INSTITUTO TECNOLOGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS MONTERREY

DIVISION EN ELECTRONICA, COMPUTACION, INFORMACION Y COMUNICACIONES

PARALELIZACION DE LU UTILIZANDO BALANCEO DINAMICO EN UNA RED DE ESTACIONES DE TRABAJO HETEROGENEAS

TESIS

MAESTRO EN CIENCIAS CON ESPECIALIDAD EN CIENCIAS DE LA COMPUTACION

ANTONIO ARMANDO AGUILETA GÜEMEZ

AGOSTO DEL 2000

INSTITUTO TECNOLOGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS MONTERREY

DIVISION EN ELECTRONICA, COMPUTACION, INFORMACION Y COMUNICACIONES

PARALELIZACION DE LU UTILIZANDO BALANCEO DINAMICO EN UNA RED DE ESTACIONES DE TRABAJO HETEROGENEAS

TESIS

MAESTRO EN CIENCIAS CON ESPECIALIDAD EN CIENCIAS DE LA COMPUTACION

ANTONIO ARMANDO AGUILETA GÜEMEZ

AGOSTO DEL 2000

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

División en Electrónica, Computación, Información y Comunicaciones

Paralelización de LU utilizando Balanceo Dinámico en una Red de Estaciones de Trabajo Heterogéneas

TESIS

Presentada como requisito parcial para obtener el grado académico de

Maestro en Ciencias con especialidad en Ciencias de la Computación

Antonio Armando Aguileta Güémez

Agosto 2000

Paralelización de LU Utilizando Balanceo Dinámico en una Red de Estaciones de Trabajo Heterogéneas

por

Antonio Armando Aguileta Güémez

TESIS

Presentada a la División en Electrónica, Computación, Información y Comunicaciones

Este trabajo es requisito Parcial
Para Obtener el Título de
Maestro en Ciencias con
Especialidad en Ciencias de la Computación

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

Agosto del 2000

Dedicatoria

A Dios,
por todo lo que me ha dado ,
por estar siempre a mi lado,
aunque a veces no lo vi.

A mis padres Armando y Reyna,
hermanas y mi familia,
por todo el apoyo brindado.

Agradecimientos

Al Dr. David Garza Salazar, paciente asesor y excelente maestro por todos sus aportaciones y consejos para la realización exitosa de este trabajo.

Al Dr. Ignacio Celis, excelente profesor, por sus valiosas aportaciones que enriquecieron este trabajo.

Al Dr. Juan Arturo Nolazco, profesor destacado, por sus recomendaciones y comentarios para la realización exitosa de este trabajo.

Al Centro de Investigación en Informática por todas las facilidades otorgadas para la utilización de los equipos para la realización de esta tesis.

A todos mis amigos que siempre me apoyaron en los momentos mas difíciles.

Resumen

La contaminación ambiental es una de la grandes preocupaciones de la humanidad hoy en día. México y EE.UU. a través de SEDESOL y la agencia de protección ambiental (EPA) desarrollaron un plan ambiental para el área de la frontera México – Estados Unidos. Mejorar la comprensión de las condiciones ambientales de la frontera es uno de los mayores objetivos de este plan, en particular se pretende conocer la calidad del agua para consumo humano.

El ITESM y SWRI tienen un proyecto conjunto para simular el transporte de sustancias peligrosas en el subsuelo de la zona fronteriza México - Estados unidos usando para tal propósito el modelo MULTIFLO desarrollado por SWRI[MUL96]. Dentro de los objetivos de este proyecto esta crear una meta-computadora regional para la paralelización eficiente de éste modelo.

Uno de los algoritmos numéricos mas importantes sobre los cuales esta basado el modelo MULTIFLO es la descomposición de matrices mediante la técnica LU, que consiste en dividir la matriz A de entrada en dos matrices la triangula superior (Upper) y la triangular inferior (Lower).

En este trabajo se realizó una investigación de las técnicas de balanceo de carga dinámica para ambientes heterogéneo y no heterogéneos de redes de estaciones de trabajo. La investigación de estas técnicas marcaron la pauta para paralelizar LU utilizando balanceo dinámico en un ambiente heterogéneo de redes de estaciones de trabajo, con el objeto de reducir el tiempo de ejecución del algoritmo LU. En este trabajo también se realizaron las implementaciones secuencial y paralela de LU cuyos tiempos de ejecución sirvieron para analizar la versión paralela balanceada de LU.

La experimentación demostró que el programa paralelo balanceado de LU es mejor que el programa paralelo para todos los ambientes de prueba. Se mostró que para matrices de 512 la mejora relativa de los tiempos de ejecución del programa paralelo balanceado con respecto del programa paralelo va desde 1.12 veces mejor hasta 1.78, para matrices de 1024 la mejora relativa va desde 1.06 veces mejor hasta 2.54 y que para matrices de 2048 la mejora relativa va desde 1.08 veces mejor hasta 2.93. Los valores del Speedup mantuvieron una tendencia incremental, que fueron del 1.31 al 2.32, al aumentar el tamaño de la matriz A y una tendencia decremental al aumentar los nodos en todos los ambientes de pruebas. La eficiencia también mantuvo una tendencia incremental relativa del programa balanceado, *(Eb)*, con respecto del programa paralelo, *(Ep)*, que va desde 12% hasta 114%, que para matrices de 512, del 6% hasta el 154% para matrices de 1024 y por último del 8% hasta el 193%, para matrices de 2048. Finalmente se afirma que el programa paralelo balanceado de LU es significativamente mejor que el programa paralelo, ya que obtuvo mejoras relativas de hasta el 193% en la eficiencia comparado con el programa paralelo para todos los ambientes de pruebas.

Índice

Lista de Tablas ix
Lista de Figuras x
Capítulo 1 **Introducción** 1
1.1 Estructura de la Tesis 3

Capítulo 2 **Antecedentes** 4
2.1 Clasificación de las Computadoras de Acuerdo a su Arquitectura 6
2.2 Clasificación de las Computadoras Paralelas Según la Forma que Acceden la Memoria sus Procesadores 7
2.3 Balanceo de Carga 7
2.4 Descomposición de Matrices Mediante la Técnica LU 8
2.5 Trabajos Relacionados 10

Capítulo 3 **Paralelización de Descomposición LU** 12
3.1 Detalles de la Implementación del Algoritmo Paralelización de LU 12
3.2 Descripción del Algoritmo Paralelo LU 13
3.2.1 Particionamiento de Datos 13
3.2.2 Asignación y Ejecución de Tareas 14
3.3 Comportamiento del Algoritmo Paralelo LU 16

Capítulo 4 **Balanceo Dinámico de LU** 19
4.1 Diagrama Conceptual del Algoritmo Paralelo LU 19
4.2 Detalles de la Implementación del Algoritmo de Balanceo Dinámico de LU 22
4.3 Descripción del Algoritmo Paralelo Balanceado de LU 22
4.3.1 Particionamiento de Datos 22
4.3.2 Asignación y Ejecución de Tareas 25
4.4 Comportamiento del Algoritmo Paralelo Balanceado de LU 28

Capítulo 5 **Experimentos** 32
5.1 Entorno Experimental 32
5.2 Metodología 33
5.3 Métricas de Evaluación 34
5.3.1 Speedup 34
5.3.2 Speedup Óptimo Heterogéneo 35
5.3.3 Eficiencia en un Ambiente Heterogéneo 36
5.4 Resultados 36
5.4.1 Resultados de Ejecutar el Programa Paralelo en el Ambiente Homogéneo. 37
5.4.2 Frecuencia de Balanceo 40
5.4.3 Predicción de Carga 41

5.4.4 Resultados de los Algoritmos Paralelo y Paralelo Balanceado de LU 43

Capítulo 6 Conclusiones 49
6.1 Trabajos Futuros 51

Referencias Bibliográficas 52
Apéndice 53
Vita 75

Lista de Tablas

5.1 Ambiente de pruebas 33

5.2 Speedup óptimo heterogéneo 36

5.3 Speedup óptimo heterogéno de todos los ambientes de prueba 36

5.4 Resultados del algoritmo paralelo de LU para matrices de 256 37

5.5 Resultados del algoritmo paralelo de LU para matrices de 512 37

5.6 Resultados del algoritmo paralelo de LU para matrices de 1024 38

5.7 Resultado de experimentar con el algoritmo LU en todas sus versiones, para matrices de dimensión 512 44

5.8 Resultado de experimentar con el algoritmo LU en todas sus versiones, para matrices de dimensión 1024 44

5.9 Resultado de experimentar con el algoritmo LU en todas sus versiones, para matrices de dimensión 2048 44

Lista de Figuras

3.1	Particionamiento de datos en la paralelización de LU	13
3.2	Asignación de segmentos de la matriz A a cada estación de trabajo	14
3.3	Paralelización del algoritmo LU	15
3.4	Particionamiento de la matriz A, en la iteración 1	16
3.5	Elementos actualizados de las matrices A y LU, reflejados a través del coloreo	17
3.6	Particionamiento de la matriz A, en la iteración 2	17
3.7	Actualización de las matrices A y LU, reflejadas a través del coloreo	18
4.1	Diagrama conceptual del algoritmo paralelo balanceado de LU	20
4.2	Particionamiento de la matriz A	23
4.3	Cálculo del particionamiento de la matriz A	24
4.4	Definición de la tabla de información	25
4.5	Instancia de la tabla de información de los datos almacenados	26
4.6	Instancia de la tabla de información, con los datos de las filas que deberán ser enviadas y recibidas	26
4.7	Instancia de la tabla de información, después del intercambio de filas entre los procesadores	27
4.8	Paralelización con balanceo de LU	27
4.9	Particionamiento de la matriz A, en la iteración 1	28
4.10	Actualización de las matrices A y LU, al finalizar la iteración 1	29
4.11	Particionamiento de la matriz A, en la iteración 2	30

4.12 Actualizaciones sobre las matrices A y LU, al finalizar la iteración 2 30

5.1 Speedup de matrices de dimensión 256 38

5.2 Speedup de matrices de dimensión 512 38

5.3 Speedup de matrices de dimensión 1024 39

5.4 Eficiencia lograda en matrices de 256 39

5.5 Eficiencia lograda en matrices de 512 39

5.6 Eficiencia lograda en matrices de 1024 40

5.7 Tiempos y frecuencias del algoritmo paralelo balanceado de LU para matrices de dimensión 1024 41

5.8 Tiempos y frecuencias del algoritmo paralelo balanceado de LU para matrices de dimensión 2048 41

5.9 Tiempos y pesos de la formula de predicción para el algoritmo paralelo balanceado de LU para matrices de 1024 42

5.10 Tiempos y pesos de la formula de predicción para el algoritmo paralelo balanceado de LU para matrices de 2048 43

5.11 Mejoras relativas de los tiempos de ejecución del programa paralelo Balanceado respecto del programa paralelo, para matrices de 512, 1024 y 2048 45

5.12 Muestra los speedups de los programas paralelo de LU (Sp), paralelo balanceado de LU (Sb) y el óptimo heterogéneo (So), para matrices de dimensión 512 46

5.13 Muestra los Speedups de los programas paralelo de LU (Sp), paralelo balanceado de LU (Sb), paralelo balanceado de LU mejorado (Sbm) y el óptimo heterogéneo (So), para matrices de dimensión 1024 46

5.14 Muestra los Speedups de los programas paralelo de LU (Sp), paralelo balanceado de LU (Sb), paralelo balanceado de LU mejorado (Sbm) y el óptimo heterogéneo (So), para matrices de dimensión 2048 46

5.15 Muestra la eficiencia lograda por los programas paralelo de LU (Ep) y Paralelo balanceado de LU (Eb), para matrices de 512 47

5.16 Muestra la eficiencia lograda por los programas paralelo de LU (Ep), paralelo balanceado de LU (Eb) y paralelo balanceado de LU mejorado (Ebm), para matrices de 1024 47

5.17 Muestra la eficiencia lograda por los programas paralelo de LU (Ep), paralelo balanceado de LU (Eb) y paralelo balanceado de LU mejorado (Ebm), para matrices de 2048. 47

5.18 Mejora relativa en la eficiencia del programa paralelo balanceado respecto del Programa paralelo para matrices de 512, 1024 y 2048 48

Capítulo 1

Introducción.

Como resultado de un programa binacional de protección ambiental del área de la frontera, la SEDESOL y la agencia de Protección Ambiental (EPA) de los Estados Unidos desarrollaron un plan ambiental para el área de la frontera México – Estados Unidos [ENV92]. Mejorar la comprensión de las condiciones ambientales de la frontera es uno de los mayores objetivos de este plan. En los últimos años se han presentado problemas de salud de los habitantes de la frontera al ingerir aguas contaminadas. Unos de los casos más impactantes es el alto número de niños que nacieron sin cerebro comparado con otras regiones de ambos países. Hay una gran preocupación acerca de los niveles de contaminación de los recursos acuíferos subterráneos en la frontera. Sin embargo poco se conoce acerca de los niveles actuales de contaminación de estos acuíferos por lo que es muy importante estimar los niveles actuales y potenciales de contaminación para contestar preguntas importantes acerca de la localización o relocalización de industrias, sobre planeación urbana e industrial, prevención de contaminación y acciones remediales, identificación y efectos de fuentes de contaminantes, que asegurará el cumplimientos de las normas ambientales de la frontera entre México y los Estado Unidos, en particular sobre calidad del agua para consumo humano.

El ITESM y SWRI tienen un proyecto conjunto para simular el transporte de sustancias peligrosas en el subsuelo de la zona fronteriza México - Estados unidos usando para tal propósito el modelo MULTIFLO desarrollados por SWRI[MUL96]. Este proyecto tiene como objetivo general combinar los recursos computacionales, experiencia y recursos humanos de ambas instituciones para crear una meta-computadora regional y explorar los problemas más relevantes relacionados con la utilización de la meta-computadora para la paralelización eficiente de éste modelo.

La tecnología de software no ha avanzado al mismo ritmo que la tecnología de hardware. En la actualidad, la eficiente y transparente ejecución de programas en computadoras con memoria distribuida es un área muy activa de investigación y desarrollo[BEC96]. La paralelización automática aún no ha madurado lo suficiente para obtener programas eficientes para cualquier aplicación y arquitectura paralela.

El computo heterogéneo no es nuevo, sin embargo, las técnicas para la paralelización eficiente en ambientes heterogéneos es un área no muy explotada, no se sabe cuál es la mejor forma de paralelizar una aplicación en estos ambiente heterogéneos, particularmente en un ambiente de desarrollo, y mucho menos la forma de reducir su tiempo de ejecución, es decir, no se sabe cual es la configuración óptima de las variables características de la plataforma de desarrollo, las técnicas utilizadas para la paralelización y el balanceo de carga, las herramienta que permiten balancear la carga y las que permiten

implantar aplicaciones paralelas ya que los trabajos desarrollados son limitados a las características particulares de los ambientes de desarrollo.

Ante esta situación que se presenta ejecutar eficientemente el modelo MULTIFLO (una aplicación que demanda computo numérico) en un ambiente heterogéneo de redes de estaciones de trabajo y sin saber cual es la mejor configuración de las variables involucradas, se hace indispensable el desarrollo de trabajos que sienten las bases para la implementación de tales aplicaciones en ambientes similares.

Uno de los algoritmos numéricos más importantes sobre los cuales esta basado el modelo MULTIFLO es la descomposición de matrices mediante la técnica LU, que consiste en dividir la matriz A de entrada en dos matrices la triangula superior (Upper) y la triangular inferior (Lower).

En este trabajo se realizó una investigación de las técnicas de balanceo de carga dinámica para ambientes heterogéneos de redes de estaciones de trabajo. La investigación de estas técnicas marcó la pauta para paralelizar LU utilizando balanceo dinámico en un ambiente heterogéneo de producción de redes de estaciones de trabajo, con el objeto de reducir el tiempo de ejecución del algoritmo LU. En este trabajo también se realizaron las implementaciones secuencial y paralela de LU cuyos tiempos de ejecución sirvieron para analizar la versión paralela balanceada de LU.

Al final de este trabajo, a través de la experimentación, se observa que el programa paralelo balanceado de LU dio buenos resultados en todos los ambientes de prueba. Se mostró que para matrices de 512 la mejora relativa de los tiempos de ejecución del programa paralelo balanceado con respecto del programa paralelo va desde 1.12 veces mejor hasta 1.78, para matrices de 1024 la mejora relativa va desde 1.06 veces mejor hasta 2.54 y que para matrices de 2048 la mejora relativa va desde 1.08 veces mejor hasta 2.93. Los valores del Speedup mantuvieron una tendencia incremental, que fueron del 1.31 al 2.32, al aumentar el tamaño de la matriz A y una tendencia decremental al aumentar los nodos en todos los ambientes de pruebas. La eficiencia también mantuvo una tendencia incremental relativa del programa balanceado, *(Eb)*, con respecto del programa paralelo, *(Ep)*, que va desde 12% hasta 114%, que para matrices de 512, del 6% hasta el 154% para matrices de 1024 y por último del 8% hasta el 193%, para matrices de 2048. En conclusión se afirma que el programa paralelo balanceado de LU es significativamente mejor que el programa paralelo, ya que obtuvo mejoras relativas considerables en los tiempos de ejecución de hasta 2.93 veces mejor, Speedups de hasta 2.93 y eficiencias relativas de hasta el 193% comparado con el programa paralelo para todos los ambientes de pruebas.

1.1 Estructura de la Tesis.

Este documento esta estructurado de la siguiente forma:

- En el capítulo 2 se describe la teoría de los sistemas distribuidos en el cual se basa el desarrollo de este trabajo, a sí como la técnica de descomposición de matrices LU y se mencionan algunos trabajos que se relacionan con el balanceo de carga dinámico en ambientes heterogéneos.

- En el capítulo 3 se describe el algoritmo paralelo de LU, presentado los conceptos y detalles de su implementación.

- En el capítulo 4 se describe el algoritmo paralelo balanceado de LU, presentando los conceptos y detalles de su implementación.

- En el capítulo 5 se definen las métricas de evaluación y se analizan los resultados obtenidos de las ejecuciones del programa secuencia, paralelo y paralelo balanceado de LU utilizando dichas métricas

- En el capítulo 6 se presentan las conclusiones basadas en los resultados y se describen los posibles trabajos futuros que podrían contribuir al tema de balanceo dinámico.

Capítulo 2

Antecedentes.

Tradicionalmente la ciencia está basada en la observación, la teoría y experimentación, ya que de la observación surgen las teorías que a través de la experimentación se aprueban o refutan. Desafortunadamente no siempre es posible probar dichas teorías experimentando, ya que el experimento podría resultar muy costoso en tiempo o dinero, podría ser poco ético o simplemente imposible de realizar. Por lo que los científicos han optado por realizar simulaciones numéricas de tales experimentos para probar sus teorías. Un ejemplo de esto lo podemos observar en las simulaciones del clima mundial.

Por otra parte la industria, el comercio, las instituciones gubernamentales y de educación requieren de un manejo más eficiente de su volumen creciente de información. Un ejemplo muy claro de esto se puede observar en las máquinas de los bancos cuyas velocidades en las transacciones son críticas.

Debido a que día con día se pretende simular problemas mucho más complejos y a que la información a tiempo en el mundo de los negocios puede significar la diferencia, el poder de cómputo alcanzado después de cierto tiempo se hace insuficiente, por lo que es necesario incrementarlo. Para no caer en un estancamiento tecnológico, emerge la computación paralela y distribuida, que consiste en conectar varios procesadores y hacer que éstos trabajen en conjunto e intercambien información para realizar uno o más trabajos. De tal manera que con el consecuente aumento de cómputo se reduce su tiempo de ejecución[MOH96].

El cómputo paralelo se puede realizar de manera muy general, en máquinas paralelas que tienen varios procesadores interconectados internamente mediante alguna topología de red (supercomputadoras o computadoras paralelas-masivas) y en una red de estaciones de trabajo. La gran mayoría de las supercomputadoras modernas son del tipo MIMD (Múltiple Instruction Múltiple Data), es decir, realizan diferentes operaciones en diferentes grupos de datos.

Existen dos grandes clasificaciones para computadoras paralelas de acuerdo a la forma en que accesan la memoria: máquinas paralelas de memoria distribuida y máquina paralelas de memoria compartida. Las máquinas paralelas de memoria distribuida presentan la ventaja de la escalabilidad ya que se puede aumentar el número de procesadores sin afectar considerablemente el rendimiento de la computadora, razón por la cual muchos fabricante prefieren este diseño para la realización de sus máquinas[QUI94]. Debido a que las máquinas paralelas – masivas son muy costosas, realizar cómputo paralelo con estas es menos factible, ya que resulta caro para la mayoría de los usuarios.

Con la popularización de las redes de área local, la mayoría de las instituciones educativas cuentan con redes de estaciones de trabajo Unix heterogéneas, es decir, redes conformadas por estaciones de trabajo bajo plataformas diferentes como por ejemplo: Sparc, Intel, Power con sistemas operativos como Solaris, Linux y AIX respectivamente, por mencionar alguna, interconectadas bajo la suite de comunicación TCP/IP. Con esta tecnología prácticamente a la mano es posible crear meta–computadoras. Una meta–computadora son varias computadoras localizadas a grandes distancias interconectadas. La meta–computadora se puede utilizar como una computadora paralela de memoria distribuida. Es claro observar que realizar cómputo paralelo en este ambiente es muy factible y menos costoso.

La tecnología de software aún no está lo suficientemente desarrollada para el eficiente y fácil aprovechamiento del paralelismo disponible en una computadora paralela[KUC94]. La programación todavía no es muy fácil, las técnicas que trabajan para escribir programas seriales no funcionan cuando se intenta escribir un programa paralelo y dado que todavía no es posible la explotación al máximo del paralelismo disponible en alguna máquina, el esfuerzo empleado para lograr paralelismo muchas veces no es recompensado con el desempeño obtenido (en términos de tiempos de respuesta, número de trabajos procesados en un tiempo dado o en confiabilidad). Finalmente los diseñadores de programas paralelos deben estar muy familiarizados tanto con el hardware como con el software y la interacción entre ellos, a sí como con el paralelismo inherente en la aplicación para obtener el máximo provecho de su plataforma[HES98].

Existen herramientas de software orientadas a facilitar la explotación del paralelismo. Estas herramientas se pueden clasificar en tres grandes categorías: Bibliotecas para el pase de mensajes, bibliotecas de algoritmos paralelos, y compiladores para el computo paralelo. En la categoría de bibliotecas destacan PVM (Parallel Virtual Machine) y MPI (Massage Passing Interface). En cuanto a bibliotecas de algoritmos paralelos algunos ejemplos son SCALAPACK y BLOCKSOLVE. En el área de compiladores se encuentra una variedad de trabajos que abarcan desde extensiones a lenguajes para la ejecución paralela hasta paralelizadores automáticos de código. Algunos ejemplos de leguajes / compiladores son High Performance Fortran, High performance C++, Sisal, Linda entre otros.

Con todo este adelanto tecnológico en el que es posible paralelizar una aplicación usando tales herramientas surgen nuevos retos tan desafiantes como el desarrollo de técnicas efectivas para la distribución de las tareas de un trabajo entre los procesadores o cómo distribuir las tareas entre los procesadores de tal manera que se minimice el tiempo de ejecución, los retardos en la comunicación, y/o maximice la utilización de los recursos en un ambiente heterogéneo de estaciones de trabajo[SHI95]. Es decir, se pretende reducir los tiempos de ejecución de una aplicación paralelizada maximizando la utilización de los recursos disponibles en el ambiente.

