UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

POSGRADO EN CIENCIA E INGENIERIA DE LA COMPUTACION

ALGORITMOS COMPUTACIONALES DE GENERACION DE CAMINATAS ALEATORIAS Y SU IMPLEMENTACION UTILIZANDO PROCESAMIENTO EN PARALELO

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS

P R E S E N T A :

FELIPE DE JESUS VARGAS TORRES

DIRECTOR DE LA TESIS: DR. VLADIMIR TCHIJOV

MEXICO, D. F. 2002

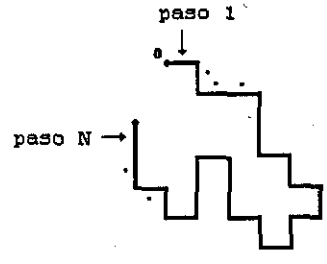
Índice

Introducción

El objetivo de la presente tesis es: estudiar los algoritmos de generación de caminatas aleatorias auto-repelentes y su implementación utilizando el procesamiento en paralelo.

Las caminatas aleatorias auto-repelentes o SARW (*Self Avoiding Random Walk*) sobre una malla son de gran importancia teórica y práctica en la física y química de macromoléculas como un modelo lineal de las cadenas de polímeros(De Gennes (1979)).

Una caminata aleatoria auto-repelente es una trayectoria sobre una malla que no visita la misma posición dos veces, como se muestra en la siguiente figura.



origen

Caminata aleatoria auto-repelente

Sin prestar mucha atención a la simpleza de la definición, se pueden plantear algunas preguntas como:

¿Cuántas trayectorias diferentes de *N* pasos se pueden seguir?

¿Cuál es la distancia promedio que recorren las caminatas de *N* pasos?

Éstas y otras preguntas acerca de las caminatas aleatorias auto-repelentes permanecen aún sin resolver en un sentido estrictamente matemático(Madras (1993)), aunque las comunidades de físicos y químicos han llegado a un consenso sobre la respuesta por medio de una variedad de métodos no rigurosos, así han surgido otros artificios como la simulación.

La simulación es el proceso de diseñar un modelo de un sistema real y llevar a cabo experimentos, utilizando el modelo, con la finalidad de comprender el comportamiento del sistema real.

La simulación Monte Carlo aún es una de las principales herramientas para obtener resultados aproximados o formular conjeturas a las preguntas que se plantearon anteriormente.

En la simulación Monte Carlo se emplean diferentes algoritmos para generar caminatas aleatoria auto-repelentes. Los algoritmos se clasifican como:

- Algoritmos básicos.
- Algoritmos dinámicos.
- Algoritmos estáticos.

En el capítulo 1 de la presente tesis, se explican conceptos fundamentales sobre las caminatas aleatorias auto-repelentes y los algoritmos de generación de las mismas, entre ellos el algoritmo de dimerización.

El algoritmo de dimerización es un algoritmo estático que genera caminatas aleatorias auto-repelentes independientes uniformemente distribuidas sobre el conjunto de todas las caminatas de N pasos. Esencialmente, es un algoritmo recursivo(Madras (1993)).

El algoritmo de dimerización es uno de los algoritmos más eficientes para generar caminatas aleatorias auto-repelentes (Madras (1993)).

En la presente tesis se utiliza el algoritmo de dimerización, por las características antes mencionadas.

La computadora es una herramienta que permite, entre otras tareas, realizar cálculos numéricos complejos. También nos permite el tratamiento de problemas para los que la modelación matemática del sistema es muy difícil o imposible. Las técnicas de simulación por medio de computadora han permitido el modelado y estudio de muchos sistemas de estructura compleja.

En ausencia de resultados matemáticamente rigurosos, la simulación por medio de computadora ha jugado un papel importante en la obtención de soluciones numéricas aproximadas o en la formulación de conjeturas.

Una computadora en paralelo es una computadora que contiene un conjunto de microprocesadores instalados capaces de trabajar coordinadamente para ejecutar un programa.

El procesamiento en paralelo se refiere al concepto de ejecutar un programa por medio de la división del mismo en múltiples fragmentos que pueden ser ejecutados simultáneamente por diferentes microprocesadores.

Existen diferentes modelos para crear programas en paralelo, por ejemplo:

- Modelo MPI (*Message Passing Interface*).
- Modelo de Hilos o *Threads*.

La implementación del modelo de hilos se hace por medio de librerías las cuales pueden ser las librerías POSIX o las librerías OpenMP (www.llnl.gov/computing/tutorials/workshops/workshop).

La programación en paralelo es la evolución natural de la programación serial. La programación en paralelo ha sido considerada como "el nivel más alto de evolución de la programación" (www.llnl.gov/computing/tutorials/workshops/workshop).

Existen dos razones básicas para el uso del procesamiento en paralelo:

1. El procesamiento en paralelo reduce significativamente el tiempo de ejecución de un programa.
2. El procesamiento en paralelo da la oportunidad de resolver problemas más robustos o complejos.

En la presente tesis se codifica el algoritmo de dimerización utilizando las librerías OpenMP para que pueda ser ejecutado por varios microprocesadores; lo cual lo hace más eficiente.

En el capítulo 2 se explican conceptos fundamentales acerca de los hilos, computadoras en paralelo y procesamiento en paralelo.

En el capítulo 3 se explica la manera de utilizar las librerías OpenMP para desarrollar programas en paralelo.

En el capítulo 4 se explican los aspectos computacionales que se tomaron en cuenta para la codificación del algoritmo de dimerización.

En el capítulo 5 se explican los diferentes experimentos que se realizaron y los nuevos resultados obtenidos generando caminatas aleatorias auto-repelentes hasta de 20000 pasos en cuatro dimensiones y además se compara el tiempo de ejecución del programa ejecutado de una manera serial y en paralelo,

Por último en el capítulo 6 se dan las conclusiones a las que se llegaron a partir de los resultados experimentales.

Los experimentos se realizan en cuatro dimensiones debido a la sospecha que existen correcciones logarítmicas que se deben aplicar al calcular los valores promedio relacionados con las caminatas aleatorias auto-repelentes (Madras (1993)).

Apéndice A muestra los valores de los exponentes críticos γ y ν (conjeturas).

Apéndice B muestra los valores de c_N para N pequeños.

Apéndice C muestra el juego de funciones OpenMP.

Apéndice D muestra algunos ejemplos utilizando las librerías OpenMP.

1

Caminatas Aleatorias Auto-repelentes

El objetivo de la presente tesis, como se mencionó anteriormente, es: **estudiar los algoritmos de generación de caminatas aleatorias auto-repelentes y su implementación utilizando el procesamiento en paralelo.** Pero antes de abordar nuestro objetivo se necesitan conocimientos fundamentales acerca de las caminatas aleatorias auto-repelentes, procesamiento en paralelo y del compilador OpenMP que se utilizará en la paralelización del algoritmo de dimerización.

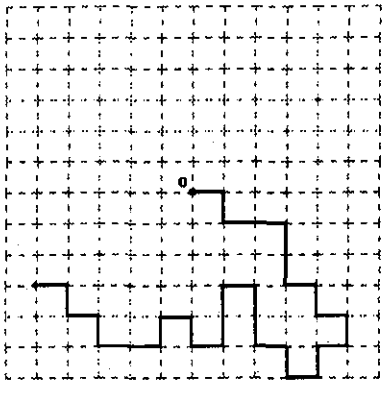
En los diferentes capítulos de esta tesis se proporcionan los conocimientos fundamentales para cumplir con nuestro objetivo.

En este capítulo se explican los conceptos fundamentales relacionados con las caminatas aleatorias auto-repelentes. Se plantean las preguntas fundamentales, se dan las respuestas exactas para las preguntas fundamentales cuando las caminatas aleatorias auto-repelentes son pequeñas. En caso contrario, se plantea un método para obtener respuestas aproximadas por computadora, llamado simulación Monte Carlo. Por último, se dan algunos algoritmos que se utilizan en la simulación Monte Carlo para generar caminatas aleatorias auto-repelentes.

1.1 Caminatas aleatorias auto-repelentes o SARW (*Self Avoiding Random Walk*)

Supongamos que uno está parado en la intersección de dos calles, llamado punto de origen, en el centro de una gran ciudad cuyas calles están dispuestas en forma cuadricular. Uno escoge una calle al azar y empieza a caminar; en cada intersección entre dos calles se debe escoger si se continua hacia adelante, se da vuelta a la izquierda o a la derecha, como se muestra en la figura 1.1.

Sólo hay una regla: no se debe regresar a cualquier intersección que va se haya visitado durante el recorrido. A este tipo de caminata se le llama caminata aleatoria auto-repelente o SARW (*Self Avoiding Random Walk*).
Si se respeta la regla anterior se puede recorrer una trayectoria de N cuadras o realizar una caminata de N pasos (vea la figura 1.2.)



0 origen

Caminata aleatoria auto-repelente
Fig. 1.1

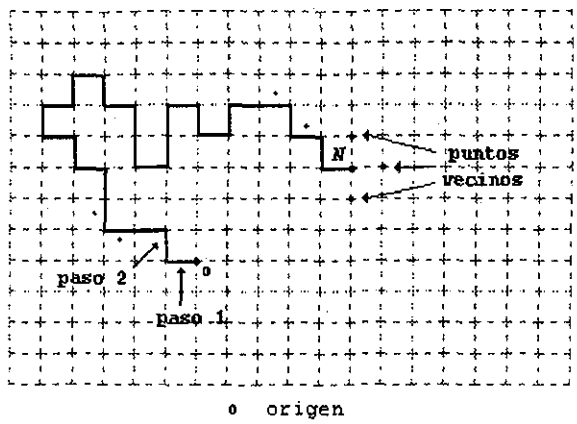
No es necesario restringir la caminata a mallas de dos dimensiones (Madras (1993)). Es posible generalizar el concepto de caminatas aleatorias auto-repelentes cambiando de una malla cuadricular a una triangular u otro tipo de malla (Madras (1993)).

Las caminatas que nos interesan son las caminatas aleatoria auto-repelentes sobre mallas de d dimensiones en Z^d. La razón principal de esto son las correcciones logarítmicas en el comportamiento asintótico de los valores promedio relacionados con las caminatas aleatorias auto-repelentes cuando $N \to \infty$ (Madras (1993)).

En general, una caminata aleatoria auto-repelente se puede representar en una gráfica.

Los puntos de la gráfica serán llamados posiciones o sitios. Las posiciones serán denotadas por las letras u, v, x, y, y sus componentes por $x = (x_1, x_2, \ldots, x_d)$, etc.

A los puntos unidos por el vector unitario a la posición más cercana se les da el nombre de vecinos más cercanos o simplemente vecinos, como se muestra en la figura 1.2.



Los puntos vecinos de una posición
Fig. 1.2

Sea ω una caminata aleatoria auto-repelente de N pasos en Z^d, con origen en la posición x y definida por las posiciones $(\omega(0), \omega(1), \ldots, \omega(N))$ con $\omega(0) = x$ que satisface $|\omega(j+1) - \omega(j)| = 1$ y $\omega(i) \neq \omega(j)$ para toda $i \neq j$.

Se escribe $|\omega| = N$ para denotar la longitud de ω. Denotamos los componentes de $\omega(j)$ por $\omega_i(j)(i = 1, 2, \ldots, d)$.

Para las caminatas aleatorias auto-repelentes se plantean dos preguntas fundamentales:

¿Cuántas trayectorias diferentes de N pasos se pueden seguir?

Suponiendo que se sigue una trayectoria cualquiera, ¿cuál es la distancia promedio que recorren las caminatas de N pasos?

Dejemos que c_N denote el número de caminatas aleatorias auto-repelentes de N pasos que inician en el origen 'o'.

La primera de las preguntas básicas nos pide el valor de c_N o sea el número exacto de todas las posibles trayectorias de N pasos. Se puede preguntar por el comportamiento de c_N cuando $N \to \infty$.

Por convención $c_0 = 1$; se pueden calcular los valores exactos de c_N (en función de d) para valores de N pequeños (Madras (1993)).

Por ejemplo $c_1 = 2d$, $c_2 = 2d(2d-1)$, $c_3 = 2d(2d-1)^2$ y $c_4 = 2d(2d-1)^3 - 2d(2d-2)$

Calcular los valores de c_N es un problema bastante difícil cuando N se incrementa.

Las tablas en el Apéndice B muestran los valores exactos de c_N para valores de N pequeños en las dimensiones $d = 2, \ldots, 6$.

Como se mencionó anteriormente, resulta difícil calcular los valores exactos de c_N cuando $N \to \infty$, pero se pueden calcular cotas simples de su comportamiento de la siguiente manera.

Una cota superior de c_N es la multiplicación del número de opciones que se tienen para el paso inicial, $2d$ opciones, por el número de opciones de los pasos faltantes, debido a que no se permite un regreso inmediato tenemos a lo mucho $2d - 1$ opciones para cada uno de los $N - 1$ pasos faltantes. De donde tenemos que una cota superior de c_N es $2d(2d - 1)^{N-1}$.

Para hallar una cota inferior multiplicamos el número de opciones sólo en dirección positiva en cada paso. Cada caminata es necesariamente auto-repelente (Madras (1993)). Entonces, c_N se encuentra en el intervalo:

$$d^N \leq c_N \leq 2d(2d - 1)^{N-1} \qquad (1.1)$$

Para la segunda pregunta, se necesita medir la probabilidad para cada caminata aleatoria auto-repelente de N pasos. La probabilidad que se usará es la uniforme, que asigna un peso igual a c_N^{-1} para cada caminata.

En las fórmulas que siguen, los paréntesis triangulares denotan valores aproximados. Así el valor aproximado de la distancia al cuadrado promedio está dado por:

$$\langle\, |\, \omega(N)\, |^2 \,\rangle = \frac{1}{c_N} \sum_{\omega:\, |\omega|=N} |\, \omega(N)\, |^2 \qquad (1.2)$$

La sumatoria sobre ω es sobre todas las caminatas aleatorias auto-repelentes de N pasos. Como en c_N, la distancia al cuadrado promedio también puede ser calculada para valores de N pequeños, pero se vuelve difícil o hasta imposible de calcular conforme N se va incrementando.

Para las caminatas aleatorias auto-repelentes, se cree que el valor de c_N tiene un crecimiento exponencial al cual se debe aplicar la corrección de la ley de potencias (Madras (1993)).

También se cree que la distancia al cuadrado promedio en función de N, no será siempre lineal de acuerdo al número de pasos (Madras (1993)).

Las conjeturas anteriores están en armonía con las propiedades conocidas de otros modelos de la mecánica estadística, y están soportados por cálculos numéricos. Las conjeturas sobre c_N y $\langle\, |\, \omega(N)\, |^2 \,\rangle$ se pueden expresar de la siguiente manera:

$$c_N \sim A\mu^N N^{\gamma-1} \qquad (1.3)$$

y

$$\left\langle |\ \omega(N)\ |^2 \right\rangle \sim DN^{2\nu} \qquad (1.4)$$

En las fórmulas (1.3) a (1.6) A, D, μ, γ y ν son constantes positivas dependientes de la dimensión. μ es una constante llamada constante de conectividad, γ y ν son los exponentes críticos. En cuatro dimensiones, las ecuaciones anteriores deben ser modificadas por un factor logarítmico(Madras (1993)):

$$c_N \sim A\mu^N \left[\log N\right]^{1/4},\ d = 4 \qquad (1.5)$$

$$\left\langle |\ \omega(N)\ |^2 \right\rangle \sim DN \left[\log N\right]^{1/4},\ d = 4 \quad (1.6)$$

Como se vió, las respuestas para las preguntas fundamentales se conocen sólo para valores pequeños de N.

Es ampliamente aceptado que buscar fórmulas exactas para estas preguntas es un problema bastante difícil, que hoy en día está fuera del alcance de los métodos tradicionales.

Una manera de dar una respuesta, por lo menos una respuesta aproximada, sería utilizar técnicas como la simulación por computadora.

Para valores grandes de N, se utiliza la simulación Monte Carlo con el fin de dar una respuesta aproximada a las preguntas fundamentales. Para valores pequeños de N se usa la enumeración exacta.

1.2 Simulación Monte Carlo

La simulación Monte Carlo es útil para obtener aproximaciones estadísticas de los valores de la constante de conectividad, exponentes críticos y otras variables relacionados con las caminatas aleatorias auto-repelentes.

Esencialmente, la simulación Monte Carlo es un experimento computacional que toma muestras aleatorias de un sistema en particular. Después de obtener suficientes muestras, se pueden utilizar técnicas estadísticas para obtener aproximaciones de los valores deseados.

Las aproximaciones numéricas pueden ser evidencias a favor o en contra de las conjeturas, aunque no son pruebas matemáticamente estrictas.

En esta tesis se plantean algunos algoritmos que se utilizan en la simulación Monte Carlo para generar caminatas aleatorias auto-repelentes y además se elegirá uno de ellos en base a sus características para su implementación computacional.

1.3 Algoritmos para generar caminatas aleatorias auto-repelentes

Existen diferentes tipos de algoritmos para generar caminatas aleatorias auto-repelentes, los cuales son:

° Algoritmos básicos.
° Algoritmos estáticos.
° Algoritmos dinámicos.

De aquí en adelante se usa la notación S_N para denotar el conjunto de todas las caminatas aleatorias auto-repelentes de N pasos que inician en el origen 'o'.

Algoritmos básicos

I. Muestra Elemental Simple ESS(*Elementary Simple Sampling*)

Este algoritmo genera caminatas aleatorias hasta que obtiene una auto-repelente de N pasos.

Los pasos del algoritmo son los siguientes:

1. Sea $\omega(0)$ el origen e $i = 0$.
2. Incrementar i en uno. Escoger aleatoriamente uno de los $2d$ vecinos de $\omega(i-1)$ y dejar que $\omega(i)$ sea esa posición.
3. Sí $\omega(i) = \omega(j)$ para algún $j = 0,1,\ldots,i-1$, regresar al paso 1. De otra manera, ir al paso 2 sí $i < N$ o parar sí $i = N$.

Cuando este algoritmo termina, la caminata $\omega \equiv (\omega(0),\ldots,\omega(i))$ es una caminata aleatoria auto-repelente. Las caminatas generadas por este algoritmo están uniformemente distribuidas sobre S_N(Madras (1993)).

Desde un punto de vista computacional, el algoritmo es demasiado lento cuando N es moderadamente grande. Si T_{ESS} representa la cantidad de tiempo de cálculos que el algoritmo ESS requiere para generar una caminata aleatoria auto-repelente de N pasos, tenemos(Madras (1993)):

$$T_{ESS} = \left(\frac{2d}{\mu}\right)^{N+o(N)}$$

Así que la complejidad del algoritmo es exponencial.

II. Muestra Simple Sin-Regreso NRSS(*Non-Reversed Simple Sampling*)

Este algoritmo genera una caminata aleatoria sin regreso inmediato hasta que obtiene una auto-repelente de N pasos.

Los pasos del algoritmo son los siguientes:

1. Sea $\omega(0)$ la posición de origen. Escoger aleatoriamente uno de los $2d$ vecinos de la posición de origen, dejar que $\omega(1)$ sea esa posición. Fijar $i = 1$.
2. Incrementar i en uno. De los $2d-1$ vecinos de $\omega(i-1)$ que son diferentes de $\omega(i-2)$, escoger uno aleatoriamente, y dejar que $\omega(i)$ sea esa posición.
3. Sí $\omega(i) = \omega(j)$ para algún $j = 0,1,\ldots,i-1$, regresar al paso 1. De otra manera, ir al paso 2 sí $i < N$ o parar sí $i = N$.

El algoritmo NRSS genera caminatas aleatorias auto-repelentes uniformemente distribuidas sobre S_N. El tiempo de cálculos es(Madras (1993)):

$$T_{NRSS} = \left(\frac{2d - 1}{\mu}\right)^{N+o(N)}$$

Los algoritmos anteriores, el algoritmo ESS y el algoritmo NRSS, sufren del mismo problema: desde un punto de vista computacional son muy poco eficientes y ambos requieren una cantidad de tiempo exponencial.

Algoritmos estáticos

Estos algoritmos generan secuencias de caminatas aleatorias auto-repelentes independientes o secuencias de conjuntos independientes de caminatas aleatorias auto-repelentes (las caminatas dentro de cada conjunto posiblemente están altamente correlacionadas).

Dos algoritmos básicos de generación de caminatas aleatorias auto-repelentes independientes se discutieron en la sección anterior, llamados algoritmo ESS y algoritmo NRSS; una generalización de éstos métodos es usar "tramos" (*strides*) para construir caminatas en vez de pasos únicos. Un m tramo es una caminata aleatoria auto-repelente de longitud m. Para el siguiente algoritmo, sea m un número entero no negativo fijo.

I. Algoritmo tramos de m-pasos SM(m) (*m-Step Stride Method*)

Este algoritmo genera una caminata aleatoria auto-repelente de longitud km (en donde k es un entero). Requiere una lista $\psi[1], \ldots, \psi[c_N]$ de todas las caminatas aleatorias auto-repelentes de m pasos.

