VNIVERSIDAD NACIONAL
AVTNOMA DE
MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

SIMULACIÓN Y ANÁLISIS DE AGREGADOS EN MODELOS COMPUTACIONALES DE CRECIMIENTO FRACTAL

TESIS
QUE PARA OPTAR POR EL GRADO DE:

DOCTOR EN CIENCIAS (COMPUTACIÓN)

PRESENTA:
JESUS ANTONIO SOSA HERRERA

DIRECTORA DE TESIS: **DRA. SUEMI RODRÍGUEZ ROMO**

FACULTAD DE ESTUDIOS SUPERIORES CUAUTITLÁN

COMITÉ TUTORAL:

DR. HÉCTOR BENITEZ PÉREZ,
DR. FABIAN GARCÍA NOCETTI.

INSTITUTO DE INVESTIGACIONES EN MATEMÁTICAS APLICADAS Y SISTEMAS

MÉXICO, D. F. Agosto 2013

INDICE GENERAL.

INDICE DE FIGURAS.

INDICE DE TABLAS.

I. INTRODUCCION.

I.1 Descripción general.

El tema de estudio en esta tesis se refiere a la simulación de modelos computacionales para crecimiento fractal. En muchos fenómenos naturales y sociales se pueden observar diversos patrones con intrincadas estructuras que, no obstante su complejidad, pueden ser representados por modelos de fractales relativamente sencillos. Entre estos modelos cabe destacar a aquellos que están basados en crecimiento fractal agregación. Históricamente, el desarrollo de la geometría fractal fue limitada por la imposibilidad de realizar la enorme cantidad de cálculos involucrados [1], y no es sino hasta las últimas cuatro décadas cuando se ha podido desarrollar plenamente gracias a la ayuda de los sistemas de cómputo electrónicos modernos. Estos conceptos han resultado útiles para la descripción y entendimiento de varios fenómenos, correspondientes a ciertas áreas de interés de la física, química, biología, economía, etc. En dichas áreas, un enfoque descriptivo utilizando matemáticas de manera tradicional suele resultar incompleto, o bien impráctico, debido a la complejidad inherente de los fenómenos que se pretende explicar. La geometría fractal, en cambio, trata de encontrar características de autosimilaridad en los fenómenos, modelos u objetos geométricos, explicándolos a través de relaciones exponenciales que rigen el comportamiento de los mismos. El desarrollo de las computadoras modernas, así como de algoritmos eficientes para las mismas, ha permitido que la geometría fractal pueda utilizarse en diversos campos de la ciencia y la tecnología, pues la evaluación de características en ciertos patrones demasiado irregulares y complejos que a menudo se presentan como objetos de estudio de aquella, resulta prácticamente imposible realizar en forma manual o mental. Por tanto, la evolución de la geometría fractal ha estado ligada a la misma evolución de los equipos de cómputo.

La aplicación práctica de estos modelos va desde la descripción de fenómenos naturales diversos, como son la electrodeposición, crecimiento bacteriano, crecimiento urbano, crecimiento cancerígeno, por citar algunos; hasta aplicaciones teóricas como lo es la descripción de comportamientos de cambios de fase. Entre algunos de los modelos más representativos de crecimiento fractal podemos destacar la Agregación por Difusión Limitada (DLA) [2] el Modelo de Eden [3], Modelo de Rompimiento Dieléctrico (DBM) [4], mapeos conformes de Hastings-Levitov [5], y diversos tipos de crecimiento basados en agentes [6,7].

Concretamente, en este trabajo se abordan las técnicas de computo científico modernas (como son las estructuras de datos espaciales multidimensionales, algoritmos eficientes de calculo de entropía y algoritmos de búsqueda multidimensional) aplicadas a determinados aspectos del crecimiento fractal. Específicamente, se utiliza el procesamiento paralelo y multitarea sobre diferentes plataformas que incluyen *clusters* de supercómputo (*HP 4000 High Performance Cluster –Kanbalam-*), servidores de multiproceso de alto rendimiento con memoria compartida (Power Edge 2900) y procesamiento sobre tarjetas gráficas con capacidades de computo de propósito general GPGPU usando la arquitectura CUDA (Nvidia GTX 460).

Con las técnicas y herramientas de procesamiento paralelo (MPI, OMP, posix thread y CUDA), desarrollo de algoritmos e implementación sobre hardware de multiprocesamiento mencionado anteriormente, se consiguió cumplir con los siguientes objetivos:

Se generaron objetos fractales *diffusion limited aggregation* (DLA) del orden de 10^10 partículas en 2 dimensiones, y de hasta 10^9 partículas en 3 dimensiones, determinándose en cada caso su dimensión fractal. De forma similar, se generaron agregados en espacios euclidianos de 4,5 y 6 dimensiones. Sin embargo, el tamaño en estos últimos casos (100,000 partículas) no puede ser considerado para cálculos precisos de dimensionalidad.

Se analizaron propiedades de lagunaridad [1] y multifractalidad [8] en agregados de hasta 10^9 partículas en 2D y 10^8 partículas en 3D. Con lo que se pudo establecer la conclusión de que los agregados DLA siguen una distribución geométrica mono fractal a pesar de que la distribución de probabilidad de crecimiento en sus puntos delimitantes es multifractal [9].
Otra de las aportaciones expuestas en este documento consiste en el desarrollo del Algoritmo de Búsqueda con Precisión Variable sobre *quadtrees*, la cual permite prescindir de los mapas de bits que suelen utilizarse al realizar búsquedas espaciales sobre este tipo de estructuras cuando se tiene una gran cantidad de datos [10-12]. Esto acelera el proceso de búsqueda sobre el *quadtree*, pues no es necesario acceder ni actualizar información sobre los mapas de bits, solamente se considera cierta información adicional acerca de los nodos, la cual puede ser calculada al momento de atravesar los nodos en el proceso de búsqueda, o calculada previamente y almacenada en forma de tabla (*look up table*). Otro algoritmo que se presenta en esta tesis es una modificación del método de conteo de cajas deslizantes (*sliding box*) [13], la implementación aquí presentada extiende la técnica de Tolle et al [14] hacia todas las dimensiones en las que se define el espacio a analizar. El rendimiento así obtenido resulta bastante considerable. Esto permitió generar agregados fractales suficientemente grandes como para realizar un análisis asintótico de su comportamiento multifractal y de lagunaridad.

En adición a lo anterior, en esta tesis se introduce una nueva metodología para determinar la entropía configuracional para un modelo de crecimiento geométrico. Esta técnica parte de una representación binaria de los objetos generados por el modelo, la cual permite el empleo un proceso de clasificación que consiste en la aplicación del método *Entropy Based Fuzzy C-Means* [15]. Una vez obtenida esta clasificación, nuestro método utiliza la distancia de Hamming para determinar la entropía del modelo que genera a los objetos en términos de la definición establecida por la teoría de la información de Shannon[16]. Al calcular la entropía para cada clase, obtenemos una aproximación de la entropía de los elementos que están dentro de la clase. Con esto se consiguió estimar la entropía configuracional para modelos de crecimiento en 2 dimensiones, lo que permite describir el comportamiento los modelos a través del tiempo en función de ciertos parámetros establecidos en los modelos de crecimiento.

Dentro del contexto descrito, se puede resumir que la presente tesis se aborda y resuelve de manera satisfactoria los siguientes problemas:

- Creación de un algoritmo de búsqueda de precisión variable sobre *quadtrees* y sus generalizaciones multidimensionales –*octrees y kd-trees* paralelización por dominios espacial del *quadtree*. Con este algoritmo se logró construir los fractales DLA mas grandes publicados a la fecha[30]

- Paralelización por dominios de tal forma que el programa de búsqueda de precisión variable se ejecutó en una arquitectura de procesamiento distribuido (*Kan Balam*) utilizando hasta 64 procesadores distribuidos en 16 nodos en forma eficiente.

- Se logró determinar la el comportamiento de la multifractalidad de masa para agregados DLA mediante una implementación eficiente [64] de la versión multidimensional método *gliding box* [14].

- Creación del algoritmo para la determinación de la entropía en conjuntos espaciales basada en el concepto de entropía de Shannon y el método de *Fuzzy C-Means*. Este algoritmo fue utilizado con éxito en aplicaciones prácticas de modelación urbana[51]

- Elaboración y simulación eficiente de modelos híbridos de crecimiento fractal y la implementación en paralelo utilizando CUDA y múltiples hilos para crear un proceso de *pipelining* GPU – CPU [51].

I.2 Revisión bibliográfica de trabajos previos.

La geometría fractal es una herramienta de gran utilidad para el estudio de muchos fenómenos físicos, una revisión general de este tema, asi como de su estado del arte actual se puede obtener en las referencias [1][23][24][35][37][58]. Debido a su aplicación práctica, la simulación de objetos fractales es de gran interés, sin embargo, por la naturaleza asintótica de la geometría fractal, es necesario generar conjuntos fractales de gran tamaño para que tales simulaciones aporten datos que permitan extraer conclusiones validas para su aplicación a fenómenos complejos. El tipo de fractales aquí estudiado se encuentran dentro de la categoría de '*modelos de crecimiento fractal*' [24] y su simulación y análisis presenta la problemática de la gran catidad de cálculos y espacio en memoria necesarios para su generación y almacenamiento.

Diversas técnicas con las se ha abordado esta complejidad pueden encontrarse en [12][24][39][41][61] para el caso de la generación de modelos DLA, estas técnicas son significativamente mejoradas por la metodología presentada en el capitulo V, el cual incluye medidas de rendimiento.

De las formas de obtener análisis mulfractal sobre conjuntos, puede revisarse en las referencias [8][17][18][19][24][57][60] [61][62], En particular las variantes que se aplican y comparan en este trabajo están en las referencias [8][18][19] y [60].

Lacunaridad. Un estudio sobre esta medida complemetaria fue propuesta conceptualmenete por primera vez en [1]. Las principles técnicas para obtenerla se encuentran en [13][14][20][25][52][53][54][55][56][61] En el capitulo VI se propone una variante de estas técnicas que permite aplicarlas sobre conjuntos grandes.

En cuanto al análisis de entropía [16], el presente trabajo es pionero en aplicar el algoritmo Entropy fuzzy c-means [15] para la caracterización de agregados fractales.

Las referencias especificas donde se pueden revisar los detalles técnicos tanto de algoritmos como de implementación en sistemas de computo paralelo, multihilo, multiproceso y uso de GPGPUs son [10][11][16][21][26][27][28][29][36][37][42][43][44][45][46][47][48][49] y [50]. Con respecto a diversos aspectos técnicos de cómputo, muchas bibliografías importantes utilizadas aquí fueron omitidas en el capítulo de referncias, por consideralas del dominio general del área de ciencia e ingeniería de la computación, indicando solamente las ya mencionadas para especificar detalles particulares de implementación.

I.3 Contenido de los capítulos.

La presentación de este trabajo de investigación se desarrolla a lo largo de los siguientes capítulos:

I.INTRODUCCION. La presente introducción.

II. MARCO TEORICO. Define los conceptos de fractalidad, multifractalidad y lagunaridad, que son el objeto de estudio de la presente.

III. DISTINTAS DEFINICIONES DE DIMENSION. En este capitulo se abordan de las definiciones formales de autosimilaridad y dimensiones fractales como son la dimensiones de Hausdorff, de similaridad, conteo de cajas, masa-radio, correlacion densidad-densidad. Asi también de los conceptos formales de lagunaridad y multifractalidad. Se proporcionan los métodos bien conocidos para estimarlos. El capitulo concluye con una presentación formal del concepto de entropía configuracional.

IV. PLANTEAMIENTO DEL PROBLEMA IMPORTANCIA. Describe los principales problemas abordados y el porque es importante resolverlos. Enuncia los objetivos que se plantearon para esta investigación.

V. SIMULACION DE LOS MODELOS DE CRECIMIENTO FRACTAL. Aquí se proporcionan los detalles de implementación elaborados para consguir la simulación a gran escala de los modelos fractales y en su caso la generación de un gran número de ejecuciones de los algoritmos que generan los modelos fractales con el fin de obtener estadísticas significativas sobre el comportamiento de los mismos. Se detallan también los métodos de visualización. Este capitulo presenta la técnica de determinación de distancias con presicion variable dentro de un *quadtree*, demostrando el teorema que valida la correctitud del mismo. Los modelos simulados en plataformas de multiprocesamiento, multihilo y uso de GPU para cálculos de procesamiento general, incluyen al modelo de Eden, modelos de percolación, modelo de agregación por difusión limitada, modelo de rompimiento dieléctrico y modelos hibridos basados en agentes.

VI. ANALISIS DE MODELOS. En este apartado se desarrollan métodos avanzados de análisis fractal, multifractal y de lagunaridad, incluyendo implementaciones introducidas por primera vez en esta investigación. La complejidad de estos análisis radica en la magnitud de los conjuntos sobre los cuales se aplica, asi como en el número de repeticiones de cada análisis. El capítulo presenta la implementación eficiente del algoritmo de cajas deslizantes para lagunaridad y un método de paralelización del mimo por dominios. Se muestran varias implementaciones para el análisis de multifractalidad y se presenta el método de estimación de entropía configuracional basado en la separación de clases, el cual hace uso de la distancia Hamming y es una mas de las propuestas originales de esta tesis.

II. MARCO TEORICO.

En este capítulo se da la descripción formal de los conceptos involucrados para el análisis de los modelos computacionales del crecimiento fractal. Algunos de estos conceptos son de aplicación estándar en el área de estudio, mientras que otros, como son la lagunaridad, multifractalidad y entropía configuracional, no obstante su enorme potencial para describir objetos geométricos complejos, han tenido una aplicación relativamente limitada, debido al hecho de que se requiere una gran cantidad de información para poder estimarlos con suficiente exactitud. Lo anterior es debido al hecho de que los fractales, como objetos matemáticos están definidos para límites infinitos, mientras que los modelos computacionales generan elementos geométricos discretos y de tamaño finito. Esto puede llevar a una estimación errónea de parámetros sensibles al tamaño de los objetos que se estudian y a la precisión aritmética manejada por los equipos de cómputo.

II.1 Fractalidad, multifractalidad y lagunaridad.

La geometría fractal tiene como principal objetivo describir las propiedades de objetos que presentan características de autosimilaridad, esto es propiedades que permanecen invariantes bajo diferentes escalas. La principal herramienta para obtener tal descripción se encuentra determinada por la dimensión fractal, que indica la relación exponencial entre una medida sobre el objeto y su invariancia a diferentes escalas. Esta invariancia puede ser exacta para fractales determinísticos, o estadística, para fractales aleatorios. Existe diferentes definiciones de dimensión fractal, la más aceptada es la dimensión de Hausdorff, sin embargo, involucra en su definición subconjuntos infinitos con ínfimos y supremos definidos, por lo que en la práctica, cuando se consideran fractales aleatorios, se utilizan diferentes técnicas para encontrar tales valores, entre otros, tenemos, el método de conteo de cajas, relaciones masa-radio, y relaciones de correlación densidad-densidad.

El concepto de multifractalidad extiende el análisis fractal de los objetos fractales o no fractales hacia las medidas que pudieran definirse sobre ellos [8]. El análisis de multifractalidad nos permite obtener el conjunto de exponentes de escalamiento que intervienen en un fenómeno autosimilar complejo. Existen varias técnicas para determinar la multifractalidad de un objeto en este proyectos se utilizan El método de momentos descrito por Halsey, *et al*. [17] está basado en una función de partición, el método de caja de arena [18] que se basa en escalamiento exponencial de la distribución de una medida, y el método de caja extendida [19], que es similar al de Halsey, pero toma una medida extendida para evitar efectos de borde.

Otra medida importante que describe a los objetos fractales es la lagunaridad, conceptualizada por primera vez por B. Mandelbrot [1] al considerar que existen objetos con la misma dimensión fractal, pero con una estructura geométrica completamente diferente. La lagunaridad considera los espacios vacíos entre el espacio y un objeto fractal y puede definirse como el grado de invariancia traslacional de un objeto [20].

III. DISTINTAS DEFINICIONES DE DIMENSION.

La geometría fractal tiene como principal objetivo describir las propiedades de objetos que presentan características de autosimilaridad, esto es propiedades que permanecen invariantes bajo diferentes escalas. La principal herramienta para obtener tal descripción se encuentra determinada por la dimensión fractal, que indica la relación exponencial entre una medida sobre el objeto y su invariancia a diferentes escalas. Esta invariancia puede ser exacta para fractales determinísticos, o estadística, para fractales aleatorios. Existen diferentes definiciones de dimensión fractal, la más aceptada es la dimensión de Hausdorff, sin embargo, involucra en su definición subconjuntos infinitos con ínfimos y supremos, por lo que en la práctica, cuando se consideran fractales aleatorios, se utilizan diferentes técnicas para encontrar tales valores, entre otros, tenemos, el método de conteo de cajas, relaciones masa-radio, y relaciones de correlación densidad-densidad.

III.1 Autosimilaridad.

La dimensión fractal trata de caracterizar el grado de autosimilaridad a diferentes escalas de un objeto dentro del espacio en el que se encuentra definido. Esta autosimilaridad puede ser exacta, o bien estadística. Es decir, puede ser que ninguna parte del objeto en estudio corresponda exactamente a ninguna otra parte bajo ninguna escala, y sin embargo, se encuentren similitudes estadísticas o patrones con cierta aleatoriedad que se repiten a diferentes escalas del objeto.

III.2 Dimensión de Hausdorff.

Una forma de caracterizar a un objeto fractal, es comprobando su *dimensión de Hausdorff*. De hecho, algunas veces se utiliza como definición de objeto fractal como aquel objeto para el cual su *dimensión euclideana* es menor que su *dimensión de Hausdorff*.

La *dimensión de Hausdorff* de un objeto se obtiene a partir de la *medida de Hausdorff* que, de alguna manera, determina la extensión de subconjuntos de $\Re^n$ que presentan geometrías complicadas, haciendo uso de cubiertas cerradas.

De manera formal la definición de medida y dimensión de Hausdorff, se pueden expresar de la siguiente manera [21]:

Sea $U \subseteq \Re^n$ con $U \neq \varnothing$, el *diámetro* de U se define como:

$$|U| = \sup\{|x - y| : x, y \in U\} \qquad (1)$$

Si $\{U_i\}$ es una colección contable (o finita) de conjuntos con diámetros de a lo más ε, que cubren a $F \subseteq \Re^n$, esto es $F \subset \cup_{i=1}^{\infty} U_i$, con $0 < |U_i| \leq \varepsilon$ para cada i, entonces se dice que $\{U_i\}$ es una $\varepsilon - cobertura$ de F.

Suponiendo que s es un número no negativo, entonces, para cada $\varepsilon > 0$ se define:

$$\mathrm{H}_{\varepsilon}^{s}(F) = \inf\{\sum_{i=1}^{\infty} |U_i|^s : \{U_i\} \textit{ es una } \varepsilon - \textit{cobertura de } F\} \tag{2}$$

La *medida* s *- dimensional de Hausdorff* del conjunto F se define como:

$$\mathrm{H}^{s}(F) = \lim_{\varepsilon \to 0}(\mathrm{H}_{\varepsilon}^{s}(F)) \tag{3}$$

La *dimensión de Hausdorff*, también conocida como *dimensión Hausdorff-Besicovitch* se define por:

$$\dim_H F = \inf\{s \geq 0 : \mathrm{H}^{s}(F) = 0\} = \sup\{s : \mathrm{H}^{s}(F) = \infty\} \tag{4}$$

Una manera intuitiva de ver la dimensión de *Hausdorff* es considerarla como el exponente en el cual ocurre una inflexión en la obtención de la medida de un conjunto. Para valores menores a dicho exponente la medida será cero, y para valores mayores divergirá a infinito. Por ejemplo, consideremos un objeto bidimensional del cual sabemos que tiene un área finita $a \neq 0$ (por simplificar la explicación consideremos a la medida discretizada), es decir, conocemos $a = \sum |U_i|^s$ requerida en la ecuación (2) con un valor de $s = 2$ expresada en unidades elementales cuadradas $|U_i|^2$. Si quisiéramos obtener una medida del objeto utilizando elementos de medida inapropiados, observaríamos como la medida tiende a infinito, cuando se mide con elementos de dimensión menor que la del objeto, o bien, resultaría en una media con valor cero cuando se usaran elementos de dimensión mayor a la del objeto. De esta manera, vemos que el objeto bidimensional presentará una longitud infinita, al ser comparado con elementos unidimensionales $|U_i|^1$, o bien un volumen cero, al ser comparado con elementos tridimensionales $|U_i|^3$. Entonces $s = 2$ es el exponente crítico para el cual la medida del objeto cambia de cero a infinito, tomando en dicho punto crítico un valor $0 < a < \infty$. Tal punto se obtiene analíticamente mediante la ecuación (4).

III.3 Dimensión de similaridad.

Existen otras definiciones alternativas de dimensionalidad para objetos fractales además de la dimensión de *Hausdorff*, Estas son ampliamente utilizadas debido a que, aunque $\dim_H F$ se define para cualquier subconjunto de $\Re^n$ en general, esta definición no siempre es aplicable en la práctica, pues la determinación de valores ínfimos y supremos requiere de una expresión analítica que describa al objeto en cuestión, la cual no siempre está disponible; por lo tanto, para propósitos de cálculo y experimentación, suelen utilizarse otras definiciones de dimensión, como por ejemplo la dimensión de similaridad.

La dimensión de similaridad D_s se aplica a objetos exactamente autosimilares y es una generalización de nuestra idea intuitiva de dimensión (dimensión euclidiana). Consiste en evaluar el número N de copias exactas que hay en un objeto de sí mismo escaladas por un factor de ε. Tomando en cuenta que la medida m del objeto debe cumplir con la expresión:

$$m \approx N\varepsilon^{D_s} \tag{5}$$

como ocurre en el caso de la dimensión euclidiana, vemos que podemos encontrar una constante adecuada que convierte a (5) en:

$$1 = N\varepsilon^{D_s} \tag{6}$$

tomando logaritmos obtenemos

$$0 = \log(N) - D_s \log(1/\varepsilon)$$

Basándose en esto, la definición de dimensión de similaridad se extiende a

$$D_s = \frac{\log(N)}{\log(1/\varepsilon)} \tag{7}$$

Ejemplo: Dimensión de similaridad de la curva de Koch.

La curva de Koch se obtiene iterando sobre una línea recta, dividiendo a esta en tres partes iguales, removiendo la parte central y sustituyéndola por dos copias de longitud igual a la de cada segmento, pero colocándolas en ángulos encontrados a sesenta grados como se muestra en la Figura 1.

Figura 1. Prefractales para la generación de la Curva de Koch.

Podemos observar que bajo cualquier escala la curva está formada por $N = 4$ copias idénticas de si misma, escaladas en un factor de $\varepsilon = 1/3$ por lo que la dimensión de similaridad para la curva de Koch es:

$$D_s = \frac{\log(N)}{\log(1/\varepsilon)} = \frac{\log(4)}{\log(3)} = 1.261859...$$

lo que nos indica que la curva de Koch tiene una dimensión de similaridad $1 < D_s < 2$, es decir, cuando se itera al infinito, la curva de Koch ocupa un espacio mayor que la que ocuparía cualquier otro objeto unidimensional, pero menor que el que ocuparía cualquier otro objeto bidimensional.

III.4 Dimensión de conteo de cajas (*box counting*).

La limitación del uso de la dimensión de similaridad a fractales exactos bajo escala puede superarse definiendo la dimensión fractal de otra manera, pero conservando la idea básica del concepto una de estas definiciones es la siguiente:

Sea $F \subseteq \Re^n$, para cada $\varepsilon > 0$ sea $N_\varepsilon(F)$ el menor número de bolas cerradas de radio ε que se necesitan cubrir a F, si

$$D = \lim_{\varepsilon \to 0} \frac{\log(N_\varepsilon(F))}{\log(1/\varepsilon)} \tag{8}$$

existe, entonces a D se le llama la *dimensión fractal* de F [22].

Consideremos ahora que las mediciones de dimensión fractal se realizarán haciendo uso de una computadora, la cual es en esencia una máquina discreta, entonces podemos redefinir a $N_\varepsilon(F)$ como el número de cajas n-dimensionales cuyo lado tiene una longitud $\varepsilon > 0$. Estas cajas son disjuntas y forma una cubierta para $F \subseteq \Re^n$. Entonces D representa a la *dimensión de conteo de cajas (box countig)* del conjunto F [23].

En la práctica, resulta difícil calcular el valor en la ecuación (8) para valores pequeños de debido a las limitaciones de la representación de punto flotante, por lo que para poder estimar el valor límite en (8) lo que suele hacerse es realizar una gráfica de los datos experimentales de $\log(1/\varepsilon)$ contra $\log(N_\varepsilon(F))$ y elaborar un ajuste de regresión de mínimos cuadrados. La pendiente obtenida en la regresión se puede considerar una aproximación empírica para D.

III.5 Dimensión de masa-radio.

Cuando el objeto a examinar presenta una geometría radial es posible utilizar la dimensión masa-radio [24] para evaluar una relación exponencial del objeto con respecto al espacio que ocupa. La dimensión masa-radio está basada en la propiedad (5) y se determina mediante la relación

$$N(R) \approx R^{D_m} \tag{9}$$

Donde $N(R)$ es la masa o cantidad de unidades elementales que contiene el objeto que caen dentro de un radio R a partir de un origen preestablecido. De manera similar al caso de la dimensión de conteo de cajas, el valor D puede obtenerse a partir una la regresión de los datos graficando $N(r)$ contra R. Es importante notar que aunque un objeto fractal posea una geometría radial, el objeto no necesariamente tiene que ser circular. Por tanto, al graficar los datos en escala logarítmica puede no encontrarse una relación enteramente lineal. Para ayudar a reducir este efecto, en vez de tomarse el radio con respecto a un origen, se considera el llamado *radio de giro* R_g, que corresponde a la distancia medida desde el centro de masa de la figura en cuestión y está dado por

$$R_g = \sqrt{\frac{1}{n}\sum_{i=1}^{n} |\vec{x}_i - \vec{x}_0|^2} \qquad (10)$$

donde n es el número correspondiente a la masa total del objeto, $\vec{x}_i$ es la posición de la *i*-ésima unidad de masa que conforma al objeto, y $\vec{x}_0$ es una posición fija, por ejemplo, el origen.

III.6 Dimensión de correlación densidad - densidad.

Esta es otra medida útil para calcular la dimensión fractal en el caso de que el objeto no presente autosimilaridad exacta, sino que ésta solamente existe en un sentido estadístico, por ejemplo en los fractales aleatorios. Para calcular la función de correlación densidad - densidad [4] se usa

$$c(\mathbf{r}) = \frac{1}{V}\sum_{\mathbf{r}'} \rho(\mathbf{r}+\mathbf{r}')\rho(\mathbf{r}') \qquad (11)$$

que es el valor esperado del evento de que dos puntos separados por una distancia $|\mathbf{r}|$ pertenezcan al objeto. En la ecuación anterior, ρ representa la densidad local, es decir, $\rho(\mathbf{r}) = 1$ si el punto $\mathbf{r}$ pertenece al objeto, de otra forma $\rho(\mathbf{r}) = 0$. Para un fractal aleatorio discreto, el valor V puede interpretarse como el número de unidades que en el objeto.

Ahora bien, un objeto es invariante bajo escala de manera no trivial (autosimilar) si su función de correlación no cambia hasta cierta constante de escalamiento cuando se escala por un valor arbitrario b, es decir cuando se cumple

$$c(br) \approx b^{-\alpha} c(r) \qquad (12)$$

con α como un valor entero mayor que cero y menor que la dimensión euclidiana del objeto d.
Puede probarse [24] que la única función que satisface la relación anterior es la ley exponencial:

$$c(r) \approx r^{-\alpha} \qquad (13)$$

A partir de esta última expresión se define la dimensión de correlación densidad-densidad como

$$D_c = d - \alpha \qquad (14)$$

III.7 Lagunaridad.

El concepto de lagunaridad fue introducido por B. Mandelbrot [1] para caracterizar objetos fractales con la misma dimensionalidad fractal pero con características geométricas completamente diferentes. El parámetro de lagunaridad cuantifica básicamente la distribución del objeto en el espacio, midiendo los intervalos de espacios vacíos (lagos) a diferentes escalas. Formalmente la lagunaridad de un objeto se define como el grado de invarianza traslacional de un objeto [25]. Esto es, si al trasladarnos a lo largo de un objeto encontramos muy poca variación entre los patrones locales determinados por la estructura del objeto, entonces el objeto tiene una lagunaridad baja. Si por el contrario, al trasladarnos sobre el objeto los patrones locales encontrados resultan significativamente diferentes, el objeto presenta entonces una lagunaridad alta.

Para ilustrar el concepto de lagunaridad se muestra el ejemplo del conjunto clásico de Cantor. Consideremos los diagramas mostrados en la Figura 2, que corresponden a prefractales distintos de formación del conjunto de Cantor.

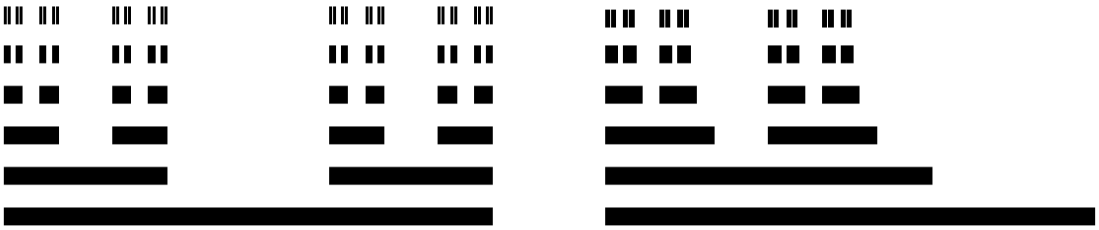

Figura 2. Conjuntos de Cantor con diferente lagunaridad.

Ambos conjuntos en la figura anterior son creados mediante sucesivas iteraciones del siguiente procedimiento: Partir de un segmento de línea recta y dividirlo en tres partes iguales, remover una de estas partes y aplicar el mismo procedimiento a las dos partes restantes. La diferencia se encuentra en que en el primer conjunto se remueve siempre la parte media y en el segundo conjunto se remueve siempre la parte de la derecha. Para ambas construcciones se tiene que existen $N = 2$ copias idénticas del conjunto total escaladas por un factor de $\varepsilon = 1/3$ por lo que la dimensión de similaridad es para ambos conjuntos la siguiente:

$$D_s = \frac{\log(N)}{\log(1/\varepsilon)} = \frac{\log(2)}{\log(3)} = 0.630929...$$

Esto nos hace ver que aunque los conjuntos en la Figura 2 presentan la misma dimensionalidad fractal, su estructura geométrica es diferente, por lo que se concluye que la dimensión fractal no es suficiente para describir la geometría de objetos autosimilares. El concepto de lagunaridad es complementario al de dimensión fractal y nos ayuda a caracterizar los ejemplos anteriores, en los cuales la variedad y distribución de los espacios vacíos es diferente.

III.8 Multifractalidad.

El concepto de multifractalidad extiende el análisis fractal de los objetos fractales o no fractales hacia las medidas que pudieran definirse sobre ellos [8]. El análisis de multifractalidad nos permite obtener el conjunto de exponentes de escalamiento que intervienen en un fenómeno autosimilar complejo. Existen varias técnicas para determinar la multifractalidad de un objeto en este proyectos se utilizan El método de momentos descrito por Halsey, *et al*. [17] está basado en una función de partición, el método de caja de arena [18] que se basa en escalamiento exponencial de la distribución de una medida, y el método de caja extendida [19] que es similar al de Halsey, pero toma una medida extendida para evitar efectos de borde.

De aquí obtenemos la extensión de la idea de autosimilaridad de conjuntos hacia las medidas que pueden definirse sobre éstos. Cuando se presenta una irregularidad sobre una medida definida un objeto, y esta irregularidad es la misma al menos en un sentido estadístico, bajo diferentes escalas, entonces se dice que la medida es multifractal [8]. El fenómeno de multifractalidad suele aparecer en intrincadas estructuras que presentan más de un exponente de escalamiento, muchas de las cuales son caracterizadas por un espectro infinito de exponentes. Las medidas multifractales, sin embargo, también pueden estar soportadas por conjuntos no fractales.

En un espacio euclidiano de dimensión E, la densidad de un conjunto S está definida por $\mu(S)/\varepsilon^E$. Cuando esta densidad varía lentamente, puede ser mapeada en forma de relieve o en forma de líneas de altura constante. Conforme $\varepsilon \to 0$ se espera este relieve tienda a un límite. Para una medida uniforme (medida de Lebesgue) la densidad es 1 en cada parte de S. Para el caso de una medida autosimilar, la medida en una ε-vecindad de un punto x_0 escala en forma exponencial:

$$\mu(x_0, \varepsilon) \sim \varepsilon^{\alpha} \tag{15}$$

para algún α, la densidad μ/ε entonces escalará como $\varepsilon^{\alpha-1}$. Si $\alpha \neq 1$, el límite de la densidad cuando $\varepsilon \to 0$ degenerará hacia 0, o bien hacia ∞. Cuando la medida en una ε-vecindad de un punto escala con una ley exponencial en el límite cuando $\varepsilon \to 0$, al exponente α se le llama *exponente local de Hölder (Hölder exponent)*. Otros términos usados son *índice de singularidad (singularity index)* o *fuerza de singularidad (singularity strenght)*. De lo anterior tenemos que, dado un punto $x \in S$, el exponente local de Hölder se define como:

$$\alpha(x) = \lim_{\varepsilon \to 0} \frac{\log \mu(B_x(\varepsilon))}{\log \varepsilon} \tag{16}$$

donde $B_x(\varepsilon)$ es una bola de tamaño $B_x(\varepsilon)$ con centro en x. Cuando el límite no existe, entonces el exponente de Hölder no está definido.

