UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

ALGORITMO PARALELO DE ENTRENAMIENTO DE REDES NEURONALES ARTIFICIALES PARA SISTEMAS DE CONTROL

T E S I S

PARA OBTENER EL TITULO DE

INGENIERO EN COMPUTACION

P R E S E N T A

JOSE LUIS GORDILLO RUIZ

DIRECTOR: DR. JAVIER VITELA ESCAMILLA

DICIEMBRE DE 1998

A papá y mamá, por enseñarme lo más importante en la vida

Quiero agradecer a toda mi familia por el gran amor, apoyo y comprensión que siempre me han brindado y sin los cuales no habría conseguido nada; a mis amigos y amigas, por su ayuda y compañia en todo momento; al Dr. Javier Vitela, por todo su apoyo a lo largo de este trabajo.

También quiero agradecer a la Facultad de Ingeniería por permitirme estudiar una carrera universitaria; al Instituto de Ciencias Nucleares y al Departamento de Supercómputo de la DGSCA, ambos de esta universidad, por todas las facilidades otorgadas; también agradezco el soporte prestado mediante los proyectos CRAY-UNAM SC008897, CRAY-UNAM SC008898 y CONACYT 3155P-A9607.

Índice General

Prefacio

El objetivo de este trabajo es el diseño, la implementación y el análisis de rendimiento de un programa de cómputo paralelo para el entrenamiento de una red neuronal artificial. Este programa fue construído utilizando la biblioteca de intercambio de mensajes MPI (Message Passing Interface) y su propósito es el diseño de redes neuronales artificiales para el control de sistemas dinámicos discretos. Mediante el análisis de rendimiento se desea mostrar las ventajas del uso de computadoras paralelas para la solución de este tipo de problemas de cómputo.

El problema de control utilizado como ejemplo en este trabajo consiste en la conducción de un objeto móvil hacia un punto determinado. La red neuronal artificial usada es un perceptrón multinivel y el algoritmo de entrenamiento usado está basado en el Algoritmo de Retropropagación en el Tiempo.

A lo largo de este trabajo se hace una revisión de los conceptos básicos tanto de las redes neuronales, con el fin de entender sus características y su funcionamiento, como del cómputo paralelo, a fin de conocer los aspectos que identifican a las distintas máquinas paralelas así como los modelos de programación paralela existentes, enfocándonos en el modelo de intercambio de mensajes. También se hace una breve descripción de la biblioteca de intercambio de mensajes MPI, la cual permite la construcción de programas paralelos portátiles, esto es, que pueden ser compilados y ejecutados en diferentes arquitecturas sin necesidad de modificaciones significativas al código fuente.

El diseño del programa paralelo se realizó enfocándose en tres de los aspectos principales que intervienen en el rendimiento de un programa paralelo: la partición del problema a resolver, la asignación de carga de trabajo a los diferentes procesadores y la determinación del esquema de comunicación.

El trabajo está organizado de la siguiente manera:

En el primer capítulo se describen los conceptos básicos de las redes neuronales artificiales y se discuten brevemente tres tipos de redes: la red de Hopfield, el Perceptrón de Rosenblatt y el perceptrón multinivel, el cual será utilizado en el resto del trabajo. También se hace una descripción del algoritmo de retropropagación en el tiempo, el cual es utilizado en el algoritmo de entrenamiento de la red neuronal

para el control de sistemas dinámicos discretos. El capítulo dos contiene una breve discusión de los conceptos de cómputo paralelo, en donde se describen algunas arquitecturas paralelas y sus características, así como algunos de los principales ambientes de intercambio de mensajes. En el capítulo tres se hace una descripción del ambiente de intercambio de mensajes MPI y se discuten brevemente algunas de las las funciones básicas de esta biblioteca. En el capítulo cuatro se presenta el diseño del algoritmo paralelo de entrenamiento de redes neuronales, así como su implementación usando el ambiente de intercambio de mensajes MPI. Además, se incluye una descripción del problema a resolver y del algoritmo secuencial en el cual se basa el algoritmo paralelo. Finalmente, en el capítulo cinco se discuten los parámetros que se usaron para evaluar el rendimiento del programa paralelo y se presentan los resultados obtenidos en un conjunto de ejecuciones de este programa en la supercomputadora paralela Silicon Graphics – Cray Origin 2000 de la UNAM.

Capítulo 1

INTRODUCCIÓN A LAS REDES NEURONALES

1.1 INTRODUCCIÓN

La mente humana tiene una gran superioridad sobre cualquier sistema de cómputo en la solución de problemas que requieren el procesamiento simultáneo de una gran cantidad de información, tales como reconocimiento de voz, de formas geométricas, etc. Algunas de sus características principales son las siguientes[1]:

- *Tolerancia a fallas.* El desempeño de la mente humana no disminuye significativamente a pesar del hecho de que un gran número de neuronas mueren día a día.
- *Flexibilidad.* Puede adaptarse a diversas situaciones y aprender rápidamente de ellas.
- *Paralelismo.* Cientos de miles de millones de células neuronales trabajan en forma paralela para procesar la información proveniente de diversas fuentes como son los ojos, los oídos, el tacto, etc.
- *Facilidad en reconocimiento de patrones.* Es capaz de reconocer y clasificar la información aún en ambientes en los cuales existen factores que dificultan estas tareas.
- *Tiene un bajo consumo de energía.*

Desde hace más de cincuenta años varios científicos se han dedicado a la investigación de esquemas de cálculo que posean algunas de las características anteriores. El cómputo neuronal, es decir, las Redes Neuronales Artificiales (RNA), se presenta como un paradigma alternativo sumamente atractivo. Este esquema

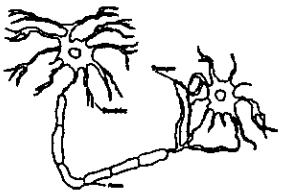

Figura 1.1: Dibujo esquemático de una neurona natural

adquiere sus conceptos básicos del funcionamiento de las células neuronales naturales sin intentar, por supuesto, copiar completamente el funcionamiento de un sistema neuronal biológico. La principal semejanza entre las RNA's y los sistemas neurobiológicos consiste en que ambos están constituídos por un gran número de unidades sencillas que son capaces de resolver en forma colectiva problemas complicados y ambiguos, así como de cambiar de su comportamiento para adaptarse a un problema específico.

1.2 CONCEPTOS BÁSICOS

1.2.1 Neuronas Naturales

La mente humana se compone de alrededor de 10^{11} neuronas (o células nerviosas) de diversos tipos[1]. La Fig. 1.1 muestra un dibujo esquemático de una de estas células. Las neuronas forman redes nerviosas a través de fibras en forma de árbol llamadas *dendritas*, las cuales están conectadas al cuerpo de la célula, conocido como *soma*. Del soma se extiende una fibra larga llamada *axón*, la cual se divide al final en varias ramas. Cada una de estas ramas tiene un botón terminal, llamado *sinápsis*, el cual sirve para establecer la conexión entre dos neuronas. El proceso de recepción en una neurona, el cual establece una conexión con otra neurona, puede llevarse a cabo en una dendrita o directamente en el soma. El axón de una neurona típica puede poseer miles de conexiones con otras neuronas.

La trasmisión de una señal desde una neurona hacia otra se realiza a través de la sinápsis, por medio de un proceso químico sumamente complejo en el cual diversas substancias son liberadas desde la neurona trasmisora. El efecto de esta substancia es el incremento o decremento del potencial eléctrico del soma de la neurona receptora. Si la integración de potenciales supera un valor umbral, la neurona envía un pulso eléctrico de amplitud y duración fijos a través del axón; cuando esto sucede se dice que la neurona ha sido "disparada". Este pulso se trasmite simultáneamente a través de todas las ramas del axón. Es importante destacar que una neurona recibe señales diferentes debido a sus múltiples conexiones con otras neuronas y que algunas señales pueden inhibir a otras; el efecto neto de estas

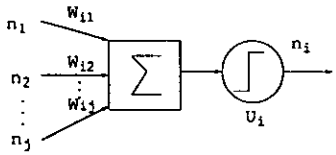

Figura 1.2: Modelo básico de una Neurona Artificial

señales debe sobrepasar un valor umbral para que la neurona sea disparada.

1.2.2 Modelo Matemático Básico de una Neurona Artificial

McCulloch y Pitts[1] propusieron en 1943 un modelo matemático simple del comportamiento de una neurona representándola como una unidad binaria con un valor umbral (ver Fig. 1.2). Esta unidad o nodo calcula la suma ponderada de todas sus entradas, proporcionando como resultado ya sea un -1 ó un 1, dependiendo de si el total de la suma es menor o mayor que el valor umbral de la unidad. La expresión matemática que modela este comportamiento es la siguiente

$$n_i(t+1) = \Theta(\sum_j w_{ij} n_j(t) - u_i), \tag{1.1}$$

en donde:

- n_i es el nivel de activación de la i-ésima unidad.
- w_{ij} representa el peso de la conexión entre la unidad i y la unidad j.
- u_i es el valor umbral de la unidad i.
- t representa los valores discretos de tiempo en los cuales el nivel de la activación de la unidad es actualizada.
- $\Theta(x)$ es la **Función Escalón** definida por:

$$\Theta(x) = \begin{cases} 1, & \text{si } x \geq 0 \\ -1, & \text{si } x < 0 \end{cases} \tag{1.2}$$

McCulloch y Pitts probaron que un ensamble síncrono de tales unidades sería capaz, en principio, de realizar cualquier operación realizada por una computadora digital escogiendo adecuadamente los pesos w_{ij} que caracterizan las conexiones entre las unidades.

1.2.3 Un Poco de Historia

La base de los modelos de Redes Neuronales Artificiales apareció con el trabajo de McCulloch y Pitts en 1943. En los siguientes años se realizaron una cantidad

considerable de trabajos sobre las redes formadas con unidades binarias[1]. Por otro lado, Rashenvsky (1938), Wiener (1948), Beurle (1956), Wilson y Cowan (1973) y Amari (1977) realizaron estudios con unidades continuas, utilizando ecuaciones diferenciales para describir los patrones de actividad de las neuronas, campo que fue denominado "Neurodinámica".

Alrededor de 1960, Rosenblatt[2] se enfocó en el problema de encontrar los pesos w_{ij} apropiados para tareas de cómputo particulares, usando redes llamadas perceptrones. El perceptrón de Rosenblatt es una red con unidades binarias organizadas en dos niveles o capas, y en donde las salidas de las unidades del primer nivel actúan como entradas a las unidades del segundo nivel (redes alimentadas hacia adelante). Rosenblatt desarrolló un algoritmo de aprendizaje para este tipo de redes conocido como *la regla delta*, mediante el cual el perceptrón era capaz de hacer generalizaciones a partir de un conjunto de patrones de entrenamiento; sin embargo, Minsky y Papert mostraron que el perceptrón simple y su algoritmo de aprendizaje solo podían aplicarse a problemas con patrones separables, existiendo problemas elementales (como el de la operación lógica NOR) cuya solución no puede ser calculada correctamente por este tipo de red neuronal. Rosenblatt estudió redes con más capas de unidades con la esperanza de que éstas pudiesen superar las limitaciones del perceptrón simple, pero en ese entonces no era conocido ningún algoritmo capaz de calcular los pesos w_{ij} apropiados para la realización de una tarea determinada.

En los años 70, el enfoque principal de las investigaciones en redes neuronales artificiales pasó a ser el de la memoria asociativa. La memoria asociativa, como su nombre lo indica, asocia diferentes patrones con otros previamente "memorizados" o "almacenados", siempre y cuando exista suficiente similitud entre ellos. Este concepto fue propuesto por Taylor (1956) y redescubierto por Anderson (1968), Willshaw (1969), Marr (1969–71) y Kohonen (1974–89). En 1981, Hopfield introdujo el concepto de *función de energía* a las investigaciones sobre la memoria asociativa, enfatizando la noción de memoria como un conjunto de atractores dinámicamente estables.

Se puede considerar que los trabajos que mayor influencia han ejercido en los últimos años en el campo de las Redes Neuronales Artificiales son aquellos que retomaron el concepto del perceptrón de Rosenblatt, añadiendo capas intermedias entre los niveles de entrada y salida y sustituyendo las unidades binarias por unidades con funciones de activación no lineales monótonamente crecientes. El origen de esta popularidad se debe al desarrollo de un algoritmo capaz de ajustar los pesos de las conexiones entre las unidades. Este algoritmo, conocido como *algoritmo de retropropagación*, fue introducido originalmente por Werbos (1974) y redescubierto y popularizado independientemente por Rumelhart, Hinton y Williams (1986), Parker (1985) y Le Cun (1985). Los perceptrones multinivel se componen de un conjunto de unidades de cálculo sencillas organizadas en capas. Cada conexión de una unidad a otra esta caracterizada por un peso. Las unidades de

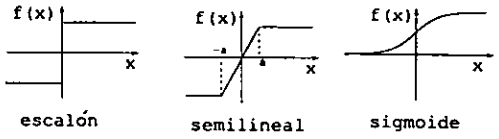

Figura 1.3: Gráficas de las funciones de activación más comunes

un nivel reciben como entrada las salidas de las unidades del nivel anterior, multiplicando cada una de éstas por el peso de su conexión. El resultado de la suma de todas las entradas es trasmitida a través de una función no lineal que caracteriza al nodo. El resultado de esta función representa el nivel de activación (o salida) de la unidad.

1.2.4 Clasificación de las Redes Neuronales Artificiales

Existen tres funciones de activación (ver Fig. 1.3) diferentes usadas comúnmente en las RNA's, estas funciones son:

- *La Función Escalón* definida en la Ec. (1.2)
- *La Función Semilineal* definida como:

$$\mathcal{F}(x) = \begin{cases} -1, & \text{si } x \leq -a \\ x, & \text{si } -a < x \leq a \\ 1, & \text{si } x > a \end{cases} \tag{1.3}$$

- *La Función Sigmoide* la cual se define de la siguiente manera:

$$\mathcal{F}(x) = \frac{1}{1 + \exp^{-x}} \tag{1.4}$$

Los pesos de las conexiones entre las unidades son modificados mediante algoritmos de aprendizaje apropiados, los cuales permitien a una RNA realizar alguna tarea específica.

En general, podemos clasificar a los distintos tipos de redes neuronales de acuerdo a tres características: el tipo de función de activación de sus unidades, la topología de la red (esto es, la forma en que las unidades se conectan entre sí) y el algoritmo de aprendizaje utilizado para la modificación de los pesos de las conexiones. Una clasificación posible es la siguiente, propuesta por Lippmann[2]:

$$
\textit{Redes Neuronales Artificiales} \begin{cases} \textit{Binarias} \begin{cases} \textit{Supervisadas} \begin{cases} \textit{Perceptrón Simple} \\ \textit{Hopfield} \\ \textit{Hamming} \end{cases} \\ \textit{No Supervisadas} \begin{cases} \textit{Carpenter} - \textit{Grossberg} \end{cases} \end{cases} \\ \textit{Continuas} \begin{cases} \textit{Supervisadas} \begin{cases} \textit{Perceptrón Multinivel} \end{cases} \\ \textit{No Supervisadas} \begin{cases} \textit{Kohonen} \end{cases} \end{cases} \end{cases}
$$

A continuación se presentará una breve descripción de la red de Hopfield y del perceptrón de Rosenblatt, para después concentrarnos en el perceptrón multinivel, que es el tipo de red con el cual se desarrollará el resto del trabajo.

1.3 LA RED DE HOPFIELD

La red de Hopfield, como se indica en la clasificación anterior, se compone de unidades binarias como las descritas en la sección 1.2.2, aunque también existen implementaciones con valores continuos. Esta red es usada principalmente como una memoria asociativa[3]. La característica de una memoria asociativa es la siguiente: sea un conjunto de patrones $\{x_i^p,\ i = 1, 2, ..., N;\ p = 1, 2, .., P\}$ donde N y P son el número de nodos y el número de patrones, respectivamente; estos patrones se "almacenan" en la red de tal forma que al presentar a ésta un patrón arbitrario diferente $\{\zeta_i,\ i = 1, 2, .., N\}$ responderá con el patrón almacenado que más se "parece" al patrón ζ_i. En este caso, la medida de la similitud entre dos patrones está dada por su distancia Hamming, la cual está definida como el número de elementos que son diferentes en dos patrones dados. Usaremos un ejemplo clásico de almacenamiento bibliográfico para ilustrar el funcionamiento de una memoria asociativa: supongamos que el elemento "La Tregua, Mario Benedetti, 1996" ha sido almacenado. La memoria asociativa debe ser capaz de recuperar el elemento completo a partir de una parte de éste, como por ejemplo "Tregua, Mari" e incluso hacerlo si un error se presenta en el elemento de entrada, como puede ser en el caso de "Tregia, Maro".

La topología de una red de Hopfield[3] se muestra en la Fig. 1.4. Los N valores binarios x_i constituyen un patrón. La unidad x_i esta conectada a la unidad x_j mediante el peso t_{ij}. La entrada total a cada unidad está determinada por

$$H_i(n+1) = \sum_j t_{ij} x_j(n) + I_i - U_i, \tag{1.5}$$

donde I_i y U_i son los valores de entrada y de umbral de la la i-ésima unidad, respectivamente. La salida del i-ésimo nodo x_i está determinada por la función escalón de la entrada total: $\Theta(H_i)$. Al presentar un patrón dado $\{x_i^e, i = 1, 2, ..N\}$,

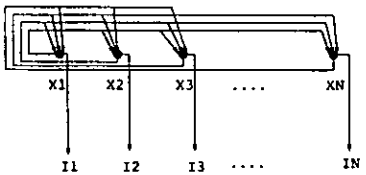

Figura 1.4: Topología de una red de Hopfield

la red itera hasta alcanzar un estado estable que corresponde al patrón almacenado con menor distancia Hamming al patrón presentado. A cada iteración, el nivel de activación de las unidades es actualizado de una manera aleatoria, tomando solo una unidad a la vez.

Hopfield introdujo a este tipo de red el concepto de *función de energía*[3] y consideró a los patrones almacenados como *atractores* en el espacio de estados (ver Fig. 1.5). La función de energía utilizada por Hopfield es la siguiente

$$E = -\frac{1}{2}[\sum_i \sum_j t_{ij} x_i x_j], \tag{1.6}$$

donde se ha supuesto que $I_i, U_i = 0$. De esta función podemos deducir que el incremento ΔE con respecto a un cambio Δx_i está dado por:

$$\Delta E = -[\sum_j t_{ij} x_j] \Delta x_i; \tag{1.7}$$

debido a que x_i puede tomar solo valores de -1 o 1, tenemos que si x_i pasa de 1 a -1, entonces $\Delta x_i < 0$ y dado que este cambio ocurre solo cuando el término entre paréntesis es a su vez negativo, como lo indica la Ec. (1.5), se deduce que $\Delta E < 0$. Un argumento similar puede exponerse cuando x_i pasa de -1 a 1. Por otra parte, cuando x_i no cambia, $\Delta E = 0$. Podemos concluir entonces que el cambio ΔE siempre es menor o igual a 0 y por lo tanto el sistema es estable.

El almacenamiento de un número P de patrones $\{x_i^p, i = 1, 2, ..N; p = 1, 2, ..P\}$ en la red de Hopfield esta representado en los valores de los pesos de las conexiones entre las unidades. La regla de asignación de tales pesos es la siguiente:

$$t_{ij} = \begin{cases} \sum_{p=1}^{P} x_i^p x_j^p & , i \neq j \\ 0 & , i = j; \end{cases} \tag{1.8}$$

en particular, si deseáramos almacenar un solo patrón $p_o = \{x_1^0, x_2^0, ..x_N^0\}$ se tendrá

$$t_{ij} = x_i^0 x_j^0; \tag{1.9}$$

si se inicializa la red con un patrón extraño p_e, la entrada subsecuente a cada unidad será

$$H_i^e = \sum_j (t_{ij} x_j^e) = \sum_j (x_i^0 x_j^0 x_j^e) = x_i^0 \sum_j x_j^0 x_j^e; \tag{1.10}$$

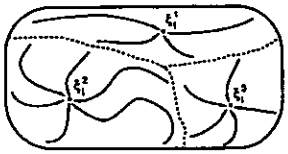

Figura 1.5: Esquema de un espacio de estado con 3 atractores

el término $\sum_j x_j^0 x_j^e$ será positivo si $p_0 \sim p_e$, esto es, si los patrones p_0 y p_e son suficientemente similares; si esta situación se cumple, el valor de H_i^e tendrá el signo de x_i^0 (el de la correspondiente unidad del patrón almacenado); en cambio, si p_0 y p_e son muy diferentes, este término será negativo, lo que llevará a la red a converger hacia un patrón igual a $-p_0 = \{-x_1^0, -x_2^0, ..., -x_N^0\}$. Esto indica que el almacenamiento de un patrón p_0, implica también el almacenamiento del patrón opuesto $-p_0$.

1.3.1 Capacidad de Almacenamiento

En el caso de tener varios patrones almacenados, al inicializar la red con un cierto patrón p_e tenemos:

$$H_i^e = \sum_j t_{ij} x_j^e, \tag{1.11}$$

lo que conduce a

$$H_i^e = \sum_j \sum_{p=1}^{P} x_i^p x_j^p x_j^e; \tag{1.12}$$

desarrollando esta sumatoria tenemos

$$H_i^e = \sum_j x_i^1 x_j^1 x_j^e + \sum_j x_i^2 x_j^2 x_j^e + ... + \sum_j x_i^P x_j^P x_j^e; \tag{1.13}$$

si separamos el término para el cual $p_a \sim p_e$ llegaremos a

$$H_i^e = \sum_j x_i^a x_j^a x_j^e + \sum_j \sum_{p \neq p_a}^{P} x_i^p x_j^p x_j^e \tag{1.14}$$

$$H_i^e \approx N x_i^a + \sum_j \sum_{p \neq p_a}^{P} x_i^p x_j^p x_j^e \tag{1.15}$$

de donde podemos observar que si

$$\frac{1}{N} \left| \sum_{p \neq p_a}^{P} \sum_{j=1}^{N} x_i^p x_j^p x_j^e \right| > 1 \tag{1.16}$$

la red podría tener un comportamiento no esperado; esto se debe a que el término de la izquierda de la Ec. (1.16) tendría la posibilidad de cambiar el signo de H_i^s, independientemente del valor de x_i^a. A fin de evitar en lo posible esta situación, es importante determinar cual es la probabilidad de que ésta se presente. Con este objetivo definimos la siguiente variable aleatoria

$$C_i^{s'} = -x_i^{s'} \frac{1}{N} \sum_{s \neq s'}^{P} \sum_{j}^{N} x_i^s x_j^s x_j^{s'}, \tag{1.17}$$

en donde $C_i^{s'}$ será positivo únicamente si x_i y $\sum_{s \neq s'} \sum_j x_i^s x_j^s x_j^{s'}$ tienen signos contrarios. Nos interesa encontrar cual es la probabilidad de que $C_i^{s'} > 1$.

Observando la Ec. (1.13) podemos pensar que entre mayor sea el número de patrones almacenados, mayor será esta probabilidad. Para demostrar esto, supondremos que x_i^s, x_j^s, $x_i^{s'}$, $x_j^{s'}$ son variables aleatorias independientes con la misma probabilidad de obtener los valores $+1$ y -1; podemos sustituir la sumatoria de la Ec. (1.17) por

$$C_i^{s'} = \frac{1}{N} \sum_{k}^{N*P} y_k, \tag{1.18}$$

donde y_k es una variable aleatoria con la misma probabilidad (1/2) de obtener los valores +1 y -1. A partir de que

$$< y_k >= (.5)(-1) + (.5)(1) = 0, \tag{1.19}$$

tenemos que el valor promedio de $C_i^{s'}$ es

$$< C_i^{s'} >=< \frac{1}{N} \sum_{l}^{N*P} y_k >= \frac{1}{N} \sum_{l}^{N*P} < y_k >= 0. \tag{1.20}$$

La varianza de $C_i^{s'}$ es calculada a partir de

$$\begin{aligned} < (C_i^{s'} - < C_i^{s'} >)^2 > &= < C_i^{s'^2} > \\ &= < (\frac{1}{N} \sum_{l}^{N*P} y_l)^2 > \\ &= \frac{1}{N^2} < \sum_{l}^{N*P} \sum_{m}^{N*P} y_l y_m >; \end{aligned} \tag{1.21}$$

usando el hecho de que las variables aleatorias son independientes, es decir,

$$< x_l x_m >= \begin{cases} 0 & l \neq m \\ < x_l^2 > & l = m \end{cases}; \tag{1.22}$$

tenemos

$$< C_i^{s'^2} >= \frac{1}{N^2} \sum_{l}^{NP} < y_l >= \frac{P}{N}, \tag{1.23}$$

Probabilidad $C_i s' > 1$	Varianza P/N
0.001	0.105
0.0036	0.138
0.01	0.185
0.05	0.37
0.1	0.61

Tabla 1.1: Valores de probabilidad de $C_i^{s'}$ en función de P/N

donde usamos

$$< x_i^2 >= (-1)^2(.5) + (1)^2(.5) = 1. \tag{1.24}$$

Aunque la función de probabilidad de $C_i^{s'}$ tiene un comportamiento multinomial, puede ser aproximada mediante una función de probabilidad gaussiana con media 0 y varianza $\frac{P}{N}$ usando el teorema de límite central. Algunos valores de la probabilidad de que $C_i^{s'} > 1$ para diferentes valores de la varianza P/N se dan en la Tabla 1.1; en esta tabla se puede observar que para obtener una probabilidad menor al 1% de que algún bit del patrón presente un comportamiento no deseado, la relación P/N debe ser menor a 0.185, es decir, $P < 0.185N$.

1.4 EL PERCEPTRÓN DE ROSENBLATT

El perceptrón simple propuesto por Rosenblatt en 1959 se compone de un conjunto de unidades de entrada y una unidad de salida, como se muestra en la Figura 1.6[2],[4]. Las componentes del vector de entrada $\vec{x} = [x_1, ..., x_n]$ son multiplicados por el vector de pesos $\vec{w} = [w_1, w_2, w_3, ..., w_n]$; estos resultados son sumados, produciendo el producto interno

$$s = \vec{x} \cdot \vec{w}, \tag{1.25}$$

al cual se le resta el umbral θ de la unidad. En este perceptrón, la unidad de salida posee una función de activación escalón, de modo que la salida de la red es

$$y = \Theta(s - \theta). \tag{1.26}$$

Durante el proceso de entrenamiento de la red se presentan K patrones y sus correspondientes valores de salida deseados $\{(\vec{x}_1, d_1), (\vec{x}_2, d_2), ..., (\vec{x}_K, d_K)\}$. Un algortimo ajusta los pesos de tal manera que la salida de la red para cada patrón sea igual a su correspondiente valor deseado. El algoritmo propuesto por Rosenblatt es conocido como *la regla delta*; este algoritmo obtuvo una gran popularidad debido a que el perceptrón presentaba salidas correctas para patrones que no habian sido incluídos en el conjunto de entrenamiento.

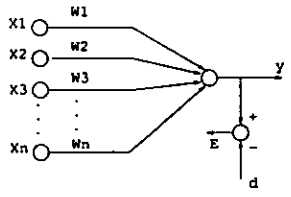

Figura 1.6: El perceptrón simple

1.4.1 La Regla Delta

El algoritmo de la regla delta puede resumirse en los siguientes pasos

- Inicializar los pesos $\{w_i\}$ con valores aleatorios pequeños.
- Presentar un patrón de entrada $\vec{x}_k$, cuya salida d_k sea conocida.
- Calcular la salida de la red

$$y_k = \Theta(\sum_i^n w_i x_{k_i} - \theta). \tag{1.27}$$

- Adaptar los pesos mediante la siguiente ecuación

$$w_i = w_i + \Delta w_i \ , \ \forall \ i; \tag{1.28}$$

en donde

$$\Delta w_i = \eta(d_k - y_k)x_{k_i} = \eta\delta \cdot x_{k_i}, \tag{1.29}$$

$$\delta = (d_k - y_k). \tag{1.30}$$

El término η es conocido como el coeficiente de aprendizaje y es utilizado para acclerar la convergencia del algoritmo. Rosenblatt desarrolló este método de una manera empírica, sin demostar analíticamente su validez; sin embargo, en otras implementaciones del perceptrón, las cuales utilizan funciones de activación derivables, [5] se ha demostrado que la regla delta implementa una técnica de gradiente descendente para encontrar el mínimo de una función de error, la cual es definida por

$$\mathcal{E} = \frac{1}{2}(d_k - y_k)^2, \tag{1.31}$$

$$y^k = f(s^k - \theta); \tag{1.32}$$

en donde y_k es la salida del perceptrón dado un patrón k y f es una función derivable. Para comprobar que la regla delta minimiza la función de error, mostraremos que

$$-\frac{\partial E}{\partial w_i} = \delta \cdot x_{k_i}, \tag{1.33}$$

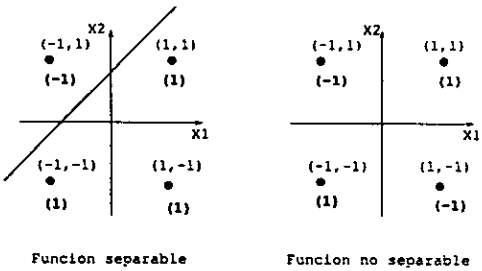

Figura 1.7: Ejemplos de funciones separables y no separables

esto es, Δw_i está dado en la dirección $-\nabla E$. Usamos la regla de la cadena para escribir esta derivada como el producto de otras dos derivadas: la derivada del error con respecto al nivel de salida y la derivada del nivel de salida con respecto a los pesos

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial w_i}; \tag{1.34}$$

derivando la Ec. (1.31) con respecto a la salida de la red y_k tenemos

$$\frac{\partial E}{\partial y_k} = -(d_k - y_k) = -\delta, \tag{1.35}$$

mientras que de las Ec. (1.25) y (1.32) podemos concluir que

$$\frac{\partial y_k}{\partial w_i} = x_{k_i} f', \tag{1.36}$$

y por lo tanto

$$-\frac{\partial E}{\partial w_i} = \delta \cdot x_{k_i} f'. \tag{1.37}$$

El perceptrón de Rosenblatt es capaz de calcular algunas funciones lógicas de sus entradas. En una red con n entradas existen 2^n diferentes patrones de entrada y por lo tanto 2^{2^n} funciones lógicas posibles; como señalaron Minsky y Papert, el perceptrón solamente puede calcular un subconjunto de estas funciones, las cuales son conocidas como *funciones linealmente separables*[2]. Una función linealmente separable es aquella en la que todos los puntos con un mismo "resultado" caen en el mismo lado de un hiperplano, el cual es formado por los pesos de la red; por ejemplo, para el caso de un perceptrón con dos entradas, la función determinada por

Entrada		Salida
+1	−1	+1
+1	+1	+1
−1	−1	+1
−1	+1	−1

es un ejemplo de función separable con un solo hiperplano, mientras que la función

Entrada		Salida
+1	−1	−1
+1	+1	+1
−1	−1	+1
−1	+1	−1

no es separable, pues no existe una línea recta capaz de lograr la separación de los patrones de entrada, como puede observarse en la Fig. (1.7).

1.5 PERCEPTRÓN MULTICAPAS

Minsky y Papert fueron los primeros en mostrar la incapacidad de los perceptrones simples para calcular funciones no separables; sin embargo, también señalaron que un nivel de unidades entre el nivel de entrada y el de salida (denominadas unidades ocultas) permitiría a la red realizar el mapeo de cualquier función desde las unidades de entrada hacia las unidades de salida[5]. Los perceptrones multicapas son redes neuronales alimentadas hacia adelante, con uno o más niveles de unidades ocultas. El trabajo de Minsky y Papert suspendió durante mucho tiempo las investigaciones sobre perceptrones debido a la inexistencia de un algoritmo capaz de modificar los pesos de las conexiones entre las unidades para la realización de una tarea específica en perceptrones de múltiples capas. En las últimas 2 décadas estos algoritmos han sido desarrollados y aunque no se ha probado formalmente su convergencia, se ha conseguido exitosamente la adaptación de los pesos en muchos problemas de interés.

En la Fig. 1.8 se muestra un perceptrón multicapas. La j-ésima unidad del l-ésimo nivel esta etiquetada como $\mathcal{O}_j^{(l)}$, para $j = 1, 2, ..., J(l)$ y $l = 1, 2, ..., L$ y en donde $l = 1$ representa la capa de entrada y L representa el nivel de salida, mientras que $J(l)$ es el número de unidades del l-ésimo nivel. El peso de la conexión entre la j-ésima unidad de la capa $l-1$, $\mathcal{O}_j^{(l-1)}$, y la i-ésima unidad de la capa l, $\mathcal{O}_i^{(l)}$, esta representado por $w_{ij}^{(l)}$.

El algortimo de aprendizaje más utilizado para este tipo de redes neuronales es el *algoritmo de retropropagación*[2] cuyo origen se describió en la sección 1.2.3, el cual es una generalización de la regla delta. Este algoritmo utiliza una técnica de gradiente descendente para minimizar una función de error definida como el cuadrado de la diferencia entre la salida actual de la red y la salida deseada. Una componente esencial de este algoritmo es el método recursivo que propaga el término de error necesario para actualizar los pesos desde las unidades de la capa de salida hacia las unidades en capas anteriores, lo que originó el nombre de retropropagación. El rendimiento mostrado en este algoritmo es algunas veces

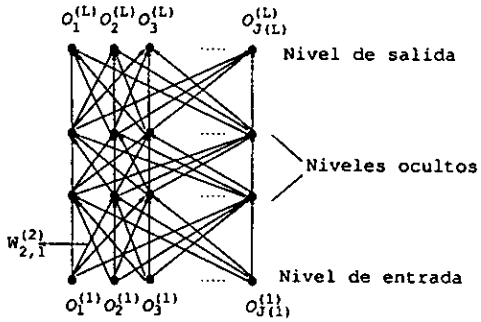

Figura 1.8: El perceptrón multinivel

sorprendente si se considera que puede encontrarse un mínimo local en la función de error en lugar de un mínimo global requerido.

