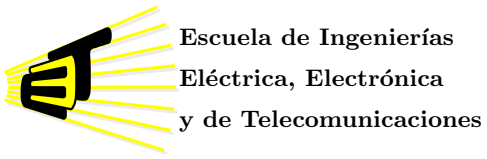
IMPLEMENTACIÓN DEL ALGORITMO DE MIGRACIÓN REVERSA EN TIEMPO (RTM) 3D EN CPU USANDO OpenMP

JESÚS ALONSO ORTIZ LANZZIANO



UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FÍSICO MECÁNICAS
ESCUELA DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y DE TELECOMUNICACIONES
BUCARAMANGA
2016

IMPLEMENTACIÓN DEL ALGORITMO DE MIGRACIÓN REVERSA EN TIEMPO (RTM) 3D EN CPU USANDO OpenMP

Trabajo de investigación presentado como requerimiento para optar al título de:

Ingeniero Electrónico

JESÚS ALONSO ORTIZ LANZZIANO

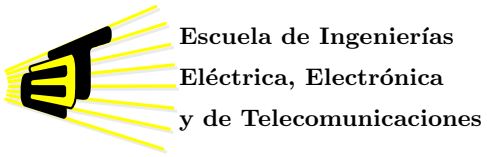
Director:

PhD(c). WILLIAM ALEXANDER SALAMANCA BECERRA

Co-Directores:

MIE. DORFELL LEONARDO PARRA PRADA

PhD. ANA BEATRIZ RAMÍREZ SILVA



UNIVERSIDAD INDUSTRIAL DE SANTANDER

FACULTAD DE INGENIERÍAS FÍSICO MECÁNICAS

ESCUELA DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y DE TELECOMUNICACIONES

BUCARAMANGA

2016

Agradecimientos

Agradezco a Dios por poner en mi camino personas tan maravillosas que han sido mi soporte y me han ayudado a lo largo de toda mi carrera universitaria, a mis padres Jesús Emiro y Genny Maritza por su incondicional apoyo, su ejemplo y sus sabios consejos, sin ellos la culminación de mi carrera no hubiese sido posible, a mi abuelo José María por sus incontables oraciones y su inmenso amor, a Erika Natalia por su compañía y por ser tan especial, al profesor William por haberme asesorado tanto y haber aguantado mi constante "preguntadera", a la Virgen de Torcoroma por iluminarme y llenarme de sabiduría en mis decisiones y a todas y cada una de las personas que hicieron esto posible, muchas muchas gracias ☺.

ÍNDICE GENERAL

Pag.

ÍNDICE DE FIGURAS

Pag.

ÍNDICE DE TABLAS

Pag.

RESUMEN

TÍTULO:

IMPLEMENTACIÓN DEL ALGORITMO DE MIGRACIÓN REVERSA EN TIEMPO (RTM) 3D EN CPU USANDO OpenMP*

AUTOR:

JESÚS ALONSO ORTIZ LANZZIANO**

PALABRAS CLAVES:

CPU, OpenMP, paralelado, optimización, tiempo de ejecución, memoria caché.

El siguiente trabajo de investigación presenta un análisis y una mejora de desempeño del algoritmo de Migración Reversa en Tiempo (RTM) 3D desarrollado por el grupo de investigación CPS, implementado sobre unidades de procesamiento central o CPU. La mejora hace referencia a una disminución del tiempo de ejecución, disminución de la cantidad de accesos a memoria caché y una mejor utilización de los recursos de la CPU. Se usó la interfaz de programación OpenMP, la cual permite distribuir la ejecución del algoritmo entre todos los núcleos de los que dispone una CPU. La toma de tiempos y la evaluación propia de la ejecución se hace con herramientas de software del lenguaje de programación C y del sistema operativo GNU/Linux. La obtención de estos datos permite registrar, tabular y visualizar los resultados del algoritmo original y el desarrollado con OpenMP. Para la realización del trabajo se partió de una implementación base secuencial, se identificaron las secciones paralelables del algoritmo, se mejoró el acceso a memoria caché y se agregaron las directivas de OpenMP. Se realizaron varias pruebas en diferentes sistemas de cómputo de las cuales se hace un análisis y una evaluación de los resultados obtenidos.

* Tesis de grado.

** Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones. Director PhD(c). WILLIAM ALEXANDER SALAMANCA BECERRA. Codirectores MIE. DORFELL LEONARDO PARRA PRADA y PhD. ANA BEATRIZ RAMÍREZ SILVA.

ABSTRACT

TITLE:

REVERSE TIME MIGRATION (RTM) 3D ALGORITHM IMPLEMENTATION ON CPU USING OpenMP*

AUTHOR:

JESÚS ALONSO ORTIZ LANZZIANO**

KEYWORDS:

CPU, OpenMP, parallelization, optimization, execution time, cache.

The following research work presents an analysis and improve the performance of the Reverse Time Migration (RTM) 3D algorithm developed by CPS research group, implemented on Central Processing Units (CPUs). Those improvements can be a decrease in the runtime, a reduction in the number of accesses to cache memory and a better use of CPU resources. The OpenMP programming interface is used in this work, which allows to distribute the execution of the algorithm among all cores available in a CPU. The time and performance measurement is made by means of C programming language software tools and GNU/Linux operating system applications. Obtaining these data will allow to record, tabulate and display the results of the original algorithm and the one developed with OpenMP. The work started from a sequential basis implementation, then paralelables sections of the algorithm were identified, the access to cache was optimized and OpenMP directives were added. Several tests were performed on different computers, an assessment analysis of the results was performed.

* Undergraduate Thesis.

** Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones. Directed by PhD(c). WILLIAM ALEXANDER SALAMANCA BECERRA, MIE. DORFELL LEONARDO PARRA PRADA and PhD. ANA BEATRIZ RAMÍREZ SILVA.

INTRODUCCIÓN

El cómputo en paralelo y las mejoras de desempeño de algoritmos tiene gran importancia en los campos de simulación de fenómenos geofísicos, en estudios de estadística, de física, entre otros. Los algoritmos son implementados en dispositivos de cómputo como: unidades de procesamiento central (CPU), unidades de procesamiento gráfico (GPU), dispositivos programables como las FPGA y supercomputadores (HPC) [9], [11]. A principios de los años 2000 la evolución de los procesadores tuvo un cambio radical; era imposible seguir aumentando la frecuencia de operación como se venía haciendo [21], fue entonces cuando tecnologías multinúcleo o *multicore* comenzaron a surgir, y con esto la necesidad de nuevas técnicas de programación orientadas a la utilización de múltiples núcleos de manera simultánea. El resultado fue una mejora significativa en el rendimiento, el volumen de datos procesados por segundo (*throughput*), y el uso de varios núcleos en un chip permitió distribuir la carga de trabajo.

El presente trabajo busca mejorar el código de Migración Reversa en Tiempo en tres dimensiones (RTM 3D) desarrollado por el grupo de investigación CPS para la implementación en CPU. Se utilizará OpenMP, un desarrollador de aplicaciones que puede modificar una ejecución de un algoritmo secuencial por una ejecución en paralelo. Se busca aprovechar todos los recursos de los que dispone una CPU moderna (e.g. el número de núcleos y la cantidad de memoria caché). Al incrementar el uso de los recursos de la CPU en la implementación del algoritmo se espera mejorar el tiempo de ejecución.

El principal objetivo de este trabajo es implementar el algoritmo de Migración Reversa en Tiempo (RTM) 3D mediante el uso de OpenMP. En primera instancia se documentará el funcionamiento y la metodología de trabajo de esta técnica de programación. Luego se analizará el código secuencial que será modificado para decidir en qué lugar serán aplicadas las directivas de OpenMP. Posteriormente se implementará el algoritmo RTM 3D utilizando las directivas y se verificará el correcto funcionamiento de la implementación comparándola con la implementación secuencial original.

Capítulo 1

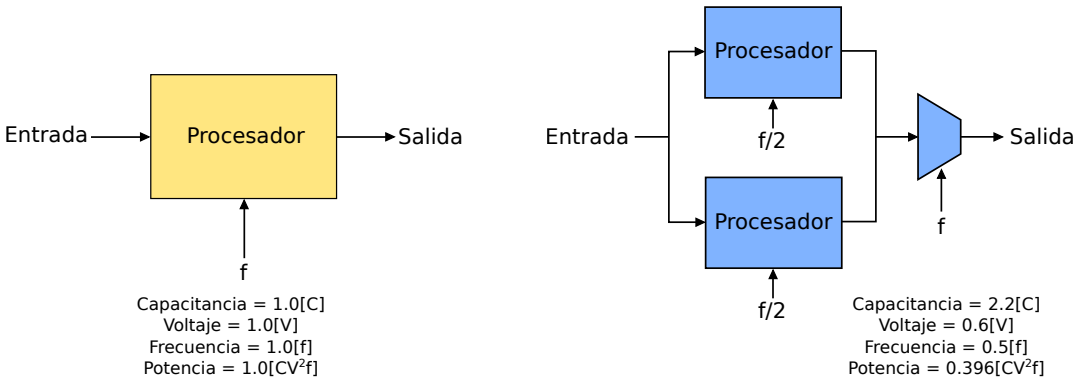
PROCESADORES MULTINÚCLEO

1.1. DE FRECUENCIA DE OPERACIÓN A MULTINÚCLEO

Años atrás la tendencia en el diseño y fabricación de un procesador o CPU era aumentar la frecuencia de operación, al aumentar la frecuencia, aumenta el calor y la potencia consumida. Llegó un momento en el que la cantidad de transistores y el calor disipado eran tan altos que fue difícil seguir priorizando la velocidad [21]. Esto debido a que el área de la que disponía el chip no contaba con el espacio suficiente para más de transistores, y los materiales de los cuales estaba hecho necesitaban refrigeración especial para no dañarse. Fue entonces donde los avances en la escala de integración de semiconductores permitieron la incorporación de 2, 4, 8, 12 y más procesadores en un chip [22], a esta configuración se le llamó multinúcleo (*multicore*). Este reparto otorga cierto nivel de paralelismo, dicho de otra manera los núcleos se distribuyen la carga de trabajo, ejecutando así instrucciones simultáneamente y acelerando de manera considerable el tiempo de procesamiento. Tal avance dio paso al desarrollo de tecnologías enfocadas al aprovechamiento de múltiples núcleos; Hyper-Threading de Intel por ejemplo, permite emular dos núcleos virtuales a partir de un núcleo físico. De acuerdo a esto procesadores con 4 núcleos pueden ejecutar 8 tareas de procesamiento simultáneamente [1].

Según [2], el consumo de potencia de los procesadores multinúcleo es menor al de su precedente. Un procesador de dos núcleos que opera a la mitad de la frecuencia de un procesador de un solo núcleo, consume aproximadamente el 60 % menos de energía, véase la Figura 1. En otras palabras, se puede conservar la frecuencia de operación disminuyendo el consumo de potencia, o en su lugar, aumentar la frecuencia de operación conservando la potencia consumida. Para el ejemplo de la Figura 1 se asume que la capacitancia aumenta poco más del doble al haber dos procesadores, debido a las conexiones extras necesarias entre ambos. Además, por esta razón también se asume que el voltaje necesario para operar los dos procesadores no es exactamente la mitad.

Figura 1: Potencia en procesadores.



(a) Procesador de un núcleo

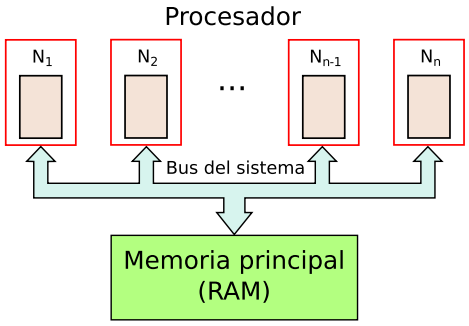
(b) Procesador multinúcleo

Fuente: Adaptado de [2]

1.2. SISTEMAS DE MEMORIA COMPARTIDA (SMP)

El término SMP (*Symmetric Multi-Processing*) está relacionado con los procesadores multinúcleo que comparten una única memoria principal (Memoria RAM). Tanto la arquitectura del hardware como del sistema operativo manejan un esquema simétrico [3]. La arquitectura SMP se caracteriza por tener dos o más núcleos interconectados mediante un bus, de tal manera que el tiempo de acceso a la memoria principal sea aproximadamente el mismo para cada uno. Estos núcleos además tienen su propia memoria, conocida como memoria caché que puede ser privada o compartida. La Figura 2 ilustra de manera general la distribución de la arquitectura SMP.

Figura 2: Sistemas de memoria compartida (SMP).

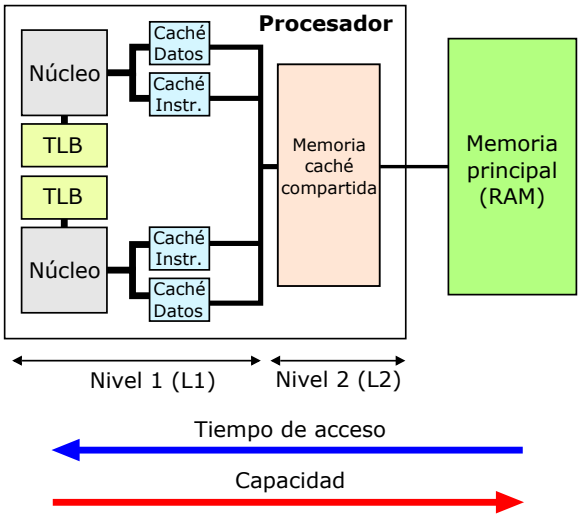


Adaptado de [3].

1.3. MEMORIA CACHÉ

Uno de los retos que enfrenta la arquitectura de computadores hoy en día es la creciente discrepancia en la velocidad del procesador y de la memoria. Los procesadores han sido consistentemente más rápidos, pero cuanto más rápido se puedan llevar a cabo las instrucciones, más rápido necesitan recibir los datos [4]. Infortunadamente, la velocidad con la que los datos se pueden leer y escribir en la memoria no ha aumentado al mismo ritmo. En respuesta las compañías de computadores han construido equipos con sistema de memoria jerárquica, en la que una pequeña, costosa y muy rápida memoria llamada memoria caché, o "caché" para abreviar, suministra al procesador datos e instrucciones a altas velocidades. Generalmente cada procesador de una arquitectura SMP necesita su propia caché privada.

Figura 3: Diagrama de bloques genérico basado en un procesador *dual core*.



