UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Cálculo en paralelo de los diagramas de Liapunov-Markus

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

A C T U A R I O

P R E S E N T A :

Carlos Ernesto López Natarén

DIRECTOR DE TESIS: Dr. Pedro Eduardo Miramontes Vidal

FACULTAD DE CIENCIAS
UNAM

VNIVERSIDAD NACIONAL
AVTONOMA DE
MEXICO

ACT. MAURICIO AGUILAR GONZÁLEZ
Jefe de la División de Estudios Profesionales de la
Facultad de Ciencias.
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

"Cálculo en paralelo de los diagramas de Liapunov-Markus"

realizado por Carlos Ernesto López Natarén

con número de cuenta 9653163-1 , **quien cubrió los créditos de la carrera de**: Actuaría

Dicho trabajo cuenta con nuestro voto aprobatorio.

A t e n t a m e n t e

Director de Tesis	
Propietario	Dr. Pedro Eduardo Miramontes Vidal
Propietario	Dr. Germinal Cocho Gil
Propietario	M. en C. Jorge Luis Ortega Arjona
Suplente	M. en C. Enrique Cruz Martínez
Suplente	M. en C. María Guadalupe Elena Ibargüengoitia González

Consejo Departamental de Matemáticas

M. en C. José Antonio Flores Díaz.

FACULTAD DE CIENCIAS
CONSEJO DEPARTAMENTAL

Cálculo en paralelo de los diagramas de Liapunov-Markus

Carlos Ernesto López Natarén

A mamá, papá y mi hermano César.

Quisiera agradecer a mi asesor, el Dr. Pedro Miramontes Vidal, quien ha sido un gran guía para mi, así también como al Dr. Octavio Miramontes Vidal, por todo el apoyo recibido en el Instituto de Física. También a mis sinodales M. en C. Jorge Luis Ortega Arjona, M. en C. Enrique Cruz Martínez, Dr. Germinal Cocho Gil, M. en C. María Gpe. Ibargüengoitia González, por haber revisado este trabajo y darme sugerencias y correcciones. A mi hermano, César Octavio López Natarén, por sus sugerencias en el área de computación que fueron de mucha ayuda. A Gabriel Cárdenas Díaz Ordaz por su entusiasmo y ayuda en la visualización y a mi querida universidad, la Universidad Nacional Autónoma de México, a quien no creo poder retribuir todo lo que me ha dado. A todos, muchas gracias.

Índice general

Índice de figuras

Introducción

Los sistemas dinámicos se encuentran en cualquier rama del conocimiento que estudie el cambio, movimiento y evolución de algún fenómeno, desde la dinámica de poblaciones, hasta la biología de plantas. El estudio de estos sistemas ha mostrado en las décadas pasadas un fenómeno hasta ese entonces no conocido o difícil de determinar, este tipo de fenómeno es conoce como caos.

Por otro lado, la computación, y en el tema de esta tesis, las supercomputadoras están cada vez afectando más nuestra vida diaria de una u otra manera. Las supercomputadoras pueden alterar el precio de la fruta, o también pueden ser la razón por la que los pantalones que traemos se ofrezcan al mercado. Hay problemas más y más complejos que requieren más y más poder de cómputo. Por ejemplo simulaciones financieras y económicas [Mantegna y Stanley, 1999] (econofísica) o el cálculo de órbitas de sistemas dinámicos que tardarían demasiado en una computadora común y corriente.

La explosión del uso de sistemas libres[1] a partir de la revolución causada por la aparición del sistema operativo Linux, también provocó otras pequeñas revoluciones en otros ámbitos de la computación actual. De manera especial en el área de supercómputo; gracias a este tipo de sistemas libres, la investigación en supercómputo se extendió más rápidamente a ámbitos académicos de recursos limitados, en particular el uso de sistemas Beowulf[2] en universidades e institutos, con lo que también países del tercer mundo tienen ahora una oportunidad de desarrollarse en este tipo de áreas para superar el atraso.

En este trabajo, se está interesado en la generación de un tipo particular de imágenes o diagramas, los cuales pueden ser pensados como *mapas del*

[1]Software con cuatro libertades básicas: Libertad de usar los programas para cualquier propósito, libertad de acceso al código fuente, libertad a distribuir copias del programa, libertad a hacer públicas las mejoras al tener acceso al código fuente.

[2]Beowulf, escrito en viejo inglés en algún momento antes de el siglo X después de Cristo, narra las aventuras de un gran guerrero escandinavo del siglo VI.

caos. Estos fueron nombrados en honor al creador del exponente de Liapunov[3]. Este tipo de diagramas toman una cantidad considerable de tiempo en generarse en nuestras computadoras actuales. Por lo que este trabajo intenta dar una solución paralela al cálculo de éstos por medio de un programa paralelo, para tener un mecanismo en el cual se pueda llevar a cabo esta tarea de la manera más rápida posible dados los recursos de cómputo con los que se cuente, además de que la estructura del problema es tal que puede ayudarnos también a hacer una abstracción de éste para tratar de solucionar problemas con condiciones similares.

Con tal objetivo, se pretende acercar al lector tanto a la terminología básica de los sistemas dinámicos discretos, así como a la historia y tendencia actual del supercómputo y en particular de la arquitectura Beowulf y al modelo de envío de mensajes, con la siguiente organización de los temas:

En el capítulo uno se hace una breve introducción a los sistemas dinámicos discretos, mapeos unidimensionales, órbitas y un acercamiento a los conceptos del caos, en especial, a una de sus características más notables, la sensibilidad a las condiciones iniciales.

El capítulo dos se concentra en el estudio de los diagramas de Liapunov-Markus, el método con el cual se pueden generar este tipo de diagramas y una primera implementación serial que se basa la implementación en paralelo.

En el capítulo tres se describe la arquitectura Beowulf y se presenta como una alternativa confiable y con proyección a futuro para el cómputo en paralelo, además de las ventajas y desventajas de su uso.

En el capítulo cuatro se expone al lector al modelo de envío de mensajes, el cual es el que se usará para paralelizar el programa visto en el capítulo dos, con conceptos básicos del envío de mensajes suficientes para intentar paralelizar el programa.

En el capítulo cinco se implementa el programa en paralelo usando técnicas para paralelizar este programa, así como un análisis de escalabilidad con la aceleración como la base para este análisis.

En las conclusiones generales del trabajo, se presentan sugerencias para mejorar el programa, se dan algunos detalles que fueron ignorados y que podrían ser importantes al intentar mapear el problema para resolver otros.

Finalmente, el programa para generar los archivos de datos y el programa para visualizarlos pueden ser encontrados para su uso en http://natorro.dyndns.org/tesis/lyapunov.tar.gz, o puedo ser contactado a través de correo electrónico en natorro@fisica.unam.mx.

[3]Alexandr Mijailovich Liapunov (1857-1918).

Capítulo 1

Sistemas dinámicos

1.1. Sistemas dinámicos

Un sistema dinámico se define como aquel que tiene una función determinista cuyos estados evolucionan a través del tiempo. El tiempo puede ser una variable continua, o puede ser una variable discreta con valores enteros.

Estamos interesados en sistemas dinámicos discretos deterministas unidimensionales, es decir, el tiempo tiene valores discretos y generalmente se denota con la variable $n = 0, 1, 2, ...$ sus valores son enteros positivos. Estos modelos se adaptan bien a situaciones donde ocurren cambios en tiempos específicos en vez de continuos, por ejemplo, el crecimiento de una población en tiempos dados tales como el final anual de un ciclo reproductivo.

Los sistemas dinámicos discretos a menudo implican el procesos de iteración, es decir, aplicar la función o procedimiento muchas veces.

Comencemos con una definición simple de un mapeo general:

$$x_{n+1} = f(x_n)$$

donde $x_n$tiene N componentes, $x_n = (x_n^{(1)}, x_n^{(2)}, ..., x_n^{(N)})$. Dada una condición inicial x_0, obtenemos el estado del sistema en el tiempo $n = 1$ sacando $x_1 = f(x_0)$. Al determinar x_1, podemos entonces determinar el estado al tiempo $n = 2$ haciendo $x_2 = f(x_1)$ y así sucesivamente. Por lo tanto, dada una condición inicial x_0, nosotros podemos generar una *órbita* (o trayectoria) del sistema discreto: $x_0, x_1, x_2, ...$

1.1.1. Mapas unidimensionales

Los mapeos unidimensionales no invertibles son los sistemas dinámicos más simples capaces de mostrar caos, sistemas que son sensibles a las condiciones iniciales. Éstos sirven entonces como un punto de inicio para el estudio del caos. Incluso una proporción de los fenómenos encontrados en sistemas de mayores dimensiones están presentes de alguna forma en los mapeos unidimensionales.

Un mapeo unidimensional está presente cuando la dimensión de x_n es igual a uno.

Tomemos por ejemplo el mapeo logístico:

$$x_{n+1} = rx_n(1 - x_n) \tag{1.1}$$

Esta ecuación es ideal como un ejemplo que genera caos porque es simple y muestra varias características clave, como veremos en las siguientes secciones.

Si escogemos un número real x_0 en el intervalo $[0, 1]$, el cual como ya vimos es el punto inicial o condición inicial, y aplicamos la función f para obtener:

$$x_1 = f(x_0) = rx_0(1 - x_0)$$

Por ejemplo, supongamos que $x_0 = 0.3$ y $r = 4$, entonces tendríamos:

$$x_1 = f(x_0) = 4(0.3)(1 - 0.3) = 0.84$$

Ahora, si aplicamos de nuevo la función a x_1, tendríamos:

$$x_2 = f(x_1) = f(f(x_0))$$

si seguimos repitiendo este proceso tendremos:

$$x_3 = f(x_2) = f(f(f(x_0)))$$

$$x_4 = f(x_3) = f(f(f(f(x_0))))$$

y así sucesivamente, por lo que para cada número $n = 0, 1, 2...$, tendríamos:

$$x_{n+1} = f(x_n)$$

A este proceso se le conoce como iterar la función f con la condición inicial x_0, como habíamos establecido en la primera sección de este capítulo, la secuencia de números:

$$x_0, x_1, x_2, x_3, ...$$

es conocida como la órbita de la condición inicial x_0.

1.1.2. Órbitas

Al estudiar estos sistemas dinámicos se está interesado en el comportamiento de las órbitas, ya que ésto ayudaría a describir a largo plazo el sistema en estudio.

El mapeo logístico es un modelo de crecimiento para algunas poblaciones de animales basado en el modelo de crecimiento exponencial ($x_{n+1} = rx_n$) pero con un factor adicional en el lado derecho de la función ($1 - x_n$).

Ahora, exploraremos un poco el comportamiento de este sistema, tenemos que si tomamos un valor pequeño para x_n (por decir, un valor positivo muy cerca del cero), la cantidad $1 - x_n$ está cerca del 1, por lo que la función 1.1 está muy cerca de rx_n. Por lo tanto la población se comporta de manera proporcional a la manera en que x_n lo está haciendo, aunque no en proporción directa a x_n.

De manera similar, a valores relativamente grandes de x_n (por decir algo, muy cerca de valores menores del valor máximo 1) la cantidad $1 - x_n$ es pequeña (cercana a cero), en otras palabras, el crecimiento es pequeño.

La gráfica de esta ecuación es una parábola invertida que pasa por $(0, 0)$, $(\frac{1}{2}, 1)$,$(1, 0)$ cuando $r = 4$. Se hablará sobre como se puede mover este valor un poco más adelante, como lo muestra la figura 1.1.

Dado un x_n en la ordenada el valor de x_{n+1} se encuentra en la abscisa (un mapeo de una dimensión), un punto puede ser calculado si x_n está entre 0 y 1 porque si tenemos puntos que son mayores que 1 los puntos no pueden existir porque darían valores negativos de la parábola, los cuales no nos interesan, ya que no existen poblaciones negativas.

La constante r regula la pendiente y la altura de la parábola sobre la abscisa. Entre más grande sea r, más alto será el pico de la parábola (es decir, más grande la población). La altura del pico mide $r/4$ unidades de x_{n+1}, por lo tanto, como x_{n+1} se encuentra en el intervalo $[0, 1]$, el valor de r para aplicaciones de la ecuación logística, sólo puede estar en el rango de 0 a 4. Si r se excede de 4 las iteraciones de la ecuación logística producen valores que son imposibles, mayores que 1 o menores que 0. Estas limitaciones en x_n, x_{n+1} y r significan que para este modelo poblacional, la parábola inicia suavemente del origen a un pico de máxima población en $x_n = 0.5$ y $x_{n+1} = r/4$. Mientras x_n crece más allá de 0.5, la curva cae, es decir, la población retrocede. Es interesante observar que para distintos valores

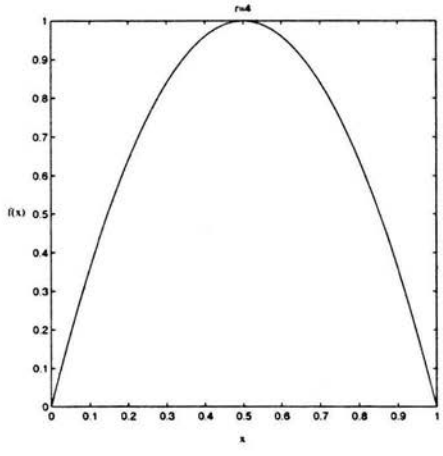

Figura 1.1: Gráfica de la función $f(x) = rx(1-x)$ con $r = 4$

de r se tienen comportamientos diferentes. esos se discuten en la siguiente sección.

Puntos estables

Haciendo $r = 0.95$ (recordemos que $0 < r < 4$), la ecuación 1.1 se vuelve entonces:

$$x_{n+1} = 0.95x_n(1 - x_n)$$

Supongamos que tenemos un valor inicial $x_0 = 0.4$. Ésto en términos del modelo poblacional quiere decir que estamos al 40 % de la población máxima ya que $0 < x_n < 1$, insertando este valor inicial, tenemos que la primera iteración de la ecuación nos da $x_1 = 0.228$, para la segunda iteración (posiblemente la población para el siguiente año) introducimos 0.228 en la ecuación logística y el nuevo valor de $x_2 = 0.167$, valores subsecuentes para x_t (iteraciones) son $0.132, 0.109, 0.092, 0.080$ y así sucesivamente, hasta converger a cero, sugiriendo que la población se extingue. La figura 1.2 muestra la serie de tiempo y el diagrama de fase (o diagrama de red) para este valor inicial de la función con el valor de r asociado.

Cálculos similares nos mostrarían que para cualquier otro valor inicial x_0

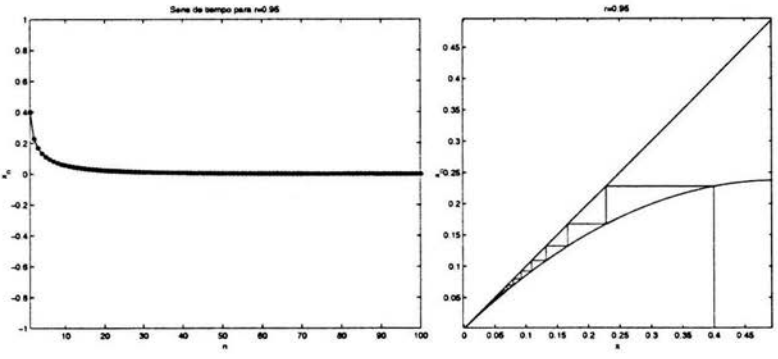

Figura 1.2: Serie de tiempo y diagrama de fase para la ecuación logística con $x_0 = 0.4$ y $r = 0.95$

y $r = 0.95$ que el valor de los puntos de la serie de tiempo u órbita todavía convergen a cero. En cierto sentido, ese punto (cero) atrae los puntos de la órbita, por lo que es conocido como un atractor y se simboliza con x^*, un atractor es el punto de espacio fase o puntos que con el curso del tiempo (iteraciones) atraen todas las trayectorias que emanan de cierto rango de condiciones iniciales también conocido como cuenca de atracción.

Hagamos ahora a $r = 1.4$ y tomemos arbitrariamente $x_0 = 0.06$, podemos observar ahora en la figura 1.3 la manera en que se comporta la órbita bajo estas condiciones iniciales es de nuevo convergente a un punto fijo $x^* = 0.286$. Se puede experimentar de nuevo y podemos ver que sin importar el valor de x_0 que se de al inicio, la órbita siempre nos llevará al mismo punto atractor.

Ahora, cambiando el valor de nuestro parámetro r haciendo $r = 2.8$, repetimos el mismo procedimiento con $x_0 = 0.4$ y hacemos una serie de iteraciones, los valores convergen en este caso a $x^* = 0.643$, ésto lo podemos observar de nueva cuenta en la figura 1.4.

Estas tres figuras muestran varias características interesantes:

- Cada x_0 da inicio a su propia trayectoria, pero todas las trayectorias para un valor de r dado llevan al mismo punto atractor. Esta implicación en términos del modelo poblacional quiere decir, que sin importar la población inicial en estudio, ésta siempre llevará a la misma población eventual. Por supuesto en este caso, para valores de $0 < r < 3$.

- El atractor es cero si $r < 1$. Si r es mayor que 1 y crece hasta valores

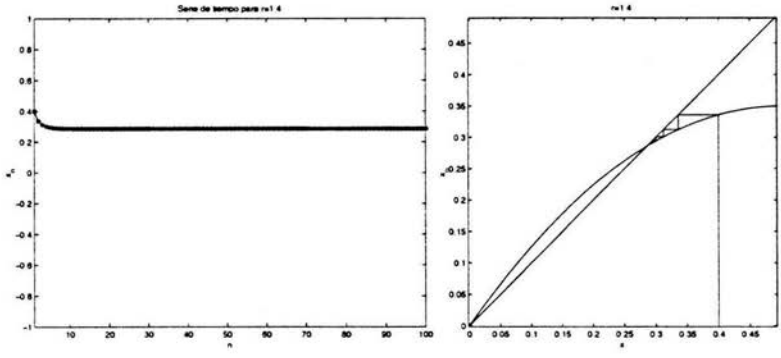

Figura 1.3: Serie de tiempo y diagrama de fase para la ecuación logística con $x_0 = 0.06$ y $r = 1.4$.

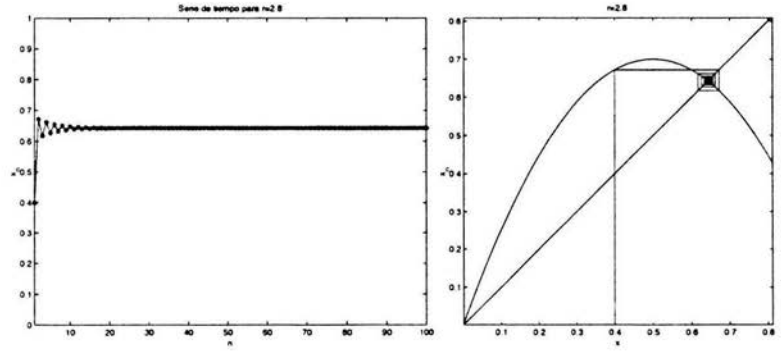

Figura 1.4: Serie de tiempo y diagrama de fase para la ecuación logística con $x_0 = 0.4$ y $r = 2.8$

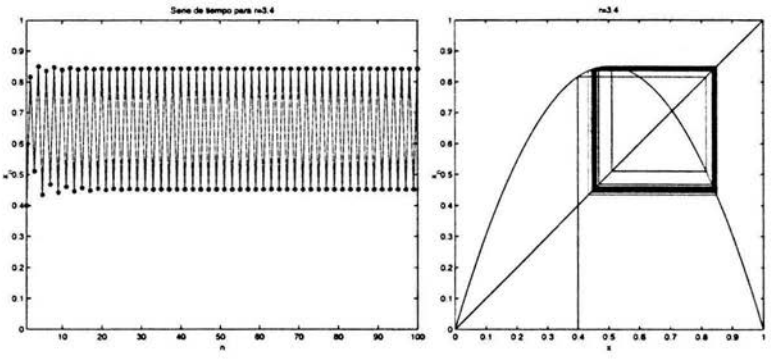

Figura 1.5: Serie de tiempo y diagrama de fase para la ecuación logística con $x_0 = 0.4$ y $r = 3.4$

muy cerca de 3 el atractor se incrementa de cero a 0.667.

- Se puede encontrar el atractor gráficamente para valores de $0 < r < 3$.
- Para $0 < r < 3$, el punto fijo del atractor reside en lugares predecibles y diferentes en el espacio fase relativos al pico de la parábola.
- Cuando $r < 2$, la trayectoria converge al atractor, dibujándose más cerca de un lado únicamente. En contraste, cuando $r > 2$, la trayectoria converge al atractor primero moviéndose en una vecindad general y luego regresando una y otra vez hasta caer en el atractor.

Puntos periódicos

Ahora, usando valores más grandes para r, si r es 3.0 o mayor, sucede algo no tan simple: la trayectoria ahora no converge a un único valor x^*. El comportamiento de la trayectoria es más sensible al valor de r elegido. Por ejemplo, si tomamos $r = 3.4$, sin importar el valor de x_0, los puntos de la órbita convergen no a un sólo valor x^*, sino a dos, en este caso $x_1^* \approx 0.452$ y $x_2^* \approx 0.842$, como se puede ver en la figura 1.5. Este comportamiento implica en nuestro modelo poblacional que un año hay un 45 % de la población máxima y el siguiente hay un 82 % y así cada dos años hay un ciclo con estos valores de la población.

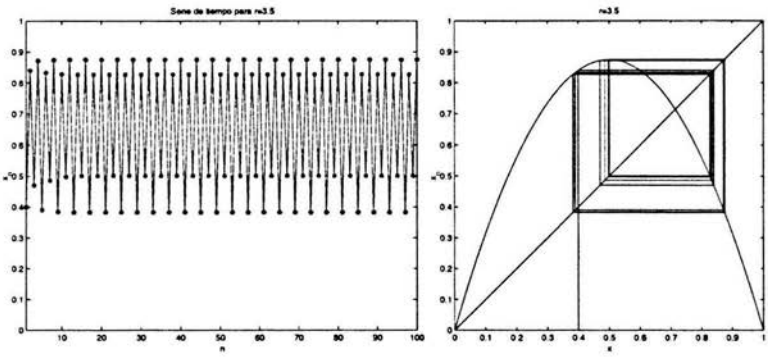

Figura 1.6: Serie de tiempo y diagrama de fase para la ecuación logística con $x_0 = 0.4$ y $r = 3.5$

Estos dos valores juntos se dice que constituyen el atractor o estado final. Así el parámetro r o parámetro de control nos muestra un tipo más complejo de atractor en el modelo de la ecuación logística, de regreso con nuestro valor de $r = 3.4$ vemos que tenemos un *atractor de dos puntos*, es decir, un atractor o estado final que consiste de dos puntos periódicos.

Si continuamos incrementando el valor de r, digamos ahora hacemos $r = 3.5$ podemos observar por la figura 1.6 que no hay dos puntos periódicos, sino cuatro. Estos cuatro puntos particulares nos indican que la población en estudio tiene un ciclo de cuatro años.

Al número de valores x^* en un ciclo se le conoce como el *periodo* del atractor. Si seguimos experimentando con los valores de r podemos ver que entre más vayamos haciendo crecer el parámetro de control, el número de x^* va duplicándose cada vez, es decir, en el principio eran 2, luego 4, luego 8 y así sucesivamente. Es decir, el resultado es que cada conjunto de puntos atractores tiene dos veces x^* puntos atractores que el anterior, a los puntos donde ocurre este cambio en el periodo del atractor se le conoce como un punto de *bifurcación*, una bifurcación es un cambio cualitativo repentino en el comportamiento del sistema, el cambio puede ser sutil o catastrófico, al tipo de bifurcación que observamos en el mapeo logístico se le conoce como una bifurcación de doble periodo. La bifurcación puede no ser de doble periodo, muchas veces aparecen tres puntos y luego nueve y así sucesivamente y muchas veces con otros periodos, por lo que cuando se refiere a un punto de

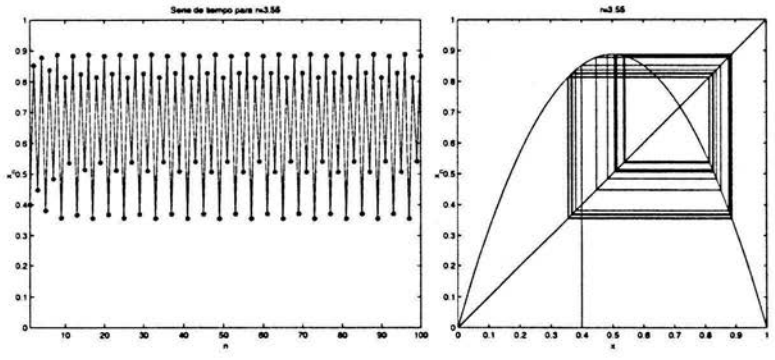

Figura 1.7: Serie de tiempo y diagrama de fase para la ecuación logística con $x_0 = 0.4$ y $r = 3.55$

bifurcación, se puede referir a que aparecen dos puntos atractores o quizá más [Devaney, 1987].

En valores de r alrededor de 3.55 y 3.56 los periodos comienzan a doblarse con pequeños incrementos en r, tales incrementos en el orden de los cinco o seis lugares decimales, ésto es, el rango de estabilidad de cualquier periodicidad se vuelve drásticamente más corta.

Al valor de r de 3.57 el número de periodos se vuelve infinitamente grande, el intervalo de 3.57 a 4.0 tiene caos, es una región donde la ruta de la trayectoria para valores de r parecen ser erráticos, sin aparente orden (figura 1.7). Otros valores de r dentro de este rango producen ventanas de aparente estabilidad.

Por todo ésto, la ecuación logística tiene lo que se conoce como una ruta de doblamiento de periodo al caos, el parámetro r en el caso de esta ecuación tiene cuatro casos distintos:

- $r < 1$, el atractor es un punto fijo que tiene valor de cero.
- $1 < r < 3$, el atractor es un punto fijo que tiene un valor mayor que 0 pero menor que 0.667.
- $3 \leq r < 3.57$, el desdoblamiento de periodo ocurre, el atractor consiste de $2, 4, 8, etc.$ puntos periódicos, mientras r crece dentro del rango.
- $3.57 < r < 4.0$, estamos en la región donde caos y ventanas de pe-

riodicidad comparten un espacio, un atractor acá puede ser errático (caótico, con infinitos puntos de atracción) o estable, acá podemos observar que hay infinitas ventanas donde por momentos hay pocos puntos de atracción.

Con ésto, se ha observado que el parámetro de control r juega un papel importante en este tipo de modelos, ya que la evolución del sistema y la observación del fenómeno dependen de este parámetro, ya que el sistema es determinista, no lineal y dinámico. Una variable puede tener una cantidad increíble de posibles evoluciones dependiendo del valor del parámetro de control. El caos es inevitable cuando se itera la ecuación logística y ciertas ecuaciones más. No se tiene opción en esos valores del parámetro de control.

Incluso, el rango completo del parámetro tiene ciertos niveles. A cada nivel crítico, el sistema cambia su comportamiento, repentina y drásticamente, con el incremento del parámetro de control. La trayectoria a cada nivel es mas compleja y menos ordenada.

Diagrama de bifurcación y caos

Una forma gráfica muy útil para observar los puntos de atracción y el caos en la ecuación logística y otros sistemas, es el diagrama de bifurcación. En este tipo de diagrama se observa la ruta al caos en los sistemas con un parámetro de control r.

