel®*, se puede crear código que aproveche mejor los núcleos y las tecnologías integradas en estos procesadores *Intel®*.

Para poder compilar un programa basado en el lenguaje *C++* pero que dispone de las funcionalidades de la librería MPI, es necesario ejecutar la siguiente línea de comandos:

```
mpicc -O3 -o pp Ejemplo.c
```

Donde la variable **-O** controla el nivel de optimización de todo el código. Al cambiar este valor, la compilación de código tomará algo más de tiempo, y utilizará más memoria, especialmente al incrementar el nivel de optimización. En este proyecto se utilizará por defecto -O3, el nivel más alto de optimización posible. Activa optimizaciones que son caras en términos de tiempo de compilación y uso de memoria. Habilita que los bucles dentro del código se vectoricen.

Por último, se utiliza **-o** “***nombre del ejecutable***” “***nombre del fichero C***”**.c** para compilar un fichero C y crear el correspondiente ejecutable.

Una vez compilado nuestro fichero.c, se realiza la ejecución de este, y para ello es necesario ejecutar el siguiente comando:

```
mpirun -hostfile mfile64 -map-by (node/core) -n 4 pp A_131072_131072
B_131072_131072
```

Donde el parámetro **-hostfile** proporciona un archivo de *hosts* (mfile64) en los que invocar los procesos.

Dependiendo de las necesidades de nuestro programa, en el siguiente parámetro **-map-by** se podría utilizar *core* o bien *node*. *Node* lanza un proceso por nodo, ciclando por nodo de manera *round-robin*. Esto distribuye los procesos de manera uniforme entre los nodos y asigna los rangos

de *MPI_COMM_WORLD* de manera *round-robin* por nodo. Mientras que *core* mapea los procesos por núcleo.

Le sigue el parámetro *-n* que indica el número de procesadores con el que queremos ejecutar nuestro programa. Y por último el ejecutable generado en la compilación, seguido de los argumentos necesarios para el programa, que en este caso son las dos matrices que se quieren multiplicar.

1.3.7 Matlab

MATLAB (*MATrix LABoratory*) es un sistema de cómputo numérico que ofrece un entorno de desarrollo integrado (*IDE*) con un lenguaje de programación propio. Entre sus prestaciones básicas se hallan: la manipulación de matrices, la representación de datos y funciones, la implementación de algoritmos, la creación de interfaces de usuario (*GUI*) y la comunicación con programas en otros lenguajes y con otros dispositivos hardware.

En este proyecto se ha utilizado *MATLAB* para generar los ficheros que contienen las matrices *sparse* con las que se trabajan.

1.3.8 RStudio

RStudio es un entorno de desarrollo integrado (*IDE*) para el lenguaje de programación R, dedicado a la computación estadística y gráficos. Incluye una consola, editor de sintaxis que apoya la ejecución de código, así como herramientas para el trazado, la depuración y la gestión del espacio de trabajo.

En este proyecto se ha utilizado *RStudio* para obtener graficas que un traceador genera en la ejecución del programa. Para la generación de grafos se ha utilizado la librería *igraph*.

2

Planificación Inicial

A continuación, se describen los ciclos por los que pasará el proyecto, la planificación del proyecto, el sistema de respaldo de la información y la estructura de descomposición del trabajo. Se presentarán los riesgos potenciales a los que está expuesto el proyecto y el plan de prevención y contingencia para cada uno de ellos. Por último, se concluirá con el análisis de la factibilidad del proyecto.

2.1 Ciclos del Proyecto

Con motivo de agrupar las diferentes tareas, se han generado los siguientes ciclos por los que pasará el proyecto:

- Gestión: Periodo que durará todo el curso del proyecto. Este apartado englobará la planificación inicial, los objetivos del proyecto, el seguimiento y control, el análisis de riesgo y el análisis de factibilidad.
- Competencias: Periodo en el que se adquirirán los conocimientos necesarios de las herramientas a utilizar para poder realizar con éxito el proyecto. Este periodo durará todo el curso, pero con especial hincapié al inicio de este.
- Diseño: Periodo en el que se definirá el algoritmo base del proyecto con sus respectivas versiones y escenarios. Esta fase durará desde el inicio hasta la mitad del curso del proyecto.
- Desarrollo: Una vez definido el algoritmo y sus versiones, se procederá a implementarlos.
- Análisis de resultados: Con los programas finales ya disponibles, se realizarán una serie de pruebas para cada escenario posible y su posterior análisis de los resultados obtenidos.
- Documentación: Periodo que dura todo el ciclo de vida del proyecto, donde se elaborarán un documento y una defensa que recogerán el proyecto realizado.

2.2 Planificación

Tareas	Dedicación
Planificación de tareas	7
Definición de objetivos	5
Seguimiento y control	10
Reuniones con el tutor	30
Gestión	15
Competencias	8
Definición del algoritmo	17
Definición de versiones	13
Implementación Vinicial	55
Implementación Voptimizada	20
Pruebas y gráficas Vinicial	10
Pruebas y gráficas Voptimizada	7
Pruebas y gráficas en los dos clústeres	15
Análisis de los resultados	20
Memoria final	70
Preparar defensa	15
Total	317

Tabla 2.1: Tabla de dedicación

Se ha generado el siguiente cronograma de actividades con el fin de repartir las tareas ya mencionadas anteriormente en las semanas que componen el tiempo de vida del proyecto.

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20
Gestión																				
Competencias																				
Diseño																				
Desarrollo																				
Análisis de resultados																				
Documentación																				

Tabla 2.2: Cronograma de tareas

2.3 Estructura de descomposición del trabajo

La estructura de descomposición del trabajo es una herramienta fundamental que consiste en la descomposición jerárquica, orientada al entregable del trabajo que será ejecutado por el equipo de proyecto, para conseguir los objetivos de éste y crear los entregables requeridos.

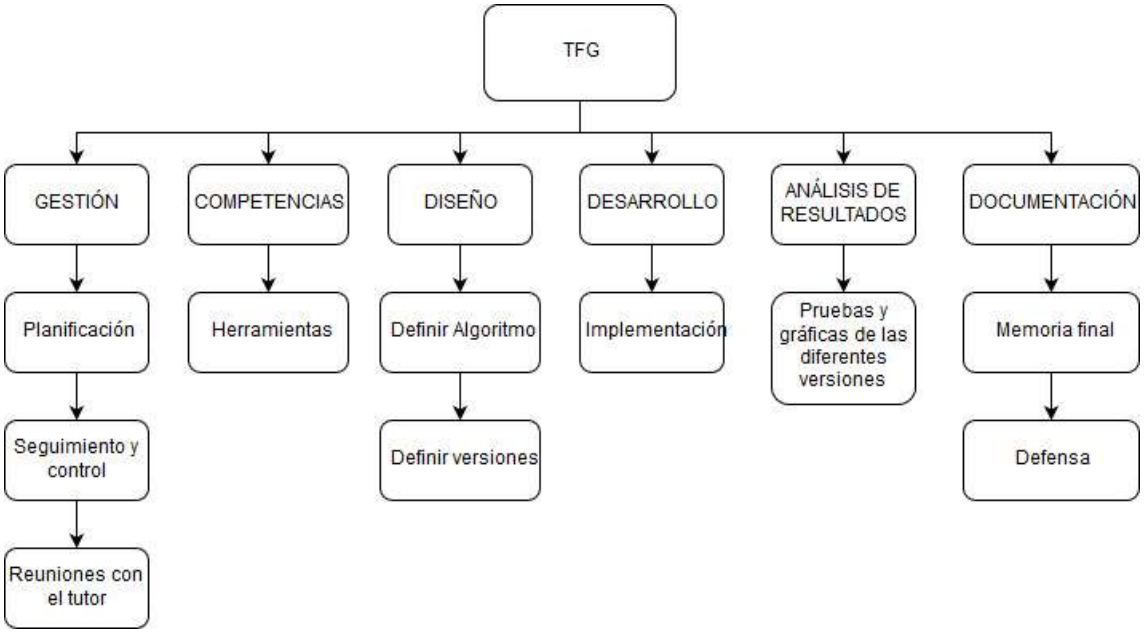

Figura 2.1: EDT

2.4 Análisis de riesgos

- Problemas con la programación
    - Nivel de riesgo: Alto
    - Impacto: Medio
    - Consecuencias: No poder avanzar hasta no solventar los errores.
    - Prevención: Detallar cada paso seguido en el código para que, en el caso de no avanzar por un error desconocido, facilite la localización de este.
    - Plan de contingencia: Invertir más tiempo en depurar el código hasta dar con el error.

- Falta de tiempo
    - Probabilidad: Baja
    - Impacto: Alto
    - Consecuencias: Examinarse en la siguiente convocatoria.
    - Prevención: Planificar bien el tiempo para llegar al plazo de entrega.
    - Plan de contingencia: Reducir el alcance del proyecto o posponer la entrega y defensa del proyecto.

- Caída del servidor
    - Probabilidad: Baja
    - Impacto: Alto
    - Consecuencias: Supondría no poder ejecutar nada en el servidor mientras este permanezca inoperativo.
    - Plan de contingencia: Invertir más horas cuando el servidor se encuentre operativo para cumplir con la planificación establecida.

- Pérdida de la información
    - Probabilidad: Baja
    - Impacto: Alto
    - Consecuencias: Únicamente se perderían los últimos cambios realizados desde la actualización anterior.
    - Prevención: Disponer de varias copias de seguridad en diferentes plataformas.
    - Plan de contingencia: Rescatar la copia de seguridad más reciente.

- Vacaciones de Semana Santa
    - Probabilidad: Media
    - Impacto: Medio
    - Consecuencias: La pérdida del ritmo de trabajo académico puede influir en el tiempo dedicado semanalmente al proyecto.
    - Prevención: Establecer una planificación
    - Plan de contingencia: Invertir más horas a las tareas retrasadas con el objetivo de terminarlas en plazo.

2.5 Análisis de factibilidad

Inicialmente no se puede afirmar a ciencia cierta que el proyecto sea factible, por lo que se va a analizar cada riesgo comentado anteriormente para asegurar la factibilidad del proyecto.

En primer lugar, gracias a haber cursado las asignaturas de Procesadores de Alto Rendimiento y Sistemas de Cómputo Paralelo, considero que dispongo de una base sólida para realizar este proyecto. Por lo que, en el caso de que surgiesen problemas con la programación, no deberían causar mayor demora en el curso del proyecto.

En segundo lugar, existe la posibilidad de que se consuma el tiempo de vida del proyecto y aún falten cosas por terminar. Para intentar solventar esto, se ha planificado todo debidamente para facilitar la realización de las tareas en sus periodos de tiempo correspondientes.

En tercer lugar, tanto el servidor en el que se ejecuta el proyecto como el sistema de información en el que se encuentra, pueden sufrir pérdidas de información o caídas. Para ello, se ha

establecido un sistema de información consistente que dispondrá de varias copias de seguridad disponibles en todo momento. En el supuesto caso de que se sufra una pérdida global del trabajo realizado, se recurrirá a invertir más tiempo para recuperar lo perdido.

Finalmente, a mitad del curso del proyecto se sitúa el periodo vacacional de Semana Santa y ya existen planes ajenos al proyecto establecidos, puesto que la recuperación de este tiempo ya se ha tenido en cuenta en la planificación.

Dado que la mayor parte de los riesgos se pueden solventar gracias a una buena gestión de la información, a una mayor inversión de tiempo o a una buena planificación de las tareas a realizar, se puede asegurar que este proyecto es factible.

2.6 Respaldo de la información

Con motivo de evitar la pérdida de información de vital importancia, se dispondrá de varias copias de seguridad. Cada copia contendrá el código del programa, las diferentes matrices y los documentos que se generarán, así como la memoria final, la presentación de la defensa de la memoria etc.

- La principal copia residirá en el ordenador del estudiante. Dado que es la copia sobre la que se realizan las modificaciones.
- Existirá una copia en el servidor, esta contendrá la última versión ejecutada en el servidor.
- Existirá una copia actualizada en una memoria USB. Se actualizará periódicamente en el caso de realizar cambios importantes.
- Existirá una copia actualizada en la plataforma de Drive. Al igual que con la memoria USB, estas se actualizarán simultáneamente. Se ha elegido Drive, debido a que es accesible por la mayoría de dispositivos con acceso a internet.

Gracias a mantener una copia en las diferentes plataformas, se podrá prevenir la perdida de información.

3

Diseño

3.1 Justificación Matrices Sparse

Las matrices *sparses* de grandes dimensiones son ampliamente usadas en la computación científica, especialmente en la optimización a gran escala, análisis estructural y de circuitos, dinámica de fluidos computacionales, *machine learning* y en general en solución numérica de ecuaciones diferenciales parciales; otras áreas de interés en donde se pueden aplicar la representación *sparse* son la teoría de grafos, teoría de redes, la combinatoria, los métodos numéricos, entre otros.

Cuando se almacenan y manipulan matrices *sparse*, es beneficioso y necesario usar algoritmos especializados y estructuras de datos que aprovechen la estructura *sparse* de la matriz. Las operaciones que utilizan algoritmos y estructuras de matrices densas son ineficientes cuando se aplican a matrices *sparses* de grandes dimensiones, ya que el procesamiento y la memoria se desperdician en las casillas que son ceros. La representación *sparse* permite almacenar los datos más eficientemente y, por lo tanto, requieren un almacenamiento significativamente menor.

Un objetivo de diseño fundamental en el uso de matrices *sparse* será el de utilizar poca memoria para poder escalar el tamaño de las matrices de forma eficiente entre los nodos de un clúster y permitir comunicaciones sencillas en el uso de los búferes.

3.2 Conceptos básicos

A continuación, se presentarán algunos conceptos necesarios para la exposición del producto de matrices *sparse*.

3.2.1 Notación y terminología utilizada

nnc = número de elementos distintos de 0.

nfil = número de filas de la matriz.

ncol = número de columnas de la matriz.

N_i = es el índice de la columna/fila de inicio para el procesador p_i en el vector *I*.

A(i,:) = i enésima fila de A.

B(:,j) = j enésima columna de B.

A(i,j) = elemento en la posición (i,j) de A.

npr = número de procesadores.

3.2.2 Multiplicación de matrices

Para multiplicar dos matrices A y B es necesario que el número de columnas de A coincida con el número de filas de B. De tal forma que, si A es una matriz de dimensión m*n y B una matriz de dimensión n*r, la matriz resultante C sería de dimensión m*r.

El elemento C_{ij} se obtiene realizando el producto interno entre el vector fila I y el vector columna J, recorrido *i, j, k*.

for *i*
  for *j*
    for *k*
      $C_{i,j} = A_{i,k} * A_{k,j}$

$$\sum_k A(i,:)\ x\ B(:,j)$$

$$A*B = \begin{Bmatrix} 2 & 0 & 1 \\ 3 & 0 & 0 \\ 5 & 1 & 1 \end{Bmatrix} * \begin{Bmatrix} 1 & 0 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 0 \end{Bmatrix} = \begin{Bmatrix} 2*1+0*1+1*1 & 2*0+0*2+1*1 & 2*1+0*1+1*0 \\ 3*1+0*1+0*1 & 3*0+0*2+0*1 & 3*1+0*1+0*0 \\ 5*1+1*1+1*1 & 5*0+1*2+1*1 & 5*1+1*1+1*0 \end{Bmatrix} =$$

$$= \begin{Bmatrix} 3 & 1 & 2 \\ 3 & 0 & 3 \\ 7 & 3 & 6 \end{Bmatrix}$$

Realizando el producto interno de esta manera, se consigue un *stride* de 1 al acceder consecutivamente a los elementos de la fila de A, mientras que al acceder a los elementos de la columna de B el *stride* es *nfil*. Esta situación empeora la tasa de aciertos de la memoria cache.

Esto se mejora realizando el recorrido *i, k, j*. Este recorrido prima el acceso a vectores fila.

for *i*
  for *k*
    for *j*
      $C_{i,j} = A_{i,k} * B_{k,j}$

$$\sum_j A(i,k) * B(k,:)$$

$$A*B = \begin{Bmatrix} 2 & 0 & 1 \\ 3 & 0 & 0 \\ 5 & 1 & 1 \end{Bmatrix} * \begin{Bmatrix} 1 & 0 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 0 \end{Bmatrix} = \begin{Bmatrix} 2*1+0*1+1*1 & 2*0+0*2+1*1 & 2*1+0*1+1*0 \\ 3*1+0*1+0*1 & 3*0+0*2+0*1 & 3*1+0*1+0*0 \\ 5*1+1*1+1*1 & 5*0+1*2+1*1 & 5*1+1*1+1*0 \end{Bmatrix} =$$

$$= \begin{Bmatrix} 3 & 1 & 2 \\ 3 & 0 & 3 \\ 7 & 3 & 6 \end{Bmatrix}$$

De esta forma se mejora el acceso a memoria. El acceso a los elementos contiguos de las filas de la matriz A sigue siendo *stride* 1, mientras que el acceso a los elementos contiguos de las filas de la matriz B pasa de ser *stride nfil* a ser *stride* 1. Para conseguir *stride* 1, la matriz C se recorre *n* veces (Figura 3.1).

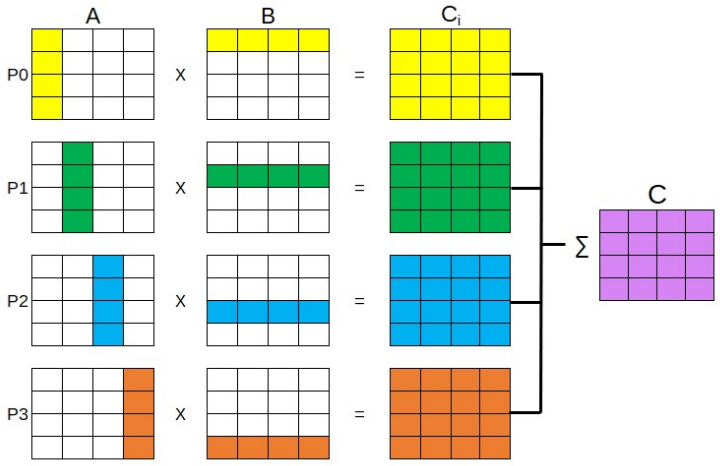

Figura 3.1: Recorrido i,k,j. npr = 4

3.2.3 Concepto Matriz sparse y sparsity

Una **matriz *sparse*** o dispersa es una matriz en la que la mayoría de sus elementos son nulos. Por el contrario, si la mayoría de elementos son distintos de cero, la matriz se considera densa.

El número de elementos de valor cero dividido por el número total de elementos se denomina ***sparsity***, que equivale a (1 – densidad de la matriz). Por lo tanto, una matriz es *sparse* cuando su *sparsity* sea mayor de 0.5, siendo 1 la suma entre los porcentajes de *sparsity* y densidad.

¿Por qué usar matrices *sparse* en lugar de matrices densas?

El uso de matrices *sparse* para almacenar datos que contienen un gran número de elementos con valor nulo puede ahorrar una cantidad significativa de memoria y acelerar el procesamiento de esos datos. Las matrices *sparse* también tienen ventajas en términos de eficiencia computacional. A diferencia de las operaciones con matrices densas, las operaciones con matrices *sparse* no realizan cálculos innecesarios, como la multiplicación de dos valores en los que uno de ellos o ambos son cero. La eficiencia resultante puede llevar a una mejora en el tiempo de ejecución para los programas que trabajan con grandes cantidades de datos dispersos. El coste a pagar es el acceso a los valores distintos de cero, que necesitan de estructuras de datos específicas.

3.3 Formato de Almacenamiento de las matrices

Los esquemas de almacenamiento tratan de explorar las características de las matrices *sparse* y almacenar únicamente sus elementos no nulos. Las matrices serán generadas sin estructura, repartiéndose los valores no nulos de manera aleatoria en la matriz.

Existen varios formatos de almacenamiento, pero únicamente nos centraremos en los tres siguientes: *COO (Coordinate Format), CSR (Compressed Sparse Row)* y *CSC (Compressed Sparse Column).*

COO: Formato de coordenadas

Los vectores: fila, columna y valor almacenan los índices de la fila, el índice de la columna y el valor de los elementos no nulos de la matriz (Figura 3.2). *COO* es una representación general de una matriz *sparse*, donde la capacidad de almacenamiento requerida depende de los valores no nulos. Este formato lo proporciona MATLAB, y es el que se ha utilizado para obtener las matrices *sparse* aleatorias. Se puede primar el orden por filas y por columnas (fundamentalmente por lenguajes como Fortran, R, MATLAB…).