Los pasos del algoritmo son los siguientes:

1. Sea ω una caminata de 0 pasos que consiste de una posición, el origen. Fijar $i = 0$.
2. Incrementar i en uno. Escoger aleatoriamente un entero j dentro del rango $(1, \ldots, c_N)$. Redefinir ω para ser igual a $\omega \circ \psi[j]$, la concatenación de la caminata actual con $\psi[j]$.
3. Sí ω no es auto-repelente, regresar al paso 1. De otra manera, ir al paso 2 sí $i < k$ o parar sí $i = k$.

La cantidad de tiempo de cálculos requerido para generar una caminata aleatoria auto-repelente usado por este algoritmo es(Madras (1993)):

$$T_{SM(m)} = \frac{(c_m)^{N/m}}{c_N} = \left(\frac{c_m^{1/m}}{\mu}\right)^{N+o(N)}$$

De aquí vemos que el tiempo crece exponencialmente con respecto a N, pero lentamente si m es grande. Pero mientras más grande sea m se necesitan más recursos de computadora para generar y almacenar la lista de las caminatas aleatorias auto-repelentes de m pasos.

II. Algoritmo de dimerización

Una manera diferente de generar caminatas aleatorias auto-repelentes uniformemente distribuidas sobre S_N es el algoritmo de dimerización. Si deseamos generar una caminata aleatoria auto-repelente, se generan dos caminatas aleatorias auto-repelentes independientes de $(N/2)$ pasos (llamadas *dimers*) y se tratan de concatenar. Si el resultado es una caminata auto-repelente, el algoritmo termina; de otra manera, se descartan los dos *dimers* y se empieza de nuevo.

Para generar cada uno de los *dimers* de $(N/2)$ pasos se generan dos *dimers* de $(N/4)$ pasos y se tratan de concatenar, y así sucesivamente. Se puede expresar ésto con el siguiente algoritmo recursivo.

Algoritmo de dimerización DIM(N)
Sea N_A un número entero pequeño llamado "longitud de Alexandrowics".
Los pasos del algoritmo son los siguientes:

1. Si $N \leq N_A$, generar una caminata ω de N pasos por medio del algoritmo NRSS y detenerse.
2. Si $N > N_A$ sea $N_1 = \lfloor N/2 \rfloor$ y $N_2 = N - N_1$.
3. Recursivamente se ejecutan DIM(N_1) y DIM(N_2), creando las caminatas aleatorias auto-repelentes ω^1 y ω^2 respectivamente.
4. Sea $\omega = \omega^1 \circ \omega^2$, la concatenación de ω^1 con ω^2. Si ω es auto-repelente, pararse; de otra manera, regresar al paso 2 y empezar de nuevo.

Debemos notar que en el paso 1 el algoritmo NRSS puede ser remplazado por cualquier otro algoritmo básico.

La cantidad de tiempo de cálculos para generar una caminata aleatoria auto-repelente por el algoritmo de dimerización es(Madras (1993)):

$$T_{DIM(N)} \approx \frac{(2BN^{\gamma-1})^k}{2^{(\gamma-1)k(k-1)/2}} T_{DIM(N_0)} \approx d_0 N^{d_1 \log_2 N + d_2}$$

Se supone que $N = 2^k N_A$, en donde k es el número de niveles de recursión. El parámetro d_i es independiente de N:

$$d_1 = \frac{\gamma - 1}{2}, \quad d_2 = \frac{\gamma - 1}{2} + \log_2(2B) = \frac{5 - 3\gamma}{2} + \log_2 A$$

d_0 depende de N_A. Se puede concluir que el crecimiento de $T_{DIM(N)}$ es menor que el crecimiento exponencial con respecto a N. Se puede anticipar que los valores para d_1 son pequeños, de acuerdo al Apéndice A, se espera que d_1 sea 11/64 para 2 dimensiones, 0.081... en 3 dimensiones y 0.0 para 4 o más dimensiones. En particular, es rigurosamente conocido que $\gamma = 1$ para 5 o más dimensiones, así el crecimiento de $T_{DIM(N)}$ es polinomial en 5 o más dimensiones.

III. Algoritmo de Render-Reynolds (*Render-Reynolds Algorith...*)

Este algoritmo genera aleatoriamente conjuntos $A_i \subset S_i$ $(i \geq 0)$ de caminatas aleatorias auto-repelentes. El algoritmo requiere de un parámetro real z de

valor entre 0 y 1. Se denota los $2d$ vectores unitarios de Z^d por $e_1, \ldots, e_{2d}$ (positivos y negativos).

Los pasos del algoritmo son los siguientes:
1. Sea A_0 el conjunto de caminatas de 0 pasos (el origen). Fijar $i = 0$. (inicialmente A_k esta vació para cada $k \geq 1$).
2. Independientemente, para cada caminata ω en A_i y para cada $j = 1, \ldots, 2d$ con probabilidad $1 - z$, no hacer nada; de otra manera (con probabilidad igual a z) sumar el paso e_j a ω, y si el resultado es una caminata aleatoria auto-repelente, agregarla al conjunto A_{i+1}.
3. Incrementar i en uno y regresar al paso 2.

Algoritmos dinámicos

Los algoritmos dinámicos generan nuevas caminatas modificando (o actualizando) las caminatas aleatorias auto-repelentes previamente generadas.

Existen tres variantes de los algoritmos dinámicos los cuales son:
1. Algoritmos dinámicos de longitud constante (*Length-conserving dynamic methods*).
2. Algoritmos dinámicos de longitud variable (*Variable-length dynamic methods*).
3. Algoritmos dinámicos de puntos finales fijos (*Fixed-endpoint Methods*).

Cada uno de éstos algoritmos corresponde a una cadena de Markov que toma una caminata y trata de cambiarla de una manera aleatoria para generar otra caminata de la misma longitud.

Algoritmos dinámicos de longitud constante

Si los algoritmos dinámicos cambian la primera parte de una caminata cualquiera, luego su punto inicial puede ya no ser el origen, en cuyo caso se asumirá implícitamente que la caminata resultante es trasladada al origen, por lo tanto pertenece al conjunto S_N.

I. El algoritmo del pivote (*The pivot algorithm*)

El algoritmo del pivote toma aleatoriamente una posición de la caminata y le llama "pivote", el cual divide la caminata en dos partes; le aplica una transformación escogida aleatoriamente a una parte de la caminata, usando la posición pivote como el origen (Madras (1993)). Como de costumbre, el resultado es aceptado sí la caminata es auto-repelente.

La versión básica del algoritmo del pivote es la siguiente:
1. Sea $\omega^{(0)}$ cualquier caminata aleatoria auto-repelente en S_N. Fijar $t = 0$.
2. Escoger un entero I aleatoriamente de manera uniforme de $\{0,1,\ldots,N-1\}$. Fijar $x = \omega^{(t)}(I)$ (posición pivote). Escoger una transformación G aleatoriamente. Fijar $\tilde{\omega} = \omega^{(t)}(l)$ para cada $l \leq I$ y $\tilde{\omega} = G(\omega^{(t)}(l))$ para cada $l > I$.

3. Si $\tilde{\omega}$ es una caminata aleatoria auto-repelente , luego fijar $\omega^{[t+1]} = \tilde{\omega}$; de otra manera, fijar $\omega^{[t+1]} = \omega^{[t]}$.
4. Incrementar t en uno e ir al paso 2.

Existen algunos otros algoritmos dinámicos de longitud constante como el algoritmo "General" de la k-posición (*"Most General" k-site algorithm*) (Madras (1993)).

Algoritmos dinámicos de longitud variable

Cuando se utilizan los algoritmos de longitud variable, el análisis estadístico de los datos puede ser más complicado (Madras (1993)).

II. El algoritmo de Berretti-Sokal (*The Berretti-Sokal algorithm*)

El algoritmo de Berretti-Sokal ésta diseñado para hacer un muestreo sobre el conjunto de todas las caminatas aleatorias auto-repelentes de todas las longitudes posibles. La idea básica es en cada paso se borra el último tramo de una caminata o se intenta incrementar la longitud de la caminata agregando un tramo al final de la caminata (rechazando el intento sí el resultado no es una caminata auto-repelente).

Algoritmo de Berretti-Sokal (B-S) (Berretti-Sokal algorithm)
Los pasos del algoritmo son los siguientes:
1. Sea $\omega^{[0]}$ una caminata aleatoria auto-repelente. Fijar $t = 0$.
2. Sea $N = |\omega^{[t]}|$. Generar una variable aleatoria X, que puede tomar el valor +1 con probabilidad $2dz/(1 + 2dz)$ o -1 con probabilidad $1/(1 + 2dz)$. Sí $X = +1$, ir al paso 3; sí $X = -1$, ir al paso 4.
3. Tratar de agregar un paso a $\omega^{[t]}$: escoger aleatoriamente de manera uniforme uno de los $2d$ vecinos más cercanos a $\omega^{[t]}(N)$; llamar a éste punto Y. Sí Y no es una posición de $\omega^{[t]}$, fijar $\omega^{[t+1]} = (\omega^{[t]}(0), \ldots, \omega^{[t]}(N), Y)$; sí Y es una posición de $\omega^{[t]}$, luego fijar $\omega^{[t+1]} = \omega^{[t]}$. Incrementar t en uno e ir al paso 2.
4. Borrar el último paso de $\omega^{[t]}$: si $N > 0$, luego fijar $\omega^{[t+1]} = (\omega^{[t]}(0), \ldots, \omega^{[t]}(N-1), Y)$; sí $N = 0$, luego fijar $\omega^{[t+1]} = \omega^{[t]}$ (el paso 0 de la caminata). Incrementar t en uno e ir al paso 2.

Existen algunos otros algoritmos dinámicos de longitud variable como el algoritmo de pegar y cortar (*The join-and-cut algorithm*).

Algoritmos dinámicos de extremos fijos

III. Algoritmo de BFACF (*BFACF algorithm*)

Los pasos del algoritmo son los siguientes:
1. Sea $\omega^{[0]}$ cualquier caminata. Fijar $t = 0$.
2. Escoger aleatoriamente un entero I de manera uniforme sobre rango $\{0, 1, \ldots, |\omega^{[t]}|-1\}$.
3. Considerar las $2d - 2$ caminatas ω que serían obtenidas por el movimiento de *I-esima* de $\omega^{[t]}$ en una de las direcciones perpendiculares al vector

$\omega^{[t]}(I-1) - \omega^{[t]}(I)$. Escoger una de éstas caminatas aleatoriamente, con probabilidad $p(|\tilde{\omega}| - |\omega^{[t]}|)$. (Sí estas $2d-2$ probabilidades agregar más a $q < 1$, luego también $\omega = \omega^{[t]}$ con la probabilidad $1-q$).

4. Sí $\tilde{\omega}$ es auto-repelente, luego $\omega^{[t+1]} = \tilde{\omega}$; de otra manera, fijar $\omega^{[t+1]} = \omega^{[t]}$.
5. Incrementar t en uno e ir al paso 2.

1.4 Observaciones

Todos los algoritmos que generan caminatas aleatorias auto-repelentes independientes sufren del problema de la ineficiencia, en diferente grado.

Los algoritmos básicos tienen el mismo problema, son muy poco eficientes al generar caminatas aleatorias auto-repelentes, requieren de una cantidad de tiempo exponencial con respecto a N.

El problema de los algoritmos dinámicos, en general, es que generan caminatas correlacionadas debido a que generan caminatas modificando las caminatas aleatorias auto-repelentes previamente generadas.

En contraste, el algoritmo de dimerización es uno de los mejores y más eficientes algoritmos estáticos para la generación de caminatas aleatorias auto-repelentes en cualquier dimensión; cada caminata de N pasos tiene la misma probabilidad para ser generada (Madras (1993)).

Los algoritmos más populares para la generación de caminatas aleatorias auto-repelentes son el algoritmo del pivote (Madras (1988); Eizenberg (1993); Caricciolo 1997)), el algoritmo de enriquecimiento (Rapoport (1985); Rapoport 1984); Madras (1993)) en su implementación recursiva y aleatoria (Grassberger and Schafer (1994); Grassberger (1994)) y el algoritmo de dimerización (Alexandrowicz (1969); Dayantis (1991); Everaers (1995); Rodríguez-Romo (1998)).

El algoritmo de dimerización ha sido aplicado de una manera exitosa en la simulación Monte Carlo en 2 y 3 dimensiones (Dayantis (1991); Everaers (1995)) y en la modelación de tiempo continuo en 4 dimensiones (Rodríguez-Romo (1998)). El algoritmo de dimerización también se ha usado como "inicio equilibrado" (*equilibrium start*) en el algoritmo de pivote (Madras (1988); Eizenberg (1993)).

Se ha probado rigurosamente (Madras (1993), lemma 9.3.1) que el algoritmo de dimerización genera caminatas aleatorias auto-repelentes estadísticamente independientes y uniformemente distribuidas sobre el conjunto de todas las caminatas aleatorias auto-repelentes de N pasos. Esta es una ventaja, que implica que se puede utilizar como un método estándar en la estimación de los valores promedio (como distancia al cuadrado promedio etc.).

En la presente tesis se toma el algoritmo de dimerización, por las razones expuestas anteriormente, para su implementación en procesamiento en paralelo.

Antes de intentar implementar el algoritmo de dimerización, de tal manera que pueda ser ejecutado en una computadora en paralelo, necesitamos conceptos fundamentales de la programación en paralelo como, por ejemplo el concepto de hilo, las diferentes arquitecturas de las computadoras en paralelo, las

diferentes maneras de ejecutar un programa en una computadora en paralelo y por último, las herramientas que se pueden utilizar para paralelizar un programa. Éstos y otros conceptos se explican en el siguiente capítulo.

2

Hilos y Procesamiento en paralelo

En este capítulo se explican los conceptos fundamentales relacionados con el procesamiento en paralelo.

Se explica que es un hilo y los diferentes tipos de hilos. También se explican las arquitecturas básicas de las computadoras en paralelo; las diferentes maneras que existen de ejecutar un programa en una computadora en paralelo.

Por último se explican algunos otros modelos de programación en paralelo y las posibles herramientas que se utilizan para su implementación(en la presente tesis se utiliza el modelo de hilos).

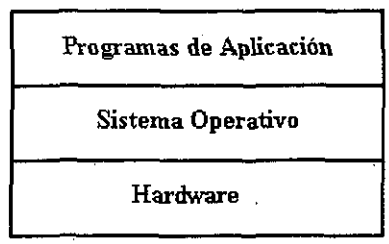
2.1 Sistema operativo

El programa básico en cualquier sistema de computo es el sistema operativo, pues controla todos los recursos de la computadora y proporciona una base en la cual pueden ejecutarse programas de aplicación (como procesadores de texto, compiladores, hojas de cálculo, juegos, etc.).

De lo anterior se puede ver que el sistema operativo tiene dos funciones asignadas(Silberschatz (1992)):
1) Ejecutar los programas del usuario.
2) Administrar los recursos del sistema de computo.

Ejecutar los programas del usuario

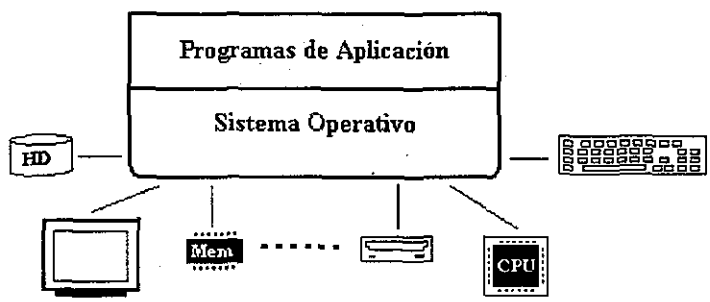
Una de las funciones del sistema operativo es permitir al usuario ejecutar sus programas de aplicación. Es decir, actúa como interface entre el usuario y el hardware de la computadora para poder ejecutar un programa de aplicación(Silberschatz (1992); Márquez García (1996)), como se muestra en la figura 2.1.



Sistema operativo como interface
Figura 2.1

Administrar los recursos del sistema de computo

Otra de las funciones del sistema operativo es asignar los recursos del sistema para proporcionar un ambiente en el cual el usuario pueda ejecutar sus programas de aplicación, como se muestra en la figura 2.2.

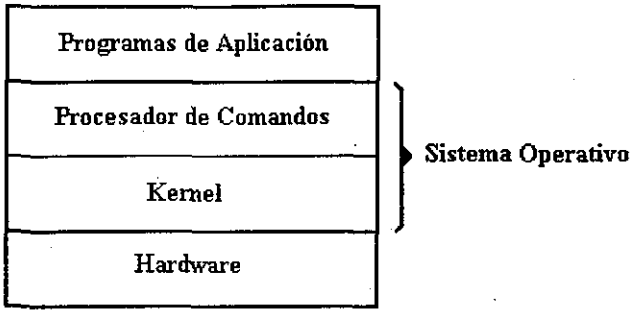


Sistema operativo como asignador de recursos
Figura 2.2

Se puede pensar que el sistema operativo es un conjunto de programas que administran el hardware y el software para ejecutar programas de aplicación.

2.2 Módulos del sistema operativo

Para construir el ambiente, en el cual se puedan ejecutar los programas de aplicación, el sistema operativo LINUX se divide en diferentes módulos(Silberschatz (1992)), como se muestra en la figura 2.3.



Módulos del sistema operativo
Figura 2.3

Procesador de comandos

El procesador de comandos o interface con el usuario, también llamado *shell*, es generalmente un programa que permite al usuario solicitar la ejecución de un programa de aplicación al *kernel*. Cuando el programa de aplicación termina, el shell recobra el control, que despliega el prompt para que el usuario pueda ejecutar otro programa de aplicación(Duncan (1988)).

Núcleo *(Kernel)*

Para ejecutar un programa de aplicación, un número de tareas necesitan ejecutarse antes. Las instrucciones y datos del programa de aplicación deben ser cargados en la memoria, los dispositivos de Entrada/Salida deben ser inicializados y algunos otros recursos de sistema deben ser preparados. El *kernel* realiza todas éstas tareas para poder ejecutar un programa de aplicación.

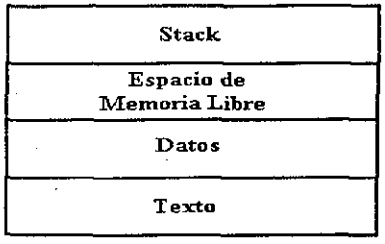
Un concepto clave en cualquier sistema operativo es el de proceso; el proceso es la unidad de trabajo del *kernel*.

Un proceso es básicamente un programa de aplicación en ejecución. Los programas de aplicación pueden usar todos los recursos del sistema disponibles por medio de un proceso(Vahalia (1996)).

Cuando el *kernel* lee un programa y lo carga en memoria primaria para ejecutarse, lo convierte en un proceso cuyas instrucciones serán ejecutadas por el microprocesador(Silberschatz (1992)).

Un proceso se compone de tres bloques fundamentales que se conocen como segmentos(Northup (1996)) (vea la figura 2.4). Estos segmentos son:

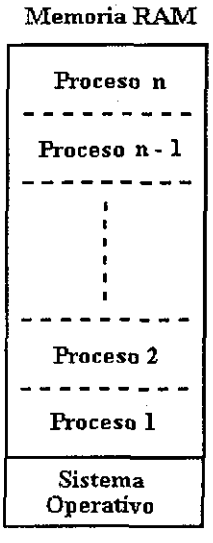
1) Segmento de texto. Contiene las instrucciones que ejecutará el microprocesador. Este bloque es una copia del bloque de texto del programa que se encuentra almacenado en un archivo.
2) Segmento de datos. Contiene los datos del programa.
3) Segmento de pila o stack. Lo crea el kernel automáticamente para almacenar las variables automáticas de un programa.



Segmentos de un proceso
Fig. 2.4

A la colección del código, datos y stack se le conoce como imagen del proceso (*process image*).

Cada proceso corre en su propia memoria asignada y no puede interactuar con otros procesos excepto por medio del kernel (vea la figura 2.5).



Mapa de memoria
Figura 2.5

Si hay varios procesos ejecutándose, éstos procesos son tareas separadas, cada uno de ellos con sus propios derechos y responsabilidades.

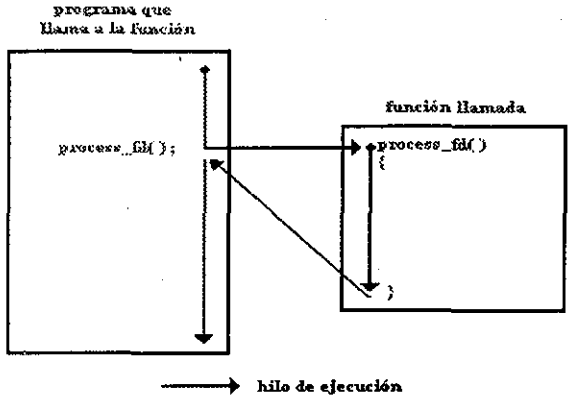
Si un proceso se corrompe, al fallar no afectará a otros procesos dentro del sistema.

Cuando se esta ejecutando un proceso, se dice que el sistema se está ejecutando en el contexto del proceso(Vahalia (1996); Bach (1986)).

2.3 Introducción a los hilos

Cuando se ejecuta un programa, el flujo de instrucciones correspondiente se denomina hilo de ejecución o simplemente hilo (*thread*). Un hilo es una secuencia de instrucciones a ser ejecutadas.

