Ejecución paralela de algoritmos incrementales aleatorizados

Arturo González-Escribano [*] Diego R. Llanos [*] David Orden [**] Belén Palop [*]

Resumen

Hoy en día es cada vez más fácil tener acceso a un supercomputador. Sin embargo sólo los programadores expertos, con un elevado conocimiento del problema a paralelizar y de la arquitectura de la máquina en que se desea ejecutar el programa, son capaces de diseñar algoritmos que se ejecuten en paralelo eficientemente. En este trabajo presentamos una técnica de paralelización automática que, con un mínimo esfuerzo de implementación, permite ejecutar código secuencial en paralelo en una máquina de memoria compartida con varios procesadores. Se mostrará que, en el contexto de la Geometría Computacional, los algoritmos incrementales aleatorizados son claros candidatos a beneficiarse de esta técnica. Se presentan resultados experimentales para el cálculo de la envolvente convexa y del menor círculo contenedor de nubes de puntos en 2D, que muestran aceleraciones significativas en la ejecución paralela de bucles frente a su versión secuencial.

1. Introducción

La paralelización de algoritmos de Geometría Computacional ha sido objeto de estudio en los últimos tiempos (ver, por ejemplo, [9], [13] o [20]). En su mayor parte, estos estudios se dirigen al diseño de algoritmos paralelos bajo ciertos modelos de computación que permiten analizar adecuadamente su complejidad. El hecho de que los supercomputadores se estén haciendo accesibles a un número cada vez mayor de investigadores contribuye a este interés. Sin embargo, a la hora de diseñar un algoritmo paralelo se necesita una gran experiencia así como un alto grado de conocimiento tanto del problema como de la arquitectura de la máquina. Frente a esta necesidad, las técnicas emergentes de paralelización automática y especulativa [2, 7] nos permiten ejecutar en paralelo un algoritmo y mejorar, en algunos casos, los resultados de la ejecución secuencial, con un mínimo esfuerzo de desarrollo e implementación.

En este trabajo presentamos los beneficios de la paralelización especulativa de algoritmos incrementales aleatorizados. Para ello, comenzamos recordando las características generales de estos algoritmos en la Sección 2. En la Sección 3 se estudian los problemas que presentan los algoritmos aleatorizados para su paralelización. A continuación, la Sección 4 introduce los conceptos fundamentales de la paralelización automática y, en concreto, de la paralelización especulativa. Con el fin de comprobar que la paralelización especulativa resulta especialmente adecuada para estos algoritmos, en la Sección 5 se presentan los resultados experimentales obtenidos para dos de los problemas clásicos en Geometría Computacional: el cálculo de la Envolvente Convexa y del Menor Círculo Contenedor de una nube de puntos en el plano.

2. Algoritmos incrementales aleatorizados

Los algoritmos incrementales aleatorizados han sido ampliamente estudiados en áreas como la Geometría Computacional y la Optimización [17, 18, 19]. Su uso ha permitido desarrollar algoritmos simples, fáciles de implementar y eficientes para una gran variedad de problemas, entre los que podemos citar la intersección de segmentos rectilíneos [5], los diagramas de Voronoi [5, 11], las triangulaciones de polígonos simples [24], la programación lineal [15] y muchos otros.

[*] Departamento de Informática, Universidad de Valladolid, España. {arturo,diego,b.palop}@infor.uva.es. Diego R. Llanos está parcialmente financiado por RII3-CT-2003-506079. Belén Palop está parcialmente financiada por MCYT TIC2003-08933-C02-01.
[**] Departamento de Matemáticas, Universidad de Alcalá, España. david.orden@uah.es.

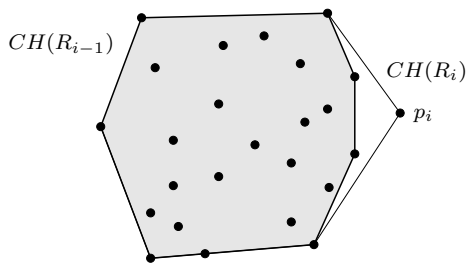

Figura 1: Inserción de un nuevo punto a la Envolvente Convexa con el algoritmo de Clarkson *et al.*

En su formulación más general, la entrada de un algoritmo incremental aleatorizado es un conjunto de elementos (no necesariamente puntos), para los que debe calcularse una cierta salida. El algoritmo procede de manera incremental, generalmente con un bucle que itera sobre todos los elementos, y su principal característica es que los elementos se van insertando según un orden aleatorio.