0	0	1	0
0	4	0	0
0	0	0	0
3	0	0	7

I	J	V
0	2	1
1	1	4
3	0	3
3	3	7

Figura 3.2: Formato COO para una matriz de 4x4, ordenado por filas

Este formato es muy eficiente cuando se quiere acceder de forma secuencial a todos los elementos, como por ejemplo cuando se quiere hacer el producto de matriz por vector. El uso de memoria de este formato sigue la siguiente expresión:

$$Tamaño\ COO = 3 * nnc$$

CSR: Comprimido por filas

Al igual que el formato *COO*, los índices de columnas y valores se almacenan explícitamente. En cambio, se añade un tercer vector de punteros que contiene los índices del inicio de cada fila. Gracias a este vector, se puede saber los elementos que hay en cada fila. La longitud del vector es n+1, siendo n el número de filas de la tabla. En la posición n+1 del vector de índices de filas se encuentra el número de elementos no nulos de la matriz (Figura 3.3).

Las ventajas principales de este formato son:

- Si lo comparamos con el formato anterior, el *COO*, aparentemente el uso de memoria es menor, ya que la idea de este formato es almacenar el índice de fila implícitamente. La memoria que usa este formato sigue la siguiente expresión:

$$Tamaño\ CSR = 2 * nnc + nfilas + 1$$

  Igualando ambas ecuaciones se consigue calcular la condición en la que ambos métodos son iguales:

$$3 * nnc = 2 * nnc + nfilas + 1 \rightarrow \boxed{nnc = nfilas + 1}$$

  Como podemos observar en la anterior expresión, únicamente el formato *COO* será mejor que el *CSR* cuando *nnc* sea mayor o igual al número de filas más uno. En este proyecto se trabaja con matrices *sparses* que tienen en cada fila del orden de 75 *no ceros*. En la siguiente expresión podemos observar que, trabajar con el formato *COO* frente al *CSR* conllevaría una carga de memoria 1.5 veces mayor.

$$^{3nnc}/_{2nnc + (nfil + 1)} = {}^{3}/_{(2 + \frac{10^6}{75 * 10^6})} \approx 1.5$$

  Además, el uso de punteros favorece el acceso a los elementos *nnc* de una fila, facilitando el acceso con *stride* 1. Es idóneo para nuestro algoritmo (i,k,j).

Dadas las ventajas que ofrece, en este proyecto se utilizará este formato.

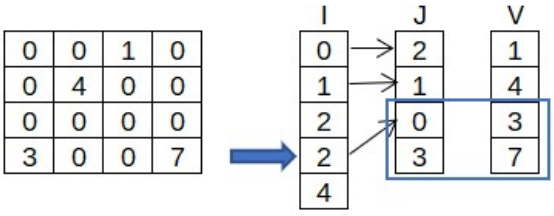

Figura 3.3: Formato CSR, acceso a los elementos de la fila 3. (3,0)→3 y (3,3)→7

CSC: Comprimido por columnas

Al igual que CSR, pero almacenando por columnas (Figura 3.4). Este formato se suele utilizar para lenguajes como Fortran, R etc.

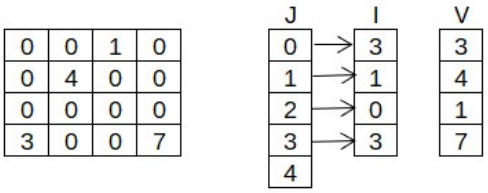

Figura 3.4: Formato CSC

3.4 Consideraciones Iniciales

- Se trabajará con matrices cuadradas, a semejanza del artículo [1].
- Las matrices a multiplicar serán *sparses*, rellenadas de manera aleatoria.
- Las dimensiones de las matrices van desde 128x128 hasta 131072x131072.
- Los ficheros generados con *Matlab* que contienen cada matriz, dispondrán de tres líneas extras a los datos de las matrices, en las cuales guardarán el número de filas, el número de columnas y el número de valores distintos de cero [Anexo 8.2].
- Los valores de las matrices son en coma flotante (*float*), mientras que los índices de filas y columnas son enteros (*int*).
- Se trabajará con un número de procesadores potencia de 2 hasta 256 procesadores.
- Los algoritmos serán ejecutados en dos clústeres. Uno de *1Gb* de 64 nodos con 4 *cores* cada uno y otro de *10Gb* con 16 nodos de 4 *cores* cada uno.

A continuación, se muestra un ejemplo (Figura 3.5) de una matriz *sparse* en fichero de texto generada con MATLAB:

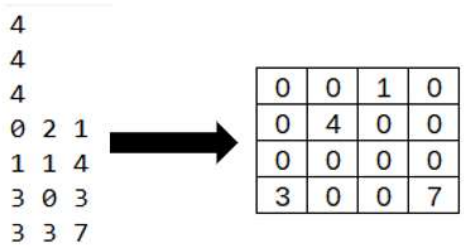

Figura 3.5: Matriz en Fichero de texto obtenido desde MATLAB. Formato COO

4

Desarrollo

4.1 Equipos Utilizados

Como ya se ha comentado anteriormente, el programa está alojado en un *clúster* de la facultad de informática de la UPV y para poder acceder a él, se ha hecho remotamente mediante *vpn*. Por lo tanto, en este proyecto únicamente influyen las características de los servidores a la hora de realizar las ejecuciones del programa.

4.2 Franja vertical y horizontal

Con motivo de mejorar la lectura de este proyecto, se describe a continuación los conceptos franja vertical y franja horizontal en el producto matricial C = A*B. (Figura 4.1)

Definimos como franja vertical el conjunto de columnas que corresponden a cada procesador de la matriz A. Siendo una matriz de 7x8 y dos procesadores, al primer procesador (P0) le corresponderán las 4 primeras columnas, formando así la primera franja vertical de dimensiones 7x4, y al segundo procesador (P1) le corresponderán las 4 siguientes columnas, formando así la segunda franja vertical de dimensiones 7x4.

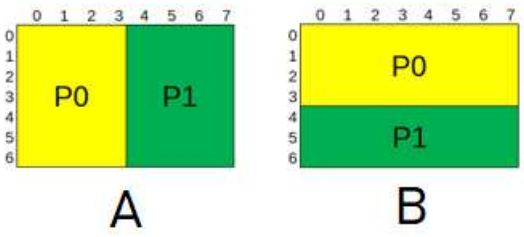

Figura 4.1: Franja Vertical y Horizontal para 2 procesadores de matrices A y B

Definimos como franja horizontal el conjunto de filas que corresponden a cada procesador de la matriz B. Siendo una matriz de 7x8 y dos procesadores. Al primer procesador (P0) le corresponderán las 4 primeras filas, formando así la primera franja horizontal de dimensiones 4x8. Y al segundo procesador (P1) le corresponderán las 3 siguientes filas, formando así la segunda franja horizontal de dimensiones 3x8.

4.3 Descripción del algoritmo

El algoritmo se ha dividido en fases. A continuación, se describirá cada una de ellas, incluyendo un ejemplo para favorecer la explicación:

4.3.1 FASE 1

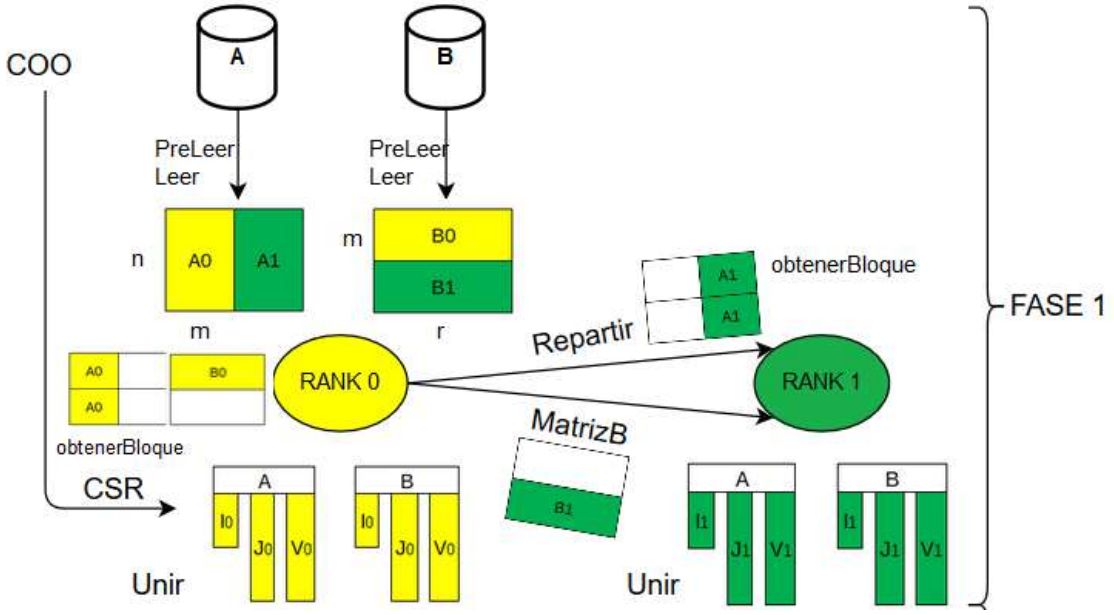

Figura 4.2: Fase 1. Creación y reparto de las estructuras asociadas a las matrices A y B a los procesadores

La fase 1 engloba la lectura de las matrices A y B, la reserva de memoria, el reparto de las submatrices a los procesadores y la unión de las submatrices en cada procesador (Figura 4.2).

Inicialmente, ambas matrices están contenidas en un fichero de texto cada una en formato *COO*. Previo al algoritmo optimizado, cada *rank* almacenaba la franja en su totalidad. A medida que aumentaban las dimensiones de las matrices, también aumentaba el tamaño de las franjas. Dado que un *Rank* no puede almacenar en memoria el tamaño total de sus franjas correspondientes, *root* (*Rank* 0) llama a la función ***PreLeer*** (Figura 4.3).

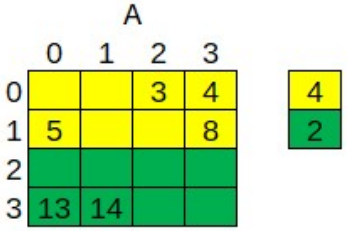

Figura 4.3: Preleer la matriz A. Obtiene los valores *nnc* para cada franja. *npr*=2

Esta función se encargada de leer los datos del fichero y almacenar en cada posición de una estructura auxiliar el *nnc* que va a tener cada franja. Gracias a esta función evitaremos sobreestimar a la hora de reservar los tamaños de cada estructura, puesto que se reserva para cada *rank* exclusivamente el *nnc* y no toda la franja. En el ejemplo de la figura 4.3, corresponden 4 *nnc* para P0 y 2 *nnc* para P1.

Una vez *root* (*Rank* 0) dispone de los *nnc* que componen cada franja horizontal, puede reservar ese tamaño para las estructuras *J* (Vector de columnas) y *V* (Vector de Valores) y almacenar en formato *CSR* la franja leída mediante la función ***Leer*** (Figura 4.4).

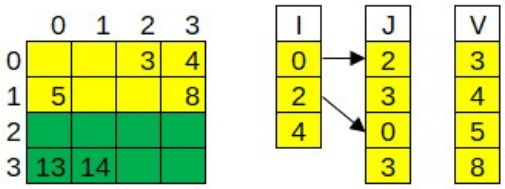

Figura 4.4: Leer la primera franja horizontal de la matriz A

Esta función se encarga de leer del fichero los datos correspondientes a una franja horizontal y almacenarlos en las estructuras. Toda esta tarea recae en *Root*. En total son tres estructuras, manteniendo el formato CSR: una para los elementos de cada Fila(I), una para las Columnas(J) y una última para los Valores(V).

El siguiente fragmento de pseudocódigo resume la funcionalidad del reparto de la matriz A.

```
for (i=0;i<npr;i++){
      if (pid == root){
            //Reservar las estructuras de la franja horizontal
            Leer();
      }
      for(j=0;j<npr;j++){
            Repartir();
      }
      if (pid == root) //Liberar estructuras
}
```

Para llevar a cabo el producto de matrices, como se ha explicado en el apartado 3.2.2, es necesario que una de las dos matrices se reparta por columnas, mientras que la restante se puede repartir por filas. En este caso se ha elegido la matriz A para ser repartida por columnas al resto de procesadores. Existe una complejidad en el reparto de la matriz A y es que, dado que las matrices están ordenadas por filas, es necesario que *root* (*Rank* 0) lea la franja de filas correspondientes. Una vez obtenida esa franja, repartir a cada procesador el bloque formado por las columnas que corresponden a cada uno. Para repartir la matriz A en su totalidad, excluyendo al *rank* 0, puesto que este ya tiene a su disposición sus estructuras, habría que realizar un total de $npr(npr-1)$ comunicaciones punto a punto. Por otro lado, si se hiciese con *Scatterv* se conseguiría paralelizar los envíos. Esto reduciría considerablemente el tiempo de reparto de las

matrices, pero a su vez conllevaría una desventaja ya que, un nodo tendría que disponer de la memoria suficiente para tener almacenadas las matrices. Para minimizar la memoria necesaria (*proporcional a nfil x npr*) hemos optado por comunicación punto a punto.

Todos los procesadores llaman a la función ***Repartir*** (Figura 4.5).

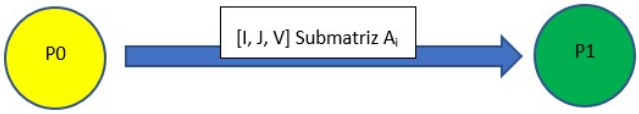

Figura 4.5: Repartir la submatriz A por bloques. Siempre se envía I y nnc, y opcional J y V

Esta función se encarga de realizar las transferencias de bloques entre *Root* y el resto de procesadores. *Root* es el único encargado de llamar a la función ***obtenerBloque*** (Figura 4.6).

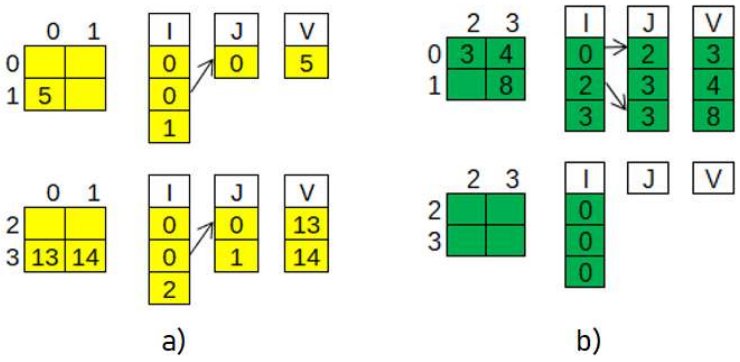

Figura 4.6: a) Obtener los bloques de la submatriz A que corresponden a P0. b) Obtener los bloques de la submatriz A que corresponden a P1.

Esta función se encarga de seleccionar de la franja horizontal el bloque correspondiente al procesador en cuestión. Una vez dispone de un bloque, este lo envía al procesador correspondiente. La comunicación se realiza de la siguiente forma: Siempre se envía *nfil*, la estructura de *I (int)* y *nnc*. Se podría evitar enviar el *nnc*, ya que la estructura *I* los contiene en la última posición *(Se ha hecho por claridad del código).* Si *nnc > 0*, entonces se procede a enviar las estructuras de *J (int)* y *V (float).* Dicho procesador llama a la función ***unir*** (Figura 4.7) para construir su franja vertical.

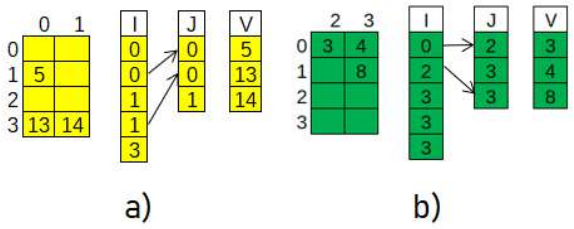

Figura 4.7: a) Unir los bloques para formar la franja vertical del P0. b) Unir los bloques para formar la franja vertical del P1.

Esta función se encarga de concatenar el bloque recibido a lo acumulado. De esta forma, cada procesador construye su propia franja vertical.

Una vez repartida la matriz A, se procede a realizar el reparto de B. El procedimiento es más sencillo que repartir A, dado que las matrices están ordenadas por filas y el reparto también lo es por filas. Al igual que con la matriz A, el *rank 0* llama a la función ***preLeer*** (Figura 4.8) para conocer el *nnc* que contiene cada franja horizontal y así evitar sobreestimar a la hora de reservar el tamaño de las estructuras.

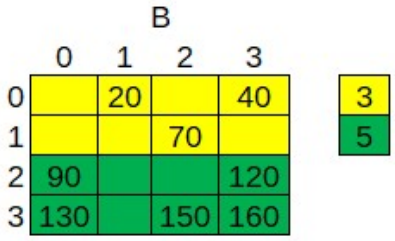

Figura 4.8: PreLeer la matriz B. Obtiene los valores nnc para cada franja horizontal. npr = 2.

Después el *rank* 0 mediante la función ***MatrizB*** lee las franjas horizontales con la función ***Leer*** (Figura 4.9) y envía cada una a su *rank* correspondiente (Figura 4.10). En número de envíos que se realizan es proporcional al número de procesadores excluyendo al *rank* 0 *(npr -1 comunicaciones).*

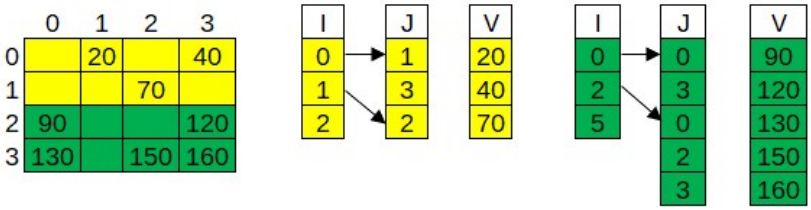

Figura 4.9: Leer la matriz B. La primera franja horizontal corresponde a P0 y la segunda a P1.

La función *MatrizB* se encarga de llamar a la subrutina leer para obtener una franja horizontal y enviarla al procesador correspondiente.