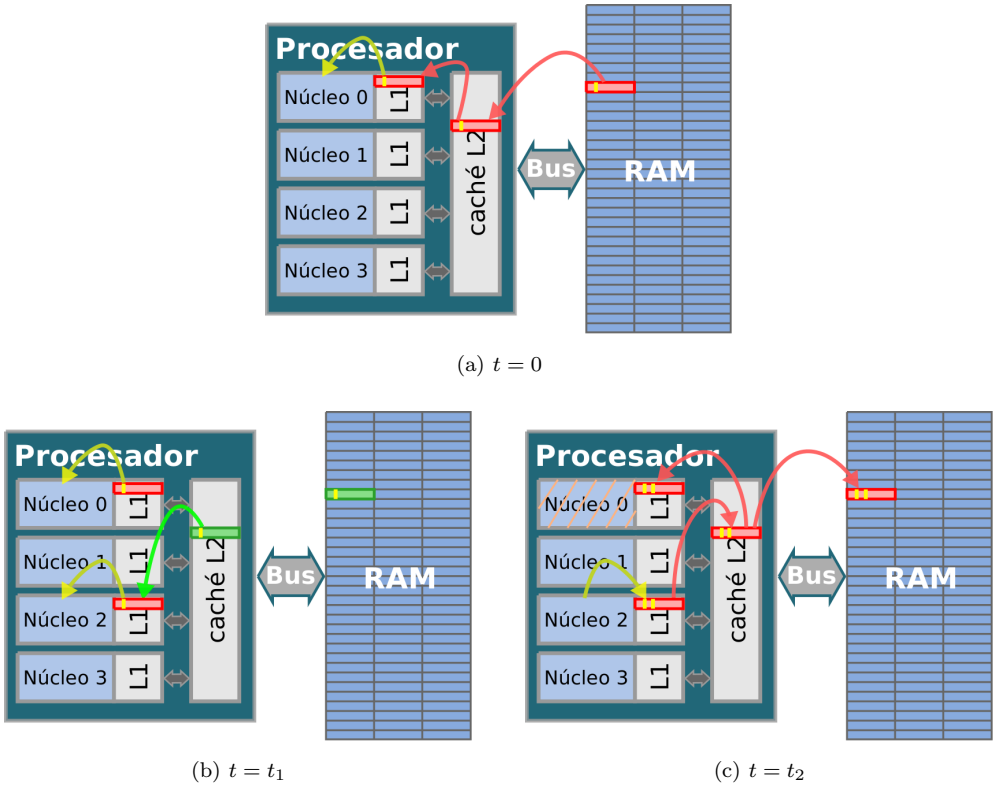
Adaptado de [4].

La Figura 3 muestra un ejemplo genérico de un procesador de doble núcleo con dos niveles de caché. El término nivel es usado para indicar que tan lejos (en términos de tiempo de acceso) está la memoria caché de la CPU, o núcleo. Cuanto más alto sea el nivel, más tiempo se tarda en acceder a la(s) caché(s). En el nivel 1 se distingue una memoria caché para datos (Caché Datos), una para instrucciones (Caché Instr.), y el "*Translation-Lookaside Buffer*" (o TLB para abreviar). La última de ellas es una memoria caché de direcciones. Estas tres cachés son privados a cada núcleo (i.e. otro(s) núcleo(s) no puede(n) acceder a ellas).

1.3.1. Implicaciones de la memoria caché

En un sistema de un procesador, los nuevos valores calculados por el procesador se vuelven a escribir en la memoria caché, donde permanecen hasta que se requiera su espacio para otros datos. Hasta este momento, cualquier nuevo valor que aún no ha sido copiado de nuevo a la memoria principal se almacena allí. Esta estrategia no funciona para sistemas SMPs. Cuando un núcleo de una SMP almacena los resultados de los cálculos locales en su caché privada, los nuevos valores son accesibles sólo al código que se ejecuta en ese núcleo. Si no se toman precauciones, los valores no estarán disponibles para ejecutar instrucciones en otro lugar de la máquina, hasta después de que los bloques correspondientes de los datos se desplacen desde la memoria caché a la memoria principal. Pero no se puede saber con certeza cuando ocurrirá esto. En efecto, los valores antiguos podrían seguir almacenados en otras memorias caché privadas, el código que se ejecuta en otro núcleo podría continuar usando datos antiguos hasta entonces [4].

Figura 4: Sistema SMP con coherencia de caché.



(a) $t = 0$

(b) $t = t_1$

(c) $t = t_2$

Adaptado de [5].

Esto es conocido como problema de consistencia de memoria (*memory consistency problem*) [4]. Se han desarrollado una serie de estrategias para ayudar a superar este problema. Cuyo propósito es asegurar que las modificaciones a los datos que se han producido en un núcleo sean conocidos por el programa que se ejecuta en los otros núcleos, y hacer disponibles los valores modificados para estos, si es necesario. Un sistema que proporciona esta funcionalidad se dice que tiene coherencia de caché, la Figura 4 muestra el funcionamiento de un sistema con esta característica.

En la Figura 4a el núcleo 0 solicita trabajar con `d[1]`, pero `d[1]` no existe en la caché L1 del núcleo 0, por lo tanto se produce un *cache-miss*. `d[1]` se tiene que copiar de la RAM al caché L2 y al caché L1 del núcleo 0, así el núcleo 0 puede trabajar con `d[1]`.

En la Figura 4b el núcleo 2 pide trabajar con `d[1]`, este existe en L2, de modo que se produce un *cache-hit*, `d[1]` se copia al caché L1 del núcleo 2. Ahora el núcleo 2 también puede trabajar con `d[1]`.

En la Figura 4c el núcleo 2 modifica `d[1]` mientras el núcleo 0 trabaja con el `d[1]` de su caché L1. El valor de `d[1]` en la caché L1 del núcleo 2 es diferente al valor de `d[1]` en la caché L2 y al valor de `d[1]` en la caché L1 del núcleo 0, se produce un *cache-fault*, el núcleo 0 se detiene. Las cachés se tienen que sincronizar. `d[1]` de la caché L1 del núcleo 2 se copia al caché L2 y al caché L1 del núcleo 0. Así, el núcleo 0 de desbloquea y trabaja con el nuevo valor de `d[1]`. Posteriormente, cuando L2 se llene, el valor de `d[1]` se actualiza a la RAM.

Existen diferentes técnicas de programación en paralelo compatibles con arquitecturas SMPs, como por ejemplo la *Automatic parallelization* [23], MPI (*Message Passing Interface*) [24], *Pthreads* (POSIX *Threads*) [25] y OpenMP. Para este trabajo se escogió la técnica de programación en paralelo OpenMP, debido a que posee una interfaz sencilla en comparación con las demás y ofrece mejoras importantes en el tiempo de ejecución de algoritmos según anteriores estudios [26], [27].

OpenMP no necesita entender cómo funciona la coherencia de caché en un equipo determinado. De hecho, se puede implementar OpenMP en un equipo que no facilita la coherencia de caché, ya que tiene su propio conjunto de reglas sobre cómo los datos son compartidos entre los núcleos. En su lugar es quien programa el que debe ser consiente del modelo de memoria OpenMP, el cual proporciona datos compartidos, privados y establece cuando los valores compartidos modificados están garantizados y disponibles para todo el código en un programa [4]. En el siguiente capítulo se entrará en detalle acerca del funcionamiento, modo de operación y sintaxis de OpenMP.

Capítulo 2

OpenMP

OpenMP es una interfaz de programación para aplicaciones (API por sus siglas en inglés) que fue desarrollada para permitir la programación en paralelo en sistemas con memoria compartida de tipo SMP. Su objetivo es facilitar el paralelado de algoritmos en numerosas disciplinas. OpenMP proporciona un enfoque relativamente fácil de aprender, así como de explicar. Es compatible con los lenguajes de programación C, C++ y Fortran, asimismo con las arquitecturas de diversos sistemas operativos, incluyendo Solaris, AIX, HP-UX, GNU/Linux, MAC OS X y plataformas basadas en Windows. En entornos que no usan OpenMP, las directivas propias de la interfaz son tratadas como comentarios e ignoradas por el compilador [6].

Entre las características de OpenMP se destacan:

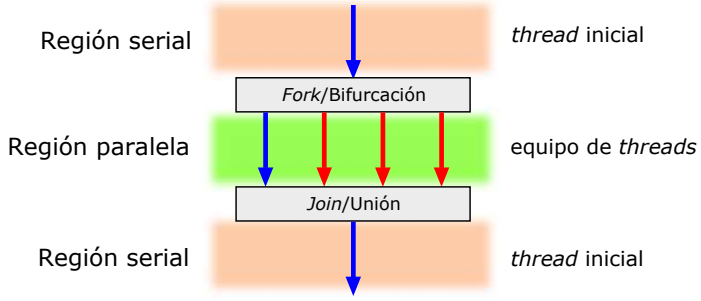
- Enfoque estructurado para la programación paralela.
- Portabilidad.
- Paralelismo "incremental".
- Interfaz simple.
- Independencia del hardware.

2.1. FUNCIONAMIENTO DE OpenMP

Un hilo o *thread* es la unidad de procesamiento más pequeña capaz de ejecutar de manera independiente una serie de instrucciones. Los *threads* son asignados por el sistema operativo a la ejecución de un programa [8]; se establecen algunos recursos a este proceso, incluyendo memoria y registros para guardar los datos. Si varios *threads* trabajan en conjunto para ejecutar un proceso, compartirán recursos, incluyendo el espacio de direcciones del proceso correspondiente. Individualmente cada *thread* necesita sólo unos recursos propios: un puntero de instrucciones y un área de memoria para guardar variables que son específicas a la misma. Múltiples *threads* ejecutándose simultáneamente en uno o

varios procesadores pueden trabajar para acelerar el tiempo de procesamiento de un algoritmo [4].

Figura 5: Modelo de programación *fork-join*.



Adaptado de [4].

Existen diversas formas de escribir algoritmos que se ejecuten en múltiples *threads*, algunas de ellas permiten interacciones complejas. OpenMP utiliza el modelo de programación llamado *fork-join*, que se ilustra en la Figura 5. Bajo este enfoque, el programa inicia con un sólo *thread* de ejecución, tal como lo hace un programa secuencial, a este *thread* se le conoce como *thread* inicial. Si se encuentra una directiva paralela mientras el algoritmo se está ejecutando, el *thread* inicial crea un equipo de *threads* (*fork*/bifurcación), y todos colaboran en conjunto ejecutando la sección del código para la cual la directiva está asignada. Una vez finalizada la directiva sólo el *thread* inicial continúa a partir de este punto, mientras que los demás finalizan (*join*/unión) [4].

2.2. EL MODELO DE MEMORIA DE OpenMP

OpenMP está basado en un modelo de memoria compartida; por lo tanto, los datos se comparten entre los *threads* y son accesibles para todos ellos. Cuando cada *thread* tiene su propia copia de una variable, se dice que la variable es privada. Por ejemplo, cuando un equipo de *threads* ejecuta un ciclo *for* en paralelo, cada uno de ellos necesita su propio valor de la variable de iteración. Esto asegura que cada *thread* escriba sus datos en una posición específica de memoria y no existan conflictos. Los datos pueden declararse como compartidos (*shared*) o privados (*private*), dependiendo de cómo se distribuye el trabajo en la región del código que se desea paralelar [4].

El uso de variables privadas puede resultar muy útil; pueden reducir la frecuencia con que se actualiza la memoria compartida. Por lo tanto, pueden ayudar a evitar los llamados cuellos de botella, o la competencia por el acceso a ciertas posiciones de memoria [4].

2.3. SINTAXIS DE OpenMP EN C

OpenMP proporciona directivas, funciones de librería y variables de entorno para crear y tener control de la ejecución de programas en paralelo. Debido a la gran cantidad de directivas, en esta sección se enfatizarán las más comunes y las que fueron usadas en el presente proyecto. Toda la sintaxis presentada a continuación fue tomada de [4].

En general cada directiva OpenMP posee un constructor y una o varias cláusulas de operación. El constructor puede crear y/o sincronizar una región paralela, así como también asignar *threads* a diferentes secciones del código. Las cláusulas le notifican al compilador cómo se debe acceder a los datos y como deben interactuar estos entre los *threads*. La Figura 6 muestra la sintaxis general de una directiva de OpenMP.

Figura 6: Sintaxis general de una directiva OpenMP.

```
#pragma omp <constructor> [<cláusula>[[,] <cláusula>]...]
```

2.3.1. Constructor `parallel`

Este constructor crea un equipo de *threads* en una región paralela definida. Su sintaxis se muestra en la Figura 7.

Figura 7: Sintaxis del constructor `parallel`.

```
#pragma omp parallel [<cláusula>[[,] <cláusula>]...]
        {
          bloque de código a paralelar
        }
```

2.3.2. Constructor `for`

Este constructor identifica un trabajo compartido, el cual especifica que las iteraciones del ciclo asociado deben ser ejecutadas en paralelo por *threads* que ya hayan sido creados con anterioridad usando el constructor `parallel`. Su sintaxis se muestra en la Figura 8.

Figura 8: Sintaxis del constructor `for`.

```
#pragma omp for [<cláusula>[[,] <cláusula>]...]
          ciclo for a paralelar
```

2.3.3. Constructor `sections`

Este constructor asigna a cada *thread* una región diferente del código. Cada sección identificada es ejecutada por un *thread*. Su sintaxis se observa en la Figura 9.

Figura 9: Sintaxis del constructor `sections`.

```
#pragma omp sections [<cláusula>[[,] <cláusula>]...]
        {
         [#pragma omp section ]
           bloque de código estructurado
         [#pragma omp section
           bloque de código estructurado]
         ...
        }
```

2.3.4. Constructores combinados

El constructor `parallel` puede combinarse con otros constructores para así crear un equipo de *threads* y posteriormente paralelar un ciclo *for* o asignar diferentes *threads* a distintas secciones del código. En la Figura 10a se muestra la sintaxis completa y abreviada para la combinación de los constructores `parallel` y `for` (se crea un equipo de `threads` que trabajan paralelando un ciclo `for`), en la Figura 10b se puede observar el mismo caso anterior pero aplicado a la combinación de los constructores `parallel` y `sections`.

