TRABAJO DE FIN DE MÁSTER.
Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería (IUSIANI)
Universidad de Las Palmas de Gran Canaria

PARALELIZACIÓN DE MÉTODOS DE OPTIMIZACIÓN DE MALLAS 2D

Autor: Juan Pablo Quesada Nieves
Las Palmas de Gran Canaria, 2 de Diciembre de 2013

Tutores: Domingo Benítez Díaz
Eduardo Rodríguez Barrera

A Lot.

AGRADECIMIENTOS

Agradezco a mis tutores Domingo Benítez Díaz y Eduardo Rodríguez Barrera todo el apoyo y esfuerzo aportados para afrontar este Trabajo de Fin de Máster, así como su inmensa aportación de conocimientos para llevarlo a cabo. A Rafael Montenegro Armas porque es inestimable su experiencia. A José Iván López González por haber aportado los ficheros con los nodos de las mallas utilizados en la experimentación. A la Universidad de Las Palmas de Gran Canaria por haberme permitido realizar este TFM como Profesor Asociado a Tiempo Parcial en el Departamento de Informática y Sistemas. Por último, agradezco a todos la paciencia que han tenido conmigo por no haberles dedicado el tiempo que merecían durante el desarrollo de este TFM.

ÍNDICE

1. INTRODUCCIÓN

Una *malla* puede definirse como un conjunto de vértices (también denominados nodos o puntos) y aristas. Los vértices pueden unirse por esas aristas de diversas maneras, lo que conforma la malla definida, es decir, en el espacio de trabajo 2D se podría hablar de una malla triangular si tres son los vértices que conforman la unión de las aristas, o de malla rectangular si cuatro son los vértices que conforman esa unión. La **Figura 1.1** muestra dos ejemplos de malla rectangular, una con geometría regular (izquierda) y otra con geometría irregular (derecha).

La idea es construir una malla uniforme en el espacio paramétrico 2D y transformarla con una buena aproximación de la geometría a una malla, topológicamente equivalente, en el espacio físico 2D. Como ejemplo, se puede considerar la construcción de una malla que represente la isla de Gran Canaria, cuya geometría es irregular, a partir de una rejilla regular definida en el espacio paramétrico (un cuadrado), tal y como se muestra en la **Figura 1.1**. Existen diversos métodos que permiten realizar una transformación entre el espacio paramétrico y el espacio real. Entre ellos destacan los *parches de Coons* y la transformación mediante el *suavizado Laplaciano*. Estos métodos pueden implementarse mediante técnicas de paralelización, que permitirían acelerar el proceso. De esta forma, sería posible trabajar con mallas de un gran número de grados de libertad.

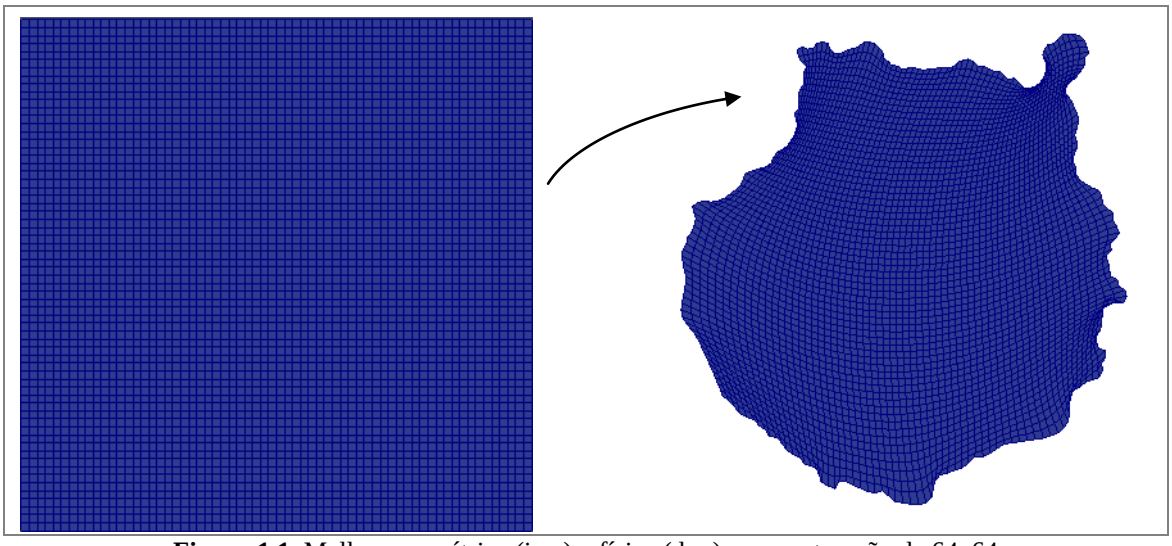

Figura 1.1. Mallas paramétrica (izq.) y física (der.) para un tamaño de 64x64

Este Trabajo de Fin de Máster pretende mostrar el estudio y la implementación de los métodos mencionados anteriormente para realizar una transformación entre el espacio paramétrico y el espacio real. Para ello, se han utilizado técnicas de paralelización tales como las ofrecidas por OpenMP y CUDA. El estudio también incluye una comparativa entre los métodos y las implementaciones realizadas, así como de las mejoras obtenidas, sobre todo con respecto a la versión secuencial de los métodos.

Hay que destacar que las mallas obtenidas con las implementaciones desarrolladas son mallas de calidad, pero no totalmente desenredadas. Para realizar el desenredo total de una malla se tendría que hacer uso de algún software para desenredo, tal como podría ser *SUS Code* [SUS], software implementado para el desenredo y la optimización simultáneos de mallas de tetraedros.

2. MÉTODOS DE OPTIMIZACIÓN DE MALLAS 2D. FUNDAMENTOS TEÓRICOS

2.1. Parches de Coons

El problema que se pretende abordar [FH99] es encontrar una superficie paramétrica $x(u,v)$ cuyo contorno es dado por cuatro curvas

$$x(u,0) \quad x(u,1) \quad x(0,v) \quad x(1,v)$$

Sin pérdida de generalidad, se asume que el dominio de la superficie paramétrica $x(u,v)$ es el cuadrado unidad $0 \le u,v \le 1$.

Una solución a este problema es el método conocido como los *parches de Coons* (*Coons* de ahora en adelante), que interpola a curvas de contorno dadas:

$$x(u,v) = (1-u)x(0,v) + ux(1,v) + (1-v)x(u,0) + vx(u,1) - \begin{bmatrix} 1-u & u \end{bmatrix} \begin{bmatrix} x(0,0) & x(0,1) \\ x(1,0) & x(1,1) \end{bmatrix} \begin{bmatrix} 1-v \\ v \end{bmatrix}$$

Si se considera la superficie como un array de puntos, donde el contorno de dicha superficie lo forman puntos conocidos (discretización de las curvas de contorno mencionadas), una versión discreta de lo anterior podría ser

$$b_{i,j} = (1-i/m)b_{0,j} + i/m\,b_{m,j} + (1-j/n)b_{i,0} + j/n\,b_{i,n} - \begin{bmatrix} 1-i/m & i/m \end{bmatrix} \begin{bmatrix} b_{0,0} & b_{0,n} \\ b_{m,0} & b_{m,n} \end{bmatrix} \begin{bmatrix} 1-j/n \\ j/n \end{bmatrix}$$

para $0 < i < m$ y $0 < j < n$, donde los $b_{i,j}$ son los puntos interiores a obtener por interpolación partiendo de un contorno especificado. Un ejemplo de este array de puntos donde el contorno es conocido podría ser el siguiente, para un tamaño de puntos de 8x8:

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$	$b_{0,4}$	$b_{0,5}$	$b_{0,6}$	$b_{0,7}$
$b_{1,0}$							$b_{1,7}$
$b_{2,0}$							$b_{2,7}$
$b_{3,0}$							$b_{3,7}$
$b_{4,0}$							$b_{4,7}$
$b_{5,0}$							$b_{5,7}$
$b_{6,0}$							$b_{6,7}$
$b_{7,0}$	$b_{7,1}$	$b_{7,2}$	$b_{7,3}$	$b_{7,4}$	$b_{7,5}$	$b_{7,6}$	$b_{7,7}$

La **Figura 2.1** muestra un ejemplo de aplicación de Coons, donde se puede observar que el contorno de la superficie es bastante irregular.

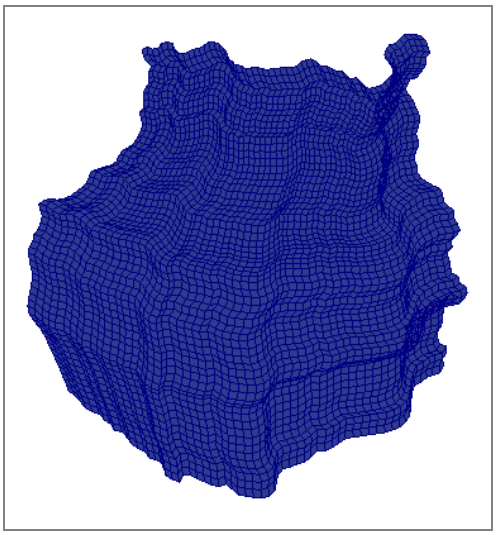

Figura 2.1. Ejemplo de aplicación de Coons

2.2. Suavizado Laplaciano

El *suavizado Laplaciano* (*Laplaciano* de ahora en adelante) es un método para suavizar mallas poligonales. Para cada nodo interior en una malla, su nueva posición se calcula en base a la posición de sus vecinos. En el caso de una malla rectangular, es decir, cada nodo interno está conectado con cuatro vecinos, esta operación produce el *Laplaciano* de la malla.

Formalmente, la operación de suavizado en una malla rectangular puede ser descrita por nodo como:

$$\bar{x}_{i,j} = \frac{1}{4}\left(x_{i,j+1} + x_{i+1,j} + x_{i,j-1} + x_{i-1,j}\right)$$

donde 4 es el número de nodos adyacentes al nodo $x_{i,j}$ y $\bar{x}_{i,j}$ es la nueva posición para el nodo $x_{i,j}$.

En la malla de puntos siguiente (tamaño 8x8), la nueva posición del nodo $x_{1,1}$ se calcula en base a sus cuatro vecinos $x_{1,2}$, $x_{2,1}$, $x_{1,0}$ y $x_{0,1}$.

$x_{0,0}$	$x_{0,1}$	$x_{0,2}$	$x_{0,3}$	$x_{0,4}$	$x_{0,5}$	$x_{0,6}$	$x_{0,7}$
$x_{1,0}$	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$x_{1,4}$	$x_{1,5}$	$x_{1,6}$	$x_{1,7}$
$x_{2,0}$	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	$x_{2,4}$	$x_{2,5}$	$x_{2,6}$	$x_{2,7}$
$x_{3,0}$	$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	$x_{3,4}$	$x_{3,5}$	$x_{3,6}$	$x_{3,7}$
$x_{4,0}$	$x_{4,1}$	$x_{4,2}$	$x_{4,3}$	$x_{4,4}$	$x_{4,5}$	$x_{4,6}$	$x_{4,7}$
$x_{5,0}$	$x_{5,1}$	$x_{5,2}$	$x_{5,3}$	$x_{5,4}$	$x_{5,5}$	$x_{5,6}$	$x_{5,7}$
$x_{6,0}$	$x_{6,1}$	$x_{6,2}$	$x_{6,3}$	$x_{6,4}$	$x_{6,5}$	$x_{6,6}$	$x_{6,7}$
$x_{7,0}$	$x_{7,1}$	$x_{7,2}$	$x_{7,3}$	$x_{7,4}$	$x_{7,5}$	$x_{7,6}$	$x_{7,7}$

La **Figura 2.2** muestra un ejemplo de aplicación del Laplaciano, con el mismo contorno irregular utilizado en Coons (**Figura 2.1**).

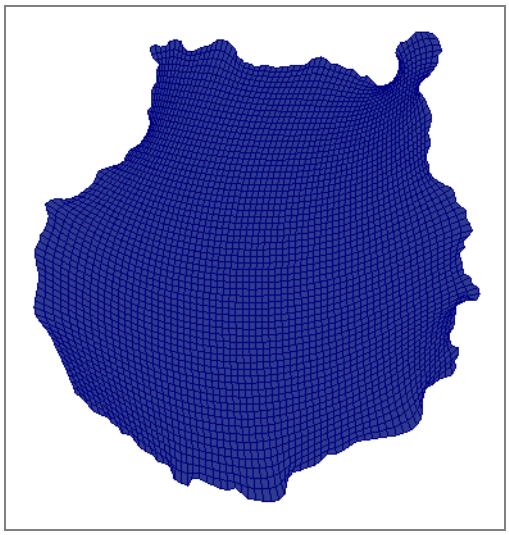

Figura 2.2. Ejemplo de aplicación del Laplaciano

3. TECNOLOGÍAS DE PARALELIZACIÓN

3.1. OpenMP

OpenMP [AGMV08] es una API (Application Programming Interface o Interfaz de Programación de Aplicaciones) que puede ser utilizada para diseñar explícitamente programas paralelos multithread en memoria compartida. Básicamente, un programa OpenMP es un programa secuencial con directivas OpenMP dispuestas en los puntos apropiados, tal y como muestra la **Figura 3.1**.

```
#include <omp.h>

int main()
{
int nthreads, tid;

#pragma omp parallel private(tid)
{
      tid = omp_get_thread_num();
      nthreads = omp_get_num_threads();
      printf("Hola desde el thread %d de %d threads\n",
             tid, nthreads);
}
}
```

Figura 3.1. Ejemplo de código OpenMP [AGMV08]

OpenMP está compuesto de tres elementos:

- Las directivas de compilación
- Las rutinas de librería
- Las variables de entorno

Se trata de un entorno portable y estandarizado. La API ha sido especificada para C/C++ y Fortran y extendida a muchas plataformas.

En [OMP] pueden encontrarse los documentos y archivos oficiales de las especificaciones.

El modelo de programación en OpenMP hace uso de que un proceso puede consistir en múltiples threads de ejecución en la máquina de memoria compartida. Utiliza el modelo *fork-join* (**Figura 3.2**) y el programador tiene control explícito de la paralelización. Se supone que el programa tiene secciones de código secuenciales (no paralelizables) y secciones paralelizables, y que el programador interviene sobre las secciones paralelizables.

