UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES

"A C A T L A N"

SIMULACION DEL PROCESO PARALELO POR MEDIO DE MAQUINAS DE TURING

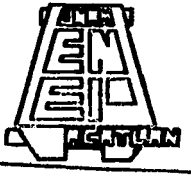
T E S I S

QUE PARA OBTENER EL TITULO DE:

LICENCIADO EN MATEMATICAS APLICADAS Y COMPUTACION

P R E S E N T A

GABRIELA MAGOS SOSA



NAUCALPAN, EDO. DE MEXICO 1993

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

TESIS

MAL

COMPA

GINADA

INDICE

OBJETIVO GENERAL

Simular en una computadora normal una serie de máquinas de Turing para implementar técnicas de procesamiento paralelo, con especial énfasis en las instrucciones que permiten distribuir ordenar y colectar los datos para éste tipo de procesos.

INTRODUCCION

Siempre se ha considerado que todas las acciones que realiza el hombre se desarrollan en forma "secuencial", como lo es la comunicación, la cual es una *secuencia* de palabras expresadas en forma escrita u oral. Aún así, muchas de estas acciones se realizan al mismo tiempo en diferentes lugares, por ejemplo, dentro de una empresa la comunicación entre los jefes y los empleados se da en los diferentes niveles mas o menos al mismo tiempo, es decir en forma paralela.

Ahora bien, desde la construcción de las primeras computadoras se ha aplicado el concepto de secuencia para su operación. Sin embargo, el procesamiento paralelo se está considerando como el futuro de la computación, aunque este futuro está tardando más tiempo del que se esperaba. Una razón de esta tardanza es que las arquitecturas y algoritmos paralelos no son tan bien entendidos como los secuenciales y son aún objeto de investigación.

La introducción al campo de trabajo de máquinas paralelas en sus muy diversas formas, que van desde los procesos paralelos en software hasta la forma extrema de computadoras totalmente independientes trabajando sobre un mismo problema; requieren un nuevo paradigma o forma de entender la disciplina de la computación tanto en su teoria como en su práctica.

Este trabajo va dirigido a todas aquellas personas y principalmente a los estudiantes de la carrera de Matemáticas Aplicadas y Computación, que teniendo ya conocimientos de programación avanzada, deseen conocer y experimentar con las

técnicas de programación paralela, ya que éstas son cada día más solicitadas, sobre todo en el área de la investigación.

En el presente escrito se analizan las diferentes formas en que se puede obtener el paralelismo dentro de las computadoras, por ejemplo *las computadoras de segmentación encauzada*, las cuales exploran el paralelismo temporal mediante computaciones solapadas; *los sistemas multiprocesadores*, los cuales logran el paralelismo al trabajar con un conjunto de procesadores que disponen de recursos compartidos (memoria, impresora, bases de datos, etc); *los sistemas multicomputadores*, los cuales son un conjunto de procesdores con su propia memoria, que trabajan en paralelo comunicándose con los demás procesadores a través de una red.

Dentro de los sistemas multicomputadores podemos encontrar diferentes formas de conexión o topologías, desde la topología más sencilla en la cual las computadoras se conectan en línea, hasta la topología en la que cada computadora esta conectada con otras tres (*hipercubo*), la cual es más eficiente.

Así como el paralelismo se puede obtener a nivel de arquitectura es también posible obtenerlo a nivel de programación, para lo cual es necesario conocer las técnicas que permiten la creación de procesos superpuestos en el tiempo, en este trabajo se ha tomado como muestra una extención del lenguaje de programación Pascal que permite, con ciertas instrucciones especiales, la creación de procesos paralelos simulados en una computadora normal.

Para demostrar las ventajas de la computación paralela se han desarrollado dos aplicaciones: la primera es la resolución de operaciones artméticas sencillas utilizando las bases de la computación, la máquina de Turing. Aunque en los años 30's se desarrollaron varias teorías que trataban en principio un mismo

tema (las funciones recursivas), se ha elegido la teoría de Turing ya que es una estructura general con la que se puede representar y resolver casi cualquier problema si se definen las instrucciones correctas.

La segunda es la optimización de una función por medio de una técnica no muy usada pero si muy eficiente llamada Algoritmo Genético, la cual toma principios de la evolución natural de los seres vivos y la aplica en la exploración del espacio de solución del problema, ésta técnica se implementó en paralelo para demostrar la eficiencia de la combinación de ambas.

I. Procesamiento paralelo vs procesamiento serial

I.1 *¿POR QUE LA PROGRAMACIÓN PARALELA?*

La "*secuencia*" es uno de los muchos aspectos de la actividad humana y de la ley natural. El lenguaje humano es secuencial y el conocimiento es expresado y transmitido como una secuencia de palabras, ya sea hablado o escrito. El concepto del tiempo, el cual es vital para la planeación y ejecución de acciones exitosas, está basado en el concepto de secuencia.

Por esto, es natural que los algoritmos y los programas para computadora fueran formulados en un principio de acuerdo con el concepto de "*secuencia*". Aún antes de que la primera computadora fuera construida, el concepto matemático de un "algoritmo" estaba definido como una "secuencia" finita de operaciones [6].

Sin embargo, como la ciencia computacional ha madurado gradualmente, ha ido aclarando, sobre todo en los últimos diez años, que lo *secuencial* es sólo una parte de la historia. Las actividades humanas y las leyes naturales no son sólo secuenciales, sino también altamente paralelas: las acciones no sólo se desarrollan secuencialmente, sino las acciones ocurren simultaneamente en cualquier lado al mismo tiempo. El *paralelismo* es tan importante y fundamental como lo secuencial.

Los individuos pueden hablar y actuar secuencialmente, pero los individuos son parte de organizaciones, las cuales consisten de muchos individuos trabajando todos en forma paralela. El mismo principio abarca las actividades de la naturaleza: el desarrollo de acciones y ciclos dentro de la naturaleza son ciertamente secuenciales, pero la naturaleza funciona en cualquier lugar al mismo tiempo con ilimitado *paralelismo*. Así, una imagen completa de la actividad humana y de la acción de las leyes de naturaleza en el universo requieren fuertemente de ambos conceptos: *secuencial* y *paralelo*.

I.2 *UN POCO DE HISTORIA*

A través del tiempo, y desde que se construyó la primera computadora, éstas han ido evolucionando y mejorando su forma de trabajo para lograr el máximo de eficiencia con un mínimo de costo tanto económico (costo del equipo) como material (tamaño y cantidad de los componentes).

Al principio estas mejoras se dieron a nivel de *hardware*: de los bulbos a los transistores, éstos cambiaron por los circuitos integrados, después llegaron los circuitos de pequeña, mediana y alta escala de integración (SSI, MSI, y LSI respectivamente) y en los 90's tienen auge los VLSI (chips integrados de muy alta escala) con este dasarrollo, poco a poco el tamaño de las computadoras se ha ido reduciendo al mismo tiempo que su capacidad se ha incrementado.

En un principio las computadoras eran programadas por medio del lenguaje de máquina codificado en binario, cuando en 1945-46 se crea la primera computadora de programa almacenado (EDVAC), se marca el inicio del uso del *software* para sistemas y a medida que los lenguajes se fueron desarrollando, son cada vez más accesibles a ser programados por el usuario.

I.3 *PROCESAMIENTO SERIAL*

Ahora bien, desde el punto de vista del sistema operativo, las computadoras han mejorado cronológicamente en cuatro fases:

- Procesamiento por lotes
- Multiprogramación
- Tiempo compartido
- Multiprocesamiento

El **procesamiento por lotes** era el modo de operación más normal durante la época entre 1952 y 1963, proporcionando una ejecución secuencial de los programas de usuario, es muy eficiente en aquellas aplicaciones que requieren que la información se organice y procese en un cierto orden. Este tipo de proceso tambien se conoce como:

> *Monoprogramación:* "Sistema de explotación de una computadora en la que se ejecuta un solo trabajo en un momento dado y no se puede ejecutar otro hasta que no se haya terminado el anterior"[1]

La **multiprogramación** se desarrolló entre 1965 y 1975 con el propósito de permitir la ejecución simultánea de muchos segmentos de programas intercalados con operaciones de Entrada/Salida.

Los sistemas operativos de **tiempo compartido** estuvieron disponibles a finales de los 60's, el concepto es una extensión de la multiprogramación que asigna *intervalos* fijos o variables de tiempo a múltiples programas, es decir, proporciona igualdad de oportunidades a todos los programas que compiten por el uso de la Unidad Central de Proceso.

Las computadoras actuales tienden al **multiprocesamiento** que es la ejecución de dos o más programas o secuncia de instrucciones en una computadora de forma que cada uno conserve su identidad, ésto puede lograrse por medio de la multiprogramación, el procesamiento paralelo o con ambos.

---

[1] Galan Farrilla Jesús. "Sistemas Operativos y Compiladores". McGraw Hill. España. 1988. pag. 35.

I.4 *PROCESAMIENTO PARALELO*

En los cuatro modos de operación mencionados anteriormente (por lotes, multiprogramación, tiempo compatido y multiprocesamiento), el grado de paralelismo se incrementa rápidamente de fase en fase. Formalmente definimos *procesamiento paralelo* como sigue:

Definición "El procesamiento paralelo es una forma eficaz de procesamiento de información que favorece la explotación de los sucesos concurrentes en el proceso de computación. Concurrencia implica paralelismo, simultaneidad y solapamiento. Los sucesos paralelos son los que pueden producirse en diferentes recursos durante el mismo intervalo de tiempo; los sucesos simultáneos son los que pueden producirse en el mismo instante de tiempo; los sucesos solapados son los que pueden producirse en intervalos de tiempo superpuestos. Estos sucesos concurrentes pueden darse en un sistema computador en varios niveles de procesamiento. El procesamiento paralelo exige la ejecución concurrente en la computadora de muchos programas."[2]

El proceso paralelo se puede dar en cuatro niveles que son:

1.-*Nivel de Programación o Trabajos:* el cual se logra por medio de la multiprogramación, el tiempo compartido y el multiprocesamiento. La implementación de algoritmos paralelos depende de la asignación eficaz de limitados recursos de software-hardware a los múltiples programas que estén siendo utilizados para resolver un extenso problema de cálculo.

---

[2] Hwang, Kai y Briggs, Fayé A. "Arquitectura de Computadoras y procesamiento paralelo". McGraw Hill, México, 1988 pag 7.

2.- *Nivel de procedimientos o tareas:* esto supone la descomposición de un programa en múltiples tareas.

3.- *Nivel interinstrucciones:* trata de explotar la concurrencia entre múltiples instrucciones. Con frecuencia, se realiza un análisis de dependencia de datos para revelar paralelismos entre instrucciones.

4.- *Nivel intrainstrucción:* es deseable disponer de operaciones más rápidas y concurrentes dentro de cada instrucción.

El nivel superior (de trabajos) se aborda a menudo algoritmicamente. El nivel inferior (intrainstrucción) se implementa con frecuencia directamente por medios hardware. La participación de hardware se va incrementando desde los niveles altos hasta los bajos. Contrariamente, las implementaciones de software se incrementan desde los niveles bajos hasta los altos. El balance entre las técnicas hardware y software para resolver un problema es siempre un asunto muy controvertido. La tendencia está respaldada también por la creciente demanda de tiempo-real más rápido y la compartición de recursos.

Las características anteriores sugieren que el procesamiento paralelo es ciertamente un campo combinado de estudios, su último objetivo es alcanzar alto rendimiento a menor coste al realizar tareas de computación científica a gran escala en las diferentes áreas de aplicación.

La mayoría de los fabricantes de computadoras comenzaron por el desarrollo de sistemas con un solo procesador central, denominados *sistemas monoprocesador*. La potencia de cálculo en estos sistemas puede incrementarse si se permite el uso de elementos de procesamiento paralelo bajo el mando de un controlador. También se puede ampliar la estructura de la

computadora para incluir procesadores múltiples con espacio de memoria y periféricos compartidos bajo control de un sistema operativo integrado. A este tipo de computadoras se le denomina *sistema multiprocesador*.

Por lo que se refiere al procesamiento paralelo, la tendencia general de la arquitectura se ha desplazado desde los sistemas monoprocesadores convencionales hasta sistemas multiprocesadores.

I.5 *RENDIMIENTO DE LAS COMPUTADORAS PARALELAS.*

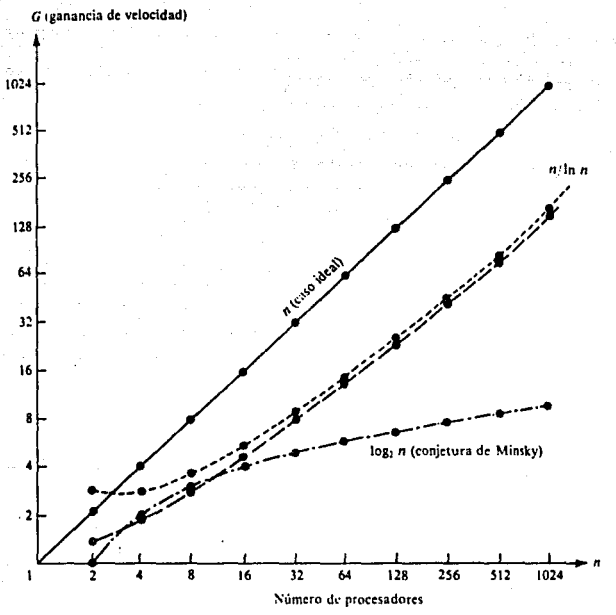
La ganancia de velocidad que puede conseguir una computadora paralela con n procesadores idénticos trabajando concurrentemente en un solo problema es como máximo n veces superior a la de un procesador único.

Claro que esto es lo ideal, pero en la práctica la ganancia es menor, ya que algunos procesadores permanecen inactivos en algunos instantes debido a conflictos en los accesos a memoria, en los caminos de datos o al uso de algoritmos ineficaces en la explotación de la concurrencia natural del problema que se computa.

En la figura 1.1 se muestran diferentes estimaciones de la ganancia real de velocidad, que van desde una cota inferior de $\log_2 n$ hasta una cota superior de $n/\ln n$.

La cota inferior $\log_2 n$ es conocida como la *conjetura de Minsky*. Aplicando ésta, sólo cabe esperar una ganancia de velocidad de 2 a 4 para los actuales multiprocesadores de 4 a 16 procesadores. Una estimación más optimista de la ganancia de velocidad está limitada superiormente por la expresión $n/\ln n$.

Fig. 1.1



Diferentes estimaciones de la ganancia de velocidad de un sistema de n-procesadores. [12]

II. Diferentes clases y tipos de máquinas para procesamiento paralelo

NO
EXISTE
PAGINA

II.1 *TIPOS DE MÁQUINAS*

Las computadoras paralelas son aquellos sistemas que favorecen el procesamiento paralelo, de los diferentes tipos que existen, tomaremos como muestra las siguientes:

- **Computadoras de Segmentación Encauzada:** las cuales explotan el paralelismo temporal mediante computaciones solapadas, (Vectorización, PIPELINE, Multifeching).

- **Sistemas Multiprocesadores:** alcanzan paralelismo asíncrono gracias a un conjunto de procesadores interactivos que disponen de recursos compartidos (memoria, base de datos, impresora, etc.).

- **Sistemas Multicomputadores:** los cuales logran paralelismo asíncrono al trabajar cada procesador con su propia memoria y comunicándose con los demás procesadores a través de una red de comunicación.

Aún dentro de los sistemas monoprocesadores se puede lograr el paralelismo por medio de algunos mecanismos, por ejemplo:

Multiplicidad de Unidades Funcionales: Las primeras computadoras disponían de una sola unidad aritmético-lógica (UAL) en la Unidad Central de Proceso. Además, la UAL sólo podía realizar una función cada vez, proceso demasiado lento. En la práctica muchas de las funciones de la UAL pueden estar distribuidas sobre múltiples unidades funcionales especializadas que pueden operar simultáneamente. Se emplea un *marcador* para registrar la disponibilidad de las unidades y los registros que se soliciten.

Segmentación encauzada de la Unidad Central de Proceso: Las diferentes fases de ejecución de las instrucciones se fraccionan en etapas entre las que se incluyen: la extracción de la

instrucción, su decodificación, la extracción del operando, la ejecución aritmético-lógica y el almacenamiento del resultado. Estas etapas o segmentos se conectan en cascada formando un cauce. Para facilitar las ejecuciones solapadas de instrucciones a través del cauce, se han desarrollado técnicas de preextracción de instrucciones y de memorización intermedia de datos.

II.2 *COMPUTADORAS DE SEGMENTACIÓN ENCAUZADA*

La segmentación encauzada ofrece un modo económico de realizar paralelismo temporal en las computadoras digitales. El concepto de procesamiento encauzado dentro de una computadora es similar al de las líneas de montaje en una planta industrial.

Para conseguir la segmentación encauzada, debe subdividirse la tarea (el proceso) de entrada en una secuencia de subtareas, cada una de las cuales puede ser ejecutada por una etapa hardware especializada que actúe en concurrencia con otras etapas del encauzamiento. Las tareas sucesivas circulan dentro del cauce y van ejecutándose en modo solapado a nivel de subtarea.

La subdivisión del trabajo en las líneas de montaje ha contribuido al éxito de la producción en masa en la industria moderna. Por la misma razón, el procesamiento encauzado ha conducido a una tremenda mejora en la productividad de la computadora digital moderna.

Como ya se mencionó, la ejecución de una instrucción en una computadora digital, normalmente, implica cuatro pasos principales: *búsqueda de la instrucción (BI)* desde la memoria principal; *decodificación de la instrucción (DI)*, para identificar la operación a efectuar; *búsqueda del operando (BO)*, si es preciso; y *ejecución (EJ)* de la operación aritmético-lógica decodificada. En una computadora NO-encauzada, estos pasos deben

finalizar antes de procesar la instrucción siguiente. En una computadora encauzada las instrucciones sucesivas se ejecutan en modo solapado. En la figura 2.1 se pueden ver estas etapas dispuestas en cascada lineal.

Figura 2.1

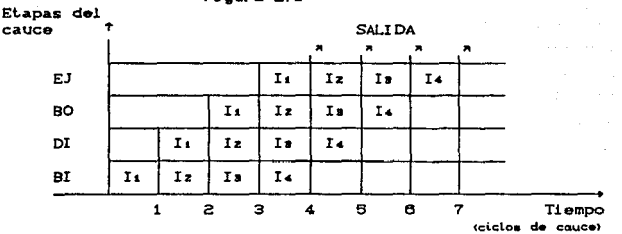
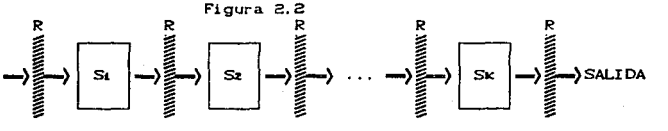


Diagrama espacio-tiempo para un procesador encauzado

Un procesador de encauzamiento lineal básico es el mostrado en la figura 2.2. El cauce consiste en una cascada de etapas de proceso. Las *etapas* son circuitos combinacionales puros que efectúan operaciones aritméticas o lógicas sobre el flujo de datos que circulan a través del cauce.

Figura 2.2



Estructura básica de un procesador de segmentación encauzada lineal

La etapas se separan mediante registros de acoplo rápidos. Estos registros retienen los resultados intermedios entre las etapas. Simultáneamente, los flujos de información entre etapas adyacentes están bajo el control de un reloj común aplicado a todos los registros de acoplo.

Al observar la figura 2.1 vemos que, una vez que el cauce está lleno, producirá un resultado por cada periodo de reloj. Idealmente, un cauce lineal con k etapas puede procesar k tareas en $Tk = k + (n-1)$ periodos de reloj, donde k ciclos se emplean para llenar el cauce o completar la ejecución de la primera tarea y los $(n-1)$ ciclos son necesarios para completar las $n-1$ tareas restantes. El mismo número de tareas se ejecutarían en un procesador no encauzado funcionalmente equivalente en un tiempo $T1 = n * k$.

II.2.1 *CLASIFICACIÓN DE LOS PROCESADORES ENCAUZADOS*

Encauzamiento aritmético.- Las unidades aritmético-lógicas de una computadora pueden segmentarse para realizar operaciones encauzadas en varios formatos de datos.

Encauzamiento de instrucciones.- La ejecución de un flujo de instrucciones puede adoptar una estructura de segmentación encauzada que permita solapar la ejecución de la instrucción actual con la búsqueda, decodificación, y busqueda de los operandos de las instrucciones siguientes.

Encauzamiento de procesadores.-Se refiere al procesamiento encauzado del mismo flujo de datos por parte de una cascada de procesadores, cada uno de los cuales procesa una tarea especifica.

El flujo de datos pasa por el primer procesador y los resultados se almacenan en un bloque de memoria que es también

accesible al segundo procesador. El segundo procesador pasa entonces los resultados refinados al tercero, y así sucesivamente.

Dependiendo de las configuraciones del cauce y de las estrategias de control, se han propuesto los siguientes esquemas de clasificación:

Cauces unifunción o multifunción Una unidad encauzada con una función fija y dedicada se denomina *unifunción*. Por otro lado, un cauce *multifunción* puede efectuar diferentes tareas, en diferentes momentos o a la vez, mediante la interconexión de diferentes subconjuntos de etapas en el cauce.

Cauces estáticos o dinámicos Un *cauce estático* puede asumir una sola configuración funcional cada vez. Los cauces estáticos pueden ser uni ó multifunción. El encauzamiento sólo es posible en los cauces estáticos si se ejecutan continuamente instrucciones del mismo tipo. La función efectuada por un cauce estático no debería modificarse frecuentemente. En caso contrario, su rendimiento puede ser muy bajo.

Un procesador de *encauzamiento dinámico* permite la existencia simultánea de varias configuraciones funcionales. En este sentido, un cauce dinámico debe ser multifunción. Por otro lado, un cauce unifunción debe ser estático. La configuración dinámica necesita mecanismos de control y secuenciamiento mucho más elaborados que los de los cauces estáticos.

Cauces escalares o vectoriales Un procesador de *Cauce escalar*: procesa una secuencia de operandos escalares bajo el control de un bucle DO. Las instrucciones de un pequeño bucle DO son con frecuencia preextraídas y almacenadas en la memoria temporal de instrucciones. Los operandos escalares necesarios para instrucciones escalares repetidas se trasfieren a una cache de datos para alimentar continuamente el cauce con operandos.

Cauces vectoriales: estos están diseñados especialmente para manejar instrucciones vectoriales sobre operandos vectoriales. Las computadoras que disponen de instrucciones vectoriales suelen denominarse *procesadores vectoriales*. El diseño de un cauce vectorial es una extensión del diseño de un cauce escalar.

II.3 *SISTEMAS MULTIPROCESADORES*

La investigación y desarrollo de estos sistemas están dirigidos a mejorar la productividad, fiabilidad, flexibilidad y disponibilidad de los sistemas.

Contienen dos o más procesadores de capacidades aproximadamente comparables. Todos los procesadores comparten acceso a grupos comunes de módulos de memoria, canales de Entrada/Salida y dispositivos periféricos, y lo más importante, el sistema entero debe estar controlado por un único sistema operativo integrado que facilite las interacciones entre los procesadores y sus programas a diferentes niveles. Además cada procesador dispone de su propia memoria local y de dispositivos privados.

Los sistemas multiprocesadores pueden ser caracterizados atendiendo a dos criterios: primero, un multiprocesador es una sola computadora que incluye múltiples procesadores, y segundo los procesadores se pueden comunicar y cooperar a diferentes niveles para resolver un problema dado. La comunicación se puede realizar enviando mensajes de un procesador a otro o compartiendo una memoria común.

II.4 *SISTEMAS MULTICOMPUTADORES*

Un sistema *multiprocesador* se caracteriza, como ya se mencionó, por una memoria compartida por todos los procesadores. En un sistema *multicomputador*, cada procesador cuenta con su

propia memoria local y la interacción de los procesadores se da a través de *paso de mensajes*.

En un sistema multiprocesador todos los datos usados por el programa son almacenados en la memoria compartida y ahí son accesibles a todos los procesadores. Sin embargo, en un sistema multicomputador los datos deben ser distribuídos a través de las memorias locales de los procesadores. Un procesador no tiene acceso directo a la memoria local de otro procesador, pero puede enviar o recibir bloques de datos de otros procesadores a través de la red de interconexión de los procesadores.

