DESARROLLO DE APLICACIONES PARALELAS PARA *CLUSTERS* UTILIZANDO MPI (*MESSAGE PASSING INTERFACE*)

Bernal C. Iván, Mejía N. David y Fernández A. Diego

Escuela Politécnica Nacional
Quito-Ecuador

Resumen

En la actualidad, es factible disponer de alta capacidad computacional mediante clusters de computadoras personales independientes, de bajo costo, interconectadas con tecnologías de red de alta velocidad, y empleando software de libre distribución.

El aparecimiento de la computación paralela permitió que emerjan métodos de programación que hagan posible la implementación de algoritmos, utilizando recursos compartidos: procesador, memoria, datos y servicios.

Este artículo presenta las ideas básicas involucradas en el desarrollo de aplicaciones paralelas, presentando aspectos relacionados a lenguajes de programación que proveen soporte para desarrollo de aplicaciones paralelas a través de librerías de paso de mensajes. Se presentan las diferentes alternativas de librerías de paso de mensajes, su arquitectura y algunas consideraciones de diseño de aplicaciones. También se presenta una aplicación que permite resolver Cadenas de Markov y que fue ejecutada sobre un cluster construido con la herramienta NPACI Rocks, y se presentan los resultados obtenidos al ejecutar la aplicación.

1. Introducción a los *clusters*

Un *cluster* es una solución computacional conformada por un conjunto de sistemas computacionales muy similares entre si (grupo de computadoras), interconectados mediante alguna tecnología de red de alta velocidad, configurados de forma coordinada para dar la ilusión de un único recurso; cada sistema estará proveyendo un mismo servicio o ejecutando una (o parte de una) misma aplicación paralela. La característica inherente de un *cluster* es la compartición de recursos: ciclos de CPU *(Central Processing Unit)*, memoria, datos y servicios.

Los *clusters* están conformados por computadoras genéricas con uno o más procesadores, denominados nodos. Dichos nodos pueden estar dedicados exclusivamente a realizar tareas para el *cluster*, por lo que no requieren de monitor, teclado o mouse; o pueden estar dedicados a diferentes actividades y se utilizarán los ciclos libres del procesador para realizar las tareas que requiera el *cluster*.

La idea de los *clusters* tomó impulso en los 90s, cuando se dispuso de microprocesadores de alto rendimiento, redes de alta velocidad, y herramientas estándar para computación distribuida (*Message Passing Interface*, MPI, *Parallel Virtual Machine*, PVM ([1], [2])) y a costos razonables. Pero también el desarrollo de los *clusters* fue impulsado por deficiencias de los Sistemas Multiprocesador Simétricos (*Symmetric MultiProcessors*, SMP*s* [3]). Las grandes máquinas SMP son costosas, propietarias, tienen un único punto de falla, no están ampliamente disponibles, y sufren de problemas de escalabilidad, en términos de número de procesadores y capacidad de memoria. Según [4], los sistemas SMP más grandes conocidos, escalan hasta alrededor de 128 CPU*s*.

En 1994, T. Sterling y D. Becker, trabajando en CESDIS (*Center of Excellence in Space Data and Information Sciences*) bajo el patrocinio del Proyecto de la Tierra y Ciencias del Espacio (ESS), construyeron un *cluster* de computadoras que consistía de 16 procesadores 486DX4, usando una red Ethernet a 10Mbps, con un costo de $40,000. El rendimiento del *cluster* era de 3.2 Gflops. Ellos llamaron a su sistema *Beowulf*, un éxito inmediato, y su idea de proporcionar sistemas en base a COTS (*Components Of The Shelve*) para satisfacer requisitos de cómputo específicos, se propagó rápidamente a través

imbernal@mailfie.epn.edu.ec,
david_dam33@hotmail.com,
d_fernandezayala@hotmail.com

de la NASA y en las comunidades académicas y de investigación. En la actualidad, muchos *clusters* todavía son diseñados, ensamblados y configurados por sus propios operadores; sin embargo, existe la opción de adquirir *clusters* prefabricados.

Existen paquetes de software que automatizan el proceso de instalación, de configuración y de administración de un *cluster,* denominados *toolkits*. Este conjunto de paquetes permite configurar un *cluster* completo en una fracción del tiempo que tomaría el hacerlo de forma manual. Estos *toolkits,* para instalación automática de *clusters,* pueden incluir una distribución de Linux; mientras que otros se instalan sobre una instalación existente de Linux. Sin embargo, incluso si primero se debe instalar Linux, los *toolkits* realizan la configuración e instalación de los paquetes requeridos por el *cluster* de forma automática. Del conjunto de *toolkits* existentes se pueden mencionar a NPACI *Rocks* [5] y a OSCAR [6].

NPACI (*National Partnership for Advanced Computational Infrastructure*) *Rocks* es una colección de software de código abierto para crear un *cluster* sobre Red Hat Linux. *Rocks* instala tanto Linux como software para *clusters*. La instalación toma unos pocos minutos.

