UNIVERSIDAD MICHOACANA DE
SAN NICOLÁS DE HIDALGO

FACULTAD DE CIENCIAS FÍSICO MATEMÁTICAS
"Mat. Luis Manuel Rivera Gutiérrez"

PARALELIZACIÓN DE MÉTODOS DE APROXIMACIÓN DE FUNCIONES REALES USANDO UN CLÚSTER EN MPI

TESIS

PARA OBTENER EL TÍTULO DE:
LICENCIADO EN CIENCIAS FÍSICO MATEMÁTICAS.

PRESENTA:
LEONARDO GERVACIO ARCINIEGA.

ASESOR:
DR. MARIO CÉSAR SUÁREZ ARRIAGA.

MORELIA, DICIEMBRE 2009.

Resumen

El alto rendimiento, la alta disponibilidad, la disminución de costos y la alta confiabilidad son factores, que hoy en día son importantes en muchos de los sectores de las diferentes ramas de la computación. Dichos factores pueden influir en otro, como por ejemplo tiempo o recurso monetario. Algunos parámetros que dependen de estos factores son: aumentar la productividad al disminuir el tiempo de desarrollo y/o aumentar la seguridad. Para ello se debe contar con un equipo de cómputo que, además de ser bajo precio, sufrague las demandas de las características expuestas previamente. Un ejemplo donde es necesario este equipo es en la programación de métodos numéricos, donde tanto el tiempo de respuesta como el volumen de cálculos son puntos de interés muy importantes para el desempeño computacional.

MPI (Message Passing Interface, interfaz de paso de mensajes) es una librería estándar de paso de mensajes, la cual nos brinda una forma de explotar el rendimiento de un programa sobre varias computadoras trabajando simultáneamente.

En el presente trabajo se exhibe el paradigma del paralelismo, así como también los antecedentes históricos que han llevado a su desarrollo, los problemas que se pueden resolver con él, la forma de medición de la paralelización mediante la granularidad, la aceleración y la eficiencia.

Asimismo se definirá el concepto de clúster, los tipos de clústeres que hay y las consecuencias de cada una de sus implementaciones.

Se presentará la forma para implementar un clúster con OPENMPI, así como también se presentarán sus características y su desempeño en distintos ámbitos de la computación, sus ventajas y desventajas ante otras implementaciones. También se darán antecedentes históricos sobre implementaciones en la facultad de ciencia Físico–matemáticas de la Universidad Michoacana San Nicolás de Hidalgo.

De forma análoga se analizarán algunos métodos numéricos de aproximación, como la interpolación de Lagrange, y el ajuste de datos por mínimos cuadrados, para

determinar si cada uno de ellos puede ser paralelizable. Después se mostrará la forma de implementar en MPI los métodos numéricos paralelizables.

Por último se darán los resultados y las conclusiones que surgen de la implementación de estos métodos numéricos en un clúster con OPENMPI y MPICH, lo cual fue el objetivo general de esta tesis.

Dedicatoria

En memoria de mi abuelo,
quien me enseño a esforzarme
en todas mis labores difíciles de mi vida.

A mi padre y madre,
de quienes he tenido un gran
apoyo y cariño incondicional.

A mis hermanos,
con quienes he convivido gran
parte de experiencias de toda vida.

A todo ser humano,
que con su luna llena ha iluminado de luz mi vida.

Agradecimientos

Es difícil recordar a todas las personas que de alguna forma contribuyeron para que finalizara este trabajo. Algunas me ayudaron técnicamente, otras por otro lado me brindaron momentos de alegría, lo cual me motivo para seguir con muchas ganas en mis labores.

Primeramente quiero agradecer a mi familia; a mis padres, quienes con tanto esfuerzo sufragaron la mayoría de mis gastos académicos y por su cariño; a mis hermanos con quienes conviví una gran parte de mi vida; a mis tíos Domingo, Carmen, Elvira, Gregorio, León y mi abuelo Juan Gervacio (que en paz descanse) quienes en muchas ocasiones han influido mucho en mi vida.

Agradezco a mi asesor, Mario César Suárez, por sus enseñanzas y por su gran apoyo.

A Sofer, quien es admirable persona. Gracias por compartir su gran experiencia en lo que refiere a Clusteres.

Es imposible no estar agradecido con mis amigos. Particularmente a los de la facultad Alejandra García, Adriana, Erandi, Rafa, Rodrigo, Pastrana, los dos Dantes, Yuri, Gabriel, Omar, Irma, Juanito, Rosy, Lupita, Elideth por su valiosa amistad. Por otro parte a mis demás amigos: Alejandra González, Isabel González, Rosaura, Carla, Flor, Felisa, Azul, Mary Carmen, Pepe, Rigo, Joan, Carmen, Marbella por su gran confianza. Muy particularmente a Ma. Guadalupe Mejía Vilchis, Sandra, Brenda Karina y Mariana Cruz Gervacio.

A todos mis profesores y asesores: Azucena Chávez, Armando Sepúlveda, Gloria Andablo, Mary Carmen, Gabriel Arroyo, Mauricio, Juan de Dios, Mariano Hernández, Daniel Juan, Rita Zuazua, Eugenio Balanzario, Homero Díaz, Francisco Sidartha, Edgardo Morales, Félix Jiménez, Rafael Campos, Salvador Jara, Rigoberto , Gerardo Tinoco, Víctor Yepez, que me han empapado de sus conocimientos. De manera muy especial a Adán Cortes, Karina Figueroa, Héctor Tejeda, Antonio Camarena,

Ma. Luisa Pérez, Francisco Domínguez Mota, Edgar Chávez, Cristóbal Villegas, Luis Enrique Olivares, Abdón Choque Rivero.

Gracias a los años de mi vida, que me han dado la dicha de conocerlos a cada uno de ustedes y a los que he omitido.

Índice general

Capítulo 1

TEORIA DE SUPERCOMPUTACIÓN

Por procesamiento en paralelo se entiende la capacidad de utilizar varias partes independientes de un programa (elementos de proceso) para ejecutar dicha partes simultáneamente, en diferentes procesadores bajo un comando de control común.

La idea procesamiento en paralelo está basada en un viejo y conocido método divide y vencerás [1] usado para resolver problemas de carácter computacional.

Una analogía que ayuda para comprender las ventajas y los límites de este método es la siguiente: se ha asignado la tarea de ordenar los 200 libros de una biblioteca por nombre de autor. Una solución al problema es apilar los libros y que una sola persona los ordene. En la solución paralela añadiría una segunda persona, de tal forma que cada persona ordenara la mitad de libros, y así reducir el tiempo de ordenación. Esta ventaja que da la solución paralela es grande. ¿Qué sucedería si se añadieran más personas? . En principio el proceso de solución sería en menos tiempo si se añaden cada vez más personas. Pero sería muy absurdo poner 200 personas a ordenar los 200 libros de la biblioteca, ya que el hecho de solo poner en 200 pilas los libros para que cada persona ordene una pila seriá equivalente a que una sola persona los ordenara.

1.1. Visión histórica

En lo que refiere al mundo de la tecnología se observó que el procesamiento en paralelo podría resolver de manera más rápido ciertos problemas. Desde 1955 se ha

investigado sobre el campo del paralelismo, gente como Gene Amdahl [2] (quien en aquellos años trabajaba para IBM como principal diseñador del Modelo 704, primer ordenador comercial en tener unidad de punto flotante), investigaban sobre estas arquitecturas y trabajaban para que la relación costo–rendimiento aumentara. Empresas como IBM [3] (International Business Machines) y DEC [4] (Digital Equipment Corporation), se llevaban interesadas por la computación paralela desde las décadas 50–60.

En la década de los 80, la compartición de recursos mediante redes computacionales hizo un nuevo planteamiento para aprovechar ciclos de procesamiento. En los 70s personas como Bruce J. Nelson de Xerox expusieron nuevas ideas teóricas de cómo se podía explotar la capacidad de procesamiento paralelo mediante software [5].

En la década de los 90, el uso de computadoras aumentó exageradamente. El progreso de redes de computadoras hizo que dichas redes se volvieran atractivas por obtener alto rendimiento al realizar procesamiento en paralelo a bajo costo. Esto se dio gracias al bajo precio y a la gran disposición de componentes de software estándares para realizar implementaciones en ambientes de programación en paralelo.

Por otra parte, la historia local sobre programación en paralelo en la facultad de ciencias Físico–Matemáticas, de la Universidad Michoacana San Nicolás de Hidalgo, empezó cuando los investigadores en matemáticas aplicadas observaron que era necesario acudir una nueva área de investigación para hacer más eficiente el computo de sus implementaciones de algoritmos y métodos que desarrollaban, con un costo bajo y alto rendimiento. Por ello, se construyó un clúster [6] bajo el sistema operativo Fedora Core 1 con OpenMosix, con el objetivo de Simular numéricamente sistemas complejos, como por ejemplo la simulacion numérica de de reservorios multiporosos [7]; esta fue la primera vez que se construyó una computadora tipo Clúster en el área de matemáticas aplicadas en la Facultad de Ciencias y se inició la programación en paralelo como una nueva área de investigación, de cómputo práctico y de desarrollo con aplicaciones.

1.2. Marco teórico

Cabe señalar que el concepto de proceso toma una gran importancia al momento de hablar de programación paralela (al igual que para sistemas operativos), ya que mediante la repartición de tareas entre procesos es como se logra la gestión de las tareas; por lo cual se define un proceso como un programa en ejecución. Cada proceso tiene un espacio de direcciones, una lista de posiciones de memoria, que el proceso

puede leer o escribir.

Otro concepto importante es el de clúster de computadoras [8]. Se define clúster computacional como un englomeramiento de computadoras unidas por una red de comunicación trabajando conjuntamente. Según el tipo de servicio que pueden dar los clústers, estos se clasifican en clústers de alto rendimiento, de alta disponibilidad y de alta confiabilidad.

Para evaluar el rendimiento del un programa paralelo es necesario considerar los costos asociados a las ejecuciones de paralelización de las tareas. De este modo serán necesarios los siguientes parámetros de rendimiento en computación paralela.

1.2.1. Aceleración (speedup Sp)

La aceleración de un programa paralelo [9], es la relación entre el tiempo de ejecución de la versión paralela de un algoritmo, para una cantidad p de procesadores homogéneos, y el tiempo de ejecución de la versión serial del mismo algoritmo. Para poder calcularlo, necesitamos definir primero

- t_1: tiempo requerido para realizar la ejecución de un programa en 1 procesador.
- t_p: tiempo requerido para realizar la ejecución del mismo programa en p procesadores homogéneos.

Se define entonces como aceleración de un programa paralelo ejecutándose en p procesadores homogéneos a:

$$S_p = \frac{t_1}{t_p} \tag{1.1}$$

Esta medida sirve para comparar el rendimiento del algoritmo paralelo con respecto al serial. Cuando $S_p = p$, se dice que se cuenta con un programa de *aceleración lineal*.

La aceleración lineal es el óptimo desde el punto de vista de la paralelización, puesto que indica una proporción directa y sin desperdicios entre el tiempo de ejecución y la cantidad de procesadores que intervienen. Es así que $0 < S_p \leq p$.

1.2.2. Eficiencia

La eficiencia de un programa paralelo [9], como su nombre lo indica, expresa el aprovechamiento de los recursos de procesador, por parte de dicho programa. Se

define como:

$$E_p = \frac{S_p}{p} \tag{1.2}$$

de donde se deduce que, para el caso óptimo de aceleración lineal $E_p = 1$. Para el caso de la eficiencia, las cotas son: $0 < E_p \leq 1$

1.2.3. Granularidad

La granularidad [10] en el paralelismo se encuentra en paralelizar código de procesos, rutinas, módulos o bucles a nivel de instrucción. Así, granularidad se usa como el mínimo componente del sistema que puede ser preparado para ejecutarse de manera paralela. Dependiendo del grado de la granularidad del sistema se diferencia en:

1. Sistemas de Granularidad fina.

   - bucles.
   - sentencias.

   Estos sistemas generalmente son explotados por hardware muy caro con los nodos o procesadores fuertemente acoplados. Todo su funcionamiento se basa en propiedades del hardware.

2. Sistemas de granularidad media.

   - Módulos.
   - Rutinas.
   - tareas o procesos

   Dentro de estos sistemas se encuentran diversas librerías como PVM (Parallel Virtual Machine, Máquina Virtual Paralela) o MPI (Message Passing Interface, interfaz de paso de mensajes), y generalmente dichos sistemas son explotados por el programador o por el compilador.

3. sistemas de granularidad gruesa.

   - Trabajos y programas.
   - Módulos o procesos.

   En este tipo de granularidad el programador no necesita la utilización de algún tipo de librería externa, si no solo el conocimiento para paralelizar un algoritmo. Estos sistemas son débilmente acoplados.

1.3. Limites del rendimiento de un algoritmo en paralelo [11]

1.3.1. Ley de Amdahl

Para entender mejor esta ley es conveniente recuperar la analogía planteada para entender las ventajas y límites de la programación paralela, en el cual se llegaba a la conclusión de que era muy burdo asignar a 200 personas para ordenar 200 libros de una biblioteca. En cualquier programa en paralelo es imposible dejar que el código secuencial deje de existir, el cual no puede ser paralelizable, y por lo cual deber ser ejecutado por un solo procesador. Si la fracción del problema inherentemente secuencial es f, entonces el tiempo de ejecución de la parte secuencial de programa es $f * t_1$ y el tiempo en ejecutar la parte paralelizable en p procesadores es $\frac{(1-f)*t_1}{p}$, y luego que el tiempo de ejecución del programa paralelizado será

$$t_p = f * t_1 + \frac{(1-f) * t_1}{p} \tag{1.3}$$

y por lo tanto, la aceleración será

$$S_p = \frac{t_1}{t_p} = \frac{1}{f + \frac{(1-f)}{p}}. \tag{1.4}$$

De lo cual se observa, que aunque se contara con una infinidad de procesadores, la aceleración estará limitada por $\frac{1}{f}$

$$\lim_{p\to\infty} S_p = \lim_{p\to\infty} \left[\frac{1}{f + \frac{(1-f)}{p}} \right] = \frac{1}{f} \tag{1.5}$$

De manera análoga, la eficiencia de un programa se acercaría a cero a medida que se incorporan cerca de una infinidad procesadores.

$$\lim_{p\to\infty} E_p = \lim_{p\to\infty} \left[\frac{S_p}{p} \right] = 0 \tag{1.6}$$

Esta ley fue descrita por Gene Amdahl [2] en 1967 [12]. A pesar de que no toma en cuenta las características en concreto, una de las implicaciones que tiene esta ecuación es: cuanto mejor paralelizado esté un programa más susceptible será de aumentar su aceleración y por lo tanto explotar el rendimiento del sistema en que lo ejecute.

1.4. Modelos para la programación paralela

Un modelo de programación [10] es una abstracción de la programación que se le brinda al programador de una forma simplificada y transparente de la arquitectura. De esta forma, los modelos de programación son mecanismos disponibles al programador para escribir la estructura lógica de un programa.

Para hacer una buena implementación de un programa y obtener un buen rendimiento sobre arquitecturas paralelas es necesario determinar un modelo de programación adecuado.