Si se itera 100 veces para eliminar puntos transitorios y se grafica los siguientes 200 puntos para cada valor de r se obtiene la figura 1.8. Esta figura permite observar de manera única la forma en que las órbitas se comportan para valores dados de r. Ésto ya se ha podido observar en las gráficas anteriores donde se mostraba la serie de tiempo para cada valor r determinado.

1.2. Caos

1.2.1. Sensibilidad a las condiciones iniciales

Ahora, añadiendo un pequeño error a la condición inicial x_0, como por ejemplo:

$$\mu_0 = x_0 + 0.001$$

observando la órbita de $\mu_0, \mu_1, \mu_2, ...$ se esperaría que los errores pequeños no fueran importantes, pero ésto dista mucho de ser cierto, ya que la función

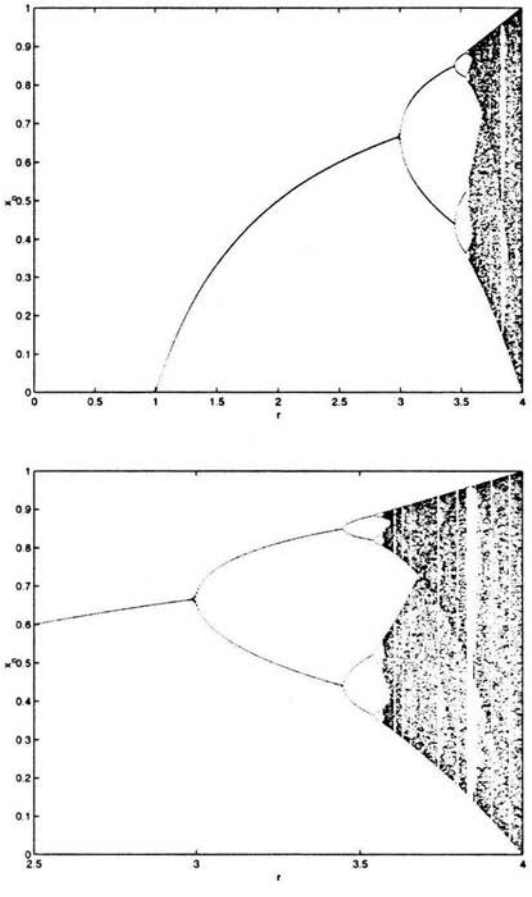

Figura 1.8: Diagrama de bifurcación de la función $x_{n+1} = rx_n(1 - x_n)$ y un acercamiento

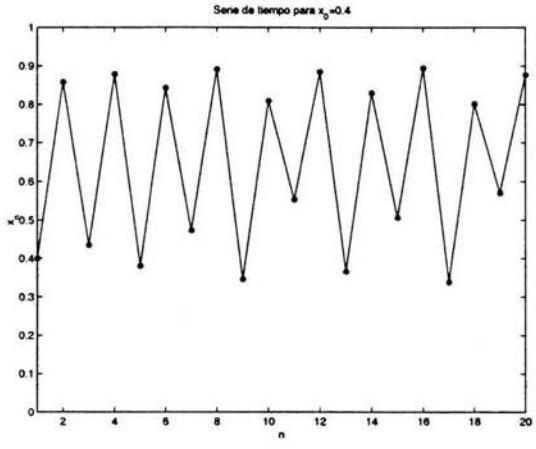

Figura 1.9: Órbita del punto inicial x_0

tiene a amplificar los pequeños errores hasta hacerlos muy grandes dentro del régimen caótico. Así, la diferencia o error entre dos órbitas dada por:

$$(\mu_0 - x_0), (\mu_1 - x_1), (\mu_2 - x_2), ...$$

tiende a ser tan grande como x_k mismo.

La figura 1.9 ilustra este fenómeno. La gráfica es la órbita de la condición inicial x_0 y su órbita los cuales son los puntos x_k. A este tipo de gráfica se le conoce como la gráfica de la serie de tiempo o de la órbita. La figura 1.10 muestra ahora en comparación las órbitas de ambos puntos, el que tiene el pequeño error y el que no contiene al error, que están muy cerca uno del otro al inicio de la iteración, es decir, añadimos un pequeño error como se mostró en el primer párrafo, y se puede observar que en los primero pasos, del lado izquierdo de la gráfica de la serie de tiempo, las órbitas permanecen cerca una de otra durante algunos pasos, pero después de algunos iteraciones más, hacia el lado derecho de la gráfica de la serie de tiempo, las órbitas empiezan a tener distintas valores. El error crece tanto que las órbitas al final son completamente diferentes.

La figura 1.11 muestra como el error va creciendo en cada paso. Nótese como el error comienza muy pequeño, pero conforme se va avanzando en la iteración, este error se ha amplificado tanto como los valores mismos de las

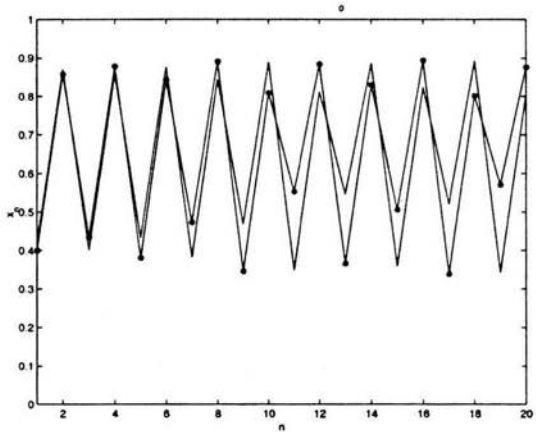

Figura 1.10: Órbitas de dos puntos iniciales x_0 y μ_0

órbitas. De hecho, se puede demostrar que no importan que tan pequeño hagamos el error inicial, éste tenderá a ser amplificado. Este comportamiento, en el cual los errores pequeños tienden a ser amplificados hasta ser tan grandes como los valores de las órbitas es conocido como dependencia sensible a las condiciones iniciales. Comúnmente este fenómeno es conocido como el efecto mariposa [Lorenz, 1996], llamado así por la idea de que un pequeño aleteo de mariposa podría ser ese error que hemos analizado y que tiende a crecer tanto que puede provocar eventualmente un huracán en algún lugar de la Tierra. Ésto es porque el clima es sensible a las condiciones iniciales de la misma manera en que la función descrita hasta ahora lo es, no importa que tan exactas se hagan las mediciones, los errores en las mediciones por muy pequeños que sean serán amplificados hasta que las predicciones a largo plazo sean completamente impredecibles. Este efecto es uno de las características más importantes del caos, todos los sistemas caóticos tienen esta característica (pero no necesariamente todos los sistemas sensibles a condiciones iniciales son caóticos).

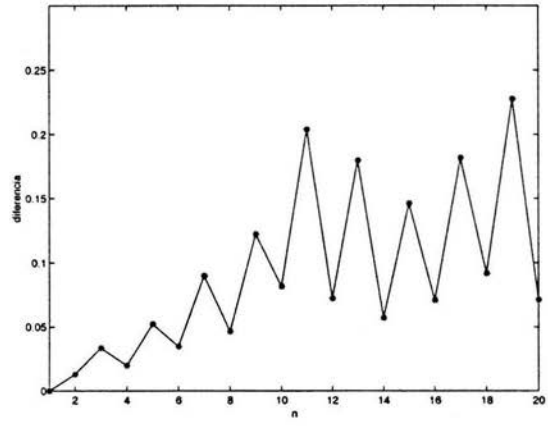

Figura 1.11: Gráfica del error entre las dos órbitas

1.3. El Exponente de Liapunov

El exponente de Liapunov es usado para medir qué tanto los errores pequeños son amplificados. En otras palabras, éste mide que tan sensible es el sistema a errores.

A manera ilustrativa, recordemos que se inicia con un error inicial, sea este error E_0, así que tendríamos:

$$\mu_0 = x_0 + E_0$$

es decir, la condición inicial x_0 y el error E_0. Entonces, comparamos la órbita "exacta" $x_0, x_1, x_2, x_3, ...$contra la órbita con el error $\mu_0, \mu_1, \mu_2, ...$ y se observa de qué manera la diferencia entre estas crece. Es decir, se sigue la secuencia de errores:

$$\begin{aligned} E_0 &= \mu_0 - x_0 \\ E_1 &= \mu_1 - x_1 \\ E_2 &= \mu_2 - x_2 \\ &\cdots \end{aligned}$$

La figura 1.12 muestra gráficamente lo que sucede. El error E_0 crece para dar el error E_1 en el siguiente paso de la función f descrita.

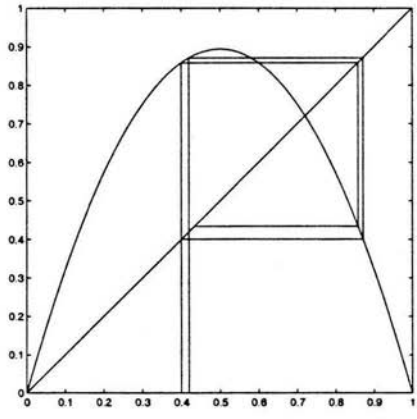

Figura 1.12: Crecimiento del error gráficamente

Para determinar que tanto es amplificado el error en cada iteración k se calcula:

$$\left|\frac{E_{k+1}}{E_k}\right|$$

es decir, con esta fórmula se observa qué tan grande es el siguiente error E_k comparado con el error actual E_k. Después de n pasos, el error inicial E_0 habrá sido amplificado por un factor de:

$$\left|\frac{E_n}{E_0}\right| = \left|\frac{E_n}{E_{n-1}}\right| \cdot \left|\frac{E_{n-1}}{E_{n-2}}\right| \cdot \cdots \cdot \left|\frac{E_1}{E_0}\right|$$

Supónga que se tuviera la siguiente función lineal:

$$g(x) = cx$$

donde c es una constante mayor que 1. Entonces, cada error es simplemente amplificado c veces en cada iteración:

$$\begin{aligned} g(x+E) &= c(x+E) \\ &= cx + cE \\ &= g(x) + cE; \\ g(g(x)+cE) &= c(g(x)+cE) \\ &= cg(x) + (c^2)E \\ &= g(g(x)) + (c^2)E, \end{aligned}$$

por lo que después de n pasos, el error inicial ha sido amplificado:

$$\left|\frac{E_n}{E_0}\right| = c^n$$

nótese que si c fuera menor que 1, el error sería reducido en vez de incrementado. Si se quiere encontrar cuál es el valor de c para tal función lineal g, entonces se tiene:

$$\ln(c) = \frac{1}{n} \ln\left(\left|\frac{E_n}{E_0}\right|\right)$$

Esta es la idea detrás del exponente de Liapunov. Se toma una función general f y usamos esta misma fórmula para observar cómo los errores pequeños tienden a ser amplificados. Se hacen más y más iteraciones, es decir se hace n cada vez más grande, y se calculan los valores correspondientes a c. Si, al hacer crecer n estos valores convergen a una constante, entonces ésto sugeriría que el error tiende a crecer en promedio c^n. Tomando el logaritmo se convierte en la siguiente suma:

$$\begin{aligned} \frac{1}{n} \ln\left(\left|\frac{E_n}{E_0}\right|\right) &= \frac{1}{n} \ln\left(\left|\frac{E_n}{E_{n-1}}\right| \cdot \left|\frac{E_{n-1}}{E_{n-2}}\right| \cdot \ldots\ldots \left|\frac{E_1}{E_1}\right|\right) \\ &= \left(\frac{1}{n}\right) \sum_{k=1}^{k=n} \ln\left(\left|\frac{E_k}{E_{k-1}}\right|\right) \end{aligned}$$

el exponente de Liapunov es definido como el valor límite de la cantidad expresada arriba, al aproximar E_0 a cero y el número de la iteración, n, es aproximado a infinito.

Ahora, de la definición del error, tenemos que:

$$E_k = f(x_{k-1} + E_{k-1}) - f(x_{k-1})$$

por lo tanto

$$\frac{E_k}{E_{k-1}} = \frac{f(x_{k-1} + E_{k-1}) - f(x_{k-1})}{E_{k-1}}$$

Ahora, suponiendo que la función f es diferenciable, en este caso, se define $h = E_{k-1}$ y $x = x_{k-1}$, entonces se tiene:

$$\frac{E_k}{E_{k-1}} = \frac{f(x_{k-1} + E_{k-1}) - f(x_{k-1})}{h}$$

Si se deja h tender hacia cero, entonces, esta es justo la expresión de la derivada, por lo que si $h \to 0$ se obtiene:

$$\frac{E_k}{E_{k-1}} = f'(x_{k-1})$$

entonces, finalmente se llega a:

$$\frac{1}{n} \ln\left(\left|\frac{E_n}{E_0}\right|\right) = \left(\frac{1}{n}\right) \sum_{k=1}^{k=n} \ln\left(\left|\frac{E_k}{E_{k-1}}\right|\right)$$

$$= \left(\frac{1}{n}\right) \sum_{k=1}^{k=n} \ln\left(\left|f'(x_{k-1})\right|\right) = \lambda \qquad (1.2)$$

Si este número es positivo, entonces los errores pequeños en la condición inicia x_n tenderán a ser amplificados y el sistema mostrará sensibilidad. Si el número es negativo, entonces los errores pequeños tienden a ser reducidos, por lo que el sistema mostrará estabilidad. Es de esta manera como el exponente de Liapunov ayuda a medir la estabilidad de un sistema.

Ahora, sintetizando, se tiene que si el valor del exponente de Liapunov es:

$\lambda < 0$ Las órbitas son atraídas a un punto fijo estable o a una órbita periódica estable. Los exponentes de Liapunov negativos son característicos de sistemas no conservativos y disipativos por ejemplo. Este tipo de sistemas exhiben estabilidad asintótica; entre más negativo sea el exponente, mayor será la estabilidad. Puntos fijos y puntos periódicos superestables tienen un exponente de Liapunov $\lambda = -\infty$.

$\lambda = 0$ La órbita es un punto fijo (o un punto fijo eventual). Un exponente de Liapunov cero, indica que el sistema está de algún modo de estado constante. Un sistema físico con este exponente es conservativo, este tipo de sistemas exhiben estabilidad de Liapunov. Como ejemplo tenemos dos osciladores armónicos simples con diferentes amplitudes.

$\lambda > 0$ La órbita es inestable y caótica. Puntos cercanos, no importando que tan cerca estén, se separarán a una distancia arbitraria. Todos las vecindades en el espacio fase eventualmente serán visitados. Estos puntos se dicen que son inestables.

Capítulo 2

Diagramas de Liapunov-Markus

Ya se ha visto que sistemas simples son capaces de tener comportamiento tanto ordenado como caótico y que estos sistemas hacen estas transiciones cuando un parámetro varía y estos cambios son visualizados en las bifurcaciones.

Una manera de entender como cambia el comportamiento de una función al mover el parámetro es graficar el valor del exponente de Liapunov para algún valor inicial x_0 contra el parámetro r. La figura 2.1 muestra este tipo de gráfica para la función $f(x_{n+1}) = rx_n(1 - x_n)$ y condición inicial $x_0 = 0.5$, son notables los valores muy negativos de esta gráfica, en realidad estos valores muy negativos van hasta menos infinito, mostrando que el comportamiento de la función bajo ese parámetro r es extremadamente estable, recordemos que hemos calculado un estimado del logaritmo del error, si este valor es muy grande y negativo entonces el error es muy pequeño, lo que indica que el error tiende a desvanecerse y que la órbita del sistema es superestable.

En algunos puntos, la gráfica toca al eje r, es decir, el exponente de Liapunov tiene un valor de cero, ésto indica que con estos parámetros se obtiene una órbita que está en el límite del comportamiento estable e inestable, es aquí donde se suceden cambios de repente, conocidos como bifurcaciones, y es justo en estos puntos donde se suceden cambios en los comportamientos de las órbitas de estables a caóticas.

El valor del exponente en la gráfica permanece debajo del cero hasta que alcanza un punto cerca del lado derecho, aquí es donde la sensibilidad aparece por primera vez, ésto indica que al ser el exponente de Liapunov

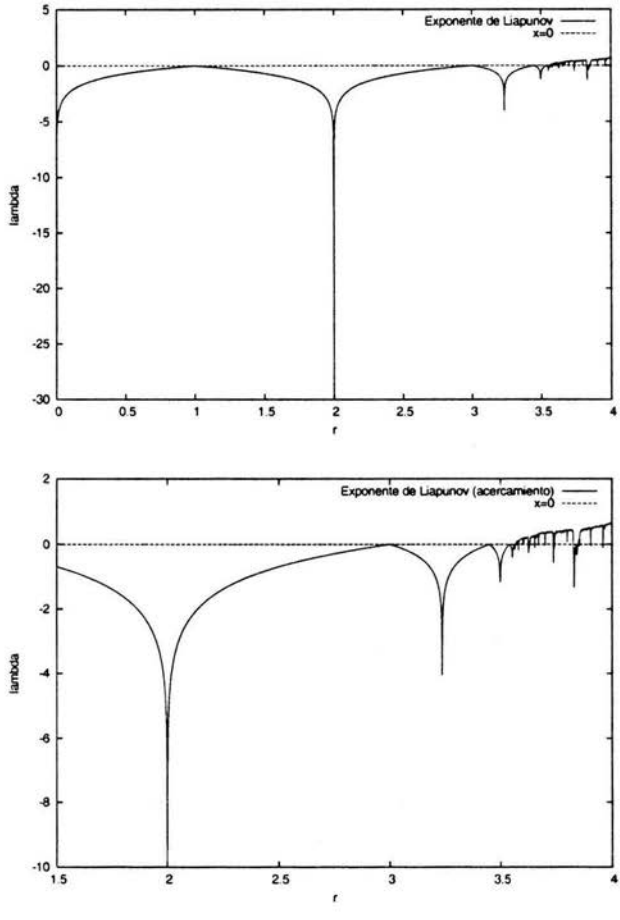

Figura 2.1: Exponente de Liapunov λ vs. parámetro r y un acercamiento

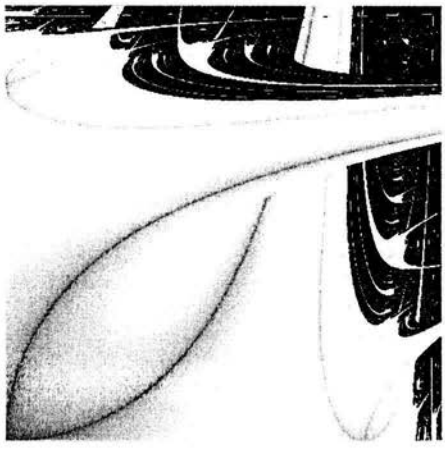

Figura 2.2: Diagrama de Liapunov para el mapeo logístico con $\{r_n\} = \{BABA...\}$ y $2 < A < 4$ y $2 < B < 4$

positivo, lo errores tienden a ser magnificados, aún así, después de estos puntos, aparecen órbitas superestables, hay algunas características de esta gráfica que son universales, es decir, que las vemos aún cuando estuviéramos analizando a funciones más complicadas en vez de este sencillo ejemplo.

También hemos notado que el parámetro de control (o parámetro de bifurcación) juega un papel crucial en la generación de caos.

Ahora, ¿qué sucedería si se hace al parámetro de control r dependiente del tiempo?. Es decir, en los modelos que se han visto hasta ahora, r permanecía constante en cada órbita, el parámetro en estos modelos se refiere a la descripción de las condiciones ambientales del sistema. Ahora, si este mismo parámetro es cambiado periódicamente, por lo que se escribe ahora r_n en vez de r, para mantener un poco la simplicidad y encontrar una fácil representación gráfica, se hace una hipótesis dicotómica [Markus y Hess, 1989b]: r_n puede tomar únicamente dos valores, A ó B.

Es interesante observar qué sucede con la dinámica del mapeo logístico y otros mapeos unidimensionales cuando el mapeo es visto como una función de A y B para diferentes secuencias $\{r_n\}$, por ejemplo, $\{BABABA...\}$, $\{BBABABBABA...\}$, $\{BBABABABBABABABA...\}$, $\{B^6A^6\,B^6A^6...\}$, u otra secuencia arbitraria.

Para poder observar el comportamiento de estas órbitas es necesario una representación gráfica. Sería complicado intentar hacer una representación gráfica de cada combinación posible de A y B como la de la figura 2.1. Mario Markus del Instituto Max Planck de Dortmund, Alemania, desarrolló una técnica gráfica para representar este tipo de sistemas dinámicos en un plano definido por parámetros de bifurcación, y no por las variables de fase [Markus, 1995], como comúnmente se hacía. En su representación, los puntos no se mueven en el plano mientras el proceso dinámico se encuentra evolucionando, en vez de ésto, el proceso dinámico es calculado en cada punto (A, B) en el plano y el exponente de Liapunov es calculado como un promedio sobre el proceso de iteración. Se ahondará más sobre esta representación en la siguiente sección.

2.1. Método

Para cada par de parámetros (A, B) se calcula el exponente de Liapunov con la ecuación 1.2. En el caso del mapeo logístico que es el sistema dinámico en estudio, el valor de la derivada que requiere la función 1.2 es $f'(x_n) = r_n - 2r_n x_n$, recuérdese que la ecuación del exponente de Liapunov es un límite:

$$\lambda = \lim_{n\to\infty} (\frac{1}{n}) \sum_{k=1}^{k=n} \ln \left(\left| f'(x_{k-1}) \right| \right)$$

por lo que para este particular ejemplo tenemos que:

$$\lambda = \lim_{n\to\infty} (\frac{1}{n}) \sum_{k=1}^{k=n} \ln \left(|r_k - 2r_k x_k| \right) \tag{2.1}$$

donde

$$x_n = r_{n-1} x_{n-1} (1 - x_{n-1})$$

Como la aproximación a la ecuación 2.1 es altamente irregular, no se puede encontrar un criterio de convergencia confiable [Markus y Hess, 1989b], por lo que haciendo $n = 4 \times 10^3$ sobre todo el plano genera resultados satisfactorios. Para permitir que los transitorios no interfieran en el cálculo del valor de λ, se hacen 600 iteraciones y se desechan antes de empezar a calcular el exponente con la ecuación 2.1.

Cada pixel en el área desplegada corresponde a un valor (A, B) y el color o la escala de grises usada en cada pixel indica los valores de λ para cada par

(A, B) en cuestión. Como ya sabemos, en general el caos es determinado por valores de $\lambda > 0$ y la periodicidad (orden) por valores de $\lambda < 0$. Una figura generada por este método es la figura 2.2, en ésta, una discontinuidad en la escala de grises es usada como referencia para marcar la transición entre orden y caos.

La ejecución más natural es una implementación secuencial, es decir, en la cual cada pixel del área en la cual estamos mapeando el intervalo de A y de B sea calculado uno por uno y en un orden determinado.

2.2. Implementación secuencial

Se ha descrito un poco en la subsección 2.1 la forma en que es generado un diagrama de Liapunov, ahora, estamos interesados en su implementación en un lenguaje de programación. El algoritmo secuencial para generar las imágenes se hace con la suposición de que éstas muestran una sección rectangular del plano (A, B). Muchas veces se supone que no necesariamente la ventana en estudio sea paralela a sus ejes [Markus, 1995]. Para los fines de este trabajo no se otorga esa libertad, ya que observar a cualquier área rotada se puede hacer con programas externos y tiene más fines estéticos que una razón matemática práctica.

Tenemos que la ventana está definida por tres cualesquiera de los cuatro puntos AI (arriba izquierda), DI (debajo izquierda), AD (arriba derecha), DD (debajo derecha). Para cada punto (A, B) del rectángulo se analiza el comportamiento de la iteración $x_{n+1} = f(x_n)$, tomando r los valores A y B alternativamente, según el esquema elegido. El esquema AB designa, como ya dijimos, la sucesión $\{ABABAB...\}$, y así sucesivamente con la sucesión elegida.

El algoritmo es el siguiente[Markus, 1995]:

1. Dos ciclos asignan valores sucesivos a A y B, cubriendo el rectángulo paramétrico (una cuadrícula) con mayor o menor densidad, según la resolución deseada.

2. Para cada uno de estos valores de A y de B, se itera la ecuación recursiva $x_{n+1} = f(x_n)$, siendo f una función como la descrita en la sección 1.1. Ésto significa que partiendo de un valor inicial x_0 dado, se calcula la órbita del sistema alternando también el valor de r con la secuencia definida en un principio. Se realizan 400 iteraciones para que desaparezca la influencia de los valores iniciales, después se determina un exponente de Liapunov aproximado λ, para el que vuelven

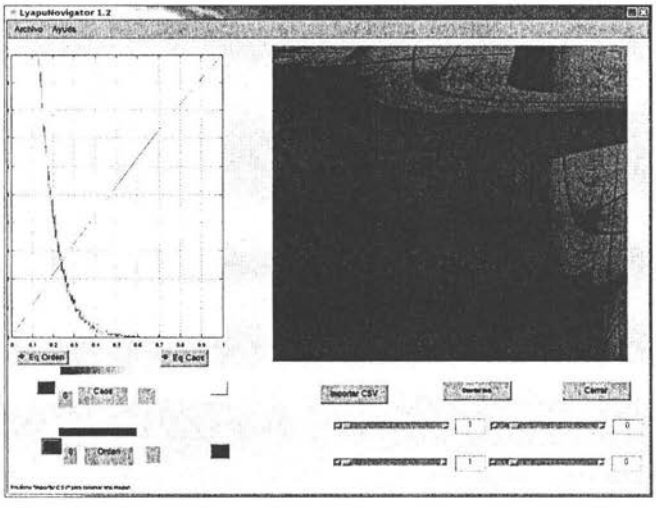

Figura 2.3: Una imagen del Lyapunovigator

a calcularse 4×10^3 iteraciones [Markus y Hess, 1989b] para tener la mejor precisión posible. Con estos valores x_n se determina la derivada de $df(x_n)/d(x_n)$ (con el valor vigente de r) y luego se calcula el valor medio de esta segunda serie de iteraciones dado por la la ecuación 2.1. Recordemos que el número de iteraciones que se hacen modifican dos cosas: la calidad de las imágenes generadas y el valor del λ más cercano a su convergencia. Si se hacen pocas iteraciones, las imágenes parecerán agujeradas o desgarradas, además, los valores que se irían obteniendo no serían valores cercanos al límite que se converge.

3. Se le asigna al punto del plano (A, B) un color correspondiente a su valor de λ. Es decir, se usa una gama de colores para los valores negativos y otra para los positivos, generalmente se recomienda, si se está trabajando en escala de grises, blanco moviéndose al negro cuando $\lambda < 0$ y negro moviéndose al gris con $\lambda > 0$.

El principal interés de este trabajo son los dos primeros pasos del algoritmo, ya que la asignación de colores se hace con una aplicación externa

que leerá un archivo CSV[1] llamado Lyapunovisualizer[2]. El archivo CSV únicamente tiene valores separados por comas, por lo que se crea la matriz de exponentes de Liapunov y es escrita en este formato. Luego se leen estos datos usando el Lyapunovisualizer. Cada retorno de carro en este archivo es el inicio de un nuevo renglón de la matriz, por lo que el Lyapunovisualizer (tomando en cuenta este formato) despliega y guarda en un mapa de pixeles la imagen generada por estos valores, la figura 2.3 nos muestra la interfaz gráfica de este programa. La forma en que estos valores son desplegados es similar a la descrita por el paso 3 del algoritmo descrito en la sección anterior.

En este trabajo se usa el lenguaje de programación C, pero el método es fácilmente trasladable a Fortran 77 o Fortran 90, con excepción de algunos casos importantes en el problema secuencial y se usan las mismas funciones (con su respectiva sintaxis) en el caso de la implementación en MPI.

