IMPLEMENTACIÓN DE COMPUTACIÓN DE ALTO RENDIMIENTO Y PROGRAMACIÓN PARALELA EN CÓDIGOS COMPUTACIONALES

CARLOS ANDRÉS ACOSTA BERLINGHIERI

UNIVERSIDAD EAFIT
ESCUELA DE INGENIERÍAS
ÁREA DE DISEÑO
MEDELLÍN
2009

IMPLEMENTACIÓN DE COMPUTACIÓN DE ALTO RENDIMIENTO Y PROGRAMACIÓN PARALELA EN CÓDIGOS COMPUTACIONALES

CARLOS ANDRÉS ACOSTA BERLINGHIERI

Proyecto de grado para optar al título de Ingeniero Mecánico

Asesor:
Dr. Jorge Luís Restrepo Ochoa

UNIVERSIDAD EAFIT
ESCUELA DE INGENIERÍAS
ÁREA DE DISEÑO
MEDELLÍN
2009

A mis padres y abuelos.

AGRADECIMIENTOS

Al Doctor Jorge Luís Restrepo Ochoa, asesor de este proyecto por su apoyo, por aportar su conocimiento y experiencia para la satisfactoria culminación del mismo.

A Manuel Julio García y a los compañeros del laboratorio ¨Mecánica Aplicada¨ por permitirme aprender tantas cosas junto a ellos.

Al Doctor Harry Millwater por sus aportes y continuo apoyo en este proyecto, Igualmente al ingeniero Juan David Ocampo y a todas aquellas personas que han contribuido con mi formación.

CONTENIDO

Pág.

LISTA DE FIGURAS

Pág.

LISTA DE TABLAS.

Pág.

LISTA DE ECUACIONES

Pág.

LISTA DE ANEXOS

Pág.

INTRODUCCIÓN

La solución de sistemas lineales en ingeniería mediante técnicas computacionales requiere largo tiempo de ejecución de programas, sobre todo cuando se implementan análisis de matrices con muchos datos y los sistemas lineales cuentan con muchas variables. Esto se debe a que en general los métodos clásicos utilizan un solo procesador para computar la solución de todo el sistema, y es necesario en el mundo real obtener soluciones detalladas casi en tiempo real.

Los métodos para solucionar sistemas lineales de ecuaciones se concentran en principio en los métodos iterativos y métodos directos de eliminación de incógnitas. Los métodos que se basan en técnicas iterativas o métodos sucesivos son los que se aproximan o convergen a una solución en cada iteración. Mientras que los métodos directos se enfocan en utilizar técnicas directas de sustitución o factorización de filas y columnas para logar una combinación de ecuaciones efectiva que permita la eliminación de variables en las ecuaciones.

Los métodos iterativos como la simulación de Monte Carlo consumen fracciones significativas de los ciclos de procesamiento en la evaluación probabilística de un código computacional. La precisión de los resultados es influenciada por la calidad de los números aleatorios que son generados y el tiempo de ejecución depende de la cantidad de ciclos que se ejecuten.

El campo de computación de alto rendimiento (HPC, High performance computing en inglés), es una rama de la ciencia de sistemas y computadores que estudia las técnicas y tecnologías que permiten que muchos computadores con sus aplicaciones trabajen en conjunto para resolver problemas comunes con un rendimiento eficiente para disminuir el tiempo de procesamiento de datos. Una de las técnicas que ofrece la computación de alto rendimiento es la programación paralela.

La programación paralela es un estilo de programación que consiste en agrupar operaciones en tareas para luego ejecutarlas en diferentes procesos que son concurrentes en el tiempo (Pacheco, 2007). Existen varias técnicas entre las que se destacan: La interface de paso de mensajes o MPI (Message passing interface en inglés) para modelos de programación en memoria distribuida a través de procesadores y OpenMP que es un modelo de programación paralela para memoria compartida para procesadores dentro del mismo sistema o computador.

Este documento está compuesto por un capítulo inicial en el que se presentan términos y conceptos relacionados con la programación paralela. El segundo capítulo contiene la forma de implementar computación paralela con memoria compartida y con memoria distribuida. También se presentan ejemplos con el fin de exponer la aceleración que es posible obtener con programación paralela. Al final de este capítulo se presenta un ejemplo clásico de programación paralela con memoria compartida que se encuentra en los tutoriales de Intel para memoria compartida dirigidos a supercomputadores o Clústers, en el que se resalta claramente la relación entre número de procesadores y la aceleración en el tiempo de ejecución

En el capítulo tres se muestran modelos de programación paralela con el fin de acelerar los resultados que entrega un código que soluciona un sistema de ecuaciones lineales mediante el método de eliminación Gaussiana y un código que soluciona la ecuación de Laplace para flujo potencial, ambos implementados en el lenguaje de programación C++. En el capítulo cuatro se presentan los resultados de la aceleración en el problema de la eliminación Gaussiana y la solución de la ecuación de Laplace para flujo potencial que se presentaron en el capítulo tres.

1 DEFINICIONES Y MARCO TEÓRICO

Este capítulo tiene como objetivo dar las definiciones y los elementos teóricos necesarios para comprender de una mejor manera las técnicas de programación paralela en donde muchas instrucciones se ejecutan en forma simultánea.

1.1 COMPUTACIÓN PARALELA Y SU APLICACIÓN EN INGENIERÍA.

La computación paralela es una técnica de programación en donde muchas instrucciones se ejecutan en forma simultánea. Se enfoca en solucionar grandes problemas mediante la agrupación y división de pequeñas tareas para después resolver de forma concurrente con el fin de obtener soluciones en menos tiempo.

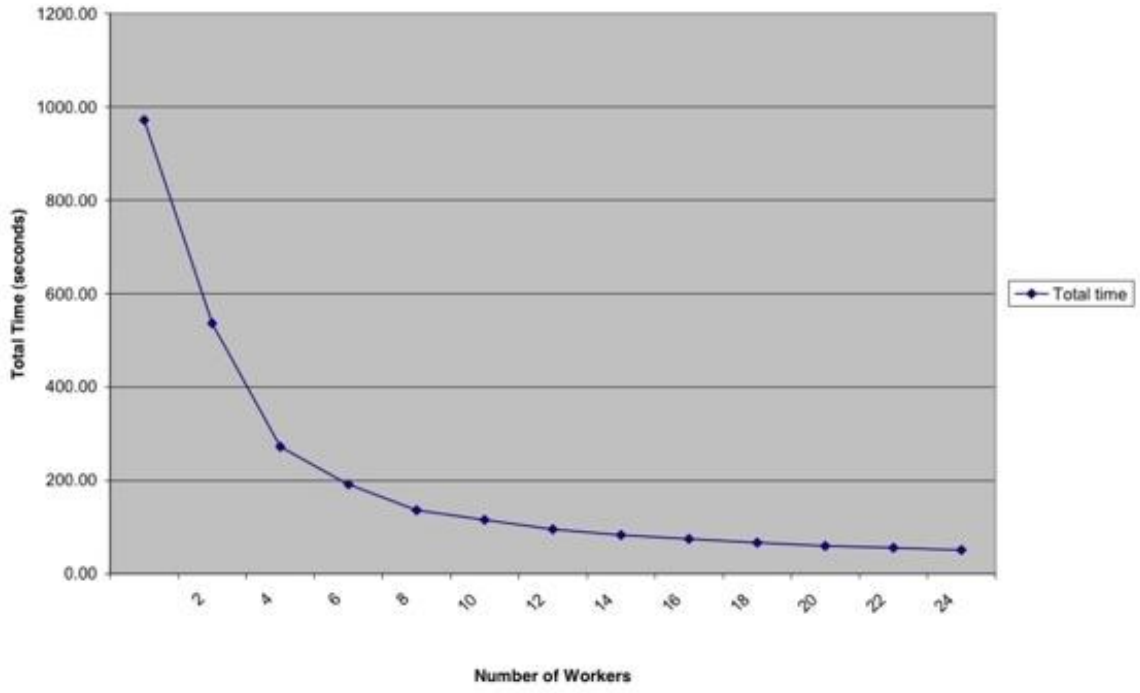
Las aplicaciones de la computación paralela en ingeniería son notables dado que es una herramienta que facilita realizar cálculos de manera rápida y eficiente mediante el uso de supercomputadores. Con esta herramienta se pueden calcular soluciones o aproximaciones para problemas de ingeniería.

Los problemas que se pueden paralelizar cuentan con la cualidad de ser ejecutados a través de varios procesadores con comunicaciones y sincronizaciones por medio de la red de comunicación a la cual están conectados. Las tareas que se pueden ejecutar de manera paralela por lo general tienden a reducir el tiempo total de ejecución del problema; de esta manera es posible obtener soluciones en menos tiempo.

Los problemas que con frecuencia utilizan supercomputadores y computación paralela son aquellos que cuentas con una amplia base de datos y bastantes operaciones iterativas; ese es el caso del estudio y predicción de tornados, la búsqueda de yacimientos petrolíferos con grandes bases de datos sísmicos, el diseño de aeronaves, el estudio de la deformación de materiales no convencionales, problemas estadísticos y financieros, etc.

La computación paralela se caracteriza porque los procesadores de una máquina pueden trabajar de forma coordinada al mismo tiempo, además cada procesador puede acceder a su memoria local y así compartir información con otros procesadores, con el fin de computar información de manera más efectiva.

Ilustración 1. Tiempo de ejecución a través de varios procesadores**.**



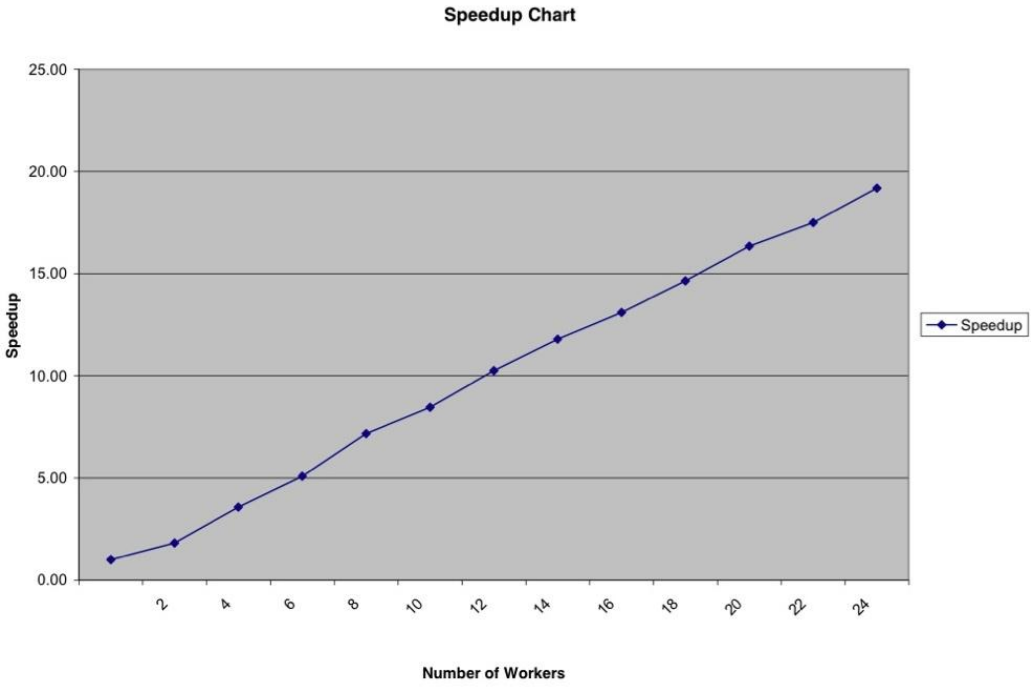
Revolution Computing, 2003

Un ejemplo de los beneficios de la programación paralela se muestra en las guías de usuario de computación de alto rendimiento que ofrece Hewlett-Packard a los usuarios de súper computadores enfocados a estudiar problemas probabilísticos en ingeniería. La Ilustración 1 compara el tiempo que consume ejecutar computaciones probabilísticas bajo el método de simulación iterativo de Monte Carlo al utilizar programación paralela. Se demuestra que al utilizar mayor cantidad de procesadores el tiempo de ejecución desciende.

La Ilustración 2 muestra la aceleración en la ejecución de la aplicación paralela. La aceleración en la ejecución en una aplicación paralela se entiende como un factor a dimensional que muestra la cantidad de veces que resulta ser más

rápido el tiempo de ejecución de un código a través de varios procesadores con respecto al tiempo de ejecución del mismo código con un solo procesador. Se percibe que entre más procesadores son utilizados para ejecutar el programa, el tiempo de computación es menor. Resulta necesario tener en cuenta la cantidad de procesadores que se pueden utilizar para ejecutar la aplicación final, dado que en los diseños preliminares de partición y comunicación de tareas dentro del programa se presentan problemas de escalabilidad o limitaciones con los grandes conjuntos de procesadores por la inadecuada división del problema en tareas.

Ilustración 2. Aceleración de un código probabilístico en paralelo**.**



Revolution Computing, 2003

1.2 CONCEPTOS BÁSICOS DE LA PROGRAMACIÓN PARALELA.

La mayoría de los problemas típicos de ingeniería que utilizan códigos de programación utilizan bloques iterativos para la solución o aproximación de ecuaciones. Es estos programas se gasta la mayor parte del tiempo en ciclos. Los ciclos a nivel de programación paralela tienden a reducir el tiempo de procesamiento de bloques iterativos mediante la ejecución de tareas concurrentes en diferentes procesadores. Los siguientes conceptos y términos son usados con frecuencia para aplicar programación paralela (Quinn, 2004).

1.2.1 Sistemas paralelos de computación.

Los sistemas paralelos de computación son sistemas con varios procesadores interconectados a través de una red de comunicación que soportan aplicaciones compartidas. Las unidades de procesamiento en diferentes computadores interactúan entre sí mediante actividades como enviar y recibir mensajes a través de la red de comunicación.

Los computadores que contienen más de un núcleo de procesamiento para efectuar sus operaciones son conocidos como sistemas paralelos de computación. Sistemas con gran cantidad de procesadores conectados para ejecutar operaciones de forma concurrente son conocidos como súper computadores.

El tipo de conexión que utilizan las redes de computación paralela para comunicar los procesadores y su memoria puede variar. La clasificación de las máquinas paralelas se basa en el tipo de cadenas de información ejecutadas, así que la misma instrucción para diferentes unidades de procesamiento se conoce como SIMD (Single Instruction Multiple Data en inglés), que es opuesto a múltiples cadenas de información ejecutadas por múltiples procesadores. Esta última se conoce como MIMD (Multiple Instructions Multiple Data en inglés).

1.2.2 Supercomputador o Clúster.

Un supercomputador o Clúster es el tipo de computador más potente y más rápido que existe en este momento. Estas máquinas están diseñadas para procesar enormes cantidades de información en poco tiempo.

1.2.3 Programación Paralela.

Es el uso de sistemas paralelos de computación para reducir el tiempo de ejecución de programas computacionales. La programación paralela es considerada una herramienta importante para resolver modelos climáticos, diseño de aeronaves, modelos financieros, etc.

La programación paralela se expresa con un lenguaje de programación que permite identificar las porciones de código que deben ser ejecutadas de forma concurrente a través de diferentes procesadores.

1.2.4 Hilo de ejecución.

Es una entidad que se encarga de ejecutar operaciones de forma serial con un espacio para almacenar información asociado a cada entidad. Un hilo de ejecución es la función operativa del procesador

1.2.5 Múltiples hilos de ejecución.

Es la ejecución de más de un Hilo de ejecución en un programa paralelo. Es el conjunto de procesadores que trabajan de forma concurrente en el tiempo para ejecutar un trabajo en paralelo.

1.2.6 Lenguaje base.

Lenguaje de programación que utiliza las aplicaciones de OpenMP con operaciones en paralelo para memoria compartida y MPI con operaciones en paralelo para memoria distribuida.

1.2.7 Bloque paralelo.

Es un bloque de las declaraciones ejecutables que realizan cómputos paralelos; este bloque tiene solamente un punto de entrada y una punta de salida.

1.2.8 Directivas de OpenMP.

Son las declaraciones que permiten realizan el trabajo paralelo a través del compilador (#pragma para C/C y !$omp para FORTRAN).

1.2.9 Hilo de ejecución inicial.

Hilo de ejecución que ejecuta las partes secuenciales del código. Es la parte del código donde no ocurre trabajo en paralelo. Esta parte se refiere con frecuencia a la designación de variables y librerías.

1.2.10 Hilo de ejecución principal.

Hilo de ejecución que encuentra el comienzo de una región paralela. Se entiende como el procesador principal desde cual se bifurca la información para ser procesada en paralelo.

1.2.11 Equipo de hilos de ejecución.

Conjunto de hilos de ejecución que realizan el trabajo paralelo. Es el grupo de procesadores que trabajan en paralelo dentro de una aplicación para resolver un problema.

1.2.12 Barrera.

Es el punto en el cual el equipo de hilos de ejecución comienza a realizar el trabajo en paralelo.

1.2.13 Variable.

Dato especificado cuyo valor se puede definir durante la ejecución de un programa.

1.2.14 Variable privada.

Variable cuyo nombre proporciona el acceso a una memoria diferente a la memoria donde se almacenan datos para un equipo de hilos de ejecución. Las variables que se definen privadas tienen varias copias almacenas en memoria, cada hilo de ejecución tiene su propia copia en su memoria y sólo puede actualizar el valor de la variable que le corresponde.

El modelo de programación entre un código serial y uno paralelo es diferente, debido a que cada hilo de ejecución necesita almacenar en memoria una copia de las variables privadas. Los programas que van a ser ejecutados paralelamente deben contener y desarrollar las técnicas de programación para memoria compartida para no cometer infracciones en lectura o escritura de posiciones de vectores o procesamiento de operaciones. La forma de especificar de forma correcta la ejecución en paralelo de un programa con control apropiado de las variables es la siguiente:

```
!$omp parallel do shared(n) private(indices)
      Do índice = 1, n
            Trabajo que se realiza dentro del ciclo
      END DO
¡$omp end parallel do
```

El trabajo que se realiza dentro del bloque iterativo contiene una sola ubicación en memoria para la variable ¨n¨, mientras que contiene varias copias de la

variable ¨índice¨, tanta copias como se especifique en las variables de contorno para ejecutar a través de varios procesadores.

1.2.15 Variable compartida.

Variable cuyo nombre proporciona al acceso a la misma memoria del almacenamiento para cada hilo de ejecución en un equipo. Las variables compartidas tienen la cualidad de sólo ser almacenadas en una ubicación de memoria durante la ejecución del programa. Cuando un grupo de hilos en una región paralela efectúa una operación, las variables que declaran compartidas son actualizadas por cualquier hilo de ejecución mediante lectura y sobre escritura de datos.

```
!$omp parallel do shared(n) private(índice)
        Do índice = 1, n
                Trabajo que se realiza dentro del ciclo
        END DO
¡$omp end parallel do
```

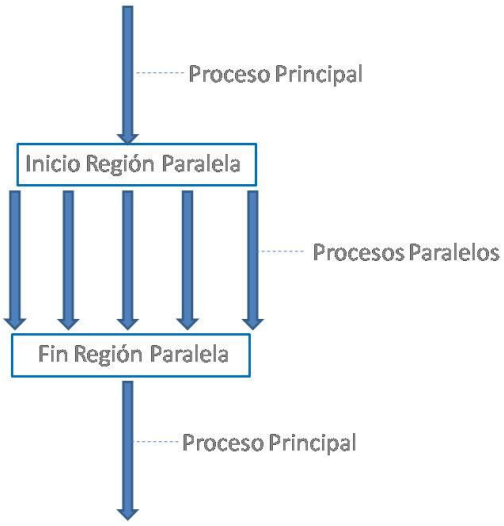
1.2.16 Regiones paralelas.

El modelo de programación de OpenMP se implementa mediante aplicaciones programadas, estas aplicaciones interactúan con el sistema operativo mediante un amplio conjunto de estructuras de datos, clases y protocolos. La Ilustración 3 muestra el modelo de programación paralela mediante hilos de ejecución dentro de una región paralela.

Las librerías de OpenMP soportan múltiples arquitecturas de programas escritos en Fortran 77/90 y C/C++, además se pueden ejecutar en ambientes como Linux, Unix y Windows NT.

El modelo de operación y de programación paralela mediante el uso de aplicaciones de memoria compartida como OpenMP se muestra en la Ilustración 3. En este modelo de programación paralela con memoria compartida las variables dentro del programa sólo se pueden declarar como variables compartidas o como variables privadas para cada proceso de ejecución, y se deben declaran antes de cualquier bloque iterativo (en C/C++ bloque *for*, en Fortran 77/90 bloque iterativo *do*).

