Paralelización especulativa de la triangulación de Delaunay

Alvaro García-Yágüez [1], Diego R. Llanos [1], David Orden [2], Belén Palop [1]

Resumen— **En este trabajo utilizamos la técnica de la paralelización especulativa (también conocida como "paralelización optimista") para ejecutar automáticamente en paralelo un algoritmo secuencial con un patrón irregular de violaciones de dependencia. Nos centramos en un importante problema: la triangulación de Delaunay de un conjunto de puntos de dos dimensiones. Mostramos que nuestro motor de paralelización especulativa basado en software, SPECENGINE, es efectivo en aplicaciones y entornos reales. Nuestros resultados experimentales muestran una buena escalabilidad de la solución para más de diez procesadores (5,32×), siendo esta solución aplicable a conjuntos de entrada muy grandes (del orden de millones de puntos).**

Palabras clave— **Paralelización automática, paralelización especulativa, paralelización optimista, algoritmos incrementales aleatorizados, geometría computacional.**

I. INTRODUCCIÓN

LA s técnicas de paralelización automática atraen mucho interés desde los últimos 30 años. Se han propuesto muchas técnicas para extraer el paralelismo de los bucles usando análisis estático en tiempo de compilación. Al no disponer de cierta información que sólo se obtiene en tiempo de ejecución, los compiladores son muy conservadores en la elección de los bucles a paralelizar, conduciendo a resultados poco eficientes excepto en casos muy especiales. La paralelización especulativa supera esas limitaciones usando información obtenida durante la ejecución, ampliando así el rango de aplicaciones que pueden beneficiarse de la paralelización automática.

En este artículo nos centramos en la paralelización especulativa de la triangulación de Delaunay de un conjunto de puntos en dos dimensiones. Al ser la estructura que mejor capta la proximidad entre puntos de un conjunto, tiene numerosas aplicaciones y se le ha dedicado mucho esfuerzo a su construcción, con varios algoritmos teóricos paralelos e implementaciones paralelas. Muchos de ellos pueden verse en el extenso estudio de Kolingerova y Kohout [1], donde se describe un algoritmo incremental aleatorizada *ad hoc*. Por otro lado, Kulkarni et al. han desarrollado un sistema para la paralelización automática de aplicaciones irregulares en el que el conocimiento experto del problema se usa para guiar el *scheduling* para 75.000 puntos y con hasta 4 procesadores, obteniendo un *speedup* de 3,03× para pequeños conjuntos de entrada de hasta 75.000 puntos.

El objetivo de este trabajo es la extracción automática de paralelismo en la construcción incremental aleatorizada de la triangulación de Delaunay, sin proporcionar un conocimiento experto del problema. Utilizamos un motor de paralelización especulativa basado en software [2], [3], llamado SPECENGINE, que permitirá que diferentes procesadores inserten puntos automáticamente al mismo tiempo. Si surge una violación de dependencia, SPECENGINE detiene automáticamente el hilo que está trabajando con datos erróneos hasta que el hilo no especulativo los termina de actualizar. Nuestros resultados muestran un máximo de *speedup* de $5,32\times$ para 13 procesadores en un sistema real, con diferentes conjuntos de entrada de hasta 10 millones de puntos. Se analiza en detalle el tiempo de ejecución en paralelo, mostrando la baja sobrecarga debida a la ejecución especulativa.

El trabajo se organiza como sigue. La Sección II introduce la triangulación de Delaunay, su construcción incremental aleatorizada y la estrategia de *jump-and-walk* para balancear el coste de la localización. La Sección III describe el funcionamiento y algunos detalles internos de SPECENGINE. La Sección IV muestra cómo la localización afecta de manera directa a la ejecución especulativa. La Sección V muestra algunos de los resultados experimentales obtenidos por SPECENGINE sobre un sistema real para diferentes conjuntos de entrada. La Sección VI resume nuestras conclusiones.

II. LA TRIANGULACIÓN DE DELAUNAY

La triangulación de Delaunay y su estructura dual, el diagrama de Voronoi, se encuentran entre las estructuras más importantes en Geometría Computacional, con numerosas aplicaciones en diferentes campos que van desde los sistemas de información geográfica hasta los métodos de elementos finitos, ver [4] y sus referencias.

Sea $S \subset \mathbb{R}^2$ un conjunto de n puntos. Una *triangulación* de S se define como un grafo geométrico plano (tomando los puntos como vértices y segmentos rectos como aristas) tal que si se añadiera una arista, ser rompería la propiedad planar (*maximal planar*). Es decir, todas las caras del grafo están delimitadas por tres aristas, tomando como vértices los puntos de S. La *triangulación de Delaunay* ,$\mathcal{DT}(\mathcal{S})$, es la triangulación que maximiza el ángulo más pequeño. Esta triangulación cumple también que el circuncírculo de cada triángulo de $\mathcal{DT}(\mathcal{S})$ no contiene ningún otro punto de S (lo que se conoce como *la propiedad del círculo vacío*).