En este apartado describiremos los modelos de programación más usados. Entre estos modelos están los modelos de memoria compartida, los modelos de memoria distribuida y los modelos híbridos. En cada uno, se especificaran las ventajas, desventajas y sus características que definen donde se deben de implementar éstos modelos de programación.

1.4.1. Introducción

A principios de los 80's se creía que la única forma de mejorar el rendimiento de una aplicación era sólo creando procesadores más potentes. Sin embargo, a principios de los 90's, con el surgimiento de redes de computadoras, el procesamiento en paralelo desafío esta idea al unir dos o más procesadores para resolver el problema.

Por otro lado, los problemas paralelos surgen de lo económico y científico, y por lo cual tienen mucho impacto en el mundo de muchas aplicaciones, por ejemplo en el procesamiento de imagenes [13] o en la exploración de grafos [14]. Sin embargo, el desarrollo de aplicaciones paralelas es complejo, ya que dependen de la disponibilidad de herraminetas de software y del entorno [15].

De esta manera, la programación resulta exitosa, si el software paralelo es capaz de ocultar detalles de la arquitectura, la red de comunicación y de la tolerancia a fallas.

Existen dos tipos de modelos de programación paralela [16], los cuales son:

1. Paralelismo implícito: en este modelo, quizá el más atractivo, es característico de algunos lenguajes y compiladores paralelizadores. Todo su funcionamiento da lugar a que el programador sepa lo mínimo del sistema para paralelizar.

   Entre los lenguajes y compiladores de programación implícita se encuentran los

siguientes: Higher Order Functional Hasbell, Maude, OBJ, Unity, PPP, REDUCE OR, Opera, Palm, P3L, Cole, Darlington, Celland, Carpet, CDL, Ceprol, Cristal, Sisal, ld, Concurrent Prolog, Delta Prolog, Strand, Multilisp, pSETL, Gamma, PEI, APL, MOA, Nial, AT, CamlFlight, NESL, MOSIX, OPENMOSIX [9] [17], OPENSSI [18] [19].

2. Paralelismo explícito: En este modelo el programador es responsable de la descomposición de las tareas, así como también de la comunicación entre ellas y su mapeo.

   Este tipo de modelo supone que el programador explotara el paralelismo de una aplicación en particular. Dicho modelo se caracteriza por las librerías para el manejo de paralelismo. Entre estas librerías se mencionan: Express p4, RPC (Remote Procedure Calls), PVM, MPI, OPENMP [21], threads (POSIX threads, SOLARIS threads, Win32 threads, entre otras).

1.4.2. Modelos de programación Paralela

Existen diversos modelos de programación que se utilizan para desarrollar aplicaciones paralelas. Estos modelos se clasifican en:

1. Modelos de memoria compartida.
2. Modelos de memoria distribuida.
3. Modelos híbridos.

A continuación de determinara en qué casos es más adecuada la utilización de cada uno de ellos.

Modelos de memoria compartida

En los modelos de memoria compartida todos los procesos tienen el espacio de memoria en común.

Una ventaja de esto es que no se debe de hacerse explícitamente la comunicación entre procesos, por lo que el desarrollo de la programación se simplifica.

Por otra parte, como todos los procesos tienen acceso a la misma memoria, y si en un dado caso uno de los procesos corrompe el contenido de dicha memoria, entonces

los resultados de los demás procesos terminarían por ser también afectados. Por eso, para tener acceso a tal, se hace uso de mecanismos como semáforos y monitores [20, Cap. 2].

Entre los modelos de memoria compartida destacan principalmente:

1. Los modelos de programación multi–threaded con POSIX Threads (Pthreads).

2. Modelos de programación OPENMP [21].

Un thread (hilo) es un semi–proceso que ejecuta un conjunto de instrucciones de manera independiente.

Un proceso puede descomponerse en varios threads que comparten todos los recursos del proceso, y por esto el programador es responsable de sincronizar los threads para garantizar la exclusividad de operaciones y el acceso simultaneo a los datos.

Modelos de memoria distribuida.

En los inicios de la programación paralela con memoria compartida, algunos investigadores notaron que si a cada procesador se le asigna una localidad de memoria diferente para datos, de modo que minimizara la comunicación entre procesadores, entonces se obtenía alto rendimiento. De esta forma surge la idea de dividir el dominio de los datos que se asignaran a diferentes procesos, cada uno ejecutado en un procesador diferente.

Los modelos de memoria distribuida más destacados son los modelos de paso de mensajes, los cuales se decriben enseguida.

Modelos de paso de mensajes.

Debido a que la comunicación entre procesos en estos modelos no es tan fácil, entonces las librerías para el paso de mensajes toman una importancia muy grande entre dichos modelos. Para que un programador pueda utilizar librerías de paso de mensaje, es necesario que tenga una versión del programa en todos los procesadores.

A continuación se comentan las características uno de los principales estándares para el paso de mensajes: PVM. De MPI que es otro de ellos se describirá con más detalle en el siguiente capítulo.

PVM

PVM [22] ("Parallel Virtual Machine", Máquina Virtual Paralela) es una librería de paso de mensajes libre y portable, que puede ser usada tanto con procesadores homogéneos como en procesadores heterogéneos.

El usuario escribe su aplicación como un conjunto de tareas que cooperan entre sí. Dichas tareas acceden a los recursos de PVM a través de una librería de funciones con una interfaz estándar. Estas funciones permiten la inicialización y finalización de las tareas a través de la red, así como la comunicación y la sincronización entre dichas tareas. Las operaciones de comunicación utilizan estructuras predefinidas, tanto las encargadas del envío y recepción de datos como aquellas más complejas.

Debido a que PVM ha sido una distribución libre desde hace tiempo, muchos lenguajes de programación, librerías de aplicaciones, herramientas de programación y depuración de errores han utilizado PVM como su librería portable de paso de mensajes.

Modelos híbridos

Estos tipos de modelos se componen de dos o más modelos de programación paralela anteriormente expuestos. Uno de los ejemplos bastante común de modelos híbridos es resultado de la implementación combinada de OpenMP con algún modelo de paso de mensajes [23] [24].

Para nuestro caso, se hizo uso del paralelismo explícito, con memoria distribuida bajo el modelo de paso de mensajes en la interfaz MPI.

Capítulo 2

INTERFAZ DE PASO DE MENSAJES MPI

MPI [25] ("Message Passing Interface", Interfaz de Paso de Mensajes) como se mencionó previamente, es un estándar que define funciones contenidas en una librería para el paso de mensajes diseñada para la explotación de múltiples procesadores. Dicha librería puede ser utilizada por varios usuarios para la implementación de programas que utilicen paso de mensajes.

Desde su primera versión en junio de 1994, ha sido ampliamente aceptado y usado [27]. Existen varias implementaciones muy estables, eficientes y disponibles para domino público. Gracias a ello MPI ha alcanzado uno de sus principales objetivos: darle credibilidad al procesamiento en paralelo. Hoy en día muchos investigadores y compañías tienen una forma sencilla y segura de desarrollar programas en paralelo portables de paso de mensajes.

2.1. Origen

Los diseñadores de MPI pretendieron incorporar la mayoría de las características más atractivas de los sistemas existentes de paso de mensajes, antes que seleccionar uno de ellos y adoptarlo como el estándar. Así, en su origen MPI estuvo muy influenciado por el trabajo del Centro de Investigación Thomas John Watson de IBM [26], Vertex de nCUBE [28] y PARMACS [29]. Otras contribuciones importantes fueron las realizadas por PVM, Chameleon [30] y PICL (Portable instrumented Communication Library). Los diseñadores de MPI identificaron algunos defectos críticos de los

sistemas de paso de mensajes existentes en áreas como la composición de datos complejos, la modularidad y las comunicaciones seguras. Esto permitió la introducción de nuevas características en MPI.

2.2. Historia

El esfuerzo de estandarización de MPI involucró a unas 60 personas procedentes de 40 organizaciones, principalmente de Estados Unidos y Europa. La mayoría de las principales compañías de computadores paralelos del momento estuvieron involucradas en MPI, así como investigadores provenientes de universidades, laboratorios de gobierno y la industria. El proceso de estandarización comenzó con el Seminario sobre Estándares de Paso de Mensajes en Entornos de Memoria Distribuida, patrocinado por el Centro de Investigaciones sobre Procesamiento Paralelo, mantenido durante los días 29–30 de Abril de 1992 en Williamburg, Virginia (EEUU). En este seminario se discutieron las características esenciales que debía tener una interfaz estándar de paso de mensajes y se estableció un grupo de trabajo para continuar con el proceso de estandarización.

Un borrador preliminar, conocido como MPI1, fue propuesto por Dongarra, Hempel, Hey y Walker en noviembre de 1992 y una versión revisada fue completada en Febrero de 1993 [31]. MPI1 abarcaba las características principales que fueron identificadas en el seminario de Williamburg como necesarias en un estándar de paso de mensajes. Dado que fue generado para promover la discusión, este primer borrador se centró principalmente en las comunicaciones punto–a–punto. Aunque tocaba muchos asuntos referentes a la estandarización, no incluía operaciones colectivas ni tenía en cuenta la utilización de hilos.

En noviembre de 1992 una reunión del grupo de trabajo de MPI tuvo lugar en Minneapolis, en la cual se decidió hacer el proceso de estandarización de una manera más formal, adoptando la organización y los procedimientos del “High Performance Fortran Forum” [32] (Foro Fortran de Alto Rendimiento). Fueron formados subcomités para cada una de las principales áreas que componen el estándar, y se estableció un foro de discusión vía e–mail para cada una de ellas. De esta manera quedó conformado el Foro MPI [33], a cuyo cargo quedaría el proceso de estandarización de MPI

En dicha reunión se propuso el objetivo de presentar un borrador del estándar MPI en el otoño de 1993. Para conseguirlo el Foro MPI mantuvo reuniones cada 6 semanas durante los primeros 9 meses de 1993, presentando el borrador del estándar MPI en la conferencia Supercomputing'93 en noviembre de 1993 [34]. Después de

un período de comentarios públicos, que resultaron en algunos cambios en MPI, la versión 1.0 de MPI fue presentada en junio de 1994.

A principios de marzo de 1995 el Foro MPI empezó a reunirse de nuevo para considerar correcciones y extensiones al documento original del estándar MPI. El primer producto fruto de estas deliberaciones fue la versión 1.1 de MPI, presentada en junio de 1995.

En julio de 1997 aparecen al mismo tiempo la versión 1.2 de MPI y la especificación MPI–2. La versión 1.2 de MPI contiene correcciones y clarificaciones referentes a la versión 1.1. Sin embargo MPI–2 añade nuevos elementos al estándar MPI–1, definiendo la versión 2.0 de MPI. Todavía hoy continúan las discusiones acerca de las áreas hacia las cuales podría ser útil expandir el estándar MPI; sin embargo en muchas de ellas es necesario obtener más experiencia y realizar más discusiones. De todo ello se encarga un documento separado, el MPI "Journal Of Development" [35] (Periódico de Desarrollo), que no forma parte de la especificación MPI–2. Y por supuesto el Foro MPI sigue abierto a las aportaciones que puedan realizar los usuarios del estándar, con el objetivo de mantenerlo y desarrollarlo.

2.3. Objetivos

Portabilidad

El principal objetivo de MPI, es conseguir un alto grado de portabilidad entre diferentes máquinas. El mismo código fuente de paso de mensajes debería poder ser ejecutado en una variedad de máquinas tan grande como la soportada por las distintas implementaciones de MPI. A pesar de que el paso de mensajes muchas veces es considerado algo propio de los sistemas paralelos de memoria distribuida, el mismo código podría ser ejecutado en un sistema paralelo de memoria compartida. Puede ejecutarse en clústers o incluso en un conjunto de procesos ejecutándose en una misma máquina. El hecho de saber que existen implementaciones de MPI eficientes para una amplia variedad de sistemas nos da cierto grado de flexibilidad en el desarrollo del código, la búsqueda de errores y la elección de la plataforma que finalmente utilizaremos.

Heterogeneidad

MPI tiene la capacidad para ejecutarse en sistemas heterogéneos de manera transparente, es decir, MPI se puede ejecutar en equipos con diferente arquitectura de manera que el programador no tenga que hacer configuraciones diferentes, ni programar distinto para equipos con arquitectura diferente. Así pues, una implementación MPI debe ser capaz de extender una colección de procesos sobre un conjunto de sistemas con arquitecturas diferentes, de manera que proporcione un modelo de computador virtual que oculte las diferencias en las arquitecturas. De este modo el usuario no se tiene que preocupar de si el código está enviando mensajes entre procesadores de la misma o distinta arquitectura. La implementación MPI hará automáticamente cualquier conversión que sea necesaria y utilizará el protocolo de comunicación adecuado. Sin embargo MPI no prohíbe implementaciones destinadas a un único y homogéneo sistema. En definitiva, los usuarios que quieran ejecutar sus programas en sistemas heterogéneos deberán utilizar implementaciones MPI diseñadas para soportar heterogeneidad.

Rendimiento

Un punto crucial es que MPI fue cuidadosamente diseñado de manera que permite implementaciones eficientes. Las elecciones en el diseño parecen haber sido hechas correctamente, dado que las implementaciones MPI están alcanzando un alto rendimiento en una amplia variedad de plataformas; de hecho el rendimiento alcanzado en dichas implementaciones es comparable al de los sistemas presentados por las compañías, los cuales están diseñados para arquitecturas específicas y tienen menor capacidad de portabilidad. Un objetivo importante de diseño en MPI fue el de permitir implementaciones eficientes para máquinas de diferentes características.

Las implementaciones pueden beneficiarse de las ventajas que ofrecen los subsistemas de comunicación de varias máquinas a través de sus características específicas. En máquinas con coprocesadores de comunicación inteligentes podemos cargar sobre dichos coprocesadores la mayoría del procesamiento relativo al protocolo de paso de mensajes. En otros sistemas la mayoría del código de comunicación será ejecutada por el procesador principal.

Otro ejemplo es el uso de objetos opacos en MPI; por ejemplo los elementos grupo y comunicador son objetos opacos. Desde un punto de vista práctico esto significa que los detalles de su representación interna dependen de la implementación MPI particular, y como consecuencia el usuario no puede acceder directamente a ellos.

En vez de ello el usuario accede a un manejador que referencia al objeto opaco, de manera que dichos objetos opacos son manipulados por funciones MPI especiales. De esta manera cada implementación es libre de hacer lo mejor en determinadas circunstancias.

MPI ha sido diseñado para reducir la sobrecarga en la comunicación producida por el procesamiento, utilizando agentes de comunicación inteligentes y ocultando latencias en la comunicación. Esto se ha logrado usando llamadas no bloqueantes, que separan el inicio de la comunicación de su final.

Escalabilidad

MPI puede contener una cantidad variable de recursos, es decir, MPI permite o soporta la escalabilidad a través de algunas de sus características de diseño. Por ejemplo, una aplicación puede crear subgrupos de procesos que, en turnos, puedan ejecutar operaciones de comunicación colectiva para limitar el número de procesos involucrados.

2.4. Implementaciones [36]

Aunque MPI tienen la ventaja de operar en clústers heterogéneos, MPI no permite interoperabilidad entre diferentes implementaciones.