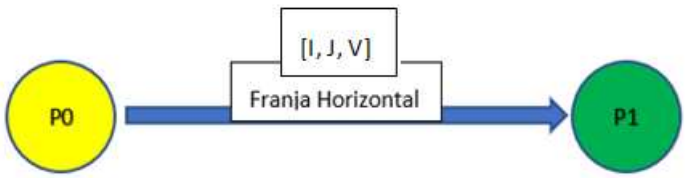

Figura 4.10: Reparto de la franja horizontal de la matriz B

4.3.2 FASE 2

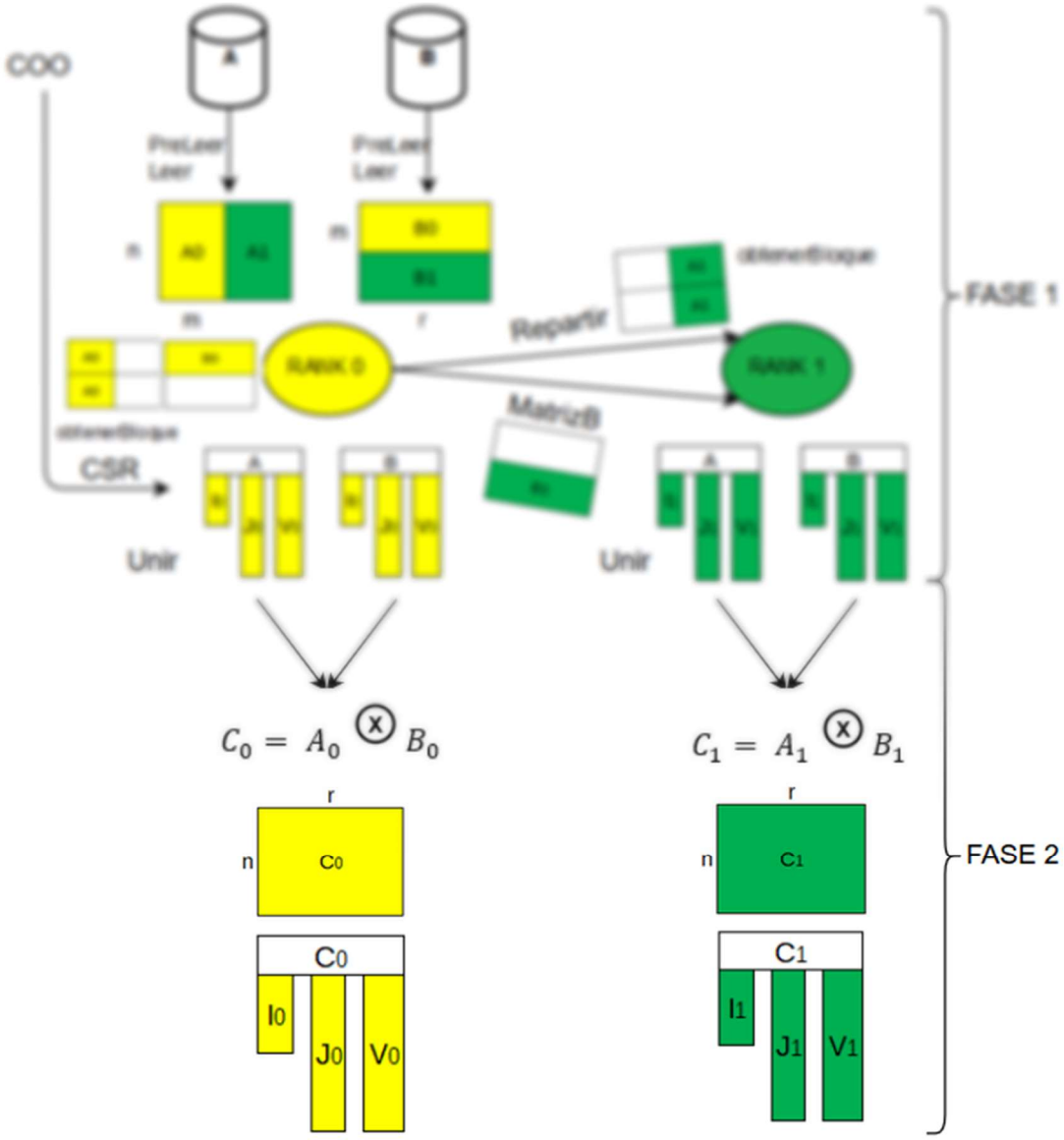

Figura 4.11: Fase 2. Producto i,k,j en cada procesador

La fase 2 engloba el recorrido i,k,j que realiza cada procesador para realizar el producto de matrices (Figura 4.11).

Una vez que cada *rank* tenga su franja vertical y horizontal, se disponen a realizar el producto entre ambas. El algoritmo está basado en el *recorrido i,k,j*. Previamente a realizar el producto, se predice el tamaño que van a tener las estructuras para almacenar la matriz C y así evitar sobreestimar en memoria innecesaria. Para predecir este tamaño, se siguen los mismos pasos que para realizar el producto. Se recorren los elementos de la franja vertical y se van acumulando en una variable, por cada uno de ellos, el número total de los elementos que le corresponden de la franja horizontal. Cada procesador mediante la función ***Producto*** (Figuras 4.12 y 4.14) realiza el sumatorio de los productos entre las columnas y filas que le corresponden. Así mismo, el cálculo de C_i (Figuras 4.13 y 4.15) de cada procesador se puede resolver de la siguiente forma:

$$C_i = \sum_{k=N_i}^{N_{i+1}-1} A(:,k)\ x\ B(k,:)$$

Donde N_i es el índice de la columna/fila de inicio para el procesador p_i en el vector I y $A(:,k)$ corresponde a la k enésima columna de A.

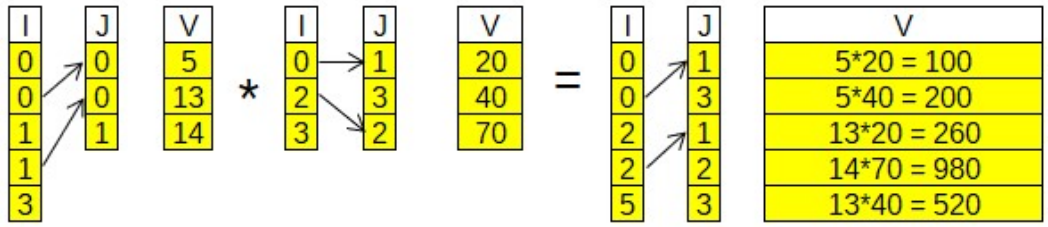

Figura 4.12: Producto entre la franja vertical y horizontal realizado por el rank 0

	0	1	2	3
0				
1		100		200
2				
3	260	980		520

Figura 4.13: Matriz C_0 resultante del producto entre la franja vertical y horizontal del rank 0

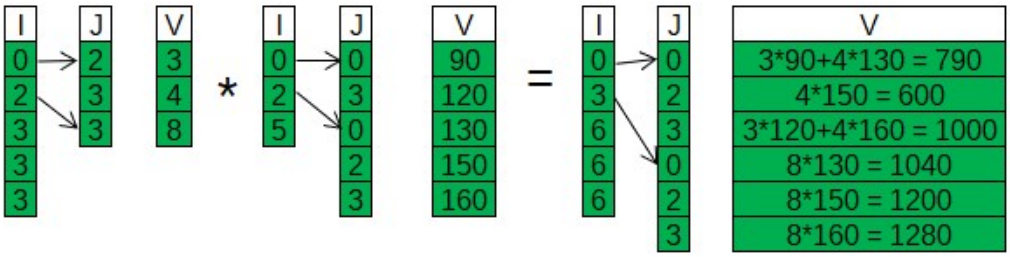

Figura 4.14: Producto entre la franja vertical y horizontal realizado por el rank 1

	0	1	2	3
0	790		600	1000
1	1040		1200	1280
2				
3				

Figura 4.15: Matriz C_1 resultante del producto entre la franja vertical y horizontal del rank 1

La función Producto se encarga de realizar el producto entre la franja vertical de la Matriz A y horizontal de la Matriz B de cada procesador y almacenarlos en las estructuras. Para ello, se recorren los elementos de la franja vertical y por cada uno de ellos se realiza el producto con sus elementos correspondientes de la franja horizontal. Antes de realizar el producto se llama a la

subrutina ***ordenar*** (Figuras 4.16 y 4.17). Esta función se encarga de comparar cada nuevo elemento a introducir con los ya existentes en las estructuras, ordenándolos de menor a mayor. Se pueden dar varios casos: Aún no hay valores introducidos en las estructuras de C_i, por lo que se introducen *J* y *V* sin problemas. Dentro de la misma fila *I*, las *J* coinciden, dando lugar a una acumulación de los valores (Figura 4.16). Por último, la *J* del elemento a introducir es menor que la del último elemento introducido (Figura 4.17). Si sucede esto, hay que desplazar la *J* y *V* del último elemento introducido, generando un hueco para colocar las nuevas *J* y *V* del nuevo elemento a introducir.

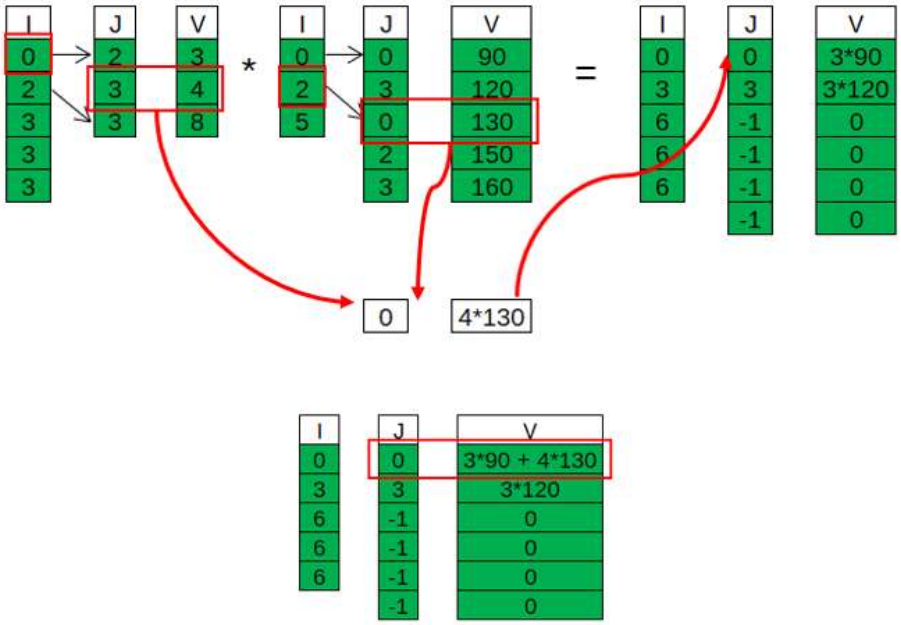

Figura 4.16: Ordenar. Las columnas coinciden y los valores se acumulan.

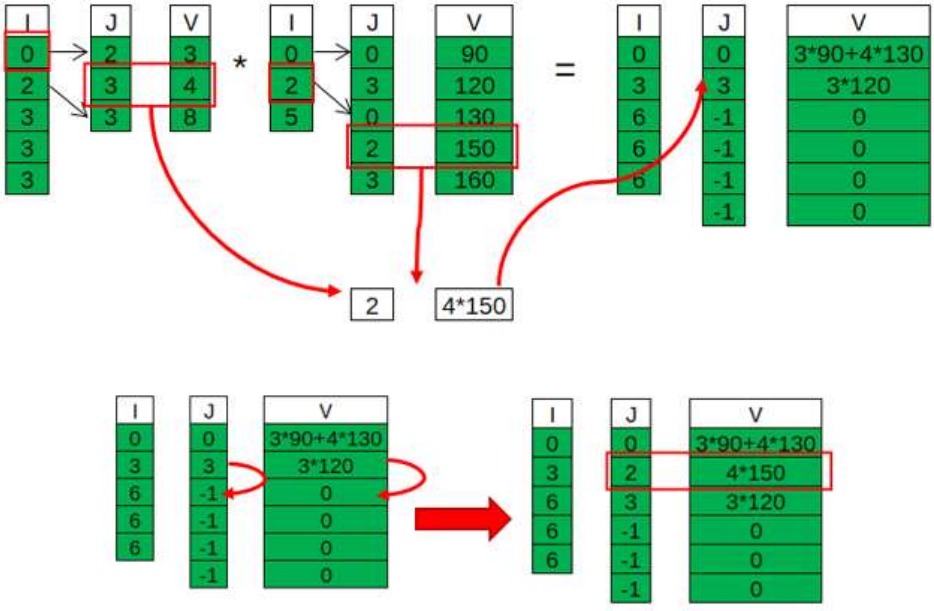

Figura 4.17: Ordenar. La columna del elemento a introducir es menor que la del elemento ya introducido.

4.3.3 FASE 3

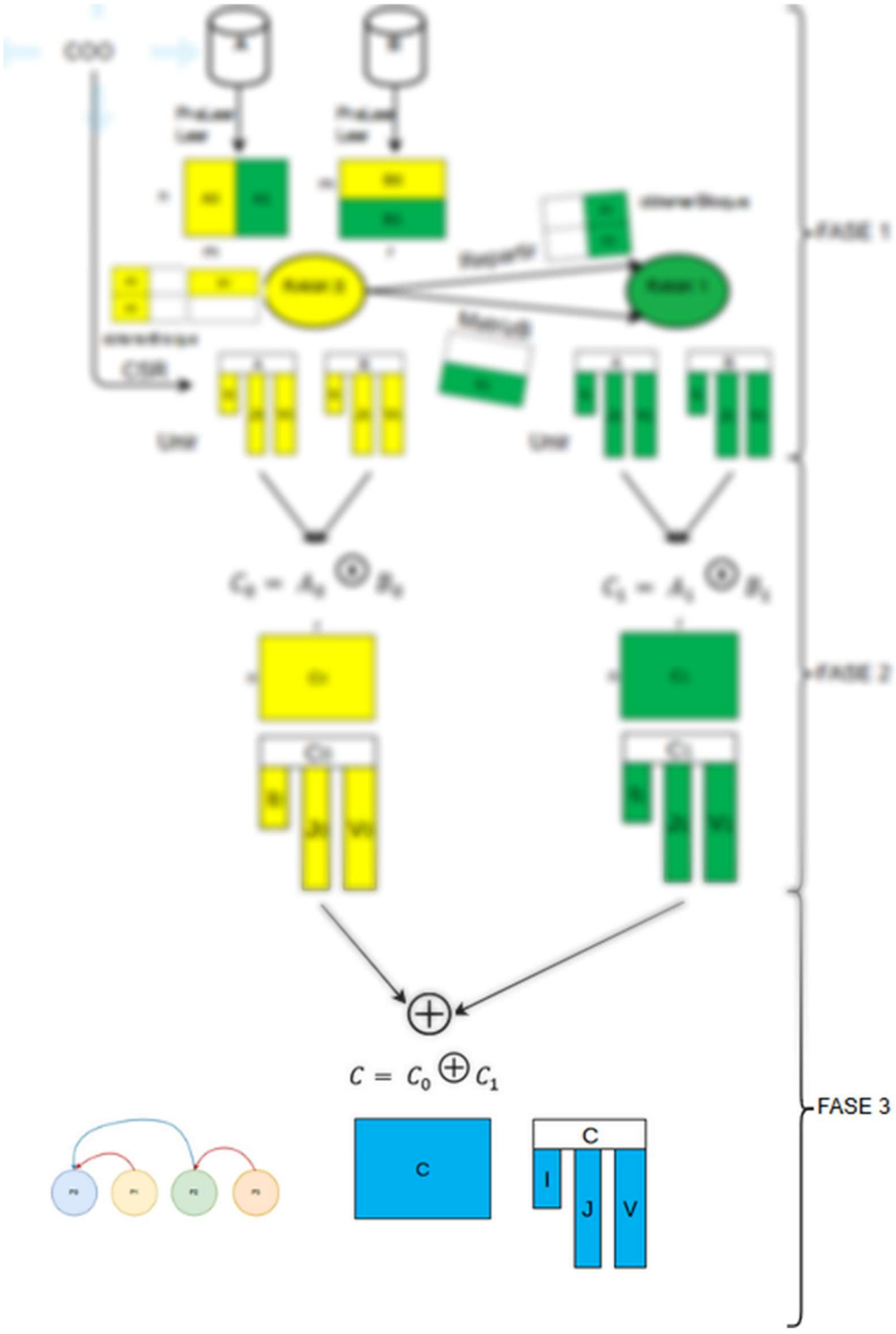

Figura 4.18: Fase 3. Obtención de la matriz C, a partir de la suma de las Ci de cada procesador i

La fase 3 engloba la comunicación de las matrices C_i parciales y el sumatorio de estas, que da lugar a la matriz C final (Figura 4.18).

El paso final consiste en que, mediante la función ***Sumar*** (Figura 4.19), cada procesador envíe su matriz al *Root* y este realice el sumatorio mediante la función ***acumular*** de todas las matrices recibidas, dando origen a la matriz C final (Figura 4.20).

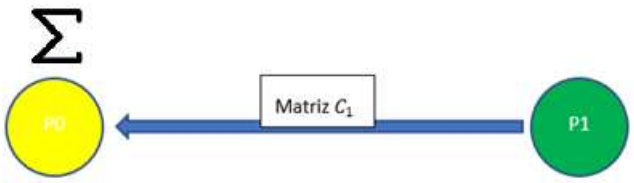

Figura 4.19: Sumar y acumular. Se envía la matriz C_1 al rank 0 y este la concatena con la suya.

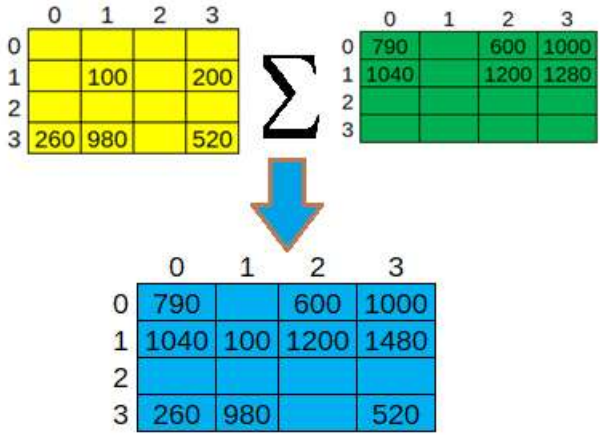

Figura 4.20: Matriz C definitiva

Para una correcta ordenación, puesto que cada estructura recibida puede contener elementos anteriores o posteriores a los ya acumuladas, se llama a la función ***AcumularFila***. Esta función se encarga de comparar cada nuevo elemento a introducir con los ya existentes en las estructuras. Si una de las dos estructuras está vacía entonces introducirá los elementos de la no vacía. Si por el contrario las dos tienen elementos. Comprobará ambas, ordenándolas de menor a mayor o acumulando los valores en el caso de que las columnas de ambos elementos coincidan. Este proceso es muy similar al de la función ***ordenar*** (Figuras 4.16 y 4.17).

4.4 Optimización del algoritmo

Con motivo de conseguir un algoritmo más eficiente se han realizado una serie de cambios al algoritmo presentado en el apartado anterior.

El principal cambio radica en la comunicación del envío de los productos parciales del resto de procesadores a *root* (*Rank* 0). Se ha propuesto una comunicación basada en un árbol binario cuya finalidad es reducir la carga de trabajo de *root,* paralelizando la comunicación (Figura 4.21). El algoritmo de esta comunicación quedaría de la siguiente manera (Figura 4.20):

```
for (i=0;i<(log2(npr));i++){
    if(pid%2^(i+1) == 0)MPI_Recv from pid-i^2;
    if(pid%2^(i+1) == 2^i)MPI_Send to pid-i^2;
}
```

Figura 4.21: Pseudocódigo de la comunicación punto a punto del algoritmo en árbol binario utilizado

Además, reparte las sumas parciales de C_i: aprovechando las propiedades asociativa y conmutativa de la suma de matrices. Cada procesador que recibe un producto parcial de otro procesador tiene que acumularlo a lo suyo. De esta manera *root* únicamente tendrá que realizar tantas acumulaciones como $\log_2(npr)$.

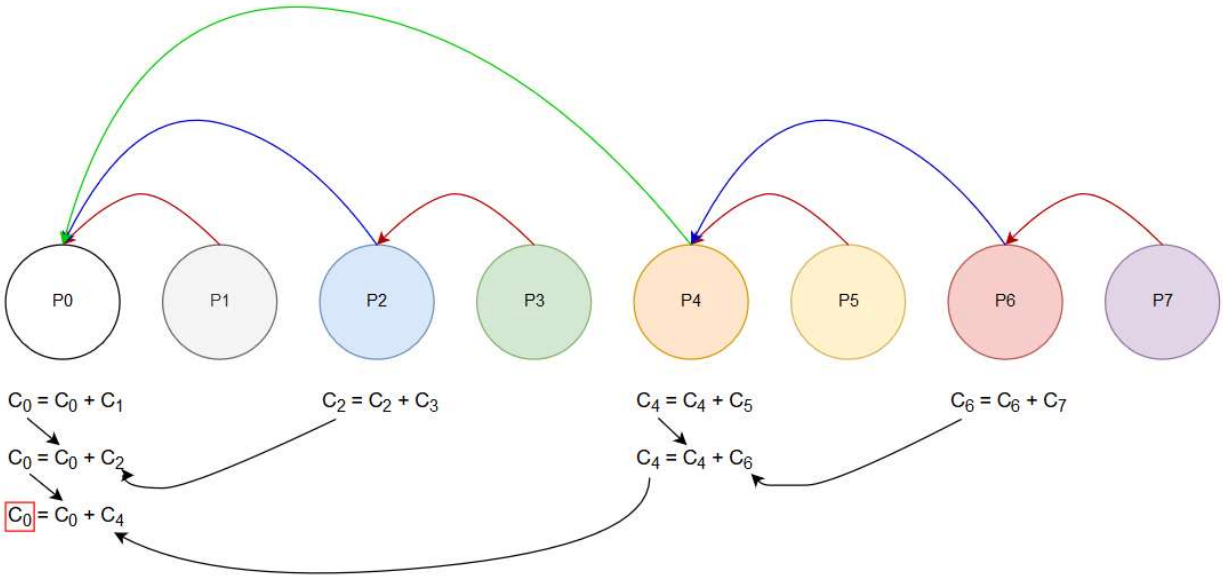

Figura 4.22: Comunicación usando un árbol binario y sumas parciales realizadas en cada proceso. npr = 8

El segundo cambio consiste en evitar sobreestimar en tamaño cuando se reservan las estructuras que almacenarán el producto parcial de cada procesador. Para calcular el tamaño de la matriz Ci se recorren los elementos de la franja vertical y se van acumulando en una variable, por cada uno de ellos, el número total de elementos que le corresponden de la franja horizontal.