La triangulación de Delaunay se usa ampliamente debido a la eficiencia y variedad de algoritmos conocidos para su computación. Los paradigmas habituales en Geometría Computacional como el barrido [5] o divide y vencerás [6], conducen a una complejidad del cálculo de $\mathcal{DT}(\mathcal{S})$ del orden de $O(n \log n)$ en el peor caso. El paradigma incremental aleatorizado, en el que los puntos se insertan en orden aleatorio, muestra un buen rendimiento en la práctica, a pesar de una complejidad esperada del orden de $O(n \log n)$ sin ser aún el peor de los casos [7].

Desde el punto de vista práctico, no es sencillo ele-

[1]Departamento de Informática, Universidad de Valladolid, Spain, e-mail: alvaro.garcia.yaguez@alumnos.uva.es, {diego,bpalop}@infor.uva.es

[2]Departamento de Matemáticas, Universidad de Alcalá, Spain, e-mail: david.orden@uah.uva.es

gir el *mejor* algoritmo. La decisión depende de muchos factores tales como el tiempo de ejecución, las particularidades del conjunto de entrada, la cantidad de memoria necesaria, la posibilidad de añadir nuevos puntos y, por supuesto, el esfuerzo necesario para su implementación. El correcto equilibrio que muestra en dichos factores el algoritmo incremental aleatorizado le ha convertido en uno de los más populares, formando parte de la librería `CGAL` [8], [9] y del software `Triangle` [10].

A. *Construcción incremental aleatorizada de la triangulación de Delaunay*

El algoritmo incremental aleatorizado para la construcción de la triangulación de Delaunay de un conjunto de puntos S sigue la estructura básica de todos los algoritmos de esta clase. En esta sección se da sólo una descripción general del algoritmo. Una descripción detallada junto con un análisis de su complejidad se puede encontrar en [11]. El algoritmo 1 muestra su pseudocódigo. La Figura 1 muestra un ejemplo de los tres pasos (cuatro al seis) del bucle principal del algoritmo, donde el punto p_i tiene que insertarse en la triangulación de Delaunay de los $i-1$ puntos anteriores.

Algorithm 1 Construcción incremental aleatorizada de la triangulación de Delaunay

Require: S es un conjunto de n puntos $p_1, \ldots, p_n$ en orden aleatorio
Ensure: $\mathcal{DT}(\mathcal{S})$ es la triangulación de Delaunay del conjunto S
1: Encontrar un triángulo $t_0 = \{p_{-2}, p_{-1}, p_0\}$ que encierre a S
2: Encontrar $\mathcal{DT}(\mathcal{S}) = t_0$
3: **for all** $p_i \in S$ **do**
4: Localizar en $\mathcal{DT}(\mathcal{S})$ el triángulo t (o el borde, en caso de degeneraciones) que contenga a p_i
5: Insertar el punto en $\mathcal{DT}(\mathcal{S})$, añadiendo los tres bordes desde p_i a los vértices de t (cuatro bordes, en caso de degeneraciones)
6: Si la propiedad del circuncírculo vacío no se cumple, recursivamente hacer un *flip* (modificación de aristas) a los correspondientes bordes hasta que se cumpla la propiedad para cada triángulo
7: **end for**
8: Eliminar de $\mathcal{DT}(\mathcal{S})$ los vértices $\{p_{-2}, p_{-1}, p_0\}$ y sus bordes adyacentes

El tiempo esperado del Algoritmo 1 es $O(n \log n)$, ejecutando el paso 4 en $O(\log n)$; el paso 5 en $O(1)$; y siendo el paso 6 constante amortizado. Esta complejidad logarítmica en la localización de un punto dentro de la triangulación se logra mediante elevados costes en términos de memoria empleada para las estructuras de datos. Por lo tanto, las alternativas no óptimas para el paso 4 que usen simples estructuras de datos (o ninguna, como en nuestro caso) reciben cada vez un interés cada vez mayor (ver [12] y sus referencias).

En este trabajo nos centramos en la estrategia *jump-and-walk* introducida por Mücke, Zhe et al. [13], [14], e implementada en software estándar como `Triangle` [10]. La complejidad esperada de todo el algoritmo es $O(n^{4/3})$ con una pequeña constante y sin sobrecarga de espacio, ya que no son necesarias estructuras de datos adicionales. A continuación discutiremos brevemente cómo esta estrategia resuelve el paso más costoso, el de la localización del punto (paso 4 en el Algoritmo 1).