1.5.1 El Algoritmo de Retropropagación

Los perceptrones multinivel utilizan funciones no lineales en sus unidades, las cuales deben ser crecientes y diferenciables, de tal manera que

$$O_i^{(l)} = f(\sum_q^{J(l-1)} w_{iq}^{(l)} O_q^{(l-1)}); \tag{1.38}$$

la función de error a minimizar por el algoritmo de retropropagación se define como

$$\mathcal{E} = \frac{1}{2}\sum_j^{J(L)} (d_j - O_j^{(L)})^2, \tag{1.39}$$

en donde d_j es el valor deseado en la j-ésima unidad del nivel de salida. El algoritmo de retropropagación de Rumelhart puede resumirse en los siguientes pasos:

1. Escoger los pesos $w_{ij}^{(l)}$ con valores aleatorios pequeños.
2. Calcular el error para cada uno de los P patrones de entrenamiento.
3. Si el error total satisface el criterio de convergencia, detener el algoritmo, de otra manera actualizar los pesos con la formula

$$w_{ij}^{(l)}(k+1) = w_{ij}^{(l)}(k) + \Delta w_{ij}^{(l)}(k), \tag{1.40}$$

$$\Delta w_{ij}^{(l)}(k) = -\eta \nabla \mathcal{E}(w_{ij}^{(l)})(k) + \alpha \Delta w_{ij}^{(l)}(k-1); \tag{1.41}$$

en donde $\nabla\mathcal{E}(w_{ij}^{(l)})(k)$ es calculado con las contribuciones de cada uno de los P patrones, η es conocido como el coeficiente de aprendizaje y α es conocido como el coeficiente de momento. Si el coeficiente de momento es cero, los pesos se modifican en dirección al gradiente del error como función de los pesos, esto es:

$$\Delta w_{ij}^{(l)} \sim -\frac{\partial \mathcal{E}}{\partial w_{ij}^{(l)}};$$

este gradiente puede expresarse como el producto de dos derivadas:

$$\frac{\partial \mathcal{E}}{\partial w_{ij}^{(l)}} = \frac{\partial \mathcal{E}}{\partial \mathcal{O}_i^{(l)}} \frac{\partial \mathcal{O}_i^{(l)}}{\partial w_{ij}^{(l)}}, \tag{1.42}$$

en donde:

$$\frac{\partial \mathcal{O}_i^{(l)}}{\partial w_{ij}^{(l)}} = \frac{\partial f(\sum_{q=1}^{J(l-1)} w_{iq}^{(l)} \mathcal{O}_q^{(l-1)})}{\partial w_{ij}^{(l)}} = f_i^{'(l)} \mathcal{O}_j^{(l-1)}, \tag{1.43}$$

y por lo tanto

$$\frac{\partial \mathcal{E}}{\partial w_{ij}^{(l)}} = \frac{\partial \mathcal{E}}{\partial \mathcal{O}_i^{(l)}} f_i^{'(l)} \mathcal{O}_j^{(l-1)}. \tag{1.44}$$

Si definimos

$$\delta_i^{(l)} \equiv \frac{\partial \mathcal{E}}{\partial \mathcal{O}_i^{(l)}} f_i^{'(l)}, \tag{1.45}$$

entonces tenemos la ecuación general

$$\frac{\partial \mathcal{E}}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} \mathcal{O}_i^{(l-1)}, \tag{1.46}$$

en donde las $\delta_i(l)$ se calculan de la siguiente manera: podemos calcular la derivada del error con respecto a los niveles de activación $\mathcal{O}_i^{(l)}$ como el producto de dos derivadas: la derivada del error con respecto a los niveles de activación de las unidades de un nivel dado multiplicada por la derivada del nivel de activación de estas unidades con respecto al nivel de activación de las unidades de la capa inmediata anterior, esto es:

$$\frac{\partial \mathcal{E}}{\partial \mathcal{O}_i^{(l)}} = \sum_{k=1}^{J(l+1)} \frac{\partial \mathcal{E}}{\partial \mathcal{O}_k^{(l+1)}} \frac{\partial \mathcal{O}_k^{(l+1)}}{\partial \mathcal{O}_i^{(l)}}; \tag{1.47}$$

en el segundo término de esta derivada tenemos

$$\frac{\partial \mathcal{O}_k^{(l+1)}}{\partial \mathcal{O}_i^{(l)}} = \frac{\partial f(\sum_q w_{kq}^{(l+1)} \mathcal{O}_q^{(l)})}{\partial \mathcal{O}_i^{(l)}} = f_k^{'(l+1)} w_{ki}^{(l+1)}, \tag{1.48}$$

con lo cual la Ec. (1.47) queda como

$$\frac{\partial \mathcal{E}}{\partial \mathcal{O}_i^{(l)}} = \sum_{k}^{J(l+1)} \frac{\partial \mathcal{E}}{\partial \mathcal{O}_k^{(l+1)}} f_k^{'(l+1)} w_{ki}^{(l+1)}; \tag{1.49}$$

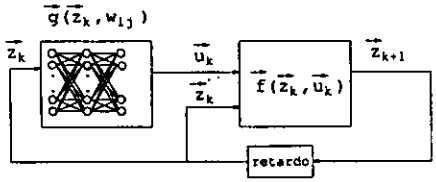

Figura 1.9: Esquema de un sistema dinámico con una red neuronal como controlador

utilizando la definición de $\delta_i^{(l)}$ dada en la Ec. (1.45) podemos escribir

$$\frac{\partial \mathcal{E}}{\partial \mathcal{O}_i^{(l)}} = \sum_k^{J(l+1)} \delta_k^{(l+1)} w_{ki}^{(l+1)}; \tag{1.50}$$

multiplicando ambos lados de esta ecuación por $f_i^{'(l)}$ tenemos

$$\delta_i^{(l)} = f_i^{'(l)} \sum_k^{J(l+1)} \delta_k^{(l+1)} w_{ki}^{(l+1)}. \tag{1.51}$$

Este es un sistema de ecuaciones recursivo el cual se inicializa en la capa de salida con:

$$\delta_i^{(L)} = \frac{\partial \mathcal{E}}{\partial \mathcal{O}_i^{(L)}} f_i^{'(L)}$$

ó

$$\delta_i^{(L)} = -(d_i - \mathcal{O}_i^{(L)}) f_i^{'(L)} \tag{1.52}$$

en donde hemos hecho uso de la definición de $\mathcal{E}$ de la Ec. (1.39).

Aunque el método de retropropagación es esencialmente una técnica para calcular el gradiente del error con respecto a los pesos de las conexiones del perceptrón, se ha dado en llamar algoritmo de retropropagación al método iterativo propuesto por Rumelhart; sin embargo, existen otros métodos que emplean la retropropagación para calcular el gradiente pero que actualizan los pesos de las conexiones en foma distinta, tales como el método de gradientes conjugados, de gradiente óptimo, etc.

1.6 APLICACIÓN AL CONTROL DE SISTEMAS DINÁMICOS DISCRETOS.

Consideraremos ahora una aplicación de las redes neuronales al control de sistemas dinámicos[6]. La configuración en la que estaremos interesados esta esquematizada en la Fig (1.9), en donde el sistema dinámico discreto evoluciona de acuerdo a la regla $\vec{z}_{k+1} = \vec{f}(\vec{z}_k, \vec{u}_k)$, donde $\vec{z}_k$ representa el estado del sistema a los tiempos

$\{k, k = 1, 2, ...\}$ y $\vec{u}_k$ es la variable de control. El objetivo de la red neuronal artificial es el generar la secuencia de acciones de control $\{\vec{u}_k, k = 1, 2, ...\}$ de acuerdo a la ley de control $\vec{u}_k = \vec{g}(\vec{z}_k, w_{ij}^{(l)})$, donde $w_{ij}^{(l)}$ son los pesos o parámetros de interconexión entre los nodos de la red neuronal. Esta secuencia de señales de control debe conducir al sistema desde un estado inicial $\vec{z}_i$ hasta un estado final deseado $\vec{z}_t$. Debido a la capacidad mostrada por los perceptrones multinivel para mapear cualquier función no lineal, éstos han sido utilizados exitosamente en los últimos años como controladores, es decir, para modelar la función $\vec{g}$.

Como vimos en la sección anterior, en el uso de redes neuronales artificiales en problemas convencionales de reconocimiento de patrones, el algoritmo de retropropagación propuesto por Rumelhart y colaboradores nos permite determinar el gradiente de la función de error con respecto a los pesos de la red $\partial E/\partial w_{ij}^{(l)}$, propagando una señal de error determinada a partir de salidas deseadas en la red; sin embargo, en sistemas como el mostrado en la Fig. (1.9) las salidas del perceptrón (i.e. las señales de control) son en la mayoría de los casos desconocidas, siendo el único dato disponible el estado deseado del sistema $\vec{z}_t$. El algoritmo de *retropropagación en el tiempo* es un método que propaga la señal de error necesaria para la modificación de los pesos en el proceso de entrenamiento a partir de la diferencia entre un estado dado y un estado deseado. Este algoritmo se basa en el concepto de *derivadas parciales ordenadas*, el cual revisaremos a continuación.

1.6.1 Derivadas Parciales Ordenadas

Para definir las derivadas parciales ordenadas, primero definiremos el concepto de conjunto de ecuaciones algebráicas ordenadas[6]. Sea $\{y_1, y_2, ..., y_n\}$ un conjunto de variables cuyos valores son determinados a partir de n ecuaciones. Se dice que este conjunto de ecuaciones es ordenado si cada variable y_i es una función únicamente de las variables $\{y_1, y_2, ..., y_{i-1}\}$, esto es, la ecuación de cualquier variable perteneciente al conjunto de ecuaciones ordenadas puede escribirse como:

$$y_i = f_i(y_1, y_2, ..., y_{i-1}); \tag{1.53}$$

en otras palabras, dada la naturaleza ordenada del conjunto de ecuaciones, para obtener el valor de y_i es necesario conocer los valores de las variables $\{y_1, y_2, ..., y_{i-1}\}$. Tomemos por ejemplo el siguiente conjunto de ecuaciones ordenadas

$$y_1 = 3 \tag{1.54}$$
$$y_2 = 4y_1 + 2 \tag{1.55}$$
$$y_3 = 2y_1^2 - 5y_2. \tag{1.56}$$

Como todos sabemos, la derivada parcial ordinaria de y_3 con respecto a y_1 es

$$\frac{\partial y_3}{\partial y_1} = 4y_1; \tag{1.57}$$

sin embargo, si hacemos las sustituciones pertinentes para resolver el sistema tendremos que

$$y_3 = 2y_1^2 - 20y_1 - 10, \tag{1.58}$$

y por lo tanto en este caso

$$\frac{\partial y_3}{\partial y_1} = 4y_1 - 20, \tag{1.59}$$

resultado que obviamente es diferente al obtenido en la Ec. (1.57).

A diferencia de las derivadas parciales ordinarias, en donde todas las variables de la función a derivar permanecen constantes a excepción de la variable con respecto a la cual se deriva, en las derivadas parciales ordenadas las variables y las constantes son determinadas usando un conjunto ordenado de ecuaciones. Así, por ejemplo, los términos constantes en el cálculo de la derivada parcial ordenada de y_j con respecto a y_i (denotada por $\partial^+ y_j/\partial y_i$) serán $\{y_1, y_2, ...y_{i-1}\}$, mientras que las variables serán $\{y_i, y_{i+1}, ..., y_j\}$. La definición formal de una derivada parcial ordenada es entonces

$$\frac{\partial^+ y_j}{\partial y_i} = \left(\frac{\partial y_j}{\partial y_i}\right)_{\{y_1, y_2, ..., y_{i-1}\} constantes.} \tag{1.60}$$

Las derivadas parciales ordenadas cumple con las siguientes propiedades

$$\frac{\partial^+ y_{i+1}}{\partial y_i} = \frac{\partial y_{i+1}}{\partial y_i}, \tag{1.61}$$

$$\frac{\partial^+ y_j}{\partial y_i} = 0 \ si \ j < i, \tag{1.62}$$

$$\frac{\partial^+ y_j}{\partial y_j} = 1. \tag{1.63}$$

Para $j \geq i+1$ tenemos las siguientes definiciones equivalentes

$$\frac{\partial^+ y_j}{\partial y_i} = \frac{\partial y_j}{\partial y_i} + \sum_{k=i+1}^{j-1} \frac{\partial^+ y_j}{\partial y_k}\frac{\partial y_k}{\partial y_i}, \tag{1.64}$$

$$\frac{\partial^+ y_j}{\partial y_i} = \frac{\partial y_j}{\partial y_i} + \sum_{k=i+1}^{j-1} \frac{\partial z_j}{\partial y_k}\frac{\partial^+ y_k}{\partial y_i}. \tag{1.65}$$

Resolviendo la derivada del ejemplo anterior mediante la Ec. (1.64) tenemos:

$$\frac{\partial^+ y_3}{\partial y_1} = \frac{\partial y_3}{\partial y_1} + \frac{\partial^+ y_3}{\partial y_2}\frac{\partial y_2}{\partial y_1}, \tag{1.66}$$

en donde

$$\frac{\partial y_3}{\partial y_1} = 4y_1, \tag{1.67}$$

$$\frac{\partial^+ y_3}{\partial y_2} = \frac{\partial y_3}{\partial y_2} = -5, \tag{1.68}$$

$$\frac{\partial y_2}{\partial y_1} = 4; \tag{1.69}$$

sustituyendo obtendremos

$$\frac{\partial^+ y_3}{\partial y_1} = 4y_1 - 20; \tag{1.70}$$

resultado que es idéntico al obtenido en la Ec (1.59).

1.6.2 Algoritmo de Retropropagación en el Tiempo

Revisemos ahora el funcionamiento del sistema mostrado en la Fig. (1.9) para describir el algoritmo de retropropagación en el tiempo con base en el concepto de derivadas parciales ordenadas. Supongamos que al tiempo $k = 1$ el sistema dinámico se encuentra en un estado $\vec{z}_1$; en este tiempo el perceptrón tiene asociados los pesos $w_{ij}^{(l)}$, los cuales podrían haber sido asignados aleatoriamente. Con el objeto de simplificar la discusión denotaremos por Y_1 el conjunto de pesos que caracterizan al perceptrón, esto es,

$$Y_1 = \{w_{ij}^{(l)}\}; \tag{1.71}$$

las señales de control, esto es, las salidas del perceptrón, son una función de estos pesos así como del estado actual del sistema, es decir,

$$Y_2 = \vec{u}_1 = g(\vec{z}_1, w_{ij}^{(l)}), \tag{1.72}$$

a su vez, el estado del sistema al tiempo $k = 2$ esta determinado por la ecuación de evolución,

$$Y_3 = \vec{z}_2 = f(\vec{z}_1, \vec{u}_1); \tag{1.73}$$

si el sistema evoluciona durante N pasos en el tiempo tendremos las siguientes ecuaciones adicionales

$$Y_4 = \vec{u}_2 = g(\vec{z}_2, w_{ij}^{(l)}) \tag{1.74}$$

$$Y_5 = \vec{z}_3 = f(\vec{z}_2, \vec{u}_2) \tag{1.75}$$

$$Y_6 = \vec{u}_3 = g(\vec{z}_3, w_{ij}^{(l)}) \tag{1.76}$$

$$Y_7 = \vec{z}_4 = f(\vec{z}_3, \vec{u}_3) \tag{1.77}$$

$$\vdots$$

$$Y_{K-2} = \vec{u}_{N-1} = g(\vec{z}_{N-1}, w_{ij}^{(l)}) \tag{1.78}$$

$$Y_{K-1} = \vec{z}_N = f(\vec{z}_{N-1}, \vec{u}_{N-1}), \tag{1.79}$$

además, si la función de error esta definida por

$$E = \frac{1}{2}\sum_i (z_{N_i} - z_{t_i})^2 \tag{1.80}$$

entonces tendremos una última ecuación

$$Y_K = E = h(\vec{z}_N). \tag{1.81}$$

El conjunto de variables así definidas se identifica claramente como un conjunto de variables ordenadas, de donde

$$\frac{\partial E}{\partial w_{ij}^{(l)}} = \frac{\partial^+ Y_K}{\partial Y_1}. \tag{1.82}$$

Como vimos anteriormente

$$\frac{\partial^+ Y_K}{\partial Y_1} = \frac{\partial Y_K}{\partial Y_1} + \sum_{j=2}^{K-1} \frac{\partial^+ Y_K}{\partial Y_j}\frac{\partial Y_j}{\partial Y_1}; \tag{1.83}$$

las derivadas ordenadas del segundo término de la ecuación anterior pueden escribirse como

$$\begin{aligned}\frac{\partial^+ Y_K}{\partial Y_1} &= \frac{\partial^+ Y_K}{\partial Y_2}\frac{\partial Y_2}{\partial Y_1} + \frac{\partial^+ Y_K}{\partial Y_3}\frac{\partial Y_3}{\partial Y_1} \\ &+ \frac{\partial^+ Y_K}{\partial Y_4}\frac{\partial Y_4}{\partial Y_1} + \frac{\partial^+ Y_K}{\partial Y_5}\frac{\partial Y_5}{\partial Y_1} \\ &\vdots \\ &+ \frac{\partial^+ Y_K}{\partial Y_{K-1}}\frac{\partial Y_{K-1}}{\partial Y_1},\end{aligned} \tag{1.84}$$

donde hemos usado las Ec. (1.72)-(1.80) para deducir que

$$\frac{\partial Y_K}{\partial Y_j} = 0; \quad j = 1, 2, 3, ...K-2. \tag{1.85}$$

Observando las Ec. (1.73), (1.75), (1.77) y (1.79) podemos deducir que para cualquier Y_k correspondiente a una varible de estado $\vec{z}_n$

$$\frac{\partial Y_k}{\partial Y_1} = \frac{\partial \vec{z}_n}{\partial w_{ij}^{(l)}} = 0 \tag{1.86}$$

para $k = 1, 2, 3...k-1$, $n = 1, 2, ..., N$, debido a lo cual la Ec. (1.84) queda como

$$\frac{\partial E}{\partial w_{ij}^{(l)}} = \sum_{n=1}^{N-1} \frac{\partial^+ E}{\partial \vec{u}_n}\frac{\partial \vec{u}_n}{\partial w_{ij}^{(l)}}, \tag{1.87}$$

donde N es el número total de pasos en el tiempo en la evolución del sistema. De las Ecs (1.72), (1.74), (1.76) y (1.78) podemos deducir que la ecuación

$$\frac{\partial^+ Y_K}{\partial Y_j} = \sum_{m=j+1}^{K-1} \frac{\partial^+ Y_K}{\partial Y_m}\frac{\partial Y_m}{\partial Y_j} \quad para \ j \neq K-1 \tag{1.88}$$

se reduce a

$$\frac{\partial^+ E}{\partial \vec{u}_n} = \frac{\partial^+ E}{\partial \vec{z}_{n+1}} \frac{\partial \vec{z}_{n+1}}{\partial \vec{u}_n}; \quad n = 1, 2, ...N-1 \tag{1.89}$$

para todo $Y_j = \vec{u}_n$, esto debido a que

$$\frac{\partial \vec{z}_n}{\partial \vec{u}_m} = 0 \quad si \quad n \neq m+1; \tag{1.90}$$

de forma similar, de las Ecs. (1.73)-(1.80) podemos observar que

$$\frac{\partial \vec{u}_l}{\partial \vec{z}_m} = 0 \quad si \quad l \neq m \tag{1.91}$$

$$\frac{\partial \vec{z}_q}{\partial \vec{z}_r} = 0 \quad si \quad q \neq r+1 \tag{1.92}$$

de donde podemos deducir que para todo $Y_j = \vec{z}_n$

$$\frac{\partial^+ E}{\partial \vec{z}_{n+1}} = \frac{\partial E}{\partial \vec{u}_{n+1}} \frac{\partial \vec{u}_{n+1}}{\partial \vec{z}_{n+1}} + \frac{\partial^+ E}{\partial \vec{z}_{n+2}} \frac{\partial \vec{z}_{n+2}}{\partial \vec{z}_{n+1}}; \quad n = 1, 2, 3, ...N-2. \tag{1.93}$$

Las Ecs. (1.87), (1.89) y (1.93) forman un sistema de ecuaciones recursivas que nos permiten calcular las componentes del gradiente del error $\partial E / \partial w_{(ij)}^{(l)}$, teniendo como inicialización la siguiente expresión

$$\frac{\partial^+ E}{\partial \vec{z}_N} = \frac{\partial E}{\partial \vec{z}_N} = \sum_{i=1}^{J(L)} z_{N_i} - z_{t_i}. \tag{1.94}$$

Este sistema de ecuaciones, que constituyen el método de retropropagación en el tiempo, será utilizado en el Capítulo 4 para la construcción de un algoritmo de entrenamiento aplicado a la solución de un sistema dinámico. En el siguiente capítulo se hará una breve presentación de las arquitecturas paralelas, mostrando cuales son sus características, algunos ejemplos de máquinas paralelas existentes y de algunos paradigmas de programación paralela.

Capítulo 2

ARQUITECTURAS Y PROGRAMACIÓN DE MÁQUINAS PARALELAS

2.1 INTRODUCCIÓN

Una computadora paralela consiste en un conjunto de procesadores capaces de trabajar en forma cooperativa para resolver un problema específico de cómputo [7]. Esta definición abarca tanto supercomputadoras con cientos o miles de procesadores como conjuntos de estaciones de trabajo de un solo procesador o de múltiples procesadores conectadas a través de una red de comunicación. El cómputo paralelo es un área de la computación relativamente nueva y el interés en ésta ha crecido considerablemente en los últimos años debido a tres razones principales[7]:

- La magnitud de los problemas que se intentan resolver con una computadora ha aumentado. Por ejemplo, la simulación del clima en el planeta involucra un número del orden de 10^{16} operaciones de punto flotante, lo que implica 10 dias de ejecución a una velocidad de 10^{10} flops (i.e. operaciones de punto flotante por segundo). Como punto de comparación, la supercomputadora Cray Y-MP 4/64 de la UNAM puede trabajar a una velocidad máxima de 10^9 flops.
- El rendimiento de un sistema de cómputo está limitado por el intervalo de tiempo requerido para realizar una operación básica; éste es igual a la duración del ciclo de reloj de ese sistema. Los avances tecnológicos han reducido la duración de cada ciclo de reloj sistemáticamente a través de los años, sin embargo, esta reducción es cada vez menor, acercándose a límites fijados por la velocidad de la luz (ver Fig. 2.1). A pesar de este hecho, existe otra opción para incrementar el rendimiento del sistema: la de realizar simultáneamente

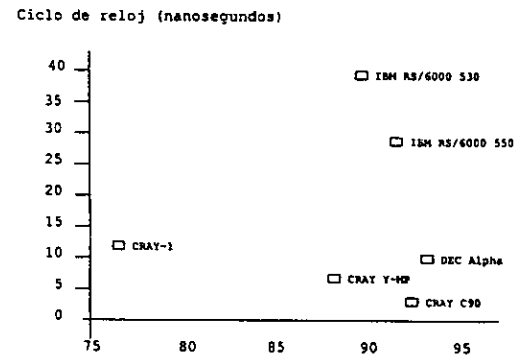

Figura 2.1: Disminución de la duración de los ciclos de reloj en los últimos años.

varias operaciones usando procesadores que trabajan en paralelo.

- La capacidad de las redes que conectan a las computadoras se ha incrementado grandemente. Por ejemplo, no hace mucho tiempo, la red más veloz transportaba datos a una velocidad de 1.5 Mbits por segundo. Se espera que al final de esta década dichas velocidades superen los 10^3 Mbits por segundo. Esta tendencia hará viables aplicaciones que requieran recursos físicamente distribuídos como si fueran parte de una misma máquina.

En este capítulo se presenta una breve revisión de algunos aspectos importantes del cómputo paralelo como son: tipos de máquinas paralelas, topologías de comunicación entre procesadores y paradigmas de programación concurrente.

2.2 TIPOS DE COMPUTADORAS PARALELAS

Existen varios cientos de proyectos de computadoras paralelas alrededor del mundo. La Tabla 2.1 muestra un subconjunto de éstos, pertenecientes tanto a universidades como a empresas, cubriendo un amplio rango de arquitecturas diferentes. Aunque varios de estos proyectos muy probablemente no conduzcan a máquinas viables, un número importante de éstos ha producido (o producirá) prototipos útiles. En esta tabla también se han incluido varias computadoras paralelas comerciales, tales como la CRAY Research T3D, la Intel iPSC2 y la Intel Paragon, la Meiko CS-2, la nCUBE 2, etc., así como el "cluster" de estaciones de trabajo Avalon, el cual es un conjunto de estaciones de trabajo conectadas mediante una red de alta velocidad.

Es posible clasificar a las computadoras paralelas usando diversas características como son: la forma en que los procesadores tienen acceso a la memoria de la máquina, el número de procesadores que constituyen a la máquina, la forma en que éstos se comunican y los paradigmas de programación soportados por la

Alliant Campus /800	Alliant FX /2800
Alliant FX /8	BBN Butterfly
CalTech Hypercube	Cedar Project
Connection Machine CM-2	Conecction Machine CM-5
CRAY T3D y CRAY T3E	Data-floy Machines
Denelcor HEP-1	Encore Multimax
ETA-10	FPS T-Series
Flex 1	Evans and Sutherland ES1
Fujitsu AP-1000	Goodyera MPP
IBM GF-11 and TF1	IBM SP-1
ICL DAP	Illiac IV
Intel DELTA	Intel iPSC /860 Hypercube
Intel /PSC2 Hypercube	Intel iWarp
Intel Paragon	Kendall Square KSR1
Intel Touchstone	Meiko MK860 y CS-2
Masspar MP1000	Myrias SPS-3
Multiflow	nCUBE2 Hypercube
Navier-Stokes Machine	Parsytec
NYU /IBM RP3	SUPRENUM-1
Sequent Balance	TERA
Symult 2010	Origin 2000
Avalon	

Tabla 2.1: Proyectos de computadoras paralelas

máquina[8]. A continuación se describirán algunas de las clasificaciones más comunes.

2.2.1 SIMD y MIMD

Las computadoras paralelas pueden dividirse en dos tipos: SIMD y MIMD (acrónimos para Single Instruction-Multiple Data stream y Multiple Instruction-Multiple Data stream, respectivamente). En una computadora tipo SIMD cada procesador ejecuta la misma instrucción con diferentes datos en cada ciclo de operación, mientras que una máquina MIMD cada procesador ejecuta instrucciones independientes de los otros. Una computadora MIMD incrementa el número de formas en las que el paralelismo puede ser implementado, sin embargo, la dificultad para programar estas máquinas es también mayor; es por esto que muchos diseños incorporan aspectos de ambos tipos.

Las computadoras SIMD son controladas por un solo programa, lo cual reduce la complejidad en su programación. Por otra parte, las computadoras MIMD, en principio, tendrán un programa diferente ejecutándose en cada procesador, lo cual produce un ambiente de programación sumamente complejo. Un paradigma de programación frecuentemente usado en máquinas MIMD es el modelo SPMD (Single Program-Multiple Data). En este modelo, el mismo texto es ejecutado por todos los procesadores pero cada ejecución puede seguir rutas diferentes a través

del programa en diferentes procesadores. Claramente, cualquier conjunto de P programas MIMD puede ser sustituido por un solo programa, con el costo de un pequeño incremento en la memoria utilizada por éste.

Algunos ejemplos de máquinas SIMD son la CM-200, la Maspar MP-2, la Thinking Machine CM-2 y la mayoría de las estaciones de trabajo de múltiples procesadores, mientras que entre las máquinas MIMD tenemos la Intel Paragon, la Thinking Machine CM-5, la Cray T3D, la Meiko CS-2, la nCUBE, la Silicon Graphics Origin 2000, entre otras.

2.2.2 Memoria Distribuída y Memoria Compartida

Otra categorización sencilla es dividir a las computadoras paralelas en máquinas con memoria local y máquinas con memoria global. En las máquinas con memoria local, también conocidas como sistemas de memoria distribuída, la comunicación entre procesadores es manejada completamente por una red de comunicación. Por otra parte, en las máquinas con memoria global, también conocidas como sistemas de memoria compartida, existe una única memoria que es accesible directamente a todos los procesadores. Los sistemas de memoria compartida ofrecen la ventaja de ser mucho mas fáciles de programar, sin embargo, la construcción de este tipo de máquinas que tengan también la propiedad de escalabilidad es extremadamente difícil.

En algunos sistemas de memoria distribuída es posible simular memoria compartida mediante software, o con una combinación de software y hardware. Estos sistemas son llamados de Memoria Virtual Compartida (VMS) y ofrecen la facilidad de uso de los sistemas de memoria compartida, preservando la escalabilidad del sistema distribuído. Aunque poderoso, este sistema es disponible en solo un número limitado de arquitecturas. Es importante enfatizar que este concepto no esta relacionado en modo alguno con el de Memoria Virtual basada en disco que manejan algunos sistemas operativos.

2.2.3 Topologías de Comunicación

La variedad en las topologías de comunicación es una de las características más importantes para diferenciar entre las distintas computadoras paralelas. Las vías de comunicación entre procesadores típicamente son de dos tipos: usando un bus o de punto a punto. Un bus tiene la ventaja de que muchos procesadores pueden utilizar una única vía de comunicación, sin embargo, el rendimiento efectivo del ancho de banda disminuye conforme el número procesadores aumenta. Con conexiones punto a punto los procesadores directamente conectados tendrán una comunicación muy eficiente, pero los procesadores sin esta conexión directa probablemente tendrán una sobrecarga importante, que incluye tanto un incremento en la latencia

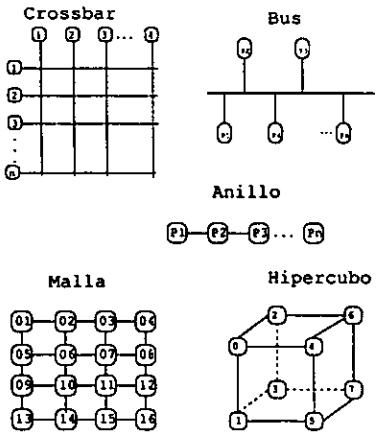

Figura 2.2: Topologías de conexión

entre los procesadores (es decir, el tiempo que un procesador tiene que esperar para que otro procesador responda a cierta petición) como una reducción del ancho de banda disponible.
En los últimos años, el ancho de banda en las comunicaciones ha aumentado prácticamente al mismo ritmo que la velocidad de los procesadores. Los sistemas Intel, por ejemplo, han incrementado este ancho de banda de unos cientos de Kbytes/sec en la iPSC1 a cerca de 200 Mbytes/sec en la Paragon. La latencia en las comunicaciones también es un aspecto que ha sido significativamente mejorado, aunque no al mismo ritmo, pasando de 5000 microsegundos en la iPSC1 a 60 microsegundos en la Paragon.

Las estrategias más comunes de interconexión utilizan arreglos simétricos simples, tales como anillos, mallas, hipercubos y conexiones completas (en donde cada uno de los procesadores tiene una conexión hacia todos los demás) también llamadas crossbars (ver Fig. 2.2). Tres aspectos caracterizan a una topología de intercomunicación:

- Número de alambres: El número total de conexiones punto a punto. Representa la complejidad en la construcción de la topología.
- Conectividad: El número de conexiones punto a punto asociadas a un solo procesador. Este parámetro determina la facilidad para aprovechar el máximo ancho de banda disponible.
- Distancia Máxima: El número de nodos que separan a los procesadores más lejanos. La distancia máxima es un indicador del rendimiento en el peor de los casos.

Topología	No. de alambres	Conectividad	Distancia Máxima
Crossbar	$P(P-1P)$	P	1
Anillo	$P-1$	2	$P-1$
Malla	$2(P-P^{\frac{1}{2}})$	4	$2(P^{\frac{1}{2}}-1)$
Hipercubo	$\frac{1}{2}P\,log_2P$	log_2P	log_2P

Tabla 2.2: Parámetros de las topologías

En la Tabla 2.2 se muestran los valores de estos parámetros para algunas de las topologías más simples, de donde podemos observar, por ejemplo, que una conexión crossbar tiene una una alta conectividad y la menor distancia máxima posible, pero que debido al alto número de alambres tiene un alto costo de construcción. Estos datos también proporcionan una idea de la escalabilidad de una máquina paralela, esto es, como se comporta el rendimiento de un sistema conforme el número de procesadores aumenta. Un sistema es escalable esencialmente si su rendimiento se incrementa linealmente con el número P de procesadores , es decir,

$$Perf(P) = O(P),$$

aunque en la práctica, si el rendimiento se mantiene por debajo de este valor por un factor logarítmico

$$Perf(P) = O(P/(lnP)^c)$$

se considera que el sistema es escalable.

En una conexión crossbar, por ejemplo, la inclusión de un nuevo nodo (un nuevo procesador) no disminuye en nada el ancho de banda disponible para cada procesador. En general, tanto la conectividad, la complejidad de construcción y la distancia máxima entre nodos nos proporcionan una idea de hasta dónde puede crecer el número de procesadores manteniendo el mismo rendimiento. En la siguiente sección se hará una revisión más detallada de estas topologías y sus características.

2.3 ARQUITECTURAS DE MÁQUINAS PARALELAS

Más alla de la simple clasificación de computadoras en los tipos mencionados en la sección anterior, existe una amplia gama de formas diferentes de construir una máquina paralela[8]. A continuación daremos un vistazo a las causas de este amplio rango discutiendo algunas de las diferentes topologías de comunicación que existen, así como algunos aspectos referentes al diseño de los nodos.

2.3.1 Diseño de Nodos

Entendemos por nodo una unidad de procesamiento individual junto con su electrónica de comunicación asociada y su memoria local, si se dispone de ésta. El diseño

de nodos tiende a ser el aspecto menos variable en las computadoras paralelas, ya que la mayoría de las arquitecturas utilizan tanto microprocesadores estándar como unidades de punto flotante y circuitos de memoria existentes en el mercado. Esto proporciona dos ventajas: el tiempo de inicio del proyecto se reduce y existen ya una cantidad importante de códigos de bajo nivel escritos para tales procesadores, como son compiladores, ensambladores y depuradores. Este es el motivo por el cual una gran cantidad de computadoras paralelas están basadas en alguno de los siguientes procesadores: DEC Alpha, IBM R6000, Inmos T8000 y T9000, Intel 80X86 e i860, Motorola 680X0, Sparc, MIPS R1000, así como de unidades de punto flotante Weitek; sin embargo, existen también procesadores desarrollados específicamente para computadoras paralelas, como es el caso de las máquinas CM-2, ES-1, HEP-1, iWarp, KSR1, Navier-Stokes y nCUBE. La memoria ocupa una cantidad importante de espacio en los nodos; los sistemas actuales poseen memorias en el rango de 1 a 128 Mb; la mayoría soporta varios niveles de memoria en cada nodo, como son memoria principal, memoria cache primaria y secundaria y registros. El manejo adecuado de la memoria cache es un aspecto crítico para el buen rendimiento de un nodo, así que el diseño de esta memoria y la calidad de los compiladores son aspectos esenciales en todos los sistemas másivamente paralelos.

Recientemente ha aparecido la tendencia de aumentar la densidad de empaquetamiento en los nodos. Los primeros ejemplos de ésto son las máquinas CM-1 y CM-2, las cuales tienen 16 procesadores en un solo microcircuito. En el caso particular de la CM-2, 32 procesadores (en dos microcircuitos adyacentes) comparten un procesador vectorial Weitek.

2.3.2 Aspectos de Comunicación

En esta sección se hará una descripción un poco más detallada de las topologías de comunicación mencionadas anteriormente.