2.1 Clasificación de las Computadoras de Acuerdo a su Arquitectura.

La arquitectura de las computadoras se clasifica de acuerdo a su flujo de datos e instrucciones, según Flynn, donde un flujo de instrucciones es una secuencia de instrucciones ejecutadas por una computadora; Un flujo de datos es un conjunto de datos con los cuales trabaja un flujo de instrucciones. Bajo este punto de vista surgen cuatro tipos de computadoras:

- SISD (Single Instruction, Single Data). Las computadoras de este tipo pueden ejecutar una sola instrucción en una unidad de tiempo.

- SIMD (Single Instruction, Multiple Data). Las computadoras de este tipo algunas veces también conocidas como máquinas de arreglos de procesadores, tienen la capacidad de que en cualquier unidad de tiempo, una sola operación esta en el mismo estado de ejecución en las múltiples unidades de procesamiento, cada una manipulando datos diferentes.

- MIMD (Múltiple Instruction, Múltiple data). Las computadoras de este tipo tienen un grupo de procesadores independientes, cada uno con su propio contador de programas y datos. Un sistema distribuido que se usa para cómputo paralelo es MIMD. Estas computadoras son de dos tipos de memoria compartida y de memoria distribuida:

  - En las computadoras paralelas bajo ésta arquitectura con memoria compartida o multiprocesador todos los procesadores acceden a una memoria común. Los programas desarrollados para multicomputadoras pueden también ejecutarse eficientemente en máquinas multiprocesador, debido a que la memoria compartida permite la implementación eficiente del paso de mensajes.

  - En las computadoras paralelas con memoria distribuida o multicomputador con esta arquitectura cada procesador puede ejecutar un conjunto diferente de instrucciones con sus propios datos locales, y cada procesador tiene asociado un segmento de memoria propio.

2.2 Clasificación de las Computadoras Paralelas Según la Forma que Acceden la Memoria sus Procesadores

- Computadoras con memoria compartida. En esta cada procesador tiene acceso directo a la memoria principal a través de una red. En las computadoras de memoria virtual compartida, cada procesador tiene memoria local pero puede acceder la memoria global.

- Computadoras con memoria distribuida. Cada procesador accede su memoria local para cálculos. Los procesadores se comunican entre sí mediante una red. Si se requiere que un procesador obtenga datos almacenados en la memoria local de otro procesador es necesario alguna herramienta adicional como por ejemplo las de paso de mensajes.

2.3 Balanceo de Carga.

El problema del balanceo de carga es como distribuir los procesos entre todos los procesadores disponibles de tal manera que se logren las metas de desempeño deseadas, como por ejemplo: la minimización del tiempo de ejecución, la minimización de los retardos de comunicación, y/o maximización de la utilización de los recursos.

Las técnicas de balanceo de carga pueden ser clasificadas de manera general en local y global.

- El balanceo de carga local es el realizado, por ejemplo, por el sistema operativo de un procesador y consiste en la asignación a cada procesador de un proceso por un tiempo definido.

- El balanceo de carga global, por otra parte, es el proceso de decidir dónde se ejecutará un proceso en un sistema multiprocesador. El balanceo global puede ser controlado por una simple autoridad central o bien distribuido entre los procesadores. El balanceo de carga global puede ser clasificado en dos grandes grupos: Balanceo de carga estático y balanceo de carga dinámico.

    - En balanceo estático la asignación de las tareas a los procesadores es hecha antes de que la ejecución del programa comience, la información del tiempo de ejecución de las tareas y de la disponibilidad y capacidad de los recursos de procesamiento se supone conocida en tiempo de compilación. Una tarea siempre es ejecutada en el procesador que es asignada. La meta del balanceo de carga estático es minimizar el tiempo de ejecución de programa,

minimizando los retardos de la comunicación. En resumen el balanceo de carga estático pretende:

- Predecir el comportamiento del programa en tiempo de compilación (esto es, estimados del tiempo de ejecución y retardos de la comunicación).

- Realizar un particionamiento de las tareas en una granularidad tal que se reduzca el costo de la comunicación.

- Asignar procesos a los procesadores.

o El balanceo de carga dinámico está basado en la redistribución de las tareas entre los procesadores durante el tiempo de ejecución. Esta redistribución es realizada transfiriendo tareas de un procesador cargado a un procesador con menos carga, con la idea de mejorar el desempeño del programa de aplicación. Un algoritmo típico de balanceo de carga dinámico es definido por tres políticas inherentes:

- Políticas de información, las cuales especifican la cantidad de información disponible para el que realiza la decisión de asignar un trabajo.

- Políticas de transferencia, las cuales determinan las condiciones bajo las cuales un trabajo debería ser transferido, esto es, la carga del actual "host" y el tamaño de trabajo en consideración.

- Políticas de asignación, las cuales identifican los elementos de procesamiento hacia los cuales un trabajo debería ser transferido.

Las operaciones de balanceo de carga pueden ser centralizadas en un simple procesador o distribuidas en todos los procesadores que participen en los procesos de balanceo de carga. Muchas políticas combinadas podrían existir.

2.4 Descomposición de Matrices Mediante la Técnica LU.

La descomposición *LU* consiste en decomponer la *matriz A* en dos matrices, la matriz triangular superior y la matriz triangular inferior, para ello asumiremos que A es una matriz de orden (nxn) no singular. Debemos encontrar una factorización $A=LU$. Esto se logra mediante eliminación Gaussiana. El algoritmo para implementar esta estrategia es recursivo. Se desea construir una descomposición LU de una *matriz A* no singular de orden

(nxn), Si $n=1$, el proceso termina dado que podemos seleccionar $L=I1$ y $U=A$, para $n>1$, partimos a A en cuatro partes:

$$
A = \left[\begin{array}{c|cccc}
a_{1,1} & a_{1,2} & \cdots\cdots\cdots & a_{1,n} \\
\hline
a_{2,1} & a_{2,2} & \cdots\cdots\cdots & a_{2,n} \\
\cdot & \cdot & \ddots & \\
\cdot & \cdot & & \cdots \\
\cdot & \cdot & & \\
\cdot & \cdot & & \\
a_{n,1} & a_{n,2} & \cdots\cdots\cdots & a_{n,n}
\end{array}\right]
$$

$$
= \begin{bmatrix} a_{1,1} & w^T \\ V & A' \end{bmatrix}
$$

Donde V es un vector columna de $(n-1)$ elementos, w^T es un vector renglón de $(n-1)$ elementos, y A' es una matriz de orden $((n-1)x(n-1))$. Posteriormente utilizando álgebra matricial, podemos factorizar A como:

$$
A = \begin{bmatrix} a_{1,1} & w^T \\ V & A' \end{bmatrix}
$$

$$
= \begin{bmatrix} 1 & 0 \\ V / a_{1,1} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{1,1} & w^T \\ 0 & A' - Vw^T / a_{1,1} \end{bmatrix}
$$

Los $0's$ en las matrices de la factorización son vectores renglón y columna, respectivamente, de tamaño $(n-1)$. El término $Vw^T / a_{1,1}$ es una matriz de orden $((n-1)x(n-1))$. La matriz $A' - V\, w^T / \, a_{1,1}$, se le conoce como el complemento de Schur de A con respecto a $a_{1,1}$.

Ahora recursivamente, encontramos una descomposición LU para el complemento de Schur. Digamos que:

$$
A' - Vw^T / a_{1,1} = L'U',
$$

donde *L'* es una matriz triangular inferior unitaria y *U'* es una triangular superior. Utilizando álgebra matricial, tenemos:

$$A = \begin{bmatrix} 1 & 0 \\ V/a_{1,1} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{1,1} & w^T \\ 0 & A'-Vw^T/a_{1,1} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 \\ V/a_{1,1} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{1,1} & w^T \\ 0 & L'U' \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 \\ V/a_{1,1} & L' \end{bmatrix} \begin{bmatrix} a_{1,1} & w^T \\ 0 & U' \end{bmatrix}$$

$$= LU$$

Con lo que obtenemos la descomposición *LU* (dado que *L'* es triangular inferior unitaria, también los es *L*, y dado que *U'* es triangular superior, también los es *U*).

Está claro que si $a_{1,1} = 0$ este método no funciona por la división por 0. Tampoco funciona si el elemento superior izquierdo del complemento Schur $A'-Vw^T/a_{1,1}$ es 0 porque lo utilizamos en una división en el siguiente paso de la recursión. Los elementos por los que dividimos en la descomposición *LU* se llaman pivotes y ocupan la diagonal principal de la *matriz U*.

Una clase importante de matrices para las cuales la descomposición *LU* siempre trabaja correctamente es la clase de matrices simétricas definidas positivas. Tales matrices no requieren pivoteo.

2.5 Trabajos Relacionados.

Con la proliferación de las redes de estaciones de trabajo Unix en las universidades y a los grandes avances en tecnologías de redes se hizo factible que se empiece a investigar sobre la distribución de procesos en tales ambientes heterogéneos y más aún que se empiece a investigar sobre técnicas para asignar tareas a los procesadores proporcionales a

su desempeño, minimizando el tiempo de ejecución de un programa. Es de esta manera como podemos encontrar algunos avances que se describen a continuación:

Zaki, Wei y Parthasarathy [ZAKI, et al., 1996], examinan la conducta de las estrategias de balanceo de carga centralizada contra distribuida y global contra local. Muestran que diferentes esquemas son mejores para diversos programas variando los parámetros del sistema y de los programas. Más aún, en este artículo se comenta: "un esquema de balanceo de carga adaptado es esencial para un buen desempeño". También presentan un modelo híbrido (se realiza en tiempo de compilación y tiempo de ejecución) y un proceso de decisión el cual selecciona el mejor esquema, junto con la generación automática de código paralelo con llamadas librerías para balanceo de carga.

Maheshwari, El-Rewini y Shriver [MAHESHWARI, et al., 1996], presentan un algoritmo de balanceo de carga basado en prioridades para ambientes de computación paralela. El algoritmo determina las precedencias de las tareas del grafo dinámicamente en tiempo de ejecución y asigna las prioridades correspondientes a los procesos para resolver las dependencias.

Por su parte Diekmann, Monien y Preis [DIEKMANN, et al., 1998], dan una clasificación de los diferentes problemas de balanceo de carga basada en aplicaciones características. Para el caso de aplicaciones fuera del campo de ciencias de la computación, describen los métodos más usados en detalle.

Estos trabajos sentaron las bases para la implementación paralela del algoritmo LU con balanceo dinámico en el ambiente heterogéneo de estaciones de trabajo.

Capítulo 3

Paralelización de Descomposición LU

En este capítulo se detalla la implementación paralela del algoritmo distribuido para la descomposición de matrices LU, presentado las ideas principales sobre las cuales se basó el desarrollo, y los detalles de implementación con mayor relevancia.

3.1 Detalles de la Implantación del Algoritmo Paralelización de LU

Por paralelización entenderemos repartición de trabajo que deba realizar una aplicación entre todos los procesadores disponibles para realizar dicho trabajo de manera concurrente, con la finalidad de que resulte en menor tiempo de ejecución con respecto de la solución secuencial en el procesador más rápido.

Particularmente en nuestro caso la aplicación será descomposición de matrices mediante la técnica LU. LU recibe una matriz como datos de entrada y como resultado genera una matriz del mismo orden, que contiene la matriz triangular inferior y la superior. El algoritmo que soluciona LU tiene una alta dependencia de datos, en el sentido de que para realizar una iteración es necesario utilizar cálculos y actualizaciones hechas en la iteración anterior.

El algoritmo que se utilizará para la parelelización LU corresponde a la clase de algoritmos distribuidos. En esta clase de algoritmos todos los procesadores realizan las operaciones necesarias para determinar la mejor forma de distribuir los datos, así como también todos participan en el envío y recepción de datos.

La paralelización de LU consistirá en dividir, en cada iteración, la matriz A de entrada en segmentos de igual tamaño y asignarlos a los procesadores disponibles para la ejecución concurrente de las operaciones necesarias para dividir A en las matrices inferior y superior.

Debido a que este método para paralelizar LU no toma en cuenta la carga que pueda existir en un momento dado en los procesadores, así como tampoco considera las congestiones en la red, dará excelentes resultados en un sistema homogéneo, sin embargo en un sistema heterogéneo, en el cual los procesadores tienen distintas capacidades y las cargas de trabajo son también diferentes, el algoritmo no se comportará adecuadamente debido a que la paralelización ha sido realizada tomando como base un sistema homogéneo.

3.2 Descripción del Algoritmo Paralelo LU.

En esta sección se describe el algoritmo paralelo de LU a través de sus pasos más importantes como son: el particionamiento de datos y la asignación y ejecución de tareas.

3.2.1 Particionamiento de Datos

La importancia de particionar la matriz A radica en que al realizar esta partición, también se divide el trabajo, debido a que para solucionar LU se realizan operaciones sobre cada elemento de la matriz A.

El enfoque que se seguirá para la paralelización LU, será particionamiento por renglones de la matriz de entrada en partes iguales, según el número de procesadores. En la Figura 3.1 se muestra como la matriz A es fragmentada en partes iguales, de tal manera que queda dividida lógicamente en segmentos de igual tamaño y todos los segmentos tienen igual número de filas contiguas, de esta forma cada segmento puede ser localizado a partir de la posición de su primera y última fila que ocupa dentro de la matriz A.

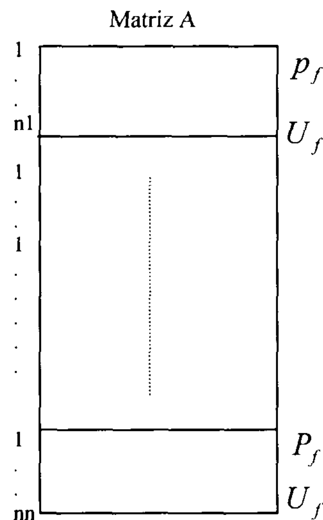


Figura 3.1.- Particionamiento de datos en la paralelización de LU.

3.2.2 Asignación y Ejecución de Tareas.

La idea en la que esta basado el algoritmo paralelo LU para realizar la distribución de datos es simplemente conocer los datos que cada procesador tiene y los datos que cada uno de ellos necesita para realizar el intercambio de datos, de tal manera que al final cada procesador cuente con la información que requiere para realizar los cálculos propios del algoritmo.

La Figura 3.2 muestra el procedimiento que sigue este algoritmo para realizar dicho intercambio de datos. Este procedimiento inicia en la iteración 1 donde se particiona por primera vez la matriz A y se guarda la información de la primer y última fila de cada segmento en las variables Pf (primer fila) y Uf (última fila) al término de la ejecución de los cálculos en esta iteración, se mantiene guardada la información de primera y última fila de cada segmento en las variables Pf_ant (primer fila de la iteración anterior) y Uf_ant (última fila de la iteración anterior). A partir de la iteración 2 y hasta terminar, el algoritmo se comporta de igual forma; particiona la matriz A en partes iguales, ya que ha disminuido su dimensión y se pretende mantener las mismas cargas para todos los procesadores, almacenando la información de la primera y última fila de cada segmento de A para cada procesador en las variables Pf y Uf respectivamente, inmediato después cada procesador compara estas últimas variables con las variables Pf_ant y Uf_ant, para determinar cuales filas deberán enviar o recibir, de tal manera que cada procesador únicamente reciba las filas que completa el segmento que requiere, ya que como se puede observar en la Figura 3.2 la diferencia entre los segmentos que cada procesador requiere en la iteración 2 difiere de los segmento que le fue asignado en la iteración 1 en solo de algunas filas. Realizar el particionamiento de esta forma es óptimo pues únicamente se transfiere los datos necesarios y con ello se genera menos tráfico en la red.

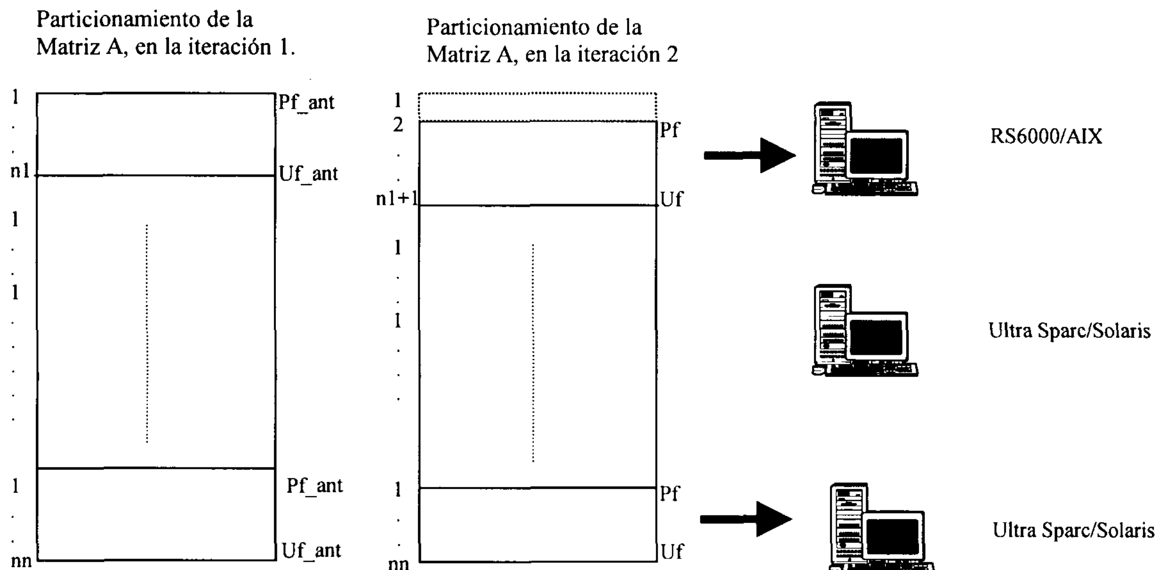


Figura 3.2.- Asignación de segmentos de la matriz A a cada estación de trabajo

Una vez asignado la carga de trabajo según el número de procesadores se procederá a la ejecución de las operaciones, este proceso continuará hasta que la dimensión de la matriz sea uno. En la Figura 3.3 se resume este proceso.

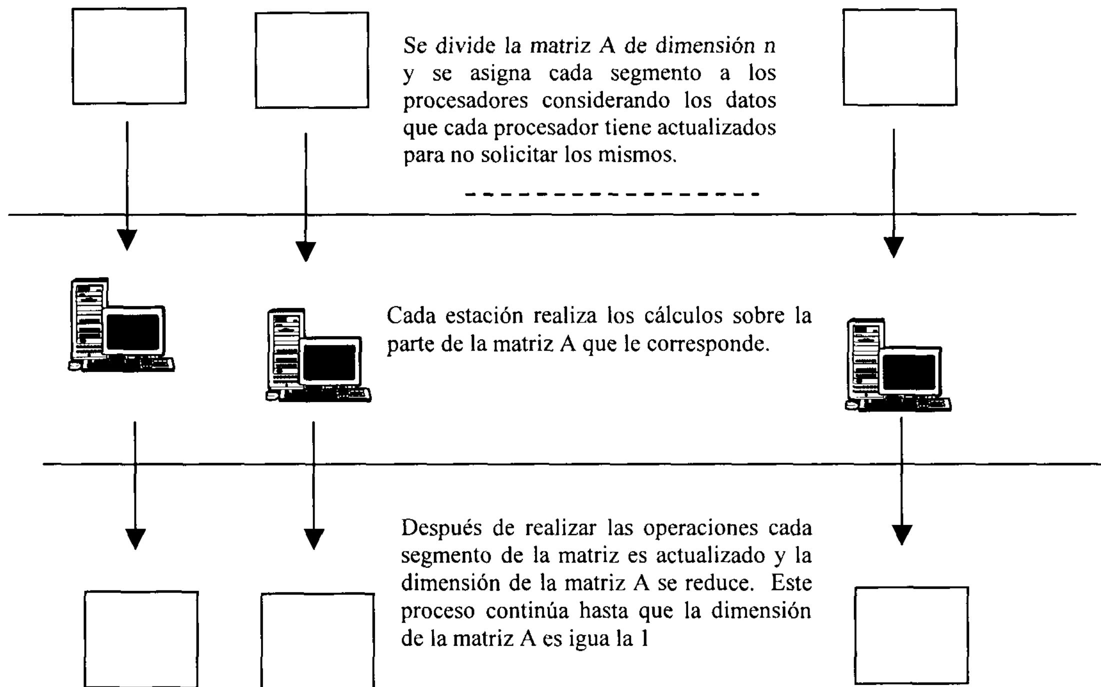


Figura 3.3.-Paralelización del algoritmo LU.

3.3 Comportamiento del Algoritmo Paralelo LU.

A continuación presentamos un ejemplo para mostrar los pasos del algoritmo. El algoritmo paralelo LU comienza particionando la matriz A en partes iguales según el número de procesadores. Para hacer más sencilla la explicación asumiremos que únicamente tenemos tres procesadores como se observa en la Figura 3.4, por lo que tendremos tres segmentos de la matriz A y una matriz LU para cada procesador.

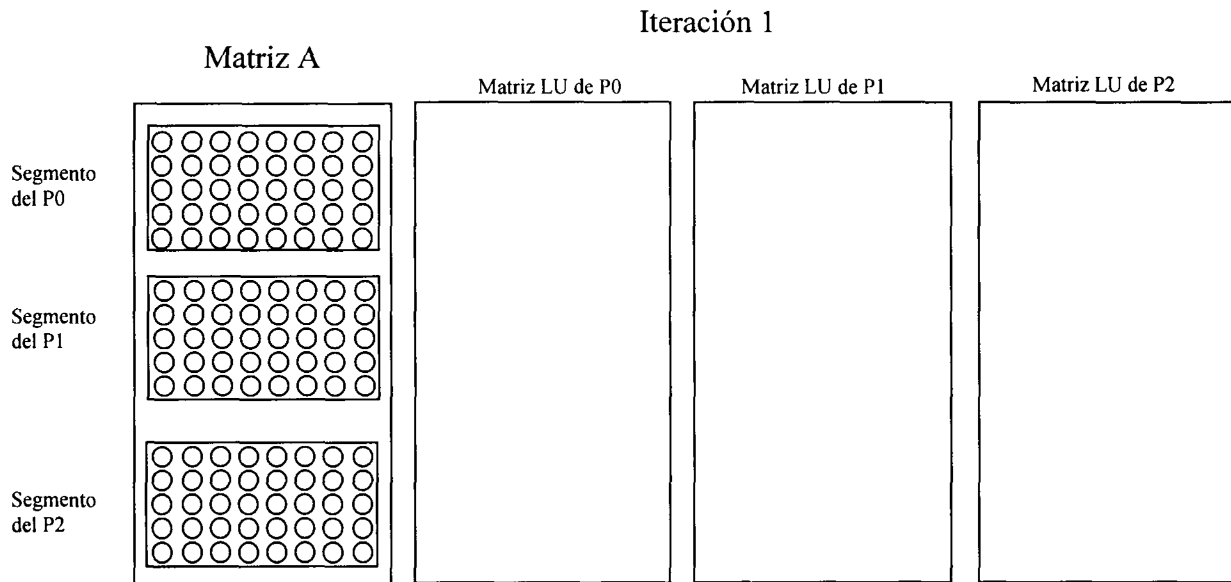


Figura 3.4.- Particionamiento de la matriz A, en la iteración 1.

Una vez asignados los segmentos de la matriz A a cada procesador, el algoritmo inicia la ejecución de las operaciones. Cada procesador calcula los datos del segmento que le corresponde de la matriz LU, basados en las operaciones realizadas sobre el segmento de la matriz A que le corresponde. La Figura 3.5 muestra lo que sucede en la iteración 1. El procesador P0 lee los datos del "segmento P0", realiza cálculos sobre ellos y escribe los datos de la primera fila de la matriz LU y los primero 5 elementos de la primera columna de LU, de manera concurrente el procesador P1 lee los datos del segmento de la matriz A que le corresponde, realiza cálculos sobre ellos y escribe los datos de la primera fila de la matriz LU y los elementos 6 al 10 de la primera columna de LU, de manera análoga se sigue lo mismo para el procesador P3.