Figura 10: Sintaxis completa y abreviada de los constructores combinados.

```
#pragma omp parallel
 {
   #pragma omp for
     ciclo for a paralelar
 }
```

```
#pragma omp parallel for
     ciclo for a paralelar
```

(a) Combinación de los constructores `parallel` y `for`

```
#pragma omp parallel
 {
   #pragma omp sections
   {
    [#pragma omp section ]
      bloque estructurado
    [#pragma omp section
      bloque estructurado]
    ...
   }
  }
```

```
#pragma omp parallel sections
 {
    [#pragma omp section ]
      bloque estructurado
    [#pragma omp section
      bloque estructurado]
    ...
 }
```

(a) Combinación de los constructores `parallel` y `sections`

2.3.5. Cláusula `schedule`

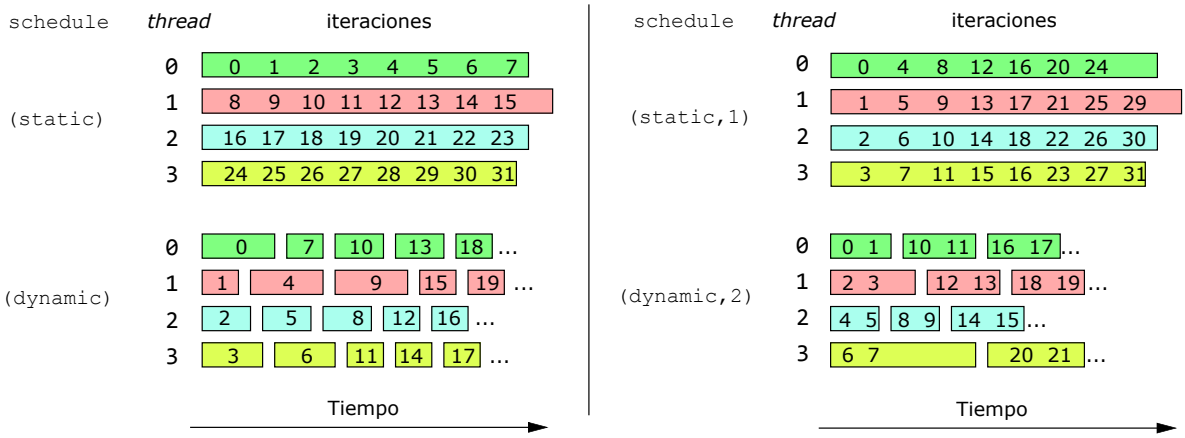
Desarrollada específicamente para ciclos tipo *for*, esta cláusula describe de qué forma son distribuidas las iteraciones entre los *threads*. Las formas más comunes de distribución son la `static` y la `dynamic`. Se puede además asignar un tamaño determinado que define un número de iteraciones por *thread*. A continuación, se muestra la sintaxis de esta cláusula para sus distribuciones más comunes [4].

- `schedule(static,tamaño)`: Las iteraciones se dividen en fragmentos según el tamaño. Estos fragmentos se asignan estáticamente entre los *threads*. El último fragmento puede tener asignado un número más pequeño de iteraciones, esto ocurre cuando el número total de iteraciones no es múltiplo del número total de *threads* utilizados. Si no se indica el tamaño, las iteraciones se dividen en fragmentos de igual dimensión. Esta distribución es recomendada cuando la duración de cada una de las iteraciones dentro del ciclo *for* es aproximadamente la misma.

- `schedule(dynamic,tamaño)`: Las iteraciones son asignadas dinámicamente entre los *threads* a medida que estos van acabando su trabajo. El *thread* ejecuta un fragmento con un número determinado de iteraciones (controlado por el parámetro tamaño), posteriormente solicita otro fragmento hasta que no haya más iteraciones por realizar. Al igual que `static` el último fragmento puede tener un número de iteraciones más pequeño que el asignado. Cuando no se especifica ningún tamaño, el valor predeterminado es 1. Se sugiere hacer uso de esta distribución cuando la duración de las iteraciones dentro del ciclo *for* no sea la misma, de esta manera habrá momentos en que un *thread* se encuentre "libre", así se le asignará otro fragmento que procesar.

Para visualizar la forma de operación de la cláusula *schedule* véase la Figura 11. Se asumen 4 *threads* y un ciclo *for* con un total de 32 iteraciones.

Figura 11: Operación de la cláusula *schedule* para `static` y `dynamic`.



2.3.6. Cláusula `shared`

La cláusula `shared` es usada para especificar cuáles datos serán compartidos entre los *threads* que ejecutan la región paralela asociada. Cada *thread* puede leer o modificar libremente estos valores. La sintaxis de esta cláusula es `shared(lista)`. Todas las variables en la lista serán compartidas entre el equipo de *threads*. La Figura 12 muestra un ejemplo del uso de la cláusula `shared` (todos los *threads* pueden leer o escribir en el vector `a`).

Figura 12: Ejemplo de la cláusula `shared`.

```
#pragma omp parallel for shared(a)
   for (i=0; i<n; i++)
     {
       a[i] += i;
     } /*-- Fin del for paralelo --*/
```

Una implicación importante de los datos compartidos es que múltiples *threads* pueden intentar actualizar simultáneamente una misma posición de memoria, o que un *thread* podría tratar de leer una ubicación que otro *thread* está actualizando. Se debe tener especial cuidado para asegurar que ninguna de estas situaciones se presente y que el acceso a los datos compartidos esté en el orden que el algoritmo lo requiera. Por defecto, todas las variables dentro de un ciclo *for* son compartidas, excepto la variable de iteración del contador.

2.3.7. Cláusula `private`

La cláusula `private` es usada para especificar cuáles datos serán privados a los *threads*. La sintaxis es `private(lista)`. Cada variable en la lista se replica de tal manera, que cada *thread* del equipo de *threads* tiene acceso exclusivo a una copia local de esta variable. Los cambios realizados en los datos por un *thread* no son visibles para los otros. La Figura 13 muestra un ejemplo del uso de la cláusula `private` (cada *thread* tiene su propia copia de las variables `i` y `a`).

Figura 13: Ejemplo de la cláusula `private`.

```
#pragma omp parallel for private(i,a)
   for (i=0; i<n; i++)
     {
       a = i+1;
     } /*-- Fin del for paralelo --*/
```

Si la variable `a` del ejemplo anterior fuera especificada como compartida, múltiples *threads* intentarían actualizarla con valores diferentes de una manera incontrolada. Por defecto en un ciclo *for*, la variable de iteración del contador es privada.

2.4. MEJORA DEL RENDIMIENTO CON OpenMP

OpenMP está orientado a los ciclos iterativos del algoritmo, llamados también *bucles*, como por ejemplo la sentencia *for*. Esta, sin embargo, no debe depender de datos generados por la misma u otra sentencia, es decir, los resultados más eficientes serán aquellos en que los datos sean absolutamente independientes, estas son las secciones de interés del código para OpenMP. Datos que no dependan de otros, aseguran que múltiples *threads* puedan ejecutar varias iteraciones de un ciclo *for* simultáneamente sin alterar el resultado final. Las operaciones punto a punto, como la suma de vectores o la multiplicación de matrices son paralelables.

2.4.1. Acceso a la memoria caché

La jerarquía de memoria (con raras excepciones) no es programable explícitamente por el usuario o el compilador. Los datos de la memoria caché por ejemplo, son almacenados dinámicamente a medida que se van necesitando. Dado el costo computacional que conlleva traer datos almacenados en otros niveles de memoria al procesador, se han creado varias estrategias que ayudan al compilador y al programador (indirectamente) a reducir la frecuencia con que esto ocurre. El principal objetivo es organizar el acceso a los datos, de tal manera que aquellos que se usen más a menudo permanezcan en la caché el mayor tiempo posible. Los lenguajes de programación por lo general pueden guardar los

elementos de una matriz de forma contigua en la memoria. Por lo tanto, si un elemento de la matriz se encuentra en la memoria caché, los elementos de la matriz "cercanos" estarán en el mismo bloque de memoria. Si un cálculo utiliza cualquiera de estos valores mientras aún permanecen en la memoria caché, será favorable para el rendimiento [4].

Como ejemplo se puede considerar la memoria caché usada por una matriz de dos dimensiones en el lenguaje de programación C. Asumiendo que el almacenamiento de la matriz se hace por filas, el elemento `[0][2]` es el siguiente del `[0][1]` que a su vez es el siguiente del elemento `[0][0]` en la memoria. El elemento `[1][1]` es seguido por el `[1][2]`, y así sucesivamente. Cuando un elemento de la matriz es transferido a la caché, los vecinos de la fila también son trasferidos en la misma línea de memoria. Por esta razón para obtener un mejor rendimiento, se debe acceder a los elementos de la matriz fila por fila y no columna por columna. La Figura 14 ilustra ambos casos.

Figura 14: Acceso la a memoria caché

```
for (int i=0; i<n; i++)
   for (int j=0; j<n; j++)
      suma += a[i][j];
```

(a) Ejemplo de un acceso por filas.

```
for (int j=0; j<n; j++)
   for (int i=0; i<n; i++)
      suma += a[i][j];
```

(b) Ejemplo un acceso por columnas.

En la Figura 14a se accede a la matriz a lo largo de las filas. Este enfoque asegura un mejor rendimiento del sistema de memoria, en la Figura 14b por otro lado se accede a la matriz por columnas, esto se traduce en una peor utilización del sistema de memoria, cuanto mayor sea el tamaño de la matriz, peor será su rendimiento [4].

2.5. MEDICIÓN DE DESEMPEÑO

Para evaluar y presentar un análisis comparativo entre el algoritmo original y el desarrollado con OpenMP, se propone el uso del *speed-up* y la aplicación de la Ley de Amdahl.

2.5.1. *Speed-up*

El *speed-up* es la razón entre el tiempo de ejecución de una y otra implementación modificada de un mismo código (e.g. la implementación serial y la implementación en paralelo de un programa). El valor del *speed-up* es adimensional y establece cuántas veces más rápido puede ser ejecutado un algoritmo [5].

$$speed{-}up = \frac{t_1}{t_2} \tag{2.1}$$

Donde:

t_1 es el tiempo de ejecución del algoritmo sin modificar y t_2 el tiempo de ejecución del algoritmo modificado.

2.5.2. Ley de Amdahl

La ley de Amdahl establece que hay un límite en el *speed-up* para un código que ha sido paralelado. Este límite relaciona la porciones entre la cantidad de código que se ha paralelado y la cantidad de código que continúa en serie [5].

Sea t_{real} el tiempo que tarda en ejecutarse un algoritmo, sean t_s y t_p los tiempos de las porciones del algoritmo serie y paralelo respectivamente, tal que $t_s + t_p = t_{real}$. El tiempo de ejecución según la Ley de Amdahl ($t_{teórico}$) para n *threads* es:

$$t_{teórico} = t_s + \frac{t_p}{n} \tag{2.2}$$

Si se asume un número de *threads* infinitos, se puede encontrar el límite para el *speed-up*.

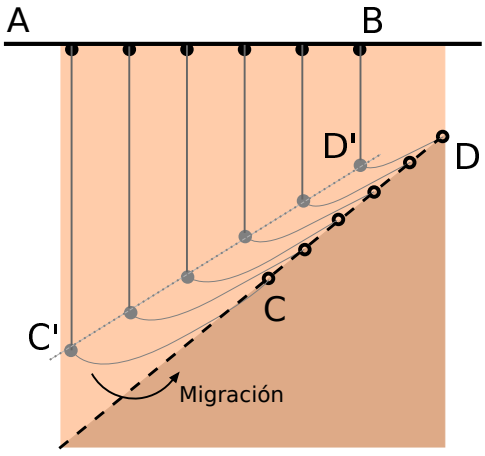
$$speed{-}up_{máximo} = \frac{(t_p + t_s)_{\ 1\ thread}}{t_s} = \frac{(t_{real})_{\ 1\ thread}}{t_s} \tag{2.3}$$

Capítulo 3

MIGRACIÓN SÍSMICA

La exploración del subsuelo en búsqueda de hidrocarburos conlleva varias etapas que van desde la adquisición de los datos, hasta una representación digital del terreno estudiado, pasando por múltiples procesamientos de carácter electrónico y computacional. Por lo general la adquisición de los datos comienza con una perturbación conocida también como fuente (e.g. explosión) en la superficie del terreno que se desea explorar, la energía generada tiene tal magnitud que viaja a través del subsuelo y va reflejándose por las capas que lo componen. Estas reflexiones son registrados y transformados en señales eléctricas conocidas como trazas sísmicas [9], el proceso de esta transformación se hace mediante transductores, conocidos como geófonos o receptores. Existen diversas razones por las cuales los datos registrados y la posición real de las capas del subsuelo no coinciden [10]. La migración se encarga de mover las trazas sísmicas a su verdadera ubicación y crear una representación digital del subsuelo.

Figura 15: Migración sísmica.



Adaptado de [10].

En la Figura 15 se muestra un arreglo de fuentes y receptores en la superficie a lo largo de la línea AB (puntos negros sólidos). La pendiente CD (puntos negros huecos) detectada por los receptores es desplazada incorrectamente a C'D' (puntos grises) como resultado de los datos obtenidos. El propósito de la migración es reorganizar los datos de información sísmica y proyectar la posición real de las capas del subsuelo [10].

Existen diferentes algoritmos para realizar la migración sísmica, entre los más usados se encuentran la migración Kirchhoff, la migración por extrapolación en profundidad, la migración de ondas planas, la migración reversa en tiempo, entre otras [15]. La migración reversa en tiempo o RTM (*Reverse Time Migration*) pertenece al tipo de migraciones TWWE (*Two Way Wave Equation*) que a diferencia de las OWWE (*One Way Wave Equation*) proporcionan resultados superiores para terrenos con características complejas [18]. Esta característica ha hecho que la migración RTM tenga una gran acogida en el campo de la geofísica, pues la hace mejor en muchos aspectos, debido a que permite obtener detalles que con otras migraciones no era posible observar.

La migración RTM posee varias etapas entre las que se destacan una propagación, una retropropagación y la aplicación de una condición de imagen. La propagación hace referencia a la simulación del recorrido de la onda generada por la explosión, la dirección de la onda simulada va desde la superficie hacia el subsuelo. La retropropagación por otro lado simula la misma onda pero en la dirección opuesta, de allí su nombre reversa en tiempo. La condición de imagen es una fórmula matemática que relaciona ambas propagaciones y crea un modelo digital del terreno [10]. El desarrollo de las etapas descritas recibe el nombre de migración de un disparo, el número de disparos está relacionado con la cantidad de explosiones realizadas. El costo computacional de la migración RTM es muy elevado y más cuando se realiza en tres dimensiones [16]. Es por ello que aunque el concepto apareció en 1983 [12] no fue sino hasta hace unos años que comenzó a implementarse.

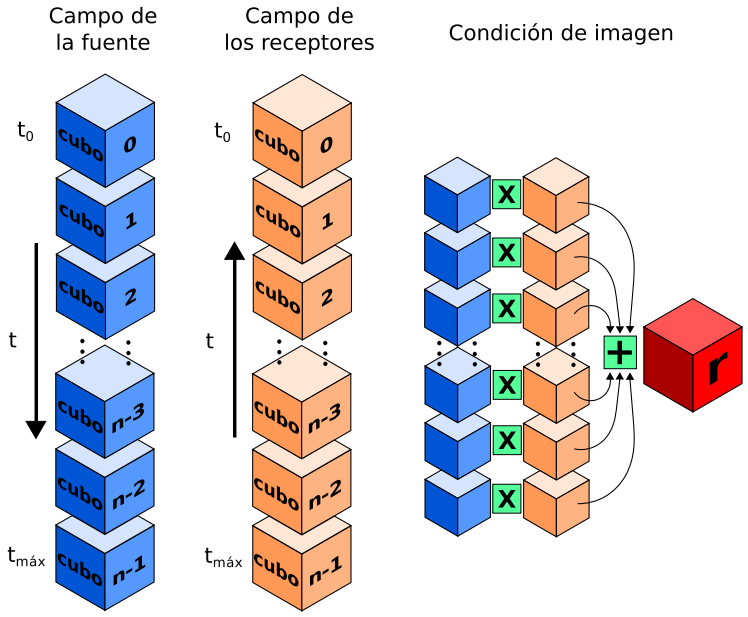
3.1. ALGORITMO DE MIGRACIÓN REVERSA EN TIEMPO (RTM) 3D

Existen diferentes condiciones de imagen para relacionar el campo de propagación y retropropagación, el algoritmo RTM 3D desarrollado por el grupo de investigación CPS de la UIS utiliza la condición de imagen descrita en la ecuación 3.1 para hallar el resultado de la migración $r(x, y, z)$. Se evalúan dos campos de propagación; el campo $d(x, y, z)$ generado por la fuente y el campo $u(x, y, z)$ generado por los receptores [13].

$$r(x, y, z) = \sum_{t_0}^{t_{máx}} u(x, y, z) d(x, y, z) \tag{3.1}$$

Cada disparo migrado lleva consigo pasos de tiempo que simulan la propagación de la onda a lo largo del terreno que se desea estudiar. Para el campo de la fuente, los pasos de tiempo van desde el inicio de la explosión ($t = t_0$) hasta el tiempo establecido en la adquisición de los datos ($t = t_{máx}$). Para el campo de los receptores, por el contrario, va desde el tiempo de la adquisición hasta el inicio de la explosión. La idea general del algoritmo RTM 3D se ilustra en la Figura 16, se calcula un campo de propagación (Campo de la fuente), uno de retropropagación (Campo de los receptores) y se aplica la condición de imagen para hallar el modelo migrado (r).

