Universidad Politécnica de Valencia
Departamento de Sistemas Informáticos y Computación

Máster en Computación Paralela y Distribuida

PARALELIZACIÓN DE LA HERRAMIENTA DE SIMULACIÓN ELECTROMAGNÉTICA FEST3D

Autor: Mario Rodríguez Maya
Director: Víctor Manuel García Mollá

Valencia, Diciembre de 2010

Agradecimientos

Sin mis compañeros y amigos de AURORASAT, que en todo momento han estado y están pendientes de mi (Jordi, Carlos, Jaime, Sergio, Teresa, Javi, María, Carlos, Iñaqui, Carmen) y sin la ayuda técnica y a veces moral que me dan, no habría conseguido terminar este trabajo, sinceramente.
También a Javi Monge por aguantarme al principio, a mi director Víctor Manuel, sin ellos sería imposible también. A Vicente y a Benito.

A mis padres por todo.

Paralelización de la herramienta electromagnética FEST3D

Índice de contenido

1 Introducción

1.1 Estado del arte

La alta demanda de servicios de telecomunicaciones obliga a la industria a elaborar sistemas cada vez más complejos; hoy en día, resulta impensable que la industria espacial pueda desarrollar *hardware* sin el apoyo de herramientas informáticas para el diseño y optimización de sus dispositivos. Este tipo de herramientas evitan las técnicas clásicas de desarrollo de nuevos dispositivos basadas en sucesivas implementaciones prácticas y medidas de prototipos, minimizando, de este modo, los costes y tiempos de diseño y producción de nuevos componentes y facilitando, en consecuencia, la competitividad de las empresas fabricantes en el mercado actual.

Los satélites tienden a ser más complejos debido a este aumento de la demanda de servicios y a la necesidad de utilizar más potencia de salida para poder cumplir los requisitos de los clientes. Por esta razón, cada vez se necesita realizar diseños más avanzados y capaces de analizar el impacto del uso de potencias elevadas en la integridad del componente. Para ello, es indispensable disponer de herramientas informáticas que permitan al ingeniero conocer el nivel máximo de potencia que un componente es capaz de manejar sin ver deteriorado su comportamiento para no tener que recurrir a campañas de puesta a prueba que son extraordinariamente largas y caras.

Es por ello que la industria espacial está especialmente interesada en tener una plataforma de diseño por ordenador capaz de diseñar componentes para su uso no solamente a baja potencia, con el fin de conocer simplemente su respuesta en frecuencia, sino también para su uso a alta potencia.

Las empresas del sector no suelen disponer del tiempo ni de los recursos requeridos, básicamente de formación, para desarrollar ciertas herramientas informáticas, que por su carácter innovador e integrador de conocimientos científico-tecnológicos, deben llevarse a cabo en plena colaboración con grupos de investigación universitarios.

En este contexto, nos encontramos con empresas como Aurora Software and Testing (AURORASAT), una empresa de base tecnológica y reciente creación (fundada en enero 2006), entre cuyos objetivos está la producción de *software* para el análisis y diseño de dispositivos pasivos de RF (radiofrecuencia), y para la predicción de los efectos de la descarga eléctrica de señales a niveles de alta potencia en estos componentes para su uso por las industrias del sector espacial a nivel mundial.

AURORASAT, en uno de los contratos que tiene actualmente en ejecución con la Agencia Espacial Europea (ESA), está mejorando y distribuyendo FEST3D propiedad de la ESA. Dicho software es una herramienta CAD (*Computer-Aided Design*), capaz de analizar componentes pasivos de microondas en tecnología guía onda de una manera extremadamente eficiente y precisa.

FEST3D está orientado al análisis y diseño de componentes pasivos de microondas y ondas

milimétricas así como al análisis de efectos de alta potencia para la posterior fabricación industrial de dichos elementos.
Cuenta con una sección de análisis que permite simular el comportamiento electromagnético de un gran número de componentes pasivos de radiofrecuencia.
Desde el punto de vista del diseño de circuitos pasivos de microondas, esta herramienta consta de dos secciones. Una primera de optimización que permite al usuario elegir como parámetros las características del circuito que considere oportuno para buscar así con distintos algoritmos de optimización la respuesta electromagnética deseada. Y una segunda sección, la de síntesis, en la que el usuario dispone de ciertas topologías específicas, y con ellas obtiene la geometría física de un circuito a partir de la respuesta electromagnética requerida.
FEST3D posee dos cualidades fundamentales imprescindibles en cualquier herramienta informática de diseño industrial: precisión y rapidez.

Los algoritmos electromagnéticos que se emplean en FEST3D presentan ventajas frente a los algoritmos empleados por otras herramientas existentes en el mercado, ya que combina flexibilidad (analiza geometrías arbitrarias), eficiencia (las técnicas que se desarrollan presentan una gran precisión y un óptimo uso de recursos computacionales) con capacidad de síntesis y optimización avanzadas.

Uno de los puntos más fuertes del software es el desarrollo de algoritmos capaces de analizar y predecir el fenómeno de tensión de ruptura en aplicaciones de alta potencia. Esta prestación permitirá que los desarrolladores diseñen componentes de RF asegurando que estén exentos de esos fenómenos que pueden degradar los dispositivos de los satélites e incluso destruirlos.

La oferta en el mercado de un *software* capaz de predecir efectos de ruptura a alta potencia es muy limitada e incompleta en el mercado. Por lo tanto, la industria espacial y la comunidad científica deben realizar amplias y caras campañas de ensayos para cualificar todos sus equipos para los vuelos espaciales. El uso de un software como FEST3D ayuda a la industria espacial a diseñar nuevos componentes pasivos en procesos más cortos y con menores costes de fabricación.

En la figura 1 se presenta una captura de pantalla del software FEST3D. En ella se aprecia un circuito de FEST3D, una representación 3D del mismo, y su respuesta electromagnética en forma de gráfica.

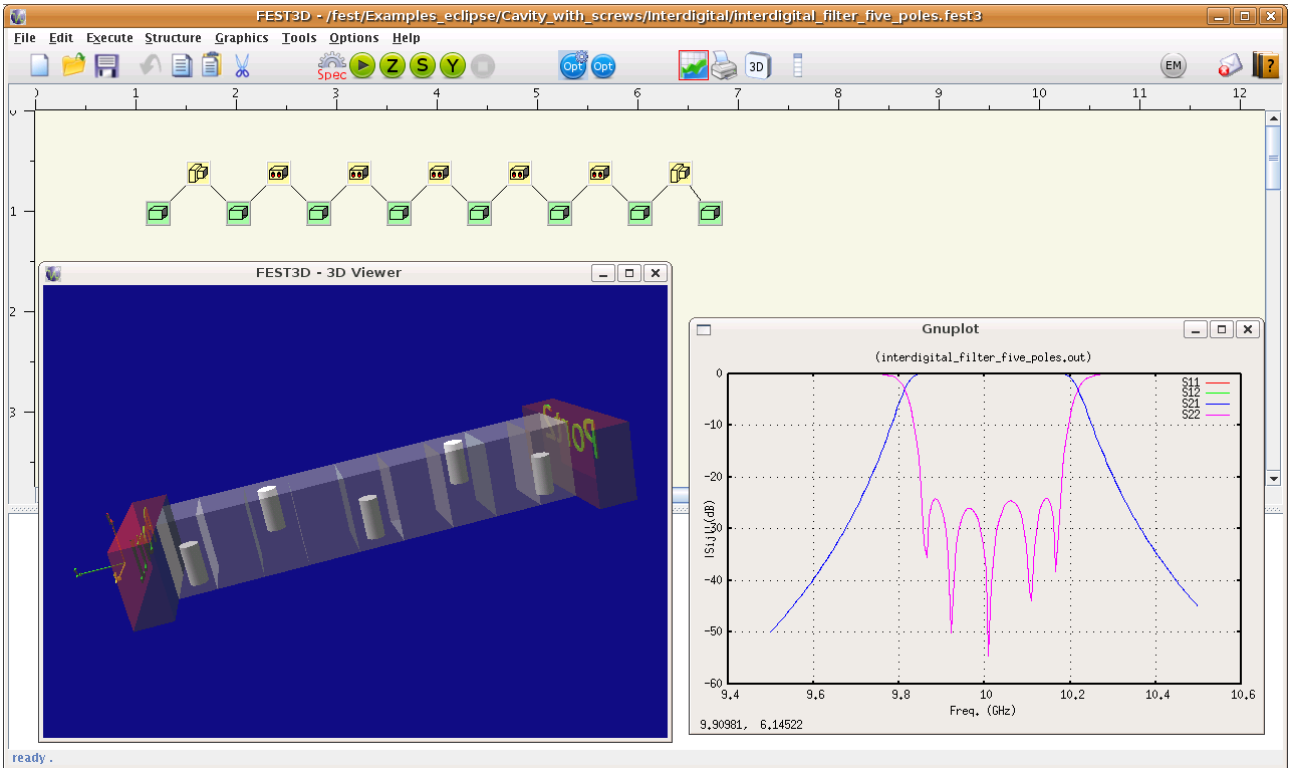

Figura 1: Captura de pantalla de FEST3D

1.2 Motivación

En la actualidad existen varias empresas que se dedican a desarrollar herramientas CAD para el diseño de circuitos pasivos de microondas y ondas milimétricas. Muchas de estas empresas están bien asentadas en el mercado y proporcionan software muy versátil basado en técnicas de segmentación capaces de analizar estructuras muy complejas. Ejemplos de éstos son HFSS de ANSOFT Co., FEMLAB de Comsol Inc., CST Microwave Studio de Computer Simulation Technology Co. Sin embargo, los más directos competidores de FEST3D son programas con otro tipo de algoritmos numéricos (técnicas modales). Ejemplos de este tipo de productos son uWave Wizard de Mician GmbH y Wasp-net de MIG Microwave Innovation Group. La ventaja de este segundo grupo de software es que estas herramientas, a pesar de ser de propósito menos general (no son capaces de analizar cualquier tipo de estructura), utilizan recursos computacionales y tiempos de cálculo sustancialmente menores e incluso contienen partes de código paralelizadas.

En este contexto, AURORASAT ha detectado dos posibles mejoras para hacer que el software que distribuye sea más competitivo que los productos existentes:

- La capacidad de análisis de estructuras complejas sin recurrir a cálculos intensivos por cada punto en frecuencia de la banda de trabajo.

- La resolución óptima de problemas algebraicos y paralelización del código implementado.

El presente proyecto incide, precisamente en la segunda de las mejoras, paralelizando FEST3D para ajustarse a las necesidades competitivas del mercado y a la creciente oferta de procesadores multinúcleo.

1.3 Objetivos

De partida, el objetivo principal consiste en crear un primer nivel de paralelización que incluya ciertas partes del código, dejando otras para un futuro. Además, se trata de crear un código mantenible a la hora de añadir funcionalidades al software y a la hora de paralelizar a niveles más internos, con el fin de favorecer la escalabilidad.

La paralelización está pensada para un modelo de memoria compartida.
En principio, no se marca ningún objetivo acerca de los resultados que se van a obtener ya que se prevén demasiado dependientes del circuito, tipo y número de elementos, etc.

2 Modelos de paralelismo

Existen varias clasificaciones que engloban los distintos tipos de paralelismo.

- Según el tipo, el número de procesadores y teniendo en cuenta el grano de las operaciones realizadas por los procesadores. Por ejemplo, máquinas masivamente paralelas (MPP).

- Según el sistema de sincronización de procesadores: Desde sistemas muy acoplados, en los que se ejecuta la misma instrucción concurrentemente, hasta sistemas totalmente desacoplados (clusters).

- Según la forma de interconexión de los procesadores: Procesadores comunicados mediante memoria compartida o a través de mensajes.

En concreto, la clasificación que más se suele utilizar para describir el tipo de paralelismo que existe en una máquina es la taxonomía de Flynn; se trata de una clasificación que atiende a dos factores: el flujo de instrucciones y el flujo de datos.
Según esta, nos encontramos con los siguientes sistemas:

- Sistemas SISD (*Single Instruction, Single Data*):

Son los sistemas de computación tradicional de PC. No hay paralelismo y las instrucciones se ejecutan consecutivamente.

- Sistemas SIMD (*Single Instruction, Multiple Data*):

Son los computadores vectoriales. Su paralelismo basa su funcionamiento en la existencia de registros vectoriales y ALUs paralelas, que ejecutan una misma operación sobre todo un vector de datos. Por ejemplo: Instrucciones multimedia MMX, SSE...

