APLICACIÓN DE LA COMPUTACIÓN PARALELA EN MODELOS DE ILUMINACIÓN EN COMPUTACIÓN GRÁFICA

Autor: Núñez Zuleta, José Vicente

Profesor Guía: Hernandez, Domingo

PASANTÍAS DE INVESTIGACIÓN

MÉRIDA. JULIO 1997

TABLA DE CONTENIDO

copyright 5

Introducción 1

Antecedentes 2

planteamiento del problema: 2

objetivos 2

alcance 3

revisión bibliográfica sobre conceptos de paralelismo 4

Tendencias en las aplicaciones 4

Tendencias en el diseño de computadoras 4

Tendencias en la interconección de redes 5

Modelos de maquinas paralelas 5

El MULTICOMPUTADOR: *5*

Múltiples Instrucciones, Múltiples Datos (MIMD, Multiple Instruction Multiple Data): *6*

MIMD de memoria compartida: *7*

Una sola Instrucción, Múltiples Datos (SIMD, Single Instrucction Single Data): *7*

Sistemas Distribuidos (LAN y WAN) *7*

Modelos de programación paralela 8

tareas y canales 8

DISEÑANDO ALGORITMOS PARALELOS 11

diseño metodico *11*

Herramientas paralelas *25*

modelos de iluminación 34

Tipos de modelos de iluminación.....34

Luz ambiental.....*34*

LuZ DIFUSA.....*34*

LUZ ESPECULAR.....*36*

Paralelización del modelo de iluminación 37

Particionamiento.....37

Particionamiento utilizando descomposición por dominio: *37*

particionamiento utilizando descomposición funcional: *37*

Revisión de puntos de partcionamiento para el plan 1 y el plan 2: 38

comunicación: 38

puntos clave en los requerimientos de comunicación: *39*

Aglomeración: 39

mapeo: 39

implantación de la paralelización del modelo de iluminación 41

Plataforma paralela en cecalcula:.....41

PLAtaforma distribuida.....46

Características de cada uno de los equipos: *46*

Configuración del servidor: *47*

Configuración del cliente: *48*

tiempos de ejecución y analisis de resultados 50

conclusiones 52

referencias bibliográficas 53

anexos.....55

instalación de mpi bajo linux.....55

aplicaciones de prueba en mpi.....55

codificación del modelo de iluminación 59

archivo de constantes comunes: constantes.hpp.....*59*

rutinas para el cálculo de los valores del modelo de iluminación.....*60*

codificación del modelo secuencial.....*62*

codificación de la implementación paralela.....*64*

ILUSTRACIONES

Ilustración 1: Modelo del multicomputador 6

Ilustración 2: Modelo MIMD.....6

Ilustración 3: MIMD de memoria compartida 7

Ilustración 4: Modelo LAN 8

Ilustración 5: Modelo de Tareas y canales.....9

Ilustración 6: Sistema de moléculas.....12

Ilustración 7: Sistema de moléculas particionado.....12

Ilustración 8: Modelo de Iluminación con luz difusa.....35

Ilustración 9: Modelo de Iluminación con luz especular ..36

Ilustración 10: Descomposición funcional del problema del modelo de iluminación 37

Ilustración 11: Particionamiento funcional del modelo de iluminación clásico 37

Ilustración 12: Configuración de la SP2 presente en CecalcULA 41

Ilustración 13: Topología de Bus del ambiente distribuido46

TABLAS

Tabla 1: Latencia y ancho de banda entre MPL,MPI, PVMe, PVM (Tomado de http://www.mhpcc.edu/training/workshop/html/ibmwaw/ibmwaw.html#bech). 32

Tabla 2: Comparación entre librerías de pase de mensajes y recomendaciones (Tomado de http://www.mhpcc.edu/training/workshop/html/ibmwaw/ibmwaw.html#bech). 33

Tabla 3: Tiempos de computo y comunicación en el ambiente distribuido según la cantidad de procesadores.....50

Tabla 4: Tiempos de computo y comunicación en el ambiente paralelo según la cantidad de procesadores.....51

COPYRIGHT

Este trabajo siempre estará bajo construcción y puede contener errores.

Esta trabajo viene sin garantías de ningún tipo, y se prohibe su uso para fines comerciales.

Este documento se publica bajo la Licencia General GNU, lo que permite su libre distribución gratuita siempre que se conserve en su integridad y se retenga el copyright original.

Este trabajo tiene un copyright de 1997 por José Vicente Núñez Zuleta. Puede ser reproducido en cualquier forma y distribuido libremente mientras el fichero se mantenga intacto , incluyendo esta línea.

Si tiene dudas, adiciones, comentarios acerca de esta guia puede contactarme en la siguiente dirección:

Jose@ing.ula.ve

La última versión de este documento puede ser conseguida en:
http://www.ing.ula.ve/~jose

Todas las marcas registradas pertenecen a sus respectivos autores.

INTRODUCCIÓN

Para nadie es un secreto que hoy en día la computación gráfica ha avanzado muchísimo en cuanto al realismo que puede tener una imagen generada por computador. Sin embargo, pocos se han preguntado como se realiza este proceso y cuanto tiempo toma hacerlo.

Cuando se le agrega color a un objeto tridimensional, primero se llama a *un modelo de sombreado*, el cual me dice como se van a aplicar los colores sobre la superficie del objeto. Sin embargo, hacer esto pixel por pixel es muy costoso, por lo que el modelo de sombreado llama a un *modelo de iluminación*, el cual determina la intensidad y color de ciertos pixeles, mientras que los valores de color de los demás se determinan por interpolación. Dependiendo de la complejidad de la imagen a ser coloreada, este proceso puede tardar desde minutos hasta días (si no meses) utilizando los métodos convencionales.

Si nos damos cuenta, este es un problema en donde aplicamos un mismo conjunto de operaciones de cálculo sobre grupos diferentes de datos para obtener los valores de color deseado.

Nuestro problema es entonces el siguiente: ¿Existe alguna manera de realizar esas operaciones cálculo de manera simultanea (paralela) para reducir el tiempo de cómputo?

Nuestra meta es utilizar los lenguajes de programación, compiladores y sistemas operativos para ejecutar **más rápido** nuestro algoritmo.

El éxito de las ciencias computacionales en el modelado del mundo real, ha hecho que se disparen las necesidades de mayor poder computacional a un bajo costo. Los científicos siempre están buscando probar los limites de sus teorías, usando la computación de alto nivel para simular sistemas más realísticos con mayor detalle. La computación paralela ofrece una manera práctica y económica de atacar estos problemas.

¿ Porqué debemos utilizar múltiples procesadores para hacer esto?

- Un solo procesador tiene límites físicos en cuanto a la velocidad con que este puede mover los datos (Velocidad de la luz)

- Se vuelve caro hacer un sólo procesador especializado más rápido, pero los procesadores de uso general *relativamente* rápidos se hacen cada día más baratos (Por esta razón podemos utilizar varios procesadores no especializados en paralelo, en vez de uno caro).

- Aparece entonces la pregunta de como correr algoritmos en el hardware disponible. Aun si tenemos un algoritmo muy paralelo y un hardware muy paralelo, eso no significa que sean compatibles.

- Algunos compiladores ofrecen directivas las cuales pueden ser insertadas en nuestro código secuencial para permitirle tomar ventajas de ejecutarse en múltiples procesadores (Por ejemplo el High Performance Fortran permiten eso). Desgraciadamente en muchos casos, el algoritmo serial necesita ser reescrito para tomar una verdadera ventaja del nuevo entorno. La idea aquí no es paralelizar completamente el algoritmo secuencial sino *solamente la parte de computo intensivo* de mi algoritmo secuencial.

El paralelismo es visto muchas veces como un área rara y exótica de la computación. Un estudio de las tendencias en aplicaciones, arquitectura de las computadoras, interconexión de redes muestra que ese punto de vista no es sostenible.

El paralelismo no es nuevo. Hace más de 100 años a Charles Babbage se le ocurrió la idea de realizar cálculos en paralelo cuando diseñaba el motor analítico (Sólo que la tecnología no se lo permitía). Luego Jhon von Neumann (En los años 50) ideo un tipo de máquina conceptualmente simple: Una sola unidad de procesamiento conectada a una unidad de almacenamiento (La cual sólo podía procesar un evento a la vez). Este modelo era barato pero sus componentes ofrecían poco rendimiento. Fue entre los años 60-70 que se desarrollaron componentes de estado sólido y software con conocimiento de concurrencia los que permitieron a la computación paralela pasar de ser una curiosidad a ser una realidad.

ANTECEDENTES

El laboratorio de Computación Avanzada Multimedia y Videoconferencia (LCAMV) de la División de Postgrado de la Universidad de Los Andes está creando actualmente una unidad de Visualización Científica. Por lo general las aplicaciones de esta área requieren alto cómputo en el procesamiento de los modelos de iluminación que se utilizan para el despliegue de las escenas que se desean presentar. Debido a el alto volumen de datos y cálculos los expertos en el área han decidido utilizar técnicas y herramientas de programación paralela y distribuida a fin de disminuir el tiempo de cálculo de los valores de luminosidad de los pixeles que se van a presentar en la pantalla del computador.

Por esta razón el LCAMV ha decidido probar las técnicas de programación distribuida y paralela a fin de ser aplicadas en esta área.

PLANTEAMIENTO DEL PROBLEMA:

Disminuir el tiempo de computo de un modelo de iluminación clásico utilizando para ello el paralelismo como herramienta, ya sea tanto en un ambiente paralelo como en un ambiente distribuido.

OBJETIVOS

- Involucrar al estudiante en el conocimiento de técnicas de programación paralela y modelos de iluminación.
- Estudiar las distintas técnicas para programación paralelas que existen actualmente.
- Estudiar las características de los distintos lenguajes y compiladores que puedan ser utilizados en programación de sistemas paralelizables.
- Estudiar los distintos modelos de iluminación clásicos que se utilizan en la computación gráfica.
- Estudiar las fortalezas y debilidades de cada uno de los lenguajes.
- Realizar la programación de un pequeño problema en los lenguajes que permitan la paralelización disponibles en el Centro de Cálculo de la ULA (CeCalCULA)

ALCANCE

Una vez adquiridos los conocimientos básicos en el área de computación paralela y modelos de iluminación en computación gráfica se paralelizará el modelo de iluminación clásico tanto en el ambiente distribuido como en el ambiente paralelo.

REVISIÓN BIBLIOGRÁFICA SOBRE CONCEPTOS DE PARALELISMO

TENDENCIAS EN LAS APLICACIONES

A medida que las computadoras se vuelven más rápidas, se vuelve tentador suponer que ellas "serán" lo suficientemente rápidas como para saciar el apetito computacional existente. La historia demuestra que este no es el caso ya que cuando una tecnología satisface una demanda dada, nuevas aplicaciones basadas en esa misma tecnología saturarán su uso.

Tradicionalmente, desarrollos en los campos más avanzados de la computación han sido motivados por simulaciones numéricas de sistemas complejos como el tiempo, clima, dispositivos mecánicos, circuitos eléctricos, procedimientos de manufactura y reacciones químicas. Hoy en día las fuerzas más significativas que manejan el desarrollo de computadoras rápidas son las aplicaciones comerciales que requieren que un computador sea capaz de procesar largas cantidades de datos de manera sofisticada. Estas aplicaciones incluyen videoconferencia, elementos de redes cooperativos, sistemas asistidos por computador en medicina, bases de datos paralelas usadas para soporte de decisiones y gráficos avanzados y realidad virtual, particularmente en la industria del entretenimiento.

En resumen, la necesidad de computadoras más rápidas por demandas de aplicaciones de uso intensivo de datos en el comercio y aplicaciones de uso intensivo de computación en las ciencias e ingeniería se ha incrementado. Además los requerimientos de los campos científicos y comerciales se están fusionando: en la medida que las aplicaciones científicas y de ingeniería se vuelven más intensivas en el uso de datos y las aplicaciones comerciales efectúan cálculos más sofisticados.

TENDENCIAS EN EL DISEÑO DE COMPUTADORAS

El desempeño de las computadoras más rápidas ha crecido exponencialmente desde 1945 hasta el presente, con un factor promedio de 10 cada 5 años. Mientras que la primera computadora efectuaba unos pocos de cientos de operaciones de punto flotante por segundo, las computadoras paralelas de mediados de los 90 alcanzaron decenas de billones de operaciones por segundo. Tendencias similares pudieron ser observadas en computadoras de bajo nivel en diferentes eras: las calculadoras, los computadoras personales y las estaciones de trabajo.

Se sugiere poco que este crecimiento no continuará. Entonces, las arquitecturas usadas para mantener este rendimiento computacional cambiarán de secuencial a paralela.

El desempeño de una computadora depende directamente del tiempo requerido para realizar una operación básica y el número de esas operaciones que pueden ser ejecutadas concurrentemente (Al mismo tiempo). El tiempo para desarrollar una operación básica está limitado por el ciclo de reloj del procesador (Tiempo requerido para ejecutar la operación más primitiva). Sin embargo, los ciclos de reloj parecen estar decrementándose lentamente y parece ser que nos aproximamos al límite de la velocidad de la luz cada vez más. No podemos depender de procesadores más rápidos para proveer un incremento de desempeño computacional.

Para sobrepasar esas limitaciones, el diseñador podría tratar de utilizar la concurrencia interna de un chip, por ejemplo, para operar de manera simultánea sobre todos los 64 bits de 2 números

que van a ser multiplicados. Existe un resultado fundamental en compleja teoría de la *integración a gran escala (VLSI, Very Large Scale Integration) dice que esta integración es costosa.* Es entonces cuando los diseñadores de computadoras utilizan una variedad de técnicas para superar esas limitaciones reduciendo el número de componentes en el computador (y por ende su costo). Esto trae como consecuencia que se pueden incluir más procesadores dentro de una tarjeta madre, lo cual facilita la implementación de arquitecturas paralelas.

TENDENCIAS EN LA INTERCONECCIÓN DE REDES

Otra tendencia que está cambiando el aspecto de la computación es el enorme incremento de las capacidades de las redes que conectan computadoras. No hace mucho tiempo, las redes de alta velocidad corrían a 1.5 Mbits por segundo, pero para el final de los noventa ese ancho de banda sería un poco mayor que los 1000 Mbits por segundo. Se esperan mejoras significativas en la confiabilidad. Esas tendencias hacen posible el desarrollar aplicaciones que usen recursos distribuidos físicamente como si fueran parte de la misma computadora. Una aplicación típica de este tipo podría utilizar procesadores de computadoras remotas para realizar los cálculos requeridos en la visualización de cierto tipo de problema a medida que este se muestra en tiempo real en un sólo computador (La computación paralela no es un subcampo de la computación distribuida. La computación distribuida se enfrenta a problemas como confiabilidad, heterogeneidad y seguridad los cuales son considerados como tangenciales en la computación paralela; Puede verse que ambos mundos están convergiendo).

MODELOS DE MAQUINAS PARALELAS

La rápida entrada de las computadoras en los ámbitos de la ciencia, comercio y educación se debe mucho a la rápida estandarización de un sólo tipo de máquina: La computadora de *Von Neumann* (Una computadora de Neumann no es más que una *Unidad Central de Procesamiento* conectada a *una unidad de almacenamiento*). En esta maquina el CPU ejecuta un programa almacenado que especifica una secuencia de operaciones de escritura o lectura en la memoria. Este modelo de computadora, más o menos robusto, es el que ha permitido programar para un tipo de computadora "X" en vez de un tipo de máquina en particular.

En el estudio de la computación paralela se hizo necesario identificar una serie de modelos los cuales tenían que ser iguales de robustos, sencillos (Para mejorar su programación) y realistas (Para poder implementarlos en la vida real). Estas necesidades condujeron a los siguientes modelos:

EL MULTICOMPUTADOR:

En este modelo de maquina cada nodo es una maquina de von Neumann las cuales se conectan por una red interconectada (O memoria compartida), Cada nodo ejecuta su propio programa. Este programa puede acceder a la memoria local y puede enviar y recibir mensajes por la red (Los mensajes son usados para comunicarse con otras computadoras, o de manera equivalente, para leer y escribir en las memorias remotas. En esta maquina se

desea que el programa local escriba y lea la memoria local la mayor parte del tiempo en vez de enviar y recibir datos de la memoria remota (A esto se le llama localidad), ya que es más costosa utilizar el recurso remoto (Debido a la distancia a la que este, el costo de iniciar una comunicación, el ancho de banda disponible, etc.). El costo de la comunicación no depende de la distancia entre los nodos ni del trafico de la red, más si depende de la longitud del mensaje (Esto es de manera idealizada).

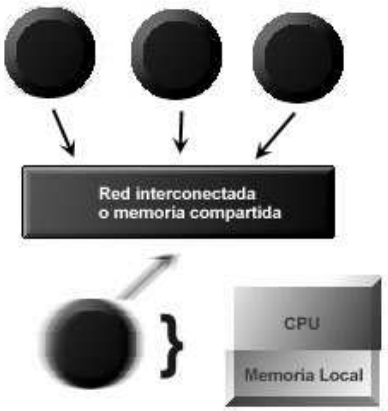


Ilustración 1: Modelo del multicomputador

MÚLTIPLES INSTRUCCIONES, MÚLTIPLES DATOS (MIMD, MULTIPLE INSTRUCTION MULTIPLE DATA):

Un Multicomputador es muy similar a los que frecuentemente se llama un MIMD de memoria distribuida. MIMD significa que cada procesador puede ejecutar un flujo separado de instrucciones con sus datos locales; La memoria distribuida significa que la memoria está distribuida entre los procesadores en vez de estar en un lugar central. La diferencia principal entre un multicomputador y un MIMD de memoria distribuida radica en que en este último el costo de enviar un mensaje entre 2 nodos puede ser no independiente de la localización del nodo y del tráfico en la red.

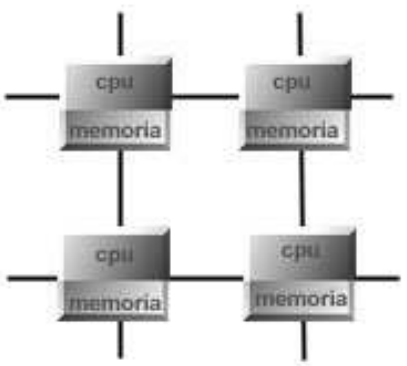


Ilustración 2: Modelo MIMD

Ejemplos de estas maquinas incluyen *IBM SP, Intel Paragon, Thinking Machines CM5, Cray T3D* y *nCUBE*.

MIMD DE MEMORIA COMPARTIDA:

Aquí todos los procesadores comparten el acceso a una memoria compartida típicamente vía bus o por una jerarquía de buses. Para optimizar el rendimiento, cada maquina emplea una *memoria cache* para guardar los datos más usados con más frecuencia (Con lo cual se reduce la cantidad de veces que se accede la memoria compartida).

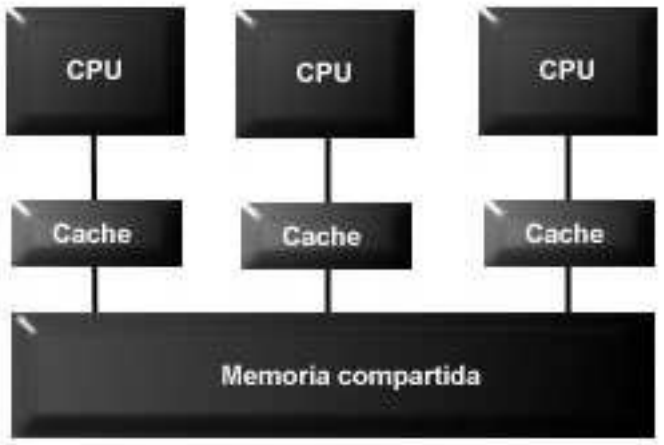


Ilustración 3: MIMD de memoria compartida

UNA SOLA INSTRUCCIÓN, MÚLTIPLES DATOS (SIMD, SINGLE INSTRUCCTION SINGLE DATA):

En estas máquinas todos lo procesadores ejecutan el mismo flujo de instrucciones en una pieza diferente de datos. Este enfoque reduce la complejidad del hardware y software pero es apropiada sólo para problemas caracterizados por un alto grado de regularidad, como el procesamiento de imágenes y ciertas simulaciones numéricas. Los algoritmos de multicomputación no pueden ser ejecutados con éxito en máquinas SIMD. El *MasPar MP* es un ejemplo de este tipo de máquina.