En los siguientes apartados se describen algunas de las implementaciones de MPI de dominio público.

2.4.1. MPICH (MPI/CHAMELEON)

Es la implementación más importante de MPI. La primera versión de MPICH fue escrita durante el proceso de estandarización de MPI, por el Laboratorio Nacional de Argonne y la Universidad del Estado de Mississipi [37].

2.4.2. LAM

LAM [38] [39], acrónimo de Local Area Multicomputer es un ambiente programación y sistema de desarrollo sobre redes de computadoras heterogéneas. Cuenta con

herramientas de depuración y monitorización. Esta implementación es del Centro de Supercomputación de Ohio.

2.4.3. CHIMP

CHIMP (Common High–level Interface to Message Parssing) fue Desarrollado por el Centro de Computación Paralela de Edimburgo [40].

2.4.4. UNIFY

Provee de una parte de MPI dentro del ambiente de PVM para obtener ventaja de las llamadas ya disponibles en PVM. Fue desarrollado en la Universidad del estado de Mississippi [41].

2.4.5. MPICH/NT

Esta implementación es para redes de estaciones con Windows NT. Basada en MPICH y disponible también en la Universidad del estado de Mississippi [42].

2.4.6. OPENMPI

OPENMPI [43] tiene como objetivo la construcción de la siguiente generación de interfaz de paso de mensajes. OPENMPI es una obra colaborada por el Laboratorio Nacional de Los Álamos, el Laboratorio de Sistemas abiertos de la Universidad de Indiana, el Laboratorio de Innovación de Computación de la Universidad de Tennesse y el Centro de Computación de Alto Rendimiento de la Universidad de Stuttgart. Esté proyecto está basado en LAM/MPI, PAX–MPI y FT–MPI. Sin embargo, OPENMPI es un nuevo MPI.

OPENMPI se basa en un componente de arquitectura modular. Dicha arquitectura soporta ejecuciones de componentes las cuales son optimizadas por un envoltorio específico.

La implementación de comunicación punto a punto (p2p) en OPENMPI se basa en un armazón Múltiple MCA. La fig. 2.1 ilustra la arquitectura de este armazón.

Como se observa en la fig. 2.1 la arquitectura consiste de 4 capas. Esas capas son

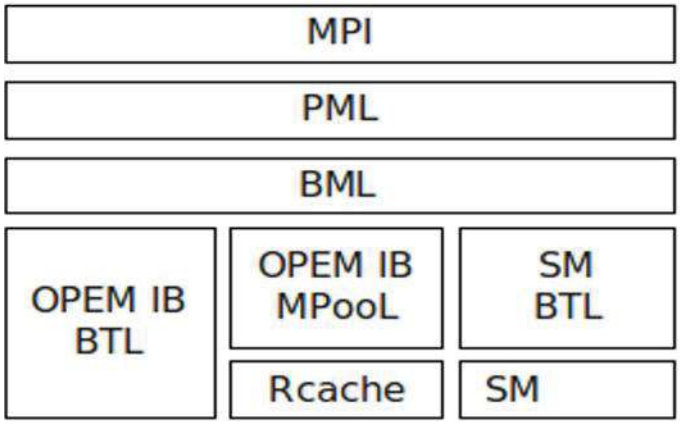

Figura 2.1: Arquitectura OPENMPI p2p.

la capa de transferencia (Byte Transfer Layer), la capa de administración, la capa de comunicación punto a punto y la capa de MPI.

En seguida se mostrara como hacer la instalación de un clúster en MPI bajo la implementación OpenMPI.

Instalación de OPENMPI [44]

Se instalarán en todos los equipos la distribución de Debian Denny v.5.0.1 utilizando los dos primeros Dvd's de instalación, que se pueden obtener de la pagina de descargas para LINUX [45], o directamente de la página oficial de Debian [46].

La instalación se realizará tomando las opciones que aparecen por defecto en el asistente, hasta llegar al punto de los programas a instalar donde solo seleccionamos **sistema estándar**, lo cual indica que solo se instalara el mínimo de programas necesarios para que arranque nuestro sistema operativo.

Una vez realizada la instalación y reiniciado el equipo, entramos al equipo como usuario (login) **root** con contraseña (password) de superusuario e introducimos los paquetes del segundo Dvd de la siguiente manera:

apt–cdrom add

Con lo cual nos pedirá el Dvd–2, lo introducimos y esperamos hasta que termine de leer los índices de los paquetes en dicho Dvd.

Configuración del Nodo Maestro

Lo primero que se debe hacer es configurar una carpeta *compartido* accesible desde todos los nodos de la red, donde se ubicara el programa, ya que como se mencionó en anteriormente los modelos por pasos de mensajes deben tener una versión del programa en cada nodo. También en esta carpeta se pondrán archivos necesarios para la instalación.

Para ello se utilizará NFS [47, Cap. 7] (Network File System, Sistema de Archivos de Red), protocolo que permite acceder a ficheros remotos dentro de una red. Como este paquete está incluido en la distribución Debian Denny v.5.0.1, se puede instalar cómodamente utilizando la herramienta de instalación de paquetes.

apt–get install nfs–kernel–sever

Ahora crear la carpeta que se va a compartir:

mkdir /mnt/compartido

Luego configurar NFS para que permita el acceso remoto desde otros equipos en la red. Editar el archivo */etc/exports* (*pico etc/export*s donde pico es un editor de texto) añadiendo una línea por cada equipo que va acceder al recurso definida de la siguiente forma:

mnt/compartido dirección_ip_del_nodo_que_accedera_al_recurso(rw,sync)

Para obtener la dirección IP de algún nodo, entrar al nodo como root y ejecutar el comando *ifconfig eth0*.

Por ejemplo, si se tienen tres nodos con IP 's 192.168.2.232, 192.168.2.178 y 192.168.2.180 entonces el archivo */etc/exports* del nodo maestro quedaría de la siguiente manera:

/mnt/compartido 192.168.2.232(rw,sync)

/mnt/compartido 192.168.2.178(rw,sync)

/mnt/compartido 192.168.2.180(rw,sync)

Actualizar la información de este archivo para NFS.

exportfs –ra

Ahora modificar el fichero */etc/hosts.allow* para permitir el acceso de los nodos al recurso compartido, incluyendo una línea por cada nodo con el siguiente formato:

portmap: dirección_ipnodo

Para el ejemplo previo el fichero */etc/hosts.allow* del nodo maestro queda de la siguiente manera:

portmap: 192.168.2.232

portmap: 192.168.2.178

portmap: 192.168.2.180

Editar el fichero */etc/hosts.deny* y añadimos la siguiente línea para que ningún equipo, excepto los especificados en el fichero *hosts.allow*, puedan acceder a este recurso.

portmap: ALL

Falta añadir al fichero */etc/hosts* los nodos esclavos según la siguiente nomenclatura:

dirección_ip_del_nodo nombre_del_equipo

por ejemplo si los nombres de los equipos son nodo1, nodo2 y nodo3 , y las direcciones IP son con IP's 192.168.2.232, 192.168.2.178 y 192.168.2.180, respectivamente, entonces se debe de añadir al fichero */etc/hosts* las siguientes líneas:

192.168.2.232 nodo1

192.168.2.178 nodo2

192.168.2.180 nodo3

El siguiente paso es establecer la forma de comunicación entre el nodo maestro y los esclavos. Para hacerlo se puede utilizar rsh (Remote Shell), ssh [48] (Secure Shell), entre otras opciones. En este caso se usara el protocolo ssh, que utiliza un par de claves pública y privada, basado en algoritmos de cifrado de criptografía asimétrica, y que proporciona medidas de seguridad superiores a rsh.

Lo primero es instalar el paquete *Openssh*:

apt–get install openssh–server

Desde el directorio home del usuario que vaya a tener permisos para ejecutar aplicaciones con *OpenMPI*, generar sus claves públicas y privadas.

cd $HOME

ssh–keygen –t dsa

Aparecerán una serie de preguntas a las que se contestaran con el valor por defecto. Este comando generara una carpeta llamada *.ssh*, donde se haya las claves públicas

y privadas generales.

Copiar la clave pública al recurso compartido para que esté accesible a los nodos.

cp .ssh/id_dsa.pub /mnt/compartido/root_maestro.pub

Se ha llegado al punto donde sólo queda instalar y configurar OPENMPI, para lo cual es necesario instalar los paquetes necesarios para la compilación.

apt–get install build–essential gfortran gcc

Instalar OPEMPI

apt–get install libopenmpi–dev openmpi–bin

En el archivo */etc/openmpi/openmpi_default_hostfile* se especifican las maquinas que van a participar en el clúster.

Para el ejemplo previo, suponiendo que el nombre del nodo maestro es master, el archivo */etc/openmpi/openmpi_default_hostfile* contendría lo siguiente:

master

nodo1

nodo2

nodo3

Configuración del nodo Esclavo

Se comienza la configuración permitiendo el acceso del nodo maestro a los esclavos, para lo que se modifica el archivo */etc/hosts.allow*, al cual se le añade la siguiente línea:

ALL:dirección_IP_del_nodo_maestro

A continuación se cambia el archivo */etc/hosts* añadiéndole la línea:

dirección_IP_del_nodo_maestro nombre_del_equipo_del_nodo_maestro

Posteriormente se debe configurar el acceso al recurso compartido que se creó en el nodo maestro, verificando que exista y que esté bien definido con:

showmount –e nombre_del_equipo_del_nodo_maestro

que mostrará las carpetas compartidas en el equipo especificado y los nodos que tienen acceso al recurso.

Ahora se crea la carpeta donde montaremos los archivos compartidos:

mkdir /mnt/compartido

Para poder montar el fichero, se declara en el fichero */etc/fstab* una línea, en la cual se debe definir el equipo donde se encuentra y el nombre del recurso compartido, el lugar donde se va a montar, el sistema de ficheros (NFS) y las operaciones que se pueden realizar sobre él. Entonces la declaración es como sigue:

IP_del_nodo_maestro:/mnt/compartido /mnt/compartido/ nfs rw 0 0

Después se monta el fichero.

mount /mnt/compartido

Seguidamente se establece las formas de comunicación a través de ssh, al igual que se hizo en el nodo maestro, instalando el paquete openssh, y creando las claves públicas y privadas. Luego, se crea (si no existe) el archivo *authorized_keys* en el directorio *.ssh/* que se genero en el directorio *HOME*.

cd $HOME

cd .ssh/

echo "> authorized_keys

Como se copio la clave pública del nodo maestro en el archivo compartido y ya es accesible, se copia dicha clave al archivo *authorized_keys* para permitir la conexión desde el nodo maestro al nodo esclavo. Se puede copiar al ejecutar la siguiente instrucción:

cat /mnt/compartido/root_maestro.pub >> $HOME/.ssh/authorized_keys

Por último se instala OPENMPI de la misma manera que se hizo en el nodo maestro.

2.5. Elementos básicos de la programación en MPI

Como se ha descrito ante descrito anteriormente, MPI es una librería de rutinas y funciones, la cual está disponible en los lenguajes C, C++ [49] y Fortran77 [50]; en este caso se hará uso de los lenguajes C y Fortran77. Para el desarrollo de programas en paralelo con MPI se debe conocer dichas rutinas (en fortran) o funciones (en C y C++), las cuales tienen una semántica independiente del lenguaje. Todas las rutinas y funciones comienzan con el prefijo MPI_. En funciones en C, luego del prefijo se

coloca el nombre de la función con la primera letra en mayúscula. En tanto en fortran el nombre de la rutina va todo con mayúsculas. Casi todas las rutinas de fortran tienen el parámetro adicional ierr. Finalmente, todas las funciones o rutinas que no son de MPI se ejecutan de manera local.

2.5.1. Compilación

Para un programa con MPI, se tienen los comandos *mpicc* y *mpif77* para compilar en C y en fortran respectivamente, y cuyo funcionamiento es similar al de los compiladores de *cc* y *f77*.

La semántica para compilar un programa con *mpicc* es la siguiente:

mpicc –o nombre_del_archivo_de_salida nombre_del_archivo_de_dentrada.c

y para *f77* tenemos:

mpif77 –o nombre_del_archivo_de_salida nombre_del_archivo_de_dentrada.c

Cabe señalar que también es posible poner más opciones, como por ejemplo –lm en C, que indica que se esta haciendo uso de la libreria $< math.h >$.

2.5.2. Ejecutando programas en MPI

Ahora es momento de aprender a ejecutar programas en MPI. En MPI no se especifica cómo se arrancan los procesos. Para dicha tarea, cada implementación se encarga de definir los mecanismos para el arranque de procesos; por ello algunas de ellas coinciden en el uso de un programa especial denominado *mpirun*, que no es parte del estándar de MPI.

La opción *–t* muestra todos los comandos disponibles para ejecución en *mpirun*.

Una forma de ejecutar un programa con *mpirun* es la siguiente:

mpirun –np 2 nombre_del_programa

donde el parámetro *np* indica a *mpirun* que debe crear dos instancias del programa, las cuales debe ejecutar en nodos distintos si se dispone de ellos, si no se disponen entonces se ejecutan sobre el mismo.

Si en la implementación no se está seguro en que maquinas serán corridos los procesos (en este caso se especifico a la hora de configurar OpenMPI y editar el archivo */etc/openmpi/openmpi_default_hostfile*), entonces se puede correr el programa con:

mpirun –np 2 –machinefile maquinas nombre_del_programa

donde *machinefile* indica a *mpirun* que en el archivo *maquinas* están los nombres de las maquinas donde se deben de correr los procesos y el parámetro *np* indica a *mpirun* que debe crear dos instancias del programa, que deberan de ser ejacutadas en las maquinas indicadas por *machinefile* en el archivo *maquinas*.

La primera función que se debe de conocer de la librería MPI es *MPI_Init*, esta indica la inicialización de MPI, y le indica a la implementación si requiere alguna configuración previa antes de ejecutar llamadas a funciones o rutinas MPI.

Paralelamente existe una función, *MPI_Finalize*, que permite purgar o eliminar todos los estados de MPI. Una vez llamada esta función ya no será posible hacer una llamada a alguna otra función de MPI.

El código siguiente corresponde a un programa sencillo en el lenguajes C con MPI, en donde cada proceso imprime el mensaje: "Hola mundo":

```
#include<stdio.h>
#include<mpi.h>
int main(int argc, char **argv )
{
   MPI_Init(&argc, &argv);
   printf("Hola Mundo\n");
   MPI_Finalize();
}
```

Este código se puede escribir en un archivo, llamado por ejemplo *hola_mundo.c*, compilado y ejecutado como sigue:

mpicc –o hola hola_mundo.c

mpirun –np 3 hola

El código en fortran que se muestra enseguida hace la misma acción:

```
 program main
 include ' mpif.h'
 integer ierr

call MPI_INIT( ierr)
print *,'Hola Mundo'
call MPI_FINALIZE( ierr)
end
```

Este escrito en un archivo, llamado por ejemplo *hola_mundo.f*, es posible compilarlo y ejecutarlo con las siguientes líneas:

mpif77 –o hola hola_mundo.f

mpirun –np 3 hola

2.5.3. Información sobre los procesos

Algunas cuestiones nos llevan a requerir saber cuántos procesos se están corriendo, que procesos se están corriendo en cierta maquina, y como es que se identifican dichos procesos. El número de procesos se obtiene con la función *MPI_Comm_size*. El identificador para un proceso es un numero entre cero (0) y el total de procesos menos uno (*procesos*–1), este identificador es obtenido con la función *MPI_Comm_rank*. Si se quiere saber en qué maquina se está corriendo, entonces se tiene la función *MPI_Get_processor_name*. En las tablas 2.1, 2.2 y 2.3 se detallan los parámetros de dichas funciones.