Figura 16: Propagación de campos y condición de imagen.



Para la obtención de los campos es necesario resolver la ecuación 3.2, la cual modela la propagación de la onda generada por la explosión y extrapola el frente de onda de los receptores en la superficie [13].

$$\nabla^2 P = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2} = \frac{1}{c^2}\frac{\partial^2 P}{\partial t^2} \tag{3.2}$$

Debido a la naturaleza discreta de los datos de entrada, la solución de la ecuación de onda en el algoritmo se hace mediante el método numérico de diferencias finitas. Este método discretiza y aproxima las derivadas espaciales y la derivada temporal de la ecuación de onda. La ecuación 3.3 muestra como ejemplo la ecuación de onda en 3 dimensiones discretizada por el método de diferencias finitas con una

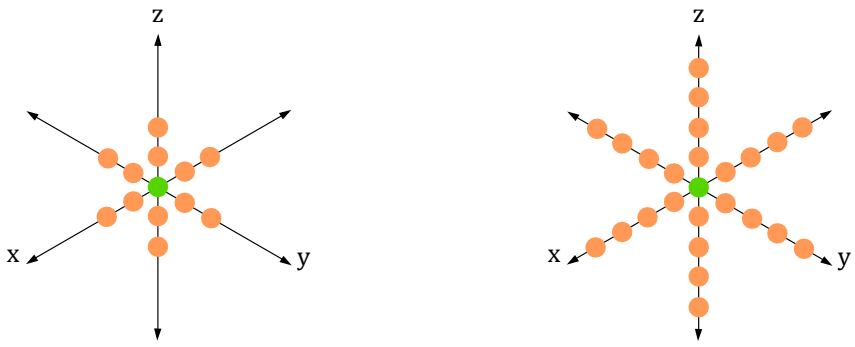
aproximación de orden 2 [19].

$$\frac{C_{-1}P^0_{-1,0,0} + C_0P^0_{0,0,0} + C_1P^0_{1,0,0}}{\Delta x^2} + \frac{C_{-1}P^0_{0,-1,0} + C_0P^0_{0,0,0} + C_1P^0_{0,1,0}}{\Delta y^2} + \frac{C_{-1}P^0_{0,0,-1} + C_0P^0_{0,0,0} + C_1P^0_{0,0,1}}{\Delta z^2} = \frac{1}{c^2}\frac{P^{-1}_{0,0,0} - 2P^0_{0,0,0} + P^1_{0,0,0}}{\Delta t^2} \tag{3.3}$$

$P^n_{(x,y,z)}$ representa la posición de la onda en las coordenadas (x, y, z) para el paso de tiempo n. Los coeficientes para cada término de las diferencias finitas se muestran como C_i, donde i es el término indexado a cada posición. Δx, Δy, Δz son los pasos espaciales correspondientes a cada coordenada y Δt es el paso de tiempo. Por último el término c hace referencia a un modelo de velocidades, el cual es el conjunto de velocidades que representa las propiedades de propagación del subsuelo.

El método de diferencias finitas puede representarse gráficamente como operaciones entre *stencils*. Un *stencil* en 3 dimensiones por ejemplo, es un arreglo de puntos en coordenadas x, y, z tal que, el punto a calcular está en el centro, y el orden de aproximación de las diferencias finitas es el número de puntos vecinos a este en cada eje de coordenadas. Al aumentar el orden de aproximación, aumenta la cantidad de términos $C_iP^n_{(x,y,z)}$ en la ecuación 3.3 y por ende la precisión. La Figura 17 ilustra dos *stencils* en 3 dimensiones para una aproximación de 4^{to} y 8^{vo} orden.

Figura 17: *Stencils* en 3D.



(a) *Stencil* con orden 4. (b) *Stencil* con orden 8.

Fuente: Adaptado de [19]

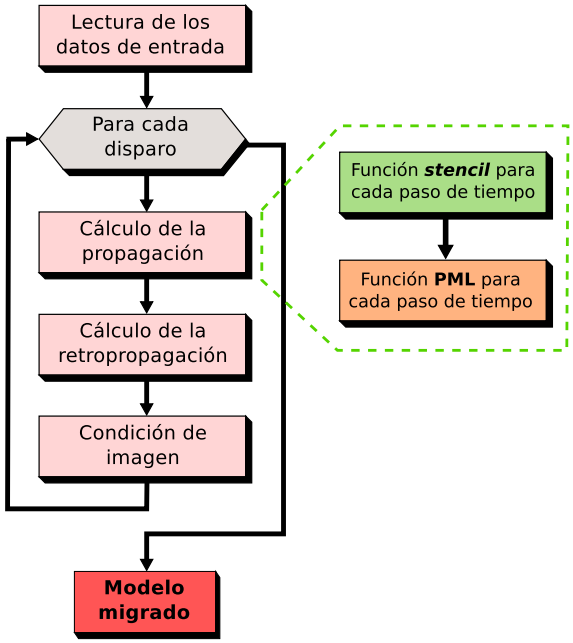
Existen diversas técnicas que impiden la reflexión de la onda propagada en los límites del modelo estudiado, esta propiedad ofrece un comportamiento más acorde a la realidad física de la propagación.

El algoritmo desarrollado por el grupo CPS utiliza la técnica de condición de frontera *Perfectly Matched Layer* (PML) la cual realiza un escalamiento espacial, confinando un medio disipativo infinito a un borde de ancho finito [14], por lo tanto, la energía de la onda es absorbida por dicho borde (zona PML), evitando así reflexiones indeseadas que puedan alterar el resultado final de la migración.

3.2. EJECUCIÓN, MEJORA Y PARALELADO DEL ALGORITMO RTM 3D EN CPU

La ejecución del algoritmo comienza con la lectura de los datos de entrada, que corresponden al modelo de velocidades y a las trazas sísmicas. Para el algoritmo implementado en este trabajo ambos conjuntos de datos son sintéticos. Cabe resaltar que utilizando los mismos parámetros descritos en el siguiente capítulo, los tiempos de ejecución no se verán alterados si se llegasen a utilizar datos reales.

Figura 18: Etapas del algoritmo RTM 3D.



La Figura 18 muestra las etapas del algoritmo RTM 3D. Para el cálculo de la propagación se usa una función que de aquí en adelante será conocida como *stencil*, la cual calcula un cubo completo mediante el método de diferencias finitas (ecuación 3.3) para cada paso de tiempo, uno a la vez. Esta función se compone de tres ciclos *for* encargados de recorrer espacialmente los puntos del cubo en las tres dimensiones, véase el Pseudocódigo 1. Cuando el frente de onda se encuentra cerca de los límites del modelo, se usan seis funciones relacionadas a la condición de frontera PML, una para cada cara del cubo. Cada una de estas funciones calcula su resultado resolviendo la ecuación de onda mediante el uso de diferencias finitas con el agregado de varios términos. Cada función PML posee tres ciclos *for* que al igual que la función *stencil* recorren los puntos del cubo, pero solo en la zona PML, véase el Pseudocódigo 2. El tiempo de de ejecución de las funciones *stencil* y PML comprende aproximadamente el 95 % del tiempo de ejecución total del algoritmo, además debido a la naturaleza de su cómputo tienen un alto potencial de paralelado.

La Figura 19 ilustra el volumen calculado por las funciones *stencil* y PML.

Pseudocódigo 1: Función *stencil*.

Lectura de los datos de entrada, entre ellos:

> `Nx, Ny, Nz` => *tamaño del modelo*
> `dx, dy, dz` => *resolución del modelo*
> `Lim*` => *límites del modelo*

Recorrido de los puntos del cubo sin la zona PML

```
for(iz=Limzinf; iz<Limzsup; iz++) {
for(ix=Limxinf; ix<Limxsup; ix++) {
for(iy=Limyinf; iy<Limysup; iy++) {
```

Cálculo de la ecuación de onda por el método de diferencias finitas. `}}}`

*Cálculo de la función **stencil** para un paso de tiempo.*

Pseudocódigo 2: Una de las funciones PML.

Lectura de los datos de entrada, entre ellos:

> `Nx, Ny, Nz` => *tamaño del modelo*
> `dx, dy, dz` => *resolución del modelo*
> `Lim*` => *límites del modelo*
> `L` => *tamaño de la zona PML*

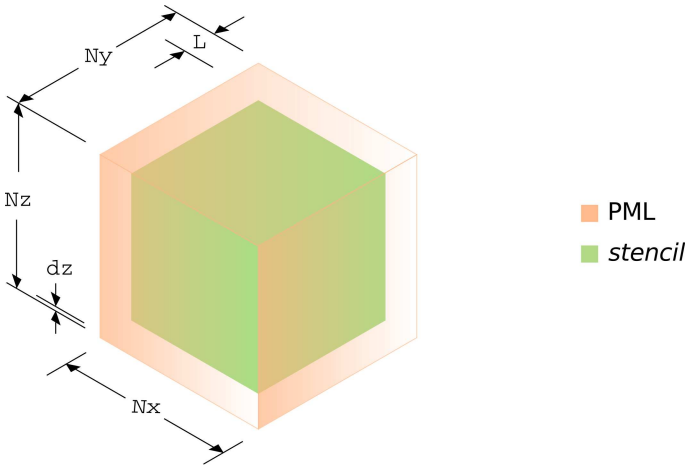
Recorrido de los puntos del cubo en la zona PML

```
for(iz=Limzinf; iz<Limzsup; iz++) {
for(ix=Limxinf; ix<Limxsup; ix++) {
for(iy=Limyinf; iy<Limysup; iy++) {
```

Cálculo de la ecuación de onda más otros términos por el método de diferencias finitas. `}}}`

*Cálculo de la función **PML** para un paso de tiempo.*

Figura 19: Volumen calculado por las funciones *stencil* y PML.



En la retropropagación el principio del cálculo de los cubos es el mismo, sin embargo, en esta ocasión las fuentes serán las trazas sísmicas y el frente de onda irá hacia atrás en el tiempo hasta llegar al origen de la misma. Como paso final el algoritmo aplica la condición de imagen a los cubos generados por el campo de la propagación y el de la retropropagación, dando como resultado la migración de un disparo.

Tanto en la propagación como en la retropropagación es necesario conocer el cubo presente (t_0) y el cubo pasado (t_{-1}) para calcular el cubo futuro (t_{+1}). Esta condición hace imposible paralelar el eje temporal, sin embargo, debido a que espacialmente los puntos del cubo son independientes uno de otro, el cálculo del cubo futuro si es paralelable. La independencia entre puntos crea la posibilidad de distribuir el cómputo del cubo en múltiples *threads*.

3.2.1. Mejora número 1: Reorganización de las sentencias *for*.

Una vez ubicados los ciclos *for* de la función *stencil* y las seis funciones relacionadas al PML, se procedió a mejorar el acceso a la memoria caché reordenando la posición de las sentencias, como se vio en la sección 2.4. En total, las siete funciones comprometidas con el cálculo del cubo futuro se modificaron de esta manera. Véase las secciones de código 3.1 y 3.2.

Sección de código 3.1: Ciclo *for* original.

```
for(iz=Limzinf; iz<Limzsup; iz++) {
  Limzn = iz;     Limzp = Nz-iz-1;
   for(iy=Limyinf; iy<Limysup; iy++) {
     Limyn = iy;     Limyp = Ny-iy-1;
      for(ix=Limxinf; ix<Limxsup; ix++) {
         Limxn = ix;     Limxp = Nx-ix-1;

         r(ix,iy,iz)=-p(ix,iy,iz)+2*q(ix,iy,iz)+dt*dt*c(ix,iy,iz)*c(ix,iy,iz)*lap;
      }
   }
}
```

Sección de código 3.2: Ciclo *for* modificado.

```
for(ix=Limxinf; ix<Limxsup; ix++) {
for(iy=Limyinf; iy<Limysup; iy++) {
for(iz=Limzinf; iz<Limzsup; iz++) {
    Limxn = ix;     Limxp = Nx-ix-1;
    Limyn = iy;     Limyp = Ny-iy-1;
    Limzn = iz;     Limzp = Nz-iz-1;

         r(ix,iy,iz)=-p(ix,iy,iz)+2*q(ix,iy,iz)+dt*dt*c(ix,iy,iz)*c(ix,iy,iz)*lap;
    }
  }
}
```

Donde `r(ix,iy,iz)` es el punto i del cubo a calcular, `p(ix,iy,iz)` es el punto i del cubo pasado, `q(ix,iy,iz)` el punto i del cubo presente, `c(ix,iy,iz)` el punto i del modelo de velocidades y `lap` el operador laplaciano de las diferencias finitas.

En el ciclo *for* original, los puntos del cubo son generados en el orden z, y, x y la función que los evalúa los procesa en el orden x, y, z, tal como se vio en la sección 2.4 esto se traduce en una mala utilización del sistema de memoria caché. Al modificar las sentencias y generar los puntos del cubo en el orden x, y, z (ciclo *for* modificado) se espera una mejora en el uso de la memoria caché y por ende un menor tiempo de ejecución en la funciones *stencil* y PML.

3.2.2. Mejora número 2: Inclusión de las directivas OpenMP.

El paso a seguir fue la aplicación de las directivas OpenMP sobre los ciclos *for*. Para ello se incluyó la librería `omp.h` en el algoritmo y se agregaron los `pragmas` correspondientes, siguiendo las indicaciones de variables compartidas, variables privadas y distribución de iteraciones entre *threads*, temas tratados en la sección 2.3. Las secciones de código 3.3 y 3.4 muestran como ejemplo la aplicación de las directivas OpenMP para la función *stencil* y para una de las funciones de PML.