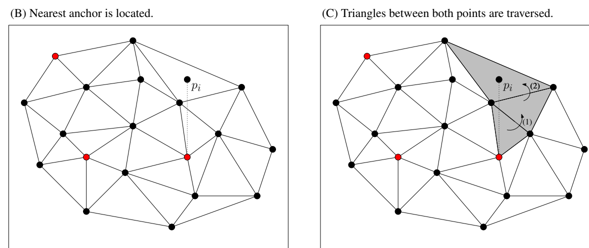

Fig. 2. Estrategia *Jump-and-walk* para añadir nuevos puntos a la actual triangulación.

Algorithm 2 Localización de los puntos con *Jump-and-Walk*

Require: $\mathcal{DT}(\mathcal{S}^*_{i-1})$ es la triangulación de Delaunay de los primeros $i-1$ puntos en el conjunto $S \cup \{p_{-2}, p_{-1}, p_0\}$; p_i es el siguiente punto a insertar; hay m vertices $q_1, \ldots, q_m$ de $\mathcal{DT}(\mathcal{S}^*_{i-1})$ (*anchors*), de los que se conoce el triángulo al que pertenecen.
Ensure: t es el triángulo en $\mathcal{DT}(\mathcal{S}^*_{i-1})$ que contiene a p_i
1: (Jump) Encontrar el anchor más cercano a p_i, es decir, determinar el índice $j \in \{1, \ldots, m\}$ que minimiza $d(q_j, p_i)$.
2: (Walk) Atravesar $\mathcal{DT}(\mathcal{S}^*_{i-1})$ a lo largo del segmento desde q_j hasta p_i, realizándose consultas de orientación en cada triángulo hasta llegar al triángulo t donde se encuentra p_i.

Según Devroye et al. [13], el coste total de estos dos pasos es del orden de $O(m + \sqrt{i/m})$, donde el primer y el segundo sumando miden el paso del *jump* y del *walk*, respectivamente.

Por tanto, el número de *anchors* utilizados tiene que buscar un equilibro entre el coste del *jump* y del *walk*. En la ejecución secuencial esto se logra cuando ambos sumandos son iguales, lo que conduce a $m \in O(i^{1/3})$. Por otro lado, cuando el algoritmo se ejecuta en paralelo, los caminos seguidos por diferentes hilos para la localización de puntos pueden cruzarse y dar lugar a dependencias. Se sabe [13] que el número esperado de triángulos de $\mathcal{DT}(\mathcal{S}^*_{i-1})$ que cruzan el segmento $\overline{q_j p_i}$ es $O(|\overline{q_j p_i}|\sqrt{i})$. De este modo, cuanto más corta es la distancia entre el *anchor* más cercano q_j y el punto p_i a insertar, menor será el número de triángulos visitados en la fase de *walk* y menor será la probabilidad de dependencias. Por otro lado, cuanto mayor es el número de *anchors*, mayor número de comparaciones de distancias han de realizarse.

III. Paralelización especulativa basada en software

Una de las técnicas más prometedoras para paralelizar bucles automáticamente cuando las dependencias no

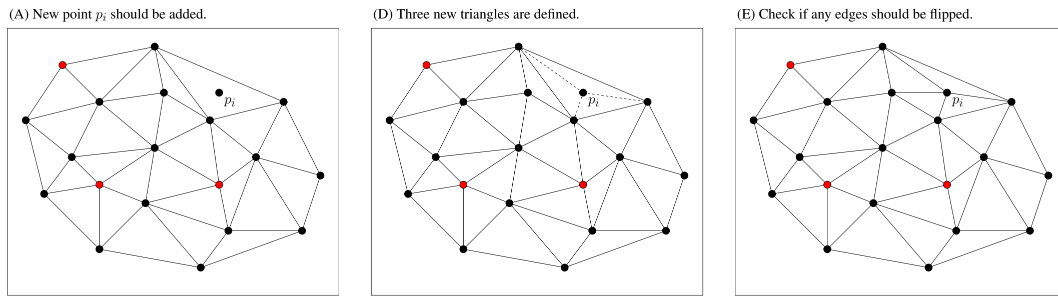

Fig. 1. Bucle principal de la construcción incremental aleatorizada.

pueden determinarse en tiempo de compilación es la paralelización especulativa. Esta técnica, también llamada *especulación a nivel de hilo* [2], [15], [16], o *paralelización optimista* [17], [18] asigna la ejecución de *bloques* de iteraciones consecutivas a diferentes hilos, ejecutándose cada uno en su propio procesador. Mientras se lleva a cabo la ejecución, un mecanismo hardware o software monitoriza el proceso para asegurarse que ningún hilo consuma una versión incorrecta de un valor que debiera haber sido calculado por un predecesor, lo que violaría la semántica secuencial. Si se produce una *violación de dependencia*, se descartan las iteraciones calculadas incorrectamente y se reinicia su ejecución en orden, usando los valores correctos.