El último cambio se basa en modificar los *MPI_Send* (envíos bloqueantes) por los *MPI_Isend* (envíos no bloqueantes). De esta forma se consigue agilizar el envió de los mensajes, ya que estos ocupan bastante tamaño y puede llevar tiempo cargarlos en el buffer. Al ser no bloqueantes, la función devuelve el control de la ejecución del programa lo antes posible, esto permite que, si se realizan operaciones o cálculos entre envíos, se superpongan.

5

Análisis de resultados

Para determinar el grado de cumplimiento de los objetivos propuestos en este proyecto, se han analizado los resultados obtenidos de las ejecuciones del programa mediante tablas, gráficas y trazas.

Para generar este material gráfico se han recogido los tiempos mediante la función *MPI_Wtime()* de las siguientes funcionalidades del programa:

Repartir Matriz A	Memoria
	Comunicación
Repartir Matriz B	Memoria
	Comunicación
Producto	Cálculo
Sumar	Comunicación
	Cálculo

Tabla 5.1: Funcionalidades

Como se puede observar en la tabla 5.1 cada funcionalidad principal está dividida a su vez en dos funcionalidades secundarias como son la comunicación, la memoria y el cálculo.

- **Repartir Matriz A. Memoria:** Se recoge el tiempo relacionado con el proceso de leer del fichero y almacenar los datos en sus respectivas estructuras.
- **Repartir Matriz A. Comunicación:** Se recoge el tiempo relacionado con los envíos de las estructuras entre los diferentes procesadores.
- **Repartir Matriz B. Memoria**: Se recoge el tiempo relacionado con el proceso de leer del fichero y almacenar los datos en sus respectivas estructuras.
- **Repartir Matriz B. Comunicación:** Se recoge el tiempo relacionado con los envíos de las estructuras entre los diferentes procesadores.

- **Producto. Cálculo:** Se recoge el tiempo relacionado con la multiplicación de ambas matrices.
- **Sumar. Comunicación:** Se recoge el tiempo relacionado con los envíos de las estructuras entre los diferentes procesadores.
- **Sumar. Cálculo:** Se recoge el tiempo relacionado con el proceso de acumular las estructuras que recibe cada procesador.

Una vez analizadas estas funcionalidades se procederá a agruparlas entorno a las funcionalidades secundarias ya vistas anteriormente, quedando así:

Memoria	Comunicación	Cálculo

Tabla 5.2: Funcionalidades secundarias

Por último, la memoria y el cálculo se agruparán como no-comunicación. Dado que se trabaja con grandes cantidades de memoria y un gran número de procesadores, gracias a estas divisiones de funcionalidades se puede observar cual es el verdadero cuello de botella.

5.1 VInicial vs VOptimizada

Con motivo de encontrar una mejoría en la versión optimizada frente a la inicial, se han evaluado las funcionalidades para la matriz de dimensiones 16384 x 16384 con ambas versiones. Se ha utilizado la matriz de 16384 x 16384 dado que es la matriz más grande con la que podemos trabajar con la versión Inicial. Esto es debido a que la VInicial no predice el tamaño que va a reservar, por lo que mediante *mallocs* se reserva toda la franja, el tamaño es tan grande que no entra en las caches.

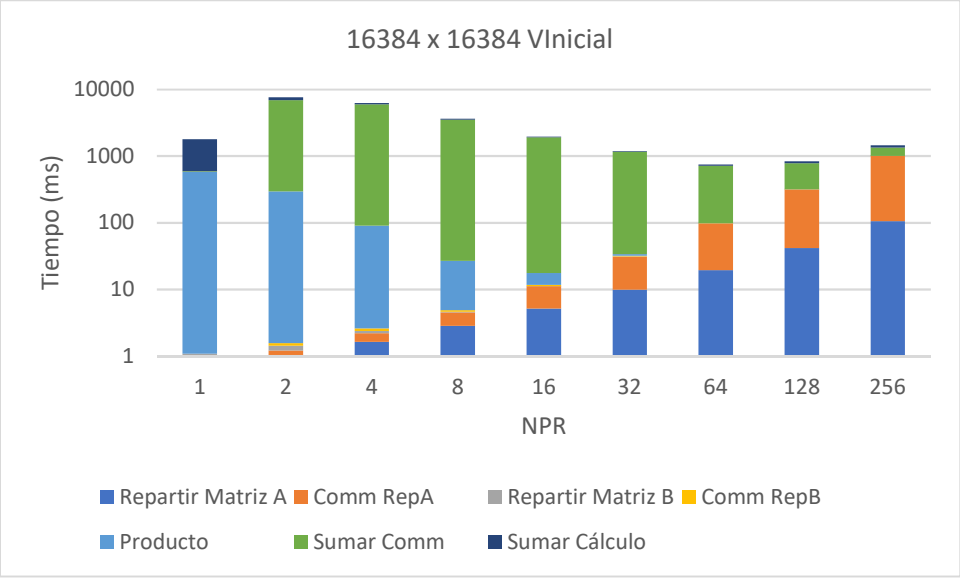

Figura 5.1: VInicial en matriz de 16384 x 16384

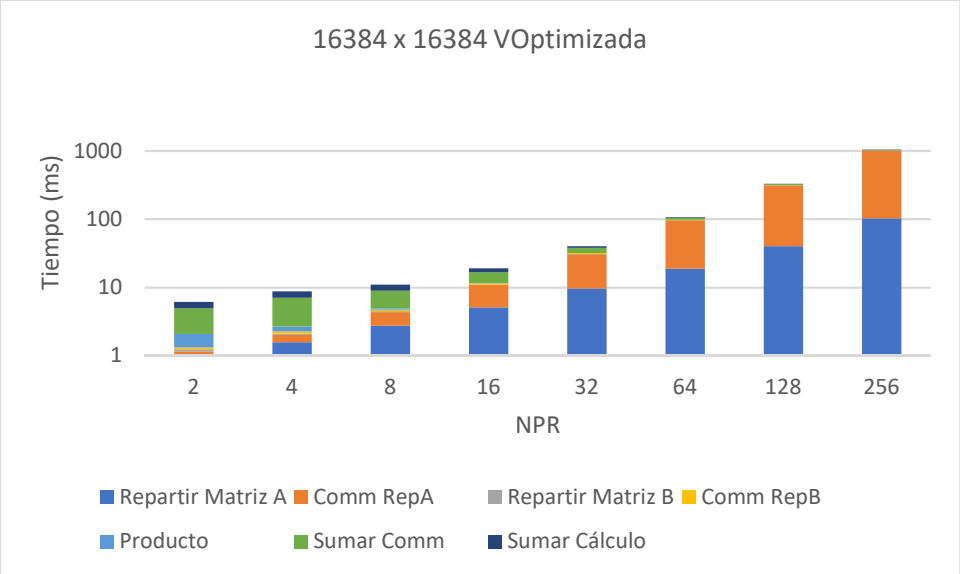

Figura 5.2: VOptimizada en matriz de 16384 x 16384

Como podemos observar en ambas figuras (Figuras 5.1 y 5.2), se nota una potencial mejoría en la versión optimizada con respecto a la original. Podemos encontrar similitudes entre ambas versiones en el producto y en los repartos de las matrices A y B. El producto disminuye a medida que aumenta el número de procesadores. Los repartos de las matrices A y B son prácticamente similares en las dos versiones debido a la *sparsidad* con la que se trabaja. En la matriz de 16384x16384 la mayoría de sus elementos son nulos. Por otro lado, la principal diferencia entre ambas versiones radica en la comunicación de la suma.

	Tiempo (ms) Sumar Comunicación		Speedup
Número de Procesadores	VInicial	VOptimizada	
1	9,773016		
2	6591,24	2,89	2280,71
4	5883,24	4,38	1343,20
8	3490,38	3,95	883.64
16	1890,29	5,20	363.52
32	1126,83	5,96	189.06
64	613,45	7,18	85.44
128	459,23	10,47	43.86
256	330,03	17,36	19.01

Tabla 5.3: Comparación Comunicación Sumar para matrices de 16384 x 16384

En la tabla 5.3 se puede observar que la diferencia entre ambas versiones es notoria. En la *VInicial* no se predice el tamaño que va a tener una franja, por lo que, en la comunicación de la suma existen muchos -1 innecesarios fruto de la sobreestimación. En la comunicación de la *VInicial*, el tiempo de ejecución disminuye a medida que el número de procesadores aumenta. Esto es debido a que, cuantos más procesadores se usen, más compactos serán los paquetes que haya que enviar, ya que la sobreestimación es menor. Por otro lado, en la comunicación de la *VOptimizada*, el tiempo de ejecución aumenta con el número de procesadores. Esto se debe a que, cuantos más procesadores haya, más envíos hay que realizar. No obstante, si observamos el *speedup*, para casos donde la sobreestimación es muy mala, el *speedup* es del orden de 2000, mientras que, en el caso de 256 procesadores, el *speedup* es del orden de 20.

Analizando los resultados previos obtenidos, se concluye que, todos los resultados que vienen a continuación serán obtenidos a través de la versión Optimizada.

5.2 Escalabilidad con el número de procesadores

Las figuras 5.3 y 5.4 muestran el tiempo de ejecución para el diferente número de procesadores de la matriz de 131072 x 131072. En la primera gráfica (Figura 5.4) se muestran hasta 256 procesadores, mientras que, en la segunda (Figura 5.3) solo hasta 64 procesadores. Esto es debido a que, posteriormente se harán comparaciones entre las ejecuciones realizadas con *core* versus las realizadas con *node* en los diferentes clústeres. El clúster de 1Gb posee 64 nodos con 4 *cores* cada uno, mientras que el clúster de *10Gb* tiene 16 nodos con 4 *cores* cada uno. Por este motivo hay que realizar las comparaciones respecto el que tenga el menor número de procesadores, como es el caso del clúster de *10Gb*.

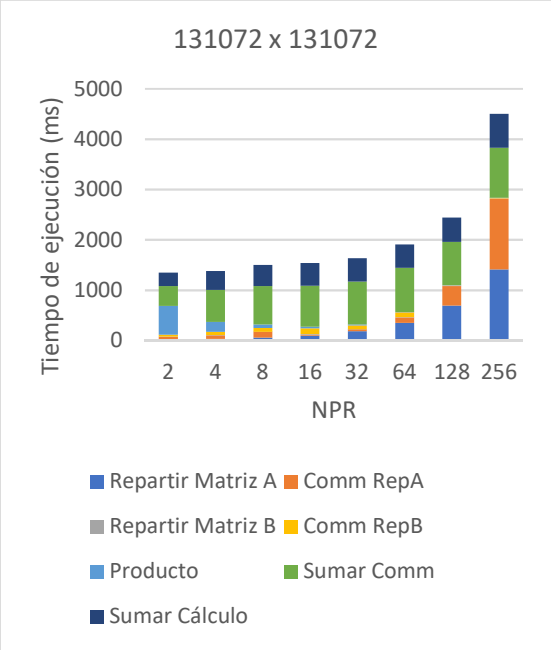

Figura 5.4: Escalabilidad con npr en 131072 x 131072

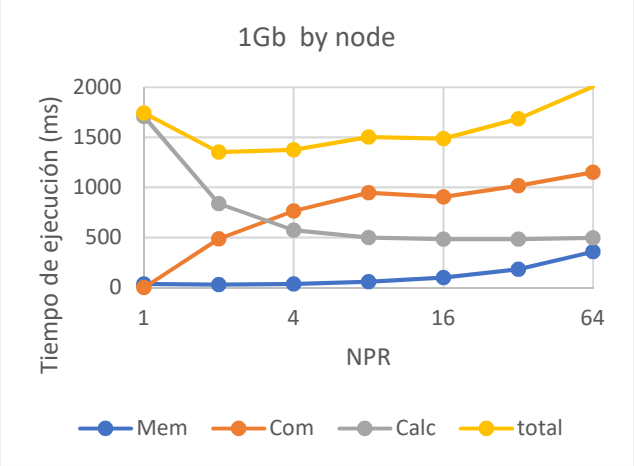

Figura 5.3: 1Gb by node. 131072 x 131072

Como se puede observar en las gráficas, la comunicación aumenta hasta llegar a 64 procesadores y luego se estabiliza. Por otro lado, el computo disminuye hasta llegar a 8 procesadores y a partir de ahí, permanece estable. Finalmente, la memoria permanece constante hasta que, a partir de 16 procesadores comienza a aumentar significativamente. Esto se debe a la cantidad de *mallocs* que se realizan para reservar las estructuras.

Analizando los resultados previos, comprobamos que los mejores resultados obtenidos son con 2 y 4 nodos.

5.3 Escalabilidad con las dimensiones de las matrices

La figura 5.6 muestra la comparación del tiempo de ejecución entre las matrices de dimensiones diferentes. Los resultados están calculados con 16 procesadores. Se han obtenido más resultados para diferente número de procesadores mostrando tendencias similares.

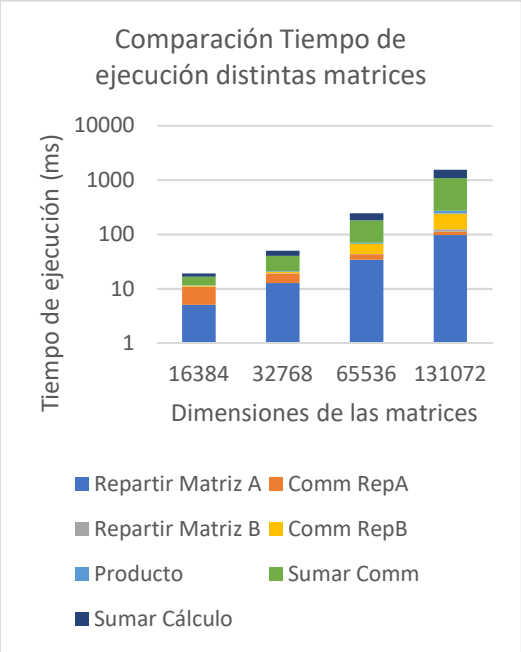

Figura 5.6: Comparación del tiempo de ejecución para distintas matrices. npr = 16

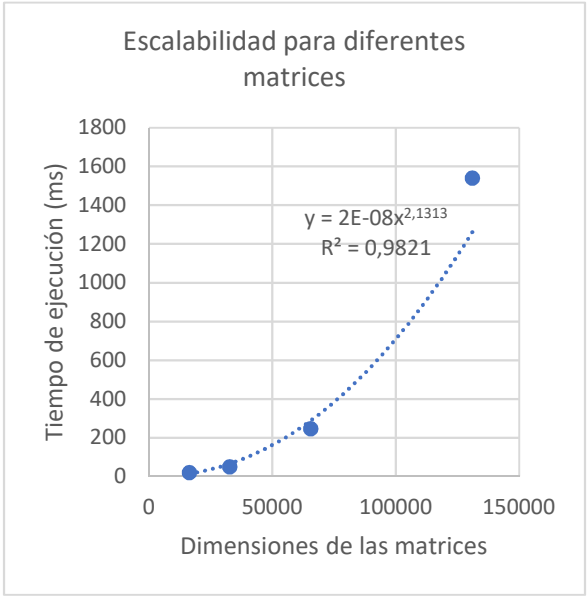

Figura 5.5: Escalabilidad para diferentes dimensiones de las matrices. npr = 16

Como se puede observar en ambas gráficas (Figuras 5.5 y 5.6), a medida que crecen las dimensiones de las matrices también crecen los tiempos de ejecución de cada una. En la segunda gráfica (Figura 5.5), podemos ver que sigue una distribución algo mayor que cuadrática. Esto se debe a la *sparsidad* con la que se trabaja, puesto que a medida que aumentan las dimensiones de las matrices, también aumentan los *nnc*.

5.4 1 Gb by node vs 1 Gb by core

En las siguientes figuras (Figuras 5.7 y 5.8) se muestran las diferencias obtenidas entre ejecuciones realizadas con *by node* y *by core*. Ambas se realizan en el mismo clúster de 1Gb hasta 64 nodos con matrices de 131072 x 131072. En las gráficas de la izquierda se analizan las funcionalidades: memoria, comunicación y cálculo. Mientras que en las gráficas de la derecha se analizan las funcionalidades: comunicación y no comunicación.

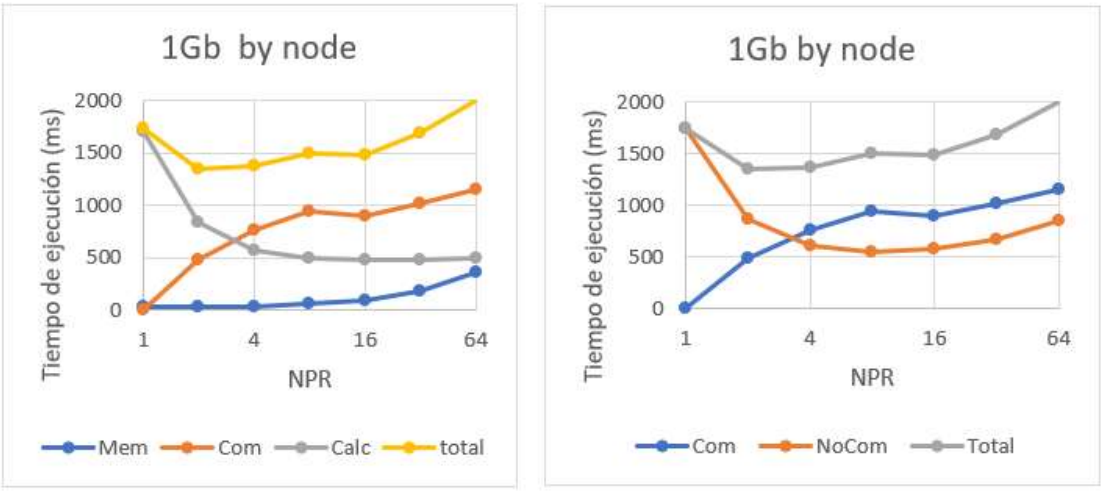

Figura 5.7: 1Gb by node para la matriz 131072 x 131072

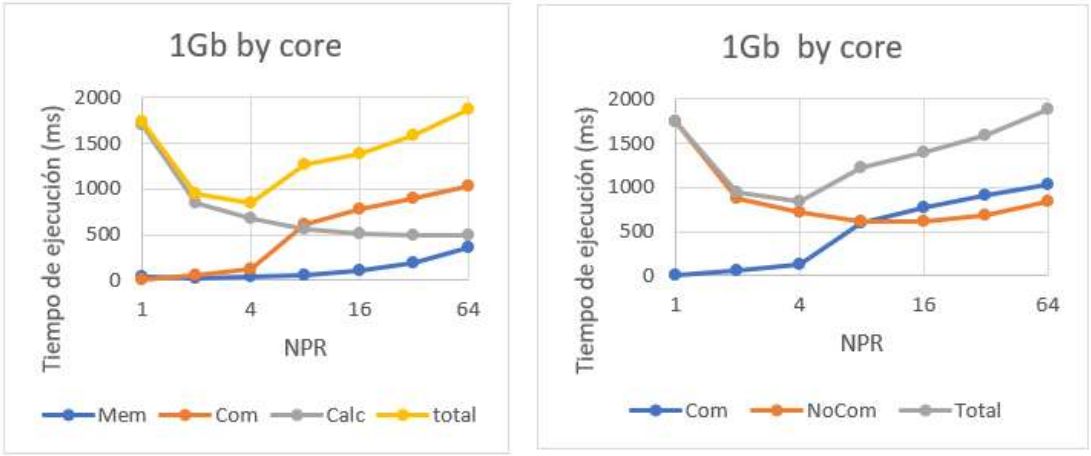

Figura 5.8: 1Gb by core para la matriz 131072 x 131072

En la figura 5.7, reparto *by node*, se puede observar que hay un mínimo en el tiempo total en 2 nodos. Mientras que en el reparto *by core* (Figura 5.8) es de 4 *cores,* es decir 1 nodo. El tiempo empleado en la memoria es el mismo para ambas versiones, este se mantiene estable hasta 16 procesadores que es cuando empieza a dispararse. El tiempo de cómputo es muy similar entre las dos versiones, disminuyendo hasta llegar a 8 procesadores y a partir de ahí permaneciendo estable. Finalmente, es en la comunicación donde radica la verdadera diferencia. Esta diferencia se debe a la asignación de los nodos y de los *cores*, como se muestra en el siguiente apartado.

