Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXIX, págs. 5999-6012 (artículo completo)
Eduardo Dvorkin, Marcela Goldschmit, Mario Storti (Eds.)
Buenos Aires, Argentina, 15-18 Noviembre 2010

REMALLADO EN PARALELO UTILIZANDO UN ESQUEMA DE PARTICIÓN IMPLÍCITA DE DATOS

Juan P. D'Amato[a,b] ,Pablo Lotito[a,b], Marcelo Vénere[a]

[a]*Grupo de Visualización y simulación, Instituto PLADEMA, Universidad Nacional del Centro, 7000 Tandil, Argentina,*

[b] *Consejo Nacional de Investigaciones Científicas y Técnicas CONICET, Argentina*
({jpdamato,plotito,venerem }@exa.unicen.edu.ar)

Keywords: Paralelización, Remallado, GPU.

Abstract. Uno de los métodos más efectivos para la generación de mallas de elementos finitos a partir de descripciones geométricas automáticas (scanners, segmentación de imágenes digitales, etc.) se basa en modificaciones locales para optimizar calidad. El principal problema de este algoritmo es que requiere de muchísimas operaciones hasta converger a una malla satisfactoria, por lo que en general se debe interrumpir el proceso luego de un determinado número de mejoras. En este trabajo se presenta un nuevo esquema, que denominamos "partición implícita", para la paralelización masiva del mismo. La idea es procesar sobre GPUs con grados de paralelismo de cientos de *threads* simultáneos. Se presentan los resultados de "speed-up" obtenidos, a la vez que se introduce a un nuevo esquema de procesamiento simultáneo de los datos espaciales.

1 INTRODUCCIÓN

Con el tiempo, las simulaciones y procesamientos basados en representaciones discretas se han vuelto más grandes y complejas, teniendo como única alternativa para resolverlos dividir las mallas para que sean procesadas en forma distribuida. Muchos de los algoritmos hasta hoy propuestos (Kopyssov & Novikov 2002), (Eppstein et al 2008) estiman las particiones óptimas de elementos, denominadas *Patchs* utilizando principios de conectividad de grafos. Pero la asignación del procesamiento también tiene en cuenta las demoras de transferencia, la aplicación de estrategias de *locking* de datos y las diferencias de capacidad de cada procesador. Esta metodología puede aplicarse en el procesamiento local cuando se cuentan con múltiples unidades de cálculo utilizando un modelo de memoria compartida, pero no es la única alternativa.

Por otra parte, una técnica muy aceptada de generación de mallas aptas para el cálculo científico es la de remallado mediante cambios locales adoptada desde (Hoppe et al 1993), (Garland and Heckbert 1998) hasta (Wang et al 2006) entre otros. Dicha estrategia permite generar discretizaciones con fuertes diferencias de densificación garantizando buena calidad de elementos, lo que la hace apta para su utilización en un contexto adaptivo.

En este trabajo evaluaremos una variante del método de remallado que ejecuta en arquitecturas en paralelo. Las mallas son previamente preparadas, asignando los elementos en múltiples *threads* a partir de información espacial y de conectividad de cada uno, estrategia que denominaremos de *partición implícita*. Dicho método evita la generación de particiones y el uso de estrategias de locks, similar a los algoritmos *lock-free* en sistemas de tiempo real (Fich et al 2004) garantizando el acceso simultáneo a los datos al administrar temporalmente los procesamientos. Dicha estrategia está concebida principalmente para ser ejecutada en PCs con gran cantidad de unidades de *procesamiento* como son las GPUs.

En las próximas secciones se explicará en detalle el método de remallado utilizado y la metodología ideada de paralelización, y se propone una meta-heurística de procesamiento, sobre la cual sea posible variar el método de asignación hasta elegir una secuencia adecuada. Para seleccionar dicha estrategia, se realiza una evaluación cuantitativa de paralelización de distintas mallas y se escoge un esquema de distribución apropiado.

2 ANTECEDENTES

Las estrategias más conocidas para generar particiones de mallas con conectividad mínima se denominan *geométricas*, entre las cuales se puede nombrar "circle bisection" o "coordinate bisection" y variaciones de las mismas (Moulitsas 2002). En este tipo de estrategias, los elementos de la malla son distribuidos por sus coordenadas en el espacio buscando minimizar los vínculos entre grupos de elementos. Otra técnica posible de partición, se basa en la discriminación de los

elementos de acuerdo a otras características geométricas, como puede ser la tasa de cambio de superficie o curvatura definida entre elementos contiguos. Para este caso, es posible utilizar una variante de "k-means" generando grupos de triángulos de acuerdo al plano normal de cada elemento. Esta solución fue evaluada y descartada, a pesar de ser eficiente, en (Cifuentes et al 2010) ya que en mallas muy irregulares generaba demasiadas particiones desasociadas.

Una vez que se obtuvieron los *patchs* mediante alguno de los métodos nombrados, es posible iniciar el procesamiento simultáneo de cada uno, tratando de forma diferencial los elementos compartidos (vértices, aristas). Estos elementos comunes definen la frontera de cada *patch* la cual debe ser omitida del tratamiento inicial o se debe aplicar algún mecanismo de control o "lock inteligente" de escritura/lectura (Hudson et al 2007).

