CONVENIO INSTITUCIONAL
FUNDACIÓN PARA LA FORMACIÓN DE INVESTIGACIÓN EN VENEZUELA (FIVE)
UNIVERSIDAD DE MÁLAGA
UNIVERSIDAD EZEQUIEL ZAMORA
ESPAÑA - VENEZUELA

La Universidad que Siembra

PROGRAMA DEL TERCER CICLO
DOCTORADO EN INFORMATICA

MEMORIA DE ACTIVIDAD INVESTIGACIÓN TUTELADA

Curso 2002 - 2003

Estudio de distintas técnicas de distribución automática de datos en multiprocesadores de memoria compartida - distribuida

Autor: Darjeling Silva.

Tutores: Juan López, Maria de los Ángeles González

Año 2003

"*El paralelismo es inherente a la naturaleza y a la forma como el hombre resuelve sus problemas cotidianos*."

FERNANDO ROJAS MORALES

Resumen

El estudio de los multiprocesadores hoy en día está orientado hacia el desarrollo de programas y aplicaciones para eliminar el trabajo tedioso de manejarlos. Estos multiprocesadores son computadoras con dos o más unidades de procesamiento que trabajan sobre una memoria común bajo un control integrado. El procesamiento de datos y cómputos dentro de ellas se realiza en lo que se denomina el paradigma de la programación paralela o paralelismo. Su objetivo es conseguir que se ejecute un programa en menos tiempo usando varios procesadores. Si el programa es grande se dividirá en programas o problemas más pequeños, por lo que se repartirán entre los procesadores disponibles. De manera de ir entrelazando los datos y los cómputos para darle un significado sencillo al programa. Y así analizar la paralelización y especificar como se distribuirán los datos con la finalidad de reducir significativamente los tiempos de ejecución de un programa automáticamente requiriendo para ello conocimientos tanto hardware como en software.
El reto principal de los programadores es proporcionar al usuario un método sencillo para obtener programas paralelos con la misma facilidad con la que se desarrollan los programas secuénciales. Ya que el programador para paralelizar lo hace manualmente, lo que implica un problema a la hora de distribuir los datos y los cómputos apropiadamente, lo que reduciría significativamente los tiempos de ejecución. De ahí que el estudio de la paralelización sea una opción interesante para conseguir paralelizar eficientemente los códigos, con la mínima intervención del usuario.
El estudio que a continuación se presenta es de tipo documental, donde se localizo la información, para dar respuesta al problema de la paralización automática. Que tiene como objetivo desarrollar compiladores y sistemas operativos lo suficientemente inteligentes para que sean capaces de, a partir del programa secuencial que suministra el programador, generar la versión paralela más eficiente que se ejecuta en el multiprocesador, minimizando tanto como sea posible la iteración del usuario, para una determinada arquitectura paralela.
Existe una variedad de arquitecturas paralelas, en este trabajo nos centraremos en la arquitectura NUMA donde el tiempo de acceso no es el mismo para todas las direcciones de memoria, por lo que su acceso a memoria es compartida físicamente distribuida. Actualmente los modernos multiprocesadores se basan en este tipo de arquitecturas, en las que la generación de código paralelo es muy sensible al uso apropiado de la jerarquía de memoria. En trabajos previos tales como los de Anderson, Lam, García, Ayguadé, Labarta, Navarro y Zapata sean centrado en este problema, dotando al compilador de la capacidad de analizar el patrón de acceso a las referencias de memoria para acercar los datos que consume cada procesador al nivel de memoria más cercano.
La paralelización automática se enmarca dentro de la distribución automática de datos, como también la de explotar la localidad. Nos referimos a está ultima como la organización de las distribuciones de iteraciones y datos que facilitan el alojamiento de los datos en las memorias locales de los procesadores, es decir el procesador realiza la operación consumiendo los datos que requiere directamente de la memoria local. Lo antes expuesto condujo a la consulta de las distintas técnicas de distribución automática de datos en multiprocesadores de memoria compartida – distribuida.

Palabras claves: Paralelización automática, Arquitectura NUMA, Programación paralela, Distribución de iteraciones y datos, Localidad de los datos.

Estudio de distintas técnicas de distribución automática de datos en multiprocesadores de memoria compartida - distribuida

Autor: Darjeling Silva
Directores: Juan López, Maria de los Ángeles González

Índice General

Índice de Figuras

1.- Introducción.

Actualmente la dificultad de distintos problemas tanto en las áreas de la ciencia, la ingeniería y de los procesos de datos, hace necesario el estudio de nuevas estrategias de distribución de datos en máquinas de alto rendimiento compuestas por un gran número de procesadores capaces de trabajar cooperativamente para resolver un problema computacional. El objetivo es el de mejorar el rendimiento de programas desarrollados bajo el paradigma de la programación paralela en máquinas multiprocesador. Y facilitar la tarea del desarrollo de estos programas paralelos al programador. Estos multiprocesadores son equipos masivamente paralelos (MPPs por sus siglas en inglés). Actualmente son las computadoras más potentes en el mundo, y pueden combinar desde unos cientos de procesadores hasta miles en un solo gabinete, a través de una red especial que conecta de centenares a miles de gigabytes de memoria. El enorme potencial de los MPPs los hace ideales para el tratamiento de problemas computacionales difíciles.

Estas máquinas trabajan con procesamiento paralelo lo cual consiste en dejar que varias unidades de procesamiento (como computadoras independientes) resuelvan un problema extenso, y ha surgido como una tecnología clave en la computación de nuestros días.

Los multiprocesadores se han dividido en diversos tipos según la configuración de memoria (multiprocesadores de memoria compartida, multiprocesadores de memoria distribuida y multiprocesadores de memoria compartida - distribuida), en estos momentos nos atañe tratar los multiprocesadores de memoria distribuida y los de memoria compartida - distribuida. La computación distribuida forzosamente necesita integrar trabajo colaborativo a través de una red de comunicación a la que se conectan las computadoras y por medio de la cual cooperan en la tarea asignada. El hecho de que más organizaciones necesiten redes locales de alta velocidad, en combinación con computadoras modernas puede producir un conjunto agregado que podría (en ciertos casos y tareas) exceder el rendimiento de un MPP. Además se han hecho varios experimentos de interconexión de MPPs para lograr un desempeño inigualable. Los sistemas de computación distribuida son una colección de

procesadores interconectados por una red de transferencia de información en la cual cada procesador posee su propio espacio de memoria y otros periféricos.

Para un procesador en particular sus recursos son locales, en tanto que otros procesadores y sus recursos son remotos. El conjunto de un procesador y sus recursos generalmente recibe el nombre de nodo. En teoría un multiprocesador tiene una potencia de cálculo N, donde este valor está dado por el número de procesadores que lo integran. En la práctica el resultado puede ser inferior a N puesto que existe cierta sobrecarga en el sistema y retrasos en las comunicaciones que impiden acercarse suficientemente a dicho valor asintótico teórico. Sin embargo se tiene un muy buen incremento en la capacidad combinada del sistema. En un sistema multiprocesador con memoria compartida-distribuida entre los nodos, el tiempo de acceso a los datos es diferente según si dichos datos se encuentran en la porción de memoria correspondiente al nodo local o en un nodo remoto. Es decir, son sistemas NUMA (donde el tiempo de acceso no es el mismo para todas las direcciones de memoria), dicha falta de uniformidad en el acceso se hace visible explícitamente al programador, permite controlar a través de mecanismos software la localización de los datos [22]. En los sistemas NUMA, cada nodo contiene uno o varios procesadores con caches privadas y un módulo de memoria que almacena una parte determinada del espacio compartido de direcciones. Esta parte del espacio de direcciones puede accederse desde cualquier nodo, aunque el tiempo de acceso variará según si el acceso se produce desde el nodo local o desde un nodo remoto.

Pese a la simplicidad de implementación de esta clase de sistemas, su rendimiento se ve seriamente perjudicado debido a la latencia de la red de comunicaciones. Para minimizar el número de accesos remotos, debe prestarse especial atención a la distribución inicial de las páginas de memoria en cada uno de los nodos, una tarea que depende del patrón de acceso a la memoria de la aplicación. Esta tarea debe ser llevada a cabo por el sistema operativo, por el compilador o por el programador, lo que incrementa el coste de desarrollo. También puede mejorarse la localización incorporando elementos hardware, a través del uso de caches consistentes.

Sin embargo, los multiprocesadores paralelos continúan sin estar al alcance de muchas organizaciones debido a sus elevados costes en hardware, mantenimiento y programación. Una alternativa de menor coste ampliamente utilizada y consolidada son las

redes ya que al conectarse a través de Internet, los multiprocesadores y redes de ordenadores personales dispersas geográficamente abre una nueva vía para mejorar el rendimiento de ciertas aplicaciones.

El compilador se puede encargar de adaptar el código secuencial escrito por el programado, al multiprocesador sobre el que se ejecuta, lo que puede facilitar el proceso de programación de ciertas aplicaciones. Con ello se reduce la participación del usuario en la definición de los elementos que determinan su problema. El propósito de este proyecto es estudiar la importancia de distribuir los datos de forma automática en multiprocesadores de tipo NUMA.

Los paralelizadores automáticos actuales han alcanzado cierto grado de madurez y se han convertido en la herramienta más cómoda para generar una versión paralela a partir del código secuencial. Pero se debe tener en cuenta que un programa paralelizado pierde prestaciones debido a que estos paralelizadores no explotan la localidad. Por eso es necesario incorporar un nuevo módulo de compilación que sea capaz de generar las distribuciones de iteraciones y datos óptimos para un código. Al paralelizar los códigos manualmente aplicando la distribución de datos apropiada a cada caso, se puede reducir significativamente los tiempos de ejecución de los códigos paralelizados automáticamente para arquitecturas NUMA. No obstante, la paralelización manual implica invertir una gran cantidad de tiempo en la paralelización. De ahí que la paralelización automática sea una opción interesante para conseguir paralelizar eficientemente los códigos, con la mínima intervención del usuario.

El documento está organizado en la sección 1 se da una introducción al trabajo elaborado para el segundo año Doctoral de tipo documenta y bibliográfico. En la sección 2 se analizará las redes de interconexión de los procesadores entre sí y como se programa dicho multiprocesador. En la sección 3 se realizará una lectura y compendio de los diferentes métodos propuestos con respecto a las técnicas desarrolladas para resolver el problema de la distribución automática de datos en arquitecturas tipo NUMA. Luego, se tomarán cada uno de los métodos estudiados y se realizará una discusión y comparación entre ellos, como también las aportaciones y limitaciones de cada técnica. Por ultimo se darán las conclusiones y trabajos futuros.

2.- Arquitecturas Paralelas y su Programación.

La creciente demanda de memoria y CPU por las aplicaciones computacionales ha originado el desarrollo de procesadores cada día más rápido y con mejores prestaciones. Con esta evolución los computadores y el estudio de arquitecturas paralelas han sido un paso inevitable en la consecución de mayores velocidades de procesamiento.

Un multiprocesador se puede definir como una computadora que contiene dos o más unidades de procesamiento que trabajan sobre una memoria común bajo un control integrado (Sistemas MPPs; Massively Parallel Processor o procesador masivamente paralelo) [22]. En tal sentido, si el sistema de multiprocesamiento posee procesadores de aproximadamente igual capacidad, estamos en presencia de multiprocesamiento simétrico; en el otro caso se habla de multiprocesamiento asimétrico. Todos los procesadores deben poder acceder y usar la memoria principal. De acuerdo a esta definición se requiere que la memoria principal sea común aunque puedan existir pequeñas memorias locales en cada procesador. Podemos establecer una clasificación de estos multiprocesadores según la configuración de memoria.

2.1.- Clasificación según la memoria.

2.1.1.- Memoria Compartida. (shared-memory multiprocessors), también llamados multiprocesadores estrechamente acoplados (tightly coupled multiprocessors). Son sistemas con múltiples procesadores que comparten un único espacio de direcciones de memoria. Cualquier procesador puede acceder a los mismos datos, al igual que puede acceder a ellos cualquier dispositivo de entrada/salida. El sistema de interconexión más empleado para estos casos, es el de bus compartido (shared-bus). El solo hecho de tener muchos procesadores conectados a un único bus tiene el inconveniente de limitar las prestaciones del sistema a medida que se añaden nuevos procesadores. La razón es la saturación del bus, es decir, su sobre utilización; en un sistema de bus compartido, se produce un fenómeno de contienda entre los diferentes dispositivos y procesadores para obtener el control del bus. Finalmente hablar de memoria compartida es hablar de multiprocesadores donde todos los

procesadores pueden acceder a toda la memoria. Para esto, varios procesadores pueden acceder a la misma posición de memoria y en el mismo momento. En la Figura 1. Podemos observar este tipo de multiprocesador.