En el caso que concierne a este trabajo, tenemos que los arreglos bidimensionales en C son guardados en locaciones de memoria ordenadas por renglones[3], y de arriba hacia abajo, es decir, de acuerdo con la figura 2.4 los arreglos se recorren de izquierda a derecha y de arriba hacia abajo, la cuadrícula es para mostrar la forma en que se mapea una ventana de valores continuos A y B en una ventana finita y discreta, donde dependiendo del tamaño de la matriz será el número de valores correspondiente a cada celda a calcular.

Por ejemplo, continuando con la figura 2.4, se puede ver que únicamente se mapean los valores, se avanza de acuerdo al número de pasos que está definido por el tamaño de la matriz, o lo que es lo mismo, la resolución de la imagen a generar, y se calcula en cada punto dados los valores (A, B) en el que se esté en ese momento. Es notable entonces que simplemente se deben de tener dos puntos para caracterizar a la ventana que vamos a estudiar: AI (arriba izquierda) y DD (debajo derecha), estos puntos son pasados como parámetros al programa.

Para tener mayor flexibilidad en las áreas a explorar, el programa debería aceptar parámetros en línea como los siguientes:

s La *semilla.* Es el punto inicial x_0 que inicia la iteración.

w *El ancho de la imagen.* Comúnmente este valor es 2000, ya que éste permite tener una resolución aceptable.

[1] Comma Separated Values, es decir, de Valores Separados por Comas.

[2] Desarrollado por Gabriel Cárdenas en el ambiente Matlab.

[3] Contrario a Fortran, en el cual los arreglos bidimensionales son guardados en locaciones de memoria que hacen referencia a las columnas [Gropp et al., 1999].

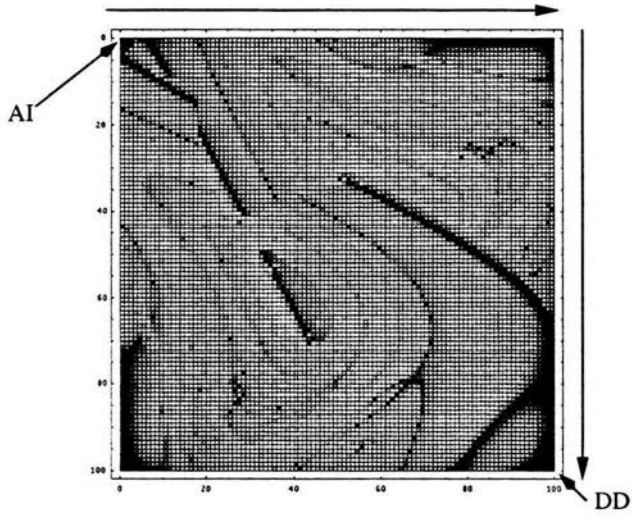

Figura 2.4: Una matriz de exponentes de Liapunov

h *La altura.* Al igual que el ancho usamos 2000 ya que generalmente los punto AI, DD forman una ventana cuadrada.

S *La secuencia a usar.* Se usaría A y B, por ejemplo, AABB querría decir que usaríamos justo esa secuencia.

x *El valor de x inicial.* La componente x del punto AI.

y *El valor de y inicial.* La componente y del punto AI.

X *El valor final de x.* La componente x del punto DD.

Y *El valor final de y.* La componente y del punto DD.

t *El transitorio.* El cual se usa para eliminar los efectos de las primeras iteraciones, se sugiere usar 600 iteraciones[Markus y Hess, 1989b].

i *Número de iteraciones.* Que se hacen después del transiente. Este valor, al igual que en el anterior parámetro según la bibliografía, deberá ser mayor o igual a 4×10^3.

El cálculo del exponente de Liapunov se hace por medio del código de la figura 2.5. Como se puede ver, es justo esta función la que calcula el exponente dados un par de puntos A y B y una secuencia de AB definida en toda el área a explorar.

El primer ciclo lee la secuencia y la transforma en un arreglo entero en el que sólo pueden haber dos valores, 0 y 1 donde se le asigna el valor 1 si estamos frente a un valor de A y 0 si estamos frente a un valor B, es decir, si pasáramos a la función `lyapunov_exp` el parámetro *aabbabaaabbb* en el primer ciclo esta secuencia se convertiría en un arreglo $\{1,1,0,0,1,0,1,1,1,0,0,0\}$. Y este arreglo toma el valor de A o B dependiendo del paso en el que vayamos al comparar el arreglo `forcing[bindex]`, `bindex` se mueve de tal manera que al llegar al tamaño del arreglo, éste regresa a 0 de nuevo para empezar un nuevo ciclo de la secuencia AB elegida.

En los dos siguientes ciclos donde se calcula el exponente de Liapunov, el primero, hace la iteración del transiente y desecha esos valores, mientras al mismo tiempo utiliza el valor de A o B que le corresponde siguiendo el esquema explicado en el párrafo anterior. El segundo ciclo funciona de la misma manera pero ahora calculando tanto el valor de la función como su derivada, usando de igual forma el valor de A o B según sea el caso, mientras va agregando los valores en una variable llamada `sum`.

Después de estos dos ciclos, únicamente dividimos `sum` entre el valor del total de iteraciones y regresamos este valor.

```
double lyapunov_exp (double x, double y, char *ff)
{
 int maxindex = MAXINDEX;
 int forcing [MAXINDEX];
 int bindex = 0;
 int i;
 double xn = X0;
 double xn_alt, sum, r;
 maxindex = strlen(ff);
 while (bindex < maxindex) {          /* First loop */
   if (*ff == 'a') {
     forcing[bindex] = 1;
     bindex++;
   }
   else if (*ff == 'b') {
     forcing[bindex] = 0;
     bindex++;
   }
   ff++;
  }
  bindex = 0;
  for (i = 0; i < transient; i ++) { /* Second loop */
   r = (forcing [bindex]) ? x : y;
   xn_alt = fmod ((xn + r), DOS_PI);
   xn = B * sin (xn_alt) * sin (xn_alt);
   bindex++;
   if (bindex >= maxindex)
     bindex = 0;
  }
  sum = 0.0;
/* Third loop */
  for (i = transient; i < (transient + total_iterations); i++){
    r = (forcing [bindex]) ? x : y;
    xn_alt = fmod ((xn + r), DOS_PI);
    xn = B * sin (xn_alt) * sin (xn_alt);
    sum = sum + log (fabs (2.0 * B * sin (xn_alt) * cos \
         (xn_alt)));
    bindex++;
    if (bindex >= maxindex)
      bindex = 0;
}
sum = sum / (double)total_iterations;
return sum;
}
```

Figura 2.5: La función que calcula el exponente de Liapunov

La función utilizada en el código de la figura 2.5 utiliza una función del tipo II, de acuerdo con la clasificación dada por Mario Markus [Markus, 1995]:

$$f(x_{n+1}) = b\,\mathrm{sen}^2(x_n + r) \tag{2.2}$$

esta función seno recibe como parámetro un valor de punto flotante pero que es antes reducido mediante la función módulo para que el valor que entre únicamente esté en el intervalo $(0, 2\pi)$. Ésto es únicamente relativo al lenguaje de programación C [Oualline, 1997]. Muchas funciones trigonométricas son calculadas usando una serie de funciones, ya que la función es exacta hasta para 4 dígitos, si introducimos dígitos mayores, se generan pequeños errores, pero recordemos que estamos frente a funciones que tienden a aumentar los errores conforme las iteraciones, por lo que el valor de la función seno debe de ser lo más exacto posible. En Fortran por ejemplo, el valor del parámetro pasado a la función se pasa a través del módulo y así el valor de la función es calculada con el valor del parámetro que nosotros calculamos a mano. Hay bibliotecas de funciones que hacen ésto automáticamente, pero eso significaría que el programa no sería 100 % portable a otras plataformas UNIX, la biblioteca estándar de C define funciones matemáticas en ésta y que son distribuidas con sus compiladores.

El ciclo principal, el cual es mostrado en la figura 2.6 del programa serial hace únicamente tres cosas, primero recibe los parámetros a través de la línea de comandos por medio de la función *parsea_parametros(argc, argv)* y define el paso que irá haciendo basado en estos parámetros, inicializa el primer punto a calcular, ésto es, inicia el ambiente y las variables para iniciar el cálculo, en segundo lugar se va moviendo por la malla que es creada y va calculando para cada punto el valor del exponente de Liapunov al mismo tiempo que va imprimiendo este valor en el formato CSV, y cada vez que pasa un renglón imprime un retorno de carro para que la salida sea coherente con la entrada que espera recibir Matlab.

Como se observa, el programa envía toda su salida a la salida estándar, porque con ésto se puede hacer que el programa redireccione la salida, ya sea por medio de un redireccionamiento[4] o con una *tubería* para enviar esa salida a otro programa que entienda este formato.

Este programa en realidad es sencillo, ya que podría correr en cualquier tipo de computadora que tuviera un compilador simple de C y que entienda el redireccionamiento o tuberías[5].

[4]Por ejemplo, podemos correr el programa escribiendo a un archivo en el disco duro: ./serial -s2.5 -w500 -h500 -b2.5 -Sab -x0 -y4 -X4 -Y0 -t400 -i4000 >archivo.csv.

[5]Esto nos mueve automáticamente a ambientes tipo UNIX.

```
int main (int argc, char *argv [])
{
 double delta_x, delta_y, a, b, point;
 int k, j;
 parsea_parametros (argc, argv);
 delta_x = fabs (x_final - x_init) / WIDTH;
 delta_y = fabs (y_init - y_final) / HEIGHT;
 a = x_init;
 b = y_init;
 for (k = 0; k < HEIGHT ; k++) {
   for (j = 0; j < WIDTH; j++) {
     point = lyapunov_exp (a, b, sequence);
     printf ("%f, ", point);
     a = a + delta_x;
   }
   printf ("\n");
   a = x_init;
   b = b - delta_y;
 }
}
```

Figura 2.6: Ciclo principal

Procesador	G3 a 350 Mhz	PIII a 600 Mhz	Athlon a 2.1 Ghz
Tiempo	13715.208 seg.	5943.223 seg.	1898.015 seg.

Cuadro 2.1: Diferentes tiempo para diferentes arquitecturas para generar una imagen de 1000 × 1000 pixeles

2.3. El tiempo como limitante

La implementación secuencial del algoritmo de Markus, ha sido sencillo, pero tiene un par de limitantes, la primera y más importante es el tiempo que tomaría explorar una región del plano definido por AB, al correr este programa con un tamaño de imagen de 1000 × 1000 pixeles en distintas arquitecturas obtenemos los resultados desplegados en la tabla 2.1.

Ésto muestra la cantidad de tiempo que se puede gastar con resoluciones relativamente pequeñas, al menos se requieren imágenes con una resolución de 2000 × 2000 pixeles para tener imágenes con una calidad razonable, o quizá en una aplicación en tiempo real que requiriera el uso de estas figuras podría ser de importancia que el tiempo en el que son generadas fuera lo suficientemente corto para ser analizadas.

La parte en donde se tiene que calcular el logaritmo, éste es calculado por medio de una aproximación, aunque la función logarítmica ha sido tabulada en un arreglo para reducir el tiempo de cómputo [Markus et al., 1998], en este trabajo no se lleva a cabo este método y se calcula el valor de los logaritmos para cada punto (A, B), por dos razones:

1. Se quiere tener valores lo más exactos posible, ya que para alguna aplicación podría ser necesario tener valores λ de este tipo para diferenciar ciertos puntos de otros.

2. Las figuras generadas son más suaves con valores más exactos, estamos calculando valores de funciones que generan caos, por lo que son sensibles a errores, cualquier cambio en el valor de las funciones puede llevar a valores insospechados y erróneos.

Es por eso que en este trabajo se trata de encontrar una solución a este problema generando estos diagramas en paralelo, la forma en que son calculados por medio del algoritmo descrito en cada punto da pie a que se pueda usar el paralelismo de manera casi natural, por lo que es interesante observar la manera en que estos diagramas podrían ser generados así. Los siguientes

capítulos profundizan, entonces, en el tipo de hardware y software a usarse para llegar a este objetivo.

Capítulo 3

Clusters Beowulf

3.1. La necesidad del cómputo paralelo

Desde el inicio de las computadoras, el objetivo de poder realizar la mayor cantidad posible de operaciones por segundo ha sido tanto una obsesión como una necesidad para los usuarios de éstas, ya que por un lado al ser más rápidas, pueden resolver ciertos problemas y descubrir patrones que hubieran sido imposibles intentar con lápiz y papel, como el descubrimiento del atractor de Lorenz [Lorenz, 1996]. Por otro lado, los científicos siempre pueden generar nuevos conceptos, modelos y preguntas más complejas que se adaptarían fácilmente a las nuevas capacidades del poder de cómputo.

A través de varios siglos, la ciencia ha seguido el paradigma básico de observar, luego teorizar, y finalmente probar la teoría a través de la experimentación. Similarmente los ingenieros primero diseñaban en papel y luego construían y probaban prototipos para finalmente construir un producto. En nuestros días es mucho más barato llevar a cabo experimentos en la computadora que construir una serie de prototipos. La experimentación y observación en el paradigma científico y diseño, el prototipo en el paradigma ingenieril están siendo reemplazados cada vez con más frecuencia por la simulación por computadora. Incluso, ahora es posible estudiar y simular fenómenos que no podrían ser analizados con suficiente certidumbre en la vida real, por ejemplo el clima y la evolución del universo.

Aunque se podría pensar que la velocidad actual de las computadoras es suficiente el siguiente ejemplo ligeramente modificado [Pacheco, 1997] nos muestra que un problema simple podría necesitar una gran cantidad de operaciones por unidad de tiempo:

"Supóngase que se quiere predecir el clima sobre la república para los

próximos dos días. También supóngase que se quiere modelar la atmósfera desde el nivel del mar hasta una altitud de 20 kilómetros y que se necesita hacer una predicción del clima a cada hora de los siguientes dos días."

Una aproximación válida y común para este tipo de problema es cubrir toda la región de interés con una malla y luego predecir el clima para cada vértice de ésta. Así que supóngase entonces que se usa una malla cúbica, con cada cubo midiendo 0.1 kilómetros en cada lado. Como el área del país es aproximadamente 2,000,000 millones de kilómetros cuadrados, se necesitarían al menos:

$$2.0 \times 10^6 km^2 \times 20km \times 10^3 cubos = 4 \times 10^{10} puntos$$

Si toma 100 operaciones para determinar el clima en un punto cualquiera de la malla, entonces para predecir el clima de una hora desde este momento, se necesitarían hacer al rededor de 4×10^{12} operaciones. Como se quiere predecir el clima de cada hora durante las próximas 48 horas, se necesita entonces un total de:

$$4 \times 10^{12} operaciones \times 48 horas \approx 2 \times 10^{14} operaciones$$

Si se cuenta con una computadora que puede ejecutar un billón (10^9) de operaciones por segundo, este cálculo tomará aproximadamente:

$$2 \times 10^{14} \div 10^9 operaciones/sec = 2 \times 10^5 segundos \approx 48\ horas!$$

En otras palabras, el cálculo es completamente inútil para cuando haya terminado si solamente se pueden llevar a cabo un billón de operaciones por segundo. Si se pudieran llevar a cabo un trillón de operaciones por segundo, tomaría tres minutos aproximadamente llevar a cabo el cálculo, por lo que se podría hacer una predicción aproximada para las próximas 48 horas.

No sería complicado hacer este problema mucho más grande de tal manera que se llegara a necesitar una computadora que tuviera la capacidad de llevar a cabo más de un trillón de operaciones por segundo. Incluso, es fácil encontrar problemas que necesiten llevar a cabo el mismo número de operaciones por segundo para poder llevar a cabo simulaciones o resolver problemas, por ejemplo simulaciones a nivel atómico de biomoléculas, o en el interés particular de este trabajo, calcular los exponentes de Liapunov de cierta área.

Además de la necesidad de velocidad computacional, también es necesario en problemas de gran escala el almacenamiento de grandes cantidades

de información. Aún cuando se pudiera tener la velocidad requerida, si sólo se tiene acceso a algunos gigabytes de memoria RAM. Una computadora así sería de poca utilidad, por lo tanto, el desarrollo de más velocidad requiere también el desarrollo de mejores medios de almacenamiento.

Una forma simple de tener más poder de cómputo es esperar que las tecnologías existentes sigan creciendo de acuerdo a la ley de Moore [Moore, 1965]. Sin embargo, cada vez es más claro que si no se encuentran nuevas y mejores formas de introducir más transistores en un espacio cada vez más reducido la ley de Moore no aplicará por limitaciones físicas, ésto, en términos de tamaños implicaría que tuviéramos que encontrar la manera de escribir una palabra de memoria del tamaño de un átomo [Pacheco, 1997] lo cual hasta el momento no es posible.

La solución más natural se puede dislumbrar en el siguiente problema análogo: "Pedro es un excavador, y puede excavar 1000 metros cúbicos diariamente, de pronto tiene un trabajo urgente y necesita excavar 10,000 metros cúbicos en un sólo día, por lo que simple y sencillamente solicita la ayuda de 9 personas más que tengan la misma capacidad de excavación que él, incrementando con ellos su fuerza de trabajo de 1 a 10."

La analogía es clara, Pedro es el procesador y memoria y los 10,000 metros cúbicos es la capacidad que se necesita por un día, por lo que lo único que se necesita es usar más computadoras y ponerlas a trabajar en el problema al que estamos enfrentándonos, ésto es lo que se llama una *computadora paralela*, una colección de múltiples procesadores que pueden trabajar juntos en la solución de un problema.

3.2. Arquitecturas del cómputo en paralelo

En el cómputo paralelo se han definido varias taxonomías dependiendo en el tipo de característica que sea de interés. Por ejemplo, la clasificación más ampliamente utilizada [Sterling et al., 1998] está basada en el grado de acoplamiento (figura 3.1).

En esta taxonomía se define el grado de acoplamiento como una combinación de dos factores básicos en el cómputo en paralelo, el ancho de banda disponible y la latencia.

En el nivel más alto de esta taxonomía podemos encontrar a las metacomputadoras (grids) como las menos acopladas, ya que éstas al estar interconectadas por medio de redes (que pueden tener incluso diferentes dominios de administración) las hace tener una latencia alta. En el extremo opuesto tenemos a las computadoras vectoriales las cuales al *entubar* mu-

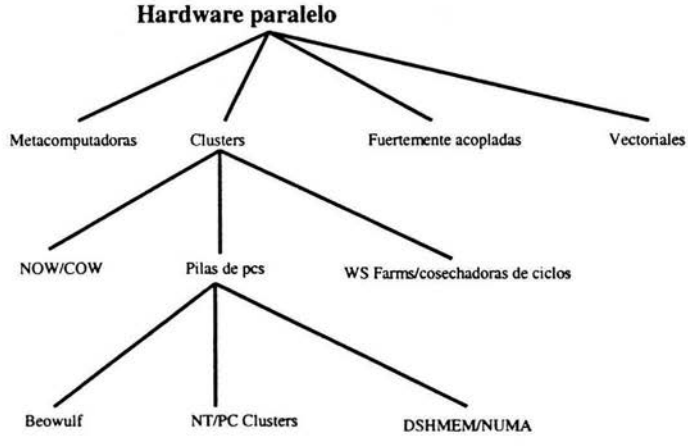

Figura 3.1: Taxonomía por grado de acoplamiento

chas de sus operaciones llegan a tener niveles muy bajos de latencia. Entre estas dos extremos tenemos a los clusters, los cuales son el tipo de sistemas que interesa en este trabajo, sistemas con un nivel de mediano a fuertemente acoplados que pueden llegar a tener componentes independientes que, a diferencia de las metacomputadoras tienen relativa *localidad* bajo el mismo dominio, éstos están conectados por medio de una red para tener un medio de interacción, muchas veces se puede hacer un cluster por medio de PCs conectadas directamente a una red y usarlas cuando éstas no están siendo usadas, por ejemplo después de horas de oficina, a este tipo de clusters se les conoce como *granjas de estaciones de trabajo* o *cosechadoras de ciclos*, este tipo de cúmulos[1] pueden ser de mucha utilidad bajo ciertas circunstancias. Dentro de la familia de los *clusters* se tiene a los del tipo Beowulf, la cual es la arquitectura usada en este trabajo, y también hay otro tipo de clusters en este mismo nivel, los clusters creados por medio de sistemas cerrados como son los clusters de computadoras Windows NT y también otro tipo que tiene hardware muy particular o que usan memoria compartida por medio de hardware propietario del tipo NUMA.

[1] O clusters, se usa el término de manera indistinta.

3.2.1. Taxonomía de Flynn

Además de la taxonomía basada en el grado de acoplamiento, la clasificación original de las computadoras en paralelo fue propuesta por Michael Flynn, quien clasificó a este tipo de sistemas de acuerdo al número de flujos de instrucciones y de flujos de datos. La máquina clásica de Von Neumann tiene por ejemplo un sólo flujo de instrucciones y un flujo de datos, por lo que se le conoce como una máquina SISD[2], en el extremo opuesto, se encuentran las máquinas MIMD[3] en las cuales una colección de varios procesadores autónomos operan en sus flujos de datos propios. Ésta es la arquitectura más general, pero también existen pasos intermedios: máquinas SIMD[4] y MISD[5], cada tipo será explicado en las siguientes secciones con mayor profundidad.

Máquina de Von Neumann

La computadora clásica de Von Neumann está dividida en un CPU[6] y memoria principal. El CPU a su vez está dividido en una unidad de control y una unidad lógica-aritmética[7]. La memoria guarda tanto instrucciones como datos, la unidad de control dirige la ejecución de los programas y la unidad lógica-aritmética lleva a cabo los cálculos que se llaman en el programa. Cuando están siendo usado por el programa, las instrucciones y datos son guardados en los *registros*. Por supuesto, la memoria de este tipo es muy cara por lo que hay pocos registros.

Los datos e instrucciones se mueven entre la memoria y los registros en el CPU: por donde viajan se le conoce como *bus* el cual es una colección de cables paralelos que tienen un controlador de acceso al bus, entre más rápido sea un bus más cables tendrá.

Una máquina clásica de Von Neumann necesita también dispositivos adicionales antes de que sea útil, dispositivos de entrada y salida (como mouse, teclado, monitor, etcétera) y usualmente un dispositivo de almacenaje de información como lo es un disco duro.

Hay un inconveniente en este tipo de computadoras, y éste es la transferencia de datos e instrucciones entre la memoria y el CPU. No importa que tan rápidos sean los procesadores, la velocidad de ejecución de los programas está limitada por la tasa de transferencia de las secuencias de instrucciones y

[2]Single Instruction Single Data, Instrucción Única Datos Únicos.
[3]Multiple Instruction Multiple Data, Instrucciones Múltiples Datos Múltiples.
[4]Single Instruction Multiple Data, Instrucción Única Múltiples Datos.
[5]Multiple Instruction Single Data, Instrucciones Múltiples Datos Únicos.
[6]Central Processing Unit, Unidad Central de Procesamiento.
[7]Mejor conocida como ALU.

datos entre la memoria y el CPU. Por lo mismo, son pocas las computadoras de nuestros días que son estrictamente máquinas clásicas de Von Neumann, actualmente tienen una memoria intermedia entre la memoria principal y los registros llamada *caché*, la idea detrás del caché es que los programas tienden a acceder datos e instrucciones secuencialmente y contiguos. Por lo tanto, si guardamos un bloque pequeño de datos y un bloque pequeño de instrucciones en memoria que es tan rápida como los registros, la mayoría de los accesos a memoria de los programas accederían esta memoria rápida en vez de la memoria principal lenta.

Pipeline y computadoras vectoriales

La primera extensión al modelo de Von Neumann que se utilizó ampliamente fue el *pipelining* o *entubamiento*. Si los circuitos que forman el CPU se dividen en unidades funcionales, y a estas unidades se les pone en una *tubería*, entonces la tubería podría producir un resultado en cada ciclo de ejecución después de la inicialización. Por ejemplo:

```
float x[100], y[100], z[100];
for (i = 0; i < 100; i++)
    z[i] = x[i] + y[i];
```

Y supongamos que una suma simple consiste de la siguiente secuencia de operaciones:

1. Extraer los operandos de la memoria.
2. Comparar los exponentes.
3. Recorrer uno de los operandos.
4. Sumar.
5. Normalizar el resultado.
6. Guardar el resultado en la memoria.

Ahora, supóngase que se tienen unidades funcionales (de las que hablábamos en el párrafo anterior) que llevan a cabo estas operaciones, y estas unidades están alineadas en una tubería. Ésto es, la salida de una unidad funcional es la entrada de la siguiente. Entonces, mientras, por ejemplo, $x[0]$ y $y[0]$ están siendo sumados, alguno de $x[1]$ y $y[1]$ pueden ser recorridos, los exponentes

en $x[2]$ y $y[2]$ pueden ser comparados y $x[3]$ y $y[3]$ pueden ser extraídos de la memoria. Por lo que una vez que la *tubería* está *llena*, se pueden producir resultados 6 veces más rápidos que lo que lo se haría sin el entubamiento.

Una mejora más que puede implementarse en hardware es la que se obtiene al añadir instrucciones vectoriales al conjunto de instrucciones básicas de máquina. En el ejemplo, de la suma de 100 pares de flotantes, si no tenemos instrucciones vectoriales, una instrucción que pertenece al conjunto básico de instrucciones tiene que ser extraída y decodificada 100 veces. Con instrucciones vectoriales, cada una de estas instrucciones básicas necesita ser usada una vez. La diferencia es de cierta manera análoga a la diferencia entre código en Fortran 77:

```
      do 100 1 = 1, 100
         z(i) = x(i) + y(i)
100 continue
```

y su equivalente en Fortran 90:

```
z(1:100) = x(1:100) + y(1:100)
```

Otra de las ventajas de la arquitectura vectorial es que en este tipo de máquinas se tienen bancos de memoria múltiples. Las operaciones que acceden a la memoria principal, como extraer o guardar datps, son varias veces más lentas que las operaciones que sólo usan el CPU como el sumar datos. El uso de bancos de memoria independiente pueden, hasta cierto grado, mejorar este problema.

Muchas veces se les refiere a este tipo de arquitectura como máquinas MISD, aunque hay un número considerable de investigadores que establecen que no existe tal cosa como una máquina MISD, y que éstas no son más que variantes de las máquinas SIMD. Algunos más afirman que estas máquinas no son paralelas estrictamente hablando.

Una gran virtud de los procesadores vectoriales es que están bien estudiados y que hay compiladores optimizados para estos procesadores[8] por lo que es relativamente fácil escribir programas que obtienen muy buen desempeño usando este tipo de procesadores. Como una consecuencia, aún son populares entre la comunidad científica de cómputo de alto desempeño.

[8]La supercomputadora más rápida del mundo usa este tipo de procesadores, el Earth simulator, instalada en Japón, una descripción amplia del hardware y software usado por esta supercomputadora se encuentra en http://www.top500.org.

Este tipo de procesador también tiene inconvenientes, los principios del entubamiento y vectorización no trabajan bien en programas que usan muchas ramas[9] y usan estructuras irregulares, ésto se debe a que la clave para el buen desempeño de estos procesadores es llenar la tubería y mantenerla lleno, si los operandos no están ordenados correctamente en la memoria, es imposible mantenerla llena. Por ejemplo, si un programa tiene muchas ramas condicionales entonces habrá pocas posibilidades de usar instrucciones vectoriales. Aún peor, el inconveniente más grande es que no parecen escalar bien. Esto quiere decir que aún no está claro cómo modificarlos de tal manera que puedan manejar problemas más grandes. Aún cuando se añadieran más tuberías y se pudiera mantenerlas llenas, el límite superior de su velocidad será un pequeño múltiplo de la velocidad del CPU.