En la figura 5.9 se muestra el *speedup* conseguido con las dos versiones (by core y by node) en el clúster de 1Gb.

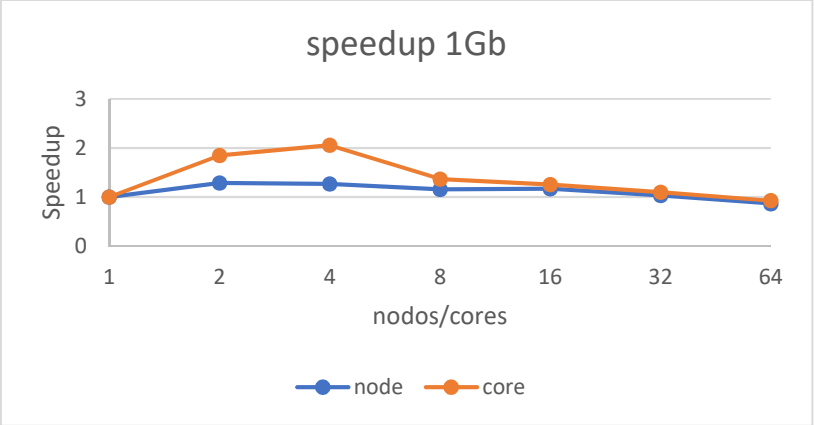

Figura 5.9: Speedup 1Gb para la matriz de 131072 x 131072

Como se puede observar en la anterior figura, el *speedup* se mantiene similar respecto a las dos versiones, excepto para 2 y 4 *cores*. El mejor *speedup* se consigue con 4 *cores*, un *speedup* de 2, casi el doble respecto al de los 4 nodos.

Como conclusión de este apartado se obtiene que el mejor tiempo de ejecución se consigue con 4 *cores*, es decir 1 nodo.

5.5 Asignación by node y by core

Como ya se ha explicado en el anterior apartado (Apartado 5.4), la principal diferencia entre las ejecuciones mediante *by node* y *by core* radica en la comunicación. Esto es debido a la asignación que se realiza de los nodos y de los *cores*. A continuación, se muestran dos ejemplos de asignaciones para 16 procesadores (Figuras 5.10 y 5.11):

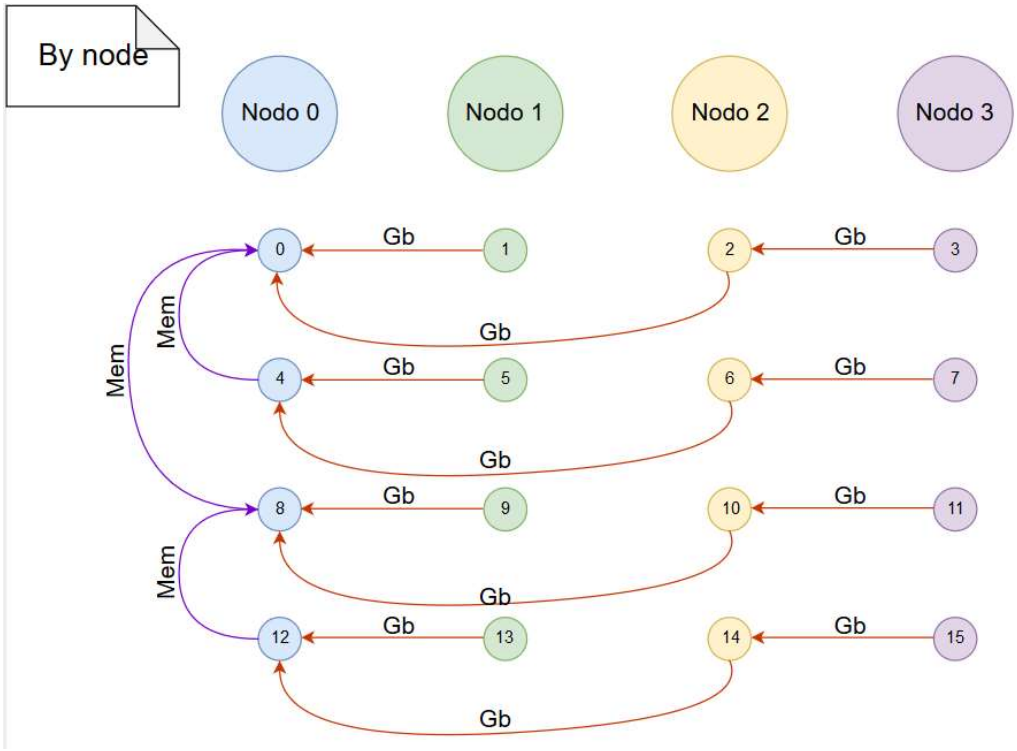

Figura 5.10: Asignación by node para npr = 16

La asignación de procesadores mediante *by node* se realiza de la siguiente manera: los procesadores se reparten equitativamente siguiendo un orden secuencial, pero alternando entre los nodos. De esta forma, estableciendo una comunicación en forma de árbol binario, los procesadores que realicen envíos a procesadores de otros nodos, lo harán mediante la red. Mientras que los procesadores que realicen envíos a procesadores del mismo nodo, lo harán mediante memoria. Resultando que, para el ejemplo expuesto (Figura 5.10), se realizan un total de 12 envíos a través de la red y 3 envíos a través de memoria.

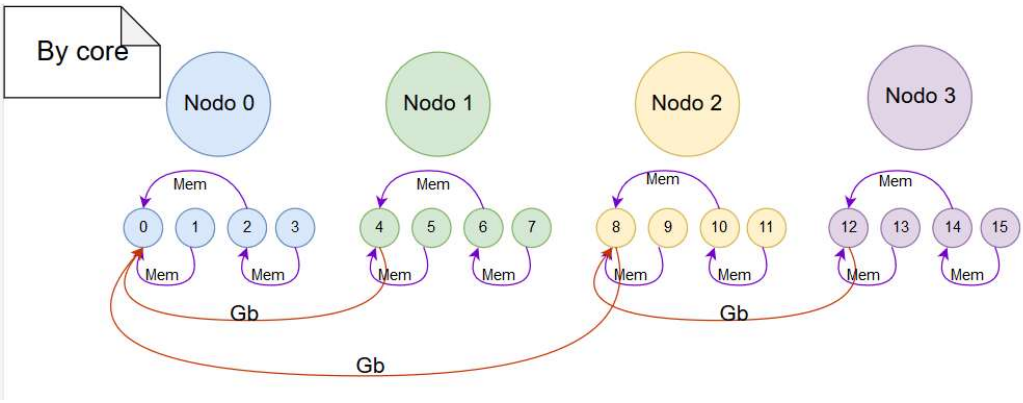

Figura 5.11: Asignación by core para npr = 16

La asignación de procesadores mediante *by core* se realiza de la siguiente manera: los procesadores se reparten equitativamente entre los nodos, siguiendo un orden secuencial. De esta forma, estableciendo una comunicación en forma de árbol binario, los procesadores que realicen envíos a procesadores de otros nodos, lo harán mediante la red. Mientras que los procesadores que realicen envíos a procesadores del mismo nodo, lo harán mediante memoria. Resultando que, para el ejemplo expuesto (Figura 5.11), se realizan un total de 12 envíos a través de memoria y 3 envíos a través de la red.

Se puede sacar en conclusión que, la diferencia en la comunicación viene dada por la asignación *by core* frente a la asignación *by node*. Haciendo uso de los ejemplos anteriores propuestos, *by core* realiza 12 envíos a través de memoria y 3 a través de la red, mientras que *by node* realiza 12 envíos a través de la red y 3 a través de memoria. *By node* emplea un tiempo de ejecución mayor debido a que satura la red con los envíos de los paquetes, mientras que *by core* realiza la gran mayoría de los envíos en memoria.

5.6 10 Gb by node vs 10 Gb by core

En las siguientes figuras (Figuras 5.12 y 5.13) se muestran las diferencias obtenidas entre ejecuciones realizadas con *by node* y *by core*. Ambas se realizan en el mismo clúster de 10Gb hasta 16 nodos con matrices de 131072 x 131072. En las gráficas de la izquierda se analizan las funcionalidades: memoria, comunicación y cálculo. Mientras que en las gráficas de la derecha se analizan las funcionalidades: comunicación y no comunicación.

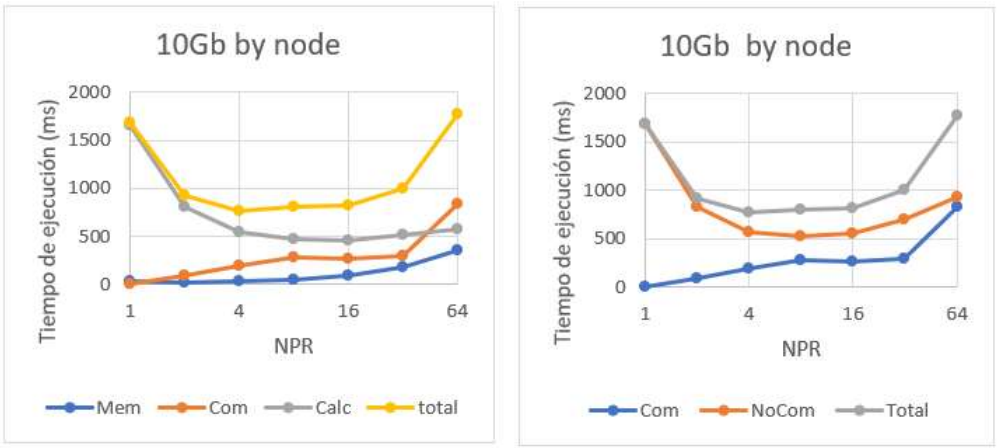

Figura 5.12: 10Gb by node para la matriz 131072 x 131072

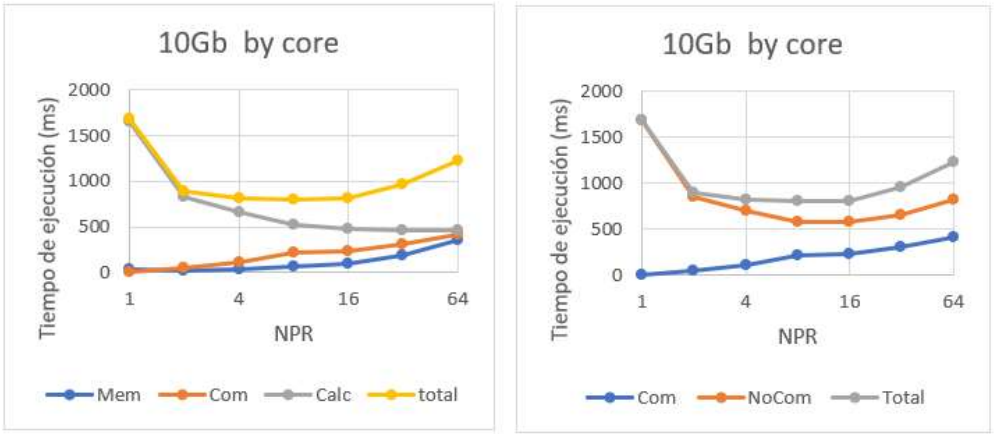

Figura 5.13: 10Gb by core para la matriz de 131072 x 131072

Como se puede observar en las anteriores gráficas, la diferencia entre ambas es casi nula. El tiempo empleado en la memoria es el mismo para ambas versiones, este se mantiene estable hasta 16 procesadores que es cuando empieza a dispararse. El tiempo de cómputo es muy similar entre las dos versiones, disminuyendo hasta llegar a 8 procesadores y a partir de ahí permaneciendo estable. Finalmente, donde se puede observar una pequeña diferencia es en la comunicación. Esta diferencia se debe a la asignación de los nodos y de los *cores*. Esta similitud entre ambas versiones es debido a que se van acercando a la velocidad de la memoria principal.

En la figura 5.14 se muestra el *speedup* conseguido con las dos versiones (by core y by node) en el clúster de *10Gb*.

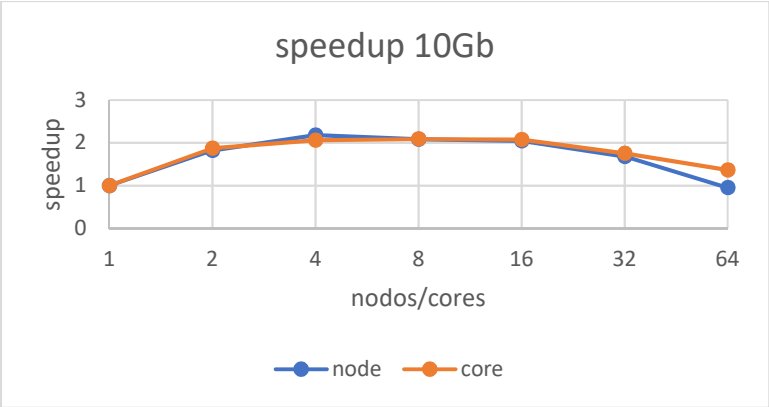

Figura 5.14: Speedup 10Gb para la matriz de 131072 x 131072

Como se puede observar en la anterior figura (Figura 5.14), el *speedup* se mantiene similar respecto a las dos versiones. El mejor *speedup* se consigue con 4 *cores* y 4 *nodos*, un *speedup* de 2.

5.7 Cálculo del tiempo para matrices de dimensiones mayores a 2^{17}

Con motivo de previsualizar el tiempo de ejecución que tardaría el producto de dos matrices de dimensiones superiores a 2^{17}, se ha calculado el tiempo de ejecución para matrices superiores a 2^{17} empleando una solución 2D.

Para ponerse en situación, se quiere calcular el tiempo de ejecución del producto de dos matrices de dimensiones 2^{20}. Puesto que el límite de memoria está destinado a una matriz de dimensiones 2^{17}, se van a formar submatrices de estas dimensiones hasta conseguir las matrices de las dimensiones que se quieren. Para poder formar una matriz de dimensiones 2^{20} con submatrices de dimensiones 2^{17}, sería necesario formar una matriz de dimensiones 2^3 y completar cada celda de esta con una submatriz de tamaño 2^{17} (Tabla 5.4).

$$2^{20}/_{2^{17}} = 2^3$$

La matriz A de dimensiones 2^{20}x 2^{20} quedaría de la siguiente manera:

2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}
2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}
2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}
2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}
2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}
2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}
2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}
2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}	2^{17}

Tabla 5.4: Matriz 2^20 x 2^20

Para poder realizar el producto entre ambas matrices (A x B = C), es necesario realizar la multiplicación elemental de matrices (Figura 5.15).

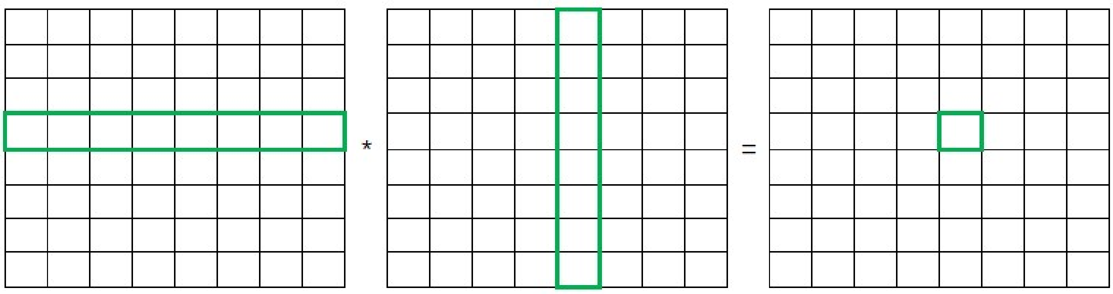

Figura 5.15: Multiplicación de matrices 8x8. Se muestra la submatriz Ci,j

Para realizar este cálculo de tiempos de ejecución, se ha tenido en cuenta el mejor tiempo de ejecución del programa de entre todos los tiempos disponibles para las matrices de dimensiones de 2^{17}. Este tiempo de ejecución se multiplica por el número de operaciones necesarias, consiguiendo así, predecir el tiempo de ejecución que un nodo emplea en realizar el producto de matrices de dimensiones 2^{20}. Cada nodo tiene que realizar la siguiente ecuación:

$$nodo_{i,j} \rightarrow C_{i,j} = \sum A_{i,k} * B_{k,j}$$

Aplicándola al ejemplo anterior (Figura 5.15), para conseguir C_{ij} cada nodo tiene que realizar el sumatorio de 8 productos, es decir 8 productos y 7 sumas. El tiempo menor en realizar el producto de matrices se ha conseguido con 4 *cores,* es decir un nodo, que es de 847,44 *ms*. Mientras que el tiempo en realizar una suma es de 953,8 *ms*. Ambos resultados los aproximaremos a *1 segundo* para facilitar los cálculos. Por lo que, el tiempo que emplea un nodo en realizar el cálculo de C_{ij} es aproximadamente de: *8*1s + 7*1s ≈* $\boxed{15s}$. Al realizarse todo en paralelo, únicamente importa el tiempo que tarde un nodo. Actualmente, las lecturas de los ficheros se realizan en formato *COO*. Como existen métodos más óptimos, como: lecturas en paralelo, lecturas codificadas etc.… no se van a tener en cuenta en el tiempo total (*por lo tanto el tiempo es el mínimo necesario*).

En este proyecto, en todo momento se ha tenido en cuenta el artículo Parallel Sparse Matrix-Matrix Multiplication: A Scalable Solution with 1-D Algorithm [1]. Por lo que se va a realizar una comparación para ver cuánto nos aproximamos a él.

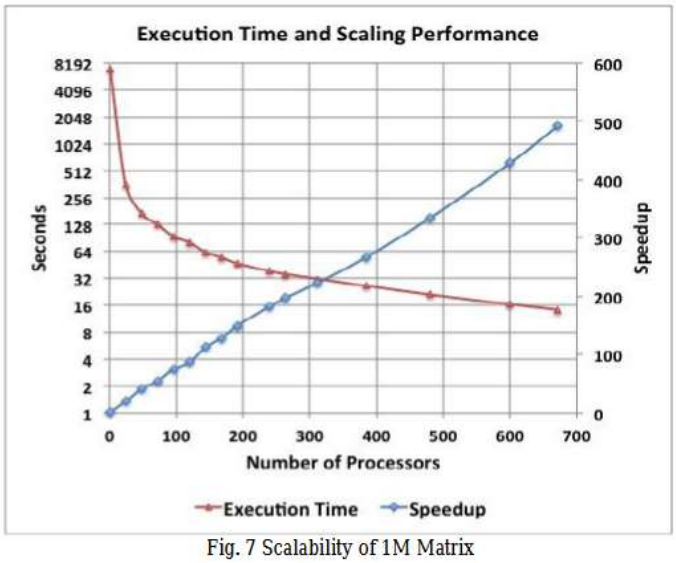

Fig. 7 Scalability of 1M Matrix

Figura 5.16: Fig. 7 Scalability of 1M Matrix [1]. Comparación entre ambas versiones

Como se puede observar en la figura 5.16, los autores del artículo han conseguido un tiempo de ejecución del orden de 40 segundos para 256 *cores*, mientras que nuestra solución emplea un tiempo de ejecución del orden de 15 segundos (sin tener en cuenta las lecturas de las matrices; ellos tampoco dejan claro si lo hacen). La comparación solo puede ser grosera debido a las diferencias entre los recursos computacionales que ellos y nosotros usamos.

