Proyecto

"Estudio Experimental del Efecto del Paso de Mensajes en Ambiente GRID para el Desarrollo de Sistemas que Tratan con Problemas NP-Completos"

REPORTE FINAL

Colaboradores en el proyecto

Universidad Autónoma del Estado de Morelos
Centro de Investigación en Ingeniería y Ciencias Aplicadas
Investigador Líder
Dr. Marco Antonio Cruz Chávez
Investigadores
Dr. Martín Heriberto Cruz Rosales
Estudiantes
Fredy Juárez Pérez
Erika Yesenia Ávila Melgar
Alina Martínez Oropeza
Pedro Moreno Bernal
David Flores Cuevas

Departamento de Sistemas y Computación
Investigador Principal
Dr. Abelardo Rodríguez León
Investigadores
M.C. Rafael Rivera López
Estudiantes
Jonathan Damaro Lope Torres
Gabriela Pérez Reyes
Rodrigo Eugenio Morales Navarro

Universidad Politécnica del Estado de Morelos
Departamento de Ingeniería en Informática
Investigador Principal
C. Dr. Irma Yazmín Hernández Báez
Estudiantes
Roberto Estrada Alcázar
René López Ruiz
Wendy Torres Manjarrez

1. Objetivos

General

Diseñar, implementar y evaluar un modelo de paralelismo de tres niveles (SMP, Cluster y Grid) con paso de mensajes eficiente, aplicado a algoritmos genéticos que resuelvan problemas duros de tipo NP-Completos.

Específicos

Definir tipos de estructuras de datos en algoritmos, adecuadas para implementar en los algoritmos desarrollados por el grupo de trabajo, ruteo de vehículos, calendarización de sistemas de manufactura.

Diseñar e implementar un algoritmo con paralelismo a nivel de SMP y de cluster, para los dos tipos de problemas seleccionados, tomando en cuenta el tipo de estructura de datos adecuada para el paso de mensajes.

Diseñar e implementar un algoritmo con paralelismo a nivel Grid para los dos tipos de problemas NP-Completos definidos.

Evaluar y comparar en forma separada cada uno de los algoritmos paralelos desarrollados

Integrar los algoritmos desarrollados en los niveles anteriores.

Evaluar los resultados de la versión integrada.

2. Actividades desarrolladas

2.1. Medida de latencias en la Grid Tarántula (UAEM-ITVer-UPEMOR)

Las pruebas de latencia sin carga se realizaron en la grid para los clusters CIICAp y UPEMOR. Las figura 1 a la 5, presentan el comportamiento de la latencia en hora pico de 8 a 11 hrs. En estas pruebas no existe carga, es decir se evito el uso de comunicación entre los clusters por otros procesos, únicamente se ejecuto el programa que mide las latencias. Se observa que el lunes la latencia aumenta en forma considerable entre 600 y 900 ms a las 11 hrs. De 8 a 10 hrs., la latencia es constante y no pasa de 200ms. Un comportamiento parecido se observa el día martes. Los días miércoles y viernes no se tiene un comportamiento definido. El viernes conforme avanza el día, la latencia va en aumento de 700 a cerca de 4200 ms. La figura 6., presenta la medición de la latencia en promedio de toda la semana en hora pico y sin carga en comunicación de procesos entre clusters de la grid. En esta figura, se observa que la latencia aumenta en promedio conforme pasa el día y el intervalo de aumento va desde 500 a 2500 ms.

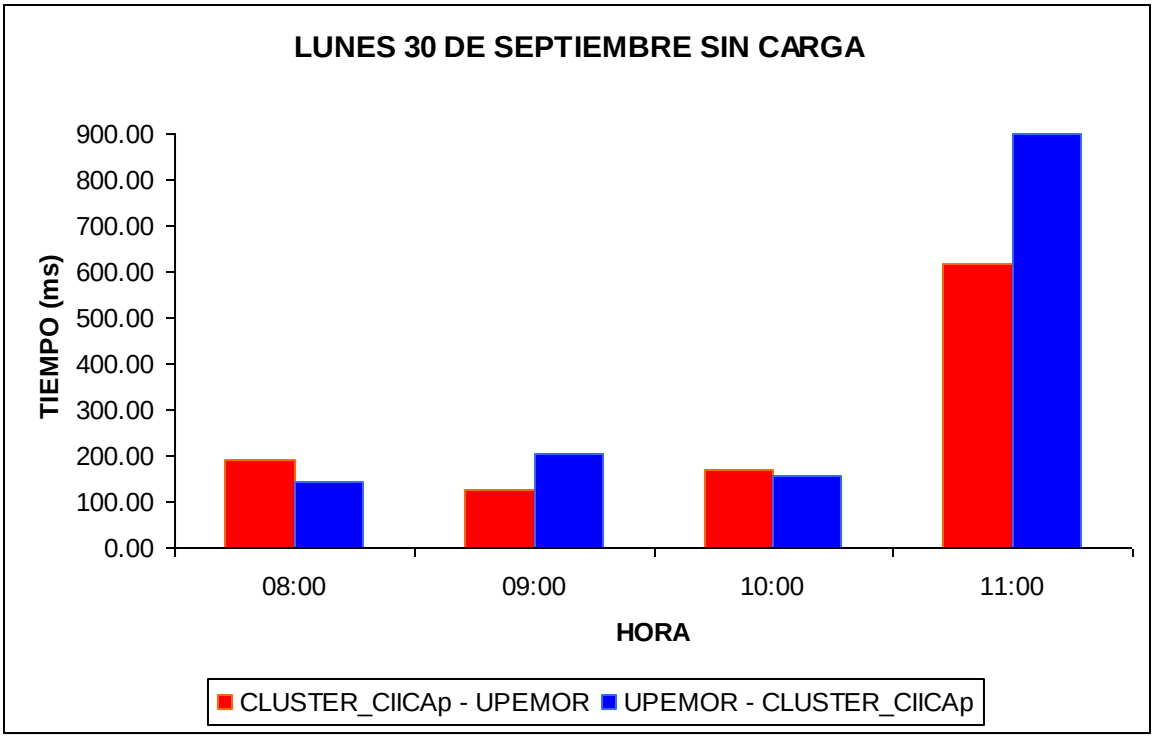


Figura 1. Pruebas de latencia en grid sin carga en hora pico, lunes 30 de septiembre**.**

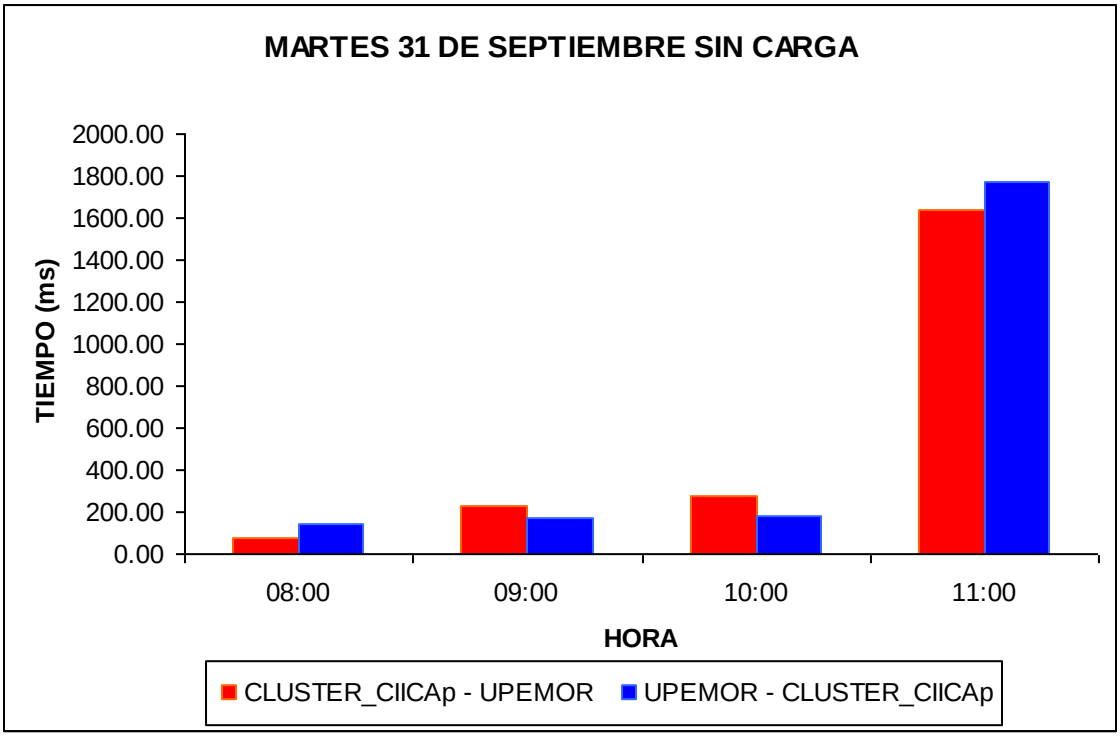


Figura 2. Pruebas de latencia en grid sin carga en hora pico, martes 31 de septiembre.

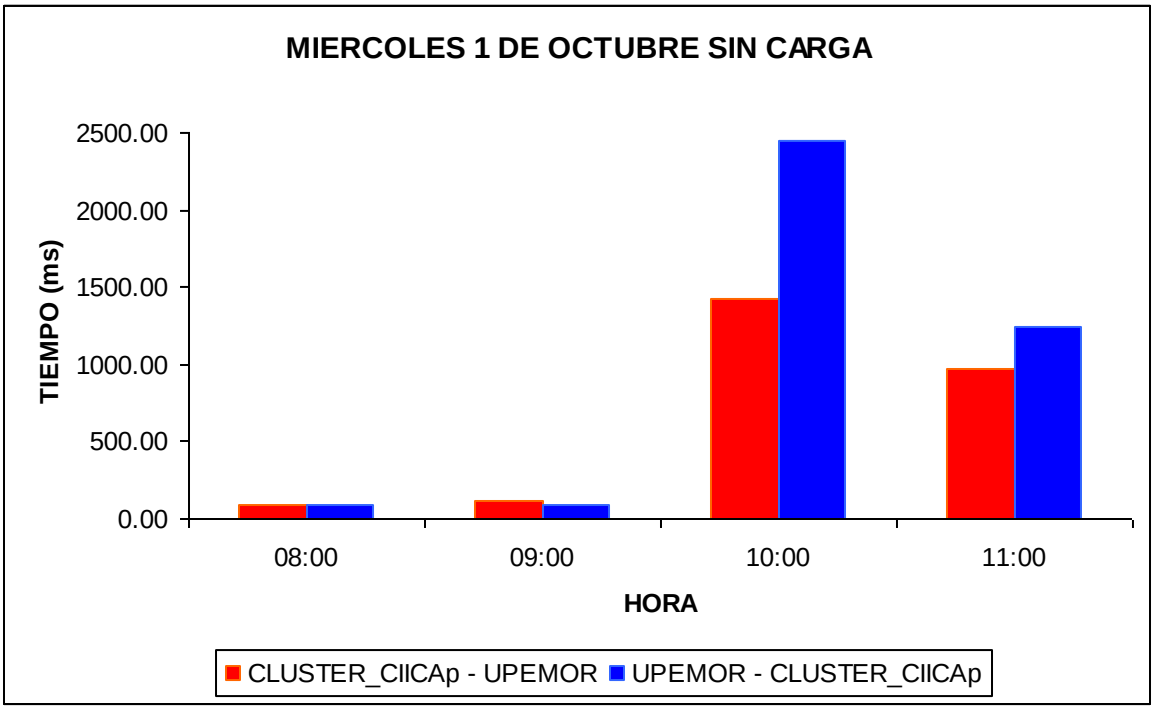


Figura 3. Pruebas de latencia en grid sin carga en hora pico, miércoles 1 de enero.

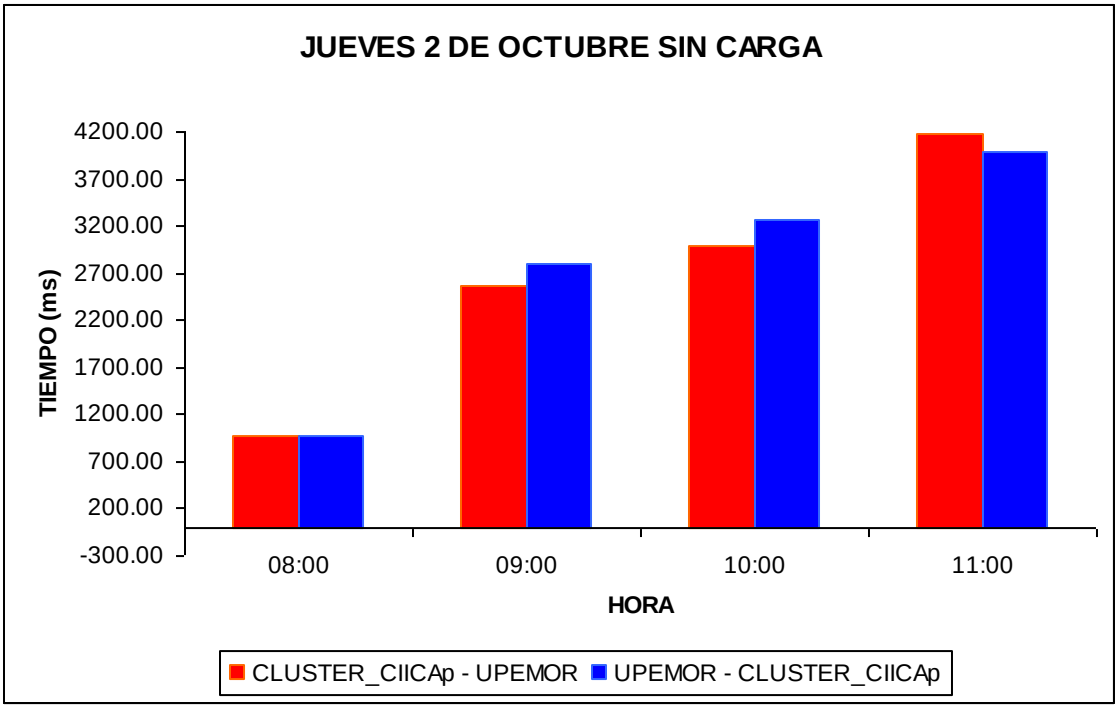


Figura 4. Pruebas de latencia en grid sin carga en hora pico, miércoles 1 de enero.

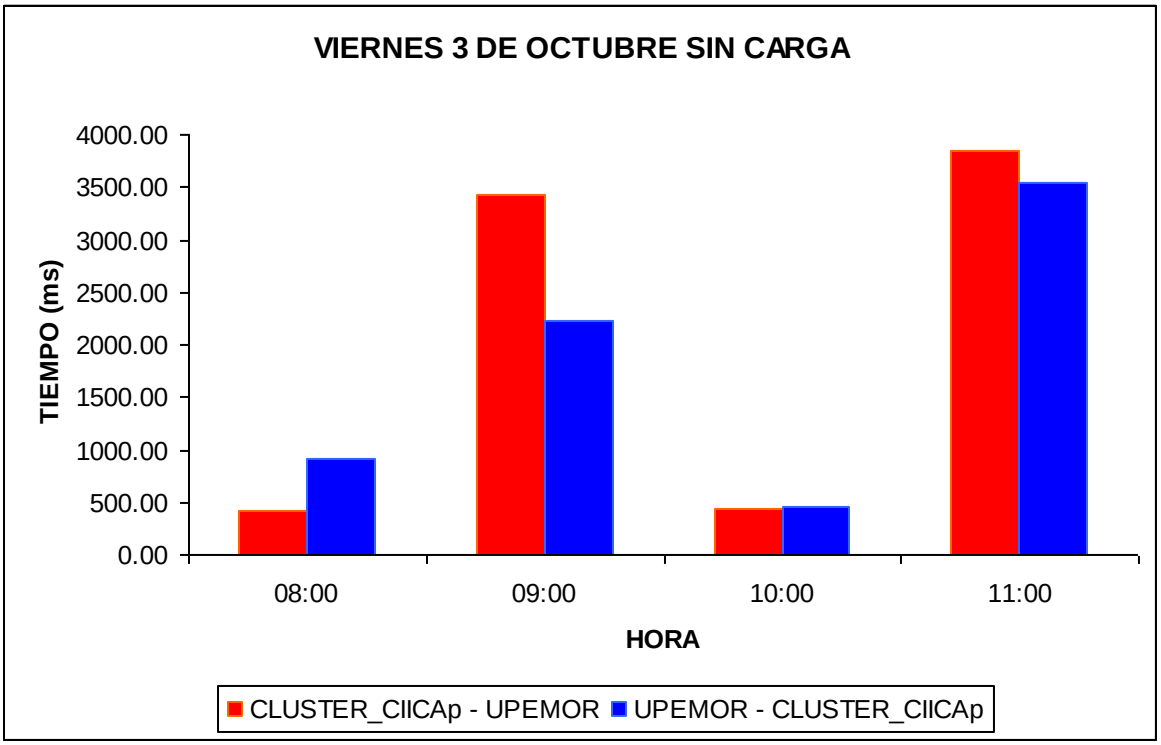


Figura 5. Pruebas de latencia en grid sin carga en hora pico, jueves 2 de enero.