En la mayoría de las aplicaciones prácticas, es difícil calcular el exponente local de Hölder debido a los errores de precisión cuando $\varepsilon \to 0$. Por lo que se tiene que trabajar con el concepto de *exponente de Hölder no refinado* (*coarse-grained Hölder exponent*). Para cualquier caja $B(\varepsilon)$ de tamaño lineal ε el exponente de Hölder no refinado se define como:

$$\alpha = \frac{\log \mu(B(\varepsilon))}{\log \varepsilon} \tag{17}$$

Para cada valor α, se evalúa el número de cajas $N_\varepsilon(\alpha)$ de tamaño lineal ε que tienen un exponente de Hölder no refinado igual a α. Supóngase ahora que una caja del lado ε ha sido seleccionada al azar de entre un número de cajas cuyo total es proporcional a ε^{-E}. La probabilidad de obtener el valor α para el exponente de Hölder no refinado es $p_\varepsilon(\alpha) = N_\varepsilon(\alpha)/\varepsilon^{-E}$.

Para obtener una estimación del comportamiento de la distribución de frecuencias de α, sabiendo que la distribución no tiende a un límite cuando $\varepsilon \to 0$ podemos considerar alguna de las siguientes funciones:

$$f_\varepsilon(\alpha) = \frac{\log N_\varepsilon(\alpha)}{\log \varepsilon} \tag{18}$$

o bien

$$C_\varepsilon(\alpha) = \frac{-\log p_\varepsilon(\alpha)}{\log \varepsilon} \tag{19}$$

Estas dos funciones tienden a límites bien definidos $f(\alpha)$ y $C(\alpha)$ respectivamente cuando $\varepsilon \to 0$. Cuando $f(\alpha)$ existe se tiene que

$$C(\alpha) = f(\alpha) - E \tag{20}$$

La definición de $f(\alpha)$ nos indica que para cada α, el número de cajas con exponente α incrementa conforme ε se hace más pequeño de acuerdo con

$$N_\varepsilon(\alpha) \sim \varepsilon^{-f(\alpha)} \tag{21}$$

El exponente $f(\alpha)$ es una función continua de α. En los casos más simples, la gráfica de $f(\alpha)$ tiene una forma similar al símbolo I, usualmente inclinada hacia algún lado.

Para calcular numéricamente los valores de la función $f(\alpha)$ a partir de datos experimentales, se puede emplear *el método de histograma* o bien el *método de momentos*.

III.9 El método de histograma para multifractalidad.

Dada una medida μ, el método de histograma involucra los siguientes pasos:

1. Obtener una cubierta para el conjunto S que soporta a la medida con cajas de tamaño ε. Esto resulta en una colección de cajas $\{B_i(\varepsilon)\}_{i=1}^{N(\varepsilon)}$ donde $N(\varepsilon)$ es el número total de cajas que se necesita para cubrir S.

2. Calcular el exponente de Hölder no refinado para cada caja i como $\alpha_i = \mu(B_i)/\log\varepsilon$.

3. Construir un histograma dividiendo la variable α en intervalos de tamaño apropiado $\Delta\alpha$ y estimar la densidad $N_\varepsilon(\alpha)$ contando el número de veces $N_\varepsilon(\alpha)\Delta\alpha$ que un valor específico para el exponente de Hölder cae entre α y $\alpha+\Delta\alpha$.

4. Repetir el paso 3. para diferentes valores de ε.

5. Puesto que se espera $N_\varepsilon(\alpha) \sim \varepsilon^{-f(\alpha)}$, graficar $-\log N_\varepsilon(\alpha)/\log\varepsilon$ para diferentes valores de α.

III.10 El método de momentos para multifractalidad.

El método de momentos descrito por Halsey, *et al*. [17] está basado en una función de partición definida como

$$\chi_q(\varepsilon) = \sum_{i=1}^{N(\varepsilon)} {\mu_i}^q \tag{22}$$

considerando el comportamiento de las medidas autosimilares descrito en la ecuación (15) podemos escribir $\mu_i = \varepsilon^{\alpha_i}$ lo que da como resultado

$$\chi_q(\varepsilon) = \sum_{i=1}^{N(\varepsilon)} (\varepsilon^{\alpha_i})^q \tag{23}$$

Tomando el caso continuo para obtener una expresión de la función de partición, se denota por $N_\varepsilon(\alpha)d\alpha$ al número de cajas, de un total de $N(\varepsilon)$ para las cuales el exponente α_i satisface $\alpha < \alpha_i < \alpha + d\alpha$. Luego, la contribución del subconjunto de cajas con α_i entre α y $\alpha+d\alpha$ a la función de partición es $(\varepsilon^\alpha)^q N_\varepsilon(\alpha)d\alpha$. Si en vez de sumar la contribución de cada caja i por separado se integra sobre $d\alpha$ se obtiene

$$\chi_q(\varepsilon) = \int (\varepsilon^\alpha)^q N_\varepsilon(\alpha)d\alpha \tag{24}$$

usando la ecuación (21) en la ecuación (23) obtenemos

$$\chi_q(\varepsilon) = \int \varepsilon^{q\alpha - f(\alpha)} d\alpha \tag{25}$$

En el límite $\varepsilon \to 0$ la contribución dominante de la integral en la ecuación (25) proviene de los valores de α cercanos al valor que minimiza el exponente $q\alpha - f(\alpha)$. Si $f(\alpha)$ es diferenciable, entonces una condición necesaria para la existencia de un punto extremo es

$$\frac{\partial}{\partial \alpha}(q\alpha - f(\alpha)) = 0 \tag{26}$$

Para un valor dado de q, el valor extremo ocurre para el valor $\alpha = \alpha(q)$ que satisface

$$\frac{\partial}{\partial \alpha} f(\alpha)\Big|_{\alpha=\alpha(q)} = q \tag{27}$$

y el punto extremo es un mínimo cuando (nótese que $f(\alpha)$ tiene signo negativo en (26)):

$$\frac{\partial^2}{\partial^2 \alpha} f(\alpha)\Big|_{\alpha=\alpha(q)} < 0 \tag{28}$$

Conservando solamente la parte dominante de la contribución en la ecuación (25) y haciendo

$$\tau(q) = q\alpha(q) - f(\alpha(q)) \tag{29}$$

encontramos que

$$\chi_q(\varepsilon) \sim \varepsilon^{\tau(q)} \tag{30}$$

Derivando la ecuación (29) obtenemos

$$\frac{\partial}{\partial q}\tau(q) = q\frac{\partial}{\partial q}\alpha(q) + \alpha(q) - \frac{\partial}{\partial q} f(\alpha(q))\frac{\partial}{\partial q}\alpha(q) \tag{31}$$

haciendo uso de la ecuación (31) obtenemos

$$\frac{\partial}{\partial q}\tau(q) = q\frac{\partial}{\partial q}\alpha(q) + \alpha(q) - q\frac{\partial}{\partial q}\alpha(q) = \alpha(q) \tag{32}$$

lo que nos muestra que $f(\alpha)$ puede calcularse a partir de $\tau(q)$ y viceversa mediante la ecuación (29) y usando (31) para calcular $\alpha(q)$. A ecuación (29) se le conoce como transformada de Legendre.

Las dimensiones generalizadas D_q se definen para cada $q \in \Re$ como

$$D_q = \frac{\tau(q)}{q-1} \tag{33}$$

Lo anterior nos da el siguiente método para estimar $f(\alpha)$ a través de la función de partición:

1. Obtener una cubierta para el conjunto S que soporta a la medida con cajas de tamaño ε. Esto resulta en una colección de cajas $\{B_i(\varepsilon)\}_{i=1}^{N(\varepsilon)}$ con medidas $\mu_i = \mu(B_i(\varepsilon))$

2. Calcular la función de partición haciendo uso de la ecuación (26) para varios valores de ε.

3. Verificar si las gráficas de $\log \chi_q(\varepsilon)$ contra $\log \varepsilon$ son lineales. Si lo son entonces $\tau(q)$ es la pendiente de la línea correspondiente al exponente q.

4. Construir $f(\alpha)$ a partir de la transformada de Legendre de $\tau(q)$.

III.11 Entropía configuracional.

Cuando tratamos de analizar objetos geométricos generados mediante mecanismos complejos, como es el caso de los agregados fractales, un concepto que resulta útil es el de entropía configuracional. Este concepto está basado en términos de la entropía de Shannon [16], en la cual está fundamentada la teoría de la información. A grandes rasgos podemos considerar a la entropía configuracional como una medida de que tan uniformemente distribuido se encuentra un objeto en el espacio, de aquí que podamos considerarla como una métrica alternativa a la lagunaridad. Sin embargo, es importante hacer notar, que a diferencia de la lagunaridad, la entropía en sí misma no está definida como una medida invariante bajo diferentes escalas. Es por esta razón que, para los propósitos de esta tesis, introducimos una técnica para obtener tal invariancia mediante normalización. Los detalles se describen en la sección correspondiente al análisis de la entropía en los modelos.

De acuerdo con Shannon [16], La entropía se define para un conjunto $A = \{a_i : i = 1,..n\}$ de n secuencias de bits y esta expresada por

$$H(A) = -\sum_{i=1}^{n} p_i log_2(p_i) \tag{34}$$

donde p_i es una probabilidad asociada cada elemento a_i.

En nuestro desarrollo consideramos al espacio que contiene a un agregado como una secuencia de bits, en la cual el valor de cada bit se establece como $'1'$ si la posición correspondiente se encuentra ocupada, y $'0'$ para una posición vacía. En este esquema, un agregado bidimensonal contenido en un área de tamaño lineal l quedara representado por una secuencia de bits de longitud $l \times l$. Además tomamos a p_i como la probabilidad de que la evolución del modelo de crecimiento, una vez establecidos sus parámetros iniciales, dé como resultado la configuración a_i.

IV. **PLANTEAMIENTO DEL PROBLEMA E IMPORTANCIA.**

Dado que las propiedades descriptivas de los objetos fractales están definidas en límites infinitos, cuando se requiere simularlos por computadora es necesario realizar experimentos extraordinariamente grandes para poder determinar su comportamiento en límites asintóticos. Por ejemplo, a la fecha no se conoce una solución analítica para la descripción completa del fenómeno DLA, los avances hacia un mejor entendimiento del mismo pueden obtenerse mediante simulación computacional. Una forma de simular este fenómeno es mediante la ejecución del algoritmo de Witten y Sanders [2] mediante simulación Montecarlo, la cual requiere de una cantidad considerable de recursos de cómputo. Hay que considerar que inclusive aproximaciones con modelos laplacianos, de percolación y mapeos conformes es necesario realizar una gran cantidad de cálculos y espacio en memoria para la obtención de información aplicable a la descripción y predicción de los fenómenos a los cuales se aplica. Al realizar simulaciones a mayor escala, se obtienen datos más precisos acerca del fenómeno de difusión limitada, pero se incrementa la carga computacional. Por otra parte, los modelos basados en la ecuación de Laplace requieren de la solución de un sistema de ecuaciones simultáneas para cada uno de los puntos del dominio a una resolución dada para cada iteración de crecimiento, lo que implica un alto costo computacional. Con los modelos de percolación se tiene una situación similar.

El trabajar con modelos a gran escala implica el uso de una gran cantidad de recursos de cómputo no solamente en la generación de los agregados, sino también en su análisis. Por ejemplo los algoritmos para cálculos de dimensionalidad fractal tienen una complejidad $O(n)$, considerando a n como una medida del tamaño del agregado, digamos el número de partículas que contiene. No obstante, los cálculos de multifractalidad y lagunaridad, en términos de los procedimientos tratados aquí tienen una complejidad del orden $O(n^E)$, donde E representa a la dimensión encluideana sobre la cual se encuentra embebido el objeto fractal. Cabe hacer notar que la constante de proporcionalidad para el algoritmo de lagunaridad es mucho mayor que para los algoritmos de multifractalidad, por lo que en la práctica estos últimos se ejecutan a una velocidad relativamente mayor.

Otro aspecto a considerar es que el tipo de modelos tratados en esta investigación contienen elementos estocásticos, de tal forma que no se conocen de manera determinista, la forma de generarlos y por tanto, tampoco existe un procedimiento determinista para analizarlos. Como ejemplo, observemos los valores de probabilidad de aparición de cierta distribución de un agregado generado a partir de un modelo (no determinista) bajo ciertos parámetros. Tal probabilidad correspondería a los valores p_i (34). Estas cantidades deben, por tanto ser determinadas de manera empírica, en nuestro caso concreto, a través de simulación computacional. Un aspecto a considerar bajo este enfoque es la dificultad practica para calcular tal valor dada la gran cantidad de configuraciones posibles para un espacio de tamaño lineal l, puesto que el número total de configuraciones posibles es de $2^{l\times l}$, cada probabilidad p_i es muy baja, por lo que aun con ayuda de computo de alto rendimiento, no es posible medir directamente este valor a partir de un numero de ocurrencias de a_i, pues a una escala práctica, es difícil que a_i ocurra más de una vez. Por ejemplo, bajo un enfoque sencillo para la determinación de p_i, se procedería a generar una gran cantidad de agregados, y se registrarían las frecuencias en que aparece cada configuración. Desafortunadamente, en la práctica, aun contando con decenas de miles de repeticiones del experimento que genera a los agregados, es poco probable que una configuración se presente en más de una ocasión, lo que llevaría a una determinación errónea y no representativa de los valores p_i.

Por lo anterior, es este proyecto se considera de gran importancia el desarrollo de algoritmos que mejoren la eficiencia de las simulaciones, que permitan analizar los experimentos a gran escala y que sean aplicables a los fenómenos a los cuales representa el modelo. Asimismo es importante explotar en forma adecuada los recursos de hardware para cómputo intensivo de acuerdo con las características de los algoritmos que se requiera implementar sobre los mismos.

IV.1 Objetivos.

Este proyecto plantea los siguientes objetivos.

- Desarrollo de algoritmos (*variable presicion distance search* [30, 38], particionamiento de dominios quadtree en *kanbalam*) para la generación de agregados fractales a gran escala considerando y modificando las técnicas actualmente empleadas.

- Desarrollo de algoritmos (*entropy fuzzy c-means* [51], *enhanced gliding box* [64]) que permitan el análisis de agregados a gran escala contemplando los aspectos de fractalidad, multifractalidad, lagunaridad y entropía configuracional.

- Ejecutar los algoritmos en plataformas de alto rendimiento, mediante el uso de técnicas de procesamiento en paralelo con tecnologías híbridas. Analizar la interacción de agregados entre ellos mismos y el espacio en el que se definen.

- Considerar aplicaciones a las cuales se adaptan los modelos de interacción de acuerdo a la información obtenida en las simulaciones.

V. **SIMULACION DE MODELOS DE CRECIMIENTO FRACTAL.**

Para la simulación de diversos modelos computacionales de crecimiento fractal se utilizaron técnicas de programación en paralelo basadas en arquitectura de memoria compartida y memoria distribuida. Las técnicas utilizadas para obtener paralelismo son:

- *Pthreads* (*POSIX threads*) [26].
- OpenMP (*Open Multi-Processing*) [27].
- MPI (Mes*sage Passing Interface*) [28].
- CUDA (*Compute Unified Device Architecture*) [29].

Es importante mencionar que cada una de estas técnicas es aplicada de acuerdo a la arquitectura y tipo de algoritmo que se desea paralelizar. Y en este proyecto las utilizamos como se describe a continuación.

Pthreads y OpenMP se usan en este proyecto para paralelizar algoritmos en arquitectura multiprocesador *multicore* de memoria compartida. La utilizamos para paralelizar el algoritmo de búsqueda en un *quad-tree* [30] generación de caminatas aleatorias de partículas y paralelización de los algoritmos de detección de multifractalidad como son conteo de cajas -*box counting*- [1] cajas de arena -*sand box*- [31] cajas extendidas -*extended box*- [19] así como en algoritmos recursivo de cajas deslizantes -*sliding box*- [13]. Esta tecnología nos permite escalar el procesamiento paralelo, con el equipo que se cuenta hasta 8 núcleos de procesamiento o *cores*. Este escalamiento es, en términos del estado del arte del cómputo en paralelo actual relativamente bajo, pero se tiene la ventaja de que al estar en memoria compartida la comunicación y sincronización entre los hilos de ejecución es bastante rápida.

Otra técnica que utilizamos es la de paso de mensajes con MPI y la utilizamos para distribuir los procesos en varios nodos de cómputo conectados a través de una red. Esto nos permite distribuir un objeto fractal grande en varios nodos de cómputo de forma que se puede aprovechar la memoria de todos los nodos en conjunto para contener al fractal como se estuviera procesándose dentro de una sola computadora. Esta tecnología, con las propiedades de la cuenta de acceso que se tiene en la computadora *HP Cluster 4000 Kanbalam* nos permite escalar el paralelismo hasta 64 *cores*, lo cual es un escalamiento moderado, con la desventaja de que la comunicación entre procesos es lenta, debido a que la información tiene que pasar por una red entre los nodos que conforman el *cluster*. Las principales ventajas que se tiene con esta tecnología es que al distribuir los procesos entre nodos también se expande la memoria disponible –aunque dividida en dominios- y de que se cuenta con una gran cantidad de almacenamiento. Esto nos permitió generar agregados fractales a una escala muy grande.

Como otra opción de paralelismo contamos con la tecnología reciente de las arquitectura masivamente paralela proporcionada por tarjetas gráficas con capacidades de cálculo de propósito general (GPGPU), que constan con varios cientos de núcleos de procesamiento (GPU's) y nos proporciona otra alternativa de programación paralela, la cual estamos empleando para explorar variantes de crecimiento laplaciano y mapeos conformes para generar agregados fractales. De momento, para el proyecto se cuenta con una tarjeta *Nvidia GTX 460* con 336 *cores* lo que nos da un escalamiento mayor inclusive al que se nos proporciona los recursos asignados en Kanbalam.

La principal desventaja que presenta para nuestro proyecto esta tecnología, es que debido a la arquitectura de las GPU's, la cual difiere de los tradicionales CPU's de una computadora convencional, no se cuenta con un mecanismo para implementar algoritmos recursivos. Además, la elaboración de un mecanismo propio para conseguir recursividad resultaría ineficiente, debido a que la arquitectura de las GPU's solamente asigna unos cuantos *KB* de memoria por *core* para el manejo de la pila de llamadas [29], la cual se vería desbordada rápidamente por un procedimiento recursivo.

La memoria disponible para cálculos también representa una restricción en estas tarjetas, pues está limitada al tamaño de la memoria de video disponible. Sin embargo, contar con esta cantidad de núcleos de procesamiento resulta bastante atractivo para nuestro proyecto, por lo que hemos estado usando esta tecnología con éxito para explorar variantes del crecimiento laplaciano.

A continuación se describen los detalles de implementación que usamos para simular diversos modelos de crecimiento fractal.

V.1 Visualización.

Como parte de este proyecto se ha desarrollado código para convertir los datos de agregados generados en diversos formatos gráficos, como son bmp (*windows bitmaps*) y jpeg *(joint photographic experts group*) para representaciones bidimensionales utilizando las librerías OpenCV (*Open Computer Vision*), también se creó código para convertirlos a formato DXF para representaciones tridimensionales. Este último formato puede ser visualizado con aplicaciones de licencia libre como DWGViewer, o bien comerciales como AutoCad, o 3DStudio. Asimismo se emplearon en el presente proyecto los kits de desarrollo de CUDA [29] en interacción directamente con las librerías OpenGL(*Open Graphics Library*) para efectos de visualización.

V.2 Modelo de Eden.

El modelo de Eden [3], fue uno de los primeros modelos de agregación fractal con aplicaciones a fenómenos físicos, propuesto hace más de medio siglo, es un modelo bastante sencillo, que consiste en la construcción de un agregado de partículas sobre una rejilla, el agregado está compuesto de sitios colindantes ocupados en la rejilla. El crecimiento del agregado ocurre cuando a cada sitio vacío contiguo a un sitio ocupado, se le asigna una probabilidad de crecimiento p, independiente de los demás sitios, y en cada iteración del ciclo de crecimiento, esta probabilidad es evaluada mediante el método de simulación Monte Carlo. Este modelo resulta interesante debido a que variantes del mismo introducidas por Williams y Bjerknes [32], que incluyen crecimiento y desaparición de sitios, describen de manera notable el crecimiento cancerígeno [32,33]. Otras variantes de este modelo que incluyen reducción de ruido en la simulación de las probabilidades p [34]. El modelo de Eden puede crear estructuras de superficie compleja, debido a que se trata de un modelo de crecimiento local, resulta sencillo de implementar en arquitecturas paralelas. En esta tesis no se analiza el modelo de Eden, no obstante, lo se menciona aquí como ejemplo comparativo con respecto a los demás modelos.

V.3 Modelo de Percolación.

El modelo de percolación es otro modelo clásico de crecimiento fractal [35]. Este modelo representa un ejemplo bastante intuitivo de un fenómeno físico complejo denominado cambio de fase. El modelo consiste en generar sitios de crecimiento en una rejilla. La geometría de la rejilla puede ser variada, aquí nos enfocaremos en el caso de una rejilla rectangular. Para producir un sitio de crecimiento se asigna primero a cada localidad i en la rejilla, una probabilidad p, de que la localidad ocupada. Después, siguiendo un método de simulación tipo montecarlo, se genera una secuencia de números aleatorios $\{\alpha_i : i = 1, \ldots, L^E\}$ con L como el tamaño lineal de la rejilla y E la dimensión ecuclideana y $\alpha_i \in [0,1]$ siguiendo una distribución uniforme. Si $\alpha_i < p$ entices el sitio i se marca como ocupado. En el caso del fenómeno de percolación, resulta interesante observar el comportamiento de los agregados producidos, i.e. de los conjuntos conectados consistentes de localidades adyacentes ocupadas. El número y extensión de estos puede variar de acuerdo al valor de p. Por ejemplo, es fácil ver que para valores de p cercanos a cero, en un experimento finito, no se obtendrán agregados por lo que el volumen típico de estos agregados será $V = 0$; conforme p se incremente, comenzaran a aparecer algunos agregados pequeños. Por otra parte, para valores cercanos a uno, los agregados tenderán a rellenar el espacio que los contiene, y conforme p disminuye se verá que los agregados generados ocupan un volumen $V \propto E$. Considerando un proceso continuo en este fenómeno, se deduce que existe al menos un valor p_c de p, en el que las tendencias descritas anteriormente se revierten. A p_c se le conoce como punto crítico.

En la Figura 3 muestra un agregado de percolación generado de la manera anteriormente descrita utilizando cálculos en paralelo utilizando una GPU con capacidad de cálculo de propósito general. Puesto que los valores α_i son independientes, la generación de los mismos en paralelo resulta sencilla. La identificación de agregados se realiza mediante un sencillo algoritmo de etiquetado de una sola pasada [36] El valor utilizado en este caso es $p = 0.59$. Con este valor, la probabilidad de encontrar un agregado que atraviese todo el plano (es decir, que logre percolarse completamente) es alta. Siguiendo esta observación se puede definir la probabilidad $P(p)$ de que un punto en la rejilla pertenezca al agregado con mayor número de puntos. Si se toma al fenómeno como invariante bajo diversas escalas, entonces $P(p)$ es independiente del tamaño lineal de la rejilla sobre la cual se modela el fenómeno de percolación. Experimentalmente se ha observado, y se puede probar mediante una técnica de renormalización [23] que $P(p)$ sigue un comportamiento exponencial en valores cercanos a p_c. Esto es:

$$P(p) \propto (p - p_c)^{\beta} \qquad (35)$$

Con $\beta = 5/36$. De la relación (n) se observa que un pequeño incremento de p en valores cercanos a p_c producen cambios drásticos en $P(p)$. A este tipo de efectos se les conoce como cambios de fase y guardan una relación conceptual con los fenómenos físicos conocidos bajo el mismo nombre.

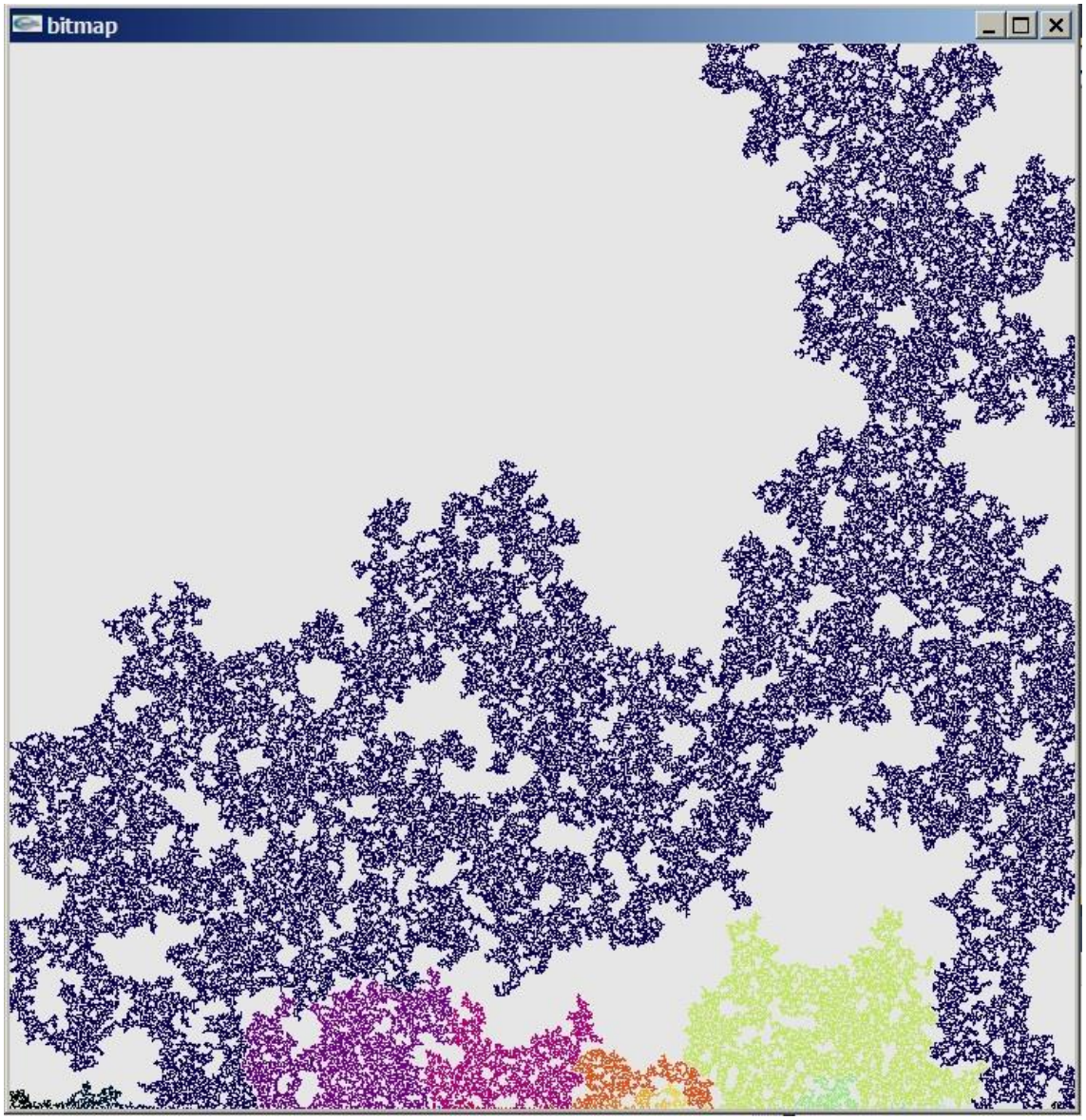

Figura 3. Agregados a partir del modelo de percolación generados en una GPU GTX 460. Procesando en paralelo, en una rejilla de 720x720, el tiempo de generación en paralelo no excede de 7 ms.

V.4 Agregación por Difusión Limitada (DLA).

El modelo de agregación por difusión limitada (DLA) introducido por Witten & Sander [2] describe un gran número de fenómenos naturales de crecimiento [23,24,35]. En el modelo de agregación por difusión limitada, solamente se permite un agente o partícula moviéndose en el espacio a la vez (de ahí el nombre 'difusión limitada'). Otra diferencia con respecto a los modelos basados en agentes, es que la probabilidad de moverse hacia cualquier dirección es siempre la misma. Este modelo puede ser descrito como el siguiente proceso iterativo: Una partícula es colocada en el origen, este es el agregado inicial. Otra partícula realiza una caminata aleatoria comenzando desde un lugar que se encuentra a una distancia R_{bh} conocida como radio de nacimiento. La caminata aleatoria termina cuando la partícula visita una localidad adyacente al agregado, entonces la partícula en movimiento pasa a formar parte del agregado. Si la

partícula excede los límites establecidos por el radio de muerte R_{dh} , entonces es descartada. El proceso es repetido hasta que se alcanza un determinado tamaño del agregado. El diagrama de flujo correspondiente se muestra en la Figura 4. En este diagrama, los pasos sombreados en gris corresponden a secciones críticas que deben ser protegidas cuando se utiliza procesamiento en paralelo para su ejecución. La comprobación en líneas punteadas es solo necesaria en la versión paralela. Esta comprueba que no ocurran colisiones entre partículas manipuladas por diferentes procesadores.

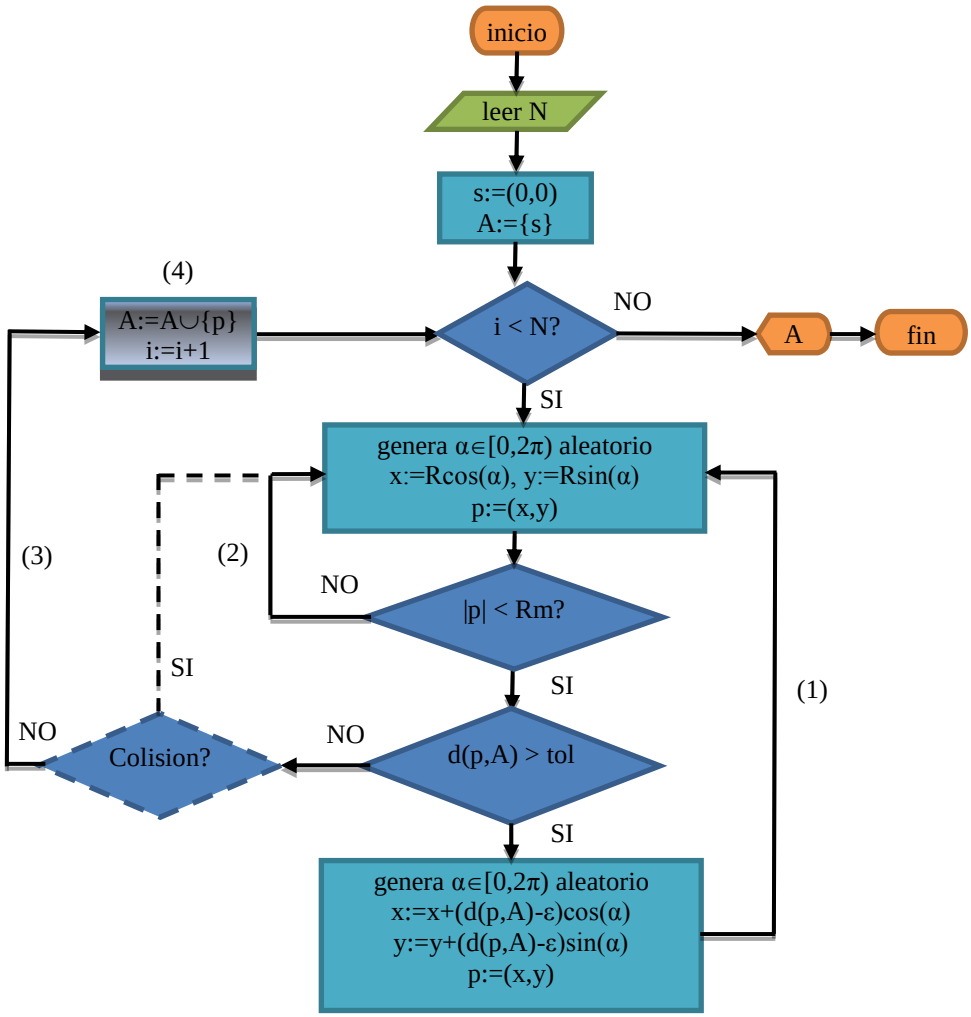

Figura 4. Diagrama de flujo para generar agregados DLA off-lattice.