Ilustración 3. Programación paralela mediante hilos de ejecución.



TACC, 2009

1.2.17 Funciones y librerías de MPI.

El diseño de una aplicación paralela con MPI se realiza gracias el uso de diferentes funciones o subrutinas que se encuentran agrupadas en librerías estandarizadas como las librarías OpenMPI o MPICH. Más allá de la estructura de las librarías y las especificaciones de la máquina o computador, el uso de estas funciones crea canales de comunicación entre procesadores. Algunas de las funciones más utilizadas son:

- **MPI_Init().** Se encarga de inicializar las variables y actividades que se realizan durante la ejecución del programa en los procesadores.

- **MPI_Barrier().** Se encarga de detener las operaciones de los procesadores en un punto específico del programa. **Esta función** permite que cada procesador bloquee la actividad de enviar o recibir mensajes hasta que cada proceso global termine su trabajo.

- **MPI_scatter().** Se encarga de distribuir la información entre todos los procesadores.

- **MPI_gather().** Se encarga de reagrupar los datos de la aplicación paralela en diferentes procesadores.

Las aplicaciones de MPI siempre terminan con la función **MPI_Finalize()**. Esta es una función que especifica la conclusión de la estructura paralela del código después de que se terminen los cómputos paralelos.

1.3 RESTRICCIONES PARA ACELERAR PROCESOS PARALELOS

La ley de Amdahl plantea las restricciones en aplicaciones paralelas que impiden una aceleración lineal en la ejecución de la aplicación. La aceleración lineal o aceleración óptima en computación de alto rendimiento se refiere a la aceleración que un programa puede alcanzar a medida que es ejecutado en diferentes escalas de procesamiento; es decir, a medida que es ejecutado con mayor cantidad de procesadores. La ley de Amdahl (ver Ecuación 1) expone la aceleración de un programa paralelo en función de la porción del programa que ejecuta computaciones en serial y la parte que es ejecutada en paralelo.

Ecuación 1. Ley de Amdahl de aceleración en aplicaciones paralelas.

$$S = \frac{1}{f_s + f_p / N}$$

TACC, 2009

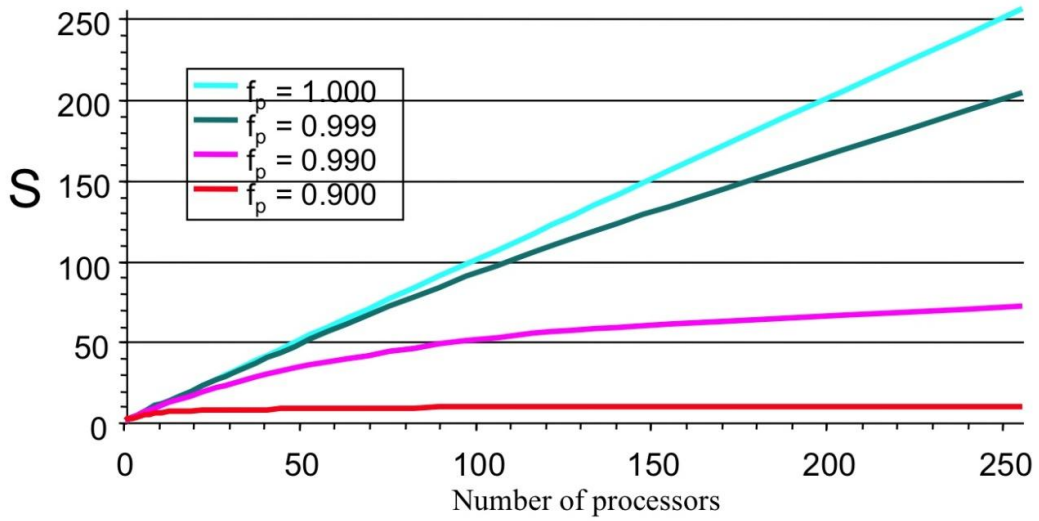
Donde:

$f_s =$ Porción del código que ejecuta computaciones en serial.
$f_p =$ Porción del código que ejecuta computaciones en paralelo.
$N =$ Número de procesadores.
$S =$ Aceleración en la ejecución del programa paralelo

La ley de Amdahl no es la única restricción para acelerar el tiempo de ejecución de códigos computacionales, dado que existen otros factores que pueden entorpecer la aceleración de una aplicación paralela. Esos factores se refieren a la manera poco eficiente como se programa una aplicación paralela. Se entiende que una aplicación paralela es poco eficiente cuando existen más comunicaciones y sincronizaciones de las necesarias dentro del código.

La Ilustración 4 expone las restricciones de aceleración en computación paralela en cuanto a la ley de Amdahl se refiere. Se muestra que la mayor aceleración se presenta cuando la mayor parte del programa está paralelizado.

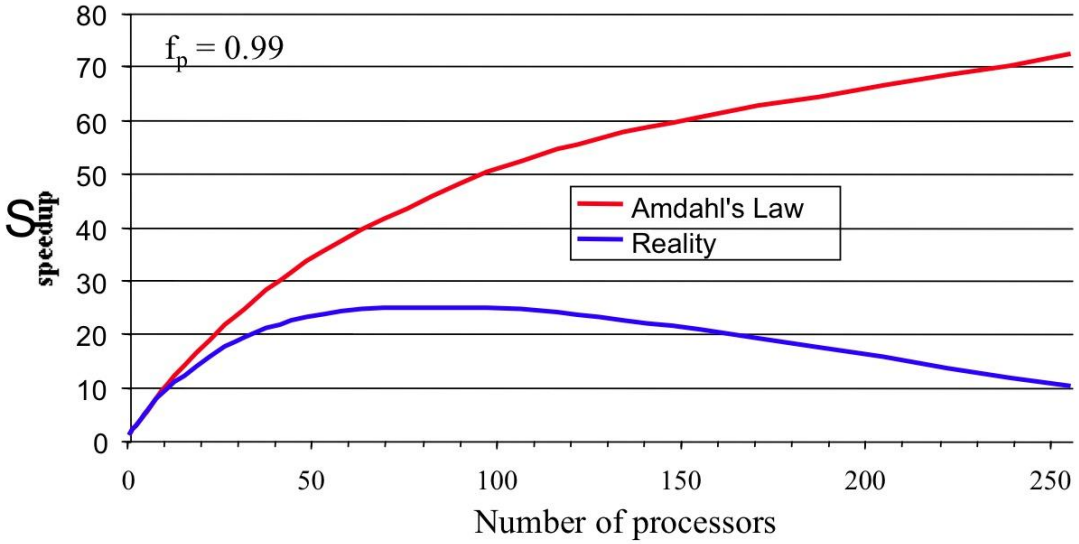
Ilustración 4. Ejecución de aplicaciones según la ley de Amdahl.



TACC, 2009

La Ilustración 5 muestra la comparación entre la aceleración de un código restringido por la ley de Amdahl y el rendimiento al utilizar varios procesadores.

Ilustración 5. Modelo teórico y real de la ley de Amdahl.



TACC, 2009

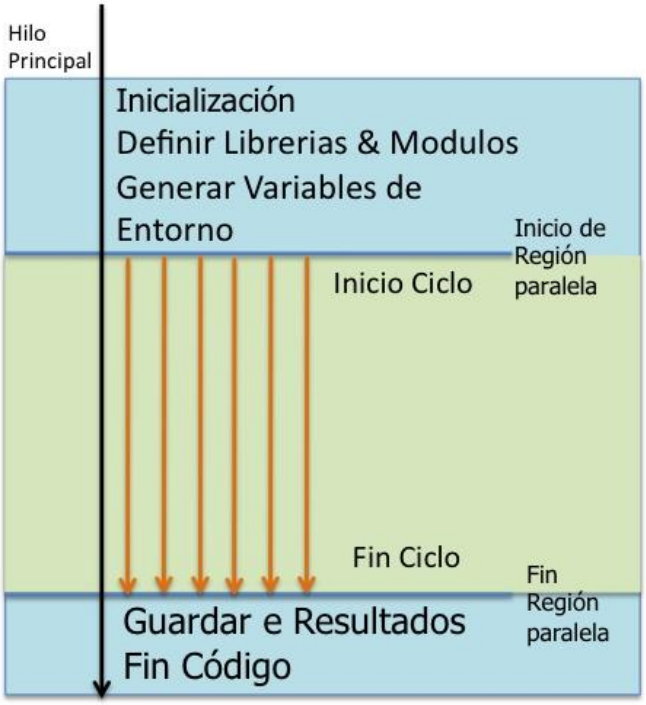
Las aplicaciones paralelas que utilizan largas cadenas de procesadores incurren en problemas de comunicación y sincronización por enviar flujos de mensajes de gran tamaño a través de la red de comunicación.

1.4 COMPUTACIÓN PARALELA Y USO DE RECURSOS.

Las alternativas más comunes que existen para crear programas paralelos se centran en construir aplicaciones con memoria distribuida (el caso de MPI) o con memoria compartida (el caso de OpenMP).

Los tipos de computación paralela más conocidos se refieren al paralelismo a nivel de instrucción, paralelismo de datos y paralelismo de tareas. Estos tipos de paralelismo buscan reordenar la secuencia de ejecución del programa al dividir tareas en procesos, es decir, cambian el orden de lectura y escritura de datos durante la ejecución del programa a través de diferentes hilos de ejecución en una región paralela. En la Ilustración 6 se muestra el esquema de una región paralela.

Ilustración 6. Región paralela con seis hilos de ejecución.



Chandra, 2001

La programación paralela a nivel de instrucciones se basa en controlar la cantidad de operaciones que se realizan en una región dentro de un código

(Quinn, 2004), con el fin de asegurar que durante la ejecución del programa no exista ningún tipo de corrupción de datos.

El paralelismo a nivel de datos es una forma de paralelismo que se centra en el agrupamiento y división de información para ejecutar a través de múltiples procesos. Esto se realiza al distribuir la información en los nodos de un súper computador o Clúster.

La computación paralela a nivel de tareas es una forma de paralelización de rutinas a través de diversos procesadores al enfocar la ejecución en secuencias de hilos o cadenas de información que se ejecutan concurrentes en el tiempo. OpenMP se apoya en la computación paralela a nivel de tareas y de instrucciones.

La computación paralela permite acelerar la ejecución de códigos computacionales. Para implementar directivas en programas paralelos se debe tener en cuenta que la memoria necesaria para utilizar aplicaciones compartidas es hasta cinco veces mayor que la memoria requerida para ejecutar un programa en serial (Quinn, 2004). Por eso es necesario correr las aplicaciones en máquinas que cuentan con amplia disposición de memoria, como es el caso de un Clúster.

Los computadores diseñados para facilitar un ambiente adecuado para la ejecución de tareas en paralelo proveen formas de coordinación en procesadores para obtener y facilitar información en orden de sincronizar actividades o realizar operaciones en forma concurrente. Sin embargo, tanto la comunicación entre procesos, como la sincronización entre actividades se llevan a cabo por medio de funciones y librerías. El uso excesivo de estas funciones altera el adecuado funcionamiento y tiempo de ejecución del programa. El uso eficiente de funciones y la correcta programación evitan el uso de sincronizaciones y comunicaciones innecesarias.

Las arquitecturas que soportan programación paralela, tanto para memoria compartida como para memoria distribuida se aferran a la estructura de paso

de mensajes. Este tipo de estructuras tienen múltiples secuencias de instrucciones para múltiples secuencias de ejecuciones (MIMD). En este caso todos los procesadores disponibles en un sistema efectúan diferentes operaciones al mismo tiempo. Mientras que una sola secuencia de instrucciones o parámetros se divide en múltiple secuencias de ejecuciones (SIMD).

Los modelos de programación paralela con memoria compartida o con memoria distribuida son enfocados a las aplicaciones escritas en los lenguajes de computación Fortran o C/C++, y cuentan con librerías especializadas que se encargan de realizar operaciones de comunicación y sincronización dentro de las aplicaciones.

La realidad muestra que existen casos avanzados de programación paralela que combinan los dos métodos de programación paralela relacionados con memoria compartida y distribuida. Está práctica tiene el fin de combinar los beneficios de la memoria distribuida con los beneficios de la memoria compartida. Este caso se refiere al modelo hibrido de programación en paralelo.

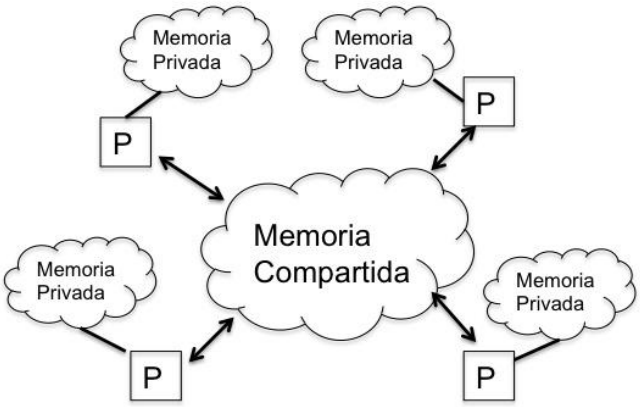
La programación paralela con memoria compartida le permite a todos los procesadores de una estructura computacional acceder al mismo espacio de memoria disponible en una máquina. La Ilustración 7 muestra el modelo de memoria compartida que utiliza OpenMP.

La programación de aplicaciones con memoria compartida permite que todos los procesadores de una estructura computacional accedan al mismo espacio de memoria de forma sincronizada y concurrente.

Para realizar programación paralela con MPI se utiliza el modelo de memoria distribuida. Para programar con este modelo se deben comprender las técnicas, funciones estándar y las herramientas propias de MPI, con el fin de realizar una apropiada división del dominio para descomponer un problema o un código en pequeñas tareas para ejecutarlas en paralelo con varios

procesadores. El uso inapropiado de las herramientas conduce a una solución equivocada al problema por corrupción de variables en los procesos de división, distribución o reagrupamiento de variables.

Ilustración 7. Modelo de computación con memoria compartida.



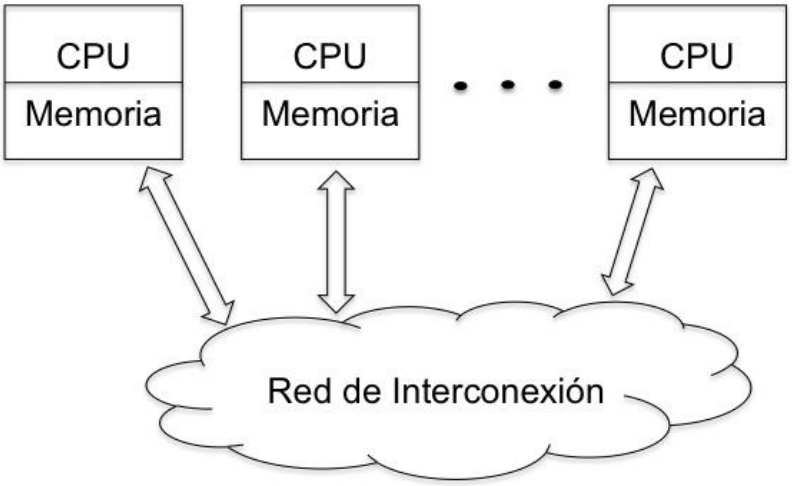
Chandra, 2001

La ejecución de programas paralelo con la interface de paso de mensajes (MPI) alcanzan aceleraciones aceptables cuando la dotación física de la máquina y las especificaciones operativas del sistema cuentas con opciones de optimización de procesos y aplicaciones de alto rendimiento. El concepto de paso de mensajes es el mismo para la mayoría de las librerías de MPI sin importar el diseño del código o de la aplicación paralela y las características del Clúster que se utiliza para ejecutar el programa. El paso de mensajes entre procesos es la actividad de comunicación que se lleva a cabo entre los procesos con el fin de sincronizar instrucciones y compartir información. Esta sincronización se realiza por medio de la red de interconexión del sistema (ver Ilustración 8).

Las librerías de MPI están almacenadas en una biblioteca de funciones y macros que pueden ser utilizadas en la escritura de programas paralelos para la implementación de computación de alto rendimiento. Las operaciones de MPI dentro del programa compilar archivos ejecutables a través de todos los procesos. Diversos procesos pueden ejecutar diversas instrucciones gracias a la división del dominio dentro del programa. Es común encontrar que la división

del dominio se basa en la identificación tareas o y la reorganización de datos dentro de cada proceso.

Ilustración 8. Modelo de computación con memoria distribuida.



TACC, 2009

1.5 OPENMP PARA PROGRAMACIÓN PARALELA

La programación paralela con memoria compartida se implementa mediante directivas de ejecución, las librerías del lenguaje y las variables de entorno. La Ilustración 9 muestra el ciclo de ejecución de OpenMP.

Ilustración 9. Arquitectura de OpenMP.

TACC, 2009

1.5.1 DIRECTIVAS DE OPENMP

Las directivas de OpenMP contienen funciones que facilitan el almacenamiento de datos en memoria para la ejecución de tareas paralelas, sincronización implícita y explícita de datos durante la ejecución del programa, secciones para dividir trabajo entre procesos y mecanismos para balancear trabajo entre hilos,

entre otras opciones.

Las directivas de OpenMP se encargan de encadenar la arquitectura del compilador con la estructura del programa paralelo. Sin estas directivas el compilador no tiene forma de crear hilos para diferentes secuencias de ejecución entre diferentes procesadores.

1.5.2 FUNCIONES Y LIBRERIAS

Las funciones y librerías controlan la apropiada ejecución de las aplicaciones paralelas de memoria compartida con OpenMP mediante sincronización de variables con comunicación de procesos a lo largo de la ejecución del programa. En general se sostiene que un buen código que se ejecuta en paralelo debe contener la mínima cantidad de comunicaciones posible, dado que las sincronizaciones desaceleran la ejecución del programa.

Las funciones para la ejecución de aplicación paralela se encuentran en ¨omp_lib" para Fortran y en "omp.h" para C/C++.

El ambiente de OpenMP permite la ejecución de instrucciones en paralelo mediante el uso de sus funciones. Esas funciones son:

- **OMP_SET_NUM_THREADS**: Establece el número de hilos de ejecución.
- **OMP_GET_NUM_THREADS**: Regresa el máximo número de hilos de ejecución que se usan de forma concurrente en la aplicación.
- **OMP_GET_MAX_THREADS**: Regresa el número total de hilos de ejecución paralelos creados en la aplicación.
- **OMP_GET_THREAD_NUM**: Regresa el número de hilos de ejecución que trabajan de manera concurrente en la aplicación.
- **OMP_GET_NUM_PROCS**: Regresa el número de procesadores disponibles.
- **OMP_PARALLEL**: Activa o desactiva el modo de concurrencia en la máquina.
- **OMP_INIT_LOCK**: Se encarga de blindar el valor de cierta variable.
- **OMP_DESTROY_LOCK**: Desactiva el blindaje de las variables.

- **OMP_SET_LOCK**: Se encarga de activar o desactiva el blindaje de las variables.
- **OMP_TEST_LOCK**: Se encarga de actualizar el valor de las variables blindadas.
- **OMP_GET_WTIME**: Regresa el tiempo de ejecución de la aplicación.
- **OMP_SET_DYNAMIC**: Se encarga de activar o desactivar el paralelismo dinámico entre hilos de ejecución; es decir, reasigna instrucciones entre hilos durante la ejecución de la aplicación paralela.
- **OMP_GET_DYNAMIC**: Verifica el estado dinámico o estático de la aplicación paralela.
- **OMP_SET_NESTED**: Se encarga de habilitar o deshabilitar el paralelismo en red; es decir, el paralelismo a nivel de distribución de instrucciones.
- **OMP_GET_NESTED**: Verifica el estado estándar o en red de la aplicación paralela.