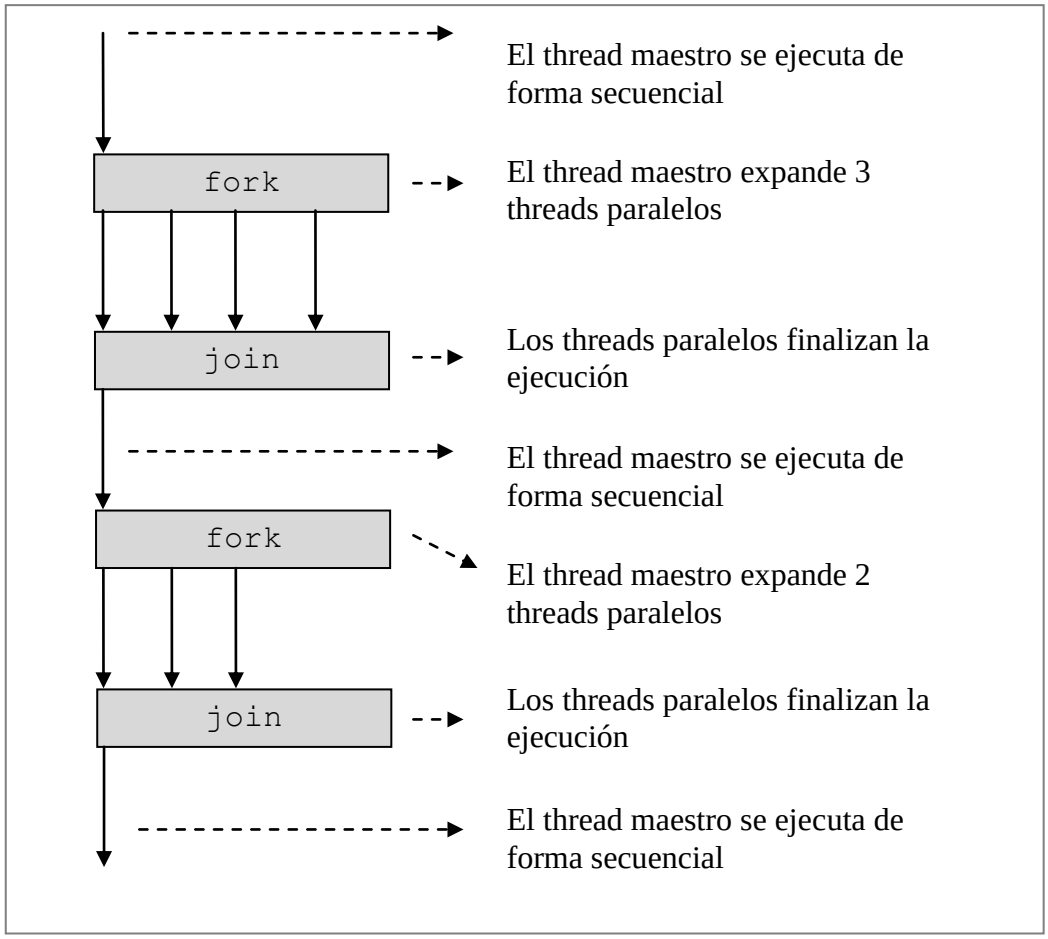

Figura 3.2. El modelo *fork-join* de memoria compartida [AGMV08]

3.2. CUDA

CUDA [N13] es una arquitectura de cálculo paralelo de NVIDIA que aprovecha la gran potencia de la GPU (Graphics Processing Unit o Unidad de Procesamiento Gráfico) para proporcionar un incremento extraordinario del rendimiento del sistema.

Gracias a millones de GPUs CUDA vendidas hasta la fecha, miles de desarrolladores, científicos e investigadores están encontrando innumerables aplicaciones prácticas para esta tecnología en campos como el procesamiento de vídeo e imágenes, la biología y la química computacional, la simulación de la dinámica de fluidos o el análisis sísmico, entre otras.

Los sistemas informáticos están pasando de realizar el "procesamiento central" en la CPU (Central Processing Unit o Unidad Central de Procesamiento) a realizar "coprocesamiento" repartido entre la CPU y la GPU. Para posibilitar este nuevo paradigma computacional, NVIDIA ha inventado la arquitectura de cálculo paralelo CUDA, que ahora se incluye en las GPUs GeForce, ION Quadro y Tesla GPUs, lo cual representa una base instalada considerable para los desarrolladores de aplicaciones.

La plataforma de cálculo paralelo CUDA proporciona unas cuantas extensiones de C y C++ que permiten implementar el paralelismo en el procesamiento de tareas y datos con diferentes niveles de granularidad. El programador puede expresar ese paralelismo mediante diferentes lenguajes de alto nivel como C, C++ y Fortran (**Figura 3.3**).

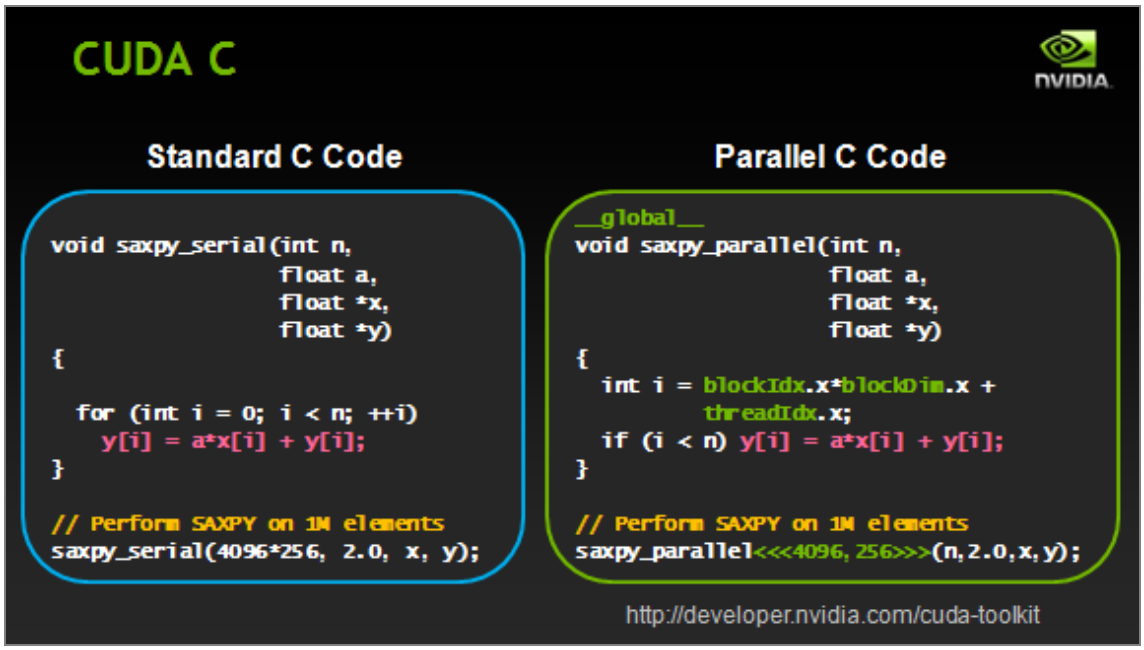

Figura 3.3. Comparación de código C y código C paralelo [N13]

GPU Computing

GPU Computing [N13] es el uso de la GPU junto con una CPU para acelerar operaciones de cálculo científico o técnico de propósito general.

El cálculo en la GPU ofrece un rendimiento de aplicaciones sin igual al descargar en la GPU las partes de la aplicación que requieren gran capacidad computacional, mientras que el resto del código sigue ejecutándose en la CPU. Desde la perspectiva del usuario, las aplicaciones simplemente se ejecutan más rápido.

Una CPU + GPU constituye una potente combinación porque la CPU está formada por varios núcleos optimizados para el procesamiento secuencial, mientras que la GPU consta de millares de núcleos más pequeños y eficientes diseñados para el rendimiento en paralelo (**Figura 3.4**). Las partes secuenciales del código se ejecutan en la CPU mientras que las paralelas se ejecutan en la GPU.

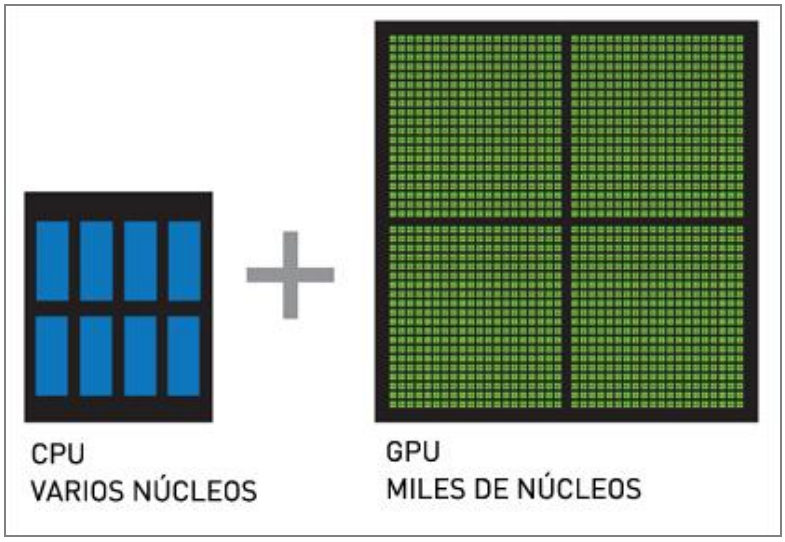

Figura 3.4. Combinación CPU + GPU [N13]

Estructura de un programa CUDA

Un programa en CUDA [N13] consiste en una o más fases que se ejecutan o en el host (CPU) o en un dispositivo tal como una GPU. Las fases que muestran poco o ningún paralelismo se implementan en código del host. Las fases que muestran gran paralelismo de datos se implementan en código de dispositivo. Un programa CUDA es código que unifica tanto código del host como del dispositivo. El compilador C de NVIDIA (nvcc) separa los dos códigos durante el proceso de compilación. El código del host es ANSI C; el código del dispositivo es código ANSI C extendido con palabras clave para etiquetar funciones con paralelismo a nivel de datos, denominadas *kernels*, y sus estructuras de datos asociadas.

Las funciones kernel típicamente generan un gran número de hilos que explotan el paralelismo a nivel de datos. Hay que destacar que los hilos de CUDA son mucho más ligeros que los hilos de CPU. Los programadores de CUDA pueden asumir que estos hilos toman pocos ciclos de reloj para generarse y planificarse debido a hardware eficiente. Esto contrasta con los hilos de CPU, que típicamente requieren miles de ciclos para generarse y planificarse.

La ejecución de un programa típico CUDA se ilustra en la **Figura 3.5**. La ejecución comienza con la ejecución del código de host (CPU). Cuando una función kernel se invoca, o se lanza, la ejecución se mueve a un dispositivo (GPU), donde un gran número de hilos se generan para aprovechar el alto poder de paralelismo. Todos los hilos que se generan en una invocación de un kernel se denominan colectivamente *grid* (la **Figura 3.5** muestra la ejecución de dos grids de hilos). Cuando todos los hilos de un kernel completan su ejecución, el grid correspondiente termina y la ejecución continúa en el host hasta que otro kernel sea invocado.

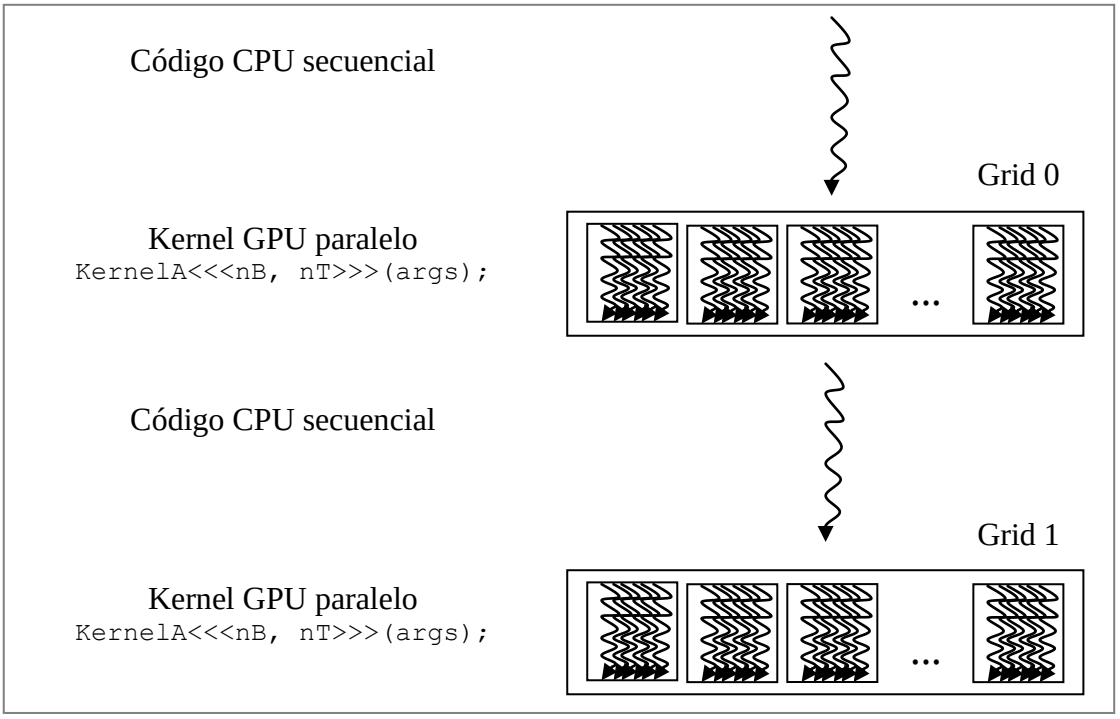

Figura 3.5. Ejecución de un programa CUDA [KH10]

3.3. Paralelización de los métodos de estudio

Parches de Coons

La implementación de los *parches de Coons* con OpenMP consiste en lanzar una serie de *threads* donde cada uno de ellos se encarga de obtener las nuevas posiciones ($b_{i,j}$) de los nodos interiores de una serie de filas asignadas, pues se hace un reparto más o menos equitativo de las filas de la malla por thread. La **Figura 3.6** muestra un diagrama de actividad que refleja los pasos involucrados en la implementación.