OSCAR es una colección de software de código abierto que se instala sobre una instalación existente de Linux (Red Hat, Mandrake, Mandriva, Fedora).

Los programas desarrollados para *clusters* usualmente están escritos en C o en Fortran, y utilizan librerías de paso de mensajes para realizar operaciones paralelas; también pueden hacer uso de librerías matemáticas para resolución de problemas que involucren matrices, derivación e integración compleja.

Las librerías para paso de mensajes permiten escribir programas paralelos eficientes, proveen rutinas para inicializar y configurar el ambiente de mensajes, así como para enviar y recibir paquetes de datos. Los sistemas más populares de paso de mensajes son: PVM (*Parallel Virtual Machine*) del Laboratorio Nacional Oak Ridge y MPI (*Message Passing Interface*) definido por el Foro MPI.

PVM es una librería de paso de mensajes. Puede usarse para desarrollar y ejecutar aplicaciones paralelas en sistemas que están dentro del rango que va desde supercomputadoras hasta *clusters* de estaciones de trabajo.

MPI es una especificación de paso de mensajes, diseñada para ser el estándar de computación paralela de memoria distribuida usando paso de mensajes. Esta interfaz intenta establecer un estándar práctico, eficiente, portátil y flexible para el paso de mensajes.

2. Clasificación de los *clusters*

El término *cluster* tiene diferentes connotaciones para diferentes grupos de personas. Los tipos de *clusters*, establecidos en base al uso que se de a los *clusters* y los servicios que ofrecen, determinan el significado del término para el grupo que lo utiliza.

2.1 *High Performance*

- Para tareas que requieren gran poder computacional, grandes cantidades de memoria, o ambos a la vez.
- Las tareas podrían comprometer los recursos por largos períodos de tiempo.

2.2 *High Availability*

- Máxima disponibilidad de servicios.
- Rendimiento sostenido.

2.3 *High Throughput*

- Independencia de datos entre las tareas individuales.
- El retardo entre los nodos del *cluster* no es considerado un gran problema.
- La meta es el completar el mayor número de tareas en el tiempo más corto posible.

Los *clusters* se los puede también clasificar como *Clusters* de IT Comerciales (*High Availability, High Throughput*) y *Clusters* Científicos (*High Performance*) [4]. A pesar de las discrepancias a nivel de requerimientos de las aplicaciones, muchas de las características de las arquitecturas de hardware y software, que están por debajo de las aplicaciones en todos estos *clusters*, son las mismas. Más aun, un *cluster* de determinado tipo, puede también presentar características de los otros.

3. Herramientas de Desarrollo para Aplicaciones

Mediante la implementación de tareas paralelas es posible proveer solución a ciertos

problemas computacionales de cálculos intensivos.

El paralelismo se puede implementar mediante una aproximación del modelo cliente–servidor, llamado maestro–esclavo (*master–worker)*. En el modelo maestro-esclavo se divide el problema computacional en tareas independientes, el maestro coordina la solución del problema computacional, asignando tareas independientes al resto de procesos (esclavos).

El maestro realiza la asignación inicial de las tareas a los esclavos, realiza sus tareas y espera por la finalización del procesamiento de las tareas en los esclavos, para proceder a recopilar los resultados desde los esclavos.

Para escribir programas paralelos, se puede hacer uso del modelo de paso de mensajes, con PVM o MPI, o se puede utilizar un modelo de memoria compartida, con OpenMP.

La diferencia entre el modelo de paso de mensajes y el modelo de memoria compartida está en el número de procesos o hilos activos. En un modelo de paso de mensajes, durante la ejecución del programa, todos los procesos se encuentran activos. Por el contrario, en un modelo de memoria compartida, sólo existe un hilo activo al iniciar y al finalizar el programa; sin embargo, durante la ejecución del programa, la cantidad de hilos activos puede variar de forma dinámica.

3.1. OpenMP

OpenMP es un conjunto de librerías para C y C++, regidas por las especificaciones ISO/IEC (*International Standard Organization - International Engineering Consortium*), basado en el uso de directivas para ambientes paralelos.

OpenMP tiene soporte para diferentes sistemas operativos como UNIX, Linux, y Windows.

Un programa escrito con OpenMP inicia su ejecución, en un sólo hilo activo, llamado maestro. El hilo maestro ejecuta una región de código serial antes de que la primera construcción paralela se ejecute. Bajo el API de OpenMP la construcción paralela se obtiene mediante directivas paralelas. Cuando se encuentra una región de código paralelo, el hilo maestro crea (*fork*) hilos adicionales, convirtiéndose en el líder del grupo. El hilo maestro y los nuevos hilos ejecutan de forma concurrente la sección paralela (realizan trabajo compartido). Unirse al grupo (*join*) es el procedimiento donde al finalizar la ejecución de la región de código paralelo los hilos adicionales se suspenden o se liberan, y el hilo maestro retoma el control de la ejecución. Este método se conoce como *fork & join*.

