EVALUACIÓN DEL ALGORITMO DE MIGRACIÓN DE KIRCHHOFF EN 2D EN UN CLÚSTER DE PROCESADORES

GERARDO ENRIQUE TERAN JULIO
JULIAN RODRIGO PITA ESCAMILLA

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERIAS FÍSICO MECÁNICAS
ESCUELA DE INGENIERIAS ELÉCTRICA, ELECTRÓNICA Y DE TELECOMUNICACIONES
BUCARAMANGA
2009

EVALUACIÓN DEL ALGORITMO DE MIGRACIÓN DE KIRCHHOFF EN 2D EN UN CLÚSTER DE PROCESADORES

GERARDO ENRIQUE TERAN JULIO
JULIAN RODRIGO PITA ESCAMILLA

Este proyecto se presenta como requisito para optar al titulo de Ingeniero Electrónico

Director
M.S.E.E. ANA BEATRIZ RAMÍREZ

Codirector
ING. WILLIAM SALAMANCA
ING. SERGIO ABREO

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FÍSICO MECÁNICAS
ESCUELA DE INGENIERÍAS ELÉCTRICA, ELECTRÓNICA Y DE TELECOMUNICACIONES
BUCARAMANGA
2009

AGRADECIMIENTOS

Agradecemos a nuestras familias por estar siempre apoyándonos en cada momento de nuestras vidas.

A nuestra directora Ana Beatriz Ramírez por su respaldo y a quien agradecemos la confianza que nos brindó en la ejecución de este trabajo.

A los Ingenieros Electrónicos William Salamanca y Sergio Abreo por el respaldo como codirectores de este proyecto y por su gran colaboración al momento de resolver inquietudes.

A nuestros amigos que han estado a nuestro lado apoyándonos durante esta etapa.

A la escuela de Ingeniería Eléctrica, Electrónica y de Telecomunicaciones que hizo posible nuestra formación como estudiantes de ingeniería electrónica. También a todos los profesores y personal administrativo.

Al CENTIC por haber puesto a nuestra disposición los equipos con los que no s fue posible culminar nuestro proyecto.

TABLA DE CONTENIDO

Pág.

LISTA DE FIGURAS

Pág.

LISTA DE TABLAS

Pág.

LISTA DE ANEXOS

RESUMEN

TÍTULO: Implementación Y Evaluación Del Algoritmo De Migración De Kirchhoff En 2d En Un Clúster de procesadores[*]

AUTORES: TERAN JULIO, Gerardo Enrique & PITA ESCAMILLA, Julián Rodrigo.[**]

Palabras clave: Algoritmo paralelo, migración de Kirchhoff, clúster, computación de alto rendimiento.

La migración 2D de Kirchhoff con Pre-apilamiento en tiempo utilizada para el análisis de datos sísmicos con el fin de crear un modelo en computadora de el subsuelo, es un procesos que puede llegar a demorar una gran cantidad de tiempo debido al tamaño de los datos que esté debe procesar y sumado a eso el proceso consta de bucles iterativos sobre los datos analizados.

El presente trabajo muestra el proceso de implementación de algoritmo de migración de Kirchhoff programado para ejecutarse en paralelo[1] sobre varios procesadores en un clúster de procesadores de alto rendimiento. Luego se comparan los tiempos empleados por el programa en paralelo con el programa secuencial con lo cual se hace una demostración de las grandes ventajas del programa paralelo y de utilizar computación de alto rendimiento para resolver problemas similares con otros algoritmos.

En este documento se muestran claramente tanto los resultados de la evaluación del algoritmo de Kirchhoff en paralelo como su modo de implementación. Adicionalmente a eso se detalla la instalación de todas las herramientas necesarias para el desarrollo de los algoritmos en paralelo, configuración de un clúster de procesadores y herramientas necesarias para la evaluación. Obviamente se hace una revisión del estado del arte con anterioridad y se presentan los conceptos básicos donde se incluyen las teorías necesarias en lo que respecta a sísmica, informática y electrónica; necesaria cuando se desea entender a fondo el contenido de este documento.

.

[*] Proyecto de grado

[**] Facultad de Ingenierías Físico-Mecánicas, Escuela de ingenierías Eléctrica, Electrónica y Telecomunicaciones. Director M.S.E.E. Ana Beatriz Ramírez, Codirectores Ing William Salamanca, Ing Sergio Abreo.

ABSTRACT

TITLE: Implementation and Evaluation of the 2D pre-stack time Kirchhoff migration parallel algorithm in a computer *cluster*.*

AUTHORS: TERAN JULIO, Gerardo Enrique & PITA ESCAMILLA, Julián Rodrigo.**

Keywords: Parallel Algorithm, Kirchhoff time migration, *cluster*, high performance computation.

The 2D pre-stack time Kirchhoff migration parallel algorithm, used for seismic data analysis in order to create a computer model of the subsurface, is a process that could last a lot of time due to the input data size to be processed and added to that the process consists of iterative loops on the analyzed data.

This work shows the implementation process of 2D Kirchhoff migration algorithm programmed to run in parallel[2] on multiple processors in a high performance computers *cluster*. Later, it's compared the time spent by the program in parallel and the sequential program as a demonstration of the great advantages when using parallel and high performance computing to solve similar problems with other algorithms.

This paper will make clear whether the results gotten when evaluating the parallel Kirchhoff migration algorithm, as the way it was implemented. In addition, here it is detailed the installation of all necessary tools to develop parallel algorithms, configuring a *cluster* of computers and tools for their evaluation. Of course, first it is made a review of the state of the prior art and presents basic concepts which are necessary theories with regard to seismic data processing, computer sciences and electronics, all necessary to entirely understand the content of this document.

.

* Grade thesis

** Ability of Physical-Mechanical Enginnerings, Faculty: Electrical, Electronic and Telecommunications, Director M.S.E.E. Ana Beatriz Ramírez, Codirectors Ing William Salamanca, Ing Sergio Abreo

1. INTRODUCCIÓN

A lo largo de los últimos años la exploración sísmica ha tratado de buscar, con ayuda de las imágenes sísmicas, lugares con presencia de yacimientos de petróleo o minerales. El primer paso en del proceso de exploración en la recolección de datos. La adquisición de datos es la base para el ciclo de vida de la sísmica, esta información debe ser muy precisa para tomar decisiones acerca del potencial de un yacimiento.

Los métodos de exploración sísmica se basan en la generación de ondas sísmicas producidas por una fuente artificial ya sea una explosión, un martillo, una pistola de aire y otros. Estas ondas se propagan a través del subsuelo. A medida que avanzan, parte de la energía de estas ondas es reflejada hacia la superficie y es registrada en una serie de sensores conocidos con el nombre de geófonos. Esta información es procesada luego mediante un software para obtener la respectiva imagen sísmica. La generación de imágenes es el método por el cual las reflexiones de las ondas sísmicas se despliegan a su posición correcta, los elementos principales son el apilamiento y la migración. El apilamiento mejora la relación señal-ruido y la migración usa un modelo de velocidades para redistribuir la energía reflejada a su verdadera posición. Existen varios tipos de migración, los cuales se usan dependiendo la complejidad de las zonas en estudio, para cuestiones de este trabajo nos enfocamos en el algoritmo de migración de Kirchhoff en 2D.

El entorno para la generación de las imágenes sísmicas es Linux (sistema operativo de distribución libre) y también el uso de software *Seismic Unix* (SU). Este software es de código abierto sobre plataformas Linux.

El mayor problema que tiene la construcción de una imagen sísmica es el tiempo que se requiere para procesar los datos para la construcción de la imagen. Mejorar el tiempo de procesamiento de una imagen es el principal objetivo de este

trabajo. Para alcanzar esto, se usa la programación en paralelo en redes de computadores. De esta manera se analiza la utilización de los clústeres (grupo de computadores conectados en red) Linux, los cuales ofrecen un gran número de posibilidades que facilitan el acceso a dispositivos y comunicaciones necesarias para la ejecución de programas en paralelo. Para la implementación de este tipo de programación en paralelo hemos usado la interfaz de paso de mensajes MPI (Message Passing Interface). Esta interfaz es un estándar de funciones contenidas en una librería de paso de mensajes para ser usada en programas que usen múltiples procesadores.

Este trabajo de grado se divide en cuatro partes. Empezando con una breve introducción y luego en el capítulo 2 de este documento hacemos un estudio general del procesamiento de datos sísmicos, programación en paralelo y la parte teórica de los clústeres de computadores, en el capítulo 3 realizamos un estudio de la implementación del algoritmo sobre el clúster de procesadores y finalmente en el capítulo 4 generamos un estudio de los resultados obtenidos en la ejecución del algoritmo en paralelo.

2. MARCO TEORICO

2.1 PROCESAMIENTO DE DATOS SÍSMICOS

La técnica usada para obtener la profundidad y la geometría de las estructuras geológicas en el subsuelo emplea el uso de ondas de sonido. Para adquirir los datos sísmicos se entierran y detonan pequeñas cargas explosivas para poder provocar una onda de sonido. A medida que estas ondas se desplazan, parte de la energía de estas ondas es reflejada hacia la superficie y es registrada en una serie de sensores conocidos con el nombre de geófonos. Los geófonos son transductores de desplazamiento, velocidad o aceleración que convierten el movimiento del suelo en una señal eléctrica.

Actualmente es un reto poder determinar con precisión el lugar de un pozo petrolero o un yacimiento de mineral antes de la perforación del suelo donde la forma geológica es complicada. Aquí es donde los métodos sísmicos especializados, como la migración en profundidad, puede dar una solución bastante confiable.

La exploración sísmica emplea ondas que se propagan a través del terreno y que se han generado de forma artificial. El objetivo es el estudio del subsuelo en general, con lo cual se puede conocer la estructura y los materiales que lo conforman.

La investigación sísmica consiste en generar ondas sísmicas mediante una fuente emisora y registrarlas en una serie de estaciones (geófonos) distribuidas sobre el terreno. Seguidamente con el análisis de las diferentes formas de onda, y sus tiempos de trayectoria se puede construir una imagen sísmica que después se relaciona con las capas geológicas.

Cuando una onda sísmica choca con un cambio de capa geológica, parte de la energía de la onda continúa en el medio, parte se refleja y el resto se transmite al otro medio con cambios en la dirección y velocidad de propagación.

Con el fin de conseguir un mejor reconocimiento de la zona de estudio se generan varias ondas sísmicas. El resultado es un grupo de trazas procedentes de todos los disparos que se crean (ondas sísmicas generadas), las cuales para la generación de una imagen sísmica se procesan y luego se reordenan en puntos de reflexión común los cuales poseen información de todas las reflexiones halladas. Los elementos principales en la generación de una imagen sísmica son el apilamiento y la migración. El apilamiento mejora la relación señal al ruido y la migración usa un modelo que lleva la energía sísmica recogida por los receptores a su posición geológica verdadera. La migración puede llevarse a cabo antes o después del apilamiento.

A continuación se muestran algunos conceptos empleados en la construcción de una imagen sísmica.

2.1.1 Apilamiento

El apilamiento (stack) es un proceso de compresión de los registros sísmicos en las que se le ajustan correcciones a cada punto común y se suman las amplitudes para obtener finalmente una sola traza asociada a cada punto común asumiendo ruido aleatorio no correlacionado se espera tener un proceso de apilamiento de la sección obtenida con una mejor relación señal-ruido.

2.1.2 Migración

La sísmica de migración de datos es una técnica de tratamiento de datos en la que se crea la imagen de la estructura de la tierra a partir de datos registrados por el estudio de reflexión de ondas. El objetivo de la migración es mejorar la resolución espacial moviendo los eventos bruscos a su verdadera posición.

Dos métodos importantes de migración son, migración en pre-apilado y post-apilado

2.1.2.1 Migración pre-apilado

La migración pre-stack se obtiene cuando los datos sísmicos son ajustados antes del apilamiento. El uso más común para la migración pre-stack es en la migración a profundidad. La migración a profundidad requiere que el usuario conozca las velocidades de las capas. Una vez que el usuario introduce estos datos de velocidad supuestos, obtenidos con métodos de análisis, se produce un error en la imagen. Este error se produce por inmersión o reflectores difracciones. La migración a profundidad ajustará estos errores en la imagen de acuerdo a la velocidad dada.

A menudo la migración pre-stack se aplica sólo cuando en las capas que se han observado perfiles complicados velocidad, o cuando las estructuras son demasiado complejas. En general, en la migración pre-stack, la profundidad y el tiempo, son una herramienta valiosa para mejorar la imagen de los datos sísmicos, pero está limitado por la cantidad de tiempo y dinero necesarios para llevar a cabo la migración pre-stack. Con los avances de la informática, la migración pre-stack es eventualmente más económica.

2.1.2.2 Migración post-apilado

La migración post-stack es el proceso de migración en la que los datos se apilan después de que se hayan migrado. Este proceso es por muchas razones más usadas, principalmente debidas a su costo razonable en comparación con la migración pre-stack. Al igual que en la migración pre-stack, la migración pos-stack se basa en la idea de representar las reflexiones y difracciones en una imagen. Esto se hace mediante una operación que implica la reorganización de la información sísmica para que las reflexiones y difracciones se organicen en su verdadera ubicación.

Una desventaja de la utilización de la migración post-stack frente a la migración pre-stack es que no dan resultados tan claros como pre-stack. La migración post-stack generalmente da buenos resultados cuando la depresión del subsuelo es pequeña.

2.2 SOFWARE DE PROCESAMIENTO DE DATOS SISMICOS SEISMIC UNIX

Seismic Unix (SU) es un software libre desarrollado en el Centro para Fenómenos Ondulatorios (CWP) de la escuela de minas del colorado (CEM) que contiene herramientas para el procesamiento y el modelado de los datos sísmicos y opera sobre el sistema operativo UNIX. Una característica importante de SU es que es una herramienta de código abierto lo cual permite un acceso total a los códigos de sus programas y esto lleva a que estos códigos puedan ser usados para emplearlo en el desarrollo de herramientas más elaboradas.

Para descargar de forma gratuita el paquete de *Seismic Unix* con su respectiva documentación necesaria para el manejo de las funciones se puede descargar de la página http://www.cwp.mines.edu/cwpcodes. Es necesario que la máquina en

la cual se quiera instalar use sistemas operativos UNIX y también que contengan un compilador C-ANSI

Como se mencionó *Seismic Unix* opera bajo el sistema operativo Linux. Por esto se hace necesario el manejo de Shell Scripting para el manejo de las funciones. S*eismic Unix* puede ser tratado como un lenguaje de computación y visto desde este punto de vista es necesario conocer cierta cantidad de vocabulario para poder realizar operaciones útiles, el paquete SU brinda ayuda acerca de estos comandos, para conocer la función de estos basta con escribir en la terminal de LINUX el nombre de la función que se desea utilizar.

Además de las ayudas con los comandos, *Seismic Unix* ofrece otras formas de ayuda como son:

Demos: contiene demostraciones del manejo de los programas que se deben usar en el Shell, para el uso y la forma de ejecución de los demos basta solo con seguir las instrucciones que vienen el archivo “README” que viene en cada clasificación de los demos.

Examples: esta utilidad es muy similar a los demos consiste en una serie de Scripts que muestran el manejo de los programas de SU, la diferencia con los demos viene en que estos Scripts no son ejecutables de forma inmediata, es necesario introducir datos de entrada para la ejecución de los programas.

Ayudas en la web[3]

[3] http://sepwww.stanford.edu/oldsep/cliner/files/suhelp/suhelp.html donde se puede encontrar una muy buena clasificación de los comandos de SU

2.3 PROGRAMACIÓN PARALELA

En un computador se distinguen dos partes, el hardware y el software. El hardware es la parte física de la máquina mientras que el software son los programas que se ejecutan sobre ella. El crecimiento del hardware es impulsado por el desarrollo de chips y nuevas tecnologías de fabricación, microprocesadores más rápidos y baratos, así como también el software produce avances rápidos, como sistemas operativos, lenguajes de programación y muchas herramientas complementarias.

El desarrollo de software se ha orientado hacia la programación en serie construyendo algoritmos cuyas instrucciones se implementan y ejecutan secuencialmente desde una unidad central de procesamiento de un ordenador y donde la velocidad de procesamiento depende de la rapidez con que los datos se mueven a través del hardware.

Debido a las actuales mejoras tecnológicas durante las últimas décadas, la capacidad de procesamiento que tiene una computadora ha aumentado, pero debido a que estas mejoras tienen un límite físico el aumento en la velocidad de procesamiento no es ilimitado. Existe una serie de problemas cuando se requiere un procesamiento más rápido y a partir de este problema surge la programación en paralelo.

El procesamiento en paralelo consiste en acelerar la ejecución de un programa mediante una descomposición de este, de tal forma que cada parte de código se ejecute en unidades de procesamiento diferentes. El problema de esta división es que sus partes sean independientes, cada parte es dividida en un conjunto de instrucciones y cada parte es ejecutada secuencialmente sobre una CPU de tal forma que cada una de las partes es resuelta simultáneamente.

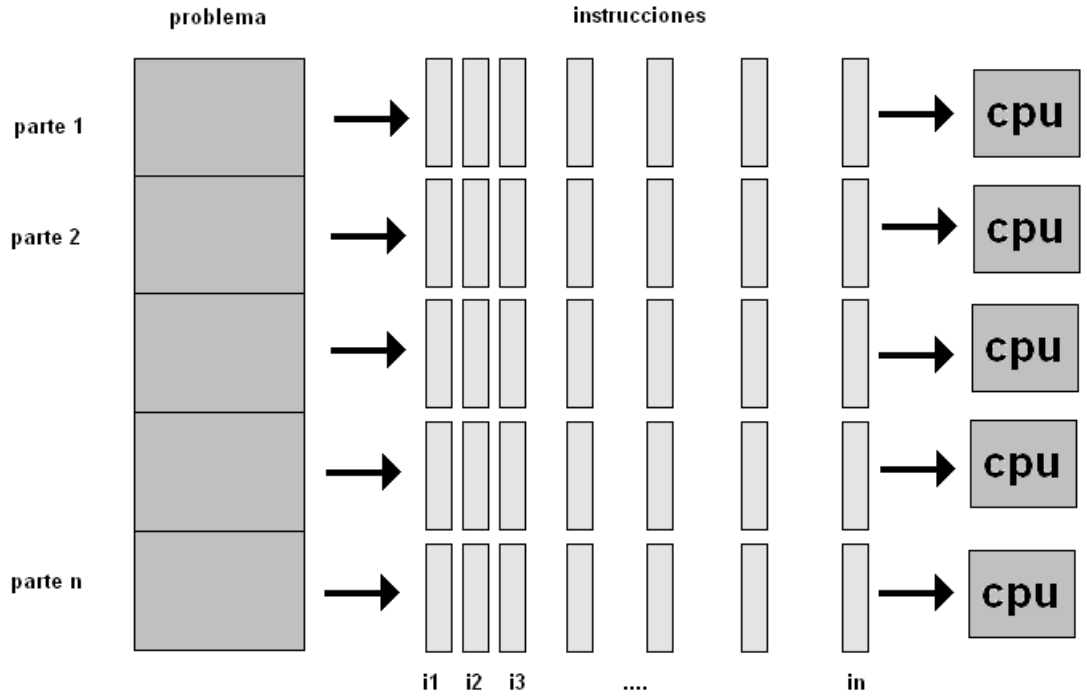

Figura 1 Programa en paralelo

Ref. Autores de este proyecto

Una de las razones por las cuales se desarrolló la programación en paralelo se da como una solución al problema de cuello de botella que significa la velocidad de un único procesador. Este desarrollo de programación lleva a una aceleración en el tiempo de ejecución. La aceleración que se puede alcanzar depende tanto del programa como de la arquitectura del computador en paralelo. La clave es poder separar un código y que cada parte de este opere sobre un procesador sin la necesidad de recurrir a los demás procesadores.

Las formas de obtener los trozos de código de un programa en paralelo son: descomposición funcional y descomposición de datos.

2.3.1 Descomposición de datos

Un buen ejemplo de paralelizado por descomposición de datos es el cálculo de una integral numérica o dicho de otro modo es el área bajo una curva. Basta con dividir los límites de integración entre los procesadores disponibles y que cada uno resuelva un fragmento sin preocuparse por los demás cálculos. Por último el resultado arrojado por cada procesador se suma. Con n procesadores es posible acelerar el cálculo como máximo n veces que haciendo el cálculo con un solo procesador.

2.3.2 Descomposición funcional

Consiste en que cada procesador realice una tarea determinada, cada tarea es responsable de una parte del proceso completo y es independiente de las demás. Los algoritmos de cada tarea son diferentes.

El procesamiento paralelo se clasifica en:
SIMD (Single Instruction, Multiple Data)
MIMD (Multiple Instruction, Multiple Data)

SIMD[4]

Esta arquitectura está construida sobre procesadores idénticos, donde cada cual posee una memoria local y ejecutan la misma secuencia de instrucciones pero con diferentes datos por esto los SIMD requieren menos memoria y el diseño de programas es más sencillo. El conjunto de los procesadores están siempre sincronizados. La memoria no se comparte. Este tipo de arquitectura requiere que exista una estación de trabajo que haga el trabajo de controlador.

[4] Peter S. Pacheco *parallel programing with MPI* Morgan Kaufmann Publishers Inc.1996 pag 14

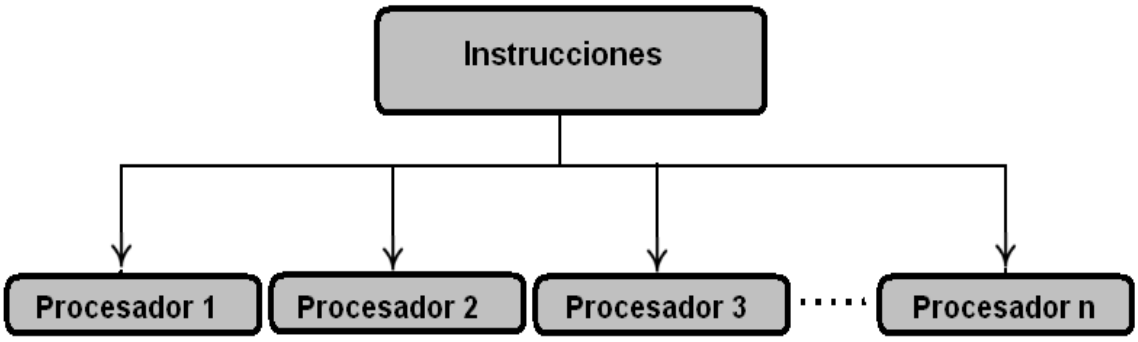

Figura 2 SIMD
Ref. Autores de este proyecto

MIMD[5]

En este tipo de arquitectura cada procesador trabaja de manera independiente y cada uno de los procesadores realiza un trabajo específico sobre una parte de la tarea completa. Cada procesador está ejecutando una tarea distinta esto es lo que se llama descomposición de control. Hay dos tipos: de memoria compartida y de memoria distribuida.

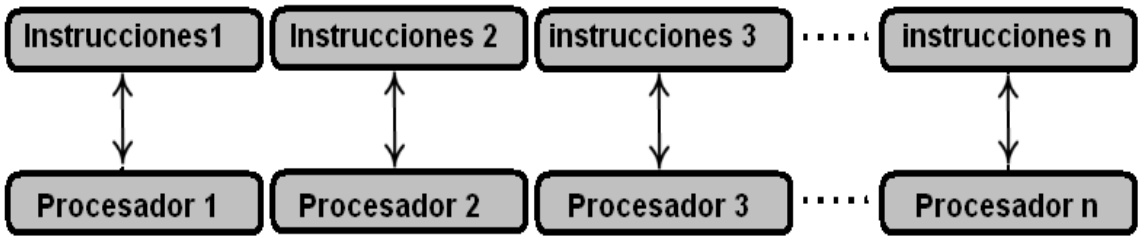

Figura 3 MIND
Ref. Autores de este proyecto

[5] Peter S. Pacheco *parallel programing with MPI* Morgan Kaufmann Publishers Inc.1996 pag 16

2.3.3 Memoria compartida

Una forma eficaz que tienen los procesadores para comunicarse consiste en compartir una zona de memoria. Para enviar datos a un procesador basta solo con escribirlos en la zona de memoria compartida y automáticamente estos datos estarán disponibles para otro procesador. Los accesos a memoria son uniformes puesto que todos los procesadores se encuentran igualmente comunicados con la memoria principal donde sus escrituras y lecturas tienen la misma latencia y el acceso a la memoria se hace por medio del canal común, este tipo de memoria es más fácil de programar que un sistema de memoria distribuida. Una desventaja es el acceso a memoria simultáneo. Para esto los sistemas operativos controlan el acceso a memoria con mecanismos tipo semáforo. Es decir, ya que los procesadores se encuentran conectados mediante el mismo canal, en el caso en el cual varios procesadores quieran acceder a la memoria de forma simultanea se crea un sistema que le da paso a un procesador mientras detiene el flujo de datos hacia los otros procesadores, luego le da paso a otro procesador y detiene las demás solicitudes a acceso a memoria de los otros procesadores, así de esta forma todos los procesadores van accediendo a los datos de la memoria.

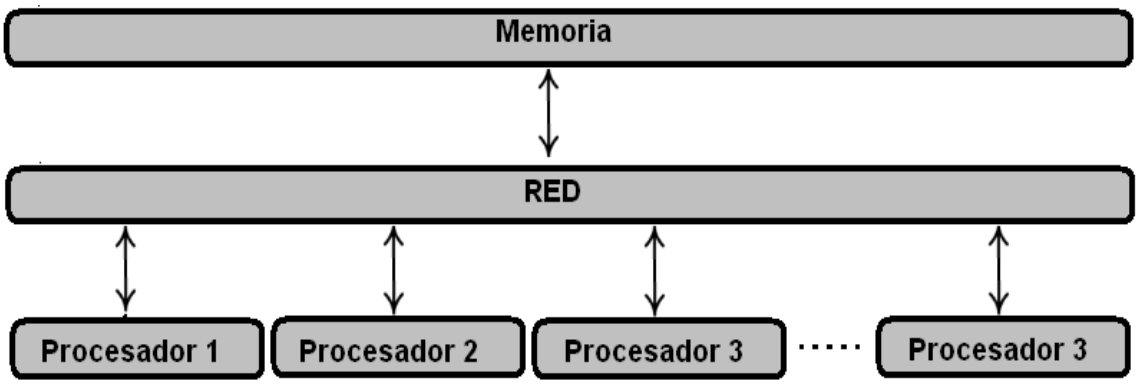

Figura 4 Memoria compartida

Ref. Autores de este proyecto

2.3.4 Memoria distribuida

Este tipo de sistemas tiene su propia memoria local, comparten memoria solo a través de mensajes. Cuando un procesador requiera utilizar los datos contenidos en la memoria de otro equipo deberá enviar un mensaje solicitándolos. Este tipo de comunicación se conoce como paso de mensajes.