Topologías basadas en bus

En una topología de comunicación basada en un bus, todos los procesadores comparten una única vía de comunicación. Como se ha mencionado, esta topología no es escalable debido a que el bus sólo puede ser utilizado por un procesador a la vez, lo que reduce el ancho de banda (AB) efectivo disponible para cada procesador, esto es, $AB \approx \frac{1}{p}$ si todos los procesadores intentan comunicarse simultáneamente.

A pesar de esto, las conexiones basadas en bus son atractivas debido a su facilidad de construcción y pueden ser usadas siempre y cuando el número de procesadores sea moderado, digamos entre 16 y 32. Más alla de este rango. la disminución en el ancho de banda es ya inaceptable y el uso de esquemas jerárquicos se hace necesario. Por ejemplo, la máquina SUPRENUM-1 utiliza un bus para

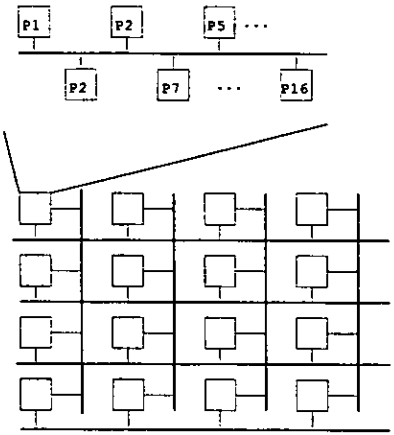

Figura 2.3: Topología jerárquica de la SUPRENUM-1

conectar 16 grupos de 16 procesadores. La configuración de estos grupos es un arreglo rectangular y la comunicación entre ellos se realiza usando buses separados para las columnas y para los renglones (ver Fig. 2.3).

Crossbars

La topología crossbar proporciona el mayor ancho de banda efectivo entre los procesadores. En esta topología, cada uno de los P procesadores del sistema tiene una conexión directa hacia todos los demás, lo que resulta en $P(P-1)$ alambres y una conectividad igual a P. Además, la máxima distancia que separa a dos procesadores es igual a 1. Debido a estas dos últimas características, la topología crossbar es altamente escalable, sin embargo, la alta cantidad de cables necesarios para su construcción hace demasiado costosa la construcción de máquinas con un número muy grande de procesadores usando esta topología.

Mallas y anillos

En una topología de malla de dimensión D, cada procesador esta conectado directamente a sus $2D$ vecinos (un anillo es una malla de una dimensión). Así por ejemplo, en una configuración de anillo cada procesador esta conectado a otros 2 procesadores, lo que da una conectividad de 2, mientras que en una malla de 2 dimensiones cada procesador esta conectado a 4 vecinos, lo que resulta en una conectividad de 4. El número de alambres necesarios para construir un anillo es $P-1$, sustancialmente menor que en un crossbar, mientras que la distancia entre los procesadores más lejanos también es $P-1$. En una malla de 2 dimen-

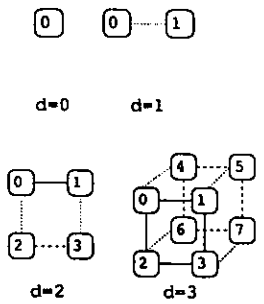

Figura 2.4: Esquemas de hipercubos de 0, 1, 2 y 3 dimensiones.

siones con P procesadores, estos forman un arreglo bidimensional con $P^{\frac{1}{2}}$ procesadores por lado. El número de alambres necesario para conectarlos es entonces $2P^{\frac{1}{2}}(P^{\frac{1}{2}}-1) = 2(P-P^{\frac{1}{2}})$, mientras que la distancia entre los procesadores más lejanos es $2(P^{\frac{1}{2}}-1)$.

Aunque la conectividad en las topologías de malla permanece constante aún con un aumento en el número de procesadores, la cantidad de alambres necesarios para su construcción es mucho menor que en un crossbar, lo que proporciona a esta topología una buena escalabilidad.

Hipercubos

La definición de hipercubo es recursiva. Un hipecubo de dimensión 0 esta formado por un solo nodo, mientras que un hipercubo de dimensión $D+1$ consiste en la unión de 2 hipercubos de dimensión D. Así, un hipercubo de dimensión 1 es formado por 2 nodos conectados entre sí; un hipercubo de dimensión 2 es una malla de 2 por 2 nodos, mientras que un hipercubo de dimensión 3 es un cubo (ver Fig. 2.4). Un hipercubo de dimensión D tiene 2^D nodos y cada nodo está conectado directamente a otros D nodos, por lo que se necesitan $2^D(D/2)$ alambres para construirlo.

La conectividad de esta topología aumenta con el número de nodos, mientras que el número de alambres necesarios para construirlo es menor que en el caso del crossbar. Estas características explican la popularidad de los diseños basados en hipercubos, ya que ofrecen una muy buena escalabilidad con un relativamente bajo costo de construcción.

Continuamente se proponen nuevas configuraciones de conexión. Una técnica común es el uso de "worm-hole" para el ruteo en sistemas distribuídos. La idea básica de esta técnica es establecer canales entre procesadores remotos para evitar la necesidad de interrumpir a los nodos intermedios. El proceso de creación de estos

canales, aunque representa una baja sobrecarga, permite a los datos transitar por el canal sin sobrecargas adicionales tales como múltiples costos de inicio de la comunicación en los nodos intermedios.

2.4 PARADIGMAS DE PROGRAMACIÓN CONCURRENTE

En contraste con los procesadores secuenciales, que son controlados por un programa que contiene un conjunto de instrucciones que se ejecuta en un orden específico y los cuáles usan la memoria disponible para desarrollar los cálculos deseados, las computadoras paralelas son mucho más complejas; como se ha mencionado, éstas están formadas por un número determinado P de procesadores secuenciales, cada uno con una memoria local generalmente no accesible en forma directa a los otros procesadores. Una computadora paralela es controlada por un programa que define y ordena las operaciones ejecutadas en cada procesador y que además realiza los procedimientos necesarios para tener acceso a la memoria local del resto de los procesadores. Así, un programa paralelo consiste en un conjunto de P programas secuenciales cada uno ejecutándose en un procesador diferente, junto con un conjunto de instrucciones que sincronizan las operaciones entre los diversos procesadores y que permiten el acceso de cada procesador a la memoria de los otros[8].

Las herramientas de programación para las computadoras paralelas disponibles en la actualidad son limitadas. En la mayoría de los casos los fabricantes de estas máquinas proveen un compilador de C y/o Fortran, los cuales comúnmente no poseen conceptos de paralelismo o capacidades para comunicación. Ejemplos de éstos son los sistemas suministrados por Intel, Meiko y nCUBE. En este tipo de sistemas, todas las comunicaciones y procesos de control deben ser iniciados explícitamente por el usuario, lo que conduce a una cantidad importante de modificaciones al código y por lo tanto, en pérdida de portabilidad. Los procesos de control y de comunicaciones se realizan a través de bibliotecas de bajo nivel (i.e. llamadas a funciones que son propias de una arquitectura particular).

Solo unos cuantos fabricantes han ido más allá, suministrado a los usuarios extensiones de dichos lenguajes que capturan algunos aspectos del hardware paralelo. Thinking Machines, por ejemplo, provee un Fortran paralelo llamado CMF en sus máquinas CM-2 y CM-5. Este lenguaje soporta las extensiones de arreglos del F90; la convención en este lenguaje consiste en que los objetos utilizados como arreglos son distribuídos entre los procesadores. Este ambiente de programación es fácil de implementar en la máquina CM-2 debido a su naturaleza SIMD. Nuevos lenguajes, como el HPF (Fortran de Alto Rendimiento) tienen un funcionamiento similar al CMF, pero ofrecen ventajas solo para máquinas esencialmente SIMD.

Las máquinas SPS-2 y SPS-3 (Myrias Corporation) soportan un espacio virtual de direcciones de memoria entre procesadores. Si un procesador intenta tener acceso a una localidad de memoria que no está en su memoria física, la página de memoria apropiada es transladada desde el procesador que la posee. En estas máquinas también se ha implementado un mecanismo sofisticado para el balance de carga de trabajo entre los procesadores. El sistema Myrias fue el primero en proveer memoria compartida virtual en una arquitectura de memoria distribuída. La máquina KSR2, de diseño más reciente, también utiliza este método; implementa un espacio global de direcciones de 40 bits, por medio de un hardware asistente para operaciones de memoria virtual compartida.

El intercambio de mensajes es otro paradigma comúnmente utilizado para escribir un programa en una máquina paralela[7],[8]. Este paradigma provee los dos aspectos claves para la programación en este tipo de máquinas: a) la sincronización de procesos y b) el acceso de lectura/escritura de cada procesador a la memoria de los restantes. El intercambio de mensajes es definido por la forma en la cual se lleva a cabo el acceso a las memorias de los procesadores: un proceso *a* puede mandar una copia de alguna parte de su memoria (a la cual se denomina *mensaje*) a otro proceso *b*. El proceso *b* que recibe el mensaje, lo escribe en un lugar predeterminado de su memoria. Esto es equivalente a que el proceso que envía un mensaje modifique la memoria del proceso que recibe dicho mensaje. Además, procesos que trabajan en conjunto pueden hacer uso del intercambio de mensajes como un mecanismo de sincronización. Estas características proporcionan todas las funciones necesarias para construir un método de programación paralela.

Un programa paralelo basado en este paradigma consiste en P programas secuenciales que usan instrucciones de intercambio de mensajes para acceder a la memoria de otros procesos y sincronizarse con ellos. Típicamente, estas instrucciones pertenecen a un conjunto limitado que constituye el ambiente de intercambio de mensajes y generalmente son disponibles al programador en forma de bibliotecas. Es decir, este ambiente de programación de computadoras paralelas consiste de un lenguaje secuencial estándar, tal como C o Fortran, junto con una biblioteca de intercambio de mensajes.

Mientras que todos los sistemas de intercambio de mensajes son similares desde el punto de vista lógico, existen diferencias en la forma en la cual se implementan las dos funciones clave (i.e. sincronización e intercambio de datos). Esto ha conducido a la existencia de un gran número de sistemas de intercambio de mensajes nativos diferentes e incompatibles. Algunos de los más conocidos son: Caltech's CROS, IBM's EUI, Intel's NX, Meiko's CS, nCUBE's PSE y Thinking Machines CMMD. Varios grupos han intentado superar las peculiaridades específicas de cada fabricante en los ambientes de intercambio de mensajes mediante el desarrollo de los llamados ambientes portátiles de intercambio de mensajes. La idea fundamental de estos proyectos es escoger las especificaciones de un ambiente en particular e implementarlo en todas las plataformas (ya sea como bibliotecas de intercambio

nativas o haciendo las llamadas apropiadas al ambiente de intercambio residente). Los programas escritos usando estos ambientes portátiles son así mismo portátiles a todos los sistemas que soporten estos ambientes, lo cual constituye una gran ventaja. Algunos ejemplos de este tipo de ambientes son EXPRESS, Linda, P4, PARMACS, PVM y ZIPCODE.

El gran número de ambientes portátiles ha originado a su vez algunos de los mismos problemas que los ambientes nativos presentan, esto es, debido a la gran variedad de interfaces de los fabricantes la mayoría de estos sistemas son incompatibles, además de que sólo soportan un subconjunto de los aspectos de paralelismo de los sistemas propietarios sin aprovechar completa y eficientemente los recursos disponibles. Esta razón originó la formación del comité MPI (Message Passing Interface) en el año de 1992. El principal objetivo de este comité fue definir una interfaz estándar de intercambio de mensajes, la cual fue llamada precisamente MPI. Esta interfaz contiene los mejores aspectos de los diversos ambientes, tanto nativos como portátiles[10].

2.5 EJEMPLOS DE AMBIENTES DE INTERCAMBIO DE MENSAJES

En esta sección se presentarán brevemente algunos ejemplos de ambientes de intercambio de mensajes, tanto nativos como portátiles.

2.5.1 Arquitecturas Paralelas y sus Ambientes de Programación Nativos

Caltech Hypercube

La computadora Caltech Hypercube es una computadora paralela sobre la cual se han basado muchos proyectos posteriores; fue construída en 1984 basándose en un hipercubo de 6 dimensiones. Cada vértice del hipercubo contenía un paquete de procesador-coprocesador Intel 286/287 y 64 KBytes de memoria, mientras que cada arista del mismo constituía un canal de comunicación permitiendo la transferencia de datos entre procesadores adyacentes.

Además de los 64 (2^6) nodos, esta máquina poseía un procesador adicional llamado Intermediate Host (IH). Todas las comunicaciones del usuario con el hipercubo se realizaban a través del IH. El ambiente de programación del sistema estaba formado por compiladores de los lenguajes Fortan 77 y C, junto con una biblioteca de comunicaciones conocida como Crystaline Operating System (CROS).

La numeración de los nodos del hipercubo se hizo asignando números binarios

Funciones nodo-host	
wtIH(data,CUBE)	Escribe un paquete del IH al nodo 0
rdsig(data)	Cada nodo esta listo para recibir datos
wtrest(data)	Envía un dato de un nodo al IH
rdbuff(data,CUBE,p)	El IH lee todos los datos envíados por los nodos
Funciones nodo-nodo	
wtELT(data, chan)	Envía datos a través de chan
rdELT(data, chan)	Recibe datos de chan

Tabla 2.3: Conjunto de funciones del Caltech Hypercube CROS

que difieren en tan solo un bit entre dos nodos adyacentes. Con base en esta numeración fue definido el concepto de *canal*. Un canal es la conexión entre dos nodos vecinos. Como hemos visto anteriormente, en un hipercubo cada nodo tiene $d = log_2(p)$ conexiones directas a otros nodos. Esto es, cada nodo tiene d canales numerados 0,1,...,$d-1$. Se definió también un canal extra de comunicación entre el nodo 0 y el nodo IH; de esta forma todos los intercambios de datos pueden especificarse por medio de un canal.

El conjunto de instrucciones básicas de este ambiente se da en la Tabla 2.3. Las primeras cuatro instrucciones permiten la comunicación entre los nodos del hipercubo y el nodo IH. Un programa típico en este ambiente comienza con una llamada a la función *wtIH* la cual envía un dato del nodo IH al nodo 0. Este nodo responde con una llamada a la función *rdsig*. Este proceso se usa para indicar a los nodos del hipercubo el inicio del trabajo. Al término de éste cada nodo entrega sus resultados mediante una llamada a la función *wtres*, la cual permite que cada nodo comunique un dato de 8 bytes al nodo IH. Este nodo puede leer los datos provenientes de todos los nodos mediante la función *rdbufIH*. Las otras dos funciones de la tabla permiten la comunicación entre nodos. La función *wtELT* envia un paquete de 8 bytes al nodo vecino conectado al canal *chan*, mientras que *rdELT* recibe un dato de 8 bytes proveniente del canal *chan*. Nótese que solo comunicaciones entre nodos adyacentes son permitidas[8].

Intel iPSC1

La computadora paralela iPSC1 fue diseñada por Intel poco tiempo después de la construcción del Caltech Hipercube, con un diseño muy parecido. El sistema se basó en procesadores 80286 con 512 Kbytes de memoria. La iPSC1 fue diseñada para soportar un hipercubo de 7 dimensiones, y al igual que la de Caltech, utilizó una máquina anfitriona, en este caso una estación de trabajo Xenix.

Aunque el hardware básico de la iPSC1 era fundamentalmente igual al de la Caltech, el ambiente de programación desarrollado, conocido como el NX1 (ver Tabla 2.4), fue totalmente diferente. Este ambiente se basó en el *Reactive Kernel*,

Funciones nodo-nodo	
chan=copen(pid)	Abre un canal virtual
send(chan,type,mesg,len,node,pid)	Envía un mensaje al proceso pid en el nodo node
recv(ci,type,msg,len,&ent,&node,&pid)	Recibe un mensaje del proceso pid en el nodo node
length=probe(chan,type)	Obtiene la longitud de los datos de un mensaje a recibir
status(chan)	Obtiene el estado del canal (ocupado o desocupado)
flick()	Retardo
sendw(chan,type,mesg,len,node,pid)	Envía un mensaje (bloqueo) al proceso pid del nodo node
recvw(ci,type,mesg,len,&ent,&node,&pid)	Recibe un mensaje (bloqueo) del porcesador pid en el nodo node
syslog(pid,string)	Envia un mensaje al nodo anfitrión.
Funciones del anfitrión	
sendmsg(chan,type,mesg,len,node,pid)	Envía un mensaje (bloqueo) del host al proceso pid en el nodo node
recvmsg(ci,&type,msg,len,&ent,&node,&pid)	El host recibe el mensaje (bloqueo) del proceso pid en el nodo node

Tabla 2.4: Funciones del NX1

desarrollado también por Caltech. Los aspectos clave desarrollados en el Reactive Kernel se han convertido en aspectos importantes de los ambientes de intercambio de mensajes modernos. A continuación hacemos una breve revisión de éstos.

Independencia de la topología. Las rutinas de comunicación intentan ocultar tanto como sea posible los detalles de la topología de conexión. Cada procesador es representado por un entero en el rango de [0,P-1], donde P es el número de procesadores. El anfitrión es denotado por un entero especial HOSTID, el cual es un entero negativo.

Nodos multiprocesos. A diferencia del sistema Caltech, cada procesador de la iPSC1 puede tener varios procesos corriendo simultáneamente, distinguiéndose por un número identificador de proceso *pid*, el cual es asignado localmente en cada nodo. Como los procesos son las entidades lógicas que realizan las comunicaciones, una operación de comunicación debe especificarse por el procesador destino y el pid del proceso receptor.

Acceso transparente. El sistema NX1 permite la comunicación de un proceso dado con cualquier otro proceso y no solo con procesos ejecutándose en nodos vecinos. Además, no hace distinción entre nodos vecinos y nodos no vecinos, ya que todos son representados por un número entero.

Comunicación MIMD. En este sistema pueden participar en un proceso de comunicación todos los nodos, algunos de ellos o solamente uno. Aunque existen operaciones globales que exigen la participación de todos los nodos, la mayoría de las operaciones de comunicación consisten únicamente de un proceso que envía un mensaje y otro que lo recibe.

Comunicación de no bloqueo. Las operaciones *sendw* y *recvw* no permiten la continuación del flujo del programa hasta que la operación que realizan sea completada. Esta situación puede provocar "candados" en la ejecución del programa paralelo si el mensaje enviado no especifica correctamente que proceso será el receptor o si existe algún otro error en la semántica del programa. Las rutinas de no bloqueo (*send* y *recv*) permiten continuar con el flujo del programa aunque éstas no hayan completado sus operaciones, por lo que es necesario utilizar alguna otra rutina para obtener información acerca del estado de dicho proceso de comunicación (*status*).

Comunicación asíncrona. La función *probe* permite a un proceso determinar si existe algún mensaje dirigido a él. El proceso receptor puede así decidir tomar este mensaje o ignorarlo.

Mensajes tipificados. Algunas veces puede suceder el hecho de que varios mensajes sean dirigidos a un proceso y éstos lleguen en un orden diferente al que se desea recibir. Para soslayar esta situación, un mensaje en NX1 requiere del uso de una etiqueta definida por el usuario mediante un número entero como identificación extra.

Longitud de mensaje desconocida. Frecuentemente sucede que el proceso receptor desconoce la longitud del mensaje a recibir; a pesar de que las rutinas de recepción requieren un buffer en donde se colocará el mensaje y la especificación de su longitud, este puede ser recibido aún si su longitud es diferente a la especificada en el proceso receptor. El parámetro *ent*, almacena el número de bytes recibidos en una operación de comunicación.

Intel iPSC2

El sistema NX2 (ver Tabla 2.5) se utiliza en las computadoras Intel iPSC/860 e Intel Paragon. Este sistema introdujo algunos pequeños cambios al NX1. Al igual que en este último, el NX2 identifica a cada procesador con un número entero en el rango de [0,P-1] y cada procesador puede tener varios procesos ejecutándose simultáneamente. Algunas de las capacidades agregadas a este sistema son:

Funciones de configuración	
numnodes()	Regresa el número de nodos existentes en el sistema
mynode()	Regresa la identificación del nodo del proceso
myhost()	Regresa la identificación del host
setpid()	Utilizado por el host para asignar un pid
mypid()	Regresa el pid del proceso
Funciones síncronas	
esend(type,msg,len,node,pid)	Envía un mensaje al proceso pid en nodo node
eprobe(type)	Espera hasta que exista un mensaje de tipo type
erecv(type,buf,len)	Recibe un mensaje
infocount()	Regresa la longitud del último mensaje
infotype()	Regresa el tipo del último mensaje
infonodet()	Regresa el nodo origen del último mensaje
infopid()	Regresa el proceso origen del último mensaje
Funciones asíncronas	
isend(type,msg,len,node,pid)	Envía un mensaje al proceso pin en nodo node
irecv(type,buf,len)	Recibe un mensaje
iprobe(type)	Averigua si existe un mensaje de tipo type destinado a él
msgwait(mid)	Espera hasta la conclusión del mensaje identificado como mid
msgdone(mid)	Averigua si el mensaje mid ha concluido
msgcancel(mid)	Cancela el mensaje mid
flick()	Retardo
Funciones globales	
gsync()	Sincroniza todos los nodos
gopf(x,len,work,f)	Operación global $f(x, work)$
gdsum(x,len,work)	Suma global de los elementos $x(len)$

Tabla 2.5: Funciones del NX2

Identificadores de mensaje. NX2 introduce el concepto de Identificador de Mensaje (MID) como un entero asociado a una comunicación de no bloqueo incompleta. Tres operaciones fueron definidas para el procesamiento de estos mensajes utilizando el MID: a) *msgdone* reporta cuando la operación ha sido completada, b) *msgwait* regresa al flujo del programa hasta que la operación ha sido completada y c) *msgcancel* cancela la operación si ésta no ha sido completada. Estas operaciones permiten un estilo de programación en el cual los procesos que envían un mensaje realizan la operación inmediatamente, mientras que los procesos receptores posponen la recepción hasta que están completamente listos para usar los datos ha recibir, realizando alguna otra operación o cálculo mientras tanto. Con esto es posible traslapar procesos de comunicación con procesos de cálculo.

Funciones de información de procesos de recepción. En la rececpción de mensajes, la información de la fuente del mensaje, tal como son el número de nodo, el pid del proceso que envío el mensaje, el tipo del mensaje y su longitud no son obtenidas por las llamadas de recepción como sucede en el NX1. En cambio, se han incluido 4 rutinas para este propósito. Una llamada a estas rutinas proporciona información del proceso de comunicación concluído más recientemente.

Argumentos comodines. NX2 incluye también la capacidad de tener argumentos comodines en las rutinas de comunicación.

Operaciones globales. NX2 soporta un conjunto muy completo de operaciones globales tales como sincronización y operaciones aritméticas con datos pertenecientes a distintos procesos. Estas operaciones son cuidadosamente codificadas utilizando mapeos óptimos de árboles binarios a la arquitectura[8].

Máquinas Paralelas nCUBE

La serie de computadoras paralelas nCUBE (nCUBE1 y nCUBE2) están basadas en un hipercubo de 13 dimensiones. Cada nodo tiene 14 canales de comunicación bidireccionales interconstruídos (13 para los nodos vecinos y 1 para el canal de entrada/salida). Los procesadores de las máquinas nCUBE son escalares y la sencillez de su diseño ha conducido a un buen rendimiento para aplicaciones no vectorizables. En la nCUBE-2 los mensajes se trasmiten aproximadamente a 2.75 MB/sec y el tiempo de inicialización del mensaje es aproximadamente de 150 μs.

El nCUBE Parallel Software Environment (PSE) provee un conjunto de primitivas de comunicación semejantes a las del Intel NX. Estas rutinas de comunicación utilizan un identificador de proceso que combina tanto el identificador del procesador como el número de proceso en un entero de 32 bits. Las rutinas principales

son *nwrite* y *nread*. La función nwrite es siempre una función de bloqueo, mientras que la función nread es de no bloqueo. La rutina *ntest* permite a un proceso receptor verificar la existencia de un mensaje mandado hacia él, así que nread solo se utiliza cuando se está seguro de que un mensaje será recibido. PSE soporta también rutinas de comunicación globales, además de cálculos basados en mallas. En este caso es posible distribuir los datos como arreglos multidimensionales, y PSE provee rutinas explícitas para el intercambio entre fronteras de sub-arreglos, tal como lo requieren muchos algoritmos[8].

IBM EUI

El sistema IBM EUI (External User Interface) es el ambiente de intercambio de mensajes para la serie de computadoras IBM SP. EUI también fue diseñado para ejecutarse en conjuntos de estaciones de trabajo acopladas (i.e. del mismo tipo) tales como las RS-6000. EUI soporta comunicaciones de bloqueo y no bloqueo y las formas usuales de comunicación globales. El número de tareas especificadas en un programa paralelo es fijo. Un aspecto interesante de este sistema es que soporta la creación dinámica de grupos de tareas. Un grupo es un subconjunto de las tareas existentes en la partición del usuario, conocida como *allgrp*. Los grupos son creados especificando la lista de tareas que lo conformarán a la rutina *mp_group*, o partiendo otro grupo en varias partes disconexas mediante la rutina *mp_partition*. Las rutinas de comunicación global trabajan dentro de estos grupos y no en el conjunto completo de tareas. Las aplicaciones típicas de grupos incluyen crear grupos entre las columnas y los renglones en algoritmos basadas en mallas. Estas rutinas colectivas incluyen barreras, corrimientos, difusiones, etc. Las barreras pueden ser de bloqueo o no bloqueo.

Los mensajes en EUI está identificados por un tipo definido por el usuario y pueden ser seleccionados para la recepción ya sea por la fuente o por el tipo, permitiéndose valores comodines para ambos. Las capacidades disponibles en este ambiente son muy similares a las del NX2. Se espera que en el futuro también soporte un concepto de canal similar al del sistema CROS de Caltech. Un canal conectando un par específico de procesos proveerá una comunicación muy eficiente entre éstos[8].

Sistema Meiko CS

El sistema operativo Computing Surface (CS) de Meiko fue desarrollado para la linea de máquinas paralelas basadas en procesadores i860 y Sparc de Meiko. Estos sistemas son típicos de intercambio de mensajes y la CS-2 es una de las computadoras paralelas más poderosas de la actualidad.

Las rutinas de comunicación del CS (ver Tabla 2.6) difieren drásticamente de

Funciones de configuración	
esgetinfo(numnodes,mynode,mypid)	Regresa el número de nodos y la identificación del nodo y el proceso
Transport	Objeto de comunicación, accesible mediante un netid global
csnopen(index,tr)	Cierra un transporte
csnregname(tr,name)	Establece un nombre global para el transporte tr
csnderegname(tr)	Borra todos los nombres globales asociados al transporte tr
csnlookupname(netid,name,block)	Regresa el netid del transporte name
csngetid(tr)	Regresa el netid del transporte tr
csngetnode(netid)	Regresa el nodo asociado a un netid
csngetnet(netid)	Regresa el número de red asociado a un netid
csnettransport(netid)	Regresa el transporte asociado a un netid
Funciones síncronas	
csntx(tr,netid,flag,msg,len)	Envía un mensaje (bloqueo) al transporte netid
csnrx(tr,netid,buf,len)	Recibe un mensaje (bloqueo) y regresa el netid
Funciones asíncronas	
csntxb(tr,netid,flag,msg,len,mid)	Envía un mensaje (no bloqueo) al transporte netid
csnrxnb(tr,buf,len,mid)	Recibe un mensaje (no bloqueo) y asigna el mid
csntes(t,,,9mid,,,)	Verifica si el mensaje mid ha sido completado
csncancel(tr,,,mid)	Cancela el mensaje mid

Tabla 2.6: Funciones del Meiko CS

otros sistemas debido a que estan basadas en un espacio de nombres. Los procesos crean objetos llamados *transportes*. Los transportes son útiles solo cuando son registrados con un nombre global que es implementado por la función *csnregname*. Una vez que un transporte ha sido registrado otros procesos que conocen su nombre pueden buscarlo y si tienen éxito, obtienen un *netid* para usarlo en futuras comunicaciones. Como en el caso de Intel, este sistema soporta comunicaciones de bloqueo, de no bloqueo, síncronas y asíncronas. Todas éstas son desarrolladas a través de un transporte previamente abierto, el cual espera ser conectado a otros procesos[8].

SUPRENUM

La computadora alemana SUPRENUM es un sistema de 256 nodos basado en un diseño jerárquico de 2 niveles. En el nivel inferior, 16 procesadores conectados mediante un bus forman un grupo. El sistema completo consiste en 16 grupos conectados por una malla de buses horizontales y verticales (ver Fig 2.3). A pesar de esta aparente complejidad, los conceptos de comunicación en la SUPRENUM son bastante similares a los de otros sistemas. Un aspecto poco usual es que este sistema soporta extensiones del Fortran para el control de tareas y para asistir en operaciones de comunicación. Por ejemplo, SUPRENUM utiliza extensiones del lenguaje similares a las listas de entrada/salida de Fortran para llamadas a

funciones de comunicación. Esto permite a los compiladores la optimización de comunicaciones. La desventaja es que los programas realizados con estas extensiones no son portátiles.

SUPRENUM es el único sistema que provee una sofisticada biblioteca de alto nivel como interface al sistema de comunicación. Esta biblioteca soporta un conjunto de operaciones orientada a mallas de 2 y 3 dimensiones. Además de estas poderosas herramientas de programación, estos sistemas permiten la posibilidad de una sustancial portabilidad de los programas entre arquitecturas que soportan este conjunto de primitivas[8].

Thinking Machines CMMD

La máquina Connection Machine de la empresa Thinking Machines soporta dos modelos distintos de programación. El más simple y elegante es el CMF (Connection Machine Fortran), el cual provee al sistema con un estilo de programación SIMD. Sin embargo, para programación estilo MIMD es necesario escribir programas utilizando intercambio de mensajes. El ambiente de intercambio de mensajes en la CM-5 es conocido como CMMD. Una complicación adicional en la programación se debe a la complejidad de los nodos, cada uno de los cuales contiene un procesador escalar Sparc y cuatro procesadores vectoriales. Para utilizar estos últimos, los programas deben ser escritos en un lenguaje vectorial. Además de las operaciones de comunicación estandar send/recv, CMMD soporta un segundo modelo de comunicación basado en el concepto de canal. En este caso, una vez que un canal ha sido creado, las operaciones de comunicación subsecuentes sobre ese canal tendrán una baja sobrecarga. CMMD soporta también las capacidades de comunicación familiares en otros sistemas (como son bloqueo-no bloqueo, síncrono-asíncrono, etc). El hardware de esta máquina tiene canales separados para comunicaciones punto a punto y para comunicaciones colectivas.

La empresa Thinking Machines ha sido muy efectiva para reducir el tiempo de latencia de un mensaje. La clave ha sido implementar el sistema CMMD en términos de un mecanismo simple de comunicación con baja sobrecarga llamado Active Message Layer (AML). Un mensaje AML es un mensaje que consiste en un apuntador a una función, junto con un conjunto de argumentos[8].

2.5.2 Ambientes Portátiles

Los ambientes portátiles son ambientes de intercambio de mensajes desarrollados con el objetivo de utilizar el mismo ambiente en varias arquitecturas paralelas diferentes. Las arquitecturas heterogéneas (en donde computadoras de 2 o más tipos diferentes trabajan en forma conjunta para desarrollar una tarea de cómputo) han obtenido una importancia creciente debido a la necesidad de utilizar todos los

recursos disponibles. Como los ambientes portátiles funcionan en diferentes arquitecturas, automáticamente tienden a ser utilizadas en arquitecturas heterogéneas. A continuación haremos una breve descripción de algunos de estos ambientes.

RPROC

El sistema RPROC, desarrollado por McBryan en 1982, anticipó muchos de los aspectos de los sistemas heterogéneos de intercambio de mensajes, particularmente su uso en arquitecturas heterogéneas y los conceptos de mensajes activos, paquetes de datos mezclados y conversión automática de datos. El sistema fue diseñado para interconectar computadoras con sistemas operativos distintos. Como además muchos sistemas importantes (como CRAY) no soportaban los protocolos TCP/IP, el sistema RPROC fue diseñado suponiendo sólo las mínimas capacidades de comunicación. La única suposición hecha en este sistema fue la existencia de un mecanismo de transferencia de archivos entre los sistemas a interconectar. Todos los mensajes RPROC eran entregados como un par de archivos transferidos. El primer archivo entregaba el mensaje, mientras que el segundo, llamado el archivo de señal, indicaba que el archivo de mensaje había sido completado. El programa receptor no podía considerar que el mensaje estaba terminado hasta recibir el archivo de señal. Estos aspectos de la transferencia de archivos eran invisibles al usuario y podían ser reemplazados por un protocolo diferente de transferencia de archivos. Los métodos usados típicamente incluian rcp, ftp y decnet.

Los mensajes RPROC eran empaquetados en una forma similar a la propuesta para el MPI, que será discutido posteriormente. Este empaquetamiento era necesario por la sobrecarga impuesta por el acceso a los archivos. Los mensajes consistían de una cabecera seguida por uno o mas arreglos de datos, cada uno precedido por un contador. La cabecera proveía información concerniente al proceso fuente de los datos, lo que implicaba el tipo de máquina que los enviaba. La creación, el empaquetamiento y la trasmisión de los mensajes eran operaciones separadas lógicamente. Una vez creada la cabecera de un mensaje, este podía empaquetar varios datos en sucesión mediante llamadas a rutinas de empaquetamiento definidas por el usuario. Una vez completado este proceso, el mensaje era enviado. En el extremo receptor, una rutina localizaba y aceptaba el mensaje utilizando otra rutina, proporcionada por el usuario, para decodificarlo.

Para empaquetar los mensajes, se proporcionó un conjunto de rutinas que leían y escribían arreglos de datos de distintos tipos; éstas utilizaban la cabecera del mensaje para determinar si alguna conversión era requerida; dicha conversión no siempre era realizada en la máquina receptora, dependiendo esta decisión de que máquina podía realizar esta tarea de forma más rápida.

Los mensajes en RPROC eran mensajes activos; cada uno especificaba en su cabecera cual era la rutina a ejecutar en el proceso receptor. Estos mensajes

también eran asíncronos, tanto de bloqueo como de no bloqueo. RPROC fue implementado sobre 10 arquitecturas diferentes y fue usado extensivamente en muchos sitios para construir aplicaciones en máquinas heterogéneas[8].

Macros P4

El sistema P4, creado en el Laboratorio Nacional de Argonne de los Estados Unidos, fue el primer intento de desarrollo de un ambiente portátil. El sistema original, conocido como MonMacs, fue un conjunto de macros del preprocesador M4 para la computadora paralela de memoria compartida HEP. Diseños posteriores agregaron soporte para otros sistemas de memoria compartida, intercambio de mensajes y por último soporte para sistemas mezclados, tales como conjuntos de máquinas de memoria compartida.