- Sistemas MIMD (*Multiple Instruction, Simple Data*):

Ejecutan instrucciones sobre distintos datos de forma paralela. Para ello, se necesita una unidad de proceso completa que sea capaz de ejecutar distintos flujos de programa. Por ejemplo: clúster, multicore.

Dependiendo del tipo de concurrencia, puede ser útil estimar el coste de un algoritmo paralelo, con el fin de poderlos comparar teóricamente. Dos de los modelos de coste más utilizados se exponen a continuación.

2.1 Modelos de coste

El concepto de coste se puede expresar en Flops (Operaciones en punto flotante por segundo). El

coste computacional se expresará en la medida de Flops relativa a la talla del problema.

- Modelo PRAM (Parallel Random Access Machine) - Modelo de memoria compartida

  Consiste en que un número *p* de procesadores acceden de manera síncrona a la memoria, teniendo todos ellos acceso completo a la memoria.

- Modelo DRAM (Distributed Random Access Machine) - Modelo de memoria distribuida

  Un número *p* de procesadores se comunican a través de una red de interconexión mediante el intercambio de mensajes. Cada procesador tiene acceso a una única área de memoria.

2.2 Conceptos de paralelismo

A modo aclaratorio se explican a continuación ciertos conceptos que pueden ser utilizados más adelante explicando los algoritmos desarrollados.

- Tiempo en secuencial (T_1): tiempo que tarda un algoritmo en ejecutarse en secuencial.

- Tiempo paralelo (T_p): es lo que tarda un algoritmo en ejecutarse con *p* procesadores. Medido en Flops y experimentalmente en segundos.

- Tiempo de comunicaciones de paso de mensajes (T_c): en el modelo PRAM no se maneja este concepto.

- Tiempo de contención (Tm_n): es el tiempo consumido en las sincronizaciones para el acceso concurrente a memoria (PRAM). Es muy difícil de medir tanto teórica como experimentalmente.

- Tiempo aritmético (T_a): tiempo que tarda un algoritmo paralelo en realizar solamente las operaciones aritméticas, sin contabilizar T_c ni Tm_n.

- Condición de carrera: existe una condición de carrera cuando hay varios procesadores intentando acceder a la vez a un mismo sector de memoria, pudiendo producir inconsistencias o interbloqueos.

- Balanceo de carga: en un algoritmo paralelo, es fundamental que el trabajo que se va a realizar se distribuya lo más equitativamente posible entre los distintos procesadores. Así, si el número de operaciones a realizar en un algoritmo es calculable, se puede hacer un estudio del equilibrio de la carga.

$$Eq_p = T_a(p) - \frac{T_1}{p}$$

- Speed-up/Eficiencia: el *Speed-up* nos da el factor de ganancia de prestaciones de un determinado algoritmo paralelo con respecto a su mejor algoritmo secuencial conocido, o sobre el mismo algoritmo ejecutado con un solo procesador.

$$S_p = \frac{T_1}{T_p}$$

Speed-up máximo teórico que se puede conseguir es *Sp = p*. Un *Speed-up* mayor que *p* se podría dar en el caso de tener una mayor cantidad de recursos disponibles, por ejemplo en cantidad de memoria caché por cada procesador.

La eficiencia es el grado de aprovechamiento que un algoritmo hace de cada procesador.

$$E_p = \frac{S_p}{p}$$

2.3 Ley de Amdahl

La mayoría de los algoritmos secuenciales no pueden ser paralelizables al completo. En un algoritmo secuencial, podríamos distinguir las siguientes partes:

a) Parte en secuencial
b) Zona paralelizable utilizando *k* procesadores (*k*<*p*) de los *p* procesadores disponibles.
c) Parte paralelizable utilizando los *p* procesadores.

Para simplificar los cálculos, vamos a utilizar el modelo PRAM donde despreciaremos el coste de las comunicaciones y las contenciones.

El coste en paralelo se podría escribir en el modelo PRAM como:

$$T_p = a \cdot T_1 + b \cdot \frac{T_1}{k} + c \cdot \frac{T_1}{p}$$

sustituyendo sobre el *Speed-up* (T_1/T_p):

$$S_p = \frac{T_1}{a \cdot T_1 + b \cdot \frac{T_1}{k} + c \cdot \frac{T_1}{p}}$$

Siendo *a*, *b* y *c* el porcentaje correspondiente a cada parte (*a + b + c = 1*)

Si el algoritmo paralelo no tiene parte secuencial y toda la parte paralelizable lo es para *p* procesadores, entonces:

$$a = b = T_c = 0 \rightarrow S_p = p$$

Y si el algoritmo no tiene parte secuencial, y toda la parte paralelizable lo es para *k* procesadores siendo $k<p$:

$$a = c = T_c = 0 \rightarrow S_p = k < p$$

y si el algoritmo tiene parte secuencial y toda la parte paralelizable lo es para *p* procesadores:

$$b=Tc=0 \rightarrow S_p = \frac{T_1}{a \cdot T_1 + c \cdot \frac{T_1}{p}}$$

Como a + c = 1,

$$S_p = \frac{p}{(p-1)\cdot a + 1}$$

Por ejemplo, si un algoritmo es paralelizable en un 50% para *p* procesadores, entonces su *Speed-up* máximo será:

$$S_p = \frac{p}{(p-1)\cdot 0.5 + 1} < \frac{1}{0.5} = 2$$

En la figura 2 se aprecia claramente el impacto en las prestaciones que tiene que un algoritmo no sea 100% paralelizable.
Llama la atención el hecho de que si un algoritmo es ejecutado en paralelo en un 95% de su tiempo total de ejecución, el *Speed-up* máximo teórico que se puede alcanzar es de 20, o de 10 cuando se ejecute el 10% en secuencial.

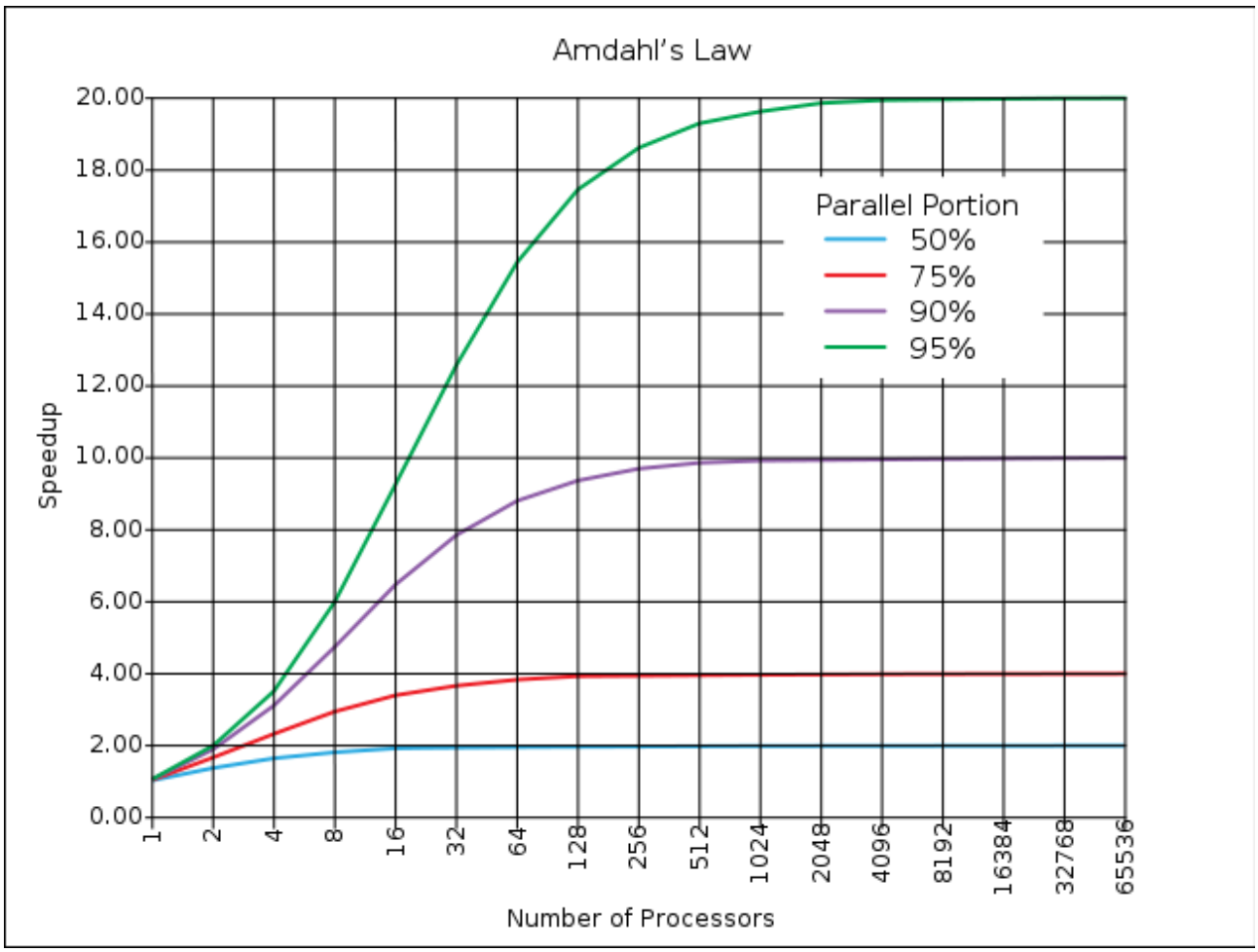

Figura 2: Speedup máximo teórico dependiendo de la cantidad de código que se ejecute en paralelo

3 Tecnología utilizada para la paralelización

Tras analizar los requisitos de la empresa y el proyecto, se tomó, de forma casi inmediata, la decisión sobre la tecnología que se debe de utilizar en FEST3D para dotarlo de concurrencia.

En primer lugar se trata de un software destinado a ser utilizado en ordenadores de sobremesa. En ningún caso se van a realizar cálculos sobre clusters u otros modelos de memoria distribuida. Por tanto el modelo debía ser de memoria compartida (PRAM).

En segundo lugar primaba la claridad del código y su futura mantenibilidad.

Para finalizar, se requería una tecnología compatible con los compiladores utilizados en FEST3D, es decir, GCC (GNU Compiler Collection) y que no fuera necesario usar otros ejecutables/compiladores distintos como podría ser el caso de MPI (*Message passing interface*).

Por todo ello, por su simplicidad, facilidad de uso y comprensión se decidió utilizar OpenMP. Al ser solo una especificación, permite que el mismo código pueda ser compilado con distintas implementaciones realizadas por un fabricante de compiladores distinto (GNU, Microsoft, Intel, Sun...).

A continuación se describe tanto OpenMP (PRAM), que no puede actuar en memoria distribuida, como MPI, que puede actuar tanto en memoria distribuida como en compartida aunque.

3.1 OpenMP

OpenMP es un API que permite la programación concurrente sobre memoria compartida. Al ser una especificación, cada fabricante define su librería OpenMP ciñéndose a la API. Por tanto, OpenMP, es multiplataforma, y está disponible para C/C++ y Fortran.

La programación en OpenMP se realiza utilizando ciertas directivas de compilador y funciones de la propia librería. Está basado en el modelo de ejecución *fork-join*, en el que en un determinado punto de un programa se crean varias copias (hijos) de un proceso padre y que pueden realizar trabajos distintos entre sí.

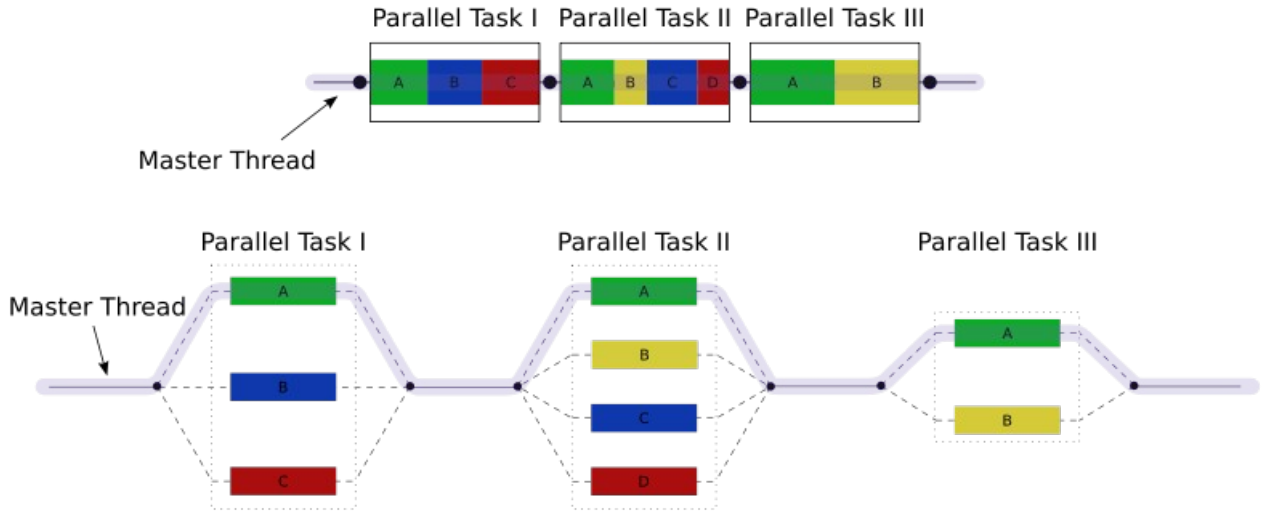

Figura 3: Creación de hilos en OpenMP

Para esto, nos proporciona una interfaz simple y de muy alto nivel, en el que se pueden encontrar funciones y directivas de sincronización, reparto de trabajo, información, etc...