1.5.3 VARIABLES DE ENTRONO

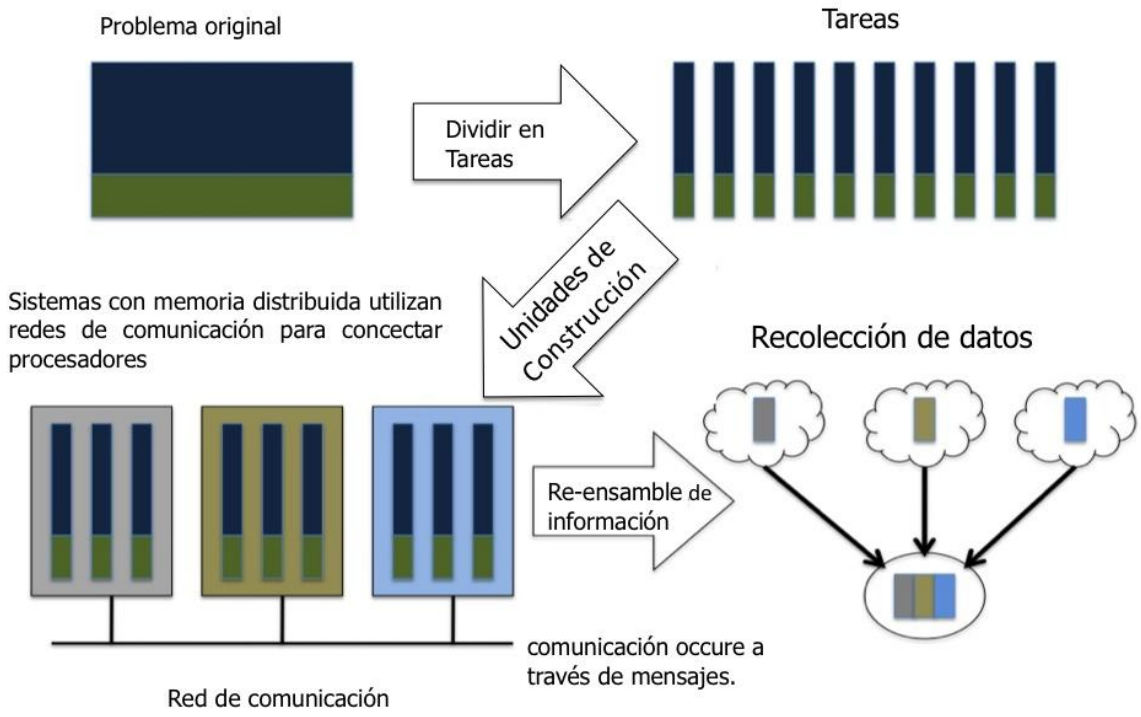
Las variables de entorno en OpenMP se encargan de especificar al compilador la clase de ejecución paralela que el usuario establece. Las variables de entorno son funciones que se encuentran en las aplicaciones y librerías programadas de OpenMP. Con las variables de entorno se puede especificar el número de procesadores o hilos de ejecución con que deben correr el programa en paralelo, también se puede conocer el tiempo que tarda el programa en ejecutarse. Las variables de entorno especifican de manera explícita las siguientes actividades.

- Números de hilos de ejecución concurrentes en el tiempo
- Actividades de reconocimiento de hilos.
- Se encargan de ejecutar modelos de programación avanzados en memoria compartido como programación paralela en red.
- Se encarga de ejecutar aplicaciones de maneras convencionales para asegurar que la no existencia de corrupción en la comunicación y sincronización de las aplicaciones.
- Se encarga de ejecutar aplicaciones paralelas de forma dinámica o con reasignación de trabajo durante la ejecución de los hilos.
- Para determinar el tiempo total de ejecución.

1.6 MPI PARA PROGRAMACIÓN PARALELA

La arquitectura de paso de mensajes en memoria distribuida permite el acceso a instrucciones y datos que se salvan en la memoria de cada procesador mediante el uso de la red de interconexión implícito en las actividades de comunicación.
La Ilustración 10 muestra la forma clásica de construir una aplicación con programación paralela bajo el método de memoria distribuida. Con este método se deben agrupar tareas independientes para ejecutar de forma concurrente a través de diversos procesadores conectados entre sí. La sincronización y comunicación entre los procesadores es importante para recolectar los datos una vez concluyen las computaciones paralelas.

Ilustración 10. División de tareas a través de procesos.



TACC, 2009

La programación paralela con MPI utiliza las técnicas específicas de programación mediante el uso de las funciones estándar para descomponer un problema o un código en pequeñas tareas para ejecutarlas de manera concurrente.

El método que utiliza la interface de paso de mensajes para ejecutar instrucciones en diferentes procesadores consiste en construir tareas o procesos con el fin de intercomunicar las actualizaciones o cambios de variables mediante el uso de distribución de mensajes a través de los procesadores. De esta forma se construye un canal para cada par de procesos a través de la red de interconexión.

El programador debe especificar el número de procesos simultáneos cuando la ejecución comienza. El número de procesos activos sigue constante durante la ejecución del programa cuando la cantidad de trabajo entre procesos se define de forma apropiada.

2 IMPLEMENTACIÓN DE ALGORITMOS PARALELOS

El diseño de un algoritmo paralelo no se expresa en reglas de reducción, no existe una serie de pasos que se puedan aplicar siempre. Se requiere creatividad, conocimiento y destreza para manejar las herramientas que facilitan la creación de algoritmos paralelos (Lastovetsky, 2003).

2.1 IMPLEMENTACIÓN DE COMPUTACIÓN PARALELA

Existen varias técnicas y herramientas con el fin de paralelizar códigos computacionales; entre ellas, es común encontrar dos técnicas. Se refieren a la forma de paralelizar instrucciones por medio de procesos (este es el método de programación paralela que utiliza la interface de paso de mensaje, MPI, con memoria distribuida) y por medio de hilos de ejecución (este es el método de programación paralela que utiliza la interface de mensaje de paso abierto, OpenMP, con memoria compartida). Algunos conceptos necesarios para implementar computación paralela son los siguientes.

2.1.1 PARTICIÓN DEL PROBLEMA

Es la etapa de división en tareas de un algoritmo. Se realiza con la intención de encontrar el diseño óptimo para la ejecución paralela más eficiente de un código computacional; por lo tanto, el objetivo está en la definición de una gran cantidad de pequeñas tareas para la adecuada descomposición de un problema.

Si las tareas comparten información durante la ejecución, el proceso se denomina descomposición del dominio, pero si las tareas sólo comparten información al finalizar la ejecución, el proceso se denomina descomposición funcional.

La fase de división del código debe producir una o más posibles descomposiciones de un problema. La división de tareas debe definir por lo menos un orden de magnitud mayor que el número de procesadores

disponibles para la aplicación. Si existen más procesadores que tareas a ejecutar, la flexibilidad del algoritmo en etapas de ejecución es limitada.

Las tareas deben ser similares en cuanto a su tamaño, puesto que de esta manera se asigna igual cantidad de trabajo a cada procesador disponible. El número de tareas creadas para ejecutar la aplicación de forma concurrente debe tener una apropiada relación con respecto a la cantidad de iteraciones y operaciones dentro del programa, dado que el aumento de tamaño del problema significa el aumento en el número de tareas asignadas para resolver.

2.1.1.1 Descomposición del dominio

La intención de crear varias tareas que ejecuten porciones del problema en paralelo se enfoca en reducir el tiempo de ejecución de problemas con muchas operaciones. Es común encontrar situaciones donde las operaciones que se realizan en una tarea requieren datos asociados a otra tarea. Este concepto de paralelización se denomina descomposición del dominio. Si es posible, se divide la información en pequeños pedazos de instrucciones con tamaño aproximadamente igual. Después, se divide el problema que se debe solucionar, mediante la asociación de cada operación con el conjunto de datos que se analiza. De la división resulta un número de tareas determinado, cada división comprende ciertos datos más un conjunto de operaciones que utilizan la información para realizar cálculos.

Una operación puede requerir datos almacenados en varias tareas. En este caso, la comunicación y sincronización es necesaria para mover información entre las tareas. Los datos divididos en tareas pueden ser en muchos casos la información de entrada del programa, la salida calculada por el programa, o valores intermedios utilizados por el programa. Existen varias maneras de afrontar la descomposición de datos, unas pueden basarse en reorganizar y dividir estructuras de datos, otras se concentran primero en la estructura de datos principal o en la estructura de datos que se accede con más frecuencia.

Es común encontrar diversas fases de descomposición de tareas para diferentes estructuras de datos, o existen diseños de algoritmos que exigen diversas descomposiciones con el fin de efectuar las operaciones computacionales con mayor rapidez.

2.1.1.2 Descomposición funcional

La descomposición funcional es una manera de dividir el problema en trozos o tareas paralelas. Se enfoca en la ejecución de operaciones a través de tareas que son concurrentes en el tiempo con base en la información disponible para resolver el problema.

La descomposición funcional tiene un papel importante si se entiende como una técnica de estructuración de algoritmos paralelos. Una descomposición funcional con una adecuada implementación busca dividir no sólo las operaciones que deben ser realizadas de forma concurrente, también busca reducir la complejidad del diseño total del algoritmo.

2.1.2 COMUNICACIÓN ENTRE TAREAS

En el proceso de descomposición del dominio la información debe ser transferida a través de las tareas implicadas en el proceso. Este flujo y transferencia de información se conoce como fase de comunicación. El programador tiene la responsabilidad de implementar procesos de comunicación a través de tareas en la fase de diseño del algoritmo paralelo.

La comunicación se conceptualiza como el canal que se crea entre dos o más tareas con el fin de facilitar las acciones de sincronización. La creación de un canal implica un costo operacional en el algoritmo paralelo, por tanto, La inadecuada creación de canales puede afectar la rapidez de ejecución del programa. Es necesario optimizar el funcionamiento del programa al distribuir operaciones e información de forma coherente a través de las tareas, para ordenar las comunicaciones de una manera eficiente.

2.1.2.1 Comunicación local entre tareas.

Las estructuras en aplicaciones paralelas se diferencias por su enfoque local o global según las variables del programa. La estructura local de comunicación se presenta cuando una operación o proceso requiere datos de una pequeña cantidad de tareas. Así es como se logran definir los canales que conectan las tareas responsables de realizar las operaciones con aquellas tareas que ejecutan los datos.

2.1.2.2 Comunicación global entre procesos.

La comunicación global es una operación en la cual muchas o todas las tareas deben participar cuando se ejecutan operaciones que afectan el curso del resto del problema. Puede ser insuficiente identificar tareas individuales dentro del problema puesto que da lugar a muchas comunicaciones o puede restringir las oportunidades para la ejecución en paralelo de las aplicaciones.

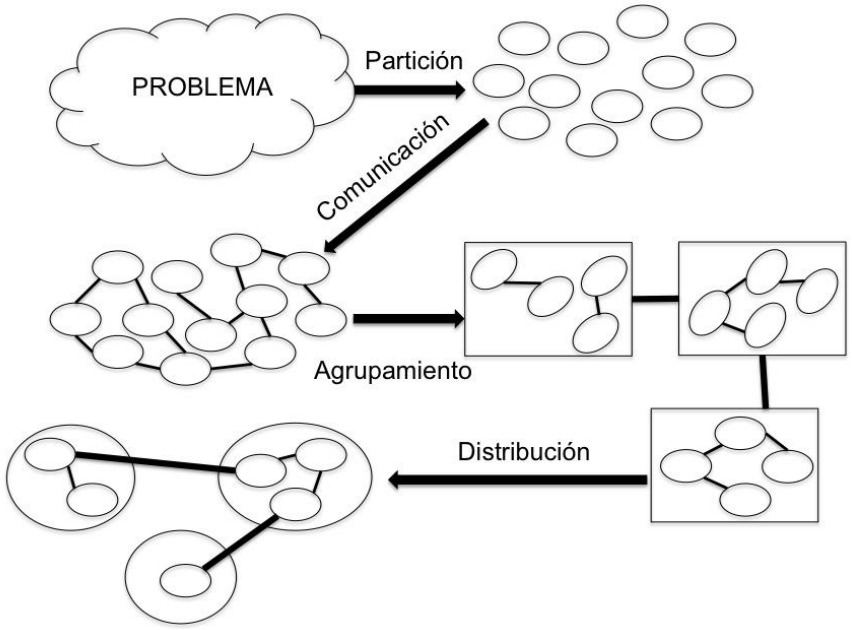
2.1.3 AGRUPACIÓN DE DATOS

Independiente de la forma como se descompone el problema, la agrupación de datos consiste en agrupar los conjuntos de soluciones que se obtienen a partir de las fases de división y comunicación para obtener una aplicación que ejecute de forma eficiente un algoritmo paralelo. En la fase de agrupación de datos es necesario considerar la utilidad de combinar o reunir tareas identificadas por la fase de división. De este modo se puede proporcionar un número de tareas más pequeño, o reunir en una sola tarea el contenido total de la solución.

La Ilustración 11 muestra una estrategia de Descomposición y reordenamiento del problema con el fin de establecer procesos y asignar identidades con el fin de computar la información almacenada en tareas. La estrategia consiste en establecer las fronteras del problema para dividirlo en muchas porciones pequeñas de actividades o sub problemas. Entre estas porciones del problema se establecen lazos de comunicación entre las actividades que requieran

sincronización. La sincronización hace posible ordenar de nuevo la información en grupos de instrucciones para obtener soluciones a cada sub problema.

Ilustración 11. Descomposición y reordenamiento del problema.



TACC, 2009

2.2 COMPILACIÓN DE APLICACIONES

El adecuado uso de opciones de compilación es fundamental para obtener eficiente aceleración de procesos paralelos. El impacto que produce utilizar un compilador en específico sobre el código depende de varios factores como el conocimiento del programador acerca del compilador y las opciones de implementación intrínsecas en cada compilador (ver Tabla 1).

Tabla 1. Opciones de compilación

Sistema Operativo Linux – Unix	**Opciones de compilación (Intel/PGI/GNU)**
`-O0`	No realiza optimización al código.
`-O1`	Optimiza el código en aspectos dirigidos a la capacidad de almacenar datos en memoria sin añadir tamaño adicional al código.
`-O2`	Maximiza la velocidad de ejecución del programa y utiliza operaciones de “vectorización” o

	"encapsulación" para acceder con mayor facilidad a bloques de instrucciones.
-O3 -fast(PGI)	Permite la optimización de instrucciones con -O2 con acceso de memoria agresivo y en algunos casos existen cambios en la estructura matemática de las operaciones dentro del código.
-g	Habilita opciones de depuración del programa.
-openmp (Intel) -mp(PGI) -fopenmp(GNU)	Habilita opciones para ejecutar instrucciones a través de diferentes procesadores. Permite el uso específico de librerías propias de OpenMP.
mpif90 (Intel/GNU) mpif95(PGI)	Crea canales de comunicación entre procesos con el fin computar instrucciones de forma concurrente a través de diferentes procesadores. Habilita el uso de funciones y librerías propias de MPI.
mpif90 -openmp(Intel) mpif90 -fopenmp(GNU) mpif95 -mp(PGI)	Compila las librerías de MPI y las librerias de OpenMP en una misma aplicación.

La optimización del compilador permite obtener alta velocidad y computación de alto rendimiento con las soluciones exactas sin esfuerzo adicional de codificación. El máximo provecho de los beneficios de los compiladores se obtiene en aplicaciones paralelas, dado que combinan las ventajas del la arquitectura y optimización de un compilador más la eficiencia de varios procesadores conectados para solucionar un mismo problema.

La eficiencia de compilación también depende del tipo de ambiente o sistema operativo (Windows, Unix, Linux, etc) de la máquina que se utiliza para ejecutar la aplicación, entre otros factores.

2.3 ESPECIFICACIONES TÉCNICAS DEL CLÚSTER

La tecnología más rápida y avanzada en sistemas computacionales es la tecnología de un Clúster o supercomputador puesto que cuentan con redes de interconexiones complejas y procesadores muy potentes. Los

supercomputadores se caracterizan por su alta velocidad de operación y el uso eficiente de la energía que utilizan, más la capacidad de enfriamiento con que cuentan. La Tabla 2 muestra las especificaciones del Clúster para la implementación de computación de alto rendimiento en este trabajo.

Tabla 2. Especificaciones técnicas del Clúster.

Nombre del sistema	Ranger
Conexión	ranger.tacc.utexas.edu
Sistema operativo	Linux (CentOS)
Número de nodos	3,936
Total procesadores	62,976
Procesadores por núcleo	4 Quad-Core AMD Opteron
Procesadores por nodo	16
Velocidad de procesamiento	2.0 GHz
Total memoria	123TB
Memoria del disco compartida	1.73PB
Memoria del disco local	31.4TB

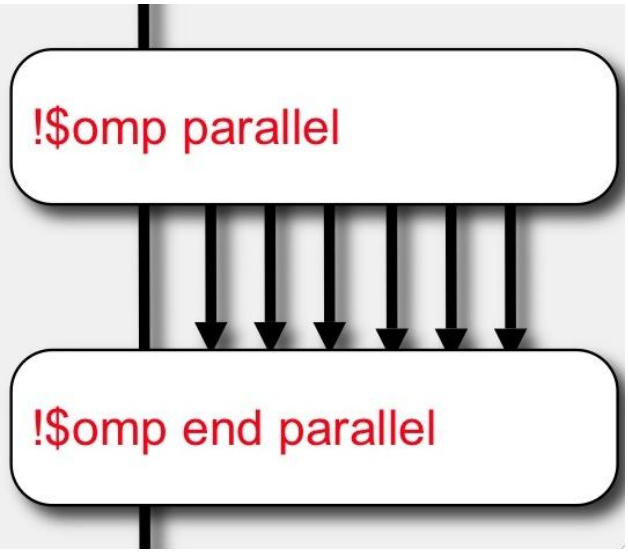
2.4 PROGRAMACIÓN PARALELA CON OpenMP

OpenMP es una aplicación estandarizada basada en una interfaz programada que se utiliza para escribir códigos paralelos con múltiples hilos de ejecución. OpenMP utiliza la programación paralela en todas las configuraciones de procesadores simétricos e incluye las plataformas de Unix/Linux y de Windows NT.

2.4.1 PROGRAMACIÓN CON MEMORIA COMPARTIDA

OpenMP es definido por un grupo de constructores de “hardware” y “software” de proveedores tales como Intel, HP, SGI, IBM, SOL, Compaq, KAI, PGI y otros. La primera aplicación de OpenMP apareció en 1998 y los compiladores que utilizan las aplicaciones de OpenMP en la actualidad incluyen los comandos que interpretan las directivas de todas las versiones de OpenMP. Las aplicaciones de la programación con memoria compartida son presentadas a los usuarios por medio de librarías especializadas, donde el mismo usuario especifica las acciones que deben realizar el compilador y el sistema operativo para ejecutar la programación paralela (Graham, 1999). La Ilustración 12 muestra el modelo de operación paralelo con seis hilos de ejecución.

Ilustración 12. Modelo de ejecución de OpenMP.



Chandra, 2001

La programación paralela con memoria compartida utiliza directivas de compilación, variables de entorno y librerías de ejecución que pueden ser utilizadas en programas escritos en los lenguajes Fortran 77/90 y C/C++. OpenMP se basa en estas directivas para ofrecer un modelo de programación paralelo flexible para diferentes aplicaciones.

Las directivas, rutinas y variables de entorno definidas en la aplicación de OpenMP permiten a los programadores crear códigos portables para ser ejecutados en diferentes sistemas operativos. Estas aplicaciones son ejecutadas bajo el modelo computacional de un único proceso con múltiple información. Este modelo permite que un problema traducido a un código computacional se divida en otros problemas más pequeños llamados tareas o hilos, con el fin de utilizar el poder de un súper computador para ejecutar las tareas de forma simultánea o paralela.

Un hilo en programación paralela se entiende como una entidad que tiene capacidad de almacenar información, la cual ejecuta o procesa datos en serie.
El modo de ejecución de OpenMP es flexible, de modo que dentro de una región que ejecuta hilos de manera paralela se puede especificar la cantidad de trabajo que cada hilo debe realizar. En memoria compartida para las

aplicaciones de OpenMP se pueden tener hilos de ejecución según los procesadores disponibles en una máquina.

La forma sintáctica para especificar la programación paralela con OpenMP es de la siguiente manera para programas escritos en lenguaje Fortran 77/90:

```
!$omp parallel do
        Do índice = 1, n
                Trabajo que se realiza dentro del ciclo
        END DO
        ¡$omp end parallel do
```

Para C/C++ la sintaxis para especificar programación paralela es:

```
#pragma omp parallel for
        for(índice = 0; n; índice++)
        {
        Trabajo que se realiza dentro del ciclo
        }
```

El anterior ejemplo muestra cómo se especifica programación paralela para ciclos iterativos en programas escritos en Fortran 77/90 y C/C++, sin embargo está incompleto porque falta especificar el tipo de variables que están dentro del bloque iterativo. El ejemplo expone la forma como específica al compilador que ejecute la aplicación en forma paralela, esto es mediante las directivas: ¡$omp parallel y #pragma omp parallel.