SISTEMAS DISTRIBUIDOS (LAN Y WAN)

Finalmente hay 2 clases de sistemas computarizados que son usados como computadoras paralelos: Las redes de área local (**LAN**, las cuales las computadoras están conectadas unas muy cerca de otras) y las redes de área ancha (**WAN**, las cuales están distribuidas geográficamente). Este tipo de sistemas introducen aspectos relacionados como confiabilidad y seguridad; Sin embargo, pueden ser vistos como multicomputadores para fines prácticos. Las tecnologías de interconexión más utilizadas son ethernet y ATM.

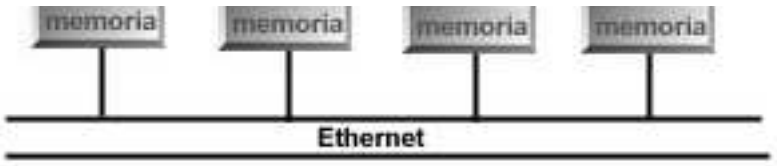


Ilustración 4: Modelo LAN

MODELOS DE PROGRAMACIÓN PARALELA

El modelo de la máquina de von Neumann asume un procesador capaz de ejecutar una secuencia de instrucciones. Es prohibitivo muchas veces tratar de trabajar a bajo nivel ya que la cantidad de instrucciones lógicas a controlar es muy grande, por lo que se usan técnicas de diseño modular en las cuales los programas complejos son construidos a partir de componentes más simples (Y los componentes son estructurados en términos de abstracciones de alto nivel con estructuras de datos, lazos interactivos y procedimientos). Abstracciones como procedimientos explotan la modularidad de fácil manera al permitirle a los objetos ser manipulados sin importarles su estructura interna.

De esta manera, lenguajes de alto nivel como *Pascal, C , Ada y Fortran*, permiten que diseños expresados de manera lógica puedan ser llevados automáticamente a código ejecutable.

La computación paralela introduce fuentes adicionales de complejidad: Si tuviéramos que programar a bajo nivel no solo el numero de instrucciones ejecutadas se elevaría, pero también se necesitaría manejar explícitamente millones de interacciones *entre* procesadores. Aquí la *abstracción* y la *modularidad* juegan un papel importante (Al igual que en la programación secuencial).

Antes de ver algunos modelos de programación paralela veamos los siguientes conceptos:

- Concurrencia : Se refiere a la habilidad de llevar varias acciones de manera simultanea (Esto es esencial si un programa se va a ejecutar en varios procesadores).
- Escalabilidad : Es la capacidad de incrementar el uso de procesadores a medida que estos son agregados al sistema.
- Localidad : Es el radio más alto de accesos a la memoria local con respecto a los accesos de memoria remota (Comunicación).
- Modularidad : Es la descomposición de entidades complejas en componentes más simples.

TAREAS Y CANALES

- Un programa paralelo consiste en una o más tareas. Las tareas se ejecutan de manera concurrente. El número de tareas se puede variar durante la ejecución de un programa.
- Una tarea encapsula un programa secuencial y la memoria local (En efecto, es una maquina virtual de von Neumann). Además una serie de puertos de entrada (*inports*) y salida (*outports*) definen su interfaz con el entorno.
- Una tarea puede ejecutar 4 acciones básicas además de leer y escribir en su memoria local : Enviar mensajes a sus puertos de salida, recibe mensajes de sus puertos de entrada, crea y termina tareas.

- Una operación de envío es asincrónica: termina inmediatamente (Los mensajes quedan en una cola); En cambio la operación de recepción es sincrónica: causa que la ejecución de la tarea se bloquee hasta que haya un mensaje disponible.

- Los puertos de entrada / salida se pueden conectar por colas de mensajes llamados canales. Los canales pueden ser creados, borrados y las referencias a los canales (puertos) pueden ser incluidos en los mensajes, así que la conectividad varía dinámicamente.

- Las tareas pueden ser mapeadas a procesadores físicos de varias maneras ; El mapeo empleado no afecta la semántica del programa. En particular, múltiples tareas pueden ser mapeadas en un sólo procesador.

  La siguiente figura ilustra un poco los conceptos anteriores:

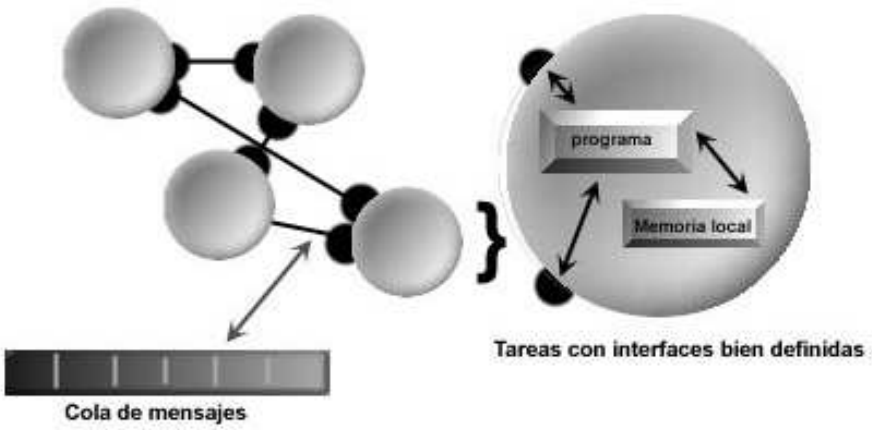


Ilustración 5: Modelo de Tareas y canales

Las tareas están representadas como círculos, los canales son los arcos que unen a los nodos en el grafo. Puede verse en detalle que cada tarea es un *CPU + Dispositivo de almacenamiento.*

Este modelo representa las siguientes propiedades:

- Desempeño : Las abstracciones de programación como procedimientos y estructuras de datos son efectivas porque ellas pueden ser manejadas de manera simple y eficiente por la computadora de von Neumann. La tarea y el canal tienen el mismo efecto similar de mapeo directo en el multicomputador. Una tarea representa una pieza de código que puede ser ejecutada secuencialmente en un solo procesador. Si las tareas que comparten un canal son mapeadas a diferentes procesadores, la conexión del canal es implementada como una comunicación entre procesos. Si ellas son mapeadas dentro del mismo procesador entonces otro método más eficiente puede ser usado (Es decir aquí el paradigma de paralelismo no es útil).

- Independencia de Mapeo: Debido a que las tareas interactúan usando el mismo mecanismo (Canales) independientemente de la ubicación de la tarea, el resultado calculado por un programa no dependen de donde se ejecuta la tarea. Además, los

algoritmos pueden ser diseñados sin importar el número de procesadores en los cuales se ejecutarán. De hecho, los algoritmos son diseñados frecuentemente para crear más tareas que procesadores (Esta es una forma de obtener *escalabilidad*): A medida que el número de procesadores aumenta, el número de tareas por procesador se reduce pero el algoritmo no es modificado.)

- Modularidad :En el diseño de programas modulares, varios componentes de un programa son desarrollados separadamente, como módulos independientes, y pueden ser combinados entonces para obtener un programa completo. La interacción de módulos está restringida a interfaces bien definidas (de esta manera se puede cambiar la implementación del modulo sin modificar otros componentes), Cuando se aplica correctamente se reduce la complejidad del problema y se facilita la reutilización del código.

- Determinismo : Un algoritmo o programa es determinístico si la ejecución de una entrada particular siempre produce la misma salida. Es no determinístico si múltiples ejecuciones con la misma entrada pueden producir diferentes salidas. Aunque lo no determinístico es útil algunas veces y debe ser soportado por cualquier implementación, un modelo de programación paralela que hace fácil escribir programas determinísticos es altamente deseable. Los programas determinísticos son más fáciles de entender y solo hace falta una secuencia de ejecución para hacer depuraciones.

- Pase de mensajes: El pase de mensajes es probablemente el modelo de programación más ampliamente utilizado hoy en día. Programas de pase de mensajes, como los programas de tareas/canales, crean múltiples tareas, con cada tarea encapsulando datos locales. Cada tarea es identificada por un nombre único, y las tareas interactuan enviando y recibiendo mensajes desde y para las tareas con nombres (Como se puede ver, el pase de mensajes es sólo una pequeña variación de tareas y canales).

  El modelo de pase de mensajes no evita la creación dinámica de tareas, la ejecución de múltiples tareas por procesador, o la ejecución de diferentes programas por diferentes tareas. En la practica muchos sistemas de pases de mensajes crean un conjunto fijo de tareas idénticas al inicio del programa, y no permiten que las tareas sean creadas o destruidas durante la ejecución de un programa. Se dice que estos sistemas incrementan un modelo de un *solo programa múltiples datos (SMPD, Single Program Multiple Data)* debido a que cada tarea ejecuta el mismo programa pero opera en datos diferentes.

- Paralelismo de Datos: Se busca la explotación de la concurrencia que deriva de la aplicación de la misma operación a múltiples elementos de una estructura de datos, por ejemplo, "multiplique por 7 a todos los elementos de este vector". Un programa basado en el paralelismo de datos consiste en una secuencia idéntica de operaciones pero aplicada a diferentes partes de los datos. Como cada operación de cada elemento de datos puede ser tratado como una tarea independiente, la *granularidad* natural de la computación paralela es pequeña. Pero ¿Qué es granularidad?. La granularidad puede ser de 2 tipos: Grano pequeño y grano grueso; En el caso de grano fino la duración es corta entre las comunicaciones y permite una mayor oportunidad de balancear las cargas con pocas demoras mientras se espera que la última pieza finalice. Aquí hay muchas tareas pequeñas que se comunican entre sí. Con grano grueso hay mayor duración entre las comunicaciones, permite menos comunicación y hay que esperar más a que "terminen" de ejecutarse otros procesos.

- Memoria Compartida: En el modelo de memoria compartida, las tareas comparten un espacio común de direcciones en las cuales ellas pueden leer y escribir de manera asincrónica. Varios mecanismos como bloqueos y semáforos pueden ser usados para controlar el acceso a la memoria compartida. Una ventaja de este modelo desde el punto de vista del programador es la noción de "propiedad de datos" se pierde y por consiguiente no hay que especificar la comunicación de datos desde productores a consumidores. Este modelo puede simplificar el desarrollo de programas. Pero entonces, es más difícil manejar conceptos como localidad y es más difícil escribir programas determinísticos.

DISEÑANDO ALGORITMOS PARALELOS

El diseño de algoritmos paralelos no se reduce fácilmente a recetas simples. Requiere creatividad y se puede beneficiar de un enfoque metódico que máximize el rango de opciones consideradas, que provea mecanismos para evaluar alternativas y que reduzca el costo de rastrear malas elecciones.

DISEÑO METODICO

Este enfoque es independiente de la maquina (lo cual se deja para el final). Esta manera de ver los problemas paralelos divide el análisis del problema en 4 etapas:

1) **Particionamiento** : La computación que va a ser realizada y los datos operados por esta pueden ser descompuestos en pequeñas tareas (Se ignora la cantidad de procesos existentes y se buscan más bien oportunidades de ejecución paralela)

2) **Comunicación** : La comunicación requerida para considerar la ejecución de las tareas es determinada y se definen las estructuras de comunicación apropiadas.

3) **Aglomeración** : Se evalúan las tareas y estructuras definidas en cuanto a costos y requerimientos de desempeño y si es necesario son combinadas en tareas más grandes para mejorar cualquiera de los aspectos que marcan su rendimiento.

4) **Mapeo** : Cada tarea es asignada a un procesador de manera de satisfacer las metas de máximo uso del procesador y minimización de costos de comunicación (El mapeo puede ser determinado de manera *estática* o por algún algoritmo de *carga balanceada*).

Vamos a ver con más detalle cada una de estas etapas de estudio :

particionamiento

En esta etapa de diseño se pretende exponer las oportunidades de ejecución paralela. Aquí, se enfoca en la definición de un número grande de pequeñas tareas de manera de hacer una *descomposición en grano fino* del problema. Un buen Particionamiento divide en pequeñas piezas tanto a los cómputos asociados con el problema como a los datos sobre los que el algoritmo opera.

Hay dos métodos para hacer un buen particionamiento:

Descomposición por dominio:
Buscamos primero descomponer los datos asociados al problema en varios pedazos (Si es posible del mismo tamaño) y luego particionamos los cálculos computacionales que van a ser llevados a cabo típicamente *asociando cada operación con los datos sobre los que opera.*

Al final quedamos con un conjunto de tareas con un conjunto de datos asociados.

Ejemplo : Dinámica molecular de un sistema.

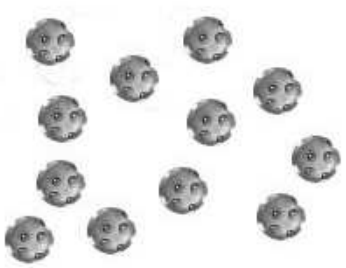


Ilustración 6: Sistema de moléculas

En nuestro problema, tenemos un sistema de N moléculas. A cada molécula se le da una posición inicial ir(0) y una velocidad vi(0) (para i= 1,2,...,N). Al realizar varios cálculos para resolver un sistema de ecuaciones, nuestro algoritmo puede determinar la posición, velocidad, y aceleración de una molécula en un momento dado, t : ri(t), vi(t), ai(t) (Para i=1,2,...,N).

Supóngase, que decidimos agrupar a las moléculas basándonos en su ubicación. Entonces el sistema podría ser dividido de la siguiente manera :

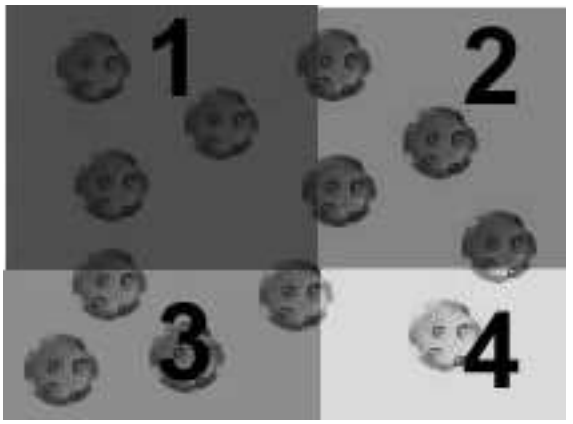


Ilustración 7: Sistema de moléculas particionado

Los datos correspondientes a las moléculas contenidas dentro de la partición1 podrían ser guardados en el primer lugar de procesamiento, aquellas que están en la partición 2 podrían ser guardadas en el procesador 2, etc. (Es decir cada lugar hace el mismo cálculo, pero sobre diferentes conjuntos de datos).

Descomposición funcional:
Representa una manera diferente y complementaria de pensar acerca de los problemas. En este enfoque nos preocupamos más en los cómputos que van a ser realizados en vez de los datos manipulados por dichos cómputos. Si somos exitosos dividiendo el cómputo en

tareas distintas, entonces procederemos a examinar los requerimientos de esas tareas. Esos requerimientos de datos pueden ser distintos en cuyo caso la partición es completa (Si por el contrario se produce solapamiento, debemos pensar en comunicación para evitar duplicidad de datos y quizás el problema sea candidato para una descomposición por dominio.

Ejemplo : Dinámica molecular de un sistema.

Utilizando este enfoque, y siguiendo con el ejemplo anterior, deberíamos determinar cuales de los cálculos pueden ser desarrollados de manera concurrente y de forma independiente de los otros. Entonces, cada conjunto de cálculos independientes pueden ser realizados de manera simultánea en diferentes lugares de procesamiento.

puntos a verificar después de particionar el PROBLEMA:

- El particionamiento realizado define al menos un orden de magnitud de tareas mayor que la cantidad de procesadores en la computadora objetivo. Si no es así, se tendrá poca flexibilidad en las etapas siguientes de diseño.
- Su partición evita cómputos y requerimientos de almacenamiento redundantes. Si no, el algoritmo resultante puede no ser escalable para tratar con problemas grandes.
- Son las tareas de tamaño comparativo. De no ser así, puede ser difícil de ubicar cantidades iguales de trabajo por cada procesador.
- Tiene la escala del número de tareas un problema de tamaño. Idealmente, un incremento en el tamaño del problema debería aumentar el número de tareas en vez del número de tareas individuales. Si este no es el caso, el algoritmo paralelo podría no ser capaz de resolver problemas grandes cuando haya más procesadores disponibles.
- Ha investigado varias alternativas de particionamiento (Descomposiciones funcionales o descomposiciones de dominio).

comunicación

La comunicación es fundamental para coordinar e integrar varias piezas separadas de trabajo.

La comunicación es necesitada para:

- Asignación de tareas
  - Envío a los sitios de procesamiento piezas de trabajo
- Transferencia e intercambio de datos
  - Envío de datos entre lugares de procesamiento
- Sincronización
  - A veces ocurre antes o conjunto cuando se produce una transferencia de datos

- También se usa para tratar de mantener la integridad de elementos de datos compartidos y prevenir condiciones de cambio

- Tolerancia a fallos

- Una barrera es un mecanismo de sincronización que fuerza a todos los procesos a esperar el último proceso.

Las tareas generadas por una partición son pensadas para ser ejecutadas de manera concurrente pero no pueden ser ejecutadas, en general, de manera independiente. El cómputo que va a ser realizado por una tarea necesitará generalmente de datos asociados con otra tarea. Los datos deben ser transferidos entre tareas para así permitir que el cómputo prosiga. Este flujo de información es especificado en la fase de diseño de comunicación.

Nosotros conceptualizamos la necesidad de comunicación entre 2 canales como un enlace de tareas, en la cual una tarea envía los mensajes y otra puede recibirlos.

La comunicación asociada con un algoritmo puede ser especificada en 2 fases. Primero definimos una estructura de canales que enlaza (directa o indirectamente) tareas que requieren datos (consumidores) con tareas que poseen esos datos (productores). Segundo, nosotros especificamos los mensajes que van a ser enviados y recibidos por esos canales.

Identificación de los requerimientos de comunicación:

- Comunicación Local: Se obtiene una estructura de comunicación local cuando una operación requiere datos de un pequeño número de otras tareas (Sus vecinos).

- Comunicación Global: Una comunicación global es una en la que las muchas tareas deben participar. Entonces cuando esas operaciones son implementadas, podría no ser suficiente simplemente identificar pares individuales de productores/consumidores.

- Comunicación Dinámica no estructurada: En una comunicación estructurada, una tarea y sus vecinos forman una estructura regular como un árbol o una malla; En contraste en una comunicación no estructurada la red de comunicación puede crear un grafo arbitrario (Como en cálculos de elementos finitos).

- En una comunicación estática la identidad de los compañeros de comunicación no cambia con el tiempo; En contraste la identidad de los compañeros de comunicación en una estructura dinámica puede ser determinada por el cómputo de datos en el momento de ejecución y es altamente variable.

- Comunicación sincrónica/asincrónica: En la comunicación sincrónica, los productores y consumidores se ejecutan de manera coordinada, con pares de productores/consumidores cooperando en las operaciones de transferencias de datos ; En contraste, en la comunicación asincrónica la comunicación puede requerir que el consumidor obtenga datos sin la cooperación del productor.

