Aplicación de Procesamiento en Paralelo MPI a la Solución en Estado Estacionario Periódico en el Dominio del Tiempo de Sistemas Eléctricos No Lineales

TESIS

Que para obtener el grado de:

MAESTRO EN INGENIERIA ELECTRICA

Presenta:

ROSALIA MORA JUAREZ

Director de Tesis:

DR. J. AURELIO MEDINA RIOS

Co-Director de Tesis

DR. ANTONIO RAMOS PAZ

UNIVERSIDAD MICHOACANA DE SAN NICOLAS DE HIDALGO

Morelia, Michoacán. Noviembre 2008

APLICACIÓN DE PROCESAMIENTO EN PARALELO MPI A LA SOLUCIÓN EN ESTADO ESTACIONARIO PERIÓDICO EN EL DOMINIO DEL TIEMPO DE SISTEMAS ELÉCTRICOS NO LINEALES

Los miembros del Jurado de Examen de Grado aprueban
la **Tesis de Maestría en Ingeniería Eléctrica** de Rosalía Mora Juárez

Dr. Norberto García Barriga
Presidente del Jurado

Dr. J. Aurelio Medina Ríos
Director de Tesis

Dr. Antonio Ramos Paz
Vocal

Dr. Juan Manuel García García
Vocal

Dr. Domingo Torres Lucio
Examinador Externo
Instituto Tecnológico de Morelia

Dr. J. Aurelio Medina Rios
Jefe de la División de Estudios de Posgrado en Ingeniería Eléctrica

UNIVERSIDAD MICHOACANA DE SAN NICOLAS DE HIDALGO
Septiembre del 2008

**“… disipó el orgullo de los soberbios,
trastornando sus designios … ”**

Gracias a Dios y a Sebastián Antonio,
por recordarme el incalculable valor de la vida.
Un solo latido de tu corazón, hijito, es más importante que cualquier título.
Aprenderé de ti a ser fuerte y a luchar día a día,
y venceré la adversidad como lo hiciste tu.
Que esta Tesis sea mi primer paso…
y es para ti.

Mi más sincero agradecimiento también

A mi Asesor de Tesis, Dr. J. Aurelio Medina Ríos, por permitirme realizar esta Tesis bajo su dirección. Gracias por su paciencia y confianza, porque no obstante lo sinuoso del camino siempre me manifestó su apoyo, que hoy se ve reflejado en la culminación de esta Tesis

Al Dr. Antonio Ramos Paz, por su ayuda en todo lo referente a la plataforma paralela PVM; por su apoyo en el manejo del cluster; por sus enseñanzas y por su amistad.

A los Profesores que participaron como revisores de mi Tesis, Dr. Norberto García Barriga, Dr. Domingo Torres Lucio y Dr. Juan Manuel García García. Les agradezco sus observaciones y sugerencias porque con ellas he logrado enriquecer mi Tesis

A mis profesores en la Maestría, quienes compartieron generosamente sus conocimientos conmigo. Todo lo aprendido forma parte de la mejor riqueza que poseo.

A la Facultad de Ingeniería Eléctrica, en especial al Ing. Gilberto I. López, al Dr. Leonardo Romero, al M.I. Samuel Pérez Aguilar, y a mis compañeros del Laboratorio de Computación M.I. Moisés García Villanueva e Ing. Rodrigo Guzmán Maldonado. Gracias a todos por su apoyo y solidaridad, y por las facilidades que me otorgaron para poder concluir esta Tesis.

Al Dr. Jesús Alberto Verduzco Ramírez del Laboratorio ID-IMAG del Institute Nationale Politechnique de Grenoble Francia, por su valiosa ayuda en lo referente a la plataforma paralela MPI.

A mi madre, Profra. Bertha Juárez Cortés. No me alcanzará la vida para agradecerte tu apoyo incondicional. Soy lo que soy gracias ti mamí, a tu amor, tu tiempo, tu ejemplo y tu esfuerzo.

A mi esposo M.A. Nelson Martínez Cedeño. Por su amor y apoyo, por creer en mi. Gracias por tu comprensión y paciencia, porque siempre me das ánimo en los momentos difíciles y no me dejas caer. Nelsuk, tiamunzin!.

Mi agradecimiento muy especial y con mucho cariño para la Sra. Yolanda Pacheco, por su asesoría y ayuda en los trámites administrativos relacionados con esta Tesis, pero más por su amistad y sus consejos.

A mis compañeros y amigos, M.I Constantino Sumila, Ing. Carlos Martínez, L.I. Estela Arriaga. y M.I. José Luis Guillen, por ser parte de mi historia y compartir conmigo alegrías, tristezas, desvelos y éxitos. Al *niño invisible*, por su ayuda en la configuración del cluster Beowulf y … por todo lo demás.

A mis queridas amigas y hermanas M.B. Jaquelina Calderón y Arq. Mayleth Salas, por brindarme su cariño, apoyo y amistad en las buenas y en las malas.

RESUMEN

Esta Tesis describe la aplicación de procesamiento en paralelo MPI al algoritmo de Diferenciación Numérica para obtener la solución en estado estacionario periódico de sistemas eléctricos con componentes no lineales. En este trabajo de Tesis se revisan de manera general los conceptos básicos del procesamiento en paralelo, los factores de desempeño y las recomendaciones para el diseño de algoritmos paralelos.

La programación en la plataforma paralela MPI corresponde al ambiente LAM-MPI. Todos los códigos que se desarrollaron están escritos en lenguaje C++ y fueron implementados con programación orientada a objetos. El programa paralelo del algoritmo de Diferenciación Numérica propuesto en este trabajo de Tesis se ejecutó en un cluster tipo Beowulf.

Se obtuvo la solución de tres Casos de Estudio que representan redes eléctricas de distinto tamaño. Para cada Caso de Estudio se calculó el Speed-up y la eficiencia relativa entre los tiempos de cómputo, empleando diferente número de elementos de proceso. En todos los Casos de Estudio se hizo la comparación de eficiencia obtenida con MPI y PVM.

Palabras claves: Procesamiento en paralelo, LAM-MPI, Diferenciación Numérica, cluster Beowulf, speed-up, eficiencia relativa.

ABSTRACT

This Tesis describes a MPI parallel processing application to Numeric Differentiation to obtain the periodic steady state solution of power systems containing non-linear elements. This work revises, in general, the basic concepts of parallel processing and recommendations for parallel algorithms design.

The MPI parallel methodology programming corresponds to LAM-MPI behavior. The codes development are made in C++ language and implemented with object oriented programming techniques. The parallel program of the Numeric Differentiation algorithm proposed in this Tesis was executed in a Beowulf cluster.

In this work was computed the solution for three Study Cases that represent electric networks. For each Study Case was computed the Speed up and Relative Efficiency and obtained the computing times with different number of processors. For all Study Cases was compared the efficiency obtained with MPI and PVM.

Keywords: Parallel Processing, LAM-MPI Numeric Differentiation, Beowulf Cluster, Speed-up, Relative Efficiency.

INDICE

CAPITULO 4
PARALELIZACIÓN DE ALGORITMOS PARA LA SOLUCIÓN DE SISTEMAS ELECTRICOS

CAPITULO 5
CASOS DE ESTUDIO

LISTA DE SIMBOLOS

A	Matriz real, simétrica y no singular.
Φ	Matriz de Identificación.
ξ	Ypsilón.
$\sum$	Sumatoria.
$\int$	Integral.
I	Matriz identidad.
h	Paso de integración.
∞	Infinito.
10^{-10}	Expresión matemática equivalente a 1×10^{-10}.
x^{∞}	Vector de variables de estado en el Ciclo Límite.
Δ	Incremento.
tag	Etiqueta.
$O(f)$	Orden de complejidad.
shell	Interprete de comandos de Unix.
>>	Operador concatenación.
k_n	Constante del método de Runge-Kutta.
V	Voltaje.
$J(t)$	Jacobiano.
C	Matriz que tiene los valores de las variables en el Ciclo Límite.
ρ	Plano Poincaré

AD	Aproximación Directa.
C++	Lenguaje de programación C más más.
CA	Corriente alterna.
CD	Corriente directa.
CP	Contador de programa.
CPU	Unidad de procesamiento central.
DN	Método de Diferenciación Numérica.
EDA´s	Ecuaciones Algebraicas y Diferenciales.
EDO	Ecuación Diferencial Ordinaria.
EEP	Estado Estacionario Periódico.
EP	Elemento de Proceso.
FB	Método de Fuerza Bruta.
FLOPs/seg	Operaciones de punto flotante por segundo.
ID	Número que identifica a un nodo en el universo LAM.
LCK	Ley de corrientes de Kirchhoff.
LU	Producto de matriz triangular superior U y matriz triangular inferior L.
LVK	Ley de voltajes de Kirchhoff.
MB	Megabytes.
Mbps	Millones de bips por segundo.

MD	Matriz de Diferencias.
ME	Matriz Exponencial.
MIPs/Seg	Millones de instrucciones por segundo.
MPI	Interface para Paso de Mensajes (*Message Passing Interface*).
MPP	Procesamiento Masivamente Paralelo.
PC	Computadora personal.
POO	Programación orientada a objetos.
PVM	Máquina Virtual Paralela (*Parallel Virtual Machine*).
RAM	Memoria de acceso aleatorio.
RK4	Runge-Kutta 4º orden.
RPM	Sistema de administración de paquetes en ambiente GNU-Linux.
SMP	Multiprocesamiento Simétrico.
TCR	Reactor conmutado por tiristor.
TCSC	Tiristor controlado capacitancia serie.
ANN	*Artificial Neural Network*. Red neuronal artificial.
ANSI	*American National Standar Institute*. Instituto Internacional Estadounidense de Estándares.
API	*Application Programming Interfase*. Interfaz de Programación de Aplicaciones. Conjunto de funciones y procedimientos que ofrece cierta librería para ser utilizado por otro software como una capa de abstracción [Wikipedia].
ASCII	*American Standard Code for Information Interchange*.
CC-UMA	*Cache-Coherent Uniform Memory Access.*
COMA	*Cache Only Memory Access.*
FACTS	*Flexible A.C. Transmission Systems.*
IEEE	*Institute of Electrical and Electronics Engineers.*
LAM	*Local Area Multicomputer*.
MIMD	*Multiple Instruction stream, Multiple Data stream.*
MISD	*Multiple Instruction stream, Single Data stream.*
MT	*Multithreading.*
NFS	*Network File System.*
NUMA	*Non-Uniform Memory Access.*
POSIX	*Portable Operating System Interface for uniX*.
RISC	*Reduced Instruction Set Computer*.
SIMD	*Single Instruction stream, Multiple Data stream.*
SISD	*Single Instruction stream, Single Data stream.*
SSH	*Secure SHell.*
UMA	*Uniform Memory Access.*
Buffer	Espacio de memoria en el que se almacenan datos de manera temporal.
CNSCC	Centro Nacional de Supercómputo de la Red de Centros Públicos CONACyT.

Demonio	Es un programa que se ejecuta a nivel del sistema operativo y permanece activo sin la intervención directa del usuario. En ingles se denomina *daemon*.
DEP-FIE	División de Estudios de Posgrado de la Facultad de Ingeniería Eléctrica.
DSA	Llave pública para autentificación en sistemas remotos bajo el protocolo rd.
FIFO	*First In, First Out*. Indica una estructura tipo cola, donde el primer elemento en entrar es el primero en salir.
fork()	Bifurcación. Creación de un proceso distinto del principal, pero que tiene el mismo código.
GNU	Acrónimo recursivo que significa *GNU no es Unix*. Organización de software libre compatible con Unix.
Host	Es una máquina que forma parte de una red. También se le denomina anfitrión.
IPICyT	Instituto Potosino de Investigación Científica y Tecnológica.
IPN	Instituto Politécnico Nacional.
MAC	Macintosh. Computadora personal desarrollada y comercializada por la compañía Apple Inc.
RSA	Llave pública para autentificación en sistemas remotos bajo el protocolo rs.
Speed up	Factor de mejora de rendimiento como ganancia de velocidad.
script	Guión o conjunto de instrucciones.
TCP/IP	Conjunto de protocolos de internet. TCP = Protocolo de control de transmisión. IP = Protocolo de internet.
UAM	Universidad Autónoma Metropolitana
UDP	*User Datagram Protocol*. Protocolo de datos de usuario. Envía datagramas IP sin tener que establecer una conexión.
UNAM	Universidad Nacional Autónoma de México.
X11	Protocolo versión 11 del la interfaz grafica (ventanas) desarrollado para Unix.
XDR	*eXternal Data Representation.* Protocolo de presentación de datos. Permite la transferencia de datos entre diferentes arquitecturas y sistemas operativos.

LISTA DE FIGURAS

LISTA DE TABLAS

INDICE DE ECUACIONES

CAPITULO 1

INTRODUCCION

1.1. REVISION DEL ESTADO DEL ARTE

Tradicionalmente, la simulación numérica de sistemas complejos como dinámica de fluidos, clima, circuitos eléctricos y electrónicos, reacciones químicas, modelos ambientales, estudios de sistemas en tiempo real y procesos de manufacturación, han requerido de computadoras cada vez más potentes. Inclusive, hoy en día estas máquinas están siendo demandadas para aplicaciones comerciales que necesitan procesar grandes cantidades de datos como la realidad virtual, vídeo conferencias, bases de datos paralelas y diagnóstico médico asistido por computadoras por mencionar solo algunas [Foster 1995].

La eficiencia de una computadora depende directamente del tiempo requerido por su procesador para ejecutar una instrucción básica y del número de instrucciones básicas que pueden ser ejecutadas concurrentemente. Mejorar la eficiencia de un procesador, desde el punto de vista del hardware, implica reducir el tiempo de procesamiento, y esto sólo ha sido posible gracias a los avances tecnológicos, como la integración de tecnología VLSI (*Very Large Scaled of Integration*) [Jin 1994]. Sin embargo, tener la posibilidad de realizar estudios en tiempo real y aumentar el tamaño de los problemas ha dado origen a una solución alternativa: emplear varios procesadores que trabajen *en paralelo*.

El *procesamiento en paralelo* es una forma de procesamiento de la información en la cual dos o más procesadores, juntos y mediante algún sistema de comunicación interprocesador, *co-operan* para la solución de un problema. En otras palabras, el procesamiento en paralelo se basa en una arquitectura de procesadores que realizan tareas simultáneas las cuales resuelven un problema de grandes dimensiones [Jájá 1992].

1.1.1. Procesamiento en paralelo aplicado a sistemas eléctricos de potencia

La solución a sistemas eléctricos ordinariamente requiere de un conjunto grande de ecuaciones lo cual genera matrices de datos de gran tamaño. Un algoritmo secuencial eficiente explotará el alto grado de dispersidad de la conectividad de la red, para así ganar velocidad de procesamiento y llegar a la solución. Sin embargo, al elevarse el número de cálculos, se incrementa considerablemente el tiempo de procesamiento, razón por la cual se ha pensado en aplicar procesamiento paralelo.

En sistemas eléctricos de potencia, gran parte de las aplicaciones de procesamiento en paralelo que se han obtenido se enfocan al análisis y solución de flujos de potencia [Mariños *et al*. 1994].

En el análisis de los sistemas eléctricos de potencia es necesaria la solución de grandes sistemas de ecuaciones lineales, como es el caso de los flujos de potencia, el análisis de fallas, el análisis armónico, etc. En [Oyama *et al.* 1990] se propuso la aplicación de procesamiento en paralelo a un algoritmo para la solución de ecuaciones lineales.

El desarrollo de procesamiento en paralelo para flujos de potencia ha sido una tarea ardua debido a que el patrón de dispersión de las matrices que se generan suele ser irregular. Sin embargo, [Kaiser *et al*. 1989] ha logrado resultados satisfactorios en el análisis de contingencias de una red de 1000 nodos y 1500 ramas, con tiempos de respuesta del orden de los milisegundos, considerando comunicación a distancia vía Ethernet; en [Feng and Fleck 2002] se reporta un algoritmo Newton para la solución de flujos de potencia ejecutado en un cluster GNU/Linux de estaciones de trabajo y basado en el paradigma de paso de mensajes, obteniendo la solución paralela de sistemas lineales dispersos con el algoritmo GMRES (*Generalised minimal residual*).

Otras aplicaciones de procesamiento en paralelo son el estudio del comportamiento armónico en redes eléctricas [Mariños *et al*. 1994] y así como al algoritmo de detección de fallas en tiempo real para manejadores de motores de corriente directa de [Stavrakakis *et. al* 1990].

La respuesta la a frecuencia de redes eléctricas se obtuvo aplicando procesamiento en paralelo empleando PVM ([A.Medina, A. Ramos-Paz 2005]) y MT ([A. Medina *et al*. 2004]).

El análisis para la solución en estado estacionario periódico de redes eléctricas no lineales demanda cálculos de matrices muy grandes y la mayoría de las veces con alto grado de dispersidad. Para este análisis, se han propuesto soluciones implementadas en diferentes plataformas paralelas: Multithreading (MT) [Garcia *et al*. 2001] [García y Acha 2004], la Máquina Virtual Paralela (PVM) [Medina, Ramos-Paz y Fuerte-Esquivel 2003], y la Interface para Paso de Mensajes (MPI) [García 2005].

Una aportación importante en cuanto a procesamiento en paralelo aplicado a sistemas eléctricos de potencia es el trabajo de [García 2005], donde utiliza la plataforma de procesamiento paralelo MPI empleando 256 procesadores para el análisis de un sistema eléctrico trifásico de 118 nodos que incluye dispositivos FACTS.

1.1.2. Ecuaciones algebraicas y diferenciales en sistemas eléctricos de potencia

La experiencia indica que el paralelismo inherente no es muy obvio en la estructura matemática de los problemas a resolver. Por lo tanto, para un problema en particular, debe hacerse una formulación paralela (o cercana a lo paralelo), y la forma adecuada es expresarse como un algoritmo paralelo. El algoritmo tiene que ser implementado en una máquina de arquitectura paralela [Jin 1994], y se deben tomar en cuenta los parámetros de eficiencia computacional que determinan el desempeño del algoritmo paralelo.

Los algoritmos de problemas de sistemas eléctricos presentan algunas operaciones que pueden ser paralelizadas; se ha observado que la mayoría de éstos expresan la solución de ecuaciones a través de la forma de álgebra lineal:

$$Ax = b \tag{1.1}$$

donde *A* es una matriz real, simétrica y no singular, con alto grado de dispersidad. *x* y *b* pueden ser o no dispersas. *x* es el vector solución.

Existe un gran número de algoritmos directos e indirectos para resolver (1.1). Un método eficaz de solución en forma secuencial es la factorización por sustitución atrás/adelante definida como:

$$LPU = A \tag{1.2}$$
$$Ly = b \tag{1.3}$$
$$PUx = y \tag{1.4}$$

donde *L* es matriz triangular inferior, *U* es matriz triangular superior, *P* es una matriz permutación (las columnas de *P* son *m* vectores unitarios, en algún orden de permutación) [Ascher *et al.* 1991].

Varias contribuciones se han hecho referentes al algoritmo de la factorización triangular paralela en [Abur 1988] y [Alvarado *et al.* 1990], donde los algoritmos básicamente retoman la forma secuencial del problema de factorización/sustitución y explotan el paralelismo a través del reordenamiento y particionamiento de la matriz *A*. Este método ha probado ser efectivo y se han obtenido buenos resultados teóricos.

De las primeras aportaciones en el desarrollo de software para máquinas paralelas se encuentran las de [Lee *et al*. 1989] y [Lau *et al.* 1990]. Aquí se reportan resultados de factorización/sustitución en paralelo con ganancia máxima en velocidad en el orden de 2.

La posibilidad de explotar las capacidades de las máquinas vectoriales se reporta en [Gómez y Bentacourt 1990]. Aquí se produjeron resultados experimentales para resolver (1.2) en una computadora de arquitectura vectorial.

Los problemas de estabilidad transitoria están definidos por un conjunto de ecuaciones algebraicas y diferenciales no lineales (EDA´s) de la forma:

$$\dot{x} = f(x,t) \tag{1.5}$$
$$o = g(x,y) \tag{1.6}$$

donde (1.5) describe la dinámica de los elementos y (1.6) el comportamiento estático de la red.

Los algoritmos secuenciales de la solución de (1.5) fueron desarrollados hace varias décadas y usan dos propuestas básicas: [Chua y Ushida 1981] y [Hornbeck 1975]. La primera propuesta se refiere a particionar (1.5) y resolver por métodos de integración tradicionales, como Runge-Kutta, aplicando pasos de integración en el tiempo y (1.6) se resuelve por separado. La segunda propuesta es discretizar (1.5) por el método de la integración trapezoidal, y

posteriormente se resuelve junto con (1.6) para cada incremento o paso en el tiempo usando el método Newton-Raphson.

Se observó que en los problemas de estabilidad transitoria, métodos como Runge-Kutta requieren pasos de integración muy pequeños, y por lo tanto los tiempos de cálculo suelen se bastante grandes, además de que existe una alta dependencia de sus variables k_n.

Las propuestas de [Chua y Ushida 1981] y [Hornbeck 1975] pueden ser programadas en computadoras paralelas, pero la descomposición del problema presenta las siguientes diferencias:

- La descomposición de las variables de sistema en grupos conocida como *paralelización en espacio*. La paralelización en espacio más obvia es la descomposición de (1.5) en un conjunto de ecuaciones, que se resolverán entre varias máquinas. Cada máquina resuelve ecuaciones por separado, la interconexión será proporcionada por (1.6) [Hatcher *et al.* 1997].
- La *paralelización en el tiempo*. Se logra mientras varios pasos o tiempos de integración puedan ser resueltos simultáneamente. La primera sugerencia para la paralización en tiempo fue hecha en [Alvarado 1979], formando las ecuaciones de Newton para cada paso de integración y resolviéndolas simultáneamente.

La versión discreta de la (1.5) junto con (1.6) se describe en [LaScala *et al.* 1989]. Aquí la ecuación es descompuesta en sus variables y luego resueltas simultáneamente para todos los pasos. Este método proporciona la paralelización máxima posible en el espacio y en el tiempo.

Otros esquemas para obtener la solución de estabilidad y transitorios usando métodos Newton se plantean en [LaScala *et al.* 1990], [LaScala y Sbrizzai 1990] y [Chai *et al.* 1991].

La implementación de algoritmos para la solución de sistemas eléctricos de potencia usando procesamiento paralelo ha avanzado en los últimos años debido a la cada vez mayor disponibilidad de arquitecturas paralelas. A continuación se resumen algunos reportes interesantes:

- [Tulukdar y Cardozo 1988] reportaron el desarrollo de software para el control y sincronización de los elementos que intervienen en el procesamiento paralelo distribuido (diferentes tipos de procesadores que ejecutan diferentes programas, los cuales pueden estar escritos incluso en diferentes lenguajes y con velocidad de comunicación variable.
- [Pottle 1988] usa el método EMTP para ejecutar la regla trapezoidal aplicada a las ecuaciones de una red de transmisión mediante un arreglo de transputers, uno por bus. Se logró simular el volumen balanceado del sistema, con un paso de integración de tamaño menor a 80 microsegundos, considerándolo como procesamiento en tiempo real de operación del sistema.
- En máquinas con memoria compartida, [Lewis y Berg 1998] lograron generar procesamiento concurrente al separar procesos que se ejecutaban de manera simultánea. [García et al. 2001] y [Ramos 2002] usan MT en una computadora con dos procesadores, respectivamente, para la solución de sistemas eléctricos en el dominio del tiempo.

- [Ramos 2002] utilizó la PVM para realizar operaciones matriciales en metodologías para la solución en estado estacionario periódico de redes eléctricas.
- En [García y Acha 2004] y [García 2005] se aplican las técnicas de acercamiento rápido de las variables de estado al Ciclo Límite en el análisis de redes eléctricas de gran escala con componentes no lineales y variantes en el tiempo.

1.1.3. Análisis de sistemas eléctricos en el dominio del tiempo

Un sistema eléctrico que opere en condiciones ideales es aquel que está completamente balanceado, que opera a una frecuencia única y constante, y cuyas formas de onda de voltaje y corriente en el sistema son senoidales, de amplitud constante. Se dice entonces que la calidad de energía obtenida es perfecta [Medina 2001].

En la práctica, la condición ideal no se manifiesta debido a que todos los componentes de una red eléctrica presentan un efecto indeseable conocido como *distorsión armónica.* Esta tiene el efecto indeseable de distorsionar la característica senoidal de la forma de onda del voltaje y corriente, ocasionando pérdidas en el sistema, interferencias en el control de la red y reducción de la vida útil de componentes, entre otros efectos adversos. Lo anterior ha hecho necesario que se adopten estándares y regulaciones para limitar la distorsión armónica como son [IEEE519 2002].

Los elementos que aportan mayor distorsión armónica son cargas y componentes no lineales y variantes en el tiempo como los convertidores de potencia, dispositivos FACTS, componentes de núcleo magnético que presentan el efecto no lineal de saturación, hornos de arco, etc. [Arrillaga *et al.* 1985]. Si se logra la detección y predicción de armónicos en sistemas eléctricos se puede realizar un diagnostico y evaluación adecuados de la calidad de la energía de la red.

El estudio del comportamiento armónico puede hacerse en tiempo real, mediante monitoreo directo en la red eléctrica, o bien mediante modelos matemáticos que describen el comportamiento periódico de los componente lineales y no lineales. La Tesis que aquí se presenta se fundamenta en modelos matemáticos para hacer simulación digital de sistemas eléctricos, con la particularidad de que los cálculos digitales que se hacen bajo plataformas de procesamiento en paralelo.

La ventaja de analizar una red eléctrica en el dominio del tiempo es que se puede encontrar la solución implícitamente en las componentes armónicas asociadas con las formas de onda de las variables que representan el comportamiento de los elementos de la red.

La desventaja del análisis en el dominio del tiempo es que el esfuerzo computacional para llegar al estado periódico estacionario se incrementará excesivamente si existen en la red elementos con pobre o nulo amortiguamiento, tales como los transformadores de potencia, la máquina síncrona y los dispositivos FACTS [Chua y Ushida 1981].

Otra desventaja del análisis en el dominio del tiempo es la formulación de las ecuaciones diferenciales que describen un sistema. Recientemente, [Ramos 2007] ha propuesto una herramienta computacional que permite la construcción del conjunto de ecuaciones diferenciales ordinarias que modelan redes eléctricas de gran escala.

El estudio del comportamiento periódico de la red en el dominio del tiempo se ha realizado mediante diversos métodos; [Chua y Ushida 1981] los clasifica como:

- *Métodos* de *Fuerza Bruta* (FB): realizan la integración del conjunto de EDO's que modelan la dinámica del sistema hasta que se alcanza el EEP. En [Chua y Ushida 1981] y en [Hornbeck 1975] se hace un análisis a detalle de estos métodos.
- *Métodos de Disparo***:** consisten en técnicas utilizadas para acelerar el cálculo del EEP en sistemas eléctricos de potencia. El objetivo de estos métodos es encontrar una condición inicial del vector $x(0)$ tal que cuando se integre el sistema de ecuaciones $\dot{x} = f(x,t)$, sobre un periodo completo de tiempo *T*, a partir de la condición inicial, se obtenga $x(T) = x(0)$. Algunos métodos de disparo se proponen en [Aprille y Trick 1972].
- *Métodos de perturbación*: se basan en un proceso iterativo, el cual comienza con la determinación de una solución inicial. Esta solución inicial se obtiene mediante la linearización de la ecuación que describe la dinámica del sistema [Chua y Ushida 1981].
- *Métodos de Balance Armónico*: representan a cada variable de estado por medio de una serie de Fourier que satisface el requisito de periodicidad [Wylie 1951]. Posteriormente se aplica un algoritmo de optimización para ajustar los coeficientes de las Series de Fourier de manera que las ecuaciones del sistema se satisfagan con el error mínimo.

Otra clasificación de métodos para el análisis en el dominio del tiempo se hace en [Kunder *et al.* 1990] clasificándolos como: Métodos en el Dominio del Tiempo, Métodos Híbridos Frecuencia-Tiempo y Métodos de Balance Armónico.

En la presente Tesis se utiliza como Método de FB el método de Runge-Kutta de 4º. Orden, y el Método de Disparo propuesto por [Semlyen y Medina 1995] denominado *Técnica Aceleración de la Convergencia al Ciclo Límite.* Esta técnica determina la solución periódica en estado estacionario de sistemas no lineales y variantes en el tiempo, y se fundamenta en la aceleración de la convergencia de las variables de estado al Ciclo Limite, el uso de mapas Poincaré y la aplicación de métodos iterativos de tipo Newton. Dicha técnica se ha usado con éxito en el modelado en el dominio del tiempo de componentes tales como la máquina síncrona [Rodríguez y Medina 2004] y [Medina *et al*. 2004b], transformador de potencia [García *et al*. 2000], TCSC (*Thyristor Controller Series Compensator*) [Medina, Ramos-Paz y Fuerte-Esquivel 2003], SVS's (*Static Var System*) [García y Medina 2003], TCR´s [Medina y García 2004], y STATCOMS (*Static Synchronous Compensator*) y DVR (*Dynamic Voltaje Restorer*) [García 2006].

[Semlyen y Medina 1995] proponen tres Métodos Newton de extrapolación al Ciclo Límite basados en el concepto del Plano de Poincaré: método de Matriz Exponencial (ME), método de Aproximación Directa (AD) y método de Diferenciación Numérica (DN). En [Medina y Ramos-Paz 2005] se hace un análisis comparativo las técnicas Newton de DN, AD y Matriz de Diferencias (MD) aplicadas al análisis del estado periódico estacionario de redes eléctricas con componentes no lineales y variantes en el tiempo. En [Ramos 2007] se reporta la aplicación de procesamiento en paralelo a los métodos de DN, ME, AD y DN respectivamente, empleando la plataforma paralela PVM.

Se ha paralelizado parte del algoritmo de la Aceleración de la Convergencia al Ciclo Límite, empleando como plataforma de procesamiento en paralelo MT [García 2001], PVM y MT [Ramos 2002], y MPI [García 2005], lográndose una mejora considerable en el tiempo de procesamiento con respecto al cálculo secuencial.

La presente Tesis pretende lograr la paralelización del algoritmo de Diferenciación Numérica (DN) por medio de la plataforma de procesamiento paralelo MPI para encontrar la solución en estado estacionario periódico de sistemas eléctricos no lineales. En esta Tesis se explora una variante de MPI que utiliza un *cluster* de computadoras personales para la solución en estado estacionario periódico de los sistemas eléctricos no lineales analizados. Los resultados obtenidos se compararán con la plataforma paralela PVM.

1.2. JUSTIFICACION

En la mayoría de los trabajos citados en la revisión del estado del arte que se presenta en esta Tesis, se puede observar que el procesamiento en paralelo se implementa como alternativa para disminuir el costo computacional que implica resolver los cada vez más complejos sistemas de ecuaciones que describen el comportamiento de los sistemas eléctricos de potencia y sus componentes.

El modelado de sistemas eléctricos cada vez más complejos, la aplicación de métodos numéricos donde intervienen operaciones algebraicas con matrices dispersas de grandes dimensiones, así como el tiempo de procesamiento requerido para la solución en estado estacionario periódico de sistemas de eléctricos de gran escala, requieren de cómputo de alto desempeño. Son éstos solo tres ejemplos que justifican la búsqueda de herramientas, tanto software como hardware, que proporcionen un menor coste computacional, así como la optimización en los tiempos de cálculo, y la posibilidad de escalar los sistemas de estudio.

Si bien se han aplicado principalmente procesamiento en paralelo tales como PVM y MT en la solución de sistemas eléctricos, existen también otras alternativas para cómputo paralelo interesantes de implementar, como la plataforma paralela MPI. En una reciente contribución [García 2005] se aplica procesamiento en paralelo MPI a la solución en estado estacionario periódico de redes eléctricas de gran escala.

En esta Tesis se propone implementar computo paralelo MPI aplicado al análisis de redes eléctricas no lineales. Se propondrá la paralelización de los algoritmos de DN (FB, matriz de identificación y cálculo de matriz inversa) de [Semlyen y Medina 1995]. La justificación de la Tesis es el interés de emplear una plataforma paralela diferente a PVM y MT, de manera que se pueda obtener un comparativo de las características de operación y eficiencia de MPI con respecto a PVM.

Se busca además que las conclusiones obtenidas puedan aportar elementos de selección para investigaciones futuras, cuando de elegir entre las tres plataformas paralelas disponibles en la División de Estudios de Posgrado de la Facultad de Ingeniería Eléctrica.

En base a los trabajos que han sido presentados en el campo del análisis de redes eléctricas utilizando procesamiento en paralelo, se busca el uso de una plataforma de procesamiento en paralelo eficiente que permita la posibilidad de realizar el análisis de cualquier tipo de red eléctrica, y contar con un cluster que pueda ser utilizado con fines educativos y de investigación.

1.3. METODOLOGIA

El trabajo que se presenta en esta Tesis se desarrolló en las siguientes etapas:

1. Revisión del estado del arte. Recopilación de los estudios realizados sobre el procesamiento paralelo aplicado a sistemas eléctricos de potencia. Investigación de los elementos que se requieren para implementar procesamiento paralelo, las técnicas de programación paralela que existen y la metodología para diseñar algoritmos paralelos.

2. Implementación del método Newton de Diferenciación Numérica que utiliza la aceleración de la convergencia al Ciclo Límite. Análisis de los algoritmos de: Runge-Kutta 4º orden, de la matriz de identificación, y métodos para calcular la solución de sistemas de ecuaciones de la forma $Ax = b$.

3. Obtención de información referente a la plataforma de procesamiento en paralelo MPI.

4. Obtención de información y configuración de la arquitectura conocida como *cluster Beowulf*, que dé soporte al procesamiento paralelo que emplea el paso de mensajes.

5. Revisión de las técnicas para desarrollar algoritmos paralelos. Implementación de las estructuras lógicas recomendadas para trabajar con matrices de gran tamaño.

6. Realizar las pruebas y corrección de errores del algoritmo secuencial de DN para después aplicar técnicas de paralelización.

7. Casos de estudio. Se realiza el cómputo paralelo utilizando la plataforma MPI. Se obtiene el análisis comparativo de la solución con respecto a PVM.

1.4. OBJETIVO

- Aplicar procesamiento paralelo distribuido utilizando la plataforma paralela para paso de mensajes MPI, a la solución eficiente en estado estacionario periódico en el dominio del tiempo de sistemas eléctricos con componentes no lineales y variantes en el tiempo, implementado el algoritmo de Diferenciación Numérica en una arquitectura paralela tipo Cluster Beowulf, construida con los recursos existentes en la DEP-FIE.

1.4.1.Objetivos particulares

1. Aplicar una plataforma paralela para procesamiento paralelo distribuido a la solución en estado estacionario de sistemas eléctricos no lineales.
2. Explorar la aplicación de procesamiento en paralelo a métodos numéricos asociados con la descomposición triangular de sistemas de ecuaciones lineales y no lineales y de la solución numérica del método de Runge-Kutta de cuarto orden.
3. Comparar la eficiencia computacional de la solución en estado estacionario de sistemas eléctricos no lineales obtenida con las plataformas de procesamiento en paralelo MPI y PVM.

1.5. APORTACIONES

- Aplicar procesamiento en paralelo a los métodos de Runge-Kuta de 4º orden, matriz de identificación y matriz inversa, usando la plataforma de paso de mensajes MPI.
- Análisis comparativo de la solución secuencial y solución paralela utilizando las plataformas de paralelización MPI y PVM.
- Implementación y configuración de un cluster tipo Beowulf, que permitirá ejecutar procesamiento en paralelo MPI.

1.6. DESCRIPCION DE CAPITULOS

En el Capítulo 1 se hace una revisión de las contribuciones más significativas hechas respecto a técnicas de procesamiento en paralelo aplicado a sistemas eléctricos, teniendo especial interés en aquellas técnicas aplicadas al análisis de armónicos. Se hace también una breve descripción de los capítulos que contiene el trabajo de Tesis.

En el Capítulo 2 se describe el marco de referencia para el desarrollo de procesamiento en paralelo. Se propone un concepto de procesamiento en paralelo y se justifica la necesidad de desarrollar cómputo de alto desempeño orientado a la solución de sistemas eléctricos. Se mencionan los conceptos básicos para su implementación como son la disposición de la memoria, las características de la arquitectura de la(s) máquina(s) y su topología. También se mencionan los beneficios de la programación en paralelo así como las desventajas que presenta.

En el Capítulo 3 se describe de manera general los factores de desempeño del cómputo paralelo, así como las consideraciones para la implementación de algoritmos paralelos. Se presentan también dos técnicas mediante las cuales se puede efectuar procesamiento en paralelo por software: la Interfase para Paso de Mensajes (MPI) y la Máquina Virtual Paralela (PVM).

En el Capítulo 4 se describe de manera global el procedimiento para diseñar algoritmos paralelos. Se proponen los esquemas de paralelización de los algoritmos de la Fuerza Bruta, la matriz inversa y cálculo de la Matriz de Identificación Φ del método de DN. Se incluyen los detalles técnicos de instalación del entorno MPI y del cluster Beowulf.

En este Capítulo 5 se obtiene la solución periódica en estado estacionario de sistemas eléctricos de prueba, utilizando los algoritmos paralelos propuestos en el Capítulo 4. Se presentan y comparan los resultados obtenidos con el procesamiento secuencial y con el procesamiento paralelo con MPI. Se hace también una comparación de la plataforma MPI, en relación a PVM.

Por último, en el Capítulo 6 se aportan las conclusiones obtenidas así como las sugerencias que puedan servir para investigaciones futuras.

CAPITULO 2

MARCO DE REFERENCIA PARA DESARROLLAR PROCESAMIENTO EN PARALELO

2.1 INTRODUCCION

La mayoría de las computadoras de propósito general se fabricaron bajo el modelo de John Von Newmann (1903-1957), que se basa en sistemas con una sola unidad de procesamiento central (*CPU* por sus siglas en inglés). La arquitectura o modelo de Von Newmann propuso la CPU ejecutando una instrucción a la vez, hasta llegar al cálculo o computo final de un problema. A este esquema se le conoce como ejecución *o procesamiento secuencial* [Stallings 1998]. Sin embargo, las tareas y eventos que suceden en el mundo real, por su complejidad, requieren de grandes volúmenes de operaciones y tiempo de procesamiento. El procesamiento secuencial ha demostrado ser limitado, por lo que se plantea la necesidad de implementar una mejora del mismo. Una propuesta es construir modelos de sistemas de cómputo para realizar *procesamiento en paralelo* [Jin 1995].

El procesamiento en paralelo es una forma eficaz de procesamiento de información que favorece la explotación de los sucesos concurrentes en un proceso de computación, entendiendo por *concurrencia* el número de operaciones simultáneas que es capaz de realizarse en un intervalo de tiempo. El procesamiento en paralelo se aplica tanto a nivel de hardware como del software del sistema.

El procesamiento en paralelo puede abordarse básicamente desde cuatro niveles:

1. **Nivel de programación:** exige el desarrollo de algoritmos ejecutables en paralelo. La implementación de algoritmos paralelos depende de la asignación eficaz de recursos hardware-software para resolver un extenso problema de cálculo.
2. **Nivel de procedimientos:** se aplica a procedimientos o tareas (segmentos de programa) dentro del mismo programa. Esto supone la descomposición de un programa en múltiples tareas.
3. **Nivel de inter-instrucciones:** explota la concurrencia entre múltiples instrucciones. Se debe realizar el análisis de dependencia de datos para revelar paralelismos entre instrucciones.
4. **Nivel intra-instrucción:** Se implementa directamente por medio de hardware.

2.1.1 Necesidad del cómputo de alto desempeño en la solución de sistemas eléctricos.

El *cómputo de alto desempeño* se refiere al empleo eficiente de las herramientas computacionales que intervienen en un proceso general o problema a resolver. Tradicionalmente, la simulación numérica de sistemas complejos correspondía solo a la ciencia y la ingeniería y se realiza con máquinas con un solo procesador. Hoy en día se solicitan máquinas capaces de soportar aplicaciones comerciales y de servicios. La Tabla 2.1 presenta algunas de las aplicaciones que requieren cómputo de alto desempeño.

Tabla 2.1 Aplicaciones que requieren cómputo de alto desempeño.

Aplicaciones que requieren muchos ciclos de máquina. Dependen en gran medida de la velocidad y el procesamiento de punto flotante.	**Aplicaciones que dependen de la capacidad para almacenar y procesar grandes cantidades de información. Requieren de acceso rápido y seguro a una gran cantidad de datos almacenados.**	**Aplicaciones relativamente nuevas que exigen velocidad en comunicación. Son llamadas "servicios por demanda" y requieren de recursos computacionales conectados a redes con anchos de banda considerables.**
1. Dinámica de fluidos computacional. 2. Simulaciones electromagnéticas. 3. Modelado ambiental. 4. Dinámica estructural. 5. Modelado biológico. 6. Dinámica molecular. 7. Simulación de redes eléctricas. 8. Modelado financiero y económico.	1. Análisis de data sísmica. 2. Procesamiento de imágenes. 3. Minería de datos. 4. Análisis estadístico de datos. 5. Análisis de mercados.	1. Procesamiento de transacciones en línea. 2. Sistemas colaborativos. 3. Diagnostico médico a distancia. 4. Texto por demanda. 5. Vídeo por demanda. 6. Imágenes por demanda. 7. Simulación por demanda.

Respecto al hardware para computo de alto desempeño, es deseable que los cálculos se realicen a la mayor velocidad posible, que se optimice el uso de memoria y dispositivos de entrada y salida, que se tenga una potencia de cálculo elevada (la potencia de cálculo es proporcional a la escala de integración del procesador expresado en MIPs/Seg. o FLOPs/seg.), y que se permita mayor transferencia de datos en el menor tiempo posible.

Respecto al software para cómputo de alto desempeño, éste debe considerar un sistema operativo robusto, lenguajes de programación y sus compiladores, algoritmos adecuados para resolver el problema así como protocolos de comunicación para evitar errores o colisiones entre dispositivos.

Los estudios y aplicación de *sistemas paralelos* comenzaron hace ya varios años. Desde entonces, la computación paralela se ha dedicado a resolver necesidades y aplicaciones tales como:

- Cálculo de la dinámica de flujos: predicción del clima, cambios climatológicos, ciencias del mar, diseño de reactores nucleares, diseños de automóviles, aviones hipersónicos y submarinos.
- Cálculo para el diseño de estructuras de nuevos materiales: catalizadores químicos, agentes inmunológicos, diseño de fármacos, genoma humano, semiconductores y superconductores.
- Plasmas dinámicos para aplicaciones de energía de fusión: fusión nuclear, sistemas de combustión, en aire, en mar y con recursos submarinos.

- Computación simbólica: reconocimiento de voz, procesamiento de lenguaje natural, minería de datos, modelado de procesos financieros.
- Cálculos para entender las características fundamentales de la materia: cuantums cromodinámicos, astrofísica, análisis estructural, sismología, teoría de materia condensada.
- Realidad virtual, bases de datos distribuidas, diagnóstico médico asistido por computadora, procesamiento de imágenes para entretenimiento.

Las metodologías para la solución de redes eléctricas se implementaban y ejecutaban en sistemas de un solo procesador, y su programación era secuencial. Las soluciones que se obtuvieron fueron aceptables pero no escalables en la medida en que fueron creciendo el número y complejidad de los cálculos. Fue hasta hace aproximadamente dos décadas que empezaron las investigaciones que aplican procesamiento paralelo al computo digital de sistemas eléctricos de potencia. Algunas de las consideraciones por las cuales se hace necesario el cómputo de alto desempeño para sistemas eléctricos se resumen a continuación:

- El modelado de sistemas eléctricos debe describir completamente los elementos individuales (lineales y no lineales) de la red así como las relaciones que gobiernan la interconexión de estos elementos. El modelado se hace por medio de ecuaciones diferenciales y algebraicas, por lo que la complejidad del análisis aumenta [Oyama *et al.* 1990].
- Se aplican métodos numéricos donde intervienen operaciones algebraicas con matrices de gran tamaño y con alta dispersión (incluso procesos con matrices ordinarias para estocásticos y comportamientos no lineales) [LaScala *et al.* 1989], [Medina y Ramos-Paz 2005].
- La determinación de solución en estado estacionario periódico se realiza en base a algoritmos especiales, así que a medida que aumenta el número de variables que describen el comportamiento, se incrementa el tiempo de procesamiento. [Lee *et al*. 1989], [García *et al.* 2001], [Ramos 2002], [García y Acha 2004].
- El análisis armónico (detección y predicción de armónicos) es un tópico que requiere, por si mismo, un procesamiento aparte porque involucra un gran volumen de datos [Medina, Ramos-Paz y Fuerte-Esquivel 2003].

Actualmente se busca explotar las técnicas de procesamiento en paralelo como alternativa para hacer procesamiento cada vez más eficiente. La investigación y desarrollo del procesamiento en paralelo avanza en dos vertientes complementarias: diseño de hardware (con arquitecturas de procesadores en paralelos) y diseño de software (sistemas operativos y lenguajes paralelos, así como diseño de algoritmos paralelos).

2.1.2 Ventajas y desventajas de la programación en paralelo.

Ventajas:

- La potencia de cálculo no depende ya de la estructura física de los procesadores (excepto el caso de procesadores vectoriales), sino más bien de la cantidad de procesadores que intervienen en el proceso y la manera como se comunican entre si.
- Incrementa la velocidad efectiva de cómputo, permitiendo que un número mayor de instrucciones puedan ser ejecutadas en un mismo tiempo.
- Se puede escalar un proceso. Técnicas como PVM y MPI permiten ir aumentando el número de procesadores involucrados en el proceso, de manera que la eficiencia y la capacidad de cómputo se incremente.
- Volúmenes grandes de información pueden ser procesados. Aplicaciones tales como manejo de base de datos distribuidas, procesamiento digital de imágenes, o monitoreo de sistemas eléctricos en tiempo real son posibles.
- Se puede implementar cómputo de alto desempeño para cualquier área del conocimiento, inclusive se pueden diseñar nuevas arquitecturas de propósito específico.

Desventajas:

- La competencia por los recursos compartidos. La manera como se reparten y/o accede a la memoria, al bus de comunicación, a las peticiones de entradas/salidas, etc. es un problema que esta presente en los sistemas paralelos con arquitecturas fuertemente acopladas.
- Mayor complejidad de programación, comparada con la programación secuencial. Al fragmentar la memoria en segmentos muy pequeños, se requieren comandos especiales de paso de mensajes, entre otros. El empaquetamiento y envió de información en los sistemas distribuidos requiere de programadores especializados.
- Software especial para programación en paralelo. Sistemas operativos y lenguajes de programación deben ser compatibles con la arquitectura paralela.
- Algoritmos paralelos. No todos los algoritmos secuenciales pueden ser paralelizados.
- Complejidad de la comunicación entre procesadores en arquitecturas con memoria distribuida. Se debe cuidar y seguir minuciosamente el flujo de datos, su sincronización y su bloqueo.

2.2 CONCEPTOS BASICOS DEL PROCESAMIENTO PARALELO

Un *proceso* es un fragmento de código en ejecución que convive con otros fragmentos de código [Stallings 1998].

En el procesamiento paralelo se involucra la ejecución de diferentes procesos en dos o más elementos procesadores al mismo tiempo, para que todos juntos resuelvan un problema completamente. Los procesos pueden distinguirse en tres formas:

- *Procesos paralelos*: son los que pueden producirse en diferentes recursos durante el mismo intervalo de tiempo.
- *Procesos simultáneos* o *concurrentes*: son los que pueden producirse en el mismo instante de tiempo.
- Los *procesos solapados* son los que pueden producirse en intervalos de tiempo superpuestos.

2.2.1 Características del paralelismo.

El *paralelismo* ha sido definido como el uso de múltiples unidades funcionales para aplicar la misma operación simultanea a elementos de un conjunto de datos.

Los sistemas que emplean computadoras con un solo procesador son conocidos como *sistemas monoprocesador* y son los más utilizados en la actualidad. Sin embargo, tienen límites en su rendimiento debido a su potencia de cálculo, velocidad y su ancho de banda (*through put*).

La velocidad del procesador siempre será proporcional a la frecuencia de reloj con la que trabaja. Entonces, el límite físico de la velocidad de la luz será la máxima velocidad posible que puede alcanzar un procesador.

Un inconveniente encontrado en máquinas de un procesador es la escala de integración de sus componentes. A medida que aumenta la escala de integración, se incrementa la potencia de cálculo del procesador y su rendimiento general, pero la reducción del tamaño físico estará limitada por el diámetro de las moléculas de los elementos con los que esta construido el procesador [Jin 1995].

El *ancho de banda* que tiene el procesador indica el número de resultados que produce en una unidad de tiempo. Si se aumenta el ancho de banda en el diseño del procesador, aumenta también su velocidad y su grado de concurrencia.

Para incrementar el rendimiento del sistema monoprocesador se hicieron modificaciones a nivel arquitectura; se incluyeron múltiples procesadores con espacio de memoria y periféricos compartidos bajo el control de un sistema operativo integrado. Este nuevo sistema fue denominado *sistema multiprocesador*, el cual explota mucho mejor el paralelismo.

En la paralelización de una aplicación intervienen factores cuantitativos (qué tanto se puede paralelizar) y cualitativos (la técnica a través de la cual se explota el paralelismo). A la técnica de explotación del paralelismo se le denomina *fuente*, y se distinguen tres fuentes principales [Pardo 2002]:

- El paralelismo de control.
- El paralelismo de datos.
- El paralelismo de flujo.

2.2.1.1 El paralelismo de control

El paralelismo de control tiene la característica de que, para una aplicación dada, existen acciones que se pueden realizar "al mismo tiempo". Las acciones (también llamadas tareas o procesos) pueden ejecutarse de manera más o menos independiente sobre los procesadores elementales, que no son otra cosa que recursos de cálculo. La Figura 2.1 muestra la concepción del paralelismo de control.

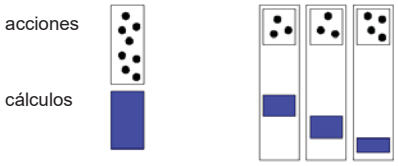


Figura 2.1 Paralelismo de control.

En el caso ideal, todas las acciones son independientes, y con asociar un recurso de cálculo a cada una de las acciones será suficiente para obtener una ganancia en tiempo de ejecución lineal, esto es:

"N acciones independientes se ejecutarán N veces más rápido sobre N elementos de proceso que sobre uno solo."

Sin embargo, en un caso real, las acciones de un programa suelen presentar dependencias entre ellas. Las dependencias suponen una sobrecarga de trabajo durante el proceso. Se distinguen dos tipos de dependencias:

- Dependencia de control de secuencia: correspondiente a la secuenciación en un algoritmo clásico.
- Dependencia de control de comunicación: una acción envía informaciones a otra acción.

En el paralelismo de control se administran las dependencias entre las acciones de un programa, con la finalidad de asignar recursos de cálculo lo más eficazmente posible, al minimizar las dependencias. La Figura 2.2 muestra un ejemplo de paralelismo de control.

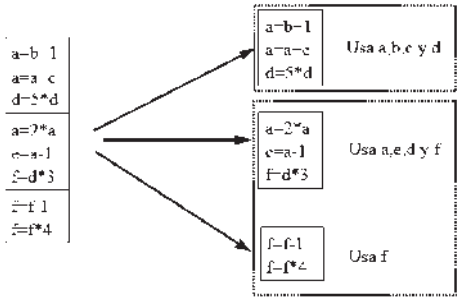


Figura 2.2 Ejemplo de paralelismo de control.

2.2.1.2 El paralelismo de datos.

En el paralelismo de datos se trabaja con estructuras de datos muy regulares, tales como vectores y matrices. Cada operación repite una misma acción sobre cada elemento de la estructura. Los recursos de cálculo se asocian a los datos. Los datos (millares o más) se reparten en los procesadores elementales disponibles. La Figura 2.3 muestra en forma gráfica el concepto de paralelismo de datos.

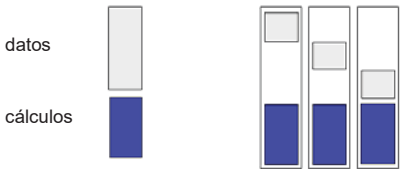


Figura 2.3 Paralelismo de datos.

Como las acciones efectuadas en paralelo sobre los procesadores elementales son idénticas, es posible centralizar el control. Siendo los datos similares, la acción para repetirlos tomará el mismo tiempo sobre todos los procesadores elementales y el controlador podrá enviar, de manera síncrona, la acción a ejecutar a todos los procesadores elementales.

2.2.1.3 El paralelismo de flujo

El paralelismo de flujo se basa en la idea de que ciertas aplicaciones funcionan en modo de cadena: sobre un flujo de datos semejantes se efectúa una sucesión de operaciones en cascada. La Figura 2.4 muestra de forma gráfica el concepto de paralelismo de flujo.

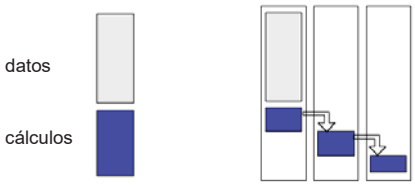


Figura 2.4 Paralelismo de flujo.

Los recursos de cálculo se asocian a las acciones en cadena, de manera que los resultados de las acciones efectuadas en el instante t pasen al procesador elemental siguiente en el instante $t+1$.

Este concepto de paralelismo se denomina también *segmentación* o *pipelining*. El flujo de datos puede provenir de dos fuentes:

- Datos de tipo vectorial ubicados en memoria (semejante al caso del paralelismo de datos).
- Datos de tipo escalar provenientes de un dispositivo de entrada (captura de datos, colocado en un entorno de tiempo real).

En ambos casos, la ganancia obtenida está en relación con el número de etapas (número de procesadores elementales).

2.2.2 Seudoparalelismo

El paralelismo se presenta también a niveles de micro-operación, en sistemas monoprocesadores debido a las técnicas de supersegmentación y técnicas superescalares conocidas como seudoparalelismo o *Segmentación Encauzada* (*pipelining*) [Duncan 1990] y [LaScala y Sbrizzai 1990]**.**

La segmentación se basa en la idea de trabajar con el *cause* (*pipe*) de instrucciones de manera simultanea. A niveles de micro-operación, se generan al mismo tiempo múltiples señales de control. La segmentación de las instrucciones, al menos en cuanto al solapamiento de las operaciones de captación y ejecución, se ha estudiado y utilizando desde hace tiempo y se aplica en procesadores con tecnología VLSI [Jin 1995], de manera que se puede hablar de funciones que se realizan en paralelo.

El aprovechamiento del seudoparalelismo hace que a los sistemas monoprocesadores se les denomine *procesadores vectoriales*, cuyas características se expondrán en la Sección 2.3.2.5.

2.2.3 Procesamiento distribuido

El *procesamiento distribuido* se refiere a sistemas multiprocesador donde los procesos se pueden ejecutar concurrentemente, mediante una interacción planificada y controlada denominada *proceso de comunicación* o proceso de sincronización.

El procesamiento paralelo y el procesamiento distribuido están estrechamente relacionados. En algunos casos, se utilizan ciertas técnicas distribuidas para conseguir paralelismo. Conforme la tecnología de las comunicaciones de datos progresa, la distinción entre procesamiento paralelo y procesamiento distribuido se hace más y más pequeña. En este sentido, se considera al procesamiento distribuido como una forma de procesamiento paralelo en un entorno especial cuyas características son:

- Procesos *de comunicación* que deben tener lugar a través de variables compartidas o globales.
- Procesos *cooperantes* que deben comunicarse para sincronizar o limitar su concurrencia.

La relación entre dos procesos cooperantes respecto a un recurso los convierte en *procesos competidores* o en *procesos productores-consumidores*. Como la comunicación entre procesos tiene lugar a través de memoria compartida, los procesos competidores acceden a la memoria para apoderarse o liberar recursos permanentes o reutilizables. Los procesos productores-consumidores acceden a la memoria para pasar recursos temporales o consumibles tales como mensajes y señales.

Los procesos que necesiten el acceso exclusivo a un recurso pueden competir por él. Las mismas competiciones aparecen con relación a los accesos a los que se les denomina recursos virtuales, tales como tablas del sistema o directorios, y memorias intermedias (*buffers*) de comunicación entre los procesos cooperantes.

2.2.4 Multiprocesamiento simétrico

El *Multiprocesamiento Simétrico* SMP (*Symmetric Multiprocessing*) tiene una arquitectura de diseño simple: interconexión de procesadores que comparten la misma memoria RAM y el bus del sistema. A este diseño se le conoce también como *estrechamente acoplado* (*Tightly Coupled*), o *compartiendo todo* (*Shared Everything*). La Figura 2.5 ilustra el esquema del SMP [Fernández 1996].

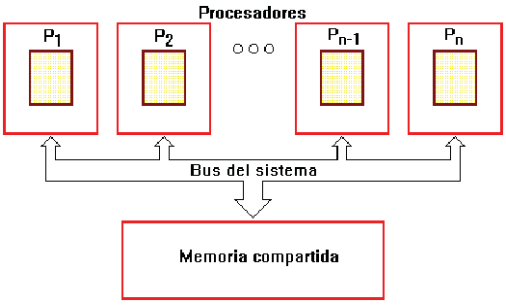


Figura 2.5 Esquema del SMP.

El hecho de compartir una misma memoria RAM tiene como ventajas:

- Uso de un espacio único de memoria, lo que permite que un sistema operativo con multiconexión o *multihilos* (*Multithreaded Operating System*) distribuya las tareas entre los procesadores.
- La sincronización de los datos se hace más fácil.
- Permite que las aplicaciones obtengan la memoria que les sea necesaria cuando realizan procesos complejos.
- Simplifica tanto el hardware como la programación de aplicaciones.

El uso de memoria única tiene también desventajas: a medida que se escala el sistema el tráfico en el bus se satura. Para evitar la saturación se opta por añadir memoria caché a cada procesador, desahogando un poco el tráfico en el bus. Sin embargo, esta medida es insuficiente al tener ocho o más procesadores, ya que el bus se convierte en un cuello de botella. Por esta razón

el SMP no se considera una tecnología escalable. En la Figura No. 2.6 se muestra la comparación entre rendimiento y cantidad de procesadores en SMP [Fernández 1996].

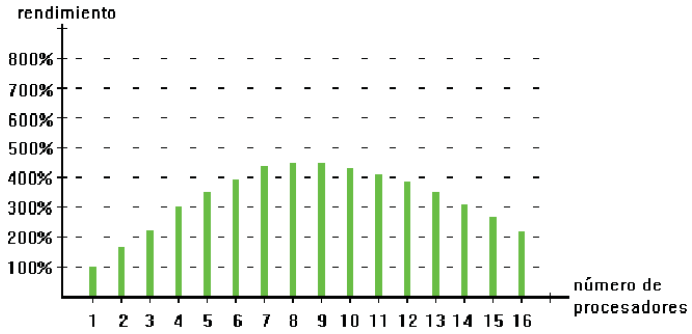


Figura 2.6 Rendimiento del SMP.

2.2.5 Procesamiento masivamente paralelo

El *Procesamiento Masivamente Paralelo* MPP (*Massively Parallel Processing*) es un diseño mejorado que pretende evitar los cuellos de botella que se presentan en el bus del SMP. En su diseño, MPP no utiliza memoria compartida, sino que distribuye la memoria RAM entre los procesadores de forma semejante a una red donde cada procesador tiene su memoria distribuida asociada a memoria de los demás procesadores. Debido a que la memoria RAM se encuentra distribuida, a esta arquitectura se le conoce también como *dispersamente acoplada* (*Loosely Coupled*), o *compartiendo nada* (*Shared Nothing*). En la Figura 2.7 se presenta el esquema para el MPP.