Para programar efectivamente un sistema multicomputador, es necesario saber un poco más sobre la arquitectura de hardware. Si se toma un programa eficiente diseñado para un sistema multiprocesador y se pone a correr en un sistema multicomputador generalmente será menos eficiente. Las características básicas del lenguaje y las técnicas de programación usadas en sistemas multiprocesadores se pueden adaptar a la programación de sistemas multicomputadores. Sin embargo, esto requiere del entendimiento de la organización general del hardware de los multicomputadores y su diferencia con los multiprocesadores.

En un sistema multicomputador, cada procesador cuenta con su propio módulo de memoria física privada para almacenar y recuperar datos durante el cálculo. Además, cada procesador tiene una o más conexiones con otros procesadores, a través de los cuales se pueden transmitir datos. Si un procesador no tiene conexión directa con otro procesador, éstos se pueden comunicar a través de procesadores intermedios que van pasando los datos. La transmisión de datos entre los procesadores requiere una cantidad significativa de tiempo y si los procesadores se encuentran distantes requieren aún más tiempo. Por esto, si hay una frecuente comunicación entre los procesadores durante la ejecución del

programa, el retardo de comunicación resultante puede incrementar significativamente el tiempo de ejecución del mismo.

El modelo completo de la conexión directa de los procesadores es usualmente llamado **topología** del multicomputador. Para un algoritmo particular, el retardo de ejecución resultante de la comunicación depende de la topología específica.

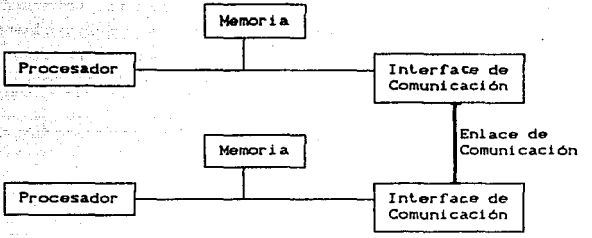
II.4.1 *COMUNICACIÓN DE LOS PROCESADORES*

En estos sistemas, cada procesador tiene un módulo de memoria local separada para almacenar sus datos y su código de programa. La actividad de cada par de *procesador-memoria* es muy parecida a una computadora secuencial ordinaria. El procesador ejecuta una serie de instrucciones determinada por el código de programa almacenado en la memoria.

En la forma en que los pares de *procesadores-memoria* trabajen juntos en un sólo problema computacional, deben ser capaces de comunicarse e intercambiar datos durante el cálculo. Esto se logra con una red de comunicación hardware que conecta a los procesadores y permite a los datos ser transmitidos desde cualquier procesador hacia cualquier otro. La construcción básica de esta red de comunicación es un *enlace de comunicación* directo procesador a procesador (Fig. 2.3). Cada par de *procesadores-memoria* es conectado a su propia *interface*[1] *de comunicación*, la cual es una pieza de hardware capaz de transmitir y recibir datos a través del *enlace de comunicación*.

[1] **INTERFACE:** Dar acceso, comunicar. Dispositivo que permite el entendimiento entre dos entidades diferentes.

Fig. 2.3



Construcción básica de comunicación entre procesadores

La función de una interface de comunicación con respecto al procesador es similar a una interface de entrada-salida (E/S) dentro de una computadora ordinaria. La interface (E/S) recibe comandos de entrada o salida del procesador y entonces las interfaces con un dispositivo específico, como son la impresora o el disco, llevan a cabo los comandos y envían o reciben datos. La única diferencia es que los datos son enviados a través de un enlace de comunicación, en lugar de un dispositivo (E/S).

El *enlace de comunicación* es bidireccional, lo cual significa que los datos pueden ir en ambas direcciones. Cada interface de comunicación es capaz de ambas cosas enviar o recibir datos. Este enlace es simplemente una conexión eléctrica capaz de transportar secuencias de bits de una interface de comunicación a otra.

II.4.2 *TOPOLOGÍA DE LOS SISTEMAS MULTICOMPUTADORES*

El número de enlaces con que cuenta el sistema dependerá de la estructura de la red de comunicación, algunas veces llamada la *topología* del sistema multicomputador. Dos paramentros importantes

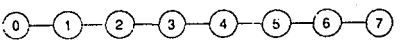
que caracterizan a cada topología son la **conectividad** que es el número de conexiones directas por procesador, factor determinante del costo de la red; y el **diámetro**, el cual es el máximo número de conexiones intermedias requeridas para comunicar procesadores distantes, factor importante en el desarrollo funcional de la red.

La topología más sencilla tienen una menor conectividad y por lo tanto un mayor diametro. Las topologías más complejas tienen una alta conectividad y por lo tanto un menor diámetro.

Topología Lineal y de Anillo

En una topología multicomputadora **Lineal**, las interfaces de comunicación estan conectadas en una línea recta, como se muestra en la Fig. 2.4. Se supone que cada circulo enumerado en la figura contiene un procesador, una memoria y una interface de comunicación. Cada línea entre un par de círculos representa un enlace de comunicación directo en la red.

Fig. 2.4



Topología Lineal

Cuando dos procesadores están conectados directamente en una topología dada, se dice que son procesadores *adyacentes*. La *distancia* entre cualquier par de procesadores está definida como el número de enlaces de comunicación que un mensaje debe atravesar de la manera más directa entre los dos procesadores.

El *diámetro* de una topología está definida como la distancia mas larga entre cualquier par de procesadores en la red. Por

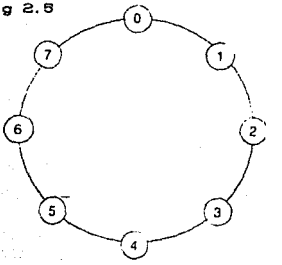
ejemplo, en la topología de *línea* de la figura 2.4 con ocho procesadores, el diámetro es 7. En general, una topología de línea con n procesadores tiene conectividad de 2 y diámetro $n-1$. La distancia entre cualquier par de procesadores i y j, siempre es $|i-j|$.

Sin incrementar el costo, la representación de una topología Lineal puede ser grandemente mejorada añadiendo simplemente un enlace de conexión más entre el primer y último procesador de la línea. A este se le llama topología **Anillo** (Fig. 2.5). Esta conexión adicional reduce el promedio de distancia entre los procesadores por un factor de dos. Una topología de Anillo de n procesadores tiene un diámentro de $n/2$.

Topología de Malla y Torus

Al incrementar el número de enlaces de comunicación conectados a cada procesador, es posible reducir el diámetro de la red y el retardo promedio de comunicación. Un tipo de topología es la **Malla** (Fig. 2.6). Esta topología de multicomputadoras consta de

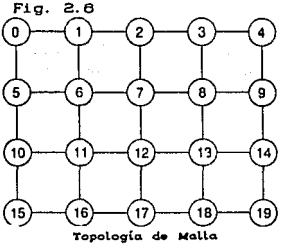
Fig 2.5



Topología de Anillo

procesadores dispuestos en un arreglo bidimensional. Un procesador en el renglón i y columna j, está conectado a los cuatro procesadores vecinos inmediatos a la izquierda, derecha, arriba y abajo: en las localidades $(i-1,j)$, $(i+1,j)$, $(i,j-1)$, $(i,j+1)$. Todas las conexiones son horizontales entre columnas adyacentes o verticales entre renglones adyacentes, no hay conexiones diagonales. Los procesadores que se encuentran en las orillas tienen dos o tres vecinos.

Fig. 2.6



Topología de Malla

Los mensajes que necesiten enviarse a procesadores distantes deben viajar a través de rutas horizontales o verticales entre los procesadores intermedios. Cada par de procesadores tendrá una ruta de longitud mínima entre ellos, medida por la suma de la distancia entre renglones y la distancia entre columnas.

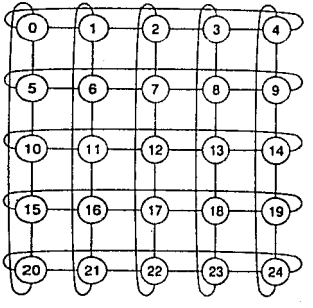
Si la demora en la comunicación básica entre procesadores adyacentes es T, entonces el tiempo de comunicación interprocesador para cualquier par de procesadores es la longitud de ruta multiplicada por T.

Para una Malla de m por m, el diámetro de la red es simplemente la longitud de ruta entre los procesadores de las esquinas contrarias. Esto es siempre $2(m-1)$.

La eficiencia de una Malla puede mejorarse adicionando conexiones a cada renglón y columna. Esto cambia a cada renglón y columna de una Lineal a un Anillo como se ilustra en la Fig. 2.7. Cada procesador en el extremo izquierdo está conectado a su contraparte del extremo derecho. De manera similar cada procesador del límite superior está directamente conectado a su correspondiente del límite inferior. Esta estructura es conocida como topología Torus.

La conectividad de cualquier topología Torus es siempre 4, ya que ahora cada procesador tiene exactamente 4 enlaces de conexión. Los procesadores en esquinas contrarias están separados unicamente por una distancia de dos. La distancia máxima en el Torus esta entre los procesadores de cualquier esquina y el procesador central. Por lo tanto, una topología de Torus de m por m tendrá diámentro m.

Fig 2.7



Topología Torus

Topología de Hipercubo

En una topología de interconexión en Hipercubo, el número de procesadores es siempre una potencia exacta de dos. Si el número de procesadores es 2, entonces d es llamado la dimensión del Hipercubo.

Cada procesador tendrá un número cuya representación binaria tendrá d bits. Cualquier procesador dado con número binario i tendrá conexiones directas con todos los procesadores con número binario j, tales que j difiera únicamente en un dígito binario de i. La distancia entre procesadores en un Hipercubo es igual al número de bits en los cuales sus números binarios difieran.

En general para un Hipercubo de dimensión d, los procesadores tendrán d vecinos inmediatos y una distancia máxima de d pasos entre procesadores. Por lo tanto, la conectividad y diámentro de la red será d.

Una topología en Hipercubo puede estar definida con una simple regla de construcción recursiva. Un Hipercubo de dimensión 1 esta definida para tener dos procesadores, numerados 0 y 1 con una sóla conexión directa entre ellos. Un Hipercubo de dimensión $d+1$ esta definido recursivamente por el siguiente procedimiento de construcción:

1. Crear un duplicado exacto del Hipercubo de dimensión d, incluyendo los números de procesador.

2. Crear una conexión directa entre los procesadores con el mismo número en el original y en el duplicado.

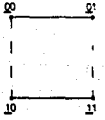
3 Adicionar un 1 a la izquierda de cada número de procesador en el duplicado y un 0 a la izquierda en cada número de procesador en el original.

Esta construcción recursiva esta ilustrada para las dimensiones 1-4 en la Figura 2.8:

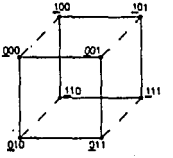
Fig. 2.8

Dimension 1:

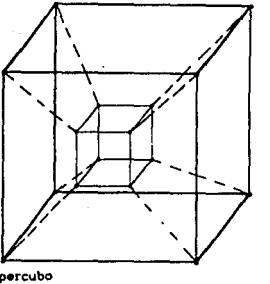
Dimension 2:



Dimension 3:



Dimension 4:



Topología Hipercubo

III. Las técnicas más importantes de software para procesamiento paralelo

Al programar en paralelo es muy importante determinar el tipo de sistema en el que se va a trabajar, es decir si será un sistema *Multiprocesador* o un sistema *Multicomputador*, ya que la comunicación entre los procesos es vital para lograr la máxima eficiencia y la forma de comunicación es diferente para cada uno de estos sistemas.

Existen en el mercado algunos sistemas operativos ó lenguajes que facilitan esta comunicación, por ejemplo: *OCCAM*, que fué creado para trabajar máquinas paralelas transputer con sistema operativo UNIX; el sistema operativo *LINDA* que es un modelo abstracto creado en la Universidad de Yale; el sistema operativo *PARALATION*, desarrollado en lenguaje LISP, y el lenguaje *MULTI-PASCAL*; entre otros [8].

Todos ellos centran su atención en la forma de encapsular instrucciones para enviarlas a los demás procesadores y en las instrucciones que permiten recibirlas, como podrían ser: *wait* (espera), *send* (envía), *receive* (recibe) y *collect* (colecciona).

El *Multi-Pascal*, desarrollado en la Universidad Internaconal Maharishi en Fairfield, Iowa; cuenta con un conjunto de abstracciones de programación paralela de alto nivel, con el suficiente poder para representar algoritmos paralelos tanto para multiprocesadores como multicomputadores.

Como lo indica el nombre *Multi-Pascal* es una extensión del popular lenguaje de programación *Pascal* más algunas características especiales para la creación e interacción de procesos paralelo simulados.

III.1 *CARACTERÍSTICAS DE LA PROGRAMACIÓN*

Para programar en lenguajes paralelos, en especial para el MULTI-PASCAL[1], se requiere de ciertas instrucciones especiales para comunicar los resultados obtenidos de los diferentes procesos para conformar el resultado final.

Dentro de los programas paralelos la construcción más importante es el *proceso*. Informalmente, un proceso puede ser visto como un fragmento del *código del programa* asignado para su ejecución a un procesador dado. Los programas secuenciales ordinarios pueden ser entendidos como un caso especial de programas paralelos, en los cuales hay un único proceso y un único procesador.

En el caso particular del Multi-Pascal, el programa principal es el primer *proceso* y es asignado para su ejecución al primer procesador. El programa principal puede contener cualquiera de los tipos ordinarios de instrucciones que son encontrados en los programas secuenciales como asignaciones, ciclos, condiciones e instrucciones de entrada/salida.

Sin embargo, existe también la posibilidad de usar un tipo completamente nuevo de instrucciones que no se encuentra en los programas secuenciales: una instrucción de "*creación de procesos*". En Multi-Pascal hay instrucciones cuya ejecución causarán la creación de procesos completamente nuevos y asignados a otros procesadores para su ejecución. Es así como se inicia la actividad paralela dentro del programa: un proceso existente que ya está corriendo en un procesador ejecuta una instrucción de "*creación de procesos*".

[1] Lester, P. Bruce. "The art of Parallel Programming". Ed. Prentice Hall, E. U. A. 1993.

III.2 *INSTRUCCIONES ESPECIALES*

III.2.1 *FORALL*

El mejor método para crear procesos paralelos en Multi-Pascal es la instrucción *FORALL*: la forma paralela de un ciclo iterativo FOR, donde las iteraciones se realizan en paralelo, en vez de secuenciales. Usando esta instrucción se pueden crear cientos de procesos paralelos, lo cual resulta en un gran incremento de la velocidad de ejecución del programa.

Por ejemplo, considere el siguiente fragmento de programa secuencial que obtiene la raiz cuadrada de cada elemento de un arreglo:

```
PROGRAM   Raiz_Cuadrada;
VAR     A: ARRAY [1..100] OF REAL;
        i: INTEGER;
BEGIN
  . . .
FOR     i := 1 TO 100 DO
  A[i] := SQRT ( A[i] );
  . . .
END.
```

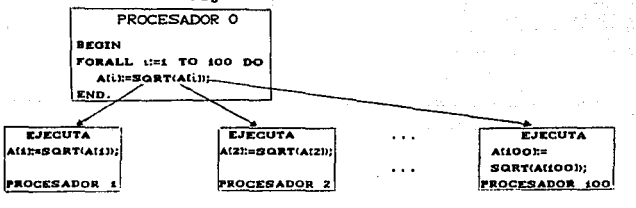
Este programa puede ser modificado de secuencial a paralelo para operar sobre los 100 elementos del arreglo con 100 procesadores paralelos:

```
PROGRAM    Raiz_Cuadrada_Paralela;
VAR     A: ARRAY [1..100] OF REAL;
        i: INTEGER;
BEGIN
  . . .
FORALL i := 1 TO 100 DO
  A[i] := SQRT ( A[i] );
  . . .
END.
```

La instrucción FORALL crea 100 copias de la instrucción de asignación siguiente y hace cada una un proceso paralelo separado

con su propio y único valor de la variable i. Cada uno de estos 100 procesos puede ser ejecutado en un procesador diferente, todos en paralelo. El programa principal que corre en el procesador 0 es el proceso *"padre"* (fig. 3.1), y este crea 100 procesos *"hijos"* para los 100 procesadores físicos del multiprocesador.

Figura 3.1



Creación de procesos paralelos

Para crear un nuevo proceso se requiere de alguna actividad computacional y por lo tanto se consume cierto tiempo de máquina. La creación de un proceso nuevo usualmente involucra añadir algunas nuevas entradas a las tablas del sistema operativo, posiblemente cambien algunos apuntadores o algo más. Típicamente, la creación de un proceso requiere la ejecución de 30 a 50 instrucciones de máquina, o posiblemente más en algunos sistemas computacionales.

Si el tiempo de creación de procesos necesita 10 unidades de tiempo, entonces la instrucción FORALL, del ejemplo de la Raiz_Cuadrada_Paralela, requiere 100 * 10 = 1000 unidades de tiempo en el Procesador 0 para crear todos los proceso hijos. Supóngase que el cuerpo de instrucciones de cada proceso hijo requiere sólo 10 unidades de tiempo para su ejecución. Como todos los procesos

hijos son ejecutados en paralelo en diferentes procesadores físicos, ellos necesitan sólo 10 unidades más sobre el tiempo general. Así, el tiempo total de ejecución del FORALL es 1010.

Sin embargo, para el ejemplo secuencial se requieren 10 unidades por cada iteración del ciclo FOR, así 10 * 100 = 1000 unidades de tiempo son necesarias para completarlo ¡ 10 menos que el paralelo !.

Por otro lado, supóngase que el tiempo de ejecución del cuerpo del ciclo es ahora de 10,000 unidades. Calculando para el mismo ejemplo el tiempo de creación de los hijos sería el mismo: 1000 unidades + 10,000 de la iteración= 11,000 unidades de tiempo se necesitan para completar el ciclo. Mientras que para un ciclo secuencial se necesitan 100 * 10,000 = 1'000,000 unidades de tiempo para completarlo.

Para ayudar a superar este problema del tiempo de ejecución de los procesos con la instrucción FORALL, *Multi-Pascal* cuenta con la posibilidad de agrupar los valores de los índices en un mismo procesador.

Por ejemplo, para el programa de Raiz_Cuadrada_Paralela, es posible crear solo 10 procesadores, cada uno con la posibilidad de evaluar 10 valores del índice *i*. En la instrucción FORALL, la palabra *GROUPING* es usada para agrupar un cierto número de índices en cada procesador como sigue:

```
PROGRAM          Raiz_Cuadrada_Paralela;
VAR    A: ARRAY [1..100] OF REAL;
       i: INTEGER;
BEGIN
  . . .
FORALL i := 1 TO 100 GROUPING 10 DO
  A[i] := SQRT ( A[i] );
  . . .
END.
```

La notación *GROUPING* 10 ocasiona que la variable índice se forme en grupos de 10 en cada proceso. Así, solo se crean 10 procesadores, el primero evaluará secuencialmente la variable índice de 1 a 10, el segundo de 11 a 20, el tercero de 21 a 30 y así sucesivamente.

La sintaxis general del FORALL es:

```
FORALL <variable_indice> := <inicial> TO <final>
        ( GROUPING <tamaño> ) DO
    <Instrución>
```

donde <inicial>, <final> y <tamaño> pueden ser expresiones de valor entero. *GROUPING* es opcional, si este se omite el tamaño del grupo por default es 1.

III.2.2 *FORK*

Al programar existen algunas circunstancias donde es útil la capacidad para convertir una instrucción individual en un proceso hijo. Esto se hace con otra instrucción primitiva de creación de procesos: el operador *FORK*. Al anteponer el operador FORK a cualquier instrucción, esta se convierte en un proceso hijo que corre en paralelo con su padre. La sintaxis general es:

```
FORK <instrucción>;
```

dónde <instrucción> puede ser cualquier instrucción válida en *Multi-Pascal*. Esta <instrucción> será un proceso hijo que será ejecutado en un procesador diferente. El padre continuará con la ejecución inmediatamente sin esperar por su hijo en ningún aspecto.

III.2.3 *TERMINACION DE LOS PROCESOS*

Para comprender la diferencia entre la instrucción *FORALL* y el operador *FORK*, es necesario considerar la terminación de los procesos con más cuidado. Un proceso termina cuando llega al final de su código. Sin embargo, pequeñas variaciónes en la velocidad de los procesadores o alguna otra influencia externa puede ocasionar que el tiempo de terminación varie.

En cualquier caso, el procesador padre que contiene una instrucción *FORALL*, siempre esperará a que terminen todos los procesos hijos para continuar con la ejecución del programa. Por ejemplo:

```
FORALL i := 1 TO n DO
    < instrucción >;
```

este fragmento de programa seguiria los siguientes pasos:

Crear los n procesos hijos para <instruccion>
Esperar hasta que los n procesos hijos hayan terminado
Continuar con la instrucción siguiente al *FORALL*

La situación es algo diferente para el operador *FORK*. En éste caso solo se crea un proceso hijo y el padre continúa con la ejecución sin esperar a que el proceso hijo termine, ejemplo:

```
FOR i := 1 TO n DO
    FORK <instrucción>;
```

cada iteración de este ciclo crea un nuevo proceso hijo para <instrucción>. Tan pronto como cada nuevo hijo es creado la siguiente instrucción del ciclo es ejecutada. Después de que los n hijos son creados el ciclo termina y la ejecución del proceso principal continúa inmediatamente con la siguiente instrucción del FOR. Así, el proceso padre se ejecuta en paralelo junto con sus n hijos.

Como no está permitido que el proceso padre termine hasta que todos sus hijos hayan teminado, si el padre llega al fin del código de su programa mientras uno o más de sus hijos aún estan corriendo, entonces la ejecución del proceso padre será suspendida hasta que todos los hijos hayan terminado y sólo entonces el proceso padre termina.

III.2.4 *JOIN*

En algunos programas es conveniente que el padre espere en algún punto a la terminación de uno o todos sus procesos hijos. Para esto, *Multi-Pascal* cuenta con la instrucción *JOIN*.

Si el padre tiene un sólo hijo y se ejecuta la instrucción *JOIN*, este se ve forzado a esperar que el hijo termine su ejecución.

Si por el contrario, el padre tiene varios procesos hijos, entonces la terminación de cualquiera de ellos satisface cualquier *JOIN* que se encuentre dentro del proceso padre. Cuando se ejecuta un *JOIN*, el padre no especifica la espera de ningún hijo en particular, por ello es conveniente, si se tienen múltiples instrucciones *FORK*, tener el mismo número de instrucciones *JOIN* para esperar a la terminación de todos ellos. Ejemplo:

```
FOR i := 1 TO n DO
    FORK (instrucción);
FOR i := 1 TO n DO
    JOIN;
```

Si el padre ejecuta erróneamente, más instrucciones *JOIN* que hijos creados, entonces el proceso padre se suspende indefinidamente ocasionando un "deadlock" en el programa (la detención del programa).

III.3 *TIPOS DE DATOS ESPECIALES*

III.3.1 *CHANNEL*

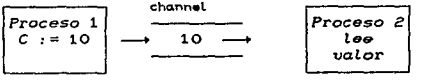
Cuando se necesita que los resultados de un proceso sean utilizados en otros, éstos resultados se transfieren a través de una variable de tipo *channel*, la cual se puede considerar como una cola en la que se van almacenando los datos.

Los valores se asignan a una variable *channel* de la misma forma que se asignan a cualquier otra variable. Suponiendo que *C* es una variable *channel*, ejemplo:

```
Proceso 1
C := 10;
```

esta asignación colocara el numero 10 al principio de la cola, si se encuentra vacia, para que sea leida por algún otro proceso (fig. 3.2).

Figura 3.2



La sintaxis general para declarar una variable *channel* es:

```
VAR
<Nombe_variable> : CHANNEL OF <tipo_componente>;
```

donde <tipo_componente> puede ser cualquier tipo ordinario de Pascal como INTEGER, REAL, CHAR ó BOOLEAN.