Aunque el esquema algorítmico del modelo DLA es bastante simple, las estructuras generadas por este modelo son bastante complejas, y para obtener agregados de gran tamaño es necesario implementar diversas técnicas de simulación por computadora. La obtención de agregados de gran tamaño es importante para poder determinar el comportamiento asintótico de las propiedades fractales de los fenómenos que describe este modelo. La principal dificultad técnica que aparece cuando se trabaja con agregados grandes consiste en que cada partícula en movimiento debe conocer la posición (al menos relativa) de las demás partículas que conforman el agregado en cada paso de la caminata aleatoria, por lo que se requiere realizar un número masivo de consultas de distancia entre partículas.

Un elemento que desempeña un papel clave en la simulación del modelo DLA es la determinación de la posición de la partícula con respecto al resto del agregado, lo cual implica la ejecución de un algoritmo de determinación de distancia. La consulta de distancia en estructuras de datos multidimensionales son un tema muy importante para diversas áreas de la computación, como son: robótica, detección de colisiones, graficas computacionales, reconocimiento de patrones, planeación de rutas, entre otros. Actualmente existe una amplia variedad de algoritmos para realizar consultas de distancia tanto en forma aproximada y exacta [38]. En esta investigación introducimos una nueva variante de búsqueda para *quadtrees* y *octrees* que produce un valor estimado de la distancia entre un punto y su vecino más cercano dentro de un conjunto. El valor aproximado tiene un margen de error proporcional a la magnitud del mismo.

El problema específico que se resuelve con el algoritmo aquí presentado se basa en el trabajo anterior desarrollado como tesis de maestría [38], extendiendo su aplicación a hardware especializado a gran escala. El punto clave consiste en calcular de manera rápida la distancia entre un punto en movimiento y un conjunto consistente en un número de puntos distribuidos en un dominio espacial. Para mejorar el desempeño de la consulta de distancia, sacamos provecho del grado de precisión conocido como aceptable en cada consulta para el cálculo de distancia entre un punto (*query point*) y un conjunto grande de puntos mediante la observación de los nodos de la estructura (un *quadtree* u *octree*) en la cual los datos correspondientes a las coordenadas de los puntos se encuentran almacenados, el área que cubren en el espacio y la distancia relativa del punto en consulta y su entorno en términos de una relación de escala.

Previo al algoritmo aquí desarrollado, una idea de tener varias representaciones para diversos conjuntos de escalas había sido aplicada de manera satisfactoria para la generación de agregados fractales en simulaciones mediante el uso de mapas de bits [39]. La alternativa desarrollada en esta investigación es un método diferente el cual presenta la ventaja de que no hace uso de ningún tipo de mapa para almacenar información adicional acerca de una variedad de vistas en diferentes escalas. En su lugar, se aprovecha la estructura interna del *quadtree* u *octree* que almacena los datos en forma jerárquica, esto nos permite ahorrar memoria y acelerar los cálculos de manera que podemos realizar simulaciones dinámicas teniendo una enorme cantidad de elementos. Haciendo uso de esta técnica no se necesita actualizar estados de mapas de bits externos conforme la configuración del agregado cambia. A continuación damos la descripción del método para una búsqueda en un espacio con dimensión euclidiana $d = 2$, el cual se extiende directamente a dimensiones mayores, remarcando el hecho de que el número de hijos n cada nodo de un *quadtree* es de 2^d.

Aquí utilizamos la técnica de búsqueda de precisión variable en *quadtrees* aplicada a la simulación del modelo DLA, no obstante, esta técnica de búsqueda puede ser aplicada a una variedad de problemas para los cuales los requerimientos de precisión puedan ser manejados en la misma forma. La búsqueda de precisión variable puede ser ajustada a devolver un valor exacto en los niveles del *quadtree* que lo requieran, intercambiando precisión por velocidad.

La generación de agregados con el modelo DLA conformados por un gran número de partículas ya ha sido tratado anteriormente en diversas ocasiones, esto debido a la importancia de la reproducción del modelo a gran escala para el establecimiento de parámetros asintóticos de muchos fenómenos naturales que se pueden describir mediante modelos de crecimiento fractal [24, 35].

El modelo de agregación por difusión limitada descrito por Witten & Sanders [2], fue en un principio simulado para unos pocos miles de partículas. Una mejora significativa para la simulación del modelo fue dada por Ball & Brady [39] mediante la introducción de saltos aleatorios para las partículas que se mueven en áreas vacías. Esto fue conseguido utilizando una jerarquía de mapas de bits representado las áreas ocupadas en diferentes escalas. Estas técnicas permitieron a Tolman & Meakin [40] construir agregados DLA de hasta 10^7 partículas. Kaufman y otros coautores [12] introdujeron el uso de *quadtrees* para un almacenamiento eficiente de partículas, obteniendo agregados de hasta 10^8 partículas utilizando 32 procesadores trabajando en paralelo. En este trabajo comenzamos a partir de las técnicas mencionadas en las referencias anteriores introduciendo una nueva técnica de búsqueda sobre quadtrees, lo que nos permite generar agregados DLA con hasta 10^9 partículas utilizando solamente un procesador, en un tiempo de generación promedio de 14 horas. Para mejorar estas cifras, también se hizo uso de sistemas de multiprocesamiento para generar agregados siguiendo el modelo PDLA, el cual converge al modelo DLA para una gran cantidad de partículas con respecto al número de procesadores utilizados.

V.5 Técnica de determinación de distancias con precisión variable.

La técnica de búsqueda con precisión variable para *quadtrees* y *octrees* explota la información implícita acerca del espacio contenida en la estructura de datos mediante la determinación del primer nodo representado un área libre de partículas que se encuentra desde la posición a consultar, y después, dependiendo del nivel en el cual esta área fue encontrada, establece una distancia aproximada. Es decir, para determinar la distancia entre un punto determinado y un conjunto de puntos se revisan los centros de los nodos ocupados en los primeros niveles del *quadtree*. Si el punto de consulta se encuentra lo suficientemente alejado, entonces se devuelve una distancia aproximada, de otra forma, el algoritmo verificara si se trata de una hoja conteniendo puntos y entonces regresara una distancia exacta, de lo contrario, bajara al siguiente nivel. Este procedimiento es llevado a cabo recursivamente a través de todos los niveles del *quadtree*. El siguiente listado describe los detalles exactos a través de pseudo código, dando también una explicación de las funciones usadas.

Listado 1. Pseudocódigo para función de distancia con precisión variable:

```
00 precondition: min_dist=infinity
01 recursive real function VPDIST(node,min_dist)
02 /*buscar distancia exacta o aproximada */
   /*desde un punto hacia un conjunto de puntos */
03 value pointer node node
04 reference real min_dist
05 pointer node son, real d
06 if not ISNULL(node) then
07     if DIST(p,CENTER(node)) > DIAGONAL(node)*2 then
08         /*devuelve una distancia aproximada */
09         if DIST(p,CENTER(node)) -DIAGONAL(node)/2 < min_dist then
10             return DIST(p,CENTER(node))-DIAGONAL(node)/2
11         endif
12     else
13         if ISLEAF(node) then
14             /*si es una hoja, devolver distancia exacta */
15             d:=min_dist
16             for each point in node do
17                   if DIST(p,point)< d then d:=DIST(point)
18             enddo
19             if d< min_dist then return d endif
20         else
21             /*si no es una hoja, pide la dirección del punto */
22             son := DIR_POINT(p, node)
23             /*ahora buscar recursivamente en la estructura */
24             min_dist:= VPDIST(son,min_dist)
25             if INTERSECTS(CIRCLE(p,min_dist),
                                 LEFT_SIB(son)) then
26                 min_dist:=VPDIST(LEFT_SIB(son),min_dist)
27             endif
28             if INTERSECTS(CIRCLE(p,min_dist),
                                 RIGHT_SIB(son)) then
29                 min_dist:=VPDIST(RIGHT_SIB(son),min_dist)
30             endif
31             if INTERSECTS(CIRCLE(p,min_dist),
                                 OPP_SIB(son)) then
32                 min_dist:= VPDIST(OPP_SIB(son),min_dist)
33             endif
34         endif
35     endif
36 endif
37 return min_dist
```

Descripcion de funciones invocadas:

`DIR_POINT(`*point, node*`)` Regresa un apuntador al hijo de *node* que esta en el mismo cuadrante que *point*.

`CENTER(`*node*`)` Regresa el punto central del espacio cubierto por *node* en el *quadtree*

`DIAGONAL(`*node*`)` Regresa la longitud de la diagonal que va a través del área cubierta por *node* en el *quadtree*, regresa *infinity* si *node* es un apuntador nulo.

`DIST(`*point1*,*point2*`)` Regresa la distancia euclidiana entre *point1* y *point2*.

`ISLEAF(`*node*`)` Regresa *true* si *node* es una hoja del quadtree

`CIRCLE(p, r)` Regresa una referencia a un objeto *circle* con radio r y centro p. Esta función es remplazada SPHERE (*p, r*) en tres o mas dimensiones.

`INTERSECTS(circ, node)` Funcion booleana que regresa *true* si el circulo (o esfera) referenciado por *circ* intersecta a el área cudarada (o volumen cubico) correspondien te *node* en el quadtree

`LEFT_SIB(node)` y `RIGHT_SIB(node)` Estas funciones hacen referencia a los vecinos izquierdo y derecho de *node* en el mismo nivel de jerarquía dentro del *quadtree*

`OPP_SIB (node)` Regresa una referencia al vecino diagonalmente opuesto de *node* en el mismo nivel del *quadtree*.

Para examinar el potencial de nuestro algoritmo, analizamos los posibles casos y valores retornados por el procedimiento descrito arriba. Las conclusiones quedan establecidas por el siguiente teorema.

V.6 Teorema.

Supongamos que el algoritmo definido por la función `VPDIST()` dado en el listado anterior es aplicado para encontrar la distancia entre un punto de consulta p y un conjunto de puntos $A \neq \emptyset$ almacenado en un quadtree Q, supongamos que p está dentro del dominio geométrico de Q. Sea d_g la longitud de la diagonal que atraviesa el área cubierta por un nodo en el último nivel n de Q y sea $d(p,q)$ la distancia entre p y su vecino más cercano $q \in A$. Entonces el algoritmo regresa $d(p,q)$ si la distancia desde p hasta el centro del nodo en el nivel n que contiene a q es menor que $2d_g$, de otra forma, regresa $d(p,q)$ con un error de al menos $+0.2d(p,q)$ y a lo mas de $+0.4d(p,q)$.

Demostración:

Para probar esto, analizaremos los casos para los cuales cual la función `VPDIST()` regresa distancias exactas y aproximadas. Para el caso exacto, es claro que la condición de la línea 13 será cierta para una búsqueda en el nivel n y la línea 07 nos dice que una verificación exacta es realizada cuando $d(p,q) < 2d_g$. Más aun, la línea 07 también nos dice que si $n \leq 2$ el algoritmo siempre da como resultado $d(p,q)$. Las llamadas en las líneas 24 a 32 aseguran que todos los nodos potenciales a contener al vecino más cercano serán buscados en cualquier lugar de Q debido a las condiciones de intersección en las líneas 25, 28 y 30. Por otra parte, si una distancia aproximada d_{app} es regresada, entonces esto será realizado mediante la inspección de un nodo η en un cierto nivel k con $2 < k \leq n$. Sea q_c el centro del área cuadrangular cubierta por η y sea r la distancia entreq_c y alguna de las esquinas de tal área. Tenemos que $d_{app} = d(p,q_c) - r$ como resultado de la ejecución de la línea 10. Podemos ver que lo más cercano que p puede estar de q_c es $4r$ debido a la condición establecida en la línea 07, del mismo modo, lo más lejos que p puede estar de q_c es $9r$. De otra forma, la condición en la línea 07 hubiera sido cierta para el nivel precedente al nivel $k-1$, puesto que esta verifica una distancia mayor a $8r$ desde p al centro q_{cf} del padre de η, la cual es la longitud de la diagonal atravesando un nodo en el nivel $k-2$. Nótese que $d(q_{cf}, q_c) = r$ de forma que tenemos $d(p,q_c) \leq 9r$. Para poder establecer la expresión $d(p,q) \in [d_{app}, d_{app} + 2r]$ considérese que $d(p,q) \geq d_{app}$ debido a que en caso contrario, el circulo con centro en p y radio d_{app} intersectaría con un nodo no nulo en el nivel $k-1$ y las llamadas recursivas en las líneas 24 a 32 hubieran regresado con un distancia menor que d_{app}. Más aun, aunque η no es

necesariamente el nodo en el nivel k que contiene a q, contiene al menos un punto, de forma que el vecino más cercano no está más allá de $d_{app} + 2r$ de q. Entonces tenemos que para los casos aproximados, la relación $d(p,q) \in [d_{app}, d_{app} + 2r]$ se cumple. Esto es equivalente al rango de error positivo para d_{app} enunciado en el teorema.

La Figura 5 ilustra las ideas claves dadas en la demostración del teorema, en el caso en el cual un punto a consulta p_a esta lo más cercano al centro q_c del nodo que es inspeccionado para una distancia aproximada d_{app}. También se muestra el caso en que un punto de consulta se encuentra lo más alejado posible de q_c antes de que la precisión disminuya, pues de estar mas alejado, la distancia del punto q_c seria evaluada en otro nivel del quadtree.

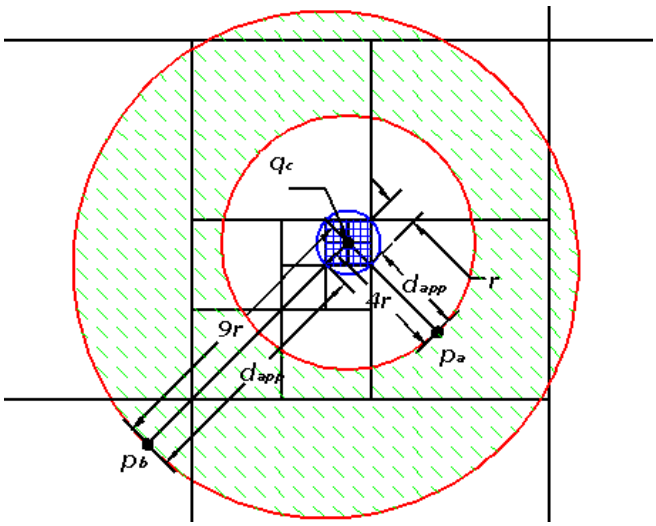

Figura 5. Esquema de la técnica de búsqueda de precisión variable sobre un quad-tree. La precisión en la distancia d_{app} hasta q_c para un punto de consulta es la misma para puntos más cercanos a q_c que p_a es mayor que para puntos más lejanos que p_b.

Hay que tener en cuenta que el algoritmo de búsqueda de precisión variable descrito aquí pudiera no ser adecuado para muchas aplicaciones de búsqueda de distancia. Sin embargo, resulta de gran utilidad en procesos en los que un error grande en los pasos intermedios no es de importancia si esto permite proseguir con los siguientes pasos con una precisión mayor. La técnica descrita anteriormente es una mejora para el uso de estructuras de datos jerárquicas del tipo *quadtree* y *octree*. Kaufman et al [12] fueron los primeros en proponerlas para su uso en la simulación del DLA, aquí mejoramos la implementación con la técnica de búsqueda de precisión variable. Adicionalmente a las simulaciones del modelo DLA, trabajamos también con el modelo PDLA el cual consiste en permitir a varias partículas realizar una caminata aleatoria en forma similar a los modelos basados en agentes descritos en una sección anterior. Nuevamente, tomamos un procesador independiente para manejar a cada partícula individual. El modelo PDLA converge hacia el modelo DLA cuando el número de partículas N en agregado alcanza un gran tamaño con respecto al número de procesadores p, es decir, cuando $N \gg p$. También empleamos una manera eficiente de almacenar los puntos en memoria y utilizamos apuntadores a grupos de partículas para reducir el espacio requerido. El hecho de que con nuestra técnica se evite el uso de mapas de bits, además de acelerar el proceso de búsqueda, permite que el procedimiento se pueda extender hacia varias dimensiones y no está limitado a dos dimensiones.

En la simulación por computadora del modelo DLA la situación en la que se requiere el algoritmo de búsqueda se presenta en la parte del movimiento browniano, debido a que la partícula se encuentra en un entorno complejo definido por todos los demás puntos que ya

forman parte del agregado. Para ejecutar el movimiento browniano es esta situación se necesita conocer la distancia correspondiente al espacio libre en la cual la partícula se puede mover libremente. Contar con un alto grado de precisión para esta tarea no resulta crítico cuando la partícula se encuentra relativamente alejada de las ramas. Además de no utilizar mapas de bits para esta tarea, tampoco utilizamos técnicas de no eliminación (*killing-free techniques*) [39,41]. En su lugar ocupamos una sencilla técnica descrita por [24] que consiste en que cuando una partícula en un punto p con $p > R_{bh}$ con R_{bh} como el radio de nacimiento de las partículas, se toma a $|p - R_{ext}|$ como la longitud del desplazamiento para el siguiente movimiento en una dirección aleatoria, con R_{ext} como el radio exterior o máximo que tiene el agregado en ese momento. En nuestras simulaciones, la distancia máxima en la que se puede continuar con la simulación conocida como radio de muerte R_{dh} está limitada por la representación de la aritmética entera con 64 bits. Utilizando representación de punto flotante encontramos que tomar radios manteniendo una relación tan grande como $\frac{R_{dh}}{R_{ext}} = 10^{20}$ no afecta de manera significa al tiempo de procesamiento, y sin embargo, ayuda a reproducir el efecto de que las partículas pueden moverse hasta el infinito en forma aceptable desde el punto de vista numérico.

Para la simulación, se aplicó primero el procedimiento para la dimensión euclidiana $d = 2$. Para construir un agregado a partir del modelo PDLA constando de 10^{10} partículas, se ejecutó la simulación en la supercomputadora *HP ClusterPlatform 4000* 'Kanbalam'. Los trabajos de ejecución (*jobs*) se corrieron utilizando grupos de 4 núcleos de los procesadores *AMD Opteron* a 2.4 GHz con un ancho de banda a la memoria de 400 MHz. Para acelerar la convergencia del modelo PDLA al modelo DLA se utilizó solamente un procesador para hacer crecer el agregado hasta 10^7 partículas bajo el régimen de difusión limitada, es decir, utilizando un solo núcleo de procesamiento, y después incorporamos los demás núcleos para alcanzar un tamaño de 10^{10} partículas en el modelo PDLA. El tiempo promedio que tomo generar un agregado de esta magnitud fue de 42 horas. El espacio de memoria requerido fue de 42GB de RAM. Almacenar estos datos fuera de la estructura *quadtree* con representación no compactada en un disco duro ocupo un espacio de almacenamiento de 150GB por agregado. Adicionalmente generamos agregados DLA constando de 10^9 partículas utilizando solamente un núcleo de procesamiento. Para estos casos el programa se corrió sobre un servidor del alto rendimiento *PowerEdge 2900* con 8 núcleos de proceso *Intel Xeon* a 2.66 GHz con 32GB de memoria compartida con un ancho de banda de 1333MHz. Como en este caso los requerimientos de memoria no fueron tan extensivos, fue posible ejecutar las tareas fue posible generar varios agregados al mismo tiempos dentro del régimen DLA en forma concurrente. En este caso el tiempo promedio de generación de un agregado fue de 10 horas.

Para trabajar con una dimensión euclidiana $d = 3$ modificamos la estructura de datos pasando de un quadtree hacia un octreee, también modificamos varias funciones para adaptarlas a la nueva estructura, pero los pasos principales del algoritmo se conservaron en la misma forma. Las simulaciones con el algoritmo de búsqueda de precisión variable se ejecutaron para generar agregados siguiendo el modelo PDLA de un tamaño de hasta 10^9 partículas en tres dimensiones, una vez más se tuvo un procesador trabajando en la etapa inicial del crecimiento, en este caso, hasta contar con 10^7, posteriormente lanzando otros 3 procesadores simulando el proceso en paralelo hasta alcanzar 10^9 partículas. Tomó en promedio 12 horas construir un PDLA tridimensional en esta forma. Para conseguir agregados en el modelo DLA (un procesador) se utilizaron 4 horas en promedio para cada modelo. En la Figura 6 podemos ver un agregado PDLA de 10^{10} partículas**.** La Figura 7 nos muestra un pequeño agregado DLA tridimensional consistiendo de 3500 partículas. Se usa este pequeño tamaño para la imagen para poder mostrar la estructura del DLA en 3D, de otra forma, la visualización de la estructura se perdería completamente al transformar a imagen a un formato 2D.

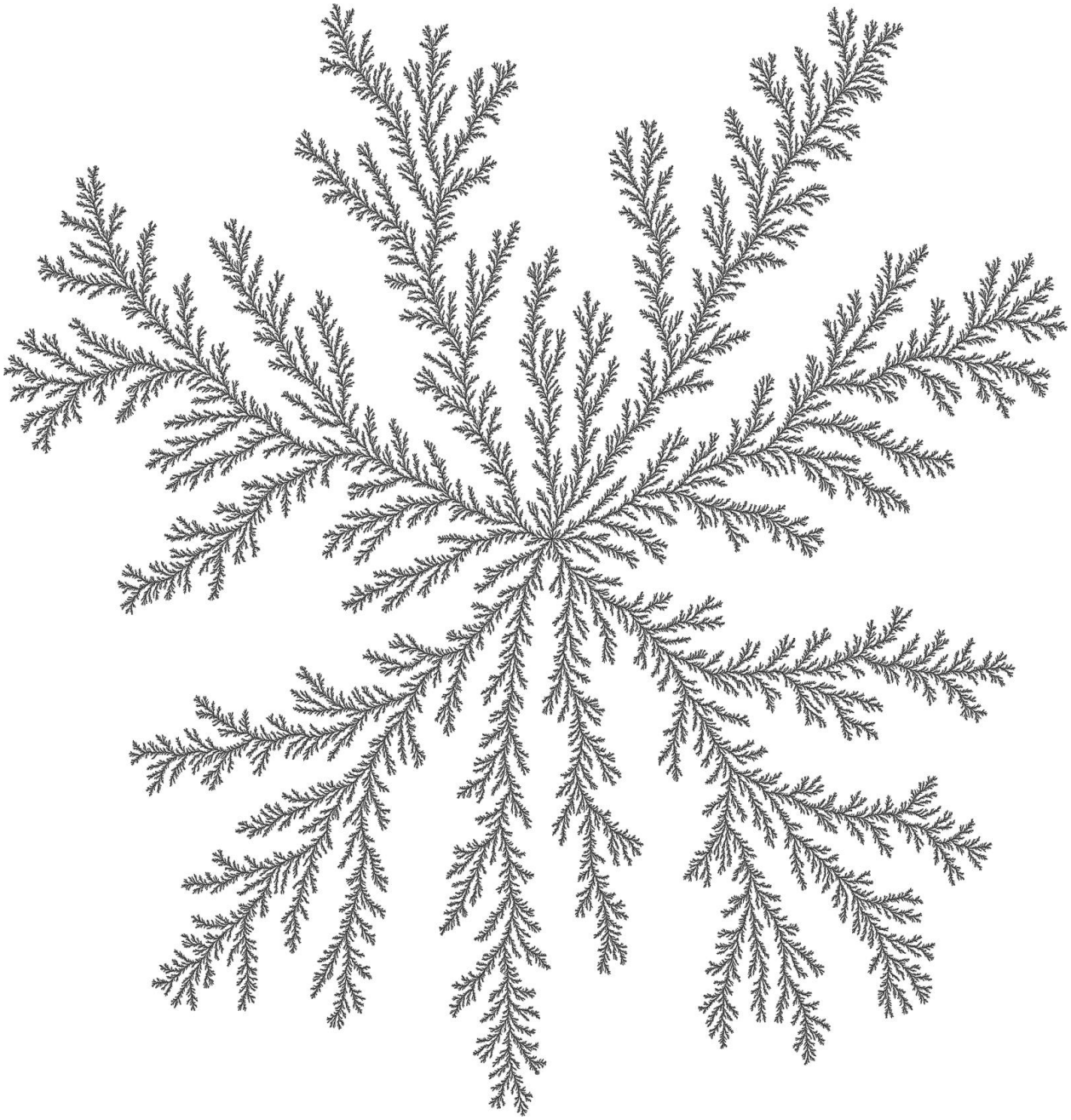

Figura 6. Agregado DLA integrado por 10^10 partículas obtenido mediante procesamiento paralelo utilizando la supercomputadora Kanbalam de la UNAM.

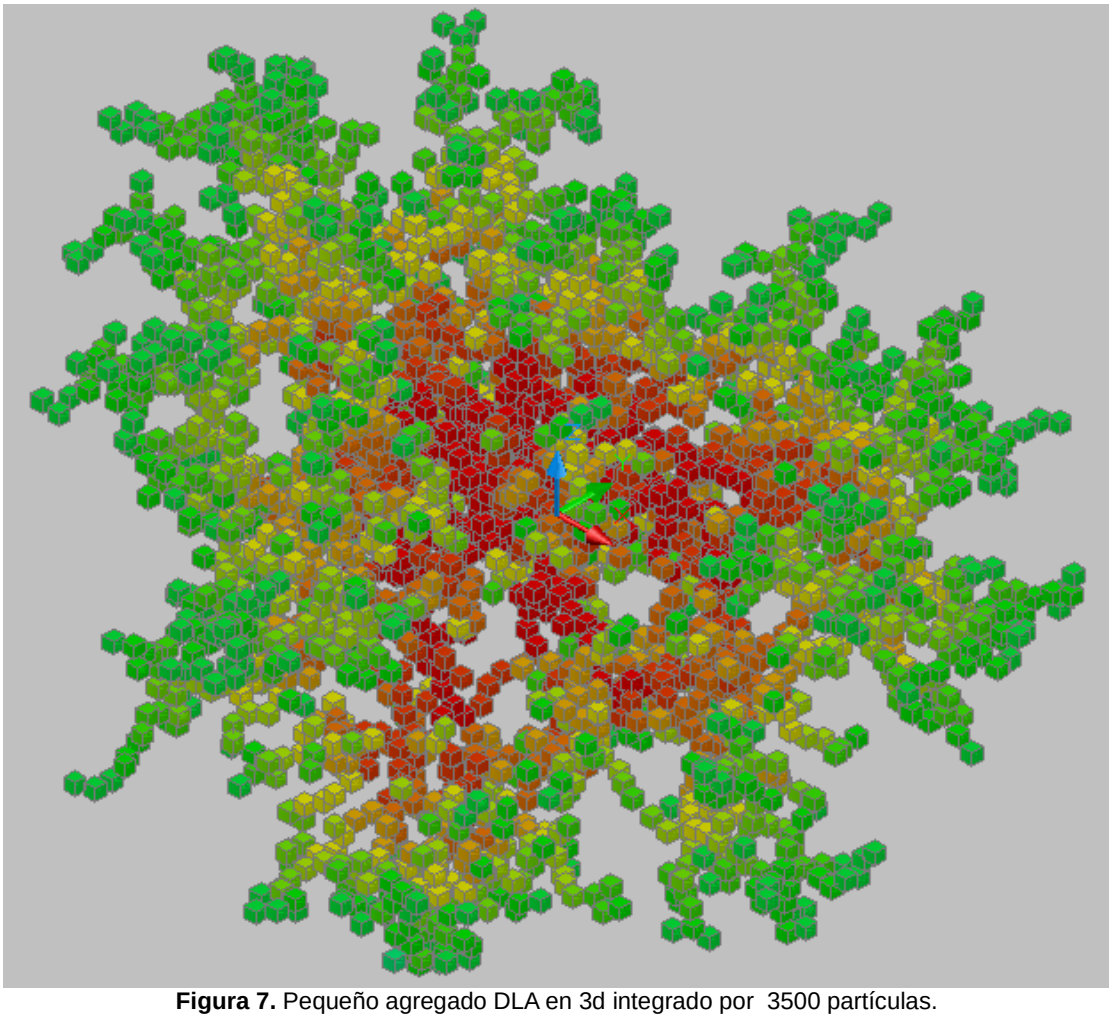

Figura 7. Pequeño agregado DLA en 3d integrado por 3500 partículas.

El código para las simulaciones DLA fue escrito en lenguaje C y compilado con la herramienta *GNU Compiler Collection* (*gcc*) [42]. Para agregar la característica de multiprocesamiento se utilizó la librería *POSIX-threads* [26] corriendo sobre el núcleo *Linux kernel* 2.6.39.3 [43] del sistema operativo Linux con las distribuciones Fedora y RedHat. Se utilizaron objetos tipo *mutex* para evitar condiciones de carrera (*race conditions*) que potencialmente pueden aparecer en el proceso generando inconsistencia de datos y traslape de partículas. La función principal en nuestro código ejecuta la simulación del movimiento Browniano para una partícula. Los procedimientos secuencial y paralelo fueron implementados dentro de una misma aplicación, dando la opción de especificar el número de hilos que estarán ejecutando la función de movimiento browniano para cada diferente partícula moviéndose en el espacio a un tiempo dado. Cuando se requiere ejecutar el modelo paralelo, una referencia a esta función es pasada para cada hilo de ejecución. Note que para conseguir paralelismo, el número de procesadores disponibles en el sistema debe ser igual o mayor al número de hilos. De otra forma, la simulación se ejecutara bajo un mecanismo de procesamiento concurrente, el cual no necesariamente acelerara el procedimiento de simulación.

El diagrama de flujo de la Figura 4 detalla el proceso ejecutado por esta función central. El Apéndice 1 muestra la implementación de esta función para ser llamada por diversos hilos. El diagrama de flujo representa un esquema de este proceso para el caso de la simulación *off-lattice* DLA. En este diagrama N representa el número de partículas. A es el conjunto de puntos que conforman el agregado. R y Rm representan el radio de nacimiento y el radio de muerte, respectivamente. La etiqueta tol denota la tolerancia en la distancia que las partículas tienen

para formar el agregado y adherirse al conjunto A, de tal forma que este valor puede ser interpretado como el diámetro de una partícula. El valor $\varepsilon \ll tol$ es un ajuste necesario para para evitar problemas relacionados con la representación numérica discreta para valores de punto flotante. $d(p, A)$ indica la distancia entre el punto p el conjunto A. Esta distancia es obtenida mediante el algoritmo de búsqueda de precisión variable `VPDIST()` descrita en el Listado 1. El código completo de esta función utilizando el lenguaje C, puede consultarse en el Apéndice 2. Refiriéndonos al diagrama de flujo, el ciclo principal de la simulación del movimiento browniano está identificado por la etiqueta (1). Este bloque ejecuta el movimiento aleatorio de la partícula actual. El ciclo identificado con la etiqueta (2) reinicia el movimiento de una partícula que ha viajado más allá del radio de muerte Rm, descartando el proceso de movimiento actual. Claramente esto representa una pérdida de eficiencia en la simulación y un ligero desvío del modelo DLA original, el cual contempla partículas provenientes de un punto infinito y desplazándose en un espacio no acotado. Desafortunadamente, lo anterior no se puede reproducir en un equipo de cómputo discreto y de capacidad acotada. Para disminuir los efector de representación finita utilizamos la técnica descrita anteriormente en sustitución de técnicas libres de muerte (*killing-free techniques*) [41]. El ciclo marcado con la etiqueta (3) realiza la inclusión de una nueva partícula al conjunto A. Si el número deseado de partículas aún no ha sido alcanzado, el proceso de movimiento comienza de nuevo. En esta etapa, se ejecuta la verificación de colisión marcada por el proceso en líneas punteadas. Este chequeo se realiza solamente en el caso de que más de un hilo se encuentre ejecutando la función de movimiento browniano. La finalidad de esto último es evitar el traslape de partículas de forma que se asegura que la nueva partícula no se coloque en el espacio en el que otro hilo planea colocar otra partícula de forma simultanea o concurrente debido a una condición de carrera o competencia entre hilos. En caso de ocurrir una colisión, el criterio que se toma es que solamente se conservara una de las partículas y las demás partículas dentro de la condición de competencia serán descartadas. El proceso sombreado correspondiente a la etiqueta (4) en el diagrama de flujo de la Figura 4, indica las secciones críticas en el proceso de actualización de la estructura de datos que están protegidas de la ejecución simultánea por distintos hilos mediante el uso de objetos *mutex*. Para evitar un excesivo tiempo de bloqueo se utilizaron tres *mutex* separados para esta sección crítica. La operación de agregar una nueva partícula en la estructura puede involucrar la creación de nuevos nodos en el *quadtree* y en todo caso se requiere reservar memoria dinámica en el sistema. Se utiliza un *mutex* para buscar y para la posible creación de un nodo en el cual una nueva partícula será agregada, y otro *mutex* para modificar o crear una lista ligada tipo bucket [38] para almacenar las partículas. Un tercer *mutex* es usado para la actualización de los datos empleados por los cálculos involucrados en la determinación del radio de giro. Los números aleatorios utilizados para obtener valores para α en cada iteración fueron generados con una versión rentrante de la función estándar `drand48_r()` [42]. Con esta utilería es posible colocar puntos aleatorios uniformemente distribuidos sobre un circulo de radio R tomando coordenadas polares $x = Rcos(\alpha)$, $y = Rsen(\alpha)$. En el caso de los agregados dentro de una dimensión euclideana $d = 3$, la obtención de puntos uniformemente distribuidos sobre una esfera se realiza generando dos variables aleatorias independientes u y θ aplicando las ecuaciones

$$x = R\sqrt{1-u^2}\cos(\theta),\ y = R\sqrt{1-u^2}\sin(\theta),\ z = Ru. \qquad (36)$$

Estas ecuaciones remplazan a las mostradas en el diagrama de flujo. Las funciones `INTERSECTS()` y `CIRCLE()` mostradas en el listado también son sustituidas para el caso 3D por funciones equivalentes. La función `VPDIST()` se codifico manejando distancias ortogonales siempre que fue posible, evitando en esta manera llamadas innecesarias a la función de la librería estándar `sqrt()`, de otra manera, el tiempo promedio de ejecución se incrementa drásticamente debido a que `sqrt()` se ejecuta por la unidad de punto flotante consumiendo una cantidad considerable de ciclos de instrucción con respecto a las operaciones aritméticas [47]. Para manejar la diferencia de distancias con implementación de distancias ortogonales hacemos

$$h = DIST\big(p, CENTER(node)\big) - DIAGONAL(node)/2 \quad (37)$$

para la implementación de las líneas 09 y 10 del Listado 1, y tomamos

$$h^2 = (|x - x_c| - \delta/2)^2 + (|y - y_c| - \delta/2)^2 \quad (38)$$

Donde (x, y) y (x_c, y_c) son respectivamente las coordenadas de la partícula en movimiento p y el centro del nodo que está siendo analizado. Aquí δ es el valor previamente calculado para la diagonal que atraviesa el área del espacio definido por el nodo y este valor es el mismo para todos los nodos que se encuentran en el mismo nivel del *quadtree*.