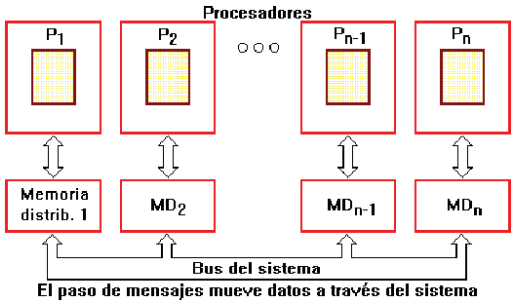


Figura 2.7 Esquema del MMP.

Para tener acceso a la memoria de otro procesador se utiliza un esquema de paso de mensajes semejante a los paquetes de datos que se empelan en redes. Con este sistema se reduce el tráfico del bus ya que cada sección de memoria observa únicamente aquellos accesos que le son destinados, en vez de observar todos los accesos, como ocurre en un sistema SMP. En caso de que un procesador no disponga de la memoria RAM suficiente se utilizará la memoria RAM sobrante de los otros procesadores. De esta manera los sistemas MPP permiten la integración de un número considerable de procesadores, haciendo del MPP una tecnología escalable. En la Figura 2.8 se puede observar el rendimiento en relación a la cantidad de procesadores en el MPP [Fernández 1996].

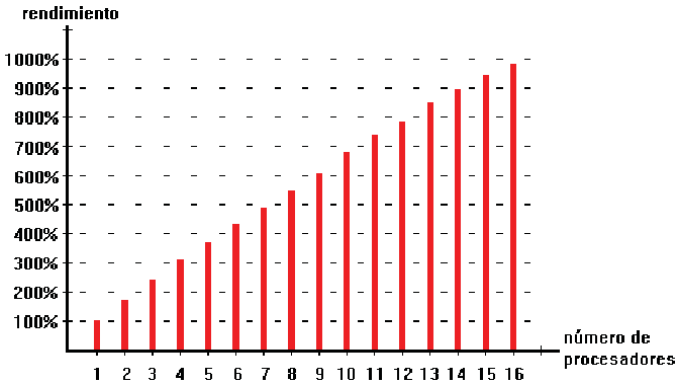


Figura 2.8 Rendimiento del SMP.

La principal desventaja del MPP es la programación. A medida que se aumenta el número de procesadores, la memoria se fragmenta en pequeños espacios separados. Al no tener un espacio de memoria común, escribir y ejecutar una aplicación que requiere una gran cantidad de memoria, puede sobrepasar la cantidad de RAM local de cada procesador; además, la sincronización de datos entre tareas distribuidas también se torna difícil, particularmente si un mensaje debe pasar por muchas fases hasta alcanzar la memoria del procesador al que va destinado.

Formular código de programa para MPP requiere insertar comandos de paso de mensajes, lo cual resulta complicado porque tales comandos pueden crear dependencias de hardware en las aplicaciones, lo que es altamente indeseable. Para simplificar en lo posible el paso de mensajes en el MPP se han adoptando mecanismos de dominio público como son la *Máquina Virtual Paralela* (PVM), o un el estándar llamado *Interfase de Paso de Mensajes* (MPI).

2.3 CLASIFICACION DE LOS SISTEMAS PARALELOS

2.3.1 Taxonomía de Flynn

Las arquitecturas paralelas pueden ser clasificadas dependiendo de su modo de ejecución. Una clasificación ampliamente aceptada es la taxonomía introducida por Michael Flynn [Flynn 1972]. Esta taxonomía de las arquitecturas está basada en la clasificación atendiendo al flujo de datos e instrucciones en un sistema.

Un *flujo de instrucciones* es el conjunto de instrucciones secuenciales que son ejecutadas por un único procesador. Un *flujo de datos* es el flujo secuencial de datos requeridos por el flujo de instrucciones. Flynn clasifica los sistemas en cuatro categorías:

- Flujo único de instrucciones y flujo único de datos o SISD (*Single Instruction stream, Single Data stream*).
- Flujo múltiple de instrucciones y único flujo de datos o MISD (*Multiple Instruction stream, Single Data stream*).

- Flujo de instrucción simple y flujo de datos múltiple o SIMD (*Single Instruction stream, Multiple Data stream*).
- Flujo de instrucciones múltiple y flujo de datos múltiple o MIMD (*Multiple Instruction stream, Multiple Data stream*).

2.3.1.1 Modelo SISD. Flujo único de instrucciones y flujo único de datos.

El concepto de arquitectura SISD se refiere a la arquitectura clásica de Von Neumann. Aquí un único procesador interpreta una única secuencia de instrucciones para operar con los datos almacenados en la única memoria. La Figura 2.9 ilustra el diagrama SISD.

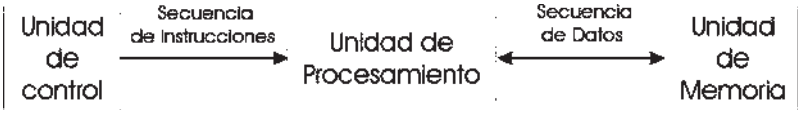


Figura 2.9 Diagrama a bloques de un procesador SISD.

Todas las máquinas SISD poseen un registro simple mejor conocido como *contador de programa*, y asegura la ejecución en serie del programa. Conforme se van leyendo las instrucciones de la memoria, el contador de programa se actualiza para que apunte a la siguiente instrucción a procesar en serie. Prácticamente, ninguna computadora totalmente SISD se fabrica hoy en día, ya que la mayoría de procesadores modernos incorporan algún grado de paralelización, como es la segmentación de instrucciones (*pipelining*) o la posibilidad de lanzar dos instrucciones a un tiempo (superescalares).

2.3.1.2 Modelo MISD. Flujo múltiple de instrucciones y único flujo de datos.

Está clasificación considera varias instrucciones que actúan sobre un único "paquete o segmento" de datos. Sin embargo, el modelo MISD es considerada una arquitectura impráctica, y no existen ejemplos totalmente aceptados por la comunidad científica que operen siguiendo este modelo.

Recientemente se consideró otra forma de interpretar el modelo MISD, entendiéndolo como una clase de máquinas donde un mismo flujo de datos fluye a través de numerosas unidades procesadoras. Los procesadores vectoriales, arquitecturas altamente segmentadas, suelen ser clasificados a menudo bajo este concepto de máquina.

Las arquitecturas segmentadas, o encauzadas, realizan el procesamiento vectorial a través de una serie de etapas, cada una ejecutando una función particular produciendo un resultado intermedio. La razón por la cual dichas arquitecturas son clasificadas como MISD es que los elementos de un vector pueden ser considerados como pertenecientes al mismo dato, y todas las etapas del cauce representan múltiples instrucciones que son aplicadas sobre ese vector.

2.3.1.3 Modelo SIMD. Flujo de instrucción simple y flujo de datos múltiple.

En el modelo SIMD una única instrucción de máquina controla paso a paso la ejecución simultánea y *sincronizada*[1] de un cierto número de elementos de proceso. Esto significa que una única instrucción es aplicada sobre diferentes datos al mismo tiempo.

Cada elemento del proceso tiene una memoria asociada de modo que cada instrucción es ejecutada por cada procesador para un conjunto de datos diferentes. En las máquinas de este tipo, varias unidades de procesamiento diferentes son invocadas por una única unidad de control. La Figura 2.10 muestra el diagrama SIMD.

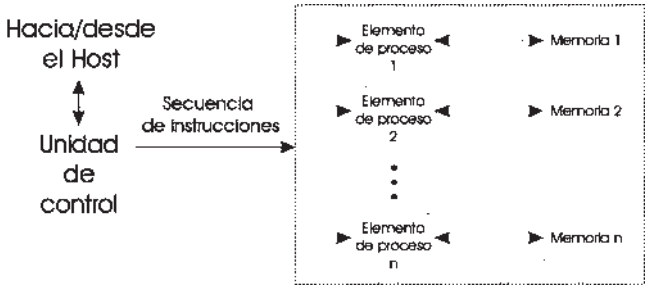


Figura 2.10 Diagrama a bloques de un procesador SIMD con memoria distribuida.

Dentro de esta clasificación están incluidos los procesadores vectoriales y procesadores matriciales.

2.3.1.4 Modelo MIMD. Flujo de instrucciones múltiple y flujo de datos múltiple.

El modelo MIMD es un conjunto de procesadores que ejecuta una secuencia de instrucciones diferentes con conjuntos de datos diferentes en forma simultánea. Estas arquitecturas son las más complejas, pero son también las que ofrecen una mayor eficiencia en la ejecución concurrente o paralela. La concurrencia implica que no sólo hay varios procesadores operando simultáneamente, sino que además hay varios procesos (fragmentos de programas) ejecutándose también al mismo tiempo. Aquí, de manera sincrona, la unidad de control determina el tiempo sobre todos los procesadores y la acción a ejecutar para cada procesador.

El modelo MIMD se puede subdividir según la manera en que se comparten los datos. Si lo hacen desde la memoria principal, se comunican a través de la memoria y se conoce como *multiprocesador* (Figura 2.11). Si la comunicación es a través de buses o algún mecanismo de comunicación de mensajes, el sistema es *multicomputador* (Figura 2.12).

[1] La sincronización es el conjunto de reglas que restringen el orden en que se ejecutarán los procesos, forzándolos a detener o continuar su ejecución

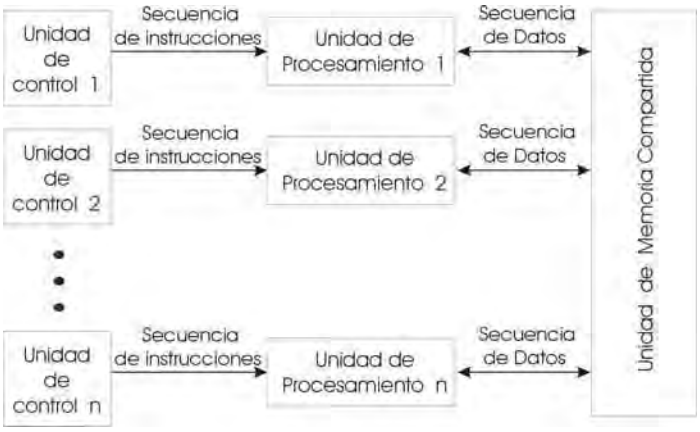


Figura 2.11 Diagrama a bloques de un procesador MIMD con memoria compartida.

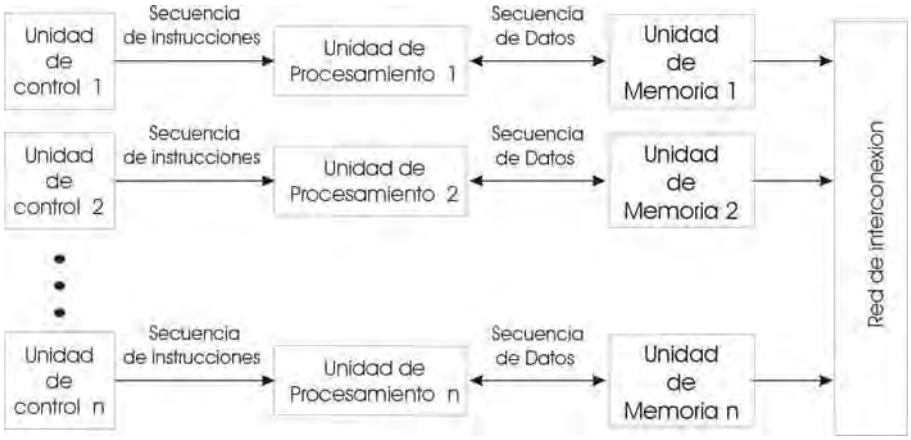


Figura 2.12 Diagrama a bloques de un procesador MIMD con memoria distribuida.

El multiprocesamiento simétrico SMP y el procesamiento masivamente paralelo MPP se desarrollan en la arquitectura MIMD.

2.3.2 Otra clasificación de arquitecturas paralelas

Los avances en tecnología de hardware han llevado al diseño de sistemas que no son tan fáciles de clasificar dentro de los 4 tipos básicos de Flynn. Por ejemplo, los procesadores vectoriales no encajan adecuadamente en esta clasificación, ni tampoco las arquitecturas híbridas. Para solucionar esto se han propuesto otras clasificaciones, donde los tipos SIMD y MIMD de Flynn se suelen conservar, pero además se incorporan otros nuevos modelos [Hwang y Briggs 1984]. La Figura 2.13 muestra una taxonomía ampliada que incluye los avances en arquitecturas paralelas existentes en los últimos años.

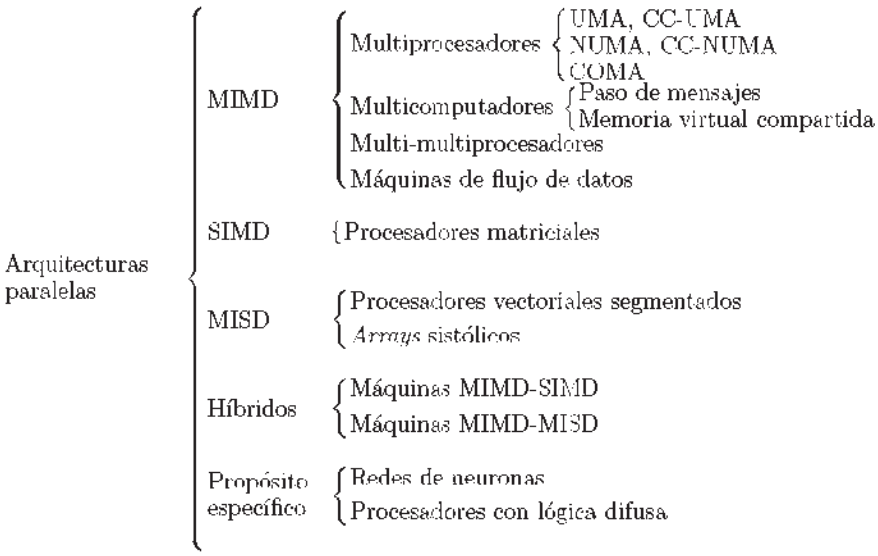


Figura 2.13 Clasificación actual de las arquitecturas paralelas.

Tal y como se observa en la Figura 2.13, el modelo MIMD pueden subdividirse además en *multiprocesadores*, *multicomputadores*, *multi-multiprocesadores* y en *máquinas de flujo de datos*. El tipo SIMD quedaría con los procesadores matriciales y el MISD se subdividiría en procesadores vectoriales y en arreglos sistólicos. Se han añadido dos tipos más que son el híbrido y los de aplicación específica.

2.3.2.1 Multiprocesador

Un multiprocesador es una arquitectura paralela compuesta por varios procesadores interconectados que pueden compartir una misma área de memoria. Los procesadores se pueden configurar para que ejecuten cada uno una parte de un programa o varios programas al mismo tiempo.

El multiprocesador está formado por *n* procesadores (*P*1, *P*2, ... , *Pn*) y *m* módulos de memoria (*M*1, *M*2, ... , *Mn*) interconectados por una red o bus. A los multiprocesadores se les conoce como sistemas de memoria compartida debido a que comparten los diferentes módulos de memoria, además de que varios procesadores pueden tener acceso a un mismo modulo. Un diagrama de bloques de esta arquitectura se muestra en la Figura 2.14.

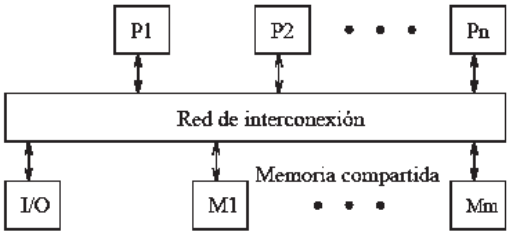


Figura 2.14 Modelo de un multiprocesador.

Incluso los multiprocesadores pueden ser subdivididos, según el modelo de memoria compartida, en:

- Memoria de Acceso Uniforme UMA (*Uniform Memory Access*).
- Memoria de Acceso No-Uniforme NUMA (*Non-Uniform Memory Access*).
- Únicamente Acceso a Memoria Caché COMA (*Cache Only Memory Access*).

En [Pardo 2002] se especifican las características principales de UMA, NUMA y COMA.

2.3.2.2 Multicomputador

Un *multicomputador* se puede ver como una máquina paralela en la cual cada procesador tiene su propia memoria local. La memoria del sistema se encuentra *distribuida* entre todos los procesadores y cada procesador sólo puede direccionar su propia memoria local. En otras palabras, un procesador tiene acceso directo sólo a su memoria local, y acceso indirecto al las memorias del resto de procesadores. Para acceder a las memorias de los demás procesadores debe hacerlo por medios indirectos como el de *paso de mensajes*.

La diferencia entre un multiprocesador y un multiprocesador es que la red de interconexión no permite un acceso directo entre memorias, sino que la comunicación se realiza por paso de mensajes. Las características de un multicomputador son las siguientes:

- La transferencia de datos se realiza a través de la red de interconexión que conecta un subconjunto de procesadores con otro subconjunto.
- La transferencia de procesadores a otros se realiza a través de múltiples transferencias entre los procesadores conectados y dependiendo de cómo sea establecida la red de comunicación.
- Debido a que la memoria está distribuida entre los diferentes elementos de proceso, a estos sistemas se les llama *distribuidos*.
- Son sistemas llamados débilmente acoplados, debido a que sus módulos funcionan de forma casi independiente unos de otros.

2.3.2.3 Máquinas de flujo de datos

Hay dos formas de procesar la información. La primera es mediante la ejecución en serie de una lista de comandos coordinados por un contador de programa. (arquitectura de Von Neumann). En la segunda forma se ejecutan las instrucciones en el momento que tienen los datos necesarios para ello, pudiéndose ejecutar todas las instrucciones demandadas en un mismo tiempo. Los lenguajes que se adaptan a este tipo de arquitectura comandada por datos son Prolog, y ADA, ya que son lenguajes que explotan la concurrencia de instrucciones.

Una *máquina de flujo de datos* es una arquitectura que permite que una instrucción esté lista para su ejecución cuando los datos que necesita están disponibles. La disponibilidad de los datos se consigue al canalizar los resultados de las instrucciones previamente ejecutadas a los operandos de las instrucciones que esperan. Esta canalización forma un flujo de datos que van

disparando las instrucciones a ejecutar, evitando así la ejecución de instrucciones basada en contador de programa.

Las instrucciones en un flujo de datos son puramente *autocontenidas*, es decir, no direccionan variables en una memoria compartida global, sino que llevan los valores de las variables en ellas mismas.

En una maquina de flujo de datos la ejecución de una instrucción no afecta a otras que estén listas para su ejecución. Por ello, varias instrucciones pueden ser ejecutadas simultáneamente, lo que lleva a la posibilidad de un alto grado de concurrencia y paralelización.

La Figura 2.15 muestra el diagrama de bloques de una máquina de flujo de datos. Las instrucciones, junto con sus operandos, se encuentran almacenados en la memoria de datos e instrucciones (D/I). Cuando una instrucción está lista para ser ejecutada, se envía a uno de los elementos de proceso (EP) a través de la red de arbitraje. Cada elemento de proceso es un procesador simple con memoria local limitada. El elemento de proceso, después de ejecutar la instrucción, envía el resultado a su destino a través de la red de distribución.

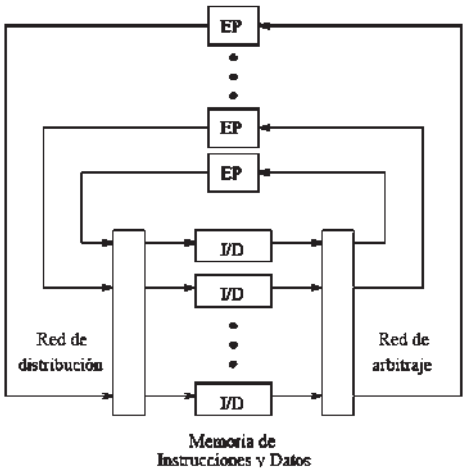


Figura 2.15 Diagrama de bloques de una máquina de flujo de datos.

2.3.2.4 Procesadores matriciales

Un procesador matricial consiste en un conjunto de elementos de proceso y un procesador escalar que operan bajo una unidad de control. Esta arquitectura es representativa del tipo SIMD, es decir, hay una sola instrucción que opera concurrentemente sobre múltiples datos. Su funcionamiento es el siguiente:

- La unidad de control busca y decodifica las instrucciones de la memoria central y las manda ya sea al procesador escalar o a los nodos procesadores, dependiendo del tipo de instrucción.
- La instrucción que ejecutan los nodos procesadores es la misma simultáneamente, los datos serán los de cada memoria de procesador y por tanto serán diferentes. Por esto, un

procesador matricial sólo requiere un único programa para controlar todas las unidades de proceso.

La idea de utilizar procesadores matriciales es explotar el paralelismo en los datos de un problema, más que paralelizar la secuencia de ejecución de las instrucciones.

Un tipo de datos altamente divisible es el formado por vectores y matrices, por eso a estos procesadores se les llama matriciales. La Figura 2.16 muestra la estructura de un procesador matricial (*A* y *B* y *C* son matrices)

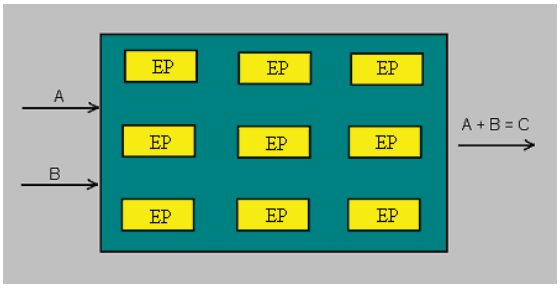


Figura 2.16 Procesador matricial.

2.3.2.5 Procesadores vectoriales

Un procesador vectorial ejecuta de forma segmentada instrucciones sobre vectores. La diferencia con los procesadores matriciales es que mientras los matriciales son comandados por las instrucciones, los vectoriales son comandados por flujos de datos continuos. A este tipo se le considera MISD puesto que varias instrucciones son ejecutadas sobre un mismo dato (el vector), si bien es una consideración algo confusa aunque aceptada de forma mayoritaria.

2.3.2.6 Arreglos sistólicos

El *arreglo sistólico* es una máquina cuya arquitectura tiene un gran número de elementos de proceso idénticos con una limitada memoria local. Los elementos de proceso están colocados en forma de matriz (arreglo) de manera que sólo están permitidas las conexiones con los elementos de proceso vecinos. Por lo tanto, todos los procesadores se encuentran organizados en una estructura segmentada de forma lineal o matricial. Los datos fluyen de unos elementos de proceso a sus vecinos a cada ciclo de reloj, y durante ese ciclo de reloj, o varios, los elementos de proceso realizan una operación sencilla.

El adjetivo *sistólico* viene del hecho de que todos los procesadores están sincronizados por un único reloj que simula un “corazón” que hace moverse a la máquina. La Figura 2.17 ilustra el flujo de datos en arquitectura sistólica. Este tipo de máquinas se suelen considerar MISD también.

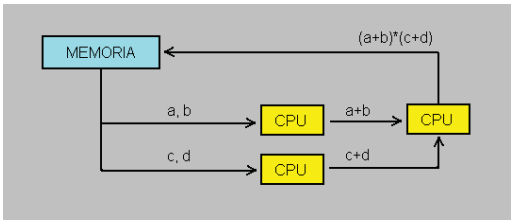


Figura 2.17 Flujo de datos en arquitectura sistólica

2.3.2.7 Arquitecturas específicas

Las *arquitecturas específicas* son conocidas también con el nombre de *arquitecturas VLSI* ya que muchas veces llevan consigo la elaboración de circuitos específicos con una alta escala de integración. Dos ejemplos de estas arquitecturas son:

Neurocomputadoras. Basadas en *redes neuronales* ANN (*Artificial Neural Network*) consisten en un elevado número de elementos de proceso muy simples que operan en paralelo. Estas arquitecturas se pueden utilizar para el reconocimiento de patrones, comprensión del lenguaje, etc. La diferencia con las arquitecturas clásicas es la forma en que se programa. Mientras en la arquitectura de Von Neumann se aplica un programa o algoritmo para resolver un problema, una red de neuronal aprende a fuerza de aplicarle patrones de comportamiento.

Procesadores basados en *lógica difusa*. Estas arquitecturas aplican los principios del razonamiento aproximado. La lógica difusa trata con la complejidad de los procesos humanos eludiendo los inconvenientes asociados a lógica de dos valores clásica.

2.4 TOPOLOGIAS DE RED DE PROCESADORES

En el modelo MIMD los procesadores deben comunicarse mediante paso de mensajes, y el elemento clave es el diseño de su red de interconexión.

Una *topología* se refiere a una red que da soporte a un amplio número de procesadores, y además considera la máxima longitud de los enlaces entre nodos (procesadores) y el número de conexiones físicas para proporcionar escalabilidad y comportamiento eficiente [Stallings 1998].

2.4.1 Modelos de interconexión comunes

Aún cuando se han desarrollado varios modelos de topologías para procesamiento en paralelo, tales como el anillo, estrella, malla, hipercubo, árbol etc. [Stallings 1998], en la actualidad han tomado especial relevancia las topologías basadas en cluster [Hoeger 2002]. En esta Sección se presentará solamente la topología tipo estrella y el cluster, debido a que son las redes que se van usar en esta Tesis. En la Tabla 2.2 se resumen las características más importantes de los modelos de conexión de redes.

Tabla 2.2 Resumen de las características de los modelos de conexión de redes.

Tipo de red	Grado nodal (d)	Diámetro de la red (D)	Anchura bisección (B)	Simetría	Notas sobre el tamaño
Malla simple	2	$N - 1$	1	No	N nodos
Anillo	2	$N/2$	2	Sí	N nodos
Conectado completo	$N - 1$	1	$(N/2)^2$	Sí	N nodos
Arbol binario	3	$2(h - 1)$	1	No	Altura árbol $h = [\log 2\ N]$
Estrella	N - 1	2	$(N/2)$	No	N nodos
Malla 2D	4	$2(r - 1)$	r	No	Malla r x r con $r = \sqrt{N}$
Malla Illiac	4	r - 1	$2r$	No	Equivalente con malla 2D
Toro 2D	4	$2(r=2)$	$2r$	Sí	Toro r x r con $r = \sqrt{N}$
Hipercubo	n	n	$N/2$	Sí	N nodos $n = \log_2 N$ (dimensión)

- El *grado nodal* define el número de enlaces que posee un nodo con otros. Si el grado nodal de la red es menor que 4, indica que es una topología aceptable.
- El *diámetro de la red* esta dado por la mayor distancia entre nodos. Mientras menor sea el diámetro menor será el tiempo de comunicación entre nodos.
- La *anchura de bisección* de la red es el número menor de enlaces que deben ser removidos para dividir la red a la mitad. Un ancho de bisección alto es preferible porque hace a la red tolerante a fallas, como la desconexión de algún nodo, y garantiza la reducción de la congestión entre nodos porque la información puede viajar por caminos alternos.
- La *simetría* relaciona diámetro (D) y el número de enlaces (l) de la red. En general, la propiedad de simetría afecta a la escalabilidad y a la eficiencia, es decir, el coste total de la red crece con D y l , por tanto, es deseable que D sea pequeño.

Analizando la información de la Tabla 2.2 se observa que, para la configuración tipo estrella que se implementó en esta Tesis, se tiene un grado nodal que crecerá en función de N, lo cual no es muy recomendable porque a medida que se adicionen nodos, aumentará el coste de la red.

2.4.1.1 Topología Estrella

Consiste en una conexión donde un grupo de procesadores comparten un mismo recurso de comunicación. Esta topología en muy popular tanto en multiprocesadores como en multicomputadores. Las características de la estrella son muy semejantes a las de la topología anillo, excepto por que la distancia máxima entre dos nodos cualesquiera es 2. En cuanto al recurso de comunicación, la estrella presenta dos variantes: tipo *bus* y tipo *Ethernet*. La configuración tipo bus es fácil y económica de implementar, pero es altamente no escalable si los nodos comparten una misma memoria. El tipo Ethernet es muy popular en estaciones de trabajo y clusters formados por computadoras personales. La Figura 2.18 ilustra variantes de la topología tipo estrella.

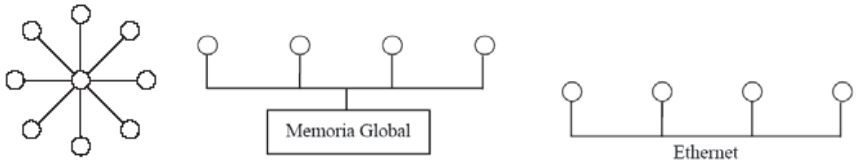


Figura 2.18 Interconexión tipo estrella.

El ancho de banda biseccional de la conexión estrella puede mejorarse con un canal de comunicación más ancho.

2.4.2 Cluster

Un *cluster* es un conjunto de computadoras independientes interconectadas, usadas como un recurso de cómputo [Foster 1995]. Al combinar el poder de muchas máquinas del tipo estación de trabajo o servidor, se pueden alcanzar niveles de rendimiento similares a los de las supercomputadoras, pero a menor costo.

2.4.2.1 Componentes de los Clusters

En un cluster, el principal componente son los *nodos*. Un nodo se refiere a una computadora sola, que contiene recursos específicos, tales como memoria, interfaces de red, uno o más CPU, etc.

Un segundo componente del cluster es el sistema operativo, mismo que debe tener su kernel creado especialmente para soportar una topología cluster. Además del sistema operativo se requiere un conjunto de compiladores y aplicaciones especiales, que permiten que los programas que se ejecutan sobre esta plataforma tomen las ventajas de esta tecnología de clusters.

Un tercer componente es el compilador, que puede paralelizar automáticamente un código de programa. La parte más complicada se refiere al análisis de dependencia de accesos de datos (manejar la localidad de datos en los caches y en la memoria principal). Los compiladores aún no son capaces de paralelizar eficientemente programas complejos, especialmente aquellos que hacen un uso substancial de apuntadores, pues la dirección que guarda un apuntador no se conoce durante la fase de compilación.

El cuarto componente es la interconexión de hardware entre los nodos del cluster. Se han desarrollado interfaces de interconexión especiales muy eficientes para la optimización de la comunicación basándose en las características de desempeño de la arquitectura. Comúnmente las interconexiones se realizan mediante una red Ethernet de alta velocidad, de manera que los nodos del cluster puedan intercambiar entre si la asignación de tareas, actualizaciones de estado y datos del programa.

El último componente es la interfaz de red que conecta al cluster con el mundo exterior.

2.4.2.2 Tipos de tecnología de clusters

Cluster de alto rendimiento. Un número grande de máquinas individuales actúan como una sola máquina muy potente. Este tipo de clusters se aplica mejor en problemas grandes y complejos que requieren una cantidad enorme de potencia computacional. Entre las aplicaciones más comunes de este tipo de cluster se encuentra el pronóstico numérico del estado del tiempo, astronomía, investigación en criptografía, análisis y procesamiento de imágenes, etc.

Cluster de servidores virtuales. Se trata de un conjunto de servidores de red que comparten la carga de trabajo de tráfico de sus clientes. Al balancear la carga de trabajo de tráfico en un arreglo de servidores, se mejora el tiempo de acceso y la confiabilidad. Además, como es un conjunto de servidores el que atiende el trabajo, la falla de uno de ellos no ocasiona una falla catastrófica total. Este tipo de servicio es de gran valor para compañías que manejan grandes volúmenes de tráfico en sus sitios Web.

Clusters de alta disponibilidad: Son un conjunto de servidores que actúan entre ellos como respaldos de la información que sirven. Este tipo de clusters se les conoce también como *cluster de redundancia*. Los servidores de un cluster de *alta disponibilidad* normalmente no comparten la carga de procesamiento que tiene un cluster de *alto rendimiento*. Tampoco comparten la carga de tráfico como lo hacen los clusters de *balance de carga*. Por lo anterior, se concluye que cada tecnología de cluster va encaminada hacia un propósito de computación específico.

Los beneficios de construir un cluster se observan en una variedad de aplicaciones y ambientes, tales como:

- Incremento de velocidad de procesamiento ofrecido por los clusters de alto rendimiento.
- Incremento del número de transacciones o velocidad de respuesta ofrecido por los cluster de balance de carga.
- Incremento de confiabilidad ofrecido por los clusters de alta disponibilidad.

El procesamiento de señales digitales, incluido audio y video, o la exploración del universo o la investigación meteorológica, en donde para realizar el pronóstico del estado del tiempo, son ejemplos de aplicaciones de clusters; y cada vez están más presentes en aplicaciones donde se requiere el manejo de cantidades masivas de datos y cálculos numéricos muy complejos.

Los clusters ocupan el 82 % de las 500 principales supercomputadora del mundo [TOP500 2008]. En la Tabla 2.3 se muestra la distribución de las 500 principales supercomputadoras según su arquitectura.

Tabla 2.3 Distribución de arquitecturas de las 500 principales supercomputadoras.

ARQUITECTURA	NUMERO	%	GFLOPS	PROCESADORES
Constelaciones (Cluster de SMP)	2	0.40	94970	17648
MPP	88	17.60	6654298	1412525
Cluster	410	82.00	10209332	1691406
Total sistemas	500	100	16958600.19	3121579

Hasta la fecha, la supercomputadora *Cray XT5 Jaguar* es la más rápida del mundo con un poder de cómputo de 1.059 petaflop [TOP500 2008]. En México la UNAM, UAM-I, IPN y CNSCC- IPICyT son los principales centros de supercómputo académico. El CNSCC-IPICyT está a la cabeza con la *Cray XD1*, que tiene un poder de cómputo de 950 GFlops en un solo equipo [CNSCC- IPICyT 2004]. La UNAM posee la supercomputadora paralela *KanBalam* de memoria distribuida que tiene 1368 procesadores y un poder de cómputo de más de 7 GFlops/seg. Además cuenta con tres clusters para proyectos específicos o docencia [UNAM 2008]. Información referente a las supercomputadoras y su clasificación se puede consultar en el Apéndice D.

2.4.2.3 Cluster tipo Beowulf

La tecnología *Beowulf* es la tecnología de clusters de alto rendimiento para Linux [BEOWULF1 2003].

En 1994 Thomas Sterling y Don Becker, trabajando en CESDIS (*Center of Excellence in Space Data and Informarion Sciencies*) bajo el patrocinio del "Proyecto de la Tierra y Ciencias del Espacio (*ESS*)", construyeron un cluster que consistía en 16 computadoras personales (PC´s) con procesadores DX4 conectados por un canal Ethernet a 10 Mbps. A esta configuración de máquina la llamaron *Beowulf*. El cluster de PC´s desarrollado logró una eficiencia de 70 Mflops (millones de operaciones de punto flotante por segundo).

El *cluster Beowulf* se puede definir como una colección de nodos (PC´s), que se dedican exclusivamente a ejecutar tareas asignadas al cluster, y están interconectados por medio de una red privada de alta velocidad, teniendo como sistema operativo alguna distribución de Linux [BEOWULF1 2003].

Por lo general, el cluster Beowulf se encuentra comunicado al mundo exterior a través de un solo nodo, llamado *nodo maestro*, el cual también está reservado para acceder, compilar y manejar las aplicaciones a ejecutar. La Figura 2.19 muestra un esquema del cluster tipo Beowulf.

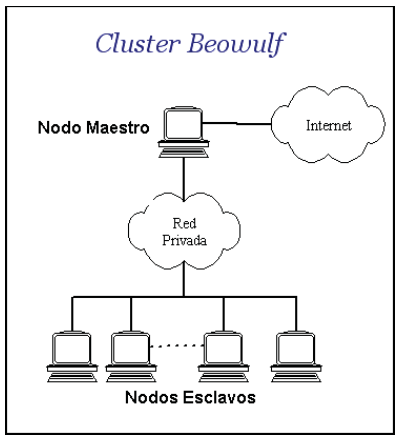


Figura 2.19 Esquema del Cluster tipo Beowulf.

El cluster Beowulf fue construido específicamente para programación paralela. En cuanto al software, en general es necesario repartir las tareas paralelas que se comunican usando alguna plataforma paralela como MPI, PVM, o *sockets* [BEOWULF1 2003].

Los Beowulf se distinguen de otros sistemas de clusters porque no imponen una topología del sistema. Los componentes primarios de la arquitectura del sistema se pueden descomponer en el procesador, memoria, red de trabajo y sistema de almacenamiento secundario (que puede existir o no). En la taxonomía de las computadoras paralelas, el cluster Beowulf se localiza entre los sistemas MPP (como nCube, CM5, Convex SPP, Cray T3D, Cray T3E, etc.) y red de estaciones de trabajo NoW´s (*Network of Workstations*). En la Tabla 2.4 se muestran las características que diferencian al cluster Beowulf de las estaciones de trabajo (NOWs) [BEOWULF2 2003].

Tabla 2.4 Características que diferencian un cluster Beowulf y una estación de trabajo.

CARACTERISTICA	CLUSTER BEOWULF	ESTACION DE TRABAJO (NOW)
Acceso	Se hace a través de una terminal conectada a la red del cluster. Esta terminal por lo general no se utiliza como elemento de proceso para cálculos.	Se puede acceder desde cualquiera de los nodos que conforman la red.
Independencia de los nodo	Cada nodo tiene el hardware mínimo que le permite funcionar como unidad de cálculo. Los nodos carecen de monitores, teclados, disqueteras, ratones y tarjetas de video (en ocasiones hasta de disco duro).	Cada nodo es una estación de trabajo completamente funcional que, con el software apropiado, puede funcionar sin necesidad de participar en el sistema distribuido
Mecanismo de comunicaciones	Conexión de tarjeta a tarjeta por cable UTP cruzado con conectores RJ- 45.	Utiliza un switch central que determina el rendimiento para evitar colisiones.
Exclusividad	Cada nodo se dedica exclusivamente a los procesos propios del cluster.	Cada nodo puede ejecutar simultáneamente los procesos correspondientes a la NOW y cualquier otro programa o aplicación para monoprocesador.
Direccionamiento	El nodo maestro tiene una dirección IP válida. El resto de los nodos tienen direcciones de red privada (192. 168.xxx.xxx), de forma que no es posible acceder directamente a un nodo Beowulf desde otro nodo.	Todos los nodos suelen tener direcciones IP válidas, por lo que todos los nodos pueden estar accesibles desde otros nodos de la red de trabajo.
Servicios de los nodos	Los servicios y comandos para transferencia de información entre todos los nodos generalmente están inhabilitados. Únicamente se conectan con el nodo maestro.	Están disponibles todos los servicios y comandos habituales de red entre todos los nodos (FTP, Telnet, ssh, etc.)
Topología	Se diseña el modelo de paralelismo, se determinan como van a ser las comunicaciones entre los nodos y finalmente se implementan físicamente, por lo que se evita el hardware innecesario. En caso de que cambie la aplicación, se puede mantener el mismo modelo de paralelismo o rehacer la parte física de la red.	Puede tener cualquier topología de red. Para programar, la red NOW abstrae la topología.
Conocimientos del usuario	Habitualmente el usuario y el administrador son dos personas distintas. Sin embargo, el equipo de desarrollo del programa (o el usuario en caso de ser solo uno) tienen que saber sobre la topología de la red, como funcionan las aplicaciones paralelas, y en casos específicos saber montar y desmontar el hardware.	El usuario no es experto. No tiene por qué saber nada acerca de la máquina, ni como configurarla, ni como montarla.
Número de usuarios	Comúnmente sólo se tiene un usuario que va a usar el cluster para resolver un determinado tipo de problemas. Es común que el usuario sea al mismo tiempo quien administra el cluster y opera las máquinas.	Suele tener un número determinado de usuarios que forman un grupo heterogéneo, con una cuenta por usuario, un administrador y un conjunto de problemas indeterminados que cada usuario puede resolver

En esta Tesis se implementó un cluster que sigue la filosofía de construcción e instalación del tipo Beowulf, pero sin llegar a tener estrictamente las características mostradas en la Tabla 2.3. La descripción del cluster Beowulf utilizado en esta Tesis se expone en la Sección 4.2.3. en el Capítulo 4.

2.4.2.4 Cluster tipo MOSIX

MOSIX es una tecnología basada en Linux que permite realizar balance de carga para procesos particulares en un cluster [MOSIX 1998]. Este cluster trabaja como un cluster de alto rendimiento en el sentido de que está diseñado para tomar ventaja del hardware más rápido disponible en el cluster para cualquier tarea que le sea asignada; esto lo realiza balanceando la carga de las tareas en varias máquinas. La arquitectura MOSIX está basada en la arquitectura MIMD de tipo multicomputador, aunque se comporta como un sistema SMP (basado en memoria compartida) con tantos procesadores como nodos existan en el cluster. El espacio común de procesos y direcciones proporciona la idea de la existencia de una red local de transmisión de información entre nodos, emulando así una computadora de grandes dimensiones de procesamiento.

MOSIX se basa en un conjunto de parches aplicados al kernel de Linux que se asignan a todo el grupo de nodos un espacio de direcciones y de procesos común, gracias al cual los procesos migran de un nodo a otro con el fin de equilibrar favorablemente la carga del sistema global. Este esquema permite un mejor aprovechamiento de procesadores y memoria mediante la utilización de un *algoritmo de planificación adaptativo* desarrollado en la Universidad de Jerusalém. La migración de un proceso entre un nodo y otro con menor carga de trabajo se hace de forma totalmente transparente al usuario y al programador. Al igual que la arquitectura Beowulf, MOSIX tiene nodos conectados mediante una topología o red de comunicación. La programación de aplicaciones para un cluster MOSIX puede ser realizada mediante las herramientas típicamente utilizadas en los desarrollos de Beowulf, como es PVM y MPI.

Otra forma eficiente se programar aplicaciones para el cluster MOSIX es simplemente creando procesos hijos y continuar trabajando, con la técnica "*fork and forget*", llamada así por los propios creadores de MOSIX. La función "*fork and forget*" permite al programador de aplicaciones paralelas la designación de procesadores así como la paralelización del código fuente. Con simples llamadas a *fork* solventarán ampliamente el trabajo. MOSIX se encarga de otorgar el resto de decisiones por el programador y optimizarlas lo más posible. Ni el programador ni el usuario que ejecuta la aplicación se ocupan del cluster, simplemente lo verán como si se tratara de una computadora única. La intención del presente trabajo de Tesis no pretende abundar en la técnica "*fork and forget*" debido a que no se explotará ésta como plataforma de paralelización. El cluster MOSIX permite un trabajo muy seguro frente a una caída de procesadores o por eliminación de nodos. Permite además una mejora en el rendimiento gracias a la posibilidad de añadir nodos en el momento en que una aplicación se este ejecutando.

2.5 CONCLUSIONES

En este Capítulo se mencionaron las necesidades del cómputo de alto desempeño en la solución de sistemas eléctricos. Se describieron las características del procesamiento en paralelo, sus variantes, ventajas, y limitaciones. Se indicaron las clasificaciones comunes para sistemas paralelos y las características técnicas del hardware diseñando expresamente para realizar operaciones en paralelo, como son modelos SMP y MMP, además de los procesadores matriciales y vectoriales. Se describió el concepto de cluster como una clase de máquina paralela, y se mencionaron las características generales de los clusters tipo Beowulf y Mosix.

CAPITULO 3

TÉCNICAS DE PROCESAMIENTO EN PARALELO

3.1 FACTORES DE DESEMPEÑO DEL PROCESAMIENTO EN PARALELO

En todo sistema paralelo existen parámetros de medición que indican el desempeño de los mismos. Los más importantes son la eficiencia, el rendimiento del sistema, el grado de paralelismo, el tiempo de ejecución y el factor de mejora del rendimiento conocido como *Speed up*.

En el presente Capítulo se abordará la Ley de Amdahl, que demuestra por qué el rendimiento no se incrementa al aumentar hasta cierto número de procesadores en un sistema. Su aportación ha motivado al desarrollo de compiladores paralelos tendientes a reducir el cuello de botella secuencial a fin de mejorar el rendimiento de los sistemas paralelos. Además, se explicará por qué la correcta división del software, es decir, la granularidad de la aplicación, juega un papel importante en el rendimiento final del sistema. Todas las definiciones mostradas en el presente Capítulo fueron consultadas de [Torres 2003].

3.1.1 Factores de desempeño relativos al sistema

La *eficiencia* $E(n)$ mide la porción útil del trabajo total realizado por n procesadores. En el cálculo de la eficiencia de un sistema se consideran básicamente tres parámetros: el número total de operaciones elementales realizadas por un sistema $O(n)$, el número n de elementos de proceso de dicho sistema, y el tiempo de ejecución en pasos unitarios de tiempo $T(n)$.

Con los parámetros anteriores, se deduce que, para un sistema con un único procesador el tiempo de ejecución es $T(1) = O(1)$.

En sistemas con varios procesadores, si los n procesadores realizan más de una operación por unidad de tiempo, es decir $n \geq 2$, el tiempo de ejecución será $T(n) < O(n)$.

El *factor de mejora del rendimiento*, denominado también *Speed up*, indica el grado de ganancia de velocidad de una computación paralela. Se define como:

$$S_n = \frac{T(1)}{T(n)} \qquad (3.1)$$

Existen tres modelos de medición del Speed up [Torres 2003]:

- La Ley de Amdahl (1967), que se basa en una carga de trabajo fija o problema de tamaño fijo.
- La Ley de Gustafson (1987), que se aplica a problemas escalables, donde el tamaño del problema se incrementa al aumentar el tamaño de la máquina o se dispone de un tiempo fijo para realizar una determinada tarea.
- El modelo de mejora del rendimiento de Sun y Ni (1993), aplicado a problemas escalables limitados por la capacidad de la memoria.

En este trabajo de Tesis se abordará solamente el modelo de la Ley de Amdahl.

Eficiencia

Para un sistema con *n* procesadores, la *eficiencia* está definida por la ecuación:

$$E(n) = \frac{S_n}{n} = \frac{T(1)}{nT(n)} \tag{3.2}$$

La eficiencia es una comparación del grado de Speed up conseguido frente al valor máximo. Dado que el Speed up es $1 \leq S_n \leq n$, La eficiencia será $1/n \leq E(n) \leq 1$. Entonces:

- Cuando $E(n) \rightarrow 0$ se obtiene la eficiencia más baja, correspondiente para el caso en que todo un programa se ejecuta en un único procesador, de forma completamente secuencial.
- Cuando $E(n) = 1$ se obtiene la eficiencia máxima posible, y resulta cuando todos los procesadores se utilizan completamente durante todo el periodo de ejecución.

Redundancia

La *redundancia* en un cálculo paralelo que se define en correspondencia al número de operaciones elementales $O(n)$ y $O(1)$, e indica la proporción entre el paralelismo software y hardware del sistema. La ecuación de la redundancia es:

$$R(n) = \frac{O(n)}{O(1)} \tag{3.3}$$

donde $1 \leq R(n) \leq n$.

En términos generales la redundancia $R(n)$ mide el grado del incremento de la carga.

Escalabilidad

Se dice que un sistema es *escalable* para un determinado rango de procesadores [1..*n*], si la eficiencia $E(n)$ del sistema se mantiene constante y cercana a la unidad en todo ese rango. Por lo general, todos los sistemas llegan a un determinado número de procesadores a partir del cual la eficiencia comenzará a disminuir notablemente.

Ampliabilidad

Un sistema es ampliable si físicamente se le pueden adicionar más módulos hardware (más procesadores, memorias, tarjetas de entrada/salida, etc.). Un sistema puede ampliarse con más procesadores pero no necesariamente su rendimiento va a aumentar de forma proporcional. Es importante no confundir la escalabilidad con la ampliabilidad: un sistema puede ser ampliable, pero no significa que sea escalable (es decir, que mantenga su eficiencia).

Calidad del paralelismo

La *calidad* $Q(n)$ combina el efecto del Speed up, la eficiencia y la redundancia en una única expresión para indicar el mérito relativo de un cálculo paralelo sobre un sistema. La calidad de un cálculo paralelo es directamente proporcional al Speed up y a la eficiencia, e inversamente proporcional a la redundancia. Así, se observa que:

$$Q(n) = \frac{S_n E(n)}{R(n)} = \frac{T^3(1)}{nT^2(n)O(n)} \tag{3.4}$$

Dado que $E(n)$ es siempre una fracción y $R(n)$ es un número entre 1 y n, la calidad $Q(n)$ está siempre limitada por el Speed up.

3.1.2 Ley de Amdahl

En 1967, Gene Amdahl se basó en la definición del *Speed up armónico medio ponderado* para determinar un factor de mejora del rendimiento que relaciona la parte secuencial y la parte paralela de un problema dado. Al resultado que obtuvo se le conoce como *Ley de Amdahl* y es la base para el análisis de desempeño de los sistemas paralelos.

Amdahl supuso un sistema que trabaja bajo dos modos: a) secuencial puro, y b) usando n procesadores. Consideró una carga fija W_i, que también está compuesta por una parte secuencial W_1 y una parte paralela W_n. La cantidad total de trabajo T_n (carga) = W_1 + W_n. Bajo estas consideraciones, Amdahl encontró el Speed up por la ecuación:

$$S_n = \frac{W_1 + W_n}{W_1 + \frac{W_n}{n}} \tag{3.5}$$

Posteriormente, determinó que $W_1 = \alpha$ y $W_n = 1 - \alpha$, siendo α la fracción serie del programa (también llamada fracción del cuello de botella secuencial) y $1 - \alpha$ la fracción paralelizable. Sustituyendo en (3.13) se obtuvo:

$$S_n = \frac{\alpha + (1-\alpha)}{\alpha + \left(\frac{1-\alpha}{n}\right)} = \frac{n}{n\alpha + (1-\alpha)}$$

$$S_n = \frac{n}{1+\alpha(n-1)} \tag{3.6}$$

nótese que cuando $n \to \infty$ entonces $S_n \to 1/\alpha$.

La ecuación (3.6) describe la Ley de Amdahl, la cual dice:

"*Independientemente del número de procesadores que se emplee, existe un límite superior del Speed up debido a la parte serie de todo programa*".

En la Figura 3.1 se han trazado las curvas correspondientes a (3.6) para 4 valores de α. El Speed up ideal se obtiene para $\alpha = 0$, es decir, el caso en que no hay parte serie a ejecutar y todo el código es paralelizable.

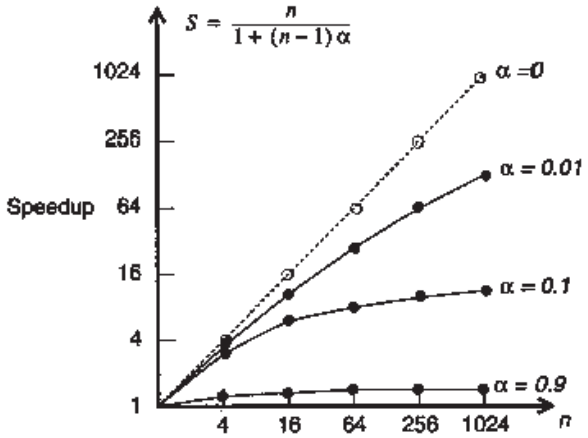


Figura 3.1 Mejora del rendimiento para diferentes valores de α.

La Ley de Amdahl se ilustra también en la Figura 3.2. Cuando el número de procesadores aumenta, la carga ejecutada en cada procesador decrece. Sin embargo, la cantidad total de trabajo ($W_1 + W_n$) se mantiene constante como se muestra en la Figura 3.2a. En la Figura 3.2b, el tiempo total de ejecución decrece porque $T_n = W_n/n$. Finalmente, el término secuencial domina el rendimiento porque $T_n \to 0$ al hacer n muy grande siendo T_1 constante.

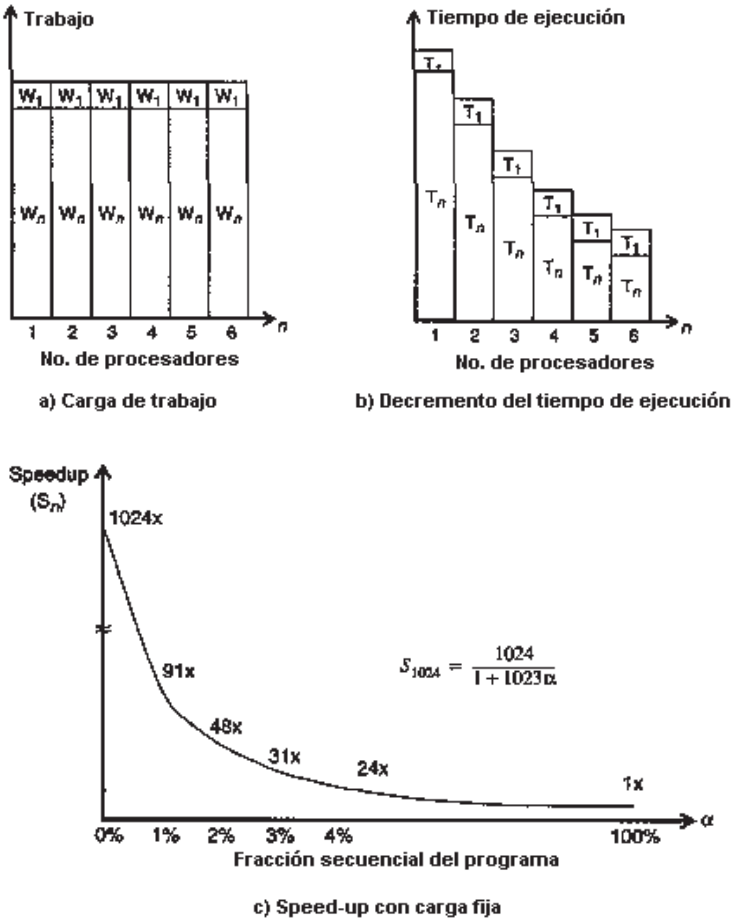


Figura 3.2 Modelo del Speed up de carga fija y la ley de Amdahl.

Cuello de botella secuencial.

La Figura 3.2c muestra una gráfica de la ley de Amdahl para diferentes valores de $0 \le \alpha \le 1$. El máximo Speed up, $S_n = n$, se obtiene para $\alpha = 0$. El mínimo Speed up, $S_n = 1$, se obtiene para $\alpha = 1$. Cuando $n \rightarrow \infty$, el valor límite es $S_\infty = 1/\alpha$. Esto implica que el Speed up está acotado superiormente por $1/\alpha$. independientemente del tamaño de la máquina.

La curva del Speed up en la Figura 3.2c cae rápidamente al aumentar α, lo que significa que con un pequeño porcentaje de código secuencial, el rendimiento total no puede ser superior a $1/\alpha$. A este α se le denomina *cuello de botella secuencial* de un programa.

El problema de un cuello de botella secuencial no puede resolverse incrementando el número de procesadores del sistema, puesto que el problema real está en la existencia de una fracción secuencial (*s*) del código.

3.1.3 Factores de desempeño relativos a la granularidad

3.1.3.1 Sobrecarga

La *sobrecarga en el sistema* es el fenómeno que ocurre cuando, al aumentar el número de procesadores, se presentan latencias de comunicaciones debidas a retrasos en el acceso a la memoria, latencias de comunicaciones a través de un bus o red, exceso de carga sobre el sistema operativo, retrasos causados por las interrupciones etc.

La *sobrecarga* se expresa como $L(n)$, y es la suma de todas las sobrecargas en un sistema con n procesadores. Introduciendo este parámetro, la ecuación del factor de mejora del rendimiento o Speed up sería:

$$S_n = \frac{T(1)}{T(n) + L(n)} \tag{3.7}$$

donde $L(n)$ depende siempre de la aplicación y de la máquina.

Siendo que es muy difícil obtener una expresión generalizada para $L(n)$, es muy común que en los cálculos de rendimiento se suponga que $L(n) = 0$.

La experiencia indica que sistemas con pocos procesadores, de 4 hasta 16, logran obtener una eficiencia bastante alta. Un mayor número de procesadores provocará que la eficiencia se reduzca drásticamente por efectos de sobrecarga [Pardo 2002].

3.1.3.2 Rendimiento

El *rendimiento de un sistema* de n procesadores depende de la relación entre la capacidad de ejecución del sistema y la sobrecarga producida por las comunicaciones presente en el sistema. La relación de rendimiento se calcula por la expresión [Torres 2003]:

$$\text{rendimiento del sistema} = \frac{R}{C} \tag{3.8}$$

donde R es una unidad de ejecución (con unidades de tiempo o instrucciones por segundo), y C es la sobrecarga debida a las comunicaciones producidas por R.

3.1.3.3 Granularidad del sistema

La *granularidad del sistema* es un parámetro que indica qué tanto se ha dividido el sistema (en “trozos”), y está determinado justamente por el factor R/C. Un sistema puede tener dos variantes de granularidad:

- Sistema de grano grueso: Tiene un factor *R/C* relativamente grande debido a que los trozos *R* son grandes y producen un costo de comunicaciones pequeño.
- Sistema de grano fino: Tiene un factor *R/C* pequeño puesto que los trozos *R* en que se ha dividido el problema son muchos y pequeños.

Calcular el factor *R/C* aporta también información para saber si es recomendable paralelizar o no un problema:

- Si *R/C* es un número pequeño, indica que no resulta provechoso paralelizar porque existe mucha sobrecarga (C es muy grande).
- Si *R/C* es un número muy grande indica que se obtiene beneficio al paralelizar, puesto que la sobrecarga que aparece es pequeña.

Por lo general, el factor *R/C* da como resultado un valor alto cuando el problema se divide en trozos grandes, ya que entonces las comunicaciones serán pequeñas comparativamente.

3.1.3.4 Tiempo de ejecución

Supóngase que cierta aplicación tiene *M* tareas a realizar, las cuales se pueden llevar a cabo de forma paralela. El objetivo está en ejecutar estas *M* tareas en un sistema con *n* procesadores en el menor tiempo posible.

Para hacer el análisis de tiempo mínimo de ejecución se toman como punto de partida las siguientes suposiciones:

- Que cada tarea se ejecuta en *R* unidades de tiempo.
- Que cada tarea de un procesador se comunica con todas las tareas del resto de procesadores con un costo de sobrecarga de *C* unidades de tiempo.
- El costo de comunicaciones con las tareas en el mismo procesador es cero.

El *tiempo de ejecución* esta compuesto por dos términos:

1. Costo de ejecución de las tareas (función de *R*).
2. Sobrecarga por las comunicaciones (función de *C*).

Existen dos posibilidades críticas para repartir las tareas entre los procesadores de un sistema:

a) Asignar todas las tareas a un único procesador.
b) Repartir por igual las tareas entre todos los procesadores.

En ambos casos se presentan situaciones donde un procesador tiene *k* tareas mientras que los otros tiene (*M* – *k*) tareas, donde *k* puede ser cualquier reparto entre 0 y *M*. La expresión para calcular el tiempo de ejecución de un sistema de dos procesadores es la siguiente:

$$T_e = R_{\max}\{M-k,k\} + C(M-k)k \tag{3.9}$$

donde: $R_{\max}\{M-k, k\}$ es el término que indica el tiempo propio de ejecución de las tareas y es lineal con respecto a *k*, y $C(M-k)k$ es el término que indica la sobrecarga, y crece cuadráticamente con respecto a *k*.