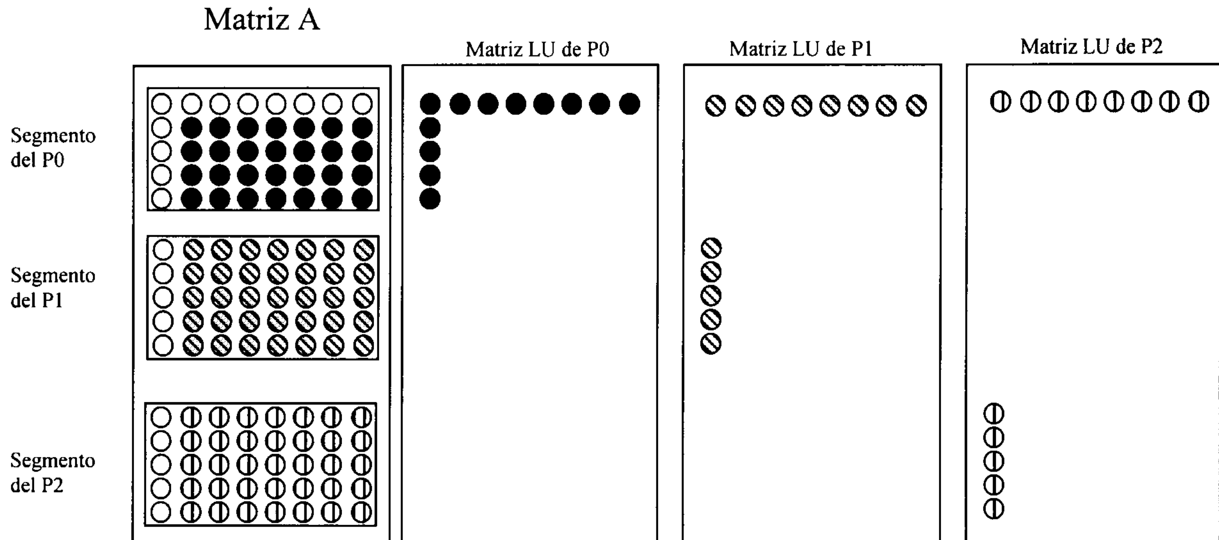


Figura 3.5.- Elementos actualizados de las matrices A y LU, reflejados a través del coloreo.

Debido a que la dimensión de la matriz A ha disminuido y se pretende mantener siempre balanceado el tamaño de los segmentos de la matriz A que cada procesador tiene el algoritmo vuelve a particionar la matriz A en partes iguales como se muestra en la Figura 3.6

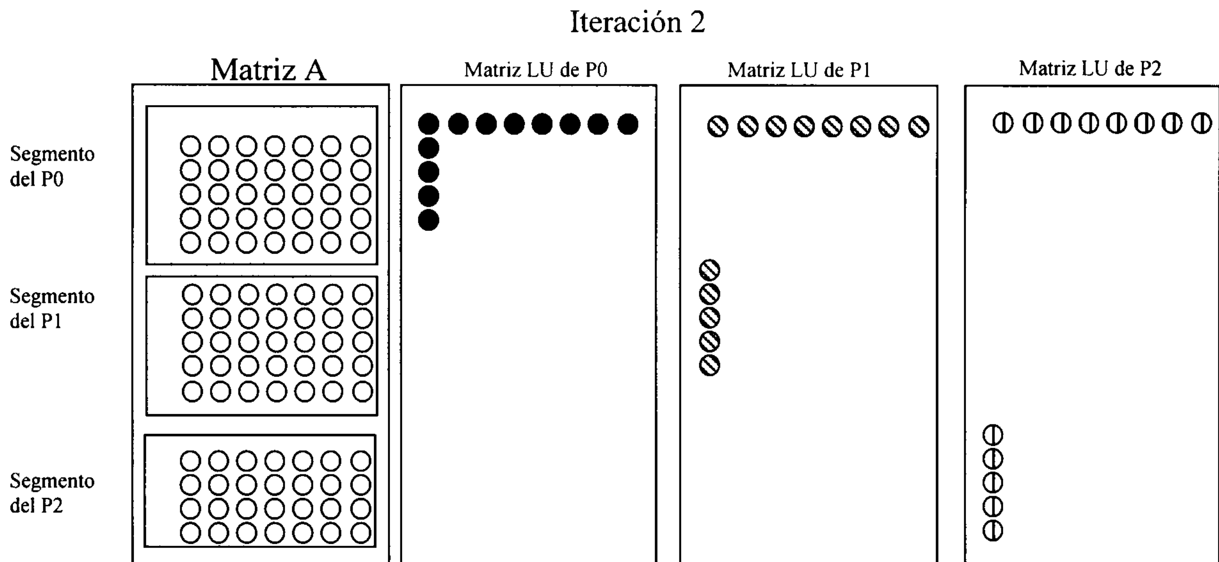


Figura 3.6.- Particionamiento de la matriz A, en la iteración 2.

La Figura 3.7 muestra lo que sucede en la iteración 2. El procesador P0 realiza cálculos con los datos del segmento de la matriz A que le corresponde y con base en estos obtiene los valores que corresponde a la matriz LU, como se observa en la "matriz LU de P0", es decir, completa la segunda fila de la "matriz LU de P0" y completa la segunda columna en sus elementos del 2 al 6. El procesador P1, concurrentemente con P0 toma los

datos del segmento de la matriz A que le corresponde, realiza cálculos con ellos y coloca los resultados en la "matriz LU de P1" que le corresponde, es decir llena la fila 2 en sus elementos del 2 al 8 y la columna 2 los elementos 7 al 11. De manera análoga el procesador P3 realiza la misma secuencia de pasos que los procesares anteriores y de manera concurrente.

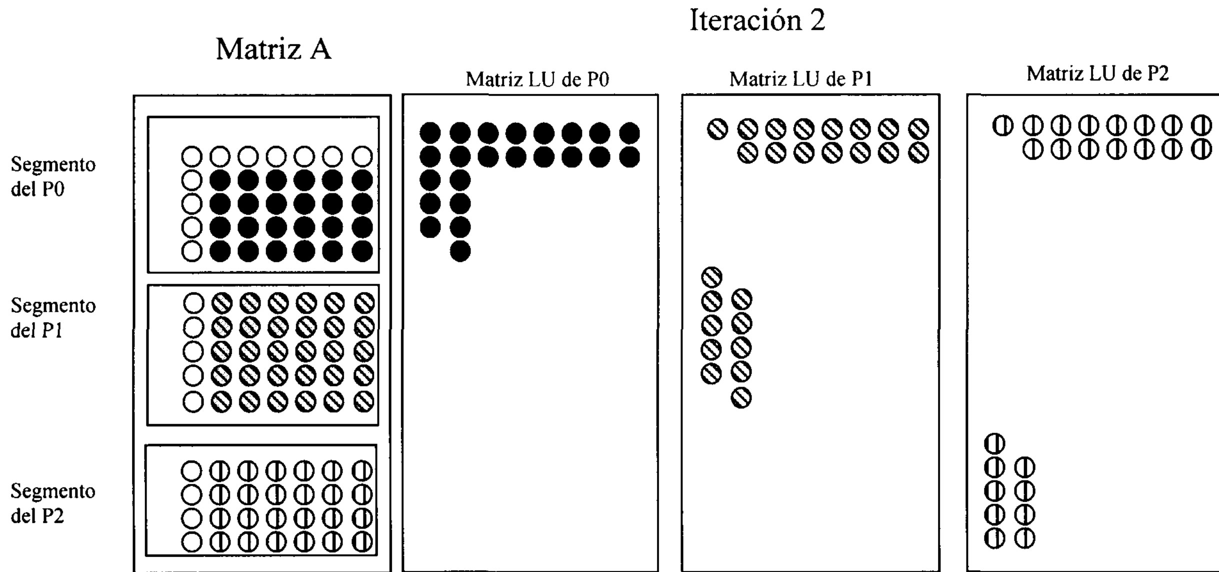


Figura 3.7.- Actualización de las matrices A y LU, reflejadas a través del coloreo.

En las iteraciones restantes y hasta que la dimensión de la matriz A se reduzca a 1, el algoritmo de paralelización LU continuará realizando lo mismo que en las iteraciones 1 y 2, particionar A, calcular y actualizar A y LU. Se observa también en la Figura 3.7 que conforme la dimensión de la matriz A se reduce en cada iteración, los procesadores llenaran diferentes partes de la matriz LU. Como ejemplo se observa el caso del procesador P1, en la iteración 1 llenó los elementos 1 al 5 de la columna 1 y en la iteración 2, los elementos del 2 al 6 y los mismo se sigue para el resto de los procesadores.

En este capítulo se hizo una descripción del algoritmo paralelo de LU, presentando los conceptos y los detalles más relevantes para su implementación. Este algoritmo sirvió de base para el desarrollo del algoritmo paralelo balanceado para ambientes heterogéneos de LU cuyos detalles se describen en el siguiente capítulo.

Capítulo 4

Balanceo Dinámico de LU

En este capítulo se detalla la implementación paralela balanceada dinámicamente del algoritmo de descomposición de matrices LU presentado las ideas principales sobre las cuales se basó el desarrollo, y los detalles de implementación con mayor relevancia.

4.1 Diagrama Conceptual del Algoritmo Paralelo LU.

En un ambiente de red de estaciones trabajo heterogéneas las condiciones de carga de los procesadores y el tráfico en la red cambian según el tipo de aplicaciones que se ejecutan, los patrones de uso de los usuarios y las características propias de las estaciones y los dispositivos de red.

El reto en este ambiente es ejecutar un algoritmo que solucione LU de manera distribuida, de tal manera que maximice la utilización de las estaciones de trabajo, con el objeto de reducir el tiempo de ejecución de dicho algoritmo con respecto del tiempo de ejecución de la solución secuencial en el procesador más rápido y con el tiempo de ejecución de la solución paralela.

Este reto lo enfrentamos utilizando balanceo dinámico, en el algoritmo paralelo de LU, que consiste en reasignar las cargas de trabajo, en tiempo de ejecución, entre los procesadores según vayan cambiando sus condiciones y las condiciones en los medios de transmisión.

Para introducir el comportamiento del algoritmo paralelo de LU con balanceo dinámico se presenta un diagrama conceptual, Figura 4.1, que se basa en cuatro pasos generales que son: particionamiento de la matriz A, intercambio de datos entre los procesadores, ejecución de operaciones y medición de los tiempos de comunicación, procesamiento y overhead de balanceo.

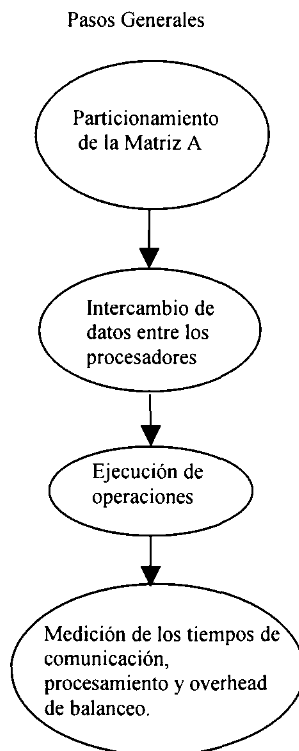


Figura 4.1.-Diagrama conceptual del algoritmo paralelo balanceado de LU.

A continuación se explicará cada uno de los pasos:

1.- Particionamiento de la matriz A

El particionamiento por filas de la matriz A determina la cantidad de trabajo que se asigna a los procesadores, ya que LU consiste en realizar operaciones sobre cada elemento de dicha matriz, y de aquí la importancia de este paso.

La idea del particionamiento es encontrar la cantidad de trabajo que cada procesador puede realizar de tal manera que todos los procesadores terminen al mismo tiempo.

Para esclarecer esta idea y comprender como se aplica en la solución de LU, se asume que la *matriz A* es de dimensión *n* y que el número de procesadores es *Npe*, entonces el algoritmo que soluciona LU particiona la *matriz A n* veces, disminuyendo en cada vez en uno su dimensión, en *n/Npe* segmentos de tamaño diferentes, es decir, se tiene *n* iteraciones y en cada iteración se particionar la *matriz A* en *n/Npe* segmentos, donde cada segmento esta formado por un conjunto de filas de *A*. La idea es encontrar el tamaño adecuado de cada segmento de la *matriz A*, según la capacidad de procesamiento de las estaciones de trabajo involucradas en el cálculo que le será asignado, de tal manera que todos los procesadores terminen de ejecutar su segmento al mismo tiempo con el objeto de pasar a la siguiente iteración al mimo tiempo.

Esta forma de determinar el particionamiento es óptima, debido a que de esta manera se garantiza que no hay retardos, en el sentido de que ningún procesador espera a algún otro y que todos avancen al mismo tiempo a la siguiente iteración. El cálculo de la capacidad de procesamiento de cada procesador se realiza estableciendo la relación de desempeño de ellos con respecto del procesador más rápido y de la configuración utilizada y en base a esta relación se determina la cantidad de trabajo, tamaño del segmento de la matriz A, que cada uno puede realizar al mismo tiempo. El cálculo de la capacidad de procesamiento se detalla en la Sección 4.3.1

2.- Intercambio de datos entre los procesadores.

Debido a que la dimensión de la matriz A disminuye en uno en cada iteración y que las cargas en las estaciones de trabajo y en la red cambian constantemente, el tamaño de los segmentos que corresponde a cada procesador cambia entre dos iteraciones, puesto que hay un balanceo. Es decir, al inicio de cada iteración, con excepción de la primera, cada procesador tiene un tamaño de segmento asignado, cierto conjunto de filas de A, y almacenado en memoria, con los cálculos del balanceo se establece el nuevo tamaño de dicho segmento por lo que la diferencia en este tamaño determina las filas que se deberán enviar y/o recibir para mantener el balanceo.

La forma óptima de realizar este intercambio de filas es que cada procesador conozca las filas que forman parte de cada segmento, de tal manera que reasigne las filas que sean necesarias a los procesadores que les haga falta. Es importante notar que con el balanceo los segmento de A pueden contener filas no contiguas ya que es óptimo solicitar filas a otra máquina que cuente con filas extra y enviar a la máquina que necesite filas sin importar a quien se le envía o solicita filas. Es por eso que cada procesador mantiene una tabla de los elementos de los segmentos que contiene la matriz A de cada estación de trabajo. La explicación respecto al intercambio de datos y la forma como se mantiene la dicha tabla se encuentra en la Sección 4.3.2.

3.- Ejecución de Operaciones.

En este paso se realizan las operaciones necesarias para convertir la matriz A en dos matrices la triangular inferior y superior.

4.- Medición de tiempos de comunicación, procesamiento y overhead de balanceo.

El conocimiento de las cargas en los procesadores y en la red es determinante para el balanceo óptimo, es decir, para el particionamiento óptimo de la matriz A. La idea que se sigue para conocer tales cargas es medir el tiempo que le toma a cada procesador realizar un mismo trabajo a partir de que se le envía la información necesaria para realizarlo. En LU esta idea se lleva a cabo midiendo en cada procesador el tiempo de envio/recepción de una fila, el tiempo de procesar una fila y el tiempo de realizar cálculos de balanceo. La suma de estos tres tiempos establece cuanto tiempo le toma a cada procesador realizar un trabajo, procesar una fila; un procesador demasiado cargado tardará mas que uno con poca carga.

Debido a que estos tiempos sirven de base a cada procesador para determinar la relación de desempeño, que varía durante la ejecución, entre las estaciones de trabajo involucradas en los cálculos y para calcular la cantidad de trabajo óptima que cada procesador debe realizar, paso 1, todos los procesadores mantienen la información de los tiempos en que cada procesador ejecuta una fila, la sincronización de esta información de tiempos ocurre según la frecuencia de balanceo y la predicción de carga, ambas determinada por experimentación. El calculo de la frecuencia de balanceo y la predicción de la carga son tratadas a detalle en la Seccíónes 5.4.1 y 5.4.2 respectivamente.

4.2 Detalles de la Implantación del Algoritmo de Balanceo Dinámico de LU

La implementación del balanceo de carga dinámico comienza en un estado inicial en el que la fragmentación de la matriz de entrada es equitativa para todas las estaciones de trabajo involucrados en la ejecución, esta consideración es debido a que no se conoce información de cargas de la red y de las cargas de trabajo en las máquinas.

En el momento en que todos las máquinas terminan con el trabajo asignado en la primera iteración, la aplicación entra en un estado de balanceo en el cual se conocen las cargas de trabajo actuales y el tráfico en la red, basándose en dicha información se hacen los particionamientos posteriores de la matriz de entrada hasta finalizarla la ejecución de la aplicación

Otra consideración importante es que la matriz de entrada, por las características muy particulares de la aplicación va disminuyendo su dimensión en uno en cada iteración, siendo éste otro factor que afecta el balanceo, por lo que para mantener la carga óptima en cada procesador este algoritmo realiza un balanceo de carga en cada iteración, aunque las cargas de los procesadores y en la red se mantengan iguales.

4.3 Descripción del Algoritmo Paralelo Balanceado de LU.

En esta sección se describe el algoritmo paralelo balanceado de LU a través de sus pasos más importantes como son: el particionamiento de datos y la asignación y ejecución de tareas.

4.3.1 Particionamiento de Datos

El particionamiento por filas de la matriz A, que genera segmentos de A, se realiza a través de dos etapas. La primera etapa, que únicamente se presenta en la primera

iteración, como se muestra en la Figura 4.2, pretende conocer el estado actual del sistema, es decir, se quiere conocer las cargas en los procesadores y en la red, para esto asigna segmentos de igual tamaño a cada estación de trabajo y cada una de ellas mide el tiempo que le toma recibir su respectivo segmento, de procesamiento del segmento, y de overhead de balanceo. Cada máquina suma estos tres tiempos y los divide entre el número de filas que forman el segmento para tener el tiempo que le tomaría realizar el trabajo de procesar una fila de A.

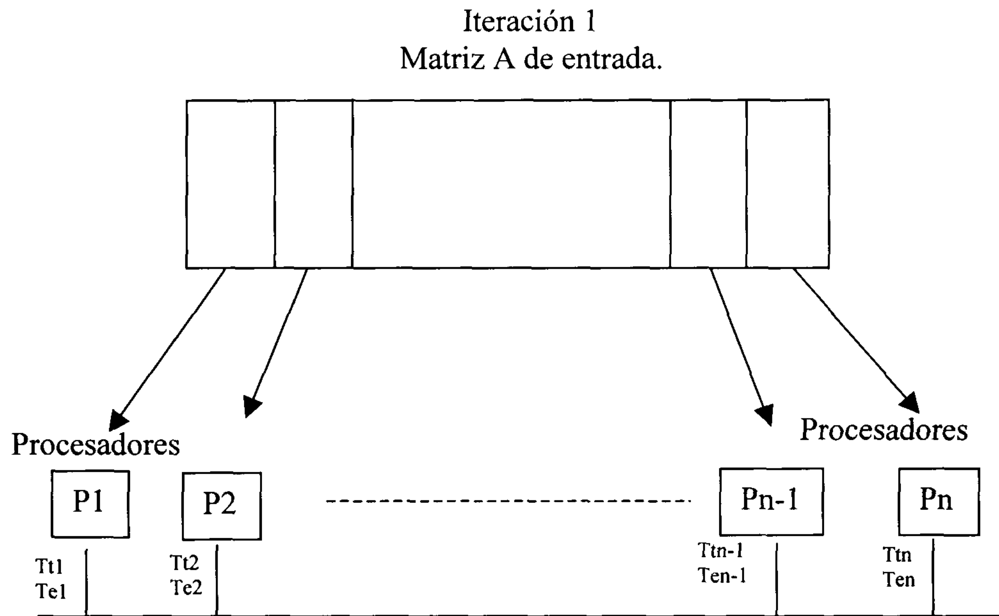
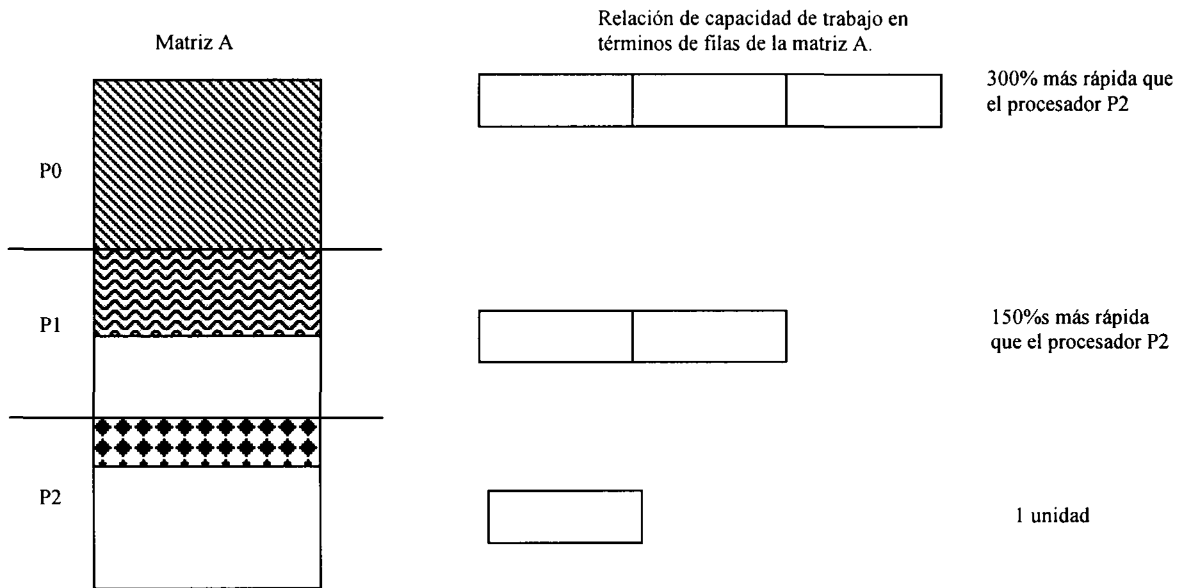


Figura 4.2.-Particionamiento de la matriz A

La segunda etapa en la división de datos inicia a partir de la iteración 2 y dura hasta que la dimensión de la matriz A es 1, se recuerda que el número de iteraciones es igual a la dimensión de la matriz A. A partir de esta segunda etapa el particionamiento de la matriz A se realiza basada en la información de los tiempos que le llevaría a cada estación de trabajo realizar el trabajo de procesar un fila de A. Cada máquina mantiene la información de tales tiempos del resto de ellas. La actualización de estos tiempos se realiza cada cierto número de iteraciones. A esta periodicidad de medición se le llama frecuencia de balanceo, cuyo valor óptimo estará determinada por experimentación (ver la sección 5.4.1).