Un proceso dentro del sistema operativo LINUX consiste de un único hilo. Las instrucciones se ejecutan en secuencia, una instrucción a la vez, siguiendo la secuencia lógica del programa hasta terminar(Robbins (1997))(vea la figura 2.6).

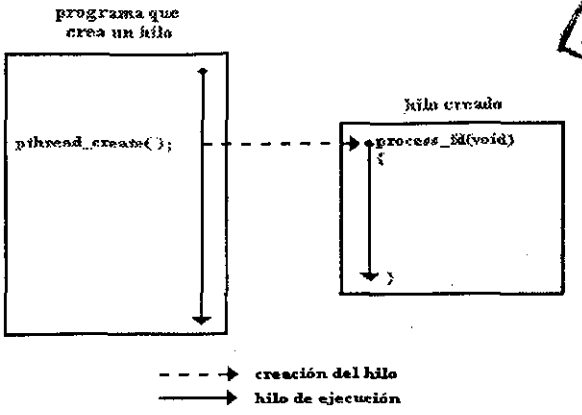


Sólo un hilo en ejecución
Figura 2.6

Al proceso tradicional de LINUX que consta de un sólo hilo de ejecución se le conoce como proceso pesado (*heavy weight process*).

Así como se puede tener múltiples procesos corriendo bajo el control del sistema operativo LINUX, también se puede tener múltiples hilos corriendo dentro de un mismo proceso.

Un proceso puede crear hilos independientes para ejecutar diferentes bloques de código(Robbins (1997)), como se muestra en la figura 2.7.



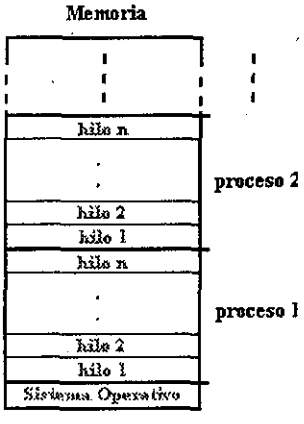
Múltiples hilos de ejecución
Figura 2.7

El hilo se separa y ejecuta un flujo de instrucciones independiente, y nunca vuelve al punto de invocación.

El programa principal o invocador se sigue ejecutando de forma normal.

Sí dentro de un proceso existen múltiples hilos, a éstos se les conoce como procesos ligeros (*light weight process*).

En el sistema operativo LINUX, un hilo existe dentro de un proceso, y utiliza los recursos asignados al proceso, como se muestra en la figura 2.8.



Múltiples hilos dentro de un proceso
Figura 2.8

A diferencia de los procesos, los hilos que existen dentro de un mismo proceso comparten la memoria con los otros hilos. De esta manera los hilos pueden accesar las mismas variables globales, la misma memoria dinámica, el mismo conjunto de descriptores de archivos, etc.

Todos éstos hilos se ejecutan en paralelo (usando intervalos de tiempo, o si la computadora tiene varios microprocesadores, ellos corren de verdad en paralelo).

Un hilo ejecuta un flujo de instrucciones independiente de otros y por lo tanto mantiene su propio:

- Stack.
- Prioridad de ejecución.
- Datos específicos de hilo (TDS o *Thread Specific Data*).

Por otro lado como un grupo de hilos comparte la misma memoria, si un hilo corrompe el contenido de la memoria, los otros hilos sufrirán el mismo daño.

2.4 Tipos de hilos

Se distinguen dos tipos de hilos(Northup (1996)):

- Hilos a nivel Kernel (KLT o *Kernel Label Threads*).
- Hilos a nivel usuario (ULT o *User Label Threads*).

Algunos sistemas operativos soportan los dos tipos de hilos como por ejemplo el sistema operativo Solaris.

Hilos a Nivel Kernel (KLT)

En hilos a nivel kernel, el sistema operativo conoce la existencia de los hilos. La administración (la conmutación entre hilos, sincronización entre hilos etc.) de los hilos la realiza el kernel por medio de llamadas al sistema(Vahalia (1996); Northup (1996)).

Ventajas de usar KLT(Vahalia (1996)):

- Varios hilos del mismo proceso pueden ser ejecutados simultáneamente por diferentes microprocesadores; el bloqueo se hace a nivel de hilos.
- *Las rutinas del kernel pueden ser multihilos.*
- El programador puede ajustar el número de KLTs.

Desventajas de usar KLT:

- El cambio de un hilo a otro dentro de un mismo proceso involucra al kernel, esto resulta en una perdida de tiempo.

Hilos a nivel usuario (ULT)

En los hilos a nivel usuario, el sistema operativo no se entera de su existencia. La administración (el cambio de un hilo a otro, la sincronización entre los hilos, etc.) de los hilos la realiza la aplicación por medio de librería de hilos; no requiere de los privilegios propios del modo kernel(Vahalia (1996); Northup (1996)).

Ventajas de usar ULT

- El cambio de un hilo a otro no involucra al kernel.
- La sincronización es realizada por la aplicación.
- ULTs pueden correr en cualquier sistema operativo - sólo necesita las librerías de hilos.

Desventajas de usar ULT:

- *Las llamadas al sistema bloquean el proceso* - entonces todos los hilos dentro del proceso serán bloqueados
- El kernel sólo puede asignar el microprocesador a procesos - dos hilos dentro del mismo proceso no pueden correr simultáneamente en dos microprocesadores

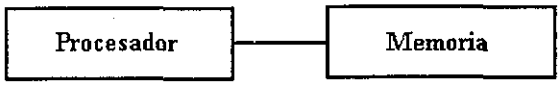
2.5 Procesamiento en paralelo

Tradicionalmente los programas han sido escritos para ser ejecutados de manera secuencial; para ser ejecutados por una única computadora que contiene un sólo microprocesador.

Los problemas son resueltos por una serie de instrucciones, ejecutadas una tras otra por el microprocesador. Sólo una instrucción puede ser ejecutada en un momento en el tiempo.

Por más de 40 años, las computadoras han seguido un modelo introducido por John von Neumann(www.llnl.gov/computing/tutorials/workshops/workshop). Como se muestra en la figura 2.9, éste modelo consiste de una Unidad Central de Procesamiento, o microprocesador, y memoria (Kumer (1994); www.llnl.gov/computing/tutorials/workshops/workshop).

- El microprocesador ejecuta las instrucciones del programa secuencialmente.
- La memoria se utiliza para almacenar los datos e instrucciones del programa.



Modelo de John von Neumann
Figura 2.9

La velocidad de computo de estas computadoras está limitada por dos factores:
1) La velocidad promedio de ejecución de las instrucciones.
2) La velocidad de intercambio de información entre la memoria y el microprocesador.

Una manera alternativa de incrementar la velocidad de ejecución de las instrucciones de un programa es usar múltiples microprocesadores y unidades de memoria interconectadas(Kumer (1994)).

La velocidad de ejecución de instrucciones aumenta cuando el número de microprocesadores y unidades de memoria se incrementa(Kumer (1994)).

El procesamiento paralelo se refiere a la división de un programa en múltiples fragmentos que pueden ser ejecutados simultáneamente, por diferentes microprocesadores(www.llnl.gov/computing/tutorials/workshops/workshop).

Un programa que se ejecuta por n microprocesadores puede ser ejecutado hasta n veces más rápido que si usara un sólo microprocesador(www.llnl.gov/computing/tutorials/workshops/workshop).

¿Es el procesamiento paralelo lo que quiero?
Aunque varios microprocesadores pueden aumentar la velocidad de ejecución de un programa muchas aplicaciones no pueden beneficiarse con el procesamiento paralelo.

Básicamente el procesamiento paralelo es apropiado si:

- Las aplicación tiene suficiente paralelismo para hacer uso de múltiples microprocesadores. En parte ésto involucra la identificación de las partes del programa que pueden ser ejecutadas independientemente y simultáneamente por varios microprocesadores separados pero también se encontrará que algunas cosas que pueden ser ejecutadas en paralelo pueden ser aun todavía lentas si se ejecutan en un sistema en particular. Por ejemplo, un programa puede ser ejecutado en cuatro segundos en una computadora con un único microprocesador y puede ser ejecutado en un segundo por una computadora con cuatro microprocesadores pero no aumenta la velocidad por que ésta computadora toma tres segundos mas para coordinar las acciones.
- La aplicación ha sido ya paralelizada (reescrita para tomar ventaja del procesamiento en paralelo) o se está conciente de volver a codificar la aplicación para tomar ventaja del procesamiento en paralelo.
- Estamos interesados en la investigación, o por lo menos en familiarizarnos con el procesamiento paralelo.

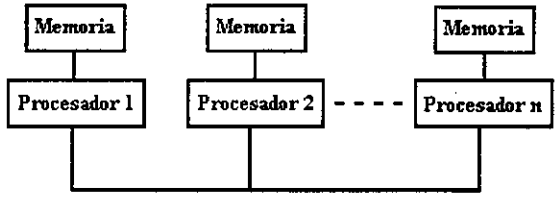
2.6 Arquitectura de las computadoras en paralelo

Una computadora en paralelo es una que tiene un conjunto de microprocesadores capaces de trabajar conjuntamente para resolver un problema computacional. Pero estos microprocesadores tienen que comunicarse y además guardar los programas y datos en memoria. De esta manera, para construir una computadora en paralelo existen dos arquitecturas básicas:

1) Arquitectura de memoria distribuida

La memoria es asignada físicamente a cada microprocesador (vea la figura 2.10); cada microprocesador tiene derecho a acceder a su propia memoria; la comunicación se realiza moviendo los datos entre los microprocesadores.

- Los microprocesadores tienen su propia memoria local.
- Como cada microprocesador tiene su propia memoria, éstos operan independientemente. Los cambios que realiza en su memoria no tienen efecto sobre la memoria de otros microprocesadores.
- Cuando un microprocesador necesita accesar la memoria de otro es responsabilidad del programador especificar cómo se comunicarán.

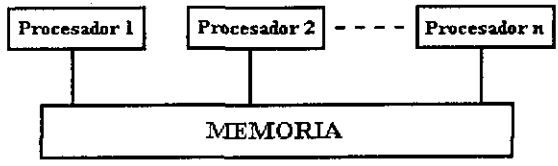


Arquitectura de memoria distribuida
Figura 2.10

2) Arquitectura de memoria compartida

La memoria compartida se piensa como un sólo bloque, en donde todos los datos tendrán asignada una dirección única dentro del bloque y todos los microprocesadores tienen acceso al bloque de memoria(Kumer (1994); www.llnl.gov/computing/tutorials/workshops/workshop), (vea la figura 2.11).

- Todos los microprocesadores pueden accesar la memoria como un espacio global.
- Los microprocesadores pueden operar independientemente.
- Los cambios en la memoria hechos por un microprocesador, son visibles por los otros microprocesadores.



Aquitectura de memoria compartida
Figura 2.11

Las computadoras que comparten la memoria se pueden dividir en dos tipos de acuerdo al tiempo de acceso a la memoria(Kumer (1994); www.llnl.gov/computing/tutorials/workshops/workshop):

1. Memoria de Acceso no-Uniforme o NUMA (*Non-Uniform Memory Access*).
2. Memoria de Acceso Uniforme o UMA (*Uniform Memory Access*).

Memoria de Acceso no-Uniforme

Las características de éste tipo de arquitectura son:

- Idénticos o diferentes microprocesadores.
- No todos los microprocesadores tienen el mismo tiempo de acceso a la memoria.
- El acceso a la memoria es más lento por medio de la red.

Representada por computadoras en red.

Memoria de Acceso uniforme

Este tipo de arquitectura tiene las siguientes características:

- Idénticos microprocesadores.
- Tiempo de acceso a la memoria igual para cada microprocesador.

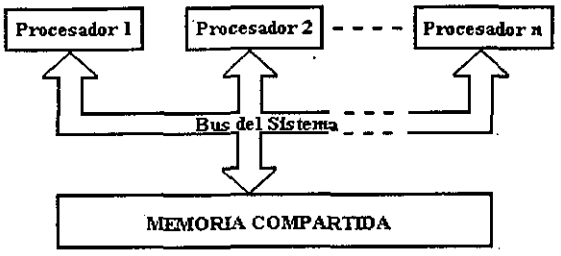
Representada por las computadoras SMP (*Symmetric Multiprocessing*).

2.7 Multiprocesamiento simétrico

Multiprocesamiento simétrico (*symmetric multiprocessing* o SMP) es uno de los diseños de procesamiento paralelo más maduro(www.llnl.gov/computing/tutorials/workshops/workshop). Apareció en las supercomputadoras Cray X-MP y en sistemas similares en 1983.

El SMP tiene un diseño simple pero efectivo.

En SMP múltiples microprocesadores que comparten la memoria y el bus del sistema como se muestra en la figura 2.12. Este diseño es también conocido como estrechamente acoplado (*tightly coupled*) o compartiendo todo (*shared everything*).



Multiprocesamiento simétrico
Figura 2.12

Debido a que SMP comparte globalmente la memoria, tiene solamente un espacio de memoria, lo que simplifica tanto el sistema físico como la programación de aplicaciones.

La memoria globalmente compartida también vuelve fácil la sincronización y comunicación. Sin embargo, esta memoria global contribuye al problema más grande de SMP, conforme se añaden microprocesadores, el tráfico en el bus de memoria se satura(www.llnl.gov/computing/tutorials/workshops/workshop).

SMP es considerado una tecnología no escalable. En éste modelo sólo una copia del sistema operativo existe en la memoria.

2.8 Taxonomía de Flynn

Hay muchas maneras de construir una computadora en paralelo y también hay diferentes maneras de clasificarlas. La manera más usada es la clasificación en uso desde 1966, llamada Taxonomía de Flynn(Kumer (1994); www.llnl.gov/computing/tutorials/workshops/workshop), la cual se basa en la manera en como la computadora en paralelo ejecutará un programa, como se muestra en la figura 2.13.

Flujo de Instrucciones	Flujo de Datos: Único	Flujo de Datos: Múltiple
Único	**SISD** Único Flujo de Instrucciones Único Flujo de Datos	**SIMD** Único Flujo de Instrucciones Múltiple Flujo de Datos
Múltiple	**MISD** Múltiple Flujo de Instrucciones Único Flujo de Datos	**MIMD** Múltiple Flujo de Instrucciones Múltiple Flujo de Datos

Taxonomía de Flynn
Figura 2.13

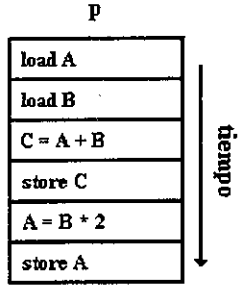
Flynn propuso cuatro tipos de procesamientos los cuales son:

1) Único flujo de instrucciones - Único flujo de datos (SISD o *Single Instruction - Single Data*)

Este es el tipo de procesamiento más antiguo, y aún es uno de los más importantes pues todas las computadoras personales y la mayor parte de los diseños de las computadoras actuales se encuentran en ésta categoría.

- Una computadora serial(no-paralela). Único flujo de instrucciones: sólo un flujo de instrucciones puede ser ejecutado por el microprocesador durante cualquier instante, como se muestra en la figura 2.14.
- Único flujo de datos: sólo un flujo de datos será usado como entrada en cualquier instante.
- Ejecución determinista, lo cual significa que se conoce el estado de la instrucción y además podemos repetir el proceso. Porque cada instrucción tiene un único lugar de ejecución dentro del flujo de instrucciones y un tiempo asignado mientras es procesado por el microprocesador.

Ejemplos:
PC´s, estaciones de trabajo con un único microprocesador.



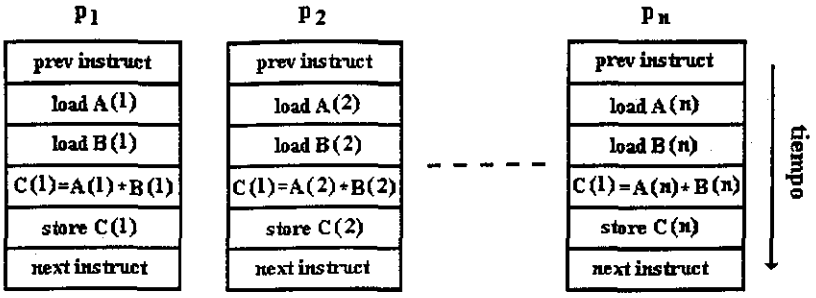
Único flujo de instrucciones - Único flujo de datos
El microprocesador P ejecuta un programa
Figura 2.14

2) Único flujo de instrucciones - Múltiple flujo de datos (SIMD o *Single Instruction - Múltiple Data*)

Este es un tipo de categoría muy importante en la historia de la computación, las computadoras del tipo SIMD son capaces de aplicar el mismo flujo de instrucciones a múltiples flujos de datos simultáneamente. Como se muestra en la figura 2.3.

- Es un tipo de computadoras en paralelo.
- Único flujo de instrucciones: todos los Microprocesadores ejecutan la misma instrucción en cualquier instante.
- Múltiple flujo de datos: cada microprocesador puede operar en diferentes elementos de datos.
- Ejecución determinista.

Ejemplos:
Processor Arrays: Connection Machine CM-2, Maspar MP-1, MP-2.
Vector Pipelines: IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820.



Único flujo de instrucciones - Múltiple flujo de datos
Los microprocesadores P1, P2 ... Pn ejecutan el mismo programa
Figura 2.15

3) Mutiple flujo de instrucciones - Único flujo de datos (MISD o *Múltiple Instruction - Single Data*)

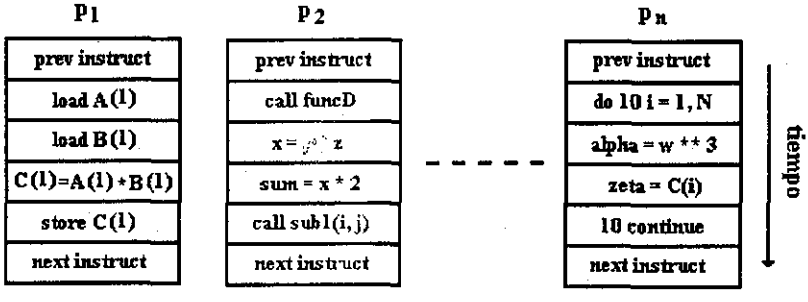
Esta categoría se considera más para completar nuestra clasificación que como tipo de procesamiento paralelo.

4) Múltiple flujo de instrucciones - Múltiple flujo de datos (MIMD o *Múltiple Instruction - Múltiple Data*)

Esta es la categoría más general; proporciona para múltiples flujos de instrucciones múltiples flujo de datos, como se muestra en la figura 2.16. Las computadoras MIMD pueden ser programadas para trabajar como cualquiera de *los tipos anteriores*.

- Normalmente, el tipo más común de computadora en paralelo.
- Múltiple flujo de instrucciones: cada microprocesador puede ejecutar un flujo de instrucciones diferente.
- Múltiple flujo de datos: cada microprocesador puede estar trabajando con diferentes flujos de datos.
- La ejecución puede ser determinista o no determinista.

Ejemplos:
Supercomputadoras, computadoras SMP.



Múltiple flujo de instrucciones - Múltiple flujo de datos
Los microprocesadores P1, P2 ... Pn ejecutan programas diferentes
Figura 2.16

Múltiple flujo de instrucciones o único programa

Los sistemas MIMD son capaces de correr múltiple flujo de instrucciones, con cada microprocesador haciendo algo diferente o cada microprocesador ejecutando el mismo flujo de instrucciones; a este último caso se le llama SPMD (único programa - múltiple flujo de datos) (www.llnl.gov/computing/tutorials/workshops/workshop).

2.9 Modelos de programación en paralelo

Hay varios modelos de programación en paralelo(www.llnl.gov/computing/tutorials/workshops/workshop), algunos son:

- Modelo MPI (*Message Passing Interface*).
- Modelo de Hilos (*Threads*).

Qué modelo usar es a menudo una combinación de lo que está disponible y de una elección personal. No hay un mejor modelo, aunque ciertas implementaciones en algunos modelos son mejores que otras, dependiendo del problema.

Modelo MPI (*Message Passing Interface*)

El modelo MPI tiene las siguientes características:

- El conjunto de tareas tienen su propia memoria local, mientras se ejecutan. Múltiples tareas pueden residir en la misma computadora así como también en diferentes computadoras.
- Las tareas intercambian datos por medio de mensajes.

Modelo de hilos

El modelo de hilos tiene las siguientes características:

- En el modelo de programación en paralelo por medio de hilos, un proceso puede tener múltiples hilos de ejecución.
- Cada hilo tiene sus datos locales, pero también, comparte los recursos del sistema asignados al proceso.
- La comunicación entre hilos se hace por medio de la memoria compartida. Esto requiere de la sincronización entre los hilos para asegurar que no más de un hilo actualice la misma dirección de memoria al mismo tiempo.
- Los hilos son comúnmente asociados con la arquitectura de memoria compartida.

2.10 Modelos de programación en paralelo

Algunas implementaciones del modelo de hilos son(www.llnl.gov/computing/tutorials/workshops/workshop):

1) Hilos POSIX
   - Se basa en librerías conocidos comúnmente como Pthreads o hilos POSIX.
   - Se utiliza el lenguaje C o C++.

2) Librerías OpenMP
   - Compilador basado en las librerías OpenMP; usa código escrito serialmente en lenguaje C o C++.
   - Portable/Multiplataforma, incluye las plataformas LINUX y Windows NT.