Para el caso de un sistema con *n* procesadores se supondrá que cada uno ejecuta k_i tareas, de manera que:

$$M = \sum_{i=1}^{n} k_i \tag{3.10}$$

Generalizando (3.9), se obtiene la expresión para el tiempo total de ejecución con *n* procesadores:

$$T_e = R_{\max}\{k_i\} + \frac{C}{2}\sum_{i=1}^{n} k_i(M-k_i)$$

$$T_e = R_{\max}\{k_i\} + \frac{C}{2}\left(M^2 - \sum_{i=1}^{n} k_i^2\right) \tag{3.11}$$

El *tiempo mínimo* se obtiene cuando las tareas se reparten por igual [Pardo 2002]. Para el caso del reparto igualitario de tareas, éstas se deben repartir dándole a cada procesador un número [*M/n*] de tareas hasta que queden menos de [*M/n*] tareas que repartir entre de los procesadores restantes, con la posibilidad de que algún(os) procesador(es) no reciba(n) ninguna tarea.

Él único reparto posible que garantiza el tiempo mínimo (en el caso del reparto equivalente) será cuando *p* procesadores tengan [*M/n*] tareas, a otro se le asignen *M*–[*M/n*]*p* tareas, y al resto de procesadores no se asigne ninguna tarea [Pardo 2002].

3.2 INTERFASE PARA PASO DE MENSAJES

El paso de mensajes es un modelo de comunicación ampliamente usado en computación paralela. En años recientes se ha logrado desarrollar aplicaciones importantes apoyadas en este paradigma, demostrando que es posible implementar sistemas basados en el pase de mensajes de una manera eficiente y portable. Una explicación más amplia del paso de mensajes se presenta en la Sección 4.2 del Capítulo 4 de esta Tesis. El contenido de la presente Sección 3.2 ha sido consultado de los manuales, tutoriales y documentos publicados en [LAM-MPI 2003].

3.2.1 Introducción

MPI o *Message Passing Interface* es un estándar desarrollado para la implementación de sistemas de paso de mensajes diseñado por un grupo de investigadores de la industria y la academia. MPI actúa sobre una amplia variedad de computadoras paralelas, de tal forma que los códigos son portables. Su diseño esta inspirado en máquinas con una arquitectura de memoria distribuida, en donde cada procesador es propietario de cierta memoria y la única forma de intercambiar información se logra a través de mensajes y una red de interconexión.

La plataforma MPI es de dominio público, por lo que existen varias implementaciones de MPI que han sido desarrolladas principalmente en universidades del extranjero. Para esta Tesis se eligió LAM (*Local Area Multicomputer*), que es un entorno de programación MPI y un sistema de desarrollo sobre redes de computadores heterogéneas. Esta implementación se desarrolla y mantiene en el Centro de Supercomputación de Ohio. E.U., y se puede conseguir vía Internet accediendo a la página web http://www.lam-mpi.org y descargando los archivos de instalación.

3.2.1.1 Características generales de MPI

MPI es un estándar conformado por alrededor de 129 funciones, muchas de la cuales tienen numerosas variantes y parámetros que le permiten lograr la portabilidad a través de diferentes máquinas. Si bien MPI no es un lenguaje de programación como tal, si permite ejecutar, de manera transparente, aplicaciones sobre sistemas paralelos heterogéneos. A continuación se enlistan las características de MPI:

- Permite que las aplicaciones puedan usarse en entornos heterogéneos.
- Soporta conexiones de la interfase con lenguajes *C*, *C++* y Fortran 77, con una semántica independiente del lenguaje.
- Define una interfase que puede implementarse en diferentes plataformas sin tener que hacer cambios significativos en el código.
- Proporciona una interfase de comunicación eficiente y confiable.
- No difiere significativamente de algunas implementaciones tales como PVM.

La flexibilidad de MPI permite escribir programas paralelos usando sólo 6 funciones básicas. En esté trabajo de Tesis se describen únicamente la funciones que se emplearon en la programación del algoritmo paralelo propuesto en el Capítulo 4.

De manera general, las características del estándar MPI se presentan en la Tabla 3.1. Para mayor información de la tarea que ejecutan las funciones y su sintaxis, se puede consultar [LAM-MPI 2003].

Tabla 3.1 Características del estándar MPI.

CARACTERISTICAS	DESCRIPCION
Generales	• Los comunicadores combinan procesamiento en contexto y por grupo para seguridad de los mensajes. • Seguridad en la manipulación de aplicaciones.
Manejo de entorno	MPI incluye funciones para: • Inicializar y finalizar. • Interacción con el entorno de ejecución. • Temporizadores y sincronizadores. • Control de errores.
Comunicación punto a punto	• Heterogeneidad para buffers estructurados y tipos de datos derivados. • Varios modos de comunicación: ‥Normal, con y sin bloqueo. ‥Síncrono. ‥Retardados, utilizando buffers.
Comunicaciones colectivas	• Capacidad de manipulación de operaciones colectivas con operaciones propias o definidas por el usuario. • Gran número de funciones para el movimiento de datos. • Subgrupos que pueden definirse directamente o en relación a la topología.
Topologías por procesos	• Incluye soporte para topologías virtuales (mallas y grafos).

3.2.2 Programación con MPI

Para que un programa escrito en código de MPI sea portátil, debe ser capaz de ejecutarse en cualquier sistema que tenga instalado el estándar MPI. No obstante, Algunas implementaciones pueden requerir cierta configuración previa del sistema antes de poder ejecutar llamadas a funciones de MPI. Por ello, el estándar incluye la función de inicialización *MPI::Init* y y la función *MPI::Finalize* que permite limpiar todos los estados MPI. Las indicaciones para escribir código en MPI son:

- La instrucción `#include <mpi.h>` es obligatoria en todo programa MPI ya que proporciona los tipos y definiciones básicas del estándar.
- Todas las llamadas a funciones que no forman parte del estándar MPI se ejecutarán de manera local.
- En el lenguaje C, todas las funciones MPI utilizan el prefijo *MPI_*., luego del prefijo se coloca el nombre de la función con la inicial en mayúsculas. En las funciones de C++ la sintaxis difiere debido a que se trata con clases y objetos. Generalmente se hace el llamado de la clase correspondiente y sus funciones miembro, por ejemplo *MPI::*.
- Los argumentos `argc` y `argv` en *MPI::Init* sólo son requeridos para lenguaje C y C++.
- Una vez que *MPI::Finalize* es invocada, ninguna otra función (ni siquiera *MPI::Init*) puede ser llamada, por lo que el usuario debe asegurarse de que todas las acciones pendientes sean completadas antes de llamar a *MPI::Finalize.*

En la Figura 3.3 se muestra un programa de ejemplo escrito en C, en donde cada proceso imprime el mensaje “Hola Mundo.c”.

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

int main(int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );     // Inicializa la ejecucion del entorno MPI
    printf("Hola al mundo\n");
    MPI_Finalize();               // Finaliza la ejecucion del entorno  MPI

    return 0;
}
```

Figura 3.3 Programa *Hola mundo.c.*

3.2.2.1 Compilación y ejecución

Compilación

Cuando se ejecuta un programa sencillo, se usa el formato de compilación tradiciona. LAM proporcionan los comandos *mpicc*, *mpiCC* y *mpif77* para compilar programas en C, C++ y Fortran respectivamente. Un ejemplo de comandos de compilación se muestra en la Figura 3.4.

```
mpicc holamundo.cc –o holamundo
mpiCC holamundo.c –o holamundo
```

Figura 3.4 Comandos de compilación para C y C++.

Para proyectos grandes, es más recomendable usar un *Makefile* para efectos de compilación.

Ejecución

La mayoría de las versiones MPI coinciden en el uso de un programa especial denominado *mpirun*, para ejecutar los programas (aunque éste no es parte del estándar MPI). En la Figura 3.5 se muestra el comando para ejecutar programas MPI. El parámetro *np* le indica al programa *mpirun* que debe crear dos instancias de “*holamundo.c”*; mientras que *machinefile* indica el nombre del archivo (*maquinas*) que contiene la descripción de las máquinas que serán utilizadas para correr los procesos.

```
mpirun –np 2 –machinefile maquinas holamundo
```

Figura 3.5 Ejecutando el programa *holamundo.c*.

Adicionalmente, *mpirun* tiene un conjunto de opciones que permiten al usuario ordenar la ejecución de un programa bajo determinadas condiciones. Estas opciones pueden verse en la Figura 3.6.

```
Man mpirun   ó   mpirun help
```

Figura 3.6 Obteniendo ayuda sobre *mpirun.*

3.2.2.2 Información del entorno

MPI maneja un sistema de memoria para almacenamiento de mensajes y representaciones de objetos por medio de *grupos*, *comunicadores*, *identificadores* y *tipos de datos*. El usuario no puede acceder directamente a esta memoria y los objetos ahí almacenados.

Grupos

En MPI se considera a un *grupo* como un conjunto ordenado de procesos, y cada proceso se identifica por medio de un número entero denominado *rango* (*rank*). Las funciones para grupos en lenguaje C son *MPI_GROUP_EMPTY* y *MPI_GROUP_NULL.*

Comunicador

El intercambio de información (comunicaciones) entre grupos se hace a través del elemento fundamental de MPI: el *comunicador* (*communicator*), que es un objeto opaco con un número de atributos y reglas que gobiernan su creación, uso y destrucción.

Un comunicador especifica un dominio de comunicación, es decir, identifica un grupo de procesos y el contexto en el cual se llevará a cabo una operación. Proporciona también mecanismos para identificar subconjuntos de procesos para el desarrollo de programas modulares, garantizando que los mensajes concebidos para diferentes propósitos no sean confundidos. Hay dos tipos de comunicadores:

- *Intracomunicador* (*intracommunicator*): usado para las comunicaciones entre los procesos de un mismo grupo.
- *Intercomunicador* (*intercommunicator*): usado para comunicaciones punto a punto entre procesos de diferentes grupos.

El comunicador por omisión en MPI es el denominado *MPI::COMM_WORLD*, mismo que se emplea en el código que se desarrolló en esta Tesis.

Identificadores

En un programa paralelo es importante conocer en todo momento cuántos procesos hay, el *identificador* del cada proceso, y qué proceso(s) está(n) ejecutándose. Las funciones que reportan dicha información son:

- *MPI::COMM_WORLD.Get_size()* devuelve un entero con el valor del número de procesos.

- El identificador de cada proceso lo proporciona la función *MPI::COMM_WORLD.Get_rank()*, y es un número entre cero y el total de procesos menos uno.
- El proceso que esta "corriendo" lo indica la función *MPI::COMM_WORLD. Get_processor_name().*

3.2.2.3 Medición del tiempo de proceso

Medir el tiempo de ejecución del programa propuesto en esta tesis es importante para calcular el Speed up y la eficiencia. MPI ofrece la función *MPI::Wtime,* y regresa el número de segundos transcurridos a partir de cierto tiempo pasado.

3.2.2.4 Comunicación punto a punto

La *comunicación punto a punto* en MPI es el mecanismo básico de transferencia de datos entre un par de procesos, uno que envía y otro que recibe. MPI proporciona un conjunto de funciones de envío y recepción de mensajes que permiten la comunicación de datos de cierto tipo con una etiqueta (*tag*) asociada.

La identificación del contenido de los mensajes se hace a través del tipo de dato. Esta información es importante para mantener la heterogeneidad, es decir, si se envía un dato tipo entero de una arquitectura a otra, la que recibe debe hacer las conversiones necesarias para identificar al dato recibido como un entero.

El identificador o *tag* del mensaje permite, del lado del receptor, seleccionar un mensaje en particular o recibir cualquier mensaje.

3.2.2.5 Envío y recepción de mensajes

En el envío y recepción de un mensaje es necesario intercambiar información entre los procesos, de manera que se conozca a qué proceso(s) se envían los datos, qué es lo que se envía, y cómo el receptor va a identificar el mensaje. Los aspectos que MPI proporciona para el envío de datos y lograr la portabilidad son:

- Especificación de los datos a enviar con tres campos: dirección de inicio, tipo de dato y contador, donde el tipo de dato puede ser:
  - Elemental: Todos los tipos de datos de C y C++.
  - Arreglos continuos de tipos de datos.
  - Bloques dispersos de tipos de datos.
  - Arreglos indexados de bloques de tipos de datos.
  - Estructuras generales.
- Las especificaciones de tipos de datos permiten comunicaciones heterogéneas.

La equivalencia de los tipos de datos elementales que reconoce MPI y *C* se indican en la Tabla 3.2 (*C++* reconoce las mismas equivalencias de *C*).

Tabla 3.2 Equivalencias de tipos de datos ente MPI y C.

TIPO DE DATO MPI	TIPO DE DATO C
MPI_CHAR	SIGNED CHAR
MPI_SHORT	SIGNED SHORT INT
MPI_INT	SIGNED INT
MPI_LONG	SIGNED LONG INT
MPI_UNSIGNED_CHAR	UNSIGNED CHAR
MPI_UNSIGNED	UNSIGNED INT
MPI_UNSIGNED_LONG	UNSIGNED LONG INT
MPI_FLOAT	FLOAT
MPI_DOUBLE	DOUBLE
MPI_LONG_DOUBLE	LONG DOUBLE
MPI_BYTE	--
MPI_PACKED	--
NOTA: El tipo *MPI_BYTE* consiste de un byte (8 dígitos binarios). El tipo *MPI_PACKED* se emplea en el empaquetamiento de datos.	

3.2.3 Comunicación

Los programas que usan el modelo de paso de mensajes por lo general son no determinísticos, es decir, el orden de llegada de los mensajes enviados por dos procesos *p* y *q*, a un tercer proceso *r*, no esta definido. Sin embargo, MPI garantiza que los mensajes enviados de un proceso *p* a otro proceso *q* llegan en el orden que fueron enviados.

Las funciones *MPI::Send* y *MPI::Recv* son *funciones con bloqueo*, esto significa que una llamada a *send* permanecerá bloqueada hasta que el mensaje haya sido enviado y el buffer de envío pueda ser re-utilizado. Similarmente, la función de recepción *recv* se bloquea hasta que el buffer de recepción contenga el mensaje esperado. Las funciones que proporciona MPI para la transferencia sin bloqueo se pueden consultar en [LAM-MPI 2003].

3.2.3.1 Comunicaciones colectivas

Existen aplicaciones donde un único proceso requiere recibir datos de otros procesos, o al contrario, enviar un conjunto de datos a varios procesos. En estos casos se utilizan *comunicaciones colectivas*.

Las comunicaciones colectivas permiten la transferencia de datos entre todos los procesos que pertenecen a un grupo específico. En comunicaciones colectivas no se utilizan etiquetas para los mensaje sino identificadores de los grupos. A estos identificadores se les llama *comunicadores*. Las operaciones colectivas de MPI pueden ser clasificadas en tres clases:

- *Sincronización*: Barreras para sincronizar.
- *Transferencia de datos*: Operaciones para difundir, recolectar y esparcir.
- *Cálculos colectivos*: Operaciones para reducción global, tales como suma, máximo, mínimo o cualquier función definida por el usuario.

Todas las funciones, excepto la de difusión, tienen dos variantes:

1. Todos los datos transferidos son homogéneos (del mismo tamaño).
2. Cada ítem de datos puede tener un tamaño distinto (vector).

Algunas de estas funciones, como las de difusión y recolección, tienen un proceso de origen o un proceso receptor. A este proceso se le llama *raíz* (*root*). Las funciones colectivas para lenguaje C son *MPI_Barrier*, *MPI_Bcast*, *MPI_Gather*, *MPI_Scatter*, *MPI_Reduce*, y *MPI_Allreduce*. Los parámetros de las funciones colectivas para sincronización y difusión, y transferencia que se utilizaron en esta Tesis se indican en la Tabla 3.3.

Tabla 3.3 Funciones colectivas para sincronización, difusión y transferencia.

FUNCION COLECTIVA PARA SINCRONIZACION Y DIFUSION				
MPI::COMM_WORLD.Bcast(inbuf, incnt, intype, root, comm) *int MPI_Bcast(void *inbuf, int incnt, MPI_Datatype intype, int root)*	*ES*	inbuf	Dirección del *buffer* de recepción, o *buffer* de envío en *root.*	Difunde datos desde *root* a todos los procesos.
	E	incnt	Número de elementos en el *buffer* de envío.	
	E	intype	Tipo de dato de los elementos del *buffer* de envío.	
	E	root	rango del proceso *root.*	
FUNCION COLECTIVA DE TRANSFERENCIA				
MPI::COMM_WORLD.Gather(inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm) *int MPI_Gather(void *inbuf, int incnt, PI_Datatype intype, void *outbuf, int outcnt, MPI_Datatype outtype, int root)*	*ES*	inbuf	Dirección del *buffer* de entrada.	Función de transferencia colectiva de datos.
	E	incnt	Número de elementos a enviar a cada uno.	
	E	intype	Tipo de dato de los elementos del *buffer* de entrada.	
	S	outbuf	Dirección del *buffer* de salida.	
	E	outcnt	Número de elementos a recibir de cada uno.	
	E	outtype	Tipo de dato de los elementos del *buffer* de salida.	
	E	root	Rango del proceso *root.*	

MPI proporciona también variantes de las funciones anteriores, tales como: *Alltoall, Alltoallv, Gatherv, Allgather, Scatterv, Allreduce, Scan, ReduceScatter.* Todas éstas envían el resultado de la operación a los procesos participantes. Las versiones con el sufijo *v* permiten que los *trozos* a transferir tengan distintos tamaños. Para más información sobre las funciones y sus variantes consultar [LAM-MPI 2003].

3.2.3.2 Transferencia de datos no homogéneos

En algunas aplicaciones es necesario transferir datos no homogéneos o que no están en memoria contiguas. Lo ideal es tener un mecanismo que permita estas transferencias sin necesidad de utilizar grandes cantidades de mensajes. Para lograrlo, MPI proporciona dos esquemas:

1. El usuario puede definir tipos de *datos derivados* que pueden ser usados en las funciones de comunicación de MPI en vez de los tipos de datos básicos predefinidos.
2. El proceso que envía puede explícitamente *empaquetar* datos no contiguos en un buffer contiguo y luego enviarlo, mientras que el proceso receptor desempaqueta los datos de un buffer contiguo para almacenarlos en localizaciones no contiguas.

En [LAM-MPI 2003] se describen los parámetros y operación de las principales funciones que maniobran sobre grupos y permiten crear comunicadores en base a ellos, así como las funciones para el empaquetado y desempaquetado de mensajes.

3.3 LA MAQUINA VIRTUAL PARALELA

La Máquina Virtual Paralela o PVM (*Parallel Virtual Machine* por sus siglas en inglés), es una herramienta computacional que proporciona un marco unificado para desarrollar en forma eficiente programas paralelos utilizando el hardware existente [PVM 1994]. PVM permite que un grupo heterogéneo de computadoras pueda ser visto como una sola máquina virtual paralela, de tal manera que se logra el manejo transparentemente todos los mensajes de enrutamiento, la conversión de datos, y programación de tareas a través de una red de arquitecturas de computadoras diferentes.

3.3.1 Características generales de PVM

- PVM se puede conseguir como shareware en www.csm.ornl.gov/pvm.
- Costo. Si ya se cuenta con una red Unix, únicamente se instala el software de PVM, por lo que el costo de tener una supercomputadora o una red en paralelo es casi nulo.
- Tolerancia a fallas. Si por cualquier razón falla una de las máquinas o nodos, la aplicación podrá seguir funcionando sin problemas en el hardware disponible.
- Heterogeneidad. PVM abstrae al usuario de la topología de la red, la tecnología de la red, la cantidad de memoria en cada máquina y la forma como se almacenan los datos. PVM se puede representar como un conjunto de máquinas secuenciales heterogéneas que trabajan en paralelo.
- Recursos. Si bien PVM presenta una abstracción que realiza funciones paralelas, independientemente del hardware, la codificación para lograrlo tiene un costo en recursos (tiempo y memoria). Esta desventaja no se observa cuando las aplicaciones desarrolladas tiene un grado de acoplamiento bajo.
- Lenguaje. PVM proporciona funciones escritas en C, C++ y Fortran. Basta con incluir en el encabezado del programa la biblioteca adecuada para el lenguaje deseado.
- Limitantes. PVM no se recomienda en aplicaciones de paralelismo fuertemente acoplado, debido a que al incrementar la cantidad de transmisiones de datos implicados en un proceso, fallan las comunicaciones debido a colisiones en el buffer activo. Esta limitante se relaciona con el tipo de tecnología de la red, donde, por ejemplo, si se usa tecnología Ethernet, las aplicaciones de PVM no funcionaran debido a la cantidad de colisiones producidas.

3.3.1.1 Arquitectura

La arquitectura PVM se compone de dos partes:

1. Un *demonio*, llamado *pvmd3* (o *pvmd*), que se encuentra en todas las máquinas que componen la máquina virtual. Este demonio pvmd3 es el responsable de ejecutar los programas, administrar los mecanismos de comunicación entre máquinas, hacer la conversión automática de datos, y "ocultar" la red al programador.
   El demonio pvmd3 puede ser instalado por cualquier usuario válido en su directorio personal. Una vez que un usuario o el superusuario instaló PVM, todos los usuarios pueden hacer uso de esa instalación, siempre y cuando el directorio donde se instaló PVM tenga los permisos correspondientes. Cada usuario arrancará el demonio como cualquier otro programa; inmediatamente se ejecuta el código de PVM. El demonio se queda residente y realizando las funciones ya descritas.
2. Una biblioteca para desarrollo de aplicaciones. Contiene las rutinas para operar con los procesos, transmitir mensajes entre procesadores y alterar las propiedades de la máquina virtual. Toda aplicación se deberá enlazar a la biblioteca para poderse ejecutar después. Se tiene tres archivos de bibliotecas en la distribución estándar de PVM: *libpvm3.a* (biblioteca para C), *libgpvm3.a* (biblioteca para grupos) y *libfpvm3.a* (biblioteca para Fortran).

3.3.2 Paso de mensajes

Para la recepción de mensajes, PVM incluye primitivas *de bloqueos* y de *no bloqueo* o con un tiempo máximo de espera. Para la comunicación utiliza dos protocolos: UDP y TCP. UDP es un protocolo de comunicación no orientado a conexión, lo que significa que se pueden enviar dos mensajes y no necesariamente llegarán en el orden en que fueron enviados. TCP es un protocolo orientado a la conexión, por tanto, reordena los mensajes antes de pasarlos a la capa superior. Sin embargo, TCP tiene el inconveniente de establecer, para *n* host, un mínimo de $n(n-1)$ conexiones activas entre los host, provocando que se ocupen todos los puertos disponibles, aún cuando *n* sea un numero pequeño.

Para evitar que se "acaben" los puertos en un a conexión TCP, la comunicación se establece solamente entre tareas y el demonio, y se hacen empleando TCP. Con esto se logra que las comunicaciones locales tengan un tiempo mínimo de apertura y cierre de canal, liberando el puerto casi inmediatamente. Dado que las tareas no se "conectan" entre si, sino que únicamente se comunican directamente con su demonio, las conexiones activas vía TCP se reducen a *n*, mucho menor que $n(n-1)$. En la Figura 3.7 se muestra un ejemplo del diagrama de comunicación PVM.

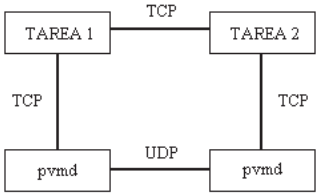


Figura 3.7 Diagrama comunicación PVM.

El demonio *pvmd* puede convertirse en un cuello de botella si todas las tareas tratar de hablar entre sí a través de él. Por eso, *pvmd* utiliza memoria asignada dinámicamente para guardar los paquetes de mensajes en su ruta entre tareas. Hasta que acepta recibir los paquetes de una tarea, éstos se acumulan en una estructura FIFO conocida como *cola de mensajes*. La cola de mensajes permite el paso mensajes de manera *no bloqueante*, es decir, si envía un mensaje no se tiene que esperar a que éste llegue a su destino, sino que solamente se espera a que sea puesto en la cola. La cola de mensajes asegura que los mensajes de una misma tarea llegarán en orden entre si. El inconveniente de la cola de mensajes de PVM es que ésta podría almacenar demasiados mensajes, hasta que no pueda conseguir más memoria. Si una aplicación está diseñada para que pueda mantener el envío de tareas, incluso cuando el receptor este apagado o haciendo otra cosa, se puede acabar con la memoria del sistema [PVM 1994].

3.3.3 Consola PVM

A fin de comprobar si PVM funciona correctamente, se inicia la consola *pvm* en el *host central* (host donde se inicia el entorno PVM). Esto se debe realizar con una cuenta de usuario normal. La línea de comando es:

```
Shell:~/$ pvm
```

Al ejecutar el programa *pvm*, se inicia el demonio *pvmd3* en el host central y se presenta la consola de control de PVM, que aparece como sigue:

```
pvm>
```

Esto indica que la máquina virtual está lista para aceptar comandos. En este momento, la máquina virtual consta únicamente de un host (el host central). A fin de anexar nodos a la máquina virtual para que se forme la máquina paralela y puedan participar en las tareas colectivas, se utiliza el comando *add*, indicando el nombre de los host que se van a agregar. Si el proceso es exitoso, se tendrá el despliegue que se muestra en la Figura 3.8.

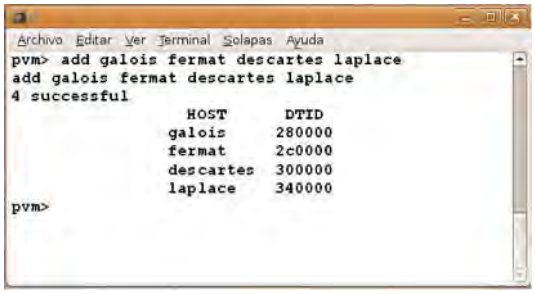


Figura 3.8 Comando *add*.

Si no desea escribir todos los nombres de los hosts cada vez, se puede crear una lista de los nombres de host en una sola línea, y almacenarla como un archivo *hostfile*. Bastará con ejecutar únicamente la declaración:

```
Shell:~/$ pvm hostfile.
```

3.3.4 Programación con PVM

Un programa en PVM es un conjunto de tareas que cooperan entre si. Las tareas van a intercambiar información empleando paso de mensajes. PVM, de forma transparente al usuario, oculta las transformaciones de tipos asociadas al paso de mensajes entre máquinas heterogéneas. Toda tarea en PVM puede incluir o eliminar máquinas, iniciar o parar otras tareas, mandar datos a otras tareas o sincronizarse con ellas.

PVM se apoya en una combinación de paralelismo de control (cada tarea realiza una función diferente) y paralelismo de datos (las tareas son las mismas, pero cada una resuelve sólo una parte de los datos). PVM podrá ejecutar tareas en paralelo ó sincronizar o intercambiar datos.

Cada una de las tareas en PVM se identifican mediante un número entero único en toda la máquina virtual, conocido como TID (*Task Identification Number*). Los mensajes son enviados y recibidos a partir de TIDs. PVM contiene varias rutinas que devuelven valores TID, de manera que el usuario pueda identificar las tareas.

Hay aplicaciones paralelas en las que se requiere hacer un conjunto de acciones sobre un conjunto de tareas. PVM incluye el concepto de *grupos*. Un grupo es un conjunto de tareas a las que se puede hacer referencia mediante un mismo código y un mismo identificador de grupo.

Cualquier tarea PVM puede unirse o dejar cualquier grupo en cualquier momento sin tener que informar a cualquier otra tarea de los grupos afectados. Asimismo, los grupos se superponen, y las tareas pueden difundir mensajes a los grupos de los que no son miembros. Para utilizar cualquiera de las funciones del grupo, un programa debe estar vinculado con la biblioteca *libgpvm3.a*.

3.3.4.1 Compilación y ejecución

Se requiere que la tarea a ser realizada por las máquinas que integran PVM se encuentre en todas ellas. Por tanto, se deben transferir los programas compilados a todas las máquinas esclavas. El procedimiento para compilar un programa de PVM es un tanto complejo si no se tiene la experiencia en la modificación de los archivos de sistema de Unix. Una manera sencilla es la que propone [Ramos 2007], consistente en generar un *scrip* al que llamó “comp”, y que tiene como objetivo compilar un programa de forma rápida, sin necesidad escribir línea por línea todos los parámetros que PVM requiere. El scrip modificado de [Ramos 2007] se muestra en la Figura 3.9.

```
#!/bin/sh
source=$1
g++ -g -o $PVM_BIN/$source $PVM_SOURCE/$source.c -lm -lpvm3 -w
```

Figura 3.9 Scrip “comp” para compilación en PVM.

Una vez que se ha escrito el scrip “comp”, los programas paralelos en PVM se pueden compilar escribiendo en la consola la declaración:

```
Shell:~/$ comp nombre_programa_fuente
```

y la ejecución del programa compilado se realiza de la forma tradicional:

```
Shell:~/$ ./nombre_programa_salida
```

3.3.4.2 Ejecución de un programa desde la consola PVM

Generalmente para ejecutar un programa desde la consola de PVM, se inicia una tarea madre desde uno de los host. La tarea se activa con el comando *spawn*, desde un host de la maquina virtual, que a su vez se activará con el comando *pvm*. Esta tarea se encargará de iniciar todas las demás tareas, bien desde la función *main()* o bien desde alguna subrutina invocada por ella. La declaración es:

```
pvm> spawn -> nombre_programa
```

Para activar nuevas tareas se emplea la función *pvm_spawn*, que devolverá un código de error, asociado a si pudo o no crearla, y el TID de la nueva tarea.

PVM tiene definidas clases de arquitecturas (hardware). Antes de mandar un dato a otra máquina, comprueba su clase de arquitectura. Si es la misma, no hace conversión de datos, lo cual incrementa el rendimiento. En caso de contrario, PVM emplea el protocolo XDR para codificar el mensaje [Plaza 2003].

En general, las funciones de PVM se agrupan en las categorías siguientes:

a) Control de procesos.
b) Operación de grupos.
c) Envío de mensajes.
d) Recepción de mensajes.
e) Buffers de mensaje.
f) Señalamiento.
g) Información.

Las funciones y sus parámetros, así como ejemplos de programas, se pueden consultar en los manuales en línea proporcionados en [PVM 1994].

3.4 CONCLUSIONES

En el presente Capítulo se mencionaron los principales factores que miden el desempeño relativo al cómputo paralelo, como son Speed up, eficiencia, granularidad y tiempo de ejecución. Se indicaron las características del paradigma de paso de mensajes. Se describieron los procesos de instalación, configuración y verificación de funcionamiento de la plataforma MPI para cómputo paralelo. Por último, describió la plataforma PVM para cómputo paralelo, indicando algunas de sus principales características.

CAPITULO 4

PARALELIZACIÓN DE ALGORITMOS PARA LA SOLUCIÓN DE SISTEMAS ELECTRICOS

4.1 ALGORITMOS PARALELOS

Dado un problema concreto, siempre habrá una serie de algoritmos aplicables para resolverlo. En ocasiones es suficiente encontrar una buena aproximación para resolver el problema, ya que un algoritmo puede encontrar rápidamente una solución, aunque ésta no sea necesariamente la óptima, pero sí es válida dentro de un cierto rango de error.

En la solución de un problema conviene elegir un algoritmo eficiente, es decir, aquel que utilice los menos recursos, que pueda ser escalable, que llegue a una solución válida y que el tiempo de procesamiento sea rápido. En problemas pequeños, casi todos los algoritmos tienen un comportamiento más o menos igual. La eficiencia del algoritmo se manifestará siempre en problemas grandes.

4.1.1 Diseño de algoritmos paralelos

El diseño de algoritmos paralelos involucra cuatro etapas [Foster 1995] que se indican en la Tabla 4.1; gráficamente, dichas etapas se representan en la Figura 4.1.

Tabla 4.1 Etapas del diseño de algoritmos paralelos.

ETAPA	DESCRIPCION
Partición	Las operaciones y los datos sobre los cuales se opera se descomponen en *tareas*. Se ignoran aspectos como el número de procesadores de la máquina a usar y se concentra la atención en explotar oportunidades de paralelismo
Comunicación	Se determina la comunicación requerida para coordinar las tareas. Se definen estructuras y algoritmos de comunicación
Agrupación	El resultado de las dos etapas anteriores es evaluado en términos de eficiencia y costos de implementación. De ser necesario, se agrupan tareas pequeñas en tareas más grandes
Asignación	Cada tarea es asignada a un procesador tratando de tener el máximo rendimiento de los procesadores y de reducir el costo de comunicación. La asignación puede ser estática (se establece antes de la ejecución del programa) o en tiempo de ejecución mediante algoritmos de balanceo de carga

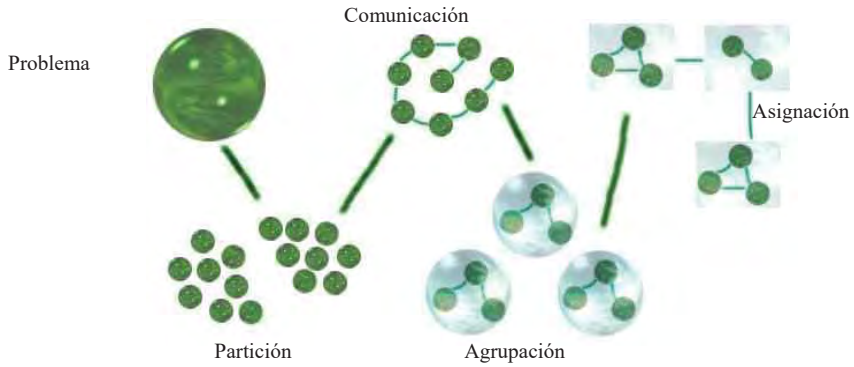


Figura 4.1 Etapas en el diseño de algoritmos paralelos.

4.1.1.1 Partición

En la etapa de partición se buscan oportunidades de paralelismo. Se subdivide el problema a una *granularidad fina*, entendiéndose por *grano* la medida del trabajo computacional a realizar. Una buena partición divide tanto los cálculos como los datos. Existen dos formas de realizar la descomposición [Foster 1995]:

1) **Descomposición del dominio**: se enfoca sobre los datos. Aquí se determina la partición apropiada de los datos y luego se trabaja en los cálculos asociados con ellos.
2) **Descomposición funcional**: primero se descomponen los cálculos u operaciones y luego se ocupa de los datos.

Ambas técnicas son complementarias y pueden ser aplicadas a diferentes componentes de un problema, inclusive al mismo problema para obtener algoritmos paralelos alternativos.

Al particionar se deben tener en cuenta los siguientes aspectos:

- Evitar operaciones y almacenamientos redundantes o de lo contrario el algoritmo puede ser no escalable (Sección 3.1.1) a problemas más grandes.
- Las tareas deben ser de tamaños equivalentes, ya que esto facilita el balanceo de la carga de los procesadores.
- El número de tareas debe ser proporcional al tamaño del problema. De esta forma el algoritmo será *escalable* (será capaz de resolver problemas más grandes cuando se tenga más disponibilidad de procesadores).
- Considerar otras alternativas de paralelismo. Si se hace en esta etapa, se pueden flexibilizar etapas subsecuentes.

En esta Tesis se emplea la descomposición del dominio, ya que se aplica un paralelismo de datos (Sección 2.2.1.2). Así mismo, se verificó que se cumpliera lo mejor posible con todos los aspectos anteriores.

4.1.1.2 Comunicación

Se denomina *fase de comunicación* a la etapa donde los datos son transferidos o compartidos entre tareas. La comunicación requerida por un algoritmo puede ser definida en dos fases:

a) Se definen los canales que conectan las tareas que requieren datos con las tareas que los poseen.
b) Se especifica la información o mensajes que deben ser enviado y recibidos en estos canales.

En la etapa de comunicación se deben tener en cuenta los siguientes aspectos:

- Todas las tareas deben efectuar aproximadamente el mismo número de operaciones de comunicación. De no ser así, es muy probable que el algoritmo no sea extensible a problemas mayores porque se producirán cuellos de botella.
- La comunicación entre tareas debe ser tan pequeña como sea posible.
- Las operaciones de comunicación deben poder ejecutarse concurrentemente.
- Los cálculos de diferentes tareas deben poder ejecutarse concurrentemente.

La forma de organizar la comunicación entre tareas varía dependiendo del tipo de arquitectura en que se implementará el algoritmo (memoria distribuida o memoria compartida). En esta Tesis se utiliza ambiente de memoria distribuida (la ejecución del programa paralelo propuesto se hizo en un cluster), y la fase de comunicación se realiza directamente a través de las librerías de paso de mensajes de MPI.

4.1.1.3 Agrupación

En esta etapa se revisa el algoritmo obtenido tratando de que éste se ejecute eficientemente sobre cierta arquitectura. En particular se considera si es útil agrupar tareas y si vale la pena repetir datos y/o cálculos. Por medio de la agrupación de tareas se puede reducir la cantidad de datos a enviar y así reducir el número de mensajes y el costo de comunicación.

La comunicación no sólo depende de la cantidad de información enviada, sino que además cada comunicación tiene un costo fijo de arranque. Si se reduce el número de mensajes, a pesar de que se envíe la misma cantidad de información, se puede disminuir el costo de comunicación. Por otro lado, se debe considerar también el costo de creación de tareas y el costo de *cambio de contexto* (*context switch*) en caso de que se asignen varias tareas a un mismo procesador. El cambio de contexto se produce cada vez que el procesador cambia de una tarea a otra, porque se deben cargar registros y otros datos de la nueva tarea que ejecutará. Los puntos principales que se deben considerar para la agrupación son:

- Verificar si la agrupación redujo los costos de comunicación.
- Si se han repetido cálculos y/o datos, se debe verificar que los beneficios son superiores a los costos.

- Verificar que las tareas resultantes tengan costos de cómputo y comunicación similares.
- Revisar si el número de tareas aumenta con el tamaño del problema.
- Si el agrupamiento ha reducido las oportunidades de ejecución concurrente, se debe verificar que aún se logre suficiente concurrencia o mejor considerar diseños alternativos.
- Analizar si es posible reducir aun más el número de tareas sin provocar desbalance de cargas.

4.1.1.4 Asignación

No existen todavía mecanismos generales de asignación de tareas para máquinas distribuidas, lo que resulta en una difícil problemática que debe tomarse en cuenta a la hora de diseñar algoritmos paralelos [Thomson 1992]. No obstante, se observa que la asignación de tareas puede ocurrir en dos formas:

- **Asignación estática**: las tareas son asignadas a un procesador al comienzo de la ejecución del algoritmo paralelo y se procesan ahí mismo hasta el final. La asignación estática (en ciertos casos) puede dar como resultado un tiempo de ejecución menor respecto al de asignaciones dinámicas y también puede reducir el costo de creación de procesos, sincronización y terminación.
- **Asignación dinámica**: se hacen cambios en la distribución de las tareas entre los procesadores durante el tiempo de ejecución con el fin de balancear la carga del sistema y reducir el tiempo de ejecución. Sin embargo, el costo de balanceo puede ser significativo y por ende incrementar el tiempo de ejecución.

El balanceo de carga en los algoritmos se efectúa de tres posibles formas o métodos [Thomson 1992], como se indica en la Tabla 4.2.

Tabla 4.2 Métodos para balanceo de carga.

METODO	DESCRIPCION
Balanceo centralizado	Un procesador ejecuta el algoritmo y mantiene el estado global del sistema. Este método no es aplicable en problemas muy grandes porque el nodo encargado del balanceo se convierte en un cuello de botella.
Balanceo completamente distribuido	Cada procesador mantiene su propia información del sistema intercambiando ésta con sus vecinos para poder hacer cambios locales. Aunque el costo de balanceo se reduce, no es un método óptimo debido a que sólo se dispone de información parcial.
Balanceo semi-distribuido	Divide los procesadores por regiones, cada una con un algoritmo centralizado local, mientras otro algoritmo balancea la carga entre las regiones. El balanceo puede ser iniciado por *envío* o *recepción*. Si el balanceo es iniciado por envío, un procesador con mucha carga envía trabajo a otros. Si el balanceo es iniciado por recepción, un procesador con poca carga solicita trabajo de otros.

Los puntos que se revisan en la etapa de asignación son:

- Considerar algoritmos con un número estático de tareas y algoritmos de creación dinámica de tareas.
- Si se usan algoritmos centralizados de balanceo, habrá que asegurarse de que no se producirá un cuello de botella.
- Evaluar los costos de las diferentes alternativas de balanceo dinámico, en caso de que se usen, y que su costo no sea mayor que los beneficios.

4.1.2 Técnica de divide y vencerás

Dado un problema a resolver planteando en términos de una entrada de tamaño *n*, la técnica de "*divide y vencerás"* en [Gálvez y Gonzáles 1993] se propone la división de la entrada en *k* subproblemas, donde:

$$1 < k < n \tag{4.1}$$

éstos *k* subproblemas se resuelven independientemente y después se combinan sus soluciones parciales para obtener la solución del problema original.

La técnica divide y vencerás se usa sólo en el caso en que los subproblemas sean de la misma clase del problema original. Esta restricción permite una formulación y resolución recursiva de los subproblemas [Sean 1998]. Si el tamaño de dos subproblemas es el mismo (o casi), el tiempo de cómputo para una función $f(x)$ bajo el esquema divide y vencerás se describe con la relación:

$$T(n) = g(n) \tag{4.2}$$

siempre y cuando *n* sea pequeño. En el caso contrario, la relación es:

$$T(n) = 2T\left(\frac{n}{2}\right) + f(n) \tag{4.3}$$

donde $T(n)$ es la función de tiempo de la función *X* para entradas de tamaño *n*.

$f(n)$ es el tiempo necesario para partir el problema y combinar las subsoluciones.

$g(n)$ es el tiempo que tarda la función *X* en resolver problemas de tamaño pequeño (normalmente una constante).

La eficiencia final del algoritmo depende de la función $f(n)$ concreta que aparezca durante el análisis. Para que esta técnica resulte eficiente, todos los subproblemas deben ser de tamaño parecido. La solución de los subproblemas se puede hacer en paralelo siempre y cuando:

a) la división produzca subproblemas de costo balanceado.
b) la división y la combinación impliquen comunicaciones y sincronización de las tareas.
c) Se divida en función del número de procesadores.

4.2 IMPLEMENTACION DEL ENTORNO LAM-MPI

Como se mencionó en la Sección 3.2 del Capítulo 3 de esta Tesis, la plataforma de programación en paralelo MPI es un estándar desarrollado para la implementación de sistemas de paso de mensajes. A continuación se resume el entorno LAM-MPI [LAM-MPI 2003] para cómputo paralelo usado en el desarrollo del programa propuesto en esta Tesis.

4.2.1 Paso de mensajes entre nodos.

Se dice que existe transmisión o *paso de mensajes* entre dos procesadores o *nodos* cuando se ejecuta un comando para enviar un mensaje en el *nodo emisor*, y se ejecuta un comando de recepción de este mensaje en el *nodo receptor*. El proceso exacto de paso de mensajes que se realiza después de ejecutarse los comandos depende de cada máquina, pero suele involucrar *procesos de inicialización* de los buffers de mensajes en ambos nodos, así como el establecimiento de la *ruta de comunicación* entre ellos, además de la *transferencia del mensaje* actual a través de los enlaces de comunicación.

Si ß representa el tiempo requerido para preparar el hardware y software necesario para transmitir un mensaje y *t* es el tiempo empleado en enviar un byte de datos a través del enlace que une a dos nodos, el tiempo necesario para enviar *k* bytes de datos desde un nodo a uno de sus vecinos más cercanos está dado por la expresión:

$$t_{comun} = \beta + k \tag{4.4}$$

Normalmente, el tiempo necesario para enviar un mensaje es mucho mayor que el tiempo necesario para realizar una operación en punto flotante. La relación se expresa por:

$$\beta >> w^3 t \tag{4.5}$$

donde *w* es el tiempo necesario para realizar una operación en punto flotante.

Los métodos de comunicación con el nodo vecino más próximo han mejorado el rendimiento a medida que ha mejorado la tecnología en las computadoras. Sin embargo, todavía se cumple que $\beta >> t$. Esto significa que es mejor enviar muchos bytes de datos en un único mensaje que enviar la misma cantidad de datos utilizando muchos mensajes pequeños.

En esta Tesis se presenta un algoritmo paralelo en plataforma MPI tomando en cuenta que se ejecutará en una arquitectura MIMD y por tanto se envían mensajes lo más grande posible para contrarrestar los tiempos de comunicación.

4.2.2 El entorno de ejecución LAM-MPI

El código fuente de LAM-MPI se obtuvo de [LAM-MPI 2003], de donde se eligió la versión 7.0.4, que corresponde al entorno MPI desarrollado en la Universidad de Indiana. La instalación de LAM se hace en modo de super usuario (*root*) y las instrucciones se detallan en [LAM-MPI 2003].

4.2.2.1 Comandos de LAM-MPI

Una vez instalado el entorno LAM-MPI, antes de iniciar la operación se necesita crear un archivo de texto donde se listen los host o direcciones IP´s de los nodos que integran el cluster. A esto se le conoce como “esquema de carga” (*boot schema*). En este caso, se generó un archivo y se le llamó *lamhosts*. La Figura 4.2 muestra el contenido de *lamhosts* para este trabajo de Tesis.

```
#nodo maestro
192.168.10.1
#nodo esclavo 1
192.168.10.2
#nodo esclavo 2
192.168.10.3
#nodo esclavo 3
192.168.10.4
```

Figura 4.2 El archivo *lamhosts*.

Comando lamboot

Antes de ejecutar cualquier programa MPI de debe iniciar el entorno de ejecución de LAM; esto se hace invocado el comando de carga *lamboot* ("*booting LAM*") en cada máquina listada en lamhosts. La respuesta es una lista con los nodos activos que aparecen en el esquema de carga. Una vez inicializado el entorno LAM se dice que los nodos activos constituyen el "Universo LAM". La Figura 4.3 muestra la pantalla de inicio al cargar el entorno de ejecución LAM con el comando *lamboot*.

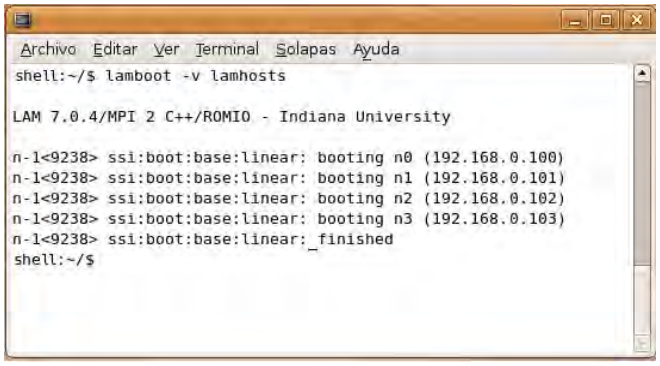


Figura 4.3 Inicio del entorno de ejecución LAM.

Una vez que se invoca el comando *lamboot*, se puede navegar por el entorno LAM. Los posibles argumentos de *lamboot* están indicados en [LAM-MPI 2003].

Comando lamnodes

El comando *lamnodes* muestra cuantos nodos están activos dentro del entorno LAM. Por ejemplo, una vez creado el universo LAM, si se ejecuta este comando resultará un reporte como el que se muestra en la Figura 4.4.

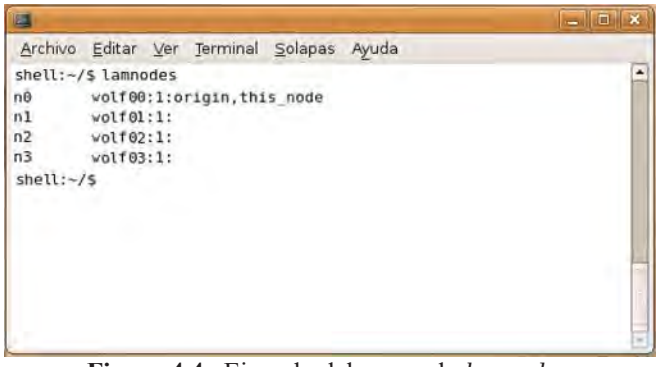


Figura 4.4 Ejemplo del comando *lamnodes*.

Nótese que *n*0 se refiere al nodo maestro o de mayor jerarquía. Puede darse el caso de que un nodo o máquina tenga más de un procesador; entonces se indicará el numero de procesadores que tiene ese nodo tal como se muestra para el nodo 3 (*n*2) y nodo 4 (*n*3) del ejemplo de la Figura 4.4. Finalmente, en la primera línea la notación *origin* indica desde cual nodo fue ejecutado LAM, y *this node* indica en cuál nodo está corriendo LAM. La Tabla 4.3 indica los comandos del entorno LAM_MPI que más se usaron en esta Tesis.

Tabla 4.3 Comandos del entorno LAM-MPI

COMANDO	FUNCION
laminfo	Puede ser usado para consultar las capacidades de la instalación MPI-LAM. Si se invoca *laminfo* sin parámetros muestra en el shell un pequeño resumen de la información.
lamclean	Elimina todos los procesos que estén corriendo en el universo LAM y libera todos los recursos que estaban asociados a dichos procesos.
lamhalt	Se utiliza para "salir" de LAM, es decir, detiene todos los procesos del entrono LAM y libera los recursos que estaban asociados a dichos procesos. Para volver al entorno LAM es necesario ejecutar el comando *lamboot*.
En [LAM-MPI] se pueden consultar la sintaxis y parámetros *laminfo*.	

Comando tping

Este comando es similar al *ping* de Unix. *tping* puede ser usado para verificar la funcionalidad del universo LAM; manda un mensaje de prueba (*ping*) entre los demonios de LAM que constituyen el entorno de ejecución. Comunmente este comando tiene dos argumentos: el conjunto de nodos a probar, (expresado por la notación *N*), y el tiempo de respuesta a la prueba. En la Figura 4.5 se presenta un ejemplo del *tping*.

```
Shell$ tping N -c 3
1 byte from 3 remote nodes and 1 local node:0.002 secs
1 byte from 3 remote nodes and 1 local node:0.002 secs
1 byte from 3 remote nodes and 1 local node:0.002 secs
```

Figura 4.5 Ejemplo del comando *tping*.

Compilar programas en LAM-MPI

LAM realiza la compilación de manera transparente para el programador. En la línea de comandos únicamente se debe agregar la llamada y la opción para el tipo de compilación que se desee. LAM incluye tres compiladores que se indican en la Tabla 4.4.

Tabla 4.4 Compiladores integrados en LAM

LLAMADA	LENGUAJE
mpicc	Compilador para programas escritos en lenguaje C.
mpiCC	Compilador para programas escritos en lenguaje C++.
mpif77	Compilador para programas escritos en lenguaje Fortran77

Para compilar programas MPI escrito en lenguaje C++ se escribe la llamada como se indica en la Figura 4.6.