- Análisis de dependencia / independencia: La diferencia principal entre un programa secuencial y un programa paralelo es la condición potencial de "cambio". Una condición de cambio es un estado del programa en donde una variable puede tener más de un valor, dependiendo de las rutas de un programa hacia la variable. En pocas palabras, el valor actual guardado en la variable podría o no depender del orden en que el programador escriba las sentencias.

Debido a la carga de los diferentes lugares de procesamiento, la velocidad de estos, y la posibilidad de tener diferentes arquitecturas (como un racimo heterogéneo de estaciones de trabajo), nosotros no podemos determinar siempre el orden de

ejecución. La sincronización es usada para cuantificar el orden desconocido de ejecución y prever las condiciones de cambio.

Hay dos tipos de condiciones de cambio:

- Dependencia de salida: Supongamos que tenemos las siguientes sentencias de asignación:

x=0x=1

Si estas dos sentencias son ejecutadas en el mismo instante por dos sitios diferentes de procesamiento, ¿Cual es el valor de x ?

- Dependencia de flujo: Suponga que el programador quiere que a y se le asigne el valor de x antes de que esta sea actualizada.

y = x

x = 1

No podemos reasignar a *x* hasta que *y* haya copiado su valor.

Mecanismos de comunicación

Los mecanismos de comunicación varían dependiendo del tipo de máquina en la cual estemos trabajando. Si es una maquina de *memoria compartida* (En la cual todos los procesadores tienen acceso a un sólo espacio de memoria) utilizamos ***objetos compartidos***, en cambio si utilizamos una maquina de memoria distribuida (En la cual cada procesador tiene su propia memoria) utilizamos ***pase de mensajes.***

Pase de mensajes

Los lugares de procesamiento se comunican con otros enviando y recibiendo mensajes sobre enlaces de redes.

El programador debe colocar explícitamente un mensaje de llamada

No se asume que los sitios de procesamiento tienen alguna memoria en común

Los mensajes proveen una manera de que varios sitios se comuniquen e intercambien datos

Un mensaje es sólo un conjunto de datos.

El mensaje puede ser enviado de un punto especifico a otro (*point to point*), a todos los sitios (*broadcast*), o a un subconjunto de sitios (*multicast*).

Conceptualmente, enviar un mensaje es un proceso de dos partes:

- Ordenar los datos
- Enviar los datos

Analogía : Ordenar los datos es como juntar a la gente en una parada de autobús y hacerlas abordar el autobús. Enviar los datos es como enviar el autobús a otra parada.

Igualmente, recibir un mensaje es un proceso de dos partes:

- se reciben los datos
- se desordenan los datos

Analogía: Recibir los datos es análogo a que el autobús llegue a la parada de autobuses. Desordenar los datos es análogo a que la gente salga del autobús y tome rumbos diferentes.

El objetivo del pase de mensajes es gastar más tiempo haciendo cálculos que comunicaciones.

Cuando utilizar pase de mensajes:

- Control preciso / máxima eficiencia
  - Aplicaciones comerciales
  - Códigos de gran desafío
- Entornos heterogéneos / estaciones de trabajo
  - Deseos de usar los recursos locales
  - Super - Racimos
- Mensajes con datos de tipo obvio (Como problemas moleculares)
- Problemas de balanceo de cargas
  - Dinámicas moleculares
  - Manejo de información, bases de datos
  - Procesamiento de transacciones

¿Cómo implementar el pase de mensajes?

- Sockets (bajo nivel, nivel de redes)
- RPC (Llamadas de procedimientos remotos)
- Otros (PVM,P4,MPI)

Paralelismo de datos

En este tipo de comunicación solo existe un ejecutable; Además se hacen cómputos en arreglos de datos utilizando operadores matriciales y comunicaciones que utilizan corrimientos en matrices (O operaciones de acomodamiento).

Este enfoque tiene las siguientes bondades:

- Se puede escalar de manera transparente en maquinas de diferente tamaño
- Es fácil de depurar y sólo hay una copia de código ejecutándose de una manera altamente sincronizada.

Pero presenta las siguientes debilidades:

- Se desperdicia mucha sincronización
- Es difícil balancear las cargas

¿Cuándo utilizar paralelismo de datos?

- Programas muy orientados hacia el uso de matrices
  - Dinámicas de fluidos
  - Redes neuronales
  - Modelado del tiempo
- Operaciones muy sincronizadas
  - Procesamiento de imágenes
  - Análisis matemáticos

Trabajo compartido

Se separan tareas (En vez de arreglos en paralelismo de datos) como lazos entre procesadores separados. Se hacen cómputos en lazos que son distribuidos de manera automática.

Este enfoque tiene las siguientes bondades:

- Está basado en directivas, por lo que puede ser agregado al código serial existente
- Bueno para implementaciones de memoria compartida.

Debilidades :

- Flexibilidad limitada
- La eficiencia varia dependiendo de la estructura del código serial existente
- Puede ser pobre en memoria compartida.

¿Cuándo utilizar trabajo compartido?

- Códigos muy largos, complejos o viejos
- Códigos que hacen multitarea

Puntos a verificar en el diseño comunicacional:

Estas normas están pensadas para identificar características de no escalabilidad, en vez de ser reglas duras y de fácil aplicación.

- Todas las tareas desempeñan el mismo número de operaciones de comunicación. Los requerimientos de comunicación desbalanceados sugieren una construcción no escalable. Por ejemplo, si una estructura de datos utilizada está encapsulada en una sola tarea, considere distribuir o replicar esa estructura de datos.

- Cada una de las tareas se comunica sólo con un número de vecinos. Si cada tarea debe comunicarse con muchas otras tareas evalúe la posibilidad de reformular esta comunicación global en términos de una estructura de comunicación global.

- Todas las operaciones de comunicación se ejecutan de manera concurrente. Si no, el algoritmo es posiblemente ineficiente y no escalable.

- Los cómputos asociados con las diferentes tareas son capaces de proceder concurrentemente. Considere si puede reordenar las operaciones de cómputo de manera que la comunicación mejore.

aglomeración

En esta etapa nos movemos de lo abstracto a lo concreto. Verificamos decisiones hechas en las fases de comunicación y particionamiento con la visión de obtener un algoritmo que se ejecute eficientemente en alguna clase de computador. En particular veremos si es conveniente aglomerar tareas en la parte de particionamiento para obtener un número menor de tareas de mayor tamaño. También determinaremos si vale la pena replicar datos y/o cómputos.

El número de tareas obtenidas en la fase de aglomeración, aunque reducido, puede ser aún más grande que el número de procesadores (Lo cual puede resolverse en la fase de mapeo). Otra alternativa es que durante la fase de aglomeración reduzcamos el número de tareas a un procesador.

Tenemos tres puntos de vista (A veces conflictivos) que tratan de orientar la forma en que implementaremos la aglomeración y el replicamiento:

- Reducir los costos de comunicación **incrementando la granularidad** de las comunicaciones y cómputos.

- **Retener la flexibilidad** con respecto a la escalabilidad y decisiones de mapeo

- **Reducir los costos de ingeniería del software**.

Incrementando la granularidad

En la fase de particionamiento del proceso de diseño, nuestros esfuerzos están enfocados en definir la mayor cantidad de tareas posibles. Esta es una disciplina útil porque nos fuerza a considerar un rango más amplio de oportunidades para la ejecución paralela. Debemos darnos cuenta que el definir un numero largo de tareas de grano fino no produce necesariamente un algoritmo paralelo eficiente.

Un aspecto crítico que influencia la computación paralela es el costo de comunicación. En muchas computadoras paralelas tenemos que dejar de hacer cómputos para enviar y recibir mensajes. Debido a que típicamente nosotros deberíamos estar haciendo cómputos, podemos aumentar el rendimiento reduciendo el tiempo que gastamos en comunicarnos (El rendimiento se puede aumentar enviando menos datos o se pueden utilizar menos mensajes pero de mayor tamaño (Debido al costo asociado a iniciar una comunicación). Aparte del costo de comunicación existe un costo relacionado a la creación de tareas.

Algunas formas de incrementar el rendimiento:

- Efectos de superficie a Volumen: Si el número de compañeros de comunicación es pequeño, entonces podemos reducir tanto el número de operaciones de comunicación y el volumen total de comunicaciones incrementando la granularidad de nuestra partición, eso es, aglomerar varias tareas en una sola. Cuando nos topamos con un problema de granularidad el programador debe tener en cuenta el siguiente factor:

$$\frac{Computo}{Comunicacion}$$

La meta del programador debe ser maximizar este cociente (En otras palabras mantenga la comunicación al mínimo y haga una larga cantidad de cómputos).

- Replicando cómputos: A veces podemos intercambiar el repetir cómputos por requerimientos de comunicación reducidos (o tiempo de ejecución), La idea es mantener la concurrencia y propagar cualquier cambio de manera simultánea, reduciendo así los tiempos de ejecución.

- Evitando la comunicación: La aglomeración es casi siempre benéfica si el análisis de comunicación revela que un conjunto de tareas no puede ser ejecutada de manera concurrente. Si hay muchos mensajes por enviar puede ser mejor agruparlos para que formen un mensaje grande.

Preservando la flexibilidad

Es fácil cuando se aglomeran tareas el hacer decisiones de diseño que limitan innecesariamente la escalabilidad de un algoritmo. También la habilidad de crear un número variante de tareas es crítica si nuestro programa a ser portable y escalable. Si una tarea se bloquea esperando por datos remotos, puede ser mejor mapear otras tareas a ese mismo procesador ya que en vez de tener una tarea ociosa (bloqueada por espera de datos) podemos tener otras tareas haciendo cómputos. El número óptimo de tareas se determina mejor usando una combinación de modelado analítico y estudios empíricos.

Reducir los costos de la ingeniería de software

Otro punto importante cuando se paralelizan códigos existentes secuenciales, son los costos relativos de desarrollo asociados con las diferentes etapas de particionamiento. Desde esta perspectiva, las estrategias más interesantes pueden ser aquellas que eviten cambios extensivos en el código. Por ejemplo, en un código que opera en una malla multidimensional, podría ser ventajoso evitar particionar todo en una sola dimensión, si al hacerlo permite que las rutinas existentes sean reutilizadas sin cambio en un programa paralelo.

Frecuentemente, estamos preocupados por el diseño de un algoritmo paralelo que se debe ejecutar como parte de un sistema grande. En este caso, otro problema de la ingeniería del software que debe ser considerado son las distribuciones de datos utilizados utilizadas por otros componentes del programa.

Lista de verificacion para la aglomeracion

Varias de estas preguntas hacen énfasis en un análisis de desempeño, el cual se vuelve más importante a medida que nos movemos de lo abstracto a lo concreto.

- La aglomeración ha reducido los costos de comunicación al incrementar la localidad. Si no, examine su algoritmo para determinar cuando se puede alcanzar esto usando una estrategia de aglomeración distinta.

- Si la aglomeración ha replicado cómputo, ¿Ha verificado que los beneficios de este replicamiento minimizan los costos, a cambio de un rango de tamaños de los problemas y cantidad de procesadores usados?

- Si la aglomeración ha producido un replicamiento de los datos, ¿Verificó que esto no comprometa la escalabilidad de su algoritmo al restringir el rango de tamaño de los problemas o cantidad de procesadores?

- .La aglomeración ha producido tareas con costos similares de computación y comunicación. Entre más grandes sean las tareas creadas por la aglomeración, mayor importancia tendrá que tengan costos similares. Si se ha creado una tarea por procesador, entonces esas tareas deberán tener casi los mismos costos.

- El número de tareas es aún escalable con el tamaño del problema. Si no, entonces su algoritmo no será capaz de resolver grandes problemas en grandes computadoras paralelas.

- Si la aglomeración eliminó oportunidades de ejecución concurrentes, ¿Verificó que hay suficiente concurrencia para computadoras actuales y futuras? (Un algoritmo con poca concurrencia puede ser aún el más eficiente).

- ¿Puede ser reducidas aún más el número de tareas sin introducir desbalances de cargas, incrementar los costos de ingeniería del software o reducir la escalabilidad?

- Si esta paralelizando un programa secuencial, ¿Ha considerado el costo de las modificaciones requeridas para el código secuencial?

Mapeo

En la cuarta y última etapa del diseño de un algoritmo paralelo, nosotros especificamos donde (procesador) se debe ejecutar cada tarea. Este problema de mapeo no surge en uniprocesadores o maquinas de memoria compartida que proveen planeamientos de tareas automática. En estas máquinas, un conjunto de tareas y sus requerimientos asociados de comunicación tienen ya planteadas las suficientes especificaciones para su ejecución (Se deja al sistema operativo o al hardware la responsabilidad de repartir esas tareas entre los procesadores disponibles). Podemos darnos cuenta que los mecanismos de mapeo de propósito general está aún lejos de ser desarrollados para computadoras paralelos escalables.

Nuestra meta en el desarrollo de algoritmos de mapeo es normalmente minimizar el costo de tiempo de ejecución total. Utilizamos dos estrategias para alcanzar esa meta:

- Colocaremos tareas que son capaces de ejecutarse de manera concurrente en diferentes procesadores, para ampliar la concurrencia.

- Colocaremos tareas que se comuniquen frecuentemente en el mismo procesador para incrementar la localidad.

Claramente, estas 2 estrategias van a estar en conflicto, en cuyo caso nuestro diseño tendrá desventajas. Además, las limitaciones de recursos pueden restringir el número de las tareas que pueden ser colocadas en un sólo procesador.

El problema de mapeo es conocido como NCT, lo cual significa que no es computacionalmente tratable (No hay algoritmo existente para evaluar las desventajas en el caso general). Se puede ver se puede generar un conocimiento considerable usando estrategias especializadas y algoritmos heurísticos en el tipo de problemas en donde se crea que son efectivos.

Muchos algoritmos desarrollados usando técnicas de descomposición por dominio producen un número fijo de tareas de igual tamaño con comunicaciones globales locales estructuradas. En esos casos un mapeo directo es eficiente.

En algoritmos más complejos basados en descomposición por dominio con cantidades variables de trabajo y patrones no estructurados de comunicación, las técnicas más eficientes de

mapeo y aglomeración pueden no ser obvias para el programador. En estos casos podemos emplear algoritmos de cargas balanceadas que buscan identificar la aglomeración y mapeo eficiente, usando típicamente técnicas heurísticas. Sin embargo otra alternativa con una menor sobrecarga son los métodos de carga balanceada probabilística los cuales explotan la estructura de una aplicación.

Los problemas más complejos son aquellos en los cuales tanto el número de tareas o la cantidad de cómputos o la cantidad de comunicación por tarea cambia dinámicamente durante la ejecución de un programa.. En el caso de problemas desarrollados utilizando técnicas de descomposición por dominio, podremos usar una estrategia de balanceo de cargas dinámica en el cual un algoritmo de balanceo de cargas es ejecutado periódicamente para determinar nueva aglomeración y mapeo. Debido a que el balanceo de cargas debe ser realizado varias veces durante la ejecución de un programa, los algoritmos locales pueden ser preferidos a aquellos que requieren un conocimiento global del estado computacional.

Los algoritmos basados en composición funcional a veces producen cómputos consistentes en muchas tareas de corta duración que coordinan con otras tareas solo al comienzo y al final de la ejecución. En este caso, podemos usar un algoritmo de planeamiento de tareas, el cual asigna las tareas que están activas o van a serlo.

Algoritmos de carga balanceada

Estas técnicas están pensadas para aglomerar tareas de grano fino definidas en una partición que produce una tarea de grano grueso por procesador. Veamos algunos algoritmos:

Bisección recursiva

Las técnicas de bisección recursiva son utilizadas para particionar un dominio (Ejemplo: una malla de elementos finitos) en subdominios de costo aproximadamente igual mientras tratan de minimizar los costos de comunicación, es decir, el número de canales que cruzan los límites de las tareas. Se utiliza un enfoque de divide y vencerás. El dominio es cortado en una dimensión para producir dos subdominios. Se hacen cortes de manera recursiva en los nuevos subdominios hasta que tenemos tantos subdominios como tareas requeridas.

La forma más directa de las técnicas de bisección con recursión es la bisección recursiva coordinada, la cual es aplicada normalmente a mallas irregulares que tienen básicamente una estructura de comunicación local. Esta técnica hace cortes basados en las coordenadas físicas de los puntos de la malla en el dominio, en cada paso subdividiendo a lo largo de una dimensión X, para obtener un subdominio. Este enfoque tiene la ventaja de ser simple y barato, pero no optimiza el desempeño de la comunicación.

Una variante de la bisección recursiva llamada bisección recursiva no balanceada trata de reducir los costos de comunicación al formar submallas que tengan los mejores radios de trazado. En vez de dividir la malla de manera automáticamente por la mitad, ella considera a las P-1 particiones obtenidas al formar submallas con 1/P y (P-1) de la carga, con 2/P y (P-2)/P de la carga y así sucesivamente, seleccionando a la partición que minimice el radio de trazado. Este método incrementa los costos de computación de la partición pero puede reducir los costos de comunicación.

Otra técnica, llamada gráfico de bisección recursiva pueden ser útil en el caso de mallas más complejas no estructuradas, por ejemplo, mallas de elementos finitos. Esta técnica utiliza información de conectividad para reducir el número de bordes de la malla que cruzan los límites de los subdominios, con el fin de poder reducir los requerimientos de comunicación. Una malla es tratada como un grafo con N vértices. El algoritmo identifica primero las 2 extremidades del grafo, eso es, los 2 vértices que están más separados en términos de la distancia del grafo (La distancia del grafo entre dos vértices es el número más pequeño de arcos que deben ser cruzados para ir de uno al otro). A cada vértice le es asignado entonces un subdominio correspondiente a la extremidad más cercana.

Algoritmos locales

Las técnicas descritas con anterioridad son relativamente costosas porque ellas requieren el conocimiento global del estado de computo. En contraste, los algoritmos de balanceo de cargas locales compensan los cambios en la carga computacional usando sólo la información obtenida de un pequeño número de procesadores vecinos. Por ejemplo, los procesadores podrían estar organizados en una malla lógica; Periódicamente, cada procesador compara su carga computacional con la de sus vecinos y transfiere cómputos si esa carga excede cierto umbral. Tenemos el problema de que ellos pueden ser lentos para ajustar los cambios mayores en características de carga. Por ejemplo, si una carga aparece súbitamente en un procesador, se requerirán múltiples operaciones de balanceo de cargas antes de la carga se *difumine* entre los otros procesadores.

Métodos probabilisticos

Otro enfoque simple para el balanceo de cargas es seleccionar aleatoriamente entre los procesadores. Si el número de tareas es largo, podemos esperar que a cada procesador se le asigne la misma cantidad de cómputos. La ventaja de esta estrategia son los costos y la escalabilidad. La desventaja es que la comunicación entre procesadores es requerida virtualmente por cada tarea y una distribución de carga es alcanzada sólo si hay más tareas que procesadores posibles. La estrategia tiende a ser más efectiva cuando hay relativamente poca comunicación entre tareas y/o poca localidad en patrones de comunicación.

Mapeo cíclico

Si nosotros conocemos tanto la carga computacional variable por punto en la malla y que hay localidad espacial en los niveles de carga; Es entonces cuando un mapeo cíclico de las tareas en los procesadores puede ser apropiada. Eso es cada uno de los procesadores P son asignados a cada P - ésima tarea de acuerdo con cierta numeración de las tareas. La meta es que, en particular, a cada procesador le sea asignada la misma carga computacional. Los beneficios de un balance de carga deben ser contrastados con el incremento en los costos de comunicación debido a la reducida localidad.

