Centro de Investigación y de Estudios Avanzados del

Instituto Politécnico Nacional

Departamento de Ingeniería Eléctrica

Sección Computación

Título

Algoritmos de ordenamiento y teoría de gráficas en una computadora paralela de tipo hipercubo.

Trabajo realizado para obtener el grado de Maestro en Ciencias, en la especialidad de Ingeniería Eléctrica

Por el Alumno :

Carlos Alberto Hernández Hernández

Trabajo dirigido por:

Dr. Sergio V. Chapa Vergara.

México, D.F.

septiembre de 1995

A mis Padres.

A mis Hermanos.

Introducción

Esta tesis tuvo su motivación del curso de Autómatas Celulares (AC), impartido por el Dr. Harold V. McIntosh, ya que los AC[1] presentan un magnífico paradigma para la computación paralela. Los AC fueron propuestos por Von Neumann, como un modelo el cual opera en una manera altamente paralela, mediante copias o variantes de si mismas y en modelos para reproducción biológicas. Esta propuesta consistió en una nueva alternativa de enfoque paralelo para la solución de problemas. En los AC se presentan los problemas típicos de computación paralela, donde una de las preguntas más importantes es encontrar métodos generales para realizar aplicaciones útiles. Por otra parte una de las mayores problemáticas de las computadoras paralelas de propósito general es el pobre desempeño que se tiene por lo que uno de los cuellos de botella es el uso eficiente del hardware que depende del software. Por lo que tomando en cuenta estos dos antecedentes se induce al planteamiento de la pregunta ¿Cómo utilizar (programar) eficientemente las computadoras paralelas con métodos generales, para resolver problemas comunes?, que es el centro de motivación de este trabajo.

Aunque la aplicación de la tecnología en computación masivamente paralela se tiene disponible desde varios años atrás; básicamente, la mayoría de los algoritmos y sus aplicaciones más populares han sido diseñados bajo un esquema secuencial. La generación de código para un maquina masivamente paralela es un tarea no trivial y requiere de: a) una nueva forma de pensar los problemas, b) del desarrollo de los algoritmos en forma paralela y c) de su adecuada implantación de los modelos paralelos propuestos. Los anteriores son pasos que se darán para obtener un mayor rendimiento en la resolución de problemas. En esta dirección es necesario diseñar nuevos algoritmos los cuales se apliquen a un modelo de paralelismo (arquitectura) con el fin de resolver problemas generales. El trabajo y la dificultad que implican el desarrollo de dichos algoritmos requiere que el problema sea interesante, lo suficientemente general y regular para compensar el esfuerzo realizado.

Por lo que el objetivo de esta tesis es mostrar las ventajas en el estilo de programación paralela de datos múltiples para código similar (SCMD, del inglés same-code/multiple-data), en la programación eficiente de computadoras paralelas, por lo que se eligieron problemas tipicos de computación como el ordenamiento y dos problemas de teoria de gráficas. Además se tomo como base la arquitectura hipercubo. Por lo que otro punto importante en esta tesis es el estudio de las características de la arquitectura hipercubo ya que de esta forma es posible mapear eficientemente tanto los algoritmos presentados como además en un futuro poder mapear otros algoritmos. Es conveniente hacer énfasis en que el modelo SCMD es aplicable a otras arquitecturas.

[1] Una buena revisión de autómatas celulares se puede encontrar en [Wolfram 86].

Esta tesis consta de tres capítulos como se muestra en la figura F1, el capítulo uno presenta un marco téorico, orienta al lector en el contenido de esta tesis además , el capitulo dos abarca los algoritmos de ordenamiento y además plantea una clasificación de algoritmos paralelos de ordenamiento, el capítulo tres es acerca de dos de los algoritmos de teoría de gráficas más estudiados, el árbol de extensión mínimal y el problema del vendedor viajero y por último se dan conclusiones generales de los planteamientos iniciales así como perspectivas futuras. Los algoritmos que se tratan se eligieron ya que una gran cantidad de problemas pueden ser reducidos a estos algoritmos. Contando en la sección de computación con una computadora paralela del tipo red de interconexión con arquitectura hipercubo NCUBE 2, se planteó la programación paralela SCMD en particular para las computadoras de red de interconexión de tipo hipercubo.

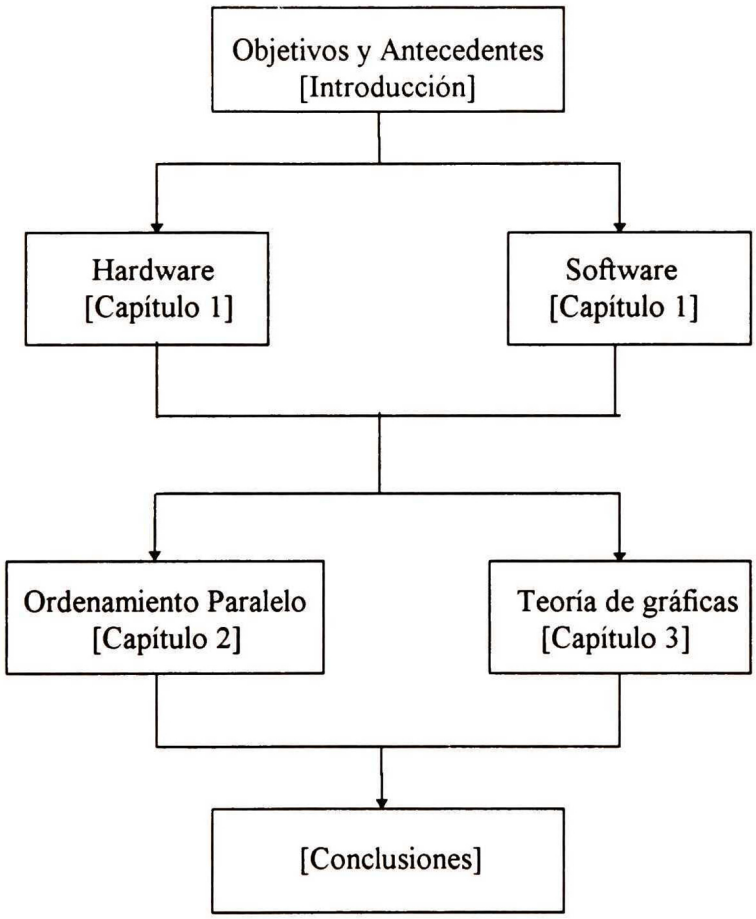


Figura F1 *Estructura del trabajo*

Agradecimientos

Varios amigos, compañeros y profesores han contribuido en la realización de este trabajo. Agradezco a todos la ayuda así como las críticas recibidas.

Quiero decir que la sección de computación del CINVESTAV, me proporcionó un ambiente de trabajo ideal para la realización de esta tesis.

Agradezco en particular (con riesgo de omitir a alguien ó a muchos) a mi maestra de primaria Adela por enseñarme el inicio del camino, al profesor de secundaria José Parada por su apoyo, al Dr. Jesús Figueroa por sus consejos, al Dr. Ibarra Zanatha y a la M. en C. Carlota Tamayo por su confianza en mí, al Dr. Sergio Chapa por su paciencia y guía en esta tesis, al M. en C. Alejandro Tinoco por sus valiosos comentarios, al M. en C. Oscar Olmedo por su tiempo en la revisión de esta tesis, al Lic. Ignacio Vega y a Felipa Rosas por su ayuda en la elaboración de este escrito.

Quiero agradecer a mis compañeros Alfredo, Efren, Manuel, Nacho, Raúl, Tanit, a los demás compañeros del CINVESTAV y a los MUPPETS, por su amistad y los buenos momentos que pasamos juntos.

También agradezco al CINVESTAV por darme la oportunidad de superarme, a mi país y al CONACyT por los recursos brindados para realizar la maestría.

Contenido

Capítulo 1

Computación paralela

El principal objetivo de este capítulo es de presentar un marco teórico de los modelos de paralelismo y programación, también se definen algunos conceptos y notaciones que se utilizarán en el resto de este trabajo. Primero en 1.1 Computación paralela, se mencionan algunas características generales de la computación paralela, tanto a nivel software como a nivel hardware; en 1.2 Clasificación de arquitecturas, se revisa la clasificación proporcionada por Michael S. Flynn en [Flynn 72], que divide a las computadoras en base a el número de instrucciones y datos utilizados simultáneamente en cuatro categorías, estas categorías nos permite analizar las características de las computadoras paralelas; en 1.2.1 Computadoras MIMD se revisan los dos principales modelos de computadoras MIMD, el modelo de memoria compartida y el modelo de red también conocido como de paso de mensajes; en 1.2.1.1 Arquitectura hipercubo se define en que consiste un hipercubo así como algunas propiedades importantes de los hipercubos; en 1.2.1.2 Computadora NCUBE 2 se describen las características principales de la computadora NCUBE 2, tales como los procesadores, canales de comunicación, así como el algoritmo de ruteo que utiliza la computadora NCUBE 2; en 1.2.2 Desempeño de arquitecturas paralelas se definen algunos criterios para medir el desempeño de computadoras paralelas tales como: la aceleración y eficiencia de los algoritmos paralelos, además se revisa el efecto de la ley de Amdahl en el desempeño de las computadoras paralelas; en 1.3 Programación de sistemas paralelos se revisa el modelo tradicional que consiste en realizar compiladores que detecten el paralelismo en código secuencial; así como los cinco principales estilos de programación: procedural, orientada a objetos, funcional, lógica y basada en conocimiento y por ultimo se explica la metodología seguida en este trabajo, tanto en el análisis como el diseño de los algoritmos paralelos tratados en esta tesis; en 1.3.1 Complejidad paralela y 1.3.2 Clases *P* y *NC* se analizan las clases de complejidad de la programación paralela y se revisan las características de los problemas *NC* así como los problemas *P-Completos*.

1.1 Computación Paralela

Desde tiempo atrás ha existido la necesidad de resolver problemas de computación muy grandes, las necesidades por una computación más rápida han estado en contextos que involucran desde: ecuaciones diferenciales parciales en dinámica de fluidos y predicciones de clima, hasta problemas de optimización y procesamiento de imágenes. El deseo de resolver problemas cada vez más complejos, ha propiciado que se sobrepasen las capacidades de cómputo motivando el desarrollo de computadoras más rápidas. Los avances tecnológicos han permitido construir computadoras cada vez más poderosas. Ya que en la construcción de computadoras se está llegando a los límites en la integración de circuitos una opción para obtener más capacidad de cómputo es el paralelismo, se utiliza principalmente dos enfoques

[Fox 88]. El primero es tomar ventaja del paralelismo local, en este caso se logra con unidades de aritmética vectorizada, que permiten efectuar operaciones aritméticas en paralelo sobre un conjunto de datos, como en las computadoras CRAY YMP™ El otro enfoque se basa en el paralelismo global, que permiten ejecutar instrucciones y bloques de código en un conjunto de datos, como en las máquinas CONNECTION MACHINE™, NCUBE 2™ y en el modelo CRAY T3D™, se espera que en el futuro las computadoras paralelas sean construidas con procesadores que incorporen aritmética vectorial[2] La disponibilidad de computadoras paralelas poderosas no esta acompañada con el software para su utilización, por lo que los modelos de programación, algoritmos y herramientas de desarrollo son de gran importancia, ya que definen la utilidad y facilidad de uso de los sistemas paralelos; por lo tanto la computación paralela actualmente es una área de intensa investigación, ya que se necesitan tanto computadoras más rápidas como también software más eficiente.

1.2 Clasificación de arquitecturas

Uno de los problemas de computo paralelo; típico de las nuevas áreas, es las confusión en los términos utilizados. Empezaremos introduciendo algunos términos usados frecuentemente:

Multiprogramación.- Es una característica que ofrece el sistema operativo (por ejemplo UNIX ™), que permite ejecutar programas independientes de una forma semi-simultanea, a través de mecanismos tales como la asignación de rebanadas de tiempo a los programas.

Multitareas.- Es una característica que ofrece el sistema operativo (por ejemplo UNIX ™), que permite particionar un programa en varias tareas que se ejecutan en forma concurrente.

Multiprocesamiento.- Es la ejecución simultánea de dos o más secuencias de instrucciones, por dos o más procesadores.

Multiprocesador.- Es una red de procesadores unidos por una red o un interruptor de comunicación. Cada procesador tiene acceso a su propia memoria distribuida local o a una memoria compartida común.

En las computadoras paralelas con múltiples elementos de proceso, existe un conflicto entre el número y *tamaño de grano* de los procesadores. La capacidad de procesamiento de los nodos o elementos de procesamiento en una arquitectura paralela, es usualmente referido como tamaño de grano, podemos escoger entre "procesadores pequeños o procesadores poderosos" como elementos básicos para la máquina de

[2]La computadora CRAY T3D™ se puede integrar a una computadora CRAY Y-MP™ y de esta forma atacar los dos enfoques de paralelismo, aunque en realidad siguen siendo dos computadoras y no una computadora paralela.

procesamiento paralelo, lo que se referirá como máquinas de grano fino o grano grueso respectivamente. La *granularidad* se refiere al poder de cada elemento en la arquitectura que puede variar de procesadores de un bit a treinta y dos o sesenta y cuatro bits, como se comenta en [Rietman 90].

Han sido propuestas varias clasificaciones de sistemas paralelos de cómputo, una de las primeras y más utilizada. es la clasificación proporcionada por Michael S. Flynn en [Flynn 72], también aparece en [Rietman 90], esta clasificación se basa en el número de instrucciones y datos utilizados simultáneamente, durante la ejecución del programa correspondiente. Sea n_i y n_d las variables que denotan respectivamente, el número de instrucciones y datos que pueden ser procesados al mismo tiempo en una computadora, estos valores nos permitirán analizar el grado de paralelismo del sistema.

- Una Instrucción Un Dato (Single Instruction Stream Single Data Stream (SISD)).- Con $n_i = n_d = 1$. El tipo más convencional de computadora con un procesador conteniendo una ALU con aritmética escalar.

- Múltiples Instrucciones Un Dato (Multiple Instruction Stream Single Data Stream (MISD)) .- $n_i > 1$, $n_d = 1$. Un conjunto de procesadores ejecutan distintas instrucciones sobre los mismos datos, aunque podemos decir que este modo de operación es generalmente irreal y no hay computadoras que se ubiquen en esta categoría.

- Una Instrucción Múltiples Datos (Single Instruction Stream Multiple Data Stream (SIMD)) .- $n_i = 1$, $n_d\ 1 > 1$. Esta clase incluye máquinas con una unidad de control y múltiples ALUs. En estas computadoras varios elementos de información pueden ser accesados de la memoria y procesados simultáneamente por una sola instrucción, los arreglos de procesadores pertenecen a esta categoría. También podemos ubicar a los procesadores vectoriales en esta clase, haciendo la consideración que los vectores son un arreglo de procesadores multiplexados en el tiempo.

- Múltiples Instrucciones Múltiples Datos (Multiple Instruction Stream Multiple Data Stream (MIMD)) .- $n_i > 1$, $n_d > 1$.- Las máquinas que caen dentro de esta categoría son capaces de ejecutar varios programas independientes simultáneamente, las máquinas con multiprocesadores pertenecen a esta categoría.

En base a estas clases que acabamos de definir se presenta en la figura 1.1, una clasificación con algunas de las computadoras paralelas actuales, esta clasificación se puede ver a detalle en [Bell 94] . En la siguiente sección se verán las caracteristicas de las computadoras paralelas de tipo MIMD.

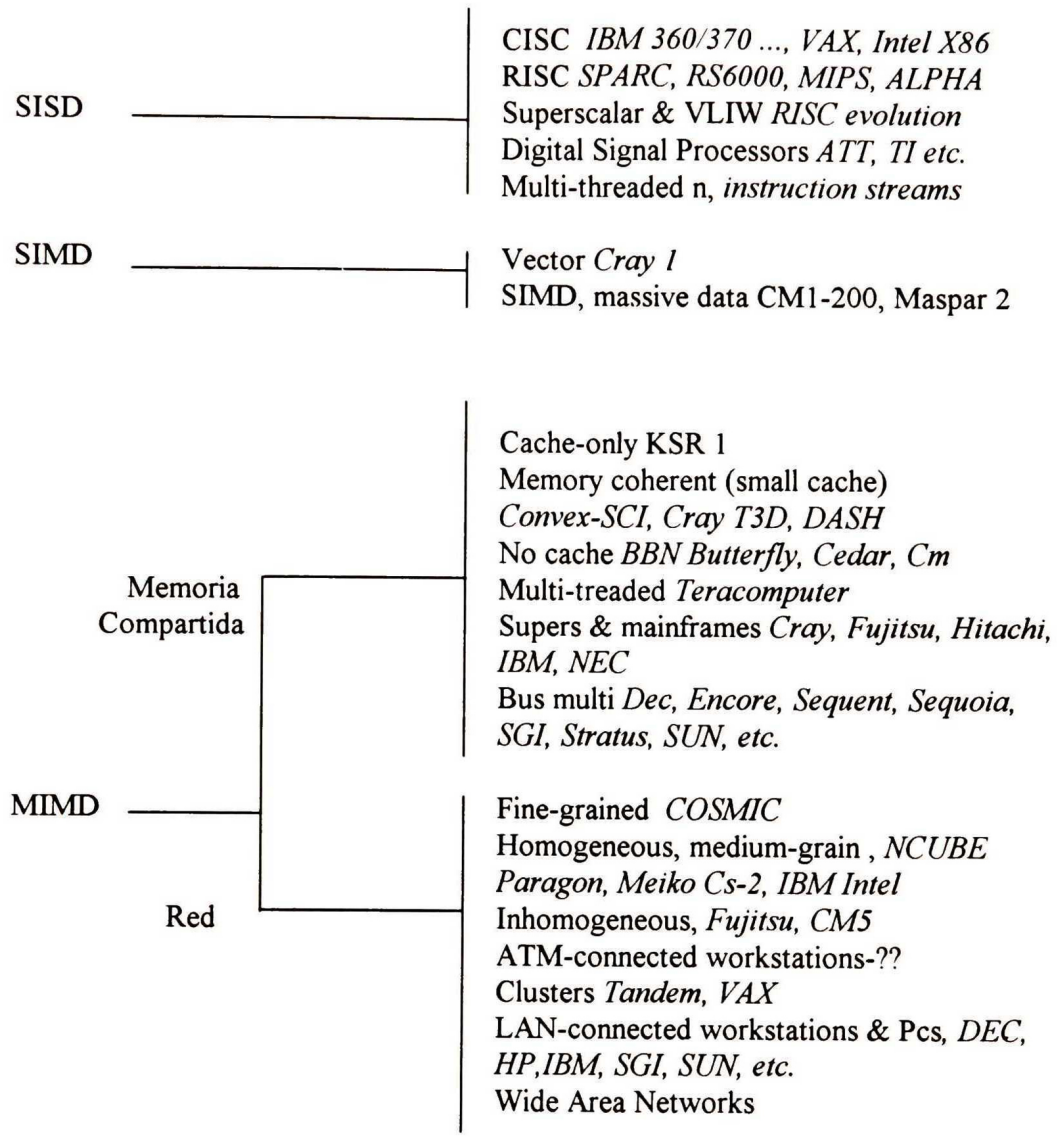


Figura 1.1 *Clasificación de Computadoras Paralelas*

1.2.1 Computadoras MIMD

Las computadoras MIMD se pueden dividir en computadoras de memoria compartida y de red de interconexión, como se puede observar en la figura 1.2, estas divisiones son en realidad una extrapolación, de una computadora secuencial convencional de tipo Von Neumann, a una computadora paralela [Jájá 92].

El modelo de memoria compartida consiste en un conjunto de procesadores, los que pueden tener su propia memoria local y ejecutar programas localmente, los procesadores se comunican entre si, intercambiando datos a través de la unidad de memoria compartida .

El modelo de red, también llamado de paso de mensajes, puede ser visto como una gráfica $G = (V,L)$, donde cada nodo j e $i \in V$ representa un procesador y cada lazo $(j,i) \in L$ representa un canal de comunicaciones entre el procesador j y el i, cada procesador tiene su propia memoria local y no hay memoria compartida disponible. En esta tesis nos ocuparemos del modelo de red y en particular de la red de interconexión de tipo hipercubo.

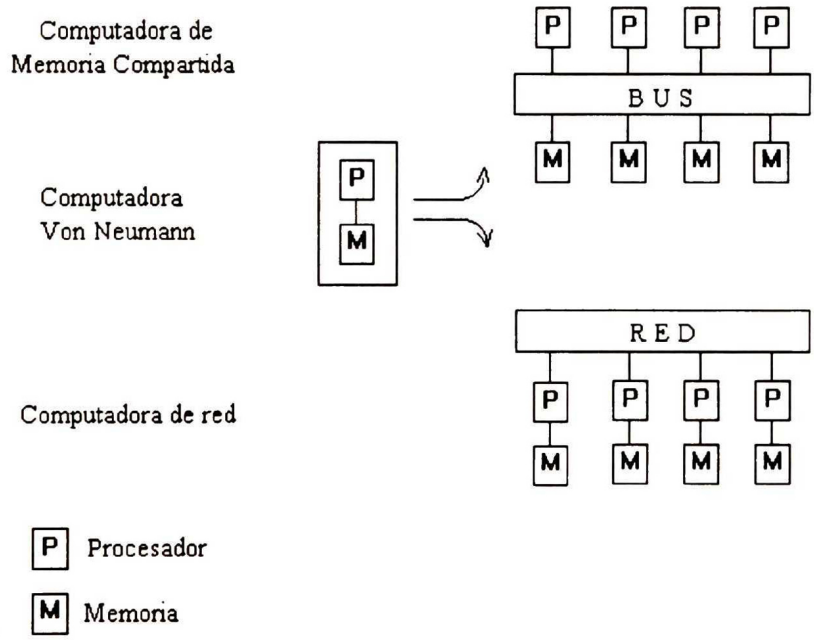


Figura 1.2 *Modelo de memoria compartida y de red*

1.2.1.1 Arquitectura Hipercubo

La red de interconexión de tipo hipercubo es una de las más eficientes y versátiles redes para computadoras paralelas de tipo MIMD, ya que es utilizada tanto para tareas de propósito general como para tareas de propósito específico. Además un modelo de hipercubo puede simular eficientemente otro tipo de redes, como las de tipo de arreglo, árbol y malla entre otras como se puede ver en [Leighton 92] y [Ranka 90].