P4 destaca entre otros ambientes pórtatiles de intercambio de mensajes por su tratamiento tanto para sistemas de memoria compartida como de memoria distribuída. Es también altamente eficiente, debido a que evita introducir sobrecargas en llamadas a funciones, a través del uso de macros. P4 provee también capacidades interconstruídas para la generación de procesos usando un archivo "procgroup". Una restricción notable de este ambiente es que el intercambio de mensajes es completamente de bloqueo[8].

PARMACS

PARMACS es un ambiente de intercambio de mensajes basado en macros, desarrollado inicialmente a partir de los macros de P4. La ejecución de un programa utilizando este ambiente comienza con un proceso anfitrión que puede crear procesos en otros nodos utilizando la macro *remote_create()*, la cual lee un archivo de entrada que especifica los programas a ejecutar en los nodos y el pid asignado a cada uno de ellos. PARMACS soporta rutinas de envío tanto síncronas como asíncronas y los mensajes pueden ser identificados tanto por el tipo asignado como por el identificador de la fuente. PARMACS soporta cómputo heterogéneo y la macro *msg_format* se utiliza antes del envío de un mensaje para especificar el contenido de un buffer; esta información es utilizada por el proceso receptor para la conversión automática de los datos si es que ésta es necesaria.

Uno de los aspectos más interesantes de PARMACS es el soporte para topologías determinadas. La macro *torus* mapea procesos en anillos y mallas de 2 ó 3 dimensiones, mientras que la macro *graph* mapea procesos en configuraciones arbitrarias definidas por un grafo. La macro torus crea el archivo a utilizar por la macro remote_crate, lo cual permite optimizar un mapeo topológico hacia el hardware disponible. Debido a ésto se proveen también macros especiales para que los procesos puedan localizar las coordenadas de su posición en una malla o en un grafo.

PARMACS influenció en forma importante los aspectos topológicos provistos en el MPI[8].

EXPRESS

El sistema EXPRESS se derivó directamente del sistema CROS de Caltech. EXPRESS es un producto de Parasoft. Inicialmente fue orientado a implementarse en un amplio rango de arquitecturas, con el objetivo de obtener el más alto rendimiento para cada una de ellas, lo cual fue obtenido con un éxito significativo, reduciendo a menudo la latencia del mensaje por factores de cuatro o mas. Más recientemente, Parasoft se ha enfatizado en la fácilidad de uso y ha intentado ocultar muchos de los detalles inherentes en EXPRESS. Esto ha conducido al desarrollo de bibliotecas de mapeo y comunicación para anillos, mallas, toros, etc., cada una optimizada para una plataforma específica. EXPRESS también ha empezado a superar el problema de operaciones de entrada/salida paralelas asi como el balanceo dinámico de carga de trabajo, dos aspectos universalmente ignorados en la mayoría de los ambientes de intercambio de mensajes, incluyendo al MPI[8].

PVM

PVM (Parallel Virtual Machine) representa un ejemplo bastante exitoso entre los ambientes de intercambio de mensajes para cómputo heterogéneo. Desarrollado en el Laboratorio Nacional de Oak Ridge de los Estados Unidos, utiliza una biblioteca simple de envío/recepción para controlar la interacción de un número arbitrario de computadoras. Las primeras implementaciones de PVM utilizaban sockets TCP/IP para todas las comunicaciones, sin embargo, las implementaciones más recientes proveen formas más eficientes de comunicación dentro de máquinas paralelas. PVM es quizá el ambiente de intercambio de mensajes más utilizado debido a su generalidad y aplicabilidad a redes de estaciones de trabajo; difiere de la mayoría de los ambientes de intercambio de mensajes en que soporta la creación dinámica de procesos, utiliza constructores fuertemente tipificados para la construcción de los buffers que almacenan el contenido de los mensajes y es considerablemente pequeño. A pesar de que su simplicidad es una clave del actual éxito de PVM, suministra rutinas para registros de una colección de procesos cooperativos, para iniciar y terminar tareas, para sincronizarse con otras tareas de PVM y para obtener información de configuración. Soporta las operaciones de sincronización entre procesos, envíos de bloqueo y no bloqueo y recepción de no bloqueo. Los mensajes pueden ser seleccionados ya sea por la fuente o por su etiqueta, mientras que las barreras y otras operaciones colectivas usan el nombre del grupo de procesos como identificador[8].

Zipcode

Zipcode es un ambiente portátil que ha sido usado como un laboratorio para el intercambio de mensajes. Durante su desarrollo muchos conceptos nuevos han sido probados y el resultado ha tenido una influencia considerable en el diseño del MPI (ver última sección). Zipcode hizo un esfuerzo especial en superar el problema de proveer un ambiente de desarrollo para bibliotecas paralelas. En general los ambientes de intercambio de mensajes no son adecuados para esta tarea debido al peligro de confusión entre mensajes de una biblioteca y mensajes del programa usuario. Esto se debe a que no existe ningún mecanismo que pueda restringir el tipo de mensajes que el programa usuario puede recibir. Debido a esto, el programa usuario puede interceptar mensajes de la biblioteca, introduciendo errores en su funcionamiento.

Zipcode superó estos problemas utilizando un espacio de comunicación separado para las bibliotecas y suministrando operaciones colectivas que funcionan en solo un subconjunto de todos los procesos. Para implementar estos aspectos introdujo los conceptos de grupos, rangos relativos y comunicaciones colectivas definidas dentro de los grupos. Los grupos (o contextos de comunicación) se forman por un conjunto de procesos, los cuales tienen un rango (i.e. identificador) que es válido solo dentro del grupo. El concepto de grupo separa universos de mensajes, ya que procesos de un grupo no pueden recibir mensajes de procesos que pertenecen a otro grupo[8].

Linda

Linda es un sistema de comunicación que se encuentra en una categoría diferente a la de los sistemas de intercambio de mensajes que hemos discutido anteriormente. Linda es un sistema de memoria virtual compartida. Esta memoria es llamada espacio de "tuplas" (i.e. estructuras de datos). Las operaciones de Linda aplicadas a este espacio proveen las funciones de administración, sincronización y comunicación requeridas para una programación MIMD. Los objetos en este lenguaje son conocidos como tuplas y el ambiente provee cuatro operaciones básicas sobre éstos: *out, eval, in, rd.* Out genera una tupla de manera serial, eval genera una tupla asíncronamente (que es usada para crear tuplas en paralelo) , rd lee una tupla e in lee y destruye una tupla. El operador in proporciona una forma para encontrar una tupla determinada en el espacio de tuplas.

Las implementaciones más interesantes de Linda son extensiones a los estándares de C y Fortran (C-Linda y Fortran-Linda), las cuales permiten a programas de memoria compartida ejecutarse en máquinas de memoria distribuída. Todo el intercambio de mensajes necesario para su ejecución es generado por el traductor de Linda. Linda es apropiado para problemas que involucran muchos procesos ejecutándose en un mismo nodo, comunicaciones no bien definidas, comunicación

global y asíncrona extensiva. Algunos estudios han mostrado que Linda también puede competir con otros ambientes de intercambio de mensajes incluso en aplicaciones típicamente numéricas[8].

MPI

MPI es un estándar de ambiente de intercambio de mensajes producto de una serie de encuentros sostenidos entre noviembre de 1992 y enero de 1994 por el Comité MPI. Este comité está constituído por miembros de aproximadamente 40 instituciones incluyendo la mayoría de los fabricantes de máquinas paralelas, así como universidades y laboratorios gubernamentales alrededor del mundo, que están involucrados en el cómputo paralelo.

Se ha intentado que el MPI comprenda la mayoría de los aspectos de todos los sistemas nativos, siendo al mismo tiempo suficientemente eficiente para permitir a los fabricantes proveer implementaciones nativas de MPI en sus arquitecturas en lugar de sus sistemas actuales.

MPI está basado en aspectos de todos los ambientes que hemos descrito anteriormente; generaliza el concepto de buffer del mensaje permitiendo al usuario utilizar tipos elementales de datos, arreglos contiguos y estructuras generales. Además, generaliza el concepto de etiqueta o tipo de mensaje introduciendo un campo adicional en la sintaxis de las operaciones de comunicación; este campo representa el contexto en el cual el proceso está contenido y su objetivo es definir familias de mensajes. MPI provee grupos de procesos, así como rutinas para la administración de estos grupos. Todas las comunicaciones ocurren dentro de *comunicadores*, los cuales son creados a partir de grupos de procesos y contextos de comunicación. Tanto la fuente como el destinatario en rutinas de envío/recepción son números enteros que representan un rango dentro de un comunicador. MPI soporta también aplicaciones orientadas a topologías, y tiene interconstruído soporte para mallas y grafos. Algunos de los aspectos que MPI no provee son: mecanismos para administración de procesos, transferencias remotas de memoria, mensajes activos y hebras de memoria compartida virtual[8],[10].

En el siguiente capítulo se hará una descripción de los aspectos básicos de este ambiente pórtatil de intercambio de mensajes.

Capítulo 3

AMBIENTE DE INTERCAMBIO DE MENSAJES MPI: UNA INTRODUCCIÓN

3.1 INTRODUCCIÓN

En este capítulo se hace una breve descripción de los principales aspectos del ambiente portátil de intercambio de mensajes MPI (Message Passing Interface). Esta descripción abarcará las principales funciones del MPI, incluyendo comunicaciones punto a punto, operaciones colectivas y creación de grupos de procesos.

Como se mencionó en el capítulo anterior, en el MPI se han implementado los aspectos más atractivos de la mayoría de los ambientes de intercambio de mensajes existentes, tanto nativos como portátiles, siendo influenciado principalmente por los sistemas Intel NX/2, Express, PARMACS, Zipcode y PVM[8].

La creación del MPI involucró a alrededor de 40 organizaciones, principalmente de Estados Unidos y Europa. Este grupo fue formado por la gran mayoría de los fabricantes de máquinas paralelas, así como por investigadores de universidades y laboratorios gubernamentales[9]. El principal objetivo del desarrollo del MPI fue la creación de un estándar. Este proceso se inició con el Taller sobre Estándares para Intercambio de Mensajes en Ambientes de Memoria Distribuída, patrocinado por el Centro para la Investigación en Cómputo Paralelo, en Williamsburg, Virginia, en abril de 1992[10].

El MPI se ha convertido en uno de los ambientes de intercambio de mensajes más utilizados para la programación de aplicaciones paralelas, tanto en máquinas que contienen muchos procesadores como en conjuntos de estaciones de trabajo conectadas en red. Algunas de las razones de esta popularidad son las siguientes[11]

- Existen varias implementaciones gratuitas disponibles, como son MPICH (Laboratorio Nacional de Argonne), LAM (Centro de Supercómputo de Ohio) y CHIMP (Centro de Cómputo Paralelo de Edimburgo), entre otras.
- Es posible realizar comunicaciones completamente asíncronas, lo que permite traslapar procesos de cálculo con procesos de comunicación.
- Los grupos de procesos son sólidos y eficientes; éstos son creados a partir de operaciones colectivas y la información de los miembros que conforman un grupo es accesible a todos los miembros del mismo.
- Maneja eficientemente los buffers de los mensajes; éstos son enviados y recibidos por medio de estructuras de datos definidas por el usuario y no usando estructuras definidas internamente por las bibliotecas de comunicación.
- Permite la programación eficiente tanto de máquinas paralelas como de grupos de estaciones de trabajo conectadas en red.
- Es totalmente portátil. El código de un programa MPI puede ser compilado y ejecutado en cualquier arquitectura sin necesidad de modificación alguna.
- Es un estándar. Un programa escrito en una implementación dada del MPI debe funcionar bajo cualquier otra implementación sin ninguna modificación.
- Está formalmente especificado. Existe un documento oficial que contiene todas las características que debe contener una implementación estándar.

En el MPI se han incluído los siguientes aspectos importantes, deseables en todo ambiente de cómputo paralelo[10]:

- Comunicaciones punto a punto.
- Operaciones colectivas.
- Grupos de procesos.
- Contextos de comunicación.
- Formación de topologías.
- Enlaces con los lenguajes Fortran 77 y C.

3.2 SEMÁNTICA

A continuación describiremos el significado de algunos términos así como las convenciones necesarias para la descripción de las funciones y operaciones del MPI. Se presentarán los tipos de argumentos y su clasificación; los términos que describen

el comportamiento de las operaciones de comunicación, los tipos de datos que incluye el MPI, los conceptos de rango, grupo, contexto y comunicador así como los enlaces con los lenguajes C y Fortran[10].

3.2.1 Tipos de Argumentos

Los argumentos en las llamadas a funciones MPI se clasifican en tres tipos: IN, OUT o INOUT, cuya semántica es la siguiente:

- IN: Un argumento IN no es alterado por la función, ésta solo toma su valor y lo utiliza internamente.
- OUT: Los argumentos OUT son utilizados por las funciones para colocar en ellos valores que son resultado de la operación de dichas funciones. Por lo tanto, el valor de este tipo de argumento es alterado después de la ejecución de la función.
- INOUT: Estos argumentos son utilizados tanto para proporcionar valores a las funciones (IN) como para que éstas modifiquen su valor de acuerdo al resultado de alguna operación (OUT).

Esta clasificación de argumentos está basada en la manera en que una función particular utiliza un argumento dado y no en el tipo de dato de la variable usada como argumento y es análoga a la distinción que hace el lenguaje C entre parámetros trasmitidos a una función por referencia o por valor (en el lenguaje Fortran, todos los parámetros son trasmitidos a las funciones y subrutinas por referencia).

3.2.2 Rangos, Grupos, Etiquetas, Contextos y Comunicadores

En la descripción de los ambientes de intercambio de mensajes realizada en el capítulo anterior, se mencionaron varias formas de identificación de los procesos que participan en un cómputo paralelo. El modo en que el MPI identifica a los diversos procesos consiste en asignar a cada uno de éstos un número entero diferente. Dicho número se conoce como rango. Debido a que el rango permite la distinción entre procesos diferentes, es utilizado en las operaciones de comunicación para determinar el proceso que envía un mensaje y el proceso que debe recibirlo. Por otra parte, debido a que un proceso dado puede mandar dos o más mensajes diferentes a un mismo proceso receptor, es necesario utilizar un mecanismo que permita a este último distinguir los distintos mensajes que recibe. Esto se logra mediante el uso de etiquetas en los mensajes. Las etiquetas son números enteros escogidos arbitraria pero cuidadosamente por el usuario, con el objeto de distinguir los diversos mensajes de un programa.

En la operación de recepción del MPI pueden especificarse valores comodines tanto para el rango del proceso origen del mensaje como para la etiqueta de éste. Debido a esta característica, un proceso podría interceptar mensajes interfiriendo con la lógica del programa; esta situación puede causar errores, principalmente cuando se usan bibliotecas paralelas, en las cuales grupos de procesos intercambian información para realizar cierta tarea. Para evitar estos errores es posible crear canales de comunicación específicos llamados *comunicadores* usando dos conceptos: *grupo* y *contexto*. Un grupo es un conjunto de procesos que trabajan concurrentemente para alcanzar un determinado objetivo. El concepto de contexto es usado para separar los objetivos para los cuales los diversos procesos realizan las tareas encomendadas. Los comunicadores combinan estos dos conceptos y son usados por el MPI para, a petición explícita del usuario, crear grupos de procesos independientes de acuerdo a los diferentes contextos. Los comunicadores no pueden especificarse mediante valores comodines. Estas condiciones permiten que únicamente los procesos pertenecientes al grupo del comunicador puedan recibir los mensajes enviados dentro de ese contexto.

3.2.3 Tipos de Funciones y Operaciones

Las funciones del MPI se clasifican dado su comportamiento en funciones de **bloqueo** y de **no bloqueo**. Se dice que una función es de bloqueo si no permite la continuación del flujo del programa hasta que se haya cumplido el objetivo de dicha función. Similarmente, una función es de no bloqueo si el flujo del programa puede continuar aún cuando la función no ha cumplido completamente su objetivo. Por otra parte, se dice que una operación es **local** si no requiere que otro proceso realice alguna operación para poder ser completada, mientras que una operación es **no local** si requiere de la intervención de otro proceso para completarse. Por último, se dice que una operación es **colectiva** si requiere de la intervención de todos los procesos pertenecientes a un grupo para completarse.

3.2.4 Tipos de Datos

El MPI maneja las siguientes estructuras:

Objetos opacos y manijas

Un programa MPI maneja 2 tipos de memoria: la *memoria asignada al usuario* y la *memoria asignada al sistema*. La memoria del sistema es usada como buffer para el envío de mensajes y para el almacenamiento de diversos *objetos* que representan distintas entidades tales como grupos, contextos, tipos de datos, etc. Ya que la memoria del sistema no es directamente accesible al usuario se dice que los

objetos almacenados en ésta son *opacos*: su tamaño y forma no son visibles. Los objetos opacos son accesibles a través del uso de parámetros llamados *manijas*, las cuales son definidas en la memoria del usuario. Por lo tanto, las funciones de MPI que operan sobre objetos opacos requieren argumentos de tipo manija para poder accesar a tales objetos. Las manijas son variables declaradas por el usuario; para el caso de Fortran son del tipo INTEGER, mientras que para C el tipo de variable a utilizar como manija varía para los distintos tipos de objetos.

Los objetos opacos son alojados y desalojados de la memoria del sistema mediante funciones que son específicas a cada objeto. Las funciones que alojan objetos requieren un argumento OUT de tipo manija para colocar en él una referencia válida al objeto alojado, mientras que una función que desaloja el objeto requiere un argumento INOUT que indique la referencia al objeto y en el cual se coloca el valor de una "manija vacía" (NULL). Una función que desaloja un objeto invalida su manija y marca dicho objeto para su desalojo. Debido a que la manija ha sido invalidada, el objeto no es accesible para el usuario después de realizada dicha función, sin embargo, éste no es desalojado inmediatamente si existe alguna operación pendiente que lo involucre. En tal caso, el objeto será desalojado hasta que dicha operación haya terminado.

Constantes

MPI define un conjunto de valores constantes utilizados por algunas funciones. Estas constantes tienen un significado especial, por ejemplo, MPI_ANY_TAG es un valor utilizado como comodín para la etiqueta (tag) de un mensaje. MPI define también algunas constantes como valores válidos para manijas como es el caso de MPI_COMM_WORLD, el cual es un valor de manija utilizado para acceder a un comunicador que engloba a todos los procesos existentes al inicio del programa y que permite la comunicación con cualquiera de ellos.

Tipo de dato "choice"

Las funciones de MPI algunas veces utilizan argumentos con un tipo de dato *choice*, lo que indica que distintas llamadas a una de estas funciones pueden tener argumentos de distintos tipos. Por ejemplo, en el envío de mensajes se utiliza una misma función para trasmitir datos de diferentes tipos, tales como enteros, reales, complejos, etc. El usuario debe, sin embargo, especificar el tipo de dato transmitido.

3.2.5 Enlaces con Fortran 77

En Fortran 77, todos los nombres de las funciones y constantes MPI tienen el prefijo MPI_. Con objeto de evitar errores en la definición de variables y funciones, no deben declararse nombres de variables que empiecen con este prefijo. Todas las subrutinas de MPI regresan un código en su último parámetro indicando si la operación fue realizada exitosamente. Todas las declaraciones predefinidas por el MPI deben suministrarse a través de un archivo el cual generalmente es llamado *mpif.h*. A menos que se mencione explícitamente lo contrario, todos los aspectos del MPI referentes al Fortran 77 son consistentes con el estándar ANSI Fortran 77

3.2.6 Enlaces con el lenguaje C

En C, todos los nombres de las funciones MPI comienzan con el prefijo MPI_ seguidas por una letra mayúscula, mientras que todas las constantes definidas son nombres con letras mayúsculas. Como en el caso del Fortran, debe evitarse la declaración de variables o funciones cuyos nombres comiencen con el prefijo MPI_.

La definición de constantes, los prototipos de las funciones, y la definición de nuevos tipos deben suministrarse mediante la inclusión de un archivo denominado *mpi.h*. La mayoría de las funciones retornan un código que indica si la función finalizó exitosamente o no.

3.3 COMUNICACIONES PUNTO A PUNTO

Entendemos por comunicación punto a punto el proceso de envío y recepción de un mensaje realizado entre dos procesos diferentes. En la Sec. 2.4 se mencionó que las dos funciones básicas para la construcción de un programa paralelo son la sincronización de procesos y el intercambio de información entre éstos. También se hizo énfasis en que el intercambio de mensajes provee una fácil y eficiente implementación de estas funciones. Es por esto que las comunicaciones punto a punto constituyen los procesos básicos de comunicación en el MPI.

Un mensaje está definido como un conjunto de datos almacenados en la memoria perteneciente a un proceso, que son enviados y copiados en la memoria de otro proceso. El contenido de un mensaje se define a partir de 3 características: la dirección de memoria en donde se encuentra el primer dato, el número de datos que contendrá el mensaje y el tipo de éstos; por otra parte, en un mensaje se deben especificar los datos necesarios para identificar al proceso receptor y para permitir su recepción selectiva, estos últimos son conocidos en su conjunto como el *sobre* del mensaje y son: el proceso originario del mensaje, el proceso destinatario, la etiqueta del mensaje y el comunicador dentro del cual es enviado. La

especificación del proceso origen de los datos y la del proceso receptor permiten el establecimiento de la comunicación entre dos procesos; como vimos anteriormente, las etiquetas permiten distinguir entre dos mensajes enviados por un proceso dado a un mismo proceso receptor, mientras que los comunicadores determinan tanto el contexto de comunicación como el grupo de procesos que pueden intervenir en la recepción del mensaje. Esta información es análoga a la que se utiliza, por ejemplo, en los memoranda emitidos por una organización: El proceso destinatario es equivalente al nombre de la persona que debe recibir el memorandum, la etiqueta al asunto particular y el comunicador al departamento dentro del cual circula el memorandum.

Las operaciones que realizan las comunicaciones punto a punto son llamadas *send* y *receive*. Las funciones MPI que realizan estas operaciones pueden ser tanto de bloqueo como de no bloqueo. A continuación haremos una descripción de las funciones que realizan las operaciones *send* y *receive*.

3.3.1 Funciones de Bloqueo

Envío de mensajes

La función de bloqueo básica para el envío de mensajes es MPI_SEND, la cual tiene la siguiente sintaxis:

MPI_SEND(buf, count, datatype, dest, tag, comm)

buf	IN	Dirección del primer dato del mensaje (choice)
count	IN	Número de elementos que contiene el mensaje (entero)
datatype	IN	Tipo de dato de cada elemento del mensaje (manija)
dest	IN	Rango del proceso destino (entero)
tag	IN	Etiqueta del mensaje (entero)
comm	IN	Comunicador a través del cual circula el mensaje (manija)

La descripción del contenido del mensaje (dirección inicial, número de datos y tipo de éstos) se da mediante los parámetros *buf*, *count* y *datatype*, mientras que el sobre del mensaje se especifica mediante *dest*, *tag* y *comm* (la identificación del proceso originario del mensaje está implícito).

Los tipos de datos (datatype) que pueden ser especificados corresponden a los tipos de datos básicos del lenguaje utilizado. Los valores posibles para C y Fortran se dan en la Tabla 3.1. Existen además los tipos de datos MPI_BYTE y MPI_PACKED definidos por el MPI y que no corresponden a tipos de datos de C o de Fortran.

Tipo de dato MPI	Tipo de dato Fortran
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
Tipo de dato MPI	**Tipo de datos C**
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed log int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Tabla 3.1: Equivalencias de tipos de datos

Modos de comunicación para el envío de mensajes

Como se mencionó anteriormente, la función MPI_SEND es de bloqueo, esto es, no permite la continuación del flujo del programa hasta que los datos del mensaje y del sobre hayan sido asegurados, de manera que el proceso que envía los datos tenga la libertad de volver a escribir en el area de almacenamiento sin comprometer la integridad del mensaje. Se puede considerar que los datos del mensaje han sido asegurados en dos situaciones: cuando han sido copiados a la memoria del proceso receptor o cuando han sido copiados en un buffer temporal que reside en la memoria del sistema.

La acción de copiar el mensaje a un buffer desacopla las operaciones de *send* y *receive*, permitiendo que la operación *send* de bloqueo pueda completarse tan pronto como el mensaje haya sido copiado en el buffer, aún cuando la operación de recepción no haya sido ejecutada todavía por el proceso receptor; sin embargo, el uso de buffers en las operaciones *send* tiene un alto costo, tanto por la sobrecarga adicional que representa la acción de copiar el mensaje al buffer como por los mayores requerimientos de memoria para el sistema. Dada la posibilidad de copiar los datos de un mensaje en un buffer temporal se tienen varios modos de comunicación posibles para el envío de mensajes. Estos modos permiten la elección explícita de la forma en que la operación *send* debe realizarse. Los modos de comunicación existentes son: *con buffer*, *síncrono*, *ready* y *estándar*, los cuales serán descritos a continuación.

En el modo de comunicación *con buffer*, la operación *send* inicia e incluso se

completa aún cuando el proceso receptor no haya comenzado la operación de recepción. La función que realiza la operación *send* en este modo (MPI_BSEND) es local, ya que todos los mensajes son copiados a un buffer temporal, independientemente de que el proceso destinatario ejecute o no la operación de recepción. En el caso de que el tamaño del buffer sea insuficiente para copiar un mensaje dado, ocurrirá un error que generalmente produce la finalización anormal del programa.

Una operación *send* en el modo *síncrono* puede iniciarse aún si no existe alguna operación de recepción, sin embargo, finaliza hasta que el proceso de recepción se haya completado, es decir, hasta que el mensaje ha sido copiado en la memoria del proceso receptor. Este modo provee una comunicación síncrona, ya que ambas operaciones (envío y recepción) finalizan simultáneamente. La función que realiza la operación *send* en modo *síncrono* es MPI_SSEND y es, obviamente, una función no local.

En el modo de comunicación *ready*, una operación *send* puede iniciar únicamente si el proceso de recepción ha sido también iniciado, de otra manera, la operación es errónea y su comportamiento es indefinido. Este modo de comunicación funciona de la misma forma que el modo *síncrono*, con la excepción de que evita sobrecargas debidas a la espera del inicio de la operación de recepción. MPI_RSEND es la función que realiza la operación send en este modo y es no local.

La sintaxis de las funciones MPI_BSEND, MPI_SSEND y MPI_RSEND se muestra a continuación:

MPI_BSEND(buf, count, datatype, dest, tag, comm)
MPI_SSEND(buf, count, datatype, dest, tag, comm)
MPI_RSEND(buf, count, datatype, dest, tag, comm)

buf	IN	Dirección del primer dato del mensaje (choice)
count	IN	Número de elementos del mensaje (entero)
datatype	IN	Tipo de dato de cada elemento del mensaje (manija)
dest	IN	Rango del proceso receptor (entero)
tag	IN	Etiqueta del mensaje (entero)
comm	IN	Comunicador a través del cual circula el mensaje (entero)

En el modo *estándar* de comunicación, el MPI decide automáticamente cuales mensajes serán enviados en el modo *buffer* y cuales en el modo *síncrono*. Esta elección dependerá principalmente de la cantidad de memoria disponible. La función MPI_SEND descrita anteriormente realiza la comunicación en este modo y dado que puede emplear el modo *síncrono*, se considera no local.

Recepción de mensajes

A diferencia de la operación de envío para la cual existen varias modalidades, la operación de recepción tiene un solo modo de comunicación. La función de bloqueo que realiza esta operación es MPI_RECV, cuya sintaxis es:

MPI_RECV(buf, count, datatype, source, tag, comm, status)

buf	OUT	Dirección a partir de la cual se almacenará el mensaje (choice)
count	IN	Número de elementos a recibir (entero)
type	IN	Tipo de dato de cada elemento del mensaje (manija)
source	IN	Rango del proceso originario de los datos (entero)
tag	IN	Etiqueta del mensaje (entero)
comm	IN	Comunicador a través del cual circula el mensaje (manija)
status	OUT	Estatus de la operación (status)

Estos argumentos son determinados por el proceso receptor y se utilizan para seleccionar un mensaje para su recepción. El espacio de memoria especificado por *buf* consiste en *count* elementos consecutivos del tipo *datatype*. La longitud del mensaje recibido debe ser menor o igual a la longitud del espacio de memoria definido por *buf*, un error de *overflow* ocurrirá en caso contrario. La selección de un mensaje para su recepción se realiza con base en los parámetros especificados en la función MPI_RECV; un mensaje será recibido solo si los parámetros de su sobre coinciden con los parámetros especificados en esta función.

Como se discutió anteriormente, el proceso receptor puede especificar valores comodines tanto para el origen como para la etiqueta de un mensaje (definidos como MPI_ANY_SOURCE y MPI_ANY_TAG, respectivamente); el origen y la etiqueta del mensaje recibido son desconocidos si se han utilizado estos valores comodines en la función de recepción. La información acerca de estos parámetros, así como el código de error producido por la operación de recepción, son almacenados en el parámetro *status*. Para el lenguaje C, el tipo de variable MPI_Status esta definido como una estructura que contiene 3 campos denominados MPI_SOURCE, MPI_TAG y MPI_ERROR; en el lenguaje Fortran, las variables de status deben ser arreglos del tipo INTEGER y de tamaño MPI_STATUS_SIZE. Las constantes MPI_SOURCE, MPI_TAG y MPI_ERROR son los índices de dichos arreglos correspondientes a los campos de origen, etiqueta del mensaje y el código de error de la operación, respectivamente.

El argumento *status* también contiene información sobre la longitud del mensaje recibido, sin embargo, esta no es accesible directamente como un campo mas de este argumento. La función MPI_GET_COUNT proporciona dicha longitud a partir de los valores de la variable status. La sintaxis de esta función es:

MPI_GET_COUNT(status, datatype, count)

status	IN	Estatus de la operación de recepción (status)
datatype	IN	Tipo de dato recibido (manija)
count	OUT	Número de elementos recibidos (entero)

3.3.2 Funciones de No Bloqueo

El rendimiento en un cálculo paralelo puede ser incrementado traslapando las operaciones de cómputo con las operaciones de comunicación. Esto se consigue utilizando operaciones de comunicación de **no bloqueo**. Una operación *send* o *receive* de no bloqueo inicia el envío o la recepción de un mensaje permitiendo la continuación del flujo del programa aún antes de que esta sea completada. Es necesaria la ejecución de otra función que determina si la operación de comunicación ha concluido. De esta manera, las operaciones de no bloqueo se realizan en dos pasos: inicio y finalización.

Las funciones que inician una operación de comunicación de no bloqueo utilizan un objeto opaco del tipo **request** como identificador de la operación de comunicación que inician. Los objetos *request* almacenan información como: el modo de comunicación (en el caso de que la operación sea el envío de un mensaje), el espacio de memoria que contiene el mensaje, su contexto, la etiqueta del mensaje, el origen o el destino (ya sea el caso de una operación *send* o *receive*), además del estatus de la operación. A continuación describiremos las funciones de no bloqueo que inician las operaciones *send* y *receive* así como las funciones que finalizan dichas operaciones.

Envío de mensajes

El envío de mensajes de no bloqueo puede realizarse en los 4 modos de comunicación disponibles para el envío de bloqueo, teniendo éstos el comportamiento ya descrito. Las funciones de no bloqueo que inician una operación *send* son las siguientes:

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
MPI_IBSEND(buf, count, datatype, dest, tag, comm, request)
MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)
MPI_IRSEND(buf, count, datatype, dest, tag, comm, request)

buf	IN	Dirección del primer dato del mensaje (choice)
count	IN	Número de elementos del mensaje (entero)
datatype	IN	Tipo de dato de cada elemento del mensaje (manija)
dest	IN	Rango del proceso receptor (entero)
tag	IN	Etiqueta del mensaje (entero)
comm	IN	Comunicador a través del cual circula el mensaje (manija)
request	IN	Identificador de la operación de comunicación (manija)

Los argumentos *buf*, *count*, *datatype*, *dest*, *tag* y *comm* tienen el mismo significado que en la función de bloqueo MPI_SEND. Las funciones aqui descritas alojan un objeto tipo *request* y lo asocian con la manija del argumento *request*. Este objeto será utilizado más adelante dentro de las funciones de finalización.

El uso de una función de no bloqueo que inicia la operación *send* indica al sistema que puede empezar a copiar los datos del mensaje, ya sea en la memoria del proceso receptor o en un buffer. El proceso que envía el mensaje no debe modificar los datos de éste hasta que la operación haya sido completada.

Recepción de mensajes

Al igual que en la recepción de bloqueo, la operación de recepción de no bloqueo tiene un solo modo de comunicación, el cual es realizado por la función MPI_IRECV, cuya sintaxis es:

MPI_IRECV(buf, count, datatype, source, tag, comm, request)

buf	OUT	Dirección del buffer que recibirá el mensaje (choice)
count	IN	Número de elementos a recibir (entero)
type	IN	Tipo de dato de cada elemento a recibir (manija)
source	IN	Identificación del proceso originario de los datos (entero)
tag	IN	Etiqueta del mensaje (entero)
comm	IN	Comunicador a través del cual circula el mensaje (manija)
request	IN	Identificador de la operación de comunicación (manija)

Los argumentos *buf*, *count*, *datatype*, *source*, *tag*, *comm* tienen el significado descrito para la función MPI_RECV. La función MPI_IRECV aloja en la memoria del sistema un objeto de tipo *request*, asociado a la manija usada en el argumento *request*. El uso de la función MPI_IRECV indica al sistema que puede iniciar la escritura en el área de memoria señalada por los parámetros *buf*, *count* y *datatype* de la llamada a la función. El proceso receptor no debe modificar dicha memoria mientras la recepción no se haya completado.

Finalización de operaciones de no bloqueo

Las funciones MPI_WAIT y MPI_TEST son utilizadas para determinar si una comunicación iniciada por una función de no bloqueo ha terminado. La finalización de una operación de envío significa que el proceso que envió el mensaje tiene ya la libertad de escribir en el area de memoria en que éste se encuentra almacenado sin comprometer su integridad. La finalización de una operación de recepción significa que el mensaje ha sido completamente copiado a la memoria del proceso receptor y que, por lo tanto, éste puede tener acceso a dicha area de memoria; además, la finalización de una operación de recepción indica también que el status de la operación ha sido obtenido. La sintaxis de la función MPI_WAIT es:

MPI_WAIT(request, status)

request	INOUT	Identificador de la operación de comunicación (manija)
status	OUT	Estatus de la operación de comunicación (Status)

Esta función permite la continuación del flujo del programa hasta que la operación de comunicación identificada por *request* haya terminado. El objeto asociado con *request* es desalojado de la memoria al término de esta función, colocando en la manija *request* el valor nulo definido como MPI_REQUEST_NULL. La variable *status* contiene la información descrita en la sección 3.3.1.