MPI_Comm_rank(comm, my_id)	
Retorna el identificador o rango de un proceso	
comm	comunicador del cómputo
nproc	variable donde se retornara el identificador del proceso en el grupo comm.
como función	int MPI_Comm_rank(MPI_Comm comm, int * my_id)
como rutina	MPI_COMM_RANK(comm, my_id, ierror) INTEGER comm, my_id, ierror

Tabla 2.1: Función con la que se obtienen el rango de un proceso.

MPI maneja el sistema de memoria que es usado para ordenar mensajes y las representaciones internas de objetos de MPI como grupos, comunicadores, tipos de datos. El usuario no puede accesar directamente a esta memoria y los objetos son opacos (su tamaño y forma no son visibles al usuario). Estos objetos son accesados por el usuario mediante asas.

MPI_Comm_size(comm, nprocs)	
Retorna el número de procesos involucrados en un cómputo.	
comm	comunicador del cómputo
nproc	variable donde se retornara número de procesos involucrados del proceso en el grupo comm.
como función	int MPI_Comm_size(MPI_Comm comm, int * nprocs)
como rutina	MPI_COMM_RANK(comm, nprocs, ierror) INTEGER comm, nprocs, ierror

Tabla 2.2: Función con la que se obtienen el número de procesos.

Un comunicador identifica un grupo de procesos. Ellos proveen un mecanismo para identificar un subgrupo de procesos en el desarrollo de un programa. El valor por omisión es *MPI_COMM_WORLD*, que identifica a todos los procesos involucrados en un cómputo.

MPI_Get_processor_name(name, resultlen)	
Retorna el nombre del procesador sobre el cual se está corriendo al momento de la llamada	
name	variable donde se regresa el nombre del procesador
resultlen	longitud del número de caracteres de name
como función	int MPI_Get_processor_name(char *name, int * resultlen)
como rutina	MPI_GET_PROCESSOR_NAME(name, resultlen, ierror) INTEGER resultlen, ierror CHARACTER*(*) name

Tabla 2.3: Función con la que se obtienen el rango de un proceso.

En el siguiente código se muestra la modificación del programa *hola_mundo* en C, en donde cada proceso solicita su rango, la maquina sobre cual se está corriendo y el número total de procesos.

```
 #include<stdio.h>
#include<mpi.h>
int my_id, nproc, resultlen;
char name[20];
int main(int argc, char **argv )
{
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&my_id);
   MPI_Comm_size(MPI_COMM_WORLD,&nproc);
   MPI_Get_processor_name(name,&resultlen);
   name[resultlen]='\0';
   printf("hola mundo Soy %d de %d procesadores,
  corriendo en %s",my_id,nproc,name);
   MPI_Finalize();
}
```

Y en lenguje fortran el código quedaria de la siguiente manera:

```
     program main
    include 'mpif.h'
    integer ierr, my_id, nproc

   call MPI_INIT( ierr)
   call MPI_COMM_RANK(MPI_COMM_WORLD, my_id, ierr)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc,ierr)
   print *,'hola mundo soy ',my_id, ' de ',nproc
   call MPI_FINALIZE( ierr)
   end
```

2.5.4. Midiendo el tiempo

Obtener el tiempo de ejecución de un programa es importante para establecer su eficiencia y aceleración como se estableció en algunos apartados del capítulo I. MPI ofrece funciones para medir dicho tiempo (Tabla 2.4).

La función *MPI_Wtime* devuelve un punto flotante que representa el número de segundos transcurridos a partir de un cierto tiempo pasado, el cual garantiza que no cambia durante la vida del proceso.

MPI_Wtime()	
Retorna el número de segundos transcurridos a partir de cierto tiempo pasado	
Como función	MPI_Wtime(void)
Como rutina	MPI_WTIME()

Tabla 2.4: Función para medir el tiempo.

El código para el uso de *MPI_Wtime* debe ser similar al siguiente:

```
 .   .   .
double tiempo_inicial, tiempo_final;
tiempo_inicial=MPI_Wtime();
. . . codigo al que se le quiere medir el tiempo . . .
tiempo_final=MPI_Wtime();
printf("El codigo fue ejecutado en %f segundos",
 tiempo_final-tiempo_inicial);
. . .
```

2.5.5. Comunicación punto a punto

La comunicación punto a punto es un mecanismo básico en la trasferencia de mensajes entre un par de procesos, uno enviando y el otro recibiendo. La mayoría de MPI esta basado en comunicación punto a punto, por lo cual este tema es fundamental para el aprendizaje y uso de MPI.

MPI tiene diversas funciones para el envio y recepción de mensajes para la comunicación de datos de cierto tipo. Estas funciones por ello cuentan con argumentos que indican el tipo de dato que es enviado o recibido, esto para dar soporte a la heterogeneidad, y argumentos que indican cuantos datos de este tipo son enviados. La información sobre el tipo de dato es para realizar la conversión en la representación cuando se envían datos de una arquitectura a otra. Las tablas 2.6 y 2.5 muestran la equivalencia entre tipos de datos. Estas funciones también cuentan con una etiqueta (tag), para que el receptor se le permita seleccionar un mensaje en particular si así lo desea.

Tipo de dato en MPI	**Tipo de dato en fortran**
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISON	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER

Tabla 2.5: Equivalencias de tipos de datos entre MPI y fortran.

Tipo de dato en MPI	**Tipo de dato en C**
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Tabla 2.6: Equivalencias de tipos de datos entre MPI y C.

Las funciones básicas para el envio y recepción con bloqueo de datos son *MPI_Send* y *MPI_Recv* respectivamente (Tablas 2.7 y 2.8). Estas funciones tienen respectivamente el argumento para saber a qué proceso van destinados los datos y de que proceso se reciben datos. Es bueno resaltar que la función *MPI_Recv* en el receptor debe de tener su correspondiente *MPI_Send* en el destinatario, dado que estas funciones son con bloqueo, es decir, *MPI_Recv* no procede hasta haber recibido el mensaje. Por otro lado, en C, *status* es una estructura del tipo *MPI_status* que contiene información como el proceso que envía el mensaje (*status.MPI_SOURCE*), la etiqueta (*status.MPI_TAG*) y el código de error (*status.MPI_ERROR*). En el caso de fortran, *status* es un arreglo y los valores estarán almacenados en *status (MPI_SOURCE)*, *status (MPI_TAG)* y *status (MPI_ERROR)*.

MPI_Recv(buf, count, datatype, source, tag, comm, status)	
Recibe un mensaje	
buf	dirección del buffer de recepción.
count	número de elementos a recibir
datatype	tipo de datos de los elementos del buffer de recepción.
dest	identificador del proceso enviador
tag	etiqueta del mensaje
comm	comunicador
status	objeto status
Como función	int MPI_Recv(void *buf,int count,MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status status)
Como rutina	MPI_SEND(BUF,COUNT,DATATYPE,DEST,TAG,COMM, STATUS,IERROR) INTEGER COUNT, DATATYPE,DEST,TAF, COMM,IERROR

Tabla 2.7: Función básica de recepción de mensajes.

2.5.6. Más funciones en MPI

Con las ocho funciones básicas que se han descrito en seciones anteriores, es posible escribir una variedad de programas.

Por otro lado, MPI no solo cuenta con solo estas funciones, si no que el estándar MPI es extenso, el cual cuenta con alrededor de 129 funciones [36], muchas de las cuales cuentan con numerosas variantes y parámetros, algunas de ellas se encargan de comunicaciones colectivas, otras para comunicación sin bloqueo. Todo esto para permitir flexibilidad cuando se requiera. Además no se debe ser un experto en todas las funciones en MPI para usarlo satisfactoriamente. Para lograr el objetivo de este trabajo solo fueron necesarias ocho funciones básicas.

MPI_Send(buf, count, datatype, dest, tag, comm)	
Envía un mensaje con datos de un proceso a otro	
buf	dirección del buffer de envio.
count	número de elementos a enviar
datatype	tipo de datos de los elementos del buffer de envio
dest	identificador del proceso destino
tag	etiqueta del mensaje
comm	comunicador
Como función	int MPI_Send(void *buf,int count,MPI_Datatype datatype, int dest,int tag,MPI_Comm comm)
Como rutina	MPI_SEND(BUF,COUNT,DATATYPE,DEST,TAG, COMM,IERROR)

Tabla 2.8: Función básica de envio de mensajes.

Capítulo 3

MÉTODOS NÚMERICOS Y PARALELIZACIÓN BÁSICA.

Una gran variedad de problemas pueden ser resueltos a través del álgebra lineal, entre ellos podemos citar: la determinación del potencial en redes eléctricas, el cálculo de tensión de una estructura metálica en construcción civil. Un problema matemático al que se reducen todos estos casos es resolver un sistema de ecuaciones lineales simultáneas [51], que toman la forma:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \tag{3.1}$$

o en forma compacta:

$$Ax = b \tag{3.2}$$

3.1. Determinante

Un concepto teórico importante en algunos métodos para la solución de un sistema de ecuaciones lineales simultáneas es el de determinante [52, Cap. 5] asociado a una matriz cuadrada. Primero se tratara brevemente las permutaciones, las cuales se usarán en la definición de determinante.

Sea $S = \{1, 2, 3, 4 \cdots, n\}$ el conjunto de enteros del 1 al n, ordenados de manera

ascendente. Un reordenamiento $j_1, j_2, j_3, \cdots j_n$ de los elementos de S se denomina permutación de S.

Una permutación $j_1, j_2, j_3, \cdots j_n$ de S tiene una inversión si un entero mayor, j_s procede a un entero menor j_r. Una permutación es par si su número total de inversiones es par, o impar si su número total de inversiones es impar.

Luego, el determinante de una matriz cuadrada A, denotado por $|A|$, esta dado por

$$|A| = \sum_{j_1,j_2,j_3,\cdots j_n \text{ permutacion de } s} (-1)^{\text{paridad}} a_{1j_1} a_{2j_2} \cdots a_{nj_n} \tag{3.3}$$

donde $(-1)^{\text{paridad}}$ es igual a 1 si la permutación $j_1, j_2, j_3, \cdots j_n$ de S tiene permutación par, de lo contrario deberá tomar el valor de -1.

3.2. La inversa de una matriz

Un método utilizado para resolver un sistema de ecuaciones lineales simultáneas (y tal vez el más primitivo) es mediante el cálculo de la matriz inversa [52], la cual al multiplicarla por la matriz original da como resultado la matriz identidad. Dicho cálculo puede resolverse por varios métodos. Aquí solo se describe uno de ellos. Para ello es necesaria la descripción de cofactor y de matriz adjunta.

Sea A una matriz cuadrada, sea M_{ij} la submatriz obtenida de haber eliminado la i–ésima fila y la j–ésima columna de A. El determinante $|M_{ij}|$ se denomina menor de a_{ij}. Así el cofactor A_{ij} de a_{ij} se define como $A_{ij} = (-1)^{i+j}|M_{ij}|$. De la misma manera, la matriz $adj(A)$, denominada adjunta de A, es la matriz cuya entrada (i, j) es el cofactor A_{ij} de a_{ij}.

Con estos conceptos es posible demostrar (lo cual no se hará aquí) que si el determinante de una matriz es diferente de cero, entonces la inversa de dicha matriz es igual a la matriz adjunta entre el determinante de la matriz original.

$$A^{-1} = \frac{1}{|A|} adj(A), \text{ para } |A| \neq 0 \tag{3.4}$$

Luego para el sistema de ecuaciones lineales simultáneas (ecuación 3.1) se obtiene:

$$x = A^{-1}b \tag{3.5}$$

3.2.1. Análisis

Obsérvese que para el cálculo de la inversa de la matriz para obtener la solución a un sistema de ecuaciones, es necesario primeramente obtener el determinante de la matriz, y en segundo lugar se debe de obtener la adjunta de la matriz.

Suponiendo que la matriz es de $n \times n$, entonces para obtener el determinante se debe de considerar $n!$ operaciones, las cuales corresponden a las permutaciones.

Por otro lado, para determinar cada cofactor de la matriz adjunta es necesario hacer $(n-1)!$ operaciones. Por lo que el cálculo de la adjunta de la matriz nos costara $n \times n \times (n-1)!$ operaciones.

De esta forma, el número de operaciones [53] para determinar la inversa de una matriz de $n \times n$ es $n \times n + n^2 \times n! = O(n^2 \times n!)$. Lo cual dice claramente que ningún analista numérico considera un algortimo basado en el determinante para matices grandes.

3.2.2. Implementación en MPI

Para llevar la teoría de los apartados anteriores a la práctica, se implemento el cálculo de la solución de un sistema de ecuaciones simultáneas en forma paralela en MPI.

Primero se hace que el proceso que tiene rango cero tome los datos. Después este mismo proceso deberá, por medio de mensajes, darles los datos a los demás proceso. Luego, al calcular el número de procesos y almacenarlo en *nprocs*, para el reparto de tareas, se enumeran las entradas de la matriz adjunta del 1 hasta $n \times n$, entonces a cada uno de estos números se les saca modulo *nprocs*, dicho valor coincidirá con el rango de uno de los procesos, entonces al proceso que tenga dicho rango deberá calcular la entrada correspondiente de la matriz adjunta, dado que cada proceso ya tiene a su disposición los datos. Por último, el proceso que repartió los datos calcula el determinante de la matriz original y recibe los resultados de los demás procesos y los ordena en las entradas correspondientes de la matriz, al mismo tiempo que divide cada elemento entre el determinante de la matriz original, para así obtener la matriz inversa. El código de esta implementación esta de manera explícita en el apéndice B

(código de implementación de mínimos cuadrados en MPI) en lenguaje Fortran.

3.3. Interpolación

En general, en la interpolación se hace una representación que consiste establecer combinaciones lineales de funciones seleccionadas arbitrariamente. Sea $f(x)$ la función a aproximarse, $b = \{b_0(x), b_1(x), \cdots, b_n(x)\}$ el conjunto de funciones tomadas a priori o base de interpolación, entonces:

$$f(x) \approx \sum_{i=0}^{n} c_i b_i(x) \tag{3.6}$$

donde c_i son los coeficientes de interpolación, los cuales se deben calcular para obtener la interpolación.