Una computadora con una arquitectura de tipo hipercubo consiste de $p = 2^k$ procesadores interconectados, donde k nos indica la dimensión del hipercubo, que es la generalización de un cubo tridimensional a varias dimensiones y se define como sigue:

Se asume que los procesadores se etiquetan con un numero i, donde $0 \leq i \leq p-1$. Sea $i_{k-1}\, i_{k-2} \ldots i_0$ la representación binaria de i, entonces el procesador P_i es conectado al procesador P_j, donde $j = i_{k-1} \ldots \bar{i}_j \ldots i_0$, y $\bar{i}_j = 1 - i_j$, para $0 \leq j \leq k-1$, en otras palabras, dos procesadores están conectados, si sólo si, sus índices difieren exactamente en un solo bit, es decir dos procesadores están conectados si sólo si, la *distancia de Hamming* entre sus etiquetas de identificación es igual a uno. La distancia de Hamming es igual al número de bits en que son diferentes dos etiquetas binarias.

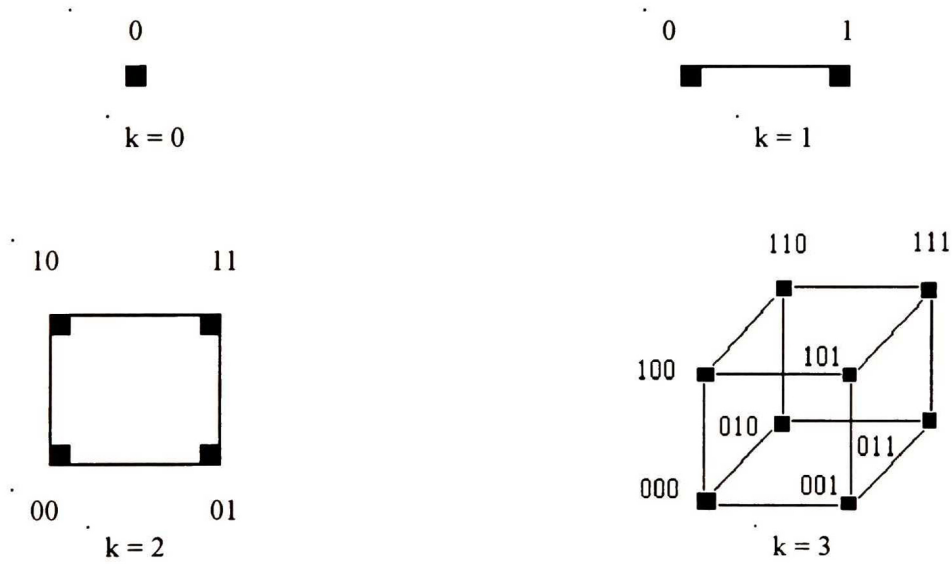


Figura 1.3 *Hipercubo con k=0,1,2 y 3*

Un hipercubo es completamente simétrico, es decir todos los patrones de interconexión son iguales en todos los procesadores, además los procesadores tiene una estructura recursiva y podemos extender un hipercubo k dimensional a uno $(k+1)$ dimensional, adicionando a un hipercubo k dimensional el bit más significativo igual a cero y otro hipercubo k dimensional el bit más significativo igual a uno y conectando los

procesadores correspondientes de los cubos; esta propiedad de poder extender o descomponer[3] un hipercubo, permite la implantación de una manera sencilla de los algoritmos de tipo divide y vencerás. Un hipercubo con tres dimensiones tiene 2^3 nodos y 3 lazos por nodo, y por tanto un hipercubo con k dimensiones tiene 2^k nodos y k lazos por nodo. Tal como se ilustra en la figura 1.3. La computadora NCUBE 2 que se utilizó en el desarrollo de este trabajo cuenta con una arquitectura hipercubo.

1.2.1.2 Computadora NCUBE 2

El CINVESTAV cuenta con una computadora NCUBE 2 con ocho procesadores y 26 canales de comunicación y 2 de I/O por procesador, esta computadora es de tipo MIMD, utiliza una red de interconexión de tipo hipercubo y cuenta con el sistema operativo IRIX™ y consiste en lo siguiente:

- Procesadores (Elementos de proceso): La computadora NCUBE 2 es una red de CPUs independientes, donde cada elemento tiene su propia unidad de punto flotante integrada, memoria local, y hardware de comunicación. Cada procesador ejecuta una copia completa del sistema operativo nCx además opera a 2.5 MFLOPS. El número de procesadores es escalable y puede crecer hasta 8192 procesadores.

- Canales de comunicación: Un procesador es conectado a los otros procesadores por medio de canales de comunicación con acceso directo a memoria (DMA de sus siglas en inglés) y hardware de ruteo independientes (26 canales de comunicación por procesador, por lo que se cuenta con 13 canales Full Dúplex), ver [Schmidt 94]. El tiempo de inicio de los canales es T_s = 150 μs y copiar de la memoria del procesador al buffer de comunicación tiene un ancho de banda de R_c= 21 MBytes/s, la transferencia es de Rt = 2.75 MBytes/s y el tiempo de paso entre un procesador intermedio es T_h = 2μs, cuando $n \rightarrow \infty$, donde n es el número de datos a transferir. El tiempo de transferencia entre procesadores es de 2.2 MBytes por segundo. Aun cuando sólo existen canales directos de hardware entre procesadores vecinos en la red, el hardware de ruteo permite que la velocidad de comunicación entre procesadores lejanos sea tan rápida como entre procesadores cercanos.

Para el envío de mensajes entre los procesadores la computadora NCUBE 2 utiliza el algoritmo *e-cube,* para la generación de la trayectoria entre dicho procesadores (esto es a nivel hardware). El algoritmo *e-cube* obtiene la trayectoria de ruteo realizando un OR - exclusivo lógico entre las direcciones de los nodos fuente y destino, los bits en uno de la trayectoria calculada se procesan de derecha a izquierda, proporcionando las coordenadas de como viajara el mensaje en el hipercubo. Por ejemplo, para enviar un mensaje del nodo "000" al nodo "101" en el hipercubo con k =3, se sigue la trayectoria que se muestra en la figura 1.4.

[3]Aumentando o disminuyendo la dimensión del hipercubo, en caso de disminuir la dimensión se realiza el proceso inverso al indicado en el texto obteniendo dos sub-hipercubos.

De el procesador	Puerto de salida	A el procesador	Puerto de salida	A el procesador
000	0	001	2	101

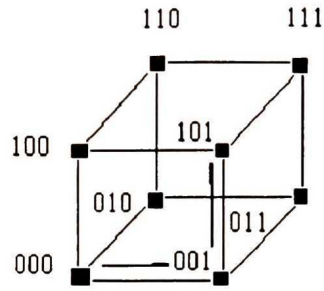


Figura 1.4 *Recorrido del algoritmo e-cube*

- Procesadores de entrada/salida (I/O). Al igual que los canales de comunicación descritos anteriormente, cada procesador tiene canales separados con DMA dedicados para transferir y recibir del procesador de I/O (cada procesador cuenta con 2 canales de comunicación de I/O o un canal Full Duplex). Estos canales de I/O trabajan a la misma velocidad que los canales de comunicación entre los procesadores. Los procesadores de I/O tienen la misma configuración de CPU y memoria, que los procesadores del sistema, pero con interfaces adicionales de hardware y software para discos, pantallas y otros recursos periféricos.

1.2.2 Desempeño de arquitecturas paralelas

Aunque no existe una manera generalmente aceptada para la evaluación de computadoras, uno de los criterios más utilizados para la evaluación de computadoras es el desempeño, podemos especificar el desempeño computacional por el número de operaciones de punto flotante que puede ser ejecutada en un segundo (FLOPS).

El desempeño pico de un sistema de cómputo, es el mayor desempeño que puede ser alcanzado por el sistema, esto es cuando todos los módulos de cómputo son utilizados completamente.

El desempeño sostenido de un sistema, es el desempeño que tiene el sistema en alguna tarea en particular, es decir es el desempeño en que usualmente funciona el sistema.

Se analizará la *aceleración* de un algoritmo de la siguiente forma,

$$A(n) = \frac{TS(n)}{TP(n)}$$

También la *eficiencia* del algoritmo que se puede definir como,

$$E(n) = \frac{TS(n)}{pTP(n)}$$

donde las variables significan :

n = número de datos
p = número de procesadores
E (n) = Eficiencia
TS (n) = Tiempo del mejor algoritmo secuencial conocido
TP (n) = Tiempo del algoritmo paralelo a analizar

Existen varios factores que pueden impedir que una computadora alcance su desempeño pico cuando corre un programa en paralelo, una fuente importante de degradación del sistema es que hay código que a causa de algunas dependencias es inherentemente secuencial, además de la utilización de instrucciones secuenciales para preparar y controlar la ejecución de los programas paralelos. La ley de Amdahl expresa el hecho que la secuencialidad inherente de los algoritmos es un factor para limitar el desempeño en cualquier computadora paralela, en [Zima 91] y en [Waldrop 88] se comenta más a detalle este punto.

Considerando que α es la parte de código que debe ser ejecuta en forma secuencial, que una operación secuencial se realiza en una unidad de tiempo y además que τ es la cantidad de partes de código u operaciones que pueden ser ejecutadas en paralelo[4]

Se pueden ejecutar m operaciones secuenciales en un tiempo m y en caso de utilizar operaciones paralelas se realizan las m operaciones en un tiempo que se calcula por $m\left(\alpha + \frac{(1-\alpha)}{\tau}\right)$, donde la *aceleración* es $\sigma(\alpha,\tau) = \left(\frac{1}{\alpha + \frac{(1-\alpha)}{\tau}}\right)$.

[4]Si cada parte u operación es ejecutada en un procesador, τ es el número de procesadores que se pueden utilizar en el algoritmo.

Para toda α y $\tau(0 \le \alpha \le 1, \tau \ge 1)$ nosotros tenemos que $1 \le \sigma(\alpha, \tau) \le \tau$, es decir σ nos indicara que tan paralelizable es un algoritmo; por lo que para toda τ, $\sigma(0, \tau) = \tau$ y $\sigma(1, \tau) = 1$, indica que si el valor de σ es igual a τ el código será completamente paralelizable, en caso que el valor de σ sea igual a 1, indica que el algoritmo funcionará de manera secuencial no importando en número de procesadores que se utilicen. En la mayoría de sistemas τ es una constante, típicamente de orden 10, en base a esto σ se transforma en una función de α. Se observa que un pequeño incremento en α puede ocasionar un descenso considerable en la aceleración como se observa en la figura 1.5. En particular se nota que cuando el 90% del código pude correr en paralelo, sólo la mitad o la tercera parte del desempeño pico puede ser alcanzado. En otras palabras Gene Amdahl propuso que toda computación contiene al menos algún paso que debe ser resuelto secuencialmente, por ejemplo una computadora no puede adicionar una cantidad x a otra y, a menos que se conozca el valor de las cantidades. Por lo tanto Amdahl mantiene que éste es un límite a que tan rápido puede ser una computadora multiprocesadora ya que eventualmente el embotellamiento secuencial dominará.

De cualquier forma α es una medida dinámica, además que la mayoría del tiempo de ejecución de un programa (80% al 90%) es gastado en pequeñas secciones (10% al 20%) del código. Por lo tanto las opciones son: por un lado tratar de optimizar las secciones críticas y por otro lado redistribuir las cargas de trabajo rediseñando los algoritmos.

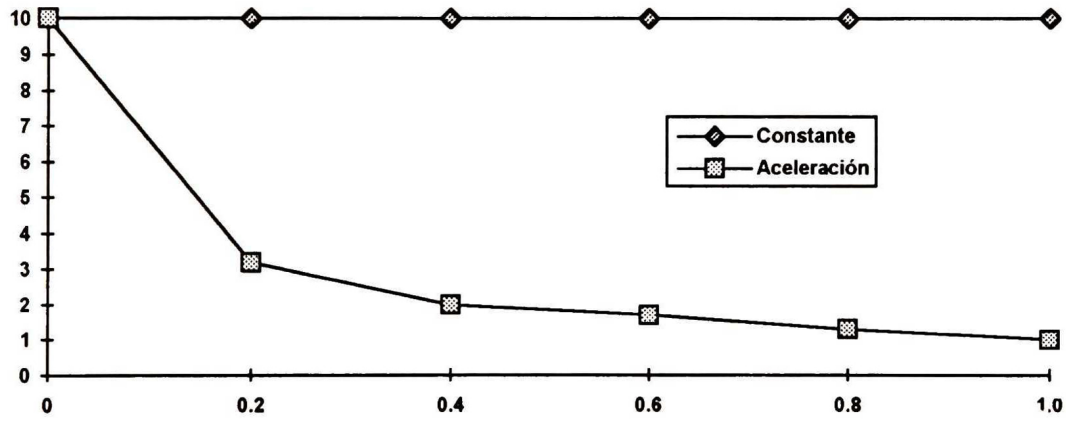


Figura 1.5 *Aceleración $\sigma(\alpha, \tau)$ como función de α para $\tau = 10$*

1.3 Programación de Sistemas Paralelos

El enfoque tradicional en la programación de sistemas paralelos, como se menciona en [Suaya 90] y [Treleaven 90], consiste en extraer el paralelismo de los sistemas de software existentes, tanto de C o Fortran, como de los lenguajes lógicos como prolog, del leguaje de bases de datos SQL y del sistema operativo UNIX™, para que un programa secuencial pueda correr en una computadora paralelas. Esto se puede lograr con un compilador que detecte el paralelismo en un programa fuente secuencial, que puede ser escrito en lenguajes

como C, Fortran, Pascal, Lisp, Prolog, SQL etc. y que se produzca código objeto paralelo como se ilustra en la figura 1.6. Este enfoque es aplicable en la programación de sistemas paralelos con "pocos" procesadores.

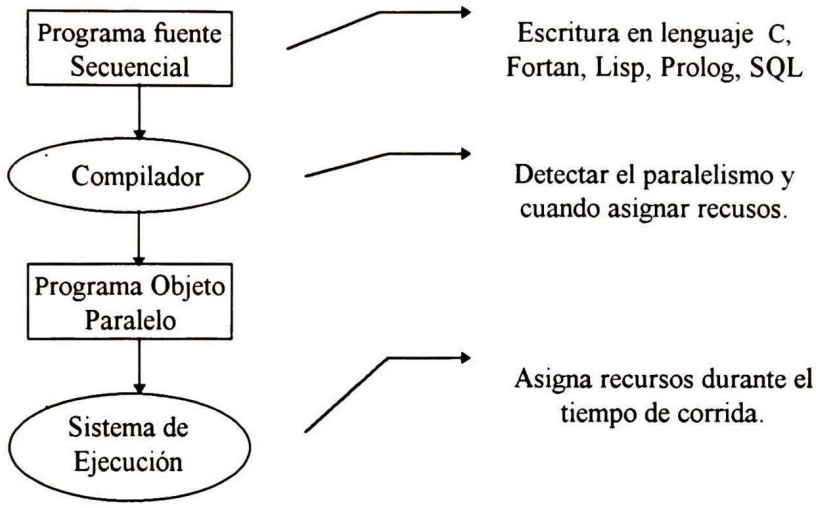


Figura 1.6 *Compilador que detecta el paralelismo*

En general, para la programación de computadoras paralelas, existen al menos cinco grandes estilos de lenguajes de programación que pueden ser usados. Estos estilos son:

Programación Procedural.- Esta basada en los siguientes conceptos: memoria global, acciones básicas como asignaciones, estructuras de control secuenciales implícitas para la ejecución de instrucciones. Existen dos clases de lenguajes procedurales, los llamados lenguajes secuenciales convencionales (como C, Fortran o Basic), y los lenguajes concurrentes que cuentan con estructuras de control paralelas (ADA , OCAM). Este estilo de programación se revisara más adelante ya que es el que se utiliza en el desarrollo de este trabajo.

Programación Orientada a Objetos.- Esta basada en objetos que se comunican por medio de pasos de mensajes. Todos los objetos pertenecen conceptualmente a una clase que define sus propiedades. En este estilo se reduce la distinción entre datos y procedimientos, ya que estos elementos están especificados en los objetos del sistema. Ejemplos de lenguajes secuenciales orientados a objetos son Smalltalk y C++ [Stroustrup 86], mientras que de lenguajes paralelos son C++ Concurrente en [Olmedo 87] y OB-OCAM[5]en [Hernandez R. 95].

[5]El leguaje OB-OCAM es una extensión al lenguaje al lenguaje de programación para trasputers OCAM™, y consiste en incorporar objetos a OCAM™, para obtener un leguaje basado en objetos.

Programación Funcional.- Opera tomando a un programa como una colección de funciones. Donde a una función se le aplica un argumento de entrada y se obtiene valores de salida, este estilo libre de los efectos y restricciones de la operaciones de asignación, contribuye a obtener un modelo semántico más puro. Dos clase importantes de lenguaje funcionales son: los lenguajes aplicativos (como Lisp) y los lenguajes de asignación única que son diseñados, para facilitar la programación de computadoras de flujo de datos[6], se puede ver una revisión de los lenguajes de flujo de datos en [Whiting 94].

Programación Lógica.- Se caracteriza por la tesis *computacion = demostracion*. El lenguaje Prolog es el lenguaje predominante en este estilo y esta basado en las cláusulas de Horn. Las cláusulas de Horn son un subconjunto del calculo de predicados, donde los conceptos básico son: las instrucciones son relaciones de una forma restringida y la ejecución es controlada a través de la deducción lógica de las instrucciones, en [Tick 91] y en [Takeuchi 92] se presentan las principales caracteristicas de la programacion logica paralela así como el paralelismo AND y OR.

Programación Basada en Conocimiento.- Este estilo incluye a los lenguajes desarrollados por formalismos de representación de conocimiento específicos. Los lenguajes basados en conocimiento son los relacionados con la representación del conocimiento humano y los mecanismos de inferencia de este conocimiento. Dentro de estos lenguajes se incluyen los lenguajes de sistemas de producción y los de redes semánticas, se puede ver en [Tuthill 90] un buena exposicion de las caracteristicas de los sistemas basados en conocimiento.

Después de esta breve revisión de los estilos de programación revisaremos más a detalle el estilo procedural, que es la categoría más predominante.

La programación procedural se divide en dos categorías:

Lenguajes Convencionales.- Esta clase de lenguajes corresponden a la única clase de lenguajes de programación, que la mayoría de los usuarios de las computadoras conocen. Esta categoría ha sido desarrollada para la programación de computadoras de tipo Von Neumann. La semántica de los lenguajes convencionales reflejan las características del modelo de programación Von Neumann que son: memoria global, un arreglo fijo de celdas de memoria, asignación y ejecución secuencial.

Lenguajes Concurrentes.- Extienden este modelo de programación de flujo de control con estructuras de control paralelas basadas en procesos, adicionando mecanismos de comunicación y sincronización. Un proceso es un programa independiente y consiste de una estructura de datos privada y código secuencial que puede operar en los datos. Los procesos se ejecutan de manera concurrente operando en sus propios datos y solo interactuan entre si utilizando los mecanismos de comunicación y sincronización. Los mecanismos de comunicación es la manera en que los procesos comunican datos entre ellos;

---

[6]Actualmente se esta realizando en la sección de computación del CINVESTAV, el trabajo de doctorado "Arquitecturas orientadas a flujo de datos" por Rafael Almaraz Rodríguez.

los mecanismos más comúnmente usados son: pasos de mensajes, memoria compartida global no protegida, memoria compartida protegida por módulos o monitores y rendezvous. Los mecanismos de sincronización es la manera en que los procesadores efectúan las restricciones en la secuenciacion de ellos mismos. Los mecanismos comúnmente usados incluyen: señales, mensaje de sincronización, memoria auxiliar, eventos semáforos, condiciones, colas y regiones criticas, etc. En general los lenguajes concurrentes pueden ser clasificados por la naturaleza de sus mecanismos de comunicación en: paso de mensajes y memoria compartida.

En particular el enfoque realizado en esta tesis es en programación procedural en la categoría de lenguajes concurrente con mecanismo de comunicación de paso de mensajes; en particular en el estilo SCMD (del inglés same-code/multiple-data) para la utilización eficiente de computadoras de tipo hipercubo. Este estilo consiste en distribuir los datos en la memoria de cada procesador y ejecutar el mismo programa en cada procesador. Los programas de tipo SCMD consisten en bloques de código con llamadas a rutinas de paso de mensajes; los procesadores ejecutan de manera asincrona el mismo bloque de código utilizando los datos disponibles en su memoria local; entonces se intercambian datos entre los procesadores; este punto de comunicación sirve también como una función de sincronización.