En la Figura 1 se indica como un hilo maestro encuentra una sección de código paralelo y crea hilos adicionales para ejecutar dicha sección. Una vez realizadas las tareas de ejecución, el hilo maestro retoma el control del programa.

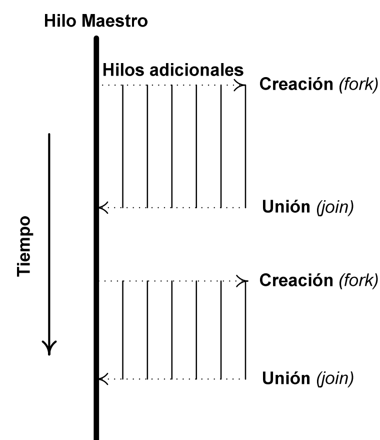

Figura 1. Modelo de ejecución fork & join

3.2. PVM

PVM fue desarrollado a principios de los 90s, por el Laboratorio Nacional Oak Ridge, en Estados Unidos.

PVM se encuentra disponible para sistemas Linux o Windows NT/XP. PVM está disponible para los lenguajes C, C++ y Fortran.

PVM es un conjunto de herramientas y librerías. Está compuesto de dos partes: de un proceso demonio y de librerías basadas en rutinas. El proceso demonio se denomina pvmd3.

La interfaz de la librería PVM contiene las primitivas necesarias para la cooperación entre las tareas de una aplicación. Define las rutinas para paso de mensajes, sincronización de tareas y creación de procesos.

Las implementaciones de PVM para los lenguajes de programación C y C++ utilizan una interfaz con funciones basadas en las

convenciones del lenguaje C, que permiten acceder a sus diferentes librerías. En el lenguaje Fortran, la funcionalidad de PVM se implementa como subrutinas en lugar de funciones.

3.3. MPI

La primera versión del estándar MPI aparece en Mayo de 1994. A mediados de 1995, aparece la Versión 1.1, en la cual se agregaron algunas aclaraciones y refinamientos. Las Versiones 1.0 y 1.1 fueron diseñadas para los lenguajes C y Fortran 77.

En Marzo de 1995, se extendió la versión original con la creación del estándar MPI Versión 2. La Versión 2 incluye la implementación para C++ y Fortran 90.

MPI no es un lenguaje de programación, es un conjunto de funciones y macros que conforman una librería estándar de C y C++, y subrutinas en Fortran.

MPI ofrece un API, junto con especificaciones de sintaxis y semántica que explican como sus funcionalidades deben añadirse en cada implementación que se realice (tal como almacenamiento de mensajes o requerimientos para entrega de mensajes). MPI incluye operaciones punto a punto y colectivas, todas destinadas a un grupo específico de procesos.

MPI realiza la conversión de datos heterogéneos como parte transparente de sus servicios, por medio de la definición de tipos de datos específicos para todas las operaciones de comunicación. Se pueden tener tipos de datos definidos por el usuario o primitivos.

4. Fundamentos de MPI

Con MPI el número de procesos requeridos se asigna antes de la ejecución del programa, y no se crean procesos adicionales mientras la aplicación se ejecuta.

A cada proceso se le asigna una variable que se denomina *rank*, la cual identifica a cada proceso, en el rango de 0 a p-1, donde p es el número total de procesos.

El control de la ejecución del programa se realiza mediante la variable *rank*; la variable *rank* permite determinar que proceso ejecuta determinada porción de código.

En MPI se define un *comunicator* como una colección de procesos, los cuales pueden enviar mensajes el uno al otro; el *comunicator* básico se denomina MPI_COMM_WORLD y se define mediante un macro del lenguaje C. MPI_COMM_WORLD agrupa a todos los procesos activos durante la ejecución de una aplicación.

Las llamadas de MPI se dividen en cuatro clases:

1. Llamadas utilizadas para inicializar, administrar y finalizar comunicaciones.
2. Llamadas utilizadas para transferir datos entre un par de procesos.
3. Llamadas para transferir datos entre varios procesos.
4. Llamadas utilizadas para crear tipos de datos definidos por el usuario.

La primera clase de llamadas permiten inicializar la librería de paso de mensajes, identificar el número de procesos *(size)* y el rango de los procesos *(rank)*. La segunda clase de llamadas incluye operaciones de comunicación punto a punto, para diferentes tipos de actividades de envío y recepción. La tercera clase de llamadas son conocidas como operaciones grupales, que proveen operaciones de comunicaciones entre grupos de procesos. La última clase de llamadas provee flexibilidad en la construcción de estructuras de datos complejos.

En MPI, un mensaje está conformado por el cuerpo del mensaje, el cual contiene los datos a ser enviados, y su envoltura, que indica el proceso fuente y el destino. En la Figura 2 se muestra un mensaje típico de MPI.