A continuación se describen las directivas y funciones utilizadas:

3.1.1 Parallel

Crea una nueva zona paralela en la que se van a activar el número de threads que marque la variable de entorno OMP_NUM_THREADS, y crea un grupo de threads. En este proyecto, *parallel* siempre se combinada con la directiva *for*, que permite repartir iteraciones de un bucle *for* para cada thread.

(C++)
```
#pragma omp parallel
#pragma omp for
     for(int i=0;...
```

Las iteraciones en el bucle en OpenMP se pueden repartir incluyendo “schedule(<planificación>)” en la directiva. Como más adelante se va a justificar la elección de cada tipo de reparto, se incluye una gráfica que explica visualmente cómo se comporta cada planificación.

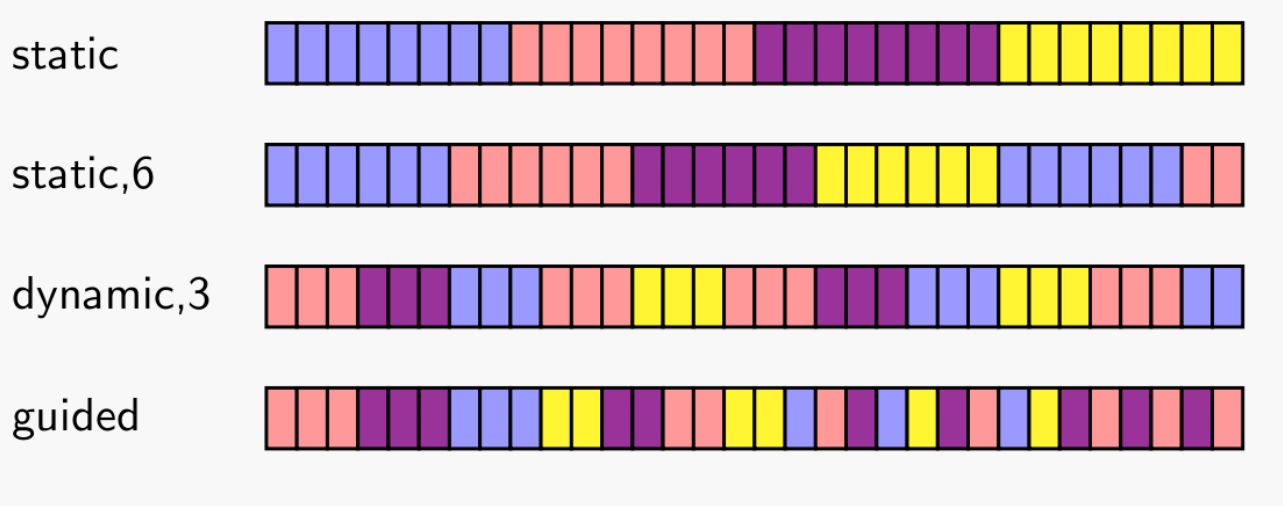

Figura 4: Distintos tipos de planificación de reparto de trabajo

En esta imagen se ve la diferencia de reparto de trabajo entre distintos threads con los tipos de planificaciones existentes. Cada color representa un thread y cada celda una iteración de un determinado bucle.
En el caso de la planificación *static* y *dynamic*, se le puede añadir un *chunk*, que es el número que marca la cantidad de iteraciones que se le da a cada thread.

- La planificación *static*, por defecto, reparte las iteraciones de manera más equitativa posible y se utilizaría en el caso de que cada iteración de un bucle tenga una complejidad parecida. Si se especifica el *chunk*, a cada thread se le va a asignar ese mismo número de iteraciones, volviendo a empezar por el primer thread si la división no es exacta. El reparto de iteraciones se realiza antes de empezar el bucle, y es fijo.
- La planificación *dynamic* es el caso contrario. Se trata de un reparto totalmente dinámico en el que cuando un thread esté libre, se le asigna una iteración (o un número de iteraciones igual al *chunk*). Esto añade cierto *overhead* al obligar a OpenMP a añadir cierta lógica de reparto en todo momento, cosa que no pasa en *static*, por lo que podría afectar a las prestaciones dependiendo del grano de paralelismo utilizado.
- La planificación *Guided, en cambio,* es una mezcla de ambos: se comporta como *dynamic*, pero su *chunk* va decreciendo a lo largo de la ejecución.

3.1.2 Critical

Como en la mayoría de los programas, en FEST3D existen secciones de código que no pueden ser realizados en paralelo, como puede ser una lectura de fichero, el uso de ciertas herramientas externas que no son *thread-safe*, etc...

La directiva *critical* evita las condiciones de carrera, permitiéndonos crear un ámbito en el que solo pueda estar un thread a la vez. Los demás threads, que intenten entrar en esa zona crítica, detendrán su ejecución en ese mismo punto y esperarán a que finalice el bloqueo.
Esta directiva también es muy útil a la hora de realizar una depuración en paralelo.

(C++)
```
#pragma omp critical
```

3.1.3 Task

La directiva *task*, y todo lo que englobe su ámbito, crea literalmente una tarea (un trozo de código almacenado para ser ejecutado en el futuro) que podrá ser llevada a cabo por cualquier thread, sin garantizar que su inicio sea en un momento determinado. Resulta útil cuando hay unas pocas operaciones independientes entre sí fuera de un bucle.

```
#pragma omp task
```

3.1.4 Threadprivate

En una aplicación no diseñada pensando en la concurrencia, es habitual el uso de variables globales o estáticas no constantes, que pueden provocar un mal funcionamiento del programa al existir condiciones de carrera. Ejemplos de esto son las variables declaradas *common* o módulos en Fortran, y *static* en C/C++.
Esta directiva aplicada sobre una determinada variable global hace que dicha variable sea global solamente a nivel de thread. Esta técnica se conoce como *TLS* (Thread local storage).
Este método en el momento de realización del proyecto era relativamente nuevo, y aún existían problemas con algunos compiladores que no lo implementaban correctamente.

(C++)
```
#pragma omp threadprivate(var1)
```

3.2 MPI (Message Passing Interface)

MPI es una especificación para una API que permita a un computador comunicarse con otro mediante el paso de mensajes. Se utiliza en clusters y supercomputadores. En él, mediante una serie de funciones cuyo funcionamiento interno depende de la implementación, se permite la comunicación punto a punto o broadcast, sincronizaciones, etc.
A diferencia de OpenMP, MPI utiliza el modelo DRAM para memoria distribuida.
Cuando se inicia el paralelismo, se crean varios procesos iguales al máster (proceso 0), cada uno con su número identificador.

A partir de ese momento, cada procesador irá recibiendo o enviando los datos necesarios y realizando unas tareas u otras dependiendo de su identificador. A priori, el flujo de ejecución es el mismo, a no ser que se hagan distinciones por identificador. Toda comunicación se realiza mediante sockets, aunque MPI abstrae al programador de su uso.
Al poder escoger la información a transferir, es mucho más flexible que OpenMP tanto en reparto de trabajo como en el control de cada procesador.

4 Descripción de FEST3D

4.1 Introducción

El objetivo del proyecto FEST3D es desarrollar algoritmos electromagnéticos para el diseño de dispositivos pasivos de microondas y de ondas milimétricas en tecnología de guía de onda para comunicaciones espaciales y terrestres. Dichos algoritmos se basan en técnicas numéricas modales avanzadas, lo que permiten, una vez se hayan implementado en una herramienta informática, realizar diseños de forma rápida y precisa, analizar filtros, acopladores, etc… en tiempos extremadamente cortos y con una gran precisión, lo que permite abordar estructuras muy grandes y complejas, siendo posible sintetizar distintos tipos de componentes.

FEST3D, como se puede observar en la figura 5, posee dos partes fundamentales, claramente diferenciadas:

- Entorno gráfico (denominado también GUI, del inglés Graphical User Interface)

  Es la parte que el usuario visualiza en pantalla y ofrece facilidades al usuario para que el empleo de FEST3D sea lo más intuitivo posible.

- Motor computacional (denominado también EMCE, del inglés ElectroMagnetic Computational Engine)

  Es el encargado de realizar las simulaciones, síntesis y cualquier cálculo pesado para producir el diseño final del circuito pasivo de microondas solicitado por el usuario.

  Asimismo se debe explicar que el motor computacional se subdivide en varias partes claramente diferenciadas dependiendo de su funcionalidad:

  - Análisis electromagnético: Esta funcionalidad es la encargada de realizar el análisis de circuitos que el usuario ha introducido a través del interfaz gráfico. Una vez el usuario ha acabado de definir su circuito, puede iniciar el análisis que le dará la respuesta electromagnética del circuito.

  - Optimizador: Una vez analizado el circuito, es probable que deseemos realizar optimizaciones para mejorar su respuesta. El proceso de optimización es un proceso iterativo cuya finalidad es hacer que la respuesta del circuito se asemeje lo más posible a una respuesta ideal que introduce el usuario. Al optimizador se le

introducen dos tipos de datos, por un lado, la respuesta que nos gustaría que alcanzara nuestro circuito, y por otro lado los parámetros del circuito que se le permiten cambiar para cumplir dichos objetivos. Una vez fijados estos parámetros, el optimizador realizará cambios de manera iterativa de las dimensiones del circuito hasta alcanzar la respuesta óptima. Como se ha mencionado, el optimizador realiza cambios en las dimensiones del circuito previamente simulado, ya que realizando un ajuste fino de éste puede llegar al diseño final sin partir de una solución previa bastante aproximada a la que se busca.

- Síntesis: Se encarga de la síntesis de filtros, adaptadores y diplexores. El usuario introduce la respuesta esperada del circuito y se inicia un proceso de síntesis que acaba produciendo un circuito que cumple dichas especificaciones. El circuito generado queda perfectamente definido, ya que su salida aporta todas las dimensiones de los elementos, por lo que en este momento se estaría en disposición de poder fabricar el dispositivo.

- Análisis de efectos de alta potencia: Este módulo realiza el análisis de efectos indeseados de alta potencia que puedan degradar o destruir la respuesta del dispositivo, en concreto determina la potencia umbral de ruptura RF de efectos Multipactor o Corona del circuito definido por el usuario.

Como ya se ha comentado, FEST3D tiene como objetivo el análisis y diseño de estructuras complejas optimizando los recursos computacionales empleados. Es decir, se tiene la necesidad de poder realizar simulaciones en estaciones de trabajo como PCs bajo sistemas operativos Windows y Linux sin necesidad de recurrir a clusters de ordenadores.