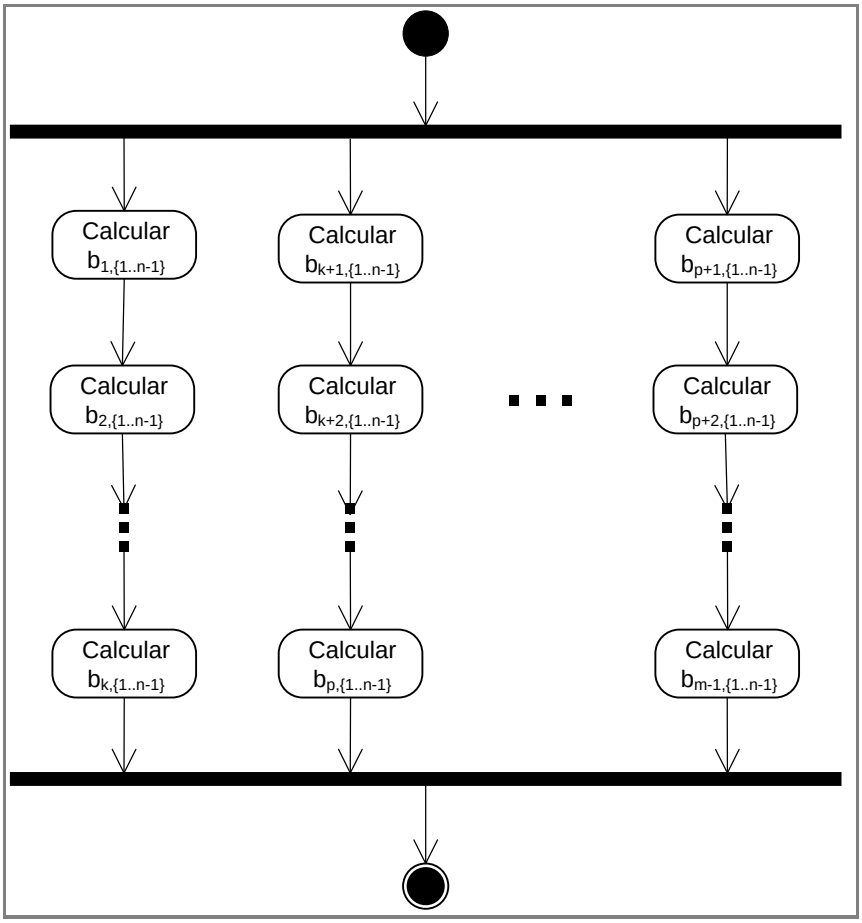

Figura 3.6. Diagrama de actividad para la implementación de Coons con OpenMP

En el caso de la implementación de los *parches de Coons* con CUDA, ésta se lleva a cabo lanzando tantos hilos como nuevas posiciones de nodos interiores haya que calcular, encargándose cada hilo de la obtención de una única posición de nodo. La **Figura 3.7** muestra el diagrama de actividad que refleja los pasos involucrados en la implementación.

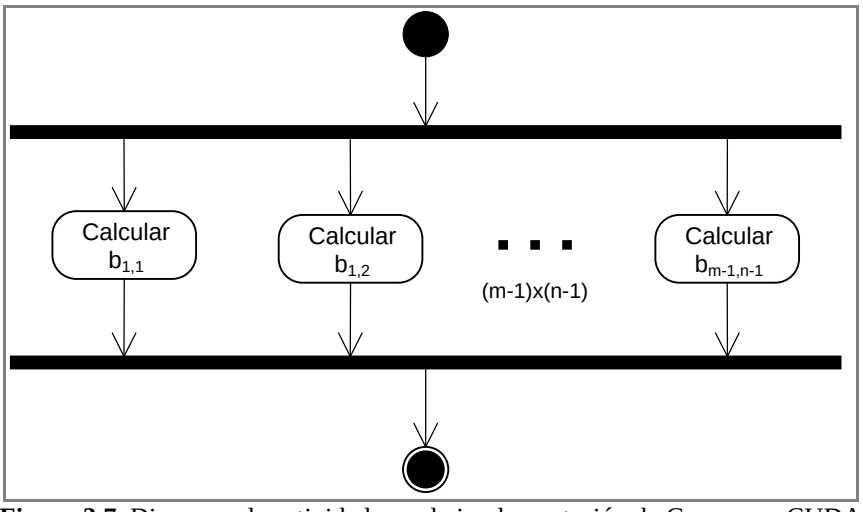

Figura 3.7. Diagrama de actividad para la implementación de Coons con CUDA

Suavizado Laplaciano

Para la implementación del *suavizado Laplaciano*, tanto en OpenMP como en CUDA, es interesante aplicar una técnica a la que se ha denominado *coloreado*, que permite calcular de forma paralela aquellos nodos interiores que no se afectan entre sí en los cálculos para obtener su nueva posición. La **Figura 3.8** muestra una ilustración de dicho coloreado, donde se observa que en una iteración del Laplaciano se podrían obtener de forma paralela las nuevas posiciones de los nodos identificados con el color verde, por ejemplo, mientras que en otra se obtendrían las nuevas posiciones de los nodos identificados con el color rojo. Las etiquetas I y P hacen referencia a si la columna (también podría utilizarse la fila) es impar o par, respectivamente, dato a tener en cuenta en la paralelización.

La implementación del método del Laplaciano con OpenMP consiste en lanzar una serie de *threads* donde cada uno de ellos se encarga de obtener las nuevas posiciones ($\bar{x}_{i,j}$) de los nodos interiores de una serie de filas asignadas, pues se hace un reparto más o menos equitativo de las filas de la malla por thread, tal y como sucedía en la implementación de Coons. Además, se aplica la técnica del coloreado mencionada anteriormente, donde primero se obtienen las nuevas posiciones de los nodos interiores de un color y, a continuación, las nuevas posiciones de los nodos interiores del otro color. La **Figura 3.9** muestra un diagrama de actividad que refleja los pasos involucrados en la implementación.

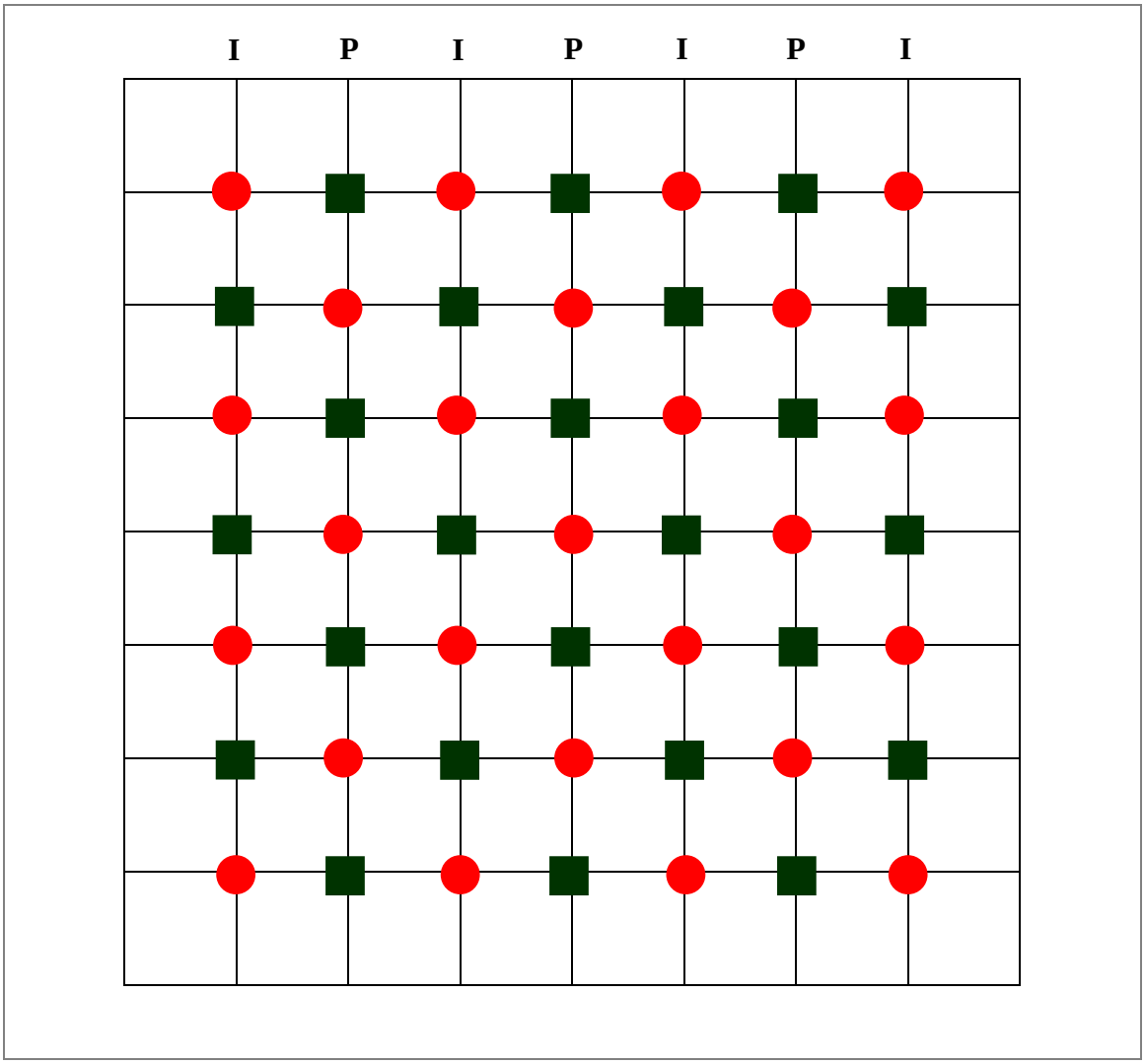

Figura 3.8. Ejemplo de *coloreado* en una malla de 8x8

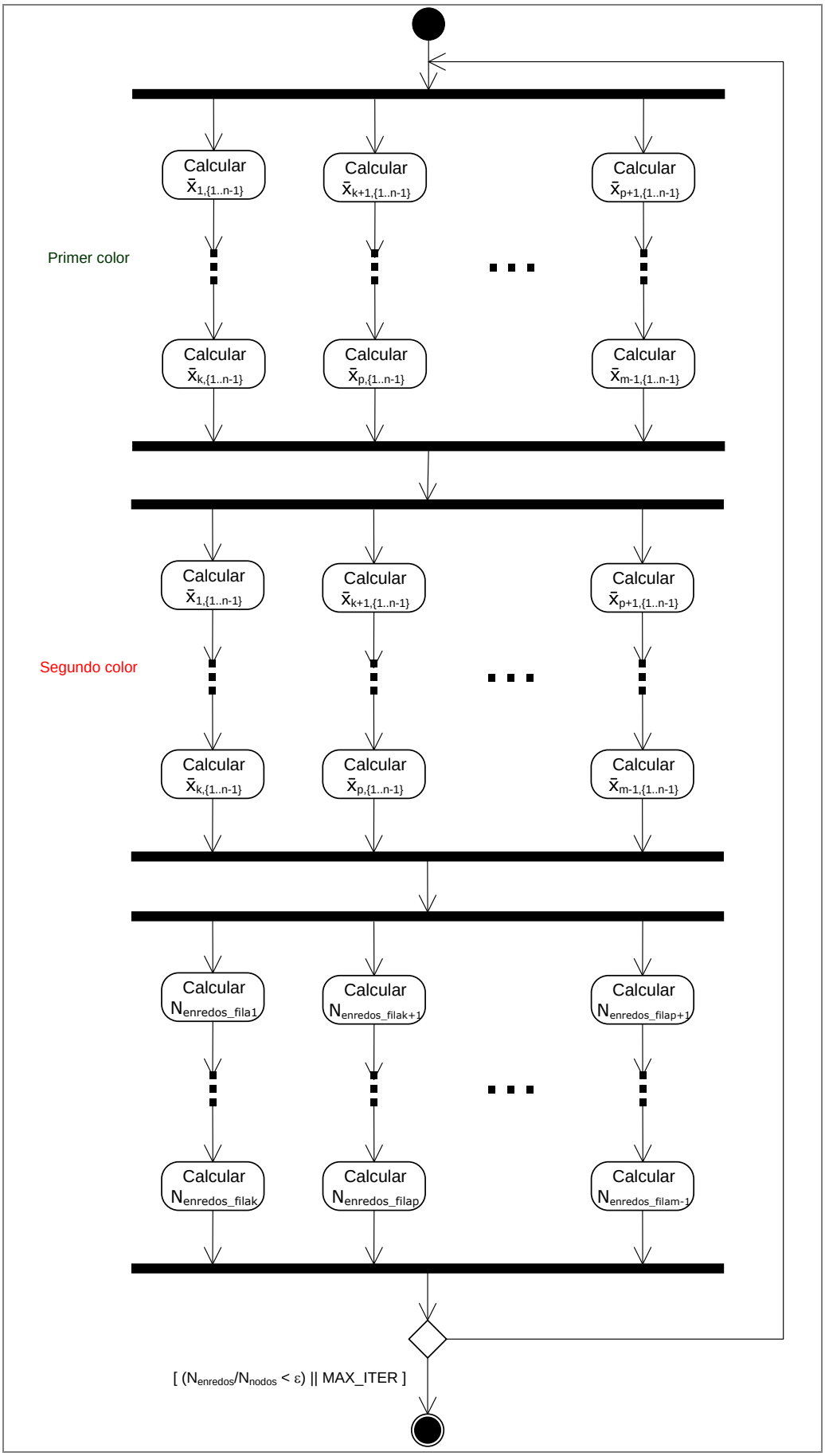

Figura 3.9. Diagrama de actividad para la implementación del Laplaciano con OpenMP

Para el cálculo del número de nodos enredados, cada thread se encarga de analizar los nodos interiores situados en un conjunto de filas de la malla y contabilizar la cantidad de enredos localizados. La operación de reducción de OpenMP permite aunar todos los resultados obtenidos por todos los threads.

En la implementación con CUDA se lanzan tantos hilos como filas tenga la malla. Cada hilo se encargará de obtener las nuevas posiciones de los nodos interiores en la fila correspondiente. También se aplica la técnica del coloreado, es decir, primero se realiza lo mencionado para un color y, a continuación, para el otro color. La **Figura 3.10** muestra un diagrama de actividad que refleja los pasos involucrados en la implementación.

