W_ILU_ GMRES: UN DISEÑO EN PARALELO

Yisleidy Linares Zaila, Marta L. Baguer Díaz Romañach, Ángela León Mecías y
Universidad de La Habana, Facultad de Matemática y Computación, San Lázaro y L, Vedado 10400, Habana 4, Cuba

ABSTRACT
This paper proposes a parallel design of W_ILU_GMRES strategy, which is an iterative technique for solving systems of linear equations with dense and large matrices. First basic concepts related to parallelism are explained. Then we propose a parallel design, which uses multi-core technology and therefore has shared memory architecture. As a language for developing W_ILU_GMRES was chosen FSharp. Some experiments were performed in order to measure the quality of the parallel implementation.

KEY WORDS: parallel computing, preconditioning, parallel wavelet compression, Krylov subspaces methods.

MSC: 68W10

RESUMEN
En este trabajo se propone un diseño en paralelo de la estrategia W_ILU_GMRES, la cual es una técnica iterativa para resolver sistemas de ecuaciones lineales con matriz densa y de grandes dimensiones. Primeramente se explican conceptos básicos relacionados con el paralelismo. Después se presenta el diseño en paralelo que utiliza tecnología *multi-core* y por tanto está implementado sobre una arquitectura de memoria compartida. Como lenguaje para el desarrollo de W_ILU_GMRES se escogió FSharp. Se realizaron algunos experimentos para medir la calidad de la implementación en paralelo.

1. INTRODUCCIÓN

La solución de sistemas de ecuaciones lineales de grandes dimensiones con matrices densas por métodos directos trae aparejado un alto costo computacional, de ahí que se empleen como vía alternativa técnicas iterativas, especialmente las basadas en los subespacios de Krylov (Y. Saad, 2000), que aplicadas de conjunto con precondicionadores permiten acelerar la convergencia. En un trabajo reciente (L. Villarín, A. León, M. L. Baguer, Y. Linares, 2012), se propuso la construcción de un precondicionador para GMRES, al cual denominamos W_ILU_GMRES.
La construcción del precondicionador está basada en la factorización LU incompleta de la aproximación dispersa no singular de la matriz densa A. El diseño e implementación del algoritmo fue presentado en forma secuencial.
Como resultado de la experimentación numérica realizada en (Y. Saad, 2000), se obtuvo que cuando la cantidad de elementos no nulos en la aproximación dispersa crece, la calidad del precondicionador también crece, sin embargo, su costo de obtención crece en el mismo orden. Además, la estrategia de selección automática del umbral que se propone se basa en un algoritmo de ordenación, con costo $O(n^2 \log_2 n)$, donde n es la dimensión de la matriz, que se impone como cota inferior de la propuesta. Como también se explica en (L. Villarín, A. León, M. L. Baguer, Y. Linares, 2012), para grandes dimensiones se requieren aproximaciones dispersas con una cantidad de elementos no nulos mayor que $O(n \log_2 n)$, entonces la obtención de la factorización LU Incompleta (ILU) tiene un costo mayor que $O(n^2 \log_2 n)$, dominando el costo total de la estrategia. Por lo tanto, al aumentar la cantidad de elementos no nulos de la matriz nz, aumenta el costo de la obtención de la factorización ILU. Para matrices dispersas $nz = n * O(n)$, obtener el precondicionador ILU tendría un costo $O(n^3)$, con lo que no se podría mejorar la solución vía métodos directos. La interrogante sería, ¿cómo encontrar una alternativa para superar esta limitante?
En los últimos años el número de aplicaciones que involucran el uso del paralelismo ha sido incrementado en gran medida. Existe una amplia variedad de algoritmos cuyo objetivo es paralelizar tanto precondicionadores como técnicas iterativas que resuelven sistemas de ecuaciones lineales de grandes dimensiones. Sin embargo, la mayoría de estos, véase (R. D. D. Cunha, T. Hopkins, 1994), (J. Erhel, 1995), (M. Benzi, W. Joubert, G. Mateescu, 1999) están desarrollados para plataformas y arquitecturas específicas tales como clústeres y sistemas con memoria distribuida respectivamente. En la revisión bibliográfica realizada no se encontró trabajo alguno acerca del diseño en paralelo de estos algoritmos implementados sobre las nuevas tecnologías, como la tecnología *multi-core*.

En el presente trabajo se propone un diseño en paralelo de W_ILU_GMRES, estructurado en tres secciones además de la introducción. En la segunda sección se introducen algunos conceptos esenciales relacionados con el desarrollo de programas en paralelo, así como las principales características del hardware utilizado y del lenguaje FSharp, este último seleccionado para la implementación de la estrategia. En la tercera sección se presenta el diseño en paralelo así como la estructura de datos utilizada para la representación de las matrices dispersas, que constituye el núcleo fundamental del trabajo. El análisis de los costos de cada una de las etapas y los resultados de los experimentos realizados aparecen en la última sección.

2. COMPUTACIÓN EN PARALELO. ALGUNAS DEFINICIONES

2.1. Hardware

En este trabajo se utilizó la tecnología *multi-core*. Esta consiste en computadoras cuyo procesador contiene más de un núcleo en el mismo paquete físico. El objetivo de este diseño es lograr que varias tareas sean ejecutadas simultáneamente, y por tanto mejorar el desempeño de todo el sistema.
Esta tecnología, según la *Taxonomía de Flynn*[1], encontrada en (B. Barney, 2009) se sitúa en el modelo MIMD (*Multiple Instruction Multiple Data*) donde diferentes procesadores ejecutan diferentes instrucciones sobre distintos segmentos de datos (Figura 1)
Por otra parte, atendiendo al acceso a la memoria RAM, la tecnología *multi-core* presenta una arquitectura de memoria compartida (*Shared Memory*)(B. Barney, 2009).

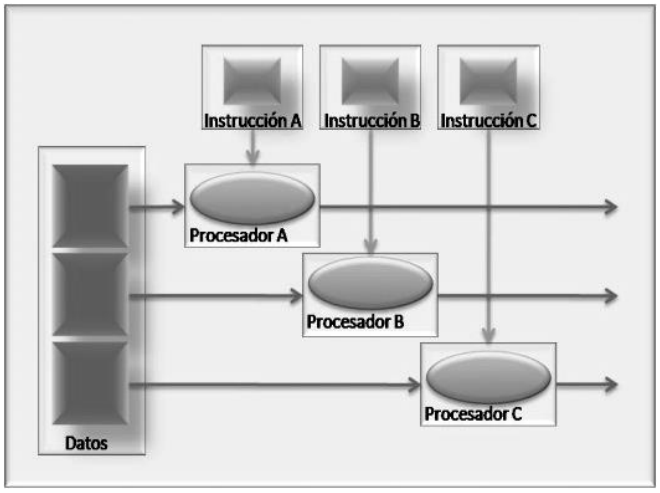

Figura 1Modelo MIMD

2.2. Software

El primer paso para diseñar un algoritmo en paralelo es identificar aquellas partes que pueden ejecutarse simultáneamente. Es preciso enfocarse en los puntos críticos, es decir, aquellas secciones que requieren un alto consumo de CPU e identificar las partes que no se pueden paralelizar. Posteriormente se pasa al diseño de la solución, el cual está dividido en fases entre las que se encuentran: el particionado, la comunicación entre los diferentes procesadores y la fase de sincronización. La primera es la etapa donde se define la forma en la que se fragmenta el problema para poder diseñar una solución del mismo en paralelo. En la segunda se determina el modo en el que los procesadores trasmitirán la información. En la última fase se analiza la dependencia de las distintas tareas garantizando que no se viole el orden de precedencia. Para una explicación más detallada véase(I. Foster, 1995).
Otro aspecto importante a tener en cuenta es la eficiencia del programa en paralelo. El factor determinante en el costo de un algoritmo en paralelo no es el número de procesadores sino el porciento del problema que puede escribirse en paralelo. Así lo demuestra la ley de Amdahl[2], formulada en 1960