Para que dos iteraciones puedan calcularse simultáneamente en distintos procesadores, no deben existir dependencias entre los resultados calculados en la primera y los datos utilizados en la segunda. Independientemente del problema que resuelvan, estos algoritmos presentan un patrón similar de dependencias entre iteraciones del bucle (como veremos en detalle en la próxima sección). De manera informal podemos decir que, al principio de la ejecución, muchos elementos que se insertan cambian la solución que se está calculando, de la cual depende la inserción de los siguientes elementos. Sin embargo, a medida que la ejecución avanza, cada vez aparecen menos dependencias entre iteraciones, es decir, menos elementos modifi can la solución. Por un lado esto hace que la complejidad esperada de los algoritmos incrementales aleatorizados sea notablemente más baja que la complejidad en el caso peor. Por otra parte, esta distribución de dependencias hace de la paralelización especulativa la mejor técnica para su ejecución en paralelo.

Resumimos a continuación las construcciones incrementales aleatorizadas que se utilizarán en la experimentación: El algoritmo de Clarkson *et al.* para el cálculo de la Envolvente Convexa, y el de Welzl para calcular el Menor Círculo Contenedor de una nube de puntos en el plano.

2.1. Algoritmo de Clarkson et al. para la envolvente convexa

Dada una nube S de puntos en el plano, se denota por $CH(S)$ la lista ordenada de vértices de su envolvente convexa. El algoritmo de Clarkson *et al.* [5] consiste en un bucle principal que itera sobre cada uno de los puntos de la nube en un orden aleatorio: Dada una permutación aleatoria R de los puntos de S, llamamos R_i al conjunto de los i primeros puntos de la permutación. Supongamos que el bucle ha avanzado hasta el i-ésimo punto p_i. Si p_i está dentro de $CH(R_{i-1})$ (ya calculada), entonces $CH(R_i) = CH(R_{i-1})$ y la iteración termina. En caso contrario (ver la Figura 1), $p_i \in CH(R_i)$, por lo que se deben calcular las dos tangentes a $CH(R_{i-1})$ desde p_i, eliminar de la lista $CH(R_{i-1})$ los vértices entre los puntos de tangencia (si los hay) y añadir p_i a la lista en su sustitución.

Para el estudio de la complejidad del algoritmo, es necesario tener en cuenta que se utiliza una estructura de datos que almacena información sobre la solución parcial a medida que se va construyendo. Esta estructura permite realizar la localización de un punto en tiempo esperado $O(\log n)$. Por otro lado, es fácil ver que cada punto puede dar lugar como máximo a dos aristas que se pueden borrar una única vez, con lo que la complejidad esperada del algoritmo de Clarkson *et al.* es $O(n \log n)$.

2.2. Algoritmo de Welzl para el menor círculo contenedor

El algoritmo de Welzl [25] para calcular el menor círculo contenedor de una nube de puntos S es recursivo: ver la Figura 2 (izquierda).

La primera llamada se hace con la nube de puntos en orden aleatorio y un conjunto vacío. En el segundo conjunto se devuelven los puntos que defi nen la solución parcial obtenida hasta el momento, esto es, la circunferencia que pasa por ellos y contiene a los del primero. El algoritmo termina cuando consigue encontrar los

```
B_MINIDISCO(P,R)
  si P = ∅ o |R| = 3 entonces
    B ← R
  si no
    elegir el último p ∈ P
    B ← B_MINIDISCO(P \ {p}, R)
    si p ∉ CírculoPor(B) entonces
      B ← B_MINIDISCO(P \ {p}, R ∪ {p})
    fi n si
  fi n si
  devolver B
```

```
iMCC(P)
 B ← ∅
 para i = 1 a n
 |  si p_i ∉ CírculoPor(B) entonces
 |   B ← {p_i}
 |   para j = 1 a i − 1
 |   |  si p_j ∉ CírculoPor(B) entonces
 |   |   B ← {p_i, p_j}
 |   |   para k = 1 a j − 1
 |   |   |  si p_k ∉ CírculoPor(B) entonces
 |   |   |   B ← {p_i, p_j, p_k}
 |   |   |  fi n si
 |   |   fi n para
 |   |  fi n si
 |   fi n para
 |  fi n si
 fi n para
 devolver B
```

Figura 2: Pseudocódigos recursivo (izquierda) e iterativo (derecha) para el algoritmo de Welzl.

tres (o dos) puntos por los que pasa la circunferencia del menor círculo que contiene a la nube. Para simplificar su descripción, reescribiremos el algoritmo de Welzl de manera iterativa. El algoritmo resultante, similar al que aparece en el capítulo 4 de [6], se muestra en la Figura 2 (derecha).

