UNIVERSIDAD NACIONAL DEL LITORAL
Facultad de Ingeniería y Ciencias Hídricas
Centro de Investigación de Métodos Computacionales

GEOMETRÍA COMPUTACIONAL APLICADA A LA GENERACIÓN EN PARALELO DE MALLAS DE ELEMENTOS FINITOS

Pablo José Novara

Tesis remitida al Comité Académico del Doctorado
como parte de los requisitos para la obtención
del grado de
DOCTOR EN INGENIERIA
Mención Mecánica Computacional
de la
UNIVERSIDAD NACIONAL DEL LITORAL

2016

Comisión de Posgrado, Facultad de Ingeniería y Ciencias Hídricas, Ciudad Universitaria, Paraje ”El Pozo”,
S3000, Santa Fe, Argentina.

UNIVERSIDAD NACIONAL DEL LITORAL
Facultad de Ingeniería y Ciencias Hídricas

Santa Fe, 19 de abril de 2016.

Como miembros del Jurado Evaluador de la Tesis de Doctorado en Ingeniería titulada ***"Geometría Computacional Aplicada a la Generación en Paralelo de Mallas de Elementos Finitos"***, desarrollada por el Ing. Pablo José NOVARA en el marco de la Mención "Mecánica Computacional", certificamos que hemos evaluado la Tesis y recomendamos que sea aceptada como parte de los requisitos para la obtención del título de Doctor en Ingeniería.

La aprobación final de esta disertación estará condicionada a la presentación de dos copias encuadernadas de la versión final de la Tesis ante el Comité Académico del Doctorado en Ingeniería.

Dr. Juan Pablo D'Amato

Dr. Mario Storti

Dr. Enzo Dari

Dr. JOSÉ LUIS MACOR
SECRETARIO DE POSGRADO
Facultad de Ingeniería y Ciencias Hídricas

Dr. Rainald Löhner (*)

Santa Fe, 19 de abril de 2016

Certifico haber leído la Tesis, preparada bajo mi dirección en el marco de la Mención "Mecánica Computacional" y recomiendo que sea aceptada como parte de los requisitos para la obtención del título de Doctor en Ingeniería.

Dr. Norberto Nigro
Codirector de Tesis

Dr. Nestor Calvo
Director de Tesis

(*) Participó mediante videoconferencia

Universidad Nacional del Litoral
Facultad de Ingeniería y
Ciencias Hídricas

Secretaría de Posgrado

Ciudad Universitaria
C.C. 217
Ruta Nacional Nº 168 – Km. 472,4
(3000) Santa Fe
Tel: (54) (0342) 4575 229
Fax: (54) (0342) 4575 224
E-mail: posgrado@fich.unl.edu.ar

Agradecimientos

En primer lugar, quiero agradecer a mi familia. Especialmente a mis padres por su ejemplo y por haberme dado la posibilidad elegir mi carrera y de dedicarme al estudio sin otras preocupaciones. También a mi esposa, Natalia, por su apoyo incondicional en esta última y difícil etapa, sin el cual habría sido imposible completarla.

Deseo agradecer además a mi director, Nestor Calvo, por compartir su experiencia, por todo el tiempo dedicado a tantos debates, discusiones y revisiones, y por confiar en mi para llevar adelante este proyecto. Y a mi codirector, Norberto Nigro, por haber desviado mi interés hacia este tema de tesis, aportando su conocimiento sobre el campo de aplicación de estos desarrollos y su valiosa ayuda en la revisión de este texto.

Finalmente, a todos mis compañeros de los dos lugares de trabajo en los que se desarrollaron estas ideas (FICH-UNL y CIMEC) quienes de una u otra manera contribuyeron a este logro, tanto desde lo técnico como desde lo humano. En particular a Juan Gimenez por el tiempo dedicado a probar los desarrollos en sus primeras versiones y por aportar valiosos casos de prueba para la validación de los mismos.

Resumen

En esta tesis se presentan dos algoritmos para la tetraedrización de un conjunto de puntos. El primero de ellos toma como entrada solo un conjunto de puntos y genera una malla de tetraedros para el interior de la envolvente convexa (convex-hull) de dicho conjunto. La malla resultante es una malla Delaunay, y el algoritmo generado es robusto tanto frente a las ambigüedades conocidas del criterio Delaunay, como frente a los errores numéricos debidos a la precisión finita con que se realizan los cálculos. El segundo algoritmo agrega a los datos de entrada un conjunto de conectividades de frontera (una malla de superficie cerrada) que deberá ser respetado en la malla de salida que se genere. Esta malla entonces cubrirá, también con elementos tetraédricos, el volumen delimitado por la malla de frontera de entrada, aunque no todos sus elementos respetarán la condición Delaunay, dado que en caso de conflicto se respeta la frontera impuesta por sobre la condición Delaunay. Esta variación del algoritmo permite además mejorar la calidad de los elementos generados, evitando la formación de slivers, elementos de muy baja calidad muy frecuentes en mallas Delaunay en 3D. Todos los algoritmos presentados mantienen invariante el conjunto de puntos, por lo que los problemas encontrados se resuelven sin agregar, quitar ni mover puntos. Estas limitaciones son comunes en ciertas operaciones de interpolación y en algunos métodos de simulación por partículas, donde las partículas se corresponden con los nodos de la malla. Para ambos métodos de tetraedrización se proponen estrategias de paralelización, tanto para arquitecturas de hardware de memoria compartida, como para arquitecturas de hardware de memoria distribuida e híbridas. Se discuten las ventajas y desventajas de cada una, los problemas encontrados y las posibles soluciones, las diferencias importantes en las implementaciones para cada tipo de arquitectura, y finalmente se presentan resultados y se analiza la eficiencia y escalabilidad de estas implementaciones. Se describen también en este trabajo todas las estructuras de datos utilizadas en ambos métodos y los algoritmos asociados a las mismas (válidos tanto en 2D como en 3D, y tanto en su versión serie como paralela), junto con las justificaciones correspondientes para cada una de estas elecciones.

Abstract

This thesis presents two algorithms for parallel generation of unstructured all-tetrahedral meshes for a given set of points. The first method generates a Delaunay mesh for the interior of the point set's convex hull. This algorithm is robust in the sense that it solves all the problems related to numerical errors and Delaunay criterion's ambiguities. The second method adds to the input a fixed boundary mesh, and generates a Delaunay-dominant mesh for the domain defined by such boundary. This generated mesh fits the given boundary mesh connectivities and also improves mesh quality avoiding the generation of slivers, low-quality elements very common in Delaunay meshes. None of this methods will neither add nor move or remove nodes. This makes these algorithms suitable for many common interpolation operations, and for some particle-based simulations where nodes represents particles. Parallel implementations for both shared memory and distributed memory architectures are proposed for the two mesh generation problems presented. Advantages and disadvantages of each one, all problems found and the proposed solutions, and the major differences in the implementations for the two most usual kinds of parallel hardware architectures are described in this thesis. Finally, some results are presented and the parallel scalability and efficiency of the method is discussed. This thesis also includes descriptions for all the necessary data structures for the current implementations and the associated algorithms (for 2D and 3D, and both serial and parallel versions), along with all important details to justify those elections.

Índice general

Índice de figuras

Capítulo 1

Introducción

Durante gran parte de la historia de la informática, el hardware evolucionó de forma tal que la potencia de cómputo creció exponencial y sostenidamente por años, ajustándose en muchos aspectos a lo que se conoce como Ley de Moore. Esta ley (observación en realidad) afirma que el número de transistores en un circuito integrado puede duplicarse (aproximadamente) cada 2 años. Hasta alrededor de 10 años atrás, la observación se verificaba y podía además hacerse extensiva a las velocidades de reloj de los procesadores y otros aspectos similares. En consecuencia, el crecimiento exponencial de la potencia de cómputo permitía desarrollar software para resolver operaciones cada vez más complejas, o aplicar los desarrollos existentes a problemas cada vez más grandes (mayor volumen de datos). Pero hemos llegado un límite práctico impuesto por la física. Evidencia de ello es que las velocidades de reloj de los procesadores han estado estancadas alrededor de los 3GHz[1] durante la última década (más detalles sobre este argumento y gráficas ilustrando la evolución del hardware que lo confirman en [1]). Para poder continuar incrementando la potencia de cómputo luego de alcanzada esta barrera, los fabricantes de hardware han tenido que cambiar su enfoque tradicional y buscar nuevos caminos.

El paralelismo parece ser, en el presente (y seguramente por varios años más), el principal camino a seguir. Desde hace ya tiempo, aún en las PCs hogareñas y hasta en los teléfonos celulares disponemos usualmente de varios procesadores, procesadores multi-core, y/o tecnologías afines como HyperThreading. Más aún, componentes específicos como las placas gráficas han po-

[1]En realidad, el máximo teórico para la velocidad de reloj se encuentra alrededor de 5GHz, pero el hardware actual intenta mantenerse por debajo de ese límite por razones prácticas tales como el consumo de energía y la generación de calor

dido aprovechar aún más el cambio de dirección debido a la mayor libertad que implica el hecho de que su evolución no esté limitada por el requisito de backward-compatibility, como sí lo están en mayor o menor medida los fabricantes de CPUs. Y este nuevo modo de crecimiento, a diferencia de lo que ocurría anteriormente, afecta no solo al hardware sino también al software. Si un procesador aumenta su velocidad, cualquier software puede beneficiarse por ello sin requerir cambios. Pero si, por el contrario, reemplazamos un procesador single-core por uno multi-core, en general un software desarrollado para el primero no será capaz de aprovechar las ventajas del segundo. El desarrollo de software también ha debido modificar sus tendencias y cambiar sus paradigmas para acompañar y aprovechar el cambio en el hardware sobre el que corre. Ahora el software se desarrolla directamente para arquitecturas paralelas y en muchos casos heterogéneas.

Este cambio ha generado nuevos problemas y ha vuelto poner en vigencia muchos otros. Concurrencia y paralelismo han sido campos de investigación activos desde los años 60. El cambio en las técnicas y en los problemas relacionados a la implementación del software repercute no solo a bajo nivel, sino que también se manifiesta directamente niveles de abstracción superiores. La programación funcional, por ejemplo, cuyos orígenes se remontan a los años 30, muy popular en la década del 80, ha recobrado su vigencia en gran parte gracias al auge del paralelismo. Es decir, el cambio no es un detalle de implementación, sino que debe ser tenido en cuenta desde el comienzo para el diseño de un método o algoritmo que luego se plasmará en software. Por ello, las ciencias de la computación y todas las demás áreas de investigación y desarrollo que dependen fuertemente del uso de computadoras como soporte esencial para sus métodos y procesos (como lo es la mecánica computacional), han comenzado dedicar grandes esfuerzos a adaptarse al cambio y replantear sus métodos acorde a los nuevos escenarios tecnológicos.

En el campo de la mecánica computacional en particular, esta "revolución" no es realmente reciente ni novedosa. Debido al tamaño y la naturaleza de los problemas computacionales involucrados, las arquitecturas paralelas (mayormente en forma de clusters) se han estado utilizando ya por muchos años en los institutos de investigación especializados, y en algunos casos también en la industria. Sin embargo, la tendencia actual del mercado de PCs hogareñas ha hecho mucho más variado y accesible el hardware paralelo disponible, y ha incrementado la necesidad de explotar estas nuevas arquitecturas. Además, el creciente interés y desarrollo por parte de otras áreas de las cuales la mecánica computacional se nutre (como lo son las ciencias de la computación en general, o la geometría computacional en particular) le han abierto nuevos campos de aplicación e investigación, como la industria

del entretenimiento o la simulación en tiempo real, una sub-área de mucho interés en la actualidad y en marcado crecimiento. Esto es utilizar métodos que permitan realizar las simulaciones temporales empleando un tiempo de cómputo proporcional al tiempo simulado, con constantes de proporcionalidad relativamente bajas. Aún cuando se deba sacrificar en parte la precisión del resultado, es útil y valioso en el proceso completo de modelado y resolución de un problema real, un método numérico que permita obtener un primer resultado de forma relativamente rápida (esto es, por ejemplo, en términos de minutos u horas, en lugar de días de cómputo).

En este campo, por su multidisciplinariedad, se ha atacado el problema desde varios frentes. Un primer enfoque consistió en revisar los algoritmos numéricos utilizados y sus implementaciones, intentando aprovechar el paralelismo para realizar un mismo volumen de cálculos en un menor tiempo, y de esta forma potenciar métodos de simulación numérica tradicionales y establecidos, como el Método de los Elementos Finitos, el método de los Volúmenes Finitos, etc. Por otro lado, se han explorado recientemente nuevos métodos (o revisado métodos ya conocidos pero no tan utilizados) que, al menos en apariencia, son naturalmente paralelizables. En general son métodos basados en partículas y similares, como SPH ([2]). Combinando ambos enfoques, métodos como PFEM y PFEM-2 ([3],[4],[5],[6]) han introducido variantes novedosas a los métodos clásicos con el objetivo de reducir el volumen de cálculo necesario para un determinado problema. Por ejemplo, PFEM-2 permite utilizar una discretización más gruesa del eje temporal, reduciendo así la cantidad de pasos de tiempo necesarios en una simulación, utilizando para ello mecanismos de integración temporal diferentes que permiten mantener la estabilidad en las soluciones.

La gran mayoría de los métodos utilizados en mecánica computacional requieren una malla sobre la cual realizar los cálculos. La generación de una malla puede ser un proceso costoso (en términos de tiempo de ejecución y/o consumo de memoria) si el problema es grande, la geometría demasiado compleja, o se requiere mucha precisión. Pero, en muchos casos, se utiliza una o unas pocas mallas durante todo el proceso de cálculo, construidas en una etapa previa que forma parte del preproceso. En estos casos, suele ocurrir que el costo computacional de la etapa de cálculo supera ampliamente el costo de la generación de la malla sobre la cual se realizan dichos cálculos. Sin embargo, hay métodos de simulación que requieren la regeneración o adaptación de la malla en cada paso o cada pocos pasos, y en estos escenarios el costo de estas operaciones puede ser comparable, o hasta muy superior al costo del cálculo (la resolución de las ecuaciones físicas que modelan el problema, utilizando esa malla como soporte). Este es en general el caso de las formula-

ciones Lagrangianas, y en particular de muchos métodos numéricos basados en partículas, que requieren mallas como soporte para algunos de sus cálculos. Estos últimos resultan ser también los más naturalmente paralelizables. Esta situación puede convertir al proceso de mallado/remallado en un cuello de botella para el proceso de simulación completo.

Es en este contexto donde se evidencia la importancia de desarrollar algoritmos de geometría computacional paralelizables, para dar soporte a estos métodos de simulación numérica y explotar al mismo tiempo las capacidades del hardware disponible. Es decir, que la generación de mallas también debe adaptarse a las nuevas formas de crecimiento en el poder de cómputo que provee el hardware, ya que en general si un proceso tiene más de una etapa consumiendo una parte significativa del tiempo total de ejecución, no se puede obtener grandes beneficios si no se paralelizan todas ellas (ley de Amdahl[7]). Si, por ejemplo, en una simulación en la cual se identifica una etapa crítica que consume alrededor del 80 % del tiempo total de ejecución, y solo esta etapa se paraleliza (aún con un 100 % de eficiencia paralela, lo que significa que con suficiente cantidad de procesadores su tiempo de ejecución se vuelve despreciable), el tiempo total solo se verá reducido como máximo en un factor de 5, ya que el 20 % restante pasará a ser dominante.

Por otro lado, en muchas situaciones, donde la geometría del problema varía durante la simulación, aunque se utilice en cada paso de cálculo de la simulación una malla fija (es decir, un método que no requiere remallado), esta malla podría no ser la misma en todos los pasos. Por ejemplo, la simulación de un proceso de estampado de chapa que resuelva la parte física mediante el método de los volúmenes finitos, puede requerir en un paso intermedio simular un corte realizado en la chapa. Además de modificarse la malla para representar la nueva geometría, el proceso de simulación del corte debe transferir los valores nodales y las variables asociadas a los puntos de integración de Gauss calculados sobre la malla anterior, a los nodos y elementos de la nueva malla. Esta clase de procesos suele involucrar la construcción de una malla auxiliar (por ejemplo, utilizando como nodos los puntos de Gauss), para obtener a partir de la misma los pesos con los que interpolar los nuevos valores de las variables involucradas (detalles y ejemplos en [8], [9], [10]). En el proceso de interpolación, la generación de la malla auxiliar es (con mucha diferencia) el paso más costoso, y suele ser más costoso aún (tanto en tiempo como en memoria) que las operaciones geométricas necesarias para adaptar la malla de cálculo a la nueva geometría. En conclusión, estas simulaciones, como cualquier otra donde el tiempo de generación de una malla (auxiliar o no) sea el que domine el tiempo de ejecución total, también se verán beneficiadas por el uso de algoritmos paralelos.

Por todo lo expresado en esta introducción se puede concluir que el diseño y la implementación de algoritmos de geometría computacional en general y de generación de mallas en particular que puedan aprovechar las arquitecturas de hardware paralelas actuales es efectivamente un área de interés, tanto teórico como práctico, cuyos resultados además tendrían aplicación directa en la industria. En esta tesis se aborda la solución de un caso particular de este problema: la generación de una malla de tetraedros cuando tanto la frontera como las posiciones de los nodos interiores se encuentran prefijadas. Este es el caso en varios de los ejemplos mencionados, como PFEM-2 y el problema de interpolación de valores nodales y puntos de Gauss. Se podrá concluir además que el desarrollo presentado puede utilizarse como base para desarrollar otros casos más generales con diferentes restricciones.

1.1. Objetivos

El objetivo de este trabajo es explorar y desarrollar técnicas de geometría computacional y de programación paralela, para su aplicación a la generación de mallas Delaunay tanto 2D como 3D, para obtener una implementación robusta y eficiente de los algoritmos que se propongan, y finalmente evaluar sus desempeños tanto en arquitecturas de hardware paralelo de memoria compartida, como de memoria distribuida. En particular, se buscará mejorar un algoritmo conocido resolviendo diferentes problemas propios del mismo, y extender su aplicación a un mayor rango de problemas desarrollando las variantes necesarias para ello. Esto incluye:

- el análisis detallado del método inicial para lograr reproducir sus resultados,
- la caracterización de sus fortalezas y limitaciones,
- la resolución de los problemas asociados a la aritmética de precisión finita, que limitan considerablemente su robustez y su campo de aplicación,
- el desarrollo de algoritmos de búsqueda y ordenamiento espacial específicos para reducir los tiempos de cómputo en etapas clave del proceso,
- el desarrollo y la implementación de las variantes necesarias para obtener un nuevo algoritmo que logre respetar en su resultado una frontera previamente impuesta por el usuario,

- la mejora de la eficiencia paralela mediante modelos mixtos con características tanto de arquitecturas de memoria compartida como de arquitecturas de memoria distribuida,
- la reducción de los tiempos de comunicación en modelos de memoria distribuida buscando aplicar coherencia y considerar el proceso de simulación en el que se encuentra inserto,
- la evaluación objetiva de los resultados, los cuales deberán ser completamente reproducibles a partir de lo expuesto en este trabajo,
- y otras mejoras menores en distintos aspectos.

Se pretende entonces lograr una implementación 3D que mejore la eficiencia paralela que presentan los métodos disponibles actualmente y permita su utilización tanto en arquitecturas de memoria distribuida como en arquitecturas de memoria compartida, para contribuir así a que los métodos de partículas basados en mallas móviles compitan en tiempos de cómputo con otros métodos sin malla o de malla fija (que presentan otro tipo de inconvenientes). Finalmente, se pretende discutir brevemente también la posibilidad de adaptar los métodos desarrollados para problemas de mallado general (sin puntos interiores impuestos) y otras aplicaciones alternativas.

1.2. Organización de la tesis

En el capítulo 2 se presentan las definiciones y los conceptos matemáticos y geométricos básicos relacionados a la generación de mallas y su aplicación en problemas de mecánica computacional, lo cual permite describir con mayor detalle y precisión el problema que aborda esta tesis. Se presenta además una breve reseña acerca de los métodos comúnmente utilizados para resolver el problema planteado y algunas estrategias de paralelización ya presentes en la literatura específica.

El capítulo 3 introduce los conceptos básicos relacionados a las arquitecturas de hardware paralelo y su programación, que serán de interés en capítulos posteriores para justificar los mecanismos de paralelización escogidos y las estructuras de datos planteadas, y para explicar además los resultados y conclusiones del capítulo 7. Se discuten además algunos problemas de más bajo nivel y detalles de implementación particulares que serán referenciados posteriormente.

El capítulo 4 presenta las estructuras de datos y los algoritmos de ordenamiento espacial utilizados y considerados para los capítulos 5 y 6, haciendo énfasis en sus ventajas y desventajas desde un punto de vista funcional, y analizando además su eficiencia.

Los capítulos 5 y 6 describen los métodos de tetraedrización para los escenarios en los que solo se tiene por entrada un conjunto de puntos, y en los que se tiene además una frontera impuesta a respetar, respectivamente. Estas descripciones detallan todas las decisiones de diseño, tanto desde el punto de vista geométrico, como así también considerando su implementación, pero sin incluir nivel alguno de paralelización.

El capítulo 7 introduce las modificaciones necesarias para desarrollar versiones paralelizables de dichos algoritmos, en los dos tipos de arquitecturas de hardware paralelo más usuales, e incluye tablas y gráficos ilustrando los tiempos de ejecución obtenidos para las implementaciones de los algoritmos presentados, junto con las interpretaciones de dichos resultados.

Finalmente, el capítulo 8 resume las conclusiones más importantes de esta tesis, y plantea posibles caminos a seguir para continuar desarrollando los métodos propuestos o similares.

Se incluye además una sección adicional anexa con descripciones y detalles de aspectos (mayormente de implementación) que no son de lectura obligada para el seguimiento de este trabajo, pero o bien incluyen pequeños aportes indirectamente relacionados a los algoritmos desarrollados, o bien detallan problemas y soluciones necesarias para reproducir completamente sus resultados.

Capítulo 2

Generación de mallas

Desde un punto de vista geométrico, una malla es una subdivisión (partición) conforme de un espacio (dominio) en celdas poliédricas[11]. "Partición" implica en este contexto que la unión de los volúmenes interiores de todos los elementos equivale al interior del dominio completo, y que la intersección es nula. "Conforme" significa que la intersección entre dos elementos solo puede ser un elemento (de menor dimensión) común a ambos (por ejemplo, una cara común a dos tetraedros, o una arista común a dos triángulos). Esto implica que en una malla 2D no pueden existir, por ejemplo, uniones en T. Se dice que el conjunto de celdas resultantes conforma una representación "discreta" de dicho dominio. Existen múltiples campos de aplicación en los cuales se requiere generar mallas a partir de dominios. Cada campo de aplicación definirá diferentes requisitos sobre el tipo de malla a utilizar y las propiedades de las celdas (elementos) de la misma. Por ejemplo, para modelar personajes y escenarios 3D en videojuegos o cine animado, se requieren usualmente mallas que representen sólo la superficie de los objetos, ya que el objetivo es generar una visualización del mismo y el interior en general no es visible ni relevante para los cálculos de visualización en la superficie visible. Además, las propiedades que se requieren de los elementos están determinadas mayormente por un compromiso entre los requerimientos de las técnicas de aplicación de texturas y cálculos de iluminación, y el costo computacional aceptable según la aplicación. Cuando se requiere simular la física de un problema, en cambio, es generalmente necesario conocer y representar el volumen (interior) de un cuerpo 3D, y no solo su superficie (frontera). En estos casos, las propiedades deseables, tanto del tipo de malla como de sus elementos individuales, están determinadas por la variabilidad de los campos simulados dentro del dominio, y por los métodos numéricos que utilizan estas mallas como soporte para la resolución de sistemas de ecuaciones diferenciales y los mecanismos

de interpolación tanto de los datos de entrada como de los resultados sobre la malla. En este trabajo se hará énfasis en este segundo tipo de aplicaciones.

Los métodos clásicos para simular la física de un problema de mecánica de sólidos o de fluidos, como el método de los elementos finitos (FEM) y el método de los volúmenes finitos (FVM) han dado lugar al desarrollo de múltiples técnicas de generación de mallas tanto 2D como 3D. Un problema físico-mecánico complejo, modelado mediante ecuaciones diferenciales, se resuelve descomponiendo el dominio en celdas simples y resolviendo las ecuaciones que gobiernan el problema de forma aproximada en cada una de estas celdas. Se reemplaza así un conjunto de pocas ecuaciones diferenciales muy complejas sobre un dominio continuo arbitrario, por un sistema de miles o millones de ecuaciones simples aplicadas sobre dominios casi triviales. Por ejemplo, ecuaciones lineales que modelan lo que ocurre dentro de un tetraedro, aplicadas a miles de tetraedros que conjuntamente representan el dominio original.

En general, se denomina "elemento" a cada celda de la malla. Los elementos suelen definirse como polígonos/poliedros a partir de sus vértices, utilizando algún criterio predefinido para el orden de los mismos. Por ejemplo, en una malla 2D se pueden listar los vértices de un triángulo recorriéndolos en sentido anti-horario, definiendo así un orientación para el mismo. Si un elemento tiene sus vértices listados en el sentido opuesto, se dice que está invertido. A los vértices de los elementos se los denomina "nodos" de la malla. Dos nodos pertenecientes a un mismo elemento se denominan "adyacentes". Dado que una malla válida para este tipo de aplicaciones determinará una partición del espacio (cubrirá el dominio completo, y sin superposiciones entre elementos), bastará determinar el conjunto de nodos y sus adyacencias ordenadas (también denominadas "conectividades") para definirla.

La calidad de una malla se mide en función de cuan bien representa el dominio original y de la calidad individual de sus elementos. El área o volumen abarcado por el conjunto de elementos debe aproximar correctamente el área o volumen que define el dominio del problema. Además, el tamaño de un elemento (típicamente denominado con la letra h) estará directamente relacionado con la precisión con que se podrá aproximar la solución al problema físico modelado. Es decir, cuanto mayor cantidad de elementos se utilicen (elementos más pequeños y mayor cantidad de nodos), mejor será la aproximación al dominio y a la solución del problema. Sin embargo, reducir el tamaño promedio de los elementos de una malla implica aumentar el costo computacional tanto en la generación de la misma, como en la resolución de las ecuaciones del problema sobre estos elementos. Por ello, se debe adoptar

una solución de compromiso que proporcione un balance adecuado entre costo y precisión. En general, el tamaño de elemento deseado será especificado por el usuario en función de los factores mencionados, y podrá ser variable a lo largo del dominio, para obtener mayor precisión en las zonas de mayor interés o en las que la solución presenta una mayor variabilidad. Entonces, cuando el algoritmo de generación de la malla tiene la responsabilidad de generar los nodos interiores, el tamaño de elemento deseado se suele tomar como dato de entrada. Este problema, en el cual el algoritmo debe generar tanto los nodos como las conectividades, es el caso más usual, pero no es el que se analiza en profundidad en el presente trabajo.

Es importante garantizar propiedades adicionales sobre la forma de los elementos y su densidad (además de la fidelidad con que los elementos representan el dominio), debido al tipo de ecuaciones que resuelven dentro de cada uno, y a características propias de las técnicas de resolución numéricas y la aritmética de precisión no-infinita que ofrece una PC. Esto es, garantizar una cierta calidad en los elementos, que además de considerar su tamaño considera otras medidas importantes en este contexto, como su relación de aspecto, la amplitud de sus ángulos, su similaridad con un elemento ideal, o hasta las propiedades de la submatriz que el mismo generará en el sistema de ecuaciones del problema (ejemplo en figura 2.1). No existe consenso absoluto acerca de cómo obtener una medición objetiva de estas propiedades, sino que existen varios criterios aceptables, y la selección de uno u otro, junto con los umbrales dentro de los cuales un elemento se considera aceptable, dependen en gran medida del problema simulado (su geometría, el método de cálculo utilizado, el tipo de ecuaciones que se resuelven y su interpretación física, el poder de cómputo disponible, etc). Un resumen comparativo de criterios habituales para mallas de triángulos o tetraedros y sus ventajas y desventajas se puede encontrar en [12].

Las consideraciones y técnicas utilizadas para la generación y paralelización de mallas estructuradas pueden ser muy diferentes a las necesarias para mallas no estructuradas. El tema de estudio de esta tesis es la generación de mallas no estructuradas, y en particular mallas Delaunay. Por este motivo, en este capítulo se omitirán intencionalmente las menciones a métodos de generación de mallas estructuradas, y se hará énfasis solamente en las propiedades de los algoritmos de generación de mallas no estructuradas.

Las dos formas más comunes de generación de mallas no estructuradas se basan en técnicas de Avance Frontal (AFT por sus siglas en Inglés), o en generalización de la triangulación Delaunay (GDT) [14]. Las técnicas AFT construyen la malla generando de a uno los elementos a partir de una fron-

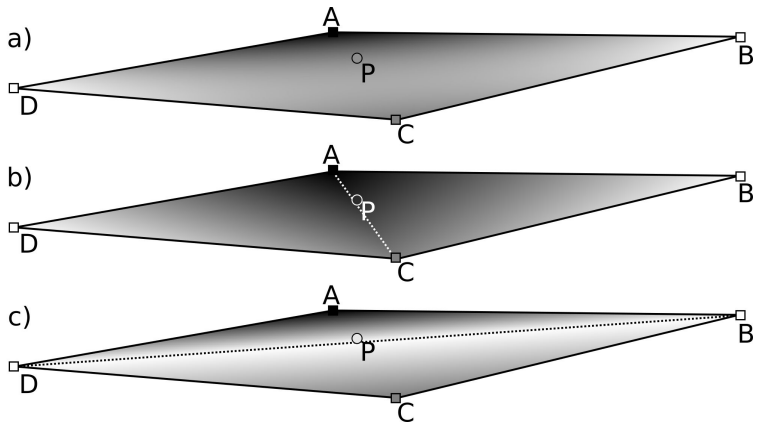

Figura 2.1: Tres formas de interpolar los valores (colores) asociados a los puntos A, B, C y D dentro del cuadrilátero que conforman: a) interpolación bilineal (a modo de referencia), b) y c) interpolaciones lineales, donde el valor de cada punto interior se obtiene como un promedio ponderado (utilizando coordenadas baricéntricas) de los tres puntos del triángulo en el que se encuentra. En c) se obtiene un valor en el punto P muy diferente a los valores de los puntos conocidos más cercanos (A y C). Esta es una de las razones por las cuales es conveniente evitar ángulos cercanos a 180°[13]

tera o superficie denominada "frente de avance", mientras que los métodos basados en GDT parten de una malla gruesa que abarca al dominio (usualmente un único elemento) y agregan nodos modificando las conectividades y refinando dicha malla hasta obtener el resultado deseado. Ambas técnicas son inicialmente seriales por naturaleza, pero se utilizan como base (o a veces como piezas) para la construcción de métodos de generación paralelizables.

2.1. Generación vs. triangulación

El problema de generación de mallas que se aborda en esta tesis consiste en encontrar un conjunto de conectividades (elementos) para un conjunto de nodos dado (tanto de frontera como interiores). Si bien el algoritmo a desarrollar tendrá en este caso comparativamente una responsabilidad menos (no deberá determinar la cantidad y las posiciones de los nodos interiores), esta situación no debe ser vista como una ventaja/facilidad, ya que limitará considerablemente las operaciones que el algoritmo podrá aplicar para generar la malla. Es decir, el algoritmo deberá respetar la restricción adicional de no mover, eliminar ni insertar nodos. Esto le impedirá utilizar soluciones simples a problemas frecuentes durante el mallado. Por ejemplo, puede ocurrir que en algún paso intermedio durante la generación de la malla, una porción del dominio que aún no ha sido cubierta por elementos presente una frontera tal que

no existan conectividades posibles para generar una malla válida y/o aceptable en su interior utilizando solamente los nodos dados. En [15] se presentan ejemplos de configuraciones 3D irresolubles (no existen conectividades para las fronteras y los nodos dados que no generen elementos invertidos o parcialmente superpuestos), y en [16] se analiza el problema (en general NP-hard) de determinar si una configuración dada es o no resoluble. En otros casos, el problema podría ser teóricamente resoluble, a costa de generar elementos bien orientados y sin superposiciones, pero con calidades inaceptablemente bajas. Es decir, podría ser posible generar conectividades topológicamente correctas, pero elementos, por ejemplo, con ángulos muy cercanos a 0°y/o a 180°. En estas situaciones, agregar o mover un nodo podría solucionar el problema con relativa facilidad, pero dadas las restricciones del caso esto no es posible, y por ello el problema no tiene solución. El algoritmo deberá entonces deshacer algunos de los elementos ya generados que dieron lugar a dicha situación y generar un nuevo conjunto de conectividades diferentes en la periferia, para directamente evitar el problema.

Como se aclaró en el capítulo introductorio, esta variación del problema general de generación de mallas (a veces denominado triangulación/tetraedrización de un conjunto de puntos), tiene aplicación directa en múltiples problemas de ingeniería. Particularmente, en métodos de simulación basados en partículas, como algunas implementaciones de PFEM-2, donde los nodos de la malla tienen una correspondencia directa con las partículas simuladas; o también en el problema de la proyección de un conjunto de valores asociados a puntos de Gauss desde una malla inicial hacia una nueva malla modificada. En estos casos, las restricciones sobre las calidades esperadas de los elementos expuestas continúan siendo válidas, y se agrega la necesidad de respetar el conjunto de nodos de entrada (su cantidad y sus coordenadas).

2.2. Triangulación/Tetraedrización Delaunay

Un símplice es un elemento N-dimensional formado por $N + 1$ nodos. Esto es, un segmento en 1-D, un triángulo en una superficie, un tetraedro en un volumen, etc. Una triangulación es una malla de símplices 2D (triángulos representando una superficie), y una tetraedrización es una malla de símplices 3D (tetraedros representando un volumen). Para un conjunto de nodos dado (ya sea nodos generados por un algoritmo de mallado o impuestos como datos de entrada por el problema) existen en general múltiples triangulaciones o tetraedrizaciones válidas (como mallas). La tetraedrización conocida como Delaunay es aquella en la cual las esferas que circunscriben los nodos

de cada tetraedro no contienen ningún otro nodo en su interior. Análogamente, la triangulación Delaunay es aquella en la cual las circunferencias que definen los nodos de cada triángulo no contienen otros nodos en su interior (ver figura 2.2). Este tipo de triangulación presenta propiedades generalmente deseables para muchas aplicaciones en ingeniería, y en particular para interpolación[17][18][19][20][21].

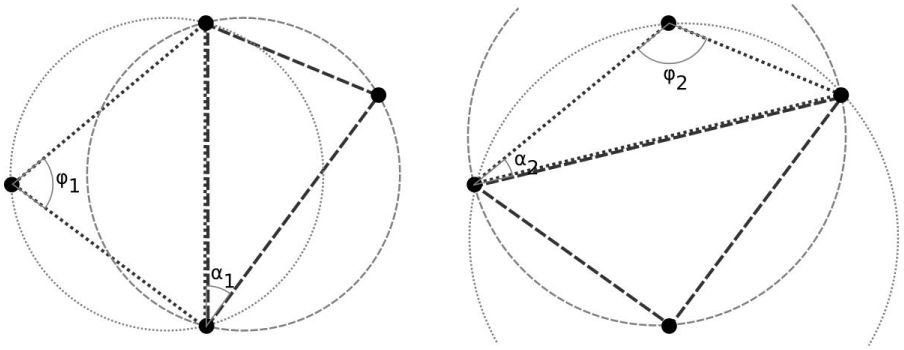

Figura 2.2: Dos posibles triangulaciones para el convex-hull de cuatro puntos. La triangulación de la izquierda cumple con el criterio Delaunay, mientras que la de la derecha no. Se indican en cada configuración los ángulos máximo(φ) y mínimo (α). Se verifica que $\varphi_1 <= \varphi_2$ y $\alpha_1 >= \alpha_2$.

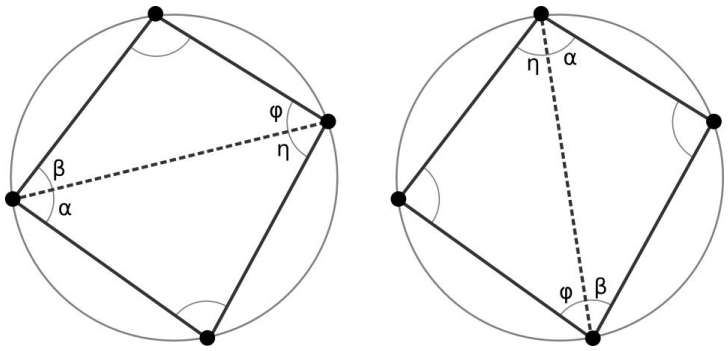

Figura 2.3: Dos posibles triangulaciones para el convex-hull de cuatro puntos cocirculares. Se señalan los ángulos equivalentes. Uno de ellos será el menor de la triangulación, por lo cual en ambos casos el mínimo ángulo es idéntico. Sin embargo, el máximo puede variar, ya que los 4 ángulos no etiquetados en general serán todos diferentes.

Cuando en una triangulación Delaunay no hay más de 3 nodos cocirculares, o en una tetraedrización no hay más de cuatro nodos coesféricos, se denomina a dicha distribución de nodos "general". En una distribución "general", la triangulación/tetraedrización Delaunay es única. Cuando esto no ocurre, se denomina a la distribución "degenerada". En este caso, el criterio

Delaunay es ambiguo, ya que hay más de una forma de triangular/tetraedrizar el conjunto de puntos cocirculares/coesféricos, generando en todos los casos círculos/esferas sin nodos en su interior (todos los nodos del conjunto estarán justo en la frontera de la esfera, ver ejemplo en figura 2.3). Este es el caso que se presenta en cada círculo o esfera cuando los nodos se ordenan en una grilla regular, distribuidos de forma equiespaciada en cada coordenada, y es una configuración de partida muy usual en muchas aplicaciones.

En un problema 2D la propiedad Delaunay puede considerarse en algún sentido garantía de calidad de forma en sus elementos. Esto se deba a que de entre las triangulaciones posibles para un conjunto de nodos, la triangulación Delaunay es la que maximiza el mínimo ángulo. Aunque esto no quiere decir que se minimice además el máximo ángulo[22], en general los ángulos de una triangulación Delaunay también presentan una cota superior adecuada como consecuencia (un ángulo muy grande en un triángulo, implica que sus dos ángulos restantes son pequeños).

Esta propiedad no es directamente transferible al problema 3D. La propiedad Delaunay garantiza en realidad una cota para la razón entre el radio de la esfera y la arista más corta de un elemento. Esto se traduce en una cota inferior para el mínimo ángulo en 2D, pero no en 3D [23]. En 3D los ángulos diedros pueden resultar muy pequeños cuando la densidad de nodos en el espacio no es uniforme.

Es común que una tetraedrización Delaunay contenga un número relativamente bajo, pero aún significativo, de *slivers* en su interior. Un *sliver* es un elemento con dos aristas opuestas muy próximas entre sí, cuyos nodos son aproximadamente cocirculares. Esto provoca errores numéricos en el cálculo del centro de la esfera y es la principal fuente de falta de robustez en algoritmos de mallado. Es además un tipo de elemento usualmente inaceptable para el cálculo, por presentar ángulos interiores muy cercanos tanto a 180°como 0°. Otro tipo de elemento de muy baja calidad y que frecuentemente se genera en las fronteras de mallas Delaunay 3D, son los conocidos como *caps*. En un *cap* también se encuentran 4 nodos aproximadamente coplanares, pero en este caso distribuidos de forma que uno de ellos se encuentre cercano al centro del triángulo definido por los otros tres. Esto implica una circunsfera muy grande (tiene su centro muy alejado del centro del elemento) y por ello, en una distribución regular de nodos, usualmente se encuentran solo en la frontera. La figura 2.4 muestra ejemplos de diferentes elementos junto con sus esferas.

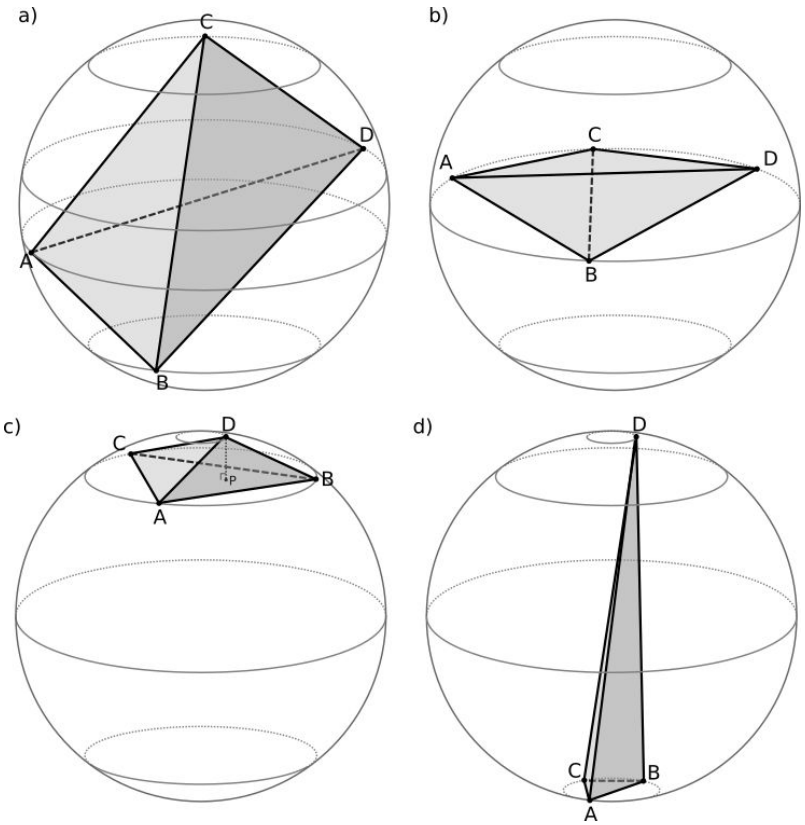

Figura 2.4: Distintos tipos de elementos y sus esferas. En b) los cuatro nodos son cocirculares (*sliver*). En c) el nodo D se acerca al triángulo ABC (*cap*), lo cual puede resultar en un tetraedro con una relación de aspecto inadecuada (altura ($|D - P|$) pequeña, y esfera de gran radio, en comparación con las longitudes de las aristas). En d) el elemento resultante también presenta una relación de aspecto inadecuada, pero el radio de su esfera es comparable a la longitud de sus aristas (*needle*).

Este problema se agrava además al considerar los errores numéricos en los cálculos involucrados. La triangulación Delaunay no es una función continua del conjunto de puntos de entrada. Una pequeña variación en la posición de un punto puede modificar la triangulación (por ejemplo, generar un swap de diagonales en 2D, ver figura 2.5). Cuando los puntos no se encuentran en posición general un conjunto de más de $N + 1$ puntos coesféricos en N dimensiones puede generar diferentes configuraciones de acuerdo a detalles de implementación tales como el orden de los cálculos, por motivos asociados a la precisión numérica. Esto suele ser una fuente de problemas y falta de robustez en muchos algoritmos de mallado.

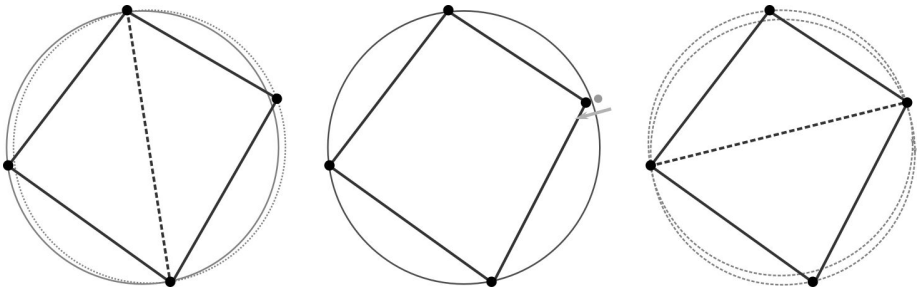

Figura 2.5: Una pequeña perturbación en la posición de un nodo (real o numérica) puede producir un cambio en la triangulación generada.

Por último, la tetraedrización Delaunay de un conjunto de puntos cubre con sus elementos todo el convex-hull de dicho conjunto. Cuando se quiere tetraedrizar un dominio arbitrario, cuya frontera (discretizada en una malla de contorno/superficie) no coincide con el convex-hull del conjunto de puntos, se está en presencia de una variante del problema conocida en general como "constrained Delaunay"[17]. En este caso se fuerza la presencia de ciertas conectividades dentro de la malla resultante (para que su frontera coincida con la impuesta) que pueden no ser parte de la verdadera configuración Delaunay. Por lo tanto, en general, una malla Delaunay-constrained no es una malla 100 % Delaunay.

Este tipo de mallas, además de presentar slivers en su interior, puede presentar también caps en su frontera. Muchos algoritmos de generación de mallas sacrifican la calidad de sus elementos en pos de obtener mayor robustez y/o velocidad, permitiendo la generación de caps y/o slivers, y añadiendo luego una etapa de postproceso en la cual se intenta remover estos elementos problemáticos mediante diferentes técnicas de edición local, usualmente basadas en plantillas[24][25][26][27][28].

2.3. Generación de triangulaciones Delaunay

Los métodos más utilizados para la generación de triangulaciones Delaunay son en general variaciones del algoritmo de Bowyer-Watson[29][30]. Este es un algoritmo incremental. En cada paso del algoritmo, se parte de una triangulación Delaunay válida y se inserta un nodo nuevo en su interior. Para ello se debe identificar y reemplazar un conjunto de elementos denominado "cavidad". Este conjunto se forma con los elementos de la triangulación previa cuyas esferas contienen al nodo que se inserta en ese paso. Es decir, los elementos que dejarán de ser válidos según el test de la esfera Delaunay luego

de insertar el nodo (ver figura 2.6). El conjunto es reemplazado por un nuevo conjunto de elementos formados uniendo caras de la frontera de la cavidad con el nodo insertado. Se puede demostrar que en posición general el conjunto es simplemente conexo y bien orientado, de modo que los elementos generados nunca serán elementos invertidos[31][32]. Al comenzar el algoritmo, se debe generar una primera triangulación trivial que abarque todo el dominio del problema. Se utiliza generalmente un tetraedro o cubo virtual cuyos nodos deberán ser removidos al finalizar el proceso. El tiempo de ejecución de este algoritmo está dominado por las operaciones de búsqueda utilizadas para determinar la cavidad correspondiente en cada inserción. Utilizando estructuras de búsqueda y ordenamiento espacial adecuadas se pueden obtener tiempos $O(n \log(n))$ para un conjunto arbitrario de n puntos.

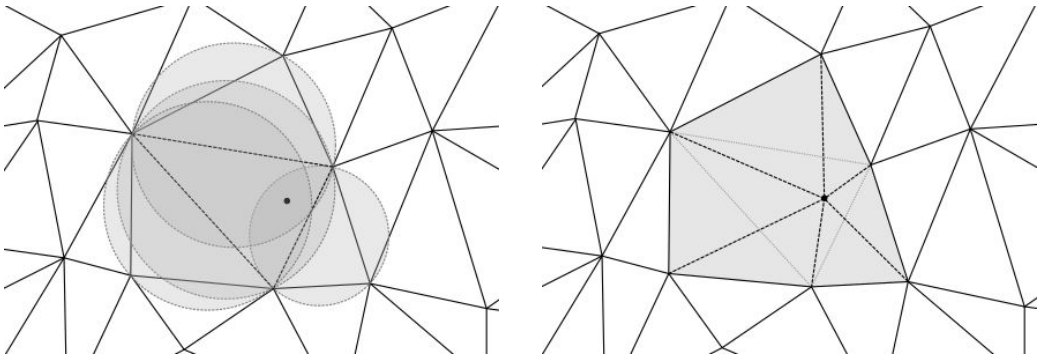

Figura 2.6: Inserción de un nodo en una triangulación Delaunay. La cavidad que se modifica al insertar un nodo corresponde a los cuatro triángulos cuyas circunferencias contienen a dicho nodo.

Las características claves que diferencian a las múltiples variaciones de este algoritmo incluyen la robustez ante conjuntos de punto en posición no general (debido a la sensibilidad frente a errores numéricos en los cómputos involucrados en la determinación de la cavidad), los diferentes métodos de ordenamiento y búsqueda utilizados para reducir el tiempo de ejecución, y la posibilidad de respetar una frontera impuesta, ya sea agregando restricciones en el algoritmo para generar las conectividades necesarias, o añadiendo una segunda etapa en la cual se modifican las conectividades generadas por el algoritmo original para transformar la frontera del convex-hull en la frontera impuesta. En estas últimas versiones, lo que generalmente distingue a los diferentes algoritmos no es la calidad de la malla generada, sino la capacidad de obtener triangulaciones válidas para mallas de frontera impuesta de baja calidad[33].

Una triangulación 2D puede modificarse fácilmente para respetar una

frontera impuesta utilizando la operación de edición local conocida como "swap de diagonales". Se toman dos triángulos adyacentes como si fueran un cuadrilátero, donde la arista común a ambos triángulos constituye una diagonal. La operación de "swap de diagonales" consiste en utilizar la otra diagonal del cuadrilátero para dar origen a dos nuevos triángulos que reemplazan a los originales. Se debe garantizar que las diagonales se crucen para que esta operación no genere elementos invertidos. En 3D, estas operaciones resultan mucho más complejas aún, y no siempre pueden utilizarse para restaurar la frontera impuesta sin añadir nuevos nodos a la malla[34][35][36][37][38].

Un algoritmo alternativo es el que se conoce como algoritmo de Tanemura-Ogawa-Ogita[39][40][41]. La iteración principal de este algoritmo toma como entrada una arista y el conjunto de puntos y determina cual de los puntos restantes genera una esfera vacía. De esta forma, va procesando y generando nuevas aristas, modificando en cada paso la frontera de forma similar a como lo hace un mallador de avance frontal. Este tipo de algoritmos se adaptan más fácilmente que los basados en Bowyer-Watson a la versión constrained de la triangulación Delaunay. Es decir, permiten generar una frontera impuesta, evitando generar elementos que no respeten dicha frontera modificando el criterio de selección, en lugar de requerir de una segunda etapa de recuperación de frontera. Sin embargo, se utiliza usualmente en problemas 2D, pero no así en 3D. En 3D, se pueden encontrar configuraciones de frontera irresolubles (arribar a fronteras interiores que no pueden resolverse sin agregar o mover nodos). El algoritmo del cual se parte en el capítulo 5 y que se extiende luego en el capítulo 6, podría considerarse una variación más dentro de esta familia de algoritmos. En dichos capítulos se describirán con más detalles los problemas asociados a este tipo de generación de mallas y triangulaciones, y las modificaciones y algoritmos alternativos necesarios para resolver además el problema de la generación con frontera impuesta en 3 dimensiones.

Existen muchos otros algoritmos que no pueden asociarse a ninguna de las dos grandes familias mencionadas. Muchos de estos algoritmos tienen por el momento solo utilidad teórica, ya que no son utilizados en la práctica para problemas reales. Una excepción, podría ser la familia de algoritmos que se basa en la generación de un convex-hull. Se puede obtener la triangulación Delaunay de un conjunto de puntos N dimensionales generando un convex-hull de un conjunto de puntos $N + 1$ dimensionales generado a partir del primero, y re-proyectando a N dimensiones las caras del convex-hull resultante. Esta relación es además útil para analizar y/o demostrar algunas propiedades de la triangulación Delaunay[17].

2.4. Generación de mallas y triangulaciones en paralelo

Las arquitecturas paralelas más comúnmente utilizadas para resolver estos problemas pueden clasificarse en dos grandes grupos: arquitecturas de memoria compartida, y arquitecturas de memoria distribuida. La diferencia más importante entre ambas se encuentra en el mecanismo de acceso a memoria por parte de los distintos procesadores. En el primer caso, todos los procesadores acceden directamente a una misma memoria común (Ej: PCs multi-core), mientras que en el segundo cada procesador tiene su propia memoria local, y debe comunicarse por red con los demás para compartir información (Ej: clusters). Esta distinción será la más importante al clasificar en forma general los diferentes algoritmos de generación de mallas, ya que determina el volumen de comunicación requerido por un algoritmo en cada tipo de arquitectura y el nivel de sincronización necesario para garantizar que los intentos de acceso simultáneos a las estructuras de datos involucradas no generen problemas. Se describirán con más detalle esta clasificación, los mecanismos de comunicación y sincronización, los errores más usuales, y otros aspectos técnicos a tener en cuenta en el capítulo 3.

El incremento del poder de cómputo mediante el paralelismo puede aprovecharse para la simulación computacional de problemas de mayor tamaño. Cuando el dominio de simulación crece, crece el tamaño de la malla, y con ella el tiempo de generación de la misma y el uso de memoria para almacenarla. Eventualmente, una malla podría no caber en la memoria de un procesador, sino estar distribuida en múltiples memorias locales a diferentes procesadores. Un algoritmo de generación de mallas para este tipo de problemas estará diseñado casi exclusivamente para una arquitectura de memoria distribuida, y el costo computacional a pagar por la paralelización estará generalmente asociado a la comunicación entre los diferentes procesadores. Por otra parte, la mayor potencia de cómputo también puede aprovecharse para reducir el tiempo de ejecución requerido para un determinado problema, manteniendo su tamaño. En este caso, podrían utilizarse ambas arquitecturas, aunque generalmente será más simple y redituable (a igual número de procesadores) utilizar arquitecturas de memoria compartida. El costo computacional añadido estará generalmente relacionado a los mecanismos de sincronización del acceso a esa memoria compartida.