La detección de violaciones de dependencia puede implementarse tanto vía hardware como software. Las soluciones hardware (ver por ejemplo [19], [20]) cuentan con módulos hardware adicionales para detectar las dependencias, mientras que los métodos software [2], [15], [16] añaden al bucle original un conjunto de instrucciones encargadas de comprobar que no se produzcan violaciones de dependencia durante la ejecución paralela. En [2], [3] presentamos un motor software de paralelización especulativa, SPECENGINE, para ejecutar en paralelo bucles secuenciales, sin conocimiento previo del número de dependencias entre iteraciones. La ventaja principal de esta solución es que es posible para un compilador generar automáticamente una versión paralela de una aplicación secuencial y obtener *speedups* en una máquina paralela evitando la ardua tarea de la paralelización manual. Para su desarrollo, el compilador instrumenta la aplicación añadiendo código con llamadas a funciones encargadas de acceder a la estructura de datos compartida y monitorizar la ejecución paralela del bucle.

Para comprender mejor el funcionamiento de la paralelización especulativa es necesario distinguir entre variables *privadas* y *compartidas*. Informalmente hablando, las variables *privadas* son aquellas que siempre se modifican en el contexto de una iteración antes de utilizarse *en esa misma iteración*. Por el contrario, los valores que se almacenan en variables *compartidas* pueden utilizarse en diferentes iteraciones en cualquier momento de su ejecución. Si todas las variables presentes en un bucle son *privadas*, no pueden surgir violaciones de dependencia y el bucle puede ejecutarse en paralelo. Las variables *compartidas* no siempre provocan violaciones de dependencia. Esto sólo sucede cuando un valor se modifica en una iteración determinada y un sucesor ha consumido con anteriordad una valor antiguo de esa variable. Esto se denomina "dependencia RAW" (*Read-after-Write*). En este caso, el hilo que ha consumido el valor antiguo de la variable y todos sus sucesores tienen que detenerse y volverse a ejecutar usando los valores correctos. Esto se conoce como una operación de *squash*.

Para simplificar el descarte de valores incorrectos, los hilos que ejecutan cada bloque no cambian directamente la estructura compartida. En vez de eso, cada hilo mantiene una copia local de la estructura. Solamente al final de la ejecución existosa de un bloque, los cambios se reflejan en la estructura compartida original, a través de una operación de consolidación (*commit*) . Esta operación debe de preservar el orden para cada bloque de iteraciones, desde el hilo menos especulativo (el que ejecuta el bloque de iteraciones con el índice más bajo) al más especulativo. Si la ejecución de un bloque de iteraciones falla, los valores calculados se descartan.

La ventaja clave de la paralelización especulativa es que no es necesario desarrollar una versión paralela de un algoritmo para poder ejecutarlo en paralelo: SPECENGINE lo hace automáticamente. Para ello, deben hacerse en el código algunos ajustes simples, que están al alcance de un compilador moderno. A continuación se presenta una breve descripción de estos cambios.

- **Lecturas especulativas** Como cada hilo mantiene su propia copia de la estructura de datos compartida, todas las lecturas a esta estructura deben reemplazarse con código que realiza una búsqueda hacia atrás de la versión más actualizada del valor que está siendo leído. Esta operación es conocida como *forwarding*.
- **Escrituras especulativas** Cualquier cambio en la estructura de datos compartida realizado por un hilo conducirá a una violación de dependencia si un hilo sucesor ha leído con anterioridad ese valor. Por lo tanto, todas las escrituras sobre la estructura de datos deben reemplazarse con código que realiza una búsqueda hacia adelante para comprobar si algún hilo ha consumido un valor incorrecto. Si se encuentra alguno, el consumidor del dato incorrecto y todos sus sucesores son descartados y sus ejecuciones son reiniciadas con el valor correcto.
- **Consolidación y gestión de hilos** Cuando un hilo acabe de ejecutar un bloque de iteraciones llamará a una función que compruebe su estado y realice la consolidación de los datos cuando corresponda. En vez de realizarse una única consolidación al final de la ejecución exitosa del bucle paralelo, se realizan consolidaciones parciales cada vez que un hilo

completa la ejecución de un bloque de iteraciones correctamente. En cuanto a la gestión de los hilos, SPECENGINE utiliza un mecanismo de ventana deslizante de tamaño fijo donde se mantienen las copias de la estructura de datos compartida que son actualizadas por los hilos. Para las pruebas en este trabajo hemos utilizado un tamaño de ventana igual al número de procesadores [3]. Cada hilo trabaja en su propio espacio sobre la ventana. Cuando el hilo menos especulativo acaba, consolida sus datos y avanza la ventana, permitiendole comenzar la ejecución de un nuevo bloque de iteraciones.