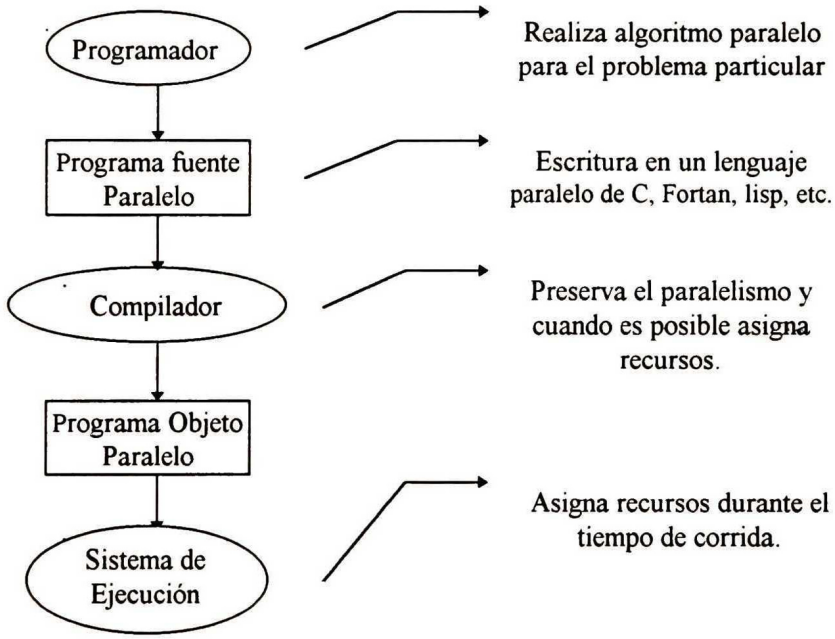


Figura 1.7 *Diseño de algoritmos paralelos*

La metodología que se seguirá en la implantación de algoritmos en este trabajo, consiste en el mejor aprovechamiento de sistemas que cuentan con "muchos" procesadores, donde se busca desarrollar algoritmos que se adapten a las características físicas de la computadora en particular, esta metodología se puede observar en la figura 1.7. y consiste primero en realizar el algoritmo paralelo del problema que se desea atacar[7], en el modelo SCMD, se realiza la codificación en un leguaje paralelo[8]para obtener un código fuente, después se compila con un compilador paralelo con lo que se obtiene código objeto paralelo el que finalmente se puede ejecutar en una computadora paralela.

En las nuevas aplicaciones serán de sumo interés el costo y la rápidez de los sistemas es decir el hardware no debe ser prohibitivo por su precio y los cálculos deben terminar en un tiempo aceptable para cada aplicación en particular, como se indica en [Bertsekas 89]. Podemos decir que el desarrollo de software paralelos estará guiado por un lado, por la relación entre las nuevas y viejas necesidades de computo y por otro, por los progresos tecnológicos.

1.3.1 Complejidad paralela

Existe una gran cantidad de problemas posibles de estudio, además de los tratados en esta tesis, de los cuales algunos parecen más o menos dispuestos para el procesamiento paralelo, aun contando con la presencia de un gran número de procesadores. Un punto interesante es clasificar a los problemas de acuerdo a una noción bien definida de paralelizabilidad. Esta noción es complicada ya que debemos considerar dos parámetros en forma simultánea: tiempo y número de procesadores.

Las clases de complejidad son típicamente definidas en términos de lenguajes. Un lenguaje L, es un subconjunto de cadenas definidas sobre un alfabeto $\{0,1\}$. Un algoritmo de decisión que reconoce al lenguaje L, primero toma una cadena arbitraria $x \in \{0,1\}^*$ y entonces determina cuando x esta o no en L. El problema de reconocer un lenguaje es un problema de decisión y su respuesta es si o no. Nos enfocaremos en los problemas de reconocimiento de lenguajes ya que la teoría es más simple y los detalles omitidos son superfluos a la teoría además que se pueden asociar fácilmente los problemas de decisión con cualquier problema de computación. En general un problema de computación es tan difícil de resolver como lo es cualquier problema de decisión asociado con el, por lo que la evidencia que muestre que un problema de decisión no es paralelizable puede ser usada para mostrar que el problema de computación asociado no es paralelizable.

---

[7]En nuestro caso los problemas de ordenamiento, árbol de extensión mínimal y el problema del vendedor viajero

[8]La computadora NCUBE 2 que se utilizo cuenta con un leguaje C con extensiones para soportar paralelismo, por lo que se codificaron los algoritmos en este lenguaje

1.3.1.1 Las Clases *P* y *NC*

Sea *P* la clase de todos los lenguajes que pueden ser reconocidos por una maquina de turing determinista, utilizando un número polinomial de pasos, como se enuncia en [Schneier 94] y en [Cormen 92]. La clase *P* puede ser definida como la clase de todos los problemas que pueden ser resueltos en tiempo polinomial en una computadora secuencial, o también se acepta como la clase que representa a los problemas que pueden ser resueltos por un procesador secuencial de una forma eficiente. La clase *NC* es una clase de complejidad paralela que se puede definir como la clase de los lenguajes *L*, tal que en todas las entradas de tamaño *n*, el lenguaje *L* puede ser reconocido con un algoritmo *PRAM*[9] en un tiempo de corrida polilogarítmico $log^c(n)$, utilizando un número polinomial de procesadores, donde *c* es una constante, que no depende de *n* y además es la potencia del logaritmo. La idea es que el número de procesadores sea "pequeño", en la práctica se quiere que más que polinomial el número de procesadores sea lineal o casi lineal. El nombre *NC* significa "Nick's Class" después que Nick Pippenger definió esta clase en 1979, dicha clase no se definió para *PRAMs*, el definió la clase *NC* para circuitos eléctricos, los que también son modelos de computación paralela por lo que es aplicable en ambos casos. También se dice que la clase *NC* es la clase de problemas eficientemente paralelizables

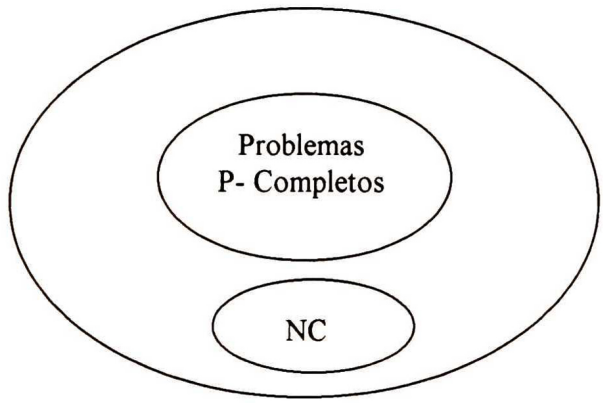


Figura 1.6 *La estructura de P*

[9] *Computadora con Acceso Paralelo a Memoria* (*PRAM* de sus siglas en inglés), es un modelo clásico para el diseño de algoritmos paralelos y consiste en varios procesadores compartiendo un espacio de memoria común, este modelo se subdivide en base a la forma en que se realizan accesos concurrentes a la misma localidad de memoria, por otra parte este modelo tiene la desventaja que no incluye variables como topología de red, sincronización, contención de memoria y latencia, ver [Goodrich 93].

Existe una relación interesante entre las clases P y NC, por un lado se observa que $NC \subseteq P$ convirtiendo los algoritmos NC en secuenciales, esto se realiza en forma obvia, un problema en la teoría de la complejidad es si es cierto o no que $P \subseteq NC$, como se menciona en [Jaja 92], la suposición más aceptada es que $P \not\subset NC$ y por lo tanto existen problemas en P que no pueden ser resueltos eficientemente en paralelo con un número polinomial de procesadores, ver figura 1.6. La clase de los problemas *P-Completos* consiste en los candidatos en P que aparentemente no están en NC. Como en el caso de los problemas *NP-Completos* la noción de reducibilidad es de gran importancia.

La noción de reducibilidad.- Sea L_1 y L_2 dos lenguajes, el lenguaje L_1 es *NC-Reducible* al lenguaje L_2 si existe un algoritmo que transforme una entrada arbitraria u_1 para L_1 en una entrada u_2 para L_2 tal que $u_1 \in L_1$ si y sólo si $u_2 \in L_2$. Se nota que la noción de reducibilidad no es simétrica y que una instancia arbitraria de L_1 es transformada en una instancia especial de L_2. La existencia de las transformaciones NC implican que un algoritmo para resolver L_1 se puede obtener dando un algoritmo para reconocer L_2. En particular tenemos los siguientes lemas:

Lema 1.1.- Sean L_1 y L_2 dos lenguajes tal que L_1 es *NC-Reducible* a L_2, entonces $L_2 \in NC \rightarrow L_1 \in NC$.

Lema 1.2.- Sean L_1 y L_2 dos lenguajes tal que L_1 es *NC-Reducible* a L_2 y L_2 es *NC-Reducible* a L_3 entonces L_1 es *NC-Reducible* a L_3; esto es, la *NC-Reducibilidad* es transitiva.

La demostración de los lemas 1.1 y 1.2, se pueden ver en [Jaja 92 pp. 529-561]

Noción de *P-completitud.*- Un lenguaje L es *P-Completo* si $L \in P$ y cualquier lenguaje en P es *NC-Reducible* a L. De esta definición se deduce los siguiente.

Lema 1.3.- Sea L un problema *P-Completo*. Si $L \in NC$ entonces $NC = P$.

Este lema puede ser enunciado de la siguiente forma: Si $P \neq NC$, que es una conjetura muy fuerte, entonces ningún problema *P-Completo* pertenece a NC. Por lo que un problema *P-Completo* representa a un candidato en P que no puede ser resuelto rápidamente con una *PRAM* con un número polinomial de procesadores.

Para ver que un problema es *P-Completo* la prueba tiene que mostrar que un lenguaje arbitrario $L \in P$ es *NC-Reducible* a este problema.

El problema de evaluación de un circuito (*PEC*).- El problema de evaluación de un circuito (*PEC*) consiste en determinar el valor de salida de un circuito booleano que consiste de los elementos NOT, AND y OR, para un conjunto de entradas. El circuito C puede ser

especificado por una secuencia $C = <g_1, g_2, ..., g_n>$, donde cada g_i es una entrada y además $g_i = g_j \vee g_k$, o $g_i = g_j \wedge g_k$,o $g_i = \neg g_j$, donde $j,k < i$

El PEC se plantea de la siguiente forma: Dado in circuito booleano C representado por la secuencia $C = < g_1, g_2, ..., g_n >$, y con un conjunto dado de entradas determinar cuando el valor del circuito es igual a uno.

Teorema 1.1.- El *PEC* es *P-Completo*.

Demostración : se pueden ver en [Jaja 92 pp. 529-561] y en [Goldschlager 77].

Sabiendo que el PEC *es P-Completo* la tarea de establecer la *P-Completitud* de problemas adicionales es más fácil por el siguiente hecho.

Lema 1.4.- Sea L un lenguaje que es *P-Completo.* Si L es *NC-Reducible* a otro $L' \in P$ entonces L' es también *P-Completo*.

En [Jaja 92 pp. 540-552] se demuestra que el problema de flujo máximo en una red, búsqueda ordenada en profundidad y desigualdades lineales pertenecen a la clase de problemas P-Completos.

Capítulo 2

Ordenamiento en paralelo

El objetivo de este capítulo es mostrar la forma de resolver el problema de ordenamiento desde un enfoque paralelo. En la sección 2.1 Ordenamiento en paralelo, se plantea en que consiste el ordenamiento de una lista de objetos y además se clasifica a los algoritmos de ordenamiento paralelo en algoritmos basados en mezclas y basados en particiónes; en 2.2 Ordenamiento Paralelo con Quicksort y Mergesort (OPQM) se comentan las características del algoritmo OPQM; en 2.2.1 Algoritmo OPQM se describen las características del algoritmo[10], como funciona, como se mapea a un hipercubo y se da un ejemplo, en 2.2.2 Complejidad se analiza la complejidad del algoritmo, que nos dice como es el comportamiento del algoritmo en tiempo; en 2.2.3 Resultados experimentales, se analizan las pruebas realizadas en la computadora NCUBE 2 en base a la aceleración y eficiencia por medio de las gráficas respectivas; en 2.3 Ordenamiento Paralelo por Muestreo Regular (OPMR), se comentan las características del algoritmo OPMR de la misma forma en que se reviso al algoritmo OPQM, lo mismo sucede en las secciones 2.3.1 Algoritmo OPMR, en 2.3.2 complejidad y en 2.3.3 Resultados experimentales; por último, en 2.4 Conclusiones se analizan las características de los algoritmos de ordenamiento presentados, así como se incluyen el análisis comparativo de los algoritmos paralelos contra una computadora secuencial de alto desempeño[11]

2.1 Ordenamiento en paralelo

El ordenamiento es uno de los problemas más estudiados en la ciencia de la computación, por ser de interés teórico y de importancia práctica. Con la llegada del procesamiento en paralelo, el ordenamiento en paralelo se ha convertido en un área importante para el desarrollo de algoritmos. En este capítulo nos enfocaremos en los algoritmos de ordenamiento que se llevan acabo en la memoria principal de los procesadores, no se incluyen los algoritmos que utilizan memoria secundaria como discos o cintas. También se asume que los objetos a ordenar son registros que consisten de uno o mas campos; donde un campo es la llave y para el tipo de esta llave se define una relación de orden lineal ($\leq$). Los números enteros, reales y las cadenas de carácter son ejemplos de posibles tipos de llaves, de cualquier forma se puede generalizar el uso de cualquier tipo de llave para la que se definan las relaciones de orden "menor que" o "menor igual que".

El problema de ordenamiento consiste en modificar la secuencia de los registros de tal forma que los campos llave formen una secuencia en base al orden definido.

---

[10]Para la descripcion de los algoritmos en el capítulo 2 y 3, se utiliza el estilo propuesto por Donald E. Knuth en [Knut 84], y retomado en [Bentley 86], llamado programación literaria.

[11]La computadora secuencial que se utilizo es la computadora DEC-ALPHA™, que es de tipo conjunto reducido de instrucciones (RISC de sus siglas en inglés), que trabaja a 40 MFLOPS.

Se sabe que un algoritmo de ordenamiento secuencial que use comparaciones binarias ordena n elementos en un tiempo $O((n) \log (n))$ [Knuth 73]. Por lo tanto un algoritmo de ordenamiento en paralelo óptimo con p procesadores debería ordenar n elementos en un tiempo $O(((n) \log (n))/p)$. El problema relativo a la rápidez de un ordenamiento en paralelo consiste en minimizar los tiempos de acceso a memoria, transferencia de datos y sincronización de los procesadores. En base a estrategias generales [Shi 92], la mayoría de los algoritmos paralelos de ordenamiento disponibles para computadoras paralelas pueden dividirse en dos categorías: los ordenamientos basados en mezclas y los ordenamientos basados en particiónes.

- Los ordenamientos basados en mezclas, ordenan los datos en base a mezclar las listas de datos en los procesadores y consisten primero en lograr un orden local, es decir cada procesador ordenará una sublista y después mediante múltiples mezclas que llevan acabo los procesadores, se logra ordenar todos los datos. Su desempeño es bueno, sólo con un número pequeño de procesadores, cuando son muchos procesadores los tiempos de sincronización, transferencia de información y administración del sistema reducen la rapidez del algoritmo.

- Los ordenamientos basados en particiónes, ordenan los datos en base a particiónes de datos que intentan distribuir lo mejor posible a los datos en subconjuntos ordenados, estos ordenamientos constan de tres etapas. Primero un orden local, después se particióna el conjunto de datos de tal forma que todos los elementos de un subconjunto sean mayores que cualquier elemento de otro subconjunto esto es que cualquier elemento en el procesador i es mayor que cualquier elemento en el procesador j siempre y cuando $i>j$ donde i,j son los identificadores de los procesadores y por último cada procesador ordena un subconjunto en paralelo. El desempeño de este algoritmo depende de que tan bien fueron particiónados los datos en la etapa dos .

En este capítulo se presenta la implantación de dos algoritmos de ordenamiento paralelo, el denominado Ordenamiento Paralelo con Quicksort y Mergesort, que se describe en [Loots 92] una versión para transputers, este algoritmo es del tipo de algoritmos basados en mezclas, que tiene complejidad $O((n/p) \log (n)) + O(n)$ y el Ordenamiento Paralelo por Muestreo Regular, que se describe en [Shi 92] y [Abali 93], que es del tipo de algoritmos basados en particiónes, que tiene complejidad $O((n/p) \log (n)) + O((n/p) \log (p))$, ambos algoritmos se implantaron en la computadora paralela con arquitectura *Hipercubo* NCUBE 2.

2.2 Ordenamiento Paralelo con Quicksort y Mergesort (OPQM)

El algoritmo OPQM que se implantó está basado en mezclas, y es una combinación de los algoritmos de ordenamiento Quicksort y Mergesort, los cuales son algoritmos del tipo divide y vencerás [Knuth 73], el algoritmo consiste de dos etapas. La primera, en que cada procesador ordene n/p elementos logrando un orden local, esto utilizando el algoritmo de orenamiento Quicksort. Después la segunda etapa consiste de que los procesadores realicen múltiples mezclas para obtener un orden global, esto mediante el algoritmo Mergesort, que

realiza las mezclas en paralelo. El algoritmo mapea el problema a una arquitectura hipercubo.

2.2.1 Algoritmo OPQM

1. Algoritmo OPQM, para el procesador i (i). Este algoritmo ordena un lista de datos en una computadora hipercubo k dimensiónal; es decir, se puede utilizar con el número de procesadores disponibles en la computadora en que se implante. Una implantación en *trasputers* de este algoritmo se pude revisar en [Loots 92].
variables utilizadas:

n, es el número de elementos a ordenar
k, es la dimensión del hipercubo
p, es el número de procesadores en el hipercubo y $p = 2^k$
i, es el identificador de cada procesador, donde $0 \leq i \leq p-1$
r, es el identificador del procesador fuente o destino, donde $0 \leq r \leq p-1$
s, nos indica la dimensión de hipercubo actual, o el nivel en el árbol de mezcla
Tab, es el arreglo que contiene los elementos a ordenar

2. Estructura de datos. La entrada de este programa es una lista de datos *Tab*, distribuida en los *p* procesadores, la salida será una lista de datos *Tab* que será depositada en el procesador 0.
Entrada :

Arreglo *Tab*[*0..n*], sea *w=n/p* y el subarreglo *Tab*[*i*w..(i+1)*w-1*], está almacenado en la memoria local del procesador *i* del hipercubo con dimensión *k*.

Salida :

Arreglo Tab[*0..n*] datos ordenados en el procesador *i*, donde *i = 0*.

3. Elementos del algoritmo. Aquí se mostrara los elementos y/o subrutinas del procedimiento llamado OPQM:

```
void OPQM (){
  /* (4. Primera etapa)*/
  quicksort();
  /* (5. Segunda etapa)*/
  mezcla_paralela();
  if (pn==0)
    resultado();
}
```

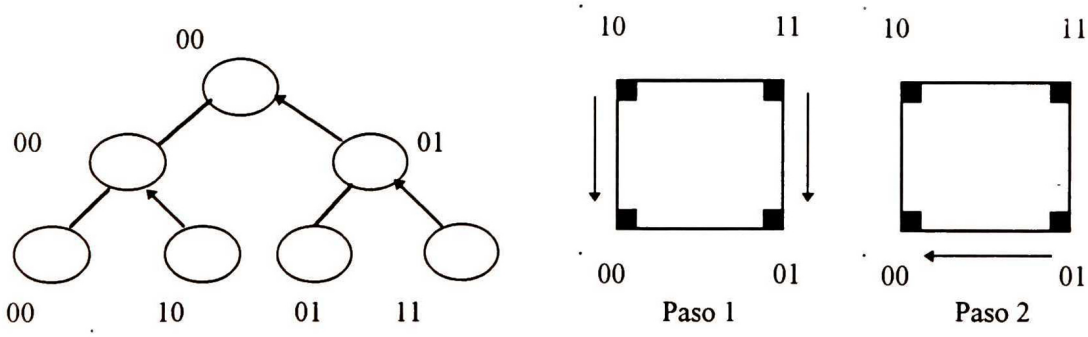
4. Primera etapa. Esta etapa consiste en que cada procesador ordene una sublista secuencial de longitud *w*, que estará en la memoria local del algoritmo, el ordenamiento local se realiza, utilizando el algoritmo de ordenamiento secuencial más rápido conocido (se utilizó el **quicksort**()), durante esta etapa los *p* procesadores se encuentran trabajando.

