Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXX, págs. 1933-1944 (artículo completo)
Oscar Möller, Javier W. Signorelli, Mario A. Storti (Eds.)
Rosario, Argentina, 1-4 Noviembre 2011

IMPLEMENTACIÓN DE UN MÉTODO PARALELO DE TRIANGULACIÓN DELAUNAY EUCLÍDEO

Pablo J. Novara y Nestor A. Calvo

Facultad de Ingeniería y Ciencias Hídricas, Universidad Nacional del Litoral, Ciudad Universitaria, Santa Fe, Argentina, zaskar_84@yahoo.com.ar

Centro Internacional de Métodos Computacionales en Ingeniería - CIMEC (INTEC). Parque Tecnológico Litoral Centro, Santa Fe, Argentina.

Palabras Clave: Mallas, Delaunay, triangulación, paralelización, Delaunay paralelo.

Resumen. En el presente trabajo se muestran los avances en la implementación de un método paralelizable para triángulación Delaunay de un conjunto de puntos. El método se basa en la subdivisión recursiva del dominio mediante un plano/recta y la identificación de todos los elementos Delaunay que intersectan dicho plano/recta. El método se ha implementado tanto con modelos de memoria compartida como con modelos de memoria local y se presentan los analisis de tiempo y complejidad algorítmica de cada uno, compartivamente con respecto al mismo método no paralelizado y con respecto al método tradicional de insersión de nodos en orden aleatorio.

1. INTRODUCCIÓN

En este trabajo se analiza la implementación y el desempeño de un método para generar triangulaciones que cumplen la condición Delaunay a partir de un conjunto desordenado de puntos, en paralelo, tanto para modelos de memoria local como de memoria compartida. De todas las triangulaciones posibles, la triangulación Delaunay es aquella que resulta optima desde cierto punto de vista: maximiza el mínimo ángulo. Esta propiedad resulta adecuada para una gran cantidad de aplicaciones de ingeniería. En este trabajo se analiza el caso 2D (triángulos), pero se mostrará que la extensión al caso 3D (tetraedros, de mayor interés práctico) es directa.

2. MÉTODOS PARA LA GENERACIÓN DE TRIANGULACIONES DELAUNAY

Uno de los métodos secuenciales más utilizados para la construcción de triangulaciones Delaunay consiste en partir de un triángulo suficientemente grande como para abarcar todo el conjunto de nodos (construido mediante 3 nodos virtuales adecuadamente posicionados), e insertar los nodos reales uno por uno. Cuando se inserta un nodo en la triangulación, el triángulo que lo contiene se reemplaza por tres nuevos triángulos más pequeños que tienen al nuevo nodo como vértice común. Cada uno de estos nuevos triángulos es verificado junto con sus triángulos vecinos para detectar si cumplen la condición de Delaunay (esto es, que el vértice opuesto de un triángulo no se encuentre dentro de la circunferencia que forman los otros tres vértices del otro). Si un par de triángulos vecinos no cumple la condición, se realiza un intercambio de diagonales (tomando el conjunto como un cuadrilátero divido por un de sus diagonales) y los dos triángulos modificados vuelven a ser verificados junto con cada uno de sus triángulos vecinos. Se prosigue de esta forma iterativamente hasta que todos los triángulos verificados cumplan la condición y ya no se modifique ninguno. Es por esto que una modificación puede propagarse eventualmente por toda la malla. Esta propagación hace que sea muy difícil en la práctica paralelizar este algoritmo con un modelo de memoria local (el costo de la comunicación entre procesos puede ser mucho mayor al costo de la inserción del nodo y las verificaciones). Sin embargo, dado que en la mayoría de los casos las modificaciones no se propagan más que en uno o dos niveles de vecinos, un modelo de memoria compartida permitiría que varios procesadores inserten nodos en disitintas zonas de la triangulación. Un enfoque posible consiste en realizar un pool de nodos a insertar y un pool de triángulos a verificar y que los distintos hilos de ejecución tomen tareas de estos pools. En todo momento debe asegurarse que dos hilos no tomen tareas que utilicen un mismo triángulos (por ejemplo, con un sistema de flags por elemento). Utilizando técnicas de búsqueda adecuadas (como linear walk) el algoritmo puede generar las triangulaciones en tiempos esperados quasi-lineales. El mayor problema en estos métodos es el uso de nodos virtuales para inicializar el proceso, ya que estos nodos pertenecerán durante la construcción a un gran número de elementos, y al final de la misma deben removerse.