SIMD y MISD

Una máquina tipo SIMD, contrastándola con un procesador vectorial, tiene un sólo CPU dedicado únicamente al control y un conjunto grande de ALUs subordinados, cada uno con su cantidad de memoria determinada. Durante cada ciclo de instrucciones, el procesador de control envía una instrucción a todos los procesadores subordinados, y cada uno de éstos ejecutan la instrucción o se quedan parados.

El siguiente ejemplo servirá para aclarar lo anterior. Supóngase que se tiene de nuevo tres arreglos x, y y z, distribuidos de tal manera que cada procesador contiene un elemento de cada arreglo. Ahora suponiendo que se quiere ejecutar la siguiente secuencia de instrucciones secuenciales:

```
for (i = 0; i < 1000; i++)
    if (y[i] != 0.0)
        z[i] = x[i]/y[i];
    else
        z[i] = x[i];
```

Entonces, cada procesador subordinado ejecutaría algo parecido a la siguiente secuencia de operaciones:

```
Paso 1: Probar local_y != 0.0
Paso 2:
a. Si local_y fue distinto de cero, z[i] = x[i]/y[i].
b. Si local_y fue cero, no hace nada.
```

[9] Branches.

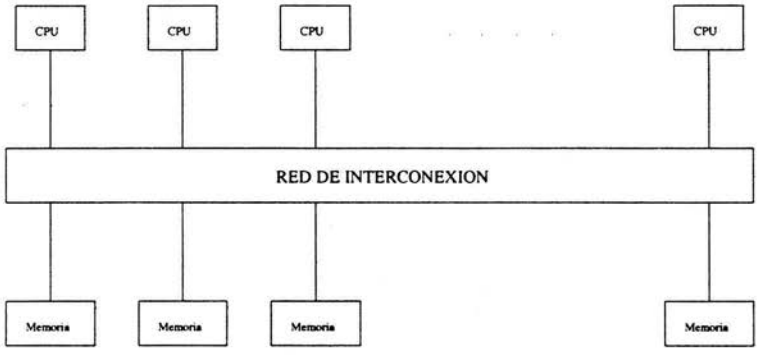

Figura 3.2: Sistema de memoria compartida genérica

```
Paso 3:
a. Si local_y fue no cero, no hacer nada.
b. Si local_y fue cero, z[i] = x[i].
```

En esta secuencia de instrucciones se observa que, los pasos en cada procesador estén completamente sincronizados. Es decir, en un momento del tiempo, cada procesador está haciendo o no la operación, es decir, da una clara muestra del problema más grande de este tipo de arquitectura, la cual es que en caso de que haya muchas sentencias condicionales podría haber procesadores que estarían sin trabajar por periodos largos de tiempo.

Este ejemplo, no muestra la relativa facilidad con que este tipo de arquitecturas pueden ser programadas si el problema tiene una estructura regular, aunque la comunicación es costosa en sistemas MIMD, no es más costosa que los cálculos en una máquina SIMD [Pacheco, 1997].

MIMD

La diferencia entre máquinas MIMD y SIMD es que en las primeras los procesadores son completamente autónomos. Cada procesador tiene una unidad de control y una unidad lógico-aritmética, por lo que cada programa es capaz de ejecutar sus programas, es decir, a diferencia de las máquinas SIMD, éstos son asíncronos, sucede a menudo que los procesadores puedan estar ejecutando el mismo programa sin tener correspondencia entre ellos.

Esta arquitectura se divide en dos tipos: de memoria compartida y de memoria distribuida.

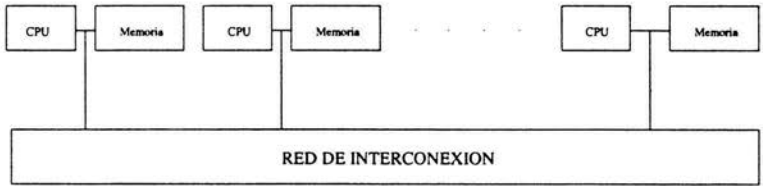

Figura 3.3: Sistema genérico de memoria distribuida

Memoria compartida: Estos sistemas, consisten en una colección de procesadores y módulos de memoria interconectados por una red fuera del bus de memoria, usualmente hay de dos tipos: basadas en bus y basadas en switch (NUMA). La diferencia entre ambas es que en la que está basada en el bus puede llegar a saturarse en caso de que varios de los procesadores intenten acceder a la memoria en un momento determinado, limitando el número de procesadores, mientras que en el caso de la basada en switch el acceso a memoria es manipulado por un interruptor electrónico que funciona de manera parecida a un semáforo, esto permite el escalamiento, la limitante de este tipo de memoria compartida es que es muy costosa, generalmente se encuentran en el mercado máquinas de este tipo que incluyen de dos a cuatro procesadores.

La memoria compartida tiene además un problema más general, el cual es la coherencia del caché. Si un procesador está accediendo en un momento determinado una variable compartida entre los demás procesadores y ésta se encuentra en el caché en ese momento justo, ¿cómo saber que el valor de tal variable es el actual?, la respuesta es que hay algunos protocolos que se encargan de llevar la coherencia en el caché, entre ellos el más conocido es el llamado protocolo *snoopy*, este protocolo muestrea el bus y cuando se hace un cambio a una variable compartida, el protocolo actualiza las demás.

Memoria distribuida: En este tipo de sistemas cada procesador tiene su propia memoria privada. Por lo que un sistema de este tipo se representa en la figura 3.3.

Como se observa en la figura 3.4, se puede pensar en la red de interconexión como una gráfica en la cual cada vértice (nodo) representa a un CPU y memoria y las aristas son las conexiones. Desde el punto de vista de programación y de desempeño la mejor configuración de tal red sería una en la cual cada nodo esté interconectado con cada nodo, es decir una red completamente conectada, con una red de este tipo cada nodo puede comunicarse

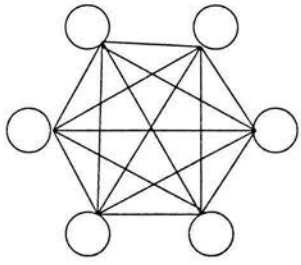

Figura 3.4: Una red completamente interconectada

directamente con cada nodo, incluso la comunicación no tendría retraso con lo cual, idealmente una red de este tipo sería perfecta, en el mundo real ésto es imposible más que para unos cuantos nodos, ya que el costo de construir una red de este tipo la haría completamente impráctica.

Sin embargo, en la práctica se usan redes simples y posiblemente algunas modificaciones al hardware que pueden llegar a mejorar el desempeño en una red. La más sencilla es el uso de una red de bus, ésto es, una red de computadoras conectadas a través de switch ethernet, el cual es el ejemplo más usado para ejemplificar una red de este tipo. El problema más general que tienen es que este tipo de redes tienden a ser generalmente lentas comparados con accesos a memoria compartida, y aún peor, tienden a saturarse rapidamente si muchos nodos intentan comunicarse al mismo tiempo.

3.2.2. Software en arquitecturas paralelas

Por un lado ya se ha visto que la memoria y el procesador son unidades fundamentales del hardware paralelo. Se tiene también que un concepto básico en cualquier computadora y en particular en las paralelas es el de *proceso*. Un proceso es un programa o una instancia de éste (en este caso un subprograma), que está ejecutándose independientemente en un procesador en un momento dado. Un programa es paralelo si tiene varios procesos que, durante el tiempo de ejecución, estén trabajando en una parte del problema dado. Para que un programa paralelo tenga utilidad es necesario que existan mecanismos que puedan especificar, crear y destruir procesos, así como formas de que estos procesos tengan interacción o comunicación entre ellos y se coordinen.

En las siguientes subsecciones se verán algunos paradigmas de programa-

ción que se encargan de trabajar en estas tareas. Por supuesto, éstos están, pueden o podrían ser implementados en todas o casi todas las arquitecturas de hardware que vimos en las secciones anteriores. Por ejemplo, es perfectamente razonable emular un acceso a memoria compartida usando memoria distribuida o viceversa.

Programación en memoria compartida

Los sistemas de memoria compartida tienen mecanismos que permiten la creación estática y dinámica de procesos, es decir, los procesos pueden ser creados al principio de la ejecución del programa por medio de una directiva del sistema operativo, o pueden ser creados dinámicamente. La función más conocida de creación de procesos dinámicos es *fork*, una implementación típica de fork permitiría iniciar un proceso *hijo* (*child*) al hacer una llamada a esta función. El proceso *padre* (*parent*), es decir el proceso que inició al proceso hijo, puede esperar a que el proceso hijo termine usando la función *join*.

La coordinación entre procesos en este paradigma es típicamente manejado por tres primitivas. La primera especifica qué variables pueden ser accedidas por todos los procesos. La segunda previene que los procesos accedan impropiamente a recursos compartidos. La tercera provee de medios para sincronizar los procesos.

Para asegurar que solamente un proceso ejecute cierta secuencia de instrucciones en cierto momento, se está tratando de hacer una *exclusión mutua*, y a la secuencia de instrucciones se le conoce como una *sección crítica*. Para limitar este acceso se usan *semáforos binarios* [Dijkstra, 1968]. También es necesaria una manera de sincronizar los procesos, ésto se hace con un mecanismo llamado *barrera*.

Programación con envío de mensajes

El envío de mensajes es el método más común para programar sistemas de memoria distribuida MIMD, la idea básica detrás del envío de mensajes se basa en la coordinación de los procesos. La coordinación de los procesos se hace mediante el uso explícito de funciones que crean mensajes que serán enviados o recibidos. Cada proceso tiene un identificador único llamado rango (*rank*), esta flexibilidad es la que nos provee el paralelismo al tener ramas condicionales del tipo:

```
if (process_rank == 0)
    send_parts ();
```

```
else
    recieve_parts ();
```

Donde `send_parts` y `recieve_parts` son funciones que son ejecutadas dependiendo de qué proceso esté corriendo, es decir, cada procesador está corriendo una copia idéntica del programa, pero el hacer cosas distintas se logra al incluir este tipo de condiciones. A esta aproximación para programar sistemas MIMD se le conoce como SPMD[10].

El estándar más utilizado del modelo de programación con envió de mensajes es MPI[11], aunque también existen otros que siguen este paradigma como ejemplo tenemos a PVM[12]. Hablaremos más a fondo sobre MPI en el siguiente capítulo.

Lenguajes Paralelos

Una forma de programar sistemas paralelos es el llamado paralelismo de datos. En este modelo, una estructura de datos es distribuida entre los procesos y cada procesador individual ejecuta las mismas instrucciones en sus respectivas partes de datos que le corresponden. Obviamente este tipo de programación es adecuado para sistemas tipo SIMD, aunque también es muy común en sistemas tipo MIMD. Uno de sus aspectos más atractivos es que para estructuras regulares, el usuario únicamente tiene que indicar que la estructura que se le va a pasar debe ser distribuida entre los procesos, el compilador automáticamente reemplazará la instrucción en el código fuente por instrucciones que distribuirán los datos y llevarán a cabo las operaciones en paralelo.

Existe una implementación de Fortran de este modelo, llamado High Performance Fortran (HPF). Una de sus mayores desventajas es que el mapear una estructura de datos a varios procesadores no es una tarea simple a menos que la estructura en cuestión sea estática y regular (por ejemplo una matriz definida). Esto podría llegar a causar dificultades en cualquier programa en paralelo.

RPC, Mensajes Activos

RPC[13] y los mensajes activos, generalizan el envío de mensajes, los procesos pueden comunicarse tanto intercambiando datos como solicitando la

[10] Single Program Multiple Data, Programa Único Datos Múltiples.
[11] Message Passing Interface o Interfaz de Envío de Mensajes.
[12] Parallel Virtual Machine o Máquina Virtual Paralela.
[13] Remote Procedure Call o Llamada a Procedimientos Remotos.

ejecución de instrucciones en otros procesos.

RPC provee de métodos para que los procesos ejecuten subprogramas en procesadores remotos. Éste es esencialmente síncrono, ya que para que se llame a un *procedimiento remoto*, el proceso cliente llama a un procedimiento *plantilla*[14] que envía una lista de argumentos a otro proceso, el proceso servidor. La lista de argumentos es usada por el proceso servidor en una llamada al procedimiento actual. Después de terminar este procedimiento, los argumentos, posiblemente modificados, se regresan al proceso cliente. El cliente se queda sin hacer nada mientras espera por los resultados del proceso servidor. Esta ineficiencia refleja la naturaleza en la que fue ideado, los sistemas distribuidos.

Los mensajes activos remedian este problema de ineficiencia eliminando la sincronía requerida por RPC. El mensaje enviado por el proceso fuente contiene en una cabecera la dirección de un controlador que residen en el procesador del proceso receptor. Cuando el mensaje llega a su destino, el proceso receptor es notificado vía una interrupción y entra en acción corriendo el manejador. Los argumentos del manejador son los contenidos del mensaje. Por lo tanto no hay sincronía: el primer proceso deposita su mensaje en la red y procede con sus cálculos. Al momento que el mensaje llega al proceso receptor, éste es interrumpido, el manejador invocado y el proceso receptor continúan su trabajo.

3.2.3. Una analogía: Las cajas registradoras

En la vida real no se tienen sistemas que sean puramente de un tipo u otro en hardware y software como ejemplo ilustrativo de cada una de las arquitecturas tanto de hardware como de software presentado, muchas veces se tiene una combinación de ellas, incluso es común encontrar computadoras que no tienen los compiladores necesarios para que tomen ventaja de las características del hardware con que se cuenta, también existen los casos en los cuales hay una combinación de arquitectura de hardware y software, por ejemplo, en el caso del hardware paralelo MIMD de memoria compartida, es común ver que se usen diferentes sistemas de software para programar este tipo de máquinas; el siguiente ejemplo [Radajewski y Eadline, 1999], muestra como podría suceder ésto e ilustra de una manera simple las diferencias entre ellos.

Considere una tienda grande con 8 cajas registradoras agrupadas enfrente de la tienda. Suponga que cada caja registradora es un CPU y que

[14] En la literatura en inglés es conocido como *stub*.

cada cliente es un proceso. El tamaño del proceso (cantidad de trabajo) es el tamaño de cada orden de cada cliente.

- Sistema operativo monotarea

Una caja registradora abierta (está en uso) y debe procesar a cada cliente uno por uno.
Ejemplo: MS-DOS

- Sistema operativo multitarea

Una caja registradora abierta, pero ahora se procesa sólo una parte de la orden de cada cliente en un momento dado, se mueve a la siguiente persona y procesa otra parte de ésta. Todos los clientes parecen estar moviéndose a través de la fila juntos, pero si no hay nadie en la línea en un momento dado, uno puede pasar y procesar mucho más rápido.
Ejemplo: UNIX, NT usando un sólo procesador.

- Sistema operativo multitarea con múltiples procesadores

Ahora si se abren varias cajas registradoras en la tienda. Cada orden puede ser procesada por una caja registradora y la línea puede moverse más rápido (ésto es llamado SMP, Symmetric Multi-processing). Aunque hay varias cajas abiertas, si sólo hay un cliente esperando no podría ir más rápido que si estuviera con una sola caja registradora.
Ejemplo: UNIX, NT con múltiples procesadores.

- Hebras o hilos (threads) en un sistema operativo multitarea con varios procesadores

Si uno parte los elementos de una orden, uno podría ser capaz de moverse en la línea mucho más rápido usando varias cajas registradoras en un momento dado. Primero se debe suponer que se tiene una orden grande porque si uno va a partir estos elementos ésto tomaría tiempo y debería ser mayor al tiempo que se tomaría hacer todo en una sola caja registradora. En teoría, uno debería moverse a través de la línea n veces más rápido, donde n es el número de cajas registradoras abiertas en ese momento. Cuando un cajero necesita obtener subtotales para hacer la suma de todos éstos, éste puede intercambiar la información rápidamente hablando con todos los otros cajeros y viéndose a través de una caja registradora *local*, incluso pueden solicitar información para ser más rápidos por medio de este método. Hay un límite

y éste es el número de cajas registradoras que se pueden poner en un sólo lugar.

La ley de Amdahl explica que la limitante en la velocidad de la aplicación se debe a la parte secuencial más lenta del programa, por lo que esta limita la velocidad del programa.

Ejemplo: UNIX o NT con múltiples procesadores corriendo un programa con multihilos.

- Envío de mensajes en un sistema operativo multitarea con varios procesadores

Para mejorar el rendimiento, la tienda añade 8 cajas registradoras más en la parte trasera de la tienda. Como las nuevas cajas están lejos de las cajas registradoras del frente, los cajeros deben llamar por teléfono para enviar sus subtotales al frente de la tienda. Esta distancia añade una carga extra en tiempo, en la comunicación entre los cajeros, pero si la comunicación es minimizada, ésto no debería ser un problema. Si se tiene una orden grande, que requiriera todos las cajas registradoras, entonces justo como antes, se podría mejorar la velocidad al usar a todos los cajeros al mismo tiempo, el tiempo de comunicación debería ser considerado para que se minimice.

Ejemplo: Una o varias copias de Linux o NT con múltiples CPUs en la misma o diferentes tarjetas madres comunicándose con mensajes a través de una red de interconexión.

El tipo de máquinas paralelas que usaremos son las que caen dentro de esta última categoría, los clusters Beowulf.

3.3. Clusters Beowulf

3.3.1. Un poco de historia

En 1994, Tomas Sterling y Donald Becker, investigadores del CESDIS[15] de la NASA, iniciaron un proyecto en el cual el objetivo era construir una supercomputadora paralela de componentes *de la tienda*[16], es decir una computadora paralela que fuera barata y a la vez eficiente para procesar cantidades muy grandes de datos generados en este centro de investigación. Su primer sistema fue una red 16 de estaciones de trabajo con procesadores 486DX4 conectados a través de una red novedosa llamada *channel bonding*[17] en la

[15] Center of Excellence in Space Data and Information Sciences.

[16] Off the shelf.

[17] La cual consistió en usar múltiples tarjetas de red para repartir el tráfico entre estas de manera eficiente.

cual era posible balancear de una manera óptima el tráfico de la red sin necesidad de usar switches que en esa época eran costosos, con la ayuda de sistemas de envío de mensajes como PVM y MPI, éstos fueron capaces de construir una supercomputadora bastante efectiva con un bajo presupuesto. Al resultado de su trabajo se le conoce como el primer *cluster Beowulf.*

Es inevitable preguntarse por qué razón no fue pensada antes la idea de crear clusters de este tipo. La razón es simple: hasta hace poco la disparidad entre microprocesadores usados en computadoras personales y aquellos usados en supercomputadoras y estaciones de trabajo gráficas era muy grande. Esto ha cambiado, y ahora es posible encontrar en el mercado tecnologías que están muy cerca del desempeño de procesadores de alto desempeño de antaño. Incluso, muchas compañías dedicadas al supercómputo construyen con estos microprocesadores sus nuevas supercomputadoras (no necesariamente clusters).

3.3.2. Arquitectura del Beowulf

Existen tantas definiciones sobre clusters Beowulf como clusters mismos [Radajewski y Eadline, 1999], pero hay dos que son amplias, y que en general se toman como válidas, y en las cuales se basa la gran mayoría de los integradores estas son :

> *"Beowulf es una arquitectura multicomputador la cual puede ser usada para cómputo paralelo. Es un sistema que usualmente consiste de un nodo que actúa como servidor y uno o más clientes conectados vía ethernet o algún otro tipo de red. Es un sistema construido usando componentes de hardware que pueden ser encontrados en cualquier tienda de computadoras y que puedan correr un sistema operativo Linux, con adaptadores ethernet estándares y switches, no contienen algún tipo de hardware no estándar y es trivialmente reproducible. Beowulf también usa sistemas operativos libres, Linux generalmente es este sistema, y sistemas de envío de mensajes ya sean PVM o MPI. El servidor controla el cluster entero y sirve archivos a los nodos cliente. Es también la consola del cluster y la puerta de entrada al mundo exterior. Beowulfs grandes pueden llegar a tener más de un servidor y posiblemente otros nodos dedicados a tareas particulares, por ejemplo, consolas o estaciones de monitoreo y nodos de acceso. En la mayoría de los casos los nodos clientes son* tontos, *entre más tontos mejor y están controlados por el servidor. En*

una configuración sin discos duros, los clientes ni siquiera saben su dirección IP o nombre hasta que el servidor les dice cual es. Una de las principales diferencias entre un cluster Beowulf y un cluster de estaciones de trabajo es el hecho de que Beowulf se comporta más como una sola entidad en vez de estaciones de trabajo independientes. Es la mayoría de los casos, los nodos cliente no tienen teclados ni monitores y son accedidos únicamente vía remota o por medio de una terminal serial. Los nodos de un Beowulf pueden ser pensados como un CPU (o varios en el caso de nodos SMP) y memoria que puede ser conectada al cluster, justo como un módulo de memoria puede ser insertado en una tarjeta madre."

la otra definición ampliamente usada es [Sterling et al., 1998]:

> *"Un Beowulf es una colección de computadoras personales (PC) interconectadas por tecnología de red ampliamente disponible corriendo uno de varios sistemas operativos libres (open source) del tipo Unix. Los programas que corren en Beowulf están usualmente escritos en C o FORTRAN, adoptando el envío de mensajes como modelo para hacer computación en paralelo, aunque también existen alternativas libres y abiertas basadas en estándares que pueden ser utilizadas, como lo son paralelismo a nivel de procesador, memoria compartida (OpenMP, BSP), otros lenguajes (Java, Lisp, Fortran90), y otras estrategias de comunicación (RPC, CORBA)."*

No existe una arquitectura particular de clusters tipo Beowulf. De hecho, la propia definición habla de ésto, por lo que en las siguientes subsecciones se definen de manera muy general la arquitectura de hardware y software que podría usarse para construir uno. Esta parte no pretende ser una lista exhaustiva de componentes de hardware, software y métodos para la administración de un cluster de este tipo. Se pueden encontrar muchas referencias en el web y también en forma impresa [Sterling et al., 1998, Radajewski y Eadline, 1999].

Arquitectura de Hardware

Un Beowulf, es un conjunto de nodos, donde cada nodo generalmente es el hardware que se encuentra comúnmente en una computadora personal. La explotación del poder de este tipo de hardware, o posiblemente de hardware

un poco más especializado, hace que tengamos poder y simplicidad en este tipo de máquinas. Uno puede dejar la toma de decisiones de armar un cluster de este tipo a un *integrador*[18], pero de cualquier manera, es bueno tomar las mejores decisiones, ya que una pequeña diferencia en algún componente puede crear una gran diferencia en el desempeño de las aplicaciones a usarse.

- *Procesador.* Éste es uno de los componentes más críticos de un nodo en un cluster, desde los primeros procesadores usados en el proyecto de la NASA, han incrementado la velocidad en un factor de 8, aún más impresionante es el hecho que el desempeño en operaciones de punto flotante para aplicaciones científicas y de ingeniería ha aumentado por un factor de 40, una PC de nuestros días es varias veces más rápida que el primer Beowulf construido en 1994. Gracias al uso amplio que se le da al sistema operativo Linux y su implementación en otras plataformas, ahora se tiene, además del procesador Pentium, a procesadores, Alpha, Itanium, Opteron y PowerPC G5 para usarlos en la construcción de nodos de un Beowulf. Hoy en día, los procesadores Alpha, Itanium, Opteron y Power PC G5 van a la cabeza del desempeño en operaciones de punto flotante, en particulas Opteron ha mostrado una efectividad en el costo/desempeño en los últimos meses.

- *Cache.* Ésta aparente de una mayor velocidad en la memoria de la que se tiene disponible, ya que como dijimos antes, ésta es comparable a la velocidad que se tienen en los registros, varios niveles de cache pueden ser empleados, 16 kilobytes de nivel 1 (L1) y 512 de nivel 2 (L2) son comunes, los efectos de su uso pueden ser dramáticos, pero no todos los programas se beneficiaran de tener una cantidad grande de cache.

- *Memoria principal.* El acceso a la memoria principal debería ser por lo menos de 70 nanosegundos, esta velocidad puede ser encontrada en módulos SIMM y DIMM de memoria con capacidades desde 128 MB hasta 2 GB.

- *Controlador del disco duro.* Esta unidad es importante ya que maneja la operación del disco duro, las unidades de CD-ROM, espacios temporales de memoria y bloques de datos, además de que controla directamente la transferencia de datos hacia o desde la memoria principal.

[18] Una empresa dedicada a la venta de clusters.

- *Disco duro.* Esta es la unidad de almacenamiento secundario permanente, de capacidades desde un GB hasta los 200 GB actualmente son comúnmente encontrados en el mercado, y hay de dos estándares principales: EIDE y SCSI. El acceso a los datos es sumamente importante, ya que en problemas en los cuales se tienen conjuntos de datos grandes es crucial tener un acceso a escritura al disco duro lo más rápido posible.

- *Controlador del diskette.* Principalmente su uso es para inicio del nodo, sus tiempos de acceso son grandes pero no tiene mucha importancia ya que sólo se usan una vez cada vez que el cluster se reinicia.

- *Tarjeta madre.* Ésta coordina y maneja las interacciones entre los diferentes componentes de un nodo, juega un papel importante en el manejo de la memoria, especialmente para máquinas con múltiples procesadores (SMP) en donde la coherencia del cache es mantenida a través de protocolos implementados en esta tarjeta.

- *Memoria BIOS.* Éstas son un conjunto de instrucciones binarias mínimas que son necesarias para que un nodo pueda tener funcionalidad básica, esta funcionalidad básica consiste en iniciar el sistema y verificar el hardware, los sistemas más modernos cuentan con un BIOS que pueden ser reescritos en caso de que se tengan actualizaciones de estas instrucciones.

- *Bus PCI.* Éste es el estándar universal para controladores de alta velocidad, un bus PCI estándar opera a 32 bits con 33 Mhz. El nuevo estándar ya soporta 64 bits a 66 Mhz.

- *Controlador de video.* Una tarjeta de video convierte señales digitales del procesador en señales analógicas que son usadas para el despliegue de video, las tarjetas modernas tienen procesador en ellas que controlan las operaciones específicas para el uso de video y generalmente tienen varios megabytes de memoria, generalmente no es necesario tener tarjetas de video muy potentes a menos que se esté haciendo operaciones de video o cálculos de escenas.

- *Tarjeta de red.* Ésta es otra parte medular en este tipo de sistemas, ya que como se menciona en la sección 3.2.1, el talón de Aquiles de estos sistemas es que la memoria está distribuida entre varios nodos y en caso de ser necesario una comunicación interprocesos la latencia podría incrementar el tiempo de ejecución de manera crítica. En nuestros días

es común encontrar sistemas con tarjetas de red con anchos de banda de 10/100 Mbps o incluso los más modernos que cuentan con una capacidad de 1000 Mbps.

- *Switch.* Éste es el componente principal para la interconexión entre los nodos, hay de muchos tipos, principalmente se usan switches de 100 Mbps, aunque los de 1000 Mbps se están popularizando, en este rubro es donde se utiliza hardware más especializado, ya que se se pueden usar switches diferentes a los ethernet como myrinet, los cuales hacen menor la latencia.

- *Sistema de enfriamiento.* Dependiendo del número de nodos, el enfriamiento puede llegar a ser crucial, para esto se necesita un análisis del calor que será generado, un buen análisis de esto puede ser encontrado en [Spector, 2000].