Para leer el valor de una variable *channel*, simplemente se asigna el valor de la variable *channel* a otra variable, ejemplo:

```
X := C;
```

esta asignación lee el valor del frente de la "cola" de valores almacenados en *C* y lo escribe en la variable *X* que debe ser del mismo tipo_componente de *C*:

```
C : CHANNEL OF REAL;
X : REAL;
```

Cuando un proceso trata de leer el contenido de una variable *channel* y ésta se encuentra vacia, el procedimiento se detiene hasta que algún otro proceso escriba en esa variable. Este tipo de variables no tienen limitante en la capacidad de almacenamiento.

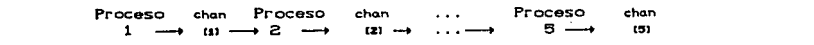
Para saber si una variable *channel* contiene valores se puede crear un valor booleano al utilizar el nombre de la variable y ?, ejemplo:

```
IF  C?  THEN  X := C (lee el canal)
        ELSE X := 0; (no lee el canal)
```

esta expresión evalúa TRUE (verdadero) si *C* contiene valores o FALSE (falso) si el canal está vacio.

También se pueden utilizar arreglos, apuntadores o registros, aplicando las mismas reglas, ejemplos:

```
VAR  chan : ARRAY [1..5] OF CHANNEL OF INTEGER;
```



En este caso: el proceso 2 lee datos del chan[1] y escribe datos en chan[2].

REGISTRO

```
D: CHANNEL OF RECORD
     izq, der : INTEGER;
     centro: REAL;
     END;
```

APUNTADOR

```
TYPE    item: RECORD
          x,y: REAL;
          END;
VAR  e: CHANNEL OF ^item;
```

NOTA: Para manipular los elementos de estos tres últimos tipos de datos es necesario asignarlos antes a variables sencillas del mismo tipo, ya que no es válido leer o escribir uno solo de los elementos de la variable *channel*, ejemplo:

```
TYPE   artyp = ARRAY [1..10] OF REAL;
VAR    c: CHANNEL OF artype;
       a: artyp;

c[3] := 20;    <------ NO es válido
a := c;
a[3] := 20;    <------ SI es válido.
```

III.3.2 *DATOS COMPARTIDOS*

Durante la corrida de un programa, existen instrucciones que alteran el contenido de los datos de la memoria, en el caso de trabajar con varios procesos, si un proceso accesa un dato para modificarlo, y antes de que esto suceda se permite accesar a otro proceso el mismo dato entonces puede producirse un error. Para evitar este tipo de interferencias entre los procesos paralelos podemos definir *operaciones atómicas*.

Una *operación atómica* es una operación que no puede ser interrumpida por otro proceso. Para crear una *operación atómica*, se requiere en el lenguaje de programación algún mecanismo que haga esperar a los procesos.

En *Multi-Pascal* esto puede ser realizado con *spinlock*, que crea una "puerta" a través de la cual sólo puede pasar un proceso a la vez. Cuando un proceso pasa por la "puerta" esta se cierra automáticamente para prevenir que entre otro proceso. Después de que el proceso termina la *operación atómica*, la puerta se abre otra vez para permitr la entrada a otro proceso.

El *spinlock* es implementado como una variable asociada a un área de memoria. Si la memoria tiene 0 representa el estado "UNLOCKED" (abierto) y si vale 1 representa el estado "LOCKED" (cerrado).

En *Multi-Pascal spinlock* es un tipo definido de dato. Una vez que una variable es definida de este tipo entonces se pueden utilizar los procedimientos "LOCK" o "UNLOCK". Ejemplo:

```
PROGRAM Busqueda;
VAR A: ARRAY [1..200] OF INTEGER;
    i,valor,n: INTEGER;
    L: SPINLOCK;
BEGIN
      ...
  n := 0;
  READLN(valor);
  FORALL i := 1 TO 200 GROUPING 10 DO
     IF A[i] = valor THEN
     BEGIN
        LOCK(L);
           n := n + 1;
        UNLOCK(L);
     END;
  WRITELN('Total concurrencias', n);
END.
```

El *spinlock* puede también combinarse con las estructuras de datos Arreglo y Registro:

```
VAR   G: ARRAY[1..N] OF SPINLOCK;
      item: RECORD
               dato: INTEGER;
                  I: SPINLOCK;
            END;
```

```
LOCK(O(3));
    (instruccion);
UNLOCK(L);

LOCK(item^.I);
    (instruccion);
UNLOCK(item^.I);
```

III.3.3 PROGRAMAS DE PASO DE MENSAJES

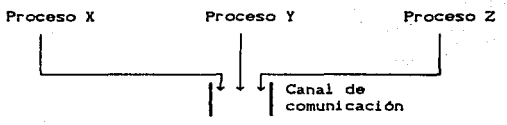
Cuando se programa en un sistema multiprocesador, el programador ve conceptualmente sus programas como una colección de procesos accesando a un banco central de variables compartidas. En un sistema multicomputador, el programador debe ver su programa como una colección de procesos, cada uno con sus propias variables locales, privadas, más la habilidad de mandar y recibir datos de otros procesos a través de *message-passing (paso de mensajes)*.

En estos sistemas no existen las variables compartidas y ningún procesador tiene acceso directo a ninguna memoria local excepto la suya. Si un procesador necesita algún dato almacenado en la memoria local de un procesador remoto. Ese procesador remoto debe leer los datos de la memoria local y enviarlos como mensaje a través de la red.

En una implementación de sistema multicomputador en *Multi-Pascal*, la comunicación entre procesos se hace a través de "puertos" *(ports)* asociados a cada proceso.

El uso y propiedades de estos puertos de comunicación son casi idénticos a las variables *channel*, la principal diferencia es la restricción de que un proceso puede recibir mensajes sólo de un puerto dado (Fig. 3.3).

Figura 3.3



Puertos de comunicación

Un puerto de comunicación en *Multi-Pascal* es simplemente una variable *channel* que ha sido asignada a un proceso especifico, con el propósito de recibir mensajes. Hay dos partes en la declaración de cada puerto de comunicación.:

1. La declaración como una variable *channel* al principio del programa principal.
2. Una declaración *PORT* que aparece con la instrucción de creación de procesos.

```
PROGRAM Ejemplo;
VAR COm:ARRAY [1..80] OF CHANNEL OF INTEGER;
        . . .                          (canales de comunicacion)

PROCEDURE Proceso(i: INTEGER);
VAR inval, outval, k : INTEGER;
        . . .
BEGIN
    . . . .
  inval:=Com[i];(leo de mi propio canal de comunicación)
        . . .
  Com[k]:=outval; (envio 'outval' al proceso k)
        . . .
END;

BEGIN
        . . .
  FORALL i := 1 TO 80 DO
     (PORT Com[i]) Proceso(i);
        . . .
END.
```

Cada canal del arreglo puede ser leído por un sólo proceso y ese proceso es el asignado como puerto por la declaración PORT, así cada canal puede tener múltiples escritores, pero un sólo lector. La sintaxis general de la declaración PORT es:

(PORT <lista_canales>) <instrucción>;

La <instrucción> forma el cuerpo del proceso que será creado, y <lista_canales> es una lista separada por comas de:

1. Variables *channel*
2. Un arreglo de *channels*.

Cualquier canal declarado al principio del programa principal, que no se haya incluido en ninguna declaración PORT, es asignado por default a ser el puerto de comunicación para el programa principal.

III.3.4 *PRIMITIVAS AUXILIARES*

LA @

Con la @ y un número de procesador deseado , un proceso puede ser asignado para correr en cualquier procesador, la sintaxis general es:

(@ <expresion>) <instrucción>;

donde <instrucción> es el cuerpo del proceso que será creado, y <expresión> es cualquier expresión valida en *Multi-Pascal* que resulte en un valor entero, ejemplo:

```
FORK (@ i * 10) Proceso(i);

FORALL i := 1 TO 50 DO
    (@ i-1) Proceso (i);
```

EL %SELF

En algunos programas es útil para el proceso conocer su propio número de procesador como punto de referencia para poner los procesos creados recientemente. La sintaxis es:

%SELF **función predefinida.**

Ejemplo:

```
FORK (@ %SELF-1) Proceso(i);
```

IV. LÁ MÁQUINA DE TURING Y SUS PRIMOS Y PRIMAS

Durante la decada de los años treinta se desarrollaron varias teorías, en diferentes lugares del mundo (Inglaterra, Rusia y Estados Unidos), las cuales trataban, en el fondo, un mismo principio: *"Todo procedimiento que sea computable puede ser resuelto"*,[12] dicho en otras palabras, todo prrocedimiento que pueda ser descrito con un número de pasos precisos puede ser resuelto. Este principio es concíderado la base de la computación moderna.

Por otro lado, la investigación en computadoras paralelas requiere de harware caro, Sin embargo, es posible crear un simulador de computación paralela, el cual se puede utilizar para experimentar con los algoritmos y principios básicos.

Para poder desarrollar este simulador es necesario conocer primero las teorías antes mencionadas, para poder entender por qué se utilizó la teoría de Turing. por ello, a continuación se menciona el contenido de las teorías de Turing, Post y Church.

IV.1 *LA MÁQUINA DE TURING*

Turing, demostró que una máquina calculadora que tenga un mínimo de información adecuada, puede imitar a otra sin que importe lo grande que sea el repertorio de instrucciones de ésta última. Para probar que ésto es posible él diseñó el bosquejo de una computadora ideal que no estuviese limitada en su utilización por una cantidad máxima de almacenaje de información, como las computadoras en realidad lo están.[10]

IV.1.1 *COMPONENTES*

Conceptualmente la máquina de Turing es una cabeza lectora y escritora que puede correr a lo largo de un cinta infinita de símbolos de un alfabeto finito, la cabeza puede leer o escribir estos símbolos según las instrucciones de un programa definido por estados, y la máquina puede pasar de un estado a otro, para mayor comprensión tomemos la siguiente

Definición

Una máquina de Turing consta de tres partes:

1. Unidad de memoria.
2. Unidad de control.
3. Programa.

Unidad de memoria

La memoria de una máquina de Turing consiste de una cinta potencialmente infinita dividida en cuadrados, cada cuadrado capaz de contener un símbolo de un conjunto de símbolos predefinidos llamado *el alfabeto* de la máquina. Estos son los símbolos con los cuales representamos los datos, por eso, es importante especificar primero el alfabeto sobre el cual se ejecutará el procedimiento.

Unidad de control

Esta unidad determina qué instrucción será ejecutada después y realiza la operación. La unidad de control se comunica con la cinta por medio de la *Cabeza Lectora/Escritora* (CLE) y con el programa por medio de la *Cabeza solo de Lectura* (CSL). La CLE es capaz de leer o escribir el contenido de un sólo cuadro mientras la CSL lee sólo una instrucción del programa a la vez (Fig 4.1). Una máquina de Turing para (es decir, detiene la ejecución del programa) si y sólo si se encuentra una instruccion *HALT*.

IV.1.2 *PROGRAMA*

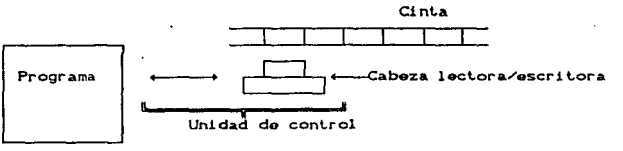
Es una secuencia finita de instrucciones primitivas que la máquina de Turing es capaz de ejecutar.

Las instrucciones primitivas y su interpretación son:

Left(L).....Muevo la CLE un cuadro a la izquierda
Right(R).....Mueve la CLE un cuadro a la derecha.
Write α.....Remplaza el símbolo del cuadro que se encuentra debajo de la CLE por el símbolo α.
Goto n.....Mueve la CSL a la instrucción con etiqueta *n* (o bien, cambia al estado n).
Halt.....Termina el cálculo.

Este pequeño conjunto de instrucciones, representa actualmente, el conjunto mínimo de instrucciones requeridas para ejecutar cualquier función calculable, esto es, uno puede encontrar el conjunto de instrucciones adecuadas que incremente el poder de la máquina de Turing, por que debido a su inherente simplicidad es posible definir la semántica de cada instrucción sin ambigüedades.

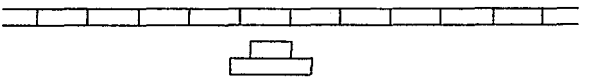
Figura 4.1



Componentes de la máquina de Turing

IV.1.3 *FUNCIONAMIENTO*

La actividad de una máquina de Turing, está definida por pasos instantáneos y discretos y cada paso es determinado por dos condiciones iniciales: El estado actual de la máquina y el símbolo que esta leyendo en la cinta. Dado algún par de condiciones iniciales, la máquina recibe una instrucción de dos partes para su siguiente operación. La primera parte de la instrucción designa el símbolo que la máquina dejará en la cinta o indica si la CLE se moverá a la izquierda o derecha. La segunda parte de la instrucción especifica el siguiente estado de la máquina que puede ser el mismo o cualquier otro existente dentro del programa. Por ejemplo, Supongamos que la cinta es la siguiente,



y la máquina se encuentra en el estado S_1. Dadas estas condiciones el programa podría indicar (CL, S_2), lo cual significa que la CLE se moverá un cuadro a la izquierda y la CSL pasará al estado 2.

IV.1.4 *EJEMPLO*

La forma más sencilla de escribir (representar) un programa para la máquina de Turing es por medio de una *Tabla de Estados* (Fig. 4.2) con la que podremos determinar que instrucción va a realizar la máquina en el siguiente paso, de acuerdo al estado en que se encuentra la CSL y el símbolo que está leyendo.

Figura 4.2

Estado	Símbolo leído	
	1	B
S_1	1, S_2	1, S_2
S_2	R, S_3	HALT
S_3	R, S_3	B, S_2

Tabla de estados para la máquina de Turing

Supongamos que la cinta no contiene ningún símbolo diferente de blanco (B), es decir, está vacía (el alfabeto utilizado es Σ={ 1, B }) y la CLE se encuentra frente a un cuadro como se muestra en la figura 4.3, y la CSL se encuentra en el estado S_1. En el primer paso se ejecutará la instrucción (1,S_2) y ésto significa: escribir un 1 y pasar al estado S_2 (Fig. 4.4); en el segundo paso se realizará la instrucción (R,S_3), lo cual mueve la CLE un cuadro a la derecha y el control pasa al estado S_3 (Fig. 4.5). En el tercer paso tendremos como instrucción a realizar (B,S_2), por lo tanto, regresa el control al estado S_2 y no se realiza ningún movimiento; al llegar a este cuarto paso la instrucción es un *HALT*, con ésto la máquina se para y el proceso termina.

Como podemos observar, el único propósito de éste programa ha sido el imprimir una marca en la cinta e identificar como se transfiere el control entre los estados del mismo.

Figura 4.3

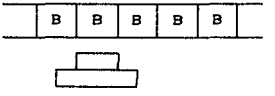
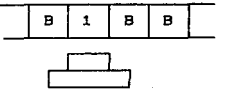
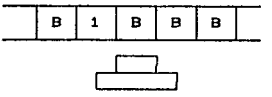


Figura 4.4



Movimientos de la máquina de Turing

Figura 4.5



Movimientos de la máquina de Turing

IV.2 *LA MÁQUINA DE POST*

IV.2.1 *COMPONENTES*

La máquina de Post representa de por sí una estructura que sólo existe en nuestra imaginación y es por eso que se dice que es una máquina "abstracta".

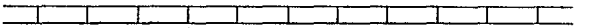
La máquina de Post consta de la *cinta* y del *carro* (que se llama también *cabeza de lectura y de registro*). La cinta es infinita y se divide en *células* de igual dimensión, considere que la cinta está colocada horizontalmente (Fig. 4.5).

Aunque pretender que la cinta sea infinita es casi imposible, en la construcción de ésta máquina se podría considerar, de igual modo que en la máquina de Turing, que la cinta *crece indefinidamente* por ambos lados: por ejemplo, podríamos considerar que la cinta crece en una sóla célula en cuanto el carro llega hasta el fin de la cinta y debe seguir moviéndose, o bien podríamos considerar que por cada unidad de tiempo crece una célula a la izquierda y a la derecha.

Sin embargo, por ser más fácil considerar que todas las células estan ya creadas, se dirá que la cinta es infinita por ambos lados.

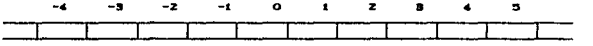
El orden en que están dispuestas las células es semejante al orden en que se encuentran todos los números enteros. Por consiguiente, es natural introducir en la cinta el "Sistema de coordenadas de números enteros", numerando las células mediante los números enteros ...,-3,-2,-1,0,1,2,3,... (Fig. 4.7).

Figura 4.6



La cinta de la máquina se divide en células y se extiende infinitamente a la izquierda y a la derecha.

Figura 4.7



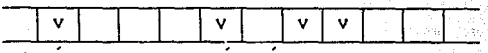
Cinta de la máquina de Post

Considere que el sistema de coordenadas está vinculado rígidamente con la cinta y obtendremos de este modo la posibilidad de indicar una célula cualquiera de la cinta, nombrando su número de orden o coordenada.

En cada célula de la cinta puede no estar escrito nada (tal célula se denomina *vacía*), o bien escrita la *marca V* (en este caso, la célula se llama *marcada*) (Fig. 4.8).

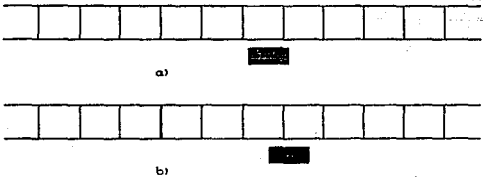
La información de qué células están vacías y cuáles marcadas constituye el *estado de la cinta*. Con otras palabras, el estado de la cinta es la distribución de las marcas por sus células. En el lenguaje matemático exacto el estado de la cinta es una función que a cada uno de los números (número de la célula) le corresponde la marca o la palabra "vacío". Como se verá a continuación el estado de la cinta varía en el proceso de funcionamiento de la máquina.

Figura 4.8



Cada célula de la cinta está vacía o contiene una marca.

Figura 4.9



Cuando el carro está inmóvil, éste se encuentra frente a una de las células de la cinta como se muestra en la fig. a). La situación que se expone en la fig. b) puede surgir sólo durante el proceso de movimiento del carro.

El carro puede desplazarse a lo largo de la cinta a la izquierda y a la derecha. Cuando el carro está inmóvil, éste se encuentra exactamente frente a una sola célula de la cinta (Fig. 4.9a; en ésta y en las siguientes figuras el carro viene representado en forma de un cuadrado ennegrecido); se dice que el carro *observa* ésta célula o bien la mantiene en el *campo visual*.

La información sobre qué el estado de la cinta y la posición del carro constituye el *estado de la máquina de Post*. En cada unidad de tiempo (la que llamaremos *paso*) el carro puede desplazarse a una célula a la derecha o bien a la izquierda. Además, el carro puede imprimir o eliminar la marca en aquella célula frente a la cual se encuentra, así como identificar si hay o no marca en la célula observada por el mismo.

IV.2.2 *PROGRAMA*

El funcionamiento de la máquina consiste en que el carro se desplaza a lo largo de la cinta e imprime o borra las marcas. Este trabajo transcurre según instrucciones de determinado aspecto que se denominan *programa*. Para la máquina de Post se pueden elaborar diversos programas.

Cada uno de los programas de la máquina de Post consta de instrucciones. Llamamos *instrucción de la máquina de Post* a la expresión que tiene uno de los siguientes seis aspectos:

Primer. Instrucciones de movimiento a la derecha.

$i. \rightarrow j$

Segundo. Instrucciones de movimiento a la izquierda.

$i. \leftarrow j$

Tercer. Instrucciones de impresión de la marca.

$i. \smile j$

Cuarto. Instrucciones de borrado de la marca.

$i. \in j$

Quinto. Instrucciones de salto de control.

$i. \ ?\begin{cases} \nearrow j_1 \\ \searrow j_2 \end{cases}$

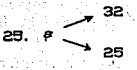
Sexto. Instrucciones de parada.

$i.$ *stop*

donde, el número i que se encuentra en el comienzo de la instrucción, se denomina *número de instrucción*; los número j, j_1 y j_2 que culminan la instruccón, los llamaremos salto, en especial j_1 es el salto superior y j_2 el salto inferior. Por ejemplo:

$$137 \rightarrow 1$$

es la instrucción de movimiento a la derecha.



es la instrucción de salto del control, mientras que

6388. *stop*

es la instrucción de parada.

Así en las instrucciones que acabamos de citar los números de instrucción son 137, 25, 6388, respectivamente. Los números de salto son 1, 32 y 25, con la particularidad de que el número 32 es el salto superior y 25, el inferior. Note además que las instrucciones de parada no tienen salto.

Denominaremos *programa para la máquina de Post* la lista finita no vacía (es decir, la que contiene al menos una instrucción) de las instrucciones de la máquina de Post, que posee las dos propiedades siguientes:

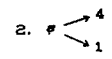
1) En primer lugar en esta lista se encuentra la instrucción con el número 1, en el segundo (si existe), la instrucción con el número 2, etc.; en general, en el k-ésimo lugar está la instrucción con el número k.

2) El salto de cualquiera de las instrucciones que figuran en la lista coincide con el número de cierta instrucción, que puede ser otra o la misma.

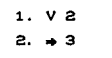
Por ejemplo, la siguiente lista será el programa para la máquina de Post:

1. V 2
2. → 3
3. ? 4, 1
4. *stop*

En tanto que las siguientes dos listas no servirán de programas para la máquina de Post, aunque están compuestas de instrucciones válidas:



(no está cumpliendo la primera condición);



(no está cumpliendo la segunda condición);

IV.2.3 *FUNCIONAMIENTO*

Para que la máquina de Post comience a trabajar es necesario prefijar, primeramente, cierto programa y en segundo lugar, cierto estado de ella, es decir, distribuir de cualquier modo las marcas por las células de la cinta (en particular, se pueden dejar todas las células vacías) y colocar el carro frente a una de las células. Como regla, supondremos que en el estado inicial de la máquina, el carro siempre se instala frente a la célula con el número (coordenada) cero. Llegando a este acuerdo, el estado inicial de la máquina queda definido completamente por el estado de la cinta.

El funcionamiento de la máquina a base del programa prefijado (también con el estado inicial asignado) transcurre del modo siguiente. La máquina se pone en el estado inicial y comienza a cumplir la primera instrucción del programa. En general cada instrucción se cumple en un sólo paso, mientras que el tránsito después de haber ejecutado una instrucción, al cumplimiento de la otra, transcurre según la siguiente regla.

Si se tiene que en el k-ésimo paso se cumplió la instrucción con el número i, en el paso $k+1$ se cumplirá:

- j si la instrucción sólo tiene un salto ($i. \Rightarrow j$)
- j_1 ó j_2 si la instrucción tiene 2 saltos ($i. ?$ → j_1, j_2)
- Ninguna si no tiene salto alguno la instrucción ($i.$ *stop*). La máquina se detiene.

Ahora queda por explicar qué significa "cumplir la instrucción " y cuál de los saltos, cuando hay dos, se elige como número de la instrucción siguiente.

El cumplimiento de la instrucción de movimiento a la izquierda o derecha consiste en que el carro se desplaza una célula en la dirección indicada.

El procedimiento para ejecutar la instrucción de impresión de la marca consiste en que el carro la escribe en la célula observada; esta instrucción se puede cumplir sólo en caso de que la mencionada célula, antes de ejecutar la instrucción, esté vacía, pero si en la misma ya se encuentra la marca, la instrucción se considera irrealizable. El cumplimiento de la instrucción para borrar la marca consiste en que el carro la elimina en la célula observada; ésta instrucción puede ejecutarse sólo en caso de que dicha célula esté marcada, sin embargo, si la

célula en cuestión no contiene marca, la instrucción también se considera irrealizable.

El cumplimiento de la instrucción de salto del control con salto superior j_1 e inferior j_2 no cambia de manera alguna el estado de la máquina, ninguna de las marcas se elimina ó se imprime; además el carro queda inmóvil (la máquina realiza, por así decirlo, un "paso sin movimiento"). Si la célula observada está vacía, antes de comenzar el cumplimiento de ésta instrucción, entonces acto seguido debe ejecutarse la instrucción con el número j_1, si por el contrario, esta célula contenía la marca, la siguiente instrucción que deberá realizarse será la número j_2.