El modelo que ofrece OpenMP para paralelizar aplicaciones se basa en tres aspectos:

1. Especificaciones o directivas para ejecuciones concurrentes: La ejecución de hilos a través de diferentes procesos ocurre mediante librerías, funciones y variables de entorno.

2. Comunicación entre hilos de ejecución: La comunicación entre hilos de ejecución se especifica dentro de los mecanismos de control para asignar trabajo equivalente a cada proceso; es decir, se prefiere evitar

que varios procesos finalicen antes que algún proceso continúe en ejecución.

3. Sincronización explicita: Este método de comunicación entre procesos permite agrupar y asignar trabajo a diferentes hilos de ejecución con el fin de realizar el trabajo paralelo con el mejor uso de los recursos.

2.4.2 Código para calcular números primos con OpenMP

Con el fin de ilustrar el modo de aplicación y ejecución de una aplicación paralela con memoria compartida se presenta un código escrito en el lenguaje de programación Fortran con directivas de OpenMP. El código se presenta en el Anexo A. Con este problema se presenta una solución con óptima aceleración. Se muestra que entre más procesadores se utilizan, la aceleración crece de forma notable debido a que gran parte del código está en paralelo, y en este trabajo el ejemplo que se presenta a continuación sirvió como apoyo para los desarrollos propios.

Este programa calcula los números primos en un intervalo de números reales positivos. El código se encuentra en los ejemplos que ofrece Intel para ilustrar aplicaciones de programación paralela con OpenMP.

El programa presenta la implementación de OpenMP para acelerar un código computacional. En el código se utilizan funciones de OpenMP que permiten realizar operaciones dinámicas entre hilos de ejecución combinadas con procedimientos de sincronización entre actividades paralelas. Para esto se usa la opción “reducción”. En muchos casos esta opción ayuda a eliminar gastos indirectos ocasionados por el encabezado de las rutinas de OpenMP dedicadas a realizar sincronizaciones entre hilos.

El programador se encarga de ejecutar la aplicación de OpenMP al establecer la cantidad de iteraciones y el número de hilos de ejecución del código. El programa devuelve la cantidad de hilos que utiliza la aplicación, la cantidad de números primos calculados y el tiempo de total de ejecución. La Ilustración 13 muestra la ejecución del código ejemplo que suministra Intel con un sólo procesador, opción de compilación –O2.

Ilustración 13. Ejecución del programa con un procesador

```
login3% ./sample.out
  Range to check for Primes:          1   100000000
  We are using          1  thread(s)
  Number of primes found:     5761455
 Time for serial execution   586.027254104614
```

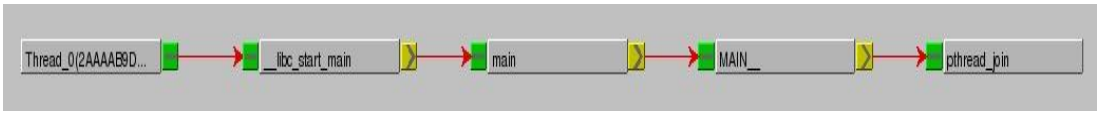
El código se ejecuta en primera instancia de forma serial (con un procesador). Al finalizar la ejecución se computa el tiempo total que tarda el código en calcular los números primos en el intervalo. El código tarda 586 segundos (nueve minutos y diez y seis segundos) en calcular los números primos en el intervalo de 1 a 1E8. En total se calculan 5761455 números primos dentro del intervalo.

La Ilustración 14 muestra el análisis realizado al código con el fin de ilustrar su comportamiento en un sólo procesador y el comportamiento secuencial de las instrucciones donde se evidencia que cada operación depende de la operación anterior.

Para realizar la evaluación del código se utiliza el programa “Vtune” de Intel. Este programa analiza el funcionamiento de las aplicaciones con el fin de identificar los puntos críticos que surgen cuando se accede a grandes porciones de memoria de la máquina y corresponden a la ejecución de la aplicación. En este caso el programa no muestra alertas o violaciones de memoria porque el algoritmo no exige demasiados recursos computacionales. El programa Vtune de Intel es efectivo para ejecutar operaciones de muestreo de datos y para direccionar la ejecución de las aplicaciones sobre el sistema operativo de la máquina con varios hilos de ejecución. La Ilustración 14 evidencia que sólo existe una cadena de ejecución.

La implementación de OpenMP se efectúa mediante el análisis de variables con el fin de establecer los grupos de variables privadas en cada hilo de ejecución y las variables compartidas para todos los procesadores.

Ilustración 14. Esquema del programa con un procesador



La Ilustración 15 muestra la solución que calcula el código después de utilizar dos procesadores. En total se utilizan dos hilos de ejecución paralelos que tardan 316 segundos (cinco minutos y veintiséis segundos) en calcular 576155 números primos en el intervalo 1 a 1E8. El código paralelo con dos hilos de ejecución es 1.85 veces más rápido que el código serial.

Ilustración 15. Ejecución del programa con dos procesadores

```
login3% ifort -openmp -fpp -o sample.out openmp_sample.f90
openmp_sample.f90(92): (col. 7) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
openmp_sample.f90(83): (col. 7) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
login3% ./sample.out
  Range to check for Primes:          1   100000000
  We are using           2  thread(s)
  Number of primes found:     5761455
 Time for parallel execution   316.164958000183
```

El programa paralelo con cuatro hilos de ejecución se analiza con la herramienta Vtune de Intel. Vtune muestra que durante la ejecución del programa con cuatro procesadores se crean cuatro hilos de procesamiento de instrucciones que se ejecutan de forma paralela (ver Ilustración 16). En esta ilustración se muestra la correcta división del problema para solucionar en paralelo a través de cuatro procesadores. Cada hilo de ejecución se encarga de computar las operaciones que se encuentran en la región paralela. En el interior de cada uno de los hilos se almacenan las variables privadas que establece el usuario en el momento en que diseña la aplicación paralela. Los hilos apuntan de manera continua al hilo principal que se encarga de almacenar las variables compartidas.

La implementación de OpenMP puede alcanzar máxima aceleración al aprovechar las cualidades de los superordenadores, dado que estos se desempeñan con el más alto nivel de funcionamiento.

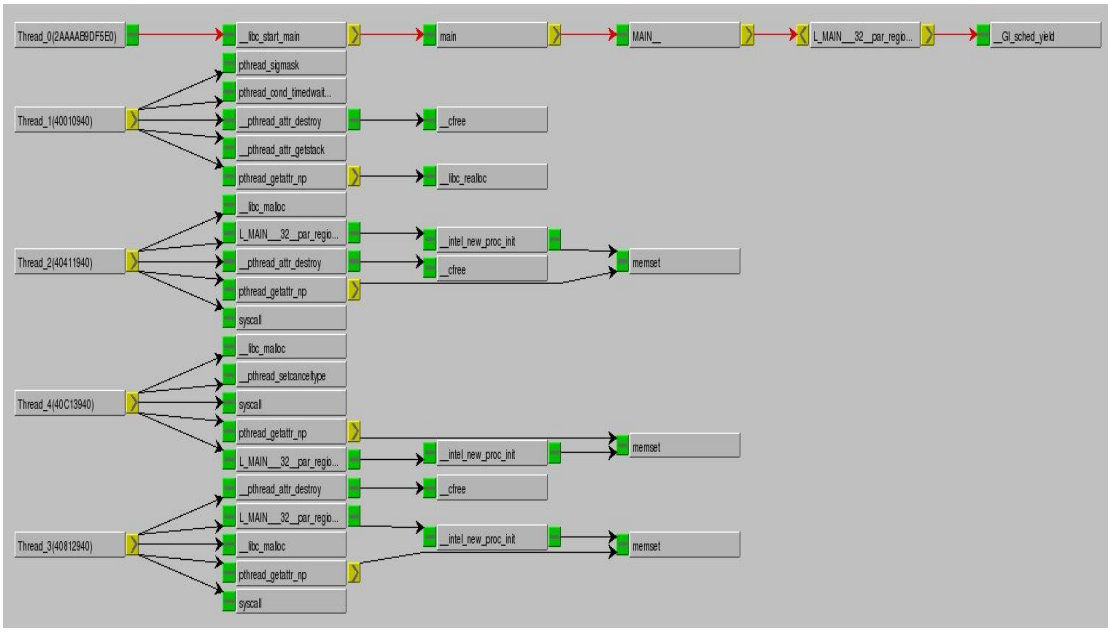
Ilustración 16. Ejecución del programa con cuatro procesadores.

```
login3% ifort -openmp -fpp -o sample.out openmp_sample.f90
openmp_sample.f90(92): (col. 7) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
openmp_sample.f90(83): (col. 7) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
login3% ./sample.out
  Range to check for Primes:          1   100000000
  We are using           4  thread(s)
  Number of primes found:     5761455
 Time for parallel execution   201.529462099075
login3%
```

Las aplicaciones que utilizan las funciones de OpenMP muestran mayor desempeño en conjuntos con varios procesadores conectados a la misma memoria o hacen parte de componentes con memoria compartida.

La Ilustración 17 muestra la correcta división del problema para solucionar en paralelo a través de cuatro procesadores mediante la herramienta de evaluación Vtune de Intel. Se evidencia que cada procesador tiene una carga de trabajo similar, lo que evita que un procesador calcule más operaciones que otro. Este es un claro ejemplo de un apropiado balance de trabajo entre procesadores.

Ilustración 17. Esquema del programa con cuatro procesadores.



La Ilustración 18 muestra la solución que calcula el código después de utilizar ocho procesadores. Los resultados de la ejecución muestran que el código tarda ciento siete segundos en calcular los números primos en el mismo

intervalo de 1 a 1E8. Este resultado muestra que el código es 5.47 veces más rápido que la ejecución del código con un procesador.

Ilustración 18. Ejecución del programa con ocho procesadores

```
login3% ifort -openmp -fpp -o sample.out openmp_sample.f90
openmp_sample.f90(92): (col. 7) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
openmp_sample.f90(83): (col. 7) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
login3% ./sample.out
  Range to check for Primes:          1   100000000
  We are using          8  thread(s)
  Number of primes found:     5761455
 Time for parallel execution   107.052415847778
```

La Ilustración 19 muestra la solución que calcula el código después de utilizar Diez y seis procesadores. Los resultados de la ejecución muestran que el código tarda setenta y seis segundos en calcular los números primos en el mismo intervalo de 1 a 1E8. Este resultado muestra que el código es 7.7 veces más rápido que la ejecución del código con un procesador.

Ilustración 19. Ejecución del programa con diez y seis procesadores.

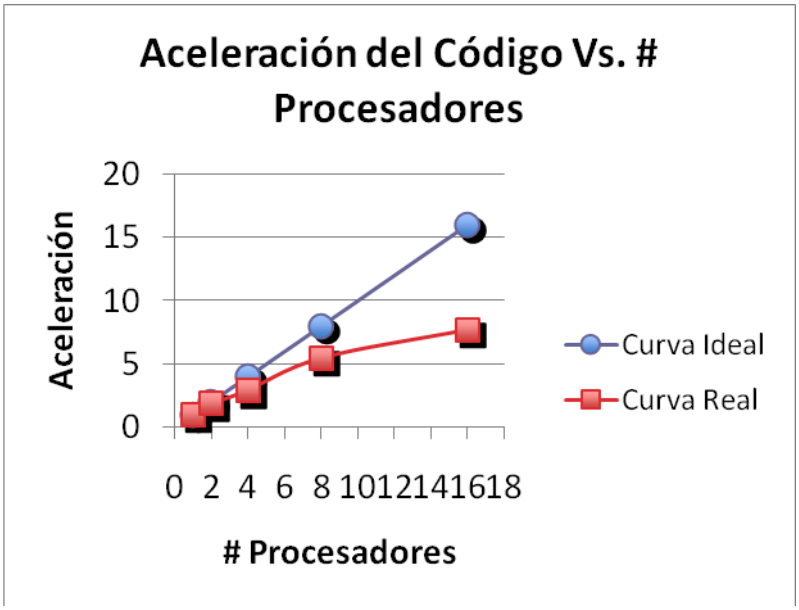
```
login3% ifort -openmp -fpp -o sample.out openmp_sample.f90
openmp_sample.f90(92): (col. 7) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
openmp_sample.f90(83): (col. 7) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
login3% ./sample.out
  Range to check for Primes:          1   100000000
  We are using         16  thread(s)
  Number of primes found:     5761455
 Time for parallel execution   76.0859398841858
login3%
```

La Ilustración 20 muestra la aceleración de la aplicación paralela de Intel que calcula números primos. Se muestra que existe una aceleración en la solución obtenida con el código paralelo similar a la aceleración ideal de las aplicaciones paralelas para 2 procesadores.

Las ejecuciones de aplicaciones paralelas en supercomputadores son muy eficientes dado que entre más procesadores se utilicen mayor resulta la aceleración de cualquier aplicación paralela. La aceleración no es linear porque se presentan retardos por comunicación y sincronización entre procesos. Entre más procesadores se utilizan en una aplicación, más comunicación se presenta entre los hilos de ejecución. El número de hilos de ejecución se establecen de

manera acorde al tamaño del código, puesto que cuando el código es pequeño las comunicaciones son redundantes entre procesos; es decir que, es posible que varios procesadores asignen la misma orden a cualquier proceso.

Ilustración 20. Aceleración de la aplicación de números primos.



2.4.3 Aproximación numérica con OpenMP mediante la regla del trapecio

La aproximación numérica con OpenMP para encontrar el área bajo la curva mediante la regla del trapecio requiere menos trabajo y esfuerzo que la paralelización de la aproximación con el modelo de memoria distribuida y las funciones de MPI. El código se presenta en el Anexo B. En el caso de la paralelización con OpenMP es necesario evaluar cada una de las variables y la relación entre ellas mismas con el fin de agrupar las variables privadas. La función $f(x) = x^2$ se utilizó para evaluar el método del trapecio.

El reconocimiento de las variables privadas en fundamental en las aplicaciones paralelas con OpenMP dado que estas variables deben ser actualizadas en cada iteración. Las variables compartidas por su parte se actualizan con actividades de lectura y escritura que se efectúan en los hilos paralelos de ejecución.

El código que encuentra la solución al problema con la regla del trapecio cuenta con cinco variables compartidas, estas variables son: el valor inicial del intervalo (a), el valor final del intervalo (b), el número de trapecios (n) y la base de cada trapecio (h). Las variables ¨a¨ y ¨b¨ son privadas porque cada hilo de ejecución debe conocer el valor del intervalo de evaluación en cada iteración, y dado que el intervalo es el mismo durante toda la ejecución del programa los valores de estas variables no necesitan ser almacenados más de una vez. En la Tabla 3 se muestran los datos iniciales del programa para ser ejecutados con un procesador.

Tabla 3. Datos iníciales del programa con un procesador.

a	0.0
b	1.0
n (trapecios)	100
Np (procesadores)	1

Las variables privadas que se guardan en la memoria de cada procesador o hilo de ejecución incrementan el tamaño de la aplicación. Por otra parte, si una variable que debe ser compartida se asigna privada por equivocación la solución del problema adquiere un valor diferente porque se presenta corrupción de variables. Esto sucede porque en cada iteración varios procesadores están actualizando la variable con incorrecta sincronización y a destiempo. La Ilustración 21 muestra la solución del código con un procesador.

Ilustración 21. Muestra la solución del código con un procesador

```
carlos:06 carlosacosta$ ifort -O2 -o TrapecioSerial.out SerialTrapecio.f90
SerialTrapecio.f90(24): (col. 7) remark: LOOP WAS VECTORIZED.
carlos:06 carlosacosta$ ./TrapecioSerial.out
 With n =         100  trapezoids, our estimate
 of the integral from   0.0000000E+00  to    1.000000      =   0.3333500
carlos:06 carlosacosta$
```

Para la segunda compilación y ejecución del programa se utilizaron dos procesadores y se incrementó el número de trapecios. La Tabla 4 muestra la inicialización de las variables para la segunda ejecución.

La Ilustración 22 muestra la solución del programa con 2 procesadores. Se evidencia que no existe ninguna corrupción de variables puesto que la respuesta es la misma que se obtuvo en la solución exacta.

Tabla 4 Datos iníciales del programa con dos procesadores

a	0.0
b	1.0
n (trapecios)	10000
Np (procesadores)	2

Ilustración 22. Muestra la solución del código con Dos procesadores.

```
carlos:06 carlosacosta$ ifort -openmp -o Parallel_Trape.out paralleltrap.f90
paralleltrap.f90(29): (col. 13) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
paralleltrap.f90(28): (col. 13) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
carlos:06 carlosacosta$ ./Parallel_Trape.out
 Current number of threads           2
 With n =       10000  trapezoids, our estimate
 of the integral from   0.0000000E+00  to    1.000000      =   0.3333433
carlos:06 carlosacosta$
```

Es importante corroborar los resultados de las aplicaciones paralelas, dado que pueden ocurrir errores de sincronización y comunicación entre hilos de ejecución.

2.5 PROGRAMACIÓN PARALELA CON MPI

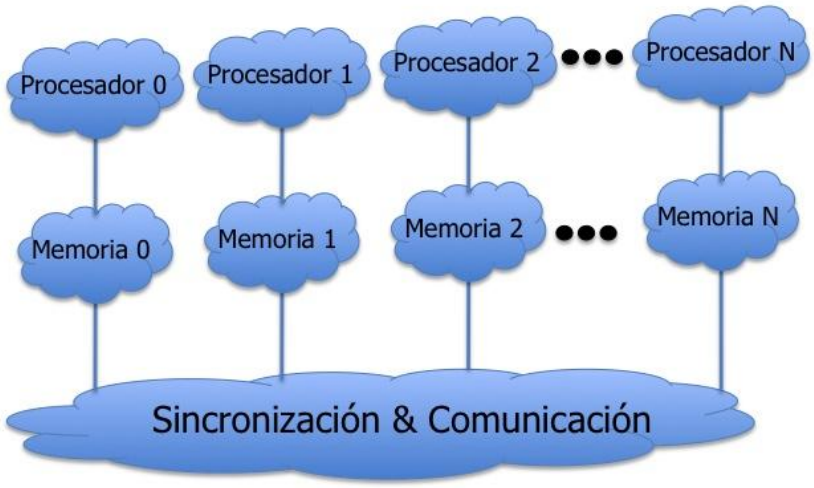
MPI es una interfaz que soporta el paso de mensajes a través de la red de intercomunicación en un Clúster o supercomputador. MPI es un modelo de programación paralela diseñado para que los sistemas con memoria distribuida utilicen aplicaciones paralelas mediante funciones y rutinas que transmiten datos entre procesos y se pueden utilizar en muchos sistemas operativos. Estas funciones se encuentran implementadas en librarías portátiles y estandarizadas tales como mvapich, MPICH y openmpi.

2.5.1 PROGRAMACIÓN CON MEMORIA DISTRIBUIDA

Las aplicaciones con memoria distribuida habilitan a cada procesador en el sistema para trabajar sólo con la memoria que le corresponde. La

responsabilidad de trabajar con la memoria de otros procesadores se transfiere de manera directa al programador. Los procesadores se comunican por medio de mensajes a través de una red que une varios procesadores.
El modelo de programación paralela mediante memoria distribuida tiene como característica principal el uso de la memoria local de cada procesador durante los procesamientos de datos. Esta identifica el grupo de tareas independientes que son computadas de forma concurrente a través de diferentes procesadores por medio de comunicaciones. La Ilustración 23 muestra el modelo de programación paralela con memoria distribuida

Ilustración 23. Modelo de programación con memoria distribuida.



Pacheco, 1997

La eficiencia de los cálculos depende de las comunicaciones oportunas que se lleven a cabo mediante mecanismos o funciones que permiten enviar y recibir mensajes por medio de redes de comunicaciones.

La implementación de esta técnica de programación paralela se lleva a cabo por medio de funciones y librerías estandarizadas que interactúa con los lenguajes de programación Fortran y C/C++. El usuario es responsable de asignar todas las porciones paralelas de la aplicación a diferentes procesadores.

La apropiada división de un problema en tareas es fundamental para obtener eficiente rendimiento de la ejecución de los programas paralelos. Se debe asegurar que diferentes procesos sean enviados a diferentes procesadores y que diferentes procesadores ejecuten diferentes tareas, puesto que es posible que varios procesadores intenten resolver las mismas tareas.

La coordinación de los procesos es importante para obtener resultados correctos, dado que implementaciones equivocadas pueden producir datos equivocados por lectura y escritura de variables escondidas en procesos que deben ser ejecutados en iteraciones posteriores.