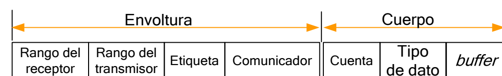

Figura 2. Formato de un mensaje de MPI

El cuerpo del mensaje en MPI se conforma por tres piezas de información: *buffer*, tipo de dato y *count*. El *buffer*, es la localidad de memoria donde se encuentran los datos de salida o donde se almacenan los datos de entrada. El tipo de dato, indica el tipo de los datos que se envían en el mensaje. En casos simples, éste es un tipo básico o primitivo, por ejemplo, un número entero, y que en aplicaciones más avanzadas puede ser un tipo de dato construido a través de datos primitivos. Los tipos de datos derivados son análogos a las estructuras de C. El *count* es un número de secuencia que junto al tipo de datos permiten al usuario agrupar ítems de datos de un mismo

tipo en un solo mensaje. MPI estandariza los tipos de datos primitivos, evitando que el programador se preocupe de las diferencias que existen entre ellos, cuando se encuentran en distintas plataformas.

La envoltura de un mensaje en MPI típicamente contiene la dirección destino, la dirección de la fuente, y cualquier otra información que se necesite para transmitir y entregar el mensaje. La envoltura de un mensaje en MPI, consta de cuatro partes: la fuente, el destino, el *comunicator* y una etiqueta. La fuente identifica al proceso transmisor. El destino identifica al proceso receptor. El *comunicator* especifica el grupo de procesos a los cuales pertenecen la fuente y el destino. La etiqueta (*tag*) permite clasificar el mensaje.

El campo etiqueta es un entero definido por el usuario que puede ser utilizado para distinguir los mensajes que recibe un proceso. Por ejemplo, se tienen dos procesos A y B. El proceso A envía dos mensajes al proceso B, ambos mensajes contienen un dato. Uno de los datos es utilizado para realizar un cálculo, mientras el otro es utilizado para imprimirlo en pantalla. El proceso A utiliza diferentes etiquetas para los mensajes. El proceso B utiliza los valores de etiquetas definidos en el proceso A e identifica que operación deberá realizar con el dato de cada mensaje.

4.1. Llamadas utilizadas para inicializar, administrar y finalizar comunicaciones

MPI dispone de 4 funciones primordiales que se utilizan en todo programa con MPI. Estas funciones son MPI_Init, MPI_Comm_size, MPI_Comm_rank y MPI_Finalize.

MPI_Init permite inicializar una sesión MPI. Esta función debe ser utilizada antes de llamar a cualquier otra función de MPI.

MPI_Finalize permite terminar una sesión MPI. Esta función debe ser la última llamada a MPI que un programa realice. Permite liberar la memoria usada por MPI.

MPI_Comm_size permite determinar el número total de procesos que pertenecen a un *comunicator*.

MPI_Comm_rank permite determinar el identificador (*rank*) del proceso actual.

4.2. Llamadas utilizadas para transferir datos entre dos procesos

Ls transferencia de datos entre dos procesos se consigue mediante las llamadas MPI_Send y MPI_Recv. Estas llamadas devuelven un código que indica su éxito o fracaso.

MPI_Send permite enviar información desde un proceso a otro. MPI_Recv permite recibir información desde otro proceso. Ambas funciones son bloqueantes, es decir que el proceso que realiza la llamada se bloquea hasta que la operación de comunicación se complete.

Las versiones no bloqueantes de MPI_Send y MPI_Recv son MPI_Isend y MPI_Irecv, respectivamente. Estas llamadas inician la operación de transferencia pero su finalización debe ser realizada de forma explícita mediante llamadas como MPI_Test y MPI_Wait. MPI_Wait es una llamada bloqueante y retorna cuando la operación de envío o recepción se completa. MPI_Test permite verificar si la operación de envío o recepción ha finalizado, esta función primero chequea el estado de la operación de envío o recepción y luego retorna.

4.3. Llamadas utilizadas para transferir datos entre varios procesos

MPI posee llamadas para comunicaciones grupales que incluyen operaciones tipo difusión (*broadcast*), recolección (*gather*), distribución (*scatter*) y reducción. Algunas de las funciones que permiten realizar transferencia entre varios procesos se presentan a continuación.

MPI_Barrier permite realizar operaciones de sincronización. En estas operaciones no existe ninguna clase de intercambio de información. Suele emplearse para dar por finalizada una etapa del programa, asegurándose de que todos los procesos han terminado antes de dar comienzo a la siguiente.

MPI_Bcast permite a un proceso enviar una copia de sus datos a otros procesos dentro de un grupo definido por un *comunicator*. En la Figura 3 se muestra la transferencia de información entre diferentes procesos usando la llamada MPI_Bcast.

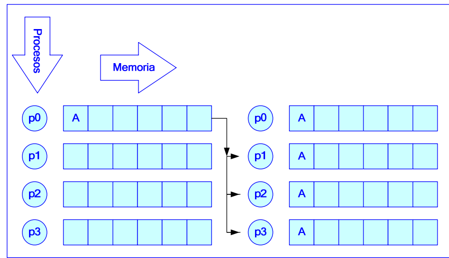

Figura 3. Operación de MPI_Bcast

MPI_Scatter establece una operación de distribución, en la cual un dato (arreglo de algún tipo de datos) se distribuye en diferentes procesos. En la Figura 4 se muestra esta operación.