El cumplimiento de la instrucción de parada tampoco cambia de ninguna manera el estado de la máquina y consiste en que la máquina se detiene.

Si se pone en marcha la máquina después de prefijar el programa y cualquier estado inicial, , se realizará una de las tres variantes siguientes

1) Al realizar el programa, la máquina alcanza el cumplimiento de la instrucción de parada; el programa se considera ejecutado, la máquina se para y ocurre la llamada *parada de resultados*.

2) Durante la ejecución del programa la máquina llega al cumplimiento de una instrucción irrealizable (impresión de la marca en una célula no vacía o borrado de la marca en la célula vacía), cesa el cumplimiento del programa, la máquina se para y ocurre la llamada *parada sin resultados*.

3) Durante la ejecución del programa la máquina no llega hasta el cumplimiento de ninguna de las instrucciones indicadas en

las dos primeras variantes; con ello, el cumplimiento del programa nunca cesa, la máquina nunca se para; el proceso de funcionamiento de la máquina transcurre infinitamente.

IV.2.4 *EJEMPLO*

Vamos a prefijar, por ejemplo, el estado inicial indicado en la Fig 4.10 y el siguiente programa:

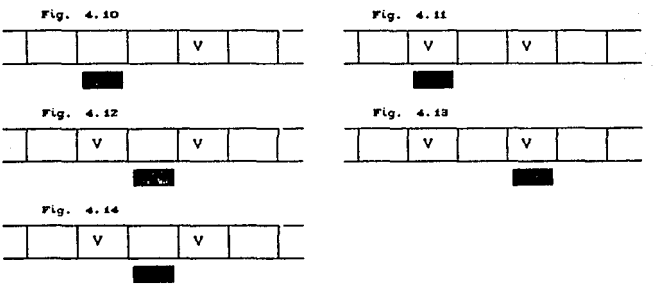
1. V 4
2. ← 3
3. ← 2
4. → 5
   4
5. ?
   3

Veamos como trabajará la máquina con estas condiciones.

En el primer paso se ejecutará la instrucción No. 1, esto es, escribir una marca y pasar a la instrucción No. 4 realizado ésto el estado de la máquina será el de la figura 4.11; durante el segundo paso se moverá la cabeza una célula a la derecha (Fig. 4.12) y la próxima instrucción a realizar será la No. 5. Ésta se realiza en el tercer paso, a resultas del cual, el estado de la máquina no cambiará y quedará tal como se muestra en la figura 4.12.

En vista de que la célula observada en este caso está vacía, a continuación debe cumplirse la instrucción, cuyo número es igual al salto superior, es decir, al número 4. Durante el cuarto paso se ejecutará esta instrucción con lo cual la cabeza se moverá una célula a la derecha (Fig. 4.13). Ahora la instrucción No. 5 se ejecutará en el quinto paso, como en esta ocasión la célula observada está marcada, se ejecutará la instrucción cuyo número es igual al salto inferior, es decir, la número 3.

En el sexto paso se cumplirá la instrucción No. 3. y la máquina llegará al estado que se muestra en la figura 4.14. Al tratar de cumplir la instrucción No. 2 en el séptimo paso es claro que es irrealizable, puesto que ordena borrar la marca en la célula vacía, por lo tanto, durante el séptimo paso ocurrirá una parada sin resultados.



Movimientos de la máquina de Post

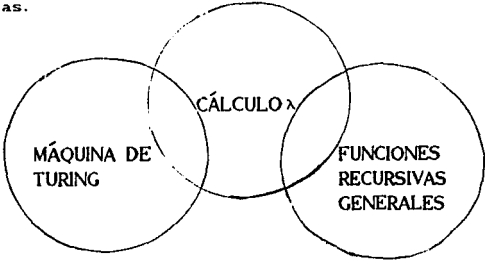
Los diversos programas aplicados al mismo estado inicial pueden llevar a distintos finales: a paradas sin resultados o de resultados, o bien al funcionamiento de la máquina sin paradas (infinito).

IV.3 LA TESIS DE CHURCH

En 1936 el lógico estadounidense Alonzo Church formuló una amplia tesis que expresaba precisamente qué se quería decir con *calcular*. Este concepto fue llamado *computabilidad efectiva*, lo cual significaba: cualquier proceso o procedimiento realizado

gradualmente por reglas bien definidas. Church creyó que había encontrado la mejor forma de definirlo por medio de un sistema formal llamado el *Cálculo* λ. Su tesis sostenía que cualquier cosa que pudiera ser llamado *efectivamente computable*, puede ser considerado dentro del *Cálculo* λ .

Cuando la tesis de Church apareció, ya había otro sistema formal que representaba algo similar, es decir, la clase de funciones llamadas *recursividad general*. Por esta misma época, apareció una tercera afirmación, llamada *La máquina de Turing*. Un juicio a priori de éstos tres paradigmas de cómputo podría representarse por un diagrama de Venn, que muestra algunas intersecciones entre los conceptos, pero también algunas diferencias.



La tesis de Church, mostraba que las funciones recursivas generales eran precisamente aquellas formadas por el *Cálculo* λ. Entonces, Turing demostró que la máquina que él tenía definida era equivalente al *Cálculo* λ de Church.

IV.3.1 *DEFINICIÓN*

El *Cálculo* λ es un procedimiento para definir funciones en terminos de *expresiones* λ.

Una *expresión* λ es un identificador, una cadena de expreciones λ, un número o tiene la forma

λ (expresion límite de la variable)• expresión

Una *expresión límite de la variable* es un identificador, los símbolos ( ), o una lista de identificadores.

Empezando con algunas expresiones λ elementales por números y operaciones sobre ellos, uno puede usar este formalismo (según Church) para expresar cualquier función computable.

El hecho de que los tres conceptos de computabilidad mencionados sean equivalentes fortalece la evidencia en favor de la tesis de Church. De cualquier modo, parece que no hay forma de definir un mecanismo de cualquier tipo que calcule más de lo que una máquina de Turing es capaz de calcular. Ningún esquema general calculará una función que no sea recursiva. Ningún procedimiento efectivo saldrá del espacio de aplicación del *Cálculo* λ.

A la inversa, las fallas de cualquier esquema general (como lo es el problema de detener la Máquina de Turing) es un fallo para todos. La tesis de Church, pone aparentemente un límite natural sobre lo que pueden hacer las computadoras: todas las computadoras (suficientemente generales) son creadas igual.

V. Implementacion de varias máquinas de Turing

V.1 *IMPLEMENTACION DE LA MAQUINA DE TURING*

El elemento de proceso básico del sistema *PriPar* es una computadora muy primitiva que se comunica con otras computadoras similares de tal forma que ellas pueden cooperar para realizar un cálculo. Esta máquina primitiva se deriva de la máquina de Turing.[6]

La máquina de Turing es aún uno de los conceptos básicos de la ciencia computacional, gracias al trabajo de Turing que demostró que con un tiempo suficiente, cualquier cosa que es computable puede ser calculada por este dispositivo primitivo. Todas las computadoras reales son, de alguna forma, implementaciones de la máquina de Turing básica. Este proyecto, que podría llamarse una *Computadora Primitiva Paralela (PriPar)*, se realizó en Turbo Pascal 5.5 de Borland

En especial esta simulación es una versión simple que sólo puede leer o escribir dos símbolos, un espacio en blanco y una marca (representada por el /) y la cinta es infinita sólo hacia el lado derecho.

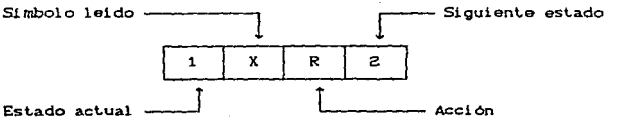
El diagrama de Flujo No. 1 del apéndice muestra la definición de esta máquina de Turing básica. Las primeras cuatro asignaciones corresponden a las partes de una máquina de Turing clásica: estado (STATE), posición actual de la cabeza lectora (HP), instrucciones del programa (PROG), y la cinta (TAPE).

Las siguientes tres son para propósistos puramente prácticos como el conteo de ciclos de máquina (CC), HALTED como señal de que la máquina ha entrado a un estado de detención y TRACING para imprimir en pantalla cada una de las acciones que se realicen, estos realmente no son parte de la máquina de Turing.

Para una sóla máquina de Turing es posible llamar al procedinmiento *halt* para detener la simulación, pero en una máquina paralela con varios procesadores, no se puede permitir que un sólo procesador detenga a todos los demas.

Los estados están representados por números, las letras B y X representan Blanco y Marca, L y R implican mover la cabeza un lugar a la izquierda o a la derecha respectivamente. Así, una instrucción como 1XR2 se traduce como "*Si estds en el estado* 1 y *hay una marca en la cinta, mueve la cabeza lectora/escritora una posición a la derecha y cambia al estado* 2" (Fig. 5.1). En esta versión de la máquina, el único estado de detención es 0.

Figura 5.1



Estructura de una instrucción para la máquina de Turing

V.2 *MAQUINAS PRIMITIVAS PARALELAS (PRIPAR)*

La máquina PriPar es descendiente de la máquina de Turing añadiendo dos acciones extras a su repertorio de instrucciones. (Diagrama 2 del apéndice). Las dos acciones nuevas son: !,que significa envia, e ?, que significa recibe.

Las acciones de enviar y recibir permiten una acción sincronizada entre dos o más máquinas PriPar. Una instrucción como 1B?2 puede recibir mensajes de cualquier número de máquinas, pero una instrucción como 1X!2 puede enviar datos sólo a una máquina, es decir, la comunicación es de varias a una.

Estas instrucciones actúan como condicionales: Si ellas tuvieran éxito, entonces se cambia al siguiente estado; si no tuvieran éxito, entonces la máquina permanece en el mismo estado. Para una instrucción !, éxito significa que su mensaje es leído por otra máquina; para ?, éxito sería que llegara un mensaje.

Las conexiones entre las máquinas se crean al asignar a la variable REMOTE de cada máquina, la dirección de un compañero remoto, con el que tendrá comunicación.

V.3 *DETENCION*

¿Cómo se puede detener un programa que usa instrucciones de enviar y recibir? En general la respuesta es: no se puede. Una máquina PriPar que recibe un flujo de mensajes de otra, quedará en un estado de espera infinito cuando el flujo cese, y no exite forma directa para romper éste estado.

Cuando todas las máquinas PriPar en una configuración particular están esperando, el sistema se encuentra en una condición que los programadores en paralelo llaman **deadlock** *(detención)*: ninguna de las máquinas puede seguir por que no hay mensajes que sean mandados. Normalmente para el programador en paralelo el *deadlock*es lo peor que puede suceder, pero esto se puede manejar para que sea de utilidad.

Para utilizar el *deadlock* para terminar programas es necesario reconocer el estado de espera desde afuera del procedimiento CICLO. Cuando el programa principal detecta que todas las máquinas estan esperando entonces termina la ejecución del mismo.

V.4 *PROGRAMACION DE LAS MAQUINAS PRIPAR*

Para programas que se pueden resolver en paralelo, las máquinas PriPar son más fáciles de programar que las máquinas de Turing.

Con pequeños programas se pueden obtener grandes logros, por ejemplo, con un conjunto de operaciones elementales que pueden ser combinadas (listado 3 del apéndice) se pueden obtener las cuatro funciones aritméticas sencillas.

El más simple es el programa llamado *Inject*, el cual corre a través de su cinta y manda un mensaje por cada marca que lee hasta que alcance un Blanco. El flujo de salida de mensajes de *Inject* es usada para controlar otros programas.

El programa *Multiply* lee un número de su propia cinta y un flujo de mensajes de entrada y entonces produce un flujo de mensajes igual al producto. *Divide*, *Subtract*, y *Lesser* trabajan de forma similar.

Note que únicamente los enteros positivos pueden ser representados, por lo tanto *Subtract* no regresará ningun valor cuando el substraendo es el número mayor. *Divide* regresará el cociente entero de la división.

El programa *Tally* acepta cualquier número de flujos de entrada y escribe una marca en su cinta por cada mensaje recibido, actuando como unificador, sumador e impresor en uno sólo.

No es necesario un programa de suma o Add, ya que dos o más flujos enviados a la misma entrada serán automaticamente acumulados. Por lo que, *Multiply*, *Divide*, y *Subtract* pueden recibir entradas desde varias máquinas para calcular el producto, cociente o diferencia. En contraste, el programa *Lesser* debe tener

únicamente una entrada y produce el mínimo entre el contenido de su cinta y la entrada.

V.5 *FUNCIONAMIENTO DE LAS MAQUINAS PRIPAR*

Las máquinas PriPar trabajando en paralelo son más eficientes en tiempo que las máquinas de Turing. Por ejemplo, un programa para la máquina de Turing que multiplica 9*2 tarda 308 ciclos mientras que *Multiply* con 3 máquinas PriPar tarda 82 ciclos para realizar el mismo trabajo.

Las máquinas PriPar no necesitan repetir pasos sobre la misma cinta como lo hace la máquina de Turing secuencial y muchas operaciones están superpuestas en el tiempo. Por ejemplo, *Tally* comienza a escribir su salida mientras *Inject* está aún revisando la entrada.

V.6 *PROGRAMA PRINCIPAL*

Como ya se mencionó el programa principal realiza operaciones aritméticas básicas (suma, resta, multiplicación y división). Por ejemplo, si se desea multiplicar 5 * 2, se asigna el número 5 a la máquina uno (los números se representaran por '/') y el número 2 a la máquina 2 y el resultado se almacenará en la máquina 3 después de ejecutar el programa *Multiply* del listado 3 del apéndice.

En la parte inferior de la pantalla aparece el dato *No. de Ciclos* al terminar el cálculo, éste representa el número de acciones que se realizaron para obtener el resultado.

Así, los datos de entrada serán el número de máquinas, los números que contendrán cada máquina y la forma en que se desea operar las mismas.

Ahora bien, como nuestra forma habitual de escribir expresiones aritméticas con el operador entre sus operandos (forma infija) es un poco ilógica para la computadora, ya que la instrucción: *Tome el número 5 y multiplíquelo por...* es incompleta mientras no se dé el segundo factor, conviene entonces convertir esta expresión a: *Tome los números 5 y 2 y luego multiplíquelos.* Este método de escribir todos los operandos antes de sus operadores recibe el nombre de *Notación Polaca Inversa, forma Sufija* o *Postfija.* [13]

En la forma infija, la prioridad de los operandos se puede alterar utilizando parentesis y por lo tanto no es lo mismo escribir 3*5+2 que 3*(5+2), el resultado de la primera expresión es 17 y de la segunda es 21.

La Notación Polaca Inversa es la mejor forma para trabajar con computadoras debido a su rapidez de evaluación, pues no es preciso hacer varios barridos a través de la expresión para descifrarla, y la síntesis en sí de la expresión, dado que esta notación no usa paréntesis.

Para evaluar las expresiones en Notación Polaca Inversa se toma el operador más a la izquierda y se evalúa con los dos operandos que le siguen en ese mismo sentido (izquierda) sustituyendo a éstos por el resultado. Ejemplos:

	Notación Infijo	Notación Polaca Inversa
1)	3 * (5 + 2) 3 * 7 21	3 5 2 + * 3 7 * 21
2)	3 * 5 + 2 15 + 2 17	3 5 * 2 + 15 2 + 17

VI. Ejemplos y diferentes corridas con diferentes programas

A continuacion se muestran los resultados que se obtienen en la computadora al realizar las corridas de los programas con diferentes datos de entrada.

En primer lugar se muestra la corrida de los programas de la Maquina de Turing tanto en su implementación serial como en paralelo.

Además, se implementaron otros programas tomando como base la teoría de los Algoritmos Genéticos, para optimización de funciones. Estos se desarrollaron también, en forma serial y paralela.

IV.1 *EJEMPLOS CON MÁQUINAS DE TURING*

EJEMPLO 6.1

Este es un ejemplo de una multiplicación utilizando sólo una máquina de Turing. El programa realiza la multiplicacion de cualquier número por dos.

DATO DE ENTRADA : 3

El cual esta representado por las marcas (/) en la figura 6.1.

Después de realizar la operación, el resultado de multiplicar 3*2 queda como muestra la figura 6.2.

Durante el proceso de multiplicación se van borrando uno a uno los datos de entrada, es por eso que al finalizar el cálculo, sólo queda sobre la cinta el resultado (6 marcas).

Figura 6.1

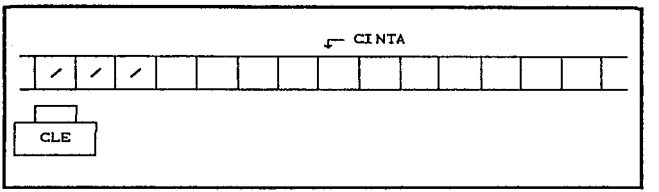
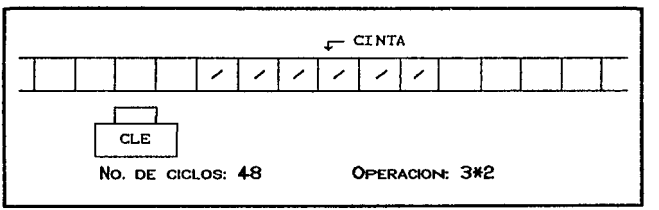


Figura 6.2

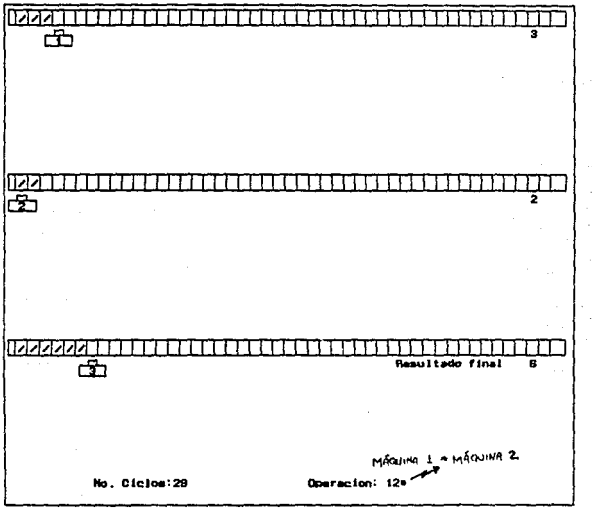


Multiplicación con una máquina de Turing

El No. de Ciclos es el número de acciones que tuvo que realizar la cabeza de la máquina para llegar al resultado, estas acciones pudieron ser: escribir marca, borrar marca o moverse un cuadro a la derecha o a la izquierda.

Por otro lado, si se resuelve la multiplicación anterior por medio de las máquinas Primitivas Paralelas (PriPar) obtendremos, al finalizar el cálculo, la pantalla 6.3:

Pantalla 6.3



Multiplicación con varias máquinas en paralelo

Como se puede observar, el número de ciclos utilizados en éste ejemplo es casi la mitad del utilizado por la máquina sencilla. La Tabla 6.1 muestra el número de ciclos que necesita una sóla máquina de Turing para resolver diferentes multiplicaciones y los que utiliza la máquina PriPar para resolver la misma operación.

Si se grafican los datos de ésta tabla (Gráfica 6.1), se puede ver que la reducción del número de ciclos al utilizar dos o más máquinas es considerable respecto de utilizar una sola y ésto

genera una reducción en el tiempo de resolución de problemas más complicados como lo muestran los siguientes ejemplos.

Tabla 6.1

Número de ciclos necesarios para resolver cada multiplicación

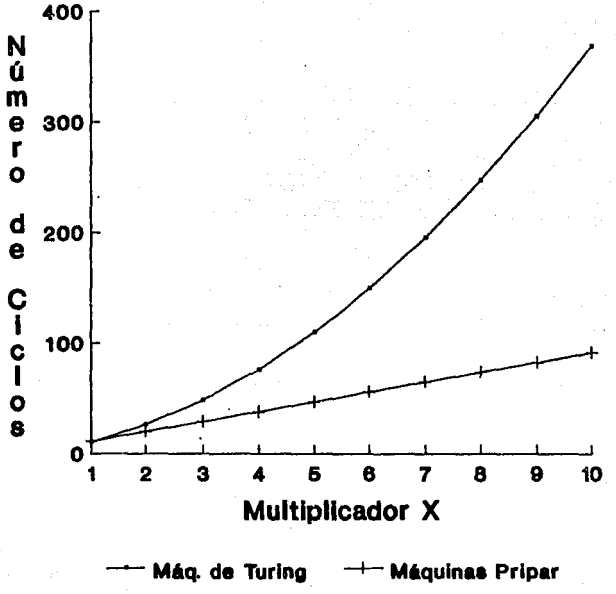
Operacion	Numero de ciclos	
	Maq. de Turing	Maq. PriPar
1 * 2	10	11
2 * 2	26	20
3 * 2	48	29
4 * 2	76	38
5 * 2	110	47
6 * 2	150	56
7 * 2	196	65
8 * 2	248	74
9 * 2	306	83
10 * 2	370	92

Las siguientes son pantallas que muestran los resultados de algunas operaciones. Los numeros que se encuentran en el extremo derecho es el numero de marcas que contiene cada cinta.

Los números dentro de las cabezas representan el número de cada máquina. En la esquina inferior derecha se encuentra el número de cada maquina y la operación que se realizó con ellas, esta operación se encuentra en notación polaca[1]. Por ejemplo 1 2 * , significa que se multiplicara el contenido de la máquina uno por el contenido de la máquina dos.

[1] Para simplificación de texto se usará "Notación Polaca" cuando se hable de la Notación Polaca Inversa.

Gráfica comparativa
operación 2 * X



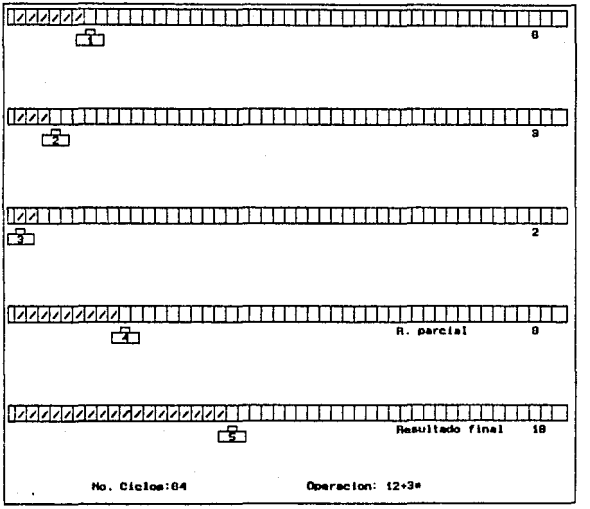
Gráfica 6.1

EJEMPLO 6.2

Operación realizada: (6 + 3) * 2
Notación polaca: 6 3 + 2 *

RESULTADO: 18

Pantalla 6.4



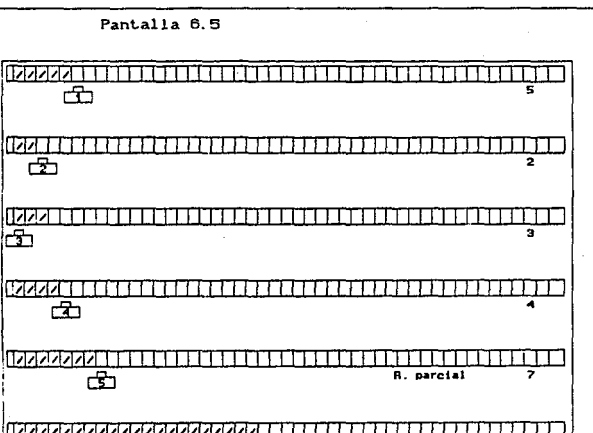
Las tres primeras máquinas son los datos de entrada, en la cuarta máquina se encuentra el resultado de la operación que se encuentra entre parentesis (6 + 3) y en la última cinta se encuentra el resultado final (18).

