Análisis comparativo de la ejecución del algoritmo voraz de PRIM en modo lineal y paralelo (LAM-MPI)

José Márquez Díaz*, José David Cortés**, Alex de Moya***

Resumen

En este trabajo se muestra cómo funciona el algoritmo voraz PRIM, *tanto en ambientes paralelos como en secuenciales. El objetivo de estas pruebas es ver cómo la herramienta* mpi *puede trabajar en red y mostrar cómo los algoritmos voraces en la fase de ejecución pueden arrojar resultados satisfactorios al momento de utilizarlos. Para poder ejecutar el algoritmo voraz* PRIM *se usó un entorno de procesamiento secuencial utilizando un único computador y un entorno de procesamiento paralelo a partir de la implementación dada con la programación de paso de mensajes (*MPI*), denominada* LAM *(Local Area Multicomputer) sobre el sistema operativo Linux. Los resultados obtenidos permiten concluir que a través del paralelismo virtual se logra disminuir el tiempo de procesamiento de un programa de esta naturaleza.*
Palabras clave: Linux, MPI, paralelismo, PRIM, paso de mensajes.

Abstract

In this paper we are going to explain how greedy algorithm PRIM *works, in both parallel and sequential environment. The objective of these tests is to see how can the* MPI *tool work in network and show how can the greedy algorithms in the execution phase yield satisfactory results at the time of using them. In order to be able to execute greedy algorithm* PRIM, *a sequential processing environment was used with an only computer and parallel processing environment comes from the implementation given with the Messages Passing Programming (*MPI*), denominated* LAM *(Local Area Multicomputer) on the operating system Linux. The obtained results allow us to conclude that through virtual parallelism, the time of processing of a program of this nature can be decreased.*
Key words: Linux, MPI, parallelism, PRIM, messages passing.

Fecha de recepción: 1 de septiembre de 2003
Fecha de aceptación: 31 de octubre de 2003

* Ingeniero de Sistemas, Universidad del Norte; Magíster en Ciencias Computacionales, ITESM-UNAB-CUTB. Profesor del Departamento de Ingeniería de Sistemas de la Universidad del Norte. *jmarquez@uninorte.edu.co.*

** Estudiante de Ingeniería de Sistemas, Universidad del Norte. *cortesj@unimail.uninorte.edu.co.*

*** Ingeniero de Sistemas, Universidad del Norte. *ajdemoya@msn.com.*

1. INTRODUCCIÓN

Una máquina paralela virtual es la implementación de un computador paralelo usando una red, que puede estar compuesta de máquinas paralelas reales, microcomputadores o cualquier otro tipo de computador. La comunicación en la máquina paralela se realiza a través de un protocolo común mediante el cual se comparten datos, se asignan tareas y se coordina todo el procesamiento.

El modelo de programación que subyace tras una máquina paralela virtual es MIMD (*Multiple Instruction Multiple Data*), aunque se dan especiales facilidades para la utilización del modelo SIMD (*Single Instruction Multiple Data*), un caso especial de MIMD en el que todos los procesos ejecutan el mismo programa, aunque no necesariamente la misma instrucción al mismo tiempo.

El paso de mensajes es una aplicación de máquina paralela virtual en la que cada computador de la red ejecuta su propio programa, el cual puede acceder a la memoria local o a memorias de otros computadores mediante el paso de mensajes en la red. En una red apropiada para el procesamiento paralelo se necesitaría, idealmente, que el costo de enviar mensajes entre dos computadores sea independiente de la distancia que los separa y el tráfico de la red, pero sí dependería de la longitud del mensaje.

En el procesamiento paralelo a través de redes, el acceso a memorias de otros computadores se hace con una menor rapidez que a la memoria local; la diferencia en velocidad puede ser de dos o tres órdenes de magnitud, dependiendo de la velocidad del computador, la red y cualquier otro dispositivo que se encuentre entre ellos.

En una computación paralela, el número de tareas puede variar durante la ejecu- ción; éstas poseen su propio programa secuencial, memoria y datos para trabajar, además posee un conjunto de puertos de entrada y salida de interfaces mediante los cuales interactúan con las demás tareas.

En el esquema de procesamiento paralelo a través de redes, adicionales a las operaciones de leer y escribir en memoria local, se utilizan cuatro primitivas básicas para coordinar y controlar las tareas en que se subdividen los procesos:

- **Enviar.** Mediante esta primitiva las tareas envían mensajes a otras. El envío de mensajes se lleva a cabo en forma asíncrona, es decir, se completa apenas se ha enviado el mensaje.
- **Recibir.** Consiste en obtener un mensaje de otra tarea, y se lleva a cabo de forma síncrona; es decir, hace que se quede esperando hasta recibir el mensaje.

- **Crear.** Por medio de esta primitiva las tareas tienen la facultad de crear otras tareas.
- **Terminar.** Esta primitiva es utilizada para finalizar una tarea.

2. DISEÑO DE UN ALGORITMO PARALELO

En el diseño de un algoritmo paralelo existen las siguientes cuatro etapas: **particionamiento, comunicación, aglomeración** y **asignación**; esta metodología se conoce como PCAM (*Partitioning, Communication, Aglomeration, Mapping*). Ver figura 1.