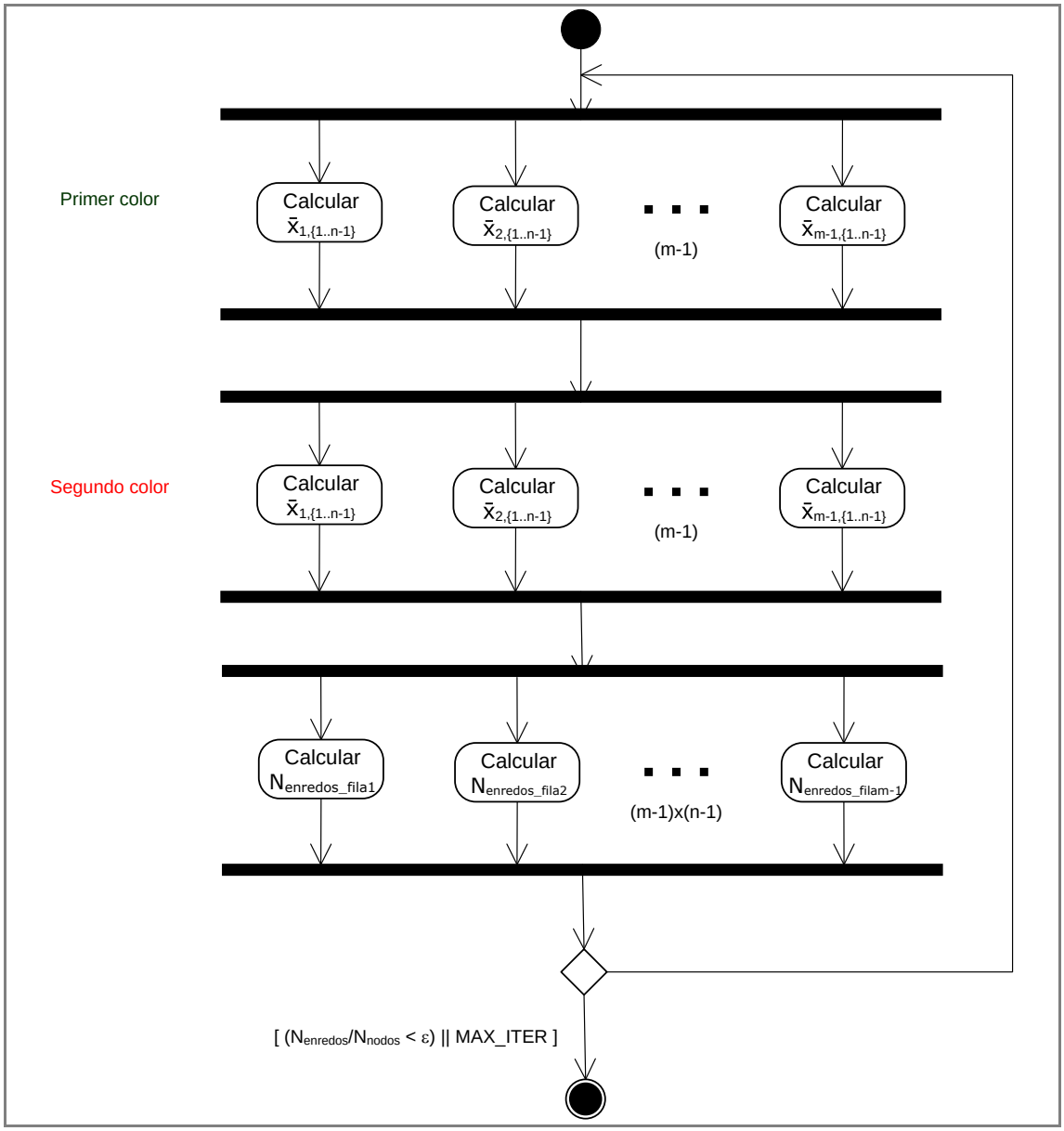

Figura 3.10. Diagrama de actividad para la implementación del Laplaciano con CUDA

La obtención del número de nodos enredados en cada iteración se obtiene lanzando tantos hilos como nodos interiores haya. Cada hilo se encargará de comprobar si el nodo que le ha correspondido está enredado o no, guardando el valor 1 en la posición correspondiente a su identificador en un vector inicializado a 0 al comienzo de la

iteración. Se aplicará una operación de reducción para computar el número total de nodos enredados en la iteración.

4. RESULTADOS

Después de implementar los métodos de *Coons* y *Laplaciano* de forma secuencial y paralela (con OpenMP y CUDA) se pasó a realizar pruebas del funcionamiento de cada código generado con diferentes figuras, dadas estas últimas por los puntos de su contorno. Recordar que uno de los obejtivos a alcanzar era obtener los puntos (o nodos) interiores de una malla especificada por su contorno y partiendo de una malla de nodos paramétricos.

En el caso de Coons, con una sola ejecución del método se obtienen los nodos interiores y la malla representa la figura deseada. Sin embargo, en el caso del Laplaciano, el método requiere una serie de iteraciones para alcanzar la figura. El saber el número exacto de iteraciones no es posible a priori, lo que supone un problema, puesto que los tiempos con gran número de iteraciones podrían dispararse, y al final no conseguir la figura deseada, mientras que con un número de iteraciones pequeño se podría no obtener la malla física que represente la figura en cuestión.

Un estudio realizado consistió en analizar la evolución del número de nodos enredados en la malla física con respecto al número de iteraciones, pensando que tal vez, en algún momento, se podría alcanzar un número de nodos enredados constante, y se podría parar la ejecución del Laplaciano. Esto supondría que el número de iteraciones y, por ende, el tiempo, disminuirían.

La **Figura 4.1** muestra el caso del Laplaciano para una malla de tamaño 512x512, donde se puede observar que, durante aproximadamente las 1036 primeras ejecuciones del método, el número de nodos enredados va en aumento, pero luego empieza a descender llegando a estabilizarse en un valor en la iteración 12000 aproximadamente.

Figura 4.1. *nodos enredados* vs. *iteraciones* en la obtención de una malla física de tamaño 512x512. Malla física obtenida con Laplaciano

La **Figura 4.2** muestra un comportamiento similar para una malla de tamaño 1024x1024, pero en este caso el número de nodos enredados empieza a estabilizarse a partir de la ejecución 30000, todo debido a la inmensa cantidad de nodos presentes (1024*1024 = 1048576).

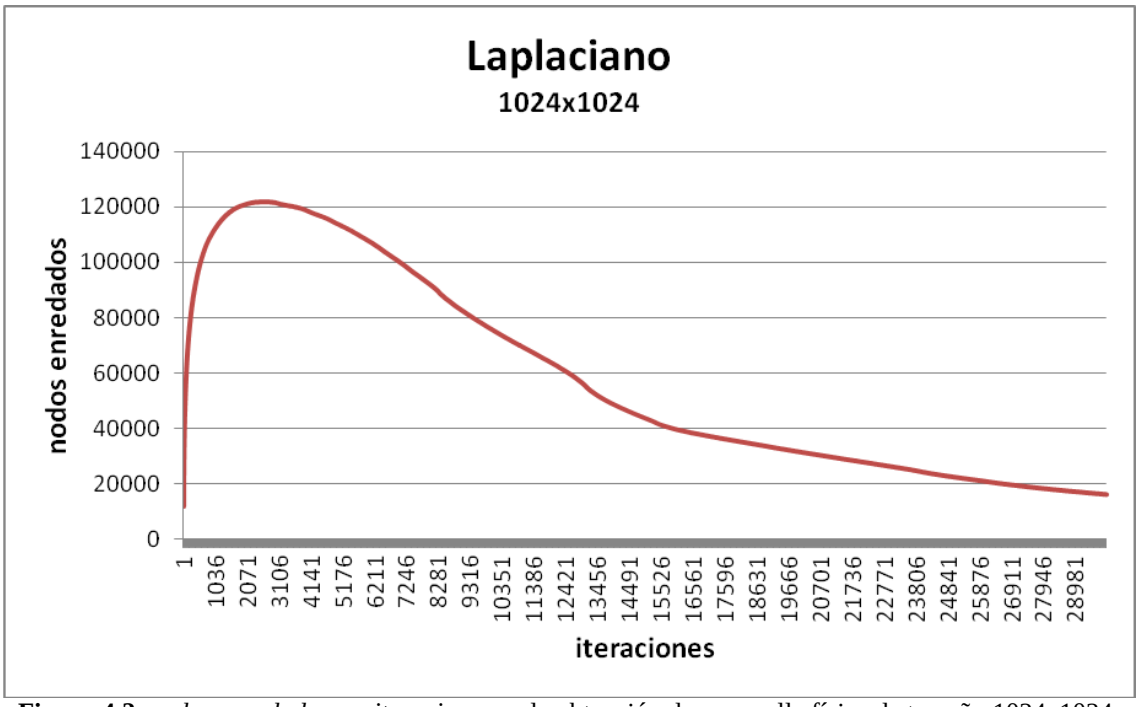

Figura 4.2. *nodos enredados* vs. *iteraciones* en la obtención de una malla física de tamaño 1024x1024. Malla física obtenida con Laplaciano

La **Figura 4.3** muestra el caso del uso de ambos métodos para una malla física de tamaño 512x512, ejecutando Coons en primer lugar y, considerando su resultado como primera aproximación, Laplaciano después. Esto produce que con Coons ya se obtenga la figura deseada con unos nodos enredados concretos. El iterar con el Laplaciano hace que ese número de nodos enredados vaya descendiendo hasta estabilizarse en un cierto valor. La peculiaridad es el mínimo número de enredos que se alcanza en la iteración 1036 aproximadamente, incrementándose después hasta alcanzar el valor en el que se estabiliza.

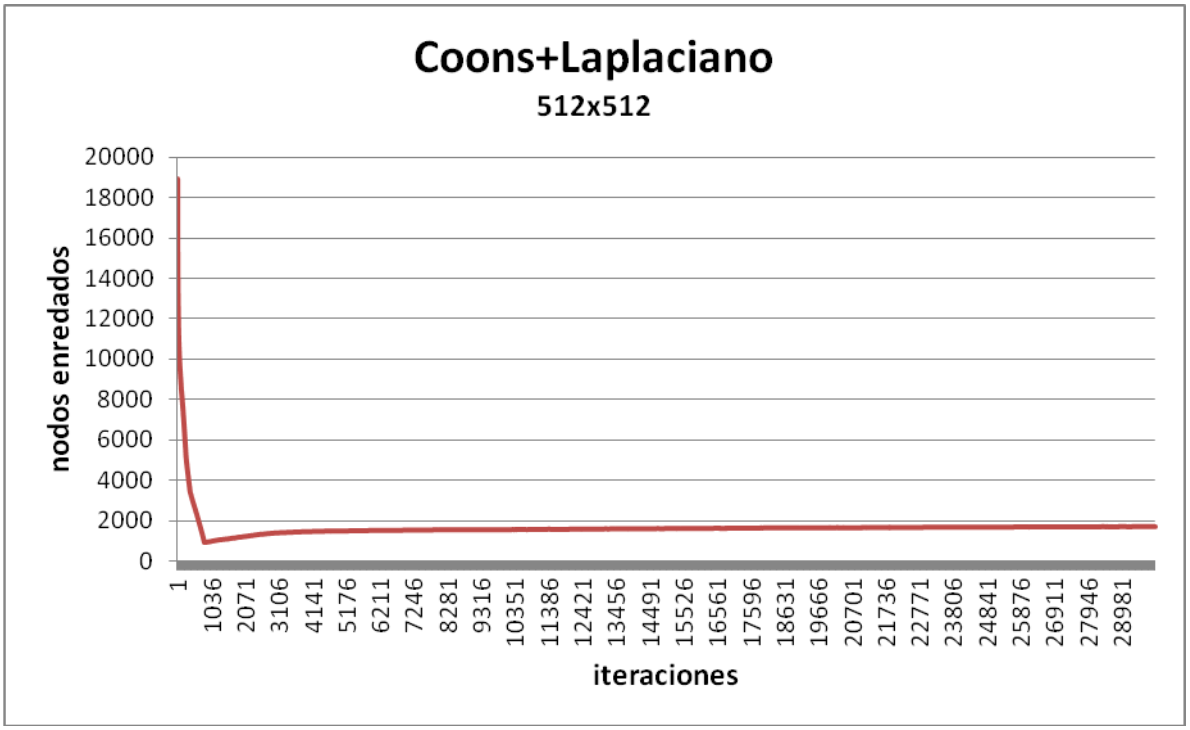

Figura 4.3. *nodos enredados* vs. *iteraciones* en la obtención de una malla física de tamaño 512x512. Malla física obtenida con Coons+Laplaciano

Tras observar que el número de nodos enredados llega a estabilizarse alrededor de un cierto valor, se podría considerar éste como criterio de parada en la ejecución del Laplaciano, es decir, cuando no haya diferencia significativa en el cálculo entre una iteración y la anterior, dejar de iterar.

Otro criterio de parada, y que ha sido el utilizado en las implementaciones, podría ser

$$\frac{N^{i}_{enredos}}{N_{nodos}} < \varepsilon$$

siendo ε un cierto umbral. Esto significaría que la proporción entre el número de nodos enredados en una iteración *i* y el número de nodos total tendría que ser menor que un cierto porcentaje para dejar de iterar. Este porcentaje, para mallas de tamaño 512x512, se podría cuantificar alrededor del 1%, lo que significaría que la malla obtenida tendría como mucho el 1% de sus nodos enredados tras finalizar las iteraciones del Laplaciano; en el caso de mallas de tamaño 1024x1024 no es descabellado pensar en un 0.5%.

El número de nodos enredados en la iteración *i* se podría obtener (y así se ha implementado) calculando por nodo el producto vectorial de los vectores resultantes considerando sus cuatro vecinos, al ser una malla rectangular (**Figura 4.4**). Primero se calcularía $\vec{1}x\vec{2}$, luego $\vec{2}x\vec{3}$, luego $\vec{3}x\vec{4}$ y, por último $\vec{4}x\vec{1}$. Con que un cálculo de los cuatro resultara negativo o nulo, el nodo ya se consideraría enredado. Así, por ejemplo, el producto vectorial entre los vectores $\vec{1}$ y $\vec{2}$ quedaría como:

$$\vec{1}x\vec{2} = \begin{vmatrix} 1x & 1y \\ 2x & 2y \end{vmatrix} > 0$$

siendo $1x$ y $1y$ las componentes x e y, respectivamente, del vector $\vec{1}$, y $2x$ y $2y$ las componentes x e y, respectivamente, del vector $\vec{2}$. Que el producto vectorial sea mayor que cero significa que el triángulo definido por los vectores no está invertido y, por ende, el nodo, por ahora, no está enredado.

