Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXXI, págs. 3153-3164 (artículo completo)
Alberto Cardona, Paul H. Kohan, Ricardo D. Quinteros, Mario A. Storti (Eds.)
Salta, Argentina, 13-16 Noviembre 2012

PARALELIZACIÓN DE UN CÓDIGO DE ELEMENTOS FINITOS EN MULTIPROCESADORES DE MEMORIA COMPARTIDA

Emmanuel Rojas Fredini[b], Federico Benitez[b], Alejandro Cosimo[a,b] y Alberto Cardona[a,b]

[a]*Centro Internacional de Métodos Computacionales en Ingeniería (CIMEC-INTEC), Argentina, cimec@ceride.gov.ar, http://www.cimec.com.ar/*

[b]*Universidad Nacional del Litoral-CONICET, Güemes 3450, S3000GLN Santa Fe, Argentina, fich@fich.unl.edu.ar, http://www.unl.edu.ar/*

Palabras Clave: Método de los elementos finitos, Computación paralela, Multiprocesadores de memoria compartida, Programación orientada a objetos, C++

Resumen. El costo computacional asociado a la resolución de problemas descriptos por ecuaciones diferenciales por medio del Método de los Elementos Finitos (MEF), se incrementa a medida que aumenta la cantidad de grados de libertad. La paralelización de los cálculos se presenta como solución natural a este problema, ya que algorítmicamente las contribuciones elementales de la discretización del MEF tienen un carácter local. En este trabajo se estudia la paralelización de los cálculos en arquitecturas multiprocesador de memoria compartida. Se utilizará *OOFELIE* (*Object Oriented Finite Element method Led by Interactive Executor*) como código base secuencial para la implementación y análisis de las ideas propuestas. *OOFELIE* fue diseñado bajo el paradigma de la programación orientada a objetos, por lo que fue necesario el análisis de dicho diseño para detectar las posibles restricciones a la paralelización del código. Se tiene como objetivo lograr que la adaptación presente una escalabilidad razonable. Para ello, primero se realiza un diagnóstico de la arquitectura de *OOFELIE* detectando posibles restricciones. Luego se plantea la solución paralela y se concluye con un análisis de escalabilidad y propuestas de posibles mejoras.

1. INTRODUCCIÓN

La simulación numérica de problemas descriptos por ecuaciones diferenciales es un área de continua expansión en la comunidad científica y la industria. Conforme avanza el tiempo, las exigencias de los problemas a resolver y la calidad de las soluciones crece de forma acelerada (NRC (2008)), exigiendo así un incremento en el tamaño y complejidad del modelo numérico asociado. Esto implica que el tiempo de cálculo y los recursos computacionales afectados a la solución aumenten. De esta forma los requerimientos computacionales han sobrepasado el poder de cálculo que ofrecen los procesadores mononúcleo buscándose otras alternativas. Se pueden encontrar arquitecturas de memoria distribuida (*Distributed-Memory Processing* -DMP) y de memoria compartida que pueden acceder de manera simétrica a la memoria (*Symmetric Multi-Processor* -SMP) o de manera asimétrica, siendo estas últimas conocidas como arquitecturas NUMA (Non-Uniform Memory Access) o arquitecturas de memoria compartida distribuida. En las arquitecturas SMP y NUMA todas las unidades de ejecución pueden direccionar el mismo espacio de memoria, aunque la primera se diferencia de la segunda en que la latencia de acceso a memoria es la misma para cada unidad de ejecución. Por otro lado están las arquitecturas DMP en donde las unidades de ejecución no direccionan el mismo paso de memoria. Otro enfoque, que se encuentra en boga actualmente, consiste en hacer uso de coprocesadores de gran potencia de cálculo como lo son las GPGPUs o *General Purpose computing on Graphics Processing Units*.

Uno de los métodos más utilizados para la simulación de procesos físicos es el MEF. En este método la localidad de los cálculos elementales presenta la paralelización como una solución natural a la problemática de la creciente complejidad. La misma está relacionada con la cantidad de grados de libertad (*degrees of freedom -DOFs*) del problema. Numerosos trabajos han sido desarrollados para adaptar el método a las distintas arquitecturas multinúcleo antes mencionadas. Se han propuesto soluciones basadas en diferencias finitas (*finite differences*) para adaptar el método utilizando GPGPU mediante *Compute Unified Device Architecture* (CUDA) (Costarelli et al., 2011). Otros trabajos han adaptado el MEF a arquitecturas DMP utilizando *Message Passing Interface* (MPI) (Anderheggen y Renau-Munoz, 2000; Brown et al., 2000). También se han implementado soluciones en arquitecturas de memoria compartida haciendo uso del estándar *Open Multiprocessing* (OpenMP) (Pantalé, 2005).