3.3.1. Matriz de interpolación

Un polinomio generalizado puede construirse a partir de cualquier base de interpolación. La técnica para hacerlo es la siguiente. Sea $\{x_0, x_1, \cdots, x_n\}$ el conjuntos de puntos en los cuales la función f es conocida (soporte de interpolación), y sea $b = \{b_0(x), b_1(x), \cdots, b_n(x)\}$ la base de interpolación. Se busca el polinomio generalizado:

$$P_n(x) = \sum_{i=0}^{n} c_i b_i(x) \tag{3.7}$$

El cual debe satisfacer las $n + 1$ condiciones:

$$P_n(x_k) = f(x_k) \text{ para } k = 0, 1, \cdots, n \tag{3.8}$$

Es decir:

$$\begin{array}{ccccc} c_0 b_0(x_0) + c_1 b_1(x_0) + \cdots + c_n b_n(x_0) = f(x_0) \\ c_0 b_0(x_1) + c_1 b_1(x_1) + \cdots + c_n b_n(x_1) = f(x_1) \\ \vdots \quad \vdots \quad \vdots \quad \cdots \quad \vdots \\ c_0 b_0(x_n) + c_1 b_1(x_n) + \cdots + c_n b_n(x_n) = f(x_n) \end{array} \tag{3.9}$$

Lo cual equivale en forma matricial a:

$$BC = F \tag{3.10}$$

El cual se puede solucionar con el método ya analizado previamente. La calidad de intepolación es dada por la base de interpolación.

3.3.2. Interpolación de Lagrange

En la sección anterior se mostro una forma teórica de construirse una función analítica apartir de un sierto número de puntos donde es conocida la función. Sin embargo, existen otros métodos de interpolación, donde los cálculos se realizan directmante; entre éstos se encuentra la interpolación de Lagrange.

Se parte nuevamente del conjunto de puntos $\{x_0, x_1, \cdots, x_n\}$ donde f es conocida, entonces buscamos el polinomio

$$P_n(x) = \sum_{i=0}^{n} a_i \left[\prod_{\substack{j=0 \\ j \neq i}}^{n} (x - x_j) \right], \tag{3.11}$$

el cual al evaluar en x_k para $k \in 0, 1, \ldots, n$, se obtiene

$$P_n(x_k) = a_k \left[\prod_{\substack{j=0 \\ j \neq k}}^{n} (x_k - x_j) \right], \tag{3.12}$$

y como se quiere que $P_n(x_k) = f(x_k)$, entonces de la ecuación 3.12 se tiene que

$$f(x_k) = a_k \left[\prod_{\substack{j=0 \\ j \neq k}}^{n} (x_k - x_j) \right], \tag{3.13}$$

luego que

$$a_k = \frac{f(x_k)}{\left[\prod\limits_{\substack{j=0 \\ j \neq k}}^{n} (x_k - x_j) \right]}, \tag{3.14}$$

y por lo tanto tenemos

$$P_n(x) = \sum_{i=0}^{n} L_i(x) f(x_i), \tag{3.15}$$

donde $L_j(x) = \prod_{\substack{i=0 \\ i\neq j}}^{n} \frac{(x-x_i)}{(x_j-x_i)}$, el cual se le conoce como polinomio de Lagrange de grado j [55], y tiene la siguiente propiedad:

$$L_j(x_k) = \delta_{jk} = \begin{cases} 1 & \text{para } j = k, \\ 0 & \text{para } j \neq k. \end{cases} \tag{3.16}$$

Análisis

Tomado como soporte teórico los puntos $x_0, x_1, \cdots, x_n$ se puede ver que para formar el polinomio de interpolación de Lagrange es necesario calcular los términos $L_j(x)$ y su marlos. Para sumarlos son necesarias $n-1$ operaciones. Los $L_j(x)$ están constituidos por un cociente, en el cual tanto la parte del denominador como la parte del numerador están formadas por n términos que se multiplican, por lo cual calcular cada uno cuesta $n-1$ operaciones, y como esto es para cada j desde 0 hasta n, entonces se concluye que el total de operaciones que se deben de hacer para determinar el polinomio de Lagrange es: $(n-1) + 2(n-1)(n+1) = O(n^2)$.

Implementación en MPI

Para implementar interpolación de Lagrange se hizo algo similar a la forma de implementar el cálculo de solución de un sistema de ecuaciones simultáneas, pero en este caso se calcularón los elementos $L_j(x)$ en diferentes procesos. Para lo cual, igualmente se calculo el numero de procesadores *nprocs*, y a cada número j del polinomio $L_j(x)$ se le saco modulo *nprocs*, lo cual ayudo a dar un procedimiento similar para la repartición de tareas a los procesos. Al final el proceso con rango cero recibe los resultados de lo demás procesos y hace los suma para obtener el polinomio de Lagrange. En el apéndice A se muestra el código en C de esta implementación.

3.4. Aproximación por mínimos cuadrados

En caso de interpolación, una condición impuesta al polinomio es que debe coincidir con la función en los puntos del soporte (ecuación 3.8). Sin embargo, en situaciones reales, donde los valores $f(x_k)$ son obtenidos experimentalmente, los valores no son tan exactos debido a múltiples factores. En estas condiciones, no es necesario exigir que el polinomio coincida con la función de aproximación en los puntos del soporte.

Por ello es necesario introducir otro tipo de aproximaciones que no requiera de estas aproximaciones. Primero se darán conceptos indispensables para dicho fin.

Sea V el espacio de las funciones suaves en intervalo $[a, b]$, entonces se define el producto escalar en el caso continuo como:

$$\langle f, g\rangle = \int_a^b f(x)g(x)dx, \text{ para } f, g \in V, \tag{3.17}$$

ó bien para el caso discreto como:

$$\langle f, g\rangle = \sum_{x_k \in S} f(x_k)g(x_k). \tag{3.18}$$

donde S es el conjunto de puntos donde se conoce f y g.

Por otro lado se define la norma sobre V como:

$$||f|| = \sqrt{\langle f, f\rangle}. \tag{3.19}$$

Con el producto escalar y la norma definidos de esta forma, se dice que V constituye un espacio de Hilbert [56]. Ahora considérese la base formada por las funciones $b_0, b_1, \cdots, b_n$, con las cuales se quiere construir el polinomio de aproximación:

$$P_m(x) = \sum_{m=0}^{n} c_i b_i(x) \tag{3.20}$$

Entonces el error cometido al aproximar f con P_m es:

$$E(c_k) = f(x) - P_m(x) \tag{3.21}$$

Este error que depende de los coeficientes c_k , los cuales controlan la calidad de aproximación de $P_m(x)$, la cual debe de ser lo más optima. Para lograr esto se requiere que el error sea mínimo.

Pero como $||E||$ no siempre es derivable en el intervalo $[a, b]$, es mejor minimizar $||E||^2$, para lo cual es suficiente que:

$$\frac{\partial ||E||^2}{\partial c_k} = 0 \tag{3.22}$$

Entonces:

$$\begin{array}{ll} \frac{\partial ||E||^2}{\partial c_k} & = \frac{\partial}{\partial c_k} \int_a^b [f(x) - P_m(x)]^2 dx \\ & = -2 \int_a^b [f(x) - P_m(x)] b_k(x) dx \\ & = -2\langle f(x) - P_m(x), b_k(x)\rangle = 0 \end{array} \tag{3.23}$$

Lo cual es equivalente a:

$$\langle f(x) - P_m(x), b_k(x)\rangle = 0 \tag{3.24}$$

Pero como $P_m(x) = \sum_{i=0}^{m} c_i b_i(x)$, entonces la expresión anterior se puede escribir como:

$$\begin{aligned}\langle f(x) - \sum_{i=0}^{m} c_i b_i(x), b_k(x)\rangle &= \langle f(x), b_k(x)\rangle - c_0\langle b_0(x), b_k(x)\rangle \\ &\quad -c_1\langle b_1(x), b_k(x)\rangle - \cdots - c_m\langle b_m(x), b_k(x)\rangle = 0\end{aligned} \tag{3.25}$$

Finalmente tenemos:

$$c_0\langle b_0(x), b_k(x)\rangle + \cdots + c_m\langle b_m(x), b_k(x)\rangle = \langle f(x), b_k(x)\rangle \tag{3.26}$$

Dada que se debe de cumplir para toda $b_k(x)$ con $k = 0, 1, 2, \cdots, m$ entonces lo anterior es equivalente al sistema lineal:

$$\begin{pmatrix} b_{00} & b_{01} & b_{02} & \cdots & b_{0m} \\ b_{10} & b_{11} & b_{12} & \cdots & b_{1m} \\ \cdots & \cdots & \cdots & & \cdots \\ b_{m0} & b_{m1} & b_{m2} & \cdots & b_{mm} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix} = \begin{pmatrix} \langle f, b_0\rangle \\ \langle f, b_1\rangle \\ \langle f, b_2\rangle \\ \vdots \\ \langle f, b_m\rangle \end{pmatrix} \tag{3.27}$$

En donde $b_{ij} = \langle b_i, b_j\rangle$. Si se considera el caso continuo, el sistema anterior se llama aproximación en *media cuadrática*. Para el caso discreto dicho sistema constituye el método de *mínimos cuadrados* [57, Cap. 4].

3.4.1. Análisis

En el método de mínimos cuadrados con un soporte constituido por n puntos y una base con m elementos, para el cálculo tanto de cada uno de los elementos $\langle f, b_k\rangle$ como el de cada uno de los elementos b_{ij} del sistema se necesitan $2n-1$ operaciones (n multiplicaciones y n-1 sumas), por lo cual para llenar todos los elementos del sistema es necesario implementar $(m(m+1))(2n-1)$ operaciones. Y como ya se analizo previamente que el número de operaciones que se requieren para resolver un sistema de ecuaciones simultáneas con n ecuaciones invirtiendo la matriz (aunque este sistema se puede resolver por el método de Cholesky (el cual se presentara mas adelante), es $O(n!)$, entonces el número de operaciones requeridas para una aproximación por mínimos cuadrados invirtiendo la matriz del sistema es $m(m+1)(2n-1) + O(m!) = O(m!) + O(n)$.

3.4.2. Implementación en MPI

Para la implementación de mínimos cuadrado, para el reparto de tareas, al igual que en implementaciones anteriormente descritas, se calcularon los elementos b_{ij} y $\langle f, b_k \rangle$ mediante el uso de modulo *nprocs*. El sistema de ecuaciones, aunque se puede resolver por el método de Cholesky (método que se expondra en la siguiente sección), se resolvió por medio del cálculo de la matriz inversa para hacer notar las diferencias de tiempo, eficiencia y aceleración entre un nodo y varios. En el apéndice B se muestra el código en Fortran de la implementación.

3.5. Método de Cholesky

En esta sección se presentara el método de Cholesky, el cual se puede usar para la solución del sistema de ecuaciones en el método de mínimos cuadrados [58] (ecuación 3.27).

Considerese una matriz de dimensión $m \times n$, $A \in M_{m \times n}$. Contruyase ahora la matriz simétrica de $n \times n$ siguiente:

$$S = A^t A \in M_{n \times n} \tag{3.28}$$

Esta matriz posee la siguiente propiedad: dado un vector columna arbitrario distinto del vector nulo y de dimensión n, x, se define el vector $y = Ax$, de la misma dimensión que el anterior, y entonces se cumple que el escalar

$$x^t S x = x^t A^t A x = (xA)^t A x = y^t y$$

no es negativo puesto que se trata de la suma de los módulos al cuadrado de los elementos del vector y:

$$y^t y = \sum_{i=1}^{n} |y_i|^2 \geq 0 \tag{3.29}$$

Esto, al ocurrir para cualquier vector x no nulo constituye la definición de una matriz definida no negativa (o matriz definida positiva). Lo denotamos como $S \geq 0$.

Supongase que se quiere resolver el sistema de ecuaciones:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \tag{3.30}$$

o en forma compacta:

$$Ax = b \tag{3.31}$$

con A simétrica y definida positiva, entonces A puede ser factorizada como $A = TT^t$, donde T es una matriz triangular inferior, esta es llamada la Factorización Cholesky [59]. La ventaja que nos da dicha factorización es que los sistemas triangulares son sencillos de efectuar.

Se determinara primero la matriz T, se quiere que:

$$TT^t x = Ax \tag{3.32}$$

Es decir:

$$\begin{aligned}\begin{pmatrix} \alpha_{11} & 0 & \cdots & 0 \\ \alpha_{21} & \alpha_{22} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ \alpha_{n1} & \alpha_{n2} & \cdots & \alpha_{nn} \end{pmatrix} \begin{pmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1n} \\ 0 & \alpha_{22} & \cdots & \alpha_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \alpha_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \\ = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}\end{aligned} \tag{3.33}$$

Entonces:

$$a_{11} = \alpha_{11}^2 \Rightarrow \alpha_{11} = \sqrt{a_{11}}$$

$$a_{21} = \alpha_{21}\alpha_{11} \Rightarrow \alpha_{21} = \frac{a_{21}}{\alpha_{11}}$$

Para $1 < j \leq n$ tenemos:

$$\alpha_{j1} = \frac{a_{j1}}{\alpha_{11}}$$

Por otro lado:

$$a_{22} = \alpha_{21}^2 + \alpha_{22}^2 \Rightarrow \alpha_{22} = \sqrt{a_{22} - \alpha_{21}^2}$$

$$a_{32} = \alpha_{31}\alpha_{21} + \alpha_{32}\alpha_{22} \Rightarrow \alpha_{32} = \frac{a_{32} - \alpha_{31}\alpha_{21}}{\alpha_{22}}$$

$$a_{42} = \alpha_{41}\alpha_{21} + \alpha_{42}\alpha_{22} \Rightarrow \alpha_{42} = \frac{a_{42} - \alpha_{41}\alpha_{21}}{\alpha_{22}}$$

Entonces para $2 < j \leq n$ se tiene:

$$\alpha_{j2} = \frac{a_{j2} - \alpha_{j1}\alpha_{21}}{\alpha_{22}}$$

Repitiendo el proceso es posible mecanizar y establecer el siguiente algoritmo:

$$a_{ii} = \sum_{k=1}^{i} \alpha_{ik}^2 \Rightarrow \alpha_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} \alpha_{ik}^2} \tag{3.34}$$

Y para $i < j \leq n$:

$$a_{ij} = \sum_{k=1}^{i} \alpha_{jk}\alpha_{ik} \Rightarrow \alpha_{ji} = \frac{a_{ji} - \sum\limits_{k=1}^{i-1} \alpha_{jk}\alpha_{ik}}{\alpha_{ii}} \tag{3.35}$$

Una vez calculados los elementos de la triangular T se tiene:

$$Ax = b \Rightarrow TT^t x = b$$

Sea $y = T^t x \Rightarrow Ty = b$ será el primer sistema a resolver.