Particionar significa tomar el problema y dividirlo en pedazos que puedan ser aptos para aplicarse paralelamente de forma independiente a través de tareas; estas tareas deben comunicarse entre sí para lograr lo que se espera, e incluso pueden agruparse para reducir el tamaño del problema. Al poseer una estructura de paralelismo de arquitecturas heterogéneas puede ser bastante productivo determinar de antemano en cuál *host* se pueden ejecutar ciertas tareas para lograr un óptimo rendimiento.

2.1. Particionamiento

Cuando se piensa en programación paralela se debe pensar en descomposición; si se quiere diseñar de forma eficiente un algoritmo que se ejecute bajo una máquina paralela, se obtienen mejores resultados si se orientan los procesos en el programa como un conjunto integrado de tareas independientes que cooperan entre sí y se asignan esas tareas para un rendimiento óptimo de todos los procesadores que conformen la máquina virtual.

Si, por el contrario, se ve el algoritmo como un único bloque integrado y se establecen dependencias inquebrantables entre sus partes, muy probablemente se desperdiciarán recursos en estado de espera de otros procesos. Para evitar lo anterior existen dos métodos reconocidos para paralelizar algoritmos, descomposición orientada a los datos y descomposición orientada a la función. Pueden aplicarse de forma independiente o complementaria sobre un mismo algoritmo, ambos de acuerdo con el tipo de problema que se vaya a resolver.

2.1.1. *Descomposición orientada a datos*

La descomposición orientada a datos trata de fraccionar los datos de entrada del problema en pequeños pedazos que pueden ser tratados independientemente. Se aplica cuando el problema es divisible sin que las partes dependan mucho unas de otras para llegar a la solución.

2.1.2. *Descomposición orientada a la función*

Esta consiste en descomponer las tareas de computación que se van a llevar a cabo para ejecutarlas en paralelo; por ejemplo, las tareas básicas de un algoritmo pueden ser entrada, proceso y salida. Estas tareas pueden manipularse de forma independiente y paralelizarse de forma que compartan datos entre sí.

El enfoque funcional se aplica cuando los datos están muy compenetrados entre sí y no pueden descomponerse, o complementada por la descomposición orientada a datos en cada una de las diferentes etapas operativas en las que se dividió el algoritmo.

La descomposición orientada a la función es otra forma de afrontar los problemas, ya que no se piensa en dividir los datos de entrada, sino que se trata de clasificar la operación general del algoritmo en unidades no dependientes, que sean susceptibles de ejecutarse al mismo tiempo.

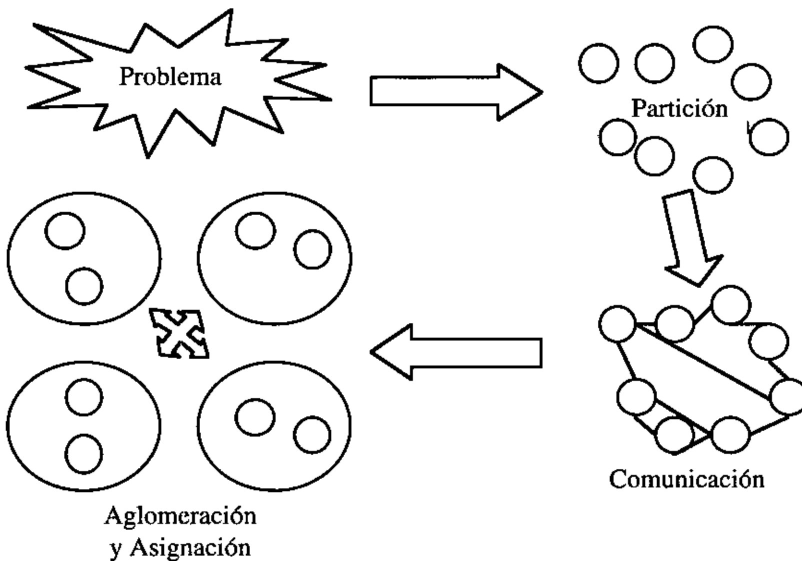

Figura 1. Etapas del proceso de programación paralela

2.2. **Comunicación**

Cuando se particiona se pretende dividir el problema para colocar tareas que se puedan ejecutar al mismo tiempo, pero no necesariamente esas tareas tienen que ser independientes. Cuando esas tareas tienen dependencias entre sí hay que comunicarlas, con el fin de compartir datos necesarios para la computación que cada una está llevando a cabo. En el caso de una descomposición orientada a

datos puede ser difícil definir cómo va a ser la comunicación entre los diferentes módulos, ya que las dependencias no siempre son evidentes, precisamente por el hecho de haber trabajado con conjuntos disjuntos; por el contrario, en el caso de descomposición funcional, la comunicación se da de forma natural a través de flujo de datos entre las diferentes etapas en que se dividió el proceso.

2.3. Aglomeración y Asignación

Una vez se han definido las tareas que se van a ejecutar y la forma como se va a proveer de datos a esas tareas, se debe definir la forma en que se van a agrupar. **Aglomerar** tiene que ver con definir exactamente el entorno en que se van a ejecutar las tareas y el número de tareas que se ajuste más eficientemente al número de procesadores con el que se cuenta. Al mismo tiempo, se reducen las comunicaciones al mínimo aceptable en el que se puedan compartir todos los datos necesarios *vs.* disminución en el rendimiento por la velocidad del medio físico de comunicación.