Dada una nube de puntos $S \subseteq \mathbb{R}^2$, denotamos por $D(S)$ su menor círculo contenedor, definido por los, a lo sumo, tres puntos por los que pasa. Llamemos R_i a los i primeros puntos de la nube y supongamos calculado $D(R_{i-1})$. Al añadir el punto i-ésimo p_i pueden pasar dos cosas. Si p_i está dentro de $D(R_{i-1})$, entonces $D(R_i) = D(R_{i-1})$. Si, por el contrario, p_i está fuera, se debe descartar $D(R_{i-1})$. A cambio, se obtiene que p_i está en la frontera de $D(R_i)$. Así, para generar este disco debemos buscar en R_{i-1} un segundo punto de su frontera. Cada candidato $p_j \in R_{i-1}$ se selecciona de manera similar a lo hecho para p_i. Análogamente, elegido un p_j puede ser necesario buscar en R_{j-1} un tercer punto p_k.

El análisis de la complejidad del algoritmo resulta más sencillo a partir de la versión iterativa. En cada iteración del bucle gobernado por la variable i se realiza un condicional en tiempo constante y, sólo cuando éste es cierto, se entra en el siguiente bucle. Si tenemos en cuenta que el orden de los puntos es aleatorio, la probabilidad de que el i-ésimo punto p_i esté en la frontera del menor círculo que contiene a los i primeros puntos es a lo sumo $3/i$. Cuando se ejecuta el segundo bucle, en cada una de sus $i-1$ iteraciones se realiza un condicional en tiempo constante, y sólo cuando éste verifica la condición se entra en el siguiente bucle. Esto sucede cuando, una vez fijado p_i, el punto p_j está también en la frontera de $D(R_i)$, lo que tiene probabilidad a lo sumo $2/(i-1)$. Finalmente, las iteraciones del tercer bucle tienen coste constante. Por tanto, si $T(n)$ es el tiempo esperado para resolver el problema con una nube de n puntos, tenemos

$$T(n) \leq 1 + 2 + \sum_{i=3}^{n} \left(1 + \frac{3}{i}(i-1) \cdot \left(1 + \frac{2}{i-1}(i-2) \cdot 1\right)\right) \leq 1 + 2 + (1 + 3 \cdot (1 + 2 \cdot 1))\, n = 10n + 3.$$

De este modo, la complejidad esperada del algoritmo es lineal. Otro algoritmo lineal, no aleatorizado, se puede encontrar en [16].

3. Análisis de dependencias en algoritmos incrementales aleatorizados

Con el fin de analizar el patrón de dependencias entre iteraciones de un bucle en los algoritmos incrementales aleatorizados, consideremos S y $\phi(S)$ respectivamente su entrada y su salida. Como ya se ha comentado, estos algoritmos comienzan eligiendo una permutación aleatoria $R = \{p_1, \ldots, p_n\}$ de los elementos de S, para después construir de manera incremental $\phi(R_i)$ para $R_i := \{p_1, \ldots, p_i\}$.

Para estudiar el número esperado de dependencias que aparecen en un determinado paso i, se denomina *violadores* a aquellos elementos de S que no han sido procesados todavía y que cambiarían la salida $\phi(R_i)$ actual, esto es, los elementos que generan una dependencia respecto de todos los elementos posteriores. Por otro lado, los elementos *extremos* son aquéllos que defi nen la salida $\phi(R_i)$.