Esto es imediato pues:

$$\begin{pmatrix} \alpha_{11} & 0 & 0 & \dots & 0 \\ \alpha_{21} & \alpha_{22} & 0 & \dots & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ \alpha_{n1} & \alpha_{n2} & \alpha_{n3} & \dots & \alpha_{nn} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix} \tag{3.36}$$

$$\Rightarrow \begin{cases} \alpha_{11}y_1 = b_1 & \Rightarrow & y_1 = \frac{b_1}{\alpha_{11}} \\ \alpha_{21}y_1 + \alpha_{22}y_2 = b_2 & \Rightarrow & y_2 = \frac{b_2 - \alpha_{21}y_1}{\alpha_{22}} \\ \vdots & \vdots & \vdots \\ \alpha_{n1}y_1 + \alpha_{n2}y_2 \ldots + \alpha_{nn}y_n = b_n & \Rightarrow & y_n = \frac{b_n - \alpha_{n1}y_1 + \alpha_{n2}y_2 \ldots + \alpha_{nn-1}y_{n-1}}{\alpha_{nn}} \end{cases}$$

De donde el algoritmo para sistemas triangulares inferiores es:

$$\begin{aligned} y_1 &= \tfrac{b_1}{\alpha_{11}} \\ y_i &= \tfrac{b_i - \sum\limits_{k=1}^{i-1} \alpha_{ik}y_k}{\alpha_{ii}},\ i = 2, \cdots, n \end{aligned} \tag{3.37}$$

Queda por resolver: $T^t x = y$:

$$\begin{pmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1n} \\ 0 & \alpha_{22} & \cdots & \alpha_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \alpha_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \tag{3.38}$$

Al revés:

$$\Rightarrow \begin{cases} \alpha_{nn} x_n = y_n & \Rightarrow & x_n = \frac{y_n}{\alpha_{nn}} \\ \alpha_{n-1n-1} x_{n-1} + \alpha_{nn} x_n = y_{n-1} & \Rightarrow & \frac{y_{n-1} - \alpha_{nn} x_n}{\alpha_{n-1n-1}} \\ \vdots & \vdots & \vdots \\ \alpha_{11} x_1 + \ldots + \alpha_{1n} x_n = y_1 & \Rightarrow & x_1 = \frac{y_1 - \alpha_{12} x_2 + \ldots + \alpha_{1n} x_n}{\alpha_{11}} \end{cases}$$

Entonces el algoritmo para sistemas tringulares superiores es:

$$\begin{aligned} x_n &= \tfrac{y_n}{\alpha_{nn}} \\ x_i &= \frac{y_i - \sum_{k=i+1}^{n} \alpha_{ik} x_k}{\alpha_{ii}} \end{aligned} \tag{3.39}$$

3.5.1. Análisis

Para resolver un sistema de ecuaciones similar al de la ecuación (3.30), primeramente se deben de calcular los elementos α_{ij} de la matriz triangular T, en donde para el calculo de cada uno de esos elementos nos toma j operaciones, por lo tanto el calculo de dicha matriz son necesarias $\sum_{i=1}^{n} \sum_{j=1}^{i} j = \sum_{i=1}^{n} \frac{i(i+1)}{2}$. Una vez teniendo la matriz T se deben calcular dos sistemas de ecuaciones triangulares, en los cuales para determinar la incognita x_j se deben realizar j operaciones, por lo cual el numero de operaciones para resolver dichos sistemas triangulares son $2(\sum_{i=1}^{n} i) = n(n+1)$. En conclusión, el numero de operaciones que son necesarias para resolver un sistema de ecuaciones con en método de Cholesky es $\sum_{i=1}^{n} \frac{i(i+1)}{2} + n(n+1) = O(n^3)$.

3.5.2. Implementación en MPI

Como se mencionó anteriormente, el sistema de ecuaciones del método de mínimos se puede resolver mediante el método de Cholesky. Para hacer la implementación de

mínimos cuadrados con Cholesky se calcularon los elementos b_{ij} y $\langle f, b_k \rangle$ de la misma manera que describio enteriormente, mediante el uso de modulo *nprocs*. Para resolver el calculo de la matriz triangular T (ecuación (3.32)), se repartio de manera análoga las tareas, primero el proceso con rango cero calcula el elemento α_{ii}, después reparte toda la información que tiene sobre la matriz T entre todos los procesos. Luego cada proceso k (rango del proceso) calcula los elementos α_{ij} de la columna i de T que cumplan con $j = nproc + k$, donde *nproc* es el numero de procesadores. dichos elementos son enviados al proceso cero para que ponga la información en la matriz T. Al final el proceso con rango cero calcula el sistema de ecuaciones triangulares. El código de esta implementación en lenguaje C se muestra en el apéndice C.

Capítulo 4

RESULTADOS Y CONCLUSIONES

En este capítulo se complementa las características de los experimentos realizados y los resultados obtenidos. Primero daremos las características de los equipos que se utilizaron, así como la forma de configuración que se uso. Luego, mostraremos los resultados obtenidos.

4.1. Descripción del equipo

Los experimentos fueron realizados en el Laboratorio de Computo 1, de la Facultad en Cs. Físico–Matemáticas, de la Universidad Michoacana de San Nicolás de Hidalgo.

Se instalaron dos implementaciones de MPI las cuales fueron MPICH y OPENMPI.

Para la implementación MPICH se hizo uso de:

- dos PC's Pentium IV, con una memoria RAM de 512 MB, caché de 512 KB, con velocidad de procesador de 2.40 GHz, disco duro de 80 GB y tarjeta de red DLINK.
- Switch Zonet 8 puertos.

Para la implementación OPENMPI se empleo:

- cuatro PC's Pentium IV, cada una con una memoria RAM de 512 MB, caché de 512 KB, con velocidad de procesador de 2.40 GHz, disco duro de 80 GB y tarjeta de red DLINK.
- Switch Zonet 8 puertos.

Ambas implementaciones se emplearon bajo la distribución Debian Denny v.5.0.1. En este trabajo solo se describe la forma de instalar OPENMPI, ya que para la instalación de MPICH se hace casi de forma similar, para ver los detalles de dicha instalación consultar la referencia [44].

4.2. Resultados

A continuación se muestran los resultados obtenidos para los diferentes métodos numéricos que se han descrito previamente.

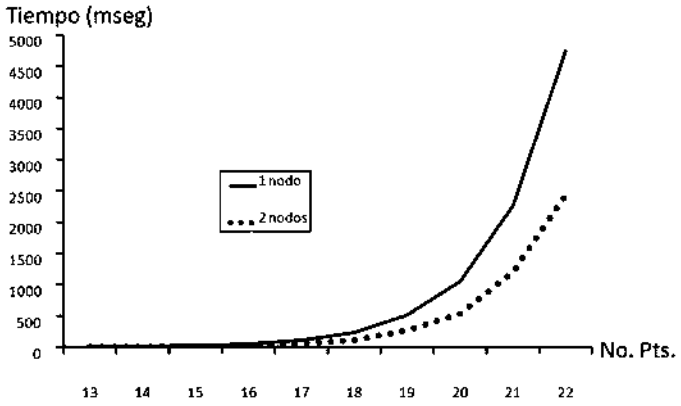

Figura 4.1: Tiempo de ejecución de la interpolación de Lagrange en MPICH.

Las figuras 4.1 y 4.2 muestran el tiempo obtenido al implementar interpolación de Lagrange sobre el Clúster con MPICH, se tomo como soporte de interpolación los puntos $0, 1, \cdots, n$ y el valor de la función en cada uno de estos puntos fue un numero aleatorio entre 0 y 100000.

Obsérvese que la fig. 4.2 respalda la Ley de Amdahl, para una interpolación de Lagrange con un soporte con menos de 10 puntos, un solo procesador obtiene el polinomio en menos tiempo que el clúster con 2 procesadores.

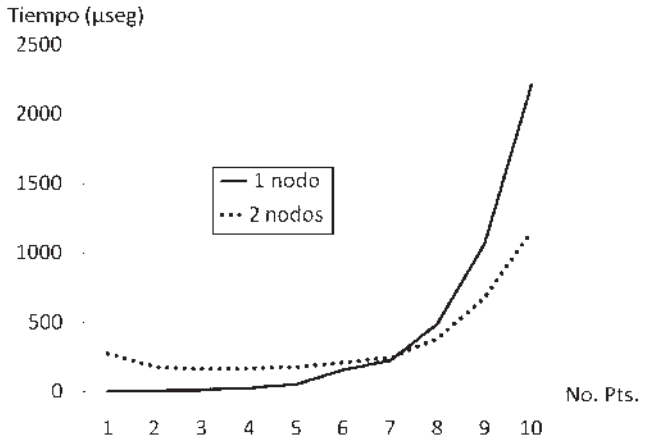

Figura 4.2: Resultados del tiempo de la implementación de Lagrange en MPICH.

Con dichos resultados se obtuvo la aceleración la cual es mostrada en la figura 4.3. Nótese que en este caso se obtuvo casi una aceleración lineal, de lo cual se concluye que el código esta cerca de un resultado optimo en lo que refiere a paralelización.

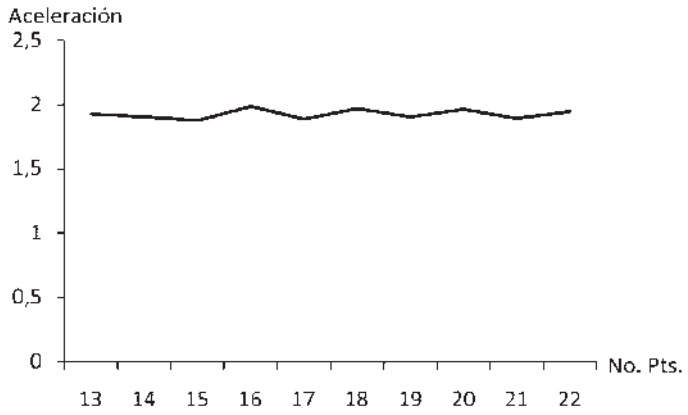

Figura 4.3: Aceleración de la ejecución de la interpolación de Lagrange en MPICH.

Por otra parte también se hizo el cálculo de la eficiencia, esto se muestra en la figura 4.4. Como se puede apreciar en la grafica de dicha figura, la eficiencia es cercana a 1, lo cual es una segunda forma de sustentar la conclusión de que el código esta cercano a una paralelización optima.

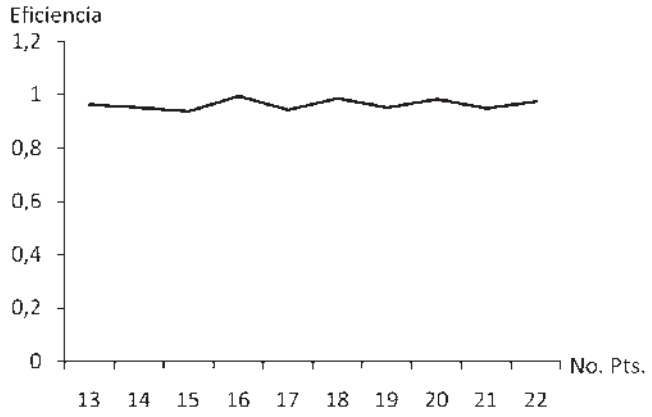

Figura 4.4: Eficiencia de la ejecución de la interpolación de Lagrange en MPICH con 2 nodos.

De manera análoga las figuras 4.5, 4.6 y 4.7 muestran el tiempo, la aceleración y la eficiencia que se obtuvo de ejecución del código de mínimos cuadrados (invirtiendo la matriz del sistema de ecuaciones) sobre el clúster con OPENMPI, respectivamente. Se tomaron los soportes de manera similar que en interpolación de Lagrange, se busco el polinomio de grado 7 para cada uno de ellos con la base $b = \{1.x.x^2, \cdots, x^6\}$.

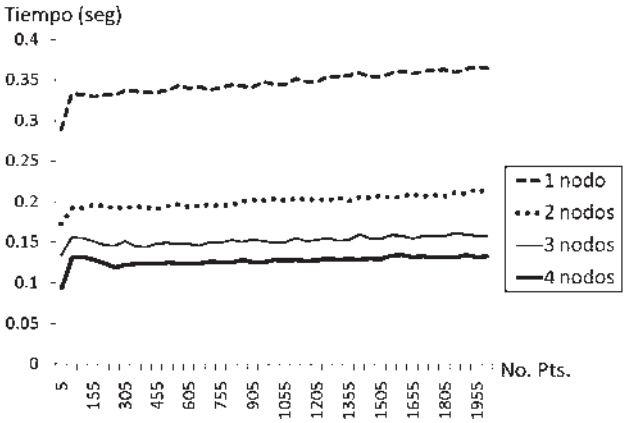

Figura 4.5: Resultados del tiempo de la implementación de la aproximación por mínimos cuadrados en OPENMPI.

En la figura 4.5 se observa que el tiempo no se reduce de manera lineal con respecto al aumento de nodos, una razón por lo cual sucede esto es porque algunas partes del

programa no se pueden paralelizar, como por ejemplo el envio y recepción de datos de un nodo a otro.

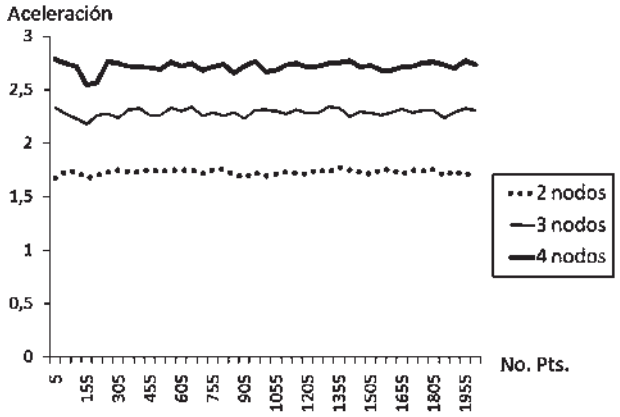

Figura 4.6: Aceleración de la ejecución de la implementación de la aproximación por mínimos cuadrados en OPENMPI.

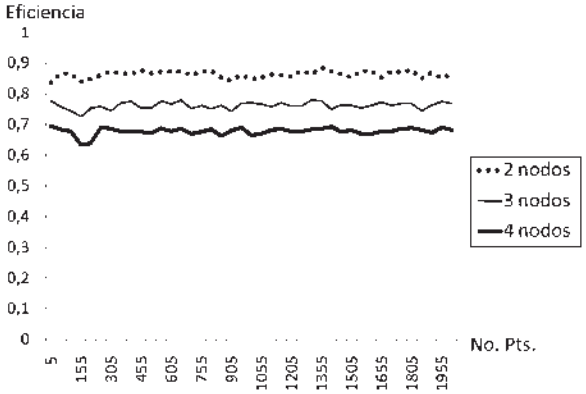

Figura 4.7: Eficiencia de la ejecución de la implementación de la aproximación por mínimos cuadrados en OPENMPI.

La aceleración y en la eficiencia se puede observar el mismo hecho que en el tiempo; la aceleración y la eficiencia no aumentan de manera lineal conforme se aumenta el número de nodos. El no obtener una aceleración cercana al número de nodos y una eficiencia cercana a 1, como lo muestran las figuras correspondientes, es otra consecuencia del hecho de que este método no se pueda paralelizar, como la

interpolación de Lagrange, en donde el envío, la recepción y ordenación de los datos toma poco tiempo en comparación con procesamiento de los mismos.

4.3. Conclusiones

El objetivo principal de esta tesis es proponer una forma de programar sobre MPI que permita el uso de varios equipos para realizar el cómputo de los programas y así disminuir el tiempo de proceso.

Las propuestas que se exponen en este trabajo demuestran que es posible paralelizar algunos métodos numéricos y ejecutarlos sobre MPI, con eficiencia y aceleración razonables.