Por último, al llevar a cabo la **Asignación** se define en detalle en qué procesador conviene ejecutar las diferentes tareas, con lo cual se logra un equilibrio entre lo que conviene ejecutar localmente y lo que conviene ejecutar remotamente (en los demás procesadores) para lograr concurrencia en el desarrollo del problema.

3. ALGORITMO DE PRIM

El algoritmo de Prim básicamente consiste en elegir el conjunto de arcos de un grafo de manera que conecten todos los vértices y que la suma de sus costos sea la menor posible. Sus características más importantes son:

- El algoritmo de Prim halla un árbol de recubrimiento mínimo de un grafo G = <N,A>.
- El ARM (árbol de recubrimiento de costo mínimo) va creciendo empezando por una raíz arbitraria.
- En cada paso se añade una nueva rama al árbol.
- El algoritmo termina cuando se ha llegado a todos los nodos.
- Se mantiene B, conjunto de nodos, y T, conjunto de aristas.
- Inicialmente B contiene un único nodo y T está vacío.
- En cada paso se busca la arista más corta posible $< u, v >$ que sale de B, es decir, tal que u pertenece a B y v *no pertenece a* B. Se añade v a B y $< u,v >$ a T.
- En cada momento T es un ARM para los nodos de B. Cuando $B = N$, T es un ARM para G.

```
Función Prim(G[1..n, 1..n]) dev T
    T := 0
    para i = 2 hasta n hacer
        más-próximo[i] := 1 ; dist-mínima[i] := G[1, i]
    fpara
    repetir n - 1 veces
        min := +∞
        para j = 2 hasta n hacer
            si 0 <= dist-mínima[ j] <= min entonces
                min := dist-mínima[ j] ; k := j
            fsi
        fpara
        T:=T U < más-próximo[k], k> ;   dist-mínima[k] := -1

    paraj = 2 hasta n hacer
        si G[k, j] < dist-mínima[ j] entonces
            dist-mínima[ j] := G[k, j]
            más-próximo[ j] := k
        fsi
    fpara
  frepetir
F-Función
```

Complejidad: $O(n^2)$, *n nodos.*

El bucle principal del algoritmo se ejecuta n-1 veces; en cada iteración el bucle **para** anidado requiere un tiempo que está en $\Theta(n)$. Por tanto, el algoritmo de Prim requiere un tiempo que está en $\Theta(n^2)$.

En muchos casos se hace necesario obtener las rutas y los costos mínimos partiendo de todos los nodos del grafo y no de un nodo en especial. Para afrontar este problema puede aplicarse el algoritmo de Prim partiendo de cada uno de los oríge-nes, por lo que su complejidad se modifica a $O(N^3)$.

Para mostrar la forma en que trabaja este algoritmo, considérese el grafo de la figura 2. Por orden creciente de longitud, las aristas son (ver tabla 1):
<1,2>,<2,3>,<4,5>,<6,7>,<1,4>,<2,5>,<4,7>,<3,5>,<2,4>,<3,6>,<5,7>,<5,6>.

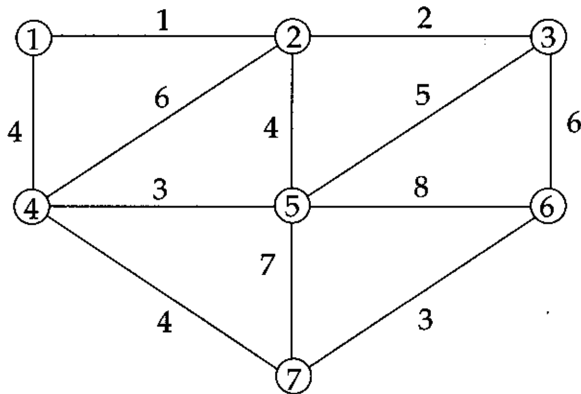

Figura 2. Grafo de ejemplo para el seguimiento del algoritmo de Prim

Tabla 1
Seguimiento del algoritmo de Prim

Paso	Arista	Componentes conexas
Inic.	-	{1}
1	<1,2>	{1,2}
2	<2,3>	{1,2,3}
3	<1,4>	{1,2,3,4}
4	<4,5>	{1,2,3,4,5}
5	<4,7>	{1,2,3,4,5,7}
6	<7,6>	{1,2,3,4,5,6,7}

4. MPI

MPI (Interfaz de Paso de Mensajes) es un estándar para la implementación de sistemas de paso de mensajes desarrollado por un comité de proveedores, laboratorios del gobierno y universidades para funcionar en una amplia variedad de computadores paralelos y de forma tal que los códigos sean portables. Su diseño está inspirado en máquinas con una arquitectura de memoria distribuida (ver figura 3), en donde cada procesador es propietario de cierta memoria y la única forma de intercambiar información es a través de mensajes; sin embargo, hoy en día también se encuentran implementaciones de MPI en máquinas de memoria compartida y redes de estaciones de trabajo. En pocas palabras, MPI representa un esfuerzo de la comunidad de programación paralela por estandarizar las subrutinas de comunicación en computadores de cómputo masivo.