Por otra parte, la sintaxis de la función MPI_TEST es:

MPI_TEST(request, flag, status)

request	INOUT	Identificador de la operación de comunicación (manija)
flag	OUT	Bandera que indica el estado de la operación (lógica)
status	OUT	Estatus de la operación de comunicación (Status)

La función MPI_TEST regresa un valor verdadero en el argumento *flag* si la operación de comunicación asociada con *request* ha terminado, en tal caso, el argumento *status* contendrá el status de dicha operación y el objeto asociado con *request* será desalojado de la memoria del sistema, colocando el valor MPI_REQUEST_NULL en el argumento *request*. En el caso de que la operación no haya sido terminada, la función regresará un valor falso en el argumento *flag*. Esta función es local.

La función MPI_REQUEST_FREE marcará un objeto tipo *request* para su desalojo. Esta función puede utizarse para desalojar objetos asociados a comunicaciones para las cuales no se realiza ninguna llamada a las funciones de finalización. Su sintaxis es:

MPI_REQUEST_FREE(request)

request	INOUT	Objeto *request* a desalojar (Manija)

La llamada a esta función marca el objeto *request* para su desalojo y coloca el

valor MPI_REQUEST_NULL en la manija asociada. Si existe una operación de comunicación pendiente asociada al objeto, esta terminará normalmente y después el objeto será desalojado.

3.4 OPERACIONES COLECTIVAS

Como se ha definido anteriormente, las operaciones colectivas son aquellas que deben ser realizadas por todos los procesos pertenecientes a un grupo dado para poder ser completadas. Esto es, una operación colectiva terminará hasta que todos los procesos miembros del grupo hayan realizado la llamada a la función correspondiente. En general, las operaciones colectivas se pueden dividir en los siguientes tipos:

- **Sincronización de procesos.** Permiten que un grupo de procesos realice las mismas operaciones simultáneamente.
- **Distribución de datos.** Éstas se encargan de la distribución de un conjunto de datos entre los diversos procesos que pertenecen a un grupo.
- **Cálculos colectivos.** Éstas realizan las operaciones de suma, producto, AND, OR, entre otras, con datos distribuídos entre los procesos miembros del grupo.

Obviamente, es posible construir cualquiera de las operaciónes colectivas utilizando las operaciones básicas *send* y *receive*; sin embargo, resulta mas conveniente utilizar las operaciones colectivas definidas en el MPI, tanto porque simplifican la programación como porque estas implementaciones regularmente ya han sido optimizadas. A continuación describiremos tanto las operaciones colectivas definidas por el MPI como las funciones que las realizan.

3.4.1 Barreras

Las barreras son operaciones colectivas que permiten la sincronización de procesos. Cuando un proceso realiza la operación de barrera, el flujo se detiene hasta que el resto de los procesos dentro del grupo ejecuta la operación. La función que realiza esta operación es MPI_BARRIER, cuya sintaxis es:

MPI_BARRIER(comm)

comm IN Comunicador (manija)

El comunicador *comm* indica qué grupo de procesos debe ejecutar la operación de barrera para que ésta pueda concluir.

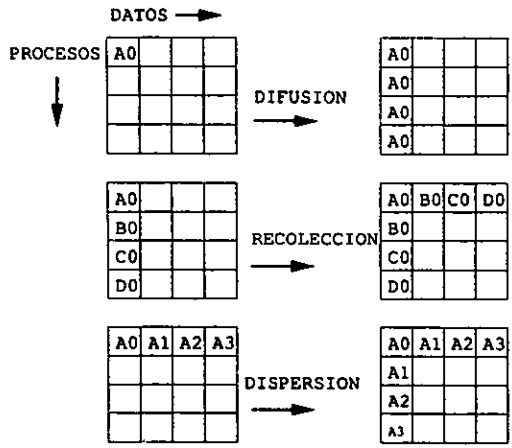

Figura 3.1: Distintas operaciones de distribución de datos

3.4.2 Difusión de Datos

La operación de *difusión* de datos consiste en la propagación de un conjunto de datos pertenecientes a un proceso particular (al cual se le conoce como proceso raíz de la operación) hacia todos los demás procesos del grupo (ver Fig. 3.1). La sintaxis de la función MPI_BCAST, que realiza esta operación, es la siguiente:

MPI_BCAST(buf, count, datatype, root, comm)

buf	INOUT	Dirección de memoria del primer dato (choice)
count	IN	Número de datos del buffer (entero)
datatype	IN	Tipo de dato (manija)
root	IN	Rango del proceso raíz de la operación (entero)
comm	IN	Comunicador (manija)

La dirección de memoria *buf* indica, para el proceso raíz, la dirección de inicio de los datos que van a ser trasmitidos, mientras que para el resto de los procesos indica la dirección de memoria a partir de la cual los datos serán almacenados. Al finalizar la operación, todos los procesos tendrán en la dirección de memoria *buf* los mismos datos.

3.4.3 Recolección de Datos

En un operación de *recolección* todos los procesos envían un conjunto de datos particular al proceso raíz de la operación. Este proceso coloca los datos recibidos en localidades contiguas de memoria, de modo que los datos procedentes del proceso i precedan a los datos procedentes del proceso $i+1$. Esto implica que el espacio de

memoria reservado para recibir los datos debe ser P veces la longitud de los datos enviados por cada proceso (en donde P es el número de procesos que intervienen en la operación) . La sintáxis de la función MPI_GATHER, la cual realiza la operación de *recolección*, es la siguiente:

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

sendbuf	IN	Dirección inicial de los datos a ser enviados (choice)
sendcount	IN	Número de datos a ser enviados (entero)
sendtype	IN	Tipo de dato a ser enviado (manija)
recvbuf	OUT	Dirección de almacenamiento de los datos (choice)
recvcount	IN	Número de datos a ser recibidos (entero)
recvtype	IN	Tipo de dato a ser recibido (manija)
root	IN	Rango del proceso raíz de la operación (entero)
comm	IN	Comunicador de la operación (manija)

Al finalizar la operación de recolección, el proceso *raíz* tendrá en la dirección señalada por *recvbuf* la concatenación del contenido de *sendbuf* de todos los procesos, incluyéndose a sí mismo.

3.4.4 Dispersión de Datos

La operación de *dispersión* de datos es inversa a la operación de *recolección*, esto es, el proceso raíz distribuye un conjunto de datos entre los diversos procesos que participan en la operación, incluyéndose a si mismo. A diferencia de la operación de difusión, en donde todos los procesos reciben exactamente los mismos datos, en la operación de *dispersión* cada proceso recibe una parte diferente del conjunto de datos. La sintaxis de la función MPI_SCATTER, la cual realiza esta operación, es:

MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root,comm)

sendbuf	IN	Dirección de los datos a ser enviados (choice)
sendcount	IN	Número de datos a ser enviados (entero)
sendtype	IN	Tipo de dato a ser enviado (manija)
recvbuf	OUT	Dirección de almacenamiento del mensaje (choice)
recvcount	IN	Número de datos a ser recibidos (entero)
recvtype	IN	Tipo de dato a ser recibido (manija)
root	IN	Rango del proceso raízz de la operación (entero)
comm	IN	Comunicador de la operación (manija)

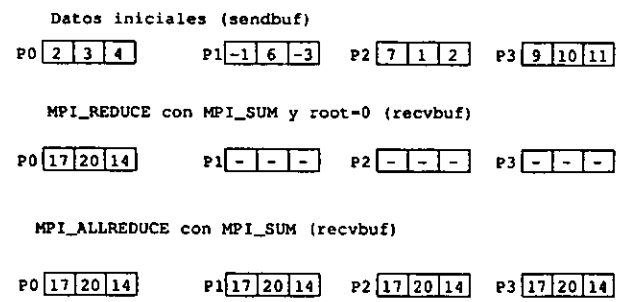

Figura 3.2: Esquema del funcionamiento de las operaciones de cálculos colectivos

3.4.5 Cálculos Colectivos

Las operaciones de cálculos colectivos realizan, entre otras, las operaciones aritméticas suma y producto, además de las operaciones logicas AND, OR y XOR, entre conjuntos de datos distribuídos a través de los procesos miembros de un grupo. Estas operaciones toman un arreglo de datos de entrada de cada proceso y realizan una operación específica, operando elemento por elemento de dichos arreglos. La función MPI_REDUCE realiza un cálculo colectivo almacenando los resultados obtenidos en la memoria del proceso raíz de la operación (ver Fig. 3.2); la sintaxis de esta función es:

MPI_REDUCE (sendbuf, recvbuf, count, datatype, op, root, comm)

sendbuf	IN	Dirección de los datos de entrada (choice)
recvbuf	OUT	Dirección de almacenamiento de los resultados (choice)
count	IN	Número de datos de la operación (entero)
datatype	IN	Tipo de datos de la operación (manija)
op	IN	Operación a ser realizada (manija)
root	IN	Rango del proceso raíz (entero)
comm	IN	Comunicador de la operación (manija)

Esta función toma los *count* datos del tipo *datatype* almacenados a partir de la dirección especificada por *sendbuf* y realiza *count* operaciones *op* elemento por elemento. Los resultados son almacenados en el proceso *root* a partir de la dirección *recvbuf*. Las operaciones posibles para el parámetro *op* se especifican en la Tabla 3.2.

La función MPI_ALLREDUCE tiene el mismo objetivo que MPI_REDUCE, con la característica adicional de que reproduce en todos los procesos participantes los resultados obtenidos. La sintaxis de esta función es:

Nombre	Significado	Tipos de datos válidos
MPI_MAX	máximo	enteros, reales
MPI_MIN	mínimo	enteros, reales
MPI_SUM	suma	enteros, reales y complejos
MPI_PROD	producto	enteros, reales y complejos
MPI_LAND	and lógico	enteros y lógicos
MPI_BAND	and bit a bit	enteros y byte
MPI_LOR	or lógico	enteros y lógicos
MPI_BOR	or bit a bit	enteros y byte
MPI_LXOR	xor lógico	enteros y lógicos
MPI_BXOR	xor bit a bit	enteros y byte

Tabla 3.2: Operaciones posibles para cáculos colectivos

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)

sendbuf	IN	Dirección de los datos de entrada (choice)
recvbuf	OUT	Dirección de almacenamiento de los resultados (choice)
count	IN	Número de datos de la operación (entero)
datatype	IN	Tipo de datos de la operación (manija)
op	IN	Operación a ser realizada (manija)
comm	IN	Comunicador de la operación (manija)

En donde *sendbuf*, *recvbuf*, *count*, *datatype*, *op* y *comm* tienen el mismo significado que para la función MPI_REDUCE y las operaciones posibles para *op* son las de la Tabla 3.2.

3.5 CREACIÓN DE GRUPOS Y COMUNICADORES

Como se discutió anteriormente, los comunicadores son utilizados por el MPI para crear canales de comunicación accesibles sólo a un conjunto determinado de procesos. Esto es útil para una mejor organización de las comunicaciones entre los procesos cuando éstos son agrupados para realizar tareas diferentes o cuando se construyen o utilizan bibliotecas paralelas.

La creación de un comunicador consiste en la creación de un grupo de procesos y la asignación a éste de un contexto de comunicación. Primero se describirán las operaciones definidas por el MPI para la creación de grupos de procesos y después las funciones para la construcción de comunicadores.

3.5.1 Creación de Grupos

Antes de describir la forma de crear nuevos grupos dentro de un programa MPI, se discutirán brevemente tres funciones útiles para obtener información acerca

de un grupo determinado. Dicha información incluye: el grupo asociado a un comunicador, el *tamaño* de un grupo y el rango que tiene un proceso dado dentro de un grupo determinado.

La función que permite obtener una referencia válida al grupo asociado a un comunicador es MPI_COMM_GROUP, la cual tiene la siguiente sintaxis:

MPI_COMM_GROUP(comm, group)

comm	IN	Comunicador (manija)
group	OUT	Grupo asociado al comunicador (manija)

Esta función coloca en *group* una referencia válida al grupo asociado con el comunicador *comm*. La función MPI_GROUP_SIZE permite conocer el *tamaño* (o número de procesos) de un grupo, la sintaxis de esta función es:

MPI_GROUP_SIZE(group, size)

group	IN	Grupo del cual se desea conocer el tamaño (manija)
size	OUT	Tamaño del grupo (entero)

mientras que la función MPI_GROUP_RANK permite a un proceso dado conocer su rango dentro de un grupo, la sintaxis de esta función es:

MPI_GROUP_RANK(group, rank)

group	IN	Grupo dentro del cual se desea conocer el rango (manija)
rank	OUT	Rango dentro del grupo (entero)

En el MPI los grupos únicamente pueden ser creados a partir de grupos previamente definidos, siendo el grupo base en todo programa el grupo asociado al comunicador MPI_COMM_WORLD. Existen 5 operaciones básicas para construir un grupo de procesos: la unión de dos grupos, la intersección entre dos grupos, la diferencia de dos grupos, la exclusión de algunos miembros de un grupo y la inclusión de los miembros de un grupo en otro. Para la descripción de estas operaciones supondremos la existencia de los procesos $\{a, b, c, d, e, f, g, h\}$, los cuales forman dos grupos con los siguientes rangos: grupo1=$\{0 = a, 1 = b, 2 = c, 3 = d, 4 = e, 5 = f\}$ y grupo2=$\{0 = g, 1 = a, 2 = h, 3 = b\}$.

Unión de dos grupos

Un grupo creado a partir de la unión dos grupos ya definidos consistirá en todos los procesos que pertenecen al primer grupo más todos los procesos que pertenecen al segundo grupo. Los miembros del primer grupo conservarán sus rangos dentro del nuevo grupo, mientras que los miembros del segundo grupo tendrán nuevos rangos, respetando su orden original y empezando por n, donde n es tamaño del primer grupo; por ejemplo, la unión de grupo1 y grupo2 tendrá como resultado

grupo=$\{0 = a, 1 = b, 2 = c, 3 = d, 4 = e, 5 = f, 6 = g, 7 = h\}$. La función que construye un grupo a partir de la unión de 2 grupos definidos es MPI_GROUP_UNION, cuya sintaxis es:

MPI_GROUP_UNION(group1, group2, newgroup)

group1	IN	Referencia al primer grupo (manija)
group2	IN	Referencia al segundo grupo (manija)
newgroup	OUT	Referencia al grupo nuevo (manija)

Intersección de dos grupos

Un grupo creado a partir de la intersección de dos grupos consistirá en todos los procesos que pertenezcan tanto al primer grupo como al segundo. Los procesos que pertenezcan al nuevo grupo tendrán rangos con el mismo orden que en el primer grupo; por ejemplo, el resultado de la unión de grupo1 y grupo2 es grupo=$\{0 = a, 1 = b\}$. La función que construye un grupo a partir de la intersección de dos grupos es MPI_GROUP_INTERSECTION, la cual tiene la siguiente sintaxis:

MPI_GROUP_INTERSECTION(group1, group2, newgroup)

group1	IN	Referencia al primer grupo (manija)
group2	IN	Referencia al segundo grupo (manija)
newgroup	OUT	Referencia al grupo nuevo (manija)

Diferencia de dos grupos

Un grupo creado a partir de la diferencia de dos grupos consistirá en todos los procesos que pertenezcan al primer grupo y que no pertenezcan al segundo. Los rangos de los procesos en el nuevo grupo estarán ordenados en la misma forma que en el primer grupo. Por ejemplo, la diferencia de grupo1 y grupo2 tiene como resultado grupo=$\{0 = c, 1 = d, 2 = e, 3 = f\}$. La función MPI_GROUP_DIFFERENCE realiza esta operación, y su sintaxis es:

MPI_GROUP_DIFFERENCE(group1, group2, newgroup)

group1	IN	Referencia al primer grupo (manija)
group2	IN	Referencia al segundo grupo (manija)
newgroup	OUT	Referencia al grupo nuevo (manija)

Exclusión de miembros de un grupo

Un grupo creado a partir de la exclusión de algunos miembros de un grupo ya definido consistirá en todos los procesos que no hayan sido excluídos. Los rangos

de estos procesos en el nuevo grupo tendrán el mismo orden que en el grupo original; por ejemplo, la exclusión de los procesos $\{0 = a, 4 = e\}$ de grupo1 tiene como resultado grupo=$\{0 = b, 1 = c, 2 = d, 3 = f\}$. La función que crea un nuevo grupo a partir de la exclusión de algunos miembros de un grupo ya definido es MPI_GROUP_EXCL, cuya sintaxis es:

MPI_GROUP_EXCL(group, n, ranks, newgroup)

group	IN	Referencia al grupo original (manija)
n	IN	Número de procesos que se excluirán del grupo (entero)
ranks	IN	Rangos de los procesos que serán excluídos (arreglo de enteros)
newgroup	OUT	Referencia al nuevo grupo (manija)

El arreglo *ranks* debe contener rangos válidos y no repetidos dentro del grupo *group*.

Inclusión de procesos a un grupo

Un grupo de procesos que es creado a partir de la inclusión de algunos miembros de un grupo ya definido, consistirá en los procesos que fueron incluídos en la operación de inclusión. Los rangos de los procesos pertenecientes al nuevo grupo tendrán rangos con el orden en que aparecen en el arreglo *ranks* (ver sintaxis de la función); por ejemplo, el resultado de la inclusión de los procesos $\{0 = a, 2 = c, 1 = b\}$ es grupo=$\{0 = a, 1 = c, 2 = b\}$. La función que realiza esta operación es MPI_GROUP_INCL, cuya sintaxis es:

MPI_GROUP_INCL(group, n, ranks, newgroup)

group	IN	Referencia al grupo original (manija)
n	IN	Número de procesos que serán incluídos (entero)
ranks	IN	Rangos de los procesos que serán incluídos (arreglo de enteros)
newgroup	OUT	Referencia al nuevo grupo (manija)

Evidentemente las operaciones realizadas por las funciones MPI_GROUP_INCL y MPI_GROUP_EXCL son complementarias, es decir, la inclusión de algunos procesos pertenecientes a un grupo es igual a la exclusión de los otros procesos de dicho grupo. Por último, MPI permite la destrucción de un grupo mediante la función MPI_GROUP_FREE, cuya sintaxis es:

MPI_GROUP_FREE(group)

group	INOUT	Grupo a ser destruído (manija)

Esta función marcará el objeto referenciado por *group* para su desalojo, colocando en la manija un valor nulo. Si existe alguna comunicación asociada con el objeto, ésta se completará normalmente y después el objeto será desalojado.

3.5.2 Creación de Comunicadores

De forma similar a los grupos, existen dos funciones que permiten obtener información acerca de un comunicador. Estas funciones son MPI_COMM_SIZE y MPI_COMM_RANK, las cuales proporcionan el *tamaño* del grupo asociado al comunicador y el rango de un proceso dentro del comunicador, respectivamente. La sintaxis de estas funciones es:

MPI_COMM_SIZE(comm, size)

comm	IN	Comunicador (manija)
size	OUT	Tamaño del comunicador (entero)

MPI_COMM_RANK(comm, rank)

comm	IN	Comunicador (manija)
rank	OUT	Rango del proceso dentro del comunicador (entero)

Los comunicadores pueden ser creados de dos formas distintas: a partir de un grupo creado previamente o a partir de un comunicador ya definido. La función MPI_COMM_CREATE crea un comunicador a partir de un grupo, su sintaxis es:

MPI_COMM_CREATE(comm, group, newcomm)

comm	IN	Comunicador a través del cual se realiza la operación (manija)
group	IN	Grupo que será asociado al nuevo comunicador (manija)
newcomm	OUT	Nuevo comunicador (manija)

Podemos observar que la ejecución de esta función requiere de un comunicador *comm*, y debe ser realizada por todos los procesos pertenecientes a éste, aún aquellos que no pertencen a *group*. A su vez, *group* debe ser un subconjunto del grupo asociado a *comm*.

Para la creación de nuevos comunicadores a partir de comunicadores ya definidos, existen dos funciones. MPI_COMM_DUP, la cual duplica un comunicador, es decir, crea un nuevo comunicador con el mismo grupo asociado pero con un contexto de comunicación distinto. La sintaxis de esta función es:

MPI_COMM_DUP(comm, newcomm)

comm	IN	Comunicador a ser duplicado (manija)
newcomm	OUT	Nuevo comunicador (manija)

Por otra parte, MPI_COMM_SPLIT parte el grupo asociado a un comunicador en conjuntos disconexos, creando un comunicador para cada uno de éstos. La sintaxis de esta función es:

MPI_COMM_SPLIT(comm, color, key, newcomm)

comm	IN	Comunicador (manija)
color	IN	Control de asignación de grupo(entero)
key	IN	Control de asingación de rango (entero)
newcomm	OUT	Nuevo comunicador (manija)

Todos los procesos pertenecientes a *comm* deben realizar la llamada a esta función para su terminación de ésta. MPI_COMM_SPLIT creará tantos subgrupos como valores diferentes de *color* sean utilizados en las llamadas de los distintos procesos. Todos los procesos que usen el mismo valor para el argumento *color* pertenecerán al mismo comunicador, con rangos asignados de acuerdo al valor utilizado como *key*.

Por último, de manera similar a los grupos, MPI permite la destrucción de comunicadores, lo cual se realiza a través de la función MPI_COMM_FREE, cuya sintaxis es:

MPI_COMM_FREE(comm)

comm	INOUT	Comunicador a ser destruido (manija)

Todos los procesos pertenecientes a *comm* deben realizar la llamada a esta función para que ésta pueda ser completada. Si existen comunicaciones pendientes asociadas a este comunicador, éstas terminarán normalmente y después el comunicador será desalojado.

Hasta aquí se han presentado brevemente los principales aspectos del MPI. Las funciones descritas son suficientes para construir un programa paralelo. En el siguiente capítulo se mostrará una aplicación de una red neuronal artificial utilizada como controlador de un sistema dinámico, se discutirá el diseño de un algoritmo paralelo de entrenamiento para dicha red y su implementación utilizando la biblioteca MPI.

Capítulo 4

ALGORITMO PARALELO DE ENTRENAMIENTO PARA EL CONTROL DE SISTEMAS DINÁMICOS DISCRETOS EN EL AMBIENTE DE INTERCAMBIO DE MENSAJES MPI

Este capítulo esta organizado de la siguiente manera: primero se describe el problema de control que se quiere resolver, así como el sistema dinámico involucrado; posteriormente se presenta el algoritmo de entrenamiento empleado y su implementación paralela usando el ambiente de intercambio de mensajes MPI.

4.1 DESCRIPCIÓN DEL SISTEMA DINÁMICO

La Fig. 4.1 esquematiza el problema de control que se desea resolver, el cual consiste en un barco que navega sobre un río cuyo ancho es de 200 unidades de longitud; el objetivo es conducir el barco hacia un pequeño muelle ubicado en una de las orillas del río, de forma tal que al llegar a éste su velocidad sea igual a cero. El barco se mueve debido a la acción de un motor de velocidad variable y puede ser dirigido hacia cualquier dirección. Se supondrá que el barco no puede tener una velocidad negativa y además que no puede rebasar una velocidad límite máxima.

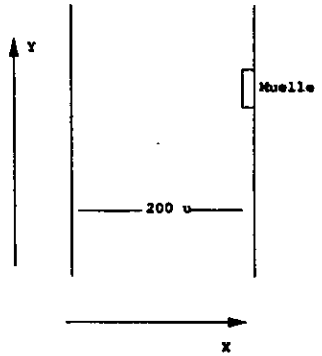

Figura 4.1: Esquema del problema a resolver

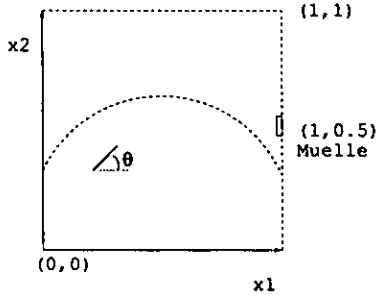

Figura 4.2: Esquema del sistema dinámico

Las ecuaciones dinámicas discretas que describen este sistema son

$$x_{k+1} = x_k + \Delta t v_k \cos\theta, \tag{4.1}$$

$$y_{k+1} = y_k + \Delta t v_k \sin\theta, \tag{4.2}$$

$$v_{k+1} = v_k + \Delta t A, \tag{4.3}$$

donde el par $\{x, y\}$ representa la posición del barco dentro del río, v su velocidad, Θ el ángulo de su dirección y A su aceleración, mientras que k es el k-ésimo paso en el tiempo y Δt es la magnitud de cada uno de estos pasos. La corriente del río estará modelada por la siguiente expresión

$$f_c(x_k) = 7.5\left(\frac{x_k}{50} - \left(\frac{x_k}{100}\right)^2\right). \tag{4.4}$$

Intentaremos resolver este problema utilizando el esquema red neuronal-sistema dinámico discutido en la Sección 1.6. La red neuronal estará encargada de generar

las señales de control necesarias para llevar al barco hasta el muelle, bajo las condiciones y restricciones impuestas; el estado del sistema dinámico estará específicado por la posicion y la velocidad del barco.

La Fig. 4.2 muestra el modelo utlizado para el entrenamiento de la red neuronal; el barco se debe de mover solamente dentro de una región delimitada del río, además de que las variables de estado han sido normalizadas a fin de que sus magnitudes correspondan a valores dentro del rango de las funciones de activación de la red neuronal. Las nuevas variables de estado son: $x_1 = x/200$, $x_2 = y/200$, $x3 = v/200$. Las ecuaciones del sistema dinámico quedan entonces de la siguiente forma

$$x_1(k+1) = x_1(k) + x_3(k)\Delta t \cos\theta, \tag{4.5}$$

$$x_2(k+1) = x_2(k) + x_3(k)\Delta t \sin\theta + f_c(x_1(k)), \tag{4.6}$$

y

$$x_3(k+1) = \begin{cases} V_{max}, & \text{si } x_3(k) + \Delta ta > V_{max} \\ x_3(k) + \Delta ta, & \text{si } 0 < x_3(k) + \Delta ta < V_{max} \\ 0, & \text{si } x_3(k) + \Delta ta < 0; \end{cases} \tag{4.7}$$

la definición de x_3 corresponde a las limitaciones de la velocidad del barco señaladas en la descripción del problema. El efecto de la corriente del rio f_c esta dada por

$$f_c(x_1) = 0.15(x_1 - x_1^2); \tag{4.8}$$

las variables de control a generar por el perceptrón serán el ángulo θ que especifica la dirección del barco y la aceleración a del mismo, las cuales estarán determinadas por

$$\theta = \pi(2u_1 - 1), \tag{4.9}$$

$$a = a_m(2u_2 - 1); \tag{4.10}$$

donde a_m es la aceleración máxima del barco, u_1 y u_2 son los niveles de activación de los nodos de la última capa del perceptrón, los cuales están acotados entre 0 y 1 si se usan funciones de activación sigmoidales.

4.2 ALGORITMO DE ENTRENAMIENTO

El método de entrenamiento a utilizar está basado en el algoritmo de retropropagación en el tiempo descrito en la sección 1.6 y consiste en presentar a la red neuronal un conjunto de M patrones de entrenamiento compuestos por un estado inicial dado y su correspondiente estado final deseado; el sistema dinámico evolucionará a partir de cada estado inicial $\vec{z}_{m1}$ hasta un estado final $\vec{z}_{mf}$, para $m = 1, 2, ..., M$, donde el error entre cada estado final alcanzado y el estado final

deseado para cada uno de estos pares será denotado como $\mathcal{E}_m$. El error total estará dado por

$$\mathcal{E} = \sum_{m=1}^{M} \mathcal{E}_m = \frac{1}{2} \sum_{m=1}^{M} |\vec{z}_{mf} - \vec{z}_i|^2 . \tag{4.11}$$

Una vez determinado éste, el siguiente paso es calcular su gradiente con respecto a los pesos, $\partial\mathcal{E}/\partial w_{ij}^{(l)}$, con el objeto de determinar la dirección en el espacio fase en la cual el conjunto de pesos $\{w_{ij}^{(l)}\}$ deberá ser modificado. A continuación se enumeran los pasos del algoritmo:

- *Paso 1.* Dividir el espacio fase en M celdas diferentes.
- *Paso 2.* Determinar aleatoriamente los pesos de las conexiones en el perceptrón.
- *Paso 3.* Generar, a partir de un punto seleccionado aleatoriamente dentro de cada una de las celdas construidas en el paso 1, un conjunto de M trayectorias las cuales evolucionarán en el tiempo hasta cumplir con alguna condición predeterminada.
- *Paso 4.* El algoritmo se detendrá si para las M trayectorias se cumple que $|x_{if} - x_{it}| < \epsilon \ \forall i$, donde ϵ es un rango de error predeterminado. En caso de no cumplirse esta condición, se continúa con el siguiente paso.
- *Paso 5.* Calcular el gradiente del error utilizando el algoritmo de retropropagación en el tiempo y actualizar los pesos usando el método de los gradientes conjugados. La función de error es la mostrada en la Ec. (4.11). En este paso se añade una búsqueda en una dimensión a partir de la cual se determina el tamaño del "paso" (i.e. la magnitud de la modificación de los pesos) que minimiza el error en la dirección del gradiente conjugado.
- *Paso 6.* Repetir los pasos 3-5 hasta cumplir con el criterio de convergencia.

A continuación se hará una descripción tanto de la metodología para el diseño utilizada como del diseño mismo de la versión paralela de este algoritmo.

4.3 DISEÑO PARALELO

En el Capítulo 2 se mencionó que el crecimiento considerable del uso de máquinas paralelas en los últimos años se debe a las necesidades cada vez mayores de capacidad de cómputo para la solución de problemas de ciencia e ingeniería, así como al notable mejoramiento en las redes de interconexión entre procesadores; además, en dicho capítulo se presentaron las diferentes clasificaciones de las máquinas paralelas, se hizo una breve descripción de algunas de estas máquinas y se discutió

brevemente algunas de las interfaces de programación paralela más utilizadas. En esta sección se hará la descripción de la implementación paralela del algoritmo de entrenamiento presentado anteriormente, discutiendo brevemente los pasos de diseño utilizados y las decisiones tomadas para la paralelización de acuerdo a estos pasos.

Aunque no existen reglas específicas para la construcción de un algoritmo paralelo, debido a que generalmente éste puede ser implementado de muy diversas maneras, es necesario realizar un diseño metódico del algoritmo con el fin de evaluar las diferentes posibilidades de paralelización y determinar cual de ellas proporcionará una mejor implementación. El método que se ha utilizado consiste en 4 pasos generales: *Partición, Comunicación, Aglomeración y Mapeo*[7].

4.3.1 Partición

Dado que un programa paralelo consiste en varios procesos que trabajan en forma cooperativa para la solución de un problema dado, es necesario determinar que parte del problema será solucionado por cada uno de las procesos; esto es, es necesario dividir o partir el problema en varias partes, cada una de los cuales estará determinada por un conjunto de datos y un conjunto de operaciones a realizar sobre dichos datos; cada una de estas partes es denominada *tarea*. Existen dos modelos diferentes de partición: *descomposición del dominio* y *descomposición funcional*, los cuales son descritos a continuación.

La *descomposición del dominio* consiste en dividir el conjunto de datos del problema en varios subconjuntos y después determinar que operaciones se realizarán sobre éstos, es decir, construir tareas a partir de la división de los datos involucrados en el algoritmo. Los datos a dividir pueden ser tanto los de entrada, los de salida o cualesquiera otros datos intermedios a calcular en el algoritmo. Una regla intuitiva generalmente seguida consiste en dividir ya sea la estructura de datos más grande o la más utilizada. Por otra parte, la *descomposición funcional* consiste en identificar todas las operaciones independientes en el algoritmo y asignar a cada una de éstas un conjunto de datos sobre los cuales operar, en otras palabras, construir tareas a partir de las operaciones involucradas en el algoritmo.

Para ejemplificar ambos modelos, consideremos un establecimiento de lavado de automóviles. En el modelo de partición del dominio cada trabajador se encargará del servicio completo a un automóvil, mientras que en el modelo de partición funcional cada trabajador estará encargado de una tarea específica como enjabonar, aspirar, etc. Un ejemplo más técnico se presenta en problemas de procesamiento de imágenes donde es necesario aplicar varios filtros diferentes a una sola imagen. En la descomposición de dominio todas los procesos cooperarán en la aplicación de un solo filtro, repartiéndose partes de la imagen, mientras que en la descomposición funcional cada proceso aplicará un filtro diferente a la imagen completa. Estos

modelos no son excluyentes y un mismo algoritmo puede hacer uso de ambos; retomando el ejemplo de procesamiento de imagenes, un conjunto de procesos puede aplicar un filtro a la imagen repartiéndose partes de ella, mientras otro conjunto de procesos realiza la misma función aplicando un filtro diferente.

En el algoritmo de entrenamiento de la red neuronal se tienen dos estructuras de datos principales: el conjunto de pesos que caracterizan a la red neuronal y el conjunto de celdas a partir de las cuales se generan las trayectorias en el proceso de entrenamiento. En el primer caso, los valores de activación de las unidades de un nivel dado en un perceptrón son calculados a partir de los valores de activación de las unidades del nivel inmediato anterior, por lo cual solo unidades pertenecientes a un mismo nivel pueden calcularse en forma concurrente. En el segundo caso, a partir de cada una de las celdas en que se divide el espacio fase se genera una trayectoria que es completamente independiente de las demás, de modo que puede hacerse fácilmente una división de este conjunto de datos. La generación de estas trayectorias representa, además, la mayor parte de la carga de trabajo en el programa, por lo cual se supone conveniente esta partición.

Las operaciones generales que se realizan durante el programa pueden separarse en los siguientes pasos:

1. Asignación inicial de los pesos.
2. Asignación del estado inicial dentro de cada celda y la generación de la trayectoria correspondiente.
3. Cálculo del gradiente parcial correspondiente a cada una de las trayectorias y determinación del gradiente total.
4. Cálculo de la dirección del gradiente conjugado en el espacio de los pesos.
5. Cálculo de la magnitud del cambio de los pesos en la dirección del gradiente conjugado.
6. Cálculo de los nuevos valores de los pesos.

Para generar las trayectorias es necesario determinar primero los pesos de las conexiones en la red neuronal y para calcular el gradiente es necesario haber generado todas las trayectorias; además, la modificación de los pesos requiere de la magnitud del cambio en la dirección del gradiente conjugado y obviamente para calcular éste, el gradiente debe calcularse previamente. Es decir, estas operaciones no son independientes, puesto que cada una necesita de los resultados de la anterior para poder realizarse y por lo tanto no pueden ser ejecutadas concurrentemente. Debido a ésto, podemos considerar a la partición del espacio fase como la mejor opción de paralelización.

4.3.2 Comunicación

La información que cada una de las tareas necesita para realizar las operaciones está constituída por: los estados iniciales de las trayectorias, los pesos de las conexiones de la red neuronal, el gradiente del error con respecto a los pesos y la magnitud del paso en la dirección del gradiente conjugado. Dado que todos los procesos necesitan la misma información, el esquema de comunicación a utilizar debe ser global, esto es, todos los procesos participarán en la misma operación de comunicación. Es importante mencionar que las comunicaciones globales generalmente producen problemas en la escalabilidad de un algoritmo paralelo, ya que la adición de nuevos procesos incrementa considerablemente el número de mensajes necesarios para completar la comunicación (ver Sección 3.4); además, un aumento en el tamaño del problema incrementará la magnitud de dichos mensajes; debido a ésto será necesario considerar los costos de comunicación en el desempeño del programa paralelo y analizar hasta que punto son aceptables y, en caso de que sea necesario, rediseñar el algoritmo de comunicación. Por otra parte, debido a que las operaciones realizadas en el algoritmo son en su mayoría dependientes, será necesario utilizar una comunicación síncrona, en donde todos los procesos deben de realizar la operación de comunicación en una etapa determinada para poder continuar con sus tareas asignadas.