Los puntos en la estructura de datos son almacenados utilizando coordenadas relativas basadas en el centro de la hoja en la cual se encuentran. Las coordenadas base no son almacenadas, en lugar de esto, se calculan en forma incremental mientras se atraviesan los nodos del *quadtree*/*octree*. En los nodos correspondientes a hojas de estos árboles, se tienen apuntadores direccionando estructuras del tipo lista ligada conteniendo conjuntos de 16 partículas. Cada coordenada relativa ocupa 1 byte para su almacenamiento. En el nodo final de cada una listas aquí utilizadas aquí no existe un apuntador del tipo `next->`, Lo que se hace es manejarlas mediante apuntadores del tipo `void` y de esta manera se puede direccionar tanto nodos con o sin apuntador `next->` definidas por diferentes estructuras, pero controladas utilizando un mismo tipo de apuntador.

Puesto que utilizamos una arquitectura de 64 bits fue posible utilizar únicamente aritmética entera para toda la simulación, mientras que para los cálculos de análisis de datos se utilizó punto flotante de doble precisión. El hecho de que la simulación se realizara utilizando exclusivamente aritmética entera permitió una mejora en la velocidad de ejecución y también en la precisión de la simulación puesto que de esta manera la precisión de cada coordenada se encontraba acotada por valores uniformemente distribuidos para representar el espacio en el que se encuentran las partículas.

V.7 Evaluación del desempeño de la implementación.

Con la finalidad de evaluar el desempeño del código utilizado en este trabajo para simular el modelo DLA con la técnica de búsqueda de precisión variable, es necesario contar con una referencia de tal forma que se corrieron experimentos usando este algoritmo y los resultados se compararon contra ejecuciones utilizando el clásico algoritmo de búsqueda del vecino más cercano (*nearest neighbor search*) [11] adaptada para un *quadtree* en lugar de usar el *kd-tree* descrito en el algoritmo original. La Figura 8 muestra el desempeño promedio de ambos métodos presentado el número de partículas N en el agregado contra el tiempo de ejecución T medido en segundos para el caso de una dimensión Euclideana $d = 2$. En la figura se muestra el ajuste de los datos a las curvas $T = 9.6\exp(9 \times 10^{-6})\,N$ y $T = 3 \times 10^{-5}N - 6.9$ con un factor de correlación $r^2 = 0.9981$. Lo que nos lleva a concluir que la complejidad computacional de la simulación del modelo DLA es reducida de una tendencia exponencial a una lineal mediante el uso de nuestra técnica de búsqueda. Este resultado es comparable con la aceleración obtenida mediante el uso de mapas de bits a diferentes niveles de resolución [12, 39, 40, 41], pero el método usado en esta tesis tiene la ventaja de que no necesita almacenamiento adicional para los mapas externos, tampoco necesita actualizar varias representaciones jerárquicas del agregado y además puede utilizarse fácilmente para dimensiones euclidianas $d > 2$.

La dimensión fractal para los agregados fue calculada en base al radio de giro. La Tabla 1 muestra los valores de dimensión fractal encontrados a partir de varios experimentos. En esta tabla N representa al número de partículas en cada agregado, P es el número de procesadores empleados en la simulación, d es la dimensión del espacio que contiene al agregado y E es el número de experimentos ejecutados para cada cálculo. La dimensión fractal está representada por D, y σ es la desviación estándar para esta dimensión fractal. Se tomaron 10^3 muestras de la medida del radio de giro para cada experimento para distintas etapas de la evolución de cada agregado.

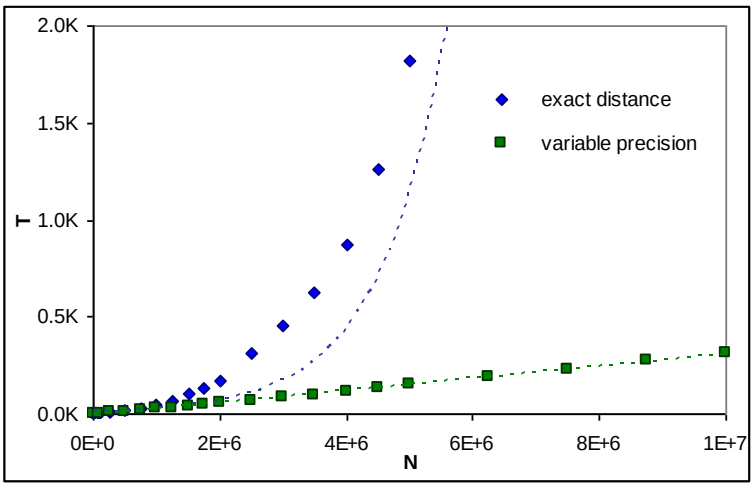

Figura 8. Rendimiento de las búsquedas de precisión variable por etapas y precisión exacta.

N	P	d	E	D	σ
10^{10}	4	2	5	1.716542	0.05793
10^9	4	2	80	1.712472	0.01292
10^9	1	2	10	1.710590	0.00466
10^9	4	3	10	2.47468	0.01644
10^8	4	3	80	2.503168	0.01715
10^8	1	3	50	2.531181	0.0082

Tabla 1. Dimensión fractal para 2D-DLA y 3D-DLA a gran escala.

Los bajos valores para la desviación estándar obtenidos cuando se utiliza un solo procesador (DLA) con respecto a cuándo se utilizan 4 procesadores (PDLA), nos indican que aun con las la técnicas utilizada para una rápida convergencia de los dos modelos, estos no son equivalentes, y esta discrepancia está cuantificada por σ. Al aumentar el número de procesadores, esta discrepancia puede incluso ser notoria en forma visual debido a la geometría más compacta de los agregados PDLA.

Para evaluar la convergencia de los agregados PDLA hacia el modelo DLA podemos considerar los datos de radio de giro obtenidos en ambos modelos. Los datos experimentales se muestran en la Figura 9. Aquí se puede apreciar la variación del radio de giro r_g definido por $r_g = \sqrt{\frac{1}{N}\sum_{i=1}^{N}|\vec{x_i}-\vec{x_0}|^2}$ con $\vec{x_i}$ como la posición de cada partícula en el agregado. Se aprecian valores para el PDLA con 4 procesadores y con uno solo para el DLA. Se puede apreciar en esta figura lo mencionado anteriormente con relación a las respectivas varianzas. Las barras de error

están amplificadas por un factor de 2 con la finalidad de ayudar a visualizar este hecho y los puntos fueron muestreados en intervalos separados de forma que no aparezcan traslapados.

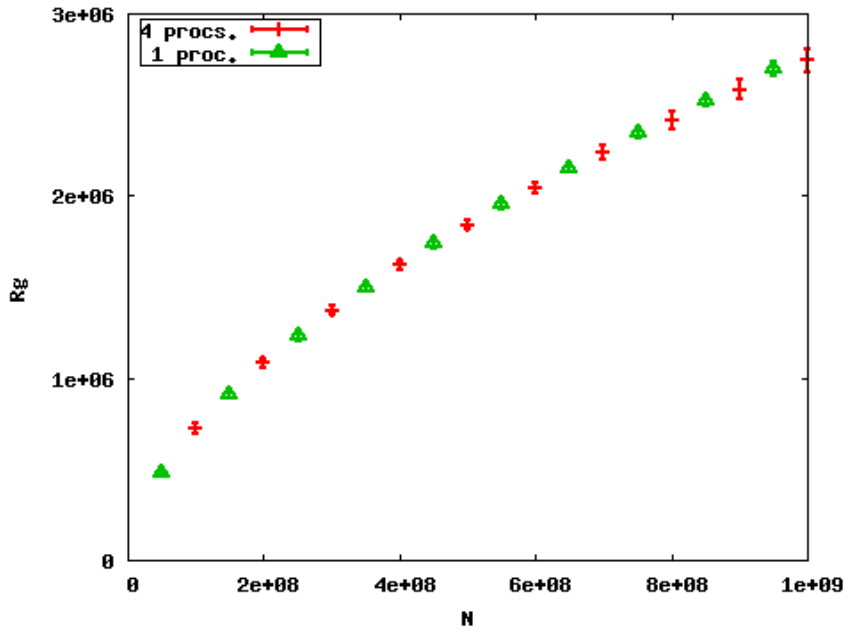

Figura 9. Variaciones en el radio de giro para modelos DLA (1 proc) y PDLA (4 procs).

Para evaluar la eficiencia en el uso del procesamiento en paralelo para la simulación del modelo PDLA observamos que la aceleración promedio fue de 3.74x cuando se trabaja con 4 procesadores, de acuerdo con la ley de Amdahl [45], se tiene que $acelracion = \frac{1}{s+p/n}$, con $s + p = 1$ con s representando a la parte serial del programa y p a la parte paralela. En este caso n representa al número de procesadores corriendo el programa simultáneamente. Con esto observamos que la fracción completamente paralela de nuestra implementación es de 97.5%, quedando 2.5% como la parte no paralelizable debida al código inherentemente no paralelizable en el DLA y al tiempo de bloqueo en el cual los hilos se encuentran esperando para la liberación de alguno de los *mutex* que protegen a la sección critica descrita anteriormente. Es decir, existe cierta dependencia de datos en el procedimiento, debidas a las actualizaciones en la estructura de datos. Este porcentaje de paralelización se considera aceptable considerando que en el modelo PDLA no podemos escalar arbitrariamente el número de procesadores cuando queremos que coincida con modelo DLA para agregados de gran tamaño.

V.8 Modelo de Rompimiento Dieléctrico.

El modelo de rompimiento dieléctrico (*Dielectric Breakdown Model-DBM*) de Niemeyer [4] permite modificar la dimensión fractal del agregado que se esté generando. Partiendo de un potencial de crecimiento $F = 1$ en la frontera de un dominio Ω y $F = 0$ en los puntos que pertenecen al agregado, el potencial de crecimiento en el resto del dominio se obtiene al resolver la ecuación diferencial $\nabla^2 F = 0$. Suponiendo que la simulación se realiza sobre una lattice bidimensional, la ecuación de Laplace puede escribirse:

$$\frac{\partial^2}{\partial x^2} f + \frac{\partial^2}{\partial y^2} f = 0 \qquad (39)$$

para cada punto (x, y)representado en la lattice por la posición (i, j) en la lattice con un valor de F denotado por $f(i, j)$ cuando se adquiere la forma discreta. Para encontrar numéricamente la solución a esta ecuación diferencial en forma discreta, consideramos la aproximación de la derivada de una función mediante diferencias centrales para un incremento $h \to 0$, observando la componente con respecto a la primer variable vemos que

$$f_x{}'(a) \approx \frac{f_x\left(a+\frac{h}{2}\right) - f_x\left(a-\frac{h}{2}\right)}{h} \qquad (40)$$

Al implementar esta fórmula en un espacio sobre una rejilla discreta, el mínimo valor del incremento que se puede tomar es $h = 1$, por lo que la discretización aproximada para la primera y segunda derivadas quedan

$$f_x{}'(a) \approx f_x\left(a + \frac{1}{2}\right) - f_x\left(a - \frac{1}{2}\right) \qquad (41)$$

$$f_x{}''(a) \approx f_x{}'\left(a + \frac{1}{2}\right) - f_x{}'\left(a - \frac{1}{2}\right) = f_x(a+1) + f_x(a-1) - 2f_x(a) \qquad (42)$$

Lo mismo ocurre con la segunda variable, por lo que la ecuación de Laplace se puede rescribir de manera discreta con f_x variando en el índice i y f_y en el índice j considerando el método de esténcil de 5 puntos mediante:

$$f(i, j-1) + f(i, j+1) + f(i+1, j) + f(i-1, j) - 4f(i, j) = 0 \qquad (43)$$

Donde la derivada ha tomado ahora forma ahora de diferencias finitas. Para encontrar el valor de $f(i, j)$ se puede aplicar el método de Euler para ecuaciones diferenciales ordinarias, que en forma general establece la ecuación iterativa

$$F_{k+1}(t) = F_k(t) + g\big(F'(t)\big)h \qquad (44)$$

para proporcionar un valor aproximado de $F(t)$ que cumple $F'(t) = g(F(t))$ mediante $F_k(t)$ a partir de una solución inicial $F_0(t)$.

Rescribiendo en forma discreta y considerando nuevamente un incremento mínimo $h = 1$. La solución aproximada que proporciona el método de Euler es:

$$f^{k+1}(i, j) = \frac{1}{4}\left(f^k(i-1, j) + f^k(i+1, j) + f^k(i, j-1) + f^k(i, j+1)\right) \qquad (45)$$

representando cada derivada parcial como una diferencia finita, cuya exactitud se incrementa evaluar la expresión sucesivamente hasta un valor de k suficientemente grande. El modelo

DBM considera la probabilidad de crecimiento en cada punto adyacente al agregado actual como:

$$P(i,j)_{\Phi} = \frac{f(i,j)^{\eta}}{\sum_{\Phi} f(i,j)^{\eta}} \qquad (46)$$

donde Φ es el conjunto de todas las posiciones adyacentes al agregado. La Figura 10 muestra la generación de agregados con el modelo DBM con valores $\eta = 2$ y $\eta = 0.2$, respectivamente. Estas figuras las generamos haciendo uso de GPU's. Las bandas de color mostradas corresponden a regiones equipotenciales, la escala en que se repite cada color es logarítmica por lo que la aparición de varias bandas consecutivas en intervalos regulares indica un decaimiento exponencial del valor del potencial. Este fenómeno aparece en las zonas profundas de las ramas (fiordos) y es menos notorio en las regiones externas de los agregados.

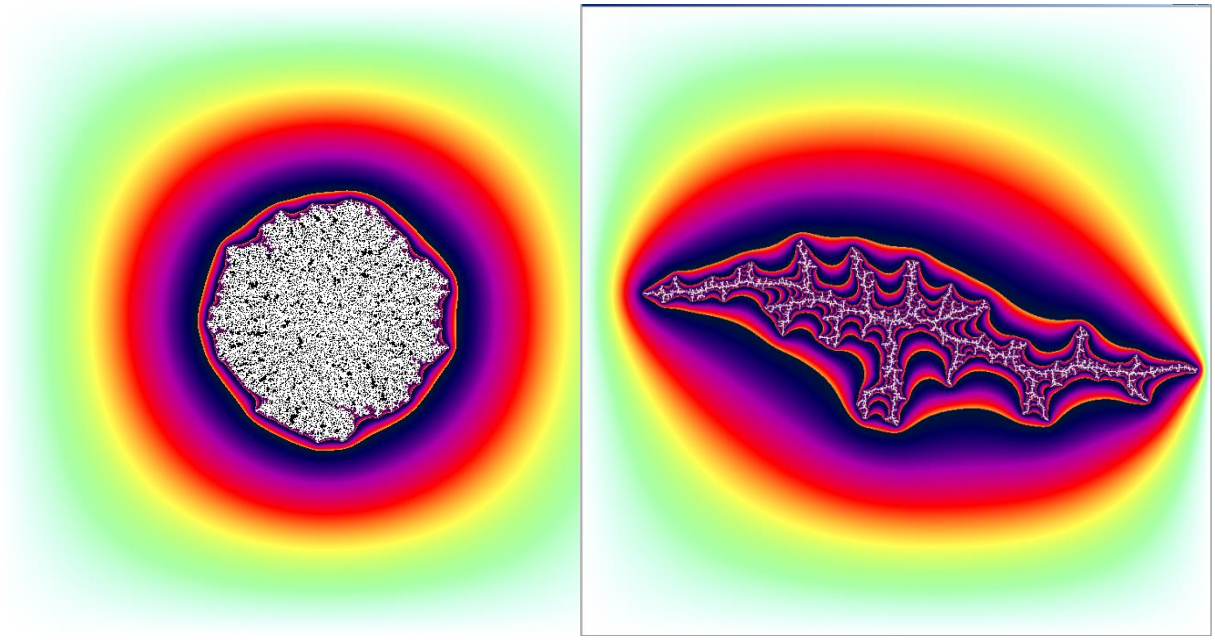

Figura 10. Modelos de DBM variando el exponente de determinación de probabilidad de crecimiento eta=2.0 y eta=0.2, respectivamente.

En la Figura 11 se observa un agregado DBM con parámetro $\eta = 1.0$, aunque se trata de un modelo diferente al DLA, la morfología es similar, sin embargo éste último muestra una dimensionalidad fractal ligeramente menor, debido en parte al procedimiento de crecimiento. Es importante notar que aunque la morfología del agregado en la Figura 11 a simple vista guarda una estrecha relación con el modelo DLA, estos dos procesos no son equivalentes, ya que en el modelo DBM no existe un difusión limitada, esto no es solamente por el hecho de que se esté realizando el cálculo del potencial en paralelo, sino porque el mismo modelo no establece un orden temporal en el que debe aplicarse la ecuación (46). Desde el punto de vista práctico de la simulación podemos remarcar el hecho notable de que el modelo DBM permite la aparición de ciclos cerrados en la estructura, algo que nunca ocurre en el modelo DLA.

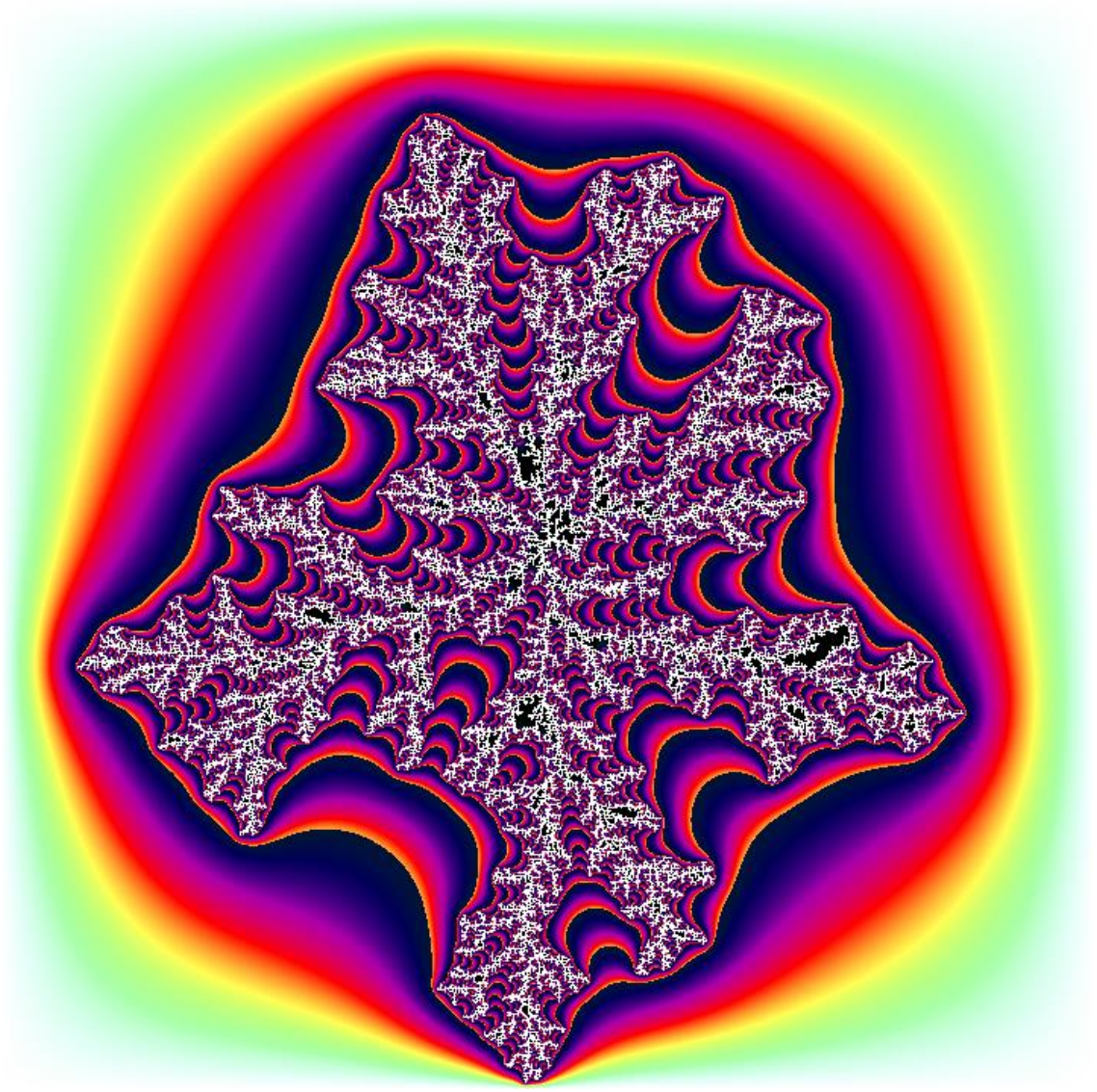

Figura 11. Cálculo del campo de potencial de crecimiento de una estructura dendrítica con el modelo DBM. Los cálculos y visualización se realizaron en los GPU's de una tarjeta gráfica nvidia GTX460 con *336 cores*.

V.9 Modelos híbridos basados en agentes.

Estos modelos están relacionados con los autómatas celulares [46]. Los modelos basados en agentes, abarcan un amplio rango de aplicación pueden describen varios fenómenos que ocurren en los campos de las ciencias biológicas, ecología y fenómenos sociales, y por supuesto, en el campo de la inteligencia artificial [48]. La principal diferencia entre los agentes de cómputo y los autómatas celulares consiste en que cada agente puede seguir diferentes reglas con respecto al resto de los elementos, lo que permite crear modelos complejos que incluso quedan clasificados dentro de modelos híbridos de simulación.

El modelado del movimiento browniano puede considerarse como un caso particular de los modelos basados en agentes. En el campo de las aplicaciones a fenómenos naturales podemos citar los modelos *diffusion limited aggregation* [2], *multiparticle aggregation* [6], *cluster-cluster aggregation* [24].

Desde un punto de vista computacional, estos modelos pueden simularse eficientemente con la ayuda del cómputo masivamente paralelo. No obstante, los modelos pueden presentar restricciones, como en el caso del modelo DLA, o bien, necesitar de sincronización excesiva entre cada elemento de proceso. En particular, para este tipo de modelos existe un alto potencial de que los agentes se encuentren en una condición de carrera (*race condition*) al modificar su estado y posición dentro de la memoria del dispositivo en el que se esté realizando la simulación.

Las técnicas modernas de cómputo paralelo proporcionan diferentes mecanismos para resolver condiciones de carrera, como son los *mutex*, los semáforos, las variables de condición y diversas operaciones atómicas [50]. Sin embargo, la introducción de mecanismos de sincronización siempre genera una sobrecarga de ejecución que repercute en el desempeño global de la ejecución del programa. Dependiendo de las características del algoritmo a implementar, esta reducción del desempeño puede superar a la ganancia introducida por el paralelismo [49].

Por tal motivo, para realizar las simulaciones en este trabajo, se busca obtener las sincronización entre procesos de la forma más eficiente posible, por esta razón es preferible utilizar directamente instrucciones de sincronización de bajo nivel. Es con estas instrucciones sobre las cuales se construyen los mecanismos anteriormente mencionados. Las instrucciones de sincronización de bajo nivel pueden ser accesadas a través de las APIs del sistema operativo o bien son proporcionadas por el controlador (*driver*) del dispositivo utilizado, y estas a su vez la reciben asistencia directa del hardware [44].

A continuación se describe la forma de conseguir esto en una GPU, con capacidades de cómputo de propósito general. Básicamente, dentro de la GPU los mecanismos de sincronización entre operaciones accesibles al programador están basadas en la operación `atomicCas()` (*compare and swap*) [29]. Cualquier operación atómica puede ser implementada a partir de `atomicCas()` en una GPU. La operación equivalente para un CPU convencional con capacidad de computo paralelo, por ejemplo con tecnología *multicore*, es `_InterlockedCompareExchange()` [50].

La operación de atómica de comparación e intercambio tiene el siguiente prototipo (en lenguaje C) :

```
int atomicCAS(int* dir, int compara, int nuevo);
```

La operación que realiza esta función es leer el valor anterior localizado en la dirección de memoria dir, y calcula:

```
(anterior == compara ? nuevo : anterior)
```

almacenando el resultado nuevamente en la dirección `dir`. Es decir, realiza el remplazo de un valor anterior con uno nuevo, siempre que el valor anterior sea igual a la referencia compara. Estas tres operaciones son realizadas en una sola transacción atómica. La función regresa el valor de anterior. La principal razón para utilizar esta función, es que la operación para calcular el valor nuevo, puede ser muy costosa en términos de tiempo de procesamiento, y realizarla de manera atómica requeriría mantener bloqueados a los demás elementos de proceso durante mucho tiempo. Además de que puede ser interrumpida en cualquier momento cuando se ejecuta en un sistema concurrente. La operación `atomicCas()` permite convertir a dicho procedimiento en una operación atómica optimizando el tiempo de bloqueo al mínimo.

Para ejemplificar la forma en que se puede lograr esto con la función `atomicCas()`, supongamos que se requiere realizar alguna operación atómica binaria, denotada por `oper()`, la cual consume un número relativamente elevado de ciclos de instrucción para actuar sobre el

dato de punto flotante de doble precisión que se encuentra en la dirección de memoria `dir`, pasando un segundo argumento proporcionado por val. Denotemos entonces a tal operación en su versión atómica por `operAt()`. Requerimos disminuir el tiempo de bloqueo de esta operación por lo que se implementa como se muestra en el Listado 2, utilizando la API de CUDA. La implementación para CPU con múltiples *cores* es similar.

Listado 2. Uso de la interrupción *Compare And Swap* para crear una operación atómica:

```
01    __device__ double operAt(double* dir, double val) {
02          double anterior = *dir, supuesto;
03          do {
04                supuesto = anterior;
05                anterior = atomicCAS(dir, supuesto,
                        oper( val, supuesto));
06          } while (supuesto != anterior);
07          return (anterior);
08    }
```

Nótese que la operación `operAt()` codificada en esta forma dará resultado de `oper()` como si se ejecutaran todas sus partes de manera atómica, supongamos que la operación es interrumpida entre las líneas 04 y 05, y el valor supuesto es cambiado por otro hilo en ejecución paralela compartiendo la misma dirección de memoria, entonces el ciclo `while` de las líneas 04 a 06 volverá a intentar la operación hasta que se realice en forma atómica, o bien, hasta que el valor anterior sea cambiado externamente de forma que coincida con el valor supuesto, dando el mismo resultado que produciría ejecutar la operación en forma atómica pero sin la sobrecarga de un tiempo excesivo de bloqueo.

El procedimiento anterior se utiliza para sincronizar a los agentes que representan partículas o elementos del agregado, de forma que se detecten colisiones entre diferentes partículas en movimiento o difusión.

Un ejemplo de generación de agregados con este tipo de modelos se muestra en la Figura 12. En este ejemplo se toma un modelo hibrido que considera probabilidades de crecimiento debido a potenciales de crecimiento inducidos por el mismo agregado bajo la condición de que se necesita un agente o partícula difundiéndose en esa misma posición en un momento determinado [51].

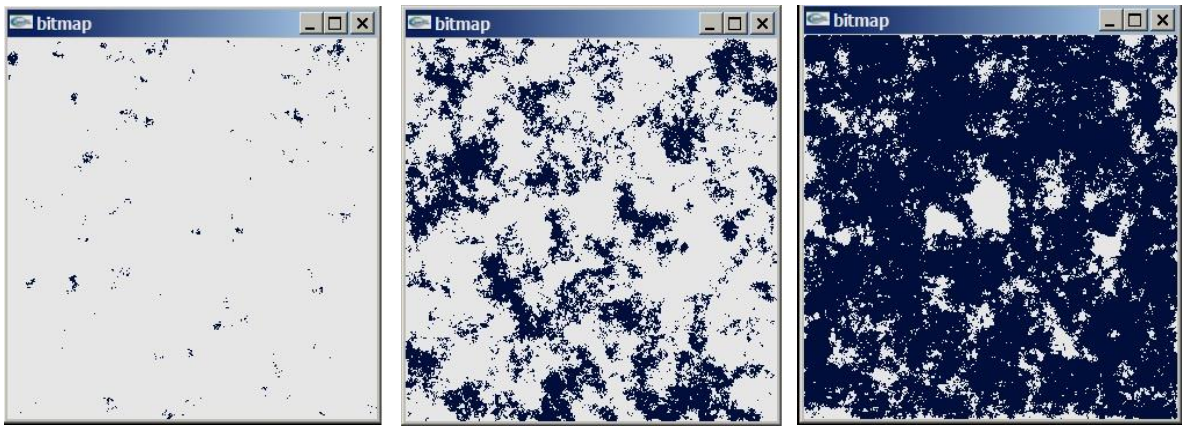

Figura 12. Modelo de crecimiento basado en agente y potenciales de crecimiento. El potencial de crecimiento es limitado en cada caso por un umbral u, los casos mostrados son u=2.0, u=3.0 y u=4.0.

El modelo utilizado para generar los agregados anteriores consiste en generar agentes o partículas con una probabilidad de aparición $p_a = 0.05$, similar al modelo de percolación, la diferencia es que en este modelo, los agentes no permanecen estáticos, sino que inician una caminata aleatoria, con lo que este modelo también exhibe características del modelo PDLA la probabilidad de agregarse cuando se encuentran entre sí está determinado por un potencial de crecimiento e_c generado por las mismas partículas el cual se dispersa en una unidad entre sus vecinos más cercanos en cada iteración, esta condición es parecida a la determinada por el modelo de crecimiento laplaciano descrita en la sección anterior. Cuando $e_c \geq u$ para un cierto u, entonces las partículas se adhieren al agregado. En la figura se muestran agregados con valores de $u = 2.0, u = 3.0, u = 4.0$, de izquierda a derecha, después de 500 iteraciones, con un potencial inicial aleatorio de ± 1 para cada localidad. El modelo se ejecuta realizando el movimiento de cada agente o partícula en paralelo, con la condición de bloqueo en caso de que la localidad a donde se moverá un agente ya se encuentre ocupada, lo que origina una condición de competencia [50]. Esta condición se resuelve mediante un bloqueo implícito originado al implementar la operación de movimiento como una operación atómica haciendo uso de la función `atomicCAS()`, -*Compare And Swap*- descrita anteriormente. La importancia de esta función es que permite implementar de forma atómica cualquier operación arbitraria, en nuestro caso, el movimiento de una partícula en un entorno lleno de otras partículas en movimiento, cuyo desplazamiento es controlado por hilos independientes. Como la condición de bloqueo solamente ocurre en caso de colisión, la eficiencia simulación no se ve afectada significativamente por tales bloqueos. Estas simulaciones se ejecutaron en una *GPU GTX 460*. El tiempo de generación de cada agregado es de 45 ms en promedio, lo cual facilita la aplicación de técnicas de análisis fractal descritas en el siguiente capítulo. Como se verá más adelante, ciertas técnicas de análisis de propiedades fractales y multifractales requieren de un espacio muestral bastante grande, por lo que la capacidad de generar agregados en una pequeña fracción de tiempo resulta de mucha utilidad.

VI. ANALISIS DE MODELOS

En este capítulo se presentan los detalles del análisis de los modelos de agregación generados con los mecanismos discutidos en el capítulo anterior. Dada la cantidad masiva de información relativa estos objetos geométricos, su análisis requiere el empleo de técnicas de cómputo paralelo, y de implementaciones eficientes. En las siguientes secciones se muestra los pasos realizados para cada aspecto analizado.

VI.1 Análisis de lagunaridad.

