Paralelización de la Factorización LU de Matrices para Clusters Heterogéneos

Trabajo de Grado

Mónica Malén Denham
Nro de legajo: 2347/7
Febrero de 2005

1. Capítulo 1: Introducción 3
1.1 Procesamiento Paralelo 4
1.2 Álgebra lineal 8
1.2.1 EISPACK 9
1.2.2 LINPACK 9
1.2.3 BLAS 10
1.2.4 LAPACK 11
1.2.5 ScaLAPACK 12
1.3 Clusters Heterogéneos 13
1.3.1 Arquitecturas paralelas 13
1.3.2 Clasificación de las arquitecturas 13
1.3.3 Redes locales como computadoras paralelas 15
1.4 Organización del contenido 19
2. Capítulo 2: Álgebra lineal y factorización LU de matrices 21
2.1 Resolución de sistemas de ecuaciones lineales utilizando la Factorización LU 22
2.2 Factorización LU secuencial 23
2.2.1 Ejemplo de la obtención de las matrices L y U y su uso para resolver sistemas de ecuaciones lineales 26
2.3. Factorización LU secuencial por bloques 28
2.4. Factorización LU paralelo por bloques 33
3. Capítulo 3: Clusters Heterogéneos 37
3.1. Clasificación de los clusters 38
3.1.1 Tipo de Aplicación 38
3.1.2 Propiedad del cluster 38
3.1.3 Tipo de nodo 39
3.1.4 Sistema Operativo de los nodos 39
3.1.5 Configuración de los Nodos 39
3.1.6 Niveles de Clusters 39
3.2 Componentes de los clusters 40
3.2.1. Nodos 40
3.2.1.1 Procesadores 41
3.2.1.2 Memoria y memoria caché 41
3.2.1.3. Discos y Entrada/Salida 41
3.2.2. Red de Interconexión 42
3.2.3 Middleware 43
3.2.4 Herramientas de programación paralela 44
3.2.5 Aplicaciones 44
3.3 Clusters Heterogéneos 44
3.3.1 Costo 45
3.3.2 Disponibilidad 47
3.3.2.1 Disponibilidad de clusters como computadora paralela 47
3.3.2.2 Disponibilidad del cluster para realizar cómputo paralelo 48
3.3.3 Evolución de los clusters 48
3.3.4 Mantenimiento 49
3.3.5 Escalabilidad 50
3.3.6 Aplicaciones en clusters heterogéneos 50
Capítulo 4: factorización LU en paralelo 52

4.1 Características de la solución paralela de la Factorización LU 53
4.2 Principios de la distribución de carga 54
4.3 Procesamiento de pivotes 56
4.4 Procesamiento local en cada nodo 56
4.5 Métodos de distribución de carga 59
4.5.1 Cluster Homogéneo 61
4.5.2 Clusters heterogéneos 63
4.5.2.1 Método "directo" 64
4.5.2.2 Método directo mejorado (secuencial) 66
Paso 1 67
Paso 2 67
Paso 3 68
Paso 4 69
4.5.2.3 Método directo mejorado (cíclico) 74
4.5.3 Resumen de los métodos de distribución 78
5. Capítulo 5: Experimentación 80
5.1 Características de las pruebas realizadas 81
5.1.1 Tiempos analizados 81
5.1.2 Configuración del cluster 81
5.1.3 Tamaño de matriz 82
5.1.4 Tamaño de bloque 82
5.2 Benchmarks 83
5.3 Rutina de comunicación 84
5.4 Experimentación 84
5.4.1 Cluster Homogéneo 85
5.4.2 Clusters Heterogéneos 85
5.4.2.1 Conjunto de Prueba 1 86
5.4.2.2 Conjunto de prueba 2 90
5.4.2.3 Conjunto de prueba 3 93
5.5 Resumen 96
6. Capítulo 6: Conclusiones 97
6.1 Problemas de álgebra lineal 98
6.2 Clusters Heterogéneos 99
6.3 Factorización LU de matrices 101
6.3.1 Métodos de distribución 102
A. Apéndice A 106
A.1 Introducción 106
A.2 Distribución Cíclica Con Peso 107
A.3 Método "alternado" 108
Referencias bibliográficas 111

1. Capítulo 1: Introducción

Este capítulo introduce los conceptos básicos de los temas principales en los que se basa este trabajo: cómputo paralelo, álgebra lineal y arquitecturas paralelas. Dentro del álgebra lineal se destaca la importancia de la obtención de rendimiento de cómputo. Por otro lado, además de los conceptos generales de arquitecturas paralelas, se introducen los conceptos más relevantes de los clusters heterogéneos.
Por último, se hace un breve resumen del contenido de cada uno de los capítulos en los que se divide el trabajo.

1.1 Procesamiento Paralelo

Hoy en día, uno de los objetivos de la utilización de la informática como herramienta de trabajo es resolver problemas de forma más precisa y más rápida. Esta búsqueda crea un gran esfuerzo por parte del mundo informático en ofrecer a los usuarios soluciones a problemas que sean cada vez más precisas y más veloces. Esto tiene gran importancia en dominios de aplicación donde las respuestas tardan mucho tiempo (tal vez días) en obtenerse.

Además, se observa una gran tendencia a utilizar la informática como herramienta en áreas cada vez más diversas. Estas tendencias crean la necesidad de brindar al usuario hardware que sea económicamente accesible y fácil de mantener como así también software que corra de forma aceptable en dicho hardware.

Particularmente existen aplicaciones donde se necesita procesar grandes cantidades de datos. Esto es muy común en áreas tales como física, química, tratamiento de imágenes, tratamiento de señales, simulaciones, etc. Para este tipo de aplicaciones el rendimiento es fundamental. En este contexto, cuando se habla de rendimiento significa obtener respuestas más rápidas u obtener mayor cantidad de respuestas en un tiempo determinado.

Para algunas aplicaciones donde se tiene gran requerimiento de cómputo se hace indispensable el uso de computadoras paralelas.
Un ejemplo de esto son las aplicaciones de ASCI (Accelerated Strategic Computing Initiative), del DOE (Department of Energy de los Estados Unidos). Este programa tiene como objetivo desarrollar simulaciones que permitan certificar la seguridad y confiabilidad de reservas nucleares en ausencia de pruebas físicas (desde la prohibición de pruebas sobre o bajo tierra). Para alcanzar el grado de confianza necesaria en estas simulaciones, se desarrollan aplicaciones que requieren plataformas que soporten cómputo en la escala de los tera-flop/s. Obviamente que este programa tiene plataformas que soportan esta escala de cómputo para poder correr dichas aplicaciones [11].

También existen problemas que son naturalmente paralelos, por lo que resolverlos secuencialmente es más difícil y probablemente, una solución paralela sea mejor que una secuencial. Un ejemplo de esto es el especificado en [1], una aplicación de tiempo real: una computadora toma datos del mundo real para realizar determinado cómputo con dichos datos. Recibe n streams de datos independientes al mismo tiempo. Si se dispone de una computadora secuencial, aunque sea de muchísima velocidad, no puede procesar más de un stream a la vez. Además puede pasar que los datos pierdan validez si no se los procesa inmediatamente, por lo tanto, no sirve almacenarlos para procesarlos más tarde. Esto hace que un sólo stream sea útil y el resto tenga información inválida. De esta forma, no se sabe cuál es el stream correcto, y al tener que elegir el próximo stream a procesar, la computadora tiene $1/n$ probabilidades de elegir el correcto y llegar a una solución válida. Por otro lado, una máquina paralela con n procesadores, cada uno dedicado a monitorear un

stream, soluciona el problema de forma correcta, pues los datos provenientes del mundo real son adquiridos de forma correcta y luego procesados para realizar el cómputo deseado.

Desde el punto de vista del usuario, el primer objetivo es obtener la respuesta correcta a su problema. Una vez que se obtiene una respuesta correcta, el tiempo en obtenerla comienza a ser importante. Si el usuario dispone de más de una solución correcta a su problema, seguramente optará por la solución más veloz, la que menos tarde en obtener el resultado. Entonces, aumentar la velocidad de procesamiento es un objetivo que incluye a los desarrolladores de software como así también a los fabricantes de hardware.

Para aumentar la velocidad de procesamiento, se puede optar por dos posibilidades: aumentar la velocidad del procesador utilizado o aumentar la cantidad de elementos de procesamiento. Aumentar la velocidad del procesador sería reducir la velocidad de ejecución de las instrucciones. Por otro lado, aumentar la cantidad de elementos de procesamiento, significa mantener la velocidad del procesador pero aumentar la cantidad de los mismos, lo que implicaría aumentar la cantidad de operaciones realizadas simultáneamente.

Aumentar la velocidad de procesador tiene como ventaja no tener que reprogramar los algoritmos para hacerlos paralelos, a lo sumo, se necesitaría un nuevo compilador, incluso las instrucciones serían las mismas, excepto que son más rápidas. Pero existen las desventajas del costo que implica esta mejora tecnológica (que suele ser elevado) y que esta mejora es limitada. Además, existen problemas que son intrínsicamente paralelos, por lo tanto, programar las soluciones a estos problemas de forma secuencial no es natural.

Por otro lado, aumentar la cantidad de procesadores tiene como ventaja adaptarse a los problemas naturalmente paralelos. Además, no se necesita cambiar la tecnología de los procesadores, pues se utilizan los mismos procesadores pero en mayor cantidad de unidades. Pero se tienen también desventajas: la programación de nuevos algoritmos paralelos o la reescritura de los algoritmos existentes secuenciales implica una desventaja ya que la programación paralela es más compleja que la programación secuencial. Además se tienen los límites dados por la necesidad de comunicación y sincronización de los procesos y la capacidad de paralelización de las aplicaciones [22].

En relación a esta última opción, la firma Intel ha expresado que sus próximos procesadores tendrán 2 o más chips en vez de uno. Estos proveerán mayor rendimiento que los actuales. Este cambio se debe a los problemas de temperatura que existen en los nodos actuales: los circuitos en los procesadores son muy pequeños y consumen mucha energía, por lo que se ha hecho muy difícil (y caro) disminuir la temperatura interna del chip. Altas temperaturas dentro del chip causan problemas en señales y componentes derretidos. Teniendo 2 procesadores y cada uno funcionando a una velocidad moderada, se pretende aumentar el rendimiento y reducir el consumo de energía. Se estima que estos procesadores saldrán a partir del año 2005 (para computadoras de escritorio) y a partir del 2005, 2006 para notebooks [12] [13].

Por todo esto, en las aplicaciones donde se requiere gran cantidad de cómputo, a veces sobre gran cantidad de datos y con un límite de tiempo, es en donde la programación

paralela y las arquitecturas paralelas se convierten en una de las mejores opciones: se utilizan múltiples procesadores para aumentar la capacidad de procesamiento y de esta forma, aumentar el rendimiento. Cada una de las unidades ejecutan distintos procesos, los cuales solucionan de forma cooperativa un problema determinado. Pero para que esto sea posible, los procesos se deben sincronizar y comunicar. O sea, se deben tener en cuenta otros aspectos no incluidos en la programación secuencial para que una solución paralela verdaderamente solucione el problema de forma correcta y realmente aumente el rendimiento [3].

Entonces, las soluciones paralelas desarrolladas incluirán características propias de la programación paralela que no existen en la programación secuencial. Estas características son: sincronización, comunicación, balance de carga, etc. Todos estos factores tienen un papel importante en el cómputo paralelo ya que pueden producir penalizaciones en el rendimiento.

Comparando el software y hardware paralelo con el secuencial, se puede notar que:

- El software paralelo es más difícil de desarrollar, depurar, mantener, entender, etc.
- Una arquitectura paralela es más difícil de construir, mantener, actualizar, etc.

Entonces, para que se justifique el uso del cómputo paralelo, una solución paralela debe lograr mayor rendimiento que una solución secuencial. Para esto, es conveniente (a veces hasta necesario) desarrollar las soluciones desde la perspectiva misma de la arquitectura utilizada. Dicho en otras palabras, se deben plantear las soluciones teniendo en cuenta las características de la arquitectura que se utilizará. De esta forma, se pueden aprovechar algorítmicamente estas propiedades para obtener beneficios a partir de ellas como así también evitar posibles penalizaciones por el uso incorrecto de las mismas. Así, en la etapa de análisis del problema, se estudian las características de la arquitectura a utilizar. Estas características se tienen en cuenta en el diseño y seguramente, determinen características importantes que luego tendrá el algoritmo.

En el contexto del cómputo paralelo, a través de los años se ha observado una tendencia a desarrollar algoritmos específicamente para arquitecturas con características particulares. Como simples ejemplos, se pueden mencionar:

- Algoritmos paralelos particularmente diseñados para computadoras con memoria compartida o computadoras con memoria distribuida.
- Librerías completas de rutinas que se desarrollaron para computadoras paralelas se reescriben para computadoras paralelas con memoria distribuida (como es el caso de LAPACK y ScaLAPACK).
- Los vendedores de procesadores ya están proveyendo librerías optimizadas específicas para los procesadores que desarrollan.

Estos ejemplos muestran que a lo largo de la historia, existe la tendencia de pensar en soluciones específicas para una arquitectura o para un grupo de arquitecturas que comparten características.

Existen laboratorios o centros de investigación importantes a nivel mundial, los cuales tienen la posibilidad de crear arquitecturas específicas para el problema que desean estudiar. En estos casos, se plantea un problema que se desea solucionar, y se llega a la solución de la siguiente manera:

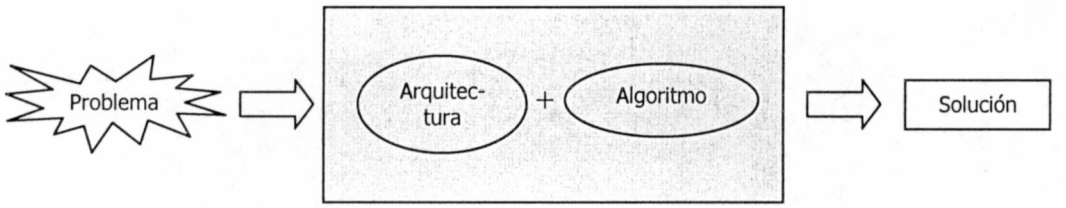

Figura 1.1: dado un problema, se crea una arquitectura y el algoritmo para llegar a la solución del problema.

Dado un problema complejo a solucionar, se desarrolla una arquitectura específica para esto, y los algoritmos para solucionarlo. Con esto, se llega a la solución del problema original. En la figura 1.1 se muestra en gris los componentes que se desarrollan.

Esta situación se puede observar repetidas veces en el [25]. Como algunos ejemplos se pueden mencionar:

1) Earth Simulator: esta supercomputadora fue desarrollada por el Earth Simulator Center, en Japón, y fue desarrollada para simular ciertos fenómenos de la tierra. Con ella se busca predecir sucesos futuros, modelando condiciones actuales en base a datos sobre el pasado.

2) Los sistemas Blue Gene/L de IBM han sido creados originalmente para estudiar el proceso biológico de plegamiento de proteínas.

3) En el caso de la supercomputadora ASCI Q, de Los Alamos National Laboratory, pertenece al programa Advanced Simulation and Computing (ASCI). El objetivo es simular el comportamiento de armas nucleares. Esta necesidad surge a partir de la prohibición de pruebas nucleares sobre o bajo tierra. Para esto, se necesita crear computadoras paralelas lo suficientemente rápidas y altamente confiables.

Estos son sólo algunos ejemplos que muestran la situación en estos centros de investigación: a partir de un problema específico, se crea la arquitectura adecuada para este problema y el algoritmo (seguramente los algoritmos) que lo solucionen.

Muy distinto es en países donde no se dispone de la posibilidad de crear una arquitectura específica para un problema dado. En este caso, la situación es muy distinta y se ilustra en la figura 1.2.

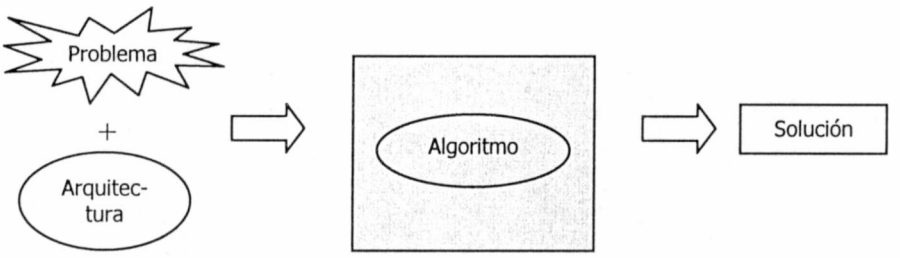

Figura 1.2: dado un problema y una arquitectura en particular, se crea el algoritmo que solucione el problema.

En estos casos, se tiene una arquitectura específica, la cual es la única que se dispone o si se dispone más de una, la arquitectura que más convenga para solucionar el problema. En estos casos, donde la arquitectura es "fija" (no se pueden cambiar aspectos de la misma para adaptarlos al problema), el algoritmo es el que se debe adaptar a la arquitectura específica a utilizar.

En esta situación, donde no se puede cambiar la arquitectura ni características de la misma, es conveniente tener en cuenta las características más relevantes de la arquitectura en las etapas de análisis, diseño y desarrollo, así se aprovechan de la mejor forma posible como así también se evitan penalizaciones que las mismas puedan llegar a producir.

Particularmente en este trabajo, se mantiene esta tendencia, y se tienen en cuenta varias características de la arquitectura a utilizar. Estas características influyen y determinan características importantes del algoritmo. Las más relevantes son: nodos de cómputo e interconexión de los mismos.

Una consecuencia de todo esto es que las soluciones desarrolladas sean más específicas: lo que implica que seguramente funcionen bien sobre arquitecturas para las que fueron desarrolladas. Pero no funcionarán de forma correcta (o no se obtendrá buen rendimiento) en arquitecturas donde dichas características no se encuentran presentes. O sea, los algoritmos son menos genéricos y más orientados a una arquitectura específica.

1.2 Álgebra lineal

Dentro del cómputo científico, un área importante es la de álgebra lineal. Las operaciones que pertenecen a ésta área se utilizan en dominios tales como astronomía, física, medicina, simulaciones, etc. Es común que este tipo de problemas requieran mucho procesamiento sobre los datos, y a su vez, operen con una gran cantidad de datos.

La obtención de soluciones optimizadas a estos problemas implica la mejora del tiempo de respuesta en los problemas que utilizan o se basan en estas operaciones de álgebra lineal. Por lo tanto, optimizar este grupo de operaciones trae como consecuencia la mejora en los tiempos de respuestas en un amplio rango de problemas.

Existen muchas aplicaciones donde se utilizan operaciones de álgebra lineal. Dentro de las operaciones de álgebra lineal, la resolución de sistemas de ecuaciones gana importancia por su amplio uso. Algunos ejemplos en que se utiliza resolución de sistemas de ecuaciones son [17]:

- Diseño de alas de aeroplanos.
- Estudios de radares.
- Como benchmark para supercomputadoras.
- Flujos alrededor de naves y otras construcciones.
- Difusión de cuerpos sólidos en líquidos.
- Reducción de ruidos.
- Difusión de luz a través de pequeñas partículas.

Esta operación es utilizada como benchmark de supercomputadoras en el ranking conocido como TOP500 [25]. En el mismo se encuentran las 500 computadoras con mayores prestaciones de cómputo. De cada una de ellas se tiene una breve descripción y los resultados de rendimiento obtenidos. La operación utilizada como benchmark resuelve un sistema de ecuaciones lineales. Se utiliza este benchmark dado su amplio uso y porque se dispone de las medidas de rendimiento utilizando esta operación de los sistemas más relevantes. El rendimiento obtenido es el rendimiento pico del sistema ya que el mismo se utiliza de forma dedicada para esta medición La versión que se utiliza como benchmark permite elegir el tamaño del problema y optimizar el software utilizado para lograr el mejor rendimiento del sistema. Esta lista se puede encontrar en la dirección especificada en [25].

Desde ya hace unos años se han desarrollado distintas bibliotecas con operaciones que se utilizan en aplicaciones de álgebra lineal. Ejemplos de estas librerías son: EISPACK, LINPACK, LAPACK, ScaLAPACK y BLAS.

1.2.1 EISPACK

EISPACK (Eigenvalue Calculations) es una colección de subrutinas implementadas en Fortran que computan los autovalores y autovectores de 9 clases distintas de matrices: generales y hermíticas de números complejos; generales, simétricas y simétricas en banda de números reales; tridiagonales simétricas y tridiagonales especiales de números reales; generalizadas y matrices simétricas generalizadas de números reales. También se incluyen 2 rutinas que usan la descomposición de valor singular para resolver problemas de mínimos cuadrados.

1.2.2 LINPACK

A su vez, LINPACK (Linear Algebraic Equations) es una colección de rutinas (también implementada en Fortran), que analizan y resuelven ecuaciones lineales y problemas de mínimos cuadrados lineales. Esta librería tiene rutinas que solucionan sistemas lineales con matrices generales, en banda, simétricas indefinidas, simétricas positivas definidas, triangulares y tridiagonales cuadradas.

LINPACK está basada en cuatro factorizaciones de matrices: LU, Cholesky, QR y descomposición de valor singular (SVD).

1.2.3 BLAS

BLAS (Basic Linear Algebra Subroutines) es un conjunto de operaciones básicas de álgebra lineal, a partir de las cuales se puede definir todo LAPACK. Estas rutinas están divididas en 3 niveles, dependiendo del tipo de operandos que participan en la operación. Los tipos de operandos también determinan el requerimiento de cómputo y de memoria necesaria por cada operación.
Los niveles son:

- Nivel 1 (Level 1 BLAS o L1 BLAS): estas son operaciones entre vectores.
- Nivel 2 (Level 2 BLAS o L2 BLAS): son las operaciones en donde participan matrices y vectores.
- Nivel 3 (Level 3 BLAS o L3 BLAS): en donde las operaciones se realizan entre matrices.

Las operaciones en que se ha puesto más énfasis y esfuerzo en su optimización son las pertenecientes al Nivel 3 de BLAS. Esto se debe a que son las de mayor requerimiento de cómputo como así también de memoria para su almacenamiento. Estas operaciones operan con matrices y, si n es el orden de las matrices, realizan n^3 operaciones sobre n^2 datos.

Las operaciones definidas dentro de BLAS Level 3 son [19]:

a) Productos de matrices generales.

$C \leftarrow \alpha\, op(A)\, op(B) + \beta C$ (op(X) es X, X^T, X^H)

b) Productos de matrices donde una de ellas es real o compleja simétrica o compleja hermética.

$C \leftarrow \alpha AB + \beta C$ o $C \leftarrow \alpha BA + \beta C$ (con A simétrica o hermítica)

c) Productos de matrices donde una de ellas es triangular.

$B \leftarrow \alpha\, op(A)\, B$ o $B \leftarrow \alpha\, B\, op(A)$

(con A triangular y op(A) es A, A^T, A^H)

d) Actualizaciones de rango k (rank-k update) de una matriz simétrica.

$C \leftarrow \alpha AA^T + \beta C$ o $C \leftarrow \alpha A^T A + \beta C$ (con C simétrica)

e) Actualizaciones de rango k (rank-k update) de una matriz hermética.

$C \leftarrow \alpha AA^H + \beta C$ o $C \leftarrow \alpha A^H A + \beta C$ (con C hermítica)

f) Actualizaciones de rango $2k$ (rank-2k update) de una matriz simétrica.

$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$ o $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$

(con C simétrica)

g) Actualizaciones de rango $2k$ (rank-2k update) de una matriz hermética.

$C \leftarrow \alpha AB^H + \alpha BA^H + \beta C$ o $C \leftarrow \alpha A^H B + \alpha B^H A + \beta C$

(con C hermítica)

h) Soluciones a sistemas de ecuaciones triangulares.

$C \leftarrow \alpha\ op(A)\ B$ o $C \leftarrow \alpha\ B\ op(A)$ (con A triangular) .

Vale la pena destacar que todas estas operaciones se pueden basar en la multiplicación de matrices, por lo tanto, disponer de una versión optimizada de la misma, implica tener buen rendimiento en todas las operaciones de este nivel [19].

1.2.4 LAPACK

La librería LAPACK (Linear Algebra PACKage) provee rutinas para solucionar sistemas de ecuaciones lineales simultáneas, soluciones del problema de mínimos cuadrados de sistemas de ecuaciones lineales, problemas de autovalores, y problemas de valor singular. También provee factorizaciones de matrices tales como LU, Cholesky, QR, SVD, Schur y Schur generalizada.

LAPACK trabaja con matrices densas y en bandas, pero no trabaja con matrices generales esparcidas o ralas (sparse). En todas las áreas, se provee la misma funcionalidad para matrices reales y complejas, con precisión simple y doble.

El objetivo de LAPACK fue hacer que las librerías EISPACK y LINPACK corrieran eficientemente en computadoras vectoriales y paralelas con memoria compartida. Para esto, LAPACK redefine los algoritmos en base a bloques (las librerías anteriores no lo hacían). [17].

La librería LAPACK fue diseñada para ser eficiente y transportable en un amplio rango de computadoras, principalmente en las computadoras de alto rendimiento (high performance). Además, se pretende que esta librería sea ampliamente disponible o sea, que el usuario pueda disponer de estas rutinas de forma fácil y rápida [2].

Dentro de esta librería, las rutinas se clasifican en [15]:

- Rutinas "driver": estas rutinas solucionan un problema completo, por ejemplo, solucionar un sistema de ecuaciones o computar los autovalores de una matriz real simétrica. Comúnmente, los usuarios utilizan estas rutinas en sus aplicaciones.

- Rutinas computacionales: realizan tareas computacionales, por ejemplo, la factorización LU. Cada rutina driver invoca una secuencia de rutinas computacionales. Existen algunos usuarios (especialmente desarrolladores de software) que invocan directamente rutinas (o secuencias de rutinas) computacionales para realizar sus tareas en vez de usar rutinas driver.

- Rutinas Auxiliares: a su vez se clasifican en:

  - Rutinas que realizan subtareas de algoritmos por bloques, en particular, rutinas que implementan versiones sin bloques (unblocked) de los algoritmos.
  - Rutinas que realizan cómputos comunes de bajo nivel, por ejemplo, computar la normal de una matriz. Algunas de estas rutinas, se podrían considerar para agregarlas a BLAS en un futuro.
  - Una pequeña extensión de BLAS, como operaciones entre vectores y matrices incluyendo matrices reales simétricas.

1.2.5 ScaLAPACK

La librería ScaLAPACK (Scalable LAPACK) extiende LAPACK para computadoras paralelas con memoria distribuida. En este tipo de computadoras, a la jerarquía de memoria de cada una de las computadoras (registros, caché y memoria local) se le agrega la memoria de los demás procesadores.

En ScaLAPACK, los algoritmos también están definidos por bloques, para evitar la transmisión de datos entre los distintos niveles de la jerarquía de memoria (esto se explicará con más detalle más adelante en este mismo trabajo). Esta librería incluye versiones para memoria distribuida de las operaciones de los niveles 2 y 3 de BLAS y un conjunto de subrutinas de comunicación (BLACS, Basic Linear Algebra Communication Subprograms).

Las interfaces de las rutinas definidas en LAPACK y ScaLAPACK son similares. Esto es posible gracias a que las comunicaciones entre las tareas ocurren entre rutinas de BLAS y BLACS, y no entre rutinas de nivel de aplicación.[17].

En este trabajo se presenta una solución a la factorización LU. En la librería LAPACK existe una subrutina que implementa dicha factorización. Dentro de la clasificación hecha en LAPACK, ésta es una rutina computacional. Es utilizada para resolver sistemas de

ecuaciones lineales. Dentro de la librería LAPACK existen otras factorizaciones, algunas de las cuales tienen el mismo patrón de procesamiento que la factorización LU. Por lo tanto, los resultados del trabajo puesto en esta factorización, pueden adaptarse y utilizarse en otras factorizaciones (por ejemplo, factorización QR, Cholesky, etc).

Si el orden de la matriz es n, entonces la factorización realiza cálculos de orden n^3 sobre n^2 datos. Por lo tanto, es del tipo de operación de álgebra lineal que tiene el máximo de requerimiento de cómputo (n^3) y el máximo de requerimiento de almacenamiento (n^2). Sobre operaciones que tienen este máximo de requerimientos es donde se observa una mayor ventaja en su optimización. Esto es debido a que se pueden optimizar desde varios puntos de vista: uso de memoria (mejora en el uso de la jerarquía de memoria instalada en las computadoras), el tipo de pasaje de mensajes (utilizando el tipo de mensaje broadcast), realizando más de una operación por ciclo de reloj (procesadores superescalares).

Estas operaciones suelen definirse en términos de bloques o submatrices. De esta forma, se divide la matriz (o matrices) en bloques y se resuelve la operación utilizando cada uno de estos bloques. El objetivo es llevar un bloque a memoria y realizar la mayor cantidad de operaciones posibles sobre estos datos antes de cambiar de bloque. De esta forma, se optimiza el uso de los distintos niveles de memoria instalados en las computadoras (que generalmente se incluye algún nivel de memoria caché en esta jerarquía).

1.3 Clusters Heterogéneos

1.3.1 Arquitecturas paralelas

Las computadoras o arquitecturas paralelas, disponen de más de una unidad de procesamiento. A su vez, los programas paralelos están formados por más de una tarea, las cuales cooperan y trabajan juntas para resolver un problema. Entonces, es posible ejecutar cada una de las tareas en distintos procesadores. De esta forma, se resuelven distintas partes del problema simultáneamente.

Además, las tareas que están trabajando de forma conjunta para resolver un problema seguramente necesiten comunicarse (para compartir datos), o sincronizarse (para esperar un estado específico), etc. Entonces, las soluciones paralelas requieren una reescritura de las soluciones secuenciales para un problema específico.

Existen diversas arquitecturas paralelas y las características que pueden variar entre las distintas arquitecturas son: tipo de nodos, cantidad de nodos, forma en que están conectados, tipo de memoria, tipo de conexión a la memoria, etc.

1.3.2 Clasificación de las arquitecturas

Existen diversas arquitecturas paralelas, y a lo largo de la historia, se han hecho distintas clasificaciones. Una clasificación muy utilizada es la de Flynn. En la misma, los distintos tipos de computadoras se clasifican según cómo ejecutan las instrucciones sobre los datos. Las computadoras monoprocesadores también están incluidas en esta clasificación.

Toda computadora sea monoprocesador o multiprocesador, está compuesta por distintos componentes. Entre las distintas arquitecturas, varía la cantidad de cada uno de estos componentes, cómo están conectados, su funcionamiento, etc.
Los principales componentes son:

- Unidad de control: la función de esta unidad es decodificar las instrucciones y enviarle la información respectiva a la (las) unidad (unidades) de procesamiento.
- Unidad de procesamiento: su función es ejecutar las instrucciones sobre los datos.
- Unidad de memoria: esta unidad almacena los datos y las instrucciones.

La clasificación de Flynn distingue 4 clases:

1) SISD: Single Instruction stream, Single Data stream: estas son las computadoras secuenciales comunes, donde las instrucciones se ejecutan una después de la otra. En estas computadoras existe una sola unidad de procesamiento, una única unidad de control y una única unidad de memoria.

2) MISD: Multiple Instruction stream, Single Data stream: esta opción es más bien teórica, ya que en la realidad no es muy utilizada. Múltiples unidades de procesamiento realizan distintas instrucciones sobre un mismo dato.

3) SIMD: Single Instruction stream, Multiple Data stream: en esta clase existen múltiples unidades de procesamiento y una única unidad de control. Se ejecuta la misma instrucción en las distintas unidades de procesamiento sobre distintos datos. Las distintas unidades de procesamiento se comunican mediante una red de interconexión o mediante memoria compartida. Además, cada unidad de procesamiento puede tener memoria local a la cual accede de forma exclusiva.

4) MIMD: Multiple Instruction stream, Multiple Data stream: existen distintos procesadores y cada uno tiene su propia unidad de control. Cada procesador ejecuta su propia secuencia de instrucciones sobre datos que pueden ser distintos en cada uno de los procesadores. Las arquitecturas MIMD son las que se consideran de propósito general.

En las arquitecturas MIMD, puede variar cómo se conectan los procesadores entre sí (su topología), y cómo se conectan los procesadores con la memoria.
Las arquitecturas MIMD son las que más variedad permiten ya que se pueden elegir distintas topologías y distinta forma de conexión a la memoria. Esto permite una nueva subclasificación:

1) Multiprocesadores: todos los procesadores comparten una única memoria, el espacio de direccionamiento es el mismo para todos los procesadores. La forma de comunicación entre los procesadores es a través de la memoria compartida, y para evitar interferencia entre los procesadores, el acceso a la memoria es sincronizada. A este tipo de computadoras se las conoce como fuertemente acopladas (tightly coupled machines).

2) Multicomputadoras: cada procesador tiene su unidad de control y su memoria. Los distintos procesadores están conectados a través de una red de interconexión, mediante la cual se comunican enviándose mensajes. También se conoce este tipo de arquitecturas como computadoras débilmente acopladas (losely coupled machines).

Otra forma de clasificar las plataformas paralelas es la propuesta en [19]. En dicha clasificación se incluyen las plataformas paralelas actuales y se tiene en cuenta la capacidad y el costo de las mismas. Esta clasificación es útil ya que los parámetros usados para dicha clasificación (capacidad y costo), son los que casi en todos los casos determinan la disponibilidad de las mismas.

En esta clasificación, se ordenan las distintas arquitecturas en una pirámide, donde, a medida que se sube de nivel, las arquitecturas tienen mayor capacidad y aumenta también el costo. El dibujo en forma de pirámide también permite una lectura que corresponde a la disponibilidad de las mismas, que se corresponde directamente con el costo: cuanto más costosas, menor es la disponibilidad.
En la base de la pirámide, se encuentran las LAN, o sea, redes locales de computadoras utilizadas como arquitectura paralela (clusters heterogéneos). Estas arquitecturas son las de mayor disponibilidad por ser las más económicas.

Según la primer clasificación, un cluster es una arquitectura MIMD y específicamente, es una multicomputadora (ilustrado en la Figura 1.3).