Para generar mallas no estructuradas aprovechando arquitecturas paralelas es necesario descomponer el problema. Esta descomposición del problema puede realizarse mediante dos enfoques [42]:

1. descomposición de las estructuras de datos de la malla
2. descomposición del dominio geométrico

El primer enfoque suele ser más directo y simple de implementar en arquitecturas de memoria compartida, resultando en una disminución del tiempo de ejecución, invirtiendo relativamente poco esfuerzo de rediseño e implementación, pero usualmente limitado a un bajo número de procesadores. No será una solución escalable a otras arquitecturas con mayor número de procesadores o de memoria distribuida. Las soluciones del segundo grupo utilizan el concepto de partición del dominio. Esto significa que el algoritmo divide el dominio geométrico del problema en subdominios de menor tamaño y complejidad, para resolver cada uno de los subdominios en un procesador diferente. El algoritmo debe requerir la menor interacción (comunicación) posible entre los subdominios (procesadores) para que la estrategia cumpla su objetivo. Sin embargo, algún mecanismo previo, o del mallado de los subdominios, debe garantizar la compatibilidad entre las mallas generadas. Esto es, dos subdominios adyacentes de una malla de volumen deben respetar una misma interfase (malla de frontera) en la superficie común a ambos. Otra forma usual y útil de clasificar los diferentes métodos de generación en paralelo, se basa en cómo y cuándo se resuelve este problema. Podemos definir entonces tres grupos[43]:

1. Aquellos que mallan la interfaz al mismo tiempo que mallan los subdominios
2. Aquellos que primero definen las mallas de interfaces, y luego mallan los interiores de los subdominios
3. Aquellos que primero mallan los subdominios, y luego "arreglan" las interfases generadas

Los métodos del primer grupo pueden requerir mayor comunicación durante el mallado de los subdominios (fuertemente acoplados), o depender de algún criterio geométrico que garantice la unicidad global del resultado (por ejemplo, el criterio Delaunay cuando los puntos se encuentran en posición general). Los del segundo requieren en general una etapa de preproceso (la generación y el mallado de las interfaces) que puede no ser paralelizable o distribuible). Finalmente, en los métodos del tercer grupo la etapa de "merging" (como se denomina a la etapa en donde se mallan o arreglan las interfaces) suelen ser muy costosas (la mayoría de los casos donde se resuelve fácilmente son casos 2D, sin extensión directa a 3D). Esta dos últimas categorías presentan entonces una etapa de trabajo desacoplada y de alta eficiencia paralela (el mallado de los subdominios), y otra etapa que presenta un nivel de acoplamiento mucho mayor, que hasta podría ser necesario resolver en serie (mallado de las

interfases, o merging).

Entonces, para sacar el máximo provecho de una arquitectura paralela genérica, sería deseable obtener una estrategia basada en la descomposición del dominio, donde los subdominios puedan resolverse de forma desacoplada, y donde además las etapas de partición y merging no representen un costo computacional excesivo. Más aún, para aprovechar los desarrollos preexistentes ya probados para generación de mallas en serie, muchos métodos (como por ej: [14]) proponen mecanismos de partición suficientemente generales como para permitir la reutilización de los algoritmos serie para la solución de cada interfaz o de cada subdominio. Otros métodos, buscan enfoques alternativos basándose en algoritmos que no han demostrado ser suficientemente competitivos para la generación en serie, pero que podrían presentar características más adecuadas para la paralelización, como por ejemplo la generación natural de subproblemas desacoplados[44] a cambio de un mayor costo computacional promedio por elemento. En [45] se puede encontrar una reseña de los métodos más usuales de generación de mallas no estructuradas en paralelo, organizados por su nivel de acoplamiento y/o por la técnica secuencial en la que se basan. En todos los casos existe un compromiso entre el nivel de acoplamiento y la eficiencia paralela que se puede obtener, siendo este aún un problema abierto.

Capítulo 3

Programación en Paralelo

La computación paralela es una forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente (paralelismo a nivel de instrucciones), y/o muchos datos se procesan simultáneamente de igual forma (paralelismo de datos). Ha tomado mayor relevancia en la última década debido a las nuevas tendencias en el diseño de hardware comentadas en la introducción. Estos cambios, a diferencia de muchos otros en la evolución del hardware y su historia, obligan al programador a repensar sus algoritmos para ser capaz de aprovechar esta posibilidad. Ahora, la responsabilidad de incrementar el poder de cómputo del sistema es compartida por los desarrolladores de hardware y de software, ya que un programa desarrollado para un modelo en serie, en general no se verá beneficiado por la disponibilidad de unidades de procesamiento adicionales (más núcleos en una PC). El desarrollador debe entonces tener en cuenta la arquitectura del hardware para el cual escribirá su código a la hora de diseñar las soluciones. Además, en general es más difícil escribir programas que operen en paralelo, ya que se introducen nuevos tipos de errores y limitaciones propias de estas arquitecturas.

En la actualidad existen varios tipos de arquitecturas de hardware paralelo, desde aquellas presentes en PCs hogareñas y otros dispositivos inteligentes de uso cotidiano (como por ejemplo smartphones y tablets, aún en productos de gama media) hasta las denominadas supercomputadoras, formadas por clusters con miles de nodos, por sus altos costos sólo disponibles en centros de investigación específicos. Cada escala presenta sus propias particularidades, y aún dentro de una misma escala hay disponibles arquitecturas muy diferentes, a veces hasta combinadas en un mismo dispositivo. En este capítulo se describen conceptos básicos sobre paralelismo y concurrencia, se presenta una clasificación relativamente gruesa de los tipos de arquitecturas parale-

las más usuales, y se detallan los principales problemas a los que se debe enfrentar un desarrollador al implementar una solución en dichas arquitecturas. El capítulo no pretende ser un compendio exhaustivo, sino introducir los conceptos básicos necesarios para comprender las decisiones de diseño e implementación de los algoritmos que se presentarán en los capítulos 5 y 6, y ahondar solo en aquellos problemas y detalles que resulten relevantes para dichos algoritmos.

3.1. Concurrencia y paralelismo

Concurrencia y paralelismo son dos conceptos fundamentales que frecuente y equivocadamente se utilizan como sinónimos. Existe una relación estrecha entre ambos, pero no son equivalentes. En un sistema concurrente existen múltiples tareas que deben ser ejecutadas en simultáneo, y que interactúan entre sí. Por ejemplo, es común que en muchas aplicaciones actuales toda la interfaz gráfica se ejecute en un hilo de ejecución, y los verdaderos cálculos que la aplicación realiza se ejecuten en otro. De esta forma, se consigue que la interfaz responda en tiempo real a los requerimientos del usuario en todo momento, independientemente del trabajo que esté efectivamente realizando la aplicación gracias al otro hilo por detrás. Ambos hilos se comunican entre sí, ya que la interfaz envía los trabajos que solicita el usuario al hilo de trabajo, y el hilo de trabajo envía los resultados a la interfaz para que los presente al usuario. Sin embargo, no es necesario que ambos hilos se ejecuten efectivamente a la vez en diferentes procesadores o diferentes núcleos de un mismo procesador, sino que pueden ejecutarse en un mismo núcleo utilizando time-sharing (se ejecutan un tiempo cada uno, siendo el tiempo lo suficientemente pequeño como para que parezca que ambos hilos avanzan al mismo tiempo). Sin embargo, sí es necesario que al menos se provea la ilusión de que se ejecutan en simultáneo, ya que si se ejecuta solo uno de ellos durante un largo período de tiempo el programa no puede funcionar correctamente.

El paralelismo, en cambio es un concepto dependiente del hardware, que se utiliza cuando dos hilos o procesos se ejecutan efectivamente a la vez en dos núcleos o procesadores diferentes. La concurrencia entonces se refiere a tareas y componentes que funcionan lógicamente en simultáneo e interactúan entre sí para cumplir un objetivo. La paralelización, en cambio, busca explotar las posibilidades del hardware cuando este presenta más de un núcleo. Se pueden ejecutar en paralelo tareas completamente independientes, como también se puede por ejemplo pensar en la ejecución paralela de varias tareas concurrentes de un mismo programa. Cuando un problema complejo se sepa-

ra en diferentes etapas o subproblemas concurrentes, y que pueden resolverse efectivamente en paralelo, para sacar el máximo provecho de esta paralelización, se requiere que la interacción necesaria entre esas tareas concurrentes se reduzca al mínimo posible. Esto se debe a que en general la interacción debe incluir mecanismos de sincronización para evitar ciertos problemas típicos de la programación paralela.

3.1.1. Tipos de paralelismo

Se pueden clasificar las estrategias de paralelización básicas en tres grandes grupos: fork-join, vector/SIMD y pipeline[46]:

- Fork-Join: El paralelismo puede verse como una propiedad teórica de un algoritmo[46]. Se pueden representar las tareas/pasos de un algoritmo como nodos en un grafo orientado, donde las aristas representan las dependencias entre ellas. El grafo comienza en un nodo, y finaliza en otro, pero contiene múltiples caminos para avanzar desde el nodo inicial al nodo final. Cuando un camino se bifurca, las tareas comprendidas entre el punto en el que se bifurcan (fork) y el punto en que se vuelven a unir (join) pueden resolverse en paralelo. Idealmente, un algoritmo óptimo generará las bifurcaciones necesarias en el primer paso, y los caminos solo se unirán al final del proceso, logrando así mantener todos los procesadores ocupados durante la mayor parte del tiempo de ejecución. En este tipo de paralelismo, se requiere que cada procesador pueda realizar tareas arbitrarias independientemente de los demás. Es el tipo de paralización que más comúnmente realiza un programa al utilizar hilos o procesos en los sistemas operativos actuales sobre hardware multi-core.

- Vector/SIMD: Single Instruction Multiple Data (SIMD) hace referencia a una arquitectura paralela en la cual un procesador puede ejecutar en simultáneo una misma operación sobre más de un dato. Si bien Vector no es exactamente un sinónimo de SIMD, para una clasificación general puede incluirse en la misma categoría. Desde hace ya muchos años los procesadores de PCs incluyen en sus juegos de instrucciones subconjuntos de instrucciones vectoriales, que realizan este tipo de operaciones. En general, los lenguajes de programación no exponen estas instrucciones directamente al programador. El programador debe utilizar extensiones específicas, o bien un compilador capaz de detectar automáticamente partes del código vectorizables y hacerlo sin requerir

ninguna instrumentación. En los últimos años se ha progresado mucho en este sentido y hoy en día los compiladores más utilizados son capaces de lograr esto en muchas situaciones. Por otro lado, las arquitecturas SIMD son actualmente las más usuales en las placas gráficas, que además ya no están dedicadas exclusivamente a la visualización, sino que pueden programarse para utilizarse como procesadores de propósito general (GPGPU). A diferencia de la vectorización de las CPUs, en la mayoría de los casos el programador debe utilizar lenguajes y herramientas específicas (a veces dependientes del fabricante) para aprovechar la potencia de estos dispositivos en operaciones no-relacionadas a la representación de gráficos.

- Pipeline: Esta estrategia de paralelización se aplica cuando el algoritmo se puede descomponer en una secuencia lineal de etapas que actúan sobre un flujo de datos, donde cada etapa toma como entrada, solamente la salida de la etapa anterior. Cada etapa se asocia a un proceso, entonces mientras la segunda etapa procesa por ejemplo el primer dato, la primer etapa puede comenzar a procesar el segundo (similar a una cadena de montaje). Desde hace muchos años este tipo de paralelismo también se aplica de forma transparente dentro de los microprocesadores, donde el flujo de entrada se conforma con las instrucciones de un programa, y las etapas son los diferentes pasos que debe realizar el procesador para ejecutar una instrucción. Pero no siempre es posible, ya que la presencia de una instrucción que implique un salto condicional obliga a esperar a que la condición se resuelva antes de saber cuál es la siguiente instrucción a procesar. Si la arquitectura logra de alguna forma predecir el resultado de la condición antes de evaluarla, puede evitar este problema. La mayoría de los microprocesadores presentan mecanismos de predicción que les permiten comenzar a ejecutar una de las posibles ramas de ejecución de un condicional antes de evaluarlo, y deshacer esos cambios en caso de que posteriormente se determine que la predicción fue incorrecta. Se conoce a estos mecanismos como mecanismos de *ejecución especulativa*.

Se puede encontrar mencionada en la bibliografía una categoría adicional denominada “paralelismo de grafo”, que se ajusta a la descripción de un algoritmo como grafo presentada anteriormente, donde puede haber varios puntos de bifurcación y unión, aún dentro de caminos paralelos, y dependencias entre nodos cualesquiera, incluyendo entre nodos intermedios de diferentes caminos. Para poder ejecutar una tarea (nodo del grafo), antes debe cumplirse un conjunto de pre-requisitos modelados como resultados de tareas previas

de las cuales depende (arcos). Esta descripción es suficientemente amplia como para considerarse una generalización y abarcar al paralelismo fork-join y pipeline a modo de casos particulares. Se debe notar que si dos caminos diferentes tienen un conjunto de tareas iniciales comunes (es decir, comienzan en el mismo nodo y luego se bifurcan), además de poder resolverse una única vez en uno de ellos y comunicar luego el resultado al otro, estas pueden también resolverse en simultáneo e independientemente en cada procesador, duplicando el esfuerzo pero evitando la interacción. Si las tareas en cambio se encuentran al final del mismo, dependiendo directa o indirectamente de nodos de ambos caminos (es decir, los caminos se unen en algún punto), la interacción es inevitable. Para obtener entonces el máximo provecho del hardware disponible, al diseñar un algoritmo se debe intentar evitar esta última situación siempre que sea posible.

3.2. Arquitecturas de hardware paralelo

Para desarrollar un software paralelo se debe tener en cuenta el tipo de hardware sobre el que este va a ejecutarse. La variedad de hardware capaz de ejecutar programas en paralelo ha aumentado considerablemente en los últimos años, y como ya se mencionó anteriormente, esto no se limita solamente al mercado de PCs y estaciones de trabajo de alto rendimiento o servidores, sino que incluye otros dispositivos de uso diario como por ejemplo tablets y smartphones. En aplicaciones relacionadas a la mecánica computacional, el hardware utilizado suele consistir en PCs/workstations (estaciones de trabajo de alto rendimiento) multi-procesador y/o multi-core, clusters, y en menor medida (aunque en crecimiento) GPUs.

Desde el punto de vista del programador, y en relación a las técnicas de paralelización, la diferencia entre las arquitecturas de las PCs de escritorio actuales y notebooks, con respecto a las de equipos más potentes como servers (servidores) y workstations ha ido disminuyendo, tornando la barrera que los separa cada vez más difusa. Cualquier procesador de escritorio o notebook actual es en realidad un procesador multi-core (un único circuito integrado, pero que contiene dentro el hardware equivalente a múltiples procesadores), y los procesadores multi-core más modernos están comenzando a presentar características que antes eran exclusivas de configuraciones multi-procesador (más comunes en servers y workstations). Se podría observar que los equipos que se presentan como personales contienen un número bajo de núcleos (típicamente de 2 a 8 núcleos), mientras que el hardware más potente diseñado para ser utilizado como workstation o server presenta un número de núcleos

superior (aproximadamente un orden de magnitud superior).

En ambos casos, la arquitectura general responde en gran medida al modelo von Neumann, lo cual evita muchos problemas al diseñar el software, garantiza en muchos aspectos la compatibilidad hacia atrás (que nuevo hardware pueda continuar ejecutando viejo software), y permite una transición más suave a este *nuevo* paradigma de desarrollo. El programador dispone entonces de múltiples núcleos o procesadores en los cuales ejecutar en simultáneo diferentes hilos de ejecución o diferentes procesos, pero todos conectados a una única memoria principal común. Esto hace que pueda ser relativamente simple y poco costoso (computacionalmente) intercambiar datos entre diferentes hilos de ejecución. Se denomina a este tipo de arquitecturas como "de memoria compartida". Sin embargo, esta ventaja trae asociada a su vez una limitación en el número de núcleos que efectivamente se pueden aprovechar. Dado que todos los hilos acceden a una misma memoria principal, la velocidad del bus de acceso a dicha memoria pasa a ser un factor clave que genera un cuello de botella en muchos algoritmos. Esto obliga en muchos casos a repensar las estrategias de paralelización, e incrementa la importancia de utilizar estructuras de datos compactas y patrones de acceso a la memoria secuenciales para aprovechar al máximo las ventajas que ofrece la presencia de diferentes niveles de memoria cache. La tabla 3.1 presenta tiempos de acceso a los diferentes niveles en la jerarquía de memoria representativos de una PC actual en función de los ciclos del procesador:

1 cycle on a 3GHz processor	∼1 ns
L1 CACHE hit	∼4 cycles
L2 CACHE hit	∼10 cycles
L3 CACHE hit, line unshared	∼40 cycles
L3 CACHE hit, shared line in another core	∼65 cycles
L3 CACHE hit, modified in another core	∼75 cycles
remote L3 CACHE	∼100-300 cycles
Local Dram	∼60 ns
Remote Dram	∼100 ns

Tabla 3.1: Comparativa de tiempos típicos de acceso a datos según su ubicación y estado en la jerarquía de memoria de una PC[47][48]. En un procesador actuales un ciclo requiere 1ns.

Existen además otras tecnologías tendientes a aumentar el paralelismo en este tipo de hardware. Algunas tecnologías, por ejemplo, buscan exponer al software (al sistema operativo) un número de núcleos mayor al número real, para ejecutar virtualmente más de un hilo de ejecución en simultáneo por

cada núcleo. Para ello, se utilizan a nivel de hardware estrategias similares al time-sharing que aplica el sistema operativo, pero aprovechando cierta instrumentación ad-hoc por parte del procesador, que además suele incorporar por duplicado algunos recursos clave para obtener así un mejor rendimiento. El ejemplo más conocido y utilizado de este tipo de tecnologías es Intel Hyper Threading, que logra bajo ciertas configuraciones de carga entre un 20 % y un 30 % de rendimiento adicional por cada hilo virtual[49].

Por otro lado, otra forma de aumentar el paralelismo dentro de una misma PC consiste en la utilización de hardware adicional con procesadores (y eventualmente también memoria) propios, como lo son las placas gráficas. La creciente demanda de complejidad y detalle en los gráficos de videojuegos y software multimedia en general, ha logrado que las placas gráficas también incorporasen capacidades de paralelización, más aún considerando que muchas de las operaciones que estas placas realizan al renderizar una escena son naturalmente paralelizables. Luego, su mayor complejidad y capacidad de cómputo, debido en parte también a la necesidad de reprogramar sus funcionalidades aún para el renderizado, ha generado la posibilidad de aprovechar estos recursos ya no solo para la generación de gráficos en 3D, sino también para el cálculo en general (por eso se las denomina GPGPUs: General Purpose GPUs). Sin embargo, estos procesadores presentan ciertas particularidades que hacen más difícil, o específica, su programación. Esto se debe en parte al tipo de operaciones que se requieren de una GPU, pero también a que los fabricantes de GPUs tienen menos requerimientos asociados a la compatibilidad hacia atrás al diseñar nuevas generaciones de hardware. Sin embargo, junto a esta mayor velocidad de evolución, está también asociada una mayor velocidad de cambio en las plataformas de software sobre las cuales se desarrolla para estos dispositivos, ya que requieren de lenguajes de programación específicos (en muchos casos asociados a un fabricante en particular), técnicas de programación en paralelo diferentes (son arquitecturas SIMD), mayor atención en detalles relacionados a la gestión de la memoria, etc. No se ha logrado todavía una convergencia completa en este sentido que garantice que un nuevo desarrollo de software para este tipo de hardware en la actualidad siga siendo válido transcurridos, por ejemplo, 5 años. Existen algunas extensiones para lenguajes como C++ que buscan evitar esto (por ejemplo: C++ AMP[50]) pero lo logran parcialmente y solo para ciertas configuraciones de hardware particulares.

Finalmente, un cluster consiste en un conjunto de PCs o workstations comunicadas entre sí, que trabajan en conjunto de forma que, desde cierto punto de vista, pueden verse como un único sistema. Se denomina nodo a cada una de las PCs o workstations que conforman el cluster. Los nodos

pueden ser diferentes entre sí, incluir múltiples procesadores o procesadores multi-core, y hasta GPGPUs. De esta forma, el hardware del cluster puede ser heterogéneo, requiriendo distintos niveles y técnicas de paralelización para explotar su máximo potencial, configuraciones denominadas híbridas o mixtas. Desde el punto de vista del diseño de un algoritmo y su implementación, una particularidad muy importante de esta arquitectura es que cada nodo tiene su propia memoria RAM, y no existe una memoria global o compartida entre ellos. Por eso se denomina a este tipo de arquitecturas como "de memoria distribuida", y en ellas el mecanismo de comunicación entre los nodos suele ser determinante para su rendimiento. Si bien protocolos muy usados como TCP/IP sobre conexiones habituales como Ethernet pueden utilizarse para comunicar los nodos de un cluster, se prefieren tecnologías alternativas de mayor rendimiento como por ejemplo Infiniband.

Existen múltiples formas de caracterizar y categorizar las diferentes arquitecturas que permiten la ejecución de programas con algún grado de paralelismo. De todas ellas, por las características de este trabajo es de especial interés realizar la clasificación en base a los mecanismos de acceso a memoria, lo cual da lugar a las dos categorías ya mencionadas: arquitecturas de memoria compartida, y arquitecturas de memoria distribuida.

3.3. Programación en paralelo

Existen varios lenguajes, extensiones, toolkits y bibliotecas para aprovechar los recursos de una arquitectura de hardware paralela. La aplicación de cada uno depende del tipo de hardware que se busca aprovechar. Para arquitecturas de memoria distribuida, se suelen utilizar bibliotecas que ofrecen interfaces de alto nivel para las tareas de comunicación entre procesos en diferentes PCs de una red o cluster, como por ejemplo Parallel Virtual Machine (PVM), Message Passing Interface (MPI), y sus variantes. Para controlar el paralelismo de tareas (tipo fork-join, o de grafo) en una PC multi-core, existen alternativas de relativamente bajo nivel y dependientes del sistema operativo como pthreads, y abstracciones modernas y portables como la biblioteca Boost, o la biblioteca estándar de C++11[1]. Para un paralelismo intermedio (como el denominado paralelismo de loops/fors), menos intrusivo pero aún

[1]Las funcionalidades de la biblioteca de C++11 ofrecen una interfaz orientada a la concurrencia, aunque estas características pueden ser utilizadas como base para construir con ellas una biblioteca con la cual implementar algoritmos paralelos. Esto se debe a que la biblioteca facilita la creación y destrucción de threads, procesos ambos muy costosos, pero ello puede utilizarse para gestionar un pool de threads en los cuales asignar tareas.

así portable existen bibliotecas que operan en conjunto con extensiones de los compiladores como Intel Cilk Plus y OpenMP (OMP). Estas extensiones en particular suelen aprovechar mejor las capacidades de vectorización del hardware, y actualmente están comenzando a incorporar la posibilidad de utilizar arquitecturas heterogéneas, aprovechando en ciertas configuraciones, por ejemplo, las GPGPUs de forma transparente. Algunas bibliotecas presentan facilidades combinadas de las últimas dos categorías, como Microsoft Parallel Pattern Library (PPL) o Intel Threading Building Blocks (TBB). Finalmente, cuando se trata de explotar al máximo el hardware particular de las GPGPUs, se implementan pequeños subalgoritmos denominados usualmente núcleos en lenguajes específicos como CUDA (de NVIDIA), OpenCL, o Microsoft DirectCompute (parte de Microsoft DirectX).

Las implementaciones desarrolladas como parte de esta tesis, y descritas en los siguientes capítulos, utilizan para arquitecturas de memoria compartida las facilidades de la biblioteca estándar de C++11, que permite controlar la creación y destrucción de hilos de ejecución, y proveen además de algunos elementos de sincronización útiles (que se describirán a continuación); y para arquitecturas de memoria distribuida la biblioteca MPI. Por la naturaleza del método en el cual se basa, al desarrollar los mecanismos de paralelización del algoritmo de mallado se hará énfasis en los mecanismos de paralelización a nivel de tareas (paralelismo fork-join), y no se considerará el uso de GPGPUs. De esta forma se describe el algoritmo paralelo de manera portable y general, sin depender de particularidades de un hardware específico, utilizando, en cambio, funcionalidades estables y disponibles en cualquier plataforma de software/hardware actual y futuro.

3.3.1. Mecanismos de sincronización

Como se observó previamente, la programación en paralelo y/o concurrente es en general más compleja y difícil que la programación *en serie*, ya que agrega nuevas dificultades, tanto al diseño de los algoritmos como a la implementación de los mismos. Los problemas de implementación más comunes están asociados al acceso en simultáneo a datos comunes por parte de dos o más hilos de ejecución diferentes. *En simultáneo* puede entenderse como el acceso efectivamente en paralelo por parte de dos hilos de ejecución, o concurrente utilizando mecanismos de time-sharing. Si un hilo se encuentra actualizando una estructura de datos, y otro hilo intenta consultar la misma, el segundo hilo podría observar un estado inconsistente producto de que la primera ha iniciado pero no completado su operación. Si el segundo hilo en

cambio modifica la estructura de datos, invalida el estado observado por el primer hilo al comienzo de su operación, invalidando entonces también el estado final que este intentará escribir producto de la misma. La manifestación de estos errores depende fuertemente del momento en que ambos hilos lleguen al punto de ejecución problemático, y esto, en un sistema operativo moderno, es imposible de controlar, ya que la velocidad con la que avanzan los hilos individualmente depende en gran medida de la política de planificación de tareas del sistema, y por ende de muchos factores externos como la carga que el sistema tenga debido a otros procesos durante la ejecución. Este tipo de situaciones se denomina *data-race* o *race condicion* (condición de carrera), y por estos motivos resultan muy difíciles de reproducir. Esto dificulta considerablemente no solo la detección de los mismos, sino también la depuración en general de programas paralelos.

Existen actualmente algunas herramientas de software destinadas a facilitar la detección de estos problemas, que se basan en instrumentar directa o indirectamente el programa para detectar el acceso simultáneo a una misma dirección de memoria por parte de dos hilos. Una instrumentación directa consiste en la introducción de código adicional específico en el ejecutable. En general el compilador es el encargado de esta tarea, lo que se conoce como *sanitizer* (por ejemplo, la funcionalidad de llvm+clang: thread-sanitizer). Una instrumentación indirecta utiliza algún tipo de máquina virtual para poder correr un ejecutable arbitrario sin modificaciones (por ejemplo, la herramienta hellgrind de la suite Valgrind). Aún utilizando este tipo de herramientas, el testing no es exhaustivo, ya que depende de que el problema pueda observarse durante una ejecución particular. Si una race-condition tiene muy poca probabilidad de manifestarse, podría no ser detectada.

Para evitar este problema se deben utilizar mecanismos de sincronización, que permitan garantizar que dos hilos no accederán a una sección crítica en simultáneo. A continuación se describen los mecanismos más usuales, disponibles en prácticamente cualquier combinación de plataforma y lenguaje o biblioteca para la programación en paralelo.

Mutexes

Los mutexes proveen un mecanismo para que un thread pueda bloquear la ejecución de otro[51]. La palabra *Mutex* deriva de Mutual Exclusion locks (candados de exclusión mutua). Un mutex es un recurso que no puede ser adquirido por más de un hilo de ejecución en simultáneo, e implementa internamente los mecanismos para garantizarlo. El mutex ofrece mínimamente

dos operaciones básicas, la adquisición del recurso por parte de un hilo, denominada lock, y la liberación del mismo, denominada unlock. Se dice que un hilo posee al mutex si ha finalizado correctamente la operación de adquisición/lock y aún no lo ha liberado/unlock. Si un hilo intenta adquirir un mutex mientras otro hilo tiene posesión del mismo, esta segunda operación de lock puede o bien demorar su finalización hasta que el primer hilo libere el mutex, o bien retornar un código de error indicando que no se ha podido realizar con éxito. La primera opción es la más usual, y se dice que el segundo hilo se bloquea, ya que su ejecución se detiene hasta que el primer hilo libere el recurso.

Para prevenir las race-conditions sobre una estructura de datos compartida, se asocia un mutex a la misma, y cualquier hilo que deba operar sobre esta debe antes adquirir el mutex, para asegurarse de ser la única que lo está haciendo en ese momento. Se debe notar que un mutex se encarga literalmente de anular el paralelismo en una sección particular del algoritmo (denominada sección crítica). Si en un algoritmo la probabilidad de que dos hilos accedan a la misma estructura de datos en simultáneo es baja, el mutex es un mecanismo útil y eficiente. Pero si esta probabilidad es alta, el uso de mutexes llevará a que la mayoría de los hilos de ejecución pasen la mayor parte del tiempo bloqueados a la espera de la liberación del recurso por parte de otro hilo (es decir, sin poder hacer trabajo útil), degradando notablemente la eficiencia paralela del algoritmo.

Finalmente, existe un problema adicional relacionado al uso de múltiples mutex para proteger el acceso a múltiples estructuras de datos, y es el que se conoce como deadlock. Un deadlock se produce cuando dos hilos se encuentran bloqueados a la espera de la liberación de dos mutexes que han sido adquiridos por los mismo dos hilos. Por ejemplo, sea H_A un hilo de ejecución que requiere adquirir en simultáneo los mutexes M_1 y M_2 para completar su operación y los intenta adquirir en ese orden, y H_B otro hilo que también requiere ambos mutexes, pero los intenta adquirir en el orden contrario. Puede ocurrir que mientras H_A comienza su operación adquiriendo M_1, en paralelo H_B también comience su operación adquiriendo M_2. En este caso, H_A no podrá continuar avanzando por que no podrá obtener M_2, y por lo tanto al no completar su operación, no liberará M_1. De igual forma, H_B no podrá obtener M_1 y tampoco liberará M_2. En esta situación, ambos hilos quedan bloqueados indefinidamente.

En conclusión, el uso de mutexes, si bien proporciona una solución simple al problema de las race-conditions, puede anular el paralelismo y (mal aplicado) hasta evitar la finalización del algoritmo. Por esto, es importante

diseñar los algoritmos de forma que eviten la necesidad de modificar datos compartidos o reduzcan al mínimo su probabilidad de ocurrencia, para que al implementarlos estos problemas no anulen las ventajas del uso de múltiples procesadores.

Semaphores

Un semáforo es un contador que puede ser utilizado para sincronizar múltiples hilos[51]. Un semáforo representa un contador sobre el que se pueden realizar de forma segura las operaciones de incremento (post) y decremento/espera (wait). El sistema operativo garantiza que estas operaciones se realizan de forma segura (sin que ello genere una race-condition). El contador nunca toma valores negativos. Si un hilo intenta decrementarlo (wait) cuando su valor es cero, éste se bloquea hasta que otro hilo lo incremente. De esta forma, un semáforo constituye una herramienta, para ciertas aplicaciones, más simple y directa que un mutex. Por ejemplo, puede utilizarse para sincronizar una cola de trabajos pendientes a resolverse en paralelo, donde el contador representa la cantidad de trabajos encolados. Algunos hilos resuelven trabajos, utilizando la operación wait para determinar cuando tomar un trabajo de la cola, mientras que otros los producen, utilizando la operación post para notificar a los primeros.

Condition-variables

Una condition-variable permite implementar una condición bajo la cual un hilo se ejecuta, e inversamente, una condición bajo la cual un hilo se bloquea [51]. Es decir, actúa a modo de bandera (variable lógica), para la cual las operaciones básicas de consulta y modificación, al igual que en el caso del semáforo, son seguras y pueden generar un bloqueo. Esta bandera representa el cumplimiento de una condición. Las operaciones básicas son señalización (signal) y espera (wait). La primera es la operación que permite modificar la bandera para indicar que la condición se cumple. La segunda consulta el estado, esperando a que la condición se cumpla en caso de ser falsa (es decir, bloqueando al hilo que hace la consulta), para luego marcarla nuevamente como falsa. Al igual que los semáforos, una condition-variable suele utilizarse para sincronizar hilos productores y consumidores de trabajo. La bandera representa la presencia de un trabajo pendiente. Cuando un hilo está ocioso realiza una operación de tipo wait para consultar si hay un trabajo pendiente. En caso negativo, el hilo se bloquea a la espera de un nuevo trabajo futuro. Cuando otro hilo coloca un trabajo en la cola, realiza una operación signal.

Esta operación dejará la bandera en verdadero si no hay hilos esperando, o desbloqueará alguno de los hilos en espera para que realice el trabajo (solo un hilo se despertará, puede ser cualquiera de los hilos en espera).

Operaciones atómicas

Las operaciones atómicas son operaciones que se ejecutan en un solo paso, razón por la cual no pueden ser interrumpidas u observadas en un estado intermedio y generar así una race-condition. Por ejemplo, el incrementar el valor de una variable regular (operación representada por una sola instrucción en lenguajes de alto nivel) suele consistir en realidad (a nivel de hardware) de tres pasos: 1) obtener desde la memoria el valor previo, 2) calcular el valor nuevo, y 3) almacenar en memoria el resultado. En programación concurrente, por ejemplo, el planificador del sistema operativo podría interrumpir a un hilo de ejecución en cualquiera de estos tres pasos, dando lugar a condiciones de carrera si otro hilo opera sobre la misma variable. Una operación se denomina atómica si aún a bajo nivel se observa como una única operación indivisible (un solo paso). Los procesadores actuales ofrecen un conjunto de instrucciones especialmente útiles para la programación en paralelo, que se ejecutan de forma atómica aunque en algunos casos representan más de una operación (por ejemplo, comparar el valor de una variable para asignarle un valor nuevo solamente si el valor previo era el esperado, operación conocida como compare-and-swap o CAS). Estas operaciones son en realidad las operaciones básicas sobre las cuales se construyen generalmente los demás mecanismos de sincronización presentados. Sin embargo, en muchos casos pueden utilizarse directamente para construir algoritmos que garanticen su correcta ejecución sin bloqueos (denominados lock-free, ver sección 4.5). En ciertos lenguajes compilados como C++, en los que el compilador tiene libertades para optimizar el algoritmo durante la compilación, las variables atómicas ofrecen garantías adicionales respecto al tipo de optimizaciones que el compilador puede realizar para no introducir problemas adicionales producto de no considerar la ejecución concurrente (más detalles en la sección 3.4.4).

3.4. Consideraciones de bajo nivel

El uso de arquitecturas paralelas (especialmente arquitecturas de memoria compartida), además de requerir el planteo de nuevos algoritmos para adaptarse al cambio de paradigma, y el uso de mecanismos de sincroniza-

ción como los mencionados anteriormente, obliga a tener en cuenta a la hora de realizar una implementación y/o evaluar su eficiencia paralela, algunas consideraciones de muy bajo nivel, relacionadas directamente al diseño del hardware sobre el que se ejecutan, o a detalles internos de los compiladores y sistemas operativos que se utilizan. En esta sección se describen algunos de ellos, por ser causas frecuentes de baja eficiencia paralela, o detalles de interés particular para las implementaciones que se discutirán en los siguientes capítulos.

3.4.1. CPU-bound vs memory-bound

Se dice que un proceso está limitado por el uso de CPU (CPU-bound) si el recurso que está limitando la velocidad de ejecución del proceso es el procesador. Es decir, si al aumentar la velocidad de reloj del procesador la velocidad de ejecución del proceso también se incrementa, ya que el procesador pasa la mayor parte del tiempo realizando cálculos. En cambio, se dice que un proceso está limitado por el uso de memoria (memory-bound) si el recurso que limita la velocidad de ejecución es la memoria RAM. En este segundo caso, la velocidad de acceso a dicha memoria (velocidad de escritura/lectura) es la condicionante (y no la cantidad de memoria disponible). Si se aumentara la velocidad del bus que comunica al procesador con la memoria principal, la velocidad de ejecución aumentaría, ya que el procesador pasa la mayor parte del tiempo esperando por los datos que provienen de dicha memoria. Este segundo caso es más común cuando se procesa un gran volumen de datos, cuando los cómputos requeridos sobre cada dato son relativamente simples, y/o cuando no se logra un buen aprovechamiento de la memoria cache.

En arquitecturas paralelas de memoria compartida, este problema se acentúa conforme crece el número de núcleos, ya que todos deben acceder a la misma memoria principal. Por ello, muchos programas que en serie o corriendo sobre pocos núcleos actúan como CPU-bounded, se convierten en memory-bounded al incrementar la cantidad de núcleos. La programación en paralelo hace más relevante en el diseño de un algoritmo diversas consideraciones respecto las estructuras de datos, requiriendo localidad (que todos los datos para un proceso se encuentren en posiciones cercanas en la memoria) y representaciones compactas (que los datos se representen con la menor cantidad posible de bytes).

En general, resulta difícil verificar empíricamente que un proceso es memory-bounded. Se sospecha que lo es cuando un análisis teórico de la eficiencia paralela esperada arroja resultados muy superiores a los observados

en la práctica, y se han descartado otras causas más fácilmente detectables, como el exceso de bloqueos debido al uso de mutexes. Es posible analizar el patrón de acceso a memoria en ciertos algoritmos, y obtener estimaciones teóricas del ancho de banda requerido para que la velocidad de acceso no sea un problema. Pero en general, para algoritmos complejos esta técnica no es viable. Por otro lado, la mayoría de las herramientas de profiling disponibles no logran detectar correctamente este problema. Si un proceso está acotado por la velocidad de acceso a disco, por ejemplo, el sistema operativo pondrá el proceso a dormir durante las esperas, con lo cual decaerá la utilización de la CPU, y este efecto será entonces observable. Cuando el problema es, en cambio, el acceso a la memoria principal (RAM), los ciclos de CPU durante los cuales el procesador espera por los datos son percibidos por el sistema operativo (y por ende por la mayoría de las herramientas de profiling) como ciclos en los que el procesador está ocupado. Para lograr evaluar este fenómeno con certeza, es necesario instrumentar la ejecución a nivel de hardware (ya que a nivel de software estaríamos modificando con la instrumentación los patrones que intentamos medir). Las últimas generaciones de los procesadores Intel han ido añadiendo en sus núcleos registros adicionales específicos para tal fin (denominados *performance counters*), que pueden ser consultados e interpretados con herramientas especiales como Intel VTune Amplifier[52]. Con estos medios, de acuerdo a qué información provea cada generación de procesadores, se puede estimar a partir de una ejecución valores significativos como el factor de carga del bus de acceso a memoria, obteniendo así mejores indicadores con los cuales diagnosticar el problema.

Para mitigar el problema desde el desarrollo del software, se deben analizar estrategias como utilizar estructuras de datos más compactas y/o cache-friendly, disminuir la necesidad de acceder a memoria (por ejemplo, con técnicas de memoization[2]) o, por el contrario, incrementar el uso de CPU recalculando en múltiples ocasiones resultados parciales a cambio de reducir el volumen de datos a mantener en memoria, etc. Sin embargo, el hardware también ha evolucionado en una dirección en la cual contribuye a mitigar el problema: arquitecturas NUMA (Non-Uniform Memory Access). En estas arquitecturas, la memoria en diferentes ubicaciones del espacio de direcciones presenta diferentes características de performance [53]. Es común en sistemas multisocket (usualmente servidores de alto rendimiento), que poseen más de un procesador, conectado cada uno directamente a una porción de la memo-

[2]Se conoce como *memoization* a un conjunto de técnicas de optimización para conseguir mayor velocidad de ejecución que se basan en almacenar resultados de funciones costosas que podrían ser invocadas nuevamente con un mismo conjunto de argumentos, para evitar recalcularlos.

ria RAM considerada local, e indirectamente a través de un bus central al resto de la memoria RAM (local a otros procesadores). De esta forma, si cada procesador accede a datos de la porción del espacio de direcciones asociada a su memoria local, el ancho de banda de acceso a memoria total virtualmente se multiplica, ya que cada uno puede utilizar en simultáneo su propio canal de comunicación directo (bus local). Dado que se expone al programador un único espacio de direcciones global que representa la suma de todas las memorias RAM de cada procesador, en general se puede programar para dicha arquitectura igual que se lo haría para cualquier otra arquitectura de memoria local con acceso uniforme (SMP), dejando al sistema operativo la tarea de aprovechar la afinidad de los procesadores con las diferentes memorias. Obviamente este caso no es óptimo, y sería deseable que cada proceso pudiera controlar el alojamiento de memoria dinámica para aprovechar mejor esta afinidad, pero para ello se deben exponer más detalles sobre la gestión de la memoria a través de los lenguajes de alto nivel. Y la forma de lograr esto aún no ha convergido de modo que sea estándar. Por ejemplo, el modelo de memoria actual de C++ no tiene en cuenta ninguna de estas posibilidades. Para aprovecharlas se deben conocer y utilizar llamadas directas a la interfaz (API) que ofrece el kernel del sistema operativo.

3.4.2. False sharing

La situación conocida como *false-sharing* ocurre cuando dos procesadores escriben repetidamente y de forma alternada en diferentes variables que se ubican en una misma linea de cache [54]. Por ejemplo, si se utiliza un arreglo de enteros para representar valores utilizados por diferentes hilos, es muy probable que dos valores consecutivos del arreglo se ubiquen dentro de una misma linea de cache, ya que valores consecutivos se ubican en posiciones consecutivas, y las lineas de cache suelen tener un tamaño superior al de una variable entera. Si dos hilos que se ejecutan en paralelo en diferentes núcleos acceden a estas variables, esta linea de cache estará replicada en las memorias cache locales (L1) de cada núcleo. El procesador debe garantizar que el uso de memoria cache sea transparente para el programa que ejecuta, por lo cual es responsable de mantener sincronizadas estas dos instancias de la misma linea de cache. Como resultado, si bien desde el punto de vista del programador que implementa su algoritmo en un lenguaje de alto nivel no hay sincronización alguna necesaria para ejecutar ambos hilos en paralelo, la arquitectura de hardware sí lo requiere, reduciendo considerablemente la velocidad de ejecución. Esto produce un efecto contra-intuitivo, ya que en un modelo serial el incremento en el tamaño de cache se traduce directamente

en un incremento en la velocidad de ejecución, mientras que una arquitectura paralela podría por este efecto generar justo lo contrario.

No es fácil identificar esta situación. En general, se debe considerar como una causa probable (además de la mencionada en la subsección anterior) cuando un análisis teórico de la eficiencia paralela de un algoritmo arroja un valor esperado para la eficiencia paralela muy superior al observado en la práctica. La solución más usual consiste en reordenar los campos dentro de una estructura de datos (si fueran campos contiguos y no elementos de un arreglo), o exigir al compilador que alinee en memoria los elementos de un arreglo. Alinear en memoria significa acomodarlos de forma que comiencen en posiciones múltiplos de algún valor dado, dejando bytes sin utilizar como relleno en medio de los datos si es necesario. Para evitar el false sharing, el valor utilizado para calcular la alineación debería idealmente coincidir con el tamaño de una línea de cache. En la mayoría de las PCs actuales, este tamaño es de 64 bytes.

3.4.3. Mecanismos de espera durante un bloqueo

Los mecanismos de sincronización que involucran bloqueos en los hilos que los utilizan (como mutexes, semáforos y condition-variables) pueden implementar internamente dos tipos de bloqueos/esperas, conocidos como espera ocupada (busy-wait, o spinning), o espera durmiente (sleep-wait), siendo el segundo el método más usual. En una espera ocupada, el hilo que requiere acceso al recurso (por ejemplo un mutex), ingresa en un loop sin acciones en su interior cuya condición de salida es la liberación del mutex. Este loop consume ciclos de CPU evaluando constantemente la condición (consultando constantemente el estado del mutex). Esto hace que el proceso detecte inmediatamente que el recurso ha sido liberado cuando esto ocurre, y continúe avanzando sin requerir ninguna operación adicional más que la adquisición del recurso. El problema de este enfoque radica en que el hilo lógicamente "bloqueado", mientras itera en dicho loop no está haciendo trabajo útil, pero sí está consumiendo recursos, ya que utiliza un núcleo de la CPU completamente solo para iterar y evaluar la misma condición repetidas veces. Como contrapartida, en una espera durmiente, cuando un hilo requiere un recurso (por ejemplo un mutex) no disponible, utiliza algún mecanismo provisto por el sistema operativo para indicarle al planificador de tareas que ese hilo puede suspenderse hasta que el recurso se libere. De esta forma, el núcleo en el cual se ejecutaba el hilo ahora bloqueado puede utilizarse mientras tanto para ejecutar otra tarea (potencialmente otro hilo u otro proceso). La desventaja

de este método radica en que las operaciones que realiza el sistema operativo para *dormir* y *despertar* un hilo, como así también para cambiar de contexto al ejecutar mientras tanto otro hilo, no son operaciones inmediatas. Es decir, desde que un recurso se libera hasta que el hilo que esperaba por dicho recurso se desbloquea, se consumen algunos ciclos de CPU en las tareas de despertar el proceso y realizar el cambio de contexto. En general, la mayoría de las APIs ofrecen mecanismos para implementar o utilizar ambos tipos de espera. La espera ocupada solo es conveniente cuando se sabe que la espera será excesivamente corta, y por ello no justifica el cambio de contexto.

3.4.4. Efecto de las optimizaciones en la compilación

No todos los lenguajes de alto nivel ofrecen acceso a operaciones atómicas. En particular, C++ ofrece acceso a las mismas mediante su biblioteca estándar desde 2011. En un lenguaje como C++, el uso de operaciones atómicas, o variables atómicas (variables cuya modificación se realiza solo mediante operaciones atómicas) tiene una importancia adicional debido a particularidades de su modelo de memoria. Lenguajes compilados como C++ dan libertades a los compiladores para modificar los algoritmos que compilan con el objeto de aumentar la performance bajo la condición de que la modificación no sea observable. Es decir, siempre que el resultado del algoritmo no varíe, el compilador puede, por ejemplo, reordenar las instrucciones para reducir el acceso a memoria y los fallos de cache. Sin embargo, en la programación concurrente esto puede generar un problema. Al reordenar las instrucciones, el compilador mantiene el resultado, pero varía los estados intermedios. Y esta variación puede percibirse desde otro hilo que accede a las mismas estructuras de datos, generando un comportamiento diferente al esperado. Los problemas que estos cambios pueden provocar son en general muy difíciles de identificar, ya que estas optimizaciones se realizan de forma transparente durante la compilación. Cuando el lenguaje le permite al programador explicitar que ciertas operaciones y/o variables serán atómicas, le provee al compilador la información necesaria para distinguir las situaciones en las que no debe reordenar las instrucciones para optimizar. En particular, al usar operaciones atómicas en C++11, el modelo de memoria utilizado obliga al compilador a respetar la posición de esa instrucción dentro del algoritmo. Es decir, puede reordenar las anteriores de forma que el resultado en ese punto sea el mismo, y reordenar las posteriores con idéntico criterio, pero no mover instrucciones entre puntos previos y posteriores a la operación atómica.

3.4.5. Gestión del heap del proceso

Por último, se debe tener en cuenta al programar en un lenguaje de alto nivel, que la compartición de datos y/o la sincronización del acceso a los mismos, podría estar oculta o implícita, de forma que o bien produzca data-races (compartición), o bien genere secciones críticas ocultas que reduzcan la eficiencia paralela (sincronización). Es decir, al construir un programa paralelo reutilizando funciones u objetos de bibliotecas, se debe tener algún conocimiento del funcionamiento interno de estas bibliotecas (algo que la programación orientada a objetos deliberadamente busca ocultar) para evitar caer en estos problemas. En C++, dos operadores que particularmente podrían generar problemas son los operadores new y delete, utilizados para solicitar al sistema un bloque de memoria dinámica, y liberarlo respectivamente. Los diferentes hilos de un mismo proceso comparten el mismo memory heap, en el cual alojan los bloques de memoria dinámica solicitados mediante new. Las implementaciones que ofrecen los compiladores de los operadores new y delete deben ser thread-safe (la sección 18.6.1.4 del estándar C++14[55] requiere que no generen data races). Esto indica que deben utilizar internamente algún mecanismo de sincronización, el cual debe ser tenido en cuenta como un potencial punto de contención entre hilos. Estrategias como el uso de thread-local allocators (ver un ejemplo en la sección 4.1.3) pueden contribuir a mitigar los potenciales efectos de este problema.

3.5. Medidas de desempeño

Al cambiar un algoritmo serie por un algoritmo paralelo y ejecutarlo sobre P núcleos o procesadores, sería deseable que para un mismo problema, la ejecución paralela tomara $1/P$ veces el tiempo de la ejecución serie. En este caso, obtendríamos un speed-up (S_P) de P. Es decir, este coeficiente indica cuanto menor fue el tiempo necesario al paralelizar en comparación al tiempo de la ejecución en serie. Si T_1 es el tiempo de la ejecución en serie, y T_P de la ejecución paralela con P hilos y/o procesadores, entonces el speed-up se obtiene como:

$$S_P = \frac{T_1}{T_P} \qquad S_P = \text{Speed-up} \tag{3.1}$$

Usualmente se obtiene $S_P < P$. Cuando S_P se acerca a P se dice que el algoritmo es eficiente. En particular, la eficiencia (E_P) paralela indica qué porcentaje o factor del speed-up ideal logra obtener un algoritmo:

$$E_P = \frac{S_P}{P} \qquad E_P = \text{Eficiencia paralela} \tag{3.2}$$

En general, se tiene que $0 < E_P < 1$. Esto puede deberse a que existan momentos durante la ejecución en los cuales algunos procesadores no estén realizando trabajo útil (estén por ejemplo bloqueados a la espera de un recurso), y/o a que la versión paralela del algoritmo deba realizar un número de operaciones mucho mayor que la versión serie. La redundancia (R_P) mide este último factor, pesando la cantidad de operaciones necesarias en una ejecución paralela (O_P) frente a la cantidad necesaria en una ejecución serie (O_1):

$$R_P = \frac{O_P}{O_1} \qquad R_P = \text{Redundancia} \tag{3.3}$$

La redundancia cumplirá en general $R_P > 1$. Una medida asociada a la redundancia es la utilización, que mide la ocupación real de los P procesadores frente a la potencial en un caso ideal (asumiendo $O_1 = T_1$):

$$U_P = R_P \times E_P = \frac{O_P}{P \times T_P} \qquad U_P = \text{Utilización} \tag{3.4}$$

La utilización cumplirá siempre $U_P <= 1$. Finalmente se puede obtener una medida objetiva de la calidad del paralelismo Q_P como:

$$Q_P = \frac{S_P \times E_P}{R_1} \qquad Q_P = \text{Calidad} \tag{3.5}$$

Se debe notar que en la práctica O_1 y O_P son muy difíciles o a veces imposibles de obtener, no solo por la dificultad de obtener un conteo de operaciones fehaciente en un algoritmo complejo, sino también porque las diferentes operaciones tendrán diferentes costos asociados, y otros factores como por ejemplo el uso de memoria cache o las optimizaciones que realiza transparentemente un compilador o internamente el microprocesador hacen que aún una misma operación pueda tener costos muy dispares en sus diferentes ejecuciones. Por ello, las tres últimas medidas presentadas tienen en general poca utilidad empírica, mientras que las primeras sí pueden reflejar resultados objetivos si se comparan mediciones de un mismo algoritmo resolviendo un mismo problema con diferentes cantidades de procesadores.

Sin embargo, en muchos procesadores actuales es posible discriminar el tiempo en que un procesador está realizando trabajo útil (Compute Time o

C) del tiempo en el cual se encuentra detenido a la espera de un recurso o de instrucciones (Idle Time o I). Se puede construir entonces una expresión alternativa para la utilización (U_p), basada en estas mediciones, que puede obtenerse empíricamente con herramientas de profiling adecuadas:

$$U_P = \sum_{i}^{P} \frac{C_i}{I_i + C_i} \qquad \begin{array}{c} C_i = \text{Compute Time del proc. } i \\ I_i = \text{Idle time del proc.} i \end{array} \tag{3.6}$$

3.5.1. Aplicación a la generación de mallas

Además de los requisitos que se espera que cumpla cualquier generador mallas o de tetraedrizaciones serie, uno que opere en paralelo debería ser escalable con respecto al tiempo de ejecución y el espacio en memoria, eficiente (en el sentido del paralelismo), y estable[43]. Se dice que el algoritmo es escalable en el uso de un recurso si su consumo aumenta "lentamente" (lineal o sublinealmente) al aumentar el número de hilos o procesos, suponiendo que el tamaño del problema varíe de forma tal que el factor tamaño/cantidad de procesadores permanezca constante. Se dice que es eficiente en el sentido de la paralelización si hace un uso sostenido de todos los núcleos o procesadores disponibles. Es decir, debe evitar la presencia de procesadores ociosos durante la ejecución. Por último, el algoritmo será estable si la calidad de la solución no se degrada al aumentar la cantidad de hilos o procesos utilizados para obtenerla.