Este esquema de procesamiento requiere una ejecución iterativa hasta abarcar todos los elementos de la malla y etapas intermedias de unificación de los resultados. Dichos métodos de partición convencionales de mallas son concebidos para distribución en *grids* o redes de computadores, generando particiones físicas independientemente del proceso que luego sufrirá. Hasta ahora el esquema ha resultado efectivo, pero es sabido que la sincronización y actualización de las partes compartidas de la malla, demandan un tiempo de transferencia de los datos que puede llegar a ser elevado e incluso superior al de procesamiento.

2.1 Remallado mediante operadores locales

Para transformar una malla en una que se ajuste a los requerimientos del método de elementos finitos, se utiliza en este trabajo una estrategia basada en tres tipos de modificaciones locales (Wang et al 2006):

1. Inserción de nodos dividiendo las aristas de elementos con tamaño mayor que el especificado h(x, y, z), denominado O_{Split}.

2. Colapso de las aristas con tamaño mucho menor que el especificado. Se remueven en esta forma dos nodos y dos elementos y se agrega un nuevo nodo, $O_{Collapse}$.

3. Cambio de diagonales entre triángulos vecinos en los casos que resulta posible y conveniente, O_{Swap}.

Esta técnica suele denominarse de Optimización de la conectividad, e involucra cambios estructurales tendientes a una mejora parcial de la malla. Esta técnica de 3 operaciones se encuentra acotada a elementos del tipo triangulares y produce en general mallas no-estructuradas. Existen variantes más inteligentes de estos operadores que aceleran la convergencia a una malla de buena calidad (Suarez et al 2007).

Los elementos tienen asociada una métrica de calidad normalizada, que indica que tan "*buenos*" son los elementos. La métrica utilizada es la presentada en (Correa Reina et al 2006) (eq.1), siendo máxima cuando los elementos triangulares son equiláteros y tienen el tamaño apropiado (función de tamaño normalizado Q*h*), y mínima para

elementos deformes o de tamaño inadecuado; los cuales tendrán prioridad para ser procesados.

$$Q = min\left(Q_h(T),\ 1/Q_h(T)\right)\frac{A_T}{\sum_{i=1}^{3}\left(L_T^i\right)^2} \qquad \text{(eq.1)}$$

La fortaleza que se pretende explotar de dicho método es la propiedad de localidad, un cambio solo afectará a ciertos vecinos de un elemento. Esto hace suponer que dos elementos que se encuentran lo suficientemente distantes no se verán afectados si son modificados de forma simultánea. Como ejemplo, en la Figura 1 se indican los elementos triangulares a ser procesados porque no cumplen un criterio de tamaño y calidad deseados. Se distingue que los elementos generalmente no son contiguos, pero es necesario asegurar esta condición para preservar la coherencia de la malla.

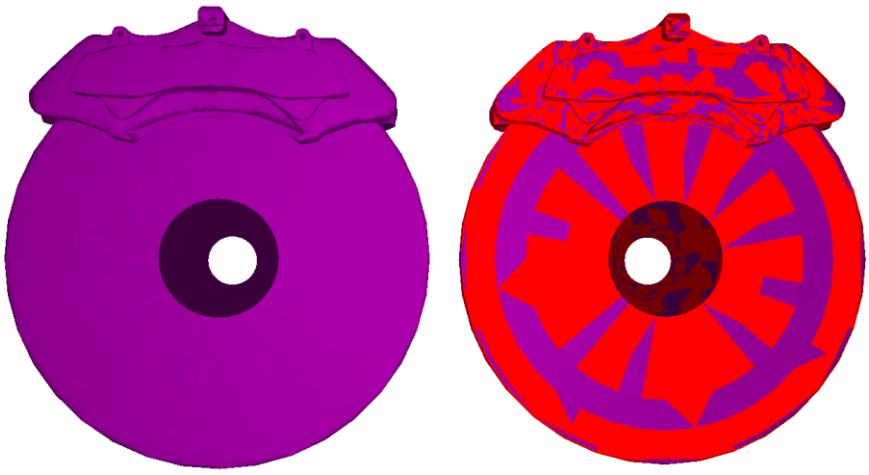

Figura 1 (izq.) malla original de un dispositivo de freno (der.) Elementos seleccionados (y sus vecinos) a mejorar

Utilizando este principio y con el objetivo de mejorar la eficiencia del método de remallado a partir del uso de hardware embebido como la GPU, se ideó un esquema que permita "libremente" modificar cualquier elemento de la malla de forma simultánea, sin generar agrupamientos de elementos o mecanismos de bloqueo. Este nuevo mecanismo pretende distribuir entre muchos procesadores o *threads* locales los elementos a ser procesados considerando las coordenadas espaciales de los elementos, la distancia entre los mismos así como también el tiempo en que se realizará la operación asegurando la integridad de los cambios.

3 PARALELIZACIÓN DE OPERADORES LOCALES

El objetivo es encontrar un mecanismo eficiente que permita modificar un cierto conjunto de elementos de forma simultánea, sin bloqueo de datos. Como se nombró

anteriormente, cada operación típica que modifica la estructura de la malla, es un *operador de elemento*. En este trabajo, los operadores usuales se denominan O_{Swap} , O_{Split}, O_{Remove} , y se reconocen otras variantes como O_{NSwap},(swap con n-niveles de prueba), O_{SSE} (división simultánea de las 3 aristas de un triángulo), O_{SS}(remover los 3 vértices de un triángulo) o incluso una combinación de operaciones O_{SWSR}, para producir mejores elementos.