En el presente trabajo se presenta la implementación de la paralelización del MEF en arquitecturas de memoria compartida sobre el programa *OOFELIE*, desarrollado por Cardona et al. (1994), que responde a un diseño de paradigma secuencial. Una característica primaria de dicho diseño fue la programación orientación a objetos, que proporciona ventajas destacables en la utilización del software y la expansión de sus funcionalidades. No obstante la performance de los diseños orientados a objetos generalmente son inferiores a los que utilizan el paradigma de programación estructurada. En el trabajo se eligió no abandonar las ventajas del paradigma orientado objetos mientras que se procuró desarrollar un código con una buena performance y con una ejecución concurrente escalable dentro de los límites de una arquitectura de memoria compartida. En general hay dos etapas del método donde es crítico paralelizar la ejecución del algoritmo: primero en el ensamblado de los vectores y matrices globales, y segundo en la resolución del sistema de ecuaciones resultante. El trabajo realiza la paralelización de la primer etapa debido a que la segunda ya es realizada por la biblioteca Intel *Math Kernel Library* (MKL) (Kumbhar et al., 2011) que ofrece un nivel de paralelización y optimización difícil de superar en el contexto de métodos de solución directa.

2. METODOLOGÍA

Desde un punto de vista general el problema de paralelizar el código de elementos finitos del que se dispone, se reduce a lo siguiente:

- Inicializar las estructuras de datos que son modificadas simultáneamente por los hilos (*threads*).
- Dividir el trabajo de forma tal de evitar secciones críticas.
- Prohibir estrictamente el uso de polimorfismo en las operaciones del núcleo.
- Evitar el uso de variables estáticas para obtener código reentrante (*reentrant code*).
- Verificar que las funciones llamadas por los hilos sean reentrantes.

A continuación se da una descripción detallada de cada uno de los puntos. Se paralelizan el ensamblado de los vectores y matrices globales, aunque en lo que sigue se hace especial énfasis en el ensamblado de las matrices globales.

2.1. DIVISIÓN CONCURRENTE DE LA ESTRUCTURA DE SOPORTE

Al realizar el ensamblado de la matriz global es necesario procesar los cálculos elementales de todos los elementos del problema; estos cálculos serán la unidad o *kernel* de paralelización. Sin embargo no es posible realizar el cálculo de todos los elementos de forma concurrente ya que es esperable que dos elementos diferentes realicen aportes a un mismo grado de libertad, lo cual da lugar a que se intente sumar un valor a una misma componente de la matriz global de forma simultanea entre dos hilos (*threads*), produciendo una condición de carrera (*race condition*). Para poder realizar los cálculos elementales de forma concurrente y *thread safe* será necesario realizar una división en conjuntos de elementos que no posean ningún grado de libertad en común. Ese proceso de división lo realizaremos mediante un algoritmo de coloreo, donde cada elemento se colorea con un color que actúa como identificador del conjunto. Luego se pueden procesar de forma concurrente todos los elementos de un mismo color, ensamblando en paralelo las contribuciones elementales sobre la estructura que implementa la matriz global.

El algoritmo de coloreo debe cumplir con dos condiciones que en general suelen ser opuestas: que se realice un coloreo con la menor cantidad de colores posible y que su ejecución esté dado por un tiempo razonable. Lo primero garantiza que se podrá utilizar la paralelización de forma más eficiente, y lo segundo que el proceso de coloreo no compita en recursos con el ensamblaje de la matriz global.

En las subsecciones siguientes se explica la estructura de datos que se utilizó, el algoritmo de coloreo y sus resultados.

2.1.1. ESTRUCTURA DE GRAFO DUAL

Como se dijo en la sección anterior el coloreo se realiza por elementos de la malla. Esto pone en manifiesto la necesidad de calcular el grafo dual (de Berg et al., 2008) asociado a esa malla para poder realizar el coloreo. Este grafo condensa en su estructura la información de las adyacencias de elementos. Sea $\mathcal{G}_\mathcal{M}$ el grafo dual de una malla $\mathcal{M}$. Luego $\mathcal{G}_\mathcal{M}$ tendrá un vértice por cada elemento de la malla asociada. Denótese al elemento de $\mathcal{M}$ correspondiente al vértice i de $\mathcal{G}_\mathcal{M}$ como $e\,(i)$. Entonces habrá una arista entre los vértices i y j de $\mathcal{G}_\mathcal{M}$ si los elementos $e\,(i)$ y $e\,(j)$ son adyacentes. Es decir que los vértices de $\mathcal{G}_\mathcal{M}$ se corresponderán con las caras

compartidas entre elementos de $\mathcal{M}$. En la figura 1 podemos ver una malla y su grafo dual. Se puede demostrar que toda malla posee un grafo dual asociado.