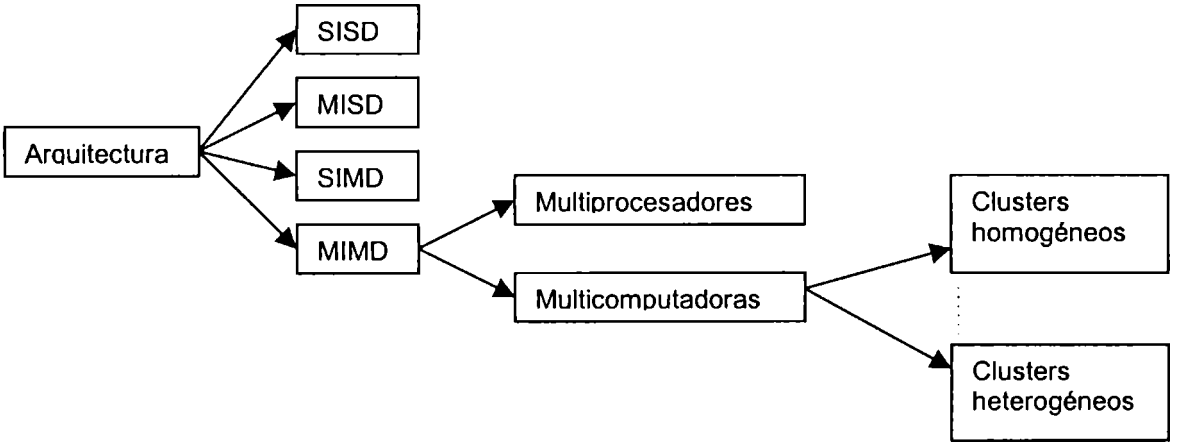

Figura 1.3: ubicación de los clusters heterogéneos dentro de la clasificación de Flynn

Según la clasificación propuesta en [19], los clusters heterogéneos son la arquitectura de menor capacidad pero las de mínimo costo.

1.3.3 Redes locales como computadoras paralelas

Las computadoras paralelas, logran buen rendimiento y altas prestaciones en diversos dominios. Un ejemplo de estas computadoras son las llamadas "supercomputadoras", las cuales son computadoras con máximas prestaciones en un momento dado.

Pero lamentablemente el precio de las computadoras paralelas es muy alto, haciendo que la adquisición y uso de dichas máquinas sea difícil [19].

Por otro lado, se puede observar que cada vez hay más redes locales instaladas y en uso. Estas redes locales están formadas por computadoras de escritorio comunes interconectadas por una red Ethernet.

Gracias al avance tecnológico y al mercado de consumo, la potencia de las computadoras de escritorio es cada vez mayor. Estas computadoras cada vez tienen más capacidad de procesamiento y de almacenamiento. Además son las computadoras más accesibles económicamente. A pesar de que su tecnología avanza día a día, el precio se mantiene constante en el mercado (o al menos son "constantes" en ser siempre las más económicas).

Para que estas computadoras estén conectadas en red, se necesita que cada una tenga una placa de red y el cableado normal de una red Ethernet (se suelen incluir switches o hubs). Estos componentes son masivamente usados y fáciles de hallar en el mercado a un costo accesible. Estas características hacen que cada vez haya más redes locales instaladas y en funcionamiento.

Por esto, es cada vez más normal pensar en utilizar una red de computadoras de escritorio para formar una computadora paralela. Dicho de otra manera, armar una computadora paralela con buenas prestaciones pero a un precio accesible. Este tipo de arquitectura paralela se la conoce como cluster. Un cluster es homogéneo cuando las computadoras que lo forman son todas iguales (todas con la misma capacidad de cómputo). En el caso contrario, el cluster es heterogéneo.

El bajo costo de los clusters heterogéneos implica las ventajas de:

- Alta disponibilidad: en la actualidad, existen cada vez más redes locales instaladas y en funcionamiento.
- Ampliamente escalable: la capacidad del cluster se puede ampliar simplemente agregando computadoras de cualquier tipo.
- El mantenimiento de la red es implícito: si la red está en funcionamiento, existen personas que se encargan del mantenimiento de la misma, por lo tanto, el funcionamiento de la máquina paralela es intrínseco al de la red.
- El mantenimiento del hardware es sencillo: como el cluster está formado por computadoras de escritorio comunes, cuando una computadora deja de funcionar, simplemente se la reemplaza por otra.
- Aumentar la capacidad del cluster es sencillo: cuando se desea actualizar el cluster simplemente se agrega una computadora cualquiera (como en el caso del mantenimiento).

La mayoría de estas ventajas no se encuentran presentes en otras arquitecturas paralelas. Esto se debe principalmente al tipo de nodo y tipo de red: computadoras comunes de cualquier tipo y capacidad y la red de interconexión es una red Ethernet común.

Por ejemplo, en el caso de los clusters homogéneos, donde todas las computadoras tienen la misma capacidad, puede suceder que:

- Un nodo del cluster deja de funcionar o no lo hace de forma correcta.
- Se desea aumentar la capacidad del cluster.

En estos casos, se utilizará una computadora nueva la cual pasará a formar parte del cluster. Pero esta computadora necesariamente debe ser igual a la que se desea reemplazar o a las que ya forman parte del cluster para mantener su homogeneidad. Esto es difícil ya que la tecnología de las computadoras de escritorio cambia en muy poco tiempo, por lo que encontrar una computadora igual a las que se tenía resulta una tarea difícil. Si encontrar el mismo tipo de procesador no es posible y se necesita agregar un nodo nuevo, el cluster pasará a ser heterogéneo.

Estas situaciones muestran que es difícil mantener un cluster homogéneo a través del tiempo. Esto también sucede en otras arquitecturas paralelas además de los clusters homogéneos. En cualquier computadora específicamente paralela, si se necesita agregar un procesador seguramente este procesador deberá ser igual al que se está reemplazando.

Esto no sucede en el caso de los clusters heterogéneos, ya que agregar un nodo implica agregar una computadora de escritorio cualquiera, sin importar la capacidad de la misma.

Pero esta arquitectura también tiene desventajas: las redes de computadoras no han sido diseñadas ni creadas para cómputo paralelo. Por lo tanto, hay ciertas características que se deben tener en cuenta en el momento del desarrollo de soluciones paralelas. Una de las más importantes es el tipo de red de interconexión. Una red Ethernet es una red de varios órdenes de magnitud más lenta que una red de interconexión utilizada en una máquina específicamente creada para cómputo paralelo. Otra característica importante es la capacidad de los nodos: estas capacidades pueden ser tan diversas como la diversidad actual de computadoras de escritorio.

Además, se debe tener en cuenta que la red necesitará herramientas para desarrollo y ejecución de programas paralelos. En la actualidad existen librerías de comunicación, algunas de las cuales son libres y se consiguen a través de Internet. Ejemplos de estas librerías son: PVM (Parallel Virtual Machine), o implementaciones de MPI (Message Passing Interface). Además son necesarias herramientas de administración de redes de computadoras para cómputo paralelo.

También existe el problema de disponibilidad de uso. Si una red está instalada, seguramente tenga usuarios que hacen uso de la red. Por lo que la disponibilidad de las computadoras no es completa. Entonces, se debería utilizar las redes específicamente para cómputo paralelo en tiempos ociosos de las mismas. Para esto, es necesario informarse

sobre el uso de la red, para determinar la disponibilidad de la misma para correr programas paralelos.

Determinar los momentos propicios para utilizar estas redes para cómputo paralelo es importante, ya que si la red está en uso, los usuarios no deberían verse perjudicados por este nuevo uso que se hace de la red.

Cuando estas redes están en ámbitos laborales o académicos (oficinas, negocios, facultades, laboratorios, etc) es muy normal que tengan poco uso los días no laborables o en los horarios nocturnos, por lo que existe la posibilidad de aprovechar estos momentos para realizar cómputo paralelo y que los usuarios dispongan de la red como lo hacían anteriormente a su uso como máquina paralela [19].

Se reconoce que aprovechar tiempos ociosos de los recursos que componen una red de computadoras implica una gran ventaja y mejora del uso de los mismos [4]. Existen proyectos tales como Condor que trabajan en este sentido. Condor es un sistema que busca obtener alta productividad de los sistemas aumentando la cantidad de trabajo realizado en un lapso de tiempo determinado. Para lograrlo, se busca hacer un buen uso de los recursos disponibles en el sistema.

Existen problemas con mucha cantidad de cómputo por períodos de tiempo largos. Este tipo de ambientes son llamados High-Throughput Computing (HTC) y lo importante es cuántos trabajos se pueden completar en un período de tiempo largo.

Es común encontrar redes de computadoras instaladas y que estas redes tengan diversos usuarios. Mientras estos usuarios no utilizan la red, la misma permanece ociosa (o por lo menos algunos recursos de la misma). Condor toma estos tiempos mal usados y los utiliza para realizar alguna tarea. En definitiva, Condor alcanza mayor productividad haciendo trabajar a computadoras ociosas.

Cuando se utiliza Condor, los programas no necesitan ser reescritos. Solamente se linkean con las librerías de Condor y de esta forma, los programas son capaces de realizar checkpoints y de realizar llamados al sistema remotos. Un checkpoint es el conjunto de toda la información de un proceso en ejecución. Condor realiza checkpoints de los programas en ejecución, entonces, si el programa debe ser interrumpido (porque la máquina donde se está ejecutando no está más ociosa o porque dicha máquina tiene que ser reboteada por alguna razón), cuando se puede seguir ejecutando (luego de una posible migración) se hace desde donde se interrumpió la ejecución y no es necesario recomenzar el trabajo.

El usuario agrega tareas a Condor y éstas se ejecutan en alguna de las computadoras disponibles por Condor. Los llamados al sistema son realizados por Condor y éste envía la información al proceso que la realiza [16].

El objetivo del proyecto Condor es aumentar la productividad. También existe mucho trabajo en aumentar el rendimiento, o sea, aumentar la cantidad de cómputo en un corto

período de tiempo. En este trabajo, se busca aumentar el rendimiento de ciertas operaciones de álgebra lineal para una arquitectura determinada.

Existe software que permite construir computadoras paralelas virtuales integrando un conjunto de computadoras conectadas en red (como es el caso de PVM). Aunque los nodos de estas computadoras virtuales y la red subyacente no han sido diseñadas para optimizar operaciones paralelas, se pueden alcanzar aumentos en el rendimiento si las aplicaciones paralelas se conciben directamente para este ambiente específico [4].

En resumen, las características que tienen estas redes hacen que sean una arquitectura recomendable para ser utilizadas para cómputo paralelo. Sin embargo hay que tener en cuenta todas sus características para que su uso en el ámbito del cómputo paralelo sea, al menos, aceptable. Y el objetivo de todo el trabajo puesto en este tema, es para que no sea sólo aceptable, sino también óptimo.

En la actualidad se observa una fuerte tendencia a utilizar CLUMPS (clusters formados por computadoras SMP) o clusters homogéneos. Aunque la tendencia sea otra, en este trabajo se utilizan clusters heterogéneos y se desarrollan algoritmos para dicha arquitectura.

En este trabajo se propone una solución a la factorización LU de matrices para ser utilizado en clusters heterogéneos.

1.4 Organización del contenido

En el capítulo 2 se muestra la factorización LU de matrices. Se muestra una solución secuencial del problema, y luego una solución secuencial utilizando bloques. Luego, se muestra una solución paralela por bloques tradicional a la factorización LU orientada a un conjunto de procesadores dispuestos en forma de grilla bidimensional. Además se describen 2 formas distintas de distribuir los bloques entre los nodos, distribución secuencial y cíclica.

En el capítulo 3 se describe la arquitectura utilizada, analizando primero las principales características de los clusters en general y luego se detallan las características específicas de los clusters de PCs heterogéneos. Dentro de las características específicas de los clusters heterogéneos se describen con mayor detalle las propiedades que pueden llegar a afectar en el rendimiento de una aplicación, por lo que es conveniente tenerlas en cuenta a la hora de desarrollar una aplicación.

En el capítulo 4, se describirá la solución paralela utilizada en este trabajo. Esta solución tiene en cuenta las principales características de los clusters heterogéneos desde el momento mismo del diseño. Se mostrará el algoritmo implementado (en forma de pseudocódigo). Aunque la solución a la factorización LU siempre es la misma (pues es una solución al método que ya ha sido probada y demostrado que es buena), se proponen distintos métodos para la distribución de datos entre los nodos. Estos distintos métodos de distribución se plantean con el objetivo de lograr balance de carga.

En el capítulo 5 se muestran los resultados de las experimentaciones realizadas. Para cada una de ellas se especifican las características del cluster utilizado, los distintos tamaños de matrices y se grafican los tiempos obtenidos realizándose un breve análisis de los mismos. En la mayoría de los casos, los comportamientos observados en los tiempos obtenidos permiten realizar mejoras sobre el método de distribución.

En el capítulo 6 se muestran las conclusiones a las que se llegaron luego de realizar este trabajo. Este trabajo abarcó principalmente 4 temas que fueron ejes: álgebra lineal, clusters heterogéneos, factorización LU paralela y métodos de distribución de carga. De cada uno de estos temas se realizó una investigación para conocer su historia y cuál es su estado actual. Se pudieron ver distintos problemas, soluciones, propuestas, etc. A partir de todo esto, se implementó una solución a la factorización LU y distintos métodos de distribución de carga. Esta solución y los distintos métodos se probaron y se analizaron. Luego, todas las conclusiones que se han podido sacar en base a esto se especifican en el capítulo 6.

En el Apéndice A se describen 2 formas más de distribuir los datos. La primera se ha implementado y probado, pero no dio ningún resultado satisfactorio, por lo tanto, no se han tomado tiempos con la misma y no se incluye como posible método. La otra es una mejora a los métodos propuestos la cual todavía no ha sido implementada, por esto no se incluye en los capítulos 4 y 5. Este método forma parte de los pasos futuros que tiene este trabajo.

2. Capítulo 2: Álgebra lineal y factorización LU de matrices

Como ya se ha mencionado, la factorización LU es un método de álgebra lineal y es utilizado para resolver sistemas de ecuaciones lineales. Este método está definido en la librería LAPACK y dentro de esta librería está definido como una rutina computacional.

Existen otras factorizaciones con el mismo patrón de procesamiento que la factorización LU. Por ejemplo, la factorización QR con el método de Householder, aplica iterativamente reflexiones de Householder, de la misma forma en que la factorización LU aplica eliminaciones Gaussianas [20].

La factorización Cholesky también tiene puntos en común con la factorización LU: el patrón de comunicación de la factorización Cholesky es similar al de la eliminación Gaussiana, excepto que la factorización Cholesky trabaja sólo con una mitad triangular de la matriz, dado que esta factorización se aplica a matrices simétricas definidas positivas [14].

Luego, las ventajas de la optimización de la factorización LU son:

- Distintos principios de una solución optimizada de la factorización LU pueden ser utilizados en otras factorizaciones. De este modo, optimizando este método, se optimizan las otras factorizaciones de una forma casi directa.

- Dentro de la librería LAPACK, existen otras rutinas que invocan a la factorización LU (rutinas driver), entonces optimizando este método, se mejorará el rendimiento de todas las rutinas de LAPACK que la invoquen.

- Dado el amplio uso de sistemas de ecuaciones lineales en distintos campos de aplicación, disponer de una solución optimizada de esta operación implicaría directamente una mejora en todas las aplicaciones que la utilicen.

Luego, las ventajas enumeradas anteriormente y las ventajas que implica usar clusters heterogéneos para cómputo paralelo, son suficientes para intentar obtener una versión optimizada de la factorización LU para la arquitectura mencionada.

2.1 Resolución de sistemas de ecuaciones lineales utilizando la Factorización LU

La factorización LU se basa en aplicar pasos de eliminación gaussiana sobre los elementos de una matriz cuadrada. Esta matriz está formada por los coeficientes de las ecuaciones del sistema de ecuaciones original que se desea resolver.

Un sistema de ecuaciones triangular es fácil de resolver. Entonces, la idea de la eliminación gaussiana es convertir un sistema del tipo

$$Ax = b \tag{2.1}$$

en un sistema triangular equivalente. Los elementos de este sistema triangular equivalente son los que se utilizan para obtener los valores de las incógnitas representados por x en la ecuación 2.1. Este sistema de ecuaciones equivalente se puede obtener realizando combinaciones lineales a los elementos de las ecuaciones originales [10].

Por ejemplo, suponiendo que L es una matriz triangular inferior, resolver un sistema de ecuaciones del tipo:

$$Lx = b \tag{2.2}$$

y suponiendo que L es de 3x3 elementos, existen 3 incógnitas y 3 resultados, el sistema de ecuaciones triangular es de la forma:

$$\begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} x \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

siendo todos los elementos de la diagonal principal distintos de cero (l_{ii} 0). De esta forma, encontrar los valores de las incógnitas es resolver:

$$l_{11} x x_1 = b_1 \quad \Rightarrow \quad x_1 = b_1 / l_{11} \tag{2.3}$$

$$l_{21} x x_1 + l_{22} x x_2 = b_2 \quad \Rightarrow \quad x_2 = (b_2 - l_{21} x x_1) / l_{22} \tag{2.4}$$

$$l_{31} x x_1 + l_{32} x x_2 + l_{33} x x_3 = b_3 \quad \Rightarrow \quad x_3 = (b_3 - l_{31} x x_1 - l_{32} x x_2) / l_{33} \tag{2.5}$$

Así, se puede ver que no es complejo resolver un sistema de ecuaciones triangular. Lo mismo sucede si se utiliza una matriz triangular superior. Esto es una motivación suficiente para tratar de encontrar un sistema de ecuaciones triangular equivalente al sistema de ecuaciones original.

Específicamente, la factorización LU convierte un sistema de ecuaciones de la forma de la ecuación 2.1 en un sistema triangular equivalente tal que se verifica la ecuación 2.6.

$$A = LxU \tag{2.6}$$

Donde L es una matriz triangular inferior (con unos en la diagonal principal) y U es una matriz triangular superior. Estas matrices triangulares se utilizan para hallar las incógnitas representadas por x en la ecuación 2.1.
Para lograrlo, a partir de las ecuaciones (2.1) y (2.6):

$$LUx = b \tag{2.7}$$
$$Ux = z \tag{2.8}$$

donde $z = L^{-1}b$, entonces utilizando L (L es una matriz triangular inferior) se determina z:

$$Lz = b \tag{2.9}$$

y a continuación se resuelve:

$$Ux = z \tag{2.10}$$

y de esta forma se obtienen los valores de las incógnitas representadas por x.

En la ecuación 2.9, donde $z = L^{-1}b$, no se calcula la inversa de L, sino que se obtienen los valores de z resolviendo el sistema de ecuaciones triangular con múltiples resultados. Es común encontrar en la bibliografía esta notación y se utiliza en este trabajo para no perder claridad.

Los nombres L y U corresponden a Lower y Upper respectivamente. Toda matriz cuadrada no singular (matriz con inversa) tiene su descomposición LU y la misma es única.

2.2 Factorización LU secuencial

La solución secuencial de la factorización LU realiza cálculos sobre los elementos de la matriz original para obtener los elementos de L y U tales que verifican la ecuación 2.6. La matriz original es sobrescrita con los elementos de L y U.

Por ejemplo, si se tiene una matriz A de orden 3:

$$A^0 = \begin{pmatrix} a_{00}^{0} & a_{01}^{0} & a_{02}^{0} \\ a_{10}^{0} & a_{11}^{0} & a_{12}^{0} \\ a_{20}^{0} & a_{21}^{0} & a_{22}^{0} \end{pmatrix}$$

Para distinguir los distintos pasos en el avance de la factorización, A^0 es la matriz A en el paso cero de la factorización (A^0 es la matriz original), A^1 es la matriz A luego del paso 1 de la factorización, y así sucesivamente.

Para obtener los elementos de L y U tales que se verifique la ecuación 2.6, se aplican pasos de eliminación gaussiana a los elementos de la matriz A. La factorización avanza iterativamente para obtener los elementos de las matrices L y U.

Vale la pena recordar que los elementos de U que se encuentran debajo de la diagonal principal son 0 y los elementos de L que están sobre la diagonal principal también son 0. Así, los datos de las matrices L y U se almacenan en la misma matriz A (pues los datos no nulos de L y de U ocupan distintas posiciones dentro de la matriz, y como L tiene unos en la diagonal principal, los mismos se sobreescriben con los valores de la diagonal principal de U).

Para obtener los elementos de U, se realizan cálculos sobre los elementos de la matriz de forma tal que se anulen los elementos debajo de la diagonal principal. Entonces, realizando los siguientes cálculos sobre los datos de la columna 0:

$$a^{1}_{10} = a^{0}_{10} - a^{0}_{00} \times \frac{a^{0}_{10}}{a^{0}_{00}} \quad \text{y} \quad a^{1}_{20} = a^{0}_{20} - a^{0}_{00} \times \frac{a^{0}_{20}}{a^{0}_{00}}$$

se anulan dichos elementos.

Entonces, se realizan los mismos cálculos sobre todos los elementos de las filas, obteniendo $A^{1'}$:

$$A^{1'} = \begin{pmatrix} a^{0}_{00} & a^{0}_{01} & a^{0}_{02} \\ a^{0}_{10} - a^{0}_{00}\frac{a^{0}_{10}}{a^{0}_{00}} & a^{0}_{11} - a^{0}_{01}\frac{a^{0}_{10}}{a^{0}_{00}} & a^{0}_{12} - a^{0}_{02}\frac{a^{0}_{10}}{a^{0}_{00}} \\ a^{0}_{20} - a^{0}_{00}\frac{a^{0}_{20}}{a^{0}_{00}} & a^{0}_{21} - a^{0}_{01}\frac{a^{0}_{20}}{a^{0}_{00}} & a^{0}_{22} - a^{0}_{02}\frac{a^{0}_{20}}{a^{0}_{00}} \end{pmatrix}$$

De esta forma, sólo se tienen los elementos de U. A su vez, los elementos de la matriz L son los coeficientes utilizados en el cálculo de U. En la matriz ejemplo, estos coeficientes son:

$$a^{1}_{10} = a^{0}_{10}/a^{0}_{00} \quad \text{y} \quad a^{1}_{20} = a^{0}_{20}/a^{0}_{00}$$

Entonces, después de la primera iteración la matriz queda de la siguiente manera:

$$A^1 = \begin{pmatrix} a_{00}^1 = a_{00}^0 & a_{01}^0 = a_{01}^0 & a_{02}^0 = a_{02}^1 \\ a_{10}^1 = \dfrac{a_{10}^0}{a_{00}^0} & a_{11}^1 = a_{11}^0 - a_{01}^0 a_{10}^1 & a_{12}^1 = a_{12}^0 - a_{02}^0 a_{10}^1 \\ a_{20}^1 = \dfrac{a_{20}^0}{a_{00}^0} & a_{21}^1 = a_{21}^0 - a_{01}^0 a_{20}^1 & a_{22}^1 = a_{22}^0 - a_{02}^0 a_{20}^1 \end{pmatrix}$$

De esta forma, en cada iteración se avanza sobre los elementos de la matriz. Los elementos de la matriz que ya han sido factorizados, no participan del resto de la factorización: en la iteración k se trabaja desde la fila k y la columna k. En particular, en el paso i de la factorización se resuelve [18]:

$$a_{jk}^i = \begin{cases} a_{jk}^{i-1} & j \le i \text{ o } k \le i \\ a_{ji}^{i-1} / a_{ii}^{i-1} & j > i \text{ y } k = i \\ a_{jk}^{i-1} - a_{ik}^{i-1} a_{ji}^{i} & j > i \text{ y } k > i \end{cases}$$

Las filas y columnas menores a i no cambian.

La columna i cambia entre las filas i+1 y n por los multiplicadores correspondientes.

Transformación gaussiana.

En la figura 2.1 se muestra el pseudocódigo del algoritmo que soluciona esta factorización secuencial para una matriz A de n x n elementos (los índices de las filas tanto como los índices de las columnas varían desde 0 a n-1).

```
for ( i = 0 ; i < n ; i++)
  for ( j = (i+1) ; j < n; j++)
  {
    a(j,i) = a(j, i)  / a(i,i);            (1)
    for ( k = (i+1); k < n ; k++)
      a(j,k) = a(j,k) - a(i,k)*a(j,i);     (2)
    }
```

Figura 2.1: pseudocódigo de la factorización LU secuencial

Como se ve en la figura, la factorización avanza en cada iteración y en cada una de ellas se realizan combinaciones lineales entre las filas para obtener los elementos de L y U. En la iteración i se obtiene la matriz A^i y se trabaja con las filas i+1 hasta la fila n-1. De cada fila, las columnas que participan en el paso i son las columnas i hasta la n-1. Esta forma de resolver la factorización se la conoce como "right-looking" debido a la forma en que se resuelve la factorización.

Existe una medida que comúnmente se utiliza para conocer la complejidad del algoritmo. Esta medida es la cantidad de operaciones en punto flotante que realiza el algoritmo y se la

conoce como flop (contracción de Floating Point Operations). La cantidad de flops que realiza esta factorización se puede obtener del pseudocódigo anterior. La línea indicada con (1) realiza una operación y la línea indicada con (2) realiza 2 operaciones (una sustracción y una multiplicación). Como estas operaciones están dentro de iteraciones, estos factores se deben multiplicar por la cantidad de veces que las mismas se ejecutan (se asume que el orden de la matriz es *n*).

La línea (1) está dentro de una iteración que se ejecuta n-(i+1)+1 veces, por lo tanto, esta operación implica n-i flops en la iteración i-ésima.

La línea (2) está dentro de 2 iteraciones anidadas. Ambas iteraciones se ejecutan la misma cantidad de veces: desde i+1 hasta *n*. Por lo tanto, cada iteración se ejecuta *n*-(i+1)+1 veces. Así, la línea (2) que se encuentra dentro de las 2 iteraciones anidadas, se ejecuta un total de (*n*-(i+1)+1)x(*n*-(i+1)+1) veces. Entonces, la cantidad de flops requerida por la línea (2) en la iteración i-ésima es 2 x (*n*-(i+1)+1) x (*n*-(i+1)+1) que es igual a 2 x (*n*-i)^2.

Para realizar la factorización completa se requieren *n*-1 iteraciones, por lo tanto, la cantidad total de flops es:

$$totalflops = \sum_{i=1}^{n-1}\left(2(n-i)^2+(n-i)\right) = 2\left(\frac{(n-1)^3}{3}+\frac{(n-1)^2}{2}+\frac{(n-1)^3}{6}\right)+\frac{n^2-n}{2}$$

El factor con mayor complejidad es de orden cúbico y se toma este factor como el determinante de la complejidad total del algoritmo. Entonces, esta factorización aplica O(n^3) flops sobre O(n^2) datos. Esta factorización tiene los mismos requerimientos de cómputo y almacenamiento que las operaciones pertenecientes al Nivel 3 de BLAS [18].

2.2.1 Ejemplo de la obtención de las matrices *L* y *U* y su uso para resolver sistemas de ecuaciones lineales

Dado el sistema de ecuaciones:

$$\begin{aligned} x_1 + 4x_2 + 7x_3 &= 1 \\ 2x_1 + 5x_2 + 8x_3 &= 2 \\ 3x_1 + 6x_2 + 10x_3 &= 3 \end{aligned}$$

las matrices *A* y los vectores *x* y *b* son:

$$A = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 10 \end{pmatrix} \quad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad \text{y} \quad b = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

se obtienen las matrices L y U tales que se verifique la ecuación 2.6. A continuación se muestran los 3 pasos que implicaría encontrar estas matrices (los elementos de A^2 son los elementos de las matrices L y U).

$$A^0 = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 10 \end{pmatrix} \qquad A^1 = \begin{pmatrix} 1 & 4 & 7 \\ 2 & -3 & -6 \\ 3 & -6 & -11 \end{pmatrix} \qquad A^2 = \begin{pmatrix} 1 & 4 & 7 \\ 2 & -3 & -6 \\ 3 & 2 & 1 \end{pmatrix}$$

De esta forma, se cumple que ($A = L$ x U):

$$A = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 10 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{pmatrix} x \begin{pmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & 0 & 1 \end{pmatrix}$$

Una vez que se obtienen L y U, estas matrices se utilizan para obtener los valores de las incógnitas (x): utilizando las ecuaciones 2.1, 2.6 y 2.7 se llega a que $LUx = b$, entonces, reemplazando se obtiene:

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{pmatrix} x \begin{pmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & 0 & 1 \end{pmatrix} x \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

utilizando la ecuación 2.9 se calculan los valores de z ($Lz = b$):

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{pmatrix} x \begin{pmatrix} z_0 \\ z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

entonces,

$1z_0 = 1 \quad \Rightarrow \quad z_0 = 1$

$2z_0 + 1z_1 = 2 \quad \Rightarrow \quad z_1 = 2\text{-}2 = 0$

$3z_0 + 2z_1 + 1z_2 = 3 \quad \Rightarrow \quad z_2 = 3\text{-}3 = 0$

luego,

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Otra forma de calcular z es resolviendo $z = L^{-1}b$, primero se calcula la matriz inversa de L

$$L^{-1} = \frac{1}{|L|}(Adj(L))^t = \frac{1}{1}\begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & -2 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & -2 & 1 \end{pmatrix}$$

y se obtiene z ($z = L^{-1}b$),

$$z = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & -2 & 1 \end{pmatrix} \text{x} \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Se puede observar que las dos formas de calcular z son equivalentes, pero en la práctica se utiliza la primer forma, esto es, resolviendo un sistema de ecuaciones triangular. La segunda posibilidad se utiliza como notación para definir los valores de z.

Una vez calculados los valores de z, se resuelve la ecuación 2.10 ($Ux = z$)

$$\begin{pmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & 0 & 1 \end{pmatrix} x \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

con lo que se obtienen los valores $x_1 = 1$, $x_2 = 0$ y $x_3 = 0$.

2.3. Factorización LU secuencial por bloques

La mayoría de las operaciones de álgebra lineal que operan con matrices se encuentran definidas secuencialmente y también por bloques. La factorización LU no es una excepción a esta característica. La definición de estas operaciones por bloques mejora el rendimiento de las mismas, debido principalmente al uso de las memorias instaladas en las computadoras.

Hoy en día las computadoras tienen distintos tipos de memoria instaladas. Estos distintos tipos de memorias forman una jerarquía determinada por su velocidad de acceso. Esta velocidad, determina también el costo y el tamaño de las mismas.

Cuando el procesador está ejecutando un programa, éste accede a la memoria principal para acceder a datos y a instrucciones. El tiempo que tarda en acceder a la memoria afecta los tiempos totales de ejecución. Por lo tanto, se debe tratar de minimizar este tiempo de acceso. Con este objetivo es que se incluye memoria caché en las computadoras.

Comúnmente, la jerarquía de memoria instaladas en las computadoras actuales es la mostrada en la figura 2.2.

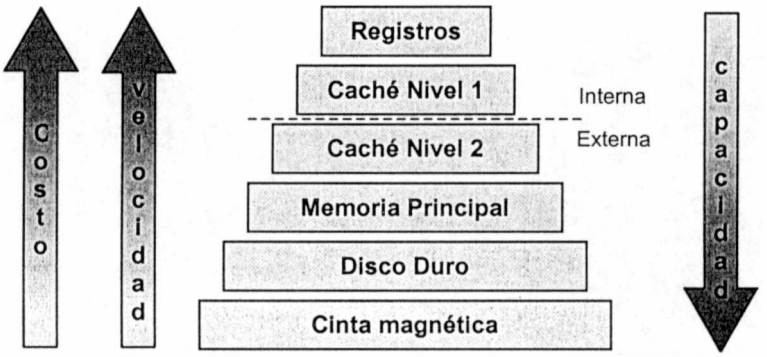

Figura 2.2: jerarquía de memorias instaladas en una computadora.

Además en la figura 2.2 se muestra cómo se comportan ciertas características importantes de las memorias: velocidad, capacidad y costo. A medida que se sube por la jerarquía, aumenta la velocidad de acceso a los datos. Esto es debido a la tecnología utilizada en los componentes de los mismos. Pero también la construcción de estas memorias es más cara: los componentes son más caros y la construcción misma es más compleja. Estas son algunas de las razones por las que el costo también aumente a medida que se sube por la jerarquía. Por otro lado, la capacidad de almacenamiento disminuye a medida que se sube. Los niveles inferiores son los que tienen mayor capacidad de almacenamiento.

Las memorias que se indican como "internas", se llaman así porque se encuentran dentro del chip donde se encuentra el procesador, en cambio las externas se encuentran fuera del mismo. Obviamente, la cercanía al procesador disminuye el tiempo de acceso. El tiempo de acceso a las memorias que se encuentran fuera de la CPU es 2 órdenes de magnitud más lento que el acceso a las memorias internas.

La memoria caché es una memoria que se agrega entre el procesador y la memoria principal y tiene menor tiempo de acceso que esta última. Entonces, si los datos que necesita el procesador se encuentran en la memoria caché, se obtienen de esta misma y se evita el acceso a la memoria principal. Comúnmente, estas memorias son chicas y esto se debe a que la complejidad de las mismas aumenta a medida que aumenta su capacidad de almacenamiento.

Para aumentar la probabilidad de que los datos buscados estén en la memoria caché, se utiliza el principio de localidad de referencia. El principio de localidad de referencia es de dos tipos: localidad temporal y localidad espacial. La primera se refiere a que es altamente probable referenciar una localidad de memoria que ha sido referenciada muy recientemente. Localidad espacial se refiere a la alta probabilidad de que la próxima referencia a memoria sea cercana a la última referencia realizada. Estos principios son utilizados para elegir que datos se asignan en memoria caché y de esta forma tratar de evitar el acceso a memoria principal [22].

Otra forma de mejorar el uso de las memorias es cambiar la forma de ejecutar las instrucciones de los algoritmos para hacer mejor uso de las memorias instaladas. O sea, analizar las secuencias de instrucciones de los algoritmos y realizar las modificaciones necesarias para aumentar la probabilidad de que el dato que se busca se encuentre en los registros o en memoria caché. En definitiva, dado un algoritmo y un conjunto de memorias se modifica la forma en que se ejecutan las instrucciones para lograr optimizar el uso de las memorias [5].

Es común encontrar a la mayoría de las operaciones de BLAS Level 3 definidas en términos de submatrices o bloques. Definir una operación entre matrices por bloques, permite aprovechar los distintos niveles de memoria instalados en las computadoras, y evitar el movimiento de datos entre los mismos. Existen bibliotecas de álgebra lineal que ofrecen versiones optimizadas de sus operaciones. La optimización de estas operaciones se basa fundamentalmente al procesamiento por bloques y a la mejora lograda en el uso de los registros y memoria caché [9].

En los algoritmos definidos por bloques, se disminuyen los accesos a memoria principal. Esto sucede porque se aumenta la cantidad de veces que un dato que se va a referenciar se encuentra en memoria caché, por lo tanto no hay que acceder a la memoria principal para obtenerlo. Esto se logra porque se maximizan la cantidad de operaciones que se realizan sobre un dato una vez que este se referencia (y es llevado a memoria caché). Dicho en otras palabras, cuando un dato es referenciado, se lleva a memoria caché y se intenta realizar la mayor cantidad posible de las operaciones en que participa ese dato [19].

Una métrica utilizada para determinar el uso de las memorias es determinar la relación entre las referencias a memoria y la cantidad de operaciones en punto flotante realizadas. Esta medida determina la forma en que se reusan los datos que se encuentran en los registros, pues los accesos a memoria se realizan cuando el dato que se necesita no se encuentra en los registros [5].