2.5.2 Aproximación numérica con MPI para la regla del trapecio

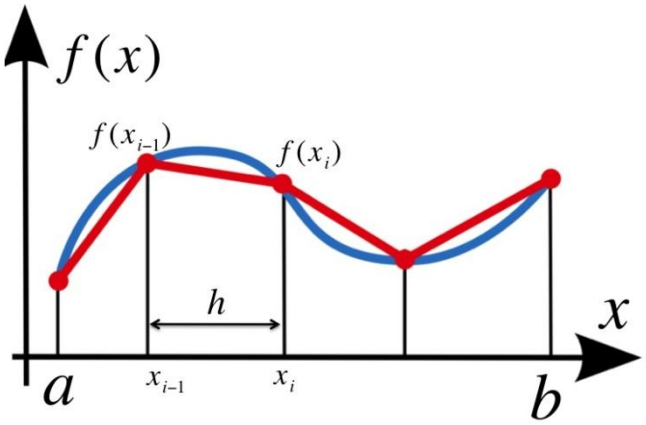
Con el fin de ilustrar el modo de programación en paralelo con MPI, se presenta un problema de integración con el método del trapecio. El código se presenta en el Anexo C. En este problema no se requiere comunicación entre tareas, por eso es apropiado utilizar la descomposición funcional del problema. Con este método se busca estimar el área bajo la curva para una función positiva (ver Ilustración 24). La regla del Trapecio establece que si hay "n" trapecios dentro de una región, el área de todos los trapecios más el factor de error en la aproximación corresponde al área bajo la cuerva descrita por la función en el plano y se puede calcular con la Ecuación 2. La función $f(x) = x^2$ se utilizó para evaluar el método del trapecio.

Ecuación 2. Área del trapezoide i.

$$\frac{1}{2} h[f(x_{i-1}) + f(x_i)]$$

Donde h corresponde a la base del trapecio, $f(x_{i-1})$ corresponde a la longitud del vértice izquierdo del trapecio y $f(x_i)$ corresponde a la longitud del vértice derecho del trapecio tal como lo muestra la Ilustración 24. La base "h" de cada trapecio es la misma dado que facilita las operaciones de distribución y recolección de información del problema entre todos los procesadores.

Ilustración 24. Regla del trapecio



La Ecuación 3 presenta el valor de "h" desde x_{i-1} hasta x_i en el intervalo $[a,b]$.

Ecuación 3. Base del trapecio

$$h = \frac{b-a}{n}$$

Donde:
a = valor inicial del intervalo a evaluar
b = valor final del intervalo a evaluar
n = número total de trapecios
h = base de cada trapecio

El intervalo del primer elemento es entonces $[a,a+h]$. El siguiente elemento tiene intervalo $[a+h,a+2h]$, el siguiente elemento tiene defino el intervalo $[a+2h,a+3h]$ hasta el último elemento que tiene intervalo $[a+(n-1)h,a+nh]$.

El área total de la región se puede calcular con la Ecuación 4, puesto que esta ecuación recopila todos los intervalos de cada elemento.

Ecuación 4. Área de toda la región

$$\frac{1}{2}h[f(x_0)+f(x_1)]+\frac{1}{2}h[f(x_1)+f(x_2)]+\ldots+\frac{1}{2}h[f(x_{n-1})+f(x_n)]$$

Existen muchos métodos para traducir las formulas de las reglas de aproximaciones numéricas a códigos computacionales (Pacheco, 1997).

También existen muchas estrategias para lograr paralelizar un programa en un nivel aceptable. El método que se presenta consiste en la división del dominio en pequeños grupos de tareas con el objetivo de solucionar las operaciones de cada tarea en diferentes procesadores para obtener la solución en menos tiempo.

Las operaciones que se ejecutan dentro de cada procesador son en esencia las mismas, la única diferencia radica en que las cadenas de instrucciones que se procesan son diferentes. El problema se inicializa con un intervalo de integración [a, b] en donde se almacenan “n” trapecios. La aplicación paralela diseñada por el programador debe estar en capacidad de dividir los elementos que están almacenados en el intervalo [a, b] por todos los procesadores disponibles para ejecutar el programa. Este código de integración numérica hace parte de los ejemplos propuestos por Peter Pacheco en su libro ¨Parallel Programming with MPI¨.

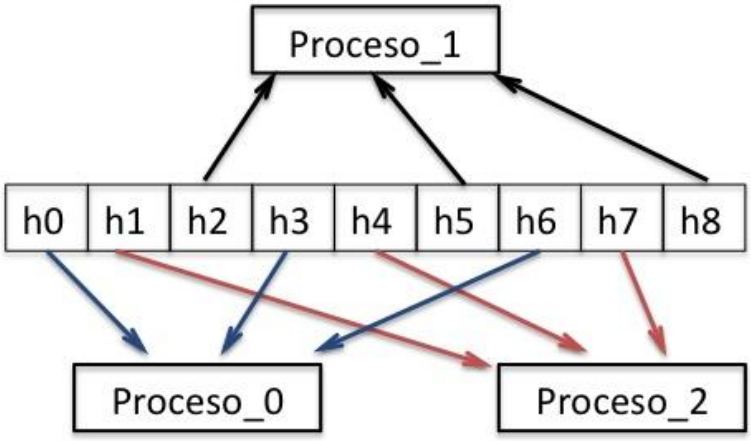
La división del problema en diferentes tareas se realiza para después reagrupar las instrucciones en procesos. Este es el paso inicial en toda aplicación paralela, la Ilustración 25 muestra el proceso de división y agrupación de actividades en procesos.

En estos problemas el dominio toma dimensiones considerable, por eso es importante realizar la sincronización correcta y las distribuciones apropiadas para la implementación de programación paralela entre diversos procesos (Pacheco, 1994). El método apropiado para enviar y recibir mensajes con porciones de información a los procesadores es crucial para diseñar una buena aplicación en paralelo.

El funcionamiento apropiado de las librearías y funciones de MPI dependen del programador, dado que en la fase de conceptualización y diseño del algoritmo debe tener en cuenta aspectos que influyen de forma directa con el apropiado funcionamiento del programa. Los aspectos principales que guían al programador hacia una apropiada implementación de estructuras paralelas tienen que ver con el número de procesadores que se esperan utilizar para la

ejecución del programa. MPI está enfocado a computadores con alto nivel de rendimiento, así que es común enfocar aplicaciones hacia el uso de cientos de procesadores (esto depende del tamaño de las instrucciones y del programa como tal).

Ilustración 25. División de tareas en procesos



El programa se ejecutó con diferentes grupos de procesadores con el fin de comparar las soluciones y evaluar aspectos de confiabilidad y eficiencia. La Ilustración 26 muestra el resultado obtenido después de ejecutar el código paralelo con diez núcleos de procesamiento y 100 elementos (trapecios).

La compilación del programa se llevó a cabo con el compilador de intel “ifort” para programas inscritos en el lenguaje de programación Fortran. La optimización aplicada corresponde a la opción de optimización –O2 con el fin de obtener los beneficios de operaciones de vectorización y transposición de datos.

La Tabla 5 muestra los datos iníciales del programa ejecutado con el estándar de paso de mensajes MPICH. El intervalo que se asigna en primera instancia para la evaluación de la función con la regla del trapecio es [0.0, 1.0]. Se establecen diez (10) procesadores y cien (100) elementos o trapecios para cubrir la región a evaluar. La solución del problema (ver Ilustración 26) muestra que el problema se divide en 10 tareas diferentes, en las cuales se evalúa el área bajo la curva con grupos de 10 trapecios.

Tabla 5. Datos iníciales del programa para la primera ejecución

a	0.0
b	1.0
n (trapecios)	100
Np (procesadores)	10

Ilustración 26. Ejecución del código paralelo con 10 procesadores.

```
[cacosta@shamu Trapezoide01]$ mpif90 -O2 -o trapezoid.out Trape.f90
Trape.f90(73): (col. 7) remark: LOOP WAS VECTORIZED.
Trape.f90(92): (col. 18) remark: LOOP WAS VECTORIZED.
Trape.f90(133): (col. 7) remark: LOOP WAS VECTORIZED.
[cacosta@shamu Trapezoide01]$ mpiexec -n 10 ./trapezoid.out
 ==> Intervalos=      100
 ==> Se dividen los intervalos entre los procesadores
 ==> Tarea    1: Rango =     1    10
 ==> Tarea    2: Rango =    11    20
 ==> Tarea    3: Rango =    21    30
 ==> Tarea    4: Rango =    31    40
 ==> Tarea    5: Rango =    41    50
 ==> Tarea    6: Rango =    51    60
 ==> Tarea    7: Rango =    61    70
 ==> Tarea    8: Rango =    71    80
 ==> Tarea    9: Rango =    81    90
 ==> Tarea   10: Rango =    91   100
Con n =  100, la respuesta es:
Intervalo de   0.00 a   1.00 =     0.33335
```

La aproximación numérica muestra que la solución es 0.33335 unidades cuadradas de área. Mientras que la solución exacta a la integral para $f(x) = x^2$ es 0.3333 como se muestra en la Ecuación 5.

Ecuación 5. Solución exacta

$$\int_0^1 x^2 dx = \frac{1^3 - 0^3}{3} = \frac{1}{3} \approx 0.3333$$

La Tabla 6 muestra la inicialización de las variables para la nueva ejecución del código computacional.

La Ilustración 27 muestra los rangos de división de tareas con 20 procesadores y 1000 trapecios. También muestra que la aproximación numérica es 0.3333. La correcta distribución de información a través de tareas se cumple cuando el número de trapecios se puede dividir de manera exacta a través de los procesadores disponibles. Este ejemplo muestra el método de implementación

de programación paralela con MPI en sistemas con memoria distribuida. Por ser una aproximación numérica mediante intervalos muestra la forma de dividir una un problema en pequeñas tareas.

Tabla 6. Datos iníciales del programa para la segunda ejecución

a	0.0
b	1.0
n (trapecios)	1000
Np (procesadores)	20

Ilustración 27. Ejecución del código paralelo con 20 procesadores.

```
[cacosta@shamu Trapezoide01]$ mpif90 -O2 -o trapezoid.out Trape.f90
Trape.f90(73): (col. 7) remark: LOOP WAS VECTORIZED.
Trape.f90(92): (col. 18) remark: LOOP WAS VECTORIZED.
Trape.f90(136): (col. 7) remark: LOOP WAS VECTORIZED.
[cacosta@shamu Trapezoide01]$ mpiexec -n 20 ./trapezoid.out
  ==> Intervalos=     1000
  ==> Se dividen los intervalos entre los procesadores
  ==> Tarea    1: Rango =     1    50
  ==> Tarea    2: Rango =    51   100
  ==> Tarea    3: Rango =   101   150
  ==> Tarea    4: Rango =   151   200
  ==> Tarea    5: Rango =   201   250
  ==> Tarea    6: Rango =   251   300
  ==> Tarea    7: Rango =   301   350
  ==> Tarea    8: Rango =   351   400
  ==> Tarea    9: Rango =   401   450
  ==> Tarea   10: Rango =   451   500
  ==> Tarea   11: Rango =   501   550
  ==> Tarea   12: Rango =   551   600
  ==> Tarea   13: Rango =   601   650
  ==> Tarea   14: Rango =   651   700
  ==> Tarea   15: Rango =   701   750
  ==> Tarea   16: Rango =   751   800
  ==> Tarea   17: Rango =   801   850
  ==> Tarea   18: Rango =   851   900
  ==> Tarea   19: Rango =   901   950
  ==> Tarea   20: Rango =   951  1000
 Con n = 1000, la respuesta es:
 Intervalo de   0.00 a   1.00 =     0.33333
[cacosta@shamu Trapezoide01]$
```

3 SOLUCIÓN DE SISTEMAS LINEALES DE EN PARALELO.

La eliminación Gaussiana es un método que se aplica para resolver sistemas lineales de ecuaciones tal como lo muestra la Ecuación 6 para sistemas de ecuaciones o matrices de n*n.

Ecuación 6. Sistema lineal de ecuaciones.

$$A * \mathrm{X} = \mathrm{B}$$

Este método permite la multiplicación en ambos lados de la igualdad por una constante diferente de cero. Las ecuaciones pueden sumarse entre sí con el fin de resolver una sustitución de variables para después despejar incógnitas.

El sistema lineal de ecuaciones toma la siguiente en arreglo vectorial para facilitar las operaciones matriciales.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Los sistemas con muchas ecuaciones necesitan ordenares de información y procesadores que computen la solución algebraica para obtener la solución al sistema. Para sistemas de ecuaciones con muchas variables, un solo computador puede tardar bastante tiempo en calcular la solución completa. Por esta razón la computación paralela ofrece tantas ventajas dado que logra realizar los cálculos de forma distribuida y particionada con el fin de acelerar la ejecución del problema.

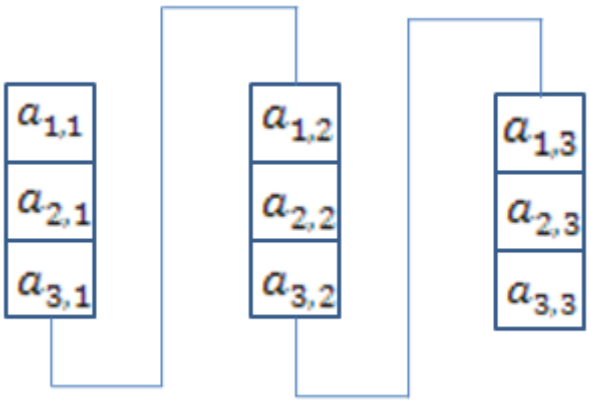
La forma clásica de acceder a la información ubicada en el arreglo bidimensional con un procesador consiste en utilizar un lenguaje de programación adecuado que permita distribuir la información en la memoria de la máquina. Los lenguajes compilados como Fortran o ANSI C utilizan posiciones contiguas de memoria para ubicar la información (Golub, 1996).

Este método de almacenamiento y lectura de datos en memoria tiene importantes cualidades en cuanto aceleración en cálculos y computación se refiere, dado que todos los elementos se encuentran próximos y resulta inviable crear funciones o subrutinas adicionales que deban encontrar la información en el disco.

3.1 FORMA SECUENCIAL DEL ALGORITMO

El método clásico para resolver sistemas lineales de ecuaciones consiste en eliminar incógnitas en las ecuaciones para resolverlas simultáneamente. Esta solución se lleva a cabo en fases. La Ilustración 28 muestra la el almacenamiento en memoria de de una matriz en $\mathcal{R}^{3x3}$.

Ilustración 28. Almacenamiento matricial por columnas.



Golub, 1996

La forma general que debe adoptar el algoritmo está dividida en dos pasos.

1. Las ecuaciones se trasforman para eliminar incógnitas de modo que al final de la operación de eliminación se obtenga una sola incógnita por ecuación. Esta operación se denomina eliminación hacia adelante.
   Como se ha dicho, el propósito es reducir el sistema de ecuaciones para hallar cada incógnita. El proceso comienza mediante la eliminación de los coeficientes en la primera columna del arreglo matricial a partir de la segunda ecuación hasta la última ecuación. Después se debe eliminar el coeficiente de la segunda columna desde la tercera ecuación. Continúa

este proceso sucesivo hasta la columna n-1. De esta manera se asegura que la última columna sólo contenga una variable desconocida. En el sistema de ecuaciones, para eliminar el coeficiente a_{ij} de la $i$ésima ecuación, la ecuación j debe ser multiplicada por $\frac{-1}{a_{ij}}$ y se suma a la ecuación i.

2. Cada ecuación se puede resolver de forma directa en un proceso inverso, sustituyendo la incógnita encontrada en un proceso secuencial. Este mecanismo se denomina sustitución hacia atrás.

   Después de aplicar la eliminación hacia adelante, la matriz original se transforma a un sistema matricial triangular superior, y la última ecuación sólo tiene una variable desconocida y queda de la forma:

   $a_{n,n}c = b_n$

   El término desconocido es x_n, y se encuentra al remplazar en la ecuación (n-1) para encontrar el valor de x_{n-1} de la siguiente manera:

   $a_{n-1,n-1}x_{n-1} + a_{n-1,n}x_n = b_n$

   Después de aplicar la eliminación hacia adelante se obtiene un sistema de ecuaciones de la siguiente manera.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & a'_{2,2} & a'_{2,3} \\ 0 & 0 & a''_{3,3} \end{bmatrix} * \begin{bmatrix} c_1 * x_1 \\ c'_2 * x_2 \\ c''_3 * x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

El caso general para aplicar la eliminación hacia adelante es:

For i = 1 to n

División de la $i$ésima fila por $a_{i,i}$

$$Ecuación\ i = {}^{E_i}/_{a_{i,i}}$$

Ceros en la columna i.

For j > i

$E_j = E_j - a_{j,i}E_i$

End

End

La sustitución hacia atrás permite encontrar las variables que permiten encontrar la solución al sistema de la siguiente manera.

$$x_3 = {c''_3}/{a''_{3,3}}$$

$$x_2 = {(c'_2 - a'_{2,3} * x_3)}/{a'_{2,2}}$$

$$x_1 = {(c_1 - a_{1,2} * x_2 - a_{1,3} * x_3)}/{a_{1,1}}$$

El caso general para aplicar la sustitución hacia atrás es:

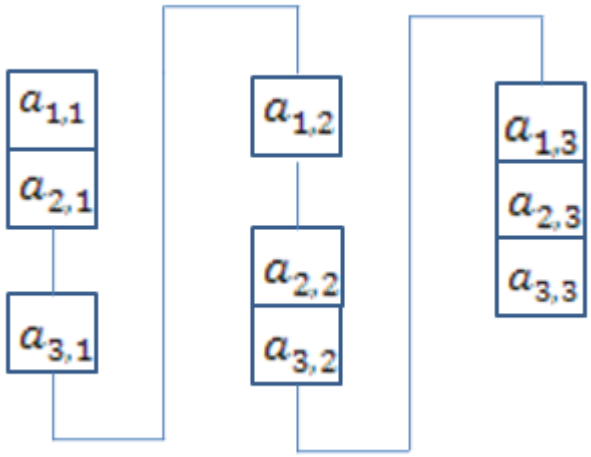
```
For j = n-1
    Temp=0
        For k=j+1 to n
            Temp=temp+a_{j,k} * x_k
        End
        x_i = (c_j - temp)/a_{j,j}
End
```

3.2 SOLUCIÓN POR BLOQUES A SISTEMAS DE ECUACIONES

El método de solución de sistemas de ecuaciones por bloques de información consiste en utilizar mecanismos de ejecución de múltiples tareas paralelas para un sistema lineal de ecuaciones con varias incógnitas. Cada hilo de ejecución representa la información que computa un procesador, así que es de esperarse que a medida que se usen más hilos de ejecución, el tiempo total de ejecución del programa se reduzca. La solución por bloques a sistemas de ecuaciones se efectúa mediante los siguientes pasos.

1. La descomposición del dominio se aplica a la lectura matricial mediante la división por bloques de información; es decir que se parte el dominio con el fin de acelerar el proceso de ubicación de información en memoria. La Ilustración 29 muestra el almacenamiento matricial en memoria por bloques. En este proceso se divide la información en pedazos de instrucciones con tamaño aproximadamente igual para ofrecer mayor efectividad en las operaciones paralelas a través de los procesadores.

Ilustración 29. Almacenamiento matricial por bloques.



Golub, 1996

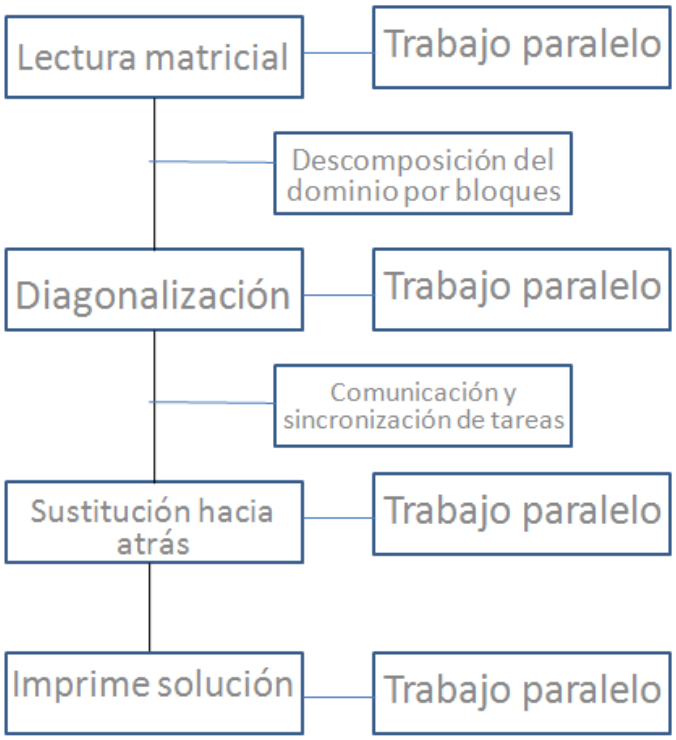
2. La descomposición funcional es la manera de dividir el problema en trozos o tareas paralelas. Se enfoca en la ejecución de operaciones a través de tareas que son concurrentes en el tiempo con base en la información disponible para resolver el problema. Esto es, enviar información a través de los procesadores para computar trabajos en paralelo.