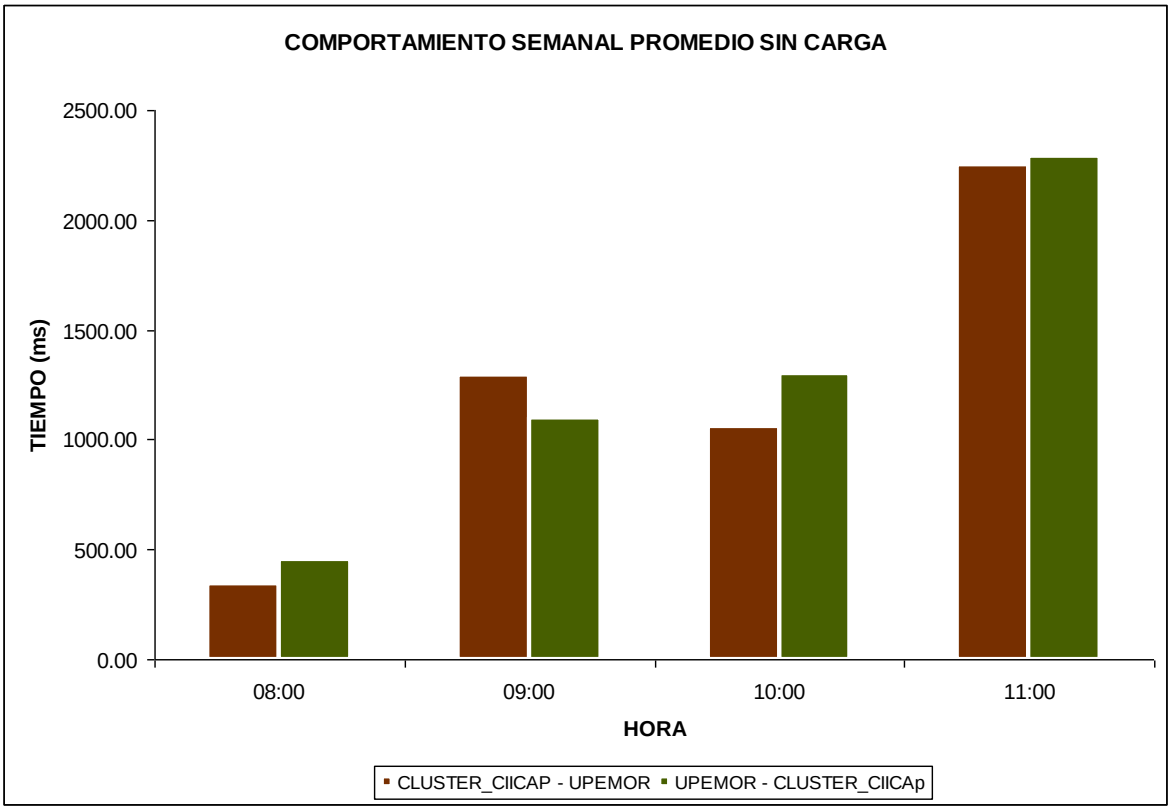


Figura 6. Pruebas de latencia en grid sin carga en hora pico, promedio semanal.

Las pruebas de latencia con carga se realizaron en la grid para los clusters CIICAp y UPEMOR. Las figura 7 a la 11, presentan el comportamiento de la latencia en hora pico de 8 a 11 hrs. En estas pruebas existe carga de procesos comunicándose entre clusters. Se observa que el lunes la latencia se presenta en un intervalo de 1500 a 3500 ms. El martes la conforme avanza el día, la latencia va en aumento de 1000 a cerca de 5000 ms. El miércoles y jueves el comportamiento es parecido, pero existe una latencia mayor el jueves. El viernes existe un comportamiento a la alza conforme avanza el día. La figura 12., presenta la medición de la latencia en promedio de toda la semana en hora pico y con carga en comunicación de procesos entre clusters de la grid. En esta figura, se observa que la latencia aumenta en promedio conforme pasa el día y el intervalo de aumento va desde 3000 a 6000 ms.

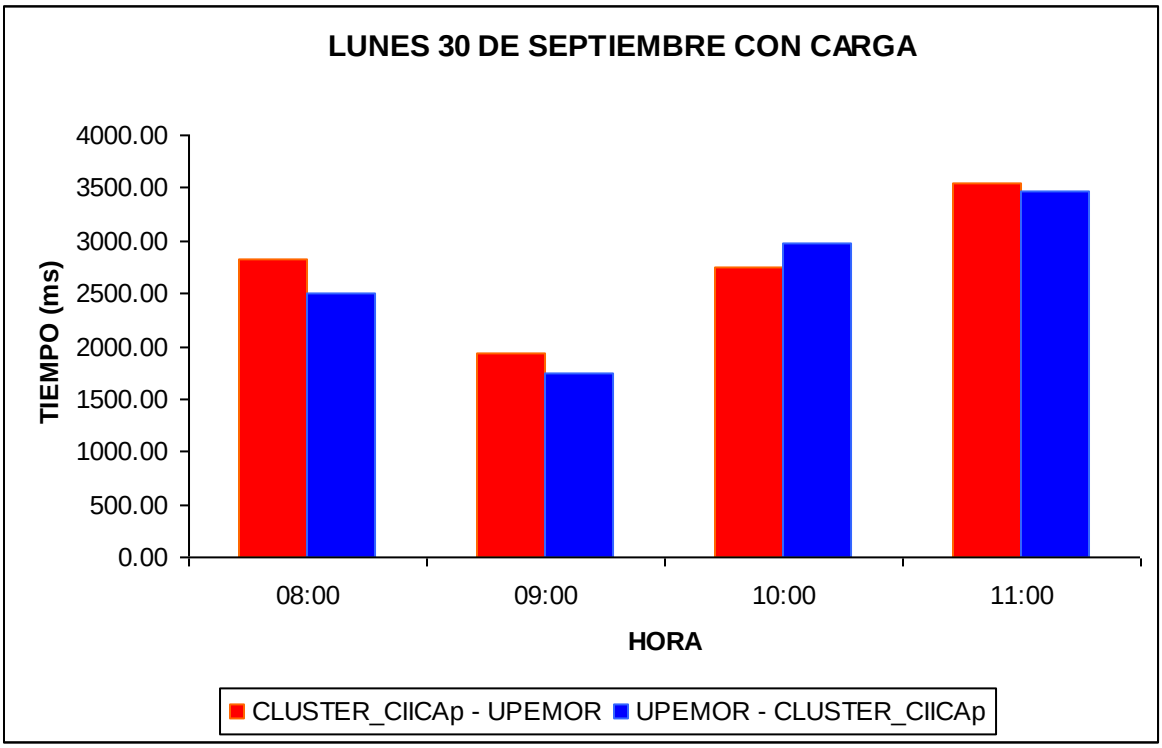


Figura 7. Pruebas de latencia en grid con carga en hora pico, lunes 30 de septiembre.

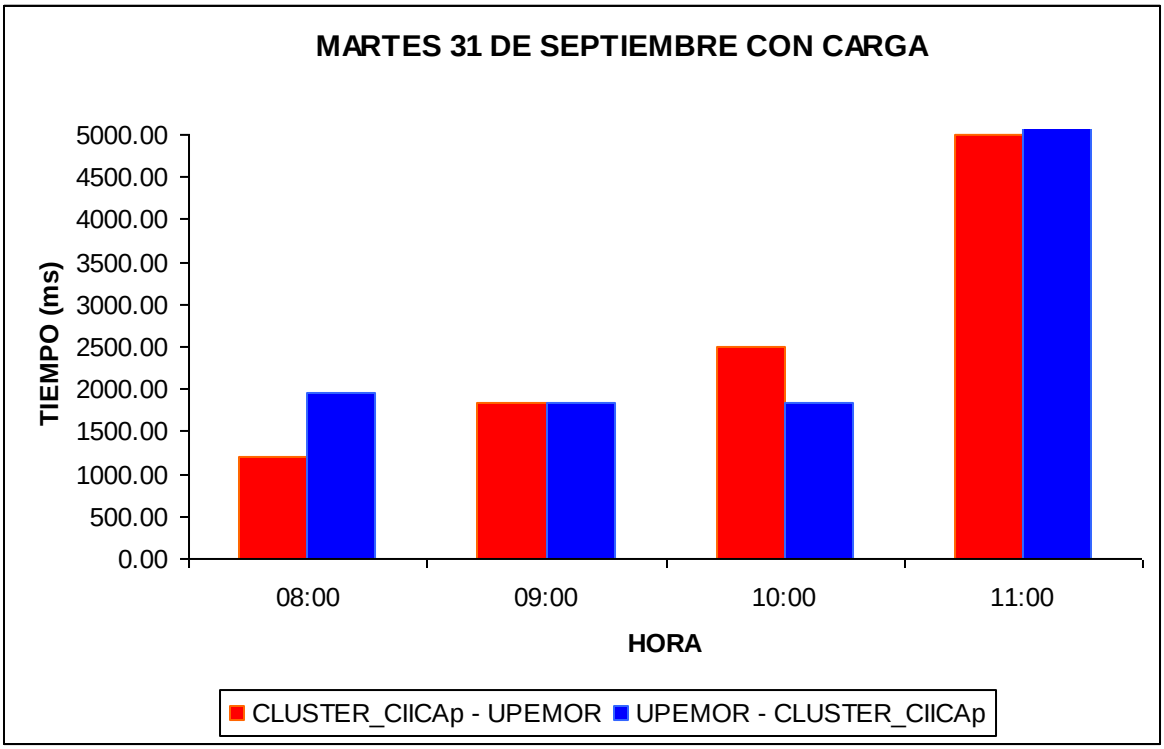


Figura 8. Pruebas de latencia en grid con carga en hora pico, lunes 31 de septiembre.

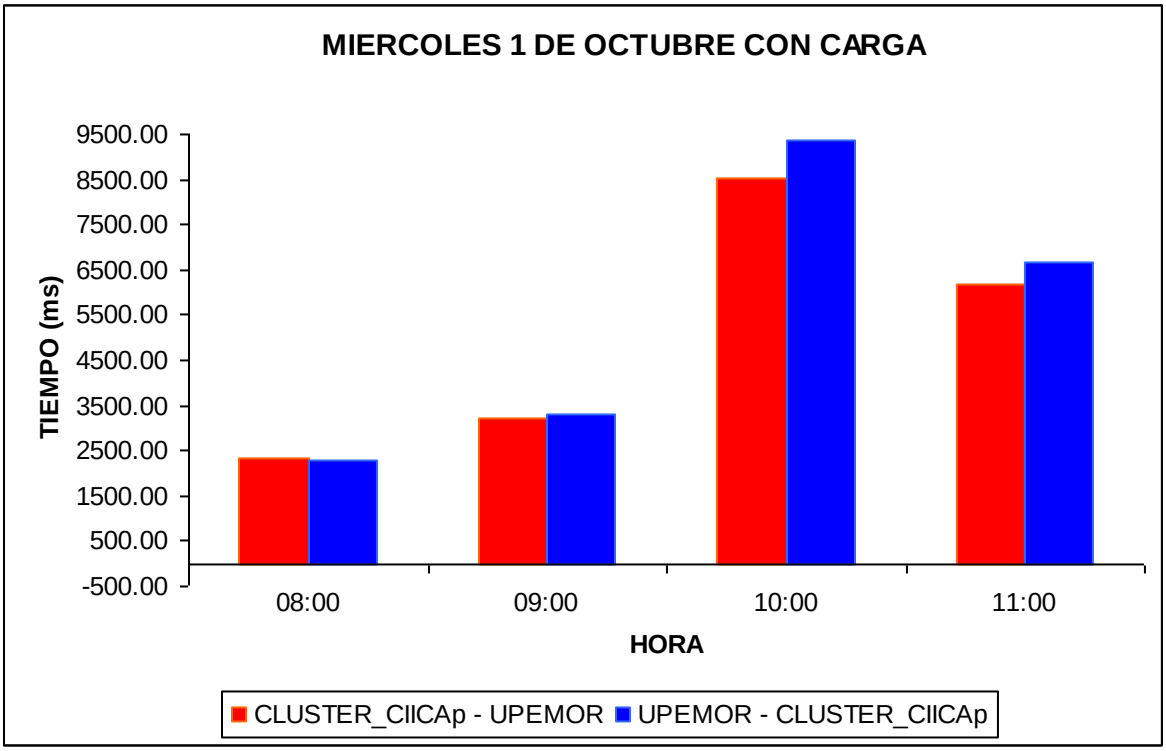


Figura 9. Pruebas de latencia en grid con carga en hora pico, lunes 1 de octubre.

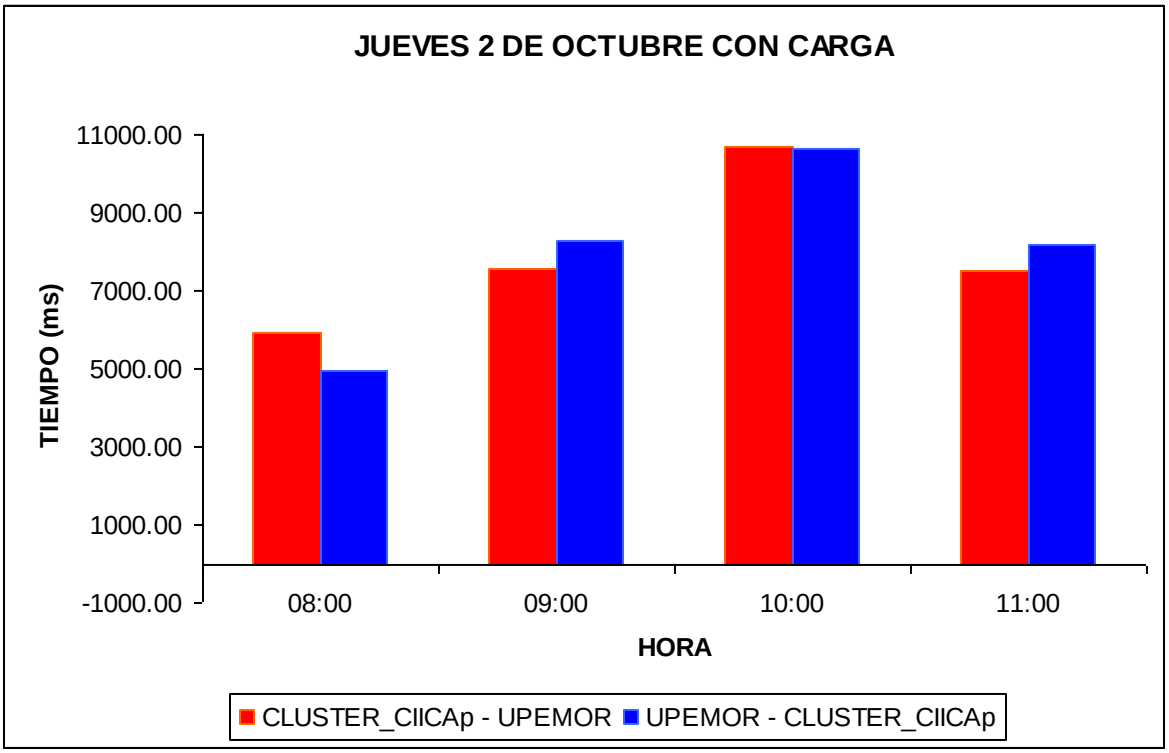


Figura 10. Pruebas de latencia en grid con carga en hora pico, jueves 2 de octubre.