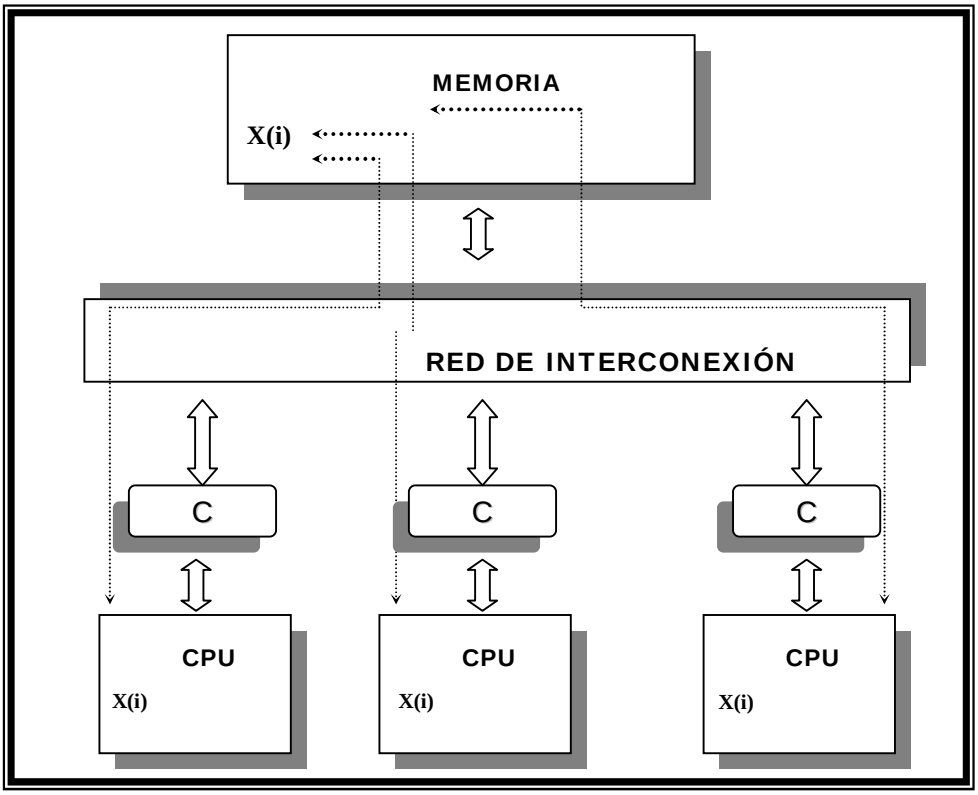

Figura 1. Esquema de un multiprocesador de memoria compartida.

En los multiprocesadores de memoria compartida todos los procesadores pueden acceder de forma directa a todas las posiciones de memoria del sistema. El tiempo de acceso que se requiere a una posición de memoria es uniforme para cada procesador, de ahí la denominación de arquitectura UMA (Uniform Memory Access); en estos sistemas cada procesador tiene acceso directo a una sola memoria compartida. Todas las ubicaciones de la memoria son equidistantes (en cuanto a tiempos de acceso) a cada procesador [1]. La mayoría de los sistemas UMA incorporan caché para eliminar las disputas de la memoria pero este mecanismo no se ve desde las aplicaciones. La programación de este tipo de máquinas es relativamente sencilla, pero en contrapartida son menos escalables ya que la memoria global sólo permite la conexión de un número reducido de procesadores. Ejemplo de este tipo de arquitectura son el SGI Power Challenge, Cray Y-MP, Cray C-90, etc [16].

2.1.2.- Memoria Distribuida [1]. (distributed-memory multiprocessors), también denominados multiprocesadores vagamente acoplados (loosely coupled multiprocessors). Se caracterizan porque cada procesador sólo puede acceder a su propia memoria. Se requiere la comunicación entre los nodos de proceso para coordinar las operaciones y mover los datos entre los distintos módulos de memoria. Los datos pueden ser intercambiados, pero no compartidos. Dado que los procesadores no comparten un espacio de direcciones común, no hay problemas asociados con tener múltiples copias de los datos, y por tanto los procesadores no tienen que competir entre ellos para obtener sus datos. Ya que cada nodo es un sistema completo por sí mismo (incluso sus propios dispositivos de entrada/salida si son necesarios), el único límite práctico para incrementar las prestaciones añadiendo nuevos nodos, esta dictado por la topología empleada para la interconexión entre nodos. De hecho, el esquema de interconexión (anillos, matrices, cubos,...), tiene un fuerte impacto en las prestaciones de estos sistemas. Además de la complejidad de las interconexiones, una de las principales desventajas de estos sistemas, como es evidente, es la duplicación de recursos caros como memoria, dispositivos de entrada/salida, que además están desocupados en gran parte del tiempo. Al hablar de sistema vagamente acoplados también estamos hablando de un multiprocesador donde la memoria del sistema está distribuida entre todos los procesadores del sistema, como se muestra en la Figura 2.

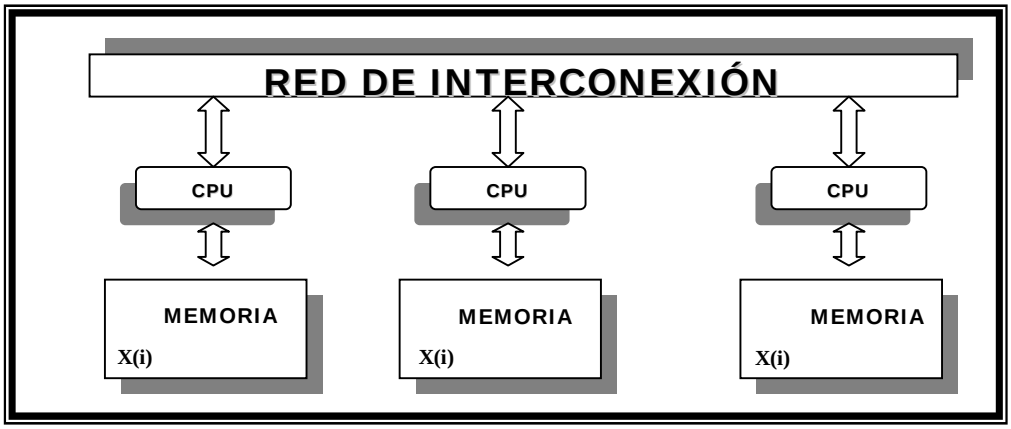

Figura 2. Esquema de un multiprocesador de memoria distribuida.

En estos sistemas cada procesador tiene acceso directo a un módulo de memoria local, es decir, a un espacio de dirección privado. El acceso a posiciones de memoria no locales se realiza a través de la red de interconexión. El tiempo de acceso a una posición de memoria es no uniforme porque depende de si dicha posición de memoria es local o no al

procesador que lo requiere, de ahí que a este tipo de arquitectura se le denomine NUMA (Non Uniform Memory Access). Los sistemas con acceso a memoria no uniforme tienen una memoria física compartida distribuida. Cada partición de esta memoria se ata directamente a un nodo pero se puede acceder a ella por procesadores en otros nodos vía red de interconexión. Así, los tiempos de acceso a la memoria difieren dependiendo de sí la ubicación requerida es local al nodo o remota a éste. Este nivel de complejidad agregado puede ser ocultado por el software de la aplicación. Para hacer mejor uso del hardware, el programador debe tomar la arquitectura en consideración. La memoria caché se usa entre procesadores y memoria local así como entre nodos. Cuando la coherencia de caché en mantenida a nivel del hardware, se llaman ccNUMA.

El modelo de comunicación de NUMA entre cada procesador es por pase de mensajes, el cual se ha de establecer de forma explícita. Este tipo de máquinas es más difícil de programar, pero en contrapartida presentan una alta escalabilidad. Ejemplos de este tipo de arquitecturas son el IBM SP2, CM-5 (connection Machine), Fujitsu AP1000/3000, Intel Paragon, Cray T3D, SGI Origin 2000 (cc-NUMA), etc [16].

2.1.3.- Memoria compartida – distribuida [1]. (distributed shared memory). Las clasificaciones anteriores son en cierto modo idealizadas, ya que a menudo las actuales máquinas se van mezclando; es decir, se trata de ir combinando las ventajas de los dos modelos expuestos anteriormente, esto es, la escalabilidad del modelo distribuido y la facilidad de la programación del modelo compartido.

Un ejemplo es un computador donde la memoria está físicamente repartida entre los nodos del sistema (cada nodo está compuesto por 2 procesadores), en el que solo hay un espacio de direcciones y todos los procesadores pueden acceder a toda la memoria. Teniendo en cuenta que un acceso a la memoria del mismo nodo es más rápido que un acceso a la memoria de otro nodo, lo que se busca en este caso es la manera de aprovechar al máximo la localidad de los datos.

En la figura 3 se muestra un ejemplo de cómo las memorias se hallan distribuidas físicamente en nodos diferentes junto con su procesador, y se ha implementado un mecanismo hardware que permite a los procesadores ver un espacio único y global de direcciones.

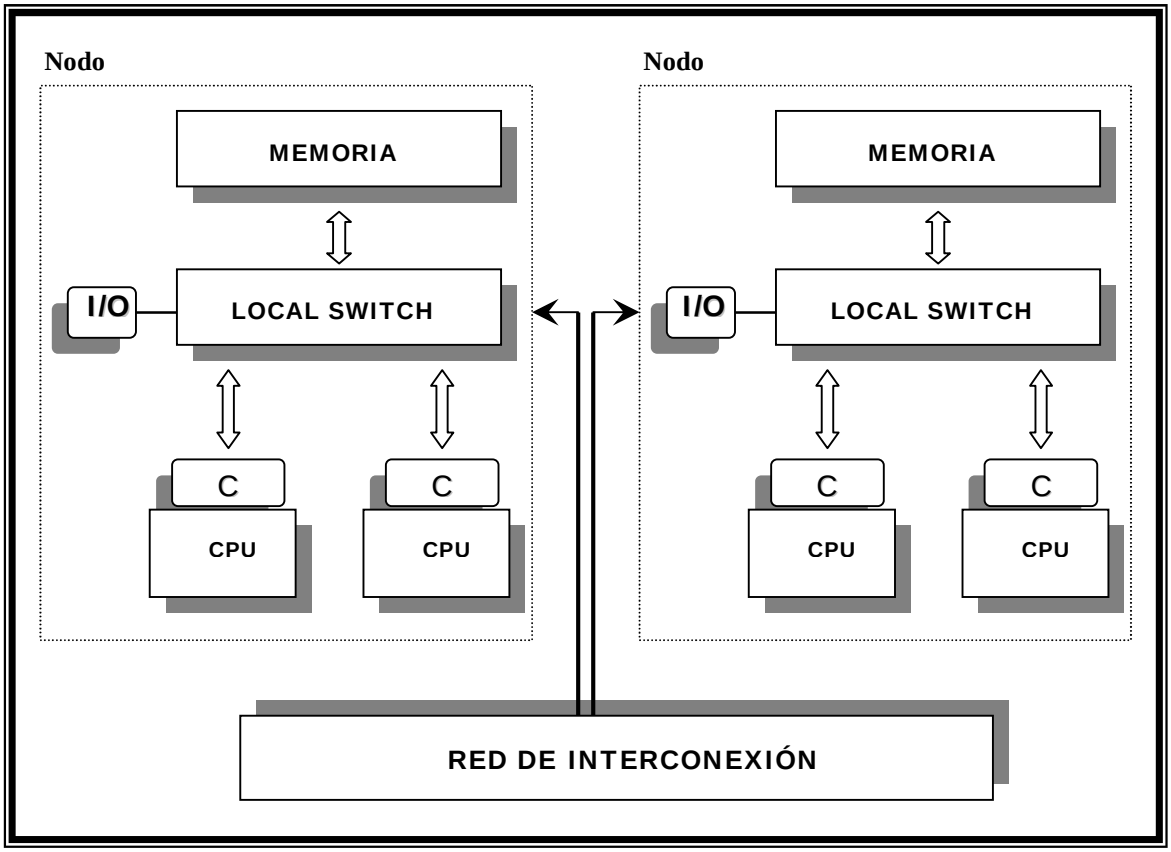

Figura 3. Esquema de un multiprocesador de memoria Compartida / Distribuida.

En este sentido, cuando la memoria solo tiene acceso a la caché son denominadas COMA (Cache – Only Memory Architecture). Son sistemas con acceso si la memoria caché provee un solo espacio físico de direcciones pero los tiempos de acceso varían dependiendo de si la ubicación de la memoria requerida está en la caché local o en una remota. El software de la aplicación puede ignorar la arquitectura del sistema ya que la máquina se comporta en forma muy parecida a una máquina UMA con caché. Por tanto, estas máquinas con un espacio de dirección dinámico en el cual las memorias locales son tratadas en su gestión como memorias cache. El ejemplo clásico es la KSR (Kendall Square Reserch) ver figura 4 . Coma simple [35] o Simple Coma, en un intento para superar los problemas de los convencionales ccNUMA y COMA, aprovechando las ventajas que tiene COMA del hardware, que son replicación automática y migración de datos, pero con una simplicidad de hardware comparable a un ccNUMA, así reduce el costo general y el tiempo de la máquina. Una máquina COMA simple es una arquitectura híbrida de hardware/software en que muchas de las operaciones complejas del hardware COMA convencional se han movido a

software, pero con hardware suficiente para operaciones de funcionamiento críticas, como la coherencia de datos. La realización más importante en COMA Simple es que la Unidad del Manejo de la Memoria (MMU: Memory Management Unit) en un procesador puede ejecutar la misma función que el hardware requerido para construir la memoria de atracción en un COMA contemporáneo: En un hardware COMA una referencia a la memoria es mostrada al sistema de atracción de memoria y la dirección se etiqueta para ubicar (sí hay) datos correspondientes en la memoria del nodo local.