Capítulo 4

Algoritmos y Estructuras de Datos para Ordenamiento Espacial

En este capítulo se describen en detalle las propiedades de diferentes estructuras de datos utilizadas directa o indirectamente en los algoritmos de mallado que se presentarán posteriormente. No es necesario comprender por completo los detalles de las mismas para entender el funcionamiento de dichos algoritmos. Para tal fin sería suficiente leer solo las descripciones generales presentadas en las introducciones de cada sección. Sin embargo, sí resulta importante analizar comparativamente las ventajas y desventajas para cada grupo de estructuras, para justificar luego su elección (o no) en los siguientes capítulos. Además, muchos detalles de implementación que normalmente se dejan de lado en las descripciones de estos algoritmos pueden ser de vital importancia al momento de llevar estos contenidos a la práctica. Por esto, se discutirán también las implementaciones particulares utilizadas a lo largo de este trabajo para permitir y facilitar la reproducibilidad de los resultados.

4.1. Listas vs. Vectores

Listas y vectores son las dos estructuras de datos más básicas y también más utilizadas. Muchas otras estructuras más complejas se basan en los mecanismos y el análisis de estas dos. Son dos clases de contenedores, es decir, estructuras de datos para "contener" un conjunto de elementos, en general del mismo tipo. La diferencia entre ambas radica en la organización de estos

datos en memoria, lo cual tiene consecuencias directas en los costos de la operaciones más básicas y frecuentes que se realizan sobre las mismas.

Operación	Lista	Vector
Insert	$O(1)$	$O(N)$
Erase	$O(1)$	$O(N)$
$N\times$ Push back	$O(N)$	$O(N \log N)$
$N\times$ Pop back	$O(N)$	$O(N)$
Clear	$O(N)$	$O(1)$
Size	$O(1)$	$O(1)$
Splice	$O(1)$	$O(N)$

Figura 4.1: Tabla comparativa de tiempos de ejecución para las operaciones más comunes en vectores y listas. Se muestra en cada caso el mejor tiempo posible, pero en general no es factible cumplir todos estos límites en una misma implementación. Por ejemplo, una lista no puede ofrecer Splice $O(1)$ y Size $O(1)$ simultáneamente; y la operación Clear será verdaderamente $O(1)$ en un arreglo solo cuando sus elementos no requieran la ejecución de un destructor.

En las siguientes dos subsecciones se repasan algunos conceptos básicos teóricos sobre las operaciones con vectores y listas (un programador familiarizado con estas estructuras puede omitir esos párrafos). La tercera y última subsección, en cambio, describe algunas propiedades derivadas de la forma en que se gestiona la memoria en cada una que no siempre resultan obvias, y que serán de especial interés para el diseño de las estructuras de datos que darán soporte a los algoritmos de mallado en capítulos posteriores.

4.1.1. Acceso

En un vector, los datos contenidos se organizan en memoria de forma contigua. Por esto, solo se requiere conocer la dirección de memoria del primer dato para poder acceder a cualquiera de los demás. Al ser éstos datos todos del mismo tipo (se dice que es una estructura homogénea), ocupan el mismo espacio en memoria, con lo cual un simple cálculo ($O(1)$) permite obtener a partir de la posición de uno, la posición de cualquier otro. En una lista, en cambio, se utiliza un sistema de enlaces para ordenar lógicamente los datos contenidos, ya que los mismos pueden estar dispersos arbitrariamente en la memoria. Cada dato se encapsula en una estructura denominada Nodo que contiene además un enlace (puntero) al siguiente. Entonces, si se conoce la dirección de memoria del primer dato, se puede llegar a cualquier otro siguiendo los enlaces, pero este proceso puede requerir tantos pasos como

elementos contenga el contenedor ($O(n)$). Así, el acceso eficiente a datos es aleatorio en un vector (se puede acceder eficientemente a cualquier posición), pero secuencial en una lista (solo es eficiente recorrer los datos contenidos en orden).

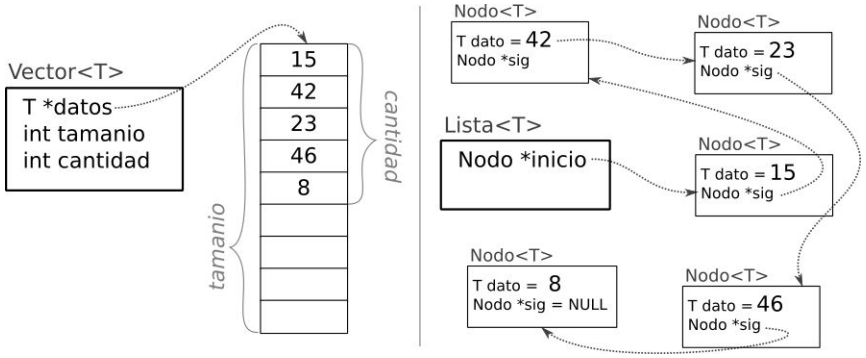

Figura 4.2: Organización en memoria de una secuencia de enteros contenida en un vector (izquierda) y en una lista simplemente enlazada (derecha).

4.1.2. Inserción/eliminación

Dado que en un vector los datos deben estar en memoria contiguos y ordenados, insertar o eliminar un dato puede implicar desplazar todos los demás para poder mantener estos invariantes ($O(n)$). En una lista, en cambio, dado que no se exige un ordenamiento particular de los datos en memoria, para agregar o eliminar un elemento, solo es necesario reconfigurar la cadena de enlaces, lo cual requiere la modificación de unos pocos enlaces en los nodos ubicados conceptualmente contiguos al insertado/eliminado ($O(1)$). En el caso de la lista entonces, se dice que es eficiente la inserción/eliminación para cualquier posición dada de la misma, mientras que para el vector, sólo es directo hacerlo sobre el último elemento (omitiendo por el momento el costo de reservar la memoria necesaria). Si se requiere una estructura para almacenar datos donde el orden de los mismos no sea importante, se puede utilizar un vector, realizando las inserciones/eliminaciones siempre sobre el último elemento (si se quiere eliminar otro simplemente se lo intercambia con el último), y manteniendo así la ventaja sobre la lista que da el acceso aleatorio.

4.1.3. Gestión de la memoria

Si bien la mayoría de los libros clásicos sobre algoritmos y estructuras de datos suelen hacer énfasis en los órdenes teóricos de los tiempos de inserción/eliminación y acceso de estas estructuras, en la práctica la forma en que se accede a la memoria a nivel hardware suele ser el factor más condicionante en la mayoría de los casos. En una lista, teóricamente insertar/eliminar es $O(1)$, pero la constante puede ser grande para las implementaciones tradicionales ya que las operaciones de solicitar y liberar memoria al sistema operativo son relativamente costosas (hasta pueden derivar en bloqueos en arquitecturas de memoria compartida), y se realizan por cada nodo (es decir, por cada elemento contenido). En el caso de un vector, si se conoce la cantidad máxima de elementos que llegará a contener, la reserva de memoria se hace sólo una vez al construir el mismo. Si el tamaño máximo no se conoce, se comienza con un tamaño pequeño, y una vez completado, se intenta reservar un bloque más grande, copiar los datos al mismo, y liberar el bloque anterior más pequeño. Es importante definir correctamente de qué forma irá creciendo el tamaño de los bloques a medida que se requiera más espacio para que este proceso de copia no domine el tiempo de inserción. Es decir, si el bloque crece el tamaño de un dato cada vez que el espacio resulta insuficiente para una inserción, se debe pagar este costo en cada inserción. Esto hace que insertar, aún al final sea $O(n)$, con la ventaja de que el vector ocupa en memoria exactamente el espacio necesario. Si, por el contrario, al requerir más espacio, se solicita un bloque considerablemente más grande que el anterior, se logra evitar el problema en las siguientes inserciones. Así, la mayoría de las inserciones será $O(1)$, y solo unas pocas serán $O(n)$, a riesgo de utilizar más memoria de la necesaria para una dada cantidad de datos. La práctica más común consiste en aumentar exponencialmente el tamaño del bloque (la mayoría de los compiladores utiliza un factor de crecimiento entre 1.4 y 2). Por ejemplo, si el tamaño se duplica, para n inserciones, tendremos $O(\log_2 n)$ aumentos, y si bien algunas inserciones serán $O(n)$, el costo amortizado de n inserciones será $O(n)$, lo que equivale a un costo promedio $O(1)$ por cada inserción. Así, cuando las inserciones se realizan sólo al final del vector, el costo de las mismas se reduce considerablemente. Si se realizan pocas inserciones, el costo es bajo porque n es pequeño. Si se realizan muchas, el costo es bajo porque las operaciones que requieren un gran movimiento de datos y una nueva reserva de memoria son sólo $O(\log_2 n)$. Además, en términos de consumo de memoria, aún reservando más espacio del necesario el costo es simplemente $2n = O(n)$.

Comparativamente, frente a una lista, además de no requerir una gran

cantidad de pequeñas reservas de memoria, el vector tiene dos ventajas adicionales determinantes, ambas relacionadas entre sí: los datos se almacenan de la forma más compacta posible, y el acceso a los mismos es *cache-friendly*. En la lista, cuando el elemento que guarda el nodo no es grande, el costo de los enlaces adicionales se vuelve representativo. Por ejemplo, en una lista doblemente enlazada (el tipo de lista más común) de enteros en C++, dado que en una arquitectura de 64bits un puntero ocupa el doble de espacio que un entero, la lista ocupa en realidad 5 veces más que un vector cuyo bloque tenga el tamaño exacto. Si el bloque del vector está sobredimensionado (tiene capacidad para más elementos, por el crecimiento exponencial que se comentó anteriormente), la relación será de 2 a 5 en el peor de los casos, aún a favor del vector. Se suma a esto el hecho de que el acceso a datos contiguos en memoria es mucho más eficiente que el acceso a posiciones de memoria aleatorias, debido a la presencia de distintos niveles de cache, utilizados en todas las arquitecturas de hardware actuales. Por eso se dice que el vector es *cache-friendly*, ya que al tener sus datos contiguos se almacenan en conjunto en cache, y al ser más compacto permite almacenar una mayor cantidad para un tamaño de cache dado. Por todo esto, en muchos casos, es preferible utilizar vectores frente a listas, aún cuando predominen operaciones de inserción y eliminación[56][47], ya que si bien la cantidad de operaciones es mayor, cada operación requerida por la lista pueden tener un costo real muy superior (aunque constante) al de las operaciones requeridas por el vector.

Sin embargo, hay dos ventajas que tiene una lista que serán explotadas más adelante. En primer lugar, mover una secuencia de elementos de una lista a otra puede ser una operación $O(1)$ (generalmente denominada splice), dado que se logra simplemente reconfigurando los enlaces del primer y último elemento de la secuencia, sin importar cuan larga sea la misma. Además, en esta operación no requiere reserva ni liberación de memoria alguna. En segundo lugar, los nodos de una lista no cambian su ubicación en memoria cuando la lista cambia, o cuando son transferidos de una lista a otra. En el caso de un vector, al insertar o eliminar un elemento, todos los elementos en posiciones posteriores dentro de dicho vector se verán desplazados. Aún cuando la inserción se realice al final de la secuencia, el bloque de memoria prealocado por el vector puede ser insuficiente, y en ese caso la operación de realocación que se mencionó anteriormente llevará a que todos los elementos cambien su posición en memoria. Finalmente, al mover elementos de un vector a otro, siempre tendremos este problema. El cambio de ubicación en la memoria (tanto sea su posición absoluta, como su posición relativa al inicio del bloque) será un problema cuando los datos del vector deban ser referenciados desde una segunda estructura externa.

Uso de allocators

Finalmente, cabe destacar que se intentó en este trabajo minimizar el número de reservas y liberación de memoria debido al uso intensivo de listas y estructuras enlazadas similares que cambian constantemente a lo largo del proceso de mallado. Para ello, se implementó en la lista desarrollada una clase auxiliar encargada de la gestión de la memoria que encapsula las operaciones de reserva y liberación (equivalente a lo que en la biblioteca estándar de C++ se conoce como *Allocator*), para poder variar fácilmente las estrategias utilizadas para estas operaciones. Se midieron los tiempos de mallado utilizando la implementación más simple (un new/delete por cada nodo insertado/eliminado), y también una versión alternativa que mantiene una lista/pool de nodos liberados[1] para reutilizar su memoria y evitar así operaciones de liberación y posterior reserva de memoria. Se denominó a la implementación del mecanismo más simple RegularAllocator, y a la implementación de la versión mejorada RecyclingAllocator. Además de evitar estas operaciones, presenta dos ventajas adicionales. Si el contenedor de objetos liberados del RecyclingAllocator funciona como una pila (LIFO), se favorece el mejor aprovechamiento de la memoria cache, ya que si se requiere un bloque nuevo inmediatamente después de haber liberado uno reservado anteriormente, se otorga dicho bloque recién liberado, que muy probablemente permanezca todavía en alguno de los primeros niveles de memoria cache del procesador. Finalmente, en arquitecturas de memoria compartida se puede utilizar un pool diferente para cada hilo de ejecución para evitar la necesidad de agregar mecanismos de sincronización en sus operaciones. Los resultados confirmaron la utilidad de dichas optimizaciones (más detalles en la sección 6.3).

4.2. Árboles

Un árbol es un grafo simple en donde todos los nodos están directa o indirectamente conectados a un nodo particular denominado "raíz" por un camino único. En consecuencia, el grafo es conexo y no presenta ciclos (estos son los invariantes que deben mantenerse al modificar el árbol). Si se recorre el árbol desde el nodo raíz a un nodo genérico X, por cada arista que se atraviesa en sentido de un nodo Y a un nodo Z, se dice que el nodo Y es padre del nodo Z, y, por contraposición, que el nodo Z es hijo del nodo Y. Un nodo que no tiene hijos se denomina "hoja". En informática, los árboles se utilizan para almacenar datos de forma ordenada, para que las búsquedas o

[1] Estructura también conocida como *free-list*

consultas sobre los mismos sean operaciones eficientes. El orden está dado por el criterio con que se determina cuantos hijos tiene un nodo y qué propiedades tienen los datos almacenados en cada sub-árbol formado a partir de un hijo.

Por ejemplo, en un árbol binario, cada nodo tiene dos hijos. Si al nodo se le asocia un valor, se puede convenir que los datos menores a dicho valor se almacenen en el sub-árbol del primer hijo, mientras que los mayores en el sub-árbol del segundo. De esta forma, al realizar una búsqueda, se parte del nodo raíz, y por cada nodo recorrido se compara el valor a buscar con el valor del nodo y en caso de ser diferentes se avanza por el hijo adecuado para continuar la búsqueda. Esta técnica es muy comúnmente utilizada para ordenamiento y búsquedas sobre datos unidimensionales, ya que si el árbol está correctamente balanceado, las inserciones y búsquedas son $O(\log_2 n)$. Un árbol balanceado es una árbol cuyo número de niveles (longitud del camino más largo desde la raíz a un nodo +1) es el menor posible (para la cantidad de nodos que contiene). Existen algoritmos que permiten mantener el balance de un árbol al insertar o eliminar datos del mismo utilizando rotaciones (operaciones internas donde sub-árboles completos son trasladados de un nodo a otro). Estos algoritmos (que dan lugar tipos de árboles particulares como red-black tree o AVL tree) no serán descritos con detalle en este capítulo ya que no serán utilizados en el presente trabajo. Cuando no se utilizan técnicas de balanceo adecuadas, el balance del árbol depende del conjunto de datos que almacena y en muchos casos del orden en que fueron insertados. El valor asociado a un nodo padre puede ser uno de los datos que se quiere almacenar en el árbol, o bien un dato arbitrario elegido con algún criterio particular si se conocen de antemano propiedades del conjunto de datos que almacenará el árbol.

La implementación de un árbol en general está basada en una estructura que representa un nodo y contiene, además un valor asociado y/o enlaces (en el mismo sentido que los enlaces de un nodo de una lista) a los nodos hijos y, opcionalmente, al nodo padre. De esta forma, operaciones como la rotación pueden realizarse en $O(1)$ (de manera similar al splice en una lista). Cuando la cantidad máxima de hijos es fija (como en un árbol binario), la implementación se simplifica, de lo contrario el nodo debe contener una lista o un vector de enlaces a hijos. Además, dependiendo del tipo de árbol, se deberá diferenciar o no la estructura utilizada para un nodo interno o raíz de la utilizada para un nodo hoja.

4.2.1. Octrees

Los octree son árboles que guardan coordenadas de puntos 3D (u objetos asociados a dichas coordenadas) representando en cada nodo interno una partición binaria para cada coordenada de un espacio previamente acotado. Por ello, en 3D, un dominio acotado por valores máximos y mínimos en 3 direcciones representa un paralelepípedo alineado con los ejes que se divide en $2^3 = 8$ partes iguales (en 2D se lo denomina quadtree porque cada nodo divide su espacio en $2^2 = 4$ subespacios). El octree es una de las estructuras de ordenamiento espacial más utilizadas en aplicaciones de geometría computacional por el balance que ofrece entre simpleza y eficiencia.

Para construir un octree se debe conocer previamente el *axis-aligned bounding-box* (AABB) del conjunto de datos (es decir, los valores máximos y mínimos para cada coordenada presentes en dicho conjunto). El nodo raíz representa ese espacio. Al insertar un nodo, se comienza por el nodo raíz. Si este tiene nodos hijos, se avanza al hijo adecuado según las coordenadas del dato a insertar. En caso contrario (si es un nodo hoja), el dato se guarda en el nodo, suponiendo que el nodo aún no tenga datos asociados. Si el nodo no tiene lugar para el dato (ya tiene datos asignados) se crean los ocho hijos para dicho nodo, se mueve el dato asociado al nodo al hijo que corresponda y se intenta nuevamente agregar el nuevo dato. No es necesario, en un octree, guardar en cada nodo los límites del espacio que representa, ya que estos pueden calcularse fácilmente al ir avanzando de nivel desde el AABB inicial (correspondiente al nodo raíz). En este árbol, entonces, los datos se almacenan en los nodos hoja, mientras que los límites asociados a los nodos interiores (los que se utilizan para elegir por cual de sus hijos proceder en una inserción o búsqueda) son fácilmente calculables.

Al utilizar un octree, se espera que la profundidad del árbol sea proporcional a la densidad de los datos que almacene, variando esta espacialmente. Es decir, que el nivel de refinamiento de la estructura para una porción del dominio dada (cuantas hojas del árbol se utilizan para cubrir dicha área) se vea relacionado (en proporción directa) con la cantidad de datos contenidos en la misma. Sin embargo, se debe notar que esto no siempre será así, ya que en casos particulares unos pocos datos pueden llevar a un alto nivel de refinamiento en la estructura, y por ende a una profundidad excesiva del árbol. Esto es debido a que los planos imaginarios que separan los subespacios que corresponden a cada nodo hijo, para un nodo padre dado, tienen una posición prefijada que no considera (no depende de) la distribución de los datos que debe contener (siempre pasarán por el centro del AABB correspondiente a ese nodo padre). Por esto, cuando el árbol almacena un nodo por hoja, la

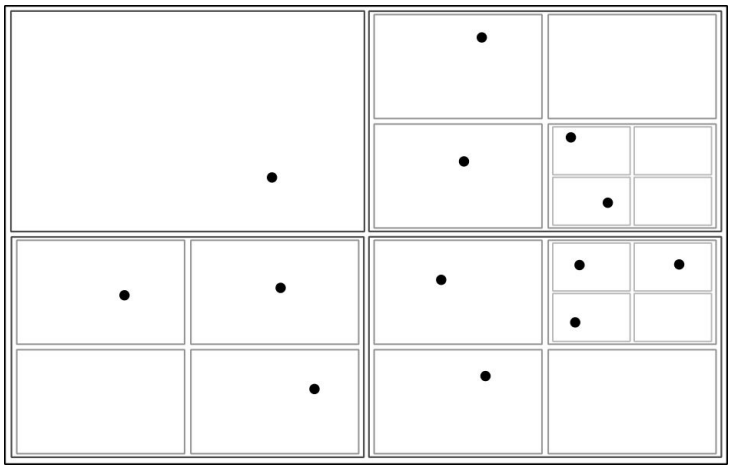

Figura 4.3: Esquematización de la jerarquía de áreas correspondientes a cada nodo de un quadtree para un conjunto de 13 puntos.

profundidad del mismo suele estar determinada en realidad por la mínima distancia entre dos datos del conjunto (ver imagen 4.4).

Cuando los datos que se almacenan son nodos de una malla bien formada, se puede esperar que esta situación no ocurra, ya que en una malla para cálculos ingenieriles por métodos numéricos se busca controlar distancia entre nodos (h), y evitar que esta sea demasiado pequeña porque en ese caso puede producir inestabilidades numéricas en los cálculos para los cuales la malla da soporte geométrico, o llevar a tiempos de cómputo excesivos en algunos casos. Sin embargo, el problema puede presentarse igualmente, o los datos que se guardan pueden ser de otra naturaleza. Por ello, para resolver este problema, en la práctica se puede utilizar lo que se conoce como *bucketed*-octree. Este es un árbol basado en el octree con el agregado de que los nodos hojas pueden contener más de un dato asociado. En general, se permite un número máximo reducido y fijo de puntos asociados a cada nodo hoja. Para que el problema se presente con esta variación, debe haber más datos con distancias entre sí muy pequeñas que los admitidos para un solo nodo hoja. Así, la probabilidad de que esto ocurra se reduce notablemente. Otra solución alternativa sería utilizar para dividir el espacio de un nodo interno en los ocho subespacios de sus hijos, planos que no pasen necesariamente por el centro del AABB, sino que dependan de la distribución de los datos que se almacenan (de forma similar a un kD-tree). Pero calcular las posiciones ideales para estos planos puede ser muy costoso, y utilizar una heurística para intentar aproximarla de forma eficiente puede llevar a configuraciones fuertemente desbalanceadas para casos simples si los datos se ingresan con cierto orden.

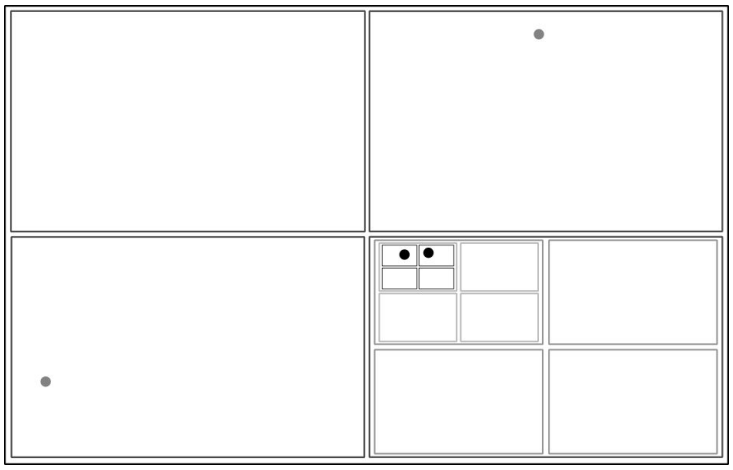

Figura 4.4: Ejemplo de un quadtree con un alto número de divisiones en comparación a la cantidad de puntos que contiene.

4.2.2. Alternating Digital Tree

El Alternating Digital Tree (ADT[57]) es un tipo de árbol que suele utilizarse para elementos de una malla, ya que permite consultar eficientemente los axis-aligned bounding-box (AABB) de sus elementos. Se basa en la utilización de un árbol de búsqueda similar a un árbol binario, y en un método para representar un bounding-box de un elemento N-dimensional como un punto en un espacio $2N$-dimensional.

Árboles N dimensionales

En un árbol binario, como se mencionó anteriormente, cada nodo tiene como máximo dos hijos (que se denominan hijo izquierdo e hijo derecho). La mayoría de las aplicaciones utilizan los árboles binarios para ordenar los datos y/o facilitar las búsquedas. Si el árbol almacena datos unidimensionales, se puede comparar el valor de un nodo, con el valor que se quiere encontrar/agregar para determinar si el algoritmo de inserción/búsqueda debe proceder por el sub-árbol del hijo izquierdo o por el sub-árbol del hijo derecho de un nodo dado. De esta manera, los datos se almacenan en el árbol directamente de forma ordenada. Es decir, al recorrer el árbol con un recorrido en profundidad "inorden", los valores almacenados se recuperan en orden creciente (o decreciente, según el criterio de comparación utilizado al insertarlos). En un recorrido en profundidad inorden (figura 4.5), a partir de un nodo, se avanza al siguiente por el izquierdo recursivamente. Al llegar a una hoja, se recupera el valor asociado al nodo hoja, se retrocede al nodo padre del nodo

hoja, se recupera también su valor, y se avanza al hijo derecho para repetir nuevamente el mismo procedimiento.

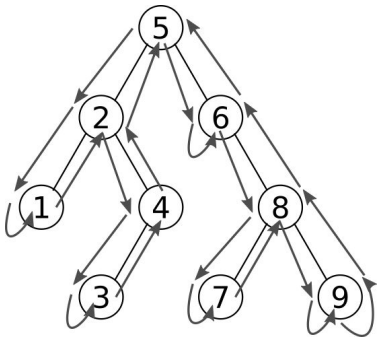

Figura 4.5: Ejemplo de recorrido inorden de un binario. Un nodo se *lista* cuando la flecha que indica el recorrido ingresa al nodo. Esto solo ocurre cuando proviene del hijo izquierdo, pero no cuando se desciende en el árbol, o se retorna a un nodo desde un hijo derecho.

Gracias a esta propiedad, es sencillo y eficiente consultar todos los valores almacenados que se encuentran en un rango determinado. Para ello, simplemente se busca en el árbol la posición donde se insertaría un extremo del rango, y se procede a partir de allí con un recorrido en profundidad inorden hasta llegar a la posición donde se insertaría el otro extremo del rango. Si el árbol se encuentra bien balanceado, las dos búsquedas necesarias serán $O(\log n)$, y el recorrido por los datos del rango será $O(k)$ donde k es la cantidad de resultados (datos del árbol en el rango de búsqueda).

Cuando los datos tienen más de una dimensión, si el número de dimensiones no es elevado, existen diferentes alternativas para ordenarlos mediante uno o más árboles, de modo que las consultas por rango resulten igualmente eficientes. Si los datos almacenados son fijos, el N-dimensional *range-tree* es una de las estructuras de datos más adecuadas. En esta estructura, se confecciona primero un árbol binario ordenando los nodos según una de sus coordenadas. Luego, a cada nodo se le asocia un nuevo árbol binario que contiene los datos del nodo y de todos sus descendientes en el primer árbol, pero ordenados según otra de sus dimensiones. El mecanismo se repite por cada dimensión. De esta forma, para buscar datos en un rango N-dimensional, se realizan búsquedas 1-dimensionales. El rango se descompone en rangos $1D$ (uno por cada dimensión de los datos). Se busca el rango $1D$ en el árbol de la primer dimensión, y en cada sub árbol asociado a los nodos encontrados en esta primer búsqueda, se repite el mismo procedimiento considerando ahora el rango en la siguiente dimensión. Así, se obtienen los datos en el rango dado combinando múltiples búsquedas $1D$ sobre sub-árboles. Se pue-

de demostrar que este árbol se puede construir en tiempo $O(n \log^{d-1}(n))$, que el espacio requerido para almacenar todos los árboles y sub-árboles es $O(n \log^{d-1}(n))$, y que las consultas se obtienen en tiempo $O(n \log^{d}(n+k))$, donde k será el número de resultados[58]. Se confirma que para dimensiones bajas la estructura resulta eficiente, pero su principal desventaja radica en el costo de actualizarla cuando se agregan o eliminan nodos en el mismo (pues se deben mantener todos los sub-árboles sincronizados y balanceados).

En un único árbol para datos 1D, es relativamente fácil agregar o quitar nodos. Más aún si los datos se almacenan sólo en las hojas, ya que nunca se eliminará un nodo interior. En este caso, los nodos interiores sólo sirven para subdividir el espacio, de forma similar a lo que ocurre en un octree. Para aprovechar esta propiedad permitiendo a la vez almacenar datos multidimensionales, se puede utilizar un árbol alternado. Esto significa que en cada nodo se utiliza una sola de las N-dimensiones para comparar un valor almacenado (el que divide el espacio que representa en dos sub-espacios), pero con la particularidad de que la dimensión elegida varía de nodo en nodo. Esta dimensión varía de forma cíclica conforme crece el nivel de profundidad del nodo en cuestión. Por ejemplo, para datos 3D, en el primer nivel (la raíz) se compara con x, en el segundo nivel con y, en el tercero con z, en el cuarto con x nuevamente, y así sucesivamente. De esta forma, el árbol crece en profundidad de forma proporcional a la cantidad de dimensiones. Nuevamente, si la cantidad de dimensiones no es alta, el crecimiento puede considerarse constante, manteniéndose así las propiedades enunciadas para un árbol binario (las búsquedas serán ahora $O(d \log n)$). Por la simpleza de esta estructura de datos, resulta relativamente fácil implementarla a partir de un árbol binario simple. La implementación utilizada en los capítulos posteriores de este trabajo toma además algunas de las propiedades descritas para un bucketed-octree, ya que almacena sus datos solo en las hojas, pudiendo estas almacenar una cantidad fija de nodos (pequeña pero mayor a 1), y en cada nivel realiza particiones n-arias y equiespaciadas (evaluaciones de performance demostraron que para esta aplicación en particular el número óptimo de hijos por nodo se encuentra entre 3 y 5). Dicha implementación se describe en el apéndice A.3.

Representación de rangos como puntos

Ya se describió como utilizar estructuras basadas en árboles binarios/ternarios para almacenar eficientemente puntos y consultar la lista de puntos almacenados cuyas coordenadas se encuentren dentro de un rango dado. En la mayoría de los algoritmos de mallado basados en avance frontal, se requie-

re verificar si un potencial nuevo elemento se intersecta con los elementos ya colocados en el frente de avance. Los elementos a intersectar serán aristas, caras, o sólidos. Para poder realizar un descarte masivo y temprano de elementos, y evitar así calcular un gran número intersecciones (todos contra todos), se requiere una estructura de ordenamiento espacial. Para simplificar el problema, se calcula y almacena en la estructura de ordenamiento espacial el AABB de cada elemento. Se requiere entonces que la estructura permita consultar cuales AABB se intersectan con el AABB del elemento a agregar. El ADT utiliza un árbol alternado $2N$-dimensional para almacenar y consultar AABBs de N dimensiones. Para ello se define cómo convertir un AABB N-dimensional en un punto $2N$-dimensional (para los elementos a almacenar), y también cómo convertir un AABB N-dimensional en un rango $2N$-dimensional (para el elemento contra el que se verificarán las intersecciones). Es decir, el AABB del elemento a insertar generará un rango de búsqueda para el árbol de los AABBs de los elementos insertados previamente.

Dado un AABB definido por las coordenadas N-dimensionales de sus extremos opuestos $\mathbf{X}_{min}$ y $\mathbf{X}_{max}$, el punto $2N$-dimensional $\mathbf{R}$ que lo representa y que será efectivamente almacenado en el árbol se obtiene de la siguiente manera:

$$\mathbf{R} = \{\mathbf{X}_{min}, \mathbf{X}_{max}\} \tag{4.1}$$

Sean $\mathbf{L}_{min}$ y $\mathbf{L}_{max}$ los límites del espacio total. Dado el AABB del área para la cual se quiere realizar la consulta, definido igualmente por las coordenadas de sus extremos opuestos $\mathbf{Q}_{min}$ y $\mathbf{Q}_{max}$, los puntos que delimitan el rango sobre el que efectivamente se realiza la búsqueda (el rango $[\mathbf{A}; \mathbf{B}]$) en el árbol se obtienen como:

$$\mathbf{A} = \{\mathbf{L}_{min}, \mathbf{Q}_{min}\} \quad \wedge \quad \mathbf{B} = \{\mathbf{Q}_{max}, \mathbf{L}_{max}\} \tag{4.2}$$

Al consultar los puntos $\mathbf{R}^k = \{\mathbf{X}^k_{min}, \mathbf{X}^k_{max}\}$ pertenecientes al rango $\mathbf{R}^k | \mathbf{A} <= \mathbf{R}^k <= \mathbf{B}$, descomponiendo la comparación según las primeras N dimensiones y las segundas N dimensiones obtenemos que los puntos resultantes cumplen que:

$$\mathbf{L}_{min} <= \mathbf{X}^k_{min} <= \mathbf{Q}_{max} \quad \wedge \quad \mathbf{Q}_{min} <= \mathbf{X}^k_{max} <= \mathbf{L}_{max} \tag{4.3}$$

Todos los puntos almacenados cumplen que $\mathbf{L}_{min} <= \mathbf{X}^k_{min}$ y $\mathbf{X}^k_{max} <= \mathbf{L}_{max}$, ya que $\mathbf{L}_{min}$ y $\mathbf{L}_{max}$ representan los límites del espacio que contiene la totalidad de los datos (el AABB del conjunto completo). Por lo tanto, los puntos que se descartan en esta búsqueda son los que cumplen:

$$\mathbf{X}^k_{min} > \mathbf{Q}_{max} \quad \wedge \quad \mathbf{Q}_{min} > \mathbf{X}^k_{max} \tag{4.4}$$

La primer condición descarta los AABB cuyo límite superior izquierdo sea mayor al límite inferior derecho del área de búsqueda. La segunda condición descarta los AABB cuyo límite inferior derecho sea menor al límite superior izquierdo del área de búsqueda (ver figura 4.6). Se obtiene así por resultado, el conjunto de AABBs que se intersectan efectivamente con el área de búsqueda.

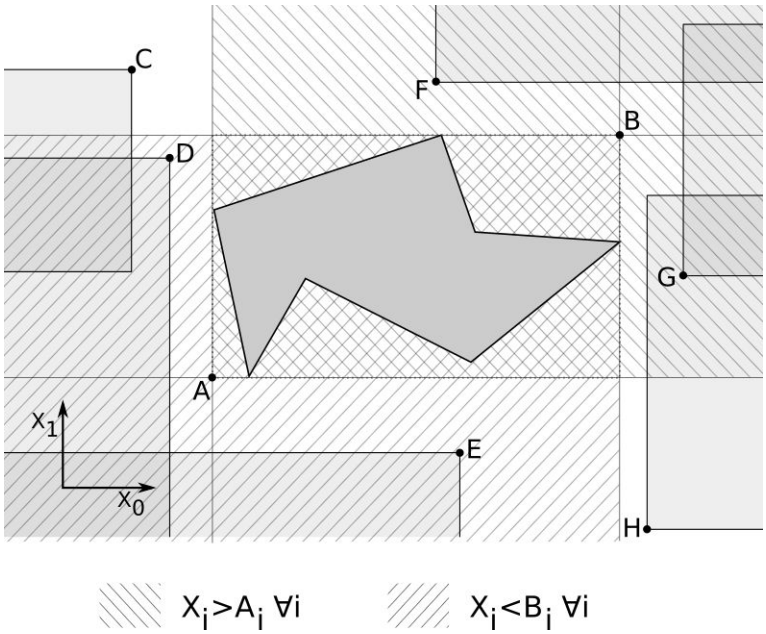

Figura 4.6: Las zonas rayadas indican las regiones para las cuales se cumple cada condición para un objeto 2D. Todo AABB de otro objeto cuyo punto de máximas coordenadas no se encuentre en el área delimitada por A (como C, D y E) será descartado para el cálculo de intersecciones. De igual forma, se descarta también cualquier otro AABB cuyo límite inferior no se encuentre en la zona delimitada por B (como F, G y H).

Distribución de puntos y balance en un ADT

Se debe notar que para todos los puntos que se almacenan en un ADT, dado que dichos puntos representan cada uno un AABB ($\mathbf{R} = \{\mathbf{X}_{min}, \mathbf{X}_{max}\}$) se cumple que $\mathbf{X}_{min} <= \mathbf{X}_{max}$. Si tomamos, por ejemplo AABB $1D$ para puntos en el rango $[0; 1]$, el espacio de los puntos que efectivamente se almacenan en el árbol es $2D$ y todos los puntos almacenados en el árbol estarán sobre un cuadrado unitario. Considerando la propiedad recién mencionada, todos los puntos se ubicarán por encima de una de las diagonales de dicho cuadrado (la recta $x = y$). Si el ADT divide en cada nodo el espacio que le corresponde en dos sub espacios iguales, se puede pensar que esta distribución de puntos generará un fuerte desbalance en el árbol resultante. En el

cuadrado de 1×1 del ejemplo de la figura 4.7, el nodo raíz dividirá el espacio dejando el rectángulo que va del punto $(0, 0)$ al punto $(0.5, 1)$ asociado al hijo izquierdo, y el rectángulo que de $(0.5, 0)$ a $(1, 1)$ asociado al hijo derecho. Sólo $1/4$ del área del hijo derecho puede contener puntos, mientras que para el hijo izquierdo la proporción es $3/4$ (en ambos casos el área sobre la recta $x = y$).

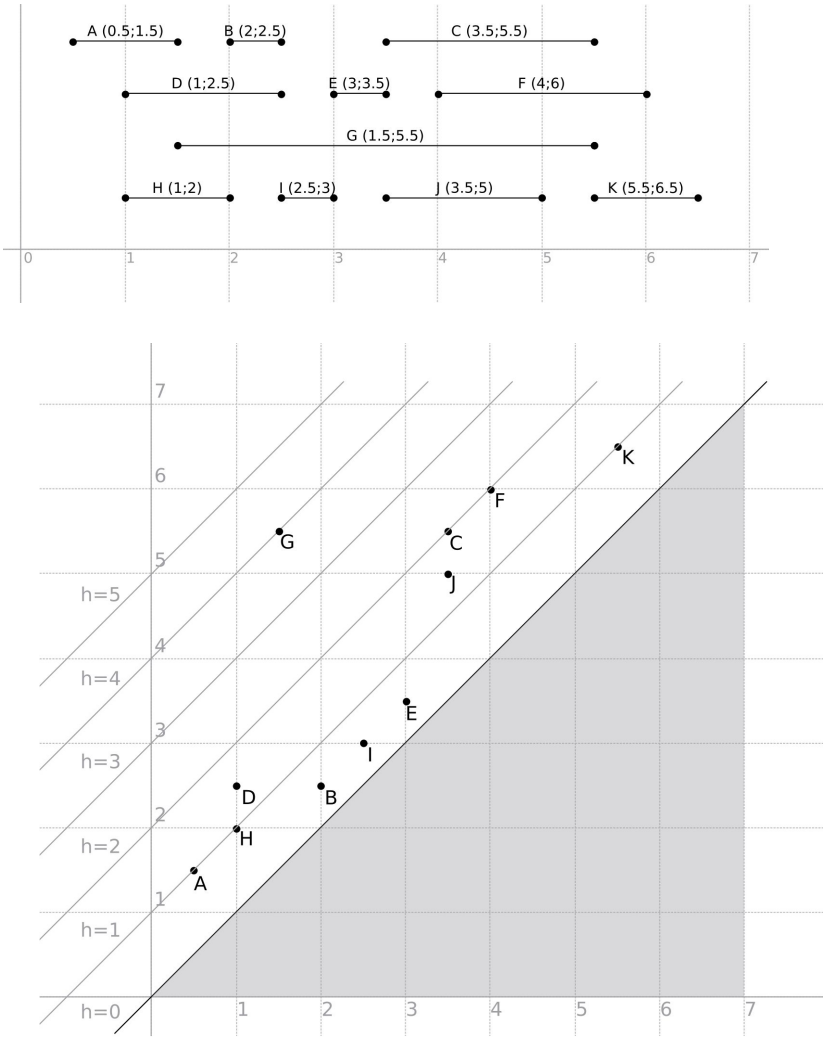

Figura 4.7: Segmentos 1D representados como puntos en un plano (pares formados por las coordenadas de los extremos del segmento). La mayoría de los elementos se ubica en la zona entre las diagonales h=.5 y h=2. G (el segmento más largo y alejado del promedio) es el único elemento fuera de esta zona.

Sin embargo, esto no será un problema para las aplicaciones que se presen-

tarán en este trabajo, ya que el ADT será utilizado para almacenar aristas, caras, o elementos de una malla. En el ejemplo, los puntos cercanos a la diagonal serán puntos con $x_{min} \simeq x_{max}$, mientras los puntos cercanos al vértice $(0, 1)$ serán puntos para los cuales $x_{min} \ll x_{max}$. Representando elementos de una malla, los primeros representan elementos excesivamente pequeños, mientras que los últimos elementos demasiado grandes, por lo que ambos casos serán muy poco frecuentes. En realidad, si el conjunto de nodos de la malla está distribuido de forma adecuada para obtener un tamaño de elemento relativamente constante (lo cual se cumple efectivamente por zonas), los puntos se agruparán en una zona con forma de banda paralela a la recta $x = y$, donde el ancho de la banda estará directamente relacionado con la variabilidad en el tamaño de los elementos de la malla. El ancho de la banda en comparación al del cuadrado será menor cuanto más grande/refinada sea la malla. Además, también cuanto más grande/refinada sea la malla más cerca de la diagonal se encontrará esta banda. De forma que a medida que el problema de mallado crece, la forma de la banda tiende a la línea diagonal. Esto es completamente cierto para un ejemplo 1D (segmentos) como el de la figura 4.7. Para AABB N-dimensionales, si se analizan las proyecciones individuales de cada coordenada, la banda efectivamente llega hasta la diagonal, ya que alguna de las dimensiones de un AABB puede ser 0 (aunque nunca pueden serlo todas a la vez).

Si consideramos una distribución uniforme de puntos sobre la diagonal (caso límite) para el ejemplo del árbol 2D, cada nodo que divida su espacio en X, deja igual cantidad de puntos a ambos lados, mientras que para cada hijo siguiente, las divisiones en Y dejarán todos los puntos de un solo lado. Esto ocurre en cada nivel. Puede decirse entonces que en un problema $2N$-dimensional, tendremos cada $2N$ niveles, N niveles con sub-árboles con igual cantidad de nodos y N niveles completamente desbalanceados. Podemos decir que la profundidad del árbol será entonces 2 veces la ideal. Este caso, en el cual los puntos del ADT se encuentra sobre la diagonal, es un caso límite e irreal donde todos los AABB son de tamaño 0. En un caso real, donde la banda que contiene a los puntos se aproxima a la diagonal, pero no coincide, las divisiones en X dejan a cada lado áreas similares de dicha banda, y las divisiones en la coordenada Y dejan de ser inútiles y permiten compensar estas diferencias. En conclusión, para problemas verdaderamente grandes, el costo de la búsqueda de un punto en un ADT con esta organización será aproximadamente $O(\log n)$.

4.3. Grillas Regulares

Uno de los métodos más simples de ordenamiento consiste en particionar cada uno de los ejes del espacio en partes iguales, formando así una grilla con celdas rectangulares o hexaédricas alineadas con los ejes y de idéntico tamaño (de allí que se diga "regular"). La ventaja de emplear divisiones regulares en cada coordenada radica en que, de esta forma, si la grilla se almacena en una estructura de acceso aleatorio como un vector de celdas, encontrar la celda que contiene a un punto dado es una operación trivial y de tiempo constante (con una constante muy baja en comparación a otros métodos). Además, esta clase de operaciones hacen en general un buen aprovechamiento de los distintos niveles de cache si los datos de la celda se representan de forma relativamente compacta. Operaciones como la inserción y eliminación de puntos serán prácticamente constantes en grillas suficientemente refinadas, y las búsquedas de vecindades triviales. Se puede pensar que esta estructura es comparable a una tabla de hash, donde la función de hash es una operación algebraica muy simple que involucra a las coordenadas del punto y al tamaño de celda. Pero, como ventaja frente a una estructura de hashing genérica, la distribución de las celdas (gracias a su función de hashing particular) guarda una relación directa y aprovechable con la distribución en el dominio del problema de los elementos que éstas contienen.

Sin embargo, esta forma de particionar el espacio adolece de un defecto importante: no considera la distribución de los elementos en el mismo. Es decir, la forma de hacer la partición no depende de que los puntos estén o no distribuidos por todo el dominio y relativamente equiespaciados, condición necesaria para que así cada celda resultante contenga aproximadamente la misma cantidad de puntos. Dadas estas condiciones, una implementación general debe permitir almacenar una cantidad arbitraria de puntos por celda (una variante de tipo open-addressing complicaría luego las búsquedas por rango). Al utilizarla para almacenar los nodos de una malla, en las zonas donde la malla sea más densa/refinada las celdas contendrán muchos puntos, mientras que en las zonas de menor densidad/refinamiento cada celda contendrá comparativamente pocos puntos y muchas de ellas permanecerán vacías. Al utilizarla para los nodos de un método de partículas, es de esperar que las diferentes densidades no sean en verdad demasiado diferentes, ya que estos métodos remueven partículas cuando se acercan demasiado entre sí (para mantener la velocidad del cálculo), y las siembran cuando se distancian (para mantener la precisión de la solución). Entonces, en este tipo de aplicaciones, la diferencia de densidad entre distintas zonas de una malla o nube de puntos no será un factor condicionante.

Además, este tipo de grilla (como la mayoría de las estructuras de ordenamiento espacial), define los límites del espacio representado mediante valores extremos en cada coordenada (planos perpendiculares a los ejes). En consecuencia, una grilla que cubre un dominio determinado, debe cubrir al menos el AABB del mismo. Si el área que cubre la malla o la nube de puntos no es aproximadamente rectangular, la grilla tendrá celdas contenidas en el interior del dominio, celdas en el exterior, y celdas sobre la frontera. Suponiendo una grilla considerablemente refinada, se observará entonces que la mayoría de las celdas estarán en uno de dos grupos predominantes: el grupo de celdas exteriores (sin nodos), y el grupo de celdas interiores (con cantidades de nodos comparables entre sí, por lo argumentado en el párrafo anterior). De esta forma, la eficiencia de la grilla está en principio condicionada por la relación que exista entre las cantidades de celdas en cada uno estos dos grupos.

Por ejemplo, suponiendo que se tiene 3 nodos por celda para cada celda en un dominio 2D, buscar el nodo más cercano a un punto requiere explorar 9 celdas, y por ende $3 \times 9 = 27$ nodos. Como el tiempo de acceso a la celda (y por ende a su nodo) es constante, y con una constante baja, el tiempo total de la búsqueda también puede considerarse constante, y con una constante comparativamente baja. Si el conjunto de puntos cubre una zona cóncava, las celdas que coincidan con la o las concavidades no tendrán nodos. A igual cantidad de celdas que en el ejemplo anterior, esto significa que las celdas que coincidan con el interior contendrán una cantidad de puntos promedio superior. El tiempo de la búsqueda crecerá linealmente con el promedio de puntos por celda, aumentando conforme disminuye el factor de cubrimiento de la nube de puntos sobre su AABB, y la grilla contendrá un mayor número de celdas ocupando espacio memoria sin utilidad alguna.

En conclusión, la simplicidad y la poca flexibilidad de una grilla llevan a pensar que esta estructura sólo es útil en casos ideales, pero que en general no es una alternativa competente en comparación con otras estructuras basadas en árboles como octrees o kd-trees. Sin embargo, por ser una estructura más compacta en memoria, de acceso directo en tiempo constante, y más adecuada para el aprovechamiento de la memoria cache, se puede apreciar mediante benchmarks que en la práctica el uso de esta estructura supera en rendimiento a las anteriormente mencionadas en muchos casos, como los que se describirán en algoritmos de triangulación y tetraedrización en los capítulos 5 y 6.

4.4. Recorrido en orden utilizando estructuras de ordenamiento espacial

Los algoritmos de mallado que se desarrollarán en los capítulos 5, 6 y 7 harán uso de las estructuras de ordenamiento espacial por celdas presentadas para almacenar los puntos de la malla. Requerirán realizar eficientemente una consulta que permita obtener dichos puntos en un orden dado por la distancia de los mismos a un punto origen arbitrario. La consulta deberá responderse en forma incremental, es decir, permitiendo obtener los puntos de a uno en diferentes pasos, comenzando por el más cercano y pudiendo interrumpir la búsqueda en cualquier momento sin que los puntos no recuperados por la misma hayan generado costo alguno en el proceso. Para responder a esta consulta se propone la siguiente metodología: utilizar una cola de celdas pendientes de recorrer, colocando en la misma inicialmente solo la celda que contiene al punto origen, y añadiendo luego (bajo demanda) las celdas vecinas a cada celda procesada. Este mecanismo puede aplicarse a cualquier estructura de ordenamiento espacial que utilice una partición real del dominio, aunque en el caso de la grilla regular la determinación de vecindades es una operación trivial, mientras que en el caso del octree, por ejemplo, deja de serlo: pasar de una celda a otra vecina puede requerir recorrer varios nodos del árbol (subir primero, y luego bajar varios niveles de profundidad).

Esta metodología puede encapsularse en una clase especial que presente una interfaz abstracta e independiente de la estructura sobre la cual opera, similar a la de un iterador. Pero, a diferencia de un iterador, en este caso la estructura sobre la cual itera no es lineal, y el orden en que recorre los elementos está asociado al tipo de consulta, y no a la estructura en sí. Por ello, el iterador debería ser propietario de la cola de celdas pendientes de procesar, y controlar además que no se procese dos veces una misma celda. Añadir y eliminar elementos de una cola pueden ser operaciones de $O(1)$, pero determinar si una celda en particular fue procesada previamente es computacionalmente mucho más costoso si depende solo de este iterador (ya que deberá almacenar una lista adicional de celdas procesadas y realizar búsquedas sobre la misma). Para solucionar este problema se propone una estrategia que requiere de cierta instrumentación de soporte por parte de la estructura de ordenamiento, pero que permite mantener el costo de la consulta en $O(1)$.

La solución consiste en utilizar un flag por celda que indique si la misma ya fue o no visitada. Así, cuando el iterador añade una celda a la cola marca este flag para evitar luego añadirla nuevamente al procesar una celda veci-

na. Si el flag se implementa como un valor booleano, cada nueva búsqueda requerirá el reseteo de todos los flags de la estructura, operación que sería $O(n)$. Para evitar esto, se utiliza como flag un número entero, inicializado en 0 en la construcción de la celda, y un contador adicional general también inicializado en 0 al construir la estructura. El iterador incrementará el valor del contador general al construirse, y lo utilizará para marcar las celdas visitadas (lo asignará como valor del flag en dichas celdas). De esta forma, no es necesario reiniciar los flags al comenzar una nueva búsqueda, ya que cada búsqueda utiliza un nuevo valor del contador global que no ha sido utilizado previamente. Solo en el caso en que este contador se desborde se deberán reiniciar verdaderamente los flags de cada celda. En una implementación típica utilizando un entero de 32 bits, esto ocurre 1 de cada $2^{32} - 1$ veces. Esta solución permite entonces realizar toda la consulta en un tiempo $O(C)$, donde C será la cantidad de celdas visitadas, dado que cada paso requerido es $O(1)$.

La principal desventaja de este enfoque radica en que no podrían realizarse más de una consulta en simultáneo, o más de M si se reemplaza el entero utilizado para almacenar el flag por un arreglo de M enteros. Este último paso solo es razonable si M es suficientemente bajo, y muy sensible a detalles de implementación, ya que organizado en memoria de la forma en que aquí se describe (como arreglos de flags por celda) se tienen grandes probabilidades de generar false-sharing (ver sección 3.4.2). Para evitarlo se debe garantizar la alineación en memoria de los ids de acuerdo al tamaño de las lineas de cache, o bien dividir los arreglos lógicamente asociados a los nodos, en varios arreglos que agrupen en memoria los ids por hilo en lugar de hacerlo por celda.

4.5. Estructuras de datos lock-free

Cabe mencionar la existencia de técnicas de implementación que permiten desarrollar estructuras de datos lock-free[59][60]; es decir, estructuras de datos que pueden modificarse concurrentemente por varios hilos de ejecución sin requerir mecanismos de sincronización para evitar data-races. Estas implementaciones solo requieren que ciertas operaciones muy comunes en programación paralela (ej: CAS/compare-and-swap) sean ejecutadas atómicamente, y que se garantice de alguna forma el orden de ejecución de las instrucciones en cada hilo. Ambos requerimientos son fáciles de satisfacer en un lenguaje como C++11 en las arquitecturas actuales. La idea central para la implementación consiste en garantizar que cuando un hilo consulta una estructura mientras otro la está modificando, el primero vea un estado con-

sistente, aunque no necesariamente actualizado de la misma (es decir, podría consultar percibiendo el estado viejo sin cambios, aunque la modificación esté casi completa).

La técnica para lograrlo se basa usualmente en hacer que el hilo que realiza la modificación construya en memoria y de forma aislada la parte modificada de la estructura, y la enlace a la misma al final del proceso en un solo paso atómico que generalmente consiste en la modificación de un puntero. Por ejemplo: si hay que reemplazar una rama de un árbol, se construye primero la rama nueva (mientras tanto cualquier otro hilo que consulte la estructura verá la vieja), y una vez finalizada, se cambia el puntero desde la rama vieja hacia la nueva (mediante una operación de tipo CAS, que puede fallar y requerir uno o unos pocos reintentos), momento a partir del cual cualquier consulta verá la rama nueva y será entonces seguro eliminar "conceptualmente" la vieja. La aclaración de eliminar "conceptualmente" se debe a que otro hilo puede haber consultado recientemente la rama vieja y estar recorriéndola aún al momento de eliminarla. Para que esto no sea un problema se deben utilizar mecanismos alternativos de recolección de basura o punteros inteligentes (similares al std::shared_ptr de C++11) o marcados con algún indicador de su antigüedad (tagged-pointers/hazard-pointers) que ofrezcan ciertas garantías sobre los bloques de memoria liberados para evitar problemas tales como el denominado ABA[61]. Más detalles de la técnica se pueden encontrar en [62] y [63] y un ejemplo introductorio en [60].