*5. **Segunda etapa.*** En esta última etapa se realizan k transmisiones de datos en paralelo y también k mezclas paralelas, en general durante el paso s, donde $0 \leq s < k$ y se comienza con $s = k-1$, el procesador i envía o recibe una lista de longitud $n/2^{s+1}$ al procesador r, el procesador que recibe la lista es aquel que tenga el identificador con número menor, y después mezcla la lista que recibe y la propia, las dos listas tienen longitud $n/2^{s+1}$ Se calcula r realizando la operación Or - exclusiva ($\oplus$) entre el identificador i y 2^s(se obtiene realizando un corrimiento de s bits). En cada paso s se elimina la mitad de los procesadores, esto sucede hasta que el resultado de la mezcla se concentra en el procesador 0, en la figura 2.1, se muestra en el inciso a), como se construye el árbol de mezcla desde que se utilizan los p procesadores hasta que se concentra el resultado en el procesador 0, además se puede ver como se mapea el árbol de mezcla al hipercubo k dimensiónal que se utilice, en la figura se considera que la dimensión de hipercubo $k = 2$.

```
mezcla_paralela ();{
for( s = k-1 ; s>=0 ; s--)
   if( (r = i ⊕ 1<< s)< i) {
      manda_datos(); /* el procesador i manda una lista al procesador r */
      return;
   }else{
      recibe_datos(); /* el procesador i recibe una lista del procesador r */
}
      mezcla(Tab[ ], TabAux[ ] ); /* el procesador i mezcla la lista recibida con la
                                     propia */
}
```

Despues de la **mezcla_paralela**(), la lista ordenada se encuentra en el procesador 0.

```
if (pn==0)
    resultado();
```

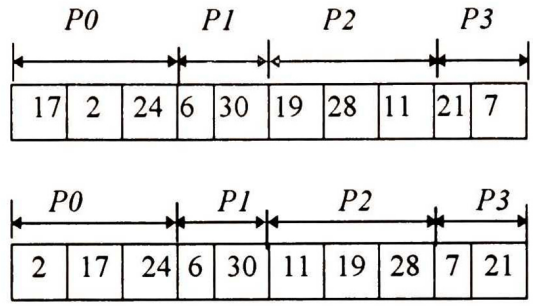


a) árbol de mezcla

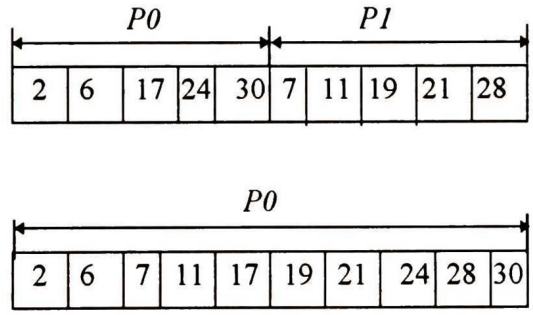
b) árbol de mezcla en el hipercubo

Figura 2.1 *Ejemplo de mezcla paralela del OPQM, k=2*

A continuación se verá un ejemplo de como funciona el algoritmo OPQM, para esto se utilizara una serie de número enteros no ordenada (17, 2, 24, 6, 30, 19, 28, 11, 21, 7) y se ordenará utilizando un hipercubo con dimesion k=2, es decir con cuatro procesadores, $p = 4 = 2^k$. Primero, se dividen los datos de manera que cada procesador tenga aproximadamente el mismo número de datos, los subconjuntos de datos distribuidos deben ser disjuntos, después se efectúa el ordenamiento local en cada procesador, esto es en ***4. Primera etapa***, ver figura 2.2 inciso a). Ahora cada procesador contará con una lista ordena y mezclara los datos intercambiando los datos entre los procesadores como se revisa en ***5. Segunda etapa***, ver figura 2.2 inciso b). Al final la lista ordenada se encuentra en el procesador 0 y es (2, 6, 7, 11, 17, 19, 21, 24, 28, 30).



a) Primera etapa, **quicksort**() Secuencial



b) Segunda etapa, **mezcla_paralela**()

Figura 2.1 *Ejemplo del algoritmo OPQM, k=2,* consta de dos etapas

2.2.2 Complejidad.

La primera etapa de ordenamiento consiste en que cada procesador ordene una sublista de longitud n/p. El promedio de comparaciones realizadas por el algoritmo **quiksort**(), para ordenar n elementos es alrededor de $O((n)\log(n))$ [Knuth 73], ordenar n/p elementos, considerando a p fijo y con un valor $p \leq \log(n)$, tiene el siguiente comportamiento:

$$\begin{aligned} Quick\_comparaciones &= (n/p)\log(n/p) \\ &= (n/p)(\log(n) - \log(p)) \\ &= (n/p)\log(n) - (n/p\log(p)) \end{aligned}$$

ya que $p = 2^k$ por la definición de hipercubo

$$= (n/p)\log(n) - nk/p$$

La segunda etapa de **mezcla_paralela**(), consiste en combinar las sublistas ordenadas en la etapa anterior y que cada procesador obtenga una nueva lista ordenada, concentrando el resultado en el procesador cero. Para mezclar dos listas de longitud 2^l toma a lo más $2^{l+1}-1$ comparaciones. Durante el primer paso el algoritmo mezcla dos listas de longitud n/p, durante el segundo paso se mezclan dos listas de longitud $2n/p$, durante el paso k-*ésimo* se mezclan dos listas de longitud $2^{k-1}n/p$. El total de comparaciones durante la etapa de mezcla es igual a:

$$\begin{aligned} Mezcla\_comparaciones &= 2n/p-1 + 2(2n/p)-1 + \ldots + 2(2^{k-1}n/p)-1 \\ &= 2n/p + 2(2n/p) + \ldots + 2(2^{k-1}n/p) - k \\ &= 2(n/p)\left(\sum_{i=0}^{k-1} 2^i\right) - k \;;\; 2(n/p)(2^k-1)-k \\ &= 2n(p-1)/p-k \\ &= 2n - 2n/p - k \end{aligned}$$

El tiempo de comunicación durante la etapa de mezcla paralela puede ser un factor que limite la velocidad. Se asume que el tiempo de comunicación entre procesadores consta de dos partes: El tiempo de inicialización TI y el tiempo de transferencia de un dato TD, estos tiempos los realizan los procesadores que intercambien datos. El tiempo total de comunicación del algoritmo se estima de la siguiente forma:

$$Tiempo\_comunicaciones = \sum_{i=1}^{k} \left(TI + TD\,\frac{n}{2^i}\right)$$

$$= TI\,k + TD\left(\sum_{i=1}^{k} \frac{n}{2^i}\right) \qquad \text{-a)}$$

Sea $S_k = \sum_{i=1}^{k} \frac{n}{2^i} = (n - \frac{n}{2^k})$

Demostración:

caso base $k=1$

$$S_k = \sum_{i=1}^{1} \frac{n}{2^1} = \frac{n}{2} ; \quad (n - \frac{n}{2^k}) = n - \frac{n}{2} = \frac{n}{2}$$

se tiene como hipótesis de inducción para $k=m$

$$S_k = \sum_{i=1}^{k} \frac{n}{2^i} = (n - \frac{n}{2^k})$$

se quiere mostrar que se cumple para $k=m+1$

$$S_{k+1} = S_k + \frac{n}{2^{k+1}} = (n - \frac{n}{2^k}) + \frac{n}{2^{k+1}} = n - \frac{n}{2^{k+1}} \quad \text{lqd.}$$

Por lo que se demuestra que la siguiente igualdad se cumple.

$$\sum_{i=1}^{k} \frac{n}{2^i} = (n - \frac{n}{2^k})$$

Sustituyendo esta igualdad en -a).

$$Tiempo\_comunicaciones = TI\, k + TD\ (n - \frac{n}{2^k})$$

De lo anterior observamos que el número de datos a transmitir depende linealmente de n. Ya que con k constante o fija, $O(nk) = O(n)$. Por lo tanto de la primera etapa de ordenamiento local requiere $O((n/p)\log(p))$, mientras la segunda etapa de mezcla requiere $O(n)$ y la comunicación entre procesadore requiere $O(n)$, entonces la complejidad es:

$$\begin{aligned} tiempo(n) &= O((n/p)\log(n)) + O(n) + O(n) \\ &= O((n/p)\log(n)) + O(n) \end{aligned}$$

2.2.3 Resultados experimentales

El algoritmo OPQM se implantó en una computadora NCUBE con ocho procesadores y se programó en lenguaje C. Fueron generadas series de datos en orden inverso para realizar las pruebas, además se probó con 100000, 200000, 300000, y 400000 datos (números enteros de 4 bytes) y con 1, 2, 4, y 8 procesadores (hipercubo de dimensión $k=0, 1, 2,$ y 3) en una computadora dedicada. Los resultados se ilustran en la figura 2.3, y no incluyen el tiempo para distribuir los datos en los procesadores. Se observa que en términos generales el comportamiento del algoritmo es bueno ya que efectivamente se reduce el tiempo de ejecución conforme aumentan los procesadores. Además el comportamiento del algoritmo es mejor, cuando se aplica a un mayor número de datos

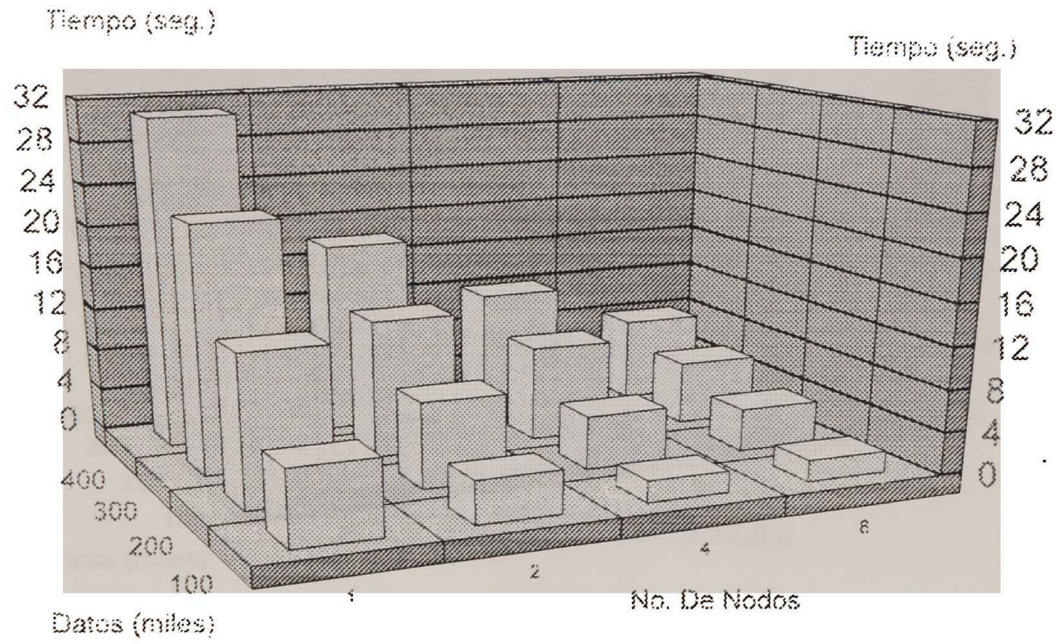


Figura 2.3 *Tiempos de proceso*

Ahora se analiza la eficiencia del algoritmo E(n), como se definió en el capítulo 1.5.

$$E(n) = TS(n)/(pTP(n))$$

Se nota que cuando se aumenta el número de procesadores la eficiencia es menor como lo ilustra la figura 2.4, ésto se debe a las características del algoritmo, que en cada nivel de mezcla se subutilizan la mitad de los procesadores que están activos, es decir, en el último paso del algoritmo cuando se mezclan las dos últimas listas ordenadas, sólo un procesador se encuentra trabajando, además en este último paso, es cuando se tiene que mezclar las listas de mayor longitud $n/2$. para evitar este problema se han propuesto los algoritmos basados en particiónes, que durante todo el algoritmo intentan utilizar al máximo los procesadores, ver [Shi 92] y [Abali 93], estos algoritmos intentan tener trabajando a los procesadores por medio particiónar los datos, el buen desempeño depende de que tan bien fueron particiónados los datos en los procesadores. En la siguiente sección se revisara un algoritmo de ordenamiento basado en particiónes, el algoritmo de ordenamiento paralelo por muestreo regular OPMR.

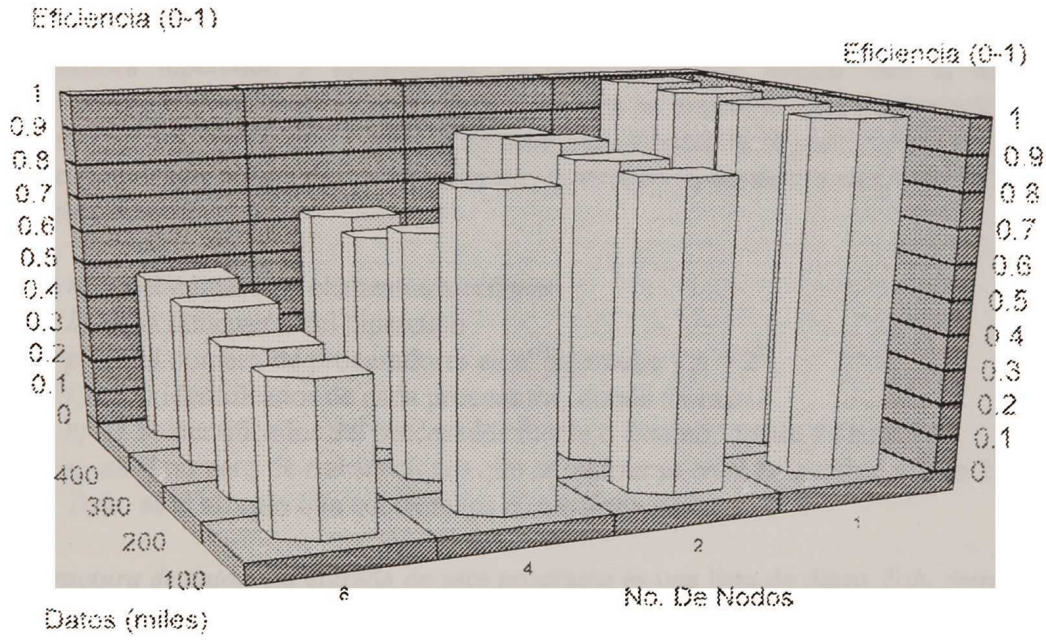


Figura 2.4. Eficiencia. *Por fines de visualización el orden de los procesadores es diferente*

2.3 Ordenamiento Paralelo por Muestreo Regular (OPMR)

El algoritmo OPMR que se implantó está basado en particiónes, y funciona en un hipercubo con k dimensiónes. El algoritmo OPMR consta de tres etapas: en la primera etapa, n elementos distintos son distribuidos sobre los $p=2^k$ procesadores del hipercubo para que cada procesador tenga aproximadamente n/p elementos, despues se obtiene un orden local con el algoritmo Quicksort. En la segunda etapa, el algoritmo intercambia elementos para obtener una lista distribuida sobre los p procesadores donde cualquier elemento en el procesador i es mayor que cualquier elemento en el procesador j siempre y cuando $i>j$, En la última etapa, cada procesador ordena los elementos que le fueron asignados, al final cada procesador es dejado con aproximadamente n/p elementos.

2.3.1 Algoritmo OPMR

1. Algoritmo OPMR, para el procesador i (i). Este algoritmo ordena un lista de datos en una computadora hipercubo k dimensiónal, es decir se puede utilizar con el número de procesadores disponibles en la computadora en que se implante. Una implantación de este algoritmo para computadoras paralelas de memoria compartida se pude revisar en [Shi 92] y una variante del algoritmo para computadoras hipercubo se encuentra en [Abali 93] .
Variables utilizadas:

n, es el número de elementos a ordenar
k, es la dimensión del hipercubo
p, es el número de procesadores en el hipercubo y $p = 2^k$
i, es el identificador de cada procesador, donde $0 \le i \le p-1$
r, es el identificador del procesador fuente o destino, donde $0 \le r \le p-1$
s, es el número de transmiciones que se realizaran en el hipercubo, donde $1 \le s \le p-1$
Tab, es el arreglo que contiene los elementos a ordenar

2. Estructura de datos. La entrada de este programa es una lista de datos *Tab*, distribuida en los p procesadores, la salida será una lista de datos *Tab* que será distribuida en los p procesadores.
Entrada :

Arreglo $Tab[0..n]$, sea $w=n/p$ y el subarreglo $Tab[i*w..(i+1)*w-1]$, está almacenado en la memoria local del procesador i del hipercubo con dimensión k.

Salida :

Arreglo $Tab[0..n]$ datos ordenados en el subarreglo $Tab[i*w..(i+1)*w-1]$.

3. Elementos del algoritmo. Aquí se mostrara los elementos y subrutinas del procedimiento llamado OPMR:

```
void OPMR (){
  /* (4. Primera etapa) */
  quicksort();
  /* (5. Segunda etapa) */
  muestreo();
  intercambio_global();
  /* (6. Tercera etapa) */
  mezclas_multiples();
  resultado();
}
```

4. Primera etapa. Esta etapa consiste en que cada procesador ordene una sublista secuencial de longitud $w=n/p$, que estará en la memoria local del algoritmo, el ordenamiento local se realiza, utilizando el algoritmo de ordenamiento secuencial más rápido conocido (se utilizó el Quicksort), durante esta etapa los p procesadores se encuentran trabajando.

quicksort(); /* ordena n/p datos */

5. Segunda etapa. - Cada procesador selecciona $p-1$ elementos regularmente espaciados por medio de muestras (muestreos regulares), en su lista local ordenada y se la envía al procesador cero, estas particiónes se encuentran por medio de un muestreo regular en los datos locales. Después de recibir los $(p-1)(p-1)$ elementos el procesador cero adiciona sus $p-1$ elementos y ordena los datos y selecciona una vez más $p-1$ elementos, entonces envía estos nuevos pivotes a todos los procesadores. Con estos pivotes cada procesador particióna su lista local en p sublistas, el procedimiento **muestreo**(), realiza estos pasos.

```
muestreo(){
  elige_pivotes();      /* se obtienen pivotes locales por medio de muestreos regulares*/
  obten_pivotes();      /* se envian los pivotes locales para obtener los pivotes globales */
  elige_pivotes();      /* de los pivotes locales se eligen los pivotes globales */
  distribuye_pivotes();      /* distrubuye los pivotes globales a todos los procesadores */
  encuentra_particiónes(); /* cada procesador con los pivotes globales partición sus
                              datos locales */
}
```

Intercambio global. el procesador i ($0 \le i \le p-1$) se queda con la partición i envía la partición j al procesador j, esto se calcula realizando la operación Or - exclusiva ($\oplus$) entre i y s. Por ejemplo el procesador 0 recibe la primera partición de todos los procesadores. Por lo que cada procesador se queda con una partición y reasigna p-1 particiónes, con lo que se obtiene una lista distribuida sobre los p procesadores donde cualquier elemento en el procesador i es mayor que cualquier elemento en el procesador j siempre y cuando $i>j$, con lo que se asegura un orden global en todos los procesadores. Para intercabiar los datos, utilizando el algoritmo de ruteo *e-cube*[12], el procedimiento **intercambio_global**(), genera trayectorias disjuntas para conectar a los procesadores, este procedimiento se puede ver en la figura 2.6

[12]El algoritmo *e-cube* se pude revisar en el capitulo 1.3.2 computadora NCUBE 2.

para un procesador con k=2 dimesiones, se realiza un análisis detallado de este procedimento en [Abali 93].

```
intercambio_global(){
  for(s = 1; s < p ; s++){
    r = i ⊕ s;          /* se obtine la direccion fuente y destino con un Or - exclusivo */
    recibe_datos();     /* el procesador i recibe la lista r-ésima del procesador fuente r */
    manda_datos();      /* el procesador i manda su lista r-ésima al procesador destino r */
    seqstart(i-1,i+1,1);        /* se sincronizan todos los procesadores */
    seqend();
  }
}
```

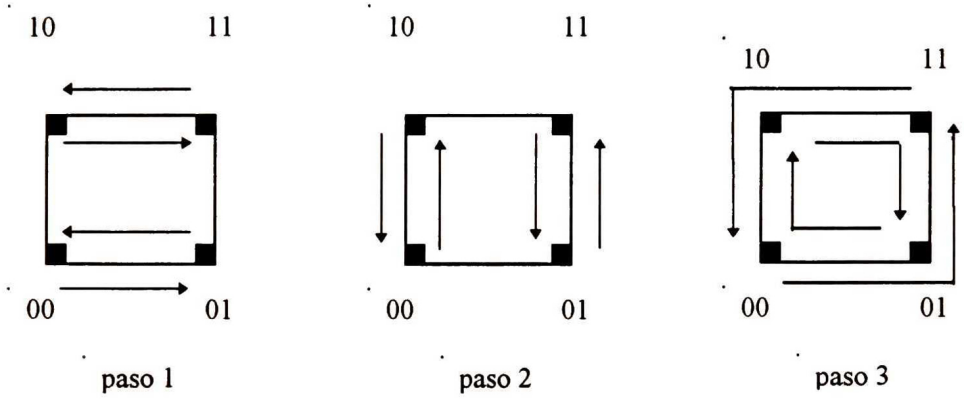


Figura 2.5 *Ejemplo de intercambio_global del OPMR, k=2*

*6. **Tercera etapa.*** - En esta última etapa, después de recibir p-1 sublistas, cada procesador realiza mezclas entre, las sublistas recibidas y la sublista local no enviada, esta etapa se puede comenzar antes de terminar la etapa anterior ya que al tener al menos dos listas ya se pueden comenzar las mezclas, en el procedimiento **mezclas_multiples()** se realizan *p-1* mezclas, para lograr un orden local, que sumado al orden gobal obtenido en ***5. Segunda etapa***, se logra ordenar la lista original.