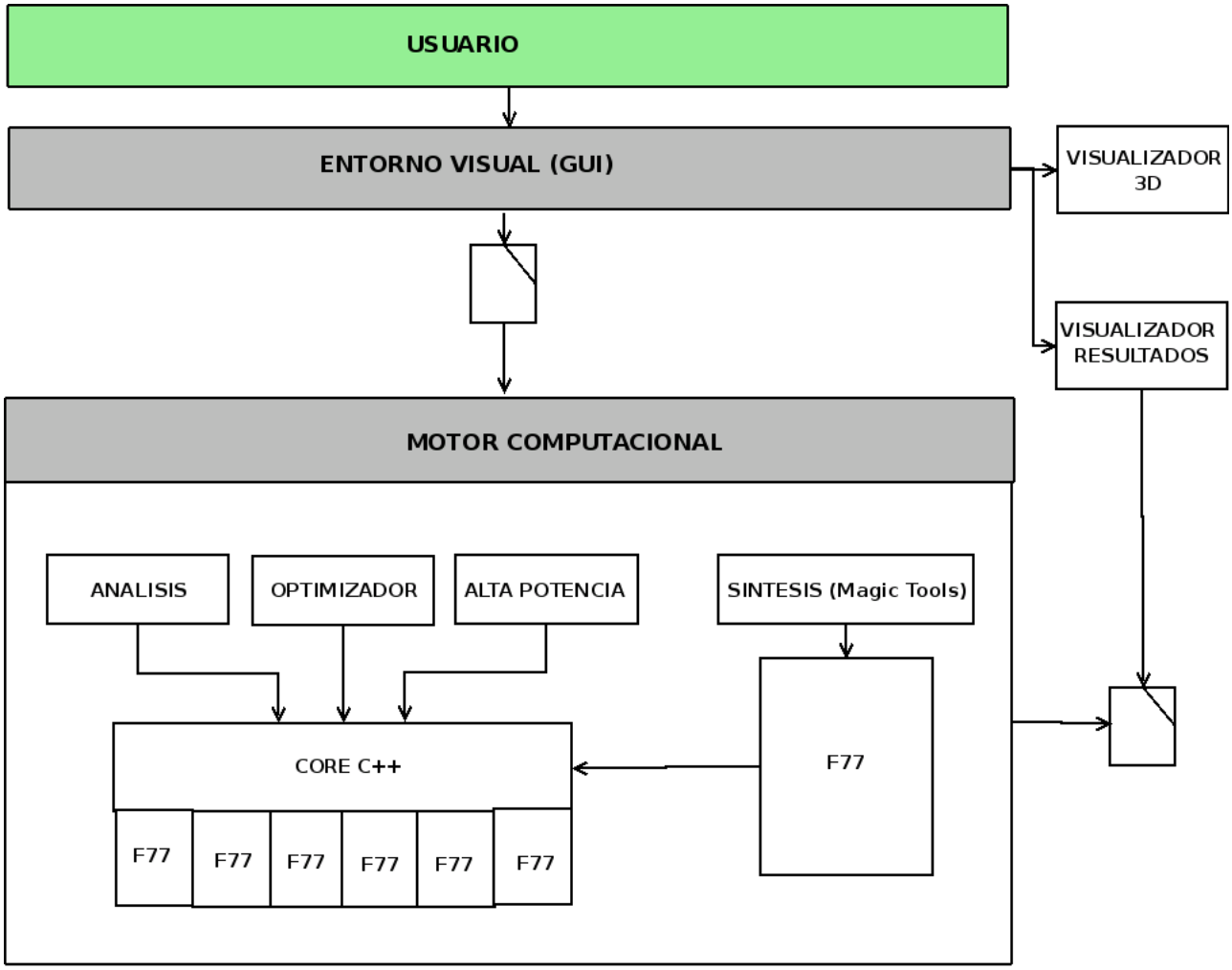

Figura 5: Esquema de los componentes de FEST3D

Tecnologías

Como ya se ha explicado, el motor computacional es el encargado de realizar las simulaciones que solicita el usuario a través del interfaz gráfico. Puesto que las simulaciones y diseños por parte de las empresas son sumamente complejos y específicos, esto implica que el motor computacional debe estar preparado para realizar gran cantidad de cálculos pesados con la mayor precisión posible.

Por ello se usa una programación mixta de Fortran77/ C/C++. FEST3D lleva en desarrollo un gran número de años. En sus comienzos, el lenguaje de programación Fortran era uno de los lenguajes de cálculo por excelencia debido a la velocidad con la que realiza dichos cálculos. Sin embargo, Fortran no posee una de las características básicas para realizar proyectos de gran envergadura: la orientación a objetos. Debido a las similitudes de cálculo entre diferentes tipos de estructuras soportadas por FEST3D, sería ilógico pensar en repetir las rutinas de cálculo para cada una de estas

estructuras. Esto sería una mala idea ya que generaría un código poco mantenible en el futuro, puesto que si se quiere cambiar el comportamiento de simulación de todas las estructuras que son parecidas, deberíamos realizar los cambios elemento a elemento.

Por esta razón se utiliza un segundo lenguaje de programación en el motor computacional: C++. C++ es un lenguaje orientado a objetos, lo que permite crear estructuras genéricas, de las cuales se podrán crear estructuras que hereden todas sus propiedades. Así pues, realizar un cambio como el propuesto anteriormente, no implicaría cambiar todas las estructuras, sino, solamente la estructura genérica, y por herencia el resto de 'estructuras hijas' heredaría estos cambios. Este tipo de programación, además, posibilita la integración de nuevos módulos informáticos de manera sencilla, de modo que el software puede ser mejorado y actualizado constantemente. FEST3D utiliza estos lenguajes de programación de manera conjunta, explotando al máximo las capacidades que ofrecen ambos, de manera que su software sea preciso, rápido y flexible.

4.2 Funcionamiento de FEST3D

Un circuito en FEST3D está formado por dos tipos de elementos: Guías y discontinuidades. Cada uno incluye variantes dependientes de la forma y/o comportamiento del elemento, por lo cual se resuelven de manera distinta.

- Guías : El principal requerimiento de un circuito de microondas es la necesidad de transferir señales de un punto a otro sin pérdida de radiación. Esto requiere el transporte de energía en forma de onda.
  Una guía es una estructura metálica, con forma de tubería y desarrollada para tal menester; se trata del elemento principal del que dispone FEST3D para que el usuario genere sus circuitos. Se pueden observar guías básicas como la rectangular o circular, o guías más complejas como la elíptica, guía Ridge, guías curvadas o codos, coaxiales etc... Estas y otros tipos se explicarán brevemente más adelante.

- Discontinuidades: Las discontinuidades son un elemento que unen dos o más guías y que modelan las uniones entre ellas. Ejemplos de discontinuidades son N-step, T-Junction, Cubic-junction, entre otras.

Cuando se realiza un análisis electromagnético de un circuito de microondas en FEST3D, el objetivo principal es obtener la respuesta eléctrica que caracteriza dicho circuito en el rango de frecuencia de trabajo del dispositivo bajo análisis.

Para ello, se distinguen dos partes fundamentales:

- Parte estática: Es la sección del programa en la que se realizan los cálculos que dependen tan solo de la geometría y son independientes de la frecuencia.

La versión sin paralelizar de FEST3D en la parte estática recorría una lista de guías y posteriormente discontinuidades de manera secuencial, computándolas y realizando su resolución una sola vez.

- Parte dinámica: Es la encargada de analizar los componentes pero esta vez realizando los cálculos necesarios que dependen de la frecuencia.

  Aquí, para cada determinado punto en un rango de frecuencias, en la versión secuencial de FEST3D se volvía a recorrer la lista con elementos y se computan. Cada elemento, al final, tendrá una matriz que representa su comportamiento electromagnético para una determinada frecuencia, y dicha matriz se ensambla con las del resto de elementos, para terminar resolviendo un sistema de ecuaciones.

4.3 Descripción del funcionamiento del lanzamiento de elementos

El cálculo de la respuesta electromagnética representa la base que necesitan todos los distintos análisis de las diversas funcionalidades que ofrece FEST3D. Por ello, el objetivo principal es paralelizarlo. En esta sección se describe brevemente el funcionamiento previo de FEST3D antes de dotarlo de concurrencia.

Vamos a distinguir entre las dos partes de código principales paralelizadas, la parte estática y la dinámica.

4.3.1 Parte estática

En el código original se computaba secuencialmente cada elemento uno tras otro. En principio, el cómputo de cada elemento es independiente de otro en el sentido de que, mientras se realiza el cálculo de uno, no se necesita calcular nada del otro, aunque sí que hay ciertas dependencias en el orden del cálculo.
Como ya se ha expuesto, una discontinuidad une dos guías o más. Por dicho motivo, existe una dependencia entre una discontinuidad y sus guías asociadas. Una discontinuidad tiene que computarse una vez que todas las guías a las cuales conecta ya han sido calculadas. En la versión secuencial, esto se hacía simplemente ordenando una lista con referencias a elementos de forma que primero se computaban las guías y después las discontinuidades.

Para cada elemento de la lista, el código llama a un método smart_initialize() que reservará la memoria necesaria primero y después realizará todo el cálculo completo distinguiendo el tipo de elemento que se trata.

El código siguiente es una simplificación del código original de lanzamiento de elementos, que es donde realmente recae todo el peso de la parte estática.

```
for (i = 0; i < elem().size(); i++) {

     Elem & ei = * elem()[i];

     actually_cache = ei.smart_initialize(find(ei, i), FS);


     FS.write(i, * this, actually_cache);
}
```

Para cada elemento de un vector de Elem (que contiene primero guías y a continuación las discontinuidades), el bucle llama a su método smart_initialize() que realizará toda la acción.

4.3.2 Parte dinámica

En esta parte se obtiene la respuesta de los elementos para cada punto en frecuencia posible dentro de un rango (el número de puntos en frecuencia es un *input* del programa). En primer lugar se tiene que realizar un determinado cómputo de cada elemento, y en segundo, como ya se ha comentado, construir una matriz con un formato determinado, y finalmente resolver un sistema de ecuaciones con ella.

En el primer paso esta vez no hay ninguna dependencia entre elementos excepto dos elementos en los que su parte dinámica se realiza de manera especial. El orden del cómputo de elementos se realizaba de la misma manera que en la parte estática.

4.3.3 Network y caché

Una simulación de FEST3D puede consumir desde pocos segundos hasta minutos u horas. Por esa razón, la optimización y rendimiento puede llegar a ser clave. Es por eso que mejorar el rendimiento de algún modo es de vital importancia de cara al usuario final.

Por esta razón, una de las características de FEST3D que más ayuda a mejorar la eficiencia es la implementación de dos sistemas de reutilización de cálculos, que se han llamado "network" y "caché". En síntesis, ambos se basan en aprovechar los cómputos realizados anteriormente en otras guías y discontinuidades.

Los distintos métodos que usa FEST3D para solucionar sus elementos son:

4.3.3.1 Network

Partimos de la lista de elementos anteriormente mencionada, que se recorre y calcula elemento a elemento con la excepción de que un elemento exactamente igual haya sido computado previamente. Esto es bastante frecuente en los circuitos, ya que se usan muchos componentes iguales (circuitos simétricos, etc...). En el momento que se va a calcular un elemento, se comprueba si es igual a otro ya realizado, y en caso afirmativo se copian sus resultados.

4.3.3.2 Caché

También es frecuente que se lleve a cabo una simulación y una vez terminada, se modifique algún elemento y se vuelva a simular casi lo mismo. El método de caché consiste en almacenar en memoria secundaria los resultados calculados de cada elemento con el fin de usarlos después. De la misma manera que la network, cada elemento comprueba que no haya uno igual en caché (que es cargada al principio de toda la simulación) y de ser así, lee sus datos y los copia.
Cuando un elemento ha sido calculado, se escribe la información necesaria a fichero.

4.3.3.3 Scratch

Un elemento que no esté calculado previamente en la network o en la caché, se tiene que calcular desde cero.

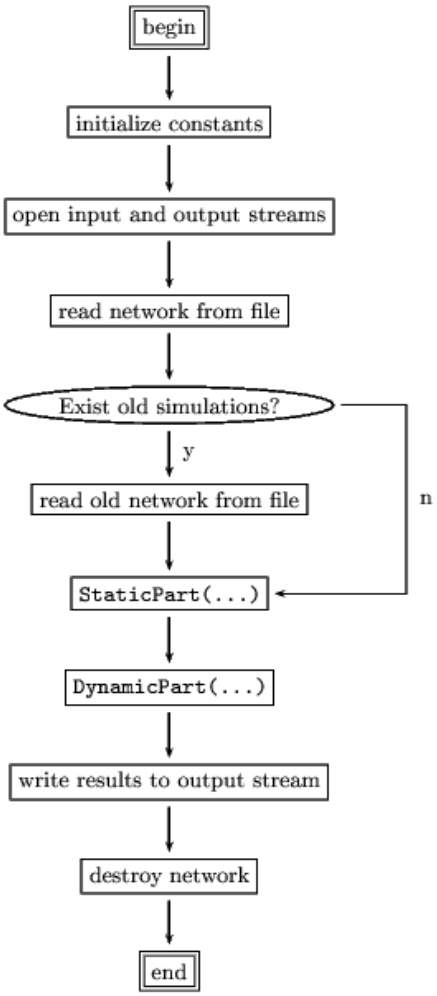

Figura 6: Flujo de funcionamiento básico en un análisis de FEST3D

En la figura 6 se esquematiza el flujo básico de FEST3D, empezando por la apertura e inicialización de flujos y constantes, lectura de ficheros de entrada, búsqueda de simulaciones anteriores en memoria secundaria, y en caso de que no existan, la parte estática.
Para terminar, la parte dinámica (que se ejecuta independientemente de que haya simulaciones previas o no), y al final, la escritura de resultados.