Arquitectura de software

Además del hardware, el software juega un papel muy importante en la construcción de un cluster Beowulf. A continuación se describen las partes fundamentales:

- *Sistema operativo.* Ya se ha mencionado el sistema operativo Linux como el sistema operativo seleccionado para este tipo de clusters. Aunque ésto no es impedimento para que se implementen este tipo de clusters con sistemas operativos del tipo BSD[19] o sistemas operativos UNIX comerciales[20]. La ventaja de Linux sobre los demás es que se tiene una base amplia de desarrolladores y este tipo de clusters ha sido adoptado de manera amplia ya que muchos de los miembros de la comunidad de cómputo de alto desempeño y científico hacen modificaciones al kernel de Linux que son distribuidas a todos. Por lo que con ésto se tiene un sistema operativo estándar *de facto.*

- *Sistema de archivos.* Cada nodo necesita tener un área común para poder ejecutar programas. Muchas veces ésto es llevado cabo con la implementación de un sistema de archivos de red. El más común es el sistema NFS, aunque dependiendo de la aplicación a usar, existen otras opciones, si la aplicación hace muchos accesos a disco duro, puede resultar en un cuello de botella por el uso de este sistema de archivos

[19] FreeBSD, OpenBSD, NetBSD.
[20] Tru64, SCO, SunOS.

por red. Alternativas a este sistema de archivos son el Parallel Virtual File System y el Mosix File System.

- *Configuración de red.* Los nodos generalmente están interconectados vía una red local con direcciones IP no homologadas, generalmente del segmento 192.168.X.0. Con lo que al ser nodos de cálculo, sólo pueden tener comunicación vía una red local destinada exclusivamente a este tipo de comunicaciones. Dependiendo del número de tarjetas disponibles en cada nodo, se podría utilizar el *channel bonding*[21] para mejorar el desempeño de la red al usar tarjetas extras.

- *Autenticación.* Una vez dentro del cluster, una máquina es designada a ser el nodo maestro, por lo que generalmente se debe tener acceso a los nodos de cálculo sin tener una contraseña, ya que ésto interferiría con los sistemas de envío de mensajes. Actualmente se puede usar el protocolo SSH para hacer estas operaciones, pero ya que el cluster corre en una red privada un poco de ligereza en la seguridad puede no tener inconveniente.

- *Bibliotecas de envío de mensajes.* Al ser este tipo de cluster de la familia de máquinas MIMD, se tiene que usar algún tipo de sistema de envío de mensajes, PVM fue la elección al principio de este proyecto, pero el estándar MPI ha tomado su lugar en años recientes. Hay varias implementaciones libres de este estándar, las más comunes son MPICH del Laboratorio Nacional Argonne y LAM/MPI de la Universidad de Indiana.

- *Sistemas de procesamiento por lotes.* Es común que en un cluster tipo Beowulf, sea necesario tener a más de un usuario, por lo que el procesamiento de los trabajos a ejecutarse debe ser manejado por un sistema diseñado específicamente para esta tarea, el más popular y que es usado ampliamente es el sistema OpenPBS y su contraparte comercial PBSpro, alternativas son Condor y NQS.

3.3.3. Ventajas de los clusters Beowulf

Comprar una supercomputadora o comprar tiempo en una es costoso, por lo que actualmente es más efectivo usar un cluster Beowulf por la ventaja

[21] Una manera simple de hacer funcionar dos tarjetas de red como una sola, donde cada una de las tarjetas actúa hacia una sola dirección, doblando el ancho de banda.

que tenemos de poder controlar el costo, con el abaratamiento de las computadoras personales, se hace más práctico el crear un cluster Beowulf.

Uno de los grandes problemas del cómputo en paralelo es su poca aceptación y aplicación, incluso entre comunidades académicas. Ésto en parte fue motivado porque cada compañía produjo su propia arquitectura (Cray, Thinking Machines, etc.) no sólo no eran compatibles entre distintas compañías, sino que incluso, diferentes generaciones de hardware podrían no ser compatibles y tenían sus propios lenguajes y bibliotecas paralelas, este problema, es además agudizado con la alta tasa de cierre de las compañías dedicadas al supercómputo.

Los cluster tipo Beowulf se desarrollaron bajo la misma filosofía de cooperación que los sistemas de software libre, por lo que desarrollar sistemas de software para estas máquinas es un proceso de desarrollo comunitario y abierto, ejemplo de ésto está en el desarrollo de implementaciones de MPI y Linux mismo. Actualmente hay una tendencia en los últimos años a mejor adaptarse a tecnologías y estándares abiertos por parte de compañías comerciales como parte de sus estrategias para sobrevivir en un mercado tan pequeño donde el volumen domina.

Las supercomputadoras se tienen partes que son propietarias, tales como buses de memoria, procesadores, medios de comunicación, discos duros, por lo que mucho tiempo y dinero es invertido en investigación para desarrollar este tipo de hardware tan especializado. Esto conlleva a que los precios de éstas sean más altos. En un cluster Beowulf no ha sido necesario hacer este tipo de investigación ya que los nodos de éste están creados con componentes que están disponible a todo el público. Aunque se está un paso atrás del *estado del arte* en el hardware, el desarrollo de la tecnología que va dentro de una computadora personal común y corriente ha avanzado tanto que esa brecha está siendo cerrada cada vez más, ésto hace a un Beowulf altamente efectivo en el plano del costo por hardware.

El impulso ganado por los sistemas operativos libres a finales de los años ochenta ayudaron al desarrollo del Beowulf. En un cluster Beowulf se necesita una copia del sistema operativo en cada nodo perteneciente a éste, usar un sistema operativo propietario subiría los costos por los precios de las licencias, por lo que es natural el uso de software libre y gratuito. De hecho, en las primeras fases del desarrollo de este tipo de clusters fue necesario hacer modificaciones al sistema operativo o extender éste para que se pudieran usar dispositivos que en ese momento no eran usables, particularmente tarjetas de red, aún más, muchos controladores de Linux, para tarjetas de red fueron desarrollados en este centro de la NASA como resultado de la investigación de los clusters tipo Beowulf y ahora son parte medular de este

sistema operativo, muchas áreas abiertas de investigación en ciencias de la computación como protocolos de envío de mensajes más rápidos, memoria compartida-distribuida serían imposibles de llevar a cabo si no se tuviera la libertad de poder examinar, analizar y modificar el código fuente del sistema operativo.

El costo-rendimiento de un cluster Beowulf está entre tres y diez veces con respecto al de una supercomputadora tradicional.

Debido al uso de un sistema operativo multiuso y que éste es usado en cada nodo, es posible hacer varias cosas a la vez, ésto es *paralelismo natural* el cual es fácilmente explotado por más de un CPU de bajo costo.

Las velocidades de los procesadores se están duplicando cada 18 meses, pero la memoria RAM y los discos duros no siguen esta misma tendencia. Desafortunadamente estas velocidades no van aumentando de la misma manera en que los procesadores. Haciendo las cosas en paralelo es una manera de resolver este problema e implementarlos en un Beowulf es la forma más fácil.

Las predicciones indican que las velocidades de procesador no se duplicarán cada 18 meses después del 2005, hay varios obstáculos serios que se tienen que evitar para mantener esta tendencia.

Dependiendo de la aplicación, la computación en paralelo puede aumentar la velocidad de los cálculos de 2 a 500 veces más rápido, en algunas casos incluso más. Tal rendimiento no está disponible usando un sólo procesador. Incluso las supercomputadoras que alguna vez fueron construidas usando procesadores propietarios ahora están siendo construidas usando procesadores *de la tienda.*

3.3.4. Desventajas de un cluster Beowulf

Haciendo referencia a la sección 3.2.1 sobre la arquitectura MIMD, tenemos que el inconveniente de este tipo de arquitectura es la comunicación entre cada procesador. Para computadoras SMP[22] este inconveniente es relativamente simple, sin embargo, también se ha visto que si hay demasiada comunicación entre procesadores el bus tiende a saturarse. Un cluster Beowulf puede estar integrado por nodos que sean computadoras SMP, con lo que se tendrían ambos inconvenientes de la arquitectura. Por un lado en cada nodo la saturación del bus en caso de comunicaciones entre procesos que corren en la misma máquina y por el otro, el más grave, la comunicación entre nodos.

[22]Simmetric Multi-Processor, Multiprocesador simétrico, computadoras con múltiples procesadores y un bus de memoria compartida.

Al tratar de analizar el problema para intentar implementar una solución con un algoritmo en paralelo que sea *tolerante a la latencia*, en casos sencillos donde cada parte a resolver es independiente de las demás y no se necesita mucha comunicación este problema es resuelto muy fácilmente, pero en el caso de un problema en el cual hay mucha intercomunicación entre procesos puede ser tan difícil hasta poder hacer un Beowulf prácticamente inútil para la tarea que se trata de llevar a cabo.

Los clusters Beowulf no son adecuados para cualquier aplicación, aún para aplicaciones en paralelo, y entender las limitaciones de esta arquitectura es importante para poder tener un uso y desempeño útil y productivo.

La escalabilidad y administración son áreas que han alcanzado más importancia, ya que muchos integradores han construido clusters de más de 1000 nodos. Con ésto la administración puede tornarse bastante difícil, y en caso de una mala instalación, imposible para usos prácticos. Más y mejores herramientas se están construyendo, la mayoría de ellas en la arena del software libre como Ganglia[23] y Bproc[24], para hacer estas tareas mucho más simples y que permitan la escalabilidad de estos sistemas.

[23]Sistema de monitoreo para clusters tipo Beowulf, http://ganglia.sourceforge.net.
[24]Interfaz para la interacción intranodos: http://www.scyld.com.

Capítulo 4

MPI

4.1. El modelo de envío de mensajes

Existen varios modelos de programación para el tipo de operaciones disponibles dentro de un programa en paralelo. Esto no es la sintaxis específica de un lenguaje de programación en particular o biblioteca y es casi independiente del hardware. Cualquiera de los modelos pueden ser implementados en cualquier computadora paralela moderna. Con un poco de ayuda de su sistema operativo, la efectividad de tal implementación depende de todas maneras de la brecha entre el modelo y el tipo de máquina en cuestión.

El envío de mensajes es el modelo computacional de interés en este capítulo, este modelo postula un conjunto de procesos que tienen sólo memoria local pero que son capaces de comunicarse con otros procesos a través del envío y recepción de mensajes. Una característica que define este modelo es que los datos transferidos de la memoria local de un proceso a la memoria local de otro requiere que se lleven a cabo operaciones de ambos lados. MPI es una estándar particular de este modelo, como puede observarse en la figura 4.1 no se muestra un tipo específico de red, ya que no es parte del modelo computacional, en particular en los clusters Beowulf la red puede ser tan variada, desde redes simple basadas en el estándar ethernet, pasando por token ring, hasta redes de alta velocidad como lo son las redes myrinet, por lo que este modelo es implementado en una variedad bastante amplia de hardware.Entre los posibles modelos cómputo en paralelo, el envío de mensajes ha sido adoptado ampliamente por su asociación tan cercana con los atributos físicos de la arquitectura de sistema multiprocesador.

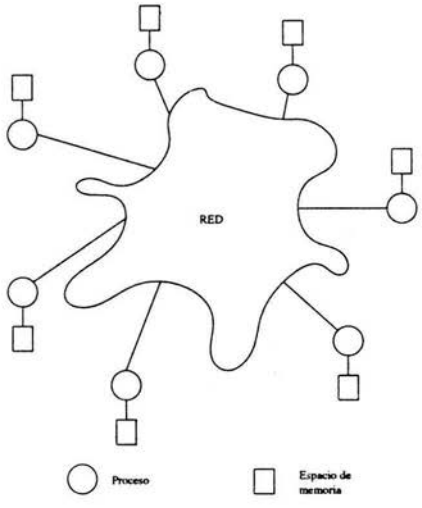

Figura 4.1: El modelo de envío de mensajes

4.1.1. Ventajas del envío de mensajes

El modelo de envío de mensajes en realidad no es el modelo más nuevo ni es superior a los otros modelos. Éste es el que tiende a ser lo más cercano a un aproximamiento estándar a la implementación de aplicaciones en paralelo, algunas de sus características que han hecho de este modelo, y en particular de MPI, uno de los que tienen mayor uso y adopción dentro de la comunidad científica y de cómputo de alto desempeño son listadas a continuación.

Universalidad El modelo de envío de mensajes es adecuado para sistemas con procesadores separados conectados vía una red lenta o de alta velocidad. Por lo que es adecuado para la mayoría del hardware de las computadoras paralelas de nuestros días, así como redes de estaciones de trabajo y clusters Beowulf que compiten con ellas. Cuando estas máquinas incluyen hardware que permiten el uso del modelo de memoria compartida el modelo de envío de mensajes puede aprovechar ésto para acelerar la velocidad de la transferencia de datos.

Expresividad El envío de mensajes ha demostrado ser un modelo útil y completo para expresar algoritmos en paralelo. Provee el control que

hacía falta en los modelos de datos paralelos y basados en compiladores al tratar de manejar datos localmente. Algunos encuentran su estilo útil a la hora de formular un algoritmo en paralelo. Satisface las necesidades en algoritmos adaptativos y de auto manejo de tiempo, y para programas que pueden ser hechos tolerantes para el desbalance en velocidades de procesador que se encuentra en redes de estaciones de trabajo heterogéneas.

Depuración Encontrar errores en programas en paralelo aún prevalece como un problema abierto en las ciencias de la computación. Mientras que los depuradores para programas paralelos son más fáciles de escribir en el modelo de memoria compartida, se puede discutir que el proceso de encontrar errores en sí mismo es más fácil en el paradigma de envío de mensajes. Ésto se debe a que una de las causas más comunes de errores es la sobre escritura inesperada de la memoria. El modelo de envío de mensajes, al controlar las referencias a memoria de una manera más explícita que cualquier otro modelo (sólo un proceso tiene acceso directo a cualquier locación de memoria) hace más fácil localizar errores en escritura y lecturas de memoria. Algunos depuradores paralelos pueden incluso desplegar colas de mensajes, las cuales son normalmente invisibles al programador.

Desempeño La razón por la cual el envío de mensajes podría continuar como una parte importante del cómputo en paralelo es el desempeño. Los procesadores modernos están siendo cada vez más rápidos, el manejo de sus cachés y de la jerarquía en la memoria en general ha sido la clave para obtener el mayor beneficio de estas máquinas. El envío de mensajes provee de un camino al programador para explícitamente asociar datos específicos con los procesos y por lo tanto permitiendo al compilador y al sistema de administración del caché, en el hardware, funcionar al tope. De hecho, uno de las ventajas de la memoria distribuida sobre otras arquitecturas, incluso las computadoras multiprocesadores es que generalmente en conjunto éstas proveen de más memoria principal y caché. Aplicaciones que están limitadas por la memoria pueden obtener mejoras lineales cuando son portadas a este tipo de arquitecturas. Incluso en computadoras de memoria compartida el uso del envío de mensajes puede mejorar el desempeño dándole al programador más control sobre los datos localmente en la jerarquía de memoria.

4.1.2. Desventajas del envío de mensajes

Históricamente, las tecnologías de envío de mensajes reflejaban el diseño de la memoria usada por la computadoras paralelas de esos años. Los mensajes requerían ser copiados, mientras que los threads de memoria compartida usan datos en el lugar. La latencia y la velocidad a la cual los mensajes pueden ser copiados son el factor limitante en este tipo de modelos.

Aunque *todo depende de la aplicación.* Muchas veces si un algoritmo paralelo es tolerante a la latencia, como se verá en el programa desarrollado en este trabajo, la latencia y la velocidad a la que los mensajes son enviados pueden incluso llegar a ser irrelevantes.

El uso explícito de una expresión para un algoritmo paralelo puede llegar a ser difícil de aprender y lleva bastante tiempo.

4.2. MPI[1]

4.2.1. ¿Qué es MPI?

MPI es una biblioteca, y especifica los nombres, secuencias de llamadas y resultados de subrutinas en Fortran, las funciones a ser llamadas en programas escritos en C y las clases y los métodos que conforman la biblioteca en el lenguaje de programación C++[Gropp et al., 1999]. Los programas que son escritos por los usuarios de esta biblioteca utilizan compiladores ordinarios y se ligan con ésta.

MPI no es un lenguaje de programación para programar computadoras paralelas. Es más bien un intento de conjuntar las mejores características de un sistema de envío de mensajes que han sido desarrollados a través de los años para mejorarlas en donde sea apropiado y estadarizarlas.

MPI define la sintaxis y semántica de su biblioteca y los programas son portables en Fortran 77, C y C++, el uso de estas funciones o subrutinas si se está usando Fortran 77, oculta mucho de los detalles de la programación en paralelo haciendo una mejor abstracción respecto a otros modelos.

MPI también es una especificación, no una implementación en particular, todas las compañías que se dedican a la venta de computadoras paralelas se ofrece una implementación de MPI optimizadas para sus máquinas, además de que hay implementaciones libres y gratuitas que pueden ser bajadas de Internet, un programa de MPI correctamente escrito debería ser capaz de correr en todas las implementaciones de MPI adheridas a la especificación sin ningún cambio.

[1] Message Passing Interface, Interfaz de envío de mensajes.

El desarrollo de MPI inició como un esfuerzo de comunidad. El MPI Forum quería definir un sistema de envío de mensajes estándar que pudiera soportar aplicaciones en paralelo y bibliotecas. Este esfuerzo conjuntó a más de 80 personas y 40 organizaciones principalmente de Estados Unidos y Europa, en particular la mayoría de las compañías dedicadas a la venta de sistemas paralelos además de investigadores de universidades, gobierno, laboratorios nacionales y la industria. En abril de 1992, el Centro para la Investigación en Cómputo Paralelo patrocinó un taller sobre estándares para el envío de mensajes en ambientes de memoria distribuida. El resultado de ese taller fue la visión de la gran diversidad de ideas en este tópico y que mucha gente estaba interesada en participar en la definición de un estándar.

En la Conferencia Supercomputing '92[2] en noviembre de ese mismo año, fue formado un comité para definir un estándar de envío de mensajes. El esfuerzo tenía como objetivos definir un estándar portable para el envío de mensajes que no sería un estándar oficial ANSI, sino que integraría tanto implementadores como usuarios y operaría de una manera completamente abierta, permitiendo con ésto que cualquiera pudiera discutir sobre el tema, también tendría como objetivo terminar en un año.

La estandarización de MPI ha sido exitosa al atraer una gran variedad de compañías y usuarios. IBM, Cray, Intel, NEC, Thinking Machines fueron de las compañías que se acercaron a la discusión. Miembros desarrolladores de tecnologías de envío de mensajes como PVM también estuvieron presentes. Incluso desarrolladores especializados en aplicaciones en paralelo también se unieron a este esfuerzo.

El primer estándar MPI fue terminado en mayo de 1994. Varios puntos fueron pospuestos para poder sacar un estándar lo más pronto posible que fuera funcional, las extensiones a este estándar fueron incluidas en 1997. Entre éstas se incluyeron operaciones remotas de memoria, E/S paralelo, administración dinámica de procesos y varias características diseñadas para incrementar la robustez y conveniencia de la biblioteca. Estas nuevas adiciones fueron finalmente liberadas con el nombre de MPI-2 el cual provee de las características no presentes en MPI-1.4 incluyendo herramientas para E/S en paralelo, llamadas para Fortran 90 y administración dinámica de procesos. Actualmente sólo un subcojunto del estándar MPI-2 ha sido implementado, pero aún no está disponible.

El modelo define los siguientes puntos:

- Un cálculo en paralelo consiste en un número de procesos, cada uno trabajando en datos locales. Cada proceso tiene solamente variables

[2] http://www.supercomp.org/history/1992/.

locales y no hay un mecanismo para que algún proceso acceda directamente la memoria de otro.

- La compartición de datos entre los procesos se lleva a cabo por medio del envío de mensajes, ésto es, explícitamente enviando y recibiendo datos.

Nótese que este modelo incluye *procesos*, los cuales en principio no necesariamente tienen que ejecutarse en diferentes *procesadores*. Se supone generalmente que diferentes procesos están ejecutándose en diferentes procesadores y los términos "procesos" y "procesadores" son usados indistintamente.

Una razón principal para la utilidad de este modelo es que es su generalidad, cualquier tipo de computadora en paralelo puede ser adecuada a la manera del envío de mensajes. Además este modelo puede ser implementado en una gran variedad de plataformas, desde computadoras multiprocesadores hasta redes de estaciones de trabajo, incluso computadoras con un sólo procesador.

4.2.2. Conceptos básicos

Esta sección tratará de explicar muchos de los conceptos básicos sobre MPI y su uso.

Los mensajes en MPI y cualquiera que utilice el modelo de envío de mensajes, consisten en múltiples instancias de un programa serial que se comunican por medio de llamadas a funciones de bibliotecas. Estas funciones pueden ser divididas en 4 clases:

1. Funciones para iniciar, manejar y finalmente terminar las comunicaciones.
2. Funciones para comunicar un par de procesos.
3. Funciones para realizar comunicaciones entre conjuntos de procesos.
4. Funciones para crear tipos de datos arbitrarios.

La primera clase de funciones son llamadas para iniciar las comunicaciones, identificar el número de procesadores en uso, crear subgrupos de procesos e identificar qué procesador está corriendo cierta instancia del programa.

La segunda clase de funciones son normalmente operaciones de comunicación *punto a punto* y consisten en diferentes tipos de operaciones de envío y recepción de mensajes.

La tercera clase de funciones son las *operaciones colectivas* que proveen sincronización para cierto tipo de comunicaciones bien definidas entre grupos de procesos y funciones que llevan a cabo operaciones de comunicación y cálculos.

La clase final de funciones proveen flexibilidad al usar estructuras de datos complejas.

Hay un conjunto de conceptos o definiciones básicas que deben de ser entendidas antes de pasar a describir MPI completamente. Ésto con el fin de poder describir las funciones de manera más general posible.

Mensajes y comunicaciones punto a punto

La comunicación más elemental en MPI es la de punto a punto, es decir, una comunicación directa entre dos procesos, uno de los cuales envía y el otro recibe. Para hacer una comunicación de este tipo se tiene que enviar un mensaje explícitamente y también explícitamente recibir el mensaje, por lo que ambos procesos tienen que participar en la comunicación.

En el envío o recepción genérico se transfiere un mensaje que consiste de un bloque de datos. Un mensaje, consta de un *sobre (envelope)*, indicando la fuente y el proceso destino, y un *cuerpo (body)* conteniendo los datos que son enviados.

Para ésto, el mensaje tiene que ser caracterizado, MPI usa tres piezas de información clave para caracterizar a un mensaje de una manera flexible y de las cuales hablaré más adelante, un espacio temporal *(buffer)*, un tipo de datos y un contador.

MPI hace estándar la designación de tipos de datos elementales, por lo que no hay que preocuparse por las diferencias de representación en ambientes en donde hay máquinas heterogéneas.

Modos de comunicación y criterios para terminación

MPI provee flexibilidad en la manera en que deben de ser enviados los mensajes. Hay una variedad de modos de comunicación que definen el procedimiento usado para transmitir el mensaje, así como un criterio para determinar cuando el evento de comunicación, es decir un envío o recepción particular, han sido completados con éxito.

Por ejemplo, un envío síncrono se dice que está completo cuando se notifica al que envía el mensaje que el receptor ha recibido el mensaje completo. Un envío con almacenamiento temporal, por otro lado, es completado cuando los datos a ser enviados han sido copiados a un buffer local. En este caso

nada se puede suponer acerca de si el mensaje ha sido entregado a su destino. En todos los casos, el que un envío esté completo quiere decir que es seguro escribir sobre las locaciones de memoria donde los datos que se iban a enviar se guardaron originalmente.

Hay cuatro modos de comunicación disponibles para los envíos:

1. Estándar
2. Síncrono
3. Con almacenamiento temporal
4. Preparado

Para la recepción sólo hay de un modo. Una recepción es completada cuando los datos entrantes, han llegado completamente y están listos para su uso.

Comunicación con y sin bloqueo

Además del modo de comunicación usado, un envío o recepción puede ser del tipo con o sin bloqueo.

Un envío o recepción bloqueado no regresan de la llamada de la subrutina hasta que la operación ha sido completada. Por lo tanto, asegura que el criterio relevante para la completez ha sido satisfecha antes de que al proceso de llamada le sea permitido seguir adelante. Por ejemplo, con un envío bloqueado, uno se asegura que las variables enviadas pueden ser sobreescritas en el proceso que envía. Con una recepción bloqueada, uno se asegura que los datos han llegado y están listos para usarse.

Un envío o recepción sin bloqueo regresan el control al programa inmediatamente después de la llamada de la función, sin poder dar información si el envío o la recepción fueron completados satisfactoriamente. Ésto tiene la ventaja de que el proceso está libre de poder hacer otras actividades mientras la comunicación se lleva a cabo fuera de línea, uno puede probar después si la operación fue completada. Tomemos como ejemplo un envío no bloqueado y síncrono, éste regresa inmediatamente aunque el envío no estará terminado hasta que se haya notificado que se conluyó. El proceso que envía puede hacer otro trabajo mientras tanto y revisar después si el envío fue completado. No se puede suponer que el mensaje fue recibido o que las variables pueden ser sobreescritas.

Comunicaciones colectivas

Además de las comunicaciones punto a punto, MPI incluye funciones para llevar a cabo comunicaciones colectivas. Estas funciones permiten que un grupo más grande de procesos se comuniquen de varias maneras. Por ejemplo, uno a varios o varios a uno. La principal ventaja del uso de este tipo de funciones es que no es necesario escribir las funciones equivalentes por medio de funciones básicas. La posibilidad de error es reducida ya que una línea de código reemplaza a varias líneas de código de llamadas a funciones de comunicación de punto a punto. El código fuente es más entendible a la hora de leerlo, por lo que se simplifica el mantenimiento y el depuramiento del programa. Generalmente formas optimizadas de las funciones colectivas son más rápidas que la operación equivalente expresada en términos de funciones punto a punto.

Las operaciones colectivas son operaciones de transmisión, operaciones de juntado y dispersión y operaciones de reducción. Sobre éstas se discute más adelante, en la sección 4.2.4.

Compilación y ejecución de programas con MPI

El estándar MPI no establece una forma única de compilación y de ejecución, por lo que las implementaciones varían de máquina a máquina, cuando se compila un programa podría ser necesario ligar a la biblioteca de MPI, pero muchas veces las implementaciones traen una *envoltura* que se encarga de hacer ésto, se hablará un poco más sobre la compilación y ejecución del programa creado para este trabajo en la sección 4.2.5.

Estructura de un programa de MPI

La estructura básica de un programa en MPI usando el lenguaje de programación C es la siguiente:

```
incluir archivo de encabezados de MPI
declaración de variables
inicializar el ambiente de MPI
hacer el cálculo y llamadas a funciones de MPI pa-
ra comunicación
cerrar las comunicaciones de MPI
```

Primero, el archivo de encabezados de MPI contiene las definiciones de macros, prototipos, constantes especiales y tipos de datos de MPI, por lo que

hay que incluirlo con la sentencia include de C, generalmente se hace de la siguiente manera:

```
#include <mpi.h>
```

Después le siguen las declaraciones de variables, cada programa debe de llamar a una función de MPI que inicializa el ambiente para el envío de mensajes, esta es la función:

```
MPI_INIT (&argc, &argv);
```

Nótese el uso de las direcciones de *argc* y *argv*, variables que contienen las opciones de la línea de comandos del programa, todas las llamadas a funciones de MPI deben de venir después de la inicialización, finalmente, el programa debe de llamar a la subrutina que termina MPI:

```
MPI_Finalize ();
```

ninguna función de MPI puede ser llamada después de esta función. De hecho, si algún proceso no llama a esta función parecerá que el programa se congela, ésta limpia todas las estructuras de datos de MPI, cancela las operaciones que no se completaron; esta función debe de ser llamada por todos los procesos ya que en caso contrario los programas parecerán en estar en pausa o congelados.