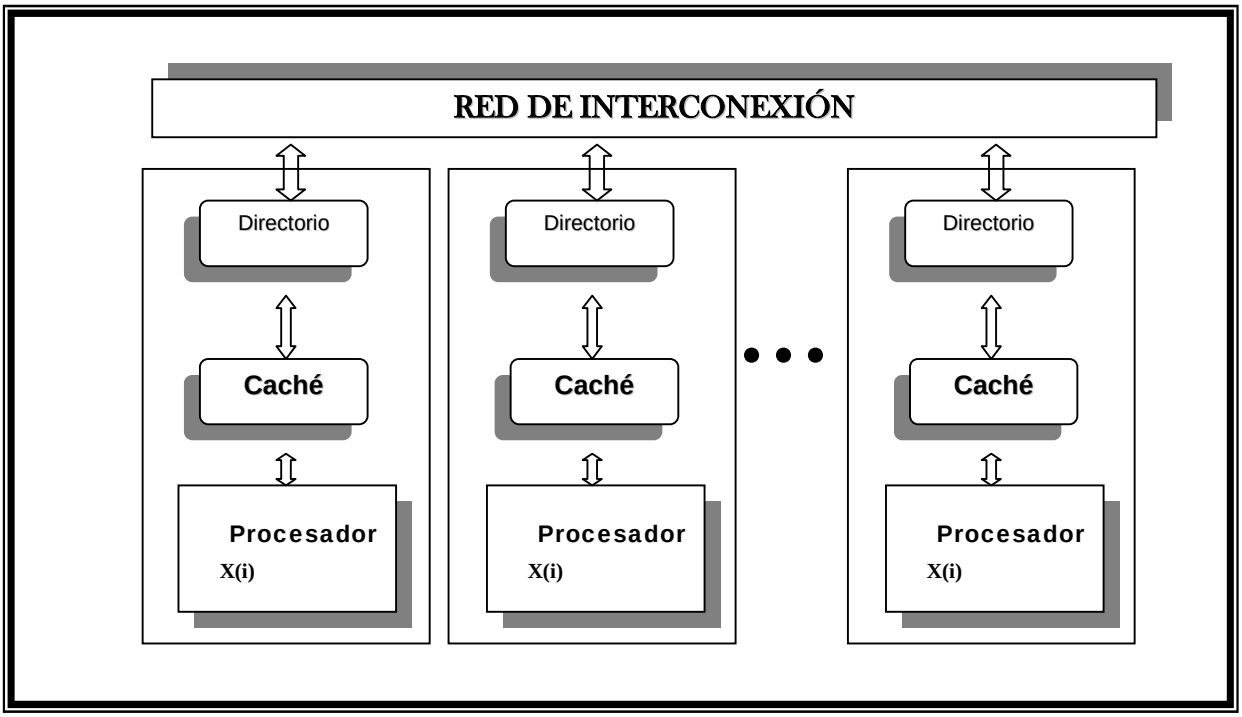

Figura 4. Modelo de multiprocesador COMA (KSR)

2.2.- Modelos de programación paralelas.

En esta sección se va a hacer referencia a las soluciones más utilizadas y extendidas en la programación paralela. En cada solución el proceso de paralelización se realiza de forma diferente, una de las interrogantes más comunes es cómo diseñar un algoritmo paralelo; el problema es el de descomponer el trabajo en problemas más pequeños. Entonces, estos pequeños problemas son asignados a los procesadores para trabajar simultáneamente. Esta descomposición se puede dar en dos grupos:

a) Descomposición de Dominio.

La descomposición de dominio o paralelismo de dato, los datos son divididos en pedazos de aproximadamente del mismo tamaño y entonces repartido entre los diferentes procesadores. Además se mantiene un simple flujo de control. Un algoritmo de paralelismo de datos consiste en una secuencia de instrucciones elementales aplicadas a los datos. SPDM (Simple – Program – Multiple – Data), sigue este modelo donde el código es idéntico sobre todos los procesadores.

b) Descomposición Funcional.

Frecuentemente, la descomposición de dominio no es muy eficiente en algunos algoritmos paralelos. Este es el caso cuando las partes de datos asignados a diferentes procesos requieren cantidades diferentes de tiempo para el procesamiento. El rendimiento del código está limitado por la velocidad de los procesos más lentos. En estos casos en la descomposición funcional o paralelismo de tarea el problema se descompone en un largo número de pequeñas tareas, que son asignadas a los procesadores cuando se encuentren disponibles.

A continuación mencionamos los modelos de programación paralela más relevantes en la actualidad:

2.2.1.- Modelo de programación paralela semiautomática (HPF y OpenMP).

Estos modelos actúan como una capa de software de más alto nivel, ya que permiten al programador trabajar con un espacio único de direcciones, aunque la memoria se halle distribuida. El programador en este caso es el que decide cómo se han de distribuir los datos, aunque no se encargue de forma explícita del movimiento de los datos entre procesadores, que ello implique. Su objetivo, es producir un código paralelo eficiente en un corto tiempo de desarrollo. Dos de estos modelos son: HPF y OpenMP.

a. **HPF** (High Performance Fortran) [10, 21, 13]**:** Es un lenguaje de programación estándar orientado al paralelismo de datos que proporcione una mayor facilidad de uso, portabilidad y eficiencia en el desarrollo de aplicaciones paralelas [22]. Este lenguaje formalmente definido en 1993, estuvo altamente influenciado por iniciativas pioneras en el campo del paralelismo de datos, como Viena-Fortran y Fortran-D [21], por tanto es una extensión de Fortran 90 de alto nivel. Esté lenguaje está dirigido a máquinas paralelas con memoria distribuida (aunque con espacio de memoria global),

por lo que emplea el modelo descomposición de dominio para ir dividiendo los datos en pedazos del mismo tamaño e ir repartiéndolos entre los diferentes procesadores. Siendo ésta su principal característica ya que permite al programador distribuir los datos o problemas a través de los procesadores para intentar minimizar el tiempo de ejecución. La programación con HPF permite a los programadores utilizar el potencial del paralelismo, sin entrar dentro de los detalles de bajo nivel del pase de mensajes y sincronización. Cuando un programa HPF es compilado, el compilador asume la responsabilidad de organizar la distribución de las operaciones y los datos en una máquina paralela, reduciendo enormemente el tiempo y esfuerzo en el desarrollo de programas paralelos. En fin, es un lenguaje muy simple de escribir y depurar ya que es portable, tanto los viejos como los nuevos códigos.

b. **OpenMP** (Application Program Interface (API)) [12, 25, 27, 28, 37]**:** OpenMP ofrece una interfaz independiente de la máquina y el sistema operativo y aspira a convertirse en el principal estándar en la paralelización de códigos C/C++ y Fortran sobre multiprocesadores de memoria compartida [12]. Esta interfaz no es una herramienta de paralelización, esto es, no permite detectar zonas de paralelismo o existencia de dependencias en el código, y es el programador el que debe asegurar la correcta aplicación de las directivas para que el programa se ejecute satisfactoriamente [27]. OpenMP está formado por un conjunto de directivas del compilador, librerías de rutinas y variables de entorno que permiten expresar, de forma sencilla, el paralelismo.

Las directivas ofrecidas por OpenMP extienden la funcionalidad del lenguaje sobre el que se aplican (Fortran, C/C++), empleando un modelo de programación SPMD (Single Program Múltiple Data) [28]. Utiliza básicamente una descomposición funcional a la hora de ir distribuyendo los datos en pequeñas tareas, de forma tal que dichas tareas son asignadas a cada procesador según se encuentren disponibles. El código a ejecutar por los procesadores es el mismo, aunque cada uno de ellos trabaje sobre una parte del total de los datos. Para ello las directivas incluyen construcciones de reparto de trabajo entre procesadores (por ejemplo iteraciones de un lazo), primitivas de sincronismo y método para compartir datos o hacerlos privados a los procesadores. El control de la ejecución del programa se realiza mediante funciones

específicas y variables de entorno, con las cuales se especifican aspectos de la ejecución como por ejemplo el número de procesadores o el tipo de planificación de los lazos.

En las especificaciones de OpenMP, no se incluye directivas relacionadas con las distribuciones de los datos entre los procesadores, siendo éste un aspecto que queda a potestad del sistema multiprocesador específico. En la mayoría de los multiprocesadores de memoria compartida distribuida actuales, como es el caso de SGI Origin 2000 la distribución de los datos es gestionada por el sistema operativo [6]. El usuario puede decidir la distribución inicial de los datos de dos maneras diferentes. Una de ellas es mediante el procedimiento conocido como first-touch por el que el dato se ubica en la memoria local del procesador que lo escribe en primer lugar. La otra es mediante el uso de directivas nativas, proporcionadas por el fabricante, que, completando a las de OpenMP, permiten las distribuciones regulares más habituales de los datos como bloques (BLOCK) o cíclica (CYCLIC). Una vez emplazados los datos en una distribución inicial, el sistema operativo gestiona la migración de estos, en función de la demanda de los datos por parte de los procesadores. El usuario dispone generalmente de variables de entorno que posibilitan la selección de umbrales que hacen más o menos agresiva la política de migración y replicación de los datos. Una limitación importante de este método, ya que el sistema operativo trabaja a nivel de página, y por tanto el tamaño de ésta limita la granularidad de las distribuciones o de la migración de los datos [29].

OpenMP está basado en un paradigma de thread. Un programa ejecutado, es denominado un proceso y es alojado en su propio espacio de memoria por el sistema operativo cuando el programa es cargado en la memoria. Dentro de un proceso, pueden existir múltiples threads. Un thread es una secuencia activa de ejecución de instrucciones dentro de un proceso. Los threads comparten el mismo espacio de memoria dentro de un proceso y pueden acceder a las mismas variables. Tienen la ventaja de permitir a los procesos ejecutar múltiples tareas, aparentemente de forma concurrente [36].

El paradigma thread es una opción lógica para multiprocesadores de memoria compartida. El concepto está basado en el modelo fork-join de la computación paralela. Un thread maestro ejecuta secuencialmente hasta que se encuentra con una directiva y se produce una bifurcación con nuevos threads. Estos threads pueden ser

distribuidos y ejecutados en diferentes procesadores, reduciendo el tiempo de ejecución logrando que más ciclos de procesos estén disponibles por unidad de tiempo. Los resultados de cada ejecución de threads pueden luego ser combinados. Un usuario puede establecer el número de threads creados para regiones paralelas estableciendo la variable de entorno omp num threads, o también puede establecerlo usando una llamada a la librería omp set num threads. En la figura 5 se muestra la ejecución de un modelo simple de un programa en OpenMP [28].

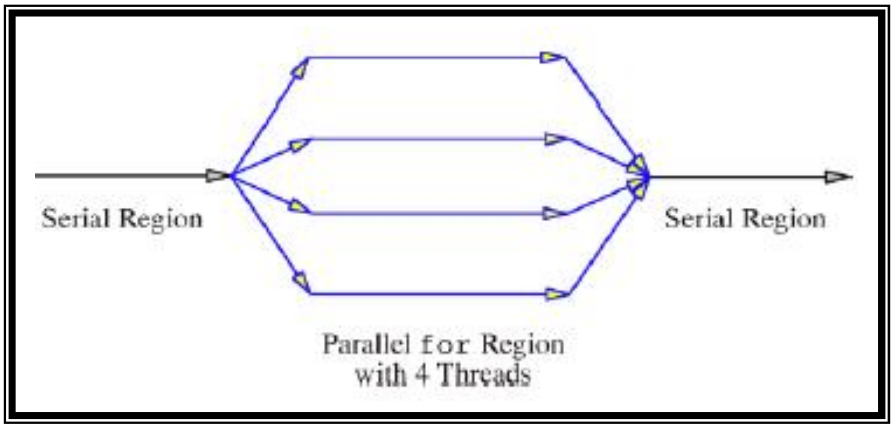

Figura 5. Flujo de un programa en un modelo de ejecución en OpenMP.

2.2.2. Modelo de programación paralela automática (Polaris, SUIF).

Los paralelizadores automáticos son herramientas, que en tiempo de compilación y sin intervención del usuario, se encargan de generar un código paralelo a partir del código secuencial de entrada. Los paralelizadores automáticos desarrollados, tanto en el ámbito comercial como en el académico, han demostrado un notable éxito en la paralelización, ya que sus prototipos académicos como lo son Polaris y SUIF son capaces de paralelizar códigos secuenciales relativamente complejos. A continuación mostraremos las características más significativas de estos paralelizadores:

a. **Polaris** [11, 15, 16, 39]: Es un prototipo académico desarrollado en la Universidad de Illinois que nació con el objetivo de superar las limitaciones existentes en otros paralelizadores automáticos a la hora de la transformación de códigos numéricos. Básicamente este prototipo descompone los códigos en pequeñas tareas, es decir, se realiza una descomposición funcional y las tareas se ejecutan independientemente,

comunicándose cuando éstos lo necesite. Polaris parte de un fichero fuente en Fortran 77 y tras convertirlo en una representación interna [15] (IR), realiza la transformación de dicho código a través de dos módulos básicos denominados front-end y back-end (ver figura 6).