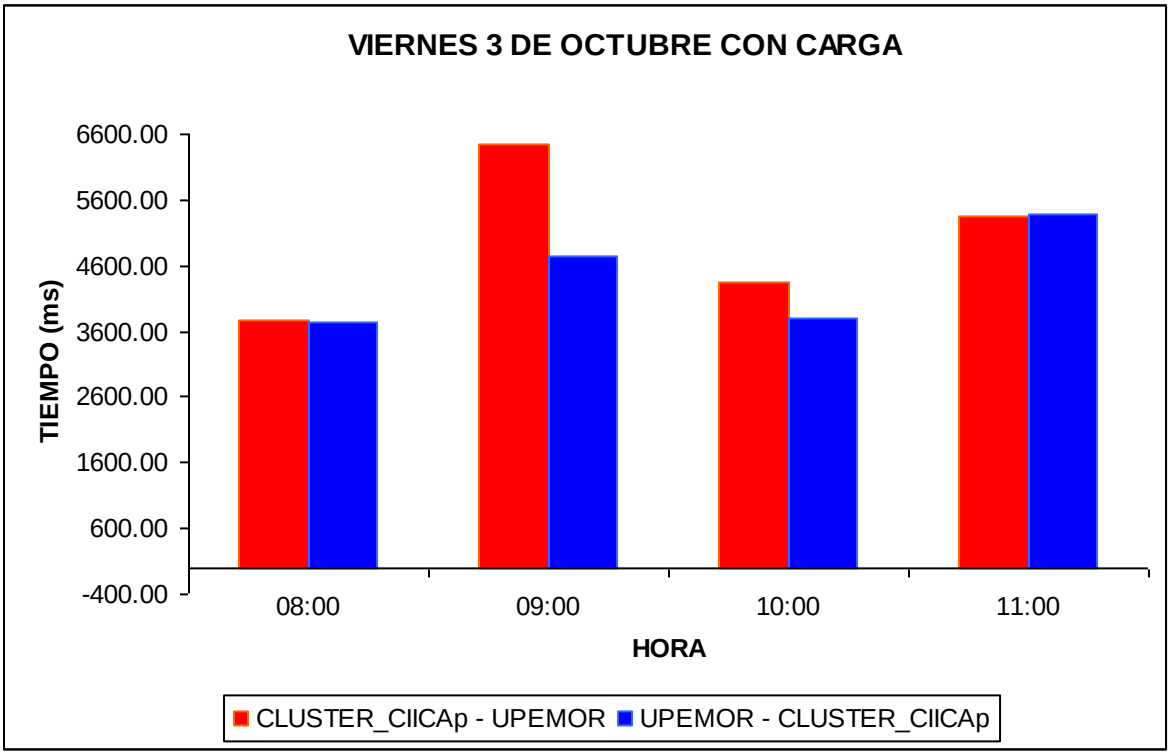


Figura 11. Pruebas de latencia en grid con carga en hora pico, viernes 3 de octubre.

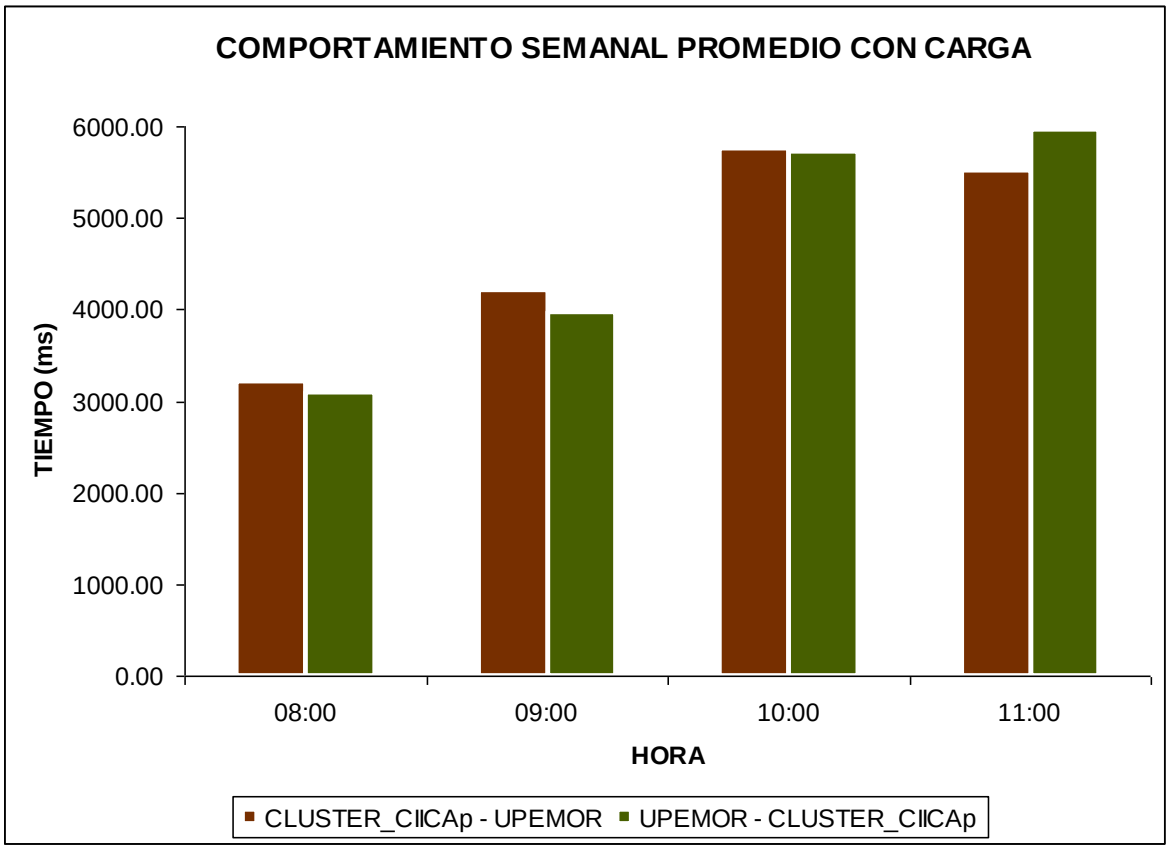


Figura 12. Pruebas de latencia en grid con carga en hora pico, promedio semanal.

De acuerdo a las figuras 6 y 12, se puede ver que el aumento de latencia en promedio cuando existe carga con respecto a cuando no existe, es en un intervalo de 2500 a 3500ms conforme avanza el día. Es decir que si existe carga de procesos el mínimo aumento de latencia es de 2500ms y el máximo aumento de latencia es de 3500ms.

2.2. Estudio del paso de mensajes con datos complejos en la Grid, utilizando memoria dinámica y memoria estática.

2.2.1. Paso de mensajes a nivel Grid con memoria dinámica y estática

Desarrollo de un procedimiento que organice la información en estructuras de datos complejas y maneje el paso de mensajes en una Grid utilizando la librería MPI, aportando una mayor comprensión en el entendimiento de programas de cómputo desarrollados para ambientes *Cluster*.

Tipos de datos existentes en MPI

La correspondencia entre los tipos de datos MPI y los que provee el lenguaje de programación C se muestra en la Tabla 1.

Tipo de dato MPI	Tipo de dato C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Tabla 1 Correspondencia entre tipos de datos soportados por MPI y el lenguaje C

Para todos los tipos de datos que provee el lenguaje C, existen un equivalente tipo de dato MPI. Sin embargo, MPI tiene dos tipos de datos adicionales que no son parte del lenguaje C. Estos son MPI_BYTE y MPI_PACKED.

MPI_BYTE corresponde a un byte (8 bits) y MPI_PACKED corresponde a una colección de elementos de datos que han sido creados en un paquete de datos no contiguos. Note que la longitud del mensaje en MPI_SEND, así como en cualquier otra rutina de MPI, se expresa en términos del número de entradas que se envían y no en el número de bytes.

MPI provee varias formas de empaquetar datos. Esto le permite maximizar el soporte de la información intercambiada en cada mensaje. El paquete de mensajes en *MPI_Send* se compone de una dirección de memoria, un número de elementos y un tipo de datos. Es evidente que este mecanismo puede ser utilizado para enviar múltiples piezas de información en mensajes distintos.

Una estructura de datos es una forma de organizar un conjunto de datos elementales (datos primitivos) con el objetivo de facilitar la manipulación o gestión de dichos datos como un todo, ya sea de manera general o particularmente

De la misma manera, si se necesita enviar más de una estructura es posible enviar dichas estructuras en diferentes llamadas a rutinas de envío y recepción. Con el objetivo de optimizar los recursos y de obtener el mejor resultado de la comunicación entre los nodos de un *Cluster,* las estructuras de datos se vuelven miembros de otras estructuras para así poder hacer una sola llamada a la rutina MPI_Send y transferir los datos complejos que se deseen.

Se desarrolló el algoritmo con estructuras de datos complejas, para resolver el problema de la organización de los datos implementando el paso de mensajes (MPI), con el objetivo de optimizar recursos al enviar los datos en una estructura que engloba otras estructuras, evitando mandar cada dato en rutinas diferentes, se hace un solo envío y una solo llamada a la rutina encargada del envío de los datos.

Una estructura puede estar dentro de otra estructura, a esto se le conoce como anidamiento o estructuras anidadas. Ya que se trabajan con datos en estructuras si definimos un tipo de

dato en una estructura y necesita definir ese dato dentro de otra estructura solamente se llama el dato de la estructura anterior.
Las estructuras de datos se emplean con el objetivo principal de organizar los datos contenidos dentro de la memoria de la PC. En base a los tipos de datos primitivos, se pueden crear nuevos tipos con estructuras compuestas por uno o más de uno de los tipos mencionados [Cairo, 2006].
Teniendo en cuenta que el principal objetivo de las estructuras es organizar los datos del programa, es importante señalar que las estructuras permiten modificar globalmente las variables sin tener que recorrer el código, separando desde el inicio el espacio de memoria que se va a utilizar. En el anidamiento de estructuras se tiene que poner atención en la longitud de los datos y los tipos de datos, ya que se podría separar espacio que nunca se utiliza y terminar con la memoria disponible.
Para declarar un miembro como una estructura, es necesario haber declarado previamente ese tipo de estructura. En particular una estructura no puede ser miembro de ese mismo tipo de estructura, pero si puede contener un puntero o referencia a un objeto del mismo tipo.

```
Struct campo_etiqueta {
  Tipo_struct nodo_1;
  Tipo_struct nodo_2;
  Tipo_struct nodo_3;
        };
```

Código ejemplo de la declaración de una estructura de datos compleja

Los miembros de una estructura son almacenados de manera secuencial, en el mismo orden en el que son declarados.
Con una variable declarada como una estructura, pueden realizarse las siguientes operaciones:

- Obtener su dirección por medio del operador &
- Acceder a uno o a todos sus miembros
- Copiar valores de una estructura de datos a otra

Las estructuras de datos estáticas utilizadas para el paso de mensajes se presentan a continuación:

Tres estructura de datos primitivos con cuatro miembros de tipo entero Figura 13.

- Una estructura de datos compleja que tiene como miembros un arreglo de n*n de cada una de las estructuras de datos anteriores Figura 14.
- Un arreglo de tamaño m, de las estructuras de datos complejas Figura 15.

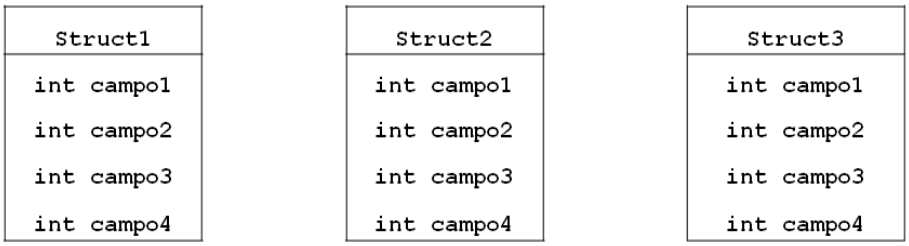


Figura 13 Ejemplo de estructuras de datos simples

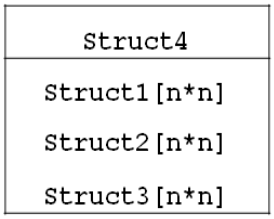


Figura 14 Ejemplo de estructura de datos compleja

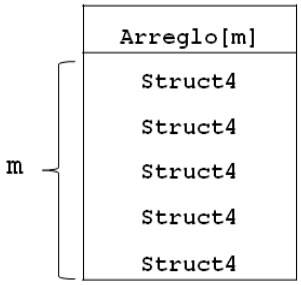


Figura 15 Representación grafica del arreglo de estructuras de datos complejos

Estructuras de datos complejos con memoria dinámica

La creación y el mantenimiento de estructuras de datos dinámicas requiere de la asignación dinámica de memoria, que permite a un programa obtener mas memoria en tiempo de ejecución para el almacenar nuevos nodos. Los operadores new y delete son esenciales para la asignación dinámica de memoria. El operador new toma como argumento el tipo del objeto que se asignará en forma dinámica y devuelve un apuntador a un objeto de ese tipo. Por ejemplo, la instrucción: Nodo *identificador = new Nodo (10).

La declaración de la estructura de datos compleja, y tomando en cuenta las mismas declaraciones de las estructuras de datos simples creados, se presenta a continuación:

```
typedef struct{
        maq **opm;
        maq **MOPM;
        maq **mopm;
        maq **opm_tmp;
        trab **opj;
        trab **MOPJ;
        trab **mopj;
        trab **opj_tmp;
}sched;
```

Declarado el arreglo de las estructuras de datos dinámicas, se procede a asignar la memoria a estas estructuras.

```
for(h = 0; h < TAMPOBLACION; h++){
        poblacion[h].opm = new maq* [nmaq];
        poblacion[h].opj = new trab* [nmaq];
        poblacion[h].MOPM = new maq* [nmaq];
        poblacion[h].MOPJ = new trab* [nmaq];
        poblacion[h].mopm = new maq* [nmaq];
        poblacion[h].mopj = new trab* [nmaq];
        poblacion[h].opm_tmp = new maq* [nmaq];
        poblacion[h].opj_tmp = new trab* [nmaq];
        for(i = 0; i < nmaq; i++){
                for(j = 0; j < nmaq; j++){
                poblacion[h].opm[i] = new maq [nmaq];
                poblacion[h].opj[i] = new trab [nmaq];
                poblacion[h].MOPM[i] = new maq [nmaq];
                poblacion[h].MOPJ[i] = new trab [nmaq];
                poblacion[h].mopm[i] = new maq [nmaq];
                poblacion[h].mopj[i] = new trab [nmaq];
                poblacion[h].opm_tmp[i] = new maq [nmaq];
                poblacion[h].opj_tmp[i] = new trab [nmaq];
                }
        }
}
```

Para la comunicación entre nodos se utiliza el modelo de comunicación bloqueante, no bloqueante y una barrera para evitar que el procesador continue hasta que todos los procesos en el comunicador (MPI_COMM_WORLD) hallan llamado la rutina MPI_Barrier.