4.3.4 Resumen de dependencias entre elementos

En las siguientes tablas que se presentan se resumen las dependencias que existen entre elementos en la parte estática y dinámica dependiendo del tipo que sea: si se va a obtener de otro elemento

igual previamente calculado (*network*), si se va a computar desde el principio (from *scratch*) o con qué se conecte cada elemento.

Dependencias (parte estática)	Elemento “from scratch”	Elemento de “network”
Guía	-	Otra guía igual
Discontinuidad	Guías a las que conecta	Otra disc. Igual

Tanto la guía *curved* como *radiating array* tienen un comportamiento ligeramente distinto en la parte dinámica. Esto se debe a que necesitan calcular sus modos para cada punto de frecuencia (el resto de elementos lo hacen solamente una vez en la parte estática). Por esta razón, en la parte dinámica, una discontinuidad conectada a este tipo de guías debe procesarse después de que estos elementos hayan computado sus modos. Es la única dependencia que existe en esta sección.

Dependencias (parte dinámica)	Conectado con Phased array o curved	Otros casos
N-step	Phased array / Curved	-

Cada dependencia marca que ese elemento ha debido de ser computado al completo antes que el otro.

4.4 Descripción de los tipos de elementos

En esta sección se muestran los distintos elementos que componen FEST3D y sobre los que se realizan los cálculos que finalmente proporcionan el resultado final.

4.4.1 Guías

Una guía es una estructura metálica, con forma de tubería y desarrollada para transferir señales de un punto a otro sin pérdida de radiación. Es el elemento principal del que dispone FEST3D para que el usuario genere sus circuitos.
En la simulación, se necesita calcular los modos electromagnéticos que se propagan en el interior de la guía. Los modos son soluciones elementales del problema que satisfacen las ecuaciones de maxwell. De esta manera, el campo electromagnético real se puede expresar como una expansión en términos de estas soluciones elementales. Además, estos dependen de la geometría de la sección transversal de la guía y no de la frecuencia.

4.4.1.1 Rectangular

Es una guía uniforme, con una sección transversal rectangular.

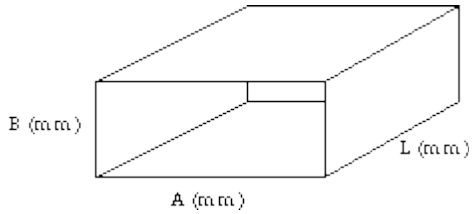

Figura 7: Guía rectangular

4.4.1.2 Circular

Guía uniforme con una sección transversal circular.

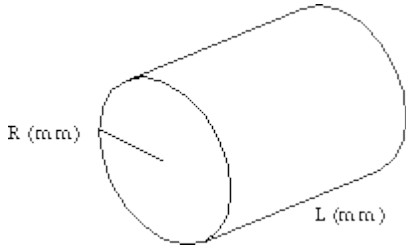

Figura 8: Guía circular

4.4.1.3 Guías basadas en la guía rectangular arbitraria

4.4.1.3.1 Rectangular arbitraria

La guía rectangular arbitraria es una guía que tiene una sección transversal delimitada por una serie de formas rectas, circulares y elípticas, y que está contenida por una guía rectangular ficticia (guía de referencia).
Esta guía sirve de base para otras que particularizan el caso de guía rectangular arbitraria. Es uno de los elementos con mayor complejidad algorítmica.

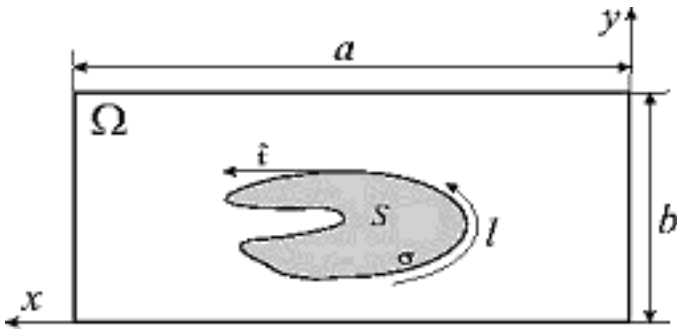

Figura 9: Guía rectangular arbitraria

Ejemplos de guías de este tipo son las siguientes:

4.4.1.3.2 Coaxial

Es una guía uniforme con un conductor interno y uno externo que puede ser circular o rectangular.

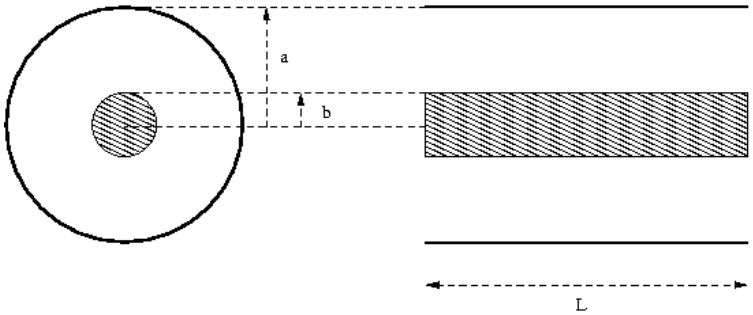

Figura 10: Guía coaxial

4.4.1.3.3 Cross WG

Guía uniforme con dos salientes de un determinado ancho, los cuales pueden ser redondeados.

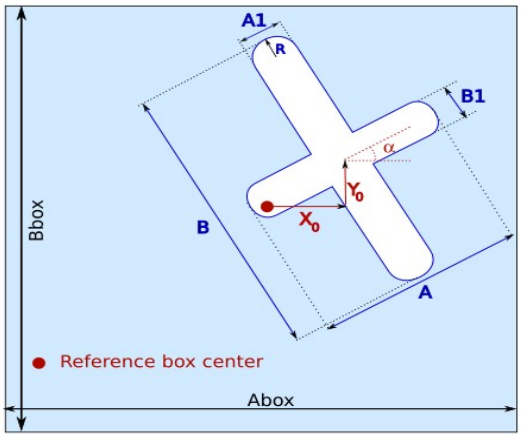

Figura 11: Guía cross

4.4.1.3.4 Ridge

Guía uniforme con uno o dos entrantes en la parte superior e inferior.

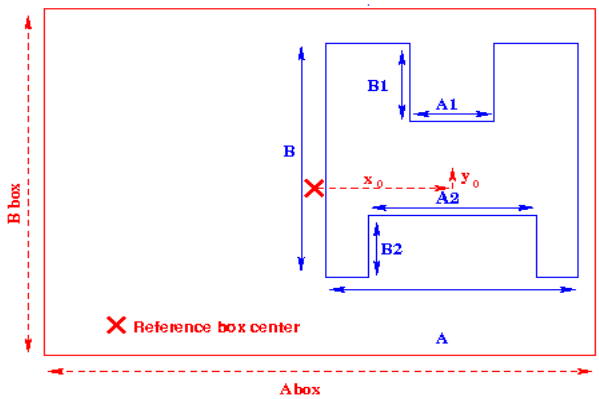

Figura 12: Guía ridge

4.4.1.3.5 Slot

Es una guía rectangular con las esquinas redondeadas.

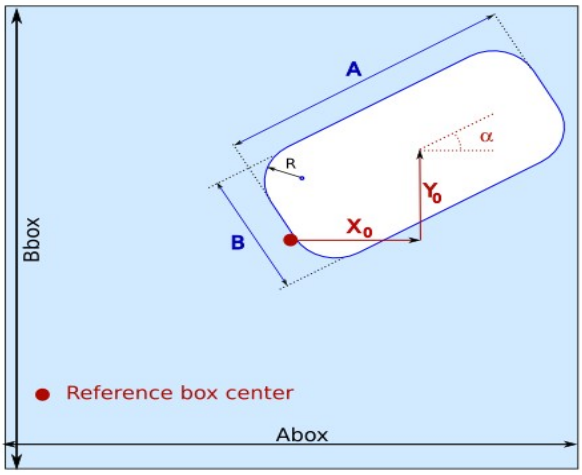

Figura 13: Guía slot

4.4.1.3.6 Waffle

Es una guía uniforme, rectangular, que tiene varias inserciones.

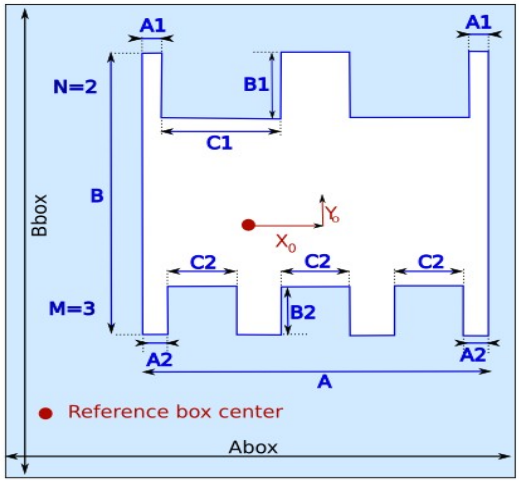

Figura 14: Guía waffle

4.4.1.3.7 Circular arbitraria

La guía circular arbitraria es una guía que al igual que la rectangular arbitraria, tiene una sección transversal delimitada por una serie de segmentos rectilíneos y formas circulares y elípticas, pero en su caso está contenida por una guía circular ficticia (guía de referencia). Cualquier guía circular arbitraria podría estar modelada con una rectangular arbitraria, pero en ocasiones interesa que la guía de referencia sea una circular.

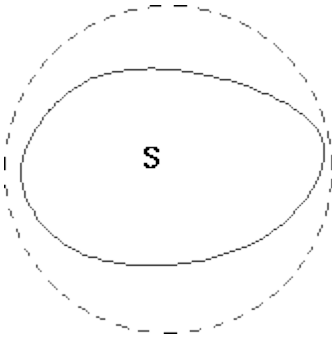

Figura 15: Guía circular arbitraria

4.4.1.4 Otras

4.4.1.4.1 Radiating Array

Es una guía que simula el comportamiento de un array infinito de guías que están abiertas hacia el exterior, es decir, no están conectadas con otras.

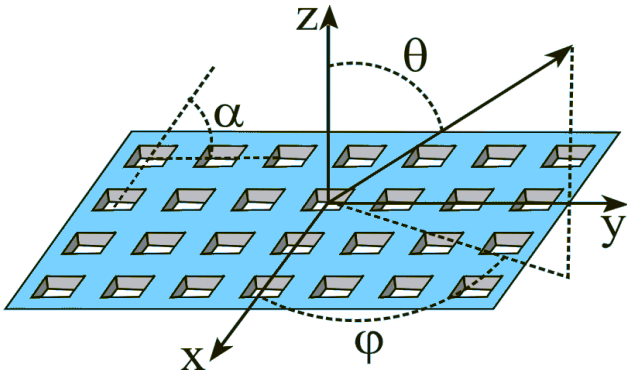

Figura 16: Guía radiating array

4.4.1.4.2 Curved

Es una guía no uniforme con una sección rectangular y curvada hacia la izquierda o hacia la derecha.

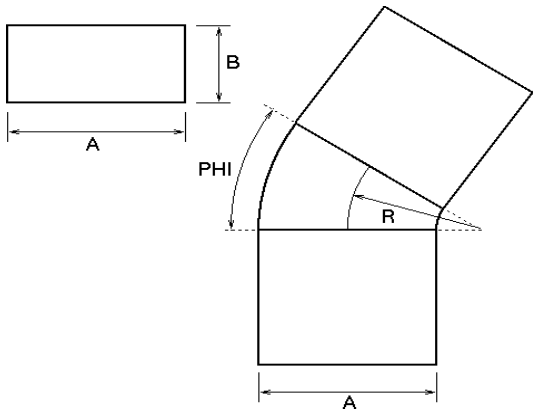

Figura 17: Guía curved

4.4.2 Discontinuidades

4.4.2.1 N-step y step

Representa la unión planar de dos o más guías. En este elemento, entre otros cálculos se necesitan calcular las integrales de acoplo entre las distintas guías en la parte estática. Las integrales de acoplo proporcionan información sobre cómo se acoplan los modos de las guías entra sí. Este cálculo depende del tipo de guía y de su geometría.
El step es una particularización del *N-step* en el que únicamente se unen dos guías.

4.4.2.2 Rectangular cavity with screws

Discontinuidad rectangular con postes cilíndricos internos, la cual se puede conectar por cualquiera de sus caras a una guía rectangular.