3. La transformación de las ecuaciones se realiza para eliminar incógnitas de modo que al final de la operación de eliminación se obtenga una sola incógnita por ecuación. Esta operación se denomina eliminación hacia adelante. El propósito es reducir el sistema de ecuaciones para hallar cada incógnita. El proceso comienza mediante la eliminación de los coeficientes en la primera columna del arreglo matricial a partir de la segunda ecuación hasta la última ecuación. Después se debe eliminar el coeficiente de la segunda columna desde la tercera ecuación. Continúa este proceso sucesivo hasta la columna n-1. De esta manera nos aseguramos de que la última columna sólo contenga una variable desconocida.

4. La comunicación entre tareas se realiza con los datos almacenados en varias tareas. En este caso, la comunicación y sincronización es necesaria para mover información entre las tareas. Los datos divididos en tareas pueden ser en muchos casos la información de entrada del programa, la salida calculada por el programa, o valores intermedios utilizados por el programa. Existen varias maneras de afrontar la

descomposición de datos, dado que diversas descomposiciones pueden ser realizadas. Unas divisiones pueden basarse en reorganizar y dividir estructuras de datos, otras se concentran primero en la estructura de datos principal o en la estructura de datos que se accede con más frecuencia.

Ilustración 30. Secuencia paralela del algoritmo.



3.3 FORMA PARALELA DEL ALGORITMO

La programación por bloques ofrece ventajas en ciertos casos por encima de la programación serial con almacenamiento matricial por columnas, dado que cuando se realizan operaciones con matrices grandes en dimensión, resulta más rápido y efectivo utilizar particiones de datos para almacenar la matriz. En la programación paralela, el almacenamiento por bloques es efectivo cuando se utilizan varios procesadores para almacenar en memoria bloques de información. En este tipo de programación acceder a los datos es un poco más complicado pero se cumple la misma regla de almacenamiento contiguo; es decir que, los bloques son almacenados en memoria aledaños entre sí.

Con el fin de obtener una solución acelerada al sistema lineal de ecuación es viable utilizar una máquina con varios procesadores y un lenguaje de programación de alto rendimiento que permita computar y solucionar operaciones en paralelo a través de los procesadores.

El caso general para aplicar la eliminación hacia adelante en paralelo es:

Variables privadas = i, j, a
Variable compartida = n, E
For i = 1 to n (en paralelo)
División de la iésima fila por $a_{i,i}$
$Ecuación\ i = {}^{E_i}/_{a_{i,i}}$

Ceros en la columna i.
For j > i
$E_j = E_j - a_{j,i}E_i$
End
End

La sustitución hacia atrás se computa de forma paralela a partir de la segunda iteración puesto que en el primer ciclo sólo se calcula una incógnita:

Variables privadas = i, j, Temp, a, x, Nprocs
Variable compartida = n, c
For j = n-1 (en paralelo) IF Nprocs>0
Temp=0
For k=j+1 to n
Temp=Temp+$a_{j,k} * x_k$
End
$x_i = (c_j - temp)/a_{j,j}$
End

3.4 SOLUCIÓN DE LA ECUACIÓN DE LAPLACE EN FORMA PARALELA PARA FLUJO POTENCIAL POR EL MÉTODO DE ELEMENTOS FINITOS.

En esta sección se presenta un problema de flujo potencial, se trata de la ecuación de La place para flujo incompresible (ver Ecuación 7) y su solución por el método de elementos finitos. Con el fin de comparar los resultados de la programación paralela con los resultados de la implementación serial.

La Ecuación 7 muestra la ecuación de La place para un flujo incompresible.

Ecuación 7. Ecuación de Laplace para flujo incompresible

$$\nabla^2 U = 0$$

Donde:

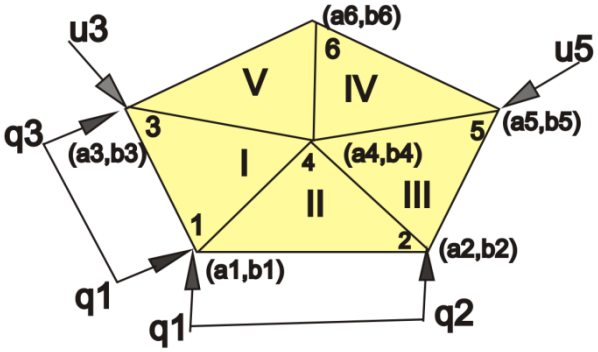
∇^2 = es el operador Laplaciano y $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$

$\mathbf{U}$ = es el campo escalar de temperatura $U(x, y)$.

El método de elementos finitos consiste en resolver ecuaciones diferenciales parciales con el fin de encontrar una aproximación a cierta solución para un problema físico específico. La geometría del cuerpo que se estudia se divide en formas simples triangulares que permiten conformar el contorno de la figura y se representa con los datos de los vértices que son almacenados en el sistema junto con condiciones de frontera o restricciones específicas (temperaturas o flujos de calor conocidos). La Ilustración 31 muestra la topología de un problema de elementos finitos en dos dimensiones. En la Figura se enseñan los puntos o nodos que caracterizan la geometría del problema y se muestran las coordenadas de los vértices. Las flechas en la Ilustración 31 muestran la localización de las restricciones o condiciones de frontera que se ubican en el contorno de la región. Se evidencia que en el problema existen dos tipos de condiciones de frontera. Estos correspondes a: restricción de temperatura, *u* y restricción de flujo de calor, *q.* La Ilustración 31 muestra una malla que contiene cinco elementos y seis nodos, una restricción de temperatura en los nodos tres y cinco y una restricción de flujo de calor en los vértices (1,3) y (1,2).

La solución de la ecuación de La place en dos dimensiones para flujo potencial por el método de elementos finitos se puede resolver con una aproximación lineal (Kwon et al) que se presenta en la Ecuación 8.

Ilustración 31. Región definida por cinco elementos triangulares.



Ecuación 8. Aproximación lineal.

$$\mathbf{ax} + \boldsymbol{by} + \boldsymbol{c}$$

Ecuación 9. Combinación lineal del sistema

$$A^D u^D = l^D - q^D$$

La solución del sistema lineal que se presenta en la ecuación 3 representa la combinación lineal de la matriz que contiene la información de la topología del cuerpo y los vectores que contienen los datos de las condiciones de frontera. El programa paralelo se encarga de solucionar este sistema a través de varios procesadores mediante el método de eliminación Gaussiana.

4 RESULTADOS

A continuación se presentan los resultados de la aceleración obtenida con procesamiento paralelo de los códigos computacionales programados en forma paralela y en forma serial, que se encargan de resolver por una parte el método de eliminación Gaussiana, donde se muestran los resultados de aceleración para el código serial y el código paralelo, y por otra parte se muestra la solución de la ecuación de La place. Le solución de la ecuación se presenta de forma paralela como fue enunciado en el capítulo tres.

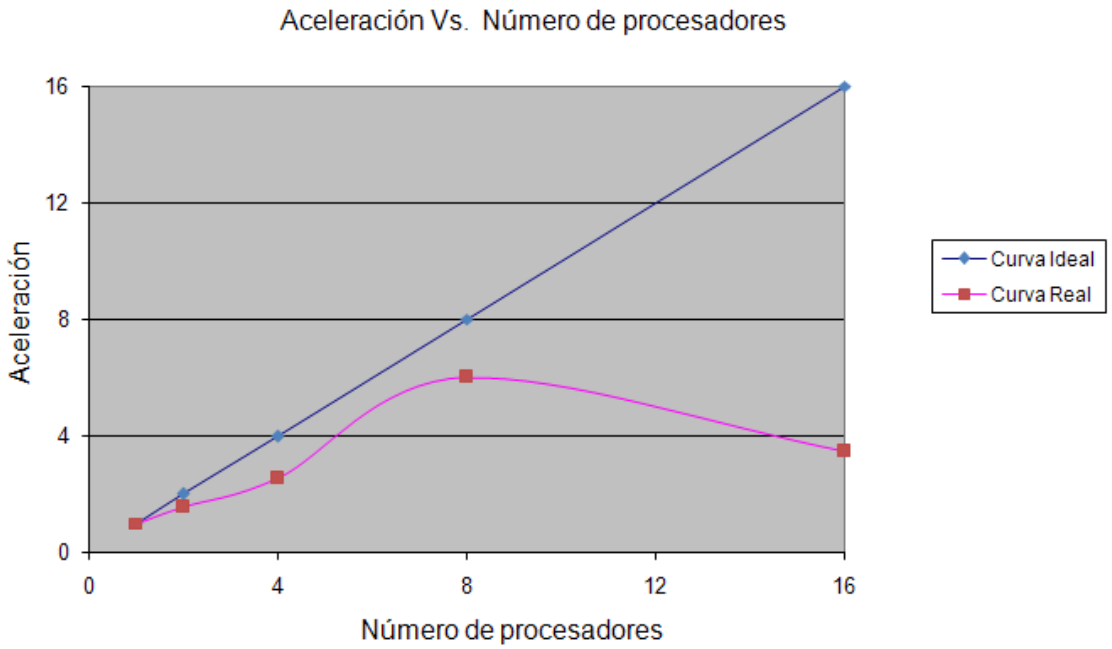
La validación de los resultados se muestran por medio de la comparación de resultados que arroja el código serial de la solución de sistemas lineales de ecuaciones mediante la eliminación Gausiana (el código se presenta en el Anexo D) y la solución de la ecuación de La place para flujo potencial (el código se presenta en el Anexo E). Los resultados de ambos códigos fueron comparados con su correspondiente código paralelo con el fin verificar la adecuada programación estructurada por bloques y partición correcta del problema.

La razón principal que impulsa al programador a aplicar programación paralela es obtener los beneficios de aceleración en el tiempo de ejecución en las aplicaciones. La Ilustración 32 muestra la aceleración de la aplicación paralela de la eliminación Gaussiana después de realizar la ejecución con Dos (2), Cuatro (4), Ocho (8) y Diez y seis (16) procesadores. La Tabla 7 muestra los valores de aceleración obtenidos para la solución del método de sustitución de Gauss para una matriz bandeada de 1600 elementos.

Tabla 7. Resultados de aceleración para la eliminación Gaussiana

Número de procesadores	Aceleración
1	1
2	1,56937273
4	2,5499193
8	5,99673766
16	3,47853777

Ilustración 32. Aceleración de la eliminación Gaussiana.



Se evidencia que con Ocho (8) procesadores ocurre la máxima aceleración (ver Tabla 7), dado que con ocho procesadores el código es 5.99 veces más rápido. Con Dos (2) y Cuatro (4) procesadores ocurre una aceleración progresiva que se asimila a la aceleración ideal de una aplicación paralela. Sin embargo, con Diez y seis procesadores cae nuevamente la aceleración en la ejecución de la aplicación dado que resulta muy costoso soportar comunicaciones y sincronizaciones entre 16 procesadores de forma simultánea.

La solución de la ecuación de Laplace para flujo potencial muestra un leve incremento en la aceleración en la ejecución de la aplicación a medida que incrementa el número de procesadores disponibles para computar las

operaciones (ver Ilustración 33). Se evidencia que con Ocho (8) procesadores ocurre la máxima aceleración (ver Tabla 8), dado que con ocho procesadores el código es casi dos veces más rápido veces más rápido que con un solo procesador. Con Dos (2) y Cuatro (4) procesadores ocurre una pequeña aceleración producto de la cantidad de ciclos que no se pueden paralelizar por la insuficiencia de datos a computar dentro de ellos.

Ilustración 33. Aceleración de la solución de la ecuación de Laplace.

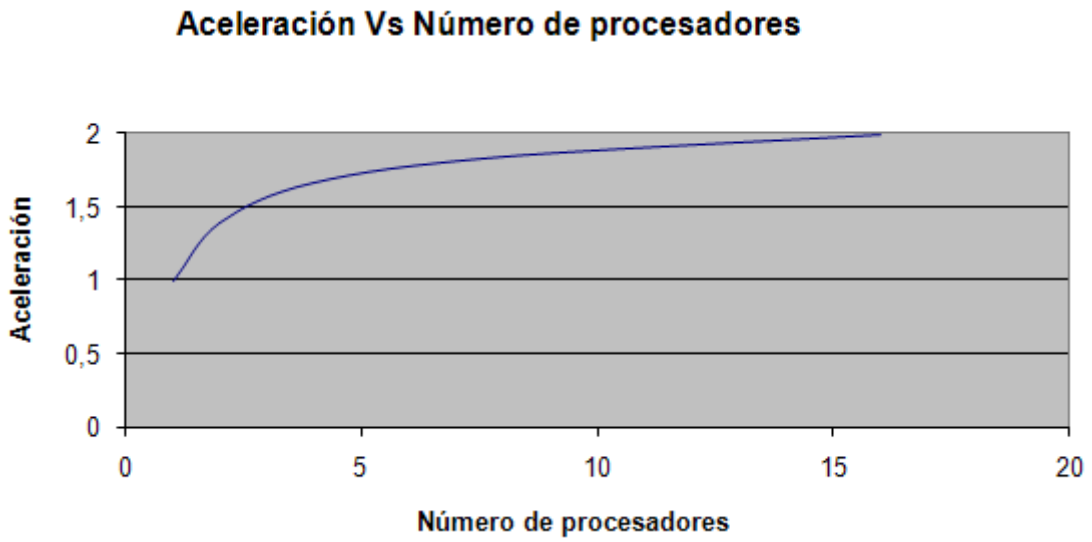
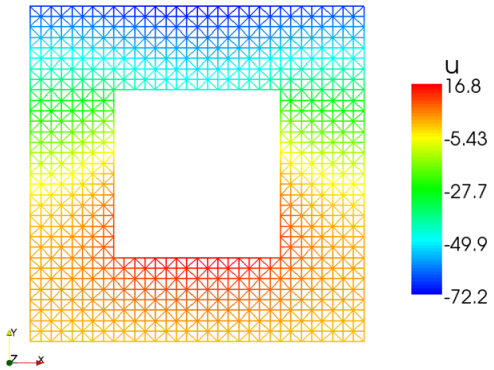


Tabla 8. Aceleración para la solución de la ecuación de Laplace.

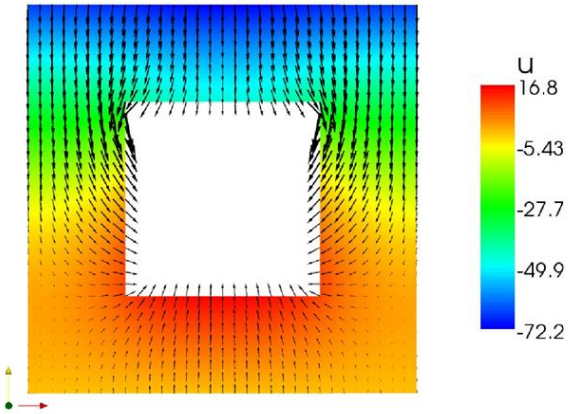
Número de procesadores	Aceleración
1	1
2	1,3903
4	1,6684
8	1,8430
16	1.9961

Ilustración 34. Solución de Laplace para el campo de temperatura.



La solución de la ecuación de la Place en forma paralela para flujo potencial en una placa plana en dos dimensiones tiene una ventaja significativa sobre los demás código implementados en forma paralela. Se trata de la forma de las pocas comunicaciones y sincronizaciones que se necesitan para partir el problema y ejecutarlo en diferentes procesadores. Porque la implementación se realizó mediante la partición del dominio, el cual no cuenta con dependencia de datos o barreras para sincronizar actividades.

Ilustración 35. Solución de Laplace para el campo de velocidad.



La solución de la ecuación de Laplace en forma paralela para flujo potencial cuenta con un atenuante. Se trata de la imposibilidad de paralelizar por completo todo el programa, dado que existen ciclos con pocas operaciones que no aportan ninguna aceleración a la ejecución de la aplicación.

El post procesamiento de la ejecución del programa arrojó la Ilustración 34 y la Ilustración 35. Estas figuras muestran son iguales para la solución de la

ecuación de La place para flujo potencial en serial y en paralelo. Esto demuestra que la programación en paralelo fue correcta, y que no hubo corrupción de datos durante la sincronización y comunicación de información. La Ilustración 35 corresponde a la solución de la ecuación de La place para flujo potencial. Se muestra que a partir de las temperaturas se identifica el sentido del flujo.

5 CONCLUSIONES Y RECOMENDACIONES

- Se describieron los conceptos básicos sobre la programación paralela y se mostró que su aplicación permite disminuir el tiempo de ejecución en códigos computacionales. Se presentaron algunos códigos que se encuentran en la literatura, y se desarrollaron códigos propios.

- La computación paralela permite disminuir el tiempo de ejecución de un proceso computacional. Para evidenciar esta característica, en este trabajo se implementó un algoritmo paralelo para resolver un sistema de ecuaciones lineales por el método de Gauss.

- Se utilizó el algoritmo de eliminación Gaussiana para resolver un problema de flujo potencial mediante la ecuación de Laplace por el método de elementos finitos y la solución se comparó entre la programación serial y la programación paralela para evaluar los tiempos de ejecución en cada caso (el programa se ejecutó con dos, cuatro, ocho y diez y seis procesadores) y se logró obtener una disminución en el tiempo de ejecución, con una aceleración máxima de hasta cinco veces más rápido utilizando ocho procesadores.

- Se mostraron diferentes opciones para paralelizar programas computacionales que se usan para calcular soluciones en problemas de ingeniería. Se resolvieron diferentes problemas con memoria distribuida y con memoria compartida mediante el uso de mecanismos de descomposición funcional y descomposición del dominio propias de cada método para balancear las tareas.

- Los resultados obtenidos muestran los beneficios de la programación paralela, en el sentido en que acelera la solución, pero se evidencia que es necesario comprender ampliamente los mecanismos y herramientas que permiten paralelizar aplicaciones o códigos computacionales. En los

casos expuestos se mostró que utilizar programación paralela tiene beneficios importantes por la reducción en el tiempo de ejecución de los programas. Sin embargo para la solución de la ecuación de Laplace para flujo potencial no se obtuvo una aceleración tan significativa dado que muchas porciones del código computacional no son paralelizables.

- Para obtener valores de aceleración adecuados en aplicaciones paralelas se recomienda utilizar supercomputadores o Clústers para ejecutar los códigos computacionales, dado que estas máquinas cuentan con muchos núcleos de procesamiento y redes de comunicación de alto rendimiento.

- Es recomendable ejecutar las aplicaciones en sistemas operativos o ambientes de ejecución tales como Linux o Unix. En este proyecto no se ejecutaron las aplicaciones en un sistema operativo Unix por limitaciones técnicas de Hardware y número apropiado de procesadores.

6 BIBLIOGRAFÍA

Chapma B, Jost G, Van Der Pas R, "Using Openmp", Massachusetts Institute of Technology, 2008, ISBN-13: 978-0-262-53302-7.

Graham P, OPENMP A Parallel Programming Model For Shared Memory Architectures, Edinburgh Parallel Computing Center, The University of Edinburgh, March 1999.

PACHECO P, Parallel Programming with MPI, Morgan Kaufmann Publishers, San Francisco, USA, 1997, ISBN 1-55860-339-5.

CHANDRA C, DAGUN D, KOHR D, MAYDAN D, MACDONALD J, MENON R, Parallel Programming in OpenMP, Morgan Kaufman Publishers, 2001, ISBN 1-55860-671-8.

ELLIS T, PHILIPS I, LAHEY T, Fortran 90 Programming, Addison-Wesley, 1994, ISBN 0-201-54446 6.

QUINN M, Parallel Programming in C with MPI and OpenMP, Mc Graw Hill, Oregon, USA, 2004, ISBN 0-07-2882256-2.

Texas Advanced Computing Center (TACC), Introduction to Parallel Programming on Ranger and Lonestar, 2009, NSF OCI-0749334.