En la implementación se procedió a utilizar funciones de la biblioteca *METIS* (Karypis, 2011). Esta biblioteca ofrece funcionalidad para realizar particionamiento de grafos, o en particular mallas de Elementos Finitos. La biblioteca recibe como parámetros el grafo de la malla en formato *Compress Store Format* (CSR), retornando el grafo dual también en formato CSR.

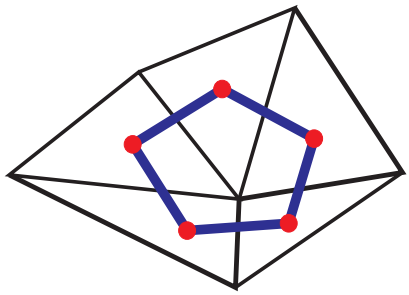


Figura 1: Una malla, en color negro, y su grafo dual asociado, aristas azules y nodos rojos

2.1.2. ALGORITMO DE COLOREO

El objetivo del algoritmo de coloreo será colorear con la mínima cantidad de colores posible y en un tiempo razonable. Esto como se dijo logrará una paralelización más eficiente, debido a que permitirá poder repartir más volumen de carga de trabajo a los distintos colores, minimizando así el *overhead* del trabajo completo. El problema de colorear se clasifica como un problema NP-duro (Kubale, 2004), lo que implica en la práctica que el no puede ser resuelto en tiempo polinomial. Es por ello que se propone utilizar un método de coloreo ávido de tipo codicioso (*greedy*), que consiste en realizar el coloreo aplicando una heurística razonable, sacrificando la optimalidad en la cantidad de colores de resultado.

Dicho método es el siguiente:

- Asignar al primer nodo un color
- Por cada nodo:
  - Identificar el color de cada vecino
  - Luego asignar el color mínimo que no sea igual a algún color de su vecindad

Siempre y cuando el grafo no sea completo, el costo computacional no es alto. En la figura 2 se puede ver el resultado de aplicar el algoritmo a una malla estructurada.

Se realizaron pruebas de rendimiento sobre este algoritmo para medir sus dos parámetros claves: orden de complejidad computacional y cantidad de colores utilizados. Estas pruebas se realizaron sobre mallas estructuradas de elementos triangulares, incrementando en cada malla el número de elementos. La mallas utilizadas fueron de $n \times n$ nodos, i.e. $(n-1) \times (n-1) \times 2$ elementos. En la figura 3 se puede ver el resultado de las pruebas. Como se puede observar estos resultados demuestran que el algoritmo posee un orden de complejidad lineal con la cantidad de elementos de la malla, $O(\eta)$ donde η es la cantidad de elementos en la malla. Además se

puede ver que, para la malla estructurada propuesta, el número de colores utilizado es bajo y no varia con el aumento de la cantidad de elementos de la malla.

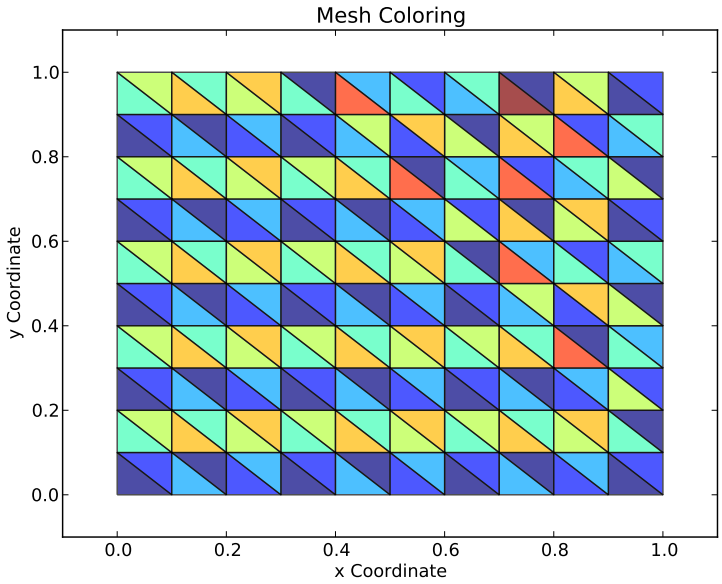


Figura 2: Ejemplo de coloreo de una malla estructurada

2.2. PARALELIZACIÓN DEL ENSAMBLAJE

Ya en parte se describió la algoritmia a seguir para paralelizar el ensamblaje de la matriz global. Siendo más específicos se sigue la algoritmia propuesta por Farhat y Crivelli. (1989), que se resume en la figura 4. Para que la paralelización propuesta sea aplicable a una gran cantidad de problemas, primero los elementos se agrupan según el tipo de problema, luego se calcula el grafo dual asociado a ese grupo de elementos. Paso seguido se colorea el grafo dual para particionar el conjunto de elementos en subconjuntos de elementos no adyacentes. De esta manera no existen condiciones de carrera al ensamblar la matriz global y se puede procesar en paralelo cada color de manera *thread-safe*.