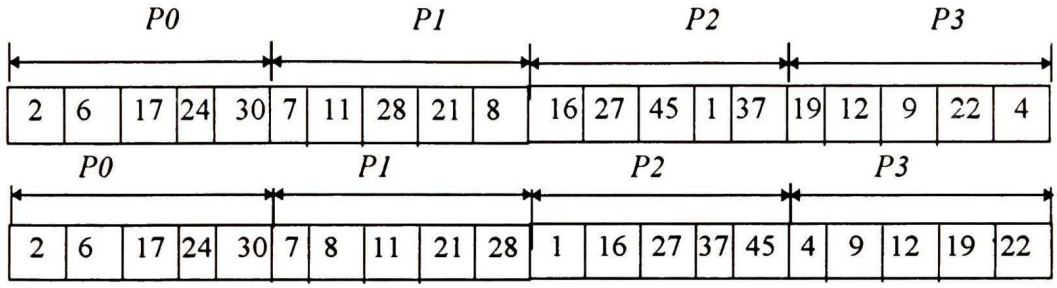
```
mezclas_multiples(){
for(aux = 0 ; aux<nump-1 ; aux++){
  mezcla();      /* se mezclan la lista aux=0, con la lista aux+1 hasta mezclar todas las listas */
}
```

Por último, el resultado se encuentra distribuido en todos los procesadores.

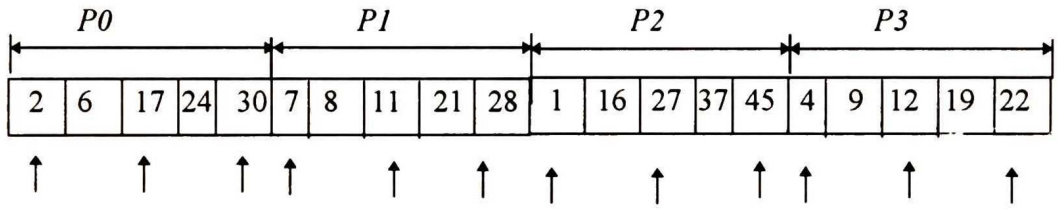
resultado();

A continuación se verá un ejemplo de como funciona el algoritmo OPMR, para esto se utilizara una serie de número enteros no ordenada (2, 6, 17, 24, 30, 7, 11, 28, 21, 8, 16, 27, 45, 1, 37, 19, 12, 9, 22, 4), y se ordenará utilizando un hipercubo con dimesion k=2, es decir con cuatro procesadores, $p = 4 = 2^k$. Primero se dividen los datos de manera que cada

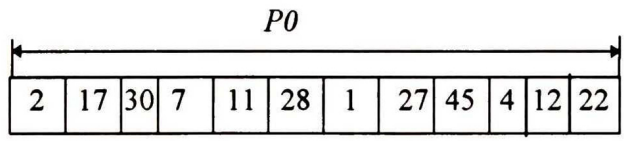
procesador tenga aproximadamente el mismo número de datos, los subconjuntos de datos distribuidos deben ser disjuntos, después se efectúa el ordenamiento local en cada procesador, esto es en ***4. Primera etapa***, ver figura 2.6 inciso a). Ahora cada procesador contará con una lista ordena y con muestreos se particiónan los datos, y se inetrcambian las particiónes entre los procesadores obteniendo un orden global, como se revisa en ***5. Segunda etapa***, ver figura 2.2 inciso b), en la tercera etapa se realiza un orden local por medio de mezclas, como se describe en ***6. Tercera etapa***, ver figura 2.6 inciso c). Al final la lista ordenada se encuentra en los p procesadores y es (1, 2, 4, 6, 7, 8, 9, 11, 12, 16, 17, 19, 21, 22, 24, 27, 28, 30, 37, 45).



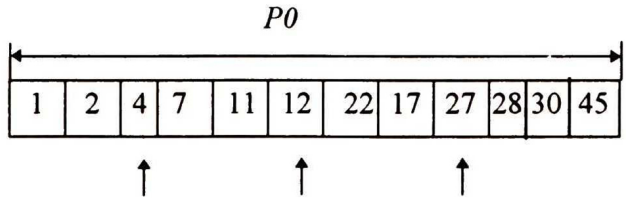
a) Primera etapa, cada procesador ordena con **quicksort()** su lista



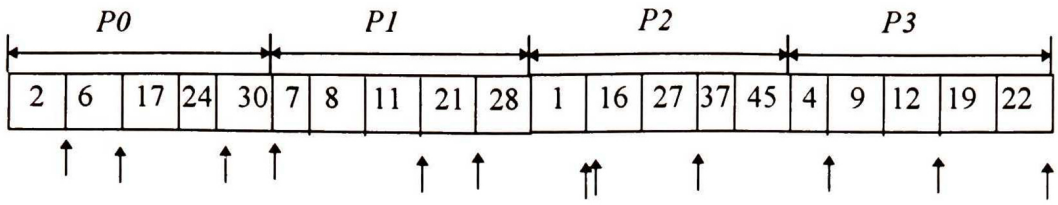
elige_pivotes()



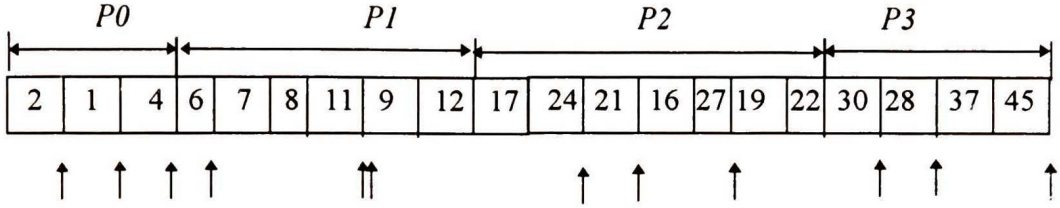
obten_pivotes(), el procesador 0 obtiene los pivotes



elige_pivotes() y **distribuye_pivotes()**

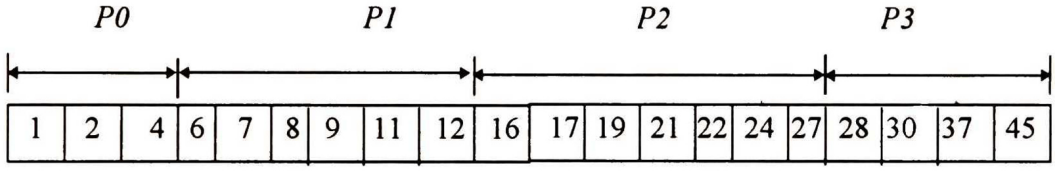


encuentra_particiónes()



Intercambio global()

b) Segunda etapa, que consta de dos procedimientos: **muestreo()** (con cinco subrutinas), e **intercambio_global()**,



c) Tercera etapa, **multiples_mezclas()**

Figura 2.6 *Ejemplo del algoritmo OPMR, k=2, consta de tres etapas*

2.3.2 Complejidad

La primera etapa del ordenamiento consiste en que cada procesador ordene una sublista de longitud n/p, considerando a p fijo y con un valor $p \leq \log(n)$, al igual que el algoritmo OPQM , por lo tanto podemos decir:

Quick_comparaciones = $(n/p) \log(n) - nk/p$

En ***5. segunda etapa*** , el procedimiento **elige_pivotes()**, por cada procesador se seleccionan p-1 elementos regularmente espaciados, ésto lo realiza en un tiempo $O(p)$, después en **obten_pivotes()**, se envían los $(p-2)^2$ pivotes locales de la muestra al procesador cero, con un tiempo de comunicación $O(p)$, y entonces el procesador cero ordena los pivotes locales en un tiempo $O(p^2 \log(p^2))$, en **elige_pivotes()**, se selecciona una vez más p-

1 pivotes, de los pivotes locales ordenados en la rutina anterior, esto en un tiempo $O(p)$, despues **distribuye_pivotes**(), envía estos nuevos pivotes globales a todos los procesadores con un tiempo de comunicación $O(\log(p))$, entonces en **encuentra_ particiónes**(), con estos pivotes globales cada procesador particióna su lista local en p sublistas, recorriendo una vez la lista local para encontrar los pivotes que particiónan la lista local en p sublistas, ésto se realiza en un tiempo $O(n/p)$, para finalizar la segunda etapa, el algoritmo **intercambio_global**(), utiliza para sicronizar a los procesadores a los procedimientos **seqstart**() y **seqend**(), que son ejecutados en un tiempo $O(\log(p))$, ya que para realizar la sincronización cada nodo envía y recibe un toquen de control, ésto es que hasta que todos los procesadores se sincronicen no se podrán seguir ejecutando los siguientes pasos del algoritmo, cuando los segmentos de datos son aproximadamente iguales ($\cong n/(p^2)$), el algoritmo trabaja en un tiempo $O(n/p + (p)\log(p))$, ya que los procesadores relizaran cada paso del algoritmo **intercambio_global**(), aproximadamente en el mismo tiempo, por lo que la sincronización de los procesadores practicamente no tendra efecto en el tiempo utilizado. Cuando la cantidad de datos sean diferentes, como en el caso que cada procesador tiene un segmento de tamaño n/p y $p-1$ segmentos de tamaño 0, el procedimiento **intercambio_global**(), puede ser serializado en la operación de sincronizar los procesadores, por lo que el límite superior en tiempo para la transmisión de datos es $O(n+(p)\log(p))$, de cualquier forma este es el caso más pesimista.

En la tercera etapa, en cada procesador tiene de manera independiente p segmentos en forma de sublistas ordenadas, el procedimiento **mezclas_multiples**(), mezcla las p sublistas en una sola lista ordenada en un tiempo $O((n/p)\log(p))$ por un árbol de mezclas binario, con lo que se termina el algoritmo OPMR.

El tiempo de comunicación durante la segunda etapa del algorimo se calcula de la siguiente forma. Se asume que el tiempo de comunicación entre procesadores consta de dos partes: El tiempo de inicialización *TI* y el tiempo de transferencia de un dato *TD*, estos tiempos los realizan los procesadores que intercambien datos. El procedimiento **obten_pivotes**(), utiliza un tiempo $O(\log(p))$, **distrubuye_pivotes**(), utiliza $O(\log(p))$, y el procedimieto **intercambio_global**(), utiliza en el caso promedio $O(n/p + (p)\log(p))$. El tiempo total de comunicación del algoritmo se estima de la siguiente forma:

$$\begin{aligned} \textit{Tiempo\_comunicaciones}(n) &= O(n/p + (p)\log(p)) + O(\log(p)) + O(\log(p)) \\ &= O(n/p + (p)\log(p)) \end{aligned}$$

Por lo que la complejidad del algoritmo OPMR es igual a $O((n/p)\log(n)) - O(nk/p)$ de la primera etapa, $O((n/p)\log(p))$ de la tercera etapa y la comunicación requiere $O(n/p+(p)\log(p))$, que es la complejida mayor de la segunda etapa, entonces la complejidad del algoritmo es:

$$\begin{aligned} \text{tiempo}(n) = {} & O((n/p)\log(n)) - O(nk/p) + O((n/p)\log(p)) + O(n/p+(p)\log(p)). \\ & O((n/p)\log(n)) + O((n/p)\log(p)). \end{aligned}$$

2.3.3 Resultados experimentales

El algoritmo OPMR también se implantó en una computadora NCUBE con ocho procesadores y se programó en lenguaje C. Fueron generadas series de datos en orden inverso para realizar las pruebas, además se probó con 100000, 200000, 300000, y 400000 datos (números enteros de 4 bytes) y con 1, 2, 4, y 8 procesadores (hipercubo de dimensión *k=0, 1, 2,* y *3*) en una computadora dedicada. Los resultados se ilustran en la figura 2.7, no incluyen el tiempo para distribuir los datos en los procesadores. Se observa que en términos generales el comportamiento de este algoritmo es mejor que el del algoritmo OPQM, ya que éste algoritmo funciona bien con pocos procesadores y cuando se aumenta el número de procesadores su comportamiento es mejor que el de el algoritmo OPQM, como se observa en las pruebas realizadas.

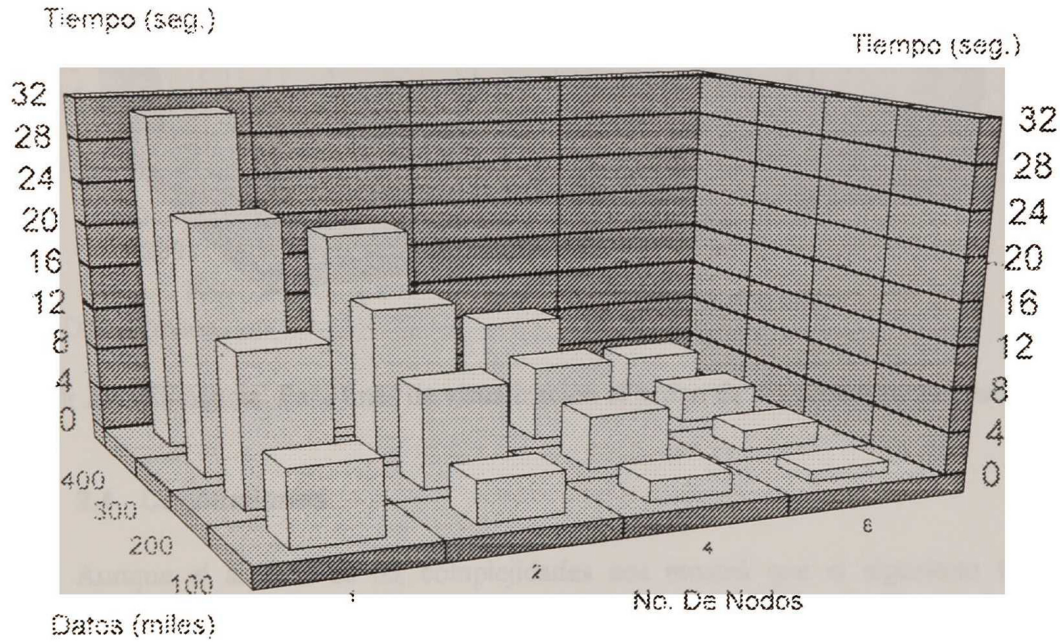


Figura 2.7. *Tiempos de proceso*

Ahora podemos analizar la eficiencia del algoritmo de la misma forma que en el algoritmo anterior del capítulo 2.2.3.

Se nota que cuando va aumentando el número de procesadores la eficiencia es ligeramente menor como lo ilustra la figura 2.8, ésto se debe a las características del algoritmo, que en cada etapa se intenta mantener ocupados los procesadores los más posible y de esta forma cuando aumenta el número de procesadores el tiempo de ejecución se reduce proporcionalmente con los procesadores empleados.

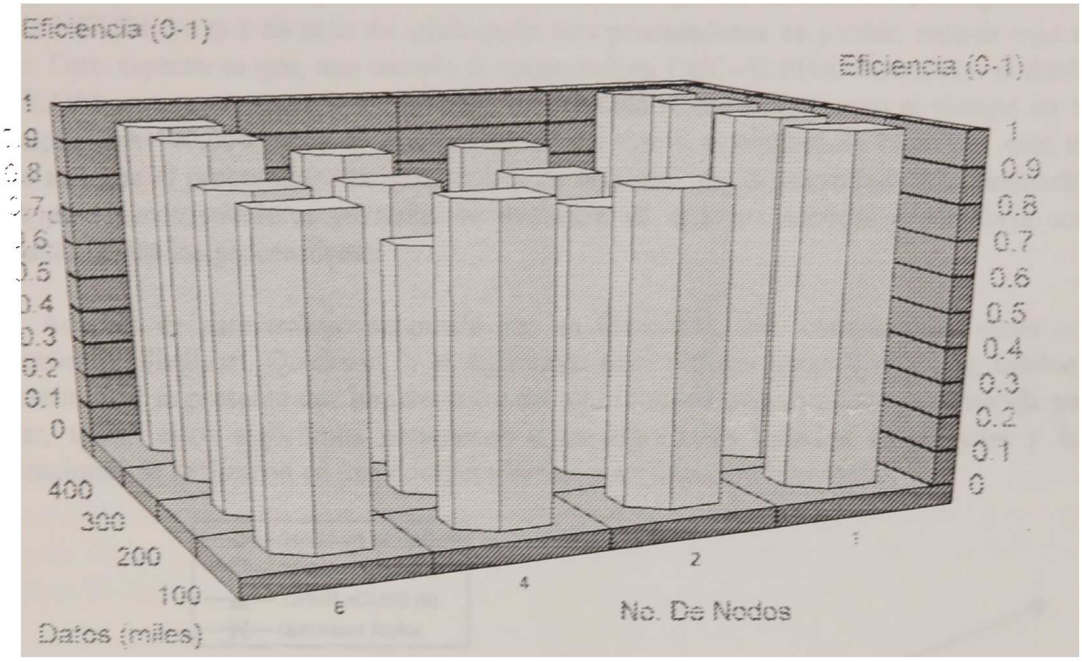


Figura 2.8. Eficiencia. Para fines de visualización el orden de los procesadores es diferente

2.4 Conclusiones

Aunque el análisis de las complejidades nos mostró que el algoritmo OPMR es superior al algoritmo OPQM ya que tiene una complejidad menor, se realizaron las pruebas experimentales que confirmaron el análisis. Ademas se comparó comportamiento de los algoritmos presentados utilizando, series de datos en orden inverso, para realizar las pruebas se probó con 100000, 200000, 300000, y 400000 datos (números enteros de 4 bytes) y con 8 procesadores (hipercubo de dimensión $k= 3$) en una computadora dedicada, se observa en la figura 2.9 que el comportamiento en tiempo del algoritmo OPMR es mejor en tiempo que el del algoritmo OPQM, se utilizaron los ocho procesadores de la computadora en la prueba ya que se desean algoritmos que funcionen lo mejor posible con el máximo número de procesadores, aunque el comportamiento del algoritmo OPMR sea mejor, es claro que no funcionará bien incrementando indefinidamente el número de procesadores ya que tiene un límite, aunque podemos decir que este límite es mayor que el de el algoritmo OPQM, basándonos en los análisis y resultados presentados.

La gráfica de la figura 2.9, también incluye, el tiempo del Quicksort implantado en una computadora DEC-ALPHA (que trabaja a 40 MFLOPS) y en un solo procesador de la NCUBE 2 (2.5 MFLOPS por procesador, con ocho procesadores tiene 20 MFLOPS), se utilizaron los mismos tipos de datos que con los algoritmos anteriores. Los resultados nos indican lo que se esperaba que la computadora DEC-ALPHA proporciona resultados más rápido que cualquiera de los dos algoritmos implantados en la computadora NCUBE utilizando todos los procesadores con los que cuenta, pero se nota que es imposible reducir el tiempo utilizado en la computadora DEC-ALPHA, lo que no sucede en la computadora paralela NCUBE ya que en caso de adicionarle más procesadores es posible reducir más el tiempo. Otro aspecto es que, aun cuando la computadora DEC-ALPHA cuenta con el doble de MFLOPS que la computadora NCUBE, los resultados nos indican que el tiempo en la DEC-alpha con 400,000 datos es ligeramente menor que la computadora NCUBE, esto es porque aun que el procesador sea muy rapido, es detenido por la velocidad de la memoria, lo que en las computadoras paralelas se minimiza ya que los accesos a memoria son distribuidos entre los procesadores.

Además de este trabajo se puede ver en [Fox 88], una comparación entre los ordenamientos Shellsort, Quicksort y el algoritmos para ordenar listas Biotonicas, también en [Nassimi 93] se presenta una implantación del algoritmo de ordenamiento por mezcla par e impar, todos estos algoritmos pertenecen a los algoritmos basados en mezclas y las implantaciones se realizaron en una computadora con arquitectura hipercubo.

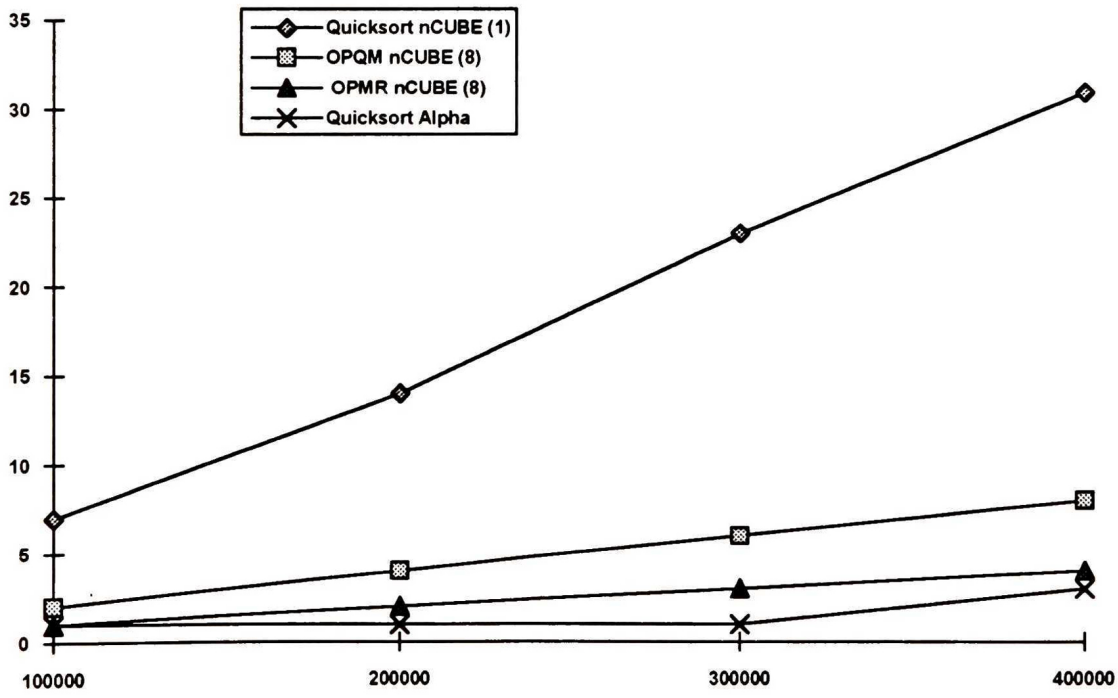


Figura 2.9 *Tiempos de proceso* (grafica comparativa)

Capítulo 3

Algoritmos de teoría de gráficas en paralelo

El objetivo de este capítulo es tratar dos problemas principales que corresponden a dos algoritmos de teoría de gráficas: el algoritmo paralelo para encontrar el árbol de extensión mínimal y una aproximación paralela al problema del vendedor viajero. En la sección 3.1 Teoría de gráficas, se dan definiciones y conceptos de gráficas, árboles, nodos, lazos y trayectorias; en 3.2 Solución Paralela al Arbol de Extensión Mínimal (SPAEM), se define en que consiste un árbol de extensión mínimal de una gráfica; en 3.2.1 Algoritmo SPAEM, se describen las características del algoritmo, como funciona, como se mapea a un hipercubo y se da un ejemplo del algoritmo, en 3.2.2 Complejidad, se analiza la complejidad del algoritmo, que nos dice cual es el comportamiento en tiempo del algoritmo; en 3.2.3 Resultados experimentales, se analizan pruebas realizadas en la computadora NCUBE 2; en 3.3 Aproximación al Problema del Vendedor Viajero (APVV), se describe en que consiste el problema del vendedor viajero y se comentan las características de los algoritmos de aproximación; en las secciones 3.3.1 Algoritmo APVV, se describen las características del algoritmo, como funciona, como se mapea a un hipercubo y se da un ejemplo del algoritmo; en 3.3.2 complejidad Complejidad, se analiza la complejidad del algoritmo, que nos dice cual es el comportamiento en tiempo del algoritmo; en 3.3.3 Razón límite, se demuestra que la razón límite del algoritmo APVV es igual a dos, es decir, cuando se minimiza un problema por un algoritmo de aproximación, el resultado no es mayor que el doble del resultado optimo; en 3.3.4 Resultados experimentales, se analizan pruebas realizadas en la computadora NCUBE 2.