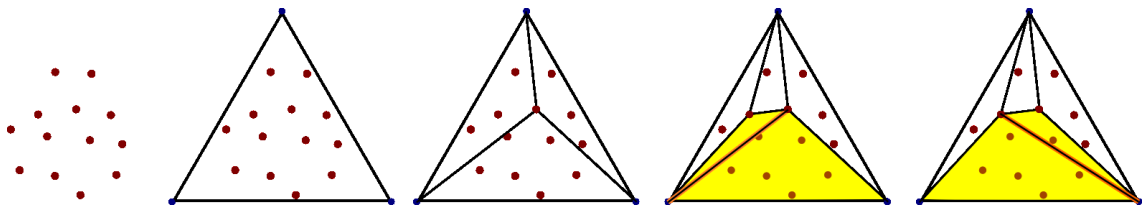

Figura 1: construcción del triángulo inicial con nodos ficticios, inserción de dos nodos y corrección local para recuperar la condición Delaunay luego de la inserción

Para realizar la generación de mallas o triangulación en paralelo, los métodos de tipo divide-and-conquer son los más estudiados. Muchos de los enfoques que se encuentran en la bibliografía buscan dividir el espacio a triangular para generar mallas en cada subespacio por separado y de forma independiente utilizando los algoritmos secuenciales ya establecidos para luego unir todas las sub-mallas conformando el resultado final. Las sub-regiones que se triangulan por separado pueden superponerse parcialmente en algunos métodos, o compartir la frontera en otros. Sin embargo, el proceso en el cual se unen las sub-mallas puede llegar a ser tan costoso como la generación de cada una de ellas, ya que implica modificaciones en los triángulos de las frontera (a veces remallados locales completos), que a su vez pueden disparar modificaciones en sus vecinos recursivamente. Muchos métodos además dependen del ordenamiento de los elementos de frontera, por lo que solo pueden aplicarse para mallas en dos dimensiones. El método que se analiza en este trabajo, denominado DeWall (Cignoni et al., 1998), invierte el orden del proceso: primero genera una pared de triángulos definitivos que cumple la condición Delaunay y divide el espacio en dos regiones inconexas, y luego distribuye los subespacios resultantes en dos procesadores imponiendo como condición respetar la frontera fijada en la división. De esta forma, la pared de triángulos permanecerá fija y las triangulaciones parciales generadas en paralelo se incorporan a la malla final sin modificaciones.

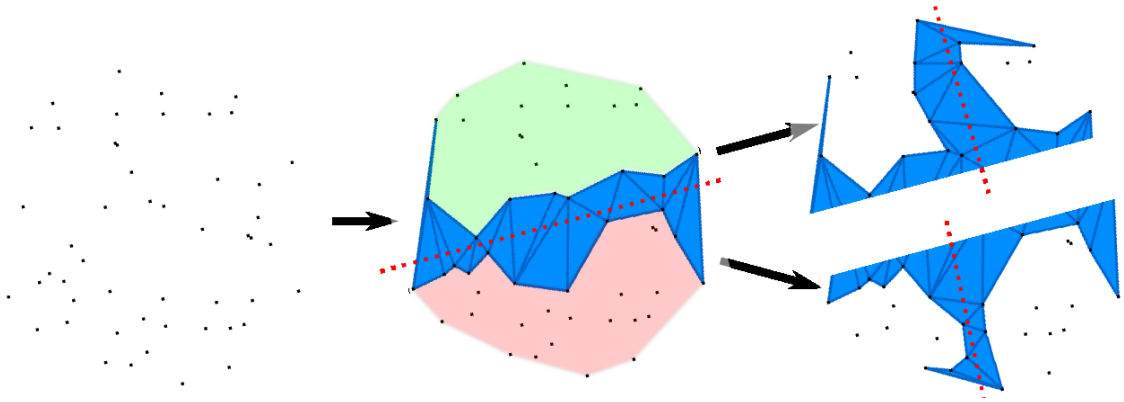

Figura 2: construcción de la primer pared de elementos, que divide el espacio en dos subespacios independientes (verde y rojo), y construcción de las segundas paredes (para cada subespacios), que dividen el problema en cuatro subespacios también independientes

Para construir esta pared se utiliza un método incremental (McLain, 1976). Este método se basa en buscar, dada una arista de la malla, el nodo (de entre los nodos ubicados en el semiplano hacia el cual avanza la triangulación) cuya distancia desde el centro de la circunferencia que forma con la arista, a la arista misma (distancia con signo, calculada a través del producto vectorial) sea mínima.