5.8 Trazas generadas

A continuación, se muestran unas trazas generadas mediante un traceador elaborado en el departamento de Arquitectura de Computadores de la Facultad de Informática de San Sebastián (FISS). Las trazas muestran una ejecución con 4 procesadores para la matriz de 131082x131082.

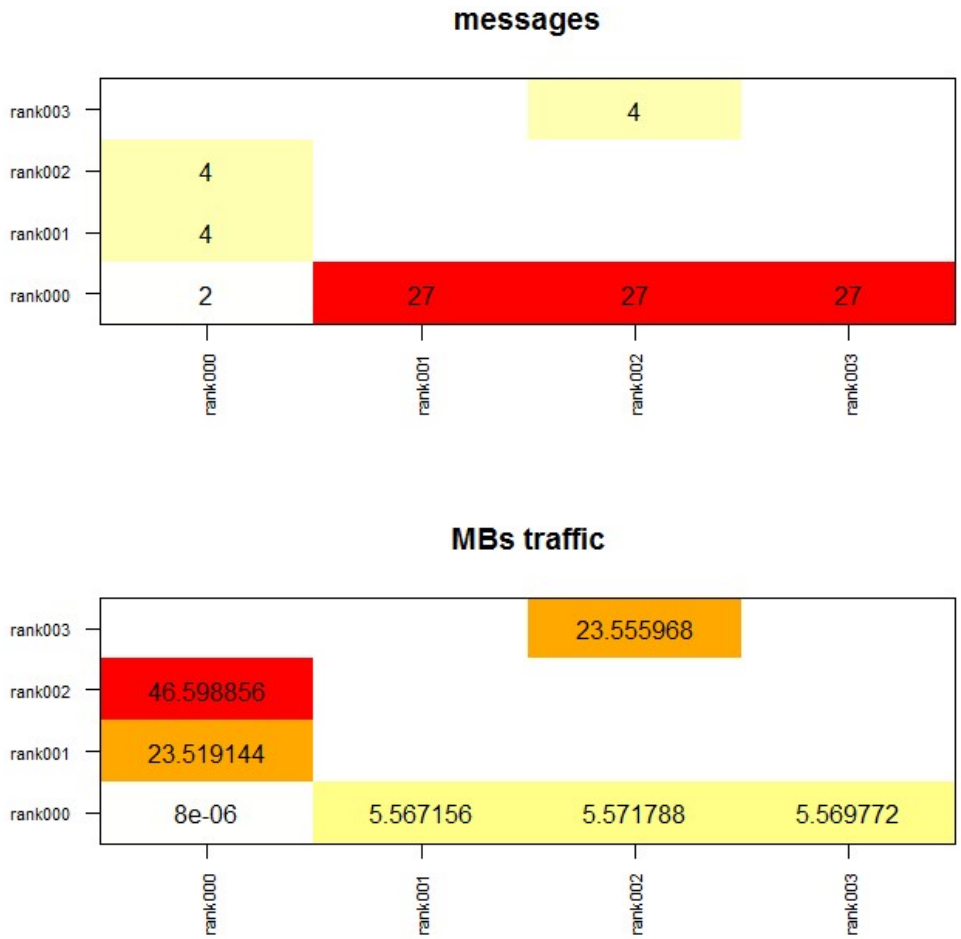

Figura 5.17: Mensajes y tráfico generado entre procesadores npr = 4

En la primera tabla de la imagen (Figura 5.16) podemos observar la cantidad de mensajes que se han mandado entre todos los procesadores. El *rank 0* ha mandado un total de 27 mensajes al resto de procesadores *(Bcast, Scatterv* y *Send*). Entre estos mensajes se incluyen las estructuras de cada bloque de la matriz A que le corresponda a cada procesador y las estructuras de la franja horizontal de la matriz B. A su vez, si nos fijamos en los mensajes que recibe el *rank 0* del resto de procesadores, podemos observar el árbol binario definido anteriormente.

Por otro lado, la segunda tabla de la imagen nos muestra el tamaño total de los mensajes que se transfieren entre procesadores. El *rank 2* es el *rank* que más tamaño envía, puesto que envía la concatenación de su C_2 con la C_3 recibida previamente. Mientras que el *rank 0* es el *rank* que más tamaño recibe, del orden de 70 *MB*.

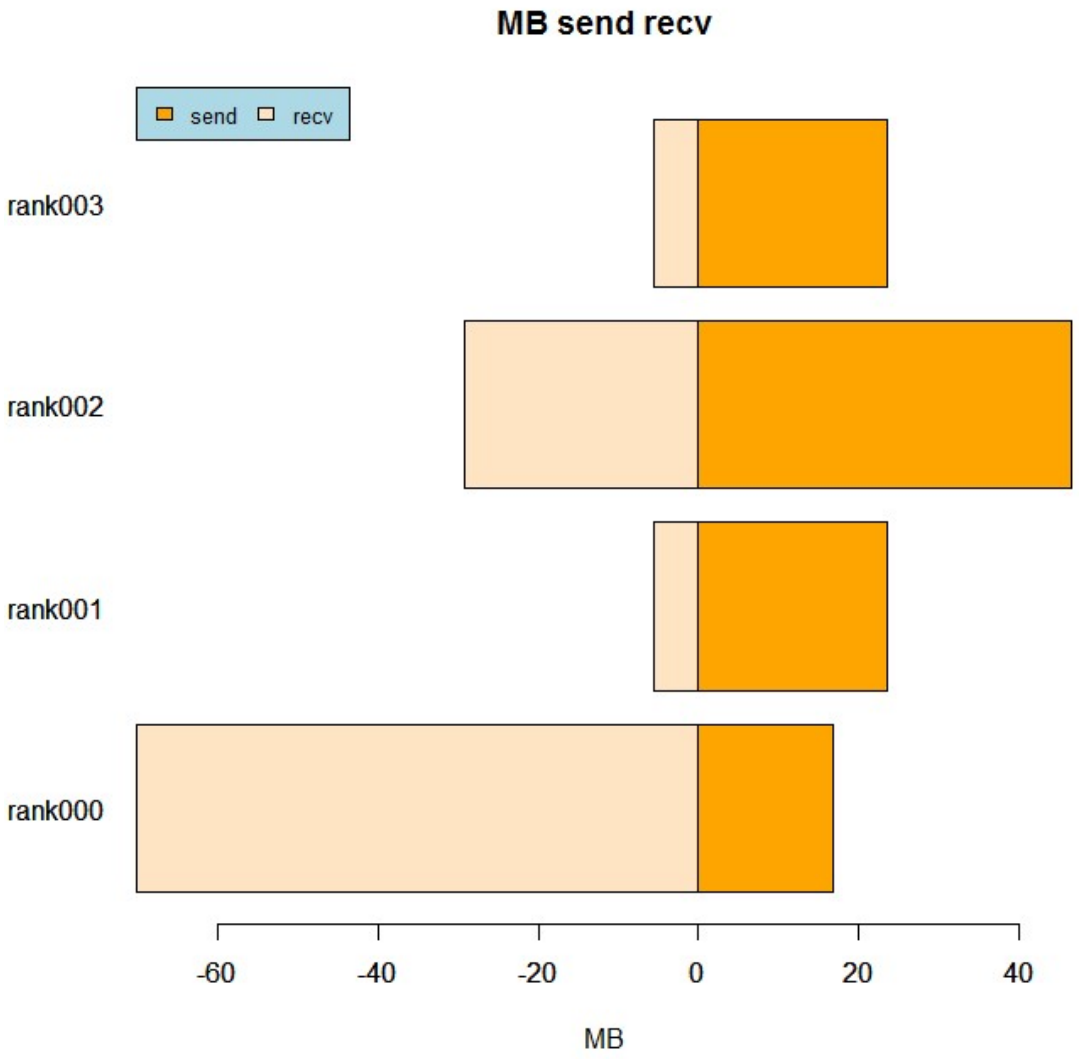

Figura 5.18: MB enviados y recibidos entre los procesadores. npr = 4

La anterior imagen (Figura 5.17) es una extensión de la Figura 5.16, donde podemos observar el tamaño en MB del total de *sends* y *recvs* que han realizado los diferentes *ranks*. El *rank 0* una vez envía los bloques de la matriz A y las franjas de la matriz B, no envía más en todo el programa y se dispone a recibir del resto de *ranks,* las C_i. Por otro lado, el resto de *ranks* únicamente recibe del *rank 0*: los bloques de la matriz A y las franjas de la matriz B. Excepto el *rank 2* que recibe la C_3 del *rank 3*. El resultado de concatenar C_2 con C_3 se envía al *rank 0*.

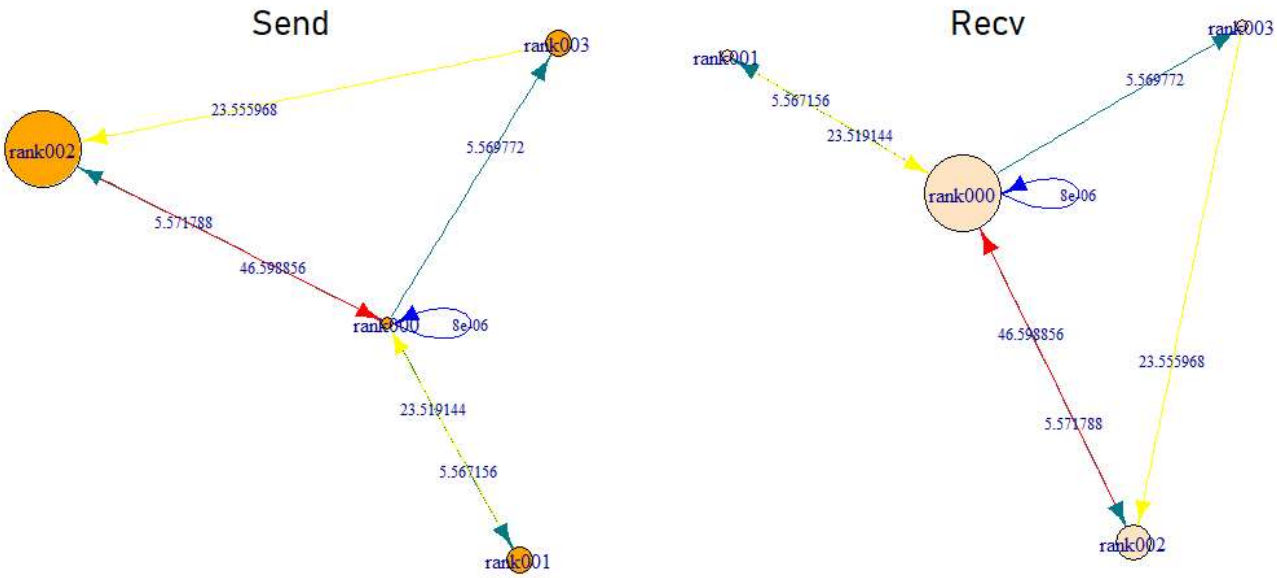

Figura 5.19: Grafos de sends y recvs entre 4 ranks

La figura (Figura 5.18) es una extensión de las anteriores figuras 5.16 y 5.17 donde muestra los grafos generados fruto de los *sends* y *recvs* realizados entre los diferentes *ranks*. En el primer grafo, se puede observar que cuanto más grande es el área del circulo que rodea a cada *rank*, mayor es la cantidad de *MB* que envía. El *rank* que más *MB* envía es el *rank 2*, mientras que el que menos *MB* envía es el *rank 0*. Por otro lado, el segundo grafo muestra lo contrario, los *MB* que recibe cada *rank*. El *rank* que más *MB* recibe es el *rank* 0, mientras que los que menos MB reciben son el *rank 1* y el *rank 3*.

6

Conclusiones

6.1 Conclusiones generales del proyecto

Tras el desarrollo del proyecto, podemos decir que se han cumplido los objetivos acordados al comienzo del mismo. Se ha propuesto un algoritmo eficiente para la multiplicación de matrices *sparse* de dimensiones del orden de 10^6 x 10^6 y probabilidad de que un elemento sea no cero de 75E-06. El algoritmo utilizado usa la descomposición de matrices en bloques basada en el número de procesadores. Nos hemos autoimpuesto la restricción de que un nodo del clúster realice como máximo el producto de matrices del orden de 10^5 x 10^5 por motivos de la memoria necesaria. Esto implica que, si queremos resolver el caso de matrices 10^6 x 10^6, se deba complementar con una solución 2D, producto de submatrices. Las matrices con las que han trabajado no tienen estructura, repartiéndose los elementos no nulos de forma aleatoria.

Se ha estudiado la escalabilidad y el paralelismo en el algoritmo utilizado, la posibilidad y las consecuencias de aplicar otros algoritmos, como el algoritmo de Cannon y se han propuesto mejoras y soluciones a los problemas detectados.

Como resumen de los resultados obtenidos se puede destacar lo siguiente. Para los clústeres con los que hemos trabajado la solución óptima es de 4 procesos en un nodo, cada proceso en un *core* del procesador del nodo. Para el clúster de *1Gb* la opción *-map by node* es la mejor. Para el clúster de *10Gb* se puede realizar el reparto de los procesos a procesadores por nodo o por core, ya que no hay diferencias significativas. Los tiempos totales del producto matricial de 10^6 x 10^6 son comparables a los del artículo de referencia. La comparación solo puede ser grosera debido a las diferencias entre los recursos computacionales que ellos y nosotros usamos, pero los tiempos avalan nuestra solución.

Pensamos que la solución propuesta puede adaptarse fácilmente a clústeres que se diferencien en el ancho de banda de la red de la comunicación, procesadores con distinto número de *cores* y diferente jerarquía de memoria.

Por otra parte, cabe destacar las dificultades obtenidas durante la etapa de desarrollo del proyecto. Los mayores quebraderos de cabeza han sido las reservas de las estructuras de memoria y las comunicaciones entre los procesadores. El principal problema de las reservas de las estructuras es que, al trabajar con matrices de grandes dimensiones, estas ocupan una gran cantidad de memoria que los procesadores no pueden manejar. Por este motivo hay que tener cuidado a la hora de sobreestimar en tamaño innecesario. Asimismo, el problema de las comunicaciones de la suma de las submatrices entre los procesadores radica en que, los tamaños de los paquetes que se envían son muy grandes. Como consecuencia de esto, hay que minimizar el número de operaciones que tiene que realizar un procesador y por ello se ha generado un algoritmo de comunicaciones basado en un árbol binario.

6.2 Futuras mejoras

Aunque el proyecto haya llegado a su fin habiendo completado todos los objetivos planteados al inicio del proyecto, siempre es posible realizar mejoras. Dado que es un proyecto en el que se desarrolla un algoritmo, a este se le pueden aplicar varias optimizaciones cuyo objetivo sea mejorar la eficiencia del programa. El mundo de la tecnología está en constante crecimiento y mejora de prestaciones computacionales, por lo que el uso de un clúster como el que hemos usado (*se acerca ya a los 10 años*) nos anima a predecir que la solución a otro más moderno no implicará más que beneficios. Gracias a esto, se podría ejecutar el programa con matrices de dimensiones de mayor orden y con un número de nodos y de *cores* mayor, mayor ancho de banda de red y mayor tamaño de memorias.

Entre las posibles mejoras, cabe destacar:

6.2.1 Aplicar el algoritmo de Cannon

Este algoritmo se podría utilizar para realizar la fase 2D del producto matricial, siendo una alternativa a la comentada en el apartado 5.7. El algoritmo de Cannon es un algoritmo distribuido para la multiplicación de matrices que utilizar una estructura de comunicación en toro (Figura 6.1).

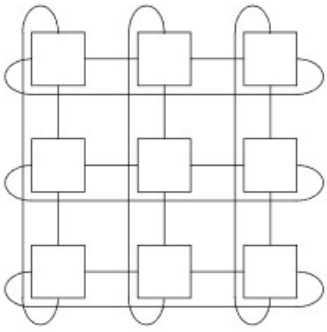

Figura 6.1: Toro de 3x3 procesadores

La distribución de los procesadores en las matrices A, B y C sigue una ordenación por filasl. Es decir, según la figura 6.1 los procesadores 0, 1 y 2 están en la primera fila.

Copiar la matriz completa a cada nodo implicaría una importante sobrecarga, tanto para las comunicaciones como para la memoria de los procesadores. Cannon proporciona a cada procesador únicamente la submatriz que necesita en cada momento (cada etapa), reduciendo los requisitos de memoria y el ancho de banda del procesador. A su vez, cada procesador envía a sus procesadores vecinos las submatrices del paso anterior. Así se evita centralizar el reparto de los datos en un único punto y se mejora el balanceo de carga.

Un proceso externo debe realizar la carga inicial de las submatrices en todos los procesadores. Para que los operandos obtenidos por cada procesador sean los adecuados es necesario desplazar las matrices originales. En la matriz A, cada submatriz se desplaza i columnas hacia la izquierda, mientras que en el caso de la matriz B, cada submatriz se desplaza j filas hacia arriba. Una vez realizado el primer paso, los procesadores almacenan localmente el resultado parcial de C_{ij}. Seguido envían la submatriz de A recibida al procesador de su izquierda y la submatriz de B al procesador superior. Del mismo modo, cada procesador recibirá los nuevos operandos de sus vecinos de abajo y derecha. Cuando disponga de ambos operandos empieza la segunda etapa, en la que realiza un nuevo producto con las submatrices recibidas. Después suma el resultado al valor local almacenado en el nodo (C_{ij}) (realizado en la etapa anterior). Así hasta completar tantos pasos como dimensiones tenga la matriz. Puede haber procesadores ejecutando etapas distintas. El único requisito para comenzar una nueva etapa es haber recibido los operandos correspondientes. A continuación, se muestra un ejemplo del algoritmo de Cannon para 3x3 procesadores:

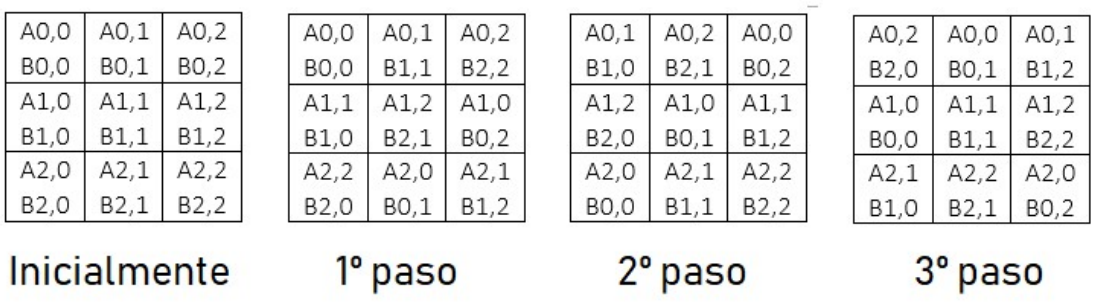

Figura 6.2: Pasos para realizar el algoritmo de Cannon con npr = 3x3

Por ejemplo, para saber las operaciones que ha realizado el procesador 4 (1,1) para calcular su submatriz de C, basta con seguir los pasos de la figura 6.2 y fijarse en los bloques que ocupan la posición (1,1) de la matriz de C:

$$C(1,1) = A(1,2) * B(2,1) + A(1,0) * B(0,1) + A(1,1) * B(1,1)$$

Tras p etapas (siendo p = 3 en el ejemplo) cada procesador contiene la submatriz final de la matriz resultado.

Las ventajas que nos ofrece el algoritmo de Cannon, son las siguientes:

El algoritmo de Cannon permite solapar la comunicación y el cómputo. Siempre que haya memoria suficiente para almacenar los nuevos bloques A y B mientras se está trabajando en los bloques actuales. Cada procesador puede enviar y recibir los bloques A y B antes de empezar la multiplicación de las matrices de esa iteración. A su vez, el algoritmo de Cannon requiere menos comunicación entre procesadores y escala mejor con el número de procesadores. El único inconveniente es que se necesita una gran cantidad de memoria para mantener las copias de los bloques en los procesadores.