Sección de código 3.3: Directiva OpenMP aplicada en la función *stencil*.

```
1  [...]
2  /*Recorrido teniendo en cuenta las distancias con el borde del modelo*/
3  #pragma omp parallel for schedule(static) private(ix,iy,iz,Limxn,Limxp,Limyn,Limyp
       ,Limzn,Limzp,lap)
4  for(ix=Limxinf; ix<Limxsup; ix++) {
5  for(iy=Limyinf; iy<Limysup; iy++) {
6  for(iz=Limzinf; iz<Limzsup; iz++) {
7      Limxn = ix;     Limxp = Nx-ix-1;
8      Limyn = iy;     Limyp = Ny-iy-1;
9      Limzn = iz;     Limzp = Nz-iz-1;
10
11     r(ix,iy,iz)=-p(ix,iy,iz)+2*q(ix,iy,iz)+dt*dt*c(ix,iy,iz)*c(ix,iy,iz)*lap;}}}
12 [...]
```

La línea número 3 de la sección de código 3.3 especifica la directiva OpenMP utilizada, para la cual se asigna un constructor combinado que crea un equipo de *threads* y establece que el bloque a paralelar será un ciclo *for* (`parallel for`). Se escoge una distribución entre *threads* estática (`schedule(static)`), pues el tiempo que tarda en calcularse un punto del cubo es aproximadamente el mismo para todos los puntos del arreglo tridimensional. Se eligen como privadas todas aquellas variables que necesitan su propia copia en cada *thread*. Por ejemplo, si no se privatiza la variable `lap` y se deja por defecto como compartida, un *thread* podría estar modificando su valor mientras que otro *thread* trataría de acceder a ella al mismo tiempo, lo que arrojaría resultados erróneos.

Las líneas 3 y 16 de la sección de código 3.4 siguen un análisis similar al descrito anteriormente, con la única excepción que en la línea 16 se debe privatizar también la variable `dpsi`.

Sección de código 3.4: Directivas OpenMP aplicadas en una de las funciones PML.

```
1  [...]
2  /* Actualizo el campo psi */
3  #pragma omp parallel for schedule(static) private(ix,iy,iz,Limxn,Limxp,Limyn,Limyp
       ,Limzn,Limzp,lap)
4  for(ix=Limxinf; ix<Limxsup; ix++) {
5  for(iy=Limyinf; iy<Limysup; iy++) {
6  for(iz=Limzinf; iz<Limzsup; iz++) {
7      Limxn = -1;     Limxp = -1;
8      Limyn = iy+Ny-L-1;     Limyp = Ny-iy-1-Ny+L+1;
9      Limzn = -1;     Limzp = -1;
10
11     Limyn = iy+Ny-L;     Lim = Ny-iy-1-Ny+L;
12
13     psi.front(ix,iy,iz)=b_y_h[iy]*lap+a_y_h[iy]*psi.front(ix,iy,iz);}}}
14
15 /* Actualizo el campo zeta y sumo los terminos adicionales al campo */
16 #pragma omp parallel for schedule(static) private(ix,iy,iz,Limxn,Limxp,Limyn,Limyp
       ,Limzn,Limzp,dpsi,lap)
17 for(ix=Limxinf; ix<Limxsup; ix++) {
18 for(iy=Limyinf; iy<Limysup; iy++) {
19 for(iz=Limzinf; iz<Limzsup; iz++) {
20     Limxn = -1;     Limxp = -1;
21     Limyn = iy;     Limyp = L-iy-1;
22     Limzn = -1;     Limzp = -1;
23
24     Limyn = iy+Ny-L;     Lim = Ny-iy-1-Ny+L;
25
26     zeta.front(ix,iy,iz)=b_y[iy]*(lap+dpsi)+a_y[iy]*zeta.front(ix,iy,iz);
27     r(ix,Ny-L+iy,iz)+=dt*dt*c(ix,Ny-L+iy,iz)*(ix,Ny-L+iy,iz)*(zeta.front(ix,iy,iz)
       +dpsi);}}}
28 [...]
```

Gran porcentaje del tiempo de ejecución del algoritmo se encuentra en las secciones de código paraleladas con OpenMP, fue por ello que fueron el principal objetivo de estudio.

Capítulo 4

PRUEBAS Y RESULTADOS

Para el desarrollo de las pruebas se usaron tres sistemas de cómputo. A continuación se indican sus características, se presentan en orden ascendente por número de núcleos.

- Sistema de cómputo **osaka**
  - 2 Procesadores Intel Xeon E5-2609 v2 @ 2.5 GHz
  - 256 GB de memoria RAM
    Cada procesador posee:
    4 núcleos sin tecnología Hyper-Threading (para un total de 8 núcleos físicos entre ambos procesadores), memoria caché: L1 de 256 KB, L2 de 1024 KB y L3 de 10 MB.
- Sistema de cómputo **tokyo_01**
  - 2 Procesadores Intel Xeon E5-2620 v3 @ 2.4 GHz
  - 256 GB de memoria RAM
    Cada procesador posee:
    6 núcleos con tecnología Hyper-Threading (para un total de 24 núcleos virtuales entre ambos procesadores), memoria caché: L1 de 384 KB, L2 de 1536 KB y L3 de 15 MB.
- Sistema de cómputo **tokyo_00**
  - 2 Procesadores Intel Xeon E5-2670 v3 @ 2.3 GHz
  - 64 GB de memoria RAM
    Cada procesador posee:
    12 núcleos con tecnología Hyper-Threading (para un total de 48 núcleos virtuales entre ambos procesadores), memoria caché: L1 de 768 KB, L2 de 3072 KB y L3 de 30 MB.

4.1. CANTIDAD DE ACCESOS A LA MEMORIA CACHÉ

Para entender y analizar el impacto de la reorganización de las sentencias de los ciclos *for*, se realizaron dos pruebas de un disparo para un modelo con una resolución de 100 x 100 x 100 puntos sin usar OpenMP. La primera de ellas sin la mejora del acceso a memoria (algoritmo original) y la segunda con la mejora realizada. La tabla 1 muestra los resultados en los sistemas de cómputo **tokyo_01** y **tokyo_00**. Para la evaluación del rendimiento se usó el comando `perf` de Linux disponible en el paquete `linux-tools`, donde:

- `cycles`: es el número total de ciclos de CPU.
- `cache-references`: es la cantidad de accesos a la memoria caché de último nivel.
- `cache-misses`: es la cantidad de datos no encontrados en la memoria caché de último nivel.
- `L1-dcache-loads`: es la cantidad de copias realizadas a la memoria caché de primer nivel.
- `L1-dcache-misses`: es la cantidad de datos no encontrados en la memoria caché de primer nivel.
- `dTLB-cache-misses`: es la cantidad de datos no encontrados en la memoria caché de direcciones.

Tabla 1: Acceso a memoria caché para el algoritmo original y el modificado.

Rendimiento	**tokyo_01**		**tokyo_00**	
	Original	**Modificado**	**Original**	**Modificado**
`cycles`	$3900{,}78 \times 10^9$	$2844{,}24 \times 10^9$	$3853{,}55 \times 10^9$	$2841{,}83 \times 10^9$
`cache-references`	$46{,}51 \times 10^9$	$0{,}82 \times 10^9$	$50{,}09 \times 10^9$	$0{,}88 \times 10^9$
`cache-misses`	$2{,}37 \times 10^9$	$0{,}33 \times 10^9$	$1{,}66 \times 10^9$	$0{,}18 \times 10^9$
`L1-cache-loads`	$3221{,}52 \times 10^9$	$3255{,}95 \times 10^9$	$3231{,}51 \times 10^9$	$3255{,}49 \times 10^9$
`L1-cache-misses`	$93{,}21 \times 10^9$	$10{,}45 \times 10^9$	$94{,}54 \times 10^9$	$10{,}35 \times 10^9$
`dTLB-cache-misses`	$4{,}20 \times 10^9$	$0{,}04 \times 10^9$	$4{,}36 \times 10^9$	$0{,}06 \times 10^9$

Se observa que el número de ciclos de CPU es menor para el algoritmo modificado, lo que asegura un menor tiempo de ejecución. Aunque el algoritmo modificado tiene más operaciones de copia, tiene menor cantidad de datos no encontrados, presenta además una importante disminución en la cantidad de accesos a la memoria caché de direcciones (TLB). La mejora implementada confirma una mayor eficiencia en el acceso a memoria caché de último nivel. En general la cantidad de accesos a caché es significativamente menor en el algoritmo modificado.

4.2. TIEMPO DE EJECUCIÓN DE UN DISPARO VARIANDO EL NÚMERO *THREADS*

Para un modelo con una resolución de 100 x 100 x 100 puntos, se midió el tiempo total de ejecución de un disparo variando el número de *threads* utilizados. Además, se tomaron los tiempos de ejecución de la función *stencil* y el tiempo total de las seis funciones PML. Todas las medidas fueron comparadas con su equivalente serial mejorado (mejora de acceso a memoria sin OpenMP visto en la sección 4.1). Se midieron también los mismos tiempos para el algoritmo original, referido a continuación como 1(s), el cual no posee modificación alguna.

Para la correcta compilación del algoritmo fue necesario agregar las banderas `-Xcompiler` y `-fopenmp` en la sección de banderas de CUDA del archivo compilador o *Makefile*. Se debió añadir también la librería `-lgomp` en el apartado de creación de programas del mismo archivo compilador, véase la sección de código 4.1. Para variar el número de *threads* a usar en las pruebas, se utilizó la variable de entorno de OpenMP `export OMP_NUM_THREADS=<número de threads a utilizar>`. Si no se utiliza la variable de entorno o algún equivalente que especifique el número de *threads* a usar, la ejecución del algoritmo hará uso del máximo número de *threads*.

Sección de código 4.1: Modificación al archivo *Makefile*.

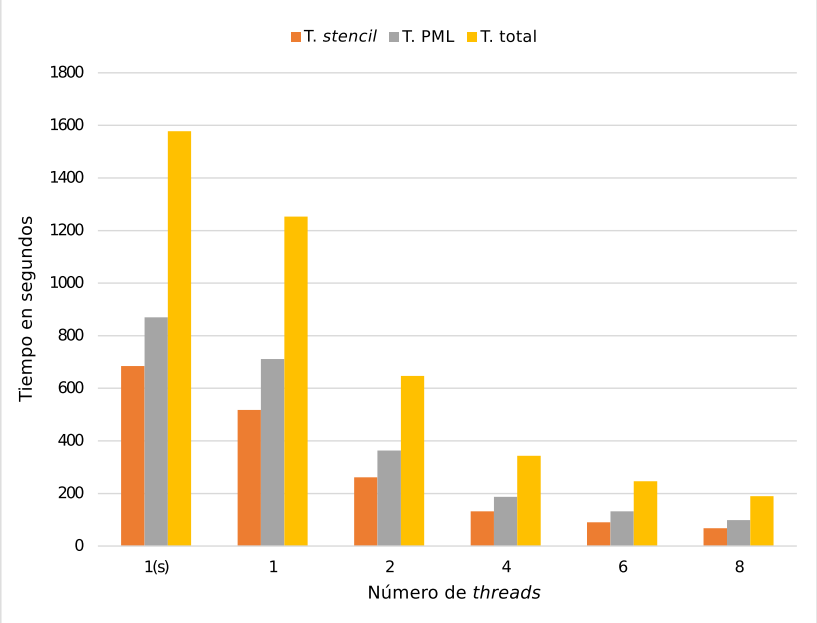
```
[...]
CUFLAGS:= $(CUFLAGS) -Xcompiler -fopenmp -g -c -I $(H_DIR)/ $(CDEBUGFLAGS) $(
    DONT_OPT) $(CU_ARCH) $(CUDEBUGFLAGS) -Xptxas=-v -maxrregcount=32
[...]
LOPENMP = -lgomp
[...]
$(PROGS): $(OBJECTS) $(addprefix $(OBJ_DIR)/, $(PROGS:bin/%=%.o)) $D
        @echo "============================"
        @echo "Creating the programs    ..."
        @echo "============================"
        @echo "Building $(B)/$@"
        @mkdir -p $(B)
        $(NVCC) -g -L $(VTKLIB_PATH) $(VTKLIBS) $(OBJECTS) $(OBJ_DIR)/$(@F).o $(
    LFLAGS) $(LOPENMP) -o $@ -lm $(CU_ARCH)
        @echo $(@F) installed in $B
[...]
```

Cada una de las pruebas realizadas fueron ejecutadas cinco veces. La desviación estándar mostró que no existieron variaciones significativas. Experimentalmente la máxima variación presentada en los tiempos de ejecución fue de 10 segundos.

Tabla 2: Tiempo de un disparo en **osaka** variando el número de *threads*.

threads	**T. *stencil* [s]**	**T. PML [s]**	**T. total [s]**	***Speed-up***
1(s)	685	870	1578	-
1	518	711	1253	-
2	261	363	647	1.94x
4	132	187	343	3.65x
6	90	132	246	5.09x
8	68	99	190	6.59x

Figura 20: Tiempo de un disparo en **osaka** variando el número de *threads*.



Las dos primeras ejecuciones de la tabla 2 son secuenciales (solo utilizan un *thread*), sin embargo el tiempo de cómputo del algoritmo se reduce en aproximadamente 5:25 minutos (de 1578 a 1253 segundos) con la mejora de acceso a memoria.

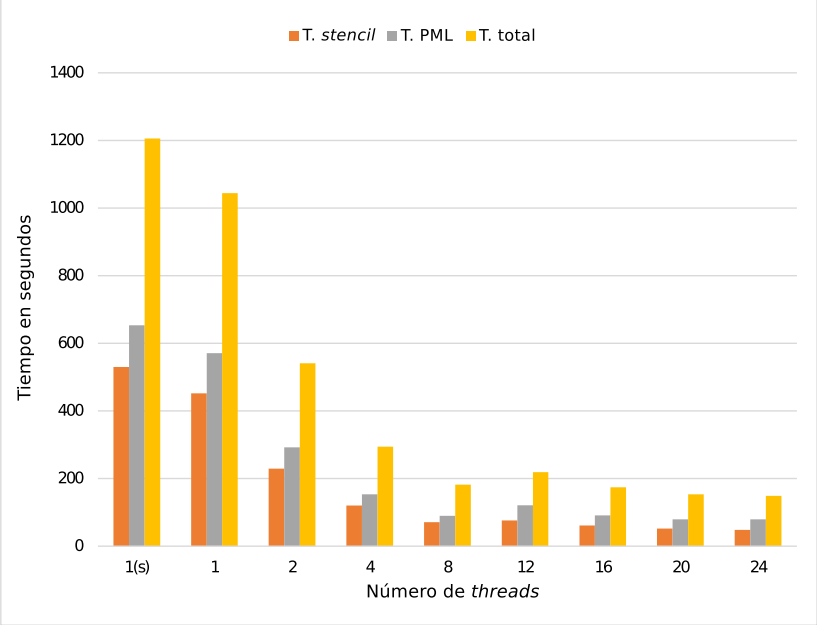
Como era de esperarse, a medida que se aumenta el número de *threads*, el tiempo de ejecución total del algoritmo va reduciéndose. Teniendo en cuenta ambos extremos de la tabla 2, el tiempo de ejecución

de un disparo en **osaka** pasó de 26:18 minutos sin modificación alguna a 3:10 minutos con la mejora de acceso a memoria y utilizando el máximo número de *threads*, lo que conlleva un *speed-up* máximo de 6.59x.