4.3.3 Aglomeración y Mapeo

Hasta este punto hemos determinado el trabajo a realizar por cada tarea y las comunicaciones que cada una de éstas debe realizar para llevar a cabo dicho trabajo, sin embargo, la partición se ha realizado sin considerar aún cuantos procesos participarán en el programa paralelo, lo cual estará determinado por el número de procesadores disponibles y por el tamaño del problema. Si el número de celdas en que se divide el espacio fase es mayor que el número de procesos entonces será necesario hacer una *aglomeración*, es decir, unir varias tareas en un solo proceso. Aunque puede considerarse que la aglomeración y el mapeo son dos pasos de diseño diferentes, el último consiste principalmente en asignar procesos formados a partir de la aglomeración de tareas a los procesadores disponibles, conforme a las características tanto del procesador como de su red de comunicación. Aquí consideraremos ambos pasos como uno solo, refiriéndonos a él como aglomeración.

Generalmente con la aglomeración de tareas se intenta optimizar algún aspecto del algoritmo paralelo, como puede ser la comunicación o el balance de la carga. El algoritmo descrito hasta ahora requiere de un esquema de comunicación global, y además dado un número fijo de procesos el costo de la comunicación no es afectado por una aglomeración particular de tareas; por lo tanto, los esfuerzos de aglomeración se concentrarán en el reparto de la carga de trabajo entre los procesos. Este reparto determina la cantidad de trabajo que cada uno de los procesos debe

realizar y su importancia se muestra en las siguientes ecuaciones. Definimos el tiempo de ejecución del programa paralelo como

$$t_p = t_c + t_{comm} + t_{ocio}, \tag{4.12}$$

donde t_c, t_{comm} y t_{ocio} son el tiempo de cálculo, el tiempo utilizado en operaciones de comunicación y el tiempo de ocio, respectivamente. Si suponemos que tenemos una carga de trabajo total de 10 unidades indivisibles y contamos con tres procesos, un posible reparto de carga sería: $C_{P_0} = 6$, $C_{P_1} = 2$, $C_{P_2} = 2$. Si suponemos que t_{comm} es despreciable y que cada unidad de carga se realiza en una unidad de tiempo entonces:

$$t_{p_0} = 6+0+0 = 6, \tag{4.13}$$
$$t_{p_1} = 2+0+4 = 6, \tag{4.14}$$
$$t_{p_2} = 2+0+4 = 6; \tag{4.15}$$

mientras que si hacemos el siguiente reparto de carga: $C_{P_0} = 4$, $C_{P_1} = 3$, $C_{P_2} = 3$, tendremos lo siguiente:

$$t_{p_0} = 4+0+0 = 4, \tag{4.16}$$
$$t_{p_1} = 3+0+1 = 4, \tag{4.17}$$
$$t_{p_2} = 3+0+1 = 4; \tag{4.18}$$

Así pues, un mejor balance de carga reduce el tiempo de ejecución de un programa paralelo; sin embargo, en el algoritmo de entrenamiento de la red neuronal la carga no está determinada por el número de trayectorias asignadas a cada proceso, sino por la cantidad de pasos en el tiempo que deben ser calculados para completar cada una de estas trayectorias; es decir, procesos con el mismo número de celdas asignadas no necesariamente tendrán la misma carga de trabajo; además, no es posible conocer de antemano cual será la carga de trabajo asociada a cada celda para cada iteración, lo cual dificulta el reparto. En este trabajo se utilizarán y evaluarán tres métodos para determinar el reparto de la carga, los cuales llamaremos: de bloque, cíclico y bin packing.

4.3.4 Métodos de reparto de carga

El método de bloque consiste en dividir el número de celdas en bloques contiguos y asignar a cada proceso cada uno de estos bloques, como se muestra en la parte superior de la Fig. 4.3. El segundo método consiste en repartir una celda a cada procesador sucesivamente y repetir este procedimiento hasta agotar el número de celdas, como puede observarse en la parte inferior de la misma figura. Ambos métodos poseen la ventaja de ser sencillos y no requerir cálculos complicados para determinar el reparto de la carga, por lo que no introducen sobrecargas significativas en el tiempo de ejecución del algoritmo; sin embargo, estos métodos solo

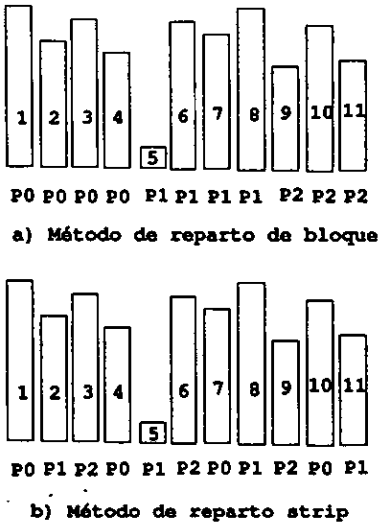

Figura 4.3: Métodos de bloque y cíclico para el reparto de carga. La altura de cada caja es una medida de su carga de trabajo asociada. En la parte inferior se indica el procesador encargado de realizar cada tarea.

consideran el número de celdas y no la carga de trabajo asociada a cada una de éstas, por lo cual podrían proporcionar repartos ineficientes. En contraste, el método bin packing toma en consideración la carga de trabajo asociada a cada celda; el reparto de carga que realiza este método se describe en los siguientes pasos, ilustrados en la Fig. 4.4:

1. Ordenar el conjunto de celdas en forma descendente de acuerdo a su carga de trabajo asociada.
2. Repartir inicialmente las primeras N celdas ordenadas, una por procesador, en el orden $P_0, P_1, P_2, ..., P_{N-1}$.
3. Asignar la siguiente celda al proceso que en ese momento tenga asignada la menor carga acumulada de trabajo.
4. Repetir el paso 3 hasta terminar el reparto de todas las celdas.

Ya que los dos primeros métodos no consideran la carga de trabajo asociada a cada celda, pueden aplicarse sin ninguna dificultad a pesar de que las cargas de trabajo son desconocidas. Para aplicar el método bin packing, estas cargas serán estimadas a partir de las cargas que cada celda ha tenido en el pasado. En el programa implementado, hemos considerado los valores de las cargas asociadas a cada celda en las últimas 5 iteraciones del entrenamiento.

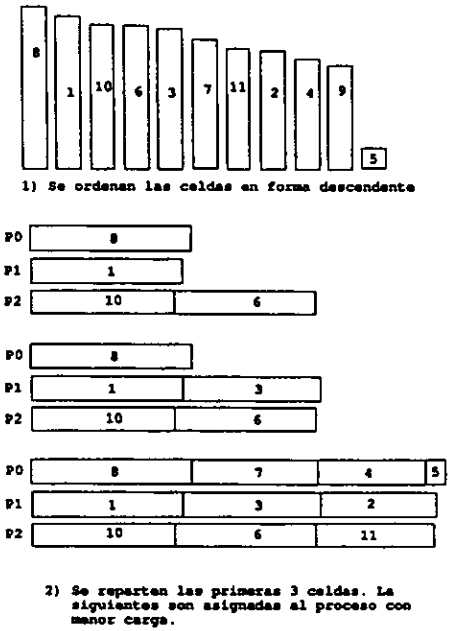

Figura 4.4: Método bin packing para el reparto de carga

En la siguiente sección mostraremos la implementación del algoritmo paralelo generada a partir de las decisiones de partición, comunicación y aglomeración tomadas en esta etapa de diseño. En esta implementación se utilizarán los tres métodos de reparto de carga descritos, con el objeto de realizar un estudio comparativo entre ellos.

4.4 IMPLEMENTACIÓN

En la implementación del algoritmo paralelo se ha utilizado el ambiente estándar de intercambio de mensajes MPI para realizar las operaciones de comunicación entre los procesos. La Fig 4.5 muestra el diagrama de flujo del programa, mostrando únicamente las operaciones generales. Las principales estructuras de datos a utilizar son:

- *procmap*: contiene la información sobre el mapeo de las celdas, es decir, indica que proceso generará la trayectoria correspondiente a cada una de las celdas. Así, *procmap(nc)=id* significa que la celda *nc* ha sido asignada al proceso *id*.
- *w,gw,cw*: contienen los valores de los pesos de las conexiones correspondientes

a la red neuronal, del gradiente del error con respecto a los pesos y del gradiente conjugado, respectivamente.

- *pgw*: contiene los valores parciales del gradiente del error con respecto a los pesos, calculados por cada proceso.
- *zt*: contiene los valores del estado final deseado de las trayectorias.
- *xi*: contiene el estado inicial de la trayectoria asociada a cada celda.

Las declaraciones de estas variables son las siguientes:

```
INTEGER   infomax(MAX_CELLS), infopre(MAX_WW,MAX_CELLS)
INTEGER   info_res(MAX_CELLS), procmap(MAX_CELLS)
DIMENSION NNL(MAX_L),ZT(MAX_SVAR),W(MAX_L1,0:MAX_NL,MAX_NL)
DIMENSION GW(MAX_L1,0:MAX_NL,MAX_NL),TC(10),WALLTIME(4)
DIMENSION XI(MAX_SVAR,MAX_CELLS),CW(MAX_L1,0:MAX_NL,MAX_NL)
DIMENSION PGW(MAX_L1,0:MAX_NL,MAX_NL),E(12000)
CHARACTER*30 prefix
CHARACTER*30 name
```

en estas declaraciones MAX_CELLS, MAX_WW, MAX_L, MAX_SVAR, MAX_NL corresponden respectivamente al máximo número de celdas, máximo número de procesos, máximo número de niveles en la red neuronal, máximo número de variables de estado y máximo número de nodos por nivel permitidos. El arreglo tridimensional w se utiliza de manera que $w(i,j,k)$ corresponde al peso de la conexión entre la j-ésima unidad del nivel i y la k-ésima unidad del nivel $i+1$, $w_{ki}^{(i+1)}$, para $j = 1,2,..,J(i)$, $k = 1,2,...,J(i+1)$, $i = 1,2,...,L-1$, donde L es el número de niveles de la red neuronal y $J(i)$ es el número de nodos del i-ésimo nivel; mientras que $w(i,0,k)$ corresponde al valor umbral de la k-ésima unidad del nivel $i+1$. Una estructura similar corresponde a las variables *pwg*, *gw* y *cw*. Los datos de entrada del programa serán los siguientes:

- Número de divisiones que se realizarán en cada dimensión del espacio de fase, esto determinará el número de celdas totales.
- Número máximo de pasos en el tiempo para dar por terminada una trayectoria si ésta no ha alcanzado ninguna de las condiciones preestablecidas.
- Número máximo de intentos para obtener la convergencia del algoritmo, cada intento esta determinado por un ciclo cuyo tamaño es igual al número total de pesos de la red neuronal.
- Número de niveles de la red neuronal y número de nodos por nivel, esto es, la topología de la red.

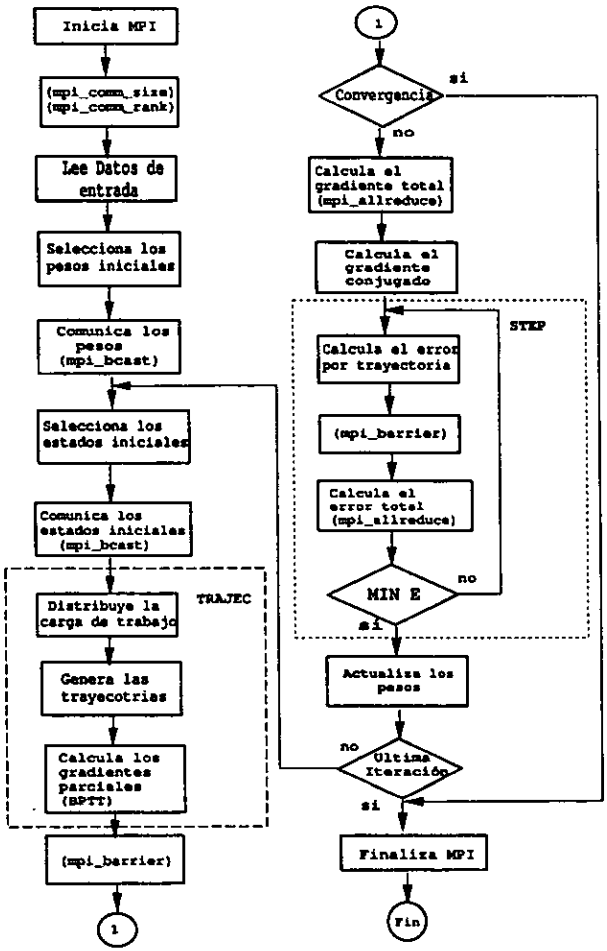

Figura 4.5: Diagrama de flujo del programa de entrenamiento

- El estado final deseado para las trayectorias.

Los pasos del algoritmo se han implementado de la siguiente manera:

* El primer paso consiste en inicializar el ambiente de intercambio de mensajes, esto se realiza mediante una llamada a la función **mpi_init**. También es necesario determinar las características del ambiente paralelo, para lo cual usamos las funciones **mpi_comm_size** y **mpi_comm_rank**, las cuales proporcionan el número de procesos que intervendrán en el programa paralelo y el número identificador de cada proceso, repectivamente (ver Sec. 3.5.2).

* El paso siguiente consiste en leer los datos de entrada, operación que debe ser realizada por todos los procesos involucrados. Una vez que los datos de entrada han sido leídos, los valores de las conexiones en el perceptrón son seleccionados aleatoriamente por uno de los procesos, en nuestro caso por el proceso cuyo identificador es igual a cero; posteriormente éstos son trasmitidos hacia todos los demás procesos utilizando la siguiente llamada a la función **mpi_bcast**:

```
      call mpi_bcast(w, lenghtw, mpi_double_precision,0,
     *               mpi_comm_world, ierror)
```

* Una vez que todos los procesos conocen los valores de los pesos del perceptrón se inicia la parte iterativa del algoritmo. Primero es necesario determinar un estado inicial dentro de cada una de las celdas en que se ha dividido el espacio fase; estos estados son determinados aleatoriamente por uno de los procesos, en este caso nuevamente escogemos el proceso con rango 0, trasmitiéndolos después al resto de los procesos usando de nuevo una llamada a la función mpi_bcast, i.e.

```
      call mpi_bcast(xi, lenghtxi,mpi_double_precision,0,
     *               mpi_comm_world,ierror);
```

un método alternativo consiste en que cada proceso determine aleatoriamente el estado inicial dentro de cada una de las celdas que le sean asignadas, sin embargo, se ha optado por el primer procedimiento por dos razones:

1. De esta manera se asegura que los resultados sean siempre iguales sin importar el número de procesos involucrados o el reparto de carga escogido.
2. La estructura de datos que contiene la información acerca de los estados iniciales es pequeña y por lo tanto no produce una sobrecarga importante en el programa al ser trasmitida.

* El siguiente paso consiste en generar las trayectorias, determinar cuales de ellas cumplen con las condiciones de convergencia y calcular el gradiente parcial correspondiente a cada trayectoria. Estas operaciones son realizadas dentro de la rutina denominada *trajec*. Es dentro de esta rutina en donde se realiza el reparto de la carga, utilizando una rutina denominada *loadbal*, la cual proporciona el arreglo *procmap* que contiene la asignación de las celdas que corresponden a los distintos procesos, como se discutió anteriormente. Con base en esta información cada proceso calcula las trayectorias que le corresponden, verifica la convergencia de cada una de éstas y calcula un gradiente parcial acumulando los gradientes parciales correspondientes a cada trayectoria; esta información es almacenada en el arreglo *pgw*.

* Una vez que todas las trayectorias y sus gradientes han sido calculados se hace la prueba total de convergencia, sumando el número de trayectorias que han convergido para cada proceso y comparándolo con el número total de trayectorias, de la siguiente manera:

```
      call mpi_allreduce(incflag, iflag, 1, mpi_integer,
     *                   mpi_sum, mpi_comm_world, ierror),
```

donde **incflag** contiene el número de trayectorias por procesador que han convergido. Si no se cumple con el criterio de convergencia, se continúa con el cálculo del gradiente total. La totalización del gradiente se realiza mediante la siguiente llamada a la función **mpi_allreduce**:

```
      call mpi_allreduce(pgw, gw, lenghtw, mpi_double_precision,
     *                   mpi_sum, mpi_comm_world,ierror),
```

esto es, sumando los elementos del gradiente calculados por cada proceso, de acuerdo a la función **mpi_allreduce** discutida en la Sec. 3.4.5. Como ya se ha mencionado, los gradientes parciales son calculados mediante una acumulación de los gradientes de cada una de las trayectorias asignadas a cada proceso. Este procedimiento conduce a que el gradiente total sea calculado a partir de sumandos diferentes cuyos valores numéricos dependen tanto del número de procesos que intervienen en el programa paralelo como del reparto de carga realizado. Este hecho, que aparentemente no representa ningún problema dada la propiedad asociativa de la suma, produce resultados finales distintos al ejecutar el programa con diferentes números de procesadores, debido a que en la aritmética de las computadoras la propiedad asociativa de la suma no se cumple como consecuencia del redondeo realizado en las operaciones que producen sobreflujos[14], como se describe a continuación.

Los números de punto flotante son representados en una computadora mediante una mantisa y un exponente, esto es, un número decimal y la potencia de alguna

base e por la cual el número decimal debe ser multiplicado para obtener el número original; así, tomando $e = 10$ tenemos .0002=.2e-3, 453.24=.45324e3, 2.0=.2e-1. En la notación estándar el primer dígito de la mantisa debe ser diferente de cero. El número de digitos con que una computadora puede representar la mantisa es siempre finito, debido a lo cual la computadora *redondea* cantidades que sobrepasan este límite. Por ejemplo, si se desean sumar las cantidades .009876 y .00002345, la aritmética convencional proporcionará .00989945 como resultado; sin embargo, una computadora que representa la mantisa con 4 digitos calcularía lo siguiente:

$$.9876e-2+.0023e-2=.9899e-2, \tag{4.19}$$

con un error de redondeo de .00000045. Consideremos ahora la suma de tres número distintos: 32.51, 0.4942 y 7219. Si sumamos los dos primeros y al resultado agregamos el tercero obtenemos 0.7252e4, como se muestra a continuación:

$$.3251e2+.0049e2 = .3300e2 \tag{4.20}$$

$$.0033e4+.7219e4 = .7252e4; \tag{4.21}$$

si agrupamos de distinta forma los operandos, por ejemplo sumando el primero con el tercero y posteriormente añadiendo el segundo tenemos:

$$.7219e4+.0032e4 = .7251e4 \tag{4.22}$$

$$.7251e4+.0000e4 = .7251e4, \tag{4.23}$$

siendo este resultado distinto al obtenido anteriormente. Aunque estos errores son poco significativos cuando se utilizan computadoras capaces de almacenar un número grande de cifras para la representación de la mantisa, en un proceso iterativo suelen acumularse, produciendo resultados finales erróneos. Esta característica conduce a resultados diferentes en el cálculo del gradiente cuando se usa diferente número de procesadores. Cabe señalar que estas diferencias no son inherentes a la solución paralela del problema sino al orden en que los operandos son sumados, esto es, si en diferentes ejecuciones del código serial se cambia este orden también se obtendrán resultados diferentes.

Para evitar este problema se ha realizado una pequeña modificación en la implementación del algoritmo de entrenamiento, la cual consiste en aumentar en una dimensión el arreglo *pgw* de manera que el elemento $pgw(i,j,k,l)$ almacena la componente del gradiente asociada al peso $w_{kj}^{(i+1)}$ correspondiente a la trayectoria de la l-ésima celda. Así, los procesos calculan y almacenan cada uno de estos valores de acuerdo a las trayectorias asignadas en vez de obtener los valores acumulados de las componentes del gradiente parcial. Si todos los elementos del arreglo *pgw* son igualados a cero antes de cada iteración, la operación *mpi_allreduce* con *pgw* proporcionará como resultado un arreglo con los valores parciales de las componentes del gradiente correspondientes a cada una de las celdas, faltando únicamente la suma de todos estos valores parciales para obtener el gradiente total, es decir,

$$gw(i,j,k)=\sum_{l=1}^{ncells} pgw(i,j,k,l), \tag{4.24}$$

donde *ncells* es el número de celdas (trayectorias).

* El algoritmo continúa con el cálculo de la dirección del gradiente conjugado, operación realizada independientemente por todos los procesos, para después continuar con el cálculo de la magnitud η^* del paso en la dirección del gradiente conjugado que minimiza el error total $\mathcal{E}$ definido en la Ec. (4.11). En esta última operación, los procesos calculan el error correspondiente a las trayectorias, utilizando el reparto de carga realizado en la generación de trayectorias, con el fin de obtener el error total para una magnitud η dada en la modificación de los pesos. Este error es calculado a partir de los errores en cada una de las trayectorias, en forma similar al cálculo del gradiente total con el fin de evitar los problemas de redondeo descritos anteriormente.

* El último paso en el algoritmo es obtener el nuevo conjunto de pesos; esto se hace usando la siguiente expresión:

$$w_{ij}^{(l)} = w_{ij}^{(l)} + \eta^* cw_{ij}^{(l)}, \tag{4.25}$$

donde η^* es la magnitud del paso que minimiza la función de error en la dirección de cw, que es la la dirección del gradiente conjugado.

En el siguiente capítulo se describirán los parámetros utilizados para evaluar el rendimiento de un programa paralelo, así como un análisis de los resultados obtenidos en un conjunto de pruebas de la implementación del algoritmo paralelo de entrenamiento.

Capítulo 5

ANÁLISIS DE RENDIMIENTO

Generalmente, el objetivo principal de un programa paralelo es reducir el tiempo de ejecución con respecto a su versión serial; por otra parte, es deseable que un programa paralelo utilice de manera eficiente los recursos disponibles. En este capítulo se introducirán los conceptos de *Speed-up* y *eficiencia*, los cuales permiten la evaluación del rendimiento de un programa paralelo. También se presentará el análisis de los resultados obtenidos en un conjunto de ejecuciones en la supercomputadora Origin 2000[12],[13] del programa paralelo de entrenamiento construído.

5.1 SPEED-UP

Una de las principales características a evaluar en un programa paralelo es la aceleración en el rendimiento o *speed-up*. El speed up es una relación entre el tiempo de ejecución del programa en forma serial y el tiempo de ejecución en forma paralela, esto es

$$Sp = \frac{T_1}{T_N}, \tag{5.1}$$

donde T_1 es el tiempo de ejecución en 1 procesador y T_N el tiempo de ejecución en N procesadores. En condiciones ideales, es decir, con un programa que es totalmente paralelizable, en donde se puede realizar un perfecto balance de carga y no existe ningúna sobrecarga debida a la ejecución paralela, el speed-up debe ser una función lineal de N, ya que

$$Sp = \frac{T_1}{T_N} = \frac{T_1}{T_1/N} = N; \tag{5.2}$$

sin embargo, existen una serie de factores que impiden esta situación; por ejemplo. si el programa no es completamente paralelizable, la *Ley de Amdahl*[7] impone un límite al valor máximo que el speed-up puede alcanzar. Esta ley enuncia que dado que un programa generalmente se compone de una parte paralela T_P la cual

puede repartirse entre los N procesadores que intervienen en el programa y una parte serial T_s cuya carga de trabajo es independiente del número de procesos involucrados en el programa paralelo, el speed-up esta dado por:

$$Sp = \frac{T_1}{T_N} = \frac{T_1}{Ts + T_p/N}. \tag{5.3}$$

Es decir, no importando el valor de N, el límite máximo del speed-up está determinado por la fracción del programa que no puede ser paralelizada. Aunque esta ley es útil para entender los límites en el speed-up de programas paralelos, generalmente no es usada para modelar el comportamiento del rendimiento de un programa específico, ya que no involucra todos los factores de los cuales depende el tiempo de ejecución de un programa paralelo. Un método sencillo para modelar el speed-up es tomar las ecuaciones (5.1) y (4.12)[7], de las cuales se deriva la siguiente expresión

$$Sp = \frac{T_1}{T_c + T_{comm} + T_{ocio}}; \tag{5.4}$$

en esta ecuación, T_c y T_{ocio} pueden estimarse a partir del número de procesadores, del tamaño del problema y del reparto realizado de la carga, mientras que el tiempo de comunicación T_{comm} puede calcularse a partir del número de mensajes necesarios para realizar la comunicación. La trasmisión de un mensaje entre dos procesos requiere de un tiempo dado por los parámetros T_i y T_w, donde T_i es el tiempo de inicialización de la comunicación y T_w es el tiempo de trasmisión de una unidad de información. Así, un mensaje de longitud l será trasmitido en un tiempo que estará determinado por

$$T_{msg} = T_i + T_w l. \tag{5.5}$$

Aunque un análisis riguroso del speed-up, Ec. (5.4), requiere no solo del modelo de comunicación descrito anteriormente, Ec. (5.5), sino también de modelos más complicados requeridos por las operaciones globales, tales como mpi_broadcast y mpi_allreduce, y debido a que el rendimiento de una computadora paralela depende fuertemente del rendimiento de su sistema de comunicación entre procesadores, es importante determinar los parámetros T_i y T_w de una arquitectura particular, con el fin de evaluar las posibles limitaciones de ésta. El Apéndice A contiene el procedimiento y determinación de los parámetros T_i y T_w en algunas máquinas de la UNAM.

5.2 EFICIENCIA

El speed-up proporciona una medida exacta del aumento en el rendimiento de un programa paralelo con respecto a su versión serial, sin embargo, no proporciona información acerca de la forma en que los recursos disponibles han sido utilizados. Por ejemplo, un speed-up igual a 10 puede considerarse aceptable, pero si es

necesario utilizar 100 procesadores para alcanzarlo, el programa paralelo no puede considerarse eficiente. Una medida representativa de la eficiencia puede obtenerse dividiendo el tiempo de ejecución serial entre el tiempo utilizado por *todos* los procesos que intervienen en el programa paralelo, esto es

$$E = \frac{T_1}{NT_n} = \frac{Sp}{N}; \tag{5.6}$$

es claro de esta ecuación que para el caso de paralelismo ideal, mostrado en la Ec. (5.2), la eficiencia es igual a 1. Si utilizamos las ecuaciones (5.4) y (5.6) tendremos

$$E = \frac{T_1}{NT_c + NT_{comm} + NT_{ocio}}; \tag{5.7}$$

en esta última ecuación podemos observar que la eficiencia decrece conforme aumenta al número de procesadores, ya que este hecho generalmente incrementa el tiempo de comunicación, además de que $NT_c \approx T_1$. Por ejemplo, en el caso de un balance de carga perfecto (i.e. $T_{ocio} = 0$), si en el programa paralelo el tiempo de comunicación iguala al tiempo de cálculo por procesador, la eficiencia será aproximadamente del 50%, ya que $NT_{comm} \approx NT_c$; si la eficiencia es menor que este valor entonces el programa consume mayor cantidad de tiempo en comunicaciones que en cálculos, por esta razón en la práctica se considera al 50% como un límite inferior aceptable de la eficiencia de un programa paralelo.

5.3 ESTIMACIONES EMPÍRICAS

Cuando no es posible obtener un modelo del speed-up y de la eficiencia de un programa paralelo, es útil hacer estimaciones de éstos parámetros de rendimiento a partir de mediciones en ejecuciones parciales del programa. Aunque estas estimaciones no modelan formalmente el comportamiento del programa, en la práctica pueden producir resultados confiables. Con el objeto de estudiar el comportamiento del programa paralelo se ha diseñado el siguiente experimento:

Primero, se realiza una corrida serial del programa, registrando el número de pasos en el tiempo necesarios para completar cada trayectoria y el tiempo de procesador que éstos han consumido. Con esta información se calcula el promedio de tiempo de cálculo para cada paso en la generación de las trayectorias y la carga de trabajo para cada una de las celdas en cada iteración del algoritmo. Una vez conocida la carga de trabajo, se estima el tiempo de ejecución como función del número de procesadores y para diferentes distribuciones de carga, considerando únicamente la información obtenida de la generación de trayectorias y suponiendo que el tiempo de comunicación es despreciable. A partir de los tiempos de ejecución estimados, se determina el speed-up y la eficiencia para cada tipo de carga y número de procesadores utilizado.

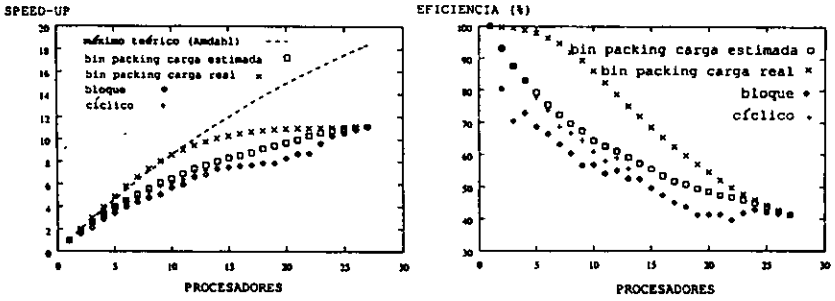

Figura 5.1: Speed-up y eficiencia estimados para el programa de entrenamiento, usando la información de la generación de trayectorias.

Se han estimado estos valores para un número máximo de procesadores igual al número de celdas en que se divide el espacio fase. Además de los tres métodos de carga descritos en la sección anterior, se ha incluido la estimación usando el método bin packing con las cargas de trabajo verdaderas correspondientes a cada iteración, con el objeto de analizar el comportamiento de este método de reparto de carga para casos en donde la carga de trabajo es conocida de antemano; también se incluye la curva de rendimiento obtenida de la Ley de Amhdal, la cual proporciona el límite superior para el speed-up. Los resultados obtenidos para el speed-up y la eficiencia se muestran en la Fig. 5.1. En estas figuras podemos observar que el comportamiento del speed-up y la eficiencia para los métodos de bloque y cíclico se espera sea similar, mientras que el método de bin packing con cargas estimadas deberá tener un mejor rendimiento que los anteriores; además, estas figuras muestran que el método bin packing proporciona el mejor reparto de carga cuando las cargas de trabajo se conocen de antemano. En algunos puntos puede observarse que el método bin packing con las cargas verdaderas sobrepasa el límite de la curva de Amdahl; esto sucede debido a que dicha curva fue calculada para el programa completo, mientras que los datos de los 4 métodos de distribución de carga corresponden solamente a la parte del programa encargada de la generación de las trayectorias. Se puede notar además que la eficiencia será menor a 50% si se utilizan más de 15 procesadores con los métodos cíclico y de bloque, más de 17 procesadores con el método bin packing con estimación de cargas de trabajo y más de 22 procesadores con el método bin packing si se conocen las cargas de trabajo reales.

Umbrales							
niveles	nodos						
1	0.0		0.0		0.0		
2	-0.40248	0.4699	-0.4374	0.20648	-0.30639	0.7526	0.05625
	-0.27007	0.1136	-0.709	-0.03184	0.34559	0.21711	-0.16292
3	-0.31456			1.0488			

pesos							
del nivel 1	al nivel 2						
nodo 1	1.4841	-2.3804	1.7013	-0.2687	0.22253	-3.016	-0.3044
	1.2399	-0.04991	1.9305	0.4555	-2.3658	-1.2933	-0.2422
nodo 2	-1.3162	1.7459	-1.3759	0.09337	-0.9195	2.1902	0.4541
	-0.8504	1.8456	-0.90643	0.10523	1.7437	0.46489	0.34528
nodo 3	0.81704	-1.9876	1.5714	-0.34323	0.2799	-02.3126	-0.5592
	0.79494	-0.3483	1.3803	0.29772	-1.5873	-0.94621	-0.2344

del nivel 2	al nivel 3
nodo 1	-0.04245
nodo 2	0.3487
nodo 3	-0.0777
nodo 4	-0.2083
nodo 5	-0.13004
nodo 6	0.4347
nodo 7	-0.27938
nodo 8	0.1881
nodo 9	-0.6608
nodo 10	-0.15618
nodo 11	0.16701
nodo 12	0.44802
nodo 13	-0.07139
nodo 14	-0.39811

	-1.4699
	3.4322
	-2.449
	0.88038
	0.2133
	4.2821
	0.9346
	-1.6957
	-.10448
	-2.5877
	-0.74743
	2.8768
	2.2287
	0.83405

Tabla 5.1: Valores de los umbrales de los nodos y de las conexiones entre ellos obtenidos después de terminado el entrenamiento.

5.4 RESULTADOS

5.4.1 Entrenamiento

Se realizó un conjunto de pruebas del programa paralelo en la supercomputadora Origin 2000 de 32 procesadores de la DGSCA-UNAM[12],[13]; estas pruebas fueron realizadas en tiempo de máquina dedicado, es decir, sin ningún otro proceso usando recursos de la máquina; los datos recabados incluyen ejecuciones del programa paralelo usando desde 1 hasta 27 procesadores para cada uno de los tres métodos de reparto de carga descritos en la Sec. 4.3.4. El programa convergió después de 1466 iteraciones utilizando un perceptrón de 3 niveles con 3 unidades en el nivel de entrada, correspondientes a los valores de la posición y la rapidez del barco; 14 unidades en el nivel oculto y 2 unidades en el nivel de salida, asociadas a los

valores de las variables de control θ y a. El tiempo requerido para la convergencia del programa en la ejecución serial fue de 348.56 secs. Los valores (pesos) de las conexiones del perceptrón resultantes están listados en la Tabla 5.1, mientras que los valores de la función de error, Ec. (4.11), para cada iteración del algoritmo se muestran en la Fig. 5.2. Se puede observar en esta figura que el error no disminuye en forma monótona, sino por el contrario, toma valores muy cercanos a cero en algunos puntos, a partir de los cuales el error se incrementa casi monótonamente hasta unas cuantas iteraciones antes de la convergencia del algoritmo. En la Fig. 5.3 se muestran las posiciones (x_1, x_2) de las trayectorias generadas en el último ciclo del programa; como puede observarse, todas las trayectorias convergen al punto con coordenadas (1.0,0.5), el cual es el punto donde está localizado el muelle. Puede notarse que las trayectorias concentran una mayor cantidad de puntos en su parte final; como veremos más adelante, esta concentración de puntos puede explicarse por las variaciones de velocidad de las trayectorias. En la Fig. 5.4 se observa un ejemplo de una trayectoria generada utilizando los valores de las conexiones obtenidos después del entrenamiento de la red neuronal, cuyas coordenadas de posición y rapidez iniciales son $(0.3, 0.45)$ y 0.5, respectivamente. En dicha figura puede notarse con mayor claridad la concentración de puntos conforme el barco se acerca al muelle. Las Figs. 5.5, 5.6 y 5.7 muestran las variaciones de velocidad en esta misma trayectoria con respecto al tiempo discreto y a las posiciones en x_1 y x_2, respectivamente; en estas figuras puede observarse que la velocidad del barco disminuye progresivamente, de manera que al final de la trayectoria su magnitud es mínima, lo cual explica la concentración de puntos en esa región. En la primera de estas figuras puede observarse que la disminución de la velocidad es muy grande en el lapso comprendido entre 15 y 36 unidades de tiempo discreto.