[1] **Michael J. Flynn** (nacido el 20 de mayo de 1934 en Nueva York) es un profesor emérito de la Universidad Stanford estadounidense, con estudios en ingeniería electrónica y ciencias de la computación. En 1995 recibió el premio *Harry H. Goode Memorial Award* por sus contribuciones al área del procesamiento de la información.
[2] **Eugene MyronAmdahl** (nacido el 16 de noviembre de 1922, Flandreau, Dakota del Sur) norteamericano de origen noruego, arquitecto computacional y una de las personalidades más importantes y excéntricas en la historia de la informática y la computación. Fundó cuatro compañías tecnológicas en diferentes ámbitos, la mayoría de las cuales, se fundaron con el objetivo de

por Gene Amdahl, la cual establece que la porción del problema que no se puede paralelizar constituye un límite en la velocidad de procesamiento, véase(I. Foster, 1995). Una forma práctica de medir el aumento de esta velocidad es mediante la fórmula:

$$Speedup = \frac{T_1(n)}{T_N(n)}$$

donde $T_1(n)$ es el tiempo de ejecución del programa secuencial y $T_N(n)$es el tiempo de ejecución en paralelo con *N* procesadores. $T_1(n)$ y $T_N(n)$ son funciones que dependen de la dimensión de la matriz. Esta fórmula introduce un concepto importante en la programación en paralelo: la escalabilidad. Un programa en paralelo se dice escalable cuando al aumentar el número de procesadores aumenta la velocidad de ejecución.

2.3. Microsoft y el paralelismo

La industria de los microprocesadores se ha inclinado hacia la producción de procesadores *multi-core*. Debido a este cambio, *Visual Studio 2010* y *.NET 4.0*, incluyen la biblioteca *Task Parallel Library* (TPL) y un nuevo lenguaje funcional y orientado a objetos llamado FSharp (*F#*) que hace la programación paralela y asíncrona más sencilla en un ambiente amigable. Más detalles acerca de cada una de estas bibliotecas se encuentran en(Microsoft Developers, 2010).

2.3.1. FSharp

F# combina las cualidades de la programación funcional, tales como código claro y conciso, con las bibliotecas, herramientas, interoperabilidad y modelo de objetos de la plataforma .Net. Es el único lenguaje con esta característica, es decir, el único capaz de sacar lo mejor de ambos paradigmas (el funcional y el orientado a objetos).
Como ventajas significativas de este lenguaje se tienen que a través del mismo se puede extender la estrategia a una aplicación MPI[3] lo cual es de gran importancia teniendo en cuenta que se lograría incrementar mucho más la dimensión de los sistemas de ecuaciones lineales. También en FSharp las estructuras de datos son inmutables, lo cual permite un mayor aprovechamiento del paralelismo. Otro factor importante es que su uso no solo se restringe a Windows, este también puede utilizarse en Linux, Apple Mac OS X and OpenBSD. Para una argumentación más profunda véanse (J. Harrop, 2008),(R. Pickering, 2007), (D. Syme, 2007).

2.4. Factorización LU Incompleta (ILU)

Este proceso se divide en dos pasos fundamentales, el primero es la obtención de los factores *L* y *U* y el segundo es la aplicación de esta factorización, es decir, la sustitución hacia adelante y retrógrada.
Existen varias formas de realizar esta factorización LU Incompleta, siguiendo diferentes estrategias de creación de relleno, ver (Y. Saad, 2000); aquí se utiliza aquella que no incorpora nuevas entradas no nulas durante su obtención, conocida como ILU (0). Este algoritmo está basado en la eliminación gaussiana y su única diferencia con la LU completa es que solo utiliza los elementos distintos de cero de la matriz a factorizar. Las optimizaciones para el indizado de matrices *sparse* hacen que el costo de este algoritmo dependa de la cantidad de elementos no nulos que tenga la matriz. Un aporte del trabajo es la modificación que se realiza a este algoritmo en cuanto a la forma en que se actualizan los elementos.
La implementación usual realiza el proceso por filas, o sea, en el momento en el que se pasa a procesar la fila $i+1$, la fila $i-ésima$ ya está completamente lista. En la Figura 2, se representa este mecanismo.
Las componentes con tonalidad más oscura son aquellas que ya tienen su resultado final. Las más claras, son las que serán modificadas utilizando el multiplicador. Nótese como m [3,1] no está listo para utilizarse como multiplicador hasta que no fue actualizado en el paso anterior por m [3,0] . El inconveniente de esta alternativa es la sincronización, pues para utilizar a m [3,1], antes este tuvo que ser actualizado por m [3,0], y lo mismo ocurre con m [3,2]. La modificación que se propone es realizar el proceso por columnas, verFigura 3.

competir con otras compañías donde él mismo había trabajado o incluso creado anteriormente, y que ya había abandonado. Un ejemplo de esto es la famosa empresa informática IBM que sigue en auge actualmente, y que abandonó para hacerle la competencia.
[3] Del inglés Message Passing Interface, plataforma que permite la ejecución de algoritmos en paralelo en una arquitectura de memoria distribuida.

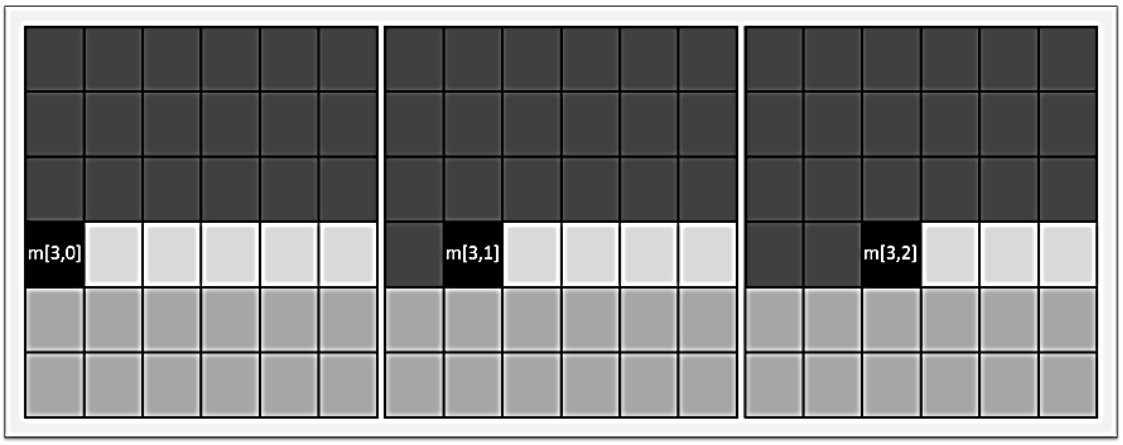

Figura 2 Proceso de factorización ILU (0) usual

Con este cambio se resuelve el problema de la sincronización, pues m [2,0] no depende de m [1,0] en ningún momento. Esta nueva implementación es equivalente a la anterior, la diferencia es que no son las filas las que se irán terminando, sino las columnas. Cuando se pase a la columna $j + 1$ la $j - ésima$ ya estará lista. De esta forma se pueden procesar las filas paralelamente