Planteada de esta forma, la búsqueda implica procesar todos los nodos de la malla por cada triángulo que se construye (verificar en cual semiplano se encuentran y, si son potenciales candidatos, construir la circunferencia y hallar la distancia perpendicular a la arista), lo cual resulta excesivamente costoso en comparación con los métodos utilizados en algoritmo secuenciales. Sin embargo, en la sección 4 se discute un conjunto de optimizaciones y estructuras de datos auxiliares que permiten reducir el orden del tiempo de ejecución. Para el análisis de este trabajo se implementó el proceso en dos dimensiones, pero la extensión al caso tridimensional es directa: se busca el centro de esfera cuya distancia a una cara (triángulo) dada sea mínima.

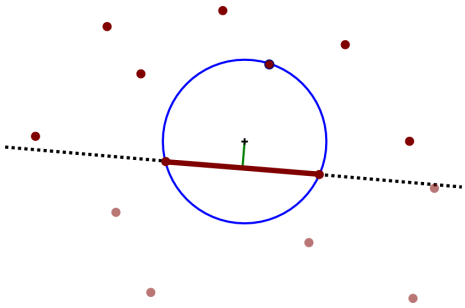

Figura 3: Determinación del nodo adecuado para construir el próximo triángulo. Los nodos inválidos se presentan semitransparentes. Para el nodo seleccionado se muestra el la circunferencia y su centro en relación a la arista

3. IMPLEMENTACIÓN BÁSICA DEL ALGORITMO NO-PARALELO

La estructura de datos más simple para almacenar una malla de triangulos consta de un contenedor de nodos y un contenedor de triángulos. Cada nodo esta compuesto por sus coordenadas (x e y), y cada triangulo por un conjunto ordenado de tres indices que corresponden a los nodos que lo definen. Para almacenar los elementos se utilizó un contenedor de tipo lista enlazada, ya que el algoritmo sólo insertará elementos. Para almacenar los nodos se utiliza un vector, ya que se requiere acceso aleatorio, y la cantidad de nodos no varía a lo largo del proceso.

El algoritmo propuesto requiere de una arista inicial para comenzar. Encontrar una arista cualquiera perteneciente a la triangulación del envoltorio convexo de la misma no es una tarea complicada, pero en las sucesivas subdivisiones se deberán encontrar aristas completamente contenidas en zonas no convexas, y en estos casos la complejidad es mayor si no se tiene la frontera definida. Además, para que las particiones sean optimas, el algoritmo debe ser capaz de encontrar una arista en una zona en particular (alguna arista atravesada por el plano medio). Por esta razón se agregó en la estructura de datos un conjunto de aristas que representan la frontera de la zona a triangular, la cual es inicialmente equivalente al envoltorio convexo de la misma, y se construye en la etapa de preparación utilizando el algoritmo de Graham, ordenando los puntos con el método quick-sort.

```
struct Malla {
    // nodos, triangulos y frontera
    vector<Nodo> n;
    vector<Elemento> e;
    vector<Arista> a;
};

struct Nodo {
    // coordenadas
    double x,y;
};
```

```
struct Arista {
    // indices de dos nodos
    int n1,n2;
};

struct Elemento {
    // indices de tres nodos
    int n[3];
};
```

Figura 4: Estructura básica para el almacenamiento de la malla y su frontera

La manera más sencilla de realizar la primer pared de triángulos que separa el conjunto de puntos en dos es la siguiente:

1. Construir el plano divisor
2. Recorrer el conjunto de aristas buscando una arista que tenga un punto a cada lado del plano divisor.

3. Mientras haya al menos un nodo del lado de la arista hacia el cual avanza la triangulación:

   a) Buscar el nodo adecuado entre los posibles y agregar un triangulo formado por dicho nodo y la arista.

   b) Elegir una de las dos aristas nuevas (del triangulo construido en el paso 3) para continuar el mismo proceso desde el paso 2. La arista que se elige es la atravesada por el plano divisor.

En la implementación analizada, el plano divisor solo puede ser vertical u horizontal, dependiendo de la relación de aspecto del bounding-box del conjunto de nodos, y se ubica de forma que divide en dos partes iguales al mismo (sera la recta $x = sum(x_i)/N$, o $y = sum(y_i)/N$).

Para la busqueda del paso 3.1, la implementación más directa consiste en: Por cada nodo del conjunto (a excepción de los dos de la arista en cuestión):

1. Comprobar utilizando el signo del producto vectorial de la arista y un vector (desde un nodo de la arista al nodo que se prueba) si el nodo esta del lado adecuado de la misma. En caso negativo, se descarta el nodo.