En la figura 5 se pueden observar simplificadamente los módulos más importantes que se implementaron como parte de solución propuesta. En primera instancia se necesitan los conjuntos de elementos coloreados. Situación implementada por la clase `CSRGraphStr`, que básicamente tiene los siguientes comportamientos:

- Calcula del grafo dual de la malla por medio del método `Meshtodual`.
- Obtiene el grafo coloreado por medio del método `Greedy`.

El ensamblaje en paralelo de la matriz global lo efectúa la clase `ParBlockMatrix` siendo sus métodos más importantes `Update` y `DryUpdate`. Para que no se genere una sección crítica relacionada al dimensionado de la matriz, se calcula su perfil antes de entrar en la etapa de ensamblado y se pide la cantidad de memoria necesaria. Esto es realizado por el método `DryUpdate` en conjunción con la clase `CSRwpMatrix`. Básicamente se recorren todos los elementos pero sin calcular sus contribuciones, determinando únicamente en qué posición de la matriz global van a contribuir. A continuación se muestra la secuencia de pasos que describe el cálculo del perfil de la matriz global y su asignación de memoria:

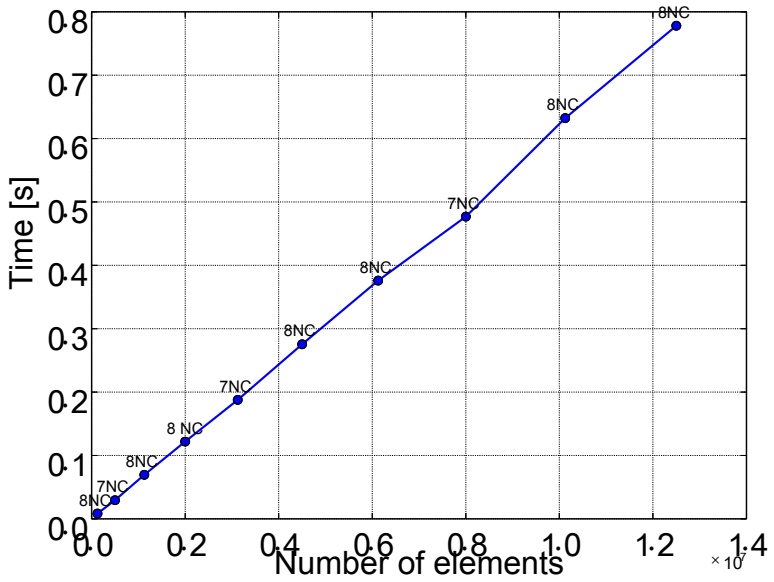


Figura 3: Tiempo vs. número de elementos. En cada punto del gráfico de dispersión se muestra la cantidad de colores (NC) diferentes que utilizó el algoritmo

- Inicio de la etapa de asignación de memoria: este inicializa las variables necesarias y se prepara para la asignación de memoria; se encuentra implementado en el método `openFillingStage`.
- Recolección de información: se determina el espacio que se necesita en la matriz global y se guarda dicha información mediante el método `setDryvalue`.
- Cierre de la etapa de asignación de memoria: una vez recolectada la información de cuánto espacio es necesitado, se procede a pedir dicho espacio; este se encuentra implementado en el método `closeFillingStage`.

Debido a la estrategia utilizada para paralelizar el ensamblaje de la matriz, sólo hace falta procesar en paralelo el bucle que recorre los elementos coloreados de un color dado. Luego, como no se hace necesaria gran flexibilidad desde el punto de vista de manejo de los hilos se eligió como API a *OpenMP*, que utiliza el modelo *fork-join* para la ejecución de secciones paralelas (Sato, 2002). Además a la razón ya mencionada, esta selección se realizó teniendo en cuenta varios puntos relacionados con el software *OOFELIE*, entre estos se encuentran:

- Ser multiplataforma, debido a que *OOFELIE* esta siendo y debe continuar siendo soportado en varios sistemas operativos, entre los que podemos nombrar Windows y Linux.
- Soportar el lenguaje de programación C++, dado que es el lenguaje en que esta implementado *OOFELIE*.
- Trabajar en la arquitectura de multiprocesadores de memoria compartida, a causa de que *OOFELIE* fue diseñado en principio para esta arquitectura.