```
Shell$ mpiCC nombre_programa.c –o nombre_salida
```

Figura 4.6 Ejemplo de compilación con mpiCC.

Nótese que la compilación se hace de manera muy semejante a como se haría para g++. La opción *–showme* mostrará el compilador adecuado para el lenguaje del código y además todas las opciones por los cuales pudo haber pasado para poder ejecutar la compilación. Nótese también que no es necesario tener LAM cargado para poder compilar programas.

Comando mpirun

El comando *mpirun* ejecuta o corre un programa. La Figura 4.7 muestra un ejemplo de compilación y ejecución del programa "hola.c". Las diferentes opciones que pueden ser usadas para controlar la ejecución de un programa en paralelo con *mpirun* se presentan en la Tabla 4.5.

Figura 4.7 Ejemplo del comando *mpirun*.

Tabla 4.5 Modos de ejecutar con *mpirun*.

LLAMADA	DESCRIPCION
mpirun C nombre_salida	La opción C significa ejecutar una copia de nombre_salida en todos CPU$ de los procesadores que están listados en el esquema de carga. Esta notación es conveniente en arquitecturas SMP.
mpirun N nombre_salida	La opción N significa ejecutar una copia de nombre_salida en cada uno de los nodos del universo LAM. Esta notación es conveniente cuando se realizan programas multithreading en MPI.
mpirun –np 4 nombre_salida	Esta opción ejecuta nombre_salida en cuatro nodos. LAM calendarizará cuantas copias del programa se ejecutarán en cada nodo de acuerdo con el número de procesadores que estén listados en el esquema de carga.

4.2.3 Arquitectura para el entorno LAM-MPI

El cluster en el cual se realizaron los Casos de Estudio descritos en el Capítulo 5 de esta Tesis pertenece a la DEP-FIE [Medina 2008]. Los componentes básicos que conforman el cluster son:

- Hardware: máquinas que fungen como nodos o elementos de procesamiento (EP), teniendo en cuenta su costo y rendimiento.
- Comunicación entre nodos. Se toma en cuenta la arquitectura del cluster y el medio de comunicación físico.
- Software para configurar el cluster.
- Aplicación de plataforma paralela. Instalación y configuración de bibliotecas para programación paralela.

4.2.3.1 Hardware del cluster

El cluster está conformado por 4 computadoras. Las características físicas de las máquinas se muestran en la Tabla 4.6. Se cuenta también con un monitor, un teclado y un ratón para todo el cluster.

Tabla 4.6 Características de las máquinas que conforman el cluster.

CARACTERISTICAS	
Marca	Compaq
Modelo	Intel Celeron D
Velocidad	3.4 GHz
Memoria RAM	1.7 MB
Unidad de Almacenamiento Primario	HD 20 GB
Interfase de Red	Ehernet 10 Mbps
Complementarios	CD-ROM.
Gabinete	minitorre

[Swendson 2004] recomienda que el nodo maestro de un cluster sea la máquina con más potencia de cálculo. En este caso, como todas las computadoras son iguales, el nodo maestro corresponde al host que se lista primero en el archivo *lamhosts* (Figura 4.2). En el nodo maestro del cluster se crearán y ejecutarán los programas, mientras que los nodos esclavos servirán como elementos de procesamiento.

Al contar con una sola tarjeta de red por máquina, los nodos no tendrán acceso a la red pública, ya que su tarjeta estará dedicada exclusivamente a la red interna de interconexión del cluster. En la Figura 4.8 se muestra la vista frontal del cluster.

Figura 4.8 Vista frontal del cluster.

El único monitor se conecta a un *switch*, que multiplexa 4 puertos para video, el teclado y el ratón, mediante cuatro botones selectores, uno para cada nodo del cluster. En la Figura 4.9 se observa la vista frontal del switch de video.

Figura 4.9 Vista frontal del switch de video.

4.2.3.2 Comunicación entre nodos y topología

Las comunicaciones en el cluster se establecen por medio de la red de conexión y la topología. Sus características son:

- Red:
  - Conexión de red interna con un *switch autosense* 3Com de 10/100 Mbits con 16 puertos.
  - Cableado UTP.
- Topología: estrella.

En la Figura 4.10 se puede observar el switch y el cableado que atiende la comunicación entre los nodos del cluster.

Figura 4.10 Vista frontal del switch para conexión de los nodos.

Para el paso de mensajes se requiere un soporte TCP/IP de transmisión de datos que permita la comunicación en la red. El cluster tiene una configuración de red local que permite comunicación ininterrumpida entre los nodos.

La red local está especificada como *privada* [Sterling 2003] y [BEOWULF2], y se identifica con el rango de IP´s 192.168.10.*n,* teniendo una máscara de subred de 24 bits (255.255.255.0); con ello, se proporcionan 253 direcciones IP utilizables, con $n = 1,...,254$.

4.2.3.3 Configuración del software del cluster

- Sistema Operativo Linux, distribución Ubuntu 7.04 (kernel 2.6.20).
- Todo el software, tanto el sistema operativo como las aplicaciones, son instaladas en los discos duros de cada nodo. Tener todas las aplicaciones instaladas es una diferencia que hace que este cluster no sea un Beowulf puro.
- Cualquier nodo puede ser nodo maestro o nodo esclavo. El usuario determina el orden en el archivo *lamhosts*.
- Los nodos esclavos obtienen los directorios *home* de los usuarios y los sistemas de archivos que contienen las aplicaciones, a través de NFS, desde el nodo maestro.
- El cluster esta configurado como una red interna de dirección 192.168.10.*n* , donde $n = 1,..,4$.

4.2.3.4 Semejanzas con el cluster Beowulf

La arquitectura implementada para soportar el procesamiento paralelo tiene características de un cluster tipo Beowulf. Aunque presenta algunas diferencias respecto al Beowulf puro, el cluster utilizado en esta Tesis se concibe como tipo Beowulf. A continuación se listan las semejanzas y diferencias del cluster implementado para esta Tesis con respecto al Beowulf.

Semejanzas con el cluster Beowulf

- Independencia de los nodos: Cada nodo tiene el hardware mínimo que le permite funcionar como unidad de cálculo. Los nodos carecen de monitores, teclados, disqueteras, ratones y tarjetas de video.
- Exclusividad: Cada nodo se dedica exclusivamente a los procesos propios del cluster, aunque puede ejecutar simultáneamente otros procesos o programas para monoprocesador.
- Direccionamiento: El nodo maestro tiene una dirección IP válida. El resto de los nodos tienen direcciones de red privada (192. 168.XXX.XXX), de forma que no es posible acceder directamente a un nodo Beowulf desde otro nodo.
- Servicios de los nodos: Los servicios y comandos para transferencia de información entre todos los nodos están inhabilitados. Únicamente se conectan con el nodo maestro.

Diferencias con el cluster Beowulf

- Acceso: Se hace a través de una terminal conectada a la red del cluster. Esta terminal si se usa como elemento de proceso.
- Mecanismo de comunicaciones: Utiliza un switch central que determina el rendimiento para evitar colisiones.

4.2.3.5 Aplicación de plataforma paralela

El cluster tiene instalado el software de bibliotecas necesario para programación paralela en MPI y PVM. Se realizó una prueba de funcionamiento del cluster en ambiente MPI, al ejecutar el ambiente LAM y mostrar los nombres de host con el comando *lamnodes*, tal como se indica en la Figura 4.11.

Figura 4.11 Prueba del cluster con MPI-LAM.

4.3 PARALELIZACION DEL ALGORITMO DE DIFERENCIACION NUMERICA.

El algoritmo paralelo para solución en EEP de sistemas eléctricos que se propone en esta Tesis se basa en las contribuciones de [Semlyen y Medina 1995], [Medina *et al* 2004] y [Medina y Ramos-Paz 2005]. Se explota el método DN en el dominio del tiempo, propuesto en [Semlyen y Medina 1995], y se propone su paralelización utilizando procesamiento en paralelo MPI [García 2005] con el cluster descrito en la Sección 4.2.3. El algoritmo propuesto esta codificado con POO, utilizando el compilador ANSI C++, y considera vectores, matrices y listas cuyos elementos están definidos para representar los componentes de las redes eléctricas.

Dominio del tiempo

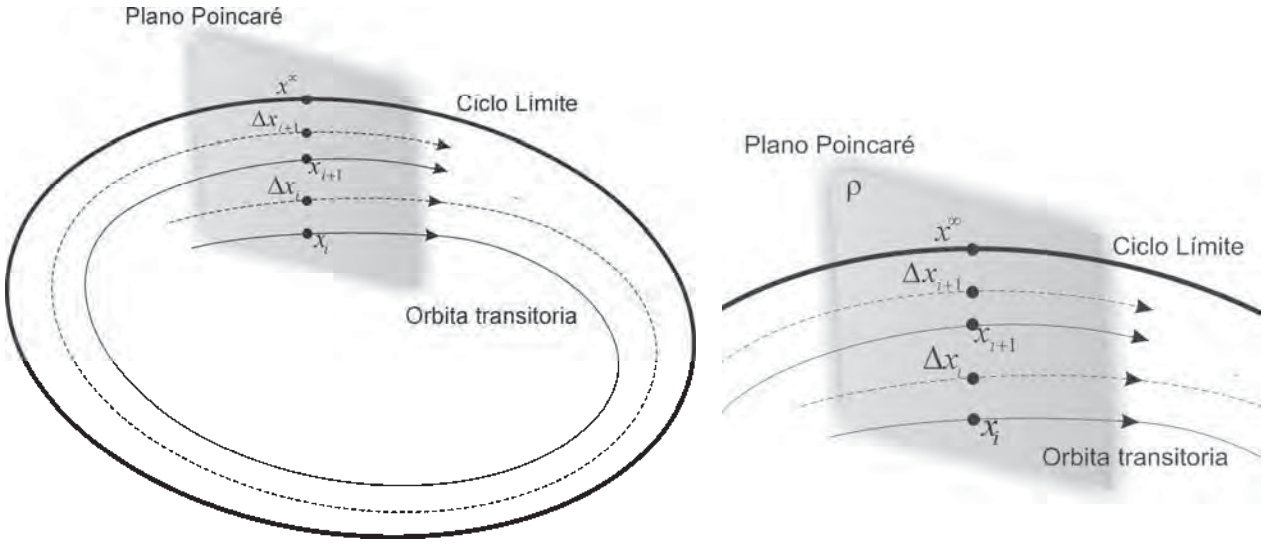
Considérese un sistema eléctrico con componentes no lineales. En general, el sistema se describe por la ecuación diferencial:

$$\dot{x} = f(x,t) \tag{4.6}$$

donde x es el vector de estados de los n elementos de x_k.

Si la fuerza conductora en el sistema es periódica, entonces $f(x,t)$ será un vector T periódico y la solución en el estado estacionario $x(t)$ también será periódica.

$x(t)$ puede representarse como el *Ciclo Limite* para x_k en términos de otro elemento periódico de x, o bien en términos de una función T periódica, o como $sen(\omega t)$, tal como se observa en la Figura 4.12. Otras variables tales como $i(t)$ se obtienen por medio de ecuaciones algebraicas.



a) Ciclo Límite b) Plano Poincaré

Figura 4.12 Orbitas del vector de estado x.

Antes de alcanzar el Ciclo Limite, los ciclos de una orbita transitoria estarán muy cerca de éste, y su posición está debidamente descrita por su representación sobre el plano de Poincaré ρ [Parker y Chua 1989]. Para un ciclo sencillo, se mapea su punto de inicio en x_i a su punto final en x_{i+1}, y se mapea un segmento de perturbación Δx_i (desde este *Ciclo Base* [Semlyen y Medina 1995]) a Δx_{i+1}. Se observa que todos los mapeos cerca de un Ciclo Limite son *cuasi-lineales*, por tanto, un método Newton (ME, AD o DN) o su aproximación puede usarse para obtener el punto para el Ciclo Limite x^{∞}. La ecuación de [Semlyen y Medina 1995] que describe la convergencia al Ciclo Límite es:

$$x^{\infty} - x_i = C(x_{i+1} - x_i) \tag{4.7}$$

donde x^{∞} representa el vector de variables de estado en el Ciclo Límite, x_i representa el vector de variables de estado al comienzo del Ciclo Base, x_{i+1} representa el vector de variables de estado al final del Ciclo Base y C es una matriz que determina la relación lineal de las variables de estado en el Ciclo Límite.

En [Semlyen y Medina 1995] la matriz C se calcula con la relación:

$$C = (I - \Phi)^{-1} \tag{4.8}$$

donde I es la matriz identidad y Φ es la *matriz de identificación*.

El punto principal en este proceso recae en la identificación de los valores de la matriz Φ, ya que una vez teniendo Φ puede calcularse la matriz C y estimar x^{∞} en el Ciclo Limite.

4.3.1 Método de Diferenciación Numérica

Asumiendo un problema no lineal con parámetros variantes en el tiempo de la forma (4.6):

$$\dot{x} = f(x,t), \quad x(0) = x_0$$

La integración de (4.6) después de T periodos de tiempo resulta en la ecuación:

$$f(x, t+nT) = f(x,t) \tag{4.9}$$

Al final de n periodos, se obtiene el vector de estado x_n para $t=nT$; calculando un periodo adicional después, se obtiene x_{n+1}.

Si se aplica una perturbación a la variable de estado, de manera que x_0 sea perturbada como $x_0 + \Delta x_0$, o de manera general $x(t) \rightarrow \Delta x(t)$, entonces (4.6) toma la forma:

$$\dot{x} + \Delta\dot{x} = f(x + \Delta x, t) \tag{4.10}$$

De (4.10) se observa que x es una variable incremental mientras que t se mantiene constante. La solución general propuesta en [Semlyen y Medina 1995] es una matriz dada por $\Delta x(T)$ para $t=T$, esto es:

$$\Delta x(T) = \Phi(T)\Delta x(0) \tag{4.11}$$

donde

$$\Phi(T) = e^{\int_0^T J(t)dt} \tag{4.12}$$

siendo $\Phi(T)$ la matriz de identificación, y el Jacobiano $J(t)$ esta determinado por:

$$J(t) = D_x f(x,t) \tag{4.13}$$

donde D_x es la matriz de derivadas parciales de $f(x,t)$ con respecto de x.

Diferenciación Numérica

Una alternativa para calcular $\Phi(T)$ es el uso de un método Newton como ME, AD o DN. En los métodos ME y AD es necesario conocer el Jacobiano $J(t)$. A menudo $J(t)$ puede ser calculado analíticamente, pero no siempre es el caso. En particular, en el caso de los dispositivos conmutados, es mas fácil usar en (4.10) el incremento Δf en lugar de $J(t)\Delta x$ [Semlyen y Medina 1995]. En esta Tesis se emplea el método DN y se obtiene $\Phi(T)$ por columnas, siguiendo una perturbación secuencial de las variables de estado calculadas en el Ciclo Base $x(t)$ [Parker y Chua 1989]. El procedimiento de DN se describe a continuación.

La perturbación de las variables de estado se expresa por:

$$x_i = x_0 + \xi e_i \tag{4.14}$$

donde e_i es la columna i de la matriz identidad I, y ξ es un número pequeño, generalmente del orden de 1×10^{-6} *p.u.* De (4.14) se obtiene que:

$$x_i - x_0 = \xi e_i \tag{4.15}$$

En general, para un problema de orden n, la relación estará dada en la forma:

$$\Delta x_{i+1} = \Phi \Delta x_i \tag{4.16}$$

Calculando la diferencia de los dos valores de x al final del ciclo, se obtienen las columnas de Δx_{i+1} de la ecuación (4.16). Esto es, usando (4.15) en (4.16), resulta la relación:

$$\Delta x_{i+1} = x_i^{'} - x_0^{'} = \Phi \xi e_i \tag{4.17}$$

con

$$x_i^{'} = \phi(x_i) \tag{4.18}$$

y

$$x_0^{'} = \phi(x_0) \tag{4.19}$$

donde $x_i^{'}$ es el vector de variables de estado perturbadas, x_0 es el vector de variables de estado al final del Ciclo Base, $\phi(x_i)$ es la solución de (4.6) con $x(0)=x_i$ y $\phi(x_0)$ es la solución de (4.6) con $x(0)=x_0$ [Semlyen y Medina 1995].

Entonces, para un problema de orden *n* con *i*=1,2,…,*n*, la matriz de identificación Φ puede ser calculada de (4.17) con la ecuación:

$$\Phi = \frac{\Delta x_{i+1}}{\xi} \tag{4.20}$$

4.3.1.1 Variables de estado en el Ciclo Límite

Por medio del conjunto de relaciones en el plano de Poincaré, se identifica la matriz *C* definida en [Semlyen y Medina 1995] de (4.7). Así, la relación entre las matrices Φ y *C* se encuentra analizando la Figura 4.2 b), donde se tiene que:

$$\Delta x_i = x^{\infty} - x_i \tag{4.21}$$

con lo que

$$\Delta x_{i+1} = x^{\infty} - x_{i+1} \tag{4.22}$$

sustituyendo en (4.16), la solución para x^{∞} resulta:

$$x^{\infty} = x_i - C(x_{i+1} - x_i) \tag{4.23}$$

y ésta es una estimación de la localización del Ciclo Limite en $C = (I - \Phi)^{-1}$.

Nótese que (4.23) conduce a un proceso Newton si Φ y *C* se actualizan en cada iteración usando (4.15) y (4.8), de manera que resulta un proceso linealmente convergente si *C* se mantiene constante o se actualiza en alguna etapa del proceso iterativo después de su primera evaluación con (4.23).

Una variante de la solución de (4.7) consiste en mantener constante Φ durante un número de subsecuentes evaluaciones de x^{∞}, entonces se recalcula otra vez, comenzando un proceso cíclico hasta que se alcanza la convergencia al Ciclo Límite (la iteración de la matriz Φ es calculada para cada DN).

En [García *et al.* 2001] se propuso la aplicación de procesamiento MT, en la que se usan hilos de control en la técnica de Aceleración de la Convergencia de las variables de estado al Ciclo Límite. Para el mismo propósito, en [Medina *et al.* 2003] se aplica PVM, en [Medina *et al.* 2006] utilizan PVM y MT, y en [García 2005] se reporta la aplicación de MPI utilizando un cluster con memoria distribuida. En esta Tesis se aplica procesamiento en paralelo MPI para el cálculo de las columnas de la matriz de identificación Φ, utilizando un esquema de computación distribuida con el cluster Beowulf implementado en la DEP-FIE.

Referente a la inversión de la matriz *C* en (4.8), se tomaron en cuenta las contribuciones de [Abur 1988] en donde se explota el paralelismo a través del reordenamiento y particionamiento de una matriz de coeficientes, partiendo de un algoritmo secuencial, y de [Asenjo 1997] quien presenta paralelización de la factorización *LU* con matrices dispersas. En esta Tesis se obtiene la matriz *C* de (4.8) por el método de *Doolittle* y *Crout* [Chapra 2003], mismo que se describe en el Apéndice B.

La programación paralela de los métodos RK4, matriz de identificación y matriz inversa se hizo en base a POO [Heileman 1998], y con el enfoque de [Medina et al. 2004].

En la Figura 4.13 se presenta el diagrama de flujo del proceso secuencial para el cálculo de la matriz Φ, utilizando el método DN.

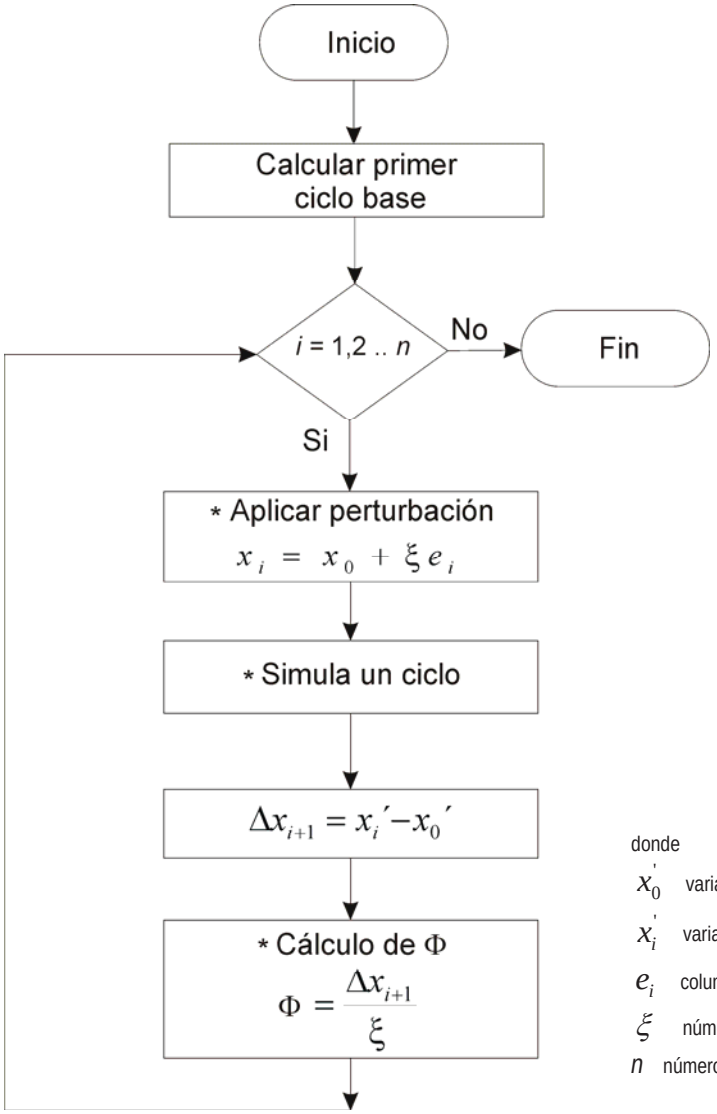


Figura 4.13 Algoritmo secuencial de DN.

4.3.2 Esquema de paralelización propuesto para el algoritmo de DN.

En la implementación paralela con MPI del algoritmo de DN mediante el cluster propuesto, se consideraron las cuatro etapas que indica [Foster 1995]:

- **Partición.** Se realizo la *descomposición funcional* al identificar las etapas del algoritmo factibles a ser tratadas mediante procesamiento en paralelo. Posteriormente se determinó una *descomposición del dominio* cuando se encuentra la relación (división) de la cantidad de elementos o datos y el número de nodos que pueden procesarlos. Se tuvieron en cuenta los siguientes aspectos: a) Se evitaron totalmente las operaciones redundantes, y b) Se estableció una función para lograr una repartición equivalente del cálculo en los nodos esclavos, de manera que se cumpla el balanceo de la carga lo mejor posible.
- **Comunicación.** Siendo que la topología del cluster es tipo estrella con memoria distribuida, cada tarea tiene una identificación única y las tareas interactúan enviando y recibiendo mensajes hacia y desde tareas específicas. Por el volumen del envío-recepción de datos, se determinó no utilizar las funciones de comunicación punto a punto *MPI::Send()* y *MPI::Recv()*, en su lugar, se utilizaron las funciones colectivas de MPI. En esta etapa todas las operaciones de comunicación se ejecutan concurrentemente (son funciones colectivas de MPI) y los cálculos de las diferentes tareas (procedimientos), una vez identificados los datos sobre los cuales van a aplicar el cálculo, se ejecutan concurrentemente.
- **Agrupación.** Se propone una granularidad gruesa agrupando el mayor número de datos posibles, de manera que se redujo el número de mensajes y por ende, el costo de comunicaciones. De hecho, el número de mensajes enviados es constante durante todo el procesamiento paralelo, y es equivalente a la relación que hay entre el número de nodos y el número de variables de estado del problema dado. Además, se verificó que el número de tareas no aumenta con el tamaño del problema.
- **Asignación.** Siguiendo las definición de asignación de [Thomson 1992], se aplicó fue una a*signación estática* de tareas.

4.3.2.1 Diagrama del algoritmo paralelo de DN.

El algoritmo paralelo que se propone en esta Tesis presenta una fracción secuencial (trabajo que realiza el nodo maestro) y una fracción paralelizable basada en la técnica divide y vencerás (trabajo que realizan todos los nodos del cluster).

En la Figura 4.14 se muestra el código del algoritmo propuesto. El diagrama de flujo del algoritmo propuesto se observa en la Figura 4.15, y en él se señalan las secciones que son factibles de paralelizar.

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

        MPI::Status status;

        MPI::Init(argc,argv);

        my_id = MPI::COMM_WORLD.Get_rank();
        nproc = MPI::COMM_WORLD.Get_size();

        reparte_procesos();
        inicializa();
        caso_estudio_1();

        while(error_max > tolerancia){
                Runge_Kutta_4_paralelo(Y_ini);
                error_max=calcula_error(Y_ini,Y_sol);
                if(ciclos >ciclos_cap){
                        calcula_Phi_paralelo();    //Calculo Matriz Identificacion
                        resta();
                        LU();                      //Factorizacion Doolittle
                        sustitucion();             //Factorizacion Crout
                        calcula_yinf(Y_sol,Y_ini);
                        for (int i=0; i<Ve; i++)
                                Y_sol[i]=Y_inf[i];
                }

                intercambia(Y_ini,Y_sol);
                t0=t;
                t=t0+intervalo;
                ciclos++;
        }

return 0;
}
```

Figura 4.14 Código del algoritmo propuesto para la solución en EEP de sistemas eléctricos.

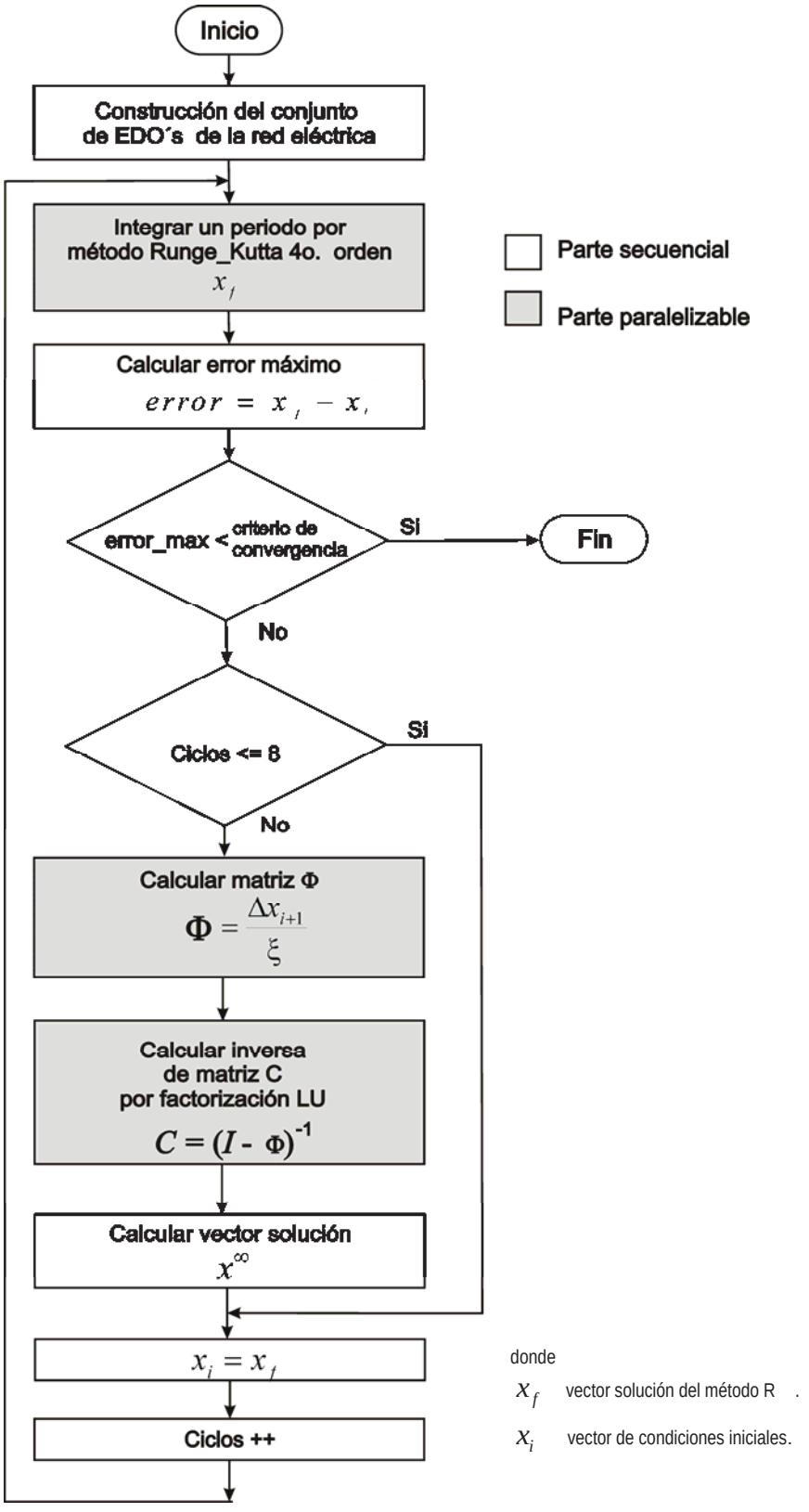


Figura 4.15 Diagrama del algoritmo propuesto para la solución en EEP de sistemas eléctricos.

Las secciones paralelas propuesta son:

- **La etapa de integración por Fuerza Bruta**. Se propone el algoritmo de Runge-Kutta de 4º orden. Dicho algoritmo se particiona en procedimientos (segmentos de programa) que calculan los vectores k_1, k_2, k_3 y k_4. Como existe dependencia entre constantes, se aplica la paralelización al dividir el vector de condiciones iniciales x_n (aplicando paralelismo de datos) y se asigna una parte de éste a cada elemento de procesamiento, de tal manera que al final de la iteración, se puedan reunir los resultados en un vector que contiene las constantes *k* que corresponden. Finalmente se calcula el vector solución x_{n+1}.
- **El cálculo de la matriz de identificación Φ**. El proceso de la identificación de la matriz Φ es totalmente asíncrono, es decir, cada columna de Φ se puede calcular por separado de manera independiente al resto de las columnas [García 2001] y [Ramos 2002], se utilizará la plataforma MPI con el cluster construido.
- **El procedimiento para la inversión matricial.** Se propone emplear la combinación del algoritmo secuencial de la *Descomposición de Doolittle* (*LU*) que utiliza la factorización triangular *LU* y posteriormente hacer una versión paralela del algoritmo de la *Factorización de Crout* (*LDU*) para resolver *n* sistemas de ecuaciones que conformaran las columnas de la matriz inversa. Los métodos de *Doolitle* y *Crout* se explican en el Apéndice B.

4.3.3 Programación de la sección secuencial del algoritmo de DN.

La parte secuencial del código del algoritmo paralelo de DN corresponde a las funciones que realiza únicamente el nodo maestro del cluster. De acuerdo con el diagrama de flujo del algoritmo propuesto (Figura 4.15), las secciones de la parte secuencial son las siguientes:

- Encabezado del programa.
- Declaración de variables globales.
- Definición de clases.
- Definición de funciones del programa.

4.3.3.1 Encabezado del programa

En la primera línea se incluyó la llamada a las librerías de C++. La segunda línea incluye la librería <*mpi.h*>, que contiene todas las funciones del entorno y comunicación para el paso de mensajes de la plataforma MPI. <*math.h*> contiene funciones matemáticas. Enseguida se indicaron el número de ecuaciones de estado *Ve* del caso de estudio, el número de puntos de integración por periodo *hh* y el número de ciclos de captura *ciclos_cap* para el método DN. Esta última variable establece los ciclos a partir de los cuales comienza el proceso de la aceleración para lograr la convergencia; se define como 8, según lo recomendado en [Semlyen y Medina 1995]. La Figura 4.16 muestra el encabezado del algoritmo paralelo propuesto.

```
#include <iostream>
#include <mpi.h>
#include <math.h>
#define Ve 7
#define hh 512
#define ciclos_cap 8
```

Figura 4.16 Código del encabezado del algoritmo paralelo propuesto.

4.3.3.2 Parámetros globales

Siguiendo con la secuencia del programa, se indicaron las variables y constantes globales. Todas ellas se declararon del tipo *double* en razón de que se requiere buena precisión de cálculo porque se trabaja con números fraccionarios. El compilador de C++ que se utilizó incluye las librerías de punto flotante del formato estándar IEEE 754[1].

Se indicaron los valores de frecuencia de las fuentes de energía *w*, el tiempo inicial de cálculo *t0*, el tiempo final *intervalo* (1/60 segundos) para el método de Fuerza Bruta, el error máximo *error_max*, la tolerancia o criterio a partir del cual se considera que la solución llega a la convergencia es *tolerancia* (1×10^{-10} *p.u.*) y la magnitud de la perturbación para el cálculo de los ciclos base en *epsilon* (1×10^{-6}). En la Figura 4.17 se observa la declaración de los parámetros globales.

```
double w = 2.0*(4.0*atan(1.0))*60.0;
double intervalo = 1.0/60.0;

double error_max=1, tolerancia=1e-10, t0=0.0, epsilon=1e-6;
```

Figura 4.17 Código de la declaración de parámetros globales.

En la Figura 4.35 se presentan las variables *my_id* y *nproc*. La variable *nproc=MPI::COMM_WORLD.Get_size()* recibe el número de elementos de proceso activos en la arquitectura paralela. *my_id=MPI::COMM_WORLD.Get_rank()my_id* toma el número del elemento de proceso en que se está ejecutando el programa (recordar que MPI relaciona cada elemento de proceso con un número de acuerdo a su ubicación en el archivo *lamhosts*). Ambas variables son la base para la implementación del procesamiento en paralelo con MPI de esta Tesis.

4.3.3.3 Funciones

Las funciones públicas que constituyen la parte secuencial del programa se describen a continuación.

- *inicializa()*. En esta función se generan los vectores de trabajo para los procedimientos de Fuerza Bruta (*v_K*1, *v_K*2, *v_K*3, *v_K*4, *Y_K*1, *Y_K*2, *Y_K*3), el cálculo de la matriz de identificación Φ y de *C* (*Phi*, *C*, *Y_aux*). También se inicializan los vectores *Y_ini*

[1] Norma publicada en 1985 que trata sobre la especificación relativa a la precisión y formato de los números de punto flotante. Incluye una lista de operaciones para dichos números y diversas conversiones, así como tratamiento del número infinito. Define el tipo *double* de 64 bits, precisión científica de 15 dígitos en un rango de $2.23\times10^{-308} \le x \le 1.79\times10^{308}$.

(contiene los valores de condición inicial de las variables de estado), *Y_sol* (de solución final parcial después de haber sido aplicado el método de integración) y el vector auxiliar *Yn*1. Todos los vectores definidos en *inicializa()* son apuntadores a datos de tipo *double* almacenados de manera contigua en memoria reservada a través del operador *new*. El código se muestra en la Figura 4.18.

```
void inicializa(void){

        V_K1=new double[Ve]; V_K2=new double[Ve];  V_K3=new double[Ve];  V_K4=new double[Ve];
        Y_K1=new double[Ve]; Y_K2=new double[Ve];  Y_K3=new double[Ve];
        Yn1=new double[Ve];  Y_sol=new double[Ve]; Y_ini=new double[Ve]; Y_aux=new double[Ve];

        Phi=new double[Ve*Ve];  C=new double[Ve*Ve];

        //inicializa vectores de condiciones iniciales
        for(i=0; i<Ve; i++){
                Y_ini[i]=0;  Y_sol[i]=0;  Yn1[i]=0;

        }
}
```

Figura 4.18 Código de la función *inicializa().*

- *intercambia()*. Esta función hace el cambio de la solución parcial encontrada para tomarla como condición inicial en el siguiente cálculo. El código se muestra en la Figura 4.19.

```
        void intercambia(double *yini, double *ysol){
                for (int i=0; i<Ve; i++) yini[i]=ysol[i];
        }
```

Figura 4.19 Código de la función *intercambia().*

- *calcula_error()*. En esta función se determina el error entre la condición inicial y la solución parcial. El código se muestra en la Figura 4.20.

```
        double calcula_error(double *yini, double *ysol){
                double error=-1, dif=0;
                for(int i=0; i<Ve; i++){
                        dif=valor_abs(ysol[i]-yini[i]);
                        if(dif>error) error=dif;
                }
                return error;
        }
```

Figura 4.20 Código de la función *calcula_error().*

- *resta()*. En esta función se realiza la operación matricial Φ-*I*. El código se muestra en la Figura 4.21.

```
    void resta(void){
            for (int i=0; i<Ve; i++)
                    for (int j=0; j<Ve; j++)
                            if(i==j)
                               Phi[i*Ve+j]=1-Phi[i*Ve+j];
                            else
                               Phi[i*Ve+j]=Phi[i*Ve+j]*-1;

    }
```

Figura 4.21 Código de la función *resta().*

- *calcula_yinf()*. Esta función realiza el cálculo del vector que contiene los valores de las variables de estado en el Ciclo Límite. Su código se puede ver en la Figura 4.22.

```
void calcula_yinf(double *ysol,double *yini){
        double suma;
        Y_inf=new double[Ve];
        for (int i=0; i<Ve; i++){
                suma = 0;
                for (int j=0; j<Ve; j++)
                        suma = suma + C[i*Ve+j]*(ysol[j]-yini[j]);
                Y_inf[i] = yini[i] + suma;
   }
}
```

Figura 4.226 Código de la función *calcula_yinf()*.

- *reparte_procesos()*. En esta función se efectúa la *descomposición del dominio* para posteriormente aplicar el paralelismo de datos. En esta función se distribuyen los *m* procesos en base al número de variables de estado del sistema de estudio y los *p* elementos de proceso o nodos de que se dispone en la arquitectura paralela. Pueden ocurrir tres posibilidades en la distribución de procesos:
    - $p < m$ El número de procesadores es menor al número de las variables de estado del caso de estudio.
    - $p = m$ Caso ideal: el número de procesadores sea igual al número de las variables de estado del caso de estudio.
    - $p > m$ El número de procesadores es mayor al número de las variables de estado del caso de estudio.

En la Figura 4.23 se muestra el código de la función *reparte_procesos()*. Los vectores *cuantos* e *inicia* almacenan el número de tareas que corresponderá ejecutar a cada elemento de proceso, y la posición de inicio de cálculo dentro de los vectores donde se almacenan los datos, respectivamente.

```
void reparte_procesos(void){
  int entera, residuo;
  cuantos=new int [nproc];
  inicia=new int [nproc];

  if(Ve>nproc){
    entera=Ve/nproc;
    residuo=Ve%nproc;
    inicia[0]=0;

    for(i=1; i<nproc+1; i++){
      cuantos[i-1]=entera;
      if(residuo!=0){cuantos[i-1]+=1; residuo=residuo-1;}
      inicia[i]=inicia[i-1]+cuantos[i-1];
    }
   }
   if(Ve==nproc){
       for(i=0; i<nproc; i++){
          inicia[i]=i;
          cuantos[i]=1;
       }
   }
   if(Ve<nproc){
       for(i=0; i<nproc-Ve; i++){
          inicia[i]=i;
          cuantos[i]=1;
       }
   }
```

Figura 4.23 Código de la función *reparte_procesos()*.

4.3.3.4 Estructura lógica para el sistema eléctrico de estudio.

En código del algoritmo propuesto contiene una sección para representar la dinámica del sistema eléctrico de estudio. Esta sección consiste en la programación de una estructura lógica implementada con un arreglo de listas ligadas simples [Medina *et al*. 2004].

Todas las EDO´s, con sus *n* términos, que representan el comportamiento del sistema eléctrico, se expresan por medio de listas ligadas simples. Cada lista simboliza una EDO, como se ilustra en la Figura 4.25.

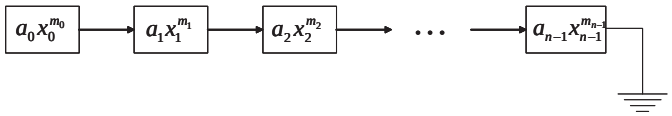


Figura 4.25 Esquema de una ODE por medio de una lista enlazada simple.

Cada uno de los elementos de la lista tiene un coeficiente que puede significar la resistencia, inductancia, capacitancia u otro valor específico usado en la representación del espacio del estado de los elementos del sistema.

El sistema de EDO´s se representa en la Figura 4.24. Se define un arreglo de listas ligadas simples, y para un sistema de *n* EDO´s se construirán *n* listas.

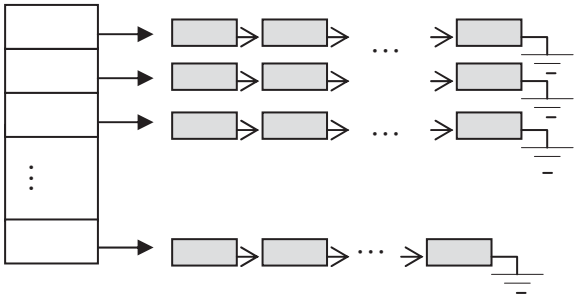


Figura 4.24 Esquema de un sistema de ecuaciones diferenciales ordinarias.

En esta Tesis se implementó un arreglo de listas ligadas simples, porque ajustando únicamente el índice del arreglo se puede acceder a la EDO respectiva, y la evaluación de la ecuación se realiza a través de un recorrido sobre la lista.

En el código del programa propuesto se establece la *clase termino* para crear los objetos que representan los términos de una EDO [Medina *et al*. 2004]. Cada objeto establece 4 miembros que simbolizan el coeficiente (*coef*), el exponente (*exp*), el número de la variable de estado a la que refiere el término (*ve*) y un apuntador (*siguiente*) al siguiente objeto del mismo tipo. En la Figura 4.26 se muestra el esquema del objeto *termino*. El equivalente en código del objeto *termino* se observa en la Figura 4.27.

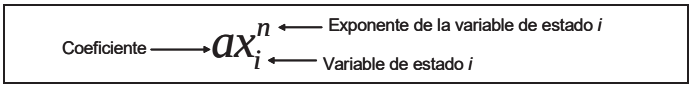


Figura 4.26 Representación del objeto término.

```
class termino{
public: double coef; int ve; double exp;
        termino *siguiente;

        termino(double coef, int ve, double exp){
                termino::coef=coef; termino::ve=ve; termino::exp=exp;
        };
};
```

Figura 4.27 Código del objeto término.

La estructura de la lista ligada simple se establece con la *clase ecuación*, misma que hereda la *clase termino*; el código se observa en la Figura 4.28. La clase ecuación define el objeto *ecuacion* e incluye la función *anexa_termino()* para insertar objetos de tipo *termino*.

```
class ecuacion:public termino, fuentes{
public: termino primero, *ultimo;
        int num_fuentes;
        fuentes *fuente;
ecuacion(void){ primero.siguiente=NULL; ultimo=&primero; fuente=0;};
void anexa_termino (termino *termino_nuevo);
};
```

Figura 4.28 Código de la clase *ecuacion*.

La *clase ecuación* hereda también a la *clase fuentes*, cuyos dos miembros están definidos por los parámetros *magnitud* y *teta* (ambos de tipo doble); éstos representan la magnitud y el ángulo de una fuente de energía asociada a un nodo de la red eléctrica.

Los sistemas eléctricos de estudio se programaron en la función *caso_estudio_n()*. Aquí, la estructura *ec*, es un vector cuyos elementos son las listas ligadas. La longitud de *ec* corresponde con la cantidad de ecuaciones de estado del caso de estudio. Mediante un índice de *ec* se apunta a las listas correspondientes, es decir, a cada ecuación del sistema de estudio. El código se observa en la Figura 4.29.

```
void caso_estudio_1(void){

  ec=new ecuacion[ve];

        //primera ecuacion
        ec[0].anexa_termino(new termino(-r1/l1,0,1.0));
        ec[0].anexa_termino(new termino(-1.0,5,1.0));
        ec[0].fuente=1;//fuente
        //segunda ecuacion
        ec[1].anexa_termino(new termino(-r2/l2,0,1.0));
        ec[1].anexa_termino(new termino(-1.0,6,1.0));
        ec[1].fuente=1;//fuente
        //tercera ecuacion
        ec[2].anexa_termino(new termino(-r3/l3,2,1.0));
        ec[2].anexa_termino(new termino(1.0,5,1.0));
        ec[2].anexa_termino(new termino(-1.0,6,1.0));
        //cuarta ecuacion
        ec[3].anexa_termino(new termino(-rm1,3,nn));
        ec[3].anexa_termino(new termino(1.0,5,1.0));
        //quinta ecuacion
        ec[4].anexa_termino(new termino(-rm2,4,nn));
        ec[4].anexa_termino(new termino(1.0,6,1.0));
        //sexta ecuacion
        ec[5].anexa_termino(new termino((1.0/(l1*c1)),0,1.0));
        ec[5].anexa_termino(new termino((-1.0/(l3*c1)),2,1.0));
        ec[5].anexa_termino(new termino((-1.0/c1),3,nn));
        //septima ecuacion
        ec[6].anexa_termino(new termino((1.0/(l2*c2)),1,1.0));
        ec[6].anexa_termino(new termino((1.0/(l3*c2)),2,1.0));
        ec[6].anexa_termino(new termino((-1.0/c2),4,nn));

}
```

Figura 4.29 Código de la función *caso_estudio_1()*.

4.3.4 Paralelización del método de la Fuerza Bruta.

La metodología conocida como Fuerza Bruta (FB) se ha empleado para alcanzar el estado estacionario periódico de sistemas eléctricos [Parker y Chua 1989]. El algoritmo de la FB contempla los siguientes pasos:

1. Establecer el conjunto de ecuaciones diferenciales ordinarias que representan la dinámica del sistema.
2. Determinar las condiciones iniciales para el análisis en el tiempo. Generalmente los valores iniciales están contenidos en un vector, y representan los valores que tiene las variables de estado en un tiempo previo.
3. Aplicar un método de integración considerando un periodo de tiempo y los puntos en que se va a dividir dicho periodo.
4. Establecer un criterio de convergencia (por ejemplo, de $1\times10^{-10}\,p.u.$). Después de aplicar el método de integración en un periodo de tiempo, se obtiene el vector solución con los nuevos valores de las variables de estado.
5. Calcular el error máximo comparando ambos vectores que contienen los valores de las variables de estado antes y después de aplicar la integración. Si el error es mayor que el valor del criterio de convergencia, se intercambian los valores del vector solución y el vector de condiciones iniciales.
6. Repetir el proceso hasta alcanzar el criterio de convergencia; entonces se habrá llegado al estado estacionario periódico de operación del sistema.

La Figura 4.30 muestra el diagrama de flujo del algoritmo de FB.

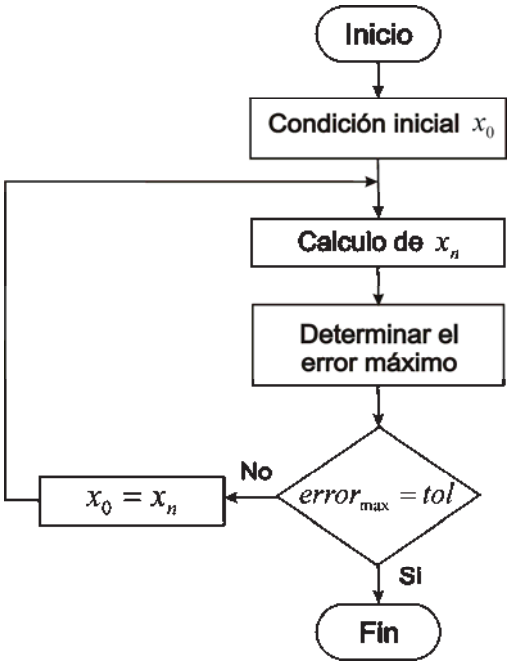


Figura 4.30 Diagrama de flujo del método de FB.

4.3.4.1 Método de Runge-Kutta.

Una primera aproximación a la solución se encuentra con el método de Fuerza Bruta. En este trabajo de Tesis se utilizó el método de Runge-Kutta de cuarto orden [Gerald 2000]. Este método es común para obtener soluciones aproximadas al problema de valor inicial $y' = f(y,t)$, con $y(0) = y_0$. Existen métodos de Runge-Kutta de distintos órdenes, los cuales se deducen a partir del desarrollo de $y(x_n + h)$ en series de Taylor con residuo, esto es:

$$\begin{aligned} y(x_{n+1}) &= y(x_n + h) \\ &= y(x_n) + hy'(x_n) + \frac{h^2}{2!}y^n(x_n) + \frac{h^3}{3!}y^m(x_n) + \ldots + \frac{h^{k+1}}{(k+1)!}y^{(k+1)}(Cn) \end{aligned} \quad (4.24)$$

donde *Cn* es una constante entre x_n, y $x_n + h$. Cuando $k = 1$ y el residuo de $\frac{h^2}{2}y''$ es pequeño, y se obtiene la fórmula conocida de iteración:

$$y_{n+1} = y_n + hf(x_n, y_n) \quad (4.25)$$

Runge-Kutta de cuarto orden

El método de Runge-Kutta de cuarto orden (RK4) consiste en la integración de (4.6) sobre un periodo de tiempo y determinar las constantes adecuadas para la fórmula:

$$y_{n+1} = y_n + \alpha k_1 + \beta k_2 + ck_3 + dk_4 \quad (4.26)$$

en que

$$k_1 = hf(x_n, y_n) \quad (4.26a)$$
$$k_2 = hf(x_n + \alpha_1 h\ ,\ y_n + \beta_1 k_1) \quad (4.26b)$$
$$k_3 = hf(x_n + \alpha_2 h\ ,\ y_n + \beta_2 k_1 + \beta_3 k_2), \quad (4.26c)$$
$$k_4 = hf(x_n + \alpha_3 h\ ,\ y_n + \beta_4 k_1 + \beta_5 k_2 + \beta_6 k_3) \quad (4.26d)$$

coincidan con un polinomio de Taylor de cuarto grado. Con lo anterior se obtienen 11 ecuaciones con 13 incógnitas. Resolviendo el sistema, el conjunto de valores de las constantes produce el siguiente resultado [Nakamura 1997]:

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad (4.27)$$
$$k_1 = hf(x_n, y_n) \quad (4.28)$$
$$k_2 = hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1\right) \quad (4.29)$$
$$k_3 = hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2\right) \quad (4.30)$$
$$k_4 = hf(x_n + h, y_n + k_3) \quad (4.31)$$

donde y_n representa el vector de condiciones iniciales de las variables de estado, y *h* representa el paso de integración.

En esta Tesis, se programaron las ecuaciones del método RK4 usando la formación de [Ramos 2007], que además define el vector x_i de condición inicial, y x_f como vector solución final de RK4. En la Figura 4.31 se muestra el diagrama de flujo del algoritmo de RK4.

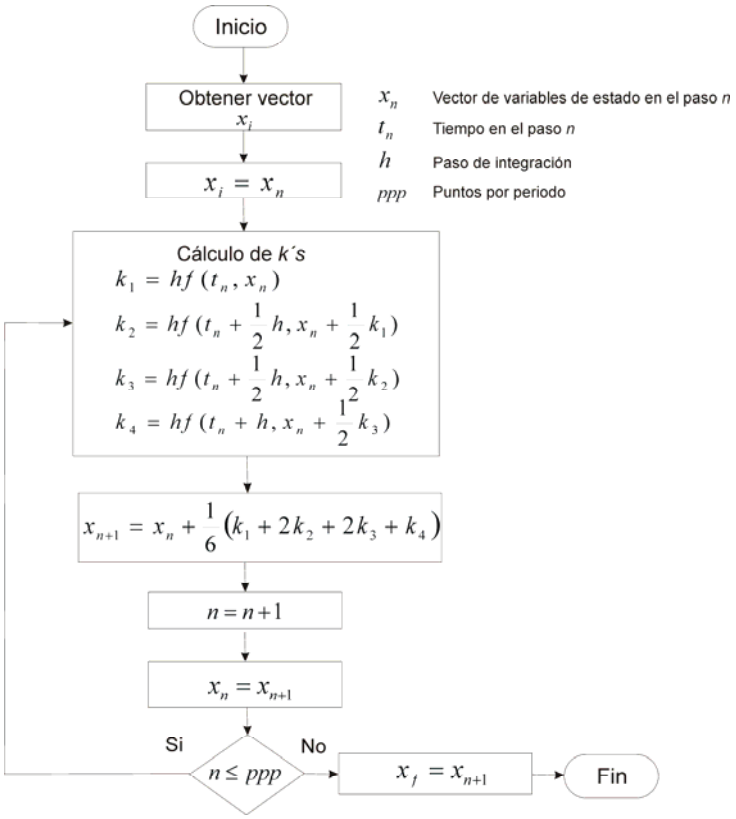


Figura 4.31 Diagrama de flujo del método de RK4.

El proceso que se sigue en la Figura 4.31 es el siguiente: Inicialmente x_i y x_n son ceros, y después de una primera integración se obtiene un valor x_{n+1}. Posteriormente, se introduce como condición inicial el valor de x_{n+1}. Este proceso se realiza tantas veces como puntos por periodo (*ppp*) se especifiquen. Obsérvese además que k_2 depende de k_1; k_3, de k_2, y k_4 de k_3. También, en k_2 y k_3 intervienen aproximaciones a la pendiente en el punto medio del intervalo entre x_n y x_{n+1}.

4.3.4.2 Propuesta de paralelización del método de Runge-Kutta

La paralelización del método de Runge-Kutta plantea realizar los cálculos de *k*1 hasta *k*4 de (4.27) dividiendo el total de las ecuaciones de estado en segmentos o grupos. Cada elemento de proceso calcula la constante en un grupo de ecuaciones, usando primero la función *evalua_K1_paralelo(), evalua_K2_paralelo(), evalua_K3_paralelo(),* y *evalua_K4_paralelo(),* en ese orden. La paralelización se resume a continuación:

1. Se indica a cada EP el inicio de su cálculo y cuantas ecuaciones va a computar según la información contenida en los vectores *inicia* y *cuantos*.
2. Considerando el vector de condiciones iniciales, cada EP hace la evaluación en la ecuación correspondiente a k_m, siendo m = 1,2,3 y 4.

3. El vector solución de k_m se envía a todos los EP; éste servirá como condición inicial en cálculo de la siguiente k, es decir, de k_{m+1}.
4. Se repiten los pasos 2 y 3, pero ahora considerando a k_{m+1}.
5. Finalmente se calcula el vector solución con (4.27).

El diagrama de flujo del método de paralelización de RK4 que se propone se muestra en la Figura 4.32.

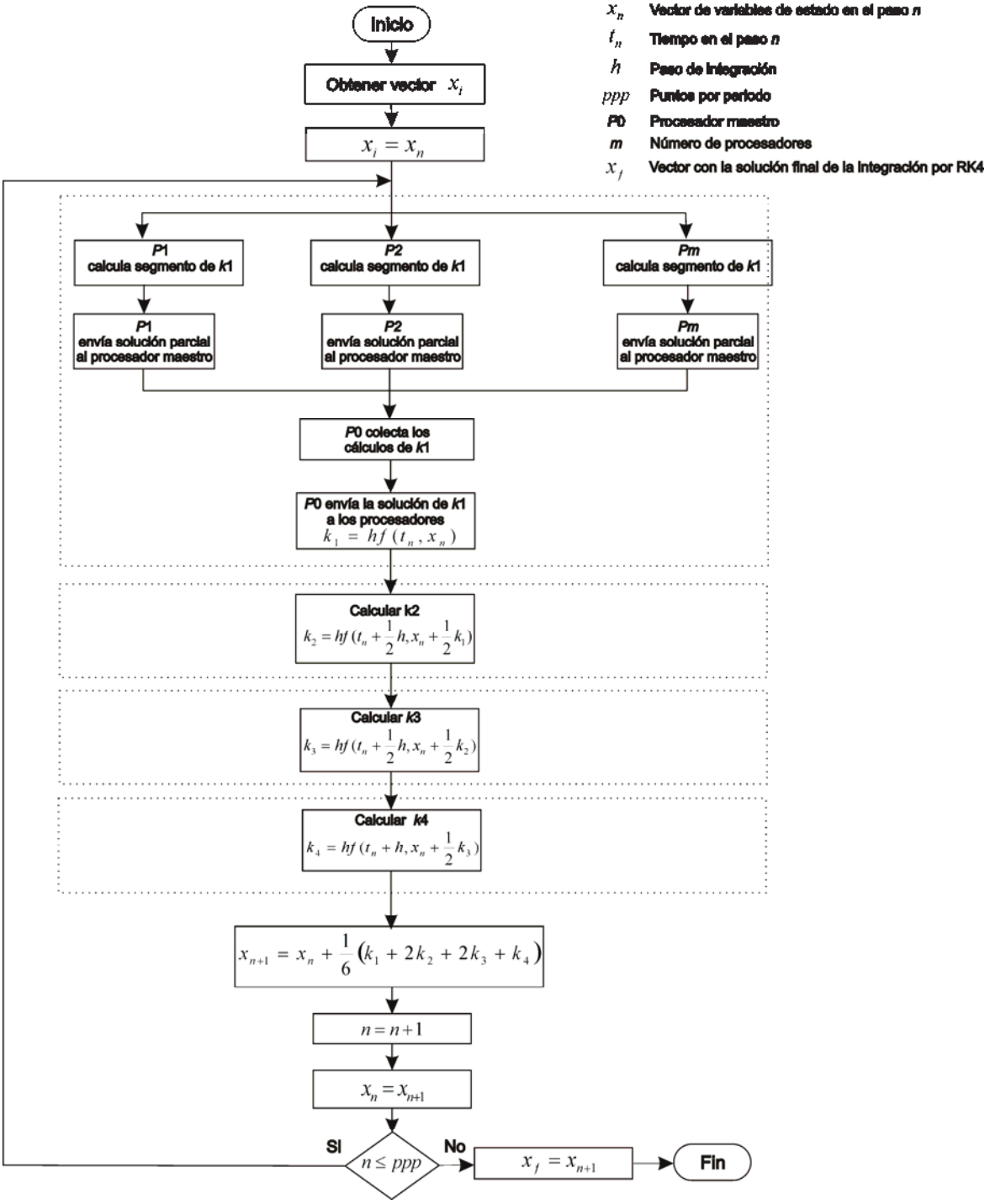


Figura 4.32 Diagrama de flujo del método de RK4 en paralelo.

Nótese que en la Figura 4.32 se presentan las etapas corresponden al cálculo de *k*2, *k*3 y *k*4 en paralelo contenidas entre líneas punteadas. El procedimiento es básicamente el mismo que se hace para *k*1, únicamente cambia la ecuación de evaluación de la constante *k* respectiva. En la Figura 4.33 se muestra la etapa del diagrama de flujo para el cálculo paralelo de *k*2.

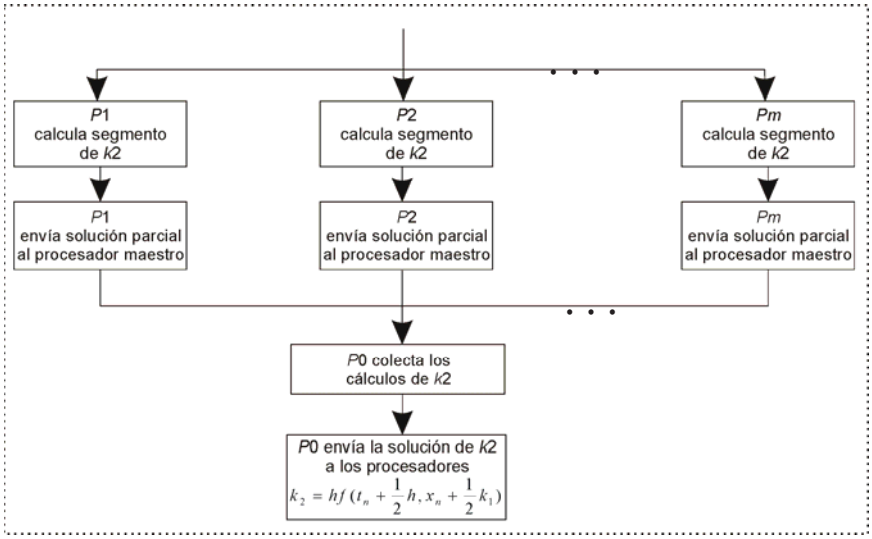


Figura 4.33 Etapa del cálculo en paralelo de *k*2.

El método paralelo de Runge-Kutta se efectúa en la función *Runge_Kutta_4_paralelo()*. El código se presenta en la Figura 4.34.

```
void Runge_Kutta_4_paralelo(double *y0){

  h=intervalo/hh;
  xn=t0+h;

        for(int i=0; i<Ve; i++)
                Yn1[i]=y0[i];

        for(int j=0; j<hh; j++){

                // ----------------- Calcula K1 -----------------
                evalua_K1_paralelo();
                MPI::COMM_WORLD.Gather(&Y_K1[inicia[my_id]],cuantos[my_id],MPI::DOUBLE,
                                       &Y_K1[inicia[my_id]],cuantos[my_id],MPI::DOUBLE,0);
                MPI::COMM_WORLD.Bcast(Y_K1,Ve,MPI::DOUBLE,0);

                // ----------------- Calcula K2 -----------------
                evalua_K2_paralelo();
                MPI::COMM_WORLD.Gather(&Y_K2[inicia[my_id]],cuantos[my_id],MPI::DOUBLE,
                                       &Y_K2[inicia[my_id]],cuantos[my_id],MPI::DOUBLE,0);
                MPI::COMM_WORLD.Bcast(Y_K2,Ve,MPI::DOUBLE,0);

                // ----------------- Calcula K3 -----------------
                evalua_K3_paralelo();
                MPI::COMM_WORLD.Gather(&Y_K3[inicia[my_id]],cuantos[my_id],MPI::DOUBLE,
                                       &Y_K3[inicia[my_id]],cuantos[my_id],MPI::DOUBLE,0);
                MPI::COMM_WORLD.Bcast(Y_K3,Ve,MPI::DOUBLE,0);

                // ----------------- Calcula K4 -----------------
                evalua_K4_paralelo();

                // ------------- Calcula Solucion --------------
                for(int i=0; i<Ve; i++)
                   Yn1[i]=Yn1[i]+h*(v_K1[i]+2*v_K2[i]+2*v_K3[i]+v_K4[i])/6;


                MPI::COMM_WORLD.Gather(&Yn1[inicia[my_id]],cuantos[my_id],MPI::DOUBLE,
                                       &Yn1[inicia[my_id]],cuantos[my_id],MPI::DOUBLE,0);
                MPI::COMM_WORLD.Bcast(Yn1,Ve,MPI::DOUBLE,0);
                xn=xn+h;
        }

        for(int i=0; i<Ve; i++)
                Y_sol[i]=Yn1[i];

}
```

Figura 4.34 Código de la función *Runge_Kutta_4_paralelo()*.

El procesamiento en paralelo que se propone en la función *Runge_Kutta_4_paralelo()* consiste en dividir la carga de trabajo que representa el cálculo de las constantes *k*1, *k*2, *k*3 y *k*4 de (4.13) entre los elementos de proceso o nodos con que se cuenta en la arquitectura paralela. La división se realizó al principio del programa con la función *reparte_procesos()* cuando se generaron los vectores *inicia* y *cuantos*.

En *Runge_Kutta_4_paralelo()* el vector *cuantos* almacena un número que corresponde a la cantidad de ecuaciones que se asignarán a cada elemento de proceso, mientras que el vector *inicia* determina una localidad dentro de la estructura *ec*.

Dado que los elementos de *ec* son apuntadores a las listas ligadas simples (equivalentes a las ecuaciones diferenciales ordinarias que representan la dinámica del sistema de estudio), el vector *inicia* aportará el número del elemento desde el cual se accede a una lista ligada simple.

La expresión *inicia[my_id]* apunta al elemento del vector *inicia* cuyo número es igual al de un elemento de proceso del entorno MPI. Lo mismo sucederá con el vector *cuantos.*

La expresión *inicia[my_id]+cuantos[my_id]* logra un desplazamiento para identificar una cantidad de ecuaciones que corresponde evaluar a cada elemento de proceso identificado por *my_id*.

De esta manera, con los índices de los vectores *cuantos* e *inicia*, se logra que cada elemento de proceso acceda a una parte de la estructura *ec* y, por ende, a las listas ligadas, lográndose así la repartición balanceada de tareas y datos. Posteriormente, cada elemento de proceso llamará a las funciones *evalua_K1_paralelo(), evalua_K2_paralelo(), evalua_K3_paralelo(),* y *evalua_K4_paralelo(),* en ese orden.

Runge_Kutta_4_paralelo() tiene como parámetro un apuntador al vector de condiciones iniciales *y*0. En esta función se calcula el paso de integración *h* para el método de Runge-Kutta; la variable *xn* es el incremento en el periodo considerando el tiempo inicial *t*0. Se hace la operación equivalente a $y(x_0) = y_0$ al pasar los valores de *y*0 al vector *Yn*1 (*Yn*1 es el equivalente a *yn* de (4.13). Durante el periodo, se inicia el cálculo de las constantes de (4.13) tantas veces como puntos por periodo se hayan considerado en *hh*.

*k*1 se obtiene invocando a la función *evalua_K1_paralelo()*. El código de ésta función se muestra en la Figura 4.35. *v_K*1 es un vector que almacena la evaluación en los terminos de cada ecuación; el vector *Y_K*1 tiene la solución de (4.14) y que servirá como condición inicial en el cálculo de *k*2.

```
void evalua_K1_paralelo(void){
     double suma;

        for(int i=inicia[my_id]; i<inicia[my_id]+cuantos[my_id]; i++){
                suma=suma_ecuacion(ec[i],Yn1,xn);
                v_K1[i]=suma;
                Y_K1[i]=Yn1[i]+h*suma/2.0;
        }
}
```

Figura 4.35 Código de la función *evalua_K1_paralelo().*

Una vez que los elementos de proceso terminan el cálculo de *k*1 en su respectivo grupo de ecuaciones, el programa paralelo solicita las funciones *Gather* y *Bcast* de MPI. *Gather* recolecta los resultados de todos los elementos de proceso y posteriormente *Bcast* difunde dichos resultados a todos los elementos de proceso para continuar con el cómputo de *k*2. El mismo procedimiento ocurre para el resto de las *k´s*. El código de *evalua_K2_paralelo(), evalua_K3_paralelo()*, y *evalua_K4_paralelo()* se muestra en la Figura 4.36.

Para *k*2
```
suma=suma_ecuacion(ec[i],Y_K1,(xn+h/2));
v_K2[i]=suma;
Y_K2[i]=Yn1[i]+h*suma/2.0;
```

Para *k*3
```
suma=suma_ecuacion(ec[i],Y_K2,(xn+h/2));
v_K3[i]=suma;
Y_K3[i]=Yn1[i]+h*suma;
```

Para *k*4
```
suma=suma_ecuacion(ec[i],Y_K3,(xn+h));
v_K4[i]=suma;
```

Figura 4.36 Código de las funciones para calcular *k*2, *k*3 y *k*4.

La función *suma_ecuacion()* regresa la suma total originada al evaluar todos los términos de una ecuación con las condiciones iniciales de *vec_ini*. El código se observa en la Figura 4.37.

```
double suma_ecuacion(ecuacion &_ec, double *vec_ini, double vs){

   double suma=0;
   termino *aux=_ec.primero.siguiente;

   while(aux){
        suma=suma+aux->coef*pow(vec_ini[aux->ve],aux->exp);
        aux=aux->siguiente;
   }

  for(int i=0; i<_ec.num_fuentes; i++)
        suma = suma + sin(w*vs)*_ec.fuente[i].magnitud;

  return suma*w;
}
```

Figura 4.37 Código de la función *suma_ecuacion().*

La función *suma_ecuacion()* tiene como parámetros a *_ec*, que es la dirección de una ecuación del arreglo de listas ligadas; al vector de condiciones iniciales *vec_ini*, que corresponde al resultado de la *k* previa, excepto en el caso de *k*1; por último, a la constante *vs*, que representa $y(x_n + h)$ en el algoritmo de Runge-Kutta. El objeto *aux* ubica al primer elemento de tipo *termino* de la lista ligada simple apuntada por *ec_*. Posteriormente, se recorre la lista evaluando cada uno de los *terminos* con las condiciones iniciales de *vec_ini*, y almacenando el resultado en la variable *suma*. Finalmente, se suman las fuentes de potencia asociadas; la función regresa el resultado total en *suma*.