Se definen ahora ciertas características de los operadores y de los elementos involucrados a fin definir el método de asignación.

Definición 1: El *alcance A* de un *operador O* de elementos en una *malla M* es el conjunto de elementos afectados por este, definido como A(O(E)) ={E1,..Ek ∈ M }. En muchos casos, el alcance del operador sobre un triángulo *E* son todos los triángulos que comparten vértices o aristas con E.

Definición 2: el alcance de varios elementos es *expansivo*, es decir, se cumple la condición que $A_O(E) \cup A_O(E') \geq A_O(E)$ con $E' \neq E$

Definición 3: Un operador O(E) es independiente de O(E'), si $A(O(E)) \cap A(O(E')) = \varnothing$ para cualquier $E, E' \in M$

Definición 4: un elemento E' es vecino de orden 1 de E, cuando $E' \in A_O(E)$

A su vez, para más elementos, podemos generalizarlo:

Definición 4.bis: un elemento E'' es vecino de orden 2 de E, cuando $E'' \in A_O(A_O(E)) \cap E'' \notin A_O(E)$

Estas definiciones nos permiten realizar un ordenamiento de los elementos. Finalmente, la propiedad más importante es que todos los elementos se procesen en diferentes *threads*, siendo:

***Definición 5*:** Un operador O en un *thread t*, llamemos O^t(E) es *paralelamente independiente* de otro O^i(E'), si se cumple que $A(O^t(E)) \cap A(O^i(E')) = \varnothing$ para cualquier $t \neq i$.

El tiempo de procesamiento de un operador no es constante. Por lo que para asegurar que se cumplan estas propiedades y evitar que se genere un corrimiento temporal (que un thread procese más de un elemento por unidad de tiempo), los threads deben estar sincronizados; tal cual sucede en sistemas de tiempo real

A continuación se analiza un caso simple, donde cada elemento se encuentra vinculado con el vecino inmediato mediante las aristas, como en la Figura 2 , y se desea aplicar un operador de cambio de aristas para la mejora de la configuración. Se detectan tres posibles cambios (indicados en rojo en la figura), y se asigna un operador por cada par de triángulos (O^1_{Swap}(t1t2) , O^2_{Swap} (t3t4) y O^3_{Swap}(t5t6)). El Alcance de O_{Swap} son todos los triángulos que contienen a la arista seleccionada y los vecinos de los mismos por arista (también indicados en la Figura 2).

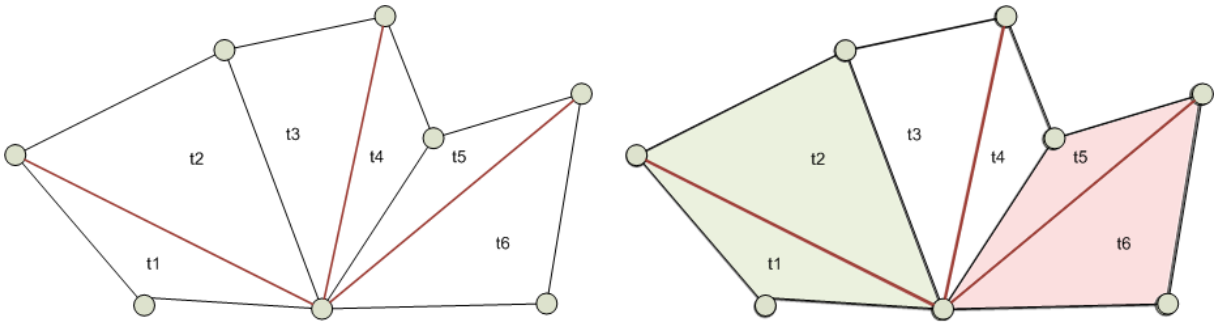

Figura 2 (izq.) elementos a procesar, (der.) Alcance para dos elementos a ser procesados

En forma secuencial, los operadores se ejecutarían en 3 tiempos distintos. En cambio si se disponen de 2 *threads*, se ordena los elementos (por ejemplo, con respecto a la coordenada X), y se asigna el primer elemento a un *thread*(O^1_{Swap})y el siguiente operador que cumple la condición de independencia al otro (O^3_{Swap}) mientras el elemento restante se procesa en el siguiente tiempo; el resultado es una secuencia igualmente válida, evaluada en 2 tiempos. Si el operador O^2_{Swap} fuera ejecutado antes que terminen ambas operaciones, los datos de vecindad perderían su integridad. De ahí la importancia de la sincronización.

Por lo tanto, el procesamiento de *n-elementos* en forma simultánea mediante un operador, implica encontrar una asignación de operaciones a *k-threads* que modifican los elementos en forma simultánea, tarea denominada de *partición implícita*. La cantidad de *threads* independientes entre los cuales distribuir el procesamiento dependerá de la cantidad de elementos de la malla, del alcance del operador y de los provistos por el hardware. Debe preverse que el espacio de memoria de almacenamiento también sea accesible en paralelo, e incluso si dichos operadores generan nuevos elementos.

El esqueleto del procesamiento propuesto para un operador es un meta-algoritmo el cual se muestra como el *Algoritmo 3.1*. Dicho algoritmo sigue un modelo PRAM: muchos procesadores, de costo uniforme y memoria compartida.