Como ya se ha mencionado, el análisis de objetos geométricos complejos puede realizarse utilizando medidas fractales y uno de los parámetros más importantes es la dimensión fractal, la cual da una descripción numérica de la cantidad del espacio rellenado por el objeto en forma invariante bajo escalas. Sin embargo, la dimensión fractal no describe completamente a un objeto con propiedades autosimilares. Por ejemplo, puede haber varios objetos con la misma dimensión fractal, pero con morfologías geométricas muy diferentes [1]. Otro parámetro útil proporcionado por la geometría fractal para caracterizar patrones geométricos intricados es el parámetro de lagunaridad [13,14,25]. La propiedad de lagunaridad proporciona una cubanización de que tan regularmente distribuido en el espacio se encuentra un conjunto. Formalmente puede definirse como la divergencia con respecto a la invariancia traslacional en un objeto [52]. Esta medida fractal ha sido satisfactoriamente empleada para clasificar la distribución espacial de varios tipos de conjuntos con respecto a varias escalas [25,53-56]. El concepto de lagunaridad puede también ser aplicado para examinar el comportamiento multifractal en conjuntos e imágenes [13,56]. Dado que la lagunaridad es una medida fractal, resulta de mucha utilidad para caracterizar estructuras con propiedades de escalamiento como las que aparecen en fracturas, agregados de partículas, interfaces entre fluidos y polímeros, entre otros.

Existen diversas formas para medir la lagunaridad [52-56]. El llamado "algoritmo de cajas deslizantes" (*gliding box algorithm*) propuesto por Allain y Cloitre [13] presenta algunas ventajas practicas al no involucrar cálculos aproximados de valores asintóticos, además de que admite como entrada tanto conjuntos fractales y no fractales y de que puede implementarse de manera directa en un sistema moderno de computo. No obstante una implementación directa requiere una gran cantidad de tiempo de cómputo. Por tanto, tienen que aplicarse varios ajustes para calcular la lagunaridad en objetos grandes. Tolle *et al* [14] han presentado una implementación del algoritmo *gliding box* que incrementalmente calcula las masas para las cajas deslizantes mediante la adición y substracción de hiperplanos que van apareciendo o van siendo dejados atrás a lo largo de la dirección de avance de la caja. Aunque el algoritmo dado en [14] es más rápido que una implementación directa [13], en este trabajo encontramos algunos problemas prácticos cuando tratamos de aplicar el procedimiento a los agregados generados por los modelos de crecimiento fractal. El principal inconveniente es la excesiva cantidad de tiempo de ejecución necesaria cuando se trabaja con simulaciones de conjuntos grandes o cuando la dimensión para los modelos va más allá de 2. Esto hace que la implementación mencionada anteriormente resulte bastante inadecuada. Esto es debido al hecho de que se necesita un gran nivel de precisión para obtener datos confiables en las simulaciones, que deben ser de un tamaño muy grande.

Con la finalidad de abordar este problema, se presenta en esta tesis una nueva implementación del algoritmo de cajas deslizantes que toma ventaja no solamente de los valores anteriormente calculados a lo largo de la dirección principal de movimiento, sino también a lo largo de cada posible dirección de movimiento relativa a las cajas que ya han sido evaluadas. Esto se hace aun si dichas direcciones no son aquellas para las cuales el movimiento se esta realizando

actualmente. Siguiendo esta idea fue posible obtener un significante incremento en la eficiencia del procedimiento de cajas deslizantes más allá de las implementaciones citadas arriba. Se presenta la codificación de la técnica mencionada empleando el lenguaje de programación C. El código puede analizar conjuntos en el espacio euclidiano N-dimensional para los cuales la dirección inicial esta especificada mediante un apuntador del tipo `void`. También se presenta una manera sencilla, pero efectiva para paralelizar el procedimiento utilizando programación multihilo compatible con el estándar POSIX [26] en un sistema de multiprocesamiento con memoria compartida.

VI.1.1 El algoritmo de cajas deslizantes.

El algoritmo de cajas deslizantes para cálculo de lagunaridad [13] mide el grado en el que un conjunto presenta variaciones traslacionales bajo diferentes escalas y puede ser descrito como sigue:

1) El conjunto S a ser analizado se coloca sobre una rejilla discreta de tal manera que cada punto de S es identificado por una posición en la rejilla.

2) Una caja de radio r es colocada sobre el conjunto, y luego el número M de puntos de S adentro de la caja (considerado como la masa de la caja) es contado.

3) El paso 2) es repetido moviendo la caja a través de la rejilla en cada posible posición sobre la rejilla.

4) Las frecuencias $n(M,r)$ dadas por el número de cajas de tamaño r con M puntos de S, se convierte a una probabilidad de distribución $Q(M,r)$ mediante la división del número total de cajas $N(r)$ que cubren al conjunto S.

5) Los momentos primero y segundo $Z^{(1)}, Z^{(2)}$ de la probabilidad de distribución $Q(M,r)$ son calculados, y entonces la lagunaridad $\Lambda(r)$ queda definida por

$$\Lambda(r) = \frac{Z^{(2)}}{(Z^{(1)})^2} \tag{47}$$

donde

$$Z^{(q)} = \sum_{M} M^q Q(M,r) \tag{48}$$

La referencia [14] muestra como la ecuación (48) puede ser expresada en términos de masas M_i contenidas en la *i*-ésima caja deslizando, de manera que los momentos pueden establecerse como:

$$Z^{(q)} = \frac{1}{B(r)} \sum_{i=1}^{B(r)} {M_i}^q \tag{49}$$

donde $B(r)$ es el número de cajas de radio r que pueden ser definidas en la rejilla.

VI.1.2 Cálculo eficiente del algoritmo de cajas deslizantes.

Como puede verse en la sección previa, la mayor parte del tiempo de cálculo requerido para obtener la lagunaridad de un conjunto es consumido por el conteo de elementos de S que caen adentro de una caja de radio r en la parte iterativa del proceso indicada por los pasos 2) y 3). Po lo cual es necesario implementar esta parte de manera que la cuenta de estos elementos se realice en forma efectiva. Los autores Tolle, *et al* [14] han presentado un método para realizar este cálculo. Su método consiste en emplear la masa m_k calculada para la última caja B_k, y luego calcular la masa m_{k+1} para la siguiente caja B_{k+1} la cual ha sido desplazada una unidad en la rejilla en una dirección específica, en su documento utilizan una expresión equivalente a la siguiente:

$$m_{k+1} = m_k + l_k - t_k \qquad (50)$$

donde l_k y t_k son respectivamente las masa de los hiperplanos anterior y siguiente que necesitan ser sumados y restados de la cajas B_k para construir la caja B_{k+1}. Un esquema ejemplificando dicha construcción para un conjunto tridimensional se muestra en la figura Fig. 1 a).

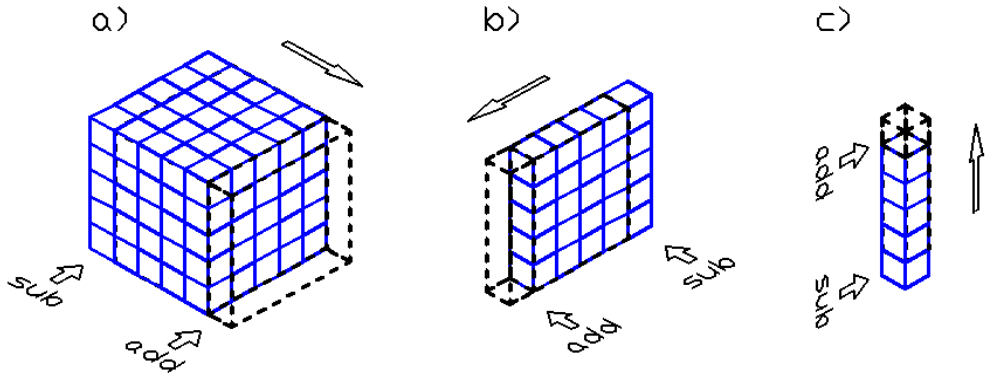

Figura 13. Calculo de masas de cajas N-dimensionales utilizando la diferencia de hiperplanos a). Esta puede ser extendida a la determinación de los mismo hiperplanos b). En el último nivel de recursión, solamente se cuentan elementos individuales c).

Aunque esta técnica notablemente mejora el tiempo necesario para contar las masas de todas las cajas necesarias para obtener $\Lambda(r)$, puede resultar insuficiente cuando se trabaja con objetos muy grandes o cuando se utiliza dimensionalidades relativamente altas. La propuesta para mejorar este método es aplicar el mismo esquema para calcular las masas de las cajas incrementalmente, pero ahora utilizada recursivamente para la determinación de las masas l_k y t_k de los hiperplanos que son usados para obtener m_{k+1}. Por ejemplo, para calcular l_k, el procedimiento requerirá también el cálculo de las masas para los sub-hiperplanos anterior y siguiente l_{lk} y t_{lk}, y así sucesivamente. Esta idea esta gráficamente representada en la Fig. 13 b) y c), donde los hiperplanos recursivamente encontrados se muestran como en una posición desfasada con respecto a las posiciones previamente evaluadas. Para aplicar este nuevo mecanismo es necesario mantener registros de las masas obtenidas en pasos previos para cada dimensión correspondiente al mismo nivel de recursión. Nótese que no todas las masas de los hiperplanos necesitan ser almacenadas, solamente aquellas calculadas en el último paso de la recursión. Puesto que tenemos que para un objeto N-dimensional de tamaño lineal L es necesario mantener L^0 registros en el nivel 1 de recursión. Este registro corresponde a la última caja evaluada. Al nivel 2 de recursión L^1 valores de masa de los hiperplanos previos son

necesarios, el nivel 3 requiere L^2 registros. Por inducción, tenemos el número total de registros requeridos $n_{rec}(L)$ es

$$n_{rec}(L) = \sum_{i=0}^{N-1} L^i \tag{51}$$

A primera vista, esto puede parecer muy demandante en cuanto capacidad de memoria, sin embargo, se debe notar que la cantidad anterior representa solamente una pequeña proporción del objeto a analizar. Por ejemplo, si se trabaja con una imagen bidimensional, la cantidad de registros adicionales necesarios será equivalente a solamente agregar una línea y un punto a la imagen. No obstante, si la cantidad de memoria disponible en un factor determinante, se puede establecer una bandera en cada nivel der recursión, indicando si existen registros previos para simplificar la evaluación de las masas, de otra forma se realizara un cálculo complete. Con este diseño, la velocidad de ejecución puede ser intercambiada por utilización de memoria.

VI.1.3 Implementación en lenguaje C.

En esta sección se presentan y explican los detalles de la implementación del algoritmo de cajas deslizante hacienda uso de la propuesta para mejorar el tiempo de ejecución mediante el uso de registros de cuentas anteriores a lo largo de todas las direcciones de desplazamiento.
La función que realiza el cálculo de las masa de los hiperplanos tomando en cuenta los valores previamente calculados a lo largo de cada dirección de movimiento es `massHyperPlane()`. Esta función recursiva regresa la masa de un hiperplano y toma como parámetros el nivel de recursión `lv`, la posición inicial `pos[]` del hiperplano, el límite superior de la extensión del objeto `llmt[]`, un vector `desp[]` indicando el desplazamiento a través del proceso recursivo y el radio *r* de la caja deslizante, y un parámetro adicional `thd` para permitir llamadas simultaneas por diferentes hilos de ejecución. El código para `massHyperPlane()` se muestra a continuación. La función `massHyperPlane()` implementa la técnica que hace posible la mejora en velocidad de ejecución conseguida en este trabajo.

Listado 3. Cálculo masas de hiperplanos.

```
00  int massHyperPlane(int lv,int pos[],int desp[],int llmt[],
                       int r,int thd){
01      int lk,tk;              //masa hiperplanos anterior y siguiente
02      int res[N];             //resultado de sumar vectores
03      int mk=0;               //cuenta total de masa
04     if (lv==N){              //aqui los hyperplanos son 0-dimensional
05         addVectors(pos,desp,res,N);
06         mk=getElement(res,AA,N);
07     }else{
08         if (pos[lv]==llmt[lv]){         //no hay registros previos
09             for(desp[lv]=0;desp[lv]<r;desp[lv]++){
10                 mk+= massHyperPlane(lv+1,pos,desp,llmt,r,thd);
                   //recursivamente calcula las masas
11                 addVectors(pos,desp,res,lv);
12                 setRecord(mk,res,lv,RCRDS[thd],N)  //salva el valor
13             }
14         }else{
15             desp[lv]=r-1;
16             lk= massHyperPlane(lv+1,pos,desp,llmt,r,thd);

               //recursion sobre los hiperplanos
17             addVectors(pos,desp,res,N);
```

```
18            res[lv]=pos[lv]-1;
19            if (lv==N-1){
                 //recupera un registro o elemento del conjunto
20                tk=getElement(res,AA,N);
21            }else{
22                tk= getRecord(res,lv+1,RCRDS[thd],N);
23            }
24            mk= getRecord(res,lv,RCRDS[thd],N)+lk-tk;
              //incrementalmente calculate la masa
25            addVectors(pos,desp,res,lv);
26            setRecord(mk,res,lv,RCRDS[thd],N);     //salva el valor
27        }
28    }
29    return mk;
30  }
```

En el código mostrado arriba podemos ver que el nivel más profundo de recursión se alcanza en las líneas 05 a 06. Aquí los elementos individuales de conjuntos bajo análisis son regresados como las masas actuales. Las líneas 08 a 13 cuentan los hiperplanos recursivamente y almacenan los valores encontrados al respectivo nivel. Las líneas 14 a 24 incrementalmente estiman la cuenta del hiperplano al nivel correspondiente mediante la obtención de sub-hiperplano siguiente y la consulta del valor del sub-hiperplano anterior. Las funciones *getRecord()* y *setRecord()* respectivamente recuperan y establecen un elemento de dato indicando la masa de una caja previamente calculada a lo largo de cada dimensión. *getRecord()* toma como parámetros un arreglo *Crds[]* indicando las coordenadas del elemento, un apuntador nulo *H* con la dirección inicial del arreglo para almacenar los registros, y el número de dimensiones *N*. *setRecord()* toma un parámetro adicional *elem*, que indica el valor a establecer. El nivel de recursión *lv* tiene que ser parado con la finalidad de almacenar el dato *elem* en su lugar correspondiente.

Listado 4. Guardar y recuperar registros.

```
00  inline void setRecord(int elem,int Crds[],int lv,void *H, int N){
01      void *p=H;                          //apuntador al elemento
02      int i;
03      p=((void **)H)[lv];
04      for (i=0;i<lv-1;i++){
            //iterativamente apunta a la direccion correcta
05          p=((void **)p)[Crds[i]];
06      }
07      ((int *)p)[Crds[lv-1]]=elem;           //guarda el registro
08  }


09  inline int getRecord(int Crds[],int lv,void *H, int N){
10      int elem;
11      void *p;                    //apuntador al elemento
12      int i;
13      p=((void **)H)[lv];
14      for (i=0;i<lv-1;i++){
            //iterativamente apunta a la direccion adecuada
15          p=((void **)p)[Crds[i]];
16      }
17      elem=((int *)p)[Crds[lv-1]];      //obten el registro
18      return elem;
19  }
```

El código para la función *getElement()* es muy similar a *getRecord()*, pero para este caso no es necesario pasar el parámetro *lv* y el apuntador enviado apunta al comienzo del arreglo N-dimensional que almacena al objeto a ser analizado en lugar de apuntar a los resultados de los cálculos a ser almacenados. La función *addVectors()*, suma los vectores apuntados por sus primeros dos parámetros y almacena el resultado en el arreglo indicado por su tercer parámetro. La suma es realizada hasta la longitud establecida por su cuarto parámetro. Otra función importante utilizada es *glidingBox()* la cual realiza el proceso especificado en los pasos 2) y 3) de la descripción del algoritmo. Esta función puede ser codificada con llamada a la función propuesta para contar las masas de los hiperplanos *massHyperPlane()* como sigue:

Listado 5. Cálculo de cajas deslizantes.

```
00  void glidingBox(int lv,int r,int pos[],int llmts[],int ulmts[],int
thd){
01     unsigned lk,tk;  //masas de hiperplanos siguiente y anterior
02     unsigned mk;           //cuenta masa total
03     int itk,i;       //itk indice del hiperplano siguiente
04     int desp[N];           //desplazamiento de la posicion actual
05     if(lv){
06          for(pos[lv]=llmts[lv];pos[lv]<=(ulmts[lv]-r+1);pos[lv]++)
               //recursivamente itera hasta el nivel 0
07              glidingBox(lv-1,r,pos,llmts,ulmts,thd);
08     }else{
09          mk=0;
10          for(pos[0]=llmts[0];pos[0]< llmts[0]+r;pos[0]++){
               //desliza la caja en el primer eje hasta r
11              for (i=0;i<N;i++) desp[i]=0;
               //cuenta elementos para la caja inicial
12              mk+=massHyperPlane(1,pos,desp,llmts,r,thd);
13          }
14          saveBoxCount (mk,thd);
15          for(pos[0]=llmts[0]+r;pos[0]<=ulmts[0];pos[0]++){
               //desliza la caja en el primer eje desde r
16              for (i=0;i<N;i++) desp[i]=0;
17              lk=massHyperPlane(1,pos,desp,llmts,r,thd);
               //evalua masa hiperplano siguiente
18              itk=pos[0]-r;
19              tk=getRecord(&itk,1,RCRDS[thd],N);
               //recupera la masa hiperplano anterior
20              mk=mk+lk-tk;
21              saveBoxCount(mk,thd);
22          }
23     }
```

En la función *glidingBox()* el parámetro *lv* representa el nivel de recursión, *r* el tamaño de la caja deslizante, *pos[]* indica la posición actual de la caja, y *ulmts[]* establece las fronteras N-dimensionales superiores de la rejilla en la cual el objeto está colocado. Las líneas 05 y 07 realizan una recursión sobre las N dimensiones; la masa para la caja inicial es evaluada en las líneas *09* a *12*. La masa para las cajas siguientes es evaluada incrementalmente en las líneas 15 a 21. Nótese en este esquema propuesto no hay necesidad de calcular simultáneamente la masa del hiperplano anterior como en la referencia [14], esto es debido a que la masa ya ha sido calculada cuando el hiperplano correspondiente fue un hiperplano siguiente, por lo que solo es necesario recuperar el registro. Las líneas *18* y *19* hacen esto. Se puede ver que existe mucha similaridad entre las funciones *massHyperPlane()* y *glidingBox()* de tal forma que pudiera pensarse que pueden haberse escrito como unas función. De hecho ese es el caso. Sin embargo, se escribieron como dos funciones separadas para evitar una excesiva cantidad de pruebas de condición adentro del proceso que es ejecutado por *massHyperPlane()*.

La función `saveBoxCount()` colecta los valores para obtener las frecuencia $n(M,r)$ indicadas en el paso 4) para obtener los momentos de distribución usados en el paso 5) del algoritmo de cajas deslizantes. El código para esta función depende de que expresión este siendo usada para obtener los momentos. Si se está usando la ecuación (50) la función tiene que registrar una ocurrencia del valor M_i por ejemplo incrementando el valore de la posición en i en un arreglo. Cuando se usa la ecuación (51), la función tiene que elevar M_i a la potencia q y luego sumar el valor a una cuenta global. Claramente el segundo caso consume mucho más ciclos de instrucción, aproximadamente 16 ciclos más [47]. El código para el prime caso se muestra a continuación.

Listado 6.

```
00  inline void saveBoxCount(int mass){
01    if (mass<=MAXMS){       //verifica indice valido
02           M[mass]++;
03    }
04  }
```

En el código anterior *MAXMS* es la definición de una macro indicando el máximo valor permitido para la masa y es proporcional a r^N. M es un arreglo global para almacenar las frecuencias $n(M,r)$. Aunque este método requiere más memoria comparado con la aplicación de la ecuación (51), esta última no introduces errores de redondeo y se ejecuta mucho más rápido cuando se requieren momentos pertenecientes a diferentes órdenes [55-56].

VI.1.4 Paralelización.

En esta sección se muestra como paralelizar el procedimiento anterior para obtener la lagunaridad de un objeto. En el esquema aquí usado existe una dependencia de datos porque cada cálculo de conteo de masa dentro de una caja utiliza los datos de cajas previamente calculadas para conseguir un aumento en la velocidad de ejecución. No obstante es posible paralelizar el código obteniendo un buen factor de ganancia en tiempo de ejecución mediante una partición de dominio simple para el objeto entero y asignando cada sección del dominio un proceso paralelo diferente. Para asegurar de que cada proceso contara todos los elementos en las cajas como si no existiera ninguna partición, los dominios de la partición necesitan ser traslapados en las orillas por $r-1$ elementos. La Figura 14 ilustra esta situación mostrando un ejemplo bidimensional donde los dominios para tres procesos han sido particionados a lo largo de una dimensión.

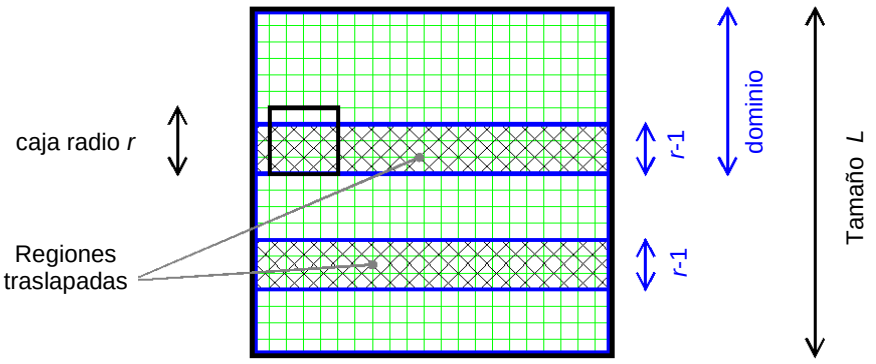

Figura 14. Particionamiento de dominio a lo largo de una dimensión para paraleliza el algoritmo gliding box.

En el siguiente código se da un ejemplo de cómo realizar tal paralelización basándose en la partición de dominios para un sistema de multiprocesamiento de memoria compartida *Shared Memory Processing* (SMP) utilizando *POSIX pthreads* [13]. Una explicación detallada de cómo usar las funciones de la librería pthreads puede encontrarse en la librería [14]. Para nuestro programa primero codificamos la función *glideBox()* que regresa un apuntador void como parámetro tal y como lo requiere la función de la librería *pthread pthread_create()*[13]. La función *glideBox()* es usada para enviar *glidingBox()* para su ejecución por distintos hilos.

Listado 7.

```
00  void *glideBox(void * s){
01    int pos[N];                          //posicion inicial de la caja
02    glidingBox(N-1,((struct params *)s)->r,pos,
03        ((struct params *)s)->llmts,((struct params *)s)->ulmts,
04        ((struct params *)s)->thd);
05    return NULL;
06  }
```

La estructura de datos referenciada por el parámetro de apuntador void *s* tiene la siguiente forma:

Listado 8.

```
00  struct Params{
01    int r;                  //radio de la caja
02    int llmts[N];           //límite inferior por  cada dimensión
03    int ulmts[N];           //límite superior por cada dimensión
04    int thd;                //número de hilo en ejecución paralela
05  };
```

Se utiliza el arreglo global *THD* del tipo *pthread_t* [13] para almacenar cada referencia al hilo creado y un valor constante *NUMTHD* indicando cuantos hilos van a ser lanzados para ejecución paralela. Para crear la partición del dominio discutida anteriormente, *glideBox()* puede ser llamada de la siguiente manera:

Listado 9. Paralelización del algoritmo cajas deslizantes.

```
#include <pthread.h>
struct Params params[NUMTHD];
//aquí va código auxiliar
01  for (thd=0;thd<NUMTHD;thd++){
02      params[thd].r=r;            //establece parámetros de llamada
03      params [thd].thd=thd;
04      for (i=0;i<N;i++){          //establece límites para las
secciones
05          params [thd].llmts[i]=0;
06          params [thd].ulmts[i]=oblmt[i];
07      }
08      secSz=(oblmt[N-1]+1)/NUMTHD;      //crea la partición de la
rejilla
09      params [thd].llmts[N-1]=thd*secSz;
10      if (!(thd==NUMTHD-1)){
            //los límites del ultimo dominio no  cambian
11          params [thd].ulmts[N-1]= params [thd].llmts[N-1]+secSz+r-
2;
12      }
```

```
13      pthread_create(&THD[thd],&attribThd,&glideBox,& params[thd]);
14  }
```

El arreglo *oblmt[]* en el código indica los límites para el objeto a ser analizado. La variable entera *secSZ* es el tamaño de la sección en unidades de rejilla para cada partición en el procesamiento multihilo.

VI.1.5 Medición del desempeño.

En esta sección comparamos el desempeño del código aquí propuesto con respecto a las implementaciones citadas en las referencias [13,14]. Para hacer esto aplicamos el código sobre conjuntos aleatoriamente generados con dimensión euclidiana D y tamaño lineal L. de forma que el número de elementos en cada rejilla es L^D . La prueba de velocidad fue ejecutada con un equipo estándar con procesador *Intel Celeron* a 1.4 GHz. El código fue compilado usando el *kernel 4.4.2* de Linux [43]. El desempeño fue probado utilizando solamente un hilo de ejecución. En la Figura 15-a) se puede ver una gráfica del tamaño lineal del objeto contra el tiempo de ejecución requerido para obtener $\Lambda(r)$ para un objeto D-dimensional utilizando un tamaño de caja $r = L/2$.

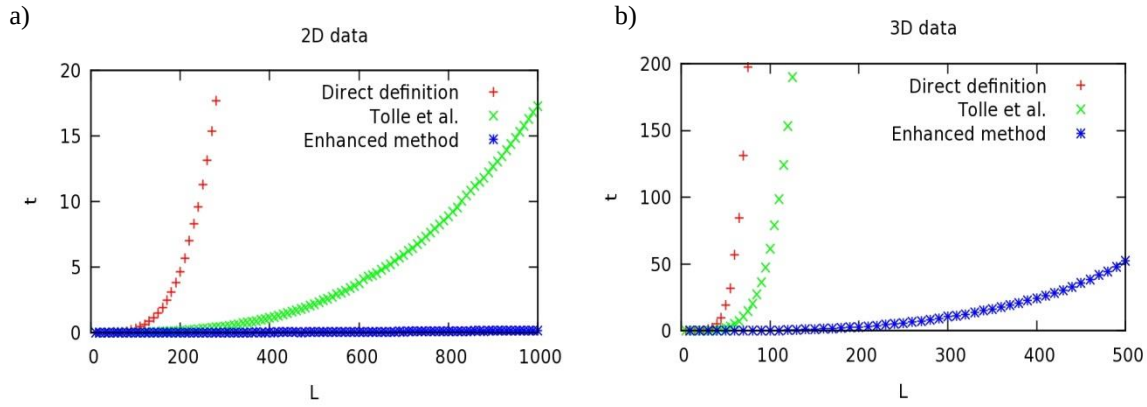

Figura 15. Tiempos de ejecución para cálculo de lagunaridad con diferentes métodos trabajando con datos 2D y 3D.

Los métodos evaluados son implementación directa, i. e. contar todos los elementos para cada caja, el método propuesto por Tolle *et al*. [14], y el método descrito en este trabajo. La Figura 15-b) muestra la misma comparación con $D = 3$. Ambas graficas muestran una mejora significante en el tiempo de ejecución al aplicar las ideas aquí propuestas para la implementación del código.

La Figura 16-a) muestra el desempeño del método dado en [14] trabajando con un conjunto de datos con dimensión D variando de 2 a 6. El tiempo de ejecución t en segundos esta mostrada en una escala logarítmica. Puede notarse que el tiempo de ejecución para $D = 6$ alcanza cerca de 8 órdenes de magnitud. En la Figura 16-b) presentamos tiempos de ejecución para el cálculo de $\Lambda(r)$ en objetos embebidos en las dimensiones 2 a 6 utilizando nuestro método. En este último caso el orden de magnitud más grande alcanzado para $D = 6$ es menor a 2 para objetos con el mismo tamaño lineal.

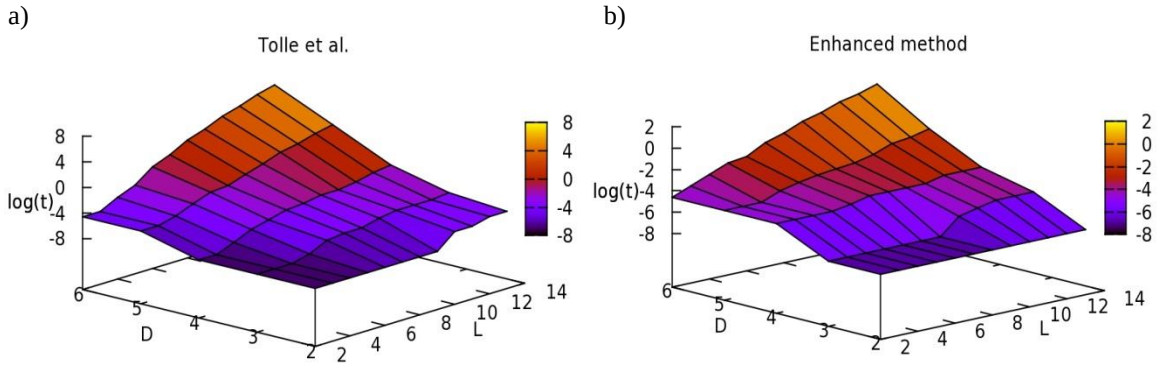

Figura 16. Tiempos de ejecución para el cálculo de lagunaridad trabajando con objetos en dimensiones euclidianas de 2-6.

VI.2 Analisis de multifractalidad.

La multifractalidad se extienda la idea de autosimilaridad de conjuntos hacia las medidas que pueden ser definidas sobre ellos. Cuando aparece una singularidad en una medida definida sobre un objeto en diversas escalas se dice que la medida es multifractal [8]. Para aplicar dicho concepto de manera práctica, nos centramos ahora en una métrica tangible dentro de nuestros modelos de simulación. La métrica escogida para esta investigación es la misma masa del objeto analizar, lo que nos lleva al concepto de multifractalidad de masa, el cual fue introducido por Tel y Vicksek [18] y se refiere a las dimensiones fractales de las distribuciones locales de masa en una estructura dada. La multifractalidad de masa no involucra medidas singulares, sino que en su lugar la masa del objeto es tomada para analizar propiedades multifractales. Mientras que las propiedades multifractales están bien documentadas para la medida armónica en agregados DLA, existen pocas investigaciones que consideren la multifractalidad de masa para estos. La medida multifractal harmónica está dada por la probabilidad de crecimiento en un sitio, en las referencias [9], [23] y [57] se encuentra un compendio detallado con respecto a este fenómeno multifractal en el DLA. Por otra parte, para el caso de la multifractalidad de masa, experimentalmente se han encontrado propiedades multifractales para la distribución geométrica de los agregados DLA mediante la estimación de diferentes valores de dimensión generalizada D_q para diferentes valores de q [31].

Un espectro continuo de diferentes valores de D_q en estas estructuras explicaría la razón de porque diferentes valores de dimensión fractal aparecen cuando se utilizan diferentes métodos de estimación. De hecho, a una escala media de simulación (con agregados cuya masa se encuentra entre 10^4 y 10^6 partículas), el método de conteo de cajas (*box-counting*) [39] da estimaciones de dimensión fractal más bajos que los métodos basados en escalamiento masa-radio [40, 41].

No obstante la evidencia anterior, el autor Lam [60] ha hecho notar algunas inconsistencias del método de caja de arena (*sand-box method*) utilizado en [31], las cuales discutiremos más adelante, atribuye las variaciones con respecto a una sola dimensión fractal a efectos de tamaño finito más que a un proceso multiplicativo que pudiera darse en la distribución de la masa en el agregado. Para abordar el punto anterior, consideramos importante realizar experimentos para el análisis del comportamiento de la multifractalidad de masa en estructuras DLA utilizando un método que puede tratar los casos de estimación de las dimensiones generalizadas D_q involucrando valores positivos y negativos de q. Incluimos el método de *sand-box* en este

estudio puesto que es importante observar su comportamiento cuando los efectos de tamaño finito son reducidos, de forma que podemos verificar si las conclusiones de la referencia [31] son experimentalmente validas desde el enfoque aquí presentado.

Para conseguir esto, generamos agregados DLA suficientemente grandes utilizando la metodología descrita en el capítulo anterior y comparamos los resultados dados por los métodos de *box-counting*, sand-box y el método de cajas extendidas (extended-box) de los autores Pastor y Riedi [19], el cual, al igual que el método de *sand-box*, presenta poca influencia por los efectos de borde de tal forma que puede ser utilizado para evaluar D_q tanto para valores positivos como negativos de q. Otro aspecto del DLA en el cual se han observado desviaciones del de la estructura geométrica con respecto a un fenómeno monofractal es en el de lagunaridad, concepto tratado en la sección anterior. B.B. Mandelbrot y colaboradores [61] han encontrado dos valores diferentes de dimensión fractal: 1.71 y 1.67 asociados a la estructura de agregados DLA cuando se analizan los espacios angulares entre las ramificaciones de los agregados. Con esto se encontraron que ambas dimensiones eran constantes a través de varias escalas. Para tratar este fenómeno, aquí también se examinan las propiedades de lagunaridad de en los agregados DLA con el enfoque de cajas deslizantes.