Las desventajas de este tipo de memoria están en el acceso remoto a memoria ya que es lento conforme crece la máquina y la programación suele ser complicada porque se requiere programación paralela explícita.

Con el uso de paso de mensajes se pueden desarrollar eficientes programas paralelos. Las librerías de paso de mensajes proveen rutinas de inicialización así como también intercambio de paquetes de datos.

Las computadoras MIMD de memoria distribuida y donde su procesadores trabajan con su propio sistema operativo, memoria y sobre partes diferentes de un programa se conoce como sistemas de procesamiento masivamente paralelas (MPP).

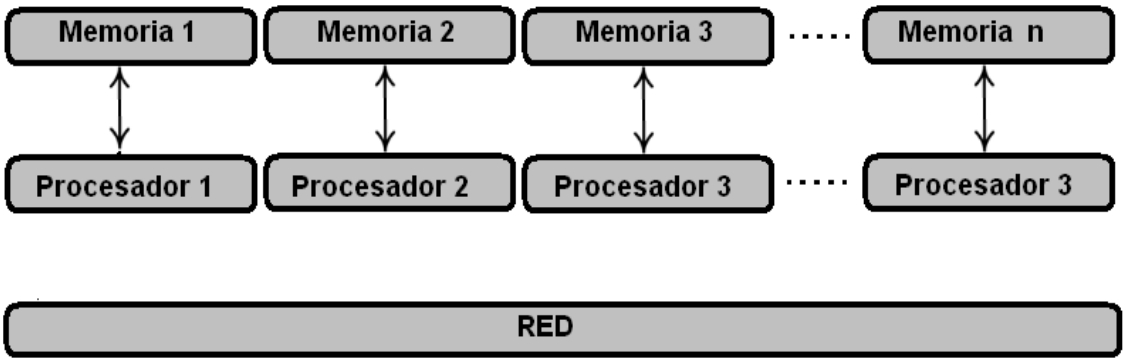

Figura 5 Memoria distribuida

Ref. Autores de este proyecto

2.3.5 Sistemas de memoria compartida-distribuida

La memoria compartida distribuida (DSM, distributed shared memory) es un conjunto de procesadores que tienen acceso a una memoria compartida pero sin un canal compartido. Cada procesador posee su memoria y se interconecta con otros procesadores con dispositivos de alta velocidad. La memoria compartida distribuida ahorra al programador lo concerniente al paso de mensajes.

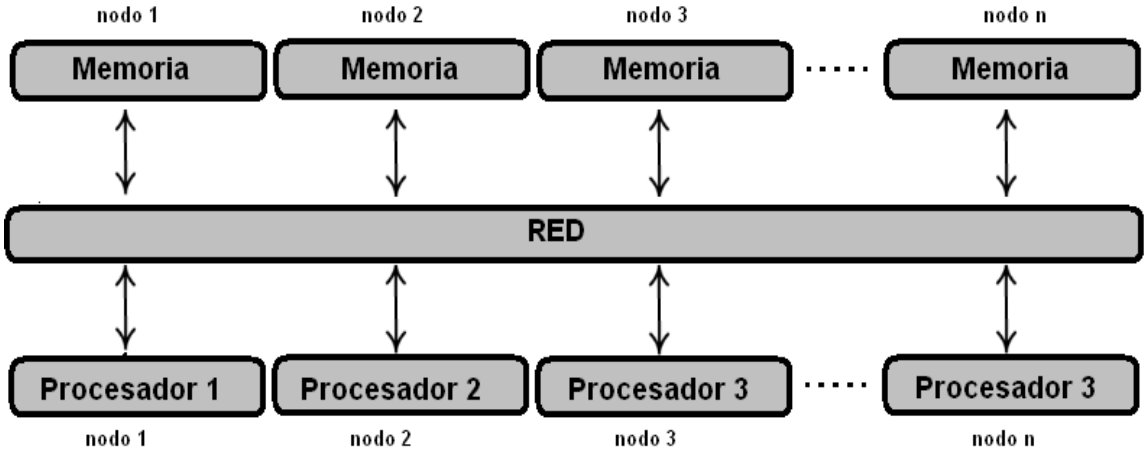

Figura 6 Memoria compartida-distribuida
Ref. Autores de este proyecto

2.3.6 Programación paralela en Linux

Linux tiene un importante impacto en la industria informática, debido a los beneficios que trae esta herramienta como lo son un sistema operativo abierto eficiente, fiable y de libre distribución.

2.4 LINUX DEBIAN

Debian es un sistema operativo gratuito y una de las distribuciones de Linux más importantes. Debian es una comunidad de desarrolladores y usuarios que mantienen un sistema operativo basado en herramientas GNU y en el kernel Linux por eso el nombre de GNU/Linux o Debian, fue creado por Lan Murdock en 1993 bajo el auspicio de la Free Software Foundation (FSF).

Debian mantiene tres versiones activas: estable, inestables y en prueba. La rama estable es el producto final de Debian, un sistema operativo completo estable y muy probado. En la rama inestable tiene lugar el desarrollo activo de Debian. Rama en prueba tiene el objetivo de detectar cualquier problema que atente contra los objetivos principales de la rama estable, en esta versión se encuentran paquetes que han estado en la rama inestable y que presentan pocos fallos estos paquetes son puestos a numerosas pruebas de confiabilidad, una vez la rama en prueba está bastante desarrollada y no presenta problemas pasa a ser la nueva rama estable

2.5 MPI[6]

MPI ("mensage Passing Interface", Interfaz de Paso de Mensajes) es un estándar de funciones contenidas en la biblioteca de paso de mensajes diseñada para ser usado en programas con aplicaciones con uso de múltiples procesadores. El modelo de programación tras MPI es MIMD (Multiple Instruction Stream, multiple Data Stream). Un caso particular de MIMD es que todos los procesadores ejecutan el mismo programa pero no la misma instrucción al mismo tiempo.

[6] Peter S. Pacheco *parallel programing with MPI* Morgan Kaufmann Publishers Inc.1996 pag 41

La importancia de MPI es dar al programador un conjunto de instrucciones para el diseño de su aplicación sin que sea necesario conocer el hardware sobre el que se va ejecutar, para así garantizar una gran portabilidad de programas en paralelo. Una gran característica es que no precisa de una memoria compartida por esto MPI es importante para sistemas con memoria distribuida.

A continuación se presentan algunas características de MPI

Operaciones colectivas. Es una operación que se ejecuta por todos los procesos que están interviniendo en el cálculo, se pueden efectuar operaciones de cálculo colectivo que son la operaciones de máximo, mínimo, suma, OR lógico etc., y operaciones de movimiento de dados utilizadas para reordenar e intercambiar datos entre un conjunto de procesos, esto es la difusión de un mensaje entre varios procesos.

Modos de comunicación. Las operaciones que MPI soporta son bloqueantes, no bloqueantes, asíncronas y síncronas. Una operación de tipo bloqueante, como su nombre lo indica, bloquea la comunicación emisor/receptor solo hasta que el mensaje se completa. Una comunicación no bloqueante permite solapar el cálculo con las comunicaciones; esto significa que la comunicación emisor/receptor esta desbloqueada aunque la operación no haya terminado.

Redes heterogéneas. Los programas hechos con MPI están pensados para poder ser ejecutados sobre redes heterogéneas con formatos y tipo de datos completamente diferentes.

MPI[7] es un sistema complejo que comprende 129 funciones con muchos parámetros y variantes enfocados a diseñar una interfaz de comunicación

[7] http://mathematica.nodo.cesga.es/component/option,com_wrapper/Itemid,98 en este sitio web se puede un toturial muy completo acerca del manejo de las instrucciones más importantes de MPI.

aplicable con una *comunicación* eficiente. También permitir implementaciones en ambientes heterogéneos, manejo de lenguajes C, C++ y Fortran 77.

Los programas hachos con MPI están generalmente ideados para ser usados en clústeres, una forma de lograr una comunicación entre las máquinas es vía SSH (Secure Shell).

MPICH es una distribución portable de MPI, usada en aplicaciones de memoria distribuida que utilizan computación paralela, es una implementación optimizada para entornos homogéneos lo que proporciona un mayor rendimiento en el paso de mensajes entre los nodos

2.6 SSH

Es un protocolo de comunicación para controlar un ordenador en remoto a través de la consola de Linux. SSH sirve para iniciar sesión en máquinas remotas su objetivo es establecer una conexión remota que haga posible la transmisión de todo tipo de datos.

2.7 CLÚSTER DE COMPUTADORES

En numerosas áreas científicas y de ingeniería se realizan simulaciones y cálculos numéricos complejos que consumen muchos recursos de cómputo y es necesario utilizar supercomputadoras que puedan dar una solución rápida al problema, otra alternativa para la solución rápida de problemas es la implementación de *clústeres* con software libre y con la utilización de equipos con pocas características que pueden reducir la inversión que se tiene que realizar. Los *clústeres* son un conjunto de piezas informáticas construidas con partes de hardware compartidas. Es decir un conjunto de microprocesadores conectados a través de una red de alta

velocidad que los relaciona de tal manera que funcione como un único procesador pero de un rendimiento mucho mayor a la normal.

2.7.1 Tipos de clústeres

Se pueden clasificar los tipos de "*clúster*" dependiendo de la función que se necesite.

2.7.1.1 Alta disponibilidad

Es conjunto de máquinas conectadas en una red, que se caracterizan por tener un servicio compartido y por estar en constante monitoreo pudiéndose clasificar en dos grupos. Su principal función es la de hacer que ninguno de sus servicios deje de funcionar. Un *clúster* de alta disponibilidad tiene como mínimo dos nodos, uno encargado de las funciones primarias y otro de copia de seguridad en modo de espera a que algún acontecimiento detenga las funciones de primer nodo, la copia de seguridad se hace cargo de brindar los servicios que se han dejado de dar sin la necesidad de una intervención externa.

2.7.1.2 Alto rendimiento

Tiene como objetivo principal alcanzar el mayor rendimiento en la velocidad de un proceso, permite tener un conjunto de computadoras trabajando en paralelo, está diseñado para el manejo de un gran número de cálculos, los motivos para usar un *clúster* de alto rendimiento es el tamaño del problema a resolver y el precio que una máquina necesita para resolver el problema.

Un *clúster* de alto rendimiento es independiente al hardware, pudiéndose adaptar a cualquier tipo de red, está basado en software libre, esto hace que sea posible la adición de nuevas funciones según se necesite.

2.7.1.3 Equilibrio de carga

Este tipo de *clúster* utiliza servidores web que verifica cual de las máquinas posee mayores recursos para asignarle el trabajo pertinente

2.7.2 Componentes de un clúster.

Un clúster necesita de varios componentes de software y de hardware para poder funcionar, como lo son: nodos, sistema operativo conexiones de red, middleware, protocolos de comunicación y servicios, aplicaciones.

2.7.2.1 Nodos

Son los computadores, un clúster puede ser construido con cualquier tipo de máquina. En informática un nodo es la intercepción de varios elementos que confluyen al mismo lugar. En relación a un clúster existen dos tipos de nodos.

2.7.2.1.1 Nodos dedicados

Son nodos que no disponen de un mouse, monitor, ni teclado y está exclusivamente destinado a realizar las tareas del clúster.

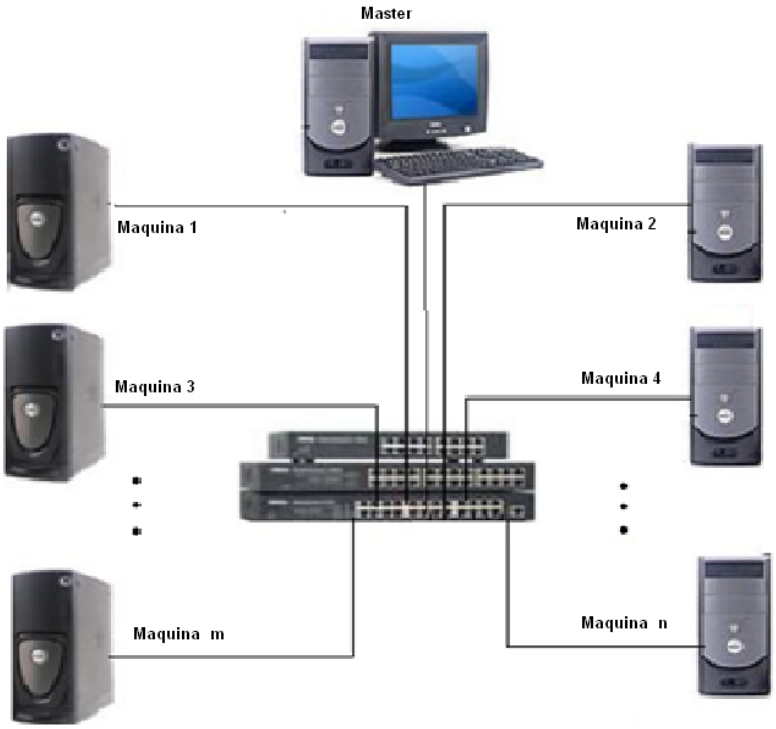

Figura 7. Nodos dedicados

Ref. Autores de este proyecto

2.7.2.1.2 Nodos no dedicados

Son nodos que poseen mouse, monitor y teclado y cuyo uso no está únicamente dedicado al clúster, el clúster hace uso de este nodo cuando su usuario no está usando la máquina.

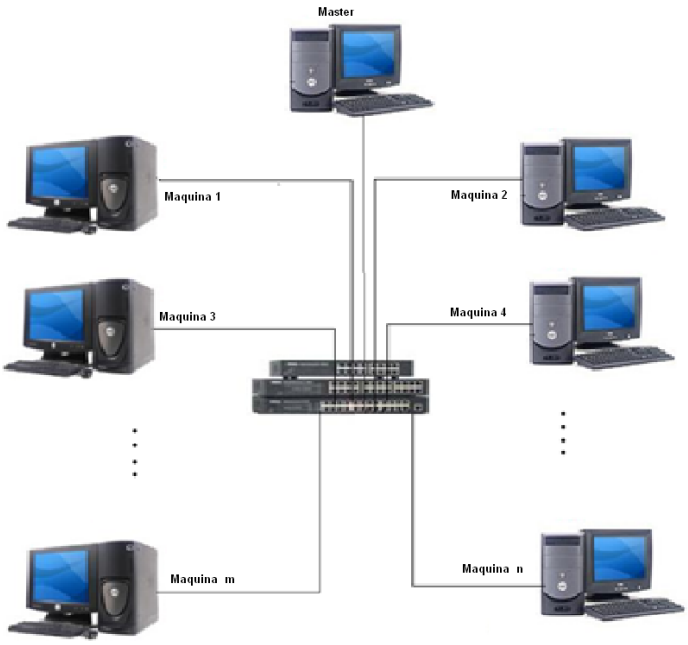

Figura 8. Nodos no dedicados

Ref. Autores de este proyecto

2.7.2.2 Sistema operativo

El sistema operativo a usar debe tener un entorno multiusuario, comúnmente son hechos con sistemas operativos UNIX.

2.7.2.3 Conexiones de red

Pueden ser muy variadas, pueden ser conexiones Ethernet con conexiones de red normal o sistemas de alta velocidad como fast Ethernet, SCI, etc.

2.7.3 Middleware

Es un software encargado de conectar componentes de software o aplicaciones, ofrece un conjunto de servicios que hacen posible en funcionamiento de aplicaciones distribuidas sobre plataformas heterogéneas, se encarga de la estabilidad del clúster detectando nuevas máquinas y añadiéndolas.
El middleware recibe los trabajos que entran al clúster, los distribuye para que el proceso sea más rápido. Esto se realiza mediante políticas que indican dónde y cómo se debe distribuir las tareas, por un sistema que controla la carga de la CPU y la cantidad de procesos actuando en el.

2.7.3.1 Protocolos de comunicación y servicios

Los protocolos de comunicación de un clúster normalmente son protocolos Fast Ethernet, *TCP/IP*

2.7.3.2 Aplicaciones

Las aplicaciones de un clúster son diversas y su uso está asociado a programas que requieren el uso de grandes recursos computacionales (velocidades de procesamiento, rapidez en la transmisión de la información, gran capacidad de disco duro y RAM, etc.). En las aplicaciones más comunes se encuentran servidores web y correo electrónico, como también bases de datos de alto rendimiento, entre otros usos.

NFS (Network File System) es un sistema de archivos virtual para entornos de red local que permite a una máquina UNIX montar un directorio de otra máquina y poder interactuar sobre él como si fuera propio.

Los usos de este sistema de directorios mediante NFS son muchos. Por ejemplo si tenemos una red de trabajo y se quiere que todas las máquinas tengan el mismo software y configuración podríamos compartir los directorios /usr para los programas y para la configuración /etc., esto desde un servidor común y con lo anterior se conseguirá una red uniforme. Otra posible aplicación la tenemos cuando se tienen clientes con poco espacio en disco y necesitan correr aplicaciones grandes, para esto se comparten los directorios de las aplicaciones mediante un servidor NFS y de esta forma se pueden correr las aplicaciones en los clientes.

3. IMPLEMENTACIÓN PARALELA DEL ALGORITMO DE MIGRACIÓN PRE APILAMIENTO EN TIEMPO DE KIRCHHOFF 2D

Las herramientas necesarias para la implementación de este trabajo fueron: un clúster de procesadores con la configuración necesaria para el paralelizado de un programa y las herramientas necesarias para la programación como lo son: el software libre *Seismic Unix* (SU) y las instrucciones y bibliotecas de paso de mensajes MPI, que para este trabajo se usó la versión libre MPICH. La instalación de las herramientas necesarias para la configuración del clúster con MPICH y SU se explica en detalle en al anexo 1

3.1 HERRAMIENTAS UTILIZADAS

Las herramientas necesarias para la programación son:

Linux. Debian Lenny, la cual es la última versión estable hasta la fecha.

El software libre *Seismic Unix* (SU)

Las instrucciones y bibliotecas de paso de mensajes MPI usada en este trabajo es MPICH.

Programa secuencial de migración de kirchhoff 2D con pre-apilado en tiempo (suktmig2d) disponible en SU. Este programa se utilizó para comparar los resultados con los resultados del programa en paralelo desarrollado e implementado en este proyecto.

La instalación de las herramientas necesarias para la configuración del clúster y la instalación y configuración de MPICH y SU sobre este mismo se explica en detalle en al anexo 1.

3.2 CLÚSTER UTILIZADO

El clúster utilizado en esta implementación consta de siete computadores con sistema Linux instalado (Debian 5) en donde se nombra uno de los equipos como maestro (master) y los otros 6 esclavos (maquina01, maquina02,…, maquina06). Para más detalles remitirse al anexo1.

El clúster utilizado es uno de nodos dedicados. A continuación se presenta esta disposición gráficamente:

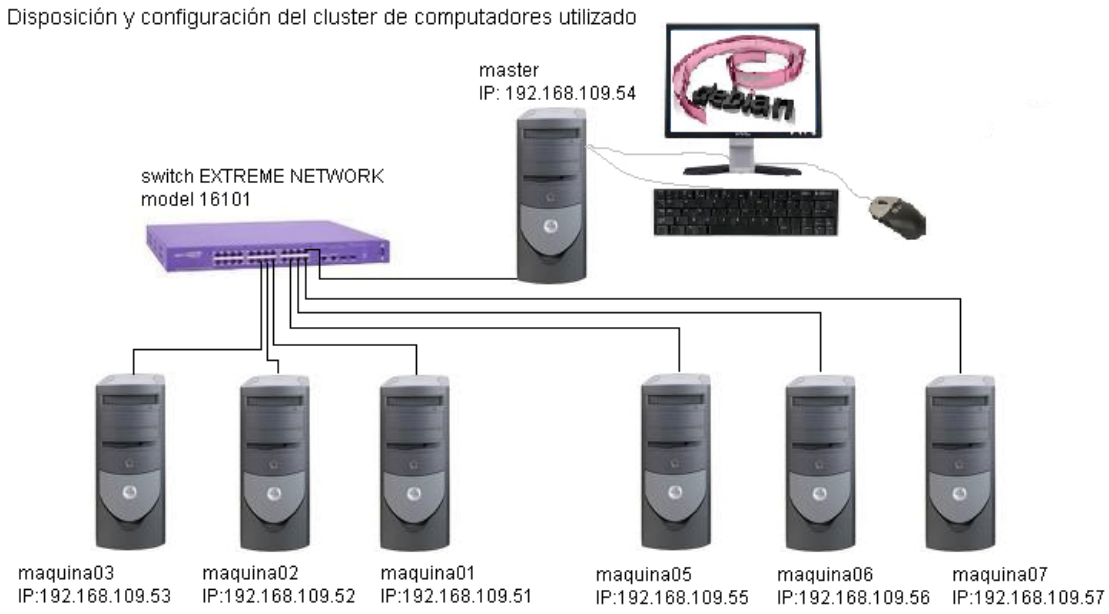

Figura 9 Disposición y configuración del clúster utilizado.

Ref.: Autores de este proyecto

Las características de cada máquina, Incluyendo el master, son:

Descripción: Computador torre mini
Producto: OptiPlex GX620
Vendedor: Dell Inc.
Peso: 32 bits
Capacidad: smbios-2.3 dmi-2.3 smp-1.4 smp

Núcleo
Descripción: Motherboard
Producto: 0HH807
Vendedor: Dell Inc.
ID Física: 0
Serial: ..CN1374063L01K1.

CPU

Descripción: CPU
Producto: Intel(R) Pentium(R) 4 CPU 3.20GHz
Vendedor: Intel Corp.
ID física: 400
Información del bus: cpu@0
Versión: 15.4.3
Serial: 0000-0F43-0000-0000-0000-0000
Ranura: Microprocessor
Tamaño: 3200MHz
Capacidad: 4GHz

Ancho: 64 bits
Reloj: 800MHz

Memoria
Descripción: System Memory
ID física: 1000
Ranura: System board or motherboard
Tamaño: 2GiB

El switch utilizado es un EXTREME NETWORKS modelo 16101 que posee las siguientes características:

Número de puertos: 48
Ranuras de expansión: (4 Total) SFP (mini-GBIC) Shared
Voltaje de entrada: 110V AC, 220V AC
Rango del voltaje de entrada: 100V AC to 240V AC
Dimensiones: 1.75" Height x 17.4" Width x 16.9" Depth
Puerto Ethernet Gigabits: Yes
Peso: 11.02 lb
Nombre del producto: Summit 400-48t Stackable Layer 3 Switch
Tipo de producto: Layer 3 Switch
Interfaces/Puertos: 48 x RJ-45 10/100/1000Base-T LAN, 1 x RJ-45 10/100/1000Base-T Management, 1 x Serial Management, and 2 x Stack
Rendimiento: 160 Mbps Switching Fabric, 101 Mpps Forwarding Rate

3.3 ALGORITMOS PARALELOS DESARROLLADOS PARA ESTA IMPLEMENTACIÓN

Como punto de referencia se muestran los dos grandes procesos de la migración secuencial:

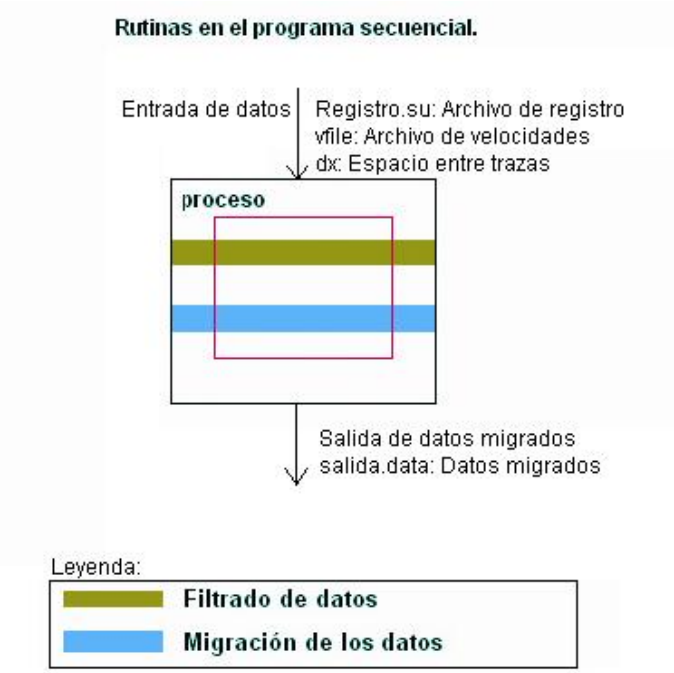

Figura 10. Rutinas en el programa secuencial

Ref.: Autores de este proyecto

Para el desarrollo de este proyecto se programaron dos algoritmos en paralelo los cuales se nombrarán de aquí en adelante como **migración b** y **migración d.**

El archivo que contiene las trazas es básicamente una matriz donde las columnas representan las trazas y a cada fila se le llama “secciones de traza”. La migración analiza las trazas de una forma independiente, es decir; el análisis de la traza *i* no depende de los resultados del análisis de otra traza. Igualmente dentro del proceso de analizar una sola traza, ésta se debe analizar por secciones de traza lo cual también es independiente del procesamiento de una sección u otra. Esto se

ha aprovechado en los algoritmos paralelos migración b y migración d para distribuir, entre todas las máquinas del clúster, el procesamiento de las trazas y secciones de trazas.

Los procesos en la Figura 10.1 representan “procesos MPI”, los cuales se ejecutan de manera simultánea distribuidos entre todas las máquinas. En este trabajo siempre se ejecutan, como máximo, tantos procesos como número de máquinas se disponga, de esta forma queda un solo proceso en cada máquina y se asegura que el clúster tiene el máximo rendimiento para esta aplicación. Debido a esto, de ahora en adelante no se hará distinción entre la máquina n y el proceso n.

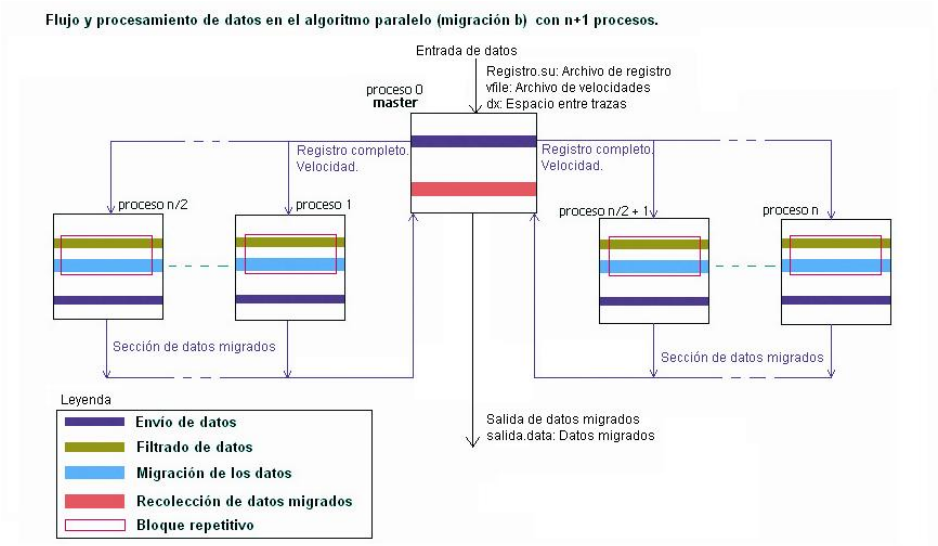

Figura 11.1 Procesos MPI

Ref.: Autores de este proyecto

Las trazas son distribuidas equitativamente entre las máquinas nombradas de 1 a n, el total de trazas son tomadas por el grupo de máquinas a la izquierda al que se llamará grupo1 (Figuras 11 y 12) y el total de trazas son tomadas por el grupo de máquinas a la derecha al que se hará referencia como grupo2.