Las características de las memorias instaladas en las computadoras son importantes de considerar en el momento de resolver un problema ya que se los puede aprovechar para obtener algoritmos que alcancen mayor rendimiento. En el caso de álgebra lineal, esto es lo que determina la resolución por bloques de las operaciones entre matrices.

En los algoritmos definidos por bloques, las matrices originales se dividen en submatrices o bloques y los algoritmos se expresan en términos de operaciones entre matrices pero se resuelven entre los bloques. El tamaño del bloque definirá el rendimiento del algoritmo. Los algoritmos por bloques requieren una versión sin bloques del mismo algoritmo, para operar en cada uno de los bloques.

Particularmente en la factorización LU, la matriz a factorizar se divide de la siguiente forma:

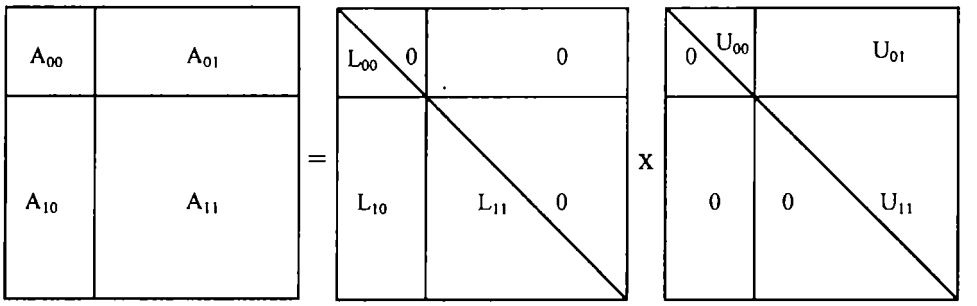

Figura 2.3: Matriz a factorizar dividida en bloques.

donde L_{00} y L_{11} son matrices triangulares inferiores con unos en la diagonal principal y U_{00} y U_{11} son matrices triangulares superiores.

Ahora, factorizar la matriz por bloques implica resolver las ecuaciones:

$$A_{00} = L_{00} x U_{00} \tag{2.11}$$
$$A_{01} = L_{00} x U_{01} \tag{2.12}$$
$$A_{10} = L_{10} x U_{00} \tag{2.13}$$
$$A_{11} = L_{10} x U_{01} + L_{11} x U_{11} \tag{2.14}$$

Estas ecuaciones surgen de la definición misma de la multiplicación de matrices (ya que se está resolviendo el producto L x U). Entonces, para resolver la factorización se deben encontrar los valores de las submatrices L_{00}, L_{10}, L_{11}, U_{00}, U_{01} y U_{11}.
Los valores de L_{00} y U_{00} se obtienen aplicando directamente la factorización LU a la submatriz A_{00}. De esta forma se resuelve la ecuación 2.11. Luego, usando la ecuación 2.12, como L_{00} es una matriz triangular inferior se utiliza el método de resolución de sistemas de ecuaciones triangulares resolviéndose la ecuación 2.15 y obteniéndose A_{01}.

$$U_{01} = L_{00}^{-1} x A_{01} \tag{2.15}$$

De la misma forma, a partir de la ecuación 2.13 reescrita como se muestra en la ecuación 2.16, como U_{00} es una matriz triangular superior, se obtienen los valores de L_{10}.

$$L_{10} = A_{10} x U_{00}^{-1} \tag{2.16}$$

En las ecuaciones 2.15 y 2.16, no se calculan las inversas de las matrices L_{00} ni U_{00}, es sólo una forma de notación. Para obtener los valores de U_{01} y L_{10}, se utilizan resoluciones de sistemas de ecuaciones triangulares.

Resta encontrar los valores de L_{11} y U_{11}. Para esto, se puede reescribir la ecuación 2.14 de la forma:

$$L_{11} x U_{11} = A_{11} - L_{10} x U_{01} \tag{2.17}$$

Entonces, se aplica el mismo método de factorización LU por bloques a la submatriz resultante de $A_{11} - L_{10}xU_{01}$, y de esta forma se obtiene la factorización de la matriz entera.

Realizar esta factorización por bloques implica que [18]:

1) Las operaciones quedan definidas principalmente por la multiplicación de matrices. Comúnmente, el tamaño de bloque es chico si se compara con el tamaño total de la matriz. Esto implica que la mayor cantidad de operaciones se realizan en la multiplicación de matrices de la ecuación 2.17, en la actualización de las filas y columnas con índice mayores al bloque factorizado (parte "activa" de la matriz).

2) Se optimiza el uso de la jerarquía de memorias de dos formas:
   a. La submatriz A_{00} puede ser procesada enteramente en caché al realizar la factorización LU secuencial en dicho bloque.
   b. La multiplicación de matrices puede ser modificada para optimizar el uso de caché. Por ejemplo, se puede resolver por bloques, cambiar el orden de las iteraciones, los niveles de "loop unroll", etc. [5].

3) Se mejora el rendimiento secuencial, pudiéndose mejorar el paralelo también. El rendimiento secuencial se mejora debido a la optimización del uso de memorias (mencionado en el punto 2). A su vez, el rendimiento paralelo se mejoraría porque resolver la operación por bloques, reduce la cantidad de movimiento de datos entre los procesadores (lo que implica la reducción de comunicaciones, factor que penaliza fuertemente el rendimiento paralelo).

En la solución por bloques de la factorización LU, la parte de la matriz que ya participó en la factorización no se vuelve a utilizar, quedando inactiva. La parte activa de la matriz (los bloques aún no factorizados) se utilizan en las actualizaciones (ecuación 2.14) y luego en la factorización propiamente dicha. De esta forma, se puede ver que la parte activa de la matriz se va reduciendo de iteración a iteración y la parte inactiva crece.

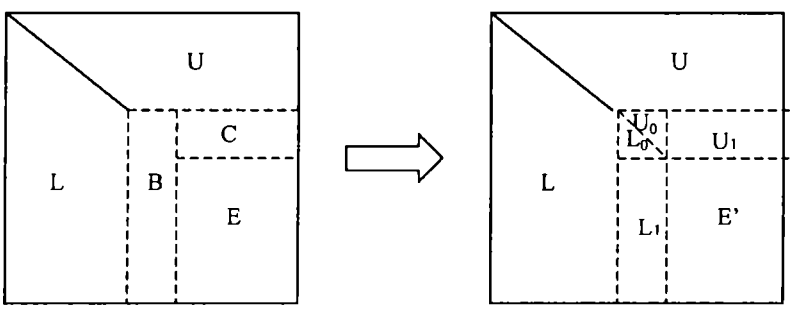

Figura 2.4: avance de la factorización LU

En la figura 2.4 se muestra el avance natural de la factorización. En el paso k+1 de la factorización, se actualizan los bloques B y C con L_1 y U_1 y la submatriz restante E también se actualiza. Las submatrices L y U ya han sido factorizadas en pasos previos. Si r es el tamaño de bloque, L tiene kr columnas y U tiene kr filas. En el paso mostrado, se factorizan otras r columnas de L (L_0 y L_1) y r filas de U (U_0 y U_1) y se actualiza E (E').

La figura 2.5 muestra el avance de la factorización para una matriz con 4 bloques, donde:

- Datos que no han sido factorizados y que participan en las actualizaciones. Específicamente, participan en la actualización de la matriz (E' en la figura 2.4).
- Datos que participan durante la factorización del bloque corriente y en las resoluciones de sistemas de ecuaciones lineales (L_0, U_0, L_1 y U_1 en la figura 2.4)
- Datos que ya han sido factorizados. Sólo participan del pivoteo de las columnas (más adelante en este capítulo se explicará este procesamiento).

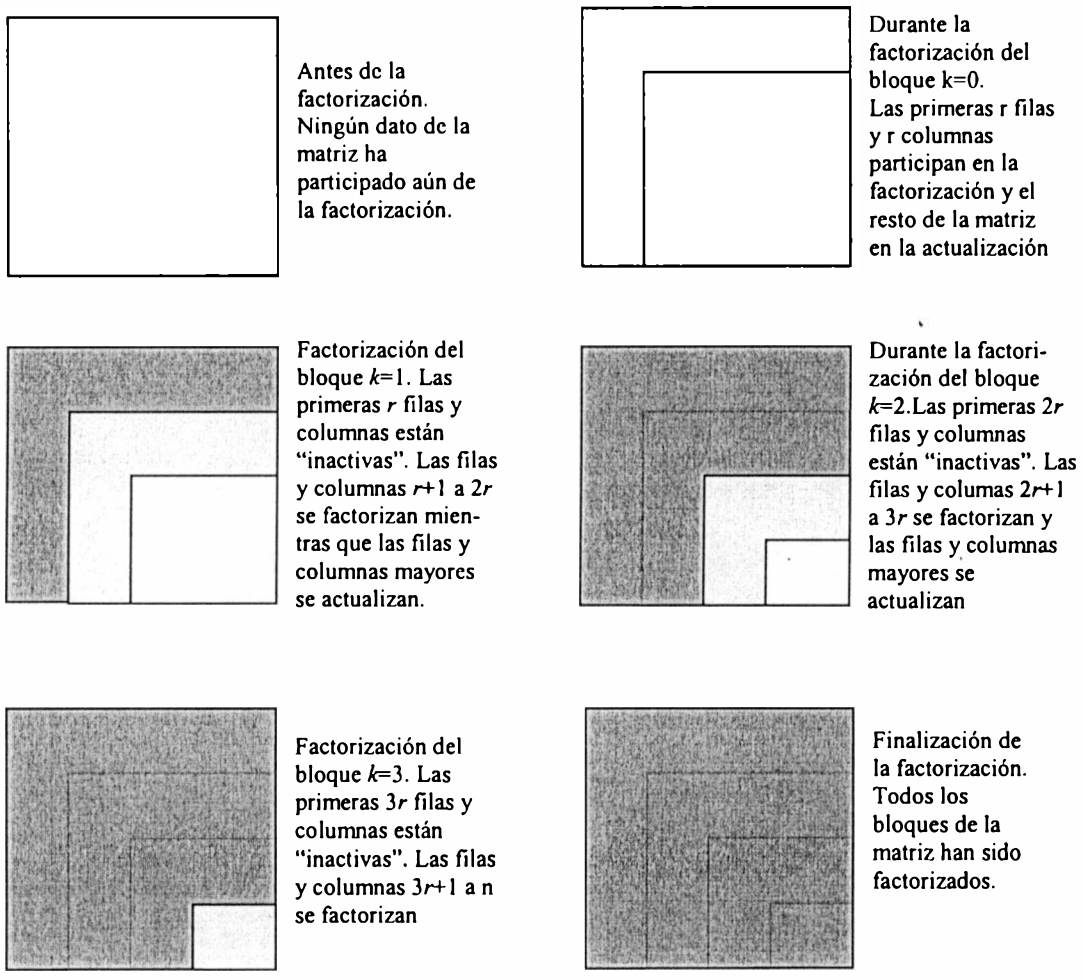

Figura 2.5: Avance de la factorización LU

2.4. Factorización LU paralelo por bloques

Como ya se ha mencionado, en una solución por bloques la factorización se resuelve entre los bloques en que se divide la matriz. Para esta factorización se utilizan bloques cuadrados de datos ya que esto simplifica el código y evita sobrecarga en el algoritmo paralelo (overhead). La forma en que se distribuyen estos bloques determinará el balance de carga, esto es, cuánto trabajo realizará cada nodo durante la factorización [17].

Existen distintas formas de distribuir los bloques entre los nodos:

- Secuencial: se asignan bloques contiguos a los procesadores. Se usa cuando el balance de carga se distribuye homogéneamente sobre la estructura de datos con que se trabaja, esto es, se realiza la misma cantidad de trabajo sobre cada elemento de la estructura de datos. La figura 2.6 muestra esta distribución, donde se distribuyen los primeros 10 bloques (0..9) entre 3 procesadores ($p_0..p_2$).

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

Figura 2.6: Asignación secuencial de bloques a los procesadores.

- Cíclico: se asignan bloques consecutivos a diferentes procesadores. Se usa para mejorar el balance de carga cuando la carga computacional se distribuye de forma no homogénea sobre la estructura de datos utilizada, o sea, la cantidad de trabajo que se realiza sobre un dato, depende de su ubicación en la estructura de datos. La figura 2.7 muestra esta distribución para los mismos elementos que en la figura 2.6.

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

Figura 2.7: Asignación cíclica de bloques a procesadores.

En el caso de la factorización LU, la cantidad de cómputo que se realiza sobre cada dato depende de su posición en la matriz. Esto se puede observar en la figura 2.5, donde se muestra el avance de la factorización. En esta figura, se puede observar que los últimos bloques participan durante toda la factorización mientras que los primeros participan sólo al comienzo de la misma. Por esto, en la factorización LU se utiliza la distribución cíclica de bloques.

Toda forma de distribuir los datos, debería ser tal que logre un buen balance de carga, esto es, que todos los nodos trabajen tiempos similares. En el caso de la factorización LU, donde la cantidad de trabajo sobre los datos se distribuye heterogéneamente, aunque los nodos tengan la misma capacidad de procesamiento y reciban la misma cantidad de bloques, puede suceder que los tiempos de cómputo sean distintos. Distribuyendo los nodos de forma cíclica, se logra que todos los nodos trabajen en la mayor cantidad de iteraciones posibles, evitando de esta forma que queden ociosos a medida que avanza la factorización.

Se mostrará con un ejemplo una forma general de distribuir los datos de forma cíclica entre nodos dispuestos en forma de grilla. Sea una matriz de M x N elementos, se divide la matriz en bloques cuadrados de r x r elementos y estos bloques se distribuyen entre los procesadores dispuestos en una grilla de P x Q procesadores. En la figura 2.8 se muestra la descomposición cíclica de una matriz dividida en M=9r y N=8r bloques los cuales se distribuyen en una grilla de P=3 y Q=4 procesadores. Cada bloque se etiqueta con el procesador donde es asignado. En la figura se sombrean grupos de bloques para mostrar cómo se mapea la configuración de los procesadores en la matriz.

p,q	0	1	2	3	4	5	6	7
0	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3
1	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3
2	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3
3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3
4	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3
5	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3
6	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3
7	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3
8	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3

Figura 2.8: Asignación de bloques a la grilla de procesadores.

Una vez distribuidos los bloques, resta explicar cómo sería una solución paralela por bloques de la factorización. Los bloques en que se divide la matriz para realizar la distribución entre los nodos, son los bloques que se utilizarán para realizar la factorización. Como ya se mencionó, suponemos la matriz de M x N elementos, por lo tanto, la matriz se divide en M_b x N_b bloques, donde $M_b = M / r$ y $N_b = M / r$.

El algoritmo es iterativo y en cada iteración se factoriza un bloque completo de la matriz. Para esto, en cada iteración (i = 0..Min(M_b, N_b)-1) se realizan principalmente 3 tareas:

1- Factorización del bloque i, obteniéndose las submatrices L_0, U_0 y U_1 de la figura 2.4 (realizando intercambio de columnas si es necesario por procesamiento de pivotes).
2- Obtención de las columnas del bloque i-ésimo de L resolviendo el sistema de ecuaciones $B = L_1 U_0$ (figura 2.4).
3- Actualización de la submatriz restante, reemplazando $E' = E - L_1 U_1$ (figura 2.4).

En una solución paralela, asumiendo la distribución mostrada en la figura 2.8, es necesario comunicar la información referida a los pivotes y el bloque factorizado a los demás nodos, ya que los elementos de la matriz se encuentran distribuidos entre todos los nodos que componen la grilla.

La figura 2.9, muestra el pseudocódigo de la solución paralela de la factorización. Este código se ejecuta en todos los nodos. Los índices p y q son la posición del procesador en la grilla (fila y columna). El pseudocódigo está dividido en 3 partes: la primer parte factoriza las filas del bloque i. La segunda obtiene los elementos de L de los bloques correspondientes a la columan i. La tercer parte actualiza la submatriz restante.

En el código, se simplifica la comunicación asumiendo que una comunicación broadcast se corresponde con el envío de los datos desde los procesos que tienen los datos de origen y recepción de los datos de los procesos que no los tienen.

```
col = q0
row = p0

for ( i = 0 ; i < min(Mb,Nb)-1 ; i++)

  for (k = 0; k < r; k++)
    if (p == row)
      encontrar el valor del pivote y su posición
    broadcast pivote y posición a todos los procesos
    intercambiar columnas según pivotes
    if (p == row)
      dividir fila por el pivote (parte activa)

  if (q == col)
    broadcast U0 a todos los procesos de la columna q
    resolver L1U0

  broadcast U1 a todos los procesos de mi columna
  broadcast L1 a todos los procesos de mi fila
  actualizar E = E - L1U1

col = (col + 1) mod Q
row = (row + 1) mod P
```

Figura 2.9: Pseudocódigo de la factorización LU en paralelo con los bloques distribuidos cíclicamente en una grilla de procesadores de P x Q.

Esta es una solución que asume una distribución cíclica de los bloques entre los procesadores ordenados en forma de grilla de P filas por Q columnas. Este caso es general para este tipo de grilla. En otro capítulo de este mismo trabajo, luego de definir las principales características de los clusters heterogéneos, se volverá sobre el algoritmo. El objetivo de las modificaciones sobre el algoritmo será obtener rendimiento óptimo en clusters heterogéneos.

3. Capítulo 3: Clusters Heterogéneos

Este capítulo describirá los conceptos más importantes de los clusters en general y luego se describirán los clusters heterogéneos formados por computadoras comunes o de escritorio. En el ámbito de la programación paralela es útil considerar la arquitectura desde el momento del diseño de una solución. Esto es debido a que el rendimiento de la arquitectura utilizada influye en el rendimiento total de la ejecución y lo que se busca en una solución paralela es lograr un mejor rendimiento. Esta idea cobra mayor importancia en el ámbito de clusters heterogéneos, donde se está ante una arquitectura que no ha sido creada específicamente para cómputo paralelo, por lo tanto las características de sus componentes pueden penalizar fuertemente el rendimiento total obtenido.

En este punto es útil mencionar que un cluster es una computadora paralela formada por nodos (PCs de escritorio, estaciones de trabajo o computadoras con procesamiento simétrico SMP), y estos nodos se encuentran conectadas por una red Ethernet. En síntesis, se utiliza una red local de computadoras como computadora paralela. A lo largo de las secciones siguientes se mencionarán otros componentes importantes de esta arquitectura.

3.1. Clasificación de los clusters

Existen distintas clasificaciones para este tipo de plataforma, cada una de las cuales se basa en criterios distintos [6]:

3.1.1 Tipo de Aplicación

a. Clusters de Alto Rendimiento (High Performance, HP).

Estos clusters intentan obtener el máximo rendimiento en la ejecución de aplicaciones. Esto es, obtener respuestas más rápidas u obtener mayor cantidad de respuestas en un lapso de tiempo (corto).

b. Clusters de Alta Disponibilidad (High Availability, HA).

El bajo costo de los clusters implica entre otras cosas:

- Existencia de gran número de redes instaladas que se las puede utilizar para cómputo paralelo.
- Existe la posibilidad de comprar y armar un cluster a bajo costo si se lo compara con una computadora paralela con prestaciones similares.

A su vez, el gran número de clusters instalados implica:

- Posibilidad de alcanzar mayor rendimiento de forma trivial, pues la gran cantidad de clusters instalados aumenta la posibilidad de disponerlos, utilizarlos y alcanzar mayor rendimiento en las aplicaciones.
- En caso de falla de un cluster (seguramente por falla de alguno de sus componentes) es fácil reemplazarlo por otro y obtener las mismas prestaciones (o al menos similares).

3.1.2 Propiedad del cluster

a. Clusters dedicados.

Las únicas aplicaciones que se ejecutan sobre este tipo de clusters son aplicaciones paralelas o manejadores de colas para alta productividad. No existen usuarios que ejecuten otro tipo de aplicaciones. Esto hace que todo el cluster esté disponible y se utilice sólo para cómputo paralelo.

b. Clusters no dedicados

En este tipo de cluster, existen usuarios propietarios de los recursos, los cuales ejecutan aplicaciones. Usualmente estas aplicaciones dejan ciclos de CPU ociosos, los cuales se pueden usar para cómputo paralelo. Para evitar conflictos entre los distintos usuarios de

estos recursos (propietarios y ejecuciones paralelas), es necesario utilizar estrategias para principalmente dos cosas: no penalizar el rendimiento de las interacciones en usuarios comunes y obtener buen rendimiento en aplicaciones paralelas. Estas estrategias abarcan, entre otras cosas, migración de aplicaciones, balance de carga, etc.

3.1.3 Tipo de nodo

a. Clusters de PCs (CoPs).
b. Clusters de Workstations (COWs).
c. Clusters de SMPs (CLUMPs).

3.1.4 Sistema Operativo de los nodos

a. Clusters que utilizan Linux.
b. Clusters con Solaris.
c. Clusters con NT.
d. Clusters con AIX.
e. Clusters Digital VMS.
f. Clusters HP-UX.
g. Clusters Microsoft Wolfpack.

3.1.5 Configuración de los Nodos

a. Clusters Homogéneos.

Todos los nodos son iguales y tienen el mismo sistema operativo. Esto implica que los nodos obviamente tengan todos la misma capacidad de procesamiento.

b. Clusters Heterogéneos

Los nodos podrían tener distintas arquitecturas y diferentes sistemas operativos. Las capacidades de cómputo pueden variar de nodo en nodo. La heterogeneidad puede ser tan grande como tipos de nodos existen.

3.1.6 Niveles de Clusters

a. Clusters Grupales (de 2 a 9 nodos).
b. Clusters Departamentales (de 10 a 100 nodos).
c. Clusters Organizacionales (algunos cientos de nodos).
d. Metacomputadoras Nacionales (algunos clusters departamentales/organizacionales interconectados, basados in WAN/Internet).
e. Metacomputadoras Internacionales (de miles a millones de nodos).

3.2 Componentes de los clusters

Un cluster es una arquitectura paralela distribuida débilmente acoplada. Está compuesta por múltiples nodos conectados que trabajan como un único recurso integrado. A su vez, cada nodo puede ser un sistema monoprocesador (PC común de escritorio, workstation) o multiprocesador, como es el caso de los SMPs (Simmetric Multiprocessor) [6].

Los nodos pueden estar separados físicamente y conectados a través de una LAN (Local Area Network). Este tipo de sistema puede estar configurado de tal forma que es visto como un sistema único por el usuario y por las aplicaciones.

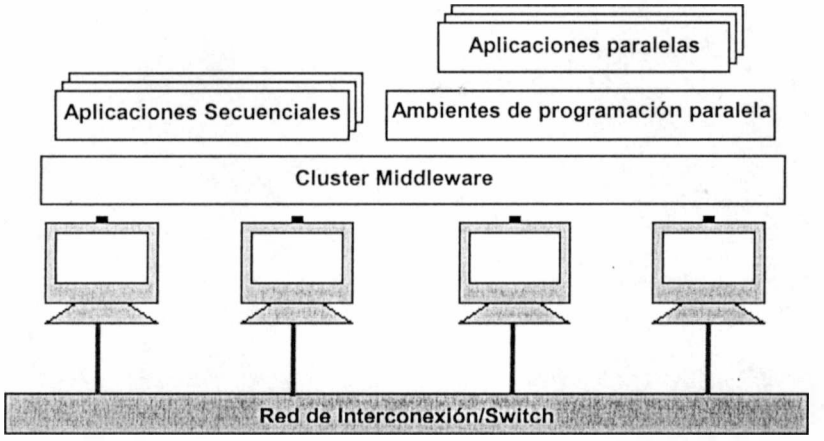

Figura 3.1: arquitectura de los clusters de computadoras.

La figura 3.1 muestra los distintos componentes de la arquitectura. En la figura se muestran los componentes más importantes: nodos, red de interconexión, middleware, herramientas para programación paralela y aplicaciones. En las subsecciones siguientes, se explicará cada uno de estos componentes.

3.2.1. Nodos

Un cluster tiene múltiples nodos cada uno de los cuales tiene su propio sistema operativo. Además, cada uno tiene hardware y software para la comunicación. Como ya se ha mencionado, los nodos utilizados en los clusters son PCs de escritorio común, estaciones de trabajo o SMPs.

Cada nodo es una computadora completa, formada por distintos componentes. Los más relevantes son: procesador, memoria y discos de almacenamiento o E/S.

3.2.1.1 Procesadores

La tecnología de los procesadores utilizados comúnmente en los clusters (sean PCs, estaciones de trabajo o SMPs) ha avanzado velozmente en las últimas 2 décadas, y como resultado de esto, se tiene que estos procesadores tienen potencia similar a los procesadores utilizados en las supercomputadoras.

En los últimos tiempos, el tiempo desde que sale al mercado un procesador y el momento en que se deja de fabricar (porque se ha fabricado alguno mejor) es cada vez más corto. Como ya se ha mencionado en este trabajo, esto afecta directamente (y en gran forma) en la capacidad de un cluster homogéneo para mantenerse como tal a lo largo del tiempo.

3.2.1.2 Memoria y memoria caché

En este tipo de plataforma, las aplicaciones están distribuidas a lo largo del cluster. De la misma forma, los datos que procesan estas aplicaciones también suelen estar distribuidos entre todas las memorias de los nodos. Por esta razón, no es necesario que toda la aplicación entera esté en memoria RAM de una computadora. Pero sí es necesario que haya la suficiente memoria para evitar el acceso a memoria swap ya que el acceso a disco es mucho más lento, lo que penalizaría el rendimiento total de la aplicación.

Para acelerar el acceso a memoria se utiliza la memoria caché, la cual contiene bloques que se supone se pueden volver a referenciar próximamente. Si el bloque se encuentra en caché se usa directamente desde esta memoria, evitando así acceder a memoria principal (esto depende de la localidad referencial, ya explicado anteriormente en este trabajo). Pero la memoria utilizada para memoria caché es mucho más cara y el circuito de control de este tipo de memoria se hace muy complejo a medida que esta crece. Estas características hacen que las memorias caché utilizadas actualmente sean de poca capacidad.

3.2.1.3. Discos y Entrada/Salida

Una parte importante del rendimiento alcanzado por una aplicación depende del rendimiento de todos los componentes de la plataforma donde se corra. Y este rendimiento total depende del rendimiento del componente más lento, el cual puede llegar a actuar como "cuello de botella" penalizando el rendimiento total. Sobre esto se refiere la ley de Amdahl, que menciona que el speed-up obtenido por el procesador más rápido está limitado por el componente más lento de todo el sistema.

Estas son razones válidas para intentar mejorar el rendimiento de las operaciones de entrada/salida para de alguna forma, acercarlas al rendimiento alcanzado por las CPUs (las cuales tienen un rendimiento mucho mayor). Una forma de mejorar las entradas/salidas es realizando estas operaciones en paralelo. Y esto es posible si se utilizan todos los discos de los nodos del cluster como un sistema RAID (Redundant Array of Inexpensive Disk). RAID es una forma de almacenar los mismos datos en diferentes lugares (teniendo así

redundancia de los mismos), en múltiples discos duros. De esta forma, teniendo los datos en múltiples discos, las operaciones de entrada/salida se pueden solapar, realizando más de una operación simultáneamente. De esta forma, se mejora el rendimiento de estas operaciones. Múltiples discos aumentan el tiempo medio entre fallas y almacenar datos redundantemente aumenta la tolerancia a fallas.

3.2.2. Red de Interconexión

La red de interconexión de los clusters es la red Ethernet la cual utiliza, inicialmente, el protocolo IEEE 802.3, en donde existe un único medio de transmisión compartido por todos los nodos. Todos los nodos compiten por el medio compartido. Cuando el emisor envía un mensaje, el mismo ocupa todo el medio, por lo que existe un sólo mensaje a la vez en este medio. El mensaje transmitido llega a todos los nodos que están conectados a la red. Cuando coexiste más de un mensaje en el medio, se detecta la colisión y se retransmiten los mensajes que colisionaron. Por estas características, se puede notar que se tiene broadcast a nivel físico de la red. El modo de acceso al bus compartido es CSMA/CD (Carrier Sense, Multiple Access / Collision Detect). No existe el concepto de prioridad entre los mensajes.

Comúnmente, se utilizan redes Standard Ethernet de 10 Mb/s (transmiten 10^6 bits por segundo). También se ha definido la Ethernet Rápida (Fast Ethernet) de 100 Mb/s. Ambas redes se utilizan en este tipo de máquina paralela. En la actualidad se ha definido la Gigabit Ethernet, la cual llega a transmitir 10^9 bits por segundo. Además se utilizan placas de interfaz de comunicación (NICs: Network Interface Card), las cuales actualmente están definidas para transmitir 10/100 Mb/s.

La red de comunicación está formada por el cableado mismo de la red y por las placas de red que tiene cada uno de los nodos. El cableado inicialmente se basó en el uso de cables coaxil. Luego, se reemplazaron por cables de par trenzado y hubs. Estos componentes cambian la interconexión física de la red a una topología estrella. Pero como los hubs distribuyen la señal a todos los nodos conectados, la topología lógica sigue siendo de tipo bus. Luego se ha reemplazado el uso de los hubs por switches. Los switches actúan como hubs en las comunicaciones de tipo broadcast, pero además, cuando hay múltiples comunicaciones punto a punto las resuelve de forma simultánea. Con las comunicaciones colectivas, los switches actúan como hubs, con lo que se sigue manteniendo la topología lógica de tipo bus [19].

Estas redes son buenas para conectar PCs que necesitan compartir algún tipo de recurso (datos, periféricos, etc) pero no son buenas para cómputo paralelo dado su bajo rendimiento. El bajo rendimiento es causado por su pobre tiempo de transferencia de datos.

Las principales características de un medio de transmisión es su ancho de banda (cantidad de datos transmitidos por unidad de tiempo) y el tiempo mínimo de transmisión (tiempo mínimo de inicialización de las comunicaciones, latencia o startup). El tiempo total de transmisión está dado por:

$$t(n) = \nabla + ®n \qquad (3.1)$$

donde n es la unidad de información que se transfiere, ∇ es el tiempo de latencia, y ® es el valor inverso del ancho de banda. Entonces, según la ecuación 3.1, el tiempo total de comunicación es el tiempo de inicialización más el tiempo que se tarda en transmitir los datos (este último tiempo es la cantidad de datos a transmitir dividido la cantidad de datos que se pueden transmitir a la vez (que es el ancho de banda)).

Estas redes son varios órdenes de magnitud más lentas que las redes de interconexión utilizadas en computadoras paralelas tradicionales. En estas computadoras paralelas tradicionales, se crean todos los componentes sabiendo que se van a utilizar para cómputo paralelo, por lo tanto tendrán características favorables para su propósito final como también se tratará de evitar características que no sean favorables para dicho propósito. En el caso de las redes de computadoras esto es totalmente distinto, las redes se crean para otro propósito el cual dista mucho de tener características en común o parecidas al cómputo paralelo.

Estas características hacen que el rendimiento total de una aplicación paralela que se ejecuta sobre esta arquitectura esté directamente relacionada con el rendimiento de la red. Por estas razones, para que el rendimiento de los algoritmos paralelos no se vean penalizados por el bajo rendimiento de la red (o por lo menos, la penalización sea la mínima posible), es necesario tener en cuenta las características de la red para:

- Realizar la menor cantidad de comunicaciones posibles (si recibir un dato requiere más tiempo que calcularlo, es mejor calcularlo y obtenerlo localmente).
- Los mensajes podrían ser de tipo broadcast para aprovechar el broadcast a nivel físico que se tiene en la red.
- Determinar el número ideal de procesadores en un cluster. Esto es debido a que la red de interconexión puede llegar a actuar como cuello de botella.

3.2.3 Middleware

Un cluster está formado por múltiples nodos interconectados. Pero se puede configurar el sistema y ver a todos los nodos interconectados al cluster como un recurso único. Esto es llamado Single System Image (SSI). El middleware es el que se encarga de soportar esta visión del sistema. El middleware es una capa que se encuentra usualmente entre el sistema operativo y el nivel de ambientes de usuario. Puede estar en cualquiera de 3 niveles: nivel de hardware, de Sistema Operativo o de aplicaciones.

Esta capa permite a todos los nodos acceder a los recursos del sistema de forma uniforme. Además, permite ver toda la colección de recursos como uno solo, o con otras palabras, administrarlos de manera unificada.

Un middleware debería ofrecer buen rendimiento tanto de aplicaciones paralelas como así también de aplicaciones secuenciales. Para esto, el middleware debería (entre otras cosas),

identificar recursos ociosos en el sistema (procesadores, memoria, etc.) y ofrecer acceso globalizado a los mismos, migración de procesos (para proveer balance de carga dinámico), comunicación rápida entre los procesos, etc.

3.2.4 Herramientas de programación paralela

Un factor que ayudó al creciente uso de clusters para cómputo paralelo es la disponibilidad de herramientas y utilitarios para programar aplicaciones paralelas en esta arquitectura. Existen diversas herramientas:

- Librerías de pasaje de mensajes tales como PVM o MPI.
- Herramientas de debugger de aplicaciones.
- Herramientas para analizar rendimiento de aplicaciones.
- Herramientas de administración de clusters.

Estas herramientas tratan de ser lo más portables, eficientes y fáciles de usar posibles. Sin embargo, aún no hay consenso sobre las mismas, la única excepción es MPI, que define una interfaz que todas las implementaciones deben respetar.

3.2.5 Aplicaciones

Los clusters pueden ser usados para correr aplicaciones paralelas o secuenciales. Por lo tanto, en los clusters coexisten estos dos tipos de aplicaciones.

Los clusters son óptimos para ejecutar aplicaciones paralelas que serían intratables en un ambiente secuencial. Anteriormente en esta tesis se han puesto algunos ejemplos de este tipo de aplicaciones. La ciencia y la ingeniería son dominios que tienen gran cantidad de aplicaciones de este estilo.

3.3 Clusters Heterogéneos

En la actualidad se puede observar una gran tendencia de utilizar computadoras como herramientas de trabajo. Es normal que en cualquier entidad se instalen computadoras y que las mismas estén conectadas para compartir recursos (información, periféricos, bases de datos, etc). El número de redes locales instaladas y en funcionamiento es cada vez mayor.

En la sección anterior se enunciaron las principales características de los elementos que componen a los clusters. Otra característica importante que comparten estos componentes es su bajo costo, pues los mismos son, actualmente, de los más accesibles en el mercado.