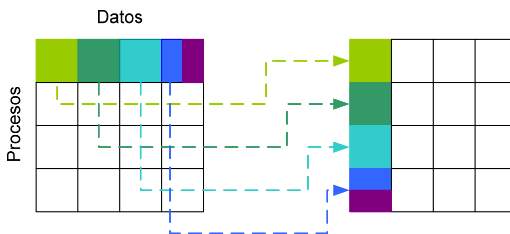

Figura 4. Operación de MPI_Scatter

MPI_Gather establece una operación de recolección, en la cual los datos son recolectados en un sólo proceso. En la Figura 5 se muestra la operación de recolección.

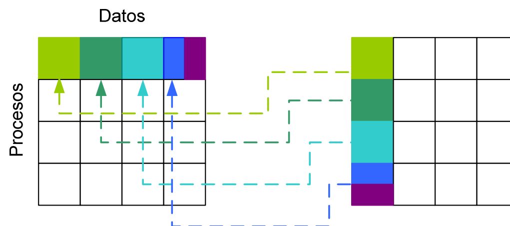

Figura 5. Operación de MPI_Gather

MPI_Reduce permite que el proceso raíz recolecte datos desde otros procesos en un grupo, y los combine en un solo ítem de datos. Por ejemplo, se podría utilizar una operación reducción, para calcular la suma de los elementos de un arreglo que se distribuyó en algunos procesos. La Figura 6 muestra como los datos son colectados en un sólo proceso.

4.4. Llamadas utilizadas para crear tipos de datos definidos por el usuario

Para definir nuevos tipos de datos se puede utilizar la llamada MPI_Type_struct para crear un nuevo tipo o se puede utilizar la llamada MPI_Pack para empaquetar los datos.

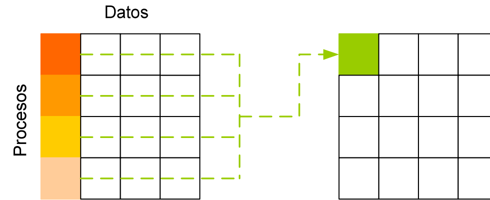

Figura 6. Operación de MPI_Reduce

5. Implementaciones de MPI

Se han creado varias implementaciones de MPI basadas en la publicación del estándar, muchas de ellas son de libre distribución y algunas tienen limitaciones de portabilidad de código. Algunas implementaciones de MPI se las puede encontrar en la página Web:
http://www-unix.mcs.anl.gov/mpi/implementations.html

LAM/MPI (*Local Area Multicomputer*), fue diseñada en el Centro de Supercomputadoras de la Universidad de Ohio. Puede ejecutarse sobre redes heterogéneas de equipos SUN, DEC, IBM, de estaciones de trabajo y computadoras personales. Se puede descargar una implementación gratuita de la página Web:
http://www.lam-mpi.org/

MPICH fue desarrollado a la par del estándar MPI. Se tienen varias implementaciones, y sus primeras versiones difieren de las más actuales, ya que fueron diseñadas para estaciones de trabajo y de computadoras personales, donde el desempeño de software estaba limitado por la funcionalidad de *sockets* de Unix. Se puede descargar una implementación gratuita de la página Web:
http://www-unix.mcs.anl.gov/mpi/mpich/download.html

Unify, provisto por la Universidad Estatal de Mississippi, en ésta se recopilan capas de software de MPI sobre una versión de PVM. Unify permite incluir llamadas de MPI y PVM dentro de un mismo programa.

6. Aplicación para resolución de Cadenas de Markov

Utilizando MPI se desarrolló una aplicación que permite resolver las Cadenas de Markov. Las Cadenas de Markov ([7], [8]) pueden hacer uso de matrices de grandes dimensiones, por lo que su solución y las operaciones asociadas pueden requerir una gran capacidad de procesamiento computacional. Las operaciones más relevantes utilizadas en la solución de Cadenas de Markov son:

- Potencia de matrices.
- Resolución de sistemas de ecuaciones lineales.

La aplicación desarrollada permite obtener la distribución al paso *n* y la distribución de régimen de Cadenas de Markov de Tiempo Discreto (CMTD) o la distribución al tiempo *t* y la distribución de régimen de Cadenas de Markov de Tiempo Continuo (CMTC).