De un modo más formal, los conjuntos de violadores y elementos extremos para R_i en S quedan defi nidos como:

$$V(R_i) := \{s \in S \setminus R_i : \phi(R_i \cup \{s\}) \neq \phi(R_i)\} \quad \text{y} \quad X(R_i) := \{s \in R_i : \phi(R_i \setminus \{s\}) \neq \phi(R_i)\}.$$

Sea por ejemplo $S \subset \mathbb{R}^2$ un conjunto de puntos. Si consideramos $\phi \equiv$ `EnvolventeConvexa`, los violadores $V(R_i)$ son aquellos puntos fuera de `EnvolventeConvexa`(R_i), mientras que los puntos extremos $X(R_i)$ son los vértices de esta Envolvente Convexa. Para $\phi \equiv$ `MenorCírculoContenedor`, los puntos fuera del Menor Círculo Contenedor de R_i son los violadores $V(R_i)$, y los puntos extremos $X(R_i)$ son los que defi nen este círculo.

El siguiente Lema, cuya demostración puede encontrarse en [8], establece una relación entre el número esperado de elementos extremos entre los $i+1$ primeros y el número esperado de violadores a partir de ese elemento:

Lemma 3.1. (de muestreo). *Sean $v_i := E(|V(R_i)|)$ y $x_i := E(|X(R_i)|)$ los números esperados de violadores y elementos extremos para R_i como antes. Entonces, para cualquier $i \in \{1, \ldots, n\}$,*

$$\frac{x_i}{i} = \frac{v_{i-1}}{n-i+1}$$

Por tanto, en un paso cualquiera i, el número esperado de dependencias es $\frac{x_i}{i}(n-i+1)$ y así la probabilidad de encontrar una dependencia en el paso i es $\frac{x_i}{i}$. Los algoritmos incrementales aleatorizados se utilizan cuando x_i es asintóticamente menor que i y por tanto esta probabilidad decrece a medida que la ejecución avanza. Para el ejemplo anterior de $\phi \equiv$ `MenorCírculoContenedor`, es claro que $x_i \leq 3$ para cualquier i. Por tanto, el número esperado de puntos fuera del menor círculo que contiene a R_{i-1} es

$$v_{i-1} \leq \frac{3}{i}(n-i+1).$$

En otras palabras, la probabilidad de encontrar una dependencia en el paso i es a lo sumo $\frac{3}{i}$, como ya vimos.

Para el caso de $\phi \equiv$ `EnvolventeConvexa` se conoce, ver [22, 23], que $x_i \in O(k \log i)$ para puntos uniformemente distribuidos en un k-gono y $x_i \in O(\sqrt[3]{i})$ para puntos uniformemente distribuidos en un disco. En ambos algoritmos, el hecho de que x_i decrezca rápidamente cuando i se hace más grande muestra por qué la probabilidad de encontrar una dependencia es mucho mayor en las primeras iteraciones que en el resto.

4. Paralelización automática y especulativa

Los mecanismos de paralelización automática buscan extraer el paralelismo que es inherente a muchos algoritmos secuenciales. Los compiladores paralelizadores se encargan de modifi car automáticamente el código fuente para su ejecución en un sistema paralelo. La técnica de paralelización automática más extendida es la *paralelización a nivel de bucle*. En ella, el compilador asigna a cada procesador un subconjunto de iteraciones de un bucle para su ejecución paralela. El compilador sólo puede aplicar esta técnica si está seguro de que ninguna iteración del bucle depende de otras, es decir, que todas las iteraciones pueden calcularse en cualquier orden. Si existen *dependencias* entre iteraciones (que es el caso más frecuente), incluso los compiladores más modernos se abstienen de intentar paralelizar el bucle.

La *paralelización especulativa* es una de las técnicas más prometedoras para extraer el paralelismo de bucles con dependencias [2, 12, 21]. En ella los bloques de iteraciones se ejecutan en paralelo de manera "optimista", esto es, suponiendo que no hay dependencias entre iteraciones. Al mismo tiempo, un mecanismo (que puede ser software o hardware) se encarga de comprobar que los resultados de la ejecución del bucle en paralelo no violan la semántica de la versión secuencial. Dado que es probable que las iteraciones del bucle se ejecuten en un orden