En la década del 80, se creía que la mejor forma de mejorar el rendimiento de las computadoras era aumentando la velocidad del procesador. El cómputo paralelo hizo cambiar esta creencia uniendo dos o más procesadores para solucionar conjuntamente un problema. Ya en la década de los 90 se observó una fuerte tendencia de pasar de utilizar supercomputadoras (las cuales eran especializadas para algún tipo de problemas y de precio muy elevado) a utilizar clusters de computadoras (de propósito general y de precio mucho

más accesible). Este giro se produjo por la mejora en el rendimiento de los distintos componentes de las PCs y workstations y de las redes de comunicación. Los avances en estas tecnologías hacen posible tener una supercomputadora de bajo costo y alta disponibilidad. En este cambio también tuvo su aporte la estandarización de muchas herramientas y utilitarios usados por aplicaciones paralelas (algunos ejemplos son la interfaz de comunicaciones MPI y el lenguaje paralelo HPF) [6].

De esta forma, se tiene una arquitectura paralela de muy bajo costo y amplia disponibilidad dado el número cada vez mayor de redes locales instaladas. Además se puede llegar a obtener buen rendimiento en las aplicaciones si se utilizan de forma correcta los recursos de estas nuevas arquitecturas paralelas.

Una característica importante de las redes de computadoras es que son computadoras paralelas de acoplamiento débil: los nodos están formados por componentes de hardware que no tienen ningún tipo de relación entre sí. Los nodos son un conjunto de computadoras completas las cuales tienen una interfaz de red (NIC) para la comunicación. Esto implica principalmente dos cosas [19]: memoria distribuida y procesamiento asincrónico. La memoria distribuida se debe a que cada nodo tiene su memoria local y no hay ningún medio que facilite compartir estas memorias. También, el procesamiento es asincrónico y no hay ningún componente que facilite la sincronización. La única forma de compartir un dato o de sincronizarse es a través de la red de interconexión. Esto es fácil, pero las redes Ethernet no han sido diseñadas para esto y esto produce una alta penalización en el rendimiento paralelo (principalmente por su gran tiempo de latencia y su bajo ancho de banda).

Comúnmente, las aplicaciones paralelas necesitan buen rendimiento en las operaciones de punto flotante, redes de interconexión con baja latencia y ancho de banda alto y escalable, y acceso rápido a los archivos que utilicen. Estos requerimientos se pueden alcanzar si se utilizan de forma correcta los recursos asociados a los clusters.

Los clusters heterogéneos tienen ciertas características las cuales, en su conjunto, justifican todo el trabajo y esfuerzo existente en querer utilizar esta plataforma para cómputo paralelo. Algunas de estas características se tratan en las subsecciones siguientes.

3.3.1 Costo

La ventaja más importante de la utilización de clusters para cómputo paralelo es el bajo costo de los mismos. Cuando la red está instalada, se tiene la posibilidad de realizar cómputo paralelo en una plataforma de costo nulo o cercano a cero.

A partir de la clasificación vista en [22], se puede hacer un análisis de las arquitecturas paralelas actuales para comparar el valor que tiene el Gflop (1000 millones de operaciones en punto flotante) en cada una de ellas. Para dicha comparación, se tomará el valor de un Gflop en las computadoras que actualmente se encuentran rankeadas como las de mayores prestaciones en el mundo [25]. Luego, se compara con el costo de un Gflop en un cluster. En esta comparación se utilizan estas supercomputadoras porque son las computadoras que

alcanzan mayor cantidad de Gflop/s (Gflop por segundo), por lo que resulta interesante utilizarlas para dicha comparación.

En la tabla 3.1 se enumeran las primeras 5 supercomputadoras de el ranking TOP500. Este ranking lista las 500 computadoras con mayores prestaciones actuales. La misma se actualiza cada 6 meses y la última actualización fue hecha en noviembre del 2004. En la tabla se muestran los datos más importantes de cada computadora. En la cuarta columna se especifica el rendimiento máximo y el rendimiento pico obtenidos utilizando como benchmark la resolución de sistemas de ecuaciones lineales de LINPACK. En este caso, el rendimiento máximo es el rendimiento máximo alcanzado utilizando este benchmark y el pico es un rendimiento teórico, que es el rendimiento de un nodo multiplicado la cantidad de nodos que componen la computadora. La última columna especifica el precio de un Gflop. Para obtenerlo se tiene en cuenta el rendimiento máximo de cada computadora.

Puesto	Lugar País/Año	Computadora / Número de procesadores Fabricante	R_{peak} R_{max} (Gflops)	Costo[1]	Costo del Ggflop[2]
1	Rochester USA / 2004	BlueGene/L / 32768 DOE/IBM	91750 70720	100 (2)(3)	1414 u$s
2	Mountain View USA / 2004	Columbia / 10160 SGI	60960 51870	50 (3)	963 u$s
3	Earth Simulator Center Japan / 2002	Earth-Simulator / 5120 NEC	40960 35860	400 (3)	11154 u$s
4	Barcelona Supercomputer Center Barcelona, España / 2004	MareNostrum / 3564 IBM	31363 20530	90 (4)	4383 u$s
5	Lawrence Livermore National Laboratory United States / 2004	Thunder Intel Itanium2 Tiger4 1.4GHz - Quadrics / 4096 California Digital Corporation	22938 19940	20 (1)	1003 u$s

Tabla 3.1: costo del Gflop en las 5 supercomputadoras más rápidas del mundo.

[1] Costo en millones de dólares.
[2] El costo del Gflop es costo / R_{max}.

La información referente al costo de las supercomputadoras se ha encontrado en los siguientes sitios web:

(1) http://www.itjungle.com/breaking/bn111403-story01.html
(2) http://www.linuxelectrons.com/article.php/20041002190224223
(3) http://www.technewsworld.com/story/37975.html
(4) http://www.technewsworld.com/story/37948.html

A partir de la tabla se puede notar que el costo del Gflop es elevado en estas primeras 5 supercomputadoras. Aunque no se aclare en todas las fuentes donde se han obtenido estos datos, es muy probable que en los costos encontrados esté incluido el costo de todo el proyecto. Es probable que en los mismos esté incluido el costo del edificio (que en la mayoría de los casos se construye exclusivamente para la supercomputadora), el costo de

los recursos humanos que participaron en el proyecto, costo de equipos de refrigeración, etc.

Utilizando un cluster heterogéneo, el costo por Gflop en un cluster formado a partir de una red local ya instalada, es cero o cercano a nulo. Pues ya se tiene todo el hardware propio de la red. Como ya se ha mencionado, cuando la red ya está instalada y se la quiere configurar y utilizar para cómputo paralelo, el único costo es el del software necesario para la administración de la máquina paralela y las herramientas de desarrollo, ejecución, depuración, etc. de programas paralelos. Actualmente existen pocas herramientas para esto. Esto se debe a que esta arquitectura no fue creada para cómputo paralelo, y es algo relativamente actual este nuevo uso que se hace de dicha arquitectura. Sin embargo existen algunas librerías tales como PVM o MPI que se las puede encontrar de forma gratuita en Internet.

Por otro lado, si se desea armar un cluster pero sin utilizar una red ya instalada (comprando cada uno de sus componentes) el costo del Ggflop es muchísimo menor que el de las supercomputadoras. Según los costos actuales de las computadoras comunes de escritorio, un Gflop vale aproximadamente 400 u$s. Este valor es considerando que el valor de una PC de 2.5 Gflop es aproximadamente 1000 u$s. Para realizar el cálculo del costo del Gflop en un cluster, se suman todos los Gflops y todos los costos de cada una de las computadoras que forman el cluster. Y esta relación se mantendría, ya que la velocidad del procesador es uno de los factores que determina su costo. Entonces, este valor se puede utilizar como costo aproximado.

Se puede ver que el costo de un Gflop en los clusters sigue siendo muchísimo menor a partir de una red local ya instalada como así también utilizando un cluster para el cual se compran todos los componentes. De esta forma, un cluster es una arquitectura que permite tener cómputo paralelo, alcanzando buen rendimiento a costo muy bajo o nulo.

En el contexto de este trabajo, el costo del cluster es cero, ya que se utilizan clusters de computadoras ya instaladas y en uso.

3.3.2 Disponibilidad

La disponibilidad en los clusters se puede ver desde dos perspectivas:

- Disponibilidad de clusters para cómputo paralelo.
- Disponibilidad de un cluster para realizar cómputo paralelo.

3.3.2.1 Disponibilidad de clusters como computadora paralela

En la actualidad es muy común encontrar redes de computadoras instaladas y en uso en cualquier entidad (oficinas, negocios, empresas, colegios, facultades, etc). Estas redes

surgieron (en la mayoría de los casos) con el fin de compartir algún tipo de recurso entre todas las computadoras.

Además, se puede ver que no sólo estas redes existen en ámbitos laborales o académicos. Se puede ver el número creciente de "Cyber Cafe" que ofrecen a los usuarios juegos en red y conexión a Internet. Las redes locales de estos negocios son comúnmente muy buenas, con gran ancho de banda.

En definitiva, se puede observar la gran cantidad de redes instaladas, y que esta cantidad tiende a aumentar. Entonces, se puede considerar que estas redes pueden ser configuradas y utilizadas para cómputo paralelo. De esta forma, existe la posibilidad de tener cómputo paralelo en una arquitectura ampliamente disponible. Esto es una gran ventaja, y más cuando se la compara con la baja disponibilidad de otro tipo de computadoras paralelas, las cuales tiene un precio muy elevado si se las compara con el precio de una red local.

Por otro lado, si se utilizan bien las características de las redes en las aplicaciones paralelas, se puede lograr buen rendimiento, con lo que la alta disponibilidad de los clusters implica directamente la alta disponibilidad de lograr buen rendimiento.

3.3.2.2 Disponibilidad del cluster para realizar cómputo paralelo

Una característica de la mayoría de los clusters formados por redes locales de computadoras, es que estas redes seguramente tengan usuarios comunes, que no necesariamente utilizan el cluster para cómputo paralelo (clusters no dedicados). Una característica de las aplicaciones de usuario es que tienen muchos ciclos de CPU ociosos. Como se menciona en el capítulo 1 de este trabajo, existen trabajos y herramientas para aprovechar estos tiempos ociosos, asignándoles trabajo. De esta forma, se pueden utilizar los ciclos ociosos de las CPUs para cómputo paralelo. Estas herramientas tratan, en lo posible, de no perjudicar al usuario común con el nuevo uso que se le da al cluster. Las herramientas tratan de realizar cómputo paralelo y obtener buen rendimiento sin afectar la velocidad de interacción de las aplicaciones de usuario [16].

3.3.3 Evolución de los clusters

Es común que las redes de computadoras sean desde su comienzo heterogéneas. Esto se debe a la naturaleza de una red local: conjunto de computadoras conectadas para compartir algún tipo de recurso. Por lo tanto, es muy difícil encontrar que el hardware de estas computadoras tenga algún tipo de relación.

Pero también existen clusters homogéneos. Estos clusters tienen la propiedad de que es difícil mantenerlo como homogéneo a través del tiempo. Existen distintos factores que hacen que un cluster homogéneo en uso se convierta rápidamente en heterogéneo para poder seguir siendo útil. La principal razón es la evolución misma de los componentes que componen los clusters. Simplemente, observando la evolución de los procesadores de las computadoras de escritorio, se puede notar la gran velocidad de cambio, y cómo estos

cambios influyen en la disponibilidad de un procesador determinado (se hace difícil encontrar el mismo modelo de procesador, misma velocidad, etc).

Existen principalmente dos problemas que atentan contra la homogeneidad de un cluster: la necesidad de reparar o reemplazar algún componente y la necesidad de aumentar la capacidad del cluster.

Cuando existe la necesidad de reparar o reponer algún componente del cluster homogéneo, se puede optar por 2 posibilidades. Si se desea mantener la homogeneidad del cluster y no se encuentra el hardware apropiado para lograrlo, se puede optar por no reponer el componente. Esta solución es apropiada siempre y cuando el cluster siga resolviendo los problemas de forma aceptable. Pero, a lo largo del tiempo, esta opción atentará contra la capacidad del cluster, la cual es muy probable que decrezca. De esta forma, se está perdiendo capacidad de procesamiento por mantener el cluster homogéneo.

Otra posibilidad es realizar la reparación con el hardware disponible, que en el caso de no conseguir el mismo componente, el cluster automáticamente pasará a ser heterogéneo.

Una cosa muy parecida pasa en el caso de querer aumentar la capacidad del clúster. Si no se dispone del mismo hardware y se instala alguna componente nueva, se pasa directamente a tener un cluster heterogéneo. Como en el caso anterior, si se desea mantener la homogeneidad del cluster, aumentar su capacidad sin disponer del hardware apropiado implica directamente tener que comprar todo el cluster nuevo (con mayor capacidad). Esta opción es bastante más difícil, por lo que es común que se sacrifique la homogeneidad y se opte por agregar una componente nueva. De esta forma, el cluster pasa a ser heterogéneo [19]. Y esto sucede reiteradas veces durante la "vida útil" del cluster, con lo que la heterogeneidad crece a medida que pasa el tiempo.

Por estas razones, se puede ver que la evolución de los clusters es hacia los clusters heterogéneos.

3.3.4 Mantenimiento

Partiendo desde la base de utilizar una red en uso para como computadora paralela, es probable que exista alguna persona que se encarga de la administración y mantenimiento de la red. Entonces es este administrador de la red el que se encarga de mantener la red en funcionamiento acorde a los requerimientos de los distintos usuarios (requerimientos de hardware y software).

Desde este punto de vista del mantenimiento, el mantenimiento de la máquina paralela es intrínseco al de la red misma. No es necesario más que mantener en correcto funcionamiento las herramientas instaladas específicamente para cómputo paralelo.

Por otro lado, el mantenimiento del hardware de la red también es muy sencillo cuando el cluster es heterogéneo. Si algún componente que conforma el cluster falla, se reemplaza por

otro que tenga las mismas prestaciones (o similares), al ser heterogéneo, no existen restricciones sobre los nuevos componentes.

Luego, el mantenimiento de un cluster heterogéneo formado por una red local es muy sencillo, tanto a nivel de software como de hardware.

3.3.5 Escalabilidad

Aumentar la potencia de cómputo de un cluster heterogéneo es muy sencillo. La potencia total de un cluster es la suma de las potencias de todos los nodos. Entonces, si se desea aumentar la potencia de un cluster, se puede agregar cualquier computadora que, junto con el resto de los nodos, cubran las nuevas expectativas de potencia. La nueva computadora agregada puede ser cualquier computadora de escritorio (dado que estamos considerando un cluster heterogéneo). En la actualidad existe un amplio rango de computadoras de escritorio las cuales varían en capacidad de procesamiento. Por lo tanto, es fácil elegir la que más concuerde con los nuevos requerimientos que se desean cubrir con el cluster.

Por otro lado, si se considera la misma situación en un cluster homogéneo, se debe encontrar el mismo tipo de nodo para mantener la homogeneidad del cluster. Como ya se ha mencionado, esto es difícil, dada la velocidad con que evolucionan los principales componentes de las computadoras (procesadores, memorias, etc.).

3.3.6 Aplicaciones en clusters heterogéneos

Una aplicación paralela siempre es más difícil de entender, diseñar, desarrollar, implementar y depurar. El cómputo paralelo aumenta la complejidad de las soluciones desarrolladas. Sin embargo se utiliza el cómputo paralelo ya que existe la posibilidad de aumentar el rendimiento en una aplicación.

Pero para que esto realmente suceda, se deben utilizar de forma correcta las características de las arquitecturas paralelas. Para esto es necesario tener en cuenta las características de la plataforma en la va a correr la aplicación. Estas características es necesario tenerlas en cuenta desde el principio del desarrollo de la aplicación.

Entonces, en este contexto, se puede decir que dentro de la etapa de análisis del problema, se debe analizar también la arquitectura a utilizar. De esta forma, se conocen todas las características que pueden llegar a influir en el rendimiento y existe la posibilidad de utilizar de la mejor forma las que pueden llegar a mejorarlo como así también de evitar el uso de las características que pueden llegar a penalizar el rendimiento de la aplicación. De esta forma, las características propias de la arquitectura se tienen en cuenta en todas las etapas siguientes.

Como ya se ha mencionado, desarrollar una aplicación teniendo en cuenta las características de la plataforma para la cual es desarrollada, penaliza de alguna forma la generalidad de las soluciones. Se desarrollan soluciones más específicas para un tipo de arquitectura en particular. Estas soluciones lograrán buen rendimiento en todo el rango de

arquitecturas que compartan algunas (o todas) de estas características. Sin embargo, es muy probable que no logren buen rendimiento en arquitecturas donde dichas características no estén presentes.

Particularmente hablando de aplicaciones desarrolladas para utilizarlas en clusters heterogéneos, existen problemas a resolver que principalmente surgen por la heterogeneidad del hardware. Estos problemas son:

- Balance de carga: al tener procesadores con distintas capacidades, se debe realizar un buen balance de carga con el objetivo de que todos los nodos tarden tiempos similares. Si todos los tiempos son similares, se evita que todo el cómputo paralelo se vea penalizado por el tiempo del nodo que más tardó en resolver el problema. De esta forma, primero se debe encontrar el factor que determina la carga (tipo de procesamiento, cantidad de datos, etc.) y luego, repartir la carga de forma correcta entre los nodos.
- Representación de datos: cada arquitectura tiene una representación interna de los distintos tipos de datos. En una aplicación paralela, es muy común que se deba compartir algún dato, por lo que existe la necesidad de que las distintas arquitecturas entiendan los datos provenientes de otras arquitecturas. En el caso de las aplicaciones numéricas, existe el estándar IEEE, que define el formato de los datos de tipos numéricos.

Con todo lo enunciado en este capítulo es fácil ver que la utilización de clusters para cómputo paralelo implica disponer de una arquitectura paralela de muy bajo costo, gran disponibilidad, gran escalabilidad y en la que se puede alcanzar buen rendimiento. Esta última característica depende principalmente de la forma en que se utilicen los recursos disponibles en un cluster. De esta forma, vale la pena plantear soluciones a distintos problemas teniendo en cuenta todas las características de los clusters, con el fin de utilizar de la mejor manera cada uno y evitar penalizaciones en el rendimiento.

Como se verá en los siguientes capítulos, las principales características de la arquitectura que definen características propias del algoritmos son la heterogeneidad de cómputo y la red de interconexión.

Capítulo 4: factorización LU en paralelo

En el capítulo 2 se muestra una solución paralela por bloques tradicional para la factorización LU. Además, se describen 2 formas distintas de distribuir los bloques entre los nodos, distribución secuencial y cíclica. Para la factorización LU se utiliza la distribución cíclica debido al patrón de procesamiento de dicha operación. La solución planteada está orientada a un conjunto de procesadores dispuestos en una grilla bidimensional.

En el capítulo 3 se describe la arquitectura utilizada, analizando primero las principales características de los clusters en general y luego se detallan las características específicas de los clusters de PCs heterogéneos. Dentro de la descripción de estas características se pone especial atención a las propiedades que afectan (o pueden llegar a afectar) el rendimiento del cómputo paralelo.

En este capítulo, se describirá la solución paralela utilizada en este trabajo, la cual tiene en cuenta todas las características de los clusters heterogéneos desde el momento mismo del diseño. Se mostrará el algoritmo implementado (en forma de pseudocódigo) y también se mostrarán distintas formas de distribuir los datos entre los procesos.

4.1 Características de la solución paralela de la Factorización LU

Existen diversas soluciones paralelas para la factorización LU. A pesar de que la idea principal de las distintas soluciones es igual para todas, cada una de estas soluciones difieren en pequeños aspectos definidos principalmente por la arquitectura a la que es orientada.

Por ejemplo, la librería ScaLAPACK implementa, como ya se ha mencionado, operaciones de álgebra lineal optimizadas para computadoras paralelas con memoria distribuida. Es sabido que esta librería logra rendimiento optimizado y buena escalabilidad en las operaciones que implementa [18]. Se toma como ejemplo esta librería ya que es la librería más conveniente (al menos la más cercana) desde el punto de vista de la arquitectura utilizada en este trabajo.

Pero, lamentablemente, el rendimiento logrado por estas operaciones depende del hardware en donde se ejecutan los algoritmos, y esta librería está orientada a arquitecturas paralelas distribuidas tradicionales. Estas arquitecturas tienen ciertas características que definen las propiedades y el rendimiento de los algoritmos implementados.

Estas arquitecturas cuentan con CPUs de alto rendimiento, memoria en cada procesador, y cuentan con una red de interconexión con bajo tiempo de latencia y buen ancho de banda. [18]. Estas características no se encuentran presentes en los clusters (ni homogéneos ni heterogéneos). La red Ethernet utilizada en los clusters tiene poco ancho de banda y mucho tiempo de latencia. Por estos motivos, la comunicación entre los procesos penaliza fuertemente el rendimiento de los algoritmos.

Uno de los objetivos del cómputo paralelo es aumentar el rendimiento de las aplicaciones, por lo tanto, la penalización causada por la red de interconexión es un problema que se debe tratar de solucionar, o en todo caso, se debe tratar de evitar. Una solución a este problema es intentar mejorar la red de interconexión de los clusters. Esto implicaría mejorar el hardware utilizado en la red y poder utilizar los algoritmos propuestos para arquitecturas paralelas tradicionales. Otra posible solución es analizar la arquitectura y revisar los algoritmos propuestos, y si es necesario, volver a escribirlos orientándolos a una arquitectura específica. En el caso de los clusters, se debería intentar escribir soluciones que tengan en cuenta la penalización que implica la comunicación de datos debido a la red de interconexión. En este trabajo se sigue la segunda de las dos posibilidades antes mencionadas: se revisan los algoritmos y se implementan soluciones en función a la arquitectura utilizada.

La utilización de clusters define al menos tres propiedades importantes en el algoritmo que se desea implementar: modelo de programación SPMD (Simple Program, Multiple Data), distribución de datos unidimensional y pasaje de mensajes de tipo broadcast. El modelo de programación SPMD implica que todos los procesos ejecutan el mismo código. Esto es útil en este contexto ya que permite que el balance de carga esté dado por la cantidad de datos asignados a cada proceso. La distribución de datos unidimensional se corresponde con la

manera en que los nodos se conectan entre sí: el protocolo de las redes Ethernet define un bus de interconexión de forma de un arreglo lineal de procesadores (al menos lógico). En las redes Ethernet, como ya se ha mencionado, un mensaje enviado se propaga por todo el bus y llega a todos los nodos conectados al mismo. Un mensaje broadcast es exactamente eso: un mensaje enviado llega a todos los posibles receptores. Luego, existe la posibilidad de utilizar rutinas de comunicación que aprovechen directamente la forma de transmisión de las redes Ethernet.

Para comprender la forma en que se procesan los datos se debe conocer la forma en que se distribuyen los datos entre los nodos. Cuando se realiza una operación por bloques, esto casi siempre implica determinar la forma en que se distribuyen los bloques entre los nodos. En las siguientes secciones se describen los principios de la distribución de carga y luego se muestra el algoritmo. Por último se describen los distintos métodos de distribución de carga implementados y evaluaron en este trabajo.

4.2 Principios de la distribución de carga

En aplicaciones numéricas, el balance de carga está definido directamente por la cantidad de operaciones en punto flotante que realiza cada nodo. Por lo tanto, la asignación de los bloques que cada nodo debe procesar es lo que definirá el balance de carga.

Existen distintas formas de dividir una matriz en bloques. Esta división depende de la arquitectura que se esté utilizando. En el capítulo 2 se mostró una división de bloques donde cada bloque tenía una cierta cantidad (r) de filas y de columnas. Esta forma de dividir la matriz en bloques es adecuada para una computadora con los procesadores dispuestos en grilla. Pero, en este caso los clusters presentan una arreglo lineal de procesadores, por lo que se dividirá la matriz en bloques en una sola dimensión. O sea, se divide la matriz en bloques de (r x n) o (n x r) elementos, donde n es el orden de la matriz y r es el tamaño de bloque. Estas formas de dividir la matriz se llaman división por filas o por columnas respectivamente.

La figura 4.1 muestra la división por filas de la matriz. Cada bloque es un número entero de filas. A su vez, los bloques están compuestos por filas enteras: si la dimensión de la matriz es n x n y la dimensión del bloque es r, entonces, cada bloque será de r x n elementos.

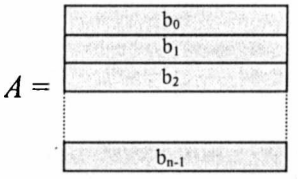

Figura 4.1: partición en bloques por filas.

La figura 4.2 muestra la división de la matriz por columnas. En este caso, cada bloque es un número entero de columnas y está formado por n x r elementos.

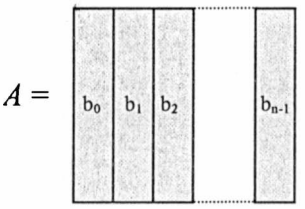

Figura 4.2: partición en bloques por columnas.

Ambas formas de dividir la matriz (por filas o por columnas) son buenas para una arquitectura donde los nodos tienen una topología lineal. En este trabajo se divide la matriz en bloques de filas. Esta elección se realiza teniendo en cuenta que en el lenguaje con el que se implementa el código los arreglos se almacenan como una única fila (un arreglo unidimensional con todas las filas de la matriz almacenadas consecutivamente).

Como ya se ha mencionado, el objetivo de todo método de distribución de datos es lograr un buen balance de carga. Obtener un buen balance de carga implica que todos los nodos trabajen tiempos similares. Para esto, en la factorización LU es necesario dividir a la matriz en muchos más bloques que nodos haya en el cluster, y estos bloques se distribuyen en forma cíclica debido al patrón de procesamiento de dicha factorización. De esta forma se logra que todos los nodos trabajen en la mayor cantidad de iteraciones posibles.

Por ejemplo, si se tiene un cluster con 4 nodos, $p_0..p_3$, y una matriz dividida en bloques por filas, la distribución es de la forma que se ilustra en la figura 4.3, donde a cada bloque se lo etiqueta con el procesador al cual es asignado.

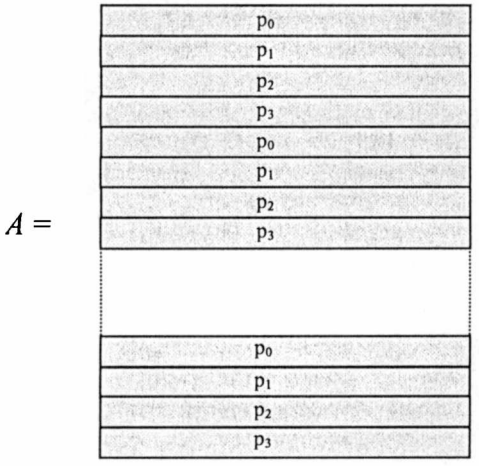

Figura 4.3: asignación de bloques cíclica

Si P es la cantidad de nodos y k es la cantidad de bloques en que se divide la matriz, entonces, el nodo p_i recibirá todos los bloques b_j tales que $i = (j \bmod P)$, con j 0 j k-1.

4.3 Procesamiento de pivotes

Es común incluir en esta factorización lo que se conoce como técnica de pivoteo. Esta técnica se incluye en la factorización con el objetivo de mantener la estabilidad numérica. Lo que se realiza en este procesamiento es, antes de factorizar cada uno de los datos, buscar el elemento con mayor valor absoluto de toda la fila a la que pertenece el elemento y se intercambian las columnas de este máximo y la del elemento a factorizar. Si la solución a la factorización se implementa por columnas, se busca el máximo en toda la columna y se intercambian las filas.

Dado que la matriz está dividida en bloques de filas, si hay más de un nodo las columnas se encuentran repartidas entre los mismos. Por esto, cada vez que se intercambian columnas, todos los nodos deben realizar el mismo intercambio, de otra forma no se estaría intercambiando toda la columna entera y los datos de la matriz quedarían inconsistentes. De esta forma, es necesario transmitir los índices de las columnas que participan en cada uno de los intercambios. Usando esta información, todos los nodos realizan el mismo intercambio y se logra intercambiar toda la columna entera. La información necesaria para realizar estos intercambios es, para cada elemento factorizado, su índice y el índice de la columna por la que ha sido intercambiado. Es posible implementar esto utilizando un vector donde el índice se corresponde con la ubicación del dato factorizado (ubicación relativa con respecto a su ubicación dentro del bloque) y el dato del vector en cada posición es el índice de la columna con la que se intercambió.

4.4 Procesamiento local en cada nodo

El método de factorización explicado en el capítulo 2 se basó en una división bidimensional de la matriz. En el caso de la solución propuesta para clusters la distribución de bloques es unidimensional, por lo que se explicará brevemente cómo afecta este hecho a la factorización.

Siguiendo una división unidimensional de bloques donde cada bloque es de r x n elementos (con r tamaño de bloque y n es la dimensión de la matriz). La figura 4.4 muestra cómo se divide la matriz a factorizar.

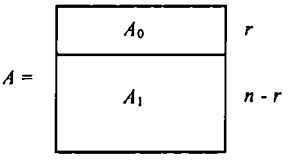

Figura 4.4: Partición de la matriz en un bloque de filas.

Entonces, la matriz A se divide en las submatrices A_0 y A_1, de r x n y $(n\text{-}r)$ x n elementos respectivamente. Si se realiza la factorización LU a la submatriz A_0 se obtienen L_{00}, U_{00} y U_{01} como se observa en la Figura 4.5:

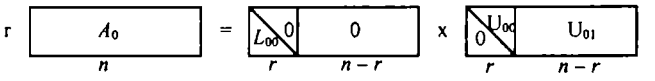

Figura 4.5: Factorización LU de un bloque de filas.

Se puede considerar a la matriz A_0 dividida en A_{00} y A_{01}, ya que la factorización LU se realiza sobre una submatriz cuadrada (A_{00}). Esta división se muestra en la figura 4.6:

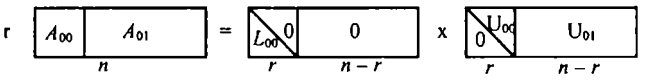

Figura 4.6: División de A_0 en A_{00} y A_{01}.

de esta forma, se resuelve:

$$A_{00} = L_{00} \text{ x } U_{00} \tag{4.1}$$
$$A_{01} = L_{00} \text{ x } U_{01} \tag{4.2}$$

estas ecuaciones se corresponden con las ecuaciones 2.11 y 2.12 (del capítulo 2) respectivamente. Luego de esto, resta hallar la submatriz A_1 de la figura 4.4.

Con el objetivo de seguir paso a paso la factorización LU con la matriz dividida en bloques de filas, se puede considerar a la matriz A_1 de la figura 4.4 dividida como se muestra en la figura 4.7 (en dicha figura se muestra toda la matriz A y las matrices resultantes correspondientes a L y U).

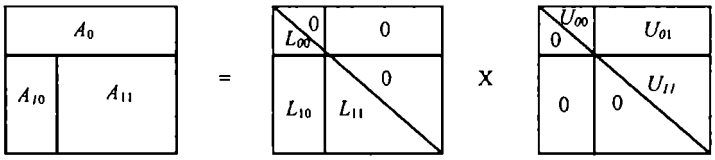

Figura 4.7: División de la submatriz A_1 por bloques.

Entonces, A_1 se divide en las submatrices A_{10} y A_{11} de *(n-r)* x *r* y *(n-r)* x *(n-r)* elementos respectivamente. Ahora, se deben hallar las matrices L_{10}, L_{11} y U_{11}

$$A_{10} = L_{10} \text{ x } U_{00} \tag{4.3}$$
$$A_{11} = L_{10} \text{ x } U_{01} + L_{11} \text{ x } U_{11} \tag{4.4}$$

que son las ecuaciones 2.13 y 2.14 respectivamente.

La submatriz L_{10} es calculada utilizando la ecuación 2.16:

$$L_{10} = A_{10} x U_{00}^{-1} \tag{2.16}$$

y las matrices L_{11} y U_{11} son halladas aplicando el mismo método iterativo al resultado de $A_{11} - L_{10}$ x U_{01} (matriz actualizada) [18].

El algoritmo básico que resuelve la factorización LU por bloques sigue el modelo SPMD, donde, como ya se mencionó, en todos los nodos se ejecuta el mismo algoritmo. Este algoritmo itera sobre cada uno de los bloques en que se divide la matriz.

Básicamente el algoritmo que resuelve la factorización es el mostrado en la figura 4.8:

```
for ( i = 0 ; i < k-1 ; i++)
  if (almaceno bloque i)
  {
    FACTORIZACION bloque i
    BROADCAST bloque factorizado i
  }
  else
    RECIBIR bloque factorizado i
  ACTUALIZACION de los bloques locales
```

Figura 4.8: factorización LU paralela por bloques.

Donde k es la cantidad de bloques en que se divide la matriz. La actualización de los bloques locales es específicamente: intercambio de columnas a partir del vector de pivotes, obtener el bloque L (L_{10}=A_{10} x U_{00}^{-1}, ecuación 2.16) y actualizar el resto de la matriz (L_{11} x $U_{11} = A_{11} - L_{10}$ x U_{01}, ecuación 2.17).

Todos los nodos realizan el mismo procesamiento, que es básicamente factorizar un bloque y actualizar el resto de la matriz (parte activa) de acuerdo al bloque factorizado. Por lo tanto, una vez que se factoriza un bloque es necesario que todos los nodos tengan dicho bloque factorizado para poder realizar la actualización. Además, todos los nodos necesitan conocer el índice de las columnas que se intercambiaron antes de cada eliminación (procesamiento de pivotes). Esto implica que en cada iteración el proceso que factorizó el bloque activo comunique a los demás nodos el bloque factorizado y el vector relativo a los pivotes. Esta comunicación se puede resolver con un único mensaje broadcast.

Se puede observar en el pseudocódigo mostrado en la figura 4.8 que el algoritmo itera tantas veces como bloques haya. En cada una de las iteraciones el nodo que tiene el *bloque activo* (bloque a factorizar) lo factoriza y luego lo envía junto con los índices de los pivotes. El resto de los nodos esperan recibir el bloque factorizado y la información de los pivotes y todos los nodos (el que realizó la factorización y los que esperaron la comunicación) actualizan los bloques locales que todavía participan en la factorización.

En cada una de las iteraciones todos los nodos que no tienen el *bloque activo* esperan recibir el bloque factorizado y después realizan la actualización. Esa espera influye en el rendimiento total del algoritmo, por lo que se debería tratar de evitar.