° Es muy fácil de usarlo.

2.11 Observaciones

En la presente tesis se utiliza una computadora en paralelo con dos microprocesadores idénticos para ejecutar el algoritmo de dimerización paralelizado.

La arquitectura de esta computadora en paralelo es la de memoria compartida, con acceso de memoria uniforme, es decir, soporta el multiprocesamiento simétrico o SPM.

Uno de los sistemas operativos que soporta SMP es el sistema operativo Linux Mandrake con un *kernel* versión 2.2.15-4mdksmp.

La manera en como se ejecutará el programa es **Único Flujo de Instrucciones - Múltiple Flujo de Datos** (SIMD). Es decir, el programa se ejecutará en ambos microprocesadores (pero se puede programar de tal manera que cada microprocesador ejecute un bloque de código diferente del programa).

Las herramientas que se utilizarán para desarrollar el algoritmo de dimerización en paralelo son las librerías OpenMP; estas librerías nos permite crear programas que puedan trabajar en los diferentes tipos de ejecución (SISD, SIMD o MIMD). En el siguiente capítulo se explican los conceptos fundamentales de las librerías OpenMP.

3

Compilador OpenMP

En esté capítulo se explican los conceptos fundamentales relacionados con el compilador OpenMP, el cual nos va a permitir paralelizar cualquier programa secuencial escrito en C o C++.

Se explican las diferentes maneras de ejecutar un programa por varios hilos, llamadas construcciones de trabajo compartido. También se explica la manera de sincronizar los hilos que están ejecutando el programa.

Y por último, se explican las librerías de funciones y las variables de ambiente que se pueden utilizar en la paralelización de un programa secuencial.

En esté trabajo se usó el compilador OpenMP de Pórtland Group, adquirido por el proyecto PAPIIT UNAM No. IN101598 (investigador responsable Dr. Vladimir Tchijov).

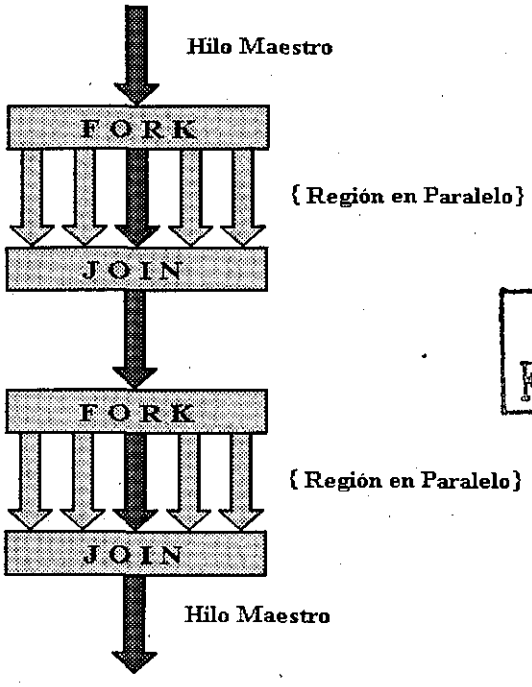
3.1 Hilos OpenMP

OpenMP es un modelo de programación explícito que ofrece al programador un control total sobre la paralelización del código de un programa.

Modelo Fork-Join:

OpenMP utiliza el modelo de ejecución Fork-Join. En este modelo un hilo maestro crea un conjunto de hilos conforme se necesita. El código secuencial que se desea paralelizar se encuentra envuelto por una región en paralelo.

- Todos los programas en OpenMP empiezan como un proceso o hilo maestro. El hilo maestro se ejecuta secuencialmente hasta que encuentra la primera región en paralelo(vea la figura 3.1).
- Fork: el hilo maestro crea un conjunto de hilos. Las instrucciones que se encuentran dentro de la región en paralelo son ejecutadas por cada uno de los hilos del conjunto(vea la figura 3.1).
- Join: Cuando el conjunto de hilos termina de ejecutar las instrucciones de la región en paralelo, los hilos se sincronizan y terminan, continuando sólo el hilo maestro(ver la figura 3.1).



Modelo Fork - Join
Figura 3.1

OpenMP es un compilador que contiene un conjunto de directivas, librerías y variables de ambiente para la creación de programas multi-hilos (*multi-threaded* o MT)en C y C++.

OpenMP tiene tres componentes:

- Conjunto de directivas
  I. Sintaxis de las directivas en C y C++
  II. Construcción de regiones en paralelo
  III. Construcción de trabajo compartido
  IV. Datos de ambiente
  V. Construcción de sincronización
- Librerías para asignar y verificar los atributos de los hilos.
- Variables de ambiente para controlar el comportamiento de los programas en paralelo al tiempo de ejecución.

3.2 Conjunto de directivas

I. Sintaxis de las directivas en C y C++

```
#pragma omp nombre_directiva [cláusula, ...] nueva_línea
```

#pragma omp	todas las directivas empiezan con **#pragma omp** en C y C++.
nombre_directiva	directivas validas en OpenMP deben aparecer después de **#pragma omp** y antes de cualquier cláusula.
[cláusula, ...]	es opcional. Las cláusulas pueden estár en cualquier orden y repetirse cuando sea necesario a menos que haya alguna restricción.
nueva_línea	requerida. Antecede el bloque de instrucciones.

Ejemplo:

```
#pragma omp parallel default(shared) private(beta,pi)
```

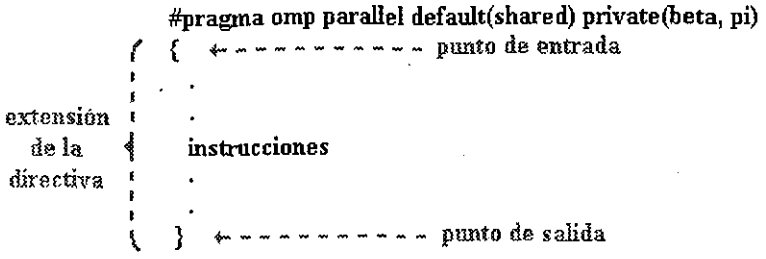
Reglas generales para las directivas:

- Las directivas siguen las convenciones estándar para compiladores C y C++.
- Las directivas son sensibles al tipo de letra.
- Sólo un **nombre_directiva** puede ser especificada por directiva.
- Las directivas se aplican cuando mucho a un bloque de instrucciones.
- Si las directivas ocupan más de un renglón, estas pueden continuar en el renglón siguiente utilizando la diagonal invertida ("\") al final de la línea.

Las directivas pueden extenderse, aplicándose a un bloque de instrucciones, como se muestra en la figura 3.2.

El bloque de instrucciones es un flujo de control que satisface lo siguiente:

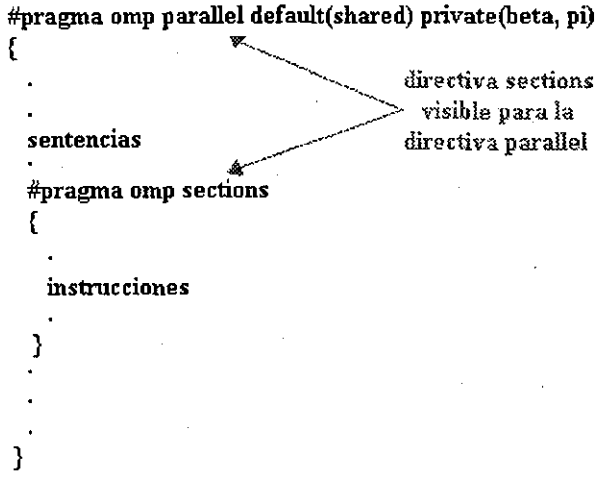
- Sólo existe un punto de entrada en el bloque de instrucciones; el inicio del bloque(ver la figura 3.2).
- Sólo existe un punto de salida en el bloque de instrucciones; el final del bloque(vea la figura 3.2).



Extensión de la directiva
Figura 3.2

Alcance de las directivas

1. Extensión estática o extensión léxica son las directivas visibles dentro de otra directiva(vea la figura 3.3).
   - Las directivas son válidas dentro del bloque de instrucciones.
   - La extensión estática de una directiva no es valida en otras funciones o archivos.



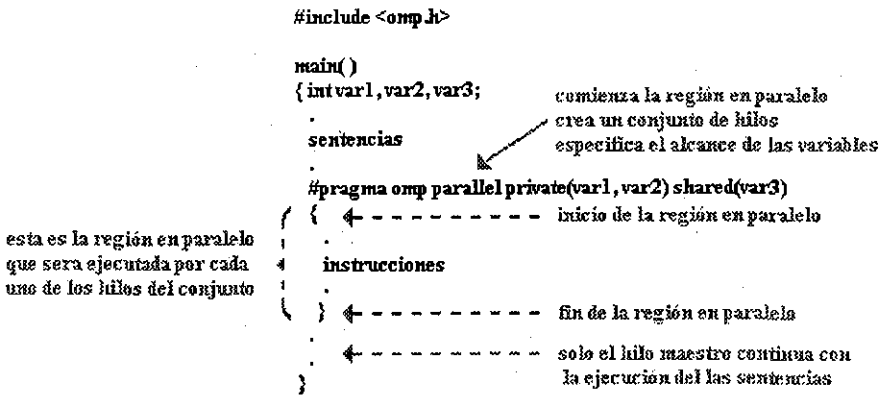
Extensión estática o extensión léxica
Figura 3.3

2. Directivas huérfanas. Las directivas que aparecen independientes en otra función o archivo, pero no en su extensión estática, se dice que son directivas huérfanas.
   - Estas directivas existen fuera de una extensión estática.
   - Estas directivas se pueden encontrar en otras funciones o archivos.
   - Las directivas de trabajo compartido pueden ser huérfanas. Esto permite que las construcciones de trabajo compartido puedan ocurrir en una función que puede ser llamada por el código serial o código paralelo.

3. Extensiones dinámicas, la extensión dinámica de una directiva incluye ambas extensiones estáticas y huérfanas de una directiva.

II. Construcción de regiones en paralelo

La región en paralelo es la manera de crear múltiples hilos en un programa por medio de las directivas OpenMP. Esta región en paralelo es un bloque de instrucciones que será ejecutado por múltiples hilos(vea la figura 3.4).



Estructura del código en OpenMP:
Figura 3.4

Esta es la construcción básica en OpenMP con las siguientes características:

- Un conjunto de hilos es creado al tiempo de ejecución para una región en paralelo. El hilo maestro es miembro del conjunto de hilos y es el hilo número 0 dentro del conjunto de hilos.
- Las instrucciones que se encuentra dentro de la región en paralelo se duplican y todos los hilos ejecutaran las mismas instrucciones.
- Existe una sincronización implícita al final de la región en paralelo. Sólo el hilo maestro continua la ejecución del código secuencial pasando el punto final de la región en paralelo.

Sintaxis:

```
#pragma omp parallel [cláusula ...]  nueva_linea
                                     if (scalar_expression)
                                     private (lista)
                                     shared (lista)
                                     default (shared | none)
                                     firstprivate (lista)
                                     reduction (operador: lista)
                                     copyin (lista)
{
   .
   instrucciones
   .
```

)

Cláusulas:

- cláusula IF: si el if está presente, esta se debe evaluar para decidir si el conjunto de hilos se creara o no.
- Las cláusulas restantes se describen en el resto del capítulo.

Restricciones:

- Una región en paralelo debe estár contenida en un bloque de instrucciones; no puede encontrarse en diferentes funciones o archivos.
- Es ilegal tratar de bifurcar el código hacia adentro o hacia afuera de la región en paralelo.
- Sólo se permite una cláusula IF.

¿Cuántos hilos se crean?

El número de hilos que se crea para la región en paralelo es determinado por los siguientes factores:

1. Implementación por default.
2. Valor de la variable de ambiente OMP_NUM_THREADS.
3. Uso de la función omp_set_num_threads() para fijar el número de hilos.
4. Los hilos son numerados desde 0 (para el hilo maestro) a N-1.

Hilos dinámicos

Por default, un programa con múltiples regiones en paralelo usara el mismo número de hilos para ejecutar la región en paralelo. Este comportamiento se puede cambiar al tiempo de ejecución para ajustar el número de hilos que será creado para cada región en paralelo.
Hay dos métodos para crear los hilos de una manera dinámica:

1. Usar la función omp_set_dynamic().
2. Asignar un valor a la variable de ambiente OMP_DYNAMIC.

Regiones en paralelo anidadas

- Una región en paralelo anidada dentro de otra región en paralelo resulta en la creación de un nuevo conjunto de hilos. Por default, consiste de un hilo.
- Algunas implementaciones permiten más de un hilo en las regiones en paralelo anidadas utilizando la variable de ambiente OMP_NESTED o la función omp_set_nested(int nested)

III. Construcciones de trabajo compartido

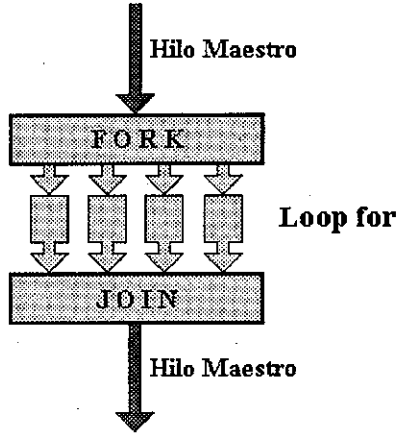
Las construcciones de trabajo compartido ejecutan la región en paralelo dividiendo esta región entre los miembros del conjunto de hilos.

- Las construcciones de trabajo compartido no lanzan nuevos hilos.
- En las construcciones de trabajo compartido no existe una sincronización entre hilos al inicio de la región en paralelo, aunque existe una sincronización implícita al final de la construcción trabajo compartido.
- Por default, existe una sincronización implícita al final de la región en paralelo. Al usar la cláusula "nowait" deshabilita la sincronización implícita de la región en paralelo.

Existen tres tipos de construcciones de trabajo compartido:
1) Trabajo compartido con la directiva **for**.
2) Trabajo compartido con la directiva **sections**.
3) Trabajo compartido con la directiva **single**.

Trabajo compartido directiva for

La directiva **for** especifica que el número de interacciones del loop **for** deben ser repartidas entre los hilos del conjunto de hilos de una manera equitativa, como se muestra en la figura 3.5.



Directiva **for**
Figura 3.5

Esto supone que la región en paralelo ya ha sido iniciada, de otra manera el código se ejecuta de una manera secuencial.

Sintaxis:

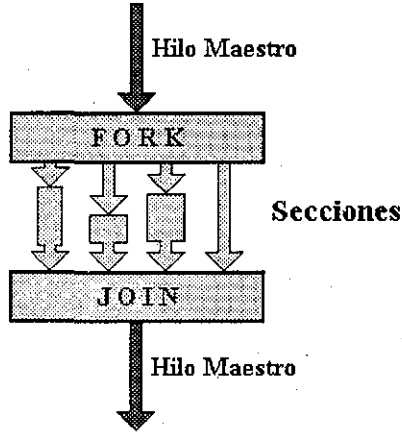
```
#pragma omp for [cláusula ...]  nueva_línea
                                schedule (type [,chunk])
                                ordered
                                private (lista)
                                firstprivate (lista)
                                lastprivate (lista)
                                shared (lista)
                                reduction (operador: lista)
                                nowait

{
    instrucciones
}
```

Restricciones:

- La variable que controla al loop debe ser un entero y debe ser la misma para todos los hilos.
- El programa no debe depender de la ejecución de una iteración en particular por un hilo.
- Es ilegal bifurcar el código hacia adentro o hacia afuera asociado con la directiva **for.**
- La variable **chunk** debe ser especificada como un entero invariable, no hay manera de sincronizar durante la evaluación por los diferentes hilos.
- En C y C++ la directiva **for** requiere que el loop **for** esté en forma canónica.
- Las cláusulas **ORDERED** y **SCHEDULE** sólo pueden aparecer una ves.

Trabajo compartido directiva **sections**

La directiva **SECTIONS** es una construcción no-iterativa de trabajo compartido. Esta especifica que lo que está encerrado por las directivas **section(s)** será ejecutado por un hilo del conjunto de hilos; si hay varias secciones cada hilo del conjunto tomara una sección diferente para ejecutarla, como se muestra en la figura 3.6.



Directiva **sections**
Figura 3.6

Las directivas **SECTION** independientes se anidan dentro de la directiva **SECTIONS,** cada **SECTION** es ejecutada sólo una ves por un hilo del conjunto. Diferentes **section** serán ejecutadas por diferentes hilos.

Sintaxis:

```
#pragma omp sections [clausula ...]  nueva_línea
                                     private (lista)
                                     firstprivate (lista)
                                     lastprivate (lista)
                                     reduction (operador: lista)
                                     nowait
{
  #pragma omp section   nueva_linea
```

```
  bloque_ de_ instrucciones

  #pragma omp section   nueva_línea
  bloque_ de_ instrucciones

 }
```

Cláusulas:

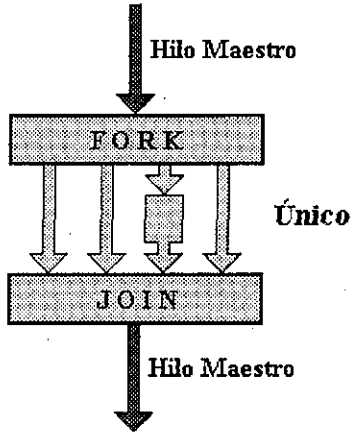
- Hay una sincronización implícita al final de la directiva **SECTIONS**, a menos que se utilice la cláusula **nowait**.
- Las cláusulas se describen más adelante, en este capítulo.

Restricciones:

- Es ilegal bifurcar el código hacia adentro o hacia afuera asociado con la directiva **section**.
- La directiva **section** se debe encontrar dentro de la extensión léxica de la directiva **sections**.

Trabajo compartido directiva **SINGLE**

La directiva **SINGLE** especifica que el bloque de instrucciones será ejecutado por sólo un hilo del conjunto, como se muestra en la figura 3.7.



Directiva **single**
Figura 3.7

Es de gran ayuda cuando se trabaja con secciones de código que no soportan los hilos.

Sintaxis:

```
#pragma omp single [cláusula ...]  nueva_línea
                                   private (lista)
                                   firstprivate (lista)
                                   nowait
```

```
{
   .
   instrucciones
   .
 }
```

Cláusulas:

° Los hilos del conjunto que no están ejecutando la directiva **SINGLE**, esperan que termine el hilo que está ejecutando la directiva **SINGLE** para continuar, a menos que se use la cláusula **nowait**.

Las cláusulas se describen más adelante, en este capítulo.

Restricciones:

° Es ilegal bifurcar el código hacia adentro o hacia afuera asociado con la directiva **SINGLE**.

Trabajo compartido combinación de la directiva **parallel** y **for**

La directiva **parallel for** especifica una región en paralelo que contiene sola una directiva **for**. La directiva **for** debe seguir en la siguiente línea de código.

Sintaxis:

```
#pragma omp parallel for [cláusula ...]  nueva_línea
                                         if (scalar_logical_expression)
                                         default (shared | none)
                                         schedule (type [,chunk])
                                         shared (lista)
                                         private (lista)
                                         firstprivate (lista)
                                         lastprivate (lista)
                                         reduction (operador: lista)
                                         copyin (lista)

{
   .
   instrucciones
   .
 }
```

Cláusulas:

° Las cláusulas pueden ser cualquiera que acepta la directiva **for**. Las cláusulas que aun no se describen, se describen más adelante en el capítulo.

Trabajo compartido combinación de la directiva **parallel** y **sections**

La directiva **parallel sections** especifica una región en paralelo contiene una sola directiva **SECTIONS**. La directiva **SECTIONS** debe seguir a cualquier línea de código.

Sintaxis:

```
#pragma omp parallel sections [clausula ...]  nueva_linea
```

```
                                        default (shared | none)
                                        shared (lista)
                                        private (lista)
                                        firstprivate (lista)
                                        lastprivate (lista)
                                        reduction (operador: lista)
                                        copyin (lista)
                                        ordered

{
   .
   instrucciones
   .
 }
```

Cláusulas:

° Las cláusulas que acepta son las cláusulas previamente vistas para la directiva **SECTIONS**. Las cláusulas que aun no se describen, se describen más adelante en el capítulo.

Cláusulas

cláusula schedule

La cláusula **schedule** controla la manera en como las iteraciones serán distribuidas en los hilos.

schedule(static [chunk])
Reparte las iteraciones en bloques de tamaño "**chunk**" para cada hilo.

schedule(dynamic[chunk])
Cada hilo ejecuta un número de iteraciones del tamaño "**chunk**" y si termina toma otro número de iteraciones hasta que terminan las iteraciones.

schedule(guided[chunk])
Los hilos toman dinámicamente bloques de iteraciones. El tamaño de los bloques empiezan grandes y se contraen hasta el tamaño "**chunk**" conforme avanza el cálculo.

schedule(runtime)
El **schedule** y el tamaño de el "**chunk**" se toma de la variable de ambiente **OMP_SCHEDULE**.

IV. Datos de ambiente

Como OpenMP se basa en el modelo de programación de memoria compartida, muchas de sus variables son compartidas por default. Es importante entender el alcance de las variables en OpenMP.

Las variables globales incluyen:

° Variables que tienes un alcance de archivo, y variables estáticas.