En base a estos tiempos se establece la relación de desempeño de cada estación de trabajo con respecto de la máquina más rápida y basado en esta relación de desempeño se establece la cantidad de trabajo que puede ser realizado por cada máquina. Para esclarecer esta idea se presenta el ejemplo mostrado en la Figura 4.3, en el cual se asume que tenemos tres máquinas y que estamos en la iteración 1, por lo que la matriz A tiene tres segmentos iguales. Observemos también que cada uno de los segmentos esta coloreado a diferentes niveles para indicar la cantidad de procesamiento de cada segmento que se realiza en un cierto tiempo, esto es, el procesador P0 es el más rápido ya que terminó de ejecutar todas las operaciones que corresponde a su segmento, por esto el segmento esta completamente coloreado y es 300% más rápido que el procesador P2. El procesador P1 es el 150% mas

rápido que el procesador P2, esta coloreado la mitad su segmento. El procesador P2 es el mas lento y esta coloreado únicamente un tercio de su segmento. En este punto en el que conocemos la relación de velocidad entre los procesadores el siguiente paso es encontrar el trabajo total realizado por todos los procesadores, esto se obtiene sumando el trabajo que cada procesador realizó de manera relativa al procesador más rápido. Es decir, en muestro ejemplo, esto sería 3/1+3/2+3/3=5.5. Con esto se procede a encontrar que porcentaje realizó cada procesador del total de trabajo hecho, dividiendo el total entre cada elemento de la suma 3/5.5=0.55,1.5/5.5=0.27,1/5.5=0.18 y estos porcentajes, que llamaremos de trabajo a realizar, nos da precisamente el porcentaje de la matriz A que deberá ser asignado a cada procesador en la siguiente iteración.



Procesadores	Tiempos	Relación de Rapidez	Porcentaje de trabajo por realizar
P0	T1 = 1	3/1 = 3	3/5.5 = 0.55
P1	T2 = 2	3/2 = 1.5	1.5/5.5 = 0.27
P2	T3 = 3	3/3 = 1	1/5.5 = 0.18
	Trabajo Total realizado	5.5	

Figura 4.3.- Cálculo del particionamiento de la matriz A.

En este punto para conocer el número de filas de la matriz A que corresponderá a cada procesador lo único que resta es multiplicar cada porcentaje del trabajo por realizar de cada procesador con el número de filas de la matriz A.

4.3.2 Asignación y Ejecución de Tareas.

Para llevar a cabo la distribución de los datos y por ende la asignación de tareas a los procesadores, el algoritmo paralelización balanceado LU se basa en una tabla de los elementos que contienen los segmentos de la matriz A, dicha tabla en un momento dado indicará las filas que cada procesador debe enviar o recibir y en otro momento las filas que cada procesador tiene.

En la Figura 4.4 podemos observar la tabla de los elementos de los segmentos de la matriz A, cada elemento de esta es un arreglo de cuatro elementos, el primero, Pe, define el procesador al que pertenece el segmento de A, el segundo, E, define si el segmento debe ser enviado o recibido al procesador Pe, dependiendo de su valor (0 = recibir, 1 = enviar), el tercero, Pf, corresponde al valor de la primera fila del segmento de la matriz A que corresponde al procesador Pe y el cuarto elemento, Uf, es el valor de la última fila del segmento de la matriz A que corresponde al procesador Pe.

El número de filas de la tabla de los elementos que contienen los segmentos de la matriz A esta determinado por el número de procesadores que participan en la ejecución del algoritmo, el número de columnas de esta tabla es diferente para cada procesador ya que depende del número de filas que le sea asignado en un momento dado, que esta directamente relacionado con las condiciones de carga de dicho procesador y las condiciones de red para llegar al mismo.

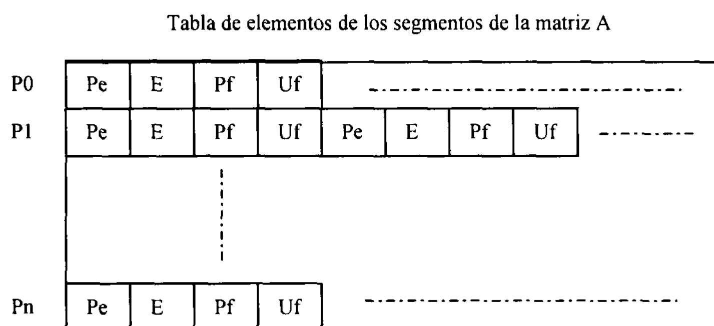


Figura 4.4.- Definición de la tabla de elementos de los segmentos de la matriz A.

Para esclarecer el funcionamiento de la tabla, asumiremos que tenemos 3 procesadores P0, P1 y P2 como se muestra en la Figura 4.5, y que estamos iniciando los cálculos de una iteración diferente a la primera digamos la iteración 2, entonces cada procesador conoce el porcentaje de filas que le corresponde de la matriz A, de dimensión 9, y cada procesador tiene en su tabla la información de las filas con que cuenta en este momento.

Tabla de elementos de los segmentos de la matriz A

	Pe	E	Pf	Uf
P0	0		1	3
P1	1		4	6
P2	2		7	9

Figura 4.5.- Instancia de la tabla de elementos de los segmentos de la matriz A, de los datos almacenados.

Con este porcentaje de filas cada procesador calcula el número de filas real que le corresponde y revisa su tabla para actualizarla determinando las filas que debe enviar o recibir.

Supongamos en este caso que de acuerdo al porcentaje de filas, al procesador P0 le corresponden 4 filas, al procesador P1 2 filas y al procesador P2 2 filas, entonces al procesador P0 le hacen falta 2 filas, al procesador P1 le sobra un fila y al procesador p2 le sobra un fila también, entonces en este momento cada procesador realiza actualizaciones a la tabla quedando como se muestra en la Figura 4.6.

Observemos en la Figura 4.6 que el procesador P0 recibirá la fila 6 del procesador 1 y la fila 9 del procesador p2. Esta forma de redistribuir las filas fragmenta los segmentos que corresponde a cada procesador en el sentido de que las filas de los segmentos que corresponde a cada procesador pudieran no estar contiguas, pero se gana eficiencia en cuanto al movimiento de información, pues únicamente en esta caso se están moviendo 2 filas, mantener los segmentos contiguos puede ocasionar mayor movimiento de filas, por ejemplo, supongamos que las cuatro filas que requiere el procesador P0 son 2, 3, 4 y 5 y que el procesador P1 requiere 6 y 7 y el procesador P2 8 y 9, notemos que se requiere un movimiento de filas mas pues la fila 4 y 5 que tenia el procesador P1 deben ser enviadas a P0, y P1 requiere la fila 7 que tiene el procesador P2, es decir P1 únicamente necesita enviar una fila y en vez de realizar esto envió dos filas y recibió una, esto es más ineficiente. Por otra parte la tabla pudiera crecer demasiado en cuanto al número de columnas por demasiada fragmentación ocasionada por los cambios continuos en las cargas de los procesadores y de la red y como consecuencia demandar memoria de manera considerable para matrices muy grandes.

Tabla de elementos de los segmentos de la matriz A

	Pe	E	Pf	Uf	Pe	E	Pf	Uf	Pe	E	Pf	Uf
P0	0		2	3	1	0	6	6	2	0	9	9
P1	1		4	5	0	1	6	6				
P2	2		7	8	0	1	9	9				

Figura 4.6.- Instancia de la tabla de elementos de los segmentos de la matriz A, con los datos de las filas que deberán ser enviadas y recibidas.

Una vez que las filas han sido enviadas a sus respectivos procesadores se actualiza la tabla para reflejar los datos con los que realmente cuenta cada procesador, como se observa en la Figura 4.7, el procesador P0 tiene las filas 2, 3, 6 y 9, el procesador P1 tiene las filas 4 y 5 y el procesador P2 tiene las filas 7 y 8.

Tabla de elementos de los segmentos de la matriz A

	Pe	E	Pf	Uf	Pe	E	Pf	Uf	Pe	E	Pf	Uf
P0	0		2	3	0		6	6	0		9	9
P1	1		4	5								
P2	2		7	8								

Figura 4.7.- Instancia de la tabla de elementos de los segmentos de la matriz A, después del intercambio de filas entre los procesadores.

Una vez asignado la carga de trabajo según el número de procesadores, sus cargas y las congestiones en la red cada procesador procederá a la ejecución de las operaciones, este proceso continuará hasta que la dimensión de la matriz sea uno. En la Figura 4.8 resume este proceso.

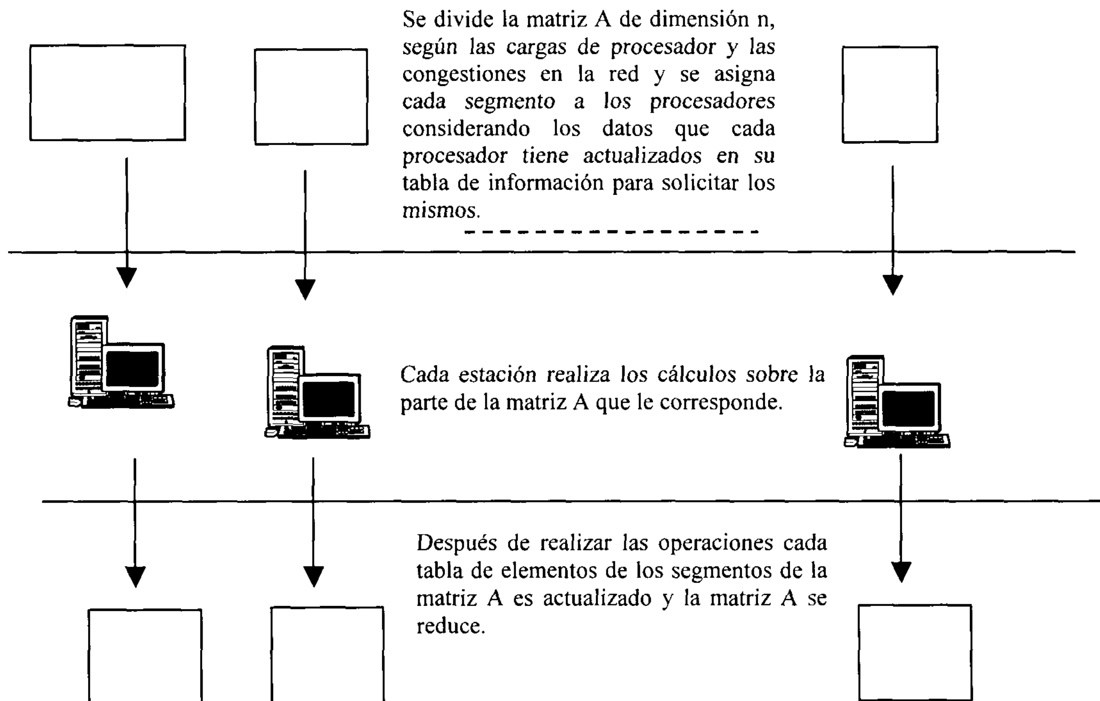


Figura 4.8.- Paralelización con balanceo de LU.

4.4 Comportamiento del Algoritmo Paralelo Balanceado de LU.

A continuación presentamos un ejemplo para mostrar los pasos del algoritmo. El algoritmo paralelo balanceado LU comienza dividiendo la matriz A en partes iguales según el número de procesadores, como se mencionó con anterioridad para conocer las cargas en el sistema tanto de los procesadores como en la red. Para hacer más sencilla la explicación asumiremos que únicamente tenemos tres procesadores como se observa en la Figura 4.9, por lo que tendremos tres segmentos de la matriz A y una matriz LU para cada procesador y la tabla de elementos de los segmentos de la matriz A no contiene datos. Este particionamiento de la matriz A en partes iguales solo se presenta en la iteración 1.

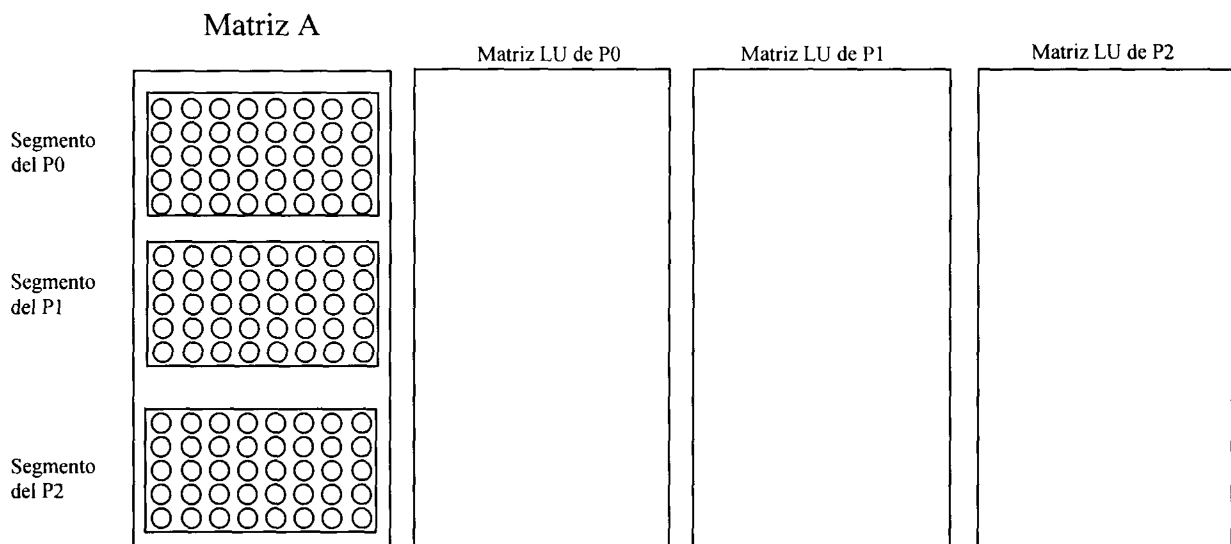


Figura 4.9.-Particionamiento de la matriz A, en la iteración 1.

Una vez asignados los segmentos de datos a cada procesador estos llenarán su tabla de elementos de los segmentos de la matriz A, para reflejar las filas con las que cuenta cada procesador y comenzarán con la ejecución de las operaciones. Cada procesador llenará la sección que le corresponde de la matriz LU, basado en las operaciones realizadas sobre el segmento de la matriz A que le corresponde, como puede observarse en la Figura 4.10. Cada procesador también medirá su tiempo de ejecución de operaciones y los tiempos de transmisión de información para posteriormente dividir la Matriz A según las capacidades de cada procesador.

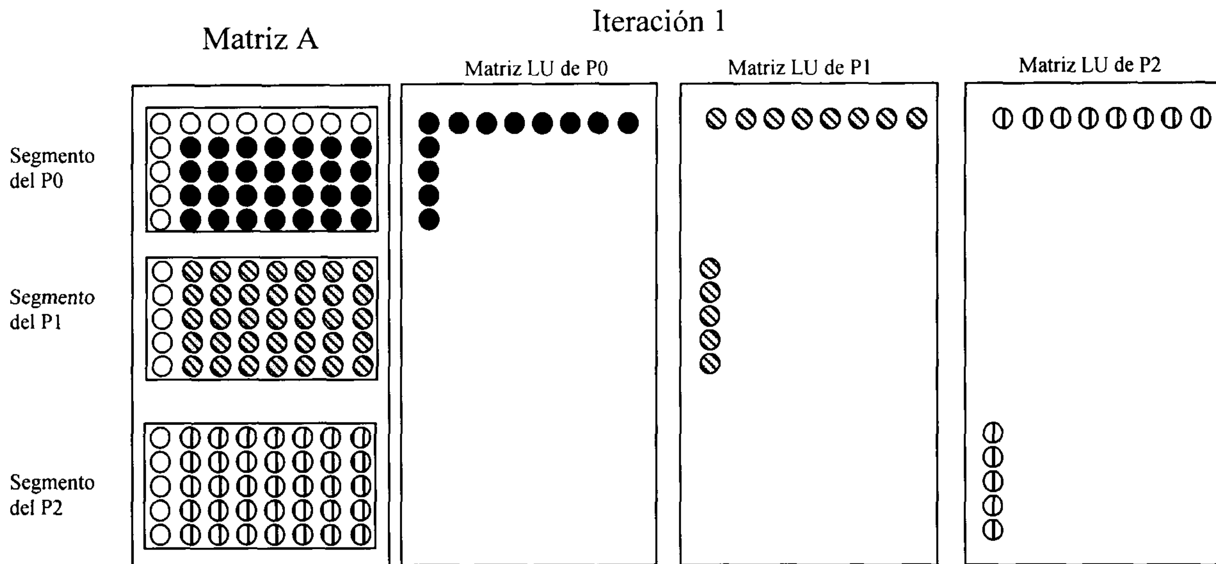


Figura 4.10.-Actualización de las matrices A y LU, al finalizar la iteración 1.

Al termino de la iteración 1, todos los procesadores se envían la información de los tiempos que le llevo a cada uno transmitir datos y ejecutar las operaciones. En el comienzo de la iteración 2 y en base a estos tiempos todos los procesadores realizan cálculos para determinar el porcentaje y el número de filas de la matriz A que le corresponde a cada procesador, a partir de entonces cada procesador analiza y actualiza su tabla de elementos de los segmentos de la matriz A para determinar las filas que deberá enviar y/o recibir.

En nuestro ejemplo de tres procesadores como se puede observar en la Figura 4.11 se observa que el procesador que se estima ejecutará las operaciones con mayor rapidez será el procesador P1 ya que se le ha asignado un segmento de mayor tamaño y se habla de que es una estimación ya que hay que considerar que los tiempos fueron medidos en la iteración anterior y pudieran no reflejar algún cambio de último momento en el sistema.

Una consideración que es importante hacer notar es que actualizar la información, de tiempos de ejecución y cargas de red, en cada iteración a los procesadores es muy costoso y resulta en una degradación en los tiempos de ejecución totales, por lo que la experimentación para encontrar cuando es el momento óptimo para realizar dicho intercambio de información arrojo como resultado de cada 30 iteraciones para el caso particular de nuestro ambiente anteriormente descrito, esta consideración será tratada a mas detalle en el capitulo de experimentación. Regresando a la Figura 4.8 observemos que el procesador segundo más rápido es el procesador P0 y el mas lento es el procesador P2.

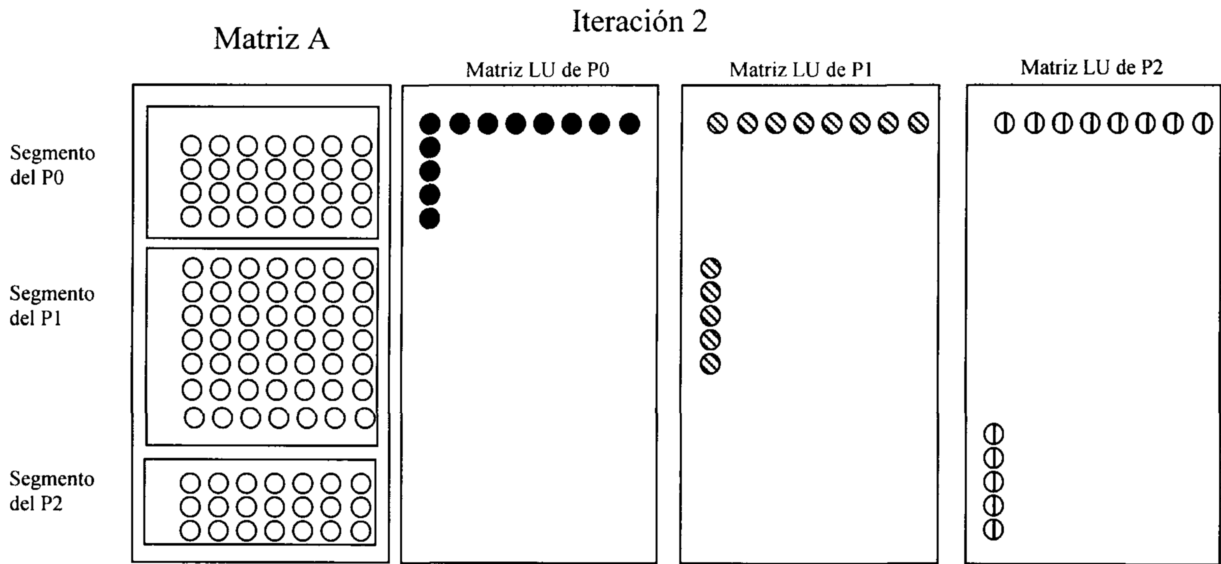


Figura 4.11.- Particionamiento de la Matriz A, en la iteración 2.

Una vez que se han asignado los segmentos de la matriz A a cada procesador y se ha actualizado la tabla de elementos de los segmentos de la matriz A, cada procesador empezará a realizar los cálculos con dichos segmentos para obtener los valores de la matriz LU que le corresponde, como puede verse en la Figura 4.12, el procesador P1 en este caso llenará mas elementos de la segunda columna de LU, dado que es el procesador mas rápido, de igual forma se sigue los procesadores P0 y P2.

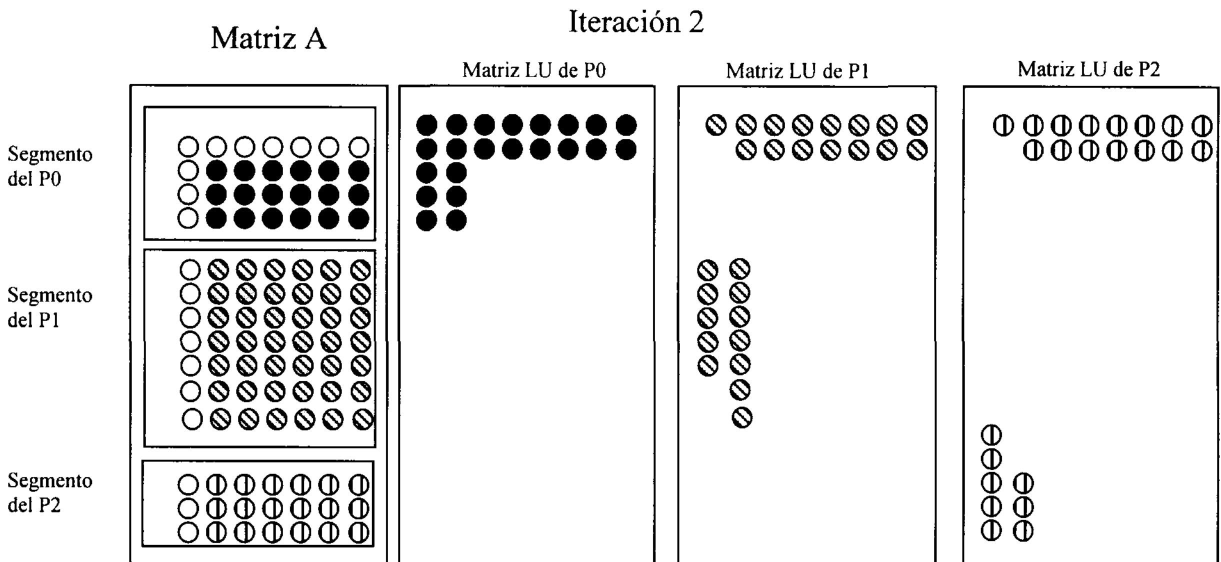


Figura 4.12.- Actualizaciones sobre las matrices A y LU, al finalizar la iteración 2.

En esta iteración 2 y en las subsiguientes iteraciones y hasta que la dimensión de la matriz A sea igual a 1, se medirán los tiempos de intercambio de información y los tiempos de ejecución, pero no se hará ninguna actualización global de dicha información si no hasta que el numero de iteración sea 30 o múltiplo, es decir, las primeras 29 iteraciones balancearán con la información de la primera iteración todos los procesadores, en la iteración 30 balancearán con la información de la iteración 29 y este mismo ciclo se seguirá hasta que la dimensión de A sea 1.

Recordemos que en este algoritmo paralelo balanceado de LU, cada procesador calcula su tiempo de ejecución y transmisión, cada procesador realiza todos los cálculos para determinar el segmento de la matriz A que le corresponde a cada procesador, que todos los procesadores mantienen actualizada su tabla de elementos de los segmentos de la matriz A y de esta forma se sincronizan, a sí como también todos participan en el intercambio de información.