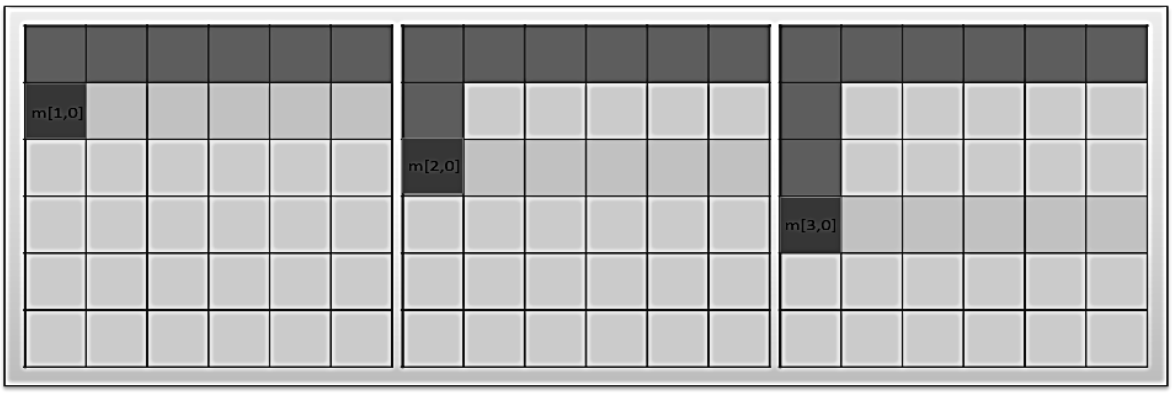

Figura 3 Factorización ILU (0) modificada

En la Figura 4 se puede apreciar en qué consiste el proceso en paralelo. La matriz se divide en bloques filas y se modifican con su multiplicador correspondiente. Nótese cómo la actualización de las filas utilizando los multiplicadores que están en la columna $j - ésima$ es independiente, por tanto no se tiene el problema de sincronización presente en el diseño usual. Otra observación importante es que no se pasa a la columna $j + 1$ hasta que todos los procesadores terminaron su tarea en la columna $j - ésima$, es decir, hasta que todas las filas fueron actualizadas con su multiplicador correspondiente. En este punto es donde se tiene en cuenta la comunicación entre los procesadores, cada uno comunica sus modificaciones y se redistribuyen las filas.

.

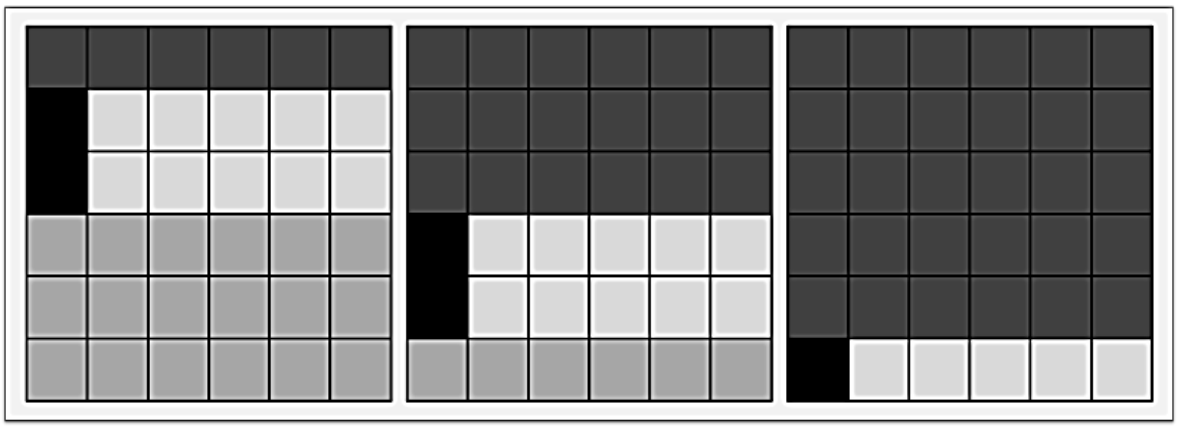

Figura 4 Factorización ILU (0) en paralelo

2.4.1. Sustitución hacia adelante y retrógrada

En la aplicación del precondicionador el análisis es muy similar, ver (Figura 5).

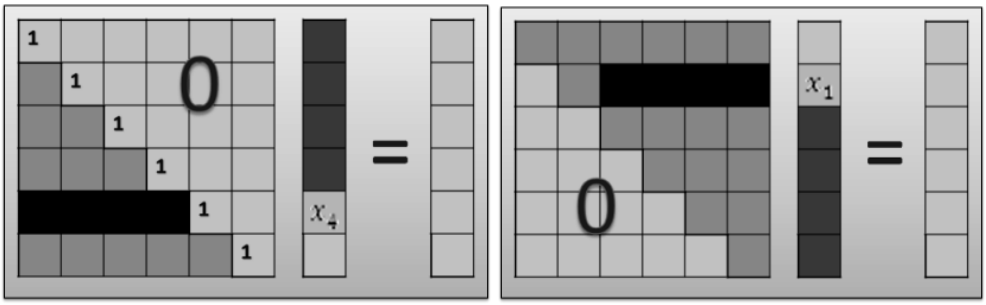

Figura 5 Sustitución hacia adelante y retrógrada usual

La modificación que fue introducida aquí es, por tanto, muy parecida a la que se realizó para la obtención del precondicionador. La actualización de las x_i se hizo por las columnas, es decir, en el paso $i-ésimo$ se calcula la componente x_i y se modifican, para la x_i calculada, las componentes x_j, con $j > i$ para la sustitución hacia adelante y $j < i$ para la retrógrada, ver Figura 6. De esta forma en el paso $i-ésimo$ se actualizan todas las componentes que dependen de la x_i calculada simultáneamente.