IV. INCLUENCIA DE LA ESTRATEGIA JUMP-AND-WALK EN LA EJECUCIÓN ESPECULATIVA

A continuación se describe cómo la paralelización especulativa puede favorecer en la construcción de la triangulación de Delaunay de un conjunto de puntos utilizando el algoritmo incremental aleatorizado con la estrategia *Jump-and-Walk* para la localización de puntos.

La implementación secuencial utiliza dos estructuras de datos. La primera es una estructura de sólo lectura que contiene las coordenadas de los puntos del conjunto de entrada. La segunda es una estructura de datos compartida que contiene la triangulación, que es, la lista de adyacencias entre triángulos calculada hasta el momento.

Cada hilo recibe un bloque de puntos consecutivos para añadir a la triangulación actual. Como el conjunto de entrada está en orden aleatorio, dos o más hilos podrían entrar en conflicto al intentar actualizar el mismo triángulo al mismo tiempo (ver Fig. 3). Por lo tanto, la estructura de datos compartida no se debe modificar directamente por los hilos. Como se dijo en la sección anterior, SPECENGINE resuelve este problema manteniendo diferentes copias de la estructura de datos compartida, una por cada hilo, permitiéndoles hacer todos los cambios necesarios localmente. Siempre que un hilo añada un nuevo punto, modificará los triángulos de su copia local y buscará entre los sucesores aquellos que hayan consumido el valor antiguo para localización[1] o inserción de nuevos puntos. Todos los hilos en esta situación (y sus sucesores) son entonces descartados y reiniciados con el fin de trabajar con los datos de la triangulación más actualizados y preservar la semántica secuencial.

La ejecución avanza cuando el hilo menos especulativo, que no puede ser descartado por un predecesor, acaba y consolida su versión local de la triangulación a la copia principal, avanzando la ventana deslizante y permitiendo la ejecución de un nuevo bloque de iteraciones.

Como vimos en la sección anterior, todos los cambios necesarios en el código secuencial para la ejecución especulativa pueden hacerse automáticamente, detectando en tiempo de compilación qué estructuras son de sólo lectura y cuáles son modificadas dentro del bucle a paralelizar. El acceso a estas últimas es reemplazado con código que realiza todas las operaciones especulativas

[1] Tenga en cuenta que el comportamiento de la paralelización en este paso es más bien conservador, donde el paso del *walk* en una triangulación es un proceso robusto que nos permitiría encontrar el triángulo contenedor incluso bajo esas circunstancias. Hemos decidido no explotar este hecho para mantener el propósito general de SPECENGINE.

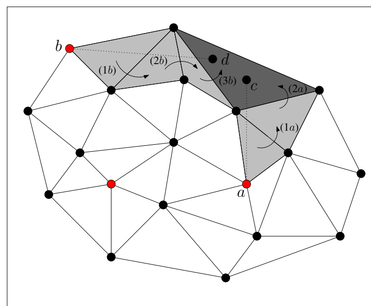

Fig. 3. Dos hilos podrían necesitar modificar el mismo triángulo simultáneamente. En este caso, el hilo más especulativo deberá esperar hasta que el menos especulativo acabe, para preservar la semántica secuencial.

necesarias. Veremos en la siguiente seccion que, a pesar del alto número de lecturas y escrituras a la estructura de datos compartida hecha por el código original, la sobrecarga debida a la ejecución especulativa de este algorimo es extremadamente baja, gracias a la eficencia de SPECENGINE, diseñado para minimizar el número de operaciones de sincronización y secciones críticas necesarias [3].

V. RESULTADOS EXPERIMENTALES

Las pruebas fueron llevadas a cabo sobre un Intel S7000FC4URE server, equipado con 4 procesadores quad-core Intel Xeon MPE7310 a 1,6GHz y 32GB de RAM. Tanto la versión secuencial como la especulativa se desarrollaron utilizando el compilador Sun Fortran 95 8.3 de Solaris. Todos los hilos tienen acceso exclusivo a los procesadores durante la ejecución de los experimentos, y hemos utilizado *wall-clock time* en nuestro experimentos. La medición del rendimiento del código se realizó usando Sun Analizer 7.6 profiler. Los flags de compilación para la versión secuencial fueron -xO4 -m64 -xmodel=medium -xchip=native -xarch=native -xcache=native -g -lcollectorAPI.