Una forma de lograrlo es reorganizar los pasos de cómputo local y comunicación de tal forma que los nodos no estén ociosos mientras esperan el bloque factorizado. Esta modificación se muestra en la figura 4.9.

```
for ( i = 0 ; i < k-1 ; i++)
  if (almaceno bloque i)
  {
    if (i > 0)
      ACTUALIZACION bloque i utilizando bloque activo (i-1)
    FACTORIZACION bloque i (próximo bloque activo)
    ENVIO del bloque i a todos los procesos (próximo bloque activo)
  }
  if (i > 0)
    ACTUALIZACION de parte activa de la matriz con bloque activo(i-1)
  if (no almaceno bloque i)
    RECIBIR nuevo bloque activo (i)
```

Figura 4.9: Pseudocódigo de la solución paralela por bloques de la factorización LU que solapa cómputo y comunicación

Donde *k* es la cantidad de bloques en que se divide la matriz. Con esta modificación, mientras el primer nodo factoriza el primer bloque, el resto de los nodos están esperando recibir el bloque factorizado (*bloque activo*) y los pivotes. Cuando estos reciben el *bloque activo*, todos los procesos actualizan todos sus bloques (de la parte activa de la matriz) excepto el nodo que almacena el *próximo bloque activo*. Este nodo actualiza, factoriza y envía el próximo nodo activo antes de realizar la actualización del resto de la matriz. De esta forma, se solapa la factorización del *próximo bloque activo* con la actualización del resto de la matriz. Esto permite que exista la posibilidad de tener resuelta la factorización del *próximo bloque activo* antes de necesitarlo.

Durante todas las iteraciones de la factorización excepto la primera se trabaja con 2 bloques: en la iteración *i*, el *bloque activo* (bloque *i*-1) y el *próximo bloque activo* (bloque *i*). El bloque activo (*i*-1) ya ha sido factorizado y se lo utiliza para actualizar el próximo bloque activo (sólo el proceso que tiene el bloque *i*) y para actualizar los bloques restantes de la matriz (todos los procesos).

El algoritmo implementado en este trabajo sigue la idea del pseudocódigo mostrado en la figura 4.9. Además se implementaron y se diseñaron distintas formas de distribuir los datos (siempre de forma cíclica) las cuales se explicarán en las próximas subsecciones.

4.5 Métodos de distribución de carga

En las aplicaciones paralelas, múltiples procesos trabajan de forma conjunta (y cooperativa) para solucionar un problema. Entonces, cada uno de los procesos resolverá una parte del problema total. Así, el problema original se divide en partes y las mismas se distribuyen entre los procesos que participan. La tarea de dividir el problema en partes y luego

distribuir dichas partes entre los procesos es lo que realiza un método de distribución de carga.

Como es sabido, el tiempo total de una aplicación paralela es determinado por el tiempo del nodo que más tiempo tarda en resolver su parte del trabajo. Entonces, logrando que todos los nodos tarden tiempos similares, se evita que el tiempo total de resolución dependa del nodo "más lento" (evitando así una posible penalización en el tiempo total de resolución).

Existen dos formas clásicas de dividir los problemas: por funciones o por datos. Cuando se divide un problema por funciones, se definen cada una de las funciones distintas que tiene el problema y estas funciones se asignan a los distintos procesos. O sea, cada proceso realiza funciones distintas y una vez que se realizan todas las funciones, se resuelve el problema completo. Cuando se divide por datos, son los datos los que se dividen y se distribuyen entre los distintos procesos. Cada proceso realiza la misma función sobre distintos datos.

Como ya se ha mencionado, en las aplicaciones numéricas el balance de carga está directamente definido por la distribución de los datos. Esto es, se dividen los datos entre todos los nodos que componen el cluster (en cada nodo se ejecuta un proceso) y la cantidad de trabajo que realiza cada nodo depende directamente de cuántos datos se le asigna (y con los cuales trabaja). Como la solución a la factorización LU se define por bloques, esto implica que durante la distribución se distribuyan bloques enteros de datos. Así, el método de distribución se encargará de determinar a qué procesador se asigna cada uno de los bloques en que se divide la matriz.

Un método de distribución de carga definirá qué trabajo (y cuánto) va a realizar cada uno de los procesos. En el ámbito de la programación paralela, donde se busca obtener buen rendimiento, es importante lograr el mínimo de tiempo posible en resolver un problema. Esto se traduce directamente a lograr un buen balance de carga: lograr que todos los nodos tarden tiempos similares en resolver su parte del problema (que no necesariamente es lo mismo que realizar la misma cantidad de trabajo). Naturalmente, el tiempo de resolución de un problema depende de la velocidad con que se lo resuelve (en este caso, la potencia de los nodos). Entonces, una forma de lograr un buen balance de carga es repartiendo el trabajo en función de la capacidad de procesamiento de cada uno de los nodos.

En el contexto de los clusters, se debe disponer de la potencia de cómputo de cada uno de los nodos que componen el cluster. Esta medida debe ser un número representativo de las diferencias de las potencias de cómputo de cada uno de los nodos. La potencia total del cluster es la suma de todas las potencias de cada uno de los nodos que lo componen. En este punto, se agrega un problema más que es calcular esta medida representativa de la potencia de cada nodo. Una forma muy común de hacerlo es calculando directamente los Mflop/s de cada uno de los nodos, utilizando algún algoritmo específico para evaluarlo.

Una forma de calcular los Mflop/s de cada nodo es utilizar el mismo problema pero en dimensiones más chicas [19], de forma tal que se pueda resolver el problema utilizando un solo nodo. Esta forma de obtener la potencia de cada nodo es útil ya que al resolver el mismo problema pero en dimensiones más chicas, seguramente el comportamiento (por lo

tanto los Mflop/s) serán aproximados al rendimiento final con el tamaño de problema real ya que se trata del mismo problema. En el próximo capítulo, cuando se den las características de las experimentaciones realizadas, se explicará cómo se obtuvo la potencia de cada nodo en este trabajo.

Sea un cluster formado por n nodos ($p_0..p_{n-1}$), y $P(p_i)$ la potencia del nodo p_i. Entonces, la potencia total del cluster, P_{total}, está dada por la ecuación 4.5:

$$P_{total} = \sum_{i=0}^{n-1} P(p_i) \tag{4.5}$$

Entonces, en un cluster de potencia P_{total}, la relación entre la potencia de cada nodo $P(p_i)$ y la cantidad de trabajo total que debería realizar está dada por la ecuación 4.6:

$$\text{Trabajo a realizar por } p_i = \frac{P(p_i)}{P_{total}} \text{ del trabajo total} \tag{4.6}$$

Entonces, se conoce la cantidad de trabajo que se debe realizar y se conoce la cantidad de trabajo que debe realizar cada nodo. Estos datos sirven para definir métodos de distribución de carga que logren asignar el trabajo en función de la potencia de cómputo de cada nodo.

Cuando el cluster es homogéneo, la distribución de datos es trivial ya que cada uno de los nodos debería recibir la misma cantidad de datos, y con esto se lograría que todos los nodos trabajen tiempos similares. En clusters heterogéneos no pasa lo mismo. Los nodos tienen distintas capacidades de cómputo, entonces, asignar la misma cantidad de trabajo implicaría que los nodos más veloces queden ociosos y todo el tiempo dependa del tiempo de resolución del más lento (o de los nodos más lentos). De esta forma, la distribución de carga en clusters heterogéneos no es trivial y es importante distribuir de forma correcta los datos.

Durante este trabajo se propusieron distintos métodos para distribuir los datos en ambientes heterogéneos. Con todos estos métodos se han realizado pruebas y analizado los resultados obtenidos. Los distintos métodos tratan de mejorar uno a uno el balance de carga obtenido.

A pesar de que en este trabajo se utilizan clusters heterogéneos, se explica un método de distribución para clusters homogéneos, pues este método incluye una técnica (dividir los bloques en grupos), la cual se utilizará en los métodos para clusters heterogéneos.

4.5.1 Cluster Homogéneo

Cuando el cluster es homogéneo, todos los nodos son iguales, por lo tanto tienen la misma capacidad de procesamiento. En este caso, la distribución de datos es directa, ya que se puede asignar la misma cantidad de bloques a cada uno de los nodos que componen el cluster. Entonces, si los nodos tienen la misma capacidad y la misma cantidad de trabajo, todos los nodos tardarán tiempos similares. Como ya se explicó anteriormente, la mejor

forma de distribuir los datos en esta operación es de forma cíclica, para evitar un posible desbalance de carga durante la factorización.

Los bloques de la matriz se dividen en grupos [4]. Los bloques de cada uno de los grupos se distribuyen de forma cíclica entre todos los nodos que componen el cluster. En el caso homogéneo, cada grupo tiene tantos bloques como nodos haya. De esta forma cada nodo recibe un bloque de cada grupo y obviamente, el tamaño de grupo es igual a la cantidad de nodos que componen el cluster.

Por ejemplo, si el cluster está compuesto por 3 nodos (p_0, p_1 y p_2) y se divide una matriz en 6 bloques, cada grupo está formado por 3 bloques y la distribución cíclica de los bloques queda como se ilustra en la figura 4.10. En dicha figura cada grupo está sombreado con un tono distinto. Las flechas muestran como se distribuyen los bloques de los primeros dos grupos.

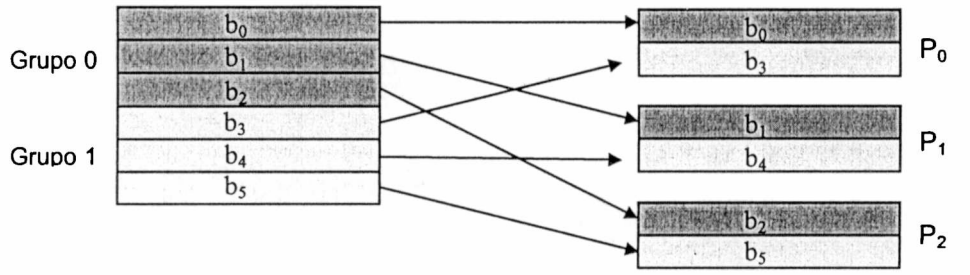

Figura 4.10: Distribución cíclica utilizando grupos en un cluster homogéneo.

Cuando la cantidad de bloques en total no es múltiplo del tamaño de grupo, queda un grupo con menos bloques que el resto. La estrategia de distribución de este bloque es la misma que para los grupos completos: se asignan de a un bloque desde el primer nodo hasta que se terminen los bloques. Esto se ilustra en la figura 4.11, donde las flechas punteadas muestran la distribución del grupo incompleto.

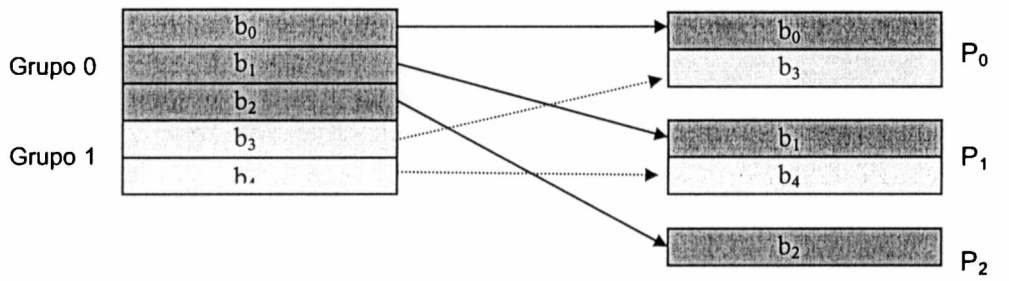

Figura 4.11: Distribución cíclica con grupos en un cluster homogéneo. El último grupo es incompleto.

Entonces, el balance de carga obtenido es cercano al óptimo. Esto es así porque los bloques se distribuyen enteros (no se dividen) y en el caso de que quede un último grupo

incompleto, habrá nodos que reciban un bloque más que otros. Entonces, como máximo habrá un bloque de diferencia entre los distintos nodos (p_2 con p_0 y p_1 en la figura 4.11).

Este método no se utilizó para las experimentaciones, ya que se han utilizado clusters heterogéneos y este método no obtendrá buen balance de carga en este contexto. Igualmente se incluye su descripción porque agrega el tratamiento de los bloques divididos en grupos, lo cual se utiliza en los siguientes métodos para clusters heterogéneos.

En este punto, vale la pena notar que si un método de distribución de carga depende de las capacidades de cómputo de los nodos, entonces, un método de distribución para clusters heterogéneos, debería servir también para clusters homogéneos. De esta forma, se puede generalizar un método de distribución de carga sin importar cuál sea el grado de heterogeneidad del cluster.

4.5.2 Clusters heterogéneos

Cuando el cluster está formado por nodos con diferentes potencias de procesamiento, la distribución de datos no es tan directa como en el caso homogéneo.

Se han analizado distintos posibles métodos para distribuir los bloques. Los distintos métodos tienen en común las siguientes características:

- Distribuyen los bloques de la matriz de forma cíclica.
- Distribuyen los bloques en función de la capacidad de cómputo de cada nodo.
- Dividen los bloques en grupos.

Pero difieren principalmente en dos cosas:

- Cantidad de bloques por grupos.
- Forma de distribuir los bloques que no forman un grupo completo.

Como en el caso homogéneo, se dividen los bloques en grupos. Si se aplica la distribución explicada anteriormente para el caso homogéneo, todos los nodos reciben la misma cantidad de trabajo (o aproximadamente la misma cantidad). Esta distribución no es buena para un cluster heterogéneo, ya que los nodos más rápidos terminarían su trabajo y tendrían que esperar a los nodos más lentos a que terminen, y de esta forma todo el tiempo de cómputo dependería del nodo más lento.

Entonces, se deben repartir los bloques de alguna forma en que los nodos más rápidos reciban más bloques que los más lentos. Los bloques están divididos en grupos y la cantidad de bloques que recibe cada nodo de cada grupo será en función de su capacidad de procesamiento (para todos los grupos es la misma cantidad).

Se asume que los nodos están ordenados de forma decreciente por su capacidad de procesamiento. De este modo, se cumple que: $P(p_0) > P(p_1) > \ldots > P(p_{n-1})$. Esta disposición de los nodos es conveniente para lograr un mejor balance de carga (esto se podrá observar a

medida que se expliquen los métodos de distribución). A continuación se describirán los diferentes métodos.

4.5.2.1 Método "directo"

Como primera aproximación, se propone un método muy simple y directo. Para este método, como las potencias están expresadas en términos de los Mflop/s de cada nodo, solamente se tiene en cuenta la parte más significativa de cada potencia (para evitar trabajar con números grandes). Entonces, se trabaja directamente en término de cientos de Mflop/s de cada nodo. Así, $P(p_i)$ son los cientos de Mflop/s del nodo p_i.

Para esta primera aproximación se hace que cada grupo esté compuesto por P_{total} bloques y se distribuyen los bloques del grupo teniendo en cuenta directamente cada potencia $P(p_i)$, o sea, a p_0 se le asignan los primeros $P(p_0)$ bloques del grupo, a p_1 se le asignan los siguientes $P(p_1)$ bloques, y así con todos los nodos y con cada uno de los grupos.

De esta forma, se obtiene que cada nodo recibe una cantidad de bloques proporcional a su capacidad de cómputo, como es el objetivo. Esto se puede comprobar del siguiente modo (en las siguientes ecuaciones se utiliza el símbolo "#" para expresar cantidad):

$$\#\,grupos = \frac{\#\,bloques}{P_{total}} \tag{4.7}$$

La ecuación 4.7 define la cantidad de grupos en que se dividen los bloques. Luego, si p_i recibe $P(p_i)$ de cada grupo, se tiene que

$$\#\text{ bloques que recibe } p_i = \quad P(p_i) * \quad \#\,grupos \tag{4.8}$$

Usando la ecuación 4.7:

$$\#\text{ bloques que recibe } p_i = \quad P(p_i) * \quad \frac{\#\,bloques}{P_{total}} \tag{4.9}$$

$$\#\text{ bloques que recibe } p_i = \quad \frac{P(p_i)}{P_{total}} * \#\,bloques \tag{4.10}$$

Luego, la ecuación 4.10 muestra que la cantidad de trabajo que recibe cada nodo es correcto, ya que esta cantidad está en función del trabajo que debería recibir según la ecuación 4.6. En la ecuación 4.6 se hace mención al "trabajo total" el cual es directamente proporcional a la cantidad de bloques en que se divide la matriz. Por esto se corresponden ambas ecuaciones (4.6 y 4.10).

Como ejemplo, se considera una matriz dividida en 14 bloques y un cluster formado por 3 nodos, p_0, p_1 y p_2, con potencias $P(p_0) = 3$, $P(p_1) = 2$ y $P(p_2) = 1$ respectivamente. Por la

ecuación 4.5 se tiene que $P_{total} = 6$. Entonces, utilizando la ecuación 4.6, p_0 debe realizar 3/6 del trabajo total (50%), p_1 debe realizar 2/6 del trabajo total (aproximadamente el 33%), y p_2 debe realizar 1/6 (aproximadamente el 17%) del trabajo total.

Realizando la distribución descrita anteriormente, cada grupo tiene 6 bloques, y se distribuyen 3 bloques para p_0, 2 bloques para p_1 y 1 bloque para p_2 de cada uno de los grupos en que se dividen los bloques de la matriz. La figura 4.12 ilustra este ejemplo.

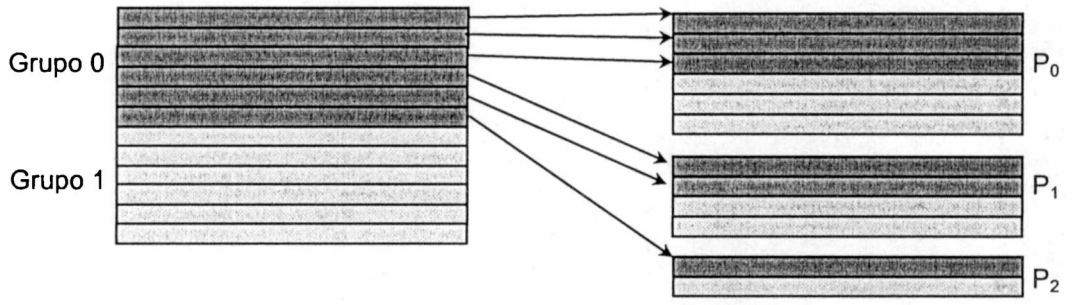

Figura 4.12: Distribución cíclica secuencial por grupos

Luego, p_0 realizará el 50% del trabajo total, p_1 realizará el 33%, y p_2 17% del trabajo total, como se quería obtener.

Este método tiene las siguientes características:

- Implementación fácil y directa.
- Cuando la cantidad de bloques y el tamaño de grupo son apropiados, se obtiene balance de carga apropiado.

De todas maneras, el buen funcionamiento del método es totalmente dependiente de la cantidad de bloques y del tamaño de grupo. La mayoría de las veces, se puede observar que:

- El tamaño de grupo resultante es muy grande.
- La cantidad de bloques del grupo (tamaño del grupo) no divide a la cantidad de bloques total de la matriz (quedando un último grupo incompleto).

La primera observación genera pocos grupos con muchos bloques cada uno. Por otro lado, este método reparte los bloques de todos los grupos de la misma forma (incluyendo el último grupo que posiblemente, pueda tener menos bloques). Esto hace que haya nodos que reciben bloques del último grupo y otros que no. Si se tiene en cuenta que p_i recibe $P(p_i)$ bloques consecutivos del grupo, y los bloques están compuestos por múltiples filas, esto puede llegar a producir una diferencia importante de trabajo asignado entre nodos consecutivos.

Por ejemplo, con el cluster idéntico al ejemplo anterior (3 nodos con potencias 3, 2 y 1), pero con una matriz con 15 bloques se obtiene:

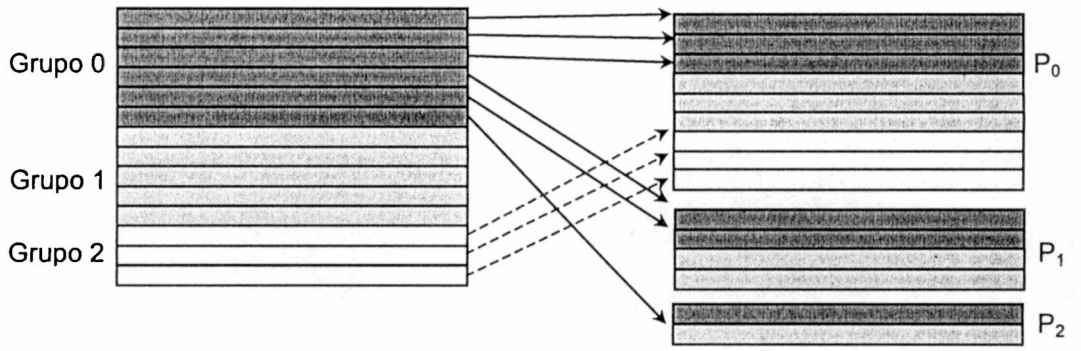

Figura 4.13: Distribución cíclica usando grupos con la primer aproximación,

En el ejemplo se puede observar cómo los bloques del último grupo (con sólo 3 bloques) es asignado al nodo p_0. De esta forma tenemos la siguiente distribución de trabajo: p_0 recibe 9/15 (60%) del trabajo total, p_1 recibe 4/15 (aproximadamente el 27%) del trabajo total y p_2 recibe 2/15 (aproximadamente el 13%) del trabajo total. Con esto, vemos que los porcentajes de trabajo asignado a cada nodo difiere de los porcentajes óptimos: 50%, 33% y 17% respectivamente.

Cuando ocurre esto, la diferencia entre el trabajo real asignado y el óptimo hace que el balance de carga no sea óptimo, y esto afectará en el tiempo total de cómputo: no se lograría que todos los nodos tarden tiempos similares en realizar el cómputo, viéndose penalizado por el/los nodos que más tarden en realizar su trabajo (que en este caso, seguramente serán los nodos que reciben más trabajo del que realmente deberían recibir).

4.5.2.2 Método directo mejorado (secuencial)

El método propuesto trata de armar grupos con pocos bloques, y distribuye de forma especial los bloques del último grupo si éste es incompleto. De esta forma, intenta obtener un buen balance de carga independientemente de factores tales como tamaño de grupo, tamaño de bloque, tamaño de la matriz, etc., que determinan la distribución final de los bloques.

Para esto, el método propuesto realiza ciertos cálculos sobre las potencias de tal manera que se mantenga la relación de potencia de cómputo entre los nodos, pero generando grupos con menos bloques. Como en el caso anterior, las nuevas potencias definen cuántos bloques de cada grupo recibe cada nodo. Una vez que se obtienen las nuevas potencias, éstas se usan de igual forma que en el método anterior para repartir los bloques entre los nodos. Además, si el último grupo es incompleto se realizan nuevos cálculos sobre las potencias para resolver la distribución de los bloques pertenecientes al grupo incompleto.

Los cálculos realizados sobre las potencias son:

1) se obtiene el nodo con potencia mínima (p_{min})
2) se dividen todas las potencias por la potencia del nodo mínimo ($P(p_{min})$) y luego se multiplican por 10.
3) se obtiene el máximo común divisor (*mcd*) de estas potencias.
4) se dividen todas las potencias por el máximo común divisor.

A continuación se describen cada uno de los pasos:

Paso 1

Se busca entre todas las potencias el nodo de potencia menor (p_{min}). Este valor será usado en los próximos pasos. Como ya se mencionó, es conveniente disponer los nodos ordenados de mayor a menor potencia. De esta forma, este paso se reduce a obtener la potencia del último nodo del cluster.

Paso 2

Se divide cada una de las potencias por la potencia mínima $P(p_{min})$. Con esto, cada una de las potencias queda en relación a la potencia mínima. O sea, se puede ver como la cantidad de trabajo que realiza cada nodo cuando p_{min} realiza "una unidad" de trabajo.

Los bloques se distribuyen como unidades (bloques enteros), por lo que las potencias nuevas que se buscan deben ser un número entero. Puede ocurrir que el resultado de la división anterior no sea entera. En este caso, se aproxima este número decimal al número entero más cercano con el objetivo de que se pierda la menor precisión posible (pues esto afectará al balance de carga final).

Por ejemplo: si se tiene una máquina paralela compuesta por 4 nodos ($p_0..p_3$) cuyas potencias son:

$P(p_0) = 2544$
$P(p_1) = 2345$
$P(p_2) = 1556$
$P(p_3) = 1288$

Así, $p_{min} = p_3$ con $P(p_{min}) = 1288$. Se dividen todas las potencias por $P(p_{min})$, y se obtiene:

$P(p_0) = 1.9751$
$P(p_1) = 1.8206$
$P(p_2) = 1.2080$
$P(p_3) = 1$

Como se distribuyen bloques enteros entre los nodos, se considera solamente la parte entera de cada potencia. Se aproximan a su número entero más próximo (si la parte decimal es mayor a 5 se aproxima al número entero siguiente), quedando:

$P(p_0) = 2$
$P(p_1) = 2$
$P(p_2) = 1$
$P(p_3) = 1$

De esta forma se puede observar que suceden particularmente 2 cosas:

1) Cuando se redondea al próximo número menor, por ejemplo p_2, estamos desperdiciando un 20% de la potencia con respecto a la menor (porque p_2 es un 20% mas rápida que p_3). Esto hace que p_2 termine (en teoría) un 20% antes que p_3 (se le está asignando el 80% de lo que se le debería asignar).
2) Cuando se aproxima al próximo número mayor, por ejemplo p_1, estamos asignando un 18% más de potencia con respecto a la menor. Esto hace que p_1 (en teoría) termine un 18% después que p_3 (se le está asignando el 118% de lo que se le debería asignar).

Estos casos se generalizan cada vez que se aproxima un número no entero. La diferencia máxima que se puede llegar a producir es de un 50% en relación a la potencia mínima (este caso se da cuando una potencia queda con valor 5 como parte decimal luego de la división). Una forma de reducir esta diferencia del 50% en relación a la menor potencia, es tomar una parte más grande de las potencias divididas para luego aproximar (además de tomar la parte entera, se considera el primer decimal). Para esto, luego de dividir las potencias por la potencia mínima, se multiplican por 10, haciendo así que la parte entera ahora incluya también al primer decimal y cuando se aproxima el número, en vez de tener una diferencia de cómo máximo 50%, sea de a lo sumo 5%.
Realizando estos cálculos con el ejemplo anterior:

$P(p_0) = 1.9751 \Rightarrow 19.751$
$P(p_1) = 1.8206 \Rightarrow 18.206$
$P(p_2) = 1.2080 \Rightarrow 12.080$
$P(p_3) = 1 \Rightarrow 10$

Y de estos resultados sólo se considera la parte entera:

$P(p_0) = 19$
$P(p_1) = 18$
$P(p_2) = 12$
$P(p_3) = 10$

De esta forma, sólo se está teniendo una diferencia de a lo sumo 5% de cada potencia en relación a la menor.

Paso 3

Se obtiene el máximo común divisor de todas estas potencias para utilizarlos en el próximo paso. Para encontrar este máximo común divisor se puede utilizar cualquier algoritmo de los conocidos como puede ser el método de Euclides [26].

Si los números generados en los pasos anteriores son coprimos (el máximo común divisor es 1), quedarían las mismas potencias que se obtuvieron en el paso 2. Esto generaría grupos con muchos bloques, ya que como mínimo tenemos 10 bloques por nodo (el nodo mínimo recibe 10 bloques y el resto recibe más). Y como se verá durante la experimentación, no es conveniente generar grupos con mucha cantidad de bloques.

Entonces, cuando el máximo común divisor es 1, se modifican todas las potencias impares, reduciéndolas en uno para lograr que todas las potencias queden pares. Luego, se vuelve a calcular el máximo común divisor (luego de la modificación, como mínimo es 2). Haciendo esta modificación en las potencias obtenidas se pierde precisión, y esto en definitiva genera un poco de desbalance de carga, pero igualmente es conveniente penalizar un poco de precisión con el objetivo de obtener grupos con menos bloques.

Paso 4

Este paso reduce todas las nuevas potencias obtenidas manteniendo la relación de capacidades entre los nodos. Una vez hallado el máximo común divisor de todos los números (paso 3), se lo utiliza para dividir todos los números. Con esto, se obtienen potencias más chicas, pero se mantiene la relación entre las mismas (pues todas se dividen por el mismo número). En este punto vale considerar que todos los resultados de esta división son números enteros (por ser común divisor).

Luego, estas potencias se utilizan para formar cada grupo y repartir los bloques de cada grupo entre los nodos. La distribución de los bloques de cada grupo entre los nodos se realiza de la misma forma que en el método anterior.

A pesar de que las potencias originales se han modificado, se sigue manteniendo (aproximadamente) la relación de capacidades entre los nodos y la cantidad de trabajo que recibe cada nodo. Esto se puede comprobar en las ecuaciones 4.11 a 4.18. Originalmente, $P(p_i)$ es la potencia de cálculo de p_i. Por otra parte, P_{total} es la potencia total del cluster. Se quiere ver que la relación mostrada en la ecuación 4.6 se sigue manteniendo a pesar de las modificaciones realizadas sobre las potencias. Utilizando la ecuación 4.6, y si se llama $P'(p_i)$ a la nueva potencia obtenida después de los cálculos enumerados, y sea P'_{total} la nueva potencia total del cluster a partir de las potencias obtenidas, se debe verificar la ecuación 4.11 para demostrar que la relación entre las potencias se mantiene a pesar de haber sido modificadas:

$$\frac{P(p_i)}{P_{total}} \cong \frac{P'(p_i)}{P'_{total}} \tag{4.11}$$

Por los pasos enunciados anteriormente, se tiene que:

$$P'(p_i) = \left(\frac{P(p_i)}{P_{min}} \right) * 10 / mcd \tag{4.12}$$

Donde *mcd* es el máximo común divisor. La ecuación 4.12 se verifica para todos los casos en que las potencias no hayan quedado coprimas en el paso 3. Por el contrario, las potencias impares se redujeron en 1 como se explicó anteriormente (con lo que en la ecuación 4.12 se utilizaría $\cong$ en vez de =). Además,

$$P'_{total} = \sum_{i=0}^{n-1} P'(p_i) \tag{4.13}$$

Entonces, usando las ecuaciones 4.12 y 4.13:

$$\frac{P'(p_i)}{P'_{total}} = \frac{\frac{P(p_i)*10}{P_{min}*mcd}}{\sum_{i=0}^{n-1} P'(p_i)} \tag{4.14}$$

Y utilizando las ecuaciones 4.12 en el denominador:

$$\frac{P'(p_i)}{P'_{total}} = \frac{\frac{P(p_i)*10}{P_{min}*mcd}}{\sum_{i=0}^{n-1} \frac{P(p_i)*10}{P_{min}*mcd}} \tag{4.15}$$

Se reescribe la ecuación:

$$\frac{P'(p_i)}{P'_{total}} = \frac{P(p_i)*\frac{10}{P_{min}*mcd}}{\sum_{i=0}^{n-1} P(p_i)*\frac{10}{P_{min}*mcd}} \tag{4.16}$$

Cancelando los términos comunes de la ecuación 4.16:

$$\frac{P'(p_i)}{P'_{total}} = \frac{P(p_i)}{\sum_{i=0}^{n-1} P(p_i)} \tag{4.17}$$

Y por la ecuación 4.5, se llega a que:

$$\frac{P'(p_i)}{P'_{total}} \cong \frac{P(p_i)}{P_{total}} \tag{4.18}$$

Con lo que se puede ver que se satisface la ecuación 4.11. En esta ecuación se utiliza el símbolo "≅" ya que se realizan aproximaciones durante los cálculos (redondeos en las divisiones y restas en el caso de números coprimos), de esta forma, es posible que los números no sean iguales.

Luego, la relación entre las potencias originales ($P(p_i)$) y las nuevas potencias ($P'(p_i)$) se mantienen. Entonces, se usan las potencias $P'(p_i)$ para formar los grupos y repartir los

bloques: $P'(p_i)$ es la cantidad de bloques que recibe p_i de cada grupo. Esta potencia se corresponde directamente con la cantidad de trabajo que realizará el nodo.

De esta forma, se utilizan estas nuevas potencias para determinar cuántos bloques forman cada grupo y cuántos bloques de cada grupo recibe cada nodo. Con esto se logra una distribución cíclica de bloque en función de la potencia de cómputo de cada uno de los nodos.

Tratamiento de bloques restantes (grupo incompleto)

En el caso de que el tamaño de grupo no divida exactamente a la cantidad de bloques, queda un grupo incompleto (con menos bloques que los demás). Como se puede observar en la figura 4.13, si se realiza el mismo tratamiento de un grupo incompleto que el tratamiento realizado sobre grupos completos, se puede provocar un desbalance de carga importante. Esto hace que se ponga especial atención en cómo se distribuyen estos últimos bloques.

En el caso de que quede un grupo incompleto se calcula la cantidad de trabajo que realizará cada nodo de este último grupo, en función de su capacidad. En este caso, se conoce:

- Tamaño de grupo completo.
- Cantidad de trabajo que realiza p_i en un grupo completo ($P(p_i)$).
- Tamaño de grupo incompleto.

Utilizando estos valores, se puede determinar cuánto trabajo resolverá cada nodo de este último bloque incompleto. La ecuación 4.19 muestra esta relación.

$$\#\text{ bloques que debe resolver } p_i = \frac{\#\text{bloques restantes} * P'(p_i)}{\#\text{ bloques grupo completo}} \qquad (4.19)$$

Esta cantidad relaciona la cantidad de trabajo que realiza el nodo p_i en un grupo completo y la cantidad de trabajo total en un grupo incompleto.

Luego de repartir los bloques de este posible grupo incompleto utilizando la ecuación 4.19 para determinar cuántos bloques se le asignan a cada nodo, existe la posibilidad de que sigan quedando bloques sin asignar, en este caso, éstos se distribuyen de a uno comenzando desde el primer nodo (con máxima potencia de procesamiento). Esta cantidad restante es menor a la sumatoria de las nuevas potencias (por ser el resto).

Un ejemplo de esta distribución se puede observar en la figura 4.14, donde se divide una matriz en n bloques y los mismos se distribuyen en un cluster formado por 3 nodos (p_0, p_1 y p_2) con potencias 4810, 2420 y 1200 respectivamente (estas potencias son los Mflop/s de cada nodo).

Así, la potencia total de cluster es de 8430 Mflop/s. Considerando la potencia total y cada una de las potencias individualmente, se puede obtener qué porcentaje de potencia tiene que

recibir cada nodo con respecto a la potencia total (lo cual se puede ver como la cantidad de trabajo que debería resolver cada nodo con respecto al trabajo total):

p_0 = (4810 Mflop/s *100 %) / 8430 Mflop/s = 57% del trabajo total (aproximadamente)
p_1 = (2420 Mflop/s *100 %) / 8430 Mflop/s = 29% del trabajo total (aproximadamente)
p_2 = (1200 Mflop/s *100 %) / 8430 Mflop/s = 14% del trabajo total (aproximadamente)

Se aplican los pasos enunciados anteriormente: se dividen todas las potencias por 1200 (P_{min}) y se multiplican por 10 y sólo se considera la parte entera de este resultado:

$P'(p_0)$ = 4810/1200 *10 = 40
$P'(p_1)$ = 2420/1200 *10 = 20
$P'(p_2)$ = 1200/1200 *10 = 10

Luego, el máximo común denominador es 10, por lo que se dividen todas las potencias por este número:

$P'(p_0)$ = 4810/1200 *10 = 4
$P'(p_1)$ = 2420/1200 *10 = 2
$P'(p_2)$ = 1200/1200 *10 = 1

Luego, el tamaño de grupo es de 7 bloques cada uno. Entonces, se puede determinar qué porcentaje de trabajo realiza cada nodo considerando el tamaño del bloque (esto se multiplicaría por la cantidad de bloques y la relación se mantiene).

p_0= (4 bloques *100 %) / 7 bloques = 57% del trabajo total en un grupo (aproximadamente)
p_1= (2 bloques *100 %) / 7 bloques = 29% del trabajo total en un grupo (aproximadamente)
p_2= (1 bloque *100 %) / 7 bloques = 14% del trabajo total en un grupo (aproximadamente)

Luego, se puede ver que el porcentaje de trabajo que realiza cada nodo en cada uno de los grupos se corresponde con el porcentaje considerando las potencias originales.