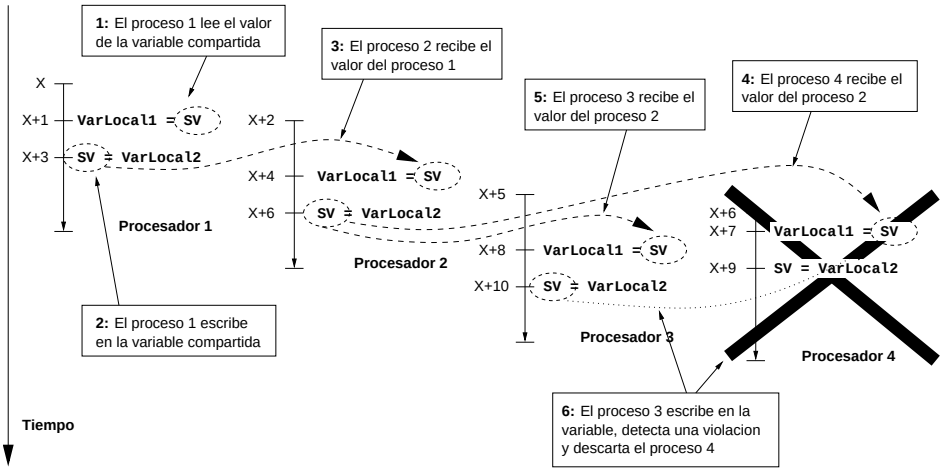

Figura 3: Paralelización especulativa.

distinto del original, puede ocurrir que un procesador utilice un valor de una variable (que denominaremos SV) que debería haberse generado en una iteración anterior pero que aún no ha sido calculado. Esta *violación de dependencia* se descubrirá al calcular el valor correcto de SV y al comprobar si algún procesador ha utilizado el valor antiguo de la variable. De ser así, se interrumpe la ejecución del procesador que ha utilizado el dato incorrecto, descartándose los valores calculados por éste y los siguientes, y reiniciando su ejecución con el valor correcto de SV. Ver la Figura 3. Es fácil observar que el hecho de interrumpir la ejecución y descartar los valores calculados por algunos procesadores afecta de manera negativa a la velocidad de la ejecución.

Cintra y Llanos [2] presentaron en 2003 un mecanismo software que ejecuta especulativamente en paralelo bucles no analizables en tiempo de compilación. Para ello, es necesario añadir al código original una serie de llamadas que se encargan de acceder a la estructura de datos compartida por los distintos procesadores, y de monitorizar la ejecución paralela del bucle. Una descripción detallada de estos cambios, que podría realizar de manera automática un compilador, se puede encontrar en [4].

De entre los algoritmos con bucles no analizables en tiempo de compilación, los algoritmos incrementales aleatorizados resultan especialmente interesantes para su paralelización especulativa. En la Sección 3 hemos visto que, para estos algoritmos, las dependencias aparecen principalmente al comienzo de la ejecución y van siendo cada vez menos probables. Esta distribución reduce, con respecto al caso peor, el coste esperado de interrumpir la ejecución y descartar los valores calculados cuando se produce una violación de dependencia. Como veremos en la Sección 5, la ejecución especulativa permite acelerar su ejecución frente a la ejecución secuencial, con un mínimo esfuerzo de implementación.

5. Resultados experimentales

Con el fin de evaluar el rendimiento de los mecanismos de paralelización especulativa, hemos paralelizado especulativamente los bucles principales de los algoritmos de Clarkson *et al.* y de Welzl, descritos en las secciones 2.1 y 2.2 respectivamente. Los bucles elegidos para su paralelización han sido los siguientes:

- El bucle principal del algoritmo de Clarkson *et al.*, que consume el 100 % del tiempo de ejecución del algoritmo.
- El bucle exterior (i) de la versión iterativa del algoritmo de Welzl, que consume también el 100 % del tiempo de ejecución secuencial.
- El bucle intermedio (j) de la versión iterativa del algoritmo de Welzl, que para el conjunto de datos de entrada utilizado consume cerca del 79 % del tiempo de ejecución.
- El bucle interior (k) de la versión iterativa del algoritmo de Welzl, que para el conjunto de datos de entrada utilizado consume aproximadamente el 45 % del tiempo de ejecución.