Ejecucion de pruebas

- Ejecución de la prueba “Programación paralela” . Los datos enviados a los nodos se reciben exitosamente y son capaces de regresar la información al nodo principal.
- Ejecución de la prueba “Comunicación bloqueante, memoria estática”. Los datos se enviaron y se recibieron satisfactoriamente, con la asignación de la memoria para las estructuras de manera estática.
- Ejecución de la prueba “Comunicación no bloqueante, memoria estática”. Los datos no se enviaron y se recibieron, las barreras fueron ignoradas, por lo tanto no hubo consistencia en los datos.
- Ejecución de la prueba “Comunicación bloqueante, memoria dinámica”. Los datos se enviaron y se recibieron satisfactoriamente, con la asignación de la memoria para las estructuras de manera dinámica. Los nodos esclavos modificaron la información recibida, pero no pudieron regresar los datos, por lo tanto los datos se pierden.
- Ejecución de la prueba “Comunicación no bloqueante, memoria dinámica”. Los datos se enviaron y se recibieron satisfactoriamente, con la asignación de la memoria para las estructuras de manera dinámica. Los nodos esclavos modificaron la información recibida, pero no pudieron regresar los datos, por lo tanto los datos se pierden.

2.2.3. Paso de mensajes a nivel Grid con memoria dinámica y estática utilizando la herramienta de AutoMap

Implementar una “Interfaz de Paso de Mensajes” (MPI) junto con la herramienta AutoMap para estructuras de datos complejas involucrando memoria dinámica, con el propósito de aplicarlas en el desarrollo de algoritmos que optimicen recursos, los cuales nos permitan evaluar la eficiencia de la transferencia de datos en sistemas distribuidos.

Introducción a AutoMap

AutoMap está diseñado para simplificar las tareas de MPI a la hora de crear estructuras de datos complejas.
La Herramienta AutoMap es un compilador diseñado para leer los datos de un código de C, reconociendo algunas directivas especiales y generando un conjunto de archivos que serán interpretados por MPI.
AutoMap es un compilador que traduce automáticamente las estructuras de datos de C en estructuras de datos de MPI. Su gramática reconoce las banderas que se encuentran en el código donde se encuentran las estructuras de datos de C. A continuación, lee estas estructuras y crea los tipos de datos necesarios por MPI.
La construcción de un programa de AutoMap simplemente se basa en la gramática. Para leer esta gramática y aplicar sus normas es necesario que el compilador lea como entrada un archivo con el mismo nombre que tiene el archivo que contiene el código MPI además de que deberá tener una extensión **.h** en el cual deberán de estar las estructuras de datos de

C para que después AutoMap sea capaz de producir sus archivos de salida los cuales contendrán las instrucciones necesarias que utilizara MPI para lograr el paso de mensajes.
Si el proceso de compilación del archivo **.h** fue satisfactorio entonces los archivos de salida generados por AutoMap serán los siguientes:
Mpitypes.h
Mpitypes.inc
Autolink.h
Al_routines.h
Al_routines.inc
Comando para compilar un código AutoMap: AutoMap **archivo.h**

Estructuras de Datos

En la práctica, la mayor parte de la información útil no aparece aislada en forma de daros simples, sino que lo hace de forma organizada y estructurada. Los Diccionarios, guías, enciclopedias, etc., son colecciones de datos que serían inútiles si no estuvieran organizadas de acuerdo con determinadas reglas. Además, tener estructurada la información supone ventajas adicionales, al facilitar el acceso y el manejo de los datos. Por ello parece razonable desarrollar la idea de la agrupación de datos, que tengan un cierto tipo de estructura y organización interna.
Un dato de tipo simple, no está compuesto de otras estructuras, que no sean los bits, por lo tanto su representación sobre el ordenador es directa, sin embargo existen unas operaciones propias de cada tipo, que en cierta manera los caracterizan.
Una estructura de datos es, a grandes rasgos, una colección de datos (normalmente de tipo simple) que se caracterizan por su organización y las operaciones que se definen en ellos. Por tanto, una estructura de datos vendrá caracterizada tanto por unas ciertas relaciones entre los datos que la constituyen, como por las operaciones posibles en ella. Esto supone que podamos expresar formalmente, mediante un conjunto de reglas, las relaciones y operaciones posibles (tales como insertar nuevos elementos o como eliminar los ya existentes).

2.2.3.1. Paso de Mensajes de Estructuras de Datos Complejas Involucrando Memoria Estática a Través de MPI Utilizando AutoMap

Se analizó y se desarrolló un programa que permitiera realizar el paso de mensajes de estructuras de datos complejas en la cual se involucra memoria estática. Las pruebas que se realizaron funcionaron de manera exitosa al realizar paso de mensajes en la grid tarántula entre nodos de varios clusters.
Para poder lograr el paso de mensajes de estructuras de datos complejas involucrando memoria estática es necesario tener dos archivos que son los fundamentales en el paso de mensajes, uno es el que guardara el código de MPI y otro administrara las estructuras de datos que se pretenden pasar del proceso maestro a los procesos maestros los cuales se muestran a continuación:

- Contenedor de las estructuras (.h)
- Contenedor de código MPI (.c)

Archivo de estructuras de datos (.h)

En este archivo se debe de colocar las estructuras de datos que se van a utilizar en el paso de mensajes siguiendo la sintaxis de AutoMap. tal y como se muestra a continuación:
Las estructuras que se necesitan pasar en este ejemplo son las siguientes:

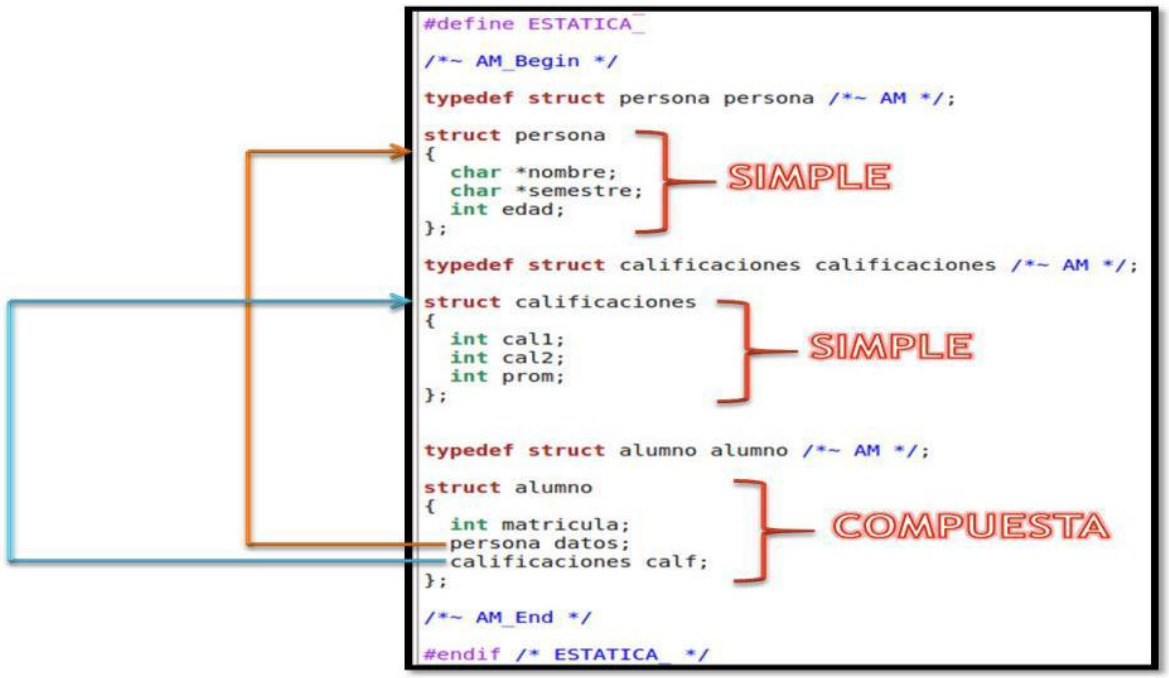


Figura 16. Declaración de Estructuras de Datos en el Archivo **.h**

2.2.3.2. Paso de Mensajes de Estructuras de Datos Complejas Involucrando Memoria Dinámica a Través de MPI Utilizando AutoMap

En esta investigación que se desarrolló no se logró realizar de manera satisfactoria el paso de mensajes de estructuras de datos dinámicas debido a que solamente se logró el envío de información a los procesos esclavos [n-1] provenientes del proceso maestro [0] además de que los procesos esclavos lograron modificar la información que se les fue entregada sin embargo el problema estuvo en el regreso de la información hacia el proceso maestro[0].
El envió y la recepción de la información se realizó de manera bloqueante y no bloqueante para verificar las posibles causas por las cuales no se logró el paso de la información utilizando todas los posibles combinaciones pero sin embargo se llegó a la conclusión de que ninguno de estos métodos lograba de manera satisfactoria el paso de mensajes involucrando memoria dinámica.
Estas son las estructuras de datos complejas que fueron utilizadas las cuales son punteros para poder trabajar con memoria dinámica:

```
#ifndef DINAMICA_
#define DINAMICA_
/*~ AM_Begin */

typedef struct persona persona /*~ AM */;
struct persona
{
  char *nombre;
  char *semestre;
  int edad;
};

typedef struct calificaciones calificaciones /*~ AM */;
struct calificaciones
{
  int cal1;
  int cal2;
  int prom;
};

typedef struct alumno alumno /*~ AM */;
struct alumno
{
  persona **datos;
  calificaciones **calf;
};
/*~ AM_End */
#endif /* DINAMICA_ */
```

Figura 17. Declaración de Estructuras de Datos en el Archivo **.h**

Utilizando la forma bloqueante, cuando se envía información del maestro a los nodos esclavos ésta no se realiza de manera satisfactoria por lo cual el regreso de la información de los nodos esclavos al maestro no se realiza debido a que existe perdida de información.

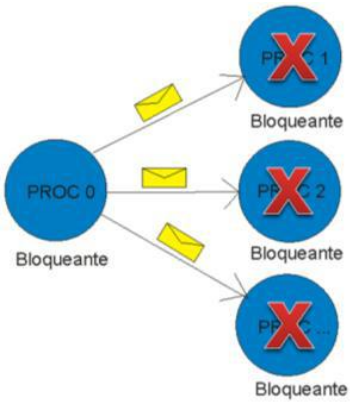


Figura 18. Error al Recibir de Forma Bloqueante

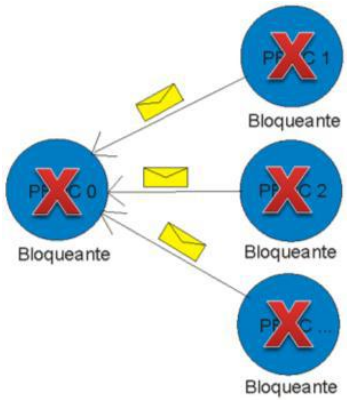


Figura 19. Error al Regresar y Recibir de Forma Bloqueante

Utilizando la forma no bloqueante, enviar información del maestro a los nodos esclavos es satisfactorio, una vez recibidos los datos, estos se procesan y son regresados al proceso maestro [0], una vez regresados a la hora de imprimir la información en el proceso maestro resulto que esta continua sin modificar. Por lo tanto este método resulto no ser útil debido a que la información que fue modificada por los nodos esclavos realmente no llego al maestro.

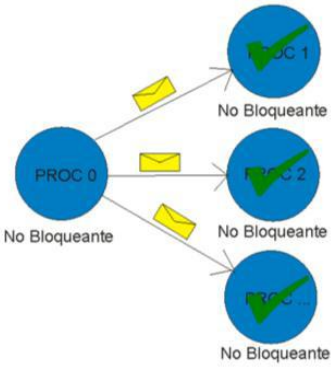


Figura 20. Éxito al Recibir de Manera No Bloqueante

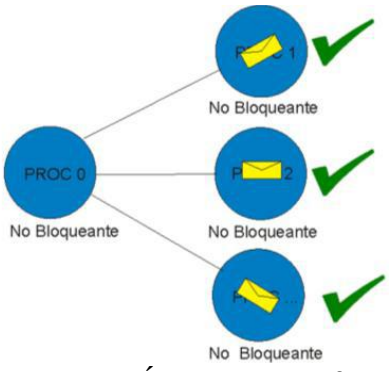


Figura 21. Éxito al Modificar Datos de Manera No Bloqueante

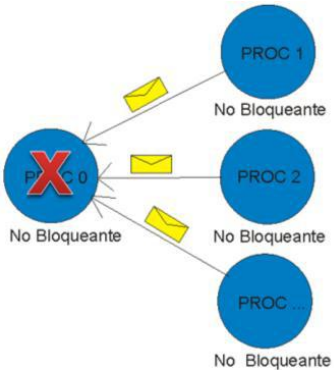


Figura 22. Error al Regresar y Recibir Información

Después de haber utilizado los dos métodos anteriores para el paso de mensajes y haber verificado que no se logró el paso correcto de la información se optó por realizar un método híbrido que permitiera el paso de mensajes. El resultado fue que cuando queríamos enviar información del maestro a los nodos esclavos se realizaba de manera satisfactoria, una vez que son recibidos los datos deberán de ser procesados, esto se realizó de manera satisfactoria, una vez que fueron procesados deben de ser regresados al proceso maestro [0] y es aquí a donde surgió el problema debido a que no se logró hacer el regreso de la información y por lo tanto el método tampoco fue exitoso.

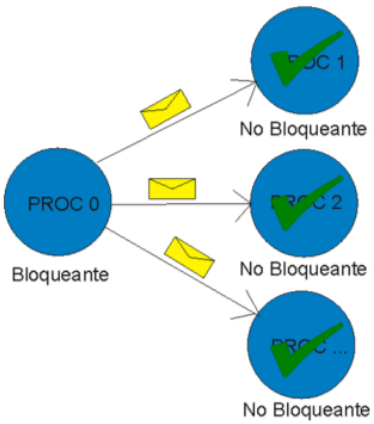


Figura 23. Éxito al enviar y recibir de forma híbrida

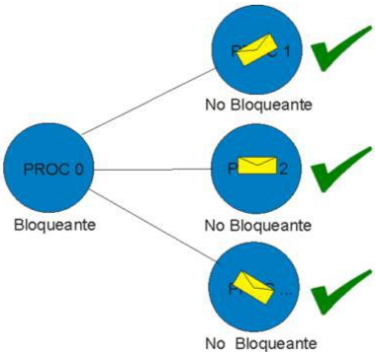


Figura 24. Éxito al modificar datos de forma híbrida

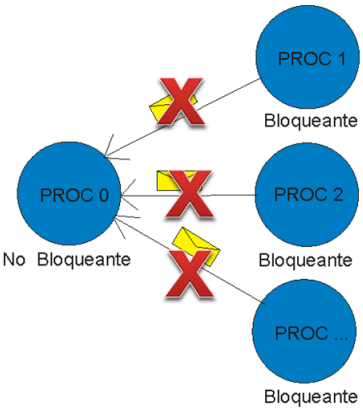


Figura 25. Error al regresar y recibir de forma híbrida

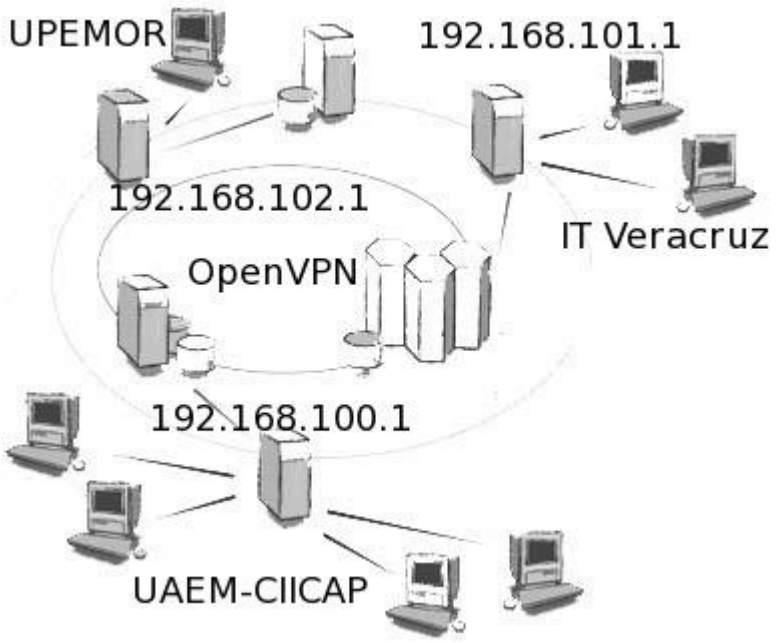
2.3. Conexión infraestructura UAEM-ITVer-UPEMOR

Conectar a través de Open VPN los clusters de las instituciones UAEM, IT Veracruz y UPEMOR, donde a través del uso de certificados de SSH, se tiene acceso remoto y se procede a compartir recursos de procesamiento distribuido, para ejecutar jobs de un algoritmo al mismo tiempo en la Grid Tarántula.
Actualmente la infraestructura de la grid Tarántula en la que se realizaron las pruebas es la siguiente:

Características técnicas Cluster Rocks UAEM.
1 equipo con procesador Pentium 4 a 2793 Mhz, 512 MB Memoria RAM, 80 GB en disco duro y 2 tarjetas de red10/100 Mbps, y 18 nodos esclavos con procesador Intel Celeron Dual Core a 2.0 Ghz, 2 GB de memoría RAM, 160 GB en disco duro y 1 tarjeta de red 10/100 Mbps.
Carcaterísticas técnicas Cluster Rocks UPEMOR.
5 equipos Sunfire x2270w con procesador Intel Core 2 Duo a 2.6 Ghz, con 2 GB en memoria RAM, disco duro de 160 GB.
Características técnicas Cluster Nopal IT Veracruz.
1 equipo con procesador pentium 4, 2394 Mhz, 512 GB Memoria RAM, 60 GB disco duro y 2 tarjetas de red 10/100 Mbps.
2 equipo con procesador Pentium 4 Dual Core, 3200 Mhz, 1 GB Memoria RAM, 80 GB disco duro y 1 tarjeta de red 10/100 Mbps.