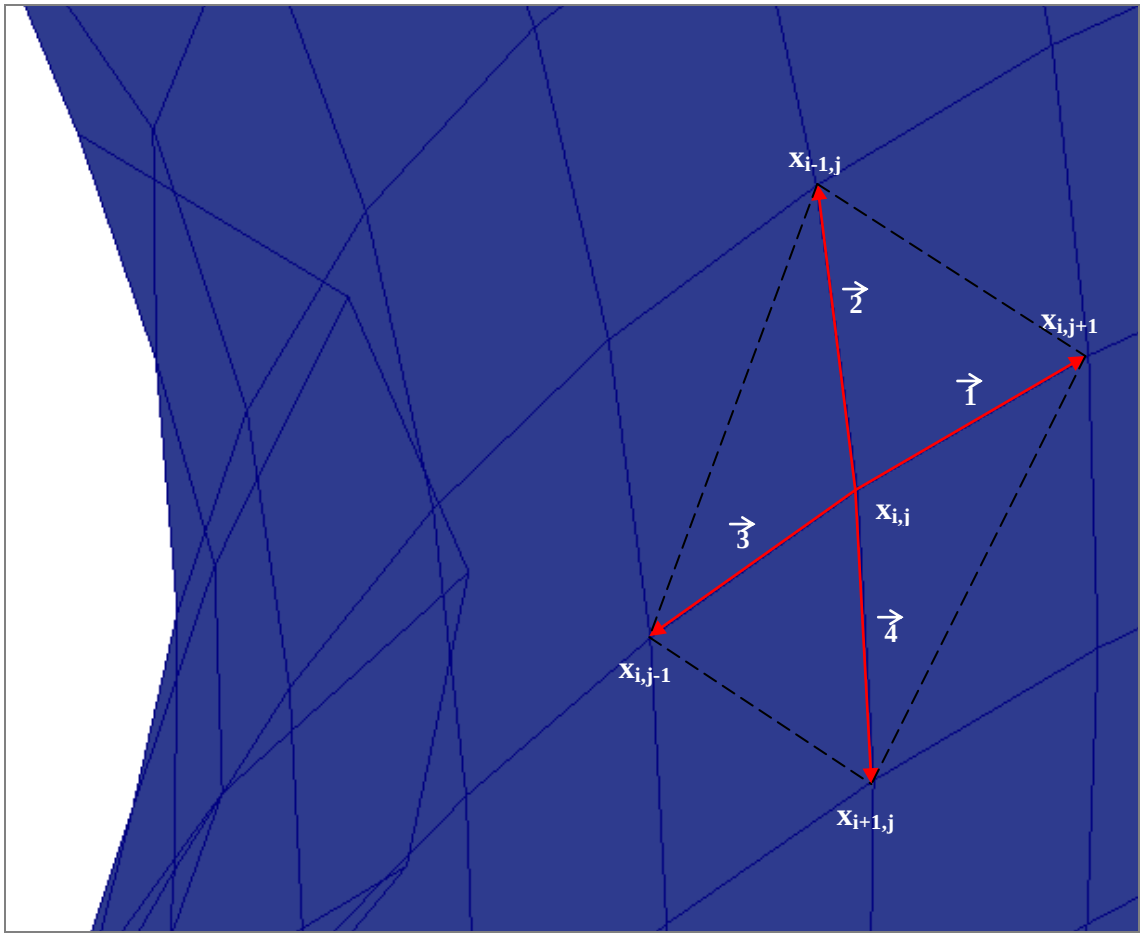

Figura 4.4. Datos necesarios para el cálculo que determina si un nodo $x_{i,j}$ está o no enredado

Cuando no se consiga parar por alguno de los criterios mencionados, se podría pensar en un número de iteraciones máximo, se llegue o no a la solución deseada.

Tiempos de Ejecución y Speedups

A continuación, se presentan varias tablas que reflejan los diferentes tiempos de ejecución (en milisegundos) según las diferentes implementaciones (secuencial, paralela con OpenMP y paralela con CUDA) de cada uno de los métodos (Coons y Laplaciano), además de la aplicación de Coons como primera aproximación para luego iterar con el Laplaciano.

La toma de datos se ha realizado haciendo uso de tres dimensiones de grid diferentes: 64x64, 512x512 y 1024x1024. Asimismo, en el caso de las implementaciones paralelas, se ha configurado cada una de ellas para operar con una serie de hilos. En OpenMP se han utilizado configuraciones de 10, 15 y 20 hilos de ejecución para tomar los tiempos con cada una de las dimensiones del grid; en CUDA, los hilos por bloque considerados han sido 16, 128 y 512, también con cada uno de los tamaños de la malla.

Por otro lado, para facilitar la comparativa entre las diferentes implementaciones, también se han elaborado tablas donde se muestra un resumen de tiempos para cada método implementado, así como tablas donde se exponen los diferentes *speedups* (aceleraciones) obtenidos. Hay que destacar que para el cálculo de estas aceleraciones se ha dividido el tiempo secuencial entre el paralelo.

$$Speedup = \frac{t_{secuencial}}{t_{paralelo}}$$

La **Tabla 4.1** refleja los tiempos de ejecución de la implementación del método de Coons según las diferentes versiones, sean secuencial, paralela con OpenMP y paralela con CUDA. Asimismo, también muestra los *speedups* alcanzados con los desarrollos paralelos.

Coons

		Secuencial	OpenMP				CUDA			
		$t_{ejecución}$			$t_{ejecución}$	Speedup			$t_{ejecución}$	Speedup
Dimensiones del grid	64x64	1 ms	Hilos	10	1 ms	1,0	Hilos por bloque	16	1 ms	1,0
				15	2 ms	0,5		128	1 ms	1,0
				20	1 ms	1,0		512	1 ms	1,0
	512x512	45 ms	Hilos	10	9 ms	5,0	Hilos por bloque	16	5 ms	9,0
				15	11 ms	4,1		128	5 ms	9,0
				20	9 ms	5,0		512	5 ms	9,0
	1024x1024	182 ms	Hilos	10	33 ms	5,5	Hilos por bloque	16	15 ms	12,1
				15	24 ms	7,6		128	14 ms	13,0
				20	24 ms	7,6		512	15 ms	12,1

Tabla 4.1. Tiempos de ejecución y *speedups* alcanzados para Coons

Analizando la **Tabla 4.1** se observa que los tiempos en la implementación con CUDA son prácticamente similares para cada grupo de datos agrupados según la dimensión de la malla. No ocurre lo mismo en el caso de la implementación de Coons con OpenMP. Si bien es cierto que para los tamaños de malla 64x64 y 512x512 los tiempos (agrupados y considerados según el tamaño) prácticamente no tienen diferencia importante, para un tamaño de malla de 1024x1024 el hacer uso de 15 ó 20 hilos paralelos mejora bastante el tiempo de ejecución, pues hay un reparto de trabajo mayor con respecto a tener sólo 10 hilos. Hay que destacar que ya con 15 hilos se obtiene una mejora importante y el hacer uso de un número mayor de hilos no aporta interés alguno, más que el interés por paralelizar con una granularidad más fina.

La **Tabla 4.2** muestra un resumen de los tiempos de ejecución obtenidos para Coons, observándose que las implementaciones paralelas para tamaños grandes de malla reducen drásticamente dichos tiempos, siendo CUDA la mejor opción. También se observa que para tamaños pequeños de malla, cualquier implementación es buena.

Resumen $t_{ejecución}$ (Coons)	Secuencial	OpenMP	CUDA
64x64	1 ms	1 ms	1 ms
512x512	45 ms	9 ms	5 ms
1024x1024	182 ms	24 ms	14 ms

Tabla 4.2. Resumen de los tiempos de ejecución para Coons

La **Tabla 4.3** muestra con claridad las aceleraciones obtenidas con las implementaciones paralelas, confirmando que el desarrollo con CUDA es la mejor opción, alcanzando un *speedup* de 13,0 para un tamaño de malla de 1024x1024. Esto significa que la implementación paralela es 13,0 veces más rápida que su versión secuencial.

Resumen Speedup (Coons)	OpenMP	CUDA
64x64	1,0	1,0
512x512	5,0	9,0
1024x1024	7,6	13,0

Tabla 4.3. Resumen de los *speedups* para Coons

La **Tabla 4.4** refleja los tiempos de ejecución de la implementación del método del suavizado Laplaciano según las diferentes versiones, sean secuencial, paralela con OpenMP y paralela con CUDA. Asimismo, también muestra los *speedups* alcanzados con los desarrollos paralelos. La diferencia con respecto a Coons, es que aquí se ha iterado varias veces hasta cumplirse un criterio de parada, siendo éste, como se comentó al principio de este apartado, que la proporción entre los nodos enredados en una iteración y los nodos totales fuera menor que un cierto umbral ε. Nótese entonces que los tiempos calculados son mucho mayores que los obtenidos para Coons.

Laplaciano

				Secuencial	OpenMP			CUDA			
				$t_{ejecución}$			$t_{ejecución}$	Speedup			$t_{ejecución}$	Speedup
Dimensiones del grid	64x64	Iteraciones (Máx. 30000)	140	50 ms	Hilos	10	71 ms	0,7	Hilos por bloque	16	30 ms	1,7
						15	88 ms	0,6		128	49 ms	1,0
						20	86 ms	0,6		512	49 ms	1,0
	512x512	Iteraciones (Máx. 30000)	10149	229.641 ms	Hilos	10	85.599 ms	2,7	Hilos por bloque	16	28.850 ms	8,0
						15	87.951 ms	2,6		128	36.148 ms	6,4
						20	86.751 ms	2,6		512	97.098 ms	2,4
	1024x1024	Iteraciones (Máx. 30000)	30000	2.721.983 ms	Hilos	10	735.992 ms	3,7	Hilos por bloque	16	238.478 ms	11,4
						15	682.464 ms	4,0		128	232.512 ms	11,7
						20	667.798 ms	4,1		512	606.651 ms	4,5

Tabla 4.4. Tiempos de ejecución y *speedups* alcanzados para Laplaciano

Otro dato a destacar de la **Tabla 4.4** es que se han obtenido también las iteraciones necesarias hasta alcanzar el criterio de parada, siendo como máximo las mostradas, independientemente de la versión (secuencial, paralela con OpenMP y paralela con CUDA).

Analizando la **Tabla 4.4** se observa que los tiempos en la implementación con OpenMP para un tamaño de malla de 64x64 son peores que su versión secuencial. Esto es debido al coste de lanzar los hilos y realizar la sincronización entre ellos. En el caso de la implementación con CUDA y para el mismo tamaño de malla se observa que el tiempo de ejecución aumenta a medida que se incrementa el número de hilos por bloque, lo que resulta comprensivo debido al alto coste de sincronización entre hilos existente. Siguiendo con CUDA y analizando los otros dos tamaños de malla, los datos reflejan que el aumentar demasiado el número de hilos por bloque hace que aumente también el tiempo de ejecución.

En el resumen de tiempos de la **Tabla 4.5**, se puede observar que para un tamaño de malla de 1024x1024 el tiempo se dispara a 2721983 ms en la versión secuencial del método, que son aproximadamente 45 minutos de cálculo para obtener la malla física. CUDA sigue siendo la mejor opción.

Resumen $t_{ejecución}$ (Laplaciano)	Secuencial	OpenMP	CUDA
64x64	50 ms	71 ms	30 ms
512x512	229.641 ms	85.599 ms	28.850 ms
1024x1024	2.721.983 ms	667.798 ms	232.512 ms

Tabla 4.5. Resumen de los tiempos de ejecución para Laplaciano

Con respecto a las aceleraciones obtenidas en las implementaciones paralelas del suavizado Laplaciano, la **Tabla 4.6** muestra con claridad que son bastante buenas para tamaños grandes de malla, obteniendo en el caso de CUDA y un tamaño de malla de 1024x1024, una aceleración aproximada de 12, lo que significa que la implementación paralela es casi 12 veces más rápida que su versión secuencial.

Nótese que para un tamaño de malla de 64x64, un tamaño muy pequeño de malla, incluso se tiene una deceleración, ya que en OpenMP el tiempo de ejecución paralela es más alto que la versión secuencial, debido, como ya se comentó en líneas superiores, al tiempo de sincronización entre los diferentes hilos de ejecución.

Resumen Speedup (Laplaciano)	OpenMP	CUDA
64x64	0,7	1,7
512x512	2,7	8,0
1024x1024	4,1	11,7

Tabla 4.6. Resumen de los *speedups* para Laplaciano

Las tablas siguientes muestran los valores obtenidos aplicando primero Coons y luego el suavizado Laplaciano. Se observa que el tiempo de ejecución se reduce drásticamente con respecto a utilizar únicamente el Laplaciano, ya que Coons devuelve una muy buena aproximación de los nodos interiores de cara a aplicar un número de iteraciones en el Laplaciano. Estas iteraciones, si se comparan con las obtenidas en el Laplaciano, son muy inferiores.

Coons+Laplaciano

				Secuencial	OpenMP			CUDA			
				$t_{ejecución}$			Speedup			$t_{ejecución}$	Speedup
Dimensiones del grid	64x64	Iteraciones (Máx. 30000)	9	6 ms	Hilos	10	18 ms	0,3	Hilos por bloque	16	3 ms	2,0
						15	16 ms	0,4		128	4 ms	1,5
						20	18 ms	0,3		512	4 ms	1,5
	512x512	Iteraciones (Máx. 30000)	533	12.184 ms	Hilos	10	4.696 ms	2,6	Hilos por bloque	16	1.522 ms	8,0
						15	4.659 ms	2,6		128	1.902 ms	6,4
						20	4.958 ms	2,5		512	5.136 ms	2,4
	1024x1024	Iteraciones (Máx. 30000)	3052	194.616 ms	Hilos	10	75.581 ms	2,6	Hilos por bloque	16	26.946 ms	7,2
						15	70.538 ms	2,8		128	25.889 ms	7,5
						20	69.633 ms	2,8		512	63.351 ms	3,1

Tabla 4.7. Tiempos de ejecución y *speedups* alcanzados para Coons+Laplaciano

Resumen $t_{ejecución}$ (C+L)	Secuencial	OpenMP	CUDA
64x64	6 ms	16 ms	3 ms
512x512	12.184 ms	4.659 ms	1.522 ms
1024x1024	194.616 ms	69.633 ms	25.889 ms

Tabla 4.8. Resumen de los tiempos de ejecución para Coons+Laplaciano

Resumen Speedup (C+L)	OpenMP	CUDA
64x64	0,4	2,0
512x512	2,6	8,0
1024x1024	2,8	7,5

Tabla 4.9. Resumen de los *speedups* para Coons+Laplaciano

Por último, se adjunta una serie de figuras que muestran de forma gráfica todo lo comentado anteriormente.