Algoritmos para el balanceo de cargas

Los algoritmos de planeamiento de tareas pueden ser usados cuando una descomposición funcional produce muchas tareas, cada una con requerimientos de localidad débiles. Se mantiene una "piscina" de tareas centralizada o distribuida, en la cual las nuevas tareas son colocadas y desde las cuales las tareas son tomadas para ser asignadas a procesadores. En efecto, nosotros reformulamos el algoritmo paralelo para que lo que fueron concebidos originalmente como tareas se conviertan en una estructura de datos representando "problemas" a ser resueltos por un conjunto de tareas de trabajo, típicamente una por procesador.

El aspecto más crítico del algoritmo de planeamiento de tareas es la estrategia usada para signar problemas a los trabajadores. Generalmente, la estrategia seleccionada representa un compromiso entre los requerimientos de operación independientes (Para reducir los costos de comunicación) y el conocimiento global de los estados de computación (Para mejorar el balance de cargas).

El balanceo de cargas puede ser *estático* (el programador debe hacer una decisión y asignar una cantidad fija de trabajo a cada sitio de procesamiento a priori) o *dinámico* (El cual puede ser a su vez *orientado por tareas*, en el cual cuando un procesador termina su tarea le es asignada otra, u *orientado por datos,* en el cual cuando un lugar de procesamiento finaliza antes que sus vecinos *toma* los datos del sitio con más trabajo).

Existen varios esquemas que podemos aplicar:

Administrador / Trabajador

Un administrador central de tareas tienen la responsabilidad de la asignación de tareas. Cada trabajador pide y ejecuta de manera repetida, tareas del administrador. Los trabajadores también envían nuevas tareas al administrador para otros trabajadores. La eficiencia de esta estrategia depende del número de trabajadores y de los costos de obtener y ejecutar problemas.

La asignación de las tareas puede ser estática o dinámica:

ESTÁTICO

El pseudocódigo que hace esto es el siguiente:

Código del Administrador

```
por cada nodo_trabajador
   inicie(codigo_trabajador,nodo_trabajador)
   envie(dato_de_control,nodo_trabajador)
fin
```

Y cuando termina el trabajo :

```
por cada nodo_trabajador
   reciba(resultados,nodo_trabajador)
   termine(nodo_trabajador)
fin
```

Código de cada nodo trabajador

```
reciba(dato_de_control,nodo_administrador)
haga_trabajo(dato_de_control)
envie(resultafos,nodo_administrador)
```

DINÁMICO :

Código del administrador: (Envío)

```
por cada nodo_trabajador
   inicie(código esclavo, nodo_trabajador)
```

Código del administrador: (Recepción)

```
Mientras (hayan tareas por despachar)
   si (un procesador no tiene tareas)
         enviar(dato_de_control,nodo_esclavo)
   si no (Los procesadores todos tienen tareas)
      recibir (resultados,nodo_trabajador) ;
```

```
                terminar(nodo_trabajador)

      finsi

   Mientras (hayan resultados afuera)

        recibir(resultados,nodo_trabajador) ;
   terminar(nodo_trabajador)

Código del trabajador :

recibir(dato_de_control,nodo_administrador)

hacer_trabajo(dato_de_control)

enviar(resultados,nodo_maestro)
```

Administrador / Trabajador jerárquicos

Una variante del esquema anterior divide los trabajadores en conjuntos distintos, cada uno con su sub - administrador, los cuales se comunican periódicamente con el administrador y con otros sub - administradores para balancear las cargas entre el conjunto de procesadores de los cuales ellos son responsables.

Esquemas descentralizados

En este esquema no hay un administrador central. En vez de eso, una piscina de tareas separadas es mantenida por cada procesador, y los trabajadores inactivos piden problemas de otros procesadores. En efecto, una piscina de tareas se convierte en una estructura de datos distribuida que es accesada por las diferentes tareas de manera sincrónica. Una variedad de políticas de acceso pueden ser definidas, por ejemplo, un trabajador puede pedir trabajo de un grupo pequeño de vecinos o puede seleccionar otros procesadores de manera automática.

ESTÁTICO

Código ejemplo para todos los nodos :

```
Si yo soy el nodo_de_comienzo entonces

   por cada uno de los otros nodos

      comienzo

         inicie(solo)codigo,otro_nodo)

         envie(dato_de_control,otro_nodo)

      fin

   sino

      reciba(dato_de_control,nodo_comienzo)

      haga_trabajo(dato_de_control)
```

Y cuando recibe:

```
Si yo soy el nodo_de_comienzo entonces por cada nodo haga
comienzo recibir(resultados,otro_nodo) ; termine(otro_nodo) ;fin
en caso contrario(resultados,nodo_comienzo) ;
```

Detección de terminación

Los algoritmos de planeamiento de tareas requieren un mecanismo para determinar cuando una búsqueda está completa; de otra forma, los trabajadores inactivos nunca pararán de pedir trabajo a otros trabajadores. Esta operación de detección de terminación es directa en esquemas centralizados, debido a que el administrador puede determinar fácilmente cuando todos los trabajadores están inactivos. Es más difícil en algoritmos descentralizados debido a que no sólo no hay un registro central de cuales trabajadores están activos, pero también los mensajes que se transmiten pueden llevar tareas aún cuando todos los trabajadores parezcan estar inactivos.

Lista de verificación para el mapeo

Nuestras decisiones de mapeo buscan balancear el conflicto que hay entre los requerimientos de una carga distribuida y los costos de comunicación (mínimos). Cuando sea posible utilizaremos un esquema de mapeo estático el cual asigne cada tarea a un sólo procesador Sin embargo, cuando el número o tamaño de las tareas es variable o no se conoce hasta la ejecución, podríamos utilizar un esquema de balanceo de cargas dinámico o reformular el problema para que la estructura de planeamiento de tareas pueda ser utilizada para planear los cómputos.

- Si consideramos un diseño de SPMD para un problema complejo, también considere un algoritmo basado en la creación y eliminación dinámica de tareas (La última forma de atacar el problema puede producir un algoritmo más sencillo pero de poco rendimiento)

- Si se consideró un diseño basado en la creación y eliminación dinámica de tareas, ¿Consideró un algoritmo SPMD? Un algoritmo SPMD provee un gran control sobre el planeamiento de la comunicación, pero puede ser más complejo.

- ¿Si usa un esquema centralizado de balanceo de cargas, ha verificado que el administrador de tareas no se convertirá en un cuello de botella? usted podría ser capaz de reducir los costos de comunicación en estos esquemas al pasar punteros a las tareas, en vez de las tareas mismas, a el administrador.

- Si usa un esquema de balanceo de cargas dinámico, ¿Evaluó los costos relativos de estrategias diferentes? Asegúrese de incluir los costos de implementación en su análisis. Los esquemas de mapeo probabilísticos o cíclicos son simples y deberían ser considerados siempre debido a que ellos pueden evitar la necesidad de operaciones de cargas repetidas.

- Si se usan métodos probabilísticos o cíclicos, ¿tiene usted un número suficiente de tareas para asegurar un balance de carga razonable (Típicamente, al menos 10 veces tareas sobre procesadores).

HERRAMIENTAS PARALELAS

Actualmente, existe una gran variedad de herramientas que permiten la programación de sistemas paralelos. Las herramientas estudiadas fueron de rango y alcance muy variado. Por comodidad fueron clasificadas de esta manera:

- Lenguajes paralelos:

  CC++ (Compositional C++)

Fortran M (FM)

High Performance Fortran (HPF)

- Librerías que permiten paralelización (Pueden ser llamadas desde C y Fortran):

  PVM (Paralell Virtual Machine)

  ESSL

  MPI (Message Passing Interface). En realidad MPI es un modelo de referencia y nos referiremos a la implementación Mpich de MPI.

Solamente *2* de todas las herramientas estudiadas se encontraban instaladas en CeCalcULA para el momento de realizar las pasantías. (Sin embargo se tratará de dar una idea de las bondades de los lenguajes y librerías estudiadas).

Lenguajes paralelos:

CC++:

Son un pequeño conjunto de extensiones a C++. Esas extensiones proveen al programador con control explícito sobre localidad, concurrencia, comunicación y mapeo, además de que pueden ser usadas para construir librerías que implementan tareas, canales de comunicación y otras abstracciones básicas de programación paralela.

Por ejemplo, CC++ está mejor adaptado para aplicaciones que tienen que ver con la creación dinámica de tareas, patrones irregulares de comunicación y cálculos en ambientes heterogéneos.

CC++ se comporta igual que su predecesor C++ (Se mantienen igual conceptos como manejo de memoria, tipos estructurados de datos, clases, herencia, etc.) y se añaden las siguientes abstracciones:

- El objeto procesador (Es un mecanismo para controlar localidad)
- Puntero Global (Es un mecanismo para enlazar juntos objetos de procesadores)
- El mecanismo hilos de ejecución (*Threads*) para especificar la ejecución concurrente.
- La variable sync (Sincroniza la ejecución de hilos)
- Función atómica (Controla el espaciado de tareas)
- Funciones de transferencia (Permite transferir estructuras de datos arbitrarias entre procesadores)

El lenguaje esta soportado directo para las siguientes plataformas:

- Solaris 2.3 y 2.4 con SunCC SC3.01 (Compilador nativo de C++ de la Sun) y el paquete Thread (Viene incorporado con el Sistema Operativo).
- SunOS 4.1.3 con g++2.6.3 (Y la librería FSUPthreads)
- IBMSP2 con el compilador xlC (AIX 3.2.5) con las librerías Thread DCE de IBM.

No se hizo un estudio más profundo de este lenguaje por las siguientes razones:

- Ninguna de las Sparc en la U.L.A tiene más de un procesador instalado.

- La mayoría de las máquinas tiene SunOs 4.1.4 (Lo que implicó montar aparte las librerías FSUPthreads, ya que el sistema operativo no las provee). Sin embargo, para compilar a CC++ había que utilizar g++2.6.2, cuando en la ULA la versión que está instalada es la g++2.7.2 (Versión más actualizada). Se intentó montar una versión ya compilada pero aparte de que no se instala en la ubicación estandar (/usr, /usr/local), nunca trabajo bien.

- No se instaló en la SP2 porque no se disponían las librerías Thread DCE (librerías comerciales).

- El paquete no estaba disponible en CeCalcULA.

Fortran M (FM)

Son un pequeño conjunto de extensiones para Fortran (De hecho cualquier Programa válido de Fortran 77 puede ser compilado bajo Fortran M).

Fortran M provee constructores de lenguaje para crear tareas y canales, y para recibir mensajes. Se asegura que los programas son determinísticos sí los constructores son utilizados, además de proveer las propiedades de encapsulamiento necesarias para la programación modular. El mapeo de constructores afecta el desempeño pero no la estructura del programa, lo cual permite que las decisiones de mapeo sean cambiadas sin alterar otros aspectos de diseño del programa.

Las extensiones de Fortran M (FM) son las siguientes:

- Una tarea es implementada como un proceso. Una definición de un proceso tiene la misma sintaxis que una subrutina, excepto por la palabra clave *PROCESS* la cual es sustituida por la palabra clave *subroutine* .

- Canales de un solo productor, un solo consumidor y combinaciones de productores múltiples y un solo consumidor son creados con la palabra *CHANNEL* y *MERGER* respectivamente. Estas sentencias toman nuevos tipos de datos, llamados importadores y exportadores como argumentos y los definen para que sean referencias a las nuevas estructuras de comunicación.

- Los procesos son creados en bloques de procesos y lazos de procesos (y les pueden ser pasados puertos de entrada y puertos de salida como argumentos).

- Se proveen sentencias para enviar mensajes a los puertos de salida, para recibir mensajes de los puertos de entrada y para cerrar un puerto de salida (SEND, RECEIVE y ENDCHANNEL respectivamente). Los mensajes pueden incluir variables de puerto, lo cual permite a un procesador transferir a otro el derecho de enviar y recibir mensajes en un canal o mezclador.

- Los constructores de mapeo pueden ser usados para especificar que un programa se ejecute en un arreglo de procesadores virtuales de forma y tamaño especificado, para localizar un proceso dentro de este arreglo de procesadores, o para especificar que un proceso se va a ejecutar en un subconjunto de ese arreglo de procesadores.

- Por conveniencia, a los procesos se les puede pasar variables ordinarias como parámetros igual que los puertos. Estas variables son copiadas en llamadas y retornan, lo que asegura una ejecución determinística. El copiado de variables puede ser omitido para mejorar el desempeño.

El lenguaje enfatiza el soporte para la modularidad. Los componentes pueden ser combinados usando composición secuencial, paralela o concurrente. En la composición paralela y concurrente,

el proceso sirve como un bloque de abstracción básico. Un proceso puede encapsular datos, cálculos, y concurrencia; Los puertos y otras variables pasadas como argumentos definen su interface con el resto del programa.

Este lenguaje no estaba disponible en CeCalcULA.

High Performance Fortran:

El término paralelismo de datos se refiere a la concurrencia que es obtenida cuando la misma operación es aplicada a alguno o todos los elementos de un conjunto de datos. Un programa de paralelismo de datos es una secuencia de esas operaciones. Un algoritmo paralelo es obtenido de un programa de paralelismo de datos al aplicar técnicas de descomposición en las estructuras de datos sobre las que opera. Las operaciones de particionamiento, frecuentemente de acuerdo a la regla de "propiedad computacional" en la cual el procesador que posee un valor es el responsable de actualizarlo. Típicamente el programador es el responsable de especificar la descomposición por dominio, pero la computadora particiona los cómputos de manera automática.

Hablemos un poco de Fortran 90 (El predecesor de HPF) antes de continuar. Fortran 90 es un lenguaje de programación orientado a el paralelismo de datos en su propio derecho. Las sentencias de asignación de arreglos y funciones propias de arreglos pueden ser usadas para especificar ciertas clases de cálculos de paralelismo de datos.

Fortran 90 extiende a Fortran 77 con punteros, tipos de datos definidos por el usuario, módulos, rutinas recursivas, asignación de almacenamiento dinámico, operaciones con arreglos, funciones intrínsecas y otras características. En cambio HPF extiende a Fortran 90 con directivas de distribución de datos entre procesadores, la sentencia FORALL, la directiva INDEPENDENT y otras funciones nuevas. La asignación de arreglos, la sentencia, la sentencia FORALL y la directiva INDEPENDIENT son usadas para identificar la concurrencia en un algoritmo, mientras que las directivas de distribución de datos especifican como los datos deben ser colocados en los procesadores físicos (y por ende se provee control sobre la localidad).

Aunque HPF define solo un conjunto pequeño de extensiones sobre F90, es un lenguaje complejo. Las extensiones tienen muchas variantes y pueden interactuar como constructores F90 de muchas manera posbiles.

High Performance Fortran (HPF) es una extensión a Fortran 90 (El cual es a su vez una extensión a Fortran 77). Fortran 90 provee constructores para especificar la ejecución concurrente pero no la descomposición por dominio; HPF aumenta a Fortran 90 con constructores paralelos adicionales y directivos de ubicación de datos, las cuales permiten que muchos programas HPF sean compilados con una eficiencia razonable dentro de un rango de maquinas paralelas. HPF soporta modelos de programación paralelos basados en el paralelismo de datos (Lo que lo hace muy atractivo para resolver problemas de computación gráfica).

Librerías paralelas:

PVM (Paralell Virual Machine)

PVM se basa en un modelo de pase de mensajes. PVM permite a una colección de computadoras actuar como una gran máquina virtual paralela (De hay su nombre), además de que puede ser usado desde lenguajes como Fortran y C. Una ventaja de PVM es que permite trabajar como una unidad a un conjunto de máquinas con características distintas en una red de manera coordinada (Ambiente heterogéneo), a diferencia de Lenguajes como HPF. PVM tiene en cuenta las diferencias de velocidades computacionales (Un nodo de cálculo puede ser un 386 mientras que otro nodo puede ser una Sparc20),las incompatibilidades entre los formatos de datos (Por ejemplo, un procesador Intel puede representar internamente los números de punto flotante de manera distinta a un procesador Sparc), el tiempo que le toma a una máquina enviar un mensaje por la red (El cual depende de la carga impuesta por los usuarios en la red). PVM maneja de manera

transparente todo el enrutamiento de mensajes, conversión de datos y planeamiento de tareas a lo largo de la red (con arquitecturas incompatibles). En PVM, el usuario escribe su aplicación como una colección de tareas cooperativas. Las tareas acceden los recursos de PVM usando una librería de rutinas estándar. Estas rutinas permiten la iniciación y terminación de tareas atraves de la red, al mismo tiempo que se sincronizan y comunican entre ellas.

En CeCalcULA también está disponible una versión de PVM llamada PVMe. Mientras que PVM fue desarrollado para trabajar en un ambiente heterogéneo, PVMe fue desarrollado para ser usado en una plataforma homogénea, en el sistema SP de IBM con HPS (High Performance Switch).

PVMe consiste en:

- Dos bibliotecas (Una para Fortran y otra para C)

- Un programa especial (La consola) para supervisar el comportamiento de cada sesión.

PVMe es compatible con PVM, pero su estructura interna es completamente nueva. Debido a que PVMe corre sobre una plataforma homogénea, muchas operaciones son más simples.

Algunas diferencias entre PVM y PVMe de dominio público:

- PVMe no actúa directamente con TCP/IP para ejecutar la comunicación de datos entre procesadores. Para esto se utiliza un sistema subsistema de comunicación (CSS), software de comunicación que corre sobre el adaptador del HPS. El intercambio de datos ocurre entre procesos, sin invocaciones del demonio.

- El demonio de PVMe corre en un solo nodo y no en todos los nodos participantes como es el caso de PVM.

- Un solo servidor maneja tanto las funciones básicas (comunicación de datos, manipulación de procesos) como el manejo de grupos dinámicos. La biblioteca básica de PVMe contiene rutinas que trabajan con grupos dinámicos, mientras que en el PVM de dominio público estas rutinas están en una biblioteca independiente.

- Un esquema mejorado de manejo de memoria de PVMe permite aprovechar mejor los recursos.

Biblioteca de funciones para C y Fortran para Ciencias e Ingeniería (High Performance): ESSL

ESSL son un conjunto de 442 subrutinas las cuales proveen un conjunto amplio de funciones matemáticas para diferentes aplicaciones científicas y de ingeniería. Sus características primarias son desempeño, capacidad funcional y facilidad de uso.

Las librerías, en nueve áreas computacionales, están entonadas para un máximo desempeño para las plataformas IBM ES/3090, ES/9000 Vector Facilities, IBM System/370M, System/390 Scalar Procesor e IBM RISC System/6000 (Cada Nodo de la SP2 es un procesador de este tipo).

Las áreas computacionales cubiertas por estas librerías son:

- Subprogramas de álgebra lineal

- Operaciones sobre matrices

- Ecuaciones Algebraicas lineales
- Transformaciones de Fourier, Convoluciones, Correlaciones y Cálculos relacionados.
- Ordenamientos y búsquedas.
- Cuadraturas numéricas.
- Generación de números aleatorios.

ESSL también ofrece un grupo de 8 subrutinas que corren en ambientes de procesamiento paralelo en procesadores Mainframe. Se incluyen en este grupo las Transformaciones de Fourier, Ecuaciones Lineales algebraicas y operaciones con matrices.

Las librerías instaladas en CeCalcULA no son paralelas (Además de que no son portables fuera de equipos IBM, por lo que no se evaluaron).

Modelo de Programación MPI (Message Passing Interface)

MPI es una biblioteca de funciones y macros que pueden ser usados desde C, Fortran y C++. MPI es ideal para programas que quieren explotar la existencia de múltiples procesadores usando el paradigma de "pase de mensajes".