MPI no exige una determinada implementación del mismo. Lo importante es dar al programador una colección de funciones para que éste diseñe su aplicación, sin que tenga necesariamente que conocer el *hardware* concreto sobre el que se va a ejecutar, ni la forma en la que se han implementado las funciones que emplea.

4.1. **Objetivos**[5]

- Diseñar una interfaz de programación aplicable (API).
- Hacer eficiente la comunicación. Evitando copiar de memoria a memoria y per-mitiendo (donde sea posible) la sobreposición de computación y comunicación, además de aligerar la comunicación con el procesador.
- Permitir implementaciones que puedan ser utilizadas en un ambiente heterogéneo.
- Soportar conexiones de la interfaz con C y Fortran 77, con una semántica independiente del lenguaje.
- Asumir un interfaz de comunicación confiable. El usuario no debe lidiar con fallas de comunicación. Tales fallas son controladas por el subsistema de comunicación interior.
- Definir un interfaz que no sea muy diferente a los actuales, tales como PVM, NX, Express, etc., y proveer de extensiones para permitir mayor flexibilidad.
- Definir un interfaz que pueda ser implementado en diferentes plataformas, sin cambios significativos en el *software* y las funciones internas de comunicación.
- La semántica del interfaz debe ser independiente del lenguaje.
- La interfaz debe ser diseñada para producir procesos seguros.

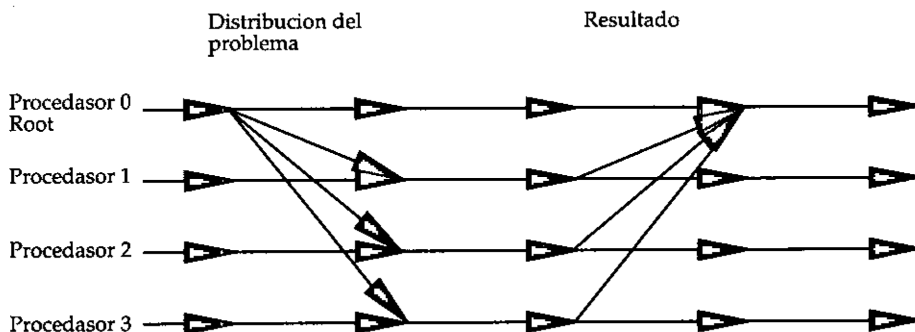

Figura 3. Modelo de Memoria distribuida

4.2. Características de MPI

A continuación se describen algunas de las principales características que ofrece MPI:

- **Operaciones colectivas.** Una operación colectiva es una operación ejecutada por todos los procesos que intervienen en un cálculo o comunicación. En MPI existen dos tipos de operaciones colectivas: ***operaciones de movimiento de datos*** y ***operaciones de cálculo colectivo***. Las primeras se utilizan para intercambiar y reordenar datos entre un conjunto de procesos. Un ejemplo típico de operación colectiva es la difusión (*broadcast*) de un mensaje entre varios procesos. Las segundas permiten realizar cálculos colectivos como mínimo, máximo, suma, OR lógico, etc., así como operaciones definidas por el usuario.

- **Topologías virtuales.** Ofrecen un mecanismo de alto nivel para manejar grupos de procesos sin tratar con ellos directamente. MPI soporta grafos y redes de procesos.

- **Modos de comunicación.** MPI soporta ***operaciones bloqueantes, no bloqueantes*** o ***asíncronas*** y ***síncronas***. Una operación de envío bloqueante bloquea al proceso que la ejecuta sólo hasta que el *buffer* pueda ser reutilizado de nuevo. Una operación no bloqueante permite solapar el cálculo con las comunicaciones. En MPI es posible esperar por la finalización de varias operaciones no bloqueantes. Un envío síncrono bloquea la operación hasta que la correspondiente recepción tiene lugar. En este sentido, una operación bloqueante no tiene por qué ser síncrona.

- **Soporte para redes heterogéneas.** Los programas MPI están pensados para poder ejecutarse sobre redes de máquinas heterogéneas con formatos y tamaños de los tipos de datos elementales totalmente diferentes.

- **Tipos de datos.** MPI ofrece un amplio conjunto de tipos de datos predefinidos (caracteres, enteros, números en coma flotante, etc.) y ofrece la posibilidad de definir tipos de datos derivados. Un tipo de datos derivado es un objeto que se construye a partir de tipos de datos ya existentes. En general, un tipo de datos derivado se especifica como una secuencia de tipos de datos ya existentes y desplazamientos (en *bytes*) de cada uno de estos tipos de datos. Estos desplazamientos son relativos al *buffer* que describe el tipo de datos derivado.

- **Modos de programación.** Con MPI se pueden desarrollar aplicaciones paralelas que sigan el modelo SPMD o MPMD. Además el interfaz MPI tiene una

semántica multithread (*MT-safe*), que le hace adecuado para ser utilizado en programas MPI y entornos multithread.