2. Encontrar el centro de la circunferencia que definen los tres puntos, y medir su distancia a la arista (utilizando el producto vectorial entre la misma y el vector que forman el nodo a comprobar y un nodo de la arista).

3. Si la distancia calculada (del centro a la arista) es la menor encontrada hasta el momento, guardar el indice del nodo.

Este algoritmo puede entrar en un bucle infinito cuando encuentra tres nodos colineales, ya que con dos de ellos se define una arista, y la búsqueda da por resultado el tercero. El algoritmo del cálculo del centro de la circunferencia por tres puntos debería detectar este caso cuando el determinante del sistema que debe resolver es cero, pero el problema puede ocurrir por errores numéricos. Para evitar que el algoritmo cicle entre estos tres nodos generando infinitos triángulos superpuestos, si se detecta el caso (cuando la distancia mínima del centro a la arista es 0), se puede aplicar a la posición del nodo una pequeña perturbación, o modificar el algoritmo para ignorar los nodos colineales a la arista sobre la que se avanza en cada paso.

Una vez dividido el conjunto de nodos en dos, se debería repetir el proceso para cada subconjunto por separado. Sin embargo, generar dos nuevas estructuras de datos para separar la malla en dos nuevas sub-mallas puede implicar un costo demasiado alto tanto en tiempo como en uso de memoria. Por esto, cada nodo y cada arista contiene una etiqueta, que se utiliza para identificar a que subconjunto pertenece. La malla contiene un atributo de tipo entero que guarda la ultima etiqueta generada. Cuando se requiere una nueva etiqueta, se incrementa en uno el valor de este atributo. El algoritmo general para realizar una división se modifica como sigue:

1. Construir el plano divisor

2. Obtener dos nuevas etiquetas $n + 1$ y $n + 2$.

3. Recorrer el conjunto de aristas con etiqueta igual a n buscando una arista que tenga un punto a cada lado del plano separador.

4. Mientras haya nodos candidatos para formar un nuevo triángulo

a) Buscar el nodo adecuado entre el conjunto de nodos con etiqueta igual a n para construir un triangulo con dicha arista.

b) Elegir una de las dos aristas nuevas (del triangulo agregado) para continuar el mismo proceso. La arista que se elige es la que es atravesada por el plano medio.

c) Etiquetar la arista descartada en el paso 4.2 y agregarla en el contenedor de aristas.

5. Recorrer el conjunto de nodos con n por etiqueta y re-etiquetarlos de acuerdo a su posición en relación al plano medio.

Se utilizó una pila de trabajos para colocar allí las etiquetas de los subespacios pendientes de triangular. En cada división, si los subespacios generados contienen tres o más nodos sus etiquetas se ingresan en la pila. La estructura es de tipo pila, ya que los primeros trabajos ingresados tienen un gran numero de nodos y los últimos son los que corresponden a subespacios más pequeños o incluso triángulos individuales. Como cada trabajo puede generar dos nuevos trabajos, una cola crecería exponencialmente antes de llegar a los trabajos que ya no generan nuevas etiquetas, llevando a que, en mallas grandes, el consumo de memoria de la pila de trabajos sea muy superior al consumo de memoria de la malla misma (nodos, aristas y triángulos).

```
struct Malla {
    qeque<int> trabajos
    vector<Nodo> n;
    vector<Elemento> e;
    vector<Arista> a;
};

struct Nodo {
    int etiqueta;
    double x,y;
};

struct Arista {
    int etiqueta;
    int n1,n2;
};

struct Elemento {
    int n[3];
};
```

Figura 5: Estructura modificada para facilitar el etiquetados de nodos y aristas, y agregando además una pila de trabajos pendientes

4. OPTIMIZACIONES PARA EL ALGORITMO NO-PARALELO

Hay dos grandes problemas en el enfoque descripto en la sección anterior:

1. Para generar un triangulo en un espacio determinado, se deben analizar todos los nodos del mismo, por lo que el tiempo total para generar la triangulación resulta O(mxn) donde m y n son la cantidad de elementos y nodos respectivamente.

2. A medida que el proceso avanza los subespacios a triangular se hacen más pequeños. Sin embargo, los arreglos de nodos y aristas continúan conteniendo los nodos y aristas de todos los espacios. Para encontrar la arista inicial para una división determinada, se deben recorrer todas las aristas y verificar sus etiquetas. Lo mismo sucede para el conjunto de nodos. Entonces, aunque el espacio y las cantidades de nodos y aristas a considerar se reducen, la implementación propuesta obliga a seguir buscando en toda la malla.