6.2.2 Modificar la entrada y salida de las submatrices

En este proyecto se ha utilizado el formato *COO* para leer las matrices de los ficheros. Este formato puede ser ampliamente mejorado, por ejemplo: por el formato *CSR*, ya que este al ser un formato comprimido, ocupa menos memoria y facilita el acceso a los datos. Por otro lado, una notable mejoría seria ampliar la entrada/salida de serie a paralelo. La entrada/salida en paralelo accede a los datos del disco simultáneamente, lo que permite lograr velocidades de escritura mucho más altas y maximizar el ancho de banda. Sin embargo, cuenta con el problema de que necesita de un hardware de unas características específicas para poder realizar la entrada/salida en paralelo.

7

Bibliografía

[1] Mohammad Hoque, Md. Rezaul Raju, Christopher Tymczak, Daniel Vrinceanu, Kiran Chilakamarri. Parallel Sparse Matrix-Matrix Multiplication: A Scalable Solution with 1-D Algorithm. Published in: Journal International Journal of Computational Science and Engineering. Volume 11 Issue 4, December 2015, Pages 391-401

[2] Gonzalo R. Zenón & Ricardo G. Apaza & Walter C. Tejerina. (2014). Programación paralela: implementación de un clúster para la resolución de multiplicación de matrices. Publicado el 27 de feb. de 2014 en SlideShare.

[3] Aydin Buluc, John R.Gilbert. (2010). Highly Parallel Sparse Matrix-Matrix Multiplication. Published in: Journal Computing Research Repository, arXiv.

[4] David Aparco Cárdenas. (Fecha). Algoritmos Paralelos para la Multiplicación de Matrices. Publicado en Scribd.

[5] José Soto Mejía, Guillermo Roberto Solarte Martínez, Luis Eduardo Muñoz Guerrero. (2013). Matrices Dispersas Descripción y Aplicaciones.

[6] David Villa Alises, Francisco Moya Fernández. (2014). Algoritmo de Cannon. Tesis, Sistemas Distribuidos, Escuela Superior de Informática, Universidad de Castilla-La Mancha.

[7] Michael J. Quinn. (2003) Parallel Programming in C with MPI and OpenMP. Chapter 11, pages 273-289.

[8] Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/

[9] Get started with R igraph. https://igraph.org/r/

[10] MATLAB Sparse Matrix https://www.mathworks.com/help/matlab/ref/sparse.html

[11] Sparse Matrix. https://en.wikipedia.org/wiki/Sparse_matrix

8

Anexos

8.1 Código del proyecto

A continuación, se adjunta el código de las funciones que componen el programa final.

8.1.1 PreLeer

```
void preLeer (FILE *fichero,int *primera,int *segunda,float *tercera,
              int *tamfranja, int numfilas,int *despl,int numvalA,int npr){
    int cont=0;
    int i,j,max,pos;
    for (i=0;i<npr;i++){
        max = (i==(npr-1))? numfilas : despl[i+1];
        if (cont==0){
            fscanf(fichero, "%d %d %f",primera,segunda,tercera);
        }
        pos = 0;
        for (j=despl[i];j<max;j++){
            if ((*primera)>j) continue;
            for (;;){
                pos++;
                cont++;
                if (cont < numvalA)
                {
                    fscanf(fichero, "%d %d %f",primera,segunda,tercera);
                }
                else
                    break;
                if ((*primera)>j) break;
            }
            if (cont >= numvalA)break;
        }
        if (cont >= numvalA)(*primera)=numfilas;
        tamfranja[i]=pos;
    }


}
```

8.1.2 Leer

```
45  int leer (FILE *fichero,int *primera,int *segunda,float *tercera,
46            int *prA_J,float *prA_V,int *prA_I,int i,
47              int npr,int *cont,int numvalA,int *ultimo,
48              int numfilas,int *despl,int *tam){
49
50      int pos=0;
51      int k,j,max;
52      int ini;
53
54      ini = (*cont);
55
56      max = (i==(npr-1))? numfilas : despl[i+1];
57      if ((*cont)==0){
58          fscanf(fichero, "%d %d %f",primera,segunda,tercera);
59      }
60
61      for (j=despl[i];j<max;j++)
62      {
63         if ((*primera)>j)
64         {
65          prA_I[j-despl[i]]=(*cont-ini); continue;
66         }
67         prA_I[j-despl[i]]=(*cont-ini);
68
69         for (;;)
70         {
71          prA_J[pos] = (*segunda);
72          prA_V[pos] = (*tercera);
73          pos++;  (*cont)++;
74          if ((*cont) < numvalA)
75          {
76              fscanf(fichero, "%d %d %f",primera,segunda,tercera);
77          }
78          else
79              break;
80          if ((*primera)>j) break;
81         }
82         if ((*cont) >= numvalA)break;
83      }
84      prA_I[max-despl[i]]=(*cont-ini);
85      if ((*cont) >= numvalA)(*primera)=numfilas;
86      return pos;
87  }
```

8.1.3 ObtenerBloque

```
89  int obtenerBloque(int *prA_J,float *prA_V,int *prA_I,
90                    int j,int npr,int pos,
91                    int numfilas,int *despl,int *tam,
92                    int *bloque_I,int *bloque_J,float *bloque_V){
93      int max,min,k,x;
94      int count = 0;
95      int pid;
96      pid =  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
97
98      max = (j==(npr-1))? numfilas : despl[j+1];
99      min = despl[j];
100
101     bloque_I[0]=count;
102
103     for (k=1;k<tam[j]+1;k++){
104         if(prA_I[k] != prA_I[k-1]){
105             for(x=prA_I[k-1];x<prA_I[k];x++){
106                 if(prA_J[x] < max && prA_J[x] >= min){
107                     bloque_J[count] = prA_J[x];
108                     bloque_V[count] = prA_V[x];
109                     count++;
110                 }
111             }
112             bloque_I[k]=count;
113         }else{
114             bloque_I[k]=count;
115         }
116     }
117     return count;
118 }
```

8.1.4 Unir

```
120  void unir (int *recibe_I,int *recibe_J,float *recibe_V,int contRecibe,
121             int tamano,int i,int j,int **actual_I,int **actual_J,
122             float **actual_V,int *contAcum,int *tamAcum){
123      int tamRecibe,contActual,tamActual,x;
124      int *final_I,*final_J;
125      float *final_V;
126      int pid;
127      pid = MPI_Comm_rank(MPI_COMM_WORLD, &pid);
128      tamRecibe = tamano+1;
129
130      if (i == 0){
131          final_I = calloc(tamRecibe , sizeof(int));
132          if (contRecibe>0)
133            final_J = malloc(contRecibe * sizeof(int));
134          else
135            final_J = NULL;
136          if (contRecibe>0)
137            final_V = malloc(contRecibe * sizeof(float));
138          else
139            final_V = NULL;
140          final_I[0:tamRecibe:1] = recibe_I[0:tamRecibe:1];
141          if (contRecibe>0)
142              final_J[0:contRecibe:1] = recibe_J[0:contRecibe:1];
143          if (contRecibe>0)
144              final_V[0:contRecibe:1] = recibe_V[0:contRecibe:1];
145
146          (*tamAcum) = tamRecibe;
147          (*contAcum) = contRecibe;
148
149      }else{
```

```
150
151         contActual = (*contAcum);
152         tamActual = (*tamAcum);
153         (*contAcum) = contActual + contRecibe;
154         (*tamAcum) = ((*tamAcum)-1) + tamRecibe;
155
156         final_I = calloc((*tamAcum), sizeof(int));
157         if ((*contAcum)>0)
158             final_J = malloc((*contAcum) * sizeof(int));
159         else
160             final_J = NULL;
161         if ((*contAcum)>0)
162             final_V = malloc((*contAcum) * sizeof(float));
163         else
164             final_V = NULL;
165
166         for(x=0;x<tamActual;x++){
167             final_I[x] = (*actual_I)[x];
168         }
169         for (x=tamActual-1;x<(*tamAcum);x++){
170             final_I[x] = recibe_I[x-(tamActual-1)]+(*actual_I)[tamActual-1];
171         }
172
173         for(x=0;x<contActual;x++){
174             final_J[x] = (*actual_J)[x];
175             final_V[x] = (*actual_V)[x];
176         }
177         for (x=contActual;x<(*contAcum);x++){
178             final_J[x] = recibe_J[x-contActual];
179             final_V[x] = recibe_V[x-contActual];
180         }
181
182         free(*actual_I);
183         if ((*contAcum) >0)
184          {
185           free(*actual_J);
186           free(*actual_V);
187          }
188     }
```

```
189
190     (*actual_I) = calloc((*tamAcum), sizeof(int));
191     if ((*contAcum)>0)
192         (*actual_J) = malloc((*contAcum) * sizeof(int));
193     else
194         (*actual_J) = NULL;
195     if ((*contAcum)>0)
196         (*actual_V) = malloc((*contAcum) * sizeof(float));
197     else
198         (*actual_V) = NULL;
199     //Volcar Final en Actual
200     (*actual_I)[0:(*tamAcum):1] = final_I[0:(*tamAcum):1];
201     if ((*contAcum)>0)
202         (*actual_J)[0:(*contAcum):1] = final_J[0:(*contAcum):1];
203     if ((*contAcum)>0)
204         (*actual_V)[0:(*contAcum):1] = final_V[0:(*contAcum):1];
205
206     free(final_I);
207     if ((*contAcum) >0)
208     {
209         free(final_J);
210         free(final_V);
211     }
212
213 }
```

8.1.5 Repartir

```
215 void repartir(int *prA_J,float *prA_V,int *prA_I,int npr,int pos,
216                int numcolA,int j,int i,int pid,int root,int *despl,
217                int *tam,MPI_Comm comm,MPI_Status info,
218                int **actual_I,int **actual_J,float **actual_V,
219                int *contAcum, int *tamAcum){
220     int count,x;
221     int k,tamano;
222     int *bloque_I,*bloque_J;
223     float *bloque_V;
224     double t0,t1;
225     MPI_Request *req;
226     MPI_Status *status;
227 
228     req = malloc(5*sizeof(MPI_Request));
229     status  = malloc(5*sizeof(MPI_Status));
230     tamano=tam[j];
231     if (pid == root){
232         bloque_I = calloc((tam[j] + 1), sizeof(int));
233         if (pos==0) bloque_J=NULL;
234         else
235           bloque_J = malloc(pos * sizeof(int));
236         if (pos==0) bloque_V=NULL;
237         else
238          bloque_V = malloc(pos * sizeof(float));
239         count = obtenerBloque(prA_J,prA_V,prA_I,j,npr,pos,
240                               numcolA,despl,tam,bloque_I,bloque_J,
241                               bloque_V);
242         if (j == root){
243 
244             unir (bloque_I,bloque_J,bloque_V,count,tamano,i,j,
245                   actual_I,actual_J,actual_V,contAcum,tamAcum);
246             free(bloque_I);
247             if (pos>0)
248             {
249              if (bloque_J != NULL) free(bloque_J);
250              if (bloque_V != NULL) free(bloque_V);
251             }
```

```
252         }else{
253             t0 = MPI_Wtime ();
254             MPI_Isend(&tamano,1,MPI_INT,j,0,comm,&req[0]);
255             MPI_Isend(bloque_I,tamano+1,MPI_INT,j,0,comm,&req[1]);
256 
257             MPI_Isend(&count,1,MPI_INT,j,0,comm,&req[2]);
258             if (count>0)
259                 MPI_Isend(bloque_J,count,MPI_INT,j,0,comm,&req[3]);
260 
261             if (count>0)
262                 MPI_Isend(bloque_V,count,MPI_FLOAT,j,0,comm,&req[4]);
263 
264             MPI_Waitall ((count>0)?5:3, req,status);
265             free(bloque_I);
266             if (count>0)
267             {
268               if (bloque_J != NULL) free(bloque_J);
269               if (bloque_V != NULL) free(bloque_V);
270             }
271 
272             t1 = MPI_Wtime ();
273             sumaA+=(t1-t0);
274         }
```

```
275     }else{
276         if (j == pid){
277             MPI_Recv(&tamano, 1, MPI_INT, 0, 0, comm,MPI_STATUS_IGNORE);
278             bloque_I = calloc((tamano+1), sizeof(int));
279             MPI_Recv(bloque_I, tamano+1, MPI_INT, 0, 0, comm,MPI_STATUS_IGNORE);
280
281             MPI_Recv(&count, 1, MPI_INT, 0, 0, comm,MPI_STATUS_IGNORE);
282             if (count>0){
283               bloque_J = malloc(count * sizeof(int));
284               MPI_Recv(bloque_J, count, MPI_INT, 0, 0, comm,MPI_STATUS_IGNORE);
285             }
286             else
287               bloque_J = NULL;
288
289             if (count>0){
290                 bloque_V = malloc(count * sizeof(float));
291                 MPI_Recv(bloque_V, count, MPI_FLOAT, 0, 0, comm,MPI_STATUS_IGNORE);
292             }
293             else
294               bloque_V = NULL;
295
296             unir (bloque_I,bloque_J,bloque_V,count,tamano,i,j,actual_I,actual_J,actual_V,contAcum,tamAcum);
297             free(bloque_I);
298             if (count>0)
299             {
300               free(bloque_J);
301               free(bloque_V);
302             }
303         }
304     }
305
306
307 }
```

8.1.6 MatrizB

```
void matrizB (FILE *ficheroB,int **mi_prB_I,int **mi_prB_J,float **mi_prB_V,
              int npr,MPI_Comm comm,int numfilasB,int numcolB,int numvalB,
              int pid, int root,
              int *tamb,int *desplb,int **vecpos,char **argv){
    int pos,cont,ultimo,tamano,i,j;
    int primera,segunda;
    float tercera;
    int *prB_I,*prB_J,*tamfranja;
    float *prB_V;
    double t0,t1,t2,t3,t4,t5,sum=0,suma=0;
    MPI_Request *req;
    MPI_Status *status;
    req = malloc(5*sizeof(MPI_Request));
    status  = malloc(5*sizeof(MPI_Status));
    tamfranja = malloc (npr * sizeof(int));
    cont=0;
    ultimo=0;
    if (pid == root){
        preLeer(ficheroB,&primera,&segunda,&tercera,tamfranja,numcolB,
                desplb,numvalB,npr);
        fclose(ficheroB);
        ficheroB = fopen(argv[2],"r");
        fscanf(ficheroB, "%d",&numfilasB);
        fscanf(ficheroB, "%d",&numcolB);
        fscanf(ficheroB, "%d",&numvalB);
        t0 = MPI_Wtime ();
        for (i=0; i<npr;i++){
            prB_I = calloc(numcolB , sizeof(int));
            if (tamfranja[i]>0){
                prB_J = malloc(tamfranja[i] * sizeof(int));
                prB_V = malloc(tamfranja[i] * sizeof(float));
            }else{
                prB_J = NULL;
                prB_V = NULL;
            }
            t2 = MPI_Wtime ();
            pos = leer(ficheroB,&primera,&segunda,&tercera,prB_J,prB_V,
                       prB_I,i,npr,&cont,numvalB,&ultimo,numfilasB,
                       desplb,tamb);
            t3 = MPI_Wtime ();
            sum += (t3-t2);
```

```
            if (i!=root){
                t4 = MPI_Wtime ();
                MPI_Isend(&tamb[i],1,MPI_INT,i,1,comm,&req[0]);
                MPI_Isend(prB_I,tamb[i]+1,MPI_INT,i,0,comm,&req[1]);
                MPI_Isend(&pos,1,MPI_INT,i,0,comm,&req[2]);
                if (pos>0)
                  MPI_Isend(prB_J,pos,MPI_INT,i,0,comm,&req[3]);
                if (pos>0)
                    MPI_Isend(prB_V,pos,MPI_FLOAT,i,0,comm,&req[4]);
                MPI_Waitall ((pos>0)?5:3, req,status);
                t5 = MPI_Wtime ();
                suma += (t5-t4);
            }else{
                (*mi_prB_I) = calloc((tamb[i]+1), sizeof(int));
                if (pos>0)
                  (*mi_prB_J) = malloc(pos * sizeof(int));
                else
                  (*mi_prB_J)=NULL;
                if (pos>0)
                    (*mi_prB_V) = malloc(pos * sizeof(float));
                else
                    (*mi_prB_V)=NULL;
                (*mi_prB_I)[0:tamb[i]+1:1] = prB_I[0:tamb[i]+1:1];
                if (pos>0)
                  (*mi_prB_J)[0:pos:1] = prB_J[0:pos:1];
                if (pos>0)
                  (*mi_prB_V)[0:pos:1] = prB_V[0:pos:1];
                (*vecpos)[pid]=pos;
            }
            if (pid == root){
                free (prB_I);
                if (tamfranja[i] > 0){
                    if (prB_J != NULL) free(prB_J);
                    if (prB_V != NULL) free(prB_V);
                }
            }
        }
        fclose(ficheroB);
        free (tamfranja);
```

```
    }else{
        MPI_Recv(&tamb[pid], 1, MPI_INT, 0, 1, comm,MPI_STATUS_IGNORE);
        (*mi_prB_I) = calloc((tamb[pid]+1), sizeof(int));
        MPI_Recv((*mi_prB_I),tamb[pid]+1,MPI_INT,0,0,comm,MPI_STATUS_IGNORE);
        MPI_Recv(&pos, 1,MPI_INT,0,0,comm,MPI_STATUS_IGNORE);
        if (pos>0)
        {
          (*mi_prB_J) = malloc(pos * sizeof(int));
          MPI_Recv((*mi_prB_J),pos,MPI_INT,0,0,comm,MPI_STATUS_IGNORE);
        }
        else
          (*mi_prB_J)=NULL;

        if (pos>0)
        {
         (*mi_prB_V) = malloc(pos * sizeof(float));
          MPI_Recv((*mi_prB_V), pos, MPI_FLOAT, 0, 0, comm,MPI_STATUS_IGNORE);
        }
        else
          (*mi_prB_V)=NULL;
        (*vecpos)[pid]=pos;
    }
    t1 = MPI_Wtime ();
    if (pid == root)
    {fprintf (stderr,"time Repartir Matriz B %1.6f comm %1.6f\n",
    (t1-t0-sum-suma)*1000,suma*1000);
     sumatotal+=(t1-t0-sum)*1000;
    }

}
```

8.1.7 Ordenar

```
420 void ordenar(int **subC_I,int **subC_J,float **subC_V,int iniC,int finC,int valor,int colB){
421     int auxJ,auxV,i;
422     auxJ=colB;
423     auxV = valor;
424     for (i=iniC;i<finC;i++){
425         if (auxJ == (*subC_J)[i]){
426             (*subC_V)[i] = auxV + (*subC_V)[i];
427             break;
428         }
429         if (auxJ < (*subC_J)[i]){
430             colB = auxJ;
431             valor = auxV;
432             auxJ = (*subC_J)[i];
433             auxV = (*subC_V)[i];
434             (*subC_J)[i] = colB;
435             (*subC_V)[i] = valor;
436         }else{
437             if ((*subC_J)[i] == -1){
438                 (*subC_J)[i] = auxJ;
439                 (*subC_V)[i] = auxV;
440                 break;
441             }
442         }
443     }
444 }
```