Las Figs. 5.8 y 5.9 muestran el comportamiento en el tiempo de las variables de control θ y a, mientras que la Fig. 5.10 muestra los ángulos Θ correspondientes a la dirección de la trayectoria en cada paso de tiempo k. En principio, se podría pensar que los ángulos Θ y θ deben ser iguales. El siguiente análisis mostrará el origen de las diferencias entre ellos.

Los ángulos Θ fueron calculados usando la fórmula de la tangente del ángulo entre dos puntos con coordenadas (x_1, y_1), (x_2, y_2)

$$\tan\Theta = \frac{y_2 - y_1}{x_2 - x_1}; \tag{5.8}$$

por lo que el ángulo entre dos puntos consecutivos de la trayectoria estará dado por

$$\tan\Theta = \frac{x_2(k+1) - x_2(k)}{x_1(k+1) - x_1(k)}; \tag{5.9}$$

si sustituimos $x_2(k+1)$ y $x_1(k+1)$ utilizando las Ecs. (4.5) y (4.6) tendremos

$$\tan\Theta = \frac{[x_2(k) + x_3(k)\Delta t \sin\theta + f_c] - x_2(k)}{[x_1(k) + x_3(k)\Delta t \cos\theta] - x_1(k)}, \tag{5.10}$$

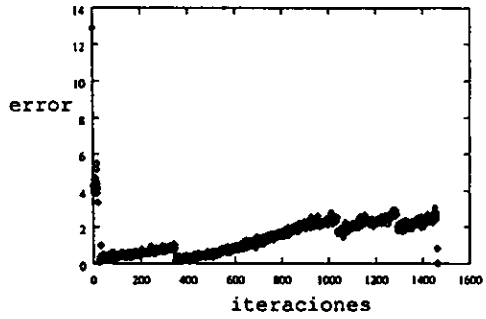

Figura 5.2: Valores de la función de error durante las diferentes iteraciones del programa.

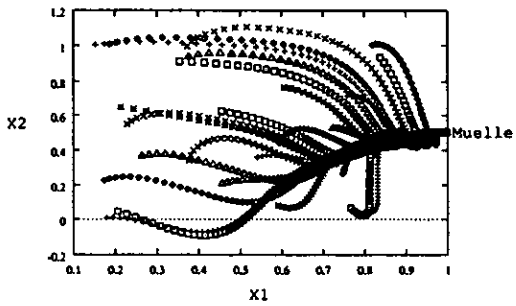

Figura 5.3: Trayectorias generadas en la última iteración del proceso de entrenamiento.

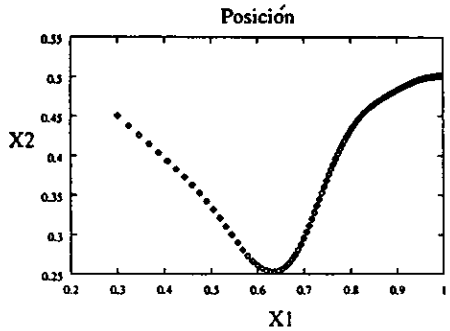

Figura 5.4: Ejemplo de trayectoria generada por el conjunto sistema dinámico - perceptrón una vez concluido el entrenamiento.

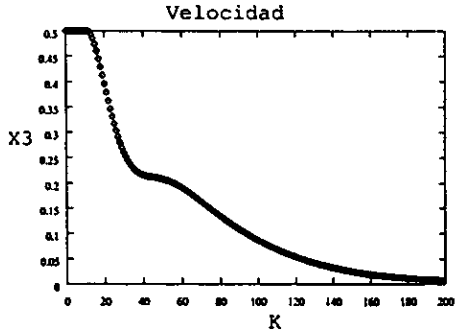

Figura 5.5: Velocidad del barco con respecto al paso de tiempo discreto k.

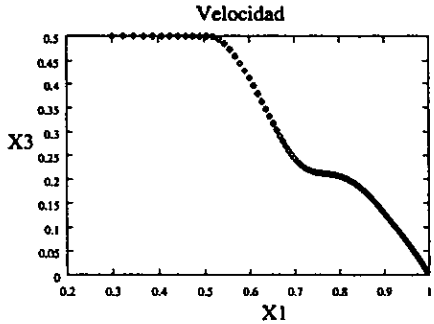

Figura 5.6: Velocidad del barco con respecto a su posición x_1.

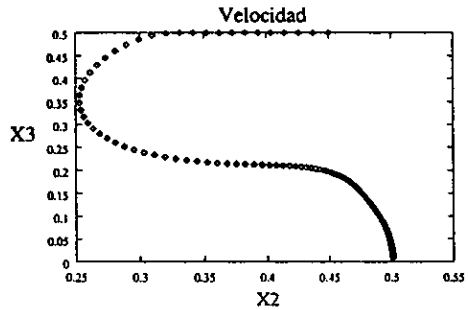

Figura 5.7: Velocidad del barco con respecto a su posición x_2.

donde θ es el ángulo generado por el perceptrón y f_c el efecto de la corriente del río; simplificando la última ecuación obtenemos

$$\tan\Theta = \frac{x_3(k)\Delta t \sin\theta + f_c}{x_3(k)\Delta t \cos\theta}; \tag{5.11}$$

de esta expresión podemos concluir que si el efecto de la corriente f_c fuese nulo entonces los ángulos Θ y θ serían iguales, de otra manera, existirá una diferencia entre ellos, como se puede notar en las Figs. 5.8 y 5.10.

5.4.2 Rendimiento del Programa Paralelo

El análisis de los resultados obtenidos en cuanto al rendimiento del programa paralelo se muestra a continuación. En las Figs. 5.11 a 5.13 se muestra el speed-up y la eficiencia del programa para las diferentes distribuciones de carga, considerando exclusivamente el tiempo total de ejecución empleado por la subrutina *trajec*; en cada una de las gráficas se incluyen los resultados de las estimaciones empíricas mostrados en la Fig. 5.1. Como puede observarse, los datos estimados y los medidos son prácticamente iguales, por lo que podemos concluir que la suposición de que el número de pasos en el tiempo de una trayectoria es proporcional a la carga de trabajo asociada a ésta es correcta, al menos dentro de la subrutina trajec. Las diferencias encontradas son debidas a una operación de sincronización, realizada mediante la llamada a la función mpi_barrier al final de la subrutina trajec.

Las figuras 5.14 y 5.15 muestran los speed-ups y las eficiencias obtenidas considerando el tiempo total de ejecución de todo el programa paralelo. El crecimiento en el speed-up se ve limitado y la eficiencia se reduce al 50% si se utilizan más de 8 procesadores, lo cual se debe a los costos de comunicación que, como se mencionó anteriormente, son independientes del método de distribución de carga utilizado; este hecho puede notarse en la Fig. 5.16, la cual muestra los tiempos de comunicación correspondientes a la función mpi_allreduce mediante la cual cada procesador obtiene la información de los gradientes parciales del error correspondientes a cada trayectoria. Estos tiempos de comunicación son proporcionales al $log_2(P)$, donde P es el número de procesadores que intervienen en la comunicación; dada esta proporcionalidad, es de suponerse que el algoritmo de comunicación empleado en la implementación de la función mpi_allreduce se basa en un algoritmo de *mariposa*[7]. Debido a que el tiempo empleado en comunicaciones es independiente del reparto de carga de trabajo utilizado y a que este tiempo es comparable al tiempo de cómputo, los valores del speed-up y las eficiencias se ven reducidos en comparación con los valores estimados, basados en la información de la generación de las trayectorias (Fig. 5.1), por lo que las ventajas de usar el método bin packing para el reparto de carga se ven disminuídas.

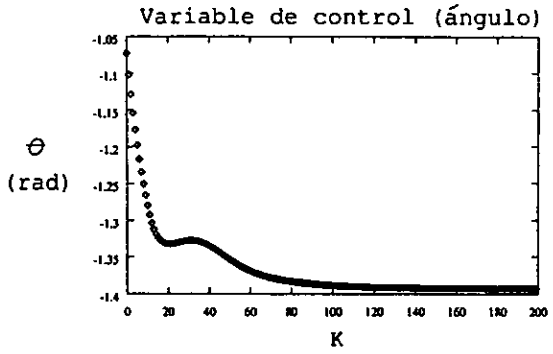

Figura 5.8: Ángulos generados por el perceptrón durante la trayectoria.

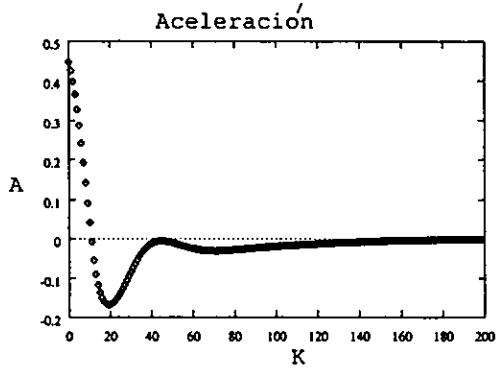

Figura 5.9: Magnitud de la aceleración generada por el perceptrón durante la trayectoria.

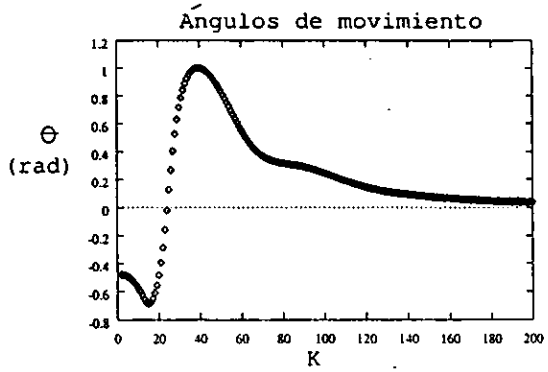

Figura 5.10: Ángulos calculados en la trayectoria generada

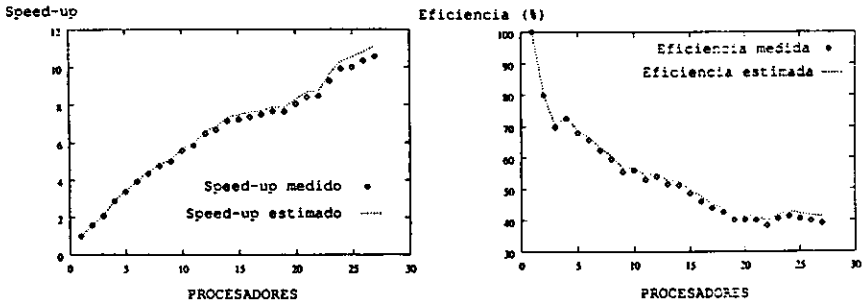

Figura 5.11: Método de bloque: speed-up y eficiencia considerando únicamente la subrutina trajec.

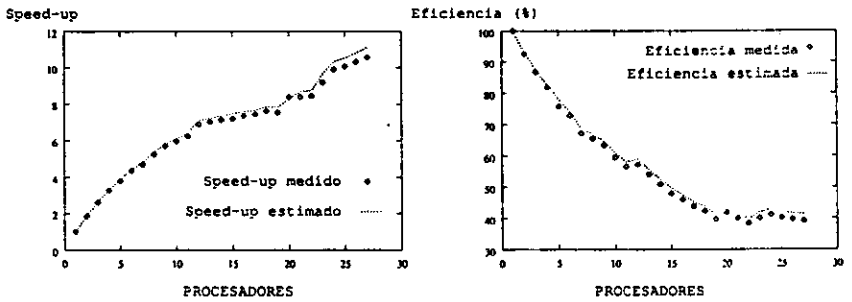

Figura 5.12: Método cíclico: speed-up y eficiencia considerando únicamente la subrutina trajec.

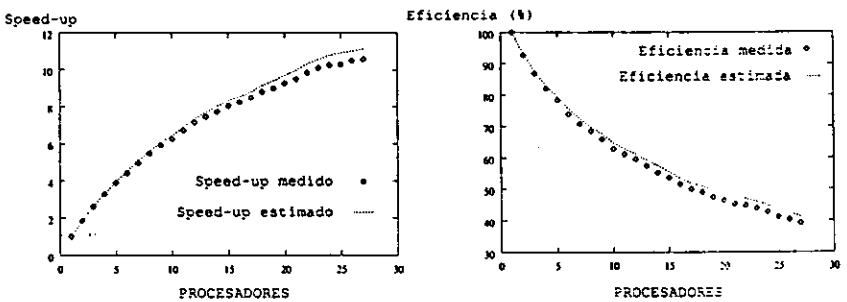

Figura 5.13: Método bin packing: speed-up y eficiencia considerando únicamente la subrutina trajec.

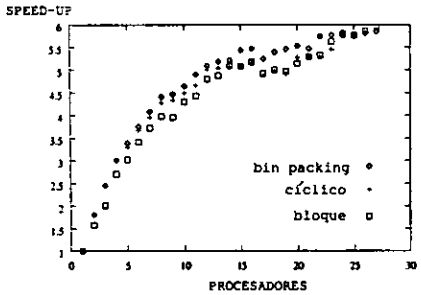

Figura 5.14: Speed-up total del programa paralelo.

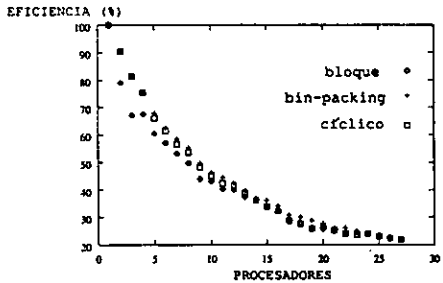

Figura 5.15: Eficiencia total del programa paralelo.

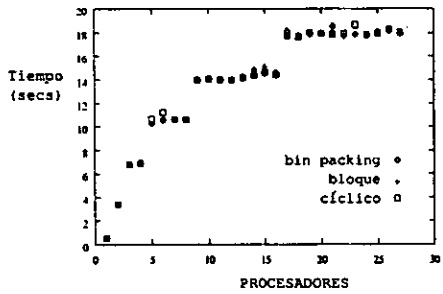

Figura 5.16: Tiempos totales de comunicación requeridos en todo el proceso de entrenamiento por la función mpi_allreduce utilizada para obtener los gradientes del error correspondientes a cada trayectoria. La operación realizada es mpi_sum utlizando 14400 datos del tipo mpi_real8

Conclusiones

A lo largo de este trabajo se han revisado los conceptos básicos tanto de las redes neuronales artificiales como del cómputo paralelo; estos conceptos han sido utilizados para diseñar e implementar un algoritmo paralelo de entrenamiento de un perceptrón multinivel a fin de obtener la solución a un problema específico de control.

En el primer capítulo se describieron brevemente las características de las redes neuronales artificiales partiendo de conceptos básicos, tales como el funcionamiento de las unidades básicas de cálculo, también conocidas como neuronas, con las cuales se contruyen estas redes y la forma en que estas unidades interactúan entre sí. La topología de conexión entre las unidades y las funciones que modelan la activación de éstas son los principales factores que diferencían los diversos tipos de redes neuronales artificiales existentes. A fin de ejemplificar estas diferencias, se analizaron brevemente la red de Hopfield, la cual es utilizada como memoria asociativa, y el perceptrón de Rosenblatt. A partir de este último se hizo un análisis del funcionamiento de los perceptrones multinivel y de su algoritmo de entrenamiento, el cual es conocido como algoritmo de retropropagación. A su vez, se revisaron las aplicaciones de los perceptrones multinivel a la solución de problemas de control de sistemas dinámicos discretos, revisión que incluyó al algoritmo de retropropagación en el tiempo. Este algoritmo fue utilizado, junto con el modelo perceptrón-sistema dinámico descrito en la Sec. 1.6, como base para la construcción de un algoritmo de entrenamiento que permite al perceptrón resolver un problema de control.

En el segundo capítulo se revisaron los conceptos básicos de cómputo paralelo, los cuales incluyen las características de las máquinas paralelas, tales como su estructura de memoria, los procesadores y las diversas topologías de conexión entre éstos. En ese mismo capítulo se discutieron los parámetros que caracterizan las topologías de conexión y se obtuvieron dichos parámetros para las topologías básicas: bus, malla, crossbar e hipercubo. A partir de estos datos se pueden deducir las ventajas que ofrecen los hipercubos sobre las demás topologías, y por consecuencia, explicar el por qué de su popularidad.

Los principios básicos de la programación paralela fueron discutidos en la Sec. 2.3. Estos principios definen un programa paralelo como un conjunto de procesos,

cada uno ejecutándose en un procesador diferente, y que actúan en forma cooperativa para obtener la solución a un problema específico. Estos principios también establecen las dos operaciones básicas de la programación paralela: intercambio de información y sincronización de los procesos que intervienen en el programa paralelo. En la Sec. 2.4 se revisaron las características de algunas máquinas paralelas que usan el modelo de intercambio de mensajes para implementar su ambiente de programación. Esta revisión se hizo con el fin de observar como la evolución de estos ambientes ha proporcionado nuevos conceptos que han facilitado cada vez más el trabajo de los programadores y ha permitido la construcción de programas cada vez más complejos y eficientes. Este capítulo finaliza con una revisión de los ambientes portátiles de intercambio de mensajes. Las aportaciones principales de los ambientes de intercambio de mensajes tanto nativos como portátiles han dado forma al MPI, el ambiente estándar de programación bajo este modelo de cómputo paralelo.

El capítulo 3 contiene una descripción del MPI; se discuten las estructuras de datos que son utilizadas por éste y los conceptos que definen el modo de operación de las funciones del MPI que implementan tanto las operaciones básicas de envío y recepción de mensajes como las operaciones colectivas, las cuales involucran un conjunto de procesos.

En el capítulo cuatro se utilizaron los conceptos revisados anteriormente con el objeto de construir un algoritmo paralelo de entrenamiento de un perceptrón. Con este algoritmo se encontró la solución a un problema de control que consiste en la conducción de un objeto móvil hacia un punto prestablecido. Primero se hizo una descripción del sistema dinámico del cual surge el problema de control y se obtuvo un modelo matemático, el cual fue utilizado dentro de la configuración perceptrón-sistema dinámico descrita en la Sec. 1.6. Después se describió el método de entrenamiento usado, el cual se basa en el algoritmo de retropropagación en el tiempo; posteriormente se hizo el diseño de la versión paralela del algoritmo de entrenamiento. Este diseño se basó en un método de 3 pasos: partición, comunicación y aglomeración. En cada uno de estos pasos de diseño se describieron las diferentes opciones existentes, se analizó cada una de éstas y se decidió cual sería la más conveniente. La partición se realizó considerando a las celdas en que se divide el espacio fase del problema como la estructura de datos a dividir. Esta decisión fue acertada, pues el speed-up del programa se comporta de acuerdo a las estimaciones hechas considerando esta división. Las comunicaciones en el algoritmo son globales, puesto que todos los procesos necesitan la misma información para realizar sus operaciones. Como consecuencia de la utilización de este modelo de comunicación, el valor máximo del speed-up del programa se ve limitado a un valor máximo de 6, mientras que la eficiencia se reduce al 50% si se utilizan más de 8 procesadores. En contraste, los valores obtenidos en las estimaciones realizadas usando una subrutina del programa, indican que el speed-up puede alcanzar un valor de 12 y que la eficiencia será menor del 50% si se utilizan más de 13 procesadores. En el caso de la aglomeración, se decidió implementar tres esquemas diferentes de

reparto de carga con el fin de hacer un análisis comparativo entre ellos. Al realizar la comparación utilizando las estimaciones en el rendimiento y la eficiencia, se concluyó que el método bin packing es el método de reparto de carga de trabajo más eficiente, pues proporciona un speed-up mayor o igual que los otros métodos, para cualquier número de procesadores que se utilice. En las mediciones del speed-up y la eficiencia del programa paralelo completo, estas ventajas se ven disminuídas debido a que el tiempo de comunicación es independiente del método de reparto de carga utilizado y a que su valor es comparable al del tiempo de cómputo.

Debido a que el acceso a arquitecturas paralelas es cada día más fácil, es necesario diseñar programas capaces de aprovechar eficientemente estos recursos. El objetivo de un programa paralelo es reducir significativamente el tiempo de espera para la obtención de resultados. Estos programas pueden ser construídos relativamente fácil usando MPI, ya que este ambiente de programación provee las operaciones fundamentales mediante funciones sencillas de usar; además, un programa escrito en MPI es completamente portátil, esto es, un mismo código fuente puede utilizarse en cualquier tipo de arquitectura, incluyendo conjuntos de computadoras conectadas por una red local de comunicaciones, tal como ethernet.

Apéndice A

VELOCIDAD DE INTERCAMBIO DE MENSAJES EN EL AMBIENTE DE CÓMPUTO PARALELO MPI

En este apéndice se presentan los resultados obtenidos de la medición de los parámetros que caracterizan el tiempo necesario para la trasmisión de mensajes entre dos procesadores en el ambiente de intercambio de mensajes MPI. Estas mediciones fueron realizadas en tres arquitecturas diferentes: una SGI Origin 2000 con 40 procesadores R10000 a 195 MHz de la DGSCA-UNAM, una Sun Sparc Server 1000 con 4 procesadores a 55 MHz del ICN-UNAM y un sistema de dos estaciones de trabajo, una SGI Indigo con un procesador MIPS R4000 a 100 MHz y una SGI con 2 procesadores R3000A a 40 MHz, conectadas vía ethernet, pertenecientes al Laboratorio de Visualización de la DGSCA-UNAM.

Introducción

Uno de los parámetros que determinan el rendimiento de un programa paralelo es el *speed-up*, el cual se define como el tiempo de ejecución del programa en forma secuencial T_1 dividido entre el tiempo de ejecución en paralelo T_p, es decir T_1/T_p, donde p es el número de procesadores que intervienen en el cómputo. El tiempo de ejecución en paralelo T_p está definido como el tiempo transcurrido desde que el primer procesador inicia la ejecución del programa hasta que el último procesador termina de trabajar. Durante este tiempo cada procesador se encuentra ya sea realizando algún cálculo, enviando o recibiendo mensajes u ocioso; de tal forma, para un procesador arbitrario, podemos determinar el tiempo de ejecución

mediante la suma de tres tiempos, esto es:

$$T_p = T_{cal} + T_{com} + T_{ocio}. \tag{A.1}$$

El tiempo T_{cal} está determinado por el programa y la distribución de la carga entre los procesadores, T_{com} está determinado por la estructura de comunicación utilizada y T_{ocio} por el balance de la carga de trabajo, principalmente. Como podemos observar, el tiempo en las comunicaciones es uno de los parámetros que influyen en el tiempo de ejecución de un programa paralelo y por lo tanto en su speed-up, debido a lo cual es necesario construir modelos cuantitativos que permitan la determinación del tiempo de comunicación en un programa paralelo.

El tiempo de trasmisión de un mensaje puede caracterizarse mediante dos parámetros: el tiempo necesario para iniciar la comunicación (tiempo de *Start-up*) T_s y el tiempo requerido para la trasmisión de un byte de información entre dos procesadores una vez establecida la comunicación, T_w; de tal forma, el tiempo de transmisión de un mensaje de una longitud de l bytes puede ser modelado por la siguiente ecuación:

$$T_{com} = T_s + T_w l. \tag{A.2}$$

Descripción del experimento

Para obtener los parámetros T_s y T_w que caracterizan el tiempo de transmisión de un mensaje se realizaron mediciones del tiempo transcurrido durante el intercambio de mensajes de diferentes longitudes, empezando con un mensaje de 25 enteros y variando su longitud en incrementos de 25 en 25 enteros hasta llegar a 5000. Los experimentos fueron realizados entre las 3:00 y las 5:00 horas de cada día, debido a que en este horario las máquinas utilizadas para los experimentos permanecen prácticamente desocupadas y por lo tanto no existe competencia por el uso de sus recursos, lo que reduce significativamente las fluctuaciones en las mediciones debidas a factores externos; para asegurar aún más esta condición, se ejecutó el programa usando el comando *renice* de Unix con un valor de -20 a fin de dar máxima prioridad a los procesos creados para la realización del experimento. El tiempo de trasmisión fue medido utilizando la función *MPI_WTIME* de MPI. Para cada longitud de mensaje se realizaron 100 mediciones, tomando como dato final el promedio de éstas después de eliminar la mayor y la menor. De este proceso fueron obtenidos un conjunto de pares (X_i, Y_i) donde Y_i corresponde al tiempo medido en segundos para la transmisión del mensaje de longitud X_i en enteros. Estos puntos fueron ajustados a una recta cuya pendiente representa el tiempo de trasmisón por byte T_w y la ordenada al origen representa el tiempo de Startup T_s. Las ecuaciones de ajuste son:

$$T_w = \frac{N\sum_i X_i Y_i - \sum_i X_i \sum_i Y_i}{N\sum_i X_i^2 - \sum_i X_i \sum_i X_i}, \tag{A.3}$$

$$T_s = \frac{\sum_i X_i^2 \sum_i Y_i - \sum_i X_i \sum_i X_i Y_i}{N\sum_i X_i^2 - \sum_i X_i \sum_i X_i}. \tag{A.4}$$

En las gráficas correspondientes a las arquitecturas Origin 2000 y Sparc Server de la Fig. 1.1 podemos distinguir dos regiones bien definidas en el comportamiento del tiempo de trasmisión T_{comm}. Esta división en regiones es debida a un cambio en el modo de comunicación utilizado por el MPI. En los experimentos correspondientes a la arquitectura Sparc Server se han analizado las 2 regiones de manera independiente, mientras que para la arquitectura Origin 2000 únicamente se han utilizado los datos correspondientes a la primera región.

Una vez obtenidos los valores de T_w y T_s, se determinó la incertidumbre asociada utilizando las siguientes fórmulas:

$$\sigma_{T_s}^2 = \frac{\sigma_y^2 \sum_i X_i^2}{N\sum_i X_i^2 - \sum_i X_i \sum_i X_i}, \tag{A.5}$$

$$\sigma_{T_w}^2 = \frac{N\sigma_y^2}{N\sum_i X_i^2 - \sum_i X_i \sum_i X_i}; \tag{A.6}$$

en donde

$$\sigma_y^2 = \frac{1}{(N-2)} \sum_i (Y_i - T_s - T_w X_i)^2. \tag{A.7}$$

Este proceso se repitió 50 veces en cada una de las tres arquitecturas. Con los 50 datos obtenidos se calcularon los promedios pesados $< T_s >$ y $< T_w >$ mediante la fórmula

$$< Z >= \frac{\sum_j Z_j/\sigma_j^2}{\sum_j 1/\sigma_j^2}, \tag{A.8}$$

en donde Z_j es el j-ésimo valor de Z y σ_j^2 es la varianza asociada a Z_j. La varianza asociada a $< Z >$ esta dada por

$$\sigma_z^2 = \frac{1}{\sum_j 1/\sigma_j^2}. \tag{A.9}$$

En la Fig. A.1 se muestran los resultados típicos de un experimento para cada una de las arquitecturas, en donde se puede observar el tiempo de trasmisión medido para mensajes de distintas longitudes; también se puede observar la existencia de distintas regiones en las arquitecturas Origin 2000 y Sparc Server, correspondientes a modos distintos de comunicación. De la Tabla A.1 a la Tabla A.4 se muestran los valores de tiempo de trasimisión por byte T_w y de tiempos de *Startup* T_s, así como sus desviaciones correspondientes σ_w y σ_s, calculados en cada uno de los 50 experimentos realizados en las tres arquitecturas. Las Fig.s A.2- A.5 muestran las gráficas de distribución de los datos presentados en las tablas A.1-A.4.

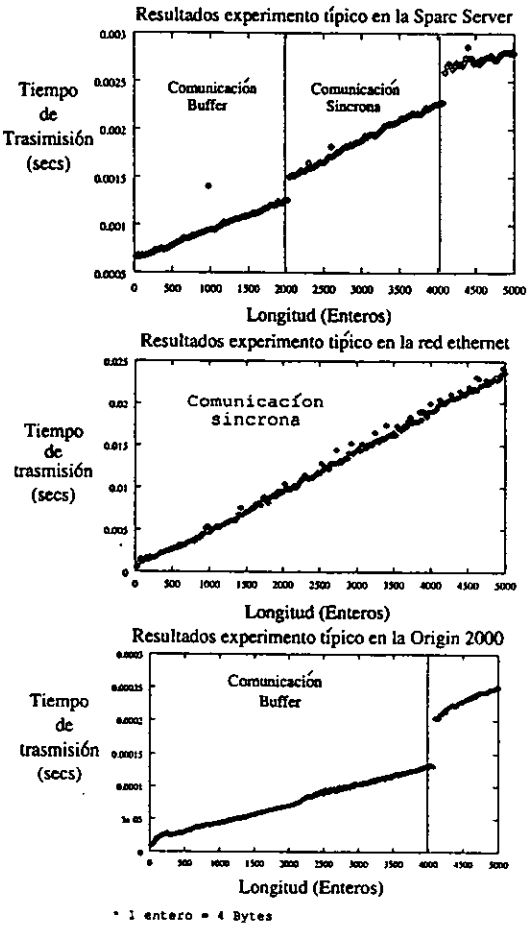

Figura A.1: Gráficas de la medición del tiempo de intercambio de mensajes en un experimento típico para las arquitecturas Sparc Server (Figura superior), Red ethernet (Figura Intermedia) y Origin 2000 (Figura inferior).