No fue necesario ni en ciertos casos posible aplicar estas técnicas a las implementaciones de los algoritmos de tetraedrización en paralelo que se describirán en este trabajo debido a que las estructuras de datos utilizadas o bien son más complejas que los ejemplos resueltos en la literatura actual (por ejemplo, no es directa la extensión del mecanismo de la lista simplemente enlazada a una lista doblemente enlazada), o bien requieren ciertas operaciones que generarían otro costo adicional. Sin embargo, dado que muchas de las estructuras requeridas, o bien se utilizarán exclusivamente (aunque sea por partes) en un único hilo, o bien serán solo consultadas (y no modificadas) concurrentemente, serán muy pocos los casos donde las estructuras requieran efectivamente sincronización, y por ello el impacto en el rendimiento del proceso completo no justificará mayores esfuerzos en este sentido.

Capítulo 5

Triangulación de un conjunto de puntos

5.1. Algoritmo DeWall

El algoritmo DeWall[44] permite obtener una triangulación o tetraedrización Delaunay a partir de un conjunto de puntos prefijados (impuestos). Para simplificar la explicación, se considerará inicialmente un problema 2D (triangulación), y se presentará sobre el final de esta sección su generalización a 3D. El área que abarca la triangulación se corresponde con el interior del convex-hull del conjunto de puntos. De todas las posibles triangulaciones para el conjunto de puntos y su convex-hull, permite obtener una triangulación Delaunay. Los datos de entrada son solamente las coordenadas de dichos puntos. El algoritmo genera en cada paso, para un dominio (volumen) a triangular, un conjunto de elementos que permite particionar a dicho dominio en tres subconjuntos: uno que contiene el área ya triangulada (que corresponde a los elementos generados en ese paso), y dos subconjuntos correspondientes a las dos áreas sin triangular (que serán disjuntas entre sí). Se denominará trabajo a la tarea de generar el conjunto de elementos que da lugar a la partición. La resolución de un trabajo genera nuevos subtrabajos, uno para cada uno de los dos subconjuntos que no se corresponden con el de los elementos ya generados. El mismo algoritmo se aplica recursivamente en cada uno de estos subconjuntos, a menos que resulten nulos. La figura 5.1 muestra la resolución de los 3 primeros trabajos en un ejemplo concreto, junto con los límites de los subtrabajos generados por cada uno de ellos.

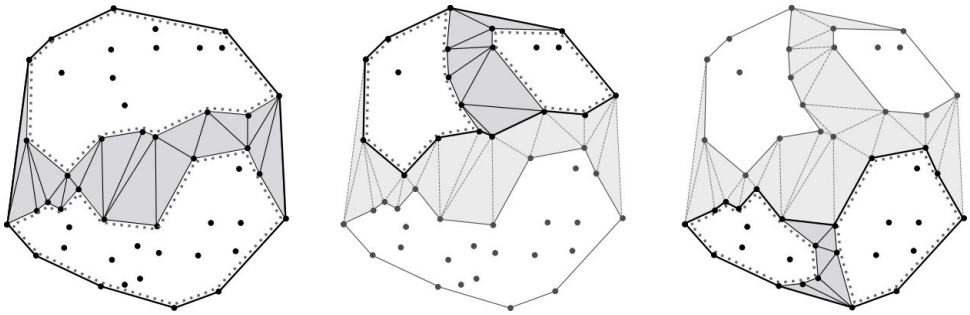

Figura 5.1: Primeros pasos del algoritmo: en cada paso la región sombreada corresponde a la parte ya triangulada, mientras que las regiones delimitadas por lineas de punto corresponden a los subdominios para los nuevos trabajos que se van generando y procesando recursivamente.

En cada paso de la recursión, para un dominio a triangular dado, se define una recta que divide al dominio en dos partes. Se construyen luego todos los elementos de la triangulación que se intersectan en al menos un punto con dicha recta. De esta forma, los elementos construidos cubren una parte del dominio, que garantiza la no-intersección de las áreas sin triangular a derecha e izquierda de la recta. Cualquier curva que atraviese el dominio a triangular puede ser una linea divisoria válida. Para simplificar los cálculos necesarios para la construcción de los elementos, suele ser conveniente utilizar una recta paralela a uno de los ejes coordenados. Además, para que los subdominios generados (donde se aplicará el algoritmo recursivamente) requieran esfuerzos computacionales similares (lo cual será deseable luego al paralelizar el algoritmo), es conveniente utilizar una recta que divida al dominio en partes similares bajo algún criterio (área que abarcan los subdominios, cantidad de puntos que quedan a cada lado, etc). Criterios simples de aplicar son los que se basan en la media o mediana de las coordenadas del conjunto de puntos.

Para generar los elementos que se intersectan con la recta divisoria, en [44] se propone un algoritmo basado en un criterio presentado en [64] para la generación de una triangulación Delaunay. En un paso cualquiera de la recursión, dada la recta divisoria, y una arista cualquiera de la triangulación que se intersecte con la recta divisoria, se busca un tercer punto para generar un triángulo con dicha arista (que denominaremos arista base). En [64] se propone un mecanismo para determinar de entre los puntos del conjunto, cual de ellos formará un triángulo que respete las condición Delaunay para una arista base dada. Si se considera una orientación para la arista, la recta que la contiene separa el conjunto de puntos en dos grupos de puntos, que generarán triángulos con dicha arista con diferente orientación. Con este criterio se descarta uno de esos grupos (el que generaría elementos invertidos según la

orientación de la triangulación). Midiendo la distancia con signo (de acuerdo a la orientación de la arista) de un punto a la recta, los puntos factibles presentan distancias positivas, mientras que los puntos descartados distancias negativas.

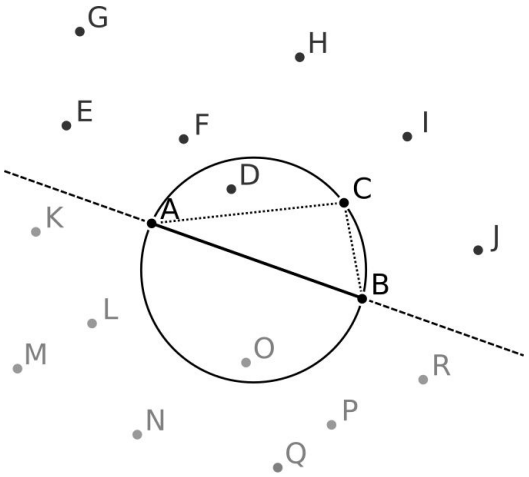

Figura 5.2: Para la arista base AB, los puntos C a J son factibles, mientras que los puntos K a Q se descartan por formar elementos invertidos (suponiendo ordenamiento CCW para esa definición de la arista base). Se marcan el círculo y el triángulo correspondientes a un punto factible seleccionado temporalmente como candidato (C).

Para seleccionar el punto adecuado de entre los factibles (ver figura 5.2) se utiliza el siguiente criterio: si para cada uno se construye un círculo definido por el punto y los dos extremos de la arista, y se miden las distancias (con signo) de los centros de cada círculo a la recta que contiene a la arista base, se debe utilizar el punto para el cual esta distancia sea menor (o más negativa). Si se fija una arista base y se construye un círculo con un tercer punto, se observa que al mover el tercer punto de forma que disminuya (se haga más negativa) la distancia del centro a la recta de la arista, la intersección entre el semiplano factible y el círculo se reduce, y lo hace de forma que cada nuevo segmento circular (la intersección del círculo con el semiplano) está completamente incluido en el anterior (ver figura 5.3). Cuando el centro del círculo presenta una distancia positiva, al mover el punto móvil hacia el interior del círculo, este reduce su tamaño, reduciendo así la distancia del centro a la arista. Cuando la distancia es cero (el centro se encuentra sobre la recta), mover el punto hacia interior del círculo hace que el círculo sea cada vez más grande, alejando el centro de la recta, pero en la dirección de las distancias negativas.

El algoritmo debe evitar que la parte del círculo que se encuentra en el

semiplano negativo contenga puntos en su interior, pues en caso de hacerlo la arista base no sería una arista Delaunay válida (perteneciente a la triangulación Delaunay). Si el algoritmo parte de una arista Delaunay, dado que siempre genera triángulos Delaunay, esta situación no puede ocurrir. Se puede argumentar entonces que para el conjunto de puntos factibles, utilizar este criterio para elegir entre dos puntos candidatos para una arista, equivale a aplicar el test de la circunferencia de Delaunay si la arista base es una arista Delaunay. El test de la circunferencia determina si un punto está en el interior del círculo que define un triángulo. Si la arista es una arista de frontera no hay puntos del otro lado. Si la arista es una arista interior, ya se definió al generarla un círculo previo con otro punto del otro lado de la recta, para el cual se verificó la condición.

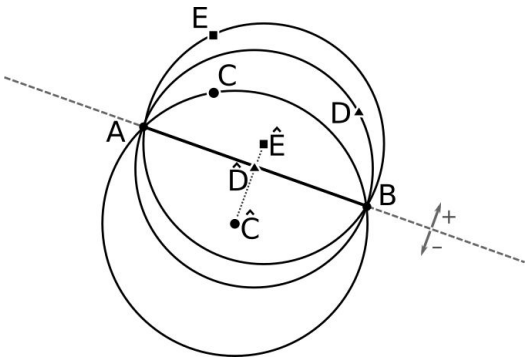

Figura 5.3: Tres círculos para tres puntos factibles (C, D y E), con sus respectivos centros ($\hat{C}$, $\hat{D}$ y $\hat{E}$). La distancia (con signo) a la recta base más negativa es la del punto C, que se encuentra dentro de la circunferencia que define el punto D, que a su vez se encuentra dentro de la circunferencia que define el punto E.

Cuando se construye un triángulo a partir de una arista base, se generan dos nuevas aristas. Si la arista base se intersectaba con la recta divisoria (precondición que siempre debe cumplirse), una de las nuevas aristas generadas también lo hará. Por lo tanto, a menos que esta arista ya haya estado presente en la triangulación (porque en un paso anterior ya se generó el otro triángulo al cual pertenece), podrá ser utilizada como nueva arista base para una iteración posterior de este mismo trabajo. La arista restante (que no se intersecta con la recta) puede ser también considerada arista base, pero en otro trabajo posterior que utilice una recta divisoria diferente. Cuando no se encuentran puntos factibles para una arista base, se puede concluir que esa arista pertenece a la frontera del dominio del trabajo (si es además frontera del dominio original, a la frontera del convex-hull).

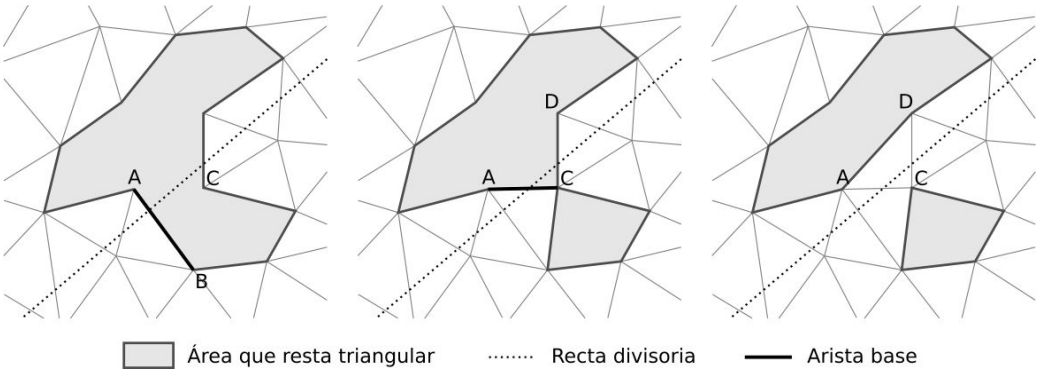

Figura 5.4: La figura muestra tres pasos del algoritmo. En cada uno se muestra la recta divisoria y se resalta el conjunto de aristas bases pendientes, y la arista base seleccionada para el próximo paso.

La figura 5.4 muestra tres estados intermedios de un ejemplo 2D. En el primero, se ha seleccionado la arista AB como base para el siguiente triángulo a insertar. Se muestran además todas las otras aristas bases que restan procesar (tanto las que procesará el trabajo actual como las que quedarán pendientes para futuros subtrabajos). Se denominará "frente de avance" a dicho conjunto. Al insertar el elemento ABC se quita del frente a la arista AB y se insertan las aristas CB y AC. En el siguiente paso, se tomará como arista base a AC y se insertará el elemento ACD. Dado que la arista CD ya se encontraba en el frente de avance (como DC), se elimina del mismo. Se dice que el frente se cierra sobre DC, dejando ahora dos subconjuntos de aristas disconexos.

La extensión a 3 dimensiones del algoritmo explicado hasta este punto es directa (ver figura 5.5). Se construyen tetraedros a partir de triángulos base buscando minimizar las distancias de los centros de las esferas que definen los tetraedros a los triángulos bases. Dado un volumen a mallar, se utiliza en cada trabajo un plano divisor, y se busca construir todos los tetraedros que se intersecten con dicho plano.

Para completar el desarrollo, resta definir cómo obtener la primera arista base al inicio de cada paso de la recursión. Para ello, en [44] se propone el siguiente mecanismo: dada la recta divisoria, seleccionar el punto más cercano a la recta y luego buscar el punto más cercano a la recta de entre los puntos que se encuentran del otro lado de la misma (en el semiplano opuesto según la división que genera la propia recta). Esos puntos formarán la arista base inicial, que debe ingresarse al conjunto de aristas pendientes de procesar dos veces, una vez con cada orientación posible (ya que podría no ser una arista de frontera). En el caso 3D, se obtiene de la misma forma una arista (con

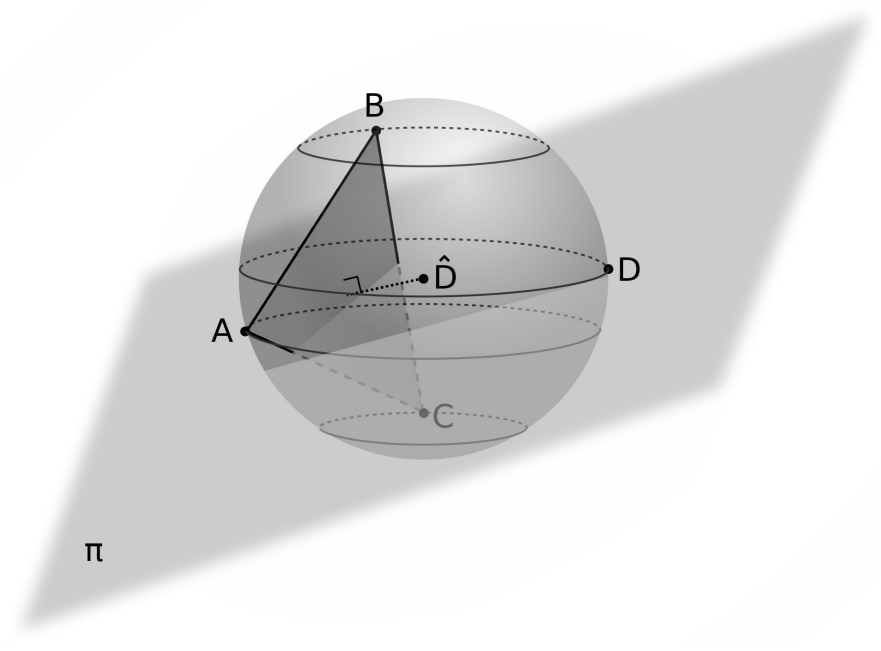

Figura 5.5: Extensión a 3D del test básico del algoritmo. π es el plano divisor, ABC la cara base, D un punto candidato, y $\hat{D}$ el centro de la esfera. En linea punteada se marca la distancia desde dicho centro a la cara base, distancia utilizada para determinar cual es el nodo ganador.

un punto a cada lado del plano), y se busca un tercer punto para formar el triángulo base de forma que minimice el radio del círculo que definiría dicho triángulo. En la implementación que se desarrolló para este trabajo, este mecanismo solo es utilizado para el primer paso, ya que se torna mucho más costoso al considerar las implicaciones y consecuencias del error numérico en los cálculos involucrados, y los criterios para detectarlo y resolver los virtuales "empates" que pueda generar. En las siguientes iteraciones se toma como arista/cara base alguna de las aristas/caras generadas por un trabajo anterior en la frontera común a ambos trabajos.

5.2. Control del error numérico

El algoritmo descrito en la sección anterior, junto con algunos detalles importantes sobre las estructuras de búsqueda y ordenamiento espacial que se utilizan en su implementación, están en mayor o menor medida desarrollados en [44]. Sin embargo, el algoritmo allí presentado se puede utilizar fácil y directamente, solo para generar triangulaciones o tetraedrizaciones de con-

juntos de puntos en *posición general*[58]. Es decir, cuando las posiciones de los puntos no producen situaciones de ambigüedad. Por ejemplo, asumiendo que al testear las distancias de los centros de las circunferencias a la recta de una arista, la menor distancia siempre es única (nunca se encontrarán 4 puntos cocirculares en 2D, o 5 coesféricos en 3D). Para este problema en particular, el algoritmo de McLain[64] proponía una solución basada en un ordenamiento angular, fácilmente implementable en 2D, sin una extensión directa a 3D. También se ignoran en estos trabajos los problemas asociados a la precisión de los cálculos, problemas que surgen del error numérico inherente a la aritmética de punto flotante y precisión finita. Las situaciones de ambigüedad no solo pueden aparecer en la práctica, sino que en muchas aplicaciones lo hacen con gran frecuencia de forma inevitable. Muchas implementaciones de métodos numéricos para simulación utilizan distribuciones iniciales de partículas o nodos regulares, ya sea por conveniencia para el cálculo de las ecuaciones físicas que rigen el problema, o por la facilidad de generación de este tipo de distribuciones. En muchos casos, cuando la solución es, al menos temporalmente, regular, esta condición puede garantizar uniformidad en la distribución y el muestreo de los datos, y en sus cómputos asociados. Por estos motivos un algoritmo de generación que no resuelva los casos ambiguos o degenerados no tendrá una utilidad real, sino puramente académica. En esta sección se describen las soluciones propuestas para lidiar con estos problemas, que permitirán generar mallas topológicamente correctas aún en condiciones degeneradas.

Ignorando por el momento los errores de precisión, si se analiza una configuración 2D en donde se tienen cuatro puntos formando un cuadrado perfecto (por ende, cuatro puntos cocirculares), el test de Delaunay no permite decidir cual de las dos posibles diagonales del cuadrilátero se debe utilizar para particionarlo en dos triángulos. El test explicado en la sección anterior tampoco, ya que resulta equivalente al test Delaunay. Suponiendo que esta configuración sea parte de un dominio mayor que está siendo mallado por el algoritmo, puede que dos aristas del cuadrilátero lleguen a ser en momentos diferentes aristas base para la construcción de un nuevo triángulo. Si esta ambigüedad se resuelve de forma diferente para cada una de ellas, se generarán elementos parcialmente solapados (cuyas nuevas aristas seguirán generando más elementos solapados en iteraciones posteriores, por no cumplir la condición de cierre).

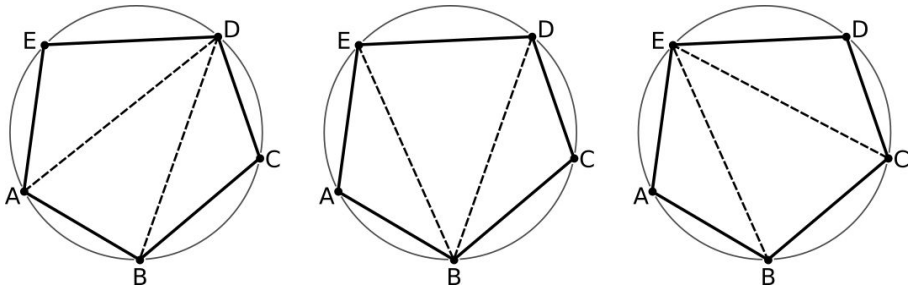

Figura 5.6: Varias configuraciones posibles para un mismo conjunto de puntos cocirculares.

Una solución relativamente sencilla para este problema puede basarse en un control sobre los triángulos ya generados al momento de analizar la factibilidad de colocar uno nuevo (muy común en algoritmos de generación de mallas por avance frontal). Es decir, si se detecta en este caso la ambigüedad, la primer arista del cuadrilátero que se procese podría generar cualquiera de las dos diagonales, pero la segunda tendrá una sola opción compatible con esa primera decisión. Para que este mecanismo sea aplicable se debe considerar que las dos aristas del cuadrilátero se procesan en instantes de tiempo diferentes (lo cual podría no ser cierto en una ejecución en paralelo), disponer de alguna estructura de datos que permita encontrar intersecciones o solapamientos entre un potencial nuevo elemento y los colocados en pasos anteriores (lo cual encarece sensiblemente el algoritmo), y disponer de algún mecanismo para detectar de forma unívoca y consistente los casos de ambigüedad (la consideración más difícil de garantizar de las tres, teniendo en cuenta el efecto del error numérico en los cálculos de centros de circunferencias y distancias). Es por ello que se hace preferible diseñar una solución alternativa que preserve la independencia en la generación de diferentes elementos a partir de diferentes aristas base, característica fundamental del algoritmo para obtener un costo computacional competitivo, y permitir la paralelización posterior del mismo.

La solución que se propone en este trabajo para el proceso de triangulación sin frontera impuesta se basa en dos mecanismos complementarios, a aplicar en diferentes partes del algoritmo, donde cálculos realizados de forma separada deben otorgar resultados consistentes. El primer mecanismo consiste en garantizar un error numérico idéntico en todos los tests geométricamente equivalentes mediante un criterio de ordenamiento único y globalmente reproducible de los operandos que intervengan[1]. El segundo consiste

[1]Esto requiere además cierta uniformidad en el soporte para operaciones de punto flotante del hardware, que se puede obtener utilizando hardware uniforme, o garantizando

en aceptar una tolerancia en las comparaciones de resultados, utilizando un algoritmo para adaptar dinámicamente el valor de dicha tolerancia cada vez que resulte determinante para una decisión. Por ejemplo, si se tiene un algoritmo para obtener el centro de un círculo definido por tres puntos a, b, c, cualquier permutación de ellos arrojaría el mismo resultado si se trabajara con aritmética de precisión infinita. Dado que este no es el escenario real, el orden en que tres puntos a, b, c son asignados a los argumento de una función que implementa dicho algoritmo en una PC puede alterar el resultado. Este cálculo se utiliza, por ejemplo, al testear la distancia para un punto factible a para una dada arista base b, c. El resultado debe ser el mismo si se testea el punto c para la arista base a, b, dado que en este test se basa el control de la propiedad Delaunay de un potencial triángulo. A lo largo de esta sección se denominará "cálculos redundantes" a estos cálculos, que geométricamente representan un mismo concepto (si el triángulo en cuestión es o no Delaunay), pero se procesan de forma diferente (tomando diferente arista como base).

5.2.1. Reordenamiento de operandos en la implementación de expresiones algebraicas

Una de las técnicas empleadas para satisfacer esta garantía consiste en resolver los cálculos redundantes siempre de la misma forma. En el ejemplo del círculo, garantizando que siempre que intervengan tres puntos cualesquiera x, y, z, ingresarán al algoritmo en el mismo orden sin importar cual de ellos era el punto a testear, y cuales los puntos extremos de la arista base. Cuando se puede garantizar que todas las versiones equivalentes de un mismo test geométrico utilizan exactamente el mismo conjunto de operandos, se puede establecer fácilmente un orden, símil lexicográfico, utilizando criterios simples como por ejemplo la numeración de los nodos o de los elementos en la malla. Este paso de ordenamiento que se agrega, puede tener algún impacto en los tiempos de cómputo, pero el impacto es suficientemente bajo ($O(1)$) como para no ser un problema mayor que el que intenta solucionar. Para analizar el costo y las modificaciones al algoritmo original que esto implica, se analizará un ejemplo.

---

el cumplimiento de algún estándar por parte del conjunto de hardware (implementación de la FPU) y software (configuración de la FPU y aplicación de técnicas de optimización de bajo nivel en la compilación). Muchos compiladores dan soporte al IEEE Standard for Floating-Point Arithmetic (IEEE 754 [65][66]) en la mayoría de las plataformas.

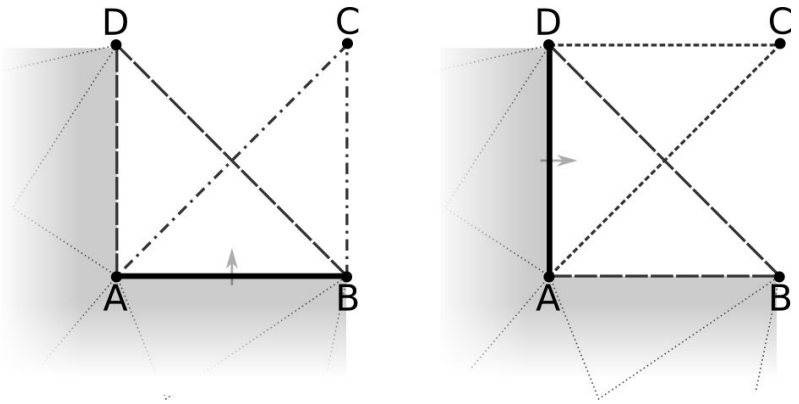

Figura 5.7: Ejemplo de una configuración de los puntos donde la elección de un triángulo entre dos posibles (ambos Delaunay) en una iteración (arista base AB) impone una restricción (no resuelta por DeWall) para otra iteración posterior (arista base DA).

En la figura 5.7, se tiene una configuración 2D de 4 puntos cocirculares. Si en una iteración la arista base es AB, puede formar los triángulo ABC y ABD. Suponiendo que el triángulo ganador sea ABC, en otra iteración podría utilizarse como base la arista DA, teniendo las opciones de generar los triángulos DAB y DAC. Para que la malla sea correcta, esta segunda iteración debería seleccionar como triángulo ganador al triángulo DAC, pues en caso contrario los triángulos ABC (formado a partir de AB) y DAB (formado a partir de DA) se solaparían, generando una malla inválida, tanto métrica como topológicamente (ya que la arista AB tendría 3 triángulos). El ejemplo muestra que para las dos iteraciones (las dos aristas bases propuestas), los nodos que intervienen en los tests son los mismos cuatro, pero los triángulos comparados nunca pueden ser los mismos (ya que en cada iteración hay un triángulo candidato que no contiene a la arista base de la otra). Entonces, en este caso, no hay ordenamiento posible en la forma en que se realizan los cálculos que deciden cual triángulo construir de entre los posibles candidatos para cada arista que garantice resultados compatibles. Esto se debe a que a pesar de que intervienen los mismos 4 puntos, se toman ternas diferentes para construir las circunferencias en cada caso.

En conclusión, garantizar que el ordenamiento de los operandos en los cálculos sea independiente del origen de dichos operandos para que el error numérico sea consistente, solo resuelve el problema cuando se puede asegurar que cálculos equivalentes utilicen siempre los mismos subconjuntos de operandos. Esto se aplica sin problemas al primer ejemplo (la determinación del centro de un círculo a partir de 3 puntos), pero como se demostró en el segundo caso, no logra resolver algunos problemas fundamentales para el éxito del algoritmo. Se demostró al comienzo de este capítulo que el test que se pro-

pone en [64] es equivalente al test tradicional de Delaunay cuando se busca determinar la diagonal correcta para un cuadrilátero. Este tipo de equivalencias se pueden tener en cuenta para intentar reemplazar en ciertas situaciones los cálculos que no pueden obtener robustez por el simple reordenamiento de sus operandos, por sus alternativas equivalentes si estas sí lo permiten. En este ejemplo, utilizar el test Delaunay clásico no resuelve completamente el problema, pero sí lo simplifica considerando el uso complementario de la segunda estrategia para resolver el problema del error numérico. Por esto, la implementación desarrollada en este trabajo lo utiliza en lugar de la propuesta original que presentaba el método en [44]. La segunda estrategia se describe a continuación.

5.2.2. Ajuste dinámico de una tolerancia numérica para comparaciones entre resultados reales

Para los cálculos en los que no se puede controlar con total exactitud el error numérico, se utiliza una tolerancia dinámica (que irá variando durante la ejecución) en cada nodo. En particular, para testear si un nodo está dentro del círculo que define un triángulo. El test esencial es entonces el que determina si un punto está o no dentro de una circunferencia. La forma más directa de hacerlo es comparando la distancia del punto al centro de la circunferencia con el radio. El signo de la diferencia entre ambos determina el resultado del test. Cuando esta diferencia sea mayor a un umbral dado, se puede asumir que el error numérico no afectará el resultado si se testea una propiedad equivalente en otra iteración (como al cambiar la arista base en el ejemplo de la figura 5.7). Pero si la diferencia es menor al umbral, el algoritmo propuesto modifica el punto (perturbando su posición) y reduce el umbral para garantizar que futuros tests arrojarán resultados equivalentes y que las posibles futuras perturbaciones no alterarán dichos resultados [11]. Para ello, el umbral se define como 2ϵ, donde ϵ es una tolerancia asociada al punto en cuestión. Cuando la diferencia es menor a ϵ el punto se perturba para lograr que la diferencia sea exactamente ϵ, y el valor de ϵ se disminuye a la mitad, para evitar que futuras perturbaciones lleven el punto al otro lado de la circunferencia (modifiquen el resultado ya utilizado si se repite el test). Cuando la diferencia es mayor que ϵ, pero menor que 2ϵ, no hace falta perturbar la posición del punto, pero sí reducir de igual forma la tolerancia (por el mismo motivo).

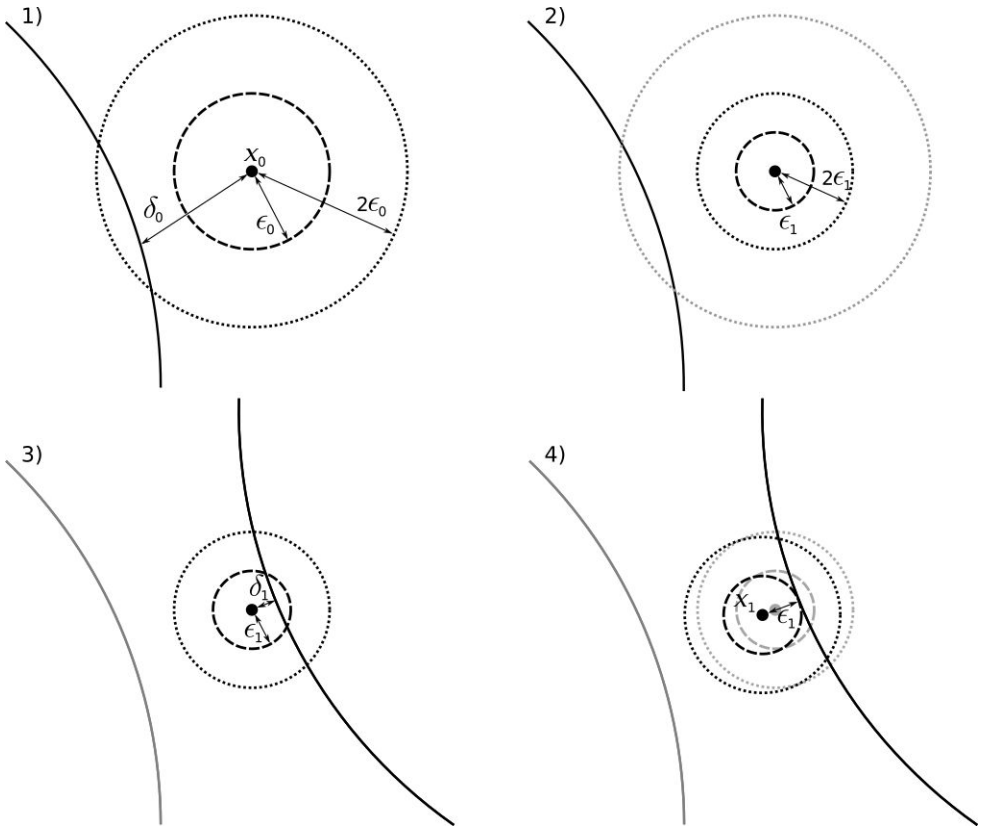

Figura 5.8: Ejemplo de evolución de una tolerancia asociada a un nodo, y de la posición del mismo.

En el ejemplo de la figura 5.8 el punto tiene una posición (x_0) y tolerancia (ϵ_0) iniciales determinadas. En la primer subfigura, un test determina que el punto está fuera del círculo, con una diferencia mayor a la tolerancia ($\delta_0 > \epsilon_0$), por lo cual no se perturba el punto, pero siendo esta diferencia menor al doble de la tolerancia ($\delta_0 < 2\epsilon_0$) se reduce dicha tolerancia. Al reducir la tolerancia, ahora el círculo se encuentra a una distancia mayor al doble de la nueva tolerancia ($\epsilon_1 = \frac{\epsilon_0}{2}$), de forma que si otro test debe perturbar el punto (como pasa en el segundo paso, dado que $\delta_1 < \epsilon_1$), el punto perturbado no puede nunca moverse al interior del círculo testeado inicialmente. Esto se debe a que la perturbación mueve el punto una distancia menor a la tolerancia actual, que ha sido reducida en el primer test para ser menor que la distancia al primer círculo ($\epsilon_1 < \delta_0$). En el ejemplo, el punto es desplazado y la tolerancia deberá reducirse nuevamente para garantizar que la distancia desde la nueva posición del punto a cualquiera de los círculos siga siendo menor que dicha tolerancia actualizada ($\epsilon_2 = \frac{\epsilon_1}{2}$). Es decir, dado que cada vez que se perturba el punto, lo hace con un movimiento menor a la tolerancia, y esta se reduce

a la mitad en cada paso, la suma de todas las perturbaciones (acotada por arriba por la serie geométrica $\sum_{i=0}^{N} (\frac{1}{2})^i \epsilon_0$) nunca puede ser mayor a dos veces la tolerancia inicial ($2\epsilon_0$).

5.3. Estructuras de datos

En la primera sección de este capítulo se describieron todas las consideraciones geométricas necesarias para resolver completamente el problema de la generación de una triangulación (o tetraedrización) Delaunay de un conjunto de puntos, aún en los casos donde el criterio Delaunay presenta ambigüedades. Al implementar esa solución en una PC, aparecen dos tópicos específicos e importantes. El primero de ellos es el tratamiento del error numérico introducido por la aritmética de punto flotante y precisión finita, el cual fue analizado en detalle en la sección anterior, y se propuso en consecuencia una solución general y cerrada para el mismo. El segundo aspecto de interés a analizar al momento de implementar el algoritmo, que merece ser discutido en detalle dado su alto impacto en el rendimiento final de la solución (en términos de tiempo de ejecución y consumo de memoria), es la selección de las estructuras de datos que dan soporte al algoritmo. Se deben seleccionar en primer lugar las estructuras básicas utilizadas para guardar la información de entrada (puntos) y salida (conectividades) del algoritmo, junto con auxiliares y temporales asociados a estos datos, necesarios para los cálculos que el algoritmo requiere. En segundo lugar, se debe considerar la utilización de estructuras de ordenamiento espacial adicionales que permitan acelerar la aplicación del test propuesto para determinar el punto ganador entre los posibles candidatos para una arista base, y la determinación de las aristas bases y puntos candidatos para un trabajo (subproblema) dado. En las siguientes subsecciones se analizan las posibles opciones para cada problema, comparando sus ventajas y desventajas para determinar su conveniencia.

5.3.1. Representación de nodos y conectividades

En la implementación desarrollada en este trabajo, toda la información de una malla, tanto la necesaria para definir la misma, como la información auxiliar necesaria para generarla, se encapsula en una clase denominada MallaCH. La mínima información necesaria para definir una malla incluye las posiciones de sus nodos y las conectividades de sus elementos. Se debe determinar cómo representar la información de un único nodo y de un úni-

co elemento, y qué contenedores utilizar para agrupar y gestionar tanto el conjunto total de nodos como el conjunto total de elementos.

Para almacenar la información de un nodo se puede utilizar una clase cuyos atributos contienen simplemente sus coordenadas como una terna de valores de punto flotante. Considerando además los problemas de precisión numérica descritos en la sección anterior, cada nodo debe incluir también un valor de ϵ para utilizar en las comparaciones. El conjunto de nodos es conocido al inicio, y fijo a lo largo de todo el proceso, de forma que no se realizarán sobre el contenedor de nodos operaciones de inserción o eliminación. Por esta razón, el contenedor utilizado para almacenar los nodos será de tipo vector, ya que es la estructura más compacta y eficiente para el acceso aleatorio.

Para representar un elemento, cuando todos los elementos son de un mismo tipo como en este caso, alcanza con identificar el conjunto de nodos que lo forman, ordenados con algún criterio que permita determinar su orientación. Por ejemplo, en 2D se podría decir que un triángulo correctamente orientado tiene tres nodos ordenados de forma tal que al recorrerlos se rota en sentido anti-horario (alrededor de su centroide), mientras que uno invertido tiene sus tres nodos en sentido horario. Este es un mecanismo intuitivo y muy fácil de implementar en un algoritmo (ya que el signo de la coordenada Z del vector producto vectorial entre dos de sus lados determinaría directamente si el triángulo está o no invertido). La clase que representa un elemento triangular debe entonces referenciar en un orden particular a tres nodos de la malla. La forma más directa de referenciar un nodo sería almacenando un puntero al mismo, pero esto tiene dos desventajas importantes. En primer lugar, los nodos no podrían cambiar de posición en memoria ya que las referencias se invalidarían. En segundo lugar, depurar este tipo de estructuras se torna excesiva e innecesariamente complicado, ya que dada una dirección de memoria no es directo determinar si la misma es válida o no, y además puede variar de ejecución en ejecución dificultando la reproducción de un error o resultado. El primer problema no será particularmente importante en el algoritmo que se propone en este capítulo, ya que el conjunto de nodos no cambia a lo largo del mismo. El segundo problema sí ha sido considerado, razón por la cual para referenciar un nodo se utiliza un índice (su posición en el vector) en lugar de un puntero. Este índice servirá para acceder al vector de nodos, permitiendo determinar rápidamente su validez y reproducir los resultados y estados de forma exacta en diferentes ejecuciones, aún en diferentes PCs o sistemas operativos. Hay dos desventajas al utilizar un índice en lugar de un puntero. La primera es que mediante un puntero se puede acceder directamente a los datos del nodo a partir del propio puntero, mientras que un índice por sí mismo no es útil, sino que para obtener los datos de

un nodo se requiere además acceso al contenedor sobre el cual opera dicho índice. Esto hace que cada función que opera sobre un elemento deba recibir directa o indirectamente acceso al contenedor global de nodos. Dado que la gran mayoría de las funciones que operan sobre un elemento se encapsulan como métodos en la misma clase MallaCH que contendrá los contenedores de nodos y elementos, esto no constituye una desventaja real para este caso. Por otro lado, el acceso mediante un índice (es decir, indirecto) implica un nivel de desreferencia adicional. Esto podría manifestarse en los tiempos de ejecución si el acceso a nodos fuera frecuente y su costo fuera comparable al de las demás operaciones que se realizan por cada nodo accedido, pero este no es el escenario del algoritmo propuesto.

Se debe remarcar que conceptualmente el triángulo determinado por tres nodos A, B, C, es exactamente el mismo triángulo que B, C, A y que C, A, B, ya que la única relación de orden de interés geométrico es la que se describió antes para determinar su orientación (horaria o anti-horaria), y este criterio admite estas equivalencias. Para hacer más fácil y eficiente la implementación de comparaciones y otras operaciones similares, al almacenar los nodos de un elemento, todas las clases que representan elementos en este trabajo eligen de entre las versiones equivalentes posibles, la que tiene por primer nodo al de menor índice, y cuando hay más de una opción que cumple este criterio (lo que ocurre con tetraedros), la que tiene un segundo nodo con menor índice.

Todas las clases que representan mallas a lo largo de este trabajo (como la clase MallaCH mencionada en esta sección), comparten funcionalidades básicas mediante herencia a partir de una clase más general MallaBase. Al implementar la clase MallaBase, se han utilizado técnicas de programación genérica (en C++, templates de clases y funciones) para que los algoritmos que operan sobre dicha clase accedan a los contenedores de nodos y elementos (y opcionalmente un tercer contenedor para la frontera) mediante interfaces genéricas, es decir, independientes del tipo de contenedor utilizado (más detalles en el apéndice A.1). Detrás de estas interfaces, pueden intercambiarse distintos tipos de contenedores de forma transparente. Esto permite evaluar fácilmente el rendimiento para diferentes tipos de contenedores y/o seleccionar el adecuado para cada tipo de dato contenido y cada algoritmo de mallado en particular (este detalle tendrá mayor relevancia al discutir la paralelización del algoritmo en el capítulo 7). Para el conjunto de nodos de la clase MallaCH, no hay dudas acerca de la conveniencia de utilizar un contenedor de tipo vector, pero es válido analizar qué estructura es conveniente para el contenedor de elementos, ya que sobre el mismo se realizarán inserciones frecuentemente. Como muestra la figura 5.9, las mediciones realizadas

no reflejan grandes diferencias entre el uso de una estructura compacta como un vector frente a una estructura teóricamente más adecuada para la inserción y eliminación de elementos en este caso. Esto se debe a que el tipo de inserción que se hace permite que una estructura basada en un vector amortice los tiempos de realocación (gestión interna de la memoria), y a que las inserciones no son operaciones dominantes en el tiempo de proceso total, ya que por cada elemento insertado se aplica el test del círculo a varios nodos candidatos, siendo estos tests comparativamente mucho más costosos.

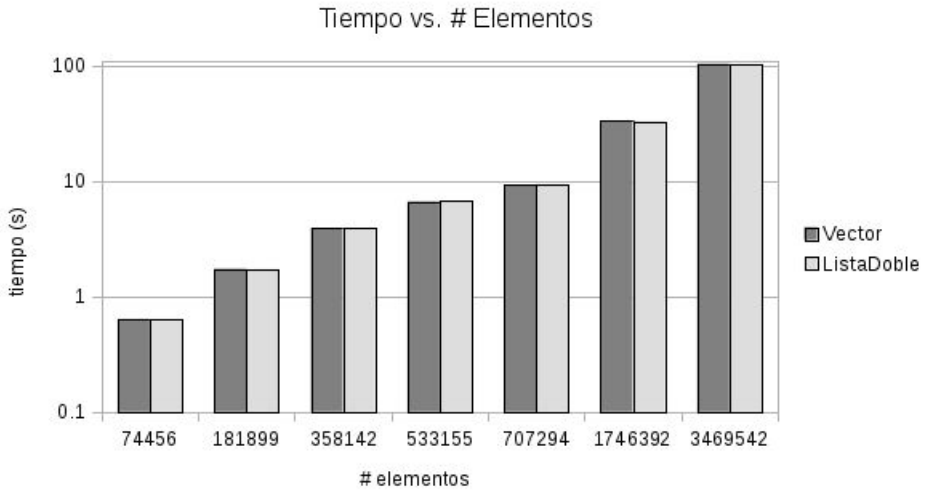

Figura 5.9: Comparación de tiempos de ejecución para el algoritmo de generación utilizando diferentes contenedores para almacenar los elementos generados.

Se podría analizar aquí también el aprovechamiento (o no) de los niveles de memoria cache de la arquitectura de hardware sobre la cual se implementa, ya que esto puede modificar sensiblemente los tiempos de cómputo. Pero se debe notar que la naturaleza básica del algoritmo hace que el resultado no sea en general *cache-friendly*, ya que el conjunto de nodos es impuesto, y se debe asumir por lo tanto que cualquier orden para el mismo es factible. No es una operación trivial reordenarlo de forma que permita optimizar su acceso para este método de mallado, y además habría que optimizar también el orden en que se procesan las aristas base y los nodos candidatos para cada una de ellas en cada trabajo. Estas operaciones en general requerirían más tiempo (a modo de preproceso) del que ahorrarían, considerando además que al paralelizar el algoritmo esta mejora sería aún menos eficaz (la etapa de preproceso debería aplicarse repetidas veces para una estrategia de memoria distribuida como la que se describe en 7.3).

5.3.2. Estructuras auxiliares y de ordenamiento espacial

Hay tres aspectos a considerar al definir las estructuras de ordenamiento espacial y otros auxiliares que requiere MallaCH para aplicar eficientemente el algoritmo descrito: la necesidad de realizar descarte masivo de puntos al buscar el ganador para una cara base, la identificación rápida del conjunto de puntos pertenecientes a un trabajo determinado, y el almacenamiento del frente de avance para que luego de finalizado un trabajo se puedan separar las componentes del mismo para delegarlas a los nuevos subtrabajos.

Descarte masivo de puntos candidatos

En primer lugar, para mantener acotado el orden del tiempo de ejecución del algoritmo de forma que pueda competir con algoritmos alternativos, es necesario utilizar alguna estructura de ordenamiento espacial que permita el descarte masivo de puntos candidatos a la hora de determinar el punto ganador para una dada arista base. Si se probaran todos los puntos por cada arista, el algoritmo sería (al menos) $O(N \times M)$, done N es la cantidad de nodos, y M de elementos. Como se detalló anteriormente, si en determinado momento de la ejecución el algoritmo ha seleccionado un punto candidato para una arista base, todos los puntos fuera de la circunferencia que el punto candidato define con la arista base pueden ser descartados. Por esto, si la estructura de ordenamiento espacial se basa en una partición del dominio de búsqueda en celdas, se puede comenzar por la celda que contenga al punto medio de la arista base (se asume que el punto ganador se encontrará relativamente cerca de la arista base) y avanzar hacia las celdas vecinas siempre que dichas celdas se intersecten con la región de búsqueda factible (es decir, con el círculo actual y el semiplano que define la arista base como positivo). En un caso ideal, las celdas deberían recorrerse por orden de distancia respecto del centro de la circunferencia (o de la arista si aún no hay circunferencia). En [44] se propone la utilización de una grilla regular como estructura de ordenamiento para el conjunto de puntos. De esta forma, se puede determinar rápida y fácilmente si una celda de la grilla podría intersectar al círculo comparando los límites de sus respectivos bounding-boxes, y es también trivial la identificación de celdas vecinas (dado que las celdas de la grilla presentan un ordenamiento trivial, es directo determinar cuando el descarte de una celda asegura el descarte de otra). Además, el orden dado por las vecindades puede corresponderse directamente con el orden según las distancias al punto medio de la arista base si se admite el uso de la métrica de

Manhattan[2] en lugar de la métrica euclidiana para medir dichas distancias. Las mediciones de tiempos realizadas en la práctica determinaron que para obtener el máximo beneficio el tamaño de la grilla debe ser tal que cada celda contenga en promedio $O(1)$ nodos (ver 5.10). A partir de estos resultados se fijó la densidad de la grilla en 5 nodos por celda. El tamaño se calcula con un algoritmo heurístico ad-hoc que busca obtener la densidad promedio deseada, calculando cada una de las 3 dimensiones de la grilla utilizando como referencia la cantidad de nodos totales de la malla, y la relación de aspecto del espacio que dicha grilla debe abarcar.

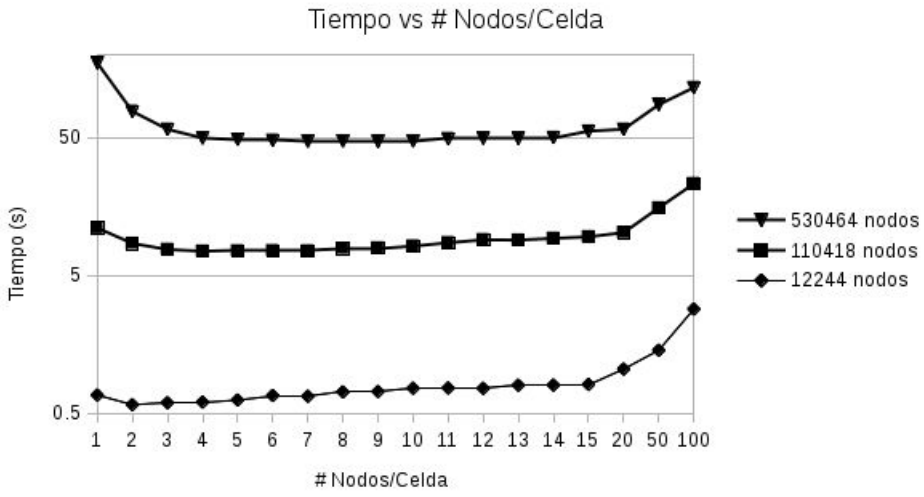

Figura 5.10: Comparación de tiempos de ejecución para el algoritmo de generación utilizando diferentes tamaños de grilla, utilizando tres mallas de tamaños diferentes. Los mejores resultados se obtienen para grillas de entre 1 y 10 nodos por celda.

La figura 5.11 muestra los tiempos de mallado medidos en la implementación final para diferentes tamaños de malla utilizando este mecanismo. Se comprueba que los tiempos crecen aproximadamente como $km^{1.12}$, siendo m la cantidad de elementos generados, para mallas de hasta $1e6$ nodos (que generan hasta $6.6e6$ elementos). Luego de este punto la pendiente de la gráfica aumenta debido a los efectos de la organización jerárquica de la memoria[3].

[2]La métrica de Manhattan (también denominada taxicab, rectilínea o norma L1) es una métrica en la cual la distancia entre dos puntos es la suma de las diferencias (absolutas) de sus coordenadas.

[3]No solo del procesador, sino también a nivel sistema operativo, ya que el análisis con herramientas de profiling muestra que el conteo de instrucciones que efectivamente realiza el procesador no se incrementa en la misma proporción en que lo hacen los tiempos.

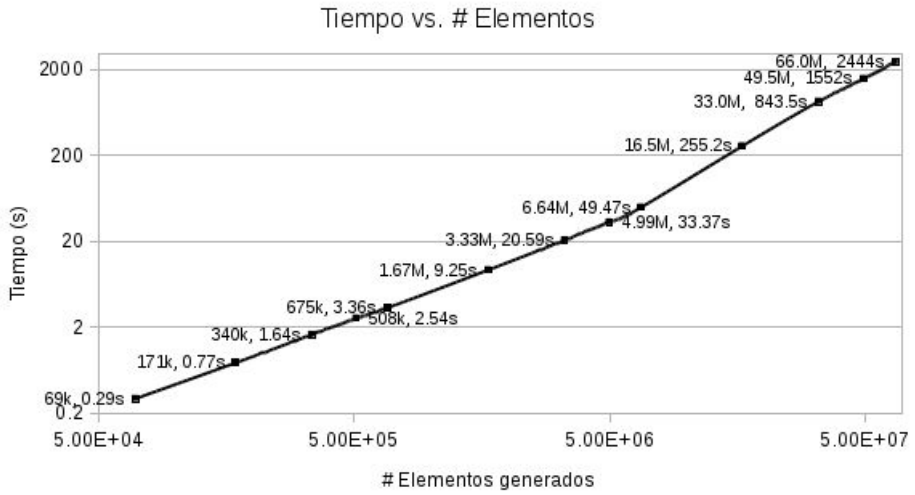

Figura 5.11: Comparación de tiempos de generación para mallas de diferentes tamaños. La velocidad promedio de generación de elementos que el algoritmo alcanza (medida en elementos generados por minuto) varía entre $14e6$ (para las mallas más pequeñas) y $1.6e6$ (para la malla más grande).

Se propuso como alternativa utilizar un octree para contrarrestar las potenciales deficiencias de una grilla regular (muchas celdas vacías si hay zonas sin puntos, o celdas con cantidades de nodos dispares si la malla presenta h variable). Es razonable esperar que cuando más dispares sean los valores de h en la malla resultante, mejor sea el rendimiento del octree y mayor la degradación con el uso de la grilla. Sin embargo, al utilizar estos algoritmos de mallado en métodos de partículas, las variabilidad de h nunca será demasiado alta. Las mediciones de tiempos realizadas con una versión previa del algoritmo[67] (utilizando mallas que no cubren completamente el AABB del conjunto de puntos, y con zonas de diferentes h donde la relación entre los valores máximos y mínimos es de $1 : 8$) verificaron que la grilla regular ofrece un mejor rendimiento en todos los casos (ver figura 5.12). Esto se debe a que todas las operaciones que se realizan sobre la grilla pueden resolverse en tiempo constante, sin importar el tamaño de la misma, mientras que en el caso de un octree (más precisamente un bucketed-octree para evitar que el árbol crezca en profundidad innecesariamente) las operaciones introducen una sobrecarga en el tiempo que depende de la profundidad del árbol, y por ende de la cantidad de puntos que contiene. Además, la determinación de vecindades entre celdas es trivial y directa en una grilla, mientras que en octree (o cualquier tipo de árbol similar) presenta algunas desventajas (por ejemplo, el número de celdas vecinas no es fijo ni está acotado) que resul-

tan más obvias cuanto más desbalanceado se encuentra el árbol (para esta aplicación, cuanto mayor es el rango entre el h mínimo y el h máximo en la malla).