Cada máquina analiza todas las trazas correspondientes a su grupo pero cada máquina dentro de un mismo grupo analiza secciones únicas dentro de las trazas.

Por ejemplo. Si en el grupo1 se cuenta con 10 máquinas y se tienen 100 trazas en total por analizar y cada traza consta de 500 secciones, corresponden 100 trazas para el grupo1 y las mismas 100 trazas para el grupo2, entonces cada máquina del grupo1 trabaja sobre las 100 trazas pero la máquina01 analiza las secciones de traza 0, 11, 21, 31,…, 491 mientras que la máquina02 analiza las secciones de traza 1, 12, 12,…, 492 así, hasta la máquina10 que le correspondería analizar las secciones de traza 9, 19, 29,…, 499. Nótese que el incremento de secciones es igual al número de máquinas del grupo que para este ejemplo fue 10. Lo mismo se aplica para las máquinas de grupo2 las cuales realizan un proceso de cálculo diferente al que realizan las máquinas del grupo1.

Si en el ejemplo anterior se aumenta el número de máquinas del grupo1 a 20, las secciones de trazas que le corresponde analizar a cada máquina serán como se lista a continuación.

Número de máquinas del grupo = 20.
Maquina01: Analiza 100 trazas. Secciones 0, 19, 39,…, 479 de dichas trazas.
Maquina02: Analiza 100 trazas. Secciones 1, 20, 40,…, 480. De dichas trazas.
Maquina03: Analiza 100 trazas. Secciones 2, 21, 41,…,481. De dichas trazas.
.
.
.
Maquina20: Analiza 100 trazas. Secciones 19, 20, 40,…,499 de dichas trazas.

Los resultados son enviados desde cada máquina hacia la máquina master (nodo maestro) en la cual son sumados los resultados del grupo1 a los resultados del grupo2 para formar la salida (salida.data ó datap).

El proceso descrito anteriormente es realizado por ambos algoritmos programados en paralelo (migración b y migración d) pero la diferencia entre los dos está en la forma como se aplica el filtrado de datos que hace parte de la migración en tiempo. En consecuencia es necesario mayor ó menor flujo de datos a través de la red local y mayor o menor carga de datos por procesar en cada máquina incluida el nodo maestro. A continuación se describen ambos algoritmos.

Migración b: Envía la totalidad del registro de entrada (datos de entrada) desde el nodo maestro a cada máquina al inicio de la ejecución del algoritmo. Cada máquina realiza sus propios filtros y devuelve una sección de datos migrados al nodo maestro, el cual se encarga de guardarlos en un solo archivo de salida (salida.data ó datap ó datap). A continuación se muestra un esquema del flujo de datos que se realiza en la migración b. Ver Figura 11.

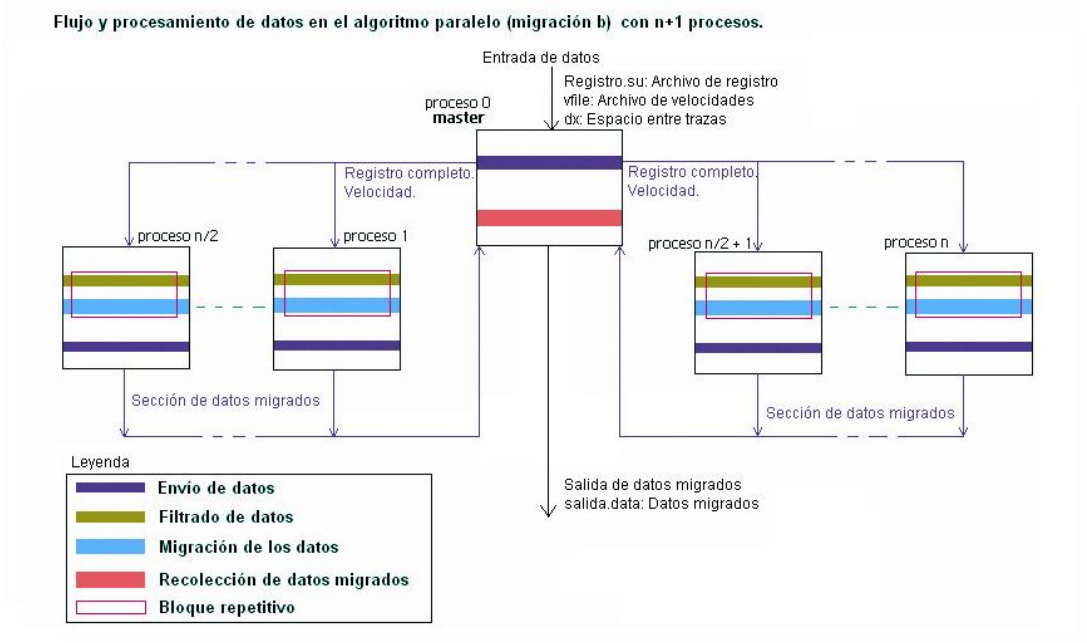

Figura 12 Flujo y procesamiento de datos en el algoritmo paralelo (migración b)

Ref.: Autores de este proyecto

Migración d: Éste a diferencia del anterior no envía la totalidad de los datos a cada máquina sino que envía las secciones de datos que le corresponderá procesar a cada máquina. En consecuencia, es necesario que el nodo maestro realice el filtrado de los datos que le corresponden a cada uno de las otras máquinas. Ver Figura 12.

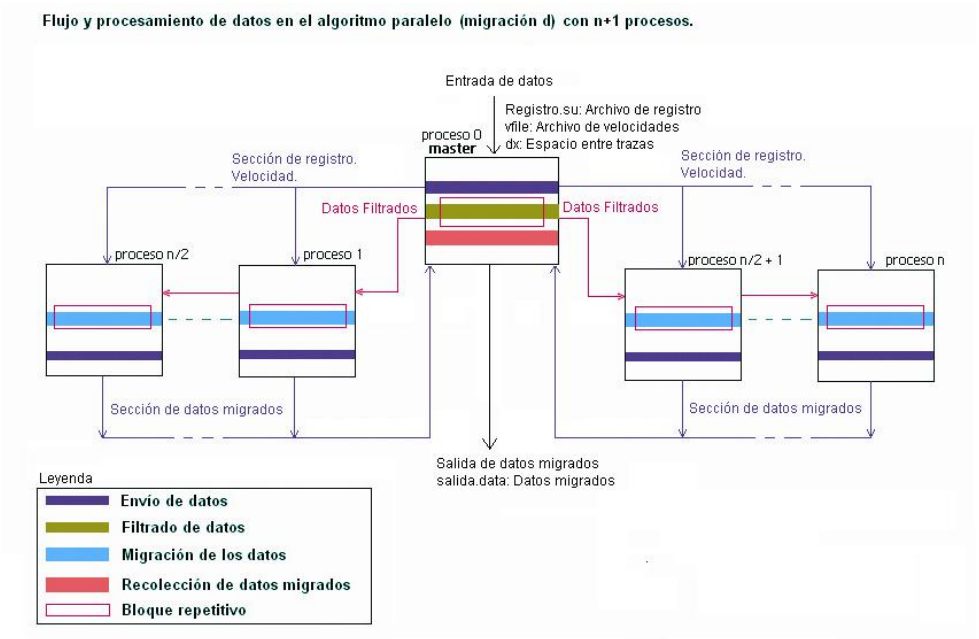

Figura 13 Flujo y procesamiento de datos en el algoritmo paralelo (migración d)

Ref.: Autores de este proyecto

Para más detalle de los dos algoritmos programados remítase al anexo2.
Para ver el código paralelo, con comentarios explicativos remítase al anexo2.

Resumen de esta sección:

Para la implementación del algoritmo de migración de kirchhoff en 2D con pre-apilamiento en tiempo se utilizó un clúster de procesadores configurado como se

muestra en la figura 9, Debian lenny, *Seismic Unix* y MPICH y se programó dicho algoritmo, en paralelo, de dos formas diferentes (migración b, migración d) con el objetivo de comparar el rendimiento de estos dos algoritmos paralelos con el algoritmo secuencial que se instala con el paquete S*eismic Unix*.

4. EVALUACIÓN DEL ALGORITMO SOBRE EL CLÚSTER DE COMPUTADORES

Se compara el rendimiento de los programas desarrollados en paralelo (migración b y migración d) con el rendimiento del programa secuencial (suktmig2d) de Seismic Unix. Incluso, se comparan el rendimiento de un algoritmo paralelo con respecto al otro con el fin de determinar cuál algoritmo tiene el mejor rendimiento con el coste más pequeño en lo que respecta al uso de la red. Esto último se hace monitorizando el uso del hardware dedicado a la comunicación entre procesos, que para este trabajo, es igual ó menor al número de máquinas utilizadas en el clúster.

Como se mencionó antes el objetivo de la implementación está enfocado a la disminución del tiempo total de ejecución del algoritmo secuencial que realiza la migración. Es por eso que se presenta en este capítulo una serie de tablas y gráficas donde la variable principal es el tiempo de ejecución de cada algoritmo (paralelo y secuencial).

4.1 COMPARACIÓN DE RESULTADOS

4.1.1 Comparación de tiempos de ejecución

A continuación se tabulan los tiempos empleados por los programas en paralelo y los tiempos empleados por el programa secuencial para diferentes tamaños de datos de entrada. Los valores tabulados son el promedio de dos ó tres resultados consecutivos obtenidos para cada tamaño de datos y número de máquinas.

Tiempos de ejecución para el algoritmo secuencial de SU y los algoritmos paralelos implementados							
Tamaño de los datos de entrada	Tiempo empleado por el algoritmo Secuencial de SU (suktmig2d)	Tiempo empelado por el algoritmo paralelo sobre el *"clúster"* (migración_b)			Tiempo empleado por el algoritmo paralelo sobre el *"clúster"* (migración_d)		
		3 máquinas	5 máquinas	7 máquinas	3 máquina	5 máquina	7 máquina
219 kb. 100 trazas	5,65s	3,20s	3,22s	3,96s	2,7s	3,2s	3,69s
1,06 Mb. 500 trazas	36,05s	24,1s	18,1s	16,28s	14,6s	12,6s	13,5s
2,13 Mb. 1000 trazas	2 min1s	1min12s	47,28s	39,60s	54,94s	28,7s	25,6s
3,20 Mb. 1500 trazas	2min11s	2min22s	1min28s	1min10s	1min 59s	1min 1s	42,3s
4,27 Mb. 2000 trazas	6min 55s	3min54s	2min19s	1min48s	3min 23s	1min 43s	1min 11s
5,34 Mb. 2500 trazas	10min 19s	5min43s	3min20s	2min32s	5min 7,5s	2min 36s	1min 47s
6,41 Mb. 3000 trazas	14min 42s	8min3s	4min35s	3min28s	7min 23s	3min 43s	2min 32s
8,55 Mb. 4000 trazas	25min 36s	13min48s	7min38s	5min36s	13min 6s	6min 35s	4min 29s
10,7 Mb. 5000 trazas	39 min 35s	19min48s	11min27s	8min17s	20min 24s	10min 16s	6min 58s

12,8 Mb. 6000 trazas	56min 40s	25min50s	18min4s	13min2s	29min 24s	14min 45s	10min 2s
21,4 Mb. 10000 trazas	2h 34min	51min15s	47min28s	38min54s	49min 9s	24min 42s	27min 42s
32,1 Mb. 15000 trazas	3h 57min	80min6s	74min44s	63min32s	74min	37min 3s	62min 22s

Tabla 4-1 Tiempos de ejecución para el algoritmo secuencial de SU y los algoritmos paralelos implementados.

La siguiente gráfica es un diagrama de barras donde se muestra el **tiempo de ejecución** de **cada algoritmo** para **diferentes tamaños de datos de entrada**. Dado un dato de entrada de tamaño fijo, se procede a correr el algoritmo paralelo sobre 3, 5 y 7 máquinas. Luego se cambia el dato de entrada por uno de diferente tamaño y se ejecuta nuevamente el algoritmo paralelo sobre 3, 5 y 7 máquinas. La barra de color azul oscuro representa el tiempo empleado por el programa secuencial para cada tamaño de datos de entrada. Recordar que se tienen dos algoritmos paralelos (migración b y migración d)

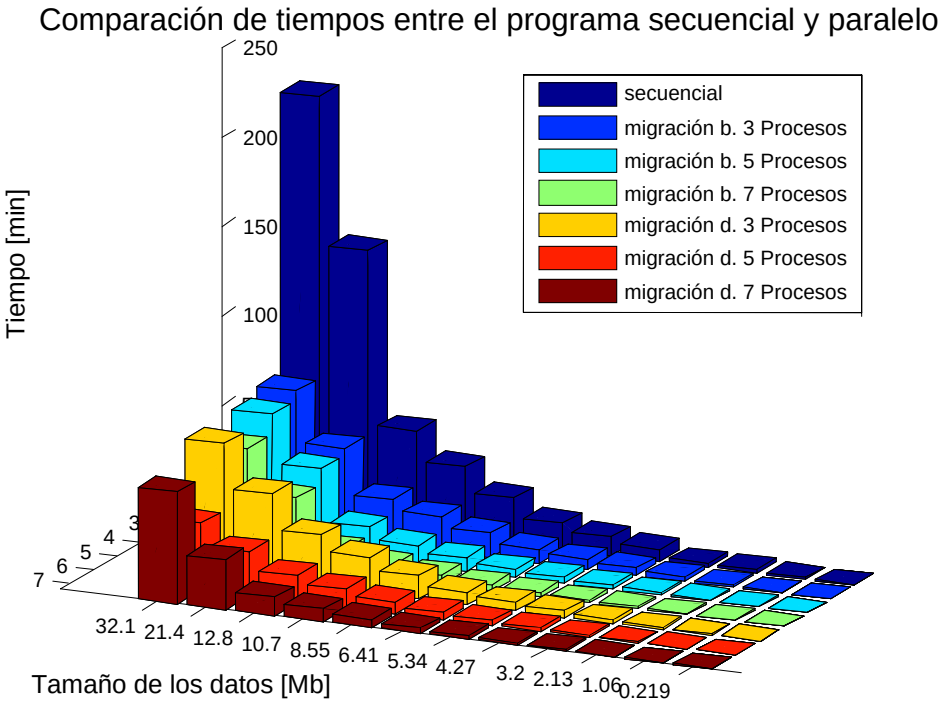

Figura 14 Tiempos de ejecución para los programas paralelos y secuencial. Diagrama de barras.

Ref.: Autores de este proyecto

A continuación se presenta la gráfica de Tiempo de ejecución vs Tamaño de los datos para los diferentes algoritmos paralelos (**migración b** y **migración d**) y para el algoritmo secuencial.

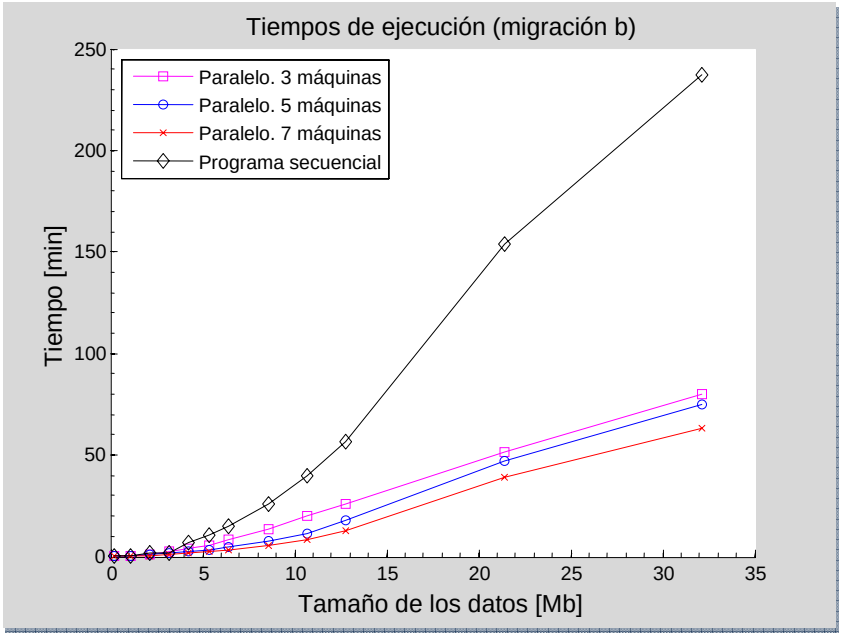

Figura 15 Tiempos de ejecución vs Tamaño de los datos. Migración b.
Ref.: Autores de este proyecto

En la Figura 14, La gráfica fucsia muestra el tiempo empleado por el programa paralelo (migración b) corriendo sobre tres máquinas, para aplicar la migración sobre datos de entrada de diferentes tamaños. La gráfica azul muestra el tiempo empleado por el programa paralelo (migración b) corriendo sobre 5 máquinas, la gráfica naranja muestra el tiempo empleado por el programa paralelo (migración b) corriendo sobre 7 máquinas. La gráfica de color negro es el tiempo que gasta el programa secuencial de Seismic Unix (suktmig2d) para aplicar la migración a los datos.

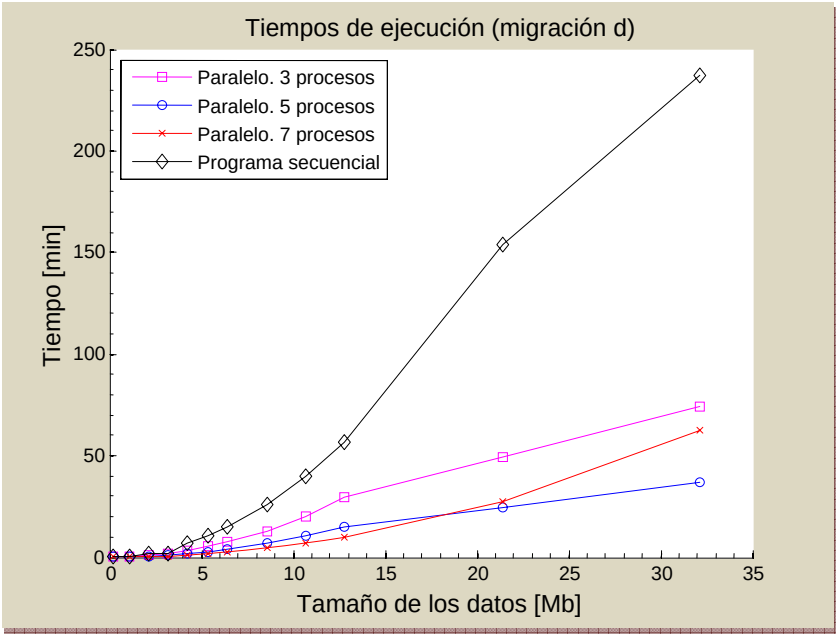

Figura 16 Tiempos de ejecución vs Tamaño de los datos. (migración d)
Ref.: Autores de este proyecto

En la Figura 15, La gráfica fucsia muestra el tiempo empleado por el programa paralelo (migración d) corriendo sobre tres máquinas, para aplicar la migración sobre datos de entrada de diferentes tamaños. La gráfica azul muestra el tiempo empleado por el programa paralelo (migración d) corriendo sobre 5 máquinas, la gráfica naranja muestra el tiempo empleado por el programa paralelo (migración d) corriendo sobre 7 máquinas. La gráfica de color negro es el tiempo que gasta el programa secuencial de Seismic Unix (suktmig2d) para aplicar la migración a los datos. Toma más tiempo la ejecución sobre 7 que sobre 5 máquinas porque durante el proceso sobre 7 máquinas se envían más cantidad de datos. La migración d envía datos durante la mayor parte del tiempo. Esto es debido a que los filtros se aplican en el nodo maestro y se envía constantemente secciones de datos filtrados hacia las máquinas (Esta es una característica de la migración d y representa la diferencia más relevante con respecto a la migración b. (Véase sección 3)

A continuación se presenta la gráfica de **Tiempo de ejecución vs Número de procesos** para los diferentes **algoritmos paralelos** (migración b y migración d) para tamaños de datos fijos.

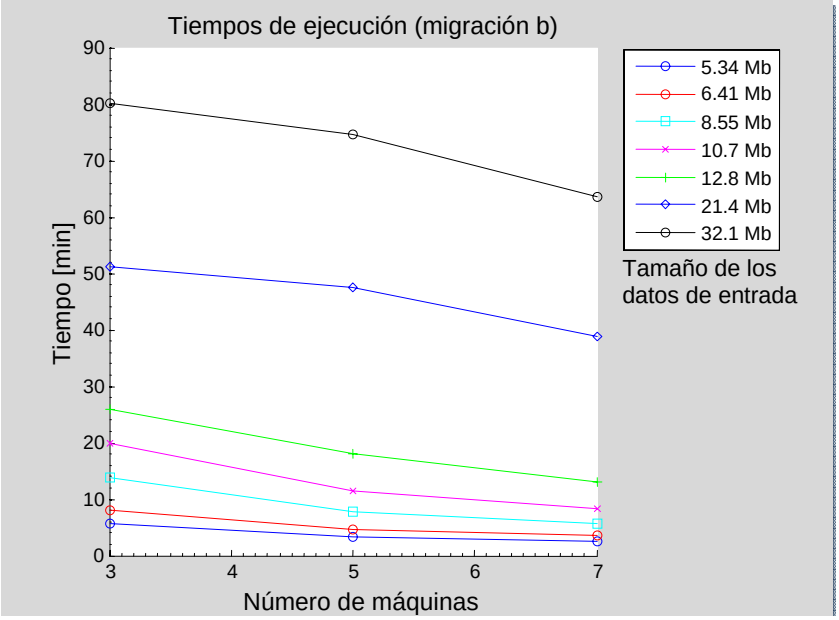

Figura 17 Tiempos de ejecución vs Número de procesos, migración b
Ref.: Autores de este proyecto

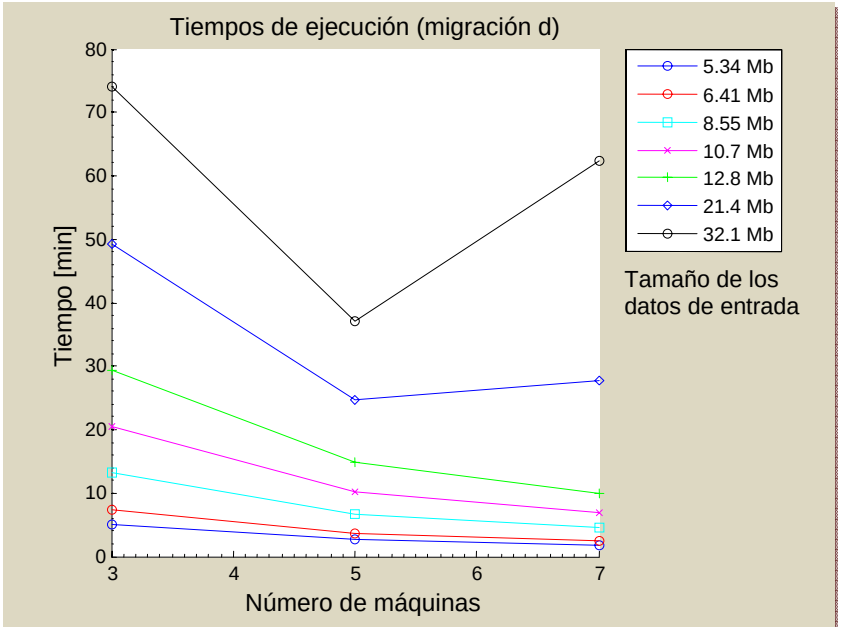

Figura 18 Tiempos de ejecución vs Número de procesos, migración d

Ref.: Autores de este proyecto

En las Figuras 16 y 17 Se muestra el tiempo que emplean los algoritmos paralelos (migración b en la Figura 16 y migración d en la Figura 17) corriendo sobre 3, 5 y 7 máquinas, para completar la migración. Se muestra los resultados para varios tamaños de datos de entrada.

A continuación se presenta el porcentaje en tiempo de la migración en paralelo con respecto a la secuencial para diferentes tamaños de datos de entrada.

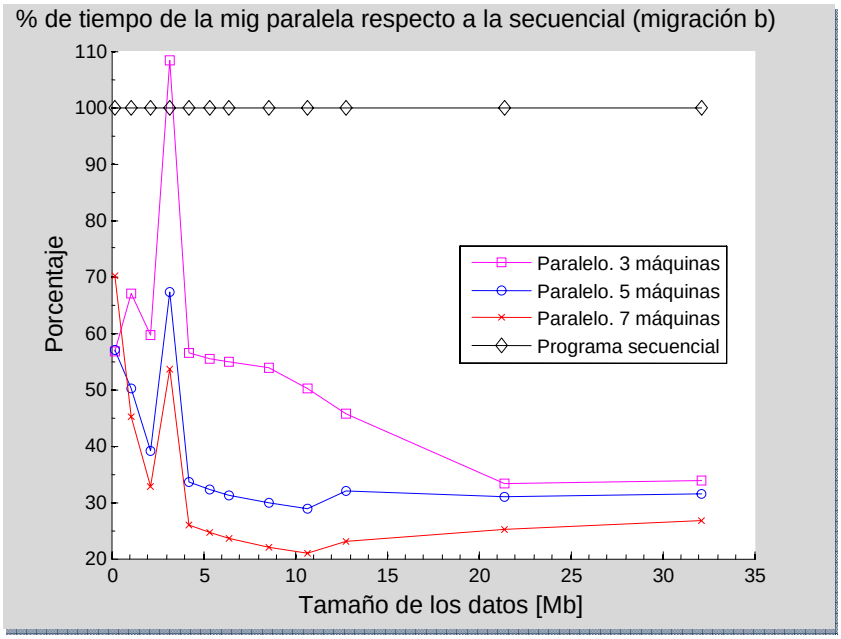

Figura 19. % de tiempo de la migración paralela respecto a la secuencial (migración b)
Ref.: Autores de este proyecto

En la Figura 18, Las gráficas fucsia, azul y naranja, muestran el porcentaje de tiempo empleado por el programa paralelo (migración b) corriendo sobre 3, 5 y 7 máquinas respectivamente, para completar la migración sobre datos de entrada de diferentes tamaños. Este porcentaje de tiempo es medido con respecto al tiempo total que emplea el programa de migración secuencial. Por ejemplo: Para datos de entrada de 21,4 Mb el programa paralelo (migración b) corriendo sobre 3 máquinas, empleó cerca de un 34 % del tiempo que le tomó al programa secuencial hacer la migración sobre dichos datos de 21,4Mb.