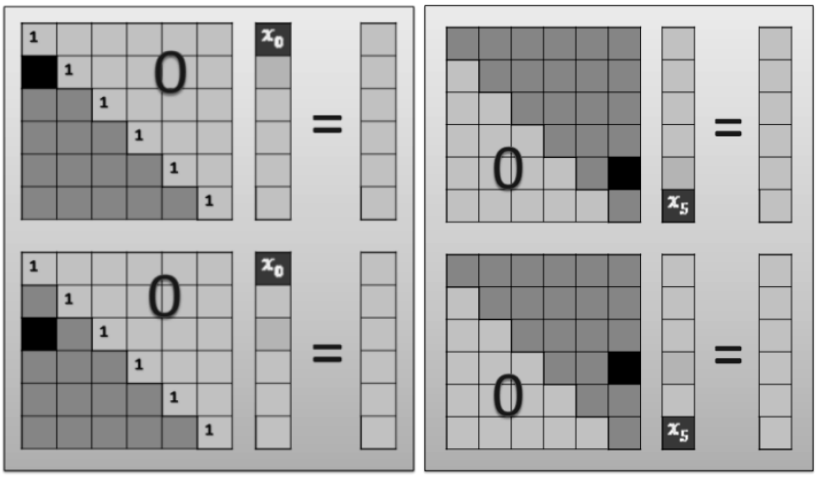

Figura 6 Sustitución hacia adelante y retrógrada modificada

En la Figura 7se muestran dos pasos en paralelo de la sustitución hacia adelante y retrógrada respectivamente. En ambos casos el paralelismo se aplica en la actualización de las x_i. No se procesa la columna $j + 1$ hasta que las actualizaciones de las x_j correspondientes a la columna $j-ésima$ no hayan concluido evitando cualquier problema de sincronización.

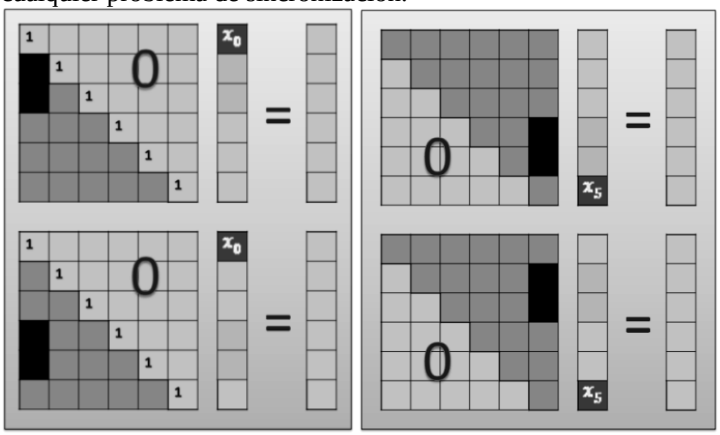

Figura 7 Sustitución hacia adelante y retrógrada en paralelo

3. W_ILU_GMRES EN PARALELO

El diseño en paralelo se desarrolló sobre la tecnología *multi-core*, el tipo de particionado utilizado fue Descomposición del Dominio (*Domain Decomposition*), el cual está centrado en los datos. Cada procesador trabaja, en paralelo, con una porción de los mismos.
Como se explica en (L. Villarín, A. León, M. L. Baguer, Y. Linares, 2012), dado un sistema de ecuaciones lineales $Ax = b$, aplicar W_ILU_GMRES significa:

a) Representar A en una base Wavelet.
b) Escoger un umbral de truncamiento respecto al cual se obtiene una aproximación dispersa.
c) Obtener el precondicionador ILU (0).
d) Resolver el sistema disperso mediante GMRES precondicionado.

En lo que sigue se explicará el diseño en paralelo de cada una de estas etapas.

3.1. Transformada Wavelet Bidimensional en paralelo

El diseño consiste en dividir la matriz en una cierta cantidad de bloques, dependiente del número de procesadores disponibles. De esta manera cada procesador se encargará de aplicar la transformada a su bloque correspondiente.
Teniendo en cuenta que se utilizó la transformada estándar, el algoritmo primero divide a la matriz en bloques por filas, cada procesador transforma la fila i que le corresponde hasta el nivel deseado y luego pasa a la $i + 1$. Cuando un procesador termina con una fila computa otra que no haya sido procesada, de esta forma un procesador se detiene solo cuando no queden filas. Una vez que se terminan de calcular todas las filas, se realiza el mismo procedimiento por las columnas de la matriz resultante, ver (Figura 8 y Figura 9).
.

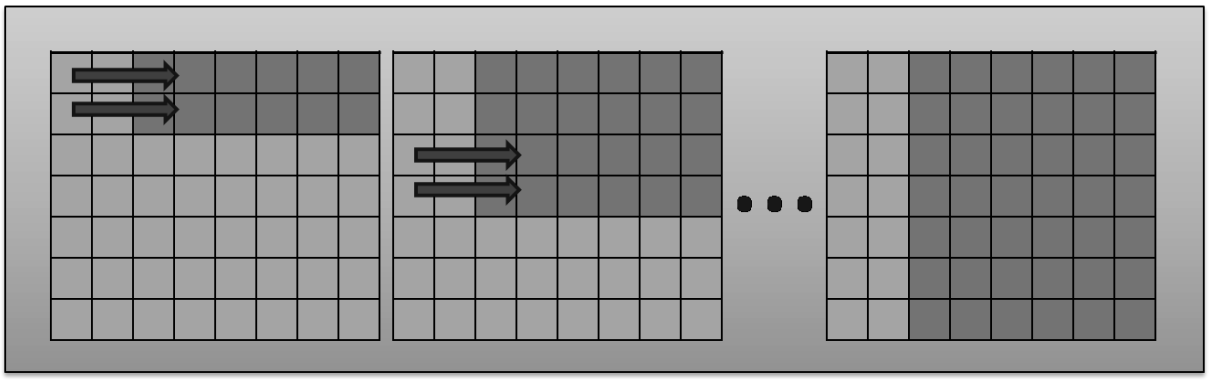

Figura 8 Wavelet por filas, en paralelo

Cuando se espera a que todos los procesadores terminen con sus bloques filas, para comenzar con las columnas, se está teniendo en cuenta la sincronización y comunicación entre estos, pues aquí es donde se manifiesta una dependencia de las tareas y de los datos.

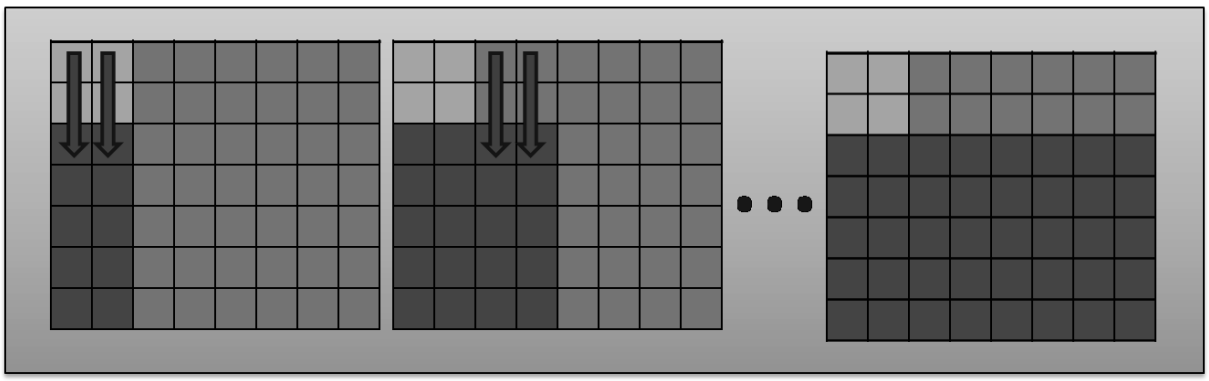

Figura 9 Wavelet por columnas, a la matriz resultante, en paralelo

3.2. Selección del umbral de truncamiento

La matriz se convierte en un vector de n^2 componentes que se ordena para obtener el $k-ésimo$ menor que constituirá el umbral a partir del cual se obtendrá la aproximación dispersa de la matriz densa original. Una descripción detallada de la estrategia, se puede ver en (L. Villarín, A. León, M. L. Baguer, Y. Linares, 2012). El proceso de conversión a vector es muy fácil de paralelizar (Figura 10).