En síntesis, la aplicación realiza las siguientes funciones:

1. El proceso maestro lee los datos de configuración y de la matriz **P** (CMTD) o **Q**(*t*) (CMTC).
2. El proceso maestro inicializa la librería de MPI.
3. El proceso maestro envía el paso al que se va a evaluar la CMTD o el tiempo al que se va a evaluar la CMTC y la dimensión de la matriz **P** o **Q**(*t*) a los otros procesos.
4. Se obtiene la distribución al paso *n* o al tiempo *t*. Para lo cual el proceso maestro calcula la porción de trabajo que le corresponde realizar a cada proceso y envía la porción correspondiente de la matriz (**P** o **Q**(*t*)) para que los otros procesos ayuden en las operaciones requeridas. Luego de realizar las operaciones, los procesos envían sus resultados al proceso maestro para que los presente en consola y los almacene en un archivo.
5. Se obtiene la distribución de régimen. Para lo cual el proceso maestro forma la matriz de conexión y el vector conocido, envía las porciones correspondientes a cada proceso para su resolución y luego recupera los resultados para presentarlos en consola y almacenarlos en un archivo.
6. Cada proceso libera la memoria utilizada y el proceso maestro se encarga además de liberar las librerías de MPI.

7. Resultados obtenidos

A continuación se presentan los resultados obtenidos al ejecutar la aplicación desarrollada usando un *cluster* construido con la herramienta *Rocks*. La aplicación fue ejecutada utilizando una sola computadora (1 procesador), y sobre el *cluster* usando 2 computadoras (2 procesadores) y 3 computadoras (3 procesadores).

Para probar la funcionalidad de la aplicación, se realizaron pruebas con matrices de orden 3, 30, 60, 90, 120, 150, 210, 270, 330, 390, 450, 510 y 600. Se realizaron pruebas resolviendo Cadenas de Markov a Tiempo Discreto y a Tiempo Continuo.

Para evaluar las Cadenas de Markov a Tiempo Discreto, se obtuvo la distribución en el paso 25 y la distribución de régimen. Se tomó el tiempo de ejecución de la aplicación, el tiempo que tardó en obtener la distribución en el paso 25 y el tiempo que tomó en calcular la distribución de régimen.

En la Figura 7 se realiza una comparación de los resultados obtenidos al calcular la distribución en el paso 25 con 1, 2 y 3 procesadores.

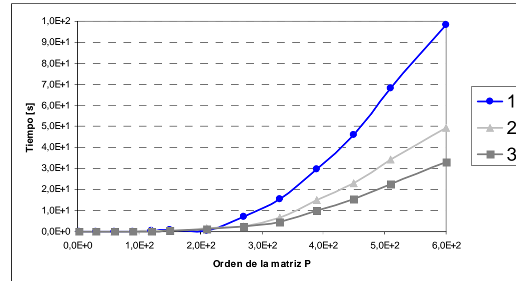

Figura 7. Comparación del tiempo requerido para obtener la distribución en el paso 25

En la Figura 8 se realiza una comparación de los resultados obtenidos al calcular la distribución de régimen con 1, 2 y 3 procesadores.

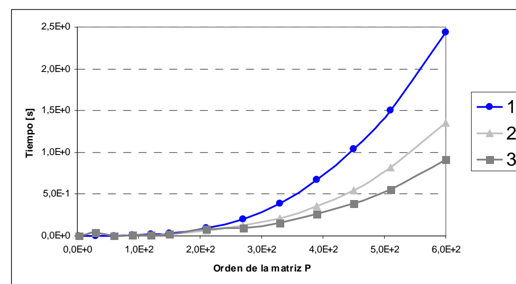

Figura 8. Comparación del tiempo requerido para obtener la distribución de régimen

En la Figura 10 se realiza una comparación de los tiempos de ejecución que tomó la resolución de la CMTD.

Para evaluar las Cadenas de Markov a Tiempo Continuo, se obtuvo la distribución en el tiempo 0,25 y la distribución de régimen. Se tomó el tiempo de ejecución de la aplicación,

el tiempo que tardó en obtener la distribución en el tiempo 0,25 y el tiempo que tomó en calcular la distribución de régimen.

Figura 9. Comparación del tiempo requerido para resolver CMTD

En la Figura 10 se realiza una comparación de los resultados obtenidos al calcular la distribución en el tiempo 0,25 con 1, 2 y 3 procesadores.

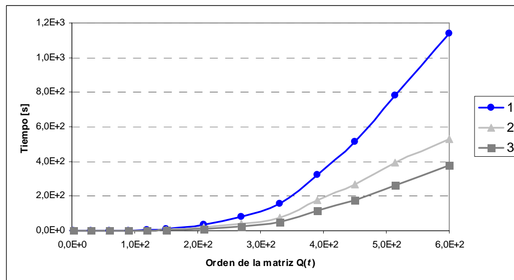

Figura 10. Comparación del tiempo requerido para obtener la distribución en el tiempo 0,25

En la Figura 11 se realiza una comparación de los resultados obtenidos al calcular la distribución de régimen con 1, 2 y 3 procesadores.

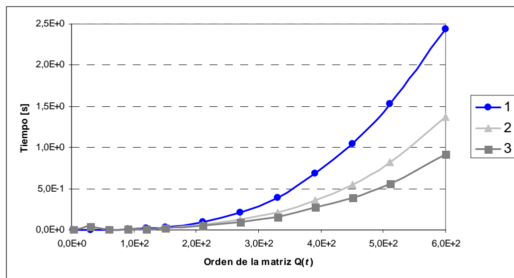

Figura 11. Comparación del tiempo requerido para obtener la distribución de régimen

En la Figura 12 se realiza una comparación de los tiempos de ejecución que tomó la resolución de la CMTD.