4.3.5 Paralelización del método para calcular la matriz de identificación Φ.

En base a las ecuaciones (4.7) y (4.8), [Semlyen y Medina 1995], se determina la matriz *C* que relaciona las variables de estado en el Ciclo Límite. Considerando que la matriz Φ es de orden $n \times n$, donde *n* es el número de variables de estado, Φ tendrá *n* columnas, cuyos elementos se pueden obtener calculando columna por columna, aplicando una perturbación secuencial en las variables de estado en la forma $x_i + \xi e_i$, donde x_0 es el vector de variables de estado, ξ es un número pequeño, el orden de 1×10^{-6} *p.u.* y e^i es la columna *i* de la matriz identidad *I*. El diagrama de flujo del algoritmo secuencial del cálculo de Φ y la representación esquemática de sus columnas se observa en la Figura 4.38.

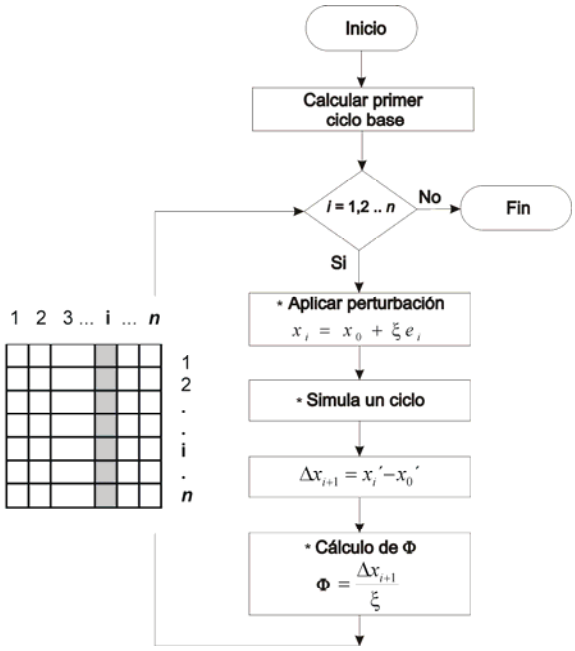


Figura 4.38 Diagrama de flujo del método para calcular la matriz de identificación Φ.

4.3.5.1 Propuesta de paralelización para calcular la matriz de identificación Φ

Para el cálculo de las columnas de la matriz de identificación Φ, esta Tesis se basa en el procedimiento indicado en [García *et al*. 2001]. Los pasos para aplicar el procesamiento paralelo mediante el cluster propuesto son:

1. Determinar el número de columnas de la matriz Φ que serán calculadas por cada uno de los EP.
2. Indicar a cada EP el inicio de su cálculo y cuantas columnas de Φ va a calcular. según la información contenida en los vectores *inicia* y *cuantos*.
3. Considerando el vector de variables de estado en el Ciclo Base, cada EP realiza la DN sobre el conjunto de ecuaciones que le pertenece.
4. Encontrar el vector solución que corresponde a la *j-ésima* columna de Φ.
5. Finalmente, una vez que cada EP termina de calcular la(s) columnas de la matriz Φ, éstas se envían al procesador maestro con el objeto de continuar el proceso de solución.

En la figura 4.39 se muestra el diagrama de flujo que se propone para calcular la matriz de identificación Φ en paralelo.

La matriz de identificación Φ se calcula en la función *calcula_Phi_paralelo()*. Está función hace uso nuevamente de los valores de los vectores *inicia* y *cuantos* obtenidos en *reparte_procesos()*. También recibe como condición inicial el vector de variables de estado en el Ciclo Base.

De manera general existen tres posibilidades de relación entre el número de variables de estado *Ve* y el número *m* de EP:

- $m < Ve$ El número de EP es menor al número de las variables de estado del caso de estudio. Cada EP calcula por lo menos una columna de la matriz de identificación.
- $m = Ve$ Caso ideal. El número de EP es igual al número de las variables de estado del caso de estudio. Aquí cada EP calculará solamente una columna de Φ.
- $m > Ve$ El número de EP es mayor al número de las variables de estado.

En *calcula_Phi_paralelo()*, cada EP toma de *inicia* la posición de la primera columna que va a determinar, y de *cuantos* el número de columnas que le corresponde calcular; de antemano cada EP conoce el rango de columnas que va a encontrar. Posteriormente, para cada variable de estado, el EP aplica la perturbación ξ de 1×10^{-6} *p.u.*, acción equivalente a $x_1 = x_0 + \xi e_1$

A continuación el programa hace una aproximación al Ciclo Límite simulando un ciclo extra, esto es, aplicando una sola vez el método de FB (función *Runge_Kutta_4_secuencial*. Hecha la aproximación con perturbación se calcula la diferencia entre los dos valores de *x* al final del ciclo, obteniéndose las columnas Δx_{i+1}. Después, realizando $\frac{\Delta x_{i+1}}{\xi}$ se obtiene la columna de la matriz Φ. Una vez que los EP completan la solución de las columnas de Φ, se utilizan las funciones *Gather* y *Bcast* de MPI para recolectar las columnas y se enviarlas al EP maestro [García 2005]. En la Figura 4.40 se presenta el código de la función *calcula_Phi_paralelo()*.

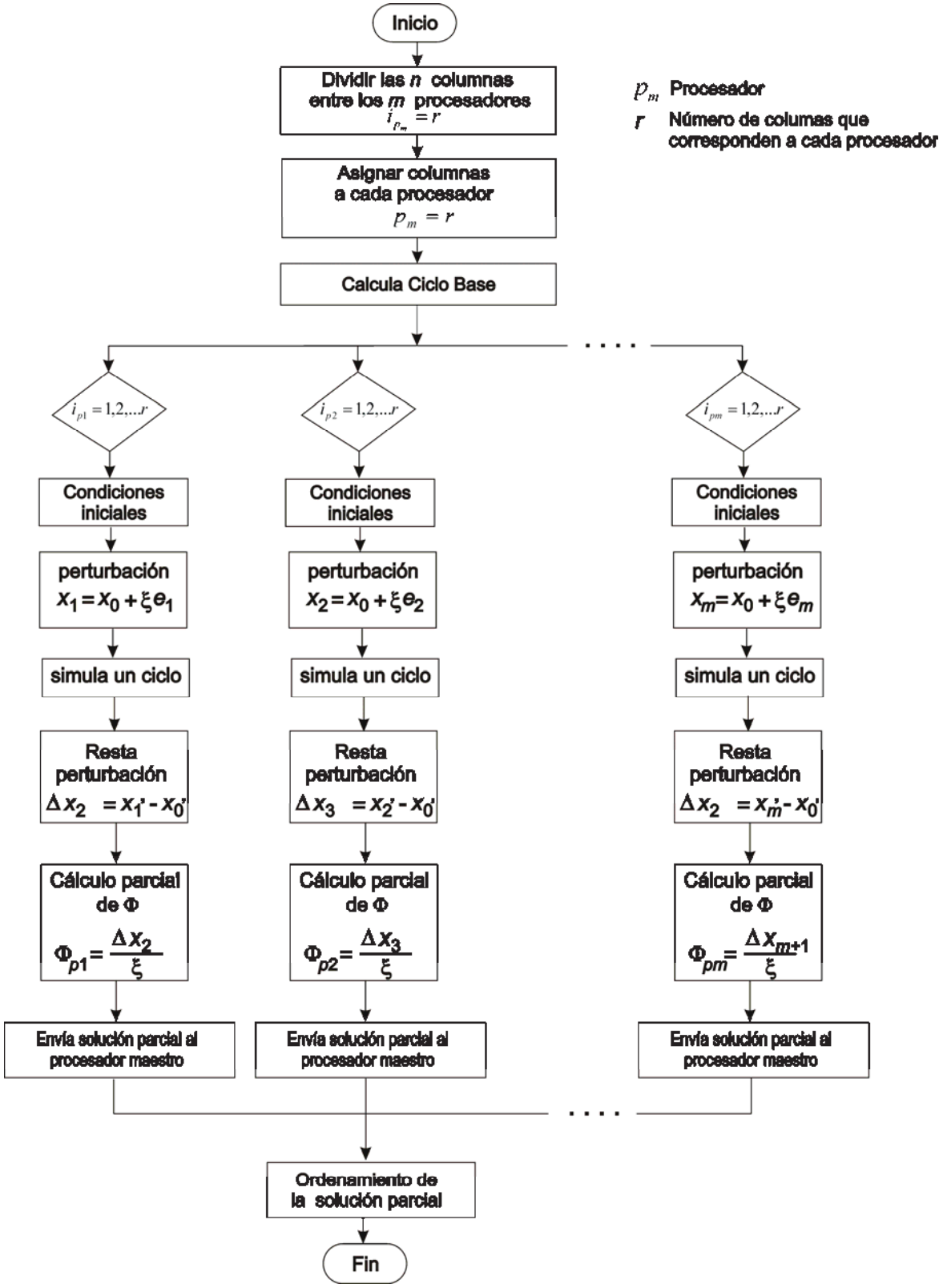


Figura 4.39 Diagrama de flujo del método para calcular la matriz de identificación Φ en paralelo.

```
void calcula_Phi_paralelo(void){

        double * Yn2;
        double * Yn3;
        Yn2=new double [Ve];
        Yn3=new double [Ve];
        phi=new double[Ve*Ve];

        for(int j=0; j<Ve; j++){ Yn2[j]=Y_sol[j]; }

        for(int i=inicia[my_id]; i<inicia[my_id]+cuantos[my_id]; i++){
                for(int j=0; j<Ve; j++){
                        Yn1[j]=Y_ini[j];
                }
                Yn1[i]=Yn1[i]+epsilon;
                Yn3=Runge_Kutta_4_secuencial(Yn1);
                for(int j=0; j<Ve; j++){
                        phi[i*Ve+j]=(Yn3[j]-Yn2[j])/epsilon;
                }
        }

        MPI::COMM_WORLD.Gather(&phi[inicia[my_id]*Ve],cuantos[my_id]*Ve,MPI::DOUBLE,
                               &phi[inicia[my_id]*Ve],cuantos[my_id]*Ve,MPI::DOUBLE,0);
        MPI::COMM_WORLD.Bcast(phi,Ve*Ve,MPI::DOUBLE,0);
}
```

Figura 4.40 Código de la función *calcula_Phi_paralelo()*.

4.3.6 Paralelización del procedimiento para cálculo de la matriz C.

Para obtener la matriz *C* de (4.8) es necesario calcular $(I-\Phi)^{-1}$. Una alternativa para calcular la inversión matricial es el método de *Descomposición LU de Doolittle* y *Crout* [Chapra 2003] en el cual se hace la descomposición triangular inferior y superior equivalente de la matriz base, como se indica en (1.2), (1.3) y (1.4); posteriormente se obtienen las columnas de la matriz inversa aplicando un proceso de *sustitución hacia delante* y *sustitución hacia atrás* del método de Gauss. Este proceso es equivalente a resolver un sistema de la forma *Ax = b*, en donde se si se resuelven *n* sistemas con una misma matriz de coeficientes se encuentran los *n* vectores solución de *x*, que corresponden justamente a las columnas de la matriz A^{-1} (la explicación a detalle del método de Doolittle y Crout se puede consultar en el Apéndice B).

El cálculo de la matriz C puede obtenerse por columnas. si se realiza un proceso de sustitución hacia delante y hacia atrás sobre la descomposición *LU* de $(I-\Phi)$ hecha con el método de Doolittle y Crout. El procedimiento consiste en descomponer primeramente la matriz $(I-\Phi)$ como el producto de una matriz triangular inferior *L* por una matriz triangular superior *U* de la forma:

$$(I-\Phi) = LU \tag{4.32}$$

donde:

$$l_{i,1} = a_{i,1} \qquad \text{para } i = 1, 2, \ldots, n \tag{4.32a}$$

$$u_{i,1} = \frac{a_{i1}}{l_{11}} \qquad \text{para } i = 1, 2, \ldots, n \tag{4.32b}$$

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}\ , \qquad \text{para } 1 \le i \le j \le n \tag{4.32c}$$

$$u_{ij} = \frac{1}{l_{ii}}\left(a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right), \qquad \text{para } 2 \le j < i \le n \tag{4.32d}$$

Una vez que se tiene la matriz $(I - \Phi)$ expresada como un producto de dos matrices *LU*, la matriz inversa se obtiene realizando la sustitución hacia delante y hacia atrás (eliminación de Gauss convencional). Sin embargo, la ventaja de emplear la descomposición *LU* de Doolittle y Crout *vs*. el método de Gauss convencional es que la descomposición *LU* tiene un orden de complejidad menor, lo cual representa un menor esfuerzo computacional, además de la mitad de requerimientos de memoria [Asenjo 1997].

Como se ha dicho, la inversa de la matriz *C* se puede calcular a partir de una descomposición *LU*, columna por columna, generando soluciones con vectores unitarios [Chapra 2003]. Considérese que el producto *LU* resuelve el problema:

$$LUx=b \tag{4.33}$$

donde *x* representa un vector de términos independientes y *b* es el vector solución.

Si *b* es un vector de ceros, donde únicamente su primer elemento tiene valor de 1, y se resuelve a través de un proceso de sustitución hacia atrás y hacia adelante, se encontrará el vector *x* el cual contendrá la primera columna de la matriz *C*.

Para encontrar las siguientes columnas la matriz *C* se repite el proceso de sustitución hacia adelante y hacia atrás, pero utilizando ahora el vector *b* con elementos ceros y solamente un elemento 1 en la posición asociada con la columna de *C* que se va a calcular. En base a este proceso, pueden deducirse de forma paralela todas las columnas de *C*, ya que éstas se pueden obtener de manera independiente. Todo el proceso se muestra en el diagrama de flujo de la Figura 4.41.

El cálculo de la matriz inversa no es el mejor método para resolver un sistema de ecuaciones. Lo mejor sería resolver directamente el sistema de ecuaciones y evitar la inversión matricial. No obstante, en esta Tesis se hace la paralelización de la inversión matricial para demostrar que es factible paralelizar dicho método.

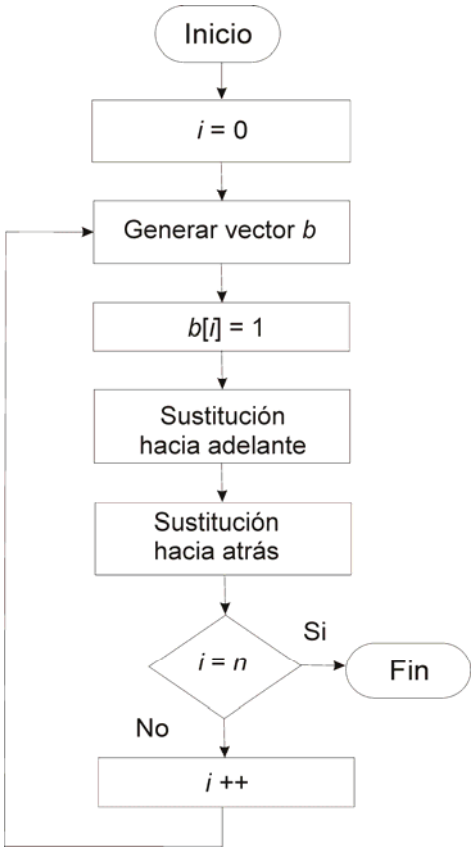


Figura 4.41 Diagrama de flujo del método para calcular la matriz *C*.

4.3.6.1 Propuesta de paralelización del cálculo de la matriz inversa.

Para aplicar el procesamiento paralelo al método para calcular la matriz *C* de (4.8), primeramente se define la partición del problema, posteriormente se aplica el método de descomposición *LU* de Doolittle y Crout en $(I-\Phi)$, y por último se calculan en paralelo las columnas de *C* en base a (4.33). Los pasos para aplicar el procesamiento paralelo al cálculo de la matriz *C* de son:

1. Determinar el número de columnas de la matriz C que serán calculadas por cada uno de los EP. Si se tienen *m* EP para el cálculo de las *n* columnas de la matriz *C*, se pueden encontrar 3 posibilidades para la distribución de columnas:
   - $n > m$. Algunos EP realizarán el cálculo de más de una de las columnas de la matriz inversa.
   - $n = m$. Cada uno de los EP calcula una columna de la matriz inversa. Caso ideal.
   - $n < m$. Cada uno de los elementos de proceso realiza el cálculo de una columna de la matriz inversa y quedan EP sin operación.
2. Indicar a cada EP el inicio de su cálculo y cuantas columnas de la matriz inversa va a calcular.

3. Inicializar en cada EP un vector de ceros *b*. Colocar un 1 en la posición *i* que corresponde con el número de la primera columna de la matriz *C* que va a calcular el EP.
4. Realizar el proceso de sustitución hacia atrás y hacia adelante para encontrar el vector solución que corresponde a la *i-ésima* columna de la *C*.
5. Enviar al procesador maestro las columnas que fueron calculadas. Se obtiene la matriz *C*.

En la figura 4.42 se muestra el diagrama de flujo que se propone para calcular la matriz inversa aplicando procesamiento en paralelo.

La matriz inversa de $(I-\Phi)$ se calcula con las funciones *LU()* y *sustitucion()*. En la primera se realiza la descomposición *LU* de $(I-\Phi)$ de manera secuencial con el método de Doolittle y Crout. Aquí se tiene la ventaja de almacenar en la misma matriz *Phi* los elementos que se van calculando. La variable *factor* va tomando valores según (4.32b). La Figura 4.43 presenta el código de la función *LU()*.

La función *sustitución()* realiza el procedimiento de sustitución hacia atrás y hacia adelante para calcular las columnas de la matriz C con la introducción del vector *b* de (4.33). En esta función se emplean nuevamente los valores de los vectores *inicia* y *cuantos* obtenidos en la función *reparte_procesos()* para determinar las columnas de la matriz C que cada EP va a calcular, y posicionar el número 1 en el vector *b* según corresponda. Una vez terminado el cálculo de una columna, el resultado se almacena en el vector auxiliar *c*. Luego, mediante las funciones *Gather* y *Bcast* de MPI, cada EP envía al procesador maestro su columna calculada. Al final del proceso el procesador maestro tiene todas las columnas la matriz *C* de (4.8). En la Figura 4.44 se muestra el código de *sustitución()*.

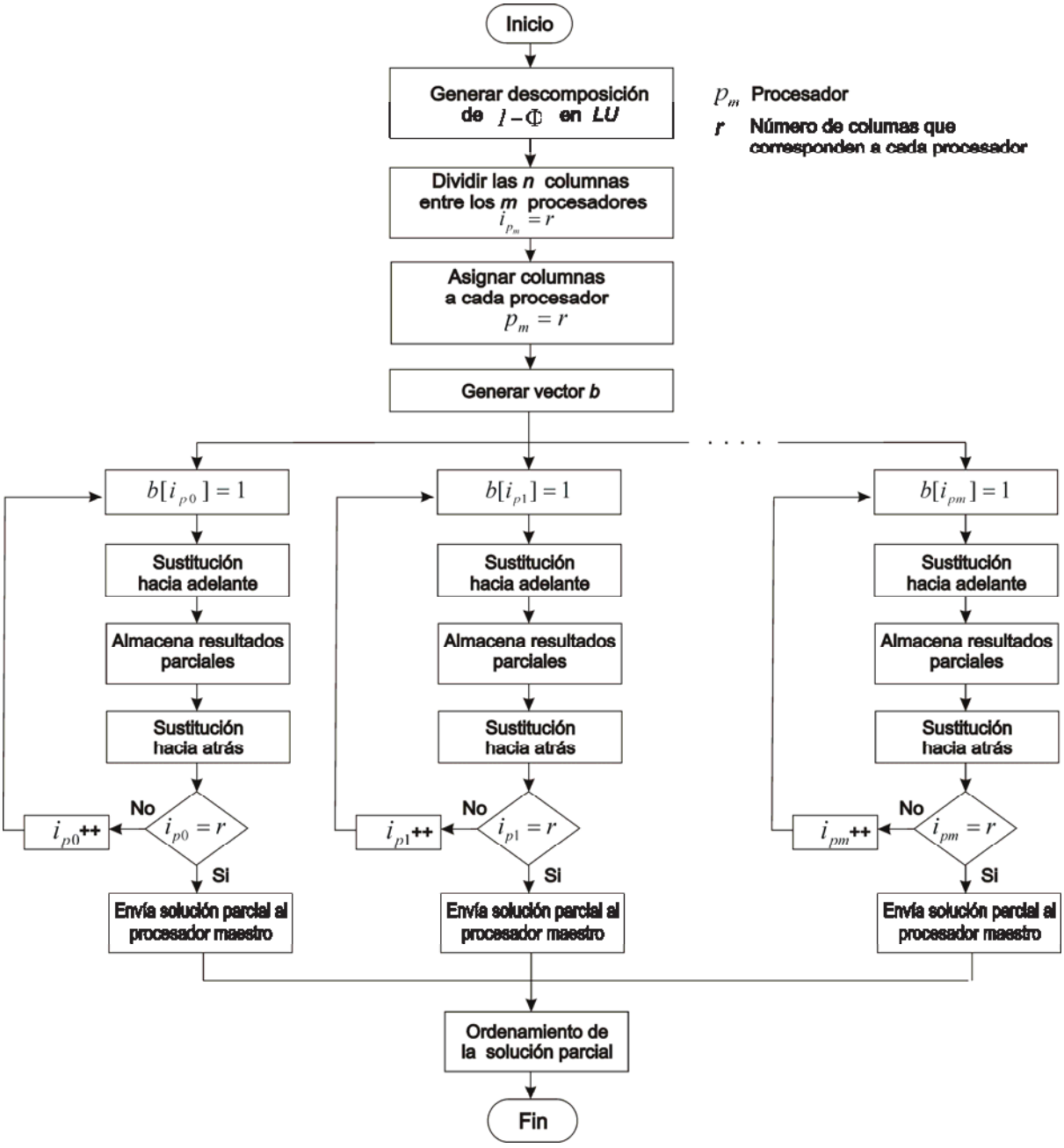


Figura 4.42 Diagrama de flujo del método para calcular *C* en paralelo.

```
void LU(){

        double factor;
        for (int k=0; k<Ve; k++){
                for (int i=k+1; i<Ve; i++){
                        factor=Phi[i*Ve+k]/Phi[k*Ve+k];
                        Phi[i*Ve+k]=factor;
                        for(int j=k+1; j<Ve; j++)
                                Phi[i*Ve+j]=Phi[i*Ve+j]-factor*Phi[k*Ve+j];
                }
        }

}
```

Figura 4.43 Código de la función *LU()*.

```
void sustitucion(){

        double suma;
        double *B, *X, *c;

        B = new double [Ve];
        X = new double [Ve];
        c = new double [Ve];
        for(int i=0; i<Ve; i++) {
                B[i]=0;  X[i]=0;
        }


        for(int inicio=inicia[my_id]; inicio<inicia[my_id]+cuantos[my_id]; inicio++){
                B[inicio]=1;

                //proceso de sustitucion hacia adelante
                for(int i=0; i<Ve; i++){
                        suma=B[i];
                        for(int j=0; j<i; j++)
                                suma=suma-Phi[i*Ve+j]*B[j];
                        B[i]=suma;
                }
                        X[Ve-1]=B[Ve-1]/Phi[Ve-1];

                //proceso de sustitucion hacia atras
                for(int i=Ve-1; i>=0; i--){
                        suma=0;
                        for(int j=i+1; j<Ve; j++)
                                suma =suma+Phi[i*Ve+j]*X[j];
                        X[i]=(B[i]-suma)/Phi[i*Ve+i];
                        c[i*Ve+inicio]=X[i];
                }

                for (int q=0; q<Ve; q++)
                        B[q]=0;
        }
        //envia la solucion a la matriz C

        MPI::COMM_WORLD.Gather(&c[inicia[my_id]*Ve],cuantos[my_id]*Ve,MPI::DOUBLE,
                               &C[inicia[my_id]*Ve],cuantos[my_id]*Ve,MPI::DOUBLE,0);
        MPI::COMM_WORLD.Bcast(C,Ve*Ve,MPI::DOUBLE,0);

        delete B;
        delete X;

}
```

Figura 4.44 Código de la función *sustitucion().*

4.4 CONCLUSIONES

En el presente Capítulo se mencionaron los cuatro factores principales a considerar en el diseño de algoritmos paralelos, mismos que fueron considerados en el planteamiento del algoritmo paralelo de DN. Se hizo una breve explicación del entorno LAM-MPI, así como de las funciones que se emplearon en la implementación del programa paralelo propuesto. Se describió además la arquitectura del cluster en el cual se realizaron los Casos de Estudio presentados en esta Tesis.

Por último, se describe el método DN de [Semlyen y Medina 1995] para la solución de sistemas eléctricos en el dominio del tiempo. Se expone la estructura lógica que se ha de utilizar para representar la dinámica de los sistemas eléctricos de estudio, así como el código desarrollado con POO. Se indica el esquema de paralelización para el algoritmo de DN, y la aplicación de procesamiento en paralelo MPI a los métodos de: a) Runge-Kutta 4º orden, b) Cálculo de la matriz de identificación Φ y c) Matriz Inversa de $(I-\Phi)$ para encontrar la matriz C. En cada caso se presenta un diagrama de flujo (tanto secuencial como paralelo) y se explica la metodología paralela empleada; se presenta también el código del programa paralelo que se ejecuta en MPI con el cluster propuesto.

CAPITULO 5

CASOS DE ESTUDIO

En este Capítulo se presenta la solución periódica en estado estacionario para tres Casos de Estudio consistentes en redes eléctricas no lineales, aplicando procesamiento en paralelo basado en MPI, mediante la arquitectura distribuida descrita en el Capítulo 4. En cada caso se indica además:

- El error máximo durante cada uno de los periodos completos de integración hasta alcanzar el estado estacionario periódico.
- El espectro armónico de las formas de onda del sistema. El espectro armónico se obtuvo por medio de la aplicación de la Transformada Rápida de Fourier.
- Las formas de onda relacionadas con el sistema de ecuaciones diferenciales ordinarias que describen el comportamiento dinámico de cada caso.
- Comparación de tiempos de procesamiento entre las plataformas paralelas MPI y PVM.
- Comparación de la Eficiencia y Speed up del procesamiento en paralelo de las plataformas paralelas MPI y PVM.

Los elementos que conforman los sistemas eléctricos presentados como Casos de Estudio en esta Tesis se han modelado siguiendo la representación monofásica de elementos de [Ramos 2007] para líneas de transmisión, generadores, bancos de capacitores y ramas magnetizantes, mismos que se describen en el Apéndice A.

La estimación del tiempo de ejecución de cada Caso de Estudio se obtiene con la función de *MPI::Wtime()* de MPI, incluida en el código del programa principal. Las gráficas de convergencia y el espectro armónico se realizaron en MatLab V7 [Nakamura 1997], mientras que los Ciclos Límite presentados se obtuvieron con el graficador GNUPLOT [GNUPLOT 2006].

El cálculo de Speed-up y Eficiencia se hicieron en base a las Ecuaciones (3.1) y (3.2) de la Sección 3.1.

Las ecuaciones que describen la dinámica de los Casos de Estudio 1 y 2 se presentan en el Apéndice C. Las ecuaciones que describen la dinámica del Caso de Estudio 3 fueron obtenidas empleando la herramienta digital desarrollada por [Ramos 2007].

Para validar la solución en EEP de los Casos de Estudio, se ejecutó tres veces el programa paralelo que se propone en esta Tesis por cada Caso de Estudio y con uno, dos, tres y cuatro EP. Para obtener el valor final de cada solución por cada Caso de Estudio se descartó el resultado de la primera ejecución y se promediaron las dos restantes. Esto se hizo para evitar posibles variaciones en el tiempo de ejecución debidos a los procesos activos propios del sistema operativo y del entorno paralelo que son ajenos al algoritmo paralelo de DN que se propone en esta Tesis.

Durante la experimentación con el algoritmo paralelo de DN que se propone en esta Tesis, se encontró que el tiempo de ejecución del método de RK4 paralelo es mucho mayor con respecto al el tiempo de ejecución secuencial, y que incluso utilizando un solo EP el tiempo de ejecución no coincide con el método RK4 secuencial.

Se observó que conforme aumenta el número de EP, se incrementaba el tiempo de ejecución del algoritmo paralelo de DN propuesto en forma exponencial. En base a este comportamiento se puede decir que el tiempo de ejecución que le toma a cada uno de los EP hacer los cálculos de las constantes k_1, k_2, k_3 y k_4 de (4.27) es mayor que el tiempo de evaluación de las ecuaciones diferenciales con la función *suma_ecuacion()*. El coste de tiempo se debe principalmente al empleo de las funciones colectivas (*Gather*), que presentan latencias debidas a mecanismos de paralelización, sincronización y transmisión de datos entre los EP. El problema se agrava cuando el cálculo de las *k´s* se ejecuta tantas veces como *ppp* (puntos por periodo) se especifican en el método RK4 paralelo (ver Figura 4.32); el tiempo de las llamadas y evaluación de las funciones *evalua_K1_paralelo(), evalua_K2_paralelo(), evalua_K3_paralelo(),* y *evalua_K4_paralelo()* tienen un coste que se multiplica *ppp* veces.

Se llego a la conclusión de que paralelizar el método de RK4 de la manera que se propone en esta Tesis no es buena estrategia, y por tal razón, se decidió no incluir la paralelización de RK4 en el algoritmo paralelo de DN con el cual se obtuvo la solución de los Casos de Estudio.

5.1. CASO DE ESTUDIO 1

Este Caso de Estudio consiste en un sistema eléctrico monofásico formado por tres líneas de transmisión, un generador, dos capacitores y dos ramas magnetizantes. La dinámica del Caso de Estudio 1 está representada por un conjunto de siete EDO´s indicadas en el Apéndice C. Las líneas de transmisión se representan por medio de ramas *R-L*, en tanto que el efecto de saturación las ramas magnetizantes es modelado por medio de una función polinomial; finalmente, el generador se representa por medio de una función senoidal de 1.0 *p.u*. de amplitud.

El efecto de saturación en las ramas magnetizantes se expresa por $i(\lambda) = \lambda^n$ con n=5. En la Figura 5.1 se presenta el diagrama del Caso de Estudio 1, y en la Tabla 5.1 los parámetros del sistema eléctrico.

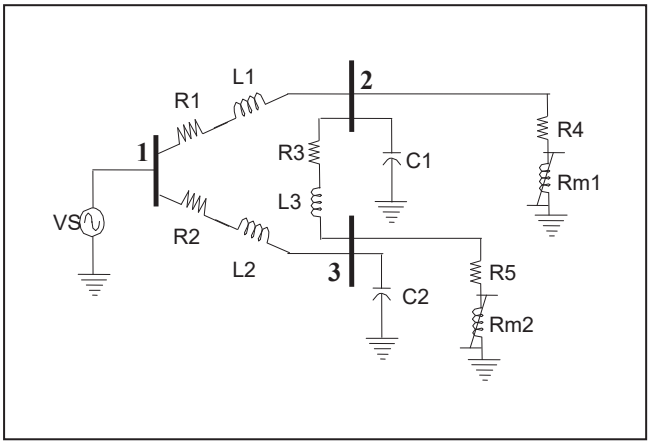


Figura 5.1 Caso de Estudio 1: sistema eléctrico monofásico de tres nodos.

Tabla 5.1 Parámetros del Caso de Estudio 1.

Parámetros		
R1 = 0.01	L1 = 0.1	C1 = 0.1
R2 = 0.01	L2 = 0.1	C2 = 0.1
R3 = 0.01	L3 = 0.1	
R4 = 0.01		
R5 = 0.01	VS = 1.0 p.u.	
Rm1=0.1		
Rm2=0.1		

5.1.1. Solución periódica en estado estacionario.

El Ciclo Límite y el EEP se logran una vez que la diferencia entre dos estimaciones sucesivas de x^{∞} es menor o igual a 1×10^{-10} *p.u*. Para la aplicación del método de DN se consideró que la magnitud de la pertubarción ξ es de 1×10^{-6} *p.u*.

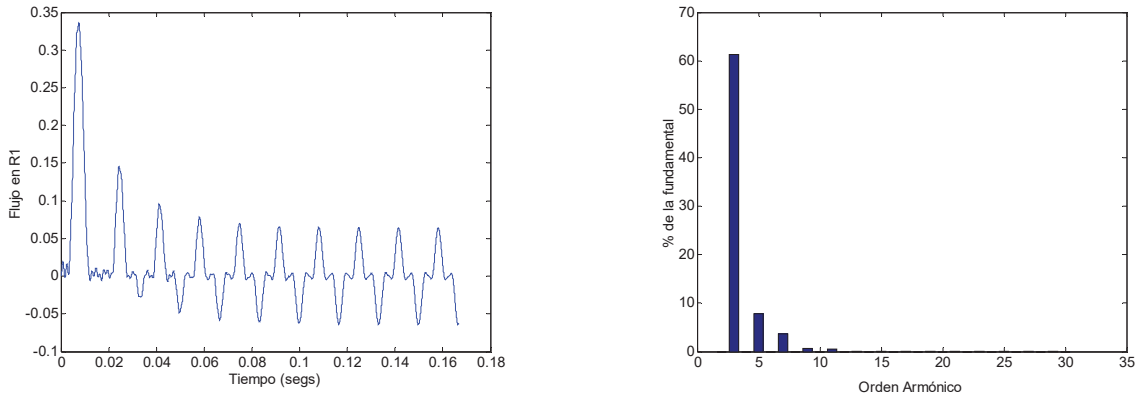
La Tabla 5.2 muestra la comparación entre la solución obtenida con el método de la FB y la solución con el método de DN. Nótese que el método de FB requiere 56 ciclos completos de integración para llegar a la solución en EEP, mientras que en el método de DN requiere solamente 24 ciclos; esto representa el 42.85% del total del número de ciclos usado por el método FB.

Tabla 5.2 Proceso de convergencia para el Caso de Estudio 1.

Número de ciclos	Fuerza Bruta	DN
1	6.485728E-01	6.485728E-01
2	1.741550E-01	1.741550E-01
3	6.333074E-02	6.333074E-02
⋮	⋮	⋮
8	6.559818E-03	6.559818E-03
⋮	⋮	⋮
16	2.658692E-04	3.536997E-05
⋮	⋮	⋮
24	8.874373E-06	**1.854558E-13**
⋮	⋮	
56	**3.513803E-11**	

5.1.2. Espectro armónico y Ciclo Límite.

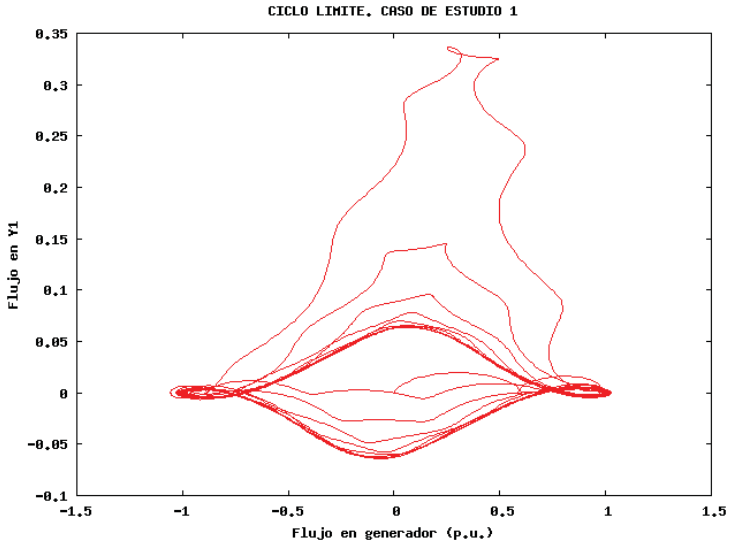
La Figura 5.2 ilustra el proceso de acercamiento al EEP y el contenido armónico de una de las variables de estado del Caso de Estudio 1. En la Figura 5.2 a) se muestra el flujo ψ_1, mientras que la Figura 5.2 b) se observa su espectro armónico. Se puede observar que la tercera armónica alcanza una magnitud aproximada del 60% de la fundamental, y la quinta armónica es aproximadamente un 10%. El alto contenido armónico de debe a que ψ_1 alimenta a un nodo que contiene una rama magnetizante (transformador de potencia) la cual presenta un comportamiento no lineal que introduce distorsión armónica.



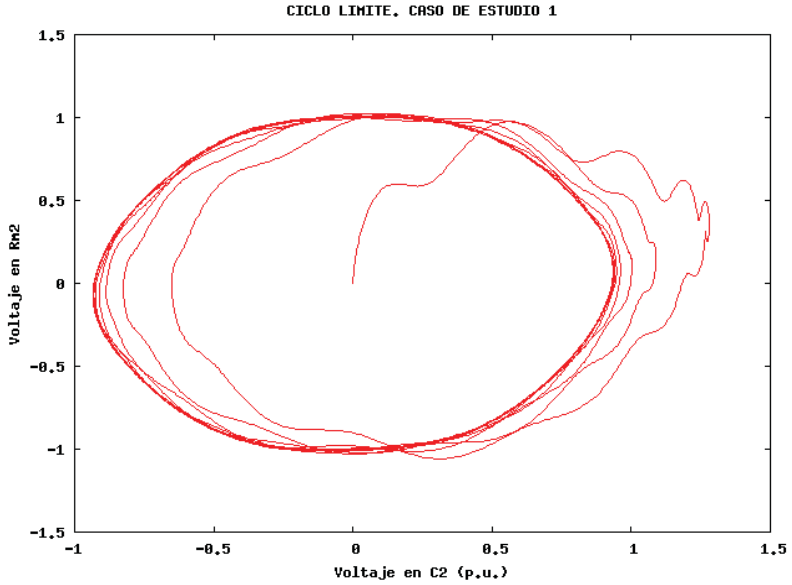
a) Flujo en Ψ_1. b) Componentes armónicas

Figura 5.2 Formas de onda en el EEP y su espectro armónico. Caso de Estudio 1.

En la Figura 5.3 a) y 5.3 b) se ilustra la convergencia al Ciclo Límite para las variables de estado. En la primera se graficó el Ciclo Límite de la variable de estado ψ_1 *vs Vs*; mientras que en 5.3b se grafica el voltaje V_{Rm2} *vs* $\dot{\psi}_{C2}$



a) Ciclo Límite considerando ψ_1 *vs Vs*



b) Ciclo Límite considerando voltaje *Rm2 vs* Ψ_{C2}

Figura 5.3 Ciclos límite del Caso de Estudio 1.

5.1.3. Análisis comparativo de solución secuencial y solución paralela.

En la Figura 5.4 se muestra la gráfica del comportamiento del tiempo de ejecución en función del número de EP del cluster. Se puede notar que el tiempo de ejecución, disminuye con una tendencia casi lineal a medida que aumenta el número de EP. La Eficiencia calculada para este Caso de Estudio 1 alcanza un máximo de 43.9% con MPI y 52% con PVM, según se observa en la Tabla 5.3.

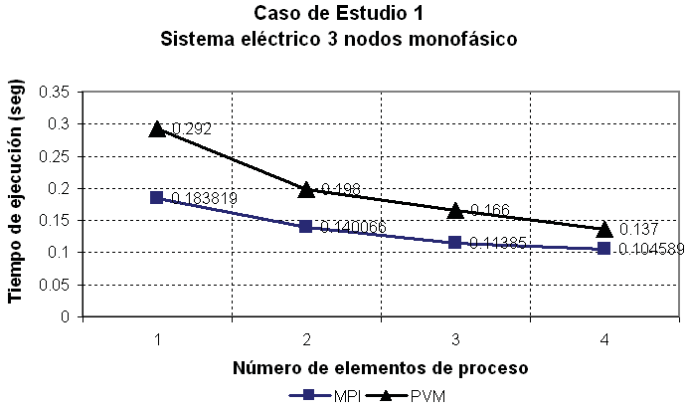


Figura 5.4 Relación Tiempo de ejecución-EP. Caso de Estudio 1

Tabla 5.3 Speed-up y Eficiencia para el Caso de Estudio 1.

Número de nodos	MPI		PVM	
	Speed-up	Eficiencia	Speed-up	Eficiencia
1	1	0.25	1	0.25
2	1.312374166	0.328093	1.474747	0.368686
3	1.615281195	0.403820	1.759036	0.439759
4	1.757536643	0.439384	2.119013	0.529753

En el Caso de Estudio 1, se observa que, aunque aumentó el número de EP, el Speed up, tanto con MPI como con PVM, no crece significativamente. Lo anterior se debe a que la parte secuencial del código del programa domina el rendimiento o ganancia de velocidad de cálculo según se explica en (3.6).

Dado que en problemas pequeños (como es el Caso de Estudio 1), casi todos los algoritmos tienen un comportamiento más o menos igual, la Eficiencia del algoritmo paralelo se manifestará siempre en problemas grandes [Pardo 2002], por lo que es de esperarse que en la medida que el tamaño del problema se incremente se observará con mayor claridad las ventajas del procesamiento en paralelo MPI, tanto en Speed up como en Eficiencia.

Se concluye que, para sistemas eléctricos pequeños, la cantidad de trabajo secuencial que hace el algoritmo paralelo da como resultado un comportamiento final *cuasi-secuencial* con MPI. Por su parte, PVM presenta mejor comportamiento en cuanto a Eficiencia, pero también presenta ganancia de velocidad muy pobre, ya que el Speed up calculado apenas llega a 2 con cuatro EP.

5.2. CASO DE ESTUDIO 2

El Caso de Estudio 2 es el sistema IEEE de 14 nodos modificado, consistente en quince líneas de transmisión, cinco generadores que se representan por medio de una función senoidal; 14 bancos de capacitores y tres ramas magnetizantes en los nodos 9, 13 y 14 del sistema, y que se han modelado por medio de una función polinomial. Las líneas de transmisión están representadas por medio de ramas *R-L*. La dinámica del Caso de Estudio 2 se representa por un conjunto de 32 EDO´s. El efecto de saturación las ramas magnetizantes se expresa por $i(\lambda) = \lambda^5$. En la Figura 5.5 se muestra el diagrama unifilar del Caso de Estudio 2 y en la Tabla 5.4 se muestran sus parámetros en *p.u*.

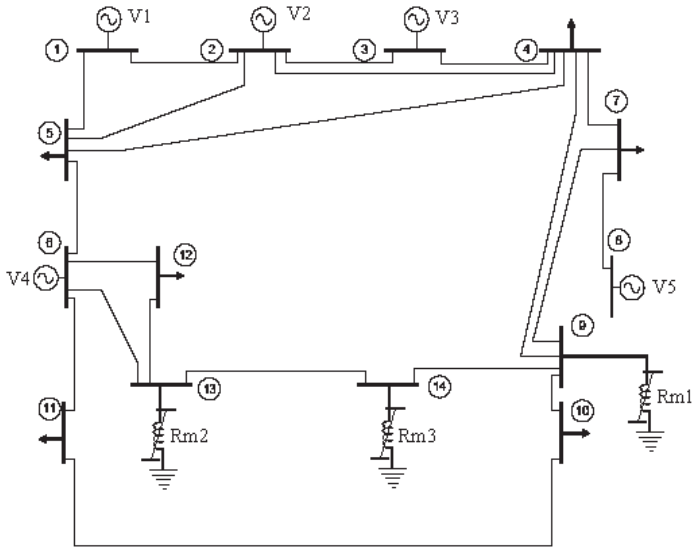


Figura 5.5 Caso de Estudio 2: sistema eléctrico IEEE modificado de14 nodos.

Tabla 5.4 Parámetros del Caso de Estudio 2.

Parámetros (p.u.)		
V1 = 1.06	R = 0.01	Rm1 = 0.1
V2 = 1.045	L = 0.1	Rm2 = 0.1
V3 = 1.01	C = 0.1	Rm3 = 0.1
V4 = 1.07		
V5 = 1.09		

5.2.1. Solución periódica en estado estacionario.

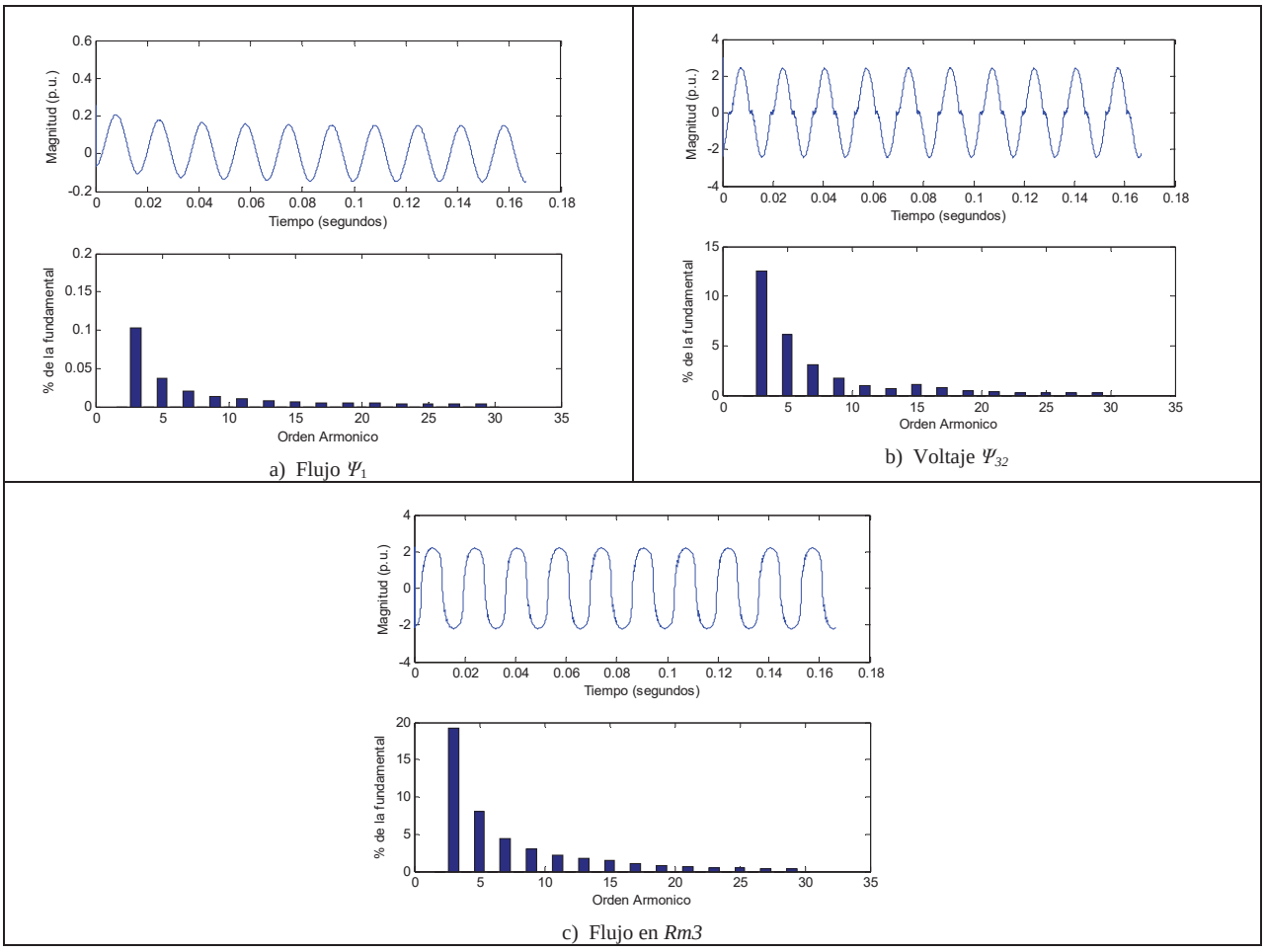
En el análisis del Caso de Estudio 2 se consideraron generadores de voltaje diferentes magnitudes, como se indica en la Tabla 5.4. La Tabla 5.5 muestra el número de periodos completos de integración necesarios para alcanzar el EEP; aquí también puede apreciarse que el método de FB requiere 63 periodos completos de integración, en tanto que el método de DN utiliza 41 periodos, lo cual representa el 65.07 % del número total de ciclos utilizados por el método FB.

Tabla 5.5 Proceso de convergencia para el Caso de Estudio 2.

Número de ciclos	Fuerza Bruta	DN
1	4.059774	4.059774
2	2.097462E-01	2.097462E-01
3	1.002306E-01	1.002306E-01
⋮	⋮	⋮
8	4.823064E-03	4.823064E-03
⋮	⋮	⋮
41	2.53967499E-04	**9.3258734E-15**
⋮	⋮	
63	**8.2591168 E-11**	

5.2.2. Espectro armónico y Ciclo Límite.

En la Figura 5.6 se observa la convergencia al Ciclo Límite y el contenido armónico de algunas variables de estado del Caso de Estudio 2. Las Figuras 5.6(a), 5.6(b) y 5.6(c) muestran las variables de estado $\dot{\psi}_1$, $\dot{\psi}_{32}$, y flujo en *Rm*3 respectivamente, y su contenido armónico.



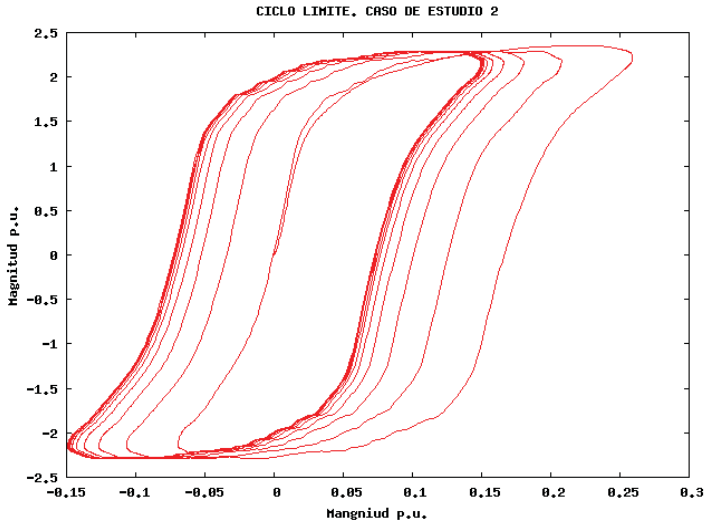
a) Flujo Ψ_1

b) Voltaje Ψ_{32}

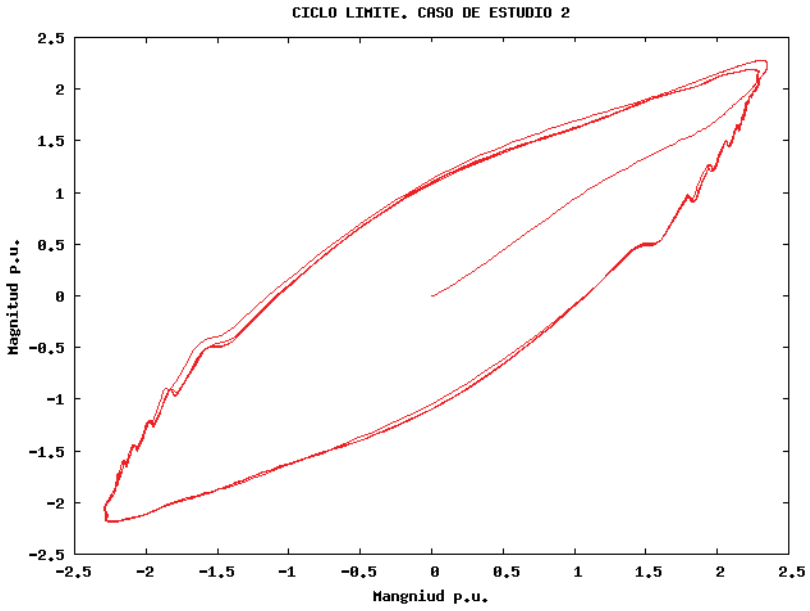
c) Flujo en *Rm3*

Figura 5.6 Formas de onda en el EEP y su espectro armónico. Caso de Estudio 2.

Para este caso en particular, únicamente la tercera armónica de Ψ_1 es de consideración (0.1%), siendo despreciables otros componentes armónicos, según se observa en la Figura 5.6 a). Para el caso de $\dot{\psi}_{32}$ (voltaje en V_{C4}) y $\Psi Rm3$, Figuras 5.6 b)- c), se observa que inyectan distorsión armónica ya se relacionan a componentes no lineales. Nótese los armónicos 3, 5, y 7 los de mayor magnitud respectivamente. Para este Caso de Estudio sería relevante considerar hasta la armónica 17, siendo 13, 7 y 3% para $\dot{\psi}_{32}$ (Figura 5.6 b)) y 19, 8 y 4% para $\Psi Rm3$, según se observa de la Figura 5.6 c). Ejemplos de Ciclo Límite se aprecian en la Figura 5.7.



a) Ciclo Límite considerando Ψ_1 *vs* $\Psi Rm3$



b) Ciclo Límite considerando $\Psi Rm2$ *vs* $\Psi Rm3$

Figura 5.7 Ciclo Límite del Caso de Estudio 2.

5.2.3. Análisis comparativo de solución secuencial y solución paralela.

El comportamiento del tiempo de ejecución en función del número de EP del cluster se puede observar en la Figura 5.8. Nótese que el tiempo de ejecución disminuye a medida que se incrementa el número de EP. El Speed up y la Eficiencia calculados se presentan en la Tabla 5.6. La Eficiencia alcanza un valor del 75.8% con MPI cuando se usan los 4 EP. Para el caso de PVM la Eficiencia alcanza un valor máximo de 45%.

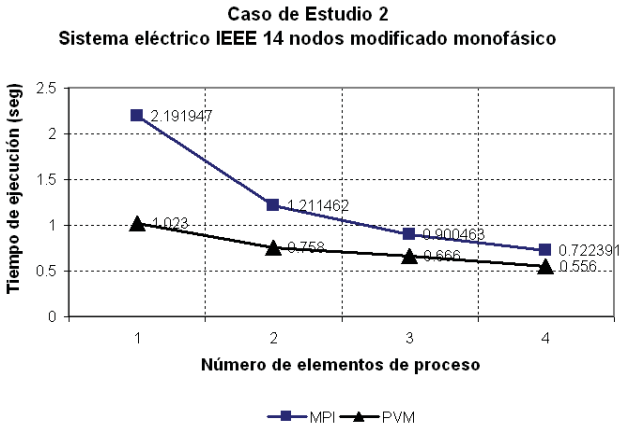


Figura 5.8 Relación Tiempo de ejecución-EP. Caso de Estudio 2.

Tabla 5.6 Speed-up y Eficiencia para el Caso de Estudio 2.

Número de nodos	MPI		PVM	
	Speed-up	Eficiencia	Speed-up	Eficiencia
1	1	0.25	1	0.25
2	1.809340	0.452335	1.349604222	0.337401055
3	2.434244	0.608561	1.536036036	0.384009009
4	3.034294	0.758573	1.839928058	0.459982014

El Caso de Estudio 2 contempla más EDO´s que el Caso de Estudio 1 para representar la dinámica del sistema eléctrico, por lo que demanda mayor coste computacional.

En la tabla 5.6 se muestra que al aumentar el número de EP se logra mejorar el rendimiento de velocidad en el trabajo computacional que se hace tanto con MPI como con PVM. Se observa que el máximo rendimiento o Speed up logrado con cuatro EP es 3, y se obtiene bajo la plataforma MPI. Lo anterior indica que el rendimiento en MPI presenta una ganancia casi lineal respecto al número de EP, y que es mucho mejor que la correspondiente en PVM.

En relación a la Eficiencia, en la Tabla 5.6, se observa que MPI presenta una mejora de aproximadamente 30% respecto a la que se obtiene con PVM. Lo anterior se debe a que el algoritmo propuesto y ejecutado en MPI presenta mejor calidad de paralelismo (según (3.4)) y designa más trabajo útil a cada EP. Con PVM se obtiene un pobre incremento del Speed up y de Eficiencia, y la tendencia es que, aún si se agregaran más de cuatro EP, la Eficiencia y Speed up no mejorarían significativamente.

Se concluye que, para el Caso de Estudio 2, la plataforma paralela MPI es mejor opción que PVM para ejecutar el algoritmo paralelo propuesto.

5.3. CASO DE ESTUDIO 3

El Caso de Estudio 3 que se analizó es el sistema de prueba de IEEE de 118 nodos modificado, está formado por 177 líneas de transmisión, 7 generadores y 177 bancos de capacitores y tiene componentes no lineales expresados por ramas magnetizantes. Este sistema es modelado por medio de un conjunto de 366 EDO´s. Cada línea de transmisión se representa por medio de ramas *R-L*. El efecto de saturación de las ramas magnetizantes se expresa como $i(\lambda) = \lambda^5$.

5.3.1. Solución periódica en estado estacionario.

En la Tabla 5.7 se muestra el número de ciclos (periodos completos) de integración que son necesarios para alcanzar el EEP por los métodos de FB y DN. En ésta Tabla se puede observar que el método de FB necesita 3356 periodos completos para llegar a la convergencia, en tanto que el método DN requiere 1109 ciclo, lo cual representa el 33.04 % del número total de ciclos requeridos por el método FB. Se observa también la característica de convergencia cuadrática del método de DN.

Tabla 5.7 Proceso de convergencia para el Caso de Estudio 3.

Número de Ciclos	Fuerza Bruta	DN
1	8.30654704E-01	8.30654704E-01
2	4.95644667E-01	4.95644667E-01
3	2.72416813E-01	2.72416813E-01
⋮	⋮	⋮
8	6.733132E-02	6.733132E-02
⋮	⋮	⋮
742	6.336288E-02	6.281720E-04
⋮	⋮	⋮
1109	5.274149E-02	**1.670330541E-13**
⋮		
3356	**9.810354E-11**	
Tiempo (seg.)	2.413625E06	2.499817E05

5.3.2. Espectro armónico y Ciclo Límite.

Las Figuras 5.9 a), 5.9 b) y 5.9 c) muestran voltajes o flujo, y su respectivo espectro armónico para el Caso de Estudio 3. Como puede observarse de la Figura 5.9 a), la tercera armónica alcanza una magnitud por arriba del 80% de la fundamental, y el valor de la quinta armónica cae por debajo del 20%. El alto contenido armónico de debe a que el flujo Ψ_{169} alimenta a un nodo que contiene componentes no lineales (rama magnetizante). Se observa que después de la 7ª armónica, los valores de los demás componentes armónicos son despreciables.