EJEMPLO 6.3

Operación realizada: (5 + 2) * 3 - 4
Notación polaca: 5 2 + 3 * 4 -

RESULTADO: 17

Pantalla 6.5



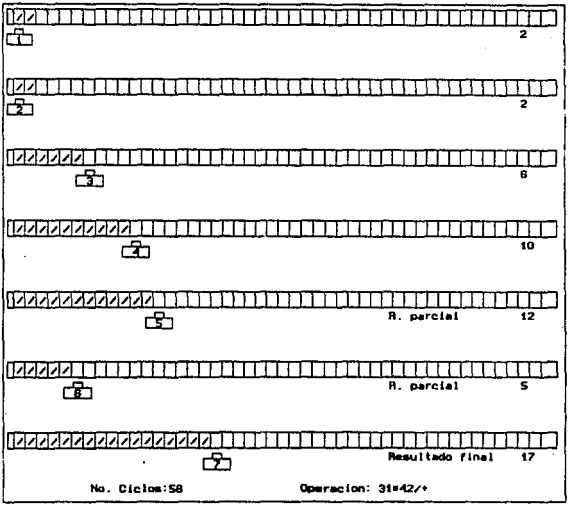
En éste caso fué necesario obtener dos resultados parciales, primero (5 + 2) y después la multiplicación de este por 3, y por último a éste (21) se le resta el contenido de la cinta número cuatro, obteniendo así el resultado (17).

EJEMPLO 6.4

Operación realizada: 6 * 2 + 10 / 2
Notación polaca: 6 2 * 10 2 / +

RESULTADO: 17

Pantalla 6.6



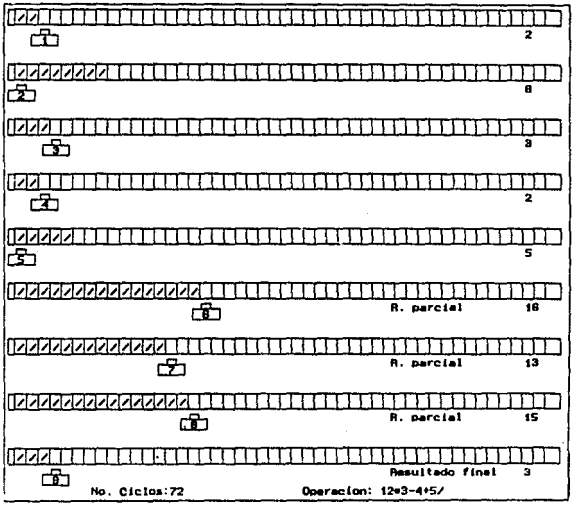
Como podemos ver en la pantalla 6.6, no importa como se den los datos a las diferentes máquinas, lo importante es que los números de las máquinas se introduzcan de forma que operen como uno lo desea; en éste caso el resultado final es correcto.

EJEMPLO 6.5

Operación realizada: ((8 * 2) - 3 +)/ 5
Notación polaca: 8 2 * 3 - 2 + 5 /

RESULTADO: 3

Pantalla 6.7



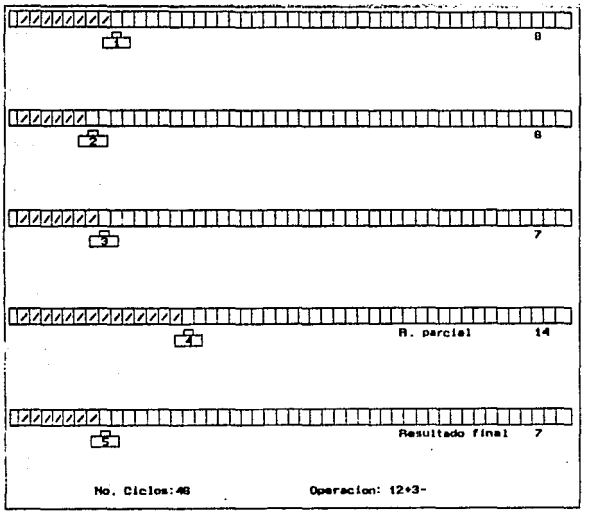
Como se mostró en la Tabla 6.1, una máquina de Turing sencilla necesita 248 ciclos para realizar la multiplicación de 8 * 2, en éste ejemplo sólo se necesitaron 72 ciclos para realizar además de esa misma multiplicación, una resta, una suma y una división. Esto es una reducción de tiempo bastante significativa.

EJEMPLO 6.8

Operación realizada: 8 + 6 - 7
Notación polaca: 8 6 + 7 -

RESULTADO: 7

Pantalla 6.8



En la pantalla 6.8 se puede apreciar que éstas máquinas trabajando en paralelo pueden resolver cualquier tipo de operaciones ya sean sencillas o complicadas, en muy poco tiempo.

VI.2 *ALGORITMOS GENÉTICOS*

Los organismos vivos poseen destreza consumada en la resolución de problemas, obtienen sus habilidades a través de mecanismos como la evolución y la selección natural.

Las analogías entre la computación y la biología son algo más que coincidencias: tanto los genes como los registros de la computadora, copian y diseminan información.

Los investigadores más pragmáticos consideran que hay que emular la eficacia de la evolución al crear programas capaces de resolver problemas que nadie comprende por completo - programas heurísticos -. Algoritmos de este tipo, denominados *genéticos*, han demostrado su capacidad para abrir nuevas brechas en el diseño de sistemas complejos.

Holland[1] notó que el aprendizaje puede ocurrir no sólo por la adaptación en un organismo aislado sino también por la adaptación evolutiva sobre muchas generaciones de una especie. Su trabajo se inspiró en las investigaciones de Darwin sobre la evolución en la cual se advierte que sólo los más aptos sobreviven.

En casi todos los organismos, la evolución se produce a través de dos procesos primarios: la selección natural y la reproducción sexual. La primera determina qué miembros de la población sobrevivirán hasta reproducirse; la segunda garantiza la mezcla y recombinación de sus genes entre la descendencia.

La selección constituye un proceso sencillo: cuando un organismo falla en alguna prueba de aptitud, como el reconocimiento y consiguiente huida de un depredador, perece. Por su parte, los informaticos no tienen dificultad en eliminar los

[1] John H. Holland. GENETIC ALGORITHMS. Scientific American. Vol 267 No.1 pp. 44-50. July 1992.

algoritmos de bajo rendimiento o funcionamiento deficiente.

En la fusión del óvulo y el espermatozoide los cromosomas homólogos se entrecruzan en zonas intermedias intercambiando así material genético. Con esta mezcla y cruzamiento, los seres vivos evolucionan a velocidad mucho mayor que si cada descendiente contuviera una mera copia de los genes de un único progenitor, modificado a veces por simple mutación.

Para muchos problemas prácticos en ingeniería y ciencia el único camino seguro para encontrar una solución optima es buscar a través de todo el conjunto de todas las posibles soluciones. Una prueba tan exhaustiva es descrita como explorar todo el "*espacio paramétrico*" del problema.

En muchos casos el espacio paramétrico es tan grande que solo una pequeña parte puede ser explorado. Entonces surge la pregunta ¿Como puede uno organizar la búsqueda para que exista una alta probabilidad de localizar la mejor aproximación de la solución óptima?

Los algoritmos genéticos permiten la explotación de un rango mucho más amplio de posibles soluciones que los programas tradicionales. La aproximación tradicional es refinar iterativamente una posible solución hasta que el refinamiento heurístico no produce una mayor mejora.

Los algoritmos genéticos de búsqueda toman una aproximación diferente. Inspirado por la evolución biológica, las posibles soluciones se cruzan y permite solo a las "mejores" soluciones, sobrevivir después de muchas generaciones [4].

En su forma más sencilla, una búsqueda genética trabaja como sigue. Primero, el problema es formulado de tal modo que cualquier solución pueda ser codificada en una cadena de dígitos binarios. A cada cadena se le asigna un valor de aptitud (fitness), basado en que tan bien la solución correspondiente satisface alguna meta fijada.

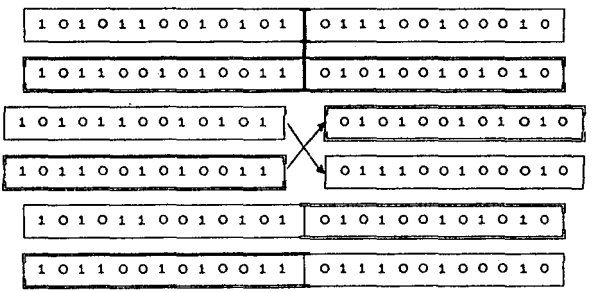
Comenzando con una población de cadenas, una nueva población del mismo tamaño es generada en dos etapas, llamadas mutación y cruza. En la etapa de mutación, la probabilidad que tiene cada cadena de ser mutada es inversamente proporcional a la "aptitud" de la cadena. Esto implicaría que algunos de los 0 cambiarán por 1 o viceversa.

La mutación, por sí sola, no suele generar progresos en la búsqueda de una solución, pero si proporciona garantías contra una población uniforme e incapaz de posterior evolución.

La etapa de cruza simula la recombinación de elementos genéticos que puede ser posible por modos sexuales de reproducción. La cruza comienza con la selección de un entero aleatorio mayor que cero y menor a la longitud de la cadena, definiendo con eso un punto de cruza. Dos cadenas son creadas juntando el prefijo de una cadena con el sufijo de la otra cadena a partir del punto de cruza.(Fig. 6.3)

Los hijos no remplazan a los padres, pero sí sustituyen a las cadenas de *aptitud* más baja las cuales se van eliminando de tal forma que la población permanece constante.

Figura 6.3



Ejemplo de cruza de dos cadenas

VI.3 *EJEMPLOS DEL ALGORITMO GENÉTICO*

VI.3.1 *El problema del paso de fluido*

Un problema sencillo dónde se puede aplicar el algoritmo genético es *"El problema del paso de un fluido por una tubería"*.

Supongase una tubería que cuenta con un número variable de puertas (Fig. 6.4), las cuales regulas la cantidad de líquido que puede pasar al tomar, cada puerta, uno de los siguientes tres estados:

|	....	cerrado
/	....	parcialmente cerrado
–	....	abierto

Si la puerta toma el primer estado, no permite pasar fluido, si toma el segundo, deja pasar media unidad y si toma el tercero, permite pasar una unidad de fluido.

Por lo tanto la cantidad de fluido que pasará por la tubería esta determinado por la sumatoria de las cantidades que permiten pasar cada una de las purtas (Fig. 6.5).

El problema sería determinar en que estado deben estar cada una de las puertas para permitir pasar la mayor cantidad de fluido.

En terminos de conjuntos cada uno de los diferentes arreglos que pueden hacerse con una parte de los elementos o con todos los elementos de un conjunto se llama *Permutación*.

Tomemos ahora el siguiente

COROLARIO: Si X acciones pueden efectuarse sucesivamente de P maneras diferentes cada una, entonces el número total de maneras diferentes en qu pueden efectuarse las X acciones sucesivas es P^X. [14]

En este caso tenemos N número de puertas, las cuales pueden tener 3 posibles valores cada una, por lo tanto, tendriamos que examinar 3^N Permutaciones como máximo para obtener la óptima.

Se intenta obtener la cadena que representa la posición óptima de las puertas por medio del Algoritmo Genetico analizando sólo un pequeño número de las permutaciones posibles.

Figura 8.4

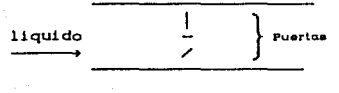


Figura 8.5

REPRESENTACION DE ARRIBA A ABAJO: 0 2 1

CANTIDAD QUE PASA
0+1+.5 = 1.5

Representación de la tubería

Para esto se deben tomar las consideraciones siguientes:

- Se crea la primera generación aleatoriamente y a partir de ella se obtiene las demás generaciones por cruza.
- Cada generación esta compuesta de un número variable de tuberias.
- Cada tubería esta representada por una cadena.
- Cada cadena contiene la representación de las posiciones de las puertas y la cantidad de liquido que puede pasar por esa tubería (Fig. 8.5), de acuerdo con la siguientes reglas:

POSICION DE LA PUERTA	REPRESENTACION	CANTIDAD DE FLUIDO
|	0	0
/	1	0.5
-	2	1

Como se explicó anteriormente, lo que hace el algoritmo genético es tomar dos cadenas e intercambiar parte de la información que contienen para lograr una mejor combinación.

En esta simulación se implementaron cuatro procedimientos diferentes para realizar este intercambio o cruza.[12]

Primero: Toma el punto medio de ambas cadenas como punto de cruza y une el prefijo de la primera cadena con el sufijo de la segunda y viceversa. Ejemplo:

Cadena 1	1 2 | 1 0 1	*hijo* 1	1 2 2 0 1
Cadena 2	0 0 | 2 0 1	*hijo* 2	0 0 1 0 1

Segundo: Este procedimiento genera un número aleatorio enter 0 y la longitud de la cadena, el cual será el punto de cruza; a partir de aquí se realiza el intercambio de información. Ejemplo:

Cadena 1	1 | 2 1 0 1	*hijo* 1	1 0 2 0 1
Cadena 2	0 | 0 2 0 1	*hijo* 2	0 2 1 0 1

Tercero: Aquí se divide en tres a las dos cadenas y se reacomoda la información de la siguiente manera:

hijo 1 - 2a. parte cadena 1 + 3a. parte cadena 2 + 1a. parte cadena 1.

hijo 2 - 2a. parte cadena 2 + 3a. parte cadena 1 + 1a. parte cadena 2.

Cuarto: En este caso se genera un número aleatorio como punto de cruza para la primera cadena y otro número aleatorio para la segunda, y se realiza el intercambio. Sin embargo, en la mayoría de los casos se obtiene cadenas de diferentes longitudes (una más grande y otra más corta) y ya que todas las cadenas deben tener el mismo tamaño, a la cadena larga se le trunca el pedazo sobrante y a la corta se le agregan número aleatorios hasta llegar al tamaño. Ejemplo:

		No. aleatorios:		
Cadena 1	1 | 0 2 2 1	1	*hijo* 1	1 0 2 0 1
Cadena 2	1 1 0 | 0 2	3	*hijo* 2	1 1 0 0 2

Para cada generación se toma la mejor cadena para cruzarla con las siguientes (total de cadenas / 2) y dos cadenas son generadas por lo que en terminos de genetica es llamado *Mutación*, esto es, tomar algunos de los elementos de las cadenas y cambiarlos aleatoriamente, lo cual permite que en algun momento se de una mejora dentro de la generación.

VI.3.2 *PROCESO*

Para comenzar se debe deterinar los siguientes datos:

- Cuantas puertas tendra una tubería (NP).
- Cuantas cadenas tendra una generación (NCG).
- De que forma se realizará la cruza (TIPO).

Con estos datos se obtiene la primera generación, se ordena (este proceso se hace con todas las generaciones), de mayor a menor según la cantidad de fluido que deja pasar cada cadena (tubería) y se cruzan hasta que se llega, por lo menos con una cadena, al óptimo ó hasta que se han obtenido 60 generaciones.

EJEMPLO 6.7

Los datos de entrada para este ejemplo fueron:

Numero de Puertas por tuberia: **4**
Numero de cadenas por generacion: **10**
Forma de realizar la cruza:
"Partiendo cada cadena en puntos diferentes."

El numero de permutaciones posibles son: **81 cadenas**

Por medio del algoritmo genético se obtuvo la combinación optima en: **3 generaciones**

El número de permutaciones analizadas como máximo fue: **30 cadenas**

La Tabla 6.2 muestra la mejor cadena de cada generación.

Tabla 6.2

GENERACION	CANTIDAD QUE PASA		POSICION DE LAS PUERTAS
1	3.0	=>	//--
2	3.0	=>	-//-
3	4.0	=>	----

Mientras menor es el numero de puertas dentro de la tubería, se necesita un menor número de generaciones para obtener la cadena ótima.

PARAMETROS
POSICION **VALOR**
| -> Cerrado -> 0 No pasa fluído
- -> Abierto -> 1 Pasa una unidad de fluído.
/ -> Semicerrado -> .5 Pasa media unidad de fluído.

La cantidad que pasa por el tubo es igual a la sumatoria de los valores de las posiciones de las puertas.

EJEMPLO 6.8

Los datos de entrada para este ejemplo fueron:

Numero de Puertas por tuberia: 10
Numero de cadenas por generacion: 12
Forma de realizar la cruza:
"Partiendo cada cadena en tres partes."

El numero de permutaciones posibles son: 59,049 cadenas

Por medio del algoritmo genético se obtuvo la combinación optima en: 6 generaciones

El número de permutaciones analizadas como máximo fue: 72 cadenas

La Tabla 6.3 muestra la mejor cadena de cada generación.

Tabla 6.3

GENERACION	CANTIDAD QUE PASA		POSICION DE LAS PUERTAS
1	7.0	=>	-//|/----/
2	8.0	=>	-----/--/|
3	9.5	=>	-----/-----
4	9.5	=>	-----/-----
5	9.5	=>	------/----
6	10.0	=>	----------

PARAMETROS
POSICION **VALOR**
| -> Cerrado -> 0 No pasa fluido
\- -> Abierto -> 1 Pasa una unidad de fluido.
/ -> Semicerrado -> .5 Pasa media unidad de fluido.
La cantidad que pasa por el tubo es igual a la sumatoria de los valores de las posiciones de las puertas.

EJEMPLO **6.9**

Los datos de entrada para este ejemplo fueron:

Numero de Puertas por tuberia: 9
Numero de cadenas por generacion: 12
Forma de realizar la cruza:
"Partiendo cada cadena exactamente a la mitad."

El numero de permutaciones posibles son: **19,683 cadenas**

Por medio del algoritmo genético se obtuvo
la combinación optima en: **37 generaciones**

El número de permutaciones analizadas
como máximo fue: **444 cadenas**

La Tabla 6.4 muestra la mejor cadena de cada generación.

Tabla 6.4

GENERACION	CANTIDAD QUE PASA		POSICION DE LAS PUERTAS
1	6.0	=>	-/||---/-
:	:		:
3	7.0	=>	-///---/-
:	:		:
36	8.0	=>	-----//--
37	9.0	=>	---------

Cuando se utiliza este método de cruza, la posibilidad de mejorar la especie depende de las mutaciones que se de en algunas de las cadenas. Los tiempos de ejecición fueron:

SECUENTIAL EXECUTION TIMR: 366529
PARALLEL EXECUTION TIME: 190382
SPEEDUP: 1.93
NUMBER OF PROCESSORS USED: 14

PARAMETROS

POSICION		VALOR	
|	-> Cerrado ->	0	No pasa fluído
-	-> Abierto ->	1	Pasa una unidad de fluído.
/	-> Semicerrado ->	.5	Pasa media unidad de fluído.

La cantidad que pasa por el tubo es igual a la sumatoria de los valores de las posiciones de las puertas.

EJEMPLO 6.10

Los datos de entrada para este ejemplo fueron:

Numero de Puertas por tuberia: 10
Numero de cadenas por generacion: 12
Forma de realizar la cruza:
"Partiendo cada cadena en puntos diferentes"

El numero de permutaciones posibles son: 59,049 cadenas

Por medio del algoritmo genético se obtuvo la combinación optima en: 4 generaciones

El número de permutaciones analizadas como máximo fue: 48 cadenas

La Tabla 6.5 muestra la mejor cadena de cada generación.

Tabla 6.5

GENERACION	CANTIDAD QUE PASA		POSICION DE LAS PUERTAS
1	6.5	=>	----||-///
2	8.0	=>	-------||--
3	8.0	=>	-||-------
4	10.0	=>	----------

Este ejemplo se realizó con el programa en paralelo con el cual se obtuvieron los siguientes tiempos de ejecución:

SECUENTIAL EXECUTION TIMR: 101472
PARALLEL EXECUTION TIME: 73923
SPEEDUP: 1.37
NUMBER OF PROCESSORS USED: 14

PARAMETROS

POSICION **VALOR**

|	-> Cerrado ->	0	No pasa fluído
-	-> Abierto ->	1	Pasa una unidad de fluído.
/	-> Semicerrado ->	.5	Pasa media unidad de fluído.

La cantidad que pasa por el tubo es igual a la sumatoria de los valores de las posiciones de las puertas.

VI.3.3 *TRABAJADORES APLICADOS*

En la mayoría de los algoritmos paralelos, el número de pasos computacionales se conoce de antemano y esto permite repartir los datos ó el trabajo en cierto número de procesadores de forma equilibrada.

Sin embargo, existen ciertas clases de algoritmos en los cuales no se conoce el número específico de pasos computacionales para resolverlos. Los algoritmos de búsqueda combinatorial caen en esta categoría (el espacio de solución de un problema es analizado por caminos de búsqueda parciales que son creados y descartados dinámicamente).

En este tipo de algoritmos, el cálculo no puede ser dividido inicialmente entre los procesadores. Para lograr un balance, los pasos del cálculo deben ser asignados a los procesadores de forma dinámica tan pronto como sean generados durante la ejecución del programa. La mejor forma de lograrlo es usando un nuevo tipo de paradigma de programación paralela llamada REPLICATED WORKERS [5] (*Trabajadores Aplicados*). Los procesos son asignados a trabajadores idénticos para ejecutarse en cada uno de los procesadores físicos y los pasos del cálculo son asignados dinámicamente a los trabajadores tan pronto como el programa es ejecutado.

En el paradigma de los Trabajadores Aplicados, se usa una estructura de datos llamada WORKPOOL, la cual es una colección de *descriptores* que especifican un paso particular de cálculo que puede ser realizado por cualquier trabajador.

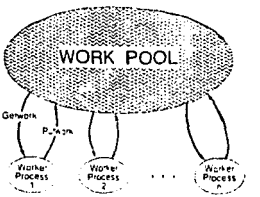
Cuando un trabajador se encuentra desocupado, recibe un nuevo descriptor de paso del Workpool y entonces realiza el cálculo requerido. Durante el proceso de un paso, el trabajador puede generar nuevos pasos, los cuales son añadidos al Workpool.

Para terminar la ejecución de los Trabajadores Aplicados, todos los trabajadores deben estar desocupados y el Workpool debe estar vacio.

WORKPOOL

El paradigma de programación de Trabajadores Aplicados se puede ilustrar a través de un diagrama como muestra la figura 6.6. Un grupo de *N* trabajadores idénticos tienen acceso a un Workpool centralizado. Los trabajadores corren en paralelo en diferentes procesadores.

Figura 6.6



Representación de los trabajadores aplicados

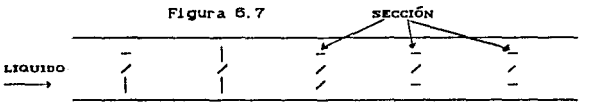
Cuando cualquier trabajador termina su tarea y se encuentra en la posibilidad de realizar un nuevo paso, el trabajador realiza una operación *Getwork* para obtener un nuevo paso de cálculo del Workpool.

En el proceso de realizar un paso, un trabajador puede generar pasos de cálculo adicionales, los cuales son anadidos al Workpool usando una operacion *Putwork*.

Cuando el Worpool se encuentra vacio, todos los trabajadores terminan al no haber mas pasos a ejecutar y una respuesta final puede ser ensamblada.

Este paradigma puede ser aplicado al problema del paso de fluido al suponer que la tuberia cuenta con más de una sección de puertas en su interior (Fig. 6.7).

Figura 6.7



Representación de una Tubería con 5 secciones de 2 puertas cada una

Como el número de Generaciones para obtener la solución para una sección de puertas es aleatorio y diferente en cada caso se ha planteado la posibilidad de crear varios trabajadores y por medio de ellos resolverlo.

La Tabla 6.6muestra en resumen las corridas del programa con los diferentes datos, el valor que aparece en el extremo derecho es el porcentaje de velocidad relacionada con el número de procesadores utilizados. Para todos los ejemplos y por limitaciones de memoria, la población de cadenas fue de 16 y el número máximo de Generaciones fue 17.

Tabla. 6.6

Ejem	NP	SEC	Número Trabajadores	Velocidad	Número Procesadores	% Vel
9	6	5	3	4.72	28	17
10			2	3.92	19	21
11		6	3	4.88	28	17
12			2	3.87	19	20
13		7	3	5.26	28	18
14			2	3.10	19	16
15	7	5	3	4.54	28	16
16			2	2.87	19	15
17		6	3	5.02	28	18
18			2	3.51	19	18
19	10	5	3	4.58	28	16
20			2	3.24	19	17

Porcentaje de velocidad relacionado con el número de procesadores utilizados.