En este capítulo se presentaron los conceptos y detalles más relevantes de la implementación del algoritmo paralelo balanceado de LU. En el siguiente capítulo se presentan los resultados de las pruebas de ejecución de los algoritmos secuencial, paralelo y paralelo balanceado de LU y se realiza el análisis del desempeño de los algoritmos.

Capítulo 5

Experimentos

En este capítulo se presentan los resultados de los experimentos realizados con los algoritmos paralelo y paralelo balanceado en el ambiente heterogéneo de redes de estaciones de trabajo.

La experimentación consistió en el diseño, desarrollo y ejecución de los programas secuencial, paralelo y paralelo balanceado del algoritmo descomposición de matrices LU, con el objeto de obtener los tiempo de ejecución, speedup y eficiencia de las versiones paralela y paralela balanceada, con diferentes tamaños de matrices de entrada, a sí como también con diferente número de procesadores.

5.1 Entorno Experimental

Para llevar a cabo los experimentos se utilizaron 7 estaciones de trabajo UNIX, conectadas en bus, con la tecnología ethernet, corriendo el protocolo de comunicación TCP/IP. Enseguida se detallan las características de las estaciones de trabajo:

a) Galileo, Newton y Davinci son estaciones de trabajo IBM RS6000 con sistema operativo AIX 4.2, 64 MB de memoria RAM y procesador Power PC.
b) Copernico es una estación de trabajo Sun UltraSparc 5 con sistema operativo Solaris versión 2.6, 128 MB en RAM.
c) Neumann y Backus son estaciones de trabajo Sun UltraSparc 5 con sistema operativo Solaris versión 2.7, 128 MB en RAM.
d) Newcop es una estacion de trabajo Sun UltraSparc 10 con sistema operativo Solaris versión 2.7, 128 MB en RAM.

Los ambientes de prueba se configuraron como se muestra en la tabla 5.1, en esta se observa que el ambiente de 4 procesadores esta integrado por Neumann, Backus, Newcop y Copernico, el ambiente de 5 procesadores esta integrado por Neumann, Backus, Newcop, Copernico y Galileo, el ambiente de 6 procesadores esta formado por Neumann, Backus, Newcop, Copernico, Galileo y Newton, el ambiente de 7 procesadores esta formado por Neumann, Backus, Newcop, Copernico, Galileo, Newton y Davinci. Todos las máquinas que forman parte de estos ambientes son máquinas en producción por lo que las pruebas fueron hechas con el uso normal de estas.

	Ambiente	De	Pruebas	
Npe	4	5	6	7
Neumann	✓	✓	✓	✓
Backus	✓	✓	✓	✓
Newcop	✓	✓	✓	✓
Copernico	✓	✓	✓	✓
Galileo		✓	✓	✓
Newton			✓	✓
Davinci				✓

Tabla 5.1.-Ambiente de pruebas

El algoritmo LU es sus versiones secuencial, paralelo y paralelo balanceado fueron implementados utilizando el lenguaje C y el compilador GNU C 2.7.2.2.

En las versiones paralelo y paralelo balanceado del algoritmo LU, se utilizó el modelo de programación de memoria distribuida bajo el paradigma de paso de mensajes a través de su implementación MPICH, basado en el estándar MPI.

5.2 Metodología

Se investigaron algoritmos de balanceo de carga para ambientes heterogéneos de producción y con esto se generaron las ideas que sirvieron en la implementación del algoritmo paralelo balanceado LU, que se describió en el capítulo 4.

Se diseñó y desarrollaron los códigos de tres programas del algoritmo LU.

a) Secuencial.
b) Paralelo.
c) Paralelo balanceado.

Para cada algoritmo se eligieron 3 tamaños de datos (matrices de 512, 1024, 2048) y diferentes configuraciones de la máquina paralela (4,5,6 y 7 nodos) para su ejecución:

Los programas secuenciales se ejecutaron en las estaciones de trabajo Neumann, Backus, Newcop, Copernico, Galileo, Newton y Davinci. Cada programa se ejecuto 4 veces procurando que la carga de los procesadores sea consistente durante toda la experimentación, así como también que la carga en la red sea normal, es decir, los

programas fueron ejecutados durante las horas de uso normal de las estaciones de trabajo. De los tiempos obtenidos en las corridas se consideró el menor como el tiempo de ejecución del programa.

Estas ejecuciones mostraron la relación de velocidad entre las máquinas, quedando en el siguiente orden respecto de Newton que es la más lenta: Neumann 515% más rápida, Backus 517% más rápida, Newcop 693% más rápida, Copernico 394% más rápida, Galileo y Davinci son iguales a Newton.

5.3 Métricas de Evaluación

El desempeño de los algoritmos se evalúo mediante las siguientes métricas: Speedup, Speedup Óptimo Heterogéneo y Eficiencia.

5.3.1 Speedup

Speedup es la medida que refleja el beneficio relativo de resolver un problema en paralelo. Se define como la razón del tiempo que toma resolver un problema en un solo procesador entre el tiempo requerido para resolver el mismo problema en una computadora paralela con *Npe* procesadores idénticos. Formalmente Speedup es la razón del tiempo de ejecución de un programa secuencial entre el tiempo del algoritmo paralelo para resolver el mismo problema en *Npe* procesadores.

$$S = Speedup = \frac{Tiempo \quad de \quad ejecución \quad en \quad un \quad procesador}{Tiempo \quad de \quad ejecución \quad en \quad Npe \quad procesadores}$$

En un ambiente homogéneo ideal todos los procesadores realizan la misma cantidad de trabajo en el mismo tiempo, la aplicación es completamente paralelizable y se considera que los tiempos de trasmisión de información entre procesadores es 0, de esta forma un programa paralelo que corre en este ambiente, tardará *Ts/Npe*, donde *Ts* es el tiempo de ejecución de la versión secuencial del mismo programa y *Npe* es el número de procesadores, en consecuencia el Speedup que se logra es el ideal y es igual al número de procesadores, como se muestra en la formula:

$$S = \frac{Ts}{Tp} = \frac{Ts}{Ts/Npe} = Npe$$

El Speedup alcanzado por un programa paralelo en cualquier ambiente no ideal se define como el Speedup observado.

5.3.2 Speedup Óptimo Heterogéneo

En un ambiente heterogéneo los procesadores realizan diferentes cantidades de trabajo en el mismo tiempo. En un ambiente heterogéneo ideal todos los procesadores están dedicados a resolver un problema a la vez y el tiempo de transmisión de información es 0. Un programa paralelo que corre en este ambiente ideal obtendrá su tiempo óptimo de ejecución si éste programa es particionado de manera balanceada, según las capacidades de los procesadores. Es decir, debemos encontrar el tiempo óptimo en que cada procesador puede realizar la mayor cantidad de trabajo, con la restricción de que este tiempo es igual para todos. Se define Speedup heterogéneo ideal, *So*, como la razón de dividir el tiempo de ejecución del programa secuencial, *Ts*, en el procesador más rápido entre el tiempo óptimo de ejecución del programa paralelo balanceado, *Tob*, cuando corre en un ambiente ideal, como se muestra en la formula:

$$So = \frac{Ts}{Tob}$$

A continuación se muestra la forma de obtener el Speedup óptimo heterogéneo, en un ambiente formado por una red de estaciones de trabajo heterogéneas en producción a través del siguiente ejemplo:

Se asume que se tiene una red de 7 estaciones de trabajo heterogéneas en producción y se requiere conocer el speedup óptimo heterogéneo. En este ambiente no es posible obtener el tiempo óptimo del algoritmo paralelo balanceado, *Tob*, para conocer el speedup óptimo a partir de su formula, ya que este ambiente no es óptimo heterogéneo, es decir, las estaciones tienen cargas distintas y lo mismo pasa con la red. El cálculo del speedup óptimo heterogéneo, como se muestra en la tabla 5.2, se obtienen a partir de los tiempos de ejecución del programa secuencial, *Ts* en la tabla, de cada una de las estaciones de trabajo. En base a estos tiempos se obtiene la relación de rapidez, de cada una de las estaciones con respecto de la más rápida, que se calcula dividiendo el tiempo de ejecución del algoritmo secuencial de la máquina más rápida (nodo 3) entre cada tiempo secuencial de las estaciones restantes, *Rr* en la tabla. A partir de estas relaciones de rapidez obtenemos la equivalencia de las 7 estaciones con respecto de la más rápida, que se calcula sumando cada valor de la relación de rapidez, y obtenemos que las 7 estaciones son equivalentes a 3.48 estaciones como la más rápida, y esta equivalencia es precisamente el speedup óptimo heterogéneo, *So* en la tabla, para este ambiente en particular.

Npe	Ts	Rr
1	7082	0.74
2	7056	0.74
3	5244	1
4	9243	0.56
5	36441	0.14
6	36506	0.14
7	36213	0.14
	So =	3.48

Tabla 5.2.- Speedup óptimo heterogéneo.

Los speedups óptimos heterogéneos para todos los ambientes de prueba (4, 5, 6 y 7 nodos) fueron calculados de esta misma manera y se muestran en la tabla 5.3.

Npe	So
4	3.05
5	3.19
6	3.34
7	3.48

Tabla 5.3.- Speedup óptimo heterogéneo de todos los ambientes de prueba.

5.3.3 Eficiencia en un Ambiente Heterogéneo

Eficiencia es la fracción de tiempo para la cual los procesador son útilmente empleados. Se define como la razón del Speedup observado entre el Speedup óptimo Heterogéneo.

$$E = \frac{S}{So}$$

5.4 Resultados

En esta sección se presenta los resultados de ejecutar el programa paralelo en el ambiente homogéneo y los resultados obtenidos a partir de las ejecuciones en el ambiente

heterogéneo de los diferente implementaciones del algoritmo LU: secuencial, paralelo y paralelo balanceado.

En esta sección se introducirán los conceptos de frecuencia de balanceo y predicción de carga, estos conceptos son muy importantes ya que afectan el tiempo de ejecución del algoritmo paralelo balanceado.

5.4.1 Resultados de Ejecutar el Programa Paralelo en el Ambiente Homogéneo.

Para conocer el comportamiento de algoritmo paralelo en un ambiente homogéneo se realizó experimentos en los ambientes de 2, 4, 6 y 8 procesadores, con matrices de dimensión 256, 512 y 1024. Los resultados son presentados en las tablas 5.4, 5.5 y 5.6 en donde se presentan diferentes parámetros de medición: tiempo secuencial *(Ts)*, Tiempo paralelo *(Tpar)*, Speedup óptimo homogéneo *(Soh)*, Speedup del programa paralelo *(Sp)* y Eficiencia del programa paralelo *(Ep)*.

La analizar los resultados presentados en las tablas 5.4, 5.5 y 5.6 observamos que el tiempo paralelo para las matrices de 256 tuvo una tendencia incremental y que para matrices de 512 y 1024 la tendencia fue decremental. Estas tendencias se explican por que el tiempo de comunicación comparado con el tiempo de procesamiento es más significativo para matrices pequeñas.

	Matriz256				
Npe	Ts	Tpar	Soh	Sp	Ep
2	278	148	2	1.88	94
4	278	136	4	2.04	51
6	278	155	6	1.79	30
8	278	191	8	1.46	18

Tabla 5.4.- Resultados del algoritmo paralelo de LU para matrices de 256.

	Matriz512				
Npe	Ts	Tpar	So	Sp	Ep
2	2191	978	2	1.99	99
4	2191	711	4	3.08	77
6	2191	709	6	3.09	52
8	2191	753	8	2.91	36

Tabla 5.5.- Resultados del algoritmo paralelo de LU para matrices de 512

Matriz1024					
Npe	Ts	Tpar	So	Sp	Ep
2	17443	7225	2	1.99	99
4	17443	4411	4	3.95	99
6	17443	3794	6	4.60	77
8	17443	3700	8	4.71	59

Tabla 5.6.- Resultados del algoritmo paralelo de LU para matrices de 1024

Las Figuras 5.1, 5.2 y 5.3 muestran una tendencia decremental en los Speedups al aumentar el número de procesadores para las matrices de 256 y 512, que para las matrices de 1024 tuvo una tendencia incremental y que al aumentar la dimensión de las matrices el Speedup también se incrementa pues toma valores desde 1.46 hasta 4.71 para matrices de 256 y matrices de 1024 respectivamente con un incremento del 40%.

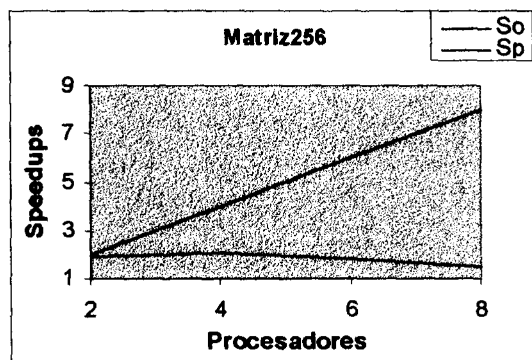


Figura 5.1.- Speedup de matrices de dimensión 256.

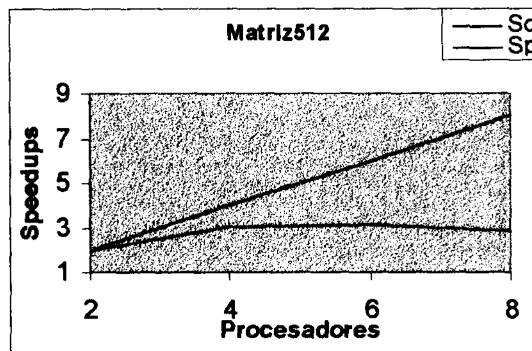


Figura 5.2.- Speedup de matrices de dimensión 512.

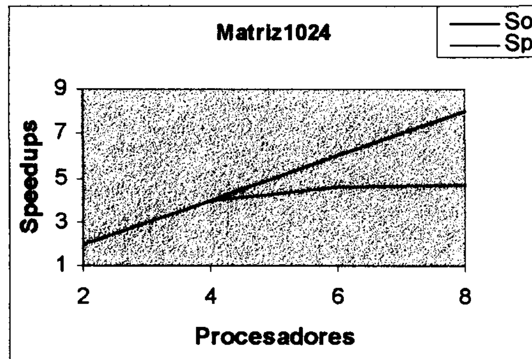


Figura 5.3.- Speedup de matrices de dimensión 1024.

En las Figuras 5.4, 5.5 y 5.6 se observa que las eficiencias son buenas y presentan una tendencia incremental al aumentar la cantidad de trabajo puesto que toman valores del 18% hasta el 99% para matrices de 256 y 1024 respectivamente. Se observa también en estas Figuras que al aumentar el número de procesadores la eficiencia presenta una tendencia decremental, la razón de este es por que el algoritmo utilizado para solucionar LU de manera paralela no es escalable.

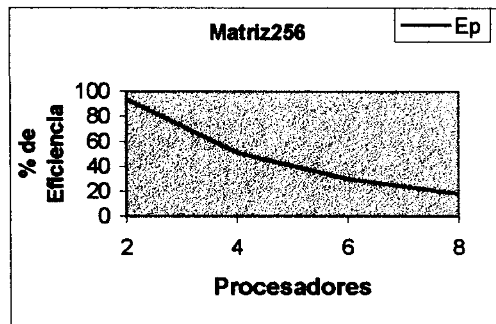


Figura 5.4.- Eficiencia lograda en matrices de 256.

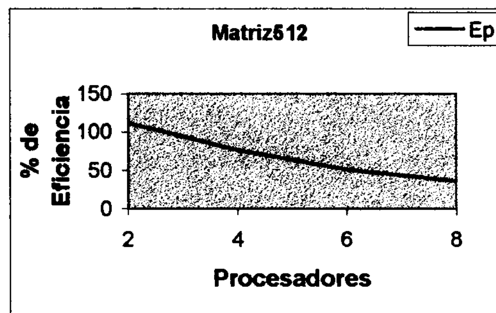


Figura 5.5.- Eficiencia lograda en matrices de 512.

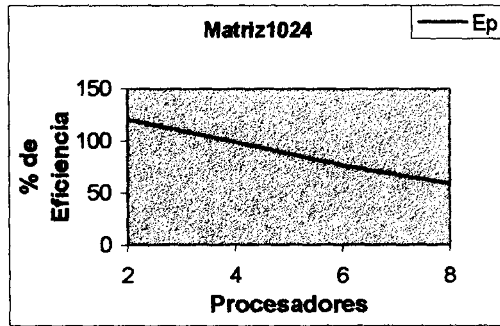


Figura 5.6.- Eficiencia lograda en matrices de 1024.

5.4.2 Frecuencia de Balanceo.

El algoritmo de balanceo dinámico de LU requiere conocer el número de iteraciones necesarias que debe esperar antes de balancear, este número se define como frecuencia de balanceo, identificar dicha frecuencia antes de ejecución es determinante para lograr buenos resultados en tiempos de ejecución, ya que si el algoritmo no balancea en la iteración adecuada, el tiempo de ejecución puede ser considerablemente mayor que el tiempo de ejecución secuencial. El algoritmo de balanceo dinámico LU, se basa en la frecuencia de balanceo para la sincronización de los tiempos de comunicación, procesamiento y overhead de balanceo, con el objeto de reaccionar ante los cambios en el ambiente.

Los experimentos que a continuación se presentan determinan la frecuencia de balanceo que usará el algoritmo. Las Figuras 5.7 y 5.8 muestran los resultados de ejecutar el algoritmo en 4, 5, 6 y 7 procesadores, con matrices de dimensión 1024 y 2048 respectivamente y con frecuencias de sincronización de los tiempos de comunicación, procesamiento y overhead de balanceo, de cada 15, 20, 25, 30, 35, 40, 45 iteraciones, es decir, el algoritmo balancea cada 15, 20, 25, 30, 35, 30, 40 y 45 iteraciones. Estas figuras muestran que tanto para la matriz de 1024 como para la matriz de 2048 el algoritmo obtuvo menores tiempos de ejecución en frecuencias de balanceo de cada 30 iteraciones, ejecutados en 4, 5, 6,y 7 procesadores.

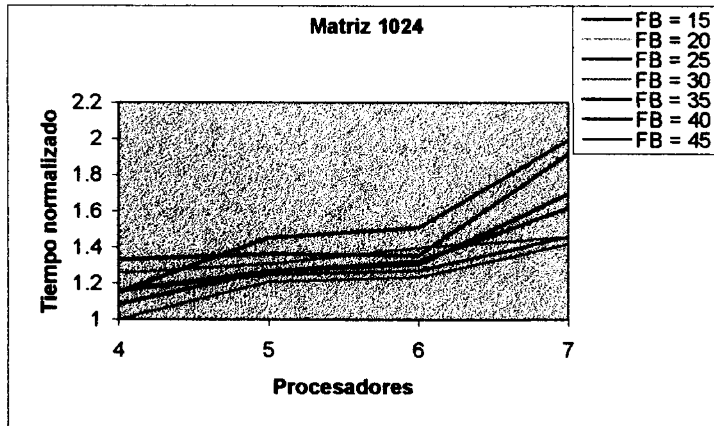


Figura 5.7.- Tiempos normalizados y frecuencias del algoritmo paralelo balanceado de LU para matrices de dimensión 1024.

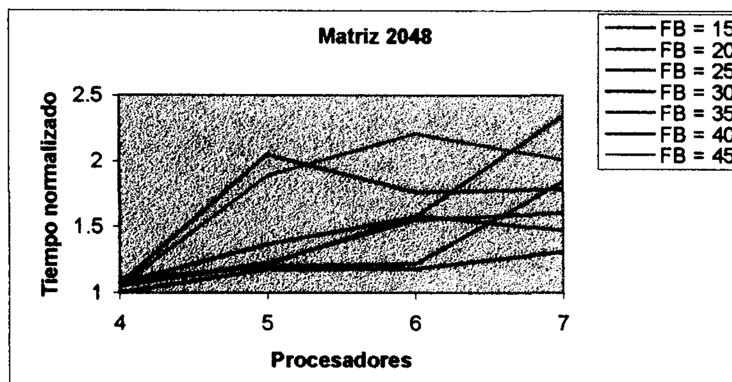


Figura 5.8.- Tiempos normalizados y frecuencias del algoritmo paralelo balanceado de LU para matrices de dimensión 2048.

Esta frecuencia de balanceo, cada 30 iteraciones, únicamente es óptima para el ambiente de pruebas, ya que depende del algoritmo de balanceo dinámico mismo y de las características de los equipos de computo y de red.

5.4.3 Predicción de Carga.

En un ambiente heterogéneo los cambios de carga ocurren de acuerdo a las características propias del ambiente, siendo los más significativos los patrones de uso de los usuarios y los tipos de aplicación. En tales ambiente heterogéneos los patrones de carga en los procesadores pueden tener una varianza muy grande para cierto periodos de tiempo.

El algoritmo de balanceo de carga dinámico reacciona de acuerdo a los tiempos de comunicación, procesamiento y overhead de balanceo, de tal manera que si en *algún punto*

específico durante la ejecución del programa la varianza de tales tiempos es muy grande, la reacción inmediatamente a tales cambios bruscos puede resultar en peores tiempos de ejecución con respecto al tiempo de ejecución del programa secuencial, ya que por ejemplo un procesador con poco carga recibirá cierta cantidad de trabajo a realizar del algoritmo LU, en la siguiente iteración recibe considerable carga y libera trabajo por realizar según el algoritmo de balanceo, inmediato después, siguiente iteración, este procesador se descarga totalmente y se le asignara de nuevo trabajo a realizar del algoritmo LU. Estos cambios alternos de carga muy marcados pueden provocar demasiada comunicación y por lo tanto mas tiempo de ejecución.

Para conocer la capacidad de reacción que deberá tener el algoritmo se usara una formula que predice tiempos basándose en la historia de tales tiempos. Esta formula se explicará a continuación, suponga que el tiempo el tiempo de ejecución de una iteración es *T0*. Ahora suponga que la siguiente iteración toma un tiempo de *T1*. Entonces el tiempo estimado de la tercera iteración esta dado por la suma pesada de estos dos tiempos, esto es, $aT0 + (1-a)T1$. La elección de a decide el olvidar las viejas corridas rápidamente, o recordar ellas por un largo tiempo. Con $a = ½$, se obtienen estimaciones sucesivas de:

$$T0,\ T0/2 + T1/2,\ T0/4 + T1/4 + T2/2,\ T0/8 + T1/8 + T2/4 + T3/2$$

Esto es, después de tres nuevas corridas, el peso de *T0* en esta última iteración ha descendedido hasta 1/8. De esta forma la capacidad de reacción esta dada por la elección de a.

Las configuraciones en los pesos, de la formula de predicción, utilizados para la experimentación serán representadas como (a,1-a). Los experimentos se realizaron con matrices de dimensión de 1024 y 2048, en 4, 5, 6 y 7 procesadores, con las configuraciones en los pesos siguientes (0.5, 0.5), (0.3, 0.7), (0.7, 0.3) y (0.0, 1.0) y con una frecuencia de balanceo de cada 30 iteraciones. Los resultados de los experimentos se muestran en las Figuras 5.9 y 5.10. Se observa que los mejores tiempos se obtuvieron en la configuración de pesos de la formula de predicción (0.0, 1.0) tanto para las matrices de 1024 como para las matrices de 2048, en 4, 5, 6 y 7 procesadores. Esto significa que para nuestro ambiente en particular lo óptimo es no tomar en cuenta la historia y si reaccionar ante los cambios inmediatos del sistema.