Se han empleado cuatro conjuntos de entrada diferente de $n = 10$ millones de puntos aleatorios sin degeneraciones: distribuciones uniformes en forma de cuadrado y de disco, una distribución normal y una Kuzmin. Todas ellas son habituales en pruebas sobre la triangulación de Delaunay [21], [22].

El algoritmo se ha implementado en FORTRAN 95 usando únicamente estructuras de memoria estáticas. Para cada triángulo, almacenamos los índices de sus triángulos adyacentes y sus vértices, permitiendo recorrer la estructura para la localización de puntos con el algoritmo del *jump-and-walk*. Siempre que se añade un nuevo punto, se modifica el triángulo localizado y se crean tres nuevos triángulos. Se realizan llamadas recursivas a una rutina que lleva a cabo los *flips* que sean necesarios hasta que se cumpla la propiedad del circuncírculo vacío. Utilizamos los primeros m puntos del conjunto como *anchors* (nótese que el conjunto de entrada está en orden aleatorio).

En la sección II hablamos sobre el compromiso entre el paso del *Jump* y del *Walk*. Vimos como el número de *anchors* utilizados afecta al tiempo de ejecución total del algoritmo desde un punto de vista asintótico.

TABLA I

DEPENDENCIAS DE LOS CONJUNTOS DE ENTRADA CON FSC

Nombre	Tamaño	Anchors	Tamaño de bloque	Violaciones	Squashes
Uniform Square	10 000 000	20 000	20 iterations	Up to 1.17 %	Up to 8.94 %
Uniform Disc	10 000 000	20 000	20 iterations	Up to 1.18 %	Up to 9.01 %
Normal	10 000 000	20 000	20 iterations	Up to 1.16 %	Up to 8.87 %
Kuzmin	10 000 000	20 000	20 iterations	Up to 1.19 %	Up to 9.18 %

El número de *anchors* también afecta a la ejecución paralela del algoritmo, ya que cuantos más *anchors* utilizemos menor será la probabilidad de que se produzcan cruces de caminos que conduzcan a *squashes*. Para asegurar una comparación equitativa entre el código secuencial y el paralelo, utilizamos $m = 20.000$ *anchors* para ambos códigos, un buen compromiso entre el coste de encontrar el *anchor* más cercano y el coste de las violaciones de dependencia producidas por cruces de caminos.

A. *Planificación FSC y su rendimiento*

Se ha utilizado una planificación de tamaño fijo para cada bloque (FSC), a través de bloques de tamaño óptimo de 20 iteraciones. La tabla I muestra el comportamiento para cada conjunto de entrada respecto a las dependencias. La columna “violaciones” muestra el porcentaje de bloques que han producido violaciones de dependencia, mientras que la columna “squahes” muestra el porcentaje de bloques cuya ejecución ha sido descartada debido a una violación de dependencia mientras se ejecutaba la aplicación con 16 procesadores. Tener en cuenta que cada violación de dependencia puede provocar el *squash* de varios bloques y que la ejecución del mismo bloque puede ser descartada varias veces. Como se puede apreciar en la figura, los conjuntos de entrada presentan un número de violaciones y un número de *squashes* similares.

La figura 4 muestra el *speedup* obtenido en la ejecución paralela especulativa del algoritmo incremental aleatorizado para el cálculo de la triangulación de Delaunay utilizando FSC. El rendimiento alcanza 5,32× para 13 procesadores, teniendo un buen comportamiento incluso para 4 procesadores (2,63×). Debe tenerse en cuenta que SPECENGINE desconoce el algoritmo que está siendo ejecutado especulativamente. Además de esto, se puede ver cómo el rendimiento es independiente de la distribución del conjunto de puntos de entrada. Desafortunadamente, la única forma de encontrar el tamaño óptimo de bloque de FSC es a base de pruebas, lo que requiere un proceso de prueba y error para fijar esta constante.

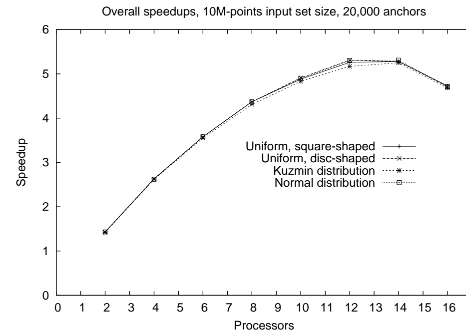

Fig. 4. Speedups para todos los conjuntos de entrada considerados.