3.1 Teoría de gráficas

Las gráficas fueron introducidas por el matemático L. Euler en 1736, y actualmente se utilizan en la representación de las relaciones que ocurren en varias situaciones del mundo real, como en los sistemas de comunicación, el rápido crecimiento de la tecnología VLSI y la disponibilidad de computadoras paralelas han permitido el desarrollo de algoritmos paralelos eficientes para varios problemas de teoría de gráficas, dada su amplia aplicación en redes de comunicaciones, eléctricas y de transportes, además en el diseño de VLSI, y en problemas de optimización entre otras áreas de la ciencia y la ingeniería, en donde los problemas requieren soluciones rápidas por lo que el procesamiento eficiente de gráficas ha sido un punto de investigación para diseñadores de algoritmos.

Sea $G = (V,L)$ una gráfica no dirigida con peso, donde V es un conjunto finito de vértices y L es un conjunto finito de lazos. Los vértices también son llamados *nodos* y los lazos se conocen como *arcos*. Un lazo esta formado por un par de vértices (v, u), si (v, u) es un lazo no dirigido, entonces $(v, u) = (u, v)$, de cualquier manera nos referiremos a una gráfica no dirigida simplemente como un gráfica. El símbolo $|\ |$ nos indicara la cardinalidad del conjunto, por lo que $n = |V|$ es el número vértices y $m = |L|$ el número de lazos. Dos vértices v y u son *adyacentes* si (v, u) es un lazo; se dice que (v, u) es incidente a los vértices v y u.

Una *trayectoria* es una secuencia de vértices $v_1, \ldots, v_n$ tal que (v_i, v_{i+1}) es un lazo para $1 \leq i < n$. Una trayectoria es *simple* si todos los vértices en la trayectoria son distintos, con la excepción que v_1, y v_n pueden ser el mismo, la longitud de la trayectoria es el número de lazos a través de la trayectoria. Se dice que una trayectoria $v_1, \ldots, v_n$ *conecta* a v_1, y v_n. Una gráfica es *conectada* si cualquier par de vértices de la gráfica esta conectado.

Sea $G = (V,L)$ una gráfica con el conjunto de vértices V y el conjunto de lazos L. Una *subgráfica* de G es una gráfica $G' = (V',L')$ donde

1. V' es un subcojunto de V.
2. L' consiste de lazos (v, u) en L, tal que v y u están en V'.

Si L' consiste de lazos (v, u) en L, tal que v y u están en V', entonces G' es llamado la *subgrafica inducida* de G.

Un *componente conectado* de una gráfica G es una subgráfica inducida que es conectada y máximal, una subgráfica inducida conectada, no es subgráfica propia de ningún otra subgráfica conectada de G.

Un *ciclo* en una gráfica es una trayectoria de longitud tres o más, que conecta un vértice con si mismo. Una gráfica es *cíclica* si contiene al menos un ciclo. Una gráfica acíclica conectada es llamada árbol.

En este capítulo presentaremos dos algoritmos de teoría de gráficas en computadora con arquitectura hipercubo NCUBE 2, el algoritmo denominado Solución Paralela al Arbol de Extensión Mínimal [Das 90a] y [Das 90b] y el algoritmo Aproximación al Problema del Vendedor Viajero [Cormen 92] y [Rosenkrantz 77].

3.2 Solución Paralela al Arbol de Extensión Mínimal (SPAEM)

Se considera una gráfica no dirigida, con peso $G = (V, L)$, con $n = |V|$ vértices y $m = |L|$ lazos, sin ciclos o lazos paralelos. Sea $V = \{v_1, \ldots, v_n\}$ el conjunto de vértices y $L = \{l_1, \ldots, l_m\}$ el conjunto de lazos, también se asume que $d(l_i) \in R^+$, donde R^+ es el conjunto de los números reales positivos y d es el peso o longitud del lazo e_i.. Una subgráfica de una gráfica conectada G es un árbol de extensión, si es un árbol que conecta todos los vértices de G. Un árbol de extensión mínimal de una gráfica con peso es aquel, donde la suma de los pesos de sus lazos es mínima.

Hay dos algoritmos secuenciales comúnmente usados para obtener el AEM de una gráfica no dirigida [Horowits 77] y [Cormen 91], que se basan en: la estrategia primero el lazo más corto dada por Kruskal, y en la estrategia primero el vecino más próximo dada por Prim y de manera independiente también por Dijkstra. El algoritmo paralelo SPAEM utiliza el algoritmo de Kruskal como subrutina para descartar sucesivamente lazos que no pertenecen al (AEM).

El algoritmo SPAEM es una combinación de los algoritmos para obtener el AEM de Kruskal que es un algoritmo de tipo voraz, y el Mergesort que es del tipo divide y vencerás [Horowits77] y [Knuth 73], el algoritmo consta de dos etapas las cuales son la aplicación de los algoritmos mencionados. El algoritmo mapea el problema a una arquitectura hipercubo. El algoritmo SPAEM que se implantó está basado en mezclas, y funciona en un hipercubo con k dimensiónes.

3.2.1 Algoritmo SPAEM

1. Algoritmo SPAEM, para el procesador i (i). Este algoritmo calcula el AEM de una gráfica no dirigida, en una computadora hipercubo k dimensiónal, es decir se puede utilizar con el número de procesadores disponibles en la computadora en que se implante. Una implantación en un hipercubo de este algoritmo se pude revisar en [Das 90a] y [Das 90b].
variables utilizadas:

- m es el número de lazos
- n es el número de vértices
- k, es la dimensión del hipercubo
- p, es el número de procesadores en el hipercubo y $p = 2^k$
- i, es el identificador de cada procesador, donde $0 \leq i \leq p-1$
- r, es el identificador del procesador fuente o destino, donde $0 \leq r \leq p-1$
- s, nos indica la dimensión de hipercubo actual, o el nivel en el árbol de mezcla
- *Tab*, es el arreglo que contiene los elementos a ordenar

2. Estructura de datos. La entrada de este programa es una lista de lazos *Tab*, que forma la gráfica *G*, distribuida en los p procesadores, la salida será una lista de lazos *Tab* que será depositada en el procesador 0 y formará el AEM del grafo dado. Cada dato estará formado por la tripleta (u, v, d), que serán nodo, nodo y distancia o peso, que nos describirán los lazos de la gráfica al que se le claculará el AEM.
Entrada :

Arreglo $Tab[0..m]$, sea $w=m/p$ y el subarreglo $Tab[i*w..(i+1)*w-1]$, está almacenado en la memoria local del procesador i del hipercubo con dimensión k.

Salida :

Arreglo Tab[$0..n$] lazos del AEM en el procesador i, donde $i = 0$.

3. Elementos del algoritmo. Aquí se mostrarálos elementos y subrutinas del procedimiento llamado SPAEM:

```
void SPAEM (){
  /* (4. Primera etapa)*/
  kruskal();
  /* (5. Segunda etapa)*/
  kruskal_paralelo();
  if (pn==0)
    resultado();
}
```

*4. **Primera etapa.*** En esta etapa un máximo de w lazos son asignados a cada uno de los procesadores, después todos los procesadores utilizan simultáneamente el algoritmo primero el lazo más corto dado por Kruskal, en su porción de lazos correspondiente y producen un AEM como una lista ordenada de lazos en base a su distancia, esto lo realizamos de la siguiente forma, en **quicksort**(), se ordenan los lazos de forma incremental en base a su distancia, después en **borra_lazos**(), si un lazo conecta, dos componentes conectados, entonces se adiciona el lazo al AEM, si el lazo conecta dos vértices en el mismo componente, el lazo se elimina ya que este lazo provocaría un ciclo en el AEM, cuando todos los vértices están en un componente, este componente es el AEM del grafo dado.

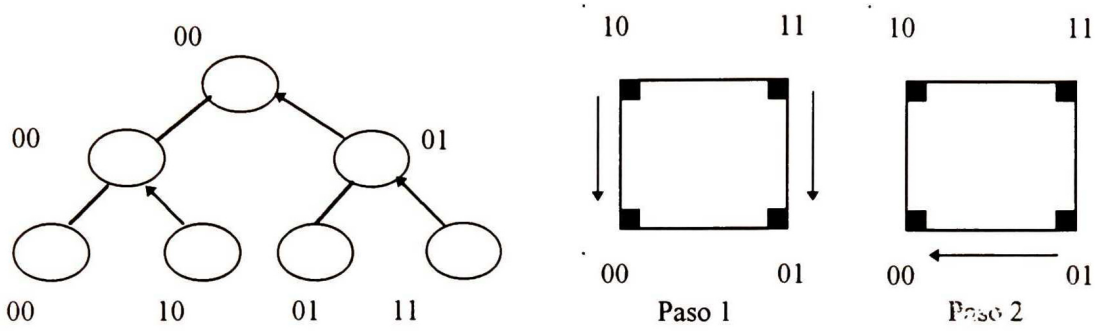
```
kruskal(){
  quicksort();
  borra_lazos();
}
```

*5. **Segunda etapa.*** En esta ultima etapa se realizan k transmisiones de datos en paralelo y también k veces se aplica el algoritmo de Kruskal, sólo que ahora para ordenar a los lazos se utiliza una mezcla paralela de dos listas en lugar de un **quicksort**(), después se emplea el procedimiento **borra_lazos**(), para encontrar el AEM, como se comento en ***4. Primera etapa***, en general durante el paso s, donde $0 \leq s < k$ y se comienza con $s = k-1$, el procesador i envía o recibe a lo más una lista de longitud n (número de vértices), al procesador r, el procesador que recibe la lista es aquel que tenga el identificador con número menor, después mezcla la lista que recibe y la propia, por medio del procedimiento **mezcla**(), las dos listas tienen a lo más longitud n, después de obtener la lista ordenada se aplica el procedimiento **borra_lazos**(), que permite obtener un AEM parcial, hasta el ultimo paso s que es donde se obtiene el AEM del grafo dado . Para calcular r *se* realiza la operación Or - exclusiva ($\oplus$) entre el identificador i y 2^s(se obtiene realizando un corrimiento de s bits). En cada paso s se elimina la mitad de los procesadores, esto sucede hasta que el resultado de la mezcla se concentra en el procesador 0, en la figura 3.1, se muestra en el inciso a), como se construye el árbol de mezcla desde que se utilizan los p procesadores hasta que se concentra el resultado en el procesador 0, además se puede ver como se mapea el árbol de mezcla al hipercubo k dimensiónal que se utilice, en la figura se considera que la dimensión de hipercubo $k = 2$.

```
kruskal_paralelo();{
for( s = k-1 ; s>=0 ; s--)
   if( (r = i ⊕ 1<< s)< i) {
      manda_datos(); /* el procesador i manda una lista al procesador r */
      return;
   }else{
      recibe_datos(); /* el procesador i recibe una lista del procesador r */
}
      mezcla(); /* el procesador i mezcla dos, listas la recibida con la propia */
      borra_lazos();
}
```

Después del **kruskal_paralelo()**, la lista que contiene los lazos con el AEM se encuentra en el procesador 0.

```
if (pn==0)
    resultado();
```



a) árbol de mezcla y eliminación de lazos

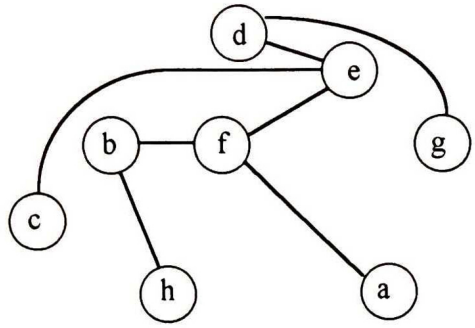
b) árbol de mezcla en el hipercubo

Figura 3.1 *Ejemplo de etapa dos del SPAEM, k=2*

A continuación se verá un ejemplo de como funciona el algoritmo SPAEM, para esto se utilizara una serie de lazos no ordenada en forma de la tripleta (vértice, vértice, distancia), por lo que los lazos de la gráfica no dirigida *G*, serán: ((a,b,20), (a,d,20), (a,f,8), (a,h,9), (b,d,8), (b,f,4), (b,h,5), (c,e,20), (c,g,26), (d,e,2), (d,g,8), (e,f,2), (f,h,5), (a,c,26), (a,e,10), (a,g,4), (b,c,2), (b,e,10), (b,g,16), (c,d,18), (c,f,10), (c,h,5), (d,f,4), (d,h,17), (e,g,2), (f,g,4), (g,h,13)) y se claculará el AEM utilizando un hipercubo con dimensión k=1, es decir con dos procesadores, $p = 2 = 2^k$. Primero, se dividen los datos de manera que cada procesador tenga aproximadamente el mismo número de datos, los subconjuntos de datos distribuidos deben ser disjuntos, después se calcula el AEM local de los lazos en cada procesador, esta se realiza en ***4. Primera etapa,*** con el procedimiento **kruskal()**, ver figura 3.2 inciso a). Ahora cada procesador contara con un AEM parcial y claculará el AEM global con el procedimiento **kruskal_paralelo()**, primero intercambiando los datos entre los procesadores se mezclaran los datos con el procedimiento **mezcla()**, después con **borra_lazos()** eliminara los lazos que no pertenecen al AEM, como se revisa en ***5. Segunda etapa***, ver figura 3.2 inciso b). Al final el AEM se encuentra en el procesador 0 y es ((d,e,2), (e,f,2), (b,f,4), (b,h,5), (b,c,2), (e,g,2), (a,g,4)).

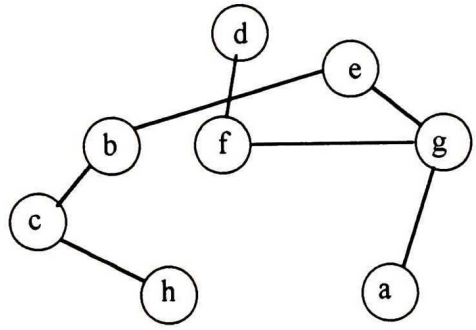
Entrada, el grafo G = [(a,b,20), (a,d,20), (a,f,8), (a,h,9), (b,d,8), (b,f,4), (b,h,5), (c,e,20), (c,g,26), (d,e,2), (d,g,8), (e,f,2), (f,h,5), (a,c,26), (a,e,10), (a,g,4), (b,c,2), (b,e,10), (b,g,16),(c,d,18), (c,f,10), (c,h,5), (d,f,4), (d,h,17), (e,g,2), (f,g,4), (g,h,13)]

P0



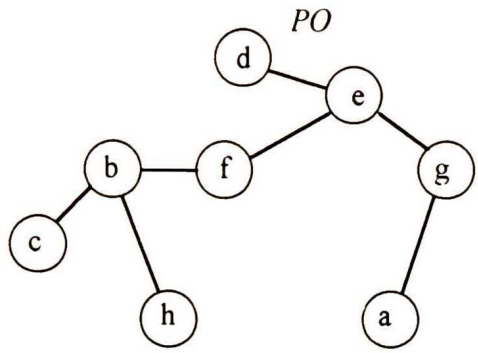
Entrada para el procesador 0 = [(a,b,20), (a,d,20), (a,f,8), (a,h,9), (b,d,8), (b,f,4), (b,h,5), (c,e,20), (c,g,26), (d,e,2), (d,g,8), (e,f,2), (f,h,5)]
Salida = [(d,e,2), (e,f,2), (b,f,4), (b,h,5), (a,f,8), (d,g,8), (c,e,20)]

P1



Entrada para el procesador 1 = [(a,c,26), (a,e,10), (a,g,4), (b,c,2), (b,e,10), (b,g,16), (c,d,18), (c,f,10), (c,h,5), (d,f,4), (d,h,17), (e,g,2), (f,g,4), (g,h,13)]
Salida = [(b,c,2), (e,g,2), (a,g,4), (d,f,4), (f,g,4), (c,h,5), (b,e,10)]

a) primera etapa, **kruskal**(), en el procesador i=1 y 2



Entrada = [(d,e,2), (e,f,2), (b,f,4), (b,h,5), (a,f,8), (d,g,8), (c,e,20), (b,c,2), (e,g,2), (a,g,4), (d,f,4), (f,g,4), (c,h,5), (b,e,10)]
Salida = [(d,e,2), (e,f,2), (b,f,4), (b,h,5), (b,c,2), (e,g,2), (a,g,4)]

b) segunda etapa, **kruskal_paralelo**()

Figura 3.2 *Ejemplo de árbol de extensión mínimal, k=1*

3.2.2 Complejidad

Para el calculo de la complejidad se considera una gráfica no dirigida, con peso $G = (V, L)$, con $n = |V|$ vértices y $m = |L|$ lazos, sin ciclos o lazos paralelos. El SPAEM consiste en que cada procesador obtenga el árbol de extensión mínimal (AEM) de una sublista de lazos de longitud m/p, considerando a p fijo y con un valor $p \leq \log(n)$, el AEM se obtiene de la siguiente forma, en la primera etapa del algoritmo, primero con el procedimiento **kruskal**(), que consta de los procedimientos **quicksort**() y **borra_lazos**(), en **quicksort**(), se ordena una lista local de lazos, de la misma forma que al inicio del algoritmo OPQM del capítulo 2.2, por lo tanto podemos decir:

Quick_ comparaciones = $(m/p) \log(m) - mk/p$

A estas sublistas ordenadas se le aplica el procedimiento **borra_lazos**(), que elimina los lazos que crean ciclos en un tiempo $O(m/p)$, estos dos procedimientos forman el AEM, en cada procesador de la misma forma que el algoritmo secuencial de Kruskal.

La segunda etapa consiste en el procedimiento **kruskal_paralelo**(), que calcula el AEM en base a varios AEM parciales, esto se realiza de la siguiente forma, primero combina sucesivamente las sublistas de lazos de cada procesador, y después cada que un procesador obtenga una nueva lista, se combina la lista propia con la que llega esto con el procedimiento mezcla en un tiempo $O(n)$, se aplica **borra_lazos**(), que ocupa un tiempo $O(n)$, estos dos procedimientos se realizan k veces por lo que la complejidad es $O(nk) = O((n)(\log(p))$.

El tiempo de comunicación durante la segunda etapa consta de dos partes: El tiempo de inicialización *TI* y el tiempo de transferencia de un dato *TD*. podemos estimar el tiempo total de comunicación del algoritmo de la siguiente forma:

$$\begin{aligned} Tiempo\_comunicaciones &= \sum_{i=1}^{k} (TI + TD\, n) = TI\, k + TD \sum_{i=1}^{k} n \\ &= TI\, k + TD\ (nk) \end{aligned}$$

De lo anterior observamos que la cantidad de datos a transmitir depende linealmente de n, ya que con k constante o fija, $O(nk) = O(n)$. Por lo tanto podemos concluir que la primera etapa que aplica el procedimiento kruskal(), requiere $O((m/p)\log(m))$, mientras que en la segunda etapa, la mezcla(), requiere $O((n)\log(p))$ y la comunicación requiere $O((n)\log(p))$ ya que $k = \log(p)$, entonces la complejidad es:

$$\begin{aligned} tiempo(m) &= O((m/p)\log(m)) + O((n)\log(p)) + O((n)\log(p)) \\ &= O((m/p)\log(m)) + O((n)\log(p)) \end{aligned}$$

3.2.3 Resultados experimentales

El algoritmo SPAEM se implantó en una computadora NCUBE con ocho procesadores y se programó en lenguaje C. Para realizar las pruebas fueron generadas series de datos aleatorios (para obtener cada valor se realizaron veinte pruebas), además se probó con datos (números enteros de 4 bytes) y con 1, 2, 4, y 8 procesadores (hipercubo de dimensión $k=0$, *1*, *2*, y *3*) en una computadora dedicada. Los resultados se ilustran en la figura 3.3, y no incluyen el tiempo para distribuir los datos en los procesadores. Podemos ver que en términos generales el comportamiento del algoritmo es bueno ya que efectivamente se reduce el tiempo de ejecución conforme aumentan los procesadores.