Sea una matriz dividida en 20 bloques, la figura 4.14 muestra cómo quedan distribuidos los bloques completos considerando el cluster con la configuración anterior y las potencias nuevas (sólo se muestran distribuidos los bloques de los 2 grupos completos):

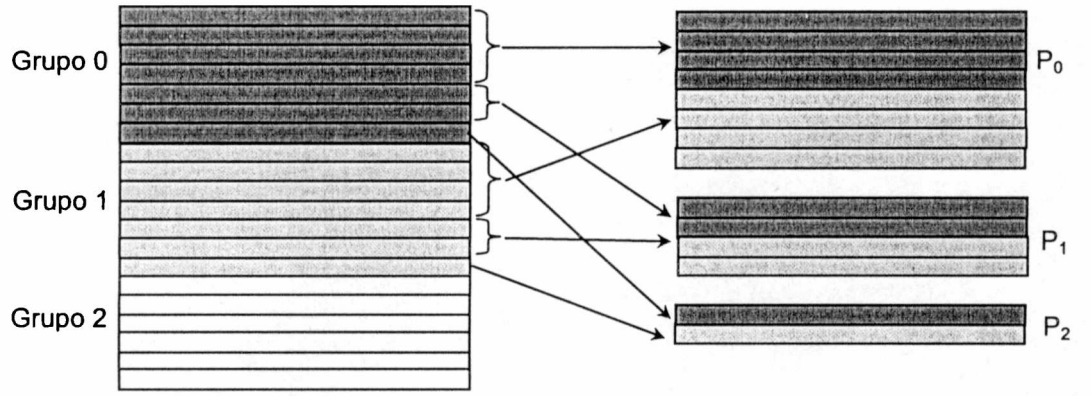

Figura 4.14: distribución cíclica por grupos calculando las potencias. Sólo se muestra la distribución de los bloques completos

En el ejemplo de la figura 4.14 se puede ver que los bloques se dividen en 2 grupos completos de 7 bloques quedando 1 bloque incompleto con 6 bloques. El nodo p_0 recibe 4 bloques de los primeros 2 grupos, p_1 recibe 2 y p_2 recibe 1. En la figura no se muestra la distribución del bloque incompleto. El tratamiento de este grupo se muestra en la figura 4.15.

Para no perder claridad, a los grupos completos se los llamará *gc* y al grupo incompleto *gi*. Se recalcula cuántos bloques debe recibir cada nodo del bloque incompleto:

p_0 = (6 bloques *gi* * 4 bloques *gc*) / 7 bloques *gc* = 3 bloques *gi*
p_1 = (6 bloques *gi* * 2 bloques *gc*) / 7 bloques *gc* = 1 bloque *gi*
p_2 = (6 bloque *gi* *1 bloque *gc*) / 7 bloques *gc* = 0 bloques *gi*

Con esto, se distribuyen los primeros 4 bloques del grupo incompleto, los 2 bloques restantes se asignan de a uno secuencialmente comenzando desde p_0.

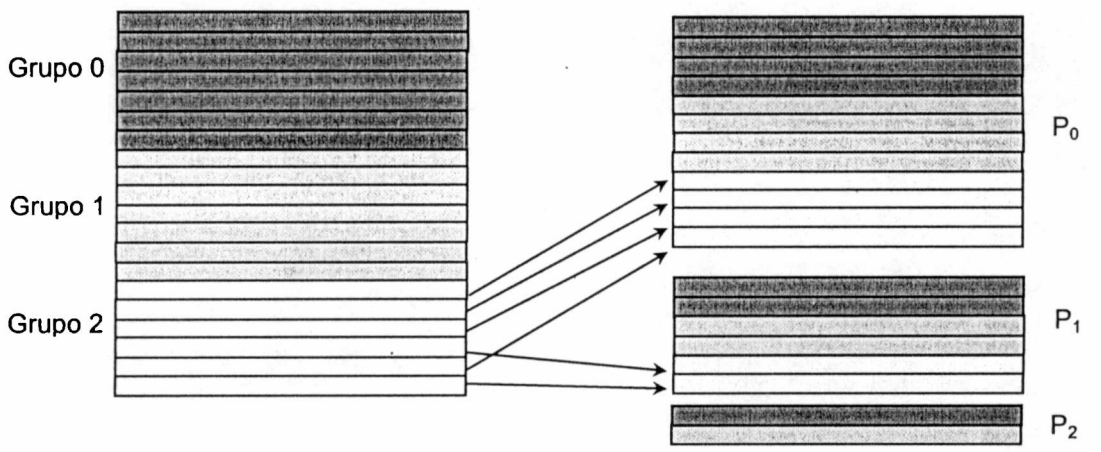

Figura 4.15: distribución de los bloques de un grupo incompleto

Luego, los porcentajes de trabajo que recibe cada nodo son:

p_0 = 60 % del trabajo total.
p_1 = 30 % del trabajo total.
p_2 = 10 % del trabajo total.

Como se puede ver, existe una pequeña diferencia con los porcentajes de trabajo que debería haber recibido cada nodo. Estas diferencias se deben principalmente a:

- Los cálculos realizados sobre las potencias originales para obtener grupos con pocos bloques.
- Los bloques se distribuyen como unidades enteras, los mismos no se dividen.

4.5.2.3 Método directo mejorado (cíclico)

En el método anterior, se utiliza la idea que se propone en [4], donde se dividen los bloques en grupos y los bloques de cada grupo se distribuyen entre todos los nodos. Sin embargo, en dicho trabajo no se especifica cómo distribuir los bloques de cada uno de los grupos. Como primera aproximación, los bloques de cada grupo se han repartido de forma secuencial.

Como ya se ha mencionado, los bloques se asignan cíclicamente entre los nodos para evitar que los nodos se queden sin bloques a medida que avanza la factorización, quedando nodos ociosos a medida que factoriza sus bloques. Se puede aplicar la misma idea dentro de cada uno de los grupos: si se reparten los bloques de un grupo de forma consecutiva a cada uno de los nodos, se puede llegar a producir este mismo desbalance de trabajo encontrado en el problema de la factorización entera pero dentro de cada grupo. O sea, la carga de trabajo en cada uno de los grupos puede no estar balanceada a pesar de que se asignen bloques en función de las capacidades de los nodos, debido al patrón de procesamiento de la factorización. Este desbalance, se repite grupo a grupo, por lo puede llegar a ser importante e influir en el rendimiento total. Luego, se propone una mejora al método anterior con el objetivo de evitar este desbalance dentro de cada grupo.

Se propone el siguiente ejemplo para mostrar este comportamiento: se dispone de un cluster formado por 3 nodos (p_0, p_1 y p_2), con potencias 3, 2 y 1 respectivamente. Además, la matriz a factorizar se divide en 12 bloques, obteniéndose la distribución como muestra la figura 4.16:

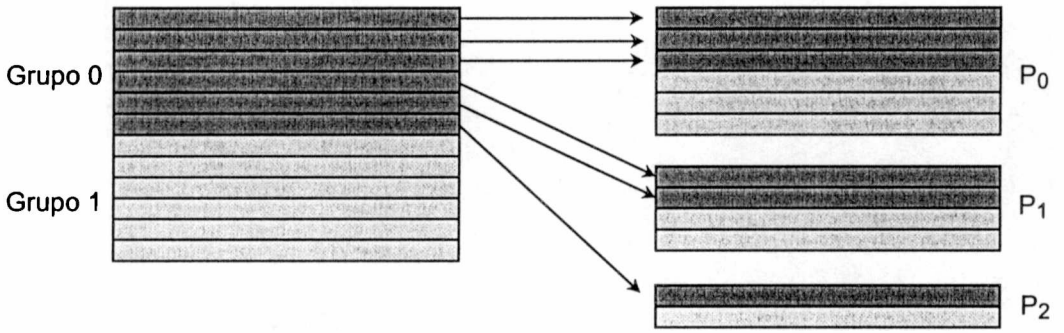

Figura 4.16: distribución cíclica de bloques divididos en grupos. Dentro de cada grupo, la distribución es secuencial.

En las figuras siguientes, se muestra cómo avanza la factorización y cómo se reduce la parte activa de la matriz en cada nodo. Vale la pena recordar que en cada iteración, un nodo factoriza uno de sus bloques (factorización LU) y luego, todos los nodos, actualizan la parte activa de la matriz (procesamiento de los pivotes, resolución de sistemas de ecuaciones triangulares y multiplicación de matrices). Por lo tanto, en todas las iteraciones todos los nodos realizan cálculos sobre los datos de los bloques activos de la matriz.

Vale la pena mencionar que en el ejemplo de las figuras se trabaja con una matriz con pocos bloques. Pero en la práctica las matrices son grandes y los grupos pueden tener muchos bloques. En el ejemplo, se divide la matriz en 12 bloques, por lo tanto, la factorización entera de la matriz se resuelve en 12 iteraciones.

La figura 4.17 muestra el comienzo de la factorización, donde toda la matriz está activa, por lo tanto, todos los nodos tienen todos sus bloques activos. En este punto, la cantidad de bloques activos en cada nodo (cantidad de trabajo a realizar) se corresponde con la potencia de cálculo de cada nodo.

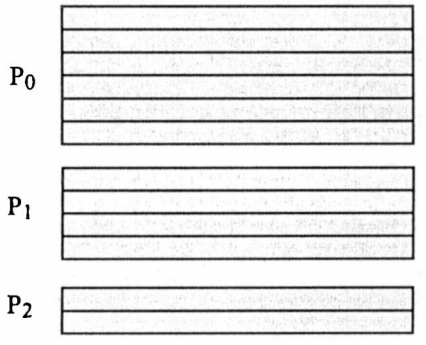

Figura 4.17. Comienzo de la factorización. Todos los bloques de la matriz están activos.

En las primeras 3 iteraciones, p_0 factoriza sus primeros 3 bloques y envía por cada uno el bloque factorizado y los pivotes. Esto se muestra en la figura 4.18. A partir de la iteración

3, p_0 (el nodo de máxima potencia) trabaja con ½ de sus bloques, y esto se va repitiendo en cada uno de los nodos a medida que avanza la factorización.

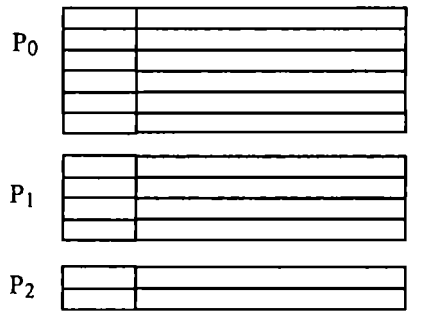

Figura 4.18: Después de las primeras 3 iteraciones, donde P_0 factorizó sus primeros 3 bloques

En las próximas 2 iteraciones, p_1 factoriza y envía sus primeros 2 bloques. A partir de esta iteración p_1 trabaja con ½ de sus bloques.

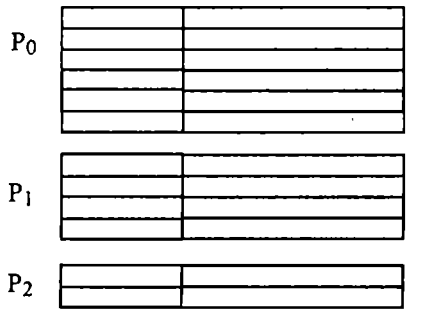

Figura 4.19: después de las primeras 5 iteraciones, donde p_0 factorizó sus primeros 3 bloques y p_1 sus primeros 2 bloques.

Las figuras muestran de forma parcial el avance de la factorización. Se muestra sombreada la parte activa de la matriz en cada uno de los nodos. En las figuras se puede apreciar cómo la parte activa de la matriz se reduce iteración a iteración y la parte inactiva avanza.

Si g es la cantidad de grupos en que se dividen los bloques, cada vez que un nodo factoriza todos los bloques de un grupo en particular, se queda con $1/g$ menos de sus bloques. En el caso del ejemplo, los bloques se dividen en 2 grupos, por lo tanto, cada vez que un nodo factoriza los bloques de un grupo, se queda con ½ de sus filas menos.

Esto produce que los nodos trabajen tiempos distintos unos a otros si tienen muchos bloques pertenecientes a un grupo. Para evitar esto, se propone una mejora al método. Este método es similar al anterior en cuanto a que se recalculan las potencias de la misma forma a como se explicó anteriormente, pero no se toman bloques consecutivos dentro de cada grupo, sino que se asignan cíclicamente entre los nodos.

Con el mismo ejemplo de la Figura 4.16, utilizando 3 nodos de potencia 3, 2 y 1, se reparten los 12 bloques en que se divide la matriz original

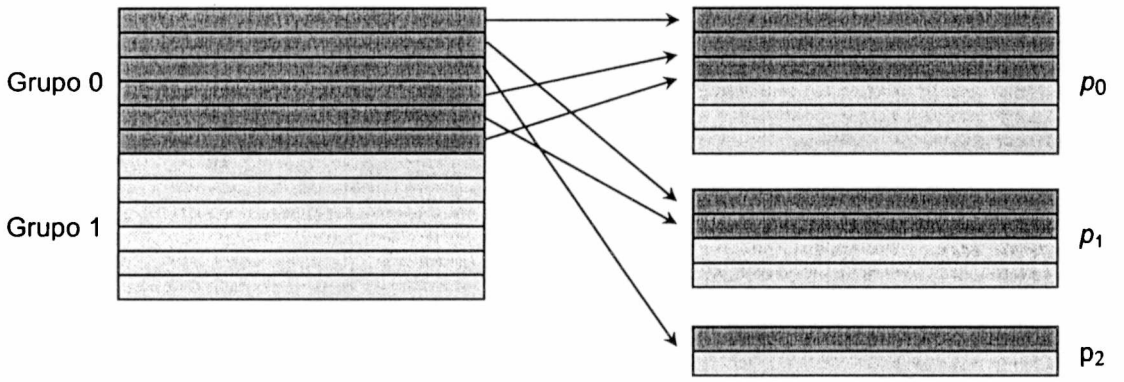

Figura 4.20: Distribución cíclica de bloques dentro de cada grupo en un cluster heterogéneo.

Se muestran sólo las flechas del primer grupo para no perder claridad. En la figura se observa que los bloques dentro de cada grupo se asignan cíclicamente entre todos los nodos, con esto se evita el desbalance producido por el progreso de la factorización.
A continuación, se muestra el avance de la factorización de la misma forma a como se hizo antes, pero asumiendo la distribución cíclica de los bloques.

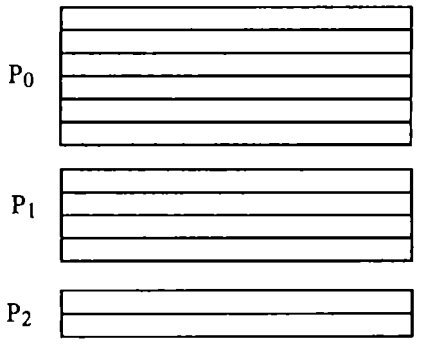

Figura 4.21: Comienzo de la factorización. Todos los bloques de la matriz están activos.

En las primeras 3 iteraciones, cada uno de los nodos factoriza 1 de sus bloques, entonces, cada nodo trabaja con un bloque menos, manteniéndose el porcentaje de trabajo que cada uno debe realizar. De esta forma, no sucede que p_0 continúe trabajando con sólo ½ de sus bloques. Este paso se muestra en la figura 4.22.

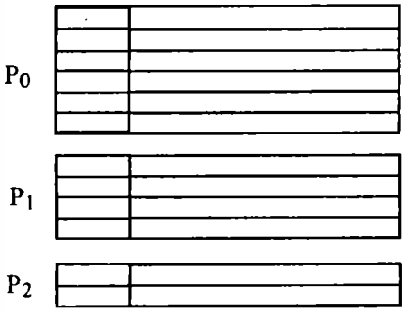

Figura 4.22: Después de 3 iteraciones, donde p_0, p_1 y p_2 factorizan cada uno su primer bloque.

Luego de las primeras 6 iteraciones, cada uno de los nodos factoriza 2 de sus bloques, como se muestra en la figura 4.23. De esta forma se puede ver que cada P iteraciones los nodos reducen uno de sus bloques activos, manteniéndose de esta forma el balance de carga en la mayor parte de toda la factorización.

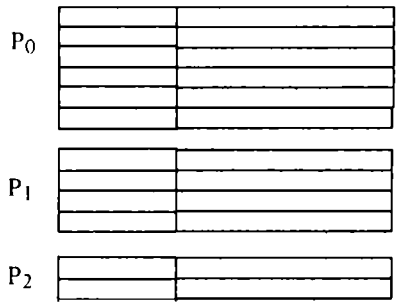

Figura 4.23: Avance de la factorización después de 6 iteraciones. Los nodos reducen sus bloques activos de forma pareja entre todos los nodos.

De esta forma, se puede ver que a medida que la factorización avanza, todos los nodos reducen sus bloques activos de forma pareja. Esto evita que haya desbalance de carga aún entre nodos que tienen la misma capacidad y reciben la misma cantidad de bloques.

4.5.3 Resumen de los métodos de distribución

Los distintos métodos de distribución de carga explicados en este capítulo fueron:

- Método homogéneo: este método está orientado a clusters homogéneos. Se dividen los bloques en grupos de tamaño igual a la cantidad de nodos existentes en el cluster. Cada nodo recibe 1 bloque de cada grupo. De esta forma, todos los nodos recibirán la misma cantidad de bloques y se obtendrá balance de carga (si la cantidad de nodos no divide a la cantidad de bloques habrá un desbalance de a lo sumo 1 bloque). Este método es explicado en este capítulo porque introduce la idea de dividir a los bloques en grupos.

- Métodos heterogéneos

  - Método "directo": se tienen en cuenta los cientos de Mflop/s de cada nodo y esto es tomado como potencia ($P(p_i)$). Se dividen los bloques en grupos de tamaño igual a la sumatoria de todas las potencias. Luego, el nodo p_i recibe $P(p_i)$ bloques consecutivos de cada grupo. Este método es muy simple de entender e implementar. Pero el balance de carga obtenido depende de: tamaño de los grupos, cantidad de bloques consecutivos que recibe cada nodo de cada grupo, cantidad de bloques que

quedan en un posible grupo incompleto, etc. Esto hace que normalmente no se obtenga balance de carga adecuado utilizando este método.

- Método directo mejorado (secuencial): para evitar los problemas del método anterior, este método realiza determinados cálculos sobre las potencias para generar grupos más chicos y hacer que cada nodo reciba menos bloques de cada uno de los grupos. Los cálculos realizados sobre las potencias son tales que tratan de no modificar la relación que tiene la potencia de cada nodo con la potencia total del cluster. Además, si queda algún grupo incompleto, se realizan nuevos cálculos para determinar cómo se reparten sus bloques. Dentro de cada grupo, los bloques se asignan secuencialmente a los nodos.

- Método directo mejorado (cíclico): este método es igual al anterior pero reparte de forma cíclica los nodos pertenecientes a un grupo y no de forma secuencial como lo hacía el último método. Con esta modificación se logra que el trabajo que realiza cada nodo dentro de un grupo sea más balanceado.

5. Capítulo 5: Experimentación

En el capítulo anterior se describe la solución propuesta a la factorización LU. Esta solución considera las características propias de la arquitectura para la cual se desarrolla. Esta arquitectura son los clusters heterogéneos y las características más importantes (que determinan algunas de las características del algoritmo) fueron la red de interconexión y la heterogeneidad de los nodos. Estas características implicaron que se preste especial atención a la cantidad de datos comunicados, forma en que se comunican, y a la forma en que se distribuyen los datos entre los nodos que participan en la resolución de la factorización.

Como consecuencia de la heterogeneidad de cómputo y con el objetivo de obtener buen balance de carga a lo largo del trabajo se han propuesto e implementado distintos métodos de distribución de datos. Pero, ¿por qué queremos obtener buen balance de carga? La respuesta es que obtener buen balance de carga implica mejorar el rendimiento total del método.

A lo largo del capítulo anterior se han explicado los distintos métodos de distribución de datos propuestos. Se mencionaron las características generales que comparten todos los métodos y las específicas de cada uno de los mismos.

En este capítulo se describirán las distintas experimentaciones que se han realizado. De cada una de las experimentaciones se describirá la arquitectura utilizada y se mostrarán los tiempos obtenidos. Además, de los tiempos obtenidos se realizará un breve análisis, considerando aspectos positivos y negativos del método analizado. Los aspectos negativos de cada método son los que sirven para proponer mejoras sobre el mismo.

5.1 Características de las pruebas realizadas

La experimentación se ha basado en ejecutar la solución a la factorización LU propuesta utilizando los distintos métodos de distribución de carga que fueron propuestos. Estas pruebas han permitido determinar problemas de cada uno de los métodos de distribución y proponer posibles soluciones a dichos problemas.

A continuación se darán detalles de ciertas características de las pruebas realizadas. Estas características sirven para entender que es lo que se ha evaluado y analizado en cada una de las experimentaciones realizadas.

5.1.1 Tiempos analizados

Si el método de distribución de datos logra que todos los nodos pasen tiempos similares en el procesamiento de los datos, entonces ese método ha logrado balance de carga. Así, es necesario determinar el tiempo que pasa cada uno de los nodos realizando cómputos sobre los datos que le han sido asignados.

Para determinar estos tiempos, se instrumenta el código de tal forma que se obtiene:

- Tiempo total: tiempo en realizar todas las iteraciones necesarias para realizar la factorización completa de la matriz.
- Tiempo de comunicación: se toma el tiempo de envío y recepción de datos.

Entonces, al tiempo total se le resta el tiempo de comunicación y este tiempo es el que se asume que el nodo está específicamente realizando procesamiento sobre los datos. En el tiempo de cómputo se está incluyendo: tiempo de factorización y actualización de los bloques activos y procesamiento de pivotes. Por otro lado, en estos tiempos no se incluyen tiempos de inicialización, distribución de carga, comunicación ni sincronización.

5.1.2 Configuración del cluster

En cada una de las experimentaciones se describirá la configuración del cluster utilizado y se mostrarán las características más relevantes de cada uno de los nodos que lo componen. En los casos en que exista más de un nodo con las mismas características, se mostrarán en forma conjunta para evitar repetir información. En estos casos, todos los nombres de los nodos se generalizarán con un único nombre representativo.

Las características que se especificarán de cada nodo (o tipo de nodo) son las que definen la capacidad de procesamiento del nodo. Estas características son:

- Tipo de nodo.
- Frecuencia del reloj (en MHz).
- Cantidad de memoria RAM (en MB).
- Cantidad de millones de operaciones en punto flotante por segundo (en Mflop/s).

5.1.3 Tamaño de matriz

En las pruebas realizadas se utilizan matrices cuadradas de elementos de tipo flotante de precisión simple. No se utilizan elementos de tipo flotante de precisión doble ya que las capacidades de cómputo de los nodos utilizados casi siempre son similares para cualquiera de las dos precisiones y además, los elementos de precisión doble ocupan más lugar para su representación, por lo que las matrices ocuparían más lugar y como consecuencia, se debería trabajar con matrices de menor tamaño [19].

Para este trabajo es conveniente utilizar matrices lo suficientemente grandes de forma tal que representen los problemas reales para los cuales el cómputo paralelo tiene sentido. Pero, por otro lado, los datos deben entrar en memoria principal. Si los datos no entran en memoria principal, se utiliza memoria virtual (memoria swap) para almacenar los datos. Cuando un dato requerido no se encuentra en memoria principal, se debe acceder a memoria virtual para obtenerlo. El acceso a la memoria virtual es varios órdenes de magnitud más lenta que el acceso a memoria principal. El tiempo de obtención de un dato está incluido en el tiempo de cómputo. Entonces, el tiempo de acceso a memoria secundaria estaría siendo evaluado como tiempo de cómputo, y al ser este acceso mucho más lento, penalizaría el tiempo de cómputo del o de los nodos que utilicen memoria virtual.

Entonces, los tamaños de las matrices elegidos en cada caso son tamaños de matrices suficientemente grandes pero tales que los datos asignados a cada uno de los nodos entren en memoria principal, evitando así el uso de memoria virtual.

Además, los distintos tamaños de matriz para los cuales se mostrarán resultados son tales que intentan mostrar algún tipo de característica de las distribuciones obtenidas. Se intenta obtener un conjunto de pruebas que muestren tanto problemas como también comportamientos ante los mismos. Así, se incluyen pruebas que aportan información útil sobre las distribuciones y el balance de carga obtenido y también se descartan pruebas redundantes o que no aportan información nueva.

5.1.4 Tamaño de bloque

Como ya se mencionó en este trabajo, las matrices se dividen en bloques de filas enteras. El tamaño de bloque especifica cuántas filas tiene cada bloque. Esto determina cuántas filas se factorizan en cada uno de los pasos de la factorización y la cantidad de datos que se comunican en cada iteración.

Como se especifica en [4] el tamaño de bloque debe ser elegido de forma tal que:

- Maximice la capacidad de procesamiento de cada nodo.
- Permitir que la carga de trabajo sea balanceada.

Aunque tal vez estos conceptos sean contradictorios (pues lograr uno tal vez implique penalizar el otro), es necesario llegar a un punto medio entre los mismos, de tal manera que se den de forma correcta (o al menos lo más correcta posible) ambas propiedades.

En [23] se puede observar que los tamaños de bloques comúnmente utilizados en experimentaciones de este tipo varían desde 1 a 256 filas por bloque. En dicho trabajo, se muestra que para este contexto específico los mejores tamaños de bloques para la factorización LU en clusters heterogéneos son 32 y 64, aunque el óptimo es 64. Utilizando la experiencia obtenida en dicho trabajo se utilizarán 64 filas por bloque para la experimentación.

5.2 Benchmarks

Como se mencionó anteriormente, en los métodos de distribución propuestos se tiene en cuenta la capacidad de cómputo de cada uno de los nodos que participan en la factorización. La capacidad de cómputo se utiliza para determinar la cantidad de trabajo que se asignará a cada uno de los ellos.

En este trabajo se utilizará directamente los Mflop/s locales de cada nodo. El algoritmo utilizado para determinar esta cantidad es lo que se conoce como benchmark. Es importante encontrar un benchmark que caracterice de la forma más correcta posible el rendimiento de cada nodo.

Aunque es posible encontrar el rendimiento máximo teórico de una computadora (obtenido analizando directamente el hardware de la misma), este rendimiento no se suele tener en cuenta ya que es difícil alcanzarlo con aplicaciones reales. Es por esto que en vez de utilizar el rendimiento máximo teórico, existen múltiples benchmarks los cuales se utilizan para obtener el rendimiento de distintas aplicaciones de usuario [19].

En [19] se propone utilizar el mismo problema a resolver pero en dimensiones menores de forma tal que sea posible resolverlo en una sola computadora. Es fácil esperar que el mismo problema a resolver caracterice de forma correcta el rendimiento local en una sola computadora. De esta forma, se utilizará en este trabajo la misma factorización LU para caracterizar el rendimiento local en cada uno de los nodos.

En este trabajo, como primer benchmark se ha utilizado la factorización secuencial (sin dividir la matriz en bloques) en cada uno de los nodos. Este benchmark dio resultados que se utilizaron para determinar la capacidad de cada nodo en las distintas pruebas realizadas. Pero al ejecutar el problema en mayores dimensiones y en forma paralela, se pudo observar que a pesar de que el trabajo asignado a cada nodo correspondía con la capacidad de ese nodo, los tiempos de trabajo obtenidos eran distintos a los esperados (esto se muestra en el conjunto de prueba 1 en el punto 5.4.2.1).

Este comportamiento ha hecho reevaluar el benchmark utilizado y se optó por utilizar la factorización LU secuencial por bloques en cada nodo. El rendimiento obtenido utilizando

la factorización secuencial por bloques dio como resultado una mejor caracterización del rendimiento local de cada nodo (conjunto de pruebas 2, punto 5.4.2.2).

5.3 Rutina de comunicación

Como se mencionó anteriormente, algunas características del algoritmo propuesto han sido definidas por la arquitectura utilizada. Una de estas características es el tipo de comunicaciones: todas las comunicaciones del algoritmo son de tipo broadcast. Entonces, es necesario utilizar una rutina broadcast que tenga buen rendimiento en este tipo de red de interconexión para que el rendimiento total del algoritmo no se vea penalizado por el rendimiento de las comunicaciones.

Actualmente existen múltiples librerías de comunicación que proveen rutinas para el envío y la recepción de mensajes. Algunas de estas librerías se encuentran gratuitas en Internet, como PVM o implementaciones de MPI. Tanto PVM como las implementaciones de MPI existentes son librerías de propósito general, o sea que no han sido definidas para un tipo de arquitectura en particular [19].

Específicamente hablando de las redes Ethernet, se debe tener en cuenta que implementa todos los mensajes como mensajes broadcast (definido por el protocolo 802.3), donde se envía un mensaje al canal compartido y el mismo llega a todos los nodos conectados al mismo.

De esta forma, conviene utilizar una rutina de comunicación colectiva que aproveche el broadcast a nivel físico que tiene una red Ethernet. Esto no se da en la librería PVM, ya que implementa el broadcast como múltiples mensajes punto a punto, haciendo que el rendimiento sea dependiente de la cantidad de nodos que participan en la comunicación. Por otro lado, como ya se mencionó anteriormente, las implementaciones de MPI son orientadas a máquinas paralelas de pasaje de mensajes en general, o sea que no tienen en cuenta las características de ningún tipo de red de interconexión específica (tampoco las características de una red Ethernet).

Por estas razones en este trabajo no se utiliza PVM ni ninguna implementación existente de MPI, sino que se utiliza una rutina broadcast especialmente desarrollada para ser utilizada en redes Ethernet. Esta rutina aprovecha todas las características definidas por este estándar. Se basa en utilizar el protocolo UDP (User Datagram Protocol), sobre IP (Internet Protocol) en donde se envían paquetes al canal y todos los nodos conectados los reciben. Entonces, esta rutina aprovecha el broadcast que se tiene a nivel físico para realizar las comunicaciones de usuario [21] [24].

Por otro lado, la librería incluye operaciones que pueden solapar y realizar cómputo y comunicación a la vez. De todas formas, la capacidad de realizar cómputo y comunicación de forma solapada depende de la capacidad que tengan de realizarlo cada uno de los nodos.

5.4 Experimentación

El algoritmo se ha implementado utilizando el lenguaje de programación C. Para la creación y manejo de tareas se han utilizado rutinas de PVM.

5.4.1 Cluster Homogéneo

En el capítulo anterior, se explicó un método de distribución para clusters homogéneos. A pesar de que el contexto de trabajo es distinto (se trabaja con clusters heterogéneos), este método se incluyó en el trabajo porque introduce la idea de dividir los bloques en grupos y distribuir los bloques de cada grupo entre todos los nodos.

Este método no ha sido evaluado en la experimentación, ya que es fácil ver que este método no obtendrá buen balance de carga en un cluster heterogéneo, ya que asume que todos los nodos tienen la misma capacidad de cómputo y se distribuyen partes iguales de trabajo a todos los nodos. Esto hará que todo el tiempo de la factorización dependa del nodo más lento, del que más tarde en resolver su parte del problema. Por este motivo, se experimenta directamente con los métodos que sí tienen en cuenta la heterogeneidad del cluster y que distribuyen el trabajo en función de las distintas capacidades.

5.4.2 Clusters Heterogéneos

Se han realizado distintas pruebas con clusters heterogéneos con el objetivo de evaluar los distintos métodos de distribución de carga. Para cada uno de estos métodos, se han realizado pruebas con distintos tamaños de matrices. Además, se varía la configuración del cluster para evaluar el algoritmo en ambientes que aporten resultados útiles para analizar.

En todos los casos, los clusters están interconectados por una red Ethernet de 100 Mb/s. Además se usa un switch de 100 Mb/s con capacidad de switching completo.

Para cada prueba realizada, se mostrará una tabla con la siguiente información referente a cada uno de los nodos que participan en la factorización:

- **Filas asignadas**: cantidad de filas asignadas al nodo (determinado por la cantidad de bloques).
- **Trabajo ideal**: cantidad de trabajo que debería recibir en función de su capacidad de procesamiento con relación a la capacidad total del cluster.
- **Trabajo real**: cantidad de trabajo realmente asignado por el método de distribución.
- **Carga**: porcentaje de cuánto realmente trabajará cuando debería estar trabajando a un 100% (100% es cuando trabaja exactamente de acuerdo a su capacidad).

Además, para cada tamaño de matriz se graficarán los tiempos de cómputo de cada uno de los nodos. Este tiempo corresponde al tiempo de cómputo descrito en el punto 5.1.1. Para cada conjunto de prueba, el análisis de los resultados obtenidos se hará de forma conjunta. En el caso que exista alguna característica particular que no se observa para los otros tamaños de matriz o que valga la pena resaltar, se analizará puntualmente esa prueba de forma individual.

Con el objetivo de ganar claridad en los gráficos no se especifica el nombre completo de cada uno de los nodos, sino que se opta por nombrar con un número que lo distingue de los demás. Este número es un número que forma parte de su "nombre". Por ejemplo, existen nodos llamados "lidipar65", "lidipar66" hasta "lidipar80". En este caso, se utilizan los números 65, 66 al 80 para nombrar a cada uno de estos nodos. En el caso de las computadoras Pentium 4, son llamadas WatsonX, donde X son números pero en este caso no son consecutivos. De todas formas, como son números que no se repiten y que no están en el rango de 65 a 80, se usa el mismo criterio: sólo se ponen los números que forman parte de su nombre. Esto queda además definido por las tablas correspondientes a las configuraciones de los clusters.

En cada conjunto de prueba se incluirán 4 pruebas donde varía el tamaño de la matriz factorizada. Estos tamaños fueron elegidos de tal forma que los resultados obtenidos permiten relacionar: tamaño de matriz, cantidad de grupos, tamaño de los grupos, tamaño del grupo incompleto, etc.