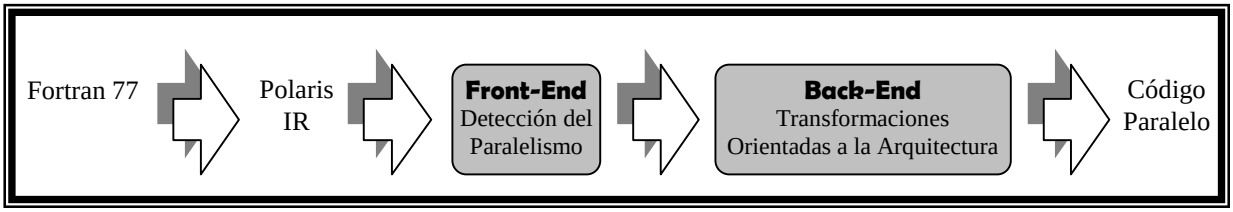

Figura 6. Paralelización de un código Fortran 77 con Polaris

El módulo front-end se encarga de detectar el paralelismo implícito, a nivel de lazo, existente en el código fuente [11]. Con la información generada por el front-end, el módulo back-end genera el código final realizando transformaciones dependientes de la arquitectura sobre la que será ejecutado el programa. Las arquitecturas soportadas por la versión actual de Polaris incluyen máquinas de memoria distribuida como Cray T3D, de memoria compartida como SGI Power Challenge, y de memoria compartida – distribuida como SGI Origin 2000 y Cray T3E. Polaris ha sido probado en una amplia colección de programas de prueba como el Perfect Benchmark y el SPEC95fp, demostrando en la mayoría de los códigos unas mejores prestaciones que las de paralelizadores comerciales como PFA [14].

b) **SUIF** (Stanford University Intermediate Format): Es un sistema de recopilación creado en la Universidad de Stanford que proporciona una infraestructura orientada al desarrollo de nuevas etapas de compilación [19]. Dicho sistema pone especial énfasis en el rehúso de técnicas que ya se hayan desarrollado, ofreciendo un entorno que facilita el desarrollo de nuevas técnicas y permite de forma sencilla interrelacionar diferentes etapas o módulos del compilador. El núcleo de SUIF realiza tres funciones importantes [34]:

- Define la representación intermedia de programas. Esta representación se utiliza tanto para transformaciones de alto nivel del programa así como para análisis bajo nivel y optimizaciones.
- Proporciona funciones de acceso o manipulación de la representación intermedia. Ocultar los detalles de bajo nivel de representación.
- Estructura la interfaz entre los pasos del recopilador. Los pasos de SUIF son los programas separados que se comunican vía archivos. El formato de estos archivos es igual para todas las etapas de compilación. El sistema apoya la experimentación permitiendo datos definidos por el usuario en anotaciones.

El sistema de SUIF es una infraestructura recopiladora en una representación intermedia del programa. La infraestructura del compilador SUIF tiene las características dominantes siguientes [34]:

- Un diseño modular que permite combinar la flexibilidad de diversos componentes (las representaciones del programa y los pasos del recopilador).
- Una representación extensible del programa permite a los usuarios capturar y poner en ejecución el programa en C. Ejecutando así la representación del programa mientras que la jerarquía del objeto y los usuarios puedan agregar más pasos para satisfacer sus propias necesidades.

En la paralelización automática de códigos SUIF la realiza en dos etapas. La primera etapa de análisis donde localiza el paralelismo disponible en el código combinándolo con la potente capacidad de análisis interprocedural que posee. Durante la segunda etapa se realiza la optimización del código paralelo y la generación del mismo. En esta etapa el sistema incorpora transformaciones de lazos y datos orientadas a aumentar la granularidad del paralelismo así como a mejorar el comportamiento de la memoria y eliminar puntos de sincronización innecesarios [18].

SUIF incluye la descomposición de datos y cómputo para las máquinas compartidas y distribuidas, es decir, utiliza la descomposición funcional y la de dominio, de manera que el recopilador aproveche tanto el paralelismo de tarea como el paralelismo de datos (híbrido, la combinación de los dos anteriores) produciendo así un código paralelo eficiente en un corto tiempo [3]. Este recopilador es capaz de procesar códigos Fortran y C, generando un código de salida en C con anotaciones SPMD

(Single Program Múltiple Data) que puede ser procesado con compiladores de C nativos en una gran variedad de arquitecturas de memoria compartida como SGI Power Challenge, multiprocesadores Digital 4800, y Kendall Square KSR-1.

2.2.3.- Modelo basado en Pase de Mensajes (MPI, PVM)

Se definen como un conjunto de procesos con su propio espacio de memoria, pero pueden comunicarse con otros procesos mediante el envío y la recepción de mensajes a través de la red de interconexión. Por ende el programador es el responsable de la comunicación y sincronización entre los procesadores entre los que se distribuye el trabajo, es decir, el programador se encarga de paralelizar el código de forma completamente manual, asumiendo un control total sobre éste.

Las arquitecturas paralelas que soportan un modelo de paso de mensaje suelen ser arquitecturas de memoria distribuida. A continuación estudiaremos dos librerías estándar como lo son MPI y PVM:

a) **MPI** (Message Passing Interface) [24, 25, 31]: Es un sistema de pase de mensajes portátil y estándar creado por un grupo de investigadores de la industria y la academia para funcionar en una amplia variedad de computadores paralelos, especialmente en máquinas con una arquitectura de memoria distribuida [31]. Este modelo de comunicación es ampliamente usado en la computación paralela. Su objetivo principal es lograr la portatibilidad a través de diferentes máquinas. Los programas escritos en MPI no deben requerir cambios en el código fuente cuando sea migrado de un sistema a otro. Explícitamente, el estándar no especifica cómo se inicializan los programas o qué debe hacer el usuario con su entorno para lograr ejecutar un programa MPI. Sin embargo, una implementación puede requerir una configuración previa antes de poder ejecutar la llamada a alguna rutina MPI. Entre las ventajas de MPI se encuentra la disponibilidad de varios modos de comunicación, los cuales permiten al programador el uso de buffers para el envío rápido de mensajes cortos, la sincronización de procesos o el solape de procesos de cómputo con procesos de comunicación. Esto último reduce el tiempo de ejecución de un programa paralelo, pero tiene la desventaja de que el programador debe ser más cuidadoso para evitar la corrupción de mensajes. MPI incluye una rutina de iniciación denominada *MPI_Init*.

Además, existe la rutina *MPI_Finalize* que permite limpiar todos los estados MPI. Una vez que esta rutina es llamada ninguna otra rutina (excepto *MPI_Init*) puede ser llamada [25].

Un ejemplo de la ejecución de un programa escrito en MPI comienza por descomponer un problema en un número de procesos especificados por el usuario. Basado en la posibilidad de descomponer los problemas en problemas más pequeños, utilizando dos tipos de descomposiciones o paralelismo el funcional y el de datos. El paralelismo funcional, en distintas máquinas hace distintas tareas (dependiendo de su capacidad). En el paralelismo de datos la información es distribuida a todas las tareas de los distintos procesadores. Cada proceso se comunica con otras instancias del programa, corriendo en el mismo procesador o en diferentes. La comunicación básica consiste en enviar y recibir datos de un proceso a otro. Esta comunicación toma lugar en una red de trabajo que conecta a los procesadores en un sistema de memoria distribuida. En la figura 7 se muestra un modelo de ejecución simple de MPI [31].

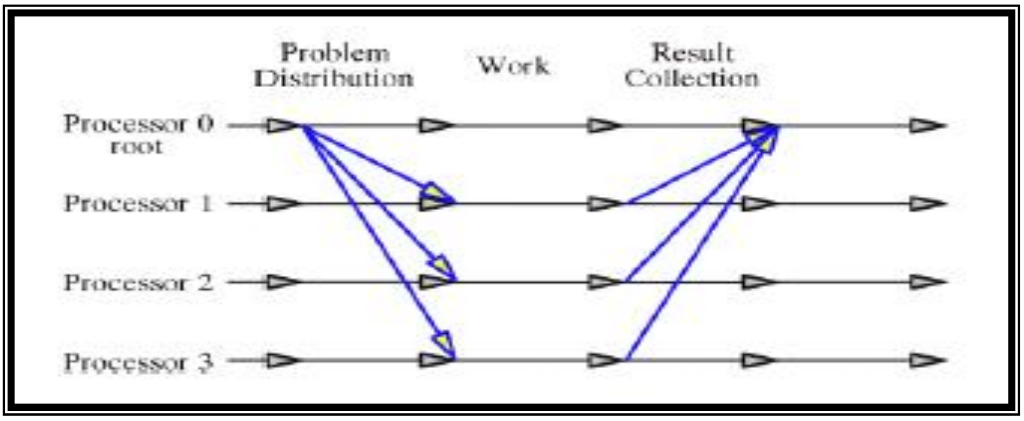

Figura 7. Flujo de un Programa en un Modelo de ejecución bajo MPI.

b) **PVM** (Parallel Virtual Machine) [17, 30]: Formalmente la podemos definir, como un conjunto de programas o librerías que convierten un conjunto de máquinas heterogéneas en un único multicomputador o máquina virtual paralela. Este software permite que una red de computadoras heterogéneas pueda utilizarse como un recurso computacional concurrente, coherente y flexible. Este software se desarrolló en 1989 en ORNL (Oak Ridge National Laboratory) para uso interno. En 1991

trasciende y se difunde su utilización, básicamente en ambientes académicos y dedicado a resolver aplicaciones científicas que demandaban gran consumo de procesador. Con la información obtenida de dichas experiencias, en 1993 PVM es re-escrito completamente y se transforma en un software de dominio público de uso masivo [30]. Su objetivo principal es Construir un sistema de computación paralela que estuviera disponible en una red local. Además, PVM al implementar las aplicaciones distribuidas usando el paradigma de pase de mensajes implementa dos tipos de descomposiciones la funcional (cada tarea desempeña una función diferente) y la de dominios (todas las tareas se comunican con los procesadores de forma explícita, pero cada una conoce y actúa sobre una parte de los datos del problema) [17]. Este software provee al desarrollador una capa de servicios de alto nivel que simplifican la programación de aplicaciones distribuidas. Las funcionalidades más importantes son [17]:

- Administración dinámica de un conjunto de estaciones de trabajo
- Pasaje de Mensajes entre procesos en ejecución en diferentes procesadores
- Sincronización entre procesos concurrentes
- Soporte para heterogeneidad (procesadores de diferente arquitectura (INTEL, SPARC, RS6000, ALPHA, etc.) corriendo diferentes sistemas operativos (Linux, Windows NT, Solaris, AIX, DIGITAL UNIX, etc.)
- Soporte para multiprocesadores

Si bien PVM es una excelente herramienta que facilita el desarrollo de aplicaciones distribuidas a ser ejecutadas en redes de computadoras, es una herramienta de bajo nivel. Esto significa que es una herramienta que se puede utilizar no sólo para implementar aplicaciones distribuidas "llave en mano", sino que además puede ser utilizada para desarrollar herramientas o utilidades (de más alto nivel) que a su vez sirvan para simplificar aún más el desarrollo de aplicaciones distribuidas.

PVM es una abstracción de un sistema que no existe, es decir, una máquina paralela virtual, que utilizará todos los recursos de todos los sistemas integrados en la red local, pero utilizando un modelo de diseño más confortable y simplificado. En realidad, es un modelo de máquina virtual que permite al programador diseñar programas para una máquina multiprocesador, ocultando al mismo la arquitectura de la red y las características del material que la compone. Para inicializar y ejecutar PVM

deben configurarse dos variables de entorno. La primera PVM_ROOT indica el lugar donde está instalado PVM. La otra variable PVM_ARCH indica la arquitectura del host desde donde se está invocando PVM y de esta manera seleccionar los archivos ejecutables más apropiados. Un diagrama ejemplar del modelo PVM se muestra en la figura 8 y una vista arquitectónica del sistema PVM, destacando la heterogeneidad de las plataformas apoyadas por PVM, se muestra en la figura 9 [30].

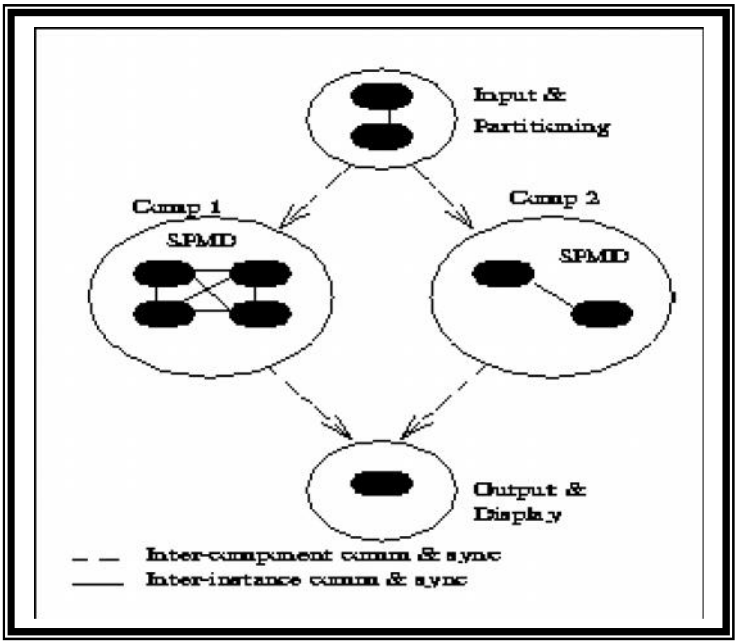

Figura 8. Descripción de un sistema PVM.