Las variables privadas incluyen:

° Las variables de control de los loops.

° Las variables de funciones que se almacenan en el stack llamadas desde las regiones en paralelo.

° Variables automáticas dentro de un bloque de instrucciones son privadas.

Las cláusulas para cambiar el alcance de las variables son:
° **private()**
° **firstprivate()**
° **lastprivate()**
° **shared()**
° **default()**
° **reduction()**
° **copyin()**

Las cláusulas para cambiar el alcance de las variables se usan junto con algunas directivas (**parallel, for** y **sections**).

Las cláusulas proporcionan la habilidad para controlar el alcance de las variables durante la ejecución de la región en paralelo.
° Las cláusulas definen como y que variables en el código secuencial del programa será transferido a la región en paralelo del programa.
° Las cláusulas definen que variables serán visibles a todos los hilos en la región en paralelo y cuales serán asignadas de una manera privada a todos los hilos.

Todas las cláusulas sólo se aplican a las extensiones estáticas del las construcciones OpenMP.

Todas la cláusulas se aplican a la región en paralelo y a las construcciones de trabajo compartido excepto para la construcción "**shared**" que sólo se aplica a las regiones en paralelo.

Nota: Las cláusulas para cambiar el alcance de las variables sólo existen en la extensión estática.

cláusula **private()**
La cláusula **private()** declara las variables en su lista como privadas para cada hilo.

Sintaxis:
private(lista)

Notas:
Las variables privadas se comportan como sigue:
° **private**(variable) crea una copia local de la variable original para cada hilo del conjunto.
° Todas las referencias a la variable original son remplazadas con referencias a la nueva variable.
° La copia de la variable privada no está asociada con la variable original.
° Las variables declaradas **private()** no son inicializadas para los hilos.

cláusula **shared()**
La cláusula **shared()** declara las variables en su lista como compartidas entre los hilos del conjunto.

Sintaxis:
shared(lista)

Notas:
- Una variable compartida existe sólo una ves en la memoria y cualquier hilo puede leer o escribir sobre esta variable.
- Es responsabilidad del programador asegurarse que los hilos utilicen la variable de manera adecuada.

Cláusula **default()**
La cláusula **default()** permite al usuario especificar el alcance de todas las variables por default como privadas (**private**), compartidas (**shared**) o ninguno (**none**) dentro de la región en paralelo.

Sintaxis:
default(shared | none)

Restricciones:
- Sólo una cláusula **default()** puede especificarse por directiva **parallel**.

Note que el atributo por default de las variables es **default(shared)** entonces no es necesario especificarla.

Para cambiar el atributo por default: **default(private)**
Cada variable dentro del bloque de código de la región en paralelo se hace privada como si se hubiera especificada privada con la cláusula **private()**.

default(none): no hay especificación por default dentro de la región en paralelo. Se deben especificar los atributos para cada variable dentro de la región en paralelo.

Cláusula **firstprivate()**
La cláusula **firstprivate()** combina el comportamiento de la cláusula **private()** con la inicialización automática de las variables en su lista.

Sintaxis:
firstprivate(lista)

Nota:
- Las variables listadas son inicializadas de acuerdo al valor de su variable original antes de entrar a la región en paralelo o alguna construcción de trabajo compartido.

Cláusula **lastprivate()**
La cláusula **lastprivate()** combina el comportamiento de la cláusula **private()** con una copia de la última iteración de el loop con la variable original.

Sintaxis:
lastprivate(lista)

Nota:
- El valor de la copia regresa dentro de la variable original que se obtiene desde el última iteración de la construcción. Por ejemplo, el hilo del

conjunto que ejecuta la última iteración del loop realiza la copia con sus propios valores de las variables.

Cláusula **copyin()**
La cláusula **copyin()** proporciona el medio para asignar el mismo valor a las variables declaradas dentro de la cláusula threadprivate() para todos los hilos del conjunto.

Sintaxis:
copyin(lista)

Nota:
- La lista contiene los nombres de las variables en donde se copiara el valor inicialización.
- La variable de hilo maestro se usa como copia fuente. El conjunto de hilos son inicializados con el valor de la copia fuente.

Cláusula **reduction()**
La cláusula **reduction()** realiza una reducción sobre las variables que aparecen en su lista.

Se crea una copia privada de cada variable de la lista para cada hilo. Al final la variable a reducir se aplica a todas las copias privadas de las variables compartidas, y el resultado final es escrito en la variable global compartida.

Sintaxis:
reduction (operador: lista)

Restricciones:
- Las variables en la lista deben ser variables escalares. Ellas no pueden ser arreglos o estructuras. También deben ser declaradas como **shared** en el contexto.
- La operación reducción puede no ser asociativa para los números reales.
- La cláusula **reduction()** se debe usar en regiones en paralelo o trabajo compartido en donde se usa la variable de reducción; la cual se usa sólo en instrucciones de las siguiente manera:

```
x = x op expr
x = expr op x (except subtraction)
x binop = expr
x++
++x
x--
--x
x es una variable escalar de la lista
expr es una expresión escalar que no referencia a x
op no se sobrecarga, y es uno de +, *, -, /, &, ^, |, &&, ||
binop no se sobrecarga, y es uno de +, *, -, /, &, ^, |
```

Cláusula **threadprivate()**

La directiva **threadprivate()** se usa para convertir una variable global (o alcance de archivo) a una variable local para un hilo; sólo existe durante la ejecución de la región en paralelo.

Sintaxis:
threadprivate(lista)

Notas:
- La cláusula **threadprivate()** debe de aparecer después de la declaración de las variables globales listadas. Cada hilo tiene su propia copia de la variable global, entonces los datos escritos por un hilo no son visibles para los otros hilos.
- En la primera región en paralelo, los datos de las variables listadas dentro de la cláusula **threadprivate()** y los bloques comunes se deben *suponer indefinidos, a menos que la cláusula* ***copyin()*** *se especifique en la* directiva PARALLEL.

Restricciones:
Los datos de las variables declaradas en la cláusula **threadprivate()** se garantiza que existirán sólo si el mecanismo de los hilos dinámicos esta deshabilitado y el número de hilos en diferentes regiones en paralelo permanece constante.

Las variables dentro de la cláusula **threadprivate()** pueden ser inicializadas usando la cláusula **copyin()**.

V. Sincronización

Considere el siguiente ejemplo, donde dos hilos en diferentes procesadores tratan de incrementar la variable x al mismo tiempo (suponga que x esta inicializada a 0).

Una posible secuencia de ejecución:
1. Hilo 1 carga el valor de x dentro del registro A.
2. Hilo 2 carga el valor de x dentro del registro A.
3. Hilo 1 suma 1 al registro A.
4. Hilo 2 suma 1 al registro A.
5. Hilo 1 carga el registro A en la localidad x.
6. Hilo 2 carga el registro A en la localidad x.

El valor resultante de x será 1 y no 2 como debe ser.

Para evitar situaciones como está, el incremento de x debe ser sincronizado entre los dos hilo para asegurar que los resultados producidos sean correctos.

OpenMP proporciona una variedad de construcciones para sincronizar y controlar el acceso a las variables, estas son:
- **atomic**
- **critical**
- **barrier**
- **flush**
- **ordered**
- **master**

Nota: La sincronización es costosa, se recomienda cambiar el alcance de la variable para minimizar las sincronizaciones.

Directiva **atomic**
La directiva **atomic** es un caso especial de la directiva **critical**; se usa sólo cuando se va a sincronizar unicamente una instrucción. Se aplica sólo para actualizar una variable por un hilo a la ves.

Sintaxis:
```
#pragma omp atomic  nueva_línea
 instrucción
```

Restricciones:
- La directiva se aplica a una instrucción, la instrucción que sigue a la directiva atomic.
- La instrucción que sigue a la directiva atomic, puede tener alguna de las siguientes formas:

```
x binop = expr
x++
++x
x--
--x
```

x es una variable escalar
expr es una expresión escalar que no hace referencia a x
binop es un operador no sobrecargado, puede ser alguno de los siguientes: +, *, -, /,&, ^, |, >>, o <<

Nota: sólo se aplica la directiva atomic al cargado y almacenamiento de x; no la expresión que evalúa x.

Directiva **barrier**
La directiva **barrier** sincroniza todos los hilos del conjunto. Cuando la directiva barrier es alcanzada por un hilo, el hilo esperara en este punto hasta que todos los hilos alcancen la directiva **barrier**. Una ves que los hilos han alcanzado la directiva **barrier** estos ejecutan en paralelo la región de código que sigue a la directiva barrier.

Sintaxis:
```
#pragma omp barrier  nueva_línea
{
   .
   instrucciones
   .
 }
```

Para C y C++ la instrucción más pequeña a la que se le puede aplicar la directiva barrier es un bloque de instrucciones. Por ejemplo:

```
INCORRECTO              CORRECTO
If(x == 0)              If(x == 0)
```

```
#pragma omp barrier     {
{                         #pragma omp barrier
  .                       {
  instrucciones             .
  .                         instrucciones
}                           .
                          }

                        }
```

Existen algunos tipos de **barrier**´s implícitos, como se muestra en la figura 3.8. Cada hilo espera hasta que todos los hilos alcancen el punto final del bloque de instrucciones.

```
#pragma omp parallel shared (A, B, C) private(id)
{
  id=omp_get_thread_num();
  A[id] = big_calc1(id);

  #pragma omp barrier
  #pragma omp for
  for(i=0;i<N;i++)
     {
       C[i]=big_calc3(I,A);
     }                                  barrier implícito al final
                                        de la región en paralelo
  #pragma omp for nowait                barrier implícito desabilitado
  for(i=0;i<N;i++)                      por la clausula nowait
     {
       B[i]=big_calc2(C, i);
     }

  A[id] = big_calc3(id);

}                                       barrier implícito al final de la
                                        construcción de tabajo compartido
```

Ejemplo de barrier´s implícitos
Figura 3.8

Directiva **ordered**
La directiva ordered especifica que las iteraciones de un loop serán ejecutadas en el mismo orden como si ejecutaran de manera secuencial.

Sintaxis:

```
#pragma omp ordered  nueva_linea
{
   .
   instrucciones
   .
}
```

Restricciones:

° La directiva ordered sólo puede aparecer en la extensión dinámica de las siguientes directivas: **for** o **parallel for** en C/C++.

- Sólo un hilo tiene permitido ejecutar el bloque de instrucciones que precede la directiva ordered.
- El ilegal bifurcar hacia adentro o hacia afuera del bloque de instrucciones precedido por la directiva ordered.
- Una iteración de un loop no debe ejecutar la misma directiva **ordered** más de una ves, y no debe ejecutar más de una directiva **ordered.**
- El loop que contiene la directiva **ordered,** debe ser un loop con una cláusula **ordered.**

```
#pragma omp parallel private (tmp)
 #pragma omp for ordered
 for(I=0;I<N;I++)
    { tmp = NEAT_STUFF(I);

      #pragma ordered
      res = consum(tmp);
     }
```

Directiva **flush**
La directiva **flush** denota un punto de sincronización en donde la implementación del código debe proporcionar una vista consistente de las variables en la memoria a los hilos. Esto significa que evaluaciones previas de una expresión que hacen referencia a las variables en memoria no serán iniciadas hasta que todas las variables tomen los valores de la expresión ya evaluada.

Sintaxis:
```
#pragma omp flush(lista)  nueva_línea
```

Directiva **master**
La directiva **master** especifica una región que sólo será ejecutada por el hilo maestro del conjunto de hilos. Los otros hilos saltaran esta sección del código.

Sintaxis:
```
#pragma omp master nueva_línea
{
   .
   instrucciones
   .
 }
```

No hay barrier implícito asociado con esta directiva.

Directiva **critical**
La directiva **critical** especifica una región de código que será ejecutada sólo por un hilo a la ves.

Sintaxis:
```
#pragma omp critical [ name ]  nueva_línea
{
   .
   instrucciones
```

)

Si un hilo esta ejecutando la sección critica y otro hilo alcanza la sección critica y trata de ejecutarla, el hilo entra en un estado de bloqueado hasta que el primer hilo termina de ejecutar la sección critica.

La opción **name** habilita la existencia de múltiples regiones criticas:

- **name** actúa como un identificador global. Diferentes regiones criticas con el mismo nombre son tratadas como la misma región critica.
- Todas las regiones criticas que no tienen nombre, son tratadas como la misma región critica.

Restricciones:
Es ilegal bifurcar hacia adentro o hacia afuera las regiones criticas.

3.3 Librerías

Las funciones que se definen en OpenMP son de diferentes tipos:

- Funciones que nos permiten manipular el ambiente de ejecución en paralelo; por ejemplo: preguntar el número de hilos o procesadores, asignar el número de hilos que se usaran, etc.
- Funciones que nos permiten manipular los candados y los candados anidados.

Para C y C++, es necesario incluir el archivo de cabecera "omp.h".

Librería de funciones OpenMP

Manipulación de los hilos	omp_get_num_threads()
	omp_set_num_threads()
	omp_get_thread_num()
	omp_get_max_threads()
Información sobre el número de procesadores	omp_get_num_procs()
Manipulación de las regiones en paralelo	omp_in_parallel()
	omp_get_dynamic()
	omp_set_dynamic()
Manipular el anidamiento	omp_set_nested()
	omp_get_nested()
Manipular los candados	omp_init_lock()
	omp_destroy_lock()
	omp_set_lock()
	omp_unset_lock()
	omp_test_lock()
Manipular los candados anidados	omp_init_nested_lock()
	omp_destroy_nested_lock()
	omp_set_nested_lock()
	omp_unset_nested_lock()
	omp_test_nested_lock()

Para una mayor descripción de las funciones ver el Apéndice C.

3.4 Variables de ambiente

OpenMP proporciona cuatro variables de ambiente para controlar la ejecución en paralelo:
1. OMP_SCHEDULE
2. OMP_NUM_THREADS
3. OMP_DYNAMIC
4. OMP_NESTED

Los nombres de las variables de ambiente deben escribirse con letras mayúsculas. Los valores asignados a estas variables después de iniciar la ejecución del programa son ignorados.

OMP_SCHEDULE
La variable de ambiente **OMP_SCHEDULE** se aplica sólo a la directiva **parallel for** que tiene asignado **runtime** en la cláusula **schedule**. El valor de esta variable determina como las iteraciones de el loop serán ejecutadas por el procesador. Por ejemplo:

```
setenv OMP_SCHEDULE "guided, 4"
setenv OMP_SCHEDULE "dynamic"
```

Si la cláusula **schedule** tiene asignado un valor diferente de **runtime**, **OMP_SCHEDULE** será ignorada. El valor por default para esta cláusula es dependiente de la implementación.

OMP_NUM_THREADS
La variable de ambiente **OMP_NUM_THREADS** fija el número de hilos que se usarán durante la ejecución de una región en paralelo. Por ejemplo:

```
setenv OMP_NUM_THREADS 8
```

Su efecto depende si el ajuste dinámico de hilos está habilitado. Si el ajuste dinámico de hilos está deshabilitado, el valor de la variable **OMP_NUM_THREADS** es el número de hilos que se usará para ejecutar la región en paralelo; hasta que el valor de la variable es cambiado explícitamente ejecutando la función omp_set_num_threads().
Si el ajuste dinámico de hilos está habilitado, el valor de la variable **OMP_NUM_THREADS** es interpretado como el número máximo de hilos a ser usado.

El valor por default de la variable de ambiente OMP_NUM_THREADS es independiente de la implementación.

OMP_DYNAMIC
La variable de ambiente OMP_DYNAMIC habilita o deshabilita el ajuste dinámico del número de hilos disponibles para la ejecución de la región paralelo. Los valores validos para esta variable de ambiente son TRUE o FALSE. Por ejemplo:

```
setenv OMP_DYNAMIC TRUE
```

Sí la variable de habiente **OMP_DYNAMIC** tiene asignado el valor TRUE, el número de hilos que se usará para ejecutar la región en paralelo puede ser ajustado al tiempo de ejecución.

Sí el valor de la variable de ambiente es FALSE, el ajuste dinámico está deshabilitado. El valor por default de la variable de ambiente **OMP_DYNAMIC** es dependiente de la aplicación.

OMP_NESTED
La variable de ambiente OMP_NESTED habilita o deshabilita el paralelismo anidado. Los valores validos para esta variable de ambiente son TRUE o FALSE. Por ejemplo:

setenv OMP_NESTED TRUE

Sí la variable de ambiente OMP_NESTED tiene asignado el valor TRUE el paralelismo anidado ésta habilitado. Si la variable de ambiente tiene asignado el valor FALSE el paralelismo anidado esta deshabilitado.

El valor por default de la variable de ambiente OMP_NESTED es FALSE.

3.5 Observaciones

El compilador OpenMP se basa en la arquitectura multiprocesamiento simétrico o SMP.

El compilador OpenMP necesita que el código del programa esté escrito de una manera secuencial en lenguaje C o C++ antes de aplicar las directivas OpenMP para paralelizar el programa secuencial. En los Apéndices C y D se explican en detalle las funciones de la librerías OpenMP y además se dan algunos ejemplos de su aplicación.

El número de hilos que se utilizará para ejecutar nuestro programa es dos hilos, se recomienda que el número de hilos sea igual al número de microprocesadores de la computadora en paralelo(www.pgroup.com).

En el siguiente capítulo se explican los aspectos computacionales que se tomaron encuenta para codificar el algoritmo de dimerización en lenguaje C++. Además se explican y muestran los resultados de los experimentos realizados con el algoritmo de dimerización paralelizado.

4

Implementación del Algoritmo de Dimerización

En este capítulo se explican los aspectos computacionales (tales como la longitud de Alexandrowicz, el tipo de generador de números aleatorios y la manera de verificar las restricciones que deben cumplir las caminatas aleatorias) que se tomaron en cuenta para codificar el algoritmo de dimerización en lenguaje C++.

También se muestran los nuevos resultados que se obtuvieron al ejecutar el programa paralelizado con caminatas aleatorias auto-repelentes de hasta 20000 pasos.

4.1 Aspectos computacionales

Esencialmente, la simulación por computadora consiste en construir un programa que describe el comportamiento de un sistema real y después se procede a realizar experimentos con el programa o modelo.

Desde un punto de vista computacional, se presentan tres problemas al tratar de implementar el algoritmo de dimerización en lenguaje C++:

1. Escoger el generador de números aleatorios (*Random Number Generator* o RNG) apropiado.
2. Escoger el método para verificar si la caminata aleatoria satisface las restricciones geométricas impuestas.
3. Escoger el valor de la "longitud de Alexandrowicz" o N_A.

4.2 Escoger el generador de números aleatorios

Elegir un buen generador de números aleatorios o RNG juega un papel fundamental en la simulación por computadora.
En nuestros cálculos se utiliza el RNG **ran3**(Press (1988)) que ha probado ser un RNG confiable en la simulación de caminatas aleatorias auto-repelentes(Rodríguez-Romo, Tchijov V. (1998)).

Si hay alguna sospecha acerca de algún problema al experimentar con el modelo, se puede tratar con otros RNG. Existen diferentes RNG; para mayores referencias vea el libro "*Numerical Recipes in C*(Press W. H. *et al.*, 2nd Edition, 1997).

En algunos casos, para proporcionar un chequeo adicional a los resultados numéricos se utiliza el RNG de recorrido de registro de retroalimentación generalizada (*generalized feedback shift register*). Este RNG fue utilizado en la simulación de caminatas aleatorias auto-repelentes en 3 dimensiones y no se encontraron desviaciones significativas entre los resultados numéricos obtenidos por medio de este último RNG y **ran3**.

Las modificaciones que se hicieron en el código fuente del RNG **ran3** para ser ejecutado en paralelo se muestran a continuación:

```
.
.
TReal ran3(int &idum)
{
  static int inext,inextp;
  static long ma[56];
  static int iff=0;
  long mj,mk;
  int i,ii,k;

  if(idum < 0 || iff == 0)
    {
     #pragma omp critical
     {
       iff=1;
       mj=MSEED-(idum < 0 ? -idum : idum);
       mj%=MBIG;
       ma[55]=mj;
```

```
        mk=1;

        for(i=1; i<=54; i++)
           { ii=(21*i)%55;
             ma[ii]=mk;
             mk=mj-mk;
             if (mk < MZ) mk += MBIG;
             mj=ma[ii];
            }

        for(k=1; k<=4; k++)
             for(i=1; i<=55; i++)
                { ma[i] -= ma[1+(i+30) % 55];
                  if (ma[i] < MZ) ma[i] += MBIG;
                 }

        inext=0;
        inextp=31;
        idum=1;

       }
      }

  #pragma omp critical
  { if (++inext == 56) inext=1;
    if (++inextp == 56) inextp=1;
    mj=ma[inext]-ma[inextp];
    if (mj < MZ) mj += MBIG;
    ma[inext]=mj;
   }

  return mj*FAC;
}
```

Cuando se ejecuta el generador de números aleatorios **ran3** en una computadora con un sólo microprocesador, el RNG **ran3** se inicializa sólo una vez; a partir de ese momento se pueden generar números aleatorios. Para generar un número aleatorio el programa realiza varias operaciones sobre la matriz **ma**[].

Las correcciones consisten en aplicar la directiva ***#pragma omp critical*** para inicializar una sóla vez el RNG **ran3**.