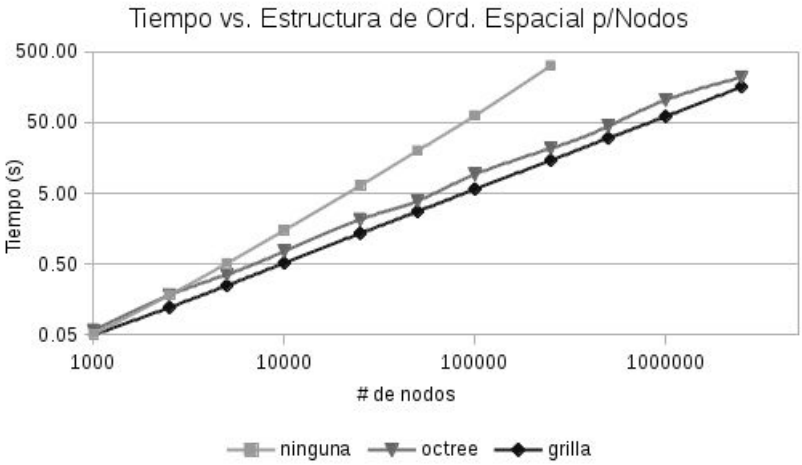

Figura 5.12: Comparación de tiempos de generación para mallas sin utilizar una estructura de ordenamiento espacial auxiliar para el conjunto de nodos, utilizando una grilla regular, y un bucketed-octree.

Además de determinar cual es la mejor estructura de ordenamiento espacial para los puntos, es importante analizar la conveniencia de construir dicha estructura al comienzo del proceso completo y utilizarla de forma compartida en todos los subprocesos frente a la alternativa de reconstruirla para cada trabajo solo con los puntos que utiliza dicho trabajo. La segunda opción agrega un preproceso a cada trabajo, pero a cambio obtiene una estructura de ordenamiento más compacta (en relación al volumen de información contenida, el consumo de memoria en general y el aprovechamiento de la memoria cache). Para el caso de la grilla, mediciones experimentales demostraron claramente la conveniencia de utilizar una única grilla durante todo el proceso de mallado.

Mantenimiento del frente de avance

En segundo lugar, luego de agregar un elemento, creado a partir de una arista base, las demás aristas de dicho elemento podrían ser nuevas aristas base para pasos posteriores o no (cuando el frente de avance se cierra sobre las mismas). Será necesario determinar rápidamente para cada nueva arista cual es el caso. Para ello, se asocia a cada nodo un conjunto de aristas que

almacenará referencias a las aristas del frente de avance que contengan dicho nodo. Así, al insertar un nuevo elemento, se puede buscar cada una de sus aristas en los conjuntos de sus nodos para determinar si ya estaban en el frente de avance (caso en que debe ser eliminada del mismo), o si no estaba y debe agregarse (tanto al conjunto de aristas del nodo como al conjunto de aristas pendientes de procesar). En realidad para determinar si una arista está o no en el frente utilizando estos conjuntos auxiliares por nodo, basta con buscarla en el conjunto de uno solo de sus nodos. Además, dado que las aristas siempre tienen sus nodos ordenados de forma que el primero sea el de menor índice, puede utilizarse solo ése para la búsqueda y para el almacenamiento. Entonces cada nodo contendrá referencias solamente a las aristas en las que sea el de menor índice. De esta forma, al agregar o quitar conceptualmente una arista del frente de avance, el mantenimiento necesario incluye agregar o quitar la arista del conjunto de uno de sus nodos, y la pregunta inicial se responde simplemente recorriendo dicho conjunto, que será en general un conjunto de muy pocos elementos. El tamaño del mismo depende linealmente de la valencia del nodo, y una valencia alta en general se asocia a elementos de baja calidad, por lo que será un caso muy poco frecuente (amortizable), permitiendo considerar esta búsqueda como una operación $O(1)$.

Identificación de nodos propios en cada trabajo

Por último, resta definir cómo identifica un subtrabajo a sus datos de entrada y salida. Es decir, si la malla contiene un conjunto global de puntos compartidos por todos los subtrabajos, cómo determinar al procesar cada subtrabajo cuáles nodos considerar, y qué otra información adicional es necesaria. Para determinar la pertenencia de un nodo a un subtrabajo se utilizaron dos mecanismos, en ciertos aspectos redundantes y en ciertos aspectos complementarios. El primer mecanismo consiste en la asignación de un identificador de trabajo a cada nodo. Cada nodo, en un momento dado solo puede pertenecer a un trabajo pendiente (un nodo que pertenece a un trabajo pasa a pertenecer a uno de los dos subtrabajos que el primero genera luego de finalizado el mismo). Cada trabajo tiene un identificador propio y único (un entero), y solo debe considerar a los nodos que tienen asociado el mismo identificador. Por ejemplo, al avanzar por las celdas de la estructura de ordenamiento espacial para determinar cuales nodos candidatos testear, se verifica la pertenencia al trabajo en cuestión antes de aplicar dicho test en un nodo de una celda, evitando así construir elementos con nodos de otro trabajo. Es de esperar que los nodos de un mismo trabajo se encuentren espacialmente localizados (cercanos), de forma que no sea importante el número

de nodos descartados. Sin embargo, en algunas operaciones será necesario recorrer todos los nodos de un trabajo. En estos casos no resulta eficiente recorrer to dos los nodos de todos los trabajos y seleccionar por id los de uno en particular. Conforme avanza el mallado y los trabajos se tornan más pequeños (serán finalmente los trabajos más numerosos) esta estrategia se torna ineficiente. Para solucionarlo, se utilizó un arreglo auxiliar de índices de nodos que inicialmente contiene índices a todos los nodos. Cada trabajo debe luego de finalizar reordena el arreglo de modo que los índices de nodos de cada uno de los subtrabajos que genera se encuentren contiguos en dicho arreglo. De esta forma, un trabajo solo necesita saber en qué posición de ese vector auxiliar están el primer y el último índice de los nodos que le corresponden, para recorrer, a través de ese nivel adicional de indirección, solo ese subconjunto de nodos. Se reordena un arreglo auxiliar y no el arreglo original de nodos porque es mucho más barato computacionalmente reordenar un arreglo de enteros, en comparación a ordenar un arreglo de objetos más complejos, y porque además (y esto es lo más importante) de esta forma se mantienen válidas las referencias a dichos nodos que guardan otras estructuras (como los elementos).

Asociación entre frentes de avance y trabajos

Además del par de índices que delimitan los nodos con que debe trabajar, cada trabajo debe identificar su propio frente de avance. Luego de resuelto un trabajo, su frente de avance queda dividido en al menos dos subconjuntos disjuntos, que pueden utilizarse como frente inicial para los dos subtrabajos que genera. Para que esto sea posible, cada trabajo mantiene tres listas de aristas(2D)/caras(3D) que en conjunto representan el frente de avance. Una lista contiene las caras por las cuales el trabajo debe avanzar (las atravesadas por el plano divisor), mientras que las otras dos corresponden a las caras que se delegarán a los subtrabajos que genere. Cada vez que se agrega un elemento a la malla, cada una de las caras del mismo se utiliza para actualizar (agregar en o quitar de) una de las tres listas. Se utilizan listas como contenedores para estos conjuntos de caras por varias razones. En estos subconjuntos serán frecuentes las operaciones de inserción y eliminación, y en general no se realizarán en los extremos del conjunto. Además, cuando un subtrabajo comienza, recibe una lista de caras generada por el trabajo que lo precedió y debe dividirla en las tres sublistas mencionadas según la posición de su propio plano divisor. Un elemento de una lista puede moverse a otra simplemente reordenando los enlaces de ese elemento y los que están lógicamente adyacentes al mismo (es decir, en tiempo $O(1)$ y sin requerir

alocar ni liberar memoria). Por último, los nodos guardan referencias a estas caras por lo expresado anteriormente, razón por la cual es importante que se pueda generar alguna referencia que no requiera mantenimiento cuando el contenedor agrega o quita elementos, o aún cuando el elemento se mueve de un contenedor a otro. En el caso de una lista, un puntero a un nodo, arista o triángulo (a diferencia de un puntero o un índice en un vector) sigue siendo válido luego de estas operaciones.

5.3.3. Generalización de las estructuras de ordenamiento y operaciones auxiliares

En la sección 5.3.1 se describió la metodología utilizada para permitir variar fácilmente los tipos de contenedores utilizados para almacenar nodos y elementos en las clases que representan mallas. En resumen, se basa en hacer estos contenedores genéricos en la implementación de la clase malla, e implementar las clases que representan a los contenedores utilizando para todas una misma interfaz, de modo que sean transparentemente intercambiables en tiempo de compilación (polimorfismo estático). Una idea similar se utilizó para poder variar la estructura de ordenamiento espacial aplicada sobre el conjunto de nodos (grilla o árbol). La ventaja de una implementación basada en templates frente a una alternativa más común como lo es el polimorfismo dinámico[4] radica en la posibilidad de resolver la utilización de una u otra en tiempo de compilación, permitiendo así al optimizador del compilador hacer optimizaciones muy valiosas para este tipo de procesos (como por ejemplo el inlining en llamadas a funciones). Para la implementación de la clase que representa al plano divisor, en cambio, se utilizó una estrategia basada en polimorfismo dinámico. Es decir, el plano divisor está representado por una clase que contiene solo un método que indica si un nodo está de un lado o del otro de dicho "plano", o eventualmente justo sobre el plano. El objetivo de esta abstracción es permitir la implementación de otras entidades geométricas diferentes al plano, pero que también permitan particionar el dominio en dos partes disjuntas, y permitir además combinar dinámicamente ambas implementaciones (este segundo objetivo en particular es el que limita la aplicabilidad de la opción basada en templates). La verdadera motivación para estas libertades está relacionada a la utilización de este algoritmo en su versión paralela, y más específicamente para memoria distribuida, por lo que

[4]El *polimorfismo dinámico* se implementa en C++ mediante métodos virtuales y se resuelve en tiempo de ejecución. La técnica utilizada aquí puede catalogarse como *polimorfismo estático*, y se implementa mediante templates. En la literatura se suele utilizar el término *polimorfismo* sin más aclaración para hacer referencia al primero

se describirá mejor en la sección 7.3.3.

5.4. Resumen del algoritmo propuesto

5.5. Estructuras de datos

Se muestran a continuación (de forma simplificada) las estructuras de datos necesarias para dar soporte a los algoritmos descritos en las secciones anteriores.

```
struct Nodo {
   float x,y,z; // coordenadas
   float epsilon; // tolerancia para el test Delaunay
   int id_trabajo; // trabajo al que pertenece
   List<it_cara> caras; // caras del frente que lo contienen
                        // como primer nodo, almacenadas como
                        // iteradores para las listas del
                        // trabajo
};

struct Cara { // cara de la frontera
   int nodos[3]; // indices de sus nodos, ordenados
};

struct Elemento { // tetraedro
   int nodos[4]; // indices de sus nodos, ordenados
};

struct Trabajo { // datos de entrada para un trabajo
   int id_trabajo; // id único del trabajo
   Divisor *divisor; // functor del plano divisor
   List<Cara> frontera; // caras iniciales
   int r0, rN; // rango en el vector ref_nods de la
               // malla que contiene los nodos de este
               // trabajo
};

class MallaCH {
```

```
    Vector<Nodo> nodos; // nodos, dato de entrada
    Vector<int> ref_nodos; // indices de nodos, para reordenar
    List<Elemento> elementos;
    Queue<Trabajo> trabajos;
    Grilla grid; // estructura para ordenamiento espacial
                 // y búsqueda de nodos
public:
    // ...funciones miembro...
};
```

5.6. Algoritmos

El siguiente pseudocódigo describe todos los pasos involucrados en la generación de una tetraedrización a partir de un conjunto de puntos.

```
Sea n un vector de N nodos (único dato de entrada):
// etapa de preparación
 1 -Inicializar la grilla (grid) cargando en ella todos los
    puntos de entrada
 2 -Crear un arreglo de índices (ref_nodos) con enteros de
    1 a N
 3 -Inicializar una lista de caras (L0)
 4 -Obtener una cara inicial (c0), agregarla en L0 y registrar
    el iterador en c0
 5 -Definir el primer trabajo, utilizando c0 y L0, y agregarlo
    a la cola de trabajos pendientes (jobs)
 6 -Asignar a todos los nodos el id del primer trabajo
// etapa de mallado
 7 -Mientras jobs no esté vacía
 8     -Calcular el AABB del conjunto de nodos del trabajo
 9     -Generar un plano divisor a partir de AABB
10     -Dividir las caras del trabajo en tres listas: lb
        (caras atravesadas por el plano), lr y ll (caras
        a derecha e izquierda del plano)
11     -Mientras lb no esté vacía
12         -Tomar una cara de lb (c) y verificar si aún es
            válida buscándola en la lista de su primer nodo
13         -Si la cara aún es válida (está registrada en
            alguno de sus nodos)
```

```
14              -Intentar obtener un nodo ganador para la
                 cara c
15              -Si se obtiene un nodo valido
16                  -Generar el elemento y agregarlo a la
                     malla
17                  -Por cada cara del elemento generado
18                       -Si estaba registrada en su priemer
                          nodo desregistrarla, sino registrarla
19                       -Determinar de qué lado del plano
                          se encuentra y agregarla o
                          eliminarla de la lista que
                          corresponda (lr, ll o lb)
    // generación de subtrabajos
20      -Reordenar ref_nodos de forma que todos los nodos que
         se encuentren a la derecha del plano se ubiquen antes
         que los que se encuentren a la izquierda
21      -Si hay caras en lr, generar un nuevo subtrabajo en
         jobs asignando dicha lista como frente inicial
22      -Si hay caras en ll, generar un nuevo subtrabajo en
         jobs asignando dicha lista como frente inicial
```

El paso 13 de este pseudocódigo es el más complejo y determinante del algoritmo. Se detalla a continuación el funcionamiento del mismo:

```
Tomando como entrada una cara base c, las listas ll, lr y lb
del trabajo actual, y las estructuras ref_nodos y grid de la
malla:
 1 -Determinar cual celda (c0) de grid contiene el centroide
    de la cara base (c)
 2 -Inicializar una cola de celdas por revisar (q) con c0
 3 -Definir el indice de nodo ganador (imin) como -1 (indica
    que no hay aun un nodo factible)
 4 -Definir la circunferencia inicial asociada a imin: con
    radio (rmin) infinito y centro (cen) en el origen
 5 -Mientras q no esté vacía
 6      -Calcular la distancia (d) del centro la celda (pc) al
         centro de la circunferencia (cen)
 8      -Obtener la longitud (l) de una de las diagonales de
         la celda
 7      -Si la celda (o parte de ella) se encuentra del lado
         adecuado del plano que define c (signed_dist(pc,c)>l/2)
```

```
        e intercepta la esfera (dist(pc-cen)<=rmin+l/2)
        entonces
 9         -Encolar en q todas las celdas vecinas aun no
            procesadas
10         -Por cada nodo de la celda (in)
11             -Si es factible (pertenece al trabajo y está del
                lado adecuado del plano de c)
12                 -Si esta dentro de la esfera actual (se
                    compara utilizando el epsilon del nodo)
13                     -Calcular la esfera que pasa por c
                        y por in, y actualizar cen, rmin e
                        imin con los datos del nodo y la
                        nueva esfera
14                 -Corregir el epsilon del nodo si es
                    necesario
15  -El nodo ganador para la cara base c es el indicado en imin
```

Capítulo 6

Triangulación de una nube de puntos respetando una frontera impuesta

6.1. Descripción del problema

En este capítulo se describe un algoritmo para generar una tetraedrización (malla 3D) de un conjunto de puntos en el interior de un dominio delimitado por una malla de superficie de frontera (triangulación en 3D). La malla generada debe respetar la malla de frontera. Es decir, la frontera de la malla de volumen generada debe coincidir exactamente con la frontera impuesta (que será información de entrada y restricción para el algoritmo). El método empleado se basa en el algoritmo presentado en el capítulo anterior, por lo que generará mayormente elementos que cumplan la condición Delaunay. Sin embargo, en general no será posible respetar dicha condición para todos los elementos, ya que la misma podría no ser compatible con la frontera impuesta. Es decir, puede haber un elemento de frontera que no forme parte de la malla Delaunay de dicho conjunto de puntos. La figura 6.1 muestra un ejemplo 2D de este problema.

Para la generación de una triangulación (2D), donde la frontera es simplemente una poli-linea, este problema puede solucionarse fácilmente con ediciones locales en la malla, basadas en la remoción de triángulos de frontera y en el swap de diagonales entre pares de triángulos vecinos. Sin embargo, en el caso 3D no hay una operación equivalente a esta última, y por lo tanto la recuperación de la frontera a partir de una malla del convex-hull es una

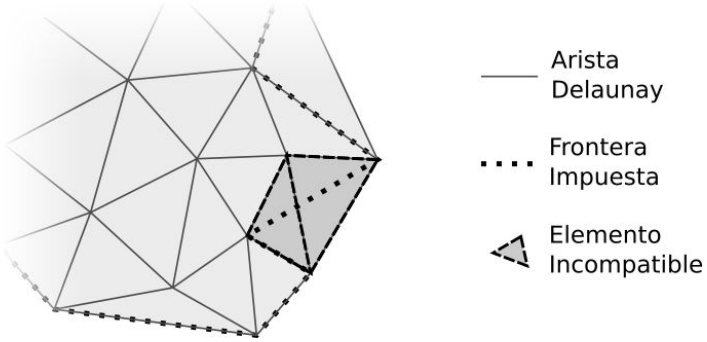

Figura 6.1: Se muestra parte de una malla Delaunay generada para un conjunto de puntos, y una posible frontera impuesta. Los elementos resaltados cumplen la condición Delaunay pero no son compatibles con la frontera impuesta.

tarea difícil y costosa ([34][35][36][37][38]). Por esto puede ser conveniente resolver el problema directamente durante la generación de los elementos. La solución debe entonces evitar colocar elementos incompatibles con la frontera impuesta, rellenando estos volúmenes con conectividades diferentes. Estas configuraciones alternativas no cumplirán la condición Delaunay, y del análisis del capítulo anterior se desprende que esta condición resulta esencial para que el algoritmo pueda colocar un nuevo elemento sin considerar los colocados anteriormente. Esta condición permitía obtener un algoritmo simple, eficiente, y en el cual las operaciones de una iteración no alteraban el resultado de otra (existía globalmente una garantía de unicidad para el resultado final). En esta nueva versión del problema (que impone una frontera no necesariamente Delaunay) se requiere, al colocar un elemento, acceso a información generada como resultado de la colocación de otros elementos previos. Esto genera la necesidad de utilizar nuevas estructuras de datos para consultar dicha información, mayor costo en cada paso por operaciones de mantenimiento de las mismas, mayor dependencia y acoplamiento entre las distintas iteraciones de un trabajo y aún entre diferentes trabajos, y en consecuencia un mayor costo de colocación por elemento. Se describirá y analizará en este capítulo la solución propuesta e implementada para este problema.

6.2. Algoritmo Propuesto

6.2.1. Mecanismo de avance

La solución desarrollada añade (al algoritmo utilizado en el caso sin frontera) operaciones típicas de métodos de avance frontal. Al perder la garantía de unicidad que ofrece Delaunay, el algoritmo debe verificar antes de colocar un nuevo elemento si dicho elemento es compatible con los demás elementos colocados anteriormente. Para ello, basta con verificar si el elemento atraviesa (se intersecta con) el frente de avance actual, dado que los frentes de avance siempre describirán uno o más dominios completamente cerrados, por generarse a partir de un frente inicial cerrado (la frontera impuesta). Cada nuevo tetraedro que se agrega a la malla reemplaza un triángulo del frente (la cara base) por un nuevo conjunto de triángulos conexo cuya frontera coincide con las aristas del triángulo original. Por ello, si en el paso inicial el frente de avance (la malla de frontera) es cerrada, no dejará de serlo en pasos posteriores. Los detalles sobre los algoritmos y estructuras de datos que permiten encontrar estas intersecciones eficientemente serán descritos en la siguiente sección.

El uso de un frente de avance para evitar generar elementos incompatibles con los ya colocados evita además los problemas que generan la ambigüedad del criterio Delaunay y su implementación con aritmética de precisión finita. Al realizar estas verificaciones (intersecciones entre un potencial nuevo elemento y el frente de avance), la colocación de dos elementos con una frontera común ambigua deja de ser independiente. Cualquiera sea la configuración (de entre las posibles) que utilice el primero que se coloque de ellos, el segundo deberá respetarla. En consecuencia ya no es necesario mantener una tolerancia por nodo, ni considerarla al realizar el test de la esfera, ya que los cambios introducidos para respetar fronteras no Delaunay sirven también como solución alternativa (aunque más costosa) para este problema.

Cuando el elemento Delaunay para una cara base y un conjunto de nodos dados no es compatible con la frontera impuesta, el algoritmo debe ser igualmente capaz de generar un elemento diferente y compatible para dicha cara base. La solución que se propuso consiste en excluir de la lista de nodos candidatos para formar un elemento con una dada cara base los nodos que generen elementos incompatibles con el frente de avance (nodos conflictivos). La implementación consiste en recorrer y testear los nodos candidatos en el mismo orden que en el caso sin frontera, pero agregando al test del círculo el test de intersecciones. Entonces, al testear un nodo conflictivo, el test falla,

y el algoritmo continúa de la misma forma en que lo haría si el test hubiese fallado en realidad por estar el nodo fuera de la esfera. Es decir, ignorándolo (como si el nodo no existiera, ver Fig 6.2). Al ignorar un nodo y no respetar así el criterio original, se debe analizar si se genera una nueva posibilidad de error. Si según el criterio combinado (Delaunay e intersecciones), se pudiese generar un nuevo elemento que deje en su interior al nodo ignorado, este nodo nunca podría formar parte de un elemento compatible con los ya generados, y quedaría erróneamente excluido de la malla final resultante. Pero no es necesario agregar una verificación adicional para evitar este problema. Cualquier otro nodo que pueda generar un elemento que contenga al primer nodo, incluirá en su interior a toda la superficie del triángulo que este primer nodo generaría. Si el primero genera una intersección, quiere decir que alguna arista de la frontera impuesta "ingresa" al interior de dicho triángulo. Por esto (sumado a que el frente es a su vez una malla cerrada) el segundo elemento también será intersectado por alguna arista de la frontera, y por ende descartado (ver Figura 6.3).

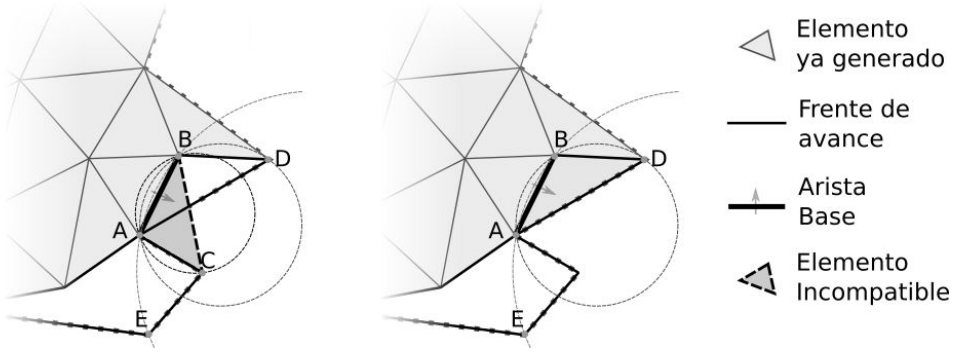

Figura 6.2: Al avanzar tomando AB como arista base, el criterio Delaunay determina que C es el nodo con el cual se debe construir el siguiente elemento. Sin embargo, dado que el elemento ABC intersectaría al frente de avance, el nodo C se se elimina virtualmente del conjunto de nodos factibles. El nodo D pasa a ser el nuevo nodo elegido, según la aplicación del criterio Delaunay sobre los nodos restantes.

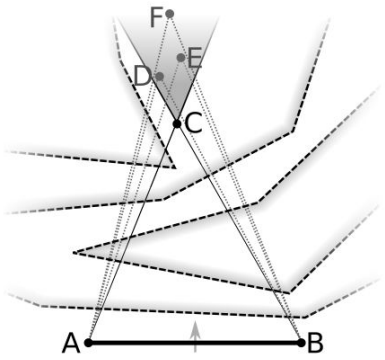

Figura 6.3: El área sombreada corresponde al área donde debe encontrarse un nodo para generar a partir de la arista base AB un elemento que deje en su interior al nodo C (por ejemplo, los elemento ABD, ABE y ABF). Las lineas de trazos representan 4 posibles fronteras/frentes que invalidarían al elemento ABC. Cualquiera de ellas genera también intersecciones con los elementos ABD, ABE y ABF.

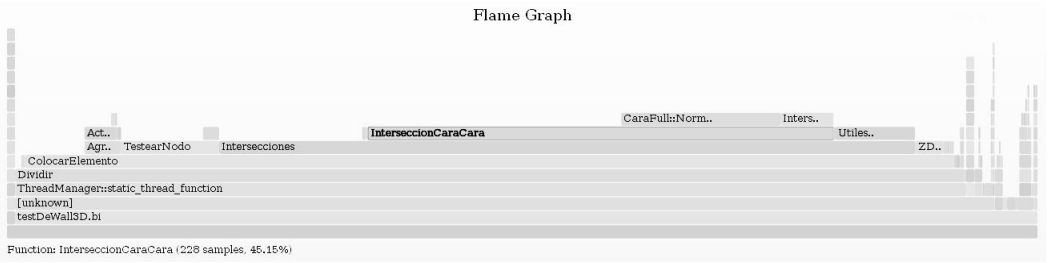

Figura 6.4: Información generada por la herramienta *perf* y presentada mediante *flame-graphs*[68]. El gráfico muestra las funciones invocadas durante un proceso mallado. Las funciones se extienden en el eje X de forma proporcional a su tiempo de ejecución. El eje Y puede interpretarse como un callstack. Se resalta la rutina para la detección de intersecciones entre caras, que consume el 45 % del tiempo total de mallado.

Como se muestra en la figura 6.4, en un proceso de mallado completo, el algoritmo de testeo de intersecciones entre caras es uno de los que más tiempo de ejecución total consume (debido tanto a su complejidad, como al gran número de veces en que se invoca). El algoritmo implementado es un algoritmo ad-hoc de base geométrica. Es decir, se basa en criterios geométricos (interpretaciones de distancias, proyecciones, perpendicularidades, etc), y no puramente algebraicos (como sería el formular directamente el sistema de ecuaciones y reducir el problema a la inversión de una matriz), para tratar de descartar la posible presencia de intersecciones en la mayor cantidad de casos posibles sin requerir la totalidad de los cálculos. Por ejemplo, se puede verificar antes de intentar encontrar el segmento de corte que sus planos no sean paralelos, que cada triángulo tenga puntos a un lado y otro del plano

del otro triángulo, que alguna arista de un triángulo intersecte al otro, etc. Estas verificaciones pueden realizarse con el solo fin de descartar tempranamente algunas intersecciones, o como pasos intermedios para la resolución final del sistema. Se utilizaron herramientas de profiling (gprof y gcov) para determinar el ordenamiento optimo de estos criterios de forma de favorecer el descarte temprano.

Para mejorar el rendimiento del algoritmo y reducir el impacto de los cálculos para la detección de intersecciones, se propuso realizar en dos etapas la selección del nodo ganador para una cara base. En la primera, se busca un nodo ganador sin considerar las restricciones. Esto es, sin calcular las intersecciones con el frente de avance, de igual forma que en el algoritmo de tetraedrización sin frontera impuesta. Luego, una vez determinado el nodo ganador se verifican las intersecciones para el elemento que generaría. Si se detectan intersecciones para este elemento, se repite la búsqueda completa por segunda vez, pero realizando en esta segunda iteración el cálculo de intersecciones y considerando sus resultados para el descarte de nodos durante la búsqueda. Dado que la relación entre la cantidad de elementos que cumplen la condición Delaunay con respecto a los que no es muy baja, es de esperar que en la mayoría de los intentos la segunda etapa no sea necesaria. Es decir, serán comparativamente muy pocos los casos en que el elemento resultante de la búsqueda sin considerar intersecciones genere efectivamente intersecciones con el frente de avance. Más aún, esta optimización reduce la cantidad de nodos testeados, ya que la flexibilización de las restricciones permite encontrar más rápidamente un primer nodo candidato para realizar descarte masivo a partir de su esfera. Además, aún en el caso en que el nodo seleccionado en la primer pasada del algoritmo se descarte como nodo válido luego del test de intersecciones, su esfera puede utilizarse para aproximar a priori el tamaño de la esfera del nodo ganador para la segunda pasada, y limitar con esta información el área de búsqueda. La efectividad de esta optimización se verificó en la práctica. La figura 6.5 presenta una comparativa de tiempos para ejemplos concretos de diferentes tamaños, mostrando reducciones del tiempo requerido de alrededor del 70 % en todos los casos. Por último, cabe mencionar que cada elemento agregado se marca con un flag que indica si fue necesaria o no la segunda etapa de búsqueda. Esto permite consultar rápidamente si corresponde o no a un elemento Delaunay (información que será útil posteriormente, en los pasos que se describen en la sección 6.2.2).

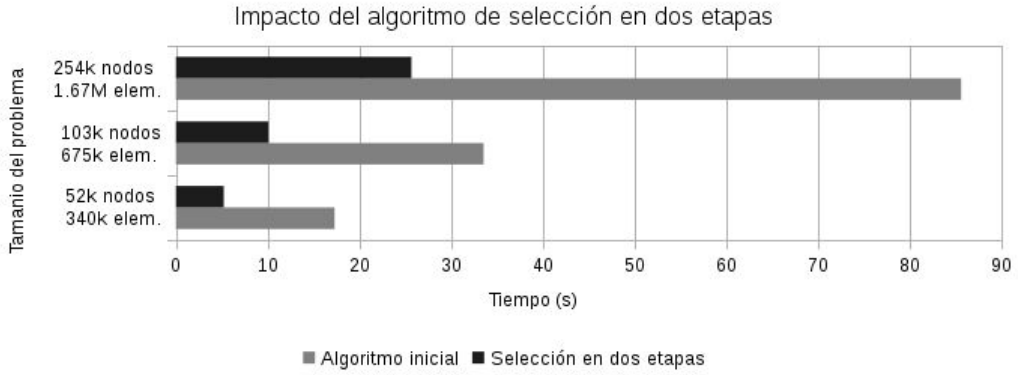

Figura 6.5: El algoritmo inicial realiza el test completo (criterio Delaunay+intersecciones) para cada nodo candidato. La versión de dos etapas intenta realizar solo los tests de intersecciones en el nodo ganador.

Finalmente, cabe destacar que esta variante del algoritmo (basada en la detección de intersecciones) permite reducir también (en la mayoría de los casos evitar completamente) la generación de caps y/o slivers. En configuraciones de puntos 3D ubicados regularmente, el criterio Delaunay, en combinación con la precisión limitada de las operaciones de punto flotante, genera usualmente una gran cantidad de slivers en el resultado. Estas configuraciones de puntos son muy frecuentes en ciertas aplicaciones, como por ejemplo al iniciar una simulación mediante un método basado en partículas móviles (usualmente se parte de una grilla regular de puntos). El tratamiento especial del error numérico presentado en la sección 5.2 del capítulo anterior permite simplemente aplicar con éxito el algoritmo de tetraedrización, considerando éxito a la finalización sin errores del mismo, y la generación de una malla topológicamente correcta, aunque no evita que el resultado contenga un alto número de estos elementos con muy baja calidad. Sin embargo, algunas operaciones geométricas útiles para detectar estos elementos están directa o indirectamente incluidas en los test de intersecciones.

Utilizar umbrales amplios en ciertos cálculos implementados para determinar la existencia de intersecciones entre un elemento y el frente de avance, permite detectar como intersección la potencial colocación de un sliver o cap. Por ejemplo: un sliver tiene sus 4 nodos *casi* cocirculares; en un caso extremo, donde los 4 nodos son efectivamente cocirculares, dos de sus aristas se intersectarán en su interior; el cálculo de intersecciones detectará este caso como una intersección entre el nuevo elemento y el frente, ya que una de estas dos nuevas aristas será parte de la cara base, y por ende del frente actual. Si la tolerancia utilizada dentro del cálculo de intersecciones entre aristas

para determinar si dos aristas se encuentran en un mismo plano se relaja, entonces el mecanismo detectará una intersección aún cuando los 4 nodos no sean exactamente cocirculares. De esta forma, una pequeña modificación mejora notablemente la calidad de la malla (ver figura 6.6), seleccionando en las situaciones de ambigüedad una configuración que evite la generación de elementos de baja calidad. Esta modificación consiste entonces en el simple cambio del valor un umbral para una comparación que forma parte de la detección de intersecciones, por lo que no genera complejidad adicional en el algoritmo, ni tiene asociado directamente un mayor costo computacional. Sin embargo, como se comentará en la siguiente sección al explicar en detalle el mecanismo de retroceso, esta variante puede degradar indirectamente la velocidad del algoritmo por aumentar la probabilidad de obtener frentes irresolubles.

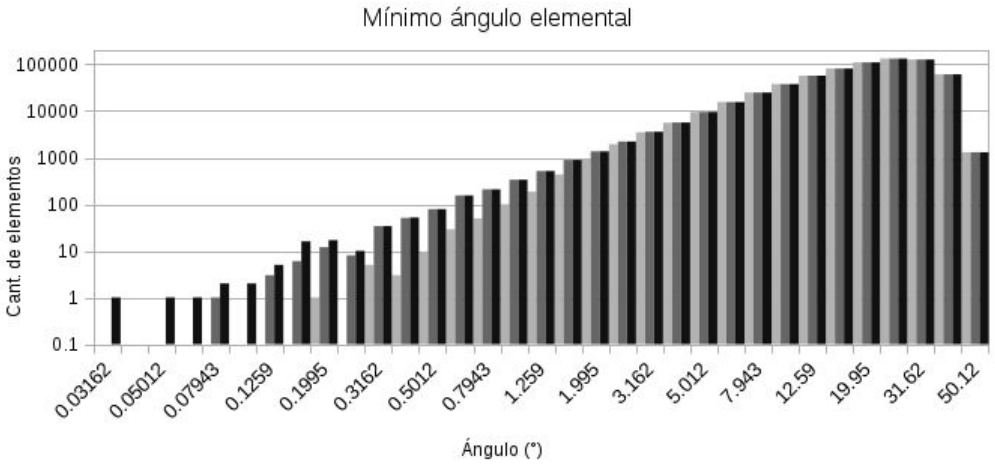

Figura 6.6: 3 histogramas que muestran los ángulos mínimos de los elementos de 3 mallas generadas a partir de una misma entrada, variando una de las tolerancias utilizadas en los tests de intersecciones, para evitar la generación de slivers.

6.2.2. Mecanismo de retroceso

La generación de elementos no conformes con el criterio Delaunay puede conducir a configuraciones irresolubles. Es decir, estados en los que el frente de avance describe un dominio que no puede tetraedrizarse sin agregar nuevos nodos, o generar elementos de muy baja calidad (slivers y/o caps). Dado que no existe un mecanismo computacionalmente aceptable (en términos de tiempo de ejecución) para predecir estas situaciones y evitar así colocar elementos que las generen, el algoritmo debe prever algún mecanismo de retro-

ceso o *rollback*. Es decir, se debe permitir deshacer algunos pasos, eliminando elementos ya colocados y volviendo así a configuraciones previas del frente, a partir de las cuales deberá proceder de diferente manera para evitar entrar en un ciclo infinito. Aquí se deben establecer criterios para identificar las situaciones irresolubles, para determinar cuántos y cuáles elementos deshacer, y para asegurar de alguna manera que al reiniciar el avance del algoritmo no se volverán a generar nuevamente los mismos elementos.

Esta situación (frentes de avance que definen dominios irresolubles) se presenta generalmente cerca de la frontera impuesta cuando esta no cumple la condición Delaunay, o cuando presenta configuraciones ambiguas para el método (por ejemplo, cuando hay más de 4 puntos cocirculares). El segundo caso se debe a que esta versión del algoritmo evita, siempre que sea posible, la colocación de slivers. Cada vez que el algoritmo evita la colocación de un sliver o cap, en realidad está trasladando el problema a un paso posterior, potencialmente hasta llegar a la frontera, donde la restricción impuesta por la malla de frontera puede generar una situación irresoluble.

Es simple detectar el problema una vez que ha ocurrido. Se está en presencia de esta situación si para una determinada cara base no se puede encontrar un nodo candidato que no genere intersecciones (en el sentido amplio: intersecciones o elementos degenerados). En todos los casos en los que el problema se manifestó y fue analizado particularmente de forma manual, fue relativamente sencillo observar que existían soluciones que solo requerían un remallado local. Es decir, en la mayoría de las situaciones, con solo deshacer unos pocos elementos cercanos y reiniciar el algoritmo evitando regenerarlos nuevamente, el problema se resuelve rápidamente. Sin embargo, en cada situación, las potenciales reconfiguraciones en las conectividades que el algoritmo podría intentar realizar, en combinación con las posibles elecciones respecto a cuáles elementos deshacer antes de reintentar el avance, hacen que la cantidad de posibilidades sea demasiado elevada como para realizar una búsqueda exhaustiva, aún para trabajos con un reducido número de nodos. Por ello, la velocidad con que el algoritmo soluciona estos problemas está fuertemente condicionada por el criterio que utilice el mismo para determinar qué elemento debe deshacer en una situación dada.

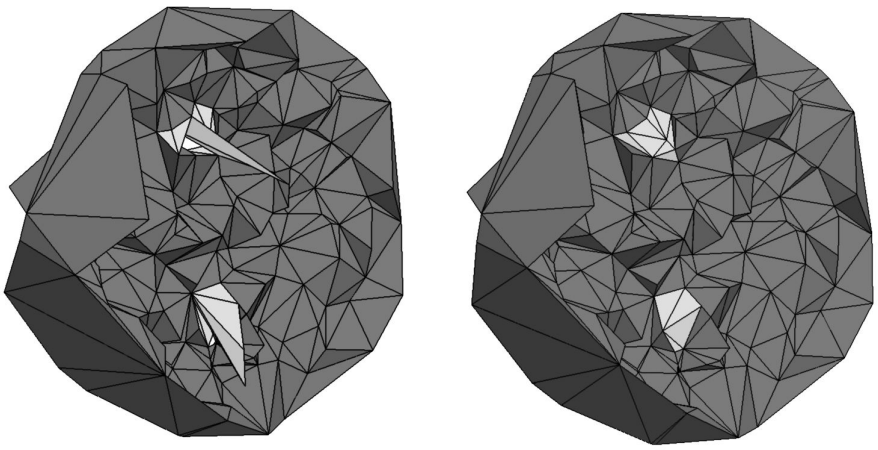

Figura 6.7: Ejemplo de aplicación del mecanismo de retroceso propuesto. Izquierda: Máximo avance posible sin realizar retrocesos; se resaltan los elementos generados que no cumplen la condición Delaunay. Derecha: Resolución final del trabajo, luego de eliminar los elementos no Delaunay del primer caso y continuar.

La solución propuesta se basa en dos premisas. En primer lugar, cuando ya no se pueden generar elementos para una cara del frente de avance, antes de proceder a deshacer algún elemento ya colocado para evitar el problema, se procede a avanzar cuanto sea posible utilizando como caras base las demás caras del frente. De esta forma, cuando el algoritmo llega al punto en que ninguna cara del frente le permite avanzar, el dominio que resta mallar es el mínimo posible, y todas las caras del frente están asociadas a elementos problemáticos. Se dirá que un elemento es problemático si no permite la generación de otro elemento (Delaunay, a partir de una cara del frente) debido a que lo intersecta. Al deshacer un elemento, solo se podrá reiniciar el avance si era un elemento problemático. Retomando entonces la premisa de que estos problemas pueden resolverse localmente en la gran mayoría de los casos, esta estrategia contribuye a la localización (acotación) del problema, sin complicar su resolución. A continuación, se debe seleccionar un criterio para determinar cuál o cuáles elementos quitar. De entre los posibles criterios que se implementaron, el criterio que finalmente permitió obtener mejores resultados (en términos de tiempos de ejecución) se basa en determinar aleatoriamente cual elemento eliminar. Es decir, eliminando un elemento al azar de entre los que efectivamente podrían eliminarse, produce mejores resultados que otras selecciones determinísticas como por ejemplo eliminar el elemento con más elementos vecinos, el que más intersecciones genera, etc. Esto se debe a que no se ha logrado identificar un criterio claro que resuelva el problema rápidamente en todas las situaciones analizadas. Para cada criterio

propuesto se puede generar fácilmente como contra-ejemplo una configuración en la que el criterio demore innecesariamente la resolución del mismo. Sin embargo, para favorecer la generación de elementos Delaunay, al encontrar un frente irresoluble, se eliminan primero los elementos no Delaunay, y luego, si el frente continúa siendo irresoluble se procede a la eliminación de los restantes en orden aleatorio. La figura 6.7 muestra un ejemplo de situación conflictiva y su resolución, generada a partir de un conjunto aleatorio de puntos en una esfera. En este caso, debido a que se utiliza el convex-hull del conjunto de puntos como frontera impuesta, las situaciones conflictivas se producen solo en el interior (debido a las restricciones agregadas en este algoritmo para evitar formar caps y slivers).

Finalmente, además de proveer un mecanismo para la eliminación de elementos, se debe proveer también un mecanismo para evitar regenerar nuevamente una configuración previamente detectada como irresoluble. Para ello, se propone almacenar un historial de frentes de avances irresolubles, y verificar antes de colocar un nuevo elemento, que la colocación del mismo no vuelva a generar un frente anteriormente marcado como irresoluble. Se guarda un frente en el historial solo cuando ya no se puede seguir avanzando por ninguna de sus caras. De esta forma, los frentes almacenados constan de muy pocos elementos. El algoritmo propuesto para la resolución de un trabajo completo tiene tres modos de operaciones. En el primer modo (el inicial), el algoritmo avanza como se describió en la sección anterior, sin verificar la validez de cada frente de avance que genera. Esto es, sin utilizar el historial, y por ende, sin que este problema produzca una sobrecarga en el costo de cada paso. Cuando un trabajo llega por primera vez a la situación en que ya no puede avanzar por ninguna de las caras del frente, se pasa a un segundo modo en el cual cada intento de avance es verificado contra el historial de frentes irresolubles. Es decir, a los tests que se aplican a un potencial nuevo elemento, se añade uno que verifica si su colocación generaría o no un frente ya conocido y marcado como irresoluble. Si luego de un determinado número de reintentos (es decir, de deshacer y reiniciar el avance), el algoritmo no logra cerrar el frente para finalizar el trabajo, se pasa como último recurso a un tercer modo en el que se reajustan las tolerancias utilizadas en los cálculos de intersecciones, para permitir la generación de elementos de menor calidad. De esta forma el algoritmo logrará finalizar el trabajo sin errores, pero con el costo de degradar el resultado. El umbral que determinan el paso del segundo al tercer modo es ajustable, pero dado que las pruebas confirman que estos problemas en general se resuelven localmente y con muy pocos reintentos, no varía sensiblemente el resultado al incrementar el umbral y obligar al algoritmo a intentar eliminar más capas de elementos ya colocados antes de

pasar al tercer modo, aunque sí aumenta notablemente en ese caso el tiempo de ejecución.

6.3. Estructuras de Datos

6.3.1. Detección de intersecciones

En esta nueva versión del algoritmo se tiene entonces un frente de avance inicial que será almacenado y referenciado por los nodos de la misma forma que se describió en el capítulo anterior. Es decir, cada trabajo toma por entrada un frente de avance, el cual consiste en una lista de caras (elementos triangulares). Al comenzar el trabajo, la lista de caras del frente se divide en tres sublistas, una con las caras que son atravesadas por el plano divisor, y otras dos con las caras que quedan a cada lado del mismo. Luego, a medida que se van colocando elementos, las caras base ya procesadas son eliminadas de la primera sublista, y las nuevas caras generadas son repartidas entre las tres sublistas según corresponda. Cuando la lista de caras que son atravesadas por el plano divisor se vacía, el trabajo se da por finalizado y cada una de las dos restantes sublistas pasan a constituir los nuevos frentes de avance iniciales para dos nuevos subtrabajos. Por el mismo motivo que se detalló en el capítulo anterior, estas listas son efectivamente listas enlazadas, y las referencias que guardarán otras estructuras de datos de la malla serán pseudo-punteros (similares a los iteradores de la biblioteca estándar) que apuntan a las celdas de las listas.

Al testear la factibilidad de un potencial nuevo elemento, se deben generar todas las caras que el mismo agregaría al frente y buscar las intersecciones entre estas y las que efectivamente contiene el frente completo (es decir, las tres sublistas) en su estado actual. Para ello se utiliza una estructura basada en un ADT (ver sección 4.2.2) para ordenamiento espacial, que almacena elementos que representan los AABB de las caras del frente. Este ADT permite realizar descarte masivo al momento de identificar las potenciales intersecciones. Entonces, esta estructura auxiliar guarda sus propias referencias al mismo conjunto de caras, y agrega así un mantenimiento extra en cada operación que modifique el frente. Dado que la búsqueda de una cara en particular en el ADT es una operación que requiere $O(logC)$ pasos (con C = cantidad de caras en el frente), no se almacena ninguna otra información adicional para optimizarla, sino que por cada cara que se agrega o elimina del frente, se realiza una consulta al árbol, agregando entonces una sobrecarga de $O(4logC) = O(logC)$ al mantenimiento requerido para cada elemento inser-

tado, y una sobrecarga similar al costo del testeo de un elemento (suponiendo que haya superado primero los demás tests, como por ejemplo el del criterio Delaunay).

Este árbol se reconstruye por cada trabajo al inicio del mismo para contener solo las caras de su propio frente de avance, en lugar de utilizar un árbol único y global para toda la malla. Es conveniente operar de esta manera por dos motivos: porque de esta forma no se requerirá sincronización adicional cuando se plantee la versión en paralelo del algoritmo (ver capítulo 7); y porque así se puede prescindir de operaciones de balanceo y/o colapsado de niveles que serían necesarias para mantener controlada la profundidad del mismo luego de múltiples inserciones y eliminaciones. Por lo expresado en la sección 4.2.1, se utiliza un bucketed-ADT, donde cada nodo del árbol puede almacenar más de un elemento (caras en este caso). La cantidad máxima de elementos por nodo del árbol es fija, por lo que cuando se intenta agregar un nuevo elemento a un nodo que ya tiene esa cantidad máxima de elementos, el dominio que el nodo representa debe ser particionado generando nuevos nodos hijos, y redistribuyendo los elementos (tanto los que ya contenía como el que se quiere agregar) entre estos nuevos nodos hijos. Por esto, al insertar elementos, la profundidad crece. Al eliminarlos (por el cierre de la frontera) se podría opcionalmente eliminar los nodos del árbol que queden vacíos y/o colapsar niveles en los que la cantidad de elementos por nodo lo permita, para reducir la profundidad máxima del árbol. Sin embargo, estas operaciones podrían ser relativamente costosas, ya que para aplicarlas se requiere determinar primero cuándo se puede eliminar o colapsar un nodo o una rama (lo cual implica un algoritmo recursivo) y la eventual eliminación de la memoria de uno o más nodo (que luego podrían volver a ser necesarios en pasos posteriores). Por esto, se optó por no implementar estas operaciones, reduciendo el costo de mantenimiento del árbol, y utilizar un árbol nuevo para cada trabajo de forma que el impacto relacionado al crecimiento innecesario que el mismo podría presentar en su profundidad, se limite solo a un trabajo, contexto en el cual es menos probable que el problema se manifieste.

La estructura utilizada se basa entonces en el concepto de ADT, pero con dos variaciones. Además de utilizar un árbol *bucketed*, el árbol es n-ario en lugar de binario. Si bien el rendimiento de la estructura no varía excesivamente con la cantidad de divisiones y de elementos por hoja, se realizaron mediciones para diferentes configuraciones y se determinó que se consiguen mejores resultados cuando cada nodo interior divide el dominio en 3 o 5 subespacios iguales (ver figura 6.8), utilizando para la división el eje en el cual dicho dominio tiene mayor longitud, y almacenando entre 16 y 32 elementos (caras) por nodo hoja (ver figura 6.9). Más detalles de esta implementación

en particular se describen en el apéndice A.3.

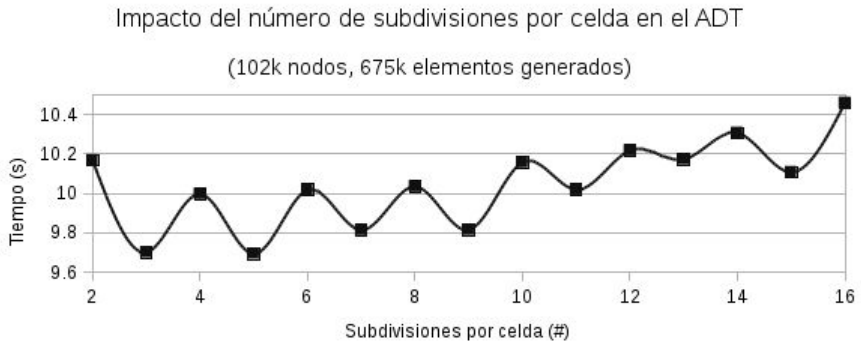

Figura 6.8: Tiempos para la generación de una misma malla utilizando diferentes configuraciones para los nodos interiores del ADT. La oscilación entre cantidades pares e impares se debe a que las impares generan nodos que en memoria ocupan tamaños múltiplos del tamaño de linea de la cache L1 del procesador. Las cantidades pares no cumplen esta propiedad, generando alrededor de un 11 % más de fallos de cache en ese nivel.

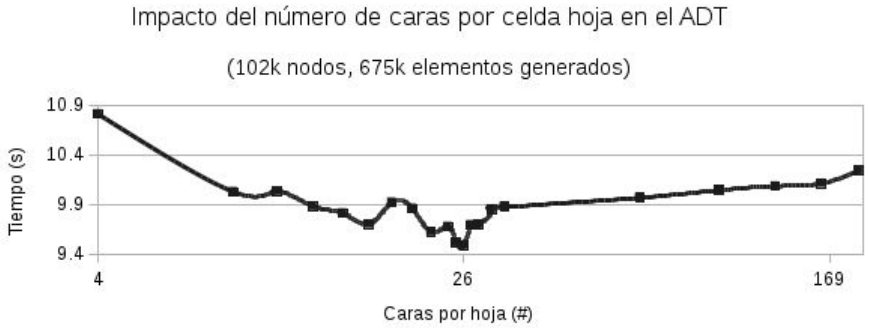

Figura 6.9: Tiempos para la generación de una misma malla utilizando diferentes configuraciones para los nodos hoja del ADT. Almacenando entre 16 y 32 caras por hoja, la variación de tiempos es muy baja, presentando un mínimo en 26.

Es importante destacar que con los cambios recién presentados, las caras del frente de avance pasan a estar referenciadas por el ADT del trabajo. Esto reemplaza a la necesidad de referenciarlas desde sus nodos, por lo que la lista de caras a las que pertenece cada nodo ya no será necesaria. Sin embargo, sí es necesario sincronizar los contenidos entre el ADT y el frente de avance cada vez que se agrega o quita una cara en alguno de ellos. Agregar una cara en alguna de las tres listas del frente es $O(1)$, mientras que agregarla en el ADT es $O(logC)$. Estos tiempos son aceptables y no degradan sensiblemente

el tiempo de ejecución del algoritmo completo. Eliminar una cara del ADT también es $O(logC)$, y eliminarla del frente de avance es $O(1)$ porque en el ADT se guarda la referencia a su posición en dicho frente.

6.3.2. Detección de configuraciones irresolubles y eliminación de elementos

Como se describió anteriormente, se determina que un frente de avance es irresoluble cuando el algoritmo no consigue colocar elementos sin generar intersecciones para ninguna de las caras base pendientes en dicho frente. Considerando que el frente completo se divide en tres sub-listas mediante el plano divisor, esto significa que ninguna de las caras de la lista que contiene las que intersectan al plano divisor permite avanzar. Cada frente detectado como irresoluble se almacena en una lista de frentes irresolubles del trabajo, para evitar que se vuelva a repetir en pasos posteriores. Esto genera la necesidad de comparar dos frentes de avance, uno potencialmente nuevo contra uno almacenado en el conjunto de irresolubles. Un frente de avance es simplemente una lista de caras. Para poder realizar esta comparación rápidamente, cada frente se almacena utilizando un contenedor de caras ordenado, en particular un std::set (que en la mayoría de las implementaciones consiste en un red-black tree[69]). Construir un frente es $O(ClogC)$, y compararlo $O(C)$ en el peor de los casos, aunque mucho menor en el caso promedio. Para almacenar y buscar rápidamente entre los frentes almacenados, se utiliza como contenedor de frentes una estructura basada en una tabla de hash. No se realizó un análisis comparativo más exhaustivo utilizando diferentes estructuras de datos porque las pruebas realizadas determinaron que el tiempo consumido por las consultas a esta estructura no presenta un impacto significativo en el tiempo total de mallado. Esto se debe a que los casos en que es necesario deshacer elementos son muy poco frecuentes (cuando no son necesarios, los frentes no se almacenan), y en los casos en que se requiere es mucho más determinante la política con que se elige cual elemento deshacer y cuando comenzar el proceso.