Como lo mencionan sus autores, las estadísticas del método de cajas deslizantes está relacionado con las dimensiones generalizadas D_q. Como aplicación práctica, el método de cajas deslizantes pueden ser extendidas al análisis multifractal obteniendo un muestreo más amplio de datos que el método de conteo de cajas, aunque estas muestras no son independientes debido al traslape de cajas contiguas. Esta dependencia acentúa los efectos de borde que aparecen debido al hecho de que las muestras con valores muy bajos (comparado con el máximo obtenido de la totalidad de las cajas) aparecen en todas las escalas dentro de las regiones cercanas a las orillas de la estructura. Por tanto, resulta demasiado difícil rastrear el comportamiento exponencial que presentan los momentos negativos para el agregado (o para cualquier otro objeto geométrico) en la distribución de la masa.

La siguiente sección describe la noción de análisis multifractal utilizando los métodos conteo de cajas y de cajas deslizantes. Se presentan y discuten los resultados al aplicar estos métodos al análisis de la distribución de la masa en agregados DLA en 2D y 3D.

VI.2.1 Análisis multifractal mediante conteo de cajas.

Como se expuso anteriormente, el escalamiento en una medida singular puede ser descrito a través del exponente de Holder α como sigue:

$$\alpha = \frac{\log \mu(B_i(r))}{\log r} \quad . \tag{52}$$

Aquí $\mu(B_i(r))$ representa a la medida adentro de la i-th caja de radio r sobre una división de dominio no refinada (*coarse-grained*) sobre la cual está colocado el objeto fractal. Para cada valor de α, se puede evaluar el número de cajas $N_r(\alpha)$ de tamaño lineal r con un exponente de Holder igual a α. Para rastrear el comportamiento de las frecuencias de los valores de α conforme $r \to 0$ se considera la siguiente función

$$f_r(\alpha) = \frac{\log N_r(\alpha)}{\log r} \quad . \tag{53}$$

Y por tanto, $f(\alpha)$ tiene un comportamiento de escala $N_r(\alpha) \sim r^{-f(\alpha)}$.

Halsey, et al. [17] propusieron el método de los momentos para estimar $f(\alpha)$. Este método utiliza la función de partición:

$$\chi_q(r) = \sum_{i=1}^{N(r)} \mu(B_i(r))^q \qquad (54)$$

donde $N(r)$ es el número total de cajas con radio r. Las cajas se toman de una rejilla sobre la cual está el objeto a analizar. La función de partición tiene el comportamiento de escalamiento descrito por:

$$\chi_q(r) \sim r^{\tau(q)} \quad , \qquad (55)$$

$\tau(q)$ puede ser encontrado experimentalmente de (55) mediante un ajuste lineal sobre el conjunto de valores $\log(\chi_q(r))$ vs $\log(r)$. Los valores de α y $f(\alpha)$ pueden entonces ser calculados de $\tau(q)$ mediante la siguiente transformada de Legendre

$$\tau(q) = q\alpha(q) - f(\alpha(q)) \qquad (56)$$

Y finalmente las dimensiones generalizadas [62] para cada $q \in \Re$ están definidas por $D_q = \dfrac{\tau(q)}{q-1}$.

Cuando aplicamos el método de los momentos a objetos que contienen espacios vacíos mayores que el tamaño de la caja, aparecen cajas con valores muy pequeños para $\mu(B_i(r))$.

Estas cajas hacen difícil de rastrear D_q para $q < 0$ debido a que los momentos correspondientes contribuyen con valores muy grandes a la ecuación (54), y por tanto la función de partición se vuelve insensible a variaciones a través de las escalas par estimaciones negativas de q. Para resolver este problema, T. Vicsek et al. [31] desarrollaron el método generalizado de cajas de arena (*generalized sand box method*) y lo aplicaron para determinar las dimensiones generalizadas en agregados DLA. El método considera únicamente las cajas centradas en puntos del objeto y se basa en la siguiente relación:

$$\sum_i \left(\frac{M_i}{M_0}\right)^q \sim \left(\frac{r}{L}\right)^{(q-1)D_q} \qquad (57)$$

donde M_i es la masa contenida en la i-ésima caja de tamaño r, M_0 es la masa total del objeto y L es su tamaño lineal; es decir, la máxima extensión del objeto a lo largo de alguna dimensión. Los autores del método *sand-box* consideran $(M_i/M_0)^q$ como el valor esperado de $(M_i/M_0)^{q-1}$ conforme a la probabilidad de distribución (M_i/M_0). Entonces aplican la ecuación (59) al promedio de la masa $M(r)$ a aquellas cajas cuyos centros coinciden con puntos pertenecientes al objeto. Luego llegan a la relación

$$\left\langle \left(\frac{M(r)}{M_0}\right)^{q-1} \right\rangle \sim \left(\frac{r}{L}\right)^{(q-1)D_q} \quad . \qquad (58)$$

Este procedimiento no produce cajas con valores de masa bajos, por tanto reduce los efectos de borde. En la referencia [31], la relación (58) es aplicada para analizar la multifractalidad de masa en agregados DLA, encontrando un rango continuo de diferentes valores D_q. Sin embargo, C. Lam [60] hace notar que debido a la homogeneidad de los agregados DLA, la

relación $M(r) \sim r^D$ implica $D_q = D$ para todo q. Lam muestra que para valores grandes de $|q|$, la relación (60) está dominada por $M_{\max}(r)$, cuándo $q > 0$ y por $M_{\min}(r)$ cuando $q < 0$, siendo $M_{\max}(r)$ el máximo valor de la masa adentro de la totalidad de las cajas y $M_{\min}(r)$ el mínimo valor dentro de las cajas, entonces la relación (58) se reduce a:

$$\frac{M_{\max}(r)}{M_0} \sim \left(\frac{r}{L}\right)^{D_\infty} \quad \text{y} \quad \frac{M_{\min}(r)}{M_0} \sim \left(\frac{r}{L}\right)^{D_{-\infty}} \quad . \tag{59}$$

Con esto la fluctuación en la densidad de masa $\Omega(r)$ es definida al dividir las relaciones (57) y (58)

$$\Omega(r) = \frac{M_{\max}(r)}{M_{\min}(r)} \sim \left(\frac{L}{r}\right)^{D_{-\infty} - D_\infty} \quad . \tag{60}$$

Es importante remarcar que para cualquier r fijo, de acuerdo con la ecuación (60), $\Omega(r) \to \infty$ conforme $L \to \infty$. Sin embargo, una caja con radio r tiene a lo más r^E elementos (siendo E la dimensión euclidiana), de esto se tiene $\Omega(r) < r^E$, lo cual es una contradicción, puesto que r no depende de L. Debido a esta contradicción, encontramos un inconveniente teórico para el uso de la ecuación (57) como base para explorar el comportamiento asintótico en objetos fractales.

Puesto que uno de los objetivos principales de esta tesis es determinar experimentalmente el comportamiento asintótico de la distribución de la masa en agregados DLA, se considera la opción alternativa que proporciona el método de cajas extendidas, el cual no depende de la ecuación (60) y de esta manera evitamos controversias teóricas aun no resueltas.

El método de cajas extendidas presentado por Pastor y Riedi [19] es un proceso que permite la estimación de dimensiones generalizadas D_q para valores positivos y negativos de q. Este método modifica la función de partición como sigue:

$$\chi_q^*(r) = \sum_{\mu(B_i(r)) \neq 0} \mu(B_i^*(r))^q \tag{61}$$

con

$$B_i^*(r) = \prod_{k=1}^{E} [(l_{ik} - 1)r, (l_{ik} + 2)r] \tag{62}$$

Donde l_{ik} es la posición en la rejilla para la i-ésima caja a lo largo de la k-ésima dimensión. La ecuación (61) proporciona una nueva medida para cualquier caja no vacía, la cual es tomada sobre un espacio con un tamaño lineal tres veces más grande que el determinado por la medida original, mediante la adición de las medidas producidas por las cajas vecinas para cada caja.

Con esto podemos aplicar la ecuación (60) a nuestros experimentos para estimar los valores D_q^*. R.H. Riedi ha demostrado que D_q^* converge al mismo limite que D_q cuando $r \to 0$. Luego, el método de cajas extendidas nos da una forma confiable para analizar propiedades

multifractales para un rango de objetos mayor al de la función de partición original, debido a que puede estimar el comportamiento de D_q para valores con $q<0$. Nótese que la ecuación (60) no es necesaria en este método y por tanto no hay conflicto alguno con los argumentos dados por Lam [60].

VI.2.2 Análisis de estructuras fractales grandes.

Como uno de los principales objetivos de esta tesis es el analizar estructuras DLA grandes mediante datos de lagunaridad y multifractalidad. El primer paso fue la construcción de los agregados. Los detalles correspondientes fueron cubiertos en el capítulo anterior. A continuación se describe la parte del análisis.
Para la parte del análisis multifractal consideramos los métodos de cajas de arena (*sand-box*), de cajas extendidas (*extended box*) y conteo de cajas (*box-counting*). Aunque existen algunos detalles teóricos con el enfoque de *sand-box*, este método presenta experimentalmente un buen ajuste para la descripción del escalamiento de los momentos de masa. La referencia [31] presenta un comportamiento multifractal para la distribución de la masa en agregados DLA de tamaño medio. Lo que aquí se pretende es establecer si esta tendencia desvanece cuando el tamaño de los objetos analizados desvanece o se acentúa cuando el tamaño de los objetos analizados aumenta. Para el caso de conteo de cajas solamente consideramos los momentos positivos puesto que no es factible aplicar este método sobre momentos negativos. Como se menciona en la referencia [13], el método de cajas deslizantes puede también ser usado para estimar las dimensiones generalizadas. Específicamente, los momentos en la ecuación (54) se relacionan con D_q de la siguiente manera:

$$Z^{(q)} = \kappa\left(\frac{r}{L}\right)^{(q-1)D_q+E} \quad , \qquad (63)$$

y

$$\Lambda(r) = \lambda\left(\frac{r}{L}\right)^{D_2-E} \quad . \qquad (64)$$

En las ecuaciones (63) y (64) E es la dimensión euclidiana y κ,λ son constantes en las correspondientes leyes exponenciales. La ecuación (64) nos da una forma directa para obtener D_2 (a menudo llamada "dimensión de correlación") a partir de los parámetros de lagunaridad $\Lambda(r)$. Desafortunadamente, el método *gliding-box* hace que los efectos de borde se incrementen debido a que registra más cajas conteniendo valores de masas demasiado bajos con respecto a los otros métodos aquí mencionados, por tanto los valores de los momentos involucrados en la función de partición (54) presentan valores anormalmente grandes cuando $q<0$. En los experimentos de este trabajo encontramos valores de D_q muy similares para todos los métodos empleados cuando $q>0$, de forma que, para simplificar la exposición solamente presentamos estos valores para el algoritmo de *box-counting* y los valores para D_2 obtenidos con los datos de lagunaridad.

Para evaluar la lagunaridad $\Lambda(r)$ en agregados DLA grandes tomamos la misma proporción $r_{\max}/L \approx 0.01$ en cada experimento para el máximo radio de las cajas $r_{\max}$. Se toma esta aproximación porque diferentes agregados con el mismo número de partículas tienen ligeramente diferente tamaño linear debido a la morfología de los objetos creados por el proceso, además, conforme $r \to L$ las áreas centrales son muestreados un número de veces considerablemente mayor que las áreas periféricas. Queda claro que para obtener datos más

confiables acerca de la distribución de masa, la condición $r \ll L$ es necesaria tanto para el análisis de multifractalidad como para el cálculo de lagunaridad. En la Figura 17 se muestra un agregado DLA bidimensional mostrando el tamaño relativo de una caja de radio $r = r_{max}$ con $\frac{r_{max}}{L} = 0.01$. Este tamaño es adecuado para obtener un buen comportamiento estadístico.

Por otra parte, para evaluar la multifractalidad, a diferencia del caso de la estimación de lagunaridad, necesitamos cambiar el radio r y su tamaño relativo con respecto a L con la finalidad de obtener un buen ajuste lineal para la regresión logarítmica al evaluar $\tau(q)$ usando diferentes métodos a diferentes escalas.

Los análisis de la multifractalidad de masa para DLA en 2D con tamaños entre el rango de 10^6 y 10^9 partículas se muestran en la Figura 18. Los datos par DLA en 3D se muestran en la Figura 19.

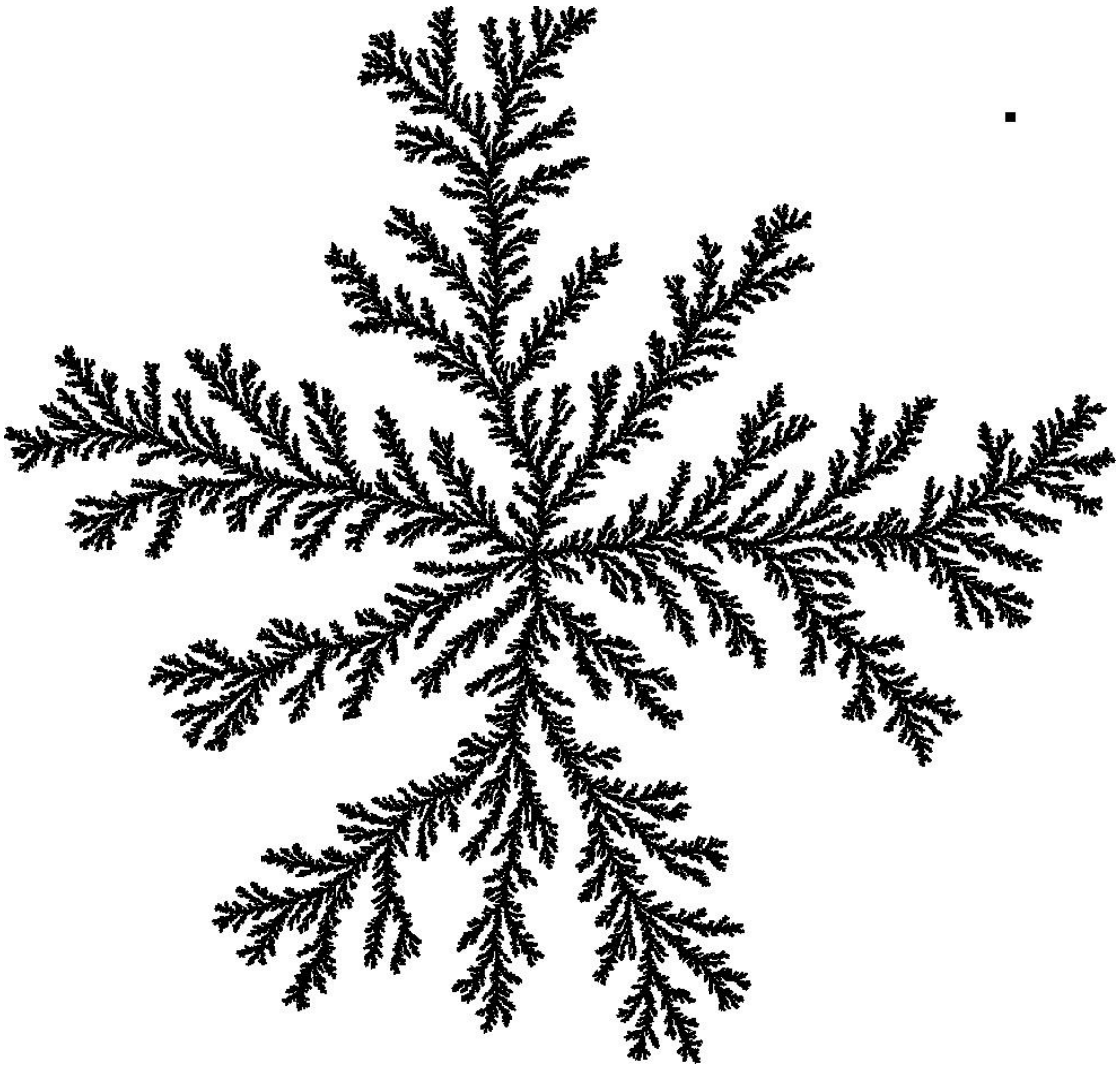

Figura 17. Relación del tamaño máximo de la caja deslizante con respecto al agregado.

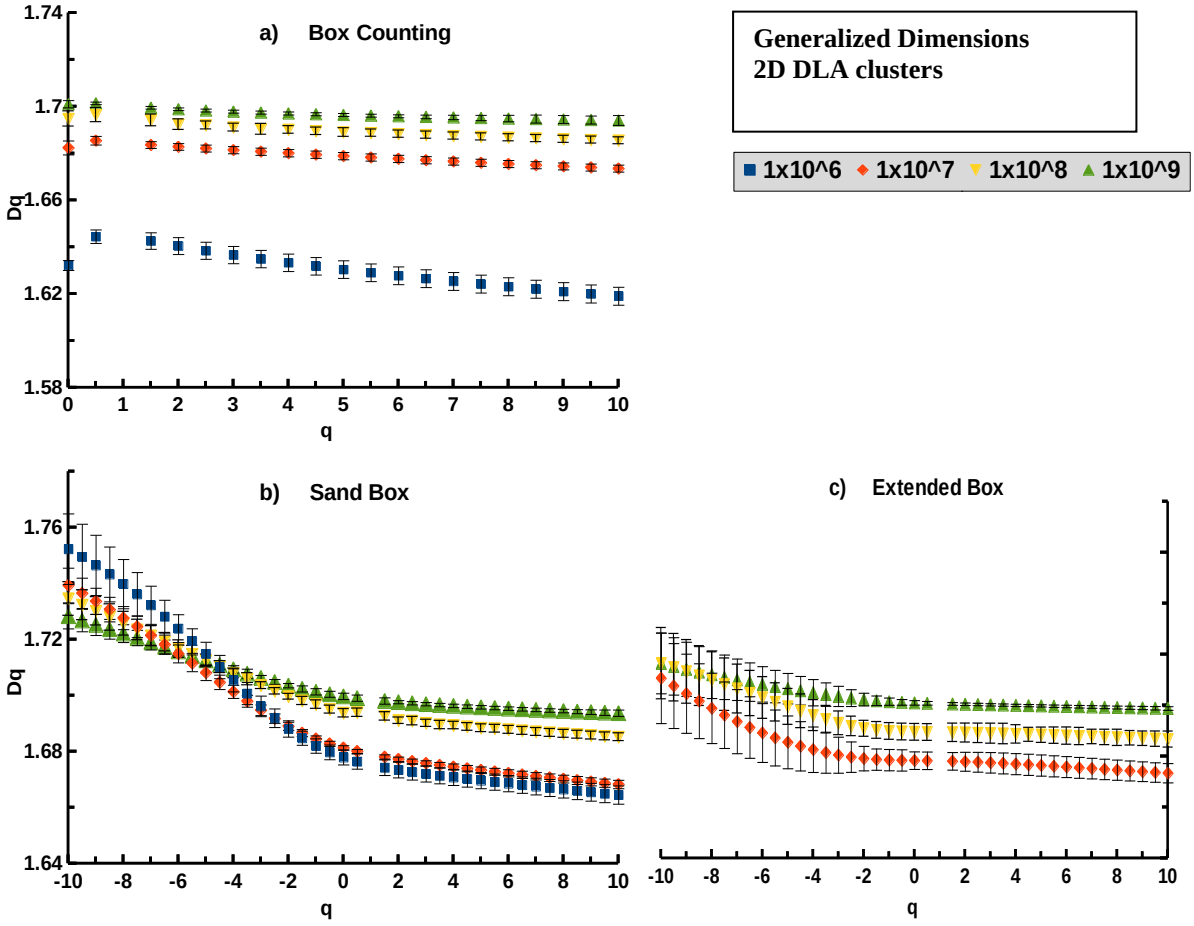

Figura 18. Estimación de las dimensiones generalizadas para agregados DLA utilizando los métodos a) box-counting, b) sand-box and c) extended-box

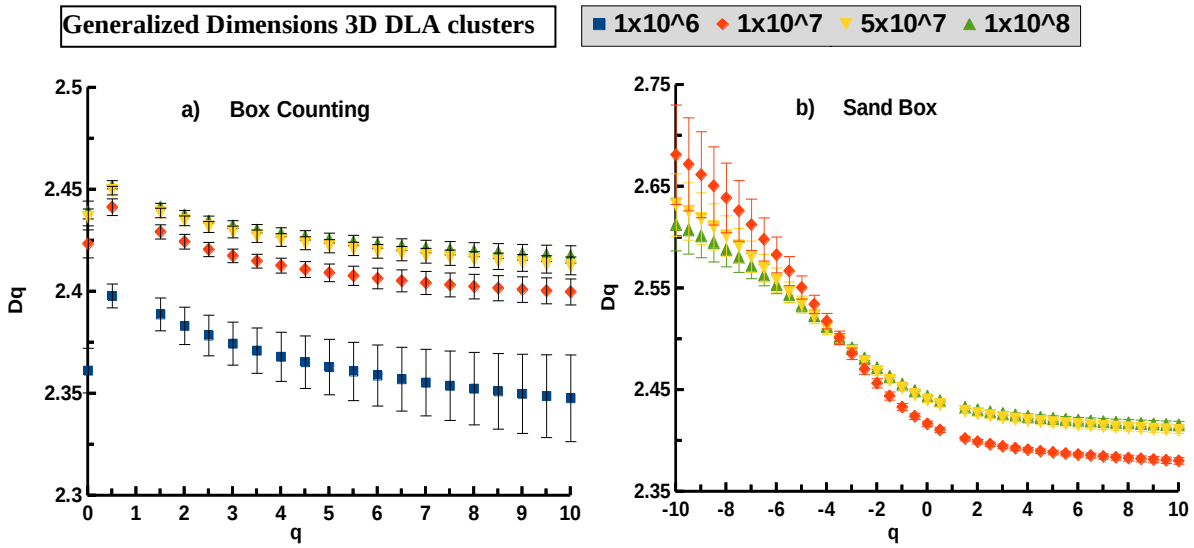

Figura 19. Estimación de las dimensiones generalizadas para agregados DLA en 3D utilizando los métodos a) *box-counting* y b) *sand-box*

La generación de los agregados y el análisis multifractal fueron realizados utilizando la supercomputadora HPC 4000 "Kanbalam". Los agregados se generaron utilizando solamente un procesador mientras que los métodos de *box-counting*, *sand-box* y *extended-box* se ejecutaron en paralelo asignando dinámicamente procesos separados para evaluar datos correspondientes a cada radio. Luego, todos los resultados parciales fueron recolectados para evaluar la regresión

lineal. Este esquema de paralelización dio mejores resultados que la división de dominios tradicional debido al hecho de que una alta dependencia de datos es introducida por el algoritmo descrito anteriormente. El programa entero se corrió usando de 8 a 32 procesadores, según el tamaño de agregado y tomo en promedio 20 horas para analizar un agregado DLA grande. Se realizaron 12 experimentos para cada tamaño de agregado usando 20 diferentes tamaños de radio.

En las Figuras 18 y 19 se puede notar que cuando el tamaño N de los agregados DLA incrementa D_q tiende a forma curvas planas las cuales se aproximan al valor 1.7 para el caso bidimensional y al valor 2.5 para el caso en tres dimensiones (aunque en el caso 3D la convergencia es más lenta). Este hecho es más evidente para momentos positivos. Asumimos que este efecto es debido al hecho de que aun los para los métodos en los que los momentos negativos no presentan gran problemática, la parte positiva del espectro multifractal es evaluada con un mayor exactitud dado que $q > 0$. También es de notar el hecho de que para el caso 3D el método de cajas extendidas no converge hacia una función $f(\alpha)$. Esto es porque el método de cajas extendidas en cada caja toma un espacio de muestra mayor que los otros métodos para un radio fijo r. La convergencia también falla en agregados 2D con 10^6 partículas puesto que la condición $r \ll L$ no queda garantizada para estos casos relativamente pequeños.

Las Figuras 20 y Figura 21 muestran el espectro $f(\alpha)$ para agregados DLA, en estas figuras podemos ver como $f(\alpha)$ tiende a conjuntarse dentro de un rango angosto de valores conforme N incrementa. Para un comportamiento multifractal se esperaría que el espectro $f(\alpha)$ se diferenciara cada vez más para valores grandes de N puesto que una variedad más amplia de patrones aparecería produciendo muestras ampliamente esparcidas conforme $N \to \infty$.

Los resultados del análisis de lagunaridad son ilustrados en la Figura 22. Aquí podemos apreciar como la lagunaridad varia a través de las escalas puesto que depende del radio de las cajas. Se puede apreciar que el comportamiento logarítmico d $\Lambda(r)$ para agregados 2D y 3D corresponde al de un objeto monofractal [13]. Con la única excepción del primer grupo de valores que aparecen en el lado izquierdo de la Figura 22 a) (con valores de D_2 claramente diferentes), no obstante, estos puntos corresponde a estimaciones de $\Lambda(r)$ con valores pequeños de r, cuando r tiene el tamaño de solamente unos pocos diámetros de partícula. Estos valores están obviamente influenciados por los efectos de tamaño finito, de forma que no pueden ser considerados como consecuencia del comportamiento multifractal, tampoco podemos asociarlos con alguna especie de dimensión textural [7]. Este efecto se desvanece conforme $N \to \infty$. La Tabla 2 muestra estimaciones numéricas de D_2 basadas en lagunaridad. En esta tabla se aprecian los valores D_{2f} obtenidos mediante un ajuste lineal tomando los dos primeros radios consecutivos, y los valores D_{2l} obtenidos de los dos últimos radios. D_{2T} representa las estimaciones usando todos los datos disponibles y D_{rg} es la dimensión fractal obtenido mediante la relación de masa $M(r_g) \sim r_g^{\,D}$ tomando r_g como el radio de giro $r_g = \sqrt{(1/N)\sum_{i=1}^{N} < R_i^2 >}$ con R_i siendo el radio de un agregado conteniendo i partículas. Los coeficientes de correlación son mostrados junto con sus desviaciones estándar r_g .

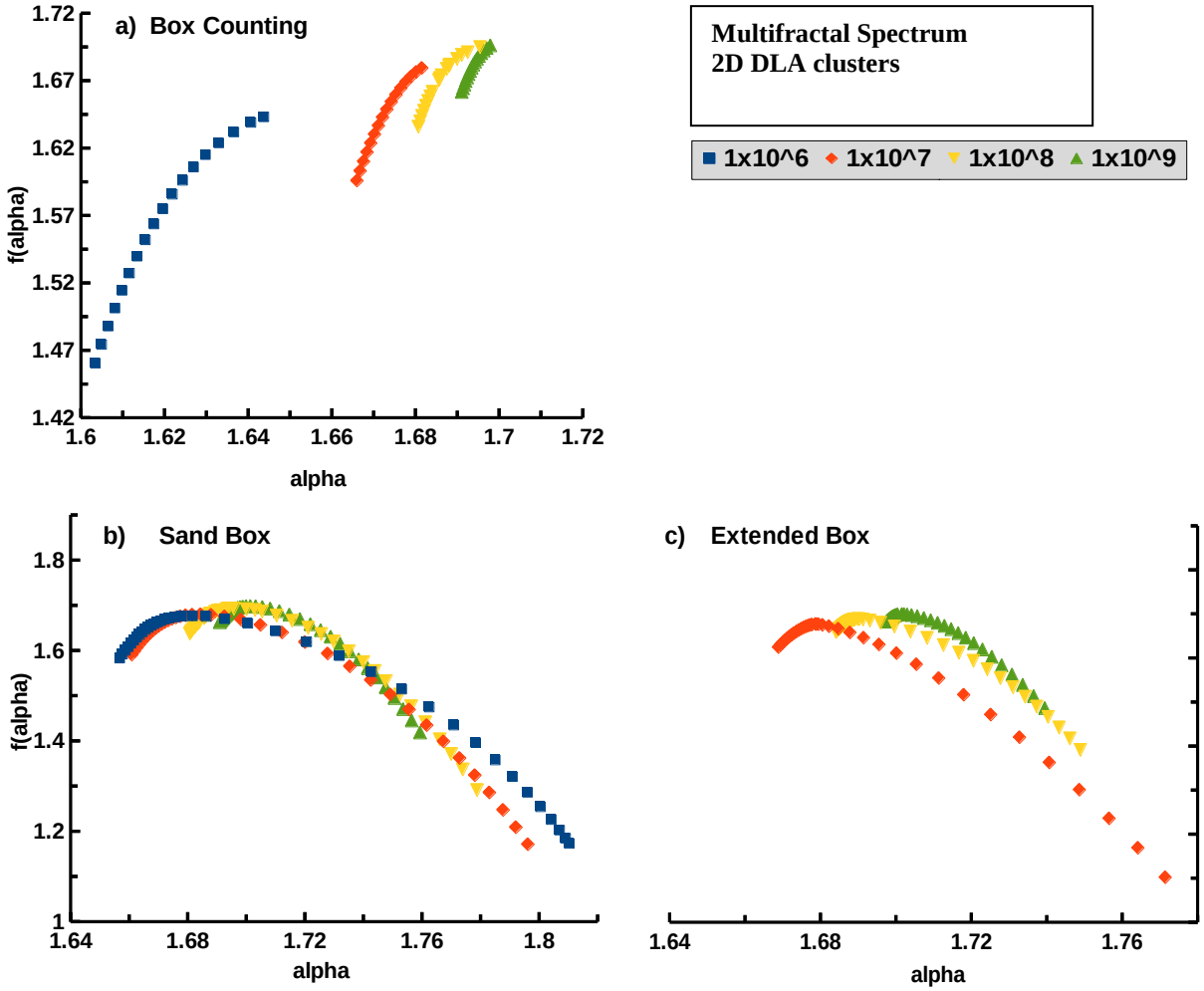

Figura 20. Espectro multifractal para agregados 2D-DLA correspondientes al promedio de las dimensiones generalizadas utilizando los métodos a) *box-counting*, b) *sand-box* y c) *extended-box*.

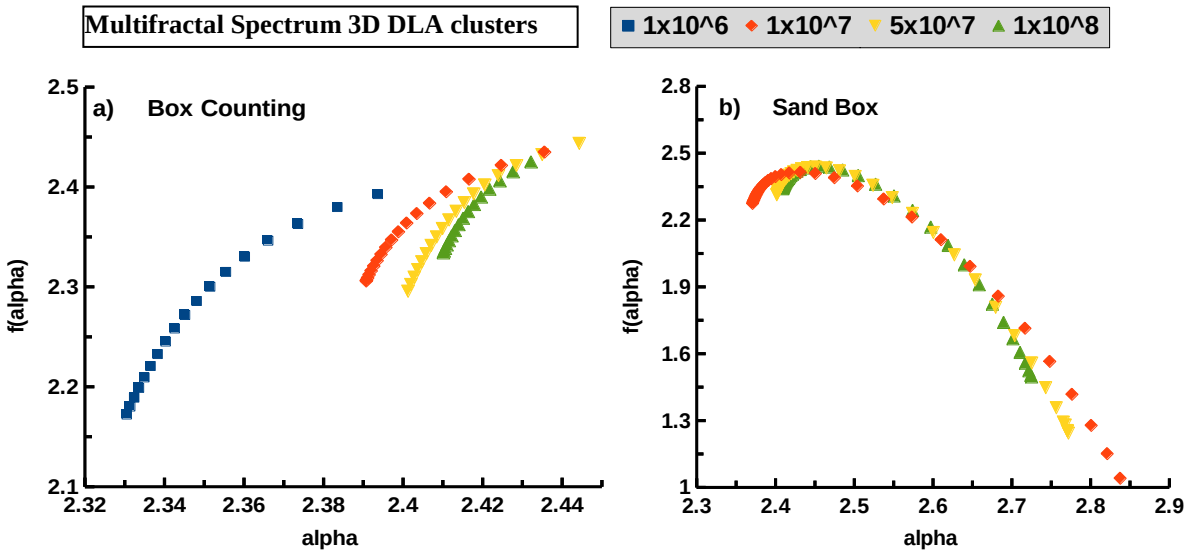

Figura 21. Espectro multifractal para agregados 3D-DLA correspondientes al promedio de las dimensiones generalizadas utilizando a) box-counting, b) sand-box.

Por tanto, lo que se aprecia es que nuestros resultados difieren notablemente de los reportados por los autores en la referencia [13] donde los espacios angulares son usados como una medida de lagunaridad. El hecho de que diferentes valores para la dimensión de correlación no prevalecen sobre diferentes escalas, nos inclina a pensar en un comportamiento monofractal para los agregados DLA. Más aun, conforme N incrementa observamos de la Figura 22 a) y de la Tabla 2 que las regresiones lineales que pueden corresponder a diferentes valores de D_2, estos convergen a un solo valor (con 1.7 como el límite para agregados 2D-DLA), en lugar de diferenciarse cada vez más.