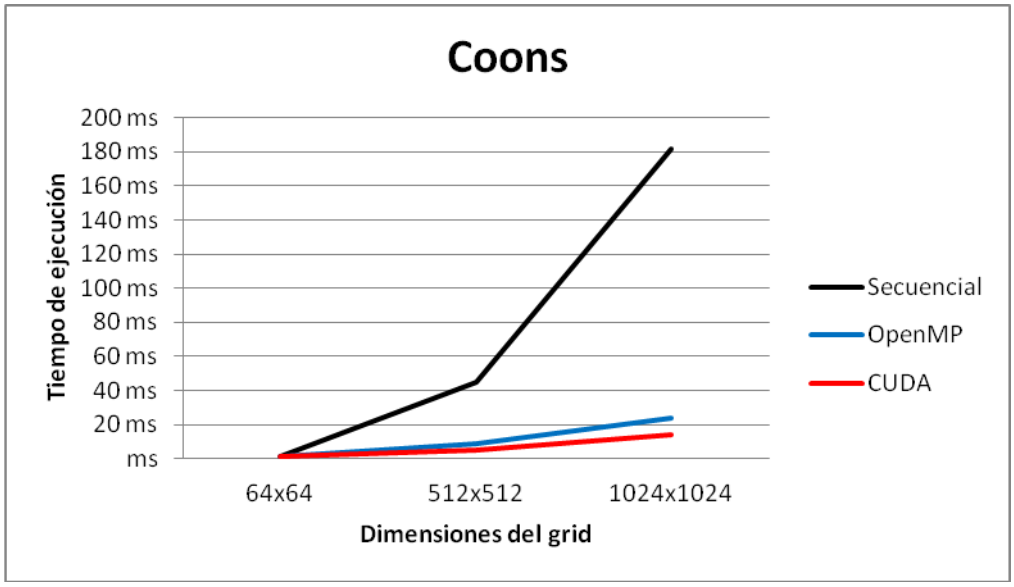

Figura 4.5. Comparativa de tiempos entre las diferentes implementaciones de Coons

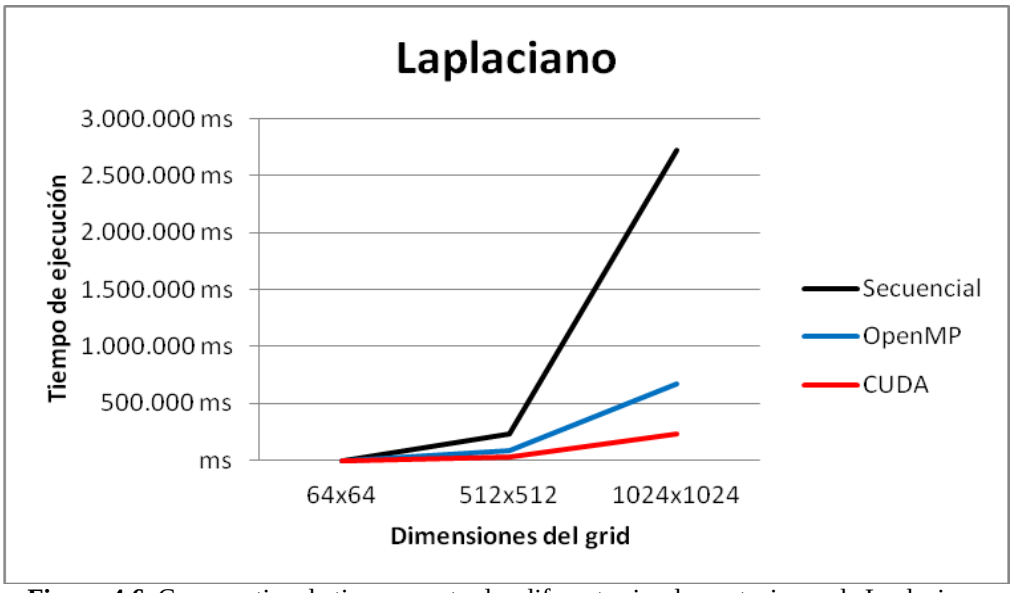

Figura 4.6. Comparativa de tiempos entre las diferentes implementaciones de Laplaciano

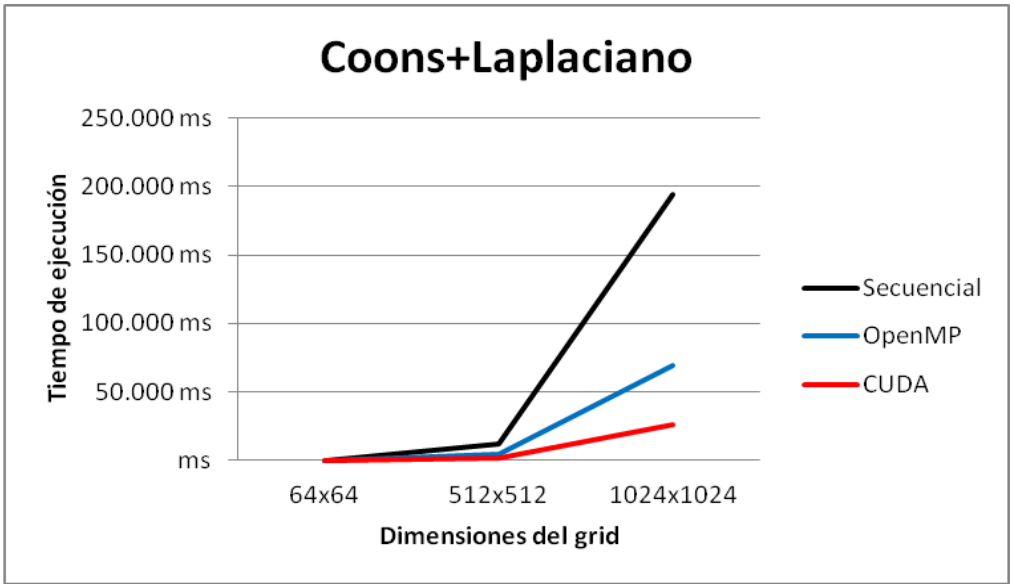

Figura 4.7. Comparativa de tiempos entre las diferentes implementaciones de Coons+Laplaciano

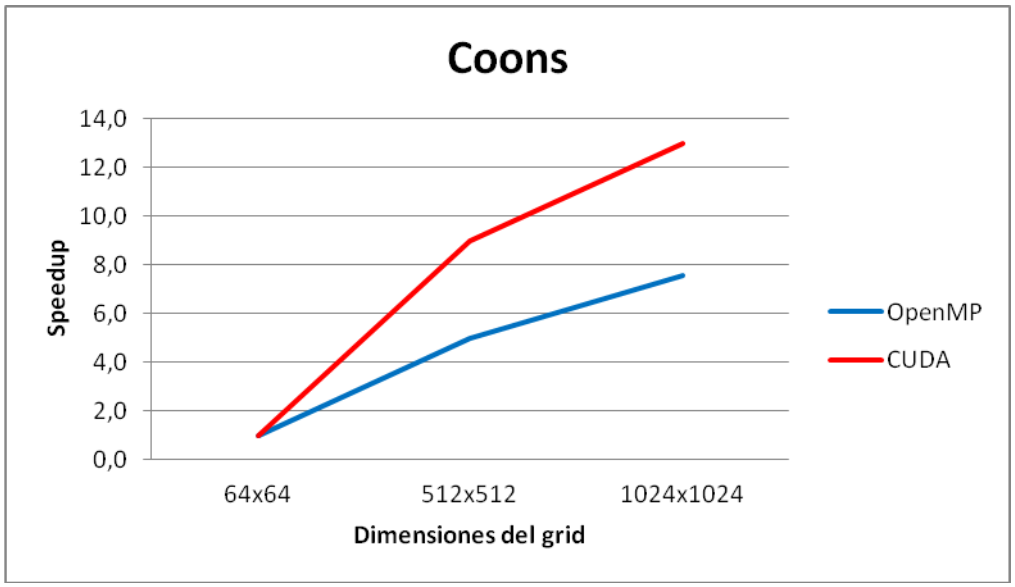

Figura 4.8. Comparativa de *speedups* entre las diferentes implementaciones con paralelización de Coons

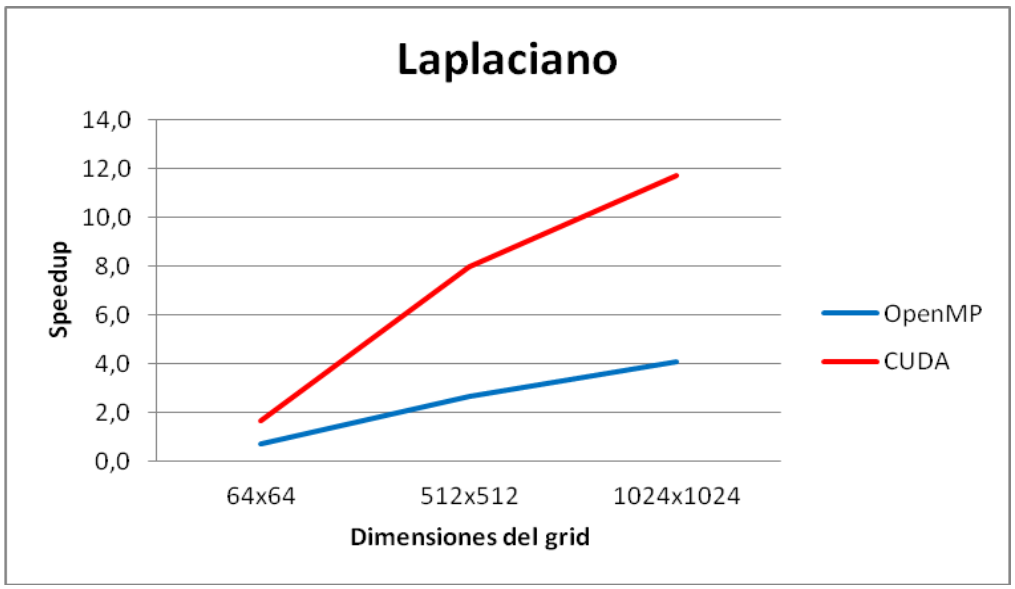

Figura 4.9. Comparativa de *speedups* entre las diferentes implementaciones con paralelización de Laplaciano

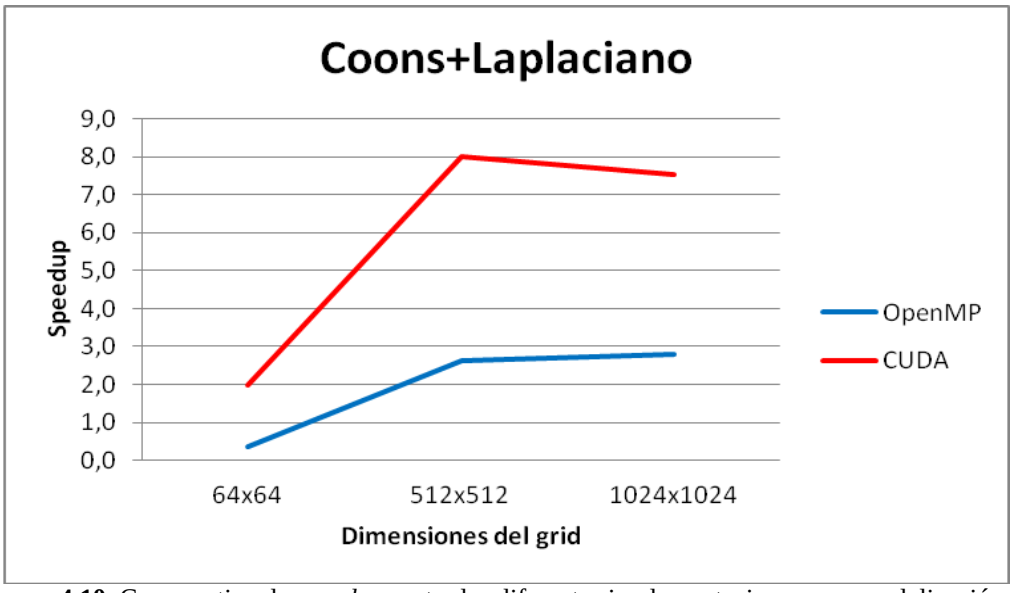

Figura 4.10. Comparativa de *speedups* entre las diferentes implementaciones con paralelización de Coons+Laplaciano

Ejemplos de mallas 2D obtenidas

La **Figura 4.11** ilustra la idea de este Trabajo de Fin de Máster: construir una malla uniforme en el espacio paramétrico 2D y transformarla con una buena aproximación de la geometría a una malla, topológicamente equivalente, en el espacio físico 2D. Como ejemplo, se puede considerar la construcción de una malla que represente la isla de Gran Canaria, cuya geometría es irregular, a partir de una rejilla regular definida en el espacio paramétrico (un cuadrado).

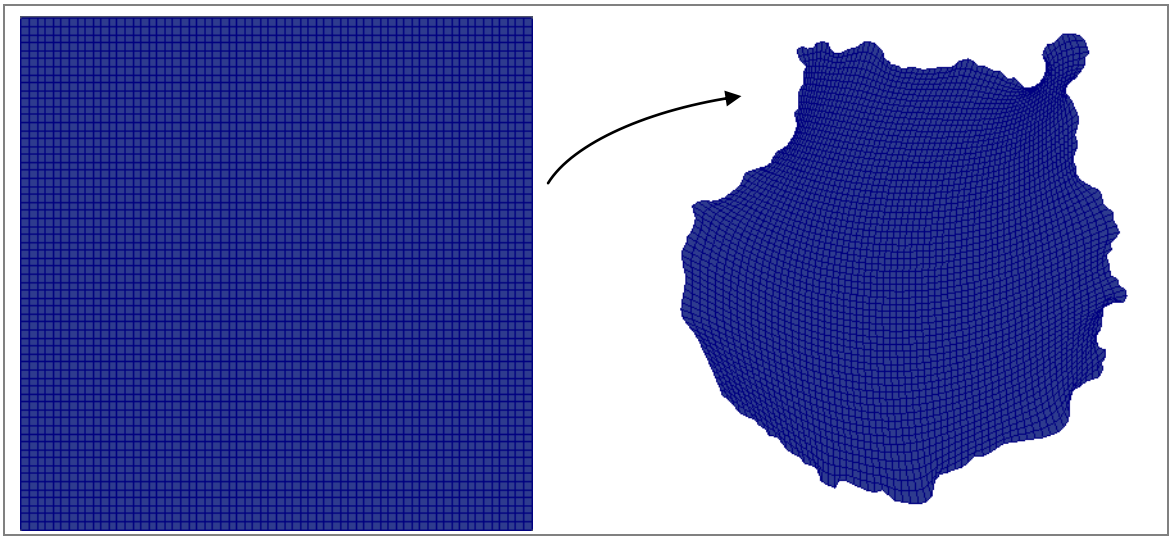

Figura 4.11. Mallas paramétrica (izq.) y física (der.) para un tamaño de 64x64

Diversos son los métodos que permiten realizar una transformación entre el espacio paramétrico y el espacio real. Entre ellos se han estudiado los *parches de Coons* y la transformación mediante el *suavizado Laplaciano*.