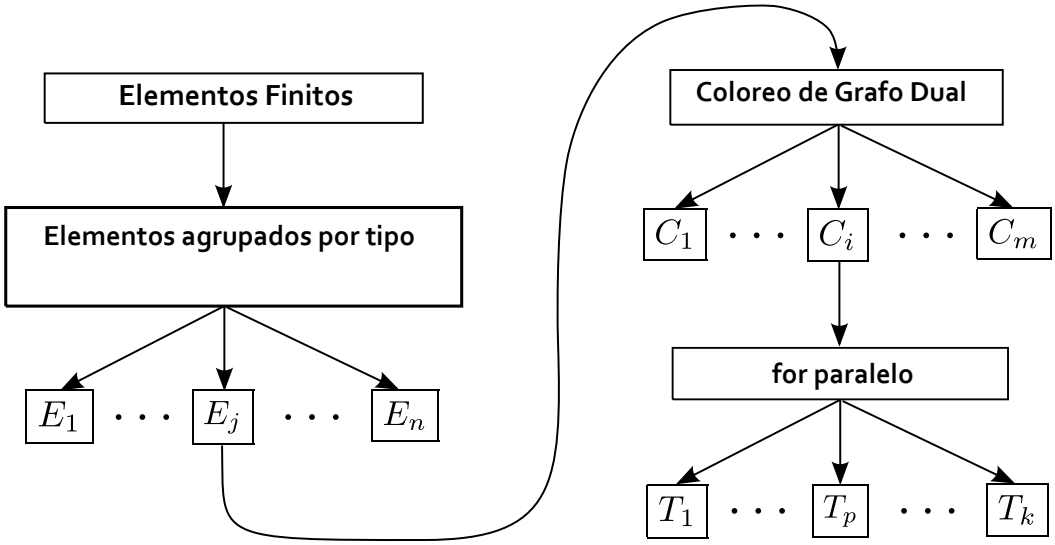


Figura 4: Diagrama de la estrategia computacional.

3. RESULTADOS

Para probar el rendimiento del método propuesto, en dos máquinas de distintas presentaciones se realizaron pruebas de *speedup*, eficiencia, escalabilidad y de comparación de tiempos entre etapas. Cabe destacar que las pruebas con un *thread* se realizaron con el algoritmo paralelo propuesto, limitando la cantidad de hilos en la ejecución. Para llevar a cabo las pruebas se resolvió un problema de conducción de calor transiente. Se utilizó el MEF para discretizarlo y a pesar de que se corrieron problemas lineales se hizo uso del esquema iterativo Newton-Raphson. A parte de esto, la formulación del problema difiere de su habitual formulación discreta MEF, ya que se introduce una burbuja cuasi-estática para estabilizarlo y así eludir posibles oscilaciones espurias características de problemas de choque térmico.

3.1. EQUIPOS DE PRUEBA

Las pruebas de rendimiento se realizaron sobre dos equipos:

1. Equipo I: *Computadora con 12 núcleos, hyperthreading desactivado*:
   - Dos procesadores Intel® Xeon™ CPU X5680 @ 3.33GHz (6 núcleos físicos cada uno).
     - 32 KB Cache L1 para instrucciones y 32 KB Cache L1 para datos por núcleo físico.
     - 256 KB Cache L2 por núcleo físico.
     - 12 MB Cache L3 compartido.
   - Memoria RAM: 98944852 KB.
2. Equipo II: *Computadora con 4 núcleos, hyperthreading activado*:
   - Intel® Core™ i7-930 @ 2.80 GHz (4 núcleos físicos).

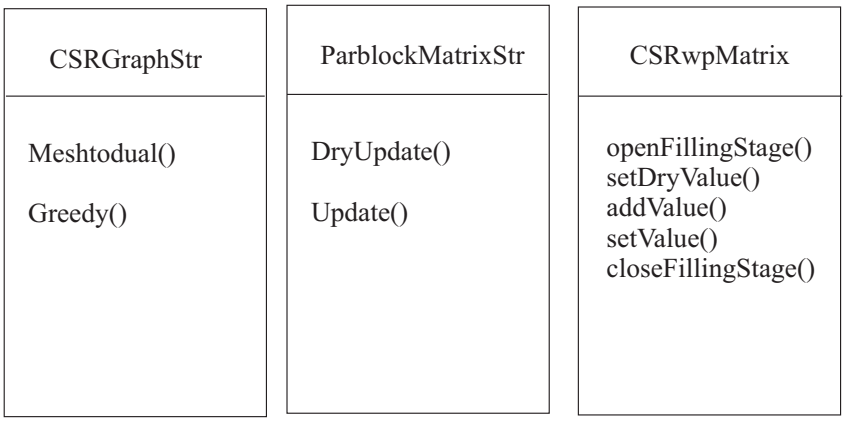


Figura 5: Módulos simplificados que intervienen en la paralelización

- 32 KB Cache L1 para instrucciones y 32 KB Cache L1 para datos por núcleo físico.
- 256 KB Cache L2 por núcleo físico.
- 8 MB Cache L3 compartido.