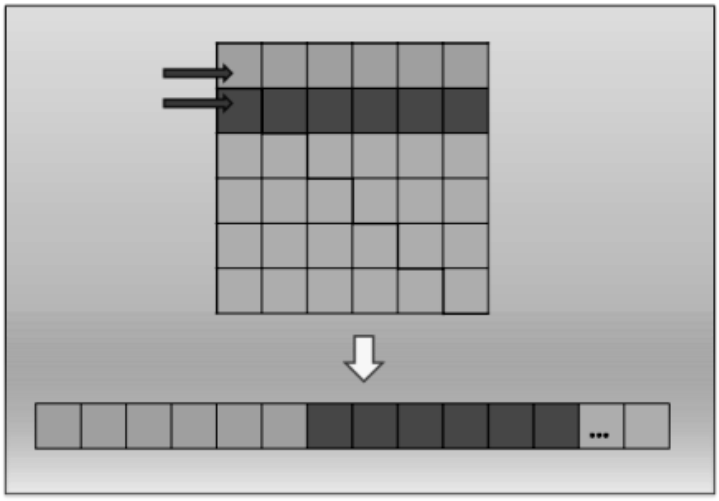

Figura 10 Proceso de conversión de matriz a vector, en paralelo

Cada procesador se encarga de sus filas correspondientes y una vez que todos terminaron, se tiene el vector con todos los datos.

Se realizó una implementación en paralelo del algoritmo QuickSort, véase (T. H. Cormen, C. E. Rivest, R. L. Leiserson, 2009), pues es, por sus características, uno de los que más aprovecha el paralelismo debido fundamentalmente a su estrategia de divide y vencerás. La idea de este algoritmo es dividir el vector (o porción del vector) a ordenar en dos partes (no necesariamente de igual tamaño) donde cualquier elemento de la primera es menor que cualquier otro de la segunda. Esta división se realiza usando un valor del mismo vector a ordenar (elemento pivote). Una vez realizada esta operación, se le aplica el algoritmo recursivamente a cada una de las dos partes obtenidas.

El diseño en paralelo que se propone aprovecha la implementación secuencial, pues el paralelismo se aplica a cada uno de los llamados recursivos. Un detalle interesante es que con un vector de tamaño relativamente pequeño $(m = 5000)$ es más eficiente, en cuanto a tiempo de ejecución, utilizar el algoritmo secuencial, pues resulta más costoso crear dos nuevos hilos de ejecución, que ordenar el segmento de datos de forma secuencial.

3.3. GMRES en paralelo

GMRES es un algoritmo iterativo de naturaleza secuencial, véase (Y. Saad, 2000), luego aquí solo se pueden paralelizar las operaciones que se realizan en una iteración. Las operaciones de mayor peso son el proceso de Arnoldi-Householder y la resolución del sistema $H_m z_m = \|r_0\|_2 e_1$. Arnoldi-Householder se basa en operaciones $matrizxvector y vectorxvector$.Ambas constituyen ejemplos clásicos de métodos en paralelo.

En la multiplicación $matriz\,x\,vector$ y $vector\,x\,vector$ secuencial el cálculo de cada una de las componentes es independiente del resto, por tanto la implementación en paralelo de estas operaciones se realiza directamente.

Tanto la matriz como el vector columna se dividen en bloques filas, (Figura 11 y Figura 12) y cada procesador se encarga de calcular su parte totalmente independiente de los otros. Esta idea es bastante similar a la encontrada en (G. Golub, C. F. Van Loan, 1996), solo que Golub tuvo en cuenta un conjunto de factores que hoy, con la evolución de la tecnología, son transparentes al programador.

Nótese que en el caso de $matriz\ x\ vector$ para calcular la componente x_i se requiere de la fila i y de todas las componentes del vector por el que se multiplica y en el caso de $vector\ x\ vector$ para calcular la fila i se necesita de la componente $i-ésima$ del vector columna. Esto pudiera suponer un problema de acceso a memoria, pero como estos valores no son modificados en ningún momento, los procesadores pueden acceder simultáneamente sin provocar ningún error en el resultado.

Para la resolución del sistema $H_m z_m = \|r_0\|_2 e_1$, con H_m matriz triangular superior, basta hacer la sustitución retrógrada.

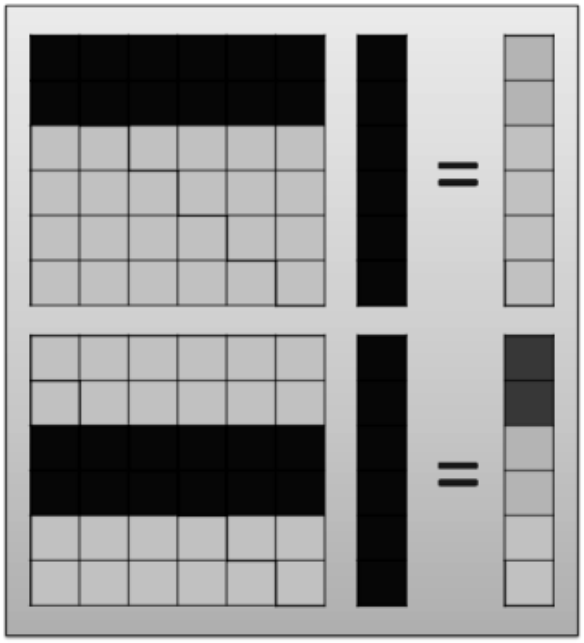

Figura 11 Multiplicación matriz x vector en paralelo

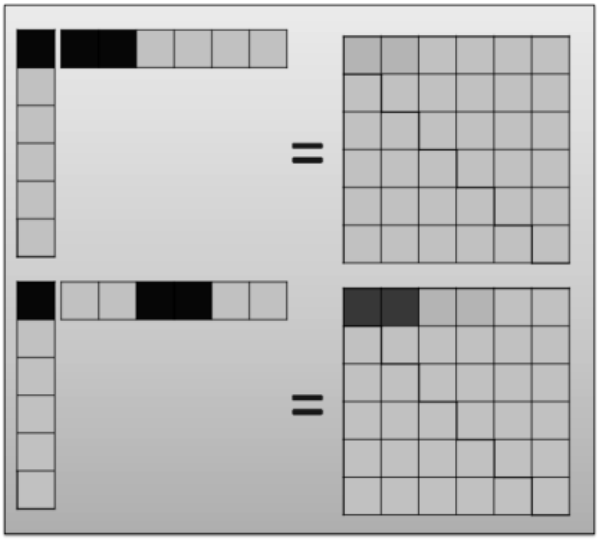

Figura 12 Multiplicación vector x vector en paralelo

3.4. Estructura *Sparse*

En el momento en el que se obtiene una aproximación dispersa de la matriz densa, se hace necesario utilizar alguna estructura que almacene los elementos distintos de cero y permita implementar las operaciones básicas aprovechando la característica dispersa. Existen distintas estructuras de datos que resuelven este problema, tales como CRS (*Compressed Row Storage*) y JDS (*Jagged Diagonal Storage*), (S. Hossian, 2006)y(G. Haase, M. Liebmann, G. Plank, 2007). Todos estos patrones de almacenamiento han sido diseñados para ajustarse a las características de problemas específicos. Cualquiera de ellos pudo haber sido utilizado, pero cuando se tiene en cuenta que es preciso realizar eficientemente: el recorrido tanto por filas, como por columnas, la multiplicación $matriz\ x\ vector$ y $matriz\ x\ matriz$ tanto secuencial como en paralelo y las operaciones relacionadas con el precondicionador ILU (0), resultó más conveniente diseñar una estructura propia afín con esta estrategia.
El diseño que se propone se basa en listas doblemente enlazadas, constituidas por nodos con los campos:

- $Value$, donde se almacenará el valor del elemento en la matriz que este representa.
- Row, indica la fila correspondiente a $Value$.
- $Column$, corresponde a la columna de $Value$.
- W, E, N, S, referencian a los elementos correspondientes a la izquierda, derecha, arriba y abajo respectivamente.

Se tienen dos arreglos unidimensionales (WE: izquierda-derecha y NS: arriba-abajo) de tamaño n (dimensión de la matriz). WE tiene en la posición j una referencia al primero y último nodo correspondiente a la columna$j-ésima$, mientras que NS tiene en la posición i una referencia al primero y último nodo correspondiente a la fila $i-ésima$ (Figura 13). Por otra parte se implementaron enumeradores que garantizan el recorrido tanto por las filas como por las columnas. Las operaciones de $matriz\ x\ vector$y $matriz\ x\ matriz$se realizan de manera muy similar al procedimiento a seguir con las matrices densas.

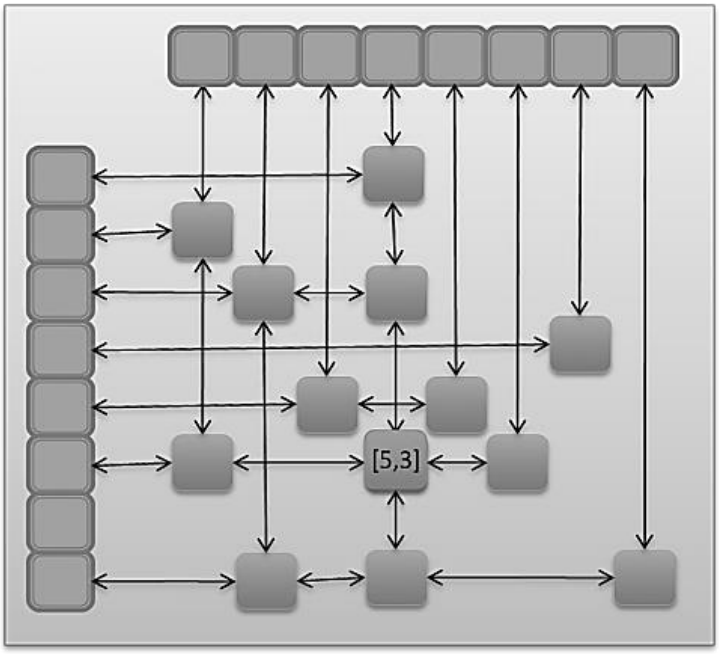

Figura 13 Estructura de datos para la matriz *dispersa*

Para optimizar las operaciones en ILU (0) se implementó otro enumerador que itera a partir del primer elemento que constituya un multiplicador en una columna, pues en el procedimiento de ILU (0) se tendrían que recorrer elementos innecesarios (Y. Saad, 2000), véase la Figura 13. Como solución a este problema se implementaron dos nuevos enumeradores uno que itera por todos los multiplicadores y otro que itera por la fila a partir de cierta columna. De esta manera el diseño cumple con todos los requisitos mencionados anteriormente.

4. EXPERIMENTACIÓN NUMÉRICA Y ANÁLISIS DE RESULTADOS

Las experimentaciones están orientadas a medir la escalabilidad del diseño en paralelo, ya que es así como se evalúa la eficiencia del programa. Se realizaron cuatro experimentos fundamentales. En el primero y el segundo se analizan los algoritmos que intervienen en la obtención de la aproximación dispersa y del precondicionador ILU (0) respectivamente. El tercer experimento está dedicado al estudio del comportamiento de GMRES en paralelo y por último se realiza un análisis de la estrategia en su conjunto.
Se utilizó la matriz de Lehmer, que es simétrica y definida positiva, y haciendo uso de su definición implícita se puede variar la dimensión tanto como se desee.

4.1. Comportamiento de la obtención de la aproximación *dispersa*

Este primer experimento se diseñó con el objetivo de analizar el comportamiento de la velocidad de ejecución en la obtención de la aproximación dispersa de la matriz original (aplicación de la transformada Wavelet bidimensional estándar y selección del umbral de truncamiento), para uno, dos y cuatro

procesadores haciendo variar la dimensión de la matriz y el nivel de resolución de la transformada Wavelet. La Figura 14 y la Figura 15 muestran el buen aprovechamiento del paralelismo. Nótese cómo el *speedup* se acerca bastante al número de procesadores, lo que indica la escalabilidad del diseño.

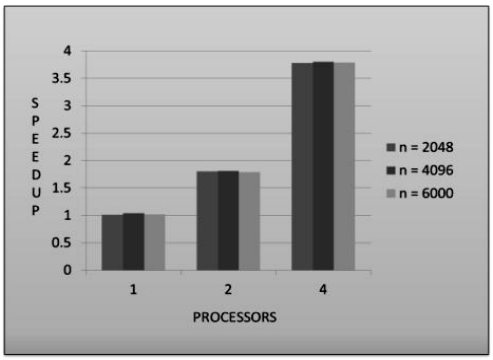

Figura 14 *Speedup* variando la dimensión de la matriz

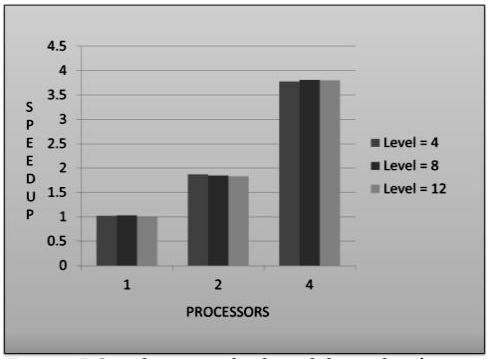

Figura 15 *Speedup* variando el nivel de resolución

La conclusión fundamental es que se puede aumentar la dimensión de la matriz y/o el nivel de resolución y el *speedup* se mantendrá constante. Esto induce a pensar que incrementar la dimensión no afecta la velocidad de ejecución. Por otra parte, según la teoría llegará un momento en que aumentar la cantidad de procesadores no supondrá un aumento de la velocidad. Asumiendo que la porción del programa que realmente se ejecuta en paralelo es ($P \approx 0.98$) y aplicando las fórmulas correspondientes que aparecen en (I. Foster, 1995) a este ejemplo, se obtiene la Tabla 1 donde se aprecia la dependencia de la velocidad de ejecución con respecto al número de procesadores.