En el modelo de programación MPI, un cómputo comprende uno o más procesos que se comunican por el llamado de rutinas de uno o más procesos que se comunican por el llamado de rutinas que envían y reciben mensajes a otros procesos.

En muchas implementaciones de MPI, un conjunto fijo de procesos son creados al inicio del programa y un proceso es creado por procesador. Sin embargo, estos procesos pueden ejecutar diferentes programas. Aquí, el modelo de programación MPI es referido a veces como MPMD (Múltiples Programas Múltiples Datos) para distinguirlo de el modelo SPMD en el cual cada procesador ejecuta el mismo programa.

La comunicación entre esos procesos puede ser punto a punto (Point to Point) para enviar un mensaje de un procesos con nombre a otro; Estas operaciones pueden ser usadas para implementar comunicaciones locales y no estructuradas. Un grupo de procesos pueden llamar operaciones de comunicación colectivas para realizar operaciones globales como *sumatorias* y *broadcasts*. La habilidad de MPI de *probar* mensajes soporta comunicaciones asincrónicas. Probablemente su característica más importante desde el punto de vista de la ingeniería del software es el soporte de la programación modular. Un mecanismo llamado comunicador (*communicator)* permite al programador MPI definir módulos que encapsulen estructuras de comunicación internas.

El objetivo de MPI es establecer un mecanismo de programación paralelo independiente de las máquinas, el lenguaje de programación utilizado.

MPI es un nuevo paradigma en cuanto a la forma de paralelizar un proceso. MPI es ampliamente utilizado en máquinas paralelas, en especial aquellas con memoria distribuida (Aunque hay implementaciones para maquinas con memoria compartida).

En el diseño de MPI se trató de hacer uso de las características más atractivas de un número significativo de sistemas de pases de mensajes, en vez de seleccionar uno de ellos y convertirlo en un estándar. MPI ha sido fuertemente influenciado por:

- Intel NX/2
- Express

- NCube Vertex
- P4
- PARMACS
- PVM

La ventaja de establecer un estándar de pase mensajes son la portabilidad y facilidad de uso. En un entorno de comunicación de memoria distribuida en el cual las rutinas de alto nivel y/o abstracciones están construidas sobre rutinas de bajo nivel de pase de mensajes, los beneficios de esta estandarización son aparentes.

Una lista completa de metas de MPI comprende:

- Diseño de una interfaz de programación (no necesariamente) para compiladores.
- Permitir la comunicación eficiente: evitar el copiado de memoria a memoria y permitir solapar cómputos con comunicación (además de bajar la tasa de comunicación entre procesadores0.
- Permitir una implementación ene sistemas heterogéneos.
- Permitir un "ensamblado" conveniente desde C y Fortran.
- Asumir una interfaz de comunicación confiable: El usuario no necesita tratar con fallas de comunicación (las cuales son manejadas por el subsistema de comunicación).
- Definir una interfaz que no sea muy diferente de las actuales (Como PVM, p4, Express, NX)
- Las semánticas de la implementación deben independientes del lenguaje.

Como una vez lo fuera PVM, MPI es el nuevo estándar de la computación paralela.

COMPARACIÓN ENTRE DIFERENTES LIBRERÍAS EN UNA SP2:

Las herramientas estudiadas en CeCalcULA no incluyeron lenguajes netamente paralelos porque no estaban disponibles. Sin embargo se encontró abundante material por la red Internet acerca del rendimiento de diversas librerías paralelas sobre una SP2.

Los tres criterios principales para seleccionar una librería de pase de mensajes son:

- Desempeño - latencia y ancho de banda
- Portabilidad
- Funcionalidad

Latencia (El tiempo necesario para transmitir un mensaje de tamaño 0) y ancho de banda varía ampliamente de maquina a maquina. El mejor rendimiento en una maquina especifica es obtenido típicamente de una librería nativa de pase de mensajes escrita específicamente para esa máquina.

La siguiente tabla mide la latencia y el ancho de banda de las librerías MPI, MPL y PVMe/PVM en la SP2:

	Latencia (microsegundos)	*Ancho de banda (Mbytes/segundo)*	*¿Es portable?*
MPI-F (MPI de IBM)	43	34	Sí
MPL	45	34	No
MPICH	58	33	Sí
PVMe	83	31	Sí
PVM con RouteDirect	642	12	Sí

Tabla 1: Latencia y ancho de banda entre MPL,MPI, PVMe, PVM (Tomado de http://www.mhpcc.edu/training/workshop/html/ibmwaw/ibmwaw.html#bech).

Librería	Ventajas	Desventajas	Recomendación
MPL	Soportada por IBM (Por ahora) Mejor desempeño (Actualmente disponible) Buena funcionalidad	No es portable IBM dejará de dar soporte a favor de MPI	Su uso es aceptable si lo que se quiere es máximo desempeño y no importa la portabilidad
PVM	Portable Tiene cierta funcionalidad no encontrada en otros paquetes (aún), por ejemplo, la creación dinámica de tareas.	Pobre desempeño Falta de funcionalidad (excepto en el control dinámico)	Para ser usada solamente en aplicaciones PVM existentes o aquellas que requieran características de control dinámico.
MPI	Portabilidad (Además de ser un estándar del pase de mensajes) Buen desempeño en muchas plataformas Buena funcionalidad Promesas de soporte por parte de los vendedores de la industria. MPI-2 podría agregar control de tareas dinámicas y entrada y salida paralela.		Altamente recomendado para nuevas aplicaciones.

Tabla 2: Comparación entre librerías de pase de mensajes y recomendaciones (Tomado de http://www.mhpcc.edu/training/workshop/html/ibmwaw/ibmwaw.html#bech).

MODELOS DE ILUMINACIÓN

En la generación de modelos en 3 dimensiones por computadora, uno de los aspectos más importantes es la búsqueda del realismo visual. Este realismo puede ser alcanzado utilizando un modelo de iluminación.

Un modelo de iluminación expresa los factores que determinan el color de una superficie en un punto especifico. Los modelos de iluminación son los algoritmos más básicos de asignación de color en la computación gráfica.

La interacción entre luces y superficies es muy compleja, y debido a esto mucho de los modelos de iluminación y sombreado que se han utilizado tradicionalmente en la graficación por computador incluyen muchos "trucos" y simplificaciones que no tienen una base teórica firme, pero que en la práctica funcionan bien.

Empezaremos viendo varios modelos de iluminación hasta llegar al modelo buscado. Un modelo de iluminación se puede expresar como una ecuación de iluminación de iluminación de variables asociadas con el objeto que se ilumina.

TIPOS DE MODELOS DE ILUMINACIÓN

LUZ AMBIENTAL

Cada objeto se presenta con una intensidad intrínseca (Es decir es un objeto algo irreal, autoluminoso). El objeto tendría un modelo de la siguiente forma:

$I=K$, donde K es la luminosidad propia del objeto.

Sin embargo, si en lugar de autoluminosidad hay una fuente luminosa difusa, no direccional, producto de reflexiones múltiples de la luz en las superficies presentes en el ambiente podríamos pensar en ella como una luz ambiental dada por:

$I= Ia * Ka$, donde Ia es la intensidad de la luz ambiental, constante para todos los objetos y Ka es el coeficiente de reflexión ambiental el cual es propiedad del material. Sin embargo, veremos que la luz ambiental por si sola no es de mucho interés, por lo que tomaremos en cuenta otros factores.

LUZ DIFUSA

Aunque los objetos iluminados por la luz ambiental se iluminan en forma más o menos brillante en proporción directa a la luz ambiental, se siguen iluminando de manera uniforme en todas sus superficies (lo cual da un efecto poco realista ya que se pierde la información de volumen).

Si consideramos la iluminación al objeto a partir de una fuente luminosa puntual cuyos rayos emanan uniformemente de un punto, veremos que la brillantes del objeto varia de una parte a otra, dependiendo de la dirección y la distancia con respecto a la fuente luminosa.

Para explicar mejor como trabaja el modelo de iluminación de luz difusa debemos saber que es la reflexión difusa (o reflexión Lambertiana). Las superficies que exhiben este tipo de reflexión aparecen con la misma brillantes desde todos los ángulos de observación porque reflejan la luz de con igual intensidad en todas las direcciones. Para una superficie, la brillantes depende únicamente del ángulo ϕ entre la dirección L a la fuente luminosa y la Normal N (L y N son vectores):

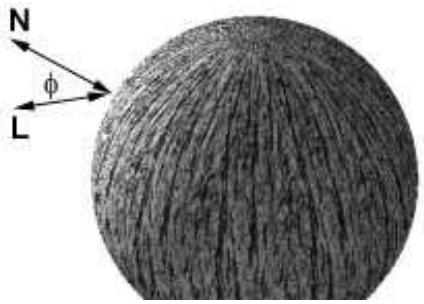


Ilustración 8: Modelo de Iluminación con luz difusa

En una superficie Lambertiana la cantidad de luz que ve el observador es independiente de la dirección de esta y sólo es proporcional al cos(ϕ), el ángulo de incidencia de la luz (ϕ debe estar entre 0 y 90 °, para que la luz tenga efecto directo en el punto que se ilumina. Esto significa que estamos trabajando con una superficie autoocluyente, de manera que la luz que es lanzada de atrás de un punto en la superficie no lo ilumina).

La ecuación para el modelo de iluminación difusa es:

$I = Ip * Kd * Cos\phi$

Donde Kd es el coeficiente de reflexión difusa.

Si normalizamos a los Vectores N y L podemos decir que $Cos\phi = N.L$ (Producto punto entre N y L). Además de esto (Para evitar que los objetos se vean duros usando solamente reflexión difusa) se agrega el factor de luz ambiental:

$I = IaKa + IpKd(N.L)$.

Sin embargo este modelo todavía presenta un problema. Si las proyecciones de dos superficies son de idéntico material, y son iluminadas desde el ojo no podremos distinguir donde comienza una superficie y donde termina la otra, sin importar cuan diferente sea su distancia a la fuente luminosa. Para ello se introduce un factor de atenuación de fuente luminosa (fatt) para obtener:

$I = Ia*Ka + fatt*Ip*Kd*(N.L)$.

Una elección obvia de fatt tiene en cuenta el hecho de que la energía de una fuente luminosa puntual que llega a una parte de una superficie disminuye en proporción inversa al cuadrado de *Dl*, la distancia que viaja la luz de la fuente puntual a la superficie. En este caso, $fatt = 1/D^2l$.

Hasta ahora solamente hemos hablado de luces monocromáticas. Las luces y superficies coloreadas se tratan de forma similar, escribiendo las mismas ecuaciones pero con diferentes valores para cada componente de color. En nuestro caso utilizaremos un modelo de 3 colores llamado RGB (*Red Green Blue*, Rojo Verde y Azul) con el cual es posible representar una amplia gama de colores. El color difuso de un objeto se representa con un valor Od para cada componente (R, G, B).

La ecuación del modelo quedaría así:

*IR = IaR*KaOdR + fatt*IpR*KdOdR*(N.L).*

*IG = IaG*KaOdG + fatt*IpG*KdOdG*(N.L).*

*IB = IaB*KaOdB + fatt*IpB*KdOdB*(N.L).*

Donde la suma de los 3 componentes me da el valor del color del punto.

LUZ ESPECULAR

La reflexión especular puede observarse en cualquier superficie brillante. Ilumine una manzana con una luz blanca brillante: El punto de máximo brillo (highlight) es ocasionado por la reflexión especular, mientras que la luz reflejada por el resto de la manzana es el resultado de la luz difusa. Observe también que en el punto de máximo brillo la manzana no aparece roja, sino blanca, el color de la luz incidente.

Ahora mueva su cabeza y note como se mueve el punto de máximo brillo. Esto sucede porque la superficie brillante refleja la luz en forma desigual en distintas direcciones. En una superficie perfectamente brillante, como un espejo perfecto, la luz se refleja sólo en la dirección de R (un vector), que es L invertida con respecto a N. De esta manera, el observador puede ver luz reflejada especularmente en un espejo solo sí el ángulo α entre los vectores R y V es 0 (El vector V es la dirección al ángulo de observación).

Es decir, gráficamente seria lo siguiente:

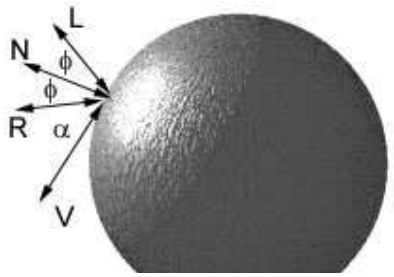


Ilustración 9: Modelo de Iluminación con luz especular

Phon Bui-Tuong desarrolló un popular modelo de iluminación para reflectores imperfectos, como la manzana. El modelo supone que la máxima reflectancia especular ocurre cuando α=0 y decrece rápidamente conforme aumenta α. Esta caída rápida es apróximada por $\cos^n\alpha$ donde n es el coeficiente de reflexión especular del material. Si consideramos que la fracción de luz reflejada especularmente viene dada por una constante llamada Ks (Coeficiente de reflexión especular) y tomamos en cuenta el color especular del objeto (Osx, donde x = R, G, B) tenemos que el modelo de iluminación completo es:
Ix = Iax *Ka * Odx + fatt * Ipx* (Kd * Odx * (N.L) + Ks * Osx * $(R.V)^n$)
En nuestro problema supondremos conocidos a N, L, R, V y a todas las constantes asociadas.
En nuestro problema secuencial, nos encontramos que hay una gran cantidad de datos para los vectores N, L, R y V (Al igual que Iax, Odx, Ipx y Osx)los cuales pueden ser cargados desde disco para ser almacenados en una matriz de n filas por 3 columnas. El resultado (Ix) se almacena en una matriz de n Filas (Cada Fila es la sumatoria de los componentes R, G, B de la luz del objeto).

PARALELIZACIÓN DEL MODELO DE ILUMINACIÓN

Una vez obtenido el modelo secuencial utilizaremos el diseño metodológico aprendido ara paralelizar nuestro modelo de iluminación.

PARTICIONAMIENTO

Expondremos aquí oportunidades de ejecución paralela.

PARTICIONAMIENTO UTILIZANDO DESCOMPOSICIÓN POR DOMINIO:

Buscamos descomponer los datos asociados al problema. Por ejemplo, podríamos dividir a las matrices que contienen los datos originales en vectores de igual tamaño, mientras que duplicamos el resto de las constantes relacionadas en los demás procesadores (lo cual no es un desperdicio, considerando el tamaño que ocupan en memoria). Es decir, podemos descomponer a las matrices en una fila (tarea) por procesador. Llamaremos a este plan 1.

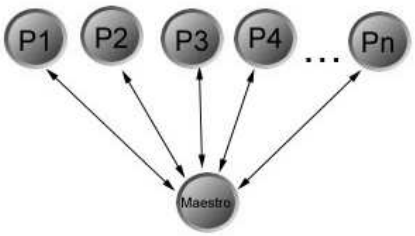


Ilustración 10: Descomposición funcional del problema del modelo de iluminación

PARTICIONAMIENTO UTILIZANDO DESCOMPOSICIÓN FUNCIONAL:

Enfocamos nuestros esfuerzos sobre los cálculos que vamos a realizar en vez de los datos sobre los que vamos a actuar:
Si pensamos de esa manera, nuestro problema podría ser resuelto de esta manera:

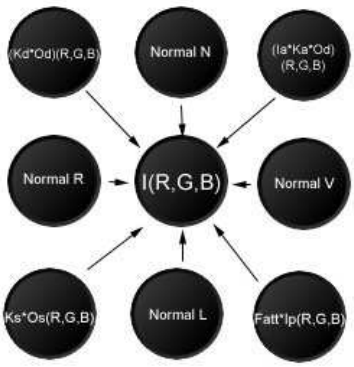


Ilustración 11: Particionamiento funcional del modelo de iluminación clásico

Donde cada esfera representa un procesador (Cada una de estas tareas se puede realizar independientemente). A esta aproximación la llamaremos plan 2.

REVISIÓN DE PUNTOS DE PARTCIONAMIENTO PARA EL PLAN 1 Y EL PLAN 2:

Plan 1:

- Tenemos más tareas que procesadores (hay flexibilidad)
- Hay redundancia de datos por procesador (No mucha)
- Las tareas son de tamaño comparable, los procesadores trabajan igual
- El problema se escala bien (Sí hay más datos se pueden usar más procesadores).

Plan 2:

- Pueden quedar procesadores sin utilizar (Tenemos solamente 8 tareas)
- No hay datos redundantes
- Algunos procesadores trabajan más que otros.
- No se escala muy bien.

Como podemos apreciar, este es un típico problema SPMD el cuál se puede solucionar mejor utilizando descomposición por dominio. El único punto débil de este enfoque es la redundancia de datos y cálculos (Veremos si vale la pena o no dejarlo así en nuestro análisis de comunicación).

COMUNICACIÓN:

Las tareas generadas por una partición son pensadas para ser ejecutadas de manera concurrente pero por lo general necesitan datos de las demás tareas para realizar un cálculo. Esto es lo que debemos estudiar.

En el plan 2, las estructuras de comunicación están definidas por la forma en que cada tarea mueve los datos de un lugar a otro. Tenemos una comunicación sincrónica ya que el cálculo de Ix no puede seguir si no ha finalizado el cálculo de los demás elementos, es estática ya que los compañeros de comunicación no cambian durante la ejecución del programa y es local porque cada tarea se comunica solamente con un vecino (Todas contra Ix). Finalmente, la comunicación es estructurada (Ix forma una estrella por la forma en que se conecta con cada tarea).

En el plan 1, la estructura de comunicación es la siguiente:

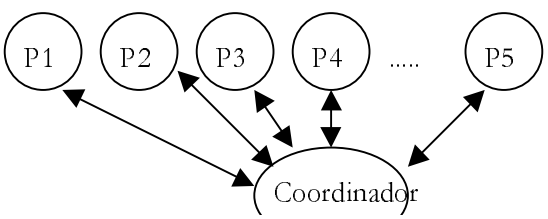


Ilustración 11: Estructura de comunicación, Plan 1.

El Coordinador reparte pedazos de todas las matrices de datos entre todos los procesadores (En pedazos del mismo tamaño) al mismo tiempo que este se queda con una porción de los datos y trabaja sobre ellos (Se oculta la latencia con trabajo útil). Hay una comunicación global (Tenemos una tarea central que recoge los resultados, después de enviar a los datos base). El envío de datos puede ser asincrónico y sin bloqueo (No hay que esperar a que llegue un reconocimiento de que llegaron los datos para enviar los siguientes), pero la recepción de estos (Tanto en el maestro como en los esclavos) debe ser sincrónica (para garantizar determinismo). Planteamos un típico modelo SPMD.

PUNTOS CLAVE EN LOS REQUERIMIENTOS DE COMUNICACIÓN:

En el plan 1 los cálculos pueden hacerse de manera concurrente, mientras que el plan 2 hay que esperar a tener todos los componentes.

En ambos planes hay que esperar a tener todas las partes antes de escribirlas en el disco duro.

El plan 1 se comunica más que el plan 2 (Hay mayor tareas por procesador. Esto se puede remediar usando aglomeración).

AGLOMERACIÓN:

Examinamos si replicamos datos y comunicaciones, a la vez que hacemos las tareas más grandes.

Si en el plan 1, en vez de asignar cada fila de las matrices de datos, asignamos una cantidad de elementos dada por el cociente del número de elementos/ # de procesadores y además replicamos algunas constantes del problema, reduciremos los costos de comunicación sin aumentar mucho los datos replicados (Manteniendo la escalabilidad y cantidad de trabajo uniforme entre cada procesador).

En el plan 2 no es conveniente una fase de aglomeración (Desperdiciaríamos la presencia de los procesadores disponibles en el sistema. Tenemos poca flexibilidad de diseño).

El plan 1 puede escalarse en computadoras paralelas, mientras que el plan 1 no.