- Memoria RAM: 6 GB.

El primer equipo posee una arquitectura **multi-procesador NUMA**, con lo cual se tendrán accesos a memoria asimétricos. El segundo equipo tiene sólo un procesador por lo que latencia de acceso a memoria es sólo una. Es esperable obtener mejores *speedups* en el segundo equipo debido a no se tienen accesos asimétricos a memoria. Un estudio más detallado del comportamiento de la paralelización propuesta en arquitecturas NUMA se deja como trabajo futuro.

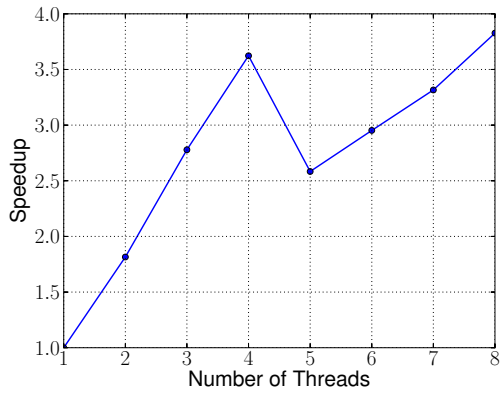
3.2. PRESENTACIÓN Y ANÁLISIS

En la figura 6 se puede ver el resultado de las pruebas ejecutadas en el Equipo II. Para realizar las pruebas se ha utilizado una malla de medio millón de elementos triangulares. En la figura 6a se ve el *speedup* del ensamblaje de la matriz global. Se aprecia que para cuatro *threads*, teniendo tanto *threads* como núcleos físicos, se tiene un *speedup* de aproximadamente 3,6. Este resultado es positivo debido a que es muy cercano al límite teórico máximo para ese equipo. También se puede observar que a partir de los cinco *threads* comienza a afectar la tecnología *HyperThreading*, lo que se ve reflejado en la curva de *speedup*. En la figura 6b se presenta el gráfico de eficiencia, el cual es otra métrica del rendimiento paralelo. Al igual que en el *speedup* a partir de los cinco *threads* la curva empeora, como se esperaba, debido a la tecnología *HyperThreading*. En la figura 6d se puede ver que al aumentar la cantidad de elementos del problema la eficiencia aumenta para una cantidad de cuatro *threads*. Por último en la figura 6c se realizó un gráfico de barras que compara los tiempos esperados que consume cada uno de los procesos del algoritmo utilizando cuatro *threads*. La última barra del gráfico es el tiempo total, es decir la composición de tiempos del resto de las barras. El proceso que más tiempo consume es el de resolución de la matriz global, utilizando *Pardiso*. Uno de los tiempos de mayor interés es el del ensamblaje de la matriz global. Por último se puede ver que los tiempos de cálculo del grafo dual y del coloreo no representan un alto costo computacional

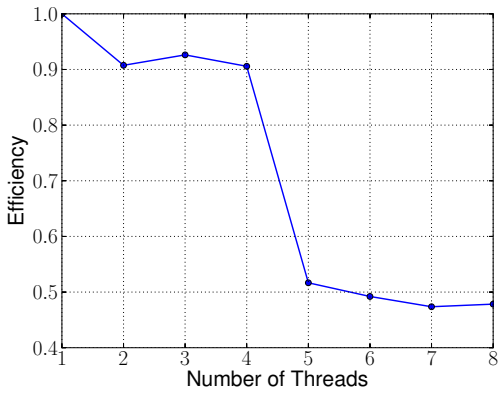
en comparación al resto de los procesos.

En la figura 7 se pueden ver los resultados de las pruebas, realizadas de forma análoga, ejecutadas en el Equipo I. Para realizar las pruebas se ha utilizado una malla de doce millones y medio de elementos triangulares. Los resultados obtenidos en este equipo son muy diferentes a los obtenidos en el equipo anterior. En la figura 7a se puede ver que si bien el *speedup* conseguido crece con el número de *threads*, el crecimiento es lineal pero con una pendiente pequeña. Entonces para una cantidad de doce *threads* apenas se consigue un *speedup* de aproximadamente $3{,}4$. Este pobre resultado es evidente en la figura 7b donde se muestra la eficiencia obtenida, la cual es una curva que rápidamente decrece. Debido a estos resultados se concluye que el método propuesto no es eficiente para arquitecturas **multi-procesador NUMA**.