Se describió hasta aquí cómo se identifica una situación irresoluble, y cómo se determina cual elemento eliminar de entre los posibles candidatos a ser eliminados. Resta analizar entonces cómo se obtiene el conjunto de elementos candidatos a ser eliminados, y qué otros cambios fueron necesarios en las estructuras de datos de la malla y del frente de avance para permitir recuperar un estado anterior consistente luego de eliminar un elemento. Para ello, se agregó a la clase que representa cada cara del frente de avance, una

referencia a un elemento. Si una cara está en el frente es porque o bien forma parte de la frontera inicial, o bien fue generada por la colocación de un elemento. En el primer caso, la referencia de la cara será nula, mientras que en el segundo apuntará hacia el elemento que la generó. Entonces, para recorrer los elementos candidatos a ser eliminados, se recorren las caras del frente de avance, procesando los elementos que estas referencian. Un elemento referenciado podría igualmente no ser eliminable, si por ejemplo fue colocado por otro trabajo. Esto se verifica fácilmente comparando el id del trabajo actual con los ids de trabajos de los nodos del elemento. Luego, al eliminar el elemento, se deben agregar al frente de avance las caras que la eliminación del mismo generó. Estas caras, deberán referenciar a otros elementos, que anteriormente eran vecinos del eliminado. Para lograr esto, cada elemento guarda referencias a sus elementos vecinos. Dado que hay una referencia por cara, y cuatro caras por tetraedro, el mantenimiento de las mismas requiere tiempo constante.

6.4. Resumen del algoritmo propuesto

6.5. Estructuras de datos

Se muestran a continuación (de forma simplificada) las estructuras de datos necesarias para dar soporte a los algoritmos descritos en las secciones anteriores.

```
struct Nodo {
   float x,y,z; // coordenadas
   int id_trabajo; // trabajo al que pertenece
};

struct Cara { // cara de la frontera
   int nodos[3]; // indices de sus nodos, ordenados
   Elemento *el; // elemento que generó esta cara en el frente
   int el_pos; // índice de esta cara entre las caras de *el
   bool flag_prob; // indica si *el es un elemento Delaunay
};

struct Elemento { // tetraedro
   int nodos[4]; // indices de sus nodos, ordenados
```

```
   list<Elemento>::iterator it; // su posición en la
                                // lista de elementos
                                // de la malla
};

struct Trabajo {
   int id_trabajo; // id único del trabajo
   Divisor *divisor; // functor del plano divisor
   list<Cara> frontera; // caras del frente
   int r0, rN; // rango en el vector ref_nods de la
               // malla que contiene los nodos de este
               // trabajo
   RollbackMode mode; // modo de trabajo (para el rollback)
   int mcount; // nro de avances fallidos en el modo actual
               // (para el rollback)
};

class MallaFull {
    vector<Nodo> nodos; // nodos, dato de entrada
    vector<int> ref_nodos; // indices de nodos, para reordenar
    list<Elemento> elementos;
    queue<Trabajo> trabajos;
    Grilla grid; // estructura para ordenamiento espacial
                 // y búsqueda de nodos
public:
    // ...funciones miembro...
};
```

6.6. Algoritmos

El siguiente pseudocódigo describe todos los pasos involucrados en la generación de una tetraedrización a partir de un conjunto de puntos y una frontera impuesta.

```
Sea n un vector de N nodos, y F la lista de caras de la
frontera impuesta (datos de entrada):
// etapa de preparación
 1 -Inicializar la grilla (grid) cargando en ella todos los
    puntos de entrada
```

```
 2 -Crear un arreglo de índices (pts) con enteros de 1 a N
 3 -Registrar cada cara de la frontera impuesta (F) en su
    primer nodo
 4 -Definir el primer trabajo, asignando al mismo la lista
    completa de caras de la frontera y agregarlo a la cola
    de trabajos pendientes (jobs)
 5 -Asignar a todos los nodos el id del primer trabajo
// etapa de mallado
 6 -Mientras jobs no esté vacía
 7     -Construir el ADT (t) a partir de la lista de
        caras del trabajo
 8     -Calcular el AABB del conjunto de nodos del trabajo
 9     -Generar un plano divisor a partir de AABB
10     -Dividir las caras del trabajo en tres listas: lb
        (caras atravesadas por el plano), lr y ll (caras
        a derecha e izquierda del plano)
11     -Establecer el modo de trabajo (mode) en "normal",
        y el contador de intentos fallidos (mcount) en 0
10     -Mientras lb no esté vacía
11        -Por cada cara (c) de lb
12            -Intentar generar un elemento utilizándola
               como cara base
13        -Si no se pudo generar ningún elemento nuevo
14           -Si mode es "normal", cambiar a mode "deshacer"
15           -Incrementar mcount
16           -Si mcount llegó al máximo permitido (max_mc)
17              -Si mode es "deshacer"
18                 -Pasar a mode "slivers" y reiniciar
                    mcount
19              -sino
20                 -Abortar el proceso, retornando un código
                    de error
21           -Registrar la frontera actual en el conjunto de
              fronteras irresolubles (tested)
22           -Repetir
23               -Intentar quitar algún elemento
24           -hasta que el frente actual no se encuentre en
              tested
25           -Si no se pudo quitar ningún elemento, asignar
              max_mc en mcount (para forzar el cambio al
              siguiente modo)
```

```
    // generación de subtrabajos
26     -Reordenar pts de forma que todos los nodos que se
        encuentren a la derecha del plano se ubiquen antes
        que los que se encuentren a la izquierda
27     -Si hay caras en lr, generar un nuevo subtrabajo en
        jobs asignando dicha lista como frente inicial
28     -Si hay caras en ll, generar un nuevo subtrabajo en
        jobs asignando dicha lista como frente inicial
```

Los pasos 12 y 23 de este pseudocódigo son los que requieren descripciones más detalladas, ya que determinan cómo se intenta agregar y quitar un elemento respectivamente. A continuación se detalla el paso 12 (intentar generar un elemento sabiendo que hay caras base pendientes:

```
 1 -Definir el modo de colocación (mcol) en "free"
 2 -Determinar cual celda (c0) de grid contiene el centroide
    de la cara base (c)
 3 -Inicializar una cola de celdas por revisar (q) con c0
 4 -Definir el indice de nodo ganador (imin) como -1 (indica
    que no hay aun un nodo factible)
 5 -Definir la esfera inicial asociada a imin: con
    radio (rmin) infinito y centro (cen) en el origen
 6 -Mientras q no esté vacía
 7    -Calcular la distancia (d) del centro la celda (pc) al
       centro de la esfera (cen)
 8    -Calcular la longitud (l) de la diagonal de la celda
 9    -Si la celda se encuentra del lado adecuado del plano
       que contiene a c (dist(pc,c)>l/2) y toca la esfera
       (pc-cen<=rmin+l/2) entonces
10       -Encolar en q todas las celdas vecinas
11       -Por cada nodo de la celda (in)
12          -Si es factible (no está en c, está del lado
             adecuado del plano de c)
13             -Si esta dentro de la esfera actual (se
                compara utilizando el epsilon del nodo)
               -Si mcol es "full"
14                -Construir el AABB del nuevo elemento
15                -Buscar en el ADT otros AABB de caras
                   del frente que puedan intersectarlo, y por
                   cada uno
16                    -Testear intersecciones con las caras y
```

```
                        aristas del nuevo elemento.
17                     -Si hay intersecciones, pasar al
                        siguiente nodo (saltar al paso 11)
18                  -Si mode no es "normal"
19                     -Construir el frente que generaría el
                        nuevo elemento
20                     -Si el nuevo frente se encuentra en
                        tested pasar al siguiente nodo (saltar
                        al paso 11)
21                -Calcular la esfera definida por c y por in,
                   y actualizar cen, rmin e  imin con los datos
                   del nodo y la nueva esfera
22  -Si no hay nodo ganador, finaliza el intento como fallido
23  -Si mcol es "free"
24     -Ejecutar pasos del 14 al 20 para dicho nodo
25     -Si el nodo no pasa los tests (de intersecciones)
26        -Definir mcol como "full"
27        -Volver al paso 2
28 -Generar el nuevo elemento y agregarlo a la malla
29 -Por cada cara del elemento generado
30    -Si estaba registrada en sus nodos desregistrarla, sino
       registrarla
31    -Determinar de qué lado del plano se encuentra y
       agregarla o eliminarla de la lista que corresponda (lr,
       ll o lb)
32    -Si mcol es "full", marcar el nuevo elemento como
       elemento no-Delaunay
```

Y a continuación se detalla el paso 23 (intentar eliminar un elemento):

```
 1 -Por cada cara (c) en lb
 2    -Si tiene un elemento asociado
 3       -Por cada nodo del elemento
 4          -Verificar si el id del nodo es el mismo que el id
             del trabajo actual
 5       -Si todos los nodos del elemento pertenecen al trabajo
          actual
 6          -Si el elemento está marcado como no-Delaunay
 7             -Agregarlo en una lista de candidatos a ser
                eliminados (to_del_prob)
 8          -sino
```

```
 9             -Agregarlo en una segunda lista de candidatos a
                ser eliminados (to_del_sec)
10 -Si to_del_prob no esta vacía
11    -Eliminar cada elemento de to_del_prob
12 -sino
13    -Elegir aleatoriamente un elemento de to_del_sec y
       eliminarlo
```

Capítulo 7

Paralelización de los algoritmos de tetraedrización

En este capítulo se describe un algoritmo para la generación de una tetraedrización de un conjunto de puntos respetando una frontera impuesta en arquitecturas de hardware paralelo. El desarrollo que aquí se presenta se basa en el algoritmo propuesto en el capítulo 6 (algoritmo serie), por lo que se tomará dicha descripción como punto de partida y se detallarán a continuación solamente los cambios necesarios para su paralelización, y cómo se resuelven los problemas que se presentan. Utilizando los mismos mecanismos es posible adaptar además el algoritmo propuesto en el capítulo 5 (para el caso sin frontera). Si bien ambos casos han sido paralelizados (efectivamente implementados como parte del desarrollo de esta tesis), el análisis que se presenta en este capítulo hace referencia mayormente al primero (frontera impuesta) por ser éste el caso más complejo y general. Solo se mencionarán detalles del segundo caso (sin frontera) cuando existan diferencias no triviales en el mecanismo de paralelización y/o su implementación.

Partiendo entonces del algoritmo descrito en el capítulo anterior, se desarrollaron dos versiones paralelas del mismo, una diseñada para arquitecturas de memoria compartida (para ejecutar en una PC multi-procesador y/o con procesadores multi-core), y otra para arquitecturas híbridas (varias PCs en red formando un cluster, pudiendo contener cada una de ellas procesadores multi-core). La sección 7.1 describe las consideraciones más importantes, de carácter general y comunes a ambos métodos, mientras que las secciones 7.2 y 7.3 detallan las particularidades de cada implementación en relación a la arquitectura de hardware para la cual está diseñada.

7.1. Propiedades comunes de los algoritmos propuestos

7.1.1. Estructuras de datos

Del algoritmo de triangulación/tetraedrización presentado en los capítulos anteriores se desprende una primer estrategia de paralelización obvia, que consiste en delegar diferentes subtrabajos a diferentes hilos de ejecución. Dado un trabajo inicial, el primer paso equivale a mallar una interfaz para realizar una partición del dominio y generar así nuevos subdominios cuyos interiores serán resueltos mediante nuevos trabajos casi totalmente desacoplados. Cada trabajo consume como datos de entrada información que le es propia y exclusiva (como la frontera del mismo, la ubicación de sus nodos dentro del arreglo de referencias a nodos, etc.) e información que comparte con todos los demás trabajos (como el vector de nodos y la grilla). En una ejecución concurrente, un trabajo puede realizar cualquier operación sobre la información de entrada que le es propia sin afectar a los demás trabajos, pero solo puede realizar consultas (es decir accesos de solo-lectura) a las estructuras que son compartidas sin perder esta garantía. Cualquier modificación en estructuras compartidas debe ser analizada e instrumentada para garantizar la correcta sincronización entre los diferentes hilos en caso de ser necesario.

Entre los datos de entrada, hay dos estructuras modificables que serán compartidas por diferentes trabajos de forma evidente: el vector de nodos y el vector de índices (vector auxiliar que se utiliza para evitar reordenar el vector real de nodos). Los nodos del vector de nodos son modificados cada vez que un trabajo añade un nuevo elemento, al registrar y desregistrar las caras en las listas de referencias a caras que los mismos contienen y al perturbarlos (solo para la versión del algoritmo sin frontera impuesta). El vector de índices es parcialmente reordenado al final de cada trabajo para agrupar los nodos de cada subtrabajo generado. Sin embargo, se debe notar que en cada instante de tiempo, cada nodo está asignado a uno y solo un trabajo, y las operaciones mencionadas solo requieren acceso a nodos propios de un trabajo. Por esto, distintos trabajos accederán a distintos elementos de cada vector, y nunca dos trabajos que se ejecuten en simultáneo accederán a una misma posición en ninguno de estos dos vectores. Es decir, al comienzo todos los nodos pertenecen al único trabajo inicial, por lo cual no hay competencia posible por dichos recursos. Cada vez que un nuevo subtrabajo se crea y encola para ser procesado, lo hace a partir de los nodos y elementos de frontera generados por otro trabajo (más grande) que acaba de finalizar, y que por

ende se destruye. Por esto, la propiedad de los nodos tampoco será compartida entre trabajos/subtrabajos de diferentes niveles. Además, dado que un trabajo particiona su conjunto de nodos en tres grupos (según su posición respecto al plano divisor), y no reasigna los nodos que caen justo en la interfaz (nodos sobre el plano divisor) a ninguno de los nuevos subtrabajos, dos subtrabajos de un mismo nivel tampoco pueden competir por el acceso a los recursos de un mismo nodo. Por lo tanto, no es necesario incluir mecanismos de sincronización para estos contenedores.

La grilla (o cualquier otra estructura de ordenamiento espacial que se utilice para los nodos), es una estructura en la cual conceptualmente solo se realizan consultas desde los trabajos, pero nunca modificaciones. Sin embargo, debido a detalles de implementación (uso de técnicas de memoización y similares), estas consultas podrían traducirse en modificaciones, como se describió en la sección 4.4, donde cada consulta a la estructura involucra la modificación de atributos auxiliares que actúan como flags en las celdas que se visitan. Para evitar la necesidad de sincronización en estas operaciones, y además por ser un requisito para la correctitud del mecanismo propuesto, se reemplaza el flag único de la celda por un vector de flags por celda, que contiene un flag por cada hilo de ejecución, técnica ya presentada en la sección 4.4.

La necesidad real de sincronización se presenta en las estructuras que almacenan la información de salida. Cada trabajo produce como resultados nuevos elementos y nuevos subtrabajos. Al finalizar el proceso, todos los elementos deben estar almacenados en un único contenedor de elementos de la malla, por lo que cada trabajo deberá directa o indirectamente agregarlos en dicho contenedor, y el acceso al mismo deberá estar sincronizado para evitar race-conditions. El otro contenedor sobre el que todos los trabajos realizan operaciones de escritura es la cola de trabajos pendientes (donde cada trabajo agrega los subtrabajos que genera). En la implementación desarrollada, cada PC en la que corre el algoritmo (una única PC en el caso de memoria compartida, varias en el caso de memoria local) utiliza una cola de trabajo con prioridades ad-hoc que incorpora internamente y de forma transparente los mecanismos de sincronización necesarios (basados en el uso de condition variables). Los detalles de esta implementación se encuentran en el apéndiceA.2.

7.1.2. Balanceo de carga

Es deseable que cuando el algoritmo corre en una arquitectura que ofrece N posibles hilos de trabajo paralelos, se minimice el tiempo en que hay hilos ociosos (sin trabajo asignado o a la espera de la liberación de un recurso compartido) para lograr así un mejor aprovechamiento de las capacidades de cómputo de dicha arquitectura.

Por ejemplo: en su versión básica, el algoritmo comienza con un solo trabajo definido, por lo que habrá $N - 1$ hilos ociosos hasta que ese trabajo finalice y genere dos nuevos subtrabajos. Desde ese momento y hasta que el primer subtrabajo finalice habrá $N - 2$ hilos ociosos, y así sucesivamente. De esta forma, cada vez que un trabajo finaliza, si hay procesadores ociosos, uno de ellos dejará de estarlo. Considerando que cada trabajo finalizado genera dos nuevos subtrabajos, y que cada vez que esto ocurre se incrementa el número de trabajos corriendo en paralelo, el número de trabajos concurrentes disponibles crece exponencialmente, por lo que luego de finalizados unos pocos subtrabajos ya no hay procesadores ociosos.

El tiempo real que efectivamente toma la generación de suficientes subtrabajos como para comenzar a aprovechar todos los procesadores disponibles depende del mecanismo de elección del plano divisor en los primeros trabajos y eventualmente de la geometría del problema. En los ejemplos presentados se elige un plano perpendicular a alguno de los ejes, de forma que divida en partes iguales al lado más largo del AABB del trabajo. Esta elección tiene un costo computacional casi nulo, pero hace que el tiempo necesario para la generación de los primeros subtrabajos dependa fuertemente de la geometría del problema. Por ejemplo, en una geometría que representa una esfera esta elección generará al comienzo del proceso de mallado los trabajos más costosos de todo el proceso. Por el contrario, en una geometría que se asemeje más su AABB y/o que presente una relación de aspecto muy diferente (donde alguna dimensión sea considerablemente mayor o menor que las restantes) esta variación de costos entre trabajos se verá reducida y en muchas ocasiones se podrá obtener una dirección adecuada para los primeros planos de modo que el tiempo necesario para la resolución de los primeros subtrabajos se reduzca sensiblemente (ver figura 7.1). Posibles estrategias alternativas para aprovechar las capacidades de los procesadores ociosos en esta primer parte de la ejecución se discuten más adelante (sección 7.2.3).

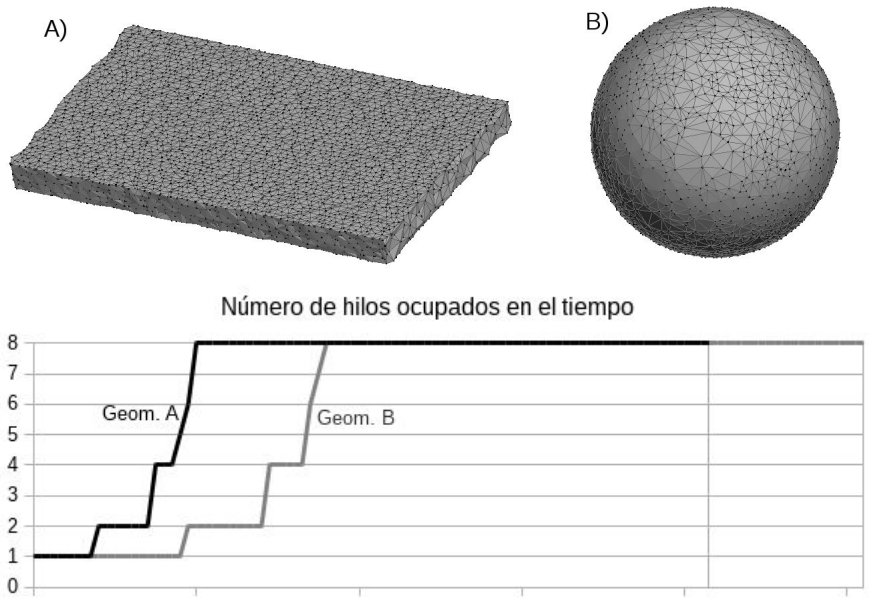

Figura 7.1: Evolución del número de procesadores ocupados durante la generación de dos mallas de tamaño similar, pero que representan geometrías con características diferentes.

El otro momento en el que puede haber procesadores ociosos ocurre cerca de la finalización del proceso. Es decir, cuando quedan pocos subtrabajos por resolver, y estos ya no generan nuevos subtrabajos (porque al generar sus elementos completan sus respectivos volúmenes). En este punto, si la carga fue correctamente balanceada, los trabajos restantes serán trabajos relativamente muy pequeños, por lo que el tiempo en que algunos procesadores estarán ociosos será despreciable. Sin embargo, una mala gestión del orden de resolución de subtrabajos podría hacer que esta situación se presentase prematuramente. Por ejemplo, si en un punto dado solo quedan por resolver 2 trabajos, correspondiendo uno de ellos a un volumen pequeño que no generará más subtrabajos, y el otro a un volumen grande que sí lo hará. El primero se resolverá rápidamente y el procesador que lo resuelva quedará momentáneamente ocioso, hasta que finalice el segundo, lo cual tomará un tiempo mayor, y entonces existan nuevamente dos subtrabajos disponibles.

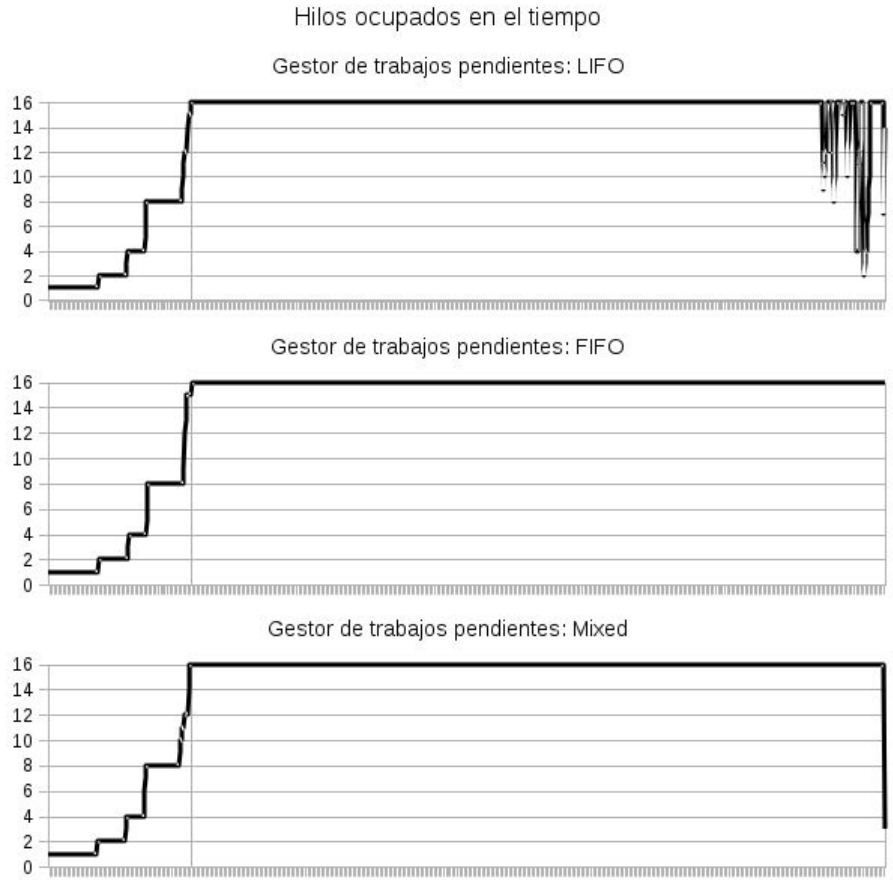

Figura 7.2: Evolución del número de procesadores ocupados durante la generación de una malla de 100k nodos y 675k elementos con 16 hilos de trabajo para los tres mecanismos propuestos. Sobre el final de la ejecución, con la estructura LIFO no se logra balancear correctamente la carga, resultando en procesadores ociosos.

Para evitar esta situación es conveniente resolver primero los trabajos más grandes, y dejar los más pequeños para el final. Dado que se comienza resolviendo el trabajo más grande (en términos de volumen), y cada trabajo genera subtrabajos siempre más pequeños (idealmente la mitad), dejar los más pequeños para el final implica utilizar una cola de trabajos que efectivamente funcione como una estructura FIFO (ver figura 7.2). Esto presenta una ventaja y una desventaja importante. Por un lado, dado que la versión de memoria compartida requerirá enviar datos del trabajo a través de una red a otro procesador, es efectivamente conveniente enviar trabajos grandes para minimizar el número de envíos. Sin embargo, esta estrategia lleva a que la cola de trabajos pendientes crezca exponencialmente, aumentando notable e innecesariamente el consumo de memoria (ver figura 7.3). Para solucionar

estos problemas, sería deseable reemplazar la cola de trabajos por una pila de trabajos, para evitar que se acumule un gran número de pequeños trabajos pendientes. La solución finalmente utilizada implementa una estrategia mixta basada en una doble-cola, sobre la cual se trabaja como si fuera una estructura FIFO o FILO de acuerdo al número de trabajos en la misma, utilizando un umbral predefinido que depende del número máximo de procesadores disponibles en la arquitectura sobre la cual se ejecuta. Entonces, cuando hay una gran cantidad de trabajos disponibles (en relación al número de hilos o procesadores) se prioriza la resolución de los de menor tamaño para evitar que la "cola" de trabajos siga creciendo en memoria; pero cuando quedan relativamente pocos trabajos se prioriza la resolución de los de mayor tamaño, para poder utilizar los de menor tamaño sobre el final para obtener un mejor balance de la carga.

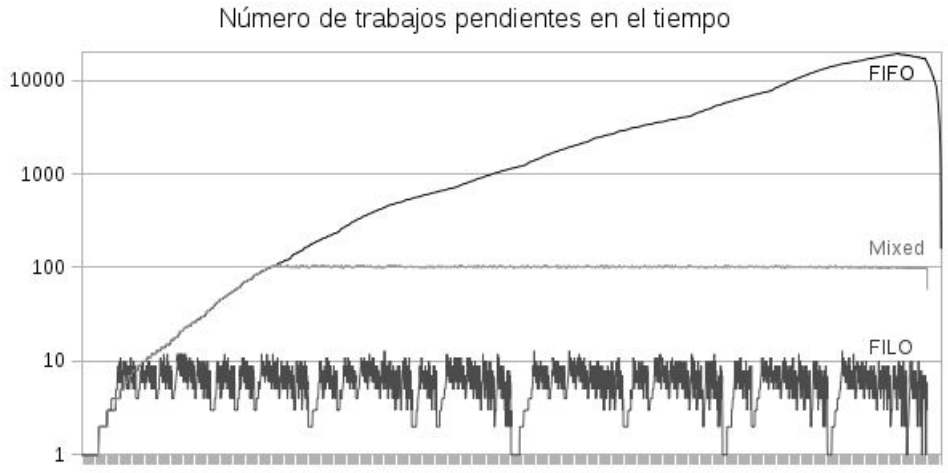

Figura 7.3: Evolución del tamaño de la pila/cola de trabajos durante la generación de una malla de 250k nodos y 1.67M elementos para los tres mecanismos propuestos. Para la estructura FIFO, el número de trabajos pendientes aumenta durante la mayor parte del tiempo, llegando a un máximo del mismo orden que la cantidad de elementos a generar. Para la estructura LIFO, el máximo se mantiene acotado en valores bajos (entre 1 y 12). Finalmente, la estructura mixta propuesta se mantiene debajo del umbral predefinido (en este ejemplo 100).

7.2. Paralelización en arquitecturas de memoria compartida

En la implementación del algoritmo para arquitecturas de memoria compartida se busca minimizar los accesos a memoria principal que requieran directa o indirectamente sincronización entre hilos (accesos de escritura a estructuras de datos compartidas). La estructura de datos propia de la malla que requiere esta sincronización es la que representa al contenedor de elementos, ya que varios hilos procesando diferentes trabajos generarán elementos para dicho contenedor en paralelo.

Al sincronizar el acceso al contenedor de elementos de la malla resultante para que varios trabajos concurrentes puedan generar y registrar elementos en el mismo al ejecutarse en paralelo, existen dos alternativas: (1) utilizar desde todos los trabajos el mismo y único contenedor de elementos general de la malla, sincronizando el acceso cada vez que se crea un nuevo elemento; y (2) utilizar un contenedor local a cada trabajo, para colocar los elementos durante el trabajo sin requerir sincronización, y hacer un solo acceso sincronizado al contenedor general al finalizar el mismo para mover todos los elementos generados en una única operación. En la implementación de referencia se califica como *ThreadSafe* al contenedor para el primer método, y *Cached* al contenedor para el segundo (más detalles en el apéndice A.1. Si la inserción de elementos en el contenedor es una operación frecuente y relativamente dominante, el segundo método debe presentar un rendimiento notablemente superior. Además, el costo extra de mover los elementos del contenedor local al contenedor global una vez finalizado el trabajo puede ser $O(1)$ en memoria compartida si las implementaciones de dichos contenedores se basan en listas enlazadas (lo que permite utilizar la operación conocida como *splice*). Ambas opciones fueron implementadas y comparadas en la versión de memoria compartida. Se encontraron diferencias favorables al segundo método cuando el número de hilos utilizados no supera al número real de núcleos (ver figura 7.4). En caso contrario, su utilización se torna contraproducente (anula las ganancias producidas por la utilización de la tecnología HyperThreadig y deteriora notablemente el rendimiento si se utilizan varios hilos por núcleo). La figura 7.4 muestra tiempos para la generación de mallas de diferentes tamaños utilizando 4 y 8 hilos, en una PC con 4 núcleos reales y tecnología Hyperthreading. La estrategia denominada *Cached* reduce los tiempos entre un 7 % y 17 % utilizando 4 hilos, y entre 21 % y 25 % utilizado 8. Además, confirma que para este algoritmo existe una mejora total de alrededor de 15 % al aprovechar la tecnología HyperThreading.

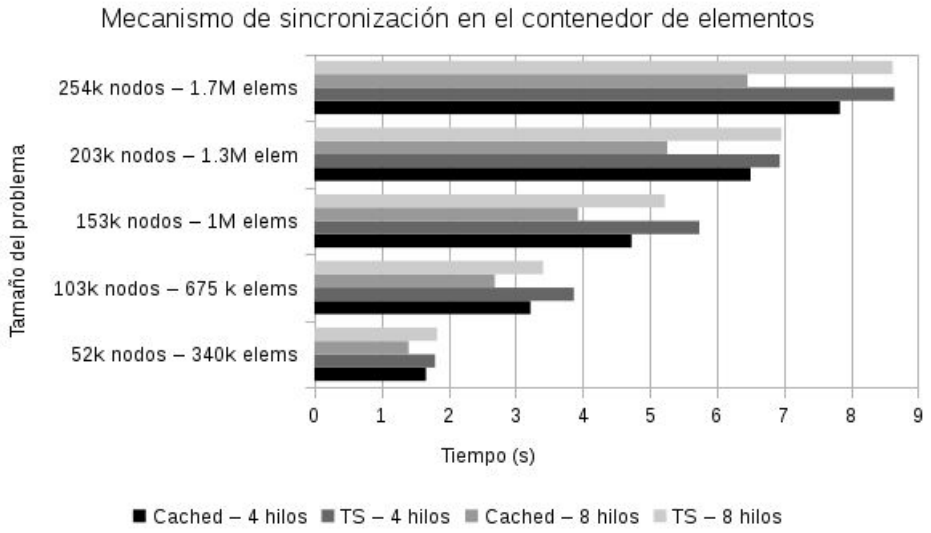

Figura 7.4: Tiempos para generaciones de mallas de diferentes tamaños utilizando las dos estrategias descritas para generar la lista final de elementos. La versión Cached supera a la versión ThreadSafe en todos los casos.

Además de las operaciones sobre el contenedor de elementos, existen otras operaciones muy frecuentes y menos obvias que podrían utilizar algún mecanismo de sincronización interno propio y transparente: la reserva y liberación de memoria dinámica (operaciones sobre el heap del proceso, ver 3.4.1). Dado que el algoritmo utiliza múltiples estructuras de datos basadas en árboles y/o listas enlazadas, éstas efectivamente serán operaciones frecuentes. Para analizar estos efectos y eventualmente paliar sus consecuencias, se modificaron las implementaciones de los contenedores principales para independizarlos de las operaciones de reserva y liberación de memoria, posibilitando la elección en tiempo de compilación entre diferentes estrategias provistas por objetos externos (similares al concepto de *allocator* de la biblioteca estándar de C++, más detalles en el apéndice A.1). De esta forma, se midieron los tiempos para la estrategia inicial, que consistía en reservar/liberar cada vez que se crea/destruye una celda de la lista o árbol, y para una estrategia alternativa (denominada Recycling) que al liberar una celda no libera realmente su memoria sino que la registra en un pool de bloques de memoria disponibles, para poder reutilizarlo en la próxima operación de reserva, evitando así también el costo de las operaciones de reserva de memoria reales. En esta implementación, la pseudo-liberación y pseudo-reserva posterior de memoria se ejecutan en tiempo $O(1)$ sin necesidad de sincronización (se mantiene un pool de recursos reutilizables por thread). Para la generación sin frontera

impuesta no se registraron diferencias en los tiempos de ejecución al utilizar estas alternativas. Sin embargo, la versión con frontera impuesta hace un uso adicional y más intensivo de las funcionalidades que provee este allocator a través del ADT-tree y algunas listas adicionales, razón por la cual la diferencia de performance en este segundo caso sí es observable. Sin embargo, en la mayoría de los casos de prueba, las mejoras son muy reducidas (alrededor del 2 %, principalmente en problemas relativamente grandes, ver figura 7.5), pero en algunos ejemplos concretos se han logrado medir diferencias de hasta un 7.5 % en el tiempo total de ejecución.

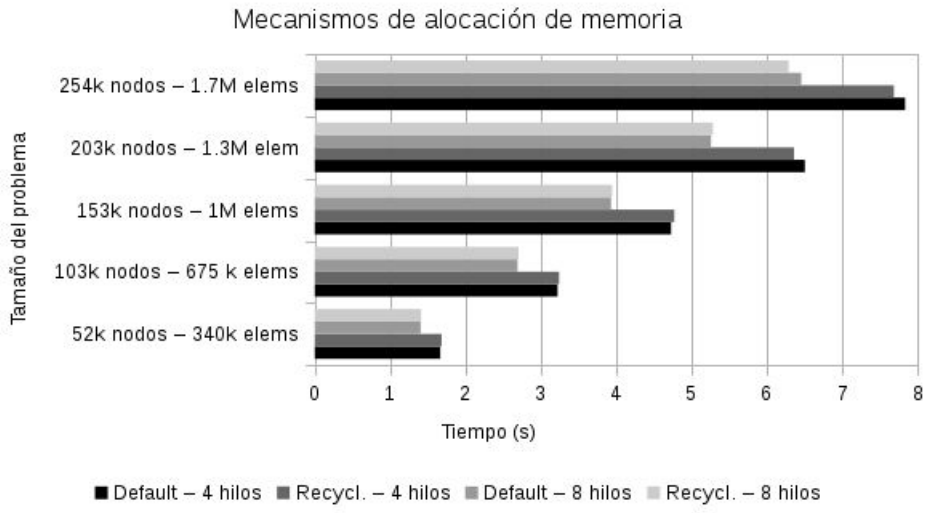

Figura 7.5: Tiempos de generación para mallas de diferentes tamaños utilizando el mecanismo de alocación por defecto, y el denominado Recycling allocator.

7.2.1. Consideraciones del hardware disponible

Se debe notar que las mediciones presentadas en la figura 7.4 corresponden a PCs con un bajo número de procesadores. Este es el caso para la mayoría del hardware usualmente disponible en la actualidad (donde los equipos denominados desktop ofrecen entre 1 y 6 nodos reales, mientras los denominados "server" pueden ofrecer un orden de magnitud adicional). En arquitecturas especiales donde el número de procesadores que utilicen una memoria compartida podría llegar a ser superior, es muy probable que el uso del contenedor local a modo de caché o buffer de resultados tenga un impacto aún mayor en los tiempos de ejecución. Por esta razón, ambas implementaciones están disponibles, pudiendo seleccionarse una u otra alternativa muy

fácilmente en tiempo de compilación (mediante polimorfismo estático).

Más aún, es común que PCs con un mayor número de procesadores presenten una arquitectura NUMA, a diferencia de la arquitectura SMP de las PCs utilizadas para las mediciones antes presentadas. En arquitecturas NUMA el acceso concurrente desde diferentes procesadores a estructuras globales únicas y compartidas (como el contenedor de nodos, o el contenedor de elementos) incluye un costo adicional que sería preferible evitar. En estos casos, aún utilizando arquitecturas de memoria local, conviene utilizar la implementación híbrida para evitar este problema. Trabajos previos[70] confirmaron la validez de esta hipótesis para una versión inicial 2D del algoritmo. En esa versión, los cálculos a realizar para resolver los aspectos geométricos del algoritmo son considerablemente menos costosos (computacionalmente), haciendo que el tiempo acceso a estas estructuras sea más significativo en el tiempo total de ejecución. Además, las estructuras de datos involucradas también resultan más simples y compactas, dando lugar a un mejor aprovechamiento de la memoria cache (considerando además que los niveles más internos de caché en los procesadores multicore se implementan por núcleo). Todas estas condiciones permiten obtener speed-ups superlineales en 2D al utilizar una estrategia similar a la que se describirá en la siguiente sección para arquitecturas de memoria distribuida. En la versión 3D que aquí se presenta no se han observado speed-ups superlineales, pero sí se ha verificado la conveniencia de utilizar el enfoque híbrido frente al enfoque para memoria compartida en arquitecturas NUMA (más detalles de las mediciones en la sección 7.3.2).

Finalmente, es necesario aclarar también que en las PCs de escritorio actuales, donde se dispone de relativamente pocos procesadores (2/4/6 reales, 4/8/12 virtuales, considerando la tecnología Hyper-Threading de los procesadores Intel), el proceso de mallado se encuentra acotado por el uso de CPU (CPU-bounded) y no por el acceso a memoria (memory-bounded). Dichas conclusiones se verificaron a través de un análisis realizado con la herramienta de profiling Intel VTune Amplifier XE[52] (en hardware específico, ya que requiere instrumentación por parte del procesador para realizar estas mediciones), mediante la cual se observa que el factor de utilización del bus de memoria no alcanza niveles críticos para ejemplos de mallado reales. Esto se debe a que las funciones que más tiempo consumen en el proceso son las relacionadas a cálculos geométricos, como cálculos de distancias, detección de intersecciones, o la construcción de las esferas a partir de 4 puntos. Estas últimas afirmaciones se verificaron analizando ejecuciones en serie del algoritmo con las herramientas de profiling de tiempos de ejecución *gprof* y *perf*.

7.2.2. Resultados

En esta sección se presentan resultados de una implementación del algoritmo utilizando la clase std::thread de la biblioteca estándar de C++11 para generar los hilos de trabajo, y los mecanismo de sincronización que la biblioteca estándar provee (std::condition_variable, std::mutex, std::atomic, etc).

Las figuras 7.6 y 7.7 muestran los resultados del algoritmo propuesto sobre una arquitectura de memoria compartida NUMA con 2 procesadores de 12 núcleos reales cada uno (Intel Xeon E5-2697, con HyperThreading deshabilitado). Se puede observar que los tiempos mejoran sensiblemente al incrementar el número de núcleos, hasta llegar a 12, punto a partir del cual ya casi no se obtienen ganancias (más aún, los tiempos se incrementan levemente luego de este punto en la mayoría de los casos). Esto se debe a que cuando se corre con 12 o menos hilos de ejecución en este hardware, la implementación utiliza para cada hilo núcleos de un mismo procesador[1]. A partir del hilo nro. 13, se comienzan a utilizar ambos procesadores, razón por la cual algunos hilos deberán acceder a zonas de la memoria que no serán locales al mismo, y además comenzará a ser necesario sincronizar las memorias cache de ambos procesadores para mantener su coherencia luego de ciertas operaciones. Estos son los dos principales factores por los cuales se degrada el rendimiento al superar el límite de 12 hilos de trabajo.

[1]Se definió la afinidad para cada hilo de trabajo de modo que cada uno utilice un núcleo diferente de un mismo procesador. Si bien no hay en el estándar C++ un mecanismo de control de la afinidad, dado que los compiladores utilizados para las pruebas implementan los std::threads en base a hilos POSIX, se pueden utilizar los mecanismos del estándar POSIX para tal fin.

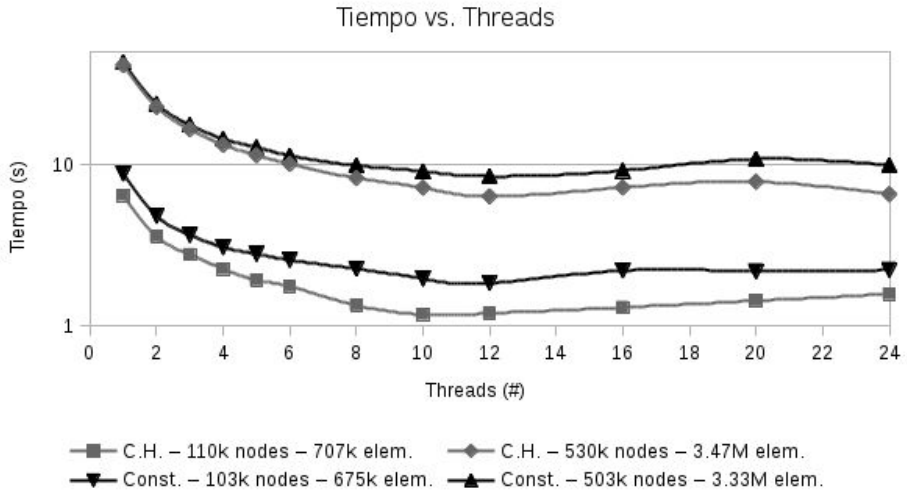

Figura 7.6: Tiempos de ejecución para los algoritmos de mallado con frontera impuesta (C.H) y sin (Const.), en su versión paralela para arquitecturas de memoria compartida, variando el número de procesadores utilizados para la generación de dos mallas de diferentes tamaños.

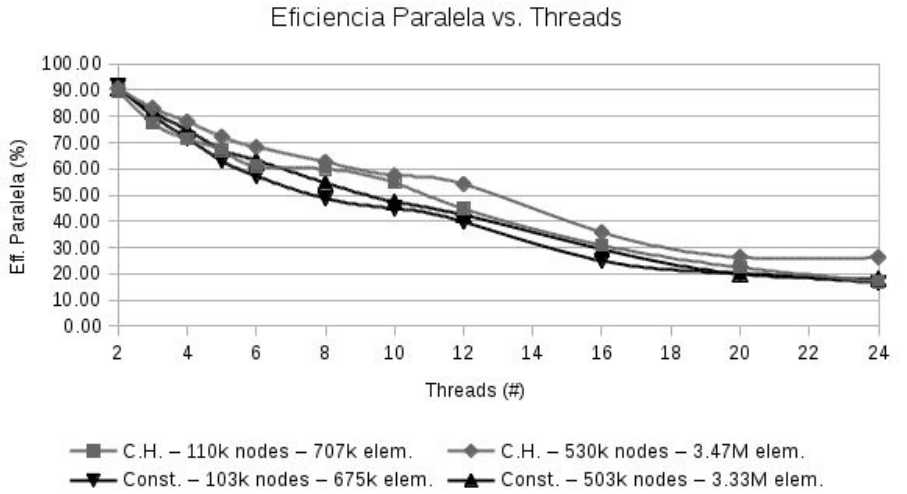

Figura 7.7: Eficiencia paralela para los tiempos presentados en la figura 7.6

Se presentarán mejores resultados para este tipo de arquitecturas en la sección 7.3.2. Considerando ahora solo las mediciones realizadas con hasta 12 procesadores, se observa que si bien la eficiencia paralela decae conforme el número de hilos crece, para el máximo número de núcleos disponibles, la

eficiencia se encuentra entre 42.4 % (en el peor de los casos) y 54.7 % (en el mejor). Se puede observar que se obtiene mayor eficiencia cuanto mayor es la malla a generar. Esta diferencia no es excesiva, pero sí resulta medible y consistente. Además el algoritmo sin frontera impuesta también presenta una mejor eficiencia frente al algoritmo constrained, especialmente para más de 4 o 6 hilos de trabajo. Se conjetura que esta diferencia se acentúa en ese punto debido a la saturación del bus de acceso a memoria, aunque en la práctica no se pudo verificar debido a la no disponibilidad de una herramienta de análisis específica en dicho sistema. La implementación del algoritmo constrained pasa de ser cpu-bounded a ser memory-bounded antes que la versión sin frontera, debido a que utiliza exhaustivamente estructuras de datos adicionales (como el ADT).

Con esta implementación, el tiempo de generación para los caso mayores puede reducirse utilizando 12 procesadores desde 41.45s a 6.57s (versión sin frontera) y desde 43.26s a 8.50s (versión constrained). Es decir, el tiempo se redujo 5.7 y 5.1 veces (speed-up) respectivamente, mientras que la velocidad de generación de elementos pasó de 4.82 a 31.38 millones de elementos por minuto en el caso sin frontera, y desde 4.63 a 23.55 en el caso constrained.

7.2.3. Potenciales mejoras

El algoritmo propuesto presenta dos deficiencias relativamente importantes, que merecen ser discutidas. El primer problema, de diseño del algoritmo, está relacionado a la sub-utilización de los recursos de hardware (particularmente procesadores/núcleos) al comienzo del proceso. El segundo problema es en realidad un detalle de implementación: las estructuras de datos y operaciones involucradas no son en general cache-friendly, y en las arquitecturas de hardware moderno esto puede hacer una diferencia sensible en la performance.

En una arquitectura paralela de N cores, $N-1$ estarán ociosos mientras se resuelve el primer trabajo, $N-2$ mientras se resuelven en paralelo los dos primeros sub-trabajos, $N-4$ mientras se resuelven en paralelo los siguientes 4 sub-trabajos, y así sucesivamente. En general, si se analiza la relación de trabajos/sub-trabajos como un DAG (particularmente un árbol binario), donde los arcos indican las dependencias directas, se puede observar fácilmente que el algoritmo, en la primer parte de la ejecución, resuelve en paralelo todos los trabajos de un mismo nivel, y por lo tanto debe llegar al nivel $log_2(N)$ para lograr ocupar todos los procesadores y núcleos disponibles. Luego de alcanzado este nivel, ya se demostró que el método propuesto

permite balancear dinámicamente la carga hasta el final del proceso. Estas conclusiones son evidentes si se analiza un escenario en el que los subtrabajos de un mismo nivel presentan una complejidad similar entre sí, y donde todos los núcleos disponibles además ofrecen las mismas prestaciones. En un escenario diferente (por ejemplo, con fuertes asimetrías en la geometría, o sobre hardware heterogéneo) no es posible asociar directamente cada nivel en el árbol de trabajo con un período de tiempo en la ejecución, pero el problema persiste de igual manera hasta el punto en el que se resolvieron $T_N = \Sigma_{i=0}^{log_2(N)} 2^i$ trabajos, aunque no se correspondan con los T_N trabajos de los primeros $log_2(N)$ niveles del árbol.

Se implementaron algunos prototipos para una de las posibles variaciones para intentar solucionar este problema, utilizando diversos mecanismos para repartir la carga de un trabajo entre varios threads, pero en ningún caso se obtuvieron resultados satisfactorios. En general, las operaciones de sincronización adicionales que deben introducirse para que esto sea posible generan una gran cantidad de bloqueos durante la ejecución, haciendo que la ganancia sea mínima, o a veces incluso haciendo que el costo de esta sobrecarga supere a los beneficios. El principal problema radica en que no hay garantías de que dos threads puedan avanzar sobre diferentes caras del frente de avance de un mismo trabajo sin generar elementos incompatibles entre sí. Uno de los enfoques posibles que se evaluó en un prototipo y se descartó por no ofrecer mejoras reales en los tiempos, consiste en separar el proceso de colocación de un elemento en 2 pasos. En el primero se determina el elemento a colocar seleccionando el nodo ganador para una dada cara base, pero no se coloca el elemento. En el segundo paso, se reevalúa la factibilidad de colocar dicho elemento, y se lo agrega efectivamente en la malla en caso de seguir siendo válido. De esta forma, el primer paso solo requiere acceso de lectura a las estructuras de datos involucradas, mientras que solo el segundo aplica las modificaciones. Esto significa que en cada instante solo un thread puede estar ejecutando el segundo paso, pero muchos pueden estar ejecutando en simultáneo el primero, tomando cada uno diferentes caras bases de las disponibles en el frente actual. Es de esperarse que en la mayoría de los casos, el elemento potencial seleccionado en el primer paso siga siendo válido a la hora de ejecutar el segundo. Dado que el primer paso es en general mucho más costoso que el segundo, podría argumentarse que este mecanismo lograría mejorar efectivamente la eficiencia paralela del trabajo. Sin embargo, se debe considerar que aunque los threads que se encuentran en el primer paso acceden a las estructuras de datos (como por ejemplo las listas de caras por nodo, o el ADT) solo para lectura, el hecho de que otro thread las esté modificando puede hacer que los primeros observen estados inconsistentes. Nuevamente,

para solucionar esto se deben agregar mecanismos de sincronización clásicos (generalmente mutexes), o utilizar estructuras de datos más complejas que garanticen la integridad en todo momento en base a operaciones atómicas (estructuras de datos lock-free). En el primer caso se genera un gran número de nuevas secciones críticas (lo cual limita su utilidad a un número de threads muy bajo), mientras que en el segundo se incrementa sensiblemente la complejidad de las estructuras de datos. Se experimentó con la primera de estas alternativas, generando un prototipo funcional, y los primeros resultados de estas pruebas determinaron que la relación costo-beneficio no es favorable. Las reducciones en los tiempos de ejecución fueron marginales o limitadas a números muy bajos de threads, frente a la complejidad introducida tanto en el método como en su implementación.

Esta primer alternativa, en sus diferentes variantes (otras formas de paralelizar las operaciones internas de un trabajo), son solo aplicables a la versión para memoria compartida. En la sección 7.3.3 se presentan más opciones, pensadas para arquitecturas de memoria local, pero también aplicables a esta implementación para memoria compartida.

En relación a los patrones de acceso a memoria, se podría decir que este es un algoritmo fuertemente *cache-unfriendly*, debido a que el acceso en la mayoría de los casos se da en orden aleatorio, y a que la mayoría de las estructuras se basan en celdas enlazadas (son variaciones de árboles y listas). Se debe aclarar nuevamente que la importancia del uso de la memoria cache fue tenida en cuenta al seleccionar y diseñar las estructuras de datos propuestas, pero la naturaleza propia del método de mallado depende de la posibilidad de acceso aleatorio a las estructuras. Esto se basa en la idea de que dada la gran cantidad de datos involucrados, en los procesos de generación de mallas no estructuradas se suele obtener un beneficio mucho mayor reduciendo el conteo de operaciones (con procesos de descarte masivo y estructuras de ordenamiento espacial), que intentando disminuir los costos de dichas operaciones (con estrategias como sistematizar el acceso a memoria para que sea ordenado). Sin embargo, aún utilizando estructuras de datos basadas en celdas enlazadas, se han implementado estrategias especiales de alocación de memoria para minimizar el impacto (ver sección 4.1.3).

7.3. Paralelización en arquitecturas de memoria distribuida

En una arquitectura de memoria local los diferentes hilos de trabajo estarán distribuidos en diferentes procesos, que a su vez se ejecutarán en diferentes procesadores de una red (cada uno con su memoria local), y deberán añadirse operaciones de comunicación entre los mismos. Por ejemplo, cuando un subtrabajo generado en un procesador deba ser enviado a otro diferente, se deberá tener en cuenta que el segundo procesador no tiene acceso directo a la memoria del primero (como sí ocurría con memoria compartida), y que por lo tanto el primero deberá comunicarle al segundo la información que este segundo aún no posea y sea necesaria para resolver el trabajo (por ejemplo, los elementos de frontera, o la lista de nodos propios). Las operaciones de comunicación son consideradas operaciones muy costosas (en tiempo), por lo que al diseñar la estrategia para arquitecturas de memoria distribuida se deben minimizar estas instancias de comunicación entre procesos ubicados en diferentes nodos de un cluster. Generalmente se ejecutará un proceso en cada procesador (nodo de un cluster), y el proceso podrá utilizar múltiples hilos de ejecución. Por esto, en los siguientes ejemplos y descripciones se asume una relación 1 a 1 entre procesos y procesadores, aunque el algoritmo resultante no impondrá esta relación como restricción para su ejecución.

La estrategia general es similar a la descrita para arquitecturas de memoria compartida: diferentes subtrabajos se asignan a diferentes procesadores. Sin embargo, los criterios de priorización y asignación de tareas desde la cola de trabajos hacia los distintos procesadores deben repensarse para obedecer ahora al alto costo adicional de la comunicación. Se debe entonces responder a dos preguntas importantes:

1. ¿qué información deberán comunicarse entre sí los procesadores, y en qué momento?
2. cuando haya procesadores ociosos y trabajos disponibles en la cola de trabajos ¿cómo decidir qué trabajo se va a asignar y a cuál procesador?