La aglomeración de datos reduce un poco la concurrencia, pero disminuye los costos de comunicación.

Los costos de modificar el programa secuencial con el plan 1 son mínimos (No ocurre lo mismo con el plan 2).

MAPEO:

Aquí vamos a especificar en donde se va a ejecutar cada tarea (¿ Qué procesador se va a usar?)

Las estrategias de mapeo a seguir son directas en cada caso (Y en nuestro problema) vienen dadas por la división hecha en la fase de aglomeración. En el plan 1 cada procesador tiene la misma cantidad de trabajo por lo que no hace falta utilizar un algoritmo de balanceo de cargas.

Por todas las razones anteriores se implementó el plan 1 y desecharemos al plan 2.

Dependiendo de la plataforma en la cual vayamos a trabajar se deben tomar ciertos pasos:

PLATAFORMA PARALELA EN CECALCULA:

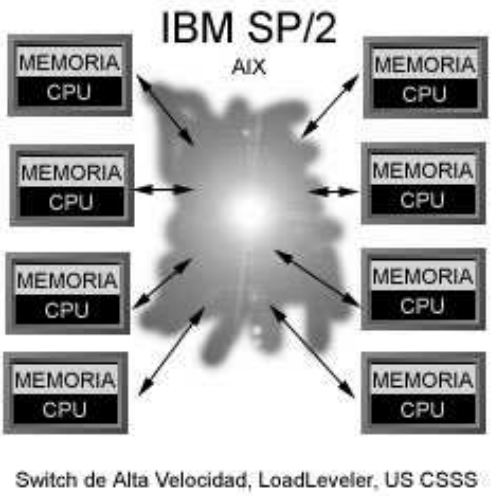


Ilustración 12: Configuración de la SP2 presente en CecalcULA

En este ambiente de trabajo, los administradores de CeCalcULA se encargarón previamente de la configuración de las herramientas de trabajo propias de un ambiente paralelo. Lo único que debe hacer el usuario es consultar la dirección Web de CeCalcULA (http://www.cecalc.ula.ve) y allí encontrará asistencia técnica en línea, así como bibliografía sobre los temas y herramientas relacionados.

Los siguientes párrafos fueron tomados del FAQ de CeCalCULA (http://www.cecalc.ula.ve/FAQ.html):

```
" P: Que significa SP2?

R: Son las iniciales de Scalable Power Parallel Systems. Los sistemas
RISC/6000 Scalable Power Parallel o SP, son una maquina de memoria
distribuida fabricada por IBM. Esta maquina esta compuesta por un
numero
determinado de nodos (procesadores con memoria interna y discos
asociados)conectados entre si por una red ethernet y por un "switch"
de alto rendimiento. Cada uno de los procesadores tiene una
arquitectura RS/6000 POWER2, cada uno es un chip superescalar capaz de
ejecutar cuatro operaciones de punto flotante por ciclo. Alcanzan un
funcionamiento pico por procesador de 266 MFLOPS (1MFLOP = 1 millón de
operaciones de punto flotante)
```

P: Cuantos procesadores, memoria RAM, capacidad de disco, etc. tiene la SP2 de CeCalCULA?

R: La SP2 con que cuenta CeCalCULA tiene 8 procesadores y la siguiente configuración:

* 64 MB RAM,
* 2 GB Disco Duro (para cada nodo),
* Un rendimiento pico de 260 MFlops (Un MFlops es un millón de operaciones de punto flotante).
* Un switch de alta velocidad (80 MBytes/sg)

- Arreglo de Discos de 18.5 Gbytes"

Una herramienta que debe dominar el usuario es el Loadleveler (Esta herramienta se encarga de balancear la carga de trabajo dentro de la maquina paralela) a la hora de ejecutar aplicaciones paralelas y secuenciales. Con el uso de Loadleveler se asegura que se utilizara de manera optima los recursos del sistema, a la vez que se compartirán recursos de manera uniforme entre los demás usuarios. Para más información de uso sobre esta herramienta, consulte las paginas Web del centro de calculo.

Citando textualmente el FAQ de CeCalCULA:

" P: Que es Load Leveler?

R: Load Leveler es un sistema despachador de procesos y balanceador de cargas. Su función básica es distribuir los trabajos de los usuarios a los diferentes procesadores de una maquina virtual paralela de tal forma que los recursos sean aprovechados al máximo, para así lograr un mejor rendimiento del sistema total.

P: Donde obtengo información sobre como utilizar Load Leveler ?

R: Puede consultar el manual en línea de Load Leveler disponible en el sitio Web de CeCalCULA. También en la Universidad de Cornell existe información sobre este programa:

http://ibm.tc.cornell.edu/ibm/pps/loadl/index.html"

Así, tanto la aplicación secuencial como la aplicación paralela utilizaron un archivo de comandos para su ejecución en la SP2:

- Contenido del archivo secuencial.cmd (Invocado con *llsubmit secuencial.cmd*)

```
# Archivo con instrucciones de ejecución para LoadLeveler
# para la aplicación secuencial
#
# Hecho Por    José Vicente Nuñez Zuleta
# Email:    jose@ing.ula.ve
# WWW:      http://www.ing.ula.ve/~jose
#  Nombre del ejecutable:
#@ executable = secuencial
#  Archivo de salida de errores
#@ error = secuencial.error
```

```
#  Nombre en la cola
#@ queue =
#  Tipo de trabajo (Pequeño, de tipo secuencial :Ssmall)
#@ class = Ssmall

# Enviar con:  llsubmit secuencial.cmd
```

- Contenido del archivo paralelo.cmd

```
#!/bin/csh
# Comandos para el Load Leveler utilizando MPI
# Nombre del trabajo
#@ job_name        = paralelo
# Directorio y usuario inicial
#@ initialdir      = /home/ingenieria/jvnunez/pasantias/paralelo
#  Archivo de salida
#@ output          = $(job_name).$(cluster).$(process).out
#  Archivo de error
#@ error           = $(job_name).$(cluster).$(process).out
#  Tipo de trabajo
#@ job_type        = parallel
#  Requerimientos
#@ requirements    = (Adapter == "hps_user")
#  Mínimo y máximo de procesadores a utilizar
#@ min_processors  = 8
#@ max_processors  = 8
#@ environment     = MP_EUILIB=us;MP_INFOLEVEL=1;MP_LABELIO=yes
#  Clase del trabajo
#@ class           = MPI
#@ queue

echo $LOADL_PROCESSOR_LIST > mpi.hosts
# Corra a paralelo, con mpirun con 8 processos, utilizando a las
maquinas # listadas en ../datos/maquinas.
mpirun -np 8 -machinefile ../datos/maquinas paralelo
```

El archivo de maquinas contiene el nombre de cada una de las estaciones de trabajo en donde pueden correr los procesos:

```
Hydra1

Hydra2

Hydra3

Hydra4
```

Para enviar nuestro programa paralelo a la cola escribiremos:

```
Llsubmit paralelo
```

Una vez corrido nuestro programa con llsubmit seremos advertidos por correo electrónico cuando su ejecución termine (Normal o anormalmente):

```
From loadl@cecalc.ula.ve Sun Jul 13 09:29:40 1997
```

```
Date: Sun, 13 Jul 1997 09:15:24 -0400
From: loadl@cecalc.ula.ve
To: jvnunez@nodo2.cecalc.ula.ve

From: LoadLeveler
Subject: nodo2.cecalc.ula.ve.903

Your LoadLeveler job step
    nodo2.cecalc.ula.ve.903.0 (secuencial )
has exited.

Status for machine nodo4.cecalc.ula.ve:
   The job step exited normally with code 80

This job step was dispatched to run 1 time.
This job step was rejected by a starter 0 times..

Submitted at: Sun Jul 13 09:15:14 1997
Exited    at: Sun Jul 13 09:15:24 1997

               Real Time:   0 00:00:10
      Job Step User Time:   0 00:00:00
    Job Step System Time:   0 00:00:00
     Total Job Step Time:   0 00:00:00

       Starter User Time:   0 00:00:00
     Starter System Time:   0 00:00:00
      Total Starter Time:   0 00:00:00
```

Otra forma de enterarnos del progreso de nuestra aplicación es con el comando *llq* (Muestra la cola de las aplicaciones ejecutadas usando Load Leveler):

```
nodo2[241] Si maestro? llq
Id                       Owner      Submitted   ST PRI Class
Running On
------------------------ ---------- ----------- -- --- ------------ --
---------
nodo4.783.0              claudio     6/2  10:33 R  50  Sxxlarge
nodo6
nodo5.902.0              sequeir     7/10 15:09 R  50  Sxxlarge
nodo7
nodo2.909.0              jvnunez     7/13 11:06 I  50  MPI(alloc)
nodo1.429.0              wilmer      3/13 13:22 RP 50  Sxxlarge

3 jobs in queue 1 waiting, 0 pending, 2 running, 0 held.
```

Por pantalla se ve por ejemplo, que el trabajo que está corriendo en el nodo 2, con identificación 909.0, del usuario jvnunez, enviado el 7/13 a las 11:06 esta detenido (Iddle, I) y es un tipo de trabajo MPI (Los trabajos enviados a la cola de ejecución también pueden ser sacados de esta utilizando el comando llcancel. Escriba man llcancel para obtener detalles).

Si sé esta programando con librerías que soportan paralelismo, se recomienda optimizar el código paralelo de acuerdo a la arquitectura sobre la que va a correr finalmente (Se hace por medio de opciones de compilación).

El uso de un archivo *Makefile* (Usado por la herramienta **Make**), puede hacerle su trabajo más fácil, ya sea a la hora de compilación, ejecución y análisis de salida de su programa:

- Contenido del archivo Makefile.sp2 para la aplicación secuencial:

```
# Este Makefile me compila la aplicacion secuencial
# Hecho por:      Jose Vicente Nunez Zuleta
# Email:       jose@ing.ula.ve
# WWW:         http://www.ing.ula.ve/~jose
CC = xlC
# Aquí "Banderas" son las banderas de optimización
BANDERAS = -O3 -qtune=pwr
RM = /usr/bin/rm
LLSUBMIT = /usr/lpp/LoadL/nfs/bin/llsubmit
COMANDOS = secuencial.cmd
VI = /usr/ucb/vi
DATOS = ../datos/salidasec_RGB.dat
secuencial: secuencial.C
   ${CC} ${BANDERAS} secuencial.C -o secuencial
limpiar:
   ( ${RM} secuencial ; ${RM} ${DATOS} )
correr:
   ${LLSUBMIT} ${COMANDOS}
salida:
   ${VI} ${DATOS}
```

- Contenido del archivo Makefile.sp2 para la aplicación paralela:

```
# Este Makefile me compila la aplicacion paralela
# Hecho por:   Jose Vicente Nunez Zuleta
# Email: jose@ing.ula.ve
# WWW:      http://www.ing.ula.ve/~jose
CC = mpicc
BANDERAS = -O3 -qtune=pwr
RM = /usr/bin/rm
LLSUBMIT = /usr/lpp/LoadL/nfs/bin/llsubmit
COMANDOS = paralelo.cmd
VI = /usr/ucb/vi
DATOS = ../datos/salidapar_RGB.dat
HOGAR_MPI=/usr/utiles/mpich
LIB=-L$(HOGAR_MPI)/lib/rs6000/ch_mpl/ -lmpi
INCLUDE=-I$(HOGAR_MPI)/include
PROCESOS = -np 8
MAQUINAS = -machinefile ../datos/maquinas
PROGRAMA = paralelo
paralelo: paralelo.C
   ${CC} ${BANDERAS} ${LIB} ${INCLUDE} ${PROGRAMA}.C -o ${PROGRAMA}
limpiar:
   ${RM} ${PROGRAMA}
correr:
   ${LLSUBMIT} ${COMANDOS}
# La forma NO OPTIMA es la siguiente:
#/usr/utiles/mpich/bin/mpirun ${PROCESOS} ${MAQUINAS} ${PROGRAMA}
salida:
   ${VI} ${DATOS}
```

Por ejemplo, para compilar el programa paralelo sólo hace falta copiar el archivo Makefile.sp2 a Makefile y escribir "Make". Una vez compilado se puede escribir "Make correr" para ejecutar la aplicación paralela (Esta orden busca la etiqueta correr y ejecuta el macro que allí esta definido).

Por omisión, el ejecutable que utiliza las librarías de MPI en CeCalCULA utiliza el subsistema de comunicación (Communication Susbsystem, CSS) que corre en los adaptadores HSP, en vez de utilizar a los protocolos TCP/IP.

PLATAFORMA DISTRIBUIDA

En este caso se instalo una red desde cero (aislada parcialmente de Internet), con la siguiente topología de bus ethernet:

Ilustración 13: Topología de Bus del ambiente distribuido

Leon es el servidor de DNS (Servidor de Dominio de Nombres), NIS (Servicios de Información de Red y NFS (Sistema de archivo de redes), mientras que Pantera y Tigre son clientes de stos servicios (Pero cada uno tiene una copia local de MPI debido a limitaciones de disco en Leon).

Es un ambiente de trabajo heterogéneo (No debido a la arquitectura, pero sí debido a al rendimiento y características físicas de los equipos).

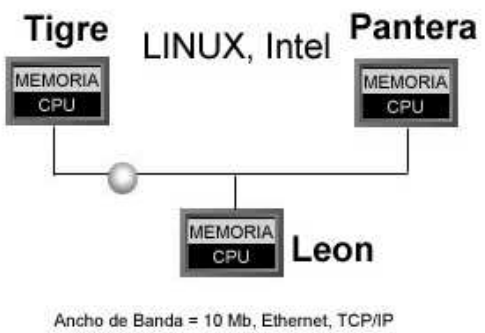
CARACTERÍSTICAS DE CADA UNO DE LOS EQUIPOS:

Leon:

- CPU: Intel 386/40 Mhz con coprocesador matemático 80387
- Memoria Ram: 8 Mb.
- Disco Duro:102 Mb.
- Sistema Operativo: Linux

Pantera:

- CPU: Intel Pentium/200 Mhz.
- Memoria Ram: 32 Mb.
- Disco Duro:502 Mb.



- Sistema Operativo: Linux

Tigre:

- CPU: Cyrix 586/133 Mhz
- Memoria Ram: 16 Mb.
- Disco Duro: 207 Mb.
- Sistema Operativo: Linux

La configuración utilizada corresponde a un ambiente distribuido de bajo costo con los servicios básicos para trabajar con el protocolo TCP/IP.

CONFIGURACIÓN DEL SERVIDOR:

Si bien la explicación de la configuración de un servidor como Leon está más allá de los objetivos originales de la pasantía de investigación se explicará someramente en que consiste cada uno de los servicios instalados, y su utilidad. En la bibliografía de esta pasantía encontrará la información adecuada que trata sobre el tema.

DNS:

DNS es una base de datos distribuida que permite la conversión de nombres a direcciones IP (y viceversa). Esto permite un control local sobre los segmentos de la base de datos en general, permitiendo que cada segmento este disponible a lo largo de toda la red utilizando un esquema de cliente - servidor. DNS utiliza "dominios" jerárquicos (los cuales se semejan a un árbol invertido) para manejar un conjunto de maquinas y servicios (Por ejemplo "*leon.felinos.ve*" significa que la maquina leon pertenece al dominio "felinos", el cual a su vez pertenece al dominio "ve").

Gracias a DNS es posible manejar un gran número de máquinas en un sistema distribuido de manera rápida y confiable (En vez de utilizar un archivo de datos para cada máquina como es el /etc/hosts).

DNS permite el utilizar el formato de nombres usado en el archivo de "maquinas"utilizado por *mpirun*.

Se configuró un dominio para la red clase C 192.168.1.0 con mascara 255.255.255.0 llamado *felinos.ve*.

NIS y NFS:

NIS es una base de datos centralizada la cual es compartida utilizando el modelo cliente servidor. NIS provee un mecanismo consistente para acceder a todos los archivos y la información de configuración se asegura que todos los usuarios puedan tener acceso a las maquinas correctas. Se utilizo NIS para compartir los siguientes archivos:

- /etc/passwd (Archivo que contiene información como el directorio hogar del usuario, palabra clave, concha, etc.)

- /etc/group (Archivo de grupos al cual pertenece. Util para compartir archivos).

- /etc/host.equiv (En este archivo definimos aquellas máquinas que pueden utilizar servicios como rlogin y rsh sin ningún tipo de autenticación. NOTA: Úsese con cuidado).

NFS provee transparencia del sistema de archivos debido a que oculta la ubicación física actual de un sistema de archivos (Es decir, los clientes "montan" ese sistema de archivos y lo tratan como si fuera local, cuando en realidad se encuentra físicamente en otro servidor).

En nuestro caso, las cuentas se encuentran físicamente en el servidor leon, pero son utilizadas de manera transparente en pantera y tigre, con lo cual se mantiene sincronización entre los archivos. La permisología de los archivos (dueños y usuarios) permanecen correctas entre esas tres computadoras gracias a NIS.

CONFIGURACIÓN DEL CLIENTE:

Debido a limitaciones de espacio, hubo que instalar dos copias separadas de MPI tanto en pantera como en tigre. El paso normal habría sido colocar una sola copia en el directorio */usr/local* de leon , para compartir luego ese directorio con las clientes. Para los detalles de configuración consulte el apendice de estas pasantías.

Luego se edito el archivo /etc/csh.cshrc tanto en tigre como en pantera para reflejar las rutas tanto de las *paginas man de mpi* como la *ubicación de los archivos binarios* (mpirun y mpicc por ejemplo):

```
Set path= $path:/usr/local/mpi/bin
Setenv $MANPATH $MANPATH /usr/local/mpi/manpath
```

Una vez hechos estos cambios el sistema puede ser probado utilizando alguno de los ejemplos que vienen con la distribución.

Para la plataforma distribuida se hicieron algunos cambios a los Makefiles (No existe loadleveler en estas plataformas, además de que el compilador es el compilador de GNU llamado gcc).

- Archivo Makefile para la aplicación secuencial:

```
# Este Makefile me compila la aplicación secuencial
# Hecho por:   José Vicente Nuñez Zuleta
Email:      jose@ing.ula.ve
# WWW:      http://www.ing.ula.ve/~jose
CC = g++
BANDERAS = -O6 -fexpensive-optimizations
DEBUG = -g
RM = /bin/rm
VI = /usr/bin/vi
DATOS = ../datos/salidasec_RGB.dat
PROG = secuencial
secuencial: secuencial.C
   ${CC} ${BANDERAS} ${PROG}.C -o ${PROG}
debug:
   ${CC} ${BANDERAS} ${DEBUG} ${PROG}.C -o ${PROG}
limpiar:
      ${RM} ${PROG}
correr:
   ./${PROG}
salida:
   ${VI} ${DATOS}
```

- Archivo Makefile para la aplicación distribuida:

```
# Este Makefile me compila la aplicación paralela
```

```
# Hecho por:   José Vicente Nuñez Zuleta
# Email: jose@ing.ula.ve
# WWW:      http://www.ing.ula.ve/~jose
CC = mpicc
BANDERAS = -O6 -fexpensive-optimizations
RM = /bin/rm
VI = /usr/bin/vi
DATOS = ../datos/salidapar_RGB.dat
HOGAR_MPI=/usr/local/mpi
LIB=-L$(HOGAR_MPI)/lib/LINUX/ch_p4/ -lmpi
INCLUDE=-I$(HOGAR_MPI)/include
paralelo: paralelo.C
   ${CC} ${BANDERAS} ${LIB} ${INCLUDE} paralelo.C -o paralelo
limpiar:
   ${RM} paralelo
correr:
   mpirun -np 2 -machinefile ../datos/maquinas.felinos paralelo
salida:
   ${VI} ${DATOS}
```

TIEMPOS DE EJECUCIÓN Y ANALISIS DE RESULTADOS

Número de Procesos	Tiempos para 3000 datos (μSeg.)			Tiempos para 30000 datos (μSeg.)			Tiempos para 240000 datos (μSeg.)		
	Cómputo	Comunic	Total	Cómputo	Comunic	Total	Cómputo	Comunic	Total
1	13883	0	13883	147024	0	147024	1155094	0	1155094
2	7044	556669	563713	73023	5187387	5260410	961119	42726025	43687144

Tabla 3: Tiempos de computo y comunicación en el ambiente distribuido según la cantidad de procesadores

Los resultados indican como el tiempo de cálculo en el procesador maestro baja considerablemente, pero los tiempos de comunicación son muy elevados lo que hace que el algoritmo paralelo tenga un desempeño pobre en comparación al algoritmo secuencial.