Como se aprecia en la **Figura 4.12** y en la **Figura 4.14**, Coons no desenreda la malla que se obtiene tan bien como lo hace el Laplaciano, como se muestra en la **Figura 4.13** y en la **Figura 4.15**. La transformación mediante el suavizado Laplaciano es capaz de obtener

una malla mucho más desenredada, pero no totalmente desenredada, para lo cual habría que aplicar otro método adicional, que no se ha contemplado como objetivo de este Trabajo de Fin de Máster.

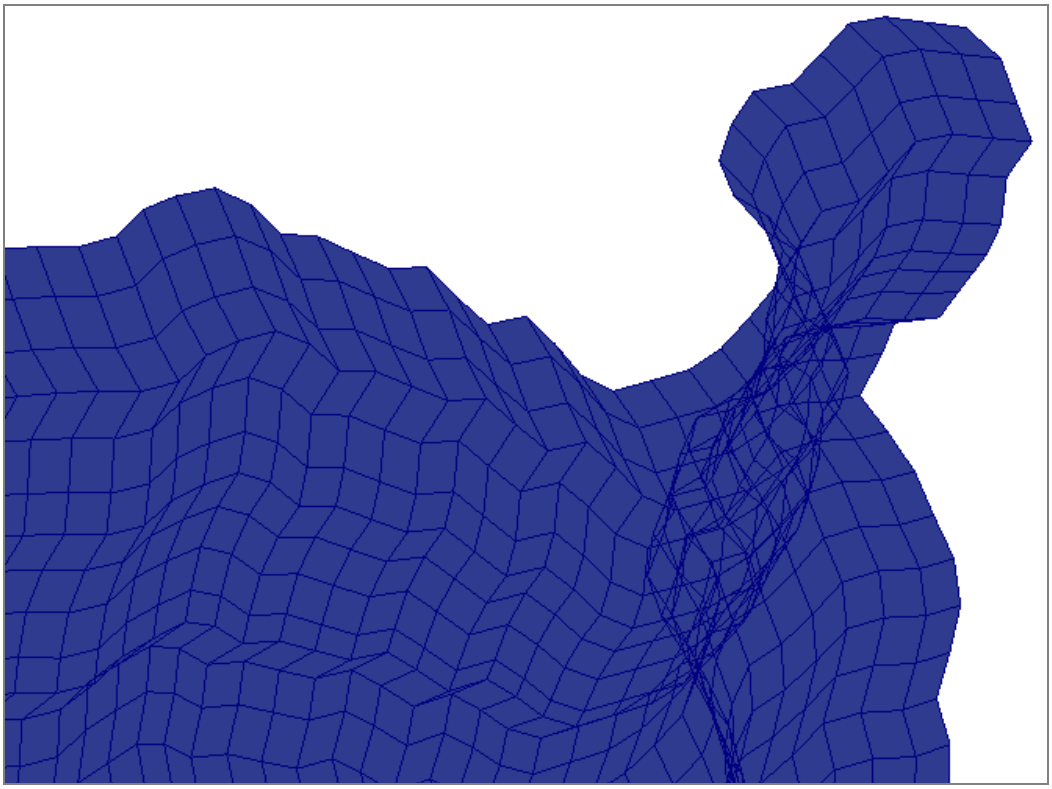

Figura 4.12. Nodos enredados en la malla de Gran Canaria (64x64).
Malla física obtenida con Coons

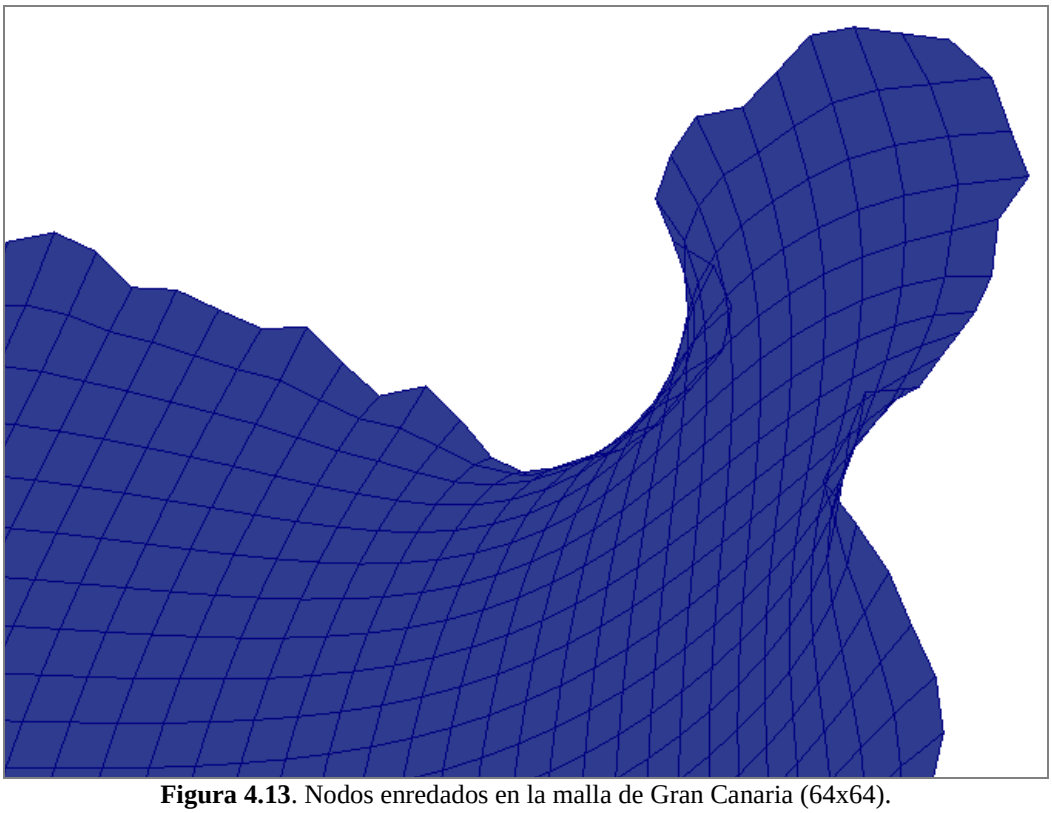

Figura 4.13. Nodos enredados en la malla de Gran Canaria (64x64).
Malla física obtenida con Laplaciano

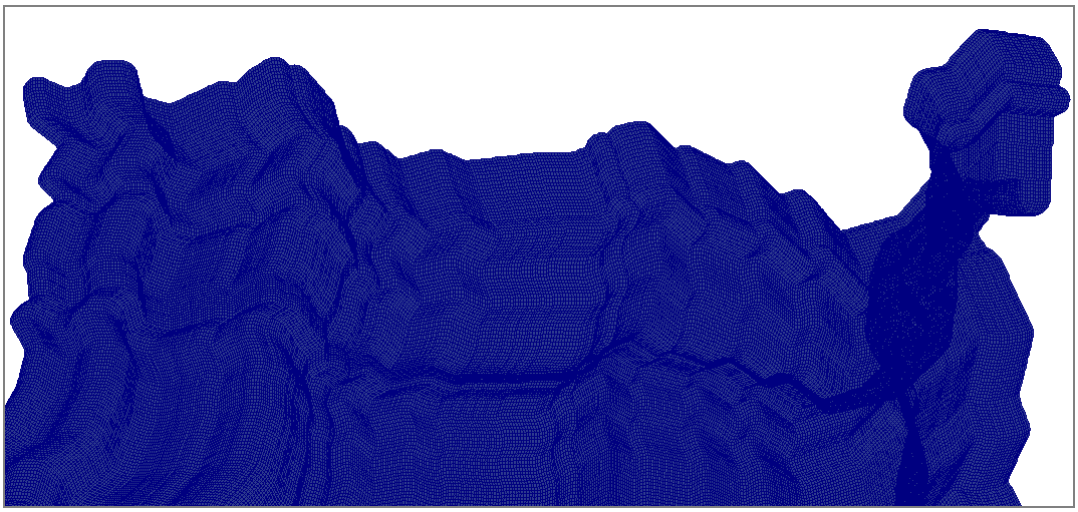

Figura 4.14. Sección de la malla fisica de la isla de Gran Canaria para un tamaño de 512x512. Malla física obtenida con Coons

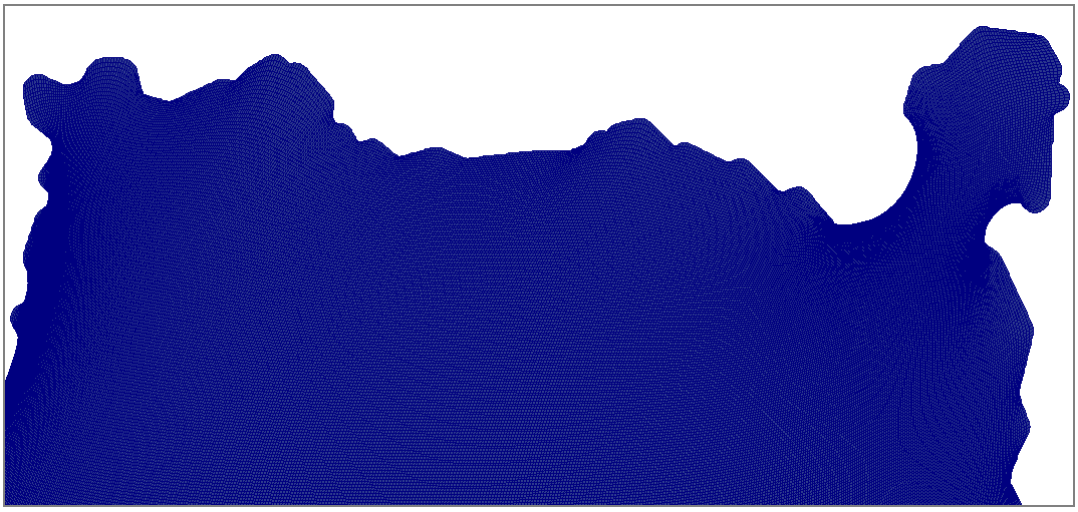

Figura 4.15. Sección de la malla fisica de la isla de Gran Canaria para un tamaño de 512x512. Malla física obtenida con Laplaciano

Las siguientes dos figuras muestran el resultado de aplicar en primer lugar Coons y a continuación una serie de repeticiones de Laplaciano para obtener la malla física de la isla de Gran Canaria para un tamaño de 512x512.

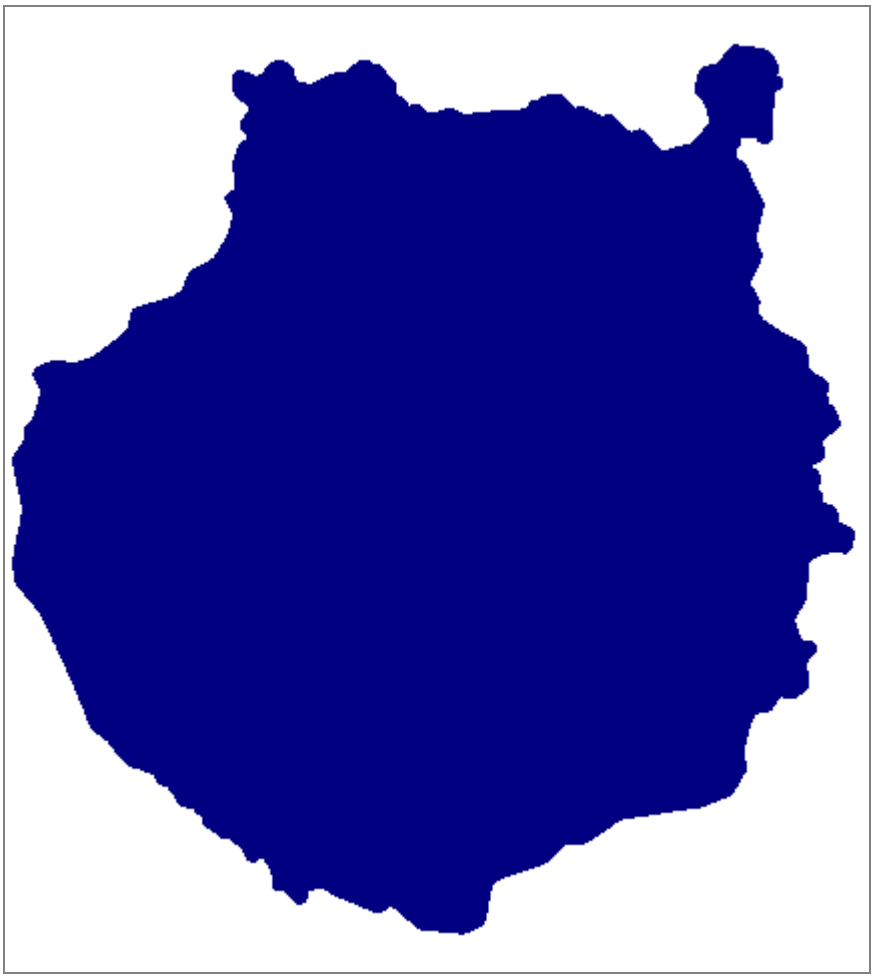

Figura 4.16. Malla fisica de la isla de Gran Canaria para un tamaño de 512x512. Malla física obtenida con Coons+Laplaciano

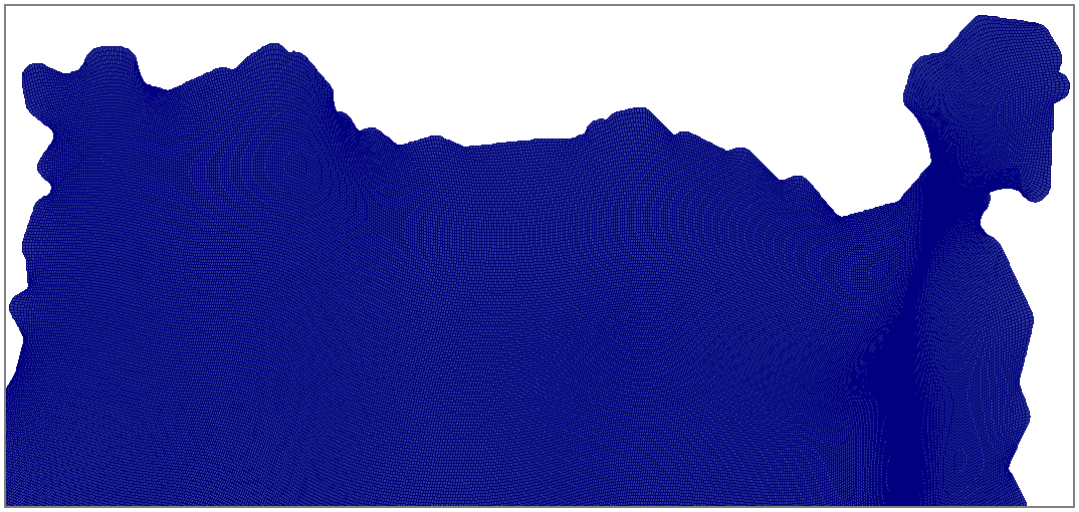

Figura 4.17. Sección de la malla fisica de la isla de Gran Canaria para un tamaño de 512x512. Malla física obtenida con Coons+Laplaciano

No sólo hay que considerar la isla de Gran Canaria como ejemplo para la construcción de la malla en el espacio físico a partir de una rejilla regular definida en el espacio paramétrico (un cuadrado). En este Trabajo de Fin de Máster también ha sido interesante estudiar algunas otras geometrías como las mostradas en la **Figura 4.18**.