En este trabajo, se propone reutilizar equipos (posiblemente "obsoletos") para realizar la programación de métodos numéricos en MPI. Se recomienda utilizar OpenMPI, ya que es una buena implementación de MPI. Sin embargo, aunque en muchos trabajos se muestran resultados favorables de MPICH [44], en la instalación de la última versión de MPICH que se realizo en el laboratorio de computo de la facultad de Ciencias Físico Matemáticas no se pudo escalar a mas de dos nodos, dado que la hora de correr el programa mostraba un error, el cual decía que no se podía trabajar sobre todos los nodos.

4.4. Expectativas

Los trabajos futuros que se proponen a raíz de los resultados obtenidos en esta tesis son:

- Usar MPI para la paralelización de los diversos métodos numéricos, como por ejemplo, para encontrar soluciones de ecuaciones parciales.
- Utilizar la paralelización con MPI para problemas reales.
- Programar en PVM y comparar su aceleración y eficiencia con MPI.

Apéndice A

Código en lenguaje C de Interpolación de Lagrange en MPI.

```
#include"mpi.h"
#define tam 1010
MPI_Status status;
    /*+++++++++++++++++++++++++++++++coefnjk++++++++++++++++++++
    | regresa el coeficiente de x^[n-j-1] del producto:  |
    | (x+x1)(x+x2)...(x-(x[k-1]))(x-(x[k+1]))...(x-xn) |
    | init  desde que parte de x va a empazar  a multiplicar|
    | n nÃºmero de elementos |
    | j coeficiente que va a regresar |
    | k ke elemento no debe considerar en el producto   |
    | x los elementos x1,x2,...,xn |
    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
int coefnjk(int ini,int n,int j,int k,float *x)
{
 int l=ini,sum;
 if(j==0)
  return(1);
  for(sum=0;l<=n-j;l++)
   if(l!=k)
   {
    sum=sum+x[l]*coefnjk(l+1,n,j-1,k,x);
   }
```

```
  return(sum);
}
int main(int argc, char **argv)
{
 float xi[tam],f[tam],coef[tam],coefp[tam],aux;
 int npts,i,j,prod,size,tag=99,rank;
 double ini, fin;
 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD,&size);
 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
 if(rank==0)
 {
  printf("Â¿cuantos pts son?");
  scanf("%d",&npts);
  for(i=0;i<npts;i++)
  {
   printf("Damen el punto %d:",i);
   scanf("%f",&aux);
   xi[i]=aux;
   printf("Dame f(%f):",aux);
   scanf("%f",&aux);
   f[i]=aux;
//pa hacelo de la forma (x+(-x1))(x+(-x2))...
   xi[i]=(-1)*xi[i];
  }
  ini=MPI_Wtime();
  for(i=0;i<npts;i++)
   coef[i]=0.0;
  for(j=1;j<size;j++)//mandando los datos
  {
   MPI_Send(&npts,1,MPI_INT,j,tag,MPI_COMM_WORLD);
   MPI_Send(xi,npts,MPI_FLOAT,j,tag,MPI_COMM_WORLD);
   MPI_Send(f,npts,MPI_FLOAT,j,tag,MPI_COMM_WORLD);
  }
  //calculando mi parte  los polinomios L[n,k]
  for(j=0;j<npts;j++)
   if(j%size==0)
   {
    prod=1;
```

```
    for(i=0;i<npts;i++)
     if(i!=j)
      prod=prod*(xi[j]-xi[i]);
//calculando los coeficientes del polinomio
    for(i=0;i<npts;i++)
 coef[i]=coef[i]+(f[j]*coefnjk(0,npts,npts-1-i,j,xi)/prod);
    }
//recibiendo resultados de los demas
  for(j=1;j<size;j++)
  {
 MPI_Recv(coefp,npts,MPI_FLOAT,j,tag,MPI_COMM_WORLD,&status);
   for(i=0;i<npts;i++)
    coef[i]=coef[i]+coefp[i];
  }
  fin=MPI_Wtime();
  //imprimir el resultado
  printf("\nEl polinomio de aproximacion es:\n");
  for(i=npts-1;i>=0;i--)
   if(0!=((int) (1000*coef[i])))
    if(coef[i]<0.0 || i==npts-1)
     printf("%fx^[%d]",coef[i],i);
    else
     printf("+%fx^[%d]",coef[i],i);
  printf("\ntiempo de ejecucion:%f seg\n",fin-ini);
 }
 else
 {
  MPI_Recv(&npts,1,MPI_INT,0,tag,MPI_COMM_WORLD,&status);
  MPI_Recv(xi,npts,MPI_FLOAT,0,tag,MPI_COMM_WORLD,&status);
  MPI_Recv(f,npts,MPI_FLOAT,0,tag,MPI_COMM_WORLD,&status);
  for(i=0;i<npts;i++)
   coef[i]=0.0;
  //calculando mi parte de los polinomios L[n,k]
  for(j=0;j<npts;j++)
   if(j%size==rank)
   {
    prod=1;
    for(i=0;i<npts;i++)
     if(i!=j)
```

```
      prod=prod*(xi[j]-xi[i]);
//calculando los coeficientes del polinomio
    for(i=0;i<npts;i++)
     coef[i]=coef[i]+(f[j]*coefnjk(0,npts,npts-1-i,j,xi)/prod);
   }
  //mandando mis resultados al proceso 0
  MPI_Send(coef,npts,MPI_FLOAT,0,tag,MPI_COMM_WORLD);
 }
 MPI_Finalize();
}
```

Apéndice B

Código en fortran de mínimos cuadrados sin cholesky en MPI.

```
  program hola
include 'mpif.h'
c ____vec(grado)___b(grado*grado)____
c __a(grado,grado)______ainversa(grado,grado)____
integer i,j,npts,vect(3),n,yo,nproc,ierr,tag,m
real c(10),s,a(3,3),det,b(9),detaux,ainversa(3,3),d(3)
real fc(10)
c ____iniciando MPI___________________________________
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,yo,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)
tag=99
c _____n=grado________________________________________
n=3
if(yo==0) then
write (*,*)'Cuantos puntos son:'
read *,npts
c ____introduciendo los datos_________________
do i=1,npts,1
write(*,*)'Dame el punto ',i,'-esimo'

read *,c(i)
```

```
write(*,*)'Dame f(',c(i),')='
read *,fc(i)
end do
c _____calculando los elementos <i,j>_________
do i=0,n-1,1
do j=0,n-1,1
a(i+1,j+1)=prodsklar(c,npts,i,j)
end do
end do
c ________calculando <b,f>_____________________
s=bf(d,c,fc,npts,n)
c _______calculando el determinante____________
det=0
s=determinante(a,vect,1,det,n)
write(*,*)'El determinante es:',det
c  ___________________________________________
c |______calculando la inversa_________________|
c ______mandando la matriz a los demÃ¡s__________
do i=1,nproc-1,1
       call MPI_SEND(a,n*n,MPI_REAL,i,tag,MPI_COMM_WORLD,ierr)
end do
c _______haciendo mi parte de la inversa________
do i=1,n,1
do k=1,n,1
if(MOD(((i-1)*n)+k,nproc)==0)then
do j=1,n,1
b(j)=a(j,i)
a(j,i)=0
end do
a(k,i)=1
detaux=0
s=determinante(a,vect,1,detaux,n)
ainversa(i,k)=detaux/det
c write(*,*)'El determinante es:',det
do j=1,n,1
a(j,i)=b(j)
end do
end if
end do
```

```
end do
c _______recibiendo lo de los demas_____________
do i=1,nproc-1,1
call MPI_RECV(j,1,MPI_REAL,i,tag,MPI_COMM_WORLD,status,ierr)
call MPI_RECV(b,j,MPI_REAL,i,tag,MPI_COMM_WORLD,status,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)
n=3
c write(*,*)j,n,b
l=1
do j=1,n,1
do k=1,n,1
m=mod(((j-1)*n)+k,nproc)
if(m==i) then
ainversa(j,k)=b(l)/det
l=l+1
end if
end do
end do
end do
s=multiplica(ainversa,d,c,n)
write(*,*)'______>>>El polinomio es:<<<______'
do i=1,n,1
if(c(i)<0) then
print*,' ',c(i),'x','^',i-1
end if
if(c(i)>0) then
write(*,*)'+',c(i),'x^',i-1
end if
end do
write(*,*)'_________________________________
else
n=3
call MPI_RECV(a,n*n,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
n=3
c _______haciendo mi parte de la inversa_______
l=1
call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)
do i=1,n,1
do k=1,n,1
```

```
if(MOD((i-1)*n+k,nproc)==yo)then
do j=1,n,1
ainversa(j,1)=a(j,i)
a(j,i)=0
end do
a(k,i)=1
detaux=0
s=determinante(a,vect,1,detaux,n)
b(l)=detaux
l=l+1
do j=1,n,1
a(j,i)=ainversa(j,1)
end do
end if
end do
end do
l=l-1
call MPI_SEND(l,1,MPI_INTEGER,0,tag,MPI_COMM_WORLD,ierr)
call MPI_SEND(b,l,MPI_REAL,0,tag,MPI_COMM_WORLD,ierr)
end if
c _______finalizando MPI_________________________________
call MPI_FINALIZE(ierr)
stop
end program
c        ____________________________________________________________
c       /__________funcion producto escalar_____________________/
c      /   calcula <i,j>donde                                  /
c     /   r es el conjunto de elementos y n es el num de elementos /
c    /_____________________________________________________________/
c   /_____________________________________________________________/
  real function multiplica(mat,vet,reg,n) result(e)
   integer h,k,n
   real mat(n,n),vet(n),reg(n),sum
   e=0
   do h=1,n,1
   sum=0
   do k=1,n,1
         sum=sum+(vet(k)*mat(h,k))
      end do
```

```
      reg(h)=sum
    end do
  end function multiplica
c        ____________________________________________________________
c       /___________funcion producto escalar_____________________/
c      /   calcula <i,j>donde                                  /
c     /   r es el conjunto de elementos y n es el num de elementos /
c    /_____________________________________________________________/
c   /_____________________________________________________________/
  real function prodsklar(r,n,i,j) result(e)
    integer i,j,k,n
    real r(10),a,l,m
    e=0
    do k=1,n,1
      a=r(k)
      l=pri(a,i)
      m=pri(a,j)
      e=e+l*m
    end do
   end function prodsklar
c  ____________________________________________________________
c /_____________________funcion bf_____________________________/
c      /   calcula <f,b_j>donde                                /
c     /   a es el conjunto de elementos y n es el num de elemento  /
c    /_____________________________________________________________/
c   /_____________________________________________________________/
  real function bf(reg,a,fa,npts,n) result(e)
    integer i,j,n,npts
    real reg(n),a(npts),fa(npts),sum
    e=0
    do i=0,n-1,1
      sum=0
      write(*,*)n,npts
      do j=1,npts,1
        sum=sum+(fa(j)*pri(a(j),i))
      end do
      reg(i+1)=sum
    end do
  end function bf
```

```
c      ____________________________________________________________
c     /________funcion eleva_____________________________________/
c    /  eleva s a la r donde                                     /
c   /_____________________________________________________________/
c  /_____________________________________________________________/
   real function pri(s,r) result(e)
    integer i,r
    real s
c   write(*,*)r,s
    e=1
    do i=1,r,1
      e=e*s
    end do
  end function pri
c           ____________________________________________________
c          /___________________funcion determinan_______________/
c         / calcula el determinante de a, haciendo todas las /
c        /permutaciones posibles en vec                      /
c       /  pos es la posicion que se esta llenando, sum    /
c      /es el resultato                                   /
c     /_____________________________________________/
c    /_____________________________________________/
integer recursive function determinante(a,vec,pos,sum,n)    result(e)
    real a(n,n),sum,prod
    integer n,pos,vec(n),i,j
    prod=1
    do i=1,n,1
      vec(pos)=i
      if(cheka(vec,pos,n)==0)then
        if(pos==n) then
           prod=imparidad(vec,n)
           write(*,*)
           do j=1,n,1
              prod=prod*a(j,vec(j))
           end do
           sum=sum+prod
        else
           e=determinante(a,vec,pos+1,sum,n)
        end if
```

```
      end if
    end do
  end function determinante
c            ____________________________________________________
c           /________funcion cheka______________________________/
c          / ve si vec ya tiene el elemento vec(p) antes de   /
c         / la posicion p                                    /
c        /_____________________________________________ /
c       /_____________________________________________/
  real function cheka(vec,p,n) result(e)
      integer vec(n),h,p
      e=1
c     write(*,*)p,vec(p)
      do h=1,p-1,1
      if(vec(h)==vec(p)) then
c           write(*,*)'<<',h,'>>'
        return
      end if
    end do
    e=0
  end function cheka
c        ____________________________________________________________
c       /_________funcion imparidad_________________________________/
c      /  cheka si la permutacion de indices en vec es par o impar/
c     /_____________________________________________________________/
c    /_____________________________________________________________/
    integer function imparidad(vec,n) result(e)
    integer vec(n),i,aux,j,b(n)
c   real d
    e=1
    do i=1,n,1
    b(i)=vec(i)
    end do
    do i=1,n,1
      j=0
      do while(i.ne.b(j))
        j=j+1
      end do
      if(i<j) then
```

```
       do while(j>i)
         e=e*(-1)
         b(j)=b(j-1)
         j=j-1
       end do
       b(i)=i
      else
       if(i>j)then
         do while(j<i)
           e=e*(-1)
           b(j)=b(j+1)
           j=j+1
         end do
         b(i)=i
       end if
      end if
    end do
c d=e
  end function imparidad
```