Para solucionar el primer problema se utiliza una grilla regular (nxn celdas) como estructura auxiliar. En la etapa de preparación se asigna cada nodo a una celda de la grilla. Encontrar a qué celda corresponde un punto dado es directo en una grilla regular, razón por la cual se prefiere

este tipo de estructura frente a otras más complejas como árboles. En el proceso de construcción de un triángulo se deben recorrer los nodos. Si en un momento del recorrido se tiene un nodo en particular (in) como nodo más cercano hasta el momento, y la circunferencia que definía este nodo junto con los de la arista tiene radio r, se puede demostrar que cualquier nodo cuya distancia sea mayor a 2r no puede generar una circunferencia con centro más cercano que la generada por el nodo in. Utilizando esta propiedad se pueden descartar celdas completas sin analizar sus nodos ni calcular las circunferencias. Para que el descarte sea temprano, conviene comenzar la búsqueda por los nodos más cercanos a la arista. Dado que la grilla es regular, es directo determinar la celda donde se ubicaría el punto medio de la arista. A partir de la misma se pueden recorrer fácilmente las celdas restantes en orden creciente de distancia (medida según la norma l_∞). En la gran mayoría de los casos, no es necesario recorrer más de nueve celdas para determinar el triángulo que cumple la propiedad Delaunay. De esta forma, el algoritmo que originalmente presentaba un orden cuadrático se convierte en un algoritmo con tiempo esperado lineal. La cantidad de celdas utilizadas se determina en forma aproximada de acuerdo a la densidad de puntos, con una función ajustada experimentalmente.

Para evitar recorrer la lista completa de aristas, el contenedor de aristas se removió de la estructura de la malla y se colocó en la estructura que representa un trabajo. De esta forma, cada trabajo tiene su lista de aristas y las búsquedas se limitan a la misma. El contenedor utilizado para ello es de tipo lista enlazada, ya que se pueden mover aristas de una lista a otra simplemente cambiando los enlaces, reduciendo así el número acciones de reserva y liberación de memoria dinámica.

Para evitar recorrer el conjunto completo de nodos, se utiliza un vector auxiliar de punteros. El vector auxiliar contiene punteros a todos los nodos de la malla. Luego de la primer división, se reordena de forma que todos los nodos del primer subespacio generado se ubiquen en la primer mitad y todos los nodos del segundo subespacio en la segunda mitad del vector. Agregando los índices del primer y último puntero a nodo de un subespacio a la estructura que representa un trabajo, se puede disponer de un mecanismo para recorrer solo los nodos del subespacio. El reordenamiento se realiza en el vector auxiliar de punteros y no en el vector de nodos porque otras estructuras como aristas, elementos, o celdas de la grilla contienen referencias a estos nodos por su posición en el vector. Sin embargo, recorrer este fragmento de vector no es compatible con utilizar una grilla uniforme para agilizar la búsqueda, ya que la grilla es única y su actualización en cada división sería costosa. Para esta búsqueda, si el número de nodos del espacio es comparable a la cantidad de nodos por celda de la grilla, se utiliza el vector auxiliar, en caso contrario se utiliza la grilla uniforme y los nodos se filtran según su etiqueta.

4.1. Resultados

La tabla 1 muestra los tiempos medidos para las dos etapas del proceso: preparación (construcción de convex-hull y grilla) y mallado propiamente dicho, para diferentes cantidades de nodos.

Se observa que el tiempo requerido por la etapa de mallado presenta un orden sub-cuadrático en relación al número de nodos, confirmando así la eficacia de las optimizaciones presentadas.

5. IMPLEMENTACIÓN DEL ALGORITMO EN PARALELO

Se realizaron dos paralelizaciones para este algoritmo: utilizando un modelo de memoria local y utilizando un modelo de memoria compartida. La etapa paralelizada es la etapa de mallado por ser la que claramente consume la mayor parte del tiempo total de ejecución (el tiempo

```
struct job_info {
    // limites de la parte del arreglo de punteros
    // auxiliares que corresponde al trabajo
    int n_ini, n_fin;
    // aristas de la frontera de ese
    // subconjunto de nodos
    List<Arista> *a;
};

struct Malla {
    // vectores de enteros que contienen
    // los nodos de cada celda de la grilla
    vector<int> *celdas;
    // cola de trabajos
    deque<Job> trabajos;
    // nodos, y punteros auxiliares
    vector<Nodo> n;
    vector<Nodo*> p;
    // triangulos
    vector<Elemento> e;
};
```

```
struct Nodo {
    int etiqueta;
    double x,y;
};

struct Arista {
    int n1,n2;
};

struct Elemento {
    int n[3];
};
```