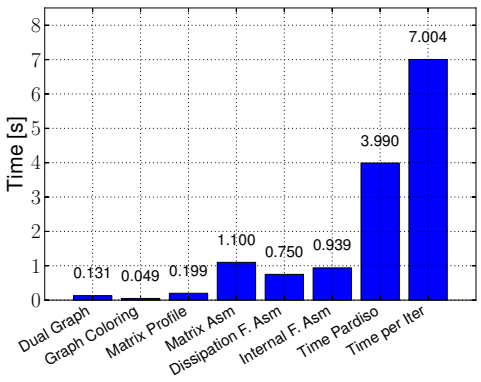
Debido a la anomalía presentada anteriormente se realizó un análisis comparativo entre el ensamblaje de la matriz global y la resolución del sistema realizada por *Pardiso*. En la figura 8 se pueden ver las gráficas resultantes. En la figura 8b se ve que *Pardiso* alcanza un *speedup* mayor al alcanzado por el ensamblaje de la matriz global, es decir que *Pardiso* consigue aprovechar mejor los recursos computacionales del equipo. En la figura 8a se repite la misma conclusión anterior. La clara ventaja de *Pardiso* puede deberse a dos motivos: la naturaleza diferente de los problemas o a un mejor aprovechamiento de la arquitectura. Esto plantea una posible línea de investigación sobre optimizaciones en esta arquitectura.



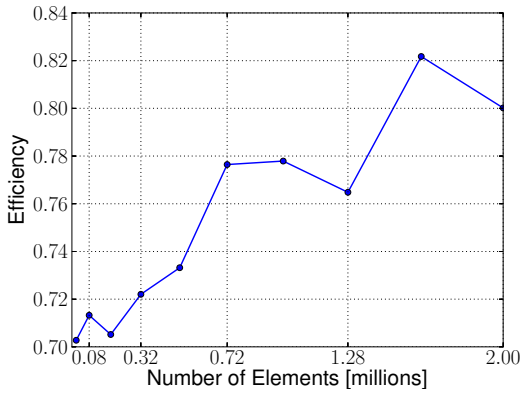
(a) Speedup (0.5 million triangular elements)



(b) Efficiency (0.5 million triangular elements)

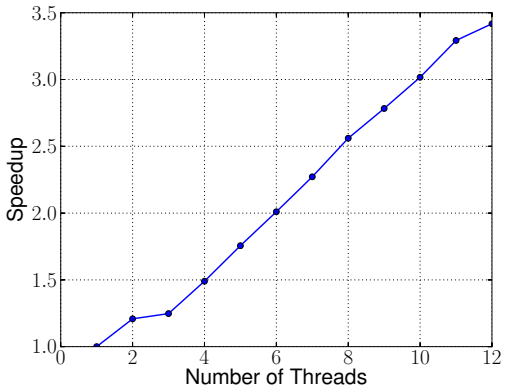


(c) Times (0.5 million triangular elements)

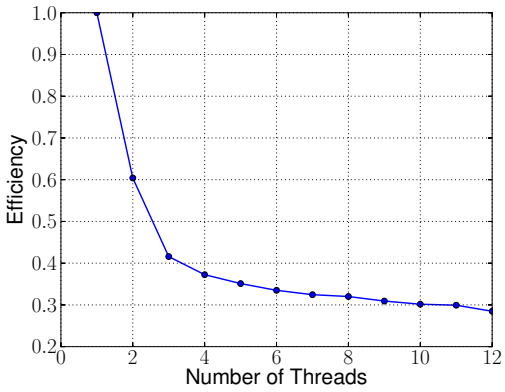


(d) Scalability study (four threads)

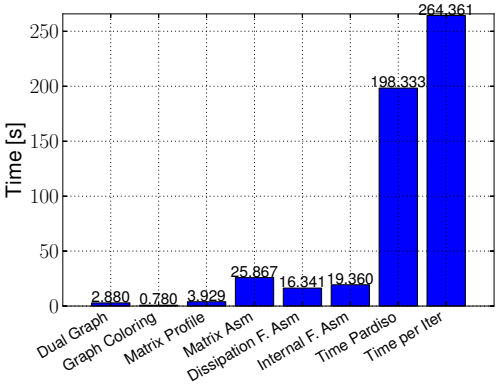
Figura 6: Prueba en Equipo II



(a) Speedup (12.5 million triangular elements)



(b) Efficiency (12.5 million triangular elements)



(c) Times (12.5 million triangular elements)