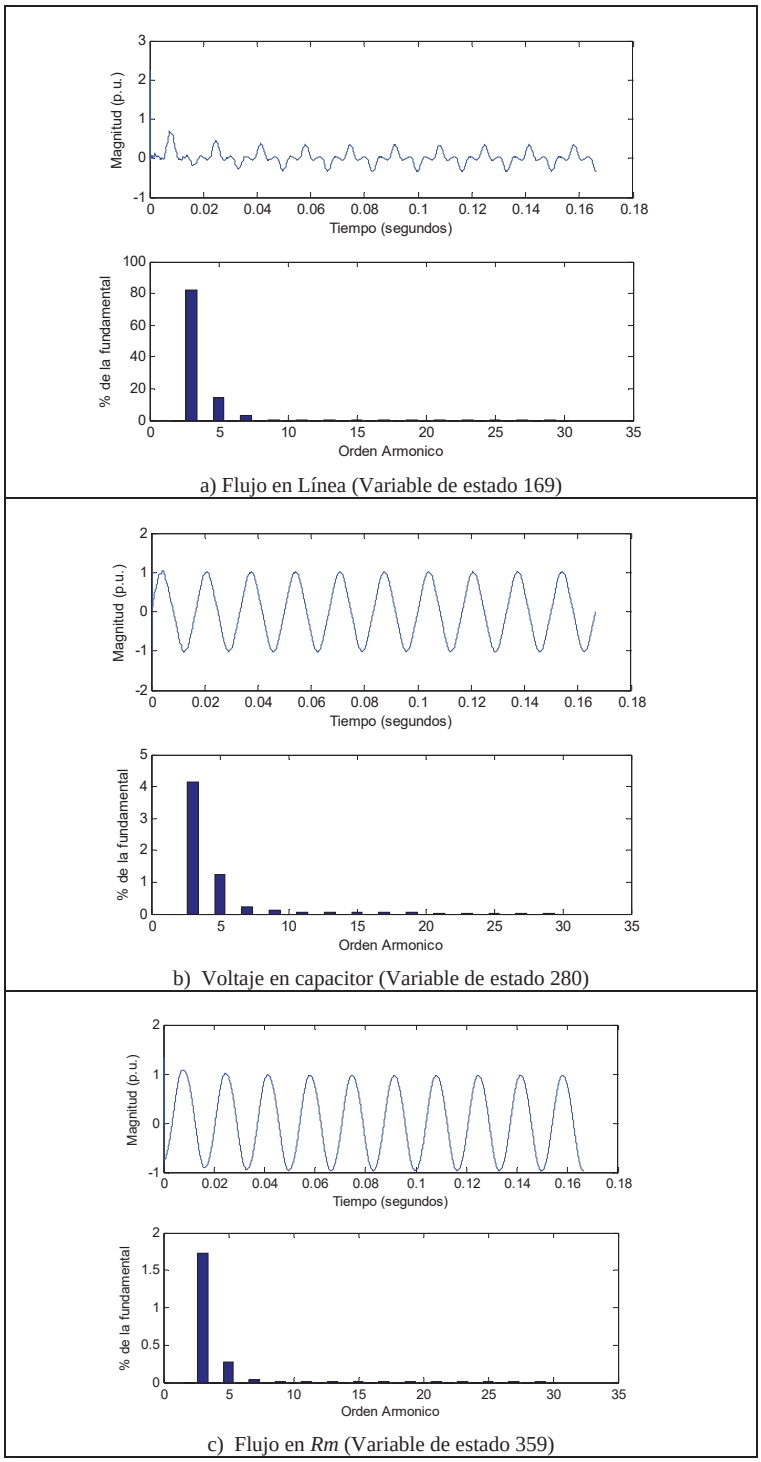
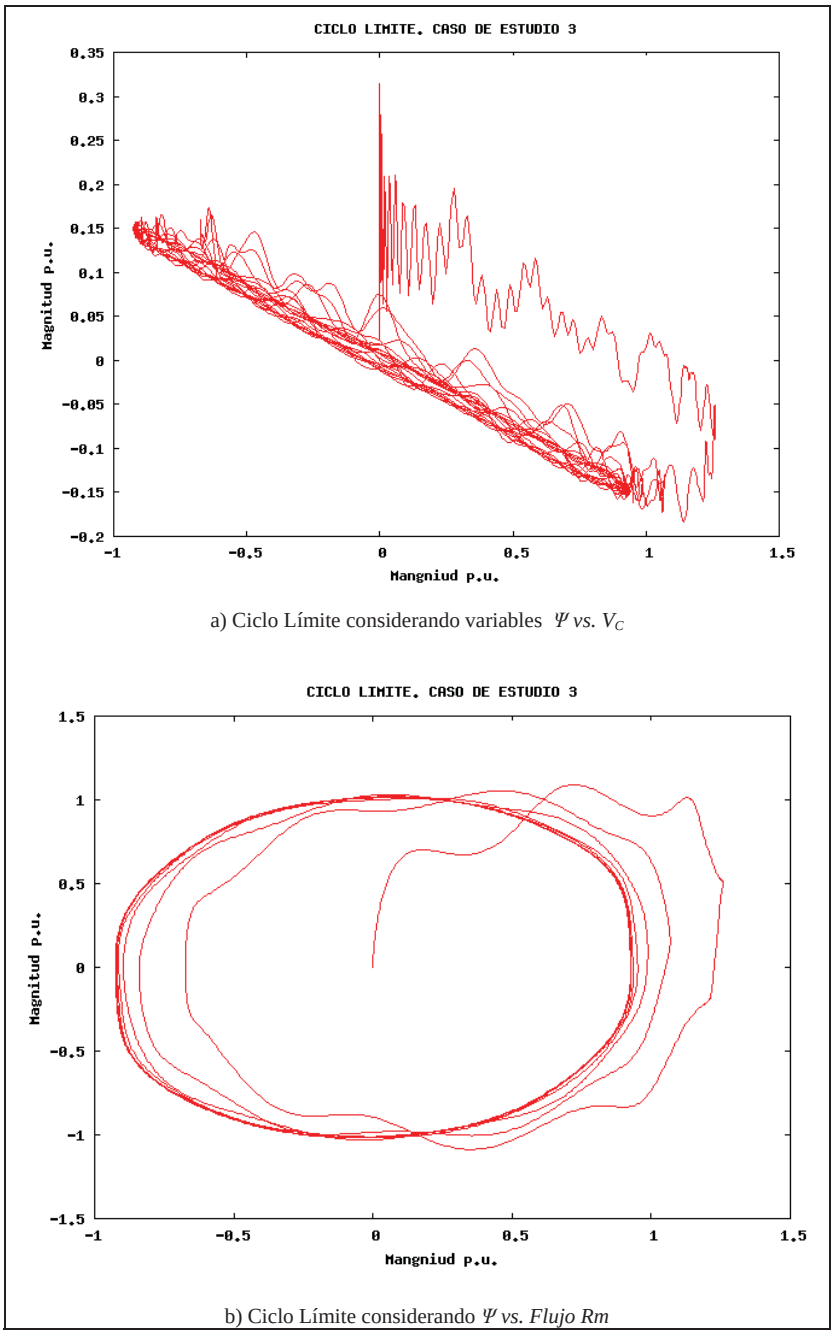


Figura 5.9 Formas de onda en el EEP y su espectro armónico. Caso de Estudio 3.



a) Ciclo Límite considerando variables Ψ *vs.* V_C

b) Ciclo Límite considerando Ψ *vs. Flujo Rm*

Figura 5.10 Ciclo Límite del Caso de Estudio 3

5.3.3. Análisis comparativo de solución secuencial y solución paralela.

El comportamiento del tiempo de ejecución en función del número de EP del cluster se puede observar en la Figura 5.11. Nótese que el tiempo de ejecución disminuye a medida que se incrementa el número de EP. La Eficiencia calculada con MPI es de 84%, mayor a 79 % que se obtiene con PVM. Los valores de Eficiencia obtenidos se presentan en la Tabla 5.10.

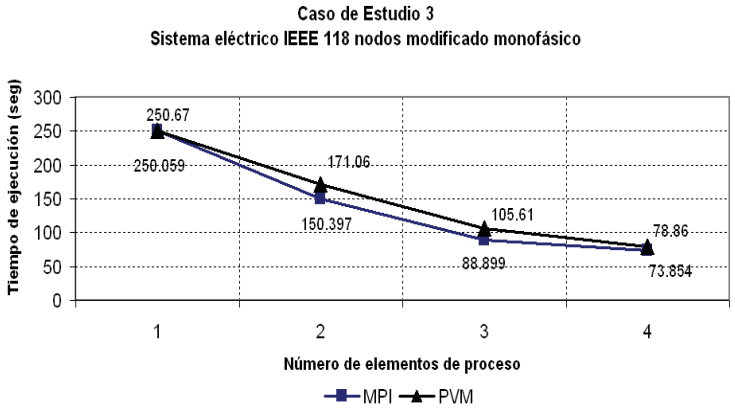


Figura 5.11 Relación Tiempo de ejecución-EP. Caso de Estudio 3

Tabla 5.8 Speed-up y Eficiencia para el Caso de Estudio 3.

Número de nodos	MPI		PVM	
	Speed-up	Eficiencia	Speed-up	Eficiencia
1	1	0.25	1	0.25
2	1.662659	0.415664	1.465392	0.366348
3	2.812843	0.703210	2.373544	0.593386
4	3.385855	0.846463	3.178671	0.794667

En el Caso de Estudio 3, se observa que en sistemas eléctricos que demandan gran esfuerzo computacional, MPI y PVM presentan un comportamiento semejante. No obstante, MPI presenta dos ventajas: la primera es que, según la Tabla 5.8, se tiene una mejora del Speed up ligeramente mayor a PVM. Sin embargo, según se explica en (3.6), a medida que se incrementen los EP, el Speed up llegará a su valor tope determinado por el comportamiento de la parte secuencial del código. La segunda ventaja es que MPI realiza mejor paralelismo, ya que la Eficiencia máxima es de 84% contra 79% de PVM. La tendencia es más favorable a MPI si el número de EP se incrementa, pues al designar más trabajo útil a cada EP, se mantiene el rendimiento, aún y cuando aumente el tamaño de la carga computacional.

Comparando los resultados del Caso de Estudio 3 con los obtenidos en los Casos de Estudio 1 y 2 de esta Tesis, se puede observar que los tiempos de un programa escrito con PVM son ligeramente mayores, lo que indica un menor desempeño en la solución en EEP de sistemas eléctricos de gran escala, y posiblemente se pueda atribuir a una implementación menos eficiente de los mecanismos de comunicación internos de PVM.

Se concluye que, para sistemas eléctricos de gran escala, el código paralelo desarrollado en esta Tesis y ejecutado en MPI distribuye mejor la cantidad de trabajo. Se observa que el rendimiento no depende completamente del número de EP, sino que depende del algoritmo paralelo que se ejecuta, y cuanto mejor paralelizadas estén las tareas del problema, se obtendrán Speed up y Eficiencias mayores.

5.4. COMPARACION DE PLATAFORMAS PARA COMPUTO PARALELO MPI Y PVM

5.4.1. Comparación de las características generales de MPI y PVM

- MPI es un estándar de programación para construir aplicaciones portables en sistemas homogéneos. Las aplicaciones se ejecutan sobre un número fijo de EP. Proporciona una serie de funciones para ejecución, comunicación y sincronización de procesos paralelos [LAM-MPI 2003].
- PVM es un conjunto de herramientas y bibliotecas que emulan aplicaciones de propósito general, en un ambiente de computación concurrente, en arquitecturas de computadoras heterogéneas [PVM 1994].
- MPI fue más sencillo de instalar. PVM requiere un proceso de instalación más complejo y se necesita experiencia en la modificación de los archivos de sistema de Unix [PVM 1994].
- En MPI, el parámetro *-np* especifica exactamente el número de procesos a iniciar. En esta Tesis se especificó un valor máximo de 4, esto es para que se inicien cuatro procesos, uno de los cuales será el maestro los tres restantes serán los esclavos.
- En PVM, la invocación del programa paralelo inicia el primer proceso bajo la máquina virtual. A éste se le especificó con cuántos procesos adicionales se debía iniciar, con el parámetro *-t*. En esta Tesis se definió el valor de 4. Primero se tendrá la instancia inicial del proceso, que a su vez engendrará otras cuatro instancias, dando un total de cinco; de los cuales uno será el proceso padre mientras que los otros cuatro efectúan los cálculos.
- El procedimiento para compilar en MPI usa una sola orden, mientras que en PVM fue necesario generar por comodidad un scrip, donde se incluyeron las órdenes para la compilación (se muestra en 3.3.4.1).
- En PVM se necesita generar siempre dos programas para una misma aplicación: el que ejecuta el nodo maestro y otro programa que ejecutan los nodos esclavos. En esta Tesis, el código escrito para MPI es el mismo para todos los EP (maestro y esclavos) lo cual simplifica la programación.
- Se hizo un ejercicio de prueba con el programa "*hola_mundo*" y se contabilizaron las líneas de código fuente efectivas, encontrándose que MPI necesita 10 líneas contra 18 de PVM (incluye líneas del programa maestro y líneas del programa para los esclavos). Se observó que el código fuente escrito en MPI es más compacto que el código fuente escrito para PVM, aunque siempre dependerá del estilo del programador.
- La invocación de la función de difusión en MPI requiere únicamente una llamada a la función, mientras que en PVM se requieren dos, una para empaquetar los datos y otra para realizar el envío.
- El tiempo total de procesamiento en MPI se obtuvo con la función *MPI::Wtime*() (ver 3.2.2.3.). Se registraron también los tiempos de ejecución de: a) método de FB, b) método para obtener la matriz de identificación Φ y c) matriz inversa. PVM no cuenta con una

función que reporte tiempos de ejecución, por lo que únicamente se pudo registrar el tiempo total de ejecución de todo el programa paralelo de DN usando el comando *time* de Unix[1].

- En esta Tesis no se reportan de tiempos de ejecución entre MPI y PVM de las partes paralelizables del algoritmo propuesto (Figura 4.15) debido a que PVM no incluye en sus librerías una función semejante a *MPI::Wtime*() que pueda aportar tiempos de ejecución de secciones de código.
- Se aprecia que MPI y PVM proporcionan las mismas funciones primitivas básicas de paso de mensajes. En esta Tesis se emplearon las más elementales y se muestran en la Tabla 5.10.
- Tanto MPI como PVM proporcionan aproximadamente la misma funcionalidad. Sin embargo, MPI presenta una implementación mejor diseñada si se va a trabajar en clusters tipo Beowulf, ya que se trata de una arquitectura homogénea.

La Tabla 5.9 muestra las similitudes y diferencias de las plataformas de cómputo paralelo MPI y PVM que se observaron en el desarrollo de esta Tesis.

Tabla 5.9 Similitudes y diferencias de MPI y PVM.

SIMILITUDES	DIFERENCIAS
• Son plataformas paralelas que permiten un ambiente de paso de mensajes donde se generan múltiples procesos independientes (cada uno con su propio espacio de direcciones) que se ejecutan utilizando el modelo de memoria distribuida. • Diseñados para aplicaciones paralelas. • Disponibles como Shareware en Internet. • Tienen funciones para intercambiar información a través de redes y clusters. • Soportan lenguaje Fortran, C y C++. • Proporcionan conversión automática de datos. • Tienen semántica de llamadas a funciones muy parecida. • Los procesos que participan en la aplicación paralela deben ser compilados en cada nodo.	• MPI funciona mejor en ambientes homogéneos, en redes con latencias pequeñas y tasas de velocidad altas. Soporta diferentes topologías de interconexión con número fijo de procesadores, lo que permite mínima sobrecarga en las comunicaciones. • MPI no proporciona mecanismos para especificar el estado de los procesos. • MPI no soporta la creación dinámica de tareas. • MPI soporta las comunicaciones colectivas. • MPI no es un lenguaje, sino que integra funciones y macros que conforman C y C++. • PVM funciona mejor que MPI cuando las tasas de velocidad no son tan óptimas. • PVM esta diseñado para ambientes heterogéneos. • PVM proporciona escalamiento dinámico de la red, por lo que no existe número fijo de procesadores que lo límite. Sin embargo, la sobrecarga en la red es mayor que en MPI. • PVM incluye una consola que se utiliza para monitorear y controlar los estados de la maquina virtual. • PVM soporta la creación dinámica de tareas y grupos dinámicos.

La Tabla 5.10 compara las funciones de paso de mensajes de MPI y PVM que se utilizaron en esta Tesis.

[1] El comando *time* proporciona tres valores de salida: *real*, *user* y *system*. En esta Tesis se tomó el valor correspondiente a *real* [FEDORA 2003].

Tabla 5.10 Comparación de funciones de MPI y PVM.

Tarea	Función en MPI	Función en PVM
Inicialización de la plataforma paralela	MPI::Init()	Implícito
Determinación de número de proceso	MPI::COMM_WORLD.Get_rank()	pvm_mytid() y pvm_joingroup()
Determinación de número de procesos en un grupo o comunicador	MPI::COMM_WORLD.Get_size()	pvm_gsize()
Obtención del nombre de procesador	MPI::COMM_WORLD.Get_processor_name()	no aplica
Inicialización de buffers de envío	no aplica	pvm_initsend()
Empacado de mensajes	no aplica	pvm_pk()
Desempacado de mensajes	no aplica	pvm_upk()
Envío de datos	MPI::Send()	pvm_send()
Recepción de datos	MPI::Recv()	pvm_recv()
Función de difusión de datos	MPI::COMM_WORLD.Bcast()	pvm_bcast() pvm_recv()
Finalización de plataforma paralela	MPI::Finalize()	pvm_exit()

5.4.2. Desempeño obtenido con MPI y PVM en la ejecución del algoritmo paralelo de DN

En la evaluación del desempeño de las plataformas paralelas MPI y PVM se consideró el número n de EP, el tiempo de ejecución $T(n)$ de (3.1), el Speed up y la Eficiencia calculados.

En todos los Casos de Estudio de esta Tesis se obtuvieron incrementos en Speed up y Eficiencia conforme aumentó el número de EP. La diferencia en el desempeño de MPI respecto a PVM se consideró al comparar los resultados mostrados en las Tablas 5.3, 5.6 y 5.8.

En el desempeño de las plataformas paralelas MPI y PVM influye también el tamaño del problema a resolver (cantidad de datos y cálculos de los Casos de Estudio), así como la programación del algoritmo paralelo propuesto (influencia de la parte secuencial) y el ambiente homogéneo del cluster Beowulf. De los resultados obtenidos en los Casos de Estudio de esta Tesis se observó lo siguiente:

1. El desempeño de MPI y PVM no es óptimo cuando se aplica procesamiento paralelo a problemas pequeños -como el Caso de Estudio 1- debido a que, por lo general, en este tipo de problemas se requieren pocos cálculos y el tiempo de ejecución es de corta duración. En la Tabla 5.3 se muestra que tanto MPI como PVM no presentaron mejora significativa en Speed up y Eficiencia; se observa también un desempeño de MPI ligeramente menor debido a que las comunicaciones globales (en especial la función *gather*) que presentan latencias debidas a mecanismos de paralelización, sincronización y transmisión de datos, además de que el nodo maestro no solo asigna las tareas a los nodos esclavos, sino que también ejecuta tareas y cálculos.

   Se concluye que no es recomendable aplicar procesamiento paralelo a problemas pequeños por el coste computacional que implica la creación de los procesos en los nodos del cluster, el comportamiento de las comunicaciones en la red, el arranque de procesos remotos, etc.

2. En problemas no tan pequeños -como el Caso de Estudio 2- MPI presentó ganancia en Speed up y Eficiencia, aproximadamente 30% superior, respecto a PVM[2]. Si bien se aprecia un incremento de Speed up con PVM, en realidad no se tiene ganancia real, ya que con el máximo número de EP disponibles no se logró siquiera duplicar el Speed up que se obtuvo con un solo EP. Además, los valores de Speed up y Eficiencia de PVM empleando cuatro EP son similares a los que obtiene MPI con tan solo dos EP como se muestra en la Tabla 5.6.

   La desventaja que presenta PVM respecto a MPI de se debe al coste que genera la creación de tareas de forma dinámica (balanceo de carga) y el coste del cambio de contexto (ver 4.1.1.3) que se produce cada vez que un EP cambia de una tarea a otra, (se debe cargar registros y otros datos de la nueva tarea que ejecutará). En MPI las tareas son asignadas de forma estática a los EP desde un inicio (ver 4.1.1.4), dando como resultado que el tiempo de ejecución sea ligeramente menor a PVM porque se reduce el costo de creación de procesos, sincronización, terminación, etc.

   Se concluye que el coste de comunicaciones globales de MPI es menos significativo que el coste de cambio de contexto de PVM, cuando se aplica procesamiento paralelo a problemas no tan pequeños.

3. En problemas de gran escala donde existe mayor número de datos y cálculos -como el Caso de Estudio 3-, se observó un comportamiento semejante de MPI y PVM. En ambas plataformas paralelas se tuvo incremento casi lineal de Speed up en relación al número de EP, y además se obtuvieron Eficiencias de valores cercanos: 84% con MPI y 79% con PVM.

   Sin embargo, aunque moderado, el rendimiento de MPI es superior a PVM. La razón es que la parte paralelizable del algoritmo de DN (ver Figura 4.15), en un sistema homogéneo como lo es el cluster Beowulf, mantiene la mayor parte del tiempo a los EP con trabajo efectivo, es decir, que los EP están procesando todos los datos de forma paralela y que todos los EP se utilizan completamente durante ciertos periodos de ejecución. En la Tabla 5.8 se observa que independientemente del número de EP que se agreguen MPI mantiene su rendimiento por encima de PVM.

   Se concluye que mientras mejor paralelizado esté el código de un algoritmo o programa más susceptible será de aumentar su Speed up y Eficiencia. Se deduce también que los costes de tiempo debidos al cambio de contexto en PVM, son mayores que el coste de comunicaciones globales en MPI, y si bien no determinan en su totalidad el rendimiento entre ambas plataformas paralelas, si influye para considerar a MPI mejor opción que PVM.

5.5. CONCLUSIONES

Este Capítulo ha mostrado que en el análisis de los sistemas eléctricos con componentes no lineales y variantes en el tiempo son susceptibles de ser realizados con la aplicación de técnicas de procesamiento en paralelo mediante un sistema distribuido de cómputo basado en una arquitectura Beowulf de 4 nodos.

[2] Eficiencia calculada cuando actúan cuatro EP.

Se implementó la paralelización de la técnica Newton de DN para acelerar la convergencia de las variables de estado al Ciclo Límite para obtener el EEP, de los Casos de Estudio tratados en este Capítulo; los procesos de convergencia siempre son de tipo cuadrático.

Se encontró que paralelizar el método de RK4 de la manera que se propone en esta Tesis no reporta mejora en el Speed up, sino que al contrario, incrementa el tiempo de ejecución del algoritmo paralelo de DN. Por lo tanto, la solución de los Casos de Estudio no incluyó la paralelización de RK4.

Se demostró que la aplicación de procesamiento paralelo al algoritmo de DN logra reducir significativamente el total de ciclos requeridos por la técnica de FB en el proceso de alcanzar el EEP de sistemas eléctricos no lineales de gran escala como el Caso de Estudio 3, de 3356 ciclos a 1109 ciclos, lo cual representa una reducción del 66.95% respecto a la solución de FB.

Se demostró el potencial de la técnica de procesamiento en paralelo MPI aplicada a la obtención del EEP de redes eléctricas con componentes no lineales y variantes en el tiempo. Se ha mostrado que mientras más grande es el sistema de EDO´s que describen la dinámica y comportamiento del sistema eléctrico, se obtiene una mejora en el rendimiento, es decir, se obtienen mayor Speed-up, tanto en MPI como con PVM. Se encontró que la Eficiencia máxima posible para el cluster Beowulf implementado para esta Tesis, se obtiene usando MPI, y es mayor al 80%.

En cuanto al Speed up y Eficiencia logrados, se demuestra que obtiene mayor rendimiento y calidad de paralelismo en el Caso de Estudio 3.

Al desarrollar el algoritmo paralelo de DN que se propone en esta Tesis, se confirmó que existen similitudes a nivel funcional entre PVM y MPI, y que ambas plataformas poseen ventajas y desventajas que se deberán tomar en cuenta al momento de seleccionar cuál de las dos plataformas utilizar. El desempeño del cluster Beowulf utilizado en esta Tesis es aceptable ya que los resultados obtenidos fueron exitosos y se comprueban las ventajas del procesamiento en paralelo, en particular con la plataforma MPI. Específicamente para problemas de gran escala, MPI obtiene un mejor desempeño en relación a PVM.

Por último, la aplicación del procesamiento paralelo MPI en la solución en el EEP de sistemas eléctricos permite apreciar la utilidad del procesamiento en paralelo en posibles trabajos de investigación en la DEP-FIE.

CAPITULO 6

CONCLUSIONES

En este Capítulo se presentan las conclusiones particulares y generales a las que se llegó después de concluir la investigación reportada en esta Tesis. Se propone también un listado de los trabajos futuros que pueden complementar lo alcanzado.

6.1. CONCLUSIONES

- Se demostró la factibilidad de construcción de un cluster tipo Beowulf con los recursos disponibles en la DEP-FIE, siendo la plataforma resultante perfectamente utilizable y práctica.
- Los costos de configuración, operación y mantenimiento del cluster Beowulf propuesto no representaron gastos significativos para la DEP-FIE, debido a que el software utilizado tanto para implementación del cluster propuesto, como la instalación de la plataforma paralela LAM-MPI, es de dominio público.
- En todos los Casos de Estudio se observó que se obtiene un mejor desempeño al agregar más nodos al cálculo, incrementándose con ello el Speed up.
- Se logró implementar el procesamiento distribuido, al realizar los cálculos en múltiples computadoras o nodos, donde cada procesador ejecutó mediante paso de mensajes las funciones de MPI.
- Durante todas las operaciones que se realizan en el algoritmo propuesto, se trabajó el paralelismo de datos, lográndose el balanceo de carga al distribuir de manera lo más equitativamente posible los datos entre todos los nodos del cluster.
- Se observó que el tamaño del problema, así como el número de nodos del cluster utilizados en la solución del problema, son los factores más importantes para determinar su rendimiento máximo, por lo cual es especialmente importante considerar estos dos factores y balancearlos para aprovechar al máximo las capacidades del cluster.
- El cluster Beowulf que se construyó ha sido aplicado con éxito para la determinación de la solución periódica en estado estacionario en el dominio del tiempo de diversas redes eléctricas monofásicas no lineales basadas en el método DN. Con respecto a la solución secuencial, se observó que, la Eficiencia en la solución se incrementa utilizando el cluster y MPI.

- En esta Tesis se ha mostrado la construcción de un cluster tipo Beowulf con máquinas de características homogéneas pertenecientes a la DEP-FIE, el desarrollo de un algoritmo paralelo del método de DN y la aplicación de procesamiento en paralelo basado en la plataforma MPI para la solución en EEP de sistemas eléctricos.
- Se aplicó programación en paralelo al algoritmo del método Newton de DN (cálculo de la matriz de identificación Φ y cálculo de la matriz inversa por factorización *LU*), mismo que fue aplicado con éxito a los siguientes sistemas eléctricos no lineales: sistema de 3 nodos monofásico, sistema modificado IEEE de 14 nodos y sistema modificado IEEE de 118 nodos.
- Se observó que el algoritmo paralelo del método de la FB (Runge_Kutta de cuarto orden) no proporcionó el incremento esperado en el Speed up, y en cambio, aumentaron los tiempos de cálculo, por lo que se concluye que, para el algoritmo paralelizado de DN que se propone en esta Tesis, no se recomienda paralelizar el método de Runge_Kutta de cuarto orden.
- Se obtuvieron resultados asociados a las formas de onda de variables que representan los voltajes y corrientes de los Casos de Estudio.
- Se observó que la propuesta de aplicación de procesamiento en paralelo al algoritmo DN permitió aumentar la Eficiencia, en promedio, a valores mayores al 40%, y el Speed up hasta más de 70%, lo que indica que se logra una calidad de paralelismo aceptable. Se puede afirmar en base los resultados obtenidos, que conforme el tamaño de la red aumenta, se obtiene una mejora significativa en la Eficiencia computacional, tanto con procesamiento paralelo en MPI, como con PVM.
- Se observó que para problemas de tamaño pequeño, no se logra mejora en Speed up y Eficiencia; por lo tanto, no se recomienda usar procesamiento en paralelo con la propuesta desarrollada en esta Tesis.
- Se observó que conforme el tamaño del problema crece, se alcanzan mejores incrementos en Speed up y Eficiencia con MPI en relación a lo obtenido con PVM.
- En los resultados obtenidos se mostró el potencial que tiene el uso del procesamiento en paralelo MPI en la eficiente solución en estado estacionario de sistemas eléctricos de potencia.
- Se comprobó que MPI y PVM proporcionan aproximadamente la misma funcionalidad y comportamiento cuando se aplican en casos de estudio de gran escala; aunque se observó que MPI presenta ventaja en relación a PVM.
- Finalmente, se considera que los objetivos planteados al inicio de esta Tesis se alcanzaron satisfactoriamente.

6.2. RECOMENDACIONES PARA TRABAJOS FUTUROS

- Implementar un cluster como el propuesto en esta Tesis, pero con mayor número de nodos, inclusive si cada nodo pudiera tener uno o más procesadores. De esta manera se podrán explorar las librerías para threads que tiene MPI.
- Implementar un cluster como el propuesto en esta Tesis, pero empleando interfase Ethernet de alta velocidad, sobre todo si se continúa utilizando la topología estrella.
- Instalar en el cluster alguna herramienta para monitoreo del mismo. Se propone el programa conocido como *bWatch*, el cual muestra en una ventana información a cerca de cada uno de los nodos que integran el cluster (nombre del nodo, número de usuarios conectados, procesos que se ejecutan, carga en cada nodo, manejo de memoria etc.).
- Generar un entorno gráfico para el cluster, de manera que se pueda realizar la compilación y ejecución de programas desarrollados en plataformas MPI y PVM de manera más amigable al usuario.
- Un cluster, como el que se empleó en esta Tesis puede tener futuro como instrumento para permitir el estudio práctico de técnicas y herramientas de programación paralela, así como la experimentación con distintos algoritmos. Esto permitirá generar experiencia y conocimiento en esta área de la computación.

APENDICE A

MODELADO DE LOS ELEMENTOS DE LOS SISTEMAS ELÉCTRICOS DE ESTUDIO.

LINEA DE TRANSMISION

Cada una de las líneas de transmisión está representada utilizado una resistencia *R* conectada en serie con un inductor *L*, tal como se ilustra en la Figura A.1.

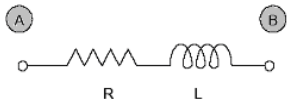


Figura A.1 Modelo de la línea de transmisión.

La ecuación que modela la línea de transmisión se determina aplicando LVK en el circuito de la Figura A.1, obteniéndose:

$$\frac{di}{dt} = \frac{V_A - V_B}{L} = \frac{iR}{L} \qquad \text{(A.1)}$$

GENERADOR

El generador se modela por medio de una fuente sinusoidal de voltaje constante, conectada a un inductor. El circuito se observa en la Figura A.2.

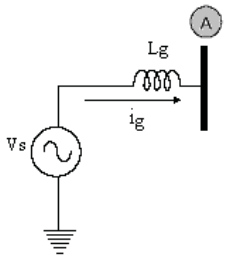


Figura A.2 Modelo del Generador.

Aplicando LVK al circuito de la figura A.2, y despejando *g di* se obtiene la ecuación:

$$\frac{di_g}{dt} = \frac{V_g - V_A}{L_g} \qquad \text{(A.2)}$$

El voltaje en terminales del generador está determinado por:

$$V_g = Msen(\omega t + \phi) \qquad \text{(A.3)}$$

donde: *M* es el valor pico del voltaje, φ es el ángulo de fase del voltaje expresado en radianes, y ω es la velocidad angular.

BANCO DE CAPACITORES

En el modelado de un banco de capacitares monofásico se consideran todas las corrientes que inciden en el nodo, así como todas las corrientes que salen del nodo, a excepción de la corriente en el capacitor, la cual está representada por i_C y se considera siempre que sale del nodo. La Figura A.3 muestra un banco de capacitores monofásico conectado a un nodo del sistema.

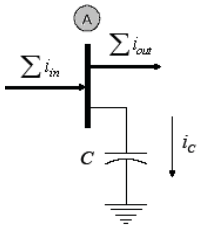


Figura A.3 Modelo del banco de capacitares.

donde: $\sum i_{in}$ representa la suma de ,corrientes que entran al nodo *A*, y $\sum i_{out}$ representa la suma de las corrientes que salen del nodo *A*.

Las corrientes que entran y salen de un nodo se obtienen aplicando LCK sobre el nodo, y se tiene que:

$$\sum i_{in} = \sum i_{out} + i_C \qquad \text{(A.4)}$$

por tanto, el voltaje en el nodo se describe con la ecuación:

$$\frac{dVc}{dt} = \frac{\sum i_{in} - \sum i_{out}}{C} \qquad \text{(A.5)}$$

RAMA MAGNETIZANTE

La Figura A.4 muestra la conexión en el nodo *A* de una rama magnetizante.

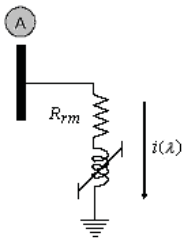


Figura A.4 Modelo de la rama magnetizante.

La aplicación de una LVK alrededor de la malla da como resultado la ecuación siguiente:

$$V_A = V_R + V_L \tag{A.6}$$

donde $i(\lambda)$ es el flujo en la rama, y V_R y V_L están dados por:

$$V_R = R_{rm} i(\lambda) \tag{A.7}$$

$$V_L = \frac{d\lambda}{dt} \tag{A.8}$$

Sustituyendo (A.7) y (A.8) en (A.6), y despejando $\frac{d\lambda}{dt}$, se llega a la ecuación que describe el circuito de la figura A.4:

$$\frac{d\lambda}{dt} = V_A - R_{rm} i(\lambda) \tag{A.9}$$

El efecto de saturación en la rama magnetizante es representado en esta Tesis mediante una aproximación basada en un polinomio de grado *n*. El comportamiento no lineal de la rama magnetizante está expresado por medio de la relación no lineal:

$$i(\lambda) = \lambda^n \tag{A.10}$$

donde: *n* es un número impar, debido a que (3.6) tiene simetría impar. En esta Tesis, se asigna a *n* un valor de 5 [Medina *et al*. 2004].

APENDICE B

FACTORIZACION LU DE DOOLITTLE Y CROUT PARA EL CALCULO DE MATRIZ INVERSA

FACTORIZACION LU DE DOOLITTLE

La eliminación de Gauss equivalente a factorizar la matriz de coeficientes $A = LU$ como el producto de una matriz triangular inferior unitaria L y una triangular superior U se denomina *factorización de Doolittle* [Chapra 2003]. El procedimiento se basa en la eliminación de Gauss descrita en forma matricial

$$A = LU \tag{B.1}$$

La eliminación de los ceros en la primera columna de $A^{(1)} = A$, se puede obtener mediante el producto matricial:

$$A^{(2)} = L_1 A^{(1)} \tag{B.2}$$

donde la matriz triangular inferior L_1 toma la forma:

$$L_1 = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ -m_{21} & 1 & 0 & \cdots & 0 \\ -m_{31} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ -m_{n1} & 0 & 0 & \cdots & 1 \end{pmatrix}, \qquad m_{i1} = \frac{a_{i1}^{(1)}}{a_{11}^{(1)}} \tag{B.3}$$

Llamando $m_1 \in \Re^n$ al vector cuyas componentes son $(0, m_{21}, m_{31}, ..., m_{n1})$, y llamando e_1 al primer vector de la base canónica de $\Re^n$, L_1 se puede escribir como:

$$L_1 = I - m_1 e_1^{\mathrm{T}} \tag{B.4}$$

para el siguiente paso, se tiene que $A^{(3)} = L_2 A^{(2)}$, donde:

$$L_2 = I - m_2 e_2^{\mathrm{T}} = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & -m_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ 0 & -m_{n2} & 0 & \cdots & 1 \end{pmatrix}, \qquad m_2 = \begin{pmatrix} 0 \\ 0 \\ m_{32} \\ \vdots \\ m_{n2} \end{pmatrix}, \quad m_{i1} = \frac{a_{i2}^{(2)}}{a_{22}^{(2)}} \tag{B.5}$$

y e_2 es el segundo vector de la base canónica. De esta manera, la etapa *k*-ésima corresponde a:

$$A^{(k)} = L_k A^{(k-1)},\ L_k = I - m_k e_k^{\mathrm{T}},\ m_k = (0,\ldots,0,m_{(k+1,k)},\ldots,m_{nk})^{\mathrm{T}} \tag{B.6a}$$

y la matriz triangular superior *U* obtenida tras el último paso resulta como:

$$U = L_{n-1}\cdots L_2 L_1 A = (I - m_{n-1}e_{n-1}^{\mathrm{T}})\cdots(I - m_2 e_2^{\mathrm{T}})(I - m_1 e_1^{\mathrm{T}})A \tag{B.6b}$$

La matriz *A* se puede reescribir como el producto de matrices de la forma:

$$A = L_1^{-1}L_2^{-1}\cdots L_{n-1}^{-1}U \tag{B.6c}$$

La inversa de la matriz triangular L_k se obtiene invirtiendo el signo de los multiplicadores:

$L_k^{-1} = I + m_k e_k$, ya que $e_k m_k$=0,

y por tanto,

$$(I - m_k e_k^{\mathrm{T}})(I + m_k e_k^{\mathrm{T}}) = I - m_k e_k^{\mathrm{T}} m_k e_k^{\mathrm{T}} = I \tag{B.6d}$$

Si $e_i m_k = 0$ para $k \geq i$, el producto de las inversas de las matrices triangulares es una matriz triangular:

$$L = (I + m_1 e_1^{\mathrm{T}})(I + m_2 e_2^{\mathrm{T}})\cdots(I - m_{n-1}e_{n-1}^{\mathrm{T}}) = I + \sum_{i=1}^{n-1} m_i e_1^{\mathrm{T}} \tag{B.6e}$$

se puede escribir como:

$$L_1 = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ m_{21} & 1 & \cdots & 0 & 0 \\ m_{31} & m_{32} & \ddots & \ddots & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ m_{n1} & m_{n2} & \cdots & m_{n,n-1} & 1 \end{pmatrix},\ m_{i1} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}} \tag{B.7}$$

FACTORIZACION LU DE CROUT

La factorización *LU* de Crout es equivale a una eliminación de Gauss pero supone que se puede factorizar la matriz *A* en una matriz triangular inferior *L* y una matriz triangular superior unitaria *U*, tal que $A = LU$. Crout propone que se puede factorizar la matriz *A* como:

$$A = \tilde{L}\tilde{D}\tilde{U} \tag{B.8}$$

donde *D* es una matriz diagonal, y $\tilde{L}$ y $\tilde{U}$ son matrices triangulares inferiores y superiores, respectivamente, unitarias.

La resolución del sistema de ecuaciones $Ax = b$ por eliminación de Gauss, matricialmente corresponde a resolver el sistema triangular:

$$Ux = L^{-1}b\,,\;\; U = L^{-1}A \tag{B.9}$$

Sin embargo, un método mas eficiente cuando se conoce la factorización *LU* de Crout es resolver los dos sistemas triangulares con matrices de coeficientes *L* y *U*, en ese orden. Es decir,

$$LUx = b \Rightarrow Ly = b \;\; \text{y} \;\; Ux = y \tag{B.10}$$

FACTORIZACION LU DE DOOLITLE Y CROUT

La combinación de la factorización *LU* de Doolittle y de Crout permite un menor coste computacional que la eliminación de Gauss. Considérese, primero, la factorización *LU* de Doolittle, en la que *L* tiene diagonal unitaria, $l_{ii} = 1$, y se observa como:

$$A = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{pmatrix}$$

Para determinar directamente la expresión de los coeficientes de *L* y de *U* en función de los de *A* basta multiplicar dichas matrices y comparar sus coeficientes con los de *A*. Multiplicando la primera fila de *L* por todas y cada una de las columnas de *U* se obtiene:

$$u_{1j} = a_{1j}\,, \qquad \text{para } 1 \le j \le n \tag{B.11a}$$

multiplicando todas las filas de *L* por la primera columna de *U*, de tiene:

$$l_{i1}u_{1j} = a_{i1}, \Rightarrow l_{i1} = \frac{a_{i1}}{u_{11}} = \frac{a_{i1}}{a_{11}}, \text{ para } 1 \le i \le n \qquad \text{(B.11b)}$$

multiplicando la segunda fila de L por las columnas de U, y utilizando los coeficientes ya calculados, se obtiene:

$$l_{j1}u_{1j} + u_{2j} = a_{2j} \Rightarrow u_{2j} = a_{2j} - l_{j1}u_{1j} = a_{2j} - \frac{a_{j1}}{a_{11}}a_{1j}, \text{ para } 2 \le j \le n \qquad \text{(B11.c)}$$

multiplicando las filas de L por la segunda columna de U, y utilizando los coeficientes ya calculados, se obtiene:

$$l_{i1}u_{1i} + l_{i2}u_{2i} = a_{i2}, \Rightarrow l_{i1} = \frac{1}{u_{2i}}(a_{i2} - l_{i1}u_{1i}), \text{ para } 3 \le j \le n \qquad \text{(B.11d)}$$

Los resultados que se van obteniendo son muy similares a los obtenidos por el procedimiento de eliminación de Gauss. Operando sucesivamente de la misma forma se llega a la expresión general para calcular los coeficientes de L y U:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}, \text{ para } 1 \le i \le j \le n \qquad \text{(B.12)}$$

$$l_{ij} = \frac{1}{u_{ii}}\left(a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}\right), \qquad \text{para } 1 \le j \le i \le n-1 \qquad \text{(B.13)}$$

que se aplicará alternativamente, primero para la fila i de U y luego para la columna j de L. Considérese ahora, la factorización LU de Crout en la que U tiene diagonal unitaria u_{ii}=1,

$$A = \begin{pmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ l_{31} & l_{32} & l_{23} & \cdots & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nn} \end{pmatrix}\begin{pmatrix} 1 & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & 1 & u_{23} & \cdots & u_{2n} \\ 0 & 0 & 1 & \cdots & u_{3n} \\ \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

se determina la expresión de los coeficientes de L y de U. Para la primera columna de L,

$$l_{i1} = a_{i1} \qquad \text{para } 1 \le i \le n \qquad \text{(B.14a)}$$

y para la primera fila de U,

$$l_{11}u_{1j} = a_{1j}, \Rightarrow u_{1j} = \frac{a_{1j}}{l_{11}} = \frac{a_{1j}}{a_{11}} \quad \text{para } 2 \le j \le n \qquad \text{(B.14b)}$$

Para las demás columnas de L y filas de U el procedimiento general que se obtiene es:

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \ , \qquad \text{para } 1 \le i \le j \le n \qquad \text{(B.15)}$$

$$u_{ij} = \frac{1}{l_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right), \qquad \text{para } 2 \le j < i \le n \qquad \text{(B.16)}$$

Generalmente las matrices L y U no se almacenan con sus elementos nulos, sino que se almacenan las dos matrices en una única matriz LU con objeto de reducir el coste en memoria. Además, se observa que en la iteración k-ésima del algoritmo no se requiere el conocimiento de los valores de a_{ij} con $1 < i, j < k$, por lo que los valores ya calculados de las matrices L y U se pueden sobreescribir sobre los valores originales de la matriz A, siempre y cuando no importe que los valores originales de la matriz A se pierdan. De esta forma se reduce el coste en memoria.

CALCULO DE LA MATRIZ INVERSA

Si se tiene la factorización LU de Doolittle de la matriz de coeficientes A, la solución para calcular la matriz inversa de A implica resolver varios sistemas con la misma matriz de coeficientes. Considérese que el sistema se reduce a:

$$Ax=b \qquad \text{(B.17)}$$

que es equivalente a dos sistemas triangulares, el primero triangular inferior y el segundo triangular superior (B.10)

$$Ly=b \ \text{ y } \ Ux=y$$

La inversa de una matriz no singular $A \in M_{nn}$ es una matriz no singular de la forma:

$$X = \begin{matrix} x_{11} & \cdots & x_{1n} \\ \vdots & \cdots & \vdots \\ x_{n1} & \cdots & x_{nn} \end{matrix}$$

tal que $AX=I$; esta igualdad se reduce a resolver n sistemas de la forma:

$$A = \begin{pmatrix} X_{j1} \\ \vdots \\ X_{jn} \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} \leftarrow_j \quad \text{donde } j = 1,2,3,\ldots,n$$

donde *X* se puede considerar como una matriz particionada en columnas, lo que denota *n* vectores de *n* elementos [Davis 2006], tal que :

$$X_{jn} = (x_{j1}, x_{j2}, \ldots x_{jn}) \text{ donde } j = 1,2,3,\ldots,n$$

por tanto, *AX=I* es equivalente a:

$$A\begin{pmatrix} X_{j1} \\ X_{j2} \\ X_{j3} \\ \vdots \\ X_{jn} \end{pmatrix} = \begin{pmatrix} AX_{j1} \\ AX_{j2} \\ AX_{j3} \\ \vdots \\ AX_{jn} \end{pmatrix} = \begin{pmatrix} e_1 \\ e_2 \\ e_3 \\ \vdots \\ e_n \end{pmatrix} \qquad \text{para } j = 1,2,\cdots,n$$

$$\text{donde } e_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \qquad e_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \qquad e_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix}, \qquad \cdots e_n = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}$$

son los vectores columna de la matriz identidad. Bastará, por tanto, resolver por separado cada una de las ecuaciones matrices $AX = e$ utilizando la factorización *LU* de *A*, resultando:

$$LUX = e \tag{B.18}$$

Si *LyX=b* y *UxX=y*, resultan *n* sistemas por resolver:

$$Ly_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \qquad Ly_2 = \begin{pmatrix} 0 \\ 2 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \qquad Ly_n = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}$$

Aplicando la sustitución hacia atrás del método de Gauss se obtienen los *n* vectores solución *y* correspondientes a las columnas de la matriz *U*. Resolviendo $Ux = y$, y se obtienen las columnas de la matriz inversa de *A*. Esto es:

$$Ux_1 = y_1,\ Ux_2 = y_2,\ \ldots,\ Ux_n = y_n$$

donde $x_1, x_2, \ldots x_n$ son vectores solución que conforman las columnas de A^{-1}.

Posteriormente se resuelve la matriz triangular superior $Ux=y$ por sustitución hacia delante o gaussiana. A medida que se construye la matriz inversa, se pueden utilizar los espacios de los ceros en la parte inferior de la matriz *U* para almacenar los elementos de *L*. La diagonal de *L* no se calcula ya que se conoce que de antemano que es unitaria.

Orden de Complejidad

La factorización $A=LU$ es equivalente al procedimiento de eliminación de Gauss para solución de sistemas lineales debido a que ambos métodos tienen el mismo coste computacional $O(n^3)$ [Chapra 2003]. Sin embargo, la ventaja fundamental de la factorización de Doolittle y Crout resulta cuando hay que resolver varios sistemas lineales con una misma matriz de coeficientes *A* y diferentes términos no homogéneos, *b*, en cuyo caso basta almacenar las matrices *L* y *U* una sola vez, entonces el cálculo de las sucesivas soluciones realizado con el coste de la solución de dos sistemas triangulares, que es del orden $O\left(\frac{2n^3}{3}\right)$ [Chapra 2003], resulta un menor coste computacional.

Calcular la inversa de una matriz resolviendo los sistemas $Ax=e_i$ para $i=1,..n$ precisa del orden de $\frac{n^3}{3}$ operaciones para factorizar la matriz *A*, y $n \times n^2$ operaciones para resolver los *n* sistemas triangulares [Ascher *et al.* 1991], lo que hace un total de $\frac{4n^3}{3}$, por tanto, calcular la matriz inversa supone un costo de 4 veces lo que supone resolver un sistema lineal y no *n* veces como se puede pensar en un principio.

APENDICE C

ECUACIONES DE LOS CASOS DE ESTUDIO

CASO DE ESTUDIO 1

Las ecuaciones de Estado que describen el sistema eléctrico del Caso de Estudio 1 son:

Número de Variable de Estado	Ecuación de Estado
1	$\dot{\psi}_1 = -\psi_1 \frac{r_1}{l_1} - V_{C1} + Vs$
2	$\dot{\psi}_2 = -\psi_2 \frac{r_2}{l_2} - V_{C2}$
3	$\dot{\psi}_3 = -\psi_3 \frac{r_3}{l_3} + V_{C1} - V_{C2}$
4	$\dot{\psi}_4 = -\psi_4^n r_4 + V_{C1}$
5	$\dot{\psi}_5 = -\psi_5^n r_5 + V_{C2}$
6	$\dot{V}_{C1} = -\left(\frac{\psi_1}{l_1} - \frac{\psi_3}{l_3} - \psi_4^n\right)\frac{1}{C_1}$
7	$\dot{V}_{C2} = -\left(\frac{\psi_2}{l_2} + \frac{\psi_3}{l_3} - \psi_5^n\right)\frac{1}{C_2}$

CASO DE ESTUDIO 2

Las ecuaciones de Estado que describen el sistema eléctrico del Caso de Estudio 2 son:

Número de Variable de Estado	Ecuación de Estado
0	$\dot{\psi}_1 = (V_1 - V_2 - \psi_1 r_1)/l_1$
1	$\dot{\psi}_2 = (V_1 - \psi_{22} - \psi_2 r_2)/l_2$
2	$\dot{\psi}_3 = (V_2 - V_3 - \psi_3 r_3)/l_3$
3	$\dot{\psi}_4 = (V_2 - \psi_{21} - \psi_4 r_4)/l_4$
4	$\dot{\psi}_5 = (V_2 - \psi_{22} - \psi_5 r_5)/l_5$
5	$\dot{\psi}_6 = (V_3 - \psi_{21} - \psi_6 r_6)/l_6$
6	$\dot{\psi}_7 = (\psi_{21} - \psi_{22} - \psi_7 r_7)/l_7$
7	$\dot{\psi}_8 = (\psi_{21} - \psi_{23} - \psi_8 r_8)/l_8$
8	$\dot{\psi}_9 = (\psi_{21} - \psi_{24} - \psi_9 r_9)/l_9$
9	$\dot{\psi}_{10} = (\psi_{22} - V_6 - \psi_{10} r_{10})/l_{10}$
10	$\dot{\psi}_{11} = (V_6 - \psi_{27} - \psi_{11} r_{11})/l_{11}$
11	$\dot{\psi}_{12} = (V_6 - \psi_{26} - \psi_{12} r_{12})/l_{12}$
12	$\dot{\psi}_{13} = (V_6 - \psi_{28} - \psi_{13} r_{13})/l_{13}$
13	$\dot{\psi}_{14} = (\psi_{23} - V_8 - \psi_{14} r_{14})/l_{14}$
14	$\dot{\psi}_{15} = (\psi_{23} - \psi_{24} - \psi_{15} r_{15})/l_{15}$
15	$\dot{\psi}_{16} = (\psi_{24} - \psi_{25} - \psi_{16} r_{16})/l_{16}$
16	$\dot{\psi}_{17} = (\psi_{24} - \psi_{29} - \psi_{17} r_{17})/l_{17}$
17	$\dot{\psi}_{18} = (\psi_{25} - \psi_{26} - \psi_{18} r_{18})/l_{18}$
18	$\dot{\psi}_{19} = (\psi_{27} - \psi_{28} - \psi_{19} r_{19})/l_{19}$
19	$\dot{\psi}_{20} = (\psi_{28} - \psi_{29} - \psi_{20} r_{20})/l_{20}$
20	$\dot{V}_{C4} = (\psi_4 + \psi_6 - \psi_7 - \psi_8 - \psi_9)/C_4$
21	$\dot{V}_{C5} = (\psi_2 + \psi_5 + \psi_7 - \psi_{10})/C_5$

22	$\dot{V}_{C7} = \left(\psi_8 - \psi_{14} - \psi_{15}\right) / C_7$
23	$\dot{V}_{C9} = \left(\frac{1}{l_9} + \frac{1}{l_{15}} - \frac{1}{l_{16}} - \frac{1}{l_{17}} - \psi_{rm9}^n\right)\frac{1}{C_9}$
24	$\dot{V}_{C10} = \left(\psi_{16} - \psi_{18}\right) / C_{10}$
25	$\dot{V}_{C11} = \left(\psi_{12} - \psi_{18}\right) / C_{11}$
26	$\dot{V}_{C12} = \left(\psi_{11} - \psi_{19}\right) / C_{12}$
27	$\dot{V}_{C13} = \left(\frac{1}{l_{13}} + \frac{1}{l_{19}} - \frac{1}{l_{20}} - \psi_{rm13}^n\right)\frac{1}{C_{13}}$
28	$\dot{V}_{C14} = \left(\frac{1}{l_{17}} + \frac{1}{l_{20}} - \psi_{rm14}^n\right)\frac{1}{C_{14}}$
29	$\psi_{rm9} = -\psi_{rm9}^n r_{m9} + \frac{\psi_{24}}{C_9}$
30	$\psi_{rm13} = -\psi_{rm13}^n r_{m13} + \frac{\psi_{28}}{C_{13}}$
31	$\psi_{rm14} = -\psi_{rm14}^n r_{m14} + \frac{\psi_{29}}{C_{14}}$

La dirección del flujo se considera del nodo del número menor al nodo del número mayor. Las resistencias r representan la resistencia de la línea entre los dos nodos donde se estableció el flujo.

CASO DE ESTUDIO 3

Las 354 ecuaciones de estado que describen el sistema eléctrico del Caso de Estudio 3 fueron generadas utilizando la herramienta computacional de [Ramos 2004] y están disponibles en el formato electrónico de esta Tesis.

APENDICE D

CLASIFICACION DE SUPERCOMPUTADORAS PARA PROCESAMIENTO PARALELO.

Al inicio de la década de los 90 fue necesario acordar la definición de "*supercomputadora*" para producir estadísticas confiables. Tras experimentar con diversas métricas basadas en el número de procesadores, en 1993 la Universidad de Mannheim en Alemania propuso utilizar un listado de los sistemas de computo en producción como base comparativa. Un año después, se incluye al proyecto el software "*benchmark Linpack*" que serviría como instrumento de prueba para la clasificación. La primera lista de prueba y los resultados se publicaron en Internet bajo el nombre de *Top500*. Todas las listas publicadas desde el inicio del proyecto están publicadas en la página web http://top500.org.

En la lista Top500 se incluyen supercomputadoras de memoria compartida y memoria distribuida. No se incluyen sistemas basados en computación GRID ni a la supercomputadora MDGRAPE ya que esta última no es supercomputadora de propósito general ni puede ejecutar el software de prueba. [TOP500 2008].

Para medir la potencia de los sistemas, actualmente se utiliza el *benchmark HPL*, una versión portable del benchmark Linpack para computadoras de memoria distribuida.

Hasta la fecha (Noviembre de 2008) la supercomputadora Cray XT5 Jaguar es la más rápida del mundo [TOP500 2008]; es operada por el Departamento de Energía del Laboratorio Nacional de Oak Ridge en los Estados Unidos y es la segunda en romper la barrera de los petaflop/s, tardando únicamente 1.059 petaflop/s en ejecutar la aplicación de prueba benchmark Linpack; contiene procesadores Opteron Quad Core (4 núcleos) de AMD, apoyados por 362 TB de memoria RAM y puede manejar más de 284 GB de información por segundo y almacenarla en sus discos duros, que en conjunto tienen 750 TB de capacidad. En la Tabla D1 se muestran las diez computadoras de mayor poder de cálculo según [TOP500 2008].

De un total de 379 sistemas, (75.8%) usan procesadores Intel, lo que constituye a dicha compañía como la principal proveedora de procesadores en la lista. En segundo lugar están los procesadores IBM Power (12%) y la familia Opteron AMD (11.8%). 336 sistemas utilizan procesadores de cuatro núcleos, 153 de dos núcleos y solamente cuatro sistemas tienen procesadores de un solo núcleo. En cuanto a las marcas, HP mantiene el 41.8% de los sistemas, contra el 36,6% de IBM [TOP500 2008].

Tabla D1 Distribución de arquitecturas de las 500 principales supercomputadoras.

LUGAR	LUGAR DE OPERACION	COMPUTADORA / FABRICANTE	NUCLEOS	GFLOPS	AÑO	Consumo de energía KW.
1	**DOE/NNSA/LANL** Estados Unidos	Roadrunner - BladeCenter QS22 / LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz , Voltaire Infiniband **IBM**	129600	1105	2008	2483.47
2	**Oak Ridge National Laboratory** Estados Unidos	Jaguar - Cray XT5 QC 2.3 GHz **Cray Inc.**	150152	1059	2008	6950.60
3	**NASA/Ames Research Center** Estados Unidos	Pleiades - SGI Altix ICE 8200EX, Xeon QC 3.0/2.66 GHz **SGI**	51200	487.01	2008	2090.00
4	**DOE/NNSA/LLNL** Estados Unidos	BlueGene/L - eServer Blue Gene Solution **IBM**	212992	478.20	2007	2329.60
5	**Argonne National Laboratory** Estados Unidos	Blue Gene/P Solution **IBM**	163840	450.30	2007	1260.00
6	**Texas Advanced Computing Center/Univ. of Texas** Estados Unidos	Ranger - SunBlade x6420, Opteron QC 2.3 Ghz, Infiniband **Sun Microsystems**	62976	433.20	2008	2000.00
7	**NERSC/LBNL** Estados Unidos	Franklin - Cray XT4 QuadCore 2.3 GHz **Cray Inc**.	38642	266.30	2008	1150.00
8	**Oak Ridge National Laboratory** Estados Unidos	Jaguar - Cray XT4 QuadCore 2.1 GHz **Cray Inc**.	30976	205.00	2008	1580.71
9	**NNSA/Sandia National Laboratories** Estados Unidos	Red Storm - Sandia/ Cray Red Storm, XT3/4, 2.4/2.2 GHz dual/quad core **Cray Inc.**	38208	204.20	2008	2506.00
10	**Shanghai Supercomputer Center** China	Dawning 5000A - Dawning 5000A, QC Opteron 1.9 Ghz, Infiniband, Windows HPC 2008 **Dawning**	30720	180.06	2008	---

DOE: Department of Energy.
NNSA/LANL: Los Alamos National Laboratory
LLNL: Lawrence Livermore National Laboratory

EQUIPOS DE SUPERCOMPUTO EN MEXICO

En México existen cuatro centros de supercómputo en el ámbito académico reconocidos como tales: UNAM, UAM-I, IPN y CNSCC- IPICyT [CNSCC- IPICyT 2004]; en la Figura D1 se observa un comaparativo del poder combinado de procesamiento de cada institución hasta 2004. En Latinoamérica el total de centros de supercómputo en el área académica tiene un poder combinado de cómputo que supera el par de TFlops. El poder combinado se trata de un caso ideal donde se pudieran conjuntar todos los equipos de supercómputo de cada institución y pudiesen

trabajar como un solo recurso de cómputo. En este contexto, el CNSCC-IPICyT cuenta con un poder de cómputo 950 GFlops en un solo equipo, mientras que el poder combinado de computo de UAM-I y IPN es menor [CNSCC- IPICyT 2004].

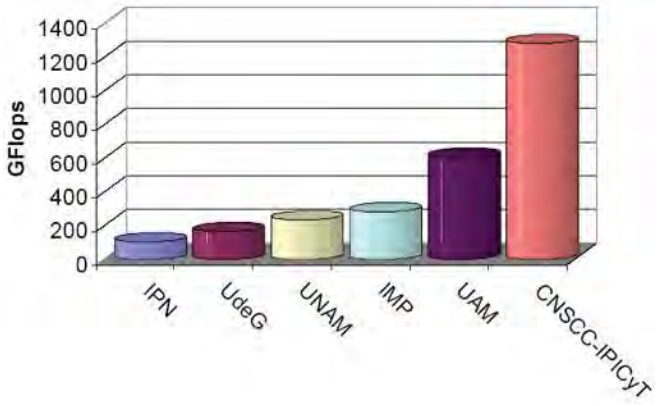


Figura D1 Comparativo del poder combinado de procesamiento por cada uno de los centros de supercómputo en México hasta 2004.

El equipo Cray XD1, cuenta con 216 procesadores AMD Opteron 275 Dual Core, x86-64 con una velocidad de reloj de 2.2 GHz, que realizan dos operaciones en punto flotante por ciclo de reloj, configurados en nueve módulos de 24 procesadores cada uno. Posee un rendimiento teórico pico combinado de 950 GFlops y una de las latencias de intercomunicación mas bajas del mercado 1.7 µs. La arquitectura del equipo Cray XD1 permite desarrollos en paralelo bajo los esquemas de memoria compartida y/o distribuida (ShareMemory, OpenMP y MPI respectivamente). El sistema cuenta con 216 GB de memoria RAM total, con la posibilidad de que un solo procesador pueda acceder al total de esta. Posee un almacenamiento primario de 4.4 TBytes y opera bajo el sistema operativo Linux de la distribución SuSE Linux Enterprise Server 9.0. Este sistema operativo soporta la creación de archivos tan grandes como 1 TByte. Las características de la supercomputadora Cray XD1 se muestran en la Tabla D2 [CNSCC- IPICyT 2004].