Tabla 3: Tiempo de un disparo en **tokyo_01** variando el número de *threads*.

threads	**T. *stencil* [s]**	**T. PML [s]**	**T. total [s]**	***Speed-up***
1(s)	530	653	1206	-
1	452	571	1044	-
2	229	292	541	1.93x
4	120	153	294	3.55x
8	71	90	182	5.74x
12	76	121	219	4.77x
16	61	91	174	6.00x
20	52	79	153	6.82x
24	48	79	149	7.01x

Figura 21: Tiempo de un disparo en **tokyo_01** variando el número de *threads*.



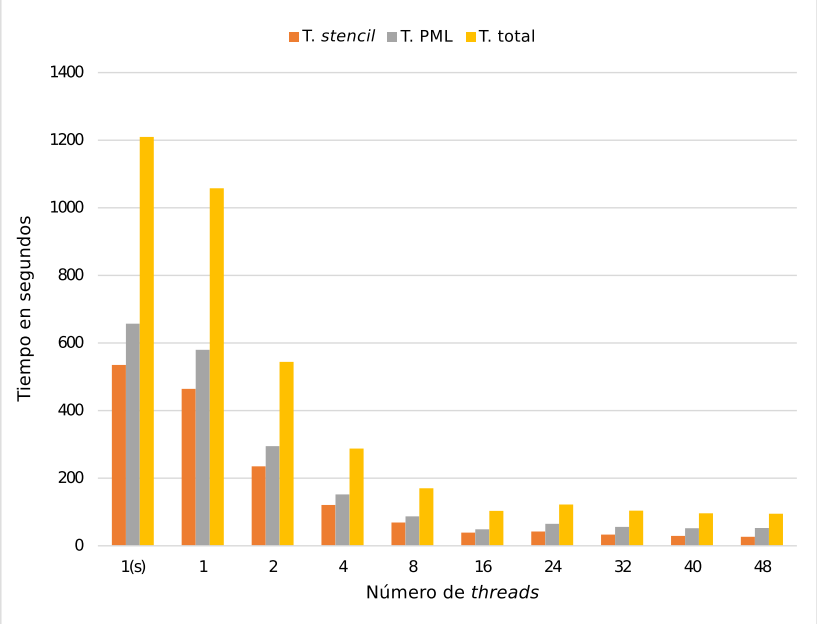
tokyo_01 tiene un comportamiento similar al de **osaka** en el cómputo del código. Reduciendo en 2:42 minutos el tiempo de ejecución con la mejora de acceso a memoria. Teniendo en cuenta ambos extremos de la tabla 3, el tiempo de ejecución de un disparo en **tokyo_01** pasó de 20:06 minutos sin ninguna

modificación a 2:29 minutos aproximadamente, con la mejora de acceso a memoria y utilizando el máximo número de *threads*, para un *speed-up* máximo de 7.01x.

Tabla 4: Tiempo de un disparo en **tokyo_00** variando el número de *threads*.

threads	**T. *stencil* [s]**	**T. PML [s]**	**T. total [s]**	***Speed-up***
1(s)	535	657	1210	-
1	464	580	1058	-
2	235	295	544	1.94x
4	121	152	288	3.67x
8	69	87	170	6.22x
16	39	49	103	10.27x
24	42	65	122	8.67x
32	33	56	104	10,17x
40	29	52	96	11.02x
48	27	53	95	11.14x

Figura 22: Tiempo de un disparo en **tokyo_00** variando el número de *threads*.



La tabla 4 muestra el máximo *speed-up* de todas las pruebas (11.14x), pues **tokyo_00** posee la mayor

capacidad de procesamiento entre los sistemas de cómputo utilizados. El tiempo de ejecución de un disparo en **tokyo_00** pasó de 20:10 minutos sin modificación alguna a 1:35 minutos mejorando el acceso a memoria y utilizando el máximo número de *threads*.
Al aumentar el número de *threads* se registró un comportamiento exponencial decreciente en todas las pruebas realizadas. Para los procesadores con tecnología Hyper-Threading (**tokyo_01** y **tokyo_00**) se aprecia una conducta inusual cuando el número de *threads* iguala el número de núcleos físicos, nótese que en esos valores el *speed-up* en las tablas 3 y 4 disminuye respecto a su antecesor y no aumenta como se esperaría. Estudios y análisis de procesadores con tecnología Hyper-Threading muestran resultados simliares e incluso más variantes al aumentar el número de *threads* [20], siendo esta condición propia del hardware del procesador.

Sin embargo, está claro que el máximo rendimiento con OpenMP para este algoritmo se presenta cuando el número de *threads* utilizados sea igual al número total de *threads* disponibles (independientemente si el procesador es compatible o no con la tecnología Hyper-Threading).

4.2.1. Aplicación de la Ley de Amdahl

Una consecuencia directa de la medición de todos los tiempos registrados es la posible aplicación de la Ley de Amdahl. En cada prueba se obtuvo el tiempo total que tardó la ejecución del algoritmo y los tiempos que tardaron las funciones paraleladas, por lo tanto se tiene la porción serial y la porción paralela del código. Con la Ley de Amdahl se puede establecer una comparación entre el tiempo ideal ($t_{teórico}$) y el tiempo que realmente tomó la ejecución del algoritmo (t_{real}). A continuación se presenta un ejemplo de cómo fue calculado el tiempo teórico y el *speed-up* máximo para 6 *threads* en **osaka**.

Refiérase a la tabla 2, el tiempo de la porción paralela se halló sumando los tiempos de las funciones paraleladas: 90 segundos de la función *stencil* y 132 segundos de la función PML, para un total de 222 segundos. El tiempo de la porción serial se calculó restándole al tiempo total del algoritmo el tiempo de la porción paralela calculado anteriormente (ecuación 4.1).

$$\begin{aligned} t_s &= t_{real} - t_p \\ t_s &= 246\ [s] - 222\ [s] \\ t_s &= 24\ [s] \end{aligned} \tag{4.1}$$

Teniendo el tiempo de las porciones serial y paralela, se calculó el tiempo teórico con la ecuación 2.2.

$$
\begin{aligned}
t_{teórico} &= t_s + \frac{t_p}{n} \\
t_{teórico} &= 24\ [s] + \frac{222\ [s]}{6} \\
t_{teórico} &= 61\ [s]
\end{aligned}
\tag{4.2}
$$

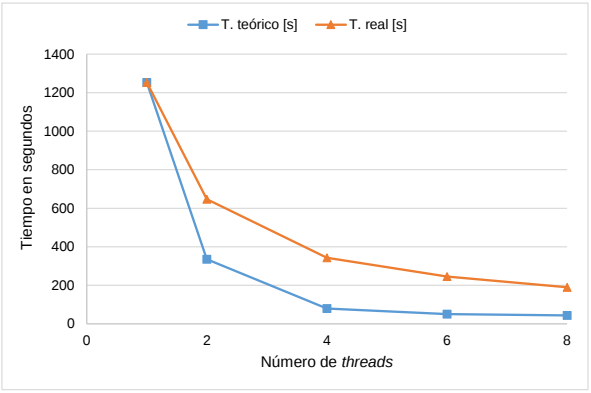
Para hallar el *speed-up* máximo se tomó el tiempo real que le tomó al algoritmo ejecutarse en un *thread* y se aplicó la ecuación 2.3.

$$
\begin{aligned}
speed\text{-}up_{máximo} &= \frac{(t_{real})\ _{1\ thread}}{t_s} \\
speed\text{-}up_{máximo} &= \frac{1253\ [s]}{24\ [s]} \\
speed\text{-}up_{máximo} &= 52x
\end{aligned}
\tag{4.3}
$$

A continuación se tabulan y se ilustran los resultados obtenidos para los tres sistemas de cómputo.

Figura 23: Ley de Amdahl para las pruebas en **osaka**.

osaka	$speed\text{-}up_{máximo} = 52x$			
threads	t_s [s]	t_p [s]	$t_{teórico}$ [s]	t_{real} [s]
1	24	1229	1253	1253
2	23	624	335	647
4	24	319	80	343
6	24	222	51	246
8	23	167	44	190



Es interesante observar como el tiempo de la porción paralela se ve significativamente reducido, mientras que el de la porción serial se mantiene relativamente constante. Dado este comportamiento, puede concluirse que existe una parte del algoritmo imposible de paralelar y cuyo tiempo de ejecución no se ve alterado al aumentar el número de *threads*. Además reafirma que el tiempo de las funciones paraleladas tienen un gran porcentaje dentro del tiempo de ejecución total del algoritmo.

Figura 24: Ley de Amdahl para **tokyo_01**.

tokyo_01	$speed{-}up_{máximo} = 50x$			
threads	t_s [s]	t_p [s]	$t_{teórico}$ [s]	t_{real} [s]
1	21	1023	1044	1044
2	20	521	281	541
4	21	273	61	294
8	21	161	46	182
12	21	197	37	218
16	21	152	31	173
20	21	131	28	152
24	22	127	27	149

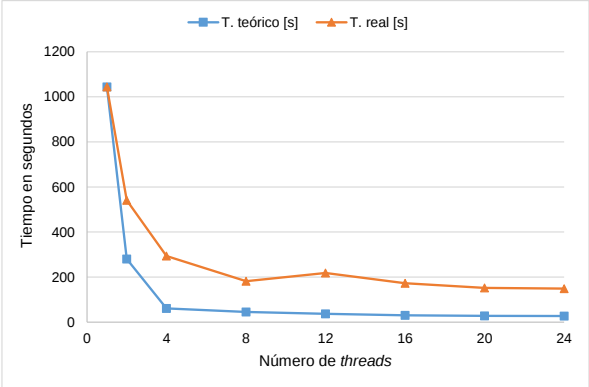
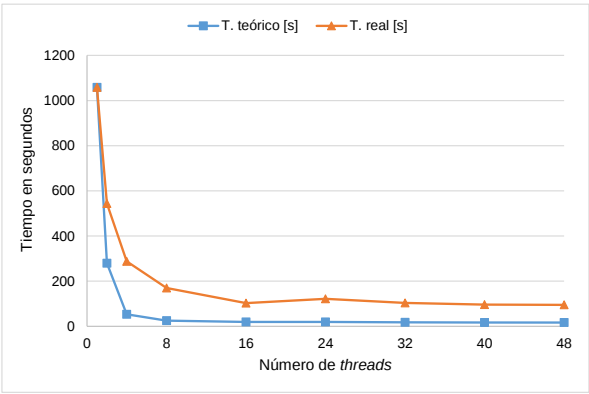


Figura 25: Ley de Amdahl para las pruebas en **tokyo_00**.

tokyo_00	$speed{-}up_{máximo} = 76x$			
threads	t_s [s]	t_p [s]	$t_{teórico}$ [s]	t_{real} [s]
1	14	1044	1058	1058
2	15	529	280	544
4	15	273	53	288
8	14	156	25	170
16	14	89	20	103
24	15	107	19	122
32	15	89	18	104
40	15	81	17	96
48	15	80	17	95



El tiempo teórico según la Ley de Amdahl no tiene en cuenta los tiempos de demora debidos a la competencia de acceso a memoria (cuellos de botella), no contempla el tiempo que tarda el sistema operativo en solicitar al procesador la ejecución de una instrucción o el lanzamiento de un *thread*, en otras palabras, la Ley de Amdahl asume que la ejecución del algoritmo se realiza en un sistema de cómputo ideal. Para el cálculo del *speed-up* máximo se asume además que el número de *threads* es infinito.

En las Figuras 23, 24 y 25 se puede apreciar que los tiempos teóricos y reales muestran un comportamiento exponencial decreciente a medida que el número de *threads* aumenta. Como es de esperarse el tiempo real de la ejecución siempre será mayor al tiempo teórico.

4.3. TIEMPO DE EJECUCIÓN DE UN DISPARO VARIANDO LA RESOLUCIÓN DEL MODELO

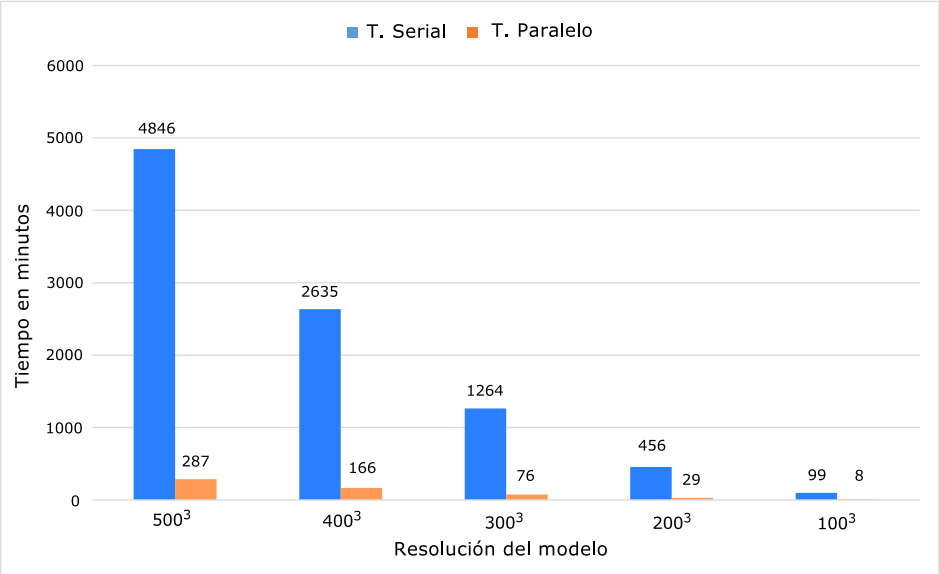
Utilizando **tokyo_00** a la máxima capacidad de cómputo de CPU (48 *threads*), se hicieron pruebas para un disparo variando la resolución del modelo. Una resolución mayor implica un mayor requerimiento de memoria y un mayor tiempo de ejecución. Se inició con un modelo de 100 x 100 x 100 puntos. Posteriormente se fue aumentando de forma equidistante en 100 hasta llegar a un modelo de 500 x 500 x 500 puntos.

Para asegurar que los pasos de tiempo entre pruebas fueran los mismos sin importar la resolución del modelo, se realizó un ajuste de estabilidad en el algoritmo. Esto implicó que el mismo modelo de 100 x 100 x 100 puntos calculado en la sección de pruebas anterior tardara más tiempo en ejecutarse. Se comparó el tiempo de ejecución total en paralelo con su equivalente serial mejorado. Los resultados se muestran en la tabla 5 y la Figura 26.