4.3. LAM

LAM (del inglés *Local Area Multicomputer*) es una implementación del estándar MPI, pensado para redes de trabajo con computadores independientes, lo que trae como implicaciones que el código fuente de cualquier aplicación desarrollada bajo LAM sea portable a cualquier otra implementación de MPI. Además, con el fin de proveer depuración (*debugging*) de aplicaciones, LAM permite monitorear el desempeño de las mismas. Este monitoreo se realiza en dos niveles: en un primer nivel LAM permite «fotografiar» el estado de los procesos y los mensajes a medida que una aplicación está corriendo; y en un segundo nivel, la librería MPI produce un registro acumulado de la comunicación en una aplicación; tal registro puede ser visualizado en tiempo de ejecución.

LAM se ejecuta en cada computadora como un demonio de UNIX y se encuentra estructurado por un nano-kernel y un conjunto de procesos o subsistemas que proveen servicios específicos. El nano-kernel provee paso de mensajes simples, y servicio de *rendez-vous* para procesos locales; mientras que los subsistemas permiten comunicación (vía UDP) entre los distintos nodos de la red, empaquetamiento y *buffering* para realizar sincronización, ejecución remota de programas y acceso remoto de archivos. Algunos de estos servicios pueden ser configurados por el usuario.

5. IMPLEMENTACIÓN DEL ALGORITMO DE PRIM SECUENCIAL Y PARALELO

5.1. **Algoritmo de Prim Secuencial**

Las estadísticas obtenidas a partir de las pruebas de este algoritmo fueron las siguientes (ver tabla 2 y figura 4):

Tabla 2
Datos estadísticos de Prim Secuencial

N	Tiempo
1000	88 s , 625 min
500	10 s , 613 min
250	1 s , 145 min
100	0 s , 110 min

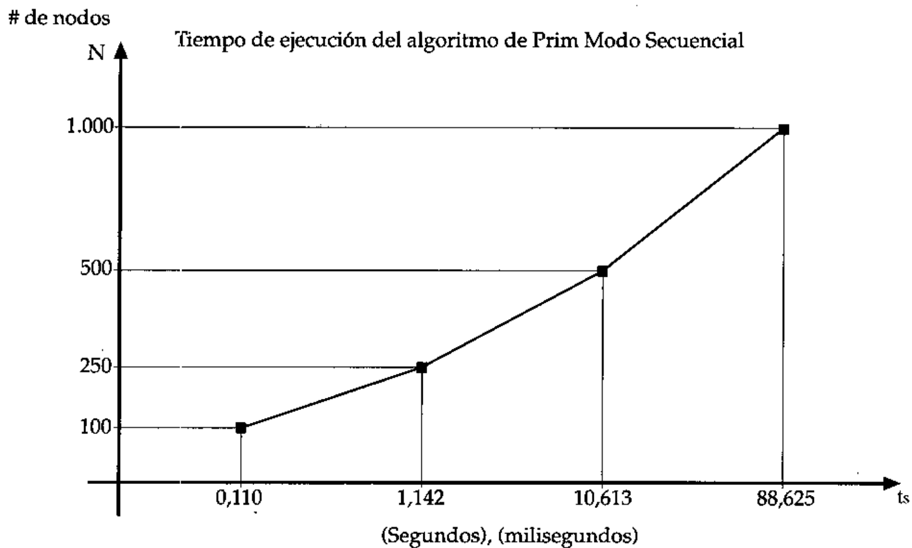

Figura 4. Gráfica del tiempo de ejecución de Prim Secuencial

5.2. **Algoritmo de Prim Paralelo**

Para realizar la ejecución de este algoritmo se estructuró una red con 3 computadores, conectados a través de un concentrador. Las especificaciones técnicas de los computadores utilizados en esta prueba son las siguientes (ver tabla 3):

Tabla 3
Especificaciones de *hardware* de los computadores

Características\CPU y RAM	MAESTRO	ESCLAVO 1	ESCLAVO 2
Nombre del modelo	Pentium III	Pentium II	Pentium II
CPU MHZ	501.134	349.187	349.187
RAM	64 MB	64 MB	64 MB

Para medir los tiempos de respuestas de este algoritmo se prepararon pruebas con dos y tres computadores.

- **Esquema con dos computadores** (ver figura 5): el esquema general de la implementación del algoritmo de Prim en forma paralela es la siguiente:

  - El algoritmo de Prim trabaja con un grafo de N nodos, el cual es representado por una matriz, llamada la matriz de adyacencia. Esta contiene

información acerca de la longitud entre los nodos y si de un nodo a otro nodo hay camino. Esta matriz de adyacencia es enviada a cada esclavo junto con la posición de inicio y la cantidad de datos que se va a procesar; enseguida estos procesos se ejecutan paralelamente (ver figura 5). Las instrucciones para realizar estos envíos son las siguientes:

```
MPI_Isend(&vp, 1, MPI_INT, i, tag, MPI_COMM_WORLD, & request);
MPI_Isend(&mat[0][0], a*a, MPI_INT, i, tag, MPI_COMM_WORLD, &request);
```