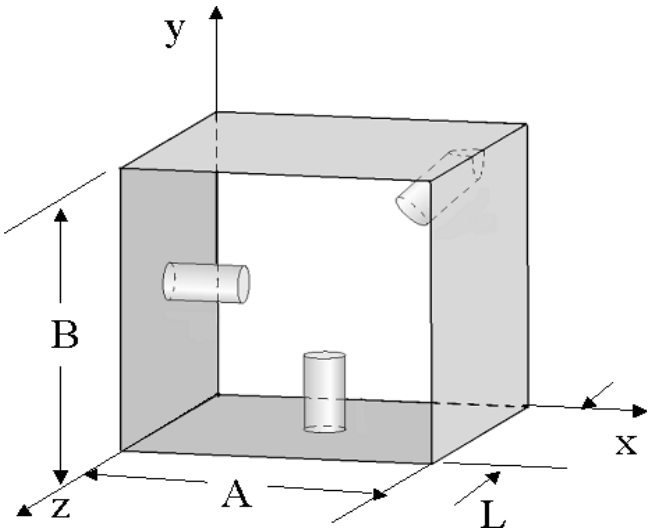

Figura 18: Guía rectangular cavity with screws

4.4.2.3 General 3D cavity

Es una cavidad rectangular que permite introducir un obstáculo metálico en su interior de forma arbitraria.

4.4.2.4 Arbitrary 2D cavity

Es una cavidad arbitraria en 2D. Tiene una forma arbitraria en planta y una altura o anchura constante.

5 Propuesta de paralelización

5.1 Parte estática

Como ya se ha mencionado, en la parte estática todo el peso recae sobre un bucle en el que se van lanzando elementos, entre los que existen dependencias en algunos casos.

El primer paso de la paralelización era lanzar estos elementos en paralelo, uno por thread. El objetivo era crear, con posterioridad, una o más capas adicionales por debajo, cubriendo otros cómputos pesados que ocurren en la resolución propia de ciertos elementos.

Dada la facilidad y el alto nivel de abstracción que nos ofrece OpenMP, lo primero que se hizo fue crear una zona paralela con la directiva *parallel for*, con el fin de repartir el número de iteraciones del bucle mediante la directiva:

```
#pragma omp parallel for schedule(dynamic)  //Define parallel zone
     for (int i = 0; i < n; i++)
          …
```

Esto hacía que cada núcleo computacional se encargara de un elemento a la vez. Como existe una gran variedad de elementos, cada uno conteniendo propiedades dispares y métodos de resolución distintos, los tiempos de cálculo de cada uno pueden variar con varios ordenes de magnitud.

Por esta razón se utiliza la directiva *dynamic*,1 (o *dynamic* simplemente), que marca el reparto dinámico de las iteraciones en tiempo de ejecución según vaya habiendo threads libres. Esto tendría el inconveniente de una mayor sobrecarga si se tratara de un paralelismo de grano muy fino, pero una estructura típica de FEST3D suele contener un número de elementos del orden de decenas o cientos y además con la suficiente complejidad computacional.

Un reparto de iteraciones previo (*static* y en cierto modo *guided*) podría derivar en situaciones muy poco óptimas por el reparto previo que realiza debido a que se produzca un posible desequilibrio de carga.

El proceso de lanzamiento de los elementos ha ido sufriendo un proceso de mejora continuo para tratar de lidiar con los problemas que surgen por el hecho de paralelizar:

- Dependencias antes comentadas entre las discontinuidades y las guías a las que están asociadas.

- Dependencias entre elementos que no se van a calcular, porque se van a copiar de otro ya calculado (de *network*). No deben lanzarse dos elementos iguales a la vez, ya que no es

necesario.

- Escritura de caché simultánea.

Estos problemas surgen directamente al lanzar elementos en paralelo. Posteriormente en otra sección se explicarán los distintos contratiempos que fueron surgiendo a nivel más bajo en cada tipo de elemento por el modo en el que estaba programado FEST3D.

-Primera aproximación de lanzamiento paralelo de elementos:
En el algoritmo que se ideó en un primer momento se creaban varias listas de elementos en lugar de la lista original:

1. Guías que han de computarse (*scratch*)
2. Guías que se copian de otras ya hechas (de *network*)
3. Discontinuidades que se computan
4. Discontinuidades copiadas de otras

De esta manera, cada lista recorrida por un bucle paralelo distinto. Esto evitaba todos los problemas anteriormente citados (a excepción de la escritura en caché que se decidió realizar en secuencial al final), pero era una organización no demasiado óptima, ya que frecuentemente quedaban threads libres en el primer bucle, mientras había guías que se podían resolver en el segundo o en el tercero, al estar alguna guía sacada "de *network*" o discontinuidad disponibles para ser computadas por tener sus dependencias cumplidas.

-Algoritmo definitivo de lanzamiento paralelo de elementos:
Por tanto, se decidió crear un método repartidor de tareas, constituyendo cada tarea un elemento a resolver y unificando los cuatro bucles en uno. Dicho método tendrá que escoger un elemento disponible manteniendo las condiciones de dependencias cumplidas y devolver una referencia a dicho elemento al thread que lo está pidiendo.

Entonces, cada thread por cada iteración "pedirá" una nueva tarea de forma controlada y lo resolverá. El aspecto que presenta ahora el bucle de elementos es el siguiente:

```
 //Define parallel zone
#pragma omp parallel for schedule(dynamic)
 for (int i = 0; i < elem().size(); i++) {

      Elem* ei;

      #pragma omp critical (selectionstatic)
      ei = i_selectionStatic(); //Repartidor de tareas
      …
      ei->smart_initialize(network, FS); //Resolución
      }
 }
```

Lógicamente se necesita un *lock* (directiva *critical*) que controle que solamente un thread acceda al repartidor de tareas a la vez.

En este momento cualquier elemento bien guías o bien discontinuidades, sean de *scratch*, *network* o *caché*, pueden estar siendo lanzados a la vez, siempre que se cumplan sus dependencias propias.

5.1.1 Repartidor de tareas

Se encarga de que las dependencias anteriormente expuestas se cumplan en todo momento. Va a ser llamado tantas veces como elementos haya en el circuito, y además devolverá el elemento disponible de mayor prioridad, o peso.

5.1.2 Sistema de prioridades

El hecho de que exista tal variedad en los tiempos de cálculo entre cada tipo de guía o discontinuidad, hace que sea útil cierto orden en la finalización de los elementos, aprovechando el repartidor de tareas.

En este caso, el objetivo es que un elemento muy pesado del circuito, cuyo tiempo de cómputo pueda representar un elevado porcentaje del tiempo total, no sea el último en ser lanzado, sino el primero. Así se minimiza el tiempo en el que hay threads libres, y por tanto se optimizan tiempos.

La prioridad de un determinado elemento es un nuevo atributo añadido a la clase *Elem*, de la cual heredan todas las guías y discontinuidades y se decide al principio de la simulación, teniendo en cuenta el tipo de elemento de que se trata y algunos de sus atributos. Esto no es más que una mera aproximación basada en la experiencia, puesto que estimar un peso exacto de un elemento para que se pueda comparar con otros es imposible.

5.2 Parte dinámica

En la parte dinámica se actuó de una manera muy parecida a en la parte estática. La única diferencia es que el bucle de lanzamiento de elementos, se repite una vez para cada punto en frecuencia que se vaya a analizar (parámetro de entrada del programa). La parte dinámica tiene su propio repartidor de tareas y unas prioridades distintas, pero su funcionamiento es igual (ver cuadros de dependencias, sección 4.3.4).

5.3 Adaptación del código para la paralelización

En esta sección se describen el tipo de problemas que han surgido a lo largo de la adaptación del código para ser ejecutado en paralelo. FEST3D es una herramienta que reúne algoritmos pensados en un época donde el paralelismo MIMD no tenía gran protagonismo, y en consecuencia ciertas prácticas utilizadas debían ser modificadas.

- En primer lugar, FEST3D incluía una gran cantidad de variables estáticas (C/C++) y common (Fortran) no constantes. De esta forma se almacenaban datos fuera de un objeto, o se ampliaba su ámbito, haciendo la programación más cómoda.
  Al dotar el código de concurrencia, estos datos podían ser accedidos para escritura por varios threads. Por tanto, fue necesario utilizar la directiva *threadprivate* con la finalidad de almacenar una copia de los datos en cada thread.
  En algunos casos en los que el código no era demasiado extenso se fue cambiando el código para que en cada subrutina se pasaran las variables comunes por parámetro en lugar de hacer uso de variables globales.

- Por otro lado, otro de los problemas era el uso en ocasiones de una herramientas externas en FEST3D (*triangulate*, *getfem*, *gmesh*...) que no son *thread-safe*, y por lo tanto si son utilizadas simultáneamente pueden dar lugar a fallos de ejecución o precisión. Estos problemas se solucionaron sincronizando los accesos a los mismos de manera que únicamente un thread accediera a ellos. En el futuro, es de esperar que estas herramientas tengan soporte para multithread.

- A pesar de que en la mayoría de los casos un elemento no modificaba otro dentro de su cálculo, en el *N-step* se cambiaban temporalmente algunos parámetros de sus guías asociadas. Como una guía puede estar asociada a varias discontinuidades, si ambas discontinuidades se procesaban a la vez, podían producirse errores de precisión detectables en el resultado final ya que las discontinuidades podían cambiar algún parámetro de la misma guía.
  Para solucionarlo, se ha realizado una reestructuración del código, almacenando dicha información temporal sobre el propio *N-step* y cambiando el comportamiento sobre todo el código del programa en el que apareciera este elemento.
  Este elemento era de los que más variables estáticas tiene, sobre todo en la parte del código donde se realizan cálculos independientes de la frecuencia (parte estática), y se tienen en cuenta el tipo y forma de las guías conectadas (integrales de acoplo).
  Dentro de este mismo proceso, existía un problema similar en las integrales de acoplo entre la guía *rectangular* y la *curved*, en la que se necesitó usar *locks* ya que se trataba de un proceso de muy baja complejidad y solventarlo requería un gran y profundo cambio en el código.

- Por último, resaltar también la discontinuidad *Arbitrary 2D cavity*. Este elemento presentaba problemas al computarse varias concurrentemente, ya que había una gran cantidad de

variables globales en la parte Fortran. En este caso no fue posible un uso *TLS* (*threadprivate*) de las numerosas variables globales que había puesto que en la mayoría se utilizaba *equivalence* de Fortran en alguno de ellos, y la directiva *threadprivate* es incompatible con él. Además, en su interior se hacía uso de lectura/escritura de ficheros con nombres iguales para todas sus instancias, lo cual evidentemente produce errores.
Aprovechando que estos elementos aparecen en un número muy reducido en los circuitos y que este código se iba a reescribir en el futuro, se impidió el cálculo simultáneo de varios elementos Arbitrary 2D cavity mediante *critical*, si bien podría ser calculado concurrentemente con otro tipo de elementos.

5.4 Otros cálculos en paralelo

5.4.1 Paralelización del cálculo de los campos electromagnéticos en la estructura

El módulo de cálculo de campos tiene como objetivo obtener otra representación de la solución del circuito para una determinada frecuencia y una excitación. Se trata de calcular el campo eléctrico en el interior de la estructura bajo análisis.

Aunque no entraba dentro de los objetivos principales, se dotó de paralelismo también a la computación de los campos electromagnéticos debido a que es algo que requiere cálculos numéricos de mayor complejidad y que se presta a la paralelización. De nuevo este método se vuelve a dividir en dos procedimientos distintos denominados parte estática y dinámica.

En la parte estática se realiza el cálculo de un elemento como se ha visto en la sección 5.1 además de que también se realizan ciertos cálculos adicionales en su interior. En la paralelización del cálculo de campos se ha actuado solamente sobre el cálculo de campos de la guía rectangular arbitraria (ARW), que es la que marca la diferencia en tiempos de cómputo.

Se le ha añadido una capa nueva, por lo que aparece el paralelismo anidado sobre el lanzamiento de elementos concurrentes.
La tarea de mayor complejidad recae sobre código Fortran. En concreto, existe un bucle en el que se van calculando los campos tantas veces como modos haya (soluciones elementales del problema).

```
!$OMP PARALLEL DO DEFAULT(SHARED)
!$OMP& PRIVATE(A_IntTerm1X)
!$OMP& PRIVATE(A_IntTerm2X)
...
!$OMP& PRIVATE(A_IntGxMxTEM)
!$OMP& PRIVATE(A_IntGyMxTEM)

      DO I=1,NuModCampi
```

La mayor dificultad en este caso fue el tedioso análisis sobre qué tipo de variable tendría que ser *shared* (por defecto) o *private* (ver sección de directivas utilizadas en OpenMP). En este algoritmo, las iteraciones se dividen equitativamente con la planificación *static*, que es la planificación por defecto. La decisión de utilizar este tipo de reparto se debe a que todas las iteraciones tienen una complejidad igual, por tanto se hará un reparto equitativo de iteraciones.