EJEMPLO 6.9

Por medio del Algoritmo Genetico Messy se han producido:
16 cadenas por generación
6 Es la cantidad maxima que podría pasar

SEC.	TOTAL GEN	GENERACION	CANTIDAD QUE PASA DE FLUIDO		POSICION DE LAS PUERTAS
1	16	1	3.5	=>	/////-
		5	4.0	=>	/-/-//
		6	4.5	=>	/-/-/-
2	16	1	4.5	=>	-/-///-
3	16	1	3.5	=>	//-1-/
		4	4.0	=>	-/-/1-
		5	4.5	=>	-/-/-/
4	16	1	4.0	=>	/--1-/
		4	4.5	=>	/--/-/
5	4	1	5.0	=>	-//---
		4	6.0	=>	------

SEQUENTIAL EXECUTION TIME: 569474
PARALLEL EXECUTION TIME: 120601
SPEEDUP: 4.72
NUMBER OF PROCESSORS USED: 28

EJEMPLO 6.10

Por medio del Algoritmo Gentico Messy se han producido:
16 cadenas por generación
6 Es la cantidad máxima que podría pasar

SEC.	TOTAL GEN	GENERACION	CANTIDAD QUE PASA DE FLUIDO		POSICION DE LAS PUERTAS
1	16	1	3.5	=>	/////-
		5	4.0	=>	/-/-//
		6	4.5	=>	/-/-/-
2	16	1	4.5	=>	-/-//-
3	16	1	4.0	=>	-//-1-
		3	4.5	=>	/--/-/
4	4	1	5.0	=>	-//---
		4	6.0	=>	------
5	16	1	3.5	=>	-//-1/
		3	4.0	=>	/1--/-
		4	4.5	=>	/-/--/

SEQUENTIAL EXECUTION TIME: 567619
PARALLEL EXECUTION TIME: 144658
SPEEDUP: 3.92
NUMBER OF PROCESSORS USED: 19

EJEMPLO 6.11

Por medio del Algoritmo Genetico Messy se han producido:
16 cadenas por generación
6 Es la cantidad maxima que podría pasar

SEC.	TOTAL GEN	GENERACION	CANTIDAD QUE PASA DE FLUIDO		POSICION DE LAS PUERTAS
1	5	1	5.0	=>	//----
		5	6.0	=>	------
2	4	1	4.5	=>	-/1---
		4	6.0	=>	------
3	8	1	4.0	=>	//-1--
		4	5.0	=>	//----
		8	6.0	=>	------
4	16	1	3.5	=>	/-1-//
		5	4.0	=>	///-/-
		9	4.5	=>	/-/-/-
5	16	1	3.5	=>	-//-/1
		2	4.0	=>	/-//-/
		4	4.5	=>	/-/--/
6	16	1	3.5	=>	-//-/1
		2	4.0	=>	/-//-/
		4	4.5	=>	/-/--/

SEQUENTIAL EXECUTION TIME: 572402
PARALLEL EXECUTION TIME: 117221
SPEEDUP: 4.88
NUMBER OF PROCESSORS USED: 28

EJEMPLO 6.12

Por medio del Algoritmo Genetico Messy se han producido:
16 cadenas por generación
6 Es la cantidad maxima que podria pasar

SEC.	TOTAL GEN	GENERACION	CANTIDAD QUE PASA DE FLUIDO		POSICION DE LAS PUERTAS
1	5	1	5.0	=>	//----
		5	6.0	=>	------
2	4	1	4.5	=>	-/1---
		4	6.0	=>	------
3	16	1	4.0	=>	-//-1-
		4	4.5	=>	/-/--/
4	7	1	4.5	=>	//---/
		4	5.0	=>	----//
		7	6.0	=>	------
5	16	1	4.0	=>	-//-1-
		2	4.5	=>	/--/-/
6	13	1	4.0	=>	1/---/
		4	4.5	=>	-/-/-/
		6	5.0	=>	-/-/--
		9	5.5	=>	-/----
		13	6.0	=>	------

SEQUENTIAL EXECUTION TIME: 537766
PARALLEL EXECUTION TIME: 140285
SPEEDUP: 3.83
NUMBER OF PROCESSORS USED: 19

EJEMPLO 6.13

Por medio del Algoritmo Genetico Messy se han producido:
16 cadenas por generación
6 Es la cantidad maxima que podría pasar

SEC.	TOTAL GEN	GENERACION	CANTIDAD QUE PASA DE FLUIDO		POSICION DE LAS PUERTAS
1	8	1	4.5	=>	//---/
		5	5.0	=>	----//
		6	5.5	=>	---/--
		8	6.0	=>	------
2	16	1	5.0	=>	---//-
		5	5.5	=>	/-----
3	16	1	3.0	=>	1/-1/-
		4	3.5	=>	///-1-
		7	4.0	=>	/-/-//
		10	4.5	=>	/-/-/-
4	16	1	4.5	=>	-/-//-
		2	5.0	=>	-/---/
		8	5.5	=>	-/----
5	16	1	3.0	=>	///1-/
		3	3.5	=>	-///-1
		4	4.0	=>	//-/-/
		8	4.5	=>	-/-/-/
6	16	1	3.0	=>	-//-11
		2	4.0	=>	/--1-/
		4	4.5	=>	/-/--/
7	16	1	3.0	=>	///1-/
		2	3.5	=>	-1-///
		4	4.0	=>	//-/-/
		8	4.5	=>	-/-/-/

SEQUENTIAL EXECUTION TIME: 910783
PARALLEL EXECUTION TIME: 173245
SPEEDUP: 5.26
NUMBER OF PROCESSORS USED: 28

EJEMPLO 6.14

Por medio del Algoritmo Genetico Messy se han producido:
16 cadenas por generación
6 Es la cantidad maxima que podría pasar

SEC.	TOTAL GEN	GENERACION	CANTIDAD QUE PASA DE FLUIDO		POSICION DE LAS PUERTAS
1	8	1	4.5	=>	//---/
		5	5.0	=>	----//
		6	5.5	=>	---/--
		8	6.0	=>	------
2	16	1	5.0	=>	---//-
		5	5.5	=>	/------
3	7	1	4.5	=>	-/--//
		4	5.5	=>	-----/
		7	6.0	=>	------
4	5	1	4.5	=>	///---
		5	6.0	=>	------
5	16	1	3.5	=>	-/1-//
		3	4.0	=>	1--/-/
6	7	1	3.5	=>	1/--//
		4	4.5	=>	----1/
		7	6.0	=>	------
7	16	1	3.5	=>	/-1-//
		3	4.0	=>	1-/-/-

SEQUENTIAL EXECUTION TIME: 653892
PARALLEL EXECUTION TIME: 210642
SPEEDUP: 3.10
NUMBER OF PROCESSORS USED: 19

EJEMPLO 6.15

Por medio del Algoritmo Genetico Messy se han producido:
16 cadenas por generación
7 Es la cantidad maxima que podría pasar

SEC.	TOTAL GEN	GENERACION	CANTIDAD QUE PASA DE FLUIDO		POSICION DE LAS PUERTAS
1	16	1	4.0	=>	///1/--
		4	4.5	=>	1//--/-
		5	5.0	=>	/--/--1
		6	5.5	=>	-//--/-
		9	6.0	=>	---/--/
		15	6.5	=>	-----/-
2	10	1	5.0	=>	//-/--/
		4	5.5	=>	//--/--
		5	6.0	=>	--/---/
		8	6.5	=>	/------
		10	7.0	=>	-------
3	16	1	3.5	=>	//-1///
		5	4.0	=>	/-///-1
		8	4.5	=>	//-///-
4	16	1	3.5	=>	///1//-
		4	4.0	=>	-1//-//
		7	5.0	=>	/--//-
		13	5.5	=>	---/-//
		14	6.0	=>	-/-/---
5	8	1	5.0	=>	--//1--
		4	5.5	=>	-/1----
		5	6.0	=>	-----//
		7	6.5	=>	--/----
		8	7.0	=>	-------

SEQUENTIAL EXECUTION TIME: 648363
PARALLEL EXECUTION TIME: 142682
SPEEDUP: 4.54
NUMBER OF PROCESSORS USED: 28

EJEMPLO 6.16

Por medio del Algoritmo Genetico Messy se han producido:
16 cadenas por generación
7 Es la cantidad maxima que podría pasar

SEC.	TOTAL GEN	GENERACION	CANTIDAD QUE PASA DE FLUIDO		POSICION DE LAS PUERTAS
1	16	1	4.0	=>	///1/--
		4	4.5	=>	1//--/-
		5	5.0	=>	/--/--1
		6	5.5	=>	-//--/-
		9	6.0	=>	---/--/
		15	6.5	=>	-----/-
2	10	1	5.0	=>	//-/--/
		4	5.5	=>	//--/--
		5	6.0	=>	--/---/
		8	6.5	=>	/------
		10	7.0	=>	-------
3	16	1	4.0	=>	-/1-///
		5	4.5	=>	///-//-
		11	5.0	=>	/--//-/
		14	5.5	=>	-/--//-
4	10	1	4.0	=>	-/-1/1-
		2	4.5	=>	-1/-/-/
		3	5.0	=>	/---/-1
		6	5.5	=>	----1/-
		7	6.5	=>	---/---
		10	7.0	=>	-------
5	16	1	4.5	=>	//-1/--
		4	5.0	=>	-//--//
		8	5.5	=>	-/--//-
		11	6.0	=>	--/---/
		15	6.5	=>	-----/-

SEQUENTIAL EXECUTION TIME: 687104
PARALLEL EXECUTION TIME: 239655
SPEEDUP: 2.87
NUMBER OF PROCESSORS USED: 19

EJEMPLO 6.17

Por medio del Algoritmo Genetico Messy se han producido:
16 cadenas por generación
6 Es la cantidad maxima que podría pasar

SEC.	TOTAL GEN	GENERACION	CANTIDAD QUE PASA DE FLUIDO		POSICION DE LAS PUERTAS
1	16	1	5.0	=>	//----1
		2	5.5	=>	--/--//
		5	6.0	=>	/-/----
		6	6.5	=>	/------
2	14	1	4.5	=>	-/-//-1
		4	5.0	=>	-//-/-/
		8	5.5	=>	/---//-
		11	6.5	=>	-----/-
		14	7.0	=>	-------
3	16	1	4.0	=>	//-1-//
		4	4.5	=>	-/-////
		5	5.0	=>	-/-//-/
		10	5.5	=>	/---//-
		12	6.0	=>	/---/--
		15	6.5	=>	----/--
4	16	1	6.0	=>	//-----
5	16	1	4.0	=>	//-//-1
		3	4.5	=>	-/-////
		6	5.0	=>	-//-/-/
		10	5.5	=>	/---//-
		13	6.5	=>	-----/-
		16	7.0	=>	-------
6	14	1	4.0	=>	-1//-//
		3	4.5	=>	-/--11-
		4	5.5	=>	---//-/
		8	6.0	=>	-//----
		9	6.5	=>	-/-----
		14	7.0	=>	-------

SEQUENTIAL EXECUTION TIME: 885541
PARALLEL EXECUTION TIME: 176430
SPEEDUP: 5.02
NUMBER OF PROCESSORS USED: 28

EJEMPLO 6.18

Por medio del Algoritmo Genetico Messy se han producido:
16 cadenas por generación
7 Es la cantidad maxima que podría pasar

SEC.	TOTAL GEN	GENERACION	CANTIDAD QUE PASA DE FLUIDO		POSICION DE LAS PUERTAS
1	16	1	5.0	=>	//----1
		2	5.5	=>	--/--//
		5	6.0	=>	/-/-----
		6	6.5	=>	/------
2	14	1	4.5	=>	-/-//-1
		4	5.0	=>	-//-/-/
		8	5.5	=>	/---//-
		11	6.5	=>	-----/-
		14	7.0	=>	-------
3	11	1	5.5	=>	-/--//-
		2	6.0	=>	---/--/
		8	6.5	=>	-----/-
		11	7.0	=>	-------
4	16	1	3.5	=>	///1//-
		4	4.0	=>	-1//-//
		6	4.5	=>	-/--1//
		7	5.5	=>	---//-/
		15	6.5	=>	-----/-
5	13	1	4.0	=>	-//-11-
		4	4.5	=>	/-1/--/
		7	5.5	=>	//--/--
		11	6.5	=>	--/-----
		13	7.0	=>	-------
6	8	1	4.5	=>	1-/1---
		3	5.0	=>	---1-//
		4	6.0	=>	-1-----
		7	6.5	=>	-/-----
		8	7.0	=>	-------

SEQUENTIAL EXECUTION TIME: 778734
PARALLEL EXECUTION TIME: 221912
SPEEDUP: 3.51
NUMBER OF PROCESSORS USED: 19

EJEMPLO 6.19

Por medio del Algoritmo Genetico Messy se han producido:
16 cadenas por generación
10 Es la cantidad maxima que podría pasar

SEC.	TOTAL GEN	GENERACION	CANTIDAD QUE PASA DE FLUIDO		POSICION DE LAS PUERTAS
1	10	1	6.0	=>	//----111-
		2	8.5	=>	----/--//-
		6	9.5	=>	-/--------
		10	10.0	=>	----------
2	14	1	5.5	=>	-/-1/1-///
		3	7.0	=>	-//1-/--/-
		7	8.5	=>	--/--/---/
		11	9.5	=>	------/---
		14	10.0	=>	----------
3	16	1	5.0	=>	/11-1//-/-
		4	6.0	=>	-/1/-/--1/
		9	8.5	=>	/-///-/--1
		13	7.0	=>	/---/-/1/-
		14	7.5	=>	-/-///-/--
4	16	1	6.5	=>	--1/1--/-/
		2	7.0	=>	//--/-/--1
		4	8.0	=>	/---/-/-/-
		7	8.5	=>	-/---/-/--
		14	9.0	=>	/-----/---
5	16	1	6.5	=>	//-1/--1--
		5	7.0	=>	-1--//--//
		7	7.5	=>	--/--1---/
		10	8.0	=>	/----//--/
		16	9.0	=>	---/-----/

SEQUENTIAL EXECUTION TIME: 881024
PARALLEL EXECUTION TIME: 192217
SPEEDUP: 4.58
NUMBER OF PROCESSORS USED: 28

EJEMPLO 6.20

Por medio del Algoritmo Genetico Messy se han producido:
16 cadenas por generación
10 Es la cantidad maxima que podría pasar

SEC.	TOTAL GEN	GENERACION	CANTIDAD QUE PASA DE FLUIDO		POSICION DE LAS PUERTAS
1	10	1	6.0	=>	//----111-
		2	8.5	=>	----/--//-
		6	9.5	=>	-/--------
		10	10.0	=>	----------
2	14	1	5.5	=>	-/-1/1-///
		3	7.0	=>	-//1-/--/-
		7	8.5	=>	--/--/---/
		11	9.5	=>	------/---
		14	10.0	=>	----------
3	13	1	6.5	=>	-/--//-11-
		2	7.5	=>	-//-//--/-
		6	8.5	=>	--/--/---/
		10	9.5	=>	------/---
		13	10.0	=>	----------
4	11	1	6.5	=>	1////---/-
		2	7.0	=>	--1--/-/-1
		4	8.0	=>	--/--/---1
		7	8.5	=>	------1-/-
		10	9.5	=>	---/------
		11	10.0	=>	----------
5	16	1	6.5	=>	-///1//---
		4	8.0	=>	/-//---/--
		6	8.5	=>	-/--/--/--
		12	9.0	=>	-----/---/

SEQUENTIAL EXECUTION TIME: 777838
PARALLEL EXECUTION TIME: 239689
SPEEDUP: 3.24
NUMBER OF PROCESSORS USED: 19

CONCLUSIONES

Una de las revoluciones por las que está atravesando la ciencia de la computación es: la introducción en el campo de trabajo, de las computadoras paralelas.

Si bien, la computación paralela implementada tanto en software como en hardware, son un cambio sustancial a la computación clásica, los principios de programación, es decir, el uso de las máquinas y no de su diseño, siguen conservando una cantidad sustancial y mayoritaria de técnicas y teorías de programación clásica; ésto se pudo ver en la implementación y desarrollo de máquinas de Turing en paralelo, lo cual permite demostrar en forma empírica, cómo los principios clásicos de la teoría de las funciones recursivas de Turing, Post y Church, pueden ser implementados y se pueden tener ventajas sustanciales cuando se les trabaja en paralelo.

Durante el desarrollo de ésta tesis se presentó cómo las computadoras paralelas son un continuo que van desde las totalmente paralelas en hardware hasta las totalmente paralelas en software, y se demostró que son muchos los caminos que en la actualidad se están desarrollando para hacer avanzar la teoría y la práctica, y no se puede restringir a simplemente esperar tener máquinas paralelas en hardware, para poder tener desarrollos sustanciales en éste campo.

Es por eso que se presentan dos simulaciones de procesos paralelos: uno por medio de máquinas de Turing y otro de Pascal extendido paralelo, para demostrar específicamente varias técnicas teóricas como la implementación en paralelo de procedimientos y herramientas nuevas, como es el caso particular del Algoritmo Genético.

El Algoritmo Genético es una técnica general para resolver problemas matemáticos lineales de optimización, esto desde un punto de vista matemático aplicado es una herramienta muy

poderosa, ya que en muchos casos el formalismo matemático que describe el problema es no analítico, y en la mayoría de los casos importantes y significativos, el problema es no lineal, por lo que la solución matemática directa es muy difícil.

En este caso se presentó un problema no lineal de optimización para el cual los procedimientos combinatorios simples son extraordinariamente exaustivos, y de hecho el problema es NP. Sin embargo, con la utilización del algoritmo genético y con su implementación en paralelo se logró resolver éstos problemas de una manera eficiente.

El hecho de haber escogido dos técnicas, en cierta forma extremas, como el algoritmo genético que en su escencia es intrínsecamente paralela; y por el otro lado haber tomado la máquina de Turing, que conceptualmente se le caracterizaba como la herramienta y teoría básica de la computación clásica y haberlas implementado en paralelo demuestra entre otras cosas:

a) Que las diferencias entre técnicas paralelas de programación y técnicas clásicas no son tan grandes como se podría suponer.

b) Que si una persona conoce perfectamente bien un lenguaje como puede ser Pascal y conoce en general lo que es la teoría computacional clásica, puede pasar a trabajar en computación paralela de una manera muy rápida y directa.

c) Que es posible y además muy práctica la implementación de procedimientos paralelos en máquinas clásicas con fines didácticos ó de investigación.

d) Que es posible simular diferentes formas de máquinas paralelas para investigación o con fines didácticos.

g) Que es posible, en estos simuladores especiales,

resolver problemas computacionales de un grado bastante alto de complejidad.

Todas éstas conclusiones nos llevan a lo que es el objetivo central de la tesis, que es: que la computación paralela, en sus muy diversas formas, es un desarrollo sustancial dentro de la disciplinas al cual "hay que entarle" sin esperar a que la tecnología nos rebase y en cierta forma nos quedemos ignorantes de la computación moderna.

El mejor ejemplo de ésta situación es lo que ocurrió cuando llegó la primera supercomputadora a México, en donde la gran sorpresa para muchos de los investigadores que la iban a utilizar, es que era una máquina paralela en hardware y que podía ser programada realmente en paralelo. Lo que causó ésta máquina fue un gran revuelo y hubo que reconsiderar por un lado, la necesidad de entender paralelo en toda su extensión y por el otro, que el lenguaje más importante para éstas máquinas particulares es el Fortran, el cual, estaba cayendo en el olvido.

En la actualidad, el proceso de la utilización de máquinas paralelas que ya existen en México, ha sido muy lento y difícil, ya que en la mayoría de los casos se tiene las máquinas pero no se tiene el personal humano capaz de aprovecharlas en su máxima potencia.

BIBLIOGRAFIA

[1] Adams, L. M. ITERATIVE ALGORITHMS FOR LARGE SPARSE LINEAR SYSTEMS ON PARALLEL COMPUTERS. Ph. D Thesis, University of Virginia. 1982.

[2] Akl, S. G. PARALLEL SORTING ALGORITHMS. Academic Press, New York. 1985.

[3] Baena, Guillermina. MANUAL PARA ELABORAR TRABAJOS DE INVESTIGACION DOCUMENTAL. Editores Mexicanos Unidos, México. 124 pp. 1991.

[4] Boolos, George S. & RichardC. Jeffrey. COMPUTABILITY AND LOGIC. Cambridge University Press. Great Britain. 2a. ed. 285 pp. 1980.

[5] Denning, Peter J. GENETIC ALGORITHMS. American Scientist. Enero-Febrero. E.U.A. Vol 80, pp. 12 -14. 1992.

[6] EXPERIMENTAL PARALLEL COMPUTING ACHITECTURES. Edited by J. J. Dongarra, North-Holland, Amsterdam. 1987

[7] Flores Flores, Angel G. DE LAS IDEAS DE TURING A LA COMPUTACION EN PARALELO. 010 Revista de Computación, Vol 9, No. 211, México. 1988

[8] Forsyth, Richard. MACHINE LEARNING. Principles and Techniques. Ed. Chapman and Hall Computing. New York, 255 pp. 1989.

[9] Gibbons Alan & Paul Spirakis. LECTURES ON PARALLEL COMPUTATION. Cambridge University Press. Gran Bretaña. 437 pp. 1993.

[10] Holland, John H. GENETIC ALGORITHMS. Scientific American. Vol 267, No. 1. pp. 44 - 50. July 1992.

[11] Hopcroft John E. & Jeffrey D. Ullman. INTRODUCTION TO AUTOMATA THEORY, LANGUAGES AND COMPUTATION. Addison-Wesley. E.U.A. 1979.

[12] Hwang, Kai & Briggs, Fayé A. AQUITECTURA DE COMPUTADORAS Y PROCESAMIENTO PARALELO. Ed. Mc-Graw Hill, México. 914 pp. 1988.

[13] Kasantkin V. N., SIETE PROBLEMAS DE CIBERNETICA. Kiev, (traducción). 1978.

[14] Koza, John R. GENETIC PROGRAMMING. Ed. The NIT Press. London, England. 786 pp. 1980.

[15] Kruse L., Robert. ESTRUCTURA DE DATOS Y DISEÑO DE PROGRAMAS. Ed. Prentice Hall Hispanoamericana. México, 488 pp. 1988.

[16] Lehmann, Charles H. ALGEBRA. Ed. Limusa. México 446 pp. 1964.

[17] Lester, P. Bruce. THE ART OF PARALLEL PROGRAMMING. Ed. Prentice-Hall, E.U.A. 375 pp. 1993.

[18] Lister, A.M. FUNDAMENTOS DE LOS SISTEMAS OPERATIVOS. Ed. Gustavo Gili. 2ª ed., España, 181 pp. 1986.

[19] Mandado, Enrique. SISTEMAS ELECTRONICOS DIGITALES. 3ª ed. Boixareu Editores. Barcelona, España, 503 pp. 1980.

[20] Moret, B.M.E. & H.D. Shapiro. ALGORITHMS FROM P TO NP. Vol. I Design & Efficiency. Ed The Benjamin/Cummings Publishing Co. E.U.A. 576 pp. 1991.

[21] Salas Parrilla, Jesús. SISTEMAS OPERATIVOS Y COMPILADORES. Ed. Mc-Graw Hill, España. 212 pp. 1988.

[22] Schwel H.P. & Männer R. (Eds.). PARALLEL PROBLEM SOLVING FROM NATURE. Ed Springer - Verlag. Alemania. 481 pp. 1991.

[23] Zorrilla A., Santiago & Miguel Torres X. GUIA PARA ELABORAR LA TESIS. Ed. Mc-Graw Hill. 2ª ed. México, 110 pp. 1982.