En este año se mejorará la infraestructura de la grid tarántula, adicionando dos clusters ubicados uno en el CIICAp y el otro en la UPEMOR. Estos dos nuevos clusters pasarán a formar parte de la Grid Tarántula dentro del proyecto para crear la grid Morelos. De forma general se aumentara la grid con 120 nuevos núcleos de procesamiento y 4 tarjetas de video especiales para la programación con GPU's. La comunicación entre nodos de cluster será con infiniband y se independizará la grid del tráfico de datos institucional para un funcionamiento mas eficiente en cuanto a latencias.

Diagrama infraestructura Grid Tarántula en conexión clusters instituciones por openVPN.



2.4. Pruebas del algoritmo Genético para la optimización de recursos utilizando todos los nodos de la Grid

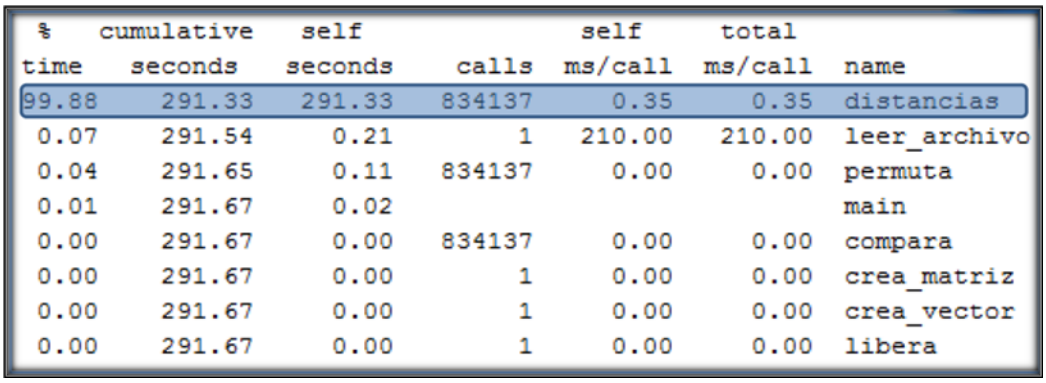
El trabajo realizado consistió en la paralelización de un algoritmo secuencial que emplea una estructura de vecindad híbrida para resolver el Problema del Agente Viajero [CRUZ-CHÁVEZ2010]. En la literatura [CRUZ-CHÁVEZ2010, LIU2007] se proponen distintas estructuras para generar caminos y realizar búsquedas locales de manera iterativa. En [CRUZ-CHÁVEZ20010] se muestran gráficas con los resultados obtenidos aplicando un algoritmo secuencial con una estructura de vecindad híbrida, los cuales resultan ser favorables. A partir de este algoritmo se realizaron ajustes y fue la base para realizar la paralelización del mismo para su posterior aplicación en un ambiente grid.

2.4.1. Evaluación y Modificación del Algoritmo Secuencial

Inicialmente se realizaron algunas modificaciones al algoritmo secuencial para que fuera más fácil de hacer los ajustes de la paralelización.Posteriormente se realizó un *profile* de la ejecución para determinar las funciones que pudieran paralelizarse, obteniéndose el resultado mostrado en la figura 26.

A partir de estos resultados, se determinó que la función a paralelizar sería distancias debido a que es la que mayor tiempo demora en ejecutarse de manera individual (después de la función leer_archivo).

La ejecución del algoritmo secuencial original tenía la restricción de un límite de 5 minutos, realizando alrededor de 830,000 iteraciones en ese tiempo y siendo el mejor costo del camino de 194,498 para un recorrido de 4000 ciudades.



% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
99.88	291.33	291.33	834137	0.35	0.35	distancias
0.07	291.54	0.21	1	210.00	210.00	leer_archivo
0.04	291.65	0.11	834137	0.00	0.00	permuta
0.01	291.67	0.02				main
0.00	291.67	0.00	834137	0.00	0.00	compara
0.00	291.67	0.00	1	0.00	0.00	crea_matriz
0.00	291.67	0.00	1	0.00	0.00	crea_vector
0.00	291.67	0.00	1	0.00	0.00	libera

Figura 26. Profile del algoritmo secuencial.

Se cambió la restricción de ejecución a un total de 830,000 iteraciones para 4000 ciudades y se ejecutó 10 veces el algoritmo secuencial modificado en una laptop con las características mostradas en la Tabla 2.

Tabla 2. Características de laptop.

PROCESADOR	MEMORIA RAM	SISTEMA OPERATIVO
AMD Athlon(tm) 64 X2 Dual-Core Processor TK-55 1.80 GHz	1.5 GB	Ubuntu 10.04 – Lucid Linux

Los resultados obtenidos se muestran en la Tabla 3.

Tabla 3. Resultados del algoritmo secuencial obtenidos en la laptop.

CONCEPTO	VALOR
Menor tiempo ejecución obtenido:	4 min. 46 seg.
Menor distancia obtenida:	196,950

2.4.2. Propuesta de Paralelización

Se propuso una paralelización a dos niveles, en la que en primera instancia, un procesador central genere 4 caminos distintos de manera aleatoria y lo envíe a 4 núcleos diferentes (figura 27), cada uno de los cuales procese al mismo tiempo su propio camino y divida el cálculo de la distancia en 4 hilos (figura 28); una vez k se hayan realizado las 830,000 iteraciones en cada camino, se retorna el resultado al proceso maestro y éste elige el mejor camino generado. La implementación final no emplearía solo 4 caminos, si no la cantidad de caminos de acuerdo al número de núcleos disponibles en el clúster.

Además, a la propuesta se le añadió que al momento de que el proceso maestro obtenga los 4 caminos, realice un cruzamiento y se envíen los caminos generados nuevamente a los nodos, los cuales evalúen de nuevo el camino y retornen al proceso maestro la mejor solución encontrada. Este proceso se deberá realizar un número N de generaciones.

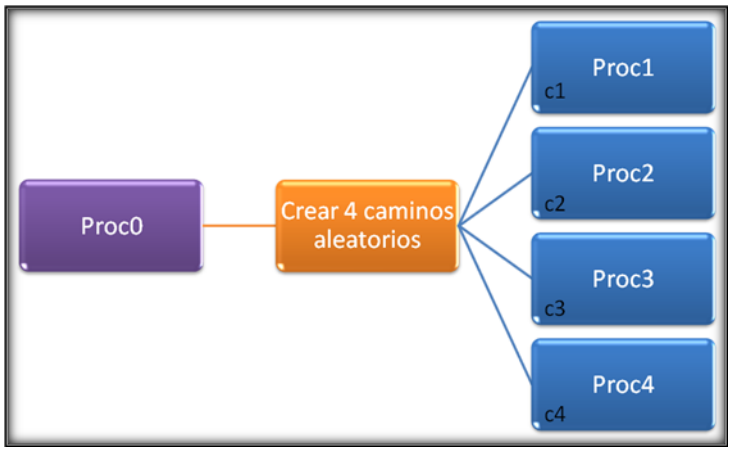


Figura 27 Propuesta de paralelización con MPI.

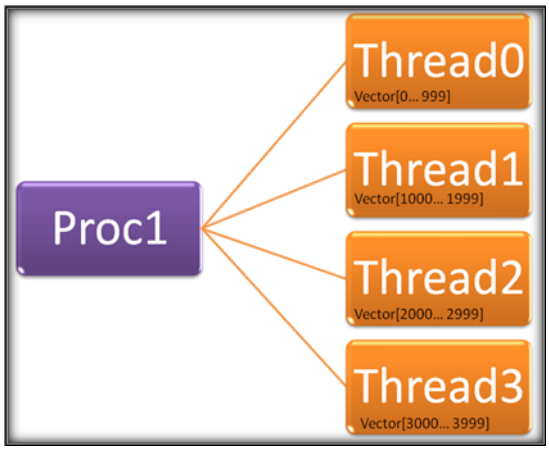


Figura 28 Propuesta de paralelización con OpenMP.

2.4.3. Implementación de Paralelización con OpenMP

Se implementó un cruzamiento normal con un punto de cruce aleatorio que posteriormente realiza un ajuste en el camino para eliminar las ciudades repetidas. El ajuste consiste en buscar aquellas ciudades que se encuentren repetidas y determinar cuáles no han sido visitadas para posteriormente reemplazar las primeras por éstas últimas. Con esto, se obtuvo un algoritmo con aplicación genética.

En la figura 29 se muestra el algoritmo secuencial ya modificado a manera de pseudocódigo. Y para una mejor comprensión del pseducódigo mostrado la figura 29 se muestra el diagrama de flujo del algoritmo secuencial con los ajustes previos a la paralelización (figura 30).

```
Genera una nueva semilla para la función rand()
Limpia la pantalla y guarda el time
Imprime información de los datos e iteraciones
Crea la matriz con las distancias entre ciudades
Crea los vectores empleados para las permutaciones y cruzamientos
Lee el archivo que contiene las distancias entre las ciudades

WHILE generación actual < número de generaciones
      Imprime el número de la generación actual
      DO
            IF generación 0
                  Crea un camino de manera aleatoria
            ELSE
                  Toma uno de los caminos hijos del cruzamiento

            Realiza el cálculo de las distancia del camino
            Imprime los resultados
      WHILE camino actual < total de caminos

      Busca el camino con la menor distancia
      Imprime el resultado

      IF generación 0
            Almacena el camino encontrado con la menor distancia
      ELSE
            Compara el camino encontrado con la menor distancia en la
            generación actual con el de menor distancia encontrado en las
            generaciones anteriores y almacena el mejor

      Cruza los caminos por parejas
      Almacena los caminos hijos para evaluarlos en la siguiente generación

      Incrementa generación
TERMINA WHILE

Imprime la información del camino con la menor distancia encontrado en todas
las generaciones
```

Figura 29 Pseudocódigo del algoritmo secuencial del TSP con modificaciones previas a la paralelización.

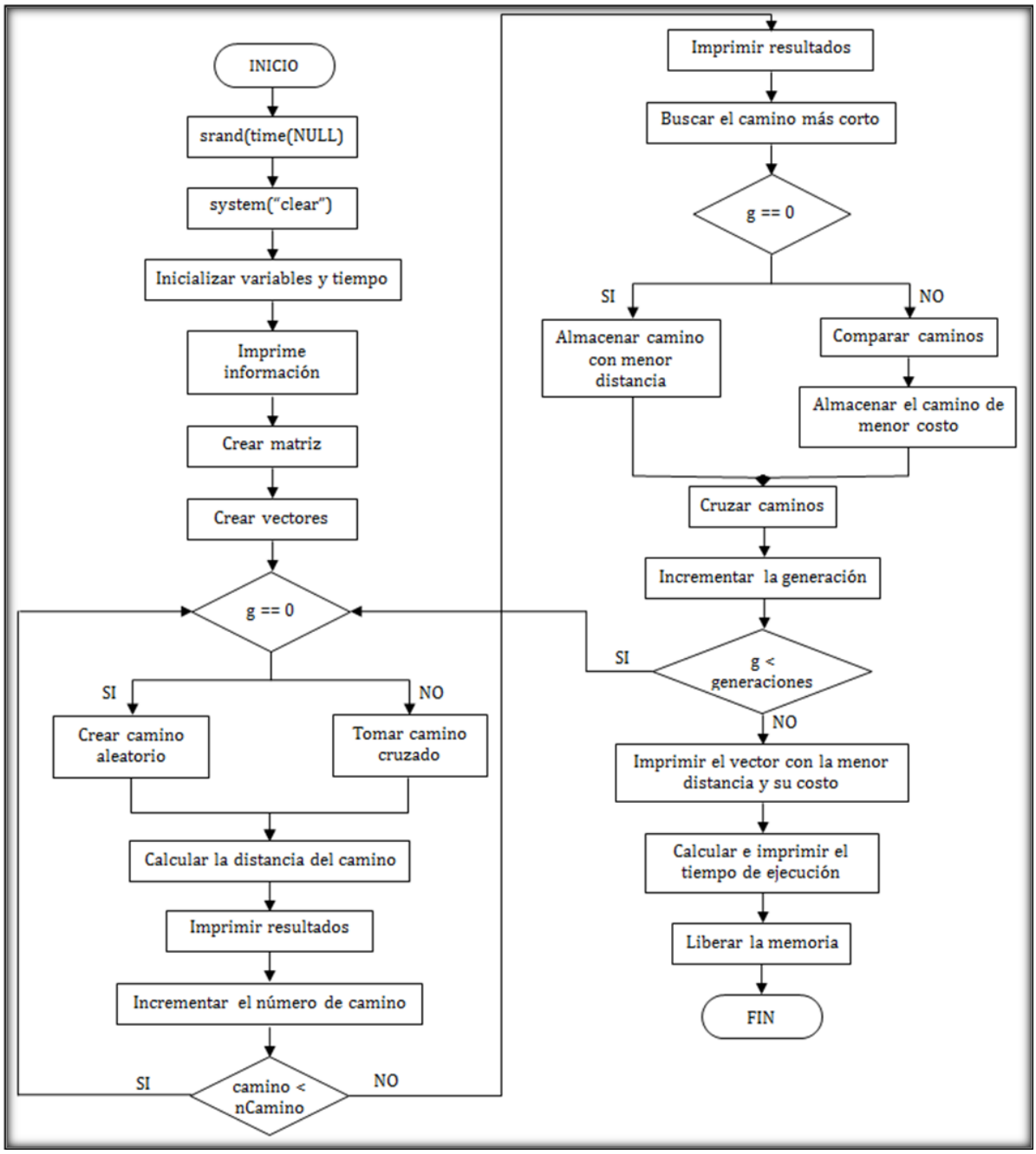


Figura 30 Diagrama de flujo del algoritmo del TSP con ajustes previos a la paralelización.