Por último aplicamos, una vez más, la directiva ***#pragma omp critical*** para sincronizar los microprocesadores al manipular la matriz **ma**[].

4.3 Escoger el método para verificar si las caminatas aleatorias satisfacen las restricciones geométricas impuestas

Checar las restricciones geométricas impuestas sobre la caminata aleatoria es la parte del algoritmo que consume más tiempo en la generación de una caminata aleatoria auto-repelente.
Para las caminatas aleatorias auto-repelentes, el método que generalmente se emplea para verificar si una caminata aleatoria cruza por el mismo punto dos

veces es la tabla de Hash. Los detalles de la codificación de la tabla de Hash juntó con la estimación de la complejidad computacional se dan en el artículo de Madras *et al.* (Madras (1988)).

Se encontró que no es práctico, en el caso del algoritmo de dimerización, utilizar la tabla de Hash para verificar si una caminata aleatoria cruza por el mismo punto dos veces. La razón principal es que la tabla de Hash debe ser reinicializada (las entradas deben ser puestas a cero o nulificadas) cada vez que se concatenan dos dimers que no cumplen con la restricción de no cruzar por el mismo punto dos veces en cualquier nivel de recursión.
El número de intentos para concatenar dos dimers hasta obtener una caminata aleatoria auto-repelente es acumulativo. La cantidad de tiempo que se acumula cada vez que se reinicia la tabla de Hash hace más lento el procedimiento, especialmente en el caso de caminatas aleatorias auto-repelentes grandes.

Se prefiere el método de búsqueda binaria sobre una tabla de enteros ordenados, aunque se pueden utilizar otros métodos de búsqueda y ordenamiento, para verificar si una caminata aleatoria cruza por el mismo punto dos veces. En la presente tesis utilizamos las funciones SearchValue() para buscar un valor en una tabla ordenada y QuickSort() para ordenar los números en forma descendente.

El procedimiento es el siguiente. Para cada posición de un dimer se asigna un valor entero llamado llave. Por ejemplo, suponga que se tienen dos dimers en dos dimensiones con las siguientes posiciones:

Dimer A: {(0, 0), (0, 1), (1, 1), (1, 2), (0, 2), (-1, 2), (-1, 1)}
Dimer B: {(-1, 0), (-1, 1), (-1, 2), (0, 2), (1, 2), (2, 2), (1, 1)}

Para el dimer A, a cada posición se le asigna una entero, llamado llave, como se muestra a continuación:

Caminata A	
Posición	Valor asignado
(0, 0)	0
(0, 1)	65536
(1, 1)	65537
(1, 2)	131073
(0, 2)	131072
(-1, 2)	196607
(-1, 1)	131071

El dimer B se traslada a la última posición del dimer A, para la concatenación, nos queda de la siguiente manera:

Caminata B	
Posición original	Posición final
(-1, 0)	(-2, 1)
(-1, 1)	(-2, 2)
(-1, 2)	(-2, 3)
(0, 2)	(-1, 3)
(1, 2)	(0, 3)
(2, 2)	(1, 3)
(1, 1)	(0, 2)

Para el dimer B, a cada posición se le asigna una llave, como se muestra a continuación:

Caminata B	
Posición	Valor asignado
(-2, 1)	131070
(-2, 2)	196606
(-2, 3)	262142
(-1, 3)	262143
(0, 3)	196608
(1, 3)	196609
(0, 2)	131072

Cada vez que, sobre un nivel de recursión, dos dimers de M pasos están por concatenarse, el conjunto de llaves del dimer A es re-ordenado en orden descendente usando $O(M \log_2 M)$ rutina de ordenamiento(Press (1988)), como se muestra a continuación.

Caminata A
llaves re-ordenadas
196607
131073
131072
131071
65537
65536
0

Las llaves del segundo dimer se usan para checar la intersección por el método de búsqueda binaria. Del ejemplo se ve claramente que las dos caminatas se cruzan en un punto debido a que una llave del dimer A coincide con una llave del dimer B y por lo tanto ésta caminata no es auto-repelente.

4.4 Escoger el valor de la "longitud de Alexandrowicz"

El tercer problema asociado con la implementación del algoritmo de dimerización es escoger la "longitud de Alexandrowics" o N_A. Se puede escoger el valor de $N_A \geq 1$ de una manera arbitraria. Aunque si N_A se escoge de tal manera que el número de pasos N de la caminada es más pequeño que N_A, la caminata aleatoria auto-repelente es genera por un algoritmo básico para el cual se espera que la cantidad de tiempo para generar la caminata crezca exponencialmente con respecto al número de pasos N (Madras (1993)).

En el siguiente capítulo se muestran los resultados experimentales para determinar el valor óptimo de N_A en cuatro dimensiones.

Concluimos esta capítulo con la descripción del procedimiento que se usó para calcular los valores promedio y como se implementaron en nuestro programa.

Dejemos que $\omega^{(i)}(N) = (x^{(i)}(0), \ldots, x^{(i)}(N))$ sea la i-esima caminata en la muestra ($i = 1, \ldots, n$). Sean $\langle |\omega(N)| \rangle$ la distancia promedio y $\langle |\omega(N)|^2 \rangle$ la distancia al cuadrado promedio; sea $\overline{|\omega(N)|}$ una muestra de la distancia promedio y $\overline{\omega^2(N)}$ una muestra de la distancia al cuadrado promedio definidas como:

$$\overline{|\omega(N)|} = \frac{1}{n} \sum_{i=1}^{n} | x^{(i)}(N) - x^{(i)}(0) | \qquad (4.1)$$

$$\overline{\omega^2(N)} = \frac{1}{n} \sum_{i=1}^{n} \left[x^{(i)}(N) - x^{(i)}(0) \right]^2 \qquad (4.2)$$

El radio de giro al cuadrado R_g^2 de la i-esima caminata aleatoria auto-repelente esta dado por la relación (Madras (1988)):

$$R_g^2(\omega^{(i)}(N)) = \frac{1}{N+1} \sum_{k=0}^{N} \left(x^{(i)}(k) - \frac{1}{N+1} \sum_{j=0}^{N} x^{(i)}(j) \right)^2$$

El radio de giro al cuadrado promedio $\langle R_{g,N}^2 \rangle$ se estima por una muestra del radio de giro al cuadrado

$$\overline{R_{g,N}^2} = \frac{1}{n} \sum_{i=1}^{n} R_g^2(\omega^{(i)}(N)) \qquad (4.3)$$

A continuación se muestra parte del código fuente del programa principal en el cual se realizan todos los cálculos antes mencionados

.
.

```
#pragma omp threadprivate(IntSeed, MyRandomSite, MyRandomTime, MyRandomProb)

main()
{ TPtrRecord p;
  word NumSteps;
  TReal Distance,d,D_lin,dl,RGyr,Rad;
  int count=0;

   NumSteps=20000;

   Distance=0.0;
   D_lin=0.0;
   RGyr=0.0;

  // set random number generators
  srandom(time(0));
  MyRandomSite=-random();
  for(long int k=1; k<=50000L; k++) {}
  srandom(time(0));
  MyRandomTime=-random();

  cout << "Number of steps: " << NumSteps << "\n";
```

```
  printf("numero de cpus= %d\n", omp_get_num_procs());
  omp_set_dynamic(0);
  omp_set_num_threads(2);

  #pragma omp parallel private(p,d,d1,Rad) firstprivate(NumSteps)
shared(Distance,D_lin,RGyr)
   {
     printf("thread id: %d\n", omp_get_thread_num());

     #pragma omp for schedule(static) lastprivate(count)
reduction(+:Distance,D_lin,RGyr)
       for(count=1; count<=100000;count++)
           { Dimer(NumSteps, p);

             d=EndToEndDistance_2(NumSteps,p);
             d1=sqrt(d);
             Rad=GyrationRadius_2(NumSteps,p);
             Distance=Distance+d;
             D_lin=D_lin+d1;
             RGyr=RGyr+Rad;
             if ((count % 1000) == 1) cout << count << endl;

             DeleteWalk(p);
            }

     }

     cout << "Number of steps in SARW    : " << NumSteps << "\n";
     cout << "Number of SARWs processed : " << count-1 << "\n";
     cout << "End-to-end distance^2      : " << Distance/(count-1)<< "\n";
     cout << "End-to-end distance        : " << D_lin/(count-1)<< "\n";
     cout << "Radius of Gyration         : " << RGyr/(count-1)<< "\n";
     cout<<endl;

  return 0;
}
```


La idea principal del programa es dividir el número de caminatas aleatorias auto-repelentes que se va a generar entre el número de microprocesadores. Ésto se logra utilizando la directiva ***#pragma omp parallel*** para indicar que se van a utilizar varios microprocesadores. Utilizamos la directiva ***#pragma omp for*** *schedule(static)* para que cada microprocesador genere la misma cantidad de caminatas.

4.5 Observaciones

En el siguiente capítulo se realizan las pruebas experimentales utilizando el algoritmo de dimerización paralelizado.

5

Análisis de resultados

En este capítulo se explican lós diferentes experimentos que se realizaron y también se muestran los nuevos resultados que se obtuvieron al ejecutar el programa paralelizado con caminatas aleatorias auto-repelentes de hasta 20000 pasos.

5.1 Pruebas experimentales

En esta sección, se describen los diferentes experimentos que se realizaron con el algoritmo de dimerización paralelizado ejecutado en una computadora con las siguientes características:

Dos microprocesadores pentium II (i686) a 450 MHz.
Aquitectura SMP (Symetric Multiprocessing).
Memoria RAM 256Mb
Sistema operativo Linux Mandrake con kernel versión 2.2.15-4mdksmp el cual soporta la arquitectura SMP.
Programa en paralelo del algoritmo de dimerización.

Cada microprocesador ejecuta un hilo(número de hilos óptimo)(www.pgroup.com).

En todos los experimentos se generaron caminatas aleatorias auto-repelentes por medio del algoritmo de dimerización.

5.2 Experimento 1

Para estudiar la influencia de la longitud de Alexandrowicz en la eficiencia del algoritmo de dimerización, se han realizado una serie de experimentos ejecutando nuestro programa para generar caminatas en 4 dimensiones.

En cada ejecución, fueron generadas 10000 caminatas con un número de pasos igual a N = 200, 300, 400, 1000 y 5000.

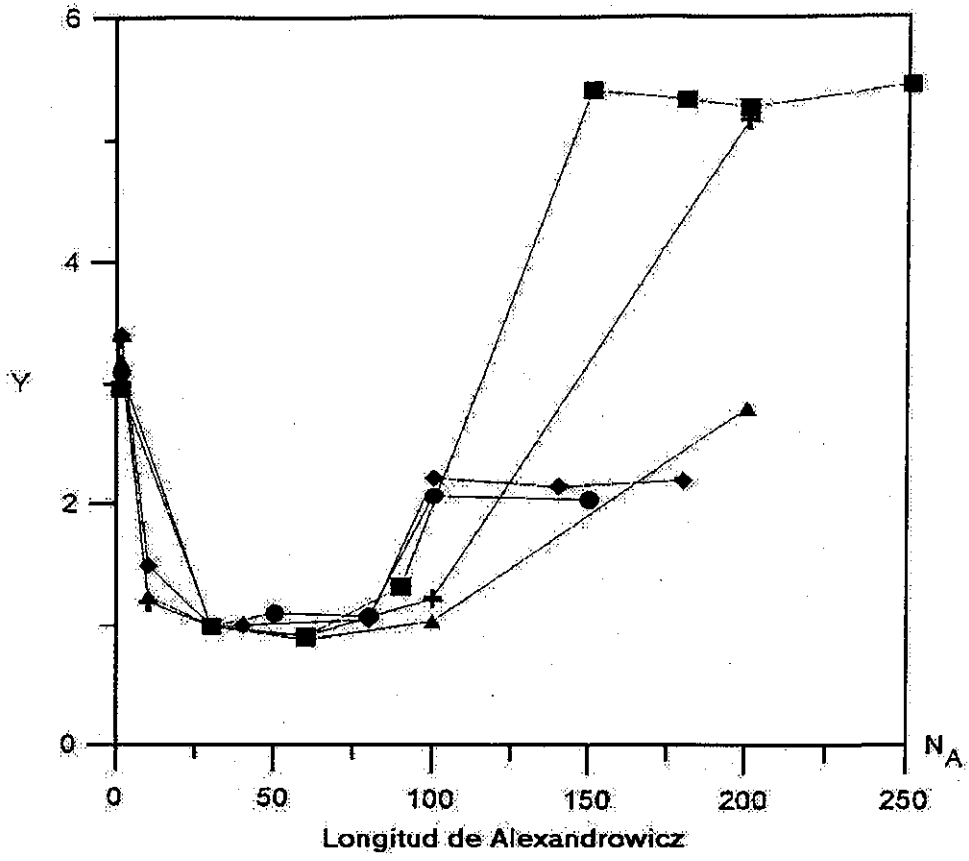
Dando el valor de N, se ejecuto el programa varias veces, cambiando la longitud de Alexandrowicz. Cada vez que se ejecutó el programa, se llevó un record del tiempo (en segundos) que tomó la computadora para generar las 10000 caminatas de N pasos.

Por otro lado, los experimentos computacionales mostraron que sí N_A = 1, el tiempo promedio necesario para generar las caminatas aleatorias auto-repelentes es más grande que para valores mayores de N_A. Por lo tanto, existe un valor de la "longitud de Alexandrowics" que minimiza el tiempo para generar caminatas aleatorias auto-repelentes.

Sea $T(N, N_A)$ es tiempo necesario para generar 10000 caminatas aleatorias auto-repelentes de N pasos usando el algoritmo de dimerización con cierta longitud de Alexandrowicz.

Se cálcula el cociente $T(N, N_A)/ T(N, 30)$ tomando el tiempo que corresponde a N_A= 30 como una referencia. Los resultados se presentan en la gráfica 5.1.

Se puede observar que existe un intervalo óptimo de valores para el parámetro N_A es cual es: $\mathbf{30 \leq N_A \leq 60}$.



Gráfica 5.1. *Y* denota la relación $T(N, N_A)/T(N, 30)$ en donde $T(N, N_A)$ es el tiempo del CPU necesario para generar 10000 caminatas aleatorias auto-repelentes de N pasos con un valor determinado de N_A. Diamante N = 200, cuadrado N = 300, círculo N = 400, triángulo N = 1000, cruz N = 5000. $T(N, 30)$ representa el tiempo del CPU cuando $N_A = 30$.

De la gráfica podemos concluir para el caso de caminatas aleatorias auto-repelentes en cuatro dimensiones, el valor óptimo para N_A está entre **30** y **60**.

5.3 Experimento 2

En nuestra simulación de las caminatas aleatorias auto-repelentes en 4 dimensiones se usa el valor de $N_A = 30$.

Las caminatas aleatorias auto-repelentes en 4 dimensiones son materia de interés por la existencia de una fuerte influencia de las correcciones logarítmicas que conducen el comportamiento asintótico de la distancia al cuadrado promedio:

$$\left\langle |\omega(N)|^2 \right\rangle \sim N \left[\log N\right]^{\alpha} \qquad (5.1)$$

El reporte(Rapoport (1985)) ha sido realizado mediante la simulación Monte Carlo para caminatas aleatorias auto-repelentes en 4 dimensiones (con $N \leq 2400$) y se llega a la conclusión que $\alpha = 0.31$. Grassberger (1994) ha realizado cuidadosos experimentos Monte Carlo sobre caminatas aleatorias auto-repelentes en 4 dimensiones con $N \leq 4000$ y encontró indicaciones definitivas sobre las correcciones logarítmicas del comportamiento asintótico de (5.1) con $\alpha = \frac{1}{4}$. Esas correcciones vienen de la teoría de grupos de re-normalización y han sido calculadas por Duplantier (1986):

$$\frac{\left\langle |\omega(N)|^2 \right\rangle}{N} = r \left[\ln(N/a)\right]^{1/4} \left[1 - \frac{17 \ln(4 \ln(N/4)) + 31}{64 \ln(N/a)} + \ldots\right] \qquad (5.2)$$

en donde r y a son dos parámetros para ajustar $\left\langle |\omega(N)|^2 \right\rangle / N$ cuyos valores han sido encontrados por Grassberger (Grassberger (1994)): $r = 1.331$, $a = 0.1237$.

Para verificar (5.2), hemos simulado caminatas aleatorias auto-repelentes usando la simulación Monte Carlo con el algoritmo de dimerización extendiendo substancialmente el rango del número de pasos en las caminatas generadas.

El valor máximo de N utilizado en nuestros cálculos fue de 20000, que es cinco veces más grande que el valor usado por Grassberger (1994). Para nuestro conocimiento, este es el valor más grande de N reportado en la simulación por computadora de caminatas aleatorias auto-repelentes en 4 dimensiones.

Nuestros resultados numéricos, incluyen la distancia al cuadrado promedio y el radio de giro al cuadrado, se listan en la tabla I y muestran en la gráfica 5.2. utilizamos (5.1) para comparar con (5.2); se gráfica $\log\left(\overline{\omega^2(N)} / N\right)$ contra $\log(\log N)$. Los cuadrados son dibujados de acuerdo a los datos presentados en la tabla I para $N \geq 3000$. La línea sólida corresponde a (5.2) con el coeficiente r y a encontrados en la Ref. (Grassberger (1994)). La línea punteada corresponde a (5.1).

El mejor valor de α obtenido de nuestros datos es 0.294.

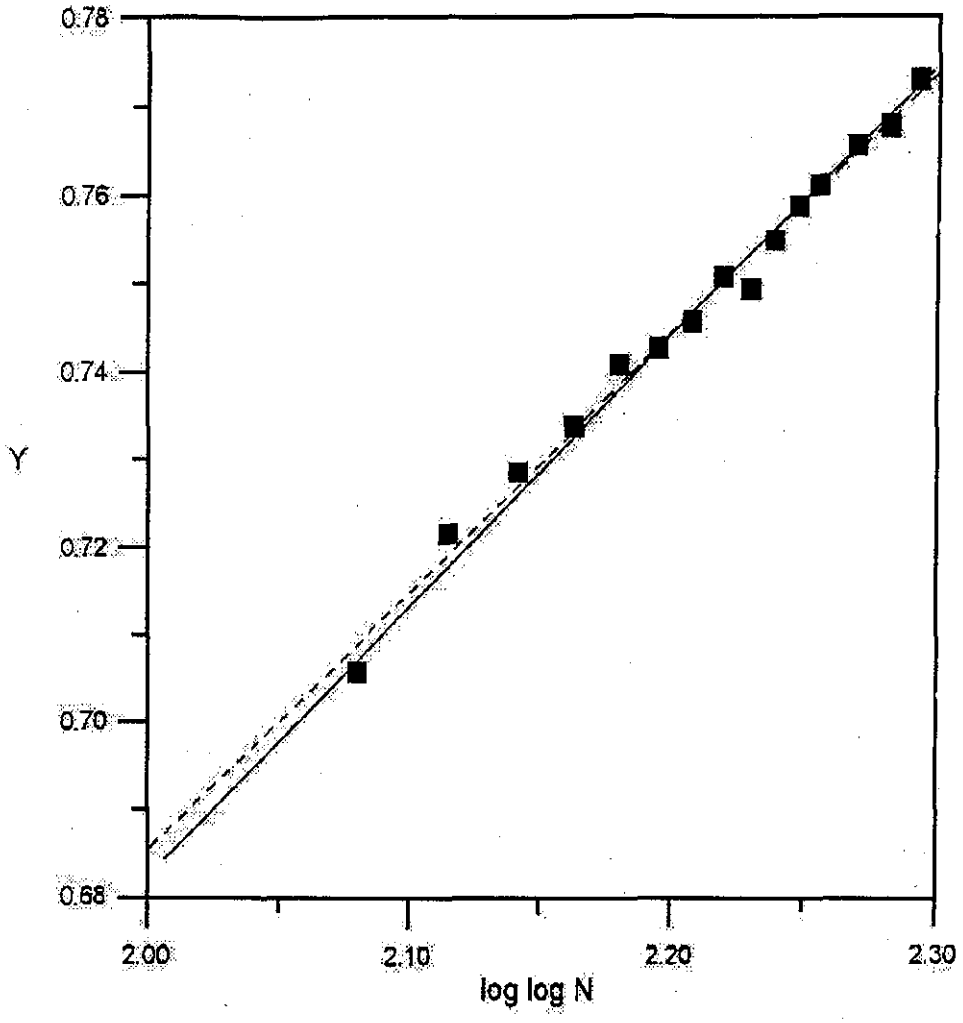
Esto confirma la conclusión hecha por Grassberger (Grassberger (1994)), la cual nos dice, aún para caminata aleatorias auto-repelente grandes (N mayores

de 20000), el comportamiento de $\langle |\omega(N)|^2 \rangle$ está muy lejos de la forma asintótica (5.1) con $\alpha = \frac{1}{4}$ y las correcciones logarítmicas (5.2) aún juegan un rol significativo.

Por último en la tabla II se muestra el tiempo utilizado por el algoritmo de dimerización para generar 100000 caminatas aleatorias auto-repelentes ejecutado de una manera serial y de una manera en paralelo. Se puede apreciar que el tiempo de computo en paralelo es aproximadamente 2 veces menor que el tiempo de cálculos sin paralelización.