Tabla 1Análisis del número de procesadores contra parte en paralelo

N	**P = 0.98**
2^0	1
2^1	1.96
2^2	3.77
2^3	7.01
2^4	12.30
⋮	⋮
2^{17}	49.98
2^{18}	49.99
2^{19}	49.99

Este análisis no tiene en cuenta los costos adicionales, como la comunicación entre procesadores y por tanto su interés es esencialmente teórico. Tanto es así que el resultado práctico obtenido para dos procesadores es de aproximadamente 1.82, véase la Figura 15. Lo que sí se puede afirmar y comprobar en la práctica (disponiendo de la tecnología adecuada) es que un aumento de la cantidad de procesadores no siempre redunda en una disminución significativa del tiempo de ejecución.

4.2. Comportamiento de ILU (0)

En esta segunda parte de la experimentación se hicieron dos pruebas, una haciendo variar la dimensión de la matriz y otra cambiando la cantidad de elementos no nulos.

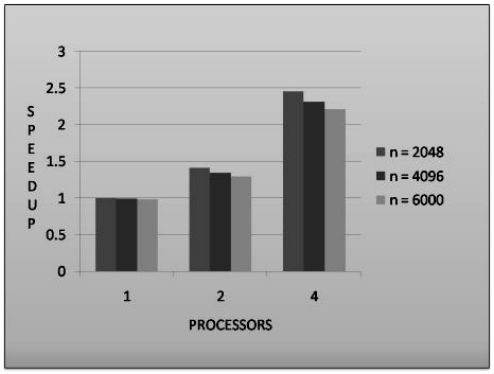

Figura 16 *Speedup* variando la dimensión en ILU (0)

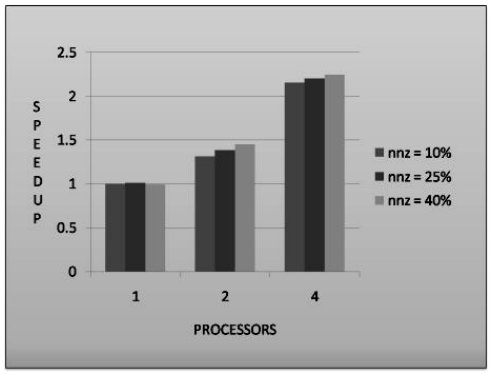

Figura 17 *Speedup* variando el umbral en ILU (0)

Esta etapa constituye el nodo crítico de la estrategia, tanto para el diseño secuencial como para el paralelo. Ambos gráficos así lo demuestran. En la implementación en paralelo el problema radica en que cuando la matriz es muy dispersa, contiene muy pocos elementos por fila, además de que, como se vio, en cada fila no se procesan todas las componentes. Esto trae como consecuencia que se queden procesadores ociosos y exista una diferencia notable entre el momento en que termina el primero y el último. Nótese cómo en laFigura 16 al aumentar la dimensión de la matriz y mantener fijo el umbral esta se fue haciendo más dispersa, por lo que se afectó la escalabilidad, y en la Figura 17 al aumentar la cantidad de elementos no nulos se aumenta la escalabilidad.

Este aspecto conlleva a una conclusión interesante. En el caso de la implementación secuencial, se tiene que a mayor cantidad de elementos nulos, se obtiene un menor costo de ILU (0). Como ILU (0) es quien domina el costo total de la estrategia, esto obliga a que se necesite una matriz bien dispersa. Por otra parte no se puede utilizar una aproximación con muy pocos elementos distintos de cero porque la probabilidad de que la matriz sea singular es muy alta, entonces hay que buscar un umbral que equilibre el costo de ILU (0) y la condición de no singularidad. Con la implementación en paralelo es cierto que el costo se mantiene. Sin embargo, resultó que mientras menos dispersa es la matriz el diseño es más escalable, es decir, aprovecha mejor los procesadores disponibles. Esto brinda la posibilidad de aumentar un poco la cantidad de elementos no nulos y por ende encontrar una mejor aproximación de la matriz.

4.3. Comportamiento de GMRES

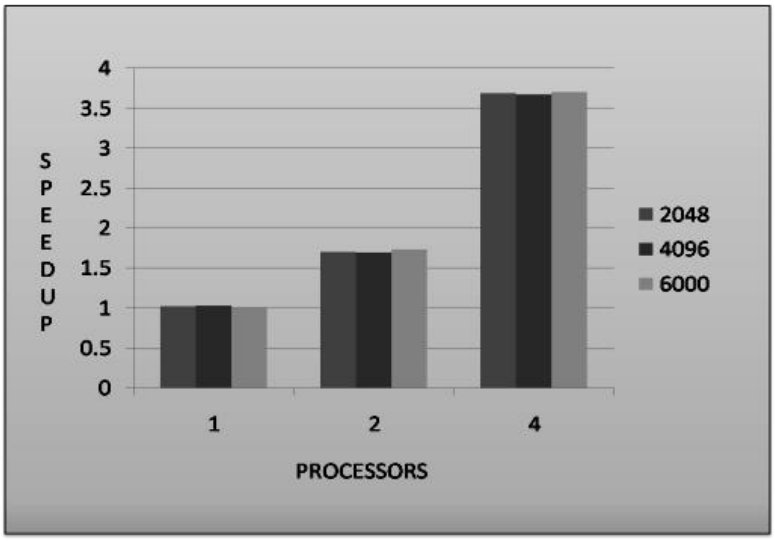

Figura 18 GMRES variando la dimensión de la matriz

Con este experimento se pretende mostrar la eficiencia de los algoritmos que se encargan de efectuar las operaciones de multiplicación $matriz\ x\ vector$ y $vector\ x\ vector$ en paralelo. Para ello se varió la dimensión de la matriz y se analizó si un aumento de la dimensión influía en la escalabilidad del algoritmo. En la Figura 18 se observa que el *speedup* se mantiene constante. Este resultado es el esperado, pues las operaciones paralelizadas son los algoritmos clásicos que sacan gran provecho del paralelismo, y prácticamente todas las operaciones que se hacen en una iteración se basan en estos.

4.4. Comportamiento de la estrategia general

Para analizar la estrategia en su conjunto se corrieron todas las partes integradas variando la dimensión de la matriz y el umbral de truncamiento.