Algoritmo 3.1 :Procesar_en_paralelo (M: a mesh; O:Operator)
Local {Ts : set of Queue; P,P':set of elements}

1. P ← ElementsToProcess (O,M->elements);
2. **while** haveElements(P)
3. $N_{Threads}$ ← maxThreadsAvailable ()
4. {Generate a List to enqueue elements to threads }
5. Qs ←new Queue[$N_{Threads}$];
6. {Assign elements to a set of thread, avoiding conflicts }
7. T_{List}← assign_to_threads (P, O, Qs)*
8. MapDataToGPU(M, T_{List});
9. **while** haveElements**(any** *T* **in** T_{List})

10. P' ← process_element_in_parallel(O, T_{List}, Qs);
11. RemoveElements(P,P')
12. Synchronize();
13. **end while**
14. updateMesh(M);
15. **end while**

El primer paso es la asignación de elementos. Cuando todos los *threads* han sido ocupados, se copia la información en memoria de GPU (Paso 8) y se inicia el procesamiento en paralelo del primer elemento de cada uno. Al terminar, se seleccionan en *P'* los elementos tratados (Paso 10), se los remueve de la lista (Paso 11), se sincronizan los threads (Paso 12) y se repite hasta que no haya más elementos asignados a ninguno de los *threads*. Una vez concluido el procesamiento, se actualiza la malla y se verifica si todos elementos cumplen con los requisitos de calidad o tamaño; si existen aún elementos (Paso 2), se vuelve a aplicar la asignación.

El tiempo de procesamiento de cada elemento no es constante, por ser mallas no regulares; pero si todos los *threads* se procesan sincronizados, la condición de independencia está asegurada. En la Figura 3 se esquematiza el resultado de la asignación de los elementos y los pasos (10) al (12) del algoritmo final de procesamiento. En una buena asignación, todos los *threads* recibirán la misma cantidad de elementos a procesar.

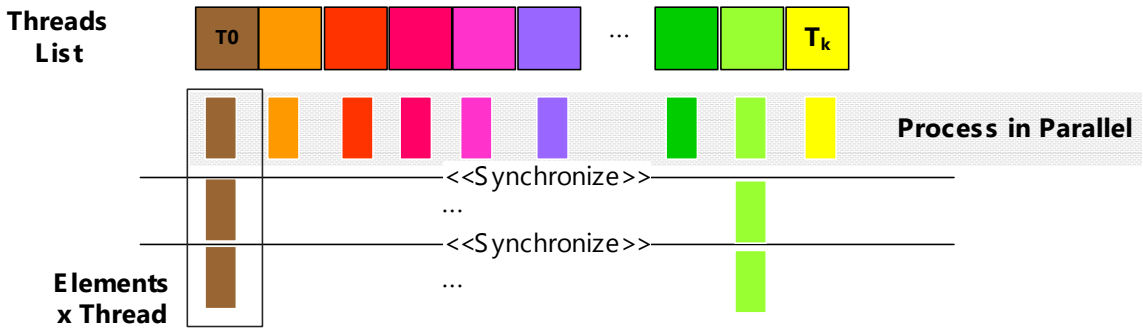

Figura 3 : Esquema de ejecución del proceso

El tiempo final de procesamiento de una malla *M* para un operador *O* comprende el *tiempo de Asignación* (Paso 3 al 7), el *tiempo de Mapeo* (Paso 8 y 14) y el *tiempo de operación* (Paso 9 al 13), de la forma de (eq.2):

$$Total_{Time} = Assignment_{Time} + Map_{Time} + \max(Proc_{Time}(O,e)) * \left\lfloor \frac{Nro_{Elementos}}{Nro_{Threads}} \right\rfloor ; \forall e \in M \quad \text{(eq.2)}$$

Encontrar la combinación óptima que paralelice el procesamiento de forma coherente para el paso (7) es un problema con complejidad NP, similar al problema clásico de asignación de *Tareas a Procesadores*, ya que es necesario probar todas las variantes de asignación de todos los elementos a su vez distintos tiempos. En las

siguientes secciones se propone una forma eficiente de generar una asignación de procesamiento que asegure las propiedades de la malla.

4 MÉTODO DE ASIGNACIÓN DE ELEMENTOS

Como se espera que todos los elementos puedan ser modificados por *k threads* independientes, debe existir una distribución de los elementos entre estos procesadores que asegure la consistencia de los cambios. Para esto, se considera la información espacial o de vecindad de los elementos y se aplica un ordenamiento temporal que evita los cambios simultáneos a datos compartidos. Esta es la mayor diferencia con los esquemas de asignación más intuitivos para GPUs, como el aplicado en (Stone et al 2007), donde el espacio se particiona en celdas regulares, y cada celda es tratada por un *thread;* método que no contempla la integridad de los datos ante cambios estructurales.

Utilizando las nociones de **Alcance** de operadores antes definidas, tomando un elemento cualquiera *E* y buscando el siguiente elemento independiente, llamado *E'*, pueden asignarse cada uno en un *thread* diferente (Paso 9 y 10) del **algoritmo 4.1**. Ahora bien, todos los elementos dentro del alcance de *E* y *E'* se marcan como visitados para descartarlos parcialmente del análisis (Paso 8). Si aún existen otros tantos elementos por procesar, siguiendo el mismo criterio, se continúa la asignación a nuevos *threads*. Para asegurarnos de primero procesar los elementos peores, ordenamos los elementos por calidad (Paso 1).