Un concepto importante en el mundo de MPI es el de *comunicador (communicator)*. Éste es una referencia a una estructura de datos que puede ser usado para representar a un grupo de procesadores que pueden comunicarse entre sí. El nombre del comunicador es un argumento requerido en todas las funciones de comunicación colectivas. El comunicador en un envío y recepción siempre debe de ser el mismo si se espera que se lleve a cabo la comunicación, un par de procesos sólo se pueden comunicar entre sí, si comparten un comunicador, puede haber varios comunicadores, y un proceso puede ser miembro de varios comunicadores. Dentro de cada comunicador, los procesos son numerados consecutivamente, comenzando desde el cero, este número identificador es conocido como el *rango* o *rank* del procesador en ese comunicador, el rango es también usado para especificar el origen y destino en una función de envío o recepción. Además, si un proceso pertenece a más de un comunicador, usualmente su rango en cada uno será diferente.

MPI provee un comunicador básico llamado:

```
MPI_COMM_WORLD
```

el cual es el comunicador más general que consiste en el que incluye todos los procesos, se usa este comunicador si se requiere que todos los procesos se comuniquen entre sí. Hay una función que sirve para determinar el rango en el que se está en un proceso determinado, esta es la función:

```
int MPI_Comm_rank (MPI_Comm comm  /* in  */,
                   int      *rank /* out */);
```

el primer argumento de esta función es un comunicador, podriamos usar `MPI_COMM_WORLD`, el segundo argumento es una variable que tendrá el valor del rango en el que se está. Otra función muy útil es:

```
int MPI_Comm_size (MPI_Comm comm  /* in  */,
                   int      *size /* out */);
```

donde el primer argumento es un comunicador y el segundo es la dirección a una variable entera que tendrá el valor del número de procesos.

Más adelante se verán las funciones para hacer comunicación punto a punto y comunicaciones colectivas.

La última función que se debe usar en un programa en MPI es la función:

```
int MPI_Finalize (void);
```

4.2.3. Comunicación punto a punto

La comunicación punto a punto es la base de las capacidades de comunicación que provee la biblioteca MPI. Conceptualmente es bastante simple. La comunicación punto a punto requiere de la participación activa de los procesos en ambos lados, un proceso *fuente (source)* envía y otro proceso *destino (destination)* recibe la información.

En general la fuente y el destino operan asíncronamente, incluso el envío y recepción de un mensaje es típicamente no sincronizado, el proceso fuente puede completar un envío de mensaje mucho antes de que el proceso destino esté listo para recibirlo y al contrario también, un proceso destino puede iniciar un proceso de recepción mucho antes de que el proceso fuente lo envíe.

De esta manera se tiene que al ser los envíos y recepciones operaciones que típicamente no están sincronizadas, los procesos tienen uno o más mensajes que han sido enviados pero no aún recibidos. A éstos se les conoce como mensajes pendientes, una característica importante de MPI es que los

mensajes pendientes no son mantenidos en una cola tipo FIFO[3], sino que cada mensaje pendiente tiene varios atributos y el proceso destino puede usar esos atributos para determinar que mensajes recibir.

Los mensajes constan de dos partes, el sobre y el cuerpo del mensaje. El sobre de un mensaje de MPI es análogo a un sobre de papel en el correo común y corriente. Éste generalmente contiene el destino, la dirección desde donde es enviado y cualquier otra información pertinente para que se envíe una carta. Con esta analogía en mente, se tiene que el sobre de un mensaje de MPI tiene 4 partes:

1. Fuente, el proceso que envía el mensaje.
2. Destino, el proceso que recibirá el mensaje.
3. Un comunicador.
4. Una etiqueta, que servirá para clasificar los mensajes.

La etiqueta es requerida, pero su uso es dejado al programador, un par de procesos que se comunican pueden usar etiquetas para distinguir entre clases de mensajes.

Las tres partes que constituyen a el cuerpo del mensaje, son las siguientes:

1. *Buffer*, la dirección de un arreglo de elementos que son los que serán enviados o recibidos.
2. *Datatype*, el tipo de datos que será enviado. En los casos más simples éste es de tipo elemental como float, int, etcétera. En aplicaciones más avanzadas estos tipos pueden ser definidos por el usuario y construidos por medio de los tipos de datos elementales. Esta característica permite que el contenido de los mensajes pueda ser muy flexible.
3. *Count*, es el número de elementos de tipo Datatype que son enviados.

Se puede pensar en el espacio temporal como si fuera un arreglo, la dimensión está dada por el contador o *count* y el tipo del arreglo es dado por el *tipo de datos* o *datatype*. Ésto permite que las comunicaciones sean transparentes entre procesadores que no son homogéneos.

Enviar un mensaje es simple: el emisor del mensaje se determina implícitamente, pero el resto del mensaje, sobre y cuerpo, son dados explícitamente por el proceso emisor.

[3]First In First Out, el primero que entra es el primero que sale.

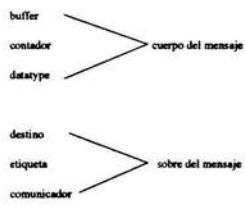

Figura 4.2: Argumentos a la función `MPI_Send`

Recibir un mensaje no es tan simple. Un proceso puede tener varios mensajes pendientes. Para recibir un mensaje un proceso especifica un sobre que MPI compara con los sobres de mensajes pendientes, si alguno es igual entonces el mensaje es recibido, si no es así la recepción no es completada hasta que se envía un mensaje con un sobre igual al especificado por el receptor. Además de ésto, el proceso receptor debe de proveer de suficiente almacenamiento en el cual el cuerpo del mensaje se va a copiar. El proceso receptor debe dar el suficiente espacio para almacenar el mensaje completo.

Envío y recepción con bloqueo

El envío y recepción con bloqueo son las funciones más básicas de comunicación punto a punto. Ambas funciones bloquean el proceso de llamada de la función hasta que la operación es terminada. Este tipo de comunicación crea la posibilidad de un *candado* o *deadlock*, el cual se explica a continuación.

`MPI_Send` toma los argumentos presentados en la figura 4.2. El cuerpo del mensaje contiene los datos a ser enviados, los elementos son contados por `count`, y son de tipo datatype. El sobre del mensaje dice a donde enviarlo, además un valor de error es regresado, el prototipo de la función es el siguiente:

```
int MPI_Send (void         *buff  /* in */,
              int          count  /* in */,
              MPI_Datatype dtype  /* in */,
              int          dest   /* in */,
              int          tag    /* in */,
              MPI_Comm     comm   /* in */);
```

`MPI_Recv` toma un conjunto de argumentos muy similares a MPI_Send, éstos son mostrados en la figura 4.3, pero muchos de ellos son usados de manera

Figura 4.3: Argumentos a la función `MPI_Recv`

diferente:

```
int MPI_Recv (void         *buff    /* out */,
              int          count    /* in  */,
              MPI_Datatype dtype    /* in  */,
              int          source   /* in  */,
              int          tag      /* in  */,
              MPI_Comm     comm     /* in  */
              MPI_Status*  status   /* out */);
```

Los argumentos en el sobre del mensaje determinan que mensajes pueden ser recibidos por la llamada a esta función. Los argumentos fuente, etiqueta, comunicador deben de ser iguales a los de algún mensaje pendiente para que sea recibido. Se pueden utilizar argumentos comodines para la fuente, para aceptar mensajes desde cualquier proceso y la etiqueta, para aceptar mensajes con cualquier valor de la etiqueta. Para el comunicador no hay argumentos comodines. El programador debe tomar en cuenta el tipo de dato que está enviando y de lo que se está recibiendo, ya que debe hacer que ambos estén concuerden. Por otro lado el argumento de `estatus` regresa información sobre la situación del mensaje que fue recibido.

Deadlock o candado

Cuando dos o más procesos son bloqueados y cada uno está esperando por el otro para proseguir se le llama un *candado (deadlock)*, como cada proceso no puede seguir porque depende del otro para poder continuar ninguno puede continuar.

Para evitar esta situación se requiere de una organización cuidadosa de la comunicación en un programa, el programador debe de ser capaz de explicar porqué el programa se bloqueará o no de esta manera.

Envío y recepción sin bloqueo

Ya vimos que tanto `MPI_Send` como `MPI_Recv` bloquean el proceso de llamado de funciones. Ninguna de las dos regresan hasta que la operación de comunicación ha sido completada. El requerimiento para que la operación de comunicación se complete puede causar retrasos e incluso un candado.

MPI tiene otra manera de invocar operaciones de envío y recepción, es posible separar la iniciación de un envío o recepción de su completez haciendo dos llamadas separadas a MPI. La primera llamada inicia la operación y la segunda llamada la completa, entre estas dos llamadas el programa está libre de hacer otras cosas.

La llamada para iniciar una operación de envío se le conoce como un posteo de envío, y similarmente, con una operación de recepción, se le conoce como un *posteo* de recepción. Una vez que una operación de envío o de recepción han sido posteadas, MPI provee de dos maneras distintas de completarlas, la primera es que un proceso puede probar para ver si la operación ha sido completada sin bloquear a la hora de la terminación, alternativamente, un proceso puede esperar a que la operación se complete.

Después de postear un envío o recepción con una llamada a una función sin bloqueo, el proceso necesita de alguna manera referirse a la operación posteada. MPI usa un identificador de la operación que fue posteada por la llamada a la función, éste puede ser usado para revisar el estatus de la operación posteada o para esperar por su completez. Las funciones de envíos y recepciones sin bloqueo regresan todas un identificador del que hablamos en la oración anterior[4], los cuales son usados para identificar la operación posteada por la llamada.

La función `MPI_Isend` es usada para postear un envío sin bloquear a la completez de una operación de envío, la función es muy similar a la llamada de `MPI_Send`, pero incluye un argumento de salida adicional, el identificador del cual se habló en el párrafo anterior:

```
int MPI_Isend (void         *buff   /* in  */,
               int          count   /* in  */,
               MPI_Datatype dtype   /* in  */,
               int          dest    /* in  */,
```

[4]También conocido como un *request handle*.

```
                int          tag     /* in  */,
                MPI_Comm     comm    /* in  */
                MPI_Request* request /* out */);
```

La función `MPI_Irecv` es similar a la anterior:

```
int MPI_Irecv (void         *buff    /* out */,
               int          count    /* in  */,
               MPI_Datatype dtype    /* in  */,
               int          source   /* in  */,
               int          tag      /* in  */,
               MPI_Comm     comm     /* in  */,
               MPI_Request* request  /* out */);
```

Al postear un envío o una recepción éstas deben de ser terminadas, si un envío o recepción son posteados por una función sin bloqueo el estatus de terminación puede ser checado por una familia de funciones de prueba. MPI provee funciones tanto con bloqueo como sin bloqueo para terminar, la funciones con bloqueo son `MPI_Wait` y sus variantes, las sin bloqueo son `MPI_Test` y sus variantes.

Un proceso que ha posteado un envío o recepción con una función sin bloqueo, puede esperar por la terminación de la operación posteada llamando, como ya mencionamos, a la función `MPI_Wait`:

```
int MPI_Wait( MPI_Request *request /* in/out */,
              MPI_Status  *status  /* out    */);
```

Un proceso que ha posteado un envío o recepción con una función sin bloqueo, puede probar por la completez de la operación posteada llamando como ya mencionamos, a la función `MPI_Test`:

```
int MPI_Test( MPI_Request *request /* in/out */,
              int         *flag    /* out    */,
              MPI_Status  *status  /* out    */);
```

El uso cuidadoso de estas funciones hace mucho más fácil escribir programas libres de candados. Ésta es una gran ventaja ya que es fácil introducir candados sin intención. En computadoras paralelas donde las latencias son grandes, postear envíos y recepciones con anticipación es una manera simple y sencilla de enmascarar las comunicaciones con que se cuentan, las latencias tienden a ser grandes en sistemas distribuidos e independientes y pequeñas

Modo de envío	*Función con bloqueo*	*Función sin bloqueo*
Estándar	`MPI_Send`	`MPI_Isend`
Síncrono	`MPI_Ssend`	`MPI_Issend`
Preparado	`MPI_Rsend`	`MPI_Irsend`
Acumulado	`MPI_Bsend`	`MPI_Ibsend`

Cuadro 4.1: Funciones y modos de envío

en sistemas de memoria compartida. En general, enmascarar la comunicación requiere mucha atención a la estructura del programa y los algoritmos usados.

Por otro lado, la desventaja del uso de estas funciones para la comunicación puede incrementar la complejidad del código haciéndolo más difícil de mantener y de depurar.

Modos de envío

MPI tiene cuatro diferentes tipos de modos de envío, el modo estándar, modo síncrono, modo preparado y modo de almacenamiento temporal.

El modo estándar es el modo más conocido y se ha usado hasta ahora, cuando MPI ejecuta un envío en modo estándar, una de dos cosas sucede, o el mensaje es copiado en un buffer interno de MPI y transferido asíncronamente a su proceso destino o la fuente y el proceso destino se sincronizan con este mensaje, ésto quiere decir que la implementación es libre de escoger entre las dos opciones, la que sea mejor, ya sea dependiendo del tamaño del mensaje, disponibilidad de recursos, etcétera. Una de las ventajas del envío en modo estándar es justamente ésto, la elección entre guardar en un buffer o sincronizar.

El modo síncrono requiere que MPI sincronice el proceso de envío y recepción. Cuando una operación de envío síncrona es completada el proceso emisor puede suponer que el proceso destino ha empezado a recibir el mensaje, el proceso destino puede no haber terminado de recibir el mensaje pero debe haber empezado a recibirlo, la llamada sin bloqueo tiene la misma ventaja sobre éste, el proceso emisor puede evitar bloquearse en una operación de envío potencialmente grande.

El modo preparado de envío requiere que un proceso de recepción haya sido posteado en el proceso destino antes de que este modo haya sido enviado. Si no es así, el resultado es indefinido. Es responsabilidad el programador asegurarse de que esta limitante sea encontrada. En algunos casos, el cono-

cimiento del estado del proceso destino es disponible sin hacer alguna tarea extra. La carga en la comunicación puede ser reducida porque protocolos más cortos pueden ser usados internamente por MPI cuando es sabido que una recepción ha sido posteada. Una llamada sin bloqueo tiene ventajas similares a este modo también, el proceso emisor puede evitar bloquearse en una operación de envío potencialmente grande.

El modo acumulado requiere que MPI use almacenamiento temporal, el inconveniente acá es quese necesita manejar el buffer explícitamente. Si en algún punto, el espacio en el buffer no es suficiente o no está disponible para completar una llamada, los resultados son indefinidos. Todas estas funciones son mostradas en el Cuadro 4.1.

4.2.4. Comunicaciones colectivas

Muchas veces es necesario hacer comunicaciones y transferencia de datos en donde estén involucrados todos los procesos pertenecientes a cierto comunicador. Además, este tipo de comunicaciones son tan comunes que escribir estas funciones cada vez es propenso a errores. Para estos casos sería útil tener funciones que lleven a cabo este tipo de comunicaciones bien definidas, ya que aunque se podrían construir desde cero. Sería complicado tratar de escribir estas comunicaciones con funciones de comunicación básicas y el programa podría crecer también en complejidad la cual es escondida por este tipo de funciones, estas funciones son conocidas como funciones de comunicaciones colectivas y son descritas en esta sección.

Hay que notar que estas comunicaciones no usan etiquetas como lo hacen las funciones de envío y recepción de la sección anterior. En vez de ésto, estas funciones son asociadas por orden en la ejecución del programa y por lo tanto el programador debe de asegurar que todos los procesos ejecutan la misma comunicación colectiva en el mismo orden. La noción de comunicadores es importante en este tipo de operaciones, ya que es necesario la identificación de los procesos que serán parte de la operación colectiva. Cabe recalcar que el usuario puede definir sus propios comunicadores, que aunque no es descrito en este trabajo, es una parte fundamental de MPI y puede llegar a ser de mucha utilidad en problemas más complejos.

Sincronización de barrera

Hay ocasiones en donde algunos procesadores no pueden proseguir con la ejecución de un programa hasta que otros procesadores hayan terminado sus propias instrucciones. Por ejemplo, suponga que un proceso raíz tiene

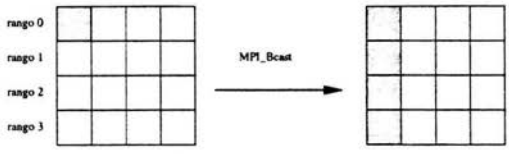

Figura 4.4: Transmisión

que leer datos y tiene que transmitir estos datos a otros procesos, los demás procesos deberían poder esperar hasta que la entrada y salida esté completa y los datos sean movidos.

La función `MPI_Barrier` bloquea o hace una pausa hasta que todos los procesos que pertenecen al comunicador que es de interés han llamado a la función. Cuando `MPI_Barrier` regresa, todos los procesos están sincronizados en el punto de esa barrera. `MPI_Barrier` está implementada en software por lo que puede llegar a cargar sustancialmente la carga en algunas máquinas. Por lo que generalmente se deben de insertar barreras en lugares donde es realmente necesario. El prototipo de la función es el siguiente:

```
int MPI_Barrier ( MPI_Comm comm /* in */);
```

Transmisión

La operación de transmisión es la más simple de las operaciones colectivas. En una operación de transmisión, todos los procesos especifican el nombre del proceso raíz el cual tiene el buffer que será enviado, y este proceso raíz envía a todos los procesos restantes una copia de los datos a los procesos elementos del comunicador. Esta operación es mostrada gráficamente en la figura 4.4, cada renglón en la columna representa un proceso diferente cada renglón gris en una columna representa la locación de datos. Bloques con el mismo patrón que están en procesos múltiples contienen copia de los mismos datos.

La función `MPI_Bcast` nos permite copiar datos de la memoria del procesador raíz a las mismas locaciones de memoria de los demás procesos contenidos en el comunicador:

```
int MPI_Bcast ( void*        buffer /* in/out */,
                int          count  /* in     */,
                MPI_Datatype dtype  /* in     */,
                int          rank   /* in     */,
```

Operación	Descripción
MPI_MAX	máximo
MPI_MIN	mínimo
MPI_SUM	suma
MPI_PROD	producto
MPI_LAND	conjunción lógica
MPI_BAND	conjunción binaria
MPI_LOR	disyunción lógica
MPI_BOR	disyunción binaria
MPI_LXOR	disyunción exclusiva lógica
MPI_BXOR	disyunción exclusiva binaria
MPI_MINLOC	mínimo y locación
MPI_MAXLOC	máximo y locación

Cuadro 4.2: Operaciones de reducción

```
                MPI_Comm      comm    /* in      */);
```

Reducción

Una reducción es una operación colectiva en la cual el proceso raíz, colecta datos de otros procesos en un comunicador y los combina en un sólo elemento. Gráficamente ésto es representado por la figura 4.5. Por ejemplo, si se colecta el valor de una variable que está presente en todos los procesadores y luego sumar esos valores y se obtiene la suma total de éstos. Además de operaciones aritméticas son posibles las operaciones que aparecen en el cuadro 4.2.

La función `MPI_Reduce` hace justo estas tareas: colecta los datos de cada procesador, reduce estos valores a un valor único, dependiendo de la operación que se defina, y guarda estos valores en el proceso raíz:

```
int MPI_Reduce ( void*        send_buffer /* in  */,
                 void*        recv_buffer /* out */,
                 int          count       /* in  */,
                 MPI_Datatype datatype    /* in  */,
                 MPI_Op       operation   /* in  */,
                 int          rank        /* in  */,
                 MPI_Comm     comm        /* in  */);
```

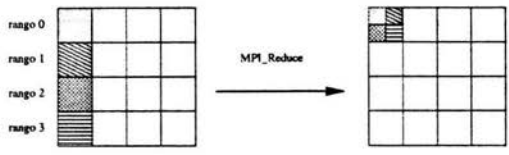

Figura 4.5: Reducción

Distribución

Distribuir es una de las operaciones colectivas más importantes ya que distribuye datos contenidos en un proceso a un grupo de procesos de manera equitativa en una operación de distribución todos los datos, un arreglo generalmente, es inicialmente colectado en un procesador único, después de la operación de distribución, piezas de esos datos están distribuidos en diferentes procesos, la diferencia de los patrones de la figura 4.6 refleja la posibilidad de los datos no seas divisibles exactamente entre todos los procesadores.

La función `MPI_Scatter` lleva a cabo esta operación, cuando es llamada, el proceso raíz divide en partes iguales, si es posible, un conjunto de locaciones de memoria contigua y envía cada pedazo a cada procesador, el resultado es igual a hacer N envíos con `MPI_Send` y cada procesador hubiera hecho un `MPI_Recv`:

```
int MPI_Scatter ( void*        send_buffer /* in  */,
                  int          send_count  /* in  */,
                  MPI_datatype send_type   /* in  */,
                  void*        recv_buffer /* out */,
                  int          recv_count  /* in  */,
                  MPI_Datatype recv_type   /* in  */,
                  int          rank        /* in  */,
                  MPI_Comm     comm        /* in  */);
```

Juntado

La operación de juntado es justo la opuesta a una operación de distribución como se puede ver en la figura 4.7. En ésta, los datos son colectados de diferentes procesos y son reensamblados en un arreglo y en el orden correcto en un sólo proceso.

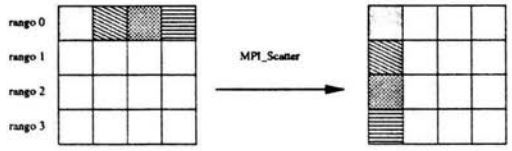

Figura 4.6: Distribución

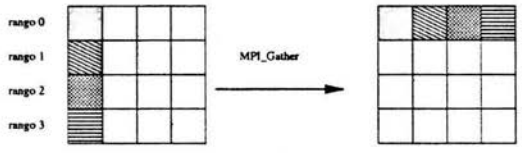

Figura 4.7: Juntado

La función **MPI_Gather** lleva a cabo esta operación, los argumentos son iguales a la función de repartición, cuando es llamada, cada proceso, incluyendo el proceso raíz, envían el contenido almacenado en el buffer al proceso raíz, el proceso raíz recibe los mensajes y los guarda en orden de rango:

```
int MPI_Gather ( void*        send_buffer /* in  */,
                 int          send_count  /* in  */,
                 MPI_datatype send_type   /* in  */,
                 void*        recv_buffer /* out */,
                 int          recv_count  /* in  */,
                 MPI_Datatype recv_type   /* in  */,
                 int          rank        /* in  */,
                 MPI_Comm     comm        /* in  */);
```

Con el grupo de funciones definidas hasta ahora se ha creado el programa que es la base de este trabajo, por lo que si se quiere profundizar aún más en las funciones de MPI se pueden encontrar discusiones más amplias sobre la forma en que trabajan éstas y el estándar completo de MPI en el MPI Forum[5] y en [Pacheco, 1997], [Gropp et al., 1999] y [Murray y VanRyper, 1996] .

[5] http://www.mpi-forum.org

4.2.5. LAM/MPI

LAM/MPI es una implementación de alta calidad de la especificación de MPI, desarrollado por el *Open Systems Lab* en la Universidad de Indiana, incluye toda la definición de MPI 1.2 y la mayor parte de MPI 2,. Ésta no es sólo la biblioteca definida por el estándar MPI, sino también el ambiente necesario para que programas que usan estas bibliotecas puedan correr correctamente ya que requieren de ciertos servicios al ser ejecutados. Ambos componentes (la biblioteca y el ambiente) de LAM/MPI están diseñados dentro de un marco de componentes por lo que son extendibles con pequeños módulos que pueden ser elegidos y configurados al tiempo de ejecución, este marco de componentes es conocido como System Services Interface o *SSI*, los componentes de SSI están ampliamente documentados. Además de ésto, LAM/MPI tiene algunas características que lo hacen más robusto a otras implementaciones de MPI como herramientas de monitoreo y de depuración, soporte para ambientes heterogéneos, se pueden añadir y quitar nodos, hay un sistema de detección y de recuperación de nodos caídos, e implementa algunas características de MPI-2 y la comunicación multiprotocolo, ésto es tanto soporte para memoria compartida como de red.

LAM corre en cada nodo como un servidor estructurado como un nanokernel. Este nanokernel provee del envío de mensajes a los procesos locales, algunos de los procesos internos de LAM forman una subsistema de red de comunicación los cuales transmiten los mensajes de un demonio de LAM a otras máquinas.

La forma en que está estructurado LAM es transparente para los usuarios, quienes únicamente pueden observar a un servidor ejecutándose. El mecanismo para ejecutar un programa e iniciar LAM está más allá del alcance de este trabajo, pero una documentación completa para realizar ésto se encuentra en [GDB y RBD, 1996].

LAM es la implementación de MPI que se ha usado en este trabajo, y fuera del mecanismo de inicio y ejecución de este ambiente, la forma de trabajo es estándar para cualquier implementación de MPI que se utilice.

Capítulo 5

Cálculo en paralelo de los diagramas de Liapunov-Markus

En este capítulo se proponen las estrategias para el programa paralelo que servirá para calcular las figuras discutidas en el capítulo 2. Con esto, lograremos que estas figuras sean generadas de manera más rápida y eficiente en un cluster de tipo Beowulf usando el envío de mensajes como nuestro modelo de programación.

La programación en paralelo es una tarea más complicada que la programación secuencial, algunas veces los problemas que se atacan no son de naturaleza paralela y no tienen paralelismo inherente, y como resultado de la ley de Amdahl [Pacheco, 1997], pueden ser escalados hasta cierto punto. Un programa es escalable si incrementado el tamaño de los datos podemos obtener un buen desempeño mientras el programa usa más y más procesos.

El diseño de un programa en paralelo puede comenzar estudiando la solución serial. Muchas veces y en este caso así parece, se tiene la suerte que el programa serial ya resolvió muchas de las partes del programa paralelo y únicamente tenemos que repartir los datos y que cada procesador se haga cargo de ellos. Esta forma de atacar el problema parece ser más bien un poco intuitiva y menos formal, por lo que para llevar a cabo de manera metódica el diseño del programa usaremos cuatros pasos generales, *Partición, Comunicación, Aglomeración y Mapeo* [Foster, 1995].

5.1. Partición

El primer paso para diseñar un algoritmo en paralelo es descomponer el problema en tareas más pequeñas. Entonces, estas tareas pequeñas son asignadas a cada procesador para que éstos trabajen en ellas, a la forma de dividir el problema en tareas se le conoce como partición. Hay dos tipos de modelos de partición [Foster, 1995]:

- *Descomposición de dominio*: En este tipo de descomposición, también conocido como *paralelismo de datos*, los datos son divididos en piezas de aproximadamente el mismo tamaño, de ahí son mapeados a cada procesador. Cada procesador trabaja entonces en la porción que le corresponde o que le fue asignada, por supuesto, estos procesadores podrían necesitar comunicarse entre sí para intercambiar datos. Este tipo de descomposición tiene la ventaja que mantiene un flujo de control único. Un algoritmo en paralelo consiste en una secuencia de instrucciones elementales aplicadas a los datos, una instrucción es iniciada solamente si la instrucción anterior ha terminado. SPMD sigue este modelo cuando el código es el mismo en todos los procesadores. Este tipo de descomposición es muy usada en problemas de diferencias finitas en donde los procesadores pueden operar independientemente en porciones grandes de datos, comunicándose de vez en cuando para intercambiar quizá los bordes de los datos que tienen a su cargo. Un ejemplo de este tipo de problema es el que se está tratando de resolver.