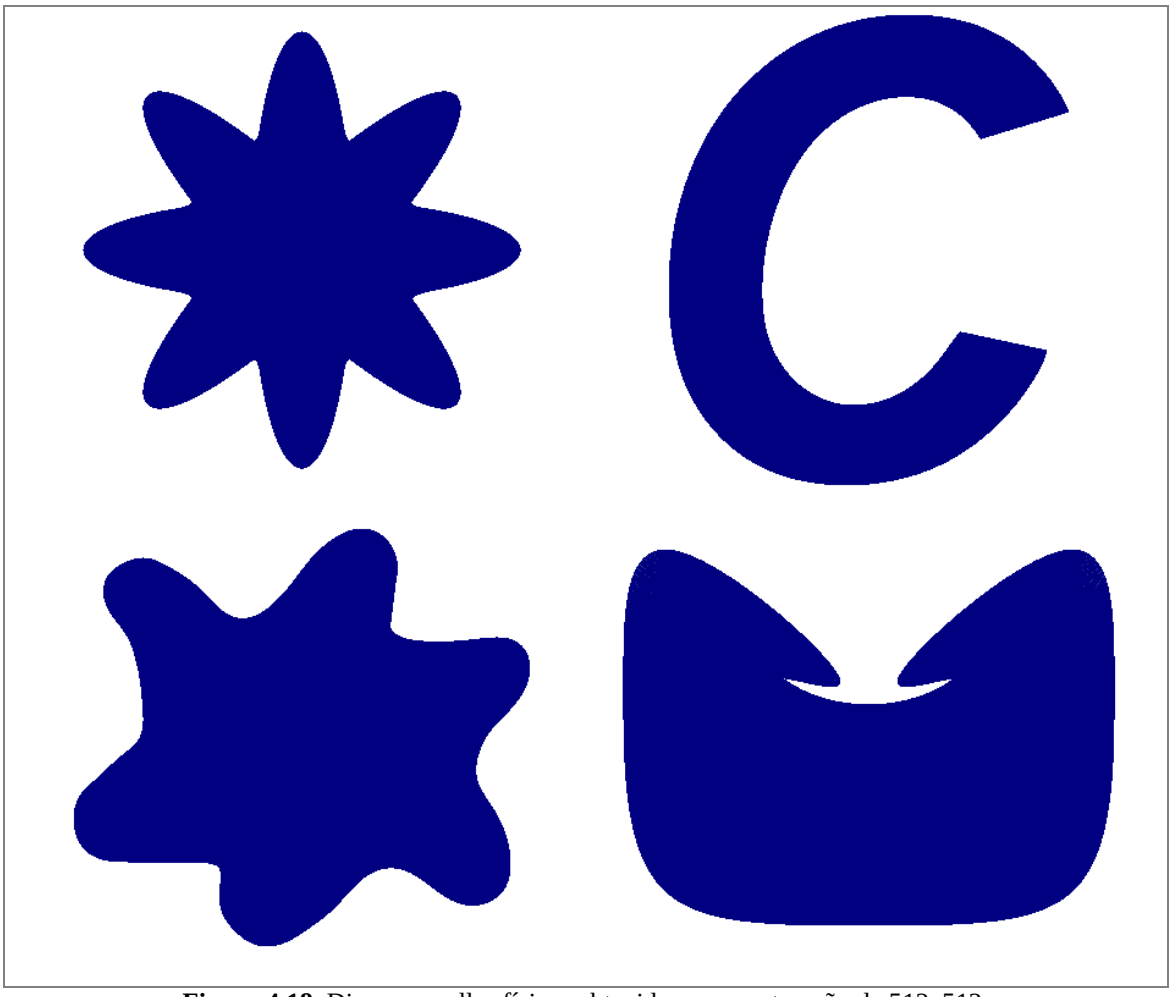

Figura 4.18. Diversas mallas físicas obtenidas para un tamaño de 512x512.
Mallas físicas obtenidas con Coons+Laplaciano

Hay que destacar que las mallas obtenidas con las implementaciones desarrolladas son mallas de calidad, pero no totalmente desenredadas. Para realizar el desenredo total de una malla se tendría que hacer uso de algún software para desenredo, tal como podría ser *SUS Code* [SUS], software implementado para el desenredo y la optimización simultáneos de mallas de tetraedros.

5. CONCLUSIONES

La idea principal sobre la que se ha centrado este Trabajo de Fin de Máster ha sido la de construir una malla uniforme en el espacio paramétrico 2D y transformarla con una buena aproximación de la geometría a una malla, topológicamente equivalente, en el espacio físico 2D. Como ejemplo bastante recurrido, se ha considerado la construcción de una malla representando la isla de Gran Canaria, cuya geometría es irregular, a partir de una rejilla regular definida en el espacio paramétrico (un cuadrado). Diversos son los métodos que permiten realizar una transformación entre el espacio paramétrico y el espacio real. Entre ellos, se han destacado y estudiado los *parches de Coons* y la transformación mediante el *suavizado Laplaciano*.

El estudio de los métodos de optimización de mallas escogidos se ha centrado en la implementación paralela de los mismos con tecnologías paralelas tales como OpenMP y CUDA, realizando comparativas entre las implementaciones obtenidas y, sobre todo, observando su mejora con respecto a la versión secuencial de los métodos. Este estudio demostró que las tecnologías paralelas mejoran considerablemente los tiempos de ejecución cuando se trabaja con tamaños de malla elevados, no ocurriendo lo mismo si los tamaños son pequeños, donde la versión secuencial es la mejor opción. Asimismo, se constató que la tecnología CUDA va muy por delante de la tecnología OpenMP, alcanzando aceleraciones o speedups de hasta 13 puntos con respecto a la versión secuencial.

Los *parches de Coons* ofrecen una muy buena optimización de la malla con una única ejecución y es aquí donde se alcanzan aceleraciones de hasta 13 puntos con respecto a la versión secuencial, haciendo uso de CUDA y de 128 hilos por bloque para una malla de gran tamaño como es 1024x124. Este método tiene gran poder de paralelización, ya que las nuevas posiciones de los nodos interiores pueden ser obtenidas todas a las vez, puesto que no hay dependencia entre ellas.

La transformación mediante el *suavizado Laplaciano* tiene la particularidad de que obtiene unas mallas más desenredadas que las obtenidas por Coons, a expensas de tener que ejecutar dicho suavizado varias veces para poder obtener una malla de calidad que represente la geometría deseada. El uso de este método sí requiere la utilización de una versión paralela, sea OpenMP o CUDA (con preferencia de esta última), puesto que los tiempos en versión secuencial son muy elevados.

El hacer uso de la combinación de ambos métodos (Coons primero, Laplaciano después) permite obtener en poco tiempo y con pocas iteraciones del Laplaciano una malla de muy buena calidad y con un pequeño número de nodos enredados con respecto al tamaño de ésta. CUDA sigue siendo la mejor opción de implementación para tamaños grandes de malla.

Recalcar nuevamente que la paralelización es la mejor opción para tamaños de malla elevados, siendo la versión secuencial la mejor opción cuando se trabaja con mallas de pequeñas dimensiones.

Hay que destacar que las mallas obtenidas con las implementaciones desarrolladas son mallas de calidad, pero no totalmente desenredadas. Para realizar el desenredo total de

una malla se tendría que hacer uso de algún método adicional, que no se ha contemplado como objetivo de este Trabajo de Fin de Máster. Este método adicional podría ser el aportado por *SUS Code* [SUS], software implementado para el desenredo y la optimización simultáneos de mallas de tetraedros.

Por último, comentar que, tal vez, cabría preguntarse si el uso de varias GPUs podría acelerar aún más la ejecución de los métodos. Después de las diversas pruebas realizadas se constató que no. El hacer uso de varias GPUs significa que también debe haber sincronización entre ellas al dividir el trabajo a realizar, lo que no incrementa la aceleración, sino que la disminuye, al aumentar el tiempo de ejecución por esa sincronización.

6. LÍNEAS DE TRABAJO FUTURAS

Actualmente, el criterio de parada utilizado en el *suavizado Laplaciano* es que la proporción entre el número de nodos enredados en una iteración i y el número de nodos total tiene que ser menor que un cierto porcentaje:

$$\frac{N_{enredos}^{i}}{N_{nodos}} < \varepsilon$$

Un trabajo futuro a abordar sería considerar como criterio de parada la calidad de la malla física obtenida, entendida aquélla como la calidad de los ángulos de sus elementos que, al ser rectangulares, deberían ser lo más cercanos a 90º, por no decir de 90º exactos (**Figura 6.1**).

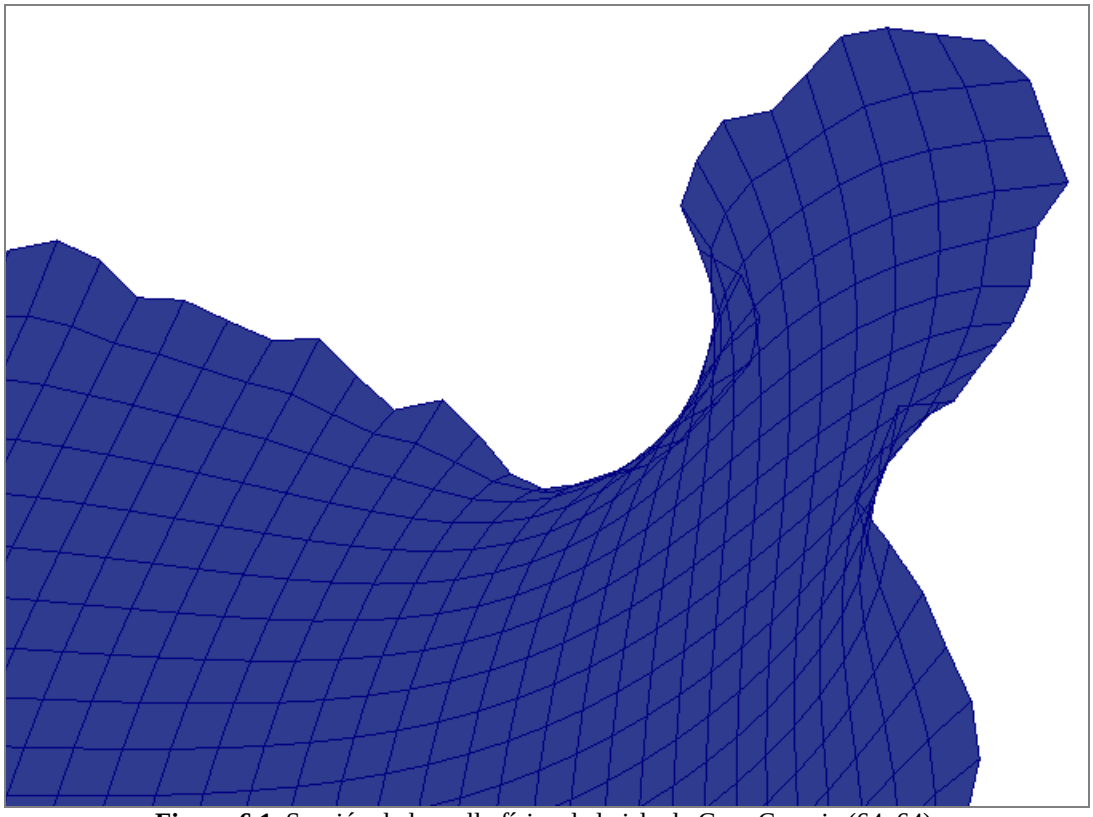

Figura 6.1. Sección de la malla física de la isla de Gran Canaria (64x64).
Los ángulos de los elementos rectangulares distan de ser de 90º

Otra línea de trabajo futura interesante sería paralelizar con CUDA una versión para mallas 2D del *SUS CODE* [SUS] lo que, aplicado a lo ya implementado en este Trabajo de Fin de Máster, significaría una importante herramienta de suavizado, optimización y desenredo de mallas 2D.

Por último, y ya como un proyecto bastante ambicioso, cabría la posibilidad de abordar todo lo implementado considerando mallas 3D.

7. REFERENCIAS

[ACM06]. Association for Computing Machinery, Inc. "*Laplacian Mesh Optimization*". 2006. http://www.cs.jhu.edu/~misha/Fall07/Papers/Nealen06.pdf

[AGMV08]. Francisco Almeida, Domingo Giménez, José Miguel Mantas, Antonio M. Vidal. "*Introducción a la Programación Paralela*". PARANINFO CENGAGE Learning. 2008

[B13]. Blaise Barney. "*OpenMP*". 2013. https://computing.llnl.gov/tutorials/openMP/

[FH99]. Gerald Farin, Dianne Hansford. "*Discrete Coons patches*". 1999. http://prism.asu.edu/publications/papers/paper99_dcp.pdf

[KH10]. David B. Kirk, Wen-mei W. Hwu. "*Programming Massively Parallel Processors*". Morgan Kaufmann. 2010

[MM10]. José Miguel Mantas Ruiz, Francisco Javier Melero Rus. "*Programación Distribuida y Paralela*". 2010. http://lsi.ugr.es/~jmantas/pdp/

[N13]. NVIDIA. "*CUDA*". 2013. http://www.nvidia.es/object/cuda-parallel-computing-es.html

[OMP]. "*The OpenMP API specificaction for parallel programming*". http://openmp.org/wp/

[SUS]. "*SUS Code. Simultaneous untangling and smoothing of tetrahedral meshes*". http://www.ana.iusiani.ulpgc.es/proyecto2012-2014/html/Software.html