El algoritmo propuesto tiene la siguiente forma

Algoritmo 4.1: AsignaciónxAlcance (P_{Orig}: set of Elements;O:Operator; Qs: set of Queue)

Local { E : Element ; Ac : set of Elements}

1. P ←sortByQuality(P_{Orig})
2. i_{thread} ←0;
3. **for all** E **in** P
4. Ac ← alcance (E , O);
5. ***If*** marked(E) ***or***
6. markedAny (Ac) **continue**;
7. **For all** E' **in** Ac
8. mark(E');
9. Push(Qs[i_{thread}] , E);
10. i_{thread} ← (i_{thread} + 1) % $N_{Threads}$;
11. **end for**
12. **End while**
13. **Return** Qs

El resultado sobre la malla es similar a la generación de *micro-patchs,* como se muestra en la Figura 4 donde cada grupo de elementos se colorea acorde al *thread* al cual es asignado. La diferencia con otros métodos es que en ningún momento se

bloquean o indican los elementos de frontera. El costo de este algoritmo es el máximo entre el tiempo de ordenamiento, (Paso 1) y el de asignación (Pasos 2 al 12).

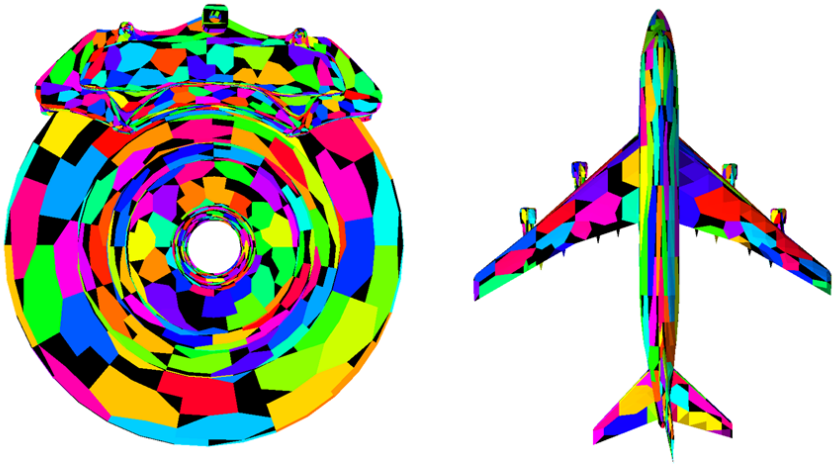

Figura 4 : Asignación de elementos por Alcance. Los elementos se indican en distintos colores de acuerdo al *thread* que es asignado

Si existen elementos no asignados al finalizar el paso (13) por no cumplir la condición de Alcance, (indicados en la Figura 4 en **color negro**), se asignarán en una próxima iteración del *algoritmo 3.1*.

4.1 Migrar los operadores a N-CPU/GPU

La memoria de la GPU se encuentra separada de la memoria de CPU, por lo que cada vez que se desea procesar en placas gráficas, es necesario copiar los datos de un espacio de memoria a otro. Al mismo tiempo, la arquitectura de las placas gráficas, concebida para trabajar con estructuras matriciales, presenta un obstáculo para otros tipos de procesamientos, debiendo replantearse los algoritmos tal cual fueron concebidos originalmente.

Las estrategias originalmente implementadas utilizando el paradigma *Object-Oriented*, donde existen miles de objetos que interactúan requieren *mapear* estas relaciones a estructuras ralas como matrices o arreglos tal cual son soportadas en GPUs. Esta conversión, indicada en el *algoritmo 3.1*, añade un tiempo al procesamiento final que es necesario considerar.

Al mismo tiempo, dado que las placas gráficas no permiten el manejo dinámico de memoria, se debe prever el espacio a ser utilizado cuando la cantidad de elementos varía. La forma más sencilla, la cual se utiliza en este trabajo, es asignar espacio en memoria de forma proporcional al número de elementos que se procesan (por cada operador se debe indicar la granularidad de entrada y salida). Por ejemplo, para el caso del Operador *Split*, se puede prever que se generan 2 triángulos nuevos por cada arista a procesar; por lo que el espacio en memoria en GPU es #Triangulos + 2*#Aristas. En otros operadores, como el de *Swap* o *Collapse*, la cantidad de memoria es constante, por lo que se reutiliza la misma estructura. Se estima en futuros trabajos proponer un esquema de utilización de memoria más inteligente.

La técnica de asignación propuesta almacena los datos en una forma matricial conveniente para el manejo en GPUs. En la Figura 5 se esquematiza el proceso seguido para una operación particular. En caso que los procesamientos posteriores lo requieran, se debe hacer un mapeo inverso, es decir, de memoria gráfica a estructura de Objetos.