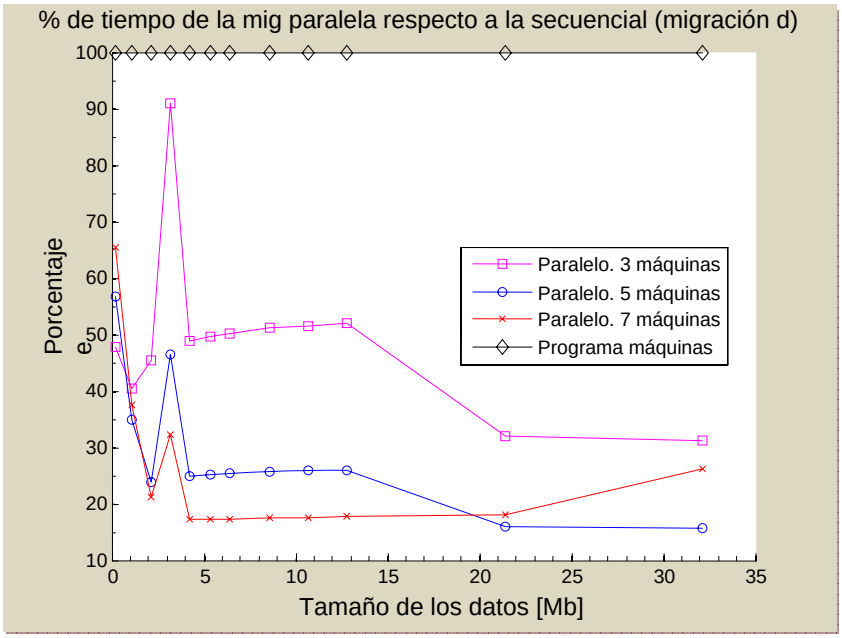

Figura 20% de tiempo de la mig paralela respecto a la secuencial (migración d)
Ref.: Autores de este proyecto

En la Figura 19, Las gráficas fucsia, azul y naranja, muestran el porcentaje de tiempo empleado por el programa paralelo (migración d) corriendo sobre 3, 5 y 7 máquinas respectivamente, para completar la migración sobre datos de entrada de diferentes tamaños. Este porcentaje de tiempo es medido con respecto al tiempo total que emplea el programa de migración secuencial. Por ejemplo: Para datos de entrada de 21,4 Mb el programa paralelo (migración d) corriendo sobre 5 máquinas, empleó cerca de un 5 % del tiempo que le tomó al programa secuencial hacer la migración sobre dichos datos de 21,4Mb.

A continuación se muestra el porcentaje de tiempo ahorrado al utilizar la migración en paralelo sobre diferente número de procesos.

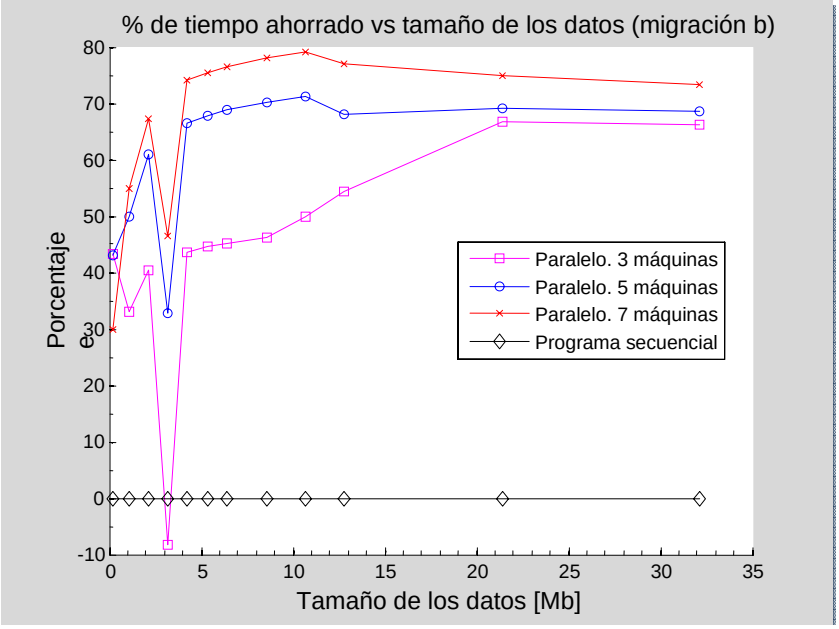

Figura 21 : % de tiempo ahorro vs tamaño de los datos (migración b)
Ref.: Autores de este proyecto

En la Figura 20, Las gráficas fucsia, azul y naranja, muestran el porcentaje de tiempo ahorrado al usar el programa paralelo (migración b) corriendo sobre 3, 5 y 7 máquinas respectivamente, para migrar datos. Este porcentaje de tiempo es medido con respecto al tiempo total que emplea el programa de migración secuencial. Por ejemplo: Para datos de entrada de 12 Mb el programa paralelo (migración b) corriendo sobre 7 máquinas, ahorró cerca de un 80 % del tiempo que le toma a la migración secuencial migrar dichos datos de 12 Mb.

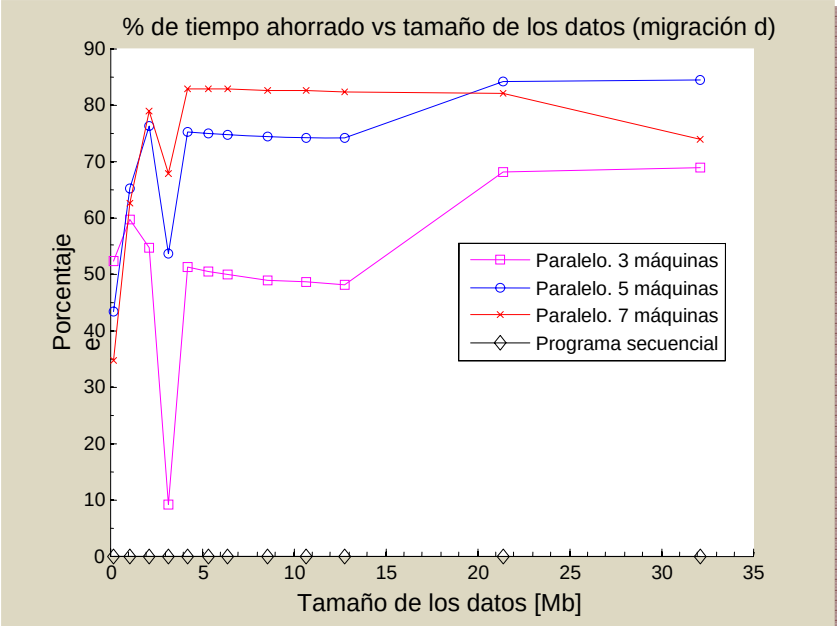

Figura 22 % de tiempo ahorrado vs tamaño de los datos (migración d)
Ref.: Autores de este proyecto

En la Figura 21, Las gráficas fucsia, azul y naranja, muestran el porcentaje de tiempo ahorrado al usar el programa paralelo (migración d) corriendo sobre 3, 5 y 7 máquinas respectivamente, para migrar datos. Este porcentaje de tiempo es medido con respecto al tiempo total que emplea el programa de migración secuencial. Por ejemplo: Para datos de entrada de 4,5 Mb el programa paralelo (migración d) corriendo sobre 5 máquinas, ahorró cerca de un 75 % del tiempo que le toma a la migración secuencial migrar dichos datos de 4,5 Mb.

4.1.2 Comparación de valores de salida

Los procesos de migración paralela arrojan datos iguales a los de la migración secuencial de SU. Dicha comparación se hizo con el siguiente algoritmo.

.

.

.

```
/*data: Datos de salida de la migración secuencial*/
/*datap: Datos de salida de la migración paralela*/
contador=0;
for (i=1;i<sizeof(data)[2]; ++i) {
        for (k=1;i<sizeof(data)[1]; ++k) {
                If (datap[k][i]==data[k][i]) ++contador;
        }
}
fprintf("contador=%i, k*i=%i",contador,k*i);
.
.
.
```

A continuación se muestra la gráfica para los resultados de la migración hecha para 3000 trazas utilizando el algoritmo en paralelo y el algoritmo secuencial.

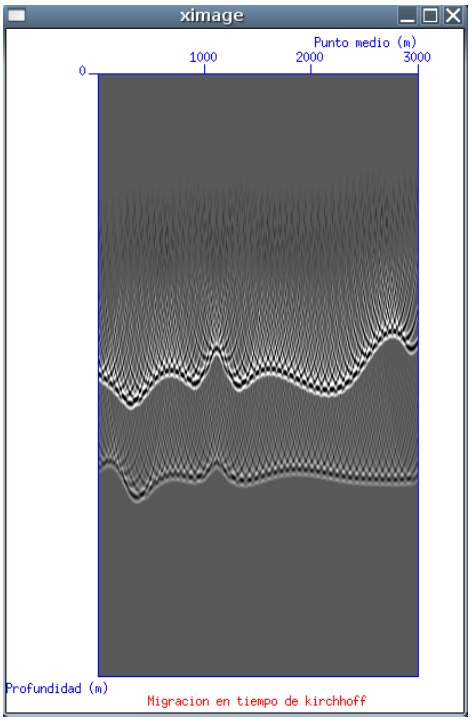

Figura 23.1 Resultado de la migración hecha para 3000 trazas utilizando el algoritmo de migración secuencial de *Seismic Unix*.

Ref.: Autores de este proyecto

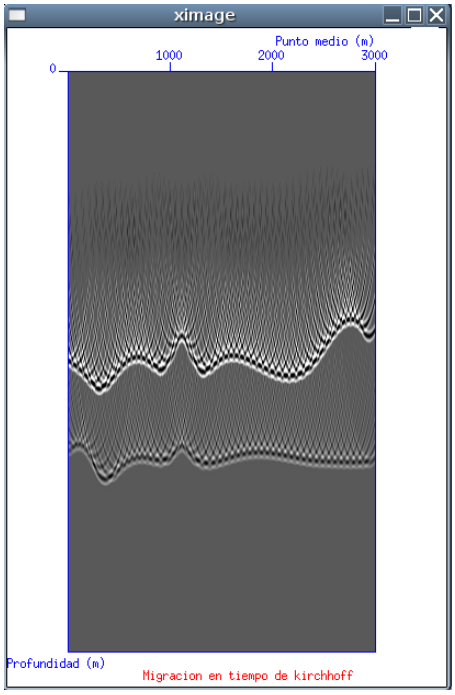

Figura 24.2 Resultado de la migración hecha para 3000 trazas utilizando el algoritmo de migración implementado en paralelo (migración b)

Ref.: Autores de este proyecto

Para comparar los resultados numéricamente; se compara punto a punto la matriz de datos migrados (data) resultante del programa secuencial con la matriz de datos migrados (datap) resultante de cada programa paralelo.

La comparación se hace con la siguiente fórmula:

$$Porcentage[t][k] = 100x\frac{|data[t][k] - datap[t][k]|}{data[t][k]}$$

Y luego se tabula el máximo de Porcentaje hallado como: max(porcentaje).

Se muestra a continuación el error numérico máximo obtenido al comparar punto a punto la matriz de datos migrados utilizando el programa secuencial y los programas paralelos.

Porcentaje máximo de error entre los resultados de la migración secuencial y los programas de migración paralelos.						
Tamaño de los datos de entrada	Porcentaje máximo de error entre la migración secuencial y el programa paralelo sobre el *"clúster"* (migración_b). [%]			Porcentaje máximo de error entre la migración secuencial y el programa paralelo sobre el *"clúster"* (migración_d). [%]		
	3 máquinas	5 máquinas	7 máquinas	3 máquinas	5 máquina	7 máquinas
219 kb. 100 trazas	0.0000000000	0.0000000201	0.0000000000	0.0000000000	0.0000000187	0.0000000000
1,06 Mb. 500 trazas	0.0000000200	0.0000000000	0.0000000015	0.0000000159	0.0000000000	0.0000000004
2,13 Mb. 1000 trazas	0.0000000000	0.0000000021	0.0000000000	0.0000000000	0.0000000008	0.0000000000
3,20 Mb. 1500 trazas	0.0000000202	0.0000000000	0.0000000018	0.0000000160	0.0000000000	0.0000000007
32,1 Mb. 15000 trazas	0.0000000261	0.0000000000	0.0000000073	0.0000000240	0.0000000000	0.0000000033

Tabla 4-2 Porcentaje máximo de error entre los resultados de la migración secuencial y los programas de migración paralelos.

4.1.3 Análisis del uso de red de los algoritmos paralelos implementados sobre el clúster.

Se presenta gráficamente el uso de la red de ambos algoritmos paralelos y finalmente de superponen las gráficas para elaborar un mejor análisis de rendimiento debido al uso de red. Se tratará posteriormente de predecir el punto en el cual cada algoritmo encontraría un cuello de botella en la velocidad de transferencia del *"clúster"* debido al tamaño de los datos ó al número de máquinas del *"clúster"*. Debido a la aplicación, el análisis se hizo en el nodo maestro (master) pues todo el flujo de datos es entre dicho nodo maestro y los demás nodos pero nunca entre dos nodos que no involucren al master. Por lo tanto, es lo mismo hablar de flujo de datos en la red que hablar de flujo de datos en el nodo maestro.

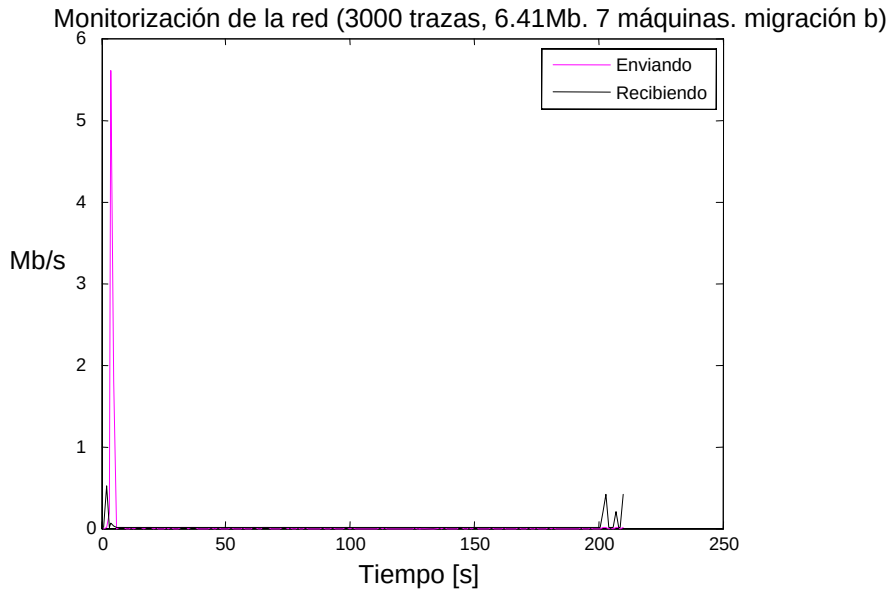

Figura 25. Monitorización de la red (3000 trazas, 6.41Mb. 7 procesos. migración b)
Ref.: Autores de este proyecto

En la Figura 22, Se muestra el flujo de datos por la red local durante toda la ejecución del programa paralelo (migración b) cuando este está migrando datos de entrada de tamaño 6,41 Mb y se está corriendo sobre 7 máquinas. La gráfica de

color fucsia representa el flujo de datos saliente del nodo maestro y la gráfica de color negro representa el flujo de datos entrantes a este nodo.

Datos enviados y Recibidos: Todos los datos enviados desde el nodo maestro no regresan a este nodo. Los datos que se envían desde el nodo maestro en la migración b, son el archivo de registro y el archivo de velocidades. Estos archivos son enviados a cada máquina conectada en el clúster y es por eso que se puede observar un gran flujo de datos enviados al inicio de la ejecución. Para finalizar la ejecución solamente se recibe la matriz de datos migrados, es decir; de cada máquina se recibe una matriz relativamente pequeña que al unirse con las matrices recibidas de las otras máquinas, forma la matriz de datos migrados. Según esto, para la ejecución sobre 7 máquinas debe enviarse el archivo de registro, completo, 6 veces al igual que el archivo de velocidades; mientras que solo se recibe seis veces la sexta parte de la matriz de datos migrados.

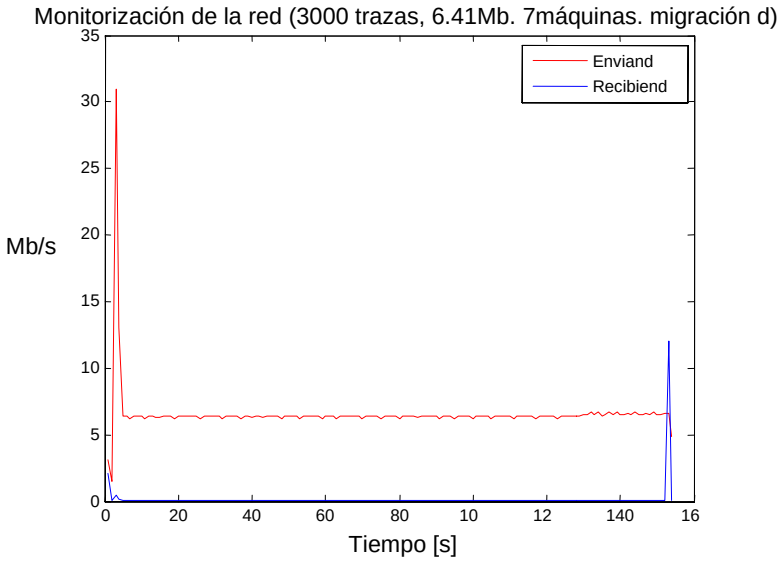

Figura 26 Monitorización de la red (3000 trazas, 6.41Mb. 7 procesos. migración d)
Ref.: Autores de este proyecto

En la Figura 23, Se muestra el flujo de datos por la red local durante toda la ejecución del programa paralelo (migración d) cuando este está migrando datos de entrada de tamaño 6,41 Mb y se está corriendo sobre 7 máquinas. La gráfica de color rojo representa el flujo de datos saliente del nodo maestro y la gráfica de color azul representa el flujo de datos entrantes a este nodo.

La explicación de por qué hay tanta diferencia entre el flujo de datos enviados y el flujo de datos recibidos, es la misma que para la migración b, mostrada en la Figura 22, Con la única diferencia de que al principio sólo se envía el archivo de velocidades pero no el archivo de registros. Dicho archivo de registros se envía procesado por secciones (filtrado) durante todo el proceso.

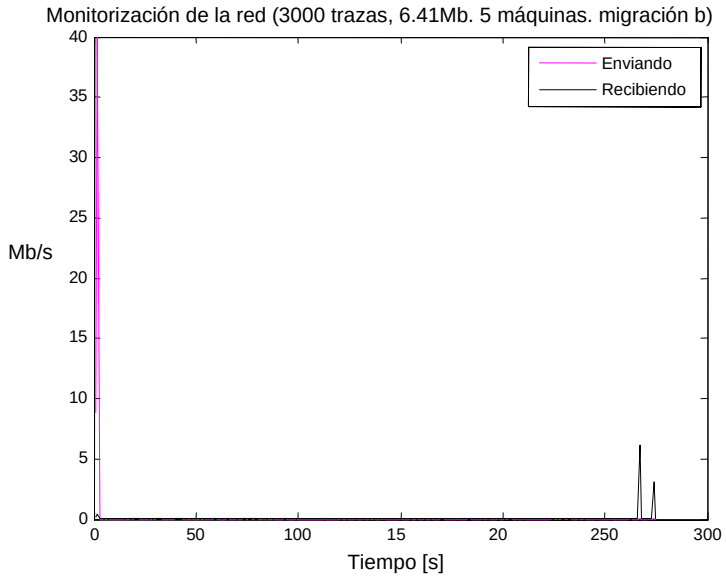

Figura 27 Monitorización de la red (3000 trazas, 6.41Mb. 5 máquinas. migración b)
Ref.: Autores de este proyecto

En la Figura 24, Se muestra el flujo de datos por la red local durante toda la ejecución del programa paralelo (migración b) cuando este está migrando datos de entrada de tamaño 6,41 Mb y se está corriendo sobre 5 máquinas. La gráfica de

color fucsia representa el flujo de datos saliente del nodo maestro y la gráfica de color negro representa el flujo de datos entrantes a este nodo.

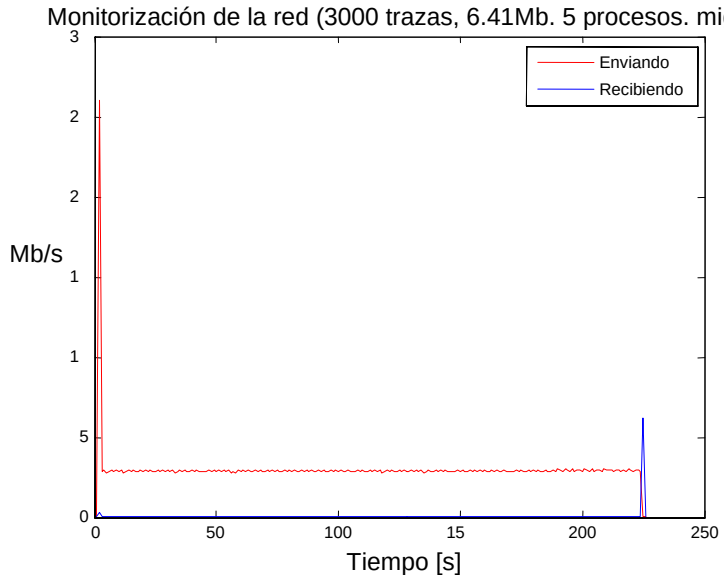

Figura 28 Monitorización de la red (3000 trazas, 6.41Mb. 5 máquinas. migración d)
Ref.: Autores de este proyecto

En la Figura 25, Se muestra el flujo de datos por la red local durante toda la ejecución del programa paralelo (migración d) cuando este está migrando datos de entrada de tamaño 6,41 Mb y se está corriendo sobre 5 máquinas. La gráfica de color rojo representa el flujo de datos saliente del nodo maestro y la gráfica de color azul representa el flujo de datos entrantes a este nodo.

Se puede notar que hay menos flujo de datos salientes durante la ejecución de la migración d sobre 5 máquinas (Figura 25) que sobre 7 máquinas (Figura 23). Esto es debido a que durante el proceso, el nodo maestro debe enviar las matrices de datos filtrados a cada máquina (principal característica del algoritmo migración d) y si hay más máquinas, obviamente, el flujo de datos enviados será mayor.

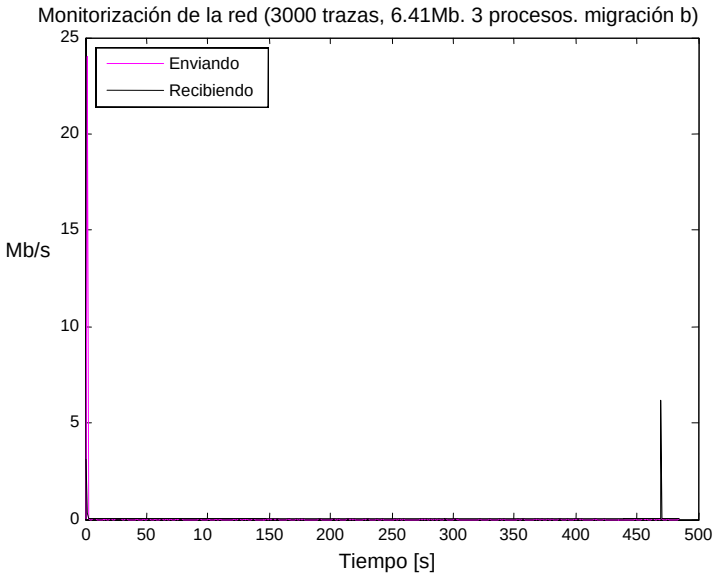

Figura 29 Monitorización de la red (3000 trazas, 6.41Mb. 3 procesos. migración b)
Ref.: Autores de este proyecto

En la Figura 26, Se muestra el flujo de datos por la red local durante toda la ejecución del programa paralelo (migración b) cuando este está migrando datos de entrada de tamaño 6,41 Mb y se está corriendo sobre 3 máquinas. La gráfica de color fucsia representa el flujo de datos saliente del nodo maestro y la gráfica de color negro representa el flujo de datos entrantes a este nodo.

Se puede notar que la migración b no presenta actividad en la red durante la mayor parte del tiempo de ejecución. Esto es debido a que, a diferencia de la migración d, al principio de la ejecución de la migración b se envían los datos necesarios para que cada máquina construya sus propias matrices de datos filtrados, utilizadas para migrar los datos.

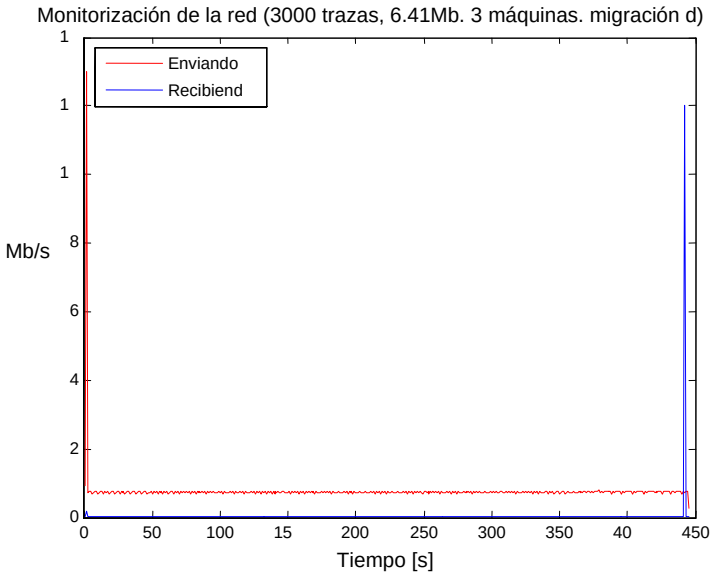

Figura 30 Monitorización de la red (3000 trazas, 6.41Mb. 3 máquinas. migración d)
Ref.: Autores de este proyecto

En la Figura 27, Se muestra el flujo de datos por la red local durante toda la ejecución del programa paralelo (migración d) cuando este está migrando datos de entrada de tamaño 6,41 Mb y se está corriendo sobre 3 máquinas. La gráfica de color rojo representa el flujo de datos saliente del nodo maestro y la gráfica de color azul representa el flujo de datos entrantes a este nodo.

A continuación se superponen las gráficas de uso de red para ambos programas paralelos implementados.