B. *Análisis desglosado del tiempo de ejecución*

La figura 5 muestra el tiempo de ejecución desglosado para el procesamiento paralelo del conjunto de entrada en forma de cuadrado, para 10 millones de puntos utilizando FSC con 20.000 anclas. Podemos ver que hay diferentes secciones de tiempo implicadas en la ejecución paralela. Esos tiempos son: el tiempo de contención debido al acceso a secciones críticas, el tiempo dedicado a la operación de consolidación, el coste del algoritmo de planificación, las operaciones de memoria especulativas (lecturas y escrituras a una copia de la estructura compartida), y el tiempo de espera (“Spin” time) empleado por los hilos esperando a que la ventana deslizante se desplace hacia adelante para emitir un nuevo bloque de iteraciones. Finalmente, el tiempo ocupado representa el tiempo efectivamente empleado en el algoritmo original.

La sobrecarga es baja si tenemos en cuenta la complejidad de la ejecución especulativa. La mayoría del tiempo se emplea en operaciones de memoria especulativa, donde cada hilo se encarga de localizar el punto que debe añadir dentro de la triangulación actual (realizando varias lecturas a la estructura compartida, muchas de ellas disparando una operación de *forwarding*) y entonces modificar la estructura para colocar los nuevos triángulos y las aristas correspondientes (que implica más lecturas y escrituras). Nótese el poco tiempo empleado en contención y en esperas, así como el hecho de que el tiempo empleado en tareas de planificación es aproximadamente independiente del número de procesadores empleados. Finalmente, como se esperaba, los resultados muestran que la adición de más procesadores divide el tiempo de ejecución correspondiente al algoritmo original, pero cuando el número de procesadores es demasiado grande eleva el coste de la contención, imponiendo un límite al *speedup* efectivo de la aplicación.

VI. CONCLUSIONES

Este trabajo presenta los resultados obtenidos en la paralelización automática de un algoritmo incremental aleatorizado para calcular la triangulación de Delaunay. El problema es representativo y tiene aplicaciones en muchas áreas. Se ha elegido uno de los algoritmos más utilizados de entre todos los disponibles para su implementación. La paralelización se ha conseguido haciendo uso de un motor especulativo basado en software, SPECENGINE, que procesa el conjunto de entrada en paralelo sin un análisis previo de dependencias. Los resultados experimentales muestran el potencial de esta técnica.

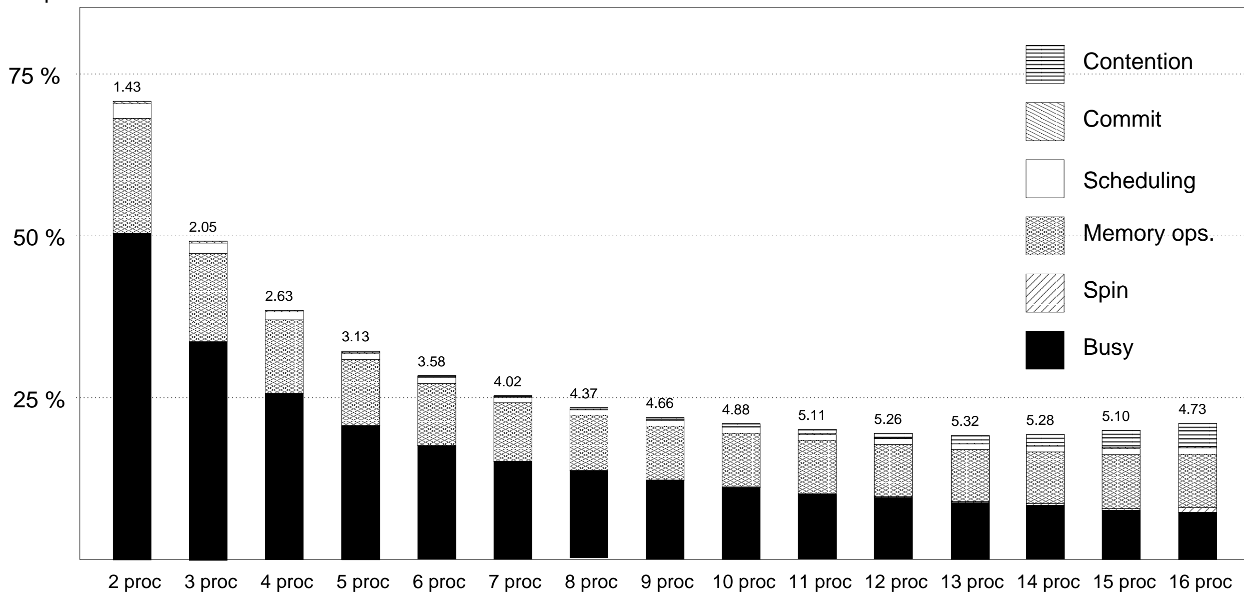

Fig. 5. Distribución del tiempo de ejecución para el conjunto de entrada cuadrado, con 10M puntos, planificación FSC y 20K anchors.