La primer pregunta es importante, dado que un procesador no necesita conocer el estado completo de todo el sistema para resolver su trabajo. Por ejemplo, no necesita conocer nada acerca de los nodos o elementos de otros trabajos, ni necesita conocer nada acerca de la cola de trabajos. Además, al comunicar el estado desde un proceso a otro, hay datos que pueden omitirse porque pueden ser recalculados por el segundo proceso. Por ejemplo, las estructuras de ordenamiento espacial (como la grilla de nodos) pueden ser reconstruidas por cada procesador a partir del conjunto de nodos, y evitar

así la comunicación de la misma.

La segunda pregunta es importante porque se debe buscar una estrategia que permita que un mismo procesador resuelva varios subtrabajos que involucren a un mismo conjunto de nodos, para minimizar la comunicación necesaria. Es decir, cada subtrabajo generado se debería resolver (siempre que sea posible) en el mismo procesador que lo generó. Por otro lado, al seleccionar un subtrabajo de entre varios disponibles para asignar a un procesador determinado del cluster, un subtrabajo de menor tamaño se traduce directamente en menor tiempo de comunicación necesario para que dicho procesador comience a trabajar, pero un subtrabajo de mayor tamaño en general implica que el procesador podrá trabajar mucho más tiempo sin volver a requerir una etapa de comunicación. En general, para minimizar el impacto de las sobrecargas propias de los mecanismos de comunicación, es preferible comunicar un mensaje de gran tamaño, frente al mismo volumen de datos repartido en muchos mensajes pequeños. Por estos motivos, se buscó priorizar los envíos de trabajos grandes, y se delegó la responsabilidad de reducir estos tiempos de comunicación a posibles estrategias de compresión de los datos comunicados. Por otro lado, para poder aplicar estas políticas, es necesario contar con cierta información global del estado del sistema (¿qué procesadores están ocupados? ¿qué trabajos están pendientes? etc).

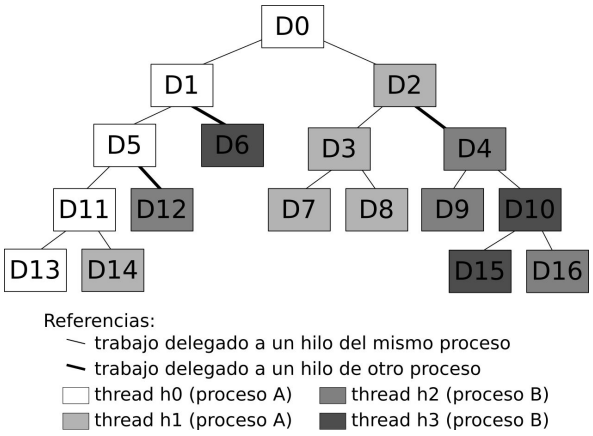

Figura 7.8: Jerarquía de trabajos para el ejemplo de la figura 7.9. El código de colores muestra cómo los trabajos D0 a D16 se distribuyen entre 4 hilos en 2 procesadores.

La figura 7.9 muestra como ejemplo un diagrama temporal para la generación de una malla que requiere la ejecución total de 17 sub-trabajos (figura 7.8), utilizando para ello dos procesadores que ejecutan 2 hilos de trabajo cada uno. El proceso comienza en el tiempo $t0$, instante en el cual se asigna el trabajo inicial $D0$ al primer hilo de trabajo ($h0$, en el procesador A). En el

instante $t1$ este trabajo finaliza y obtiene por resultado dos nuevos trabajos $D1$ y $D2$, que se distribuyen entre los dos threads disponibles en ese mismo procesador. Se prefiere priorizar la asignación a threads de un mismo procesador para evitar operaciones que requieran comunicación entre procesos. En el instante $t2$, finaliza el trabajo $D2$ dando lugar a dos nuevos trabajos $D3$y $D4$. Dado que en el procesador A solo hay solo un thread disponible (el del trabajo recién finalizado), el segundo trabajo se transfiere a el procesador B. El tiempo entre $t2$ y $t3$ es el tiempo que consume la comunicación de los datos del trabajo entre ambos procesadores. En $t4$, el thread $h1$ produce dos nuevos trabajos ($D7$ y $D8$, a partir de $D3$), pero ya no quedan hilos ociosos en ningún procesador, por lo que uno de estos trabajos ($D8$) permanecerá en la cola de trabajos del proceso (A) hasta que algún otro thread se desocupe. Esto ocurre en $t5$, cuando $h1$ finaliza con el trabajo $D7$, y este trabajo ya no genera nuevos subtrabajos. Otra situación importante para el análisis ocurre en $t6$, cuando el thread $h1$ finaliza el trabajo $D8$ (que tampoco produce nuevos subtrabajos) y no hay en ese momento trabajos pendientes encolados en ningún procesador. Este thread queda ocioso hasta que finaliza $D11$ en $t7$ y genera dos nuevos trabajos. Finalmente, en $t8$, finaliza el último trabajo ($D16$) y este thread ($h2$) pasa a estado ocioso. En este momento, se determina la finalización del proceso de mallado, ya que todos los threads se encuentran ociosos.

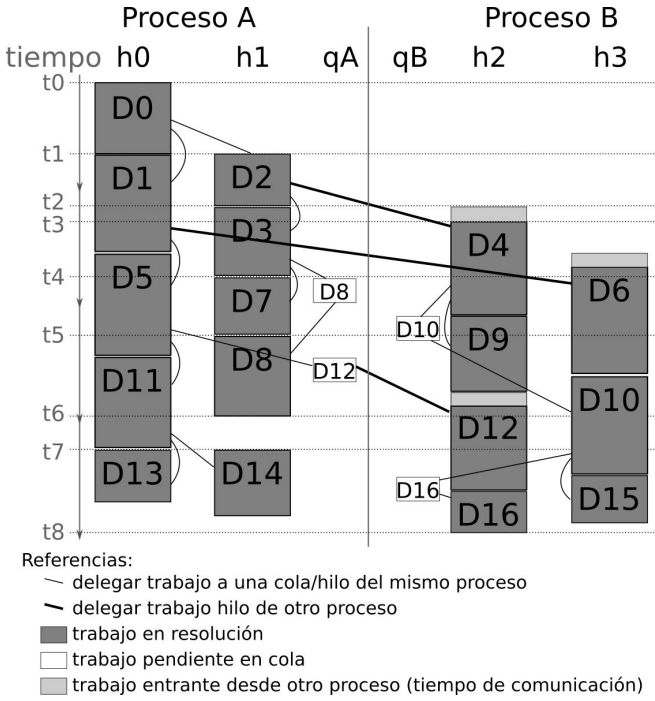

Figura 7.9: Ejemplo de el avance de la resolución de los 17 trabajos para un mallado, indicando cómo se distribuyen y comunican los trabajos entre hilos (h0-h3) de los procesos (A,B) y sus respectivas colas de trabajos pendientes (qA,qB) a medida que se generan.

7.3.1. Arquitectura Master-Slave

Por todo lo expresado en las secciones previas, la solución propuesta para arquitecturas híbridas presenta dos diferencias importante con respecto a la versión de memoria compartida. En primer lugar, se agrega en el sistema un hilo adicional, al que se denominará *master*, que no realiza directamente tareas de mallado, sino que se encarga de gestionar y sincronizar la asignación de trabajos a procesadores y las comunicaciones entre ellos. En segundo lugar, cada procesador tendrá además otro hilo adicional que tampoco realizará directamente tareas de mallado, que se denominará *slave*. Cada hilo slave intervendrá en las comunicaciones desde o hacia ese procesador en el cluster, y dispondrá además de una cola de trabajos propia. Ya no existe entonces una "cola" de trabajos centralizada, sino que la misma se encuentra distribuida entre los diferentes nodos de la red. En cada momento, cada trabajo pendiente se encontrará en solo una de las colas locales (no habrá duplicados). El hilo master tendrá información general acerca del estado de cada proceso y de su cola local de trabajos, a partir de la cual determinará la asignación de trabajos a procesadores ociosos, y señalizará a los hilos slave que corres-

ponda para que estos efectúen la comunicación de los datos específicos de los subtrabajos entre ellos. Para esto, cada nodo comunicará al hilo master, también mediante el hilo slave su nuevo estado al finalizar un subtrabajo.

La estrategia final combina la metodología mencionada, con la presentada anteriormente para arquitecturas de memoria compartida. Cada nodo de un cluster puede presentar un arquitectura local de memoria compartida. El hilo slave priorizará la resolución de los subtrabajos que un procesador genere mediante los mecanismos de memoria compartida, para disminuir la comunicación entre nodos. Cuando un procesador finaliza con la asignación de todos sus subtrabajos y aún tiene hilos ociosos, este estado es notificado al hilo master. Por otra parte, cuando un procesador que tiene todos sus hilos ocupados, genera nuevos trabajos, este estado de "sobrecarga" también es notificado al hilo master. El master entonces puede utilizar esta información para decidir cuando es conveniente que un procesador delegue uno de sus trabajos pendientes a otro. Desde este punto de vista, al hilo master solo le interesan dos propiedades de los estados internos de un determinado slave: a) si su procesador dispone de al menos un hilo ocioso para recibir sub-trabajos de otro slave; o b) si dispone de sub-trabajos adicionales pendientes que puedan ser enviados a otro slave. Hay finalmente una tercer propiedad que puede ser de interés para el master: consiste en c) conocer cuando un procesador no solo dispone de hilos ociosos, sino que todos sus hilos están ociosos. Esto se debe a que la condición de parada (finalización) del proceso total puede plantearse como el estado del sistema en el cual ningún hilo de ningún procesador está ocupado.

La implementación propuesta para el hilo master utiliza solo un identificador de estado por cada slave, ya que los tres estados de interés mencionados pueden pensarse como mutuamente excluyentes (modificando b para no incluir c). El master solo actualiza estos estados cuando recibe notificaciones desde los slaves. Cuando ocurre dicho evento, utiliza el vector de estados para determinar si debe ordenar la transferencia de un sub-trabajo entre dos hilos. Las propiedades importantes del vector de estado (como cantidad de procesadores sub-ocupados o sobre-ocupados) se encuentran memoizadas para evitar recorrerlo en cada evento. El envío (la comunicación de un proceso a otro) será en realidad llevado a cabo por los dos hilos slave de los dos procesadores involucrados (el que tiene trabajos extra en su cola, y el que estaba ocioso y debe recibir uno de estos trabajos). De esta forma, las comunicaciones más costosas se dan solo entre pares de slaves, mientras que los mensajes desde y hacia el master son siempre mensajes muy cortos. Esto evita que el diálogo entre los slaves y el master se convierta en un cuello de botella para el sistema. Analizado los slaves como máquinas de estados, se

puede reconocer una variedad de estados mayor a la propuesta en la vista del master. Se analizarán a continuación los posibles estados, y sus transiciones.

Se pueden identificar 4 estados básicos para un slave: Libre (cuando todos sus threads se encuentran ociosos), Parcial (cuando al menos un hilo se encuentra trabajando, y al menos un hilo se encuentra ocioso), Ocupado (cuando todos los threads se encuentran trabajando, pero no hay trabajos adicionales pendientes en la cola de trabajos) y Extra (cuando todos sus hilos están trabajando y además hay trabajos en espera en la cola). En cualquiera de los dos primeros estados, el slave dispone de recursos sub-utilizados (threads ociosos) y es por lo tanto un candidato para que otro thread le delegue trabajo. En el tercer estado, el thread no debería recibir nuevos trabajos, ya que todos sus recursos están siendo utilizados. En este estado, al igual que en los dos primeros, no hay trabajo pendiente que convenga delegar a otro proceso. Solo en el último estado el thread dispone de más trabajos que threads y por lo tanto convendría delegar trabajo a otros procesos. La distinción entre los dos primeros estados solo es necesaria para el criterio de finalización del mallado completo (para que el master identifique cuando ya no se deben esperar nuevos trabajos de ningún hilo). Las transiciones en esta lista inicial de posibles estados se dan solo entre estados consecutivos en dicha lista, y ocurren en un sentido cuando se genera un nuevo trabajo, y en otro cuando un thread finaliza todos sus trabajos. El primer caso se da cuando se asigna al primer thread el trabajo inicial, o cuando un thread finaliza un trabajo que genera dos nuevos subtrabajos. En ese caso, uno de ellos será resuelto por el mismo thread que lo generó, mientras que el otro se asignará a otro thread o a la cola de trabajos pendientes del proceso. La transición en el sentido contrario se da cuando un trabajo finaliza y no genera nuevos subtrabajos, caso en el cual el thread que lo resolvió queda libre. La figura 7.10 muestra estos 4 estados y sus transiciones básicas.

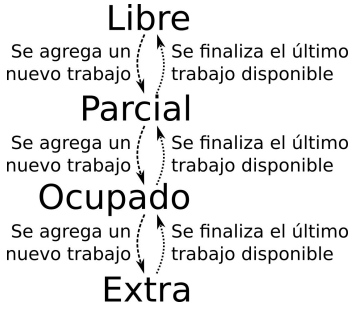

Figura 7.10: Estados básicos de un slave y sus transiciones cuando no interactúa con otros procesos.

Para evitar que un thread envíe o reciba por error dos trabajos en simultáneo, se añaden dos estados adicionales que se identifican con estas operaciones. El slave pasará a estos estados solo cuando el master se lo ordene. Por ejemplo, si un slave está en estado Extra, debe notificárselo al master y esperar a que el master identifique otro slave en estado Libre o Parcial, para ordenar la delegación de trabajo del primero al segundo. En este escenario entonces, el master ordena al primero que pase a estado Enviando, y al segundo a estado Recibiendo. La figura 7.11 muestra el diagrama extendido incluyendo estos estados adicionales y sus nuevas transiciones.

Figura 7.11: Estados adicionales de un slave y sus transiciones al interactuar con otros procesos.

La figura 7.12 muestra en los arcos que representan los cambios de estado, los cambios que son de interés particular para el master, y que deberían ser directa o indirectamente notificados.

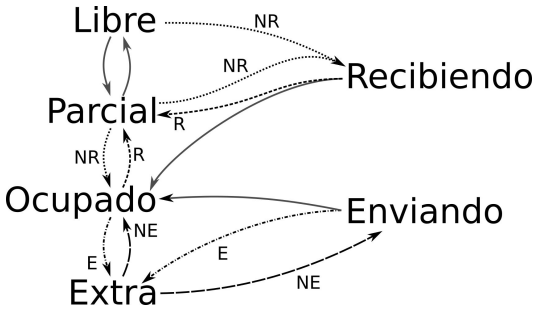

Figura 7.12: Notificaciones de un slave al master al cambiar su estado interno. E: tiene trabajos extra para enviar a slave, NE: ya no tiene trabajos en cola para enviar, R: tiene threads ociosos que pueden recibir trabajos de otros slaves, NR: ya no tiene threads ociosos.

No todos los cambios de estado son de interés para el master, ya que algunos no modifican la conveniencia de enviar/recibir trabajos desde/hacia

ese slave. Notar además que dado que las transiciones hacia los estados Recibiendo y Enviando se producen solo por orden del master, no sería necesario que los slaves notifiquen al master sobre las mismas. Sin embargo, por la naturaleza asíncrona de las comunicaciones entre slaves y master, y dado que estas comunicaciones usualmente involucrarán transmisiones por red y por ello la latencia será comparativamente alta, se podrían generar situaciones no previstas en las figuras 7.11 y 7.12.

La figura 7.13 muestra dos ejemplos de situaciones no deseables que puede producir la naturaleza asíncrona de estas comunicaciones. En el primer caso, el master arregla la delegación de un trabajo hacia un procesador en estado Parcial, pero en el tiempo que consume la comunicación de este mensaje hacia los slaves involucrados (o en el tiempo que consumió previamente la comunicación del slave al master para notificar su estado Parcial) el slave cambia a estado Ocupado (porque otro thread de dicho slave generó nuevos subtrabajos). En este caso, cuando el thread recibe la orden de prepararse para recibir un trabajo de otro slave, pasa de Ocupado a Recibiendo (transición no prevista anteriormente) y debe luego cancelar el envío (comunicar la cancelación al otro slave) para evitar sobrecargarse. En el segundo caso, mientras el master recibe la notificación de que un slave tiene trabajos extra y ordena la delegación de un trabajo a otro slave, el primero desocupa uno de sus threads y comienza a resolver el trabajo extra por sí mismo. Este slave debe entonces cancelar el envío y pasar de estado Enviando a Ocupado nuevamente (transición asociada anteriormente a otro evento diferente).

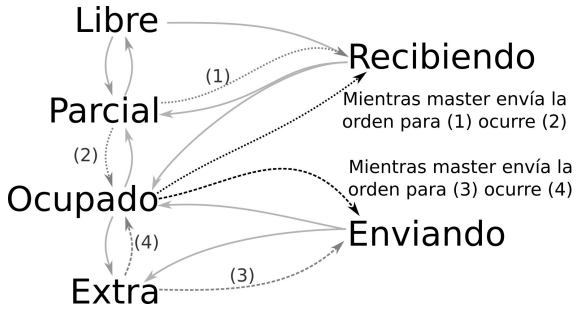

Figura 7.13: Transiciones adicionales debido a la latencia en la comunicación con el thread master.

Estos problemas llevan a la necesidad de enviar un mayor número de mensajes, debido a la comunicación de ciertos cambios de estado que en un escenario ideal serían redundantes (podrían ser deducidos por la otra parte), y también a la necesidad de comunicar la ocurrencia de transiciones entre estados no previstas (que se originan producto de la latencia en la comunica-

ción). Es por esto que el slave debe comunicar en general cualquier cambio de su estado al master, sin intentar omitir los que podrían ser redundantes, y que el master debe determinar por sí mismo cuales de estos mensajes ignorar. Es importante entonces considerar estas situaciones para evitar distribuir la carga de forma incorrecta por la falta de coherencia entre los estados reales y los percibidos por el master. Pero también es importante para garantizar la finitud del algoritmo, ya que estas diferencias pueden generar condiciones de carrera que le impidan al master la detección de la condición de finalización, o que generen deadlocks en las comunicaciones entre los slaves. Por ejemplo, si a un slave le llega la orden de recibir un trabajo, y mientras la orden se envía el slave sobrecargado finaliza sus trabajos, siendo este el último slave ocupado, el master podría indicarle la finalización del proceso completo antes de que este alcance a notificarle al otro slave acerca de la cancelación del envío. Entonces, el slave que estaba inicialmente ocupado finaliza, mientras que el slave que estaba libre queda a la espera del envío.

La solución para evitar estos problemas, se basa en asignar en el master un estado especial a los slaves involucrados en una comunicación, e ignorar todos los mensajes de estados de dichos slaves hasta que los mismos indiquen mediante un mensaje especial que han recibido la orden de comunicación. Los slaves, al recibir una orden, además de hacer efectivo, o cancelar el envío del trabajo, deben enviar al master ese mensaje especial junto con su nuevo estado.

Finalmente, se puede notar que en un cluster compuesto por N nodos y M núcleos distribuidos entre esos nodos (siempre $M >= N$), la solución propuesta utiliza $C = M + N + 1$ threads de ejecución. En general, cuando la carga se distribuye uniformemente entre los threads y las operaciones de bloqueo/suspensión no son frecuentes, no es conveniente utilizar un número de threads mayor a la cantidad de núcleos disponibles. Sin embargo, en esta implementación, los threads master y slave tienen una carga sensiblemente menor, y sus principales operaciones son operaciones de I/O a través de la red. Por estos motivos, en un sistema operativo moderno, estos threads estarán en estado "suspendido" la mayor parte del tiempo, y su actividad no representará un impacto negativo en el rendimiento de los demás threads (los que efectivamente resuelven los subtrabajos y generan elementos en la malla).

7.3.2. Resultados

En esta sección se presentan resultados de una implementación del algoritmo utilizando la biblioteca MPI (particularmente la implementación MPICH2[71]) para la comunicación entre procesos, y nuevamente el soporte para threads y los mecanismos de sincronización de C++11 para los múltiples hilos de cada proceso.

Estrategia para memoria distribuida

Las figuras 7.14 y 7.15 muestran los tiempos de generación de dos mallas de diferentes tamaños de la implementación para memoria distribuida exclusiva ("flat-mpi", solo un hilo de trabajo por proceso), utilizando hasta 6 nodos de un cluster, y hasta 4 procesadores (reales) en cada nodo. Las figuras muestran diferentes variantes que incluyen en algunos casos más de una forma de distribuir una misma cantidad de procesos. Por ejemplo, se tomaron tiempos utilizando 4 procesos distribuidos de tres formas diferentes: todos en nodos diferentes utilizando 4 nodos, dos procesos por nodo utilizando 2 nodos, y 4 procesos en un mismo nodo. Para cada número de procesos, no se registraron grandes diferencias entre las distintas distribuciones de los mismos en distintas cantidades de nodos del cluster. En general se espera que a igual cantidad de procesos, una configuración que utilice menor cantidad de nodos del cluster sea más rápida, ya que la comunicación entre procesos de un mismo nodo se hace a través de la memoria RAM, mientras que entre nodos diferentes a través de una red (en este caso Ethernet de 1Gb). El hecho de que las diferencias resulten mínimas indica que el número y/o el volumen de datos de las comunicaciones requeridas es relativamente bajo, y que por lo tanto la sobrecarga introducida por dichas operaciones no es determinante para el rendimiento en la solución propuesta e implementada.

La eficiencia paralela decae hasta 38.3 % y 47.9 % en el peor caso, que se corresponde con la configuración con mayor número de procesos (16 procesos). Nuevamente se observa que la eficiencia aumenta cuando aumenta el tamaño del problema. Esta diferencia se acentúa más en este mecanismo de paralelización (en comparación con los resultados anteriores para el mecanismo para memoria compartida). A pesar de estar igualmente lejos de la eficiencia ideal, sigue siendo suficientemente alta como para justificar la paralelización y permitir obtener ganancias tangibles al agregar mayor poder de cómputo al sistema (nuevos nodos al cluster).

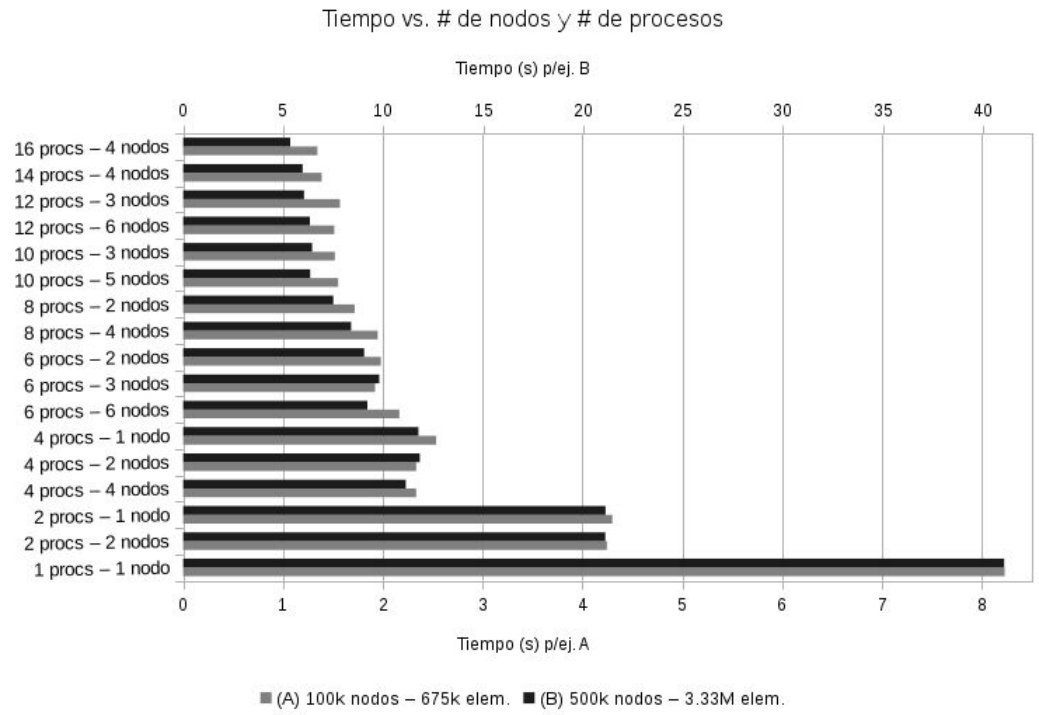

Figura 7.14: Tiempos de ejecución para el algoritmo de mallado con frontera impuesta en su versión paralela para arquitecturas de memoria distribuida, variando el número de procesos y/o la distribución de los mismos en nodos multinúcleo de un cluster, generando dos mallas de diferentes tamaños.

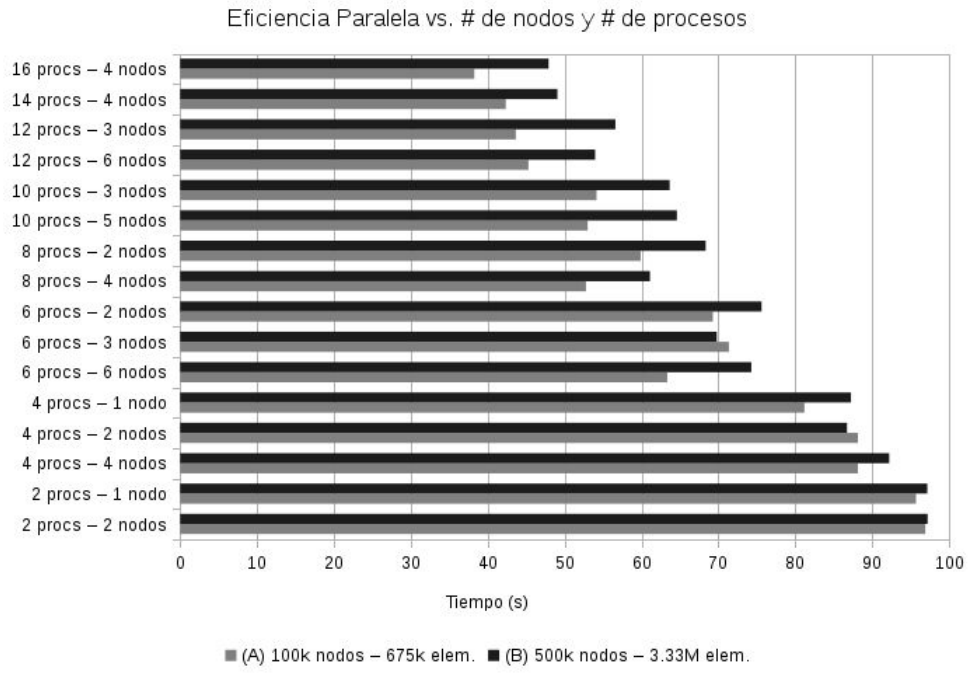

Figura 7.15: Eficiencia paralela para los casos presentados en la figura 7.14.

La figura 7.16 resume la evolución de los tiempos para los dos casos de ejemplo (dos mallas de diferentes tamaños) en función de la cantidad de procesos utilizado. Por ejemplo: el tiempo de generación para el caso mayor puede reducirse desde 41.1s a 5.35s utilizando los 4 procesadores, y los 4 núcleos de cada uno de ellos. Es decir, el tiempo se redujo 7.8 veces (speed-up), mientras la velocidad de generación de elementos pasó de 4.87 a 37.37 millones de elementos por minuto.

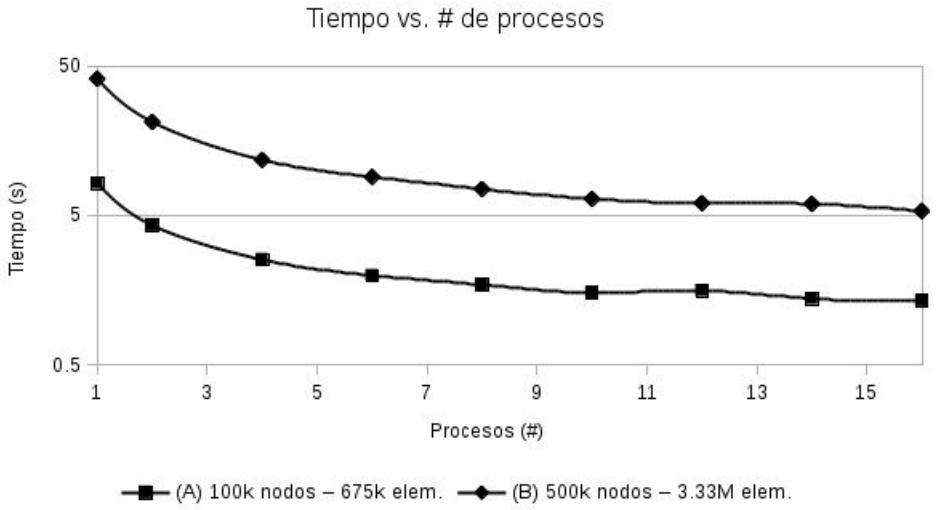

Figura 7.16: Tiempos de ejecución para la versión de memoria distribuida en función del número de procesos utilizados.

Estrategia híbrida

Las figuras 7.17 y 7.18 resumen los resultados para la versión híbrida del algoritmo, distribuyen un proceso por nodo (en el mismo cluster que los resultados de la sección anterior, pero utilizando ahora hasta 6 nodos), y permitiendo al algoritmo administrar más de un thread por proceso. Si se comparan tiempos individuales de configuraciones similares de la sección anterior, se puede observar que el mecanismo híbrido efectivamente mejora la eficiencia y reduce los tiempos. Por ejemplo, en el caso mayor de la sección anterior, se lanzan 16 hilos de trabajo en 4 nodos multicore de 4 núcleos cada uno. Es decir, 4 procesos por procesador, 1 proceso por núcleo. En esta sección, una de las mediciones realizadas también involucra 16 hilos de trabajo, pero utilizando 4 hilos de trabajo por procesador. Es decir mediante solo 4 procesos (uno por procesador), con 4 threads cada uno (1 thread por núcleo). De esta forma, el algoritmo puede evaluar mejor a qué hilo

delegar un trabajo (priorizando las delegaciones que eviten la comunicación mediante la red), y cuando se delega un trabajo de un hilo a otro del mismo proceso, no es necesario duplicar los datos, ya que utilizan internamente el mecanismo de memoria compartida. El tiempo de mallado se reduce de 5.35s a 4.72s, aumentando la eficiencia paralela desde 47.92 % a 54.33 %, y el speed-up desde 7.67 a 8.69. Es decir, con una ganancia en tiempos del 11 %, demostrando la ventaja de utilizar dicho mecanismo.

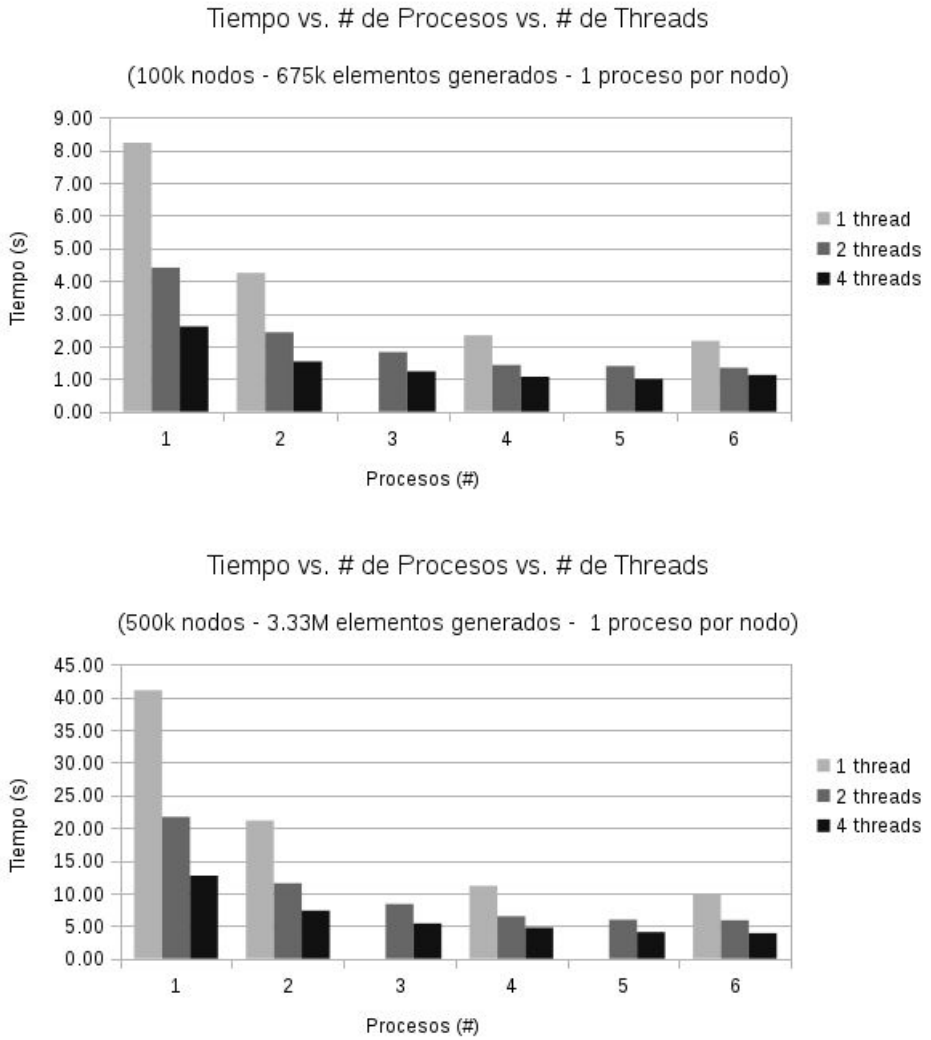

Figura 7.17: Tiempos de ejecución para la versión híbrida en función del número de procesos y threads utilizados para dos mallas de diferentes tamaños.

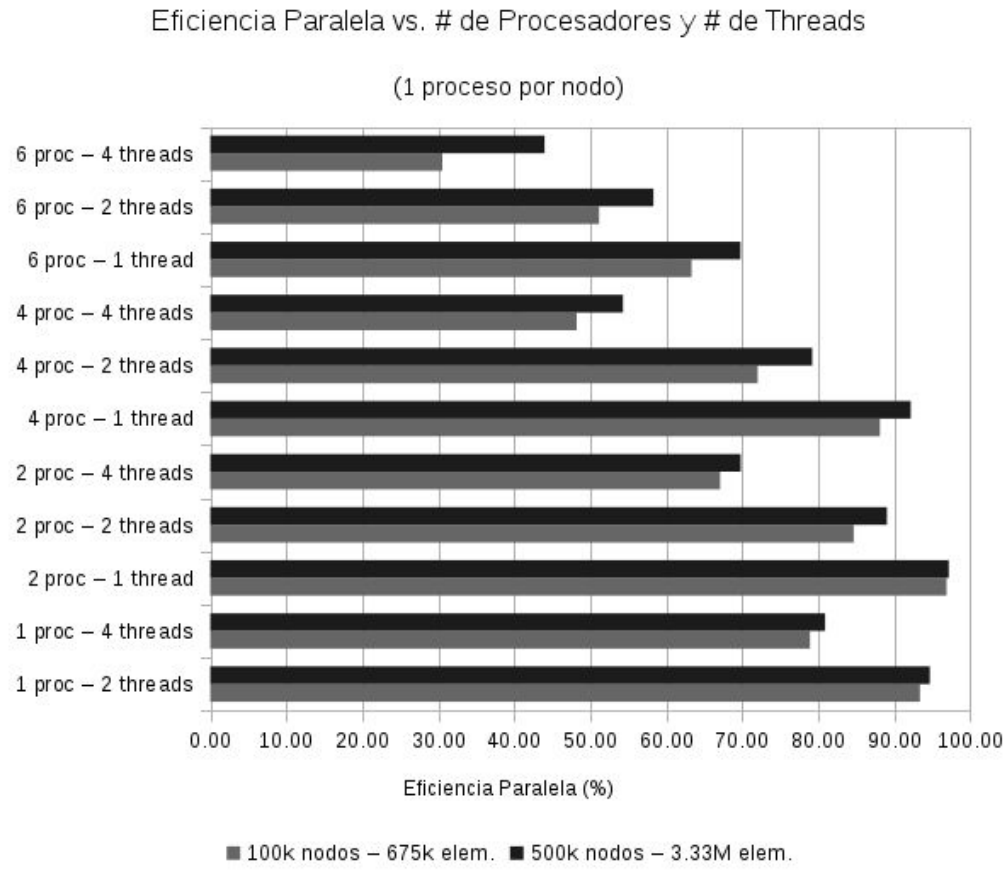

Figura 7.18: Eficiencia paralela para los casos de la figura 7.17.

La figura 7.19 muestra el resultado de utilizar la estrategia híbrida en la arquitectura de memoria compartida NUMA de 2 procesadores con 12 núcleos cada una presentado en la sección 7.2.2. Se confirma la conveniencia de utilizar las técnicas de memoria local en arquitecturas NUMA para reducir la necesidad de acceso a memorias no locales de cada procesador. Se presentan diferentes distribuciones entre procesos para un mismo número total de hilos. En todos los casos, el tiempo de ejecución mejora ampliamente al que se obtiene utilizando solamente la estrategia para memoria compartida. Las mejoras varían en reducciones del tiempo de entre 26 % y 40 %. En el mejor caso, el speed-up aumenta de 5.1 a 8.45, logrando una taza de generación de elementos de 39.1 millones de elementos por segundo.

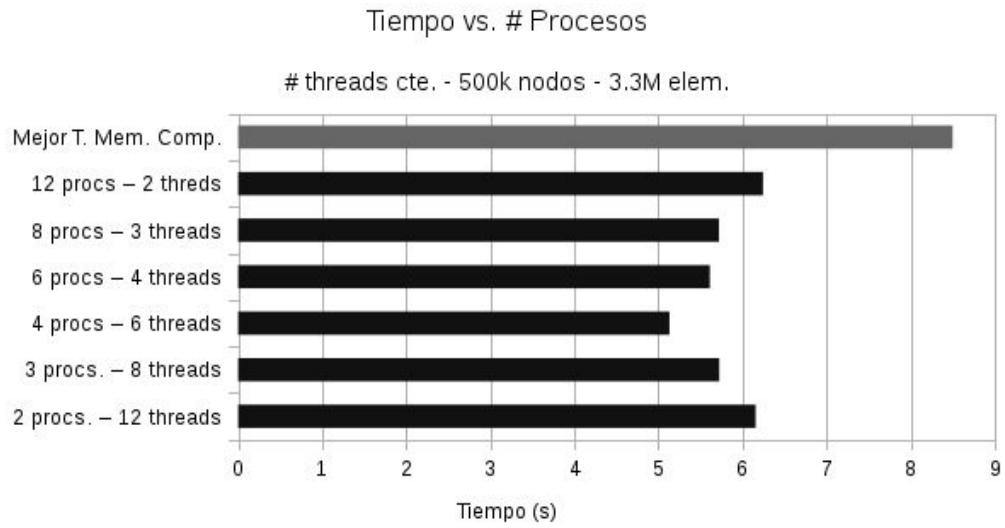

Figura 7.19: Tiempos de ejecución para el ejemplo de mayor tamaño utilizando la estrategia de paralelización híbrida en una arquitectura de memoria compartida NUMA. Se resalta a modo de referencia el mejor tiempo obtenido para el mismo caso con la estrategia de memoria compartida.

7.3.3. Limitaciones y potenciales mejoras

Para proponer potenciales mejoras del algoritmo, se deben describir primero sus falencias, considerando en este caso los factores directamente relacionados con la eficiencia de la paralelización. La eficiencia paralela puede resultar baja por diferentes motivos. Uno de ellos es debido a la sobrecarga del mecanismo de paralelización. En el algoritmo propuesto, la única tarea adicional (en comparación con la versión original no paralela) de peso es la comunicación de un trabajo desde un procesador a otro. Sin embargo, las mediciones realizadas indican que el tiempo de comunicación no es determinante en la estrategia propuesta. A modo de ejemplo, se puede comparar los tiempos para un mismo número de hilos de trabajo, pero distribuyéndolos en diferentes cantidades de procesadores multi-núcleo. Si los tiempos de comunicación fueran determinantes, los tiempos totales deberían variar sensiblemente para las distintas distribuciones. Esto se debe a que la comunicación entre hilos que se ejecutan dentro de un mismo procesador (implica simplemente copiar datos de una posición a otra de la memoria RAM) es varios ordenes de magnitud más rápida que la comunicación entre nodos diferentes de un cluster (que implica transmisión a través de la red). Los resultados obtenidos no muestran grandes diferencias en estos casos.

Otro problema importante, puede ser la contención en el acceso a recursos compartidos. En este algoritmo, el principal recurso es la memoria, y su acceso es efectivamente un problema solo en la versión para memoria compartida.

El tercer motivo de interés, es el desbalance de la carga entre los procesadores. Ya se demostró en este capítulo que la estrategia de gestión del contenedor de trabajos garantiza un correcto balanceo durante la mayor parte del tiempo de ejecución (figura 7.2). Sin embargo, el principal problema radica en el tiempo y procesamiento necesarios al comienzo del mismo para generar suficientes trabajos para todos los procesadores y núcleos disponibles. Si se desprecian los tiempos de comunicación y se asume que todos los trabajos de un mismo nivel del árbol de trabajos requieren el mismo tiempo de cómputo (simplificaciones que no se alejan mucho de los casos más usuales), denominando t_i al tiempo que toma resolver un trabajo del nivel i del árbol de trabajos para la generación de un malla, el algoritmo tarda $\sum_{i=0}^{log_2 Np} t_i$ (siendo Np el número de núcleos disponibles) en llegar al punto a partir del cual ya no habrá procesadores ociosos. Este tiempo sí es representativo en el total (en general es de un orden menos que el tiempo total), y podría potencialmente reducir en una constante considerable la diferencia entre el speed-up ideal y el logrado. Más aún, en la mayoría de las geometrías a mallar (garantizado si es convexa) cada nivel del árbol de trabajos tendrá trabajos menos costosos que el nivel anterior. Esto se debe a que cada trabajo particiona el convex-hull de su dominio en 2 subdominios estrictamente menores, abarcando cada uno idealmente la mitad de dicho dominio. Entonces, generalmente, durante la resolución de los más costosos $log_2 Np$ niveles la potencia de cálculo del sistema está siendo subaprovechada.

Los datos de entrada utilizados para las mediciones de tiempos de la figura 7.2, corresponden a un conjunto de puntos cuyo AABB se asemeja a un cubo (similares al ejemplo B de la figura 7.1). Como se mencionó anteriormente, en estos casos, donde las tres dimensiones son iguales o muy similares, no existe una elección para la orientación del primer plano divisor que resulte mejor que las otras posibles. En un problema con la misma cantidad de nodos o elementos, pero donde una de las dimensiones del AABB sea mucho menor que las otras (como el ejemplo A de la figura 7.1), el algoritmo utilizará un plano paralelo a esa dimensión, para de esta forma minimizar el área de intersección entre el mismo y el dominio de la malla. Esto generará un primer trabajo mucho menor (en cuanto a la cantidad de elementos que debe generar), logrando así que la carga se balancee más rápidamente y el coeficiente de utilización de los procesadores se incremente. Cuanto mayor sea la diferencia entre la mayor y la menor dimensión del AABB inicial, más evidente será este efecto. Es decir, que la eficiencia paralela y los speed-ups

que se obtienen pueden variar de acuerdo a la geometría de entrada. Los casos de ejemplo que se analizaron en este capítulo constituyen algunos de los peores escenarios posibles desde el punto de vista del balanceo de la carga al comienzo del proceso. Se utilizaron estos casos para evaluar los resultados por representar geometrías usuales en la práctica para las aplicaciones propuestas en la motivación de este trabajo, y para evitar obtener resultados poco representativos.

Dadas las restricciones introducidas por las perturbaciones para eliminar las ambigüedades del criterio Delaunay en la versión sin frontera impuesta, y las introducidas por la propia frontera en la versión constrained, un trabajo de un nivel j no puede resolverse de la forma propuesta antes que otro de un nivel i (con $i < j$), porque desconoce sus restricciones. Si el trabajo de nivel j comienza a resolverse antes de que finalice el del nivel i, y por ende, ignorando las restricciones que ese nivel i genera, la fronteras entre trabajos pierden su garantía de compatibilidad, generando así un nuevo problema. Una solución podría ser introducir una etapa de *merging* al final del proceso para remediar estas incompatibilidades, pero es una solución que deliberadamente se buscó evitar en este trabajo, ya que puede llegar a ser un proceso muy costoso y anular así el beneficio de mejorar el balance de la carga.

Otra posible mejora consiste en reproducir el trabajo t_0 en todos los procesadores, cada t_1 en cada mitad de ellos, y así sucesivamente. De esta forma todos los procesadores estarán ocupados todo el tiempo, aunque realizando trabajo redundante. Realizar trabajo redundante puede ser útil, como lo es en este caso, para reducir la necesidad de comunicación. Es decir, se invierte el poder de cómputo excedente en esas primeras etapas en reducir la necesidad de comunicación. Esta solución efectivamente reduce los tiempos, pero dado que lleva a que un mismo mallado en paralelo requiera muchas más operaciones que en serie, la reducción de tiempos es en realidad muy baja, y el potencial para acercarse al speed-up ideal muy limitado.

Finalmente, las alternativas más prometedoras se basan en dos ideas simples: (a) solapar los dominios de los subtrabajos, y (b) requerir garantías (o imponer restricciones) adicionales respecto a tamaño y forma, u otras propiedades relacionadas de los elementos. En general el impacto de las perturbaciones o de la presencia de un elemento no Delaunay en una frontera impuesta es local. Es decir, que la resolución de la ambigüedad, o la colocación de elementos no Delaunay, se reduce a un volumen alrededor del nodo o la cara en cuestión de unos pocos elementos de radio (distancia topológica). Por esto, permitir diferentes áreas de solapamiento entre subdominios reduce considerablemente el trabajo en una posterior etapa de merging. Los elemen-

tos generados sobre un plano divisor tenderán a coincidir entre los trabajos a ambos lados de dicha frontera, si cada trabajo considera nodos cercanos del otro trabajo para generarlos (ya que tendrá más posibilidades de captar las restricciones adicionales importantes). Sin embargo, la condición Delaunay por sí misma no puede ofrecer garantías a priori acerca del volumen que es necesario inspeccionar para construir un elemento. En un mecanismo de avance como el que aquí se describe, no se puede acotar solo con este criterio, antes de comenzar, cuan lejos estará el nodo ganador para una cara base, de modo que permita definir áreas de solapamiento útiles para la división de trabajos que ofrezcan garantías de unicidad. En una construcción incremental, por ejemplo, no se puede acotar el tamaño de la cavidad a reemplazar.

En una aplicación de ingeniería real, en cambio, en muchos casos es posible aceptar el error de asumir una determinada cota, o deducir la cota de otras propiedades del problema. Por ejemplo, si los nodos representan partículas, en general los métodos de cálculos basados en partículas intentan mantener la concentración de partículas entre niveles preestablecidos, insertando o eliminando partículas según corresponda. Algunos de estos mecanismos pueden proveer directa o indirectamente cotas útiles para el mallado. Algunos métodos recientes aprovechan estas particularidades para mejorar la eficiencia de la paralelización de la etapa de mallado (por ejemplo [72]). Además, puede ocurrir que la naturaleza de estos mecanismos y los problemas que modelan permita flexibilixar el problema del mallado y tornar factibles soluciones alternativas. Muchos problemas de ambigüedad, merging, o incluso de factibilidad de una frontera, podrían resolverse mucho más fácil y rápidamente insertando o eliminando estratégicamente un nodo. Si el sistema en el cual el mallador se utiliza contempla la generación y eliminación de nodos (por ejemplo partículas), entonces el mallador podría permitirse utilizar estas soluciones alternativas como último recurso para resolver sus problemas.

Capítulo 8

Conclusiones

Como resultado de este trabajo se generaron dos nuevos algoritmos de triangulación/tetraedrización tanto para nubes de puntos sin frontera (Delaunay clásico), como para conjuntos de puntos con mallas de frontera preimpuestas (Delaunay constrained). Como parte de los desarrollos originales se resolvieron en ambos casos los problemas relacionados tanto a los errores numéricos y a las ambigüedades del criterio Delaunay, como al desafío de preservar la compatibilidad con una frontera impuesta. También se resolvieron en el proceso numerosos problemas de implementación no abordados en detalle previamente en la literatura disponible, se verificaron empíricamente los resultados mediante benchmarks, y se documentaron todos los detalles relevantes, las alternativas posibles y las justificaciones correspondientes para cada elección.

Las soluciones desarrolladas no exigen cambios en la definición del problema, sino que respetan todas las restricciones planteadas al comienzo, y evitan completamente la necesidad de agregar, eliminar o mover nodos, o de realizar cambios en la malla de frontera impuesta.

Ambos algoritmos resultantes preservan las propiedades del algoritmo DeWall que los hacen aptos para su paralelización, y todas las modificaciones introducidas mantienen el orden de los tiempos de ejecución. Como consecuencia, se ha logrado paralelizar ambos métodos tanto para arquitectura de memoria compartida, como para arquitecturas de memoria distribuida y aún para arquitecturas híbridas (las más usuales en un cluster actual). Los resultados presentados en el capítulo 7 muestran niveles de eficiencia paralelas competitivos, y un alto porcentaje de aprovechamiento de los recursos computacionales disponibles.

Más aún, las implementaciones desarrolladas permiten variar en cada trabajo la superficie que se utiliza como divisoria (en los ejemplos presentados, un plano medio), posibilitando la reutilización de divisiones de dominios generadas externamente por el contexto en el cual se ejecuta el algoritmo de tetraedrización (por ejemplo, por la división de dominio utilizada en una etapa de cálculo sobre la malla previa) para minimizar la necesidad de comunicación inicial.

La versión constrained del algoritmo integra nuevas estructuras de datos y estrategias adaptadas de otros métodos para lograr respetar la frontera sin requerir una etapa de preproceso. Pero también permite, mediante la misma solución, reducir o en muchos casos eliminar por completo la presencia de elementos de muy baja calidad como slivers, muy comunes en generadores Delaunay.

Finalmente, se identificaron las principales falencias de la solución propuesta, y potenciales caminos para resolver o minimizar la más importante de ellas en trabajos futuros.

8.1. Trabajos Futuros

Como trabajos futuros, del análisis de la sección 7 se desprende la posibilidad de mejorar la eficiencia paralela tratando de aprovechar todos los procesadores disponibles desde el tiempo 0.

Para ello, en arquitecturas de memoria compartida se propone reemplazar la estructura de datos utilizada para almacenar el frente de avance por una estructura lock-free. Este cambio facilitaría el aprovechamiento de los núcleos ociosos para resolver en conjunto un único trabajo. Es decir, refinar el nivel de paralelismo del algoritmo para paralelizar las tareas dentro de un subtrabajo (diferentes hilos utilizando diferentes caras base de un mismo frente), pero utilizando estructuras lock-free para evitar que la contención adicional que se genere anule las ventajas de este nivel de paralelización adicional.

Por otro lado, en arquitecturas de memoria distribuida, se propone experimentar con técnicas basadas en ejecución especulativa y superposición de dominios, eventualmente asumiendo restricciones adicionales al criterio Delaunay (pero razonables en el contexto de aplicación) para obtener las garantías que faltan en pos de evitar que estos nuevos mecanismos introduzcan mayor comunicación, o requieran demasiado procesamiento en una posterior y nueva etapa de merging.

Para tratar de mejorar el rendimiento del algoritmo base (serie) y no solo su eficiencia paralela, también se puede continuar analizando alternativas más complejas para el criterio de eliminación de elementos en la versión con frontera impuesta, dado que es esta la elección que resultó determinante para el tiempo de trabajo cuando se llega a un frente irresoluble. Actualmente, si bien la probabilidad de ocurrencia de este problema es muy baja, el tiempo de resolución de dicho problema es suficientemente alto como para generar un desvío considerable en la distribución de tiempos de generación para distintos conjuntos de datos de entrada de tamaños similares. Más aún, en determinados escenarios de aplicación existe la posibilidad de agregar o eliminar nodos, y esta flexibilización de los requisitos podría aprovecharse para resolver más rápidamente este problema en muchos casos sin deshacer ningún elemento.

Por último, dadas las altas tazas de generación de elementos logradas, y las propiedades geométricas del criterio Delaunay, se puede explorar la posibilidad de modificar el algoritmo para que incluya la generación de los puntos interiores, y resolver así un problema diferente y más habitual: la generación de una malla de interior (tanto nodos como conectividades) a partir de solo una malla de frontera. Como ejemplo, el algoritmo podría pregenerar rápidamente potenciales puntos en el interior del dominio, con la densidad y distribución deseada, utilizando alguna estructura de ordenamiento espacial auxiliar, y luego intentar conectarlos, aprovechando al generar los elementos la nueva posibilidad de moverlos o descartarlos para resolver más fácilmente los problemas y también para optimizar en el proceso criterios de calidad de forma (por estar directamente relacionados a los criterios de selección de nodos ganadores para caras base) a medida que avanza.