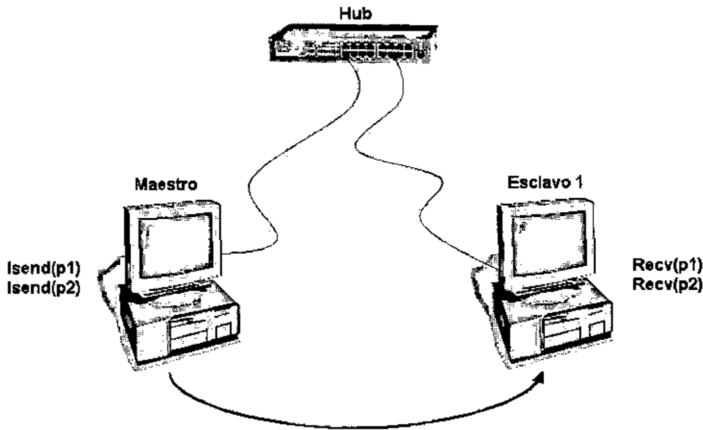

Figura 5. Esquema de la máquina virtual para dos computadores

- Después que el esclavo termine su trabajo enviará un vector con los resultados obtenidos durante la ejecución, el cual contiene la suma mínima para poder conectarse con todos los nodos (ver figura 6). Esto también se puede realizar con más de dos computadores.

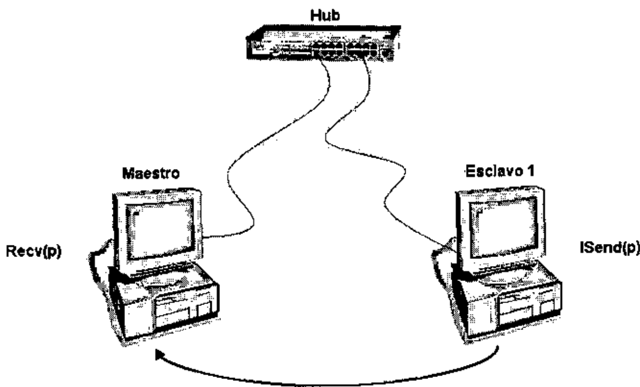

Figura 6. Envío de la respuesta por parte del esclavo al maestro

- Las estadísticas obtenidas en las pruebas fueron las siguientes (ver tabla 4 y figura 7):

Tabla 4

Datos estadísticos de Prim Paralelo en dos computadores

N	Tiempo
1000	86 s , 194 min
500	11 s , 948 min
250	1 s , 428 min
100	0 s , 117 min

- **Esquema con tres computadores:** se presentarán los resultados de las pruebas obtenidas de la ejecución del algoritmo de Prim en tres computadores en red, donde los pasos de la ejecución de este algoritmo son similares a las pruebas que se llevaron a cabo con dos computadores (ver figuras 8 y 9).