5.4.2.1 Conjunto de Prueba 1

En este conjunto de prueba se utiliza el método de distribución directo mejorado secuencial. El benchmark utilizado es la factorización LU secuencial sin procesamiento por bloques. La configuración del cluster utilizado es como se muestra en la Tabla 5.1.

Cantidad	Nombre	Tipo CPU	Frec. Reloj	Memoria RAM	Mflops LU
8	Lidipar65..72	Pentium III	700 MHz	64 MB	1561
8	Lidipar73..80	AMD-Duron	850 MHz	256 MB	1894

Tabla 5.1: configuración del cluster del conjunto de prueba 1

En base a estas potencias, se determina que:

- ✗ Cada nodo Pentium III debe realizar un 5.64% del trabajo total.
- ✗ Cada nodo AMD-Duron debe realizar un 6.85% del trabajo total.

Los tamaños de matrices utilizados son:

1) 14336 x 14336 flotantes.
2) 13312 x 13312 flotantes.
3) 12288 x 12288 flotantes.
4) 11264 x 11264 flotantes.

1) Matriz de 14336 x 14336 flotantes

Nodo	Filas asignadas	Trabajo ideal	Trabajo real	Carga
Pentium III (65..72)	768	5.64%	5.35%	94.85%
Durón (73..80)	1024	6.85%	7.14%	104.23%

Tabla 5.2: distribución obtenida en el conjunto de prueba 1, matriz de 14336 x 14336 flotantes

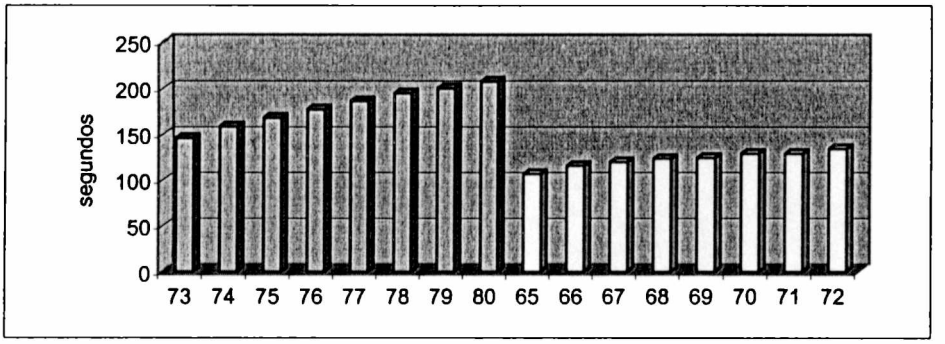

Figura 5.1: resultados obtenidos en el conjunto de prueba 1, matriz de 14336 x 14336 flotantes

2) Matriz de 13312 x 13312 flotantes

	Filas asignadas	Trabajo ideal	Trabajo real	Carga
Pentium III (65..72)	704	5.64%	5.28%	93.61%
Durón (73..80)	960	6.85%	7.21%	105.25%

Tabla 5.3: distribución obtenida en el conjunto de prueba 1, matriz de 13312 x 13312 flotantes

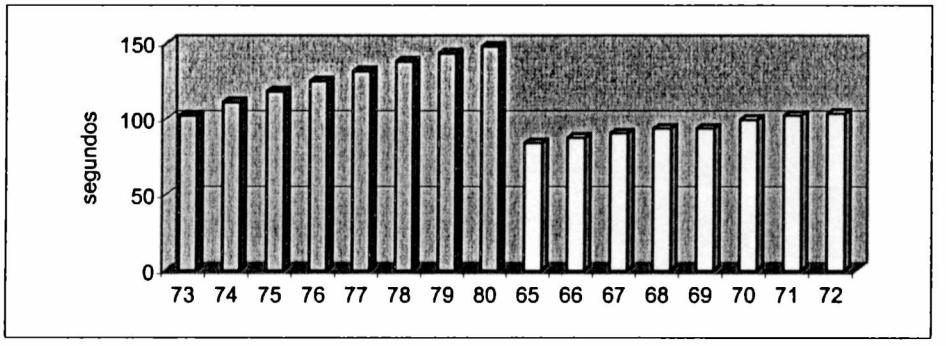

Figura 5.2: resultados obtenidos en el conjunto de prueba 1, matriz de 13312 x 13312 flotantes

3) Matriz de 12288 x 12288

	Filas asignadas	Trabajo ideal	Trabajo real	Carga
Pentium III (65..72)	640	5.64%	5.20%	92.19%
Durón (73..80)	896	6.85%	7.29%	106.42%

Tabla 5.4: distribución obtenida en el conjunto de prueba 1, matriz de 12288 x 12288 flotantes

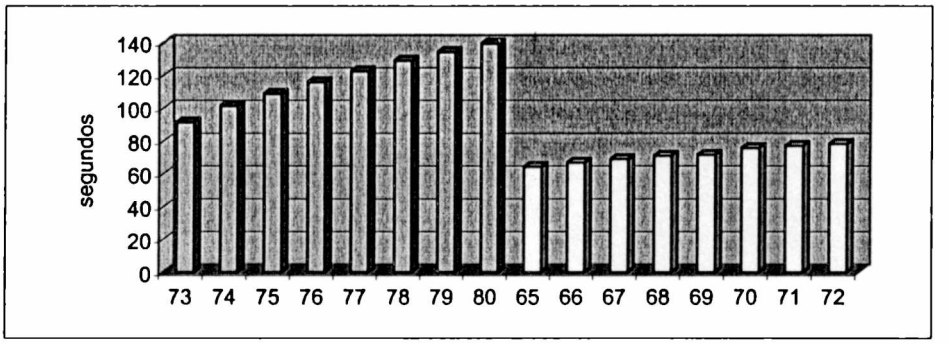

Figura 5.3: resultados obtenidos en el conjunto de prueba 1, matriz de 12288 x 12288 flotantes

4) Matriz de 11264 x 11264 flotantes

	Filas asignadas	Trabajo ideal	Trabajo real	Carga
Pentium III (65..72)	640	5.64%	5.68%	100.7%
Durón (73..80)	768	6.85%	6.81%	99.41%

Tabla 5.5: distribución obtenida en el conjunto de prueba 1, matriz de 11264 x 11264 flotantes

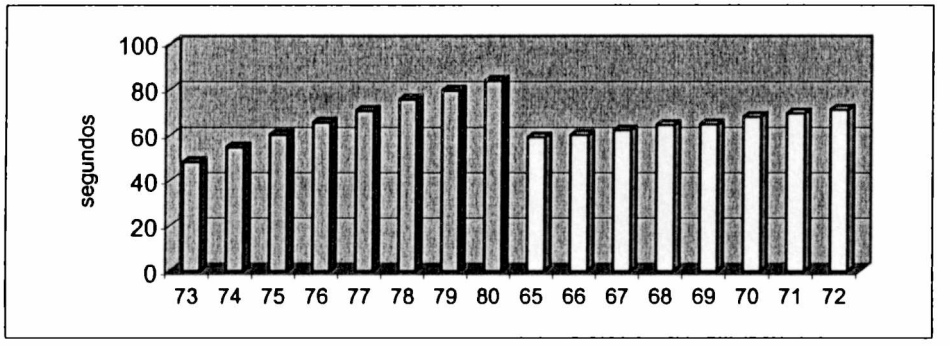

Figura 5.4: resultados obtenidos en el conjunto de prueba 1, matriz de 11264 x 11264 flotantes

Los tiempos obtenidos permiten observar:

1. El porcentaje de trabajo que recibe cada nodo es cercano al ideal pero no es el ideal que le correspondería por su capacidad de procesamiento (diferencia entre las columnas "Trabajo ideal" y "Trabajo real" de cada tabla de distribución). Esto es debido a que se reparten un número entero de bloques (64 filas cada uno). Esto genera un pequeño desbalance generado por la misma distribución de los bloques.

2. A pesar de que todos los nodos AMD-Duron reciben la misma cantidad de bloques para procesar y tienen la misma potencia, se observa un comportamiento "en escalera" en los tiempos de cómputo de estas máquinas. Este comportamiento también se observa en las Pentium III pero de una forma menor.

3. En todas las pruebas se observa una gran diferencia entre los tiempos de cómputo de los nodos AMD-Duron y los nodos Pentium III. Esta diferencia es mayor a la que se esperaba debido al desbalance de carga producido por la distribución propia de los bloques, explicado en el punto 1. Este desbalance de carga no debería existir, pues los datos se han distribuido en función de la capacidad de cómputo de cada nodo. De esta forma se esperaba que los tiempos de cómputo sean similares entre todos los nodos, evitando de esta forma que el tiempo de procesamiento del nodo más lento penalice el tiempo total de la factorización.

Como posible explicación de este último comportamiento observado (punto 3), se puede considerar que las potencias de los nodos no son correctas: el cálculo de la potencia de los nodos no es el correcto para este tipo de procesamiento. Para determinar la potencia de los nodos se ha utilizado como benchmark la factorización LU secuencial. La diferencia entre el código secuencial (benchmark) y el código en la solución paralela, es que este segundo resuelve la factorización por bloques, en tanto que el código secuencial utilizado como benchmark no lo hace de la misma manera, pues resuelve la factorización sin dividir la matriz en bloques.

A partir del análisis de los resultados del primer conjunto de prueba, se proponen distintas soluciones a los problemas encontrados. Estas posibles soluciones afectan el método de distribución utilizado. La primera modificación intenta solucionar el problema enunciado en el punto 3. La gran diferencia en los tiempos de cómputo entre un tipo de nodo y otro hacen evaluar otra forma de caracterizar las potencias de los nodos. Se utiliza como segundo benchmark la factorización LU secuencial por bloques en cada uno de los nodos que participan (como se había mencionado en el punto 5.2).

Las pruebas utilizando la factorización LU secuencial por bloques en cada uno de los nodos utilizados en las pruebas anteriores dan como resultado:

1) Mflops factorización LU por bloques de nodos tipo Pentium III: 1418
2) Mflops factorización LU por bloques de nodos tipo AMD-Duron: 1557

Manteniendo el mismo formato que se usó para describir la configuración del cluster anterior:

Cantidad	Nombre	CPU	Frec. Reloj	Memoria	Mflops LU por bloques
8	Lidipar65..72	Pentium III	700 MHz	64 MB	1418
8	Lidipar73..80	AMD-Duron	850 MHz	256 MB	1557

Tabla 5.6: configuración del cluster del conjunto de prueba 1 utilizando la factorización LU por bloques como benchmark

Se puede observar que para ambos tipos de nodos los tiempos crecieron (o sea, tenemos menos Mflops). Pero los nodos AMD-Duron lo hicieron de forma más significativa, y la cantidad de Mflops para esta operación por bloques es similar entre los dos tipos de nodos. Esto explica parte del comportamiento observado en el punto 3: la diferencia de capacidad entre los distintos tipos de nodos era menor, por lo tanto se estaba sobrecargando a los nodos AMD-Duron y dando menos trabajo a los nodos Pentium III.

Se puede ver que el procesamiento por bloques afecta de una forma mucho más significativa a los nodos AMD-Duron que a los Pentium III.

Con las nuevas potencias (calculadas con el segundo benchmark), se puede observar que las capacidades son parecidas, y por los cálculos que realiza el método de distribución (enunciados en el capítulo 4 punto 4.5.2.2), las nuevas potencias calculadas son iguales para los dos tipos de nodos. De forma abreviada, se muestran los cálculos que se realizan sobre las potencias (se muestra solamente los cálculos sobre 2 potencias ya que son iguales para cada tipo de nodo):

Pasos 1 y 2 (división de las potencias por la menor potencia y multiplicación por 10) :

1557 / 1418 = 1.09 * 10 = 10
1418 / 1418 = 1 * 10 = 10

luego, el máximo común divisor es 10 y las nuevas potencias son 1 para ambos nodos (pasos 3 y 4). De esta forma, se repartirán la misma cantidad de bloques para todos los nodos que componen el cluster.

Este caso no es útil para evaluar el método propuesto de distribución de carga ya que se tiene un cluster "no heterogéneo" para esta operación. Entonces, se va a utilizar otro conjunto de nodos para mantener heterogeneidad en el cluster. Se utilizan nuevos nodos y el benchmark utilizado es la factorización LU secuencial por bloques.

5.4.2.2 Conjunto de prueba 2

En este conjunto de prueba se utiliza el método de distribución directo mejorado secuencial. El benchmark utilizado es la factorización LU secuencial por bloques. La configuración del cluster utilizado es como se muestra en la Tabla 5.7.

Cantidad	Nombre	CPU	Frec. Reloj	Memoria	Mflops LU por bloques
8	WatsonX	Pentium 4	2392.099 MHz	1GB	5994
8	Lidipar65..72	Pentium III	700 MHz	64 MB	1418

Tabla 5.7: configuración del cluster del conjunto de prueba 2

En base a estas potencias, se determina que:

- Cada nodo Pentium 4 debe realizar un 10.10% del trabajo total.
- Cada nodo Pentium III debe realizar un 2.39% del trabajo total.

Los tamaños de matrices máximos para los que no se utiliza memoria swap, para este cluster son:

1) 20160 x 20160 flotantes
2) 19136 x 19136 flotantes
3) 18112 x 18112 flotantes
4) 15360 x 15360 flotantes

Para este conjunto de prueba, es interesante realizar observaciones para algunos tamaños de matriz en particular. Para los mismos se analizan inmediatamente después de mostrar los resultados. Al final de todas las pruebas se realiza un análisis en conjunto.

1) Matriz de 20160 x 20160 flotantes

	Filas asignadas	Trabajo ideal	Trabajo real	Carga
WatsonX	2048	10.10%	10.15%	100.49%
PentiumIII 65, 66 y 67	512	2.39%	2.53%	105.85%
Pentium 68..72	448	2.39%	2.22%	92.88%

Tabla 5.8: distribución obtenida en el conjunto de prueba 2, matriz de 20160 x 20160 flotantes

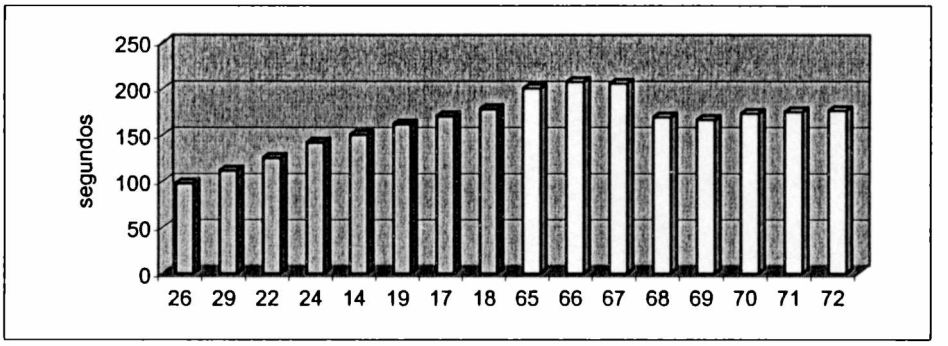

Figura 5.5: resultados obtenidos en el conjunto de prueba 2, matriz de 20160 x 20160 flotantes

En esta prueba se observa que los primeros 3 nodos Pentium III reciben un bloque más que el resto de las Pentium III y reciben más trabajo del que realmente deberían recibir en base a su potencia.

Este caso es interesante porque quedan bloques restantes que se distribuyen entre los primeros 11 nodos, entre los cuales hay 3 Pentium III (con menos capacidad). Estos nodos son los que determinan el tiempo total de la factorización. Esta diferencia se nota con 1 bloque (diferencia mínima), entonces, si la diferencia hubiese sido de más bloques, el tiempo se hubiese visto penalizado de una forma aún más importante por estos últimos bloques. Esto muestra que es muy importante distribuir de forma correcta los bloques pertenecientes a todos los grupos, incluyendo un último posible grupo incompleto. Esto justifica el tratamiento especial que se propuso para el caso en que el tamaño de bloque no divida exactamente a la cantidad de bloques y quede un último bloque incompleto.

2) Matriz de 19136 x 19136 flotantes

	Filas asignadas	Trabajo ideal	Trabajo real	Carga
Watson 26, 29 y 22	1984	10.10%	10.36%	102.57%
Watson X	1920	10.10%	10.03%	99.3%
Pentium 65..72	448	2.39%	2.34%	97.9%

Tabla 5.9: distribución obtenida en el conjunto de prueba 2, matriz de 19136 x 19136 flotantes

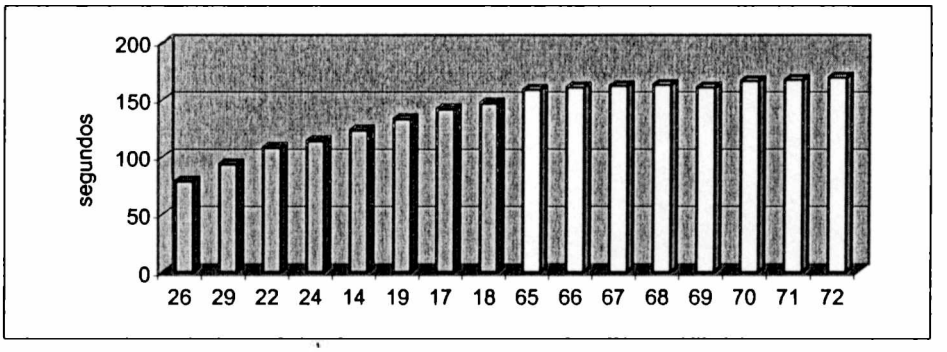

Figura 5.6: resultados obtenidos en el conjunto de prueba 2, matriz de 19136 x 19136 flotantes

En este caso, se observa que la carga está bien balanceada, los primeros 3 nodos reciben un bloque más que el resto de los nodos de su mismo tipo. Pero en estos primeros nodos, no influye de la forma significativa a como lo hacía en los nodos Pentium III. En este caso se puede ver que es importante ordenar los nodos de mayor a menor potencia para que en estos casos los bloques restantes se comiencen a asignar desde el nodo con mayor potencia.

3) Matriz de 18112 x 18112 flotantes

	Filas asignadas	Trabajo ideal	Trabajo real	Carga
Watson 26, 29 y 22	1856	10.10%	10.24%	101.38%
Pentium III 65..67	448	2.39%	2.47%	103.34%
Pentium III 68..72	384	2.39%	2.12%	88.7%

Tabla 5.10: distribución obtenida en el conjunto de prueba 2, matriz de 18112 x 18112 flotantes

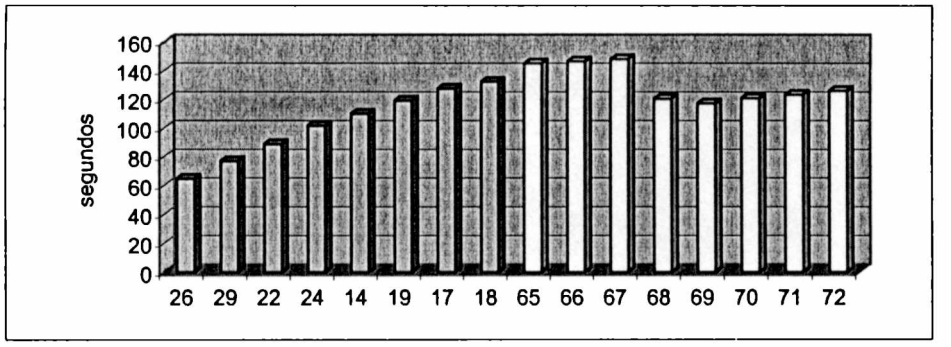

Figura 5.7: resultados obtenidos en el conjunto de prueba 2, matriz de 18112 x 18112 flotantes

En este caso, el comportamiento de los tiempos es similar al del primer caso de este conjunto de pruebas, utilizando una matriz de orden 20160.

4) Matriz de 15360 x 15360 flotantes

	Filas asignadas	Trabajo ideal	Trabajo real	Carga
Watson 26, 29 y 22	1600	10.10%	10.41%	103.06%
Pentium III 65..72	320	2.39%	2.08%	87.02%

Tabla 5.11: distribución obtenida en el conjunto de prueba 2, matriz de 15360 x 15360 flotantes

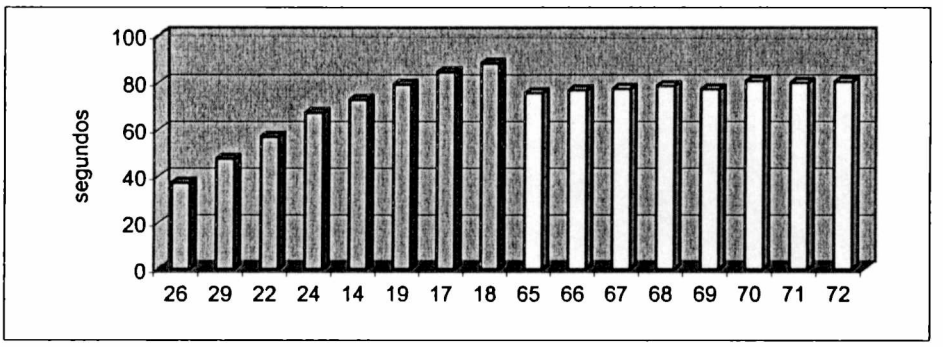

Figura 5.8: resultados obtenidos en el conjunto de prueba 2, matriz de 15360 x 15360 flotantes

En todo el conjunto de prueba 2, se observa que los tiempos de los nodos de distintos tipos logran estar más balanceados que en el conjunto de prueba 1. Esto muestra que el benchmark correcto (o al menos más correcto) para este problema paralelo es la factorización LU secuencial por bloques.

Otra característica de este conjunto de prueba es que se vuelve a observar el comportamiento "en escalera" de los tiempos de los nodos de tipo Pentium 4, aunque los mismos reciban la misma cantidad de bloques y tengan la misma capacidad. Esto permite pensar que la ubicación de los bloques en la matriz influye en la cantidad de procesamiento que realiza cada nodo. Esta característica se ha explicado en el capítulo anterior en la sección 4.5.2.3. Las figuras 4.17 a 4.19 de ese mismo capítulo permiten observar este comportamiento. El siguiente conjunto de pruebas corresponde a la mejora realizada en el método de distribución para intentar solucionar este problema.

5.4.2.3 Conjunto de prueba 3

En este conjunto de prueba se utiliza el método de distribución directo mejorado cíclico. El benchmark utilizado es la factorización LU secuencial con procesamiento de bloques. La configuración del cluster utilizado es como se muestra en la Tabla 5.12 (igual configuración que en el conjunto de prueba anterior).

Cantidad	Nombre	CPU	Frec. Reloj	Memoria	Mflops LU por bloques
8	WatsonX	Pentium 4	2400 MHz	1GB	5994
8	Lidipar65..72	Pentium III	700 MHz	64 MB	1418

Tabla 5.12: configuración del cluster del conjunto de prueba 3

Los tamaños de matrices utilizados son también los mismos que para la prueba anterior. Tanto la configuración del cluster como así también los tamaños de matrices se mantienen igual que en el conjunto de pruebas 2 con el fin de verificar si realmente la forma de distribuir los bloques pertenecientes a cada grupo determina el comportamiento observado en el conjunto de pruebas 2.

Los resultados obtenidos son los siguientes:

1) Matriz de 20160 x 20160 flotantes

	Filas asignadas	Trabajo ideal	Trabajo real	Carga
WatsonX	2048	10.10%	10.15%	100.49%
PentiumIII 65, 66 y 67	512	2.39%	2.53%	105.85%
Pentium 68..72	448	2.39%	2.22%	92.88%

Tabla 5.13: distribución obtenida en el conjunto de prueba 3, matriz de 20160 x 20160 flotantes

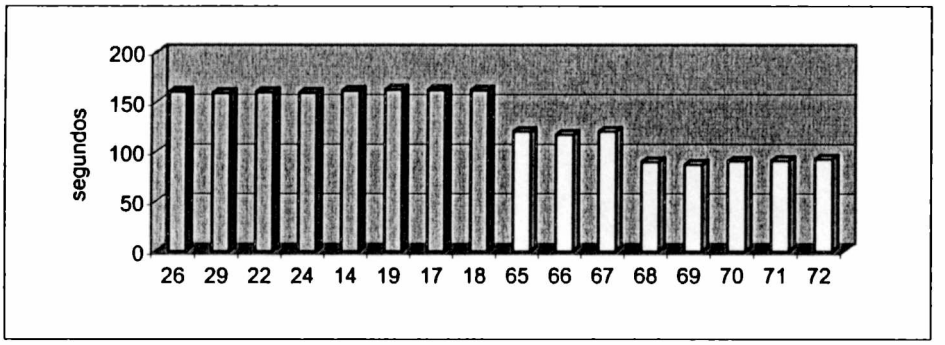

Figura 5.9: resultados obtenidos en el conjunto de prueba 3, matriz de 20160 x 20160 flotantes

2) Matriz de 19136 x 19136 flotantes:

	Filas asignadas	Trabajo ideal	Trabajo real	Carga
Watson26,29,22	1984	10.10%	10.36%	102.57%
Watson24,14,19,17,18	1920	10.10%	10.03%	99.3%
Pentium 68..72	448	2.39%	2.34%	97.9%

Tabla 5.14: distribución obtenida en el conjunto de prueba 3, matriz de 19136 x 19136 flotantes

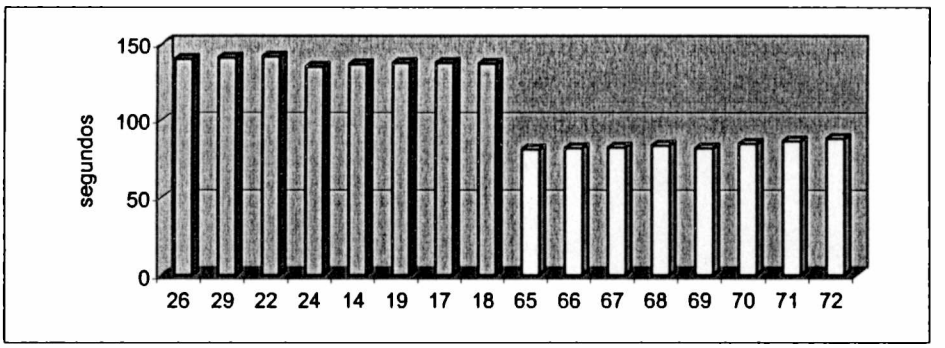

Figura 5.10: resultados obtenidos en el conjunto de prueba 3, matriz de 19136 x 19136 flotantes

3) Matriz de 18112 x 18112 flotantes

	filas asignadas	Trabajo ideal	Trabajo real	Carga
Watson26,29,22	1856	10.10%	10.24%	101.38%
Pentimu 64..67	448	2.39%	2.47%	103.34%
Pentium 68..72	384	2.39%	2.12%	88.7%

Tabla 5.15: distribución obtenida en el conjunto de prueba 3, matriz de 18112 x 18112 flotantes

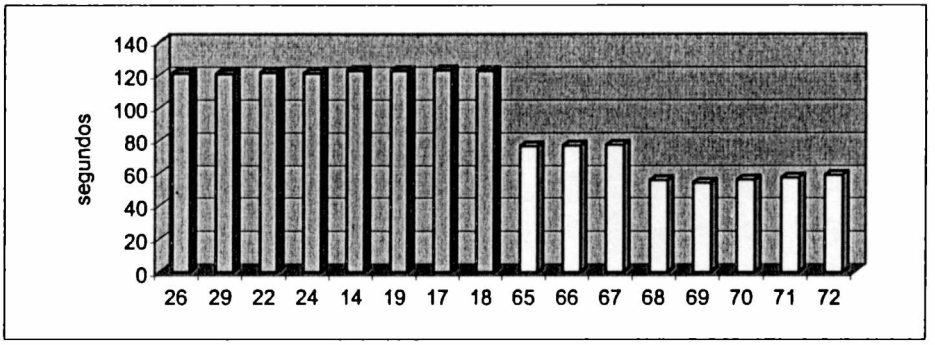

Figura 5.11: resultados obtenidos en el conjunto de prueba 3, matriz de 18112 x 18112 flotantes

4) Matriz de 15360 x 15360 flotantes

	Filas asignadas	Trabajo ideal	Trabajo real	Carga
WatsonX	1600	10.10%	10.41%	103.06%
Pentimu 65..72	320	2.39%	2.08%	87.02%

Tabla 5.16: distribución obtenida en el conjunto de prueba 3, matriz de 15360 x 15360 flotantes

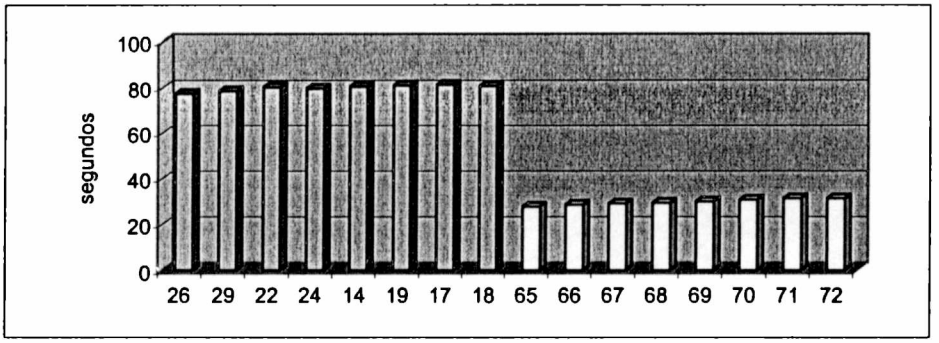

Figura 5.12: resultados obtenidos en el conjunto de prueba 3, matriz de 15360 x 15360 flotantes

Este conjunto de pruebas permite realizar las siguientes observaciones:

1) Desaparece el comportamiento en escalera observado en los conjuntos de prueba anteriores. Este comportamiento era más notorio en los primeros 8 nodos por recibir más bloques consecutivos de cada uno de los grupos. A partir de esta distribución, los nodos de un mismo tipo, los cuales reciben la misma cantidad de bloques trabajan aproximadamente el mismo tiempo.

2) El tiempo total observado en cada una de las pruebas es menor si se las compara con las pruebas realizadas en el conjunto de pruebas 2.

3) Aunque el método de distribución provoca un pequeño desbalance de carga, la diferencia entre los nodos que más trabajan y los que menos trabajan es grande (más grande del que se esperaba a causa del desbalance de carga del método de distribución). Esta diferencia debería ser menor.

En el punto 2 se concluye que los tiempos totales logrados con este método son menores que en los conjuntos de pruebas anteriores. Comparando los tiempos para cada tamaño de matrices se tiene que:

- Matriz de 20160 x 20160: el peor de los tiempos distribuyendo secuencialmente los bloques de cada grupo, es de aproximadamente 200 segundos, mientras que distribuyendo cíclicamente se logra aproximadamente 150 segundos. La mejora es de alrededor del 25%.

- Matriz de 19136 x 19136: se logra aproximadamente 160 segundos distribuyendo secuencialmente y 140 distribuyendo cíclicamente. La mejora es del 12.5% (aproximadamente).

- Matriz de 18112 x 18112: secuencialmente tarda en el peor de los tiempos alrededor de 140 segundos y cíclicamente 120 segundos. La mejora es del 14.28% aproximadamente.

- Matriz de 15360 x 15360: secuencialmente y el peor tiempo supera los 80 segundos y cíclicamente no llegan a los 80 segundos. La mejora es del 5% aproximadamente.

Estas observaciones permiten ver que la mejora propuesta en este método, además de evitar el comportamiento en escalera de nodos del mismo tipo, disminuye los tiempos totales de resolución de la factorización LU.

Además, se puede observar que este método no resuelve del todo el problema del balance de carga. Se puede ver una diferencia de carga entre los nodos de los distintos tipos. Este problema ya se había notado y se solucionó en parte con el cambio y la elección correcta del benchmark para esta factorización. Pero no se solucionó del todo, y posibles mejoras para evitar este desbalance se enumeran en el Apéndice A.

5.5 Resumen

En este capítulo se muestran las experimentaciones realizadas y los resultados obtenidos. Primero se enumeran las principales características de las experimentaciones, para poder entender qué aspectos del algoritmo se quisieron evaluar y cómo se hizo.

La experimentación evalúa los distintos métodos de distribución de carga que se propusieron y cómo funcionan para distintos clusters y para distintos tamaños de matrices.
El conjunto de pruebas elegido y mostrado fueron elegidas ya que mostraron distintas características de los métodos evaluados. Las pruebas se dividieron en grupos, y cada conjunto de pruebas es para un cluster específico, un benchmark determinado y se varía el tamaño de matriz para evaluar distintos comportamientos del método.

6. Capítulo 6: Conclusiones

En la actualidad existe una fuerte tendencia de utilizar las computadoras para cualquier tipo de trabajo. Estos trabajos pueden tener un amplio rango de necesidad de cómputo. Desde las pequeñas aplicaciones para PCs comunes de escritorio hasta aplicaciones con mucha carga de procesamiento o procesamiento sobre grandes volúmenes de datos, donde tal vez dicho procesamiento lleve días de cómputo.

Estas últimas aplicaciones son las que motivan al campo científico de la informática y de la ingeniería a desarrollar computadoras con cada vez más potencia de cálculo, más capacidad de memoria, mayor velocidad de entradas/salidas, etc. De todas formas, se puede tener una computadora con rendimientos máximos en estos aspectos, pero si el software que corre sobre la misma no explota estas capacidades, no sirve de nada toda la potencia ofrecida por el hardware.

Por otro lado, se está usando la informática para solucionar problemas cada vez más complejos. Esto hace que se esté desarrollando software con gran carga de cómputo y cómputo cada vez más complejo.

Estas dos tendencias hacen que el desarrollo del software y del hardware no sean caminos separados, sino todo lo contrario, el avance de uno de los dos campos no puede darse si no se avanza en el otro. El resultado de esto se puede ver en la actualidad, el avance de la informática en general incluye tanto hardware como software.

Este trabajo sigue esta tendencia, donde se consideran tanto el software como el hardware: se toma una operación en particular y se la trata de optimizar para una arquitectura de hardware específica, donde dicha arquitectura se puede ver como una arquitectura con grandes posibilidades de perdurar y de crecer en el mundo de la informática.

En este capítulo se mostrarán las conclusiones arribadas durante el desarrollo de la presente tesina.

6.1 Problemas de álgebra lineal

Durante el desarrollo de este trabajo se estudiaron un conjunto de problemas de cómputo científico en particular: problemas de álgebra lineal. El estudio de este campo permitió ver que existe mucho trabajo en él y que también queda mucho por hacer, pues a medida que la tecnología siga avanzando, será necesario proponer y encontrar nuevas soluciones. Esto no es algo nuevo, sino que está sucediendo desde hace ya un tiempo, se puede ver en todo el esfuerzo y trabajo que existe en encontrar soluciones a este tipo de problemas y en cómo las mismas han ido "evolucionando".