Apéndice A

Esta sección contiene descripciones relativamente detalladas de problemas de implementación que pueden resultar de interés aún cuando no sean esenciales para comprender (algunos sí para reproducir) los métodos de mallado y triangulación descritos anteriormente.

A.1. Implementación de contenedores genéricos y thread-safe en C++

Para el desarrollo de las clases MallaCH (sección 5.3) y MallaFull (sección 6.3) se utilizaron diferentes contenedores genéricos, tanto para la estructura base de la malla (listas de puntos de entrada, elementos de frontera, y elementos generados) como para las estructuras auxiliares específicas de cada algoritmo de mallado (por ejemplo: para almacenar el conjunto de caras de un trabajo en la versión con frontera impuesta).

Las clases MallaCH y MallaFull heredan de especializaciones de la clase MallaBase, clase que implementa los métodos que son independientes del proceso de mallado (por ejemplo, métodos para cargar los datos de entrada de un archivo y para guardar luego la malla resultante en otro, o estructuras auxiliares para generar información de depuración). Esta es una clase genérica, donde el tipo de contenedor utilizado para cada conjunto de datos es genérico, pudiendo elegir en cada caso entre los múltiples contenedores desarrollados. Todos ellos presentan una interfaz básica común (métodos para agregar y eliminar elementos, mecanismos para recorrerlos, etc). De esta forma, se independizan los algoritmos más básicos y generales de la malla del tipo de estructura de datos que le da soporte, permitiendo así elegir luego la más conveniente para cada algoritmo de mallado (para cada clase heredada), o probar variar de forma casi transparente la estructura utilizada para deter-

minar cual conviene mediante benchmarks. De igual forma, los contenedores son genéricos, y al especializarse contendrán clases que representen nodos/elementos que cumplirán con una interfaz básica de nodo/elemento común a todas las versiones de Malla (a todos los algoritmos de mallado).

```
template< class ContenedorNodos,
          class ContenedorCaras,
          class ContenedorElementos >
class MallaBase {
public:
  // contenedores genericos
  ContenedorNodos n;
  ContenedorCaras c;
  ContenedorElementos e;

  // nombres genericos para tipos especializados
  using Nodo = typename ContenedorNodos::ElementType;
  using Cara = typename ContenedorCaras::ElementType;
  using Elemento = typename ContenedorElementos::ElementType;

  // metodo de ejemplo... todos los contenedores tienen un
  // metodo "Cantidad" y pueden recorrerse con range-base for loops
  bool Save(const char fname[]) const {
    FILE *fout=fopen(fname,"w"); if (!fout) return false;
    fprintf(fout,"%i Nodes x y z base=0\n",n.Cantidad());
    for (const auto &x:n)
      fprintf(fout,"%f %f %f\n",x[0],x[1],x[2]);
    unsigned int cn=c.Cantidad();
    if (cn) {
      fprintf(fout,"%i Triangles\n",int(cn));
      for(const auto &x:c)
        fprintf(fout,"%i %i %i\n",x[0],x[1],x[2]);
    }
    unsigned int en=e.Cantidad();
    if (en) {
      fprintf(fout,"%i Tetrahedrals\n",int(en));
      for(const auto &x:e)
        fprintf(fout,"%i %i %i %i\n",x[0],x[1],x[2],x[3]);
    }
    fclose(fout);
    return true;
  }

  /*...otros atributos y metodos...*/

};
```

Las estructuras de datos siguen una filosofía de diseño similar a la de los contenedores de la biblioteca de plantillas estándar STL, lo cual se observa en tres aspectos fundamentales. En primer lugar, los métodos para las operaciones comunes tienen nombres consistentes en las diferentes estructuras, de modo que un código genérico que las utilice pueda ser especializado (en tiempo de compilación) sin cambios para cada una de ellas. En segundo lugar, para recorrerlos completamente, independientemente de cómo organicen

por dentro los datos contenidos, se utilizan clases proxy muy similares a los iteradores de los contenedores STL (en algunos casos, hasta utilizando exactamente la misma interfaz para poder aprovechar funcionalidades estándar como range-based for loops). Y por último, en algunos casos se independiza la gestión de memoria mediante clases similares a lo que en la biblioteca STL se conoce como allocators, para poder analizar diferentes estrategias de gestión de memoria en situaciones donde la creación y destrucción de elementos se realiza con mucha frecuencia.

```
1  template< class T,
2            template<class U> class Allocator = RegularAllocator >
3  class ListaDoble {
4
5  protected:
6      Nodo *primero, *ultimo;
7    /*...otros atributos...*/
8
9  public:
10   // nombre generico para el tipo de elemento especializado
11   using ElementType = T;
12
13   // clase iterador, funcionalidad e interface simil STL
14   class Iterator { /*...implementacion...*/ };
15
16   // metodo ejemplo donde se utiliza el allocator
17   void AgregarAlPrincipio(const T &t) {
18     primero = Allocator<Nodo>::New(t,NULL,primero);
19     if (primero->siguiente)
20       primero->siguiente->anterior=primero;
21     else ultimo=primero;
22   }
23
24   /*...otros metodos...*/
25
26   // interfaz necesaria para range-based for loops
27   Iterator begin() { return Iterator(/*...*/); }
28   Iterator end() { return Iterator(/*...*/); }
29
30 };
```

El uso de métodos de la clase especial Allocator en lugar de los operadores new y delete permite cambiar fácilmente la forma en que se reserva y libera memoria para los contenedores. Esto es de especial interés para contenedores basados en listas enlazadas, ya que estos se utilizarán cuando sean muy frecuentes las operaciones de inserción y eliminación de elementos sobre los mismos, y en estos casos cada operación requiera la creación o destrucción de una celda. La clase RegularAllocator que menciona en el ejemplo anterior, equivale a utilizar los operadores new y delete, efectivamente creando y destruyendo celdas en cada operación. Es de esperar que el optimizador del compilador compile de forma inline las llamadas a los métodos New y Delete, de forma que no exista sobrecarga alguna si se compara con el uso directo de

los operadores (por ejemplo, con gcc esto ocurre al utilizar el argumento de compilación -O3). Sin embargo, la ventaja de separar estas operaciones de la implementación del contenedor mismo permite desarrollar allocators más avanzados. En este trabajo, por ejemplo, se desarrolló una clase RecyclingAllocator que mantiene una lista de celdas eliminadas (mediante llamadas al Delete), para reutilizar su memoria luego cuando se intente crear nuevas celdas (llamadas a New). Utilizando este allocator para las listas de caras de los trabajos pendientes del algoritmo de malla, por ejemplo, se puede reducir la cantidad de operaciones de reserva y liberación de memoria a la mitad. Los resultados de utilizar este mecanismo para la aplicación particular descrita en esta tesis se presentaron en la figura 7.5. Más aún, en las versiones paralelas del algoritmo, este allocator utiliza un pool diferente para cada hilo de ejecución, evitando así la necesidad de sincronizar su acceso, y reduciendo el impacto de las potenciales sincronizaciones internas de los métodos new y delete.

```
1  template<class T>
2  struct RecyclingAllocator {
3    static T *first;
4    template<class... Args>
5    // se utiliza la memoria conceptualmente liberada como puntero
6    // para mantener una lista simpl. enlazada de celdas libres
7    static T *New(Args... args) {
8      static_assert(sizeof(T)>sizeof(T*),"sizeof(T) too small");
9      if (!first) return new T(args...);
10     T *ret = first;
11     first = *(reinterpret_cast<T**>(first));
12     return new (ret) T(args...); // placement new
13   }
14   static void Delete(T *n) {
15     n->~T();
16     *(reinterpret_cast<T**>(&n)) = first;
17     first = n;
18   }
19 };
20 template<class T> T *RecyclingAllocator<T>::first=nullptr;
```

Todos los contenedores genéricos implementados se basan en dos tipos básicos, el vector lineal representado por la clase Vector, y la lista doblemente enlazada representada por la clase ListaDoble (cuyo funcionamiento se describe y discute en la sección 4.1). Estas clases implementan las operaciones básicas sobre estas estructuras de datos sin realizar consideraciones relacionadas al paralelismo. Es decir, sus métodos no son thread-safe. Por ello, garantizar la ausencia de data-races será responsabilidad del código cliente para estas clases. Sin embargo, a partir de estas clases, se pueden generar fácilmente por herencia versiones thread-safe que agreguen en pocas lineas las condiciones necesarias para garantizar la correcta ejecución cuando

son compartidas por más de un hilo. Para ello, se desarrollaron adaptadores genéricos, es decir, que pueden extender cualquiera de los dos contenedores base, ya que sus interfaces son compatibles. Estos adaptadores respetan la misma interfaz básica que los contenedores en los cuales se basan (mismos métodos), de forma que pueden utilizarse en una malla en lugar de dichos contenedores sin que esto requiera ningún cambio en los algoritmos de la malla que los utilizan.

```
1  template<typename Container>
2  class ThreadSafeContainer : public Container {
3    std::mutex the_mutex;
4  public:
5    using ElementType = typename Container::ElementType;
6    // metodo ejemplo, encapsula el Agregar original
7    // utilizando un mutex para evitar data-races
8    void Agregar(const ElementType &e) {
9      std::lock_guard<std::mutex> lck(the_mutex);
10     count++; Container::Agregar(e);
11   }
12   /*...otros metodos/wrappers...*/
13 };
```

Pero hay otra ventaja en el uso de adaptadores para los casos de algoritmos paralelos, y es la posibilidad de realizar en ellos ciertas optimizaciones de forma transparente para el mallador. Por ejemplo, se puede pensar que si varios hilos están trabajando sobre una misma lista de elementos, se generarán muchos bloqueos al intentar acceder a la misma, lo cual se traduce en la presencia de hilos durmiendo a la espera de que se libere un recurso (el mutex). Como resultado se puede esperar una pérdida de performance del proceso completo. Si un adaptador recibe además algún tipo de identificación del hilo que lo utiliza, puede generar contenedores locales (a cada hilo, thread-local) en los cuales guardar los elementos generados durante el trabajo del hilo sin necesidad de utilizar mecanismos de sincronización, y transferirlos en una sola operación sincronizada luego de finalizado el trabajo. De esta forma, se gana en eficiencia porque se evita no solo bloquear un hilo, sino también comprobar el estado del mutex, y porque además los contenedores thread-local podrían hacer mejor uso de la memoria caché de cada procesador (ignorando o combatiendo los efectos conocidos como *false-sharing*[1]).

```
1  template<typename Container, int MAX_THREADS>
2  class ThreadSafeCachedContainer:public Container  {
3    std::mutex the_mutex;
4    // contenedores locales, uno por hilo
```

[1]En el código del ejemplo "__attribute__((aligned(1024))" es un atributo para indicarle al compilador que debe separar los elementos del arreglo de forma tal que sus direcciones de comienzo sean múltiplo de 64 bytes (tamaño más usual para una linea de caché L1) para evitar este efecto.

```
5    Container local_container[MAX_THREADS] __align64__;
6
7  public:
8    using ElementType = typename Container::ElementType;
9
10   // metodo ejemplo, para agregar al contenedor local
11   void Agregar(const ElementType &e, int thread_id) {
12     local_container[thread_id].Agregar(e); count++;
13   }
14   // metodo ejemplo, para transferir todo al contenedor global
15   void Commit(int thread_id) {
16     the_mutex.lock();
17     Container::MoverOCopiar(local_container[thread_id]);
18     the_mutex.unlock();
19     local_container[thread_id].Vaciar();
20   }
21   /*...otros metodos/wrappers...*/
22
23 };
```

Finalmente, un adaptador podría ser utilizado para almacenar elementos en un contenedor que no es de su propiedad, sino externo, mediante un adaptador que de forma transparente redireccione las operaciones a un contenedor fuera del mismo. O bien, un contenedor podría no tener atributos (solo métodos vacíos para cumplir con la interfaz básica de un contenedor genérico) de forma que pueda ser utilizado cuando una especialización de malla pueda prescindir de uno de los conjuntos de datos (por ejemplo, el de los elementos de frontera en MallaCH) para evitar por completo que tenga impacto alguno ya sea en tiempo o en memoria.

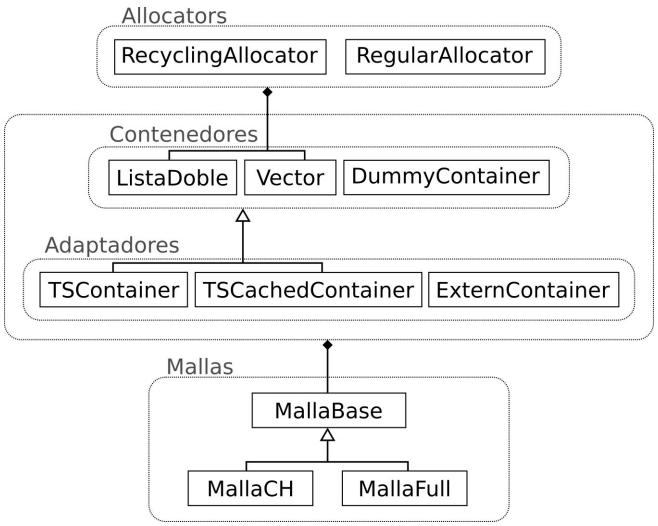

Figura A.1: Jerarquía de clases asociada al mecanismo de contenedores genéricos implementado.

A modo de ejemplo, la clase MallaCH (para la versión sin frontera paralelizada en una arquitectura de memoria compartida) sería:

```
1  class NodoCH : public Punto { ... };
2  class ElementoCH : public Elemento { ... };
3
4  class MallaCH : public MallaBase <
5    // para los nodos se utiliza un Vector, sin sincronizacion
6    Vector<NodoCH>,
7    // no se requiere malla de frontera
8    DummyContainer<Cara>,
9    // para los elementos se puede utilizar lista o vector,
10   // pero mediante un adaptador para garantizar la correcta
11   // sincronizacion del acceso por multiples hilos
12   ThreadSafeContainer< Vector<ElementoCH> > >
13 {
14     ...
15 };
```

A.2. Implementación eficiente de una cola de trabajos thread-safe con prioridades en C++11

Una cola es una estructura de datos que contiene un conjunto ordenado de elementos y opera en modo FIFO (First In First Out). Esto significa que en una cola se insertan y retiran los elementos en un mismo orden. A diferencia de un vector o una lista, al insertar o eliminar en una cola, no se puede elegir en qué posición de la secuencia de datos que contiene se lo hace. Conceptualmente, se puede pensar que los elementos siempre se insertan en un extremo y se consulta o eliminan en el otro. Es común que la implementación de una cola se base en la de una lista enlazada. De hecho la clase std::queue de C++ suele ser en realidad un adaptador para la clase std::list. Esto es, la cola se obtiene por herencia a partir de la lista, exponiendo parte de su funcionalidad y ocultando el resto (mediante herencia privada y métodos wrappers).

Una cola con prioridad es una cola en donde se pueden insertar elementos con prioridades asociadas, de forma que al retirar un elemento la cola retorne el elemento de mayor prioridad. Si hay más de uno con la máxima prioridad, se retorna el primero en haber ingresado (FIFO). Esto implica que o bien el método para retirar un elemento de la cola debe realizar una búsqueda para encontrar cual es la máxima prioridad disponible, o bien el método para insertar un elemento en la cola debe insertar de forma ordenada para mantener internamente la secuencia de trabajos por orden de prioridad. El

costo de buscar y mantener ordenado varía según la estructura que se utilice para dar soporte a la cola con prioridad, pero en general no será $O(1)$ como en la cola simple. En general las colas de prioridad se implementan usando como soporte heaps. Un heap es un caso particular de árbol binario que por sus propiedades puede almacenarse en un vector, haciendo que la inserción y recuperación de los datos (suponiendo que no se deba realocar memoria al operar sobre el vector) sean $O(\log n)$.

En este trabajo en particular, se utilizó una cola con prioridad para almacenar los trabajos pendientes de realizar en un proceso de mallado. La prioridad de cada trabajo en la cola estaba directamente determinada por el tipo del mismo. Además, los tipos posibles eran muy pocos (menos de 10, a veces solo 1). Por ello, en lugar de utilizar un heap como base, se utilizaron múltiples sub-colas simples para almacenar los trabajos, una por cada tipo posible. Esto tiene como ventaja adicional que la estructura que representa un trabajo no debe ser lo suficientemente general como para abarcar todos los tipos posibles (como ocurriría con una cola que utilice internamente un único contenedor). Así, la estructura que representa un trabajo se modeló con un functor (una clase que tiene una sobrecarga del operador () para que pueda ser utilizada como una función), permitiendo colocar en el mismo tanto código ejecutable (métodos) como datos (atributos) de forma arbitraria.

```
1  template <class TMalla, class TInfoTrabajo>
2  class FunctorTrabajo {
3    TMalla *malla;
4    TInfoTrabajo info;
5  public:
6    // al construir el functor se recibe la malla sobre la que
7    // operara y los argumentos necesarios, que seran guardados
8    // en la clase TInfoTrabajo (redireccionados a su constructor)
9    template<class... ArgsTrabajo>
10   FunctorTrabajo(TMalla *pmalla, const ArgsTrabajo... args)
11     : malla(pmalla), info(args...) { }
12   // este es el trabajo que realiza este functor
13   void operator()(int thread_id) {
14     malla->Dividir(info,thread_id);
15   }
16 };
```

Además, esta cola de trabajos difiere de una cola de propósito general en que no tiene un método que retorna un trabajo contenido en la misma, sino que tiene un método que lo ejecuta. De esta forma, es más simple colocar en esta clase la lógica que, utilizando un mutex, sincroniza los acceso a la misma para que actúe de modo seguro si una instancia se utiliza desde más de un hilo.

```
1  // clase que representa una cola para un unico
2  // tipo de trabajo, y que se auto-gestiona de modo
```

```
3  // thread-safe mediante un mutex propio
4  template <class TFunctorTrabajo>
5  class ColaThreadSafe {
6    // contenedor con trabajos pendientes
7    std::queue<TFunctorTrabajo> cola;
8    // mecanismo de sincronizacion para accesos a cola
9    std::mutex un_mutex;
10 public:
11   // el using permite a partir de una instancia conocer el
12   // tipo de functor con el que fue especializada
13   using TipoTrabajo = TFunctorTrabajo;
14   // procesa un trabajo de la cola en el hilo nro thread_id
15   bool Procesar(int thread_id) {
16     // el mutex evita procesar dos veces el mismo trabajo
17     std::unique_lock<std::mutex> lock(un_mutex);
18     // si no hay trabajo, salir con falso
19     if (cola.empty()) return false;
20     // si hay trabajo sacarlo de la cola
21     auto un_trabajo = cola.front(); cola.pop();
22     // liberar la cola para otros accesos
23     lock.unlock();
24     // ejecutar el trabajo y salir con verdadero
25     un_trabajo(thread_id);
26     return true;
27   }
28   // agrega un trabajo pendiente a la cola
29   void Agregar(const TFunctorTrabajo &un_trabajo) {
30     std::lock_guard<std::mutex> lock(un_mutex);
31     cola.push(un_trabajo);
32   }
33 };
```

Para implementar la cola de forma eficiente en C++ y de manera que pueda reutilizarse en distintos algoritmos de mallado o distintas versiones de un mismo algoritmo (es decir, variando los tipos de trabajo), se utilizaron técnicas de programación genérica. En C++03, la programación genérica (templates) permite independizar una implementación de uno o más (una cantidad fija) tipos de datos. Los denominados *variadic templates*[73][55] agregados a C++ en el estándar C++11 permiten además que la cantidad de tipos de datos que recibe el template sea también variable y arbitraria. Las clases basadas en variadic templates se definen generalmente mediante herencia recursiva. Es decir, que un variadic template de clase especializada con N argumentos hereda de la misma clase especializada con N-1 argumentos, utilizando una especialización explícita para el caso base, que regularmente será 0 o 1 argumento genérico. De esta manera, el uso de templates en C++ se asemeja a un lenguaje de programación (dentro de otro) con características propias del paradigma funcional, pero cuyas construcciones serán resueltas en tiempo de compilación por el compilador, y no en tiempo de ejecución por el programa resultante.

```
1  // Clase templatizada con argumentos variables
```

```
2  template<typename... TVariosFunctores> class ColaVA;
3
4  /// Cola con un unico tipo de trabajo
5  //"paso base" en la recursion del variadic template
6  template<typename TFunctorTrabajo>
7  class ColaVA<TFunctorTrabajo> {
8    // cola de trabajos con prioridad 0
9    ColaThreadSafe<TFunctorTrabajo> subcola;
10 public:
11   /// Intenta ejecutar un trabajo de esta cola
12   virtual bool Procesar(int thread_id) {
13     // intentar ejecutar un trabajo
14     return subcola.Procesar(thread_id))
15   }
16   /// Permite agregar un trabajo a la cola de este nivel
17   void Agregar(const TFunctorTrabajo &un_trabajo) {
18     subcola.Agregar(un_trabajo);
19   }
20 };
21
22 // "paso recursivo" en la jerarquia del variadic template
23 template<typename TFunctorTrabajo, typename... TOtrosFunctores>
24 class ColaVA<TFunctorTrabajo,TOtrosFunctores...>
25       : public ColaVA<TOtrosFunctores...>
26 {
27   // cola de trabajos con prioridad!=0
28   ColaThreadSafe<TFunctorTrabajo> subcola;
29 public:
30   /// Intenta ejecutar un trabajo de esta cola en el hilo thread_id
31   virtual bool Procesar(int thread_id) {
32     // intentar ejecutar un trabajo en este nivel
33     if (subcola.Procesar(thread_id)) return true;
34     // o reintentar en los niveles superiores
35     // (de menor prioridad)
36     return ColaVA<TOtrosFunctores...>::Procesar(thread_id);
37   }
38   // hacer visibles las sobrecargas de Agregar
39   // definidas en los demas niveles
40   using ColaVA<TOtrosFunctores...>::Agregar;
41   /// Permite agregar un trabajo a la cola de este nivel
42   void Agregar(const TFunctorTrabajo &un_trabajo) {
43     subcola.Push(un_trabajo);
44   }
45 };
```

Alternativas posibles podrían ser: utilizar funciones de preprocesador, utilizar una única clase para todos los tipos de trabajo, guardar en la cola punteros sin tipo y utilizar conversiones explícitas al extraerlos, o utilizar una jerarquía de clases basada en polimorfismo dinámico para los trabajos y almacenarlos también mediante punteros. Una gran ventaja de utilizar templates radica en que gran parte de la lógica se resuelve en tiempo de compilación. Por ejemplo, con este mecanismo, una cola para 3 tipos de trabajo tendrá 3 niveles de herencia, donde cada uno agrega una cola para un tipo particular, en orden inverso de prioridad. Cada nivel de la herencia implementa un

método para agregar un trabajo a una cola, entonces el método para agregar se resuelve por sobrecarga (varios métodos homónimos con distinto tipo de argumento). Así, al agregar un trabajo en la clase resultante desde un programa cliente, se decide en tiempo de compilación a cual versión del método llamar, resultando en una inserción $O(1)$, sin costo adicional ni en tiempo ni en espacio comparando con una cola simple. Al invocar al método que recupera un trabajo para ejecutarlo, estaríamos invocando un método del último nivel de la jerarquía (el que corresponde a la sub-cola con trabajos de mayor prioridad), y si este no tiene trabajos para ejecutar, invoca al del nivel anterior, y así recursivamente hasta llegar a la base. Por ello, en el peor caso el costo de la invocación será del orden de la cantidad de tipos de trabajo, que es fija y que para nuestra aplicación en particular puede considerarse $O(1)$.

Así vemos cómo una implementación basada en templates provee la máxima eficiencia en cuanto a tiempos de ejecución ya que permite resolver muchas cosas en tiempo de compilación (a diferencia de, por ejemplo, una jerarquía basada en clases abstractas). También evita la necesidad de utilizar punteros simplificando el manejo de memoria y la implementación de las subcolas individuales. Más aún, dado que los templates se especializan en tiempo de compilación, otorgan seguridad de tipos (los errores de tipado son detectados por el compilador) y favorecen la aplicación de optimizaciones de bajo nivel (el compilador, al conocer el tipo exacto de los datos con que trabaja la clase, tiene más información para aplicar optimizaciones al compilar). Una de las optimizaciones de mayor impacto que permite utilizar el polimorfismo estático (y no así el dinámico) es el inlining de funciones. Si se hace un uso exhaustivo de esta cola de trabajos, este detalle puede generar diferencias perceptibles en la ejecución. No es este el caso de las implementaciones desarrolladas en esta tesis, pues las operaciones sobre la cola de trabajo no resultaron determinantes en las ejecuciones. Pero la situación podría variar si se implementan niveles de paralelismo dentro de los subtrabajos (alternativa planteada como posible trabajo futuro) y se utiliza esta cola de trabajo para gestionar todos los niveles.

El mayor inconveniente en el uso exhaustivo de templates en C++ radica en el diagnóstico que el compilador hace de los errores (la complejidad de sus mensajes). Sin embargo, desde la popularización del toolchain llvm-clang, los compilador han mejorado notablemente en este aspecto[74], y se prevee que la introducción de *concepts*[75] en C++ estándar prevista para 2017 (al menos en una versión reducida denominada *concepts lite*) permita reducir sustancialmente el problema. Mientras tanto, algunas técnicas de template-metaprogramming, basadas en el concepto de SFINAE[76] (como la clase

std::enable_if de la biblioteca estándar) pueden suplir estas limitaciones, al menos desde el punto de vista del usuario de la biblioteca.

Finalmente, cabe destacar que en la implementación utilizada el argumento del template no es el tipo de trabajo, sino el tipo de la sub-cola para cada tipo de trabajo (que es a su vez una clase genérica especializada para un tipo de trabajo). De esta manera, la cola de trabajos permite variar fácilmente (cambiando simplemente un argumento en la instanciación) no solo la cantidad y variedad de trabajos que gestiona, sino también los contenedores que utiliza para cada tipo para, por ejemplo, poder elegir entre las estrategias LIFO, FIFO y Mixed presentadas en la sección 7.1.2.

```
1  template<typename... TVariasSubcolas> class ColaVA;
2
3  template<typename TUnaSubcola>
4  class ColaVA<TUnaSubcola> {
5    UnaSubcola subcola;
6  public:
7    virtual bool Procesar(int thread_id) {
8      return subcola.Procesar(thread_id))
9    }
10   void Agregar(const TUnaSubcola::TipoTrabajo &un_trabajo) {
11     subcola.Agregar(un_trabajo);
12   }
13 };
14
15 template<typename TOtraSubcola, typename... TVariasSubcolas>
16 class ColaVA<TOtraSubcola,TVariasSubcolas...>
17 : public ColaVA<TVariasSubcolas...> {
18   TOtraSubcola subcola;
19 public:
20   virtual bool Procesar(int thread_id) {
21     if (subcola.Procesar(thread_id)) return true;
22     return ColaVA<TVariasSubcolas...>::Procesar(thread_id);
23   }
24   using ColaVA<TVariasSubcolas...>::Agregar;
25   void Agregar(const TOtraSubcola::TipoTrabajo &un_trabajo) {
26     subcola.Push(un_trabajo);
27   }
28 };
29
30 // ejemplo de instanciacion de la clase con tres subcolas
31 // que ordenan sus trabajos de forma diferente (los cuales
32 // son ademas de diferente tipo)
33 ColaVA <
34     PilaThreadSafe<FunctorPriorida1>,
35     ColaThreadSave<FunctorPriorida2>
36     ColaThreadSafe<FunctorPriorida3>,
37   > ejemplo;
```

A.3. Implementación de un árbol para búsquedas de AABB basado en el concepto de ADT

Como se describió en la sección 6.3.1, el algoritmo propuesto para la generación de tetraedrizaciones en 3D respetando una frontera impuesta hace uso de una estructura de datos basada en el concepto de Alternating Digital Tree (ADT) para optimizar la detección de intersecciones. El ADT es originalmente un árbol binario de búsqueda para puntos multidimensionales "alternado". Esto quiere decir que en cada nivel del árbol se utiliza una de las dimensiones del punto para realizar la búsqueda o el ordenamiento, y estas dimensiones van rotando cíclicamente a medida que se avanza en el recorrido nivel. Se suele utilizar un ADT $2N$-dimensional para buscar Axis-Aligned Bounding Boxes (AABB) de triángulos en N dimensiones (más detalles en la sección4.2.2). La variante utilizada en este trabajo funciona como un *bucketed*-tree (es decir, que puede contener más de un punto por hoja) y es en realidad n-ario (cada nodo interior tendrá n hijos, con n fijo pero configurable). La implementación utilizada, denominada internamente RTree, gestiona la estructura del árbol, y realiza las operaciones a partir de los puntos $2N$-dimensionales. Es decir, queda como responsabilidad para el programa cliente realizar la conversión de AABB a punto (como se detalló en la sección 4.2.2). Permite definir en tiempo de compilación, mediante polimorfismo estático (templates) la dimensionalidad (DIMS) de los puntos, un tipo de dato arbitrario (Data) para asociar a cada punto almacenado, la cantidad de hijos por nodo interior del árbol (SUBS), la cantidad máxima de puntos por hoja (BUCKETS), y una clase auxiliar a la cual delegar la gestión de memoria (Allocator):

```
1  template <int DIMS, class Data,
2            int SUBS, int BUCKETS,
3            template<typename U> class Allocator>
4  class RTree {
5
6  public:
7    // punto almacenado en el arbol
8    struct Point { double x[DIMS]; ... };
9    // intervalo de busqueda
10   struct Interval { Point min, max; ... };
11
12 private:
13   // base para ambos tipos de nodo
14   enum node_type{ NT_INNER, NT_LEAF };
15   struct Node { node_type type; };
16   // nodo hoja
17   struct LNode : public Node {
```

```
18      Point point[BUCKETS];
19      Data data[BUCKETS];
20      int count;
21      ...
22    };
23    // nodo interior
24    struct INode : public Node {
25      INode *parent;
26      Interval interval;
27      Node *child[SUBS];
28      double val[SUBS-1];
29    };
30
31  public:
32    // rango de nodos (resultado de una busqueda)
33    class Range { ... };
34    // iterador para recorrer el arbol o un Range
35    class Iter { ... };
36
37    // operaciones basicas sobre el arbol
38    RTree(Interval _interval):interval(_interval);
39    Iter Find(const Point &_point) const;
40    void Add(const Point &_point, const Data &_data);
41    void Add(const Point &_point, const Data &_data, Iter _it);
42    bool Del(const Point &_point, const Data &_data);
43    bool Del(const Data &_data, Iter _it);
44    Range Search(const Interval &_interval) const;
45  };
```

La estructura utiliza internamente dos clases INode y LNode para representar los nodos interiores y hojas respectivamente. Se utilizan clases separadas (lo cual genera la necesidad de realizar conversiones explícitas y utilizar operaciones que no serán *type-safe*) y no una *union* (o mejor aún alguna implementación de *variant*) porque ambos tipos de nodos pueden llegar a tener tamaños muy diferentes (dependiendo de los parámetros actuales con los que se especialice el template). Para poder gestionar ambos tipos de nodos con un mismo tipo de puntero, ambas clases derivan de Node, que mediante un enum permite identificar el tipo real de una instancia. Expone como parte de su interfaz dos structs: Interval y Range. El primero hace referencia a un intervalo del espacio de búsqueda (es decir, dos puntos máximo y mínimo de DIMS dimensiones) y se utiliza como entrada en las búsquedas. El segundo representa un conjunto de nodos resultado de una búsqueda. Es decir, un rango de nodos del árbol, que podrá recorrerse mediante un iterador propio (Iter), o más fácilmente mediante invocaciones reentrantes a uno de sus métodos (ejemplo más abajo). La clase contiene entonces métodos para agregar (Add) y quitar (Del) puntos (tanto conociendo el iterador que le apunta, como teniendo que realizar internamente una búsqueda previa), y para realizar las consultas (Find para buscar un elemento, Search para un intervalo). La única condición importante para su utilización en tiempo de ejecución,

es la necesidad de conocer al momento de la inicialización el intervalo total en el cual se encontrarán los puntos (ya que se utilizará para decidir cómo subdividir el dominio en cada nivel). En la aplicación de esta tesis, esto no es un problema ya que se construye un ADT por trabajo, y dicho intervalo se obtiene directamente a partir de AABB de los puntos del trabajo.

Se mostró en las figuras 6.9 y 6.8 la conveniencia de usar entre 16 y 32 buckets, y tres hijos en lugar de dos para cada nodo interior. En la figura 7.5 se muestra también la conveniencia de utilizar una estrategia ad-hoc para la gestión de memoria. La interfaz de esta estructura utiliza la misma técnica presentada en el apéndice A.1 para delegar esta estrategia a una clase Allocator, y permite reutilizar las mismas implementaciones de allocators ya propuestas.

```
1 using RTreeCaras = RTree< 6, // DIMS
2                           ListaDoble<CaraFull>::Iterator, // Data
3                           3, // SUBS
4                           24, // BUCKETS
5                           DefaultListAllocator >; // Allocator
```

Con la interfaz planteada, la forma más eficiente y usual de recorrer el árbol para una consulta es la siguiente:

```
1 RTreeCaras::Range range = dinfo.tree->Search(interval);
2 CaraLista it;
3 while (range.GetNext(it)) {
4   // utilizar it
5 };
```

La llamada a RTree::Search busca los extremos del intervalo en el árbol y genera un Range para recorrer todos los nodos del árbol entre dichos extremos. A partir de esta instancia de RTree::Range, se pueden recuperar todos los punteros a caras (Data) asociados a todos los puntos en el rango. Cada llamada a RTree::Range::GetNext asigna una nueva cara en it y retorna true para indicar el éxito de la operación, o retorna false para indicar que se llegó al final del intervalo. La clase RTree::Range se encarga de realizar el recorrido por el árbol de forma eficiente (gestionando todas las variables de estado necesarias para el recorrido) y de seleccionar uno por uno todos los puntos de los nodos hoja que contengan datos, y saltear los nodos hoja vacíos.

A.4. Sobre redondeo y precisión numérica en variables de punto flotante

La forma más usual de representar números reales en la memoria de una PC mediante una secuencia binaria de longitud fija se basa en lo que se conoce como representación de punto flotante. El estándar IEEE 754-1985[65] define una forma de almacenar y operar con números de punto flotante que se utiliza en la mayoría de los lenguajes de programación actuales. En la representación de punto flotante un número real se guarda descomponiéndolo en mantisa y exponente. Cada parte se almacena de forma similar a un entero sin signo (simplemente escribiendo el entero en base 2) en dos conjuntos de bits dentro de la secuencia, y se utiliza además un bit adicional para el signo. Dado que cada parte se guarda con precisión limitada (en un float, 8 bits para el exponente y 23 para la mantisa), no todos los número reales pueden ser representados. Por ello, la mayoría de las operaciones arrojan un resultado aproximado, pero las operaciones aritméticas especificadas en el estándar IEEE garantizan que el valor resultante será redondeado correctamente al valor representable más cercano. En un cálculo complejo (que requiere varias operaciones). Se debe notar que la diferencia entre dos reales representables consecutivos es menor cuanto menor sean los valores. Es decir, la granularidad no es constante a en el intervalo que contiene todos los flotantes representables. Además, en un cálculo complejo (que requiere varias operaciones) el redondeo se produce para cada resultado parcial hasta llegar al resultado final. Por todo esto, cuando en un cálculo intervienen operandos de ordenes de magnitud muy diferentes, el orden de las operaciones puede afectar fuertemente el error resultante. Esto significa que se pierden en muchos casos las propiedades de conmutatividad y asociatividad.

Más allá de los errores que acarrean estas operaciones, sería esperable que si se realiza dos veces una misma secuencia de operaciones y siempre en un mismo orden, el resultado sea exactamente el mismo, de modo que se pueda comparar con seguridad ambos resultados. Pero en la práctica, las arquitecturas de hardware actuales, en combinación con los compiladores, introducen una fuente de discrepancia adicional que lleva a que esto no siempre suceda. El problema es que las CPUs actuales tienen internamente FPUs (floating point unit) dedicadas exclusivamente a realizar operaciones con valores de punto flotante, y la precisión en las mismas puede no ser igual a la precisión fuera de ellas. En C++ un valor de tipo float se almacena con 32 bits y un valor de tipo double se almacena con 64 bits. Muchas FPUs operan internamente siempre con registros de 80 bits, obteniendo un resultado con mayor

precisión, que será truncado cuando pase del registro a la memoria principal para poder ser almacenado allí. Si este pasaje no se realiza, el valor no se trunca y entonces el resultado no es el mismo. La mayoría de los compiladores actuales son capaces de optimizar el uso de la memoria y los registros del procesador para evitar el ida y vuelta entre ambos cuando un resultado de una operación será requerido pocas instrucciones después de haber sido calculado. En estos caso entonces, el compilador no reserva lugar en memoria para el mismo, sino que lo mantiene directamente en un registro, evitando así el truncamiento. Por esto, las comparaciones que se mencionaban al principio del párrafo dejan de ser confiables. Una misma secuencia de cálculos utilizada en dos contextos diferentes puede sufrir diferentes optimizaciones.

```
1  double StrToDbl(const string &s) { ... }
2  int main(int argc, char *argv[]) {
3      string s1="3.6055512754639891";
4      string s2="3.6055512754639891";
5      double d1=StrToDbl(s1);
6      cerr<<d1<<endl;
7      double d2=StrToDbl(s2);
8      cout<<(d1<d2)<<endl;
9      return 0;
10 }
```

El ejemplo anterior, compilado con gcc-4.7.3 en un sistema GNU/Linux de 64 bits con el argumento -O1 (que activa un conjunto de optimizaciones incluyendo la que aquí se discute, denominada *float-store*) arroja diferente resultado si se elimina la linea 6. Esto se debe a que en un caso d1 y d2 se optimizan almacenándose en registros, mientras que en el otro solo ocurre con d2. Se puede observar fácilmente si una variable ha sido optimizada de esta forma intentando inspeccionar su valor con un depurador (como gdb). Si la variable no existe en memoria, el depurador no logra mostrar su contenido.

Este comportamiento suele estar igualmente habilitado por defecto en muchos compiladores actuales, ya que en la mayoría de las aplicaciones esto no es un problema y la ganancia en el rendimiento es apreciable. Se han logrado medir mejoras de más de 30 % en el tiempo de ejecución en aplicaciones que hacen cálculos con punto flotante intensivamente. Como primera solución, los compiladores permiten desactivar esta optimización para evitar el problema a costa de una pérdida de rendimiento (por ejemplo, en gcc y llvm-clang con el argumento "-ffloat-store" . Una segunda solución es reconfigurar la FPU para operar siempre en 64bits o menos, pero no hay forma de hacerlo con C++ estándar, sino que se deben utilizar funciones propias del sistema operativo o bibliotecas escritas con lenguaje ensamblador. La palabra clave "volatile" es otra alternativa que se propone frecuentemente en foros de C++, pero en este caso su uso no es correcto (no aporta la garantía

que se busca). Una solución, que aunque no presenta una sintaxis estándar se puede expresar en todos los compiladores modernos, se presenta en [77]. Esta propuesta consiste en utilizar inline-assembler para anularle al optimizador toda garantía que pueda deducir sobre una dirección de memoria:

```
1 void escape(void *p) {
2   asm volatile("" : : "g"(p) : "memory");
3 }
```

La función "escape" presentada (utilizando la sintaxis para inline-asm de gcc y clang) no añade código adicional en el ejecutable, pero indica al compilador que el contenido de la memoria apuntada podría haber sido modificado arbitrariamente. Esta es la técnica utilizada en el generador desarrollado como parte de este trabajo, para anular selectivamente las optimizaciones en ciertas comparaciones críticas del código (por ejemplo, al determinar de qué lado del plano divisor se encuentra un determinado nodo).

Bibliografía

[1] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005.

[2] J J Monaghan. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68(8):1703, 2005.

[3] S.R. Idelsohn, E. Oñate, F. Del Pin, and Nestor Calvo. Fluid-structure interaction using the particle finite element method. *Computer Methods in Applied Mechanics and Engineering*, 195(17-18):2100–2123, 2006. Fluid-Structure Interaction.

[4] Eugenio Oñate, Sergio R. Idelsohn, Miguel A. Celigueta, and Riccardo Rossi. Advances in the particle finite element method for the analysis of fluid-multibody interaction and bed erosion in free surface flows. *Computer Methods in Applied Mechanics and Engineering*, 197(19-20):1777–1800, 2008. Computational Methods in Fluid-Structure Interaction.

[5] Su Hao, Wing Kam Liu, and Ted Belytschko. Moving particle finite element method with global smoothness. *International Journal for Numerical Methods in Engineering*, 59(7):1007–1020, 2004.

[6] N. Nigro et al. A new approach to solve incompressible Navier-Stokes equations using a particle method. In *Mecánica Computacional*, volume XXX, XIX Congreso Sobre Métodos Numéricos y sus Aplicaciones, 2011.

[7] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[8] P. Novara and N. Calvo. Remallado mínimo en problemas de spinning-forming. In *Mecánica Computacional*, volume XXVIII, Congreso sobre Métodos Numéricos y sus Aplicaciones, 2009.

[9] P. Novara and N. Calvo. Corte de mallas de elementos finitos 2.5-dimensionales. In *Mecánica Computacional*, volume XXIX, IX Argentinean Congress on Computational Mechanics, II South American Congress on Computational Mechanics and XXI Iberian-Latin-American Congress on Computational Methods in Engineering, 2010.

[10] P. Novara and N. Calvo. Refinamiento guiado por una curva de mallas 2.5 dimensionales. In *Mecánica Computacional*, volume XXXI, X Congreso Argentino de Mecánica Computacional, 2012.

[11] N. Calvo. *Generación de mallas tridimensionales por métodos duales*. PhD thesis, Facultad de Ingeniería y Ciencias Hídricas de la Universidad Nacional del Litoral., Santa Fe, Argentina, 2005.

[12] Jonathan Richard Shewchuk. What is a good linear element? - interpolation, conditioning, and quality measures. In *In 11th International Meshing Roundtable*, pages 115–126, 2002.

[13] I. Babuška and A. Aziz. On the angle condition in the finite element method. *SIAM Journal on Numerical Analysis*, 13(2):214–227, 1976.

[14] Rainald Lönher. A 2nd generation parallel advancing front grid generator. *21st International Meshing Roundtable*, pages 457–474, Oct 2012.

[15] Jörg Rambau. On a generalization of schönhardt's polyhedron. In *Goodman, Jacob E. ; Pach, János ; Welzl, Emo (Hrsg.): Combinatorial and computational geometry*, volume 52 of *Mathematical Sciences Research Institute publications*, pages 501–516. Cambridge Univ. Press, Cambridge, 2005.

[16] Gill Barequet, Matthew Dickerson, and David Eppstein. On triangulating three-dimensional polygons. *Computational Geometry*, 10(3):155–170, 1998.

[17] Jonathan Richard Shewchuk. General-dimensional constrained Delaunay and constrained regular triangulations i: Combinatorial properties. In *Discrete and Computational Geometry*, 2005.

[18] Charles L. Lawson. C^1 surface interpolation for scattered data on a sphere. *Rocky Mountain Journal of Mathematics*, 14(1):177–202, Mar 1984.

[19] Samuel. Rippa. Minimal roughness property of the Delaunay triangulation. *Computer Aided Geometric Design*, 7(6):489–497, Oct 1990.

[20] E. F. D'Azevedo and R. B. Simpson. On optimal interpolation triangle incidences. *SIAM J. Sci. Stat. Comput.*, 10(6):1063–1075, Nov 1989.

[21] R. Sibson. A brief description of natural neighbor interpolation. In V. Barnett, editor, *Interpreting Multivariate Data*, chapter 2, pages 21–36. John Wiley, 1981.

[22] Herbert Edelsbrunner, Roman Waupotitsch, and Tiow Seng Tan. An $O(n^2 logn)$ time algorithm for the minmax angle triangulation. Technical Report UIUCDCS-R-90-1575, University of Illinois (Urbana Champaign, IL US), 1990.

[23] Jane Tournois, Rahul Srinivasan, and Pierre Alliez. Perturbing slivers in 3d Delaunay meshes. *Proceedings of the 18th International Meshing Roundtable*, pages 157–173, 2009.

[24] Herbert Edelsbrunner, Xiang-Yang Li, Gary Miller, Andreas Stathopoulos, Dafna Talmor, Shang-Hua Teng, Alper Üngör, and Noel Walkington. Smoothing and cleaning up slivers. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 273–277, New York, NY, USA, 2000. ACM.

[25] Marelo Venere Walter Sotil, Nestor Calvo. Optimización de conectividades de una malla de tetraedros mediante retriangulaciones locales. In *Mecánica Computacional*, volume XXIX, IX Argentinean Congress on Computational Mechanics, II South American Congress on Computational Mechanics and XXI Iberian-Latin-American Congress on Computational Methods in Engineering, 2010.

[26] Nestor Calvo Walter Sotil. Retriangulación local para eliminación de slivers. In *Mecánica Computacional*, volume XXVIII, Congreso sobre Métodos Numéricos y sus Aplicaciones, 2009.

[27] François Labelle. Sliver removal by lattice refinement. In *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry*, pages 347–356, New York, NY, USA, 2006. ACM Press.

[28] Jianfei Liu, Shuli Sun, and Yongqiang Chen. Spr - a new method for mesh improvement and boundary recovery. In *Computational Mechanics*, pages 180–186. Springer Berlin Heidelberg, 2009.

[29] A. Bowyer. Computing Dirichlet tessellations. *The Computer Journal*, 24(2):162–166, Ene 1981.

[30] D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoï polytopes. *The Computer Journal*, 24(2):167–172, Ene 1981.

[31] Timothy Baker. Three dimensional mesh generation by triangulation of arbitrary point sets. *8th Computational Fluid Dynamics Conference*, 1987.

[32] T. J. Baker. Automatic mesh generation for complex three-dimensional regions using a constrained Delaunay triangulation. *Eng. with Computers*, 5:161–175, 1989.

[33] Timothy J. Baker. Delaunay-Voronoï methods. In Joe F. Thompson, Bharat K. Soni, and Nigel P. Weatherhill, editors, *Handbook of Grid Generation*. CRC Press, 1999.

[34] P. L. George, F. Hecht, and E. Saltel. Constraint of the boundary and automatic mesh generation. In *Proc. 2nd Int. Conf. on Numer. Grid Generation in Comp. Fluid Mechanics*, 1988.

[35] N. P. Weatherhill and O. Hassan. Efficient three-dimensional grid generation using the Delaunay triangulation. In *Proc. of the 1st. European Computational Fluid Dynamics Conf.* Elsevier, 1992.

[36] K. Sharov, D.; Nakahashi. A boundary recovery algorithm for Delaunay tetrahedral meshing. *International conference on numerical grid generation in computational fluid dynamics and related fields*, 1996.

[37] Jonathan Richard Shewchuk. Constrained Delaunay tetrahedralizations and provably good boundary recovery. In *In Eleventh International Meshing Roundtable*, pages 193–204, 2002.

[38] Hamid Ghadyani, John Sullivan, and Ziji Wu. Boundary recovery for Delaunay tetrahedral meshes using local topological transformations. *Finite Elem. Anal. Des.*, 46(1-2):74–83, Ene 2010.

[39] Masaharu Tanemura, Tohru Ogawa, and Naofumi Ogita. A new algorithm for three-dimensional Voronoï tessellation. *J. Comput. Physics*, 51(2):191–207, Ago 1983.

[40] Dimitri J. Mavriplis. An advancing front Delaunay triangulation algorithm designed for robustness. *Journal of Computational Physics*, 117(1):90–101, 1995.

[41] Marshal L Merriam. An efficient advancing front algorithm for Delaunay triangulation. *AIAA, Aerospace Sciences Meeting*, 1991.

[42] Leonidas Linardakis. *Decoupling Method for Parallel Delaunay Two-dimensional Mesh Generation*. PhD thesis, College of William & Mary, Williamsburg, VA, USA, 2007. AAI3282512.

[43] Mark S. Sheperd Hugues L. de Cougny. Parallel unstructured grid generation. In Joe F. Thompson, Bharat K. Soni, and Nigel P. Weatherhill, editors, *Handbook of Grid Generation*. CRC Press, 1999.

[44] P Cignoni, C Montani, and R Scopigno. DeWall: A fast divide and conquer Delaunay triangulation algorithm. *Computer-Aided Design*, 30(5):333–341, 1998.

[45] Nikos P. Chrisochoides. A survey of parallel mesh generation methods. Technical report, Brown University, 2005.

[46] Pablo Halpern. Overview of parallel programming in C++. In *CppCon 2014*, Bellevue, WA, United States, Sep 2014.

[47] Chandler Carruth. Efficiency with algorithms, performance with data structures. In *CppCon 2014*, Bellevue, WA, United States, Sep 2014.

[48] Intel® Corporation David Levinthal. Performance analisys guide for Intel Core I7 and Intel Xeon 5500 processors. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf, 2009.

[49] D. T. Marr et al. Hyper-Threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), 2002.

[50] Microsoft Corporation. C++ AMP: Language and programming model (version 1.0), Ago 2012.

[51] Mark Mitchell and Alex Samuel. *Advanced Linux Programming*. New Riders Publishing, Thousand Oaks, CA, USA, 2001.

[52] Intel®. Intel VTune Amplifier 2015. https://software.intel.com/en-us/intel-vtune-amplifier-xe, 2015.

[53] Christoph Lameter. NUMA (Non-Uniform Memory Access): An overview. *ACM Queue*, 11(7):40, 2013.

[54] W. J. Bolosky and M.L. Scott. False sharing and its effect on shared memory performance. *4th USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, 1993.

[55] ISO/IEC JTC1 SC22 ISO/IEC CD 14882. *Standard for Programming Language C++, Working draft, ISO/IEC JTC1 SC22 WG21 N3690*, 2013.

[56] Bjarne Stroustroup. Keynote: C++11 style. In *Going Native 2012*, Redmon, WA, USA, Feb 2012.

[57] J. Bonet and J. Peraire. An alternating digital tree (ADT) algorithm for 3D geometric searching and intersection problems. *International Journal for Numerical Methods in Engineering*, 31(1):1–17, 1991.

[58] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000.

[59] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Lock-free dynamically resizable arrays. In MariamMomenzadehAlexanderA. Shvartsman, editor, *Principles of Distributed Systems*, volume 4305 of *Lecture Notes in Computer Science*, pages 142–156. Springer Berlin Heidelberg, 2006.

[60] Herb Sutter. Lock-free programming (or, juggling razor blades), part II. In *CppCon 2014*, Bellevue, WA, United States, Sep 2014.

[61] M.M. Michael. ABA prevention using single-word instructions. *Technical Report RC 23089, IBM T.J. Watson Research Center*, Ene 2004.

[62] Herb Sutter. Lock-free programming (or, juggling razor blades), part I. In *CppCon 2014*, Bellevue, WA, United States, Sep 2014.

[63] Keir Fraser. Practical lock freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[64] D.H. Mclain. Two dimensional interpolation from random data. *Computer Journal*, 19(2):178–181, 1976.

[65] IEEE. IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, Ago 2008.

[66] ISO. ISO/IEC/IEEE 60559:2011 Information technology - Microprocessor Systems- Floating-Point arithmetic. Technical report, International Organization for Standardization, Geneva, Switzerland, 2011.

[67] P. Novara and N. Calvo. Generación de mallas de tetraedros Delaunay en paralelo a partir de una nube de puntos y una frontera impuesta. In *Mecánica Computacional*, volume XXXI, X Congreso Argentino de Mecánica Computacional, 2012.

[68] Brendan Greg. Flamegraphs. http://www.brendangregg.com/ flamegraphs.html. Herramienta para la visualización de información de profiling.

[69] C/C++ reference. http://en.cppreference.com/w/cpp/container/set. Online; accessed 19 August 2015.

[70] P. Novara and N. Calvo. Implementación de un método paralelo de triangulación Delaunay euclídeo. In *Mecánica Computacional*, volume XXX, XIX Congreso Sobre Métodos Numéricos y sus Aplicaciones, 2011.

[71] William Gropp. MPICH2: A new start for MPI implementations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 7–, London, UK, UK, 2002. Springer-Verlag.

[72] Yannis Fragakis and Eugenio Oñate. Parallel Delaunay triangulation for particle finite element methods. *Communications in Numerical Methods in Engineering*, 24(11):1009–1017, 2008.

[73] D. Gregor and J. Järvi. Variadic templates for C++0x. *Journal of Object Technology*, 7(2):31–51, 2008.

[74] Chandler Carruth. Clang: Defending C++ from Murphy's million monkeys. In *Going Native 2012*, Redmon, WA, USA, Feb 2012.

[75] Andrew Sutton and Bjarne Stroustrup. Design of Concept Libraries for C++. In Anthony M. Sloane and Uwe Aßmann, editors, *Revised Selected Papers of the Fourth International Conference on Software Language Engineering*, volume 6940 of *Lecture Notes in Computer Science*, pages 97–118. Springer International Publishing, 2011.

[76] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 1 edition, Nov 2002.

[77] Chandler Carruth. Tuning C++: Benchmarks, and CPUs, and compilers! Oh my! In *CppCon 2015*, Bellevue, WA, United States, Sep 2015.