Se puede ver que en promedio, usando 2 procesadores, el tiempo que toma obtener la distribución en el paso 25 de Cadenas de Markov a Tiempo Discreto, es un valor cercano a la mitad del tiempo que toma el hacerlo con un sólo procesador; y usando 3 procesadores, el tiempo que toma se reduce a un valor cercano al tercio del valor que toma sobre uno sólo.

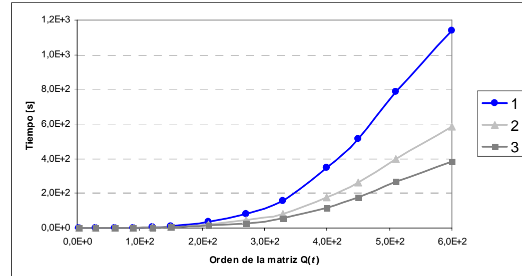

Figura 12. Comparación del tiempo requerido para resolver CMTC

También, algo similar a lo descrito en la obtención de la distribución en el paso 25, ocurre para la distribución en el tiempo 0,25 de Cadenas de Markov a Tiempo Continuo.

Además, se puede apreciar que en el caso de la distribución de régimen, tanto en CMTD como en CMTC, se tienen valores que no muestran una mejora al compararlos con respecto a los valores obtenidos en un procesador, para matrices de tamaño pequeño (de orden 3 a 150), debido a que la cantidad de operaciones es baja comparada con el tiempo requerido para el envío de datos entre procesos. Pero se puede apreciar una mejora en las matrices de mayor tamaño (orden superior a los 150), usando 2 procesadores el tiempo que se tarda en obtener la solución del sistema de ecuaciones se reduce en un 50% aproximadamente, y utilizando 3 procesadores el tiempo se reduce en promedio en un 60%.

Finalmente, se puede mencionar que el tiempo de ejecución total se comporta de forma similar a la descrita para la distribución en el paso 25 o en el tiempo 0,25. Se puede apreciar también, que el tiempo en leer la matriz **P** o **Q**(*t*), y almacenar los resultados, es decir las operaciones no paralelizadas, consumen poco tiempo comparado con las operaciones realizadas en paralelo.

En la Figura 14 se presenta el diagrama de Gantt generado mediante la herramienta de visualización Jumpshot-4 [9] que incluye *Rocks*. En esta figura se pueden ver las diferentes llamadas a MPI que realiza la aplicación para resolver la CMTD especificadas por la leyenda de la Figura 13.

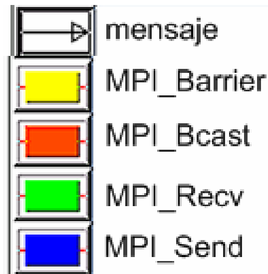

Figura 13. Leyenda utilizada por Jumpshot

8. Comentarios

Gracias al financiamiento de La Escuela Politécnica Nacional (EPN) y del FUNDACYT, en el Departamento de Electrónica, Telecomunicaciones y Redes de Información de la EPN se construyó un *cluster* básico, que se utilizó para ejecutar el programa desarrollado y obtener los resultados presentados. El *cluster* contaba originalmente con tres nodos con procesadores con tecnología Hyper Threading de 3 GHz, utilizando tecnología Gigabit Ethernet, así como discos duros SATA para las computadoras. El rendimiento del *cluster* es de aproximadamente 3,17 GFlops. Se espera que antes de finalizar el año se amplíe el *cluster* a siete nodos.

A futuro se espera desarrollar nuevas aplicaciones usando MPI para poder ejecutarlas sobre el *cluster* implementado buscando promover el desarrollo del procesamiento paralelo en beneficio del país.

9. Bibliografía

1. Quinn M, *Parallel Programming in C with MPI and OpenMP,* McGraw-Hill, 2003.
2. Pacheco P, *Parallel Programming with MPI*, Morgan Kaufmann, San Francisco, 1997.
3. Culler, D. y Singh J. *Parallel Computer Architectures: A hardware/Software Approach*, Morgan Kaufmann, San Francisco, 1999.
4. Lucke R, Building *Clustered Linux Systems*, Prentice Hall, Upper Saddle River, New Jersey, 2005.
5. http://www.rocksclusters.org
6. http://oscar.openclustergroup.org/
7. Norris, J. *Markov Chains.* Universidad de Cambridge, Inglaterra, 1998.
8. http://www.cms.wisc.edu/~cvg/course/491/modules/Markov/Markov/node2.html
9. Chan A. y Gropp W. *User's Guide for Jumpshot-4*, Laboratorio Nacional Argonne, Estados Unidos.