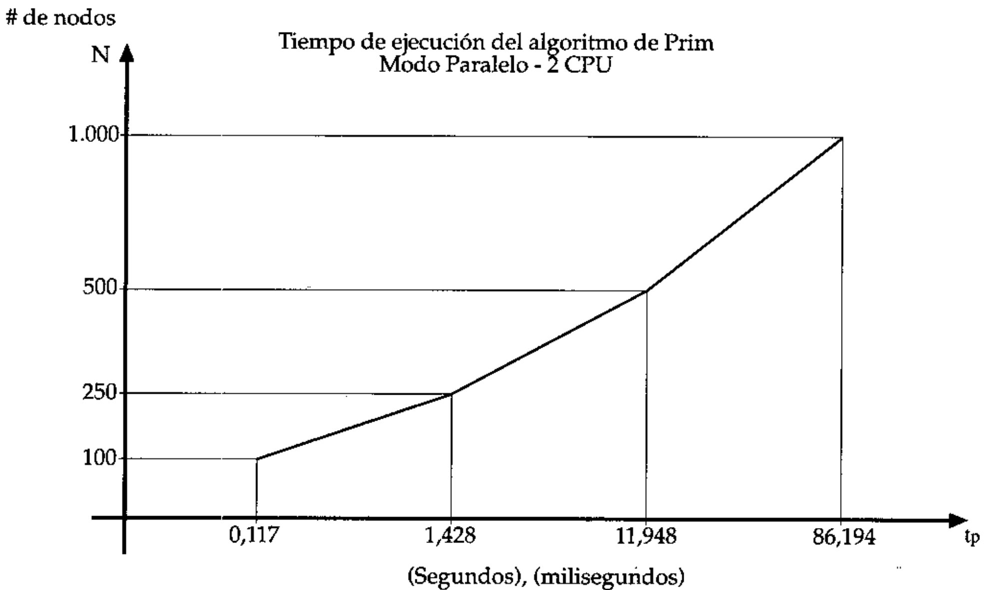

Figura 7. Gráfica del tiempo de ejecución de Prim Paralelo en dos computadores

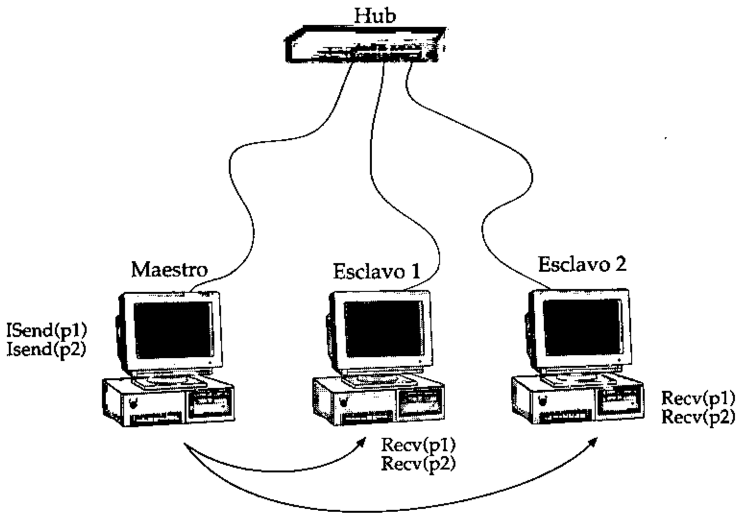

Figura 8. Envío de datos Maestro – Esclavos

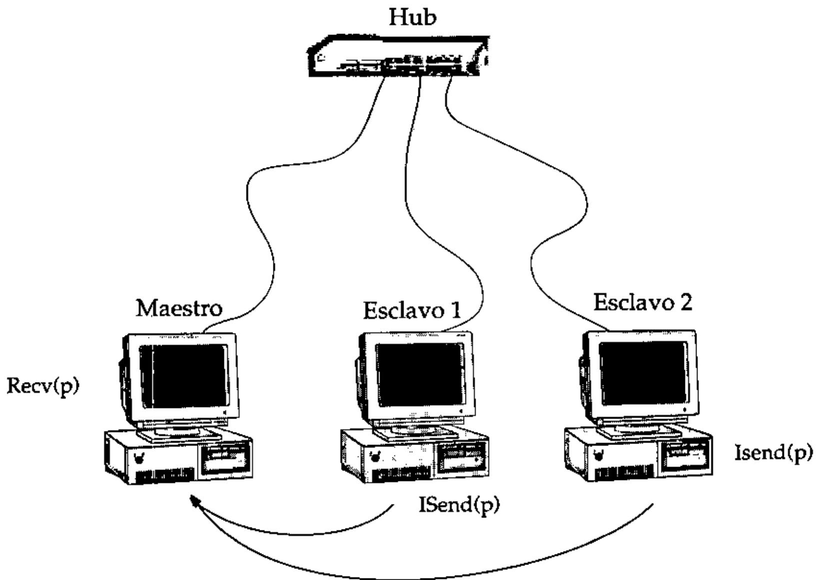

Figura 9. Envío de las respuestas por parte de los esclavos al maestro

Las estadísticas obtenidas en las pruebas fueron las siguientes (ver tabla 5 y figura 10):

Tabla 5
Datos estadísticos de Prim Paralelo en tres computadores

N	Tiempo
1000	66 s , 128 min
500	9 s , 354 min
250	1 s , 349 min
100	0 s , 163 min

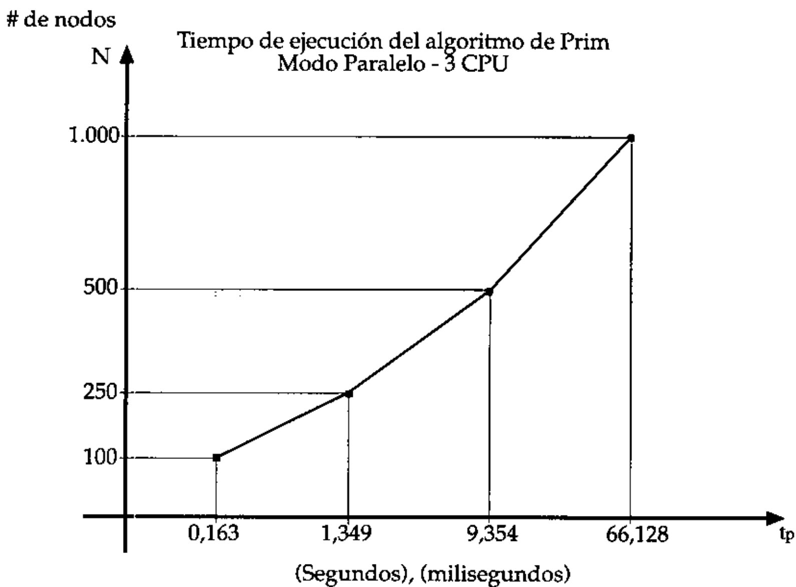

Figura 10. Gráfica del tiempo de ejecución de Prim Paralelo en tres computadores

Para obtener una disminución en los tiempos de ejecución fue necesario aplicar la técnica de paralelización orientada a los datos, ya que la misma función Prim se aplica para todos los nodos. Entonces el problema original se subdivide en *m* tareas que calculan independientemente los caminos óptimos para un subconjunto asignado del grupo inicial de nodos, envían sus resultados a una tarea *maestra*, y ésta los recopila en un vector para formar la respuesta al problema inicial.

5.3. **Comparación**

Ahora se observará una gráfica (ver figura 11) donde se presenta la comparación entre el algoritmo de Prim en forma secuencial, en forma paralela con dos y tres computadores.