La parte dinámica con los campos no se hace de forma anidada, y presenta diferencias con la parte dinámica anteriormente explicada. En este apartado están paralelizados todo tipo de elementos mediante un bucle que recorre todos los elementos. No existe ningún tipo de dependencias, pero se utiliza otro repartidor de tareas con el fin de lanzar por prioridad (estas prioridades son distintas a las de la parte estática y la parte dinámica).

Los problemas que surgieron en el cálculo de campos están relacionados en su mayoría con el uso de herramientas externas, como *geomesh* y *opencascade*, para tareas relacionadas con el mallado necesario en este tipo de análisis. El problema consistía en que estos programas hasta la fecha no han sido desarrollados de manera *thread-safe*, con lo que se vuelve a tener que utilizar la directiva *critical* para controlar las instancias a los mismos.
También había algún otro control de escritura sobre fichero que hubo que solventar.

5.4.2 Paralelización del módulo multipactor

El fenómeno de multipactor consiste en una avalancha de electrones que se genera en el interior de un dispositivo cuando éstos entran en resonancia con el campo electromagnético.
El funcionamiento del módulo de *multipactor* que se trató de paralelizar se resume en la búsqueda dicotómica de una determinada potencia a la cual se produce el fenómeno denominado *multipactor*.
Para cada potencia que se testea, existe un bucle que simula varios instantes de tiempo por los cuales una serie de electrones van modificando su posición, otros se crean y algunos se destruyen.

Dicho bucle de tiempos no se puede paralelizar al ser directamente dependiente un instante de tiempo del anterior, por lo que había que paralelizar su interior, en donde se decide la nueva posición del electrón para un instante de tiempo.

En un primer momento se paralelizó, obteniendo unos *Speed-up* razonables para la versión sin optimizaciones de compilador (-O0), hecho que no se produjo con el código optimizado (-O2 Y -O3), ya que el tamaño del problema y los tiempos de ejecución de cada iteración era tan pequeños (del orden de 10-5s) que la gestión de hilos de OpenMP contrarrestaba las ganancias, e incluso introducía pérdidas, por lo que se abandonó la paralelización de dicho módulo, dejando el lanzamiento concurrente de varias potencias (bucle más externo que busca una determinada potencia mediante búsqueda binomial) como trabajo futuro.

5.4.3 Guía rectangular arbitraria

Aunque en un principio los objetivos marcaban que se debían lanzar elementos de manera paralela, cada elemento calculándolo un thread, también se paralelizaron ciertas partes de código entre las más pesadas computacionalmente. Por ejemplo, el cálculo de los modos de la guía rectangular arbitraria (ARW), que se realiza con un método numérico complejo. En este proceso, se calculan los distintos tipos de modos (TE/TM/TEM) y se descartan las soluciones no válidas (descarte de modos TE/TM/TEM). Estos cálculos son independientes entre sí; el cálculo de los distintos modos y el descarte de cada uno.
Estas son unas condiciones perfectas para hacer uso de la técnica de tareas introducidas en OpenMP desde su versión 3.0.

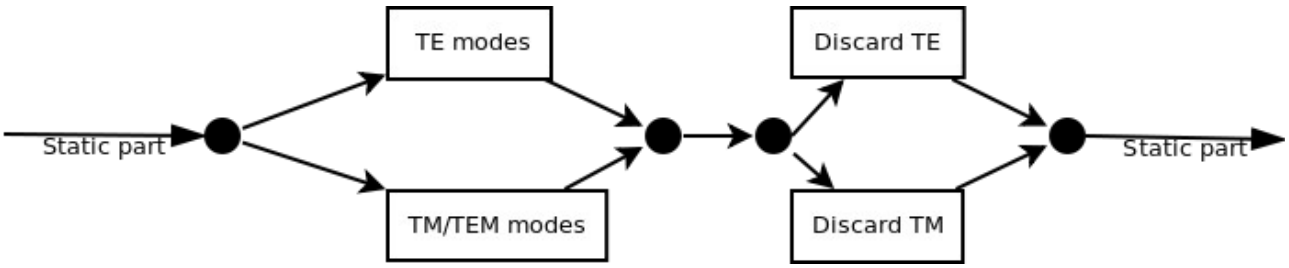

Figura 19: Esquema con las tareas creadas

La figura 19 muestra de la creación de tareas que se lleva a cabo durante el cálculo de una guía rectangular arbitraria cuando se calculan sus modos.

Aprovechando que los modos TE/TM se calculan independientemente uno del otro (en la práctica, se descubrió que los modos TEM sí dependían de los TM), se pudieron crear dos tareas que introdujeran un nivel más de paralelismo.
El hecho de usar distintas tareas para cada cálculo de modos no garantiza que vayan a ser realizados en paralelo, al estar haciendo uso de paralelismo anidado

Para implementar esto, se creó una tarea que englobara a cada una de las que componen los dos “grupos” de tareas (el grupo de cálculo de modos y el de descarte de modos).

(Fortran)

```
!$OMP TASK if(.FALSE.) DEFAULT(SHARED)

!$OMP TASK DEFAULT(SHARED)
!$OMP& PRIVATE(A_GreenSt)
!$OMP& PRIVATE(A_GreenReg)
…
!$OMP TASK DEFAULT(SHARED)
…
!$OMP TASKWAIT

…
```

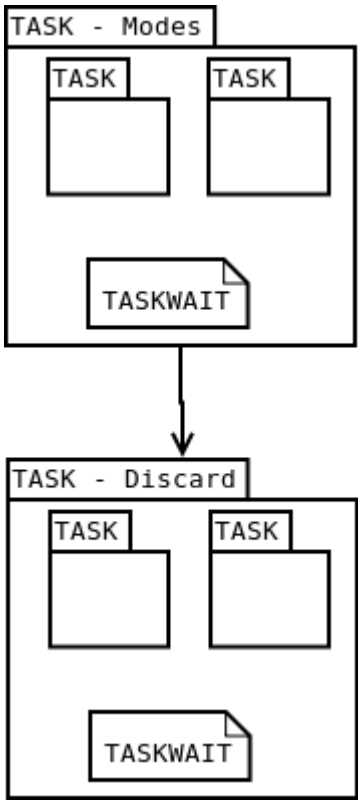

Figura 20: Estructura de

De esta manera, y usando la directiva de sincronización *taskwait*, que hace de barrera que fuerza a todos los threads pertenecientes a la tarea externa a esperar a los demás, se consigue que no se empiecen a descartar modos antes que estén calculadas todas las tareas previas.

5.4.3.1 Decisiones acerca de las tareas

La tarea que engloba a cada par, se deja con variables compartidas por no contener instrucciones propias, y se le añade la cláusula *if* para que se comience a ejecutar en el mismo momento de su creación (cosa que no está garantizada de no usar el *if*).
Según la especificación de OpenMP 3.0:

"***When an if clause is present on a task construct and the value of the scalar-expression evaluates to false, the encountering thread must suspend the current task region and begin execution of the generated task immediately, and the suspended task region may not be resumed until the generated task is completed***"

Otra de las decisiones clave era decidir sobre qué variables debían permanecer como compartidas y cuáles privadas para lo cual fue necesario de un análisis detallado del código.

En este caso la escalabilidad es muy dependiente del procesador utilizado. Cada tarea está compuesta por una serie de operaciones que culminan con una llamada a *LAPACK* que es lo que utiliza un mayor tiempo de cálculo. Dichas llamadas a *LAPACK* pueden hacer un uso intensivo de la caché del procesador, dependiendo de la talla del problema. En tal caso, y si además varios núcleos computacionales comparten caché entre sí, puede ocurrir que no se obtenga una mejoría, o que esta no sea la esperada.

5.4.4 Rectangular cavity with screws

Este elemento es uno de los que presenta una mayor complejidad en FEST3D, por lo que se estimó interesante crear una segunda capa de paralelismo que aprovechara los threads liberados en capas superiores.

Para ello se localizaron los bucles y operaciones susceptibles de ser lanzados concurrentemente.
El mayor problema surgido aquí es el uso de variables globales o *common*.

En la ejecución paralela realizada hasta ahora, cada thread se ocupaba al completo de un elemento, es decir que inicializaba ciertas variables (entre ellas globales), que más adelante se utilizarían. Dichas variables globales se declaraban como *threadprivate* y así no había problemas con otros elementos que utilizaran esa misma variable con otros valores.

Al utilizar el paralelismo anidado y crearse nuevos threads dentro, las variables declaradas *threadprivate* estarán inicializadas al valor por defecto, y no al valor que el thread que ha creado la nueva zona paralela tiene.
La cláusula *copyin* establece que en estos nuevos threads se copie el valor que tiene el thread padre de ciertas variables *threadprivate*.

5.4.5 Problema asociado al paralelismo anidado y OpenMP

En el momento que un thread se encuentra con la definición de una región paralela en OpenMP, este crea tantos hijos como hagan falta hasta un máximo que es igual al número de threads que se haya indicado con la función omp_set_num_threads(), la variable de entorno OMP_NUM_THREADS o añadiendo NUM_THREADS(...) en la propia directiva *parallel*.

En el caso del paralelismo anidado, cada uno de esos 'n' threads que se han creado genera de nuevo por defecto ese mismo número de threads, lo que provoca la existencia de *n x n* threads. Este comportamiento no es correcto, ya que se están utilizando más threads de los que en un inicio se han prefijado.

En el momento de finalización de este trabajo, las especificaciones de OpenMP no marcan que esto se deba controlar, aunque sí esté siendo sometido a investigación la manera de hacer uso únicamente de los threads que estén libres, y no crear más de los que se han fijado.

Una de las maneras de abordar esto sería el control dinámico del número de threads realizado directamente sobre FEST3D. Esto es complicado por el uso del propio paralelismo anidado y de distintos lenguajes de programación mezclados. Esto será tarea de trabajo futuro para la mejora de FEST3D.

6 Resultados (parte estática y dinámica)

Todas estas pruebas han sido realizada sobre un biprocesador *quadcore* (*Intel Xeon* E5504 a 2.00GHz). El sistema operativo elegido para las pruebas es Ubuntu GNU/Linux de 64 bits con *gcc* y *gfortran* 4.4.3.

Los casos de estudio se han escogido específicamente para mostrar la gran variabilidad de rendimiento y eficiencia que resulta de la paralelización de FEST3D.

6.1 Caso de estudio nº 1

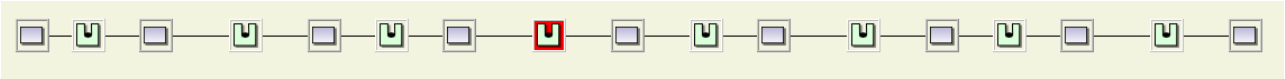

Figura 21: Caso de estudio 1

En este caso de estudio y en el siguiente se aprecia claramente la variabilidad en términos de *Speed-up* y eficiencia que se obtiene con la paralelización propuesta, dependiendo del circuito.

Este circuito contiene únicamente dos tipos de elementos, varias guías rectangulares unidas con la discontinuidad *Rectangular cavity with screws*.

Las guías rectangulares son de cómputo casi instantáneo, por tanto no afectan en demasía a el tiempo total de simulación.
Aquí, el peso de la simulación recae sobre el cálculo de las discontinuidades, las cuales se repartirán entre los distintos threads durante el proceso.