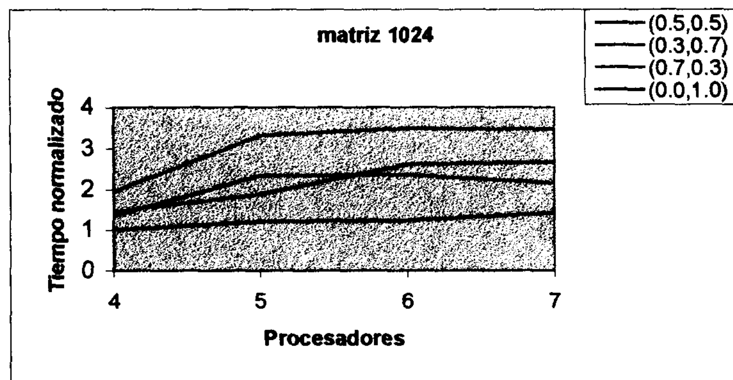


Figura 5.9.- Tiempos normalizados y pesos, de la formula de predicción, para el algoritmo paralelo balanceado de LU para matrices de dimensión 1024.

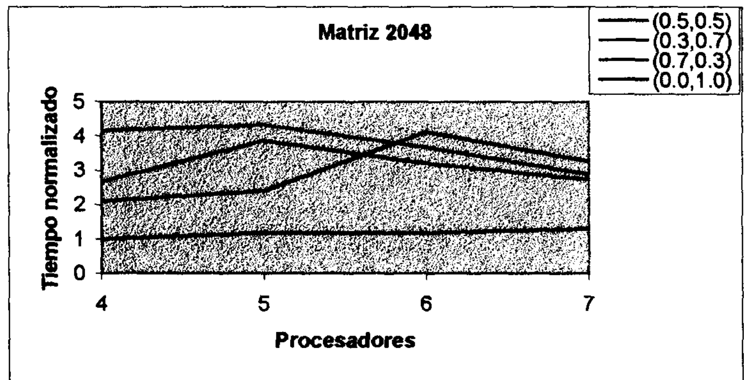


Figura 5.10.- Tiempos normalizados y pesos, de la formula de predicción, para el algoritmo paralelo balanceado de LU para matrices de dimensión 2048.

5.4.4 Resultados de los Algoritmos Paralelo y Paralelo Balanceado de LU.

Con el objeto de conocer los beneficios del programa paralelo balanceado de LU con respecto del programa paralelo de LU y el secuencial, se realizó experimentación en los ambientes heterogéneos de 4, 5, 6 y 7 procesadores, con matrices de dimensión de 512, 1024 y 2048. Los resultados arrojados son presentados en la tablas 5.7, 5.8 y 5.9 en donde se presentan diferentes parámetros de medición: tiempo secuencial (*ts)*; tiempo del programa paralelo (*Tpar)*; tiempo del programa paralelo balanceado (*TparB)*; tiempo del programa paralelo balanceado de LU mejorado (*TparBM)*; Speedup óptimo heterogéneo (*So)*; Speedup del programa paralelo (*Sp)*; Speedup del programa paralelo balanceado (*Sb)*; Speedup del programa paralelo balanceado mejorado (*Sbm)*; eficiencia del programa paralelo, (*Ep)*; eficiencia del programa paralelo balanceado (*Eb)* y eficiencia del programa paralelo balanceado mejorado (*Ebm)*.

El programa paralelo balanceado de LU mejorado es exactamente igual al programa paralelo balanceado de LU con la única diferencia que en su estado inicial la matriz A es particionada de acuerdo a las capacidades de los procesadores, dichas capacidades son calculadas previamente y asignadas en duro en el código del programa. Esta versión de programa se realizó con el objeto de conocer la ventaja de realizar esto contra el asumir inicialmente que todas las capacidades de los procesadores son iguales.

Las tablas 5.7, 5.8 y 5.9 muestran que para todas las configuraciones de los procesadores y para los distintos tamaños de matrices el programa paralelo prácticamente no obtuvo Speedups, únicamente los obtuvo en los ambientes de 4 procesadores con todos los tamaño de matrices, debido a que en este ambiente los procesadores son muy similares en cuanto a su capacidad (ver la sección entorno experimental). Los programas, paralelo

balanceado de LU y paralelo balanceado de LU mejorado, obtuvieron Speedups en todos estos ambientes indicando que estos programas son efectivos en el ambiente heterogéneo.

Al analizar los resultados presentados en las tablas 5.7, 5.8 y 5.9 observamos que los programas paralelo balanceado de LU y paralelo balanceado de LU mejorado, obtuvieron los mejores tiempos de ejecución, para los ambientes de prueba de 4, 5, 6 y 7 procesadores y para todos los tamaños de matrices de 512, 1024 y 2048. Otra observación importante es que las diferencias entre ambos son relativamente muy pequeñas. Esto indica que el programa paralelo balanceado de LU es efectivo y se adapta muy rápido al estado del sistema heterogéneo.

Matriz512

Npe	Ts	Tpar	TparB	So	Sp	Sb	Ep	Eb
4	657	559	501	3.05	1.18	1.31	39	43
5	657	1218	570	3.19	0.54	1.15	17	36
6	657	1077	604	3.34	0.61	1.09	18	33
7	657	1108	792	3.48	0.59	0.83	17	24

Tabla 5.7.- Resultados de experimentar con el algoritmo LU en todas sus versiones, para matrices de dimensión 512

Matriz1024

Npe	Ts	Tpar	TparB	TparBM	So	Sp	Sb	Sbm	Ep	Eb	Ebm
4	5244	2820	2666	2732	3.05	1.86	1.97	1.92	61	64	63
5	5244	8194	3223	2935	3.19	0.64	1.63	1.79	20	51	56
6	5244	7123	3291	2989	3.34	0.74	1.59	1.75	22	48	53
7	5244	6356	3782	3381	3.48	0.83	1.39	1.55	24	40	45

Tabla 5.8.- Resultados de experimentar con el algoritmo LU en todas sus versiones, para matrices de dimensión 1024.

Matriz2048

Npe	Ts	Tpar	TparB	TparBM	So	Sp	Sb	Sbm	Ep	Eb	Ebm
4	42451	19816	18355	19447	3.05	2.14	2.31	2.18	70	76	72
5	42451	63300	21626	20717	3.19	0.67	1.96	2.05	21	62	64
6	42451	53384	21706	21472	3.34	0.80	1.96	1.98	24	59	59
7	42451	46403	24093	22261	3.48	0.91	1.76	1.91	26	51	55

Tabla 5.9.- Resultados de experimentar con el algoritmo LU en todas sus versiones, para matrices de dimensión de 2048.

La Figura 5.11 presenta las mejoras relativas de los tiempos de ejecución del programa paralelo balanceado con respecto del programa paralelo. Esta figura muestra que para matrices de 512 la mejora relativa de los tiempos de ejecución del programa paralelo balanceado con respecto del programa paralelo va desde 1.12 veces mejor hasta 1.78, para

matrices de 1024 la mejora relativa va desde 1.06 veces mejor hasta 2.54 y que para matrices de 2048 la mejora relativa va desde 1.08 veces mejor hasta 2.93.

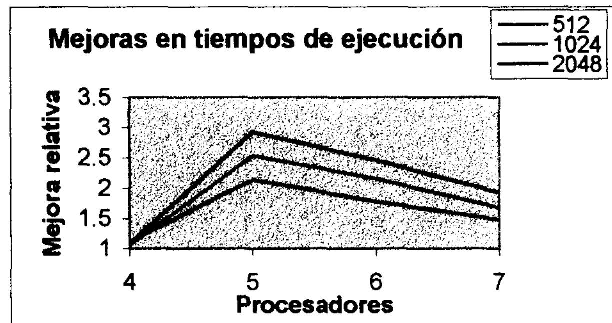


Figura 5.11.-Mejoras relativas de los tiempos de ejecución del programa paralelo balanceado respecto del programa paralelo, para matrices de 512, 1024 y 2048.

Las Figuras 5.12, 5.13 y 5.14 muestran también la tendencia en los Speedups. Se observa que en todos los ambientes y con todos los tamaños de matrices, que a mayor cantidad de procesadores el Speedup se decrementa y a mayor tamaño de la dimensión de la matriz A, el Speedup se incrementa, nótese que toma valores de 0.83 para matrices de 512 hasta 2.31 para matrices de 2048, con ganancia relativa del 178%. Notemos que el patrón que sigue la tendencia en éstos Speedups es semejante al patrón observado en los Speedups en el ambiente homogéneo con el programa paralelo de LU (fig. 5.1, 5.2 y 5.3). La razón de la tendencia en descenso de los Speedups es que a mayor número de procesadores la partición de A es mas granular y por lo tanto generará mayor overhead de comunicación. La tendencia en ascenso se debe a que con el aumento en la dimensión de la matriz A el overhead de comunicación se vuelve menos significativo comparado con el tiempo de procesamiento, pues al ser más grande la matriz A cada procesador realizara más trabajo.

Las figuras muestran también que el Speedup del programa balanceado *(Sb)* presenta una mejora significativa con respecto del Speedup del programa paralelo *(Sp)*. Esto nos indica la ganancia de balancear con respecto de no balancear en un ambiente heterogéneo, ya que si en el balanceo prácticamente resulta mejor ejecutar el programa de manera secuencial en la máquina más rápida y de aquí la importancia de este algoritmo paralelo balanceado.

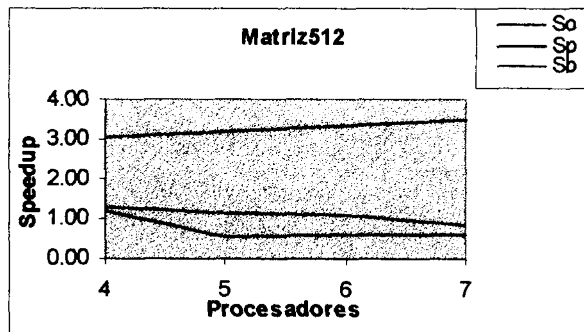


Figura 5.12.-Muestra los Speedups de los programas paralelo de LU (Sp), paralelo balanceado de LU (Sb) y el óptimo heterogéneo (So), para matrices de dimensión de 512.

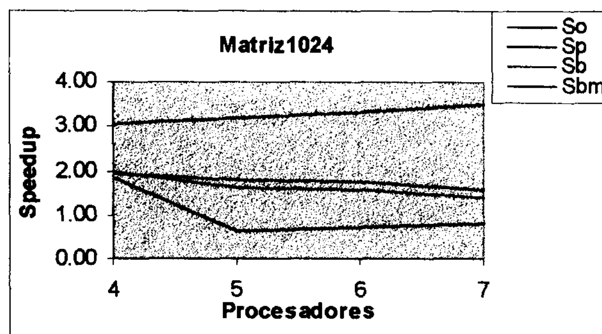


Figura 5.13.-Muestra los Speedups de los programas paralelo de LU (Sp), paralelo balanceado de LU (Sb), paralelo balanceado de LU mejorado (Sbm) y el óptimo heterogéneo (So), para matrices de dimensión de 1024

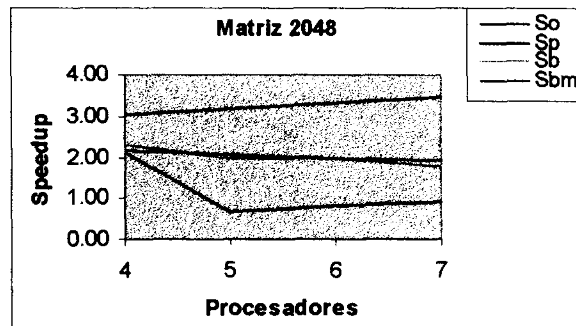


Figura 5.14.-Muestra los Speedups de los programas paralelo de LU (Sp), paralelo balanceado de LU (Sb), paralelo balanceado de LU mejorado (Sbm) y el óptimo heterogéneo (So), para matrices de dimensión de 2048.

En las Figuras 5.15, 5.16 y 5.17 se observa que la eficiencia lograda por el algoritmo paralalelo balanceado (Eb), es significativamente mejor que la eficiencia lograda por el algoritmo paralelo (Ep) para todos los tamaño de matrices y para todos los ambientes de prueba. El cálculo de las mejoras relativas en la eficiencia del programa balanceado, *(Eb)*, con respecto del programa paralelo, *(Ep)*, va desde 12% hasta 114% para matrices de 512, del 6% hasta el 154% para matrices de 1024 y por último del 8% hasta el 193% para

matrices de 2046, se observa en la Figura 5.18. Otra observación importante al respecto de estos resultados es que la eficiencia máxima relativa aumento en un 40% en promedio al duplicar el tamaño de las matrices. Estos resultados comprueban la ganancia significativa de balancear en un ambiente heterogéneo contra no balancear y son la principal aportación de este trabajo ya que se logro eficiencias relativas de hasta el 193%.

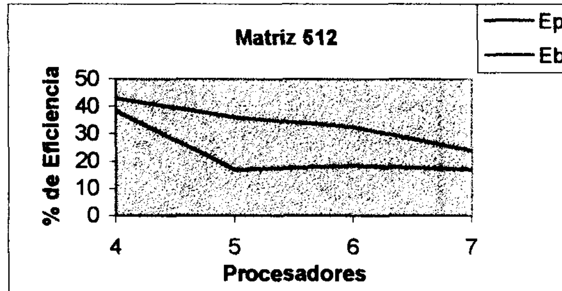


Figura 5.15.- Muestra la eficiencia lograda por los programas paralelo de LU (Ep) y paralelo balanceado de LU (Eb), para una matriz de 512.

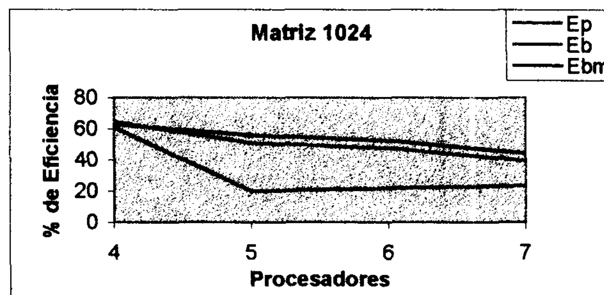


Figura 5.16.- Muestra la eficiencia lograda por los programas paralelo de LU (Ep), paralelo balanceado de LU (Eb) y paralelo balanceado de LU mejorado (Ebm) para una matriz de 1024.

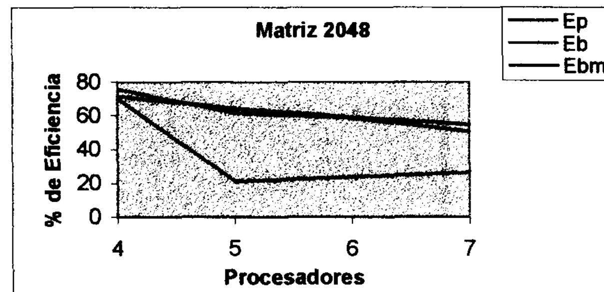


Figura 5.17.- Muestra la eficiencia lograda por los programas paralelo de LU (Ep), paralelo balanceado de LU (Eb) y paralelo balanceado de LU mejorado (Ebm) para una matriz de 2048.

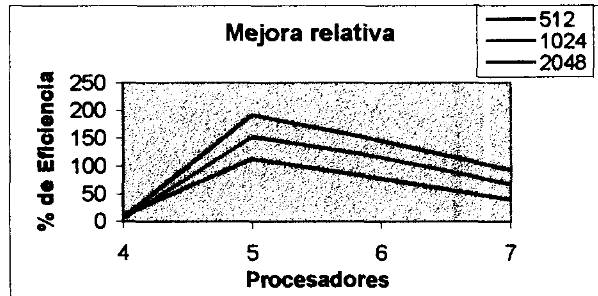


Figura 5.18.- Mejora relativa en la eficiencia del programa paralelo balanceado respecto del programa paralelo para matrices de 512, 1024 y 2048

Las pruebas presentadas en este capítulo demuestran la ventaja ofrecida por el algoritmo paralelo balanceado de LU con respecto de su versión secuencial y paralela, pues este obtuvo los mejores tiempos de ejecución, los mejores Speedups y la mejor eficiencia para todos los ambientes y con todos los tamaños de las matrices. En los resultados obtenidos se observó que con el aumento en el número de procesadores el speedup observado presentó una tendencia de decremento y que al aumentar los tamaños de las matrices de entrada el speedup observado presento una tendencia incremental, y que estas mismas tendencias tanto para los Speedups como para las eficiencias se presento en el ambiente homogéneo con el programa paralelo de LU mostrado en la sección 5.4.1. En el siguiente capítulo se presentan las conclusiones y recomendaciones para trabajos futuros alrededor de esta tesis.

Capítulo 6

Conclusiones.

Hoy en día las aplicaciones bancarias y científicas altamente demandantes en poder de cómputo requieren de técnicas funcionales de desarrollo de aplicaciones que puedan explotar al máximo la infraestructura de computo que posean, dicha estructura podría estar integrada por máquina paralelas y estaciones de trabajo. Ante este escenario se vuelve imperativo desarrollar trabajos al respecto que sienten un marco de referencia que sirvan de guía para desarrollar tales aplicaciones.

Los resultados generados a partir del desarrollo de este trabajo sentaran las bases para la paralelización del modelo MULTIFLO, ya que de manera exitosa se logro paralelizar uno de sus algoritmos base, descomposición de matrices mediante la técnica LU, obteniendo Speedups hasta de 2.31 y eficiencias de hasta el 76% en nuestro ambiente heterogéneo.

Observamos a través de toda la experimentación que el programa paralelo de LU generó buenos resultados en el ambiente de 4 estaciones de trabajo y que el Speedup y la eficiencia mejoraron mientras más grande fue la *matriz A*. Recordemos que las características de las estaciones de trabajo en el ambiente de prueba de 4, son similares y que por lo tanto asignarles la misma cantidad de trabajo es válido y óptimo, siempre y cuando, alguna de las máquinas no se cargue a tal grado que pierda la similitud con el resto de las estaciones de trabajo. La tendencia de mejora en el Speedup y por lo tanto en la eficiencia, se debe a que el overhead de la transferencia de información se vuelve menos significativa mientras mas grande es la matriz A. En conclusión se afirma que el algoritmo paralelo LU es bueno, con una eficiencia de hasta el 76%, en el ambiente con estaciones de trabajo similares, sin embargo este algoritmo puede será totalmente ineficiente en el caso en que una o mas estaciones de trabajo se carguen de tal manera que se pierda la similitud o se incremente la carga de la red.

Otras observaciones importantes de la experimentación son que el programa paralelo LU no dio buenos resultados en los ambientes de 5, 6 y 7 estaciones de trabajo, para todos los tamaños de la matriz de entrada A. Que no se obtuvieron Speedups, pues los valores fluctuaron de entre 0.54 y 0.91. La eficiencia aunque pésima en sus resultados, se notó una pequeña tendencia incremental, del 16 al 25, al aumentar la dimensión de la matriz de entrada A y muy poca diferencia al aumentar los nodos, de apenas uno cuantos puntos porcentuales. En conclusión se afirma que el programa paralelo LU no es óptimo para ambientes heterogéneos de estaciones de trabajo, pues no se adapta a los cambios en el sistema.

A través de la experimentación observamos que el programa paralelo balanceado de LU dio buenos resultados en todos los ambientes de prueba. Se mostró que para matrices de 512 la mejora relativa de los tiempos de ejecución del programa paralelo balanceado con respecto del programa paralelo va desde 1.12 veces mejor hasta 1.78, para matrices de 1024 la mejora relativa va desde 1.06 veces mejor hasta 2.54 y que para matrices de 2048 la mejora relativa va desde 1.08 veces mejor hasta 2.93. Los valores del Speedup mantuvieron una tendencia incremental, que fueron del 1.31 al 2.32, al aumentar el tamaño de la matriz A y una tendencia decremental al aumentar los nodos en todos los ambientes de pruebas. La eficiencia también mantuvo una tendencia incremental relativa del programa balanceado, *(Eb)*, con respecto del programa paralelo, *(Ep)*, que va desde 12% hasta 114%, que para matrices de 512, del 6% hasta el 154% para matrices de 1024 y por último del 8% hasta el 193%, para matrices de 2048. En conclusión se afirma que el programa paralelo balanceado de LU es significativamente mejor que el programa paralelo, ya que obtuvo mejoras relativas de hasta el 193% en la eficiencia comparado con el programa paralelo para todos los ambientes de pruebas.

Con la experimentación también se determinó la frecuencia de balanceo óptima para el algoritmo paralelo balanceado de LU para todos nuestros ambientes de prueba y para todos los tamaño de las matrices. En base a estos experimentos podemos concluir que dada las características tecnológicas de las estaciones de trabajo de prueba y de los equipos de comunicación el balanceo es óptimo si se realiza cada 30 iteraciones.

La experimentación también determinó que no es óptimo predecir el comportamiento de las cargas en las estaciones de trabajo, sino que lo mejor es usar la información actual de las cargas en el momento determinado por la frecuencia de balanceo para distribuir las cargas entre las estaciones de trabajo.

El Speedup óptimo heterogéneo es una medida determinante para el algoritmo paralelo balanceado de LU, ya que sirve de comparación para el Speedup alcanzado de este algoritmo, es por ello que se realizaron los cálculos (como se mostró en la sección 5.3.2) necesarios para encontrar dicho Speedup óptimo heterogéneo para nuestro ambiente en particular resultando en sus valores de 3.05, 3.19, 3.34 y 3.48 para nuestros ambientes de 4, 5, 6 y 7 estaciones de trabajo respectivamente.

Las conclusiones presentadas en éste capítulo son válidas para nuestro ambiente heterogéneo de estaciones de trabajo interconectadas a través de una red Ethernet.

6.1 Trabajos Futuros.

Se realizan las recomendaciones siguientes para encontrar el crecimiento en el Speedup.

La sincronización de los tiempos de comunicación, overhead de balanceo y de procesamiento es muy costoso, ya que todas las estaciones de trabajo deben actualizar tales tiempos, por lo que se recomienda modificar el algoritmo de balanceo para hacerlo menos distribuido y formar grupos de estaciones de trabajo que se sincronicen entre ellas con el objeto realizar pruebas para determinar si se reduce la comunicación global.

Se recomienda modificar el algoritmo de balanceo de tal manera que elimine las máquinas demasiado lentas, es decir, que se determine un límite mediante experimentación que establesca que una estación es demasiado lenta con respectos del resto y que por lo tanto es óptimo no utilizarla.

Se recomienda utilizar un algoritmo que solucione LU que sea escalable con el aumento en el número de estaciones de trabajo.

Se recomienda aplicar la técnica usada en otras aplicaciones similares a LU en lo referente a la alta dependencia de datos, a sí como también aplicarla en otras aplicaciones con características diferentes para conocer el desempeño y poder hacer una generalización de la técnica usada y pueda servir como una alternativa para implementar balanceo dinámico.

Se recomienda realizar pruebas en redes de área amplia para observar el desempeño arrojado por el algoritmo de balanceo aplicado en LU.

Se recomienda realizar pruebas utilizando tecnologías de red de área local con mayor ancho de banda como por ejemplo Fast Ethernet y FDDI.

Se recomienda realizar un análisis de la frecuencia de balanceo en diferentes ambientes y tratar de establecer un patrón, para emitir una recomendación de dicho valor en un ambiente dado.

Referencias Bibliográficas.

[Beck96] Beckman, P., Gannon, D. y Johnson E.,"Portable Parallel Programming in HPC++", Department of Computer Science, Indiana University, 1996.

[Diek98] Diekmann R., Monien B., Preis P., "Load Balancing Strategies for Distributed Memory Machines", AG-Monien, 1998.