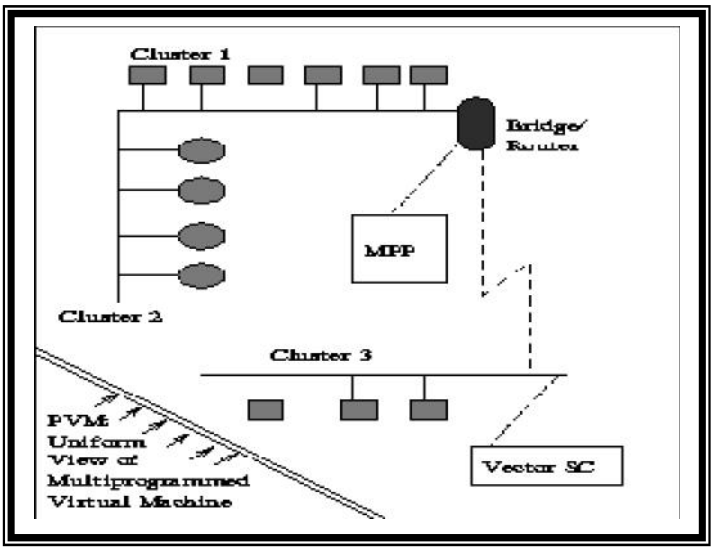

Figura 9. Descripción arquitectónica del sistema PVM.

3.- Distribución automática de datos.

Explorar las técnicas de distribución automática de datos en los paralelizadores actuales nos lleva a enfrentarnos con la idea de crear un método que sea automático, es decir crear un compilador que sin la intervención del programador pueda extraer la información del código secuencial y generar una versión paralela que se ejecute sobre cualquier multiprocesador con arquitectura compartida distribuida.

La programación de multiprocesadores con arquitectura distribuida se basa en el paradigma de pase de mensajes. La comunicación se realiza a través de primitivas que explícitamente controlen la partición y la descomposición de los datos.

Hennessy [20] realiza un breve recorrido por los principales inconvenientes de la programación paralela: El obstáculo al trabajar con arquitecturas paralelas no suele ser el precio de los procesadores, ni los defectos de las topologías de la red de interconexión, ni la falta de disponibilidad de los lenguajes de programación apropiados: la dificultad está en que pocas aplicaciones importantes están escritas para ser ejecutadas eficientemente en un sistema multiprocesador. Como es más difícil encontrar aplicaciones que aprovechen los múltiples procesadores, el reto es aún mayor para grandes multiprocesadores. Por tanto, los programas de procesamiento en paralelo son más difíciles de desarrollar que un programa en secuencial.

El hecho de escribir programas para multiprocesadores que sean más rápidos es una tarea ardua, especialmente cuando se incrementa el número de procesadores, se debe realizar una analogía, analizando la carga adicional de la comunicación para una tarea hecha por un procesador comparada con la carga adicional para una tarea hecha por un grupo de procesadores, especialmente a medida que se incrementa el número de procesadores.

Otra razón del por qué es difícil escribir programas de procesamiento en paralelo es que el programador tiene que conocer bastante información sobre el sistema. Por lo que el programador de procesamiento en paralelo necesita conocer, al menos hasta ahora, la estructura del sistema con el que trabaja para escribir programas más rápidos y capaces de utilizar varios procesadores. Además, estos programas paralelos hechos a medida no son portables a otros multiprocesadores. Aunque este segundo obstáculo es menos importante,

se pone de manifiesto un nuevo obstáculo: la ley de Amdhal (la aceleración que puede alcanzar un computador paralelo puede estar limitada significativamente por la existencia de una pequeña fracción del código inherentemente secuencial que no puede paralelizarse). Por lo que recuerda que hasta las pequeñas partes de un programa tienen que ser paralelizadas para lograr todo el rendimiento posible.

Existe un determinado número de grupos desarrollando y mejorando sus herramientas de paralelización automática, creando esqueletos algorítmicos, que pueden facilitar el proceso de programación paralela. Se han escogidos tres métodos puntuales que mencionaremos a continuación.

3.1.- Método de Anderson y Lam [3, 4, 5, 18, 19, 34].

El algoritmo propuesto organiza los datos y los distribuye a través de módulos de memoria, de manera tal que cada procesador vaya realizando las operaciones obteniendo los datos que requiere directamente de la memoria. Al repartir el cómputo entre los procesadores se determina la descomposición de cómputo, mientras que la colocación de los datos en las memorias locales de los procesadores determina la descomposición de datos. El algoritmo propuesto determina automáticamente las descomposiciones de cómputo y de datos optimizando el paralelismo y la localidad del programa paralelo. Esté algoritmo está dirigido a máquinas distribuidas y compartidas. Estas descomposiciones se realizan en dos pasos: Primero encuentra todos los posibles lazos paralelos del código. Segundo, escoge entre los lazos paralelos aquellos que minimizan las comunicaciones [3, 34].

Si la comunicación se necesitara el sistema introduce las rutinas de comunicación necesarias en las partes del programa que se ejecutan menos frecuentemente, de esta manera minimizaría la comunicación y aumentaría la localidad.

Está aproximación en dos pasos hace el algoritmo más simple, ya que se centra en el problema de descomponer cómputo y datos automáticamente, enfocándose en la minimización de las comunicaciones en la secuencia de lazos que forman el programa.

Para derivar las descomposiciones de datos automáticamente, en una matriz se representan los límites de los lazos y los subíndices de las funciones de acceso, el algoritmo va encontrando las descomposiciones para los lazos en los cuales el número de iteraciones es mucho más grande que el número de procesadores. La meta de este algoritmo es

encontrar el orden o la forma de las descomposiciones [19]. El modelo se basa en que las descomposiciones equivalentes tienen los mismos datos y cómputo asignados a un solo procesador.

Los costos de comunicación se determinan cuando hay movimientos de datos entre procesadores vecinos o movimientos generales de la estructura de datos. La comunicación que existe dentro de un lazo se llama, *comunicación pipelined*; mientras que una comunicación debida a que se necesite distintas descomposiciones lo que requiere una transferencia de estructura de datos entera se llamará, *redistribución de datos*; de este modo si existiera una sola descomposición de datos y si no hubiera redistribuciones, esto se considerará una *descomposición estática* ya que se encuentra una sola descomposición estática para cada lazo anidado, considerando sólo el paralelismo disponible en cada lazo; por el contrario, la *descomposición dinámica* es aquella en la que se permite reorganizar los datos e introducir comunicaciones.

Para encontrar las descomposiciones de datos y de cómputo se analiza tres componentes distintos [3, 5]:

- La partición: determina el cómputo y los datos que se van asignando al mismo procesador. Determina la dependencia de los datos con cada iteración del lazo. La comunicación no será necesaria cuando todos los elementos de una iteración se vayan asignando al mismo procesador.
- La orientación: especifica el procesador en el cual se asignan los datos y cómputos. Existen muchas orientaciones de comunicación–libre diferentes, pero siempre con la misma partición.
- Desplazamiento: especifica los desplazamientos de los elementos y de las iteraciones con respecto a los procesadores.

Existen diferentes descomposiciones equivalentes a la misma partición reduciendo así la complejidad de encontrar descomposición para cada lazo. Un cálculo simple puede usarse para encontrar las orientaciones apropiadas y desplazamientos que especifiquen las descomposiciones.

Cuando no se encuentra una descomposición estática de las iteraciones y datos sin comunicaciones, el problema se restringe a los lazos más frecuentemente ejecutados. La partición de datos estática va segmentando el programa, pero no se tiene en cuenta el patrón

de acceso de cada array ya que este patrón puede variar en diferentes puntos del programa, es decir, no tiene por qué variar para todos los arrays ni en todos los anidamientos.

Para modelar descomposiciones dinámicas el método expuesto utilizan un gráfico de comunicación. Donde los nodos del gráfico corresponden a los lazos en el programa, los enlaces del gráfico representan los lugares en el programa en donde la comunicación reorganiza los datos. Para cada nodo se utiliza la descomposición de cómputo y la carga del nodo para estimar el tiempo de ejecución (teniendo en cuenta el paralelismo en la jerarquía del lazo). Al reorganizar los datos la descomposición en un lazo puede diferenciarse de otro lazo. Así las descomposiciones de datos, junto con los enlaces del grafico se utilizan para poder estimar el tiempo de la comunicación. En fin, el valor del gráfico es la suma de las ventajas del paralelismo de todos los nodos del lazo menos el costo total de la comunicación.

En general, el algoritmo obtiene una descomposición de datos, en cada jerarquía de lazo, y una descomposición de cómputo para cada jerarquía de lazo.

Finalmente, Anderson y Lam proponen un método de descomposición de datos y de cómputo, que logra resolver el problema de una manera sistemática. Aumentan el paralelismo aprovechando los "forall" (indica que las iteraciones del lazo se pueden ejecutar en forma paralela), así como los "doacross" (o doacrosses: directiva fundamental, que permite la paralelización de un bucle en distintos procesos) usando bloques para dicho paralelismo. Al minimizar la comunicación, el algoritmo intenta encontrar una descomposición estática que explote el grado máximo de paralelismo disponible en el programa [3, 18].

3.2.- Método de García, Ayguadé y Labarta [7, 8, 9].

En este trabajo, se presenta una estrategia para derivar automáticamente las descomposiciones estáticas o dinámicas de distribución de datos. Es decir, el modelo diseña una herramienta de distribución de datos capaz de derivar automáticamente los datos. La distribuciones básicas que se realizan son bloque o cíclica unidimensional. Toda la información que se requiere sobre los movimientos y el paralelismo de datos está contenida en una estructura de datos, llamada el grafo de Comunicación-Paralelismo (CPG) [7].

Este grafo es una estructura que representa información simbólica sobre el movimiento de datos y el paralelismo. Su objetivo principal es el de modelar y solucionar un

problema de optimización, el tiempo de ejecución del programa paralelo, la función objetivo a minimizar. El problema de optimización se soluciona usando un resolutor de programación linear genérico. Los autores demuestran la viabilidad de usar esta aproximación para resolver el problema de la distribución automática de datos tanto en términos del tiempo de compilación como en la calidad de las soluciones generadas.

Las herramientas automáticas de la distribución de datos pueden asistir al programador en encontrar las estrategias de la distribución y de paralelización de datos obteniendo un buen funcionamiento para un sistema distribuido. Esta herramienta automática de distribución de datos genera distribuciones de dos dimensiones, y asume que la topología de los procesadores es fija, es decir, no se realizará ningún cambio del número de procedimientos durante el tiempo de ejecución. La distribución de datos generada puede ser dinámica.

El modelo utiliza para la distribución de datos en dos pasos principales: alineación y distribución. El paso de la alineación encuentra alineaciones relativas apropiadas entre todos los arrays en un bloque del código [8]:

a. Decide las dimensiones que se alinearán con la plantilla (alineación inter-dimensional)
b. Para cada dimensión alineada, se establece el desplazamiento entre cada dimensión y las dimensiones de la plantilla (alineación intra-dimensional).

Una vez que se haya realizado la alineación, el paso de la distribución decide qué dimensión o dimensiones de la plantilla se distribuyen, el número de los procesadores asignados a cada uno de ellas, y la manera de la distribución, es decir: BLOQUE o CÍCLICO. La alineación intenta reducir al mínimo la comunicación inter-procesador y la distribución intenta maximizar el paralelismo potencial del código, balancea la carga de cómputo. Estos dos pasos no son independientes y se deben considerar de una manera unificada al derivar buenas soluciones [8].

En este método al derivar automáticamente un BLOQUE unidimensional estático o dinámico o una distribución CÍCLICA, la solución a los problemas de distribución y paralelismo se da en un solo paso, y se evita el uso de heurístico. El CPG contiene la información sobre los movimientos de los datos y el paralelismo inherente en un código.

Los programas se descomponen inicialmente en bloques de código llamado fases. Es importante saber que una fase es un lazo. Cada fase tendrá su propia distribución de datos, y

las acciones de redistribución se pueden realizar entre las fases. El CPG se utiliza para formular un problema de minimización, en el cual el tiempo es la función objetivo a minimizar. Como se mencionó anteriormente la solución se obtiene con un resolutor linear genérico.

Este resolutor linear genérico encuentra la solución óptima en una pequeña cantidad de tiempo, el problema del movimiento de datos y la búsqueda del paralelismo inherente en un bloque son los dos problemas que el grafo permite resolver en un solo paso. Esto permite minimizar la comunicación mientras va aumentando al máximo el paralelismo.