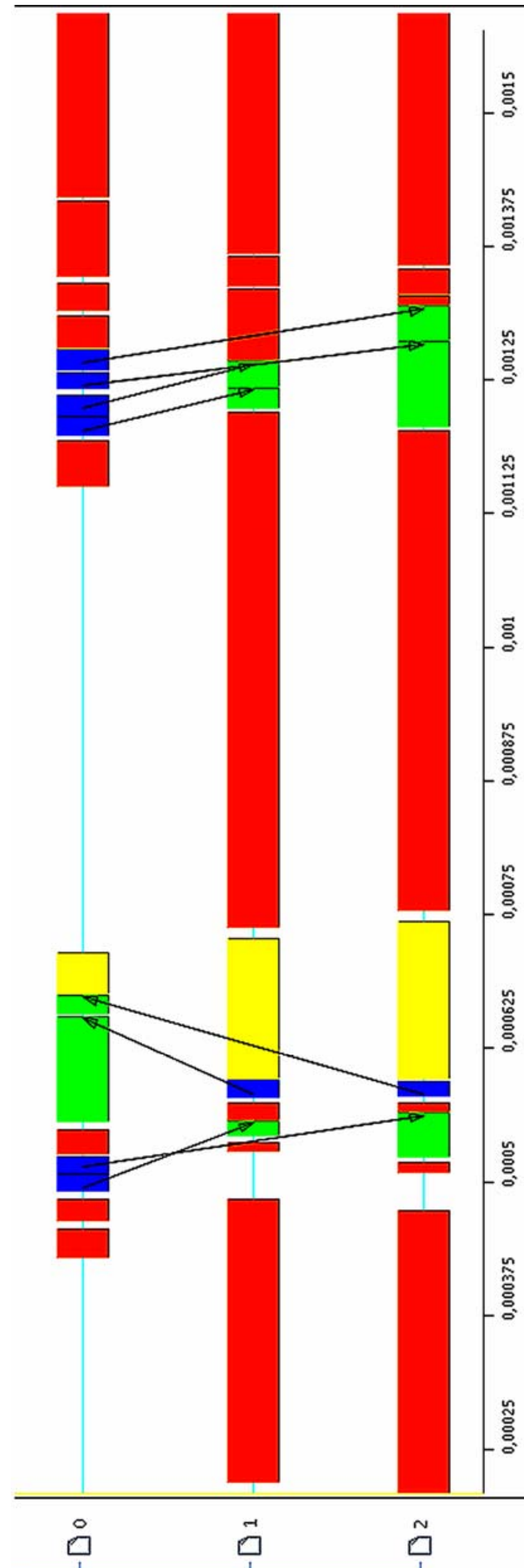

Figura 14. Diagrama de Gantt para CMTD

10. Biografías

Iván Bernal Carrillo

Ingeniero en Electrónica y Telecomunicaciones, Escuela Politécnica Nacional (EPN) en Quito-Ecuador en 1992. Obtuvo los títulos de M.Sc. (1997) y Ph.D. (2002) en *Computer Engineering* en *Syracuse University*, NY, USA. Actualmente es docente de la EPN, en el Departamento de Electrónica, Telecomunicaciones y Redes de Información.

David Mejía Navarrete

Nacido en Quito-Ecuador el 14 de enero de 1981. En el año de 1999 obtuvo su título de Bachiller en Ciencias con especialización Físico Matemáticas. En el año 2004 egresó de la carrera de Electrónica y Redes de Información de la Escuela Politécnica Nacional. En ese mismo año, cursó la certificación ACE *Advance Career* de IBM. Actualmente es instructor de la Academia Linux ACE de la Universidad Técnica Particular de Loja de la sede Quito.

Diego Fernández Ayala

Nacido en Quito, Ecuador (donde aún reside) el 21 de Marzo de 1980. Hijo de Eugenia Ayala y Luis Fernández. En el año de 1985, realizó sus estudios primarios en la Escuela Borja Nº 3 de los padres Cavannis. En el año de 1992 ingresó al Colegio Técnico Aeronáutico COTAC, en donde finalizó su educación media. En el año de 1998 obtuvo su título de Bachiller en “Humanidades Modernas” y en el año de 1999 ingresó como estudiante de pre-grado a la Escuela Politécnica Nacional. Dedicaba su tiempo a estudiar, y a pasarla con su familia y amigos, sin embargo en el transcurso de los años de estudio superior empezó a apasionarle las ciencias de la computación y decide seguir Ingeniería Electrónica con mención en Redes de la Información (su pasión es la programación con C y C++ y ciertas áreas relacionadas a las telecomunicaciones, es decir, las redes de datos). En el año de 2004, cursó la certificación ACE Advance Career de IBM. Actualmente es instructor de la Academia Linux ACE de la Universidad Técnica Particular de Loja de la sede Quito. Actualmente es administrador del área de sistemas de la Universidad Técnica Particular de Loja de la sede Quito. Entre sus pasatiempos, le gustaría hacer deporte, le gusta leer, y le gusta pintar, adora agitar el pincel sobre el lienzo y dar las formas que su imaginación precise.