Algunas posibles razones que ocasionan demoras en la comunicación:

- El ancho de banda no es dedicado sino compartido entre las aplicaciones que utilizan la red.
- Hay una sobrecarga asociada al protocolo TCP debido a que este aplica corrección de errores a cada uno de los paquetes que envía (Cosa que no ocurre con UDP). Rsh ,por ejemplo, trabaja con TCP.
- El tiempo de respuesta de los CPU de las maquinas involucradas en el proceso es desigual, por lo que pueden producirse demoras en los tiempos de respuesta.

Hay una sobrecarga asociada a iniciar cada proceso en los procesadores remotos (En el ambiente distribuido, MPICH depende mucho de NFS (La escritura y lectura de datos no es local, se hace remota, aparte de que se dispone de un ancho de banda compartido con otras aplicaciones) lo cual deteriora el mucho el rendimiento).

El programa no pudo manejar 300000 datos, por lo que se rebajo el número a 240000.

Número de Procesos	Tiempos para 3000 datos (μSeg.)			Tiempos para 30000 datos (μSeg.)			Tiempos para 240000 datos (μSeg.)		
	Cómputo	Comunic	Total	Cómputo	Comunic	Total	Cómputo	Comunic	Total
1	14632	0	14632	146177	0	147177	1183435	0	1183435
2	8165	89339	97504	79765	186888	266653	591765	634421	1226186
4	5990	234378	240368	41265	64336	105601	284328	543721	828049
8	3009	2844385	2847394	25789	1443284	1469073	167450	315362	482812

Tabla 4: Tiempos de computo y comunicación en el ambiente paralelo según la cantidad de procesadores

Para dos procesadores, los resultados son igual de desalentadores que en el ambiente distribuido (El tiempo de comunicación es mayor que el tiempo de cómputo, la granularidad del problema es muy pequeña). Sin embargo, a medida que aumentamos la cantidad de datos (y el número de procesadores) vemos como la aplicación paralela supera en rendimiento a la aplicación secuencial, alcanzando su mejor tiempo con 240000 datos y 8 procesadores (482812 μSegundos contra. 1183435 μSegundos).

Se puede ver que el protocolo de comunicación propio de la SP2 es más *liviano*, lo que permite una comunicación más eficiente, además de que los tiempos de respuesta son más uniformes debido a que la arquitectura homogénea. La SP2 utiliza US CSSS (User Space Communication Susbsystem, Subsistema de Espacio de Comunicación de Usuarios). Este protocolo está pensado para aplicaciones paralelas que requieran máximo desempeño de comunicaciones. Su problema radica en que un solo proceso por nodo puede usar comunicación US.

Se puede concluir que la implementación paralela muestra sus bondades cuando el número de datos se hace grande (Y los tiempos de comunicación se hacen más o menos iguales a los tiempos de cálculo).

CONCLUSIONES

El mundo del paralelismo es un mundo relativamente nuevo, el cual está aún sujeto a mucha experimentación y desarrollo. Para el momento de trabajar en estas pasantías, aún se estaban definiendo estándares además de que se estaban evaluando las herramientas y estrategias a utilizar en la solución de problemas que puedan utilizar el paralelismo.

Sin embargo, se obtuvieron las siguientes conclusiones:

- El paralelismo es una necesidad, no una extravagancia. La tendencia hacia máquinas con más de un procesador y ambientes que aprovechen estas características serán el lugar común en los años venideros.

- Es necesaria la aplicación de una metodología en la resolución de problemas utilizando paralelismo, no en base de recetas, sino basándose en estrategias que permitan un mejor aprovechamiento de los recursos existentes.

- Dependiendo del problema y la arquitectura en donde se vaya a resolver el problema existen muchas soluciones y paradigmas que pueden ser más apropiados que otros en la resolución del problema (Pase de mensajes, Paralelismo de datos, etc.).

- A la hora de implementar una solución paralela, esta debería ser en lo posible, portable entre arquitecturas y escalable de acuerdo al tamaño del problema. Soluciones rígidas (atadas a una estrategia o arquitectura particular) pueden ser desechadas por otras menos efectivas pero más económicas.

- La mayoría de la bibliografía existente recomienda usar MPI en vez de PVM. Si bien en estas pasantías no se pudo hacer un estudio exhaustivo entre el rendimiento de estas dos herramientas en cuanto a velocidad y tolerancia a fallas, todo parece indicar que la industria de la computación abandonó a PVM para impulsar a MPI (De hecho la implementación MPI2 tiene muchas de las características de PVM, como es la creación dinámica de tareas).

- La resolución de un problema basándose en el análisis de particionamiento, comunicación, aglomeración y mapeo proporcionará una solución cercana a la optima (Por supuesto, no puede despreciarse nunca el factor de creatividad).

- Los sistemas distribuidos son una alternativa muy económica con respecto a sistemas dedicados como la SP2. Con un diseño cuidadoso de las aplicaciones puede ahorrarse costos creando un entorno productivo de desarrollo de bajo costo.

REFERENCIAS BIBLIOGRÁFICAS

- **[Foley,1994]** Introduccón a la graficación por computador. Foley James D., Dam van Andries, Feimer K. Steven, Hugher f. Jhon, Philips L. Richard. 1994. Addison Wesley Iberoamericana.

- **[Stern, 1992]** Managin NFS and NIS. Stern Hal. 1992. O'Reilly & Associates, Inc.

- **[Hung, 1993]** TCP-IP, Network Administration. Hung Graig. 1993. O'Reilly & Associates, Inc.

- **[Albitz, 1992]** DNS and BIND in a Nutshell. Albitz Paul and Liu Cricket. 1992. O'Reilly & Associates, Inc.

- **[Fisch, 1993]** Essential System Administration. Frisch Aeleen. 1993. O'Reilly & Associates, Inc.

- **[Internet-1]** Centro de Calculo (CeCalCULA). http://www.cecalc.ula.ve/

- **[Internet-2]** Maui High Performance Computing Center. http://www.mhpcc.edu/mhpcc.text.html

- **[Internet-3]** Training and Education Documents Page. http://www.epcc.ed.ac.uk/epcc-tec/documents.html

- **[Internet-4]** A List of Courses in Parallelism, Networks, Distributed Systems. http://www.cis.temple.edu/courses-para.html

- **[Internet-5]** Yahoo! - Computers and Internet:Supercomputing and Parallel Computing:Programming:Languages. http://www.yahoo.com/Computers_and_Internet/Supercomputing_and_Parallel_Computing/Programming/Languages/

- **[Internet-6]** Introduction to the T3D. http://www.epcc.ed.ac.uk/t3d/documents/t3d-intro-course/www/Intro.book_1.html

- **[Internet-7]** Designing and Building Parallel Programs. http://www.mcs.anl.gov:80/dbpp/

- **[Internet-8]** WANG'S BOOKSHELF (Parallel Computing). http://stove.cs.yale.edu:8001/parallel.html

- **[Internet-9]** Parallel_Computing_Concepts. http://pscinfo.psc.edu/training/T3D_Oct_95/PC_Concepts/PC_Concepts.html

- **[Internet-10]** MPI - The Message Passing Interface Standard. http://www.mcs.anl.gov/mpi/index.html

- **[Internet-11]** MPI. http://www.ifs.hr/ifs/razno/doc/mpi/index.html
- **[Internet-12]** Object Oriented MPI. http://www.cse.nd.edu/~lsc/research/oompi/
- **[Internet-13]** LAM / MPI Parallel Computing. http://www.osc.edu/lam.html
- **[Internet-14]** The CC++ Programming Language. http://globus.isi.edu/ccpp/
- **[Internet-15]** pC++/Sage++ Information Home Page. http://www.extreme.indiana.edu/sage/
- **[Internet-16]** Task Parallelism and Fortran. http://www.mcs.anl.gov/fortran-m/FM.html
- **[Internet-17]** On-Line Reference Material. http://oldwww.nas.nasa.gov/NAS/SciCon/refs.html
- **[Internet-18]** High Performance Fortran in Practice. http://www.cs.rice.edu/~chk/hpf-tutorial.html
- **[Internet-19]** HPF Tutorial. http://www.ifs.hr/ifs/razno/doc/hpf/
- **[Internet-20]** PVM - Parallel Virtual Machine. http://consult.ncsa.uiuc.edu/docs/consult/Doc/Software/PVM/
- **[Internet-21]** BU CAS CS 480: Introduction to Computer Graphics--- Home Page
- **[Internet-22]** Computer Graphics. http://mambo.ucsc.edu/psl/cg.html
- **[Internet-23]** Computer Graphics at Stanford University. http://www-graphics.stanford.edu/
- **[Internet-24]** University of California at Los Angeles (UCLA) Extension 3D Graphics Programming Courses. http://users.deltanet.com/~powerg/UCLA/index.html

ANEXOS

INSTALACIÓN DE MPI BAJO LINUX

La versión de MPI instalada (tanto en CeCalCULA como en nuestro sistema distribuido) se llama MPICH. Se puede conseguir una copia en (http://www.mcs.anl.gov/mpich/download.html).

Una vez obtenido el archivo mpich.tar.Z se instala siguiendo los siguientes pasos (Para una información más detallada y actualizada consulte el archivo readme que viene con el programa):

1. `gunzip -c mpich.tar | tar xovf -`

2. `cd mpich`

3. `configure`

4. `make`

Bajo Linux la instalación no tuvo ningún problema (Se probo bajo Slackware 3.1).

APLICACIONES DE PRUEBA EN MPI

Durante la realización de estas pasantías se hicieron pequeñas pruebas con MPI antes de trabajar con el problema secuencial original. Aquí se muestra el código (debidamente comentado) de algunas aplicaciones de prueba que utilizan C++ y MPI. Esto se hizo debido a que MPI no provee herramientas fáciles para la depuración, además de que buscaba ver como funcionaban las librerías.

1. Hola.C (El típico programa que sólo muestra las capacidades básicas de MPI).

```
# Este programa muestra el proceso en el cual se realiza la
# acción y la cantidad de procesos involucrados.
#include "/usr/local/mpi/include/mpi.h"
#include <iostream.h>
#include <iomanip.h>
int main(int argc, char **argv )
{
int nRango, nTam;
MPI_Init( &argc, &argv ); // Inicio a MPI
MPI_Comm_rank( MPI_COMM_WORLD, &nRango ); // Averiguo el rango que
tiene mi proceso dentro de "mundo de comunicacion"
MPI_Comm_size( MPI_COMM_WORLD, &nTam );     // Averiguo cuantos
procesos estan siendo invocados
cout<<"Hola a todos, yo soy "<<nRango<<" con un total de procesos
corriendo de "<<nTam<<endl;
MPI_Finalize();   // Finalizo a MPI
return 0;
}
```

2. Matriz2.C (Muestra como utilizar datos de tipo compuesto para el envío de datos). El envío de datos en este caso no es optimo y no aprovecha para nada la arquitectura particular de cada sistema (Para eso se utiliza MPI_Scatter).

```
/*
Este programa me muestra como pasar una matriz
a un grupo de procesadores usando MPI y datos de tipo contiguo

Hecho por Jose Vicente Nunez Zuleta
Email:   jose@ing.ula.ve
Web:  http://www.ing.ula.ve/~jose

*/

#include <iostream.h>
#include <iomanip.h>
#include "/usr/utiles/mpich/include/mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <fstream.h>

const int MAX_LINEA=30;
const int FILAS=8;
const int COLUMNAS=3;
const int MAESTRO=0;

const int bandera=1;
int error;

int main(int argc, char *argv[])
{
   int nRango,nTam,nFuente=0,nDestino,i;
   float fMatriz[][COLUMNAS]={1,2,3,
            1,2,3,
            1,2,3,
            1,2,3,
            1,2,3,
            1,2,3,
            1,2,3};
   float fVector[COLUMNAS];
   MPI_Status estado;
   MPI_Request peticion;
   MPI_Datatype fila;   // Usando este tipo envio las filas de mi
matriz

   // Inicio a MPI y creo mi tipo de datos

   MPI_Init(&argc,&argv);

   // Averiguo mi rango y el numero de procesos

   error=MPI_Comm_rank(MPI_COMM_WORLD, &nRango);
   error=MPI_Comm_size( MPI_COMM_WORLD, &nTam );


   if (error != 0)
   {
      cout <<"Error al iniciar MPI"<<endl;
   }
```

```
   // Implemento mi tipo de dato continuo
   MPI_Type_contiguous (COLUMNAS,MPI_FLOAT,&fila);
   MPI_Type_commit (&fila);

   // Comienzo a distribuir mi matriz entre los procesadores
   // , de manera Syncronica y con bloqueo
   if (nRango == MAESTRO)
   {
   if (error !=0) MPI_Finalize();
   for (nDestino=1;nDestino<nTam;nDestino++)

   MPI_Send(&fMatriz[nDestino][0],1,fila,nDestino,bandera,MPI_COMM_WOR
LD);
   for (nDestino=1;nDestino<nTam;nDestino++)

   MPI_Recv(&fMatriz[nDestino][0],1,fila,nDestino,bandera,MPI_COMM_WOR
LD,&estado);
   for (i=0;i<FILAS;i++)
      cout<<fMatriz[i][0]<<" "<<fMatriz[i][1]<<"
"<<fMatriz[i][2]<<endl;
   }
   else if (nRango > MAESTRO)
   {

   MPI_Recv(fVector,COLUMNAS,MPI_FLOAT,MAESTRO,bandera,MPI_COMM_WORLD,
&estado);
      for (i=0;i<COLUMNAS;i++) fVector[i]=nRango*fVector[i];
      MPI_Send(&fVector,1,fila,MAESTRO,bandera,MPI_COMM_WORLD);
   }

   // Libero el tipo de dato creado
   MPI_Type_free(&fila);
   MPI_Finalize();
   return(0);
}
```

3. Finalmente, este programa muestra como enviar y recibir pedazos de igual tamaño de un cierto tipo de datos a varios procesadores utilizando MPI_Scatter y MPI_Gather respectivamente.

```
/* Este programa muestra como repartir una matriz entre diferentes
procesadores
varias filas a la vez usando MPI_Scatter
para luego recibirlas usando MPI_Gather
También se muestra el manejo dinámico de memoria
*/
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <iomanip.h>
#include "/usr/local/mpi/include/mpi.h"

const int MAX=2;  // Defino el máximo de procesadores a usar
const int MAESTRO=0; // Identidad del proceso maestro
const int FILAS=16;
const int COLUMNAS=3;
int nError;

int main (int argc, char *argv[])
{
   float fMatriz[FILAS][COLUMNAS];
```

```
   float fRes[FILAS/MAX][COLUMNAS];
   int nRango,nTam,x,y;

   // Inicializo a la matriz que recibirá mis datos
   for (x=0;x<(FILAS/MAX);x++)
      for (y=0;y<COLUMNAS;y++)
         fRes[x][y]=0;

   nError=MPI_Init(&argc,&argv);
   nError=MPI_Comm_size(MPI_COMM_WORLD,&nTam);
   nError=MPI_Comm_rank(MPI_COMM_WORLD,&nRango);

   if ( nError )
   {
      cout<<"Ha ocurrido un error en MPI"<<endl;
      exit(1);
   }
   if (nTam != MAX )
   {
      cout<<"Error: Deben utilizarse MAX procesos"<<"\a"<<endl;
      MPI_Abort(MPI_COMM_WORLD,nError);
   }
   if (nRango == MAESTRO)
   {
      // Lleno de datos a la matriz
      for (x=0;x<FILAS;x++)
      {
         for (y=0;y<COLUMNAS;y++)
         {
            fMatriz[x][y]=1;
         }
      }

   }
      // Envío a la matriz (FILAS/MAX filas x COLUMNAS a MAX
procesadores
      MPI_Scatter
(&fMatriz,(FILAS/MAX)*COLUMNAS,MPI_FLOAT,&fRes,(FILAS/MAX)*COLUMNAS,MP
I_FLOAT,MAESTRO,MPI_COMM_WORLD);
      // Cada procesador coloca un numero en la matriz
      for (x=0;x<(FILAS/MAX);x++)
                for (y=0;y<COLUMNAS;y++)
                        fRes[x][y]=nRango;
   // Recibo la matriz
   MPI_Gather(&fRes,(FILAS/MAX)*COLUMNAS,MPI_FLOAT,&fMatriz,(FILAS/MAX
)*COLUMNAS,MPI_FLOAT,MAESTRO,MPI_COMM_WORLD);
   if (nRango == MAESTRO)
   {
      // Muestro la matriz resultante por pantalla
      for (x=0;x<FILAS;x++)
                {
                        for (y=0;y<COLUMNAS;y++)
                        {
                                cout<<fMatriz[x][y]<<" ";
                        }
         cout<<endl;
                }
   }
   MPI_Finalize();
return 0;
}
```

Cada uno de estos programas fue compilado utilizando una orden similar a esta:

```
Mpicc –o salida entrada.C
```

Y fue ejecutado de la siguiente manera:

```
Mpirun  –np número_de_procesos  –machinefile archivo_de_maquinas
programa
```

CODIFICACIÓN DEL MODELO DE ILUMINACIÓN

Finalmente se codificaron tanto la versión secuencial como la versión paralela del modelo de iluminación siguiendo los lineamientos de diseño anteriores y utilizando la librería de pase de mensajes MPI (Para la aplicación paralela).