En la figura 10 se ilustra los componentes del CPG [7], es un ejemplo de la posibilidad de una alineación detectada en el análisis de los patrones de referencia entre los pares de arrays dentro de una fase P_i. Los enlaces se conectan a un nodo en el que el array es leído (aparece en el lado derecho de una declaración de asignación). El peso que se asigna al borde es una expresión simbólica que refleja el movimiento de datos que cuesta si se alinean y se distribuyen estas dos dimensiones.

```
                    do i = 2, N
          do j = i, N
                    A (i,  j)  = B  (i-1, j) + 1
                    B (i,  j)  = A (i, j)  + 2
                    C (j,  i)  = B  (i, j) + 3
          enddo
enddo
do i = 1, N
          do j = 1, i
                    C (i,  j) = D (i,  j) + 1
                    E (j,  i) = D (i,  j) + 2
          enddo
enddo
```

Figura 10. Componentes del CPG.

La información de la redistribución se incluye en los enlaces en función del movimiento de datos. Entre las acciones posibles de redistribución en una secuencia de fases, pueden extraerse dos casos [7]:

$$A(i, j) \leftarrow B(i-1, j)$$
$$B(i, j) \leftarrow A(i, j)$$
$$C(j, i) \leftarrow B(i, j)$$

Si dos dimensiones se alinean y se distribuyen en un movimiento de datos se involucran todos los procesadores mientras se realiza la ejecución del lazo. El CPG contiene información de los movimientos de datos, como se muestra en la figura 11.

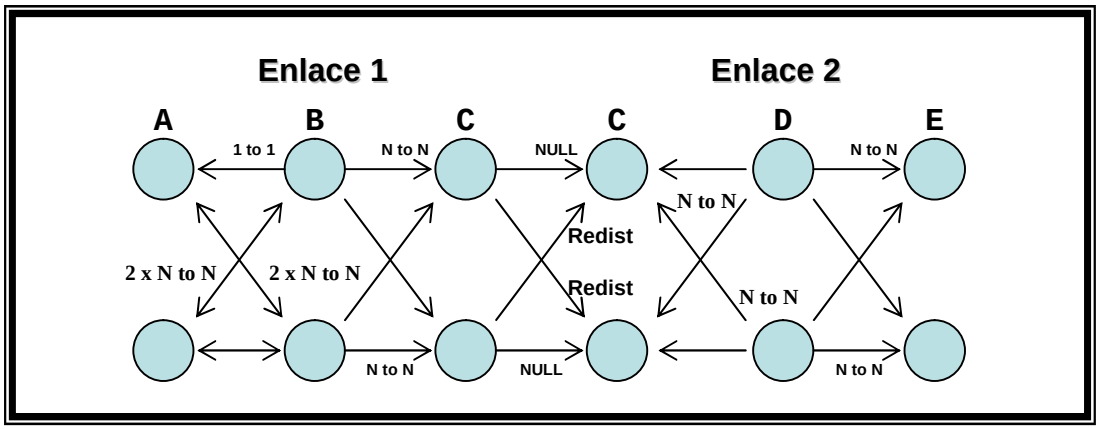

Figura. 11. Movimientos de datos en el CPG, con la información.

El CPG, incluye información de comunicación y de paralelismo en los enlaces (movimientos de los datos) y los hyperedges (paralelismo). Estos hyperedges anotan las estrategias posibles del paralelización para los lazos. Un hyperedge es una generalización de un enlace, pues puede conectar más de dos nodos. El hyperedge liga los nodos en el CPG que corresponden a las dimensiones de los datos que tienen que ser distribuidos para hacer paralelismo en el lazo. El peso del hyperedge representa el tiempo de ejecución que se ahorra cuando el lazo se hace paralelo (asumiendo un número dado de procesadores) [7]. Finalmente, si todos los nodos se conectan por un hyperedge se alinean y se distribuyen, entonces los lazos pueden paralelizarse; pero si ocurriera lo contrario que no se conectaran a un hyperedge no se alinearán ni distribuirán, por lo tanto los lazos no podrían ser paralelizados.

Al ejecutar el programa secuencial se estiman los costos de los hyperedges en el CPG, es decir, la máquina da la información especifica que se usa para estimar los costos; el analizador crea un fichero que contiene un problema de minimización. Este archivo es la entrada al resolutor. El problema de minimización entero lineal es un problema, en el que cada variable tiene dos posibles valores: 0 o 1. La solución se computa en un solo paso, e

incluye alineación, distribución, redistribución, y la estrategia de la paralelización de lazo correspondiente. Este evita el uso de heurísticas mientras se encuentra la solución óptima.

Resumiendo el CPG construye un grafo con información sobre los movimientos de los datos y el paralelismo inherentes en un programa, resolviendo la alineación y los problemas de distribución de una manera unificada y en un solo paso. La solución óptima permite minimizar las comunicaciones y maximizar el paralelismo.

3.3.- Método de Navarro y Zapata [16, 26].

Su estudio comienza con la problemática de la paralelización automática y cómo establecer un programa eficiente en multiprocesadores basados en una arquitectura tipo NUMA, ya que son muy sensibles al uso apropiado de su jerarquía de memoria. También es importante saber que cuando se paraleliza un programa secuencial sin tener en cuenta las características del sistema de memoria del multiprocesador, esos programas exhiben un pobre rendimiento cuando se ejecutan en dicho multiprocesador. Su solución está en explotar la localidad de las referencias a memoria. Por ende, su meta es explotar la localidad en la distribución de las iteraciones / datos que vayan facilitando el alojamiento de los datos en las memorias locales de los procesadores que los demanden. Es decir, ir minimizando el número de accesos a posiciones de memoria remota; como también conseguir un reparto equitativo del trabajo entre los procesadores (una selección apropiada a las distribuciones de iteraciones). O sea, elegir una distribución capaz de explotar localidad en un código mientras se va paralelizando a nivel de lazos.

Su objetivo principal es desarrollar un método de compilación automática capaz de encontrar las distribuciones de iteraciones y datos que exploten la localidad mientras se balancea la carga computacional. En otras palabras, el método es automático ya que el compilador sin intervención del programador extrae toda la información del código secuencial y genera la versión paralela que se ejecuta sobre un multiprocesador con arquitectura tipo NUMA [16].

El método de distribución de iteraciones y datos que proponen los autores consiste en tres pasos; en primer lugar establecen una nueva técnica de compilación para el análisis de localidad de referencias de memoria en un código secuencial; en segundo lugar proponen una nueva representación, en tiempo de compilación, en forma de grafo, que va capturando

la localidad exhibida por los arrays del programa, cuando se selecciona los lazos que serán paralelos; en tercer lugar la propuesta se engloba en un compilador que es capaz de generar código paralelo a partir del secuencial, de manera que se van identificando sucesivamente los accesos remotos y generando las primitivas de comunicación necesarias.

Las técnicas de compilación que desarrollan se basan en una notación precisa de la región de un array que se accede en un anidamiento, tanto para funciones de acceso afines generales, como para funciones de acceso no afines. Teniendo en cuenta que se necesita explotar paralelismo a nivel de lazo, la notación debe permitir describir la región de un array accedida al ejecutar un determinado anidamiento. Se considera que un programa es una secuencia de anidamientos (lazos que no tienen por qué estar perfectamente anidados). En estos anidamientos puede que alguno o algunos lazos sean paralelos. La notación debe ser lo suficientemente potente como para manejar funciones de acceso al array no afines, ya que éstas tienen lugar en numerosos códigos reales. Una vez que se han capturado la región de un array accedida en un anidamiento, se determina qué sub-región se accede para cada iteración del lazo paralelo. De esta forma, el explotar la localidad en un anidamiento, para un array, se traduce en colocar la región accedida en cada iteración i del lazo paralelo del anidamiento en la memoria local del procesador que ejecute esa iteración i.

El explotar localidad puede parecer relativamente sencillo, y en cierto modo lo sería si los programas reales sólo tuviesen un anidamiento accediendo a un único array. En este método se identifica si es posible o no encontrar una distribución de iteraciones y datos para un array de forma que entre la ejecución de un anidamiento y la ejecución del siguiente no sea necesario acceder a datos remotos. Cuando no se pueda explotar localidad en las referencias a un array en dos anidamientos que se ejecutan consecutivamente, el método propuesto es capaz de determinar los patrones de comunicación que satisfacen las referencias no locales.

Al final se captura la localidad y los patrones de comunicación de las referencias a cada array en todo el código, en un grafo que llaman GLC (Grafo de Localidad–Comunicación) que contiene un sub-grafo para cada array del programa. Cada sub-grafo tiene un nodo para cada anidamiento en el que el correspondiente array se referencia. Estos nodos se conectan con arcos dirigidos según el flujo de control del programa. Sobre este grafo y con la información de acceso a regiones ya disponible, se lleva a cabo el análisis de localidad que resultará en el etiquetado de los arcos que unen los nodos de cada grafo. Las

etiquetas indicarán si es posible o no encontrar una distribución de iteraciones / datos para los dos anidamientos conectados, de forma que entre la ejecución de un anidamiento y la ejecución del siguiente no sea necesario acceder a datos remotos.

Para entender mejor el grafo GLC y la forma cómo los investigadores demostraron la eficiencia del código paralelo y el impacto que las técnicas de distribución de datos consiguen, utilizaran tres de evaluación: TRFD del Perfect Club Benchmarks, TOMCATV del conjunto SPECfp95 y TFFT2 del NCSA (National Center for Supercomputing Applications). De estos tres se tomará el TFFT2, en este sumen este código se encarga de ejecutar varias FFTs (varios tipos de transformadas de Fourier).

En realidad es un código complejo porque se anidan gran cantidad de llamadas a subrutinas en lazos, y porque las funciones de acceso de las referencias a los principales arrays del programa, X e Y, son expresiones no lineales.

El código está organizado en un lazo externo secuencial que llama repetidamente a varias subrutinas y éstas a su vez a otras, dando lugar a una estructura compleja de lazos anidados con llamadas a subrutinas. Es un ejemplo de código real en el que el patrón de acceso a los datos de los principales arrays del programa cambia durante la ejecución del mismo [16].

```
1.    <  Inicialización de los elementos de X >
2.    F1: doall I = 0 P · Q – 1
3.          do J = 0 to 0
4.          Y (I + J) = (2 · I – 1 + J)
5.          Y (I + J + P) = X (2 · I + J)
6.          do J = 0 to Q – 1
7.          enddo
8.        enddo
9.    F2: doall I = 0 to p – 1
10.         do J = 0 to Q – 1
11.         X (I + J * 2 * P) = Y (J + I + Q)
12.         X (I + J * 2 * P + P) = Y (J + I * Q + P * Q)
13.         enddo
14.       enddo
15.   F3: doall  I = 0 to Q – 1
16.         do L = 1 to P
17.           do J = 0 to P · 2^L – 1
18.             do K = 0 to 2^(L-1) – 1
19.               Y(J · 2^L + K)
20.               Y(J · 2^L + K + P)
21.               Y(J · 2^L + 2^(L-1)+ K)
22.               Y(J · 2^L + 2^(L-1)+ K + P)
23.               X(I · 2 · P + J · 2^(L-1)+ K)
24.               X(I · 2 · P + J · 2^(L-1)+ K + P/2)
25.               X(I · 2 · P + J · 2^(L-1)+ K + P)
26.               X(I · 2 · P + J · 2^(L-1)+ K + + P + P/2)
27.             enddo
28.           enddo
29.         enddo
30.       enddo
```

Figura 12. Referencias a X e Y en las fases F_1, F_2, F_3.

En la figura 12 se muestra un fragmento del código TFFT2 con las fases F_1, F_2, F_3 ; Una fase no es más que un anidamiento constituido por lazos que definen los índices utilizados para calcular los subíndices de las funciones de acceso de los arrays referenciados en el anidamiento.

En la figura 13 se muestra una sección del código TFFT2 con ejecución lexicográfica o iterativa (flujo de control de ejecución de la fase) que incluye 8 fases de este código [16]. Este fragmento comprende las fases F_2 , F_3 y F_1 que se presentaba en la figura 12. El grafo GLC está compuesto por dos subgrafos; uno para el array X y otro para el array Y. Cada subgrafo se compone de 8 nodos (cada array es referenciado en cada una de las 8 fases), es decir, las fases en las que el correspondiente array es referenciado. Cada nodo está anotado con un atributo que identifica el tipo de acceso a memoria para el array en la correspondiente fase: **W, R, R/W** o **P**. Donde W representa accesos de escritura R, acceso de lectura X; R/W para accesos de lectura y escritura [26]. En el caso que el array sea privatizado en una fase, el correspondiente nodo es marcado con el atributo P. Los nodos están conectados con arcos dirigidos, que representan el flujo de control del programa.