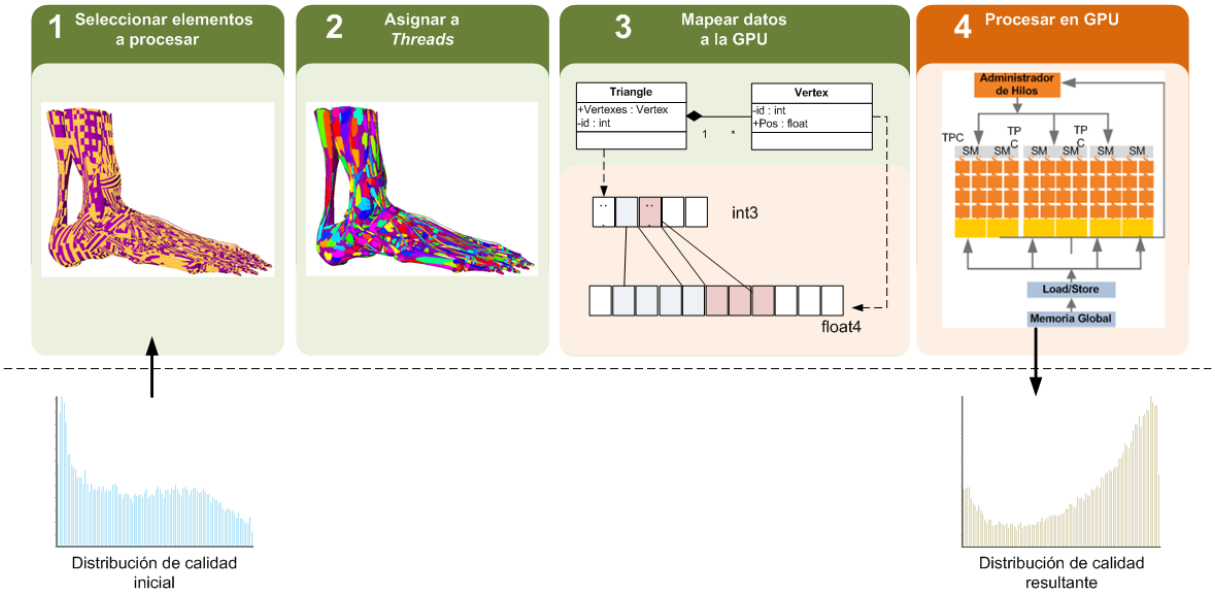

Figura 5 : Pasos del algoritmo propuesto para una mejora por intercambio de aristas

4.2 Reducción de la transferencia CPU/GPU

La transferencia de datos entre memoria de CPU a GPU agrega una demora considerable en el procesamiento, afectando el tiempo del método propuesto. Un mecanismo efectivo para reducir este tiempo es lograr todo el procesamiento posible en CPU para luego trabajar eficientemente sobre las placas gráficas. Para esto, se propone una modificación del **algoritmo 4.1**, donde se hace una reasignación de los elementos vecinos a los ya visitados, los cuales habían sido descartados del procesamiento original. Adicionalmente, se filtran los datos de entrada por calidad (Paso 1), quedando sólo aquellos elementos más deformes cuya calidad se encuentra por debajo de una tolerancia razonable dada por el usuario, llamada $Q_{Threshold}$ generalmente igual o mayor a 0.4. El algoritmo final propuesto tiene la siguiente forma.

Algoritmo 4.2: AsignaciónMejorada (P_{Orig}: set of Elements;O:Operator; Qs: set of Queue, $Q_{Threshold}$: float)

Local { E : Element ; P,Ac : set of Elements}

1. $P_{Orig} \leftarrow$ removeGreaterElements(P_{Orig} , $Q_{Threshold}$)
2. $P \leftarrow$ sortByQuality(P_{Orig})
3. **While** haveElements(P)

```
4.    i_thread ←0;
5.    P'←copy(P)
6.    for all E in P'
7.      Ac ← alcance (E , O);
8.      If  marked( E ) or
9.          markedAnyNeighbour(Ac) continue;
10.     {This step avoid the element revaluation}
11.     Remove(P_Orig, , E)
12.     For all E' in Ac
13.       mark(E');
14.       Push(Qs[i_thread] , E);
15.       i_thread ← (i_thread + 1) % N_Threads ;
16.     end for
17.   End while
18. End while
19. Return Qs
```

Esta variante del algoritmo tiene un costo considerablemente mayor al inicial, pero reduce el número de reasignaciones, logrando alcanzar más cantidad de elementos durante el mismo procesamiento en GPUs. El filtrado por $Q_{Threshold}$ reduce considerablemente el número de elementos, quedando aquellos con prioridad para ser tratados. En la Figura 6 (coloreado con *azul* mejor calidad, *rojo* peor calidad) se indica como un elemento omitido con técnica de asignación original, es procesado con esta variante del algoritmo.

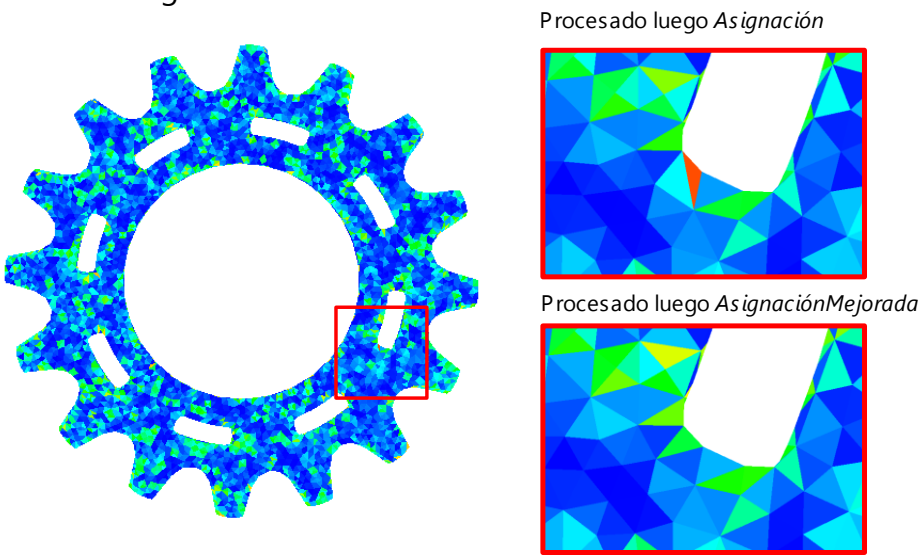

Figura 6 : Comparación del resultado de procesamiento de la malla en GPU(arriba) al aplicarse sobre la asignación inicial (Abajo) utilizando la variante con revaluación