QUE CORPORATION, Using Unix Systems, Carmel, India, 1990, ISBN 0-88022-512-x.

ETEER D, INGBER J, Engineering Problems Solving with C, 2nd edition, Prentice Hall, New Jersey, USA, 2000, ISBN 0-13-010930-4.

LOUKIDES M, System Performance Tuning for Unix System, O'Reilly & Associates, Inc, 1992, USA, ISBN 0-937175-60-9.

REVOLUTION COMPUTING, Using ParallelR ™ for High Performance Monte Carlo Simulation on Multiprocessor Computers, Revolution Computing Inc, www.revolution-computing.com, 2003.

VARGAS G. JAVIER, Computación evolutiva aplicada a procesos de diseño de Circuitos electrónicos, Seminario de Investigación I, Noviembre, 2006, Bogotá D.C, Colombia.

Golum G, Van Loan C, Matrix Computations, Third, edition, The Johns Hopkins Press, Baltimore, 1996, ISBN 0-8018-5414-8.

Kwon Young, Bang Hyochoong, The Finite Element Method Using Matlab, Second Edition, Crc Press, ISBN: 0849300967.

7 ANEXOS

ANEXO A Programa en lenguaje Fortran para el cálculo de números primos con OPENMP

```
1  !INTEL PROGRAM
2  program ompPrime
3  USE omp_lib
4
5  !#ifdef _OPENMP
6   !  include 'omp_lib.h'  !needed for OMP_GET_NUM_THREADS()
7  !#endif
8
9  IMPLICIT NONE
10 DOUBLE PRECISION T_IN, T_END !NEW LINE
11
12 integer :: start = 1
13 integer :: end = 50
14 integer :: number_of_primes = 0
15 integer :: number_of_41primes = 0
16 integer :: number_of_43primes = 0
17 integer index, factor, limit, nthr
18 real rindex, rlimit
19 logical prime, print_primes
20
21 CALL omp_set_num_threads(8) !new line
22
23 print_primes = .false.
24 nthr = 1 ! assume just one thread
25 print *, ' Range to check for Primes:',start,end
26
27 T_IN = omp_get_wtime()
28
29 #ifdef _OPENMP
30 !$omp parallel
31
32 !$omp single
33    nthr = OMP_GET_NUM_THREADS()
34    print *, ' We are using',nthr,' thread(s)'
35 !$omp end single
36 !
37 !$omp do private(factor, limit, prime) &
38    schedule(dynamic,10) &
39    reduction(+:number_of_primes,number_of_41primes,number_of_43primes)
40 #else
41    print *, ' We are using',nthr,' thread(s)'
42 #endif
43
44 do index = start, end, 2   !workshared loop
45
46    limit = int(sqrt(real(index)))
47    prime = .true.  ! assume number is prime
48    factor = 3
49    do
50       if(prime .and. factor .le. limit) then
51          if(mod(index,factor) .eq. 0) then
52             prime = .false.
```

```
53          endif
54          factor = factor + 2
55       else
56          exit  ! we can jump out of non-workshared loop
57       endif
58    enddo
59
60    if(prime) then
61       if(print_primes) then
62          print *, index, ' is prime'
63       endif
64
65       number_of_primes = number_of_primes + 1
66
67       if(mod(index,4) .eq. 1) then
68          number_of_41primes = number_of_41primes + 1
69       endif
70
71       if(mod(index,4) .eq. 3) then
72          number_of_43primes = number_of_43primes + 1
73       endif
74
75    endif   ! if(prime)
76 enddo
77 !$omp end do
78 !$omp end parallel
79
80 T_END = omp_get_wtime()
81
82 print *, ' Number of primes found:',number_of_primes
83 print *, ' Number of 4n+1 primes found:',number_of_41primes
84 print *, ' Number of 4n-1 primes found:',number_of_43primes
85 print *, 'end parallel', T_END - T_IN
86 end program ompPrime
```

ANEXO B Programa en Fortran que calcula la aproximación numérica con OpenMP para la regla del trapecio con $f(x) = x^2$

```
!OpenMP code. Integration method (Trapezoidal rule)
!Author: Carlos Acosta
!Date: 05/22/09
    PROGRAM paralleltrap
      use omp_lib !OpenMP library
      real ::  integral
      real  :: a
      real  :: b
      integer  :: n, m
      real  :: h
      real  :: x
      integer :: i, nthread
      integer, save :: id
      real :: f
   a = 0.0    !initial range value
  b = 1.0    !Final range value
  n = 100   !Number of elements
      h = (b-a)/n !Area of each element
        integral = (f(a) + f(b))/2.0
      x = a
      m = n-1
        !************OpenMP application**********
      call omp_set_num_threads(2)
      !$omp parallel default(shared) private(i)
      !$omp do
       do i = 1, m
         IF (i == 1) THEN
         nthread = omp_get_num_threads() !Find Number of active threads
         PRINT *, "# Threads", nthread
         END IF
            x = x + h
          integral = integral + f(x)
      enddo
    !$omp end do
    !$omp end parallel
    !***************End OpenMP Parallel Region**********
    integral =  integral*h
      print *,'With =', n,' Trapezoids the solution is:', integral
      print *,'Within the range ', a,b
      end

!**********************Funtion**********************
      real function f(x)
      IMPLICIT NONE
      real x
!  Calculate f(x).
f = x*x
      return
      end
!******************End Function********************
```

ANEXO C Programa en Fortran que calcula la aproximación numérica con MPI para la regla del trapecio con $f(x) = x^2$

```
1  !Parallel_Trapezoid.f90 -- Parallel Trapezoidal Rule, first version
2  !  Input: None.
3  !  Output:  Estimate of the integral from a to b of f(x)
4  !     using the trapezoidal rule and n trapezoids.
5  !  Algorithm:
6  !     1.  Each process calculates "its" interval of integration.
7  !     2.  Each process estimates the integral of f(x)
8  !         over its interval using the trapezoidal rule.
9  !     3a. Each process != 0 sends its integral to 0.
10 !     3b. Process 0 sums the calculations received from
11 !         the individual processes and prints the result.
12 ! Modification: Program changed from original form to Fortran90
13 !      Additionally, the distribution of task among
14 !           processors changed.
15 ! Date: 05/09 ! Modification By: Carlos Andrés Acosta Berlinghieri
16   PROGRAM Parallel_Trapezoid
17   IMPLICIT NONE
18   include 'mpif.h'
19   integer   my_rank  ! My process rank.
20   integer   p        ! The number of processes.
21   real      a        ! Left endpoint.
22   real      b        ! Right endpoint.
23   integer   n        ! Number of trapezoids.
24   real      h        ! Trapezoid base length.
25   real      local_a  ! Left end point for my process.
26   real      local_b  ! Right endpoint my process.
27   integer   local_n  ! Number of trapezoids for my calculation.
28   real      integral ! Integral over my interval.
29   real      total    ! Total integral.
30   integer   source   ! Process sending integral.
31   integer   dest     ! ID to send messages.
32   integer   tag
33   integer   status(MPI_STATUS_SIZE)
34   integer   iproc, interval
35   integer   ierr     ! MPI error indicator
36   integer   num_local
37   real      Trap
38   data a, b, n, dest, tag /0.0, 1.0, 1000, 0, 50/ !Range,
39   !elements and destination to processors ID.
40   integer, allocatable :: ibegin(:), iend(:)
41   !  Let the system do what it needs to start up MPI.
42   CALL MPI_INIT(ierr)   !Check MPI initialization errors
43 !  Get my process rank.
44 CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)
45 !  Find out how many processes are being used.
46 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, p, ierr)
47 h = (b-a)/n   ! h is the same for all processes.
48 local_n = n/p      ! So is the number of trapezoids.
49 IF(my_rank == 0) THEN
50 interval = n/p
51 allocate(ibegin(p)) !Distribution of elements among processors
52 allocate(iend(p))
```

```
53
54 DO iproc = 1, p
55     ibegin(iproc)   = (iproc-1)*interval + 1
56     iend  (iproc)   = ibegin(iproc) + interval - 1
57 END DO
58 write(*,'(a,i8)') '  ==> Elements= ', n
59 write(*,*) ' ==> This is the distribution'
60
61 DO iproc=1, p
62     write(*,'(a,i4,a,i5,1x,i5)') '  ==> Task ',iproc,':
63     Range= ',ibegin(iproc),iend(iproc)
64 END DO
65 !  Length of each process' interval of integration =local_n*h. So my interval
66 ! starts at:
67 local_a = a + my_rank*local_n*h
68 local_b = local_a + local_n*h
69 integral = Trap(local_a, local_b, local_n, h)
70 !  Add up the integals calculated by each process.
71 IF (my_rank .EQ. 0) THEN
72 total = integral
73 DO source = 1, p-1
74      CALL MPI_RECV(integral, 1, MPI_REAL, source, tag,
75      MPI_COMM_WORLD, status, ierr)
76      total = total + integral
77 END DO
78 CALL MPI_SEND(integral, 1, MPI_REAL, dest, tag, MPI_COMM_WORLD,ierr)
79 ENDIF
80 ¡Print the result.
81 IF (my_rank .EQ. 0) THEN
82     write(6,200) n
83     format(' ','With n = ',I4,' trapezoids, our estimate')
84     write(6,300) a, b, total
85     format(' ','of the integral from ',f6.2,' to ',f6.2,' = 97 ',f11.5)
86 ENDIF
87 !       Shut down MPI.
88 CALL MPI_FINALIZE(ierr)
89 END PROGRAM Parallel_Trapezoid
90  !Function Trap
91  real function Trap(local_a, local_b, local_n, h)
92 IMPLICIT NONE
93 real     local_a, local_b
94 integer  local_n
95 real     integral    ! Store result in integral.
96 real     x, i, f, h
97 integral = (f(local_a) + f(local_b))/2.0
98 x = local_a
99 DO i = 1, local_n - 1
100      x = x + h
101      integral = integral + f(x)
102 ENDDO
103 integral = integral*h
104 Trap = integral
105 Return
106 End
107 real function f(x)
108 IMPLICIT NONE
109 real x
110 f = x*x
```

ANEXO D Programa en C++ para el método de eliminación Gaussiana en paralelo

```
1   //Backward sustitution
2   #pragma omp parallel private(i, j, factor)
3   #pragma omp for
4       for (i = 0; i < n; i++)
5       {
6            if (i == 0 )
7           {
8               nth = omp_get_num_threads();
9               nth = omp_get_num_procs();
10              fprintf(stdout, "%d \n",nth);
11          }
12
13          for (j = i + 1; j < n;j++){
14              factor = A[j][i]/A[i][i];
15
16            for (k = i; k < n; k++ ){
17                  A[j][k] = A[j][k] - factor * A[i][k];
18              }
19              b[j] = b[j] - factor * b[i];
20                   nth = omp_get_num_threads();
21                   fprintf(stdout, "%d \n",nth);
22          }
23
24      }
25
26  //backward sustitution
27      X[n-1]=b[n-1]/A[n-1][n-1];
28  #pragma omp parallel private(j, k)
29  #pragma omp parallel for IF j > n-2
30      for (j=n-2;j>-1;j--){
31          temp=0;
32          for (k=j+1;k<n;k++)
33          {
34               temp=temp+A[j][k]*X[k];
35          }
36          X[j]=(b[j]-temp)/A[j][j];
37      }
```