[Envi92] Environmental Plan for the Mexican-US Border Area, First Stage (1992-1994), EPA Washington, 1992.

[Fost95] Foster, Ian.,"Designing and building Parallel Programs: Concepts and tools for parellel software engineering", Addison Wesley, 1995.

[Hesh98] Hesham El-Rewini, Ted G., "Distributed and Parallel Computing", Manning, 1998.

[Kuck94] Kuck, D.,"What Do Users of Parallel Computer Systems Really Need?", International Journal of Parallel Programming, Vol 22, No. 1, 1994.

[Mahe96] Maheshwari P.,"A dynamic Load Balancing Algorithm for a Heterogeneous Computing Environment",IEEE Computer Society Press,pp 338-291, 1996.

[Mult96] Multiflo User's Manual: Part I, Metra 1.0B, Two-Phase Nonisothermal Flow Simulator, M.S. Seth and P.C.Lichtner, CNWRA 96-005, 1996.

[Quin94] Quinn M.," Parallel Computing: Theory and Practice", McGraw Hill International,1994.

[Shir95] Shirazi, Behrooz A.,"Scheduling and load balancing in parallel and distributed systems", IEEE Computer Society Press, 1995.

[Zaki95] Zaki M.,Li W., Parthasarathy,"Cuztomized Dynamic Load Balancing for a Network of Workstation", IEEE Computer Society Press,pp 282-290, 1996.

Apéndice

Programa Secuencial de LU

```
/*                          Este programa resuelve el algoritmo LU de manera secuencial.
                                Este programa se ejecuta: lus < matrizA                         */
#include <stdio.h>
#include "mpi.h"
#include <math.h>
#include <sys/times.h>

float A[2048][2048],LU[2048][2048],sum;
double startwtime,endwtime;
struct tms buff1, buff2;
clock_t clock1, clock2;

main(int argc, char *argv[]){
int i,j,k,n;

    MPI_Init(&argc,&argv);
    scanf("%d",&n);                   /* Lee el número de filas de A */
    for (i = 0 ; i < n; i++){
      for ( j = 0 ; j < n; j++){      /* Inicializa valores */
        A[i][j] = 0.0;
        LU[i][j] = 0.0;
      }
    };

    for ( i = 0; i < n; i++){                /* lee la matriz A */
      for ( j = 0; j < n; j++)
        scanf("%f",&A[i][j]);
    };
    clock1=times(&buff1);
    for ( k = 0; k < n; k++){                /* comienza LU(A) */
      LU[k][k] = A[k][k];
      for ( i = k + 1; i < n; i++){
        LU[i][k] = A[i][k] / LU[k][k];
        LU[k][i] = A[k][i];
      };
      for ( i = k + 1; i < n; i++){
        for ( j = k + 1; j < n; j++)
          A[i][j] = A[i][j] - LU[i][k] * LU[k][j];
      };

    };
    clock2=times(&buff2);
    /* for ( i = 0; i < n; i++){
      for ( j = 0; j < n; j++){
        printf("%f ",A[i][j]);
      }
      printf("\n");
```

```
    };
    printf("\n");

    for ( i = 0; i < n; i++){
      for ( j = 0; j < n; j++){
        printf("%f ",LU[i][j]);
      }
      printf("\n");
    }; */
    printf("Wall clock time S = %f\n",(double)clock1);
    printf("Wall clock time E = %f\n",(double)clock2);
    printf("Wall clock time   = %f\n",(double)(clock2-clock1));
    MPI_Finalize();

}
```

Programa paralelo de LU

```
/* Esta es la implementacion paralela del algoritmo A=LU */

#include<stdlib.h>
#include <stdio.h>
#include "mpi.h"
#include <math.h>
#include <sys/times.h>

float **matriz(int m, int n){
int i;
float **ptr;
  ptr = (float **)calloc(m, sizeof(float *));
  for (i=0; i<m;i++){
      ptr[i]=(float *)calloc(n, sizeof(float));
  }
  return (ptr);
}

int *vector(int m){
int *ptr;
   ptr=(int *)calloc(m, sizeof(int));
   return (ptr);
}

int main(argc,argv)
int argc;
char *argv[];
{
 float **A, **LU,sum;
 div_t r;
 MPI_Status status;
 int myid, numprocs,dim,dim_k_iteracion,i,j,k,k_iteracion,tag,proc_enviar;
 int *posr_pfpi_ant,*posr_ufpi_ant,*posr_pfpi,*posr_ufpi,*nfpi;
 MPI_Request req;
 struct tms buff1,buff2;
 clock_t clock1,clock2;

 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
 MPI_Comm_rank(MPI_COMM_WORLD,&myid);
 if(myid == 0){  /******* Empieza la lectura y disribucion de los datos ***********/
   scanf("%d",&dim);
 }
 MPI_Bcast(&dim,1,MPI_INT,0,MPI_COMM_WORLD);
 A  = matriz(dim,dim);
 LU = matriz(dim,dim);
 posr_pfpi=vector(numprocs);
 posr_ufpi=vector(numprocs);
 posr_pfpi_ant=vector(numprocs);
 posr_ufpi_ant=vector(numprocs);
 nfpi=vector(numprocs);
 if(myid == 0){
   for(i=0;i<dim;i++){
```

```
    for(j=0;j<dim;j++){
      scanf("%f",&sum);
      A[i][j]=sum;

    }
  }
}
k_iteracion=0;
dim_k_iteracion = dim - k_iteracion;
if (numprocs >= dim_k_iteracion){
  nfpi[0]=dim_k_iteracion;
  for(k=1;k<numprocs;k++){
    nfpi[k]=0;
  }
}else{
  r=div(dim_k_iteracion,numprocs);
  for(k=0;k<numprocs;k++){
    nfpi[k]=r.quot;
  }
  for(k=0;k<r.rem;k++){
    nfpi[k]=nfpi[k] + 1;
  }
}
posr_pfpi_ant[0]=k_iteracion;
posr_ufpi_ant[0]=nfpi[0]-1;
for(k=1;k<numprocs;k++){
  posr_pfpi_ant[k] = posr_ufpi_ant[k-1]+1;
  posr_ufpi_ant[k] = posr_pfpi_ant[k]+nfpi[k]-1;
}
              /* Se Distribuye la Matriz A */
if(myid == 0){
  for(i=1;i<numprocs;i++){
    for(j=posr_pfpi_ant[i];j<=posr_ufpi_ant[i];j++){
      tag=i+j;
      MPI_Send(&A[j][0],dim,MPI_FLOAT,i,tag,MPI_COMM_WORLD);
    }
  }
}
else{
  if(myid < numprocs){
    for(j=posr_pfpi_ant[myid];j<=posr_ufpi_ant[myid];j++){
     tag=j+myid;
     MPI_Recv(&A[j][0],dim,MPI_FLOAT,0,tag,MPI_COMM_WORLD,&status);
    }
  }
}/*se recibe la parte de A por el Procesador j */
clock1=times(&buff1);
            /********* Empieza la descomposicion LU ****************/
for(k_iteracion = 0;k_iteracion < dim; k_iteracion++){     /* k_iteracion */
 dim_k_iteracion = dim - k_iteracion;
 if (numprocs >= dim_k_iteracion){
   nfpi[0]=dim_k_iteracion;
   for(k=1;k<numprocs;k++){
     nfpi[k]=0;
   }
```

```
}else{
  r=div(dim_k_iteracion,numprocs);
  for(k=0;k<numprocs;k++){
    nfpi[k]=r.quot;
  }
  for(k=0;k<r.rem;k++){
     nfpi[k]=nfpi[k]+1;
  }
}/*
if(myid==0){
 for(k=0;k<numprocs;k++){
   printf("k%d nfpi%d\n",k_iteracion,nfpi[k]);
 }
}*/
posr_pfpi[0]=k_iteracion;
posr_ufpi[0]=k_iteracion+nfpi[0]-1;
for(k=1;k<numprocs;k++){
   posr_pfpi[k]=posr_ufpi[k-1]+1;
   posr_ufpi[k]=posr_pfpi[k]+nfpi[k]-1;
}
proc_enviar=myid;
for(i=posr_pfpi_ant[myid];i<=posr_ufpi_ant[myid];i++){
   for(k=0;k<numprocs;k++){
     if( (i >= posr_pfpi[k]) && (i <= posr_ufpi[k]) ){ proc_enviar=k; }
   }
   if(myid != proc_enviar){
     tag=i;

MPI_Send(&A[i][k_iteracion],dim_k_iteracion,MPI_FLOAT,proc_enviar,tag,MPI_COMM_WORLD);
   }
}
proc_enviar=myid;
for(i=posr_pfpi[myid];i<=posr_ufpi[myid];i++){
   for(k=0;k<numprocs;k++){
     if( (i >= posr_pfpi_ant[k]) && (i <= posr_ufpi_ant[k]) ){ proc_enviar=k;}
   }
   if(myid != proc_enviar){
     tag=i;

MPI_Recv(&A[i][k_iteracion],dim_k_iteracion,MPI_FLOAT,proc_enviar,tag,MPI_COMM_WORLD,&statu
s);
   }
}/*
if(myid==0){
 for(k=0;k<numprocs;k++){
    printf("k%d pfpi%d ufpi%d pfpi_ant%d
ufpi_ant%d\n",k_iteracion,posr_pfpi[k],posr_ufpi[k],posr_pfpi_ant[k],posr_ufpi_ant[k]);
 }
}*/
for(k=0;k<numprocs;k++){
  posr_pfpi_ant[k]=posr_pfpi[k];
  posr_ufpi_ant[k]=posr_ufpi[k];
}
if(myid == 0){
 for(i=1;i<numprocs;i++){
```

```
      tag=i;
      MPI_Send(&A[k_iteracion][k_iteracion], dim_k_iteracion,MPI_FLOAT,i,tag,MPI_COMM_WORLD);
    }
    for( i = k_iteracion; i < dim; i++){
      LU[k_iteracion][i] = A[k_iteracion][i];
    };
    for( i = k_iteracion + 1; i <= posr_ufpi[myid]; i++){
      LU[i][k_iteracion] = A[i][k_iteracion] / LU[k_iteracion][k_iteracion];
    }
    for( i = k_iteracion + 1; i <= posr_ufpi[myid]; i++){
      for( j = k_iteracion + 1; j < dim; j++){
        A[i][j] = A[i][j] - LU[i][k_iteracion]* LU[k_iteracion][j];
      }
    }
  }else{
    if(myid < numprocs){
      tag=myid;
MPI_Recv(&LU[k_iteracion][k_iteracion],dim_k_iteracion,MPI_FLOAT,0,tag,MPI_COMM_WORLD,&stat
us);
      for( i = posr_pfpi[myid]; i <= posr_ufpi[myid]; i++){
        LU[i][k_iteracion] = A[i][k_iteracion] / LU[k_iteracion][k_iteracion];
      };
      for( i = posr_pfpi[myid]; i <= posr_ufpi[myid]; i++){
        for( j = k_iteracion + 1; j < dim; j++)
          A[i][j] = A[i][j] - LU[i][k_iteracion] * LU[k_iteracion][j];
      };
    }
  }
}          /* Termina la k_iteracion */
 MPI_Barrier(MPI_COMM_WORLD); /* se Inicia la Impresion de la Matriz Resultante */
 if(myid == 0){
    clock2=times(&buff2);
 }/*
 for(i=0;i<dim;i++){
    for(j=0;j<dim;j++){
      printf("%dA[%d][%d]=%f  ",myid,i,j,A[i][j]);
    }
    printf("\n");
 }
 printf("\n");
 for(i=0;i<dim;i++){
   for(j=0;j<dim;j++){
     printf("%dLU[%d][%d]=%f  ",myid,i,j,LU[i][j]);
   }
   printf("\n");
 }*/
 if(myid==0){
  printf("\n WALL CLOCK TIME S = %f\n",(double) clock1);
  printf("\n WALL CLOCK TIME E = %f\n",(double) clock2);
  printf("\n WALL CLOCK TIME   = %f\n",(double) (clock2-clock1));
 }
 MPI_Finalize();
}
```