5 EVALUACIÓN DE LOS OPERADORES

Se probaron distintos operadores utilizando el nuevo esquema de paralelización

por *partición implícita*, de los cuales se analizan en detalle dos; uno de mejoras de los elementos mediante intercambio de aristas, *Swap*, y otro de división de aristas, *Split,* donde varía la cantidad de elementos.

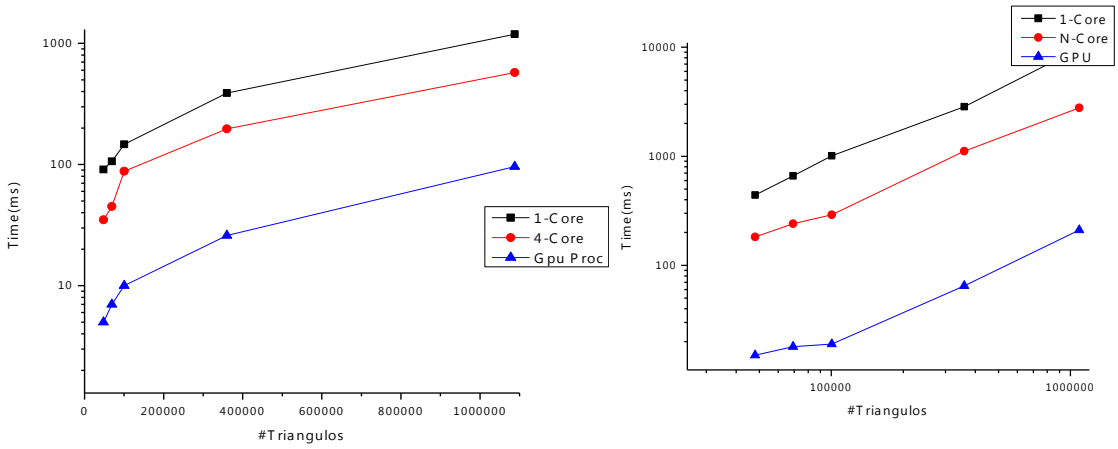

Figura 7: (izq) Tiempos de Swapping en distintas mallas (der.) Tiempos de división para distintas mallas

Las pruebas se realizaron sobre una PC de 4 núcleos con una placa gráfica Nvidia GTX 285 utilizando *OpenCL*. En CPU se implementó utilizando (Jibu 2008) un método *forloop* paralelo con sincronización por barreras que ejecuta los operadores sobre todos los elementos de forma simultánea. Se compararon en la Figura 7 sólo los tiempos de procesamiento de los operadores.

El SpeedUp alcanzado en los procesamientos de operadores para 4-CPU fue de 3x mientras que el alcanzado para GPU fue de 41x. Luego se evaluó la misma configuración para la densificación de mallas. Para la división de las mallas, el SpeedUp medio en un CPU de 4 núcleos fue de 2,13x; mientras que para la GPU la implementación alcanzó un 15x. En este caso, el *speed-up* fue menor, dado que se requiere mayor control de la calidad de los elementos.

Finalmente se computan los tiempos totales para *swapping* indicado en la Tabla 1, teniendo en cuenta el tiempo de asignación y mapeo GPU-CPU.

Malla	#Triangulos	Proc.GPU (ms)	Assign(ms)	Map Mem(ms)	**GPU+CPU (ms)**	4-CPU (ms)	SpeedUp
FerrariBrake	23000	4	43	68	**115**	364	3,17
Impeller	197450	18	684	600	**1302**	3681	2,83
Hand	654000	54	1937	1515	**3506**	9963	2,84
Terrain	360000	27	914	707	**1648**	4595	2,79

Tabla 1 : Tiempos de Swap totales

Observando estos tiempos finales, compuestos por *Asignación+Mapeo+GPUProc*, comparado con una implementación en paralelo, se logró un *speedUp* promedio final de 3x. Es de destacar, que el paso de mapeo a CPU es requerido por el motor de Rendering y por otros algoritmos actualmente utilizados, siendo en algunos casos incluso al tiempo de asignación. En futuras implementaciones, donde el proceso de

remallado se aplique íntegramente en GPU el tiempo de mapeo se reducirá a la mitad, y se estima mejorar la eficiencia del algoritmo de asignación.

6 CONCLUSIONES

Se ha desarrollado un nuevo método general para la distribución del procesamiento de una malla en múltiples *Threads o procesadores locales* en base a características de vecindad de los elementos, a las coordenadas espaciales y a una sincronización del procesamiento. Se analizaron variantes de dicho proceso, que han mostrado que el mismo es viable y aplicable tanto en arquitecturas en paralelo como placas gráficas, siendo muy escalable a mayores problemas. Las mayores ventajas de aplicar este esquema surgen cuando las operaciones locales son complejas.