Agradecimientos

Los autores de este trabajo agradecen los consejos de Kevin Buchin en la generación de los conjuntos de entrada y las fructíferas charlas con Pedro Ramos y Arturo González-Escribano. Los autores son parcialmente apoyados por la Junta de Castilla y León,España (VA094A08). Diego R. Llanos es parcialmente financiado por las becas del Ministerio de Educación, España (TIN2007-62302) y el Ministerio de Industria, España (FIT-350101-2007-27, FIT-350101-2006-46, TSI-020302-2008-89, CENIT MARTA, CENIT OASIS). David Orden es parcialmente financiado por las becas MTM2008-04699-C03-02/MTM y HU2007-0017. Belén Palop es parcialmente financiada por i-MATH C3-0159 y MTM2008-05043/MTM.

Referencias

[1] I. Kolingerová and J. Kohout, "Optimistic parallel Delaunay triangulation," *The Visual Computer*, vol. 18, pp. 511–529, 2002.

[2] M. Cintra and D. R. Llanos, "Toward efficient and robust software speculative parallelization on multiprocessors," in *Proc. of the SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2003, pp. 13–24.

[3] M. Cintra and D. R. Llanos, "Design space exploration of a software speculative parallelization scheme," *IEEE Trans. on Paral. and Distr. Systems*, vol. 16, no. 6, pp. 562–576, June 2005.

[4] S. Fortune, "Voronoi diagrams and Delaunay triangulations," in *Handbook of Discrete and Computational Geometry*, J. E. Goodman and J. O'Rourke, Eds., chapter 23, pp. 513–528. CRC Press, New York, 2004.

[5] S. Fortune, "A sweepline algorithm for Voronoi diagrams," *Algorithmica*, vol. 2, pp. 153–174, 1987.

[6] L. J. Guibas and J. Stolfi, "Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams," *ACM Transactions on Graphics*, vol. 4, no. 2, pp. 74–123, 1985.

[7] L. J. Guibas, D. E. Knuth, and M. Sharir, "Randomized incremental construction of Delaunay and Voronoi diagrams," *Algorithmica*, vol. 7, pp. 381–413, 1992.

[8] "CGAL, Computational Geometry Algorithms Library.," `http://www.cgal.org/`. Last access: 17-Feb-09.

[9] J.-D. Boissonnat, O. Devillers, S. Pion, M. Teillaud, and M. Yvinec, "Triangulations in CGAL," *Computational Geometry: Theory and Applications*, vol. 22, pp. 5–19, 2002.

[10] "Triangle," Software available at `http://www.cs.cmu.edu/~quake/triangle.html`. Last access: 17-Feb-09.

[11] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, 3 edition, 2008.

[12] L. Devroye, C. Lemaire, and J.-M. Moreau, "Expected time analysis for Delaunay point location," *Computational Geometry: Theory and Applications*, vol. 29, pp. 61–89, 2004.

[13] L. Devroye, E. P. Mücke, and B. Zhu, "A note on point location in Delaunay triangulations of random points," *Algorithmica*, vol. 22, pp. 477–482, 1998.

[14] E. P. Mücke, I. Saias, and B. Zhu, "Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations," in *Proceedings of the 12th ACM Symposium on Computational Geometry*, 1996, pp. 274–283.

[15] M. Gupta and R. Nim, "Techniques for run-time parallelization of loops," *Supercomputing*, November 1998.

[16] L. Rauchwerger and D. A. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 2, pp. 160–180, 1999.

[17] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew, "Scheduling strategies for optimistic parallel execution of irregular programs," in *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, New York, NY, USA, 2008, pp. 217–228, ACM.

[18] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2007, pp. 211–222, ACM.

[19] M. Cintra, J. F. Martínez, and J. Torrellas, "Architectural support for scalable speculative parallelization in shared-memory multiprocessors," in *Proc. of the 27th Intl. Symp. on Computer Architecture (ISCA)*, June 2000, pp. 256–264.

[20] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," in *Proc. of the 8th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 1998, pp. 58–69.

[21] G. E. Blelloch, J. C. Hardwick, G. L. Miller, and D. Talmor, "Design and implementation of a practical parallel Delaunay algorithm," *Algorithmica*, vol. 24, pp. 243–269, 1999.

[22] K. Buchin, "Organizing point sets.," Ph.D. Thesis, 2007, Available at `http://www.diss.fu-berlin.de/2008/118/index.html`.