Programa paralelo balanceado de LU

```
/* Esta es la implementacion paralela del algoritmo A=LU */

#include<stdlib.h>
#include <stdio.h>
#include "mpi.h"
#include <math.h>
#include <sys/types.h>
#include <sys/times.h>
#include <time.h>

float **matriz(int m, int n){
int i;
float **ptr;
  ptr = (float **)calloc(m, sizeof(float *));
  for (i=0; i<m;i++){
      ptr[i]=(float *)calloc(n, sizeof(float));
  }
  return (ptr);
}

float *vectorf(int m){
float *ptr;
   ptr=(float *)calloc(m, sizeof(float));
   return (ptr);
}

int *vector(int m){
int *ptr;
   ptr=(int *)calloc(m, sizeof(int));
   return (ptr);
}

struct tabla_indices{
 int proc;
 int e;
 int posr_pfpi;
 int posr_ufpi;
};


struct tabla_indices **matriz_tabla(int m, int n){
int i;
struct tabla_indices **ptr;
  ptr = (struct tabla_indices **)calloc(m, sizeof(struct tabla_indices *));
  for (i=0; i<m;i++){
      ptr[i]=(struct tabla_indices *)calloc(n, sizeof(struct tabla_indices));
  }
  return (ptr);
}

int main(argc,argv)
int argc;
char *argv[];
```

```
{
 float **A, **LU,sum,*vec_tiempos,*vec_tiempos_h,*vec_tiempos_ant,trono,*trabajo;
 div_t r;
 MPI_Status status;
 int myid,sum1,numprocs,dim,dim_k_iteracion,i,j,k,k_iteracion,tag,proc_enviar,pos,band,mide_tiempo;
 int *nfpi,*nfpi_ant,*diff,intervalo,n_veces,n,m,pos_c,x;
 double t_ini_calculos,t_fin_calculos,ent,fracc,t_ini_over,t_fin_over;
 MPI_Request req;
 struct tms buff1,buff1_1, buff2,buff2_1;
 struct tabla_indices **t_index;
 clock_t clock1,clock1_1, clock2,clock2_1;
 time_t tiempo;

 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
 MPI_Comm_rank(MPI_COMM_WORLD,&myid);
 if(myid == 0){  /******* Empieza la lectura y disribucion de los datos ***********/
   scanf("%d",&dim);
 }
 MPI_Bcast(&dim,1,MPI_INT,0,MPI_COMM_WORLD);
 A  = matriz(dim,dim);
 LU = matriz(dim,dim);
 t_index = matriz_tabla(numprocs,dim);
 nfpi=vector(numprocs);
 diff=vector(numprocs);
 nfpi_ant=vector(numprocs);
 trabajo=vectorf(numprocs);
 vec_tiempos=vectorf(numprocs);
 vec_tiempos_h=vectorf(numprocs);
 vec_tiempos_ant=vectorf(numprocs);
 if(myid == 0){
  for(i=0;i<dim;i++){
    for(j=0;j<dim;j++){
      scanf("%f",&sum);
      A[i][j]=sum;

    }
  }
 }
 vec_tiempos[0]=1.0;  /* Inicialización de los tiempos, para el cálculo del particionamiento de A*/
 vec_tiempos[1]=1.0;
 vec_tiempos[2]=1.0;
 vec_tiempos[3]=1.0;
 vec_tiempos[4]=1.0;
 vec_tiempos[5]=1.0;
 vec_tiempos[6]=1.0;
 k_iteracion=0;
 dim_k_iteracion = dim - k_iteracion;  /* inician los calculos para el particionamiento de A*/
 trono=999999999;
 pos=0;
 for(k=0;k<numprocs;k++){
   if(vec_tiempos[k]!=-1){
     if(vec_tiempos[k]<trono){
       trono=vec_tiempos[k];
       pos=k;
```

```
      }
    }
  }/*
  if(myid==0){
    printf("k%d\nMenor%f pos%d \n",k_iteracion,vec_tiempos[pos],pos);
    for(k=0;k<numprocs;k++){
      printf("vec_tiempos %f nfpi %d \n",vec_tiempos[k],nfpi[k]);
    }
  }*/
  sum=0;
  for(k=0;k<numprocs;k++){
     if(vec_tiempos[k]!=-1){
       trabajo[k]=trono/vec_tiempos[k];
       sum=sum+trabajo[k];
     }
  }
  sum1=0;
  for(k=0;k<numprocs;k++){
     if(vec_tiempos[k]!=-1){
       fracc=modf((trabajo[k]/sum)*dim_k_iteracion, &ent);
       nfpi[k]=ent;
       sum1=sum1+nfpi[k];
     }else{
       nfpi[k]=0;
     }
  }/*
  if(myid==0){
    for(k=0;k<numprocs;k++){
      printf("k%d nfpi%d\n",k_iteracion,nfpi[k]);
    }
  }*/
  k=0;
  while((dim_k_iteracion-sum1) && (k<numprocs)){
     if(nfpi[k]==0){
       nfpi[k]=1;
       sum1++;
     }
     k++;
  }
  k=0;
  while((dim_k_iteracion-sum1) && (k<numprocs)){
      if(((fracc=modf((trabajo[k]/sum)*dim_k_iteracion, &ent)) > 0.7) && ent!=0){
        nfpi[k]=nfpi[k]+1;
        sum1++;
      }
      k++;
  }
  k=0;
  while((dim_k_iteracion-sum1) && (k<numprocs)){
     if(((fracc=modf((trabajo[k]/sum)*dim_k_iteracion, &ent))>=0.3)&&(fracc<=0.7)&&(ent!=0)){
       nfpi[k]=nfpi[k]+1;
       sum1++;
     }
     k++;
  }
```

```
 k=0;
 while((dim_k_iteracion-sum1) && (k<numprocs)){
      if( (modf((trabajo[k]/sum)*dim_k_iteracion, &ent) < 0.3)&&(ent!=0)){
         nfpi[k]=nfpi[k]+1;
         sum1++;
      }
      k++;
 }

/* Empieza el llenado de la tabla de elementos de los segmentos de la matriz A, que define cada segmento
dentro de A. */
 if(nfpi[myid]==0) mide_tiempo=0;
 else mide_tiempo=1;
 t_index[0][0].proc=0;
 t_index[0][0].e=0;
 t_index[0][0].posr_pfpi=k_iteracion;
 t_index[0][0].posr_ufpi=nfpi[0]-1;
 t_index[0][1].proc=-1;
 for(k=1;k<numprocs;k++){
    t_index[k][0].proc=k;
    t_index[k][0].e=0;
    t_index[k][0].posr_pfpi=t_index[k-1][0].posr_ufpi + 1;
    t_index[k][0].posr_ufpi=t_index[k][0].posr_pfpi + nfpi[k]-1;
    t_index[k][1].proc=-1;

 }
 for(k=0;k<numprocs;k++){
   nfpi_ant[k]=nfpi[k];
 }



                /* Se Distribuye la Matriz A */
 if(myid == 0){
   for(i=1;i<numprocs;i++){
      for(j=t_index[i][0].posr_pfpi;j<=t_index[i][0].posr_ufpi;j++){
         tag=i+j;
         MPI_Isend(&A[j][0],dim,MPI_FLOAT,i,tag,MPI_COMM_WORLD,&req);
      }
   }
 }
 else{
   if(myid < numprocs){
      for(j=t_index[myid][0].posr_pfpi;j<=t_index[myid][0].posr_ufpi;j++){
       tag=j+myid;
       MPI_Recv(&A[j][0],dim,MPI_FLOAT,0,tag,MPI_COMM_WORLD,&status);
      }
   }
 }/*se recibe la parte de A por el Procesador j */
 clock1=times(&buff1);
 t_ini_over=t_fin_over=0;
 n_veces=1;
 intervalo=30;
/* intervalo=dim/intervalo;*/
              /********* Empieza la descomposicion LU ***************/
```

```
for(k_iteracion = 0;k_iteracion < dim; k_iteracion++){     /* k_iteracion */
 t_ini_over=MPI_Wtime();
 if( k_iteracion == (intervalo*n_veces) || k_iteracion==2){  /* Balanceo */
   for(k=0;k<numprocs;k++){
     MPI_Bcast(&vec_tiempos_h[k],1,MPI_FLOAT,k,MPI_COMM_WORLD);
     vec_tiempos[k]=vec_tiempos_h[k];
   }
   n_veces++;
 }                                              /*Particionamiento de A*/
 dim_k_iteracion = dim - k_iteracion;
  if(k_iteracion){
    trono=999999999;
    pos=0;
    for(k=0;k<numprocs;k++){
      if(vec_tiempos[k]!=-1){
       if(vec_tiempos[k]<trono){
        trono=vec_tiempos[k];
        pos=k;
       }
      }
    }/*
    if(myid==0){
     printf("k%d\nMenor%f pos%d \n",k_iteracion,vec_tiempos[pos],pos);
     for(k=0;k<numprocs;k++){
       printf("vec_tiempos %f nfpi %d \n",vec_tiempos[k],nfpi[k]);
     }
    }*/
    sum=0;
    for(k=0;k<numprocs;k++){
      if(vec_tiempos[k]!=-1){
        trabajo[k]=trono/vec_tiempos[k];
        sum=sum+trabajo[k];
      }
    }
    sum1=0;
    for(k=0;k<numprocs;k++){
      if(vec_tiempos[k]!=-1){
       fracc=modf((trabajo[k]/sum)*dim_k_iteracion, &ent);
       nfpi[k]=ent;
       sum1=sum1+nfpi[k];
      }else{
       nfpi[k]=0;
      }
    }/*
    if(myid==0){
     for(k=0;k<numprocs;k++){
       printf("k%d nfpi%d\n",k_iteracion,nfpi[k]);
     }
    }*/
    k=0;
    while((dim_k_iteracion-sum1) && (k<numprocs)){
       if(nfpi[k]==0){
           nfpi[k]=1;
           sum1++;
       }
```

```
        k++;
      }
      k=0;
      while((dim_k_iteracion-sum1) && (k<numprocs)){
        if(((fracc=modf((trabajo[k]/sum)*dim_k_iteracion, &ent)) > 0.7) && ent!=0){
            nfpi[k]=nfpi[k]+1;
            sum1++;
        }
        k++;
      }
      k=0;
      while((dim_k_iteracion-sum1) && (k<numprocs)){
       if(((fracc=modf((trabajo[k]/sum)*dim_k_iteracion,&ent))>=0.3)&&(fracc<=0.7)&&(ent!=0)){
            nfpi[k]=nfpi[k]+1;
            sum1++;
       }
       k++;
      }
      k=0;
      while((dim_k_iteracion-sum1) && (k<numprocs)){
        if( (modf((trabajo[k]/sum)*dim_k_iteracion, &ent)  <  0.3)&&(ent!=0)){
            nfpi[k]=nfpi[k]+1;
            sum1++;
        }
        k++;
      }
    }
  if(nfpi[myid]==0) mide_tiempo=0;
  else mide_tiempo=1;
  for(k=0;k<numprocs;k++){diff[k]=nfpi[k]-nfpi_ant[k];}
  for(k=0;k<numprocs;k++){
    j=k+1;     /* Inicia el llenado de la tabla de elementos de los segmentos de la  matriz A                de los
segmentos de cada procesador */
    while((diff[k]!=0) && (j<numprocs)){
      if(diff[k]>0){
       if(diff[j]<0){
        sum1=diff[k]+diff[j];
        if(sum1>0){
         band=1;
         m=0;
         while((t_index[j][m].proc !=-1) && (diff[j]<0)){
          if(t_index[j][m].proc==j){
           if(t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1+diff[j]==0){
             n=0;
             while(t_index[k][n].proc!=-1){n++;}
             t_index[k][n].proc=j;
             t_index[k][n].e=0;
             t_index[k][n].posr_pfpi=t_index[j][m].posr_pfpi;
             t_index[k][n].posr_ufpi=t_index[j][m].posr_ufpi;
             t_index[k][n+1].proc=-1;
             t_index[j][m].proc=k;
             t_index[j][m].e=1;
             band=0;
             diff[k]=diff[k]-(t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1);
             diff[j]=t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1+diff[j];
```

```
    }else if(t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1+diff[j]>0){
     n=0;
     while(t_index[k][n].proc!=-1){n++;}
     t_index[k][n].proc=j;
     t_index[k][n].e=0;
     t_index[k][n].posr_pfpi=t_index[j][m].posr_ufpi+diff[j]+1;
     t_index[k][n].posr_ufpi=t_index[j][m].posr_ufpi;
     t_index[k][n+1].proc=-1;
     n=0;
     while(t_index[j][n].proc!=-1){n++;}
     t_index[j][n].proc=k;
     t_index[j][n].e=1;
     t_index[j][n].posr_pfpi=t_index[j][m].posr_ufpi+diff[j]+1;
     t_index[j][n].posr_ufpi=t_index[j][m].posr_ufpi;band=n;
     t_index[j][n+1].proc=-1;
     t_index[j][m].proc=j;
     t_index[j][m].posr_ufpi=t_index[j][m].posr_ufpi+diff[j];
     diff[k]=diff[k]-(t_index[j][band].posr_ufpi-t_index[j][band].posr_pfpi+1);
     diff[j]=t_index[j][band].posr_ufpi-t_index[j][band].posr_pfpi+1+diff[j];
    }else{
     n=0;
     while(t_index[k][n].proc!=-1){n++;}
     t_index[k][n].proc=j;
     t_index[k][n].e=0;
     t_index[k][n].posr_pfpi=t_index[j][m].posr_pfpi;
     t_index[k][n].posr_ufpi=t_index[j][m].posr_ufpi;
     t_index[k][n+1].proc=-1;
     t_index[j][m].proc=k;
     t_index[j][m].e=1;
     diff[j]=t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1+diff[j];
     diff[k]=diff[k]-(t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1);
     band=1;m++;
    }
   }else{m++;}
  }
 }else if(sum1<0){
  band=1;
  m=0;
  while((t_index[j][m].proc !=-1) && (diff[k]>0)){
   if(t_index[j][m].proc==j){
    if(t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1-diff[k]==0){
     n=0;
     while(t_index[k][n].proc!=-1){n++;}
     t_index[k][n].proc=j;
     t_index[k][n].e=0;
     t_index[k][n].posr_pfpi=t_index[j][m].posr_pfpi;
     t_index[k][n].posr_ufpi=t_index[j][m].posr_ufpi;
     t_index[k][n+1].proc=-1;
     t_index[j][m].proc=k;
     t_index[j][m].e=1;
     band=0;
     diff[k]=diff[k]-(t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1);
     diff[j]=t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1+diff[j];
    }else if(t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1-diff[k]>0){
     n=0;
```

```
        while(t_index[k][n].proc!=-1){n++;}
        t_index[k][n].proc=j;
        t_index[k][n].e=0;
        t_index[k][n].posr_pfpi=t_index[j][m].posr_ufpi-diff[k]+1;
        t_index[k][n].posr_ufpi=t_index[j][m].posr_ufpi;
        t_index[k][n+1].proc=-1;
        n=0;
        while(t_index[j][n].proc!=-1){n++;}
        t_index[j][n].proc=k;
        t_index[j][n].e=1;
        t_index[j][n].posr_pfpi=t_index[j][m].posr_ufpi-diff[k]+1;
        t_index[j][n].posr_ufpi=t_index[j][m].posr_ufpi;
        band=n;
        t_index[j][n+1].proc=-1;
        t_index[j][m].proc=j;
        t_index[j][m].posr_ufpi=t_index[j][m].posr_ufpi-diff[k];
        diff[k]=diff[k]-(t_index[j][band].posr_ufpi-t_index[j][band].posr_pfpi+1);
        diff[j]=t_index[j][band].posr_ufpi-t_index[j][band].posr_pfpi+1+diff[j];
       }else{
        n=0;
        while(t_index[k][n].proc!=-1){n++;}
        t_index[k][n].proc=j;
        t_index[k][n].e=0;
        t_index[k][n].posr_pfpi=t_index[j][m].posr_pfpi;
        t_index[k][n].posr_ufpi=t_index[j][m].posr_ufpi;
        t_index[k][n+1].proc=-1;
        t_index[j][m].proc=k;
        t_index[j][m].e=1;
        diff[k]=diff[k]-(t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1);
        diff[j]=t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1+diff[j];
        band=1;m++;
       }
     }else{m++;}
   }  /* aqui */
  }else{
   band=1;
   m=0;
   while((t_index[j][m].proc !=-1) && (diff[k]>0)){
    if(t_index[j][m].proc==j){
     if(t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1-diff[k]==0){
        n=0;
        while(t_index[k][n].proc!=-1){n++;}
        t_index[k][n].proc=j;
        t_index[k][n].e=0;
        t_index[k][n].posr_pfpi=t_index[j][m].posr_pfpi;
        t_index[k][n].posr_ufpi=t_index[j][m].posr_ufpi;
        t_index[k][n+1].proc=-1;
        t_index[j][m].proc=k;
        t_index[j][m].e=1;
        band=0;
        diff[k]=diff[k]-(t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1);
        diff[j]=t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1+diff[j];
       }else if(t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1-diff[k]>0){
        n=0;
        while(t_index[k][n].proc!=-1){n++;}
```

```
        t_index[k][n].proc=j;
        t_index[k][n].e=0;
        t_index[k][n].posr_pfpi=t_index[j][m].posr_ufpi-diff[k]+1;
        t_index[k][n].posr_ufpi=t_index[j][m].posr_ufpi;
        t_index[k][n+1].proc=-1;
        n=0;
        while(t_index[j][n].proc!=-1){n++;}
        t_index[j][n].proc=k;
        t_index[j][n].e=1;
        t_index[j][n].posr_pfpi=t_index[j][m].posr_ufpi-diff[k]+1;
        t_index[j][n].posr_ufpi=t_index[j][m].posr_ufpi;
        band=n;
        t_index[j][n+1].proc=-1;
        t_index[j][m].proc=j;
        t_index[j][m].posr_ufpi=t_index[j][m].posr_ufpi-diff[k];
        diff[k]=diff[k]-(t_index[j][band].posr_ufpi-t_index[j][band].posr_pfpi+1);
        diff[j]=t_index[j][band].posr_ufpi-t_index[j][band].posr_pfpi+1+diff[j];
       }else{
        n=0;
        while(t_index[k][n].proc!=-1){n++;}
        t_index[k][n].proc=j;
        t_index[k][n].e=0;
        t_index[k][n].posr_pfpi=t_index[j][m].posr_pfpi;
        t_index[k][n].posr_ufpi=t_index[j][m].posr_ufpi;
        t_index[k][n+1].proc=-1;
        t_index[j][m].proc=k;
        t_index[j][m].e=1;
        diff[k]=diff[k]-(t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1);
        diff[j]=t_index[j][m].posr_ufpi-t_index[j][m].posr_pfpi+1+diff[j];
        band=1;m++;
       }
      }else{m++;}
     }  /* aqui */
    }
   }
  }else{
   if(diff[j]>0){
    sum1=diff[j]+diff[k];
    if(sum1>0){
     band=1;
     m=0;
     while((t_index[k][m].proc !=-1) && (diff[k]<0)){
      if(t_index[k][m].proc==k){
       if(t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1+diff[k]==0){
        n=0;
        while(t_index[j][n].proc!=-1){n++;}
        t_index[j][n].proc=k;
        t_index[j][n].e=0;
        t_index[j][n].posr_pfpi=t_index[k][m].posr_pfpi;
        t_index[j][n].posr_ufpi=t_index[k][m].posr_ufpi;
        t_index[j][n+1].proc=-1;
        t_index[k][m].proc=j;
        t_index[k][m].e=1;
        band=0;
        diff[j]=diff[j]-(t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1);
```

```
        diff[k]=t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1+diff[k];
      }else if(t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1+diff[k]>0){
        n=0;
        while(t_index[j][n].proc!=-1){n++;}
        t_index[j][n].proc=k;
        t_index[j][n].e=0;
        t_index[j][n].posr_pfpi=t_index[k][m].posr_ufpi+diff[k]+1;
        t_index[j][n].posr_ufpi=t_index[k][m].posr_ufpi;
        t_index[j][n+1].proc=-1;
        n=0;
        while(t_index[k][n].proc!=-1){n++;}
        t_index[k][n].proc=j;
        t_index[k][n].e=1;
        t_index[k][n].posr_pfpi=t_index[k][m].posr_ufpi+diff[k]+1;
        t_index[k][n].posr_ufpi=t_index[k][m].posr_ufpi;
        band=n;
        t_index[k][n+1].proc=-1;
        t_index[k][m].proc=k;
        t_index[k][m].posr_ufpi=t_index[k][m].posr_ufpi+diff[k];
        diff[j]=diff[j]-(t_index[k][n].posr_ufpi-t_index[k][n].posr_pfpi+1);
        diff[k]=t_index[k][band].posr_ufpi-t_index[k][band].posr_pfpi+1+diff[k];
      }else{
        n=0;
        while(t_index[j][n].proc!=-1){n++;}
        t_index[j][n].proc=k;
        t_index[j][n].e=0;
        t_index[j][n].posr_pfpi=t_index[k][m].posr_pfpi;
        t_index[j][n].posr_ufpi=t_index[k][m].posr_ufpi;
        t_index[j][n+1].proc=-1;
        t_index[k][m].proc=j;
        t_index[k][m].e=1;
        diff[k]=t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1+diff[k];
        diff[j]=diff[j]-(t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1);
        band=1;m++;
      }
    }else{m++;}
  }  /* aqui */
}else if(sum1<0){
  band=1;
  m=0;
  while((t_index[k][m].proc !=-1) && (diff[j]>0)){
    if(t_index[k][m].proc==k){
      if(t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1-diff[j]==0){
        n=0;
        while(t_index[j][n].proc!=-1){n++;}
        t_index[j][n].proc=k;
        t_index[j][n].e=0;
        t_index[j][n].posr_pfpi=t_index[k][m].posr_pfpi;
        t_index[j][n].posr_ufpi=t_index[k][m].posr_ufpi;
        t_index[j][n+1].proc=-1;
        t_index[k][m].proc=j;
        t_index[k][m].e=1;
        band=0;
        diff[j]=diff[j]-(t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1);
        diff[k]=t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1+diff[k];
```

```
      }else if(t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1-diff[j]>0){
        n=0;
        while(t_index[j][n].proc!=-1){n++;}
        t_index[j][n].proc=k;
        t_index[j][n].e=0;
        t_index[j][n].posr_pfpi=t_index[k][m].posr_ufpi-diff[j]+1;
        t_index[j][n].posr_ufpi=t_index[k][m].posr_ufpi;
        t_index[j][n+1].proc=-1;
        n=0;
        while(t_index[k][n].proc!=-1){n++;}
        t_index[k][n].proc=j;
        t_index[k][n].e=1;
        t_index[k][n].posr_pfpi=t_index[k][m].posr_ufpi-diff[j]+1;
        t_index[k][n].posr_ufpi=t_index[k][m].posr_ufpi;
        band=n;
        t_index[k][n+1].proc=-1;
        t_index[k][m].proc=k;
        t_index[k][m].posr_ufpi=t_index[k][m].posr_ufpi-diff[j];
        diff[j]=diff[j]-(t_index[k][band].posr_ufpi-t_index[k][band].posr_pfpi+1);
        diff[k]=t_index[k][band].posr_ufpi-t_index[k][band].posr_pfpi+1+diff[k];
      }else{
        n=0;
        while(t_index[j][n].proc!=-1){n++;}
        t_index[j][n].proc=k;
        t_index[j][n].e=0;
        t_index[j][n].posr_pfpi=t_index[k][m].posr_pfpi;
        t_index[j][n].posr_ufpi=t_index[k][m].posr_ufpi;
        t_index[j][n+1].proc=-1;
        t_index[k][m].proc=j;
        t_index[k][m].e=1;
        diff[j]=diff[j]-(t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1);
        diff[k]=t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1+diff[k];
        band=1;m++;
      }
    }else{m++;}
  } /* aqui */
}else{
  band=1;
  m=0;
  while((t_index[k][m].proc !=-1) && (diff[j]>0)){
    if(t_index[k][m].proc==k){
      if(t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1-diff[j]==0){
        n=0;
        while(t_index[j][n].proc!=-1){n++;}
        t_index[j][n].proc=k;
        t_index[j][n].e=0;
        t_index[j][n].posr_pfpi=t_index[k][m].posr_pfpi;
        t_index[j][n].posr_ufpi=t_index[k][m].posr_ufpi;
        t_index[j][n+1].proc=-1;
        t_index[k][m].proc=j;
        t_index[k][m].e=1;
        band=0;
        diff[j]=diff[j]-(t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1);
        diff[k]=t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1+diff[k];
      }else if(t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1-diff[j]>0){
```

```
            n=0;
            while(t_index[j][n].proc!=-1){n++;}
            t_index[j][n].proc=k;
            t_index[j][n].e=0;
            t_index[j][n].posr_pfpi=t_index[k][m].posr_ufpi-diff[j]+1;
            t_index[j][n].posr_ufpi=t_index[k][m].posr_ufpi;
            t_index[j][n+1].proc=-1;
            n=0;
            while(t_index[k][n].proc!=-1){n++;}
            t_index[k][n].proc=j;
            t_index[k][n].e=1;
            t_index[k][n].posr_pfpi=t_index[k][m].posr_ufpi-diff[j]+1;
            t_index[k][n].posr_ufpi=t_index[k][m].posr_ufpi;
            band=n;
            t_index[k][n+1].proc=-1;
            t_index[k][m].proc=k;
            t_index[k][m].posr_ufpi=t_index[k][m].posr_ufpi-diff[j];
            diff[j]=diff[j]-(t_index[k][band].posr_ufpi-t_index[k][band].posr_pfpi+1);
            diff[k]=t_index[k][band].posr_ufpi-t_index[k][band].posr_pfpi+1+diff[k];
           }else{
            n=0;
            while(t_index[j][n].proc!=-1){n++;}
            t_index[j][n].proc=k;
            t_index[j][n].e=0;
            t_index[j][n].posr_pfpi=t_index[k][m].posr_pfpi;
            t_index[j][n].posr_ufpi=t_index[k][m].posr_ufpi;
            t_index[j][n+1].proc=-1;
            t_index[k][m].proc=j;
            t_index[k][m].e=1;
            diff[j]=diff[j]-(t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1);
            diff[k]=t_index[k][m].posr_ufpi-t_index[k][m].posr_pfpi+1+diff[k];
            band=1;m++;
           }
          }else{m++;}
         } /* aqui */
        }
       }
      }
      j++;
    }
 }/* Inicia el intercambio de filas para completar los segmentos de A de cada procesador */
 t_fin_over=MPI_Wtime();
 t_ini_calculos=MPI_Wtime();
 n=0;
 while(t_index[myid][n].proc != -1){
  if(t_index[myid][n].proc != myid){
   if(t_index[myid][n].e){
    for(i=t_index[myid][n].posr_pfpi;i<=t_index[myid][n].posr_ufpi;i++){
     tag=i;

MPI_Isend(&A[i][k_iteracion],dim_k_iteracion,MPI_FLOAT,t_index[myid][n].proc,tag,MPI_COMM_WOR
LD,&req);
    }
   }else{
    for(i=t_index[myid][n].posr_pfpi;i<=t_index[myid][n].posr_ufpi;i++){
```

```
        tag=i;

MPI_Recv(&A[i][k_iteracion],dim_k_iteracion,MPI_FLOAT,t_index[myid][n].proc,tag,MPI_COMM_WOR
LD,&status);
       }
     }
   }
   n++;
  }      /* Inicia la actualización de la tabla de elementos de los segmentos de la matriz A */
  for(k=0;k<numprocs;k++){
    n=0;
    while(t_index[k][n].proc != -1){
     band=1;
     if(t_index[k][n].proc != k){
       if(t_index[k][n].e){
        m=n+1;
        while(t_index[k][m].proc !=-1){
          t_index[k][m-1].proc=t_index[k][m].proc;
          t_index[k][m-1].e=t_index[k][m].e;
          t_index[k][m-1].posr_pfpi=t_index[k][m].posr_pfpi;
          t_index[k][m-1].posr_ufpi=t_index[k][m].posr_ufpi;
          m++;band=0;
        }
        m--;
        t_index[k][m].proc=-1;
       }else{
        t_index[k][n].proc=k;
        if(t_index[k][n].posr_pfpi==k_iteracion){pos=k;pos_c=n;}
       }
     }else{
       if(t_index[k][n].posr_pfpi==k_iteracion){pos=k;pos_c=n;}
     }
     if(band){n++;}
    }
  } /*  Inicia las operaciones de LU */
  if(myid==pos){
   for(i=0;i<numprocs;i++){
    if(i!=pos){
      if(nfpi[i]!=0){
        tag=i;

MPI_Isend(&A[k_iteracion][k_iteracion],dim_k_iteracion,MPI_FLOAT,i,tag,MPI_COMM_WORLD,&req);
      }
    }
   }
  }else{
    if(nfpi[myid]!=0){
     tag=myid;

MPI_Recv(&LU[k_iteracion][k_iteracion],dim_k_iteracion,MPI_FLOAT,pos,tag,MPI_COMM_WORLD,&s
tatus);
    }
  }
  if(myid == pos){
   for( i = k_iteracion; i < dim; i++){
```

```
      LU[k_iteracion][i] = A[k_iteracion][i];
    };
    n=0;
    while(t_index[myid][n].proc!=-1){
     if(t_index[myid][n].proc==myid){
      if(t_index[myid][n].posr_pfpi==k_iteracion){
       for( i = t_index[myid][n].posr_pfpi+1; i <= t_index[myid][n].posr_ufpi; i++){
         LU[i][k_iteracion] = A[i][k_iteracion] / LU[k_iteracion][k_iteracion];
       }
       for( i = t_index[myid][n].posr_pfpi+1; i <= t_index[myid][n].posr_ufpi; i++){
         for( j = t_index[myid][n].posr_pfpi + 1; j < dim; j++){
          A[i][j] = A[i][j] - LU[i][k_iteracion]* LU[k_iteracion][j];
         }
       }
      }else{
       for( i = t_index[myid][n].posr_pfpi; i <= t_index[myid][n].posr_ufpi; i++){
         LU[i][k_iteracion] = A[i][k_iteracion] / LU[k_iteracion][k_iteracion];
       };
       for( i = t_index[myid][n].posr_pfpi; i <= t_index[myid][n].posr_ufpi; i++){
         for( j = k_iteracion + 1; j < dim; j++)
           A[i][j] = A[i][j] - LU[i][k_iteracion] * LU[k_iteracion][j];
       };
      }
     }
     n++;
    }
   }else{
    if(nfpi[myid]!=0){
     n=0;
     while(t_index[myid][n].proc !=-1){
      if(t_index[myid][n].proc==myid){
        for( i = t_index[myid][n].posr_pfpi; i <= t_index[myid][n].posr_ufpi; i++){
          LU[i][k_iteracion] = A[i][k_iteracion] / LU[k_iteracion][k_iteracion];
        };
        for( i = t_index[myid][n].posr_pfpi; i <= t_index[myid][n].posr_ufpi; i++){
          for( j = k_iteracion + 1; j < dim; j++)
           A[i][j] = A[i][j] - LU[i][k_iteracion] * LU[k_iteracion][j];
        };
      }
      n++;
     }
    }
   }/* Medición de los tiempos de comunicación, overhead y procesamiento */
 */
  t_fin_calculos=MPI_Wtime(); /*
  if(myid==3){
    printf("MPI_Wtick %lf diff %lf\n",MPI_Wtick(),(t_fin_calculos - t_ini_calculos));
  }*/
  if(k_iteracion==0){
      vec_tiempos_ant[myid] = (t_fin_calculos - t_ini_calculos);
      vec_tiempos_ant[myid] = (double)(((double) vec_tiempos_ant[myid])  / ((double) nfpi[myid])) ;
      vec_tiempos_ant[myid] = vec_tiempos_ant[myid] + (t_fin_over - t_ini_over);
      if(vec_tiempos_ant[myid]==0.0) vec_tiempos_ant[myid]=0.000001;
  }else{
   if(mide_tiempo){
```

```
        vec_tiempos_h[myid] =(t_fin_calculos - t_ini_calculos);
        vec_tiempos_h[myid] =(double)(((double) vec_tiempos_h[myid])  / ((double) nfpi[myid])) ;
        vec_tiempos_h[myid] =vec_tiempos_h[myid] + (t_fin_over - t_ini_over);

vec_tiempos_ant[myid]=vec_tiempos_h[myid]=(0.5*vec_tiempos_ant[myid])+(0.5*vec_tiempos_h[myid]);
        if(vec_tiempos_h[myid]==0.0) vec_tiempos_ant[myid]=vec_tiempos_h[myid]=0.000001;
      }else{
        vec_tiempos_h[myid] = -1.0;
      }
    }
    nfpi[pos]=nfpi[pos]-1;
    if(nfpi[pos]==0){
     n=0;
     while(t_index[pos][n].proc !=-1){
       t_index[pos][n].proc=-1;
       n++;
     }
    }else{
     if(t_index[pos][pos_c].posr_pfpi==t_index[pos][pos_c].posr_ufpi){
      n=pos_c+1;
      while(t_index[pos][n].proc !=-1){
        t_index[pos][n-1].proc=t_index[pos][n].proc;
        t_index[pos][n-1].e=t_index[pos][n].e;
        t_index[pos][n-1].posr_pfpi=t_index[pos][n].posr_pfpi;
        t_index[pos][n-1].posr_ufpi=t_index[pos][n].posr_ufpi;
        n++;
      }
      n--;
      t_index[pos][n].proc=-1;
     }else{
      t_index[pos][pos_c].posr_pfpi=t_index[pos][pos_c].posr_pfpi+1;
     }
    }
    for(k=0;k<numprocs;k++){
      nfpi_ant[k]=nfpi[k];
    }/*
    if(myid==0){
      printf("k%d t_over %f t_cal %f\n",k_iteracion,t_fin_over - t_ini_over,t_fin_calculos - t_ini_calculos);
    }*//*
    if(myid==0){
     for(k=0;k<numprocs;k++){
       printf("k%d vec_tiempos %f vec_tiempos_h %f\n",k_iteracion,vec_tiempos[k],vec_tiempos_h[k]);
     }
    }*/
  }           /* Termina la k_iteracion  */
    MPI_Barrier(MPI_COMM_WORLD); /* se Inicia la Impresion de la Matriz Resultante */
    if(myid == 0){
      clock2=times(&buff2);
    }/*
    for(i=0;i<dim;i++){
      for(j=0;j<dim;j++){
        printf("%dA[%d][%d]=%f  ",myid,i,j,A[i][j]);
      }
      printf("\n");
    }
```

```
  printf("\n");
  for(i=0;i<dim;i++){
    for(j=0;j<dim;j++){
      printf("%dLU[%d][%d]=%f  ",myid,i,j,LU[i][j]);
    }
    printf("\n");
  } */
  if(myid==0){
   printf("\n WALL CLOCK TIME S = %f\n",(double) clock1);
   printf("\n WALL CLOCK TIME E = %f\n",(double) clock2);
   printf("\n WALL CLOCK TIME   = %f\n",(double)(clock2-clock1));
  }
  MPI_Finalize();
}
```

Centro de Información-Biblioteca
30002005897103