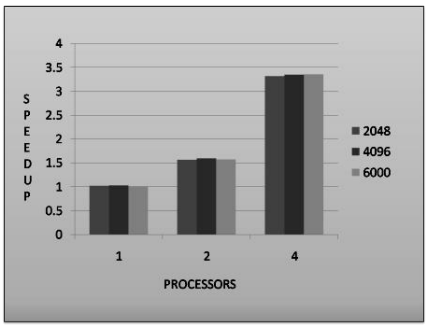

Figura 19 Variación de la dimensión en W_ILU_GMRES

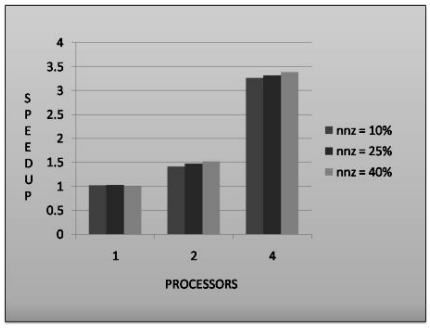

Figura 20 Variación del umbral en W_ILU_GMRES

En las figuras se observa que la obtención del precondicionador ILU (0) no afecta considerablemente el *speedup* de la estrategia general, pues aumentar la dimensión de la matriz o el umbral de truncamiento no supuso una variación de la escalabilidad, lo que mantiene el número teórico de procesadores hasta el cual se obtiene una mejora considerable de la velocidad de ejecución.

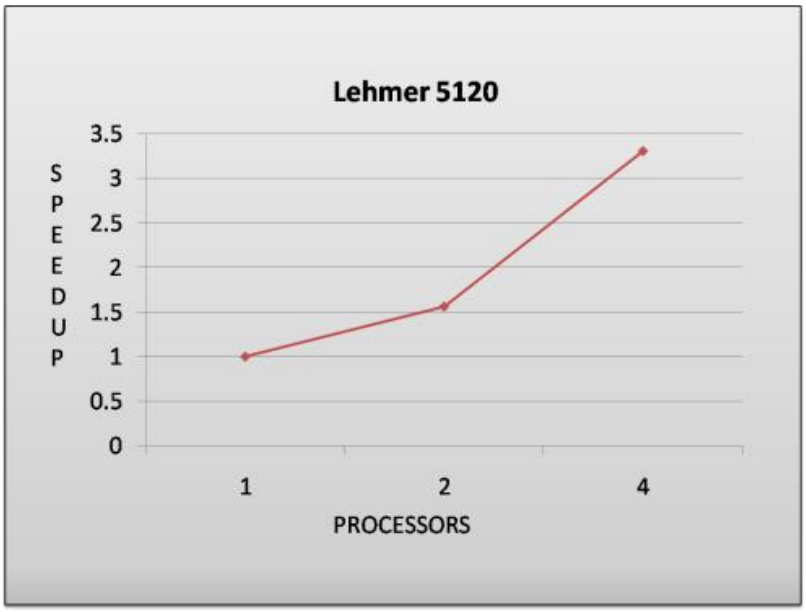

Figura 21 Estrategia general

5. CONCLUSIONES

En el presente trabajo se propuso una implementación de W_ILU_GMRES en paralelo, que supera desde el punto de vista computacional la implementación secuencial presentada en (L. Villarín, A. León, M. L. Baguer, Y. Linares, 2012). Se realizó un diseño modular y reutilizable que permite incorporar otros precondicionadores, otras bases wavelet y/o métodos iterativos.

La implementación en paralelo de la transformada Wavelet, que permitió su aplicación a cada fila y columna de manera totalmente independiente, y la elección de Quicksort para la selección del umbral de truncamiento propiciaron que para dos y cuatro procesadores se alcanzara un *speedup* de aproximadamente 1.7 y 3.8 respectivamente.

En el análisis del precondicionador ILU (0) los resultados ilustran un mejor comportamiento cuando se tiene una mayor cantidad de elementos no nulos en la aproximación dispersa. Se mostró que ILU (0) continúa siendo el nodo crítico de la estrategia, pues afecta la escalabilidad. Con dos procesadores para un 10%, 25% y 40% de elementos no nulos se obtienen *speedup* entre 1.3 y 1.4 y utilizando cuatro procesadores para los mismos porcientos de dispersión el s*peedup* toma valores entre 2.2 y 2.3 aproximadamente. No obstante, se logró minimizar el problema existente en la implementación secuencial donde para disminuir el costo de ILU (0) se necesita una matriz bien dispersa, lo que implica una mayor probabilidad de que la matriz sea singular.

En la evaluación de GMRES en paralelo se obtuvo un *speedup* de aproximadamente 1.7 y 3.7 para dos y cuatro procesadores respectivamente, lo cual es un resultado alentador.

En la estrategia general W_ILU_GMRES en paralelo se obtuvo para dos y cuatro procesadores un *speedup* de aproximadamente 1.6 y 3.5 respectivamente, lo cual indica que la influencia de la obtención del precondicionador ILU (0) no fue trascendental. Los resultados muestran la tendencia a una disminución del tiempo de ejecución con el aumento del número de procesadores, por lo cual sería recomendable experimentar con un incremento de los mismos. Además queda abierta la incorporación de nuevas técnicas de precondicionamiento y la extensión del diseño a una aplicación MPI para contar con una herramienta multiplataforma.

RECEIVED FEBRUARY, 2013
REVISED APRIL, 2014

REFERENCIAS

[1] BARNEY, .(2009): Introduction to Parallel Computing [En línea]. - Enero de 2009. - Marzo de 2010. - **https://computing.llnl.gov/tutorials/parallel_comp/.**

[2]BENZI, M.,W. JOUBERT and G. MATEESCU(1999): Numerical Experiments with parallel orderings for ILU preconditioners . **Electronic Transactions on Numerical Analysis,** 8, 88-114.

[3]CORMEN, T. H. , C. E. RIVEST and R. L. LEISERSON(2009): I**ntroducion to Algorithm**. MIT Press , Cambridge.

[4]CUNHA, R.D.D: and T. HOPKINS(1994): A parallel implementation of the restarted GMRES iterative method for nonsymmetric systems of linear equations. **Advances in Computational Mathematics. - Canterbury** 2,261-277.

[5]ERHEL, J.(1995): A parallel GMRES version for General Sparse Matrices.**Electronic Transactions on Numerical Analysis**. 3, 160-176.

[6]FOSTER, I. (1995): **Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering**. Addison-Wesley, Boston.

[7]GOLUB, G. and C. F. VAN LOAN(1996): **Parallel Matrix Computations**,Third Edition. : Johns Hopkins University Press, Maryland.

[8]HAASE, G., M. LIEBMANN and G. PLANK(2007): A Hilbert-order multiplication scheme for unstructured sparse matrix. I**nternational Journal of Parallel, Emergent and Distributed Systems,** 22, 234-243.

[9]HARROP, J. (2008): **F# for scientist**.Wiley-Interscience, Chichester.

[10]HOSSIAN, S. (2006): **On Efficient storage of Sparse Matrix**. The Proceedings of ICCSE, 2005, 1-7.

[11]MICROSOFT DEVELOPERS(2010): **What's New in Visual F# 2010**. Microsoft Developer Network. - MSDN, msdn.microsoft.com/en-us/library/vstudio/dd553242.

[12]PICKERING, R. (2007): **Foundations of F#** . Apress, California.

[13]SAAD, Y. (2000): **Iterative Methods for Sparse Linear Systems.** (2nd edition), SIAM, Philadelphia.

[14]SYME, D.(2007): **Expert F#** . Apress, California.
[15]VILLARÍN, L., A. LEÓN, M. L. BAGUER, y Y. LINARES (2012): GMRES precondicionado con Wavelets. Un algoritmo de selección del umbral para la obtención del patrón de dispersión .I **Revista Investigación Operacional**, 33, 222-232.