Figura 6: Estructura de dato final para contemplar las optimizaciones presentadas

Nodos	T prep	T mesh
1.00E+004	0.0052	0.0124
5.00E+004	0.0136	0.0656
1.00E+005	0.0276	0.1344
5.00E+005	0.1490	1.1628
1.00E+006	0.3068	9.3100
2.00E+006	0.6279	97.2342

Tabla 1: Tiempos de preparación y triangulación para el proceso secuencial

necesario para la etapa de preparación para una malla de 1e7 nodos no supera los 5 segundos).

5.1. Memoria compartida

Dadas las características del método de triangulación seleccionado, se realizó una implementación para memoria compartida utilizando phtreads, donde varios hilos de ejecución trabajan en paralelo en los distintos subdominios de forma similar al caso de memoria local. En un primer análisis se puede pensar que esta implementación resulta aún más adecuada desde el punto de vista de la eficiencia y el speed-up resultante, ya que los datos de nodos y aristas en memoria se encuentran perfectamente separados y no es necesario realizar copias ni tomar recaudos para evitar que dos hilos modifiquen el mismo sector de la memoria. Sin embargo, la lista de elementos sí es única, y la pila/cola de tareas también. Es por esto que se debe recurrir al uso de mutexs para controlar el acceso a estos dos contenedores y evitar asi las potenciales race-conditions.

Fue necesario utilizar tres mutexes para controlar la paralelización:

1. Para evitar que dos hilos asignen la misma etiqueta a sus subespacios. Cuando un hilo necesita una etiqueta bloquea a los demás mientras incrementa el contador general de la malla que las genera.

2. Para gestionar la pila de trabajos y el contador de hilos ocupados. Cuando un hilo consulta

si hay trabajos pendientes y, eventualmente extrae un trabajo de la cola, este mutex se utiliza para prevenir a los demás hilos de extraer el mismo trabajo, o modificar la cola al intentar encolar trabajos nuevos. Este es el que más bloqueos genera durante la ejecución.

3. Para gestionar la inserción de elementos en la malla. Si bien los hilos pueden construir los elementos en paralelo, solo pueden insertarlos en el contenedor común en forma secuencial. Cabe aclarar que se compararon los tiempos de una implementación que intenta agregar cada elemento a medida que lo construye y otra que los guarda en un contenedor local del hilo hasta terminar de particionar su espacio y luego los intenta agregar juntos bloqueando una sola vez este mutex y los tiempos resultantes fueron casi idénticos.

5.1.1. Resultados

Las pruebas se realizaron en un procesador Intel I7 con cuatro núcleos reales que se exponen como ocho al sistema operativo (GNU/Linux de 64bits) gracias a la tecnología HyperThreading. La tabla 2 muestra los tiempos medidos para distintas cantidades de nodos y de hilos simultáneos.

Nodos	1 hilo	2 hilos			4 hilos			8 hilos		
	t	t	su	ef	t	su	ef	t	su	ef
1E+4	0.0260	0.0205	1.27	63.41	0.0224	1.16	29.02	0.0130	2.00	25.00
5E+4	0.0743	0.0592	1.26	62.75	0.0561	1.32	33.11	0.0660	1.13	14.07
1E+5	0.1463	0.1170	1.25	62.52	0.1115	1.31	32.80	0.1590	0.92	11.50
5E+5	1.1733	0.7682	1.53	76.37	0.6059	1.94	48.41	0.6889	1.70	21.29
1E+6	9.2637	5.2072	1.78	88.95	3.3901	2.73	68.31	2.7660	3.35	41.86
2E+6	96.1801	55.0953	1.74	87.28	37.0654	2.59	64.87	28.7345	3.34	41.83

Tabla 2: Tiempo, speed-up y eficiencia de la implementación de memoria compartida para distintas configuraciones

Se observa que el tiempo del algoritmo escala correctamente para mallas grandes en relación a la cantidad de hilos, ya que los tiempos descienden notablemente cuando esta aumenta. La diferencia es más acentuada cuando se utilizan núcleos reales, aunque la utilización de núcleos virtuales (HyperThreading) también presenta una mejora en los tiempos totales. Sin embargo, aunque los tiempo disminuyen a medida que la cantidad de hilos aumenta, la eficiencia paralela decae. Observando el tamaño de la malla, se aprecia que la eficiencia aumenta conforme el número de nodos crece. En comparación con la implementación no paralela, el programa que utiliza pthreads arroja tiempos similares cuando utiliza un solo hilo, por lo que el uso de mecanismos de sincronización en la gestión de trabajos pendientes no introduce una gran sobrecarga en el proceso.