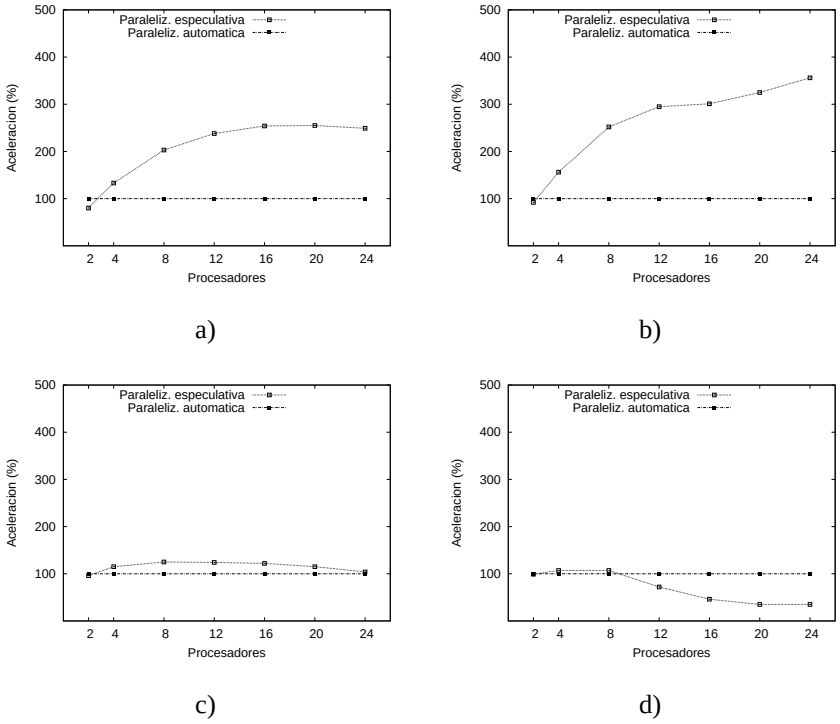

Figura 4: Paralelización especulativa y automática de bucles de algoritmos aleatorizados. a) Bucle principal del algoritmo de Clarkson *et al.* b), c) y d) Bucle interior, intermedio y exterior, respectivamente, del algoritmo de Welzl.

Las versiones secuenciales y paralelas de estos algoritmos se han ejecutado en un sistema multiprocesador de memoria compartida Sunfi re 15K, equipado con procesadores UltraSparc-III a 900 MHz. Como conjunto de datos de entrada se ha utilizado un conjunto de diez millones de puntos distribuidos uniformemente en un disco. El conjunto se ha creado utilizando el generador de puntos aleatorios que ofrece CGAL 2.4 [1], aleatorizándolo a través de su función *shuffle*.

Hemos comparado el rendimiento de los mecanismos de paralelización especulativa con respecto a la paralelización automática que ofrece el compilador Forte de Sun, basada en análisis de dependencias en tiempo de compilación. La Figura 4 muestra los resultados en terminos de aceleración con respecto a la versión secuencial de los mismos bucles. Dado que ninguno de los bucles considerados son analizables en tiempo de compilación, el compilador Forte no es capaz de paralelizarlo, por lo que el rendimiento de las versiones paralelas generadas es el mismo que el de las secuenciales. Por el contrario, las versiones especulativas ofrecen mejoras que llegan al 255 % con 20 procesadores en el caso del algoritmo de Clarkson *et al.*, y al 356 % con 24 procesadores para el bucle interior del algoritmo de Welzl. Sin embargo, los mecanismos de paralelización especulativa no obtienen buenos rendimientos para los bucles exterior e intermedio del algoritmo de Welzl. El motivo es simple: si el punto p_i no pertenece a $D(R_{i-1})$, el procesador encargado de esa iteración debe calcular la circunferencia que pasa por p_i y contiene a los $i-1$ primeros puntos. Para i sufi cientemente grande, esto supone un fuerte desequilibrio de la carga de trabajo entre procesadores, lo que explica el bajo rendimiento que se observa en las fi guras c) y d). En [14] se pueden encontrar más detalles sobre la estrategia más adecuada de distribución de carga de trabajo para algoritmos incrementales aleatorizados. Con esta estrategia las iteraciones se agrupan en bloques pequeños al principio de la ejecución, aumentando progresivamente su tamaño. Por otra parte, en [3] aparece información detallada sobre la paralelización especulativa de la envolvente convexa sobre diferentes conjuntos de entrada. Sus resultados frente al resto de técnicas de paralelización se analizan en [10].

6. Conclusiones

En este trabajo hemos mostrado las ventajas del uso de la paralelización especulativa. Hemos visto que el caso de los algoritmos incrementales aleatorizados la paralelización automática no sólo es posible, sino que permite acelerar signifi cativamente la ejecución. Esta técnica permite extraer el paralelismo de bucles no analizables en tiempo de compilación, y que suponen un desafío para los paralelizadores actuales.