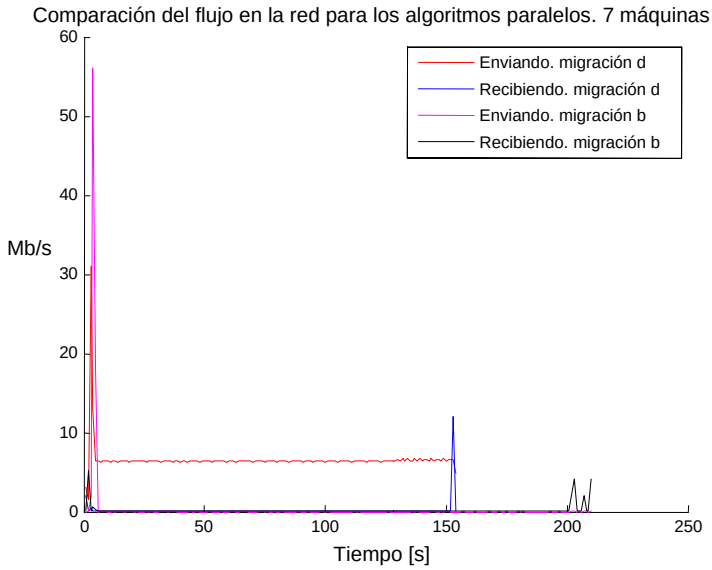

Figura 31 Comparación del flujo en la red para los algoritmos máquinas. 7 procesos
Ref.: Autores de este proyecto

En la Figura 28, se muestra el flujo de datos por la red local durante toda la ejecución de los programas paralelos (migración b y migración d) cuando estos están migrando datos de entrada de tamaño 6,41 Mb y se corren sobre 7 máquinas.

La gráfica de color fucsia representa el flujo de datos saliente del nodo maestro y la gráfica de color negro representa el flujo de datos entrantes a este nodo para la migración b.

La gráfica de color rojo representa el flujo de datos salientes del nodo maestro y la gráfica de color azul representa el flujo de datos entrantes a este nodo para la migración d.

Obsérvese la diferencia de flujo de datos por la red durante el proceso de migración para cada programa. La migración b envía datos al inicio y después de eso cada máquina procesa sus propios filtros. Mientras que la migración d filtra los

datos en el nodo maestro y se mantiene constantemente enviando secciones de datos filtradas hacia las otras máquinas. (Ver sección 3.3)

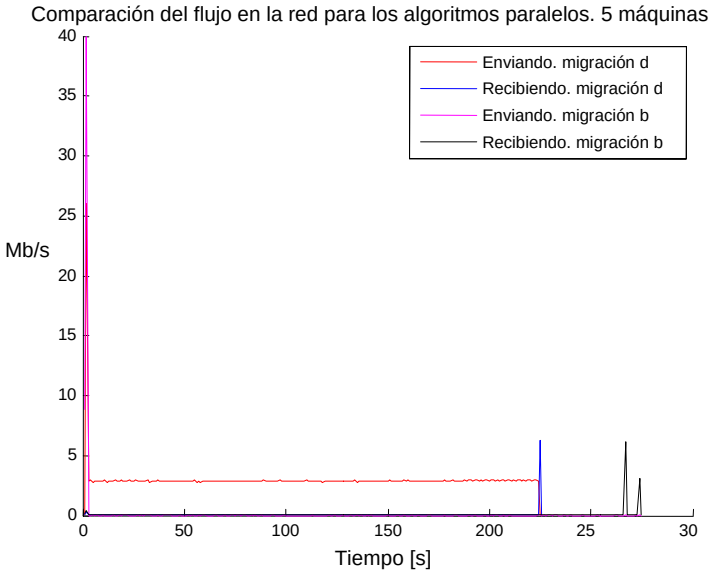

Figura 32 Comparación del flujo en la red para los algoritmos paralelos. 5 máquinas
Ref.: Autores de este proyecto

En la Figura 29, se muestra el flujo de datos por la red local durante toda la ejecución de los programas paralelos (migración b y migración d) cuando estos están migrando datos de entrada de tamaño 6,41 Mb y se corren sobre 5 máquinas.

La gráfica de color fucsia representa el flujo de datos saliente del nodo maestro y la gráfica de color negro representa el flujo de datos entrantes a este nodo durante la ejecución de la migración b.

La gráfica de color rojo representa el flujo de datos saliente del nodo maestro y la gráfica de color azul representa el flujo de datos entrantes a este nodo durante la ejecución de la migración d

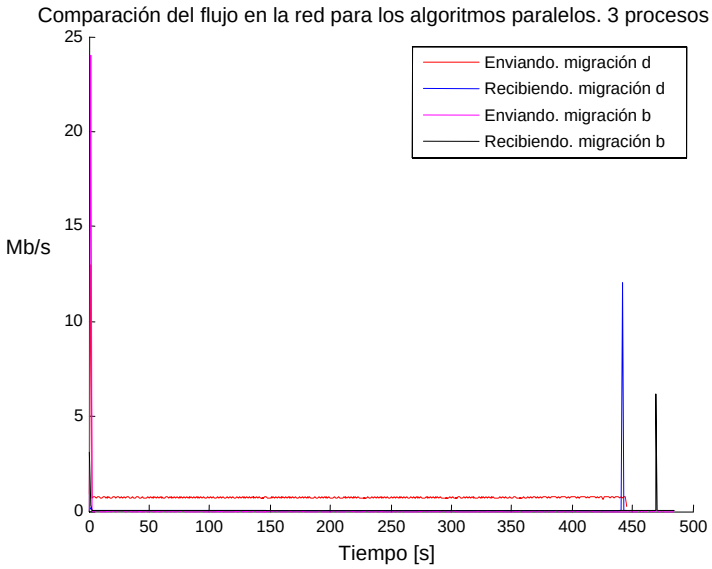

Figura 33 Comparación del flujo en la red para los algoritmos paralelos. 3 máquinas
Ref.: Autores de este proyecto

En la Figura 30, Se muestra el flujo de datos por la red local durante toda la ejecución de los programas paralelos (migración b y migración d) cuando estos están migrando datos de entrada de tamaño 6,41 Mb y se corren sobre 3 máquinas.

La gráfica de color fucsia representa el flujo de datos saliente del nodo maestro y la gráfica de color negro representa el flujo de datos entrantes a este nodo durante la ejecución de la migración b.

La gráfica de color rojo representa el flujo de datos saliente del nodo maestro y la gráfica de color azul representa el flujo de datos entrantes a este nodo durante la ejecución de la migración d.

De nuevo obsérvese la diferencia de flujo de datos por la red durante el proceso de migración para cada programa. La migración b envía datos al inicio y después

de eso cada máquina procesa sus propios filtros. Mientras que la migración d filtra los datos en el nodo maestro y se mantiene constantemente enviando secciones de datos filtradas hacia las otras máquinas (ver sección 3.3)

CONCLUSIONES

El actual desarrollo de la programación esta cada vez más enfocado en la programación en paralelo ya que las exigencias computacionales en los campos de investigación son cada vez mayores.

Este trabajo de tesis de grado presentó aspectos de la programación en paralelo sobre procesadores conectados en un clúster. Dicha programación, consistió en el procesamiento de imágenes sísmicas, específicamente en el proceso de migración sísmica. Se partió de una revisión teórica de las principales características en la generación de una imagen del subsuelo. A continuación se mostraron algunos conceptos de la programación en paralelo y las herramientas que se usaron en la construcción de la imagen y la programación paralela sobre un clúster de procesadores. Se realizó el estudio correspondiente de hardware y se configuró el clúster de de procesadores adecuándolo con las herramientas necesarias.

Las conclusiones obtenidas dentro del proceso de construcción de este documento se pueden enumerar así:

- Se usaron los programas del paquete *Seismic Unix* los cuales constituyen la base más importante de este trabajo ya que brindan una herramienta de gran utilidad en el modelado de datos sísmicos. Este paquete permite tener representaciones del subsuelo partiendo de perfiles de velocidad que son empleados para modelar las ondas de sonido a través de las propiedades físicas del suelo.
- Los principales elementos en la generación de una imagen sísmica son el apilamiento y la migración. El uso de la herramienta *Seismic Unix* gracias a que es un paquete de código abierto ofreció la posibilidad de conocer el

código de migración específicamente el de la migración de Kirchhoff 2D, el cual con ayuda de la librería de paso de mensajes MPI se paralelizó con éxito sobre un clúster de procesadores.

- Se implementó exitosamente el algoritmo de migración de Kirchhoff en 2D en paralelo obteniéndose una mejora en los tiempos de ejecución en comparación con el algoritmo secuencial. Los resultados de la implementación en paralelo fueron validados con los resultados del programa secuencial ya existente en el paquete *Seismic Unix.*
- Se implementaron dos alternativas de algoritmo en paralelo, las dos muestran mejoras en el tiempo de ejecución pero dependiendo de la cantidad de datos que se tenga y de las características de velocidad de transferencia de datos del clúster, una va a mostrar más eficiencia que la otra.
- Se validaron los resultados de los algoritmos paralelos programados, contrastándolos con el algoritmo secuencial de *Seismic Unix.* El error en los resultados es menor que 0.0000001 % y es introducido por el paso de mensajes utilizando las definiciones estándar como por ejemplo MPI_INIT y MPI_FLOAT para evitar discrepancias entre la arquitectura de las máquinas.
- Se compararon tiempos de cómputo del algoritmo secuencial de SU con los algoritmos paralelos implementados en el clúster.
- Con la implementación del algoritmo en paralelo se obtuvieron ahorros de tiempo de más del 80 %. Nótese que se evaluaron los algoritmos paralelos utilizando datos cuyo procesamiento demora aproximadamente 4 horas en el programa secuencial de *Seismic Unix*.
- Ambos algoritmos paralelos implementados lograron aproximadamente el mismo ahorro de tiempo para datos de entrada pequeños pero *migración b* da mejor resultado con datos grandes y con la posibilidad de mejorar aún más cuando se aumente el número de procesadores en el clúster. Esto es

válido hasta cuando el ancho de banda se convierta en el cuello de botella debido a la velocidad del clúster y al tamaño de los datos.

- En los resultados obtenidos Se puede notar que para datos no tan pequeños el tiempo de ejecución de los algoritmos paralelos es aproximadamente (N-1) veces menor que el tiempo de ejecución del algoritmo secuencial para el mismo dato de entrada. N es el número total de máquinas conectadas al clúster y sobre las cuales se corre el algoritmo en paralelo y se le resta 1 (uno) debido a que es la máquina que ejecuta el proceso maestro (master) y la cual solo se encarga de la administración y ordenamiento de los datos procesados por las restantes (N-1) máquinas.
- Cuando el tiempo de ejecución del algoritmo paralelo deje de ser aproximadamente (N-1) veces menor que el tiempo de ejecución del algoritmo secuencial, es porque la velocidad de transferencia de datos por la red local juega un papel importante en el tiempo de ejecución, de tal forma que al compararlo con el tiempo de procesamiento no mejora el tiempo de ejecución del proceso.
- Por el aumento casi lineal del tiempo de ejecución del algoritmo secuencial, según el tamaño de los datos, se puede predecir que para datos de entrada de apenas 180 Mb, generados de forma sintética; dicho algoritmo le tomaría aproximadamente 24 horas migrar los datos, mientras que al algoritmo paralelo *migración b* desarrollado e implementado en este trabajo le tomaría 1 hora y 36 minutos migrar los datos, siempre y cuando esté implementado sobre un clúster de procesadores de características similares al aquí utilizado y con 15 procesadores conectados al clúster. Esto se dedujo teniendo en cuenta que el tiempo de transferencia es irrelevante para dicho algoritmo con el tamaño especificado de los datos, ya que 180 Mb toma menos de 1 s enviarlo de una máquina a la otra, dada las especificaciones de el clúster utilizado.

RECOMENDACIONES

- Cuando se desee utilizar los algoritmos aquí desarrollados se recomienda utilizar clúster de procesadores con nodos dedicados. De esta forma se obtendrá el máximo rendimiento en lo que respecta a disponibilidad del hardware.
- Se hizo poca distinción entre procesos MPI y procesadores del clúster pues los algoritmos se evaluaron ejecutándolos en, máximo, el mismo número de procesos MPI como procesadores se tenían conectadas en el clúster. Esto es necesario para obtener el mejor rendimiento pues ambos algoritmos se hicieron pensando en mantener a todos los procesos ocupados al mismo tiempo, logrando así la mejor distribución de carga y el mayor ahorro de tiempo de ejecución del algoritmo en sí.

- Se recomienda la utilización del programa *migración b* cuando se tenga un clúster con bajas prestaciones de velocidad de transferencia de datos y la utilización del programa *migración d* cuando se cuente con un clúster de alta velocidad de transferencia de datos**.**

BIBLIOGRAFIA

LIBROS

[1] M.Snir, Steve Otto,Steven Huss-Lederman,David Walker,Jack Dongarra,*MPI: The Complete Reference*, **(**2nd Edition),1998.

[2] Charles Bookman*, Linux Clustering: Building and Maintaining Linux Clusters,* Sams,2003

[3] A. Lastovetsky, Tahar Kechadi, Jack Dongarra, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer,Dublin,2008

[4] H. Bauke, S.Mertens *Cluster Computing,* Springer,2006

[5] A. Francisco, *Introducción a la Programación Paralela.*(1ª edición), Thomson Paraninfo, S.A. 2008

[6] L. M. Moreno de Antonio, *Computación paralela y entornos heterogéneos,*(1ª edición), universidad de la laguna,2005

[7] E. M. Macías López, A. S. Sarmiento, *Computación paralela en un clúster LAN-WLAN controlando en tiempo de ejecución la variación del números de procesos,* Universidad de Las Palmas de Gran Canaria.2001

[8] Peter S. Pacheco *parallel programming with MPI* Morgan Kaufmann Publishers Inc.1996

PAGINAS WEB

[1] Stanford Exploration Project, *Basic Earth Imaging (2008), Zero-offset migration*, 15 Junio 2009 < http://sepwww.stanford.edu/sep/prof/bei6.2009.pdf>

[2] Universidad Politécnica de Catalunya, *Procesado de Sísmica de Reflexión superficial en el Complejo Turbidítico de Ainsa (Huesca)*, *Procesado de datos de sísmica de reflexión,* 12 mayo 2009 http://upcommons.upc.edu/pfc/bitstream/2099.1/3404/9/41205-9.pdf

[3] Universidad Politécnica de Catalunya, *Procesado de Sísmica de Reflexión superficial en el Complejo Turbidítico de Ainsa (Huesca)*, *Interpretación geofísica de las secciones sísmicas,* 12 mayo 2009 http://upcommons.upc.edu/pfc/bitstream/2099.1/3404/9/41205-9.pdf

[4] Laboratorio de Sistemas Complejos*, Curso Intensivo MPI* ,15 mayo 2009 http://lsc.dc.uba.ar/courses/curso-int*ensivo-mpi*

[5] *Nodo CES*GA, *Vídeos Curso: Introducción a la programación en MPI*,17 mayo 2009 http://mathematica.nodo.cesga.es/component/option,com_wrapper/Itemid,98

[6] Stanford Exploration Project, *Seismic Unix Help, 15 Julio 2009* **http://sepwww.stanford.edu/oldsep/cliner/files/suhelp/suhelp.html**

ANEXOS

ANEXO 1. INSTALACIÓN DE HERRAMIENTAS NECESARIAS PARA CORRER EL ALGORITMO

1.1 Configuración del *"clúster"* de computadores utilizado

El "*clúster"* se ha configurado de la siguiente manera.

Nombre de la máquina	IP	Dominio	Funciones necesarias	usuario
master	192.168.109.54	redlocal	Servidor, cliente	usuario
maquina01	192.168.109.51	redlocal	Servidor, cliente	usuario
maquina02	192.168.109.52	redlocal	Servidor, cliente	usuario
maquina03	192.168.109.53	redlocal	Servidor, cliente	usuario
maquina05	192.168.109.55	redlocal	Servidor, cliente	usuario
maquina06	192.168.109.56	redlocal	Servidor, cliente	usuario
maquina07	192.168.109.57	redlocal	Servidor, cliente	usuario

Tabla 1-1 Configuración del clúster

Con el sistema Debian lenny instalado se procedo con la instalación de las herramientas de software necesarias para correr el algoritmo.

1.2. Lista de repositorios utilizados para instalar las herramientas de software

Se utiliza la siguiente lista de repositorios con los cuales se instalan todas las herramientas necesarias sin ningún inconveniente.

```
deb cdrom:[Debian GNU/Linux 5.0.0 _Lenny_ - Official i386 DVD Binary-1
20090214-16:54]/ lenny contrib main
## Debian Stable (Lenny)
deb http://ftp.debian.at/debian/ stable main contrib non-free
deb-src http://ftp.debian.at/debian/ stable main contrib non-free
#Actualizaciones de seguridad
deb http://security.debian.org/ stable/updates main contrib
deb-src http://security.debian.org/ stable/updates main contrib
```

1.3 Simbología

1. Los comandos siempre están escritos en letras itálicas
2. Los comandos que tienen un # (numeral) a la izquierda, son ejecutados como supe usuario (root)
3. Los comandos que tienen $ (dólar) a la izquierda, son ejecutados como usuario.

1.4 INSTALANDO *SEISMIC UNIX* EN EL NODO MAESTRO (master)

1. Instalando las dependencias

El primer paso es instalar las dependencias, para eso se ejecuta, como root, lo siguiente:

#apt-get install gcc -y
#apt-get install gfortran -y
#apt-get install lesstif2 -y
#apt-get install make
#apt-get install automake1.9
#apt-get install lesstif2-dev -y
#apt-get install lesstif-bin -y
#apt-get install mesa-utils -y
#apt-get install xlibmesa-glu -y
#apt-get install build-essential -y
#apt-get install libx11-dev -y
#apt-get install libxt-dev -y
#apt-get install libglut3-dev -y
#apt-get install libxmu-dev -y
#apt-get install libxi-dev -y
#apt-get install flex -y
#apt-get install libxaw7-dev f2c -y

2. Descargar y descomprimir el instalador

El código fuente de SU se descarga de la página oficial http://www.cwp.mines.edu/cwpcodes/ y se descomprime en /home/usuario/su. Después de descomprimir queda la ruta /home/usuario/su/src

3. Configurando Makefile.config para adecuarlo a Debian 5.

El archivo Makefile.config que está dentro de la carpeta su ha sido modificado para la configuración de Debian Lenny. Dicho archivo se puede descargar de http://cid-f6f36f2ae47b3d79.skydrive.live.com/self.aspx/SEISMIC/Makefile.config y pegarlo en la carpeta /home/usuario/su/src. Este archivo de configuración ha sido modificado para que SU utilice el compilador gcc en lugar de cc y el compilador gfoltran envés de g77. También usa make gnu y no make old.

4. Preparando las rutas de instalación

Desde un Terminal, ubicarse en la carpeta src, esto se hace con el siguiente comando

~$ cd /home/usuario/su/src

5. Exportar las variables

~$export CWPROOT=/home/usuario/su
~$export PATH=$PATH:/home/usuario/su/bin

6. instalación de la base de SU

Como usuario, se hace make install:

~$make install

Dando como respuesta yes a todo e introduciendo el email cuando este es requerido.

7. Instalación de los componentes de SU

Como usuario, Se ejecutan los siguientes comandos con los cuales quedan instalados los componentes SU:

~$make xtinstall
~$make xminstall
~$make finstall
~$make mglinstall
~$make utils

8. Modificación al .bashrc

Después de instalar SU se edita el archivo .bashrc

~$nano /home/usuario/.bashrc

Al final del dicho archivo se agrega lo siguiente:

```
export CWPROOT=/home/usuario/su
PATH=$PATH:$HOME/bin:$CWPROOT/bin
export PATH
```

Se debe cargar el .bashrc:

~$source ~/.bashrc

9. Verificación de la instalación

La correcta instalación de SU es comprobada ejecutando el siguiente comando en una consola nueva:

~$suplane | suxwigb

El resultado es la Gráfica de la siguiente figura.

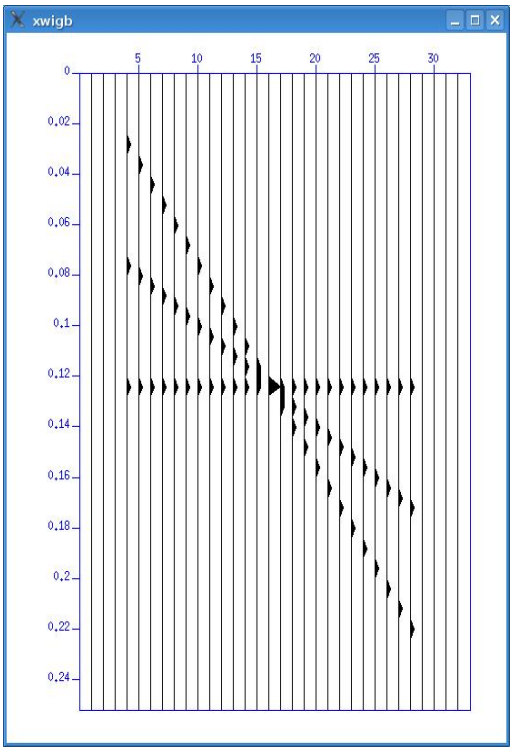

Figura 1-1 Resultado de haber ejecutado el comando *$ suplane | suxwigb* después de instalado el *Seismic Unix* (SU)

1.5. INSTALACION DE NFS

El sistema de archivos en red permite a todo el *"clúster"* de computadores compartir parte de su sistema de archivos. Esto es muy importante para correr código paralelo como lo es MPICH. De este modo una o más máquinas mantienen los archivos en su disco duro físico y actúa como un servidor NFS mientras los

otros nodos “montan” el sistema de archivos de forma local. Para el usuario parecería que los archivos existen en todas las máquinas.

Con NFS pueden existir varios servidores, Los cuales deben ser montados en lugares diferentes (Dos máquinas no pueden compartir su directorio /home y al mismo tiempo tener montado algún otro ahí. Aunque si, una máquina puede compartir /home/estudiantes y alguna otra puede compartir /home/facultad.

Para que MPICH funcione correctamente en el *“clúster”* cada nodo debe hacer las veces de servidor y de cliente por eso se instala los paquetes servidor y cliente de NFS en cada uno. La instalación se hace con todos los nodos conectados y encendidos. A estos, se les ha fijado correctamente la dirección IP y el nombre de usuario debe ser el mismo en todos. Ver tabla 1.

1. Configuración del master

Empezando por el master, se instala nfs-common, nfs-kernel-server y openssh-server; para eso se ejecuta en una consola, como root, lo siguiente

#apt-get install nfs-common nfs-kernel-server openssh-server –y

A continuación se configura el archivo /etc/exports el cual es instalado de forma automática. En este archivo va consignado la parte del sistema de archivos del master que será compartida con los otros nodos. El formato es el siguiente:

<directorio a compartir> <máquinas permitidas>(opciones)

Se puede ver que no hay espacio entre la especificación de las máquinas permitidas y el paréntesis de opciones.

El archivo /etc/exports en el master queda de la siguiente forma:

```
/home/usuario  192.168.109.101(rw) 192.168.109.52(rw)
```

Donde 192.168.109.51y 192.168.109.52 son la IP de los nodos maquina01 y maquina02 respectivamente.

Una vez editado y guardados los cambios del /etc/exports, se reinicia el servidor NFS con el siguiente comando (como root)

#/etc/init.d/nfs-kernel-server restart

De esta forma queda configurado el master.

2. Configuración de los nodos

1. Se procede ahora a instalar los respectivos paquetes en cada nodo:

Los paquetes que se instalan son nfs-common nfs-kernel-server nfs-common openshh-server los cuales son los paquetes de cliente y servidor NFS y se instalan con los siguientes comandos:
#apt-get install nfs-common nfs-kernel-server openssh-server -y

2. Se edita el archivo /etc/fstab de cada nodo. Al final de este archivo se debe se agrega lo siguiente:

#En el /etc/fstab del maquina01:

```
192.168.109.54:/home/usuario     /home/usuario nfs defaults  0   0
```

#En el /etc/fstab del maquina02:

```
192.168.109.54:/home/usuario     /home/usuario nfs defaults  0   0
```

Donde 192.168.109.54 es la IP del master, el primer /home/usuario es la carpeta compartida del master y el segundo /home/usuario es el punto de montaje en cada nodo.

- **Solución de problemas**

Si la carpeta compartida no se monta automáticamente cuando se inicia cada máquina se hace lo siguiente:

- Revisar que la IP del master no haya cambiado desde que esta se fijó en el /etc/fstab de cada nodo y que el master esté encendido antes de iniciar los nodos.

- Cambiar el /etc/fstab en cada nodo de tal forma que quede similar al siguiente:

```
        192.168.109.54:/home/usuario     /home/usuario nfs
```

- Editar el script /etc/init.d/mountnfs.sh en cada nodo. Cambiar:

```
        case "$OPTS" in
        noauto|*,noauto|noauto,*|*,noauto,*)
        continue
        ;;
        esac
```

Por

```
case "$OPTS" in
noauto|*,noauto|noauto,*|*,noauto,*|bg,*|*,bg,*|bg |*,bg)
continue
;;
esac
```

- Finalmente editar /etc/rc.local en cada nodo para incluir las palabras mount /punto/montaje como sigue:

```
#!/bin/sh -e
#
# rc.local
#
# This script is executed at the end of each multiuser runlevel.
# Make sure that the script will "exit 0" on success or any other
# value on error.
#
# In order to enable or disable this script just change the execution
# bits.
#
# By default this script does nothing.
mount /home/usuario
exit 0
```

3. Generar la llave pública sin contraseña para poder hacer ssh entre las máquinas. Este método está hecho sólo para correr MPICH pues se genera la llave pública en una sola máquina y esta es transferida a las demás máquinas en el momento que se monta la carpeta compartida.

Procedimiento:

En el master se ejecuta:

usuario@master:~$ ssh-keygen -t rsa -f ~/.ssh/id_rsa

Cuando pide passphrase , se deja en blanco. A continuación se ejecuta:

usuario@master:~$ eval `ssh-agent -s`
usuario@master:~$ ssh-add
usuario@master:~$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys

En este momento la llave pública está en el master.

4. Desde cada nodo se hace, como root, mount –a

maquina01:

#mount –a

maquina02:

#mount –a

Con esto la carpeta /home/usuario del master queda montada en /home/usuario de cada nodo por lo que es posible hacer ssh de un nodo a cualquier otro sin necesidad de introducir contraseñas, pues la llave pública está en esa carpeta.

5. Se hace ssh desde cada máquina a todos los demás nodos conectados al *"clúster"* pues por ser la primera vez, se debe contestar yes a la condición de agregar al nodo a la lista de hosts conocidos.