Además se añadió la función *realizarGeneraciones* la cual se encarga de ejecutar las N generaciones que se indiquen.

La primera implementación de paralelización fue con OpenMP. A partir del profile, se efectuó una primera paralelización de datos (función distancias), dividiendo el trabajo del cálculo de la distancia del camino en más de un hilo, para lo cual se empleó la librería *omp.h* y paralelizar la función distancias. Cabe mencionar que la ejecución se realizó sobre un solo núcleo del procesador y la paralelización se lleva a cabo con el empleo de hilos dentro del mismo núcleo.

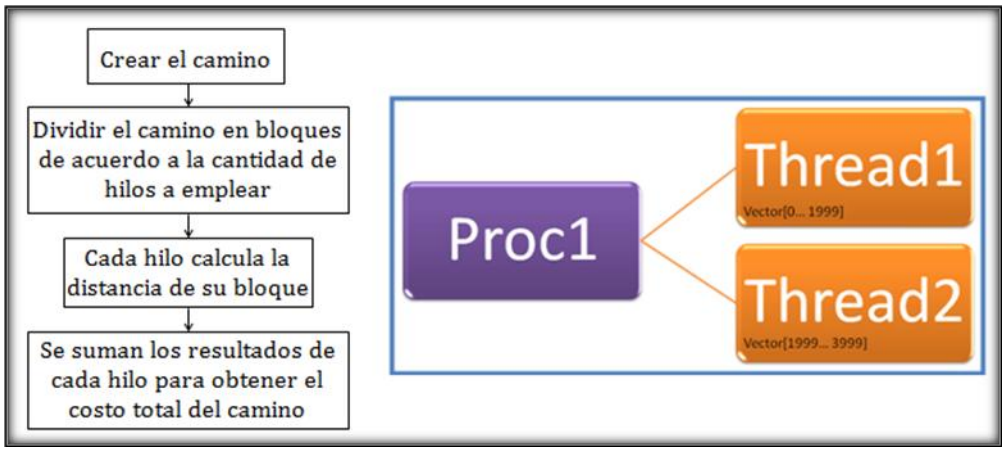


Figura 31 Esquema de paralelización con OpenMP empleando 2 hilos, en donde Proc1 se refiere a un núcleo de proceso, mientras que Threa1 y Thread2 son los hilos en los que se divide la paralelización en dicho núcleo.

Se realizaron nuevamente 10 pruebas en el mismo equipo y el mismo número de iteraciones pero empleando 2 y 4 hilos, obteniéndose los resultados que en la Tabla 4 se muestran.

Tabla 4. Resultados de la paralelización con OpenMP (2 y 4 hilos) obtenidos en la laptop.

CONCEPTO	2 HILOS	4 HILOS
Menor tiempo ejecución obtenido:	1 min. 38 seg.	2 min. 33 seg.
Menor distancia obtenida:	196,814	196,545

En comparación con el secuencial, se redujo aproximadamente a la mitad el tiempo de ejecución. Sin embargo, no fue favorable aumentar el número de hilos ya que también aumentó el tiempo de ejecución. La implementación paralelizada con OpenMP se decidió dejarla con 2 hilos.

Posteriormente se realizaron pruebas haciendo uso del clúster Nopal, ubicado en la ciudad de Veracruz, en el ITV, en donde se lanzaron pruebas del código secuencial con las mismas características de ejecución empleadas anteriormente, pero ahora se evaluaron 4 caminos en la misma corrida. Las características del nodo del clúster Nopal en el que se ejecutaron las pruebas se muestran en la Tabla 5.

Tabla 5. Características de un nodo del clúster Nopal.

PROCESADOR	MEMORIA RAM	SISTEMA OPERATIVO
Intel Pentium 4.3 GHz QuadCore, de 32 bits	1.0 GB	Rocks 5.3 (Sistema Operativo base: CentOs)

Los resultados obtenidos para la ejecución secuencial se muestran en la Tabla 6.

Tabla 6. Resultados del algoritmo secuencial ejecutado en un nodo del clúster Nopal, evaluando 4 caminos.

CONCEPTO	VALOR
Menor tiempo ejecución obtenido:	7 min. 20 seg.
Menor distancia obtenida:	189,675

Haciendo uso del clúster se obtuvo una mejora en ambos aspectos, ya que el tiempo de ejecución fue de más de menos de la mitad del estimado y la menor distancia obtenida fue considerablemente menor que la anterior.

Las pruebas con la implementación de OpenMP con 2 hilos arrojaron resultados similares a las anteriores. En estas pruebas también se obtuvieron mejores resultados de los que se esperaban, pero la proporción entre los tiempos de ejecución del secuencial y del paralelo con OpenMP (2 hilos) no se mantuvo con respecto a los resultados obtenidos en la laptop.

Una vez realizadas las pruebas con OpenMP, se procedió a la paralelización con MPI para enviar distintos caminos a diferentes núcleos y de esta manera aprovechar realmente el poder de cómputo del clúster.

2.4.4. Implementación de Paralelización con MPI

La idea de la implementación de MPI (figura 32 y 33) es generar tantos caminos como nodos (N) hayan en el clúster (en su momento, en la grid) para que el proceso maestro genere N-1 caminos y envíe uno diferente a cada nodo. Posteriormente, todos los nodos trabajarán al mismo tiempo de manera independiente con su propio camino haciendo la paralelización con OpenMP para el cálculo de la distancia total del camino. Mientras tanto, el proceso maestro estará esperando recibir de vuelta los caminos ya procesados por los nodos y, una vez que haya recibido todos esos caminos, procederá a obtener el camino con el menor costo. Seguidamente realizará el cruce de los caminos para obtener una nueva población que enviará de nuevo a los nodos para su procesamiento. El proceso maestro realizará estas tareas durante un número dado de generaciones. Una vez terminado su ciclo, procederá a mostrar el camino con la menor distancia que se haya obtenido de todas las generaciones.

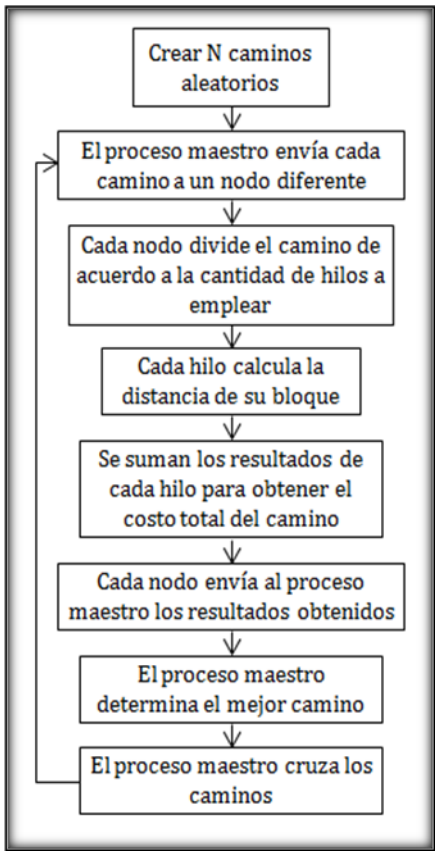


Figura 32. Esquema de paralelización con MPI y OpenMP para varias generaciones.

Debido a que para ese momento, el clúster del CIICAp ya estaba funcionando, las pruebas con MPI se realizaron en el mismo, el cual cuenta con las características mostradas en la Tabla 7.

Tabla 7. Características de un nodo del clúster CIICAp.

PROCESADOR	MEMORIA RAM	SISTEMA OPERATIVO
Intel Celeron Dual Core 2.0 GHz, de 32 bits	1.0 GB	Rocks 5.3 (Sistema Operativo base: CentOs)

En este clúster, la implementación secuencial con 4 caminos demoraba alrededor de 20 minutos en ejecución, mientras que la implementación únicamente con OpenMP era de alrededor de 10 minutos.

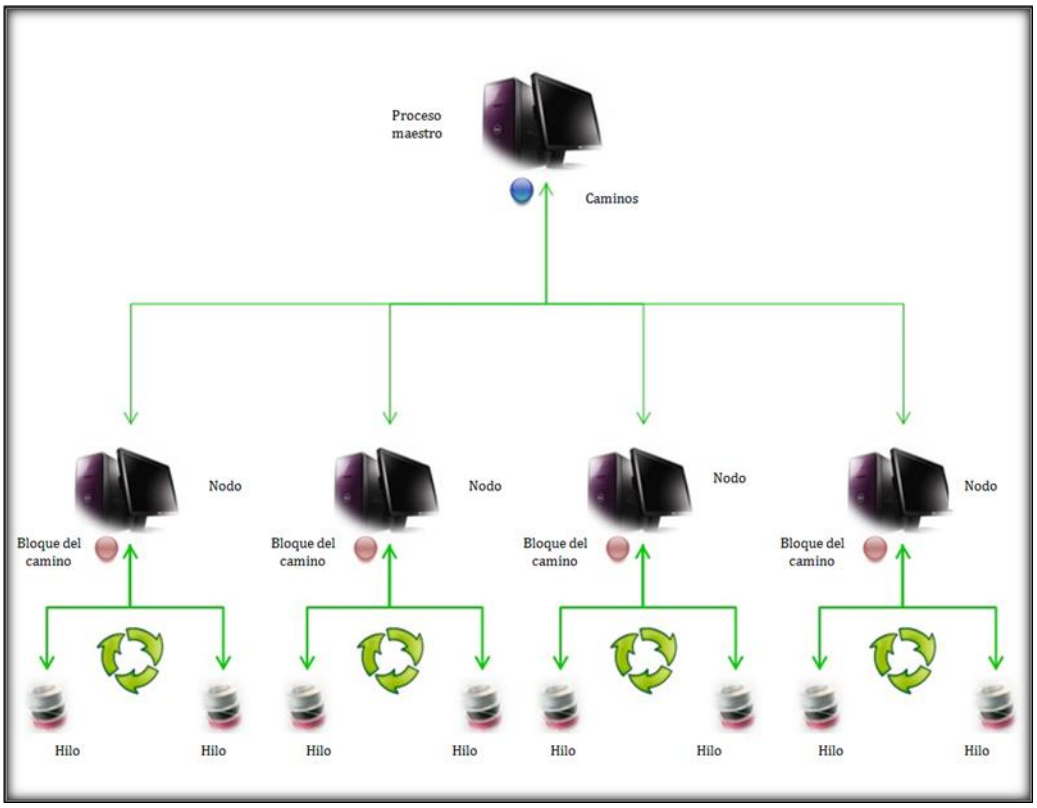


Figura 33 Esquema gráfico de la paralelización con MPI y OpenMP.

Se procedió a implementar la paralelización con MPI para reducir aún más el tiempo de ejecución. Sin embargo, el tiempo de ejecución seguía siendo similar al de la implementación secuencial, esto debido a que las funciones de envío y recepción de mensajes por parte del *proc 0* estaban en el mismo ciclo, lo cual hace que la ejecución se comporte como si fuera de manera secuencial, por lo que se realizaron algunos cambios en el código, reduciendo de esta manera el tiempo de ejecución (7 min. aproximadamente) de acuerdo al que se tenía contemplado en principio, es decir, el tiempo de ejecución del secuencial para un camino (5 min. aproximadamente).

Una de las modificaciones fue la separación de la recepción y envío de los caminos. En un principio, el envío y recepción de mensajes por parte del proceso 0 se implementó dentro del mismo ciclo; es decir:

a) Crear un camino.

b) Enviar el camino a un proceso de trabajo.

c) Recibir el camino y la distancia después de ser iterado por otro proceso.

d) Volver a (1) hasta haber creado el número establecido de caminos.

Debido a que la función *MPI_Recv()* bloquea el proceso hasta que se reciba el mensaje, el tiempo de ejecución del código continuaba siendo similar al de la ejecución secuencial, ya que el *proceso 0* enviaba el camino a otro proceso y se quedaba en espera hasta que ese proceso le enviara el mensaje del camino ya iterado, y posteriormente el

proceso 0 creaba un nuevo camino para enviarlo a otro proceso, quedándose nuevamente en espera hasta recibir el mensaje del proceso al que envió el camino. Siendo la ejecución como en el código secuencial, pero en este caso se estaban empleando también otros equipos para realizar las iteraciones de los caminos generados.

Después se eliminó la paralelización con OpenMP y se separó en dos ciclos el envío y la recepción de los mensajes del *proceso 0*.Al colocar en un ciclo la creación y el envío de los caminos del proceso 0 y luego seguido de otro ciclo con la recepción de los caminos ya iterados, se redujo considerablemente el tiempo de ejecución. Ahora la secuencia para ello es la siguiente:

1. Crear un camino.
2. Enviar el camino a un proceso de trabajo: MPI_Send().
3. Volver a (1) hasta haber creado el número establecido de caminos.
4. Recibir un mensaje: MPI_Recv().
5. Volver a (4) hasta haber ejecutado la función MPI_Recv() el número establecido de caminos.
6. Continuar con el código.

2.4.5. Ejecución del Algoritmo en la Grid UAEM – UPEMOR

El tiempo de ejecución para un camino con el algoritmo secuencial es de aproximadamente 5 minutos. Debido a que se eliminó la paralelización con OpenMP, la ejecución de N caminos en paralelo (MPI) a través de la grid debe ser de poco más de 5 minutos debido al aumento de consumo de tiempo a causa de la latencia de la red.

En el caso de la grid UAEM-UPEMOR, el tiempo de ejecución de una generación fue de alrededor de 7 minutos para realizar las iteraciones de 27 caminos. Este aumento de tiempo se debe al tiempo que se toma en dar inicio al ambiente paralelo, así como al envío de los mensajes entre los nodos. Los resultados obtenidos de las pruebas en la grid se muestran en la Tabla 8.

Tabla 8. Resultados del algoritmo paralelizado con MPI, ejecutado en la grid UAEM-UPEMOR, para la evaluación de 27 caminos en 3 generaciones.

CONCEPTO	VALOR
Menor tiempo ejecución obtenido:	20 min. 9 seg.
Menor distancia obtenida:	186,595

Implementación de Operadores Genéticos

Posteriormente se agregaron los operadores genéticos de *selección* y *mutación* al algoritmo para completar las actividades. En la Fig. 34 se muestra un grafo con las actividades que

realiza cada proceso y la secuencia de las mismas. De esta manera es como quedó finalmente el algoritmo genético paralelo para la resolución del PAV en un ambiente grid.

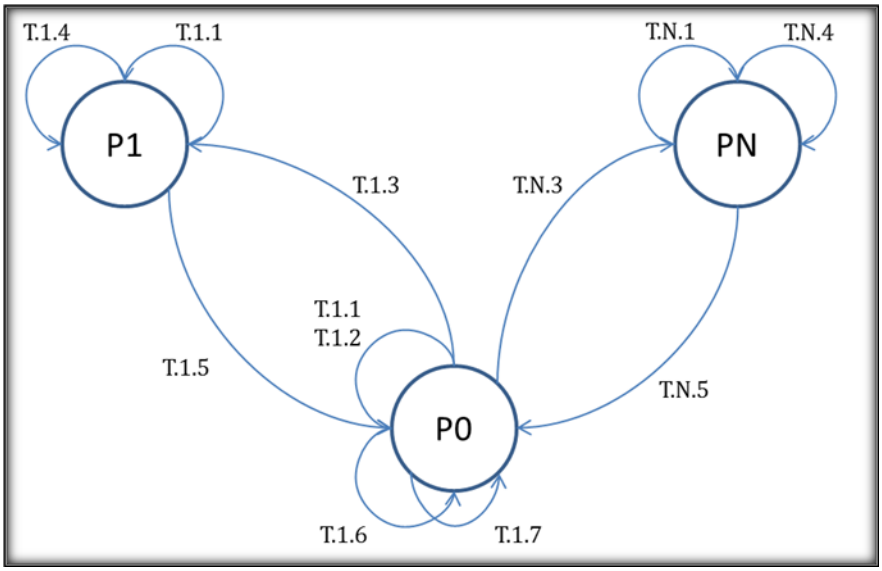


Figura 34. Grafo de la secuencia de actividades del algoritmo paralelizado únicamente con MPI.