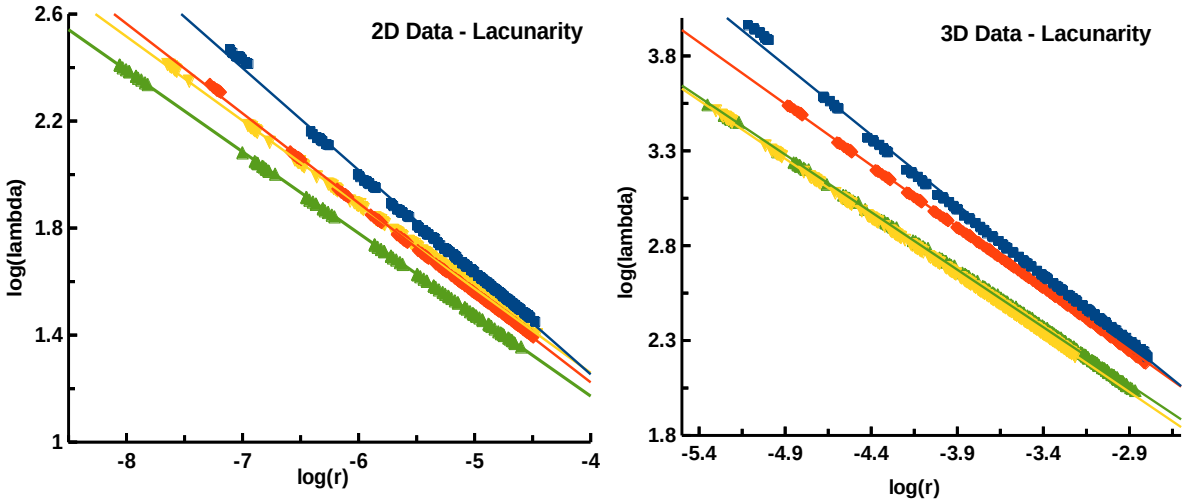

Figura 22. Estimaciones de lagunaridad para 2D-DLA y 3D-DLA. Las pendientes corresponden a los valores de D_{2T} indicados en la Tabla.

2-D DLA								
Size	**D_{2f}**	**R^2**	**D_{2l}**	**R^2**	**D_{2T}**	**R^2**	**D_{rg}**	**σ**
$1x10^6$	1.657562	0.975620	1.654261	0.993144	1.628902	0.997656	1.701978	0.016151
$1x10^7$	1.679688	0.982791	1.668276	0.984282	1.666707	0.999484	1.705274	0.013136
$1x10^8$	1.691146	0.991131	1.674407	0.991724	1.685759	0.999838	1.712187	0.012926
$1x10^9$	1.701819	0.998623	1.692488	0.998459	1.695828	0.999970	1.711226	0.008799

3-D DLA								
Size	**D_{2f}**	**R^2**	**D_{2l}**	**R^2**	**D_{2T}**	**R^2**	**D_{rg}**	**σ**
$1x10^6$	2.312831	0.982009	2.280074	0.871386	2.296164	0.998080	2.484450	0.021187
$1x10^7$	2.383937	0.983411	2.285932	0.942958	2.352101	0.999793	2.506500	0.020897
$5x10^7$	2.407770	0.986882	2.370585	0.956706	2.385214	0.999894	2.511927	0.014032
$1x10^8$	2.475064	0.999911	2.382192	0.999954	2.408482	0.999942	2.511684	0.013864

Tabla 2. Dimensiones de masa-radio y de correlación estimadas utilizando lagunaridad y radio de giro para agregados DLA en 2D y 3D. D_{2f} y D_{2l} fueron evaluados tomando respectivamente los dos primeros y los dos últimos puntos de ajuste para el tamaño de las cajas r en un rango de $r/L \approx 0.0003$ a $\frac{r}{L} \approx 0.01$. D_{2T} toma el total de los tamaños de las cajas y D_{rg} utiliza la relación radio-masa con el radio de giro. R^2 es el coeficiente de regresión y σ son las desviaciones estándar.

El comportamiento de la lagunaridad encontrado en nuestros experimentos conforme $N \to \infty$ es característico de procesos con efectos de tamaño finito. Un comportamiento similar con $D_q \to D_0$ (siendo $D_0 = 1.7$) es lo que pudimos apreciar cuando realizamos el análisis multifractal en nuestros experimentos. El caso 3D mostrado en la figura 22 b) tiene la misma tendencia que el caso 2D. Por tanto, desde un punto de vista experimental, concluimos que la distribución de la masa para agregados DLA grandes presenta un comportamiento monofractal.

VI.3 ANALISIS DE ENTROPIA

VI.3.1 Método para la estimación de entropía configuracional basado en separación de clases usando la distancia de Hamming.

En esta sección se describe una nueva metodología para la estimación de la entropía configuracional en un agregado basándose en el concepto de entropía dentro del contexto de la teoría de la información, introducida por Shannon [16]. La entropía queda entonces definida para un conjunto $A = \{a_i : i = 1, .. n\}$ de n secuencias de bits y esta expresada por

$$H(A) = -\sum_{i=1}^{n} p_i log_2(p_i) \qquad (65)$$

donde p_i es una probabilidad asociada cada elemento a_i.

En nuestro desarrollo consideramos al espacio que contiene a un agregado como una secuencia de bits, en la cual el valor de cada bit se establece como $'1'$ si la posición correspondiente se encuentra ocupada, y $'0'$ para una posición vacía. Nótese que en este esquema, un agregado bidimensonal contenido en un área de tamaño lineal l quedara representado por una secuencia de bits de longitud $l \times l$. Además tomamos a $p_i$como la probabilidad de que la evolución del modelo de crecimiento, una vez establecidos sus parámetros iniciales, dé como resultado la configuración a_i. El tipo de modelo tratado en esta investigación contiene elementos estocásticos, de tal forma que no se conocen de manera determinista los valores de p_i. Por tanto, estas cantidades deben ser determinadas de manera empírica, en nuestro caso concreto, a través de simulación computacional. Un aspecto a considerar bajo este enfoque es la dificultad practica para calcular tal valor dada la gran cantidad de configuraciones posibles para un espacio de tamaño lineal l, puesto que el número total de configuraciones posibles es de $2^{l \times l}$, cada probabilidad p_i es muy baja, por lo que aun con ayuda de computo de alto rendimiento, no es posible medir directamente este valor a partir de un numero de ocurrencias de a_i, pues a una escala práctica, es difícil que a_i ocurra más de una vez.

Por ejemplo, bajo un enfoque sencillo para la determinación de p_i, se procedería a generar una gran cantidad de agregados, y se registrarían las frecuencias en que aparece cada configuración. Desafortunadamente, en la práctica, aun contando con decenas de miles de repeticiones del experimento que genera a los agregados, es poco probable que una configuración se presente en más de una ocasión, lo que llevaría a una determinación errónea y no representativa de los valores p_i.

Para solucionar este problema, se propone que, en lugar de calcular la probabilidad p_i para cada configuración, se considere una colección de clases de estados $\Xi = \{\xi_\varkappa : \varkappa = 1, \dots, k\}$ en los cuales puede encontrarse un agregado generado por un modelo bajo ciertos parámetros, tales conjuntos estarán determinados mediante una relación de equivalencia. Las clases de equivalencia quedarán establecidas por el grado de similaridad que se presente entre en las configuraciones a_i. Existen muchas formas de establecer la similaridad entre diferentes objetos

geométricos [63]. De acuerdo al objetivo de este trabajo, optamos por considerar una medida de similaridad basada en la entropía misma, pero con respecto a una relación entre estados individuales, generados a partir de un modelo, de tal forma que sea posible definir un conjunto de clases de equivalencia de estados, para después poder determinar la probabilidad de que una configuración aleatoria pertenezca a cierta clase. Basándonos en esto, podemos determinar la entropía total del modelo de crecimiento bajo ciertos parámetros de entrada usando la ecuación (1). El procedimiento que utilizamos es el denominado *Entropy-based Fuzzy Clustering* de Yao, et al [15]. Este método de clasificación puede ser descrito de la siguiente manera:

a) Iniciar con $k = 1,\ \mathrm{X} = A$, $\xi_\varkappa = \emptyset\ \forall \varkappa$,

b) Calcular la entropía E_i para cada $x_i \in \mathrm{X}$ mediante:

$$E_i = -\sum_{j=1, j\neq i}^{\aleph}(S_{ij}log_2 S_{ij} + (1 - S_{ij})log_2(1 - S_{ij})) \qquad (66)$$

donde $S_{ij} \in [0,1]$ es la similaridad entre los elementos x_i y x_j.

c) Seleccionar el elemento $x_{i_{min}}$ con mínima entropía.

d) Para cada x_β tal que $d\left(x_{i_{min}}, x_\beta\right) < \beta$, hacer $\mathrm{X} = \mathrm{X}\backslash\{x_\beta\}$, $\xi_k = \xi_k \cup \{x_\beta\}$.

e) Si $X \neq \emptyset$, hacer $k = k + 1$ y regresar al paso c).

Con esto se consigue tener una colección de clases $\Xi = \{\xi_\varkappa\}$ cuyos elementos más representativos (o centros con respecto a la métrica usada) fueron seleccionados automáticamente de acuerdo a su entropía relativa con respecto al resto de los elementos.

La función de similaridad S_{ij} entre los elementos x_i y x_j la tomamos de la siguiente manera:

$$S_{ij} = e^{\alpha\frac{H(i,j)}{N}} \qquad (67)$$

Donde $H(i,j)$ es la distancia de Hamming entre x_i y x_j. N denota al número de partículas de los agregados a los cuales representan x_i y x_j, es decir, el número de posiciones en el espacio ocupadas. El parámetro $\alpha = 0.5$ es un elemento de normalización. Este último valor se toma considerando que $H(i,j)$ es a lo más $2N$.

Una vez que obtenemos nuestras clases de equivalencia Ξ, procedemos a calcular su entropía

$$H(\Xi) = -\sum_{i=1}^{\varkappa} \rho_i log_2(\rho_i) \qquad (68)$$

Donde $\rho_i = \frac{|\xi_i|}{\sum_k |\xi_k|}$ con $|\xi_i|$ denotando al número de elementos de la clase ξ_i.

Es importante notar que $H(\Xi) \to H(A)$ conforme $\beta \to 1$, lo que nos permite estimar $H(A)$ aun cuando la capacidad de simulación de los agregados queda relativamente restringida a una escala práctica.

Esta técnica puede ser utilizada para determinar la entropía configuracional de varios tipos de conjuntos n-dimensionales, ya que no se encuentra restringida a los agregados fractales, pues el procedimiento general no depende de restricciones específicas de las cadenas de bits ni de la métrica de la distancia de Hamming ocupada aquí.

VI.3.2 Implementación del cálculo de entropía.

La implementación de la simulación del modelo, así como de los cálculos necesarios para el análisis de los mismos, se realizó mediante la utilización de GPGPU con la interfaz CUDA [29]. Tanto la etapa de crecimiento fractal como la determinación de los valores de las matrices de similaridad S_i, y la reducción de los mismos mediante las ecuaciones (67) y (68) fueron calculadas enteramente usando procesamiento con la GPU (GTX 460). Esto dio como resultado un incremento considerable, del orden de 40x, en la aceleración de las simulaciones y cálculos con respecto a la utilización de un solo CPU (*core duo* 2.4 GHz). El código de las partes esenciales del algoritmo puede observase en el Apéndice 3.

Este algoritmo fue aplicado para extraer el comportamiento de la entropía para el modelo de simulación hibrido descrito en la referencia [51] y comentado en la sección de simulación de modelos en un capitulo anterior. Los resultados obtenidos al aplicar el procedimiento sobre 1000 agregados para cada umbral de potencial de crecimiento. En este cálculo más demandante es la obtención de la matriz de similaridad requerida en la ecuación (66).

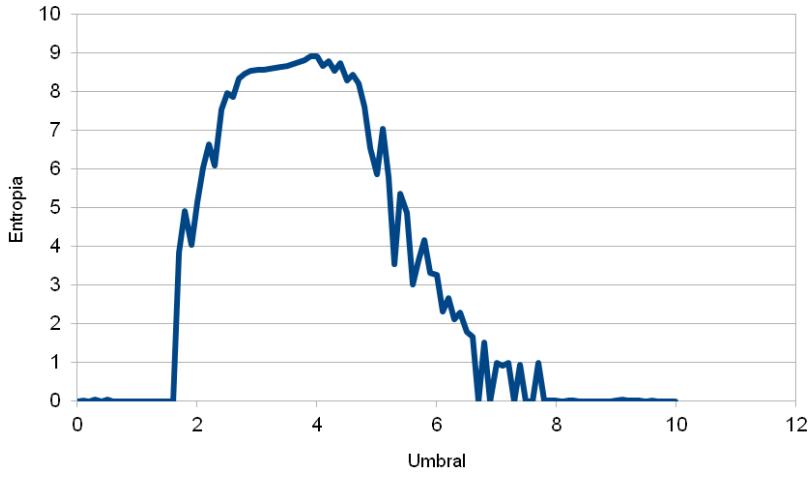

Figura 23. Calculo de entropía configuracional para un conjunto de agregados utilizando diferentes umbrales de crecimiento.

Una característica importante que se aprecia en la Figura 23 es la asimetría de la gráfica que nos indica que el comportamiento del modelo cambia para los umbrales correspondientes. Esta asimetría sugiere que ocurre un 'cambio de fase' [51] en el modelo.

VII. CONCLUSIONES.

La investigación y desarrollo de programas de cómputo intensivo presentadas en este trabajo para ejecutar los modelos de crecimiento fractal en diversas plataformas de hardware de cómputo paralelo proporcionaron una herramienta eficiente para el análisis y la generación de estos.

En cuanto a la generación de agregados fractales se obtuvieron los agregados más grandes publicados a la fecha con hasta 10^{10} partículas para el modelo PDLA y 10^{9} partículas en el modelo DLA. Estos agregados fueron creados utilizando la técnica de *Búsqueda de Precisión Variable en Quadtrees* y el hardware proporcionado por la supercomputadora "*Kanbalam*" para los agregados de 10^{10} partículas y un servidor de alto rendimiento con 8 núcleos de procesamiento para los de 10^{9} partículas y menores. Los tamaños anteriormente publicados en la literatura especializada habían sido de 10^{8} partículas para ambos modelos [12][41][61].

Similarmente se reprodujeron los modelos de percolación, y crecimiento laplaciano utilizando una GPU con capacidades de cómputo de propósito general con 336 *cores(compute shaders)*. Estos modelos fueron combinados con un procedimiento basado en agentes para crear un modelo hibrido de crecimiento fractal basado en la publicación de la referencia [51], a los cuales se les aplicó el análisis de entropía aquí descrito.

El procedimiento para evaluar la lagunaridad de un conjunto presentado aquí es una mejora con respecto a otras implementaciones de la técnica gliding-box [13] que ya habían sido usadas en otros trabajos. Se mostró que el incremento en la velocidad de ejecución obtenido puede ser de gran ayuda para realizar análisis de lagunaridad para diversos proyectos de investigación. Con la ganancia en velocidad que se consiguió se puede concluir que la implementación aquí descrita es bastante adecuada para el análisis en simulaciones de modelos como pueden ser percolación, rompimiento dieléctrico, difusión limitada por agregación y modelos híbridos. En particular, el tamaño de los objetos que se pueden manejar con el código descrito aquí permite que el análisis de lagunaridad pueda aplicarse a agregados bastante grandes que se necesitan para obtener límites asintóticos con una precisión adecuada. Este desempeño incluso abre la posibilidad de usar esta técnica como una alternativa de clasificación en tiempo real, por ejemplo para un sistema de reconocimiento automático con capacidad de discernir características fractales utilizando lagunaridad.

Se presentaron los resultados que se observaron mediante la aplicación de tres métodos de análisis multifractal (*box-counting*, *sliding-box* y *sand-box*) sobre la estructuraras de agregados DLA de gran tamaño y también se realizó análisis de lagunaridad basado en el método *gliding-box*. El concepto de medida tomado para el análisis multifractal fue la masa de los agregados, de tal forma que lo que se exploró con este análisis fue su distribución geométrica.

Lo que se encontró con esto es que el espectro de multifractalidad de masa para las estructuras DLA se reduce a un rango cada vez más estrecho de valores conforme el número de partículas aumenta. Esta tendencia es independiente del procedimiento utilizado para ejecutar el análisis. Aquí se concluye que el hecho de que se produzcan grandes variaciones para las dimensiones generalizadas D_q en los momentos negativos es debido a que estas son mucho más sensibles a efectos de tamaño finito cuando $q < 0$. De los experimentos aquí mostrados, queda claro que la representación limitada de números de punto flotante (tanto doble como simple) contribuye a las variaciones en los valores de dimensión estimados, especialmente para momentos con valor absoluto grande.

Desde la perspectiva del análisis de lagunaridad aquí realizado se obtiene el mismo resultado. Concretamente se observa la tendencia hacia una única dimensión fractal conforme el tamaño de los agregados incrementa. Se concluye entonces que para los agregados DLA no existe una distribución de masa multifractal, a pesar de que la distribución en la probabilidad de crecimiento en su superficie si lo es [58]. Nuevamente el comportamiento observado en escalas intermedias es atribuido a efectos de tamaño finito. Estos efectos tienden a desaparecer conforme el número de partículas en los agregados incrementa. De acuerdo con los resultados aquí obtenidos se puede deducir que los diferentes valores de dimensión fractal reportados anteriormente para estructuras DLA son muy probablemente debidos a la diferencia de velocidad de convergencia de los diferentes métodos conforme se alcanza un valor aceptable para la aproximación $N \to \infty$.

Adicionalmente a lo anterior, se propuso en este trabajo un nuevo método para determinar la entropía en la distribución de estructuras geométricas en el espacio. Este método está basado en la clasificación difusa y la distancia de Hamming. La implementación del procedimiento se realizó utilizando GPGPUS y se aplicó sobre estructuras generadas por el método de la referencia [51]. Lo que se obtiene al determinar la entropía para diferentes umbrales de crecimiento es que existe la posibilidad de un cambio de fase en el modelo indicado por una inflexión en la velocidad de cambio de la curva de entropía.

Como conclusión general se resume las aportaciones en el aspecto computacional del presente trabajo:

- Algoritmo de búsqueda de precisión variable sobre *quadtrees* y sus generalizaciones multidimensionales incluyendo *octrees y kd-trees*. [30]. Este algoritmo es un método general de búsqueda por lo que puede adaptarse a diferentes problemas de computo que requiera realizar una búsqueda espacial sobre estructuras de datos multidimensionales. La eficiencia de tal algoritmo sobre otros tipos de búsqueda queda evidenciada de manera práctica en el desarrollo de esta tesis.

- Se logró determinar la el comportamiento de la multifractalidad de masa para agregados DLA mediante una implementación eficiente [64] de la versión multidimensional método *gliding box* [14] haciendo uso de técnicas y equipo de cómputo científico intensivo.

- Creación del algoritmo para la determinación de la entropía en conjuntos espaciales basada en el concepto de entropía de Shannon y el método de *Fuzzy C-Means*.
  Este algoritmo también es de aplicación general y puede aplicarse a problemas generales que requieran realizar una caracterización sobre distintos conjuntos espaciales.

- Se utilizó exitosamente el procesamiento en paralelo híbrido GPU – CPU en la simulación y análisis de diversos modelos de crecimiento fractal.

VIII. REFERENCIAS.

[1] Mandelbrot B.B. *The fractal geometry of nature*. W.H. Freeman and Company. 1982.

[2] Witten T. A. Sander L. M. Diffusion-Limited Aggregation, A Kinetic Critical Phenomenon. *Physical Review Letters*. 47-19 1400. 1981.

[3] Eden, M.A two dimensional growth process, in *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability*, 1961.

[4] Niemeyer L., L. Pietronero, and H. J. Wiesmann, Fractal dimension of dielectric breakdown, *Physical Review Letters*, vol. 52, pp. 1033–1036, 1984.

[5] Hastings M. and L. Levitov, Laplacian growth as one-dimensional turbulence, *Physica D, 116*, pp. 244–252, 1998.

[6] Voss R., Muitiparticle Fractal Aggregation. Journal of Statistical Physics, VoL 36, Nos. 5/6, 1984.

[7] Michael Batty. Cities and Complexity. Understanding Cities with Cellular Automata, Agent-Based Models, and Fractals, MIT Press. 2005.

[8] Evertsz C.J.G., Mandelbrot B.B. Multifractal measures. En: Peitgen H.-O., Jürgens H., Saupe D.. *Chaos and Fractals.* Springer-Verlag. 1992.

[9] Halsey T. C., P. Meakin, I. Procaccia, Scaling structure of the surface layer of diffusion-limited aggregates, Physical Review Letters 56, 86-857, 1986.

[10] Bentley J.L. Multidimensional Binary Search Trees Used for Associative Searching. Comunications of the ACM 18(9). 1975.

[11] Finkel R.A., Bentley J.L. Quad trees: a Data Structure for Retrieval on Composite Keys. Acta Informatica. 4(1) 1974.

[12] Kaufman H., A. Vespignani, B. Mandelbrot, L. Woog, Parallel diffusion-limited aggregation, Physical Review E. 52, 5602-5609, 1995.

[13] Allain C., Cloitre M. Characterizing the lacunarity of random and deterministic fractal sets. Physical Review A. Vol. 44 No. 6. 3552-58 . 1991.

[14] Tolle R.C., T.R. McJunkin, D.J. Gorsich, An efficient implementation of the gliding box lacunarity algorithm, Physica D 237, 306-315, 2008.

[15] Yao J., M. Dash, S.T. Tan, H. Liu. Entropy-based fuzzy clustering and fuzzy modeling. Fuzzy Sets and Systems 113, pp. 381-388 .2000.

[16] Shannon. C. E. A Mathematical Theory of Communication. The Bell System Technical Journal, Vol. 27, pp. 379–423, 623–656, 1948.

[17] Halsey T.C., Jensen, M.H., Kadanoff L. P. Procaccia I., Shraiman B.I. Fractal measures and their singularities: The caracterization of strange sets. Physical Review A. Vol. 33 No. 2. 1141. 1986.

[18] Tél T., Vicsek T. Geometrical multifractality of growing structures. Journal of Physics A. Vol. 20. 1987.

[19] Pastor-Satorras R., R.H. Riedi, Numerical estimates of the generalized dimensions of the Hénon attractor for negative q, Journal of Physics A 29, L391-L398, 1996.

[20] Gefen Y., Y. Meir, B.B. Mandelbrot, A. Aharony. Geometric implementation of hypercubic lattices with noninteger dimensionality using low lacunarity fractal lattices. Physical Review Letters, 50-3, 145-148 .1983.

[21] Falconer Kenneth. Fractal Geometry. Mathematical Foundations and Applications. Second Edition. Ed. Wiley and Sons Ltd. 2003.

[22] Barnsley Michael. Fractals Everywhere. Academic Press Inc. 1988.

[23] Peitgen H.-O., Jürgens H., Saupe D. Chaos and Fractals. New Frontiers of Science. Second Edition. Springer. 2004.

[24] Vicsek T. Fractal Growth Phenomena. Second Edition. World Scientific. 1992.

[25] Plotnick R.E., R.H. Gardner, W.W. Hargove, K. Prestegaard, M. Perimutter. Lacunarity analysis: A general technique for the analysis of spatial patterns. Physical Review E. Vol. 53 No. 5. 5461-68 . 1996.

[26] Butenhof D.R., Programming with POSIX Threads Addison-Wesley Professional 1997.

[27] The Open Group Base Specifications Issue 6, IEEE Std 1003.1. 2004.

[28] Pacheco P., Parallel Programming with MPI , Morgan Kaufmann. 1996.

[29] CUDA C Programming Guide, Nvidia Corporation. 2010.

[30] Sosa-Herrera A. Rodríguez-Romo S. Variable precision distance search for random fractal cluster simulation. WSEAS Transactions in Computers Vol. 8 Issue 8. 2009.

[31] Vicsek T., Family F., Meakin P. Multifractal Geometry of Diffusion-Limited Aggregates. Europhysics Letters 12 (3). 1990.

[32] Williams T., Bjerknes R., Stochastic model for abnormal clone spread through epithelial basal layer, Nature, 236, 19. 1972.

[33] Meakin P. Fractals, Scaling and Growth Far from Equilibrium. Cambridge Nonlinear Science Series. 1998.

[34] Hermann H.G. Geometrical Cluster Growth Models and Kinetic Gelation. Physics Reports 136, 3. 1986.

[35] Bunde A., Havlin S. (Eds.) Fractals and Disordered Systems. Springer. 1991.

[36] Cormen T. H., C. E. Leiserson Rivest R. L., C. Stein, Introduction to Algorithms The MIT Press third edition edition, 2009.

[37] Samet H. Foundations of Multidimensional and Metric Data Structures. The Morgan Kaufmann Series in Computer Graphics, 2006.

[38] Sosa-Herrera A. Rodríguez-Romo S. Paralelización de Algoritmos para el Estudio de Interacción de agregados. Tesis UNAM. 2008.

[39] Ball R.C., Brady R.M. Large-Scale Lattice Effect In Diffusion-Limited Aggregation. Journal Of Physics A-18 (13): L809, 1985.

[40] Tolman S. & P. Meakin, Off-lattice and hypercubic lattice models for diffusion-limited aggregation in dimensionalities 2-8, Physical Review A 40, 428-437, 1989.

[41] Yu. A Menshutin, L. N. Shchur, V. M. Vinokur, Probing surface characteristics of diffusion-limited aggregation clusters with particles of variable size, Physical Review E. Rapid Communications 75, 010401, 2007.

[42] *http://gcc.gnu.org/, Free Software Foundation Inc, 2011.*

[43] http://www.kernel.org/ Linux Kernel Organization, Inc, 2011.

[44] Hennessy J. L., Patterson D.A., Computer Architecture, Fifth Edition: A Quantitative Approach. The Morgan Kaufmann Series in Computer Architecture and Design. 2011.

[45] Amdahl, Gene. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. AFIPS Conference Proceedings (30): 483–485. 1967.

[46] Wolfram S. A New Kind of Science. Wolfram Media, 2002.

[47] Herlihy M., Shavit N., The Art of Multiprocessor Programming. Morgan Kaufman, 2008.

[48] Niazi, M., & Hussain, A. Agent-based computing from multi-agent systems to agent-based models: a visual survey. Scientometrics, 1-21. 2011.

[49] Mattson T. G., Sanders B. A., Massingill B. L., Patterns for Parallel Programming. Addison-Wesley Professional; 1 edition, 2004.

[50] Intel Guide for Developing Multithreaded Applications. Intel Corporation 2011.

[51] R. Murcio, S. Rodriguez-Romo. Modeling large Mexican urban metropolitan areas by a Vicsek Szalay approach. Physica A, Volume 390, 16, 2011.

[52] Plotnick, R., G. Gardner, and R. V. O'Neill. Lacunarity indices as measures of landscape texture. Landscape Ecology, 8:201-211.1993.

[53] Gefen Y., Meir Y., Aharony A. Geometric implementation of hypercubic lattices with noninteger dimensionality by use of low lacunarity fractal lattices. Physical Review Letters. Vol. 50 No. 3 ,1983.

[54] Lin b., Yang Z. R. A suggested lacunarity expression for Sierpinski carpets. Journal of Physics A. Vol. 19 L49. 1986.

[55] Blumenfeld R., Mandelbrot B. B. Lévy dust, Mittag-Leffler statistics, mass fractal lacunarity, and perceived dimension. Physical Review E. Vol. 56 No. 1. 1997.

[56] Filho MB, Sobreira F. Accuracy of Lacunarity Algorithms in Texture Classification of High Spatial Resolution Images from Urban Areas. The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences. XXXVII (Part B3b). 2008.

[57] H.E. Stanley, Fractals and multifractals: The interplay of physics and geometry. In Fractals and Disordered Systems. Armin B. Shlomo H. (Editors) 2nd rev. Springer, 1995.

[58] Halsey T. C. Diffusion-Limited Aggregation: A Model for Pattern Formation. Physics Today. 2000.

[59] Y. Gefen, Y. Meir, B.B. Mandelbrot, A. Aharony, Geometric implementation of hypercubic lattices with noninteger dimensionality using low lacunarity fractal lattices, Physical Review Letters 50, 145-148, 1983.

[60] Lam C., Finite-size effects in diffusion-limited aggregation, Physical Review E 52, 2841-2847, 1995.

[61] Mandelbrot B.B., B. Kol, A. Aharony, Angular gaps in radial diffusion-limited aggregation: two fractal dimensions and nontransient deviations from linear self-similarity, Physical Review Letters 88, 05550, 2002.

[62] H.G.E. Hentschel, I. Procaccia, The infinite number of generalized dimensions of fractals and strange attractors. Physica D 8, 435-444. 1983.

[63] Rui Xu, Donald C. Wuncsh II. Clustering. IEEE Press. Wiley, 1996.

[64] Rodriguez-Romo Suemi, Sosa-Herrera Antonio. Lacunarity and Multifractal Analysis of the Large DLA Mass Distribution. Physica A. DOI: 10.1016/j.physa.2013.03.044. 2013.