ARCHIVO DE CONSTANTES COMUNES: CONSTANTES.HPP

```
/* constantes.hpp
En este archivo definimos las constantes con las que va a trabajar él
modelo de iluminación. (tanto en la implementación secuencial como en
la
paralela

Hecho por José Vicente Nuñez Zuleta.   Email:   jose@ing.ula.ve
               Web:  http://www.ing.ula.ve/~jose
Tutor:               Domingo Hernandez (dhh@ing.ula.ve)
*/

// Defino unas constantes útiles para el manejo de mi matriz

const int MAXTAREAS=8;  // Numero máximo de tareas
const int MAX_FILAS=1000000; // Numero de Filas del archivo de datos
const int MAX_COL=3;
const int MAX_LINEA=35;    // Defino el máximo de una línea a ser
leída en mi archivo

/*
Phong Bui-Tuong desarrollo un modelo de iluminación para reflectores
imperfectos (Como una manzana). En este modelo, la cantidad de luz
reflejada viene
dada por 3 componentes:

LuzReflejada=Luz Ambiental+Luz Difusa+Luz Especular

La expresion matematica que modela este comportamiento viene dada por:

Ix = IaxKaOdx + fattIpx[KdOdx(NL) + KsOsx(RV)^nExp];

donde fatt (Factor de atenuacion difuso) viene dado por:

x puede ser un componente R,G,B (Red, Green, Blue; Rojo, Verde y
Azul). Suponemos que un modelo
de 3 colores puede modelar totalmente la interacción de la luz con los
objetos (Por simplicidad).

fatt=(1/(d+d0)) (Este componente da realismo al modelo de iluminación)
```

```
N,L,R y V son vectores normalizados

*/

// ***** Definición de coeficientes para la luz Ambiental

float Ia[]={1.0,0.5,0.5};     // Intensidad Ambiental, constante para
todos los objetos
float Ka=0.30;          // Coeficiente de reflexión Ambiental, Es una
constante empírica (0,..,1)

// **** Definición de coeficientes para la luz Difusa

// Definimos las constantes para el factor de atenuación luminoso
float Ip[]={1.0,0.5,0.3};     // Intensidad de la fuente luminosa
puntual (R,G,B)
float d=1.0;            // Distancia del objeto a la fuente luminosa
(distancia que viaja la luz de la fuente puntual a la superficie)
float d0=0.5;           // Distancia al origen
float Od[]={1.0,0.5,1.0};     // Color difuso para el Rojo,Verde,Azul
float Kd=0.55;             // Coeficiente de reflexión difusa,depende
del material, (0..1)

// **** Definición de coeficientes para la luz Especular
unsigned short int nExp=200;     // Exponente de reflexión especular
del material. Depende del material simulado.
                     // Valores elevados simulan un punto de brillo
definido y enfocado.
float Ks=0.5;           // Coeficiente de reflexion especular del
material (0,..,1)
float Os[]={0.25,0.25,0.25};     // Coeficiente color especular del
objeto (Rojo,Verde,Azul)

/* Fin de las definiciones */
```

RUTINAS PARA EL CÁLCULO DE LOS VALORES DEL MODELO DE ILUMINACIÓN

```
/* Aqui estan definidas las rutinas  usadas para los cálculos del modelo de iluminación
(Secuencial y Paralelo)

Hecho por José Vicente Nuñez Zuleta
Email:  jose@ing.ula.ve
WWW:    http://www.ing.ula.ve/~jose

*/

#include <math.h>
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <strstream.h>
#include <stdlib.h>

// Esta función me calcula el producto punto entre dos vectores

void punto (const float v1[][MAX_COL],const float v2[][MAX_COL],float res[],const int
nFila)
{
    for (register int i=0;i<nFila;i++)
    {
        res[i]=0.0;
        for (register int j=0;j<MAX_COL;j++) res[i]+=v1[i][j]*v2[i][j];
    }
}

// Esta función me normaliza a los vectores dentro de mi matriz
```

```
int normalizar(float fMatriz[][MAX_COL],const int nFila)
{
    for(register int i=0;i<nFila;i++)
    {
        float mod=0.0;
        for(register int j=0;j<MAX_COL;j++) mod+=(fMatriz[i][j]*fMatriz[i][j]);
        mod=(float)sqrt(mod);
        if (mod != 0)
        {
            for(register int k=0;k<MAX_COL;k++) fMatriz[i][k]=(fMatriz[i][k]/mod);
        }
        else
        {
            cerr << "División por cero (Modulo vale 0)" << endl;
            exit (1);
        }
    }
    return (0);
}

/* Esta función levanta el archivo de datos de disco a RAM */

// Supongo que el formato de mi archivo esta correcto, no se valida el contenido

int cargar (float fMatriz[][MAX_COL],const char *pszArchivo)
{
    register int nLinea=0;
    char cLinea[MAX_LINEA];
    ifstream datos;
    datos.open(pszArchivo,ios::in);
    if (!datos.good())
    {
        cerr <<"No pude abrir el archivo de datos "<<pszArchivo<<endl;
        exit(1);
    }
    while ((!datos.eof()) || (nLinea < MAX_FILAS))
    {
        datos>>fMatriz[nLinea][0]>>fMatriz[nLinea][1]>>fMatriz[nLinea][2];
        nLinea++;
    }
    datos.close();
    return(0);
}

// Esta función me escribe los resultados en disco

int guardar_disco(const float fMatriz[][MAX_COL],const int nFilas,const char
*pszArchivo,const int nT1,const int nT2)
{
        fstream resultado;
        resultado.open(pszArchivo,ios::out); // Sobreescriba el archivo
        if (! resultado.good() )
        {
                cerr<<"No pude crear el archivo de salida "<<endl;
        exit(1);
        }
        for(register int nFila=0;nFila<nFilas;nFila++)
        {
                resultado
<<fMatriz[nFila][0]<<"\t"<<fMatriz[nFila][1]<<"\t"<<fMatriz[nFila][2]<<endl;
        }
    resultado <<"Se procesaron "<<MAX_FILAS*MAX_COL<<" datos"<<endl;
    resultado <<"Tiempo de calculo:\t\t\t"<<nT1<<" uSegundos ("<<nT1/1000000<<"
segundos)"<<endl;
    resultado <<"Tiempo de comunicacion:\t\t\t"<<nT2<<" uSegundos ("<<nT2/1000000<<"
segundos)"<<endl;
    resultado <<"Tiempo Total:\t\t\t"<<nT1+nT2<<" uSegundos ("<<(nT1+nT2)/1000000<<"
segundos)"<<endl;
        resultado.close();
    return(0);
}
/*
*****************************************************************
```

```
Las siguientes rutinas solamente se usan para fines de depuración
*****************************************************************
// Guarda el contenido de un vector a disco
int guardar_vector(const float fMatriz[],const int nFilas,const char *pszArchivo)
{
        fstream resultado;
        resultado.open(pszArchivo,ios::out); // Sobreescriba el archivo
        if (! resultado.good() )
        {
                cerr<<"No pude crear el archivo de salida "<<endl;
       exit(1);
        }
        for(register int nFila=0;nFila<nFilas;nFila++)
        {
                resultado <<fMatriz[nFila]<<endl;
        }
        resultado.close();
    return(0);
}
// Guarda el contenido de una matriz a disco
int guardar_matriz(const float fMatriz[][MAX_COL],const int nFilas,const char
*pszArchivo)
{
        fstream resultado;
        resultado.open(pszArchivo,ios::out); // Sobreescriba el archivo
        if (! resultado.good() )
        {
                cerr<<"No pude crear el archivo de salida "<<endl;
       exit(1);
        }
        for(register int nFila=0;nFila<nFilas;nFila++)
        {
                resultado
<<fMatriz[nFila][0]<<"\t"<<fMatriz[nFila][1]<<"\t"<<fMatriz[nFila][2]<<endl;
        }
        resultado.close();
    return(0);
}
// Muestra el contenido de una matriz
void mostrar_matriz(const float fMatriz[][MAX_COL],const int nFila)
{
    for (register int i=0;i<nFila;i++)
    {
        for (register int j=0;j<MAX_COL;j++)
            cout<<fMatriz[i][j]<<" ";
        cout<<endl;
    }
}


// Muestra el contenido de un vector
void mostrar_vector(const float fVector[],const int nFila)
{
    for (register int i=0;i<nFila;i++)
        cout<<fVector[i]<<endl;
    cout<<endl;
}
*/
```

CODIFICACIÓN DEL MODELO SECUENCIAL

```
/* Este programa me muestra la ejecución secuencial del calculo
de los valores RGB de reflexión de luz para un punto usando
un modelo de iluminación
Hecho por José Vicente Nuñez Zuleta
Email:  jose@ing.ula.ve
Web:    http://www.ing.ula.ve/~jose
Tutor:  Domingo Hernandez (dhh@ing.ula.ve)

*/

#include <math.h>
#include "../datos/constantes.hpp"    // Archivo con la definición de constantes del
modelo
#include "../datos/util.hpp"      // Rutinas de calculo usadas por el modelo
#include <time.h>
```

```
#include <sys/time.h>

void main(int argc,char *argv[])
{
        float  R[MAX_FILAS][MAX_COL],
           N[MAX_FILAS][MAX_COL],
           L[MAX_FILAS][MAX_COL],
           V[MAX_FILAS][MAX_COL],
           RV[MAX_FILAS],     // Producto Normal entre R y V
           NL[MAX_FILAS],     // Producto Normal entre N y L
           I[MAX_FILAS][MAX_COL]; // Aqui guardamos los valores RGB
    struct timeval tiempo1,tiempo2; // Estructura para medir el tiempo
        int nDiftiempo;            // Diferencia de tiempos
    register int nFila;       // Fila siendo procesada

    // Estos valores solo se calculan una vez asi que los guardamos aqui
    float KaIa[]={Ka*Ia[0],Ka*Ia[1],Ka*Ia[2]};
    float Ipfatt[]={Ip[0]/(d+d0),Ip[1]/(d+d0),Ip[2]/(d+d0)};

    // Cargo los datos en memoria

    cargar(R,"../datos/r.dat");
    cargar(V,"../datos/v.dat");
    cargar(N,"../datos/n.dat");
    cargar(L,"../datos/l.dat");

    // Comienzo a medir el tiempo
    gettimeofday(&tiempo1, (struct timezone* )0);

    // Normalizo a mi arreglo de vectores
    normalizar(R,MAX_FILAS);
    normalizar(V,MAX_FILAS);
    normalizar(N,MAX_FILAS);
    normalizar(L,MAX_FILAS);

    //Depuracion
    //guardar_matriz(L,MAX_FILAS,"../datos/L.dat");

    // Calculo los productos punto necesarios para el modelo

    punto(R,V,RV,MAX_FILAS);
    punto(N,L,NL,MAX_FILAS);

    // Depuracion: Escriba el producto punto
    //guardar_vector(NL,MAX_FILAS,"../datos/NL.dat");

    // Comienzo con los calculos propios del modelo de iluminacion
    for(nFila=0;nFila<MAX_FILAS;nFila++)
    {

    I[nFila][0]=KaIa[0]+(Ipfatt[0]*Kd*Od[0]*NL[nFila])+Ks*((float)pow(RV[nFila],nExp));


    I[nFila][1]=KaIa[1]+(Ipfatt[1]*Kd*Od[1]*NL[nFila])+Ks*((float)pow(RV[nFila],nExp));

    I[nFila][2]=KaIa[2]+(Ipfatt[2]*Kd*Od[2]*NL[nFila])+Ks*((float)pow(RV[nFila],nExp));

    }
    // Tomo el tiempo final
    gettimeofday(&tiempo2, (struct timezone* )0);
    nDiftiempo=((tiempo2.tv_sec-tiempo1.tv_sec)*1000000)+(tiempo2.tv_usec-
tiempo1.tv_usec);
    guardar_disco(I,MAX_FILAS,"../datos/salidasec_RGB.dat",nDiftiempo,0);
    cout<<"\a";
}
```

Una salida de ejemplo para la aplicación secuencial seria la siguiente:

(Se omite la salida de los datos, sólo se muestran los tiempos).

```
Se procesaron 3000 datos

Tiempo de calculo:    13883 uSegundos (0 segundos)
```

```
    Tiempo de comunicación:   0 uSegundos (0 segundos)

    Tiempo Total:             13883 uSegundos (0 segundos)
```

CODIFICACIÓN DE LA IMPLEMENTACIÓN PARALELA

```
/* Este programa me muestra la ejecución Paralela del calculo
de los valores RGB de reflexión de luz para un punto usando
un modelo de iluminación, paralelizado  utilizando MPI
Hecho por José Vicente Nuñez Zuleta
Email:  jose@ing.ula.ve
Web:    http://www.ing.ula.ve/~jose
Tutor:  Domingo Hernandez (dhh@ing.ula.ve)

implementación utilizando MPI

*/

#include <iostream.h>
#include <iomanip.h>
#include <time.h>      // Hay que ver cuanto tiempo demora
#include <sys/time.h>     // la ejecución del programa paralelo
#include <stdlib.h>       // Manejo de errores
#include "../datos/constantes.hpp"    // Constantes del modelo de iluminación
// Escoja un include segun su plataforma
#include "/usr/utiles/mpich/include/mpi.h" // Include para MPI (SP2)
//#include "/usr/local/mpi/include/mpi.h" // Include para MPI (LINUX)
#include "../datos/util.hpp"      // Rutinas de calculo usadas por el modelo de
iluminacion

//Depuracion
//#include <stdio.h>

int main(int argc,char *argv[])
{

/* Ahora defino variables útiles para mi ejecución paralela */
const int MAESTRO=0;   // Esta es la identificación del maestro
int nTareas,nTarea,    // Numero de Tareas
    nRango,     // Identificador de la tarea
    nDestino,   // Identificador del procesador destino
    nFuente,    // Identificador de la tarea originadora del mensaje
    nError;     // Código de retorno de error para inicio de tareas en MPI
register int nFila;    // Fila actual siendo procesada

/*

Esta constante me indica cuantos procesos se van a ejecutar
Asegure que MAX_FILAS/MAXTAREAS produce un numero entero

*/
const int nPedazo=MAX_FILAS/MAXTAREAS; // Cuantas filas corresponden a cada proceso

if ((MAX_FILAS%MAXTAREAS) != 0)
{
    cerr<<"Asegurece que MAX_FILAS/MAXTAREAS produce un numero entero"<<endl;
    cerr<<"No puedo repartir una cantidad decimal de filas entre procesadores"<<endl;
    cerr<<"\a";
    exit(1);
}

/* Comienzo la ejecución paralela */

nError=MPI_Init(&argc,&argv);         // Inicio a MPI
nError=MPI_Comm_size(MPI_COMM_WORLD,&nTareas); // # de Tareas activas
nError=MPI_Comm_rank(MPI_COMM_WORLD,&nRango); // Rango entre las tareas

if (nError != 0) // Si hay errores detenga el sistema
{
    cout<<"Hubo un error al iniciar MPI"<<endl;
    MPI_Abort(MPI_COMM_WORLD,nError); // Aborto la ejecucion de MPI
    exit(1);
}
```

```
if ( nTareas =! MAXTAREAS)
{
    cerr<<"Este programa fue diseñado para ejecutarse en "<<MAXTAREAS<<" nodos"<<endl;
    MPI_Abort(MPI_COMM_WORLD,nError); // Aborto la ejecución de MPI
    exit(1);
}

/*  Estos valores solo se calculan una vez así que los guardamos aquí
    (Cada tarea calcula estos valores, incluyendo el maestro)
*/

float KaIa[]={Ka*Ia[0],Ka*Ia[1],Ka*Ia[2]};
float Ipfatt[]={Ip[0]/(d+d0),Ip[1]/(d+d0),Ip[2]/(d+d0)};

// Note como las dimensiones son distintas, con respecto
// al programa secuencial
float   RV[nPedazo],   // Producto Normal entre R y V
        NL[nPedazo];   // Producto Normal entre N y L

// Matrices en donde están los datos originales y los datos parciales
float   fRp[nPedazo][MAX_COL],
    fVp[nPedazo][MAX_COL],
    fNp[nPedazo][MAX_COL],
    fLp[nPedazo][MAX_COL],
    fIp[nPedazo][MAX_COL],
    fR[MAX_FILAS][MAX_COL],
    fN[MAX_FILAS][MAX_COL],
        fL[MAX_FILAS][MAX_COL],
        fV[MAX_FILAS][MAX_COL],
        fI[MAX_FILAS][MAX_COL];    //Valores RGB producto del modelo de iluminación.
struct  timeval tiempo1,tiempo2;       // Estructura para medir el tiempo
int     nTcal, nTcom;                  // Diferencia de tiempos (Cálculos y comunicación)


//Depuracion
//char temp[20];

/* Cargo los datos básicos en memoria, sólo en el maestro */

if (nRango == MAESTRO) {
    cargar(fR,"../datos/r.dat");
    cargar(fV,"../datos/v.dat");
    cargar(fN,"../datos/n.dat");
    cargar(fL,"../datos/l.dat");

    // Comienzo a medir el tiempo
    gettimeofday(&tiempo1, (struct timezone* )0);
}

/* Reparto los datos necesarios entre los trabajadores */

MPI_Scatter
(&fR,nPedazo*MAX_COL,MPI_FLOAT,&fRp,nPedazo*MAX_COL,MPI_FLOAT,MAESTRO,MPI_COMM_WORLD);
MPI_Scatter
(&fV,nPedazo*MAX_COL,MPI_FLOAT,&fVp,nPedazo*MAX_COL,MPI_FLOAT,MAESTRO,MPI_COMM_WORLD);
MPI_Scatter
(&fN,nPedazo*MAX_COL,MPI_FLOAT,&fNp,nPedazo*MAX_COL,MPI_FLOAT,MAESTRO,MPI_COMM_WORLD);
MPI_Scatter
(&fL,nPedazo*MAX_COL,MPI_FLOAT,&fLp,nPedazo*MAX_COL,MPI_FLOAT,MAESTRO,MPI_COMM_WORLD);

if (nRango == MAESTRO)
{
    gettimeofday(&tiempo2, (struct timezone* )0); // Tomo el tiempo final
    nTcom=((tiempo2.tv_sec-tiempo1.tv_sec)*1000000)+(tiempo2.tv_usec-tiempo1.tv_usec);
}
// Tomo el tiempo del inicio de los calculos
if (nRango == MAESTRO) gettimeofday(&tiempo1, (struct timezone* )0);

// Normalizo los vectores antes de realizar los calculos
normalizar(fRp,nPedazo);
normalizar(fVp,nPedazo);
normalizar(fNp,nPedazo);
normalizar(fLp,nPedazo);
```

```
//Depuracion
//sprintf(temp,"../datos/fLp%d.dat",nRango);
//guardar_matriz(fLp,nPedazo,temp);

// Calculo el producto punto (necesario para el modelo)
punto(fRp,fVp,RV,nPedazo);
punto(fNp,fLp,NL,nPedazo);

//Depuracion
//sprintf(temp,"../datos/NL%d",nRango);
//guardar_vector(NL,nPedazo,temp);

// Calculo los valores RGB del modelo de iluminacion
for(nFila=0;nFila<nPedazo;nFila++)
{
         fIp[nFila][0]=KaIa[0]+(Ipfatt[0]*Kd*Od[0]*NL[nFila])+Ks*(pow(RV[nFila],nExp));
         fIp[nFila][1]=KaIa[1]+(Ipfatt[1]*Kd*Od[1]*NL[nFila])+Ks*(pow(RV[nFila],nExp));
         fIp[nFila][2]=KaIa[2]+(Ipfatt[2]*Kd*Od[2]*NL[nFila])+Ks*(pow(RV[nFila],nExp));
}
if (nRango == MAESTRO)
{
    gettimeofday(&tiempo2, (struct timezone* )0); // Tomo el tiempo final
    nTcal=((tiempo2.tv_sec-tiempo1.tv_sec)*1000000)+(tiempo2.tv_usec-tiempo1.tv_usec);
}

// Tomo el tiempo del la recoleccion de datos (comunicacion)
if (nRango == MAESTRO) gettimeofday(&tiempo1, (struct timezone* )0);

/* Recolecto el resultado de las operaciones de las demas matrices */
MPI_Gather(&fIp,nPedazo*MAX_COL,MPI_FLOAT,&fI,nPedazo*MAX_COL,MPI_FLOAT,MAESTRO,MPI_COMM
_WORLD);
if (nRango == MAESTRO)
{
    gettimeofday(&tiempo2, (struct timezone* )0); // Tomo el tiempo final
    nTcom+=((tiempo2.tv_sec-tiempo1.tv_sec)*1000000)+(tiempo2.tv_usec-tiempo1.tv_usec);
}

if ( nRango == MAESTRO )  // Soy el maestro
{
    guardar_disco(fI,MAX_FILAS,"../datos/salidapar_RGB.dat",nTcal,nTcom);
    cout<<"\a";
}

// Termino a MPI
MPI_Finalize();
return(0);
}
```

La salida del programa paralelo, corriendo en un sistema distribuido con 2 procesos:

```
Se procesaron 3000 datos

Tiempo de calculo:                      7044 uSegundos (0 segundos)

Tiempo de comunicación:                 556669 uSegundos (0 segundos)

Tiempo Total:                          563713 uSegundos (0 segundos)
```