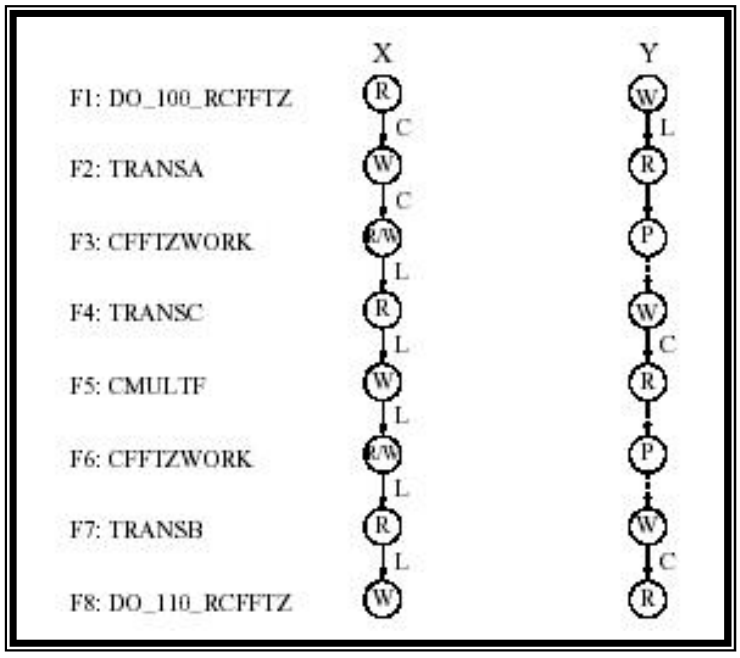

Figura. 13. GLC una sección del código TFFT2 con ejecución iterativa.

El GLC es la herramienta que los investigadores utilizan en el algoritmo de distribución de iteración de datos y generación de las comunicaciones. En otras palabras, la información de entrada que distribuirán procede del grafo GLC del programa, el grafo va capturando la información de localidad y los patrones de comunicación de los arrays, así como información del flujo de ejecución del código. En la figura 14, en el grafo GLC se comprueba que cada fase es asociada a una variable P_k, compartida por todos los nodos (k, j) que referencian a un array X_j en la fase F_k. Esta variable representa el tamaño del bloque de iteraciones del lazo paralelo de la distribución CYCLIC (P_k) de F_k. Las variables Pk de todas las fases del código son las variables del modelo propuesto de programación no lineal entera. La función objetivo se representará por el costo debido al overhead de la ejecución del código paralelo. Esta función objetivo está sujeta a una serie de restricciones lineales (de localidad, de balanceo de carga, de almacenamiento), derivadas del GLC. Estas restricciones cuentan cuándo se puede explotar localidad y las peculiaridades de las regiones de los arrays accedidas en las iteraciones paralelas para cada fase [26].

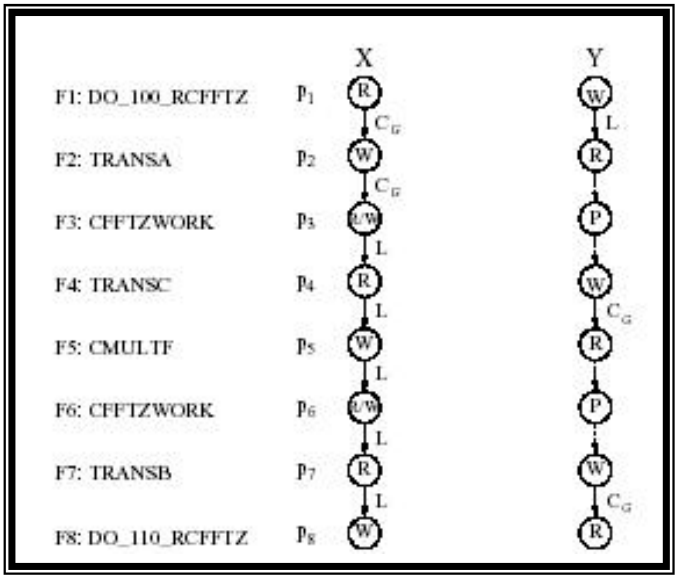

Figura 14. Variables del GLC del código TFFT2.

Una vez llegado a este punto sabemos que es posible encontrar una o varias distribuciones de iteración de datos que se adaptan a las peculiaridades del código y que evitan las comunicaciones entre dos anidamientos que se ejecutan consecutivamente.

Ahora bien, al tratar de encontrar la distribución (que saben que existe, pero no saben cuál es) que minimiza el overhead (tiempo que se pierde, en el algoritmo paralelo, en operaciones que no tienen lugar en la correspondiente versión secuencial) de la ejecución paralela, se logra mediante la formulación de un problema de optimización no lineal entero, en el que la función objetivo que se quiere minimizar es caracterizada como el overhead, y las restricciones se obtendrán principalmente del grafo GLC. La función objetivo modela el tiempo que pierde el código paralelo tanto debido a las comunicaciones como al desbalanceo de la carga computacional (que depende de la distribución). Por lo que esta función objetivo, se ajusta lo suficiente al overhead real para constituir un criterio suficiente que permite al algoritmo de optimización inclinarse por una distribución frente a otra.

Por otra parte, para generar código paralelo una vez encontrada las distribuciones es necesario generar las rutinas de comunicación entre las fases en las que estas comunicaciones son obligatorias. Los investigadores presentan los algoritmos que permiten implementar la distribución y generan las rutinas de comunicación necesarias entre aquellos anidamientos que lo necesiten.

Los investigadores demuestran que la distribución es uno de los aspectos más influyentes para la eficacia de los códigos paralelos generados por multiprocesadores con arquitectura NUMA. En conclusión, este es un diseño automático de distribución de iteración de datos con control explicito de las comunicaciones que además genera el código paralelo a partir del secuencial, insertando las rutinas de comunicación pertinentes.

3.4.- Discusión.

Los multiprocesadores con memoria físicamente distribuida están llegando a ser populares y los paralelizadores automáticos se han de encargar de paralelizar y de distribuir los datos y las iteraciones para conseguir códigos que se ejecuten efectivamente en el sistema. La latencia de acceso local frente a la latencia de acceso a los datos remotos puede variar de uno o varios órdenes de magnitud, lo que puede afectar dramáticamente al rendimiento del sistema. Se desea liberar al programador de consideraciones de bajo nivel en las arquitecturas, es decir, de tener que programar los procesos y especificar la comunicación entre esos procesos. Para evaluar los tres métodos de distribución automática de datos que

resumimos en las secciones 3.1, 3.2 y 3.3, nos concentraremos en tres aspectos 1) aportaciones; 2) limitaciones; 3) estructura.

Usualmente, en el campo de la paralelización y distribución automática de datos se han seguido dos enfoques: el análisis en tiempo de compilación o análisis estático y el análisis en tiempo de ejecución o análisis dinámico.

J. Anderson y M. Lam [3, 4, 5, 18, 19, 34] nos indican que al explotar la localidad en las referencias a memoria en arquitecturas tipo NUMA, se han de organizar los cómputos y los datos a través de distintos módulos de memoria de tal manera que el procesador realice las operaciones y consumo de los datos que requiera directamente de la memoria local. El programa de entrada consiste en lazos generalmente anidados con o sin paralelismo explícito: cada lazo puede ser de tipo forall, doacross o lazo secuencial. El límite de los lazos y los subíndices de cada referencia a memoria son por funciones afines. Su principal aportación esta basada en encontrar datos y cómputos que se vayan mapeando en un procesador virtual. El problema de la descomposición es muy complejo, por lo que se trata de calcular las descomposiciones de datos y de cómputo para los programas de una manera sistemática. Estructuran las descomposiciones en tres componentes: partición, orientación y desplazamiento. Puesto que las descomposiciones equivalentes tienen la misma partición, solucionan la partición primero y pueden por lo tanto evaluar varios diseños posibles de descomposición simultáneamente.

Si no se encuentra una descomposición estática de iteraciones / datos sin comunicaciones; se plantea un problema de redistribución imponiendo restricciones desde los lazos más frecuentemente ejecutados. Se busca verificar todas las restricciones introduciendo comunicaciones. Su algoritmo, encuentra una partición de datos estática para el segmento del programa más ejecutado, pero no tiene en cuenta el patrón de acceso a cada array que puede variar en diferentes puntos del programa, además, no tienen por qué variar todos los arrays en el punto de redistribución. Una de sus desventajas principales es que el balanceo de la carga no está resuelto, además que una vez que el algoritmo propuesto encuentra la descomposición de datos éste no identifica los patrones de comunicación y sólo trabaja con funciones de acceso afines simples.

Por su parte, García, Ayguadé y Labarta [7,8, 9] establecen un grafo de comunicación – paralelismo (CPG) que contiene la información sobre los movimientos de datos y

paralelismo de un bloque de código. Esta información se usa para formular el problema del camino mínimo con un juego de restricciones que aseguran el alineamiento de las distintas dimensiones de los arrays. Tomando la programación lineal entera 0-1 para resolver y garantizar la solución del problema del camino mínimo, se resuelve el problema de distribución de iteraciones y datos.

Estructuralmente descomponen el problema de la distribución de iteración y datos en dos pasos independientes: 1) la alineación: aquí se alinean todos los arrays, y se decide la dimensión de cada array que se alinean (alineación Inter–dimensional), y para cada dimensión alineada si es necesario un desplazamiento. Para ello se define una plantilla y con respecto a ella se alinean todos los arrays. Dos arrays no alineados dan lugar a comunicaciones, por lo tanto se intenta minimizar los accesos no locales al procesador (se minimiza las comunicaciones que generan acceso no local); 2) La distribución: acá se decide cómo se van mapeando las dimensiones de la plantilla y el número de procesadores que se asignan en cada dimensión. Se intenta explotar al máximo el paralelismo. Se intenta explotar al máximo el paralelismo. Se aplica la regla de computa el propietario para calcular la partición de los lazos.

La solución óptima al problema de distribución de datos, toma en cuenta la arquitectura del procesador. La distribución podrá ser estática (no cambia durante la ejecución del programa) o dinámico teniendo en cuenta las distribuciones entre fases.

Una de su desventaja principal es que la búsqueda de la distribución de datos para maximizar el paralelismo y minimizar los accesos no locales no pueden realizarse en dos etapas independientes, puesto que ambos factores interactúan entre sí. Es decir, no es suficiente con encontrar una solución óptima para la alineación ya que encontrar alineamiento no tiene por qué resultar en un posterior mapeo óptimo para todos los arrays (o sea el problema de la distribución). Además, el programa puede no ser portable, puesto que la distribución seleccionada para una determinada arquitectura pueden no ser la mejor en otro tipo de arquitectura. Lo que requiere volver a anotar el código con distintas directivas, en función del sistema multiprocesador en el que vaya a ejecutar. En cuanto al problema de minimizar el coste se ha de hacer mientras que la información de coste se computa con los movimientos de datos que suponen que dos dimensiones de 2 arrays no estén alineadas. Los patrones de comunicación son demasiado simples (trabajan con funciones de acceso afines simple) y se supone que son independientes de las distribuciones

de datos; y realmente, hasta que no se ha elegido la distribución de datos no se conocerá de forma precisa, las necesidades de comunicación de cada lazo, para cada array. Es importante también destacar que no le dan tampoco consideración al problema del balanceo de la carga, que puede ser decisivo a la hora de elegir una u otra distribución.

Con respecto al método expuesto por Navarro y Zapata [16, 26] al explotar la localidad de las referencias a memoria van seleccionando las distribuciones de iteraciones de datos facilitando el alojamiento de los datos en las memorias locales de los procesadores que lo demanden, o sea, se va minimizando el número de acceso a posiciones de memoria remota. Además se van repartiendo equitativamente el trabajo entre los procesadores para así obtener una selección apropiada de la distribución de iteraciones. Los investigadores se centraron en responder dos cuestiones: i) cómo explotar la localidad en un código secuencial que se paraleliza para una determinada arquitectura NUMA y; ii) cómo balancear la carga computacional de manera efectiva. El compilador planifica las iteraciones que se asignan a cada procesador en función de las distribuciones de iteraciones, y a continuación aloja y distribuye los datos a través de las memorias locales de los procesadores que las demande, e identifica cuándo alguna de las referencias es a una posición de la memoria remota, generando las rutinas de comunicación que sean necesarias. Por lo que se explota el paralelismo a nivel de lazo en multiprocesadores de memoria distribuida y en multiprocesadores de memoria compartida – distribuida.

En este método los arrays que Polaris y PFA llaman shared arrays, están distribuidos entre los distintos módulos de memoria local. Es decir cada procesador aloja y opera sobre su propia porción local de datos. Con la técnica de Navarro y Zapata [16], la afinidad entre iteraciones y datos no se basa en la regla computa el propietario, regla que puede provocar un desbalanceo de la carga en ciertas situaciones. Simplemente cada procesador, para cada lazo paralelo, computa aquellas iteraciones paralelas que se le han asignado según la distribución de iteraciones seleccionadas. Sus principales aportaciones consisten en: i) se transforman los lazos paralelos, de manera que el espacio de iteraciones de cada procesador sea el conjunto de índices locales que según la distribución de iteración seleccionada por el método de Navarro y Zapata [16] corresponda a dicho procesador; ii) traducen las referencias de memoria, desde el espacio de direcciones globales a referencias de memorias locales, de acuerdo con la distribución de datos computada para cada array; y iii) para

aquellas referencias a pociones de memoria no local las resuelven insertando explícitamente rutinas de comunicación put/get (estas primitivas simplifican el algoritmo de compilación que se encarga de generar rutinas de comunicación, así como la optimización de las comunicaciones).