- *Descomposición funcional:* Frecuentemente la descomposición de dominio llega ser una estrategia que no es la más eficiente para un programa en paralelo. Por ejemplo cuando las estructuras de datos asignadas a diferentes procesos requieren diferentes tiempos para procesarse, el desempeño del código es entonces limitado por la velocidad del proceso más lento. El tiempo que no se están usando los procesadores restantes es entonces desperdiciado. En este caso, la descomposición funcional o *paralelismo de tareas* tiene más sentido que la *descomposición de dominio*. En esta descomposición el problema es partido en un número grande de tareas pequeñas y luego las tareas son asignadas a los procesadores, los procesadores que terminan más rápido simplemente se les asigna más trabajo. El paralelismo de tareas es implementado en un paradigma cliente-servidor. Las tareas son colocadas a un grupo de procesos esclavos por un proceso maestro que también lleva a cabo algunas de las tareas, este paradigma puede ser implementado en prácticamente cualquier nivel de un programa, por ejemplo si simplemente

se desea ejecutar un programa con múltiples entradas. Una implementación de cliente-servidor en paralelo podría simplemente correr copias del proceso serial en el servidor asignándoles diferentes parámetros a cada cliente, al terminar cada proceso su tarea, se le asignan nuevos parámetros. Alternativamente, el paralelismo de tareas puede ser implementado en un nivel más profundo dentro del código.

Generalmente es más fácil y natural implementar un programa usando paralelismo por datos, en el caso que atiene a este trabajo se tiene la suerte de que hacer la descomposición de dominio es un paso natural en el problema, aunque implícitamente también se está haciendo una descomposición funcional.

Recuérdese que se tiene una una matriz de valores pares AB, que es la ventana de interés, y para cada valor AB en esta matriz se calcula un único punto, λ, que es el valor del exponente de Liapunov. En realidad estamos trabajando con una única matriz. Esta matriz puede ser repartida entre cada procesador de tal manera que si tenemos n procesadores, y la matriz es de tamaño $m \times m$ a cada procesador le tocaría aproximadamente el mismo número de pares AB para calcular valores del exponente de Liapunov para ese par. La forma en que se repartiría para este caso en particular, es decir, la implementación, se verá de manera más clara en la sección 5.3.

5.2. Comunicación

La comunicación es una parte fundamental de un algoritmo en paralelo, por lo que se tratar de minimizar las comunicaciones, en el caso referente a este trabajo, cada procesador debe tener una parte de la matriz de valores AB para poder calcular la parte que le corresponde, por lo que en un principio, se deben de tener comunicaciones cuando cada procesador reciba la parte de esa matriz que le corresponda[1].

Por otro lado, una vez calculado el pedazo de matriz que le corresponde a cada procesador estos valores tienen que ser escritos en algún lado para ser procesados después, por lo que tienen que ser reunidos en un sólo procesador para que éste se encargue de escribir estos datos, por lo que cada procesador debe de enviar al procesador que se encargaría de esta tarea los datos que generó. Ésta sería una operación global de juntado.

Recuérdese que una operación global generalmente produce problemas de escalabilidad, es decir, al agregar nuevos procesadores se incrementa el

[1]Naturalmente se haría con una operación global de repartición (`MPI_Scatter`).

número de mensajes que son necesarios para completar la comunicación. Incluso, al aumentar el tamaño del problema, el tamaño de los mensajes es más grande, haciendo con eso más grande el tiempo de comunicación también. Por lo que se tiene que considerar en el diseño para la optimización del algoritmo usado en el programa paralelo. En las secciones 5.5 y 5.7 se podrá ver que para el problema particular a este trabajo, el uso de las comunicaciones es mínimo y el algoritmo tendrá una escalabilidad y aceleración aceptables gracias a ésto. Se profundiza más en las siguientes secciones.

5.3. Aglomeración y mapeo

Hasta ahora, se ha determinado el trabajo que será realizado por cada procesador y las comunicaciones que serían necesario llevar a cabo para resolver el problema. Sin embargo, no se ha determinado la manera en que la matriz será repartida, ni la forma en que será la comunicación. Es decir, se ha trabajado de forma abstracta.

En esta tercera etapa, aglomeración, se cambia propiamente del diseño a la implementaciόm, el número de tareas que se observa en la primera parte del diseño es aproximadamente la misma en cada uno de los procesadores, y mejor aún, tomarán aproximadamente el mismo tiempo por lo que en realidad no es necesario hacer ninguna aglomeración de ningún tipo, que es para lo que nos sirve esta etapa, por ahora aún se continúa en un abstracto ya que no se sabe exactamente de qué manera se van a mapear estas tareas a cada procesador y de que manera, esta última parte, llamada *mapeo*, está muy cerca ya a la implementación del programa en sí, ya que en esta última parte definitivamente se estaría más cerca de saber de qué manera está construida la computadora paralela, en nuestro caso un cluster Beowulf.

5.4. Mapeo e implementación

Ahora, es importante determinar partes concurrentes para hacer la descomposición de dominio, es decir, las partes que pueden ser calculadas al mismo tiempo. Lo primero a considerar es usar la función que calcula el exponente de Liapunov y tratar de paralelizar esta tarea, pero resulta que al calcular el exponente e ir incrementando el valor de la variable `sum` en cada ciclo, por lo que un ciclo depende del anterior y así sucesivamente, por lo que no es eficiente intentar paralelizar de esa forma.

Una manera natural entonces sería intentar partir la matriz de exponentes de Liapunov y repartirla entre los nodos disponibles, es decir aplicar la

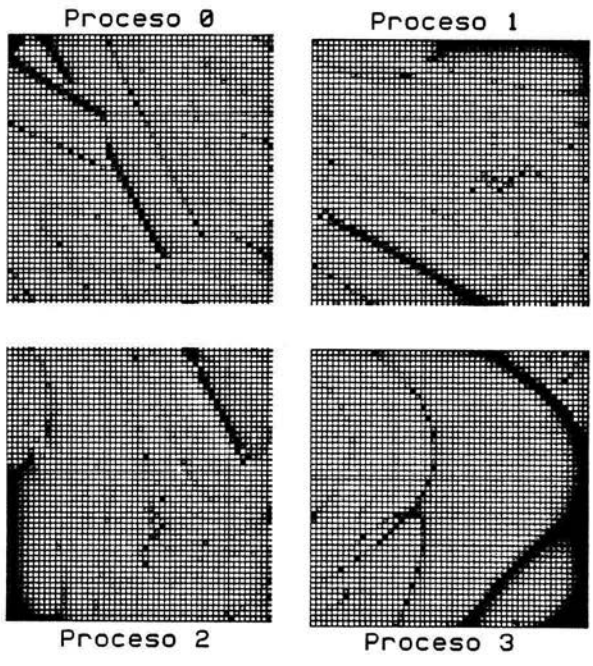

Figura 5.1: Dividiendo entre 4 procesos en paralelo

descomposición de dominio, y que cada nodo calcule la matriz de exponentes para la matriz de valores AB que le corresponda. En este caso la matriz es el dominio que está siendo repartido, ésto se puede ver en la figura 5.1. Este proceso en principio es fácil, pero tenemos un par de consideraciones a tomar: la primera es que de la manera en la que se repartiría sería únicamente un número de procesos de tamaño 2^n ya que sólo se puede partir la matriz de esa manera en múltiplos de 2^n, como en la figura 5.2.

Se tiene entonces que diseñar una estructura de datos de tal manera que se pueda ir guardando en ella los datos para después ser escritos o impresos, ésto parece ser una forma complicada de resolver el problema.

Con ésto, ahora se tienen dos problemas, primero se tiene que dividir la matriz de tal manera que cada proceso tenga aproximadamente el mismo número de puntos y se debe permitir que la matriz pueda ser dividida en

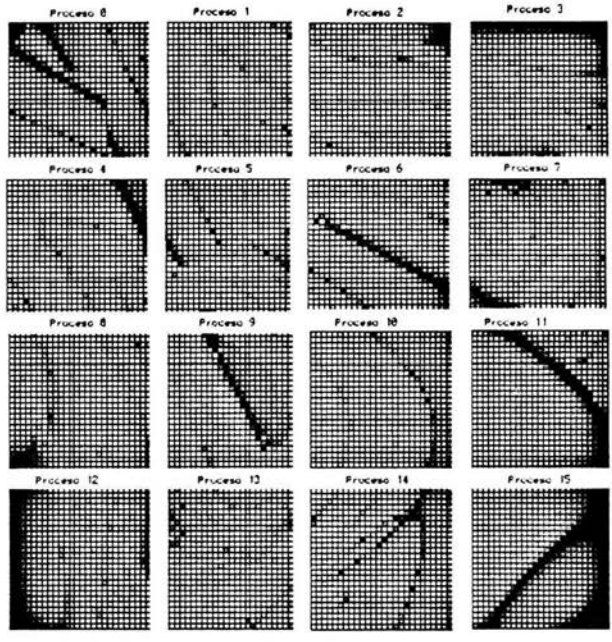

Figura 5.2: Dividiendo el problema en 2^n procesos

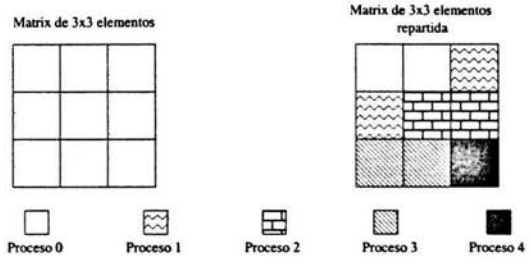

Figura 5.3: Repartiendo una matriz de 3×3 entre 5 procesadores

cualquier número n de procesos. El primer requerimiento es necesario para que cada proceso haga aproximadamente el mismo número de operaciones y por lo tanto terminen aproximadamente al mismo tiempo, esto da un escalamiento casi lineal si se hace en paralelo, el segundo requerimiento se hace para que el programa sea tanto portable entre clusters y con ésto añada flexibilidad a éste, así como para que podamos observar el comportamiento del programa con diferentes tamaños y arquitecturas en paralelo. Recordemos que nuestra implementación usará el modelo de envío de mensajes para hacer el paralelismo y eso quiere decir que el programa no únicamente correría en clusters Beowulf, sino prácticamente en casi cualquier tipo de computadora paralela moderna que use el modelo de envío de mensajes.

El primer problema a resolver es saber como repartir la matriz de valores AB entre n procesadores, en donde la parte que les corresponda a cada uno debería la misma cantidad de puntos, o al menos, aproximadamente la misma. Así que en vez de pensar en una repartición como la descrita por la figura 5.2 y aprovechandose la forma en que el lenguaje de programación C guarda los arreglos bidimensionales se repartiría la matriz de manera similar a la figura 5.3.

En esta repartición se aprovecha el hecho de que ninguno de los puntos debe de tener una comunicación entre sus vecinos, por lo que en realidad cada punto en independiente de los demás y se puede ver a la matriz de valores AB como un arreglo unidimensional, el cual es dividido entre el número de procesos y el resultado de esta división es asignado a cada procesador, se le asigna a la variable *elem_num* el número de puntos que cada proceso tendrá, para calcular este valor. Se tienen dos casos y el segundo problema:

1. El número de elementos de la matriz es divisible exactamente entre el

número de procesos.

2. El número de elementos de la matriz no es divisible exactamente entre el número de procesos.

En el primer caso tendríamos que entonces *elem_num* sería la división de el número de elementos entre el número de procesos. En el segundo caso, como no es una división exacta, lo que se hace es sumarle un elemento más a cada parte que le corresponde a cada proceso. Con ésto, el número de elementos de cada proceso sería mayor al número de elementos de la matriz y el último proceso estaría calculando de más estos elementos. Pero como cada proceso tendría el mismo número de elementos, entonces este último proceso en realidad no se estaría atrasando ya que terminaría aproximadamente al mismo tiempo. La variable `elem_num` también nos serviría para crear un buffer en memoria para guardar los valores del exponente de Liapunov a ser calculados:

```
local_points = malloc (elem_num * sizeof (double));
```

Como puede verse, cada proceso tendrá este buffer para guardar los valores que se generarán.

Ahora nuestro siguiente paso es simplemente pasarle a nuestra función *lyapunov_exp* los valores de AB que le corresponden e ir guardándolas en el buffer *local_points* para luego enviarlo a un sólo proceso y que éste lo escriba en disco o lo despliegue en pantalla.

En un primer intento, se puede pensar en generar dos matrices, una con valores de A y otra con los valores de B y luego hacer una operación colectiva de repartición (scatter) para que cada proceso tenga una parte y empiece a calcular desde ahí.

Este primer aproximamiento, tiene el siguiente inconveniente: se tiene que tratar de minimizar la comunicación, recordando la discusión en la sección 4.2.4 sobre como las comunicaciones colectivas son costosas, el costo de repartir un par de matrices a cada procesador sería costoso y estos valores de las matrices pueden ser calculados localmente.

En el siguiente segmento de código se explica como se reparte el cálculo de los exponentes de Liapunov:

```
  process_init = rank * elem_num;
  process_final = process_init + elem_num;
  i = 0;
for (p = process_init; p < process_final; p++) {
```

```
    a = x_init + ((double)(p% (int)WIDTH) * delta_x);
    b = y_init - ( p / (int)WIDTH) * delta_y;
    local_points [i] = lyapunov_exp (a, b, sequen-
ce);
    i++;
}
```

se tiene en cada proceso la información sobre qué proceso es, un entero que va desde 0 a $n-1$, entonces con esa información es posible saber en qué punto se inicia y se termina, es decir, vemos a la matriz como un vector unidimensional. Además de saber el número de pasos que deben hacerse, esta es la razón del uso del ciclo `for`. Así, con esa información, se pueden ir mapeando los valores de A y B que son necesarios para ir encontrando el valor del exponente de Liapunov en ese punto (A, B) particular, justamente ésto se hace con:

```
    a = x_init + ((double)(p% (int)WIDTH) * delta_x);
    b = y_init - ( p / (int)WIDTH) * delta_y;
```

Esta solución es particular a este problema, ya que conociendo el número de proceso, siempre podemos encontrar el renglón y la columna en la que se está trabajando.

Por último, si cada proceso ha calculado la parte de la matriz que le corresponde, entonces es necesario ahora juntar todos los pedazos en un sólo proceso para que se encargue del proceso de escritura de los datos a disco. Para ésto necesitamos tener un lugar en donde guardar los datos, si el tamaño de la matriz es un parámetro del programa, se requiere crear un buffer para recibir los datos y que fuera creado al vuelo, ésto lo se hace con el siguiente:

```
  if (rank == 0) {
    contents = malloc((int)HEIGHT * (int)WIDTH *
               sizeof(double));
    points = malloc((int)HEIGHT * si-
zeof(double *));
    for (i = 0; i < (int)HEIGHT; i++)
      points[i] = contents + i * (int)WIDTH;
  }
```

Es importante notar la manera en que este buffer es creado, ya que es un buffer que recibirá datos que están ordenados. Si se creara un buffer de

manera que los apuntadores no fueran puestos de esta manera, se tendrían resultados erróneos, es decir, imágenes que no corresponderían a los que está sucediendo en realidad.

Ya ahora con el buffer listo para recibir los datos, en esta parte del programa se podrían intentar implementar envíos desde cada proceso y recepciones desde el proceso 0 (o proceso raíz) de MPI con las funciones `MPI_Send` y `MPI_Recv`, recordando de la sección 4.2.4, se sabe que existen las comunicaciones colectivas en MPI, que permiten hacer operaciones justo como la que es necesario en esta parte del programa para juntar los datos en un sólo procesador. Además de la ventaja de minimizar el número de líneas que se tienen que escribir, la implementación LAM de MPI, hace uso de algoritmos particulares para cuando se tienen nodos que utilizan procesadores múltiples (con memoria compartida) haciendo mucha más rápida la comunicación:

```
MPI_Gather (lo-
cal_points, sndcnt, MPI_DOUBLE, *points,
            rcvcnt, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Estos algoritmos son llamados algoritmos al vuelo, es decir, se pueden pasar como argumentos a la hora de correr el programa, por lo que no interviene en el programa en sí, sino es llamado desde la implementación de MPI que se esté usando. Como se puede ver, al llamar a la función MPI_Gather se utiliza el buffer local para enviar y el local al proceso 0.

Después de ésto, únicamente tenemos que destruir los buffers locales:

```
free (local_points);
```

Y que el proceso 0 escriba o imprima y finalizar MPI:

```
  if (rank == 0) {
    for (k = 0; k < HEIGHT ; k++) {
      for (j = 0; j < WIDTH; j++) {
        printf ("%f, ", points [k] [j]);
      }
      printf ("\n");
    }
    free (contents);
    free (points);
  }
     MPI_Finalize ();
}
```

Nótese que hay dos partes claves para la paralelización del programa, la primera es la repartición de la matriz de valores (A, B) sin necesidad de que el proceso 0 hiciera esa tarea y sin ninguna operación de comunicación, la segunda parte clave es el uso de una sola operación colectiva que podría estar optimizada para el tipo de computadora paralela particular.

La única operación que es completamente serial es la operación de escritura, ya que hasta el momento, al menos para el estándar MPI 1.2, no se pueden hacer operaciones de entrada y salida con varios procesadores[2]

El balance de carga es prácticamente ideal, ya que cada proceso hace una misma carga de trabajo, a excepción de cuando tenemos que el número de elementos de la matriz no es divisible entre el número de procesos. Pero en este caso, solamente aumentamos en un punto más y para fines prácticos un punto más en todos los procesos puede ser despreciable, en este problema en particular entonces, no hay variaciones grandes entre el tiempo de proceso que uno puede requerir y las operaciones que son realizadas por todos los procesos son iguales.

Al usar la generación de los valores en (A, B), dependiendo del procesador en el que se estuviera ejecutando el código, disminuimos en una comunicación colectiva el programa, con lo cual se redujo sustancialmente el tiempo de ejecución. El uso de la operación colectiva `MPI_Gather` nos ayuda tanto en la reducción de las funciones de MPI que se escribirían como potencialmente en computadoras paralelas que usen algoritmos optimizados para cierto tipo de arquitecturas.

En el caso de este programa, sería complicado intentar tener comunicaciones y cálculos al mismo tiempo, por lo que provocaría una sobrecarga en el tiempo. Es muy difícil en la práctica llevar a cabo esta tarea.

Para concluir este capítulo, se hace un pequeño análisis de escalabilidad del problema que se acaba de implementar en paralelo. Éste no es un análisis exhaustivo y únicamente da una idea de la forma en la que se comporta este algoritmo ya que la comunicación es en realidad escasa a menos que el tamaño de la matriz crezca de manera exponencial.

5.5. Análisis de escalabilidad

El análisis de escalabilidad es la estimación de los requerimientos de computación y comunicación de un problema en particular, y el estudio de como

[2]MPI 2 resuelve este tipo de operaciones implementando las funciones para hacer E/S en parelelo.

estos requerimientos cambian cuando el tamaño del problema y el número de procesos cambia.

Tenemos una matriz de tamaño $n \times n$, con ésto en mente se puede estimar la cantidad de cómputo requerido, supongamos entonces que se tiene que para la matriz de $n \times n$ es necesario n^2 llamadas a la función *lyapunov_exp*, ya que por cada punto que se tiene se hace una llamada, por simplicidad supóngase que t_p es el tiempo necesario para calcular el exponente de Liapunov en cada punto, y como se ha visto, el tiempo de cómputo debería ser debido al cálculo de cada punto. Entonces tenemos, que si p es el número de procesadores. El tiempo t_{calc} para calcular los valores de los puntos de la matriz está dado por:

$$t_{calc} = \frac{n^2}{p} t_p$$

El tiempo de comunicación, es definido de la siguiente manera [Gropp et al., 1999]:

$$t_{comm} = t_s + t_w$$

donde t_s es conocido como la latencia o tiempo que tarda un mensaje de tamaño 0 en ser empaquetado y enviado y t_w es el tiempo que le toma al sistema de comunicaciones enviar un mensaje de tamaño w, mejor conocido como ancho de banda.

Y el tiempo que tomaría el envío de los mensajes estaría dado por:

$$t_{comm} = t_s + \frac{n^2}{p} l t_w$$

donde l es el tamaño del mensaje. En este caso, cada punto en un double, es decir, l es el tamaño de un double en C.

Entonces queremos ver de qué manera se comportaría:

$$\lim_{\substack{n \to \infty \\ p \to \infty}} \frac{t_{comm}}{t_{calc}}$$

Si se pudiera hacer que este valor se haga cero para valores grandes de n (tamaño del problema) y valores diferentes de p, entonces tendríamos que el tiempo de cómputo es mucho mayor al tiempo que nos está tomando las comunicaciones, por lo que el algoritmo sería escalable.

Sustituyendo las expresiones anteriores en la expresión anterior se obtiene:

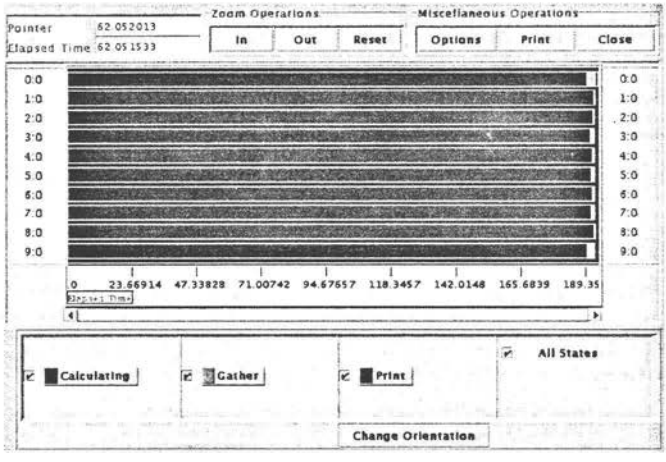

Figura 5.4: Las comunicaciones comparadas con el tiempo de cálculo

$$\lim_{\substack{n \to \infty \\ p \to \infty}} \frac{pt_s}{n^2 t_p} + \frac{lt_w}{t_p} \tag{5.1}$$

La primera parte tiende a 0 ya que n crece más rápidamente que p, pero la segunda parte es una constante que depende de los tiempos que tome calcular un punto y el tiempo que tome enviar una palabra de tamaño l. Si n es muy grande, entonces el tiempo que tome enviar mensajes más grandes será grande, pero mucho mayor será el tiempo que tomará calcular todos esos puntos, por lo que ese valor es cercano a cero, ahora, en el caso práctico tenemos que actualmente el tiempo que toma calcular todos los puntos de la matriz es mayor que el tiempo que toma enviar los datos generados y las comunicaciones que están incrementando sus anchos de banda hace que sea más rápido hacer los envíos de los datos (véase figura 5.4).

Algo que es importante recalcar, es que se requieren generar figuras de 5000×5000 por lo que n no crece tan rápido y el número máximo de procesadores que se tiene es 32 por lo que p tiene un máximo limitado.

	masterlab	clup	ultra-f
procesador	14 G3 Power PC a 350 Mhz	16 Pentium III duales a 600 Mhz	5 AMD Athlon(tm) MP 2600+ duales a 2.1 Ghz
cache	1 kb L1 y 512 kb L2	1 kb L1 y 256 L2 p/procesador	1 kb L1 y 256 L2 p/procesador
memoria	128 MB	640 MB	1 GB
discos duros	6 GB de espacio por nodo	8 GB de espacio por nodo	60 GB de espacio en el nodo principal
tarjetas de red	Ethernet a 100 Mbps gmac integrada	3Com 3c905B 100BaseTX 100 Mbps	3Com 3c905C-TX/TX-M [Tornado] 1Gbps
switch	3com SuperStack II 10/100 Mbps	3com SuperStack II 10/100 Mbps	Switch 1 Gbps

Cuadro 5.1: Características de hardware de los clusters utilizados

5.6. Clusters disponibles

La mejor manera de ejemplificar un cluster Beowulf es por medio del hardware que tengo disponible para este trabajo, el cual es muy similar a la definición de Beowulf pero también muy ilustrativo de lo que se usa en una aplicación de la *vida real* , a continuación se especifican las características tanto de hardware como de software usado en cada uno de los clusters Beowulf a los cuales tuve acceso al desarrollar este trabajo, como una manera de mostrar los recursos de manera gráfica, éstos se muestran en los cuadros 5.1 y 5.2.

5.7. Aceleración

La definición de aceleración o *speedup* es la siguiente:

$$S(n,p) = \frac{T(n)}{T(n,p)}$$

	masterlab	clup	ultra-f
sistema operativo	Linux 2.2.19-1i	Linux 2.4.7-10smp	Linux 2.4.18-18.7.xsmp
sistema de archivos	NFS desde un nodo externo	Utiliza sistema de archivos local en cada nodo[3]	NFS desde el nodo principal
configuración de red	Red interna 192.168.20.0	Red interna 192.168.1.0	Red interna 172.16.100.0
autenticación	Se usa rsh sin passwd	Se usa rsh sin passwd	Se usa rsh sin passwd
bibliotecas de envío de mensajes	LAM/MPI 7.0	LAM/MPI 6.5.6	LAM/MPI 7.0
sistema de procesamiento por lotes	Ninguno presente	Ninguno presente	OpenPBS

Cuadro 5.2: Características de software de los clusters utilizados

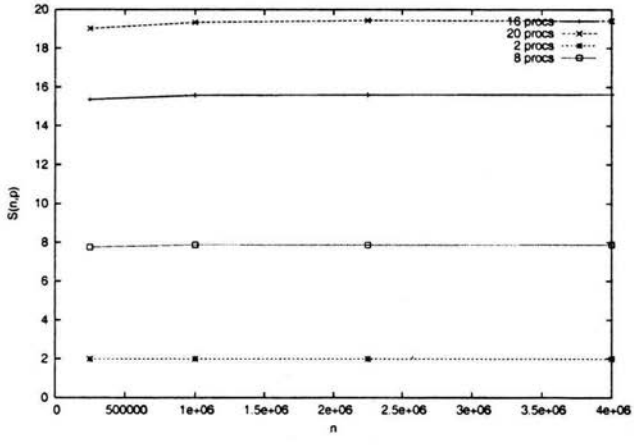

Figura 5.5: Aceleración en el cluster ultra-f para diferentes tamaños del problema

Donde $T(n)$ es el tiempo de ejecución para el problema de tamaño n en un sólo procesador, generalmente tomado del mejor programa serial que resuelva el problema y $T(n,p)$ es el tiempo de ejecución de un problema de tamaño n en p procesadores. Esta cantidad es útil para explorar la escalabilidad de un programa paralelo, en particular la cantidad $S(n,p)$ es el factor por el cual la ejecución es reducida en p procesadores.

Obsérvese en la figura 5.5 la gráfica de la aceleración para diferentes procesadores, se puede ver que son prácticamente constantes a excepción de la aceleración para 20 procesadores; esto se debe a que si el problema es pequeño, el tiempo usado en comunicaciones puede llegar a afectar la aceleración del programa, por lo que se puede encontrar un valor n para el cual el problema continúe con la aceleración que ha mostrado, por otro lado, también se puede observar, que a mayor tamaño del problema, el factor va dejando de ser constante por las comunicaciones, al hacer un `MPI_Gather` es, para fines prácticos, hacer p operaciones de envío y p operaciones de recepción con lo cual, al hacerse crecer el número de procesadores, esta operación causa mayor carga, pero se puede ver que es relativamente pequeña, ya que para el tamaño de problemas que estamos usando la sobrecarga de ésta operación colectiva aún es relativamente pequeña.

Ahora, en las figuras 5.6, 5.7 y 5.8, podemos ver la aceleración para cada cluster con un tamaño de problema de 2000×2000 el cual es un tamaño más grande de cualquier figura generada por Markus.

Obsérvese de estas figuras que todas ellas tienen un comportamiento similar, por lo que se podría pensar que el algoritmo se comporta muy parecido en todas ellas. Pero tenemos que observar además que en particular para la figura 5.6 la gráfica muestra que para un número de procesadores grande, las comunicaciones (para 15 procesadores en este cluster) comienzan a desacelerar el algoritmo pero de manera lenta, en este cluster se pudo observar ésto, ya que el switch que se usó con éste es más lento que en los otros dos.

Por otro lado, podemos ver que la comunicación en realidad no es una carga que debería preocupar, al menos para el número de nodos que se tiene en cada cluster, ya que las gráficas de cada una de ellas, están muy cerca de las gráficas ideales, es decir, el algoritmo escala, al menos de manera práctica, linealmente.

Por último al comparar todas las gráficas de la aceleración juntas (véase figura 5.9), se puede observar algo interesante, la aceleración del cluster clup es algo más cercano a la aceleración ideal que el cluster ultra-f el cual tiene un sistema de comunicaciones con mayor ancho de banda. La explicación que doy para ésto, es que el cluster clup hace los cálculos del exponente de Liapunov más lento que el cluster ultra-f, por lo que la tasa dada por la

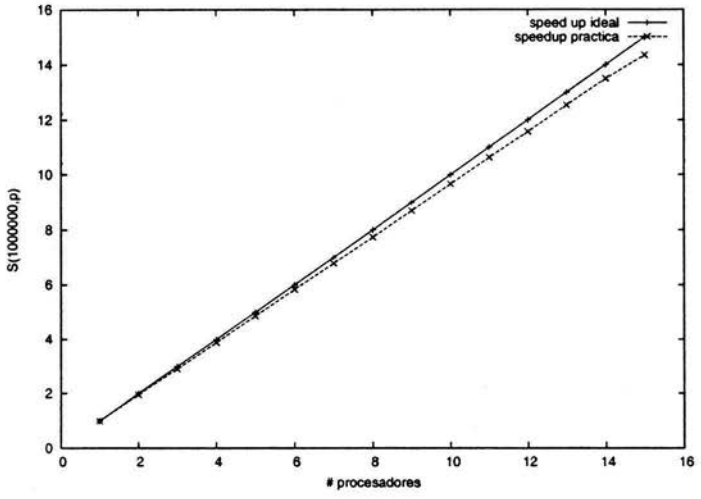

Figura 5.6: Aceleración en masterlab

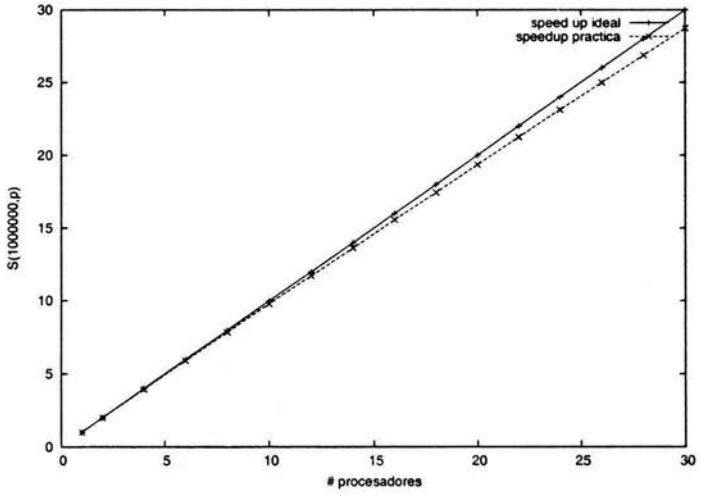

Figura 5.7: Aceleración en clup

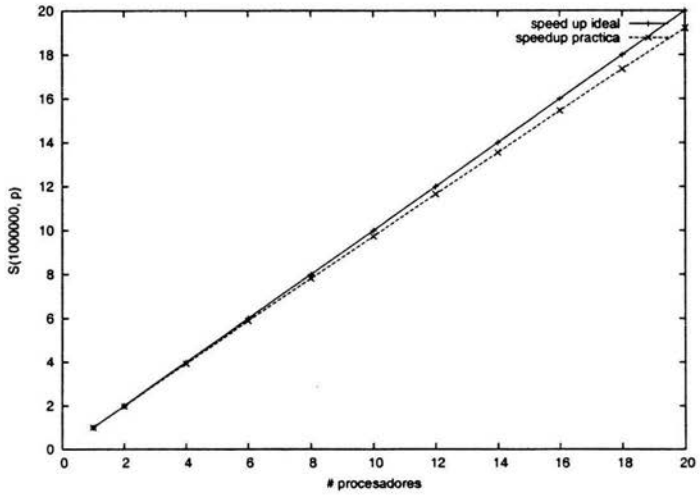

Figura 5.8: Aceleración en ultra-f

ecuación 5.1 tiende a ser mayor para el cluster ultra-f que para el cluster clup, ésto porque al moverse de uno a otro, la velocidad comparada en comunicaciones y velocidad de procesador no crecieron a la misma tasa.

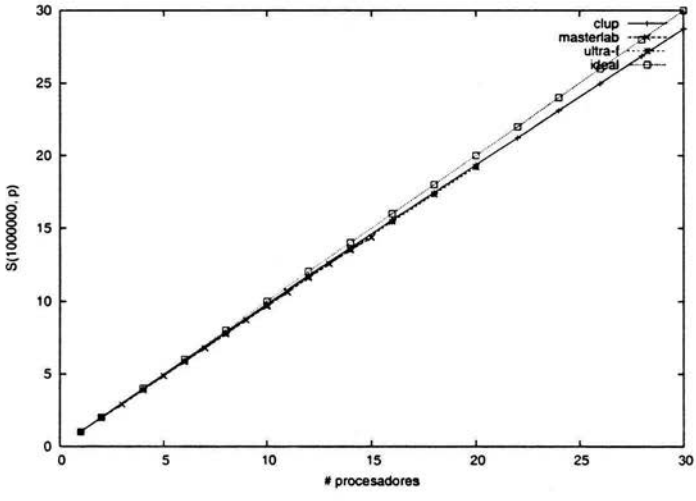

Figura 5.9: La aceleración comparada de los tres clusters usados

Conclusiones

En este trabajo se ha visto una pequeña introducción al estudio de los mapeos unidimensionales deterministas, además de la manera en que se han ido desarrollando distintos sistemas paralelos tanto en hardware como en software, también una introducción básica al envío de mensajes.

Uno de los logros de este trabajo es el hecho de haber implementado el algoritmo en un programa para calcular los diagramas de Liapunov-Markus de manera paralela, con lo cual se reducen los tiempos del cálculo de éstos diagramas logrando con ésto poder estudiar de manera más eficiente regiones o ventanas de interés, por otro lado, la forma en que se presenta el algoritmo en general puede ayudar a implementar en paralelo otros algoritmos que tengan una forma similar de calcular este tipo de matrices, es decir, no es necesario que el cálculo más largo sea el exponente de Liapunov, posiblemente podría ser algún tipo diferente de valor en el cual hubiera interés dado por cada elemento de una matriz. Este trabajo también ha servido para comprobar algunas figuras que ya han sido publicadas en diversos trabajos de Markus [Markus y Hess, 1989a, Markus, 1995, Markus y Hess, 1989b, Markus et al., 1998], finalmente, el programa generado por este trabajo puede servir como una prueba de rendimiento de un problema real para máquinas paralelas, recuérdese que ésto gracias a que MPI es una interfaz que es estándar y actualmente cualquier máquina paralela cuenta con una implementación básica de MPI.

Para concluir me gustaría señalar algunas ideas que sería recomendable se llevaran a cabo como una forma de continuar la línea que ha trazado este trabajo.

Mejoras al programa, entre ellas están, una interfaz gráfica de usuario[4] en la cual el usuario pueda por medio del ratón seleccionar áreas de interés y que éstas fueras calculadas y mostradas automáticamente, una manera rápida podría ser extender el Lyapunovisualizer con las bibliotecas de MPI

[4]Mejor conocida en la jerga computacional como una GUI.

o intentar con alguna API libre como podría ser GTK+, otra mejora importante sería el uso de un analizador léxico. De tal manera que uno pudiera (muy al estilo de programas como Mathematica, Matlab o Maple) introducir fórmulas analíticas y se pudieran calcular sus derivadas para así luego calcular sus exponentes de Liapunov, eliminando así, la necesidad de una recompilación cada vez que se necesite analizar una fórmula nueva, una última mejora importante es el usar comentarios en los archivos generados y que puedan pasarse al visualizador para que al generar una imagen, ésta lleve los valores de los parámetros que fueron usados, para así tener una mejor organización de las figuras.

Sería interesante observar que sucede si, elevamos ahora el valor de la dimensión de la ventana en cuestión. En vez de tener una ventana de valores AB ahora podríamos tener una ventana en tres dimensiones ABC y usar una secuencia ABC elegida por nosotros y generando figuras en tres dimensiones, este trabajo no se ha hecho todavía.

El programa se encuentra listo para su uso, por lo que invito al lector de esta tesis a probar áreas y fórmulas nuevas que pudieran generar figuras interesantes.

Hasta el momento las bibliotecas de MPI están únicamente implementadas para lenguajes como C, C++, y Fortran 77. Un esfuerzo en este camino hacia la implementación de lenguajes de más alto nivel como C# o Java sería importante, ya que ésto ahorraría tiempos de desarrollo en aplicaciones grandes en donde se necesite de cómputo en paralelo.

Galería de imágenes generadas

En este apéndice muestro algunas de las imágenes generadas con el programa. Todas las figuras fueron generadas por una matriz de 1000×1000 por lo que la resolución de éstas es la misma, la función usada es una del tipo II que es descrita por la ecuación 2.2, el transiente usado es 600 y el número de iteraciones hechas después son 10000 y el punto inicial $x_0 = 0.5$, al pie de cada figura, se especifican las características complementarias.

Es interesante recalcar el parecido de estas figuras con algunas figuras pintadas en el siglo pasado por M. C. Escher[5], donde la presencia de la función seno hace que los patrones se repitan por todo el plano y teselen de la misma manera que las pinturas de este artista.

[5]Maurits Cornelis Escher (1898-1972), artista gráfico famoso por sus llamadas "estructuras imposibles".

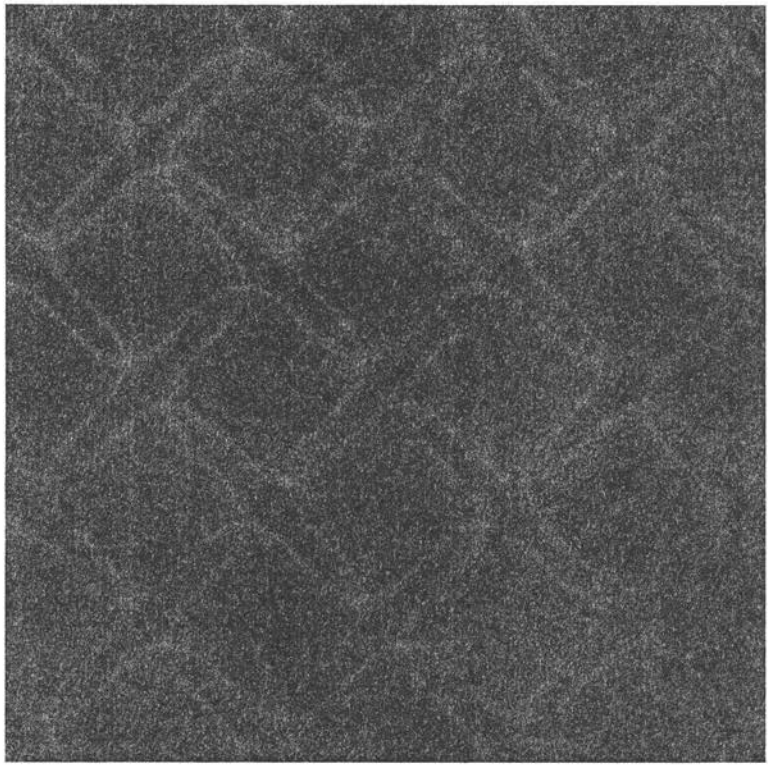

Figura 5.10: Secuencia AB, AI=(0, 10), DD=(10, 0), B=1.0

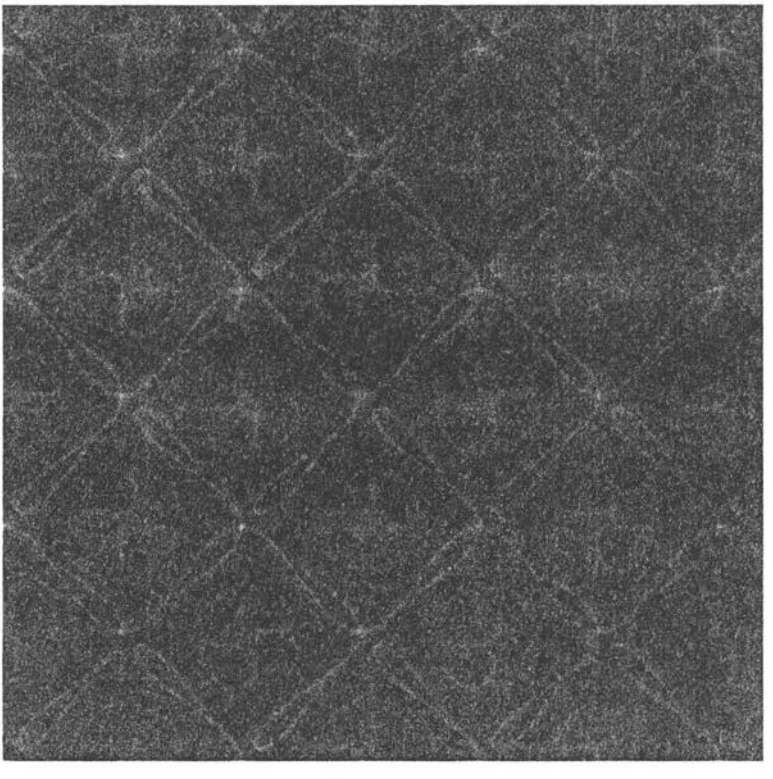

Figura 5.11: Secuencia AB, AI=$(0, 10)$, DD=$(10, 0)$, B=1.4

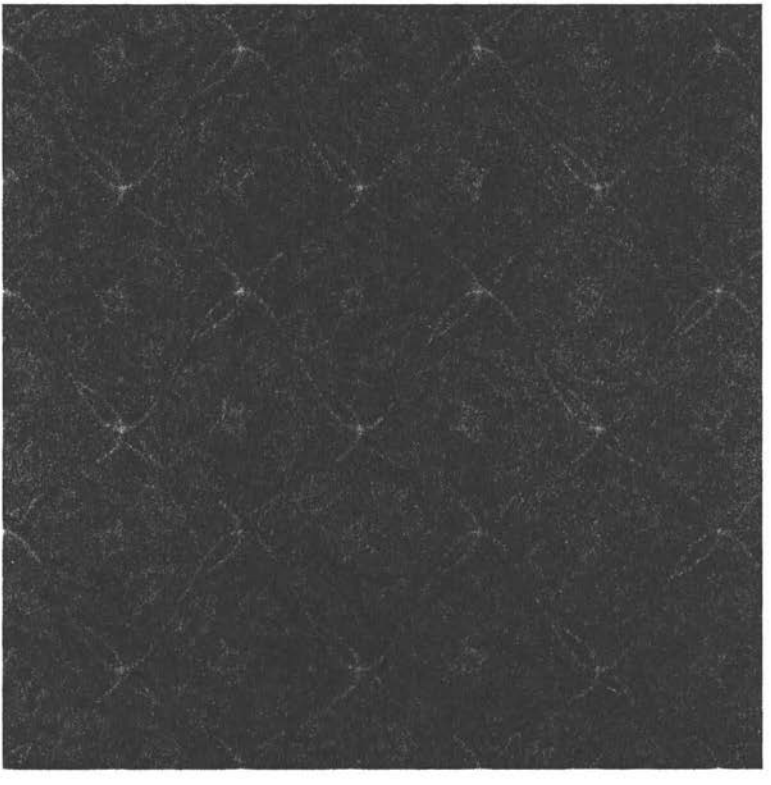

Figura 5.12: Secuencia AB, AI=$(0, 10)$, DD=$(10, 0)$, B=1.8

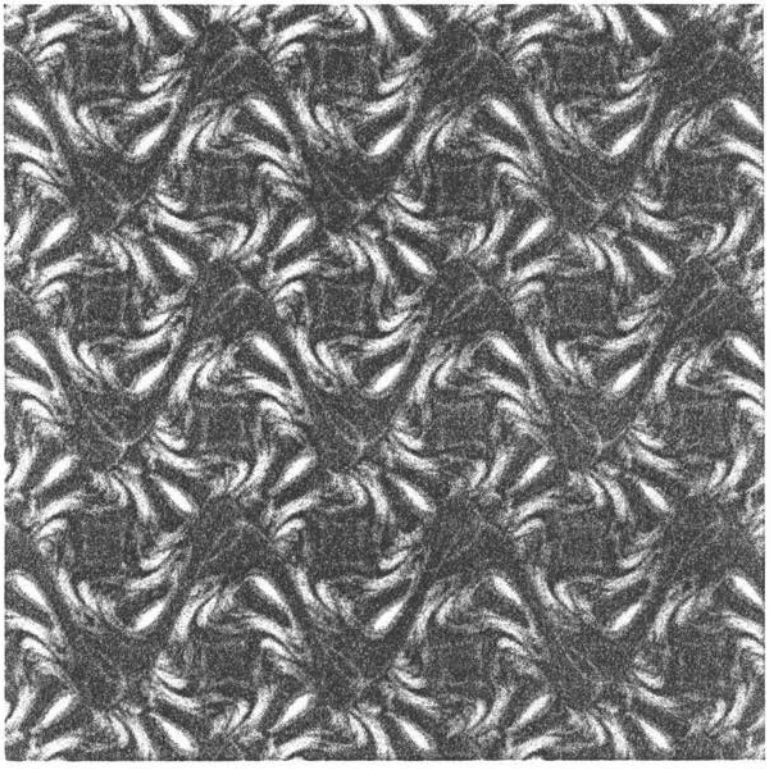

Figura 5.13: Secuencia AB, AI=(0, 10), DD=(10, 0), B=2.1

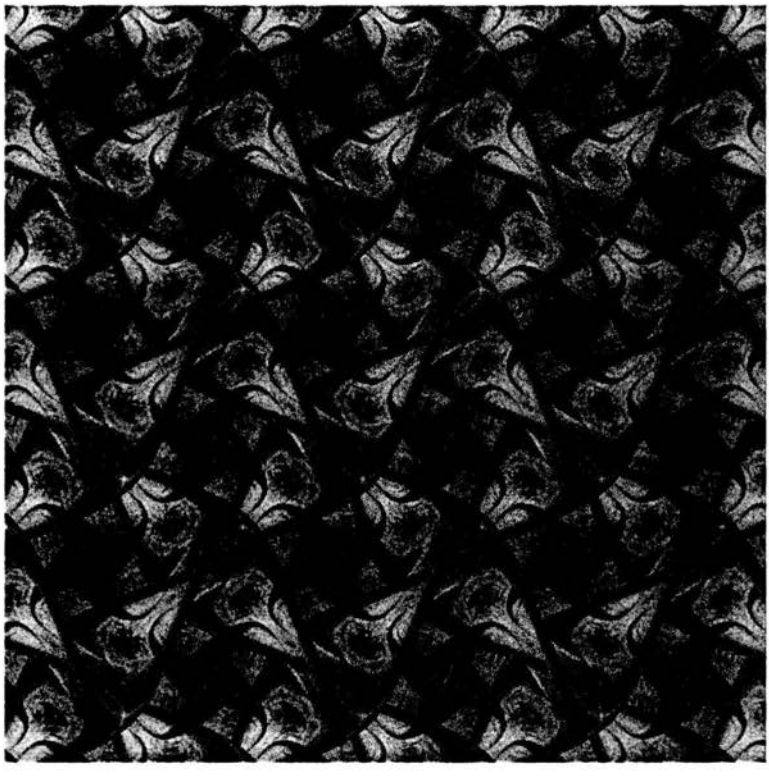

Figura 5.14: Secuencia AB, AI=$(0, 10)$, DD=$(10, 0)$, B=2.5

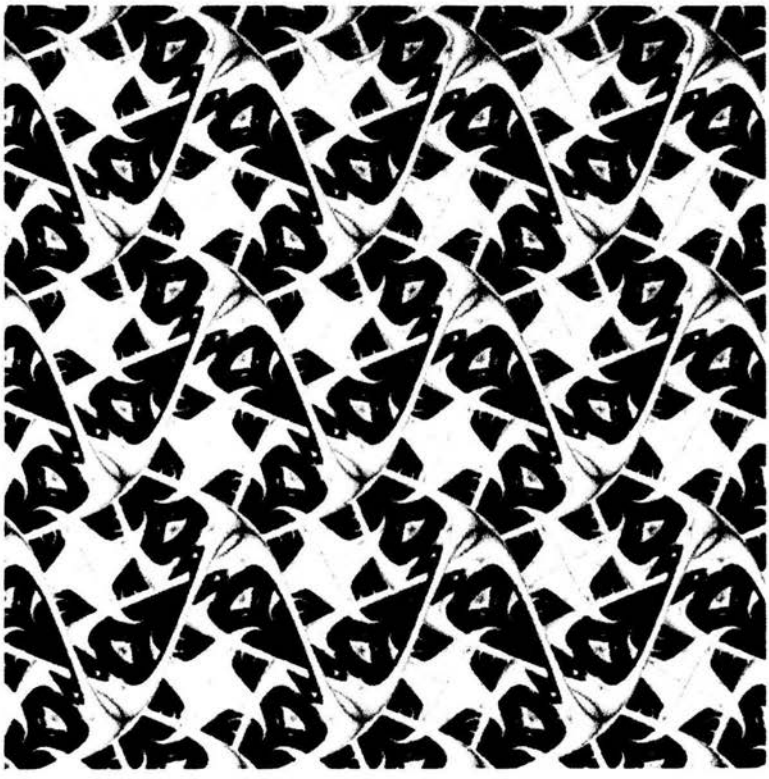

Figura 5.15: Secuencia AB, AI=(0, 10), DD=(10, 0), B=2.5

Figura 5.16: Secuencia AB, AI=(0.5, 2), DD=(2, 0.5), B=2.5

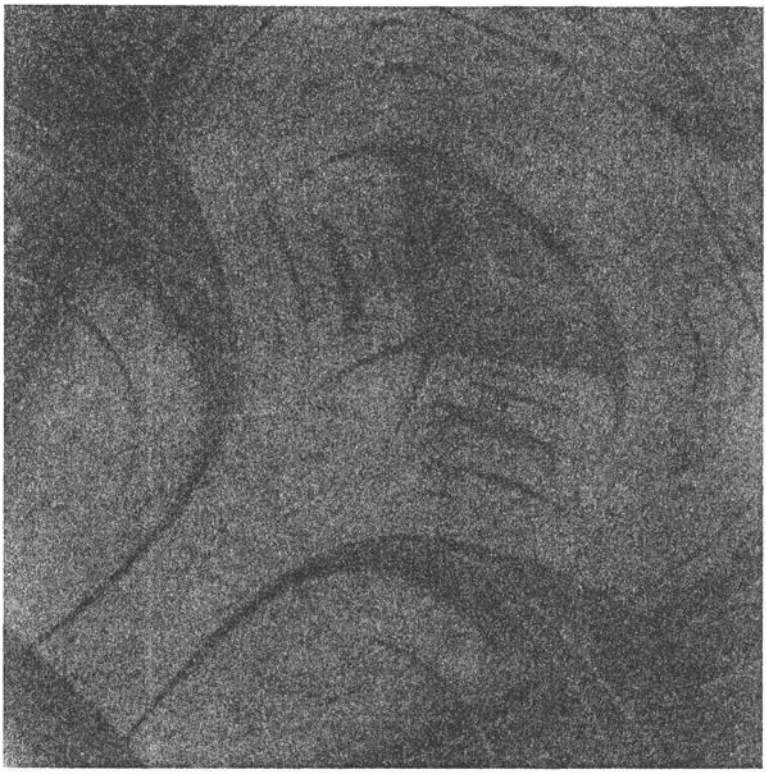

Figura 5.17: Secuencia AB, AI=(0, 1), DD=(1, 0), B=2.5

Figura 5.18: Secuencia AB, AI=(0, 1), DD=(1, 0), B=1.95

Bibliografía

[Devaney, 1987] Devaney, R. L. (1987). *An Introduction to Chaotic Dynamical Systems.* Addison-Wesley, Reading, Massachusetts, primera edición.

[Dijkstra, 1968] Dijkstra, E. W. (1968). Cooperating sequential processes. *Programming languages*, páginas 43–112.

[Foster, 1995] Foster, I. (1995). *Designing and building parallel programs.* Addison-Wesley, primera edición.

[GDB y RBD, 1996] GDB y RBD (1996). *MPI Primer / Developing with LAM.* Ohio Supercomputer Center, Columbus, OH, EUA. Disponible a través de la página http://www.osc.edu/lam.html.

[Gropp et al., 1999] Gropp, W., Lusk, E., y Skjellum, A. (1999). *Using MPI, Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, Massachusetts, EUA, segunda edición.

[Lorenz, 1996] Lorenz, E. N. (1996). *The essence of chaos.* University of Washington Press, primera edición.

[Mantegna y Stanley, 1999] Mantegna, R. M. y Stanley, H. E. (1999). *An Introduction to Econophysics: Correlations and Complexity in Finance.* Cambridge University Press, primera edición.

[Markus, 1995] Markus, M. (1995). Los diagramas de Lyapunov. *Investigación y Ciencia*, (228):70–77. Edición en español de Scientific American.

[Markus y Hess, 1989a] Markus, M. y Hess, B. (1989a). Fractals through periodic variation of control parameters of iterated maps on the interval. En Juergens, H. y Saupe, D., editores, *Visualisierung in Mathematik und Naturwissenschaften*, páginas 87–101. Springer-Verlag, Berlin, Heidelberg.

[Markus y Hess, 1989b] Markus, M. y Hess, B. (1989b). Lyapunov exponents of the logistic map with periodic forcing. *Computers and Graphics*, 13(4):553–558.

[Markus et al., 1998] Markus, M., Kötter, K., y Woltering, M. (1998). Stability graphs of recursive processes. En Barrallo, J., editor, *Mathematics and Design 98, 2nd International Conference*, páginas 495–505, Bilbao, España. Depto. de Matemáticas Aplicadas, Escuela de Arquitectura, Universidad del País Vasco, ELKAR.

[Moore, 1965] Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

[Murray y VanRyper, 1996] Murray, J. D. y VanRyper, W. (1996). *Encyclopedia of Graphics File Formats*. O'Reilly and Associates, Sebastopol, CA, EUA, segunda edición.

[Oualline, 1997] Oualline, S. (1997). *Practical C programming*. O'Reilly and Associates, Sebastopol, CA, EUA, tercera edición.

[Pacheco, 1997] Pacheco, P. S. (1997). *Parallel Programming with MPI*. Morgan Kauffman, San Francisco, California, EUA.

[Radajewski y Eadline, 1999] Radajewski, J. y Eadline, D. (1999). *Beowulf Installation and Administration HOWTO*. Disponible a través del sitio web http://www.beowulf.org.

[Spector, 2000] Spector, D. H. (2000). *Building Linux Clusters*. O'Reilly and Associates, Sebastopol, CA, EUA, primera edición.

[Sterling et al., 1998] Sterling, T. L., Salmon, J., Becker, D. J., y Savarese, D. F. (1998). *How to Build a Beowulf, A guide to the Implementation and Application o PC Clusters*. MIT Press, Cambridge, Massachusetts, EUA. Segunda Impresión.