IX. APENDICES.

APENDICE 1. Código función DLA para procesamiento multihilo con VPDIST

```
pthread_mutex_t mutex1;
/*mutex para bloqueo de dla->datos*/
pthread_mutex_t mutex2;
/*mutex para bloqueo de nodos*/
pthread_mutex_t mutex3;
/*mutex para bloqueo de listas*/


void* DLA(void * arg){
  APDLA miDla;
  PUNTO p1;
  DATRADG *dt;

  struct drand48_data edoRnd;
  //estado de la generacion iterativa
  //para drand48_r()
  double dRnd;
  //variable para guardar el numero aleatorio
  //de drand48_r()

  /*Inicializar generador de numeros
   pseudoaleatorios reentrant*/
  srand48_r(time(NULL), &edoRnd);

  INT64 x,y;
  /*coordenadas de la particula actualmente
  difundiendose*/
  INT64 r;
  /*radio de la particula con respecto al origen*/
  double angmov;
  /* el angulo aleatorio de movimiento */
  INT64 distpaso;     /*distancia del salto*/
  INT64 ddMinIni=XYMAX;
  /*distancia minima inicial de busqueda*/
  INT64 const CINCOTOL=5LL*TOL;
  INT64 const DIEZTOL=10LL*TOL;
  INT64 PUNTONUEVETOL=0.9*TOL;
  const double DOSPI=
    2.0*3.14159265358979323846264643;

  miDla=(APDLA) arg;

  /* para cada punto que se quiera,
  realizar el movimiento browniano hasta
  que se pegue al cluster */
  while(miDla->i <= miDla->numPts){

    /* seleccionar un punto en el circulo
    de nacimiento */
    drand48_r(&edoRnd, &dRnd);
```

```
    angmov = DOSPI*dRnd;
    x = (INT64)(miDla->radionac * cos(angmov));
    y = (INT64)(miDla->radionac * sin(angmov));
    r = sqrt(x*x+y*y);

    /* mientras el punto no se vaya demasiado lejos*/

    while(r <= miDla->radiofin){

      /* obtener la distancia de la particula
      al cluster*/
      if (r > miDla->radionac+DIEZTOL)
        distpaso= r-miDla->radionac-CINCOTOL;
      else{
        /*buscar en la estructura*/
        ddMinIni=miDla->radiofin*miDla->radiofin;

/***Aqui se llama a VPDIST***/
        distpaso= sqrt(VPDIST(
          miDla->raiz,&x,&y,&ddMinIni,
          miDla->distCrit2,
          NIVPROFND,0LL,0LL,(XYMAX-XYMIN)>>2));

      }
      /* si la particula esta muy cerca,
      que se pegue*/
      if(distpaso <= TOL){
        p1.x=x; p1.y=y; p1.sig=NULL;
        agregaParticula(miDla->raiz,&p1);
        /* checar si el nuevo punto
        esta muy lejos*/
pthread_mutex_lock(&mutex1);
        if(r >= miDla->radioext){
          miDla->radioext = r;
        /*con espacio para mover a
        las nuevas particulas*/
          miDla->radionac = miDla->radioext + DISTRNAC;
          miDla->radiofin = miDla->radionac*FACTFIN;
        }
        /*indicar que una nueva particula
        se ha pegado*/
        (miDla->i)++;
        /*guardar datos para calculo de
        radio de giro*/
        miDla->sumRi2act+=(double)(r*r);
        if (!(miDla->i%SALTORAD)) {
          dt=(DATRADG*)malloc(sizeof(DATRADG));
          dt->n=miDla->i;
          dt->sumRi2=miDla->sumRi2act;
          dt->sig=miDla->dtRad;
          miDla->dtRad=dt;
        }
pthread_mutex_unlock(&mutex1);
        break;  /* salir del ciclo*/
      }
      /* si la particula no esta muy cerca,
```

```
        que se mueva */
        else{
          /*obtener el angulo del
          siguiente movimiento*/
          drand48_r(&edoRnd, &dRnd);
          angmov =DOSPI*dRnd;
          /*El maximo al que puede acercase una
          particula al cluster*/
          distpaso = distpaso - PUNTONUEVETOL;
          x += (INT64)(distpaso*cos(angmov));
          y += (INT64)(distpaso*sin(angmov));
          r = sqrt(x*x+y*y);
        }
      }
    }
    return NULL;
}

int main(int argc, char *argv[]){

    int edo;
    /*estado de las operaciones pthreads*/

    APDLA miDla;
    /*Apuntador a la estructura DLA*/

    pthread_attr_t atribHilos;
    /*variable de atributos para los hilos*/
    pthread_t hilos[NUMHILOS];
    /*Apuntador a los objetos hilo*/
    /*parametro de calendarizacion de hilos*/
    struct sched_param mi_param;
    /*mascara de determinacion
    de conjunto de CPUs*/
    cpu_set_t mask;

    int k;
    /*contador para ciclos internos*/
    time_t tFinal;
    /*contador de tiempo*/

    /* Determinar el numero de procesadores */
    int NUMPROCS = sysconf(_SC_NPROCESSORS_CONF);

    printf("\nN = %i procesador(es) detectado(s).\n",
                NUMPROCS);
    printf("Ejecutando n = %i hilo(s).\n",NUMHILOS);
    /*Contar el tiempo de ejecucion*/
    time_t tInicial = time(NULL);

    /*obtener el primer argumento, si existe,
    como sufijo para los nombres de los archivos
     generados*/
    if (argc>1) SUFIJO=argv[1];

    //crea estructura dla con semilla en (0,0)
```

```
  miDla=creaDla(0LL,0LL);

  /*inicializar parametros de creacion de hilos*/

  edo=pthread_attr_init(&atribHilos);
  if(edo)fprintf(stderr,
    " attrib init err:%i %s ",
    edo,strerror(edo));
  /*inicializar parametro de prioridad c
  on prioridad maxima*/
  mi_param.sched_priority=
    sched_get_priority_max(SCHED_FIFO);
  /*permitir establecer esquemas dinamicamente*/
  edo=pthread_attr_setinheritsched(
    &atribHilos,PTHREAD_EXPLICIT_SCHED);
  if(edo)fprintf(stderr,
    " inherit err:%i %s ",
    edo,strerror(edo));
  /*competir por recursos a nivel
  sistema operativo  */
  edo=pthread_attr_setscope(
    &atribHilos,PTHREAD_SCOPE_SYSTEM);
  if(edo)fprintf(stderr,
    " scope err:%i %s ",edo,strerror(edo));
pthread_attr_setdetachstate(&
    atribHilos, PTHREAD_CREATE_JOINABLE);

  /*inicializar los mutex*/
  edo=pthread_mutex_init(&mutex1,NULL);
  if(edo)fprintf(stderr,
    " mutex init err:%i %s ",edo,strerror(edo));
  edo=pthread_mutex_init(&mutex2,NULL);
  if(edo)fprintf(stderr,
    " mutex2 init err:%i %s ",edo,strerror(edo));
  edo=pthread_mutex_init(&mutex3,NULL);
  if(edo)fprintf(stderr,
    " mutex3 init err:%i %s ",edo,strerror(edo));

  /*generar los primeros puntos con algoritmo
  lineal en un solo hilo*/
  if (NUMPTSLIN<NUMPTS){
    miDla->numPts=NUMPTSLIN-1;
    movBrown(miDla);
  }
  /*ahora ejecutar el algoritmo en
  forma paralela con varios hilos*/
  miDla->numPts=NUMPTS;

  /*crear los hilos*/
  for (k=0;k<NUMHILOS;k++){
    edo=pthread_create(&hilos[k],
      &atribHilos,&movBrown,miDla);
    edo=pthread_create(&hilos[k],
      &atribHilos,&movBrown,miDla);
    if(edo)fprintf(stderr,
      " create err:%i %s ",edo,strerror(edo));
```

```
    /* Establecer afinidad de los hilos
    sobre los procesadores */
    CPU_ZERO(&mask);
    CPU_SET(k%(NUMPROCS),&mask);
    edo=pthread_setaffinity_np(hilos[k],
      sizeof(mask),&mask);
    if(edo)fprintf(stderr,
      " affin err:%i %s ",edo,strerror(edo));
  }

  /*esperar la terminacion de los hilos*/
  for (k=0;k<NUMHILOS;k++){
    edo=pthread_join(hilos[k],NULL);
    if(edo)fprintf(stderr,
      " join err:%i %s ",edo,strerror(edo));
  }

  /*registrar el tiempo de ejecucion*/
  tFinal = time(NULL);
  printf("Particulas=  %lu\n",
    NUMPTS);
  printf("tiempo=  %li s\n",
    (long) difftime(tFinal, tInicial));
  printf("Diametro de particulas (TOL)=  %li\n",
     TOL);
  printf("Radio exterior       Re=  %lli\n",
     miDla->radioext);
  printf("Radio de nacimiento  Rn=  %lli\n",
     miDla->radionac);
  printf("Radio de muerte      Rm=  %lli\n",
     miDla->radiofin);
  printf("Radio de giro        Rg=  %g\n",
        sqrt(miDla->dtRad->sumRi2/miDla->dtRad->n));
  printf("Dimension Fractal   DRg=  %g\n",
        dimFractalRG(miDla));

  //termina el hilo principal
  /*destruye objetos de sincronizacion*/
  pthread_attr_destroy(&atribHilos);
  pthread_mutex_destroy(&mutex1);
  pthread_mutex_destroy(&mutex2);
  pthread_mutex_destroy(&mutex3);

  return 0;
}

/*Funcion para crear e inicializar un punto*/
APPUNTO creaPunto(INT64 const* x,INT64 const* y){
  APPUNTO p;
  p=(APPUNTO)malloc(sizeof (PUNTO));
  p->x=*x; p->y=*y;
  p->sig=NULL;
  return p;
}
```

```
/*Funcion para crear e inicializar un
bloque de puntos*/
APPUNTOBLK creaPuntoBlk(){
  APPUNTOBLK p;
  int i;
  p=dlaMallocPUNTOBLK();
  for (i=0;i<TAMBLKPTS;i++){
    p->Xs[i]=(char)(-128);
    p->Ys[i]=(char)(-128);
  }
  p->sig=NULL;
  return p;
}
/*Sin apuntador sig*/
APPUNTOBLKNS creaPuntoBlkNs(){
  APPUNTOBLKNS p;
  int i;
  p=dlaMallocPUNTOBLKNS();
  for (i=0;i<TAMBLKPTS;i++){
    p->Xs[i]=(char)(-128);
    p->Ys[i]=(char)(-128);
  }
  p->Ys[TAMBLKPTS-1]=(char)0;
  return p;
}

/*Funcion para crear e inicializar un nodo*/
APNODO creaNodo(){
  APNODO p;
  p=dlaMallocNODO();
  p->NE=NULL;
  p->NO=NULL;
  p->SE=NULL;
  p->SO=NULL;
  return p;
}
/*Funcion para crear e inicializar un quadtree*/
APNODO creaArbol(){
  APNODO arb;
  arb=creaNodo();
  return arb;
}

/*Funcion para crear e inicializar un dla*/
APDLA creaDla(INT64 sx, INT64 sy){
  APDLA miDla;
  PUNTO p1;
  /*crear estructura para DLA e inicializarla*/
  miDla=(APDLA)malloc(sizeof(DLA));
  /*calcula distancias cubiertas por
  los nodos del quadtree en el dla*/
  calcCoberturas(miDla);
  /*inicia el quadtree para guardar
  los puntos del dla*/
  (miDla->raiz)=creaArbol();
  /*crea primer punto semilla y
```

```
  guardala en el arbol*/
  (miDla->i)=1UL;
  p1.x=sx; p1.y=sy;p1.sig=NULL;
  agregaParticula(miDla->raiz,&p1);
  (miDla->i)++;
  /*establecer el radio exterior*/
  miDla->radioext=0LL;
  /*empezar con un radio de nacimiento
  cercano pero no tanto*/
  miDla->radionac = DISTRNACINI;
  /*establecer el radio de muerte*/
  miDla->radiofin=miDla->radionac*FACTFIN;
  /*inicializar datos de calculo de
  radio de giro*/
  miDla->dtRad=NULL;
  miDla->sumRi2act=0.0;
  /*inicializar datos de calculo de gaps*/
  miDla->lstAroPuntos=NULL;
  miDla->lstGaps=NULL;
  return miDla;
}

//calcular el cuadrado de la diagonal
//desde el centro hasta un extremo
//de el area que cubren los nodos
//del arbol en cada nivel
//dstCX,dstCY son el ancho y la altura
//que cubren los nodos
void calcCoberturas(APDLA miDla){
  int k;
  miDla->distCrit2=(INT64*)malloc(
    sizeof(INT64)*(NIVPROFND+1));
  miDla->distCrit=(INT64*)malloc(
    sizeof(INT64)*(NIVPROFND+1));
  miDla->distCXY=(INT64*)malloc(
    sizeof(INT64)*(NIVPROFND+2));
  *(miDla->distCXY+NIVPROFND+1)=(XYMAX-XYMIN)<<1;
  //distancias empiezan a contar desde
  //el ultimo nivel=0 hacia arriba
  for (k=NIVPROFND; k>=0;k--){
    *(miDla->distCXY+k)=(*(miDla->distCXY+k+1))>>1;
    *((miDla->distCrit2)+k)=
      (*(miDla->distCXY+k)**(miDla->distCXY+k))<<1;
    *((miDla->distCrit)+k)=
      (INT64)sqrt((double)*((miDla->distCrit2)+k));
  }
  return;
}

/*Funcion para agregar una
nueva particula al QUAD-Tree. */
void agregaParticula(APNODO raiz, APPUNTO p){
  APNODO q,padre;
  void *Q;
  //hoja donde se insertara el dato
  enum Sector sect=NE;
```

```
  int niv;
  INT64 valX,valY,desp;
  if (raiz){
    niv=NIVPROFND;
    q=raiz;
    padre=NULL;
    Q=NULL;

    valX=0LL;
    valY=0LL;
    desp=(XYMAX-XYMIN)>>1;

    while (niv>0){
      padre=q;
      niv--;
      desp=desp>>1;
/*bloqueo de escritura exclusiva*/
pthread_mutex_lock(&mutex2);
//determinar las coordenadas de division
      if (p->x>=valX){
        if (p->y>=valY){
          q=padre->NE;
          valX+=desp;
          valY+=desp;
          if (!niv){
            Q=padre->NE;
            sect=NE;
          }
          if (!q){
            //si no existe el hijo, crearlo
            //puede ser hoja o nodo
            if (niv){
              q=creaNodo();
              padre->NE=q;
            }else{
              Q=creaPuntoBlkNs();
              padre->NE=Q;
            }
          }
        }
        else{
          valX+=desp;
          valY-=desp;
          q=padre->SE;
          if (!niv){
            Q=padre->SE;
            sect=SE;
          }
          if (!q){
            //si no existe el hijo, crearlo
            //puede ser hoja o nodo
            if (niv){
              q=creaNodo();
              padre->SE=q;
            }else{
              Q=creaPuntoBlkNs();
```

```
                padre->SE=Q;
              }
            }
          }
        }else{
          if (p->y>=valY){
            valX-=desp;
            valY+=desp;
            q=padre->NO;
            if (!niv){
              Q=padre->NO;
              sect=NO;
            }
            if (!q){
              //si no existe el hijo, crearlo
              //puede ser hoja o nodo
              if (niv){
                q=creaNodo();
                padre->NO=q;
              }else{
                Q=creaPuntoBlkNs();
                padre->NO=Q;
              }
            }
          }else{
            valX-=desp;
            valY-=desp;
            q=padre->SO;
            if (!niv){
              Q=padre->SO;
              sect=SO;
            }
            if (!q){
              //si no existe el hijo, crearlo
              //puede ser hoja o nodo
              if (niv){
                q=creaNodo();
                padre->SO=q;
              }else{
                Q=creaPuntoBlkNs();
                padre->SO=Q;
              }
            }
          }
        }
/*desbloquear escritura exclusiva*/
pthread_mutex_unlock(&mutex2);
    }
    //en este punto Q es la hoja
    //donde se debe insertar p
    //agregar los datos de p en Q;
/*bloqueo de escritura exclusiva*/
pthread_mutex_lock(&mutex3);
    agregaDato((void *)Q,p,valX,valY,padre,sect);
/*desbloquear escritura exclusiva*/
pthread_mutex_unlock(&mutex3);
```

```
  }
  return;
}

//Agrega un dato a la lista de bloques en
//una hoja el bloque disponible, si lo hay siempre
//se encuentra a la cabeza de la lista, por lo que
//si este esta lleno hay que agregar un nuevo bloque
//y no checar el resto
void agregaDato(void * hoja,
  APPUNTO p, INT64 valX,INT64 valY,
  APNODO ant, enum Sector sect){

  APPUNTOBLK apblk;
  INT64 crX,crY;
  int cont;
  //convertir a coordenadas relativas
  crX=p->x-valX;
  crY=p->y-valY;
  //ajustar las coordenadas a la resolucion
  if (crX<-127LL) crX=-127LL;
  if (crX>127LL) crX=127LL;
  if (crY<-127LL) crY=-127LL;
  if (crY>127LL) crY=127LL;

  if (hoja){
    //insertar el dato en el siguiente
    //espacio disponible los espacios disponibles
    //estan al inicio de la lista
    apblk=(APPUNTOBLK)hoja;
    for (cont=0; cont<TAMBLKPTS;cont++){
      if(apblk->Xs[cont]==(char)-128 &&
        apblk->Ys[cont]==(char)-128){
        //espacio disponible indicado
        //por char -128
        apblk->Xs[cont]=(char)crX;
        apblk ->Ys[cont]= (char)crY;
        //salir de la funcion
        return;
      }
    }
    //hasta este punto no se encontro espacio
    //crear un nuevo bloque y agregarle el dato
    apblk=creaPuntoBlk();
    apblk->Xs[0]=(char)crX;
    apblk->Ys[0]= (char)crY;
    apblk->sig=hoja;
    //y ponerlo al inicio de la lista
    switch (sect){
      case NE:
        ant->NE=(void*)apblk;
        break;
      case NO:
        ant->NO=(void*)apblk;
        break;
      case SE:
```

```
            ant->SE=(void*)apblk;
            break;
          case SO:
            ant->SO=(void*)apblk;
            break;
          default:;
        }
    }
  return;
}
```

APENDICE 2. Código función VPDIST

```
INT64 VPDIST(void * nodo, INT64 const* x,INT64 const* y,
    INT64 *minddist, INT64 const *dstCrit2,
    int miNiv, INT64 miX, INT64 miY,INT64 desp)
{
  APPUNTOBLK ap;
  INT64 ddst;

  INT64 hX,hY;
  int cont;

  if (nodo){
    /*Si son los ultimos niveles buscar aproximadamente
    pero solo si la distancia relativa es suficientemente
    grande (distancia cubierta por un nivel anterior
     en el quadtree)*/
    if (miNiv<NIVBUSSUP){
      //busca la distancia al centro del nodo;
      ddst=(*x-miX)*(*x-miX)+(*y-miY)*(*y-miY);
      if (ddst>dstCrit2[miNiv+1]){
        //si la distancia es menor que la minima
        // distancia encontrada hasta ahora
        //ddst=((INT64)sqrt((double)ddst))-dstCrit[miNiv];
        //hX y hY como auxiliares
        hX=(absol(*x-miX)-(desp<<1));
        hY=(absol(*y-miY)-(desp<<1));
        ddst =hX*hX+hY*hY;
        if (ddst<*minddist) *minddist=ddst;
        return *minddist;
      }
    }
    /*Si es una hoja, buscar punto por punto exactamente*/
    if(miNiv==0){
      ap=(APPUNTOBLK)nodo;
      while(ap){
        cont=0;
        //mientras el dato no sea char -128
        while(ap->Xs[cont]!=(char)-128 && cont<TAMBLKPTS){
          //convertir las coordenadas relativas
          //en absolutas
          hX=miX+(INT64)(ap->Xs[cont]);
          hY=miY+(INT64)(ap->Ys[cont]);
          //calcular distancia ortogonal
          ddst=(*x-hX)*(*x-hX)+(*y-hY)*(*y-hY);
          /*actualizar la distancia mas corta,
           segun sea el caso*/
          if (ddst<*minddist) *minddist=ddst;
          cont++;
        }
        if(ap->Xs[TAMBLKPTS-1]==(char)-128 &&
          ap->Ys[TAMBLKPTS-1]==(char)0){
          //indica fin de lista
          ap=NULL;
        }else{
          //siguiente elemento
```

```
      ap=ap->sig;
    }
  }
}else{
/*De otra forma, descender un nivel en la busqueda*/
  /*si el punto esta al este, busca primero ahi*/
  if (*x>=miX){
    /*si el punto esta al norte, busca primero ahi*/
    if (*y>=miY){
      /*como hay puntos posiblemente mas
      cercanos al noreste, buscarlos*/

      *minddist=VPDIST(((APNODO)nodo)->NE,x,y,minddist,
        dstCrit2,miNiv-1,miX+desp,miY+desp,desp>>1);
      /*si hay puntos posiblemente mas cercanos en otros
       cuadrantes, buscarlos*/
      if((*x-miX)*(*x-miX)<=*minddist){
        *minddist=VPDIST(((APNODO)nodo)->NO,x,y,minddist,
          dstCrit2,miNiv-1,miX-desp,miY+desp,desp>>1);
      }
      if((*y-miY)*(*y-miY)<=*minddist){
        *minddist=VPDIST(((APNODO)nodo)->SE,x,y,minddist,
          dstCrit2,miNiv-1,miX+desp,miY-desp,desp>>1);
      }
      if( (*x-miX)*(*x-miX)+
        (*y-miY)*(*y-miY)<=*minddist ){
          *minddist=VPDIST(((APNODO)nodo)->SO,x,y,minddist,
            dstCrit2,miNiv-1,miX-desp,miY-desp,desp>>1);
      }
    }
    /*si el punto esta al sur, busca primero ahi*/
    else{
      /*como hay puntos posiblemente mas cercanos al
       sureste, buscarlos*/
      *minddist=VPDIST(((APNODO)nodo)->SE,x,y,minddist,
        dstCrit2,miNiv-1,miX+desp,miY-desp,desp>>1);
      /*si hay puntos posiblemente mas cercanos
      en otros cuadrantes, buscarlos*/
      if((*x-miX)*(*x-miX)<=*minddist){
        *minddist=VPDIST(((APNODO)nodo)->SO,x,y,minddist,
          dstCrit2,miNiv-1,miX-desp,miY-desp,desp>>1);
      }
      if((*y-miY)*(*y-miY)<=*minddist){
        *minddist=VPDIST(((APNODO)nodo)->NE,x,y,minddist,
          dstCrit2,miNiv-1,miX+desp,miY+desp,desp>>1);
      }
      if( (*x-miX)*(*x-miX)+
        (*y-miY)*(*y-miY)<=*minddist ){
          *minddist=VPDIST(((APNODO)nodo)->NO,x,y,minddist,
            dstCrit2,miNiv-1,miX-desp,miY+desp,desp>>1);
      }
    }
  }
  /*si el punto esta al oeste, busca primero ahi*/
  else{
    /*si el punto esta al norte, busca primero ahi*/
```

```
        if (*y>=miY){
          /*como hay puntos posiblemente mas cercanos al
          noreste, buscarlos*/
          *minddist=VPDIST(((APNODO)nodo)->NO,x,y,minddist,
            dstCrit2,miNiv-1,miX-desp,miY+desp,desp>>1);
          /*si hay puntos posiblemente mas cercanos en otros
          cuadrantes, buscarlos*/
          if((*x-miX)*(*x-miX)<=*minddist){
            *minddist=VPDIST(((APNODO)nodo)->NE,x,y,minddist,
              dstCrit2,miNiv-1,miX+desp,miY+desp,desp>>1);
          }
          if((*y-miY)*(*y-miY)<=*minddist){
            *minddist=VPDIST(((APNODO)nodo)->SO,x,y,minddist,
              dstCrit2,miNiv-1,miX-desp,miY-desp,desp>>1);
          }
          if((*x-miX)*(*x-miX)+
            (*y-miY)*(*y-miY)<=*minddist ){
              *minddist=VPDIST(((APNODO)nodo)->SE,x,y,minddist,
                dstCrit2,miNiv-1,miX+desp,miY-desp,desp>>1);
          }
        }
        /*si el punto esta al sur, busca primero ahi*/
        else{
          /*como hay puntos posiblemente mas cercanos al
          sureste, buscarlos*/
          *minddist=VPDIST(((APNODO)nodo)->SO,x,y,minddist,
            dstCrit2,miNiv-1,miX-desp,miY-desp,desp>>1);
          /*si hay puntos posiblemente mas cercanos en otros
          cuadrantes, buscarlos*/
          if((*x-miX)*(*x-miX)<=*minddist){
            *minddist=VPDIST(((APNODO)nodo)->SE,x,y,minddist,
              dstCrit2,miNiv-1,miX+desp,miY-desp,desp>>1);
          }
          if((*y-miY)*(*y-miY)<=*minddist){
            *minddist=VPDIST(((APNODO)nodo)->NO,x,y,minddist,
              dstCrit2,miNiv-1,miX-desp,miY+desp,desp>>1);
          }
          if( (*x-miX)*(*x-miX)+
            (*y-miY)*(*y-miY)<=*minddist ){
              *minddist=VPDIST(((APNODO)nodo)->NE,x,y,minddist,
                dstCrit2,miNiv-1,miX+desp,miY+desp,desp>>1);
          }
        }
      }
    }
  }
  return *minddist;
}
```

APENDICE 3. Código función calculo de entropia

```
public:
  funcSimil(T m):max(m){}
     __host__ __device__
     T operator()(const T &h) const {
         return (T)( 1 - h/(2*max)) ;
     }
};

//funcion que calcula la sumatoria de P^q con
//los datos P y q almacenados en un vector
template <class T1, class T2>
//__host__ __device__
inline T1 sumPq ( const thrust::device_vector<T1> & P,
        const T2 & q ){

  thrust::device_vector<T1> v(P.size(), 0);

  powFunc<T1,T2>  functorBinario;

  thrust::transform(P.begin(),P.end(),q.begin(),
    v.begin(), functorBinario);

  return thrust::reduce(v.begin(),v.end());

};

// funcion que calcula la sumatoria de P*ln(P) con los
//datos P y q almacenados en un vector
template <class T>
//__host__ __device__
inline T sumPlnP ( const thrust::device_vector<T> & P){

  multXlogX<T>    functorUnario;
  thrust::plus<T>   functorBinario;
  return thrust::transform_reduce(P.begin(),P.end(),
    functorUnario,(T)0,functorBinario);
};

// funcion que calcula la sumatoria de P*log2(P)
// con los datos P y q almacenados en un vector

template <class T>
//__host__ __device__
inline T sumPlog2P (
  const thrust::device_vector<T> & P){

  multXlog2X<T>   functorUnario;
  thrust::plus<T>   functorBinario;
  return thrust::transform_reduce(P.begin(),P.end(),
    functorUnario,(T)0,functorBinario);
};

//funcion calcula la distancia hamming entre vectores
```

```
template <class T1,class T2>
T2 distHamming ( const thrust::device_vector<T1> & a,
        const thrust::device_vector<T1> & b ){

  thrust::device_vector<T2> c(a.size(), (T2)0);
  notXor<T1>  functorBinario;
  thrust::transform(a.begin(),a.end(),b.begin(),
    c.begin(), functorBinario);
  return thrust::reduce(c.begin(),c.end());

};

//Funcion que calcula la matriz de proximidad H
// para un conjunto de N vectores dentro de un
//arreglo AV[]. Utiliza la distancia hamming H(i,j)
template <class T1, class T2>
void calcProximidad (
  const thrust::device_vector<T1> AV[],
  thrust::device_vector<T2> & H, const unsigned & N){

  //inicializa h con ceros
  thrust::fill(H.begin(),H.end(),(T2)0);
  //ciclo para matriz de proximidad
  //H es simetrica con ceros en la diagonal
  for (int i=0; i<N; i++){
    for (int j=i+1; j<N; j++){
      H[i*N+j] = (T2) distHamming <T1,T2>(AV[i],AV[j]);
      H[j*N+i] = H[i*N+j];
    }
  }

}

//Funcion que calcula la matriz de similaridad S para
//un conjunto de N vectores dentro de un arreglo AV[].
//Utiliza la matriz de proximidad H(i,j)
//para calcular S(i,j) 1 - H(i,j)/hMax donde hMax
//es la distancia maxima en H

template <class T1, class T2>
void calcSimilaridad (
  const thrust::device_vector<T1> AV[],
  thrust::device_vector<T2> & S, const unsigned & N,
  int nparts ){

  thrust::device_vector<unsigned> H(N*N);

  //llamada a calculo de matriz de proximidad
  calcProximidad<T1,unsigned> (AV,H,N);

  //maximo elemento de H convertido al tipo de
  //datos de S
  //T2 hMax =(T2) H[indMax];
  //inicializa S con ceros
  thrust::fill(S.begin(),S.end(),(T2)0);
  //realizar la operacion  1 - h/nparts
```

```
  thrust::transform(H.begin(),H.end(),S.begin(),
    funcSimil<T2>(nparts));
}

//funcion que calcula la entropia Ei
//para cada vector i
//de un conjunto con N vectores basandose en su
//matriz de similaridad S.
template <class T>
void calcEntropiaI (
  const thrust::device_vector<T> S,
  thrust::device_vector<T> & E,const unsigned & N){

  for (unsigned i = 0; i < N; i++){
    E[i] = -1.0 * thrust::transform_reduce(
      S.begin() + N*i,
      S.begin() + N*(i+1),
      funcEntropiaI<T>(),
      (T)0.0, thrust::plus<T>());
  }
}

struct comparar
  {
    __host__ __device__
    bool operator()(double lhs, double rhs)
    {
      return lhs < rhs;
    }
  };

// Funcion que ejecuta el algoritmo
// Entropy Fuzzy Clustering
// para un conjunto de N vectores dentro de
// un arreglo AV[].
// Cada elemento CL[i] contendra el numero
// de elementos en la clase i
// numCls es el numero de clases encontradas
// El parametro beta es el radio de
//similaridad entre clases
template <class T1,class T2>
void entropyFuzzyClustering (
  const thrust::device_vector<T1> AV[],
  thrust::device_vector<T2> & CL, unsigned &numCls,
  const unsigned & N,
  const double & beta, int nparts){

  thrust::device_vector<double> E(N);
  //entropia para cada vector i
  thrust::device_vector<double> S(N*N);
  //matriz de similaridad

  numCls=0;

  //llamada a calculo de matriz de similaridad
  calcSimilaridad<T1,double>(AV,S,N,nparts);
```

```
  //llamada a calculo de entropia
  calcEntropiaI<double>(S,E,N);

  //inicializa CL con ceros
  thrust::fill(CL.begin(),CL.end(),(T2)0);

  double minEnt = 0.0;
  while (numCls < N) {

    unsigned indMin=0;
    for(unsigned i=1; i<N; i++){
      if (E[i] < E[indMin]){
        indMin=i;
      }
    }
    minEnt= E[indMin];
    if (minEnt >= N) break;
    //para cada vector j en el conjunto
    for(unsigned j=0; j<N; j++){
      //si la similaridad con un vector
      //no marcado es mayor que beta
      if (S[indMin*N+j] > beta && E[j] < N ){
        //marcar al elemento con valor
        //maximo de entropia N
        E[j]=N;
        //incrementar cuenta clase actual
        CL[numCls]++;
      }
    }

    // evaluar siguiente clase
    numCls++;
  }
}

//Funcion para la generacion y analisis del proceso
//de crecimiento fractal
int crecimiento( ) {

  BlockDatos datos;
  //bloque de datos agrupados

  dim3 blocks(DIM/16,DIM/16);
  dim3 threads(16,16);

  GPUAnimBitmap bitmap( DIM, DIM, &datos );

  CUDA_CALL( cudaEventCreate( &datos.inicio ) );
  CUDA_CALL( cudaEventCreate( &datos.fin ) );

  CUDA_CALL( cudaMalloc( (void**)&datos.dev_ArrMask,
                         DIM*DIM*sizeof(double)));
  CUDA_CALL( cudaMalloc( (void**)&datos.dev_ArrAAM,
                         DIM*DIM*sizeof(double)));
  CUDA_CALL( cudaMalloc( (void**)&datos.dev_ArrBBM,
```

```
                                DIM*DIM*sizeof(double)));
  CUDA_CALL( cudaMalloc( (void**)&datos.dev_ArrMaskInt,
                                DIM*DIM*sizeof(int)) );
  CUDA_CALL( cudaMalloc( (void**)&datos.dev_ArrMaskChar,
                                DIM*DIM*sizeof(char)) );
  CUDA_CALL( cudaMalloc( (void**)&datos.dev_edoRndBF,
                                sizeof(curandState)
                                *TAM_BF_RAND) );

  CUDA_CALL( cudaMalloc( (void**)&datos.dev_ArrAgnts,
                                DIM*DIM*sizeof(double)));

  CUDA_CALL( cudaMalloc( (void**)&datos.dev_ArrReduc,
                                TAM_BF_RED*sizeof(int)));

  CUDA_CALL( cudaEventRecord( datos.inicio, 0 ) );

  double ArrMsk[DIM*DIM];

//CODIGO PARA CALCULO DE ENTROPIA //
  char nombArchEnt[80] = "datos\\Entropia.txt\0";
    //nombre  archivo de entropia
  ofstream ofsE(nombArchEnt);
  ofsE.precision(12);
    //imprime encabezado de lista de datos
  ofsE << "Modelo crecimiento urbano and agentes"
    << endl;
  ofsE << "T= Reps= U= Ent= beta= clases= probs= "
    << endl;

  cudaEvent_t iniU, finU;

  CUDA_CALL( cudaEventCreate( &iniU ) );
  CUDA_CALL( cudaEventCreate( &finU ) );

  //Dado T fijo calcula repeticiones
  //para cada umbral
  for (datos.umbral=UMBRAL_RMIN;
      datos.umbral <= UMBRAL_RMAX;
      datos.umbral+=INC_UMBRAL ){

  CUDA_CALL( cudaEventRecord( iniU, 0 ) );

//CODIGO PARA CALCULO DE ENTROPIA//

// arreglo que guarda resultado de
// cada repeticion como vector
thrust::device_vector<char> d_AV[REPETIR];

printf("\niters:\n");
for (int iRep = 0; iRep < REPETIR; iRep++){
//inicia repetir

      //genera experimento correspondiente
      generaIter(&datos);
```

```
      CUDA_CALL( cudaDeviceSynchronize() );
      //Traslada la mascara del dispositivo
      //a la motherboard
      CUDA_CALL( cudaMemcpy(
        ArrMsk,datos.dev_ArrMask,
        DIM*DIM*sizeof(double),
        cudaMemcpyDeviceToHost));
//fin repetir
}

//CODIGO PARA CALCULO DE ENTROPIA //
//convierte los datos de la mascara a tipo char
arregloDouble2CharValKernel2D<<<blocks,threads>>>
      ( datos.dev_ArrMaskChar, datos.dev_ArrMask );
thrust::device_ptr<char> dev_ptr(
  datos.dev_ArrMaskChar);
//guarda estos datos como un vector
d_AV[iRep].resize(DIM*DIM);
thrust::copy(dev_ptr, dev_ptr + DIM*DIM,
        d_AV[iRep].begin());
// ahora con los datos obtenidos calcula
// la entropia de
// todas las repeticiones del experimento

unsigned numClases=0;
//  numero de clases que aparecieron

// vector con el numero de elementos de cada clase
thrust::device_vector<int> d_CL(REPETIR,0);
// algoritmo de clustering basado en entropia
int indxP = (int)datos.umbral*10;

entropyFuzzyClustering<char,int>(
  d_AV,d_CL,numClases,REPETIR,
  BETA,TabNU[indxP]);

// convertir las frecuencias en probabilidades
thrust::device_vector<double> d_Prb(REPETIR,0.0);
for (int k=0; k<numClases; k++){
  d_Prb[k] = d_CL[k]/(double)REPETIR;
}
// calcular entropia para el modelo
double entropia = -1.0 *
  thrust::transform_reduce(d_Prb.begin(),
  d_Prb.end(), multXlog2X<double>(),
  0.0, thrust::plus<double>());

//guardar los datos de entropia en archivo
// "T= Reps= U= Ent= beta= clases= probs= ";
ofsE << datos.iter <<" "
   << REPETIR <<" "<< datos.umbral <<" "
   << entropia   <<" "<< BETA
   <<" "<< numClases    <<" ";
```

```
for (int k=0; k<numClases; k++){
  ofsE << d_Prb[k] <<" ";
}
ofsE << std::endl;

  //calcula tiempo por umbral
  printf( "Umbral:  %2.2f \n", datos.umbral );
  CUDA_CALL( cudaEventRecord( finU, 0 ) );
  CUDA_CALL( cudaEventSynchronize( finU ) );
  float   tmpUmb;
  CUDA_CALL( cudaEventElapsedTime(
    &tmpUmb, iniU, finU ) );
  printf( "Tiempo por umbral:  %3.1f ms\n",
    tmpUmb );

    }
ofsE.close();

  CUDA_CALL( cudaEventRecord( datos.fin, 0 ) );
  CUDA_CALL( cudaEventSynchronize( datos.fin ) );
  //evalua tiempo de calculo entre eventos
  float   tmpCalc;
  CUDA_CALL( cudaEventElapsedTime( &tmpCalc,
    datos.inicio, datos.fin ) );
  printf( "Tiempo calculo :  %3.1f ms\n", tmpCalc );
  CUDA_CALL( cudaEventDestroy( iniU ) );
  CUDA_CALL( cudaEventDestroy( finU ) );

  return 0;
}
```