N	Tamaño de la muestra	$\overline{\omega^2(N)}$	$\overline{R^2_{g,N}}$
120	100000	207.065	34.0479
120^R	≈50000	207.8	34.15
300	100000	550.111	90.3992
300^R	≈50000	549.1	90.71
600	100000	1137.67	187.939
600^R	≈50000	1131.3	187.10
1200	100000	2351.92	388.419
1200^R	≈50000	2347.6	388.34
2400	100000	4826.44	798.972
2400^R	≈50000	4811.0	797.88
3000	100000	6069.41	1005.49
4000	100000	8202.31	1358.47
5000	100000	10315.8	1711.86
6000	100000	12494.4	2071.37
7000	100000	14588.9	2424.25
8000	100000	16789	2786.49
9000	100000	18972.6	3141.4
10000	100000	21122.2	3507.74
11000	100000	23316.5	3868.6
12000	100000	25498.7	4236.33
13000	100000	27706.2	4595.63
14000	100000	30092	4974.69
16000	100000	34418.9	5705.62
18000	100000	38819.3	6438.27
20000	100000	43318.3	7185.98

Tabla I. El número de pasos de las caminatas generadas se da en la primera columna. La letra *R* sobre algunos números de pasos remarca los resultados del reporte (Rapoport 1985)). El tamaño de las muestras usado para calcular los valores promedio se listan en la segunda columna. La tercer columna muestra los valores de las muestras de la distancia al cuadrado promedio $\overline{\omega^2(N)}$ y, por último, los valores de las muestras del radio de giro al cuadrado $\overline{R^2_{g,N}}$.



Gráfica 5.2. Y representa los valores de $\log(\overline{\omega^2(N)}/N)$. Los cuadrados corresponden a los datos listados en la tabla I para $N \geq 3000$. La línea sólida representa la Ecuación (5.2) con r = 1.331 y a = 0.1237 encontrados por Grassberger (Grassberger (1994)). La línea punteada es la mejor curva (5.1) con α = **0.294**. Aún para caminatas grandes ($N \approx 20000$), el comportamiento de la distancia al cuadrado promedio está lejos de (5.1) con α = ½ y conduce a correcciones logarítmicas que aún juegan un papel importante.

N	Horas/Serial	Horas/Paralelo
120	0.033	0.016
300	0.25	0.133
600	0.883	0.438
1200	3.016	1.65
2400	10.333	5.633
3000	14.583	8
4000	27.016	14.716
5000	38.366	20.966
6000	50.566	27.7
7000	64.833	35.633
8000	94.15	51.133
9000	114.233	62.433
10000	134.533	73.166
11000	155.716	84.55

Tabla II. Muestra el tiempo que tomó el algoritmo de dimerización ejecutado de por un sólo microprocesador (serial) y ejecutado por dos microprocesadores (paralelo).

6

Conclusiones

En este capítulo se dan las conclusiones a las que se llegaron en base a los resultados experimentales y se propone un trabajo a futuro.

6.1 Conclusiones generales

Se ha estudiado el algoritmo de dimerización desde un punto de vista computacional.

En particular, se analizaron las caminatas aleatorias auto-repelentes en 4 dimensiones utilizando el algoritmo de dimerización. Se encontró que existe un rango de valores óptimo de la longitud de Alexandrowicz que minimiza el tiempo promedio de CPU requerido para generar una caminata aleatoria auto-repelente. Estos valores óptimos se encuentran en el intervalo $\mathbf{30} \leq N_A \leq \mathbf{60}$.

Se analizaron las caminatas aleatorias auto-repelentes de hasta 20000 pasos en cuatro dimensiones. Aún para caminatas tan grandes el comportamiento de la distancia al cuadrado promedio esta más allá de la forma asintótica (5.1) con $\alpha = \frac{1}{4}$. Ésto nos lleva a concluir que las correcciones logarítmicas (5.2) aún juegan un papel importante.

También, como se puede notar, el procesamiento en paralelo reduce el tiempo de ejecución de un programa de tal manera que cuando el algoritmo de dimerización es ejecutado por n microprocesadores entonces es aproximadamente n veces más rápido.

Para concluir, se puede agregar que aunque el algoritmo de dimerización no es el más eficiente para generar caminatas, su gran cualidad consiste en generar muestras independientes estadísticamente y uniformemente distribuidas sobre el conjunto de todas las caminatas aleatorias sin importar las restricciones geométricas que se impongan. Ésto lo hace una herramienta útil en la generación de caminatas aleatorias.

6.2 Trabajo a futuro

En la presente tesis se ha utilizado una computadora en paralelo con dos microprocesadores y arquitectura de memoria compartida y se llegó a resultados interesantes, aunque se redujo el tiempo de computo aún así es el principal problema.

Las computadoras en paralelo más rápidas en la actualidad son las computadoras que combinan la arquitectura de memoria compartida y memoria distribuida. En la FACULTAD DE ESTUDIOS SUPERIORES CUAUTITLAN se cuenta en la actualidad con una computadora de este tipo y se piensa utilizar para continuar la investigación sobre las caminatas aleatorias auto-repelentes.

Apéndices

Apéndice A

Valores de los exponentes críticos γ y ν

Valores de los exponentes críticos γ y ν (conjeturas).

Valores para γ	
43/32	d = 2
1.162	d = 3
1 con correcciones logarítmicas	d = 4
1	d ≥ 5

Valores para ν	
¾	d = 2
0.59	d = 3
½ con correcciones logarítmicas	d = 4
½	d ≥ 5

Apéndice B

Valores exactos de c_N

Este apéndice contiene las tablas que muestran el valor exacto de c_N, en 2,3,4,5 y 6 dimensiones, la sumatoria de la distancia al cuadrado y el número de polígonos auto-repelentes de N pasos.

N	$d = 2$	$d = 3$
0	1	1
1	4	6
2	12	30
3	36	150
4	100	726
5	284	3 534
6	780	16 926
7	2 172	81 390
8	5 916	387 966
9	16 286	1 853 886
10	44 100	8 809 878
11	120 292	41 934 150
12	324 932	198 842 742
13	881 500	943 974 510
14	2 374 444	4 468 911 678
15	6 416 596	21 175 146 054
16	17 245 332	100 121 875 974
17	46 466 676	473 730 252 102
18	124 658 732	2 237 723 684 094
19	335 116 620	10 576 033 219 614
20	897 697 164	49 917 327 838 734
21	2 408 806 028	235 710 090 502 158
22	6 444 560 484	
23	17 266 613 812	
24	46 146 397 316	
25	123 481 354 908	
26	329 712 786 220	
27	881 317 491 628	
28	2 351 378 582 244	
29	6 279 396 229 332	
30	16 741 957 935 348	
31	44 673 816 630 956	
32	119 034 997 913 020	
33	317 406 598 267 076	
34	845 279 074 648 708	

Tabla B.1: Muestra el valor exacto de c_N para $d = 2,3$ (Masand (1992); Guttmann and Wand (1991); Guttmann (1989b)). Los valores para $d = 2$ son conocidos hasta $N = 39$ (Conway, Enting and Guttmann (private communication)).

N	$d = 4$	$d = 5$	$d = 6$
0	1	1	1
1	8	10	12
2	56	90	132
3	392	810	1 452
4	2 696	7 210	15 852
5	18 584	64 250	173 172
6	127 160	570 330	1 887 492
7	871 256	5 065 530	20 578 452
8	5 946 200	44 906 970	224 138 292
9	40 613 816	398 227 610	2 441 606 532
10	276 750 536	3 527 691 690	26 583 605 772
11	1 886 784 200	31 255 491 850	289 455 960 492
12	12 843 449 288		
13	87 456 597 656		

Tabla B.2: Muestra el valor exacto de c_N para $d = 4,5,6$ (Fisher and Gaunt (1964)), para $d = 4$, $N = 13,14$ (Guttmann (1978)).

N	$d = 2$	$d = 3$
1	4	6
2	32	72
3	124	582
4	704	4 032
5	2 716	25 566
6	9 808	153 528
7	33 788	886 926
8	112 480	4 983 456
9	364 588	27 401 502
10	1 157 296	148 157 880
11	3 610 884	790 096 950
12	11 108 448	4 166 321 184
13	33 765 276	21 760 624 254
14	101 594 000	112 743 796 632
15	302 977 204	580 052 260 230
16	896 627 936	2 966 294 589 312
17	2 635 423 124	15 087 996 161 382
18	7 699 729 296	76 384 144 381 272
19	22 374 323 436	385 066 579 325 550
20	64 702 914 336	1 933 885 653 380 544
21	186 289 216 332	9 679 153 967 272 734
22	534 227 118 960	
23	1 526 445 330 900	
24	4 347 038 392 480	
25	12 341 626 847 324	
26	34 940 293 640 400	
27	98 660 244 502 668	
28	277 910 662 983 584	
29	781 060 493 709 204	

Tabla B.3: Muestra la sumatoria de la distancia al cuadrado de todas las caminatas aleatorias auto-repelentes de N pasos, para $d = 2,3$ (Guttmann and Wang (1991); Guttmann (1989b) y Guttmann (1987)). La distancia al cuadrado promedio se obtiene dividiendo por c_N.

N	$d = 2$	$d = 3$
4	1	3
6	2	22
8	7	207
10	28	2 412
12	124	31 754
14	588	452 640
16	2 938	6 840 774
18	15 268	108 088 232
20	81 826	1 768 560 270
22	449 572	
24	2 521 270	
26	14 385 376	
28	83 290 424	
30	488 384 528	
32	2 895 432 660	
34	17 332 874 364	
36	104 653 427 012	
38	636 737 003 384	
40	3 900 770 002 646	
42	24 045 500 114 388	
44	149 059 814 328 236	
46	928 782 423 033 008	
48	5 814 401 613 289 290	
50	36 556 766 640 745 936	
52	230 757 492 737 449 636	
54	1 461 972 662 850 874 916	
56	9 293 993 428 791 901 042	

Tabla B.4: Muestra el número de polígonos auto-repelentes para $d = 2,3$ (Guttmann and Enting (1988)). Valores para $d = 3$, $N \leq 14$ (Fisher and Gaunt (1964)). Valores para $d = 3$, $N \leq 16$ (Sykes (1972)).

N	$d = 4$	$d = 5$	$d = 6$
4	6	10	15
6	76	180	350
8	1 434	5 170	13 545
10	32 616	186 856	679 716
12	844 432	4 060 132	17 761 132
14	23 919 864		

Tabla B.5: Muestra el número de polígonos auto-repelentes para $d = 4,5,6$. Para $N \leq 10$ (Fisher and Gaunt (1964)) y para $N \geq 12$ (Guttmann (private communication)).

Apéndice C

Librería de funciones OpenMP

En esta sección se describen las funciones disponibles en OpenMP para C y C++; el archivo de cabecera declará dos tipos de funciones:

1. Funciones para controlar y verificar el habiente de ejecución en paralelo.
2. Funciones que manipulan los candados para sincronizar los hilos.

En las funciones que manipulan los candados, el objeto de candado debe ser del tipo omp_lock_t o omp_nest_lock_t. Estós objetos representan un candado que esta disponible o que pertenece a un hilo.

Los objetos de candado deben ser accesado por medio de las funciones que manipulan los candados.

Todas las funciones que manipulan los objetos de candado requieren de un argumento que punta al objeto de candado del tipo omp_lock_t o omp_nest_lock_t.

Funciones que manipulan el ambiente de ejecución

omp_set_num_threads()

Propósito:
La función omp_set_num_threads() fija el número de hilos que se lanzarán cuando se entré en la región en paralelo.

Prototipo:
void omp_set_num_threads(int num_threads):

Incluir:
#include <omp.h>

Descripción:
El valor del parámetro **num_threads** debe ser un número positivo. Su efecto depende del ajuste dinámico del número de hilos está habilitado o desabilitado. Sí el ajuste dinámico está desbilidato, el valor del parámetro **num_threads** se usa como el número de hilos que lanzarán las regiones en paralelo antes de volver a llamar la función omp_set_num_threads(); sí el ajuste dinámico está habilitado el valor del parámetro **num_threads** será el número máximo de hilos que se lanzarán en la región en paralelo.

La función omp_set_num_threads() tiene efecto sólo sí es llamada desde una porción serial del programa. Sí es llamada desde una región en paralelo el valor que devuelve la función es un valor diferente de cero, el comportamiento de la función es indefinido.

Esta función tiene mayor prioridad que la variable de ambiente OMP_NUM_THREADS.

Ver también:
omp_get_dynamic(),omp_set_dynamic().

omp_get_num_threads()

Propósito:
La función omp_get_num_threads() devuelve el número de hilos que están ejecutando la región en paralelo.

Prototipo:
```
int omp_get_num_threads(void);
```

Incluir:
```
#include <omp.h>
```

Descripción:
La función omp_get_num_threads() devuelve el número de hilos que están ejecutando la región en paralelo, desde donde es llamada la función. Sí es llamada de una porción serial del programa o desde una región paralela anidada que está serializada la función devuelve 1.

Sí el número de hilos no ha sido explícitamente asignado por el usuario, el número de hilos por default se utiliza para ejecutar la región en paralelo.

El número de hilos por default es dependiente de la implementación.

La función omp_set_num_threads() y la variable de ambiente OMP_NUM_THREADS controlan el número de hilos de el conjunto de hilos.

omp_get_max_threads()

Propósito:
La función omp_get_max_threads() devuelve el valor máximo que puede ser devuelto por la función omp_get_num_threads().

Prototipo:
```
int omp_get_max_threads(void);
```

Incluir:
```
#include <omp.h>
```

Descripción:
Sí se usa la función omp_set_num_threads() para cambiar el número de hilos que ejecutarán la región en paralelo, las llamadas siguientes a la función omp_get_max_threads() devolverán el nuevo valor.

La función omp_get_max_threads() devuelve el valor máximo aunque la función se llame desde una porción serial o región en paralelo del programa.

Generalmente la función omp_get_max_threads() refleja el valor de la variable de ambiente OMP_NUM_THREADS.

omp_get_threads_num()

Propósito:

La función omp_get_threads_num() devuelve el número de hilo que le pertenece a un hilo determinado dentro del conjunto de hilos al que pertenece.

Prototipo:
```
int omp_get_threads_num(void);
```

Incluir:
```
#include <omp.h>
```

Descripción:
La función omp_get_threads_num() devuelve el número de hilo que le pertenece a un hilo determinado dentro del conjunto de hilos al que pertenece. El número de hilo está entre 0 y omp_get_threads_num()-1. El hilo maestro del conjunto es el hilo número 0.

Sí la función omp_get_threads_num() es llamada de una porción serial del programa devuelve cero. Sí es llamada desde una región en paralelo anidada serializada la función devuelve cero.

omp_get_num_procs()

Propósito:
La función omp_get_num_procs() devuelve el número máximo de procesadores que pueden ser asignados al programa.

Prototipo:
```
int omp_get_num_procs(void);
```

Incluir:
```
#include <omp.h>
```

Descripción:
La función devuelve el número máximo de procesadores que pueden ser asignados al programa.

omp_in_parallel()

Propósito:
La función omp_in_parallel() determina sí la sección de código se está ejecutando en paralelo o no.

Prototipo:
```
int omp_in_parallel(void);
```

Incluir:
```
#include <omp.h>
```

Descripción:
La función omp_in_parallel() devuelve un valor diferente de cero si la función es llamada desde una extensión dinámica de una región ejecutándose en paralelo; si es llamada desde una porción de código que se está ejecutando en serie devuelve cero.

La función devuelve un valor diferente de cero si es llamada de una porción de código serial, despreciando las regiones en paralelo anidadas serializadas.

omp_set_dynamic()

Propósito:
La función omp_set_dynamic() habilita o desabilita el ajuste dinámico del número de hilos disponibles para ejecutar la región en paralelo.

Prototipo:
```
void omp_set_dynamic(int dynamic_threads);
```

Incluir:
```
#include <omp.h>
```

Descripción:
La función omp_set_dynamic() tiene efecto sólo sí se llama desde una porción serial del programa. Sí la función es llamada desde una región en paralelo devuelve cero; el comportamiento de la función es indefinido.

Sí al evaluar la variable dynamic_threads devuelve un valor distinto de cero, el número de hilos que será usado para ejecutar las siguientes regiones en paralelo puede ser ajustado automáticamente por las variables de ambiente para un mejor uso de los recursos del sistema. Como consecuencia, el número de hilos especificado por el usuario es el número máximo de hilos que se pueden utilizar. El número de hilos siempre permanece fijo mientras se ejecuta la región en paralelo y el número de hilos es reportado por la función omp_get_num_threads().

Sí al evaluar la variable dynamic_threads devuelve un valor cero, el ajuste dinámico está desabilitado.

El valor por default de el ajuste dinámico es dependiente de la implentación. Como consecuencia, el código del usuario que depende de un número especifico de hilos para su correcta ejecución debe desabilitar explícitamente los hilos dinámicos.

La función omp_set_dynamic() tiene mayor prioridad que la variable de ambiente OMP_DYNAMIC.

omp_get_dynamic()

Propósito:
La función omp_get_dynamic() determina sí el ajuste dinámico de los hilos está habilitado o no.

Prototipo:
```
int omp_get_dynamic(void);
```

Include:
```
#include <omp.h>
```

Descripción:

La función omp_get_dynamic() devuelve un valor diferente de cero si el ajuste dinámico está habilitado y si está desabilitado devuelve cero.

omp_set_nested()

Propósito:
La función omp_set_nested() habilita o desabilita el uso de regiones en paralelo anidadas.

Prototipo:
```
void omp_set_nested(int nested);
```

Incluir:
```
#include <omp.h>
```

Descripción:
Sí al evaluar la variable **nested** es cero, que es el default, las regiones en paralelo anidadas están desabilitadas, y las regiones en paralelo andidas son ejecutadas en forma serial y ejecutadas por el hilo actual. Sí al evaluar la variable **nested** es diferente de cero, las regiones en paralelo anidadas están habilitadas, y las regiones en paralelo anidadas pueden dejar de usar algunos hilos para ajustar el conjunto de hilos.

La función omp_set_nested() tiene una mayor prioridad sobre la variable de ambiente OMP_NESTED.

Cuando esta habilita la región en paralelo andidas, el número de hilos que se usa para ejecutar la región en paralelo anidada es dependiente de la implementación. Como resultado, la implementación de OpenMP permiten serializar la región en paralelo anidada aun cuando esté habilitado la región en paralelo anidada.

omp_get_nested()

Propósito:
Esta función omp_get_nested() determina si el anidamiento paralelo está o no habilitado.

Prototipo:
```
int omp_get_nested(void);
```

Incluir:
```
#include <omp.h>
```

Descripción:
La función omp_get_nested() devuelve un valor diferente de cero si la región en paralelo anidada está habilitada y cero si está desabilitada.

Sí la implementación no soporta el paralelismo anidado, la función siempre devuelve cero.

Funciones que manipulan los candados

omp_init_lock() y omp_init_nest_lock()

Propósito:
Estas funciones omp_init_lock() y omp_init_nest_lock() crean e inicializan los objetos de candados asociados con la variable critica.

Prototipo:
```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

Incluir:
```
#include <omp.h>
```

Descripción:
Estas funciones omp_init_lock() y omp_init_nest_lock(), proporcionan el único medio de crear e inicializar los objetos de candados. Cada función inicializa el candado asociado con el parámetro **lock** para su uso en las siguientes llamadas.

El estado inicial de los candados es abierto (esto es el candado no pertenece a nigún hilo). Para candados anidados, el contador de anidamiento es inicializado a cero.

omp_destroy_lock() y omp_destroy_nest_lock()

Propósito:
Las funciones omp_destroy_lock() y omp_destroy_nest_lock(), destruyen los objetos de candado asignados a las variables criticas.

Prototipo:
```
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

Descripción:
Las funciones omp_destroy_lock() y omp_destroy_nest_lock(), aseguran que el puntero al objeto de candado **lock** esté sin inicializar.

El argumento de estas funciones debe apuntar a un objeto de candado sin inicializar esto es sin candado.

Es ilegal llamar a esta función con una variable que tiene un candado sin inicializar.

omp_set_lock() y omp_set_nest_lock()

Propósito:
Estas funciones forzan al hilo que se está ejecutando a esperar hasta que el candado especificado esté disponible.

Prototipo:
```
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

Incluir:
#include <omp.h>

Descripción:
Cada una de estas funciones bloquea el hilo que ejecuto alguna de las funciones hasta que el candado especificado esté disponible. El candado está disponible si está abierto. Un candado anidado está disponible si está abierto o si ya pertenece a un hilo que ejecuta la función.

Para un candado simple, el argumento de la función omp_set_lock(), debe apuntar a un objeto de candado sin inicializar. El propietario del candado se le garantiza que el hilo ejecutará la función.

Para un candado anidado, el argumento de la función omp_set_nest_lock(), debe apuntar a un objeto de candado sin inicializar. El contador de anidamiento es incrementado, y se le garantiza al hilo mantenerse como propietario del candado.

omp_unset_lock() y omp_unset_nest_lock()

Propósito:
Las funciones omp_unset_lock() y omp_unset_nest_lock() liberan el candado.

Prototipo:
```
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

Incluir:
#include <omp.h>

Descripción:
Las funciones omp_unset_lock() y omp_unset_nest_lock(), proporcionan una manera de liberar el candado.