APENDICE

LISTADO: 1

```
CONST MaxOp = 80;

TYPE
   OpCode      = String[4];
   TapeStr     = String[255];
   ProgArray   = ARRAY[1..MaxOp] of  OpCode;
   remotetape =^apt;
   apt= record
        taper:tapestr;
        end;

   Turing = OBJECT
     State    : Char;
     HeadPosn: Word;
     Prog     : ProgArray;
     Tape     : remoteTape;

     CycleCount: Word;
     Halted     : Boolean;
     Tracing    : Boolean;

     CONSTRUCTOR Init(TP: remoteTape; VAR PR);
     PROCEDURE Actions(Op: OpCode); VIRTUAL;
     PROCEDURE Trace(Step: Word); VIRTUAL;
     PROCEDURE Cycle;
   END;

CONSTRUCTOR Turing.Init(TP: remoteTape; VAR PR);
BEGIN
  CycleCount := 1;
  State := '1';
  HeadPosn := 1;
  Halted := FALSE;
  Tracing := TRUE;
  Tape := TP;
  Prog := ProgArray(pr) {Permite tamaño variable del prog}
END;

PROCEDURE Turing.Actions(Op: OpCode);
BEGIN
  CASE Op[3] OF          {Escribe en cinta o xes mueve}
  'B': Tape^.taper[HeadPosn] := ' ';
  'X': Tape^.taper[HeadPosn] := '/';
  'L': Dec(HeadPosn);
  'R': Inc(HeadPosn);
  ELSE BEGIN Writeln(' Opcode ',Op,'está mal formado');
             Halt
       END;
  END;
  State := Op[4]         {Siguiente estado}
END;
```

```
PROCEDURE Turing.Trace(Step : Word);          LISTADO:  1
BEGIN                                         CONTINUACION
  ClrScr;
  Writeln(Tape^.taper);
  GotoXY(HeadPosn,2);
  Writeln('^');
  Write(' Head: ',HeadPosn:2,
        ' State: ',State,
        ' Op: ',Prog[Step],
        ' Cycle: ',CycleCount:2);
  IF Halted THEN Write('     HALTED')
            ELSE Write('          ');
  Delay(10)
END;

PROCEDURE Turing.Cycle;
VAR Temp: OpCode;
    i: Integer;

BEGIN
   Temp := State;              (Lee el estado)

   IF Tape^.taper[HeadPosn] = '/' (Lee la cinta)
         THEN  Temp := Temp + 'X'
         ELSE  Temp := Temp + 'B';

   i  := 0;                    (Busca la sig. instrucción)
   REPEAT Inc(i)
   UNTIL (Copy(Prog[i],1,2) = Temp) or (i > MaxOp);

   IF i > MaxOp
   THEN Halted := TRUE    (No se encontró)
   ELSE  BEGIN            (Se encontró)
         Actions(Prog[i]);   (Escribe o se mueve)
         IF State = '0' THEN  Halted  :=  TRUE; (Terminación
                                                exitosa)
         END;
   IF  NOT  Halted    THEN    Inc(CycleCount);
END;
```

LISTADO: 2

```
TYPE RemotePtr = ^PriPar;

   PriPar = OBJECT(Turing)
      ce1,cca1,fd1,
      NoMaq,Maqs,
      i,esp:    word;
      x,y:    integer;

      Message, Waiting: Boolean;
      Remote: RemotePtr;
      CONSTRUCTOR Init(max,i1,e1,a1,d1,x1,y1 :word;
                       TP: remoteTape;
                       VAR PR; RX: RemotePtr);
      PROCEDURE Actions(Op: OpCode); VIRTUAL;
      PROCEDURE CABEZACC:WORD);
   END;

CONSTRUCTOR PriPar.Init(max,i1,e1,a1,d1,x1,y1: word;
                        TP: remoteTape;
                        VAR PR;RX: RemotePtr);
BEGIN
  ce1:=e1;
  cca1:=a1;
  fd1:=d1;
  x:=x1;y:=y1;
  i:=i1;
  nomaq:=max;
  Message := False;
  Remote  := RX;
  Turing.Init(TP,PR)
END;

PROCEDURE PRIPAR.cabezaCc:WORD);
VAR S:STRING;
BEGIN
  STR(i,S);
  if i = Nomaq+1 then s:='R';
  setcolor(C);
  rectangle(15+x,25+y,25+x,30+y);
  rectangle(5+x,30+y,35+x,40+y);
  outtextxy(17+x,32+y,S);
END;

PROCEDURE PriPar.Actions(Op: OpCode);
VAR MAXX : INTEGER;
    MENSAJE1: string;
BEGIN
  MAXX:= GETMAXX;
  Waiting := FALSE;
  CASE Op[3] OF          (Escribe marca, mueve o comunica)

  'B': BEGIN
         setcolor(FD1);
```

LISTADO: 2
CONTINUACION

```
            outlextxy(17+((headposn-1)*13),10+y,'/');
            Tape^.taper[HeadPosn] := '/';
          END;

        'X': BEGIN
            setcolor(CE1);
            outlextxy(17+((headposn-1)*13),10+y,'/');
            Tape^.taper[HeadPosn] := '/';
          END;

        'L': BEGIN
            x:=(headposn-1)*13;
            while x > ((headposn-2)*13) do
            BEGIN
              CABEZA(CCA1);
              DELAY(3);
              CABEZA(FD1);
              DELAY(3);
              X:=X-1;
            END;
            cabeza(ccal);
            Dec(HeadPosn);
          END;

        'R': BEGIN
            x:=(headposn-1)*13;
            while x < ((headposn)*13) do
            BEGIN
              CABEZA(CCA1);
              DELAY(3);
              CABEZA(FD1);
              DELAY(3);
              X:=X+1;
            END;
            cabeza(ccal);
          Inc(HeadPosn);
          END;

        '!': BEGIN
            IF Remote^.Message
               THEN Waiting := TRUE
               ELSE Remote^.Message := True;
          END;

        '?': BEGIN
            IF Message
               THEN  Message := FALSE
               ELSE Waiting := TRUE;
          END
      ELSE BEGIN outlextxy(300,y,' Opcode '+Op+' está mal formado');
               Halt
          END;
      END;
```

LISTADO: 2
CONTINUACION

```
  IF NOT Waiting THEN State := Op[4];
  setcolor(ce1);
  IF WAITING THEN  BEGIN
                     SETCOLOR(FD1);
                     OUTTEXTXY(MAXX-100,Y+30,'          ');
                   END
             ELSE  begin
                     setcolor(fd1);
                     OUTTEXTXY(MAXX-100,Y+30,'          ');
                     Setcolor(ce1);
                     IF  (I<>NOMAQ+1)  THEN MENSAJE1 :=' ENVIA'
                                       ELSE MENSAJE1 :=' RECIBE';
                     OUTTEXTXY(MAXX-100,Y+30,MENSAJE1);
                   end;
END;
```

LISTADO: 3

```
CONST Inject: ARRAY[1..3] OF OpCode =
          ('1X!2',
           '1B?1',
           '2XR1'
          );

      Multiply: ARRAY[1..9] OF OpCode =
          ('1X?2',
           '2X!3',
           '3XB4',
           '4BR5',
           '5X!6',
           '6XR5',
           '5BL7',
           '7XL7',
           '7BX1'
          );

      Divide: ARRAY[1..9] OF OpCode =
          ('1X?2',
           '2XB3',
           '3BR4',
           '4X?5',
           '5XR4',
           '4BL6',
           '6XL6',
           '6B!7',
           '7BX1'
          );

      Subtract: ARRAY[1..4] OF OpCode =
          ('1X?2',
           '1B?3',
           '2XR1',
           '3B!1'
          );

      Lesser: ARRAY[1..4] OF OpCode =
          ('1X?2',
           '1B?1',
           '2X!3',
           '3XR1'
          );

      Tally: ARRAY[1..3] OF OpCode =
          ('1B?2',
           '1XR1',
           '2BX1'
          );
```

LISTADO: 4

```
PROGRAM Maquinas_de_Turing;
USES Crt,Graph,polaca,portada;

{$I apturing.pas}
{$I appripar.pas}
{$I program.lib}
{$I lee.pas}

VAR
    m:                          ARRAY [1..12] OF  PriPar;
    cinta :                     ARRAY [1..12] OF  TapeStr;
    tw,i,NoMaq,
    Fd,CE,CCi,CCa,
    nm,nc,TotMaq,
    MaxX,Maxy,Maqs,esp:         WORD;
    Driver,Mode:                INTEGER;
    opcion,op:                  STRING;

FUNCTION texC1:word):string;
VAR cad : string;
BEGIN
  STR(1,cad);                                        {Convierte a texto
  tex:=cad;
END;

FUNCTION valor(1:string):word;
VAR  uno: real;
     o:   integer;                              {Recupera el valor numrico}
BEGIN
  val(L,uno,o);
  Valor:=round(uno);
END;


PROCEDURE Limpia;
BEGIN                                    {Limpia el valor de las variables}
 FOR i:=1 to 12 DO
  BEGIN
   cinta[i]:='
   m[i].state:=' ';
   m[i].cyclecount:=0;
   m[i].tape:=nil;
   m[i].tape^.taper:='';
   m[i].headposn:=0;
  END;
  nm:=1;
END;


PROCEDURE Identifica;
BEGIN                                  {Identifica el tipo de pantalla}
  DetectGraph(Driver,Mode);
```

LISTADO: 4
CONTINUACION

```
  CASE Driver of
      Vga:       BEGIN Fd:= 7;
                       CE:= 11;
                       CCi:= 9;
                       CCa:=15;
                       Maqs:=8;
                 END;
      HercMono:BEGIN Fd:= 0;
                       CE:= 1;
                       CCi:=1;
                       CCa:=1;
                       Maqs:=8;
                 END;
      CGA:       BEGIN Fd:= 0;
                       CE:= 1;
                       CCi:=1;
                       CCa:=1;
                       Maqs:=8;
                 END;
  END;
END;


procedure sumaCa,b:word);
begin
  m[nm].InitCTotmaq,a,ce,cca,fd,1,(a-1)*esp,@cinta[a],inject,@m[nm+2]);
  m[nm+1].InitCTotmaq,b,ce,cca,fd,1,(b-1)*esp,@cinta[b],inject,@m[nm+2]
  m[nm+2].InitCTotmaq,nc+1,ce,cca,fd,1,esp*(nc),@cinta[nc+1],tally,nil)
  nm:=nm+3;
  nc:=nc+1;
end;

procedure restaCa,b:word);
begin
  m[nm].InitCTotmaq,a,ce,cca,fd,1,(a-1)*esp,@cinta[a],inject,@m[nm+1]);
  m[nm+1].InitCTotmaq,b,ce,cca,fd,1,(b-1)*esp,@cinta[b],subtract,@m[nm+
  m[nm+2].InitCTotmaq,nc+1,ce,cca,fd,1,esp*(nc),@cinta[nc+1],tally,nil)
  nm:=nm+3;
  nc:=nc+1;
end;

procedure multiplicaCa,b:word);
begin
  m[nm].InitCTotmaq,a,ce,cca,fd,1,(a-1)*esp,@cinta[a],inject,@m[nm+1]);
  m[nm+1].InitCTotmaq,b,ce,cca,fd,1,(b-1)*esp,@cinta[b],multiply,@m[nm+
  m[nm+2].InitCTotmaq,nc+1,ce,cca,fd,1,esp*(nc),@cinta[nc+1],tally,nil)
  nm:=nm+3;
  nc:=nc+1;
end;

procedure dividirCa,b:word);
begin
```

LISTADO: 4
CONTINUACION

```
  CASE Driver of
      Vga:       BEGIN Fd:= 7;
                       CE:= 11;
                       CCi:= 0;
                       CCa:=15;
                       Maqs:=6;
                 END;
      HercMono:  BEGIN Fd:= 0;
                       CE:= 1;
                       CCi:=1;
                       CCa:=1;
                       Maqs:=6;
                 END;
      CGA:       BEGIN Fd:= 0;
                       CE:= 1;
                       CCi:=1;
                       CCa:=1;
                       Maqs:=5;
                 END;
  END;
END;


procedure suma(a,b:word);
begin
  m[nm].Init(Totmaq,a,ce,cca,fd,1,(a-1)*esp,@cinta[a],inject,@m[nm+2]);
  m[nm+1].Init(Totmaq,b,ce,cca,fd,1,(b-1)*esp,@cinta[b],inject,@m[nm+2]
  m[nm+2].Init(Totmaq,nc+1,ce,cca,fd,1,esp*(nc),@cinta[nc+1],tally,nil)
  nm:=nm+3;
  nc:=nc+1;
end;

procedure resta(a,b:word);
begin
  m[nm].Init(Totmaq,a,ce,cca,fd,1,(a-1)*esp,@cinta[a],inject,@m[nm+1]);
  m[nm+1].Init(Totmaq,b,ce,cca,fd,1,(b-1)*esp,@cinta[b],subtract,@m[nm+
  m[nm+2].Init(Totmaq,nc+1,ce,cca,fd,1,esp*(nc),@cinta[nc+1],tally,nil)
  nm:=nm+3;
  nc:=nc+1;
end;

procedure multiplica(a,b:word);
begin
  m[nm].Init(Totmaq,a,ce,cca,fd,1,(a-1)*esp,@cinta[a],inject,@m[nm+1]);
  m[nm+1].Init(Totmaq,b,ce,cca,fd,1,(b-1)*esp,@cinta[b],multiply,@m[nm+
  m[nm+2].Init(Totmaq,nc+1,ce,cca,fd,1,esp*(nc),@cinta[nc+1],tally,nil)
  nm:=nm+3;
  nc:=nc+1;
end;

procedure dividir(a,b:word);
begin
```

LISTADO: 4
CONTINUACION

```
  m[nm].Init(Totmaq,a,ce,cca,fd,1,(a-1)*esp,@cinta[a],inject,@m[nm+1]);
  m[nm+1].Init(Totmaq,b,ce,cca,fd,1,(b-1)*esp,@cinta[b],divide,@m[nm+2]
  m[nm+2].Init(Totmaq,nc+1,ce,cca,fd,1,esp*(nc),@cinta[nc+1],tally,nil)
  nm:=nm+3;
  nc:=nc+1;
end;


PROCEDURE DatosIniciales;
var j,k:byte;
BEGIN
  ClrScr;
  marco;
  repeat
    GotoXY(8,5);Write('Con cuantas M quinas desea trabajar (2-',(Maqs-1
    Readln(NoMaq);
  until (NomaQ >= 2 ) and (NomaQ <= MaqS-1);
  GotoXY(8,7);Write('d=( / ) Cuantas marcas desea en la ');
  for i:= 1 to NoMaq do
  begin
     GotoXY(40,7+i);Write('Cinta No. ',i,': ');
     readln(j); cinta[i]:='';
     for k:=1 to j do
       cinta[i]:= cinta[i]+'/';
  end;
  nc:=NoMaq;                                    {numero de cintas inicial
  GotoXY(10,17);Write('Como desea operarlas: ');
  Readln(opcion);
  totMaq:=3*(NoMaq-1);
end;

procedure NoCintas;
var k,j,temp: word;
begin
  setbkcolor(fd);                          {Color del fondo de la pantalla
  setcolor(ccl);
  maxx:=getmaxx;
  maxy:=getmaxy;
  rectangle(0,0,maxX,maxY);                                        {marco
  outtextxy(maxx-100,40,opcion);
  esp:= round((maxY-5)/(2*nc-1));
  for  i := 1 to 2*nc-1 do
  begin
    setcolor(ccl);                                    {Dibuja las cintas
    rectangle(5,5+(esp*(i-1)),MaxX-10,20+(esp*(i-1)));
    k:=0;
    repeat
      k:=k+13;
      line(k,8+(esp*(i-1)),k,20+(esp*(i-1)));
    until k > MaxX-30;
    setcolor(CCa);                                   {Dibuja la cabeza
```

LISTADO: 4
CONTINUACION

```
    rectangle(15,25+(esp*(i-1)),25,30+(esp*(i-1)));
    rectangle(5,30+(esp*(i-1)),35,40+(esp*(i-1)));
    if i = TotMaq then  outtextxy(17,32+(esp*(i-1)),'R')
                  else  outtextxy(17,32+(esp*(i-1)),texi));
  end;
  setcolor(ce);
  for i:= 1 to Nomaq do
  begin
    temp:=length(cinta[i]);
    for j:=1 to temp do
      outtextxy(17+((j-1)*13),10+(esp*(i-1)),'/');  (Dibuja las marcas
  end;
end;

PROCEDURE Evalua;
VAR
  uno, dos: word;
  o,long: integer;
  aux:string;
  Cadena: ARRAY [1..25] OF string;
BEGIN
  opcion:='R'+opcion+'=';
  While opcion[3] <> '=' do
  Begin
      i:=2;
      while (opcion[i]<>'+') and (opcion[i]<>'*') and
            (opcion[i]<>'-') and (opcion[i]<>'/') do
            i:=i+1;
      uno:=valor(opcion[i-2]);
      dos:=valor(opcion[i-1]);
      if  opcion[i] =  '+' then suma(uno,dos) else
      if  opcion[i] =  '-' then resta(uno,dos) else
      if  opcion[i] =  '*' then multiplica(uno,dos) else
      if  opcion[i] =  '/' then dividir(uno,dos);
      aux:=tex(nc);
      opcion[i-2]:=aux[1];
      while opcion[i+1] <> '=' do
      begin
        opcion[i-1]:= opcion[i+1];
        i:=i+1;
      end;
      opcion[i+1]:=' ';
      opcion[i]:= ' ';
      opcion[i-1]:= '=';
    end;
end;

Procedure Opera;
var p:char;
Begin
  Evalua;                                   (Realiza las operaciones
```

LISTADO: 4
CONTINUACION

```
  Repeat
    tw:=0;
    for i:= 1 to Totmaq do
    begin
     m[i].cycle;
     if m[i].waiting then tw:=tw+1;
    end;
  until tw=TotMaq;
  setcolor(ce);
  for i := 1 to nc do
  begin
    j:=1;
    while cinta[i,j] ='/' do
      j:=j+1;
    OUTTEXTXY(maxx-50,(i-1)*ESP+33 ,tex(j-1));
    if (i > NoMaq) and (i < nc) then       { Imprime "resultado parcial"
       OUTTEXTXY(maxx-200,(i-1)*ESP+33,'R. parcial');
  end;
                                                  { Imprime "resultado"
  OUTTEXTXY(maxx-200,(nc-1)*ESP+33 ,'Resultado final ');
                                                  {imprime No. ciclos
  outtextxy(150,35,'No.ciclos:'+tex(m[1].cyclecount));
  p:=readkey;
End;


BEGIN
  Identifica;                                  {Checa el tipo de pantalla
  pantalla1;                                   {Presentaci"n del programa
  ClrScr;
  La('a:ejemplo.txt');                         {Explicaci"n del programa
  marco;
  GotoXY(15,22);write('Presione RETURN para continuar');readln(op);
  repeat
     limpia;                          {limpia el contenido de las variables
     DatosIniciales;                {Pregunta por operaciones y No. de cintas
     inipol(opcion);                              {Cambia a notaci"n polaca
     initgraph(Driver,Mode,'');
     NoCintas;                                {Dibuja las cintas necesarias
     opera;                                       {Realiza las operaciones
     restorecrtmode;
     clrscr;
     marco;
     gotoxy(10,10);write('deseas realizar otra operacion  (S/N): ');
     readln(op);
  until (op='n') or (op='N');
END.
```

LISTADO: 5

```
Program Algoritmo_Genetico_Messy_con_Apuntadores;
 const    c=5;
type
  Ap = ^reg;
  Reg = record
          fit: real;                    {*cantidad que pasa por el tubo*}
          ele: array [1..21] of integer; {*Pos. de las puertas*}
          l: Ap;                        {*Apuntador al sig. tubo*}
        end;
var
  ar: array [1..60] of Ap;                          {*Generaciones*}
  s,b,x,i,g ,NCG,TG,NP,Modo,NoCad ,xd,x2: integer;
  sum,sum1,sum2          : real;                    op: char;
  fin,aux,ant, recorre,auxd,aux2: ap;
  MG,misma: Boolean;
  l:spinlock;
  rnd: channel of integer;


Function  Random(a:integer):integer;
var  j,Rango,x:integer;
     Flag: Boolean;
                                        {*Función que devuelve*}
BEGIN                                   {*un numero aleatorio *}
  Rango := trunc(100/a);                {*entre 0 y  a-1      *}
  x := rnd;
  flag:= FALSE;
  for j := 1 to a do
  begin
    if (x <= rango * j) and (Flag <> TRUE ) then
    begin
      flag := true;
      ra dom := j - 1;
    end;
  end;
end;


Procedure  Randomize;
VAR i: integer;                         {*Procedimiento que *}
BEGIN                                   {*genera números    *}
  for i := 1 to 500 do                  {*aleatorios < 100  *}
  begin
      x2:= (81*xd + 89) mod 100;
      xd:= x2;
      rnd:= x2;
  end;
end;
```

LISTADO: 5
CONTINUACION

```
Procedure Inicia;
Begin
  For i:= 1 to 80 do                          {= Inicializa las =}
    Ar[i] := nil;                             {= variables      =}
  ant:=nil;  aux:=nil;  fin:= nil;
End;

Procedure Pide;
Begin                                         {=Pide los datos para=}
  Repeat                                      {=iniciar la simulación=}
    HG:=true;
    Inicia;
    writeln('Por favor indique:');
    repeat
      write('Cuantas PUERTAS en el tubo (3-20):      ');
      Readln(NP);
    until (NP>=3) and (NP<=20);
    repeat
      write('Cuantas CADENAS en cada generacion (2-50):     ');
      readln(NCG);
    until  (NCG>=2) and (NCG <=50);
    writeln('1.   Tomando el punto medio de las cadenas.');
    writeln('2.   Tomando un punto aleatorio en las cadenas.');
    writeln('3.   Tomando dos punto aleatorios en las cadenas.');
    writeln('4.   Tomando diferentes puntos en las cadenas.');
    repeat
      write('Como desea que se realice la cruza:      ');
      readln(modo);
    until (modo>=1 ) and (modo <=4);
    writeln('Estan correctos los datos? (S/N): '); readln(op);
  Until (op='S') or (op='s');
End;


Procedure OrdenaCg:integer;mueve:ap);
var aux3 : ap;
    final: boolean;
Begin
  if (ar[g] = nil) and (fin= nil) then
      begin
        ar[g]:= mueve;
        fin:= mueve;                          {=Ordena el nuevo tubo=}
        aux3:= mueve;                         {=dentro de la generación=}
        mueve^.l:=nil;
      end
      else
      begin
        if (mueve^.fit >= ar[g]^.fit) then
        begin
          mueve^.l:= ar[g];
          ar[g]:= mueve;
        end
        else
```

LISTADO: 5
CONTINUACION

```
        begin
          aux3:= ar[g];
          ant:= ar[g];
          writeln(g);
          final:=False;
          while Not final do
          begin
            IF (aux3^.fit > mueve^.fit) then
            begin
              ant:= aux3;
              aux3:= aux3^.l;
            end
            else final:= true;
            IF aux3 = nil then
                              final:= true;
          end;
          mueve^.l:= aux3;
          ant^.l:= mueve;
          if mueve^.l = nil then
            fin:= mueve;
        end;
      end;
End;

Procedure General;
var nuevo:ap;
    tam:integer;
Begin
  Forall s:= 1 to NCG grpuping 1 do

  var sum: real;                          {*Obtiene la primera *}
      b:integer;                          {*generación         *}

  begin
  lock(1);
      new(nuevo);
      sum:= 0;
      For b:= 1 to NP do
      begin
         nuevo^.ele[b]:= random(3);
         sum:= sum+ nuevo^.ele[b]/2;
      end;
      nuevo^.fit:=sum;
      Ordena(1,nuevo);
  unlock(1);
  end;
End;

Procedure cruza12(e,g:integer);
var a ,j:integer;
Begin
    For j:= 1 to e do
```