5.2. Memoria local

Para la implementación basada en procesadores con memoria local se utilizó mpich2. El programa utiliza el primer proceso (maestro) para coordinar a los demas (esclavos) y recibir los resultados, pero no para construir elementos. Cada esclavo tendrá una cola propia de trabajos pendientes y una versión reducida de la malla que solo incluye aristas y nodos del trabajo asignado.

El esquema de trabajo del proceso maestro es el siguiente:

1. Mientras al menos un esclavo se encuentre ocupado, esperar mensaje de cualquier esclavo
    a) Si un esclavo indica que termino todas sus tareas, recibir los elementos generados y marcar dicho proceso como disponible
    b) Si el esclavo (e1) indica que tiene tareas para distribuir
        1) Si hay otro proceso esclavo disponible (e2), enviar el id de e1 a e2, y el de e2 a e1, y marcar e2 como ocupado
        2) Si no hay otro proceso esclavo disponible, informar a e1
2. Si todos los esclavos están disponibles, informar a todos que la triangulación esta completa

El esquema de trabajo del proceso esclavo es el siguiente:

1. Esperar a que el maestro envíe el id de un proceso que tiene tareas pendientes
2. Recibir los nodos y aristas del trabajo a realizar
3. Mientras queden trabajos pendientes
    a) Si hay más de un trabajo en la cola propia, informar al maestro
        1) Si el proceso maestro envía el id de un esclavo disponible, enviar los datos del trabajo
    b) Resolver un trabajo
4. Indicar al maestro la finalización
5. Enviar al maestro los elementos generados
6. Volver al paso 1

El proceso de control guarda información del estado (ocupado o disponible) de los demás procesos. Cuando un proceso esclavo tiene más de una tarea pendiente informa al proceso maestro. Este se encarga de responder indicando qué otro proceso se encuentra disponible, o informando si todos están ocupados. En este último caso el proceso esclavo resuelve una de las tareas y luego vuelve a consultar al proceso maestro. Cuando el maestro indica a dos procesos que intercambien tareas, estos se comunican entre sí sin necesidad de supervisión por parte del proceso maestro. Cuando el proceso esclavo termina o distribuye todas sus tareas informa al proceso maestro. Este, antes de marcarlo como proceso disponible recibe los elementos generados por las tareas anteriores de dicho esclavo. Cuando el programa comienza, el primer esclavo inicia el proceso de división. Cuando ningún esclavo tiene tareas pendientes, significa que la triangulación está concluida y la malla final de elementos se encuentra en el proceso maestro.

Se debe notar que mientras se realiza la primer división solo un proceso se encuentra ocupado. Luego de la primer división, dos proceso pueden trabajar en simultáneo. Luego cuatro, y así sucesivamente, por lo que todos los procesos disponibles se ocupan rápidamente. Para aprovechar mejor los tiempos de calculo es conveniente que los procesos se envíen subespacios lo más grandes posibles, de forma que los tiempos de cálculo predominen sobre los tiempos de comunicación (es preferible realizar un envío grande frente a muchos pequeños). Sobre el final del proceso completo, sin embargo, es preferible que reste resolver las tareas más pequeñas, de

forma que el maestro pueda distribuir la carga en varios procesadores. Para lograr este balance, los contenedores de tareas pendientes de cada proceso son colas dobles. Es decir, que permiten extraer los primeros trabajos ingresados, o los más recientes según convenga. La decisión se realiza comparando la cantidad de procesadores con la cantidad de trabajos pendientes.

5.2.1. Resultados

La tabla 3 los tiempos medidos para la etapa de mallado para distintas cantidades de nodos. Las pruebas con 2, 3, 4, 5, y 8 hilos se realizaron en una PC con un procesador con 4 nucleos reales (8 virtuales). Esto implica que en esas pruebas la comunicación de datos de un proceso a otro se realiza a través de la memoria, sin utilizar una red. Por esto, el ancho de banda será el del bus de datos de la PC (muy superior a cualquier red externa) y la latencia nula. Las pruebas con 6 hilos se indican como 2+4, ya que se utilizaron 2 PCs, una con 4 nucleos reales y otra con 2, ambas de similares características, conectadas directamente a través de un cable ethernet de Gigabit.