Apéndice C

Código en C de mínimos cuadrados con Cholesky en MPI.

```
 #include<stdio.h>
#include<mpi.h>
#include<math.h>
MPI_Status status;
/*********************************************************************
eleva d a la s, solo para s entero positivo
*********************************************************************/
float eleva(float d, int s)
{
if(s==0)
return(1.0);
return(d*eleva(d,s-1));
}
/*********************************************************************+
busca <b_i,b_j> con la base b={1,x, x^2,x^3...} y con los ptos en a
*********************************************************************/
float prod_scalar(float a[],int n,int i, int j)
{
int k;
float sum;
for(k=0,sum=0.0;k<n;k++)
sum=sum+(eleva(a[k],i)*eleva(a[k],j));
```

```
return(sum);
}
int main(int argc, char **argv)
{
int i,npts,grado,j,k,rank,nproc,tag=99,l;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&nproc);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if(rank==0)
{
printf("De que grado el el polinomio que se desea encontrar?   ");
scanf("%d",&grado);
printf("Cuantos numeros son?   ");
scanf("%d",&npts);
grado=grado+1;
float a[npts+1],fa[npts],zx[grado][grado],zxt[grado][grado]
  ,zxta[grado][grado],y[grado],x[grado],sum,fzx[grado]
  ,b[2*npts/nproc+2];
for(i=0;i<npts;i++)
{
printf("Dame el punto %d-esimo:",i+1);
scanf("%f",&a[i]);
printf("Dame f(%f)=",a[i]);
scanf("%f",&fa[i]);
}
///////////////enviando datos a los demas
a[npts]=grado; //pa ahorrarme un mensaje
for(i=1;i<nproc;i++)
{
MPI_Send(&npts,1,MPI_INT,i,tag,MPI_COMM_WORLD);
MPI_Send(a,npts+1,MPI_FLOAT,i,tag,MPI_COMM_WORLD);
MPI_Send(fa,npts,MPI_FLOAT,i,tag,MPI_COMM_WORLD);
}
////////////////////Mi parte
for(i=0,j=0;i<grado*grado;i=i+nproc,j++)
{
zx[i%grado][i/grado]=prod_scalar(a,npts,i%grado,i/grado);
zx[i/grado][i%grado]=zx[i%grado][i/grado];
}
```

```
for(i=0;i<grado;i++)
{
for(sum=0,j=0;j<npts;j++)
sum=sum+(fa[j]*eleva(a[j],i));
fzx[i]=sum;
}
////////////////////////////Recibiendo la parte de lops demas
for(i=1;i<nproc;i++)
{
MPI_Recv(&j,1,MPI_INT,i,tag,MPI_COMM_WORLD,&status);
MPI_Recv(b,j,MPI_FLOAT,i,tag,MPI_COMM_WORLD,&status);
//almacenando los datos <bj,bi>
for(l=i,k=0;l<grado*grado;l=l+nproc)
{
zx[l%grado][l/grado]=b[k];
zx[l/grado][l%grado]=b[k];
k++;
}
//almacenando los datos <f,bj>
for(l=i;l<grado;l=l+nproc,k++)
fzx[l]=b[k];
}
///////resolviendo el sistema de ecuaciones
///mandando a todos los porcesos la matriz a
for(i=1;i<nproc;i++)
MPI_Send(zx,grado*grado,MPI_FLOAT,i,tag,MPI_COMM_WORLD);
zxt[0][0]=sqrt(zx[0][0]);
for(i=0;i<grado;i++)
{
//mandando a los procesos la parte que se a
//calculado de la matriz triangular T
for(j=1;j<nproc;j++)
MPI_Send(zxt,grado*grado,MPI_FLOAT,j,tag,MPI_COMM_WORLD);
for(j=i+1;j<grado;j=j+nproc)
{
for(k=0,sum=0;k<i;k++)
sum=sum+zxt[i][k]*zxt[j][k];
zxt[j][i]=(1/zxt[i][i])*(zx[i][j]-sum);
zxt[i][j]=zxt[j][i];
```

```
}
///recibir los datos de los procesos
for(j=1;j<nproc;j++)
{
MPI_Recv(zxta,grado*grado,MPI_INT,j,tag,MPI_COMM_WORLD,&status);
//almacenado datos
for(k=i+1+j;k<grado;k=k+nproc)
{
zxt[k][i]=zxta[k][i];
zxt[i][k]=zxt[k][i];
}
}
//calculando los elementos diagonal de la matriz triangular T
if(i+1<grado)
{
for(sum=0,k=0;k<i+1;k++)
sum=sum+zxt[i+1][k]*zxt[i+1][k];
zxt[i+1][i+1]=sqrt(zx[i+1][i+1]-sum);
}
}
y[0]=fzx[0]/zxt[0][0];
for(i=1;i<grado;i++)
{
for(k=0,sum=0;k<i;k++)
sum=sum+zxt[i][k]*y[k];
y[i]=(1/zxt[i][i])*(fzx[i]-sum);
printf("y=%f\n",y[i]);
}
x[grado-1]=y[grado-1]/zxt[grado-1][grado-1];
printf("x[%d]=%f\n",grado-1,x[grado-1]);
for(i=grado-2;i>=0;i--)
{
for(k=i+1,sum=0;k<grado;k++)
sum=sum+zxt[i][k]*x[k];
x[i]=(1/zxt[i][i])*(y[i]-sum);
printf("x[%d]=%f\n",i,x[i]);
}
printf("\nEl polinomio es:");
for(i=0;i<grado;i++)
```

```
if(x[i]>0)
printf(" + %fx^%d",x[i],i);
else if(x[i]<0)
printf("%fx^%d",x[i],i);
printf("\n");
}
else
{

MPI_Recv(&npts,1,MPI_INT,0,tag,MPI_COMM_WORLD,&status);
float a[npts/nproc+1],b[2*npts/nproc+2],fa[npts/nproc+1],sum;
MPI_Recv(a,npts+1,MPI_FLOAT,0,tag,MPI_COMM_WORLD,&status);
MPI_Recv(fa,npts,MPI_FLOAT,0,tag,MPI_COMM_WORLD,&status);
grado=a[npts];
for(i=rank,j=0;i<grado*grado;i=i+nproc)
{
b[j]=prod_scalar(a,npts,i%grado,i/grado);
j++;
}
for(i=rank;i<grado;i=i+nproc)
{
for(sum=0,k=0;k<npts;k++)
sum=sum+(fa[k]*eleva(a[k],i));
b[j]=sum;
j++;
}
MPI_Send(&j,1,MPI_INT,0,tag,MPI_COMM_WORLD);
MPI_Send(b,j,MPI_FLOAT,0,tag,MPI_COMM_WORLD);
float zx[grado][grado],zxt[grado][grado];
MPI_Recv(zx,grado*grado,MPI_FLOAT,0,tag,MPI_COMM_WORLD,&status);
for(i=0;i<grado;i++)
{
MPI_Recv(zxt,grado*grado,MPI_FLOAT,0,tag,MPI_COMM_WORLD,&status);
for(j=i+1+rank;j<grado;j=j+nproc)
{
for(k=0,sum=0;k<i;k++)
sum=sum+zxt[i][k]*zxt[j][k];
zxt[j][i]=(1/zxt[i][i])*(zx[i][j]-sum);
zxt[i][j]=zxt[j][i];
```

```
}
MPI_Send(zxt,grado*grado,MPI_FLOAT,0,tag,MPI_COMM_WORLD);
}
}
}
```

Bibliografía

[1] Robert Sedgewick, *Algorithms in C*, Brown University, Addison Wesley.

[2] *http://es.wikipedia.org/wiki/Gene_Amdahl*

[3] *http://es.wikipedia.org/wiki/IBM.*

[4] *http://es.wikipedia.org/wiki/Digital_Equipment_Corporation*

[5] Andrew D. Birrell and Bruce Jay Nelson,*Implementing Remote Procedure Calls*, ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984, Xerox Palo Alto Research Center.

[6] MARIO CÉSAR SUÁREZ ARRIAGA Y JAIME CABRERA, *CONSTRUCCIÓN DE UNA MICRO SUPER COMPUTADORA TIPO CLÚSTER PARA SIMULAR NUMÉRICAMENTE SISTEMAS COMPLEJOS*, FCFM–Agosto 2007.

[7] MARIO CÉSAR SUÁREZ ARRIAGA,*CONSTRUCCIÓN DE UN MODELO MATEMÁTICO PARA LA SIMULACIÓN NUMÉRICA DE RESERVORIOS MULTIPOROSOS. Parte 1: ACUÍFEROS*, 2002.

[8] Ricardo Dario Cuevas Landeros, *Clustering para el procesamiento matemático*, Tesis de ingeniería, Facultad de Ciencias de la Universidad Católica de Temuco, Chile 2006.

[9] *http://howto.ipng.be/Mosix–HOWTO*

[10] Beatriz Otero Calviño, José María Cela Espín,*Estrategias de Descomposición en Dominios para entornos Grid*, tesis doctoral, Universidad Politécnica de Cataluña, Departamento de Arquitectura de Computadores.

[11] Pablo Listingart, *Computación de alto rendimiento en el estudio de reacciones químicas en solución a través de métodos híbridos clásico–cuánticos*,Universidad de Buenos Aires Facultad de Ciencias Exactas y Naturales.

[12] Gene M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, IBM Sunnyvale, California, AFIPS spring joint computer conference, 1967.

[13] Bernd Jähne, *Digital Image Processing*, 5th revised and extended edition, Springer.

[14] Animación y Simulación de Algoritmos Paralelos de Exploración de Grafos, José Fco. Cachairo González, Manuel Díaz Rodríguez, Antonio Vallecillo Moreno, Departamento de Lenguajes y Ciencias de la Computación, E.T.S.I. Informática. Universidad de Málaga.

[15] Andrew S. Tanenbaum,*Sistemas Operativos Distribuidos*, Hije Universiteit Amsterdam The Netherlands, primera edición, Prentice Hall Hispanoamericana S. A.

[16] Andrew S. Tanenbaum,*Sistemas Operativos Distribuidos* Hije Universiteit Amsterdam The Netherlands, primera edición, Prentice Hall Hispanoamericana S. A.

[17] José Castillo, Ricardo Benjamín Olvera, *Implementación de un clúster OpenMosix para el cómputo científico en el instituto de ingeniería,*, tesis de ingeniería, Universidad Autónoma de México, Facultad de Ingeniería F.E.S. Aragón , México 2006.

[18] *REINA DE LA VELOCIDAD*, montaje de clústers de alto rendimiento con OpenSSI, LINUX–MAGAZINE número 36.

[19] Dávila Quintana Sergio, Pila Rodríguez Rubén,*Comparativa de Clusters SSI,*, Universidad Europea de Madrid, Villaviciosa de Odón, junio de 2007.

[20] Andrew S. Tanenbaum, Albert S. Woodhull, *sistemas Operativos, diseño e implementación*, segunda edición, Pretice Hall.

[21] Magnus Bruaset, Aslak Tveito, *Numerical Solution of Partial Differential Equations on Parallel Computers*, Lecture Notes Computational Science and Engineering 51, Editorial Board.

[22] *http://www.epm.ornl.gov/pvm/pvm_home.html*

[23] Rolf Rabenseifner, Georg Hager, Gabriele Jost and Rainer Keller,*Hybrid MPI and OpemMP Parallel programming*,Lecture Notes in Computer Science, Springer,High Performance Computing Center Stuttgart (HLRS), Department Parallel Computing, Stuttgart, University of Erlangen–Nuremberg, Erlangen, GermanyUniversity of Erlangen–Nuremberg, Erlangen,Sun Microsystems, Germany, 2006.

[24] Rolf Rabenseifner, Georg Hager and Gabriele Jost*Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi–core SMP Nodes*, Erlangen Regional Computing Center (RRZE), German, Texas Advanced Computing Center (TACC), Austin, TX, High Performance Computing Center Stuttgart (HLRS), Germany.

[25] *MPI: A Message–Passing Interface Standard, Version 2.1*, Message Passing Interface Forum, University of Tennessee, Knoxville, Tennessee, June 23, 2008.

[26] *http://es.wikipedia.org/wiki/Thomas_John_Watson*

[27] *http://ccpd.ciens.ucv.ve/wiki/index.php/MPI*

[28] *http://www1.worldlingo.com/ma/enwiki/es/nCUBE*.

[29] Rolf hempeland fralk Zimmermann,*On the automatic PARMACS–to–MPI transformation in application programs*, GMD –German National Research Center for Information Technology, P.O Box 1316, D–53734Â St.Augustin, Germany.

[30] *http://my.safaribooksonline.com/0596005709/highperlinuxc–CHP–9–SECT–4*

[31] J.J. Dongarra, R. Hempel, A.J.G. Hey, and D.W. Walker. *A proposal for a user–level, message passing interface in a distributed memory model*. Tech. Rep. TM–12231, Oak Ridge National Laboratory, February 1993.

[32] *http://hpff.rice.edu/*

[33] *http://www.mpi–forum.org*

[34] http://sc08.supercomputing.org/?pg=history.html

[35] *MPI 2 Journal of Development*, Message Passing Interface Forum, July 18, 1997.

[36] Francisco Hidrobo, Herbert Hoeger, *Introducción a MPI (Message Passing Interface)*,Centro Nacional De Cálculo Científico, Universidad de Los Andes (CeCalCULA), Mérida–Venezuela.

[37] *http://www.mcs.anl.gov/research/projects/mpich2/*

[38] *http://www.lam–mpi.org*

[39] *LAM/MPI User's Guide*, Version 7.1.4, The LAM/MPI Team Open Systems Lab, The Trustees of Indiana University, University of Notre Dame, The Ohio State University, July 24, 2007.

[40] *CHIMP and PUL: Support for Portable Parallel Computing*, R. A. A. Bruce, S. Chapple, N. B. MacDonald, A. S. Trew, Edinburgh Parallel Computing Centre, The University of Edinburgh, James Clerk Maxwell, Building, The Kingâs Buildings, Mayfield Road, Edinburgh, EH9 3JZ, UK. March 1993, Presented at the Fourth Annual Conference of the Meiko User Society, University of Southampton, 15thâ16th April 1993.

[41] *Recent Advances in Parallel Virtual Machine and Menssage Passing Inteface*, 16th European PVM/MPI Usser'Group Meeting, Espoo, Finland, September 2009 Proceedings, Springer.

[42] *http://www.mcs.anl.gov/research/projects/mpi/mpich1/mpich-nt/*

[43] G.M. Shipman, T.S. Woodall, A.B Maccabe, *Infiniband Scalability in OPENMPI*,Los Alamos National Laboratory, Scalable Systems Laboratory Computer Science Department.

[44] Cluster sencillo con MPI, *Asalto al trasero*, LINUX–MAGAZINE, número 38, WWW.LINUX–MAGAZINE.ES.

[45] *http://ftp.riken.go.jp/Linux/*

[46] *http://www.debian.org*

[47] Terry Collings and Kurt Wall,*Red Hat Linux Networking and System Administration*, Copyright 2002 Hungry Minds, Inc.

[48] Himanshu Dwivedi,*mplementing SSH Strategies for Optimizing the Secure Shell*, Wiley Publishing, Inc., Indianapolis, Indiana.

[49] George Em Karniadakis and Robert M. Kirby II,*arallel Scientific Computing in C++ and MPI, A seamless approach to parallel algorithms and their implementation*, Cambridge University Press.

[50] Jun Nakano, Yukiya Aoyama,*RS/6000 SP: Practical MPI Programming*,IBM International Technical Support Organization, August 1999.

[51] Gilbert Strang, *Algebra Lineal y sus aplicaciones*.

[52] Gregorio A. Caballero,*Algebra lineal*, Universidad Distrital de Francisco José de Caldas Bogotá, Colombia, Fondo Educativo Interamericano.

[53] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Revest, Cliffort Stein, *INTRODUCTION TO ALGORITHMS*, segunda edición,The MIT Press, Cambridge, Massachusetts, London, England.

[54] Mario César Suárez Arriaga,*Notas de métodos numéricos*, Facultad de Ciencias Físico Matemáticas, UMSNH.

[55] Antonio Nieves, Federico C. Domínguez, *Métodos Numéricos aplicados a la ingeniería*, , segunda edición, CECSA.

[56] Goldstein, A. A. *Convex programmining Hilbert space*, Bull. Am. Math.Soc. 70, 709–710 (1964).

[57] Stanley I. Grossman, *Aplicaciones de Álgebra Lineal*, Grupo Editorial Iberoamérica.

[58] *Numerical Linear Algebra*, Lloyd N. Thefethen, Cornel University, ithaca, New York, David Bau, III, Microsoft Corporation, Redmond, Washington.

[59] Mario César Suárez Arriaga, *Matemáticas aplicadas a la ingenieria de yaciminetos geotermicos Curso I: teoria y metodos de resolución de EDP*, texto publicado por la CFE para curso internacional, Morelia Mich.