Tabla D2 Características de la supercomputadora Cray XD1.

CARACTERISTICAS DE Cray XD1	
No. Procesadores (Tipo)	216 AMD Opteron 2.2 GHz
Memoria RAM Total	216 GB
Almacenamiento Interno Total	4.4 TB
Sistema Operativo	Linux
Registros de Memoria por Procesador	96 GB
Flops por procesador	4.4 GFlops
Ancho de Banda hacia la memoria	8000 MB/s
Latencia	1.7 µs
Rendimiento Teórico Pico	950 GFlops
Opciones de Actualización procesador	> 2.4 GHz
Disponibilidad de partes	El total en existencia
Perspectiva de Uso (a la fecha)	6 años
Costo anual mantenimiento	$50,000.00 USD

La UNAM posee la computadora *KanBalam*,[1] con capacidad para procesar más de siete billones de operaciones de punto flotante por segundo, misma que es utilizada para investigaciones en áreas como astrofísica, física de partículas, química cuántica, estudios del clima y la contaminación, ingeniería sísmica, geología o ciencias biológicas y de materiales [UNAM 2008]. Este equipo de supercómputo cuenta con 1368 procesadores Opteron de 2.6 GHz de AMD , una memoria RAM total de 3 mil gigabytes y un sistema de almacenamiento de 160 terabytes. Los 1368 procesadores están organizados en 337 nodos, cada uno con dos procesadores de dos núcleos y ocho gigabytes de memoria, y éstos, así como los 768 discos de almacenamiento se comunican a una velocidad de 10 gigabytes por segundo. KanBalam es siete mil veces más potente que la primera supercomputadora adquirida por la UNAM en 1991, la CRAY-YMP, y 79 veces más poderosa en cálculo que el equipo AlphaServer SC45, adquirido en 2003 [UNAM 2008]. Los equipos centrales de supercómputo de la UNAM se observan en la Tabla D2. Los equipos de supercómputo para proyectos específicos o docencia se muestran en la Tabla D3.

Tabla D2 Equipos centrales de supercómputo en la UNAM

NOMBRE	MODELO	DESCRIPCIÓN GENERAL
KANBALAM	HP CP 4000	Supercomputadora paralela de memoria distribuida. Contiene 1368 procesadores AMD Opteron, alrededor de 3 Terabytes de memoria y 160 Terabytes de almacenamiento.
ALEBRIJE	SGI Altix 350	Supercomputadora paralela de memoria compartida. Contiene 24 procesadores Intel Itanium 2, 24 Gigabytes de memoria y 2.4 Terabytes de almacenamiento.
BAKLIZ	HP AlphaServer SC 45	Supercomputadora paralela de memoria distribuida. Contiene 36 procesadors Alpha EV68, 56 Gigabytes de memoria y 1 Terabyte de almacenamiento.

Tabla D3 Equipos de supercómputo ara proyectos específicos o docencia en la UNAM

NOMBRE	MODELO	DESCRIPCIÓN GENERAL
IXCHEL	HP CP 6000	Cluster con 16 procesadores Intel Itanium 2, 16 Gigabytes de memoria y 2 Terabytes de almacenamiento.
MACONDO	Apple Mac Xserve	Cluster con 8 procesadores G5, 8 Gigabytes de memoria y 800 Gigabytes de almacenamiento.
MIXBAAL	Cluster Intel Pentium III	Cluster con 48 procesadores Intel PIII, 24 Gigabytes de memoria y 2 Terabytes de almacenamiento.
BERENICE	SGI Origin 2000	Computadora de memoria compartida, con 40 procesadores MIPS R10K, 10 Gigabytes de RAM y 1.5 Terabyes de almacenamiento.

[1] KanBalam es denominada así en honor a un matemático maya reconocido por la precisión en sus cálculos relacionados con el dominio del tiempo.

REFERENCIAS

[Abur 1988]
Abur A., "*A Parallel Scheme for the Forward/Backward Substitutions in Solving Sparse Linear Equations*", *IEEE Transactions on Power Systems*, PWRS-3, 1988.

[Alvarado 1979]
Alvarado F., "*Parallel Solution of Transient Problems by Trapezoidal Integration*", *IEEE Transactions on Power Systems*, May/June 1979.

[Alvarado *et al.* 1990]
Alvarado F., Yu D., Betancourt R., "*Partitioned Sparse Methods*", *IEEE Transactions on Power Systems*, May 1990.

[Aprille y Trick 1972]
Aprille T.J. y Trick T. N. "*A Computer Algorithm to Determine the Steady State Response of Nonlinear Oscillators*", IEEE *Transactions on Circuit Theory*, Vol. CT-19, No. 4, July 1972, pp. 354-360.

[Arrillaga *et al.* 1985]
Arrillaga J., Smith B., Watson N. and Wood A., "*Power Systems Harmonic Analysis*", John Wiley and Sons, 1985.

[Ascher *et al.* 1991]
Ascher M., Uri R. and Petzold Linda, "*Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*", SIAM, Philadelphia 1991, pp. 57-58.

[Asenjo 1997]
Asenjo P. Rafael. "*Factorización LU de Matrices Dispersas en Multiprocesadores*". Reporte Técnico de Tesis Doctoral UMA-DAC-97/32. Departamento de Arquitectura de Computadores, Universidad de Málaga, España, Diciembre 1997.

[BEOWULF1 2003]
NASA. *NASA described for Ethernet and the Beowulf project*, Mars 20th 2003, USA. http://cesdis.gsfc.nasa.gov/linux/beowulf/beowulf.html.

[BEOWULF2 2003]
Tutorial Beowulf. "*How to Build a Beowulf*", Mars 20th 2003, USA. http://www.cacr.caltech.edu/beowulf/tutorial/building.html.

[Chai *et al.* 1991]
Chai J.S., Zhu N., Bose A. and Tylavsky D.J., "*Parallel Newton Type Methods for Power System Stability Analysis Using Local and Shared Memory Multiprocessors*", *IEEE PES Winter Power Meeting*, New York, February 1991.

[Chapra 2003]
Chapra Steven, Canale Raymond., "*Métodos numéricos para ingenieros*", McGraw-Hill, México 2003, pág. 287. ISNB 9701039653.

[Chaves *et. al* 1999]
Cháves Juan Luis (CeCalCULA), Echeverría Carlos (ULA), Correa Manuel (UNET), Vásquez Edgar (UNET), Suárez Ma. Eugenia (USB), Mármol Xavier (LUZ), Herdes Carmelo (USB) y Díaz Gilberto (CeCalCULA), "*Instalación y Uso del Cluster Beowulf*", Universidad de Los Andes, Centro de Cálculo Científico. Mérida, Venezuela, Julio, 1999.

[Chua y Ushida 1981]
Chua L.O. y Ushida A., "*Algorithms for Computing Almost Periodic Steady-State Response of Nonlinear Systems to Multiple Input Frequencies*", *IEEE Transactions on Circuits and Systems*, Vol. CAS-28, No. 10, October 1981.

[CNSCC- IPICyT 2004]
Centro Nacional de Supercomputo de la Red de Centros Públicos de Investigación CONACYT. Boletín informativo, Acuerdo S-JD-O-I04-13, 3 de Mayo de 2004,

[Davis 2006]
Davis T., *"Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms)"*, Society for Industrial and Applied Mathematic.

[Duncan 1990].
Duncan Ralph, "*A Survey of Parallel Computer Architectures*", *IEEE Computer*, February 1990.

[FEDORA 2003]
FEDORA LINUX. *Fedora User's Guide Version 7.0.3*. Mars 2003, USA. Disponible en: http://www.fedora.org

[Feng Tu and Flueck 2002]
Feng Tu and Flueck A.J., "*A Message-passing distribuited-memory Newton-GMRES Parallel Power Flow Algorithm*", *IEEE Power Engineering Society Summer Meeting* 2002, Chicago, USA,Vol.3 pp. 1477-1482.

[Fernández 1996]
Fernandez, F., "*Sistemas Computacionales Masivamente Paralelos: Grandes Capacidades de Cálculo*". Departamento de Sistemas y Computación. Universidad de los Andes, Colombia. Revista *Sistemas* No. 67, Año 3, Abril-Junio 1996, págs. 13-17.

[Flynn 1972]
Flynn M., "*Some Computer Organizations and their Effectiveness*", *IEEE Transactions on Computers*, September 1972.

[Foster 1995]
Ian Foster, "*Designing and Building Parallel Programs"*, Addison-Wesley, 1995, ISBN 0-201-57594-9.

[Gálvez y Gonzáles 1993]
Gálvez Javier, Gonzáles Juan. "*Algorítmica, Análisis y Diseño de Algoritmos*", Editorial RA-MA (Addison-Wesley Iberiamericana), 2a Edición, USA, Septiembre 1993, pág. 502.

[García y Acha 2004]
García N. Acha E., "*Periodic Steady-State Analysis of Large Scale Electric Systems Using Poincaré Map and Parallel Processing*", *IEEE Transactions on Power Systems*, Vol. 19, No. 4, November 2004, pp. 1784-1793.

[García *et al.* 2000]
García Sigridt, Medina Aurelio, Pérez Carlos, "*A State Space Single-Phase Transformer Model Incorporating Nonlinear Phenomena of Magnetic Saturation and Hysteresis for Transient and Periodic Steady-State Analysis*", *Proceedings of the IEEE PES 2000 Summer Meeting*, July 2000, U.S.A., Vol. 4, pp. 2417-2421.

[García *et al.* 2001]
García N., Acha E. and Medina A., "*Swift Time Domain Solutions of Electrics Systems Using Parallel Processing*". University of Glasgow, UK. 2001.

[García y Medina 2003]
Garcia, N. and Medina, A., "*Swift Time Domain Solution of Electric Systems Including SVSs*". *IEEE Transactions on Power Delivery*, Vol. 18, No. 3, July 2003, pp. 921-927.

[García 2005]
García Norberto, "*Parallel Harmonic-Oriented Method for Large-Scale Electric Systems Based on MPI programming and the Limit Cycle Method", IEEE PowerTech 2005,* San Petersburgo, Rusia, Junio 27-30 2005, págs. 1-6.

[García 2006]
Garcia Norberto, "*A Parallel Processing and Newton Based Approach for Harmonic and Power Quality Analysis of Large-Scale Electric Systems Including STATCOM´s". WSEAS Transactions on Power Systems,* Vol. 1, No. 1, Enero 2006, págs. 134-137.

[Gerald 2000]
Gerald F., Wheatley O., "*Análisis Numérico con Aplicaciones*", Pearson Educación, México 2000, pp. 459. ISNB 9684443935

[GNUPLOT 2006]
GNU. *Tutorial en línea de GNUPLOT*. Enero 2006. Disponible en: http://www.gnuplot.info

[Gomez y Bentacourt 1990]
Gomez A. and Bentacourt R., "*Implementation of the Fast Decoupled Load Flow on a Vector Computer*", IEEE/PES Winter Meeting, Atlanta 1990.

[Hatcher *et al.* 1997]
Hatcher W.L., Brasch F.M. and Van Ness J.E., "*A Feasiblility Study for Solution of Transient Stability Problems by Multiprocessor Structures*", *IEEE Transactions on Power Systems*, Nov/Dec 1997.

[Heileman 1998].
Heileman G. L., "*Estructuras de Datos, Algoritmos, y Programación Orientada a Objetos*", McGraw Hill. 1998.

[Hoeger 2002]
Hoeger Herbert, *"Introducción a la Computación Paralela"*, Centro Nacional de Cálculo Científico Universidad de Los Andes, Mérida, Venezuela 2002.

[Hornbeck 1975]
Hornbeck R., "*Numerical Methods*", Quantum Publishers Inc., USA 1975.

[Hwang y Briggs 1984]
Kai Hwang y Fay´e A. Briggs, "*Arquitectura de Computadoras y Procesamiento Paralelo*". McGraw-Hill, 1987.

[Inmos Ltd 1990]
Inmos Ltd., "*Transputer Reference Manual*", Inmos U.K., 1990.

[Jájá 1992]
Jájá Joseph, "*An Introduction to Parallel Algorithms"*, Addison Wesley, 1992.

[Jin 1995].
Jin Lan, "*Parallel Processing: Exploring the architectures and algorithms. Close relationship*", *IEEE Potencials*, December'94/January'95, pp 17-20.

[Kaiser *et al*. 1989]
Kaiser W., Käch M. and Feucht D., "*A Load-Flow-Machine on a Transputer System*", PRO. LEI.TEC. AG., Lielistrasse 4, 8903 Birmensdorf, Switerland, 1989 pp. 597-604.

[Kunder *et al.* 1990].
Kundert K.S., White J.K. y Sangiovanni-Vincentelli A. "*Steady-State Methods for Simulating Analog and Microwave Circuits*", Kluver Academic Publishers, USA 1990.

[LAM-MPI 2003]
LAM/MPI. *User's Guide. Version 7.0.3*., Inidiana University, USA. November, 2003. Disponible en: http://www.lam-mpi.org.

[LaScala *et al.* 1989]
LaScala M., Bose A., Tylavsky D.J. and Chai J.S., "*A highly Parallel Method for Transient Stability Analysis*", *IEEE Power Industry Computer Applications Conference*, Seattle, May 1989.

[LaScala *et al.* 1990]
LaScala M., Brucoli M., Torelli F. and Trovato M., "*A Gauss-Jacobi-block-Newton Method for Parallel Transient Stability Analysis*", *IEEE PES Winter Meeting*, Atlanta, February 1990.

[LaScala y Sbrizzai 1990]
LaScala M., Sbrizzai R. and Torelli F., "*A Pipelined-in Time Algorithm for Transient Stability Analysis*", *IEEE PES Summer Meeting*, Minneapolis, July 1990.

[Lau *et al.* 1990]
Lau K., Tylavsky D.J. and Bose A., "*Coarse Grain Scheduling in Parallel Triangular Factorization and Solution of Power System Matrices*", *IEEE /PES Summer Meeting*, Minneapolis, July 1990.

[Lee *et al*. 1989]
Lee S.Y., Chiang H.D. and Ku B.Y., "*Parallel Power System Transient Stability Analysis on Hipercube Multiprocessors*", *IEEE Transactions on Power Industry Computer Applications Conference*, Seattle, May 1989.

[Mariños *et al*. 1994]
Mariños Z.A., Pereira J .L.R., Carneiro J., "*Fast Harmonic Power Flow Calculation Using Parallel Processing*", *IEEE Proc.-Gener. Transm. Distrib.*, Vol 141, No. 1. January 1994, pp. 27-32.

[Medina 2001]
Medina A., "*Metodologías Avanzadas para el Modelado y Análisis de Armónicos y su Impacto en la Calidad de la Energía*", CIGRÉ-México, Bienal 2001.

[Medina *et al*. 2003]
Medina, A.; Ramos-Paz, A.; Fuerte-Esquivel, C.R., "*Periodic Steady State Solution of Electric Systems with Nonlinear Components Using Parallel Processing*", *IEEE Transactions on Power Systems,* Vol. 18, No. 2, May 2003, pp. 963 – 965.

[Medina, Ramos-Paz y Fuerte-Esquivel 2003]
Medina A., Ramos-Paz A., and Fuerte-Esquivel C. R., "*Swift Computation of the Periodic Steady State Solution of Power Systems Containing TCSCs*", Electrical Power and Energy Systems, No. 25, 2003, pp. 689-694.

[Medina y García 2004]
Medina A., y García N., "*Fast Time Domine Computation of the Periodic Steady-state of Systems with Nonlinear and Time-varying Components*", International Journal of Electrical Power and Energy Systems, Elsevier Science Ltd., Vol. 26, 2003, Oct. 2004, pp. 637-643.

[Medina *et al*. 2004]
Medina, A., Ramos-Paz A. Mora-Juarez R., Fuerte-Esquivel C.R., "*Object Oriented Programming Techniques Applied to Conventional and Fast Time Domain Algorithms for the Steady State Solution of Nonlinear Electric Networks*", Power Engineering Society General Meeting 2004 IEEE, Vol. 1, June 6-10, 2004, pp. 342 – 346.

[Medina *et al*. 2004b]
Medina A, Ramos-Paz A., Martínez-Cárdenas F., Silva-Chávez J.C., "*Computer Platform Comparison on the Dynamic Operation Simulation of the Synchronous Machine*", 8th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2004) Orlando USA, July 18-21, 2004.

[Medina *et. al* 2006]
Medina, A., Ramos-Paz, A., Fuerte-Esquivel C.R., "*Efficient Computation of the Periodic Steady State Solution of Nonlinear Electric Systems Applying Parallel Processing Techniques*", Revista "*The International Journal for Computation and Mathematics in Electrical & Electronic Engineering*", Vol. 25, No. 4, 2006. pp. 900-910.

[Medina y Ramos-Paz 2005]
Medina A., Ramos-Paz A., "*PVM and MT Parallel Processing Platforms Applied to the Computation of Driving Point Impedances in Power Systems*", WSEAS Transactions on Circuits and Systems. No. 11, Vol. 4, November 2005, pp. 1702-1709.

[Medina 2008]
Medina Ríos V. Tesis "*Diseño, Construcción y Operación de una Supercomputadora No Convencional para Aplicaciones de Cómputo Avanzado*", Tesis de Maestría, Facultad de Ingeniería Eléctrica, Universidad Michoacana de San Nicolás de Hidalgo, México. 2008.

[Miltos *et al*. 2000]
Grammatikakis Milton D., Hsu D. Frank and Kraetzl Miroslav, "*Parallel Systems Interconnections and Communications*", CRC Edition, December 2000, pp 416, ISBN 0849331536.

[MOSIX 1998]
Barak A. and La´adan O. "*The MOSIX Multicomputer Operative System for High Performance Cluster Computing*". Journal of Future Generation Computer Systems (13) 4-5 pp. 361-373. March 1998.

[Nakamura 1997]
Nakamura S., "*Análisis Numérico y Visualización Gráfica con MATLAB*", Prentice Hall Hispanoamericana S.A. 1997, pp. 344 – 359. ISBN 968-880-860-1.

[Nakhla y Branin 1977]
Nakhla M. S. y Branin F. H., "*Determining the Periodic Response of Nonlinear Systems by a Gradient Method*", Circuit Theory Appl., Vol. 5, 1977, pp. 255 – 273.

[Oyama *et al.* 1990]
Oyama T., Kitahara T., Serizawa Y., "*Parallel Processing for Power System Analysis Using Band Matrix*", *IEEE Transactions on Power Systems*, Vol. 5, No. 3, August 1990, pp. 1010-1016.

[Pardo 2002]
Pardo Fernando. "*Arquitecturas Avanzadas*", Facultad de Ingeniería Informática, Universidad de Valencia, España 2002.

[Parker y Chua 1989]
Parker T.S. and Chua L.O., "*Practical Numerical Alghorithms for Chaotic Systems*", Spring-Verlag, New York, 1989.

[Parkhurst y Ogborn 1995]
Parkhurst Jeffrey R. and Ogborn Lawrence L., "*Determining the Steady-State Output of Nonlinear Oacillatory Circuits Using Multiple Shooting*", *IEEE Transactions on Computer-Aided Desing of Integrated Circuits and Systems.* Vol. 14, No. 7, July 1995, pp. 882-889.

[Plaza 2003]
Plaza Emilio J., "*Cluster Heterogéneo de Computadoras*", Centro Nacional de Cálculo Científico Universidad de Los Andes, Venezuela 2003.

[Pottle 1988]
Pottle C., "*Prospects for the Real-Time Simulation of Bulk Power Systems*", Grainger Lectures Series, Urbana IL, May 1988, pp. 82-89.

[PVM 1994]
Geist A. *et al*. "*PVM: Parallel Virtual Machine. A User´s Guide and Tutorial for Networked Parallel Computing*", The MIT Press., Gambridge Mass. USA 1994, ISBN 0-262-57108-0.

[Ramos 2002]
Ramos Paz A. "*Aplicación de Técnicas de Procesamiento en Paralelo a la Solución Eficiente en Estado Estacionario en el Dominio del Tiempo de Sistemas Eléctricos No Lineales y Variantes en el Tiempo*", Tesis de Maestría, Facultad de Ingeniería Eléctrica, Universidad Michoacana de San Nicolás de Hidalgo, México 2002.

[Ramos 2007]
Ramos Paz A. "*Técnica para la Generación Automática de Ecuaciones Diferenciales No Autónomas para Representar el Comportamiento Dinámico de Sistemas Eléctricos No-Lineales Incorporando Herramientas Avanzadas de Computo*", Tesis de Doctorado, Facultad de Ingeniería Eléctrica, Universidad Michoacana de San Nicolás de Hidalgo, México 2007.

[Rodriguez y Medina 2004]
Rodriguez O., Medina A., "*Efficient Methodology for the Transient and Periodic Steady-State Analysis of the Synchronous Machine Using a Phase Coordinates Model*", *IEEE Transactions on Energy Conversion*, Vol. 19, No. 2, June 2004, pp. 464 – 466.

[Semlyen y Medina 1995]
Semlyen A. and Medina A., "*Computation of the Periodic Steady State in Systems with Nolinear Components Using a Hybrid Time and Frequency Domain Methodology*", *IEEE Transaccions on Power Systems*, Vol. 10, No. 3, August 1995, pp. 1498-1504.

[Sean 1998]
Sean, James A., "*Análisis y Diseño de Sistemas de Información*", Editorial MacGraw Hill, 2a. Edición USA, Diciembre/1998, México, pág. 941.

[Skelboe 1980]
Skelboe S., "*Computation of the Periodic Steady-State Response of Nonlinear Networks by Extrapolation Methods*", *IEEE Transactions on Circuits and Systems*, Vol. CAS-27, March 1980, pp. 161-175.

[Stallings 1998]
Stallings William, "*Computer Organization & Architecture*", 4ª Edition, Prentice Hall, U.K. 1998.

[Sterling 2003]
Sterling Thomas. "*Beowulf Cluster Computing with Linux*", 2nd. Edition by William Group, Ewing Lusk and Thomas Sterling (eds.), The MIT Press, 2003, pp 618, ISBN 0262692929.

[Stavrakakis *et. al* 1990]
Stavrakakis G.S., Lefas C., Pouliezos A., "*Parallel Processing Computer Implementation of a Real Time DC Motor Drive Fault Detection Algorithm*", *IEEE Proceedings*, Vol. 137, No. 5, September 1990, pp. 309-313.

[UNAM 2008]
UNAM. Supercomputo UNAM, 18 de Noviembre de 2008, México.
http://www.super.unam.mx/index.php?option=com_content&task=view&id=87&Itemid=122

[Swendson 2004]
Swendson K., "*The Beowulf HOWTO*", GNU Free Software Foundation, May 2004. Disponible en: htttp://www.beowulf.org

[Tanenbaum 1995]
Tanenbaum S. Andrew, "*Distributed Operating Systems*", Prentice-Hall, Eaglewood Cliffs, New Jersey, 2nd edition, 1996.

[Thomson 1992]
Thomson A., "*Introduction to Parallel Algorithms & Architectures: Arrays, Trees & Hypercubes*", Morgan Kaufmman Publishers, 1992, ISBN 1-55860-117-1.

[TOP500 2008]
TOP500 list. November 2008 | TOP500 Supercomputing site, November 18th, 2008, PressRelease200811.pdf, http://top500.org

[Torres 2003]
Torres D., Notas del curso "*Procesamiento en Paralelo*". DEP-FIE. Agosto 2002 - Febrero 2003. Morelia, México. 2003.

[Tulukdar y Cardozo 1988]
Tulukdar S.N. and Cardozo E., "*An Environment for Rule-based Blackboards and Distributed Problem Solving*", *Readings in Distributed Artificial Intelligence*, Morgan Kaufmman Publishers, San Mateo CA. 1988.

[Werler y Glavitsch 1993]
Werler K. y Glavitsch H., "*Computation of Transients by Parallel Processing", IEEE Transactions on Power Delivery*, Vol. 8, No. 3, July 1993, pp. 137 - 145

[WIKIPEDIA 2003]
WIKIPEDIA. *Wikipedia en español*. 2003. http://es.wikipedia.org

[Wylie 1951].
Wylie C.R., "*Advanced Engineering Mathematics*", McGraw Hill. Japan. 1951.