El argumento de cada una de estás funciones debe apuntar a un objeto de candado que pertenece al hilo que ejecuta la función. El comportamiento está indefinido si el hilo no es el propietario del candado.

Para un candado simple, la función omp_set_lock(), libera al hilo que está ejecutando la función del candado.

Para un candado anidado, la función omp_set_nest_lock() decrementa el contador de anidamiento, y libera el hilo que está ejecutando la función de el candado si el resultado del contador es cero.

omp_test_lock() y omp_test_nest_lock()

Propósito:
Las funciones omp_set_lock() y omp_set_nest_lock() intentan cerrar un candado pero no se bloquean el hilo que intento cerrar el candado.

Prototipo:
```
void omp_test_lock(omp_lock_t *lock);
```

```
void omp_test_nest_lock(omp_nest_lock_t *lock);
```

Incluir:
```
#include <omp.h>
```

Descripción:
Estas funciones omp_set_lock() y omp_set_nest_lock(), intentan cerrar el candado pero no bloquean la ejecución del hilo.

El argumento debe apuntar a un objeto de candado sin inicializar. Estas funciones intentan cerrar el candado en la misma manera como lo hace la función omp_set_lock() y omp_set_nest_lock(), excepto que ellas no bloquean el hilo que se está ejecutando.

Para candados simples, la función omp_test_lock() devuelve un valor diferente de cero sí el candado es cerrado, si no consigue cerrarlo devuelve cero.

Para candados anidados, la función omp_set_nest_lock() devuelve el nuevo valor del contador de anidamiento sí el candado es cerrado; sí no consigue cerrarlo, devuelve cero.

Apéndice D

Ejemplos utilizando las librerías OpenMP

D.1 Ejemplo: Crear una región en paralelo

Programa que imprime "Hola Mundo".

- Cada hilo ejecuta el código que se encuentra dentro de la región en paralelo.
- Las funciones de la librería OpenMP se utilizan para obtener los números identificadores de los hilos y el número de hilos de la región en paralelo.

```
#include <stdio.h>
#include <omp.h>

main()
{ int nthreads, tid;

  /*Crea un conjunto de hilos */
  #pragma omp parallel private(nthreads, tid)
  {
    /* Obtiene e imprime la identidad del hilo */
    tid = omp_get_thread_num();
    printf("Hola mundo desde un hilo = %d\n", tid);

    /* Sólo el hilo maestro ejecuta este código */
    if(tid == 0)
      { nthreads = omp_get_num_threads();
        printf("Número de hilos = %d\n", nthreads);
      }

  }  /* Todos los hilos se unen al hilo maestro y terminan */

}
```

D.2 Ejemplo: Fijar el número de hilos para una región en paralelo

Parte de un programa que muestra como fijar el número de hilos que usará la región en paralelo.
Note que el número de hilos permanece constante durante la ejecución de la región en paralelo. El mecanismo dinámico de hilos determina el número de hilos que usará al iniciar la región en paralelo y se mantiene constante durante la ejecución de la región en paralelo.

```
omp_set_dynamic(0);
omp_set_num_threads(16);
#pragma omp parallel shared(x, npoints), private(iam, ipoints)
{
  if(omp_get_num_threads() != 16)abort();
  iam = omp_get_thread_num();
  ipoints = npoints / 16;
  do_by_16(x, iam, ipoints);
}
```

D.3 Ejemplo: Determinar el número de hilos usados por la región en paralelo

Para determinar el número de hilos que serán usados por la región en paralelo, la función debe ser llamada desde dentro de la región en paralelo.

```
#pragma omp paralel private(i)
{
  i = omp_get_thread_num();
  work(i);
 }
```

D.4 Ejemplo: Trabajo compartido directiva **for**

Programa que suma dos vectores, el arreglo A, B, C y la variable N serán compartidos por todos los hilos.
La variable i será privada para cada hilo; cada hilo tendrá su propia copia de la variable i.
Las iteraciones del loop serán distribuidas dinámicamente en pedazos del tamaño CHUNK. Los hilos no se sincronizaran al terminar su trabajo (NOWAIT).

```
#include <stdio.h>
#include <omp.h>

#define CHUNK 100
#define N     1000

main()
{ int i, n, chunk;
  float a[N], b[N], c[N];

  /* Inicializando los arreglos */
  for(i=0; i < N; i++)a[i] = b[i] = i * 1.0;
  n = N;
  chunk = CHUNK;

  #pragma omp parallel shared(a,b,c,n,chunk) private(i)
  {
    #pragma omp for schedule(dynamic,chunk) nowait
    for(i=0; i < n; i++)c[i] = a[i] + b[i];
   }  /* fin de las regiones en paralelo */

}
```

D.5 Ejemplo - Trabajo compartido directiva **sections**

En este ejemplo, las iteraciones del loop se reparten en dos diferentes secciones. Cada sección será ejecutada por un hilo. Hilos extras no participaran en el código de la sección.

```
#include <stdio.h>
#include <omp.h>

#define N     50
```

```
main()
{ int i, n, nthreads, tid;
  float a[N], b[N], c[N];

  /* inicialización */
  for(i=0; i < N; i++)a[i] = b[i] = i * 1.0;
  n = N;

  #pragma omp parallel shared(a, b, c, n) private(i, tid, nthreads)
  {
    tid = omp_get_thread_num();
    printf("Hilo %d inicia...\n",tid);

    #pragma omp sections nowait
    {
      #pragma omp section
      for(i=0; i < n/2; i++)
         { c[i] = a[i] + b[i];
           printf("tid= %d i= %d c[i]= %f\n",tid,i,c[i]);
         }

      #pragma omp section
      for(i=n/2; i < n; i++)
         {
           c[i] = a[i] + b[i];
           printf("tid= %d i= %d c[i]= %f\n",tid,i,c[i]);
         }

    }  /* fin de la sección */

  if(tid == 0)
    {
      nthreads = omp_get_num_threads();
      printf("Número de hilos = %d\n", nthreads);
    }

  }  /* fin de la sección en paralelo */

}
```

D.6 Ejemplo - Trabajo compartido combinación de las directivas **parallel for**

```
#include <stdio.h>
#include <omp.h>

#define N       50
#define CHUNK   5

main()
{ int i, n, chunk, tid;
  float a[N], b[N], c[N];
  char first_time;

  /* inicialización */
```

```
  for(i=0; i < N; i++)a[i] = b[i] = i * 1.0;

  n = N;
  chunk = CHUNK;
  first_time = 'y';

  #pragma omp parallel for shared(a,b,c,n) \
  private(i,tid) schedule(static,chunk) firstprivate(first_time)

  for(i=0; i < n; i++)
     { if(first_time == 'y')
       { tid = omp_get_thread_num();
         first_time = 'n';
       }

       c[i] = a[i] + b[i];
       printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
     }

}
```

D.7 Ejemplo: Trabajo compartido combinación de las directivas **parallel for**

Las iteraciones del loop for serán distribuidas equitativamente a cada hilo del conjunto (SCHEDULE STATIC).

```
#include <stdio.h>
#include <omp.h>

#define N       1000
#define CHUNK   100

main()
{ int i, n, chunk;
  float a[N], b[N], c[N];

  /* Inicialización de los vectores */
  for(i=0; i < N; i++)a[i] = b[i] = i * 1.0;
  n = N;
  chunk = CHUNK;

  #pragma omp parallel for shared(a,b,c,n) private(i) schedule(static,chunk)
  for (i=0; i < n; i++)c[i] = a[i] + b[i];

}
```

D.8 Ejemplo: Construcción **critical**

Todos los hilos del conjunto se ejecutan en paralelo, aunque, por la región crítica sólo un hilo será capaz de leer/incrementar/escribir en la variable x en cualquier instante.

```
#include <stdio.h>
#include <omp.h>
```

```
main()
{ int x;

  x = 0;

  #pragma omp parallel shared(x)
  {
    #pragma omp critical
    x = x + 1;
   }  /* fin de la sección en paralelo */

}
```

D.9 Ejemplo: Directiva **threadprivate**

```
#include <stdio.h>
#include <omp.h>

int alpha[10], beta[10], i;

#pragma omp threadprivate(alpha)

main()
{ /* Primera región en paralelo */
  #pragma omp parallel private(i,beta)
  for(i=0; i < 10; i++)alpha[i] = beta[i] = i;

  /* Segunda región en paralelo */
  #pragma omp parallel
  printf("alpha[3]= %d y beta[3]= %d\n",alpha[3],beta[3]);

}
```

D.10 Ejemplo - Uso de la cláusula **reduction**

Este ejemplo demuestra el uso de la cláusula **reduction** dentro de una construcción en paralelo. Note que el alcance de las variables está por default - no hay cláusulas que especifiquen si las variables son compartidas o privadas.

```
#include <stdio.h>
#include <omp.h>

main()
{ int   i, n;
  float a[100], b[100], sum;

  /* inicialización */
  n = 100;
  for (i=0; i < n; i++)a[i] = b[i] = i * 1.0;

  sum = 0.0;

  #pragma omp parallel for reduction(+:sum)
  for(i=0; i < n; i++)sum = sum + (a[i] * b[i]);
```

```
  printf("   Suma = %f\n",sum);

}
```

D.11 Ejemplo

En este ejemplo, las iteraciones del loop son repartidas dinámicamente entre el conjunto de hilos. Un hilo ejecutara CHUNK iteraciones.

```
#include <stdio.h>
#include <omp.h>

#define CHUNK   10
#define N       100

main()
{ int nthreads, tid, i, n, chunk;
  float a[N], b[N], c[N];

  /* inicialización */
  for (i=0; i < N; i++)a[i] = b[i] = i * 1.0;
  n = N;
  chunk = CHUNK;

  #pragma omp parallel shared(a,b,c,n,chunk) private(i,nthreads,tid)
  {
    tid = omp_get_thread_num();

    #pragma omp for schedule(dynamic,chunk)
    for(i=0; i < n; i++)
       { c[i] = a[i] + b[i];
         printf("tid= %d i= %d  c[i]= %f\n", tid,i,c[i]);
       }

    if(tid == 0)
      { nthreads = omp_get_num_threads();
        printf("Número de hilos = %d\n", nthreads);
      }

  }  /* fin de la sección en paralelo */

}
```

D.12 Ejemplo - Región en paralelo con una directiva huérfana

Este ejemplo demuestra el producto punto realizado por medio de un loop y una directiva huérfana. El alcance de la variable de reducción es critico.

```
#include <stdio.h>
#include <omp.h>

#define VECLEN 100
float a[VECLEN], b[VECLEN], sum;

float dotprod()
```

```
{ int i,tid;

  tid = omp_get_thread_num();

  #pragma omp for reduction(+:sum)
  for(i=0; i < VECLEN; i++)
     { sum = sum + (a[i]*b[i]);
       printf("  tid= %d i=%d\n",tid,i);
      }

  return(sum);
}


main()
{ int i;

  for(i=0; i < VECLEN; i++)a[i] = b[i] = 1.0 * i;

  sum = 0.0;

  #pragma omp parallel
  sum = dotprod();

  printf("Suma = %f\n",sum);

}
```

D.13 Ejemplo: producto punto de dos vectores

Las iteraciones del loop en paralelo serán distribuidas de igual manera en cada hilo del conjunto (SCHEDULE STATIC).
Al final del loop en paralelo, todos los hilos sumarán los valores de la variable "result" para actualizar la copia global del hilo maestro.

```
#include <stdio.h>
#include <omp.h>

main()
{ int   i, n, chunk;
  float a[100], b[100], result;

  /* Inicialización de los vectores */
  n = 100;
  chunk = 10;
  result = 0.0;

  for(i=0; i < n; i++)
     { a[i] = i * 1.0;
       b[i] = i * 2.0;
      }

  #pragma omp parallel for default(shared) private(i)  \
        schedule(static,chunk) reduction(+:result)

  for (i=0; i < n; i++)result = result + (a[i] * b[i]);
  printf("Resultado final result= %f\n",result);
```

```
}
```

D.14 Ejemplo - Multiplicación Matriz-vector

Este ejemplo, multiplica todos los elementos del renglón de una matriz A con los elementos b(i) y almacena la suma de los productos en el vector c(i). El total se mantiene para toda la matriz.
La actualización de la variable global total es serializada usando la directiva critical.

```
#include <stdio.h>
#include <omp.h>

#define SIZE 10

main ()
{ float A[SIZE][SIZE], b[SIZE], c[SIZE], total;
  int i, j, tid;

  /* Inicialización */
  total = 0.0;
  for(i=0; i < SIZE; i++)
     { for(j=0; j < SIZE; j++)A[i][j] = (j+1) * 1.0;

       b[i] = 1.0 * (i+1);
       c[i] = 0.0;
      }

  printf("\ninicializa los valores de la matriz A y el vector b:\n");
  for(i=0; i < SIZE; i++)
     { printf("  A[%d]= ",i);
       for (j=0; j < SIZE; j++)printf("%.1f ",A[i][j]);
       printf("  b[%d]= %.1f\n",i,b[i]);
      }

printf("\nResultado por hilo/renglón:\n");

  /* Crea un conjunto de hilos y el alcance de las variables */
  #pragma omp parallel shared(A,b,c,total) private(tid,i)
  {
    tid = omp_get_thread_num();

    /* Loop trabajo compartido - distribuye los renglones de la matriz */
    #pragma omp for private(j)
    for(i=0; i < SIZE; i++)
       { for (j=0; j < SIZE; j++)c[i] += (A[i][j] * b[i]);

         /* Actualiza y muestra el total serializado */
         #pragma omp critical
         {
           total = total + c[i];
           printf("  hilo %d renglón %d\t c[%d]=%.2f\t",tid,i,i,c[i]);
           printf("total= %.2f\n",total);
          }

       }   /* fin del loop paralelo i */
```

```
  } /* fin de la construcción en paralelo */
printf("\nMatriz-vector total - suma todo c[] = %.2f\n\n",total);
}
```

Bibliografía

1 Alexandrowicz Z., J. (1969), Chem. Phys., **51**:561
2 Bach Maurice J. (1986), The Design of the UNIX Operating System, Prentice Hall
3 Bradley R. E., indwerS. W, J. (1996), Comput. Chem., **17**:1750
4 Cantu Marco and Tendon Steve (1994), Borland C++ 4.0 Object-Oriented Programming, Random House Electronic Publishing
5 Caracciolo S., Causo M. S., Pelissetto A., J. (1997), Phys. A: Math. Gen., **14**:4939
6 Carili A., da Silva M. A. A., J. (1997), Chem. Phys., **106**:7856
7 Connell Jhon L. & Shafer Linda I. (1995), Object-Oriented Rapid Prototyping, Prentice Hall Inc.
8 Curry David A. (1991), Using C on the UNIX System A Guide to System Programming, O'Reilly & Associates Inc.
9 Dayantis J., Palierne J. F., J. (1991), Chem. Phys., **95**:6088
10 De Gennes P.G. (1979), Scaling Concepts in Polymer Physics, Ithaca, NY: Cornell University Press
11 Degreve L. and Caliry A., J. (1995), Mol. Struct., **94**:123
12 Deitel H. M. y Deitel P. J. (1999), C++ Como Programar (segunda edición), Prentice Hall Inc.
13 Donovan Jhon J. (1986), System programming, McGraw Hill
14 Donovan Madnick (1978), Operating Systems, McGraw Hill
15 Duncan Ray (1988), Advanced MS-DOS programming (second edition), Microsoft press
16 Duplantier B. (1986), Nucl. Phys., **B 275**:319
17 Eizenberg N., Klafter J., J. (1993), Chem. Phys., **99**:3976
18 Everaers R., Grahaman I. S., Zukermann M. J., J. (1995), Phys. A: Math. Gen., **28**:5
19 Fiedler David, Hunter Brece H. (1991), UNIX System V realise 4 Administration, SAMS
20 Fisher M. E. And Gaunt D. S.(1964), Ising model and self-avoiding walks on hypercubical lattice and "high density" expansions. Phys. Rev., 133:A224-A239
21 Garel T., Orland H., J. (1990), Phys. A: Math. Gen., **23**:L621
22 Gottfriend Byron S. (1984), Elements of Stochastic Process Simulation, Prentice Hall
23 Grassberger P., Hegger R., J. (1994), Chem. Phys., **102**:6881
24 Grassberger P., Hegger R., Schafer L., J. (1994), Phys. A: Math. Gen., **22**:7265
25 Gutmman A. J. (1987), On the critical behaviour of self-avoiding walks. J. Phys. A: Math. Gen., 20:1839-1854
26 Guttmann A. J. (1978), On the zero-field susceptibility in the $d = 4$, $n = 0$ limit: analysing for confluent logarithmic singularities. J. Phys. A: Math. Gen., 11:L103-L106
27 Guttmann A. J. (1989b), On the critical behaviour of self-avoiding walks: II. J. Phys. A: Math. Gen., 22:2807-2813
28 Guttmann A. J. and Enting I. G. (1988), The size and number of rings on the square lattice. J. Phys. A: Math. Gen., 21:L165-L172
29 Guttmann A. J. and Wang J. (1991), The extension of self-avoiding random walk series in two dimensions. J. Phys. A: Math. Gen., 24:3107-3109
30 Hekmatpour Sharam (1992), Guia para Programadores en C, Prentice Hall
31 Higgs P. G., Orland H., J. (1991), Chem. Phys., **95**:4506
32 Hull M.E.C, Crookes D., Sweeney P.J. (1994), Parallel processing The transputer and its applications, Addison-Wesley
33 Introduction to Parallel Computing, www.llnl.gov/computing/tutorials/workshops/workshop

34 Kernighan Brian W. & Ritchie Dennis M. (1986), El Lenguaje de Programación C, Prentice Hall
35 Kernighan Brian W. & Ritchie Dennis M. (1987), El Entorno de Programación UNIX, Prentice Hall
36 Kleiman Steve, Shah Devang, Smaalders Bart (1996), Programming with Threads, Prentice Hall
37 Kremer K., Baumgartner A., Binder K., J. (1981), Phys. A: Math. Gen., **15**:2879
38 Kumer Vipin, Grama Ananth, Gupta Ansul (1994), Introduction to Parallel Computing, Design and Analysis of Algorithms, The Benjamin/Cummings Publishing Company
39 Lafore Robert (1988), Microsoft C Programming for the IBM, Howard W. Sams & Company
40 Law Averill M., W. Kelton David, Simulation Modeling and Analysis, McGraw Hill
41 Madras N., Slade G. (1993), The self-avoiding walk, Birkhauser Boston
42 Madras N., Sokal A. D., J. (1988), Stat. Phys., **50**:109
43 Marquez Garcia Francisco Manuel (1996), UNIX Programación Avanzada (segunda edicion), Re-Ma
44 Massand B., Wilensky U. Massar J. P. And Render S. (1992), An extension of the two-dimensional self-avoiding walk series on the square lattice. J. Phys. A: Math. Gen., 25:L365-L369
45 Northup Charles J. (1996), Programming with UNIX threads, Jhon Wiley & Sons
46 OpenMP C and C++ Aplication Program Interface, www.pgroup.com
47 OpenMP, www.llnl.gov/computing/tutorials/workshops/workshop
48 Parker Y. (1983), Multi-microprocessor Systems, Academic Press
49 POSIX threads programming, www.llnl.gov/computing/tutorials/workshops/workshop
50 Press W. H., Flannery B. P., Teukolsky S. A., and Vetterling W. T. (1988), Numerical Recipes in C: The art of Scientific Computing, Cambridge University Press
51 Quinn Michael J. (1994), Parallel computing Theory and practice, McGraw Hill
52 Rapoport D. C. (1984), Phys. Rev. B, **30**:2906
53 Rapoport D. C., J. (1985), Phys. A: Math. Gen., **18**:113
54 Raynal Michael (1984), Algorítmica del Paralelismo, Ediciones Omega S.A
55 Reisdorph Kent (1998), Teach Yourself Borland C++ Builder 3.0 In 14 Days, Sams Publishing
56 Rios Insua David - Rios Insua Sixto (1997), Simulación Métodos Aplicaciones, Editorial Ra-Ma
57 Robbins Kay A. and Robbins Steven (1997), Practical UNIX Progamming A Guide to Currency, Communication and Multithreading, Prentice Hall
58 Rochkind Marc J. (1985), Advaced UNIX Programming, Prentice Hall
59 Rodriguéz-Romo S., Tchjov V., J. (1998), Stat. Phys., **90**:767
60 Schildt Helbert (1990), C: the complete reference (second edition), Osborne McGrow Hill
61 Schildt Herbert (1995), C++ Guia de Autoenseñanza, McGraw Hill
62 Schildt Herbert (1996), C++ para Programadores, McGraw Hill
63 Silberschatz A., Peterson James L., Galvin Peter B. (1992), Operating System Concepts (Third Edition), Addison-Wesley
64 Stallings William (1995), Operating Systems (second edition), Printice Hall
65 Swan Tom (1992), Mastering Borland C++ Covers Borland C++ 3.1, Sams
66 Sykes M. F.,Mckensie D. S., Watts M. G. And Martin J. L. (1972), The number of self-avoiding rings on a lattice. J. Phys. A: Gen. Phys., 5:661-666
67 Tanenbaum Andrew S. (1992), Modern operating systems, Prentice Hall
68 Vahalia Uresh (1996), UNIX Internals The New Frontiers, Prentice Hall