Nodos	2 hilos 3	hilos	4 hilos	5 hilos	8 hilos	2+4 hilos
1.00E+005	0.2448	0.1528	0.1062	0.1054	0.1794	0.5444
1.00E+006	10.0631	4.2230	2.8732	2.8696	3.1217	11.0618
2.00E+006	100.6870	25.8303	14.9040	10.6035	10.8452	50.0330

Tabla 3: Tiempos medidos para la implementación de memoria local para distintas configuraciones

La tabla 4 muestra el speed-up y la eficiencia tomando como referencia los tiempos de la implementación no paralela.

Nodos	2 hilos		3 hilos		4 hilos		5 hilos		8 hilos		2+4 hilos	
	su	ef	su	ef	su	ef	su	ef	su	ef	su	ef
1E+5	0.55	27.45	0.88	29.32	1.27	31.64	1.28	25.50	0.75	9.36	0.25	4.11
1E+6	0.93	46.26	2.20	73.49	3.24	81.01	3.24	64.89	2.98	37.28	0.84	14.03
2E+6	0.97	48.29	3.76	125.48	6.52	163.10	9.17	183.40	8.97	112.07	1.94	32.39

Tabla 4: Speed-up y eficiencia de la implementación de memoria local para distintas configuraciones

Cuando se utilizan solo dos hilos, el speedup es menor a 1, ya que solo uno de los hilos procesa los nodos (esclavo) mientras que el otro se encarga solamente de tareas de control y sincronización (maestro). Como era esperable, esta sobrecarga es mayor que en el caso de memoria local. Sin embargo, cuando se utilizan más de 2 hilos en una misma PC la eficiencia de la paralelización supera el 100 % en algunos casos. Esto se debe a que los esclavos no reciben toda la malla, sino solo la fracción que les corresponde procesar, por lo que trabajan con una menor cantidad de datos. Esto indica que es conveniente duplicar los datos a pesar de la sobrecarga que agrega dicha copia (y la reconstrucción de las grillas), ya que la ganancia que se obtiene al trabajar con conjuntos más pequeños y localizados es mayor que la pérdida por dicha sobrecarga.

Cuando los procesos se ejecutan en diferentes PCs los tiempos de comunicación de nodos y elementos se elevan considerablemente, superando en mallas con menos de 2e7 nodos los tiempos del algoritmo no paralelizado. Conforme aumenta el tamaño de malla el speed-up y la

eficiencia también aumentan, pero sin llegar a competir con los tiempos medidos al correr el mismo programa en una sola PC.

6. CONCLUSIONES Y TRABAJOS FUTUROS

Se analizó un algoritmo conocido y poco utilizado para la generación de la triangulación de Delaunay a partir de un conjunto de puntos y se implementaron con éxito diferentes optimizaciones para lograr que dicho algoritmo sea competitivo (considerando complejidad y tiempos de ejecución) frente a otras alternativas utilizadas frecuentemente para esta tareas.

Se implementaron dos versiones paralelas de dicho algoritmo utilizando diferentes modelos de paralelización. Analizando globalmente los resultados de todos las implementaciones se concluye que la utilización del modelo de memoria local en una única pc con más de dos procesador presenta los mayores beneficios, logrando una eficiencia superior al 100 % para mallas de más de 1e7 nodos.

Actualmente se está trabajando en la modificación del algoritmo para ser utilizado como generador de mallas en lugar de triangulaciones. Para la generación de malla se debe tener en cuenta que la frontera será impuesta y puede no ser convexa y/o contener aristas que no pertenecerían a la triangulación Delaunay del conjunto de puntos. Esto lleva a que la triangulación generada no cumpla completamente con los criterios Delaunay. La forma en que se resuelven estos casos debe ser unívoca para que todos los hilos asuman el mismo resultado, pues en caso contrario distintos hilos podrían generar mallas superpuestas o con fronteras incompatibles entre sí. La identificación de ciertos elementos de frontera y su ordenamiento, necesario para resolver algunos de estos, casos podría dificultar la extensión a 3 dimensiones.

REFERENCIAS

Cignoni P., Montani C., y Scopigno R. Dewall: A fast divide and conquer delaunay triangulation algorithm in ed. *Computer-Aided Design*, 30(5):333–341, 1998.

McLain D.H. Two dimensional interpolation from random data. *Comput. J.*, 19(2):178–181, 1976.