LISTADO: 5
CONTINUACION

```
    Begin
      aux1^.ele[j]:= ar[g]^.ele[j];
      sum1:= sum1 + ar[g]^.ele[j]/2;          {*Realiza la cruza*}
      aux2^.ele[j]:= recorre^.ele[j];         {*dividiendo las  *}
      sum2:=sum2+ recorre^.ele[j]/2;          {*cadenas en dos  *}
    End;                                      {*partes iguales  *}
    For j:= e+1 to NP do
    Begin
      aux1^.ele[j]:= recorre^.ele[j];
      sum1:= sum1 + recorre^.ele[j]/2;
      aux2^.ele[j]:= ar[g]^.ele[j];
      sum2:=sum2+ ar[g]^.ele[j]/2;
    End;
End;

Procedure Cruza3(g:integer);
var n,j,t: integer;
Begin
  n:=1;                                       {*Realiza la cruza*}
  t:= Trunc(NP/3);                            {*dividiendo las  *}
  For j:= (t+1) to (t+t) do                   {*cadenas en tres *}
  Begin                                       {*partes iguales  *}
    aux1^.ele[n]:= ar[g]^.ele[j];
    sum1:= sum1 + ar[g]^.ele[j]/2;
    aux2^.ele[n]:= recorre^.ele[j];
    sum2:=sum2+ recorre^.ele[j]/2;
    n:=n+1;
  End;
  For j:= (t+t+1) to NP do
  Begin
    aux1^.ele[n]:= recorre^.ele[j];
    sum1:= sum1 + recorre^.ele[j]/2;
    aux2^.ele[n]:= ar[g]^.ele[j];
    sum2:=sum2+ ar[g]^.ele[j]/2;
    n:=n+1;
  End;
  For j:= 1 to t do
  Begin
    aux1^.ele[n]:= ar[g]^.ele[j];
    sum1:= sum1 + ar[g]^.ele[j]/2;
    aux2^.ele[n]:= recorre^.ele[j];
    sum2:=sum2+ recorre^.ele[j]/2;
    n:=n+1;
  End;
End;

Procedure Cruza4(g:integer);
Var j,p,s,pos1,pos2: integer;
Begin
  repeat
    p:= random(NP-1)+1; pos1:= p+1;
  until p < (NP-1);
```

LISTADO: 5
CONTINUACION

```
  repeat
    s:= random(NP-1)+1; pos2:= s+1;
  until s <= (NP-1);
  For j:= 1 to p do
    aux1^.ele[j]:= ar[g]^.ele[j];          (*Realiza la cruza*)
  For j:= 1 to s do                        (*dividiendo cada *)
    aux2^.ele[j]:= recorre^.ele[j];        (*cadena en un lugar*)
  For j:= p+1 to NP do                     (*aleatorio diferente*)
  Begin
    if pos2 <= 20 then
    begin
      aux2^.ele[pos2]:= ar[g]^.ele[j];
      pos2:= pos2+1;
    end;
  End;
  For j:= s+1 to NP do
  Begin
    if pos1 <= 20 then
    begin
      aux1^.ele[pos1]:= recorre^.ele[j];
      pos1:=pos1+1;
    end;
  End;
  while pos1 < NP do
  Begin
    aux1^.ele[pos1+1]:= random(3);
    pos1:= pos1+1;
  End;
  while pos2 < NP do
  Begin
    aux2^.ele[pos2+1]:= random(3);
    pos2:= pos2+1;
  End;

  for j:= 1 to NP do
  Begin
    if (aux1^.ele[j]<0) or (aux1^.ele[j] >2) then
        aux1^.ele[j] := random(3);
    if (aux2^.ele[j]<0) or (aux2^.ele[j] >2) then
        aux2^.ele[j] := random(3);
    sum1:= sum1 + aux1^.ele[j]/2;
    sum2:= sum2 + aux2^.ele[j]/2;
  End;
End;

Procedure CruzaCg:integer);
var a ,p,i,j:integer;
Begin
  if (NCG mod 2)=0 then
       a:= Trunc (NCG/2)
  else  a:= Trunc (ncg/2)+1;
  recorre:=ar[g]^.1;
```

LISTADO: 5
CONTINUACION

```
    For i:= 2 to a do
    begin
      sum1:=0; sum2:=0;
      new(aux1);new(aux2);
      case modo of
           1:Begin        p:= Trunc (NP/2);       {*De acuerdo con *}
                          cruza12(p,g);  End;     {*los datos      *}
           2:Begin        p:= Random (NP-1)+1;    {*iniciales manda*}
                          cruza12(p,g);  End;     {*al tipo de cruza*}
           3: Cruza3(g);
           4: Cruza4(g);
      End;
      aux1^.fit:= sum1;
      aux2^.fit:= sum2;
      ordena(g+1,aux1);
      ordena(g+1,aux2);
      recorre:=recorre^.1;
    end;
    for j:=1 to 2 do
    begin
      sum1:=0;
      new(aux1);
      for i:= 1 to NP do
      begin
        aux1^.ele[i]:= random(3);
        sum1:= sum1+ aux1^.ele[i]/2;
      end;
      aux1^.fit:=sum1;
      ordena(g+1,aux1);
    End;
  End;

  Procedure ObtieneGeneraciones;
  Begin
    Genera1;
    x:=1;
    repeat
        fin:=nil;                          {* Obtiene las diferentes*}
        cruza(x);                          {* generaciones         *}
        aux:= ar[x];
        x:=x+1;
    until (aux^.fit >= NP) or (x>50);
    TG := x-1;
  End;

  Procedure Imprime;
  var i: integer;
  Begin
     repeat;
     writeln('Por medio del Algoritmo Genetico Messy se han producido:');
     writeln(NCG,' cadenas por generaci'n');
     writeln('La cantidad ideal de fluido es :',NP);
```

LISTADO: 5
CONTINUACION

```
    repeat
       writeC'Que generaci"n deseas ver? C1-',TG:2,'):          ');
       readlnCg);
    until Cg>=1) and Cg<= TG);
    writelnC'1. Solo la primera Cm xima cantidad)');
    writelnC'2. La primera y la #ltima ');
    writelnC'3. Todas ');
    Repeat
      writeC'Cuantas cadenas deseas ver:       ');
      readlnCNoCad);
    Until CNoCad>=1) and CNoCad <=3);
    writelnC'FITNES       CADENA');
    aux:=ar[g];
    case NoCad of
         1: Begin
                writeCaux^.fit:1:1,'      ');
                for x:= 1 to np do
                    writeCaux^.ele[x]:2);
                writeln;
            End;
         2:Begin
                for s:= 1 to NCG do
                Begin
                  if Cs=1) or Cs=NCG) then
                  begin
                    writeCaux^.fit:1:1,'      ');
                    for x:= 1 to np do
                        writeCaux^.ele[x]:2);
                    writeln;
                  End;
                  aux:=aux^.l;
                End;
            End;
         3:Begin
              for s:= 1 to NCG do
              Begin
                writeCaux^.fit:1:1,'      ');
                for x:= 1 to np do
                    writeCaux^.ele[x]:2);
                writeln;
              aux:=aux^.l;
              End;
            End;
    End;
    writeln;
    repeat
      writeC'          Deseas ver otra generaci"n? CS/N):          ');
      readlnCOP);
    Until Cop='s') or Cop='S') or Cop='n') or Cop='N');
  until Cop='n') or Cop='N');
end;
```

LISTADO: 5
CONTINUACION

```
procedure limpia;
begin
  for i:= 1 to tg do
    disposeCar[i]);                    (*Limpia las variables*)
  disposeCfin);                        (*para una segunda    *)
  disposeCant);                        (*corrida             *)
  disposeCrecorre);
  inicia;
end;


Begin                          (*Programa Principal*)

  repeat
    Fork randomize;
    x1:= c*3;
    pide;
    ObtieneGeneraciones;
    misma:=false;
    Imprime;

    repeat
      writeln('                    Deseas generar mas cadenas (S/N):');
      readln(op);
    Until op='s') or op='S') or op='n') or op='N');

    limpia;
    join;
  until op='n') or op='N');
End.
```

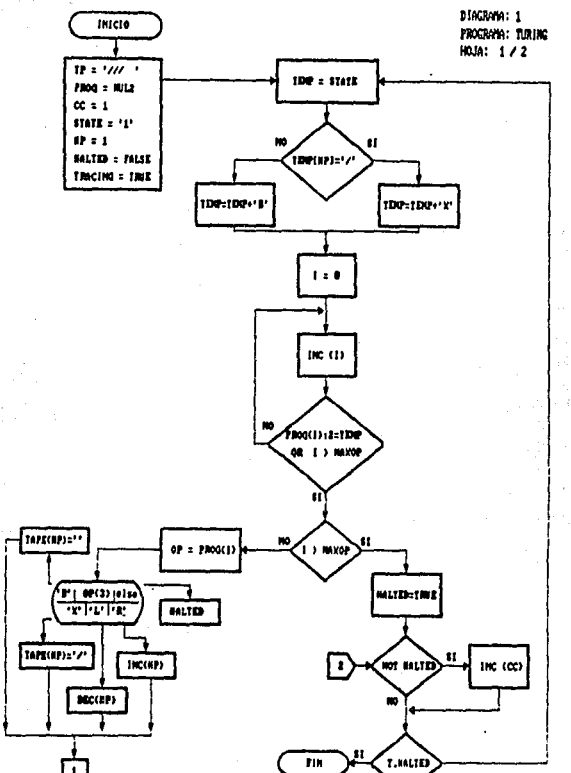


DIAGRAMA: 1
PROGRAMA: TURING
HOJA: 1 / 2
INICIO
TP = '/// '
PROG = NULL
CC = 1
STATE = '1'
HP = 1
HALTED = FALSE
TRACING = TRUE
TEMP = STATE
TEMP[HP]='/'
NO
SI
TEMP=TEMP+'B'
TEMP=TEMP+'X'
I = 0
INC (I)
PROG(I):#:TEMP OR I > MAXOP
NO
SI
I > MAXOP
NO
SI
OP = PROG(I)
TAPE(HP)=''
'B'| OP(2)|else 'X'|'L'|'R'
HALTED
HALTED=TRUE
TAPE(HP)='/'
INC(HP)
DEC(HP)
I
2
NOT HALTED
SI
NO
INC (CC)
FIN
SI
T.HALTED

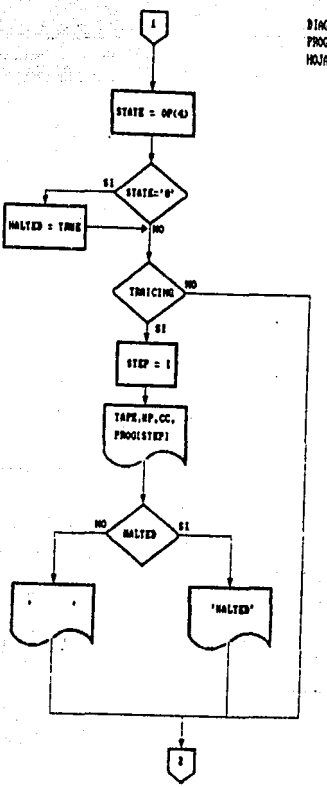
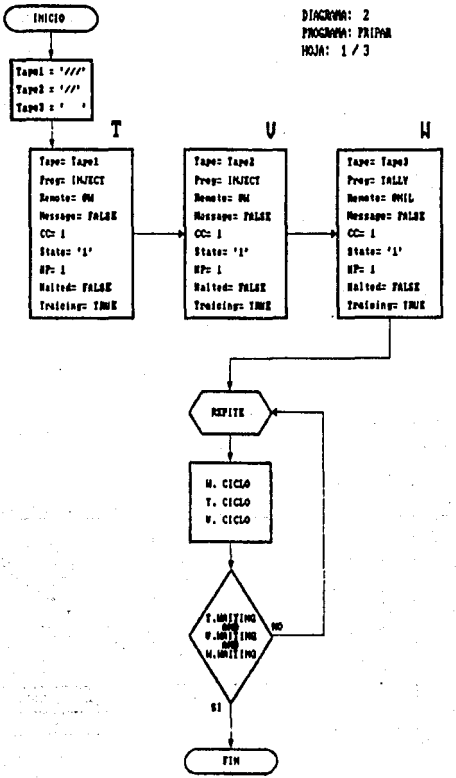
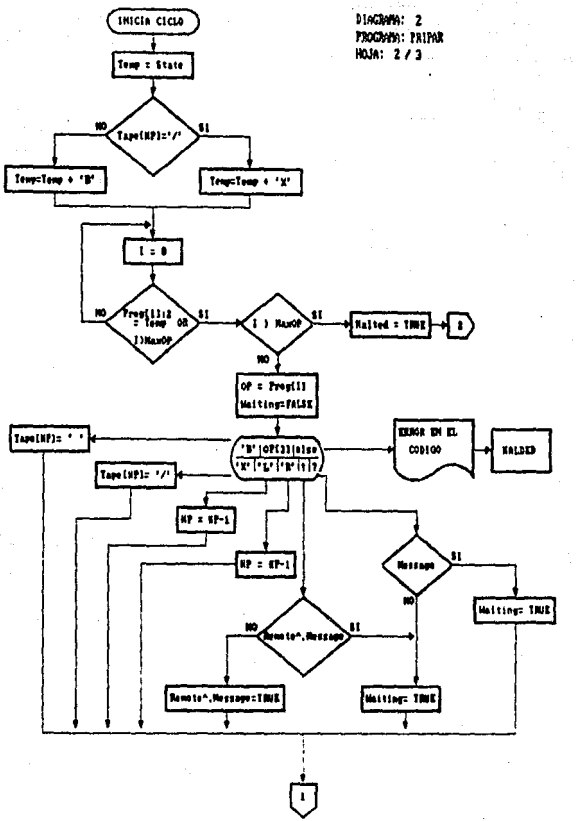


DIAGRAMA: 1
PROGRAMA: TURING
HOJA: 2 / 2


INICIO
Tape1 = '///'
Tape2 = '//'
Tape3 = ' '
DIAGRAMA: 2
PROGRAMA: PRIPAR
HOJA: 1 / 3
T
U
W
Tape: Tape1
Prog: INJECT
Remote: ON
Message: FALSE
CC: 1
State: '1'
NP: 1
Halted: FALSE
Training: TRUE
Tape: Tape2
Prog: INJECT
Remote: ON
Message: FALSE
CC: 1
State: '1'
NP: 1
Halted: FALSE
Training: TRUE
Tape: Tape3
Prog: TALLY
Remote: ONIL
Message: FALSE
CC: 1
State: '1'
NP: 1
Halted: FALSE
Training: TRUE
REPITE
M. CICLO
T. CICLO
W. CICLO
T.WAITING
AND
W.WAITING
AND
M.WAITING
NO
SI
FIN


INICIA CICLO
DIAGRAMA: 2
PROGRAMA: PRIPAR
HOJA: 2 / 3
Temp = State
Tape(HP)='/'
NO
SI
Temp=Temp + 'B'
Temp=Temp + 'X'
I = 0
Prog[I].Ir = Temp OR I)MaxOP
I) MaxOP
Halted = TRUE
A
OP = Prog[I]
Waiting=FALSE
Tape(HP)= ' '
'B'|OP[3]|else
'X'|'L'|'R'|'N'|'T'
ERROR EN EL CODIGO
HALDED
Tape(HP)= '/'
HP = HP-1
HP = HP-1
Message
Waiting= TRUE
Remote^.Message
Remote^.Message=TRUE
Waiting= TRUE
I

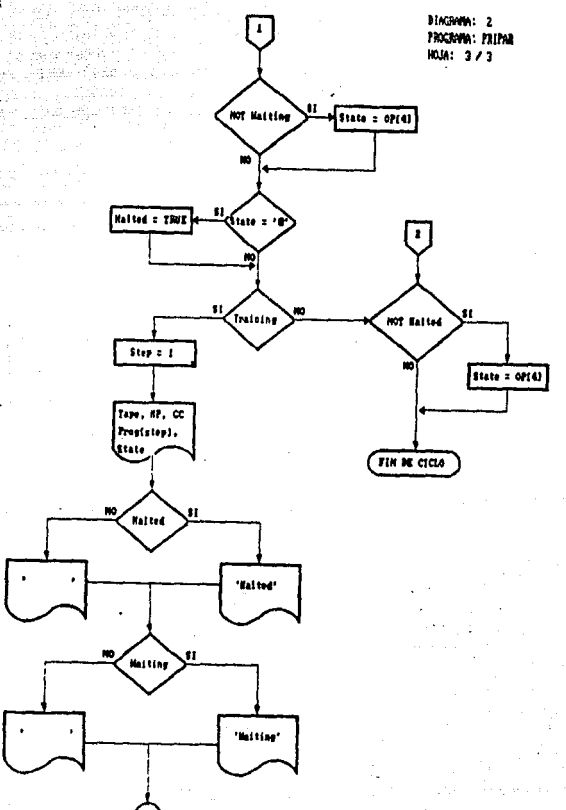


DIAGRAMA: 2
PROGRAMA: PRIPAR
HOJA: 3 / 3
1
NOT Waiting
SI
State = OP(4)
NO
Halted = TRUE
SI
State = '0'
NO
2
SI
Traising
NO
NOT Halted
SI
Step = 1
NO
State = OP(4)
Tape, AP, CC
Prog(step),
State
FIN DE CICLO
NO
Halted
SI
'Halted'
NO
Waiting
SI
'Waiting'
1

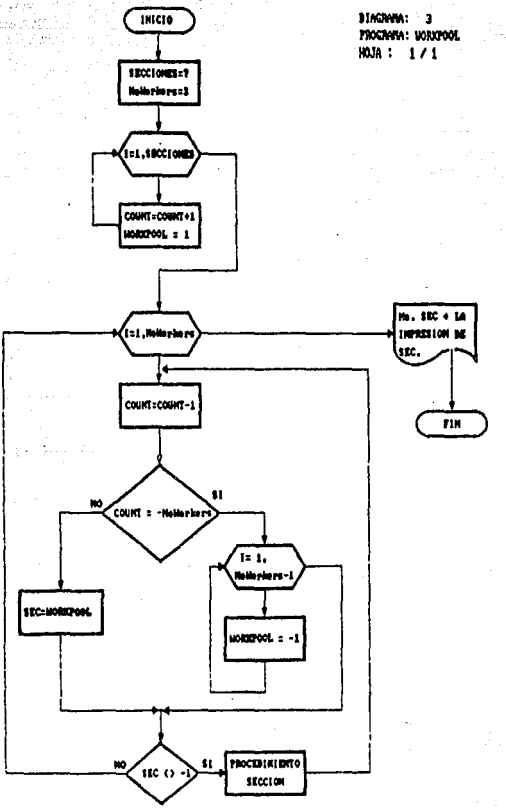


DIAGRAMA: 3
PROGRAMA: WORKPOOL
HOJA : 1 / 1
INICIO
SECCIONES=?
Nohilos=?
I:1,SECCIONES
COUNT=COUNT+1
WORKPOOL = 1
I:1,Nohilos
No. SEC = LA IMPRESION DE SEC.
FIN
COUNT=COUNT-1
NO
COUNT = -Nohilos
SI
I= 1, Nohilos-1
SEC=WORKPOOL
WORKPOOL = -1
NO
SEC () -1
SI
PROCEDIMIENTO SECCION

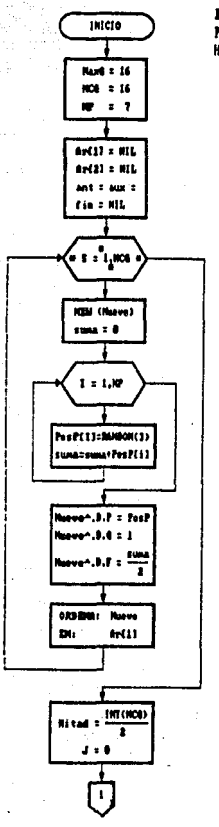


DIAGRAMA: 4
PROCEDIMIENTO: SECCION
HOJA: 1 / 3

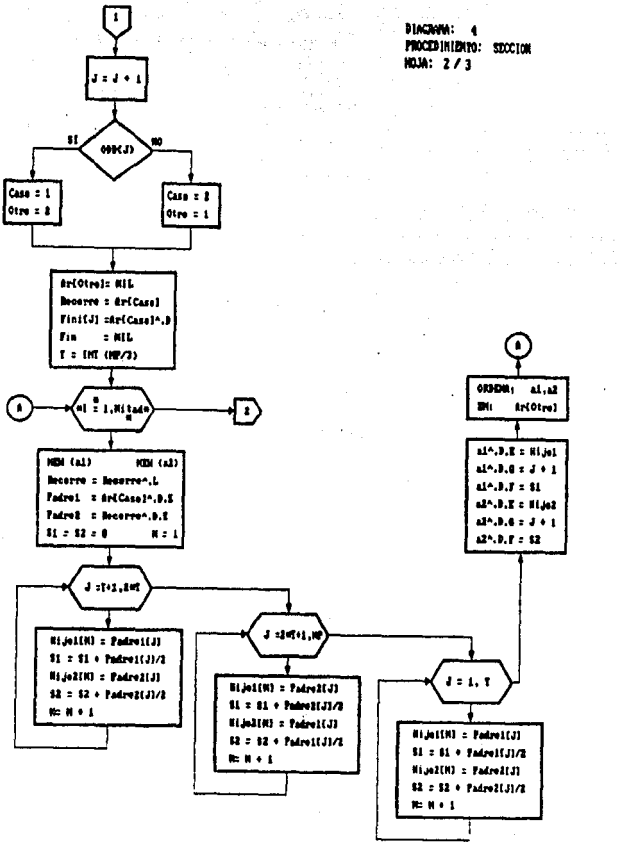


DIAGRAMA: 4
PROCEDIMIENTO: SECCION
HOJA: 2 / 3
1
J = J + 1
SI
000(J)
NO
Case = 1
Otro = 2
Case = 2
Otro = 1
Ar[Otro]: NIL
Recorre = Ar[Case]
Fini[J] =Ar[Case]^.D
Fin : NIL
T = (MT (MP/3)
A
a1 = 1,Hijado
2
MEM (a1)
MEM (a2)
Recorre = Recorre^.L
Padre1 = Ar[Case]^.D.E
Padre2 = Recorre^.D.E
S1 = S2 = 0
N = 1
J =T+1,2MT
Hijo1[N] = Padre1[J]
S1 = S1 + Padre1[J]/2
Hijo2[N] = Padre2[J]
S2 = S2 + Padre2[J]/2
N= N + 1
J =2MT+1,MP
Hijo1[N] = Padre2[J]
S1 = S1 + Padre2[J]/2
Hijo2[N] = Padre1[J]
S2 = S2 + Padre1[J]/2
N= N + 1
J = 1, T
Hijo1[N] = Padre1[J]
S1 = S1 + Padre1[J]/2
Hijo2[N] = Padre2[J]
S2 = S2 + Padre2[J]/2
N= N + 1
A
ORDENA: a1,a2
EN: Ar[Otro]
a1^.D.E = Hijo1
a1^.D.S = J + 1
a1^.D.F = S1
a2^.D.E = Hijo2
a2^.D.S = J + 1
a2^.D.F = S2

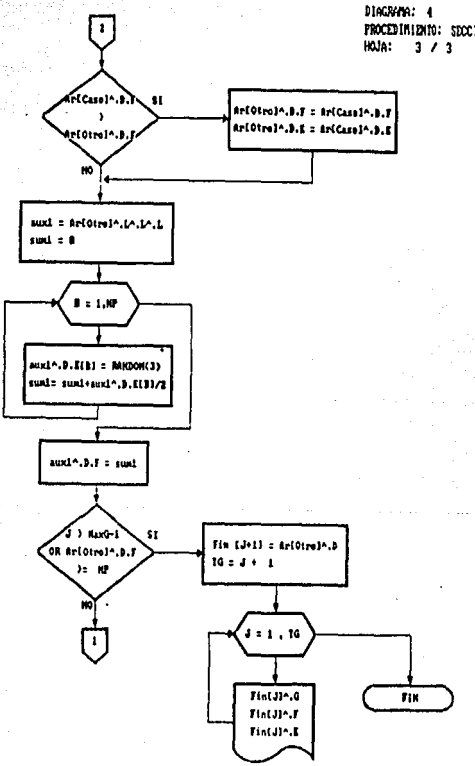


DIAGRAMA: 4
PROCEDIMIENTO: SECCION
HOJA: 3 / 3
1
Ar[Caso]^.B.F
)
Ar[Otro]^.B.F
SI
Ar[Otro]^.B.F = Ar[Caso]^.B.F
Ar[Otro]^.B.E = Ar[Caso]^.B.E
NO
auxi = Ar[Otro]^.L^.L^.L
sumi = 0
B = 1,NP
auxi^.B.E[B] = RANDOM(3)
sumi= sumi+auxi^.B.E[B]/2
auxi^.B.F = sumi
J) MaxG-1
OR Ar[Otro]^.B.F
>= MP
SI
Fin[J+1] = Ar[Otro]^.B
TG = J + 1
NO
1
J = 1 , TG
Fin[J]^.G
Fin[J]^.F
Fin[J]^.E
FIN