Para comprobar la correcta instalación y configuración de NFS adecuado para MPICH, desde una de las máquinas se hace ssh hacia cualquier otra máquina:

usuario@master:~$ ssh 192.168.109.51

Lo anterior es un ssh directo hacia el maquina01, con lo cual se accede a dicho nodo sin digitar contraseña. Se observa en la consola del master lo siguiente:

```
usuario@maquina01:~$
```

La comprobación se hace para cada nodo en el “clúster”.

1.6. INSTALACIÓN DE MPICH.

Después de instalado correctamente el servidor NFS se procede con la instalación de MPICH. Esta instalación se hace en el nodo master.

La siguiente es la manera recomendada de instalar mpich ya que el directorio de instalación es la carpeta de usuario que se comparte con los nodos.

1. Se descarga el instalador de MPICH. Para hacer eso hay que ejecutar (como usuario)

~$cd && wget
http://www.mcs.anl.gov/research/projects/mpi/mpich1/downloads/mpich.tar.gz

2. Descomprimir el archivo

~$tar -xvzf mpich.tar.gz

Verificar haciendo ~$dir para listar los archivos y fijarse en el nombre exacto de la carpeta descomprimida. En este caso fue mpich-1.2.7p1

3. Ubicarse dentro de la carpeta MPICH y ejecutar:

~$cd mpich-1.2.7p1

4. Seguidamente se ejecuta:

~$./configure --with-device=ch_p4 --prefix=/home/usuario/mpich -cc=gcc -fc=gfortran
~$make
~$make install

5. Ahora se modifica el archivo ~/.bashrc

~$nano ~/.bashrc

Donde al final deben ser agregadas las siguientes líneas:

```
export P4_RSHCOMMAND=ssh
PATH=/home/usuario/mpich/bin:$PATH
export PATH
PATH=/home/usuario/mpich/sbin:$PATH
export PATH
```

ctrl + 0 para guardar cambios y luego ctrl + x para salir y finalmente:
~$ source ~/.bashrc

1.6.1 Configurar MPICH.

Master:

MPICH utiliza las máquinas listadas en el archivo /home/usuario/mpich/share/machines.LINUX conocido como fichero de máquina. A continuación se muestra el formato de este archivo.

```
master
maquina01
maquina02 :2
.
.
.
maquina07
```

El maquina02 se lista como maquina02:2 debido a que esta máquina contiene dos procesadores.

Los nombres utilizados en /home/usuario/mpich/share/machines.LINUX deben estar listados también en el archivo /etc/hosts:

```
127.0.0.1        localhost
192.168.109.54   master
192.168.109.51   maquina01
192.168.109.52   maquina02
.
.
.
192.168.109.57   maquina07
```

Nodos:

El archivo /etc/hosts en los nodos debe contener la IP de las máquinas listadas en el archivo machines.LINUX del master:

/etc/hosts maquina01:

```
127.0.0.1       localhost
192.168.109.54     master
192.168.109.51  maquina01
192.168.109.52   maquina02
.
.
.
192.168.109.57   maquina07
```

/etc/hosts maquina02:

```
127.0.0.1       localhost
192.168.109.54     master
192.168.109.51  maquina01
192.168.109.52   maquina02
.
.
.
192.168.109.57   maquina07
```

1.6.3 Probando la correcta instalación de MPICH.

Para esto se ejecuta en el master:

~$tstmachines –v

El resultado es el siguiente.

```
Trying true on master ...
Trying true on maquina01 ...
Trying true on maquina02 ...
.
.
.
Trying ls on master ...
Trying ls on maquina01 ...
Trying ls on maquina02 ...
.
.
.
Trying user program on master ...
Trying user program on maquina01 ...
Trying user program on maquina02 ...
.
.
.
```

Que indica que MPICH está correctamente instalado.

Se puede crear, compilar y correr un programa de ejemplo de la siguiente forma:

Crear un archivo de texto que contenga

```
#include "mpi.h"
#include <stdio.h>
main(int argc, char *argv[])
{
        int myid, npes;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &npes);
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);
```

```
        printf("Soy el proceso %d de %d \n", myid, npes);
        MPI_Finalize();
}
```

Guardarlo con el nombre ejemplo.c en la carpeta /home/usuario.
Ejecutar:

~$ mpicc -o ejemplo ejemplo.c
~$mpirun -np 4 ejemplo

Se obtiene lo siguiente como resultado.

```
Soy el proceso  0 de 4
Soy el proceso  2 de 4
Soy el proceso  1 de 4
Soy el proceso  3 de 4
```

ANEXO 2. EJECUCIÓN DE LA MIGRACIÓN PARALELA (migración b y migración d) en DEBIAN.

Para ejecutar el algoritmo en paralelo, ya sea migración b ó migración d, se debe ejecutar el siguiente script.

```
#! /bin/sh
# shell for uniformly sampling velocity from a layered model
#set -v

WIDTH=400
HEIGHT=600
WIDTHOFF1=10
WIDTHOFF2=430
WIDTHOFF3=860
HEIGHTOFF1=20

#Migración preapilada en tiempo de Kirchhoff

time mpirun -machinefile nodos.LINUX -np NP migracion_b < registro.su
vfile=vfile dx=10 > salida.data

nt=501 dt=0.004 ft=0.0  nx=1000 dx=10 fx=0
suximage < registro.su d1=$dt f1=$ft d2=$dx f2=$fx  perc=99.5\
        label1="Tiempo (seg)" label2="Punto medio (m)" grid1=solid \
        hbox=$HEIGHT wbox=$WIDTH xbox=$WIDTHOFF2 ybox=$HEIGHTOFF1 \
        title="Datos sinteticos" &

suximage < salida.data perc=99.5\
        label1="Profundidad (m)" label2="Punto medio (m)"\
        d1num=500\
        hbox=$HEIGHT wbox=$WIDTH xbox=$WIDTHOFF3 ybox=$HEIGHTOFF1 \
        title="Migración en tiempo de Kirchhoff" &

suxwigb < salida.data style=seismic perc=80 \
        label1="Profundidad (m)" label2="Punto medio (m)"\
        title="Migración en tiempo de Kirchhoff" &


suxcontour < salida.data label1="Profundidad (m)" label2="Punto medio
(m)"\
             title="Migración en tiempo de Kirchhoff" &


exit 0
```

Donde:

Nodos.Linux es el archivo que contiene el mapa de las máquinas conectadas al clúster. Ver ANEXO 1.

NP es el número de procesos que se desean iniciar. Se recomienda, debido al tipo de aplicación, que sea igual o menor al número de máquinas conectadas al clúster.

Registro.su y vfile son los datos de entrada a la migración – dx también es un dato de entrada y es igual al número de trazas contenidas en registro.su.

El número de trazas contenidas en registro.su se puede conocer ejecutando:

Surange <registro.su

Los datos migrados se llaman salida.data y se grafican a continuación de ejecutar el programa de la migración.

ANEXO 3. ALGORITMOS PARALELOS DE MIGRACIÓN DE KIRCHHOFF 2D CON PRE-APILAMIENTO EN TIEMPO.

Se muestra a continuación las secciones de código que emplean tiempo significativo en el proceso secuencial y se muestra la forma como se paralelizaron.

Migración b:

Envía la totalidad del registro de entrada (datos de entrada) desde el master a cada proceso al inicio. Cada proceso realiza sus propios filtrados y devuelve una sección de datos migrados al master el cual se encarga de guardarlos en un solo archivo de salida.

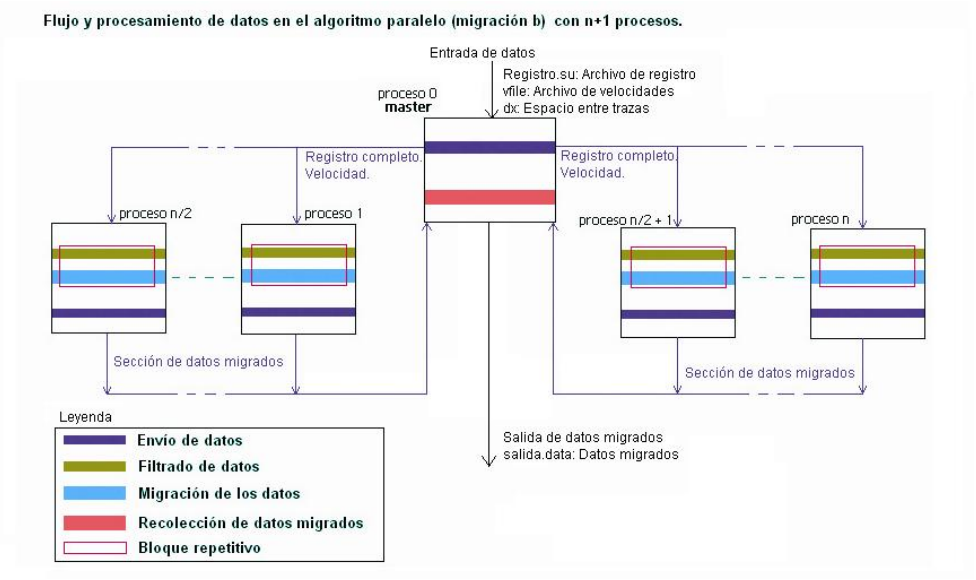

Figura 2-1: Flujo y procesamiento de datos en el algoritmo paralelo (migración b)
Ref.: Autores de este proyecto

Los procesos desde el 1 hasta el n/2 se encargan de hacer la migración para las secciones de datos por la izquierda del *"midpoint":*

```
/*-------------------------------------------------------------------------------------------------*/
/*-----Distribución del proceso para las máquinas desde la 1 hasta la XM (XM=n/2)------------------*/
/*-------------------------------------------------------------------------------------------------*/
for (maquina=1; maquina<=XM; ++maquina) { /*maquinas/procesos a las que es asignada esta rutina*/
       if (rank==maquina){              /*los procesos con rank entre 1 y XM ejecutan esta rutina*/

              int nuevo_it; /*contador*/
              int new_nt;   /*tamaÃ±o del bloque de datos semi-migrados*/

              XM=(numprocs-1)/2;

              MPI_Recv(&ntr, 1, MPI_INT, 0, 300, MPI_COMM_WORLD, &estado);
              MPI_Recv(&dt, 1, MPI_FLOAT, 0, 304, MPI_COMM_WORLD, &estado);
              MPI_Recv(&dx, 1, MPI_FLOAT, 0, 305, MPI_COMM_WORLD, &estado);
              MPI_Recv(&h, 1, MPI_FLOAT, 0, 306, MPI_COMM_WORLD, &estado);
              MPI_Recv(&tmax, 1, MPI_FLOAT, 0, 307, MPI_COMM_WORLD, &estado);
              MPI_Recv(&angmax, 1, MPI_FLOAT, 0, 308, MPI_COMM_WORLD, &estado);
              MPI_Recv(&nc, 1, MPI_INT, 0, 309, MPI_COMM_WORLD, &estado);
              MPI_Recv(&nt, 1, MPI_INT, 0, 380, MPI_COMM_WORLD, &estado);
              MPI_Recv(&ncdp, 1, MPI_INT, 0, 381, MPI_COMM_WORLD, &estado);
              MPI_Recv(&dcdp, 1, MPI_INT, 0, 382, MPI_COMM_WORLD, &estado);
              MPI_Recv(&fcdpdata, 1, MPI_INT, 0, 383, MPI_COMM_WORLD, &estado);

.

.

.

                     /*FILTRO*/
                     for(it=0; it<nt; ++it){
                            rtin[it]=data[imp][it];
                     }

                     for(ifc=1; ifc<nc+1; ++ifc){
                            memset((void *) rtout, 0, nfft*FSIZE);
                            memset((void *) ct, 0, nf*FSIZE);
                            lpfilt(nfc,nfft,dt,fc[ifc],filter);
                            pfarc(1,nfft,rtin,ct);

.

.

.

                      /*procesa las secciones de trazas correspondientes*/
                      for (it=maquina-1;it<nt;it=it+XM){
                             int lx, ux;
                             nuevo_it=nuevo_it+1;

                             t0=it*dt;
                             v=vel[imp*dcdp+fcdpdata-1][it];
                             xmax=tan((angmax+10.0)*PI/180.0)*v*t0;
                             lx=MAX(0,imp - ceil(xmax/dx));
                             ux=MIN(ntr,imp + ceil(xmax/dx));

.

.

.
```

```
                /*Envia el bloque de datos semi-migrados con su respectivo identificador*/
                MPI_Send(mig[0], new_nt*ntr, MPI_FLOAT, 0,1000+rank, MPI_COMM_WORLD);

        }
}

/*-------------------------------------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------------------------------------*/
```

Los procesos desde el n/2+1 hasta el n se encargan de hacer la migración para las secciones de datos por la derecha del *"midpoint":*

```
/*-------------------------------------------------------------------------------------------------------*/
/*-----Distribución del proceso para las máquinas desde la XM+1 hasta la 2XM (XM=n/2)----------------*/
/*-------------------------------------------------------------------------------------------------------*/
for (maquina=XM+1; maquina<=2*XM; ++maquina) {  /*maquinas/procesos a las que es asignada esta rutina*/
        if (rank==maquina){                     /*los procesos con rank entre XM+1 y 2*XM ejecutan esta rutina*/

                int nuevo_it; /*contador*/
                int new_nt;   /*tamaÃ±o del bloque de datos semi-migrados*/

                XM=(numprocs-1)/2;

                MPI_Recv(&ntr, 1, MPI_INT, 0, 300, MPI_COMM_WORLD, &estado);
                MPI_Recv(&dt, 1, MPI_FLOAT, 0, 304, MPI_COMM_WORLD, &estado);
                MPI_Recv(&dx, 1, MPI_FLOAT, 0, 305, MPI_COMM_WORLD, &estado);
                MPI_Recv(&h, 1, MPI_FLOAT, 0, 306, MPI_COMM_WORLD, &estado);
                MPI_Recv(&tmax, 1, MPI_FLOAT, 0, 307, MPI_COMM_WORLD, &estado);
                MPI_Recv(&angmax, 1, MPI_FLOAT, 0, 308, MPI_COMM_WORLD, &estado);
                MPI_Recv(&nc, 1, MPI_INT, 0, 309, MPI_COMM_WORLD, &estado);
                MPI_Recv(&nt, 1, MPI_INT, 0, 380, MPI_COMM_WORLD, &estado);
                MPI_Recv(&ncdp, 1, MPI_INT, 0, 381, MPI_COMM_WORLD, &estado);
                MPI_Recv(&dcdp, 1, MPI_INT, 0, 382, MPI_COMM_WORLD, &estado);
                MPI_Recv(&fcdpdata, 1, MPI_INT, 0, 383, MPI_COMM_WORLD, &estado);

.

.

.

                        /*FILTRO*/
                        for(it=0; it<nt; ++it){
                                rtin[it]=data[imp][it];
                        }

                        for(ifc=1; ifc<nc+1; ++ifc){
                                memset((void *) rtout, 0, nfft*FSIZE);
                                memset((void *) ct, 0, nf*FSIZE);
                                lpfilt(nfc,nfft,dt,fc[ifc],filter);
                                pfarc(1,nfft,rtin,ct);

.

.

.
```

```
                        /*procesa las secciones de trazas correspondientes*/
                        for (it=maquina-XM-1;it<nt;it=it+XM){
                                int lx, ux;
                                nuevo_it=nuevo_it+1;

                                t0=it*dt;
                                v=vel[imp*dcdp+fcdpdata-1][it];

.

.

.

                /*Envia el bloque de datos semi-migrados con su respectivo identificador*/
                MPI_Send(mig[0], new_nt*ntr, MPI_FLOAT, 0,1000+rank, MPI_COMM_WORLD);

        }
}

/*--------------------------------------------------------------------------------------------------------------------*/
/*--------------------------------------------------------------------------------------------------------------------*/
/*--------------------------------------------------------------------------------------------------------------------*/
```

Migración d: Este a diferencia del anterior no envía la totalidad de los datos a cada proceso sino que envía las secciones de datos que le corresponderá procesar a cada proceso. En consecuencia, es necesario que el master realice los filtros que le corresponderían a cada uno de los otros procesos.

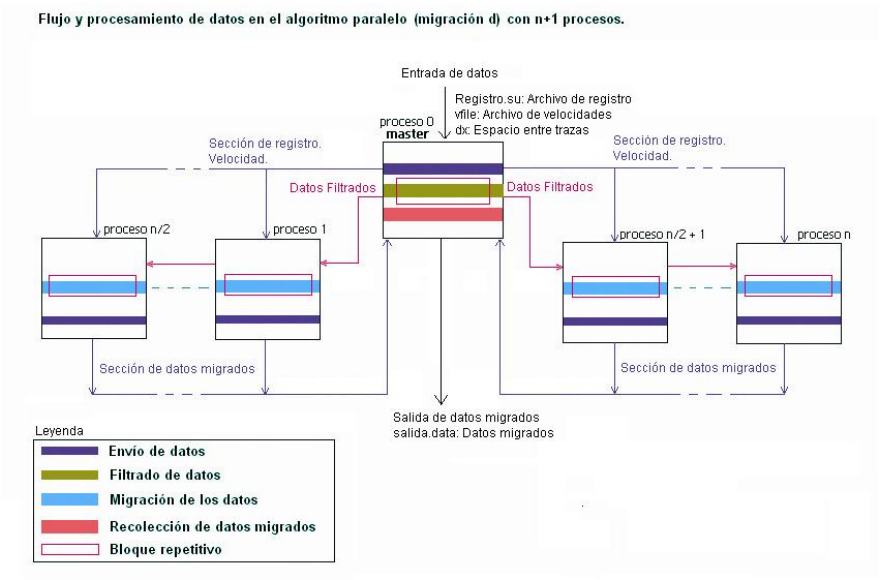

Figura 2-2: Flujo y procesamiento de datos en el algoritmo paralelo (migración d)

Ref.: Autores de este proyecto

Los procesos desde el 1 hasta el n/2 se encargan de hacer la migración para las secciones de datos por la izquierda del *"midpoint":*

```
/*-------------------------------------------------------------------------------------------------*/
/*-----Distribución del proceso para las máquinas desde la 1 hasta la XM (XM=n/2)------------------*/
/*-------------------------------------------------------------------------------------------------*/
for (maquina=1; maquina<=XM; ++maquina) { /*maquinas/procesos a las que es asignada esta rutina*/
        if (rank==maquina){              /*los procesos con rank entre 1 y XM ejecutan esta rutina*/

                int nuevo_it; /*contador*/
                int new_nt;   /*tamaÃ±o del bloque de datos semi-migrados*/

                XM=(numprocs-1)/2;

                MPI_Recv(&ntr, 1, MPI_INT, 0, 300, MPI_COMM_WORLD, &estado);
                MPI_Recv(&dt, 1, MPI_FLOAT, 0, 304, MPI_COMM_WORLD, &estado);
                MPI_Recv(&dx, 1, MPI_FLOAT, 0, 305, MPI_COMM_WORLD, &estado);
                MPI_Recv(&h, 1, MPI_FLOAT, 0, 306, MPI_COMM_WORLD, &estado);
                MPI_Recv(&tmax, 1, MPI_FLOAT, 0, 307, MPI_COMM_WORLD, &estado);
                MPI_Recv(&angmax, 1, MPI_FLOAT, 0, 308, MPI_COMM_WORLD, &estado);
                MPI_Recv(&nc, 1, MPI_INT, 0, 309, MPI_COMM_WORLD, &estado);
                MPI_Recv(&nt, 1, MPI_INT, 0, 380, MPI_COMM_WORLD, &estado);
                MPI_Recv(&ncdp, 1, MPI_INT, 0, 381, MPI_COMM_WORLD, &estado);
                MPI_Recv(&dcdp, 1, MPI_INT, 0, 382, MPI_COMM_WORLD, &estado);
                MPI_Recv(&fcdpdata, 1, MPI_INT, 0, 383, MPI_COMM_WORLD, &estado);

.

.

.

                        /*Recibe el filtro calculado para el actual valor de imp*/
                        MPI_Recv(lowpass[0], nt*(nc+1), MPI_FLOAT, 0, 5000+imp, MPI_COMM_WORLD, &estado_mig);

.

.

.

                                /*procesa las secciones de trazas correspondientes*/
                                for (it=maquina-1;it<nt;it=it+XM){
                                        int lx, ux;
                                        nuevo_it=nuevo_it+1;

                                        t0=it*dt;
                                        v=vel[imp*dcdp+fcdpdata-1][it];
                                        xmax=tan((angmax+10.0)*PI/180.0)*v*t0;
                                        lx=MAX(0,imp - ceil(xmax/dx));
                                        ux=MIN(ntr,imp + ceil(xmax/dx));

.

.

.
```

```
                /*Envia el bloque de datos semi-migrados con su respectivo identificador*/
                MPI_Send(mig[0], new_nt*ntr, MPI_FLOAT, 0,1000+rank, MPI_COMM_WORLD);

        }
}

/*-------------------------------------------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------------------------------------------*/
```

Los procesos desde el n/2+1 hasta el n se encargan de hacer la migración para las secciones de datos por encima del *"midpoint"* :

```
/*-------------------------------------------------------------------------------------------------------------*/
/*-----Distribución del proceso para las máquinas desde la XM+1 hasta la 2XM (XM=n/2)----------------*/
/*-------------------------------------------------------------------------------------------------------------*/
for (maquina=XM+1; maquina<=2*XM; ++maquina) {  /*maquinas/procesos a las que es asignada esta rutina*/
        if (rank==maquina){                     /*los procesos con rank entre XM+1 y 2*XM ejecutan esta rutina*/

                int nuevo_it; /*contador*/
                int new_nt;   /*tamaÃ±o del bloque de datos semi-migrados*/

                XM=(numprocs-1)/2;

                MPI_Recv(&ntr, 1, MPI_INT, 0, 300, MPI_COMM_WORLD, &estado);
                MPI_Recv(&dt, 1, MPI_FLOAT, 0, 304, MPI_COMM_WORLD, &estado);
                MPI_Recv(&dx, 1, MPI_FLOAT, 0, 305, MPI_COMM_WORLD, &estado);
                MPI_Recv(&h, 1, MPI_FLOAT, 0, 306, MPI_COMM_WORLD, &estado);
                MPI_Recv(&tmax, 1, MPI_FLOAT, 0, 307, MPI_COMM_WORLD, &estado);
                MPI_Recv(&angmax, 1, MPI_FLOAT, 0, 308, MPI_COMM_WORLD, &estado);
                MPI_Recv(&nc, 1, MPI_INT, 0, 309, MPI_COMM_WORLD, &estado);
                MPI_Recv(&nt, 1, MPI_INT, 0, 380, MPI_COMM_WORLD, &estado);
                MPI_Recv(&ncdp, 1, MPI_INT, 0, 381, MPI_COMM_WORLD, &estado);
                MPI_Recv(&dcdp, 1, MPI_INT, 0, 382, MPI_COMM_WORLD, &estado);
                MPI_Recv(&fcdpdata, 1, MPI_INT, 0, 383, MPI_COMM_WORLD, &estado);

.

.

.

                        /*Recibe el filtro calculado para el actual valor de imp*/
                        MPI_Recv(lowpass[0], nt*(nc+1), MPI_FLOAT, 0, 5000+imp, MPI_COMM_WORLD, &estado_mig);

.

.

.

                        /*procesa las secciones de trazas correspondientes*/
                        for (it=maquina-XM-1;it<nt;it=it+XM){
                                int lx, ux;
                                nuevo_it=nuevo_it+1;

                                t0=it*dt;
                                v=vel[imp*dcdp+fcdpdata-1][it];

.

.
```

```
.
                /*Envia el bloque de datos semi-migrados con su respectivo identificador*/
                MPI_Send(mig[0], new_nt*ntr, MPI_FLOAT, 0,1000+rank, MPI_COMM_WORLD);

        }
}

/*--------------------------------------------------------------------------------------------------------------*/
/*--------------------------------------------------------------------------------------------------------------*/
/*--------------------------------------------------------------------------------------------------------------*/
```

El filtro se aplica en el master y se envían los datos a los de más procesos:

```
/* Filtro */
for (imp=0; imp<ntr; ++imp){

                for(it=0; it<nt; ++it){
                        rtin[it]=data[imp][it];
                }

                for(ifc=1; ifc<nc+1; ++ifc){
                        memset((void *) rtout, 0, nfft*FSIZE);
                        memset((void *) ct, 0, nf*FSIZE);
                        lpfilt(nfc,nfft,dt,fc[ifc],filter);
                        pfarc(1,nfft,rtin,ct);

                        for(it=0; it<nf; ++it){
                                ct[it] = crmul(ct[it],filter[it]);
                        }
                        pfacr(-1,nfft,ct,rtout);
                        for(it=0; it<nt; ++it){
                                lowpass[ifc][it]= rtout[it];
                        }
                }

                /*Envio de los datos filtrados  para la traza actual*/
                for (until=1;until<=2*XM;++until){
                        MPI_Send(lowpass[0], nt*(nc+1), MPI_FLOAT, until, 5000+imp, MPI_COMM_WORLD);
                }
}
```

Para detallar las rutinas distribuidas en los diferentes procesos se presenta el código fuente de la migración d y se resaltan los comentarios más relevantes:

Nota: Se han quitado las tabulaciones por motivo de edición. Todos los códigos fuentes pueden descargarse de

http://cid-f6f36f2ae47b3d79.skydrive.live.com/browse.aspx/SEISMIC?uc=2

```
/*
PARA COMPILAR ESTE PROGRAMA SE DEBEN TENER LAS HERRAMIENTAS INDICADAS
INSTALADAS (VER ANEXO 1)

Y EJECUTAR:

~$mpicc -o migracion migracion.d -I/home/USUARIO/su/include -O3 -Wall -
Wno-long-long -ansi -pedantic -D_POSIX_SOURCE -D_FILE_OFFSET_BITS=64 -
D_LARGEFILE_SOURCE -D_LARGEFILE64_SOURCE -DGNU_SOURCE -DCWP_LITTLE_ENDIAN
-L/home/USUARIO/su/lib -lsu -lpar -lcwp -lm

POR:
GERARDO E. TERAN. J
JULIAN PITA E.

UNIVERSIDAD INDUSTRIAL DE SANTANDER 2009
----------------------------------------

Basado en suktmig2d de “Colorado School of Mines”, 2008.*/



#include "su.h"
#include "segy.h"
#include "header.h"

#include "mpi.h"
#include <stdio.h>


/********************** documentación **********************/
char *sdoc[] = {"migracion - prestack time migration of a common-offset
section with",
"the double-square root (DSR) operator"," Universidad Industrial de
Santander",NULL};

#define LOOKFAC 2       /* Look ahead factor for npfaro   */
#define PFA_MAX 720720  /* Largest allowed nfft           */


/* Prototype of functions used internally */
void lpfilt(int nfc, int nfft, float dt, float fhi, float *filter);

segy intrace; /* input traces */
segy outtrace;/* migrated output traces */

int
main(int argc, char **argv)
{
int k,imp,iip,it,ix,ifc;/* counters */
int ntr,nt;/* x,t */
```

```
int verbose;/* is verbose?*/
int nc;/* number of low-pass filtered versions*/
/*  of the data for antialiasing*/
int nfft,nf;/* number of frequencies*/
int nfc;/* number of Fourier coefficients for low-pass filter */
int fwidth;/* high-end frequency increment for the low-pass */
/* filters */
int firstcdp=0;/* first cdp in velocity file*/
int lastcdp=0;/* last cdp in velocity file*/
int oldcdp=0;/* temporary storage*/
int fcdpdata=0;/* first cdp in the data*/
int olddeltacdp=0;
int deltacdp;
int ncdp=0;/* number of cdps in the velocity file*/
int dcdp=0;/* number of cdps between consecutive traces */

float dx=0.0;/* cdp sample interval */
float hoffset=0.0;  /* half receiver-source */
float p=0.0;/* horizontal slowness of the migration operator */
float pmin=0.0;/* maximum horizontal slowness for which there's */
/* no aliasing of the operator */
float dt;/* t sample interval */
float h;/* offset */
float x;/* aperture distance */
float xmax=0.0;/* maximum aperture distance */

float obliq;/* obliquity factor */
float geoms;/* geometrical spreading factor */
float angmax;   /* maximum aperture angle */

float mp,ip;/* mid-point and image-point coordinates */
float t;/* time */
float t0;/* vertical traveltime */
float tmax;/* maximum time */

float fnyq;/* Nyquist frequency */
float ang;/* aperture angle */
float angtaper=0.0;/* aperture-angle taper */
float v;/* velocity */

float *fc=NULL;/* cut-frequencies for low-pass filters */
/*float *filter=NULL;*//* array of low-pass filter values */

float **vel=NULL;/* array of velocity values from vfile */
float **data=NULL;/* input data array*/
float **lowpass=NULL;   /* low-pass filtered version of the trace */
float **mig=NULL;/* output migrated data array */

/*register float *rtin=NULL,*rtout=NULL; *//* real traces */
/*register complex *ct=NULL; */  /* complex trace */

/* file names */
char *vfile="";/* name of velocity file */
FILE *vfp=NULL;
```

```
FILE *tracefp=NULL;/* temp file to hold traces*/
FILE *hfp=NULL;/* temp file to hold trace headers */

/*FILTRO*/
register complex *ct=NULL;   /* complex trace */
float *filter=NULL;
register float *rtout=NULL, *rtin=NULL;
int until;

float datalo[8], datahi[8];
int itb, ite;
float firstt, amplo, amphi;



int XM; /*Numero/2 de procesos encargados de la migracion*/
int maquina;/*Contador desde la máquina 1 hasta la 2*XM para repartir los
procesos*/
int numprocs; /* Número total de procesos*/
int rank;/*Numero del proceso actual*/
int namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME]; /*Nombre de la máquina del
actual proceso*/

/*MEDICION DE TIEMPOS*/
double tiempo_mig;


/*VARIABLES DE ESTADO Irecv e Isend de MPI UTILIZADAS POR LOS DIFERENTES
PROCESOS*/
MPI_Request reqs_right;
 MPI_Request reqs_gat_1;
MPI_Request reqs_gat_2;


/*VARIABLES DE ESTADO MPI UTILIZADAS POR LOS PROCESOS*/
MPI_Status estado;
MPI_Status estado3;
MPI_Status estado4;
MPI_Status estado_mig;
MPI_Status estado_mig3;


cwp_Bool check_cdp=cwp_false;/* check cdp in velocity file*/


  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs); /* Número de procesos
totales*/
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);/*Número del proceso actual*/
  MPI_Get_processor_name(processor_name, &namelen); /*nombre de la
mÃ¡quina donde se ejecuta el proceso actual*/

/* Hook up getpar to handle the parameters */
```

```
initargs(argc,argv);
requestdoc(0);


XM=(numprocs-1)/2;


/*RUTINA EJECUTADA EN LAS DIFERENTES MÁQUINAS. (Primera mitad del total
de máquinas sin incluir el master)*/

/*Distribuye: loop over output image-points to the left of the midpoint*/
/*Entre los diferentes procesos. Para cada proceso la variable Rank tiene
un valor diferente*/

for (maquina=1; maquina<=XM; ++maquina) { /*maquinas/procesos a las que
es asignada esta rutina*/
if (rank==maquina){  /*los procesos con rank entre 1 y XM ejecutan esta
rutina*/

int nuevo_it; /*contador*/
int new_nt;   /*tamaño del bloque de datos semi-migrados*/

XM=(numprocs-1)/2;

MPI_Recv(&ntr, 1, MPI_INT, 0, 300, MPI_COMM_WORLD, &estado);
MPI_Recv(&dt, 1, MPI_FLOAT, 0, 304, MPI_COMM_WORLD, &estado);
MPI_Recv(&dx, 1, MPI_FLOAT, 0, 305, MPI_COMM_WORLD, &estado);
MPI_Recv(&h, 1, MPI_FLOAT, 0, 306, MPI_COMM_WORLD, &estado);
MPI_Recv(&tmax, 1, MPI_FLOAT, 0, 307, MPI_COMM_WORLD, &estado);
MPI_Recv(&angmax, 1, MPI_FLOAT, 0, 308, MPI_COMM_WORLD, &estado);
MPI_Recv(&nc, 1, MPI_INT, 0, 309, MPI_COMM_WORLD, &estado);
MPI_Recv(&nt, 1, MPI_INT, 0, 380, MPI_COMM_WORLD, &estado);
MPI_Recv(&ncdp, 1, MPI_INT, 0, 381, MPI_COMM_WORLD, &estado);
MPI_Recv(&dcdp, 1, MPI_INT, 0, 382, MPI_COMM_WORLD, &estado);
MPI_Recv(&fcdpdata, 1, MPI_INT, 0, 383, MPI_COMM_WORLD, &estado);

fnyq=1.0/(2*dt);

new_nt=ceil(nt/XM);

/*Reserva memoria*/
mig = alloc2float(new_nt,ntr);
lowpass = alloc2float(nt,nc+1);
vel = alloc2float(nt,ncdp);

/*Cero todos los puntos del bloque de datos semi-migrados*/
memset((void *) mig[0], 0,new_nt*ntr*FSIZE);

/*Recibe la matriz de velocidades*/
MPI_Recv(vel[0], nt*ncdp, MPI_FLOAT, 0, 31, MPI_COMM_WORLD, &estado3);


/*tiempo1=MPI_Wtime();*/
```

```
for (imp=0;imp<ntr;++imp){

nuevo_it=-1;
mp=imp*dx;

/*Recibe el filtro calculado para el actual valor de imp*/
MPI_Recv(lowpass[0], nt*(nc+1), MPI_FLOAT, 0, 5000+imp, MPI_COMM_WORLD,
&estado_mig3);

/*Aquí se procesan las secciones de trazas correspondientes
```

Si se compara esta rutina con la empleada en el programa secuencial se puede observar que en la secuencial el bucle es `for (it=0;it<nt;++it)` **Con lo cual se analiza una sección de cada traza a la vez mientras que en este algoritmo paralelo esta rutina es ejecutada con incrementos de valor XM= [(número de máquinas dedicadas a la migración)/2]. Esta rutina se ejecuta en la mitad de las máquinas dedicadas a la migración y cada máquina empieza en la siguiente sección de traza diferente de las otras máquinas**

Note la diferencia:

Secuencial: for (it=0;it<nt;++it);
Paralelo: for (it=maquina-1;it<nt;it=it+XM); */

```
for (it=maquina-1;it<nt;it=it+XM){
int lx, ux;
nuevo_it=nuevo_it+1;

t0=it*dt;
v=vel[imp*dcdp+fcdpdata-1][it];
xmax=tan((angmax+10.0)*PI/180.0)*v*t0;
lx=MAX(0,imp - ceil(xmax/dx));
ux=MIN(ntr,imp + ceil(xmax/dx));

/* loop over output image-points to the left of the midpoint */
for(iip=imp; iip>lx;--iip){
float ts,tr;
int fplo=0, fphi=0;
float ref,wlo,whi;

ip=iip*dx;
x=ip-mp;
ts=sqrt( pow(t0/2,2) + pow((x+h)/v,2) );
tr=sqrt( pow(t0/2,2) + pow((h-x)/v,2) );
t= ts + tr;
```

```
if(t>=tmax) break;
geoms=sqrt(1/(t*v));
  obliq=sqrt(.5*(1 + (t0*t0/(4*ts*tr))
- (1/(ts*tr))*sqrt(ts*ts - t0*t0/4)*sqrt(tr*tr - t0*t0/4)));
  ang=180.0*fabs(acos(t0/t))/PI;
  if(ang<=angmax) angtaper=1.0;
  if(ang>angmax) angtaper=cos((ang-angmax)*PI/20);

  /* Evaluate migration operator slowness p to determine */
/* the low-pass filtered trace for antialiasing */
pmin=1/(2*dx*fnyq);
p=fabs((x+h)/(pow(v,2)*ts) + (x-h)/(pow(v,2)*tr));
if(p>0){fplo=floor(nc*pmin/p);}
if(p==0){fplo=nc;}
ref=fmod(nc*pmin,p);
wlo=1-ref;
fphi=++fplo;
whi=ref;
itb=MAX(ceil(t/dt)-3,0);
ite=MIN(itb+8,nt);
firstt=(itb-1)*dt;

/* Move energy from CMP to CIP */
if(fplo>=nc){
for(k=itb; k<ite; ++k){
datalo[k-itb]=lowpass[nc][k];
}
ints8r(8,dt,firstt,datalo,0.0,0.0,1,&t,&amplo);
mig[iip][nuevo_it] +=geoms*obliq*angtaper*amplo;
} else if(fplo<nc){
for(k=itb; k<ite; ++k){
datalo[k-itb]=lowpass[fplo][k];
datahi[k-itb]=lowpass[fphi][k];
}
ints8r(8,dt,firstt,datalo,0.0,0.0,1,&t,&amplo);
ints8r(8,dt,firstt,datahi,0.0,0.0,1,&t,&amphi);
mig[iip][nuevo_it] += geoms*obliq*angtaper*(wlo*amplo + whi*amphi);
}

}

}
}

/*Envía el bloque de datos semi-migrados con su respectivo indicador
MPI_Send(mig[0], new_nt*ntr, MPI_FLOAT, 0,1000+rank, MPI_COMM_WORLD);

}
}


/*--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
---------------*/
```

```
/*RUTINA EJECUTADA EN LAS DIFERENTES MÁQUINAS. (Segunda mitad del total de máquinas sin incluir el master)*/
/*Distribuye la rutina entre los diferentes procesos. Para cada proceso la variable Rank tiene un valor diferente*/

for (maquina=XM+1; maquina<=2*XM; ++maquina) {/*maquinas/procesos a las que es asignada esta rutina*/
if (rank==maquina){/*los procesos con rank entre XM+1 y 2*XM ejecutan esta rutina*/

int nuevo_it; /*contador*/
int new_nt;   /*tamaño del bloque de datos semi-migrados*/

XM=(numprocs-1)/2;

MPI_Recv(&ntr, 1, MPI_INT, 0, 300, MPI_COMM_WORLD, &estado);
MPI_Recv(&dt, 1, MPI_FLOAT, 0, 304, MPI_COMM_WORLD, &estado);
MPI_Recv(&dx, 1, MPI_FLOAT, 0, 305, MPI_COMM_WORLD, &estado);
MPI_Recv(&h, 1, MPI_FLOAT, 0, 306, MPI_COMM_WORLD, &estado);
MPI_Recv(&tmax, 1, MPI_FLOAT, 0, 307, MPI_COMM_WORLD, &estado);
MPI_Recv(&angmax, 1, MPI_FLOAT, 0, 308, MPI_COMM_WORLD, &estado);
MPI_Recv(&nc, 1, MPI_INT, 0, 309, MPI_COMM_WORLD, &estado);
MPI_Recv(&nt, 1, MPI_INT, 0, 380, MPI_COMM_WORLD, &estado);
MPI_Recv(&ncdp, 1, MPI_INT, 0, 381, MPI_COMM_WORLD, &estado);
MPI_Recv(&dcdp, 1, MPI_INT, 0, 382, MPI_COMM_WORLD, &estado);
MPI_Recv(&fcdpdata, 1, MPI_INT, 0, 383, MPI_COMM_WORLD, &estado);

fnyq=1.0/(2*dt);

new_nt=ceil(nt/XM);

/*Reserva memoria*/
mig = alloc2float(new_nt,ntr);
lowpass = alloc2float(nt,nc+1);
vel = alloc2float(nt,ncdp);

/*Cero todos los puntos del bloque de datos semi-migrados*/
memset((void *) mig[0], 0,new_nt*ntr*FSIZE);

/*Recibe la matriz de velocidades*/
MPI_Recv(vel[0], nt*ncdp, MPI_FLOAT, 0, 31, MPI_COMM_WORLD, &estado3);

for (imp=0;imp<ntr;++imp){

nuevo_it=-1;
mp=imp*dx;

/*Recibe el filtro calculado para el actual valor de imp*/
```

```
MPI_Recv(lowpass[0], nt*(nc+1), MPI_FLOAT, 0, 5000+imp, MPI_COMM_WORLD,
&estado_mig);

/* Si se compara esta rutina con la empleada en el programa secuencial se
puede observar que en la secuencial el bucle es for (it=0;it<nt;++it) Con
lo cual se analiza una sección de cada traza a la vez mientras que en
este algoritmo paralelo esta rutina es ejecutada con incrementos de valor
XM= [(número de máquinas dedicadas a la migración)/2]. Esta rutina se
ejecuta en la mitad de las máquinas dedicadas a la migración y cada
máquina empieza en la siguiente sección de traza diferente de las otras
máquinas*/

/*Note la diferencia:

Secuencial: for (it=0;it<nt;++it);
Paralelo: for (it=maquina-XM-1;it<nt;it=it+XM); */

for (it=maquina-XM-1;it<nt;it=it+XM){
int lx, ux;
nuevo_it=nuevo_it+1;

t0=it*dt;
v=vel[imp*dcdp+fcdpdata-1][it];
xmax=tan((angmax+10.0)*PI/180.0)*v*t0;
lx=MAX(0,imp - ceil(xmax/dx));
ux=MIN(ntr,imp + ceil(xmax/dx));

/* loop over output image-points to the right of the midpoint */
for(iip=imp+1; iip<ux; ++iip){
float ts,tr;
int fplo=0, fphi;
float ref,wlo,whi;

ip=iip*dx;
x=ip-mp;
t0=it*dt;
ts=sqrt( pow(t0/2,2) + pow((x+h)/v,2) );
tr=sqrt( pow(t0/2,2) + pow((h-x)/v,2) );
t= ts + tr;
if(t>=tmax) break;
geoms=sqrt(1/(t*v));
obliq=sqrt(.5*(1 + (t0*t0/(4*ts*tr))
- (1/(ts*tr))*sqrt(ts*ts
- t0*t0/4)*sqrt(tr*tr
- t0*t0/4)));
ang=180.0*fabs(acos(t0/t))/PI;
if(ang<=angmax) angtaper=1.0;
```

```
if(ang>angmax) angtaper=cos((ang-angmax)*PI/20.0);

/* Evaluate migration operator slowness p to determine the  */
/* low-pass filtered trace for antialiasing */
pmin=1/(2*dx*fnyq);
p=fabs((x+h)/(pow(v,2)*ts) + (x-h)/(pow(v,2)*tr));
if(p>0){
fplo=floor(nc*pmin/p);
}
if(p==0){
fplo=nc;
}

ref=fmod(nc*pmin,p);
wlo=1-ref;
fphi=fplo+1;
whi=ref;
itb=MAX(ceil(t/dt)-3,0);
ite=MIN(itb+8,nt);
firstt=(itb-1)*dt;

/* Move energy from CMP to CIP */
if(fplo>=nc){

for(k=itb; k<ite; ++k){
datalo[k-itb]=lowpass[nc][k];
}
ints8r(8,dt,firstt,datalo,0.0,0.0,1,&t,&amplo);
mig[iip][nuevo_it] +=geoms*obliq*angtaper*amplo;
} else if(fplo<nc){
for(k=itb; k<ite; ++k){
datalo[k-itb]=lowpass[fplo][k];
datahi[k-itb]=lowpass[fphi][k];
}
ints8r(8,dt,firstt,datalo,0.0,0.0,1,&t,&amplo);
ints8r(8,dt,firstt,datahi,0.0,0.0,1,&t,&amphi);
mig[iip][nuevo_it] += geoms*obliq*angtaper*(wlo*amplo + whi*amphi);
}
}

}
}

/*Envía el bloque de datos semi-migrados con su respectivo
identificador*/
MPI_Send(mig[0], new_nt*ntr, MPI_FLOAT, 0,1000+rank, MPI_COMM_WORLD);

}

}
/*--------------------------------------------------------------------------
--------------------------------------------------------------------------
----------------------------*/
```

```
/*RUTINA EN EL PROCESO MAESTRO (master)*/
/*Para el proceso master el valor de rank=0*/
if (rank==0){
int until_dato;
float **migRL=NULL;
int R;
int nuevo_it;
int new_nt;
int i, k;

/* Lectura de datos de entrada*/
/* Get info from first trace */
if (!gettr(&intrace))  err("can't get first trace");
nt=intrace.ns;
dt=(float)intrace.dt/1000000;
tmax=(nt-1)*dt;

MUSTGETPARFLOAT("dx",&dx);
MUSTGETPARSTRING("vfile",&vfile);
if (!getparfloat("angmax",&angmax)) angmax=40;
if (!getparint("firstcdp",&firstcdp)) firstcdp=intrace.cdp;
if (!getparint("fcdpdata",&fcdpdata)) fcdpdata=intrace.cdp;
if (!getparfloat("hoffset",&hoffset)) hoffset=.5*intrace.offset;
if (!getparint("nfc",&nfc)) nfc=16;
if (!getparint("fwidth",&fwidth)) fwidth=5;
if (!getparint("verbose",&verbose)) verbose=0;

h=hoffset;


/* Store traces in tmpfile while getting a count of number of traces */
tracefp = etmpfile();
hfp = etmpfile();
ntr = 0;
do {
++ntr;

/* get new deltacdp value */
deltacdp=intrace.cdp-oldcdp;

/* read headers and data */
efwrite(&intrace,HDRBYTES, 1, hfp);
efwrite(intrace.data, FSIZE, nt, tracefp);

/* error trappings. */
/* ...did cdp value interval change? */
if ((ntr>3) && (olddeltacdp!=deltacdp)) {

if (verbose) {
warn("cdp interval changed in data");
warn("ntr=%d olddeltacdp=%d deltacdp=%d"
```

```
,ntr,olddeltacdp,deltacdp);
 check_cdp=cwp_true;
}
}

/* save cdp and deltacdp values */
oldcdp=intrace.cdp;
olddeltacdp=deltacdp;

} while (gettr(&intrace));


/* get last cdp  and dcdp */
if (!getparint("lastcdp",&lastcdp)) lastcdp=intrace.cdp;
if (!getparint("dcdp",&dcdp))dcdp=deltacdp - 1;


/* error trappings */
if ( (firstcdp==lastcdp)
|| (dcdp==0)
|| (check_cdp==cwp_true) ) warn("Check cdp values in data!");

/* rewind trace file pointer and header file pointer */
erewind(tracefp);
erewind(hfp);

/* total number of cdp's in data */
ncdp=lastcdp-firstcdp+1;

/* Set up FFT parameters */
nfft = npfaro(nt, LOOKFAC*nt);
if(nfft>= SU_NFLTS || nfft >= PFA_MAX)
  err("Padded nt=%d -- too big",nfft);
nf = nfft/2 + 1;

/* Determine number of filters for antialiasing */
fnyq= 1.0/(2*dt);
nc=ceil(fnyq/fwidth);
if (verbose)
warn(" The number of filters for antialiasing is nc= %d",nc);


/* Allocate space ----------------------------*/
data = alloc2float(nt,ntr);
mig=   alloc2float(nt,ntr);
fc = alloc1float(nc+1);
rtin= ealloc1float(nfft);
filter= alloc1float(nf);
rtout= ealloc1float(nfft);
ct= ealloc1complex(nf);
/*Grandes que se envian:*/
lowpass=alloc2float(nt,nc+1);
vel=   alloc2float(nt,ncdp);
/*-----------------------------------------*/
```

```
/* Read data from temporal array */
for (ix=0; ix<ntr; ++ix){
efread(data[ix],FSIZE,nt,tracefp);
}

/* read velocities */
vfp=efopen(vfile,"r");
efread(vel[0],FSIZE,nt*ncdp,vfp);
efclose(vfp);

/* Zero all arrays */
memset((void *) mig[0], 0,nt*ntr*FSIZE);
memset((void *) rtin, 0, nfft*FSIZE);
memset((void *) lowpass[0], 0,nt*(nc+1)*FSIZE);
memset((void *) filter, 0, nf*FSIZE);


/* Calculate cut frequencies for low-pass filters */
for(i=1; i<nc+1; ++i){
fc[i]= fnyq*i/nc;
}


/*******Empieza el proceso de migración*************************/
*************************************************************/
/* Loop over input mid-points first */
if (verbose) warn("Starting migration process...\n");


/*Envió de datos pequeños para los procesos. De 1 a 2XM*/

for (until_dato=1; until_dato<=2*XM;++until_dato){
MPI_Isend(&ntr, 1, MPI_INT, until_dato, 300, MPI_COMM_WORLD, &reqs_right);
MPI_Isend(&dt, 1, MPI_FLOAT, until_dato, 304, MPI_COMM_WORLD, &reqs_right);
MPI_Isend(&dx, 1, MPI_FLOAT, until_dato, 305, MPI_COMM_WORLD, &reqs_right);
MPI_Isend(&h, 1, MPI_FLOAT, until_dato, 306, MPI_COMM_WORLD, &reqs_right);
MPI_Isend(&tmax, 1, MPI_FLOAT, until_dato, 307, MPI_COMM_WORLD, &reqs_right);
MPI_Isend(&angmax, 1, MPI_FLOAT, until_dato, 308, MPI_COMM_WORLD, &reqs_right);
MPI_Isend(&nc, 1, MPI_INT, until_dato, 309, MPI_COMM_WORLD, &reqs_right);
MPI_Isend(&nt, 1, MPI_INT, until_dato, 380, MPI_COMM_WORLD, &reqs_right);
MPI_Isend(&ncdp, 1, MPI_INT, until_dato, 381, MPI_COMM_WORLD, &reqs_right);
MPI_Isend(&dcdp, 1, MPI_INT, until_dato, 382, MPI_COMM_WORLD, &reqs_right);
MPI_Isend(&fcdpdata, 1, MPI_INT, until_dato, 383, MPI_COMM_WORLD, &reqs_right);
}

/*Envió de la matriz de velocidades. de 1 a 2XM*/
for (until_dato=1; until_dato<=2*XM;++until_dato){
MPI_Send(vel[0], nt*ncdp, MPI_FLOAT, until_dato, 31, MPI_COMM_WORLD);/*,
&reqs_right);*/
}

/*<<<Insert 1***/
erewind(hfp);/*movido*/
```

```
/*****************************FILTRO*****************************/
/****************************************************************/
/****************************************************************/
/* Calculate low-pass filtered versions  */
/* of the data to be used for antialiasing */
for (imp=0; imp<ntr; ++imp){

for(it=0; it<nt; ++it){
rtin[it]=data[imp][it];
}

for(ifc=1; ifc<nc+1; ++ifc){
memset((void *) rtout, 0, nfft*FSIZE);
memset((void *) ct, 0, nf*FSIZE);
lpfilt(nfc,nfft,dt,fc[ifc],filter);
pfarc(1,nfft,rtin,ct);

for(it=0; it<nf; ++it){
ct[it] = crmul(ct[it],filter[it]);
}
pfacr(-1,nfft,ct,rtout);
for(it=0; it<nt; ++it){
lowpass[ifc][it]= rtout[it];
}
}

/*Envió de la matriz filtrada para el valor actual de imp*/
for (until=1;until<=2*XM;++until){
MPI_Send(lowpass[0], nt*(nc+1), MPI_FLOAT, until, 5000+imp, MPI_COMM_WORLD);
}
}
/*--------------------------------------------------------------*/
/*--------------------------------------------------------------*/
/*--------------------------------------------------------------*/




/*Recolección de los datos migrados*/
for (R=1;R<=XM;++R){

nuevo_it=-1;
new_nt=ceil(nt/XM);

/*Reserva memoria para el bloque de datos a recibir*/
migRL=   alloc2float(new_nt,ntr);
memset((void *) migRL[0], 0,new_nt*ntr*FSIZE);

/*Recibe el bloque de datos semi-migrados del proceso R actual*/
MPI_Irecv(migRL[0], new_nt*ntr, MPI_FLOAT, R,1000+R, MPI_COMM_WORLD,
&reqs_gat_2);
```

```
MPI_Wait(&reqs_gat_2,&estado4);

/*tiempo=MPI_Wtime();*/
for(i=R-1;i<nt;i=i+XM){

nuevo_it=nuevo_it+1;

for(k=0;k<ntr;++k){
mig[k][i] += migRL[k][nuevo_it];
}

}
/*tiempo=MPI_Wtime()-tiempo;warn("Tiempo de recoleccion aprox = %lf
s",2*XM*tiempo);*/
}

for (R=XM+1;R<=2*XM;++R){

nuevo_it=-1;
new_nt=ceil(nt/XM);

/*Reserva memoria para el bloque de datos a recibir*/
migRL=   alloc2float(new_nt,ntr);
memset((void *) migRL[0], 0,new_nt*ntr*FSIZE);

/*Recibe el bloque de datos semi-migrados del proceso R actual*/
MPI_Irecv(migRL[0], new_nt*ntr, MPI_FLOAT, R,1000+R, MPI_COMM_WORLD,
&reqs_gat_1);
MPI_Wait(&reqs_gat_1,&estado4);

for(i=R-XM-1;i<nt;i=i+XM){

nuevo_it=nuevo_it+1;

for(k=0;k<ntr;++k){
mig[k][i] += migRL[k][nuevo_it];
}
}
}
/* fin de la recoleccion*/




/* Output migrated data */
/*tiempo=MPI_Wtime();*/
for (ix=0; ix<ntr; ++ix) {
efread(&outtrace, HDRBYTES, 1, hfp);
for (it=0; it<nt; ++it) {
outtrace.data[it] = mig[ix][it];
}
puttr(&outtrace);
}
```

```
/*tiempo=MPI_Wtime()-tiempo;warn("Tiempo output migrated data = %lf
s",tiempo);*/
/*-----------------------------------------------------------------------
---------------------*/


efclose(hfp);

tiempo_mig=MPI_Wtime();/*-tiempo_mig;*/warn("=> MigraciOn completa %lf s
%i Procesos XD\n",tiempo_mig,numprocs);


/*********Termina el proceso de migración************************/
/****************************************************************/

}

MPI_Finalize();

return(CWP_Exit());
}


/*ESTA FUNCION ES NECESARIA SOLO EN EL/LOS PROCESOS DE FILTRADO*/

void
lpfilt(int nfc, int nfft, float dt, float fhi, float *filter)
/*************************************************************************
*******
lpfilt -- low-pass filter using Lanczos Smoothing
(R.W. Hamming:"Digital Filtering",1977)
**************************************************************************
***
Input:
nfcnumber of Fourier coefficients to approximate ideal filter
nfftnumber of points in the fft
dttime sampling interval
fhicut-frequency

Output:
filter  array[nf] of filter values
**************************************************************************
****
Notes: Filter is to be applied in the frequency domain
**************************************************************************
****
Author: CWP: Carlos Pacheco   2006
**************************************************************************
****/



{
int i,j;  /* counters */
```

```
int nf;   /* Number of frequencies (including Nyquist) */
float onfft;  /* reciprocal of nfft */
float fn; /* Nyquist frequency */
float df; /* frequency interval */
float dw; /* frequency interval in radians */
float whi;/* cut-frequency in radians */
float w;  /* radian frequency */

nf= nfft/2 + 1;
onfft=1.0/nfft;
fn=1.0/(2*dt);
df=onfft/dt;
whi=fhi*PI/fn;
dw=df*PI/fn;

for(i=0; i<nf; ++i){
filter[i]= whi/PI;
w=i*dw;

for(j=1; j<nfc; ++j){
float c= sin(whi*j)*sin(PI*j/nfc)*2*nfc/(PI*PI*j*j);
filter[i] +=c*cos(j*w);
}
}
}
```

Se pueden descargar los códigos tanto de la migración b como de la migración d de:

http://cid-f6f36f2ae47b3d79.skydrive.live.com/browse.aspx/SEISMIC?uc=2

ANEXO 4. ALGORITMO SECUENCIAL DE MIGRACIÓN DE KIRCHHOFF 2D CON PRE-APILAMIENTO EN TIEMPO.