En la Tabla 9 se describen de manera general las actividades que se realizan en la ejecución del código paralelo con MPI.

Tabla 9. Descripción general de las actividades del código paralelo con MPI.

ACTIV.	DESCRIPCIÓN
T.1.1	Inicializar de variables
T.1.2	Crear caminos
T.1.3	Enviar caminos
T.1.4	Realizar iteraciones de los caminos
T.1.5	Enviar camino con la menor distancia y su costo
T.1.6	• Determinar y almacenar el camino con la menor distancia • Selección de caminos • Cruzamiento de caminos seleccionados • Comparar el camino de menor distancia con el de la anterior y almacenar el de menor costo (regresa a T.1.3 mientras g <generaciones)
T.1.7	Imprimir el camino de menor distancia y su costo

En la actividad T.1.1 todos los procesos inicializan las variables, vectores y matrices necesarias para realizar las operaciones requeridas durante las iteraciones. Además, se carga en memoria la matriz de distancias entre las ciudades, la cual se encuentra en un archivo de texto.

El *proceso 0* es el único que realiza la actividad T.1.2, ya que es el que se encarga de centralizar todos los caminos para determinar el de menor distancia en cada generación, así como realizar el cruzamiento de los mismo. En la *generación 0* el *proceso 0* crea *N* caminos aleatorios, mismos que serán enviados a los demás procesos (T.1.3) y de esta manera cada proceso itera con un camino diferente. Una vez que se han enviado los

caminos, el *proceso 0* se queda en espera de recibir de vuelta el camino de cada uno de los otros procesos.

La actividad T.1.4 es realizada por todos los procesos, excepto el *proceso 0*, y es en este punto en donde se lleva a cabo el trabajo más pesado del algoritmo. Cada proceso realiza una serie de permutaciones al camino que se le asignó, calculando por cada cambio la distancia del nuevo camino generado y comparándolo con el anterior para que al final del ciclo envíen al *proceso 0* el camino con la menor distancia que se haya generado (T.1.5).

Una vez que el *proceso 0* recibe todos los caminos de vuelta, los almacena en una matriz y, en un vector, las distancias en la posición correspondiente con el camino (T.1.6). Posteriormente determina la menor distancia y la almacena, así como su respectivo camino. Luego se ejecutan los operadores del algoritmo genético: primero realiza una *selección* aleatoria del 50% de la población y toma la decisión de realizar o no el cruzamiento; en caso de que el random sea menor al 80%, se *cruzan* los dos mejores caminos de la selección y se almacena el resultado en una población temporal, de lo contrario, se almacenan los padres. Si hubo cruzamiento, se decide si hay *mutación*; en caso de obtener un random de 20% se realiza una permutación aleatoria en el primer hijo del cruzamiento. Este proceso de selección, cruzamiento y mutación se realiza hasta haber llenado la población tempora, que después se vacía en la población actual. A partir de la segunda generación, realiza la comparación entre el mejor camino de la generación anterior con el de la actual para determinar el de menor distancia.

Si aún no se ha ejecutado el número de generaciones establecidas, se repite el ciclo desde la actividad T.1.3, con la diferencia que se envían los caminos resultantes del cruzamiento. Después de que se hayan realizado el número de generaciones, el *proceso 0* imprime el vector con el camino de menor distancia encontrado en toda la ejecución y su costo. Finalmente, todos los procesos liberan la memoria ocupada.

Ejecución de Pruebas Locales para el Algoritmo Genético Paralelo

Después de haber agregado al algoritmo los operadores genéticos, se realizaron algunas pruebas de manera local tanto en el clúster Nopal (ITV) como en el clúster CIICAp (UAEM). En el primero se tiene un total de 40 cores, por lo que se evaluaron 39 caminos (Tabla 10); mientras que en el segundo se tienen 12 cores, con lo que se evaluaron 11 caminos (Tabla 11). En el clúster de la UPEMOR no se realizaron pruebas debido a razones propias en la universidad

Tabla 10. Resultados del algoritmo genético paralelizado ejecutado en el clúster Nopal, evaluando 39 caminos.

CONCEPTO	VALOR
Menor tiempo ejecución obtenido:	15 min. 56seg.
Menor distancia obtenida:	188,529

Tabla 11. Resultados del algoritmo genético paralelizado ejecutado en el clúster CIICAp, evaluando 11 caminos.

CONCEPTO	VALOR
Menor tiempo ejecución obtenido:	22 min. 0seg.
Menor distancia obtenida:	188,857

A la fecha, las pruebas en la grid no se han podido realizar debido a algunos inconvenientes en la conexión de la VPN; sin embargo, con las pruebas realizadas entre la UAEM y la UPEMOR se puede notar que, a pesar de haber triplicado el número de procesos, el incremento en el tiempo no fue proporcional. Además, los nodos más rápidos (QuadCore) quedan aproximadamente la mitad del tiempo ociosos, tiempo que se puede emplear haciendo que el *proceso 0* envíe caminos por demanda, con lo que se podrán evaluar más caminos que la cantidad cores disponibles en la grid.

Resultados alcanzados

- Implementación de tres tarjetas de red para el acceso a través de Internet1 e Internet2 al clúster Nopal.
- Modificación del algoritmo secuencial y creación de funciones.
- Registro de los resultados del código secuencial obtenidos de la ejecución en el clústerCIICAp para la evaluación de un camino.
- Implementación de un método propio de cruzamiento y ajuste para los caminos ya evaluados por los nodos.
- Paralelización del código empleando OpenMPy MPI.
- Registro de los resultados del código paralelizado obtenidos de la ejecución en el clúster Nopal y CIICAp.
- Versión final del algoritmo genético paralelo.
- Realización de pruebas locales con el algoritmo genético paralelo.

2.5. Algoritmo paralelo multinivel niveles (SMP, Clúster y Grid) del GA para la solución del problema de ruteo vehicular con ventanas de tiempoVRPTW.

2.5.1. Definición del problema VRPTW

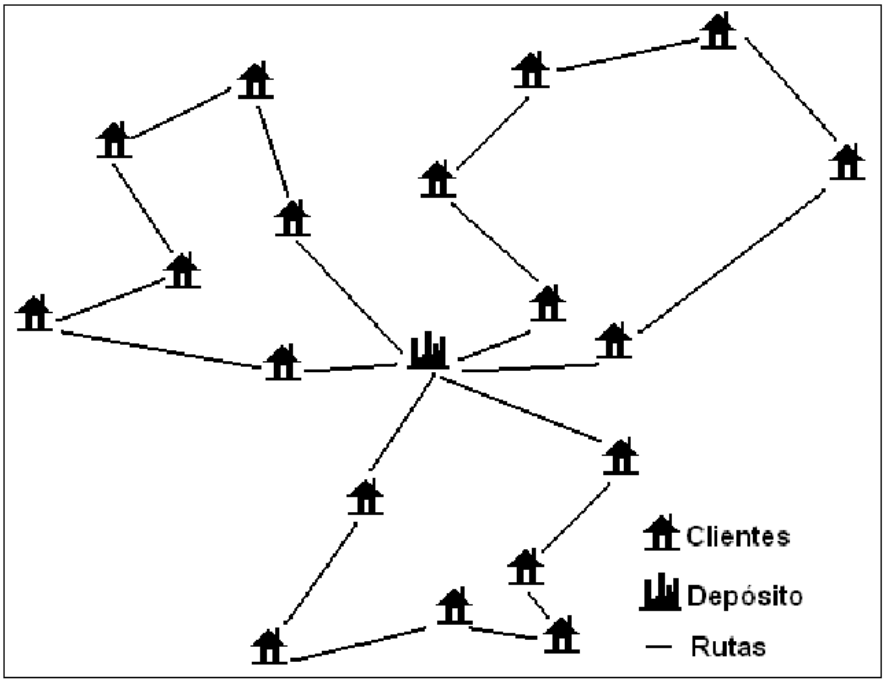


Figura 35: Muestra de problema VRPTW

En el problema de ruteo de vehículos con ventanas de tiempo consiste en obtener la menor distancia y el menor número de vehículos para repartir suministros a clientes geográficamente dispersos pero se tiene una serie de restricciones:

e) Cada vehículo tiene una capacidad limitada (para efectos de simplificación se concibe una flota de vehículos toda con igual capacidad).

f) Varios vehículos pueden satisfacer en forma conjunta un pedido de un cliente.

g) Cada cliente debe ser visitado dentro de una determinada ventana de tiempo.

h) El vehículo tiene la alternativa de llegar al punto de entrega antes del inicio de la ventana de tiempo y esperar a que esta inicie.

i) Se considera una velocidad promedio para la flota de vehículos.

j) Se visita cada punto de entrega una única vez.

2.5.2. Evaluación de rendimiento de las 2 versiones a nivel 2

No se ha podido realizar la evaluación por problemas técnicos y contratiempos en la implementación de la paralelización se seguirá con esta actividad durante las próximas semanas cuando los problemas técnicos sean resueltos ya que de momento la grid no se puede ocupar debido a que están haciendo pruebas de latencia y se está comprobando que se puedan ejecutar los programas con paso de mensaje sobre todos los clústeres de la grid sin problemas.

2.5.3. Versión paralela del VRPTW a nivel 3 (con MPI en la Grid con manejadores)

En esta actividad se busca minimizar los mensajes entre los clústeres, al crear un nodo líder por cada clúster, esto se hace porque si cada nodo tiene que difundir un mensaje a los otros nodos en todos los clústeres, en lugar de que cada nodo mande un mensaje a todos los demás nodos, es mejor que se recolecte la información en el nodo líder del clúster correspondiente y este a su vez mande la información a los otros líderes para que estos repartan la información entre los nodos de su mismo clúster. A partir de esto se empezó con el desarrollo del algoritmo genético con manejadores encontrando la dificultad de que para optimizar el algoritmo genético con manejadores se necesita asignar los rangos a las maquinas de tal forma que se sepa con certeza qué rangos de procesadores corresponden a qué nodos de que clúster.
Por esta razón se buscaron formas de resolver este inconveniente encontrando que hay una librería llamada MagPie, esta optimiza el paso de mensajes en grid de área amplia minimizando los mensajes que manda un nodo de un clúster a los nodos de otro transmitiendo solamente un mensaje por la red entre los clúster y luego repartiendo los mensajes dentro de la red interna del clúster destino, haciendo llegar el mensaje a cada nodo que lo necesite. Esta librería funciona teniendo como base un software llamado Panda que define qué nodos pertenecen a qué clúster y que tan eficiente es mandar un mensaje desde cada máquina a las otras.
Otra herramienta que se encontró es MPICHG que es la librería MPICH que es una implementación de MPI optimizada para ambientes homogéneos y redes myrinet y Globus, este último es un software para la construcción de grids en MPICHG globus y se encarga de identificar los nodos de cada clúster y en la comunicación de los mensajes entre ellos.
El problema con estas herramientas es que necesitan el soporte de software para identificar qué nodos son de cada clúster por lo que se está buscando la forma de asignar el rango de los procesadores de forma manual para no necesitar la instalación de este software pero teniendo la ventaja de poder minimizar los mensajes entre los clústeres.
Dado los problemas que se encontraron se decidió posponer la actividad para trabajos futuros para no seguir retrasando la investigación actual, siguiendo con las demás actividades mientras se encuentra una solución al problema de asignación de rango necesario para poder organizar por manejadores.

2.5.4. Integración de 3 versiones de paralelismo en una sola.

Hasta el momento se tienen dos versiones solamente las cuales se integran para conformar un solo algoritmo donde se pueda ejecutar ya sea la versión paralelizada con MPI, la que se realizo con OpenMP o con ambas versiones a la vez. Cuando se tenga la versión con manejadores también se integrara a esta.

Evaluación del rendimiento usando 2 benchmark de Solomon

En las próximas semanas se evaluará la paralelización del algoritmo usando benckmark de Solomon de forma local, solamente en el clúster del ITVER y en ambiente Grid ejecutándose entre los clústeres para evaluar el rendimiento del algoritmo.

Realizar la versión paralela del problema VRPTW a nivel 1 (con paralelismo OpenMP)

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
41.96	315.00	315.00	6902752	0.00	0.00	mayo
38.20	601.75	286.75	6902752	0.00	0.00	meno
19.80	750.42	148.66	20	7.43	37.52	mutacion
0.02	750.58	0.16	20	0.01	0.01	seccionbest
0.01	750.66	0.08	99	0.00	0.00	Permutaciones
0.00	750.67	0.01	20	0.00	0.00	calculadisttotal
0.00	750.67	0.00	100	0.00	0.00	CALCULARUTAS
0.00	750.67	0.00	20	0.00	0.00	crossover
0.00	750.67	0.00	1	0.00	0.00	ArchivaRes
0.00	750.67	0.00	1	0.00	0.00	MATDistancias
0.00	750.67	0.00	1	0.00	750.67	algoritmogeneticoKOKO
0.00	750.67	0.00	1	0.00	0.08	genpobinialgo0
0.00	750.67	0.00	1	0.00	0.00	rutakmeans

Figura 36: Profile algoritmo VRPTW original

Fueron identificaron los métodos a paralelizar con OpenMP, se escogieron estos debido a que después de realizar un profile al algoritmo (el cual se muestra en la Figura 36) se encontró que dichos métodos, aunque consumen poco tiempo cada vez que son llamados, son llamados muchas veces. Por ejemplo en una ejecución de 20 generación cada método es llamado 6,902,752 veces generando un tiempo de 750 segundos, lo que hace que consuma el 80.16 % del tiempo total de la ejecución.
Estos son métodos de ordenamiento llamados “mayo” y “meno”, su función es ordenar un arreglo. En un principio se utilizaba el método burbuja para realizar el ordenamiento, este método no es muy eficiente por lo que se remplazó por el método de ordenamiento llamado quicksort. Además se encontró que el algoritmo ordenaba un arreglo de estructura de tamaño 25 cada vez que era llamado, sin embargo realmente no se ocupaba todo el arreglo, es por ello por lo que se cambio para que solo ordene la cantidad de espacio que se esté usando del arreglo mejorando su eficiencia. Con esta mejora el nuevo el tiempo consumido por mayo y meno es del