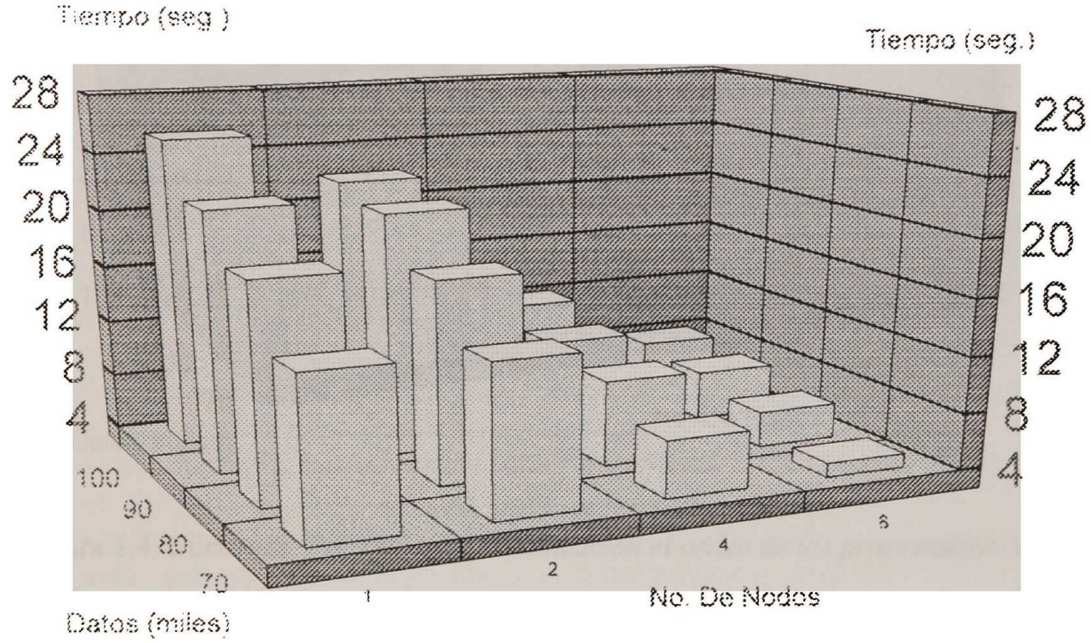


Figura 3.3. *Tiempos de proceso*

Ahora se analiza la eficiencia del algoritmo E(n), como se definió en el capítulo 1.5.

$$E(n) = TS(n)/(pTP(n))$$

Se nota que cuando va aumentando el número de procesadores la eficiencia es menor como lo ilustra la figura 3.4, ésto se debe a las características del algoritmo, que en cada nivel de mezcla se subutilizan la mitad de los procesadores que están activos, es decir, en el último paso del algoritmo cuando se mezclan las dos últimas listas ordenadas, sólo un procesador se encuentra trabajando, un punto a favor de este algoritmo es que las listas a mezclar en paralelo, siempre permanecen de longitud *n*.

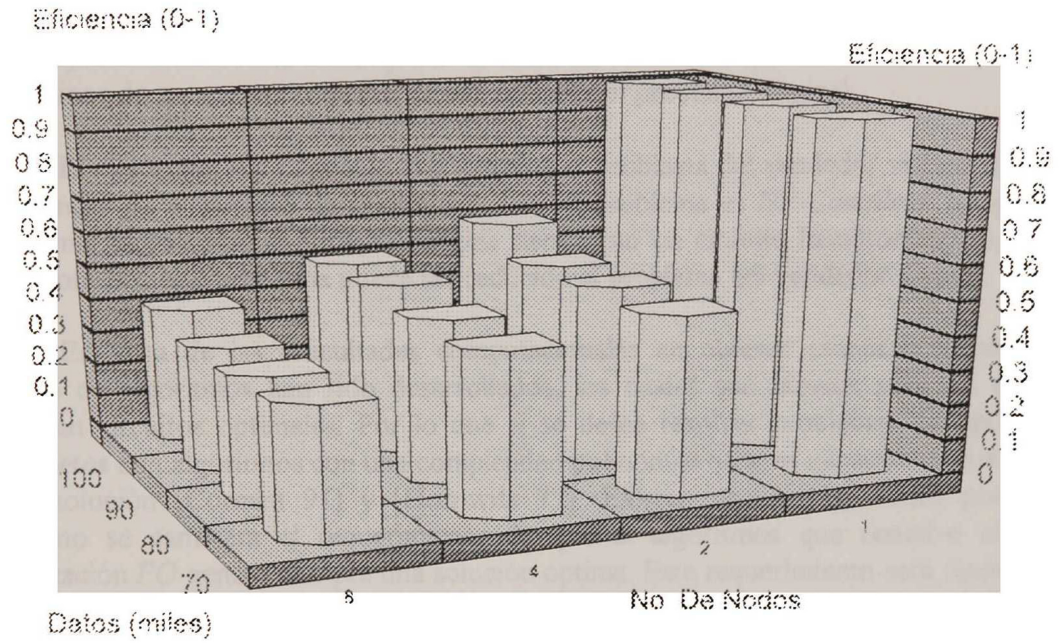


Figura 3.4. Eficiencia. *Para fines de visualización el orden de los procesadores es diferente*

3.3 Aproximación al Problema del Vendedor Viajero (APVV)

El problema del vendedor viajero (PVV), es de gran interés y ha sido planteado de diferentes maneras, el algoritmo APVV que se implantó tiene el siguiente planteamiento.

Una gráfica G del vendedor viajero es una gráfica no dirigida con peso especificada por la dupla (V,L) donde V es el conjunto de vértices y L es el conjunto de vértices, donde d es la formulacion de distancia que mapea pares de nodos o lazos, en los números reales, y donde d satisface :

a) $d(j,i) = d(i,j)$ para todo i,j en V
b) $0 \leq d(i,j)$ para todo i,j en V
c) $d(i,k) \leq d(i,j) + d(j,k)$ para todo i,j,k en V

La condición c) se conoce como la desigualdad del triángulo, el número $d(i,j)$ es llamado longitud o peso de (i,j), un recorrido del vendedor viajero en la gráfica G es un circuito en la gráfica, conteniendo cada nodo exactamente una vez (un circuito hamiltoniano). La longitud de un recorrido es la suma de la longitud de los lazos que componen el circuito. Una solución óptima o circuito óptimo para G, es un circuito con longitud mínima.

El PVV es algunas veces formulado como el problema de encontrar un circuito de longitud mínima, conteniendo cada nodo una vez en una grafica no dirigida en donde las distancias no están sujetas a la desigualdad del triángulo, no obstante un problema establecido de esta manera puede ser reducido al problema considerado aquí, cambiando cada $d(i,j)$ a la longitud de la trayectoria más corta entre i,j. Por lo tanto las resultados que inicialmente fueron propuestos en términos de un nuevo problema, puede aplicarse al problema original.

El mejor método conocido que resuelve el problema del vendedor viajero toma un tiempo exponencial en el número de nodos, aún más el problema es *NP-Completo*. Se puede ver que determinar cuando una gráfica no dirigida tiene o no un circuito hamiltoniano es *NP-Completo* [Hopcroft 79], este problema puede ser reducido al problema del vendedor viajero.

En vista de las dificultades computacionales en obtener circuitos optímales, un buen número de algoritmos han sido desarrollados, los cuales son rápidos pero no necesariamente producen circuitos optímales. Por lo que si se desea resolver problemas de optimización **NP-Completos** con algoritmos con una complejidad polinomial baja, es necesario relajar el significado de la solución [Cormen 91] y [Horowits 77]. Existen varias relajaciones posibles, en este algoritmo se cambiara el requerimiento de que el algoritmos que resuelve el problema de optimización *PO* genere siempre una solución optima. Este requerimiento será reemplazado por el de que el algoritmo para *PO* siempre genere una solución factible con un valor "cercano" al valor de la solución optima, la que es llamada solución aproximada. Un algoritmo de aproximación para *PO* es un algoritmo que genera una solución aproximada para *PO*. La razón límite es un factor que nos indica que tan optima es la aproximación y se define como : razón limite = Solución aproximada / solución optima. Entre mayor se la razón límite la aproximación será peor.

El algoritmo APVV que se planteo es una algoritmo de aproximación. El recorrido que el algoritmo APVV regresa tiene razón límite de 2, es decir la distancia del recorrido no es dos veces mayor que la longitud del circuito óptimo.

Se calcula un circuito del vendedor viajero de una grafica G no dirigida, usando el algoritmo SPAEM de la sección 3.2 para calcular el árbol de extensión mínimal de una gráfica dada, cuando la función de costo satisface la desigualdad del triángulo. El algoritmo mapea el problema a una arquitectura hipercubo. El algoritmo consta de tres etapas las cuales se describirán a continuación.

3.3.1 Algoritmo APVV

1. Algoritmo APVV, para el procesador i (i). Este algoritmo calcula una solución aproximada al problema del vendedor viajero (PVV), de una gráfica no dirigida, en una computadora hipercubo k dimensiónal, es decir se puede utilizar con el número de procesadores disponibles en la computadora en que se implante. Una versión secuencial de este algoritmo se pude revisar en [Cormen 91].

variables utilizadas:

m es el número de lazos
n es el número de vértices
k, es la dimensión del hipercubo
p, es el número de procesadores en el hipercubo y $p = 2^k$
i, es el identificador de cada procesador, donde $0 \leq i \leq p-1$
r, es el identificador del procesador fuente o destino, donde $0 \leq r \leq p-1$
s, nos indica la dimensión de hipercubo actual, o el nivel en el árbol de mezcla
Tab, es el arreglo que contiene los elementos a ordenar

2. Estructura de datos. La entrada de este programa es una lista de lazos *Tab*, que forma la gráfica *G*, distribuida en los *p* procesadores, la salida será una lista de lazos *Tab* que será depositada en el procesador 0 y formará un circuito del vendedor viajero. Cada dato estará formado por la tripleta (*u, v, d*), que serán nodo, nodo y distancia o peso, que nos describirán los lazos de la gráfica al que se le claculará una solución al PVV.

Entrada :

Arreglo *Tab*[*0..m*], sea *w=m/p* y el subarreglo *Tab*[*i*w..(i+1)*w-1*], está almacenado en la memoria local del procesador *i* del hipercubo con dimensión *k*.

Salida :

Arreglo Tab[*0..n*] datos del circuito del vendedor viajero en el procesador *i*, donde *i = 0*.

3. Elementos del algoritmo. Aquí se mostrará los elementos y subrutinas del procedimiento llamado APVV:

```
void APVV(){
  /* (4. Primera etapa)*/
  SPAEM();
  /* (5. Segunda etapa)*/
  elige_raiz();
  preorden();
  if (pn==0)
    resultado();
}
```

4. Primera etapa. Esta etapa consiste en que cada procesador utilice el procedimiento **SPAEM()**, que se revisó en la sección 3.2.1 Algoritmo SPAEM, para obtener el árbol de extensión mínimal (AEM) T, de la gráfica no dirigida G.

```
void SPAEM (){
  kruskal();
  kruskal_paralelo();
}
```

5. Segunda etapa. En esta etapa se selecciona un vértice como la raíz del AEM, $r \in V$ se puede tomar cualquier vértice.

```
elige_raiz(); /* se elige un vértice cualquiera */
```

Después se obtiene la lista de los vértices LV que son visitados en un recorrido preorden en el AEM T, y el resultado es el ciclo hamiltoniano que visita los vértices en el orden LV. El recorrido preorden en el AEM, consiste en visitar recursivamente todos los vértices del árbol, listando el vértice cuando es encontrado por primera vez, esto es antes que cualquiera de sus hijos sea visitado.

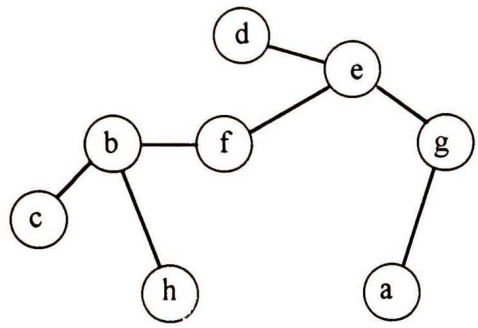
```
preorden(); /* tomada a r como raíz y recorre el AEM */
```

Después de preorden(), la lista que contiene los lazos con el circuito del vendedor viajero se encuentra en el procesador 0.

```
if (pn==0)
    resultado();
```

A continuación se verá un ejemplo de como funciona el algoritmo APVV, para esto se utilizara una serie de lazos no ordenada en forma de la tripleta (vértice, vértice, distancia), por lo que los lazos de la gráfica no dirigida G, serán: ((a,b,20), (a,d,20), (a,f,8), (a,h,9), (b,d,8), (b,f,4), (b,h,5), (c,e,20), (c,g,26), (d,e,2), (d,g,8), (e,f,2), (f,h,5), (a,c,26), (a,e,10), (a,g,4), (b,c,2), (b,e,10), (b,g,16), (c,d,18), (c,f,10), (c,h,5), (d,f,4), (d,h,17), (e,g,2), (f,g,4), (g,h,13)) y se claculará una solución al PVV utilizando un hipercubo con dimensión k=1, es decir con dos procesadores, $p = 2 = 2^k$ Primero, se aplica el procedimiento SPAEM(), para obtener el AEM esto se realiza en ***4. Primera etapa***, ver figura 3.5 inciso a). Ahora el AEM se encuentra en el procesador 0 y es ((d,e,2), (e,f,2), (b,f,4), (b,h,5), (b,c,2), (e,g,2), (a,g,4)). Después se aplicaran los procedimiento **elige_vertice()** que elige una raíz (a) y el procedimiento **preorden()**, recorrerá el árbol, como se revisa en ***5. Segunda etapa***, ver figura 3.5 inciso b) y c). Finalmente el circuito del vendedor viajero es: ((a), (g), (e), (d), (f), (b), (c), (h)).

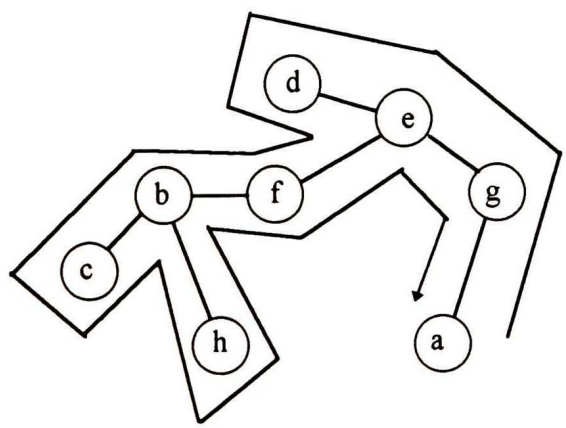
P0



Entrada, la gráfica G = [(a,b,20), (a,d,20), (a,f,8), (a,h,9), (b,d,8), (b,f,4), (b,h,5), (c,e,20), (c,g,26), (d,e,2), (d,g,8), (e,f,2), (f,h,5), (a,c,26), (a,e,10), (a,g,4), (b,c,2), (b,e,10), (b,g,16),(c,d,18), (c,f,10), (c,h,5), (d,f,4), (d,h,17), (e,g,2), (f,g,4), (g,h,13)]
Salida = [(d,e,2), (e,f,2), (b,f,4), (b,h,5), (b,c,2), (e,g,2), (a,g,4)]

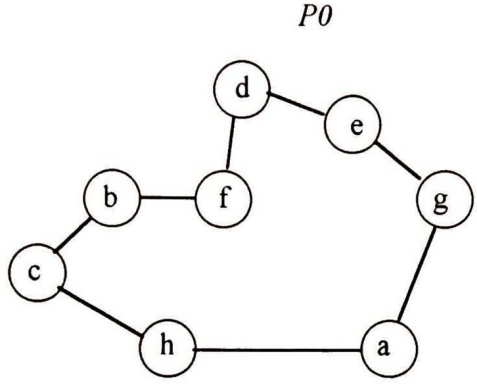
a) primera etapa, aplicar **SPAEM()**

P0



Entrada = [(d,e,2), (e,f,2), (b,f,4), (b,h,5), (b,c,2), (e,g,2), (a,g,4)]
Salida = [(a), (g), (e), (d), (f), (b), (c), (h)]

b) segunda etapa, **elige_raiz()** y **preorden()**



c) Circuito obtenido por **APVV()**

Figura 3.5 *Ejemplo del algoritmo APVV, k=1*

3.3.2 Complejidad

La primera etapa del algoritmo APVV consiste en el procedimiento **SPAEM()**, que vimos en la sección 3.2 de este capítulo, por lo tanto podemos decir :

SPAEM_Operaciones = $O(m/p) \log (m) + O(n)\log(p)$

La segunda etapa consiste de dos procedimientos, **elegir_raiz()** y **preorden()**, **elegir_raiz()** toma un nodo del lazo más corto por lo que toma un tiempo $O(1) = O(c)$, con c constante. La tercera etapa consiste en un recorrido preorden(), en el AEM T con n nodos, por lo que toma un tiempo $O(n)$.

Observamos que la cantidad de datos a transmitir depende del algoritmo SPAEM por lo que es:

$Tiempo\_comunicaciones = TI\ k + TD\ (nk)$

Por lo tanto podemos concluir que en la primera etapa, el procedimiento SPAEM(), toma $O((m/p)\log(m)) + O((n)\log(p))$, esta etapa también nos da la complejidad en la comunicación que es $O((n)\log(p))$, mientras la etapa de elegir_raiz(), requiere $O(c)$ y la etapa del recorrido preorden(), requiere $O(n)$, entonces la complejidad es:

$$\begin{aligned} tiempo(m) &= O((m/p)\log(m)) + O((n)\log(p)) + O(c) + O(n) \\ &= O((m/p)\log(m)) + O((n)\log(p)) + O(n) \end{aligned}$$

3.3.3 Razón límite

El algoritmo APVV es un algoritmo de aproximación [ver Apéndice 3A], con razón límite igual a 2, para el problema del vendedor viajero que cumple con la desigualdad del triángulo.

Demostración

Sea H^* un circuito óptimo para un conjunto V, de vértices. Una forma equivalente del teorema es costo ($H^\wedge$) ≤ 2(costo (H^*)), donde $H^\wedge$ es el circuito regresado por APVV. Ya que se obtiene un árbol de extensión mínimal (AEM) en base a los lazos de menor longitud, si T es un AEM para un conjunto de vértices, entonces

costo (T) ≤ costo (H^*) 3.3.1

un *recorrido completo* de un árbol T, lista los vértices cuando son visitados primero y también cuando quiera que regresen después de visitar al subárbol, a este recorrido lo llamaremos R. El recorrido completo del AEM del ejemplo de la figura 3.5 da el siguiente orden.

a, g, e, d, e, f, b, c, b, h, b, f, e, g, a.

Ya que el recorrido completo atraviesa todos los lazos de T exactamente dos veces, se tiene

Costo (R) = 2 (costo (T)) 3.3.2

Las ecuaciones (3.3.3.1) y (3.3.3.2) implican que

costo (R) ≤ 2 (costo (H^*)), 3.3.3

Y por lo tanto el costo de R esta dentro del factor de 2, del costo del circuito óptimo . Aun que R no es un circuito, ya que visita algunos vértices mas de una vez, por medio de la desigualdad del triángulo se puede eliminar la visita a cualquier vértice de R y el costo no se incrementa (si un vértice v que esta entre los vértices u y w es borrado de R , el orden resultante indica ir directamente de u a w). Aplicando varias veces esta operación podemos remover de R a todos los vértices excepto a la primera visita de cada vértice. en el ejemplo de la figura 3.5 nos queda el siguiente orden

a, g, e, d, f, b, c, h.

Este orden es el mismo que el obtenido al realizar un recorrido preorden en el árbol T. Sea $H^\wedge$ el ciclo correspondiente a este recorrido preorden. Este es un ciclo hamiltoniano, ya que cada vértice es visitado exactamente una vez, y este ciclo es calculado por el APVV. Ya que $H^\wedge$ es obtenido al borrar vértices del recorrido completo R, se tiene

costo ($H^{\wedge}$) ≤ costo (R) 3.3.4

combinando las desigualdades (3.3.3) y (3.3.4) se completa la demostración.

costo ($H^{\wedge}$) ≤ costo (R) ≤ 2 (costo (H^{*}))
costo ($H^{\wedge}$) ≤ 2 (costo (H^{*}))

□

3.3.4 Resultados experimentales

El algoritmo APVV se implantó en una computadora NCUBE con ocho procesadores y se programó en lenguaje C. Para realizar las pruebas fueron generadas series de datos aleatorios (para obtener cada valor se realizaron veinte pruebas), además se probó con datos (números enteros de 4 bytes) y con 1, 2, 4, y 8 procesadores (hipercubo de dimensión k=*0, 1, 2,* y *3*) en una computadora dedicada. Los resultados se ilustran en la figura 3.6, y no incluyen el tiempo para distribuir los datos en los procesadores. Podemos ver que en términos generales el comportamiento del algoritmo es bueno ya que efectivamente se reduce el tiempo de ejecución conforme aumentan los procesadores.