Tabla 5: Tiempo de un disparo en **tokyo_00** variando la resolución del modelo.

Resolución del modelo	**T. Serial [min]**	**Equivalente en días horas y minutos**	**T. Paralelo [min]**	**Equivalente en días horas y minutos**	***Speed-up***
500^3	4846	3 d 8 h 45 min	287	4 h 47 min	16.9x
400^3	2635	1 d 19 h 54 min	166	2 h 46 min	15.9x
300^3	1264	21 h 3 min	76	1 h 16 min	16.6x
200^3	456	7 h 36 min	29	29 min	15.7x
100^3	99	1 h 39 min	8	8 min	12.5x

Figura 26: Tiempo de un disparo en **tokyo_00** variando la resolución del modelo.

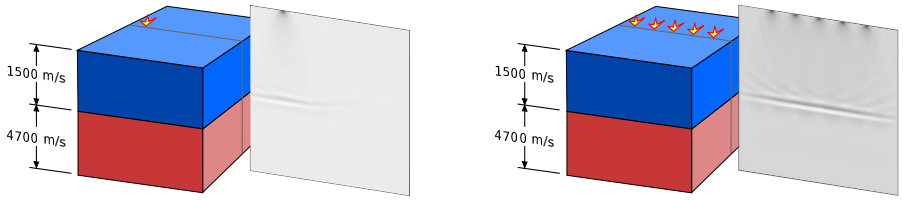


Se aprecia una mejora importante en los tiempos de ejecución en paralelo, reduciendo en casi 17 veces el tiempo de cómputo de un disparo para una resolución de 500 x 500 x 500 puntos. El *speed-up* de todas las pruebas estuvo por encima de 12.5x, al aumentar la resolución del modelo la parte serial del código se hace menos significativa. Por ende, aumenta la parte paralelable y con esta el *speed-up*.

4.4. TIEMPO DE EJECUCIÓN DE MÚLTIPLES DISPAROS

Cada disparo aporta a la migración final una sección específica del modelo. Por lo tanto, al incrementar la cantidad de disparos realizados aumenta la calidad del modelo final migrado. La Figura 27 ilustra la sección transversal de la migración realizada en un modelo sintético de dos capas para uno y cinco disparos. Nótese cómo influye el número total de disparos migrados en el resultado final del modelo.

Figura 27: Sección transversal de la migración RTM 3D.



(a) 1 disparo (b) 5 disparos

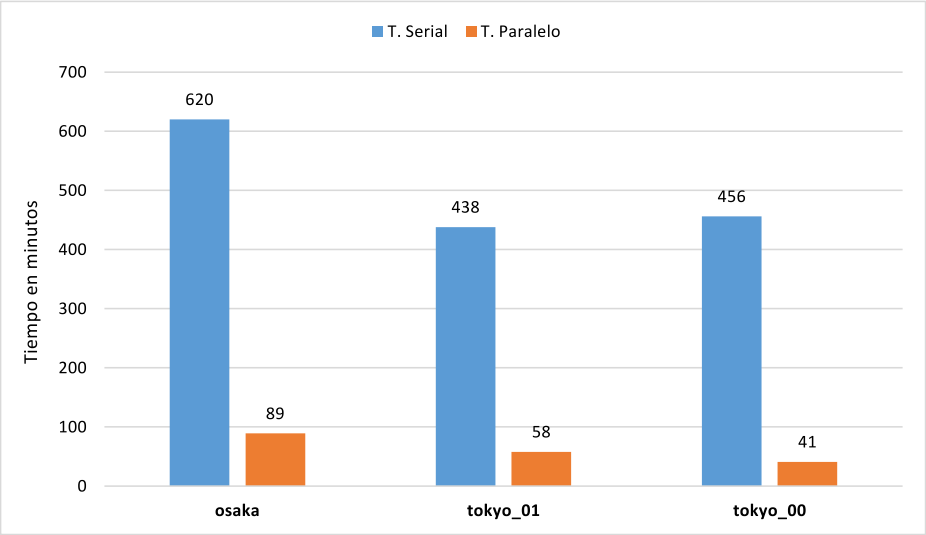
Para mejorar el resultado de la migración los disparos deben distribuirse en forma de cuadrícula, de tal manera que abarquen toda la superficie.

Se realizaron un total de 25 disparos distribuidos en una cuadrícula de 5 x 5 para el modelo con una resolución de 100 x 100 x 100 puntos y se tomaron los tiempos de ejecución total en paralelo y serial mejorado. Para la prueba en paralelo se utlizaron los tres sistemas de cómputo a máxima capacidad. La tabla 6 y la Figura 28 muestran los resultados.

Tabla 6: Tiempo de ejecución del algoritmo RTM 3D para 25 disparos.

Sistema de cómputo	**T. Serial [min]**	**T. Paralelo [min]**	***Speed-up***
osaka	620	89	6.96x
tokyo_01	438	58	7.55x
tokyo_00	456	41	11.12x

Figura 28: Tiempo de ejecución del algoritmo RTM 3D para 25 disparos.



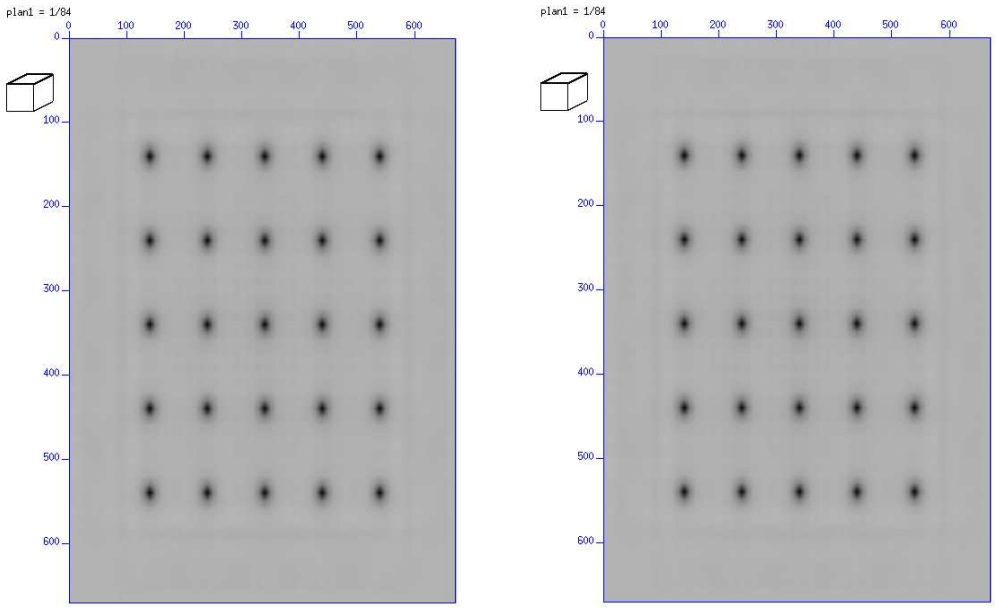
Al igual que en las pruebas anteriores se observa una mejora significativa en el tiempo de cómputo del algoritmo paralelado. En comparación para cada sistema de cómputo, el tiempo total de ejecución de esta prueba tarda aproximadamente 25 veces más que el tiempo de ejecución de un solo disparo, manteniendo una relación lineal. Si se compara esta prueba con su equivalente de un disparo estudiado en la sección 4.2, se aprecian leves mejoras en el *speed-up* de **osaka** y **tokyo_01**, mientras que en **tokyo_00** permanece casi constante.

Capítulo 5

VALIDACIÓN DEL ALGORITMO

Usando **tokyo_00** se realizaron dos pruebas de 25 disparos en un modelo sintético de dos capas con una resolución de 100 x 100 x 100 puntos. Utilizando un software de visualización se evaluó el resultado final del algoritmo para la prueba original y la modificada con OpenMP. En primera instancia se realizó una comparación visual entre ambas. El software empleado para ver el modelo ofrece la posibilidad de ir observando por secciones transversales el resultado de la migración en tres vistas diferentes; superior, frontal y lateral. Las Figuras 29, 30 y 31 muestran a simple vista que ambos resultados son similares.

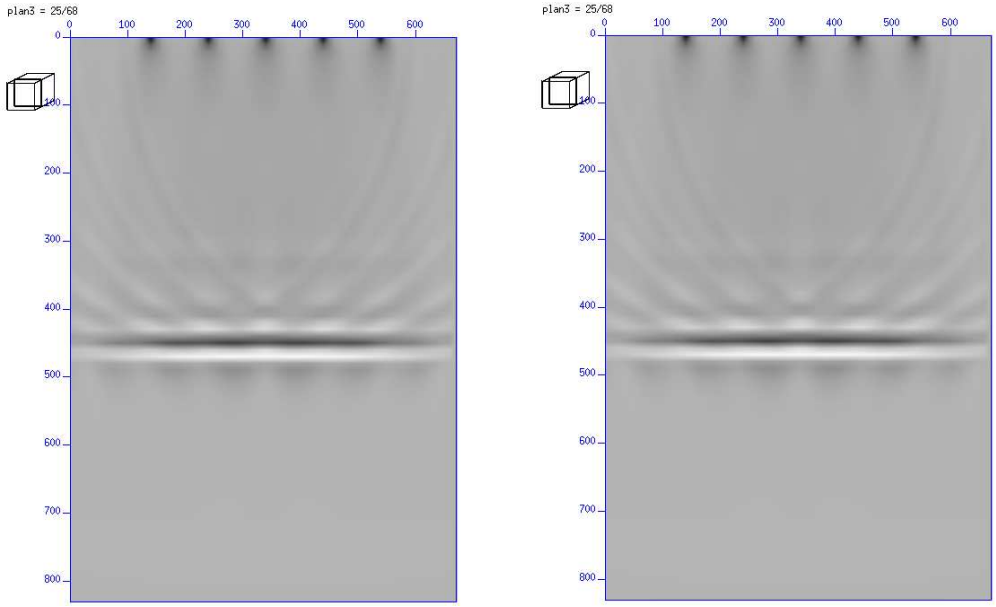
Figura 29: Vista superior del algoritmo original y el desarrollado con OpenMP.



(a) Algoritmo original

(b) Algoritmo con OpenMP

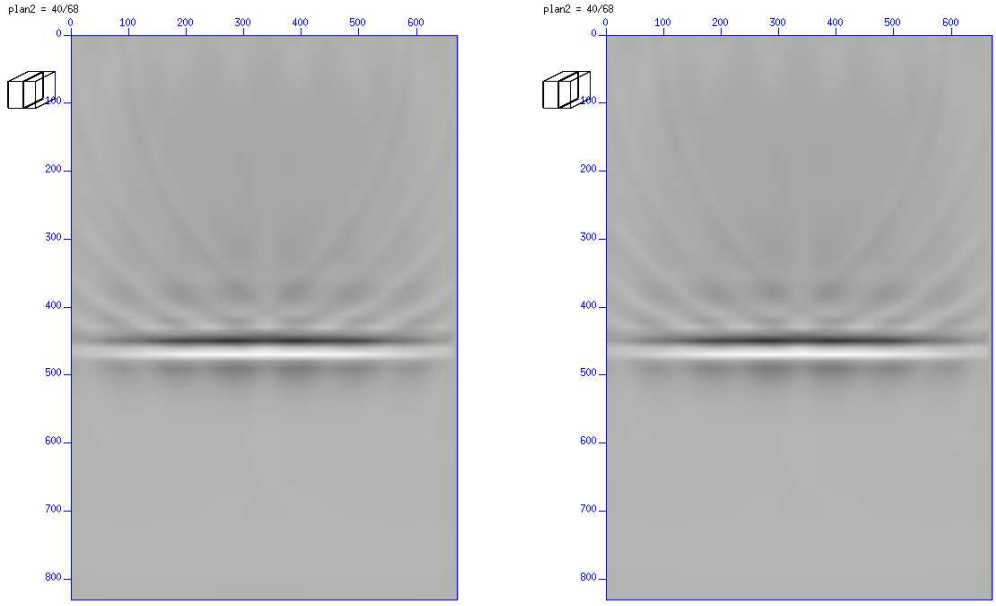
Figura 30: Vista frontal del algoritmo original y el desarrollado con OpenMP.



(a) Algoritmo original

(b) Algoritmo con OpenMP

Figura 31: Vista lateral del algoritmo original y el desarrollado con OpenMP.



(a) Algoritmo original

(b) Algoritmo con OpenMP

Posteriormente se procedió a realizar una comparación más detallada hallando un error numérico. Para esto se restaron uno a uno los puntos del arreglo tridimensional de ambas ejecuciones, dando como resultado un error de **0.0**. Esto confirmó que tanto el algoritmo original como el modificado con OpenMP proyectan exactamente el mismo modelo migrado.

Además de ser ideal, se entiende que el error arrojado por la comparación de los modelos migrados de ambas ejecuciones sea cero. Pues el algoritmo fue ejecutado por el mismo procesador, por lo tanto las unidades especializadas en el desarrollo de operaciones aritméticas son las mismas para ambos casos. El paso a seguir fue la comparación del mismo modelo migrado, pero en esta oportunidad teniendo en cuenta el resultado de la ejecución en diferentes sistemas de cómputo, véase la tabla 7.

Tabla 7: Cálculo de error del algoritmo RTM 3D original y modificado con OpenMP.

Comparación entre		**error**
Original	**Con OpenMP**	
osaka	osaka	0.0
tokyo_01	tokyo_01	0.0
tokyo_00	tokyo_00	0.0
osaka	tokyo_01	0.0
tokyo_01	tokyo_00	0.0

En esta ocasión el resultado fue el mismo, la comparación de la migración realizada en **osaka**, en **tokyo_01** y en **tokyo_00** arrojó un error de cero, por lo tanto se puede concluir que los procesadores de los tres sistemas de cómputo tienen las mismas unidades encargadas de desarrollar operaciones aritméticas.

5.1. COMPARACIÓN ENTRE CPU Y GPU

Habiendo sacado el máximo provecho a los procesadores de cada sistema de cómputo, se comparó el tiempo de ejecución del algoritmo RTM 3D entre CPU y GPU para una misma prueba. El grupo de investigación CPS suministró el tiempo de la implementación en GPU para la prueba de múltiples disparos desarrollada en la sección 4.4, utilizando los mismos parámetros. La tabla 8 resume los resultados de la comparación. El *speed-up* se calculó como la razón entre el tiempo en paralelo de CPU y el tiempo en paralelo de GPU.

Tabla 8: Tiempos de ejecución de 25 disparos para CPU y GPU.

Sistema	**T. Paralelo CPU**	**T. Paralelo GPU**	***Speed-up***
osaka	89 min (1 h 29 min)		3.7x
tokyo_01	58 min	24 min	2.4x
tokyo_00	41 min		1.7x

La GPU utilizada para implementar el algoritmo fue la NVIDIA Tesla K40M cuyas especificaciones de consumo de potencia y costo en dólares a la fecha (10/10/16) se muestran en la tabla 9 junto con el equivalente para los procesadores de los tres sistemas de cómputo. Los precios fueron consultados en la página de ventas de DELL [7], para el consumo de potencia se tuvieron en cuenta las especificaciones propias de cada dispositivo.