47.20% del tiempo total de ejecución el cual es 192 segundos. Con lo cual se logro reducir 75% el tiempo de ejecución del algoritmo para VRPTW de forma secuencial. En la figura 37 se puede apreciar la mejora en el tiempo que se obtuvo a adaptar el ordenamiento de acuerdo a la cantidad de espacio utilizado.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 52.56    101.33   101.33       20     5.07     9.62  mutacion
 28.56    156.39    55.06  6935000     0.00     0.00  meno
 12.93    181.31    24.92  6935000     0.00     0.00  mayo
  3.31    187.69     6.38  6935000     0.00     0.00  quicksortMayo
  1.01    189.63     1.95  6935000     0.00     0.00  quicksortMeno
  0.97    191.51     1.88 41754634     0.00     0.00  swapMayo
  0.50    192.48     0.97 19416274     0.00     0.00  swapMeno
  0.08    192.64     0.16       20     0.01     0.01  seccionbest
  0.06    192.76     0.12       99     0.00     0.00  Permutaciones
  0.02    192.80     0.04       20     0.00     0.00  calculadisttotal
  0.00    192.80     0.00      100     0.00     0.00  CALCULARUTAS
  0.00    192.80     0.00       20     0.00     0.00  crossover
  0.00    192.80     0.00        1     0.00     0.00  ArchivaRes
  0.00    192.80     0.00        1     0.00     0.00  MATDistancias
  0.00    192.80     0.00        1     0.00   192.80  algoritmogeneticoKOKO
  0.00    192.80     0.00        1     0.00     0.12  genpobinialgo0
  0.00    192.80     0.00        1     0.00     0.00  rutakmeans
```

Figura 37: Profile del algoritmo VRPTW modificado

Finalmente se paralelizó con OpenMP el método de ordenamiento de quicksort para mejorar el tiempo de ordenamiento del arreglo utilizado en mayo y meno. La paralelización realizada consiste en que el arreglo se divida en 4 hilos, con lo que tiene que bajar el tiempo de ordenación y el consumo de tiempo de mayo y meno. Cabe señalar que al algoritmo paralelo no se le puede hacer un profile de la misma manera que con el algoritmo secuencial, aunque existen algunas herramientas capaces de realizar esta tarea, no se cuenta con el software necesario.

Resultados alcanzados

Se logró reducir el tiempo de ejecución del algoritmo en su forma secuencial, Se remplazó el método de ordenamiento por el llamado quicksort. Además se realizó una adaptación para que el arreglo de estructura que se utiliza sólo ordene la cantidad de datos que contenga y no todo el arreglo siempre aunque este no se encuentre lleno. Con esta mejora el nuevo tiempo consumido por las funciones mayo y meno es del 47.20% del tiempo total de ejecución el cual es 192 segundos. Con ello se reduce en un 75% el tiempo de ejecución del algoritmo para VRPTW de forma secuencial.
Se paralelizó con OpenMP el método de ordenamiento de quicksort para mejorar el tiempo de ordenamiento del arreglo utilizado en mayo y meno. La paralelización realizada consiste en que el arreglo se divida en 4 hilos, con lo que tiene que bajar el tiempo de ordenación y el consumo de tiempo de mayo y meno.

La paralelización con Automap hizo posible el envió de estructuras complejas por medio de paso de mensajes y facilito la comunicación entre los nodos al poder usar envió colectivo para pasar la población mutada.

2.6. Artículos publicados.

Marco Antonio Cruz-Chávez, Abelardo Rodríguez-León, Erika Yesenia Ávila-Melgar, Fredy Juárez-Pérez, Martín H. Cruz-Rosales, Rafael Rivera-López, Genetic-Annealing Algorithm in Grid Environment for Scheduling Problems, Communications in Computer and Information Science: Security-Enriched Urban Computing and Smart Grid, Springer Verlag Pub., Berlin Heidelberg, ISSN: 1865-0929, Vol. 78, pp.1-9, 2010.

Abelardo Rodriguez-León, Marco Antonio Cruz-Chávez, Rafael Rivera-López, Erika Yesenia Ávila-Melgar, Fredy Juárez-Pérez, Ocotlán Díaz-Parra, A Communication Scheme for an Experimental Grid in the Resolution of VRPTW using an Evolutionary Algorithm, Electronics, Robotics and Automotive Mechanics Conference, CERMA2010, IEEE-Computer Society, ISBN 978-0-7695-4204-1, pp 108 - 113, September 28 - October 1, México, 2010.

2.7. Sitio Web de la Grid Tarántula acondicionada para el presente proyecto

http://www.gridmorelos.uaem.mx:8080/

3. Conclusiones

Las latencias medidas de la UPEMOR a los clusters CIICAp e ITVer en 5 días durante 16 horas de pruebas se observo que durante el día el uso de la red en la cual se encuentran conectados los *cluster* de la grid Tarántula se tenía demasiado tráfico secundario, de acuerdo a esto se observo que las latencias no resultaron bajas como se esperaba.
Se encontró que el aumento de latencia en promedio cuando existe carga con respecto a cuando no existe, es en un intervalo de 2500 a 3500ms conforme avanza el día. Es decir que si existe carga de procesos el mínimo aumento de latencia es de 2500ms y el máximo aumento de latencia es de 3500ms. Las latencias siguen estando altas de acuerdo a lo esperado, este resultado indica que la grid tarántula debe de aislarse del tráfico de entrad a y salida de cada una de las instituciones en donde se encuentran los clusters.

De acuerdo a las pruebas en el paso de mensajes utilizando librerías MPI se pudo ver que el único caso de éxito para el paso de mensajes fue en la comunicación bloqueante utilizando memoria estática, la transferencia fue exitosa. Cada envío y recepción es verificado en los nodos. Además que una vez modificada la información ésta imprime para verificar la consistencia de los datos. Gracias a que MPI provee sincronización en los nodos por medio de las rutinas de tipo bloqueante, se tiene la certeza que los datos no se cruzan, es decir que no recibe mientras envía o viceversa (Figura 5.6). Para los tipos de algoritmos generados en

esta investigación es necesario el uso de las comunicaciones bloqueantes que son la mejor opción de transferencia de datos, ya que es necesario que haya una sincronización de trabajo entre los nodos.
El paso de mensajes de estructuras de datos complejas con automap en la cual se involucra memoria estática, funcionó de manera exitosa al realizar paso de mensajes en la grid tarántula entre nodos de varios clusters.
La paralelización con Automap facilita mucho las comunicación de estructuras complejas al poder pasar una estructura completa en un sólo mensaje el cual puede ser colectivo, sólo se tiene que tener cuidado de no sobre cargar el tráfico de la red ya que los mensajes pueden hacerse muy pesados.
Ninguno de los métodos utilizados con la herramienta autopmap fue exitoso al utilizar memoria dinámica, cabe mencionar que se realizaron más métodos de forma parecida y se realizaron pruebas pero los resultados no fueron de éxito además de que se probaron la gran mayoría de instrucciones propias de MPI pero lamentablemente ninguna pudo ayudar a resolver el problema por lo cual esta investigación queda abierta para realizar un estudio apoyándose de esta investigación para entender con mayor facilidad el funcionamiento del paso de mensajes.

Para el genético que optimiza recursos en el trazo de caminos, crear un camino inicial de manera aleatoria permitió obtener mejores resultados con respecto a la implementada en la que se evaluaba inicialmente la secuencia 0, 1, 2, …, N.
Además, la aplicación de la paralelización empleando OpenMP permitió que la ejecución del algoritmo fuera muchos más rápida, obteniéndose mejores resultados en menor tiempo, mientras que la paralelización con MPI permitió ampliar el espacio de búsqueda ya que cada camino toma rumbos esparcidos, de manera que no estén tan cercanos entre sí. Aquí se nota la conveniencia de tener una grid en la cual al aumentar el número de nodos en un cluster hace posible que estos nuevos recursos se puedan utilizar de forma inmediata en la grid.
Al implementar el cruzamiento y evaluar por varias generaciones, se amplió aún más el espacio de búsqueda, con lo cual se evitó en mayor grado el estancamiento en un óptimo local.
Se debe conocer más a fondo el funcionamiento de la hibridación de MPI con OpenMP, ya que en las pruebas creó un conflicto y el tiempo de ejecución se incrementó considerablemente, por lo que es más conveniente emplear únicamente MPI para ejecutar en paralelo en todos los nodos.
De acuerdo a las pruebas realizadas en la grid, el incremento de nodos en la misma, relativamente no afectará el tiempo de ejecución del algoritmo, ya que los nodos más lentos hasta el momento demoran aproximadamente 5 minutos en terminar su trabajo. Bajo esta observación, es necesario contar con una grid lo más homogénea que sea posible.

Para el genético que optimiza recursos en el VRPTW, al mejorar el algoritmo de ordenación se logro optimizar considerablemente el algoritmo de VRPTW, lo que sirve de ejemplo que la paralelización es una parte importante de la optimización pero que sigue siendo imprescindible el buen manejo de las técnicas de programación y el conocimiento de los algoritmos.
Se paralelizó el algoritmo utilizando la herramienta OpenMP con 4 hilos al utilizar el método de ordenamiento de quicksort mejorando el tiempo de ejecución.

4. Bibliografía

1. Ignacio Martínez Fernández. “Creación y Validación de un Clúster de Computación Científica Basado en Rocks”. Leganés, Madrid. 2009.
2. Javier Panadero Martínez. “Entorno de desarrollo para clusters”. Bellaterra, Barcelona. Junio de 2008.
3. Pablo Abad, José Ángel Herrero, Rafael Menéndez de Llano, Sergio Garrido, Fernando Vallejo. “Análisis y evaluación de sistemas de colas en un entorno HTC”. Madrid, septiembre de 2003.
4. Díaz, Gilberto; Hamar, Vanessa; Hoeger, Herbert; Mendoza, Víctor; Ramírez, Yubiryn; Rojas, Freddy. “Herramientas GRID para la integración y administración de servicios de redes en Latino América”. Mérida, Venezuela. Diciembre de 2005.
5. Javier Panadero Martínez. “Entorno de desarrollo para clusters”. Bellaterra, Barcelona. Junio de 2008.
6. University of California and Scalable Systems. “SGE Roll: UsersGuide”, California, USA. Diciembre de 2009.
7. Marco Antonio Cruz-Chávez, Alina Martínez-Oropeza, Sergio A. Serna Barquera. “Neighborhood Hybrid Structure for Discrete Optimization Problems”. Cuernavaca, Morelos. 2010.
8. José Luis González Velarde, Roger Z. Río Mercado. “Investigación de Operaciones en Acción: Aplicación del TSP en Problemas de Manufactura y Logística”. Mayo, 1999.
9. Michael Hahsler and Kurt Hornik.“Introduction to TSP – Infrastructure for the Traveling Salesperson Problem”.Marzo 24, 2009.
10. Belén Melián, José A. Moreno Pérez, J. Marcos Moreno Vega. “Metaheuristics: A Global View”. Santa Cruz de Tenerife, España.
11. David S. Johnson and Lyle A. McGeoch. “The Traveling Salesman Problem: A Case Study in Local Optimization”. Noviembre 20, 1995.
12. S. B. Liu, K. M. Ng, and H. L. Ong.“A New Heuristic Algorithm for the Classical Symmetric Traveling Salesman Problem”.WorldAcademy of Science, Engineering and Technology 27 2007.
13. Jean Berger, Mohamed Barkaoui: “A parallel hybrid genetic algorithm for the vehicle routing problem with time windows”, Canada, 2004
14. Olatz Arbelaitz, Clemente Rodríguez y Ion Zamakola :”Análisis de eficiencia de una solución paralela al problema VRPTW”, Granada , 2000
15. M. Fisher: Vehicle Routing, en Handbook in Operations Research and Management Science, Vol. 8, Elsevier, 1995.
16. Barajas, N. (s.f.). Estado del arte del problema de ruteo de vehículos. Colombia.
17. Villalobos Arias, M. A. (Diciembre de 2003). Algoritmos Genéticos: Algunos resultados de convergencia. Distrito Federal, México.
18. Eveyn Quinshpe Goyes & Gustavo Recalde Vásquez (2008). Una metaheurística para la construcción de árboles filogenéticos basados en matrices de distancias.
19. Slezak, D. F., Turjanski, P., Silva, M., Monetti, J., Mocskos, E., García Garino, C., y otros. (Septiembre de 2005). Construcción de un laboratorio de GRID Virtual para HPC. Simposio Argentino de Tecnología en Computación. Argentina.
20. “MPI: A Message-Passing Interface Standard Version 2.2”. Message Passing Interface Forum, September 2009.
21. Ravi Shankar, “Message Passing Interface (MPI) Advantages and Disadvantages for applicability in the NoC Environment”. Florida Atlantic University – Computer Science & Engineering, 2005.
22. William Gropp, Ewing Lusk , “A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard”. Mathematics and Computer Science Division, Argonne National Laboratory.
23. William Gropp, “Tutorial on MPI: The Message-Passing Interface”. Mathematics and Computer Science Division, Argonne National Laboratory.
24. Judith Ellen Devaney, Martial Michel, Jasper Peeters, Eric Baland, “AutoMap: A Data Structures Compiler for the Automatic Generation of MPI Data Structures Directly From C Code”. NIST, ESIAL, University of Twente, Institut National des Telecommuncations, 1997.
25. Martial Michel, Andre Schaff, Judith Ellen Devaney, “Managing data-types: the CORBA Approach and AutoMap/AutoLink, an MPI Solution”.

26. Alonso Jordá Pedro, García Granada Fernando, Onaindía de la Rivaherrera Eva. *Diseño e implementación de programas en lenguaje C.* Valencia: Universidad Politécnica del Valencia.
27. Álvarez Cáceres Rafael. *El método científico en las ciencias de la salud.* Madrid: Díaz de Santos.
28. Cairo Osvaldo. *Fundamentos de programación. Piensa en C.* México: PEARSON EDUCACION.
29. Ceballos Francisco. Javier. *Enciclopedia del lenguaje C.* Madrid: RA-MA Editorial.
30. Cegarra José, *Metodología de la investigación científica y tecnológica.* Madrid: Diaz de Santos.
31. Coulouris, G., Dollimore, J., Kingberg, T. *Distribuited System.*China: Pearson Education Limited.
32. Deitel, Harvey M., Paul J. *Como programar en C/C++ y java.* México: Pearson Educación.
33. Fernández, M. *Nuevas tendencias en la informática.*España: COMPOBELL.
34. Garrido Carrillo Antonio, Fernández Valdivia Joaquín. *Abstracción y estructuras de datos en C++*. Madrid: DELTA publicaciones.
35. Grama, A., Gupta, A., Karypis, G., Kumar, V. *Introduction to Parallel Computing.* England: Pearson.
36. Geist Al, Buguelin Adam, Dongarra Jack, Jiang Weicheng, Mancheck Robert, Sunderam Vaidy. *Parallel Virtual Machine*. United States of America: Massachusetts Institute of Technology.
37. Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W.*MPI.*Netherlands: IOS Amsterdan.
38. Gropp William, Lusk Ewing, Skjellum Anthony. *Using MPI: Portable parallel programming with the Message-Passing Interface.* Estados Unidos de America: Massachusetts Institute of Technology.
39. Jacob Bart, Brown Michael, Fukui Ken taro, Trivedi Nihar. *Introduction to GRID computing.* United States of America: IBM.
40. Em Karniadakis, G., Kirby II, R. *Parallel Scientifc Computing in C++ and MPI.*United States of America: Cambridge.
41. Kernighan W. Brian, Ritchie M. Dennis. *El lenguaje de programación C. México*: Pearson Educación.
42. Kooper Karl. *Linux Enterprise Cluster.* United States of America: No Start Press, Inc.
43. Leithold Alfred. *Structured testing in practice.* Alemania: 2007.
44. Lence, P. *Técnicas paralelas aplicadas a optimización no lineal en sistemas de memoria distribuida.*España: USC.
45. Maozhen Li, Baker Mark. *The Grid core technologies.* Inglaterra: Wiley.
46. Martínez M. Santiago. *Difusión automático de datos bajo cómputo paralelo en clusters.* México D.F.: IPN
47. Martínez Gil Francisco, Martin Quetglás Gregorio. *Introducción a la programación estructurada en C.* España: Maite Simon.
48. Morton E. Thomas, David W. Pentico. *Heuristic Scheduling Systems.* Canadá: ETA.
49. Munilla Calvo Eduardo, García Valcárcel Ignacio. *Cómo implantar Software libre, Servicios web y el GRID computing para ahorrar costes y mejorar las comunicaciones en su empresa.* Madrid: FC Editorial.
50. Pacheco, P. *Parallel Programming with MPI.*United States of America: Morgan Kauímann Publisher, Inc.
51. Prabhu, C.S.R. *Grid and Cluster Computing.* India: Easter PHI Learning Pvt. Ltd
52. Rosano, F. L. *Tecnologia conceptos, problemas y perspectivas.* Mexico D.F.: Siglo veitiuno editores.
53. Sedgewick Robert. *Algoritmos en C++*. Massachusetts: Addison-Wesley Publishing Company, Inc.
54. Sloan D. Joseph. *High performance LINUX Clusters with OSCAR, Rocks, openMosix & MPI.* Estados Unidos de América: O'Reilly.
55. Somerville Ian. *Ingeniería del Software.* Madrid: Pearson Educación, S.A.
56. Tamayo Mario. *El proceso de la investigación científica.* México: LIMUSA.
57. Wang Jun, Yi Zhang, Zurda M. Jacke, Lu Bao-Liang, Yin Hujun. *Advances in Neuronal Networks – ISNN 2006.* China: Springer.