ANEXO E . Programa en C++ que calcula la solución de la ecuación de Laplace para flujo potencial en paralelo

```
1  #include <iostream>
2  #include <stdlib.h>
3  #include <fstream>
4  #include <string>
5  #include <math.h>
6  #include <vector.h>
7  #include <sstream>
8  using namespace std;
9  /*****************************************************************
10 *                                                               *
11 *              FUNCIONES PROGRAMADAS                            *
12 *                                                               *
13 *****************************************************************/
14 void writevtk (vector < double > U, vector < vector < double > > Nodos, vector <
     vector < int > > Elementos, vector < vector < double > > Vect_camp);
15 vector < vector < double > > leer_coordenates (string Matrix);
16
17 vector < vector < int > > leer_trianles (string Matrix);
18
19 vector < int > leer_dirichlet (string Matrix , vector < double > & g);
20
21 vector < vector < double > > Calculate_A (vector < vector < double > > cor,
     vector < vector < int > > tri);
22
23 vector < double > Rest_Vector (vector < double > X, vector < double > Y);
24
25 vector < double > Mult_Matrix_Vector (vector < vector < double > > Matrix, vector
     < double > Vector);
26
27 vector < double > leer_VonNewman (string Matrix, vector < vector < double > >
     cor);
28
29 vector < double > gauss (vector < vector < double > > A, vector < double > b);
30
31 vector < double > make_U (vector < double > g, vector < double > U_dr, vector <
     int > pos);
32
33 vector < vector < double > > Calculate_velocity_XY (vector < vector < double > >
     cor, vector < vector < int > > tri, vector < double > U);
34
35 vector < vector < double > > dirich_Matrix(vector < vector < double > > Global_M,
     vector < int > Pos);
36
37 vector < double >  dirich_Vector( vector < double > Global_V, vector < int >
     Pos);
38
39 /*****************************************************************
40 *                                                               *
41 *                    PROGRAMA PRINCIPAL                         *
42 *                                                               *
43 *****************************************************************/
```

```
44 main(){
45     int size = 0;
46     int nd, i, j;
47     cout << endl;
48     cout << "CORRIENDO ELEMENTOS FINITOS 2-D" << endl;
49     cout << endl;
50     //  Lectura del archivo de coordenadas de los nodos
51     vector < vector < double > > cor;
52     string cor_file;
53     cout << "Ingrese el archivo de datos de coordenadas a leer:" << endl;
54     cin >> cor_file;
55     cor = leer_coordenates (cor_file);
56     size = cor.size();
57     //  Lectura del archivo de los triangulos
58     vector < vector < int > > tri;
59     cout << "Ingrese el archivo de datos de trinagulos a leer:" <<endl;
60     string tri_file;
61     cin >> tri_file;
62    tri = leer_trianles (tri_file);
63    //  Lectura del archivo con las condiciones de Dirichlet y definicion del
    vector g
64     vector < double > g(size,0.0);
65     vector < int > Pos;
66     string dr_file;
67     cout << "Ingrese el archivo de condiciones de dirichlet a leer:" << endl;
68     cin >> dr_file;
69     Pos = leer_dirichlet ( dr_file , g);
70     //  Lectrura vectror de condiciones de VonNewman y escritura del vector q
71     vector < double > q;
72     cout << "Ingrese el archivo de datos de Von Newman a leer:" <<endl;
73     string vn_file;
74     cin >> vn_file;
75     q = leer_VonNewman (vn_file, cor);
76    //  Calculo de la matriz A global
77     vector < vector < double > > A;
78     A = Calculate_A ( cor, tri);
79     //  Multiplicacion A*g
80     vector < double > AG;
81     AG = Mult_Matrix_Vector ( A, g);
82     //  Calculo del vector Ag de dirichlet
83     vector < double > AG_dir;
84     AG_dir = dirich_Vector( AG, Pos);
85     nd = AG_dir.size();
86     //  Calculo del vector q de dirichlet
87     vector < double > q_dir;
88     q_dir = dirich_Vector( q, Pos);
89     //  Calculo matriz A de dirichlet
90     vector < vector < double > > A_dir;
91     A_dir = dirich_Matrix(A, Pos);
92     //  Resta de vectores
93     vector < double > b;
94     b = Rest_Vector (q_dir, AG_dir);
95     //  Solucion del Sistema
96     vector < double > U_dr;
97     U_dr = gauss(A_dir, b);
98     //  Reconstruccion del vector U
99     vector < double > U;
100    U = make_U (g, U_dr, Pos);
```

```
101     // calculo de vectores
102     vector < vector < double > > Vel;
103     Vel = Calculate_velocity_XY (cor, tri, U);
104     //  Escritura formato vtk
105     writevtk( U, cor, tri, Vel );
106     cout << "FIN DEL PROGRAMA, UTILICE ParaView PARA VER LOS RESULTADOS" << endl;
107    cout << endl;
108    return 0;
109 }
110 /*****************************************************************
111 *     FUNCION PARA ESCRIBIR LOS RESULTADOS EN FORMATO VTK        *
112 *            PARA SER VISUALIZADOS CON ParaView                  *
113 *                                                                *
114 *****************************************************************/
115 void writevtk (vector < double > U, vector < vector < double > > Nodos, vector <
      vector < int > > Elementos, vector < vector < double > > Vect_camp){
116     cout << "Escribiendo .vtk" << endl;
117     cout << endl;
118     int nNodes, nEle, k;
119     nNodes = Nodos.size();
120     nEle = Elementos.size();
121     //  Encabezado del archivo
122     ofstream myfile;
123     myfile.open ("Result.vtk");
124     myfile << "# vtk DataFile Version 3.1" << endl;
125     myfile << "this is the result file from the program" << endl;
126     myfile << endl;
127     //  Escritura de los nodos
128     myfile << "ASCII" << endl;
129     myfile << "DATASET POLYDATA" << endl;
130     myfile << "POINTS " << nNodes << " DOUBLE" << endl;
131     myfile << endl;
132    for( k = 0; k < nNodes; k++)
133    {
134       myfile << Nodos[k][0] << " " << Nodos[k][1] <<  " " << 0 << endl;
135    }
136     //  Escritura de los triangulos
137    myfile << endl;
138    myfile << "TRIANGLE_STRIPS " << nEle << " " << nEle*4 << endl;
139    for( k = 0; k < nEle; k++)
140    {
141       myfile << "3 " << Elementos[k][0]-1 << " " << Elementos[k][1]-1 <<  " " <<
      Elementos[k][2]-1 << endl;
142    }
143     // Escritura del campo escalar
144    myfile << endl;
145    myfile << "POINT_DATA " << nNodes << endl;
146    myfile << "SCALARS u double" << endl;
147    myfile << "LOOKUP_TABLE default" << endl;
148    for( k = 0; k < nNodes; k++)
149    {
150       myfile << U[k] << endl;
151    }
152    //  Esctitura del campo vectorial
153    myfile << endl;
154    myfile << "VECTORS ";
155    myfile << "Velocidades ";
156    myfile << "float" << endl;
```

```
157    for( k = 0; k < nNodes; k++)
158    {
159       myfile << Vect_camp[k][0] << " " << Vect_camp[k][1] << " " << 0 << endl;
160    }
161 return;
162 }
163 /*********************************************************************
164 *                                                                    *
165 * FUNCION PARA LEEER EL ARCHIVO CON INFORMACION DE LOS TRIANGULOS *
166 *                 Y ALMACENARLOS EN UNA MATRIZ                       *
167 *                                                                    *
168 *********************************************************************/
169 vector < vector < int > > leer_trianles (string Matrix){
170     cout << "Lellendo triangulos" << endl;
171     cout << endl;
172     ifstream matrixFile1;
173     string  cmpStr;
174     int i;
175     int n;
176     matrixFile1.open(Matrix.c_str());
177     if ( ! matrixFile1.is_open()) {
178          cout << "Error abriendo el archivo \n" ;
179          exit (-1);
180     }
181     matrixFile1 >> n;
182     vector < vector < int > > tri(n, vector < int >(3, 0));
183 #pragma omp parallel private(i)
184 #pragma omp for
185     for (i = 0; i < n; i++){
186         matrixFile1 >> tri[i][0];
187         matrixFile1 >> tri[i][1];
188         matrixFile1 >> tri[i][2];
189     }
190      return tri;
191 }
192 /*********************************************************************
193 *                                                                    *
194 * FUNCION PARA LEEER EL ARCHIVO CON INFORMACION DE LAS COORDENADAS *
195 *              DE CADA NODO Y ALMACENARLOS EN UNA MATRIZ             *
196 *                                                                    *
197 *********************************************************************/
198 vector < vector < double > > leer_coordenates (string Matrix){
199     cout << "Lellendo coordenadas" << endl;
200     cout << endl;
201     ifstream matrixFile1;
202     string  cmpStr;
203     int i;
204     int n;
205     matrixFile1.open(Matrix.c_str());
206     if ( ! matrixFile1.is_open()) {
207          cout << "Error abriendo el archivo \n" ;
208          exit (-1);
209     }
210     matrixFile1 >> n;
211     vector < vector < double > > cor(n, vector < double >(2, 0));
212     for (i = 0; i < n; i++){
213         matrixFile1 >> cor[i][0];
214         matrixFile1 >> cor[i][1];
```

```
215     }
216      return cor;
217 }
218 /*********************************************************************
219 *                                                                    *
220 * FUNCION PARA LEEER EL ARCHIVO CON INFORMACION DE LAS COORDENADAS *
221 *             DE CADA NODO Y ALMACENARLOS EN UNA MATRIZ            *
222 *                                                                    *
223 *********************************************************************/
224  vector < int > leer_dirichlet (string Matrix , vector < double > & g){
225     cout << "Lellendo dirichlet" << endl;
226     cout << endl;
227     ifstream matrixFile1;
228     string  cmpStr;
229     int i,k;
230     matrixFile1.open(Matrix.c_str());
231     if ( ! matrixFile1.is_open()) {
232          cout << "Error abriendo el archivo \n" ;
233          exit (-1);
234     }
235     matrixFile1 >> k;
236     vector < int > Pos(k , 0);
237     for (i = 0; i < k; i++){
238     matrixFile1 >> Pos[i];              // Vector con las posiciones de dirichlet
239     matrixFile1 >> g[(Pos[i]-1)];       // Vector con los valores de u para las
      posiciones de Dirichlet "g"
240     }
241      return Pos;
242 }
243 /*********************************************************************
244 *                                                                    *
245 *            FUNCION PARA CALCULAR LA MATRIZ "A" GLOBAL              *
246 *                                                                    *
247 *********************************************************************/
248 vector < vector < double > > Calculate_A (vector < vector < double > > cor,
      vector < vector < int > > tri){
249     cout << "Calculando matriz A" << endl;
250     cout << endl;
251     int n,i,NA,NB,NC,j,nod;
252     double det = 0.0;
253     double Area = 0.0;
254     vector < double > A( 2 , 0.0 );
255     vector < double > B( 2 , 0.0 );
256     vector < double > C( 2 , 0.0 );
257     vector < double > AB( 2 , 0.0 );
258    vector < double > BC( 2 , 0.0 );
259     vector < double > CA( 2 , 0.0 );
260     vector < vector < double > > A_local(3, vector < double >(3, 0.0));
261     n = tri.size();
262     nod = cor.size();
263     vector < vector < double > > A_Global(nod, vector < double >(nod, 0.0));
264     //  Calculo para cada elemento
265     for (i = 0; i<n; i++){
266
267         //  Determinacion de los nodos y sus coordenadas que conforman cada
      elemento
268        NA = tri[i][0] - 1;
269         NB = tri[i][1] - 1;
```

```
270         NC = tri[i][2] - 1;
271         A[0] = cor[NA][0];
272         A[1] = cor[NA][1];
273         B[0] = cor[NB][0];
274         B[1] = cor[NB][1];
275         C[0] = cor[NC][0];
276         C[1] = cor[NC][1];
277         //  determinante para hallar el area
278         det = (A[0]*(B[1]-C[1])) - (A[1]*(B[0]-C[0])) + (B[0]*C[1]) -
      (B[1]*C[0]);
279         if (det < 0){
280             det = det*(-1);
281        }
282         Area = (0.5)*(det);
283        // Calculo de la matriz "A local"
284         AB = Rest_Vector (B,A);
285         BC = Rest_Vector (C,B);
286         CA = Rest_Vector (A,C);
287         A_local[0][0] = (BC[0] * BC[0]) + (BC[1] * BC[1]);
288         A_local[0][1] = (BC[0] * CA[0]) + (BC[1] * CA[1]);
289         A_local[0][2] = (BC[0] * AB[0]) + (BC[1] * AB[1]);
290         A_local[1][0] = (CA[0] * BC[0]) + (CA[1] * BC[1]);
291         A_local[1][1] = (CA[0] * CA[0]) + (CA[1] * CA[1]);
292         A_local[1][2] = (CA[0] * AB[0]) + (CA[1] * AB[1]);
293         A_local[2][0] = (AB[0] * BC[0]) + (AB[1] * BC[1]);
294         A_local[2][1] = (AB[0] * CA[0]) + (AB[1] * CA[1]);
295         A_local[2][2] = (AB[0] * AB[0]) + (AB[1] * AB[1]);
296         //Actualizacion de la matriz "A global"
297         A_Global[NA][NA] = A_Global[NA][NA] + ((1/(Area * 4)) * A_local[0][0]);
298         A_Global[NA][NB] = A_Global[NA][NB] + ((1/(Area * 4)) * A_local[0][1]);
299         A_Global[NA][NC] = A_Global[NA][NC] + ((1/(Area * 4)) * A_local[0][2]);
300         A_Global[NB][NA] = A_Global[NB][NA] + ((1/(Area * 4)) * A_local[1][0]);
301         A_Global[NB][NB] = A_Global[NB][NB] + ((1/(Area * 4)) * A_local[1][1]);
302         A_Global[NB][NC] = A_Global[NB][NC] + ((1/(Area * 4)) * A_local[1][2]);
303         A_Global[NC][NA] = A_Global[NC][NA] + ((1/(Area * 4)) * A_local[2][0]);
304         A_Global[NC][NB] = A_Global[NC][NB] + ((1/(Area * 4)) * A_local[2][1]);
305         A_Global[NC][NC] = A_Global[NC][NC] + ((1/(Area * 4)) * A_local[2][2]);
306      }
307 return A_Global;
308 }
309 /*****************************************************************
310 *                                                               *
311 *                   FUNCION PARA RESTAR VECTORS                 *
312 *                          z = x - y                            *
313 *                                                               *
314 *****************************************************************/
315 vector < double > Rest_Vector (vector < double > X, vector < double > Y){
316     int n,i;
317     n = X.size();
318     vector < double > Z(n , 0.0);
319     for (i=0;i < n;i++){
320         Z[i] = X[i] - Y[i];
321    }
322 return Z;
323 }
324 /*****************************************************************
325 *                                                               *
326 *            FUNCION PARA MULTIPLICAR MATRIZ POR VECTOR         *
```

```
327 *                     Result = Matrix * Vector                     *
328 *                                                                  *
329 ********************************************************************/
330 vector < double > Mult_Matrix_Vector (vector < vector < double > > Matrix, vector
      < double > Vector){
331     int f,c,i,j;
332     f = Matrix.size();
333     c = Vector.size();
334     vector < double > Result(c,0.0);
335     for (i=0;i < f;i++){
336         for (j=0;j < c;j++){
337             Result[i] = Result[i] + Matrix[i][j] * Vector[j];
338         }
339     }
340 return Result;
341 }
342 /********************************************************************
343 *                                                                  *
344 *  FUNCION PARA LEER EL VECTOR CON LAS CONDICIONES DE Von Newman   *
345 *                     Y CREACION DEL VECTOR "q"                    *
346 *                                                                  *
347 ********************************************************************/
348 vector < double > leer_VonNewman (string Matrix, vector < vector < double > >
      cor){
349     cout << "Lellendo condicioned de Von Newman" << endl;
350     cout << endl;
351     ifstream matrixFile;
352     string  cmpStr;
353     int size, i, n, na, nb;
354     double qa, qb, h;
355     matrixFile.open(Matrix.c_str());
356     size = cor.size();
357     if ( ! matrixFile.is_open()) {
358          cout << "Error abriendo el archivo \n" ;
359          exit (-1);
360     }
361     matrixFile >> n;
362     vector < double > q(size, 0.0);
363     vector < double > A(2, 0.0);
364     vector < double > B(2, 0.0);
365     #pragma omp parallel private(i)
366     #pragma omp for
367     for (i=0; i<n; i++){
368         //  Lectura de los datos del archivo
369         matrixFile >> na;
370         matrixFile >> nb;
371         matrixFile >> qa;
372         matrixFile >> qb;
373         //  Calculo de la longuitud de la frontera "h"
374         A[0] = cor[na-1][0];
375         A[1] = cor[na-1][1];
376         B[0] = cor[nb-1][0];
377         B[1] = cor[nb-1][1];
378         h = sqrt(((A[0] - B[0]) * (A[0] - B[0])) + ((A[1] - B[1]) * (A[1] -
      B[1])));
379         //  Calculo de valor de q para nodo y actualizacion del vector "q"
380         q[na-1] = q[na-1] + (((h / 3) * qa) + (qb * (h / 6)));
381         q[nb-1] = q[nb-1] + (((h / 3) * qb) + (qa * (h / 6)));
```

```
382     }
383     return q;
384 }
385 /*********************************************************************
386 *                                                                    *
387 *       METODO DE GAUSS PARA SOLUCION DE SISTEMAS LINEALES           *
388 *                           A * X = b                                *
389 *                                                                    *
390 *********************************************************************/
391 vector < double > gauss (vector < vector < double > > A, vector < double > b){
392     cout << "Resolviendo el sistema de ecuaciones lineales" << endl;
393     cout << endl;
394     int i, j, k, n,fm,cm;
395     double temp, factor;
396     n = b.size();
397     vector < double > X(n, 0.0);
398 //Forward sustitution
399 #pragma omp parallel private(i, j, factor)
400 #pragma omp for
401     for (i = 0; i < n; i++)
402     {
403          if (i == 0 )
404         {
405             nth = omp_get_num_threads();
406             nth = omp_get_num_procs();
407             fprintf(stdout, "%d \n",nth);
408         }
409         for (j = i + 1; j < n;j++){
410             factor = A[j][i]/A[i][i];
411           for (k = i; k < n; k++ ){
412                A[j][k] = A[j][k] - factor * A[i][k];
413             }
414             b[j] = b[j] - factor * b[i];
415                  nth = omp_get_num_threads();
416                  fprintf(stdout, "%d \n",nth);
417         }
418     }
419 //backward sustitution
420     X[n-1]=b[n-1]/A[n-1][n-1];
421 #pragma omp parallel private(j, k)
422 #pragma omp parallel for IF j > n-2
423     for (j=n-2;j>-1;j--){
424         temp=0;
425         for (k=j+1;k<n;k++)
426         {
427              temp=temp+A[j][k]*X[k];
428         }
429         X[j]=(b[j]-temp)/A[j][j];
430     }
431     return X;
432 }
433 /*********************************************************************
434 *                                                                    *
435 *       FUNCION PARA UNIR LOS VECTORES A U DE DIRICHLET Y G          *
436 *                                                                    *
437 *********************************************************************/
438 vector < double > make_U (vector < double > g, vector < double > U_dr, vector <
      int > pos){
```

```
439     int i, n, m, c = 0, ce=0;
440     n = g.size();
441     m = pos.size();
442     vector < double > v(n, 0.0);
443     for (i = 0; i < n; i++){
444         if ( (pos[c] - 1) == i){
445             v[i] = g[i];
446             c += 1;
447         }
448         else if ((pos[c] - 1) != i)
449         {
450             v[i] = U_dr[ce];
451             ce += 1;
452         }
453     }
454 return v;
455 }
456 /*********************************************************************
457 *                                                                    *
458 *          FUNCION PARA CALCULAR EL CAMPO DE VELOCIDADES             *
459 *                                                                    *
460 *********************************************************************/
461 vector < vector < double > > Calculate_velocity_XY (vector < vector < double > >
      cor, vector < vector < int > > tri, vector < double > U){
462     cout << "Calculando campo de velocidades" << endl;
463     cout << endl;
464     int i, j, NA, NB, NC, n = 0, m = 0;
465     double a1, b1, c1, a2, b2, c2, a3, b3, c3, valx, valy;
466     //double xa, ya, xb, yb, xc, yc;
467     n = cor.size();
468     m = tri.size();
469     vector < vector < double > > Cor_mat_XY(3, vector < double > (2, 0));
470     vector < vector < double > >  vel_vec(n, vector < double >(2, 0));
471     vector < int > prom (n,0);
472     for (i = 0; i < m; i++){
473         NA = tri[i][0]-1;
474         NB = tri[i][1]-1;
475         NC = tri[i][2]-1;
476         Cor_mat_XY[0][0] = cor[NA][0];
477         Cor_mat_XY[0][1] = cor[NA][1];
478         Cor_mat_XY[1][0] = cor[NB][0];
479         Cor_mat_XY[1][1] = cor[NB][1];
480         Cor_mat_XY[2][0] = cor[NC][0];
481         Cor_mat_XY[2][1] = cor[NC][1];
482         /*
483         1 = a1*x1 + b1*y1 + c1
484         0 = a1*x2 + b1*y2 + c1
485         0 = a1*x3 + b1*y3 + c1
486         */
487         a1=((Cor_mat_XY[1][1]-
      Cor_mat_XY[2][1])/((Cor_mat_XY[0][0]*(Cor_mat_XY[1][1]-Cor_mat_XY[2][1]))-
      (Cor_mat_XY[1][0]*(Cor_mat_XY[0][1]-
      Cor_mat_XY[2][1]))+(Cor_mat_XY[2][0]*(Cor_mat_XY[0][1]-Cor_mat_XY[1][1]))));
488         b1 = ((-(Cor_mat_XY[1][0]-
      Cor_mat_XY[2][0]))/((Cor_mat_XY[0][0]*(Cor_mat_XY[1][1]-Cor_mat_XY[2][1])) -
      (Cor_mat_XY[1][0]*(Cor_mat_XY[0][1]-Cor_mat_XY[2][1])) +
      (Cor_mat_XY[2][0]*(Cor_mat_XY[0][1]-Cor_mat_XY[1][1]))));
```

```
489         c1 = ((Cor_mat_XY[1][0]*(Cor_mat_XY[2][1])-
      (Cor_mat_XY[2][0]*(Cor_mat_XY[1][1])))/((Cor_mat_XY[0][0]*(Cor_mat_XY[1][1]-
      Cor_mat_XY[2][1])) - (Cor_mat_XY[1][0]*(Cor_mat_XY[0][1]-Cor_mat_XY[2][1])) +
      (Cor_mat_XY[2][0]*(Cor_mat_XY[0][1]-Cor_mat_XY[1][1]))));
490         /*
491         0 = a2*x1 + b2*y1 + c2
492         1 = a2*x2 + b2*y2 + c2
493         0 = a2*x3 + b2*y3 + c2
494         */
495         a2=((-(Cor_mat_XY[0][1]-
      Cor_mat_XY[2][1]))/((Cor_mat_XY[0][0]*(Cor_mat_XY[1][1]-Cor_mat_XY[2][1]))-
      (Cor_mat_XY[1][0]*(Cor_mat_XY[0][1]-
      Cor_mat_XY[2][1]))+(Cor_mat_XY[2][0]*(Cor_mat_XY[0][1]-Cor_mat_XY[1][1]))));
496         b2=((Cor_mat_XY[0][0]-
      Cor_mat_XY[2][0])/((Cor_mat_XY[0][0]*(Cor_mat_XY[1][1]-Cor_mat_XY[2][1]))-
      (Cor_mat_XY[1][0]*(Cor_mat_XY[0][1]-
      Cor_mat_XY[2][1]))+(Cor_mat_XY[2][0]*(Cor_mat_XY[0][1]-Cor_mat_XY[1][1]))));
497         c2=((-((Cor_mat_XY[0][0]*(Cor_mat_XY[2][1])-
      (Cor_mat_XY[2][0]*(Cor_mat_XY[0][1])))))/((Cor_mat_XY[0][0]*(Cor_mat_XY[1][1]-
      Cor_mat_XY[2][1]))-(Cor_mat_XY[1][0]*(Cor_mat_XY[0][1]-
      Cor_mat_XY[2][1]))+(Cor_mat_XY[2][0]*(Cor_mat_XY[0][1]-Cor_mat_XY[1][1]))));
498         /*
499         0 = a3*x1 + b3*y1 + c3
500         0 = a3*x2 + b3*y2 + c3
501         1 = a3*x3 + b3*y3 + c3
502         */
503         a3=((Cor_mat_XY[0][1]-
      Cor_mat_XY[1][1])/((Cor_mat_XY[0][0]*(Cor_mat_XY[1][1]-Cor_mat_XY[2][1]))-
      (Cor_mat_XY[1][0]*(Cor_mat_XY[0][1]-
      Cor_mat_XY[2][1]))+(Cor_mat_XY[2][0]*(Cor_mat_XY[0][1]-Cor_mat_XY[1][1]))));
504         b3=((-(Cor_mat_XY[0][0]-
      Cor_mat_XY[1][0]))/((Cor_mat_XY[0][0]*(Cor_mat_XY[1][1]-Cor_mat_XY[2][1]))-
      (Cor_mat_XY[1][0]*(Cor_mat_XY[0][1]-
      Cor_mat_XY[2][1]))+(Cor_mat_XY[2][0]*(Cor_mat_XY[0][1]-Cor_mat_XY[1][1]))));
505         c3=((Cor_mat_XY[0][0]*(Cor_mat_XY[1][1])-
      (Cor_mat_XY[1][0]*(Cor_mat_XY[0][1])))/((Cor_mat_XY[0][0]*(Cor_mat_XY[1][1]-
      Cor_mat_XY[2][1]))-(Cor_mat_XY[1][0]*(Cor_mat_XY[0][1]-
      Cor_mat_XY[2][1]))+(Cor_mat_XY[2][0]*(Cor_mat_XY[0][1]-Cor_mat_XY[1][1]))));
506         valx = (a1 * U[NA]) + (a2 * U[NB]) + (a3 * U[NC]) ;
507         valy = (b1 * U[NA]) + (b2 * U[NB]) + (b3 * U[NC]);
508         vel_vec[NA][0] = vel_vec[NA][0] + valx;
509         vel_vec[NA][1] = vel_vec[NA][1] + valy;
510         vel_vec[NB][0] = vel_vec[NB][0] + valx;
511         vel_vec[NB][1] = vel_vec[NB][1] + valy;
512         vel_vec[NC][0] = vel_vec[NC][0] + valx;
513         vel_vec[NC][1] = vel_vec[NC][1] + valy;
514         prom[NA] = prom[NA] + 1;
515         prom[NB] = prom[NB] + 1;
516         prom[NC] = prom[NC] + 1;
517     }
518     for (i = 0; i < n; i++){
519         vel_vec[i][0] = vel_vec[i][0] / prom[i];
520         vel_vec[i][1] = vel_vec[i][1] / prom[i];
521     }
522     return vel_vec;
523 }
524 /*****************************************************************
525 *                                                               *
```

```
526 *          FUNCION PARA CALCULAR LA MATRIZ DE DIRICHLET            *
527 *                                                                  *
528 ******************************************************************/
529 vector < vector < double > > dirich_Matrix(vector < vector < double > > Global_M,
      vector < int > Pos){
530     int b = 0;
531     int c = 0;
532     int d = 0;
533     int k = 0;
534     int l = 0;
535     int m = 0;
536     int n = 0;
537     int i = 0;
538     int j = 0;
539     int co = 0;
540     int c_1 = 0;
541     n = Global_M.size();
542     l = Pos.size();
543     vector < int > G(n , 0);
544     for (i = 0; i < l ; i++){
545         m = Pos[i];
546         G[m -1] = m;
547     }
548     // Eliminacion de las filas de Dirichlet
549     vector < vector < double > > M(n, vector < double >(n, 0));
550    #pragma omp parallel private(i)
551    #pragma omp for
552     for (i = 0; i < n ; i++){
553         m = G[i];
554         if (m != (i + 1)){
555             for (k = 0; k < n ; k++){
556                 M[k][co] = Global_M[k][i];
557                 if (k == n - 1){
558                     co += 1;
559                 }
560             }
561         }
562     }
563     b = M.size();
564     c = M[1].size();
565     d = b - l;
566     vector < vector < double > > Mat_dir(d, vector < double >(c, 0));
567     // Eliminacion de las columnas de Dirichlet
568     #pragma omp parallel private(i)
569     #pragma omp for
570     for (i = 0; i < n ; i++){
571         m = G[i];
572         if (m != i + 1){
573             for (j = 0; j < c ; j++){
574                 Mat_dir[c_1][j] = M[i][j];
575                 if (j == c - 1){
576                     c_1 += 1;
577                 }
578             }
579         }
580     }
581     return Mat_dir;
582 /******************************************************************
```

```
583 *                                                                    *
584 *          FUNCION PARA CALCULAR EL VECTOR DE DIRICHLET              *
585 **********************************************************************/
586 vector < double >  dirich_Vector( vector < double > Global_V, vector < int >
      Pos){
587     int n, m, i, l;
588     int c_1 = 0;
589     n = Global_V.size();
590     l = Pos.size();
591     vector < int > G(n , 0);
592     for (i = 0; i < l ; i++){
593         m = Pos[i];
594         G[m -1] = m;
595     }
596     vector < double > Vec_dir((n-l) , 0.0);
597     for (i = 0; i < n ; i++){
598         m = G[i];
599         if (m != i + 1){
600                 Vec_dir[c_1] = Global_V[i];
601                 c_1 += 1;
602         }
603     }
604     return Vec_dir;
```