Los investigadores Navarro y Zapata [16] obtuvieron doble eficiencia al aplicar manualmente las distribuciones de iteraciones / datos que obtenían con ese método, comparado con la eficiencia que se obtenía con otros métodos. Estos autores exponen tres razones por la que los métodos de Anderson, Lam, y García, Ayguadé y Labarta no son capaces de proporcionar distribuciones más eficientes: 1) No capturan de manera precisa la región de un array que se accede en un anidamiento especial cuando las funciones de acceso de las referencias son funciones afines no simples, o son funciones no afines de los subíndices; 2) No pueden determinar que subregión de cada array se accede para cada iteración del lazo paralelo de un anidamiento; 3) Las distribuciones de iteración y datos en las que se basan sus otros métodos son demasiado simples para adaptarse a las peculiaridades (simetrías entre regiones de un array, solapamentos entre regiones del array) de los códigos reales y dan lugar a una gran cantidad de comunicaciones.

El método de Anderson, y Lam [3], fue pionero diseñando un algoritmo que distribuye automáticamente los datos aunque se basa en heurísticos. García, Ayguadé y Labarta [7] les dieron otra aproximación al problema al construir un grafo (CPG) que contiene la información sobre los movimientos de los datos y el paralelismo inherentes en un bloque del código, en el se puede resolver la alineación y los problemas de la distribución de una manera unificada (en un solo paso), permitiendo minimizar la comunicación mientras aumenta el paralelismo. Su solución óptima utiliza una herramienta de propósito general de programación lineal entera, así evita el uso de la heurística para resolver el problema de distribución de datos. Navarro y Zapata [16] lograron diseñar un método automático de distribución de iteraciones / datos con un control explícito de las comunicaciones que además genera el código paralelo a partir del secuencial, insertando rutinas de comunicación pertinentes. Tres aportaciones de este método se pueden destacar: 1) desarrollan una notación precisa y un modelo algebraico para describir las regiones (descriptor de acceso, representación abstracta de la región de un array accedida en una fase por una referencia a ese array; descriptor de fase, representa toda la región de array accedida en esa fase; descriptor de iteración, representa la sub-región del array accedida en cada iteración del lazo

paralelo además contiene información adicional para representar el tipo de acceso al array y el flujo de control de ejecución de la fase; simetrías de almacenamiento, permite simplificar el descriptor de iteraciones al mismo tiempo subrayar el tipo de región a la que accede en cada iteración del lazo paralelo); 2) un análisis de localidad (partiendo de los descriptores de iteración para los arrays y las fases lo organizaron en un grafo de localidad–comunicación (GLC) que se compone de un subgrafo para cada array y cada subgrafo contiene un nodo por cada fase que accede a ese array, proponen los conceptos de localidad intra-fase y localidad inter - fase); 3) distribución automática de iteraciones / datos mediante el planteamiento de un problema de optimización (que solucionan utilizando programación no lineal entera. El problema de optimización modela el overhead debido al coste de las comunicaciones y la penalización al desbalancear la carga dependiendo de la distribución de iteraciones seleccionadas. El problema está sujeto a un conjunto de restricciones derivadas del grafo GLC. El problema de la minimización no lineal entero se soluciona gracias a la herramienta GAMS [16].

Por tanto, los trabajos mencionados tratan de desarrollar un módulo de compilación que de manera automática, sin intervención del usuario, a partir de un programa escrito en el conocido paradigma secuencial, transforme el código para que se pueda ejecutar en un multiprocesador, teniendo en cuenta que se ha de explotar la localidad. Los paralelizadores automáticos actuales han alcanzado cierto grado de madurez y se han convertido en la herramienta más cómoda para generar una versión paralela a partir del código secuencial. Pero se debe tener en cuenta que un programa paralelizado pierde prestaciones debido a que estos paralelizadores no explotan la localidad. Por eso es necesario incorporarse un nuevo módulo de compilación que sea capaz de generar las distribuciones de iteraciones y datos óptimos para un código. Al paralelizar los códigos manualmente aplicando la distribución de datos apropiada a cada caso, se puede reducir significativamente los tiempos de ejecución de los códigos paralelizados automáticamente para arquitecturas NUMA. No obstante, la paralelización manual implica invertir una gran cantidad de tiempo en la paralelización. De ahí que la paralelización automática sea una opción interesante para conseguir paralelizar eficientemente los códigos, con la mínima intervención del usuario.

Conclusiones y Trabajos Futuros

Este trabajo presenta varias técnicas que en tiempo de compilación procuran obtener un programa paralelo a partir del código secuencial, que se efectué eficientemente en multiprocesadores de arquitectura distribuida – compartida tipo NUMA. El interés por este tipo de arquitecturas ha crecido a medida de los años, ya que lo que se quiere con estos sistemas es integrar la escalabilidad de las arquitecturas de memoria distribuida con la facilidad de la programación en el paradigma de la memoria compartida.

La importancia de explotar la localidad de las referencias a memoria en arquitecturas de memoria compartida físicamente distribuida ha sido demostrada experimentalmente [3]. E implica Significa que al ir organizando los cómputos y distribuyendo los datos en los distintos módulos de memoria el procesador realizará las operaciones en menor tiempo, si consume los datos que requiere directamente de la memoria local.

En la compilación paralela, el conjunto de técnicas que se encargan de distribuir las computaciones y los datos a través de los distintos procesadores y los correspondientes módulos de memoria, reciben el nombre de distribución automática de iteraciones/datos. Realizar las transformaciones en un programa para que se implementen distintas estrategias de distribución de iteraciones/datos, requiere un importante esfuerzo por parte del programador [16].

Por tanto, los trabajos mencionados tratan de desarrollar un módulo de compilación que de manera automática, sin intervención del usuario, a partir de un programa escrito en el conocido paradigma secuencial, transforme el código para que se pueda ejecutar en un multiprocesador, teniendo en cuenta que se ha de explotar la localidad. Los paralelizadores automáticos actuales han alcanzado cierto grado de madurez y se han convertido en la herramienta más cómoda para generar una versión paralela a partir del código secuencial.

En está investigación hemos aclarado que los métodos son un gran aporte, pero hemos comprobado que falta mucho por hacer. Por lo que se debe tener en cuenta que un programa paralelizado pierde prestaciones debido a que estos paralelizadores no explotan la localidad. Esto hace necesario incorporar un nuevo módulo de compilación que sea capaz de generar distribuciones de iteraciones y datos óptimos para un código.

En este trabajo se ha evaluado tres métodos de distribución automática de iteración / datos que intentan explotar la máxima localidad. De los tres métodos, el propuesto por Navarro y Zapata es el que genera códigos paralelos más eficientes. Por lo tanto nuestro objetivo es desarrollar un entorno de compilación que implemente este algoritmo. Para ello el partirá de una plataforma de compilación automática llamada Polares [39]. Esta plataforma es una herramienta académica de uso público que nos permitirá experimentar e implementar las estrategias de compilación propuestas por Navarro y Zapata.

Bibliografía

[1] A. Z. Aguirezabala. Multiprocesadores. Material Mimeografiado.

[2] R. Allen and K. Kennedy Optimizing Compilers for Modern Architectures: A Dependence – based Approach. IEEE. 2001

[3] J. M. Anderson and M. S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation, June 1993. (http://suif.stanford.edu/papers/anderson93/paper.html#_blank)

[4] J.M. Anderson, S.P. Amarasinghe, and M. S. Lam. Data and Computation transformation for multiprocessors. In Principles and Practice of Parallel Programing, ACM SIGPLAN, June 1995.

[5] J. M. Anderson. Automatic Computation and Data Decomposition for Multiprocessor. PhD thesis, Stanford University, March 1997.

[6] R. Asenjo, G. Bandera, G. P. Trabado, O. Plata, and E. L. Zapata. Iterative and direct sparse solvers on parallel computers. In Euroconference: Super computations in Nonlinear and Disordered Systems: Algorithms, Applications and Architectures, San Lorenzo de El Escorial, Madrid, Spain, Sep. 1996.

[7] E. Ayguadé, J. García, and J. Labarta. A novel approach towards automatic data distribu tion. In Proceeding of Supercomputing`95, San Diego, CA, December 1995. (http://www.ac.upc.es/~jordig), (http://www.ac.upc.es/~eduard)

[8] E. Ayguadé, J. García, and J. Labarta. Dynamic data distribution with control flow análisis. In Proceeding of Supercomputing´96, Pittsburgh, PA, November 1996.

[9] E. Ayguadé, J. García, and U. Kremer. Tools and techiques for automatic data layout: A caase study. Journal of Parallel Computing, May 1998.

[10] W. Blume, R. Eigenmann, J. Hoffinger, D. Padua, P. Petersen, L. Rauchwerger, and P. Tu. Automatic detection of parallelism: A grand challenge for high-performance computing. IEEE Parallel and Distributed Technology, 1994.

[11] W. Blume, R. Doallo, R. Eigenmanm, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, W. Pottenger, L. Rauchwerger, P. Tu. Parallel programming with Polaris, IEEE Computer, 1996.

[12] Chandra R., Dagun L., Kohr D., Mandan D. McDonald J. & Menom R. Parallel Programming in OpenMP, Morgan Kaufman Publishers 2001

[13] A.L. Cheung and A.P. Reeves. Sparse data representation. In Proceedings Scalable High Performance Computing Conference, 1992.

[14] R. Eigenman, J. HoeAeinger, and D. Padua. On the automatic parallelization of the perfect benchmark. IEEE Transactions on Parallel and Distributed Systems, January 1998.

[15] K. Faigin. The Polaris internal representation. Master's thesis, Center for Supercomputing Res. & Dev., University of Illinois at Urbana-Champaign, 1994.

[16] M. A. González Navarro. Distribución Automática de Datos en Multiprocesadores. PhD Thesis, Universidad de Málaga, 2000.

[17] A. Gueist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 User`s Guide and Reference Manual. Physics and Mathematics Division, May 1993.

[18] M. Hall, S. Amarainghe, B. Murphy, S. W. Liao, E. Bugnion, and M. S. Lam. Detecting coarsegrain parallelism using an interprocedural parallelizing compiler. In IEEE Supercomputing`95, San Diego, CA, December 1995.

[19] M. Hall, J. Anderson, S. Amarainghe, B. Murphy, S. W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. IEEE Computer, December 1996.

[20] J. Hennessy and D. Patterson. Computer Organization & Design. Morgan Kaufmann Publishers, San Francisco, 1998.

[21] High Performance Fortran Forum. High performance Fortran Language Specification, Version 2.0, 1996. (http://research.ac.upc.es/CAP/hpc/HPC-Publications2003.html)

[22] K. Hwang. Advanced Computer Architecture: Parallelism, Scalability, Programmability. McGraw-Hill, 1993.

[23] Juan, L. Arquitecturas Paralelas. Material Mimeografiado. 2001

[24] Message Passing Interface Forum. MPI: A message-passing interface standard, version 1.1, 1995. (mpiexmpl.tar.gz)

[25] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, November 1996.

[26] M.A. Navarro, R. Asenjo, E. Zapata, and D. Padua. Access descriptor based locality analysis for distributed-shared memory multiprocessors. In International Conference on Parallel Processing (ICPP'99), Aizu-Wakamatzu, Japan, September 1999.

[27] OpenMP Architecture Review Board. OpenMP: A Proposed industry Standard API for Shared Memory Programming. http://www.openmp.org, 1997.

[28] OpenMP C and C++ Application Program Interface DRAFT. Version 2.0. November, 2001 DRAFT 11.05.

[29] OPENMP Fortran Application Program Interface. Version 1.1, Material Mimeografiado. 1999.

[30] ORNL. PVM 3 user's guide and reference manual. Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, 1994.(

[31] PACS Training Group. Introduction to MPI. University of Illinois. 2001

[32] Pérez, M. A. Paralelismo y Distribución. Networking center, Edición Octubre, 2001

[33] J. Protic, M. Tomasevie and V. Milutinovic. Distributes Shared Memory Concepts and Systems, IEEE Computer Society, 1997.

[34] Roberto P. Wilson, Roberto S. French, Christopher S. Wilson, Saman P. Amarasinghe, M. Anderson De Jennifer, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Maria W. Pasillo, Monica S. Lam, y Juan L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. Laboratorio De los Sistemas informáticos Universidad De Stanford, Ca 94305-4055. 1994.

[35] A. Saulsbury y T. Wilkinson y J. B. Carretero y A. Landin, An Argument for Simple COMA, IEEE Symp. en arquitectura de la computadora de alto rendimiento, 1995.

[36] Silberschatz, A. Galvin, P. Gagne, G. Applied Operating System Concepts. John Wiley and Sons. New York, 2000.

[37] Visual KAP for OpenMP. http://www.kai.com/vkomp/_index.html (Oct. 28, 2001).

[38] M. Wolf and M. A Data Locality Optimizing Algorithm. In Proceedings of ACM SIGPLAN 91 Conference Programming Language Design and Implementation, Toronto, 1991

[39] David A. Padua, Josep Torrellas, and Rudolf Eigenmann. Automatic Parallelization of Conventional Fortran Programs. http://polaris.cs.uiuc.edu/polaris/polaris.html, http://polaris.cs.uiuc.edu/ , http://polaris.cs.uiuc.edu/newhome/.