Se presenta el código de la migración secuencial (suktmig2d) disponible en SU. Las secciones de código en negrilla son los bucles repetitivos que representan el mayor gasto de tiempo y sobre los cuales se paralelizó el algoritmo.

```
/* Copyright (c) Colorado School of Mines, 2008.*/
/* All rights reserved.                       */

/* SUKTMIG2D: $Revision: 1.2 $ ; $Date: 2007/04/23 23:46:42 $*/

#include "su.h"
#include "segy.h"
#include "header.h"

/********************** self documentation **********************/
char *sdoc[] = {
"                                                       ",
" SUKTMIG2D - prestack time migration of a common-offset section with",
"            the double-square root (DSR) operator              ",
"                                                       ",
"   suktmig2d < infile vfile= [parameters]  > outfile               ",
"                                                       ",
" Required Parameters:                                      ",
" vfile=     rms velocity file (units/s) v(t,x) as a function of time",
" dx=        distance (units) between consecutive traces        ",
"                                                       ",
" Optional parameters:                                      ",
" fcdpdata=tr.cdp  first cdp in data                    ",
" firstcdp=fcdpdata      first cdp number in velocity file          ",
" lastcdp=from header    last cdp number in velocity file           ",
" dcdp=from header number of cdps between consecutive traces   ",
" angmax=40  maximum aperture angle for migration (degrees)         ",
" hoffset=.5*tr.offset         half offset (m)                  ",
" nfc=16     number of Fourier-coefficients to approximate low-pass   ",
"            filters. The larger nfc the narrower the filter         ",
" fwidth=5   high-end frequency increment for the low-pass filters    ",
"            in Hz. The lower this number the more the number of     ",
"            lowpass filters to be calculated for each input trace.  ",
"                                                       ",
" Notes:                                                ",
" Data must be preprocessed with sufrac to correct for the wave-shaping    ",
" factor using phasefac=.25 for 2D migration.                       ",
"                                                       ",
" Input traces must be sorted into offset and cdp number. The velocity     ",
" file consists of rms velocities for all CMPs as a function of vertical",
" time and horizontal position v(t,z) in C-style binary floating point     ",
" numbers. It's easiest to supply v(t,z) that has the same dimensions as",
" the input data to be migrated.                            ",
"                                                       ",
" The units may be feet or meters, as long as these are consistent for     ",
" Antialias filter is performed using (Gray,1992, Geoph. Prosp), using     ",
```

```
" nc low- pass filtered copies of the data. The cutoff frequencies are     ",
" calculated  as fractions of the Nyquist frequency.                 ",
"                                                                    ",
" The maximum allowed angle is 80 degrees(a 10 degree taper is applied     ",
" to the end of the aperture)                                        ",
NULL};

#define LOOKFAC 2       /* Look ahead factor for npfaro   */
#define PFA_MAX 720720  /* Largest allowed nfft           */


/* Prototype of functions used internally */
void lpfilt(int nfc, int nfft, float dt, float fhi, float *filter);

segy intrace;      /* input traces */
segy outtrace;     /* migrated output traces */

int
main(int argc, char **argv)
{
      int i,k,imp,iip,it,ix,ifc;      /* counters */
      int ntr,nt;              /* x,t */

      int verbose; /* is verbose?                        */
      int nc;            /* number of low-pass filtered versions     */
                  /*  of the data for antialiasing      */
      int nfft,nf; /* number of frequencies        */
      int nfc;     /* number of Fourier coefficients for low-pass filter */
      int fwidth;  /* high-end frequency increment for the low-pass */
                  /* filters                      */
      int firstcdp=0;    /* first cdp in velocity file         */
      int lastcdp=0;     /* last cdp in velocity file          */
      int oldcdp=0;      /* temporary storage                  */
      int fcdpdata=0;    /* first cdp in the data        */
      int olddeltacdp=0;
      int deltacdp;
      int ncdp=0;  /* number of cdps in the velocity file      */
      int dcdp=0;  /* number of cdps between consecutive traces */

      float dx=0.0;      /* cdp sample interval */
      float hoffset=0.0;  /* half receiver-source */
      float p=0.0; /* horizontal slowness of the migration operator */
      float pmin=0.0;    /* maximum horizontal slowness for which there's */
                  /* no aliasing of the operator */
      float dt;    /* t sample interval */
      float h;     /* offset */
      float x;     /* aperture distance */
      float xmax=0.0;    /* maximum aperture distance */

      float obliq; /* obliquity factor */
      float geoms; /* geometrical spreading factor */
      float angmax;   /* maximum aperture angle */

      float mp,ip; /* mid-point and image-point coordinates */
      float t;     /* time */
      float t0;    /* vertical traveltime */
      float tmax;  /* maximum time */

      float fnyq;  /* Nyquist frequency */
      float ang;   /* aperture angle */
      float angtaper=0.0;      /* aperture-angle taper */
```

```
float v;           /* velocity */

float *fc=NULL;          /* cut-frequencies for low-pass filters */
float *filter=NULL;      /* array of low-pass filter values */

float **vel=NULL;  /* array of velocity values from vfile */
float **data=NULL; /* input data array*/
float **lowpass=NULL;   /* low-pass filtered version of the trace */
float **mig=NULL;  /* output migrated data array */

register float *rtin=NULL,*rtout=NULL;/* real traces */
register complex *ct=NULL;   /* complex trace */

/* file names */
char *vfile="";          /* name of velocity file */
FILE *vfp=NULL;
FILE *tracefp=NULL;      /* temp file to hold traces*/
FILE *hfp=NULL;          /* temp file to hold trace headers */

float datalo[8], datahi[8];
int itb, ite;
float firstt, amplo, amphi;

cwp_Bool check_cdp=cwp_false;  /* check cdp in velocity file   */

/* Hook up getpar to handle the parameters */
initargs(argc,argv);
requestdoc(0);

/* Get info from first trace */
if (!gettr(&intrace))  err("can't get first trace");
nt=intrace.ns;
dt=(float)intrace.dt/1000000;
tmax=(nt-1)*dt;

MUSTGETPARFLOAT("dx",&dx);
MUSTGETPARSTRING("vfile",&vfile);
if (!getparfloat("angmax",&angmax)) angmax=40;
if (!getparint("firstcdp",&firstcdp)) firstcdp=intrace.cdp;
if (!getparint("fcdpdata",&fcdpdata)) fcdpdata=intrace.cdp;
if (!getparfloat("hoffset",&hoffset)) hoffset=.5*intrace.offset;
if (!getparint("nfc",&nfc)) nfc=16;
if (!getparint("fwidth",&fwidth)) fwidth=5;
if (!getparint("verbose",&verbose)) verbose=0;

h=hoffset;

/* Store traces in tmpfile while getting a count of number of traces */
tracefp = etmpfile();
hfp = etmpfile();
ntr = 0;
do {
      ++ntr;

      /* get new deltacdp value */
      deltacdp=intrace.cdp-oldcdp;

      /* read headers and data */
      efwrite(&intrace,HDRBYTES, 1, hfp);
      efwrite(intrace.data, FSIZE, nt, tracefp);
```

```
        /* error trappings. */
        /* ...did cdp value interval change? */
        if ((ntr>3) && (olddeltacdp!=deltacdp)) {

                if (verbose) {
                warn("cdp interval changed in data");
                warn("ntr=%d olddeltacdp=%d deltacdp=%d"
                        ,ntr,olddeltacdp,deltacdp);
                check_cdp=cwp_true;
                }
        }

        /* save cdp and deltacdp values */
        oldcdp=intrace.cdp;
        olddeltacdp=deltacdp;

} while (gettr(&intrace));

/* get last cdp  and dcdp */
if (!getparint("lastcdp",&lastcdp)) lastcdp=intrace.cdp;
if (!getparint("dcdp",&dcdp))  dcdp=deltacdp - 1;


/* error trappings */
if ( (firstcdp==lastcdp)
        || (dcdp==0)
        || (check_cdp==cwp_true) ) warn("Check cdp values in data!");

/* rewind trace file pointer and header file pointer */
erewind(tracefp);
erewind(hfp);

/* total number of cdp's in data */
ncdp=lastcdp-firstcdp+1;

/* Set up FFT parameters */
nfft = npfaro(nt, LOOKFAC*nt);
if(nfft>= SU_NFLTS || nfft >= PFA_MAX)
  err("Padded nt=%d -- too big",nfft);
nf = nfft/2 + 1;

/* Determine number of filters for antialiasing */
fnyq= 1.0/(2*dt);
nc=ceil(fnyq/fwidth);
if (verbose)
        warn(" The number of filters for antialiasing is nc= %d",nc);

/* Allocate space */
data = alloc2float(nt,ntr);
lowpass=alloc2float(nt,nc+1);
mig=   alloc2float(nt,ntr);
vel=   alloc2float(nt,ncdp);
fc = alloc1float(nc+1);
rtin= ealloc1float(nfft);
rtout= ealloc1float(nfft);
ct= ealloc1complex(nf);
filter= alloc1float(nf);

/* Read data from temporal array */
for (ix=0; ix<ntr; ++ix){
        efread(data[ix],FSIZE,nt,tracefp);
```

```
}

/* read velocities */
vfp=efopen(vfile,"r");
efread(vel[0],FSIZE,nt*ncdp,vfp);
efclose(vfp);

/* Zero all arrays */
memset((void *) mig[0], 0,nt*ntr*FSIZE);
memset((void *) rtin, 0, nfft*FSIZE);
memset((void *) filter, 0, nf*FSIZE);
memset((void *) lowpass[0], 0,nt*(nc+1)*FSIZE);

/* Calculate cut frequencies for low-pass filters */
for(i=1; i<nc+1; ++i){
      fc[i]= fnyq*i/nc;
}

/* Start the migration process */
/* Loop over input mid-points first */
if (verbose) warn("Starting migration process...\n");
for(imp=0; imp<ntr; ++imp){
      float perc;

      mp=imp*dx;
      perc=imp*100.0/(ntr-1);
      if(fmod(imp*100,ntr-1)==0 && verbose)
            warn("migrated %g\n ",perc);

      /* Calculate low-pass filtered versions  */
      /* of the data to be used for antialiasing */
      for(it=0; it<nt; ++it){
            rtin[it]=data[imp][it];
      }
      for(ifc=1; ifc<nc+1; ++ifc){
            memset((void *) rtout, 0, nfft*FSIZE);
            memset((void *) ct, 0, nf*FSIZE);
            lpfilt(nfc,nfft,dt,fc[ifc],filter);
            pfarc(1,nfft,rtin,ct);

            for(it=0; it<nf; ++it){
                  ct[it] = crmul(ct[it],filter[it]);
            }
            pfacr(-1,nfft,ct,rtout);
            for(it=0; it<nt; ++it){
                  lowpass[ifc][it]= rtout[it];
            }
      }

      /* Loop over vertical traveltimes */
      for(it=0; it<nt; ++it){
            int lx,ux;

            t0=it*dt;
            v=vel[imp*dcdp+fcdpdata-1][it];
            xmax=tan((angmax+10.0)*PI/180.0)*v*t0;
            lx=MAX(0,imp - ceil(xmax/dx));
```

```
                ux=MIN(ntr,imp + ceil(xmax/dx));

          /* loop over output image-points to the left of the midpoint
*/
          for(iip=imp; iip>lx; --iip){
               float ts,tr;
               int fplo=0, fphi=0;
               float ref,wlo,whi;

               ip=iip*dx;
               x=ip-mp;
               ts=sqrt( pow(t0/2,2) + pow((x+h)/v,2) );
               tr=sqrt( pow(t0/2,2) + pow((h-x)/v,2) );
               t= ts + tr;
               if(t>=tmax) break;
               geoms=sqrt(1/(t*v));
               obliq=sqrt(.5*(1 + (t0*t0/(4*ts*tr))
                         - (1/(ts*tr))*sqrt(ts*ts -
t0*t0/4)*sqrt(tr*tr - t0*t0/4)));
               ang=180.0*fabs(acos(t0/t))/PI;
               if(ang<=angmax) angtaper=1.0;
               if(ang>angmax) angtaper=cos((ang-angmax)*PI/20);
               /* Evaluate migration operator slowness p to determine
*/
               /* the low-pass filtered trace for antialiasing */
               pmin=1/(2*dx*fnyq);
               p=fabs((x+h)/(pow(v,2)*ts) + (x-h)/(pow(v,2)*tr));
                    if(p>0){fplo=floor(nc*pmin/p);}
                    if(p==0){fplo=nc;}
                    ref=fmod(nc*pmin,p);
                    wlo=1-ref;
                    fphi=++fplo;
                    whi=ref;
                    itb=MAX(ceil(t/dt)-3,0);
                    ite=MIN(itb+8,nt);
                    firstt=(itb-1)*dt;
                    /* Move energy from CMP to CIP */
                    if(fplo>=nc){
                         for(k=itb; k<ite; ++k){
                              datalo[k-itb]=lowpass[nc][k];
                         }

     ints8r(8,dt,firstt,datalo,0.0,0.0,1,&t,&amplo);
                         mig[iip][it] +=geoms*obliq*angtaper*amplo;
                    } else if(fplo<nc){
                         for(k=itb; k<ite; ++k){
                              datalo[k-itb]=lowpass[fplo][k];
                              datahi[k-itb]=lowpass[fphi][k];
                         }

     ints8r(8,dt,firstt,datalo,0.0,0.0,1,&t,&amplo);

     ints8r(8,dt,firstt,datahi,0.0,0.0,1,&t,&amphi);
```

```
                              mig[iip][it] +=
geoms*obliq*angtaper*(wlo*amplo + whi*amphi);
                        }
                  }

                  /* loop over output image-points to the right of the
midpoint */
                  for(iip=imp+1; iip<ux; ++iip){
                        float ts,tr;
                        int fplo=0, fphi;
                        float ref,wlo,whi;

                        ip=iip*dx;
                        x=ip-mp;
                        t0=it*dt;
                        ts=sqrt( pow(t0/2,2) + pow((x+h)/v,2) );
                        tr=sqrt( pow(t0/2,2) + pow((h-x)/v,2) );
                        t= ts + tr;
                        if(t>=tmax) break;
                        geoms=sqrt(1/(t*v));
                        obliq=sqrt(.5*(1 + (t0*t0/(4*ts*tr))
                              - (1/(ts*tr))*sqrt(ts*ts
                                    - t0*t0/4)*sqrt(tr*tr
                                                - t0*t0/4)));
                        ang=180.0*fabs(acos(t0/t))/PI;
                        if(ang<=angmax) angtaper=1.0;
                        if(ang>angmax) angtaper=cos((ang-angmax)*PI/20.0);

                        /* Evaluate migration operator slowness p to
determine the  */
                        /* low-pass filtered trace for antialiasing */
                        pmin=1/(2*dx*fnyq);
                        p=fabs((x+h)/(pow(v,2)*ts) + (x-h)/(pow(v,2)*tr));
                        if(p>0){
                              fplo=floor(nc*pmin/p);
                        }
                        if(p==0){
                              fplo=nc;
                        }

                        ref=fmod(nc*pmin,p);
                        wlo=1-ref;
                        fphi=fplo+1;
                        whi=ref;
                        itb=MAX(ceil(t/dt)-3,0);
                        ite=MIN(itb+8,nt);
                        firstt=(itb-1)*dt;

                        /* Move energy from CMP to CIP */
                        if(fplo>=nc){
                              for(k=itb; k<ite; ++k){
                                    datalo[k-itb]=lowpass[nc][k];
                              }
```

```
      ints8r(8,dt,firstt,datalo,0.0,0.0,1,&t,&amplo);
                              mig[iip][it] +=geoms*obliq*angtaper*amplo;
                        } else if(fplo<nc){
                              for(k=itb; k<ite; ++k){
                                    datalo[k-itb]=lowpass[fplo][k];
                                    datahi[k-itb]=lowpass[fphi][k];
                              }

      ints8r(8,dt,firstt,datalo,0.0,0.0,1,&t,&amplo);

      ints8r(8,dt,firstt,datahi,0.0,0.0,1,&t,&amphi);
                              mig[iip][it] +=
geoms*obliq*angtaper*(wlo*amplo + whi*amphi);
                        }
                  }

            }
      }

      /* Output migrated data */
      erewind(hfp);
      for (ix=0; ix<ntr; ++ix) {
            efread(&outtrace, HDRBYTES, 1, hfp);
            for (it=0; it<nt; ++it) {
                  outtrace.data[it] = mig[ix][it];
            }
            puttr(&outtrace);
      }

      efclose(hfp);

      return(CWP_Exit());
}

void
lpfilt(int nfc, int nfft, float dt, float fhi, float *filter)
/*****************************************************************************
lpfilt -- low-pass filter using Lanczos Smoothing
      (R.W. Hamming:"Digital Filtering",1977)
******************************************************************************
Input:
nfc   number of Fourier coefficients to approximate ideal filter
nfft  number of points in the fft
dt    time sampling interval
fhi   cut-frequency

Output:
filter  array[nf] of filter values
******************************************************************************
Notes: Filter is to be applied in the frequency domain
******************************************************************************
Author: CWP: Carlos Pacheco   2006
******************************************************************************/
{
      int i,j;  /* counters */
      int nf;   /* Number of frequencies (including Nyquist) */
      float onfft;  /* reciprocal of nfft */
      float fn; /* Nyquist frequency */
```

```
	float df; /* frequency interval */
	float dw; /* frequency interval in radians */
	float whi;/* cut-frequency in radians */
	float w;  /* radian frequency */

	nf= nfft/2 + 1;
	onfft=1.0/nfft;
	fn=1.0/(2*dt);
	df=onfft/dt;
	whi=fhi*PI/fn;
	dw=df*PI/fn;

	for(i=0; i<nf; ++i){
		filter[i]= whi/PI;
		w=i*dw;

		for(j=1; j<nfc; ++j){
			float c= sin(whi*j)*sin(PI*j/nfc)*2*nfc/(PI*PI*j*j);
			filter[i] +=c*cos(j*w);
		}
	}
}
```

ANEXO 5. DATOS SINTÉTICOS

Con Seismic Unix (SU) es posible generar datos sintéticos de modelos geológicos y de una gran variedad de experimentos y toma de datos. En este apartado se muestra como generar el Modelo del terreno y simular una toma de datos con fuentes y geófonos distribuidos en la superficie. Es suficiente con esos datos para ejecutar tanto el programa secuencial de la migración en tiempo de Kirchhoff 2D disponible en SU (suktmig2d) como los algoritmos de migración implementados en paralelo.

Se muestra a continuación los scripts necesarios y se sombrea las modificaciones necesarias para cambiar el número de trazas y tamaño del modelo.

Nota:
nx: número de trazas.
dx: separación entre trazas.

do

```
#! /bin/sh
# shell for uniformly sampling velocity from a layered model
#set -v


WIDTH=400
HEIGHT=600
WIDTHOFF1=10
WIDTHOFF2=430
WIDTHOFF3=860
HEIGHTOFF1=20
# Generación del sismograma
sh limpiar_datos

sh Xcshot


# generación del modelo de velocidades
nz=41 dz=50 fz=.0  labelz="Depth (m)"
nx=1500 dx=6.666 fx=0.0  labelx="Distance (m)"
```

```
ninf=0 npmax=201
unif2 <input >vfile  ninf=$ninf  npmax=$npmax \
      nz=$nz dz=$dz fz=$fz nx=$nx dx=$dx fx=$fx \
      v00=2000

# trazado de rayos

rayt2d <vfile par=rayt2d.par



exit 0
```

geometric

```
1             0        :reference station number and x-coord.
6.666         0.       : Separación entre la fuente y el primer receptor
                        : Se calcula así: x=Tamaño del campo/#trazas
1   10 10 10  1.   0.  :shot 1 - r1 r2 r3 r4 Número de receptors visibles
50 200                :Número de fuentes, Separación entre fuentes.
```

rayt2d

```
dt=0.004   nt=501
fz=0 nz=41 dz=50
fx=0 nx=1500 dx=6.666
fxs=0 nxs=50 dxs=200
aperx=4999
fa=-75 na=76 da=2 amax=75
fac=0.01 ms=10 ek=1 npv=0
jpfile=jpfile.ray
tfile=tfile
```

param2

```
s                   :job option (s,r)
1  50               :first, last shot for sort. Colocar aquí el número
                     : de trazas
1  30               :first, last trace OR first last receiver.
                     : Se calcula así: x=#de trazas / #de fuentes
10.  25.   35.  50. :frequency spectrum of wavelet
.050                :wavelet length (secs)
.004                :sample rate (secs)
2.                  :record length (secs)
demoshot            :input filename
trazas              :output filename
```

Xcshot

```
#! /bin/sh
# Run CSHOT with CWP X Graphics
# Programa modificado para el curso práctico de
# Métodos Geofísicos, GEOLOGIA-UIS
# Septiembre 2007

# Parámetros del tamaño del modelo
x1beg=0
x1end=2000
x2beg=-100
x2end=10000


#cshot1 calcula los tiempos de llegada de los rayos
#Ver param1 que contiene los parámetros del trazado de rayos
#Ver Modelo que tiene la geometría del reflector
cshot1 |
cshotplot >demoplot  outpar=demopar

xgraph <demoplot par=demopar windowtitle="Rayos a offset cero en un
estrato curvado"\
      -geometry 600x400+500+500 \
      title="Trazado Rayos - Taller 4" \
      label1="Profundidad (km)"  label2="Rango (km)" \
      x1beg=$x1beg x1end=$x1end x2beg=$x2beg x2end=$x2end &

#cshot2 genera las trazas (datos) que van a ser migrados
cshot2
suaddhead <trazas ftn=1 ns=501 |
sushw key=dt,cdp,sx,gx a=4000,1,0,0 b=0,1,1,1 >registro.su
#sugain gagc=1 wagc=0.5 >registro.su
suximage <registro.su title="Registro" \
      xbox=50 ybox=75 \
      wbox=600 hbox=400 \
      label1="Tiempo en Segundos" label2="Traza" &
exit
```

Modelo

```
-100.           0.              upper surface
100000.           0.              ...
    1.      -99999.              end of upper surface

 -100.        1000.              interface 1
```

```
  500.        1050.               ...
 1000.        1100.               ...
 1800.        1020.               ...
 2500.        1000.
 3400.         990.           ...
 3700.            920.
 4000.        990.               ...
 5000.         1000.
 6000.         1010.
 8000.          1015.
10000.         1020.                ...
10500.        1200.               ...
11000.        1020.               ...
11800.         1010.                ...
12500.        1000.
13400.         900.           ...
20000.           1100.
25000.        1110.
34000.        1000.
50000.           1400.
70000.           1000.
95000.           910.
100000.          600.       ...
    1.      -99999.              end of interface 1
 -100.        1510.              interface 2
  500.        1430.               ...
 1000.        1550.               ...
 1800.        1540.               ...
 2500.        1510.
 3400.         1500.          ...
 3700.            1490.
 4000.        1500.               ...
 5000.         1525.
 6000.         1480.
 8000.          1520.
10000.         1530.                ...
10500.        1450.               ...
11000.        1540.               ...
11800.         1525.                ...
12500.        1509.
13400.         1400.          ...
20000.           1710.
25000.        1721.
34000.        1740.
50000.           1780.
70000.           1760.
95000.           1700.
100000.          1720.      ...
    1.      -99999.              end of interface 2
```

Param1

```
Modelo                            : Archivo con el modelo
```

```
2                                       : Número de interfaces
coloresmodelo                           : Archivo colores del modelo
m                                       : Dibujar el modelo (m, w para pozo,
q)
pozofalso                               : Coordenadas de pozo
s                                       : Modo de adquisición, s superficie y
d pozo
geometría                               : Geometría de receptores
sg                                      : Dibujar fuentes, receptores o
quitar (sgq)
rlt                                      : Resultados a guardar en archivo
(r-rayos,l-tiempos de propagación, t-archivo para programa cshot2)
demo                                    :Nombre archivo(s) de salida
-80.   80.                               :Rango de ángulos de búsqueda
0.1                                      :Incremento Angulo de búsqueda
2000.0  3000.0  4000.0                   :Velocidades
n                                        :Onda directa? (y or n)
                                         :Interface con refracciones (1, 2,
...)
y                                        :primarias? (y or n)
1
2
```

Input

```
      0     0
      10000 0           :Tamaño de la prueba
      1      -99999
```

Coloresmodelo

```
0          receivers
2          sources
6          well color
3          caustic rays
4          rays
6          interfaces


key:  (CWP's xgraph colors)
0      black
1      white
2      red
3      green
4      dark blue
5      light blue
6      violet
7      yellow
```

```
8       brown
```

Para generar los datos se deben guardar los anteriores scripts con sus respectivos nombres en una carpeta y luego ejecutar: sh do

De esta forma quedará generar datos sintéticos utilizables por los algoritmos de migración.

También se pueden descargar datos sintéticos de

http://cid-f6f36f2ae47b3d79.skydrive.live.com/browse.aspx/SEISMIC

Donde los autores de este libro han cargado una variedad de datos con diferentes números de trazas además de todos los scripts necesarios para generarlos.