8.1.8 Producto

```
446  int producto(int *actual_I,int *actual_J,float *actual_V,int *mi_prB_I,int *mi_prB_J,
447               float *mi_prB_V,int **subC_I,int **subC_J,float **subC_V,int numfilas,int pid,
448               int tamAcum,int *tamb,int *despl){
449
450      int elemA,elemB,dim;
451      int iniA,finA,iniB,finB,iniC,finC,i,j,k;
452      int valorA,colA,colB,valor;
453      dim = 0;
454      (*subC_I) = calloc(tamAcum, sizeof(int));
455      (*subC_I)[0] = 0;
456      for (i=0;i<tamAcum-1;i++){
457          elemA = actual_I[i+1]-actual_I[i];
458          if (elemA ==0){
459              (*subC_I)[i+1] = dim;
460              continue;
461          }
462          for (k=actual_I[i]; k<actual_I[i+1]; k++){
463              colA = actual_J[k]-despl[pid];
464              elemB = mi_prB_I[colA+1]-mi_prB_I[colA];
465              if (elemB ==0){
466                  (*subC_I)[i+1] = dim;
467                  continue;
468              }
469              dim = dim + elemB;
470              (*subC_I)[i+1] = dim;
471          }
472      }
473      (*subC_J) = malloc(dim * sizeof(int));
474      (*subC_V) = calloc(dim , sizeof(float));
475      for (i=0;i<dim;i++){
476          (*subC_J)[i]= -1;
477      }
478      for (i=0;i<tamAcum-1;i++){
479        elemA = actual_I[i+1]-actual_I[i];
480        if (elemA ==0) continue;
481        iniA = actual_I[i];
482        finA = actual_I[i+1];
483        iniC = (*subC_I)[i];
484        finC = (*subC_I)[i+1];
```

```
485          for (k=iniA; k<finA; k++)
486          {
487              valorA = actual_V[k];
488              colA = actual_J[k]-despl[pid];
489              elemB = mi_prB_I[colA+1]-mi_prB_I[colA];
490              if (elemB ==0) continue;
491              iniB = mi_prB_I[colA];
492              finB = mi_prB_I[colA+1];
493              for (j=iniB; j< finB; j++){
494                    valor = valorA*mi_prB_V[j];
495                    colB = mi_prB_J[j];
496                    if (finC > iniC){
497                        ordenar(subC_I,subC_J,subC_V,iniC,finC,valor,colB);
498                    }
499              }
500          }
501      }
502      return dim;
503  }
```

8.1.9 AcumularFila

```
void acumularfila (int *subC_I,int *subC_J,float *subC_V,int **Ac_I,int **Ac_J,
                   float **Ac_V,int *C_I,int *C_J,float *C_V,int z,int numfilas){

    int indC,indAc,finAc,indSubC,finSubC,finC,i;
    indC = C_I[z];
    finC = C_I[z+1];
    indSubC = subC_I[z];
    finSubC = subC_I[z+1];
    indAc = (*Ac_I)[z];
    finAc = (*Ac_I)[z+1];
    while ( indC < finC ){
        if (finSubC-indSubC==0){
            if (finAc-indAc==0) break;
            C_J[indC] = (*Ac_J)[indAc];
            C_V[indC] = (*Ac_V)[indAc];
            indAc++;
            indC++;
        }else{
            if (finAc-indAc==0){
                C_J[indC] = subC_J[indSubC];
                C_V[indC] = subC_V[indSubC];
                indSubC++;
                indC++;
            }else{
                if ((*Ac_J)[indAc] == -1){
                    if (subC_J[indSubC] == -1) break;
                    C_J[indC] = subC_J[indSubC];
                    C_V[indC] = subC_V[indSubC];
                    indSubC++;
                    indC++;
                }else{
                    if (subC_J[indSubC] == -1){
                        C_J[indC] = (*Ac_J)[indAc];
                        C_V[indC] = (*Ac_V)[indAc];
                        indAc++;
                        indC++;
                    }else{
                        if (subC_J[indSubC] > (*Ac_J)[indAc]){
                            C_J[indC] = (*Ac_J)[indAc];
                            C_V[indC] = (*Ac_V)[indAc];
                            indAc++;
                            indC++;
                        }else if(subC_J[indSubC] < (*Ac_J)[indAc]){
                            C_J[indC] = subC_J[indSubC];
                            C_V[indC] = subC_V[indSubC];
                            indSubC++;
                            indC++;
                        }else{
                            C_J[indC] = subC_J[indSubC];
                            C_V[indC] = subC_V[indSubC] + (*Ac_V)[indAc];
                            indSubC++;
                            indC++;
                            indAc++;
                        }
                    }
                    if ((C_J[indC-1] == numfilas-1)&& (indC<finC)){
                        C_J[indC] = -1;
                        C_V[indC] = 0;
                        indC++;
                    }
                }
            }
        }
    }
}
```

8.1.10 Acumular

```
void acumular (int *subC_I,int *subC_J,float *subC_V,int **Ac_I,int **Ac_J,
               float **Ac_V,int i,int dim,int numfilas){
    int *C_I,*C_J;
    float *C_V;
    int z,contAc,contSubC,contC,j,k;
    C_I = calloc (numfilas+1, sizeof(int));
    C_I[0] = 0;
    if (i==0){
        (*Ac_I) = calloc(numfilas+1, sizeof(int));
        (*Ac_I)[0:numfilas+1:1] = subC_I[0:numfilas+1:1];
        if(dim != 0){
            (*Ac_J) = malloc(dim * sizeof(int));
            (*Ac_V) = malloc(dim * sizeof(float));
            (*Ac_J)[0:dim:1] = subC_J[0:dim:1];
            (*Ac_V)[0:dim:1] = subC_V[0:dim:1];
        }
    }else{
        for (j=0;j<numfilas;j++){
            contSubC = 0;
            contAc = 0;
            for (k=subC_I[j];k<subC_I[j+1];k++){
                if (subC_J[k]!=-1){
                    contSubC++;
                }
            }
            for (k=(*Ac_I)[j];k<(*Ac_I)[j+1];k++){
                if ((*Ac_J)[k]!=-1){
                    contAc++;
                }
            }
            contC = contSubC + contAc;

            C_I[j+1] = C_I[j] + min(contC,numfilas);
        }
        if (C_I[numfilas]>0){
            C_J = malloc (C_I[numfilas]  * sizeof(int));
            C_V = malloc (C_I[numfilas] * sizeof(float));
            for (z=0;z<C_I[numfilas];z++){
                C_J[z] = -1;
            }
```

```
611             }else{
612                 C_J = NULL;
613                 C_V = NULL;
614             }
615             for (z=0;z<numfilas;z++){
616                 acumularfila (subC_I,subC_J,subC_V,Ac_I,Ac_J,Ac_V,
617                               C_I,C_J,C_V,z,numfilas);
618             }
619             free ((*Ac_I));
620             if ((*Ac_I)[numfilas]>0){
621                 free ((*Ac_J));
622                 free ((*Ac_V));
623             }
624             (*Ac_I) = calloc (numfilas+1, sizeof(int));
625             (*Ac_I)[0:numfilas+1:1] = C_I[0:numfilas+1:1];
626             if (C_I[numfilas]>0){
627                 (*Ac_J) = malloc (C_I[numfilas]  * sizeof(int));
628                 (*Ac_V) = malloc (C_I[numfilas]  * sizeof(float));
629                 (*Ac_J)[0:C_I[numfilas]:1] = C_J[0:C_I[numfilas]:1];
630                 (*Ac_V)[0:C_I[numfilas]:1] = C_V[0:C_I[numfilas]:1];
631             }else{
632                 (*Ac_J) = NULL;
633                 (*Ac_V) = NULL;
634             }
635             free (C_I);
636             if (C_I[numfilas]>0){
637                 free(C_J);
638                 free(C_V);
639             }
640         }
641     }
```

8.1.11 Sumar

```
643  void sumar(int *subC_I,int *subC_J,float *subC_V,int numfilas,int pid,
644             int root,int npr,int dim,int **Ac_I,int **Ac_J,float **Ac_V,
645             MPI_Comm comm){
646      int i,modulo,pot;
647      double t0,t1,t2,t3,t4,t5,suma=0,suma2=0;
648      t0 = MPI_Wtime ();
649      for (i=0;i<(log2(npr));i++){
650          pot = pow(2,i);
651          modulo = pow(2,i+1);
652          if(pid % modulo == pot){
653              if (i==0){
654                  dim = subC_I[numfilas];
655                  MPI_Send(subC_I,numfilas+1,MPI_INT,(pid-pot),0,comm);
656                  MPI_Send(&dim,1,MPI_INT,(pid-pot),0,comm);
657                  if (dim > 0){
658                      MPI_Send(subC_J,dim,MPI_INT,(pid-pot),0,comm);
659                      MPI_Send(subC_V,dim,MPI_FLOAT,(pid-pot),0,comm);
660                  }
661                  free (subC_I);
662                  if (dim > 0){
663                      if (subC_J != NULL) free(subC_J);
664                      if (subC_V != NULL) free(subC_V);
665                  }
666              }else{
667                  dim = (*Ac_I)[numfilas];
668                  subC_I = calloc(numfilas+1, sizeof(int));
669                  subC_I[0:numfilas+1:1] = (*Ac_I)[0:numfilas+1:1];
670                  if(dim != 0){
671                      subC_J = malloc(dim * sizeof(int));
672                      subC_V = malloc(dim * sizeof(float));
673                      subC_J[0:dim:1] = (*Ac_J)[0:dim:1];
674                      subC_V[0:dim:1] = (*Ac_V)[0:dim:1];
675                  }
676                  free ((*Ac_I));
677                  if ((*Ac_I)[numfilas+1]>0){
678                      if ((*Ac_J) != NULL) free(*Ac_J);
679                      if ((*Ac_V) != NULL) free((*Ac_V));
680                  }
681                  MPI_Send(subC_I,numfilas+1,MPI_INT,(pid-pot),0,comm);
682                  MPI_Send(&dim,1,MPI_INT,(pid-pot),0,comm);
```

```
683                 if (dim > 0){
684                     MPI_Send(subC_J,dim,MPI_INT,(pid-pot),0,comm);
685                     MPI_Send(subC_V,dim,MPI_FLOAT,(pid-pot),0,comm);
686                 }else{
687                     subC_J = NULL;
688                     subC_V = NULL;
689                 }
690                 free (subC_I);
691                 if (dim > 0){
692                     if (subC_J != NULL) free(subC_J);
693                     if (subC_V != NULL) free(subC_V);
694                 }
695             }
696         }
697         if(pid % modulo == 0){
698             if (i==0){
699                 t2 = MPI_Wtime ();
700                 acumular (subC_I,subC_J,subC_V,Ac_I,Ac_J,Ac_V,i,dim,numfilas);
701                 free (subC_I);
702                 if (dim>0){
703                     if (subC_J != NULL) free(subC_J);
704                     if (subC_V != NULL) free(subC_V);
705                 }
706                 t3 = MPI_Wtime ();
707                 suma+=(t3-t2);
708             }
709             subC_I = calloc(numfilas+1, sizeof(int));
710             MPI_Recv(subC_I, numfilas+1, MPI_INT, (pid+pot), 0, comm,MPI_STATUS_IGNORE);
711             MPI_Recv(&dim, 1, MPI_INT, (pid+pot), 0, comm,MPI_STATUS_IGNORE);
712             if (dim > 0){
713                 subC_J = malloc(dim * sizeof(int));
714                 subC_V = malloc(dim * sizeof(float));
715                 MPI_Recv(subC_J, dim, MPI_INT, (pid+pot), 0, comm,MPI_STATUS_IGNORE);
716                 MPI_Recv(subC_V, dim, MPI_FLOAT, (pid+pot), 0, comm,MPI_STATUS_IGNORE);
717             }else{
718                 subC_J = NULL;
719                 subC_V = NULL;
720             }

721             t4 = MPI_Wtime ();
722             acumular (subC_I,subC_J,subC_V,Ac_I,Ac_J,Ac_V,1,dim,numfilas);
723             t5 = MPI_Wtime ();
724             suma2+=(t5-t4);
725             free (subC_I);
726             if (dim>0){
727                 if (subC_J != NULL) free(subC_J);
728                 if (subC_V != NULL) free(subC_V);
729             }
730         }
731     }
732     t1 = MPI_Wtime ();
733     if (pid == root){
734         fprintf (stderr,"time Sumar comm %1.6f calculo %1.6f\n",
735                             (t1-t0-suma-suma2)*1000,(suma+suma2)*1000);
736         sumatotal+=(t1-t0)*1000;
737     }
738 }
```

8.1.12 Main

```
740  int main (int argc, char** argv)
741  {
742      int pid,root,npr;
743      int numcolA,numfilasA,numvalA,numcolB,numfilasB,numvalB,numcolC,numfilasC,numvalC;
744      int i,j,k,nloc,nlocB,resto,restoB,ultimo,cont,pos,posB;
745      int primera,segunda,dim;
746      float tercera;
747      int *prA_I,*prA_J,*tamfranja;
748      float *prA_V,*actual_V,*mi_prB_V,*subC_V,*Ac_V;
749      int *actual_I,*actual_J;
750      int *mi_prB_I,*mi_prB_J;
751      int *subC_I,*subC_J;
752      int *Ac_I,*Ac_J;
753      int *vecpos;
754      int contAcum,tamAcum;
755      int *tam,*despl;
756      int *tamb,*desplb;
757      FILE *fichero,*ficheroB,*ficheroC;
758      MPI_Status info;
759      MPI_Comm comm;
760      double  t0, t1, t2,t3,suma;
761      MPI_Init(&argc,&argv);
762      comm = MPI_COMM_WORLD;
763      MPI_Comm_rank(comm, &pid);
764      MPI_Comm_size(comm,&npr);
765      tamfranja = malloc(npr * sizeof(int));
766      tam = malloc(npr * sizeof(int));
767      despl = malloc(npr * sizeof(int));
768      tamb = malloc(npr * sizeof(int));
769      desplb = malloc(npr * sizeof(int));
770      root=0;
```

```
771      if (pid == root){
772          fichero = fopen(argv[1],"r");
773          fscanf(fichero, "%d",&numfilasA);
774          fscanf(fichero, "%d",&numcolA);
775          fscanf(fichero, "%d",&numvalA);
776          nloc = numfilasA / npr;
777          resto = numfilasA % npr;
778
779          for (i=0; i<npr;i++){
780              tam[i] = nloc;
781              if(i<resto) tam[i]++;
782              despl[i] = tam[i]*i;
783              if (i>=resto) despl[i]+=resto;
784          }
785          MPI_Scatter(despl,1,MPI_INT,
786                      MPI_IN_PLACE, 1, MPI_INT,root,MPI_COMM_WORLD );
787      }
788      else
789      {
790       MPI_Scatter(despl,1,MPI_INT,
791                   &despl[pid], 1, MPI_INT,root,MPI_COMM_WORLD );
792      }
793      MPI_Bcast (&numfilasA,1,MPI_INT,root,comm);
794      contAcum = 0;
795      tamAcum = 0;
796      cont=0;
797      ultimo=0;
798
799      if (pid == root){
800          preLeer(fichero,&primera,&segunda,&tercera,tamfranja,numfilasA,despl,numvalA,npr);
801          fclose(fichero);
802          fichero = fopen(argv[1],"r");
803          fscanf(fichero, "%d",&numfilasA);
804          fscanf(fichero, "%d",&numcolA);
805          fscanf(fichero, "%d",&numvalA);
806      }
```

```
807     suma=0;
808     t0 = MPI_Wtime ();
809     for (i=0;i<npr;i++){
810         if (pid == root){
811             prA_I = calloc(numfilasA+1 , sizeof(int));
812             if (tamfranja[i]>0){
813                 prA_J = malloc(tamfranja[i] * sizeof(int));
814                 prA_V = malloc(tamfranja[i] * sizeof(float));
815             }else{
816                 prA_J = NULL;
817                 prA_V = NULL;
818             }
819             t2 = MPI_Wtime ();
820             pos = leer(fichero,&primera,&segunda,&tercera,prA_J,prA_V,prA_I,i,npr,&cont,
821               numvalA,&ultimo,numfilasA,despl,tam);
822             t3 = MPI_Wtime ();
823             suma+=(t3-t2);
824         }
825         for(j=0;j<npr;j++){
826             repartir(prA_J,prA_V,prA_I,npr,pos,numfilasA,j,i,pid,root,despl,tam,comm,
827                   info,&actual_I,&actual_J,&actual_V,&contAcum,&tamAcum);
828            }
829         if (pid == root){
830             free (prA_I);
831             if (tamfranja[i] > 0){
832                 if (prA_J != NULL) free(prA_J);
833                 if (prA_V != NULL) free(prA_V);
834             }
835         }
836     }
837     if (pid == root){
838         free (tamfranja);
839     }
840     t1 = MPI_Wtime ();
841     if (pid == root)
842     {fprintf (stderr,"time Repartir MatrizA %1.6f comm %1.6f\n",(t1-t0-suma-sumaA)*1000,sumaA*1000);
843      sumatotal+=(t1-t0-suma)*1000;
844     }
```

```
845     if (pid == root){
846         ficheroB = fopen(argv[2],"r");
847         fscanf(ficheroB, "%d",&numfilasB);
848         fscanf(ficheroB, "%d",&numcolB);
849         fscanf(ficheroB, "%d",&numvalB);
850         nlocB = numfilasB / npr;
851         restoB = numfilasB % npr;
852         for (i=0; i<npr;i++){
853             tamb[i] = nlocB;
854             if(i<restoB) tamb[i]++;
855             desplb[i] = tamb[i]*i;
856             if (i>=restoB) desplb[i]+=restoB;
857         }
858     }
859     vecpos = malloc(npr * sizeof (int));
860     matrizB(ficheroB,&mi_prB_I,&mi_prB_J,&mi_prB_V,npr,comm,numfilasB,
861             numcolB,numvalB,pid,root,tamb,desplb,&vecpos,argv);
862     t0 = MPI_Wtime ();
863         dim = producto(actual_I,actual_J,actual_V,mi_prB_I,mi_prB_J,mi_prB_V,
864                 &subC_I,&subC_J,&subC_V,numfilasA,pid,tamAcum,tamb,despl);
865     t2 = MPI_Wtime ();
866     MPI_Barrier(comm);
867     t3 = MPI_Wtime ();
868     t1 = MPI_Wtime ();
869     if (pid == root)
870     {fprintf (stderr,"time Producto %1.6f\n",(t1-t0-(t3-t2))*1000);
871     sumatotal+=(t1-t0-(t3-t2))*1000;
872     }
873     sumar(subC_I,subC_J,subC_V,numfilasA,pid,root,npr,dim,&Ac_I,&Ac_J,&Ac_V,comm);
874
875     if (pid==root) fprintf (stderr,"time total %1.6f\n",sumatotal);
876     free(actual_I);
877     if (contAcum>0)
878     {
879      free(actual_J);
880      free(actual_V);
881     }
882     free(mi_prB_I);
```

```
883     if (vecpos[pid]>0)
884     {
885      free(mi_prB_J);
886      free(mi_prB_V);
887     }
888     free(vecpos);
889     if (pid == root){
890         free(Ac_I);
891         free (Ac_J);
892         free(Ac_V);
893         free(tam);
894         free(tamb);
895         free(despl);
896         free(desplb);
897     }
898     MPI_Finalize();
899     return (0);
900 }/*Main*/
```

8.2 Código en MATLAB para la generación de las matrices

```
function [] = Producto(n,p)

WA= sprandn(n,n,p);
[col row val]=find (WA);
elemA = size(row);
val = round(rand(elemA)*9+1);
A=sparse(row,col,val,n,n);

str = sprintf('F:\\cleo\\Ander\\Tipo2_A_%d_%d.txt',n,n);
fileID = fopen(str,'w');
fprintf(fileID,'%d %d %d\r\n',n,n,elemA(1));
for i = 1:elemA
    fprintf(fileID,'%d %d %f\r\n',row(i)-1,col(i)-1,val(i));
end
fclose(fileID);

WB= sprandn(n,n,p);
[col row val]=find (WB);
elemB = size(row);
val = round(rand(elemB)*9+1);
B=sparse(row,col,val);
str = sprintf('F:\\cleo\\Ander\\Tipo2_B_%d_%d.txt',n,n);
fileID = fopen(str,'w');
fprintf(fileID,'%d %d %d\r\n',n,n,elemB(1));
for i = 1:elemB
    fprintf(fileID,'%d %d %f\r\n',row(i)-1,col(i)-1,val(i));
end
fclose(fileID)

 C=A*B;
 [row col val]=find (C);
 elemC = size(row);
 MC=sortrows([row col val],1);
 str = sprintf('F:\\cleo\\Ander\\Tipo2_C_%d_%d.txt',n,n);
 fileID = fopen(str,'w');
 fprintf(fileID,'%d %d %d\r\n',n,n,elemC(1));

 for i = 1:elemC
    fprintf(fileID,'%d %d %f\r\n',MC(i,1)-1,MC(i,2)-1,MC(i,3));
end
fclose(fileID)

end
```