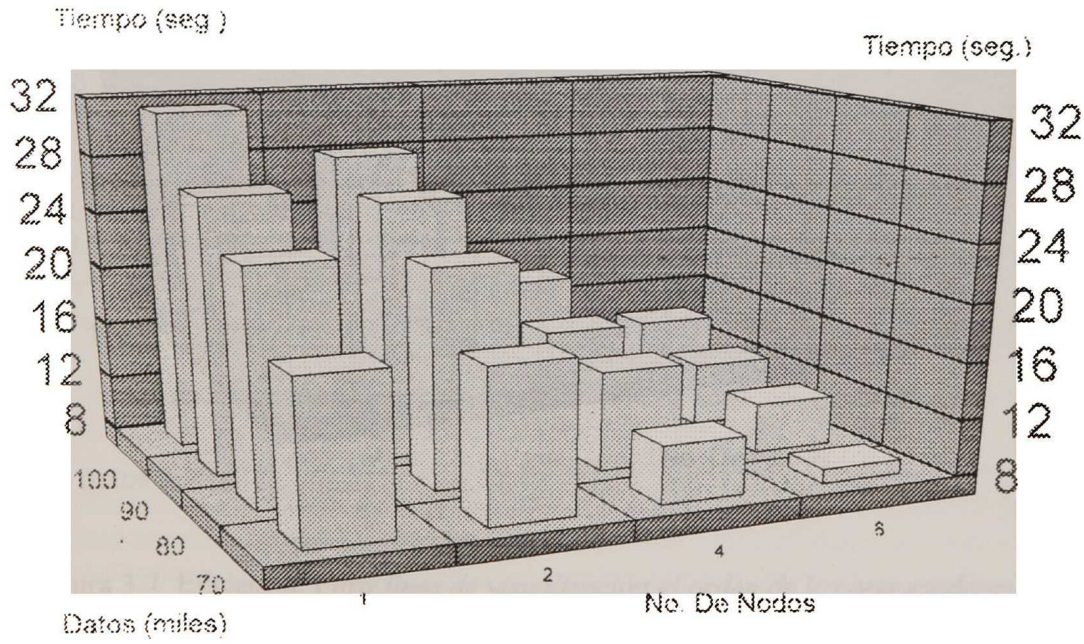


Figura 3.6. *Tiempos de proceso*

Ahora se analiza la eficiencia del algoritmo E(n), como se definió en el capítulo 1.5.

$$E(n) = TS(n)/(pTP(n))$$

Se nota que cuando va aumentando el número de procesadores la eficiencia es menor como lo ilustra la figura 3.7, ésto se debe a las características del algoritmo, ya que en la primera etapa, se utiliza el algoritmo SPAEM, el algoritmo APVV presenta un comportamiento similar es decir, en cada nivel de mezcla se subutilizan la mitad de los procesadores que están activos, ya que, en el último paso del algoritmo cuando se mezclan las dos últimas listas ordenadas, sólo un procesador se encuentra trabajando y en la segunda etapa los procedimientos **eligir_raiz**() y el recorrido **preorden**(), los realiza el ultimo procesador. Además de este algoritmo, en [Dutt 93] se presenta una solución al problema del vendedor viajero basándose en técnicas de tipo brand and bound, tambien en [Dahlhaus 88] se presenta un algoritmo para encontrar circuitos hamiltonianos en gráficas densas..

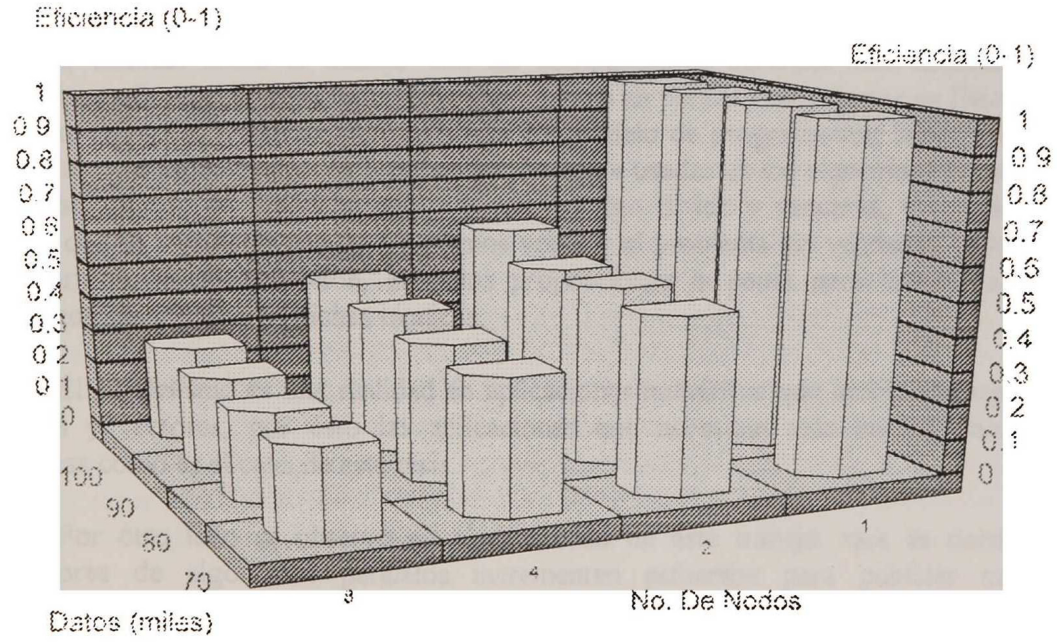


Figura 3.7. Eficiencia. *Para fines de visualización el orden de los procesadores es diferente*

Conclusiones

En el análisis y diseño de los algoritmos del capitulo 2 y 3, se observaron varias ventajas del modelo de programación SCMD: primero, el flujo de control de los algoritmos es simple ya que el comportamiento del algoritmo es el mismo en todos los procesadores, esto es muy importante ya que la idea es que la programación eficiente sea fácil de implantar ; segundo, los resultados se obtienen distribuyendo los datos en el numero de procesadores disponibles, es decir se pueden escalar los algoritmos al número de procesadores disponibles de una forma sencilla; tercero, la realización de depuradores paralelos es mas fácil, incluso se puede utilizar parcialmente los depuradores secuenciales que existen; cuarto, los compiladores actuales pueden generar código eficiente de un programa SCMD, es decir este modelo es una solución actual. Aunque por otro lado las principales desventajas son que, el software que se obtiene no es portable y que este modelo no es aplicable para aplicaciones como bases de datos distribuidas o sistemas operativos distribuidos. Por lo que este modelos no es la solución a la programación paralela, sino son sólo una solución; que tiene una gran aplicabilidad, más de la que se había pensado en un principio.

Otro punto importante es que los algoritmos presentados proporcionan solución a una gran cantidad de problemas ya que varios algoritmos se pueden reducir a los algoritmos aquí tratados, además inicia el trabajo con las computadoras paralelas con arquitectura de tipo hipercubo en la sección de computación (hay trabajo en paralelismo anterior en [Morales 87] pero con otro enfoque). Dado las características del modelo de programación SCMD, los algoritmos presentados se fundamentan en técnicas ampliamente usadas en los algoritmos secuenciales como en los algoritmos de ordenamiento que son del tipo divide y vencerás, el árbol de extensión mínima que es calculado con un algoritmos voraz y el problema del vendedor viajero que utiliza un algoritmo de aproximación, esto nos proporcionan la pauta para la realización de otros algoritmos basándonos en dichas técnicas.

El paralelismo es una realidad en aplicaciones numéricas que involucran operaciones con matrices y vectores, por ésto las aplicaciones que necesitan más investigación son las no numéricas como es el caso de esta tesis.

Por otro lado se observo en el desarrollo de este trabajo, que es necesario que los diseñadores de algoritmos paralelos incrementen esfuerzos para publicar colecciones de algoritmos paralelos en un pseudo código de alto nivel para reducir la necesidad de conocer detalladamente las características del hardware, donde este pseudo lenguaje de alto nivel deberá ser escrito específicamente para computadoras paralelas con múltiples instrucciones y múltiples datos (MIMD) de modelo de red como para computadoras (MIMD) de memoria compartida lo que fortalecerá la programación paralela.

Un punto negativo del enfoque algoritmico es que aun cuando el algoritmo central sea paralelizado manualmente, el código de una aplicación compleja no será acelerado como se desearía, aun más, los mejores algoritmos diseñados con el mayor cuidado contienen paralelismo implícito adicional, que puede ser a muy bajo nivel o demasiado irregular para ser explotado explícitamente por el diseñador humano. Por lo que para que el paralelismo sea una realidad es necesario explotar efectivamente el paralelismo a todos niveles.

Por otro lado debemos considerar que se ha invertido demasiado en software secuencial por lo que la idea de rehusar software vía compiladores que reconozcan el paralelismo en computadoras moderadamente paralelas es razonable, aunque el paralelismo a gran escala requiere una perspectiva diferente ya que son necesarios no sólo lenguajes paralelos y extensiones paralelas a los lenguajes existentes, se necesitan también depuradores paralelos, editores dirigidos a la sintaxis, herramientas manejadoras de configuración, seguir las orientaciones y requerimientos de la ingeniería de software moderna. Para que el desarrollo de software paralelo se convierta en una realidad, el software deberá ser portable a una variedad de computadoras : no es razonable realizar esfuerzos en desarrollar proyectos de software a gran escala para ser utilizados en una computadora con un futuro incierto, por lo que el software debe ser independiente de las características del hardware.

Trabajos futuros

Actualmente el procesamiento paralelo es una subdiciplina de las ciencias de la computación lo suficiente madura para tener sus propias conferencias, publicaciones y organización profesional. Pero el término es usado con considerable ambigüedad, algunas veces "paralelo" significa "concurrente", otras "distribuido" y otras veces "multiprocesadores" o "multicomputadoras" Frecuentemente las redes neuronales son llamadas paralelas aun cuando no poseen distintas instrucciones. Por lo que seria muy útil para un trabajo futuro encontrar una clasificación más descriptiva de lo que está sucediendo en el campo. La clasificación dada por Michael S. Flynn necesita extenderse o modificarse, ya que aunque algunos autores como [Anderson 89], revisan algunas clasificaciones opcionales a la clasificación de Flynn es necesario extender estas clasificaciones en términos de la topología física, protocolos de interconexión, heterogeneidad de sistemas, memoria, flujo de datos y otras variables más.

Algunos puntos interesantes a desarrollar en computación paralelas son:

Un procesamiento más efectivo (en este punto se centro esta tesis).

- Aproximar a los procesadores paralelos a su desempeño pico.
- Hacer a la programación de procesadores paralelos más fácil y automatizada.
- Lograr que el procesamiento paralelo sea mas efectivo en costo.

Un procesamiento distribuido heterogéneo, ésto es que los sistemas puedan contener distintos tipo de computadoras en forma de una computadora paralela virtual.

- Optimización estática a través de una selección óptima de los procesadores para cumplir con los requerimientos de cada aplicación en particular, ésto es posible combinando máquinas paralelas con máquinas secuenciales para atacar problemas que cuenten con ambas características.
- Optimización dinámica de los procesadores disponibles, es decir, lograr el punto anterior en cada momento, además de evitar la subutilizacion de los procesadores.

Tecnología que soporte procesamiento paralelo

- Sistemas operativos distribuidos.
- Mayor ancho de banda en redes de comunicación.

Referencias

[Abali 93] Bülent Abali, Özguner, y Abdulla Bataineh, *"Balanced Parallel Sort On Hypercube Multiprocessors"*, IEEE Transactions On Parallel And Distributed Systems, Vol. 4, No. 5, pp. 572-581, Mayo 1993."

[Akl 89] Selim G. Akl, *"The Desing and Analysis Of Parallel Algorithms "*, Prentice-Hall, Inc., 1989.

[Anderson 89] A. John Anderson, *"Multiple Processing: a System Overview"*, Prentice-Hall, 1989.

[Bell 94] Gordon Bell, *"Scalable, Parallel Computers Alternatives, Issues, and Challenges"*, International Journal of Parallel Programming, vol. 22, No. 1, 1994.

[Bentley 86] Jon Bentley, *"Programming pearls: Literate programming"*, Communications of the ACM, 1986.

[Bertsekas 89] Dimitri P. Bertsekas, John N. Tsitsiklis, *"Parallel and Distributed Computation"*, Prentice Hall, 1989.

[Bertsekas 91] D. P. Bertsekas, C. Överen, G. D. Stamoulis, P. Tseng, y J. N. Tsitsiklis, *"Optimal Communication Algorithms For Hypercubes "*, Journal Of Parallel and Distributed Computing 11, pp. 263-275, 1991.

[Chaudhuri 92] Prany Chaudhuri, *"Parallel Algorithms : Design and Analisis"*, Prentice Hall, 1992.

[Cormen 92] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *"Introduction to Algorithms"*, the MIT press y McGraw-Hill, 1992.

[Dahlhaus 88] E. Dahlhaus, P. Hajnal, M. Karpinski, Prasad *" Optimal Parallel Algorithm for the Hamiltonian Cycle Problem on Dense Graphs"*, Titulo Conf. : 29th Annual Symposium on Foundations of Computer Science (IEEE Cat. No. 88CH2652-6), pp. 186-93, 24-26 Oct. 1988.

[Das 90a] Sajal K. Das, Narsingh Deo, Sushil Prasad *"Parallel Graph Algorithms For Hypercube Computers"*, Parallel Computing, Vol. 13, pp 143-158, 1990.

[Das 90b] Sajal K. Das, Narsingh Deo, Sushil Prasad *"Two Minimum spanning Forest Algorithms On Fixed-Size Hypercube Computers"*, Parallel Computing, Vol. 15, pp 179-187, 1990.

[Dutt 93] Shantanu Dutt y Nihar R. Mahapatra, *"Parallel A* Algorithms and Their Performance On Hypercube Multiprocessors"*, Titulo Conf. : Proccedings Of Seventh International Parallel Processing Symposium (IEEE Cat. No. 93THO513-2), pp. 797-803, 1993.

[Fox 88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker, *"Solving Problems On Concurrent Processors"*, Volumen 1, Prentice Hall - Englewood cliffs, New Jersey, 1988.

[Flynn 72] Michael J. Flynn, *"Some Computer Organisations and their Efficient Supercomputing"*, IEEE T-C, V21, n9, pp 948-960, Septiembre 1972.

[Goldschlager 77] Michael T. Goodrich, *"The Monotone and Planar Circuit Value Problems are Log Space for P"*, SIGACT NEWS, Volumen 9, Número 2, pp. 25-29, Diciembre 1977.

[Goodrich 93] Michael T. Goodrich, *"Parallel Algorithms Column 1:Models of Computation"*, SIGACT NEWS, Volumen 24, Número 4, pp. 16-21, Diciembre 1993.

[Hayes 89] John P. Hayes, Trevor Mudge, *"Hypercube Supercomputers"*, Proceedings of the IEEE, Vol .77, No. 12, pp. 1829-1841, diciembre 1989.

[Hernandez R. 95] Raul Hernández Cruz, *"OB-OCAM"*. Pre-impresión.Tesis en opción al grado de Maestro en Ciencias (Especialidad Ingeniería Eléctrica). CINVESTAV IPN, México, D.F.

[Hillis 86] W. Daniel Hillis y Guy L. Steel, Jr., *"Data Parallel Algorithms"*, Communications of the ACM, Vol .29, No. 9, pp. 1170-1183, diciembre 1986.

[Hopcroft 79] John E. Hopcroft y Jeffrey D. Ullman, *"Introduction to Automata Theory Languages and Computation"*, Addison Wesley, 1979.

[Horowits 78] Ellis Horowits y Sartaj Sahni, *"Fundamentals of Computer Algorithms"*, Computer Science Press, 1978.

[Jájá 92] Joseph Jájá, *"An Introduction To Parallel Algorithms"*, Addison - Wesley Publishing Company, Inc. ,1992.

[Knuth 73] Donald E Knuth, *"The Art Of Computer Programing, Vol. 3 Sorting and Searching"*, Addison - Wesley Publishing Company, Inc. ,1973.

[Knuth 86] Donald E Knuth, *"Literate Programing"*, The Computer Journal, Vol. 27, No. 2, 1986.

[Leighton 92] F, Thomson Leigthon, *"Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes "*, Morgan Kaufman Publishers, 1992.

[Loots 92] W. Loots y T.H.C. Smith, *"A Parallel three Phase Sorting Procedure For a K-Dimensional Hypercube And a Transputer Implementation"*, Parallel Computing, 18, pp. 335-244, 1992.

[Morales 87] Guillermo Morales Luna, *"Ordenamiento de sucesiones en paralelo"*, CINVESTAV, Departamento de Ingeniería Eléctrica, Serie Amarilla No.57, Abril 1987.

[Nassimi 93] David Nassimi y Yuh-Dong Tsai, *"An Efficient Implementation Of Batcher's Odd-Even Merge On a SIMD Hypercube"*, Journal Of Parallel and Distributed Computing 19, pp. 58-63, 1993.

[Polymenakos 94] L.C. Polymenakos y D.P Bertsekas, *"Paralle Shortest Path Auction Algorithms"*, Parallel Computing, Vol. 20, pp 1221-1247, 1994.

[Olmedo 87] Oscar Olmedo A., *"El Lenguaje C++ concurrente synopsis y ejemplos"*, CINVESTAV, Departamento de Ingeniería Eléctrica, Serie Verde No.40, Abril 1987.

[Quinn 84] Michael J. Quinn y Narsingh Deo, *"Parallel Graph Algorithms"*, ACM Computing Surveys, Vol. 16, No. 3, septiembre 1984.

[Quinn 90] Michael J. Quinn y Philip J. Hatcher, *"Data-Parallel Programing on Multicomputers"*, IEEE Software, septiembre 1990.

[Ranka 90] Sanjay Ranka y Sartaj Sahni, *"Hypercube Algorith with Applications to Image Processing and Pattern Recognition"*, Springer-Verlag, 1990.

[Rietman 90] Edward Rietman, *"Exploring Parallel Processing "*, Wind Crest Books, 1990.

[Rosenkrantz 77] Daniel J. Rosenkrantz, Richard E. Stearns y Philip M. Lewis II, *"An Analysis of Several Heuristics for the Traveling Salesman Problem"*, SIAM Journal on Computing, 1990.

[Shi 92] Hanmao Shi y Jonathan Schaeffer, *"Parallel Sorting by Regular Sampling"*, Journal Of Parallel And Distributed Computing 19, No. 4, abril 1994.

[Schmidt 94] M. Schmidt Voigt, *"Efficient Parallel Communication with the nCUBE 2s Processor"*, Parallel Computing, Vol. 20, No. 4, abril 1994.

[Schneier 94] Bruce Schneier, *"NP-Completeness"*, Dr. Dobb's Journal, septiembre 1994.

[Stroustrup 86] Bjarne Stroustrup, *"The C++ Programming Languaje"*, Addison Wesley, 1986.

[Suaya 90] Robert Suaya, Graham Birtwistle, *"VLSI And Parallel Computation"*, Morgan Kaufmann Publishers, 1990.

[Takeuchi 92] Aikuzo Takeuchi, *"Parallel Logic Programming"*, John Wiley & Sons, 1992.

[Tick 92] Evan Tick, *"Parallel Logic Programming"*, Massachusetts Institute of Technology, 1991.

[Treleaven 90] P.C. Treleaven, *"Parallel Computers: Object-Oriented, Functional, logic"*, Wiley Series in Parallel Computing, 1990.

[Tuthill 90] Steven G. Tuthill and Susan T. Levy, *"Knowledge-Based Systems"*, TAB Professional and Reference books, Mc Graw Hill, 1990.

[Waldrop 88] Mitchell M. Waldrop, *"Hypercube Breaks a Programming Barrier"*, Science, Vol. 240, iss 4850, pp. 286-286, 1988.

[Whiting 94] Paul G. Whiting y Robert S. V. Pascoe, *"A History of Data-Flow Languajes"*, IEEE Annals of the History of Computing, Vol. 16, Número 4, 1994.

[Wolfram 86] Stephen Wolfram, *"Theory and applications of Cellular Automata"*, Word Scientifilc Publishing, 1986.

[Zima 91] Hans Zima, Barbara Chapman, *"Supercompilers for Parallel and Vector Computers"*, ACM Press Frontier Series, 1991.

Los abajo firmantes, integrantes de jurado para el examen de grado que sustentará el **Lic. Carlos Alberto Hernández Hernández**, declaramos que hemos revisado la tesis titulada.

"Algoritmos de ordenamiento y teoría de gráficas en una computadora paralela de tipo hipercubo"

y consideramos que cumple con los requisitos para obtener el grado de Maestro en Ciencias, con especialidad en Ingeniería Eléctrica.

Atentamente

Dr. Sergio Victor Chapa Vergara

M. en C. José Oscar Olmedo Aguírre

M. en C. Alejandro Tinoco Alvarado

AUTOR HERNANDEZ HERNANDEZ, C.A.

TITULO ALGORITMOS DE ORDENAMIENTO Y TEORIA DE GRAFICAS EN UNA COMPUTADORA PARALELA ...

CLASIF. XM 95.27 RGTRO. BI-14677

NOMBRE DEL LECTOR	FECHA PREST.	FECHA DEVOL.
Marco Nuños	21/6/96	17/7/96
Sofía	6/3/97	3/Abr./97
Fotocopiado - Rep. Tec.	2 may 97	6 may 97
Gonzalo Avila R.	25/08/97	28/8/97
Cesar Glez Serrano	8/Abr/99	15-IV-99