Tabla A.1: Datos obtenidos para la red ethernet

T_w (Bytes/seg)	σ_{t_w} (Bytes/seg)	T_s (sec)	σ_{t_s} (sec)
1.23960E-06	6.54116E-09	2.28141E-04	7.58140E-05
1.22968E-06	6.57899E-09	3.38496E-04	7.62524E-05
1.28033E-06	1.20332E-08	6.36386E-04	1.39468E-04
1.20108E-06	9.45320E-09	7.70971E-04	1.09565E-04
1.15001E-06	7.28102E-09	6.23898E-04	8.43892E-05
1.22246E-06	6.34836E-09	3.25658E-04	7.35794E-05
1.23264E-06	6.92564E-09	3.44975E-04	8.02702E-05
1.22249E-06	9.46487E-09	2.96455E-06	1.09700E-04
1.28881E-06	8.62334E-09	2.94642E-04	9.99471E-05
1.28132E-06	7.62205E-09	1.35380E-04	8.83419E-05
1.27994E-06	1.20815E-08	5.17024E-04	1.40028E-04
1.18798E-06	4.81389E-09	9.77381E-05	5.57945E-05
1.18355E-06	4.77033E-09	1.73593E-04	5.52896E-05
1.17128E-06	4.64069E-09	2.95834E-04	5.37870E-05
1.17327E-06	4.31436E-09	2.62666E-04	5.00048E-05
1.18058E-06	5.27574E-09	2.25988E-04	6.11474E-05
1.17480E-06	4.64721E-09	2.39720E-04	5.38625E-05
1.14932E-06	6.85676E-09	5.88235E-04	7.94719E-05
1.17844E-06	4.54772E-09	2.15788E-04	5.27095E-05
1.17477E-06	4.59562E-09	2.22037E-04	5.32646E-05
1.17282E-06	4.48766E-09	2.43191E-04	5.20133E-05
1.17314E-06	5.09209E-09	2.60496E-04	5.90188E-05
1.17253E-06	5.02288E-09	2.18906E-04	5.82167E-05
1.18470E-06	4.86926E-09	9.74524E-05	5.64362E-05
1.18205E-06	4.97576E-09	2.57590E-04	5.76705E-05
1.16826E-06	4.86322E-09	2.40973E-04	5.63662E-05
1.19107E-06	7.59101E-09	1.42636E-04	8.79821E-05
1.17841E-06	4.75767E-09	2.37898E-04	5.51429E-05
1.16697E-06	4.45576E-09	2.55106E-04	5.16436E-05
1.18698E-06	4.83944E-09	2.14430E-04	5.60906E-05
1.18502E-06	4.35659E-09	1.08235E-04	5.04942E-05
1.16943E-06	3.55688E-09	1.53566E-04	4.12253E-05
1.21866E-06	9.93215E-09	8.29798E-05	1.15116E-04
1.17257E-06	3.31492E-09	2.17334E-04	3.84209E-05
1.13920E-06	7.80993E-09	6.56607E-04	9.05194E-05
1.16322E-06	3.99160E-09	2.59341E-04	4.62638E-05
1.16013E-06	3.18455E-09	1.98787E-04	3.69098E-05
1.17013E-06	5.48704E-09	1.50709E-04	6.35964E-05
1.16551E-06	3.25261E-09	2.04530E-04	3.76987E-05
1.16888E-06	6.77225E-09	1.88533E-04	7.84924E-05
1.17030E-06	3.70411E-09	3.47623E-04	4.29318E-05
1.16842E-06	4.65890E-09	1.31017E-04	5.39981E-05
1.16024E-06	3.04085E-09	1.86511E-04	3.52444E-05
1.16898E-06	4.36711E-09	1.60680E-04	5.06161E-05
1.15738E-06	2.96427E-09	1.62039E-04	3.43567E-05
1.17030E-06	3.00362E-09	1.79102E-04	3.48129E-05
1.16213E-06	2.92819E-09	1.72721E-04	3.39386E-05
1.16700E-06	3.18534E-09	1.24280E-04	3.69191E-05
1.16233E-06	5.36918E-09	1.80269E-04	6.22304E-05
1.15841E-06	3.17966E-09	1.82582E-04	3.68532E-05

Tabla A.2: Datos obtenidos para la arq. Origin 2000

T_w (Bytes/seg)	σ_{t_w} (Bytes/seg)	T_s (sec)	σ_{t_s} (sec)
5.84862E-09	3.20792E-11	1.52530E-05	2.97724E-07
5.80055E-09	3.94313E-11	1.61571E-05	3.65958E-07
6.73507E-09	3.81494E-11	1.39200E-05	3.54060E-07
5.94636E-09	4.26016E-11	1.46524E-05	3.95381E-07
6.31449E-09	5.69131E-11	1.46895E-05	5.28204E-07
5.96544E-09	3.40234E-11	1.54597E-05	3.15767E-07
6.68794E-09	3.38537E-11	1.40456E-05	3.14193E-07
7.53983E-09	5.56910E-11	1.37489E-05	5.16862E-07
7.48296E-09	5.63191E-11	1.35776E-05	5.22692E-07
7.20219E-09	3.75590E-11	1.58247E-05	3.48581E-07
7.62837E-09	4.32208E-11	1.52688E-05	4.01128E-07
6.55032E-09	4.63266E-11	1.23157E-05	4.29952E-07
6.05504E-09	3.30228E-11	1.37738E-05	3.06481E-07
6.14458E-09	4.31592E-11	1.36063E-05	4.00556E-07
7.26459E-09	7.01388E-11	1.10990E-05	6.50950E-07
6.18587E-09	2.99394E-11	1.46849E-05	2.77865E-07
6.02043E-09	4.54325E-11	1.30235E-05	4.21654E-07
7.44809E-09	4.99381E-11	1.46528E-05	4.63470E-07
6.95415E-09	9.90743E-11	1.21904E-05	9.19498E-07
6.30588E-09	4.18392E-11	1.31324E-05	3.88305E-07
6.81991E-09	2.39338E-11	1.29609E-05	2.22127E-07
5.90076E-09	3.36259E-11	1.52136E-05	3.12078E-07
5.82668E-09	5.84436E-11	1.54227E-05	5.42409E-07
7.11972E-09	4.43369E-11	1.14271E-05	4.11486E-07
5.98733E-09	3.28182E-11	1.42743E-05	3.04582E-07
5.92188E-09	5.03519E-11	1.75597E-05	4.67310E-07
7.02018E-09	5.71983E-11	1.74892E-05	5.30852E-07
6.95415E-09	9.90743E-11	1.21904E-05	9.19498E-07
6.71060E-09	9.62178E-11	1.23966E-05	8.92987E-07
7.42785E-09	5.93030E-11	1.67175E-05	5.50385E-07
6.93969E-09	2.08442E-10	1.70761E-05	1.93453E-06
7.19823E-09	4.26732E-11	1.20187E-05	3.96045E-07
6.01459E-09	5.55570E-11	1.88520E-05	5.15619E-07
5.87867E-09	5.57656E-11	2.01348E-05	5.17554E-07
7.48893E-09	1.99197E-10	1.72564E-05	1.84872E-06
7.17094E-09	6.42906E-11	1.24862E-05	5.96674E-07
6.61897E-09	9.14088E-11	1.46011E-05	8.48355E-07
7.99405E-09	1.41330E-09	1.18394E-05	1.31167E-05
6.04974E-09	4.22354E-11	1.52475E-05	3.91983E-07
6.65076E-09	7.12481E-11	1.99834E-05	6.61246E-07
7.25887E-09	4.11479E-11	1.50698E-05	3.81889E-07
5.82525E-09	3.46769E-11	1.53361E-05	3.21833E-07
6.48283E-09	3.99506E-11	1.63425E-05	3.70777E-07
6.08781E-09	4.95660E-11	1.39715E-05	4.60017E-11
6.53856E-09	3.88960E-11	1.61438E-05	3.60990E-07
5.84862E-09	3.20792E-11	1.52530E-05	2.97724E-07
6.80244E-09	5.93163E-11	1.61100E-05	5.50508E-07
6.87997E-09	4.75607E-11	1.54402E-05	4.41406E-07
6.62985E-09	3.81672E-11	1.72342E-05	3.54225E-07
6.80306E-09	2.36054E-11	1.27598E-05	2.19079E-07

Tabla A.3: Datos obtenidos para la arq. Sparc Server modo Buffer

T_w (Bytes/seg)	σ_{t_w} (Bytes/seg)	T_s (sec)	σ_{t_s} (sec)
8.87180E-08	6.75671E-10	6.25851E-04	3.15003E-06
8.35116E-08	8.34386E-10	7.10452E-04	3.88998E-06
1.02598E-07	2.20350E-09	6.97418E-04	1.02729E-05
7.64471E-08	2.58010E-09	6.47619E-04	1.20286E-05
8.28140E-08	8.71878E-10	6.61459E-04	4.06476E-06
7.94452E-08	7.57570E-10	7.28111E-04	3.53185E-06
1.25475E-07	5.51406E-09	5.21577E-04	2.57070E-05
8.43088E-08	5.05132E-10	6.71848E-04	2.35496E-06
8.60759E-08	6.83068E-10	6.76892E-04	3.18452E-06
8.32800E-08	1.96143E-09	6.80817E-04	9.14436E-06
1.22326E-07	7.82626E-09	5.35473E-04	3.64866E-05
8.31738E-08	4.42100E-09	5.89588E-04	2.06110E-05
7.71368E-08	1.60675E-08	6.96925E-04	7.49080E-05
9.45670E-08	8.02952E-09	6.15875E-04	3.74343E-05
7.00168E-08	1.58809E-08	7.06612E-04	7.40381E-05
1.18347E-07	1.75479E-09	6.57499E-04	8.18095E-06
8.98243E-08	2.57678E-09	6.78313E-04	1.20131E-05
7.82464E-08	8.08767E-10	6.27876E-04	3.77054E-06
1.00398E-07	2.64980E-09	6.55690E-04	1.23536E-05
1.06726E-07	1.70259E-09	5.48362E-04	7.93762E-06
8.49343E-08	1.25104E-09	7.12719E-04	5.83245E-06
7.61589E-08	8.18398E-10	6.48770E-04	3.81544E-06
6.53215E-08	9.44999E-10	7.48830E-04	4.40566E-06
1.06234E-07	1.55230E-09	6.22769E-04	7.23694E-06
9.21726E-08	5.76702E-10	6.10468E-04	2.68863E-06
9.26222E-08	5.20889E-09	6.30759E-04	2.42843E-05
1.17502E-07	7.74197E-09	5.44060E-04	3.60937E-05
8.72376E-08	1.48435E-09	5.88208E-04	6.92015E-06
8.88035E-08	9.51602E-10	6.19981E-04	4.43645E-06
1.08156E-07	3.00326E-09	8.32681E-04	1.40014E-05
8.56785E-08	8.33951E-09	6.97311E-04	3.88795E-05
9.31849E-08	1.40245E-09	6.46935E-04	6.53835E-06
8.92123E-08	9.99084E-10	7.17179E-04	4.65781E-06
8.67922E-08	7.60337E-10	6.76637E-04	3.54475E-06
9.52522E-08	1.16792E-09	6.24940E-04	5.44492E-06
8.17328E-08	5.80290E-10	7.10978E-04	2.70536E-06
9.04931E-08	9.81231E-10	6.56904E-04	4.57458E-06
8.51871E-08	1.22140E-09	6.81617E-04	5.69427E-06
7.48539E-08	2.21149E-09	7.37452E-04	1.03101E-05
1.03254E-07	2.77784E-09	7.23353E-04	1.29505E-05
9.34236E-08	1.43667E-09	6.37746E-04	6.69788E-06
9.07467E-08	1.44735E-09	7.49097E-04	6.74765E-06
9.12526E-08	3.43336E-09	6.03833E-04	1.60066E-05
9.80738E-08	2.49677E-09	7.89344E-04	1.16401E-05
8.23613E-08	1.41839E-09	7.81828E-04	6.61264E-06
8.27493E-08	6.14234E-10	7.07263E-04	2.86361E-06
8.23281E-08	3.65686E-09	6.99847E-04	1.70486E-05
9.20159E-08	1.03334E-09	7.18323E-04	4.81752E-06
1.05149E-07	8.47884E-10	7.23233E-04	3.95291E-06
7.52626E-08	2.97623E-09	7.07689E-04	1.38754E-05

Tabla A.4: Datos obtenidos para la arq. Sparc Server modo Síncrono

T_w (Bytes/seg)	σ_{t_w} (Bytes/seg)	T_s (sec)	σ_{t_s} (sec)
9.40540E-08	2.35171E-09	7.49941E-04	2.86229E-05
9.34803E-08	2.62955E-09	8.81335E-04	3.20045E-05
1.30140E-07	3.06643E-09	6.68879E-04	3.73218E-05
1.04127E-07	1.94825E-09	6.28594E-04	2.37123E-05
1.19407E-07	3.50049E-09	5.72583E-04	4.26047E-05
1.60634E-07	2.89782E-09	2.08199E-04	3.52695E-05
7.50300E-08	8.94814E-10	1.04180E-03	1.08908E-05
9.74000E-08	9.25008E-09	8.19000E-04	1.12583E-04
1.17345E-07	2.17075E-09	6.40457E-04	2.64204E-05
1.10272E-07	3.16997E-09	6.64521E-04	3.85819E-05
1.33859E-07	3.00703E-09	4.69352E-04	3.65987E-05
8.97265E-08	3.08293E-09	7.73817E-04	3.75226E-05
1.04152E-07	7.90174E-09	7.40051E-04	9.61727E-05
9.82638E-08	8.47235E-09	8.17950E-04	1.03118E-04
1.05560E-07	2.04673E-09	6.83721E-04	2.49109E-05
1.27591E-07	4.63445E-09	8.50888E-04	5.64062E-05
9.09938E-08	4.32714E-09	8.32019E-04	5.26660E-05
9.26141E-08	1.84081E-09	7.19996E-04	2.24046E-05
9.52013E-08	2.48573E-09	9.11430E-04	3.02540E-05
1.12508E-07	2.76670E-09	6.54061E-04	3.36737E-05
6.86528E-08	2.94485E-09	1.14254E-03	3.58419E-05
1.08872E-07	2.42457E-09	5.71788E-04	2.95096E-05
1.04385E-07	5.92816E-09	6.82239E-04	7.21521E-05
9.82565E-08	2.88572E-09	9.43299E-04	3.51223E-05
1.21031E-07	2.56426E-09	5.24774E-04	3.12098E-05
1.01096E-07	5.20063E-09	8.46050E-04	6.32973E-05
9.77103E-08	2.78585E-09	8.57924E-04	3.39068E-05
1.09345E-07	2.37092E-09	6.46226E-04	2.88567E-05
1.03389E-07	1.79178E-09	6.72304E-04	2.18078E-05
1.06267E-07	2.95716E-09	1.10183E-03	3.59918E-05
1.03031E-07	2.25644E-09	7.35835E-04	2.74633E-05
8.34505E-08	1.70748E-09	8.90172E-04	2.07818E-05
1.14238E-07	3.86002E-09	7.24564E-04	4.69805E-05
8.67231E-08	2.30287E-09	8.88401E-04	2.80284E-05
8.78073E-08	2.15114E-09	9.12840E-04	2.61817E-05
7.27917E-08	2.09414E-09	1.02597E-03	2.54880E-05
9.40210E-08	4.32101E-09	8.86725E-04	5.25914E-05
9.86626E-08	2.40406E-09	8.03030E-04	2.92600E-05
8.50200E-08	3.44714E-09	9.16947E-04	4.19553E-05
9.26974E-08	2.67616E-09	1.01927E-03	3.25718E-05
1.32840E-07	4.20088E-09	6.07160E-04	5.11292E-05
8.66073E-08	2.09479E-09	1.00874E-03	2.54958E-05
1.03200E-07	2.10667E-09	8.05686E-04	2.56404E-05
8.98698E-08	3.75744E-09	1.14790E-03	4.57321E-05
1.12888E-07	3.70297E-09	7.72662E-04	4.50692E-05
1.11374E-07	7.27189E-09	6.34040E-04	8.85067E-05
8.93026E-08	2.30712E-09	8.11938E-04	2.80802E-05
1.00901E-07	2.81259E-09	8.91402E-04	3.42322E-05
1.26950E-07	2.04805E-09	7.09971E-04	2.49269E-05
1.12916E-07	7.54996E-09	5.98368E-04	9.18911E-05

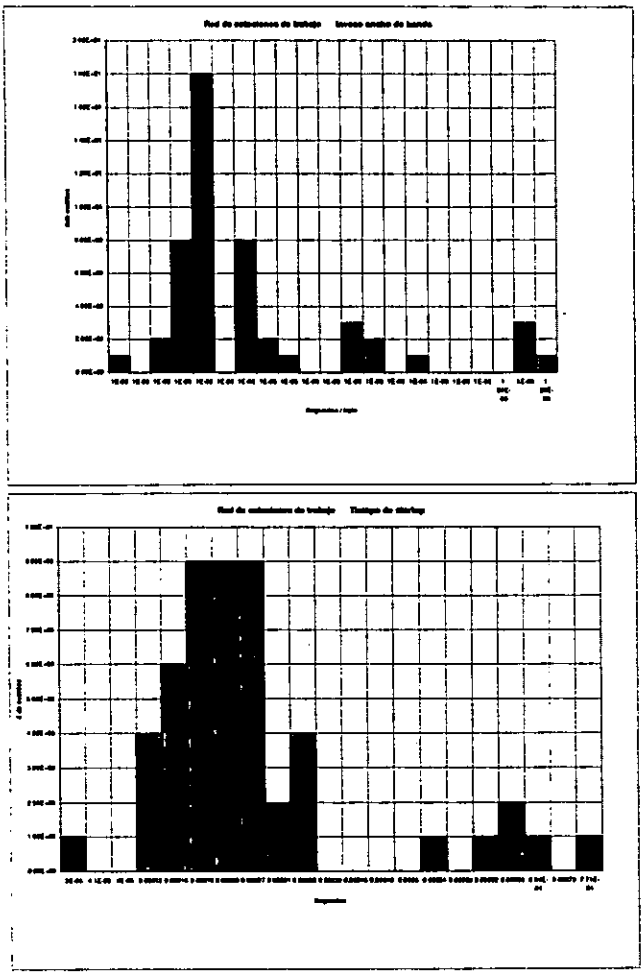

Figura A.2: Gráficas de distribución de los datos obtenidos para la red ethernet.

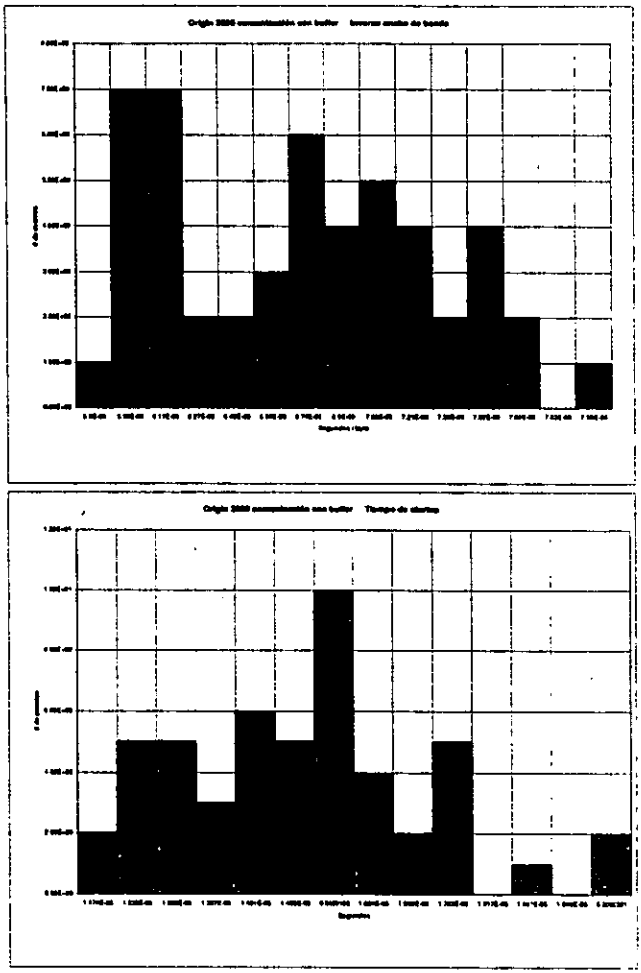

Figura A.3: Gráficas de distribución de los datos obtenidos para la Origin 2000.

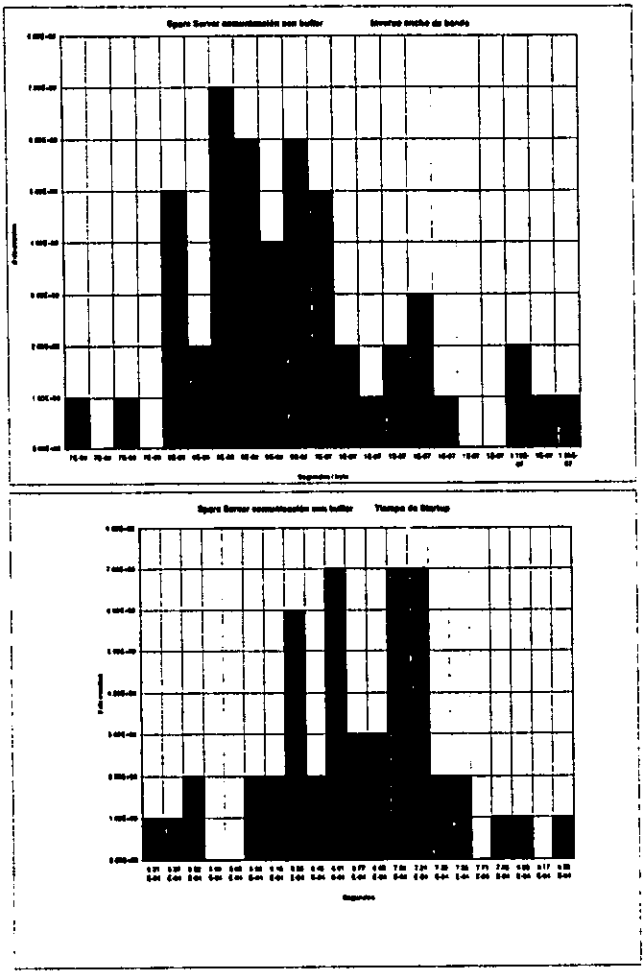

Figura A.4: Gráficas de distribución de los datos obtenidos en la Sparc Server modo Buffer.

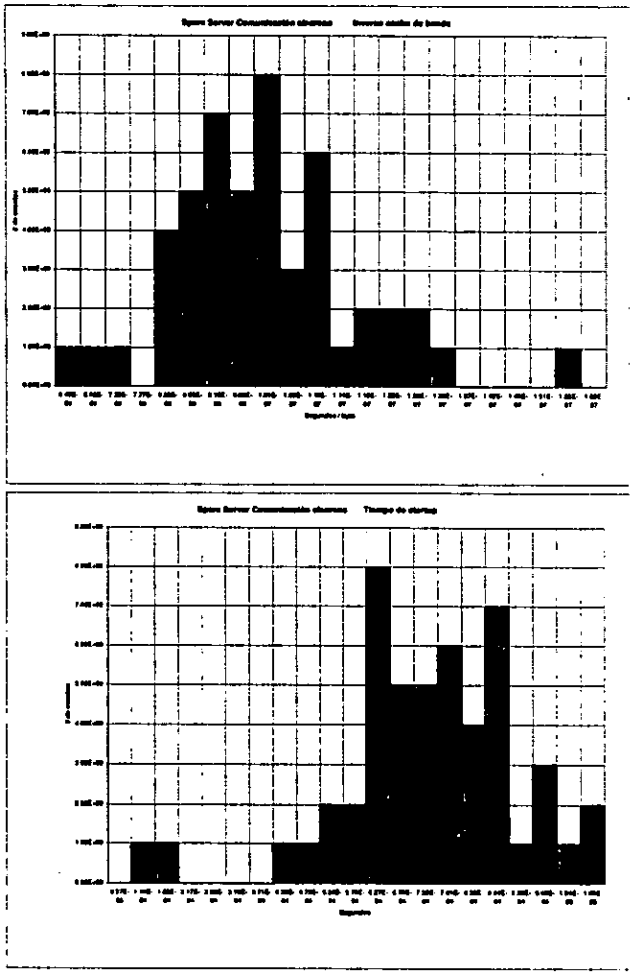

Figura A.5: Gráficas de distribución de los datos obtenidos para la Sparc Server modo Síncrono.

ARQUITECTURAS	ORIGIN 2000		SPARC SERVER		SPARC SERVER		RED ETHERNET	
	40 PROCESADORES MIPS		4 PROCESADORES		4 PROCESADORES		ESTACIONES DE TRABAJO	
	R10000 A 195 MHz		A 50 MHz		A 50 MHz		SILICON GRAPHICS	
	COMUNICACION BUFFER		COMUNICACION BUFFER		COMUNICACION SINCRONA		INDIGO	
RESULTADOS	PROMEDIO	DESV ES	PROMEDIO	DESV ES	PROMEDIO	DESV ES	PROMEDIO	DESV ES
STARTUP (MICROSEGUNDOS)	14.62	0.06	675.7	0.73	819.59	4.32	215.74	7.5
TIEMPO POR BYTE (NANOSEGUNDOS / BYTE)	6.51	0.00615	86.54	0.18	97.47	0.36	1174.3	0.65
ANCHO DE BANDA (MBYTES / SEGUNDO)	153.61	14.51	11.55	0.021	10.26	0.037	0.85	0.00047

* 1 Byte = 8 bits

Figura A.6: Valores calculados de los parámetros T_s y T_w para las 3 arquitecturas

Resultados

En la Fig. A.6 se presentan los valores $< T_w >$ y $< T_s >$ calculados para cada una de las arquitecturas. Como podemos deducir, la arquitectura Origin 2000 tiene el tiempo de *Startup* (14.62 μs) y el tiempo de trasmisión por byte (6.51 ηs) más pequeños entre las 3 arquitecturas, mientras que la red de estaciones de trabajo presenta un tiempo de Startup (215.74 μs) menor que la Sparc Server (675.7 μs modo buffer y 819.6 μs modo síncrono) aunque esta última posee un tiempo de trasmisión por byte mucho menor (86.54 ηs y 97.47 ηs en los modos buffer y síncrono respectivamente contra 1174.3 ηs en la red ethernet). También podemos observar que el tiempo de trasmisión por byte en los dos modos de comunicación analizados para la Sparc Server es muy similar ($\approx$ 90 ηs), mientras que el tiempo *Startup* es significativamente menor en el modo Buffer. Podemos concluir entonces que la diferencia entre los tiempos de comunicación de ambos modos esta dada por el parámetro T_s, es decir, por el tiempo necesario para establecer la comunicación.

En la misma tabla podemos hacer una comparación de la eficiencia de las comunicaciones entre las 3 arquitecturas en términos de su ancho de banda, es decir, la cantidad de información por segundo que pueden trasmitir. La Origin 2000 posee el mayor ancho de banda (153 Mbytes/sec), seguida de la Sparc Server (11.55 Mbytes/sec y 10.26 Mbytes/sec para los modos buffer y síncrono, respectivamente). Por último, la red ethernet posee un ancho de banda casi 200 veces menor que el de la Origin 2000, esto es, 0.85 Mbytes/sec.

En la Fig. A.7 se muestran dos comparaciones gráficas entre los tiempos de comunicación para mensajes de distintas longitudes calculados de acuerdo a los parámetros $< T_w >$ y $< T_s >$ obtenidos para cada arquitectura. En la gráfica superior podemos observar como la red ethernet realiza las comunicaciones en forma más eficiente que la Sparc Server para mensajes de longitud menor a 450 bytes, debido a tiene un tiempo de *Startup* más pequeño; sin embargo, para mensajes de longitud mayor a 450 bytes el tiempo de trasmisión T_w es menor para la Sparc Server. El tiempo de trasmisión de la Origin 2000 es prácticamente despreciable en comparación con el requerido en las otras dos arquitecturas.

Los parámetros T_s y T_w, junto con un análisis de la estructura de comunicación a utilizar que determine el número y longitud de los mensajes necesarios para la ejecución de un programa dado, son útiles en el estudio teórico de la eficiencia de algoritmos paralelos en una arquitectura específica..

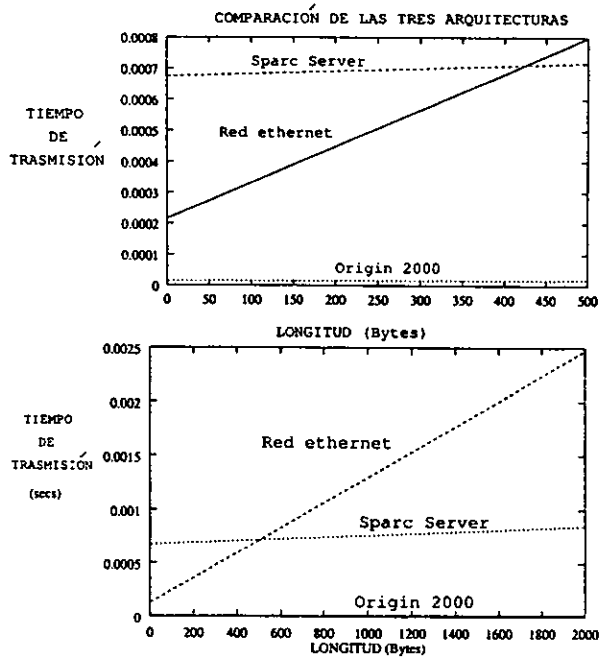

Figura A.7: Comparación gráfica de los tiempos de trasmisión estimados para las arquitecturas Sparc Server, Red ethernet y Origin 2000

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCC
CCCCCCCCC   PROGRAMA REALIZADO POR JOSE LUIS GORDILLO RUIZ
CCCCCCCCC   ICN-UNAM
CCCCCCCCC   SEPTIEMBRE de 1997
CCCCCCCCC
CCCCCCCCC   ESTE PROGRAMA TOMA VARIAS MEDIDAS DEL TIEMPO
CCCCCCCCC   TRANSCURRIDO DURANTE EL ENVIO DE UN MENSAJE
CCCCCCCCC   PARA VARIAS LONGITUDES DE DATOS
CCCCCCCCC
CCCCCCCCC   DESCRIPCION DE LAS VARIABLES UTILIZADAS
CCCCCCCCC   datos(2,200) matriz en donde se guarda la longitud del
CCCCCCCCC   mensaje y el tiempo necesario para completarlo
CCCCCCCCC   temp(100) matriz en donde se guarda las distintas mediciones
CCCCCCCCC   hechas sobre una misma longitud
CCCCCCCCC   trec(2,200) matriz en donde se guarda los datos correspondientes
CCCCCCCCC   a los tiempos de recepcion (utilizada solo por el proceso 0)


        include "/usr/include/mpif.h"
DOUBLE PRECISION datos(2,200),trec(2,200),res(2)
        dimension matriz(10000),matriz2(10000)
        integer status(5)
        DOUBLE PRECISION Tick, dinicio,fin,rmax,rmin
        DOUBLE PRECISION temp(100)
        DOUBLE PRECISION total,prom

ccccccc inicializamos la matriz

       do i=1,10000
          matriz(i)=i
       enddo
ccccccccccc  inicializamos el ambiente paralelo

        call mpi_init(ierror)

        call mpi_comm_rank(MPI_COMM_WORLD,mirank,ierror)

        Tick=mpi_wtick(ierror)
        if (mirank.eq.0) print *,"#el tick es de ",tick


cccccc numero de datos que vamos a recolectar.
```

```
 do i=1,200
          n=i*25

ccccc  numero de medidas sobre el mismo dato.

         do j=1,100

           if (mirank.eq.1) then

ccccc  medimos el tiempo del mensaje solamente

              dinicio=MPi_Wtime(ierror)
              call MPI_SEND(matriz,n,mpi_integer,
     *                  0,0,mpi_comm_world,ierror)
              fin=mpi_wtime(ierror)
           else
              dinicio=MPI_WTIME(ierror)
              call mpi_recv(matriz2,n,mpi_integer,
     *           1,0,mpi_comm_world,status,ierror)
              fin=mpi_wtime(ierror)
           endif

ccccc  calculamos el tiempo transcurrido

           temp(j)=fin-dinicio
         enddo

cccc eliminamos el menor y el mayor

         rmax=-1000000.0
         rmin=10000000.0
         imax=0
         imin=0
         do j=1,100
            if(temp(j).le.rmin) then
               rmin=temp(j)
               imin=j
            endif
           if(temp(j).ge.rmax) then
               rmax=temp(j)
               imax=j
            endif
         enddo
         temp(imax)=0
```

```
         temp(imin)=0

ccccc obtenemos el promedio
         total=0
         do j=1,100
           total=total+temp(j)
         enddo
         prom=total/(98.0)
         datos(1,i)=n
         datos(2,i)=prom

ccccc siguiente longitud de datos
       enddo


cccccc trasladamos al proceso 0 los valores medidos para la
cccccc  recepcion.
       if (mirank.eq.0) then
          call MPI_RECV(trec,400,mpi_double_precision,
     *          1,0,mpi_comm_world,status,ierror)
       else
          call MPI_SEND(datos,400,mpi_double_precision,
     *          0,0,mpi_comm_world,ierror)
       end if

ccccccccc imprimimos los resultados

       if (mirank.eq.0) then
       print *,"#    long          envio        recepcion
     * diferencia"

        do i=1,200
             print *,datos(1,i),datos(2,i),trec(2,i)
        enddo

       endif
       call MPI_FINALIZE(ierror)
       end
```

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCC
CCCCCCC   PROGRAMA DE AJUSTE DE DATOS
CCCCCCC
CCCCCCC   ELABORADO POR JOSE LUIS GORDILLO RUIZ
CCCCCCc   ICN-UNAM
CCCCCCC   SEPTIEMBRE DE 1997
CCCCCCC
CCCCCCC   ESTE PROGRAMA TOMA UN CONJUNTO DE DATOS (X,Y) Y OBTIENE
CCCCCCC   LOS PARAMETROS m (pendiente) y b (ordenada al origen)
CCCCCCC   DE UNA RECTA A PARTIR DEL CONJUNTO DE DATOS, UTILIZANDO
CCCCCCC   EL METODO DE LOS MINIMOS CUADRADOS
CCCCCCC   LOS DATOS CORRESPONDIENTES A X SE ENCUENTRAN EN ENTEROS (4 BYTES)
CCCCCCC   LOS DATOS CORRESPONDIENTES A Y SE ENCUENTRAN EN SEGUNDOS
CCCCCCC
CCCCCCC
CCCCCCC   DESCRIPCION DE VARIABLES

CCCCCCC   send(3,200)  :Almacena 200 puntos (x,y) en los 2 primeros
CCCCCCC                renglones.
CCCCCCC    res(4)      :Almacena los siguientes datos: pendiente,
CCCCCCC         ordenada, error en la pendiente, error en
CCCCCCC        la ordenada

      double precision send(3,200),res(4)

ccccccccc  Leemos los datos de la entrada estandar ...
      call lee(send,200,3)

cccccccc  lms es la rutina de ajuste
      call lms(send,160,res)

ccccccc  se imprimen los datos en el orden: pendiente, error en la
ccccccc  pendiente, ordenada, error en la ordenada.

      print *,"1 m=",res(1)*.25," sigm=",sqrt(res(3))*.25
      print *,"2 b=",res(2)," sigb=",sqrt(res(4))
      end

ccccccccccccccccc Esta subrutina lee los datos de la entrada estandar
ccccccccccccccccc m  : numero de datos que tiene cada tupla
ccccccccccccccccc n  : numero de tuplas
```

```
      subroutine lee(a,n,m)
      double precision a(m,n)
      do j=1,n
      read (*,5) (a(i,j),i=1,m)
      enddo
5     format(3f)
      end

ccccccccccccc  Esta subrutina calcula los parametros de la recta.
ccccccccccccc
ccccccccccccc
ccccccccccccc   x = sumatoria de todas las x's.
ccccccccccccc   y = sumatoria de todas las y's
ccccccccccccc   x2= sumatoria de todas las x^2's.
ccccccccccccc   xy= sumatoria de los productos xy.
ccccccccccccc   sigma = varianza de y definida como
ccccccccccccc   sigma = 1/(n-2)sum (y - (b+mx))^2
ccccccccccccc  (donde b=ordenada y m=pendiente calculadas).
cccccccccccc
cccccccccccc     n = numero de datos (x,y)

      subroutine lms(a,n,res)
      double precision a(3,n), res(4)
      double precision x,y,xy,x2,f,sigma

ccccccccccc inicializamos

      x2=0.0
      x=0.0
      y=0.0
      xy=0.0

ccccccccccc  calculamos las sumatorias
      do i=1,n
         x2=x2+a(1,i)*a(1,i)
         x=x+a(1,i)
         y=y+a(2,i)
         xy=xy+a(1,i)*a(2,i)
      enddo

cccccccc  calculamos la pendiente m
      res(1)=(n*xy-x*y)/(n*x2-x*x)

cccccccc  calculamos la ordenada b
```

```
      res(2)=(x2*y-x*xy)/(n*x2-x*x)

      sigma=0.0


ccccccccc  calculamos la varianza de y
      do i=1,n
         f=res(2)+res(1)*a(1,i)
         sigma=sigma+(a(2,i)-f)**2
      enddo

      sigma=sigma/(n-2.0)

cccccccc   calculamos los errores en b y en m
      res(4)=sigma*x2/(n*x2-x*x)
      res(3)=sigma*n/(n*x2-x*x)
      end
```

Bibliografía

[1] J. Hertz, A. Krogh and R. Palmer. *Introduction to the Theory of Neural Computation.* Addison Wesley, Vol I. 1991.

[2] R. Lippmann. *An Introduction to Computing with Neural Nets.* IEEE ASSP Magazine, pp 4-22, Abril 1992.

[3] J.J. Hopfield. *Neural Networks and Physical Systems with Emergent Collective Computational Abilities.* Proc. Natl. Sci. USA, Vol 79. April 1982.

[4] B. Widrow and M. Lehr. *30 Years of Adaptative Neural Networks: Perceptron, Madaline and Backpropagation.* Proceedings of the IEEE. Vol 7E, No. 9. September 1990.

[5] D.E. Rumelhart, G.E. Hinton and R.J. Williams. *Learning Internal Representations by Error Propagation.* Parallel and distributed processing: Explorations in the microstructure of cognition. Vol I. MIT Press. 1986.

[6] S. W. Piché. *Steepest Descent Algorithms for Neural Network Controllers and Filters.* IEEE Transactions on Neural Netrworks, Vol 1, March 1994.

[7] I. Foster. *Designing and Building Parallel Programs.* Addison Wesley. Reading. Massachussets. 1994.

[8] O. A. McBryan. *An Overview of Message Passing Environments.* Parallel Computing. 20(1994), pp 417-444.

[9] *The Message Passing Interface (MPI) Standard.* http://www.mcs.anl.gov/mpi/index.html

[10] MPI: A Message-Passing Interface Standard. Message Passing Interface Forum. Ver 1.1. University of Tennesse. Knoxville. 1995.

[11] *Top 10 Reasons to Prefer MPI over PVM.* http://www.osc.edu/Lam/mpi.

[12] *Departamento de Supercómputo, DGSCA-UNAM.* http://www.super.unam.mx.

[13] *Origin 2000 Products.* http://www.sgi.com/origin/2000.

[14] W. S. Dorn and H. J. Greenberg. *Matemáticas y Computación con programación Fortran.* Limusa. México 1976.

[15] G. E. Hinton. *How Neural Networks Learn From Experience.* Scientific American, pp 145-151. September 1992.

[16] W. Gropp, E. Lusk and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Pres. Massachussets, 1994.

[17] P. R. Bevington and D. K. Robinson. *Data Reduction And Error Analysis For the Physics Sciences.* McGraw-Hill, NY 1992.