Nthreads	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Tiempo(s)	153,5	77,5	58,4	39,4	35	30	26,8	21,1

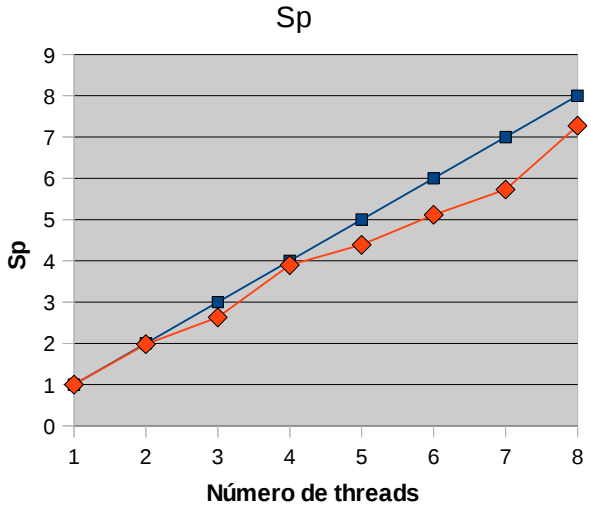

Figura 22: Speed-up: Caso de estudio 1

En la gráfica del *Speed-up* se aprecia que existe un *Speed-up* sublineal, pero que se aproxima al lineal, que es el ideal. La clave de este circuito es que todos los elementos tardan un tiempo muy parecido en ser computados. Si esto no es así, el equilibrio de la carga no sería el correcto; en este caso lo es.
Además de esto, el elemento *Rectangular cavity with screws* tiene zonas paralelizadas en su interior. De no ser así, el uso de 5 ó 6 threads resultaría igual, ya que hay 8 discontinuidades, y al no ser múltiplo del número de threads, siempre van a existir threads libres mientras otros completan un elemento al completo.
Con este elemento no es así, y estos threads libres se aprovechan para el cómputo interno de alguna discontinuidad que esté calculándose (si bien no se obtiene todo el rendimiento deseable como se puede apreciar que ocurre en la figura con 5, 6 y 7 threads, se disminuye el tiempo de cálculo).

6.2 Caso de estudio nº 2

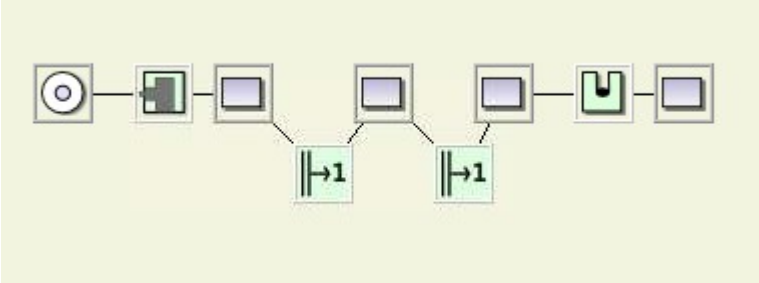

Figura 1: Caso de estudio 2

En este caso hay nueve elementos para la parte estática. De ellos, dos guías rectangulares son iguales, y los dos steps (en la parte inferior de la figura), también. Por tanto, solamente se calculará uno de ellos.

Nthreads	1	2	3	4	5	6	7	8
Tiempo(s)	18,8	18,6	18,6	18,5	18,2	18,2	18,3	18,3

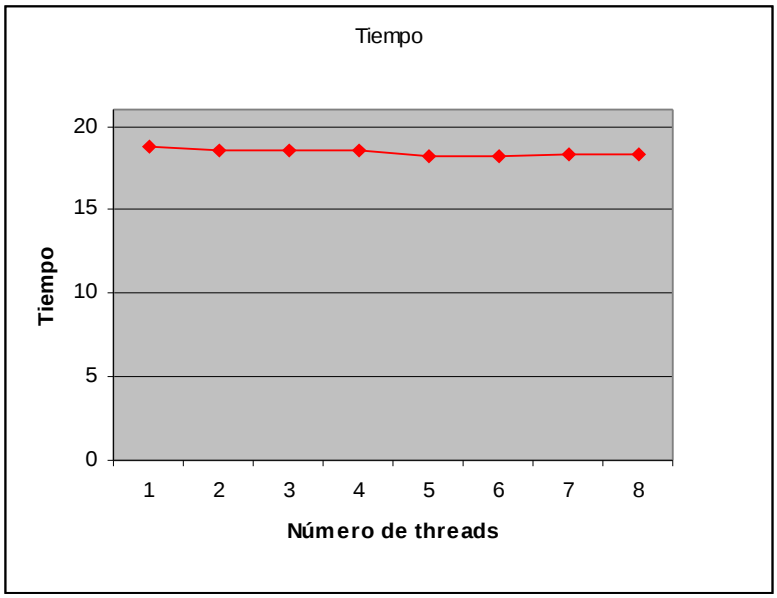

Figura 23: Caso de estudio 2

Durante la ejecución se puede comprobar que hay un elemento en el circuito (general 3D cavity) que representa 18 segundos de la ejecución. Dicho elemento es computado por un thread únicamente, al no tener ninguna capa interior de paralelización.

Las implicaciones en el rendimiento son claras: no existe apenas *Speedup* al existir un elemento tan costoso computacionalmente con respecto a los demás. Este elemento por el sistema de pesos se reparte el primero, y el resto se van a hacer entre tanto en paralelo pero, al existir varios órdenes de magnitud de diferencia de tiempos entre este y lo demás, no se nota mejoría con multithread.

Por esto queda claro que es necesario paralelizar internamente los elementos con mayor complejidad algorítmica.

6.3 Caso de estudio nº 3: Paralelización del elemento Rectangular Cavity with screws

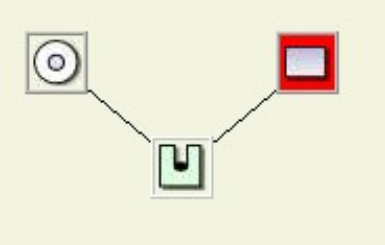

Figura 24: Caso de estudio 3

Este circuito sirve para analizar el rendimiento que se obtiene en la paralelización interna de este elemento. Para ello, se prueba con un circuito en el que solamente hay tres elementos:

- Guía rectangular
- Guía coaxial circular
- Discontinuidad *Rectangular Cavity with screws*

En la parte estática, los dos primeros resultan casi inmediatos, por tanto el tiempo pesado recae sobre el tercero, que es el interesante en esta prueba y en el cual aparece la paralelización anidada en ciertas zonas del cómputo. Al ser tres elementos de distinto tipo entre ellos, no existe ningún reaprovechamiento de los cálculos de un elemento hacia otro.

Nthreads	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Tiempo	53,7	33,3	25,7	22,4	20	18,6	18,7	18,7

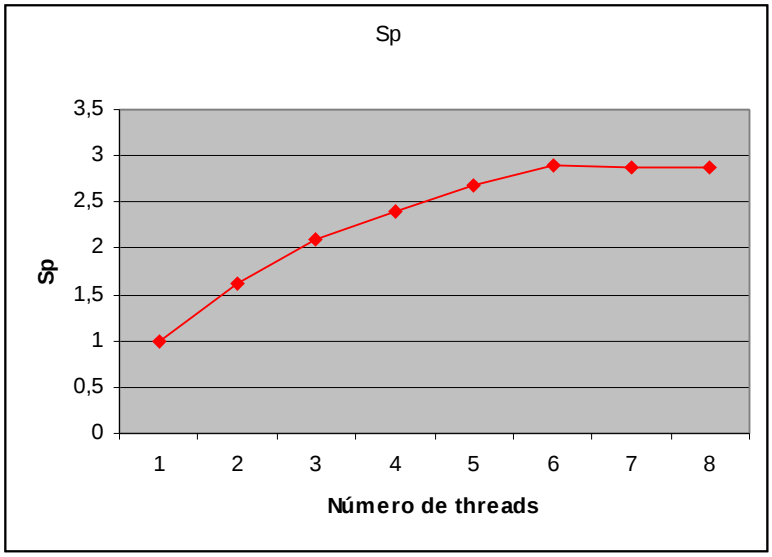

Figura 25: Caso de estudio 3

La paralelización anidada de este elemento se limita a dos zonas de código que pueden representar más o menos complejidad del total dependiendo de la geometría del elemento.

En este circuito, aproximadamente 1/3 del tiempo de cómputo del elemento, se realiza con un solo thread, por lo que según la ley de Amdahl el *Speedup* está limitado a una cota superior de 3.
Además, dentro de una de las dos zonas paralelas que aparecen en este elemento, el equilibrio de carga no es el óptimo, ya que se reparten filas de una matriz diagonal a cada thread, siendo el número de filas del mismo orden que el número de threads en esta prueba.

A pesar de que teóricamente el rendimiento está limitado por el porcentaje de tiempo que se realiza el cálculo en paralelo (y en este caso no es del 100%), este paralelismo anidado ayuda a mejorar los tiempos en los instantes de tiempo en los que haya núcleos computacionales sin carga de trabajo.

7 Conclusiones

Hemos presentado el trabajo realizado en la empresa AURORASAT a lo largo de estos meses, consistente en la paralelización del proyecto FEST3D, y alcanzando los objetivos con éxito.

La conclusión más importante que se extrae de este proyecto es la de la dificultad de paralelizar una aplicación compleja y con muchos años de desarrollo. En el momento de la conclusión de este proyecto, FEST3D es una aplicación que reúne algoritmos de hasta 20 años de antigüedad. No sólo eso, sino que dichos algoritmos están escritos en diversos lenguajes (C, C++, Fortran 77, Fortran 90) por personas que en algunos casos ya no pertenecen al proyecto y, por tanto, esto añade un plus de dificultad a la hora de comprender los cálculos que se realizan y solventar problemas modificando dichos algoritmos para dotarlos de cierta concurrencia.

Por su parte, FEST3D contiene una gran casuística en el flujo de un programa ya que es el usuario quien diseña y crea el circuito que se va a simular. Esta variabilidad se acrecienta por el uso de las tecnologías multinúcleo y dificulta una fácil depuración.

Por eso la depuración ha sido una de las tareas más arduas en la realización de este proyecto, superando con creces al diseño del comportamiento que iba a seguir el paralelismo. Al tratarse de un código tan sumamente extenso, es imposible predecir fallos, y la metodología a seguir era la de prueba y error, utilizando circuitos existentes y probando ciertos cambios, pruebas con distinto número de threads, múltiples repeticiones de cada prueba, etc.

Ante un error, el depurador muy raramente localizaba el fallo y había que realizar trazas manuales, o utilizando la directiva *critical*, *atomic*, semáforos, etc que dieran pistas sobre la naturaleza del error.

Este proyecto también ha servido para descubrir errores en ciertos algoritmos no relacionados con la paralelización de FEST3D pero que han salido a la luz gracias al uso de ella.

En el apartado de mantenibilidad se tiene que mencionar el hecho de que, como se ha actuado sobre varias capas (lanzamiento simultáneo de elementos y en algunos casos paralelización interna añadida), las nuevas características que se añadan al simulador pueden provocar errores de cómputo o excepciones. Por tanto se presume como necesario un mantenimiento mínimo pero constante en la adición o modificación de nuevas propiedades en el programa.

8 Trabajo futuro

Tal y como se señala en la sección de resultados y en otros puntos de esta memoria, queda bastante claro el trabajo futuro.

Por una parte, está la paralelización interna de ciertos elementos marcados como los más complejos algorítmicamente a priori. De esta forma se asegura un mayor aprovechamiento de todos los núcleos computacionales, que desemboca en una mayor eficiencia.

La dificultad en este apartado reside en localizar una parte del cómputo de un elemento que contenga una determinada complejidad de forma que sea rentable paralelizar. Esta situación no siempre se cumple, sino que puede que el cálculo de un elemento esté compuesto de muchas tareas pequeñas y dependientes entre sí. Para cambiar esto habría que cambiar el algoritmo entero, utilizando métodos numéricos distintos, tarea que no resulta factible.

En segundo lugar, queda la optimización de la paralelización aplicada hasta ahora en términos de memoria, es decir, la utilización compartida de variables y estructuras que no vayan a ser modificadas, o el cambio del propio algoritmo de manera que funcione de manera que utilice menos memoria principal cuando se ejecute de manera concurrente.

En tercer lugar, existen ciertos módulos en FEST3D de cómputo como *multipactor* o *corona*, que son susceptibles de ser paralelizados.

También se tratará el equilibrio entre la paralelización realizada con la que se va a añadir en forma de rutinas lapack paralelizadas por la librería MKL de Intel. Esto requerirá de un estudio en cada caso sobre qué alternativa a utilizar es la mejor o, en el caso que se estime conveniente, las dos.

Es necesario también la creación de unas directrices y buenos hábitos de programación de manera que el hecho de dotar al software de paralelismo en sus distintas capas sea mucho más fácil y contenga menos errores.

9 Bibliografía

- **OpenMP specifications** (May, 2008) - http://openmp.org
- **Paradigma de memoria compartida: OpenMP** (2008) – José E. Román (UPV)
- **Introducción a la programación paralela** (2007) - Francisco Almeida. Domingo Giménez. José Miguel Mantas. *Antonio* M. *Vidal*
- **Adaptación de la herramienta de diseño asistido por ordenador de circuitos pasivos de microondas, FEST3D, para su distribución a la industria espacial** (Jul, 2007) – Jaime Armendáriz Villalba
- **FEST 3.0 reference manual** (Ago, 2001) - Michael Mattes
- **FEST3D Online help**