Figura 7: Prueba en Equipo I

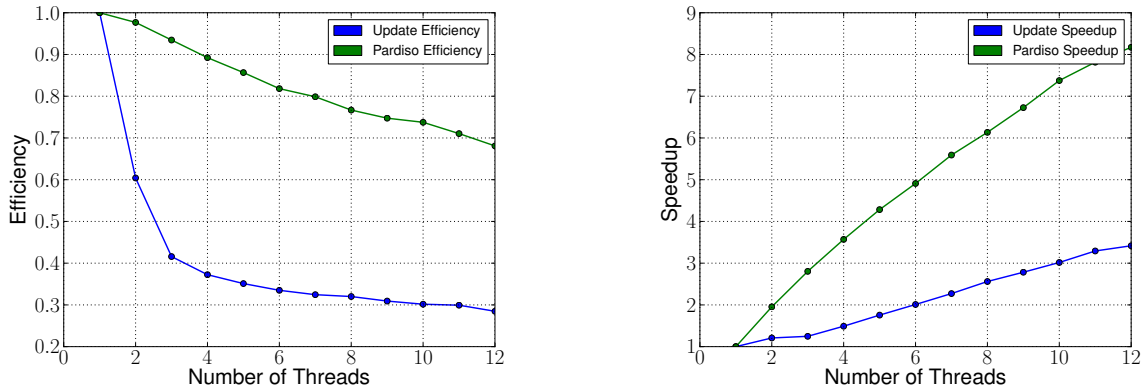
4. TRABAJOS FUTUROS

Existen una variada cantidad de mejoras a realizar sobre el método presentado. Algunas de éstas son:

- Efectuar optimizaciones al algoritmo mediante instrucciones vectoriales. En particular utilizando el nuevo set de instrucciones *Advanced Vector eXtensions* (AVX) que se incorporó en las últimas series de procesadores de AMD e Intel. Para sacar provecho se deberá reformular el algoritmo y las estructuras de datos utilizadas.

- Realizar modificaciones al módulo de almacenamiento en disco de la historia de resultados de forma tal que: permita realizar escrituras cada un lapso de tiempo determinado y no cada paso de tiempo; soporte la escritura asíncrona a disco mediante la biblioteca *Intel's C/C++ aynchronous input/output* (Intel's C/C++ AIO), permitiendo liberar al procesador de la tarea de escritura, permitiendo así una mejor performance.

- Estudiar en detalle el comportamiento de la solución propuesta en arquitecturas NUMA, y detectar posibles mejoras para obtener una escalabilidad razonable.



(a) Efficiency (12.5 million triangular elements) (b) Speedup (12.5 million triangular elements)

Figura 8: Análisis comparativo entre Pardiso y Updates en Equipo I

5. CONCLUSIÓN

En el presente trabajo se presentó un enfoque para la paralelización de un código del MEF de propósito general. El mismo permite aprovechar el poder de cálculo de las computadoras que utilizan arquitecturas SMP y NUMA para acelerar el cálculo del MEF. Esto se hizo logrando mantener las características benéficas de la Orientación a Objetos sobre la que fue diseñado el programa y a su vez alcanzando un buen *speedup* para arquitecturas de memoria compartida con un solo socket. Además se identificó un problema en arquitecturas NUMA, donde el *speedup* alcanzado no se condice con las características técnicas del equipo. Esta situación plantea una nueva línea de investigación destinada a resolver esta cuestión.

REFERENCIAS

Anderheggen E. y Renau-Munoz J. A parallel explicit solver for simulating impact penetration of a three-dimensional body into a solid substrate. *Advances in Engineering Software*, 31:901–911, 2000.

Brown K., Attaway S., Plimpton S., y Hendrickson B. The finite element method: A good friend. *Computer Methods in Applied Mechanics and Engineering*, 184:375–390, 2000.

Cardona A., Klapka I., y Geradin M. Design of a new finite element programming environment. *Engineering Computations*, 11:365–381, 1994.

Costarelli S., Paz R., y Dalcin L. *Resolución de las ecuaciones de Navier-Stokes utilizando CUDA*. 2011.

de Berg M., Cheong O., van Kreveld M., y Overmars M. *Computational Geometry: Algorithms and Applications*, páginas 47–48. Springer, tercera edición, 2008.

Farhat C. y Crivelli. L. A general approach to nonlinear FE computations on shared-memory multiprocessors. *Computer Methods in Applied Mechanics and Engineering*, 72:153–171, 1989.

Karypis G. *METIS, A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices.* 2011.

Kubale M. Graph colorings,. *American Mathematical Society*, 2004.

Kumbhar A., Chakravarthy K., Keshavamurthy R., y Rao G. Utilization of parallel solver libraries to solve structural and fluid problems. *Intel® Math Kernel Library - White Papers*, 2011.

NRC. *The Potential Impact of High-End Capability Computing on Four Illustrative Fields of Science and Engineering*. The National Academies, 2008.

Pantalé O. Parallelization of an object-oriented fem dynamics code: influence of the strategies on the speedup. *Advances in Engineering Software*, 36:361–373, 2005.

Sato M. OPENMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors. *15th International Symposium on System Synthesis*, páginas 109–111, 2002.