Se propusieron dos alternativas de asignación que resultaron aptas para el cálculo, pero que no maximizaban la localidad de los datos en memoria rápida. Se estima seguir trabajando en esta dirección, así como también en reducir la cantidad de procesamiento en CPU, a fin de mejorar la performance total del proceso de remallado.

Por otra parte, queremos destacar que este esquema es complementario de los algoritmos de procesamiento distribuido y se estima sea aplicable a procesos de remallado sobre otro tipo de mallas, como las compuestas por tetraedros.

REFERENCIAS

B.Hudson, G.Miller, T.Phillips, Sparse Parallel Delaunay Mesh Refinement, *Carnegie Mellon University*; doi=10.1.1.92.5478 (2007)

Cifuentes, D'Amato, Lotito Remallado en paralelo basado en criterios múltiples, In : *XII Workshop de Investigadores en Ciencias de la Computación*, (2010)

Cohen, Varshney, Manocha, Turk, Weber, Agarwal, Brooks, and Wright. Simplification envelopes. In: *ACM SIGGRAPH '93 Conference Proceedings*. New Orleans, Louisiana; pp. 119–28 (1996).

Correa Reina G., Vénere M, Lotito P. *Optimización de Calidad para la Generación de Mallas de Superficie* In: Mecánica Computacional Vol. XXVI, (2006).

Cozzi, P. and Loo, B. T. Parallel Processing City Models for Real-Time Visualization. http://www.seas.upenn.edu/~pcozzi/PreprocessingCityModels.pdf (2008).

Dehne, Langis, and Roth. Mesh simplification in parallel. In: *Proceedings 4th International Conference on Algorithms and Architectures for Parallel Processing*. Hong Kong; pp. 281–90 (2000).

Eppstein D., Goodrich M., Kim E., Tamstorf R. Motorcycle Graphs: Canonical Quad Mesh Partitioning In: *Eurographics Symposium on Geometry Processing* vol. 27, n.5 (2008)

Fich F., Hendler D., Shavit N. On the inherent weakness of conditional synchronization primitives. In: *23rd Annual ACM Symposium on Principles of Distributed Computing*, 2004, pp. 80-87.

Garland and Heckbert. Surface simplification using quadric error metrics. In: *ACM*

SIGGRAPH '97 Conference Proceedings. Los Angeles, Cal.; pp. 209–16 (1997).

Garland and Heckbert Simplifying surfaces with color and texture using quadric error metrics. In: *Proceedings IEEE Visualization '98, Research Triangle Park*, NC; pp. 263–9 (1998).

Hoppe, DeRose, Duchamp, McDonald, and Stuetzle Mesh optimization. In: *ACM SIGGRAPH '93 Conference Proceedings*, Anaheim, Cal.; pp. 19–26 (1993)..

Hoppe. Progressive meshes. In: *ACM SIGGRAPH '96 Conference Proceedings*, New Orleans, Louisiana; pp. 99–108 (1996).

Hu, Sander, and Hoppe. Parallel view-dependent refinement of progressive meshes. In: *Proceedings of the 2009 Symposium on Interactive 3D Graphics*. Boston, Mass.; pp. 169–76 (2009).

B. Hudson, G. Miller, T. Phillips Sparse Parallel Delaunay Mesh Refinement. In:*ACM Symposium on Parallelism in Algorithms and Architectures*, California, USA, (2007)

Kopyssov S.P., Novikov A.K. , Parallel Adaptive Mesh Refinement with Load Balancing on Heterogeneous Cluster, *Parallel and Distributed Computing Practices, vol. 5, Nº4*, (2002).

Jibu SDK. Web site: http://axon7.com/. (2008)

Moulitsas I., Graph Partitioning for Scientific Computing Applications. *Seminar in Department of Applied Mathematics*, University of Crete, Heraklion, (2007).

Rasch and Schmidt. Parallel mesh simplification. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Algorithms* (PDPTA 2000). Las Vegas, Nev, (2000).

Stone J., Phillips J., Freddolino P., Hardy D.,Trabuco L., Schulten K. *Accelerating Molecular Modeling Applications with Graphics Processors*. Journal of Computational Chemistry, vol 28:pp 2618:2640 (2007).

Suárez J., Plaza A., Carey G, A geometric diagram and hybrid scheme for triangle subdivision, In: Comp. Aid. Geom. Design, Vol. 24, N± 1, 193{27, (2007).

Tang, Jia, and Li. Simplification algorithm for large polygon model in distributed environment. In: *Lecture Notes in Computer Science: Advanced Intelligent Computing Theories and Applications with Aspects of Theoretical and Methodological Issues*. Heidelberg, Germany: Springer-Verlag; pp. 960–9, (2007)

Turk and Levoy. Zippered polygon meshes from range images. In: *ACM SIGGRAPH '94 Conference Proceedings*. Orlando, Flo.; pp. 311–8, (1994)

Walter and Healey Attribute preserving dataset simplification. In: *Proceedings IEEE Visualization 2001*, San Diego, Cal.; pp. 113–20, (2001).

Wang D., Hassan O., Morgan K. , Weatherill N. EQSM: An efficient high quality surface grid generation method based on remeshing Comput. In: *Methods Appl. Mech. Engrg*. 195 pp. 5621–5633(2006)

Yongquan, Nan, Pengdong, Chu, Jintao, and Rui A parallel memory efficient framework for out-of-core mesh simplification. In: *Proceedings 11th International Conference on High Performance Computing and Communications* (HPCC-09). Korea; pp. 666–71 (2009).