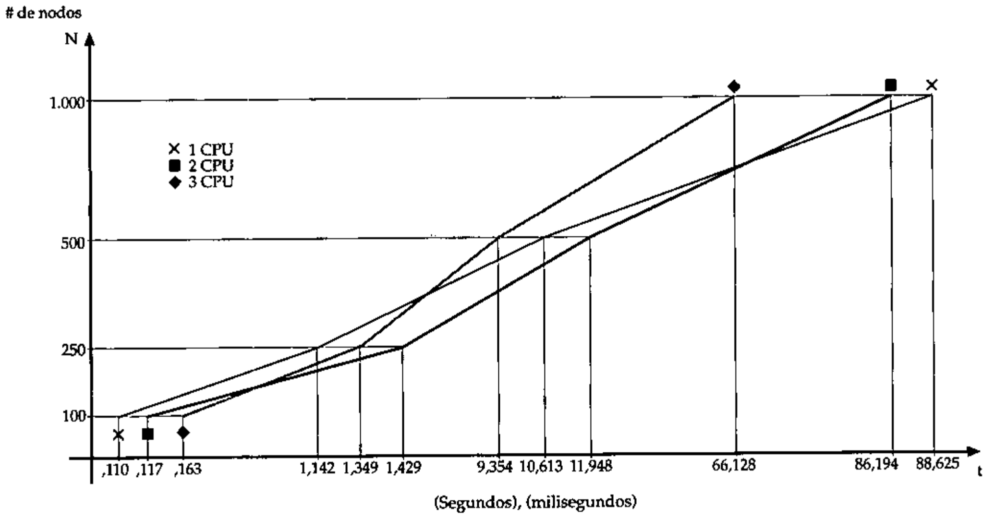

Figura 11. Análisis de Rendimiento Versión Lineal vs. Paralela

5.4. Resultados

Según las pruebas desarrolladas se obtienen los siguientes resultados:

- Es evidente que la eficiencia de usar la herramienta MPI es más notable cuando hay una entrada mayor de datos que cuando es una cantidad menor de éstos.

- Las características de *hardware* son pieza fundamental para mejorar la eficiencia de estos algoritmos; es decir, entre más alto rendimiento presenten los computadores que harán parte de la máquina virtual paralela, mayor será el rendimiento.

- Si este algoritmo, se hubiese calculado para un nodo en particular, tendría el mismo problema que se detectó en el algoritmo de la mochila, ya que su estructura algorítmica es voraz, y por lo tanto difícil de paralelizar. Pero como se implementó para **n** nodos, a cada procesador se le asignó cierta cantidad de nodos (**n/número de procesadores**), y esto permitió que se pudiera implementar paralelamente.

CONCLUSIONES

Desde la fase de planificación hasta la fase de ejecución este proyecto se ha presentado como fuente de conocimiento teórico-práctico sobre los beneficios de la simulación del paralelismo sobre redes de computadores.

Las técnicas para paralelizar algoritmos permiten ejecutar eficaz y eficientemente procesos complejos que requieren gran consumo de poder de cómputo, disgregándolos en trabajos abordables por distintas tareas cooperativas que, por consiguiente, aceleran el tiempo de respuesta de los sistemas.

El uso de procesamiento en paralelo sobre redes de computadores es una excelente alternativa para aprovechar los recursos de las máquinas que ya existen y, por ende, aumentar la capacidad de cómputo.

MPI es una muy buena herramienta para la implementación de procesamiento paralelo en redes de computadores; sin lugar a dudas es un excelente soporte para la implementación de programas que aprovechen esta arquitectura de cómputo por su eficacia, eficiencia y fácil uso de las funciones que ofrece aún en su versión gratis.

Paralelizar los algoritmos voraces es una tarea muy difícil de efectuar, porque su estructura interna no permite una divisibilidad de sus instrucciones, además éstas son muy dependientes entre sí.

Al efectuar la distribución de procesos resulta más eficaz abordarla orientada a los datos y no por división funcional, debido a que la mayoría de veces ofrece más independencia a las tareas involucradas.

Referencias

[1] AKL, Selim G., *Parallel Computation: Models and Methods.* Prentice-Hall, 1997. 608 p.

[2] ________ *The desing and Analysis of parallel Algorithms.* Prentice-Hall, 1989. 401 p.

[3] BRASSARD, G. & BRATLEY, P., *Fundamentos de Algoritmia.* Madrid: Prentice-Hall, 1997. 608 p.

[4] CORMEN, Thomas, LEISERSON, Charles & RIVEST, Ronald, *Introduction to Algorithms.* Estados Unidos: McGraw-Hill, 1990. 1028 p.

[5] HOEGER, Herbert & HIBRODO, Francisco, *Introducción a MPI (Message Passing Interface).* Mérida (Venezuela): CeCalCULA, 1998. 57 p.

[6] TACKETT, Jack Jr, *Edición Especial Linux,* 4ª Ed. Madrid: Prentice-Hall, 2000. 1110 p.

[7] http://nexus.cs.usfca.edu/mpi/, Agosto de 2002.

[8] http://www.cecalc.ula.ve/documentacion/manuales_tutoriales.html, marzo de 2002

[9] http://www.cnb.uam.es/~carazo/practica_mpi.html, mayo de 2002

[10] http://www.lam-mpi.org/ , octubre de 2001

[11] http://www.mpi-forum.org/, octubre de 2001
[12] http://www.netlib.org/mpi/ , noviembre de 2001
[13] http://www-unix.mcs.anl.gov/mpi/, enero de 2002,
[14] http://yesca.alcd.net/cluster/node56.html, enero de 2002.