Todo el estudio sobre este área permitió notar que:

- Las operaciones de álgebra lineal son ampliamente utilizadas por distintos tipos de aplicaciones. A su vez, las aplicaciones que las utilizan son de cualquier tipo, no necesariamente pertenecen a un área en particular.

- Ya existe mucho trabajo realizado con el objetivo de encontrar soluciones optimizadas a este tipo de problemas.

- A las operaciones más utilizadas de este campo se las dividió en subconjuntos determinados por la complejidad de cada una de ellas.

- Existen distintas librerías que implementan estas operaciones para distintos tipos de arquitecturas.

Se puede notar que los últimos tres puntos están muy relacionados, tal vez los dos últimos sean consecuencia de la segunda observación. La gran cantidad de trabajo existente sobre este tema permite ver que estas operaciones son muy utilizadas y es importante para las aplicaciones que las utilizan tener versiones con rendimiento óptimo de las operaciones que específicamente utilicen.

Entre las librerías existentes (a las que hace referencia el último punto), se pueden mencionar EISPACK, LINPACK, LAPACK, ScaLAPLACK, BLAS, etc. Dentro de BLAS las operaciones se dividen en 3 niveles definidos por los requerimientos de cada una de las operaciones. Se pudo observar que en el Nivel 3 de BLAS es donde se ha puesto mayor empeño en optimizar las soluciones, pues son las operaciones con más requerimiento de cómputo.

La mayoría de los textos consultados durante este trabajo permitieron observar el mismo objetivo: alcanzar máximo rendimiento en la resolución de estas operaciones. Además se pudo observar que una forma casi inevitable de alcanzar buen rendimiento es desarrollar las soluciones para una arquitectura en particular. De esta forma, se pueden tener en cuenta las características de la misma para tratar de obtener el mayor provecho como así también evitar posibles penalizaciones en el rendimiento de la solución encontrada.

Esta forma de obtener soluciones específicamente orientadas a una arquitectura en particular, atenta contra el concepto de portabilidad de software. En los últimos tiempos se puede ver que existe una fuerte tendencia a desarrollar software que corra sobre cualquier plataforma (un ejemplo puede ser el amplio uso de JAVA). Esta tendencia es válida y muy útil en áreas donde el rendimiento no es importante, o por lo menos, la portabilidad es más importante que el rendimiento.

Sin embargo, resulta directo notar que en el campo de aplicaciones donde la carga de trabajo es mucha, el rendimiento se vuelve fundamental aunque se esté penalizando portabilidad.

Esta tendencia se observa en áreas de cómputo masivo, donde existe una gran carga de trabajo. Se pueden encontrar puntos en común entre la evolución de las distintas arquitecturas para cómputo masivo (mayormente computadoras paralelas) y las distintas soluciones a un mismo problema: por ejemplo, en la bibliografía de cómputo paralelo, es fácil encontrar distintas soluciones a un mismo problema pero desarrollado para arquitecturas de memoria compartida, pasaje de mensajes, etc.

También se pudo ver que este campo es muy amplio y que es imposible trabajar sin acotar el área de trabajo. En este trabajo en particular, se toman las operaciones de álgebra lineal que están definidas en la librería LAPACK y dentro de esta librería se toma una operación en particular: la factorización LU de matrices.

6.2 Clusters Heterogéneos

Este trabajo se inicia con el objetivo de encontrar una solución paralela a la factorización LU y que esta solución tenga buen rendimiento. Para lograr este objetivo, se encontró necesario desarrollar el algoritmo desde el punto de vista de la arquitectura específica a utilizar.

La bibliografía consultada permitió ver que esta forma de desarrollar una solución considerando las características del hardware a utilizar tiene muy buenos resultados, pues se alcanza mayor rendimiento. Por esto, en este trabajo se continúa con esta tendencia.

Esto hizo necesario estudiar las principales características de los clusters heterogéneos. Pero, ¿qué es un cluster heterogéneo? Un cluster heterogéneo es básicamente una computadora paralela formada por múltiples nodos interconectados con una red Ethernet. Los nodos no tienen que ser de algún tipo en particular, pueden ser computadoras de escritorio comunes, estaciones de trabajo, etc. En definitiva, un cluster heterogéneo es una red LAN utilizada para cómputo paralelo.

El estudio de los clusters heterogéneos permitió encontrar las siguientes características en los mismos:

- Muy bajo costo.
- Alta disponibilidad.
- Fácil de mantener y actualizar.
- Alta escalabilidad.

Tal vez la primera característica determine al resto. Estas características hacen que esta arquitectura sea muy conveniente, pues se tiene cómputo paralelo a muy bajo costo. Además, si se trata de una red ya instalada, no hay costo de hardware e instalación (la red ya está en uso, sólo que ahora se utiliza para aplicaciones paralelas) y el mantenimiento es el mantenimiento de la red. Sólo se necesita instalar el software necesario para la administración y ejecución de aplicaciones paralelas. Actualmente en Internet se puede encontrar de forma gratuita todo este software necesario. Por lo tanto, se dispone de una arquitectura con muy bajo costo y con la posibilidad de alcanzar muy altas prestaciones.

Pero esta arquitectura tiene características negativas desde el punto de vista del procesamiento paralelo:

- Red de interconexión lenta.
- Heterogeneidad de procesamiento en los nodos.

La primera característica es debido a que la red de interconexión es una red Ethernet y la misma no ha sido definida ni pensada para cómputo paralelo. Esto hace que características tales como tiempo de latencia y ancho de banda, que se trata de mejorar en la definición de redes de interconexión de computadoras paralelas, en el caso de la red Ethernet no se tienen como objetivo.

Además este tipo de red sigue inicialmente el protocolo 802.3 donde todas las comunicaciones se realizan como comunicaciones de tipo broadcast, donde un mensaje se envía por un único emisor y se transmite por el único canal compartido. De esta forma llega a todos los nodos conectados al medio. Así, se puede ver que el protocolo realiza comunicaciones de tipo broadcast a nivel físico.

La heterogeneidad en el procesamiento está dada en que los nodos que se utilizan pueden ser de cualquier tipo. Esta característica es a la vez una ventaja y una desventaja. Como ya se mencionó antes, es conveniente desde el punto de vista del costo, escalabilidad, mantenimiento, etc., pues cualquier nodo puede ser parte del cluster (PC, estaciones de trabajo, etc.). Pero a su vez, la heterogeneidad en el cómputo hace que se tenga que poner especial cuidado en la distribución de carga.

Estas propiedades son justamente las que definen algunas de las características en el algoritmo implementado. Así, se pudo implementar un algoritmo que evita posibles penalizaciones debido a la baja velocidad de la red de interconexión, pues trata de maximizar la relación procesamiento/comunicaciones, e implementa todas las comunicaciones como mensajes broadcast. Además realiza la distribución de carga en función a la capacidad de cada nodo (para lograr balance de carga).

Luego, se tiene una arquitectura muy beneficiosa desde el punto de vista del costo y la disponibilidad (y una disponibilidad cada vez mayor). A esto se le agrega que si se tienen en cuenta las características de la misma, es posible obtener muy buen rendimiento en aplicaciones paralelas.

6.3 Factorización LU de matrices

Se trabajó en esta factorización por varias razones, la más importante es que esta operación es muy utilizada, por lo tanto su optimización es muy conveniente para todos los campos en que se la utiliza. Esta operación es utilizada para resolver sistemas de ecuaciones lineales. Además, existen otras factorizaciones de matrices (también definidas dentro de LAPACK) las cuales comparten ciertas características con la factorización LU, por lo tanto, todo o parte del esfuerzo puesto en esta factorización puede ser utilizado para otras factorizaciones (por ejemplo, factorización QR, Cholesky, etc.). Otra de las razones es que tiene un patrón de procesamiento de datos y dependencia de datos que hace que sea una operación interesante de solucionar desde el punto de vista que no es una tarea trivial en una arquitectura paralela no homogénea.

Durante el trabajo se han analizado distintas soluciones de la factorización. Se comenzó analizando e implementando la solución secuencial con el fin de entender en qué consiste realizar la factorización LU en los datos de una matriz. Este algoritmo secuencial sólo se incluyó de forma teórica en este trabajo, pues en la práctica se utilizó una rutina de LAPACK para factorizar los bloques en la solución paralela. No se ha utilizado el método secuencial implementado porque el método definido en LAPACK se encuentra optimizado por lo que utilizar esta rutina implica obtener mejor rendimiento.

En la bibliografía consultada se han encontrado distintas soluciones paralelas a esta factorización. La búsqueda se orientó a soluciones para ambientes paralelos distribuidos. Antes de la solución definitiva, se implementó una solución paralela para ambientes distribuidos pero dicha solución no incluía la posibilidad de solapar cómputo con comunicación. Luego, se implementó la solución definitiva que tiene en cuenta esta posibilidad.

Entonces, con relación al método (factorización LU) no se ha propuesto ni implementado nada que no existiera con anterioridad, es más, se implementa la idea propuesta en [7] que soluciona de forma correcta la factorización para el tipo de arquitectura utilizada (realizando las modificaciones y mejoras necesarias). La solución propuesta se implementó utilizando el lenguaje de programación C y las rutinas de comunicación especificadas en el punto 5.3 del capítulo anterior. A este algoritmo se le suma todo el tratamiento especial para la distribución de carga. El algoritmo implementado tiene las siguientes características:

- Modelo de programación SPMD.
- Todas las comunicaciones son de tipo broadcast.
- Se solapa cómputo con comunicación.
- Balance de carga definido por la distribución de los datos.

- Todas las rutinas utilizadas para resolver distintas operaciones pertenecen a LAPACK (factorización LU) y a BLAS (resolución de sistemas de ecuaciones triangulares y multiplicación de matrices). Todas estas operaciones están optimizadas.

Se puede notar que todas estas características excepto la última, están definidas directa o indirectamente por la arquitectura utilizada: el tipo de red define que toda la comunicación del algoritmo se haga a través de mensajes broadcast, se tiene en cuenta la posible capacidad de los nodos de solapar comunicación con cómputo y el balance de carga se obtiene repartiendo de forma correcta los datos entre todos los nodos.

En la distribución de carga es donde se puso mayor atención con el fin de encontrar algún método que obtenga balance de carga en un ambiente heterogéneo. Y como resultado se obtuvieron los distintos métodos de distribución de bloques que se han propuesto (capítulo 4) y analizado (capítulos 4 y 5).

La siguiente subsección enumera las principales conclusiones a las que se han arribado después de la evaluación de cada uno de estos métodos.

6.3.1 Métodos de distribución

Como ya se ha mencionado, la distribución de datos para esta factorización no es trivial en un ambiente heterogéneo. La distribución de datos debe lograr balance de carga para no penalizar el rendimiento total por el rendimiento de algún nodo en particular.

En un ambiente homogéneo el balance de carga en esta factorización es fácil de lograr, pues simplemente se reparten a todos los nodos la misma cantidad de datos, lo cual implica obtener tiempos de cómputo similares, logrando así balance de carga. Pero este ambiente de trabajo es distinto, y lograr balance de carga en un ambiente heterogéneo no es tan fácil.

Para lograr balance de carga en este contexto, la distribución de datos debe ser como se explica claramente en [4]:

- La carga de trabajo de cada nodo debe ser proporcional a su capacidad de cómputo: se debe asignar la misma cantidad de tiempo de procesamiento que no necesariamente se corresponde con la misma cantidad de datos.

- La cantidad de bloques asignado a cada máquina es proporcional a su capacidad comparada con la capacidad total de la máquina paralela.

El último punto permite definir:

$$T = \mathrm{P}(p_i)/\ \mathrm{P}_{\mathrm{total}} \qquad (6.1)$$

Donde T es la carga obtenida, $\mathrm{P}(p_i)$ es la potencia del nodo i y $\mathrm{P}_{\mathrm{total}}$ es la potencia total de la máquina paralela. Así, T es teóricamente correcto, pero no es completamente aplicable en la

práctica porque el número de bloques asignados debe ser un número entero de bloques, y además, pueden quedar bloques restantes los cuales también hay que repartirlos como bloques enteros entre todos los nodos.

De esta forma se han propuesto distintos métodos de distribución de carga, los cuales reparten los datos en función a las capacidades de cada uno de los nodos. Entonces, un problema que surgió fue cómo caracterizar de la forma más correcta posible el rendimiento de cada uno de los nodos para la factorización LU. La experimentación demostró que una mala caracterización produce desbalance de carga en la solución paralela. Este problema se comprobó cuando se utilizó como benchmark la factorización LU secuencial sin utilizar procesamiento por bloques. Este benchmark arrojó un rendimiento que no era el correcto para la solución paralela que resuelve la factorización por bloques. Se pudo comprobar en los resultados obtenidos un desbalance de carga que luego se pudo explicar (y evitar) utilizando como benchmark la factorización LU secuencial por bloques.

Así se puede concluir que: tener en cuenta la capacidad de los nodos agrega el problema de encontrar la forma correcta de caracterizar esta capacidad. Específicamente hablando de la factorización LU, una forma correcta de hacerlo es determinar el rendimiento de la factorización LU secuencial por bloques en cada uno de los nodos. De esta forma, se tiene un benchmark correcto para este problema.

Por otro lado, se han propuesto distintos métodos de distribución de datos con el objetivo de obtener balance de carga. El primer método para clusters heterogéneos (llamado método directo) tiene en cuenta la parte más significativa de lo Mflops de cada uno de los nodos y reparte los bloques a partir de dicha potencia. Este método resultó ser muy simple para implementar pero el balance de carga obtenido depende de:

- La cantidad de bloques que recibe cada nodo de cada grupo no sea muy grande.
- La cantidad de bloques en cada grupo sea chica.
- No quede ningún grupo incompleto.

Si se cumplen los puntos enunciados anteriormente, el método puede funcionar de forma correcta. Pero no funciona correctamente cuando alguno de estos puntos no se cumple. Los primeros dos puntos están muy relacionados, pues muchos bloques consecutivos de un grupo para un nodo genera seguramente que el tamaño de grupo sea grande. Estas características pueden generar desbalance de carga debido a cómo procesan sus bloques cada uno de los nodos. Si un nodo tiene muchos bloques consecutivos de un grupo, los resuelve todos juntos y disminuye en gran medida la cantidad de bloques activos que le quedan en el resto de la factorización.

Otro problema importante de este método es que distribuye indistintamente los bloques de todos los grupos, incluyendo un posible grupo incompleto. Como se mostró en el capítulo 4, esto puede derivar un gran desbalance de carga, pues puede pasar que nodos consecutivos con potencias parecidas (o incluso iguales), pueden tener mucha diferencia de carga de trabajo. Esto genera un desbalance de carga indeseado y que se puede evitar.

El siguiente método propuesto tiene en cuenta los problemas encontrados en el método anterior: intenta generar grupos con menos cantidad de bloques y trata de forma especial los bloques del último grupo en el caso de que el mismo sea incompleto. Para esto, realiza ciertos cálculos sobre las potencias de los nodos que componen el cluster. Los cálculos son tales que generan bloques más chicos y mantienen la relación entre la potencia de cada nodo con la potencia total del cluster. De esta forma soluciona el primer problema del método anterior. Para solucionar el segundo, se intenta repartir los últimos bloques restantes entre todos los nodos y en función a su capacidad. Entonces, se realizan nuevos cálculos para determinar cuántos bloques de ese último grupo le corresponden a cada uno de los nodos. De esta forma, se intenta que todos los nodos reciban bloques del último grupo incompleto y en función de la capacidad de cada uno. Si luego de esto siguen quedando bloques sin asignar, se reparten uno a uno entre los nodos comenzando desde el primer nodo. De esta forma, se asignan bloques a los nodos de mayor capacidad (dado el orden de los nodos). Y el desbalance es de a lo sumo un solo bloque.

Este último método demostró ser mejor que el primero, los cálculos realizados sobre las potencias originales y el tratamiento especial del último grupo si es incompleto son favorables para obtener balance de carga. A pesar de estas mejoras, este método presentó un problema. Se producía un desbalance de carga no esperado entre nodos con igual potencia e igual cantidad de bloques asignados. Se esperaba que nodos de igual potencia, con la misma cantidad de trabajo para resolver, tarden tiempos similares. Esto no sucedía, existía un comportamiento de los tiempos que crecía a medida que se avanzaba de nodo en nodo (por eso se lo llamó en "escalera"). Este comportamiento hizo pensar que el avance de la factorización podía estar influyendo y generando este desbalance entre nodos del mismo tipo.

Se comprobó que la distribución de los bloques en esta factorización acostumbra a realizarse de forma cíclica para evitar que se produzca desbalance de carga producido por el avance mismo de la factorización. Así, se propone distribuir los bloques de la matriz de forma cíclica y no secuencial. De esta forma, no se produce este desbalance y todos los nodos participan en la mayoría de las iteraciones, pues tienen bloques activos durante toda la factorización. Pero el desbalance por la distribución secuencial también se puede dar dentro de un grupo. Si se consideran los bloques pertenecientes a un grupo solamente, se puede ver que también se produce este desbalance por el avance de la factorización. Además, esto se produce en cada uno de los grupos, por lo tanto, el desbalance total producido por esto puede llegar a ser importante.

Lo enunciado anteriormente sirvió para proponer una mejora al método anterior que es distribuir los bloques de cada grupo también en forma cíclica. En la bibliografía consultada donde se propone dividir los bloques en grupos no se especifica cuál sería la mejor forma de distribuir los mismos. Este trabajo sirvió para establecer que para esta factorización es mejor distribuir los bloques de un grupo de forma cíclica y no secuencial (siempre teniendo el cuenta la arquitectura específica con la que se está trabajando).

Utilizando esta forma de distribuir los bloques se observaron las siguientes mejoras:

- Los tiempos de procesamiento total disminuyeron con respecto al método anterior.

- Desaparece el comportamiento "en escalera" observado anteriormente.

De esta forma, se han propuesto distintos métodos de distribución de bloques para la factorización LU para ambientes heterogéneos. Se han evaluado en distintos clusters y estas evaluaciones han permitido realizar mejoras sobre los mismos. El último método tiene resultados satisfactorios y se puede utilizar sin importar la heterogeneidad del cluster. Inclusive, este método sirve en el caso de que se esté utilizando un cluster homogéneo, pues repartirá la misma cantidad de datos a todos los nodos. También es posible seguir trabajando sobre este método para intentar mejorar aún más el balance de carga obtenido.

Como pasos futuros de este trabajo, se han propuesto otras variantes a los métodos de distribución analizados en los capítulos anteriores. Estas variantes intentan seguir mejorando el método, o sea, obtener mejor balance de carga con la distribución de datos. Estas variantes se muestran en el Apéndice A.

A. Apéndice A

A.1 Introducción

Los pruebas realizadas y mostradas en el capítulo 5 de este trabajo permitieron ver que el método propuesto aunque logra buen balance de carga y ha solucionado ciertos problemas encontrados, todavía tiene un desbalance que se puede intentar mejorar. El desbalance se produce entre nodos de los distintos tipos (se puede observar bien en el conjunto de pruebas 3, sección 5.4.2.3). En este apéndice se muestran 2 alternativas al método de distribución que intentan solucionar esto.

El primero de los métodos tiene en cuenta los pesos de cada bloque de la matriz. El peso de cada bloque intenta representar la cantidad de trabajo que se realiza con ese bloque durante la factorización entera de la matriz. Este método se implementó y se probó pero el balance de carga obtenido fue igual que para el último método evaluado, pues se puede ver que aunque se tuvo en cuenta el peso de cada bloque, los bloques se distribuyeron de la misma forma.

Al método anterior (con pesos) no se ha intentado mejorarlo, sino que directamente se propuso otra forma de distribuir los datos ya que se supone que puede llegar a ser mejor que el método anterior. En este último método, se tiene en cuenta la posición de los bloques dentro de cada grupo y se asignan teniendo en cuenta la misma. Este último método no se incluye en los últimos capítulos de la tesina ya que aún no se tienen resultados con el mismo.

A.2 Distribución Cíclica Con Peso

En esta distribución se mantiene la división de los bloques en grupos, y dentro de cada grupo los bloques se distribuyen en forma cíclica. De esta forma, se mantiene el balance de carga entre los nodos y dentro de cada grupo.

A este método se le agrega el procesamiento de los pesos de cada bloque. El peso de un bloque es un número que representa la cantidad de trabajo que se realiza con ese bloque. En este caso, se propone caracterizar el peso de la siguiente manera: se usará la ubicación de cada bloque como su peso. Así, el peso del bloque en la posición *i* es *i*. Se propone esta caracterización porque el bloque *i* participa en 1 factorización, en *i*-1 actualizaciones y en tantos procesamientos de pivotes como filas haya. Por lo tanto, se supone que esta caracterización puede ser correcta ya que la cantidad de actualizaciones es lo que varía entre los distintos bloques. Y la cantidad de actualizaciones es igual a la posición que ocupa en la matriz menos 1. De esta forma, la posición del bloque dentro de la matriz puede llegar a caracterizar de forma correcta el peso o la cantidad de trabajo que se realiza con ese bloque.

Entonces, en la distribución de bloques por pesos se tiene en cuenta el peso que debería recibir cada uno de los nodos (peso ideal) y se trata de distribuir los bloques de tal forma que la sumatoria de los pesos de los bloques asignados a un nodo determinado sea similar al peso ideal para ese nodo. El peso ideal es directamente proporcional a la potencia de cada nodo, y se lo calcula en base al peso total de la matriz y la potencia del nodo.
Para esto, se tiene en cuenta:

- Potencia total del cluster.
- Potencia de cada nodo.
- Peso total de la matriz.
- Peso que idealmente debería resolver cada nodo del total de la matriz (determinado por su potencia).

El peso total de la matriz es la sumatoria de los pesos de todos los bloques en que se divide la matriz. Sea *k* la cantidad de bloques en que se divide la matriz:

$$\text{Peso matriz} = \sum_{i=0}^{k-1} pesoi = \frac{(k*(k+1))}{2} \qquad \text{(A.1)}$$

Entonces, teniendo en cuenta los pesos, la distribución se realiza de la siguiente manera:

1) Se dividen los bloques en grupos.

2) Se distribuyen los bloques de cada uno de los grupos de forma cíclica. El nodo *n* recibe el bloque *i* si y sólo si:
   a. La cantidad de bloques que recibió de ese grupo es menor a la que se le debe realmente asignar (como en los métodos anteriores).

b. El peso recibido por ese nodo es menor al peso ideal que debería recibir.

3) Si quedan bloques sin asignar:
   a. Se calcula el nodo con mayor diferencia entre el peso asignado y el peso ideal que debería recibir y ese nodo es el que recibe el bloque.

En este punto, si 2 o más nodos tienen una diferencia similar, se le debería asignar el bloque al nodo con mayor potencia de cómputo. Por lo tanto, en este punto también se tiene en cuenta el peso ideal (que depende de la potencia). Pero para evitar que este peso ideal domine el valor para comparar, se lo divide por el promedio de todos los pesos ideales.

Entonces, el código que implementa esta asignación de los bloques restantes es:

```
for (i=0; i < P; i++)
{
  diferencia = pesoIdeal[i] - pesoActual[i];

  if ((diferencia * pesoIdeal[i]/promedioPesos ) >
(diferenciaMax * pesoIdeal[indice]/promedioPesos ))
  {
    indice = i;
    diferenciaMax = diferencia;
  }
}
```

Figura A.1: pseudocódigo para la distribución de los bloques restantes teniendo en cuenta el peso de cada bloque.

Donde P es la cantidad de tareas que participan en la factorización. En la variable *indice* queda el índice del nodo que recibiría el bloque. Una vez que recibe el bloque se actualiza el vector *pesoActual* agregándose el peso del bloque asignado a dicho nodo.

De esta forma, se tienen en cuenta los pesos de los bloques y la potencia de cada uno de los nodos. El objetivo es evitar que los últimos bloques (los que se suponen más pesados) causen un desbalance de carga, penalizando así el cómputo paralelo.

Como ya se ha mencionado, se realizaron pruebas con este método pero no se han tomado tiempos pues la distribución obtenida fue la misma que para el conjunto de pruebas de la sección 5.4.2.3. Tampoco se ha intentado mejorarlo, pues se propuso otro método (explicado en la próxima subsección), que se supone que puede llegar a solucionar el problema encontrado.

A.3 Método "alternado"

Esta posible mejora sobre el método propuesto, toma algunos conceptos de los métodos anteriores. Los cálculos sobre las potencias propuestos en el punto 4.5.3.2 se realizan de la misma forma. El cambio en este método es la forma en que se reparten los bloques, aunque

se sigue manteniendo que los mismos se reparten en función a la capacidad de cada uno de los nodos. Este método tiene en cuenta la posición de cada bloque dentro del grupo.

Aunque el último conjunto de pruebas (sección 5.4.2.3) mostró que mejoró mucho el balance de carga, se puede observar una diferencia de tiempos entre los nodos de distintos tipos. Se puede ver que todos los nodos Pentium III están balanceados entre si y los Pentium 4 también están balanceados entre sí pero existe una diferencia de carga entre los distintos tipos de nodos. Esta propuesta intenta evitar esta diferencia entre nodos de distintos tipos.

Primero se analizará una característica del método propuesto en la sección 4.5.3.3 de este trabajo que se supone que puede llegar a ser el causante de dicha diferencia. Las características principales de ese método son: se reparten los bloques en función de la capacidad de cada nodo, se dividen los bloques en grupos y dentro de cada grupo los bloques se distribuyen de forma cíclica.

En las experimentaciones realizadas se han utilizado 16 nodos (se enumeran de 0 a 15 para mayor claridad, los nodos 0 a 7 son los Pentium III y los nodos 8 a 15 son los Pentium 4). Los bloques se dividen en grupos, dentro de cada uno se enumeran los bloques desde 0 hasta la cantidad de bloques por grupo. Entonces, en una distribución cíclica de bloques, el nodo 0 recibe el bloque 0 del grupo, el nodo 1 recibe el 1, y siguiendo, el nodo 8 recibe el bloque 8.

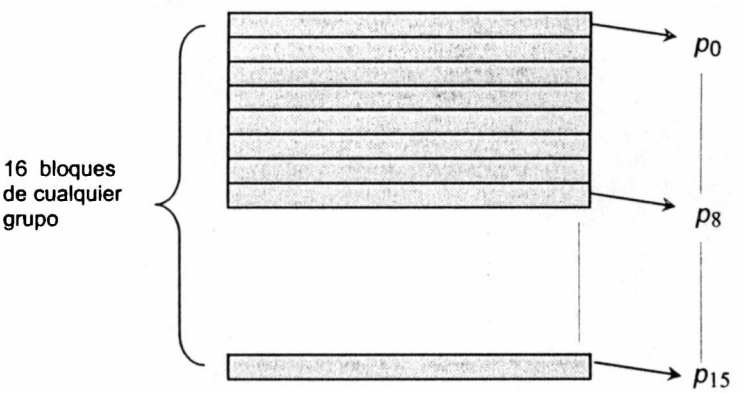

Figura A.2: distribución de los primeros 16 bloques de un grupo (esto sucede en todos los grupos).

En la figura A.2 se muestra la distribución de 16 bloques de un grupo. Se puede ver la diferencia que existe en las posiciones de los nodos que recibe el primer nodo (p_0) y primer nodo del próximo tipo de nodos (en este caso, p_8). Existen 7 bloques de diferencia, y esto se repite múltiples veces en cada uno de los grupos y en todos los grupos.

La diferencia observada de 7 bloques entre los distintos tipos de nodos tal vez esté influyendo en los tiempos obtenidos y es lo que trata de evitar la próxima modificación al método: se intenta distribuir bloques más próximos entre los distintos tipos de nodos.

Lo que se propone como posible mejora es distribuir los bloques disminuyendo las distancias de los bloques que se asignan a cada tipo de nodo. Para esto, se asignan de forma cíclica pero alternando entre los tipos de nodos. De esta forma, se propone distribuir los bloques como se muestra en la Figura A.3. Como todas, esta figura es representativa y se supone que los nodos con mayor potencia (nodo 0 a 7) son 3 veces más rápidos que los nodos de menor potencia (nodos de 8 a 15).

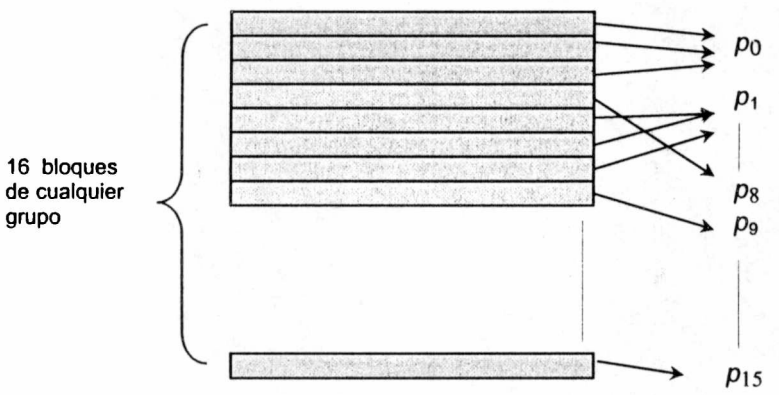

Figura A.3: distribución cíclica alternando los nodos de los distintos tipos.

Como se puede ver en la Figura A.3, la distancia entre los bloques que recibe cada tipo de nodo no es tan grande como sucedía en el método anterior. De esta forma, si la causa de las diferencias de tiempos observadas en el último método evaluado es la distancia entre los bloques que se asignan a cada tipo de nodo, entonces, con esta forma de distribuir los bloques se puede llegar a evitar.

Queda entonces como pasos futuros implementar y evaluar este método, para así determinar si las diferencias de posiciones de los bloques dentro de la matriz puede ser el causante de este desbalance observado anteriormente.

Luego del análisis de estas dos posibles alternativas presentadas en este Apéndice, se puede observar una diferencia de apreciación con respecto a los pesos de los bloques entre los distintos métodos: el método con pesos asume que el peso de los bloques aumenta a medida que crece su posición dentro de la matriz. Entonces, si esta caracterización de los pesos es correcta, el comportamiento observado en los tiempos debería ser totalmente al revés: los últimos nodos reciben los bloques con más peso de cada uno de los grupos.

Esto no sucede así, y tal vez sea porque la posición dentro de la matriz no sea una buena forma de caracterizar el peso del bloque. Si luego de la implementación y evaluación del segundo método se logra mejorar el balance de carga, quiere decir que la caracterización en el primero de los métodos de este Apéndice es incorrecta.

Referencias bibliográficas

[1] Akl S., Parallel Computation: Models and Methods Prentice Hall, Upple Saddle River, 1997.

[2] Anderson E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen. "LAPACK: A Portable Linear Algebra Library for High-Performance Computers". Proceedings of Supercomputing '90, IEEE Press, 1990.

[3] Andrews G., "Foundations of Multithreaded, Parallel, and Distributed Programming", Addison Wesley Longman, Inc. ISBN 0-201-35752-6. Año 2000.

[4] Barbosa J., J. Tavares and A. J. Padilha. "Linear Algebra Algorithms in a Heterogeneous Cluster of Personal Computers". FEUP-INEB Grupo de Arquitecturas e Sistemas.

[5] Bilmes J., Asanovic K, Demmel J. Lam, Chin C. "Optimizing Matriz Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodoly". LAPACK Working Note 111. UTK.

[6] Buyya, Rajkumar. High Performance cluster Computing. Arquitectures and Systems. Vol I. Prentice Hall Ptr. Upper Saddle River, New Jersey 07458. ISBN: 0-13-013784-7.

[7] Dackland K., E. Elmroth., "Design and Performance Modeling of Parallel Block Matrix Factorizations for Distributed Memory Multicomputers". Institute of Information Processing University of Umeå.

[8] Dackland K. E. Elmroth, "Parallel Block Matrix Factorization for Distributed Memory Multicomputers". Institute of Information Processing University of Umeå.

[9] Dongarra J., S. Hammarling, D. Walker. "Key Concepts for Parallel Out-Of-Core LU Factorization".

[10] Golub G. H., C. Van Loan "Matrix Computation", Second Edition, The John Hopkins University Press, Baltimore, Maryland, 1989. (pp.92-117)

[11] Gustafson J. "Computational Verifiability and Feasibility of the ASCI Program". Ames Laboratory, US Department of Energy. (Paper impreso)

[12] INTEL http://news.zdnet.com/2100-9584_22-5207837.html

[13] INTEL http://news.zdnet.com/2100-3513_22-5077336.html?tag=nl

[14] Kumar V, Grama A. Gupta A, Karypis G, "Introduction to Parallel Computing. Design and Analysis of Algorithms", The Benjamin/Cummings Publishing Company, Inc. 1994.

[15] LAPACK Users' Guide.Third Edition, disponible en dhttp://www.netlib.org/lapack/lug/lapack_lug.html.

[16] Proyecto Condor High Throughput Computing disponible en http://www.cs.wisc.edu/condor/overview/

[17] Sabot G.W., "High Performance Computing" (pp 94-130)

[18] Tinetti F., "LU Factorization: Number of Floating Point Operations and Parallel Processing in Clusters". Technical Report PLA-001-2003. Mayo 2003.

[19] Tinetti F. "Cómputo Paralelo en Redes Locales de Computadoras", Tesis Doctoral.Universidad Autónoma de Barcelona, Facultad de Ciencias. Marzo 2004.

[20] Tinetti F. "QR Factorization: Number of Floating Point Operations and Parallel Processing in Clusters". Technical Report PLA-001-2004. 2004.

[21] Tinetti F., A. Barbieri, "An Efficient Implementation for Broadcasting Data in Parallel Applications over Ethernet Clusters". Proceedings 17th International Conference on Advanced Information Networking and Applications (AINA'03), IEEE Press, ISBN 0-7695-1906-7, Xi'an, China, March 27 - 29, 2003, pp. 593-596.

[22] Tinetti F. A. De Giusti "Procesamiento Paralelo. Conceptos de Arquitecturas y Algoritmos". Editorial Exacta. Agosto 1998. ISBN 987-99858-5-0, La Plata, Buenos Aires Argentina.

[23] Tinetti F, M. Denham "Paralelización de la Factorización LU de Matrices en Clusters Heterogéneos", III-LIDI (Instituto de Investigación en Informática LIDI), Facultad de Informática UNLP.

[24] Tinetti F., E. Luque. "Efficient Broadcasts and Simple Algorithms for Parallel Linear Algebra Computing in Clusters",. Proceedings 17th International Parallel & Distributed Processing Symposium, IPDPS 2003. Nice, France April 22-26, 2003. IEEE Computer Society Order Number PR01926, ISBN 0-7695-1926-1, ISSN Number 1530-2075, p. 198.

[25] TOP500 SUPERCOMPUTER SITES disponible en http://www.top500.org/

[26] Weiss M. A., "Data Structures and Algorithm Analysis". (pp 31-32)