CASO DE ESTUDIO 3

Las ecuaciones de Estado que describen el sistema eléctrico del Caso de Estudio 3 se presentan en su forma de código en C++, tal y como fueron generadas utilizando la herramienta computacional de [Ramos 2004]:

```
void caso_estudio_3(void){

ec=new ecuacion[Ve];

ec[0].anexa_termino(new termino(-0.030300/0.099900,0,1));
ec[0].anexa_termino(new termino(1/0.099900,184,1));
ec[0].anexa_termino(new termino(-1/0.099900,185,1));

ec[1].anexa_termino(new termino(-0.012900/0.042400,1,1));
ec[1].anexa_termino(new termino(1/0.042400,184,1));
ec[1].anexa_termino(new termino(-1/0.042400,186,1));

ec[2].anexa_termino(new termino(-0.001760/0.007980,2,1));
ec[2].anexa_termino(new termino(1/0.007980,187,1));
ec[2].anexa_termino(new termino(-1/0.007980,188,1));

ec[3].anexa_termino(new termino(-0.024100/0.108000,3,1));
ec[3].anexa_termino(new termino(1/0.108000,186,1));
ec[3].anexa_termino(new termino(-1/0.108000,188,1));

ec[4].anexa_termino(new termino(-0.011900/0.054000,4,1));
ec[4].anexa_termino(new termino(1/0.054000,188,1));
ec[4].anexa_termino(new termino(-1/0.054000,189,1));

ec[5].anexa_termino(new termino(-0.004590/0.020800,5,1));
ec[5].anexa_termino(new termino(1/0.020800,189,1));
ec[5].anexa_termino(new termino(-1/0.020800,190,1));

ec[6].anexa_termino(new termino(-0.002440/0.030500,6,1));
ec[6].anexa_termino(new termino(1/0.030500,191,1));
ec[6].anexa_termino(new termino(-1/0.030500,192,1));

ec[7].anexa_termino(new termino(-0.002580/0.032200,7,1));
ec[7].anexa_termino(new termino(1/0.032200,192,1));
ec[7].anexa_termino(new termino(-1/0.032200,193,1));

ec[8].anexa_termino(new termino(-0.020900/0.068800,8,1));
ec[8].anexa_termino(new termino(1/0.068800,187,1));
ec[8].anexa_termino(new termino(-1/0.068800,194,1));

ec[9].anexa_termino(new termino(-0.020300/0.068200,9,1));
ec[9].anexa_termino(new termino(1/0.068200,188,1));
ec[9].anexa_termino(new termino(-1/0.068200,194,1));

ec[10].anexa_termino(new termino(-0.005950/0.019600,10,1));
ec[10].anexa_termino(new termino(1/0.019600,194,1));
ec[10].anexa_termino(new termino(-1/0.019600,195,1));

ec[11].anexa_termino(new termino(-0.018700/0.061600,11,1));
ec[11].anexa_termino(new termino(1/0.061600,185,1));
ec[11].anexa_termino(new termino(-1/0.061600,195,1));

ec[12].anexa_termino(new termino(-0.048400/0.160000,12,1));
ec[12].anexa_termino(new termino(1/0.160000,186,1));
ec[12].anexa_termino(new termino(-1/0.160000,195,1));

ec[13].anexa_termino(new termino(-0.008620/0.034000,13,1));
ec[13].anexa_termino(new termino(1/0.034000,190,1));
ec[13].anexa_termino(new termino(-1/0.034000,195,1));

ec[14].anexa_termino(new termino(-0.022250/0.073100,14,1));
ec[14].anexa_termino(new termino(1/0.073100,194,1));
```

```
ec[14].anexa_termino(new termino(-1/0.073100,196,1));

ec[15].anexa_termino(new termino(-0.021500/0.070700,15,1));
ec[15].anexa_termino(new termino(1/0.070700,195,1));
ec[15].anexa_termino(new termino(-1/0.070700,197,1));

ec[16].anexa_termino(new termino(-0.074400/0.244400,16,1));
ec[16].anexa_termino(new termino(1/0.244400,196,1));
ec[16].anexa_termino(new termino(-1/0.244400,198,1));

ec[17].anexa_termino(new termino(-0.059500/0.195000,17,1));
ec[17].anexa_termino(new termino(1/0.195000,197,1));
ec[17].anexa_termino(new termino(-1/0.195000,198,1));

ec[18].anexa_termino(new termino(-0.021200/0.083400,18,1));
ec[18].anexa_termino(new termino(1/0.083400,195,1));
ec[18].anexa_termino(new termino(-1/0.083400,199,1));

ec[19].anexa_termino(new termino(-0.013200/0.043700,19,1));
ec[19].anexa_termino(new termino(1/0.043700,198,1));
ec[19].anexa_termino(new termino(-1/0.043700,200,1));

ec[20].anexa_termino(new termino(-0.045400/0.180100,20,1));
ec[20].anexa_termino(new termino(1/0.180100,199,1));
ec[20].anexa_termino(new termino(-1/0.180100,200,1));

ec[21].anexa_termino(new termino(-0.012300/0.050500,21,1));
ec[21].anexa_termino(new termino(1/0.050500,200,1));
ec[21].anexa_termino(new termino(-1/0.050500,201,1));

ec[22].anexa_termino(new termino(-0.011190/0.049300,22,1));
ec[22].anexa_termino(new termino(1/0.049300,201,1));
ec[22].anexa_termino(new termino(-1/0.049300,202,1));

ec[23].anexa_termino(new termino(-0.025200/0.117000,23,1));
ec[23].anexa_termino(new termino(1/0.117000,202,1));
ec[23].anexa_termino(new termino(-1/0.117000,203,1));

ec[24].anexa_termino(new termino(-0.012000/0.039400,24,1));
ec[24].anexa_termino(new termino(1/0.039400,198,1));
ec[24].anexa_termino(new termino(-1/0.039400,202,1));

ec[25].anexa_termino(new termino(-0.018300/0.084900,25,1));
ec[25].anexa_termino(new termino(1/0.084900,203,1));
ec[25].anexa_termino(new termino(-1/0.084900,204,1));

ec[26].anexa_termino(new termino(-0.020900/0.097000,26,1));
ec[26].anexa_termino(new termino(1/0.097000,204,1));
ec[26].anexa_termino(new termino(-1/0.097000,205,1));

ec[27].anexa_termino(new termino(-0.034200/0.159000,27,1));
ec[27].anexa_termino(new termino(1/0.159000,205,1));
ec[27].anexa_termino(new termino(-1/0.159000,206,1));

ec[28].anexa_termino(new termino(-0.013500/0.049200,28,1));
ec[28].anexa_termino(new termino(1/0.049200,206,1));
ec[28].anexa_termino(new termino(-1/0.049200,207,1));

ec[29].anexa_termino(new termino(-0.015600/0.080000,29,1));
ec[29].anexa_termino(new termino(1/0.080000,206,1));
ec[29].anexa_termino(new termino(-1/0.080000,208,1));

ec[30].anexa_termino(new termino(-0.031800/0.163000,30,1));
ec[30].anexa_termino(new termino(1/0.163000,208,1));
ec[30].anexa_termino(new termino(-1/0.163000,210,1));

ec[31].anexa_termino(new termino(-0.019130/0.085500,31,1));
ec[31].anexa_termino(new termino(1/0.085500,210,1));
ec[31].anexa_termino(new termino(-1/0.085500,211,1));

ec[32].anexa_termino(new termino(-0.023700/0.094300,32,1));
```

```
ec[32].anexa_termino(new termino(1/0.094300,211,1));
ec[32].anexa_termino(new termino(-1/0.094300,212,1));

ec[33].anexa_termino(new termino(-0.004310/0.050400,33,1));
ec[33].anexa_termino(new termino(1/0.050400,191,1));
ec[33].anexa_termino(new termino(-1/0.050400,213,1));

ec[34].anexa_termino(new termino(-0.007990/0.086000,34,1));
ec[34].anexa_termino(new termino(1/0.086000,209,1));
ec[34].anexa_termino(new termino(-1/0.086000,213,1));

ec[35].anexa_termino(new termino(-0.047400/0.156300,35,1));
ec[35].anexa_termino(new termino(1/0.156300,200,1));
ec[35].anexa_termino(new termino(-1/0.156300,214,1));

ec[36].anexa_termino(new termino(-0.010800/0.033100,36,1));
ec[36].anexa_termino(new termino(1/0.033100,212,1));
ec[36].anexa_termino(new termino(-1/0.033100,214,1));

ec[37].anexa_termino(new termino(-0.031700/0.115300,37,1));
ec[37].anexa_termino(new termino(1/0.115300,206,1));
ec[37].anexa_termino(new termino(-1/0.115300,215,1));

ec[38].anexa_termino(new termino(-0.029800/0.098500,38,1));
ec[38].anexa_termino(new termino(1/0.098500,214,1));
ec[38].anexa_termino(new termino(-1/0.098500,215,1));

ec[39].anexa_termino(new termino(-0.022900/0.075500,39,1));
ec[39].anexa_termino(new termino(1/0.075500,210,1));
ec[39].anexa_termino(new termino(-1/0.075500,215,1));

ec[40].anexa_termino(new termino(-0.038000/0.124400,40,1));
ec[40].anexa_termino(new termino(1/0.124400,198,1));
ec[40].anexa_termino(new termino(-1/0.124400,216,1));

ec[41].anexa_termino(new termino(-0.075200/0.247000,41,1));
ec[41].anexa_termino(new termino(1/0.247000,202,1));
ec[41].anexa_termino(new termino(-1/0.247000,217,1));

ec[42].anexa_termino(new termino(-0.002240/0.010200,42,1));
ec[42].anexa_termino(new termino(1/0.010200,218,1));
ec[42].anexa_termino(new termino(-1/0.010200,219,1));

ec[43].anexa_termino(new termino(-0.011000/0.049700,43,1));
ec[43].anexa_termino(new termino(1/0.049700,218,1));
ec[43].anexa_termino(new termino(-1/0.049700,220,1));

ec[44].anexa_termino(new termino(-0.041500/0.142000,44,1));
ec[44].anexa_termino(new termino(1/0.142000,216,1));
ec[44].anexa_termino(new termino(-1/0.142000,220,1));

ec[45].anexa_termino(new termino(-0.008710/0.026800,45,1));
ec[45].anexa_termino(new termino(1/0.026800,217,1));
ec[45].anexa_termino(new termino(-1/0.026800,219,1));

ec[46].anexa_termino(new termino(-0.002560/0.009400,46,1));
ec[46].anexa_termino(new termino(1/0.009400,217,1));
ec[46].anexa_termino(new termino(-1/0.009400,220,1));

ec[47].anexa_termino(new termino(-0.032100/0.106000,47,1));
ec[47].anexa_termino(new termino(1/0.106000,220,1));
ec[47].anexa_termino(new termino(-1/0.106000,222,1));

ec[48].anexa_termino(new termino(-0.059300/0.168000,48,1));
ec[48].anexa_termino(new termino(1/0.168000,220,1));
ec[48].anexa_termino(new termino(-1/0.168000,223,1));

ec[49].anexa_termino(new termino(-0.004640/0.054000,49,1));
ec[49].anexa_termino(new termino(1/0.054000,213,1));
ec[49].anexa_termino(new termino(-1/0.054000,221,1));
```

```
ec[50].anexa_termino(new termino(-0.018400/0.060500,50,1));
ec[50].anexa_termino(new termino(1/0.060500,222,1));
ec[50].anexa_termino(new termino(-1/0.060500,223,1));

ec[51].anexa_termino(new termino(-0.014500/0.048700,51,1));
ec[51].anexa_termino(new termino(1/0.048700,223,1));
ec[51].anexa_termino(new termino(-1/0.048700,224,1));

ec[52].anexa_termino(new termino(-0.055500/0.183000,52,1));
ec[52].anexa_termino(new termino(1/0.183000,223,1));
ec[52].anexa_termino(new termino(-1/0.183000,225,1));

ec[53].anexa_termino(new termino(-0.041000/0.135000,53,1));
ec[53].anexa_termino(new termino(1/0.135000,224,1));
ec[53].anexa_termino(new termino(-1/0.135000,225,1));

ec[54].anexa_termino(new termino(-0.060800/0.245400,54,1));
ec[54].anexa_termino(new termino(1/0.245400,226,1));
ec[54].anexa_termino(new termino(-1/0.245400,227,1));

ec[55].anexa_termino(new termino(-0.041300/0.168100,55,1));
ec[55].anexa_termino(new termino(1/0.168100,217,1));
ec[55].anexa_termino(new termino(-1/0.168100,226,1));

ec[56].anexa_termino(new termino(-0.022400/0.090100,56,1));
ec[56].anexa_termino(new termino(1/0.090100,227,1));
ec[56].anexa_termino(new termino(-1/0.090100,228,1));

ec[57].anexa_termino(new termino(-0.040000/0.135600,57,1));
ec[57].anexa_termino(new termino(1/0.135600,228,1));
ec[57].anexa_termino(new termino(-1/0.135600,229,1));

ec[58].anexa_termino(new termino(-0.038000/0.127000,58,1));
ec[58].anexa_termino(new termino(1/0.127000,229,1));
ec[58].anexa_termino(new termino(-1/0.127000,230,1));

ec[59].anexa_termino(new termino(-0.060100/0.189000,59,1));
ec[59].anexa_termino(new termino(1/0.189000,229,1));
ec[59].anexa_termino(new termino(-1/0.189000,231,1));

ec[60].anexa_termino(new termino(-0.019100/0.062500,60,1));
ec[60].anexa_termino(new termino(1/0.062500,230,1));
ec[60].anexa_termino(new termino(-1/0.062500,232,1));

ec[61].anexa_termino(new termino(-0.071500/0.323000,61,1));
ec[61].anexa_termino(new termino(1/0.323000,225,1));
ec[61].anexa_termino(new termino(-1/0.323000,232,1));

ec[62].anexa_termino(new termino(-0.071500/0.323000,62,1));
ec[62].anexa_termino(new termino(1/0.323000,225,1));
ec[62].anexa_termino(new termino(-1/0.323000,232,1));

ec[63].anexa_termino(new termino(-0.068400/0.186000,63,1));
ec[63].anexa_termino(new termino(1/0.186000,228,1));
ec[63].anexa_termino(new termino(-1/0.186000,232,1));

ec[64].anexa_termino(new termino(-0.017900/0.050500,64,1));
ec[64].anexa_termino(new termino(1/0.050500,231,1));
ec[64].anexa_termino(new termino(-1/0.050500,232,1));

ec[65].anexa_termino(new termino(-0.026700/0.075200,65,1));
ec[65].anexa_termino(new termino(1/0.075200,232,1));
ec[65].anexa_termino(new termino(-1/0.075200,233,1));

ec[66].anexa_termino(new termino(-0.048600/0.137000,66,1));
ec[66].anexa_termino(new termino(1/0.137000,232,1));
ec[66].anexa_termino(new termino(-1/0.137000,234,1));

ec[67].anexa_termino(new termino(-0.020300/0.058800,67,1));
ec[67].anexa_termino(new termino(1/0.058800,234,1));
ec[67].anexa_termino(new termino(-1/0.058800,235,1));
```

```
ec[68].anexa_termino(new termino(-0.040500/0.163500,68,1));
ec[68].anexa_termino(new termino(1/0.163500,235,1));
ec[68].anexa_termino(new termino(-1/0.163500,236,1));

ec[69].anexa_termino(new termino(-0.026300/0.122000,69,1));
ec[69].anexa_termino(new termino(1/0.122000,236,1));
ec[69].anexa_termino(new termino(-1/0.122000,237,1));

ec[70].anexa_termino(new termino(-0.073000/0.289000,70,1));
ec[70].anexa_termino(new termino(1/0.289000,232,1));
ec[70].anexa_termino(new termino(-1/0.289000,237,1));

ec[71].anexa_termino(new termino(-0.086900/0.291000,71,1));
ec[71].anexa_termino(new termino(1/0.291000,232,1));
ec[71].anexa_termino(new termino(-1/0.291000,237,1));

ec[72].anexa_termino(new termino(-0.016900/0.070700,72,1));
ec[72].anexa_termino(new termino(1/0.070700,237,1));
ec[72].anexa_termino(new termino(-1/0.070700,238,1));

ec[73].anexa_termino(new termino(-0.002750/0.009550,73,1));
ec[73].anexa_termino(new termino(1/0.009550,237,1));
ec[73].anexa_termino(new termino(-1/0.009550,239,1));

ec[74].anexa_termino(new termino(-0.004880/0.015100,74,1));
ec[74].anexa_termino(new termino(1/0.015100,238,1));
ec[74].anexa_termino(new termino(-1/0.015100,239,1));

ec[75].anexa_termino(new termino(-0.034300/0.096600,75,1));
ec[75].anexa_termino(new termino(1/0.096600,239,1));
ec[75].anexa_termino(new termino(-1/0.096600,240,1));

ec[76].anexa_termino(new termino(-0.047400/0.134000,76,1));
ec[76].anexa_termino(new termino(1/0.134000,233,1));
ec[76].anexa_termino(new termino(-1/0.134000,240,1));

ec[77].anexa_termino(new termino(-0.034300/0.096600,77,1));
ec[77].anexa_termino(new termino(1/0.096600,239,1));
ec[77].anexa_termino(new termino(-1/0.096600,241,1));

ec[78].anexa_termino(new termino(-0.025500/0.071900,78,1));
ec[78].anexa_termino(new termino(1/0.071900,234,1));
ec[78].anexa_termino(new termino(-1/0.071900,241,1));

ec[79].anexa_termino(new termino(-0.050300/0.229300,79,1));
ec[79].anexa_termino(new termino(1/0.229300,237,1));
ec[79].anexa_termino(new termino(-1/0.229300,242,1));

ec[80].anexa_termino(new termino(-0.082500/0.251000,80,1));
ec[80].anexa_termino(new termino(1/0.251000,239,1));
ec[80].anexa_termino(new termino(-1/0.251000,242,1));

ec[81].anexa_termino(new termino(-0.047390/0.215800,81,1));
ec[81].anexa_termino(new termino(1/0.215800,238,1));
ec[81].anexa_termino(new termino(-1/0.215800,242,1));

ec[82].anexa_termino(new termino(-0.031700/0.145000,82,1));
ec[82].anexa_termino(new termino(1/0.145000,242,1));
ec[82].anexa_termino(new termino(-1/0.145000,243,1));

ec[83].anexa_termino(new termino(-0.032800/0.150000,83,1));
ec[83].anexa_termino(new termino(1/0.150000,242,1));
ec[83].anexa_termino(new termino(-1/0.150000,244,1));

ec[84].anexa_termino(new termino(-0.002640/0.010000,84,1));
ec[84].anexa_termino(new termino(1/0.010000,243,1));
ec[84].anexa_termino(new termino(-1/0.010000,244,1));

ec[85].anexa_termino(new termino(-0.012300/0.056100,85,1));
ec[85].anexa_termino(new termino(1/0.056100,243,1));
```

```
ec[85].anexa_termino(new termino(-1/0.056100,245,1));

ec[86].anexa_termino(new termino(-0.008240/0.037600,86,1));
ec[86].anexa_termino(new termino(1/0.037600,244,1));
ec[86].anexa_termino(new termino(-1/0.037600,245,1));

ec[87].anexa_termino(new termino(-0.001720/0.020000,87,1));
ec[87].anexa_termino(new termino(1/0.020000,246,1));
ec[87].anexa_termino(new termino(-1/0.020000,247,1));

ec[88].anexa_termino(new termino(-0.009010/0.098600,88,1));
ec[88].anexa_termino(new termino(1/0.098600,221,1));
ec[88].anexa_termino(new termino(-1/0.098600,248,1));

ec[89].anexa_termino(new termino(-0.002690/0.030200,89,1));
ec[89].anexa_termino(new termino(1/0.030200,247,1));
ec[89].anexa_termino(new termino(-1/0.030200,248,1));

ec[90].anexa_termino(new termino(-0.018000/0.091900,90,1));
ec[90].anexa_termino(new termino(1/0.091900,232,1));
ec[90].anexa_termino(new termino(-1/0.091900,249,1));

ec[91].anexa_termino(new termino(-0.018000/0.091900,91,1));
ec[91].anexa_termino(new termino(1/0.091900,232,1));
ec[91].anexa_termino(new termino(-1/0.091900,249,1));

ec[92].anexa_termino(new termino(-0.048200/0.218000,92,1));
ec[92].anexa_termino(new termino(1/0.218000,245,1));
ec[92].anexa_termino(new termino(-1/0.218000,249,1));

ec[93].anexa_termino(new termino(-0.025800/0.117000,93,1));
ec[93].anexa_termino(new termino(1/0.117000,245,1));
ec[93].anexa_termino(new termino(-1/0.117000,250,1));

ec[94].anexa_termino(new termino(-0.022400/0.101500,94,1));
ec[94].anexa_termino(new termino(1/0.101500,249,1));
ec[94].anexa_termino(new termino(-1/0.101500,250,1));

ec[95].anexa_termino(new termino(-0.001380/0.016000,95,1));
ec[95].anexa_termino(new termino(1/0.016000,248,1));
ec[95].anexa_termino(new termino(-1/0.016000,251,1));

ec[96].anexa_termino(new termino(-0.084400/0.277800,96,1));
ec[96].anexa_termino(new termino(1/0.277800,230,1));
ec[96].anexa_termino(new termino(-1/0.277800,252,1));

ec[97].anexa_termino(new termino(-0.098500/0.324000,97,1));
ec[97].anexa_termino(new termino(1/0.324000,232,1));
ec[97].anexa_termino(new termino(-1/0.324000,252,1));

ec[98].anexa_termino(new termino(-0.030000/0.127000,98,1));
ec[98].anexa_termino(new termino(1/0.127000,252,1));
ec[98].anexa_termino(new termino(-1/0.127000,253,1));

ec[99].anexa_termino(new termino(-0.002210/0.411500,99,1));
ec[99].anexa_termino(new termino(1/0.411500,207,1));
ec[99].anexa_termino(new termino(-1/0.411500,253,1));

ec[100].anexa_termino(new termino(-0.008820/0.035500,100,1));
ec[100].anexa_termino(new termino(1/0.035500,253,1));
ec[100].anexa_termino(new termino(-1/0.035500,254,1));

ec[101].anexa_termino(new termino(-0.048800/0.196000,101,1));
ec[101].anexa_termino(new termino(1/0.196000,207,1));
ec[101].anexa_termino(new termino(-1/0.196000,255,1));

ec[102].anexa_termino(new termino(-0.044600/0.180000,102,1));
ec[102].anexa_termino(new termino(1/0.180000,254,1));
ec[102].anexa_termino(new termino(-1/0.180000,255,1));

ec[103].anexa_termino(new termino(-0.008660/0.045400,103,1));
```

```
ec[103].anexa_termino(new termino(1/0.045400,254,1));
ec[103].anexa_termino(new termino(-1/0.045400,256,1));

ec[104].anexa_termino(new termino(-0.040100/0.132300,104,1));
ec[104].anexa_termino(new termino(1/0.132300,253,1));
ec[104].anexa_termino(new termino(-1/0.132300,257,1));

ec[105].anexa_termino(new termino(-0.042800/0.141000,105,1));
ec[105].anexa_termino(new termino(1/0.141000,253,1));
ec[105].anexa_termino(new termino(-1/0.141000,258,1));

ec[106].anexa_termino(new termino(-0.040500/0.122000,106,1));
ec[106].anexa_termino(new termino(1/0.122000,252,1));
ec[106].anexa_termino(new termino(-1/0.122000,258,1));

ec[107].anexa_termino(new termino(-0.012300/0.040600,107,1));
ec[107].anexa_termino(new termino(1/0.040600,257,1));
ec[107].anexa_termino(new termino(-1/0.040600,258,1));

ec[108].anexa_termino(new termino(-0.044400/0.148000,108,1));
ec[108].anexa_termino(new termino(1/0.148000,259,1));
ec[108].anexa_termino(new termino(-1/0.148000,260,1));

ec[109].anexa_termino(new termino(-0.030900/0.101000,109,1));
ec[109].anexa_termino(new termino(1/0.101000,252,1));
ec[109].anexa_termino(new termino(-1/0.101000,260,1));

ec[110].anexa_termino(new termino(-0.060100/0.199900,110,1));
ec[110].anexa_termino(new termino(1/0.199900,258,1));
ec[110].anexa_termino(new termino(-1/0.199900,260,1));

ec[111].anexa_termino(new termino(-0.003760/0.012400,111,1));
ec[111].anexa_termino(new termino(1/0.012400,260,1));
ec[111].anexa_termino(new termino(-1/0.012400,261,1));

ec[112].anexa_termino(new termino(-0.005460/0.024400,112,1));
ec[112].anexa_termino(new termino(1/0.024400,261,1));
ec[112].anexa_termino(new termino(-1/0.024400,262,1));

ec[113].anexa_termino(new termino(-0.017000/0.048500,113,1));
ec[113].anexa_termino(new termino(1/0.048500,260,1));
ec[113].anexa_termino(new termino(-1/0.048500,263,1));

ec[114].anexa_termino(new termino(-0.015600/0.070400,114,1));
ec[114].anexa_termino(new termino(1/0.070400,262,1));
ec[114].anexa_termino(new termino(-1/0.070400,263,1));

ec[115].anexa_termino(new termino(-0.001750/0.020200,115,1));
ec[115].anexa_termino(new termino(1/0.020200,251,1));
ec[115].anexa_termino(new termino(-1/0.020200,264,1));

ec[116].anexa_termino(new termino(-0.029800/0.085300,116,1));
ec[116].anexa_termino(new termino(1/0.085300,260,1));
ec[116].anexa_termino(new termino(-1/0.085300,265,1));

ec[117].anexa_termino(new termino(-0.011200/0.036650,117,1));
ec[117].anexa_termino(new termino(1/0.036650,265,1));
ec[117].anexa_termino(new termino(-1/0.036650,266,1));

ec[118].anexa_termino(new termino(-0.062500/0.132000,118,1));
ec[118].anexa_termino(new termino(1/0.132000,266,1));
ec[118].anexa_termino(new termino(-1/0.132000,267,1));

ec[119].anexa_termino(new termino(-0.043000/0.148000,119,1));
ec[119].anexa_termino(new termino(1/0.148000,266,1));
ec[119].anexa_termino(new termino(-1/0.148000,268,1));

ec[120].anexa_termino(new termino(-0.030200/0.064100,120,1));
ec[120].anexa_termino(new termino(1/0.064100,267,1));
ec[120].anexa_termino(new termino(-1/0.064100,268,1));
```

```
ec[121].anexa_termino(new termino(-0.035000/0.123000,121,1));
ec[121].anexa_termino(new termino(1/0.123000,268,1));
ec[121].anexa_termino(new termino(-1/0.123000,269,1));

ec[122].anexa_termino(new termino(-0.028280/0.207400,122,1));
ec[122].anexa_termino(new termino(1/0.207400,269,1));
ec[122].anexa_termino(new termino(-1/0.207400,270,1));

ec[123].anexa_termino(new termino(-0.020000/0.102000,123,1));
ec[123].anexa_termino(new termino(1/0.102000,268,1));
ec[123].anexa_termino(new termino(-1/0.102000,271,1));

ec[124].anexa_termino(new termino(-0.023900/0.173000,124,1));
ec[124].anexa_termino(new termino(1/0.173000,268,1));
ec[124].anexa_termino(new termino(-1/0.173000,272,1));

ec[125].anexa_termino(new termino(-0.013900/0.071200,125,1));
ec[125].anexa_termino(new termino(1/0.071200,271,1));
ec[125].anexa_termino(new termino(-1/0.071200,272,1));

ec[126].anexa_termino(new termino(-0.051800/0.188000,126,1));
ec[126].anexa_termino(new termino(1/0.188000,272,1));
ec[126].anexa_termino(new termino(-1/0.188000,273,1));

ec[127].anexa_termino(new termino(-0.023800/0.099700,127,1));
ec[127].anexa_termino(new termino(1/0.099700,272,1));
ec[127].anexa_termino(new termino(-1/0.099700,273,1));

ec[128].anexa_termino(new termino(-0.025400/0.083600,128,1));
ec[128].anexa_termino(new termino(1/0.083600,273,1));
ec[128].anexa_termino(new termino(-1/0.083600,274,1));

ec[129].anexa_termino(new termino(-0.009900/0.050500,129,1));
ec[129].anexa_termino(new termino(1/0.050500,272,1));
ec[129].anexa_termino(new termino(-1/0.050500,275,1));

ec[130].anexa_termino(new termino(-0.039300/0.158100,130,1));
ec[130].anexa_termino(new termino(1/0.158100,272,1));
ec[130].anexa_termino(new termino(-1/0.158100,275,1));

ec[131].anexa_termino(new termino(-0.038700/0.127200,131,1));
ec[131].anexa_termino(new termino(1/0.127200,274,1));
ec[131].anexa_termino(new termino(-1/0.127200,275,1));

ec[132].anexa_termino(new termino(-0.025800/0.084800,132,1));
ec[132].anexa_termino(new termino(1/0.084800,275,1));
ec[132].anexa_termino(new termino(-1/0.084800,276,1));

ec[133].anexa_termino(new termino(-0.048100/0.158000,133,1));
ec[133].anexa_termino(new termino(1/0.158000,275,1));
ec[133].anexa_termino(new termino(-1/0.158000,277,1));

ec[134].anexa_termino(new termino(-0.022300/0.073200,134,1));
ec[134].anexa_termino(new termino(1/0.073200,276,1));
ec[134].anexa_termino(new termino(-1/0.073200,277,1));

ec[135].anexa_termino(new termino(-0.013200/0.043400,135,1));
ec[135].anexa_termino(new termino(1/0.043400,277,1));
ec[135].anexa_termino(new termino(-1/0.043400,278,1));

ec[136].anexa_termino(new termino(-0.035600/0.182000,136,1));
ec[136].anexa_termino(new termino(1/0.182000,263,1));
ec[136].anexa_termino(new termino(-1/0.182000,279,1));

ec[137].anexa_termino(new termino(-0.016200/0.053000,137,1));
ec[137].anexa_termino(new termino(1/0.053000,265,1));
ec[137].anexa_termino(new termino(-1/0.053000,279,1));

ec[138].anexa_termino(new termino(-0.026900/0.086900,138,1));
ec[138].anexa_termino(new termino(1/0.086900,277,1));
ec[138].anexa_termino(new termino(-1/0.086900,279,1));
```

```
ec[139].anexa_termino(new termino(-0.018300/0.093400,139,1));
ec[139].anexa_termino(new termino(1/0.093400,263,1));
ec[139].anexa_termino(new termino(-1/0.093400,280,1));

ec[140].anexa_termino(new termino(-0.023800/0.108000,140,1));
ec[140].anexa_termino(new termino(1/0.108000,263,1));
ec[140].anexa_termino(new termino(-1/0.108000,281,1));

ec[141].anexa_termino(new termino(-0.045400/0.206000,141,1));
ec[141].anexa_termino(new termino(1/0.206000,263,1));
ec[141].anexa_termino(new termino(-1/0.206000,282,1));

ec[142].anexa_termino(new termino(-0.064800/0.295000,142,1));
ec[142].anexa_termino(new termino(1/0.295000,275,1));
ec[142].anexa_termino(new termino(-1/0.295000,283,1));

ec[143].anexa_termino(new termino(-0.017800/0.058000,143,1));
ec[143].anexa_termino(new termino(1/0.058000,277,1));
ec[143].anexa_termino(new termino(-1/0.058000,283,1));

ec[144].anexa_termino(new termino(-0.017100/0.054700,144,1));
ec[144].anexa_termino(new termino(1/0.054700,278,1));
ec[144].anexa_termino(new termino(-1/0.054700,279,1));

ec[145].anexa_termino(new termino(-0.017300/0.088500,145,1));
ec[145].anexa_termino(new termino(1/0.088500,279,1));
ec[145].anexa_termino(new termino(-1/0.088500,280,1));

ec[146].anexa_termino(new termino(-0.039700/0.179000,146,1));
ec[146].anexa_termino(new termino(1/0.179000,281,1));
ec[146].anexa_termino(new termino(-1/0.179000,283,1));

ec[147].anexa_termino(new termino(-0.018000/0.081300,147,1));
ec[147].anexa_termino(new termino(1/0.081300,282,1));
ec[147].anexa_termino(new termino(-1/0.081300,283,1));

ec[148].anexa_termino(new termino(-0.027700/0.126200,148,1));
ec[148].anexa_termino(new termino(1/0.126200,283,1));
ec[148].anexa_termino(new termino(-1/0.126200,284,1));

ec[149].anexa_termino(new termino(-0.012300/0.055900,149,1));
ec[149].anexa_termino(new termino(1/0.055900,275,1));
ec[149].anexa_termino(new termino(-1/0.055900,285,1));

ec[150].anexa_termino(new termino(-0.024600/0.112000,150,1));
ec[150].anexa_termino(new termino(1/0.112000,284,1));
ec[150].anexa_termino(new termino(-1/0.112000,285,1));

ec[151].anexa_termino(new termino(-0.016000/0.052500,151,1));
ec[151].anexa_termino(new termino(1/0.052500,283,1));
ec[151].anexa_termino(new termino(-1/0.052500,286,1));

ec[152].anexa_termino(new termino(-0.045100/0.204000,152,1));
ec[152].anexa_termino(new termino(1/0.204000,283,1));
ec[152].anexa_termino(new termino(-1/0.204000,287,1));

ec[153].anexa_termino(new termino(-0.046600/0.158400,153,1));
ec[153].anexa_termino(new termino(1/0.158400,286,1));
ec[153].anexa_termino(new termino(-1/0.158400,287,1));

ec[154].anexa_termino(new termino(-0.053500/0.162500,154,1));
ec[154].anexa_termino(new termino(1/0.162500,286,1));
ec[154].anexa_termino(new termino(-1/0.162500,288,1));

ec[155].anexa_termino(new termino(-0.060500/0.229000,155,1));
ec[155].anexa_termino(new termino(1/0.229000,283,1));
ec[155].anexa_termino(new termino(-1/0.229000,289,1));

ec[156].anexa_termino(new termino(-0.009940/0.037800,156,1));
ec[156].anexa_termino(new termino(1/0.037800,287,1));
```

```
ec[156].anexa_termino(new termino(-1/0.037800,288,1));

ec[157].anexa_termino(new termino(-0.014000/0.054700,157,1));
ec[157].anexa_termino(new termino(1/0.054700,288,1));
ec[157].anexa_termino(new termino(-1/0.054700,289,1));

ec[158].anexa_termino(new termino(-0.053000/0.183000,158,1));
ec[158].anexa_termino(new termino(1/0.183000,288,1));
ec[158].anexa_termino(new termino(-1/0.183000,290,1));

ec[159].anexa_termino(new termino(-0.026100/0.070300,159,1));
ec[159].anexa_termino(new termino(1/0.070300,288,1));
ec[159].anexa_termino(new termino(-1/0.070300,291,1));

ec[160].anexa_termino(new termino(-0.053000/0.183000,160,1));
ec[160].anexa_termino(new termino(1/0.183000,289,1));
ec[160].anexa_termino(new termino(-1/0.183000,290,1));

ec[161].anexa_termino(new termino(-0.010500/0.028800,161,1));
ec[161].anexa_termino(new termino(1/0.028800,291,1));
ec[161].anexa_termino(new termino(-1/0.028800,292,1));

ec[162].anexa_termino(new termino(-0.039060/0.181300,162,1));
ec[162].anexa_termino(new termino(1/0.181300,286,1));
ec[162].anexa_termino(new termino(-1/0.181300,293,1));

ec[163].anexa_termino(new termino(-0.027800/0.076200,163,1));
ec[163].anexa_termino(new termino(1/0.076200,292,1));
ec[163].anexa_termino(new termino(-1/0.076200,293,1));

ec[164].anexa_termino(new termino(-0.022000/0.075500,164,1));
ec[164].anexa_termino(new termino(1/0.075500,293,1));
ec[164].anexa_termino(new termino(-1/0.075500,294,1));

ec[165].anexa_termino(new termino(-0.024700/0.064000,165,1));
ec[165].anexa_termino(new termino(1/0.064000,293,1));
ec[165].anexa_termino(new termino(-1/0.064000,295,1));

ec[166].anexa_termino(new termino(-0.009130/0.030100,166,1));
ec[166].anexa_termino(new termino(1/0.030100,200,1));
ec[166].anexa_termino(new termino(-1/0.030100,296,1));

ec[167].anexa_termino(new termino(-0.061500/0.203000,167,1));
ec[167].anexa_termino(new termino(1/0.203000,215,1));
ec[167].anexa_termino(new termino(-1/0.203000,296,1));

ec[168].anexa_termino(new termino(-0.013500/0.061200,168,1));
ec[168].anexa_termino(new termino(1/0.061200,215,1));
ec[168].anexa_termino(new termino(-1/0.061200,297,1));

ec[169].anexa_termino(new termino(-0.016400/0.074100,169,1));
ec[169].anexa_termino(new termino(1/0.074100,210,1));
ec[169].anexa_termino(new termino(-1/0.074100,298,1));

ec[170].anexa_termino(new termino(-0.002300/0.010400,170,1));
ec[170].anexa_termino(new termino(1/0.010400,297,1));
ec[170].anexa_termino(new termino(-1/0.010400,298,1));

ec[171].anexa_termino(new termino(-0.000340/0.004050,171,1));
ec[171].anexa_termino(new termino(1/0.004050,251,1));
ec[171].anexa_termino(new termino(-1/0.004050,299,1));

ec[172].anexa_termino(new termino(-0.032900/0.140000,172,1));
ec[172].anexa_termino(new termino(1/0.140000,195,1));
ec[172].anexa_termino(new termino(-1/0.140000,300,1));

ec[173].anexa_termino(new termino(-0.014500/0.048100,173,1));
ec[173].anexa_termino(new termino(1/0.048100,258,1));
ec[173].anexa_termino(new termino(-1/0.048100,301,1));

ec[174].anexa_termino(new termino(-0.016400/0.054400,174,1));
```

```
ec[174].anexa_termino(new termino(1/0.054400,259,1));
ec[174].anexa_termino(new termino(-1/0.054400,301,1));

ec[175].anexa_termino(new termino(-0.000010/0.026700,175,1));
ec[175].anexa_termino(new termino(1/0.026700,191,1));
ec[175].anexa_termino(new termino(-1/0.026700,188,1));

ec[176].anexa_termino(new termino(-0.000010/0.038200,176,1));
ec[176].anexa_termino(new termino(1/0.038200,209,1));
ec[176].anexa_termino(new termino(-1/0.038200,208,1));

ec[177].anexa_termino(new termino(-0.000010/0.038800,177,1));
ec[177].anexa_termino(new termino(1/0.038800,213,1));
ec[177].anexa_termino(new termino(-1/0.038800,200,1));

ec[178].anexa_termino(new termino(-0.000010/0.037500,178,1));
ec[178].anexa_termino(new termino(1/0.037500,221,1));
ec[178].anexa_termino(new termino(-1/0.037500,220,1));

ec[179].anexa_termino(new termino(-0.000010/0.038600,179,1));
ec[179].anexa_termino(new termino(1/0.038600,246,1));
ec[179].anexa_termino(new termino(-1/0.038600,242,1));

ec[180].anexa_termino(new termino(-0.000010/0.026800,180,1));
ec[180].anexa_termino(new termino(1/0.026800,247,1));
ec[180].anexa_termino(new termino(-1/0.026800,244,1));

ec[181].anexa_termino(new termino(-0.000010/0.037000,181,1));
ec[181].anexa_termino(new termino(1/0.037000,248,1));
ec[181].anexa_termino(new termino(-1/0.037000,249,1));

ec[182].anexa_termino(new termino(-0.000010/0.037000,182,1));
ec[182].anexa_termino(new termino(1/0.037000,251,1));
ec[182].anexa_termino(new termino(-1/0.037000,252,1));

ec[183].anexa_termino(new termino(-0.000010/0.037000,183,1));
ec[183].anexa_termino(new termino(1/0.037000,264,1));
ec[183].anexa_termino(new termino(-1/0.037000,263,1));

ec[184].anexa_termino(new termino(1/0.1,302,1));
ec[184].anexa_termino(new termino(-1/0.1,0,1));
ec[184].anexa_termino(new termino(-1/0.1,1,1));

ec[185].anexa_termino(new termino(1/0.1,0,1));
ec[185].anexa_termino(new termino(-1/0.1,11,1));

ec[186].anexa_termino(new termino(1/0.1,1,1));
ec[186].anexa_termino(new termino(-1/0.1,3,1));
ec[186].anexa_termino(new termino(-1/0.1,12,1));

ec[187].anexa_termino(new termino(1/0.1,303,1));
ec[187].anexa_termino(new termino(-1/0.1,2,1));
ec[187].anexa_termino(new termino(-1/0.1,8,1));

ec[188].anexa_termino(new termino(1/0.1,2,1));
ec[188].anexa_termino(new termino(1/0.1,3,1));
ec[188].anexa_termino(new termino(-1/0.1,4,1));
ec[188].anexa_termino(new termino(-1/0.1,9,1));
ec[188].anexa_termino(new termino(1/0.1,175,1));  //correccion

ec[189].anexa_termino(new termino(1/0.1,304,1));
ec[189].anexa_termino(new termino(1/0.1,4,1));
ec[189].anexa_termino(new termino(-1/0.1,5,1));

ec[190].anexa_termino(new termino(1/0.1,5,1));
ec[190].anexa_termino(new termino(-1/0.1,13,1));

ec[191].anexa_termino(new termino(1/0.1,305,1));
ec[191].anexa_termino(new termino(-1/0.1,6,1));
ec[191].anexa_termino(new termino(-1/0.1,33,1));
ec[191].anexa_termino(new termino(-1/0.1,175,1));
```

```
ec[192].anexa_termino(new termino(1/0.1,6,1));
ec[192].anexa_termino(new termino(-1/0.1,7,1));

ec[193].anexa_termino(new termino(1/0.1,306,1));
ec[193].anexa_termino(new termino(1/0.1,7,1));

ec[194].anexa_termino(new termino(1/0.1,8,1));
ec[194].anexa_termino(new termino(1/0.1,9,1));
ec[194].anexa_termino(new termino(-1/0.1,10,1));
ec[194].anexa_termino(new termino(-1/0.1,14,1));

ec[195].anexa_termino(new termino(1/0.1,307,1));
ec[195].anexa_termino(new termino(1/0.1,10,1));
ec[195].anexa_termino(new termino(1/0.1,11,1));
ec[195].anexa_termino(new termino(1/0.1,12,1));
ec[195].anexa_termino(new termino(1/0.1,13,1));
ec[195].anexa_termino(new termino(-1/0.1,15,1));
ec[195].anexa_termino(new termino(-1/0.1,18,1));
ec[195].anexa_termino(new termino(-1/0.1,172,1));

ec[196].anexa_termino(new termino(1/0.1,14,1));
ec[196].anexa_termino(new termino(-1/0.1,16,1));

ec[197].anexa_termino(new termino(1/0.1,15,1));
ec[197].anexa_termino(new termino(-1/0.1,17,1));

ec[198].anexa_termino(new termino(1/0.1,308,1));
ec[198].anexa_termino(new termino(-1/0.1,364,5));
ec[198].anexa_termino(new termino(1/0.1,16,1));
ec[198].anexa_termino(new termino(1/0.1,17,1));
ec[198].anexa_termino(new termino(-1/0.1,19,1));
ec[198].anexa_termino(new termino(-1/0.1,24,1));
ec[198].anexa_termino(new termino(-1/0.1,40,1));

ec[199].anexa_termino(new termino(1/0.1,18,1));
ec[199].anexa_termino(new termino(-1/0.1,20,1));

ec[200].anexa_termino(new termino(1/0.1,19,1));
ec[200].anexa_termino(new termino(1/0.1,20,1));
ec[200].anexa_termino(new termino(-1/0.1,21,1));
ec[200].anexa_termino(new termino(-1/0.1,35,1));
ec[200].anexa_termino(new termino(-1/0.1,166,1));
ec[200].anexa_termino(new termino(1/0.1,177,1));

ec[201].anexa_termino(new termino(1/0.1,309,1));
ec[201].anexa_termino(new termino(1/0.1,21,1));
ec[201].anexa_termino(new termino(-1/0.1,22,1));

ec[202].anexa_termino(new termino(1/0.1,310,1));
ec[202].anexa_termino(new termino(1/0.1,22,1));
ec[202].anexa_termino(new termino(-1/0.1,23,1));
ec[202].anexa_termino(new termino(1/0.1,24,1));
ec[202].anexa_termino(new termino(-1/0.1,41,1));

ec[203].anexa_termino(new termino(1/0.1,23,1));
ec[203].anexa_termino(new termino(-1/0.1,25,1));

ec[204].anexa_termino(new termino(1/0.1,25,1));
ec[204].anexa_termino(new termino(-1/0.1,26,1));

ec[205].anexa_termino(new termino(-1/0.1,363,5));
ec[205].anexa_termino(new termino(1/0.1,26,1));
ec[205].anexa_termino(new termino(-1/0.1,27,1));

ec[206].anexa_termino(new termino(1/0.1,27,1));
ec[206].anexa_termino(new termino(-1/0.1,28,1));
ec[206].anexa_termino(new termino(-1/0.1,29,1));
ec[206].anexa_termino(new termino(-1/0.1,37,1));

ec[207].anexa_termino(new termino(1/0.1,311,1));
```

```
ec[207].anexa_termino(new termino(1/0.1,28,1));
ec[207].anexa_termino(new termino(-1/0.1,99,1));
ec[207].anexa_termino(new termino(-1/0.1,101,1));

ec[208].anexa_termino(new termino(1/0.1,312,1));
ec[208].anexa_termino(new termino(1/0.1,29,1));
ec[208].anexa_termino(new termino(-1/0.1,30,1));
ec[208].anexa_termino(new termino(1/0.1,176,1));


ec[209].anexa_termino(new termino(1/0.1,313,1));
ec[209].anexa_termino(new termino(-1/0.1,34,1));
ec[209].anexa_termino(new termino(-1/0.1,176,1));

ec[210].anexa_termino(new termino(1/0.1,314,1));
ec[210].anexa_termino(new termino(1/0.1,30,1));
ec[210].anexa_termino(new termino(-1/0.1,31,1));
ec[210].anexa_termino(new termino(-1/0.1,39,1));
ec[210].anexa_termino(new termino(-1/0.1,169,1));

ec[211].anexa_termino(new termino(1/0.1,31,1));
ec[211].anexa_termino(new termino(-1/0.1,32,1));

ec[212].anexa_termino(new termino(1/0.1,32,1));
ec[212].anexa_termino(new termino(-1/0.1,36,1));

ec[213].anexa_termino(new termino(1/0.1,33,1));
ec[213].anexa_termino(new termino(1/0.1,34,1));
ec[213].anexa_termino(new termino(-1/0.1,49,1));
ec[213].anexa_termino(new termino(-1/0.1,177,1));

ec[214].anexa_termino(new termino(1/0.1,315,1));
ec[214].anexa_termino(new termino(1/0.1,35,1));
ec[214].anexa_termino(new termino(1/0.1,36,1));
ec[214].anexa_termino(new termino(-1/0.1,38,1));

ec[215].anexa_termino(new termino(1/0.1,316,1));
ec[215].anexa_termino(new termino(-1/0.1,361,5));
ec[215].anexa_termino(new termino(1/0.1,37,1));
ec[215].anexa_termino(new termino(1/0.1,38,1));
ec[215].anexa_termino(new termino(1/0.1,39,1));
ec[215].anexa_termino(new termino(-1/0.1,167,1));
ec[215].anexa_termino(new termino(-1/0.1,168,1));

ec[216].anexa_termino(new termino(1/0.1,40,1));
ec[216].anexa_termino(new termino(-1/0.1,44,1));

ec[217].anexa_termino(new termino(1/0.1,317,1));
ec[217].anexa_termino(new termino(1/0.1,41,1));
ec[217].anexa_termino(new termino(-1/0.1,45,1));
ec[217].anexa_termino(new termino(-1/0.1,46,1));
ec[217].anexa_termino(new termino(-1/0.1,55,1));

ec[218].anexa_termino(new termino(-1/0.1,42,1));
ec[218].anexa_termino(new termino(-1/0.1,43,1));

ec[219].anexa_termino(new termino(1/0.1,318,1));
ec[219].anexa_termino(new termino(1/0.1,42,1));
ec[219].anexa_termino(new termino(1/0.1,45,1));

ec[220].anexa_termino(new termino(1/0.1,43,1));
ec[220].anexa_termino(new termino(1/0.1,44,1));
ec[220].anexa_termino(new termino(1/0.1,46,1));
ec[220].anexa_termino(new termino(-1/0.1,47,1));
ec[220].anexa_termino(new termino(-1/0.1,48,1));
ec[220].anexa_termino(new termino(1/0.1,178,1));

ec[221].anexa_termino(new termino(1/0.1,49,1));
ec[221].anexa_termino(new termino(-1/0.1,88,1));
ec[221].anexa_termino(new termino(-1/0.1,178,1));
```

```
ec[222].anexa_termino(new termino(1/0.1,47,1));
ec[222].anexa_termino(new termino(-1/0.1,50,1));

ec[223].anexa_termino(new termino(1/0.1,319,1));
ec[223].anexa_termino(new termino(1/0.1,48,1));
ec[223].anexa_termino(new termino(1/0.1,50,1));
ec[223].anexa_termino(new termino(-1/0.1,51,1));
ec[223].anexa_termino(new termino(-1/0.1,52,1));

ec[224].anexa_termino(new termino(1/0.1,51,1));
ec[224].anexa_termino(new termino(-1/0.1,53,1));

ec[225].anexa_termino(new termino(1/0.1,320,1));
ec[225].anexa_termino(new termino(1/0.1,52,1));
ec[225].anexa_termino(new termino(1/0.1,53,1));
ec[225].anexa_termino(new termino(-1/0.1,61,1));
ec[225].anexa_termino(new termino(-1/0.1,62,1));

ec[226].anexa_termino(new termino(-1/0.1,54,1));
ec[226].anexa_termino(new termino(1/0.1,55,1));

ec[227].anexa_termino(new termino(-1/0.1,362,5));
ec[227].anexa_termino(new termino(1/0.1,54,1));
ec[227].anexa_termino(new termino(-1/0.1,56,1));

ec[228].anexa_termino(new termino(-1/0.1,360,5));
ec[228].anexa_termino(new termino(1/0.1,56,1));
ec[228].anexa_termino(new termino(-1/0.1,57,1));
ec[228].anexa_termino(new termino(-1/0.1,63,1));

ec[229].anexa_termino(new termino(1/0.1,321,1));
ec[229].anexa_termino(new termino(1/0.1,57,1));
ec[229].anexa_termino(new termino(-1/0.1,58,1));
ec[229].anexa_termino(new termino(-1/0.1,59,1));

ec[230].anexa_termino(new termino(1/0.1,58,1));
ec[230].anexa_termino(new termino(-1/0.1,60,1));
ec[230].anexa_termino(new termino(-1/0.1,96,1));

ec[231].anexa_termino(new termino(1/0.1,59,1));
ec[231].anexa_termino(new termino(-1/0.1,64,1));

ec[232].anexa_termino(new termino(1/0.1,322,1));
ec[232].anexa_termino(new termino(1/0.1,60,1));
ec[232].anexa_termino(new termino(1/0.1,61,1));
ec[232].anexa_termino(new termino(1/0.1,62,1));
ec[232].anexa_termino(new termino(1/0.1,63,1));
ec[232].anexa_termino(new termino(1/0.1,64,1));
ec[232].anexa_termino(new termino(-1/0.1,65,1));
ec[232].anexa_termino(new termino(-1/0.1,66,1));
ec[232].anexa_termino(new termino(-1/0.1,70,1));
ec[232].anexa_termino(new termino(-1/0.1,71,1));
ec[232].anexa_termino(new termino(-1/0.1,90,1));
ec[232].anexa_termino(new termino(-1/0.1,91,1));
ec[232].anexa_termino(new termino(-1/0.1,97,1));

ec[233].anexa_termino(new termino(1/0.1,65,1));
ec[233].anexa_termino(new termino(-1/0.1,76,1));

ec[234].anexa_termino(new termino(1/0.1,66,1));
ec[234].anexa_termino(new termino(-1/0.1,67,1));
ec[234].anexa_termino(new termino(-1/0.1,78,1));

ec[235].anexa_termino(new termino(1/0.1,67,1));
ec[235].anexa_termino(new termino(-1/0.1,68,1));

ec[236].anexa_termino(new termino(1/0.1,68,1));
ec[236].anexa_termino(new termino(-1/0.1,69,1));

ec[237].anexa_termino(new termino(1/0.1,323,1));
ec[237].anexa_termino(new termino(1/0.1,69,1));
```

```
ec[237].anexa_termino(new termino(1/0.1,70,1));
ec[237].anexa_termino(new termino(1/0.1,71,1));
ec[237].anexa_termino(new termino(-1/0.1,72,1));
ec[237].anexa_termino(new termino(-1/0.1,73,1));
ec[237].anexa_termino(new termino(-1/0.1,79,1));

ec[238].anexa_termino(new termino(1/0.1,324,1));
ec[238].anexa_termino(new termino(-1/0.1,365,5));
ec[238].anexa_termino(new termino(1/0.1,72,1));
ec[238].anexa_termino(new termino(-1/0.1,74,1));
ec[238].anexa_termino(new termino(-1/0.1,81,1));

ec[239].anexa_termino(new termino(1/0.1,325,1));
ec[239].anexa_termino(new termino(1/0.1,73,1));
ec[239].anexa_termino(new termino(1/0.1,74,1));
ec[239].anexa_termino(new termino(-1/0.1,75,1));
ec[239].anexa_termino(new termino(-1/0.1,77,1));
ec[239].anexa_termino(new termino(-1/0.1,80,1));

ec[240].anexa_termino(new termino(1/0.1,75,1));
ec[240].anexa_termino(new termino(1/0.1,76,1));

ec[241].anexa_termino(new termino(1/0.1,77,1));
ec[241].anexa_termino(new termino(1/0.1,78,1));

ec[242].anexa_termino(new termino(1/0.1,326,1));
ec[242].anexa_termino(new termino(1/0.1,79,1));
ec[242].anexa_termino(new termino(1/0.1,80,1));
ec[242].anexa_termino(new termino(1/0.1,81,1));
ec[242].anexa_termino(new termino(-1/0.1,82,1));
ec[242].anexa_termino(new termino(-1/0.1,83,1));
ec[242].anexa_termino(new termino(1/0.1,179,1));

ec[243].anexa_termino(new termino(1/0.1,82,1));
ec[243].anexa_termino(new termino(-1/0.1,84,1));
ec[243].anexa_termino(new termino(-1/0.1,85,1));

ec[244].anexa_termino(new termino(1/0.1,327,1));
ec[244].anexa_termino(new termino(1/0.1,83,1));
ec[244].anexa_termino(new termino(1/0.1,84,1));
ec[244].anexa_termino(new termino(-1/0.1,86,1));
ec[244].anexa_termino(new termino(1/0.1,180,1));

ec[245].anexa_termino(new termino(1/0.1,328,1));
ec[245].anexa_termino(new termino(1/0.1,85,1));
ec[245].anexa_termino(new termino(1/0.1,86,1));
ec[245].anexa_termino(new termino(-1/0.1,92,1));
ec[245].anexa_termino(new termino(-1/0.1,93,1));

ec[246].anexa_termino(new termino(-1/0.1,87,1));
ec[246].anexa_termino(new termino(-1/0.1,179,1));

ec[247].anexa_termino(new termino(1/0.1,87,1));
ec[247].anexa_termino(new termino(-1/0.1,89,1));
ec[247].anexa_termino(new termino(-1/0.1,180,1));

ec[248].anexa_termino(new termino(1/0.1,329,1));
ec[248].anexa_termino(new termino(1/0.1,88,1));
ec[248].anexa_termino(new termino(1/0.1,89,1));
ec[248].anexa_termino(new termino(-1/0.1,95,1));
ec[248].anexa_termino(new termino(-1/0.1,181,1));

ec[249].anexa_termino(new termino(1/0.1,330,1));
ec[249].anexa_termino(new termino(1/0.1,90,1));
ec[249].anexa_termino(new termino(1/0.1,91,1));
ec[249].anexa_termino(new termino(1/0.1,92,1));
ec[249].anexa_termino(new termino(-1/0.1,94,1));
ec[249].anexa_termino(new termino(1/0.1,181,1));

ec[250].anexa_termino(new termino(1/0.1,93,1));
ec[250].anexa_termino(new termino(1/0.1,94,1));
```

```
ec[251].anexa_termino(new termino(1/0.1,95,1));
ec[251].anexa_termino(new termino(-1/0.1,115,1));
ec[251].anexa_termino(new termino(-1/0.1,171,1));
ec[251].anexa_termino(new termino(-1/0.1,182,1));

ec[252].anexa_termino(new termino(1/0.1,96,1));
ec[252].anexa_termino(new termino(1/0.1,97,1));
ec[252].anexa_termino(new termino(-1/0.1,98,1));
ec[252].anexa_termino(new termino(-1/0.1,106,1));
ec[252].anexa_termino(new termino(-1/0.1,109,1));
ec[252].anexa_termino(new termino(1/0.1,182,1));

ec[253].anexa_termino(new termino(1/0.1,331,1));
ec[253].anexa_termino(new termino(1/0.1,98,1));
ec[253].anexa_termino(new termino(1/0.1,99,1));
ec[253].anexa_termino(new termino(-1/0.1,100,1));
ec[253].anexa_termino(new termino(-1/0.1,104,1));
ec[253].anexa_termino(new termino(-1/0.1,105,1));

ec[254].anexa_termino(new termino(1/0.1,100,1));
ec[254].anexa_termino(new termino(-1/0.1,102,1));
ec[254].anexa_termino(new termino(-1/0.1,103,1));

ec[255].anexa_termino(new termino(1/0.1,332,1));
ec[255].anexa_termino(new termino(1/0.1,101,1));
ec[255].anexa_termino(new termino(1/0.1,102,1));

ec[256].anexa_termino(new termino(1/0.1,333,1));
ec[256].anexa_termino(new termino(-1/0.1,359,5));
ec[256].anexa_termino(new termino(1/0.1,103,1));

ec[257].anexa_termino(new termino(1/0.1,334,1));
ec[257].anexa_termino(new termino(1/0.1,104,1));
ec[257].anexa_termino(new termino(-1/0.1,107,1));

ec[258].anexa_termino(new termino(1/0.1,105,1));
ec[258].anexa_termino(new termino(1/0.1,106,1));
ec[258].anexa_termino(new termino(1/0.1,107,1));
ec[258].anexa_termino(new termino(-1/0.1,110,1));
ec[258].anexa_termino(new termino(-1/0.1,173,1));

ec[259].anexa_termino(new termino(1/0.1,335,1));
ec[259].anexa_termino(new termino(-1/0.1,108,1));
ec[259].anexa_termino(new termino(-1/0.1,174,1));

ec[260].anexa_termino(new termino(1/0.1,336,1));
ec[260].anexa_termino(new termino(1/0.1,108,1));
ec[260].anexa_termino(new termino(1/0.1,109,1));
ec[260].anexa_termino(new termino(1/0.1,110,1));
ec[260].anexa_termino(new termino(-1/0.1,111,1));
ec[260].anexa_termino(new termino(-1/0.1,113,1));
ec[260].anexa_termino(new termino(-1/0.1,116,1));

ec[261].anexa_termino(new termino(1/0.1,111,1));
ec[261].anexa_termino(new termino(-1/0.1,112,1));

ec[262].anexa_termino(new termino(1/0.1,112,1));
ec[262].anexa_termino(new termino(-1/0.1,114,1));

ec[263].anexa_termino(new termino(1/0.1,337,1));
ec[263].anexa_termino(new termino(1/0.1,113,1));
ec[263].anexa_termino(new termino(1/0.1,114,1));
ec[263].anexa_termino(new termino(-1/0.1,136,1));
ec[263].anexa_termino(new termino(-1/0.1,139,1));
ec[263].anexa_termino(new termino(-1/0.1,140,1));
ec[263].anexa_termino(new termino(-1/0.1,141,1));
ec[263].anexa_termino(new termino(1/0.1,183,1));

ec[264].anexa_termino(new termino(1/0.1,115,1));
ec[264].anexa_termino(new termino(-1/0.1,183,1));
```

```
ec[265].anexa_termino(new termino(1/0.1,116,1));
ec[265].anexa_termino(new termino(-1/0.1,117,1));
ec[265].anexa_termino(new termino(-1/0.1,137,1));

ec[266].anexa_termino(new termino(1/0.1,117,1));
ec[266].anexa_termino(new termino(-1/0.1,118,1));
ec[266].anexa_termino(new termino(-1/0.1,119,1));

ec[267].anexa_termino(new termino(1/0.1,118,1));
ec[267].anexa_termino(new termino(-1/0.1,120,1));

ec[268].anexa_termino(new termino(1/0.1,338,1));
ec[268].anexa_termino(new termino(1/0.1,119,1));
ec[268].anexa_termino(new termino(1/0.1,120,1));
ec[268].anexa_termino(new termino(-1/0.1,121,1));
ec[268].anexa_termino(new termino(-1/0.1,123,1));
ec[268].anexa_termino(new termino(-1/0.1,124,1));

ec[269].anexa_termino(new termino(1/0.1,121,1));
ec[269].anexa_termino(new termino(-1/0.1,122,1));

ec[270].anexa_termino(new termino(1/0.1,339,1));
ec[270].anexa_termino(new termino(1/0.1,122,1));

ec[271].anexa_termino(new termino(1/0.1,123,1));
ec[271].anexa_termino(new termino(-1/0.1,125,1));

ec[272].anexa_termino(new termino(1/0.1,340,1));
ec[272].anexa_termino(new termino(1/0.1,124,1));
ec[272].anexa_termino(new termino(1/0.1,125,1));
ec[272].anexa_termino(new termino(-1/0.1,126,1));
ec[272].anexa_termino(new termino(-1/0.1,127,1));
ec[272].anexa_termino(new termino(-1/0.1,129,1));
ec[272].anexa_termino(new termino(-1/0.1,130,1));

ec[273].anexa_termino(new termino(1/0.1,341,1));
ec[273].anexa_termino(new termino(1/0.1,126,1));
ec[273].anexa_termino(new termino(1/0.1,127,1));
ec[273].anexa_termino(new termino(-1/0.1,128,1));

ec[274].anexa_termino(new termino(1/0.1,342,1));
ec[274].anexa_termino(new termino(1/0.1,128,1));
ec[274].anexa_termino(new termino(-1/0.1,131,1));

ec[275].anexa_termino(new termino(1/0.1,343,1));
ec[275].anexa_termino(new termino(1/0.1,129,1));
ec[275].anexa_termino(new termino(1/0.1,130,1));
ec[275].anexa_termino(new termino(1/0.1,131,1));
ec[275].anexa_termino(new termino(-1/0.1,132,1));
ec[275].anexa_termino(new termino(-1/0.1,133,1));
ec[275].anexa_termino(new termino(-1/0.1,142,1));
ec[275].anexa_termino(new termino(-1/0.1,149,1));

ec[276].anexa_termino(new termino(1/0.1,132,1));
ec[276].anexa_termino(new termino(-1/0.1,134,1));

ec[277].anexa_termino(new termino(1/0.1,133,1));
ec[277].anexa_termino(new termino(1/0.1,134,1));
ec[277].anexa_termino(new termino(-1/0.1,135,1));
ec[277].anexa_termino(new termino(-1/0.1,138,1));
ec[277].anexa_termino(new termino(-1/0.1,143,1));

ec[278].anexa_termino(new termino(-1/0.1,358,5));
ec[278].anexa_termino(new termino(1/0.1,135,1));
ec[278].anexa_termino(new termino(-1/0.1,144,1));

ec[279].anexa_termino(new termino(1/0.1,136,1));
ec[279].anexa_termino(new termino(1/0.1,137,1));
ec[279].anexa_termino(new termino(1/0.1,138,1));
ec[279].anexa_termino(new termino(1/0.1,144,1));
```

```
ec[279].anexa_termino(new termino(-1/0.1,145,1));

ec[280].anexa_termino(new termino(1/0.1,139,1));
ec[280].anexa_termino(new termino(1/0.1,145,1));

ec[281].anexa_termino(new termino(1/0.1,140,1));
ec[281].anexa_termino(new termino(-1/0.1,146,1));

ec[282].anexa_termino(new termino(1/0.1,344,1));
ec[282].anexa_termino(new termino(1/0.1,141,1));
ec[282].anexa_termino(new termino(-1/0.1,147,1));

ec[283].anexa_termino(new termino(1/0.1,345,1));
ec[283].anexa_termino(new termino(-1/0.1,355,5));
ec[283].anexa_termino(new termino(1/0.1,142,1));
ec[283].anexa_termino(new termino(1/0.1,143,1));
ec[283].anexa_termino(new termino(1/0.1,146,1));
ec[283].anexa_termino(new termino(1/0.1,147,1));
ec[283].anexa_termino(new termino(-1/0.1,148,1));
ec[283].anexa_termino(new termino(-1/0.1,151,1));
ec[283].anexa_termino(new termino(-1/0.1,152,1));
ec[283].anexa_termino(new termino(-1/0.1,155,1));

ec[284].anexa_termino(new termino(1/0.1,148,1));
ec[284].anexa_termino(new termino(-1/0.1,150,1));

ec[285].anexa_termino(new termino(1/0.1,149,1));
ec[285].anexa_termino(new termino(1/0.1,150,1));

ec[286].anexa_termino(new termino(1/0.1,346,1));
ec[286].anexa_termino(new termino(1/0.1,151,1));
ec[286].anexa_termino(new termino(-1/0.1,153,1));
ec[286].anexa_termino(new termino(-1/0.1,154,1));
ec[286].anexa_termino(new termino(-1/0.1,162,1));

ec[287].anexa_termino(new termino(1/0.1,347,1));
ec[287].anexa_termino(new termino(1/0.1,152,1));
ec[287].anexa_termino(new termino(1/0.1,153,1));
ec[287].anexa_termino(new termino(-1/0.1,156,1));

ec[288].anexa_termino(new termino(1/0.1,348,1));
ec[288].anexa_termino(new termino(1/0.1,154,1));
ec[288].anexa_termino(new termino(1/0.1,156,1));
ec[288].anexa_termino(new termino(-1/0.1,157,1));
ec[288].anexa_termino(new termino(-1/0.1,158,1));
ec[288].anexa_termino(new termino(-1/0.1,159,1));

ec[289].anexa_termino(new termino(1/0.1,155,1));
ec[289].anexa_termino(new termino(1/0.1,157,1));
ec[289].anexa_termino(new termino(-1/0.1,160,1));

ec[290].anexa_termino(new termino(1/0.1,349,1));
ec[290].anexa_termino(new termino(1/0.1,158,1));
ec[290].anexa_termino(new termino(1/0.1,160,1));

ec[291].anexa_termino(new termino(1/0.1,159,1));
ec[291].anexa_termino(new termino(-1/0.1,161,1));

ec[292].anexa_termino(new termino(1/0.1,161,1));
ec[292].anexa_termino(new termino(-1/0.1,163,1));

ec[293].anexa_termino(new termino(1/0.1,350,1));
ec[293].anexa_termino(new termino(1/0.1,162,1));
ec[293].anexa_termino(new termino(1/0.1,163,1));
ec[293].anexa_termino(new termino(-1/0.1,164,1));
ec[293].anexa_termino(new termino(-1/0.1,165,1));

ec[294].anexa_termino(new termino(1/0.1,351,1));
ec[294].anexa_termino(new termino(-1/0.1,356,5));
ec[294].anexa_termino(new termino(1/0.1,164,1));
```

```
ec[295].anexa_termino(new termino(1/0.1,352,1));
ec[295].anexa_termino(new termino(1/0.1,165,1));

ec[296].anexa_termino(new termino(1/0.1,353,1));
ec[296].anexa_termino(new termino(1/0.1,166,1));
ec[296].anexa_termino(new termino(1/0.1,167,1));

ec[297].anexa_termino(new termino(1/0.1,168,1));
ec[297].anexa_termino(new termino(-1/0.1,170,1));

ec[298].anexa_termino(new termino(-1/0.1,357,5));
ec[298].anexa_termino(new termino(1/0.1,169,1));
ec[298].anexa_termino(new termino(1/0.1,170,1));

ec[299].anexa_termino(new termino(1/0.1,354,1));
ec[299].anexa_termino(new termino(1/0.1,171,1));

ec[300].anexa_termino(new termino(1/0.1,172,1));

ec[301].anexa_termino(new termino(1/0.1,173,1));
ec[301].anexa_termino(new termino(1/0.1,174,1));


//Se agregaron las siguientes ecuaciones


ec[355].anexa_termino(new termino(-0.1,355,5));
ec[355].anexa_termino(new termino(1,283,1));

ec[356].anexa_termino(new termino(-0.1,356,5));
ec[356].anexa_termino(new termino(1,294,1));

ec[357].anexa_termino(new termino(-0.1,357,5));
ec[357].anexa_termino(new termino(1,298,1));

ec[358].anexa_termino(new termino(-0.1,358,5));
ec[358].anexa_termino(new termino(1,278,1));

ec[359].anexa_termino(new termino(-0.1,359,5));
ec[359].anexa_termino(new termino(1,256,1));

ec[360].anexa_termino(new termino(-0.1,360,5));
ec[360].anexa_termino(new termino(1,228,1));

ec[361].anexa_termino(new termino(-0.1,361,5));
ec[361].anexa_termino(new termino(1,215,1));

ec[362].anexa_termino(new termino(-0.1,362,5));
ec[362].anexa_termino(new termino(1,227,1));

ec[363].anexa_termino(new termino(-0.1,363,5));
ec[363].anexa_termino(new termino(1,205,1));

ec[364].anexa_termino(new termino(-0.1,364,5));
ec[364].anexa_termino(new termino(1,198,1));

ec[365].anexa_termino(new termino(-0.1,365,5));
ec[365].anexa_termino(new termino(1,238,1));

ec[0].num_fuentes=0;
ec[1].num_fuentes=0;
ec[2].num_fuentes=0;
ec[3].num_fuentes=0;
ec[4].num_fuentes=0;
ec[5].num_fuentes=0;
ec[6].num_fuentes=0;
ec[7].num_fuentes=0;
ec[8].num_fuentes=0;
ec[9].num_fuentes=0;
ec[10].num_fuentes=0;
ec[11].num_fuentes=0;
```

```
ec[12].num_fuentes=0;
ec[13].num_fuentes=0;
ec[14].num_fuentes=0;
ec[15].num_fuentes=0;
ec[16].num_fuentes=0;
ec[17].num_fuentes=0;
ec[18].num_fuentes=0;
ec[19].num_fuentes=0;
ec[20].num_fuentes=0;
ec[21].num_fuentes=0;
ec[22].num_fuentes=0;
ec[23].num_fuentes=0;
ec[24].num_fuentes=0;
ec[25].num_fuentes=0;
ec[26].num_fuentes=0;
ec[27].num_fuentes=0;
ec[28].num_fuentes=0;
ec[29].num_fuentes=0;
ec[30].num_fuentes=0;
ec[31].num_fuentes=0;
ec[32].num_fuentes=0;
ec[33].num_fuentes=0;
ec[34].num_fuentes=0;
ec[35].num_fuentes=0;
ec[36].num_fuentes=0;
ec[37].num_fuentes=0;
ec[38].num_fuentes=0;
ec[39].num_fuentes=0;
ec[40].num_fuentes=0;
ec[41].num_fuentes=0;
ec[42].num_fuentes=0;
ec[43].num_fuentes=0;
ec[44].num_fuentes=0;
ec[45].num_fuentes=0;
ec[46].num_fuentes=0;
ec[47].num_fuentes=0;
ec[48].num_fuentes=0;
ec[49].num_fuentes=0;
ec[50].num_fuentes=0;
ec[51].num_fuentes=0;
ec[52].num_fuentes=0;
ec[53].num_fuentes=0;
ec[54].num_fuentes=0;
ec[55].num_fuentes=0;
ec[56].num_fuentes=0;
ec[57].num_fuentes=0;
ec[58].num_fuentes=0;
ec[59].num_fuentes=0;
ec[60].num_fuentes=0;
ec[61].num_fuentes=0;
ec[62].num_fuentes=0;
ec[63].num_fuentes=0;
ec[64].num_fuentes=0;
ec[65].num_fuentes=0;
ec[66].num_fuentes=0;
ec[67].num_fuentes=0;
ec[68].num_fuentes=0;
ec[69].num_fuentes=0;
ec[70].num_fuentes=0;
ec[71].num_fuentes=0;
ec[72].num_fuentes=0;
ec[73].num_fuentes=0;
ec[74].num_fuentes=0;
ec[75].num_fuentes=0;
ec[76].num_fuentes=0;
ec[77].num_fuentes=0;
ec[78].num_fuentes=0;
ec[79].num_fuentes=0;
ec[80].num_fuentes=0;
ec[81].num_fuentes=0;
ec[82].num_fuentes=0;
```

```
ec[83].num_fuentes=0;
ec[84].num_fuentes=0;
ec[85].num_fuentes=0;
ec[86].num_fuentes=0;
ec[87].num_fuentes=0;
ec[88].num_fuentes=0;
ec[89].num_fuentes=0;
ec[90].num_fuentes=0;
ec[91].num_fuentes=0;
ec[92].num_fuentes=0;
ec[93].num_fuentes=0;
ec[94].num_fuentes=0;
ec[95].num_fuentes=0;
ec[96].num_fuentes=0;
ec[97].num_fuentes=0;
ec[98].num_fuentes=0;
ec[99].num_fuentes=0;
ec[100].num_fuentes=0;
ec[101].num_fuentes=0;
ec[102].num_fuentes=0;
ec[103].num_fuentes=0;
ec[104].num_fuentes=0;
ec[105].num_fuentes=0;
ec[106].num_fuentes=0;
ec[107].num_fuentes=0;
ec[108].num_fuentes=0;
ec[109].num_fuentes=0;
ec[110].num_fuentes=0;
ec[111].num_fuentes=0;
ec[112].num_fuentes=0;
ec[113].num_fuentes=0;
ec[114].num_fuentes=0;
ec[115].num_fuentes=0;
ec[116].num_fuentes=0;
ec[117].num_fuentes=0;
ec[118].num_fuentes=0;
ec[119].num_fuentes=0;
ec[120].num_fuentes=0;
ec[121].num_fuentes=0;
ec[122].num_fuentes=0;
ec[123].num_fuentes=0;
ec[124].num_fuentes=0;
ec[125].num_fuentes=0;
ec[126].num_fuentes=0;
ec[127].num_fuentes=0;
ec[128].num_fuentes=0;
ec[129].num_fuentes=0;
ec[130].num_fuentes=0;
ec[131].num_fuentes=0;
ec[132].num_fuentes=0;
ec[133].num_fuentes=0;
ec[134].num_fuentes=0;
ec[135].num_fuentes=0;
ec[136].num_fuentes=0;
ec[137].num_fuentes=0;
ec[138].num_fuentes=0;
ec[139].num_fuentes=0;
ec[140].num_fuentes=0;
ec[141].num_fuentes=0;
ec[142].num_fuentes=0;
ec[143].num_fuentes=0;
ec[144].num_fuentes=0;
ec[145].num_fuentes=0;
ec[146].num_fuentes=0;
ec[147].num_fuentes=0;
ec[148].num_fuentes=0;
ec[149].num_fuentes=0;
ec[150].num_fuentes=0;
ec[151].num_fuentes=0;
ec[152].num_fuentes=0;
ec[153].num_fuentes=0;
```

```
ec[154].num_fuentes=0;
ec[155].num_fuentes=0;
ec[156].num_fuentes=0;
ec[157].num_fuentes=0;
ec[158].num_fuentes=0;
ec[159].num_fuentes=0;
ec[160].num_fuentes=0;
ec[161].num_fuentes=0;
ec[162].num_fuentes=0;
ec[163].num_fuentes=0;
ec[164].num_fuentes=0;
ec[165].num_fuentes=0;
ec[166].num_fuentes=0;
ec[167].num_fuentes=0;
ec[168].num_fuentes=0;
ec[169].num_fuentes=0;
ec[170].num_fuentes=0;
ec[171].num_fuentes=0;
ec[172].num_fuentes=0;
ec[173].num_fuentes=0;
ec[174].num_fuentes=0;
ec[175].num_fuentes=0;
ec[176].num_fuentes=0;
ec[177].num_fuentes=0;
ec[178].num_fuentes=0;
ec[179].num_fuentes=0;
ec[180].num_fuentes=0;
ec[181].num_fuentes=0;
ec[182].num_fuentes=0;
ec[183].num_fuentes=0;
ec[184].num_fuentes=0;
ec[185].num_fuentes=0;
ec[186].num_fuentes=0;
ec[187].num_fuentes=0;
ec[188].num_fuentes=0;
ec[189].num_fuentes=0;
ec[190].num_fuentes=0;
ec[191].num_fuentes=0;
ec[192].num_fuentes=0;
ec[193].num_fuentes=0;
ec[194].num_fuentes=0;
ec[195].num_fuentes=0;
ec[196].num_fuentes=0;
ec[197].num_fuentes=0;
ec[198].num_fuentes=0;
ec[199].num_fuentes=0;
ec[200].num_fuentes=0;
ec[201].num_fuentes=0;
ec[202].num_fuentes=0;
ec[203].num_fuentes=0;
ec[204].num_fuentes=0;
ec[205].num_fuentes=0;
ec[206].num_fuentes=0;
ec[207].num_fuentes=0;
ec[208].num_fuentes=0;
ec[209].num_fuentes=0;
ec[210].num_fuentes=0;
ec[211].num_fuentes=0;
ec[212].num_fuentes=0;
ec[213].num_fuentes=0;
ec[214].num_fuentes=0;
ec[215].num_fuentes=0;
ec[216].num_fuentes=0;
ec[217].num_fuentes=0;
ec[218].num_fuentes=0;
ec[219].num_fuentes=0;
ec[220].num_fuentes=0;
ec[221].num_fuentes=0;
ec[222].num_fuentes=0;
ec[223].num_fuentes=0;
ec[224].num_fuentes=0;
```

```
ec[225].num_fuentes=0;
ec[226].num_fuentes=0;
ec[227].num_fuentes=0;
ec[228].num_fuentes=0;
ec[229].num_fuentes=0;
ec[230].num_fuentes=0;
ec[231].num_fuentes=0;
ec[232].num_fuentes=0;
ec[233].num_fuentes=0;
ec[234].num_fuentes=0;
ec[235].num_fuentes=0;
ec[236].num_fuentes=0;
ec[237].num_fuentes=0;
ec[238].num_fuentes=0;
ec[239].num_fuentes=0;
ec[240].num_fuentes=0;
ec[241].num_fuentes=0;
ec[242].num_fuentes=0;
ec[243].num_fuentes=0;
ec[244].num_fuentes=0;
ec[245].num_fuentes=0;
ec[246].num_fuentes=0;
ec[247].num_fuentes=0;
ec[248].num_fuentes=0;
ec[249].num_fuentes=0;
ec[250].num_fuentes=0;
ec[251].num_fuentes=0;
ec[252].num_fuentes=0;
ec[253].num_fuentes=0;
ec[254].num_fuentes=0;
ec[255].num_fuentes=0;
ec[256].num_fuentes=0;
ec[257].num_fuentes=0;
ec[258].num_fuentes=0;
ec[259].num_fuentes=0;
ec[260].num_fuentes=0;
ec[261].num_fuentes=0;
ec[262].num_fuentes=0;
ec[263].num_fuentes=0;
ec[264].num_fuentes=0;
ec[265].num_fuentes=0;
ec[266].num_fuentes=0;
ec[267].num_fuentes=0;
ec[268].num_fuentes=0;
ec[269].num_fuentes=0;
ec[270].num_fuentes=0;
ec[271].num_fuentes=0;
ec[272].num_fuentes=0;
ec[273].num_fuentes=0;
ec[274].num_fuentes=0;
ec[275].num_fuentes=0;
ec[276].num_fuentes=0;
ec[277].num_fuentes=0;
ec[278].num_fuentes=0;
ec[279].num_fuentes=0;
ec[280].num_fuentes=0;
ec[281].num_fuentes=0;
ec[282].num_fuentes=0;
ec[283].num_fuentes=0;
ec[284].num_fuentes=0;
ec[285].num_fuentes=0;
ec[286].num_fuentes=0;
ec[287].num_fuentes=0;
ec[288].num_fuentes=0;
ec[289].num_fuentes=0;
ec[290].num_fuentes=0;
ec[291].num_fuentes=0;
ec[292].num_fuentes=0;
ec[293].num_fuentes=0;
ec[294].num_fuentes=0;
ec[295].num_fuentes=0;
```

```
ec[296].num_fuentes=0;
ec[297].num_fuentes=0;
ec[298].num_fuentes=0;
ec[299].num_fuentes=0;
ec[300].num_fuentes=0;
ec[301].num_fuentes=0;

ec[355].num_fuentes=0;
ec[356].num_fuentes=0;
ec[357].num_fuentes=0;
ec[358].num_fuentes=0;
ec[359].num_fuentes=0;
ec[360].num_fuentes=0;
ec[361].num_fuentes=0;
ec[362].num_fuentes=0;
ec[363].num_fuentes=0;
ec[364].num_fuentes=0;
ec[365].num_fuentes=0;

ec[302].anexa_termino(new termino(-1/0.001,184,1));
ec[302].num_fuentes=1;
ec[302].fuente=new fuentes[1];
ec[302].fuente[0].magnitud=1/0.001;
ec[302].fuente[0].teta=0;

ec[303].anexa_termino(new termino(-1/0.001,187,1));
ec[303].num_fuentes=1;
ec[303].fuente=new fuentes[1];
ec[303].fuente[0].magnitud=1/0.001;
ec[303].fuente[0].teta=0;

ec[304].anexa_termino(new termino(-1/0.001,189,1));
ec[304].num_fuentes=1;
ec[304].fuente=new fuentes[1];
ec[304].fuente[0].magnitud=1/0.001;
ec[304].fuente[0].teta=0;

ec[305].anexa_termino(new termino(-1/0.001,191,1));
ec[305].num_fuentes=1;
ec[305].fuente=new fuentes[1];
ec[305].fuente[0].magnitud=1/0.001;
ec[305].fuente[0].teta=0;

ec[306].anexa_termino(new termino(-1/0.001,193,1));
ec[306].num_fuentes=1;
ec[306].fuente=new fuentes[1];
ec[306].fuente[0].magnitud=1/0.001;
ec[306].fuente[0].teta=0;

ec[307].anexa_termino(new termino(-1/0.001,195,1));
ec[307].num_fuentes=1;
ec[307].fuente=new fuentes[1];
ec[307].fuente[0].magnitud=1/0.001;
ec[307].fuente[0].teta=0;

ec[308].anexa_termino(new termino(-1/0.001,198,1));
ec[308].num_fuentes=1;
ec[308].fuente=new fuentes[1];
ec[308].fuente[0].magnitud=1/0.001;
ec[308].fuente[0].teta=0;

ec[309].anexa_termino(new termino(-1/0.001,201,1));
ec[309].num_fuentes=1;
ec[309].fuente=new fuentes[1];
ec[309].fuente[0].magnitud=1/0.001;
ec[309].fuente[0].teta=0;

ec[310].anexa_termino(new termino(-1/0.001,202,1));
ec[310].num_fuentes=1;
ec[310].fuente=new fuentes[1];
ec[310].fuente[0].magnitud=1/0.001;
```

```
ec[310].fuente[0].teta=0;

ec[311].anexa_termino(new termino(-1/0.001,207,1));
ec[311].num_fuentes=1;
ec[311].fuente=new fuentes[1];
ec[311].fuente[0].magnitud=1/0.001;
ec[311].fuente[0].teta=0;

ec[312].anexa_termino(new termino(-1/0.001,208,1));
ec[312].num_fuentes=1;
ec[312].fuente=new fuentes[1];
ec[312].fuente[0].magnitud=1/0.001;
ec[312].fuente[0].teta=0;

ec[313].anexa_termino(new termino(-1/0.001,209,1));
ec[313].num_fuentes=1;
ec[313].fuente=new fuentes[1];
ec[313].fuente[0].magnitud=1/0.001;
ec[313].fuente[0].teta=0;

ec[314].anexa_termino(new termino(-1/0.001,210,1));
ec[314].num_fuentes=1;
ec[314].fuente=new fuentes[1];
ec[314].fuente[0].magnitud=1/0.001;
ec[314].fuente[0].teta=0;

ec[315].anexa_termino(new termino(-1/0.001,214,1));
ec[315].num_fuentes=1;
ec[315].fuente=new fuentes[1];
ec[315].fuente[0].magnitud=1/0.001;
ec[315].fuente[0].teta=0;

ec[316].anexa_termino(new termino(-1/0.001,215,1));
ec[316].num_fuentes=1;
ec[316].fuente=new fuentes[1];
ec[316].fuente[0].magnitud=1/0.001;
ec[316].fuente[0].teta=0;

ec[317].anexa_termino(new termino(-1/0.001,217,1));
ec[317].num_fuentes=1;
ec[317].fuente=new fuentes[1];
ec[317].fuente[0].magnitud=1/0.001;
ec[317].fuente[0].teta=0;

ec[318].anexa_termino(new termino(-1/0.001,219,1));
ec[318].num_fuentes=1;
ec[318].fuente=new fuentes[1];
ec[318].fuente[0].magnitud=1/0.001;
ec[318].fuente[0].teta=0;

ec[319].anexa_termino(new termino(-1/0.001,223,1));
ec[319].num_fuentes=1;
ec[319].fuente=new fuentes[1];
ec[319].fuente[0].magnitud=1/0.001;
ec[319].fuente[0].teta=0;

ec[320].anexa_termino(new termino(-1/0.001,225,1));
ec[320].num_fuentes=1;
ec[320].fuente=new fuentes[1];
ec[320].fuente[0].magnitud=1/0.001;
ec[320].fuente[0].teta=0;

ec[321].anexa_termino(new termino(-1/0.001,229,1));
ec[321].num_fuentes=1;
ec[321].fuente=new fuentes[1];
ec[321].fuente[0].magnitud=1/0.001;
ec[321].fuente[0].teta=0;

ec[322].anexa_termino(new termino(-1/0.001,232,1));
ec[322].num_fuentes=1;
ec[322].fuente=new fuentes[1];
```

```
ec[322].fuente[0].magnitud=1/0.001;
ec[322].fuente[0].teta=0;

ec[323].anexa_termino(new termino(-1/0.001,237,1));
ec[323].num_fuentes=1;
ec[323].fuente=new fuentes[1];
ec[323].fuente[0].magnitud=1/0.001;
ec[323].fuente[0].teta=0;

ec[324].anexa_termino(new termino(-1/0.001,238,1));
ec[324].num_fuentes=1;
ec[324].fuente=new fuentes[1];
ec[324].fuente[0].magnitud=1/0.001;
ec[324].fuente[0].teta=0;

ec[325].anexa_termino(new termino(-1/0.001,239,1));
ec[325].num_fuentes=1;
ec[325].fuente=new fuentes[1];
ec[325].fuente[0].magnitud=1/0.001;
ec[325].fuente[0].teta=0;

ec[326].anexa_termino(new termino(-1/0.001,242,1));
ec[326].num_fuentes=1;
ec[326].fuente=new fuentes[1];
ec[326].fuente[0].magnitud=1/0.001;
ec[326].fuente[0].teta=0;

ec[327].anexa_termino(new termino(-1/0.001,244,1));
ec[327].num_fuentes=1;
ec[327].fuente=new fuentes[1];
ec[327].fuente[0].magnitud=1/0.001;
ec[327].fuente[0].teta=0;

ec[328].anexa_termino(new termino(-1/0.001,245,1));
ec[328].num_fuentes=1;
ec[328].fuente=new fuentes[1];
ec[328].fuente[0].magnitud=1/0.001;
ec[328].fuente[0].teta=0;

ec[329].anexa_termino(new termino(-1/0.001,248,1));
ec[329].num_fuentes=1;
ec[329].fuente=new fuentes[1];
ec[329].fuente[0].magnitud=1/0.001;
ec[329].fuente[0].teta=0;

ec[330].anexa_termino(new termino(-1/0.001,249,1));
ec[330].num_fuentes=1;
ec[330].fuente=new fuentes[1];
ec[330].fuente[0].magnitud=1/0.001;
ec[330].fuente[0].teta=0;

ec[331].anexa_termino(new termino(-1/0.001,253,1));
ec[331].num_fuentes=1;
ec[331].fuente=new fuentes[1];
ec[331].fuente[0].magnitud=1/0.001;
ec[331].fuente[0].teta=0;

ec[332].anexa_termino(new termino(-1/0.001,255,1));
ec[332].num_fuentes=1;
ec[332].fuente=new fuentes[1];
ec[332].fuente[0].magnitud=1/0.001;
ec[332].fuente[0].teta=0;

ec[333].anexa_termino(new termino(-1/0.001,256,1));
ec[333].num_fuentes=1;
ec[333].fuente=new fuentes[1];
ec[333].fuente[0].magnitud=1/0.001;
ec[333].fuente[0].teta=0;

ec[334].anexa_termino(new termino(-1/0.001,257,1));
ec[334].num_fuentes=1;
```

```
ec[334].fuente=new fuentes[1];
ec[334].fuente[0].magnitud=1/0.001;
ec[334].fuente[0].teta=0;

ec[335].anexa_termino(new termino(-1/0.001,259,1));
ec[335].num_fuentes=1;
ec[335].fuente=new fuentes[1];
ec[335].fuente[0].magnitud=1/0.001;
ec[335].fuente[0].teta=0;

ec[336].anexa_termino(new termino(-1/0.001,260,1));
ec[336].num_fuentes=1;
ec[336].fuente=new fuentes[1];
ec[336].fuente[0].magnitud=1/0.001;
ec[336].fuente[0].teta=0;

ec[337].anexa_termino(new termino(-1/0.001,263,1));
ec[337].num_fuentes=1;
ec[337].fuente=new fuentes[1];
ec[337].fuente[0].magnitud=1/0.001;
ec[337].fuente[0].teta=0;

ec[338].anexa_termino(new termino(-1/0.001,268,1));
ec[338].num_fuentes=1;
ec[338].fuente=new fuentes[1];
ec[338].fuente[0].magnitud=1/0.001;
ec[338].fuente[0].teta=0;

ec[339].anexa_termino(new termino(-1/0.001,270,1));
ec[339].num_fuentes=1;
ec[339].fuente=new fuentes[1];
ec[339].fuente[0].magnitud=1/0.001;
ec[339].fuente[0].teta=0;

ec[340].anexa_termino(new termino(-1/0.001,272,1));
ec[340].num_fuentes=1;
ec[340].fuente=new fuentes[1];
ec[340].fuente[0].magnitud=1/0.001;
ec[340].fuente[0].teta=0;

ec[341].anexa_termino(new termino(-1/0.001,273,1));
ec[341].num_fuentes=1;
ec[341].fuente=new fuentes[1];
ec[341].fuente[0].magnitud=1/0.001;
ec[341].fuente[0].teta=0;

ec[342].anexa_termino(new termino(-1/0.001,274,1));
ec[342].num_fuentes=1;
ec[342].fuente=new fuentes[1];
ec[342].fuente[0].magnitud=1/0.001;
ec[342].fuente[0].teta=0;

ec[343].anexa_termino(new termino(-1/0.001,275,1));
ec[343].num_fuentes=1;
ec[343].fuente=new fuentes[1];
ec[343].fuente[0].magnitud=1/0.001;
ec[343].fuente[0].teta=0;

ec[344].anexa_termino(new termino(-1/0.001,282,1));
ec[344].num_fuentes=1;
ec[344].fuente=new fuentes[1];
ec[344].fuente[0].magnitud=1/0.001;
ec[344].fuente[0].teta=0;

ec[345].anexa_termino(new termino(-1/0.001,283,1));
ec[345].num_fuentes=1;
ec[345].fuente=new fuentes[1];
ec[345].fuente[0].magnitud=1/0.001;
ec[345].fuente[0].teta=0;
ec[346].anexa_termino(new termino(-1/0.001,286,1));
ec[346].num_fuentes=1;
```

```
ec[346].fuente=new fuentes[1];
ec[346].fuente[0].magnitud=1/0.001;
ec[346].fuente[0].teta=0;

ec[347].anexa_termino(new termino(-1/0.001,287,1));
ec[347].num_fuentes=1;
ec[347].fuente=new fuentes[1];
ec[347].fuente[0].magnitud=1/0.001;
ec[347].fuente[0].teta=0;

ec[348].anexa_termino(new termino(-1/0.001,288,1));
ec[348].num_fuentes=1;
ec[348].fuente=new fuentes[1];
ec[348].fuente[0].magnitud=1/0.001;
ec[348].fuente[0].teta=0;

ec[349].anexa_termino(new termino(-1/0.001,290,1));
ec[349].num_fuentes=1;
ec[349].fuente=new fuentes[1];
ec[349].fuente[0].magnitud=1/0.001;
ec[349].fuente[0].teta=0;

ec[350].anexa_termino(new termino(-1/0.001,293,1));
ec[350].num_fuentes=1;
ec[350].fuente=new fuentes[1];
ec[350].fuente[0].magnitud=1/0.001;
ec[350].fuente[0].teta=0;

ec[351].anexa_termino(new termino(-1/0.001,294,1));
ec[351].num_fuentes=1;
ec[351].fuente=new fuentes[1];
ec[351].fuente[0].magnitud=1/0.001;
ec[351].fuente[0].teta=0;

ec[352].anexa_termino(new termino(-1/0.001,295,1));
ec[352].num_fuentes=1;
ec[352].fuente=new fuentes[1];
ec[352].fuente[0].magnitud=1/0.001;
ec[352].fuente[0].teta=0;

ec[353].anexa_termino(new termino(-1/0.001,296,1));
ec[353].num_fuentes=1;
ec[353].fuente=new fuentes[1];
ec[353].fuente[0].magnitud=1/0.001;
ec[353].fuente[0].teta=0;

ec[354].anexa_termino(new termino(-1/0.001,299,1));
ec[354].num_fuentes=1;
ec[354].fuente=new fuentes[1];
ec[354].fuente[0].magnitud=1/0.001;
ec[354].fuente[0].teta=0;

}//fin de caso_estudio_3
```