Tabla 9: Comparativa de costos y consumo de potencia de los dispositivos usados.

	Dispositivo	**Potencia [W]**	**Costo [USD]**	**Cant.**	**Potencia total [W]**	**Costo total [USD]**
	GPU Nvidia K40M	235	$ 6,999.99	1	235	$ 6,999.99
osaka	CPU Xeon E5-2609 v2	80	$ 599.99	2	160	$ 1,199.98
tokyo_01	CPU Xeon E5-2620 v3	85	$ 777.99	2	170	$ 1,555.98
tokyo_00	CPU Xeon E5-2670 v3	120	$ 2,599.99	2	240	$ 5,199.99

Tal como se muestra en la tabla 8, incluso a máxima capacidad de cómputo, ningún procesador reduce tanto el tiempo de ejecución como lo hace la GPU. Los procesadores tienen un menor precio en el mercado comparado con el precio de la GPU, al igual que un menor consumo de potencia con excepción de **tokyo_00**. Con estos resultados se puede realizar una comparación objetiva entre el tiempo de ejecución de CPU y GPU, es decir, se tiene en cuenta que las pruebas fueron realizadas a la máxima capacidad de procesamiento en ambas plataformas.

Cabe resaltar que un procesador puede funcionar perfectamente sin una GPU externa como la NVIDIA Tesla K40M, pero es imposible que una GPU funcione sin un procesador. Por lo tanto es necesario agregar la potencia consumida por el procesador a la potencia consumida por la GPU. La relación costo beneficio puede verse afectada de manera drástica si se tiene en cuenta esta variante.

Capítulo 6

CONCLUSIONES

Se realizó la implementación del algoritmo RTM 3D mediante el uso de OpenMP obteniendo una aceleración máxima de 16.9 veces en el tiempo de ejecución. Se pudo observar que el algoritmo RTM 3D original y el mejorado (mejora de acceso a memoria y utilización de OpenMP), dan como resultado exactamente el mismo modelo migrado. Hubo una mejora importante en los tiempos de ejecución de todas las pruebas usando OpenMP, incluso sin la utilización de OpenMP, la mejora en el acceso a la memoria caché presenta una reducción significativa en el tiempo de ejecución del algoritmo.

Se logró establecer que el máximo desempeño posible para el algoritmo RTM 3D usando OpenMP, tiene lugar cuando se utiliza el máximo número de *threads* de los que dispone la CPU en la cual se implementa. A diferencia de la implementación en CPU secuencial de la cual se partió, la desarrollada en este trabajo utiliza toda la capacidad de cómputo de un procesador. Por ende, puede realizarse una comparación objetiva del tiempo de ejecución, el costo y el consumo de potencia entre CPU y GPU.

DISCUSIÓN Y TRABAJO FUTURO

El desarrollador de aplicaciones OpenMP ofrece una alternativa muy interesante al dar la posibilidad de paralelar un código secuencial en CPU de manera sencilla. Los conocimientos necesarios pueden adquirirse con relativa rapidez y si se aplican de forma correcta los resultados obtenidos son siempre provechosos. El algoritmo que se deseé paralelar debe ser estudiado a fondo, para interpretar, analizar y establecer correctamente el uso de las directivas de OpenMP. Con la ayuda de la documentación aquí presentada, se espera puedan realizarse implementaciones más rápidas utilizando múltiples ordenadores, es decir, llevar OpenMP no solo a la CPU de un sistema de cómputo, sino a todos los procesadores de varios sistemas de cómputo trabajando conjuntamente en paralelo.

Fue gratificante ser pionero en la investigación y aplicación del paralelado del algoritmo RTM 3D en CPU para el grupo de investigación CPS de la Universidad Industrial de Santander.

REFERENCIAS

[1] CHANDY, J.A.; SINGARAJU, J., "Hardware parallelism vs. software parallelism," *Proceedings of the First USENIX conference*. USENIX Association. 2009.

[2] CHANDRAKASAN, A.P.; POTKONJAK, M.; MEHRA, R.; RABAEY, J.; BRODERSEN, R.W., "Optimizing power using transformations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 1995. vol.14. no.1, p.12-31.

[3] GUERRERO, Juan F.; FRANCÉS, José V., "Multiprocesadores Simétricos (SMP)," Escuela Técnica Superior de Ingeniería. Universidad de Valencia. 2010.

[4] CHAPMAN, B.; JOST, G.; VAN DER PAS, Ruud., "Using OpenMP: Portable Shared Memory Parallel Programming," *The MIT Press*. 2008, p. [4-6, 23-26, 64-72, 79-82, 125-127].

[5] VARGAS, Miguel, "Cómputo en paralelo con OpenMP," CIMAT. 2015.

[6] Welcome to the OpenMP Architecture Review Board (ARB) [en línea]. Disponible en: `http://openmp.org/mp-documents/OpenMP-WelcomeGuide.pdf`

[7] DELL Search [en línea]. Disponible en: `http://pilot.search.dell.com/`

[8] MARR, J.; DEBORAH, T.; FRANK and HILL, et al., "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal*. 2002. vol 6.

[9] FAJARDO, Carlos; CASTILLO, Javier; PEDRAZA, César, "Reducción de los tiempos de cómputo de la Migración Sísmica usando FPGAs y GPGPUs: Un artículo de revisión," *Ingeniería y Ciencia*, vol. 9, no. 17, pp. 261–293, 2013.

[10] LIU, Lanbo; LIU, Zijian, "Application of Iterative Reverse Time Migration Procedure on Transcranial Thermoacoustic Tomography Imaging," School of Engineering, University of Connecticut, 2014.

[11] CABEZAS, J.; ARAYA-POLO, M.; GELADO, I.; NAVARRO, N.; MORANCHO, E. and CELA J.M., "High- Performance Reverse Time Migration on GPU," *International Conference of the Chilean Computer Science Society*, 2009, pp. 77–86.

[12] BAYSAL, E., "Reverse time migration," *Geophysics*, 1983, vol. 48, p. 1514-1524.

[13] SAVA, Paul; FOMEL, Sergey, "Generalized imaging conditions for wave-equation migration," Colorado School of Mines Meeting. 2002.

[14] SUN, Lin-Jie; YIN, Xing-Yao, "A Finite-Difference Scheme Based on PML Boundary Condition with High Power Grid Step Variation," *Chinese Journal of Geophysics*. 2011. no 3, p. 393-402.

[15] GRAY, Samuel H.; ETGEN, John; DELLINGER, Joe; WHITMORE, Dan. "Seismic migration problems and solutions," *GEOPHYSICS*, 2001, vol 66, no. 5, p. 1622-1640.

[16] FARMER, P.; GRAY, S.; HODGKISS, G.; PIEPRZAK, A.; RATCLIFF, D., "Structural Ima-ging : Toward a Sharper Subsurface View," *Oilfield Review*, 1993, vol. 1, no. 1, p. 28–41.

[17] SALAMANCA, William; ABREO, Sergio; FAJARDO, Carlos; RAMÍREZ, Ana, "Alternative Computing Platforms to Implement the Seismic Migration," *In New Technologies in the Oil and Gas Industry*, 2012, p. 83–102.

[18] MULDER, W.; PLESSIX, R., "A comparison between one-way and two-way wave-equation migration," *GEOPHYSICS*, vol. 69, no. 6 (NOVEMBER-DECEMBER 2004); p. 1491-1504.

[19] PARRA, Dorfell, "Analytical model to estimate the execution time of a 3D acoustic wave equation implementation using FDTD in a GPU," Tesis de maestría. Bucaramanga. Universidad Industrial de Santander. 2016, 90 p.

[20] ABDEL-QADER, Jareer H.; WALKER, Roger S., "Performance Evaluation of OpenMP Benchmarks on Intel's Quad Core Processors," *LATEST TRENDS on COMPUTERS*, 2010, vol 6, p. 348-355.

[21] McCALPIN, J.; MOORE, C.; HESTER, P., "The Role of Multicore Processors in the Evolution of General-Purpose Computing," *CTWatch Quarterly*, Febrero 2007, vol 3, no. 1.

[22] VENU, Balaji, "Multi-core processors-An overview," *Department of Electrical Engineering and Electronics, University of Liverpool, Liverpool, UK*, 2010, vol 5, no. 5, p. 30-32.

[23] GÓMEZ, José; TAPIA, Juan; SÁNCHEZ, Hugo, "Eficiencia de la paralelización automática en multiprocesadores utilizando modelos numéricos de circulación del océano costero," Centro de Investigación Científica y de Educación Superior de Ensenada, Ensenada, Baja California, México, 2002.

[24] MPI: The Complete Referenceby Marc Snir, Steve Otto, Jack Dongarra [en línea]. Disponible en: `http://www.netlib.org/utk/papers/mpi-book/mpi-book.html`

[25] Overview of Windows Services for UNIX 3.5 [en línea]. Disponible en: https://technet.microsoft.com/en-us/library/bb496994.aspx

[26] ZHOU, Muhong; SYMES, William W., "Wave equation based stencil optimizations on multi-core CPU," *SEG Technical Program Expanded Abstracts*, 2014 pp. 3551-3555.

[27] LUO, Qiuming; KONG, Chang; LIU, Gang, "Performance Evaluation of OpenMP Constructs and Kernel Benchmarks on a Loongson-3A Quad-Core SMP System," *College of Computer Science and Software Engineering Shenzhen University*, 2011.

BIBLIOGRAFÍA

ABDEL-QADER, Jareer H.; WALKER, Roger S., "Performance Evaluation of OpenMP Benchmarks on Intel's Quad Core Processors," *LATEST TRENDS on COMPUTERS*, 2010, vol 6, p. 348-355.

BAYSAL, E., "Reverse time migration," *Geophysics*, 1983, vol. 48, p. 1514-1524.

CABEZAS, J.; ARAYA-POLO, M.; GELADO, I.; NAVARRO, N.; MORANCHO, E. and CELA J.M., "High- Performance Reverse Time Migration on GPU," *International Conference of the Chilean Computer Science Society*, 2009, pp. 77–86.

CHANDRAKASAN, A.P.; POTKONJAK, M.; MEHRA, R.; RABAEY, J.; BRODERSEN, R.W., "Optimizing power using transformations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 1995. vol.14. no.1, p.12-31.

CHANDY, J.A.; SINGARAJU, J., "Hardware parallelism vs. software parallelism," *Proceedings of the First USENIX conference*. USENIX Association. 2009.

CHAPMAN, B.; JOST, G.; VAN DER PAS, Ruud., "Using OpenMP: Portable Shared Memory Parallel Programming," *The MIT Press*. 2008, p. [4-6, 23-26, 64-72, 79-82, 125-127].

DELL Search [en línea]. Disponible en: `http://pilot.search.dell.com/`

FAJARDO, Carlos; CASTILLO, Javier; PEDRAZA, César, "Reducción de los tiempos de cómputo de la Migración Sísmica usando FPGAs y GPGPUs: Un artículo de revisión," *Ingeniería y Ciencia*, vol. 9, no. 17, pp. 261–293, 2013.

FARMER, P.; GRAY, S.; HODGKISS, G.; PIEPRZAK, A.; RATCLIFF, D., "Structural Ima-ging : Toward a Sharper Subsurface View," *Oilfield Review*, 1993, vol. 1, no. 1, p.28–41.

GÓMEZ, José; TAPIA, Juan; SÁNCHEZ, Hugo, "Eficiencia de la paralelización automática en multiprocesadores utilizando modelos numéricos de circulación del océano costero," Centro de Investigación Científica y de Educación Superior de Ensenada, Ensenada, Baja California, México, 2002.

GRAY, Samuel H.; ETGEN, John; DELLINGER, Joe; WHITMORE, Dan. "Seismic migration problems and solutions," *GEOPHYSICS*, 2001, vol 66, no. 5, p. 1622-1640.

GUERRERO, Juan F.; FRANCÉS, José V., "Multiprocesadores Simétricos (SMP)," Escuela Técnica Superior de Ingeniería. Universidad de Valencia. 2010.

LIU, Lanbo; LIU, Zijian, "Application of Iterative Reverse Time Migration Procedure on Transcranial Thermoacoustic Tomography Imaging," School of Engineering, University of Connecticut, 2014.

LUO, Qiuming; KONG, Chang; LIU, Gang, "Performance Evaluation of OpenMP Constructs and Kernel Benchmarks on a Loongson-3A Quad-Core SMP System," *College of Computer Science and Software Engineering Shenzhen University*. 2011.

MARR, J.; DEBORAH, T.; FRANK and HILL, et al., "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal*. 2002. vol 6.

McCALPIN, J.; MOORE, C.; HESTER, P., "The Role of Multicore Processors in the Evolution of General-Purpose Computing," *CTWatch Quarterly*, Febrero 2007, vol 3, no. 1.

MPI: The Complete Referenceby Marc Snir, Steve Otto, Jack Dongarra [en línea]. Disponible en: `http://www.netlib.org/utk/papers/mpi-book/mpi-book.html`

MULDER, W.; PLESSIX, R., "A comparison between one-way and two-way wave-equation migration," *GEOPHYSICS*, vol. 69, no. 6 (NOVEMBER-DECEMBER 2004); p. 1491-1504.

Overview of Windows Services for UNIX 3.5 [en línea]. Disponible en:
`https://technet.microsoft.com/en-us/library/bb496994.aspx`

PARRA, Dorfell, "Analytical model to estimate the execution time of a 3D acoustic wave equation implementation using FDTD in a GPU,", Tesis de maestría. Bucaramanga. Universidad Industrial de Santander. 2016, 90 p.

SALAMANCA, William; ABREO, Sergio; FAJARDO, Carlos; RAMÍREZ, Ana, "Alternative Computing Platforms to Implement the Seismic Migration," *In New Technologies in the Oil and Gas Industry*, 2012, p. 83–102.

SAVA, Paul; FOMEL, Sergey, "Generalized imaging conditions for wave-equation migration," Colorado School of Mines Meeting. 2002.

SUN, Lin-Jie; YIN, Xing-Yao, "A Finite-Difference Scheme Based on PML Boundary Condition with High Power Grid Step Variation," *Chinese Journal of Geophysics*. 2011. no 3, p. 393-402.

VARGAS, Miguel, "Cómputo en paralelo con OpenMP," CIMAT. 2015.

VENU, Balaji, "Multi-core processors-An overview," *Department of Electrical Engineering and Electronics, University of Liverpool, Liverpool, UK*, 2010, vol 5, no. 5, p. 30-32.

Welcome to the OpenMP Architecture Review Board (ARB) [en línea]. Disponible en: `http://openmp.org/mp-documents/OpenMP-WelcomeGuide.pdf`

ZHOU, Muhong; SYMES, William W., "Wave equation based stencil optimizations on multi-core CPU," *SEG Technical Program Expanded Abstracts*. 2014 pp. 3551-3555.