Agradecimientos

Parte de este trabajo se realizó durante una estancia de David Orden en el Departamento de Informática de la Universidad de Valladolid y financiada por la Universidad de Alcalá. Los autores agradecen a Manuel Abellanas sus comentarios sobre la envolvente convexa, y al EPCC (Edinburgh Parallel Computing Center) de la Universidad de Edimburgo, Escocia, por el uso de sus recursos de cómputo.

Referencias

[1] CGAL, Computational Geometry Algorithms Library. `http://www.cgal.org/`.

[2] M. Cintra and D. R. Llanos. Toward effi cient and robust software speculative parallelization on multiprocessors. In *Proceedings of the SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2003.

[3] M. Cintra and D. R. Llanos and B. Palop. Speculative Parallelization of a Randomized Incremental Convex Hull Algorithm. *ICCSA 2004: Proc. Intl. Conf. on Computer Science and its Applications*, Perugia, Italy, May 2004, LNCS 3045.

[4] M. Cintra and D. R. Llanos. Design space exploration of a software speculative parallelization scheme. *IEEE Transactions on Parallel and Distributed Systems, Vol. 16, no. 6, June 2005.*

[5] K. L. Clarkson and P. W. Shor. Applications of random sampling in Computational Geometry, II. *Journal of Discrete and Computational Geometry*, pages 387–421, 1989.

[6] M. De Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry, Algorithms and Applications*. Springer-Verlag, 1997. ISBN 354061270X.

[7] R. P. Garg and I. Sharapov. Techniques for Optimizing Applications. *Sun Blueprints, Prentice Hall*, 2002.

[8] B. Gärtner and E. Welzl. A simple sampling lemma: Analysis and applications in Geometric Optimization. *Discrete and Computational Geometry*, 25(4):569–590, 2001.

[9] A. H. Gebremedhin and F. Manne. Scalable parallel graph coloring algorithms. *Concurrency: Practice and Experience*, 12(12):1131–1146, 2000.

[10] A. González-Escribano, D. R. Llanos, D. Orden and B. Palop. Parallelization alternatives and their performance for the Convex Hull problem. Preprint, April 2005.

[11] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–224, 1992.

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

[13] J. C. Hardwick. Implementation and evaluation of an effi cient parallel Delaunay triangulation algorithm. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*, pages 239–248, June 1997.

[14] D. R. Llanos, D. Orden and B. Palop. MESETA: A new scheduling strategy for speculative parallelization of randomized incremental algorithms. *HPSEC-05 (7th Workshop on High Performance Scientific and Engineering Computing), hold in conjunction with ICPP-05*. Norway, June 2005, IEEE Press.

[15] J. Matoušek, M. Sharir, and E. Welzl. A subexponential bound for linear programming. *Algorithmica*, 16:498–516, 1996.

[16] N. Megiddo. Linear-time algorithms for linear programming in $\mathbb{R}^3$ and related problems. *SIAM Journal on Computing*, 12(4):759–776, 1983.

[17] K. Mulmuley. Randomized algorithms in Computational Geometry. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter 16, pages 703–724. North-Holland Publishing Co., 2000.

[18] K. Mulmuley and O. Schwarzkopf. Randomized algorithms. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 34, pages 633–652. CRC Press, New York, 1997.

[19] S. Rajasekaran, P. M. Paradalos, J. H. Reif, and J. D. Rolim, editors. *Handbook of Randomized Computing: Volumes I and II*, volume 9 of *Combinatorial Optimization*. Kluwer Academic Publishers, 2001.

[20] S. Ramaswami. Parallel randomized techniques for some fundamental geometric problems: A survey. In *IPPS/SPDP Workshops*, pages 400–407, 1998.

[21] 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*, 10(2):160–180, 1999.

[22] H. Raynaud. Sur l'enveloppe convexe des nuages de points aléatoires dans $\mathbb{R}^n$. *Journal of Applied Probability*, 7:35–48, 1970.

[23] A. Renyi and R. Sulanke. Über die konvexe hülle von n zufällig gerwählten punkten II. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, 3:138–147, 1964.

[24] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry: Theory and Applications*, 1:51–64, 1991.

[25] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In *New results and new trends in computer science*, volume 555 of *Lecture Notes in Computer Science*, pages 359–370. Springer-Verlag, 1991.